Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ « Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ
Лабораторная работа №6
«Обработка деревьев»

Группа ИУ7-34Б

Дисциплина Типы и структуры данных

Вариант 14

Студент Козлитин Максим Александрович

Преподаватель Силантьева Александра Васильевна

2022г.

Цель работы

Получить навыки применения двоичных деревьев, реализовать основные операции над деревьями: обход деревьев, включение, исключение и поиск узлов.

Условие задачи

1. Построить двоичное дерево поиска из букв вводимой строки.
2. Вывести его на экран в виде дерева.
3. Реализовать основные операции работы с деревом:
   3.1. обход дерева;
   3.2. включение;
   3.3. исключение;
   3.4. поиск узлов.
4. Выделить цветом все буквы, встречающиеся более одного раза.
5. Удалить из дерева эти буквы.
6. Вывести оставшиеся элементы дерева при постфиксном его обходе.
7. Сравнить время удаления повторяющихся букв из дерева и из строки.
8. Сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты деревьев и степени их ветвления.

Описание исходных данных

- Выбор действия:
  Целое число от 0 до 10.
- Строка:
  Набор букв максимальным размером 1000 символов.
- Буква:
  Один символ, который является буквой.

Способ обращения к программе

Работа с программой осуществляется с помощью консоли.

Программа запускается с помощью команды: **./app.exe**

Далее пользователь выполняет взаимодействие с помощью меню.
Меню выводится в начале и, далее, каждые 3 введенные команды.

Описания возможных аварийных ситуаций и ошибок пользователя

1. Ввод не совпадающий с форматом входных данных (описано в Описание исходных данных).
2. Ввод целых чисел превышающих максимальное допустимое значение типа данных int.

Описание внутренних структур данных

Бинарное дерево, узел дерева:

```
typedef struct node_t node_t;


struct node_t
{
    char data; // элемент вершины
    size_t cnt; // количество элементов в вершине
    size_t weight; // вес поддерева с корнем в данной вершине
    node_t *left; // левый сын
    node_t *right; // правый сын
};


typedef struct
{
    node_t *root; // корень дерева
} binary_tree_t; // бинарное дерево
```

size_t *используется для задания размеров наших массивов, так как является особым макросом, значение которого позволяет адресовать массив любой теоретически возможной длины для данной машины.*

char *- символьный тип данных.*

Массив узлов:

```
typedef struct
```

```
{
    size_t size; // размер массива
    node_t *data; // содержимое массива
} nodes_t; // массив узлов
```

Массив узлов *используется для конвертации дерева в массив, позволяя сохранить содержимое узлов в него.*

size_t *используется для задания размеров наших массивов, так как является особым макросом, значение которого позволяет адресовать массив любой теоретически возможной длины для данной машины.*

Описание алгоритма

Поиск элемента - **`btree_find`**:

1. Если текущая вершина равна нулю или содержит искомый элемент - возвращаем эту вершину.
2. Если искомый элемент больше содержимого текущий вершины - рекурсивно переходим в правого сына, иначе - в левого.

Удаление неуникальных - **`btree_del_ununique`**:

1. Если текущая вершина равна нулю - возвращаем эту вершину.
2. Рекурсивно переходим в правого сына.
3. Рекурсивно переходим в левого сына.
4. Если данная вершина содержит больше одного элемента - удаляем вершину.

Напечатать постфиксный обход - **`btree_print_postfix`**:

1. Если текущая вершина равна нулю - возвращаем эту вершину.
2. Рекурсивно переходим в левого сына.
3. Рекурсивно переходим в правого сына.
4. Печатаем текущую вершину.

Данные функции возможно запускать не только от корня нашего дерева, но и от корня любого поддерева внутри него.

Оценка эффективности

Алгоритм измерения времени:

1. Измеряется текущее время - начало.
2. Запускается выполнение функции.

3. Измеряется текущее время - конец.
4. Вычисляется количество наносекунд прошедших от начала и до конца.
5. Полученное значение добавляется в сумму.
6. Данное измерение производится 100 раз, вычисляя итоговую сумму всех запусков.
7. Ответ - среднее арифметическое.

Удаление неуникальных:

Размер	Строка, нс	Дерево, нс	Разница
1	78	72	8 %
100	297	4307	93 %
200	578	3264	82 %
300	905	2407	62 %
400	737	2156	66 %
500	900	2158	58 %
600	916	2356	61 %
700	1055	1853	43 %
800	1199	1865	36 %
900	1430	1861	23 %
1000	1484	1873	21 %
50000	52410	1730	97 %

***Процент** вычисляется так: (МАКС - МИН)/МАКС, где*
МАКС - максимальное из двух чисел
МИН - минимальное из двух чисел

Зеленая ячейка - более эффективный алгоритм
Красная ячейка - менее эффективный алгоритм

Для размера один наблюдается небольшое преимущество в связи с тем что операция добавления одного элемента в дерево малозатратна, а большего и не требуется, в случае же с со строкой различные подготовительные операции слегка увеличивают время.

Для размеров до 1000 данная операция выполняется быстрее в строке, это происходит потому что операция занимает линейной время и данных проход быстрее чем удаление элементов из дерева, с последующей балансировкой.

А при 50000 элементов мы замечаем сильный прирост производительности по сравнению со строкой. На самом деле данный показатель для дерева зависит от количества непохожих элементов, для строки же выполняется обход всех элементов.
Так как количество непохожих элементов с увеличением размера уменьшается, то и эффективность алгоритма на дереве возрастает.

Добавление

Размер	Время, нс
1	75
100	550
200	391
300	344
400	314
500	293
600	282
700	284
800	277
900	272
1000	277

В данном случае время определялось, так: считается среднее время добавления случайного элемента в дерево, данная операция производится 100 раз и вычисляется среднее.

Можем заметить что при размере время начинает уменьшатся - это происходит из-за того, что количество неуникальных элементов уменьшается, и не требуется выполнять добавление нового элемента, таким образом основные траты приходятся на поиск элемента в дереве - логарифмическое время, и обновление уже существующего.
Так же, стоит отметить, что при добавление элемента требуется балансировать дерево.

Удаление

Размер	Время, нс
1	51
100	376

Размер	Время, нс
200	254
300	173
400	144
500	118
600	103
700	98
800	87
900	82
1000	79

В данном случае время определялось, так: считается среднее время удаления случайного элемента из дерева, данная операция производится 100 раз и вычисляется среднее.

Можем заметить что при размере время начинает уменьшатся - это происходит из-за того, что количество неуникальных элементов уменьшается, и может потребоваться удаление уже отсутствующих элементов, таким образом основные траты приходится на поиск элемента в дереве - логарифмическое время.
Так же, стоит отметить, что при удалении элемента требуется балансировать дерево.

Память

Структура	1 элемент	1000 элементов	Структура	Доп. память для операций (в худшем случае), байт	
Строка	50001	50001	0	70	удаление
Дерево	33	33000	8	8008	удаление
				8008	добавление
Разница	99,9 %	34,0 %	-	-11340 %	удаление

Процент *вычисляется так: (СТРОКА - ДЕРЕВО)/СТРОКА, где*
СТРОКА - память, требуемая строкой
ДЕРЕВО - память, требуемая деревом

По данной таблице можем заметить что бинарное дерево поиска превосходит строку по объему требуемой памяти. Это происходит из-за того, что для строки используется статическая память и хранятся все элементы, а не только непохожие.

Вывод

В ходе проделанной работы, я приобрел навыки применения двоичных деревьев, реализовал основные операции над деревьями: обход деревьев, включение, исключение и поиск узлов. Изучил сбалансированное дерево. Провел сравнение эффективности дерева при разных исходных данных и сравнение дерева со строкой.

По результатам, проведенной оценки эффективности, было выявлено, что дерево очень эффективная по памяти структура в тех случаях когда данные имеют много повторяющихся элементов. При этом как по памяти так и по скорости работы она превосходит строку, для больших объемов данных. Также дерево имеет достаточно стабильную эффективность, в сравнение со строкой, на которой выполнение операций скалируется от объема данных.

Подробные результаты и методы оценки эффективности программы можно изучить в пункте «Оценка эффективности».

Контрольные вопросы

1. **Что такое дерево?**

   Нелинейная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим»

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

   В памяти дерево представляется как набор узлов, связанных между собой указателями на детей или массивом связи вершины с предком.

3. **Какие бывают типы деревьев?**

   Типы деревьев могут различаться в зависимости от количества потомков. Например, если у каждой вершины дерева имеется не более двух потомков (левые и правые поддеревья), то такое дерево называется двоичным или бинарным.
   Также деревья могут варьироваться по степени сбадлансированности.

Например, если вес любой пары потомков отличается не более чем на 1, то такое дерево называется идеально сбалансированным.

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

   Поиск по дереву, добавление элемента в дерево, удаление элемента из дерева, различные обходы дерева.

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

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