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 ПО ЛАБОРАТОРНОЙ РАБОТЕ №6
ПО ДИСЦИПЛИНЕ:
ТИПЫ И СТРУКТУРЫ ДАННЫХ

Деревья

Вариант 0

Студент **Ильченко Е. А.**

Группа **ИУ7-34Б**

Название предприятия **НУК ИУ МГТУ им. Н. Э. Баумана**

Студент _____ **Ильченко Е. А.**

Преподаватель _____ **Силантьева А. В.**

2024 г.

Описание условия задачи

Построить двоичное дерево поиска из букв вводимой строки. Вывести его на экран в виде дерева. Выделить цветом все буквы, встречающиеся более одного раза. Удалить из дерева эти буквы. Вывести оставшиеся элементы дерева при постфиксном его обходе. Сравнить время удаления повторяющихся букв из дерева и из строки. Реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов. Сравнить эффективность поиска в зависимости от высоты дерева и степени его ветвлени

Описание ТЗ

1. Описание исходных данных и результатов работы программы

Входные данные:
Пользовательская команда из доступных и необходимые аргументы определенного сценария:
1: Добавить строку в дерево
2: Добавить узел
3: Удалить узел
4: Вывести в строку
5: Вывести в виде дерева
6: Вывести узел
7: Удалить повторяющиеся буквы
8: Сравнить время удаления повторяющихся букв из дерева и из строки
9: Сравнить эффективность алгоритма поиска
10: Очистить дерево
11: Вывести меню
0: Выход

Выходные данные:
Дерево, измененное в соответствии с выбранной операцией

2. Описание задачи, реализуемой в программе

Задача, реализуемая в программе, заключается в реализации операций работы с двоичным деревом поиска; сравнить время удаления повторяющихся букв из дерева и из строки и эффективность алгоритма поиска

3. Способ обращения к программе

Запуск исполняемого файла

```
./app.exe
```

Далее выбирается, какой пункт меню выполнить

4. Описание возможных аварийных ситуаций и ошибок пользователя

Список аварийных ситуаций:

- При вводе неверной команды: сообщение “Неверная команда”
- При переполнении буфера при вводе строки: сообщение “Неверный ввод строки”
- При вводе пустой команды: сообщение “Пустая строка”
- При неверном вводе элемента дерева: сообщение “Ошибка ввода символа”
- При попытке вывести дерево в виде строки или в виде дерева: сообщение “Пустое дерево”
- При попытке вывода узла пустого дерева: сообщение “Пустое дерево”

5. Описание внутренних структур данных

Узел дерева

```
typedef struct Node {
   char data;
   int count;
   struct Node *left;
   struct Node *right;
} Node;
```

6. Описание функций

```
Node *create_node(char data)
```

Создает новый узел дерева с заданным значением

```
Node *insert(Node *root, char data)
```

Рекурсивно вставляет элемент с заданным значением в дерево

```
void post_order(Node *root)
```

Выполняет постфиксный обход дерева и выводит его в виде строки

```
Node *search(Node *root, char data)
```

Ищет узел с заданным значением в дереве

```
Node *min_value_node(Node *node)
```

Находит узел с минимальным значением в поддереве

```
Node *delete_node(Node *root, char data)
```

Удаляет узел с заданным значением из дерева

```
void free_tree(Node *root)
```

Рекурсивно освобождает память всех узлов дерева

```
Node *delete_duplicates(Node *root)
```

Удаляет узлы с повторяющимися элементами

```
void generate_graphviz(Node *root, FILE *file)
```

Генерирует файл для построения дерева с помощью graphviz

```
void save_tree_to_graphviz(Node *root, const char *filename)
```

Сохраняет дерево в формат для обработки gpaphviz

```
void save_to_png_from_graphviz(Node *node, int option)
```

Сохраняет дерево в png

7. Описание алгоритма

Алгоритм вставки элемента в дерево

Инициализация узла:

1. Передача значения: В функцию `insert()` передается указатель на корень текущего дерева (root) и значение (data), которое нужно добавить в дерево.
2. Проверка, является ли текущий узел пустым:
    - Если `root == NULL` (текущего узла нет), вызывается функция `create_node(data)`, которая:
        - Выделяет память под новый узел.
        - Записывает в него значение `data`.
        - Устанавливает указатели `left` и `right` в `NULL`.
    - Возвращается созданный узел как новый корень текущего поддерева.

Добавление узла в дерево:

1. Если текущее поддерево не пустое `(root != NULL)`:
    - Сравнение добавляемого значения с текущим узлом:

- Если `data < root->data` (значение добавляемого узла меньше значения текущего узла), рекурсивно вызываем функцию `insert()` для левого поддерева (`root->left`).
- Если `data > root->data` (значение больше текущего), рекурсивно вызываем функцию `insert()` для правого поддерева (`root->right`).
- Если `data == root->data` (значение совпадает с текущим узлом), увеличиваем счётчик повторений `root->count++`.

2. После завершения рекурсивных вызовов возвращается указатель на корень текущего дерева (с уже добавленным элементом).

Алгоритм поиска узла с заданным значением в дереве

1. Проверка текущего узла:
   a. Если дерево пустое (`root == NULL`), возвращаем `NULL` (узел не найден).
   b. Если значение текущего узла равно искомому (`root->data == data`), возвращаем указатель на этот узел
2. Рекурсивный поиск:
   a. Если значение меньше текущего узла, ищем в левом поддереве.
   b. Если значение больше текущего узла, ищем в правом поддереве.
3. Возврат результата: После завершения поиска возвращается либо указатель на найденный узел, либо `NULL`, если узел не существует.

Алгоритм удаления узла с заданным значением в дереве

1. Поиск узла с заданным значением:
   a. Если дерево пустое, возвращаем `NULL`.
   b. Если значение меньше текущего узла, ищем в левом поддереве.
   c. Если больше, ищем в правом поддереве.
   d. Если значение равно текущему узлу, найден узел для удаления.
2. Удаление узла:
   a. Если узел без потомков, освобождаем память и возвращаем `NULL`.
   b. Если у узла один потомок, возвращаем указатель на этого потомка после освобождения памяти.
   c. Если у узла два потомка:
      i. Находим минимальный узел в правом поддереве.
      ii. Копируем его значение в текущий узел.
      iii. Удаляем минимальный узел из правого поддерева
3. Возврат обновлённого дерева: После удаления возвращаем корень дерева с обновлённой структурой.

Алгоритм удаления узлов с повторяющимися значениями

1. Проверка текущего узла:
    a. Если дерево пустое (`root == NULL`), возвращаем `NULL`.
2. Рекурсивное удаление дубликатов:
    a. Рекурсивно вызываем `delete_duplicates` для левого и правого поддеревьев.
3. Удаление текущего узла с дубликатами:
    a. Если счётчик повторений узла (`root->count > 1`), вызываем `delete_node` для удаления текущего узла.
4. Возврат результата: После обработки возвращаем указатель на корень дерева без дубликатов.

Тесты

Тест	**Входные данные**	**Выходные данные**
Добавление узла в дерево	Вставить символ A в пустое дерево	Узел A добавлен, дерево: A
Удаление узла из дерева	.—— D .—— C —— B `—— A Удалить узел C	Узел C удален .—— D —— B `—— A
Создание дерева из строки	Строка: "ABC"	.—— C .—— B —— A
Вывод дерева в строку (постфиксный обход)	.—— C —— B `—— A	Строка: "A C B"
Вывод дерева с повторяющимися буквами	Строка: "AABBC"	.—— C .—— B(к) —— A(к)
Поиск узла по символу и вывод поддерева	.—— D .—— C —— B `—— A Поиск C	.—— D —— C
Удаление повторяющихся букв	Дерево: корень B (2), узлы A (2), C	Повторяющиеся узлы удалены, дерево: —— C

Удаление несуществующего узла из дерева	.—— C —— B `—— A Удалить D	Дерево не изменено: .—— C —— B `—— A
Поиск несуществующего символа и вывод поддерева	Дерево: корень B, узлы A, C, поиск D	Результат пустое дерево
Пустая ввод пустой строки для дерева	Пустая строка	Сообщение: “Пустое дерево”
Попытка вывода пустого дерева	Пустое дерево	Сообщение: “Пустое дерево”

Оценка эффективности

Эффективность по времени/памяти считалась путем замера 1000 раз методов удаления повторяющихся букв из дерева и из строки и поиска элементов и усреднения результатов

Выгода считалась, как

$$\frac{String - Tree}{Tree} \cdot 100$$

Измерения проводились на MacBook Pro 13 2019

Сравнение времени удаления повторяющихся букв из дерева и из строки

Длина строки	**Строка**		**Дерево**		**Выгода**	
	Время, нс	Память, байт	Время, нс	Память, байт	Время, %	Память, %
10	106.98	10	107.77	216	-0.73	-95.37
110	3412.76	110	1675.36	2040	103.70	-94.60
210	9468.66	210	3557.12	3360	166.18	-93.75
310	16355.70	310	5555.27	4272	194.41	-92.74
410	23886.38	410	7297.20	5016	227.33	-91.82
510	30933.34	510	9036.57	5400	242.31	-90.55
610	38756.57	610	11946.42	5640	224.41	-89.18
710	47200.17	710	12497.90	5736	277.66	-87.62

810	52599.79	810	11996.33	5832	338.46	-86.11
910	52280.72	910	11855.89	5832	340.96	-84.39

Сравнение эффективности алгоритма поиска в деревьях

Глубина	**Ветвление**	**Время, нс**
2	1	71.82
4	1	62.52
3	2	60.79
4	2	59.72
4	3	62.41
5	2	59.82

Вывод

Бинарные деревья поиска представляют собой эффективные средства для организации и управления данными. Проанализировав алгоритмы, можно сделать выводы:

1. Использование деревьев позволяет значительно ускорить поиск необходимых данных относительно строк примерно в 3 раза
2. Использовать алгоритм поиска и удаления повторяющихся символов разумно с большими объемами данных, при условии что затраты по памяти больше примерно в 6 раз по сравнению со строками
3. Анализируя алгоритм поиска элементов, можно сделать вывод, что в случае дерева символов он работает достаточно быстро при любой глубине дерева.

Ответы на контрольные вопросы

1. **Что такое дерево? Как выделяется память под представление деревьев?**
   Дерево в информатике – это структура данных, состоящая из узлов, связанных между собой ребрами. Один из узлов называется корнем, остальные разделяются на узлы и листья. Узлы, соединенные ребрами, образуют поддеревья.
   Память под представление деревьев обычно выделяется динамически. Каждый узел дерева содержит информацию и указатели на своих потомков (или

нулевые указатели, если потомков нет). Для каждого узла память выделяется отдельно при добавлении новых узлов.

2. **Какие бывают типы деревьев?**
   Существует множество типов деревьев, вот некоторые из них:
   1. Бинарное дерево: Каждый узел имеет не более двух потомков.
   2. Дерево бинарного поиска: Узлы упорядочены так, что для каждого узла все узлы в его левом поддереве меньше его, а в правом — больше.
   3. N-арное дерево: Каждый узел может иметь произвольное количество потомков.
   4. Распределенное дерево: Используется в распределенных вычислениях и сетевых структурах.
   5. AVL-дерево, красно-черное дерево: Сбалансированные бинарные деревья для эффективного поиска.
3. **Какие стандартные операции возможны над деревьями?**
   Стандартные операции над деревьями включают:
   a. Добавление узла: Вставка нового узла в дерево.
   b. Удаление узла: Удаление существующего узла из дерева.
   c. Поиск узла: Нахождение узла с определенным значением.
   d. Обход дерева: Посещение всех узлов дерева в определенном порядке (прямой, обратный, симметричный).
   e. Вывод дерева в виде строки: Представление дерева в текстовой или графической форме.
4. **Что такое дерево двоичного поиска?**
   Дерево двоичного поиска – это бинарное дерево, в котором каждый узел имеет не более двух потомков. При этом для каждого узла выполнено следующее свойство: все узлы в левом поддереве меньше текущего узла, а все узлы в правом поддереве больше текущего узла. Это свойство делает дерево двоичного поиска эффективной структурой данных для поиска, вставки и удаления элементов, так как оно обеспечивает логарифмическую сложность этих операций в среднем случае.