Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентрованное программирование»

Выполнил
студент группы 3331506/80401 _______________ Г. А. Мошковский

Руководитель _______________ М. С. Ананьевский

«__»_________2021 г

Санкт-Петербург
2021

Оглавление

1 ВВЕДЕНИЕ

В работе будет рассмотрено R-дерево (рисунок 1.1) – древовидная структура данных, каждый узел которой представляет минимальный ограничивающий прямоугольник (MBR) своих потомков в d-мерном пространстве (определение, предложенное Антонином Гуттманом в 1984 году как расширение B-дерева для многомерных данных). Она подобна B-дереву, но используется для методов пространственного доступа, т.е. для индексации многомерной информации, такой как географические координаты, прямоугольники или многоугольники.

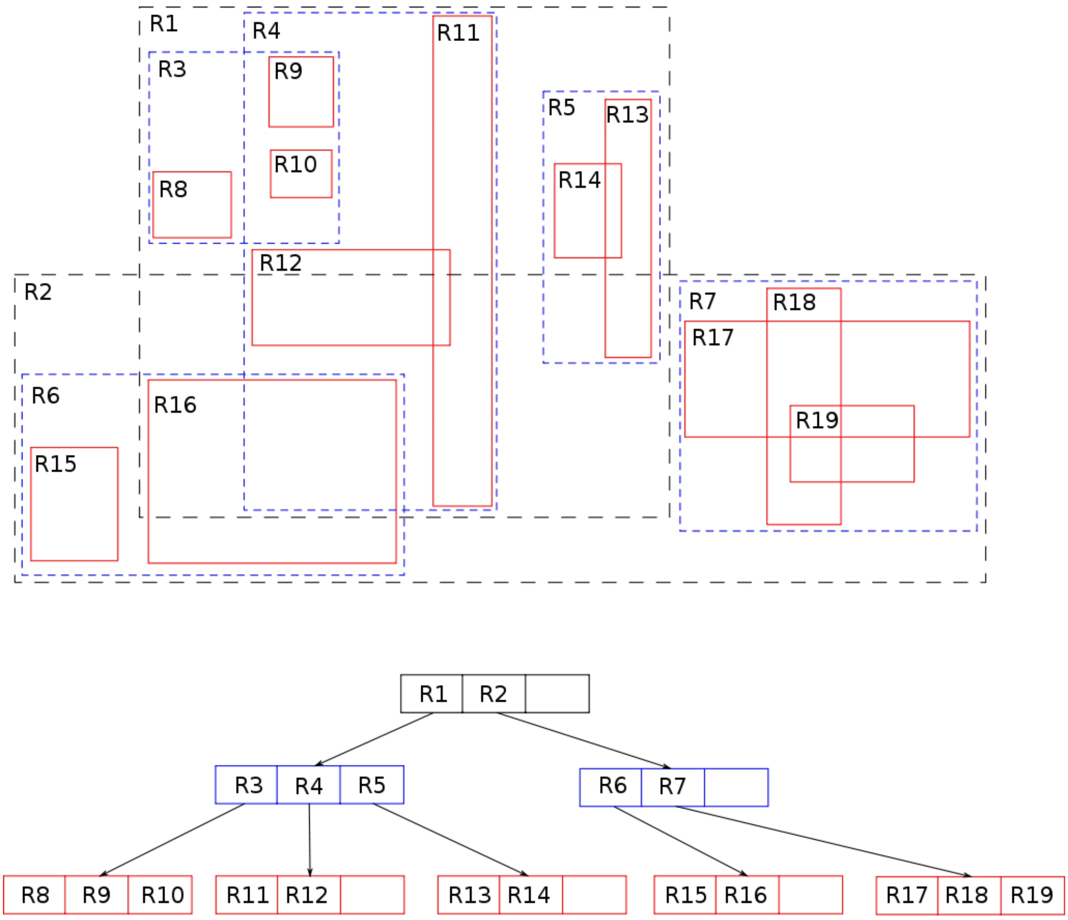

Рисунок 1.1 – Реализация R-дерева

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.

2 ИДЕЯ АЛГОРИТМА

R-дерево порядка (m, M), где m – минимальное количество записей в узле R-дерева и M – максимальное количество, должно удовлетворять следующим характеристикам:

- Каждый листовой узел, если он не является корнем, может вмещать не более M записей и не менее $m \leq M/2$. Запись представляет собой пару (mbr, oid), где mbr – минимальный ограничивающий прямоугольник пространственного объекта, а oid – его идентификатор.;
- Для внутреннего узла ограничение на количество записей является таким же, как и для листового. Однако записи имеют вид (mbr, p), где p – указатель на потомка узла, а mbr – MBR этого потомка;
- Корень может содержать минимум 2 записи, если не является листом. В противном случае минимальное количество записей 0 (пустое дерево);
- Все листовые узлы должны располагаться на одном уровне;
- Каждый объект упоминается в дереве ровно один раз.

Ключевая идея структуры данных состоит в том, чтобы сгруппировать близлежащие объекты и представить их с их минимальным ограничивающим прямоугольником на следующем более высоком уровне дерева; «R» в R-дереве означает прямоугольник. Эти ограничивающие рамки используются для поиска внутри поддерева. Таким образом, большинство узлов в дереве никогда не читаются во время поиска. Как и B-деревья, R-деревья подходят для больших наборов данных и баз данных, где узлы могут быть выгружены в память, когда это необходимо, а все дерево не может храниться в основной памяти. Даже если данные можно уместить в памяти (или кэшировать), R-деревья в большинстве практических приложений обычно обеспечивают преимущества в производительности по сравнению с простой проверкой всех объектов, когда количество объектов превышает несколько сотен или около того.

Рассмотрим алгоритм на примере, показанном на рисунке 2.1. Область поиска соответствует заданному прямоугольнику ABCD.

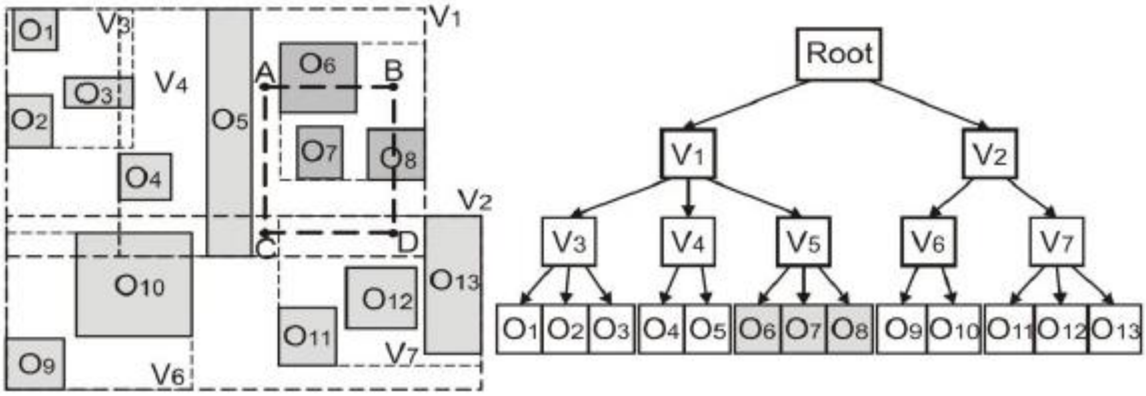

Рисунок 2.1 – Пример поиска в R-дереве

В первую очередь поиск вызывается для корня. Так как корень является внутренней вершиной, то для него выполняется первая ветка алгоритма поиска. Проверяются узлы V_1 и V_2 на пересечение с областью ABCD. Как видно из рисунка, оба этих узла имеют пересечения (общие точки) с заданным прямоугольником, следовательно для обоих этих узлов рекурсивно вызывается процедура поиска.

Во время поиска для вершины V_1 перебираются элементы V_3, V_4, V_5. Только V_5 имеет пересечение с прямоугольником ABCD, а значит вершины V_3 и V_4 пропускаются и более не рассматриваются. Дальнейший поиск для V_5 передаст в качестве результата три элемента – O_6, O_7, O_8, которые будут добавлены в множество результата Res.

Точно также будет проверена ветка V_2. Из ее потомков только V_7 пересекается с ABCD. Но ни один из элементов в данной вершине не пересекается с областью поиска. Получается, что данная ветка оказалась ложной.

В результате поиска получаем список элементов, удовлетворяющих заданному запросу:

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

3 ЭФФЕКТИВНОСТЬ

Опишем скорость работы алгоритма в зависимости от количества входных данных.

Наихудший случай – $O(n)$: представим, что мы храним много перекрывающихся прямоугольников в R-дереве. Теперь, сохранив один маленький прямоугольник, расположенный в области перекрытия всех остальных прямоугольников, получим запрос для этого прямоугольника, который должен будет пройти по всем поддеревьям: узлы $\rightarrow O(\log_M n)$ и записи $\rightarrow O(n)$.

В лучшем случае $O(\log n)$. R-дерево имеет одинаковую глубину в каждой ветви, а данные хранятся только в листовых узлах, поэтому всегда придется проходить $O(\log_M n)$ узлов и все записи в этом узле, поэтому это должно быть $O(M \cdot \log_M n)$.

Рассчитать среднее значение $O(\log_M n)$ представляется довольно сложной задачей. Предположим есть какие-то средние нормально распределенные данные с небольшим количеством перекрытий. Тогда средний запрос не должен пересекать несколько поддеревьев. Следовательно, среднее значение предположительно равно $O(M \cdot \log_M n)$ из-за обхода M записей в узле.

Данные собраны в таблицу 1.

Таблица 1 – Алгоритмическая сложность сортировки бинарным деревом

Худшая	$O(n)$
Средняя	$O(\log_M n)$
Лучшая	$O(\log n)$

4 РЕЗУЛЬТАТЫ РАБОТЫ АЛГОРИТМА

Для анализа работы алгоритма, будем опираться на данные эксперимента, полученные Антонином Гуттманом [1].

Использовалась реализация R-дерева, предложенная Гуттманом, в серии тестов на производительность, целью которых была проверка практичности структуры, выбор значения для M и m, а также оценка различных алгоритмов разделения узлов.

- Для теста использовались 5 разных страниц (рисунок 4.1). Значения проверены для m, минимальное число записей в узле были M/2, M/3 и 2.
- Первая часть в каждом тесте заключалась в использовании метода вставки, чтобы изучить производительность для каждой новой индексной записи
- Во второй части использовался метод поиска для нахождения прямоугольников, образованных произвольными числами
- В третьей части изучался алгоритм удаления из дерева, который удалял индексную запись для каждого элемента данных

Bytes per Page	Max Entries per Page (M)
128	6
256	12
512	25
1024	50
2048	102

Рисунок 4.1 – Сводная таблица

Производительность операции вставки:

- С линейным разбиением по времени вставки тратят очень мало времени на разбиение
- Ожидаемый рост с размером страницы
- Увеличение m снижает затраты на вставку потому что требуется минимальная вместимость

- Исчерпывающий (exhaustive) алгоритм требует много времени с уже меньшим количеством страниц. Линейный алгоритм, как и ожидалось, самый быстрый. С большим количеством байтов на страницу затраты ЦП не увеличиваются так сильно

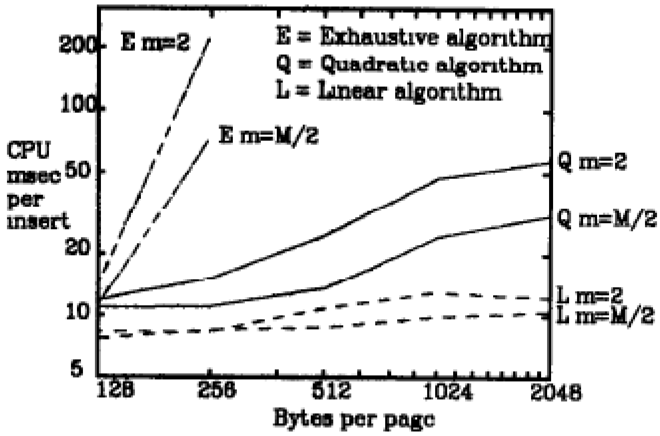

Рисунок 4.2 – Затраты ЦП на вставку записей

Производительность операции удаления:

- Затраты на удаление зависит от m. Для больших m:
- Больше узлов становятся неполными (занятость < m)
- Происходит больше повторных вставок
- Больше возможных разделений
- Довольно плоха продолжительность для $m = M/2$
- На результат сильно влияло минимальное требование заполнения узла. Если значение m маленькое, то узлы часто становились переполненными

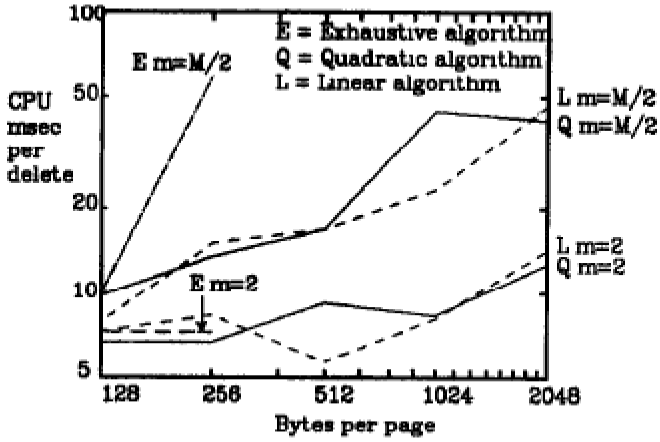

Рисунок 4.3 – Затраты ЦП на удаление записей

Производительность операции поиска:

- Поиск относительно нечувствителен к алгоритму разделения
- Меньше ввода/вывода для больших страниц
- Большая занятость ЦП у больших страниц
- Меньшие значения m сокращают среднее число записей на узел, так что тратится меньше времени на поиск в узле

Диаграммы (рисунок 4.4 и рисунок 4.5) показывают почти такой же результат с другим алгоритмом. Причина в том, что операция поиска не использует SplitNode.

Эффективность использования пространства:

- Более строгий критерий заполнения узла приводит к меньшему индексу
- Использование пространства для R-дерева и количество данных

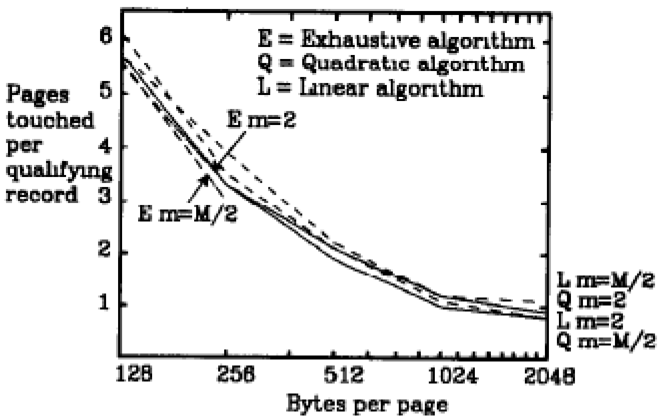

Рисунок 4.4 – Эффективность поиска и затронутые страницы

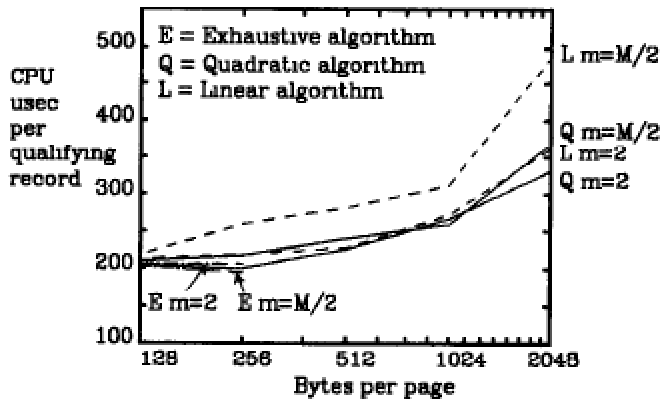

Рисунок 4.5 – Эффективность поиска и занятость ЦП

5 СПИСОК ЛИТЕРАТУРЫ

1) A. Guttman,” R-trees: A Dynamic Index Structure for Spatial Searching” Proc. ACM SIGMOD, pp. 47–57, 1984.
2) Гулаков В. К., Трубаков А. О. Многомерные структуры данных. 2010 — 387 с.
3) https://www2.cs.sfu.ca/CourseCentral/454/jpei/slides/R-Tree.pdf
4) https://en.wikipedia.org/wiki/R-tree

6 ПРИЛОЖЕНИЕ

Приложение 1

```
#include <iostream>
#include "RTree.h"

using namespace std;

typedef int ValueType;

struct Rect
{
  Rect()  {}

  Rect(int a_minX, int a_minY, int a_maxX, int a_maxY)
  {
    min[0] = a_minX;
    min[1] = a_minY;

    max[0] = a_maxX;
    max[1] = a_maxY;
  }


  int min[2];
  int max[2];
};

struct Rect rects[] =
{
  Rect(0, 0, 2, 2), // xmin, ymin, xmax, ymax (for 2 dimensional RTree)
  Rect(5, 5, 7, 7),
  Rect(8, 5, 9, 6),
  Rect(7, 1, 9, 2),
};

int nrects = sizeof(rects) / sizeof(rects[0]);

Rect search_rect(6, 4, 10, 6); // search will find above rects that this one
overlaps


bool MySearchCallback(ValueType id)
{
  cout << "Hit data rect " << id << "\n";
  return true; // keep going
}


int main()
{
  typedef RTree<ValueType, int, 2, float> MyTree;
  MyTree tree;

  int i, nhits;
  cout << "nrects = " << nrects << "\n";

  for(i=0; i<nrects; i++)
  {
    tree.Insert(rects[i].min, rects[i].max, i); // Note, all values including
zero are fine in this version
  }
```

```
  nhits = tree.Search(search_rect.min, search_rect.max, MySearchCallback);

  cout << "Search resulted in " << nhits << " hits\n";

  // Iterator test
  int itIndex = 0;
  MyTree::Iterator it;
  for( tree.GetFirst(it);
       !tree.IsNull(it);
       tree.GetNext(it) )
  {
    int value = tree.GetAt(it);

    int boundsMin[2] = {0,0};
    int boundsMax[2] = {0,0};
    it.GetBounds(boundsMin, boundsMax);
    cout << "it[" << itIndex++ << "] " << value << " = (" << boundsMin[0] <<
"," << boundsMin[1] << "," << boundsMax[0] << "," << boundsMax[1] << ")\n";
  }

  // Iterator test, alternate syntax
  itIndex = 0;
  tree.GetFirst(it);
  while( !it.IsNull() )
  {
    int value = *it;
    ++it;
    cout << "it[" << itIndex++ << "] " << value << "\n";
  }

  // test copy constructor
  MyTree copy = tree;

  // Iterator test
  itIndex = 0;
  for (copy.GetFirst(it);
       !copy.IsNull(it);
       copy.GetNext(it) )
  {
    int value = copy.GetAt(it);

    int boundsMin[2] = {0,0};
    int boundsMax[2] = {0,0};
    it.GetBounds(boundsMin, boundsMax);
    cout << "it[" << itIndex++ << "] " << value << " = (" << boundsMin[0] <<
"," << boundsMin[1] << "," << boundsMax[0] << "," << boundsMax[1] << ")\n";
  }

  // Iterator test, alternate syntax
  itIndex = 0;
  copy.GetFirst(it);
  while( !it.IsNull() )
  {
    int value = *it;
    ++it;
    cout << "it[" << itIndex++ << "] " << value << "\n";
  }

  return 0;
```