Department of Computer Science,
The University of Hong Kong

**CSIS0323/COMP3323 Advanced Database Systems 2015**

# Assignment 1: R-tree Index Part I

**Due date: 26/03/2015 Thursday 11:59pm**

## 1 Problem Description

R-tree is a height-balanced tree for indexing multi-dimensional data such as geographical coordinates and rectangles. It could be viewed as an extension of B-tree to high dimensional space. R-tree has been widely used in research and in real database systems, e.g., Oracle and MySQL.

In this assignment, an R-tree index that can support **Insert** and **Search** operation over multi-dimensional data (not just limit to 2-dimension) needs to be implemented. A program framework is provided and you need to implement the functions involving Insert and Search operations. (Next assignment will be implementing Delete operation.)

## 2 Preliminary

### 2.1 Data Representation

We consider each object as a point (with no geometric extent) in $d$-dimensional space. For each object, there is exactly one additional attribute storing the record ID of that object. Therefore, each object $o$ is represented by $\langle x_1, x_2, \ldots, x_d, rid \rangle$, where $x_i$ is its coordinate in $i$-th dimension ($i = 1, \ldots, d$), and $rid$ is its record ID. For simplicity, we assume $x_i$ and $rid$ are **integers**.

The coordinate of object $o$, $(x_1, x_2, \ldots, x_d)$ is defined as the key of $o$. Also, to make it simple, we do not allow two objects having the same key in R-tree. For example, objects $\langle 1, 2, 3, 8 \rangle$ and $\langle 1, 2, 3, 9 \rangle$ are not allowed to be stored in the 3-dimensional R-tree at the same time. Therefore, when inserting a new object $\langle x_1, x_2, \ldots, x_d, rid \rangle$, we first check whether key $(x_1, x_2, \ldots, x_d)$ exists in the R-tree. If the key does not exist, then the new object needs to be inserted into the R-tree successfully, otherwise we just omit the new object.

### 2.2 Program Skeleton

We provide the following files for this programming assignment.

- **boundingbox.h/.cpp** declares/defines **class BoundingBox**, an instance of which can be the bounding box of a point object or a group of point objects.
- **rtnode.h/.cpp** declares/defines **class RTNode**, an instance of which is a node in the R-tree, as well as **class Entry**, an instance of which is an entry in the R-tree node.

  ***Hint: In your implementation, you may modify the data structures in boundingbox.h/.cpp and rtnode.h/.cpp to suit your needs as long as the modified structures are compatible to the given test module (i.e., main.cpp).
- **rtree.h/.cpp** are the skeleton code implementing R-tree index that supports updates (insertion as well as deletion) and searches in R-tree. You are required to implement 3 functions in rtree.cpp to facilitate Insert and Search ( More details in Section 2.3 and 3).

- **main.cpp**: the driver of the implementation to help you test your R-tree implementation. It will invoke the corresponding functions in R-tree for Insert and Search.

  ***Hint: **DO NOT** edit **main.cpp**. Your program will be compatible with it as long as you do not change any interface or name of provided functions. You are not required to submit this file. We will mark your codes by the **original** copy of this file.

  ***Function **main** takes 2 or 3 command line arguments, first two of which are compulsory, while the third is optional. The first argument is the maximum number of entries an R-tree node could accommodate. The second one specifies the dimension of the R-tree to be built. The third one, if exists, should be a file containing a sequence of commands to be executed. If the number of command line arguments is 2, the program will receive commands from the standard input. Otherwise it will receive commands from the specified file.
- **Demo.txt:** this file illustrates a running example of a R-tree index according to our assignment specification (with the maximum number of entries in a R-tree node, $max\_entry\_num = 4$ and $dimension = 2$). It consists of a sequence of commands and the corresponding output. You can use this example to help you verify your program.

  *** This file cannot be used directly as the input file of the program. You can modify it to be an input file by deleting those prompts and output, as well as the blank lines.

  *** Note that your result for this demo case may be correct but different from the output in Demo.txt since we generate point objects by **rand()** in C++ to test our program. **rand()** will generate different values on different machines even with the same random seed. But do not worry. if it is correct on your machine, it will be correct on my machine.

### 2.3 Data Structures

In this section we give a detailed description of those data structures provided.
**class BoundingBox** represents an axis-aligned bounding rectangle on the domain. It consists of the minimum and maximum values of the rectangle along each dimension.

- **lowest** is a vector storing the minimum values of the box at all the dimensions.
- **highest** is a vector storing the maximum values of the box at all the dimensions.

A point $(x_1, x_2, \ldots, x_d)$ can be modeled as a box whose $lowest = (x_1, x_2, \ldots, x_d)$ and $highest = (x_1, x_2, \ldots, x_d)$. For a box (i.e., MBR) in the R-tree with $d$-dimension, both its *lowest* and *highest* should have size $d$.

**class Entry** represents an entry in R-tree node.

- **mbr** is the bounding box of the node that the entry points to.
- **ptr** is the pointer pointing to the node represented by the entry.
- **rid** is the record id.

NOTE that each entry has two variables *ptr* and *rid*, but only one of them is actually used (i.e., either *ptr* or *rid* field is used in each entry, but NOT both). If the entry is in the leaf node, we use *rid* to access the point object and *ptr* is invalid. Otherwise we use *ptr* to access the node pointed by the entry and *rid* is invalid in this case. Such implementation scheme wastes some space in an RTNode. However, it helps to simplify the implementation you are going to do.

**class RTNode** represents a node in the R-tree. In the RTNode class:

- **entry_num** is the number of valid entries currently in the node.
- **size** is the entry capacity of a node, which should be the same as *max_entry_num* defined in class RTree shown later.
- **entries** is an array of size *max_entry_num* and contains the entries of the node.

  ***Note that *entry_num* and *max_entry_num* are different. *max_entry_num* defines the maximum number of entries a node could have. It will not be changed once it has been set. However, *entry_num* indicates the number of valid entries currently in the R-tree node. It may be changed when the R-tree is updated.
- **level** is the level of the current node, where 0 means current node is a leaf node, while a positive value means current node is a non-leaf node. The level of a non-leaf node is one plus the level of its children.

**class RTree** represents an R-tree. In the RTree class:

- **max_entry_num** indicates the entry capacity in an R-tree node (default to 4). This value should not be changed once the R-tree is instantiated.
- **dimension** specifies the dimensionality of the R-tree (default to 2), which should not be changed too once the R-tree is created.
- **root** is the root node of the R-tree. You need to update this variable so that it ALWAYS points to the root of the tree.
- **print_tree( )** prints out the content of the tree. It is useful in debugging.
- **void stat()** displays the statistics of the R-tree built.
- **bool tie_breaking(const BoundingBox& box1, const BoundingBox& box2)** resolves the tie cases that are not handled in paper (see Section 3.2 for detailed usage). You SHOULD use this function to handle any unsolved tie cases in order to produce the expected results, otherwise your program may produce correct but not the same result as mine, which makes it hard for me to verify. Returns: true if you should pick box1; false otherwise.

  ***__DO NOT__ change the three functions just described above.

# 3 Tasks

## 3.1 Functions to implement

You are required to implement the following functions in **class RTree** declared in file rtree.h.

- **bool insert(const vector⟨int⟩& coordinate, int rid)** inserts a point with *coordinate* and *rid* into the R-tree. (We require that a point **MUST** be modeled as a bounding box in the way mentioned ahead for the consistency of R-tree.) Returns: true if the insertion is successful; false if another point with the same coordinate already exists in the R-tree.
- **void query_range(const BoundingBox& mbr, int& result_count, int& node_traveled)** finds the points within the query range specified by the input *mbr*. Parameters: *mbr* is the query region represented by a bounding box. Returns: the number of points within the closed query range (*result_count*) and the number of nodes traveled (*node_traveled*) in searching. You are NOT required to print out the results.
- **bool query_point(const vector⟨int⟩& coordinate, Entry& result)** finds the leaf entry with the given query *coordinate*. Parameters: vector *coordinate* is the coordinate of

the query point. Returns: true if the record is found and the qualified entry is stored in *result*; false otherwise.

***Hint: To implement these three functions, you may need to implement your own sub-functions such as choose_leaf(), adjust_tree(), etc.

### 3.2 Implementation Details

You should follow the algorithms in the paper, as well as the following guidelines, STRICTLY to implement the codes. Marks will be deducted if the results in your implementation differ from the expected R-tree from the algorithms.

- **Space utilization:** Each node (except the root node) should be at least half full in this assignment, i.e., $m = ceiling(max\_entry\_num/2)$.
- **Node splitting:** Please implement the **linear-cost** algorithm to handle node splitting.

  In the LinearPickSeeds routine, if there is a tie in finding extreme rectangle along any dimension (refer to LPS1 in paper), use tie_breaking() as described in Section 2.3 to resolve the tie; if there is a tie in picking the most extreme pair (refer to LPS3 in paper), pick the pair along the dimension with smallest id.

  Also it is possible that LinearPickSeeds returns the same entry A if this entrys *mbr* has the highest low side and the lowest high side at the same time. We need to first use tie_breaking() to **sort** the remaining entires except A. For sort, if **tie_breaking(box1, box2)** returns true, we should swap the position of the corresponding entries of box1 and box2, which can give a total order for the entries. Then we select the first entry of the sorted remaining entries to be returned with A as the return pair of LinearPickSeeds. This method may not be the optimal one, but we use it just to simplify our assignment tasks and make sure your programs generate same result.

  In the PickNext routine, again please pick the next entry to assign according to function tie_breaking() whenever a tie occurs.
- **Tie-breaking for any other cases:** According to the algorithms, ties may also happen in other cases. For example in the ChooseLeaf routine in the insertion procedure, two entries may have the same enlargement areas and the same areas. In such case, the algorithm in the paper does not tell which entry is chosen. We require all such ties should be resolved by function tie_breaking(). In other words, use tie_breaking() to resolve all the cases that the paper states choose any (arbitrary) one or does not even tell how to handle it.

## 4 Submission

The deadline of the assignment is 11:59pm Mar. 26, 2015. Please hand in a single zip file containing all your R-tree implementation files EXCEPT main.cpp via Moodle only. Printed copies will NOT be accepted.

Make sure that your codes can be compiled and run correctly on the Department UNIX server honest (or virtue, genius, belief). Mark will be deducted if your codes fail to compile on the servers.