Лабораторная работа № 2 по курсу дискретного анализа: словарь

Выполнил студент группы 08-208 МАИ *Скворцов Александр.*

Условие

1. Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до 264 - 1. Разным словам может быть поставлен в соответствие один и тот же номер.

2. Вариант задания: красно-чёрное дерево.

Метод решения

Красно-чёрное дерево - одно из самобалансирующихся двоичных деревьев поиска, для которого выполнено:

- Узел либо красный, либо чёрный.
- Корень — чёрный.
- Все листья чёрные.
- Оба потомка красного узла — чёрные.
- Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число чёрных узлов.

Данные ограничения гарантируют логарифмический рост высоты дерева от количества узлов, однако для их выполнения требуется перебалансировка дерева при вставке или удалении.

Вставка данного узла выполняется как в обычном бинарном дереве, а затем вызывается процедура восстановления баланса:

- "Дядя"этого узла тоже красный. Тогда, чтобы сохранить свойства 3 и 4, просто перекрашиваем "отца"и "дядю"в чёрный цвет, а "деда"— в красный. Проверяем, не нарушена ли балансировка для "дедушки".
- "Дядя"чёрный. Выполняем правый поворот относительно "отца". Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком.

Удаление данного узла выполняется как в обычном бинарном дереве, а затем вызывается процедура восстановления баланса. Трудности могут возникнуть только, если удаляемый узел черный, причем по построению у него только один потомок, тогда:

- Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Снова проверяем балансировку для данной вершины.
- Брат и оба его ребёнка чёрные. Красим брата в красный цвет и делаем далее отца вершины в черный и рассматриваем его.
- Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение.

Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева.

Проанализировав данные алгоритмы, становится понятно, что для корректной работы программы каждый узел должен хранить не только свой цвет, но и указатель на родителя. Также среди особенностей реализации стоит ометить, что указатели на листовые узлы не следует делать nullptr'ми, так они не могут хранить цвет, иначе это несколько усложнит алгоритм и ухудшит скорость его выполнения. Вместо этого можно использовать единственный барьерный элемент.

Описание программы

Для удобства программа разделена на несколько файлов:

1. main.cpp - Обеспечивает ввод данных и управление методами дерева.
    - char* getWord(int *wordSize) - вводит слово, состоящее из букв нижнего регистра, и записывает его длину в переменную wordSize.
    - void formattedStr(char *str) - переводит все буквы строки str в нижний регистр.
    - int main() - Обеспечивает ввод данных и управление методами дерева.
2. ttree.h - Объявляет класс красно-черного дерева TTree и предоставляет его интерфейс.
3. ttree.cpp - Описывает методы класса TTree.
    - TTree() - Создает барьерный элемент дерева и приравнивает корень к нему.
    - Node* Min(Node *n) - возвращет минимальный элемент дерева, корнем которого является n.

- Node* Max(Node *n) - возвращет максимальный элемент дерева, корнем которого является n.
- Node Min() - возвращет минимальный элемент данного объекта.
- Node Max() - возвращет максимальный элемент данного объекта.
- Node* Successor(Node *n) - возвращает следующий элемент по значению для n.
- void LeftRotate(Node *n) - осуществляет левый поворот относительно n.
- void RightRotate(Node *n) - осуществляет правый поворот относительно n.
- void FixUpAdd(Node *n) - осуществляет перебалансировку для n после добавления.
- bool Add(char *word, int wordSize, unsigned long int val) - добавляет слово word длиной wordSize со значением val в дерево.
- void Search(char *word) - осуществляет поиск слова word в дереве.
- void FixUpDelete(Node *n) - восстанавливает баланс дерева для n после удаления.
- bool Delete(char *word) - удаляет элемент с ключом word.
- void DeleteTree(Node *n) - удаляет все элементы дерева, корнем которого является n.
- void Save(const char *ch) - сохраняет дерево в бинарном виде в файл с именем ch.
- void SaveNode(std::ofstream &fout, Node *n) - сохраняет дерево, корнем которого является n, в бинарном виде в поток fout.
- void Load(const char *ch) - загружает дерево из файла с именем ch.
- TTree() - удаляет данное дерево совместно с барьерным элементом.

Дневник отладки

Номер посылки	Причина неудачи	Причина ошибки
1	Unknown file extension архив в формате zip	архив пересобран
2-5	Time limit exceeded at test 04.t	попытки оптимизаций часто вызываемых функций
6	Wrong answer at test 04.t	функция ввода не приводила строки к нижнему регисту исправленние ввода

7	Time limit exceeded at test 04.t	отимизация стандартного потока встроенными стредствами
8	Time limit exceeded at test 05.t	полное переписывание способа хранения слова
9	Runtime error at test 11.t, got signal 6	в попытках поиска дампа, обнаружена при считывании пустого файла. Исправление
10	Wrong answer at test 03.t	Исправлена ошибка в измененной функции ввода.
11	Runtime error at test 11.t, got signal 6	Обнаружен выход за пределы массива, в строчке размером 255 символов. Исправлен размер массива.

Тест производительности

кол-во данных	приблизительная высота	тест 1	тест 2	тест 3
32	$\log_2 32 = 5$	9	11	10
1024	$\log_2 1024 = 10$	19	20	20
1048576	$\log_2 1048576 = 20$	31	28	37

В колонке "тест"указана скорость добавления одного элемента в микросекундах. Тогда из приведенной таблицы видно, что скорость доступа к элементам пропорцианальна логарифму от количества данных в дереве, что согласуется с указанной сложностью.

Недочёты

Существенных недочетов, влияющих на производительность программы не выявлено, однако пристутствуют несколько логических недостатков. К примеру методы дерева не должны выводить никакой информации на экран, а их общение с пользователем должно происходить посредством возвращаемого значения. Так было бы логичней, если методы Save и Load возвращали значение типа bool, а метод Search возвращал указатель на найденный узел или nullptr.

Вывод

Красно-черное дерево — одно из самобалансирующихся двоичных деревьев поиска, гарантирующее выполнение всех основных операций за $O(\log n)$. В силу такой асимптотики оно часто используется на практике для хранения какой-либо информации, к примеру стандартные контейнеры set и map библиотеки STL C++ основаны на нём. По сравнению с популярным АВЛ деревом имеет преимущество при удалении, например в красно-чёрном может потребоваться до трех поворотов, а в АВЛ их число может быть равно высоте дерева.

Сложность программирования такой структуры данных невысокая, если заранее известен алгоритм, но в противном случае придется изрядно потрудится, чтобы скорость работы программы соответсвовала ожиданиям. Основную трудность в данном случае представляют функции восстановления баланса при добавлении и удалении элемента.