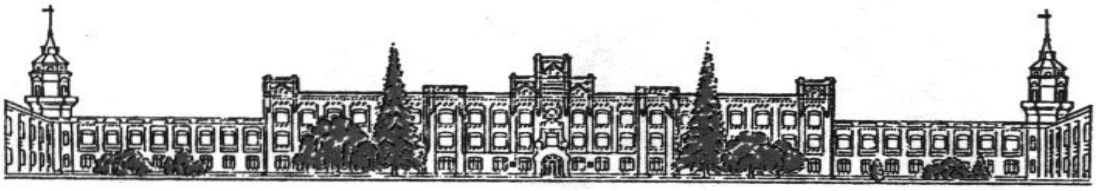

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра системного проектування

Алгоритми та структури даних

Лабораторна робота №4

“Дослідження структури даних бінарне дерево пошуку”

Київ – 2021

1 Мета роботи

Ознайомитись і дослідити структури даних бінарне дерево пошуку та префіксне дерево, розглянути механізми балансування дерева. Набути навичок реалізації бінарного дерева пошуку мовою програмування C++, порівняти власну реалізацію з готовим бібліотечним рішенням STL.

2 Короткі теоретичні відомості

- Бінарне дерево пошуку:
  https://echo.lviv.ua/dev/6964
  https://neerc.ifmo.ru/wiki/index.php?title=Дерево_поиска,_наивная_реализация
- АВЛ-дерево:
  https://coderlessons.com/tutorials/kompiuternoe-programmirovanie/izuchite-strukturu-dannykh-i-algoritmy/struktura-dannykh-i-algoritmy-derevia-avl
  http://lits.ua/article/blogs/data-structure-avl
- Пошук елементів в бінарному дереві пошуку на проміжку:
  https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1126/lectures/20/Slides20.pdf#page=102
- Операції split та merge:
  https://neerc.ifmo.ru/wiki/index.php?title=Декартово_дерево#split
- Префіксне дерево:
  https://otus.ru/nest/post/676/
- Використання STL контейнера vector:
  https://code-live.ru/post/cpp-vector/

3 Базове завдання

Обрати одну із запропонованих задач та реалізувати бінарне дерево пошуку для ефективнного вирішення цієї задачі.

1. **Створити структуру для зберігання об’єктів з характеристиками відповідно до обраної задачі, обрати критерій для порівняння двох об’єктів та перевантажити необхідні оператори.**

2. Реалізувати бінарне дерево пошуку:

2.1 Створити базовий елемент дерева Node, що буде містити в собі дані та вказівники на двох нащадків – лівого та правого. За необхідності додати вказівник на батьківський вузол.

2.2 Створити структуру BinarySearchTree з основними методами бінарного дерева пошуку без балансування:

- insert(object) додати новий елемент в дерево (без повторень)
- find(object) перевірити наявність елемента в дереві
- erase(object) видалити елемент з дерева
- size() знайти кількість елементів в дереві

2.3 Реалізувати додаткові методи для роботи з деревом:

- print() вивести всі елементи дерева у відсортованому порядку
- height() знайти висоту дерева
- findInRange(minObject, maxObject) знайти всі елементи в дереві на проміжку [minObject, maxObject], повернути їх кількість (або динамічний масив (vector) з цими елементами)

3. Провести тестування, використавши вказану нижче функцію testBinarySearchTree(). Перевірити правильність та швидкість роботи, порівнявши з готовим бібліотечним рішенням STL set.

Додаткові завдання:

1. Реалізувати методи бінарного дерева пошуку для роботи з піддеревами:
   - merge(tree1, tree2) об’єднати два дерева в одне
   - split(tree, object) розділити бінарне дерево пошуку по ключу на два інших, в першому всі елементи < object, в другому >= object
   - eraseRange(minObject, maxObject) видалити всі елементи дерева на проміжку [minObject, maxObject], використавши створені вище методи
2. Реалізувати у BinarySearchTree логіку балансування по типу АВЛ-дерева або іншого збалансованого дерева (червоно-чорне дерево, splay дерево)

4 Доповнене завдання

Реалізувати префіксне дерево для роботи програми автодоповнення слів.

Приклад виконання:

Input: > algorit

Output: > algorithm, algorithmic, algorithmically, algorithms

1. Створити базовий елемент дерева Node, що буде містити хеш-таблицю, де ключ – це текстовий символ, значення – вказівник на відповідний Node, а також булеву помітку чи є цей вузол кінцем слова. Замість хеш-таблиці можна використати статичний або динамічний масив, хоча це не дуже оптимально.
2. Створити структуру Trie та реалізувати основні методи префіксного дерева:
   - insert(word) додати нове слово в дерево
   - findByPrefix(word) знайти всі слова, які починаються на заданий префікс
3. Зчитати з файлу всі існуючі слова та побудувати з ними префіксне дерево. Файл зі словами можна взяти тут:
   https://github.com/dwyl/english-words/raw/master/words_alpha.txt
   https://github.com/dwyl/english-words/blob/master/words_alpha.zip?raw=true
4. Протестувати правильність роботи префіксного дерева, використавши різні префікси, що вводяться з клавіатури.

5 Код для тестування

```
#include <set>
#include <vector>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

bool testBinarySearchTree()
{
    srand(time(NULL));

    const int iters = 80000;
    const int keysAmount = iters * 2;
    const int itersToRangeQueries = 1000;

    vector<Data> data(keysAmount);

    vector<Data> dataToInsert(iters);
    vector<Data> dataToErase(iters);
    vector<Data> dataToFind(iters);
    vector<pair<Data, Data>> dataToRangeQueries;
```

```
for (int i = 0; i < iters; i++)
{
    dataToInsert[i] = data[generateRandLong() % keysAmount];
    dataToErase[i] = data[generateRandLong() % keysAmount];
    dataToFind[i] = data[generateRandLong() % keysAmount];
}

for (int i = 0; i < itersToRangeQueries; i++)
{
    Data minData = Data();
    Data maxData = Data();

    if (maxData < minData)
    {
        swap(minData, maxData);
    }
    dataToRangeQueries.push_back({minData, maxData});
}

BinarySearchTree myTree;
clock_t myStart = clock();
for (int i = 0; i < iters; i++)
{
    myTree.insert(dataToInsert[i]);
}
int myInsertSize = myTree.size();
int myTreeHeight = myTree.height();
int optimalTreeHeight = log2(myInsertSize) + 1;
for (int i = 0; i < iters; i++)
{
    myTree.erase(dataToErase[i]);
}
int myEraseSize = myInsertSize - myTree.size();
int myFoundAmount = 0;
for (int i = 0; i < iters; i++)
{
    if (myTree.find(dataToFind[i]))
    {
        myFoundAmount++;
    }
}
clock_t myEnd = clock();
float myTime = (float(myEnd - myStart)) / CLOCKS_PER_SEC;

set<Data> stlTree;
clock_t stlStart = clock();
for (int i = 0; i < iters; i++)
{
    stlTree.insert(dataToInsert[i]);
}
int stlInsertSize = stlTree.size();
for (int i = 0; i < iters; i++)
{
    stlTree.erase(dataToErase[i]);
}
int stlEraseSize = stlInsertSize - stlTree.size();
int stlFoundAmount = 0;
for (int i = 0; i < iters; i++)
{
    if (stlTree.find(dataToFind[i]) != stlTree.end())
    {
        stlFoundAmount++;
    }
}
clock_t stlEnd = clock();
float stlTime = (float(stlEnd - stlStart)) / CLOCKS_PER_SEC;
```

```
    clock_t myRangeStart = clock();
    int myRangeFoundAmount = 0;
    for (int i = 0; i < itersToRangeQueries; i++)
    {
        myRangeFoundAmount += myTree.findInRange(dataToRangeQueries[i].first,
dataToRangeQueries[i].second);
    }
    clock_t myRangeEnd = clock();
    float myRangeTime = (float(myRangeEnd - myRangeStart)) / CLOCKS_PER_SEC;

    clock_t stlRangeStart = clock();
    int stlRangeFoundAmount = 0;
    for (int i = 0; i < itersToRangeQueries; i++)
    {
        const auto& low = stlTree.lower_bound(dataToRangeQueries[i].first);
        const auto& up = stlTree.upper_bound(dataToRangeQueries[i].second);
        stlRangeFoundAmount += distance(low, up);
    }
    clock_t stlRangeEnd = clock();
    float stlRangeTime = (float(stlRangeEnd - stlRangeStart)) / CLOCKS_PER_SEC;

    cout << "My BinaryTree: height = " << myTreeHeight << ", optimal height = " <<
optimalTreeHeight << endl;
    cout << "Time: " << myTime << ", size: " << myInsertSize << " - " << myEraseSize << ",
found amount: " << myFoundAmount << endl;
    cout << "Range time: " << myRangeTime << ", range found amount: " << myRangeFoundAmount
<< endl << endl;
    cout << "STL Tree:" << endl;
    cout << "Time: " << stlTime << ", size: " << stlInsertSize << " - " << stlEraseSize <<
", found amount: " << stlFoundAmount << endl;
    cout << "Range time: " << stlRangeTime << ", range found amount: " <<
stlRangeFoundAmount << endl << endl;

    if (myInsertSize == stlInsertSize && myEraseSize == stlEraseSize &&
        myFoundAmount == stlFoundAmount && myRangeFoundAmount == stlRangeFoundAmount)
    {
        cout << "The lab is completed" << endl;
        return true;
    }

    cerr << ":(" << endl;
    return false;
}
```

6 Зміст звіту

Звіт має містити:

1) Титульний аркуш
2) Мету роботи
3) Варіант завдання
4) Хід виконання роботи:
    a) Умова задачі
    b) Скріншоти результатів виконання
    c) Лістинг програми (код)
5) Висновки

7 Контрольні питання

1) Навіщо потрібна стуктура даних бінарне дерево пошуку, які його переваги та недоліки порівняно з хеш-таблицею?
2) Який принцип побудови бінарного дерева пошуку? Опишіть три алгоритми обходу дерева.
3) Чому бінарне дерево пошуку може працювати як звичайний зв’язний список та як цього не допустити?
4) Що таке АВЛ-дерево, в чому його відмінності від інших збалансованих дерев? Як перетворити бінарне дерево пошуку в АВЛ-дерево?
5) Чим відрізняються між собою наступні STL контейнери: map, set, unordered_map, unordered_set? Коли який потрібно використовувати? Які структури даних використовуються для реалізації кожного контейнера?
6) Навіщо потрібна структура даних префіксне дерево? В чому його переваги над бінарним деревом пошуку? Які обмеження на його застосування?

8 Варіанти завдань

Задача 1

Структура Гравець має наступні поля: нікнейм, ранг, кількість досвіду, розмір донату тощо. Створити відсортовану “базу даних” гравців, в якій можна швидко перевіряти наявність потрібного гравця та знаходити всіх гравців на вказаному проміжку.

Задача 2

Структура Студент має наступні поля: ім’я, середній бал, бажання вчитися, кількість списаних робіт тощо. Створити відсортовану “базу даних” студентів, в якій можна швидко перевіряти наявність потрібного студента та знаходити всіх студентів на вказаному проміжку.

Задача 3

Структура Викладач має наступні поля: ім’я, оцінка в кампусі, якість лекцій, наявність статистики в телеграм каналі про викладачів тощо. Створити відсортовану “базу даних” викладачів, в якій можна швидко перевіряти наявність потрібного викладача та знаходити всіх викладачів на вказаному проміжку.

Власна задача

Самостійно придумати задачу в якій буде оптимально використати бінарне дерево пошуку.