

Звіт

Лабораторна робота №4

“Дослідження структури даних

бінарне дерево пошуку”

Студента групи ДA-12

Краковича Павла Дмитровича

Київ – 2022

1. Мета роботи

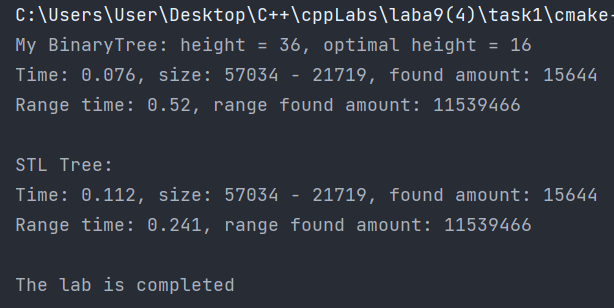
Ознайомитись і дослідити структури даних бінарне дерево пошуку та префіксне дерево, розглянути механізми балансування дерева. Набути навичок реалізації бінарного дерева пошуку мовою програмування С++, порівняти власну реалізацію з готовим бібліотечним рішенням STL.

2. Варіант роботи - №1

3. Хід виконання роботи

№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 (між двома гравцями).
* Результат роботи програми:



* Лістинг(код) програми:
  + main.cpp

#include <set>  
#include <vector>  
#include <iostream>  
#include <cstdlib>  
#include <ctime>  
#include <cmath>  
#include "searchTree.h"  
  
unsigned long long generateRandLong()  
{  
 unsigned long long result = 0;  
 int iters = rand() % 14 + 6;  
 for (int i = 0; i < iters; i++) {  
 result += (rand() % 10) \* pow(10, i);  
 }  
 return result;  
}  
  
bool testBinarySearchTree()  
{  
 srand(time(nullptr));  
 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]);  
 }  
  
// cout << "The contents of the tree:" << endl;  
// myTree.print();  
  
 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).size();  
 }  
 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;  
}  
  
int main() {  
 testBinarySearchTree();  
 return 0;  
}

* + searchTree.h

//  
// Created by User on 21.06.2022.  
//  
  
#ifndef TASK1\_SEARCHTREE\_H  
#define TASK1\_SEARCHTREE\_H  
#include <vector>  
  
using namespace std;  
  
struct Data {  
 int lvl;  
 int daysPlayed;  
  
 Data()  
 : lvl (rand())  
 , daysPlayed(rand())  
 {  
 }  
  
 bool operator < (const Data &d2) const {  
 if (lvl != d2.lvl) {  
 return lvl < d2.lvl;  
 }  
 else return daysPlayed < d2.daysPlayed;  
 }  
  
 bool operator == (const Data &d2) const {  
 return lvl == d2.lvl && daysPlayed == d2.daysPlayed;  
 }  
};  
  
struct Node {  
 Data data;  
 Node\* left;  
 Node\* right;  
};  
  
struct BinarySearchTree {  
 Node\* root = nullptr;  
  
 void insert(Data& data) {  
 root = insertInner(root, data);  
 }  
  
 Node\* insertInner(Node\* node, Data& data) {  
 if (!node) {  
 return new Node{.data = data};  
 }  
 else if (data < node->data) {  
 node->left = insertInner(node->left, data);  
 }  
 else if (node->data < data) {  
 node->right = insertInner(node->right, data);  
 }  
 return node;  
 }  
  
 Node\* minValueNode(Node \*node) {  
 if (!node->left) {  
 return node;  
 }  
 return minValueNode(node->left);  
 }  
  
 void erase(Data& data) {  
 eraseInner(root, data);  
 }  
  
 Node\* eraseInner(Node\* node, Data& data){  
  
 // Return if the tree is empty  
 if (!node) {  
 return nullptr;  
 }  
 // Find the node to be deleted  
 if (data < node->data) {  
 node->left = eraseInner(node->left, data);  
 }  
 else if (node->data < data) {  
 node->right = eraseInner(node->right, data);  
 }  
 else {  
 if (!node->left && !node->right) {  
 delete node;  
 return nullptr;  
 }  
 else if (!node->left) {  
 Node\* temp = node;  
 node = node->right;  
 delete temp;  
 }  
 else if (!node->right) {  
 Node\* temp = node;  
 node = node->left;  
 delete temp;  
 }  
 else if (node->right && node->left) {  
 Node\* temp = minValueNode(node->right);  
 node->data = temp->data;  
 node->right = eraseInner(node->right,temp->data);  
 }  
 }  
 return node;  
 }  
  
  
 bool find(Data& data) {  
 return findInner(root,data);  
 }  
  
 Node\* findInner(Node\* node, Data& data)  
 {  
 if (!node || node->data == data) {  
 return node;  
 }  
 else if (data < node->data) {  
 return findInner(node->left, data);  
 }  
 else {  
 return findInner(node->right, data);  
 }  
 }  
  
 int size() {  
 return sizeInner(root);  
 }  
  
 int sizeInner(Node\* node) {  
 if (!node) {  
 return 0;  
 }  
 size\_t rightSize = sizeInner(node->right);  
 size\_t leftSize = sizeInner(node->left);  
 return rightSize + leftSize + 1;  
 }  
  
 int height() {  
 return heightInner(root);  
 }  
  
 int heightInner(Node\* node) {  
 if (!node) {  
 return -1;  
 }  
  
 int rightHeight = heightInner(node->right);  
 int leftHeight = heightInner(node->left);  
  
 if (leftHeight > rightHeight) {  
 return leftHeight + 1;  
 }  
 else return rightHeight + 1;  
 }  
  
  
 vector<Data> findInRange(const Data& left, const Data& right) {  
 vector<Data> vector;  
 innerFindInRange(vector, root, left, right);  
 return vector;  
 }  
  
 void innerFindInRange(vector<Data>& vector, Node\* node, const Data& left, const Data& right) {  
 if (!node) {  
 return;  
 }  
 else if(node->data < left) {  
 innerFindInRange(vector, node->right, left, right);  
 }  
 else if(right < node->data) {  
 innerFindInRange(vector, node->left, left, right);  
 }  
 else {  
 vector.push\_back(node->data);  
 innerFindInRange(vector, node->left, left, right);  
 innerFindInRange(vector, node->right, left, right);  
 }  
 }  
  
  
 void clear(Node\* node) {  
 if (!node) {  
 return;  
 }  
 if (node->left) {  
 clear(node->left);  
 }  
 if (node->right) {  
 clear(node->right);  
 }  
 node = nullptr;  
 delete node;  
 }  
  
 void print () {  
 printInner(root);  
 }  
  
 void printInner(Node \*node) {  
 if (node) {  
 printInner(node->left);  
 cout << "|" << node->data.lvl << " lvl, " << node->data.daysPlayed << " days |" << endl;  
 printInner(node->right);  
 }  
 }  
  
 ~BinarySearchTree() {  
 clear(root);  
 }  
  
};  
  
#endif //TASK1\_SEARCHTREE\_H

Висновки

У цій лабораторній роботі я ознайомився з новими структурами даних, а саме: бінарним деревом пошуку та префіксним деревом.

За рахунок того, що heap sort використовує бінарну купу, його асимптотична складність дорівнює O(n\*log n), що швидше за більшість інших алгоритмів сортування, але для правильної роботи цього алгоритму потрібна, відповідно, бінарна купа, асимптотична складність створення якої = O(n).

Методи SiftUp, SiftDown в бінарній купі, за асимптотикою дорівнюють O(log n), де N — кількість елементів у купі.

З цього можна зробити висновок, що додавання та видалення елементів з бінарної купи також відбувається за O(log n). Тоді, коли інші алгоритмів сортування, які ми вчили до цього на курсі потребують O(n) часу.

Перевіривши результати замірів часу роботи програми, я можу зробити висновок, що хоч моє дерево не є збалансованим, але все одно працює швидше за STL реалізацію, бо містить менший функціонал.

Контрольні питання

1) Навіщо потрібна стуктура даних бінарне дерево пошуку, які її переваги та недоліки порівняно з хеш-таблицею?

*Бінарне дерево пошуку - це бінарне дерево, що має додаткові властивості, відмінні від звичайного бінарного дерева, а саме: значення лівого нащадка менше значення батька, а значення правого нащадка більше значення батька для кожного вузла дерева. Тобто дані в бінарному дереві пошуку зберігаються у відсортованому вигляді. При кожній операції вставлення нового або видалення існуючого вузла відсортований порядок дерева зберігається. При пошуку елемента порівнюється потрібне значення з коренем. Якщо шукане більше кореня, то пошук продовжується в правому нащадку кореня, якщо менше, то в лівому, якщо одно, то значення знайдено і припиняється пошук.*

*Бінарне дерево пошуку виконує ті ж самі завдання, що і хеш-таблиця, але має переваги. Наприклад, якщо нам потрібно працювати з даними рядковго типу, то порівнювати їх та сортувати у дереві буде набагато простіше, ніж створити хеш-функцію.*

*Також, швидко шукати найбільший або найменший елемент, вивести елементи по спаданню чи зростанню та працювати з елементами на проміжку можна робити це набагато швидше, ніж у хеш-таблці.*

2) Який принцип побудови бінарного дерева пошуку? Опишіть три алгоритми обходу дерева.

*Бінарне дерево пошуку будується за наступним принципом: значення лівого нащадка менше значення батька, а значення правого нащадка більше значення батька для кожного вузла дерева.*

*На відміну від пов'язаних списків, одновимірних масивів та інших лінійних структур даних, для яких існує канонічний обхід за допомогою лінійного порядку, дерева можна обходити у різні способи. Існує три найпоширеніші способи їх проходження вглиб: прямий (pre-order), зворотній (post-order) та серединний (in-order).*

* *Прямий порядок обходу:*

1. *Заходимо у перший вузол*
2. *Рекурсивно обходимо праве піддерево вузла*
3. *Рекурсивно обходимо ліве піддерево вузла*

* *Центрований порядок обходу:*

1. *Рекурсивно обходимо ліве піддерево вузла*
2. *Заходимо у корінь*
3. *Рекурсивно обходимо праве піддерево вузла*

* *Зворотній порядок обходу:*

1. *Рекурсивно обходимо ліве піддерево вузла*
2. *Рекурсивно обходимо праве піддерево вузла*
3. *Заходимо у корінь*

3) Чому бінарне дерево пошуку може працювати як звичайний зв’язний список, та як цього не допустити?

*Бінарне дерево пошуку може працювати як звичайний зв’язний список, якщо на кожному рівні буде присутній лише один елемент. Таке можливо, якщо вхідні дані вже будуть відсортовані. Щоб уникнути цього, потрібно збалансувати дерево.*

4) Що таке АВЛ-дерево, в чому його відмінності від інших збалансованих

дерев? Як перетворити бінарне дерево пошуку в АВЛ-дерево?

*Це звичайне бінарне дерево пошуку з додатковою умовою: Для кожної вершини висота її двох піддерев відрізняється не більше ніж на 1.*

*Щоб перетворити звичайне бінарне дерево в АВЛ-дерево, потрібно його збалансувати.*

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?

*Подібності:*

*set та map подібні тим, що вони обидва використовують червоне чорне дерево (самобалансуючий BST).*

*Відмінності:*

*Встановлено різницю використовується для зберігання лише ключів, тоді як map використовується для зберігання пар ключ-значення.*

*«map» — це асоційований контейнер, у якому зберігаються елементи, утворені комбінацією «ключ-значення» та зіставленого значення.*

*«set» — це асоціативний контейнер, який містить набір унікальних об’єктів типу Key.*

*Різниця між невпорядкованою та звичайною версією полягає в тому, що значення не впорядковані*

*"set" і "map" використовуються, коли потрібні впорядковані дані, потрібно роздрукувати / отримати доступ до даних (у відсортованому порядку) та потрібен попередник / наступник елементів.*

*Їх невпорядковані версії використовуються, коли потрібно вести підрахунок деяких даних, не потрібно впорядковувати їх, та коли потрібен доступ до одного елемента без обходу.*

6) Навіщо потрібна структура даних префіксне дерево? В чому його переваги над бінарним деревом пошуку? Які обмеження на його застосування?

*Префіксне дерево дозволяє зберігати асоціативний масив, ключами якого є рядки. На відміну від бінарних дерев, в листі дерева не зберігається ключ. Значення ключа можна набути прогляданням всіх батьківських вузлів, кожний з яких зберігає один або кілька символів алфавіту. Корінь дерева пов'язаний з порожнім рядком. Таким чином, нащадки вузла мають загальний префікс, звідки і відбулася назва даного абстрактного типу даних.*

*Префіксне дерево ефективне для вирішення задач з рядкам, забезпечують швидкий пошук і мають перевагу для пошуку слів у словниках, автодоповнення в пошукових системах і навіть для IP-маршрутизації.*

*Обмеження на роботу префіксних дерев полягають у розмірі слова, до якого можна робити доповнення через завелику кількість об’єктів у такому дереві.*