|  |  |
| --- | --- |
| Изображение выглядит как текст, небо, внешний  Автоматически созданное описание | |
| Національний технічний університет України  «Київський політехнічний інститут ім. Ігоря Сікорського»  Інститут прикладного системного аналізу  Кафедра системного проектування | |
| Алгоритми та структури даних | |
| **Лабораторна робота №2-3** | |
| **“Дослідження структур даних бінарна купа**  **та пріоритетна черга”** | |
| Студент гр. ДА-12 | Панасенко О. А. |
| Київ - 2022 | |

**Мета роботи**

Ознайомитись і дослідити структуру даних бінарна купа. Набути навичок реалізації абстрактної структури даних черга з пріоритетом на основі бінарної купи мовою програмування С++. Ознайомитись з механізмом перевантаження операторів та дослідити особливості сортування купою (пірамідального сортування). Порівняти власну реалізацію пріоритетної черги з готовимбібліотечним рішенням STL.

**Завдання** (*Варіант завдання: 18*)

*Візуалізатор (Renderer) ігрового рушія має здатність додавати об'єкти до рендер-баферу. Об'єкти, що знаходяться далі від камери (z-координата більша) мають виводитись на екран першими.*

*Renderer повинен мати змогу отримати об'єкт, який йому потрібно вивести на екран, тим не менш під час роботи у рендер-бафер можуть додаватись нові об'єкти з різною відстаню від камери.*

**Створити структуру Object для зберігання об’єктів з характеристиками відповідно до обраної задачі:**

1. Обрати певний власний критерій для визначення більш пріоритетного об’єкту на основі всіх його характеристик
2. Перевантажити оператор “менше” (operator<) для порівняння двох об’єктів за критерієм
3. Перевантажити оператор “дорівнює” (operator==) для визначення рівності двох об’єктів за цим самим критерієм

**Реалізувати пріоритетну чергу на основі незростаючої бінарної купи:**

* Створити структуру PriorityQueue, яка буде містити в собі бінарне дерево для роботи бінарної купи. Для зберігання бінарного дерева можна використати звичайний динамічний масив, що вже був реалізований у минулих роботах.
* Реалізувати основні методи переміщення по бінарному дереву:
  + getParent(index) – знайти індекс батьківського вузла
  + getLeftChild(index) – знайти індекс лівого дочірнього вузла
  + getRightChild(index) – знайти індекс правого дочірнього вузла
* Реалізувати внутрішні методи для підтримки властивості незростаючої (max-heap) купи:
  + siftUp(index) – просіяти елемент вверх по дереву
  + siftDown(index) – просіяти елемент вниз по дереву
* Реалізувати основні методи для роботи пріоритетної черги:
  + push(object) – додати новий елемент в чергу
  + top() – отримати верхній елемент з черги
  + pop() – видалити верхній елемент з черги
  + size() – знайти кількість елементів в черзі
  + empty() – перевірити чергу на пустоту

**Провести тестування, використавши вказану нижче функцію testPriorityQueue():**

* Перевірити правильність та швидкість роботи, порівнявши з готовим бібліотечним рішенням STL priority\_queue.

Продумати реалізацію сортування купою (пірамідального сортування) на базі бінарної купи, вміти обгрунтовано пояснити та показувати в коді.

Додаткове завдання:

* Реалізувати сортування купою

Порівняти його з готовим бібліотечним рішенням STL std::sort() або з іншими алгоритмами сортувань реалізованими в минулих роботах – провести тестування з замірами часу на різних вхідних даних, зробити аналіз та висновки.

**Хід виконання роботи**

Нижче буде наведено код усієї програми, виконання усіх завдань. Програму я вирішив поділити на три файли: основний файл з функціями тестування та main, файл з реалізацією приорітетної черги та файл з додатковим завданням. Більшість функцій та структур я створив з використанням темплейтів, а отже їх реалізацію я не виносив в окремий файл, а залишив в хедері. Реалізацію приорітетної черги я вирішив зробити поверх std::vector структури з бібліотеки STL. Для зручності, додаткове завдання HeapSort я вирішив виконати за допомогою ітераторів, які містять в собі всі STL структури даних. Над блоком кода буде вказана назва файлу, до якого він належить.

*Код програми:*

*Файл Main.cpp*

#include <iostream>

#include <queue>

#include <PriorityQueue.h>

#include <algorithm>

#include <Sorting.h>

#include <chrono>

#include <random>

template <typename Queue, typename T>

float testPriorityQueueSpeed(Queue&& priorityQueue)

{

const int iters = 100000;

clock\_t timeStart = clock();

for (int i = 0; i < iters; i++)

{

int insertDataAmount = rand() % 6 + 5;

for (int j = 0; j < insertDataAmount; j++)

{

priorityQueue.push(T{static\_cast<float>(j)});

// priorityQueue.push(T());

}

auto& a = priorityQueue.top();

priorityQueue.pop();

}

clock\_t timeEnd = clock();

float time = (float(timeEnd - timeStart)) / CLOCKS\_PER\_SEC;

return time;

}

template <typename T>

bool testPriorityQueue()

{

srand(time(nullptr));

const int iters = 10000;

PriorityQueue<T> myPriorQueue;

std::priority\_queue<T> stlPriorQueue;

bool isDataEqual = true;

for (int i = 0; i < iters; i++)

{

int insertDataAmount = rand() % 6 + 5;

for (int j = 0; j < insertDataAmount; j++)

{

T randData = T{static\_cast<float>((i + 1) \* (rand() % 100))};

// T randData = T();

myPriorQueue.push(randData);

stlPriorQueue.push(randData);

}

if (!(myPriorQueue.top() == stlPriorQueue.top()))

{

isDataEqual = false;

std::cerr << "Comparing failed on iteration " << i << std::endl << std::endl;

break;

}

int removeDataAmount = rand() % insertDataAmount;

for (int j = 0; j < removeDataAmount; j++)

{

myPriorQueue.pop();

stlPriorQueue.pop();

}

}

std::cout << std::endl;

int myQueueSize = myPriorQueue.Size();

int stlQueueSize = stlPriorQueue.size();

float stlTime = testPriorityQueueSpeed<std::priority\_queue<T>, T>(std::priority\_queue<T>());

float myTime = testPriorityQueueSpeed<PriorityQueue<T>, T>(PriorityQueue<T>());

std::cout << "My PriorityQueue:" << std::endl;

std::cout << "Time: " << myTime << ", size: " << myQueueSize << std::endl;

std::cout << "STL priority\_queue:" << std::endl;

std::cout << "Time: " << stlTime << ", size: " << stlQueueSize << std::endl << std::endl;

if (isDataEqual && myQueueSize == stlQueueSize)

{

std::cout << "The lab is completed" << std::endl << std::endl;

return true;

}

std::cerr << ":(" << std::endl << std::endl;

return false;

}

struct Object

{

float distance;

bool operator<(const Object& other) const { return this->distance < other.distance; }

bool operator>(const Object& other) const { return !(\*this < other); }

bool operator==(const Object& other) const { return this->distance == other.distance; }

bool operator!=(const Object& other) const { return !(\*this == other); }

};

int main()

{

std::cout << "[Log] Comparing std::priority\_queue and PriorityQueue...**\n**";

testPriorityQueue<Object>();

std::cout << "[Log] Comparing std::priority\_queue and PriorityQueue - done**\n**";

// Additional task

std::cout << "[Log] Comparing sorting algorithms...**\n**";

std::cout << "Additional task:" << std::endl;

std::vector<int> vec(1000000);

std::cout << "Created a vector of size " << vec.size() << std::endl;

std::generate(vec.begin(), vec.end(), []() -> int

{

return rand() % 123;

});

std::random\_device d;

std::mt19937 g(d());

auto start = std::chrono::steady\_clock::now();

std::sort(vec.begin(), vec.end());

auto end = std::chrono::steady\_clock::now();

std::chrono::duration<float> elapsed = end - start;

std::cout << "[Timer] std::sort() time elapsed: " << elapsed << "**\n**";

std::shuffle(vec.begin(), vec.end(), g);

start = std::chrono::steady\_clock::now();

HeapSortInplace(vec.begin(), vec.end());

end = std::chrono::steady\_clock::now();

elapsed = end - start;

std::cout << "[Timer] HeapSortInplace() time elapsed: " << elapsed << "**\n**";

std::cout << "[Log] Comparing sorting algorithms - done**\n**";

std::vector<int> heap(1000);

std::generate(heap.begin(), heap.end(), []() -> int

{

return rand() % 123;

});

std::cout << "[Log] Before MakeHeap(): " << (std::is\_heap(heap.begin(), heap.end()) ? "Is a heap**\n**" : "Not a heap**\n**");

std::cout << "[Log] Making a heap...**\n**";

MakeHeap(heap.begin(), heap.end());

std::cout << "[Log] Making a heap - done**\n**";

std::cout << "[Log] After MakeHeap(): " << (std::is\_heap(heap.begin(), heap.end()) ? "Is heap**\n**" : "Not a heap**\n**");

}

*Файл PriorityQueue.h*

#pragma once

#include <cstddef>

#include <vector>

#include <iostream>

#include <cmath>

template <typename T> concept Comparable = requires(T t) { t < t; };

template <Comparable T> class PriorityQueue

{

public:

using ValueType = T;

using SizeType = std::size\_t;

using Container = std::vector<ValueType>;

using Iterator = typename Container::iterator;

using Reference = ValueType&;

using ConstReference = const ValueType&;

PriorityQueue() = default;

~PriorityQueue() = default;

constexpr SizeType Parent(SizeType index) const { return (index - 1) / 2; }

// constexpr SizeType Parent(SizeType index) const { return (index - 1) / 2; }

constexpr SizeType LeftChild(SizeType index) const { return (2 \* index + 1); }

constexpr SizeType RightChild(SizeType index) const { return LeftChild(index) + 1; }

SizeType Size() const { return m\_Size; }

constexpr bool IsEmpty() const { return Size() == 0; }

constexpr bool IsLeaf(SizeType index) const { return LeftChild(index) >= Size(); }

constexpr bool IsRoot(SizeType index) const { return index == 0; }

constexpr Reference top() { return \*m\_Container.begin(); }

constexpr ConstReference top() const { return \*m\_Container.begin(); }

SizeType Height() const { return std::ceil(log2f(static\_cast<float>(Size()))); }

void SiftUp(SizeType index)

{

if (IsRoot(index))

{

return;

}

SizeType parentIndex = Parent(index);

Reference self = m\_Container[index];

Reference parent = m\_Container[parentIndex];

if (self < parent)

{

return;

}

std::swap(self, parent);

SiftUp(parentIndex);

}

void push(ConstReference& value)

{

m\_Container.push\_back(value);

m\_Size++;

SiftUp(Size() - 1);

}

void push(ValueType&& value)

{

m\_Container.push\_back(std::move(value));

m\_Size++;

SiftUp(Size() - 1);

}

void SiftDown(SizeType index)

{

// if true than this node is a leaf

// if not, than at least left child exists

if (IsLeaf(index))

{

return;

}

// Get left and right children

SizeType childIndex = LeftChild(index);

// Check if node at rightIndex exists

if (!(childIndex + 1 >= Size()))

{

// ensure if right index is larger than left index

childIndex = std::max<SizeType>(childIndex, childIndex + 1,

[&](const SizeType& a, const SizeType& b) -> bool

{

return m\_Container.at(a) < m\_Container.at(b);

});

}

Reference child = m\_Container.at(childIndex);

Reference self = m\_Container.at(index);

if (child < self)

{

return;

}

std::swap(child, self);

SiftDown(childIndex);

}

void pop()

{

if (this->IsEmpty())

{

return;

}

std::swap(m\_Container[0], m\_Container[Size() - 1]);

m\_Size--;

m\_Container.erase(m\_Container.end() - 1);

SiftDown(0);

}

Iterator begin() { return m\_Container.begin(); }

Iterator end() { return m\_Container.end(); }

private:

Container m\_Container{};

SizeType m\_Size{};

};

*Файл Sorting.h*

#pragma once

#include <algorithm>

#include <iostream>

template <typename Iterator> void SiftDown(Iterator iter, Iterator first, Iterator last)

{

auto index = std::distance(first, iter);

Iterator leftChild = first + (2 \* index + 1);

Iterator rightChild = first + (2 \* index + 2);

Iterator child = leftChild;

// while is not a leaf

while (leftChild <= last)

{

child = leftChild;

if (!(rightChild > last))

{

child = std::max(leftChild, rightChild,

[](const Iterator& lhs, const Iterator& rhs) -> bool

{

return \*lhs < \*rhs;

});

}

if (\*iter > \*child)

{

break;

}

std::iter\_swap(iter, child);

iter = child;

index = std::distance(first, iter);

leftChild = first + (2 \* index + 1);

rightChild = first + (2 \* index + 2);

}

}

template <typename Iterator> void MakeHeap(Iterator first, Iterator last)

{

// we start from container::end() - 1 and finish, when crossed container::begin()

last = last - 1;

for (Iterator iter = last; iter != first - 1; --iter)

{

SiftDown(iter, first, last);

}

}

template <typename Iterator> void PopHeap(Iterator first, Iterator last)

{

// we can access last - 1 here, assuming last - 1 != end

std::iter\_swap(first, last - 1);

SiftDown(first, first, last - 2); // pass extra iterator

}

template <typename Iterator> void HeapSortInplace(Iterator first, Iterator last)

{

Iterator iter = last;

MakeHeap(first, last);

while (first != iter)

{

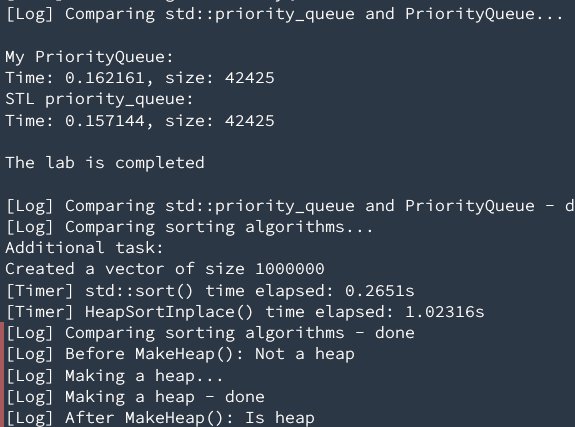
PopHeap(first, iter--);

}

std::reverse(first, last);

}

Результати виконання зображено на малюнку нижче

****

Малюнок 1. Результат виконання лабораторної роботи

**Висновки**

У цій лабораторній роботі я ознайомився з новою структурою даних — бінарною купою, а також з абстрактною структурою даних — приорітетною чергою. Окрім цього я розглянув новий спосіб сортування масивів — heap sort. Також закріпив знання з перегрузки операторів.

Бінарну купу я реалізував рекурсивно (тобто SiftUp, SiftDown — рекурсивні) і вирішив не виокремлювати її від приорітетної черги в окрему структуру. Натомість, heap sort реалізував циклами, без рекурсії, тому що це не займає великих об’ємів пам’яті, як це може бути у випадку з використанням рекурсії.

За рахунок того, що heap sort використовує бінарну купу «під капотом», його асимптотична складність дорівнює O(NlogN), 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, як не дивно за своєю асимптотикою дорівнюють O(logN), де N — кількість елементів у купі → а з цього слідує, що додавання та видалення елементів з бінарної купи також, відповідно O(logN), навідміну від інших алгоритмів сортування, які ми вчили раніше — O(N)

В якості допоміжних функцій та елементів С++, я використовував такі бібліотеки як: chrono (для заміру часу виконнаня сортування), algorithm (для порівняння сортувань, а також для допоміжних функцій, по типу std::max, std::generate, std::iter\_swap та інших) та random (для використання std::shuffle функції в додатковому завданні).

Оскільки під час виконання лабораторної я використовував стандарт С++20, я вирішив також використати нову утиліту — концепти, які, як на мене, перетворюють template на щось схоже на Generics в Java. Завдяки ним, можна перевіряти, чи задовільняє заданий шаблон темплейту умовам, вказаним у концепті.

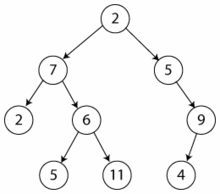
**Контрольні питання**

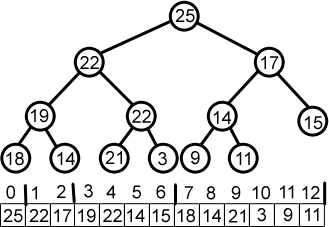
1) Приорітетна черга потрібна для випадків, коли треба отримувати об’єкт по «приорітету», або по критерію, який може переважати над іншими об’єктами. Вона відрізняється від черги тим, що, як мінімум, вона не зовсім FIFO. Але, навідміну від черги, дає змогу отримувати об’єкти з найбільшим приорітетом за O(1)

2)

* Динамічний масив: недолік полягає у тому, що після додавання елементу потрібно його сортувати — це не оптимізовано.
* Можна скористатись бінарним пошуком та динамічним масивом і тоді додавання елемента буде займати O(N), а отримати елемент можна буде за O(1)

3) Бінарне дерево — це підвид дерева, кожен вузол якої має максимум 2 дитини

Особливістю бінарної купи є те, що ця структура даних є масивом, який можна розглядати, майже, як бінарне дерево, у якому на всіх рівнях, крім останнього, дерево обов’язково має бути повністю заповненим, а кожен наступний рівень заповнюється елементами зліва-направо.

4) Бінарна купа представлене у вигляді масиву за допомогою формули. Таким чином кожному рівню дерева відповідає деяка частина масиву

5) Для побудови бінарної купи з деякого масиву даних спочатку треба, починаючи з кінця купи, пройти по всім вузлам, які не є листками. Для кожного вузла треба виконати просіювання вниз. Таким чином, складність побудови купи — O(N). Просіювання вниз являє собою порівняння двох елементів, що дещо нагадує сортування бульбашкою, до тих пір, поки умова порівняння не перестане виконуватись. Просіювання вгору, навідміну від попереднього, перевіряє чи є батькіский вузол меншим за даний, і, якщо так, міняється з ним місцями, продовжуючи ітерації далі, допоки умова не перестане виконуватись.

6) Основна ідея heap sort — якщо дано бінарну купу, то для отримання відсортованого масиву, достатньо проітеруватись по купі N разів та видалити верхній елемент. Враховуючи нюанси виконання цього алгоритму сортування, наприклад, swap&pop, просіювання вниз після перестановки, асимптотична складінсть складає O(NlogN). Недоліком є те, що для використання цього сортування, потрібно мати, відповідно, бінарну купу.