**Задание на курсовую**

**Оглавление**

**Глава 1.Структуры данных отвечающие интерфейсу очереди с приоритетами**

Очереди с приоритетами используются в таких алгоритмах как: алгоритм Дейкстры. алгоритм Прима, алгоритм Хаффмана, дискретно-событийное моделирование и так далее.

Двоичная куча – одна из простейших структур данных, отвечающая интерфейсу очереди с приоритетами. Время работы операций delete-min, insert, decreaseKey – *О*(log n), merge – *О*(n + m) где n и m кол-во элементов двух сливаемых куч.

Биномиальная куча — это структура данных, отвечающая интерфейсу приоритетной очереди. Время работы операций insert, find-min, merge, decreaseKey, delete, delete-min в худшем случае *O*(log n).[1]

Фибоначчиева куча — это структура данных также отвечающая интерфейсу приоритетной очереди. В отличии от биномиальной кучи, на деревья, из которых состоит фибоначчиева куча, не наложены никакие строгие ограничения по форме. Время работы операций create, insert, find-min, merge, decreaseKey — *O*(1), а операций delete, delete-min — *O*(log n).[1]

Тонкая куча – это структура данных похожая на фибоначчиеву кучу, и имеющая такие же асимптотические оценки. Время работы операций create, insert, find-min, merge, decreaseKey – *О*(1), а операций delete, delete-min – *О*(log n). Тонкая куча, при одинаковых асимптотических оценках, имеет меньшее кол-во констант по сравнению с фибоначчиевой кучей, что на практике должно ускорить работу и уменьшить объем затраченной памяти.[2]

Тонкое дерево – это биномиальное дерево, каждый узел которого мог потерять левого ребенка(или поддерево узлом которого являлся левый ребенок).

Более формально тонкое дерево – это упорядоченное дерево, каждый узел которого имеет положительный или равный 0 ранг, и удовлетворяет следующим свойствам:

1. узел с рангом r, либо имеет r детей с рангами r - 1, r - 2, …, 0 (такой узел называют толстым), либо имеет r - 1 детей с рангами r -2, r - 3, …, 0 (такой узел называют тонким);

2. корень является толстым узлом.

Заметим, что если соединить два тонких дерева, корни которых имеют одинаковый ранг r, сделав один из корней левым ребенком другого, то получится тонкое дерево с корнем ранга r + 1.

Тонкая куча – это набор кучеобразных(т.е. удовлетворяющих условиям кучи) тонких деревьев.

**Глава 2. Реализация структуры данных «Тонкая куча» (с максимумом в корне)**

*template <class T>*

*class Node*

*{*

*private:*

*T priority; //приоритет*

*int rank; //ранг*

*Node\* child; //указатель на самого левого ребенка*

*Node\* right; //указатель на правого брата, либо на следующий корень, если текущий узел является корнем*

*Node\* left; // указатель на левого брата , либо на родителя, если текущий узел самый левый, либо null, если текущий узел корень*

*public:*

*Node() { rank = 0; child = right = left = NULL; }*

*Node(T priority\_) { priority = priority\_; rank = 0; child = right = left = NULL; }*

*//~Node() { child = right = left = NULL; rank = -1; }*

*T getPriority() { return priority; }*

*int getRank() { return rank; }*

*Node\* getChild() { return child; }*

*Node\* getRight() { return right; }*

*Node\* getLeft() { return left; }*

*T setPriority(T priority\_) { priority = priority\_; return priority\_; }*

*int setRank(int rank\_) { rank = rank\_; return rank\_; }*

*Node\* setChild(Node\* child\_) { child = child\_; return child; }*

*Node\* setRight(Node\* right\_) { right = right\_; return right; }*

*Node\* setLeft(Node\* left\_) { left = left\_; return left; }*

*bool isThin()*

*{*

*if (getRank() == 0)return 0;*

*if (getChild() == NULL)return 1;*

*return (getRank() - getChild()->getRank() == 2 ? 1 : 0);*

*}*

*};*

Листинг 1. Структура узла

Узел состоит из приоритета, ранга (ранг соответствующего узла в биномиальном дереве) и указателей на самого левого ребенка, правого брата (либо на следующий корень в корневом списке, если текущий узел - корень) и левого брата (либо родителя, если узел самый левый из детей).

Функция isThin возвращает булево значение, 1 если корень тонкий, 0 если толстый.

Также написаны конструктор по умолчанию и конструктор принимающий приоритет.

Листинг 2. Структура тонкой кучи

*template <class T>*

*class thinheap*

*{*

*public:*

*Node<T>\* first; //указатель на корень с максимальным приоритетом*

*Node<T>\* last; //указатель на последний корень*

*thinheap() { first = last = NULL; }*

*~thinheap() { first = last = NULL; }*

*Node<T>\* getMax() { return first; }*

Тонкая куча состоит из указателей на первый и последний узлы в корневом списке.

Функция getMax возвращает указатель на первый узел в корневом списке.

*Node<T>\* insert(Node<T>\* newnode\_)*

*{*

*if (newnode\_ == NULL)return NULL;*

*if (first == NULL)*

*first = last = newnode\_;*

*else*

*{*

*if (newnode\_->getPriority() > first->getPriority())*

*{*

*newnode\_->setRight(first);*

*first = newnode\_;*

*}*

*else*

*{*

*last->setRight(newnode\_);*

*last = newnode\_;*

*newnode\_->setRight(NULL);*

*}*

*}*

*return newnode\_;*

*}*

*Node<T>\* insert(T priority\_)*

*{*

*Node<T>\* newnode = new Node<T>;*

*newnode->setPriority(priority\_);*

*return insert(newnode);*

*}*

Листинг 3. Перегруженные функции insert

Функция insert принимающая указатель на узел добавляет в корневой список тонкое дерево, корнем которого является узел newnode\_. Если приоритет добавляемого узла больше максимального, узел добавляется в начало и указатель на максимальный элемент обновляется. Если же приоритет меньше, узел добавляется в конец.

Функция insert принимающая приоритет создает новый узел с этим приоритетом и так же добавляет его в корневой список.

*thinheap merge(thinheap heap\_)*

*{*

*if (first->getPriority() > heap\_.first->getPriority())*

*{*

*last->setRight(heap\_.first);*

*last = heap\_.last;*

*}*

*else*

*{*

*heap\_.last->setRight(first);*

*first = heap\_.first;*

*}*

*return \*this;*

*}*

Листинг 4. Функция merge

Функция merge принимает тонкую кучу heap\_ и сливает корневые списки двух тонких куч и обновляет указатель на первый или последний элемент корневого списка.

Листинг 5. Функция extractMax

*Node<T>\* extractMax()*

*{*

*Node<T>\* res = first;*

*Node<T>\* cur = res->getChild();*

*while (cur != NULL)*

*{*

*Node<T>\* next = cur->getRight();*

*if (cur->isThin())*

*cur->setRank(cur->getRank() - 1);*

*cur->setLeft(NULL);*

*cur->setRight(NULL);*

*insert(cur);*

*cur = next;*

*}*

*cur = res->getRight();*

*Node<T>\* nodes[64] = { NULL };*

*int max\_rank = -1;*

*while (cur != NULL)*

*{*

*Node<T>\* next = cur->getRight();*

*while (nodes[cur->getRank()] != NULL)*

*{*

*if (cur->getPriority() < nodes[cur->getRank()]->getPriority()) std::swap(cur, nodes[cur->getRank()]);*

*nodes[cur->getRank()]->setLeft(cur);*

*if (cur->getChild() != NULL)*

*{*

*nodes[cur->getRank()]->setRight(cur->getChild());*

*cur->getChild()->setLeft(nodes[cur->getRank()]);*

*}*

*else*

*nodes[cur->getRank()]->setRight(NULL);*

*cur->setChild(nodes[cur->getRank()]);*

*nodes[cur->getRank()] = NULL;*

*cur->setRank(cur->getRank() + 1);*

*cur->setRight(NULL);*

*}*

*if (cur->getRank() > max\_rank)*

*max\_rank = cur->getRank();*

*nodes[cur->getRank()] = cur;*

*cur = next;*

*}*

*first = last = NULL;*

*for (int i = 0; i <= max\_rank; i++)*

*{*

*insert(nodes[i]);*

*}*

*return res;*

*}*

Функция extractMax удаляет из тонкой кучи узел с максимальным приоритетом, и возвращает указатель на этот узел. Сначала узел с максимальным приоритетом удаляется из корневого списка, после чего понижается ранг всех его “тонких” детей. Затем все дети добавляются в корневой список и создается массив из указателей на узлы, на i-том месте которого будет хранится тонкое дерево ранга i. После чего массив заполняется, а деревья одного ранга сливаются, пока не в массиве не останутся тонкие деревья различных рангов. Переменная Int max\_rang соответствует максимальному рангу тонких деревьев из массива и обновляется при каждом добавлении тонкого дерева в массив. Переменная max\_rang нужна для того, чтобы ограничить перебор массива. Далее обнуляются указатели на первый и последний элементы корневого списка, и затем все элементы массива добавляются в кучу, посредством функции insert, в связи с этим, после выполнения функции указатель first будет соответствовать узлу с максимальным приоритетом.

*bool sibling\_violation(Node<T>\* node\_)*

*{*

*if (node\_->getRight() == NULL && node\_->getRank() == 1)return 1;*

*if (node\_->getRight() != NULL && node\_->getRight()->getRank() + 2 == node\_->getRank())return 1;*

*return 0;*

*}*

*bool child\_violation(Node<T>\* node\_)*

*{*

*if (node\_->getRank() == 2 && node\_->getChild() == NULL)return 1;*

*if (node\_->getChild() != NULL && node\_->getChild()->getRank() + 3 == node\_->getRank())return 1;*

*return 0;*

*}*

Листинг 6. Вспомогательные функции sibling\_violation и child\_violation

Функции sibling\_violation и child\_violation возвращают булевы значения. При изменении приоритета могут возникнуть два типа нарушений целостности тонкой кучи. Так называемые “Братские” и “Родительские” нарушения. Функция sibling\_violation возвращает 1 если узел ранг узла node\_ на 2 больше ранга правого брата, или если node\_ не имеет правого брата, и ранг node\_ равен 1. 0 возвращается в том случае, если узел не является “узлом локализации нарушения”, то есть не нарушает целостность тонкой кучи.

Функция child\_violation возвращает 1, если узел node\_ не имеет детей, и его ранг равен 2, или если ранг узла node\_ на 3 больше ранга его самого левого ребенка. Аналогично функции sibling\_violation 0 возвращается если узел не нарушает целостность тонкой кучи.Листиг 7. Функция changePriority

*Node<T>\* changePriority(Node<T>\* node\_, T newpriority\_)*

*{*

*node\_->setPriority(newpriority\_);*

*if (node\_->getLeft() != NULL)*

*{*

*Node<T>\* left = node\_->getLeft();*

*if (node\_->getLeft()->getChild() == node\_)*

*{*

*if (node\_->getRight() == NULL)*

*node\_->getLeft()->setChild(NULL);*

*else*

*{*

*node\_->getLeft()->setChild(node\_->getRight());*

*node\_->getRight()->setLeft(node\_->getLeft());*

*}*

*}*

*else*

*{*

*if (node\_->getRight() == NULL)*

*node\_->getLeft()->setRight(NULL);*

*else*

*{*

*node\_->getLeft()->setRight(node\_->getRight());*

*node\_->getRight()->setLeft(node\_->getLeft());*

*}*

*}*

*while (sibling\_violation(left) && child\_violation(left))*

*{*

*if (sibling\_violation(left))*

*{*

*if (left->isThin())*

*{*

*left->setRank(left->getRank() - 1);*

*left = left->getLeft();*

*}*

*else*

*{*

*if (left->getRight()->getRight() != NULL)*

*left->getRight()->getRight()->setLeft(left);*

*left->getLeft()->setChild(left->getRight());*

*left->getRight()->setLeft(left->getLeft());*

*left->setLeft(left->getRight());*

*left->setRight(left->getLeft()->getRight());*

*left->getLeft()->setRight(left);*

*}*

*}*

*}*

*else*

*{*

*Node<T>\* newviolation = left->getLeft();*

*left->setRank(left->getRank() - 2);*

*if (left->getLeft()->getChild() != left)*

*{*

*left->getLeft()->setRight(left->getRight());*

*left->getRight()->setLeft(left->getLeft());*

*}*

*else*

*{*

*if (left->getRight() != NULL)*

*left->getRight()->setLeft(left->getLeft());*

*left->getLeft()->setChild(left->getRight());*

*}*

*left->setLeft(NULL);*

*left->setRight(NULL);*

*insert(left);*

*left = newviolation;*

*}*

*}*

*}*

*node\_->setLeft(NULL);*

*node\_->setRight(NULL);*

*insert(node\_);*

*if (node\_->isThin())*

*node\_->setRank(node\_->getRank() - 1);*

*return node\_;*

*}*

Функция changePriority получает на вход узел, и новый приоритет для этого узла, и возвращает указатель на этот узел после восстановления (если нужно) целостности тонкой кучи. Указатель left отвечает возможному после смены приоритета узлу локализации нарушения. Функция работает до тех пор, пока не будут устранены все “Братские” и “Родительские” нарушения. При “Братском” нарушении если узел локализации нарушения является тонким, его ранг понижается на 1, после чего узлом локализации нарушения будет его родитель либо левый брат. При “Родительском” нарушении тонкое поддерево, корнем которого является узел локализации нарушения, добавляется в корневой список, и если узел локализации нарушения не был самым левым из братьев, новым узлом локализации нарушения будет его левый брат. Если же узел локализации нарушения был самым старшим из братьев и его родитель не был тонким узлом, родитель становится тонким узлом, но если родитель был тонким, он становится новым узлом локализации нарушения.

Список литературы