# Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

# (СибГУТИ)

09.03.01 Информатика и вычислительная техника

Профиль: Электронно-вычислительные машины, комплексы, системы и сети

(очная форма обучения)

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

# в/на кафедре вычислительных систем СибГУТИ

(наименование профильной организации/структурного подразделения СибГУТИ)

# «РЕАЛИЗАЦИЯ СТЕКА НА БАЗЕ ОДНОСВЯЗНОГО СПИСКА»

Выполнил: Киреев И. А. студент института ИВТ

# гр. ИВ-121 / /

«27» мая 2023 г. (подпись)

# Проверил:

Руководитель от СибГУТИ / /

«27» мая 2023 г. (подпись)

# Новосибирск 2023

**План-график проведения** учебной **практики**

Вид практики

# Киреев Илья Анатольевич

Фамилия Имя Отчество студента

# института Информатика и вычислительная техника , 2 курса, гр. ИВ-121

Направление: 09.03.01 Информатика и вычислительная техника

Код – Наименование направления (специальности)

# Профиль: Электронно-вычислительные машины, комплексы, системы и сети

Место прохождения практики

Объем практики: **108/3** часов/ЗЕ Вид практики ***учебная***

Тип практики ***ознакомительная***

# Срок практики с "30" января 2023 г.

по "27" мая 2023 г.

# Содержание практики\*:

|  |  |
| --- | --- |
| Наименование видов деятельности | Дата (начало – окончание) |
| 1. Общее ознакомление со структурным подразделением предприятия, вводный инструктаж по технике безопасности | 30.01.2023–01.02.2023 |
| 2. Выдача задания на практику, деление студентов на группы (если необходимо), определение конкретной  индивидуальной темы, формирование плана работ | 02.02.2023–04.02.2023 |
| 3. Работа с библиотечными фондами структурного подразделения или предприятия, сбор и анализ материалов по теме практики | 06.02.2023–11.02.2023 |
| 1. Выполнение работ в соответствии с составленным   планом:   * 1. Разработка заголовочного файла с прототипами методов работы со структурой данных   2. Разработка системы сборки проекта СMake   3. Разработка структуры данных в отдельном файле   4. Разработка программы для тестирования полученной библиотеки для работы со структурой данных   5. Отладка и форматирование кода программы | 13.02.2023 – 20.05.2023 |
| 5. Анализ полученных результатов и произведенной работы. Составление отчета по практике, защита отчета | 22.05.2023–27.05.2023 |

\*В соответствии с программой практики

# Руководитель от СибГУТИ / /

«28» 01 2023г. (подпись)

### ЗАДАНИЕ НА ПРАКТИКУ

Реализовать программно, исследовать эффективность и описать структуру данных “Стек” на основе структуры данных «Односвязный список».

### ВВЕДЕНИЕ

Стек - это структура данных, основанная на односвязном списке, которая позволяет хранить и управлять элементами в порядке "последним пришел - первым вышел" (LIFO - Last In, First Out). Односвязный список состоит из узлов, каждый из которых содержит данные и ссылку на следующий узел. Стек на основе односвязного списка предлагает удобные операции добавления и удаления элементов. При добавлении нового элемента в стек, он становится вершиной списка, а его ссылка указывает на предыдущий элемент, образуя цепочку связанных узлов. Извлечение элемента из стека происходит путем изменения ссылки на вершину, что делает предыдущий элемент новой вершиной стека. Таким образом, только последний добавленный элемент может быть извлечен из стека.

### СТРУКТУРА ОДНОСВЯЗНОГО СПИСКА

Односвязный список является важной структурой данных, применяемой в программировании и информатике. Он представляет собой набор узлов, где каждый узел содержит данные и ссылку на следующий узел в списке. Односвязный список отличается от двунаправленного списка тем, что каждый узел содержит ссылку только на следующий узел, а не на предыдущий (Рис. 1).

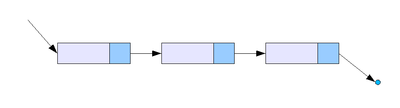


Рис. 1 — Абстрактное представление односвязного списка

Каждый узел односвязного списка состоит из двух основных частей: поля данных, которое хранит сами данные, и поля указателя, которое содержит ссылку на следующий узел в списке. Последний узел списка содержит ссылку на NULL или нулевой указатель, указывающий на конец списка. Односвязный список можно представить в виде последовательности узлов, связанных между собой ссылками. Начальный узел списка называется головой, а конечный узел - хвостом. Структура односвязного списка обеспечивает эффективность и гибкость при добавлении и удалении элементов (Рис. 2).



Рис. 2 — Программное представления односвязного списка

Добавление нового узла в односвязный список осуществляется путем создания нового узла, заполнения его данных и установки ссылки на него в поле указателя предыдущего узла. Таким образом, новый узел становится следующим за предыдущим узлом, образуя связь в списке. Удаление узла из односвязного списка происходит путем изменения ссылки в поле указателя предыдущего узла. Таким образом, предыдущий узел "пропускает" удаленный узел и связывается с последующим узлом, исключая удаленный узел из списка.

Односвязный список позволяет эффективно управлять данными, особенно при операциях вставки и удаления в середине списка. Однако доступ к элементам списка осуществляется последовательно, начиная с головы, что может потребовать больше времени для поиска определенного элемента.

Использование односвязного списка находит применение во многих областях, включая реализацию стеков, очередей, алгоритмов поиска и сортировки, а также связанных структур данных, таких как графы и деревья. Односвязный список обладает гибкостью и эффективностью, что делает его важным инструментом при разработке программного обеспечения.

### СТРУКТУРА СТЕКА

Стек является важной структурой данных, которая работает по принципу "последним пришел - первым вышел" (LIFO - Last In, First Out). Он позволяет хранить и управлять элементами в порядке их добавления, обеспечивая быстрый доступ к последнему добавленному элементу (Рис. 3).

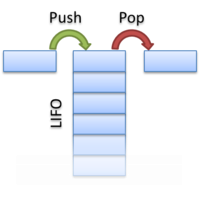


Рис. 3 — принцип работы структуры данных «Стек»

Структура стека может быть реализована на основе различных структур данных, включая односвязный список. Стек на основе односвязного списка представляет собой набор узлов, связанных между собой последовательно. Каждый узел содержит данные и ссылку на следующий узел в стеке. Последний добавленный узел становится вершиной стека, а его ссылка указывает на предыдущую вершину.

Операция добавления элемента в стек, называемая помещением или вставкой, осуществляется путем создания нового узла и установки ссылки на него в вершине стека. Новый узел становится новой вершиной стека, а его ссылка указывает на предыдущую вершину, образуя цепочку связанных узлов.

Операция удаления элемента из стека, называемая извлечением или удалением, происходит путем изменения ссылки в вершине стека на предыдущий узел. Таким образом, предыдущий узел становится новой вершиной стека, и его данные становятся доступными для использования (Рис. 4).

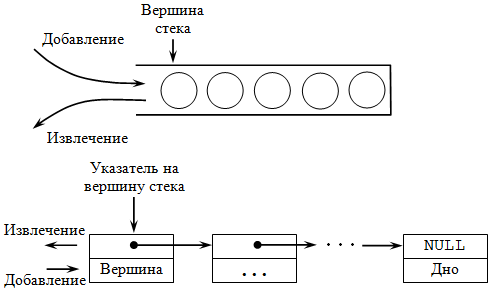


Рис. 4 — Представление операция добавления и извлечения элементов из стека

Структура стека на основе односвязного списка обладает несколькими преимуществами. Во-первых, она позволяет эффективно управлять данными, сохраняя порядок их добавления. Во-вторых, стек на основе односвязного списка может быть реализован динамически, то есть его размер не ограничен и может увеличиваться или уменьшаться по мере необходимости. Стеки на основе односвязного списка широко применяются в различных областях программирования и алгоритмических задачах. Они могут использоваться для реализации алгоритмов обхода деревьев, выполнения обратной польской записи, обработки вызовов функций и многих других задач. Стеки также являются важной составляющей в разработке компиляторов и интерпретаторов.

### АЛГОРИТМ РАБОТЫ

**Добавление элемента**

Добавление элемента в стек на основе связного списка является одной из основных операций, которую необходимо реализовать для работы со стеком. Эта операция позволяет добавить новый элемент на вершину стека.

Процесс добавления элемента в стек на основе связного списка включает несколько шагов. Вначале создается новый узел, который будет содержать добавляемый элемент. Затем устанавливается ссылка этого нового узла на текущую вершину стека. Далее обновляется ссылка вершины стека на новый узел, чтобы он стал новой вершиной.

Сложность операции добавления элемента в стек на основе связного списка составляет O(1), то есть она имеет постоянную временную сложность. При добавлении элемента, независимо от размера стека, требуется выполнить фиксированное количество операций: создание нового узла, обновление ссылок и обновление ссылки на вершину стека. Таким образом, время выполнения операции добавления элемента не зависит от количества элементов в стеке.

Ниже приведен псевдокод для операции добавления элемента в стек на основе связного списка:

procedure push(element):

// Шаг 1: Создание нового узла

new\_node <- create\_node(element)

// Шаг 2: Установка данных нового узла

set\_data(new\_node, element)

// Шаг 3: Установка связи со старым верхним элементом

set\_next(new\_node, top\_of\_stack)

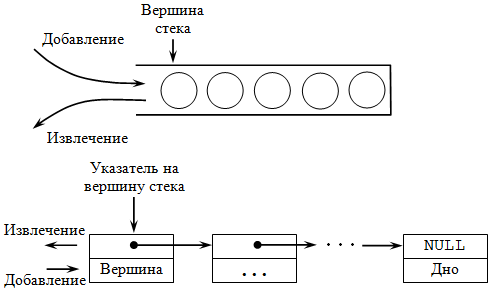
// Шаг 4: Обновление верхнего элемента стека

top\_of\_stack <- new\_node

end procedure

В этом псевдокоде используются следующие ключевые слова и функции:

* create\_node(element): функция, которая создает новый узел и возвращает его
* set\_data(node, element): функция, которая устанавливает данные узла равными указанному элементу
* set\_next(node, next\_node): функция, которая устанавливает ссылку на следующий узел у указанного узла
* top\_of\_stack: переменная, которая хранит ссылку на верхний элемент стека

Рис. 5 — Описание операции добавления элемента в стек на основе связного спика

### ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ

R-дерево как структура индексирования пространственных объектов стало почти стандартом для промышленных СУБД, которые используют различные его варианты в качестве индексных структур. Однако при конкретной реализации R-деревьев в конечном приложении зачастую встает вопрос, какие параметры выбрать для лучшей её реализации и, соответственно, достижения лучших вычислительной/пространственной сложностей. Временная сложность этих операций в большой нотации O зависит от количества записей данных в R-дереве (**n**), размерности данных, и максимального числа дочерних элементов в узле (**M**). Поиск, вставка и удаление (ключевые операции над структурой данных) имеют временную сложность в худшем случае **O( M\*logM(n) ),** в среднем - **O ( logM(n) )**, в лучшем - **O ( log(n) )**. Это связано с тем, что R-дерево представляет собой древовидную структуру данных, и поиск выполняется путем обхода дерева от корня до конечного узла, включая обход всех дочерних элементов в худшем случае. Высота дерева также логарифмично относится к количеству записей данных.

Как было отмечено ранее, одними из важнейших параметров для построения R-дерева являются минимально возможное (**m**) и максимально допустимое (**M**) количество элементов в узле. При выборе этих параметров необходимо руководствоваться следующими соображениями:

### Максимальное число элементов в узле (M)

Чем больше значение M, тем сильнее будет ветвиться дерево, а следовательно, его глубина будет меньше. Если предположить, что индексная структура разрабатывается для внешней памяти, то уменьшение глубины дерева ведет к уменьшению обращений к диску (если учесть, что проверка узла дерева вызывает одно обращение к диску). Поэтому сильноветвящееся дерево (при большом M) будет более эффективным для внешней памяти. С другой стороны, процедура поиска вынуждена просматривать абсолютно все элементы вершины. Поэтому при очень большом M индексная структура может выродиться просто к последовательному поиску. К тому же на сравнение с элементами вершины расходуется процессорная мощность. Поэтому чем больше M, тем больше нагрузка на процессор в процедурах поиска. Исходя из описанных фактов, можно сделать следующие выводы: если разрабатываемая индексная структура целиком размещается в оперативной памяти, то значение M стоит выбирать небольшим, порядка 4–10 элементов в вершине. Если же индексная структура хранится во внешней памяти, то значение M стоит вычислять по следующей формуле: M = Cluster / eSize , где Cluster – размер кластера жесткого диска (например, 512 или 1024 байт); eSize – размер одного элемента. Так, если один элемент занимает 16 байт, то в качестве верхней границы стоит взять M = 32 элементов в вершине.

### Минимальное число элементов в узле (m)

Данный параметр зависит от M и, как было описано ранее, не может превышать M / 2. Минимальный же предел параметра m равен 2 (в узле не может быть меньше двух потомков, если это не корневой узел). Маленькое значение параметра m облегчает процедуру разделения узла, потому что исчезает необходимость повторной вставки элементов. В то же время маленькое значение нижней границы может привести к неэффективному использованию памяти. По исследованию А. Гуттмана, наименее плотные индексы могут потреблять приблизительно на 50% больше места, чем самые плотные. В практических применениях наиболее часто используемой операцией является процедура поиска элементов. Поэтому нижнюю границу заполнения узла стоит выбирать равной M / 2.

Центральным звеном при построении дерева также является процедура **разбиения узла пополам** (**node\_split**). От эффективности этой процедуры зависит оптимальность построения дерева в целом. При неоптимальной структуре дерева появляется неоднозначность поиска элементов. Возможны ситуации, когда уже на уровнях, близких к корню R-дерева, охватывающие прямоугольники пересекаются не по пустому множеству данных, что значительно усложняет процедуру поиска.

С проблемой качественности изменения R-дерева можно бороться с помощью

**«исчерпывающего»** алгоритма деления. Использование данного алгоритма для деления узла изменяет структуру R-дерева лучшим из возможных способов, что, конечно, отражается на дальнейшем поиске данных в лучшую сторону, но, в свою очередь, существенно замедляет работу индексной структуры. Применение данного алгоритма оправдано при малом числе записей в узле, а также в ситуациях, когда структура дерева редко меняется, т. е. при индексировании неподвижных (например, жилых домов, складов и т. д.) или малоподвижных пространственных объектов (например, небесной карты звезд).

### ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

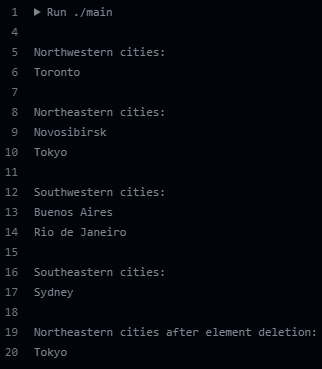
Для выполнения моделирования обозначенной структуры данных были созданы файлы rtree.c, rtree.h как модуль для работы с R-деревьями и файл main.c как тестовая программа. В заголовочном файле rtree.h описаны прототипы функций и глобальные директивы препроцессора, задающие тип данных, с которым будет работать R-дерево, минимальное и максимальное количество элементов в узле дерева, количество измерений пространства, с которыми будет взаимодействовать пользователь при работе со структурой данных, а в rtree.c - реализация этих и скрытых от пользователя функций, которые необходимы для корректной работы и построения R-деревьев. Имея псевдокод описанных выше функций, было несложно реализовать их на языке программирования C. В качестве функции разбиения узла был выбран оригинальный квадратичный алгоритм Антонина Гуттмана, в качестве значений M - 64 элемента, m - 6 элементов (10% от M (относительная реализация) соответственно). Также был создан Makefile для автоматизации компиляции программы. В файле программы для тестирования реализации полученной структуры данных (**main.c**), была создана новая структура **city**, хранящая в себе название города и его широту-долготу, и несколько экземпляров этой структуры в качестве известных городов мира, располагающихся в разных частях света: Торонто, Новосибирск, Токио, Буэнос Айрес, Рио де Жанейро и Сидней. Эти экземпляры были переданы в новосозданное R-дерево, с которым далее были выполнены операции поиска по разным широте-долготе (северо-западные, северо-восточные, юго-западные и юго-восточные города относительно нулевого меридиана и экватора, где -180 - 0 градусы - это юг и запад в понимании широты и долготы, 0 - 180 градусы - север и восток соответственно) и удаления элементов из дерева (для примера был удалён Новосибирск, находящийся в северо-восточной четверть сфере Земли). R-дерево умеет находить всю область с заданными параметрами и выводить все объекты, находящиеся в ней. Чтобы работать R-дереву с новой структурой **city**, было достаточно передавать в функции работы с R-деревом указатель на нашу новую структуру, так как в **rtree.h** в качестве рабочего формата данных используется простой указатель - **void \***. Исходный код всего проекта можно увидеть в приложении 1.

### РЕЗУЛЬТАТЫ РЕАЛИЗАЦИИ

Компиляция проекта:



Запуск проекта:



### ЗАКЛЮЧЕНИЕ

В ходе проведения работы была изучена и смоделирована структура данных «R-дерево» и тестовая программа для работы с ней. Подводя итоги, можно сказать, что главные свойства R-дерева следующие:

1. Оно состоит из внутренних узлов, листовых узлов и единственного корня
2. Корень R-дерева содержит указатель на самую большую область в пространстве
3. Родительские узлы содержат указатели на свои дочерние узлы, чья совокупная область (их сумма) покрывает область родительского узла
4. MBR — важнейший параметр, обозначающий минимальную ограничивающую область (рамку/прямоугольник), окружающую рассматриваемую область/объект в пространстве
5. Листовые узлы содержат данные об MBR объектов, на которые они ссылаются

Преимущества R-деревьев над B+деревьями заключаются в том, что для построения B+дерева необходимы данные, которые можно составлены в одном последовательном порядке. Это не всегда возможно, поскольку некоторые типы данных (например, географические координаты) не предполагают единого порядка, который можно было бы использовать для эффективного сканирования диапазона (например, всех точек в заданной области) по индексам, построенным с использованием B+Tree.

Обобщённо, преимущества R-деревьев заключаются в том, что эта структура данных эффективна для задач, включающих пространственную индексацию и поиск в двух или более измерениях. Примерами таких задач могут являться:

1. Поиск ближайшего соседа: поиск ближайшей точки к заданной точке в наборе данных.
2. Запросы диапазона: поиск всех точек в пределах заданного расстояния или площади от заданной точки.
3. Пространственное соединение: объединение двух наборов данных на основе их пространственной близости.
4. Кластеризация: группировка похожих точек на основе их пространственной близости.
5. Маршрутизация: поиск кратчайшего пути между двумя точками на карте.
6. Обработка изображений: Обнаружение и отслеживание объектов на изображениях на основе их пространственного положения.

В целом, любая задача, требующая быстрой и эффективной пространственной индексации и поиска, может выиграть от использования R-деревьев.

### СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн.*

Алгоритмы: построение и анализ, 3-е издание = Introduction to Algorithms, Third Edition.

— М.: [«Вильямс»](https://ru.wikipedia.org/w/index.php?title=Вильямс_(издательство)&action=edit&redlink=1), 2013. — 1328 с. — ISBN 978-5-8459-1794-2

1. *Курносов М.Г., Берлизов Д.М.* Алгоритмы и структуры обработки информации. – Новосибирск: Параллель, 2019. – 211 с. — ISBN 978-5-98901-230-5
2. *Гулаков В.К., Трубаков А.О., Трубаков Е.О.* Структуры и алгоритмы обработки многомерных данных: монография. - 2-е изд. - СПб., М., Краснодар: Лань, 2021. - 355 с.
3. R-tree // Wikipedia URL: https://en.wikipedia.org/wiki/R-tree (дата обращения: 23.05.2023).
4. *Guttmann A.* R-trees: A dynamic index structure for spatial searching // ACM SIGMOD. - 1984. - №14(2). - С. 47-57.
5. *Samet H.* The design and analysis of spatial data structures. - 2 изд. - Addison-Wesley Publishing Co., 1990
6. *Garcia-Molina, H., Salem, K.* R-trees: A dynamic index structure for spatial searching // IEEE Transactions on Knowledge and Data Engineering. - 1987. - №1(1). - С. 25-39.

### ПРИЛОЖЕНИЯ

**Приложение 1.** Исходный код проекта

1. Файл main.c

#include <stdio.h> #include <string.h> #include <math.h> #include "rtree.h"

struct city { char \*name;

double latitude; double longitude;

};

struct city nsk = { .name = "Novosibirsk", .latitude = 55.0333, .longitude = 82.9167 };

struct city bai = { .name = "Buenos Aires", .latitude =

-34.5997, .longitude = -58.3819 };

struct city rio = { .name = "Rio de Janeiro", .latitude =

-22.9111, .longitude = -43.2056 };

struct city tok = { .name = "Tokyo", .latitude = 35.6897,

.longitude = 139.6922 };

struct city tor = { .name = "Toronto", .latitude = 43.6532,

.longitude = -79.3832 };

struct city syd = { .name = "Sydney", .latitude =

-33.8688, .longitude = 151.2093 };

bool city\_iter(const double \*min, const double \*max, const void \*item, void \*udata) {

const struct city \*city = item; printf("%s\n", city->name); return true;

}

int main() {

struct rtree \*tr = rtree\_new();

rtree\_insert(tr, (double[2]){nsk.longitude, nsk.latitude}, NULL, &nsk);

rtree\_insert(tr, (double[2]){tor.longitude, tor.latitude}, NULL, &tor);

rtree\_insert(tr, (double[2]){bai.longitude, bai.latitude}, NULL, &bai);

rtree\_insert(tr, (double[2]){rio.longitude, rio.latitude}, NULL, &rio);

rtree\_insert(tr, (double[2]){tok.longitude, tok.latitude}, NULL, &tok);

rtree\_insert(tr, (double[2]){syd.longitude, syd.latitude}, NULL, &syd);

printf("\nNorthwestern cities:\n");

rtree\_search(tr, (double[2]){-180, 0}, (double[2]){0, 90}, city\_iter, NULL);

printf("\nNortheastern cities:\n");

rtree\_search(tr, (double[2]){0, 0}, (double[2]){180, 90}, city\_iter, NULL);

printf("\nSouthwestern cities:\n");

rtree\_search(tr, (double[2]){-180, -90}, (double[2]){0, 0}, city\_iter, NULL);

printf("\nSoutheastern cities:\n");

rtree\_search(tr, (double[2]){0, -90}, (double[2]){180, 0}, city\_iter, NULL);

rtree\_delete(tr, (double[2]){nsk.longitude, nsk.latitude}, NULL, &nsk);

printf("\nNortheastern cities after element deletion:\n");

rtree\_search(tr, (double[2]){0, 0}, (double[2]){180, 90}, city\_iter, NULL);

rtree\_free(tr);

}

1. Файл rtree.c

#include <stdlib.h> #include <stdio.h> #include <string.h> #include <stdbool.h> #include "rtree.h"

struct node \*node\_new(struct rtree \*tr, enum kind kind)

{

struct node \*node = (struct node

\*)tr->malloc(sizeof(struct node)); memset(node, 0, sizeof(struct node)); node->kind = kind;

return node;

}

void node\_free(struct rtree \*tr, struct node \*node) { if (node->kind == BRANCH) {

for (int i = 0; i < node->count; i++) {

node\_free(tr, node->children[i]);

}

}

tr->free(node);

}

void rect\_expand(struct rect \*rect, struct rect \*other) { for (int i = 0; i < DIMS; i++) {

if (other->min[i] < rect->min[i]) { rect->min[i] = other->min[i]; }

if (other->max[i] > rect->max[i]) { rect->max[i] = other->max[i]; }

}

}

double rect\_area(struct rect \*rect) {

double area = (double)(rect->max[0]) - (double)(rect->min[0]);

for (int i = 1; i < DIMS; i++) {

area \*= (double)(rect->max[i]) - (double)(rect->min[i]);

}

return area;

}

bool rect\_contains(struct rect \*rect, struct rect \*other) { for (int i = 0; i < DIMS; i++) {

if (other->min[i] < rect->min[i] || other->max[i] > rect->max[i]) {

return false;

}

}

return true;

}

bool rect\_intersects(struct rect \*rect, struct rect \*other) { for (int i = 0; i < DIMS; i++) {

if (other->min[i] > rect->max[i] || other->max[i] < rect->min[i]) {

return false;

}

}

return true;

}

bool nums\_equal(NUMTYPE a, NUMTYPE b) { return !(a < b || a > b);

}

bool rect\_onedge(struct rect \*rect, struct rect \*other) { for (int i = 0; i < DIMS; i++) {

if (nums\_equal(rect->min[i], other->min[i])) { return true;

}

if (nums\_equal(rect->max[i], other->max[i])) { return true;

}

}

return false;

}

bool rect\_equals(struct rect \*rect, struct rect \*other) { for (int i = 0; i < DIMS; i++) {

if (!nums\_equal(rect->min[i], other->min[i])) { return false;

}

if (!nums\_equal(rect->max[i], other->max[i])) { return false;

}

}

return true;

}

void node\_swap(struct node \*node, int i, int j) { struct rect tmp = node->rects[i];

node->rects[i] = node->rects[j]; node->rects[j] = tmp;

if (node->kind == LEAF) {

struct item tmp = node->items[i]; node->items[i] = node->items[j];

node->items[j] = tmp;

} else {

struct node \*tmp = node->children[i]; node->children[i] = node->children[j]; node->children[j] = tmp;

}

}

void node\_qsort(struct node \*node, int s, int e, int axis, bool rev, bool max) {

int nrects = e - s, left = 0, right = nrects - 1, pivot = nrects / 2;

if (nrects < 2) { return; } node\_swap(node, s + pivot, s + right); struct rect \*rects = &node->rects[s]; if (!rev) {

if (!max) {

for (int i = 0; i < nrects; i++) {

if (rects[i].min[axis] < rects[right].min[axis]) { node\_swap(node, s + i, s + left);

left++;

}

}

} else {

for (int i = 0; i < nrects; i++) {

if (rects[i].max[axis] < rects[right].max[axis])

{

node\_swap(node, s + i, s + left); left++;

}

}

}

} else {

if (!max) {

for (int i = 0; i < nrects; i++) {

if (rects[right].min[axis] < rects[i].min[axis]) { node\_swap(node, s + i, s + left);

left++;

}

}

} else {

for (int i = 0; i < nrects; i++) {

if (rects[right].max[axis] < rects[i].max[axis])

{

node\_swap(node, s + i, s + left); left++;

}

}

}

}

node\_swap(node, s + left, s + right); node\_qsort(node, s, s + left, axis, rev, max); node\_qsort(node, s + left + 1, e, axis, rev, max);

}

void node\_sort(struct node \*node) { node\_qsort(node, 0, node->count, 0, false, false);

}

void node\_sort\_by\_axis(struct node \*node, int axis, bool rev, bool max) {

node\_qsort(node, 0, node->count, axis, rev, max);

}

int rect\_largest\_axis(struct rect \*rect) { int axis = 0;

double nlength = (double)rect->max[0] - (double)rect->min[0];

for (int i = 1; i < DIMS; i++) {

double length = (double)rect->max[i] - (double)rect->min[i];

if (length > nlength) {

left->count-1, right);

}

}

node\_sort(right); node\_sort(left); return right;

}

node\_move\_rect\_at\_index\_into(left,

nlength = length; axis = i;

}

}

return axis;

}

void node\_move\_rect\_at\_index\_into(struct node \*from, int index, struct node \*into) {

into->rects[into->count] = from->rects[index]; from->rects[index] = from->rects[from->count - 1]; if (from->kind == LEAF) {

into->items[into->count] = from->items[index]; from->items[index] = from->items[from->count -

struct node \*node\_split(struct rtree \*tr, struct rect \*r, struct node \*left) {

return node\_split\_largest\_axis\_edge\_snap(tr, r, left);

}

int node\_rsearch(struct node \*node, NUMTYPE key) { for (int i = 0; i < node->count; i++) {

if (!(node->rects[i].min[0] < key)) { return i;

}

}

return node->count;

}

1];

} else {

from->children[index];

into->children[into->count] = from->children[index] =

double rect\_unioned\_area(struct rect \*rect, struct rect

\*other) {

double area = (double)MAX(rect->max[0], other->max[0]) - (double)MIN(rect->min[0], other->min[0]);

from->children[from->count - 1];

}

from->count--; into->count++;

}

struct node \*node\_split\_largest\_axis\_edge\_snap(struct rtree \*tr, struct rect \*rect, struct node \*left) {

int axis = rect\_largest\_axis(rect);

struct node \*right = node\_new(tr, left->kind); if (!right) return NULL;

for (int i = 0; i < left->count; i++) {

double min\_dist = (double)left->rects[i].min[axis] - (double)rect->min[axis];

double max\_dist = (double)rect->max[axis] - (double)left->rects[i].max[axis];

if (min\_dist < max\_dist) { // stay left

} else { // move right node\_move\_rect\_at\_index\_into(left, i, right); i--;

}

}

// make sure that both left and right nodes have at least min\_entries by moving items into underflowed nodes

if (left->count < MIN\_ENTRIES) { // reverse sort by min axis

node\_sort\_by\_axis(right, axis, true, false); while (left->count < 2) {

node\_move\_rect\_at\_index\_into(right, right->count-1, left);

}

} else if (right->count < 2) { // reverse sort by max axis

node\_sort\_by\_axis(left, axis, true, true); while (right->count < 2) {

for (int i = 1; i < DIMS; i++) {

area \*= (double)MAX(rect->max[i], other->max[i])

- (double)MIN(rect->min[i], other->min[i]);

}

return area; // returns the area of two rects expanded

}

int node\_choose\_least\_enlargement(struct node \*node, struct rect \*ir) {

int j = -1;

double jenlargement = 0, jarea = 0; for (int i = 0; i < node->count; i++) {

// calculate the enlarged area

double uarea = rect\_unioned\_area(&node->rects[i],

ir);

double area = rect\_area(&node->rects[i]); double enlargement = uarea - area;

if (j == -1 || enlargement < jenlargement || (!(enlargement > jenlargement) && area < jarea)) {

j = i;

jenlargement = enlargement; jarea = area;

}

}

return j;

}

int node\_choose\_subtree(struct node \*node, struct rect

\*ir) {

// take a quick look for the first node that contain the rect.

if (FAST\_CHOOSER == 1) {

int index = -1; double narea;

for (int i = 0; i < node->count; i++) {

if (rect\_contains(&node->rects[i], ir)) { double area = rect\_area(&node->rects[i]); if (index == -1 || area < narea) {

narea = area; index = i;

}

}

}

} else if (FAST\_CHOOSER == 2) {

for (int i = 0; i < node->count; i++) {

if (rect\_contains(&node->rects[i], ir)) { return i;

}

}

}

// fallback to using the choose-least-enlargment algorithm

return node\_choose\_least\_enlargement(node, ir);

}

struct rect node\_rect\_calc(struct node \*node) { struct rect rect = node->rects[0];

for (int i = 1; i < node->count; i++) { rect\_expand(&rect, &node->rects[i]);

}

return rect;

}

int node\_order\_to\_right(struct node \*node, int index) { while (index < node->count - 1 && node->rects[index

+ 1].min[0] < node->rects[index].min[0]) { node\_swap(node, index + 1, index); index++;

}

return index;

}

int node\_order\_to\_left(struct node \*node, int index) { while (index > 0 && node->rects[index].min[0] <

node->rects[index - 1].min[0]) { node\_swap(node,index, index - 1); index--;

}

return index;

}

// performs a copy of the data from args[1] & args[2], expects a rectangle (double[] double[])

// first N values are min corner, next N values - max corner, N - num of dimensions (max coords are optional) bool node\_insert(struct rtree \*tr, struct rect \*nr, struct node \*node, struct rect \*ir, struct item item, bool \*split, bool \*grown) {

\*split = false;

\*grown = false;

if (node->kind == LEAF) {

if (node->count == MAX\_ENTRIES) {

\*split = true; return true;

}

int index = node\_rsearch(node, ir->min[0]);

memmove(&node->rects[index + 1], &node->rects[index], (node->count-index) \* sizeof(struct rect));

memmove(&node->items[index + 1], &node->items[index], (node->count-index) \* sizeof(struct item));

node->rects[index] = \*ir; node->items[index] = item; node->count++;

\*grown = !rect\_contains(nr, ir); return true;

}

int index = node\_choose\_subtree(node, ir); // choose a subtree for inserting the rectangle

if (!node\_insert(tr, &node->rects[index], node->children[index], ir, item, split, grown)) {

return false;

}

if (\*split) {

if (node->count == MAX\_ENTRIES) { return true;

}

struct node \*left = node->children[index];

struct node \*right = node\_split(tr, &node->rects[index], left); // split child node

if (!right) { return false;

}

node->rects[index] = node\_rect\_calc(left); memmove(&node->rects[index + 2],

&node->rects[index + 1], (node->count - (index + 1)) \* sizeof(struct rect));

memmove(&node->children[index + 2], &node->children[index + 1], (node->count - (index + 1))

\* sizeof(struct node\*));

node->rects[index + 1] = node\_rect\_calc(right); node->children[index + 1] = right;

node->count++;

if (node->rects[index].min[0] > node->rects[index + 1].min[0]) {

node\_swap(node, index + 1, index);

}

index++; node\_order\_to\_right(node, index);

return node\_insert(tr, nr, node, ir, item, split,

grown);

}

if (\*grown) { // child rectangle must expand to accomadate new item

rect\_expand(&node->rects[index], ir); node\_order\_to\_left(node, index);

\*grown = !rect\_contains(nr, ir);

}

return true;

}

struct rtree \*rtree\_new\_with\_allocator(void

\*(\*cust\_malloc)(size\_t), void (\*cust\_free)(void\*)) { if (!cust\_malloc) cust\_malloc = malloc;

if (!cust\_free) cust\_free = free;

struct rtree \*tr = (struct rtree

\*)cust\_malloc(sizeof(struct rtree)); if (!tr) { return NULL; }

memset(tr, 0, sizeof(struct rtree)); tr->malloc = cust\_malloc;

tr->free = cust\_free; return tr;

}

struct rtree \*rtree\_new() { return rtree\_new\_with\_allocator(NULL, NULL); }

bool rtree\_insert(struct rtree \*tr, const NUMTYPE \*min, const NUMTYPE \*max, const DATATYPE data) {

struct rect rect;

memcpy(&rect.min[0], min, sizeof(NUMTYPE) \* DIMS);

memcpy(&rect.max[0], max ? max : min, sizeof(NUMTYPE) \* DIMS);

struct item item;

memcpy(&item.data, &data, sizeof(DATATYPE)); if (!tr->root) {

struct node \*new\_root = node\_new(tr, LEAF); if (!new\_root) return false;

tr->root = new\_root; tr->rect = rect;

}

bool split = false, grown = false;

if (!node\_insert(tr, &tr->rect, tr->root, &rect, item, &split, &grown)) { return false; }

if (split) {

struct node \*new\_root = node\_new(tr, BRANCH); if (!new\_root) return false;

struct node \*left = tr->root;

struct node \*right = node\_split(tr, &tr->rect, left); tr->root = new\_root;

tr->root->rects[0] = node\_rect\_calc(left); tr->root->rects[1] = node\_rect\_calc(right); tr->root->children[0] = left;

tr->root->children[1] = right; tr->root->count = 2;

tr->height++; node\_sort(tr->root);

return rtree\_insert(tr, min, max, data);

}

if (grown) {

rect\_expand(&tr->rect, &rect); node\_sort(tr->root);

}

tr->count++; return true;

}

void rtree\_free(struct rtree \*tr) {

if (tr->root) { node\_free(tr, tr->root); } tr->free(tr);

}

bool node\_search(struct node \*node, struct rect \*rect, bool (\*iter)(const NUMTYPE \*min, const NUMTYPE

\*max, const DATATYPE data, void \*udata), void \*udata)

{

if (node->kind == LEAF) {

for (int i = 0; i < node->count; i++) {

if (rect\_intersects(&node->rects[i], rect)) {

if (!iter(node->rects[i].min, node->rects[i].max, node->items[i].data, udata)) {

return false;

}

}

}

return true;

}

for (int i = 0; i < node->count; i++) {

if (rect\_intersects(&node->rects[i], rect)) {

if (!node\_search(node->children[i], rect, iter,

udata)) {

return false;

}

}

}

return true;

}

void rtree\_search(struct rtree \*tr, const NUMTYPE

\*min, const NUMTYPE \*max, bool (\*iter)(const NUMTYPE \*min, const NUMTYPE \*max, const DATATYPE data, void \*udata), void \*udata) {

struct rect rect;

memcpy(&rect.min[0], min, sizeof(NUMTYPE) \* DIMS);

memcpy(&rect.max[0], max ? max : min, sizeof(NUMTYPE) \* DIMS);

if (tr->root && rect\_intersects(&tr->rect, &rect)) { node\_search(tr->root, &rect, iter, udata);

}

}

size\_t rtree\_count(struct rtree \*tr) { return tr->count; }

void node\_delete(struct rtree \*tr, struct rect \*nr, struct node \*node, struct rect \*ir, struct item item, bool

\*removed, bool \*shrunk, int (\*compare)(const DATATYPE a, const DATATYPE b, void \*udata), void

\*udata) {

\*removed = false;

\*shrunk = false;

if (node->kind == LEAF) {

for (int i = 0; i < node->count; i++) {

if (!rect\_contains(ir, &node->rects[i])) { continue;

}

int cmp = compare ?

compare(node->items[i].data, item.data, udata)

:

memcmp(&node->items[i].data, &item.data, sizeof(DATATYPE));

if (cmp != 0) { continue;

}

// found the target item to delete memmove(&node->rects[i], &node->rects[i + 1],

(node->count - (i + 1)) \* sizeof(struct rect)); memmove(&node->items[i], &node->items[i +

1], (node->count - (i + 1)) \* sizeof(struct item)); node->count--;

if (rect\_onedge(ir, nr)) { // item was on the edge of node rect

node rect

\*nr = node\_rect\_calc(node); // recalculation of

\*shrunk = true; // notify the caller that

return;

}

rect is shrunk

}

\*removed = true; return;

}

return;

}

for (int i = 0; i < node->count; i++) {

if (!rect\_contains(&node->rects[i], ir)) { continue;

}

struct rect crect = node->rects[i];

node\_delete(tr, &node->rects[i], node->children[i], ir, item, removed, shrunk, compare, udata);

if (!\*removed) { continue;

}

if (node->children[i]->count == 0) { // underflow node\_free(tr, node->children[i]); memmove(&node->rects[i], &node->rects[i + 1],

(node->count - (i + 1)) \* sizeof(struct rect));

memmove(&node->children[i], &node->children[i + 1], (node->count - (i + 1)) \* sizeof(struct node \*));

node->count--;

\*nr = node\_rect\_calc(node);

\*shrunk = true; return;

}

if (\*shrunk) {

\*shrunk = !rect\_equals(&node->rects[i], &crect); if (\*shrunk) {

\*nr = node\_rect\_calc(node);

}

node\_order\_to\_right(node, i);

}

return;

// search the tree for an item contained within provided rect, perform a binary comparison of its data to provided, first item found is deleted

void rtree\_delete(struct rtree \*tr, const NUMTYPE \*min, const NUMTYPE \*max, const DATATYPE data) {

struct rect rect;

memcpy(&rect.min[0], min, sizeof(NUMTYPE) \* DIMS);

memcpy(&rect.max[0], max ? max : min, sizeof(NUMTYPE) \* DIMS);

struct item item;

memcpy(&item.data, &data, sizeof(DATATYPE)); if (!tr->root) { return; }

bool removed = false, shrunk = false;

node\_delete(tr, &tr->rect, tr->root, &rect, item, &removed, &shrunk, NULL, NULL);

if (!removed) { return;

}

tr->count--;

if (tr->count == 0) { node\_free(tr, tr->root); tr->root = NULL;

memset(&tr->rect, 0, sizeof(struct rect));

} else {

while (tr->root->kind == BRANCH && tr->root->count == 1) {

struct node \*prev = tr->root;

tr->root = tr->root->children[0]; prev->count = 0;

node\_free(tr, prev);

}

if (shrunk) {

tr->rect = node\_rect\_calc(tr->root);

}

}

}

}

1. Файл rtree.h

#pragma once

#include <stdlib.h> #include <stdbool.h>

#define DATATYPE void \* #define NUMTYPE double #define DIMS 2

#define MAX\_ENTRIES 64

#define MIN\_ENTRIES\_PERCENTAGE 10

#define FAST\_CHOOSER 2 // 0 - off , 1 - fast, 2 - faster #define panic(\_msg\_) { \

fprintf(stderr, "panic: %s (%s:%d)\n", (\_msg\_),

FILE , LINE ); \ exit(1); \

}

#define MIN(a,b) ((a) < (b) ? (a) : (b))

#define MAX(a,b) ((a) > (b) ? (a) : (b))

#define MIN\_ENTRIES ((MAX\_ENTRIES) \* (MIN\_ENTRIES\_PERCENTAGE) / 100 + 1)

enum kind { LEAF = 1,

BRANCH = 2,

};

struct rect {

NUMTYPE min[DIMS]; NUMTYPE max[DIMS];

};

struct item { DATATYPE data;

};

struct node {

enum kind kind; // LEAF or BRANCH int count; // number of rects

struct rect rects[MAX\_ENTRIES];

union { struct node \*children[MAX\_ENTRIES]; struct item items[MAX\_ENTRIES]; };

};

struct rtree { size\_t count; int height; struct rect rect;

struct node \*root;

void \*(\*malloc)(size\_t); void (\*free)(void \*);

};

struct rtree \*rtree\_new();

bool rtree\_insert(struct rtree \*tr, const double \*min, const double \*max, const void \*data);

void rtree\_free(struct rtree \*tr);

void rtree\_search(struct rtree \*tr, const double \*min, const double \*max, bool (\*iter)(const double \*min, const double \*max, const void \*data, void \*udata), void

\*udata);

size\_t rtree\_count(struct rtree \*tr);

void rtree\_delete(struct rtree \*tr, const double \*min, const double \*max, const void \*data);

1. Makefile

all: main

.PHONY: main main: main.c rtree.c

gcc -Wall -Wextra -o $@ $^

.PHONY: clean clean:

rm -rf main

**Отзыв о работе студента** Григорьев Юрий Вадимович (ФИО студента)

# Уровень освоения компетенций

Киреев Илья Анатольевич

(ФИО студента)

|  |  |
| --- | --- |
| Компетенции | Уровень сформированности компетенций |
| *ОПК-1 - Способен применять фундаментальные знания, полученные в области математических и (или) естественных наук, и использовать их в профессиональной деятельности* |  |

## отметка о зачете

Руководитель практики от СибГУТИ:

Должность руководителя подпись ФИО руководителя

## " " 20 г.