Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Дискретный анализ»

Студент: Л. Я. Вельтман
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2018

Лабораторная работа №2

Задача: Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь.

Структура данных: Red-Black Tree.

Вариант ключа: регистронезависимая последовательность букв английского алфавита длиной не более 256 символов.

Вариант значения: числа от 0 до $2^{64} - 1$.

1 Описание

Red-Black Tree

Красно-чёрным называется бинарное поисковое дерево, у которого каждому узлу сопоставлен дополнительный атрибут — цвет и для которого выполняются следующие свойства:

1. Каждая вершина либо красная, либо чёрная.
2. Корень и конечные узлы, так называемые NIL-листья, дерева — чёрные.
3. У красного узла оба ребёнка окрашены чёрным цветом.
4. Все пути из любого узла x до листьев содержат одинаковое количество чёрных вершин (чёрная высота).

Как сказано в [4]: «для красно-чёрных деревьев операции поиска узла» (сюда входит поиск узла по ключу, поиск младшего узла в дереве, старшего, поиск потомка и предка) «выполняются за время O(lgN), так как время их выполнения O(h), h - высота дерева, а красно-чёрное дерево с n вершинами имеет высоту O(lgN). Процедуры удаления и вставки узла работают за O(lgN)».

Операции вставки и удаления вершин могут нарушать свойства красно-чёрного дерева. Чтобы восстановить эти свойства, нужно перекрашивать некоторые вершины и менять структуру дерева. Для изменения структуры используются операции, называемые вращением(поворотами). Вращения бывают левые и правые.

При описании алгоритмов для работы с деревом я столкнулась с тем, что мне нужно проверять существование узлов, но так как обращение к несуществующим элементам (обращение по нулевому адресу памяти) ведет к неопределенному поведению программы и аварийному завершению, было принято решение использовать фиктивный лист (sentinel), проинициализированный особым образом. Это помогло в упрощении алгоритма и также сэкономило память, так как известно, каждая вершина имеет двух потомков, и лишь NIL не имеет потомков. Таким образом, каждая вершина становится внутренней (имеющей потомков, пусть и фиктивных), а листьями будут лишь фиктивные вершины NIL.

2 Исходный код

1. main.cpp (содержит основной метод main, Parser - функция, в которой считывались символы входного потока и преобразовывались к нижнему регистру с помощью метода ToLower).

2. TRBTree.hpp и TRBTree.cpp (описание и реализация класса TRBTree и класса TNode).

3. Dictionary.cpp (реализация функций работы со словарем).

main.cpp	
char LowerCase(char symb);	Перевод symb в нижний регистр.
void Parser(char &action, TKey* str, TVal &val);	Обработка входных данных.
int main (int argc, const char * argv[])	Точка входа в программу.
TRBTree.hpp и TRBTree.cpp	
TNode();	Конструктор узла.
TNode(TKey* key, TVal value, TNodeColor col);	Конструктор нового узла с заданными значениями.
TRBTree();	Конструктор дерева.
TNode* NodeSearch(TKey* key);	Поиск в дереве.
bool Insertion(TKey* key, TVal value, TNodeColor color);	Вставка узла в дерево.
void FixInsert(TNode* node);	Балансировка дерева после вставки вершины.
void LeftRotate(TNode* node);	Левый поворот.
void RightRotate(TNode* node);	Правый поворот.
TNode* MinValueNode(TNode* node);	Поиск вершины с минимальным значением ключа.
void TransPlant(TNode* old, TNode* fresh);	Смена старого узла на новый.
void Deletion(TNode* node);	Удаление узла из дерева.
void FixDelete(TNode* node);	Балансировка дерева после удаления вершины.
void Clear(TNode* node);	Удаление дерева.
~TNode();	Деструктор узла.
~TRBTree();	Деструктор дерева.
Dictionary.cpp	
void TRBTree::SavingTree(std::ofstream &output, TNode* root)	Сохранение дерева в файл.

void TRBTree::LoadingTree(std::ifstream &input, TRBTree* tree)	Загрузка дерева из файла.

```
const int KEY_SIZE = 257;
typedef char TKey;
typedef unsigned long long TVal;
typedef bool TNodeColor;
const TNodeColor RED = true;
const TNodeColor BLACK = false;

class TNode {
public:
    TNode();
    TNode(TKey* key, TVal value, TNodeColor col);
    ~TNode();
    TNode* parent;
    TNode* left;
    TNode* right;
    TNodeColor color;
    TKey* key;
    TVal value;
};

class TRBTree {
public:
    TRBTree();
    ~TRBTree();
    TNode* nil;
    TNode* root;
    TNode* NodeSearch(TKey* key);
    bool Insertion(TKey* key, TVal value, TNodeColor color);
    void Deletion(TNode* node);
    void SavingTree(std::ofstream& output, TNode* root);
    void LoadingTree(std::ifstream& input, TRBTree* tree);

protected:
    TNode* MinValueNode(TNode* node);
    void Clear(TNode* node);
    void TransPlant(TNode* old, TNode* fresh);
    void FixInsert(TNode* node);
    void LeftRotate(TNode* node);
    void RightRotate(TNode* node);
    void FixDelete(TNode* node);
};
```

3 Консоль

```
MacBook-Pro-Lina:da2 linuxoid$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -O3 -o da2
*.cpp
MacBook-Pro-Lina:da2 linuxoid$ cat test
+ fgfg 444
+ qaa 333
+ yUihs 666
+ gbvm 777
+ ttR 111
-qaA
+ vvCxx 222
+ Nbvcu 999
gbVm
+ ppskskdJJJdsd 888
FgFg
-yuihS
yuihs
Nbvcu
-vvCxx
+ xG 4545454545
-qaa
Aaaaaaaaa
-NbvcU
vvCxx
QaA
-PPskskdJJJdsd
vvCxx
Fgfg
ppSkSkdJJJdsd
xG
-xG
xG
MacBook-Pro-Lina:da2 linuxoid$ ./da2 < test
OK
OK
OK
OK
OK
OK
```

```
OK
OK
OK: 777
OK
OK: 444
OK
NoSuchWord
OK: 999
OK
OK
NoSuchWord
NoSuchWord
OK
NoSuchWord
NoSuchWord
OK
NoSuchWord
OK: 444
NoSuchWord
OK: 4545454545
OK
NoSuchWord
```

4 Тест производительности

Для теста производительности я использую std::map из STL для сравнения со своей структурой. std::map — отсортированный ассоциативный контейнер, который содержит пары ключ-значение с неповторяющимися ключами. Данный тип, как правило, реализуется как красно-чёрное дерево. Тест производительности представляет из себя следующее: 1 тест содержит в себе 28 строк, 2 тест - 1472 строки, 3 тест состоит из 500000 строк, где элементы могут добавляться и удаляться. Каждый элемент состоит из ключа и значения.

```
MacBook-Pro-Lina:da2 linuxoid$ ./map < 1test
std::map time (sec): 0.000596
MacBook-Pro-Lina:da2 linuxoid$ ./da2 < 1test
Red-Black Tree time (sec): 0.000181

MacBook-Pro-Lina:da2 linuxoid$ ./map < 2test
std::map time (sec): 0.016047
MacBook-Pro-Lina:da2 linuxoid$ ./da2 < 2test
Red-Black Tree time (sec): 0.004510

MacBook-Pro-Lina:da2 linuxoid$ ./map < 3test
std::map time (sec): 5.690939
MacBook-Pro-Lina:da2 linuxoid$ ./da2 < 3test
Red-Black Tree time (sec): 0.935431
```

Анализируя время выполнения данных тестов каждой из структур, можно сделать вывод, что красно-чёрное дерево, реализованное мной, работает намного быстрее, чем map из стандартной библиотеки шаблонов как на маленьких, так и на больших данных. Это может быть связано с тем, что std::map может быть каким-то сбалансированным бинарным деревом поиска, но он может быть сбалансирован с использованием какого-то другого алгоритма. Возможно роль играет чтение полной строки, в то время как в своей программе я осуществляю посимвольное чтение и рационально распределяю память под каждый ключ.

5 Выводы

Red-Black Tree является одним из способов решения проблемы бинарного дерева поиска. В BST методы вставки и удаления вершин, гарантируя сохранение свойства упорядоченности, не способствуют оптимизации основных операций. Например, если вставить в BST последовательность возрастающих или убывающих чисел, оно превратится, по сути, в двусвязный список, а основные операции будут занимать время, пропорциональное количеству вершин, а не его логарифму. Таким образом, для получения производительности порядка O(lgN) нужно, чтобы дерево имело как можно более высокую сбалансированность (то есть имело возможно меньшую высоту), что и позволяет красно-чёрное дерево. В данной лабораторной работе также нужно было выполнить сохранение и выгрузку дерева(словаря). Запись(сохранение) в файл работает за линейное время O(n), где n - количество вершин, а чтение из файла(выгрузка словаря) также работает за линейное время O(n), так как при сохранении узлов в файл, я указываю дополнительные характеристики(цвет) помимо самого ключа и значения, это позволяет не балансировать дерево, что заметно ускоряет работу. Красно-чёрные деревья являются наиболее активно используемыми на практике самобалансирующимися деревьями поиска. Например, красно-чёрное дерево используется в следующем:

- Java: java.util.TreeMap, java.util.TreeSet.
- C++ STL: map, multimap, multiset.
- Ядро Linux: полностью справедливый планировщик, linux/rbtree.h

Список литературы

[1] *Красно-чёрное дерево — Википедия.*
URL: `https://ru.wikipedia.org/wiki/Красно-чёрное_дерево`.
(дата обращения: 10.11.2018).

[2] *Красно-черное дерево — Университет ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево`.
(дата обращения: 10.11.2018).

[3] *Lectures.stargeo — Конспект лекций.*
URL: `http://lectures.stargeo.ru`.
(дата обращения: 10.11.2018).

[4] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))