Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»
(СибГУТИ)

02.03.02 Фундаментальная информатика
и информационные технологии
Профиль: Системное программное
обеспечение
(очная форма обучения)

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

в/на Институте Информатики и Вычислительной Техники
(наименование профильной организации/структурного подразделения СибГУТИ)

СТРУКТУРА ДАННЫХ «R-ДЕРЕВО»

Выполнил: Григорьев Ю.В.
студент института ИВТ
гр. ИС-142 _____/_____/
«27» мая 2023 г. (подпись)

Проверил:
Руководитель от СибГУТИ _____/_____/
«27» мая 2023 г. (подпись)

Новосибирск 2023

План-график проведения учебной **практики**

Вид практики

Григорьев Юрий Вадимович

Фамилия Имя Отчество студента

института Информатика и вычислительная техника, 2 курса, гр. ИС-142

Направление: 02.03.02 Фундаментальная информатика и информационные технологии

Код – Наименование направления (специальности)

Профиль: Системное программное обеспечение

Место прохождения практики _____

Объем практики: **108/3** часов/ЗЕ

Вид практики ***учебная***

Тип практики ***научно-исследовательская работа (получение первичных навыков научно-исследовательской работы)***

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
4. Выполнение работ в соответствии с составленным планом: 1. Разработка заголовочного файла с прототипами методов работы со структурой данных 2. Разработка Makefile для автоматизации компиляции программы с библиотекой структуры данных 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 “R-tree”.

ВВЕДЕНИЕ

R-дерево (R-Tree) - это индексная структура для доступа к пространственным данным, предложенная Антонином Гуттманом (Калифорнийский университет, Беркли) в 1984 году. R-дерево допускает произвольное выполнение операций добавления, удаления и поиска данных без периодической переиндексации. При этом дерево получается сбалансированным, что является одним из важных свойств любой иерархической структуры данных.

СТРУКТУРА R-ДЕРЕВА

R-дерево – это сбалансированное по высоте дерево, сходное с B+-деревом, листовые узлы которого содержат ссылки на конечные объекты. Если индексная структура находится на жестком диске, то каждый узел соответствует дисковой странице. Структура разработана так, чтобы для пространственного поиска требовалось посещение как можно меньшего числа узлов. Индексная структура полностью динамическая – добавление и удаление может выполняться одновременно с поиском, и никакой периодической реорганизации структуры производить не нужно. Для организации такой индексной структуры используют пространственную базу данных, состоящую из набора записей, каждой из которых соответствует некоторый уникальный идентификатор. Этот идентификатор используют как средство ссылки на запись из индекса. В качестве идентификатора может выступать некоторое уникальное число или номер записи в файле (второй вариант предпочтительнее, так как работает быстрее, однако для него присущи некоторые недостатки, связанные с удалением записей из файла).

Если принять описанные условия, то каждый листовой узел дерева будет состоять из элементов, имеющих вид:

[MBR, указатель_на_объект],

где *указатель_на_объект* ссылается на объект в памяти устройства, а *MBR* – это *n*-мерный прямоугольник, который является минимальным охватывающим прямоугольником для пространственного объекта, со сторонами параллельными осям координат. Обычно *MBR* задают в виде интервала размерности *n* с закрытыми концами *[a,b]*, где *n* - число размерностей (измерений). Внутренние узлы дерева содержат элементы, имеющие похожую структуру:

[MBR, ссылка_на_потомка],

где *ссылка_на_потомка* – это адрес узла низшего уровня в *R*-дереве (дочернего по отношению к данному), все записи внутри которого покрываются прямоугольником *MBR*.

И листовые узлы, и внутренние представляют собой набор из элементов описанной структуры, и даже в простейшей реализации таких элементов должно быть больше одного. Обозначим *M* как максимальное число элементов в любом узле, а *m* – минимальное. Для реализации основных алгоритмов необходимо выполнение условия $m \leq M / 2$.

R-дерево должно удовлетворять следующим требованиям:

1. Каждый узел дерева содержит не меньше *m* и не больше *M* записей. Исключение может составлять только корень.
2. Корень, если он не является листом, содержит как минимум двух потомков. Максимальное количество элементов в корне также ограничивается значением *M*.

3. Для каждой индексной записи листового узла *MBR* является минимальным прямоугольником, который полностью вмещает в себя пространственный объект, на который ссылается запись.

4. Для каждой индексной записи внутреннего узла дерева *MBR* является минимальным прямоугольником, охватывающим все *MBR* дочерних узлов.

5. Все листовые узлы дерева расположены на одном уровне (дерево является сбалансированным).

6. Каждый объект упоминается в дереве ровно один раз.

На рис. 1 показан пример структуры *R*-дерева и проиллюстрированы отношения ограничения и перекрытия, которые могут существовать между его прямоугольниками.

Имея представление о свойствах R-дерева, можно оценить его высоту при числе элементов *N*. Из свойств, описанных выше, следует, что каждый узел дерева содержит как минимум *m* потомков. Поэтому наибольшая высота *R*-дерева, содержащего *N* индексных записей, будет не больше $\lfloor log_m N \rfloor$ - 1. При этом максимальное число узлов в таком дереве будет равно $\lfloor N/m \rfloor + \lfloor N/m^2 \rfloor + \ldots + 1$.

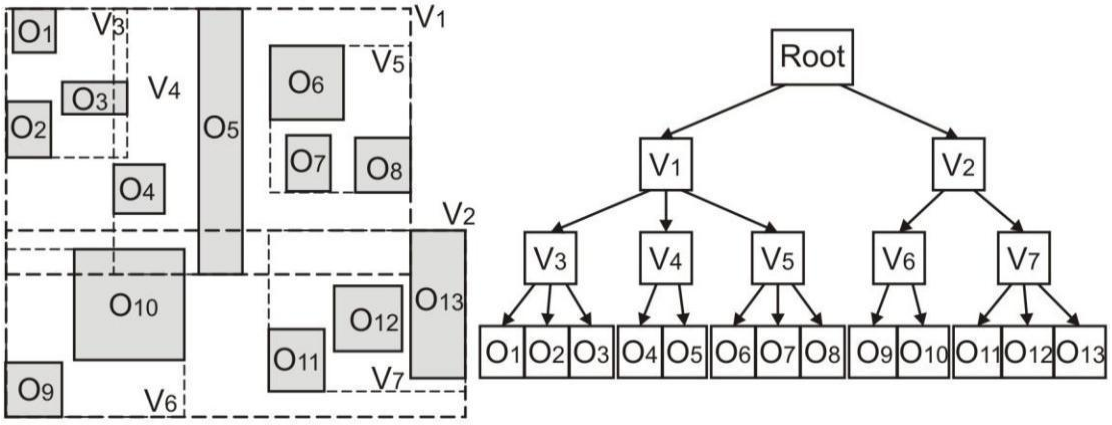

Рис. 1. Пример *R*-дерева

В худшем случае использование пространства памяти, в которой хранится индексная структура, будет *m / M*. Однако алгоритмы построения дерева разработаны таким образом, что структура будет стремиться содержать более *m* записей в узле. Это уменьшает высоту дерева и увеличивает полезное использование памяти.

АЛГОРИТМ РАБОТЫ

Поиск элемента

Алгоритм поиска в *R*-дереве похож на алгоритм поиска по *B*-дереву: он начинается в корне и опускается по нему к листовому узлу, выбирая в зависимости от заданных параметров поиска то или иное поддерево. Главное отличие состоит в том, что возможен вариант, при котором более одного поддерева текущего узла участвует в поиске: такая ситуация связана с применением метода размещения многомерных объектов, разрешающего пересекаться ограничивающим областям разных элементов. Это может привести к многократному уменьшению скорости поиска, однако алгоритмы построения и изменения дерева стараются поддерживать дерево в наиболее оптимальном виде. Пример рекурсивной процедуры поиска объектов, имеющих хотя бы одну общую точку с областью поиска *S,* в псевдокоде:

```
rtree_search (V, S, Res) // V - текущ. вершина, S - область поиска, Res - результаты поиска
   if V.kind != LEAF then
      for each V’ in V // цикл по всем записям V’ в узле V
         if V’.MBR * S != 0 then // MBR записи V’ пересекается с S
            rtree_search (V’, S, Res)
         end if
   end if
   if V.kind == LEAF then
      for each O in V
         if O.MBR * S != 0 then // MBR записи O пересекается с S
            Res += O
         end if
      end for
   end if
end
```

Рассмотрим описанный алгоритм на примере, показанном на рис. 2. Область поиска соответствует заданному прямоугольнику *ABCD*. Первоначально процедура поиска вызывается для корня. Так как корень является внутренней вершиной, то для него выполняется первая ветка алгоритма поиска. Она проверяет узлы V_1 и V_2 на пересечение с заданной областью. Как нетрудно заметить, оба этих узла имеют общие точки с областью поиска, и поэтому для обоих из этих узлов рекурсивно вызывается процедура **rtree_search**.

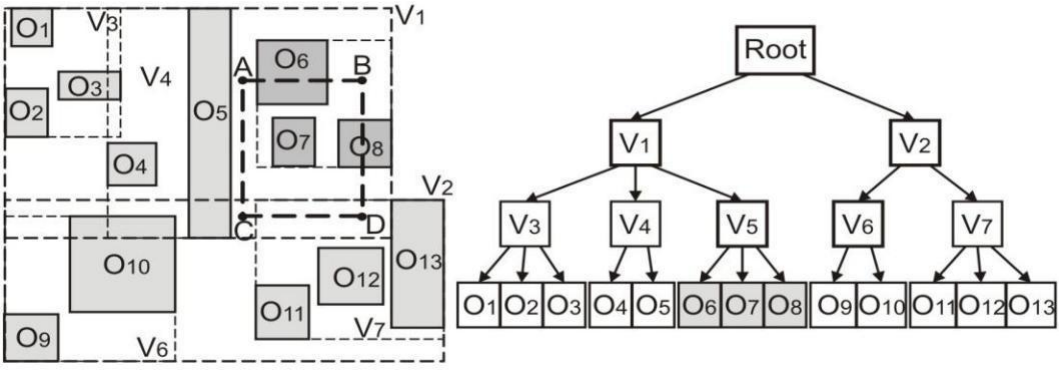

Рис. 2. Пример поиска в *R*-дереве

rtree_search для вершины V_1 перебирает элементы V_3,V_4, V_5, причем только V_5 имеет пресечение с прямоугольником *ABCD*. Поэтому вершины V_3 и V_4 пропускаются и далее не рассматриваются. Дальнейший вызов процедуры для вершины V_5 выдаст в качестве результата

три элемента – O_6, O_7, O_8, которые и будут добавлены в множество результата *Res*. Аналогичным образом будет просмотрена ветка V_2. Из ее потомков только V_7 имеет общие точки с *ABCD*. Однако не один из элементов V_7 не пересекается с областью поиска. Данная ветка поиска оказалась ложной. В результате поиска получаем список элементов, удовлетворяющих заданному запросу:

$$Res = \{O_6, O_7, O_8\}.$$

Вставка элемента

Добавление нового объекта в *R*-дерево похоже на процедуру вставки в *B*+-дерево. Новая индексная запись добавляется в листовой узел, если узел переполняется, то происходит его деление, в результате которого у предка появляется еще один потомок. Если предок также оказывается переполненным, то и он делится дальше и так далее. Таким образом, вставка одного объекта может повлиять на структуру дерева в целом. Процедура вставки объекта представлена в следующем псевдокоде:

```
rtree_insert (O) // O - вставляемый объект
   L = choose_leaf (O)
   if L.count < M then
      L += O
      L'' = NULL
   else
      L'' = node_split (L, O)
   end if
   rtree_correct (L, L'')
end
```

Во-первых, процедура ищет листовой узел, в который необходимо поместить данный объект (шаг 1). Процедура поиска такого листа является важным шагом, так как неправильно выбранная позиция может привести к неэффективности структуры в целом. После того, как узел для вставки выбран, в нем размещается вставляемый объект (шаг 2). Если в листовом узле *L* есть место для новой записи, объект *O* помещается в него и процедура заканчивает свою работу. В противном случае, если узел *L* уже содержит максимально возможное число записей, то происходит деление узла на два новых *L* и *L''*, которые содержат старые записи узла *L* и добавляемый объект *O*. После вставки объекта в дерево и возможного расщепления узла необходимо корректировать дерево (шаг 3). Эта процедура включает расширение границ *MBR* для текущего узла и всех его предков. Также эта процедура распространяется вверх по дереву при необходимости.

Рассмотрим алгоритмы упомянутых процедур подробнее в псевдокоде:

```
choose_leaf (O) // O - вставляемый объект
   V = root
   for each V' in V
      if V.kind == LEAF then
         return V
      end if
      V_new = потомок, для которого MBR(V',O)-MBR(V') - минимальный
      V = V_new
   end for
end
```

```
rtree_correct (L, L'') // L - корректируемая вершина, L'' - вершина от деления L
V=L, V''=L''
```

```
while V != tree->root then
   P = V.parent
   P_V = запись в узле P о потомке V
   rtree_correct (P_V)
   if V'' ≠ NULL then
      P_V'' = новая запись о узле V''
      if P->count < M then
         P += P_V''
      else
         P'' = node_split (P, P_V'')
      end if
   end if
   V = P, V''=P''
   if V'' ≠ NULL then
         root = tree->root
         root.data[] = V, V''
   end if
   return
end while
end
```

Как было отмечено, процедура корректировки изменяет *MBR* всех вершин дерева, которые расположены выше листа с вставленным объектом. Второй и не менее важной функций процедуры корректировки является распространение деления вершин вверх по дереву, в случае, если будет происходить переполнение на внутренних узлах дерева. В качестве параметров в процедуру передаются два новых узла, которые получились при вставке объекта в дерево. Если разбиение не произошло, то первым параметром передается старый узел, а второй параметр приравнивается в *NULL*. На первом шаге процедура заносит переданные параметры в переменные *V* и *V″*. Эти переменные будут отвечать за текущие вершины в дереве, которые необходимо исправить. После этого происходит сравнение вершины *V* корня дерева. Если данная вершина является корнем, то это означает, что изменения уже распространились до верха дерева и необходимо просто завершить процедуру корректировки. Однако стоит учитывать один момент: если после предыдущих манипуляций произошло расщепление корня на два узла (переменная *V″≠NULL*), то необходимо создать новый корень дерева, узлами-потомками которого будут *V* и *V″*. Если предыдущий пункт не выполнен, то происходит корректировка. Для этого определяется предок узла *V*, а также запись в нем об этом узле. После этого *MBR* найденной записи изменяется таким образом, чтобы включать в себя все *MBR* дочерних элементов узла *V*, но при этом не содержать лишних областей. Четвертый шаг алгоритма выполняется только в том случае, если предыдущие действия вызвали деление узла. В этом случае у нас в переменной *V″* будет находиться вершина с элементами, которые пока еще не помещены в дерево. Для этой вершины необходимо создать запись $P_{V''}$, которая будет содержать минимальный описывающий прямоугольник для данной вершины и ссылку на саму вершину. Эту запись и нужно разместить в предке узла *V*. Однако при помещении в узел *P* записи $P_{V''}$ необходимо помнить, что данная операция может привести к переполнению и тогда придется разбивать узел *P* на два новых. После всех описанных операций в переменные *V* и *V″* заносятся новые значения *P* и *P″* соответственно, и алгоритм повторяется заново с шага 2.

Удаление элемента

Для того, чтобы структуру можно было считать динамической, необходима поддержка удаления уже существующих в дереве элементов, которая также должна корректировать дерево для сохранения его свойств.

```
rtree_delete (O) // O - удаляемый объект
   V = root
```

```
   L = rtree_search (V,O)
   if L == NULL then
      return
   end if
   delete O
   V = L, Q = NULL
   if1 V == tree->root then goto if2 end if1
   P = V->Parent
   P_V  = запись в узле P о потомке V
   if V->count < m then
      delete P_V
      Q = V
      delete V
   else
      rtree_correct (V)
   end if
   V = P
   goto if1
   if2 root->children->count == 1 then
      delete tree->root
      tree->root = root->child
   end if2
   tree->insert(Q[0], Q[1], …)
end
```

Первое, что производит процедура удаления объекта *O* из *R*-дерева, это ищет листовой узел, в котором находится данный объект. Для этого используется процедура поиска rtree_search. В качестве параметров ей передается вершина, с которой нужно начать поиск (в нашем случае это корень) и объект поиска. Если объект не найден, то будет возвращён *NULL*. При этом необходимо завершить и процедуру удаления. На втором шаге удаляется объект *O* из узла *L* и подготавливаются временные переменные для коррекции дерева. В переменную *V* (текущая вершина для коррекции) заносится листовой узел *L*, а в переменную *Q* – пустое множество (это множество вершин, которые необходимо потом вставить в дерево заново). Далее необходимо проверить, является ли вершина *V* корнем. Если *V* – корневая вершина, то шаги 4–6 нужно пропустить и перейти сразу к седьмому пункту алгоритма. Иначе – находим предка для вершины *V*, и в нем определяем запись, ссылающуюся на *V* (P_V). Если в рассматриваемом узле число записей меньше минимально возможного (*m*), то необходимо удалить этот узел из дерева. При этом все элементы из *V* помещаются в множество *Q* (чтобы потом снова быть размещенными в дереве, но в других вершинах) и из вершины *P* удаляется элемент, ссылающийся на удаленную вершину (удаляется P_V). Если же записей в вершине *V* больше, чем заданный параметр *m*, то удалять вершину не нужно. При этом необходимо просто скорректировать *MBR* узла таким образом, чтобы он охватывал все прямоугольники дочерних узлов, но при этом не включал лишнего пространства (после удаления узлов вполне вероятно можно будет сузить *MBR*, который хранится в записи P_V). После проделанных операций необходимо распространить сделанные изменения вверх по дереву (скорректировать *MBR* узла предка или, возможно, даже удалить его, если он оказался не заполненным до предела *m*). Для этого в переменную *V* заносится предок текущей вершины и повторяется алгоритм с шага 3. После того, как все изменения дойдут до корня, алгоритм продолжится с шага 7. Исходя из свойств *R*-дерева, описанных в начале данного параграфа, корень должен иметь не меньше двух потомков. Поэтому необходимо просто проверить число дочерних узлов у корня и при нахождении там всего одного потомка сделать его новым корнем дерева. Последнее, что необходимо выполнить в процедуре удаления, это вставить временно удаленные узлы из множества *Q* обратно в дерево. Данная процедура выполняется полностью аналогично описанной ранее процедуре ВСТАВКА за одним лишь исключением: вершины из множества *Q*

необходимо разместить на тех же уровнях, на которых они были до процедуры удаления. Этого требования необходимо придерживаться для того, чтобы не нарушить сбалансированность дерева (одно из свойств *R*-дерева заключается в том, что все листовые узлы находятся в нем на одном уровне).

Разбиение узла

Изменение данных прикладной задачи требует частого изменения индексной структуры. Для добавления новой записи в уже заполненный узел *R*-дерева, содержащий *M* записей, необходимо распределить *M* + 1 элемент между двумя узлами. Процедура разбиения узла может быть вызвана не только при добавлении новых элементов в индекс, но и при перестройке дерева, при удалении ненужной записи, при обновлении данных или даже при его корректировке. Алгоритм, выполняющий деление узла, особенно важен, так как плохое разбиение может сильно затруднить операции поиска по дереву. Разбиение узла без учета критериев оптимальности построения дерева приводит к увеличению времени работы процедуры поиска конкретного объекта, а следовательно, к ухудшению работы индексной структуры в целом. При плохом разбиении узлы дерева разрастаются вдоль осей координат и захватывают много пространства, не содержащего ни одного объекта. Такой пример показан на рис. 3. С одной стороны вариант (а) обеспечивает нулевое перекрытие двух узлов дерева. Однако суммарная площадь этих узлов будет значительно больше самих узлов, что вызовет многократное ложное срабатывание процедуры поиска. При большой площади пространства, соответствующей узлам дерева, запросу поиска на промежуточных стадиях работы может удовлетворять большое число записей (более одной), хотя, в конечном счете, на каждом уровне интересует только одна. Следовательно, алгоритм будет ветвиться и обходить дерево неоптимальным путем, включая обход ненужных узлов, что может сильно отразиться на скорости работы индексной структуры. Кроме того, обход ненужных узлов потребует дополнительного расхода оперативной памяти. В условиях большого числа запросов это обстоятельство также может стать критичным.

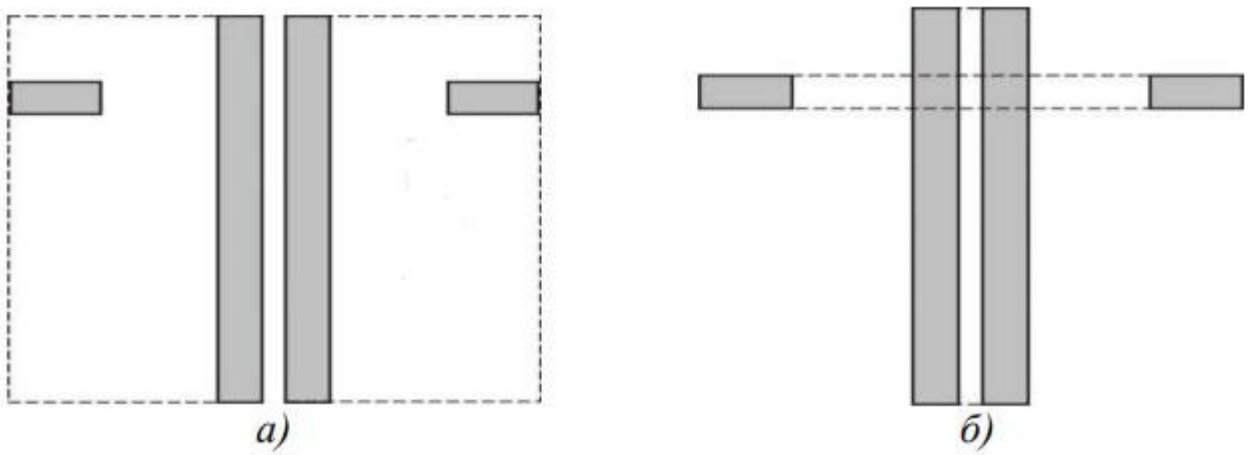

Рис. 3. Пример возможного разбиения узла на два новых:
а – “плохое” разбиение; б – “хорошее” разбиение с пересечением

Исчерпывающий перебор - самый простой из алгоритмов, при котором получается максимально оптимальное дерево: для нахождения минимальной площади покрытия и всех остальных параметров оптимальности деления необходимо произвести все возможные разбиения M + 1 записи на две группы и выбрать наилучшее. Задача разбиения в общем случае является NP-полной. Этот вариант применим без потери производительности при M не более 5, и именно поэтому данный подход применяется редко, а во всех вариантах построения R-деревьев используются эвристические подходы, дающие на реальных данных не всегда оптимальные разбиения (но практически всегда достаточно эффективные для решения конкретной задачи).

Квадратичный алгоритм разбиения был предложен основателем R-деревьев Антонином Гуттманом. В нём осуществлена попытка найти такое деление, при котором площадь охватывающих прямоугольников будет минимальна. Однако при этом не гарантируется, что это будет действительно наилучший вариант. Алгоритмическая сложность изменяется по квадратичному закону относительно M и по линейному – относительно числа измерений.

```
node_split (L,O) // O - вставляемый объект, L - разбиваемая вершина
   Q = L->children + O
   delete (L->children)
   L'' = NULL
   O1,O2 = choose_first (Q)
   L += O1, L'' += O2
   n = Q->size, n1 = L->size, n2 = L''->size
   if n == 0 then
         return L''
   end if
   if m - n1 ≥ n then
         L = Q
         return L''
   end if
   if m - n2 ≥ n then
         L'' = Q
         return L''
   end if
   O' = choose_next (L,L'',Q)
   d1 = MBR(L,O′) - MBR(L)
   d2 = MBR(L'',O′) - MBR(L'')
   if (d1<d2) or (d1 == d2 && n1 < n2) then
      L += O'
   else
      L'' += O'
      node_split()
   end if
end
```

Алгоритм начинается с подготовительных операций (шаг 1). Все элементы, которые нужно будет распределить между двумя новыми вершинами, переносятся в множество Q. Создается еще одна пустая вершина L. Таким образом мы получаем две пустые вершины (L и L) и множество элементов Q, которые необходимо распределить по этим вершинам. Затем алгоритм выбирает пару элементов, которые покрывают наибольшую площадь (соответствующая процедура представлена в листинге 3.8). Для этого рассчитывается коэффициент A = MBR(O1,O2) – MBR(O1) – MBR(O2), где MBR(O1,O2) – площадь прямоугольника, охватывающего обе записи, а MBR(O1) и MBR(O2) – площади прямоугольников соответствующих объектов. Этот коэффициент показывает неэффективность объединения двух данных объектов в одну группу. Элементы, на которых достигается максимум коэффициента A, становятся первыми элементами двух будущих групп.

```
choose_first (L,L'',Q) //  Q - нераспред. элементы, L, L'' - вершины между которыми распред.
   for each O1,O2 in Q
      A = MBR(O1,O2) – MBR(O1) – MBR(O2)
   end for
   O1,O2 = max(A)->nodes
```

```
   Q -= (O1 + O2)
   return O1, O2
end
```

Оставшиеся записи распределяются в группы по одной (шаг 4 алгоритма **node_split**). Для этого вызывается процедура **choose_next**, на которую и возложена задача выбора элемента из множества Q, который будет распределен следующим. После выбора элемента для вставки он добавляется в вершину, ограничивающий прямоугольник которой потребует минимального увеличения площади (если d1 и d2 – увеличение площади ограничивающих прямоугольников при добавлении элемента O′ в вершины L и L соответственно, то вставку нужно произвести в ту вершину, для которой d меньше). При равенстве увеличения площадей (d1 = d2), для вставки выбирается та вершина, в которой меньше число записей. Описанное действие продолжается до тех пор, пока не будут выбраны все элементы из множества Q. Однако на каждом необходимо проверять выполнимость условия минимального наполнения узла (число элементов в любой вершине дерева, кроме корня, должно быть не меньше m). Если на каком-то шаге окажется, что для выполнения этого условия необходимо все оставшиеся в Q элементы переместить в одну из вершин, то необходимо сделать это и завершить процедуру деления узла (шаг 3).

```
choose_next (L,L'',Q) //  Q - нераспред. элементы, L,L'' - вершины между которыми распред.
   for each O' in Q
         d1 = MBR(L,O′) - MBR(L)
         d2 = MBR(L'',O′) - MBR(L'')
         a = |d1-d2|
   end for
   O = max(a)->node
   Q -= O
   return O
end
```

Представленный алгоритм достаточно прост. Для каждого нераспределенного элемента рассчитывается площадь охватывающего прямоугольника, который получится после присоединения этого элемента к каждой группе (значения d1 и d2). Элемент с наибольшей разницей площадей обеих групп выбирается как следующий элемент. Существует несколько другая стратегия, предложенная позже: выбирается элемент, присоединение которого в какую либо группу минимально увеличивает площадь MBR группы.

```
choose_next (L,L'',Q) // Q - нераспред. элементы, L,L'' - вершины между которыми распред.
    for each O' in Q
         d1 = MBR(L,O′) - MBR(L)
         d2 = MBR(L'',O′) - MBR(L'')
         d = min{d1,d2}
   end for
   O = min(d)->node
   Q -= O
   return O
end
```

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 О 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:

```
▶ Run make

gcc -Wall -Wextra -o main main.c rtree.c
```

Запуск проекта:

```
▶ Run ./main

Northwestern cities:
Toronto

Northeastern cities:
Novosibirsk
Tokyo

Southwestern cities:
Buenos Aires
Rio de Janeiro

Southeastern cities:
Sydney

Northeastern cities after element deletion:
Tokyo
```

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», 2013. — 1328 с. — ISBN 978-5-8459-1794-2
2. *Курносов М.Г., Берлизов Д.М.* Алгоритмы и структуры обработки информации. – Новосибирск: Параллель, 2019. – 211 с. — ISBN 978-5-98901-230-5
3. *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.
4. R-tree // Wikipedia URL: https://en.wikipedia.org/wiki/R-tree (дата обращения: 23.05.2023).
5. *Guttmann A.* R-trees: A dynamic index structure for spatial searching // ACM SIGMOD. - 1984. - №14(2). - С. 47-57.
6. *Samet H.* The design and analysis of spatial data structures. - 2 изд. - Addison-Wesley Publishing Co., 1990
7. *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

```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);
}
```

2. Файл rtree.c

```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) {
```

```
58          return !(a < b || a > b);
59      }
60
61      bool rect_onedge(struct rect *rect, struct rect *other) {
62          for (int i = 0; i < DIMS; i++) {
63              if (nums_equal(rect->min[i], other->min[i])) {
64                  return true;
65              }
66              if (nums_equal(rect->max[i], other->max[i])) {
67                  return true;
68              }
69          }
70          return false;
71      }
72
73      bool rect_equals(struct rect *rect, struct rect *other) {
74          for (int i = 0; i < DIMS; i++) {
75              if (!nums_equal(rect->min[i], other->min[i])) {
76                  return false;
77              }
78              if (!nums_equal(rect->max[i], other->max[i])) {
79                  return false;
80              }
81          }
82          return true;
83      }
84
85      void node_swap(struct node *node, int i, int j) {
86          struct rect tmp = node->rects[i];
87          node->rects[i] = node->rects[j];
88          node->rects[j] = tmp;
89          if (node->kind == LEAF) {
90              struct item tmp = node->items[i];
91              node->items[i] = node->items[j];
92              node->items[j] = tmp;
93          } else {
94              struct node *tmp = node->children[i];
95              node->children[i] = node->children[j];
96              node->children[j] = tmp;
97          }
98      }
99
100     void node_qsort(struct node *node, int s, int e, int axis, bool rev, bool max) {
101         int nrects = e - s, left = 0, right = nrects - 1, pivot = nrects / 2;
102         if (nrects < 2) { return; }
103         node_swap(node, s + pivot, s + right);
104         struct rect *rects = &node->rects[s];
105         if (!rev) {
106             if (!max) {
107                 for (int i = 0; i < nrects; i++) {
108                     if (rects[i].min[axis] < rects[right].min[axis]) {
109                         node_swap(node, s + i, s + left);
110                         left++;
111                     }
112                 }
113             } else {
114                 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) {
            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 - 1];
    } else {
        into->children[into->count] = from->children[index];
```

```
172             from->children[index] = from->children[from->count - 1];
173         }
174         from->count--;
175         into->count++;
176     }
177
178     struct node *node_split_largest_axis_edge_snap
    (struct rtree *tr, struct rect *rect, struct node *left) {
179         int axis = rect_largest_axis(rect);
180         struct node *right = node_new(tr, left->kind);
181         if (!right) return NULL;
182         for (int i = 0; i < left->count; i++) {
183             double min_dist = (double)left->rects[i].min[axis] - (double)rect->min[axis];
184             double max_dist = (double)rect->max[axis] - (double)left->rects[i].max[axis];
185             if (min_dist < max_dist) { // stay left
186             } else {                   // move right
187                 node_move_rect_at_index_into(left, i, right);
188                 i--;
189             }
190         }
191         // make sure that both left and right nodes have at least min_entries by moving items into
    underflowed nodes
192         if (left->count < MIN_ENTRIES) { // reverse sort by min axis
193             node_sort_by_axis(right, axis, true, false);
194             while (left->count < 2) {
195                 node_move_rect_at_index_into(right, right->count-1, left);
196             }
197         } else if (right->count < 2) { // reverse sort by max axis
198             node_sort_by_axis(left, axis, true, true);
199             while (right->count < 2) {
200                 node_move_rect_at_index_into(left, left->count-1, right);
201             }
202         }
203         node_sort(right);
204         node_sort(left);
205         return right;
206     }
207
208     struct node *node_split(struct rtree *tr, struct rect *r, struct node *left) {
209         return node_split_largest_axis_edge_snap(tr, r, left);
210     }
211
212     int node_rsearch(struct node *node, NUMTYPE key) {
213         for (int i = 0; i < node->count; i++) {
214             if (!(node->rects[i].min[0] < key)) {
215                 return i;
216             }
217         }
218         return node->count;
219     }
220
221     double rect_unioned_area(struct rect *rect, struct rect *other) {
222         double area = (double)MAX(rect->max[0], other->max[0]) - (double)MIN(rect->min[0],
    other->min[0]);
223         for (int i = 1; i < DIMS; i++) {
224             area *= (double)MAX(rect->max[i], other->max[i]) - (double)MIN(rect->min[i],
    other->min[i]);
```

```
225         }
226         return area; // returns the area of two rects expanded
227     }
228
229     int node_choose_least_enlargement(struct node *node, struct rect *ir) {
230         int j = -1;
231         double jenlargement = 0, jarea = 0;
232         for (int i = 0; i < node->count; i++) {
233             // calculate the enlarged area
234             double uarea = rect_unioned_area(&node->rects[i], ir);
235             double area = rect_area(&node->rects[i]);
236             double enlargement = uarea - area;
237             if (j == -1 || enlargement < jenlargement || (!(enlargement > jenlargement) && area < jarea)){
238                 j = i;
239                 jenlargement = enlargement;
240                 jarea = area;
241             }
242         }
243         return j;
244     }
245
246     int node_choose_subtree(struct node *node, struct rect *ir) {
247         // take a quick look for the first node that contain the rect.
248         if (FAST_CHOOSER == 1) {
249             int index = -1;
250             double narea;
251             for (int i = 0; i < node->count; i++) {
252                 if (rect_contains(&node->rects[i], ir)) {
253                     double area = rect_area(&node->rects[i]);
254                     if (index == -1 || area < narea) {
255                         narea = area;
256                         index = i;
257                     }
258                 }
259             }
260         } else if (FAST_CHOOSER == 2) {
261             for (int i = 0; i < node->count; i++) {
262                 if (rect_contains(&node->rects[i], ir)) {
263                     return i;
264                 }
265             }
266         }
267         // fallback to using the choose-least-enlargment algorithm
268         return node_choose_least_enlargement(node, ir);
269     }
270
271     struct rect node_rect_calc(struct node *node) {
272         struct rect rect = node->rects[0];
273         for (int i = 1; i < node->count; i++) {
274             rect_expand(&rect, &node->rects[i]);
275         }
276         return rect;
277     }
278
279     int node_order_to_right(struct node *node, int index) {
280         while (index < node->count - 1 && node->rects[index + 1].min[0] < node->rects[index].min[0]) {
281             node_swap(node, index + 1, index);
```

```
282             index++;
283         }
284         return index;
285     }
286 
287     int node_order_to_left(struct node *node, int index) {
288         while (index > 0 && node->rects[index].min[0] < node->rects[index - 1].min[0]) {
289             node_swap(node,index, index - 1);
290             index--;
291         }
292         return index;
293     }
294 
295     // performs a copy of the data from args[1] & args[2], expects a rectangle (double[] double[])
296     // first N values are min corner, next N values - max corner, N - num of dimensions (max coords are
        optional)
297     bool node_insert(struct rtree *tr, struct rect *nr, struct node *node, struct rect *ir, struct item
298     item, bool *split, bool *grown) {
299         *split = false;
300         *grown = false;
301         if (node->kind == LEAF) {
302             if (node->count == MAX_ENTRIES) {
303                 *split = true;
304                 return true;
305             }
306             int index = node_rsearch(node, ir->min[0]);
307             memmove(&node->rects[index + 1], &node->rects[index], (node->count-index) * sizeof(struct
        rect));
308             memmove(&node->items[index + 1], &node->items[index], (node->count-index) *
        sizeof(struct item));
309             node->rects[index] = *ir;
310             node->items[index] = item;
311             node->count++;
312             *grown = !rect_contains(nr, ir);
313             return true;
314         }
315         int index = node_choose_subtree(node, ir); // choose a subtree for inserting the rectangle
316         if (!node_insert(tr, &node->rects[index], node->children[index], ir, item, split, grown)) {
317             return false;
318         }
319         if (*split) {
320             if (node->count == MAX_ENTRIES) {
321                 return true;
322             }
323             struct node *left = node->children[index];
324             struct node *right = node_split(tr, &node->rects[index], left); // split child node
325             if (!right) {
326                 return false;
327             }
328             node->rects[index] = node_rect_calc(left);
329             memmove(&node->rects[index + 2], &node->rects[index + 1], (node->count - (index + 1)) *
        sizeof(struct rect));
330             memmove(&node->children[index + 2], &node->children[index + 1], (node->count - (index +
        1)) * sizeof(struct node*));
331             node->rects[index + 1] = node_rect_calc(right);
332             node->children[index + 1] = right;
333             node->count++;
```

```
334             if (node->rects[index].min[0] > node->rects[index + 1].min[0]) {
335                 node_swap(node, index + 1, index);
336             }
337             index++;
338             node_order_to_right(node, index);
339             return node_insert(tr, nr, node, ir, item, split, grown);
340         }
341         if (*grown) { // child rectangle must expand to accomadate new item
342             rect_expand(&node->rects[index], ir);
343             node_order_to_left(node, index);
344             *grown = !rect_contains(nr, ir);
345         }
346         return true;
347     }
348
349     struct rtree *rtree_new_with_allocator(void *(*cust_malloc)(size_t), void (*cust_free)(void*)) {
350         if (!cust_malloc) cust_malloc = malloc;
351         if (!cust_free) cust_free = free;
352         struct rtree *tr = (struct rtree *)cust_malloc(sizeof(struct rtree));
353         if (!tr) { return NULL; }
354         memset(tr, 0, sizeof(struct rtree));
355         tr->malloc = cust_malloc;
356         tr->free = cust_free;
357         return tr;
358     }
359
360     struct rtree *rtree_new() { return rtree_new_with_allocator(NULL, NULL); }
361
362     bool rtree_insert
        (struct rtree *tr, const NUMTYPE *min, const NUMTYPE *max, const DATATYPE data) {
363         struct rect rect;
364         memcpy(&rect.min[0], min, sizeof(NUMTYPE) * DIMS);
365         memcpy(&rect.max[0], max ? max : min, sizeof(NUMTYPE) * DIMS);
366         struct item item;
367         memcpy(&item.data, &data, sizeof(DATATYPE));
368         if (!tr->root) {
369             struct node *new_root = node_new(tr, LEAF);
370             if (!new_root) return false;
371             tr->root = new_root;
372             tr->rect = rect;
373         }
374         bool split = false, grown = false;
375         if (!node_insert(tr, &tr->rect, tr->root, &rect, item, &split, &grown)) { return false; }
376         if (split) {
377             struct node *new_root = node_new(tr, BRANCH);
378             if (!new_root) return false;
379             struct node *left = tr->root;
380             struct node *right = node_split(tr, &tr->rect, left);
381             tr->root = new_root;
382             tr->root->rects[0] = node_rect_calc(left);
383             tr->root->rects[1] = node_rect_calc(right);
384             tr->root->children[0] = left;
385             tr->root->children[1] = right;
386             tr->root->count = 2;
387             tr->height++;
388             node_sort(tr->root);
389             return rtree_insert(tr, min, max, data);
```

```
390         }
391         if (grown) {
392             rect_expand(&tr->rect, &rect);
393             node_sort(tr->root);
394         }
395         tr->count++;
396         return true;
397     }
398
399     void rtree_free(struct rtree *tr) {
400         if (tr->root) { node_free(tr, tr->root); }
401         tr->free(tr);
402     }
403
404     bool node_search(struct node *node, struct rect *rect, bool (*iter)
        (const NUMTYPE *min, const NUMTYPE *max, const DATATYPE data, void *udata), void *udata) {
405         if (node->kind == LEAF) {
406             for (int i = 0; i < node->count; i++) {
407                 if (rect_intersects(&node->rects[i], rect)) {
408                     if (!iter(node->rects[i].min, node->rects[i].max, node->items[i].data, udata)) {
409                         return false;
410                     }
411                 }
412             }
413             return true;
414         }
415         for (int i = 0; i < node->count; i++) {
416             if (rect_intersects(&node->rects[i], rect)) {
417                 if (!node_search(node->children[i], rect, iter, udata)) {
418                     return false;
419                 }
420             }
421         }
422         return true;
423     }
424
425     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) {
426         struct rect rect;
427         memcpy(&rect.min[0], min, sizeof(NUMTYPE) * DIMS);
428         memcpy(&rect.max[0], max ? max : min, sizeof(NUMTYPE) * DIMS);
429         if (tr->root && rect_intersects(&tr->rect, &rect)) {
430             node_search(tr->root, &rect, iter, udata);
431         }
432     }
433
434     size_t rtree_count(struct rtree *tr) { return tr->count; }
435
436     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) {
437         *removed = false;
438         *shrunk = false;
439         if (node->kind == LEAF) {
440             for (int i = 0; i < node->count; i++) {
441                 if (!rect_contains(ir, &node->rects[i])) {
442                     continue;
```

```
443             }
444             int cmp = compare ?
445                 compare(node->items[i].data, item.data, udata) :
446                 memcmp(&node->items[i].data, &item.data, sizeof(DATATYPE));
447             if (cmp != 0) {
448                 continue;
449             }
450             // found the target item to delete
451             memmove(&node->rects[i], &node->rects[i + 1], (node->count - (i + 1)) * sizeof(struct
    rect));
452             memmove(&node->items[i], &node->items[i + 1], (node->count - (i + 1)) * sizeof(struct
    item));
453             node->count--;
454             if (rect_onedge(ir, nr)) {      // item was on the edge of node rect
455                 *nr = node_rect_calc(node); // recalculation of node rect
456                 *shrunk = true;             // notify the caller that rect is shrunk
457             }
458             *removed = true;
459             return;
460         }
461         return;
462     }
463     for (int i = 0; i < node->count; i++) {
464         if (!rect_contains(&node->rects[i], ir)) {
465             continue;
466         }
467         struct rect crect = node->rects[i];
468         node_delete(tr, &node->rects[i], node->children[i], ir, item, removed, shrunk, compare, udata);
469         if (!*removed) {
470             continue;
471         }
472         if (node->children[i]->count == 0) { // underflow
473             node_free(tr, node->children[i]);
474             memmove(&node->rects[i], &node->rects[i + 1], (node->count - (i + 1)) * sizeof(struct
    rect));
475             memmove(&node->children[i], &node->children[i + 1], (node->count - (i + 1)) *
    sizeof(struct node *));
476             node->count--;
477             *nr = node_rect_calc(node);
478             *shrunk = true;
479             return;
480         }
481         if (*shrunk) {
482             *shrunk = !rect_equals(&node->rects[i], &crect);
483             if (*shrunk) {
484                 *nr = node_rect_calc(node);
485             }
486             node_order_to_right(node, i);
487         }
488         return;
489     }
490     return;
491 }
492
493 // search the tree for an item contained within provided rect, perform a binary comparison of its data to
    provided, first item found is deleted
494 void rtree_delete
```

```
(struct rtree *tr, const NUMTYPE *min, const NUMTYPE *max, const DATATYPE data) {
495     struct rect rect;
496     memcpy(&rect.min[0], min, sizeof(NUMTYPE) * DIMS);
497     memcpy(&rect.max[0], max ? max : min, sizeof(NUMTYPE) * DIMS);
498     struct item item;
499     memcpy(&item.data, &data, sizeof(DATATYPE));
500     if (!tr->root) { return; }
501     bool removed = false, shrunk = false;
502     node_delete(tr, &tr->rect, tr->root, &rect, item, &removed, &shrunk, NULL, NULL);
503     if (!removed) {
504         return;
505     }
506     tr->count--;
507     if (tr->count == 0) {
508         node_free(tr, tr->root);
509         tr->root = NULL;
510         memset(&tr->rect, 0, sizeof(struct rect));
511     } else {
512         while (tr->root->kind == BRANCH && tr->root->count == 1) {
513             struct node *prev = tr->root;
514             tr->root = tr->root->children[0];
515             prev->count = 0;
516             node_free(tr, prev);
517         }
518         if (shrunk) {
519             tr->rect = node_rect_calc(tr->root);
520         }
521     }
522 }
```

3. Файл rtree.h

```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);
```

4. 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

Григорьев Юрий Вадимович

(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.