# An R-tree Index Using the STL Style

MingAn Zhong

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

February 2004

© MingAn Zhong, 2004

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 0-612-91161-6*
*Our file Notre référence*
*ISBN: 0-612-91161-6*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

# ABSTRACT

An R-tree Index Using the STL Style

Ming An Zhong

Indexes are critical for performance of database systems. Trees are effective indexes that handle both single-dimensional and multi-dimensional data. The R-tree is a commonly used multi-dimensional tree index for the spatial data and geographic information system (GIS).

By using design pattern and following the C++ STL style, the R-tree index structure in this thesis is designed and implemented using generic programming techniques. The components are designed to be the STL style containers so that they have a uniform and clear interface and can be used like a standard container.

The R-tree structure can adapt to different data types, user-defined key types, and support user-defined queries.

# Acknowledgements

I thank my research advisor, Greg Butler, for his exceptional guidance. He was always available for discussion and motivation.

I also thank the member of our search group with whom I worked closely. Among them, I must particularly mention JinXue Zhou and Bin Nie for their help and cooperation.

I thank all my friends at Concordia University for their friendship.

I am grateful to my parents and my family for their encouragement.

Finally, this work is dedicated to my wife, LiPing Wang, for her eternal patience, support and assistance, and most importantly, her love.

# Content

# List of Figures

# List of Tables

# Chapter 1 Introduction

The index structure is an important component of a modern database management system. A large variety of tree-structure indexes have been developed and applied to database management systems. Among them are those based on the hierarchical approach such as B+-tree, K-D-B-tree [Rob81], R-tree [Gutt84], SS-tree [White96], SR-tree [Kata97], and X-tree [Berc96]. An efficient implementation of an index structure is crucial for any database system to enhance the processing efficiently and make the database more applicable accordingly.

An investment in an efficient index to the database is usually a good idea. That is the reason that a specialized handcrafted index is a classical choice to achieve the maximum efficiency possible. Specialized access methods are usually hand-coded from scratch. The developers are required to know the substantial knowledge of the underlying file system to build the index structures correctly and efficiently. It costs time and needs more effort for implementation and maintenance. Compared with the handcraft codes, the framework technology is adapted to the implementation of a family of index structures due to its ability to promote the reuse of design and source code. As a result, frameworks can significantly reduce the cost of providing a new index.

## 1.1 The Problem and Related Work

### 1.1.1 The Problem

The Generalized Search Tree (GiST), realized by J.H. Hellerstein and his group [Heller95], is a framework supporting an extensible set of queries and data types. The

GiST framework enables an access developer to build any kind of balanced index tree on any kinds of data by implementing some specific methods for insertion, deletion, and search [Doel02]. GiST tries to address all possibilities in the same piece of code by generating flexible code with some hot spots that can be easily adjusted to develop different applications. This unfortunately leads to an even more generic and complex code that is bigger and less user-friendly. Furthermore, the source code itself has poor object-oriented style since the C programming language largely influences it.

With the advancement of the framework technology, the problems are recognized and addressed. The Know-It-All (KIA) project [Butler02] is investing methodologies for the development, application, and evolution of frameworks. A concrete framework for database management systems is under development. This will help getting closer to the goal of producing a framework that achieves the advantages of specialized code and framework reusable code.

### 1.1.2 Related Work

The Know-It-All project has been underway at Concordia University since 1997. The aim of this project is to research methodologies and models for framework development, application, and evolution to develop a framework for database management system, and to apply this framework to advanced database applications for bioinformatics. It provides support for all data models, integrated and heterogeneous databases, and eventually sustains incomplete and uncertain data.

The tree index framework is one of KIA's subprojects, including traditional B+-tree and multi-dimensional trees such as R-tree and its variants. It supports multiple queries such as exact match query, range query, similarity queries, and some user-

defined queries. The indexes are designed to follow good object-oriented design and use design patterns. Languages used are mainly Java, and C++.

[Gaff01] introduces a design of a generalized index framework, a sub-framework of KIA project. By designing in the STL style and applying a wealth of existing STL components, the author produced a design which is capable of producing tree-based index that are adaptable to different data /or key types, different queries, and different database application domains.

### 1.2 Our Work

In the design of [Gaff01], the page and index container are designed to be STL containers for which iterators are the only way to allow different algorithms to work with the containers. The built-in allocators hide the memory allocation and deallocation and the object persistence is achieved by using a specialized allocator.

As we know that "a framework is a set of classes that embodies an abstract design for solutions to a family of related problems" [John88], the framework usually defines the overall structure of all applications derived from it, their partitioning into classes, the key collaborate, and the thread of control. The implementation of a framework is still a very important part to be reused.

When analyzing the design of Ashraf Gaffar's framework, we found some obstacles for implementation. Since page is designed to be a generalized page, it has to be able to handle pair of key and data reference for leaf nodes, and pair of key and page pointer for non-leaf nodes. As a result, unsafe type casting would be used very frequently. In addition, it is not possible to make STL containers persistent by only using a specialized allocator, which will be discussed in Chapter 3.

Our solution is using STL generic programming and object-oriented design patterns to redesign and implement the R-tree structure. First, two kinds of nodes, i.e. index page (non-leaf node) and leaf page (leaf node), are designed to be STL style containers and derived from the base class page, an abstract interface class. A proxy is used to create the page from the non-volatile storage on demand, and maintain reference to the created page. Since the containers are designed to be template classes, and of STL style interface, they are user-friendly, readable and reusable.

### 1.3 Goal of the Thesis

The goal of this thesis is as following:

- Prove the concept of the index sub-framework of the Know-It-All project;
- Using the STL components, design and implement the reusable components of tree structures to be STL style containers that can be used for other multi-dimensional tree structures;
- Develop a generic R-tree that can adapt to different key types, and data reference types and support different queries;
- Conduct correctness testing on the components and performance testing on the R-tree implementation.

### 1.4 Contribution of the Thesis

The main objective of this paper is to describe the design and implementation of the R-tree index structure using generic programming. It also involves correctness and performance testing for R-tree structure. The major concerns are:

- Design LeafPage and IndexPage containers to be STL-style containers;

- Adopting STL generic programming to implement the R-tree index structure;
- Using design patterns;
- Using proxy (smart pointer) to load a page on demand, and to maintain the reference to the loaded page.
- Providing basic predicates, predicate binder and Cursor to support different queries and user-defined queries;

### 1.5 Layout of the Thesis

The thesis is organized as follows. We begin with an introduction with an overview of the work. In Chapter 2, we represent background on the reusable tree-based index structures, generic programming features, and some existing R-tree implementations. Then the R-tree design is presented in Chapter 3 in detail. In Chapter 4, implementation of R-tree is describes and a comparative performance study of these implementation is presented. Finally in Chapter 5, the conclusions on the contribution of this thesis are summarized.

# Chapter 2 Background

This chapter will briefly review the R-tree index structure and its algorithms. We will simply introduce the STL components and some design patterns since they are involved in the design and implementation of the R-tree index.

## 2.1 Multidimensional Data and Database Index

Traditional Database Management systems (DBMS) are very effective in storing and retrieving business and accounting data. The DBMS is organized to optimize queries on this kind of data. A traditional DBMS can efficiently process a query like "find me the top ten companies that recruit computer science students from Concordia". The DBMS will use its index to narrow down its search and efficiently return the query result. However, if we change the query slightly to include some spatial information, giving us "find me the top ten companies that recruit computer science students from Concordia and are within 20 miles of Sir George Williams Campus", the conventional DBMS will be quickly sent into a tailspin since the one-dimensional index cannot efficiently handle a multi-dimensional spatial query.

Modern database systems are commonly developed to manage and process spatial or geometric data, that is, data related to space. They are designed with capabilities for representing, querying, and manipulating spatial data. Such systems provide the underlying database technology needed to support applications such as computer aided design (CAD), and geographical information system (GIS). Therefore it is very important to be able to retrieve objects efficiently according to their spatial locations.

An efficient implementation of search index is crucial for any database system. An index is a database object that can greatly increase database performance, enabling faster query, and modification on the data. Spatial location usually involves multiple dimensions (e.g., 2-D or 3-D). Traditional indexing structures are not suitable for spatial indexing since the key space is multidimensional and range query is required. Structures using one-dimension ordering of key values, such as B-trees, are useless due to the multidimensional search space. Structures based on exact matching of values, such as hash tables, do not work because they do not support range queries.

With the ever-increasing demands for the manipulation of spatial data, spatial database systems evolve and several multidimensional data structures have been proposed such as quadtree [Fink74], kd-tree [Bent75], and R-tree [Gutt84]. Among them, the R-tree has been established as one of the most important data structures for indexing spatial data. Since its introduction in 1984, the R-tree and its later refinement the R*-tree [BKS + 90] have gained wide acceptance in the academic database community as the preferred spatial indexing method. They can be used for indexing both point and spatial data (data with spatial extends) and is the only multidimensional indexing structure known to has been incorporated as an access method into commercial data management system [Chak99].

## 2.2 R tree

### 2.2.1 Structure

The R-tree is a height-balanced and multi-dimensional generalization of B-tree with all index records in its leaf nodes [Gutt84]. It is used as an indexing structure to speed up the retrieval of spatial objects. The decision on which node to visit is made based on the evaluation of spatial predicates, so the tree must hold some sort of spatial data

on all nodes. A spatial database consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier, which can be used to retrieve needed spatial objects.

The R-tree has two types of nodes: leaf nodes and non-leaf nodes. An entry in a leaf node is a tuple of the form ( I, dataReference) where *dataReference* refers to a database record that contains the actual object, which possibly has an arbitrary shape and *I* is a k-dimensional minimum bounding rectangle (MBR) of the indexed spatial object.

$$I = ( I_0 , I_1 , I_3 , \ldots, I_i , \ldots , I_{k-2} , I_{k-1} )$$

Here $I_i$ is a closed bounded interval [a, b] describing the extent of the object along dimension i and k is the number of dimensions. Each entry in a non-leaf node is a tuple ( I, child-pointer) where *child-pointer* is a pointer to a lower level node in the tree, and *I* is the MBR that covers all the rectangles of the lower level child node pointed by *child-pointer*.

The R-tree satisfies the following properties:

- Every node except the root node contains between m and M entries where m $\leqslant$ M/2, where M and m are the maximum and minimum number of entries in a node;
- For each leaf entry, key I is the smallest rectangle (MBR) that spatially contains the n-dimensional data object represented by the indicated tuple;
- For each non-leaf entry, key I is the smallest rectangle that spatially contains the rectangles in its child nodes;
- The root node has at least two children unless it is a leaf;
- All leaves appear at the same level.

Figure 2-1 [Gutt84] shows the structure of a 2-dimentional R-tree, and its corresponding planar view, respectively.

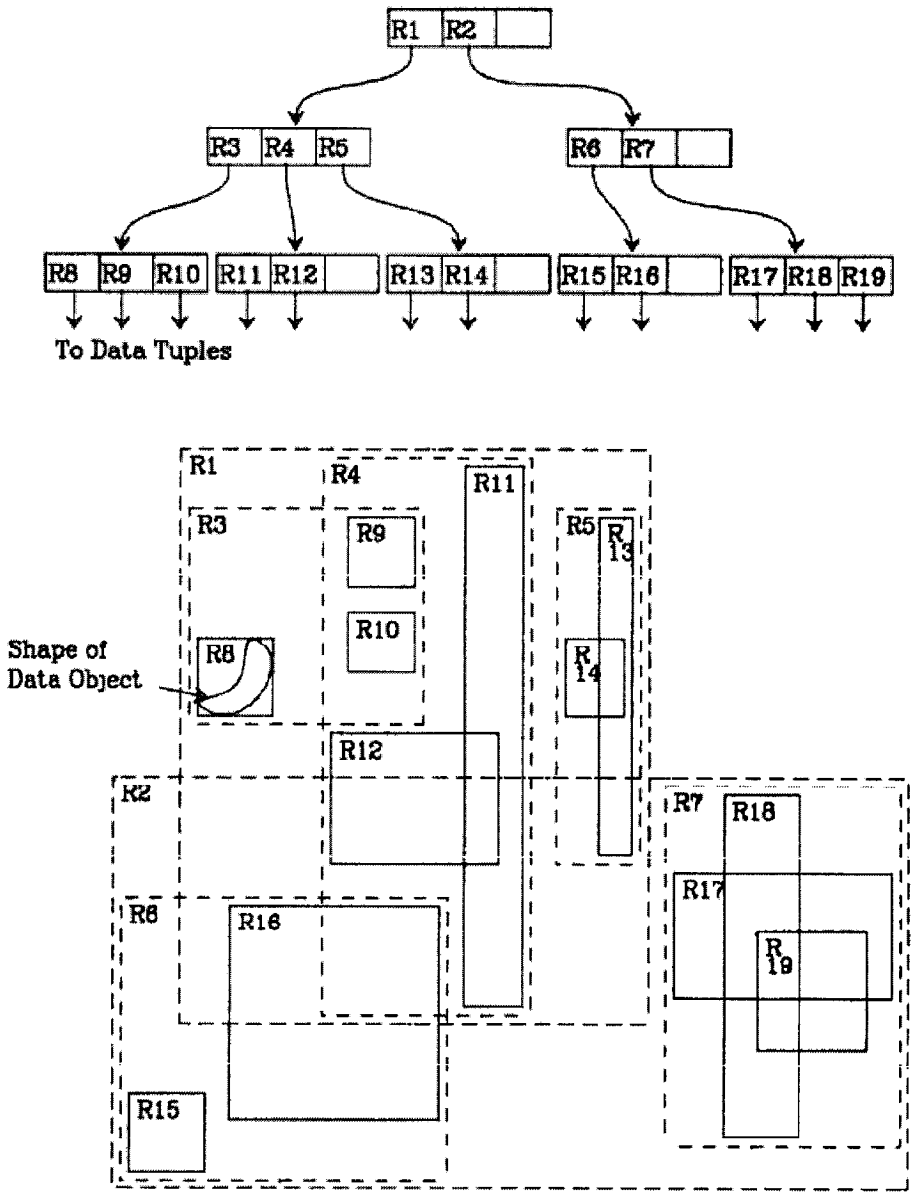



Figure 2-1 Structure and Planar View of an R-tree

### 2.2.2 Algorithms

The description of algorithms is totally from the classic sources on R-tree [Gutt84].

#### 2.2.2.1 Search

The search algorithm proceeds by descending the tree from the root in a manner similar to a B+-tree. Unlike a B+-tree, more than one subtrees under a node visited may need to be searched in an R-tree. The most typical example of search is the one where the user asks for all objects overlapping a certain area. Since the MBRs stored in the index entries are allowed to overlap, the R-tree cannot guarantee that only one search path needs to be traversed.

| Algorithm **Search** | |
|---|---|
| Input: | T: root node of an R-tree<br>S: a search rectangle |
| Output: | All tuples whose rectangle overlap S |
| S1 [Search subtree] | If T is a non-leaf node, check each entry E on T to determine whether E.I overlaps S. For each such entry, invoke **Search** on the node pointed by E.ptr |
| S2 [Search leaf node] | If R is a leaf node, check each entry E on R to determine whether E overlaps with S. If so, E is a qualified record. |

#### 2.2.2.2 Insertion

The insertion algorithm adds new index records to the leaves in the similar way of B+-tree. The leaf selection (chooseLeaf) will start from the root and will follow the path denoted by the entries whose MBRs need least enlargement in order to contain the MBR of the new record. The record is inserted to the chosen leaf. If the leaf node already has M entries, then it has to split to get two nodes to accommodate the (M+1) entries. The criterion used for split is also minimizing the area of the enclosing

rectangle in the inner nodes. The generated changes will propagate from the leaf node upwards.

| Algorithm **Insert** | |
|---|---|
| Input: | E: a new entry to be inserted |
| Output: | None |
| I1 [Find Position for New Record] | Invoke **ChooseSubtree** to select a leaf node L in which to place E; |
| I2 [Add record to leaf node] | If L has enough space for a new entry, add E to L. Else invoke **SplitNode** to obtain a new node LL and distribute the L's M entries and E into L and LL; |
| I3 [Propagate changes upward] | If no split occurs, invoke **AdjustTree** on L. Otherwise invoke **AdjustTree** on L and LL; |
| I4 [Grow tree taller] | If node split propagation caused the root to split, create a new root whose children are the two resulting nodes. |

Algorithm ChooseSubtree selects a leaf node in which to place a new index entry.

| Algorithm **ChooseLeaf** | |
|---|---|
| Input: | E: a new record to be inserted |
| Output: | a leaf node to accommodate E |
| CL1 [Initialize] | Set N to be T. |
| CL2 [Level check] | If N is a leaf node, return N. |
| CL3 [Choose subtree] | If N is a non-leaf node, let F be the entry in N whose MBR needs least enlargement to include E.I. When there are more qualify entries with the same least enlargement in N, choose the entry with the smallest area of rectangle. |
| CL4 [Descend to leaf node] | Set N to be the child node F which is pointed to by F.ptr Repeat from CL2 |

Algorithm AdjustTree ascend from a leaf node L to the root, adjusting covering rectangle and propagating node split as necessary.

| Algorithm **AdjustTree** | |
|---|---|
| Input: | L: a leaf node of an R-tree |
| Output: | none |
| AT1 [Initialize] | Set N to be L. If split node LL is passed, also set NN to be LL; |
| AT2 [Check if done] | Stop if N is the root |
| AT3 [Adjust covering MBR in parent entry] | Let P be the parent node of N and $E_N$ be the N's entry in P. Adjust $E_N$.I so that all rectangles in N are tightly enclosed. |
| AT4 [Propagate node split upward] | If there is previously split node NN, create a new entry with $E_{NN}$.ptr pointing to NN and $E_{NN}$.I enclosing all rectangles in NN. If there is room in P, add $E_{NN}$. Otherwise invoke **SplitNode** to get P and PP which include $E_{NN}$ and all old entries of P. |
| AT5 [Move up to next upper level] | Set N to be P, and NN to be PP if a split happened. Repeat from AT2 |

#### 2.2.2.3 Deletion

The deletion algorithms first identify the leaf containing the entry and then the entry is removed. If the leaf becomes under-populated (number of entries < m), all the remaining entries are saved in a list and the node is removed. The changes are propagated up the tree by updating the corresponding covering rectangles and removing the nodes that become under-populated. Then each orphan entry in the list will be reintroduced into the tree using the insertion procedure. The entries should be inserted at the same level of the tree where they originally belonged.

| Algorithm **Delete** | |
|---|---|
| Input: | E: an record to be removed |
| Output: | none |

| | |
|---|---|
| D1 [Find node containing record] | Invoke **FindLeaf** to locate the leaf node L containing E. Stop if E is not found; |
| D2 [Delete entry] | Remove E from L; |
| D3 [Propagate changes] | Invoke **CondenseTree**, passing L; |
| D4 [Shorten tree] | If the root node has only one child after the tree has been adjusted, make the child the new root. |

Algorithm FindLeaf find the leaf node containing the entry to be removed.

| Algorithm **FindLeaf** | |
|---|---|
| Input: | T: the root node of an R-tree<br>E: an entry to be removed |
| Output: | a leaf node containing E |
| FL1 [Search subtree] | Set N to be T.<br>If N is a non-leaf node, check each entry F in N to determine if F.I overlaps E.I. For each such entry, invoke **FindLeaf** on the tree whose root is pointed to by F.ptr until E is found or all entries have been checked. |
| FL2 [Search leaf node for record] | If N is a leaf node, check each entry to see if it matches E. If E is found, return N. |

| Algorithm **CondenseTree** | |
|---|---|
| Input: | L: a leaf node from which an entry has been deleted |
| Output: | none |
| CT1 [Initialize] | Set N to be L; Set Q, the set of eliminated nodes, to be empty; |
| CT2 [Find parent entry] | If N is the root, go to CT6. Otherwise, let P be the parent of N, and let $E_N$ be N's entry in P; |
| CT3 [Eliminate under-full node] | If N has fewer than m entries, delete $E_N$ from P and add N to set Q. |
| CT4 [Adjust covering rectangle] | If N has not been eliminated, adjust $E_N$.I to tightly contain all entries in N; |

| | |
|---|---|
| CT5 [Move up one level in tree] | Set N=P and repeat from CT2; |
| CT6 [Re-insert orphaned entries] | Re-insert all entries of node in set Q. Entries from eliminated leaf nodes are re-inserted in tree leaves. Entries from eliminated interior nodes are placed one level higher in the tree so that leaves of their dependent subtrees are on the same level as leaves of the main tree. |

#### 2.2.2.4 Split

In order to add a new entry to a full node with M entries, it is necessary to divide the collection of (M+1) entries between two nodes. [Gutt84] introduced two version of applicable split algorithms: quadratic-cost ($O(M^2)$) algorithm and linear-cost (O(M)) algorithm with respect to the maximum number (M) of entries in a node. In [Gutt84], experiments show that both algorithms yield the same retrieval performance. Later research [Beck90] has determined that in the conditions of more extensive trials with wider-ranging data, the quadratic cost algorithm outperforms the linear cost one.

Algorithm **QuadraticSplit**

| | |
|---|---|
| Input: | L: a leaf node or non-leaf node with (M + 1) entries |
| Output: | LL: a new leaf node or non-leaf node |
| QS1 [Pick first entry for each group] | Apply algorithm **pickSeeds** to choose two entries to be the first elements of the groups. Assign each to a group |
| QS2 [Check if done] | If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number m, assign them and stop; |
| QS3 [Select entry to assign] | Invoke algorithm **PickNext** to choose the next entry. Add it to the group whose covering rectangle will have to be |

| | |
|---|---|
| | enlarged less to accommodate it. Resolve ties by adding the entry to the group with smaller area, then to the one with fewer entries, then to either. Repeat from QS2. |

Algorithm PickSeeds selects two entries to be the first elements of the groups.

| Algorithm **PickSeeds** | |
|---|---|
| Input: | L: a node with (M + 1) entries |
| Output: | a pair of entries |
| PS1 [Calculate inefficiency of grouping entries together] | For each pair of entries E1 and E2 , compose a rectangle J including E1.I and E2.I.<br>Calculate d = area(J) – area(E1.I) – area(E2.I); |
| PS2 [Choose the most wasteful pair] | Choose the pair with the largest d. |

Algorithm PickNext selects one remaining entry for classification in a group.

| Algorithm **PickNext** | |
|---|---|
| Input: | Group1: a node with previously distributed entries from L<br>Group2: new node with previously distributed entries from L |
| Output: | an entry |
| PN1 [Determine cost of putting each entry in each group] | For each entry E not yet in a group, calculate d1 = the area increase required in the covering rectangle of Group 1 to include E.I. Calculate $d_2$ similarly for Group 2; |
| PN2 [Find entry with greatest preference for one group] | Choose any entry with the maximum difference between E1 and E2. |

## 2.3 Templates, Generic Programming and STL

### 2.3.1 Templates

Templates, also called parameterized types, are among the most powerful features of C++ since they provide a generic way to develop reusable code. Templates are

mechanisms for generating functions and classes based on type parameters. They provide us with behavior parameterization, code optimization, and information parameterization.

C++ requires programmers to declare variables, functions, and most other kinds of entities using specific types such as char, int, float, double, etc. However, a lot of code looks the same for different types. Especially when implementing algorithms, or data structures, the code looks the same despite the type used. "*Templates are a good candidate for coping with combinatorial behaviors because they generate code at compile time based on the types provided by the user*" [Alex01]. Templates allow developers use a single data structure or algorithm to handle many different types of parameters, instead of implementing the same behavior again and again for each specific data type.

C++ provides two basic types of templates: *function templates* and *class templates*. They are functions or classes that are written for one or more types not yet specified. Specified types are passed as arguments explicitly or implicitly later.

#### 2.3.1.1 Function templates

Function templates are generic functions that can be used with arbitrary types. They are implemented like regular functions, except they are prefixed with the keyword template by angle brackets "< >". They provide a way to parameterize the arguments or return types of a function, so that they can represent a family of functions. Function templates provide a functional behavior that can be called for different types. The definition of a template function depends on an underlying data type. For example:

```
template <class T>
T max(const T& a, const T& b)
```

```
{
    return  a> b? a: b;
}
```

The first line is the *template prefix*, which tells the compiler that template parameter T is a data type that will be explicitly specified later with an actual type. Template functions are used in exactly the same way as regular functions. The compiler automatically instantiates the needed versions of *max* function in terms of the given actual type. The STL algorithms are implemented as function templates.

#### 2.3.1.2 Class Templates

A template class is a class that depends on an underlying data. They provide a way to parameterize the types within a class. The definition of a template class starts with a template prefix:

```
template <class T>
class vector
{
        ...
 };
```

Inside the class, the template parameter T can be used as the name of a data type. When the class is instantiated into a concrete class, an actual data type (such as int and double) will be substituted for T. For example, to create a dynamic array-like data structure, we now just need to instantiate the class with an actual data type:

```
vector<int>     intVect;
vector<double>  doubleVect;
```

Function and class templates are particularly useful in the Standard Template Library (STL), since they make the generalization possible without sacrificing efficiency. Without templates, there were only two approaches for creating generic components

in C++; one is to implement the data structure as a C struct with void pointers to hold the data; another is to implement the data structure using a C++ class hierarchy and use inheritance and virtual functions. However, neither of these approaches could provide the efficiency required for a component in a standard library that would be put to general use.

### 2.3.2 Generic Programming

Generic programming is "programming with concepts" [Mus03]. [HypDic] defines generic programming as "a programming technique, which aims to make programs more adaptable by making them more general." Generic programming is a sub-discipline that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.

Generic programming offers the ability to parameterize functions and classes with arbitrary data types. This new technique allows us to focus on the nature of algorithms rather than on their implementations for special types.

### 2.3.3 Why Generic Programming?

Generic programming makes it possible to write programs that solve a class of problems once and for all, instead of writing new code over and over again for each different instance. Generic programs are natural candidates for incorporation in library form, and the increased reliability, due to the fact that they are stripped of irrelevant detail which often makes it easier to construct.

Generic programs often embody non-traditional kinds of polymorphism; ordinary programs are obtained from them by suitably instantiating their parameters. In contrast with traditional programs, the parameters of a generic program are often quite rich in structure.

## 2.4 The STL Style

The *Standard Template Library* (STL), which is developed by Alexander Stepanov and Meng Lee, is a general-purpose C++ programming library accepted by the ANSI/ISO committee as part of the C++ Standard (ISO/IEC (1998)). It makes heavy use of the template mechanism for parameterize components.

### 2.4.1 Why Use the STL?

#### 2.4.1.1 Code Reuse

The STL promotes software reuse. The meaning of *reusable* is roughly "widely adaptable but still efficient". The purpose of the STL has been to explore methods of developing and organizing libraries of generic and reusable software components [Kern98]. Relatively, the STL is a small library which achieves a remarkable degree of reuse through its basis in the principles of generic programming and use of C++ templates. Thus it has a particularly clear shape.

The STL itself is a collection of reusable components. The most used data structure, array, queue, list, etc., are implemented, debugged and tested by experienced programmers. Designing with existing components can significantly reduce the period of the software development and increase the reliability of the products. The time needed for implementation of many large systems might be dramatically reduced by codes simply imported from the STL.

The STL are type independent that means you can create vectors of char, integer, float, double or any other user-defined classes or types. The container classes are independent of the memory model that keeps the code portable across the numerous memory models found on the various operating systems in use today. The way the author of the STL kept the code memory model-independent was through the use of allolcators.

The STL is written in such a way that different algorithms can work for different containers, without the need to duplicate the code and rewrite the same algorithm again and again for every data structure and object type. In addition, programs using a standardized library are more portable since all compiler producers will be oriented towards the standard.

#### 2.4.1.2 Smaller Codes

The STL provides approximately fifty different generic algorithms and about a dozen major data structures. This separation helps reducing the size of source code and decreasing some of the risk that similar activities have dissimilar interfaces. If it were not for this separation, for example, each of the algorithms would have to be re-implemented in each of the different data structures, requiring several hundred more member functions. Based on the STL, the programmer can then focus on the problem at hand instead of implementing and debugging their own codes that other developers have already coded hundreds and thousands times.

The STL hides complex, tedious and error prone details. For example, using the default allocator in the STL container, you can use different standard containers

without any coding with allocator. You do not have to worry about allocating and freeing memory.

#### 2.4.1.3 Flexibility

In STL, the iterators decouple algorithms from containers. This orthogonal structure brings the STL its power, flexibility, and extensibility. The developers, who are implementing new algorithms utilizing one of the STL iterators, are guaranteed that their algorithms will work with existing containers as well as those that have not yet been developed. Another advantage of the separation is that such algorithms can be used with not only STL containers, but also conventional C++ pointers, strings and arrays. Because C++ arrays are not objects, algorithms encapsulated within a class hierarchy seldom have this ability.

The other three STL components also contribute to the flexibility; function objects encapsulate a function as an object; adaptors provide an existing component with a different interface; allocators encapsulate the memory model of the machine.

#### 2.4.1.4 Efficiency

All the STL components themselves are written with the most efficient implementation possible, and strict attention to time complexity of algorithm is paid. Therefore, they allow the system itself to be efficient. The users do not have to write their classes and algorithms that cost much time to write, debug and test. Furthermore, there are no run time losses since the evaluation of templates is carried out at compile time.

The STL in particular provides a low-level approach to application developments. This low-level approach can be useful when specific programs require an emphasis on efficient coding and speed of execution.

### 2.4.2 The STL Components

The uniform design of the interfaces allows a flexible cooperation of components and also the construction of new components in the STL-conforming style. Since the STL "provides a set of well-structured generic C++ components that work together in a seamless way", it "ensures that all the template algorithms work not only on the data structure in the library, but also on built-in C++ data structures" [Step95]. The STL is therefore a universally usable, flexible, and extensible library, which offers many advantages with respect to quality, efficiency, and productivity. Plus, it has an elegant, consistent, and easy to comprehend architecture. The successful concept has already been copies, as the Java Generic Library shows [Brey02]. Figure 2-2 [Sanc00] illustrates an overview of the STL.

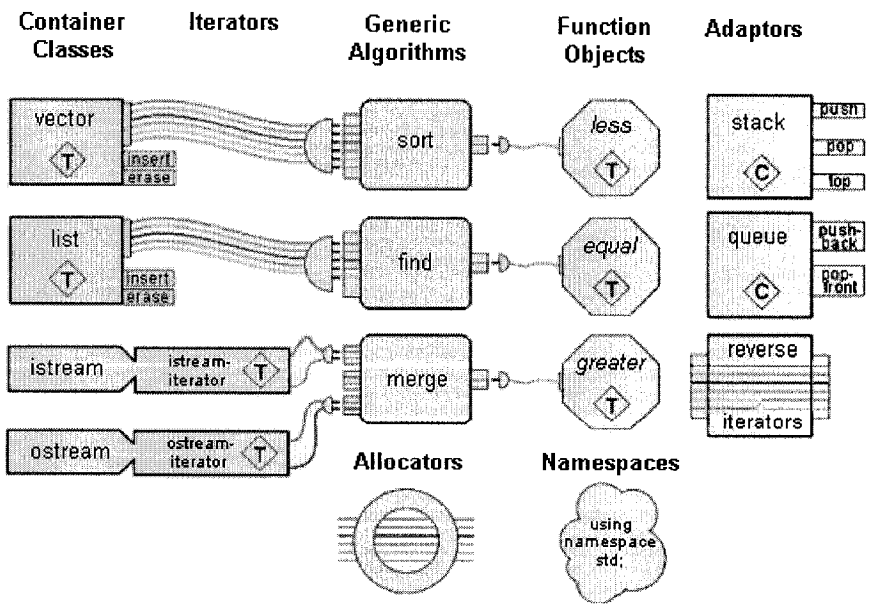



Figure 2-2 STL Overview

The STL provides a set of reliable and well-proven components that can be broadly classified into six categories: template-based containers, generic algorithms, iterators, functors (function objects), adapters, and allocators [Wise95]. The first three components, namely container, algorithm, and iterator, can be considered the foundational and core components of the library. The other three components, i.e. allocators, adaptors, and function objects, are used for support.

*Containers* are data structures that manage collections of elements and are responsible for the allocation and deallocation of those elements. Unlike traditional data structures, STL containers have only constructors and destructors along with a minimal set of operations for inserting and deleting elements.

*Algorithms*, used for processing elements in containers, are decoupled from the specific type of container that the algorithm might currently be working with. They can only interact with containers by accessing the corresponding iterators.

*Iterators* are objects to provide a way to access the elements of an aggregate object (container) like points without exposing its underlying representation [Gamm95]. They act as the glue that allows algorithms to be applied to containers. The design of separating containers and algorithms leads to a very clear orthogonal component structure [Wise95] that is shown in Figure 2-3. It facilitates the use of generic algorithms on almost all containers that conform to STL conventions.

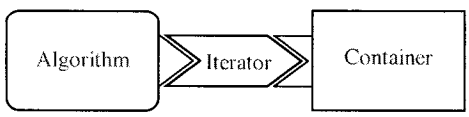



Figure 2-3 Orthogonal Component Structure

*Function objects*, or *functors*, are objects which behave like functions but have all the properties of objects. They can be generated, passed as arguments, or have their state modified.

### 2.4.2.1 Containers

Containers are objects that store other objects. They deal with the allocation and deallocation of memory through constructors and destructors, and control insertion and deletion of the stored objects [Step95]. The STL provides different kinds of containers that are all formulated as template classes. Each container offers unique advantages. The most efficient container type for a specific task depends on the way an application manipulates its data and the nature of operations it wants to perform on the objects. The STL containers are divided into two broad families: sequential containers and sorted associative containers (Figure 2-4).

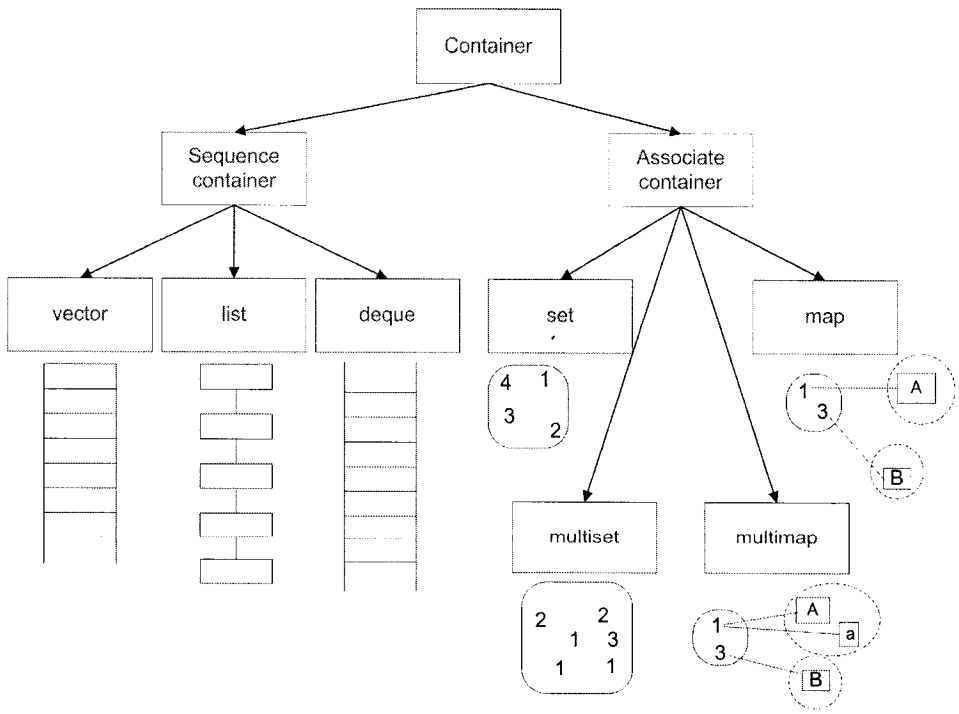



Figure 2-4 STL containers

**Sequence containers**

Sequence containers include vector, list, and deque, which contain elements of a single type organized in a strictly arrangement [Kern98].

| | |
|---|---|
| Vector | provides a dynamic array structure with access to any element. It expands at the end as necessary to accommodate additional elements. Inserting and deleting elements at the end is fast. |
| deque | provides a dynamic array structure with random access, fast insertion and deletion of elements at front and back. It is very slightly slower than vector because of an extra level of indirection. |
| list | provides linear time access to a sequence of varying length. There is no random access to the elements. Insertion and deletion anywhere is fast. |

**Associative containers**

Associative containers include set, multiset, map, multimap. They allow for the efficient retrieval of data based on their keys since they are implemented as red-black trees.

| | |
|---|---|
| set<T, Compare> | supports unique Ts, and provides for fast retrieval of the Ts themselves. |
| multiset <T, Compare> | allows duplicate Ts, and provides for fast retrieval of the Ts themselves. |
| map<Key,T,Compare> | supports unique keys, and provides for fast retrieval of another type T based on the keys |

multimap<Key,T,Compare> supports duplicate keys, and provides for fast retrieval of another type T based on the keys.

### 2.4.2.2 Iterators

Containers are only storage cabinets to put objects in. They do not provide built-in access to their elements. Although, using an index and the subscript operator can access the elements of a vector, but this is not true for other containers such as list. Stated differently, using the subscript operator and indexes is not a generic way to access an element in a container.

In order for a STL algorithm to work on any STL container in a uniform manner, some truly generic means of accessing the elements in a container is required. For this purpose, STL provides objects called iterators that can "point at" an element, access the value stored there, and move from one element to another.

Iterators are pointer-like objects that STL algorithms use to traverse the sequence of objects stored in a container. Iterators are of central importance in the design of STL and give the STL its most flexibility because they act as intermediaries between containers and generic algorithms. They decouple the components so that new algorithms can be written without concern for how data sequences are stored, and they enable containers to be written without having to code a large number of algorithms on them. Each STL container also declares its own group of iterator types that can be used to define iterator objects and at least two methods that return iterators:

begin() returns an iterator positioned at the first element

end() returns an iterator positioned at the position following the last element

An iterator may either point to an element of this container, or beyond it, using the special past-the-end value illustrated in Figure 2-5.

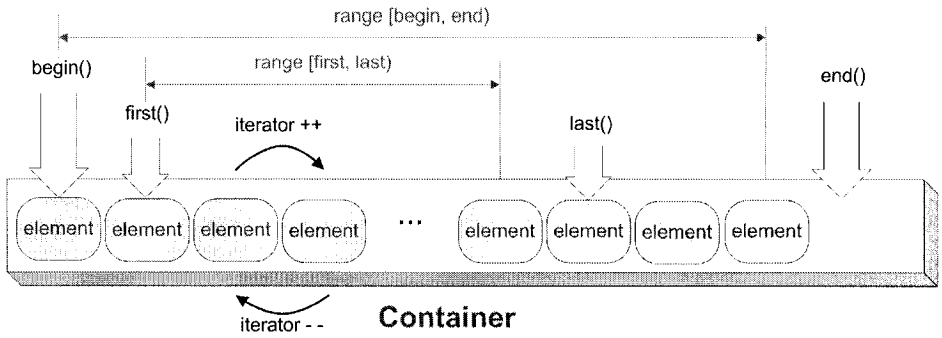



Figure 2-5 Iterator Range

Iterators are the cornerstone of the STL design and give the STL its most flexibility. Instead of developing algorithms for a specific container, five categories of iterators are developed to makes it possible to use the same algorithm with a variety of different containers. Requirements for a given iterator category are specified by a set of valid expressions for iterators in that category as well as precise semantics describing their usage. In addition, iterators must satisfy complexity requirements, which ensure that STL algorithms will work correctly and efficiently.

The STL provides a hierarchy of iterator categories as shown in Figure 2-6. Iterators at the top of the hierarchy are the most general and powerful; those at the bottom are the most restricted and have fewer requirements.

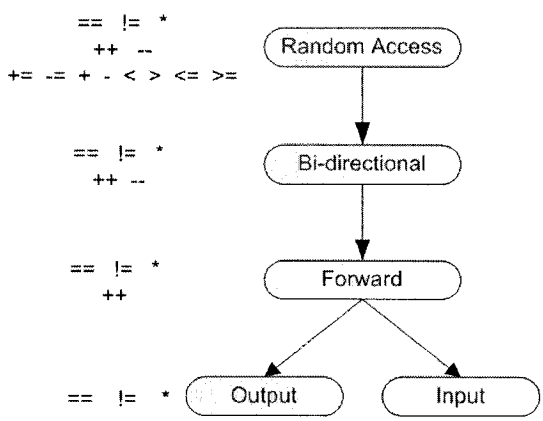



Figure 2-6 Iterator Hierarchy

#### 2.4.2.3 Algorithms

The STL algorithms are template functions to perform operations on containers through iterators. Instead of taking containers as arguments, they take iterators that specify part or all of a container. Thus not only can the functions work with different types of standard containers, but also work on custom containers, as long as they have iterator types satisfying the assumptions on the algorithms [Step95].

The STL provides a rich set of generic algorithms that fall into four broad categories, based loosely on their semantics [Muss96]:

- Nonmutating sequence algorithms: operate on containers without modify them. Examples: find, equal, and search;
- Mutating sequence algorithms: modify the contents of the container on which they operate. Examples: copy, fill, shuffle, remove, and swap;
- Sorting-related algorithms: rearrange the elements' positions based on sorting criteria. Examples: sort, and partial_sort;
- Generalized numeric algorithms: obviously conduct numerical manipulation of the data in the container; Examples: accumulate, and inner_product;

#### 2.4.2.4 Allocators

Allocators are components responsible for memory management. Allocators encapsulate information about memory allocation model the program is using. They provide a low-level interface that permits efficient allocation of many small objects.

A container is given an allocator when it is constructed. Whenever an element is inserted or removed, a container uses its allocator to allocate and deallocate the

memory automatically without knowing anything about the memory model of the computer. Different memory allocation models take different approaches to obtaining memory from the operating system. The allocator class encapsulates information about pointers, constant pointer, references, constant reference, size of objects, difference types between pointers, allocation and deallocation functions, as well as some other functions.

The STL provides different allocators and you can even create your own custom allocators. However, default allocator supplied with STL implementation is sufficient for most programmers' needs.

#### 2.4.2.5 Adaptors

An adapter in our daily life is a component that modifies the interface of another component. In computer science, an adaptor is a template class that provides an existing class with a new interface. The intent of the adapter (pattern) is to convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interface [Gamm95]. The STL adapters are defined as template classes that taken a component type as a parameter. The STL provides container adaptors, iterator adaptors, and function adaptors.

**Container Adaptors**

It is usually needed to create a new container from an existing and general container. We can use the container adaptors to create a new container by mapping the interface of the container to that of the new container. Since most of the functionality is already provided by the existing container, we can do it without much additional effort. The

STL provides three container adaptors that are stack, queue, and priority queue container adaptors.

**Iterator adaptors**

The iterator adaptors are STL components that can be used to change the interface of an iterator component, and to extend the functionality of an existing iterator. The STL provides three iterator adaptors that are reverse, insert, and raw storage iterators.

**Function Adaptors**

Function Adaptors are template classes that provide an existing class with a new interface. They help us construct a wider variety of function objects without directly constructing a new function object type with a class (or struct) definition. The STL provides three categories of function adaptors: binders, negators, and adaptors for pointer to functions [Kern98].

#### 2.4.2.6 Function objects

Although pointers to functions are widely used for implementing function callbacks, C++ offers a significantly superior alternative to them, namely function objects (also called "functors"). Function objects are entities that can be applied to zero or more arguments to obtain a value and/or modify the state of the computation [Muss96]. Object of any class that overloads the function call operator (operator()), satisfies this definition and behaviors like a regular function.

Using functors instead of a function pointer is more resilient to design changes because they can be modified internally without changing their external interface. Functors give us the ability to separate methods from objects.

A functor can also have data members that store the result of a previous call, while ordinary functions have to work with a global or local static variable, which have some undesirable characteristics. In addition, compilers can enhance the performance further by inline of a call made through a function object. In contrast, it is nearly impossible to inline a function call made through a pointer. Therefore, using function objects rather than function pointers allows the STL to generate more efficient and flexible code.

The STL provides many different predefined function objects for the most common cases, including arithmetic operations, comparisons, and logical operations. It is possible to perform very sophisticated operations simply by combining these predefined function objects and function object adaptors.

More introductions have to be given to the predicate for this thesis is much related to it. Predicates are functions or functors that return a Boolean value. A predicate cannot change the state due to a call. A copy of the predicate should also have the same state as the original one. A predicate should always return the same result for the same value. When an algorithm takes an *Unary Predicate* pred as its argument and iterator *it* of the container, the operation *pred(*it)* should work correctly. Similarly, a B*inary Predicate* should work correctly in the form of binary_pred(*it1, *it2).

## 2.5 Design Pattern

> "*... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*" [Gamm95]
>
> – Christopher Alexander

Patterns were originally conceived by Christopher Alxander and presented in his book "A Pattern Language" to provide structure for a theory of living architecture. Eric Gamma [Gamm95] describes design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." Shortly, design patterns are general solutions to frequently occurring architecture/design problems in contexts.

In software engineering, patterns attempt to describe successful solutions to common software problems by experts in software architecture and design. They make hidden design knowledge explicit and available, provide the best solution to the common problems so that people can get a head start on their own problems, and provide a common point of reference during the analysis and design phase of a project. This section simply introduces some design patterns associated with our design and implementation.

### 2.5.1 Composite

The intent of composite pattern is to compose objects into the tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. The structure of the composite pattern is illustrated in Figure 2-7. [Gamm95]

The *Component* declares the interface for objects in the *Composition* and for accessing and managing its child components. In addition, it implements default behavior for the interface common to all classes. The *composite* stores child components, defines behaviors for components having children, and implements

child-related operations. The *leaf* represents leaf objects in the *Composition* and defines behavior for primitive objects in the Composition. [Gamm95]

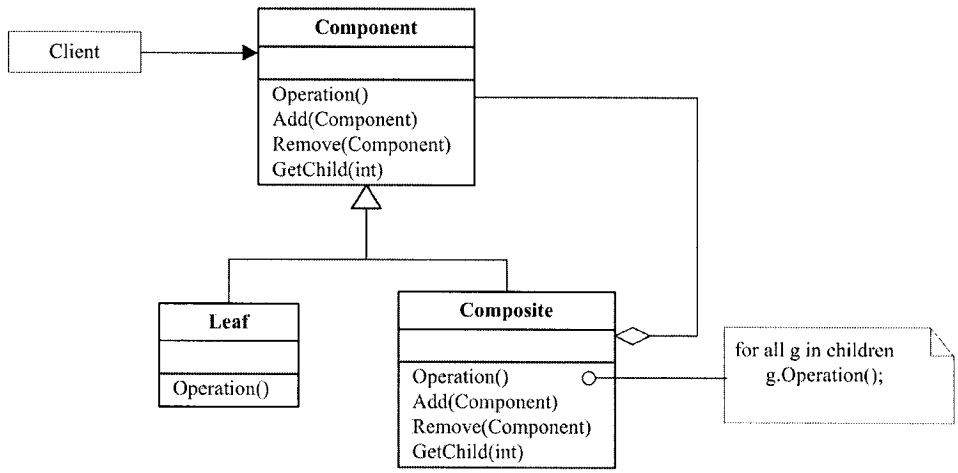



Figure 2-7 Structure of Composite Pattern

By using this pattern, we can build complex objects by recursively composing similar objects in a tree-like manner and allows the objects in the tree to be manipulated in a consistent manner. The key to the composite pattern is an abstract class that represents both primitive and their containers.

Although the composite class implements the *Add* and *Remove* operations for managing children, we still have a question that which class should declare these operations in the Composite hierarch. If we define the child management interface at the root of the class hierarchy, we can treat all components uniformly. However, it cost us safety since clients may try to add and remove object from leaves, which is meaningless. If we define child management in the *Composite* class, we get safety because any attempt to add or remove object from leaves will be caught at compile-time. However, we lose transparency because of the different interface between leaves and composites.

One approach proposed by [Gamm95] is to declare an operation Composite* GetComposite() in the *Composite* class. *Component* provides a default operation that returns a null pointer. This operation is redefined in the *Composite* to return itself through the *this* pointer (Figure 2-8). Actually, this is the same story with casting method pattern introduced below.

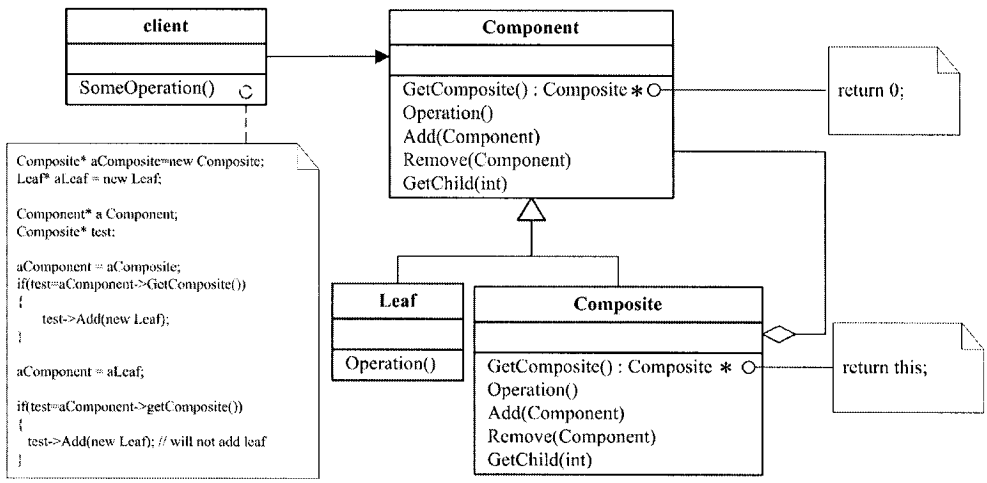



Figure 2-8 an Alternative Form of Composite Pattern

### 2.5.2 Casting Method

The intent of Casting-method pattern [Scot92] is to represent an operation to dynamically and quickly obtain a type-safe reference to a subclass in an inheritance hierarchy. The structure of the casting method is illustrated in Figure 2-9.

The Casting method pattern uses inheritance to allow subclasses to return references to themselves. This pattern is applicable when there is a need to obtain a downcast class reference from a base class, or run time type information is not available, or real time constraints require the fastest and safest solution possible.

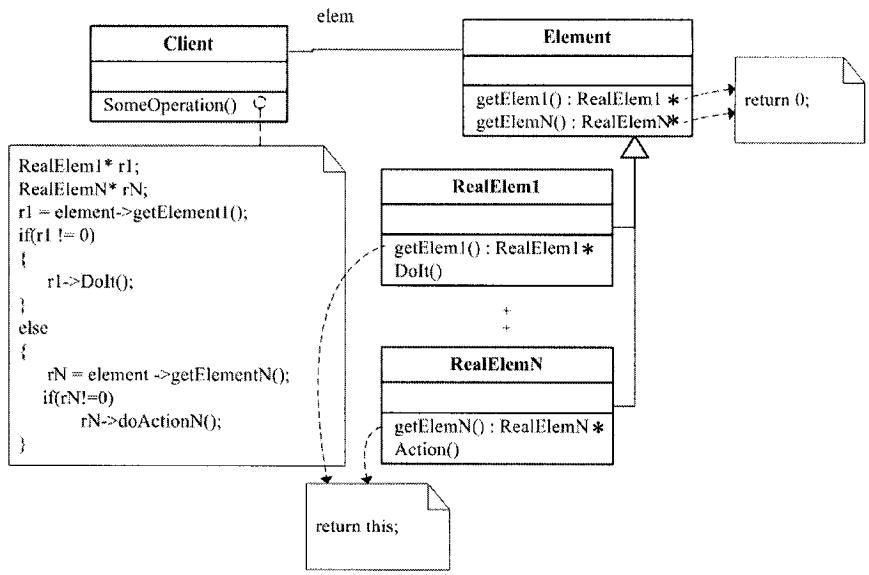



Figure 2-9 Structure of Casting Method Pattern

Although the syntax of casting method pattern is rather simple and easy to understand, it executes faster than any other technique except blind cast. However, it is difficult to add new elements since every time a new element is added, the base class must be modified by adding a new GetElementN() method. Moreover, all users of any classes derived from the base class must be recompiled whenever any new element class is added. Therefore, it is well applicable for cases with a small and stable number of subclasses.

### 2.5.3 Proxy

The intent of the Proxy pattern is to provide a surrogate or placeholder for another object to control access to it [Gamm95]. Classes for proxy objects are declared in a way that usually eliminates client object's awareness that they are dealing with a proxy. The proxy maintains a reference that lets the proxy access the real subject. It controls access to the real subject and may be responsible for creating and deleting the subject. The proxy structure of the proxy pattern is illustrated in Figure 2-10

[Gamm95]. Figure 2-11 shows a possible instance diagram of a proxy structure at run-time.

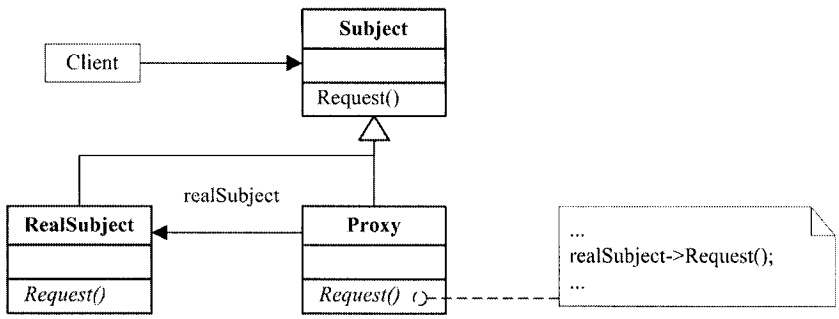



Figure 2-10 Structure of Proxy Pattern

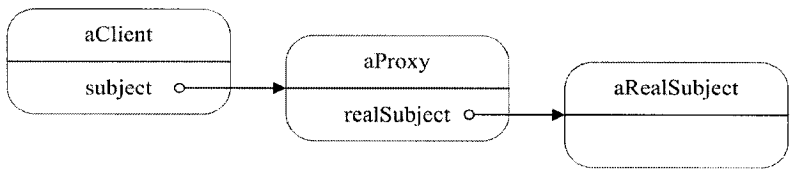



Figure 2-11an Instance of Proxy

Proxy is a very general pattern that occurs in many other patterns, but never by itself in its pure form. It is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. In the implementation of the tree structure, a smart reference is replaced for a bare pointer that performs additional actions when a node object is accessed.

### 2.5.4 Singleton

The Singleton is probably the most widely used design pattern. Its intent is to ensure that a class has only one instance, and provides a global point of access to it [Gamm95]. It's important for some classes to have exactly one instance, and it must be accessible to clients from a well-known access point. The Singleton class hides the

operation that create the instance behind a class operation, a static member function called *Instance()*, to guarantee only one instance is created. The clients access the singleton exclusively through the Instance member function. The structure of the singleton pattern is illustrated in Figure 2-12. [Gamm95]

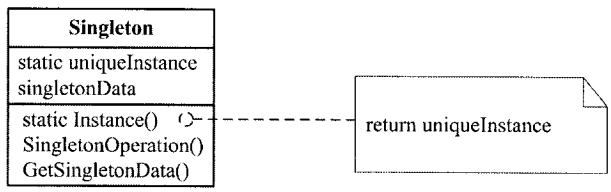



Figure 2-12 Structure of Singleton Pattern

### 2.5.5 Serializer

The intent of serializer pattern is to read arbitrarily complex object structures from and write them to varying data structure based backends[RIE97]. It lets clients efficiently store and retrieve objects from different backends, such as flat files, relational databases and RPC buffers. Participates are the Reader/Writer, the Concrete Reader/Concrete Writer, the Serializable interface, the Concrete Elements, and different Backends. The structure of the serializer pattern is showed in Figure 2-13.

The Reader part of the pattern builds an object structure by reading a data structure from a backend. The Writer part of the pattern writes an existing object structure as a data structure to a backend. The Reader and Writer hide the Backend and external representation. Both parts together constitute the Serializer pattern.

One advantage of this pattern is that the application class itself has no knowledge about the external representation format that is used to represent their instances. As a

result, introducing a new representation format or changing an old one would not require changing any class in the system.

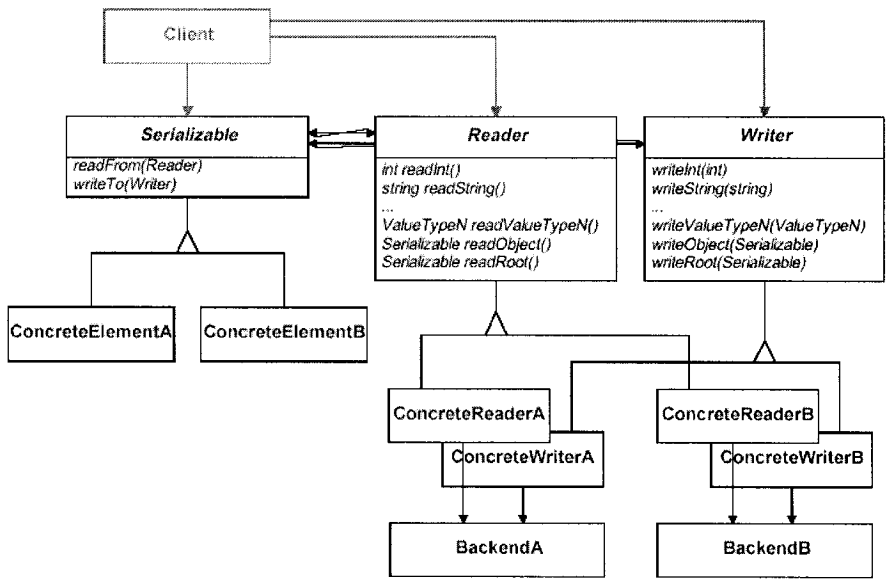



Figure 2-13 Structure of the Serializer Pattern

## 2.6 Indexing Frameworks

### 2.6.1 The GiST Framework

The theory and implementation of the GiST (Generalized Search Tree) framework was realized by Joseph. M. Hellerstein and his group at the University of California, Berkeley [Heller95]. It is an indexing framework that subsumes most of the prior indexing research in Database and Geographic Information Systems. The GiST framework enables an access method developer to build any kind of balanced index tree on any kind of data by implementing some specific method for insertion, deletion and search. Thereby it unifies disparate structures such as B+-trees, R-trees, and RD-trees in a single piece of code.

The GiST allows users to develop indices over any kind of data, supporting any lookup over that data efficiently. To make a GiST work, the users just have to figure out what to represent in the keys, and then write the following four special methods, which include Consistent, Union, Penalty, and PickSplit, for the key class that help the tree do insertion, deletion, and search.

### 2.6.2 An Framework of Indexing Structure of KIA

Under the framework of Know-It-All [Butler02] are some sub-frameworks that include the framework of indexing structure. In 2001, Ashraf Gaffar [Gaff01] designed a framework for database indexes using STL components. Figure 2-14 is the component layout of his design. "These components will provide the necessary index structure, and manage the access and storage of the index" [Gaff01].

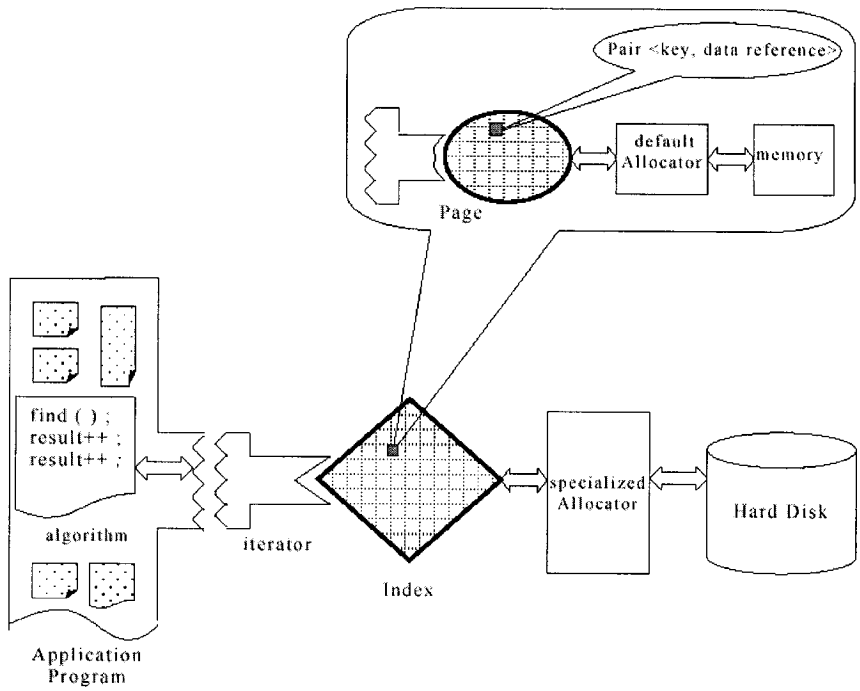



Figure 2-14 The Components Layout of Gaffar's Design

The index is a container of template type page that is passed as template type in the implementation phase. This makes the index independent of the page type so that the index can work with any page type. The page is also a container of pairs of key and data references. Again the page is independent of the type of key and data reference by having them as templates. Therefore, the page can work with any key type and data reference type that are passed to it.

The only way to interact with the container is through its iterator, which provides controlled access to the element of the container. The application therefore has to use an index iterator to iterate through pages, one page at a time.

An applicable database system should have the ability to save the data on non-volatile storage, like a hard disk or other mass storage media since the system is normally too large to fit entirely in memory. To manage the storage of the elements, the container will use a "specialized allocator that takes the responsibility of retrieving the page from the storage into memory for access and controlling the different objects accessing the same page simultaneously... This will ensure data integrity by applying a suitable access and locking policy" [Gaff01]. Figure 2-15 [Gaff01] shows the class diagram of the index system.

By using the STL components, [Gaff01] produced a generalized index framework that is adaptable to different data/key type, different query, and different database application domains.

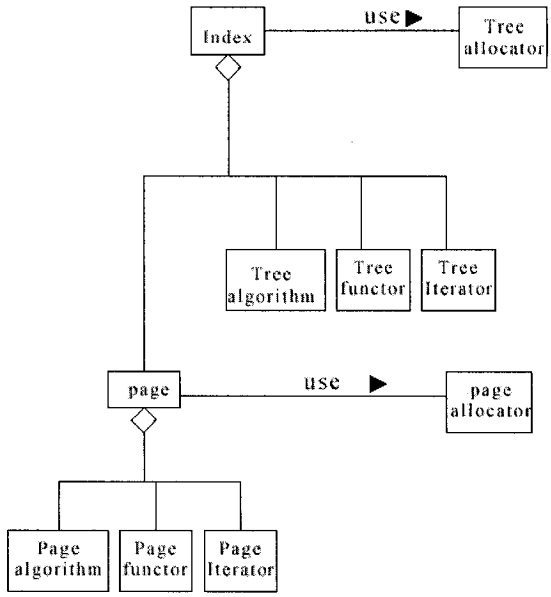



Figure 2-15 Main Classes of Gaffar's Design

## 2.7 Some Existing R-tree Implementations

The R-tree has been implemented hundreds times after the original R-tree theory was developed. Its most popular variants (R*-tree, X-tree) are proposed for index data in low or high-dimensional spaces. Some implementations of the R-tree are developed by C language and thereby not easy to read and reuse. These include the original algorithm and test code by Antonin Guttman in 1983, the implementation by Timos Sellis' group [Rtreeportal], and the GiST (version 1.0).

Some implementations are object-oriented designed. The Spatial Index Library is a well-developed package by University of California [UCalifornia]. It provides a general framework for developing spatial indices. The relationship of the main classes is illustrated in Figure 2-16. It applied the composite design pattern with maximizing the component interface, which will be discussed in Chapter 3.

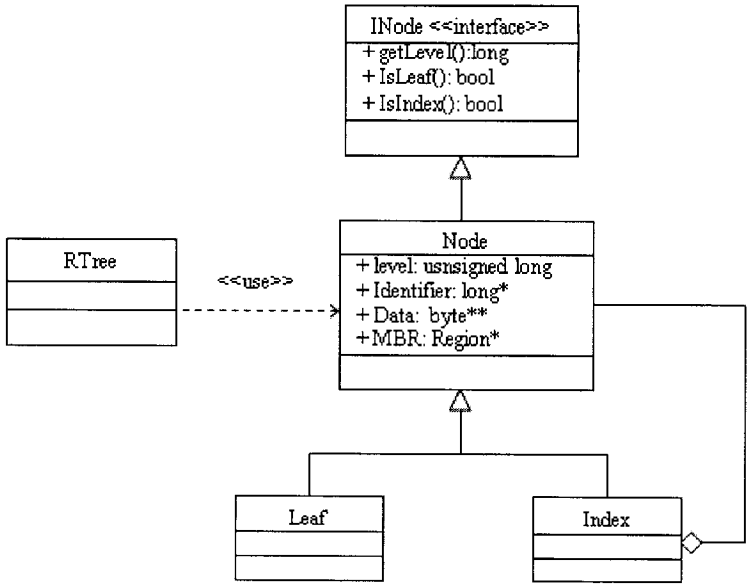



Figure 2-16 Main Classes of SpatialIndex

# Chapter 3 The R-Tree Design

The design of R-tree in this thesis is mainly based on the framework developed by Ashraf Gaffar [Gaff01]. Some problems in this framework are addressed or improved.

## 3.1 Use Case

The index is a part of the database. The R-tree is a balanced tree structure that can be customized to suit different application, applied to insert, delete and query data in the database. Figure 3-1 shows the diagram of use case.

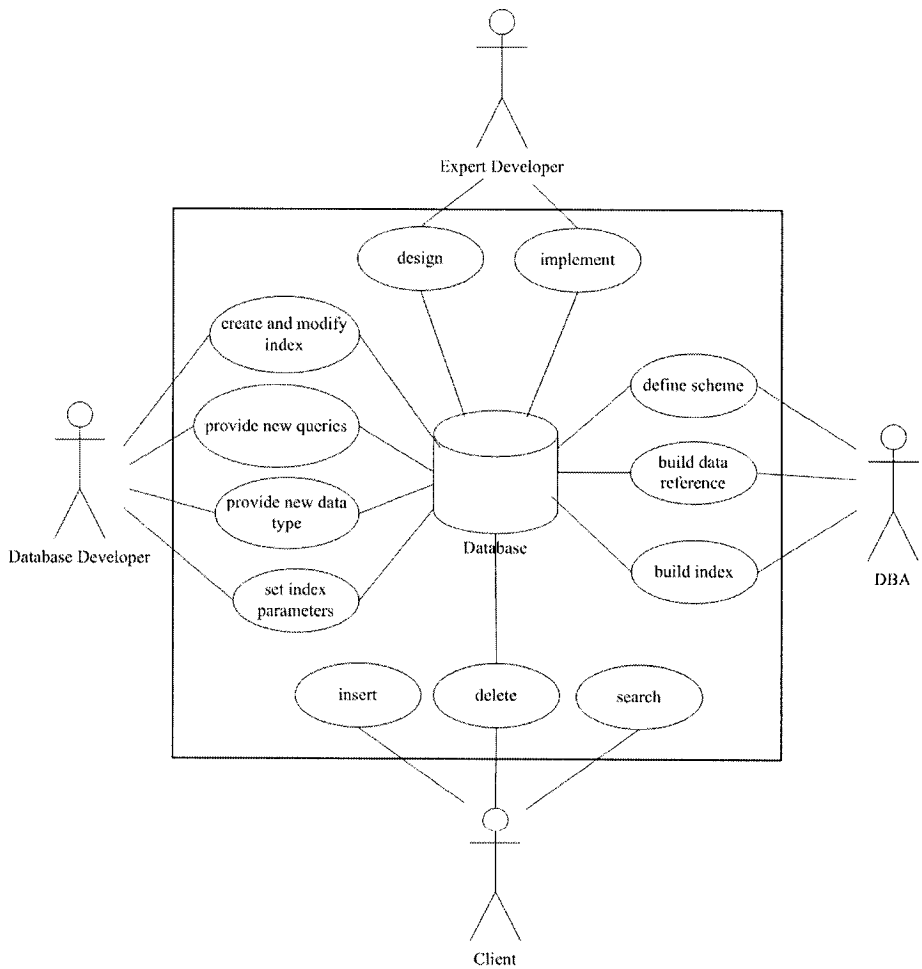



Figure 3-1 Use Case

### 3.1.1 Expert Developer

Expert developer is an experienced programmer who designs the database system to satisfy domain expert requirements and then implements the design using a programming language.

### 3.1.2 Database Developer

Database developer is a domain expert who is responsible for customizing an existing database system and setting the parameter to suit the system platform and the application variables. Database developer makes the database system to be able to support a new data or query type, or a completely new index structure with new access method by modifying some components or replacing some parts.

### 3.1.3 Database Administrator

Database administrator (DBA) is responsible for building the database system that includes building the scheme, the physical tables and the indexes used to access them.

### 3.1.4 Client

Client is the application that is using the system to search, insert data into or delete data from the database.

## 3.2 Relation between Index and Database Data

The index is a major component of a database system. Instead of providing us directly with data, it tells us where the data is. The index is built on the physical data indirectly through a data reference file, which consists of a pair of key and data reference and constitutes the data level of the index. The separation of data reference

from physical data allows multiple indexes to be created on the same dataset and guarantees that changes in the tuple physical locations will not affect the existing indexes. Figure 3-2 [Gaff01] shows the relation between the index and the database data.

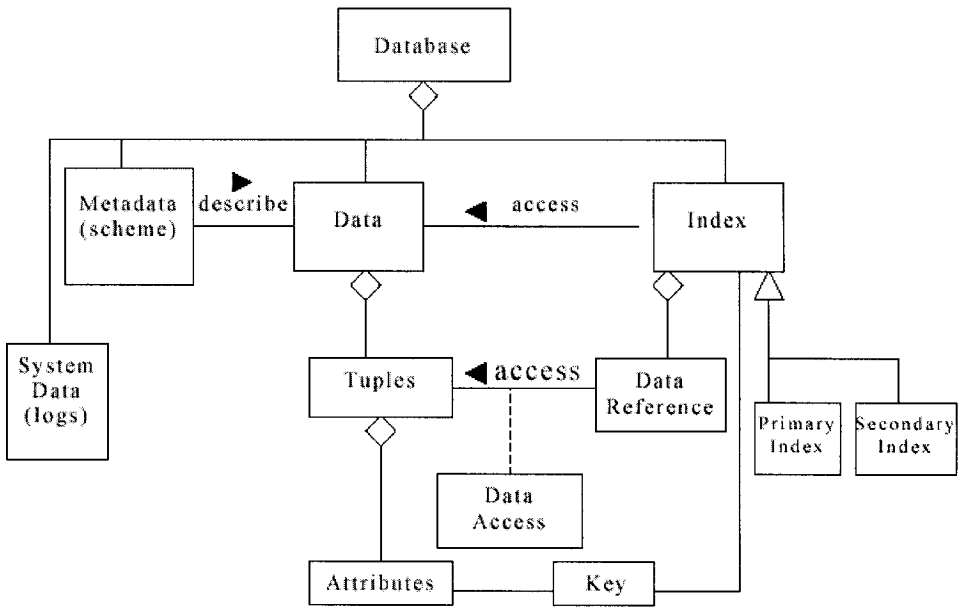



Figure 3-2 the Relation between Index and Database Data

In order to build an index, we start with physical data, build the reference file. The data access component will be responsible for binding the reference to the data tuples on the physical storage. Consequently the index is created.

Answering a query will start from the index for a certain key using some comparison criteria. We will find the way through the index down to the index leaf level, where the index data is stored. We then refer to the physical database access mechanism to get the real data we are looking for.

## 3.3 The Original Design

The original design is the framework of indexing structure by [Gaff01]. By using the STL components, [Gaff01] produced a generalized index framework that is adaptable to different data/key type, different query, and different database application domains.

However, there are some problems in the framework for developing tree-like indexes. Firstly, a tree-like index has two kinds of pages, namely index page and leaf page. Only the leaf pages are visible to the users. The application program uses index iterators to interact with the leaf page without awareness of the existence of the index pages.

[Gaff01] tries to use a special allocator to take responsibility of the management of the container's access and storage. STL itself does not provide any functionality for the serialization of its elements since the container's elements are stored in memory. Since the STL containers can be parameterized with customer allocators, it seems that a special allocator can allocate its elements on persistent storage simply. Unfortunately, the constructors of the container class do not provide an argument to pass the name of the file the elements should be stored in. The customer allocator class only allows allocating and freeing memory, constructing and destroying a given object [Strou97, ISO98]. The constructor of STL containers cannot check the allocator for previously stored elements. Therefore, it is not possible to convert an STL container into a persistent one just by supplying a special allocator class to its constructor.

In our R-tree implementation, a proxy class is used to be responsible for writing the container's elements to and reading the element from the secondary storage.

## 3.4 Issues Encounterd in the Design and Implementation

### 3.4.1 Different Designs by Using Composite Pattern

Most tree structures have two types of nodes i.e.leaf node and non-leaf node. We need a uniform interface so that the common operations can be used on any instance without knowing its exact class. It is very natural that we use the composite design pattern to design and implement the tree structure. The main idea of the composite design pattern is to provide a uniform interface to instances from different classes in the same hierarchy, where instances are all components of the same composite complex object.

#### 3.4.1.1 Maximize the Component Interface?

There are many issues to be considered when implementing the tree structure by Composite Design Pattern. One is "maximizing the component interface by defining as many common operations for composite and leaf classes as possible" [Gamm95]. It is not possible to put all attributes and operations in the base class because leaf and non-leaf node have different attributes and operations. For example, a leaf node has entries that are pairs of key and data reference, while a non-leaf node posses entries consisting of pairs of key and child pointer.

Figure 3-3 illustrates a possible design of the tree structure. The Page class is designed to be an interface that declares common attributes and methods. The inheritance class IndexPage and LeafPage implements the methods respectively in their own scope.

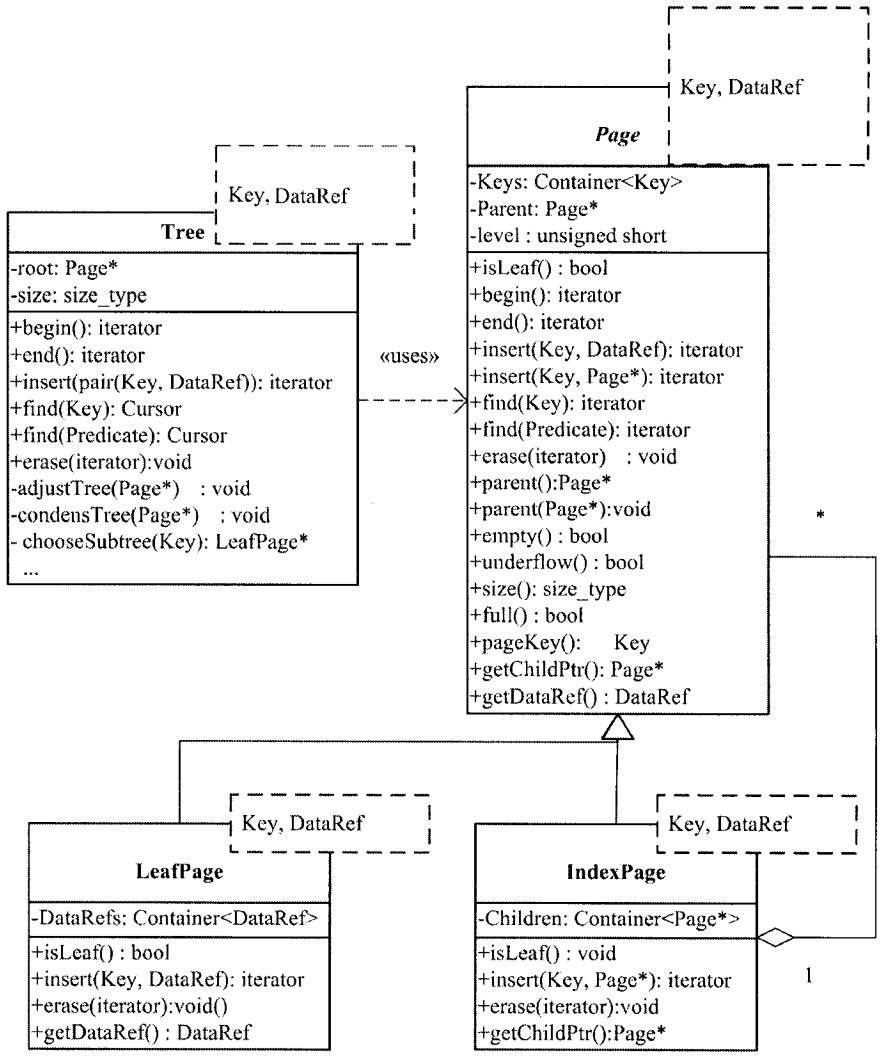



Figure 3-3 One Design to Maximize the Component Interface

The iterator is defined to be the iterator of container<Key> so that the iterator is the same for IndexPage and LeafPage class. Functions in Page class are declared as virtual functions. We implement insert, erase method in the sub-classes since they perform some operations on the container of child pointer or data reference. To access the elements of container of child pointer or reference, two more special functions:

virtual Page* getChildPointer(int n) {...} // for IndexPage

virtual dataRef getReference(int n) {...} // for LeafPage

are declared in the base class, and implemented in its subclass respectively. In this way, not only the Page interface is maximized and uniformed, but also the number of functions declared in the base class is reduced.

So far so good, the uniform interface is simple and easy to use. It seems that the user class can perform its algorithms to insert, search and erase the element in these classes by using the iterator. However, this design has serious problems. Firstly, the iterator is container<Key>::iterator. Although the user can use the iterator to perform some operations on the elements of contain<Key>, it does not have the access to the container of child pointer in index page or data reference in leaf page. Then, if we try to apply some user-defined algorithms on these classes by iterators, problems occur. Since we do not have the access to the container of child pointers or data references, the algorithm can not work absolutely if the algorithm relates to child pointers or data references. For example, given two iterators of a leaf page, say it1 and it2, I just simply want to swap their value as I usually do in STL containers. Then this leaf page is in a mess after the operation std::swap(*it1, *it2) since only the key values have been exchanged.

Moreover, this design separates all the attributes into two classes, i.e. the base class and the derived class. It obviously breaks the concept of STL container and is difficult to reuse in applications.

#### 3.4.1.2 Type Casting

To follow the STL style and make the iterator meaningful, we have to design our own container classes by putting all attributes and methods in them. In any case, we design two containers, namely IndexPage and LeafPage (Figure 3-4) that are derived classes of the interface Page.

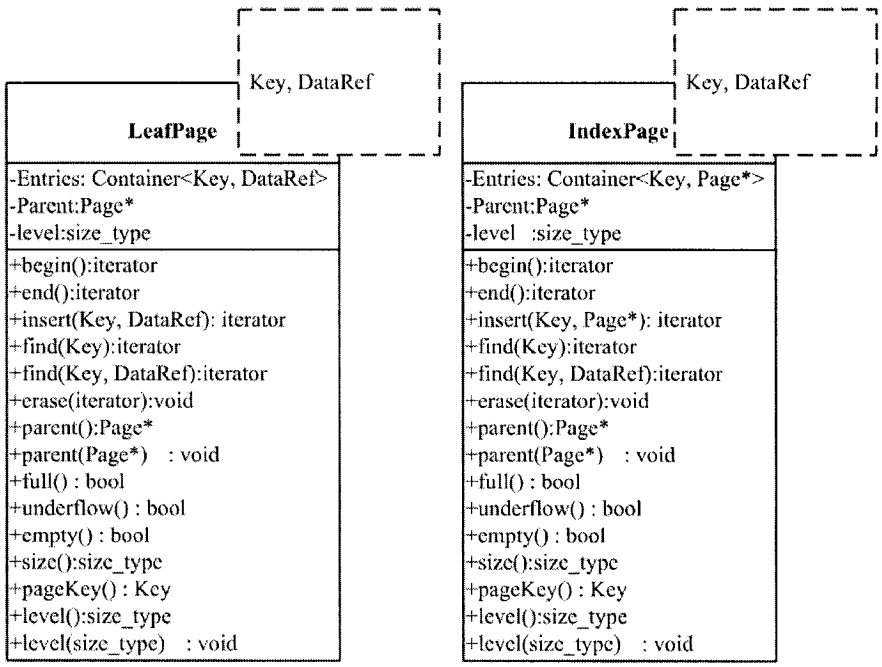



Figure 3-4 LeafPage and IndexPage Containers

Since a page pointer may point to a LeafPage or IndexPage and the function depends on the run-time page type, we must distinguish between LeafPage and IndexPage. This might be fulfilled by adding a specific method isLeaf() into the base class and its sub-classes. This method return true if the proceeding class is a leafpage, otherwise it return false. Then the page pointer can be downcast to the actual type:

```
Page ptr;
...
if(ptr->isLeaf())
{
        LeafPage* leaf = static_cast<LeafPage*> ptr;
        leafpage_iterator it= leaf->find(aKey);
}
else
{
        IndexPage* index = static_cast<IndexPage*> ptr;
        indexpage_iterator it= index->find(aKey);
}
```

Using the RTTI (Run-Time Type Identification) is another way that allows programmatically getting information about objects and classes at runtime. After

finding the actual type and check the possible type, the pointer is cast to the actual type [Mey95]:

```
const type_info& objectType = typeid(page);
if (page == typeid(LeafPage)
{
        LeafPage& leaf = static_cast<LeafPage>(page);
        leafpage_iterator it = leaf->find(aKey);
}
else if (page == typeid(IndexPage)
{
        IndexPage& index = static_cast<IndexPage>(page);
        indexpage_iterator it = index->find(aKey);
}
else
{
        throw exception of unknown type;
}
```

Down casting is dangerous and the source code is very difficult to maintain. In his book of "effective C++", Scott Meyers explained why we have to "avoid cast down the inheritance hierarchy" [Scot92].

#### 3.4.1.3 Tag Dispatching

To avoid using type cast, we have to find another way. The two containers in Figure 3-4 have alike interface except some different methods, but they are absolutely different. For example, with the same conceptual name, their iterators are distinct; iterator of IndexPage is container<Key, Page*>::iterator while that of LeafPage is container<Key, DataRef>::iterator. In the class Page, we distinguish them as indexpage_iterator and leafPage_iterator respectively. Therefore, each function in these classed must be declared in class Page showed in Figure 3-5.

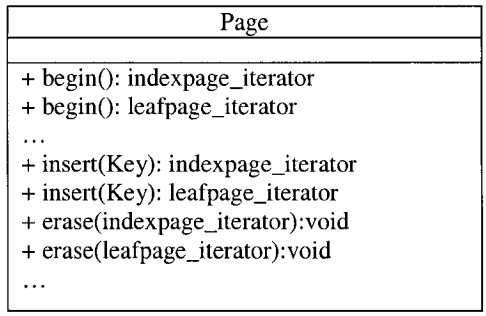



Figure 3-5 Ambiguity of Function Declaration

In C++, function overloading cannot be applied for those take same parameter but return different type. Class Page will cause function overloading ambiguity. Since virtual functions cannot be parameterized [ANSI97], in order to solve this problem, we have no choice but simply change the functions' name. Take begin() as an example, we can write:

```
leafpage_iterator      leafpage_begin()
indexpage_iterator     indexpage_begin()
```

Therefore, in the base class Page, each function has to be declared for leaf page and index page respectively.

Another way is using tag dispatching technology. It is a way of using function overload to dispatch based on properties of a type. A good example is the implementation of the std::advance() function in the C++ Standard Library [Abra01]. We define two tag classes:

```
Struct indexpage_tag{ };
Struct leafpage_tag{ };
```

Then, the function begin() can be re-written as the following:

```
leafpage_iterator      begin(indexpage_tag)
indexpage_iterator     begin(leafpage_tag)
```

The remaining functions can also be defined similarly. A tag is simply an empty class whose only purpose is to convey some property for use in tag dispatching and similar techniques. It costs very tiny and this approach can be used to implement the R tree index. However, it still violates the STL concept, and it is difficult to understand for generic programmers.

### 3.4.2 Say No to Composite Pattern?

Suppose we have a type wrapper class, called magicValue, which could hold arbitrary types. Then the index structure could be simplified. We only need one page container rather than two sub-classes derived from the interface class Page. Then for those IndexPage with pointers pointing to leaf pages, the magicValue is LeafPage*; the remaining IndexPage's mageicValue would be assigned with IndexPage*; and the LeafPage would store data reference in magicValue. Therefore, the tree structure could be very simple as shown in Figure 3-6.

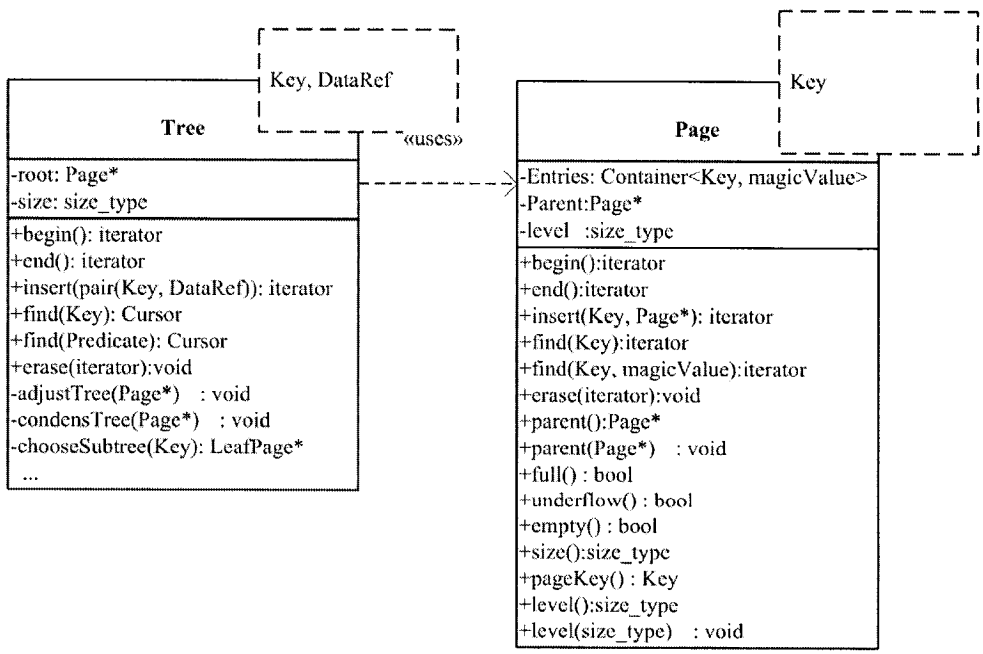



Figure 3-6 Index Structure Using a Type Wrapper

In [Sim99], a new technique called Chameleon Objects for providing a generic, type-safe wrapper class is presented. According to the author, this class can hold arbitrary data types and can be used to pass these objects between different program units while maintaining type safety. An arbitrary variable v of type T can be assigned to an instance of *value*, and therefore the value object itself can be assigned to any instance of type T, just as it were v itself. If the caller tries to assign a *value* object to a variable of a type other than T, it will throw an Incompatible_Type_Exception.

Although the magic functionality was bought with an increasing amount of memory, runtime, and space overhead, *Chameleon Objects* provide us a kind of "dynamic runtime polymorphism" not for function calls, as provided by inheritance and the virtual function mechanism, but for the data type of an object [Sim99]. Unfortunately, it can only work on Edison Design Group for their compiler *front end* at present

### 3.4.3 Container-Independent Code

At first glance, algorithms appear to work independently of container type. Some of us may strive to write container-independent code. For example, in the implementation of the R-tree index, we might think about generalizing the notion of a container so that we can use the container class with the default vector container, but still preserve the option of replacing it with some other containers later – all without changing the code that use it.

However, in practice, the possibilities for writing container independent code are more restricted than one might expected. First, the code would have to be restricted to the common interface presented by all the containers that you wish to support. In addition, the semantics of one container might still invalidate the code. In his book,

*Effective STL*, Scott Meyers lists several reasons and some examples to explain that we have to choose the container carefully instead of container-independent code. He said "this kind of generalization, well-intentioned though it is, is almost always misguided".

### 3.4.4 STL Vector Container or C-Style Array

In STL, "Arrays are generalized into containers and parameterized on the types of objects they contain." [Scot01]. Almost every STL book strongly encouraged C++ programmers to use standard vector template--a dynamically expandable array, instead of C-style arrays. Using a vector is easier and safer than using an array, because a vector is a better abstracted and better-encapsulated form of container.

What about using the C style array? It is no doubt that the user must make sure that someone will later delete the allocation. Without a subsequent delete, the operation "*new*" will yield a resource leak. Second, the "delete" must be used once and exactly only once, and a correct form of delete (delete or delete []) must be used, otherwise the results will be undefined--some programs will crash at runtime, and others will silently blunder forward [Scot01].

As a replacement for ordinary arrays, the STL provides class std::vector that provides the semantics of dynamic arrays. Thus, it manages data to be able to change the number of elements. However, this results in some overhead in case only arrays with static size are needed. For example, the keys in the R-tree index structure are mainly rectangle and point. Since the vector itself consists of three data members, e.g. iterator start, iterator end, and iterator end_of_storage, the memory overhead are significant for the implementaion of 2-d point and rectangle. As a result, the page capacity is

decreased (Figure 3-7) which will considerably affect the performance of the tree structure.

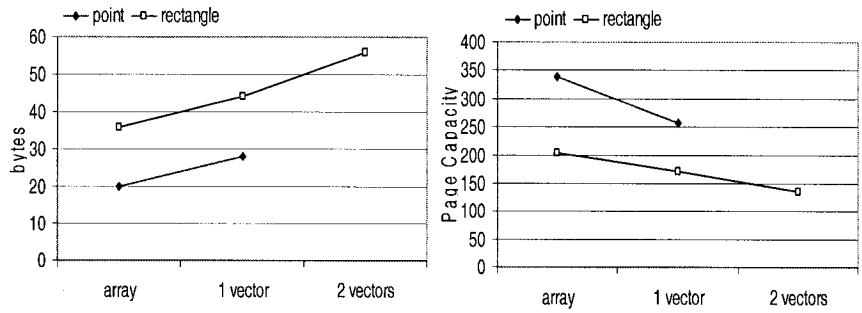



Figure 3-7 the Size of 2-d Spatial Objects and LeafPage Capacities

## 3.5 The Solution

### 3.5.1 Basic Components

The R-tree will be built using STL components which will provide the necessary index, and manage the access of the index. The tree is a container that stores objects of type page. It provides some necessary operations for insertion, search, and update. For any STL containers used in the R-tree design, the default allocators are used for the memory allocation and deallocation of their elements. Instead using a special custom allocator for object persistence, a proxy is used to control the access to the index pages and leaf pages. Figure 3-8 is the components layout of the R-tree, which is improved from the original design of [Gaff01].

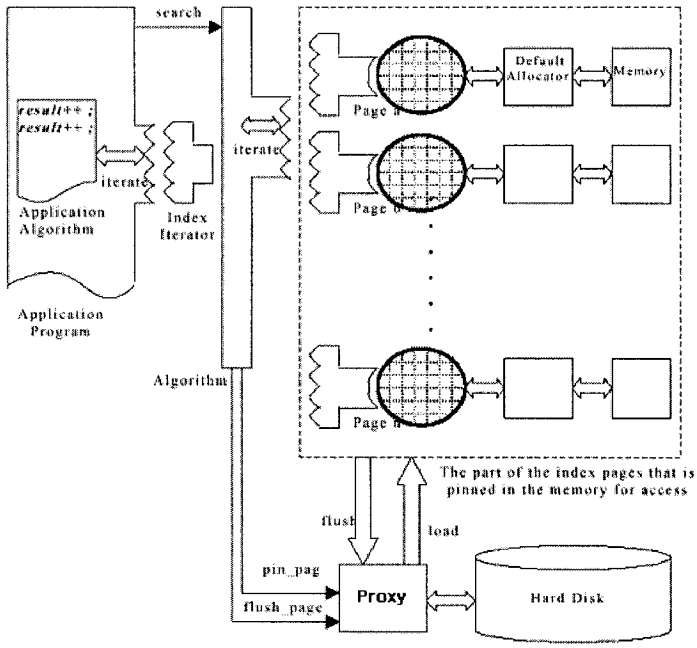



Figure 3-8 System Layout

### 3.5.2 Class Diagram

Following the STL style, we design the R-tree by using the design patter of composite and casting method. The relationship of the main classes is illustrated in the class diagram (Figure 3-9).

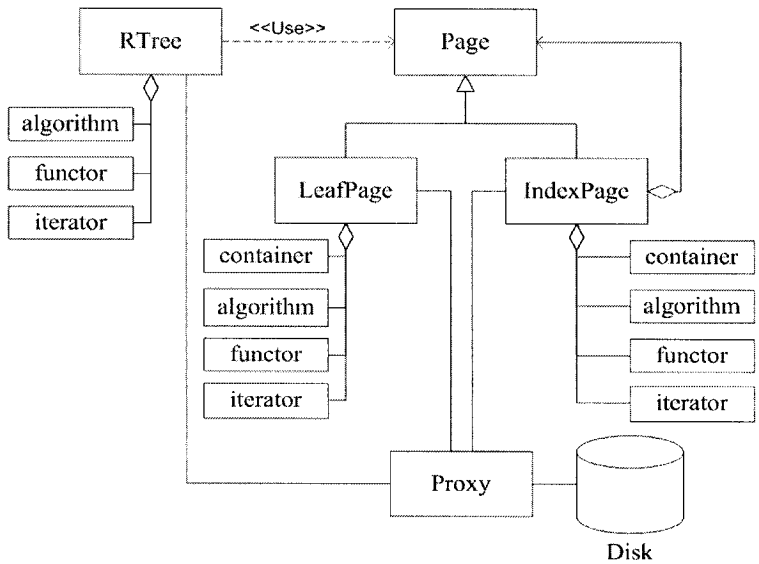



Figure 3-9 Class Diagram

### 3.5.3 The Design

#### 3.5.3.1 Keys: Point and Rectangle

Keys are used for indexing multidimensional dataset. In the R-tree, only two kinds of keys i.e. point and rectangle are used to present the feature vector of the spatial data. In Figure 3-9, the keys are designed to be custom data types that must be given to the RTree container. The RTree container can also receive other user defined types of spatial objects.

A point respresents a location in the n-dimensional space. All dimensions are stored in an array or vector. A rectangle specifies an area in the n-dimensional space that is enclosed by its lower-left point and upper-right point. For example [Kata97], from a 2D color image, we can use a criterion to extract a color histogram to be used as an effcient key that is much smaller than the image. Then the four histograms are concatenated to give a 36-dimensional feature vector. Using points and rectangles in the 2d dimensional space is the simplest example. In this thesis, we will focus on the 2-d dimensional objects.

**Class Point**

Class Point is designed like a standard container that has a standard interface. Since the STL vector has some space overhead, we have implemented the point class in two ways; one uses C-style array, and the other one uses STL vector.

**Class Rectangle**

The STL vector provides the semantics of dynamic arrays, which results in space overhead. To represent the two points, we can use two vector<T>, or one

vector<pair<T, T> >. Considering the fixed dimension, it is better to use array to represent the multidimensional coordinates. We provide the array-base Rectangle class with our final R-tree implementation while the vector-based Rectangle implementations are supplied for comparisons.

STL containers do not store the objects that you pass to them. They make copies (using copy constructor) and store those instead. Not only do the containers copy objects when adding and retrieving them, but also when the containers need to reorganize (erase or sort) themselves. In order to avoiding the copy of object, swap functions are implemented for both class point and rectangle by simply exchanging the pointer to the array or invoking the built-in swap function of the std::vector container.

**Topological Relationships**

Topological relations describe purely qualitative properties that characterize the relative positions of spatial objects. Examples of topological relationships are equal, meet, cover, coveredBy, disjoint, overlap, within, etc. These relationships can be simplified to be equal, contain, overlap, and within. Actually most spatial index structures should supported these four kinds of basic queries. Thus, several methods to judge these topological relationships are included in the class Rectangle.

### 3.5.3.2 Page Containers

The class Page is only an abstract class that declares the interface for accessing and managing the derived classes in the composite design pattern. No default behavior is implemented in this class since it is only an interface. Figure 3-10 illustrates the interface of class Page.

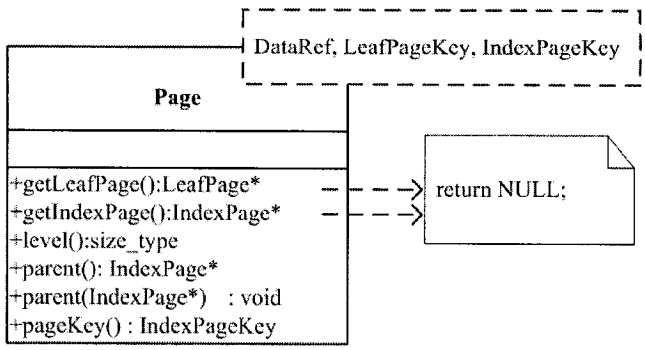



Figure 3-10 Interface of Class Page

Both class IndexPage and LeafPage, derived from class Page, are designed to be STL style containers and must conform to all STL interface characteristics. According to our application, we decided using a contiguous-memory container (also known as array-based container) which stores their elements in one or more chunks of memory. This kind of container might be the STL vector or C style array. Following the STL style programming, we choose the standard vector which is implicitly included in the class of index and leaf page. Figure 3-11 and Figure 3-12 shows their structures and interfaces respectively.

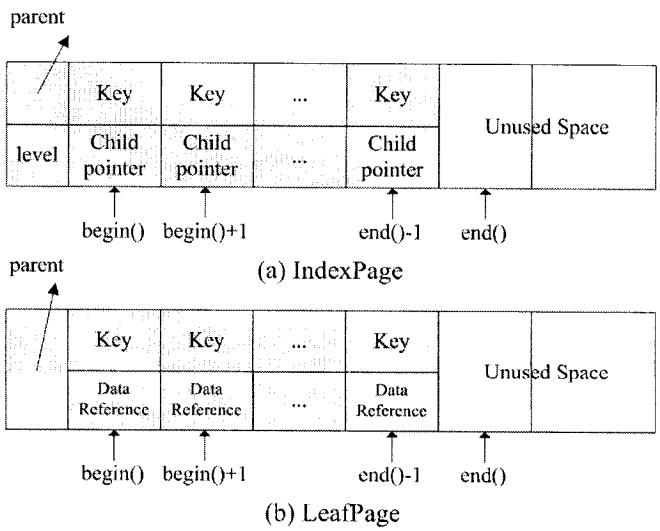



Figure 3-11 Structure of IndexPage and LeafPage

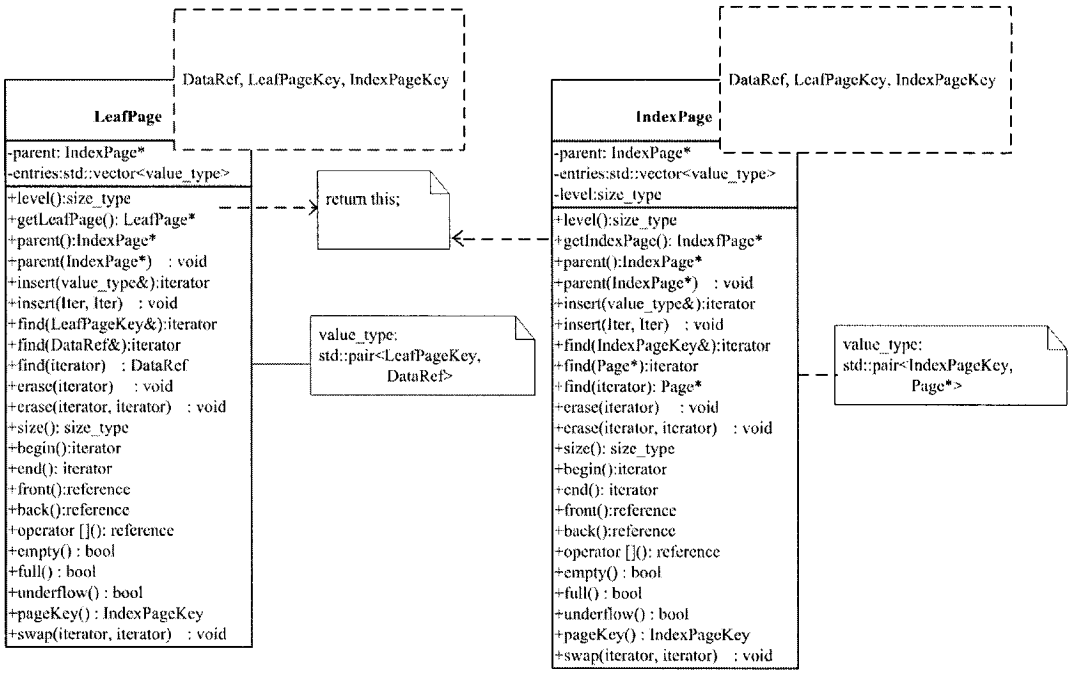
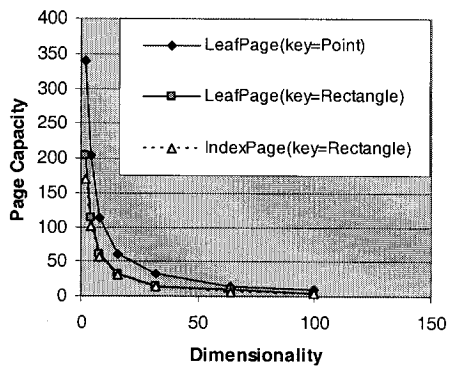



Figure 3-12 Interface of IndexPage and LeafPage

The value_type is pairs(key, child pointer) for class IndexPage or pair(key, data reference) for class LeafPage. The iterators of both classes are redefined from their implicit vectors. Therefore, not only they provide the class-specified functions such as begin(), end(), insert() and find(), but also they show us uniform interfaces.

The casting method design pattern is used to get the real page type at real-time. To implement this pattern, two essential functions, i.e. getIndexPage() and getLeafPage(), are implemented in class IndexPage and LeafPage respectively. The former return a reference to the current index page; the latter return a reference to the current leaf page. Therefore, when getting the actual page type by using the method level() (leafpage's level is zero), we can obtain the reference of a IndexPage or LeafPage object by calling one of these two special functions. Once we get a reference from a derived class, we can use the object as usual.

The IndexPage always received the rectangle (MBR) as its key. The LeafPage stores both points and rectangles. Currently, most R-tree implementation stored point in the form of rectangle by degrading a rectangle to a point. However, this leads to space overhead, and increases the execution time. Considering the different size of objects of Point and Rectangle, we designed the LeafPage to be able to hold either Point or Rectangle object so that if the dataset is a collection of points, we can store points in the LeafPage. Therefore the LeafPage can store about two times more point objects than rectangle objects (Figure 3-13). In the extrem, the leaf page is able to take only 2 points with 500 dimensions when the page size is per default 8kb. In the other word, the maximium dimensionality is internally limited to about 500.




| | |
|---|---|
| primary type: double | 8 bytes |
| data reference: int | 4 bytes |
| smartPtr<Page> | 12 bytes |
| smartPtr<IndexPage> | 12 bytes |
| Page Size | 8 kb |

Container of LeafPage and IndexPage:
std::vector<Pair>

Figure 3-13 Effect of Different Shape on Page Capacity

Since the size of key type has considerable effects on the number of entries a page can holds, we also designed some variant points and rectangles for evaluation by using STL std::vector or C-style array.

#### 3.5.3.3 RTree Container

Based on the implementation of IndexPage and LeafPage, the R-tree structure is created. The R-tree is initialized with an empty LeafPage container (root page) that is pointed by root, the only data member in RTree container. With the insertion of new

data entries, the tree will dynamically grow. With the deletion of data entries, the tree will shrink automatically. The structure and interface of the RTree container is illustrated in Figure 3-14.

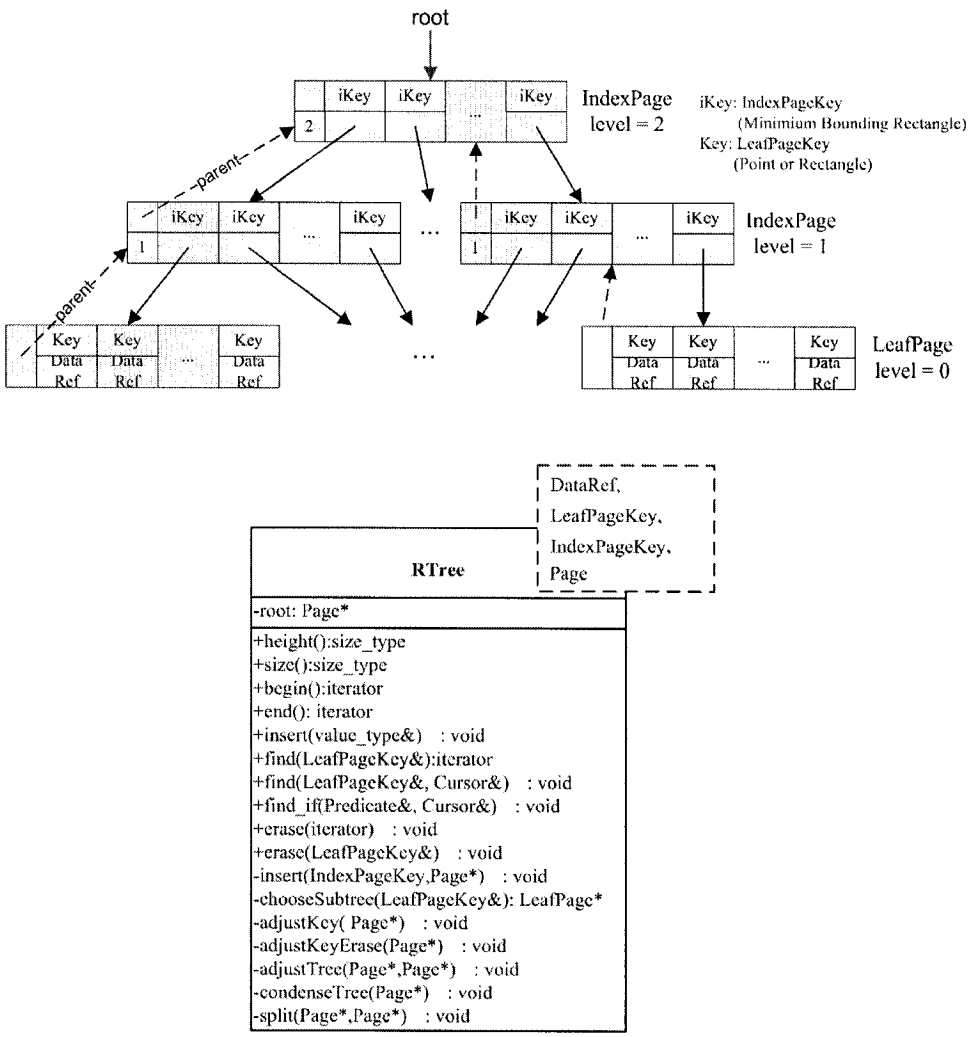



Figure 3-14 R-tree Structure and Interface

### 3.5.3.4 Class iterator

The RTree container provides an internal bidirectional iterator that allows access to elements in leaf pages within the RTree container. Unlike the B+ tree, entries in R-tree are not ordered. The iterator only provide an traversal method for data records in

the tree. In order to determine the position a query method finds, we need the page pointer to the leaf page, and the LeafPage iterator to the interested entry. Thus, iterator keeps these two parameter as data members and increment and decrement operation on the iterator. Figure 3-15 illustrates the iterator structure.

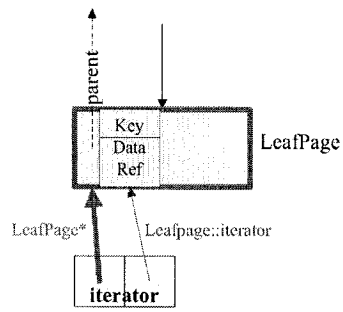



Figure 3-15 Iterator Structure and Interface

When implementing a persistent R-tree, Class iterator carries a smart pointer of LeafPage rather than a bare pointer to the LeafPage. This guarantees that the iterator would not be left with a dangling pointer since the page might be fall out of cache and be deleted from the memory. However, LeafPage::iterator will not stay valid if the page is flushed from the cache. Therefore, instead of using LeafPage::iterator, we should use the slot number.

#### 3.5.3.5 Class Cursor

The cursor provides an interface for iterating over the results of querying a database. It is created by the application when search for a data. The cursor is bi-directional and read-only. It provides a way to remember the current position and can generate the next one.

Class Cursor uses a std::vector container to store RTree::iterator. An appplication should first create a cursor and a predicate passing to the generic search function

CRTree:find_if. This function will use the predicate as a filter to iterate from the root downward to the leaf. Qualified positions, for which the given predicate returns true, in leafpages will be passed as results to the cursor object.

#### 3.5.3.6 Basic Predicates and Predicate Binder

According to the topological relationship of spatial objects, we provide four basic binary predicates i.e. equal, overlap, contain, and within. These four predicates are all callable entities [Alex01], which are objects supporting operator(). They simply forward the action to appropriate functions of the keys. For each of the predicates, overload of function-call operator (operator()) makes the predicate be able to adapt to different cases, no matter what the first parameter is an leafpage_value_type or indexpage_value_type, and the second parameter is a point or rectangle object. Figure 3-16 shows the interface of binary predicate *equal*.

```
template<typename IndexEntry, typename LeafEntry>
struct equal : public std::binary_function<IndexEntry, LeafEntry, bool>
{
        typedef typename LeafEntry::first_type  LeafKey;
        typedef typename IndexEntry::first_type IndexKey;
        template<typename EntryType, typename KeyType>
        bool operator()(const EntryType&, const KeyType&)  const;
        bool operator()(const LeafEntry&,  const IndexKey&)  const;
};
```

Figure 3-16 Binary Predicate Interface

As soon as the basic predicates are ready, we can convert types of these predicates to another. Such conversion is well known as binding [Alex01] that is a powerful feature. The STL provided two binders, i.e. std::bind1st and std::bind2nd. The binder can store not only callable entities, but also part (or all) of their arguments as well. This can greatly increase the expensive power of predicate because it allows packaging of predicates and arguments without requiring glue code.

We design the binder as a template class—class Predicate (Figure 3-17). Although it has a name of *Predicate*, it is actually a predicate binder. It has two predicate holders that take two basic predicates for index page and leaf page respectively. It can adapt to both leaf page and index page due to the overload of call operator. When applied, it receives two binary predicates and a key as parameters, and returns an object of unary predicate.

```
template<typename BinFun1, typename BinFun2, typename KeyType>
class Predicate{
private:
        BinaryFun1 mPredHandle1;                    //** for indexpage
        BinaryFun2 mPredHandle2;                    //** for leafpage
        KeyType mKey;
public:
        Predicate(const BinFun1&, const BinFun2&, const KeyType&);
        Predicate(const Predicate&);
        ~Predicate();
        bool operator=(const Predicate&);
        bool operator()(const IndexPage::value_type &) const;
        bool operator()(const LeafPage::value_type  &) const;
};
```

Figure 3-17 Interface of Predicate Binder

### 3.5.3.7 Proxy Mechanism

Current database is huge and the size of the index is possibly much bigger than the memory. It is not possible and not needed to put all the index structure in the memory although the index page and leaf page is mean to be small enough to fit into memory. The R-tree index applies a Proxy mechanism to manage controlled access to the storage of the index structure. Figure 3-18 shows the proxy mechanism.

The root page is always kept in the memory until the R-tree is destroyed. When an access to a non-root page is requested, the smart pointer will check if the demanded page is already in the memory. If positive, it returns a smart pointer that holds the bare pointer to the page. Otherwise, it checks if there is a reference to the page at once. The

smart pointer will obtain the page object if such a reference resides in the cache. If not, the cache will ask the PhysicalProxy for such a page, for which a new one is created by the Factory and then serialized from the (secondary) storage. At the moment, the smart pointer is able to meet the demand of the page.

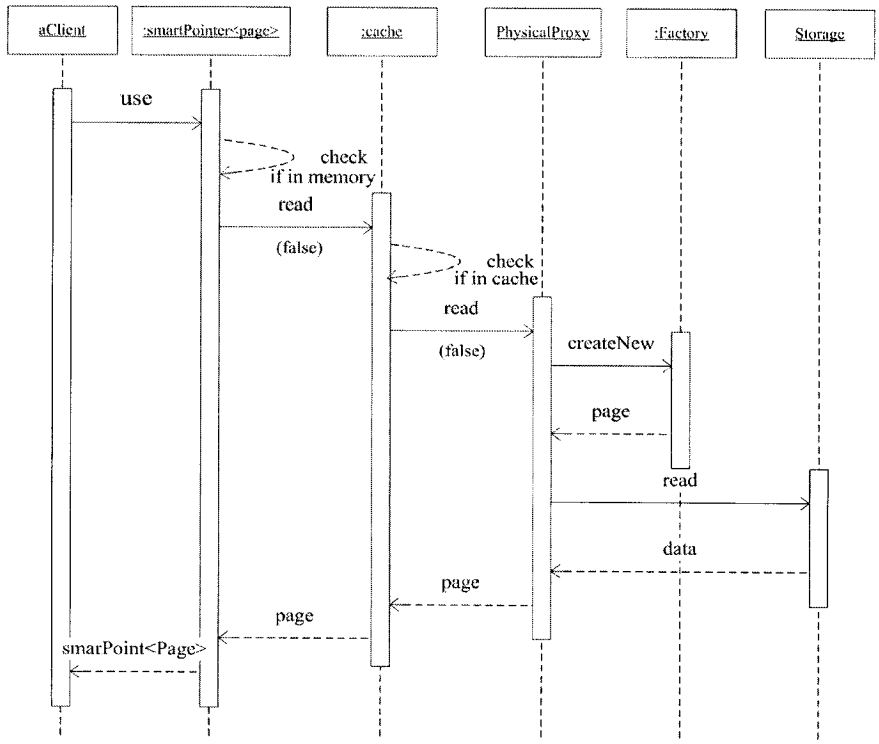



Figure 3-18 Sequence Diagram of Proxy Mechanism

## Smart Pointer

In the R-tree index structure, a smart pointer is used for many purposes. It counts the number of references to the real page so that the page can be freed automatically (garbage collection) when there are no more references. When the tree algorithm demands a non-root page, the smart pointer loads the persistent page from the physical storage in case the requested page is in neither the memory nor the cache. Figure 3-19 illustrates three reference-counted smart pointers pointing to the same page. Figure 3-20 is the interface of class SmartPtr.

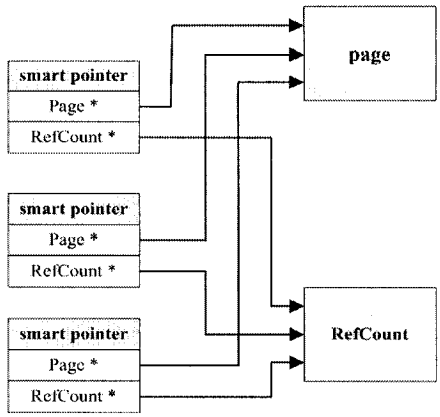



Figure 3-19 Reference-Counted Smart Pointers

The reference counting is used with the smart pointer. It tracks the number of smart pointers that point to the same page. When that number goes to zero, the page is deleted. The smart pointer hides the underlying gap between the transient and persistent objects. By using smart pointer, the persistent objects can be accessed like the objects in the memory without awareness of the smart pointers.

```
template <class T, class Cache>
class SmartPtr {
private:
        T* ptr;
        RefCounts* refCounts;
        long id;
public:
    ...
        T& operator*();
        T* operator->();
        void SetID(long);
        bool is_null();
        long GetCounter();
        void dirtied();
    ...
};

struct RefCounts {
        RefCounts();
        bool dirty;
        long totalRefs;
        long strongRefs;
};
```

Figure 3-20 Simple Interface of Class Smart Pointer

However, misusing the smart pointer will cause serious problems. For example, an indexpage P has several child pointers that are smart pointers. Its child pages also keep parent pointer to P. If we apply the smart pointer to the parent pointer, circularity arises -- a not-releasable loop would be created due to back-reference.

To solve circularity of reference counting, a weak pointer is imported. It is very similar to the smart pointer except that it does not contribution to the reference counting. Once a page is destroyed, all weak pointers pointing to the page will be set to NULL automatically.

The design of smart pointer is very flexible and easy to use. For our R-tree implementation, only few codes need to be adjusted to make the index structure persistent. For example, we use smart pointer for child pointers, and use weak pointer for parent pointers.

**Cache**

The index structure is used to retrieve data records effciently. Of course we need to find a way to make efficient use of the index structure that is too large to be held entirely in the memory. Using main-memory buffers (also called *cache*) is a great approach that can possibly obtain better performance for the tree structure. We always keep the root page in the memory until the tree is destroyed since every search through the tree requires access to the root page. We also crate a page buffer to hold some number of pages. As we read pages in from the disk in response to the user requests, the buffer is filled up. Then, when a page is requested, we access it from the cache if we can, thereby avoiding a disk access. If the page is not in cache (usually

called *page fault*), then we read it into the cache from the physical storage, replacing one of the pages that were previously there.

The global cache management includes two components that are allocation and replacement. The allocation distributes global buffer space among concurrent transaction and the replacement take responsible of the buffer access and page replacement operation. A singleton is used to manage the cache instance so that only one instance can appear.

Mostly, the performance of the R-tree structure depends on the efficiency of the caching strategy. The FIFO (First-In-First-Out) is used as the replacement strategy that will be implemented simply using a STL vector. Figure 3-21 is the cache interface.

```
template <class K, class T, class StorageType>
class Cache {
public:
    typedef PhyProxy<T,StorageType> PhyProxyType;
    typedef SingletonHolder<Cache<K,T,StorageType> > CacheType;
    typedef SmartPtr<T,CacheType> PointerType;
    typedef std::map<K,PointerType>::iterator iterator;
private:
    long MaxElement;
    PointerType Root;
    std::vector<K>  keyindex;
    std::map<K,PointerType> container;
    PhyProxyType* proxy;
public:
    ...
    size_t size() const;
    bool empty() const;
    void SetProxy(PhyProxyType*);
    PointerType GetPointer(K);
    void insert(K, PointerType)
    PointerType CreateNew(defaultIDKeyType)
    void DeleteObject(long);
    long GetRootID();
    bool isRoot(PointerType);
    PointerType GetRoot();
    void SetRoot(PointerType);
    bool HasRoot();
    void erase(iterator);
    StorageType* GetStorage();
};
```

Figure 3-21 Interface of Class Cache

### Physical Proxy and Storage

An applied index structure for the database management system must be persistent so that the index can be recreated later. A physical proxy is used for page memory allocation and deallocation— serialize or deserialize the page objects from or to the storage. When it is demand to access a page that is not in both the memory and the cache, the physical proxy creates a page instance, reads the data from the phsical storage and intials the page's data members with the data

A secondary storage such as hard disk takes a relatively long time to seek a specific location, but once the read head is positioned and ready, reading and writing a stream of contiguous bytes proceeds very rapidly. To avoide too much access to the secondary storage, a better approach is accessing a large block of contiguous location on disk at a time. The size of a block depends on many factors such as the characterixtics of the disk driver, and the amount of memory available. Generally the page size is set to be the size of a block.

Class Storage mainly takes responsible of the management and access to the index file on disk. When a new page object is required to be written, the Storage allocate a block for it on the disk. If a page object is deleted from the index structure and the deletion is demand on the file, the Storage collects the block used by the page and reallocate it. Figure 3-22 is an illustration of index structure on disk.

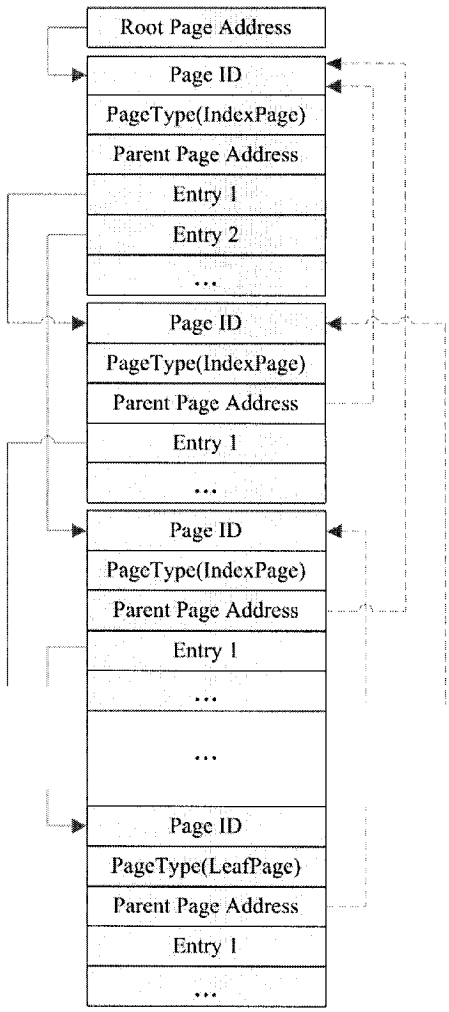



Figure 3-22 Index Structure on Disk

### 3.5.3.8 Serialization

The serializer pattern is used to make the LeafPage and IndexPage persistent (Figure 3-23). Class Page is inherited from class Serializable, in which virutal method serialize and deserialize are declared. These two functions are then simply implemented in IndexPage and LeafPage to make the concrete pages persistent. Sample code listed in Figure 3-24 shows these two operations of IndexPage.

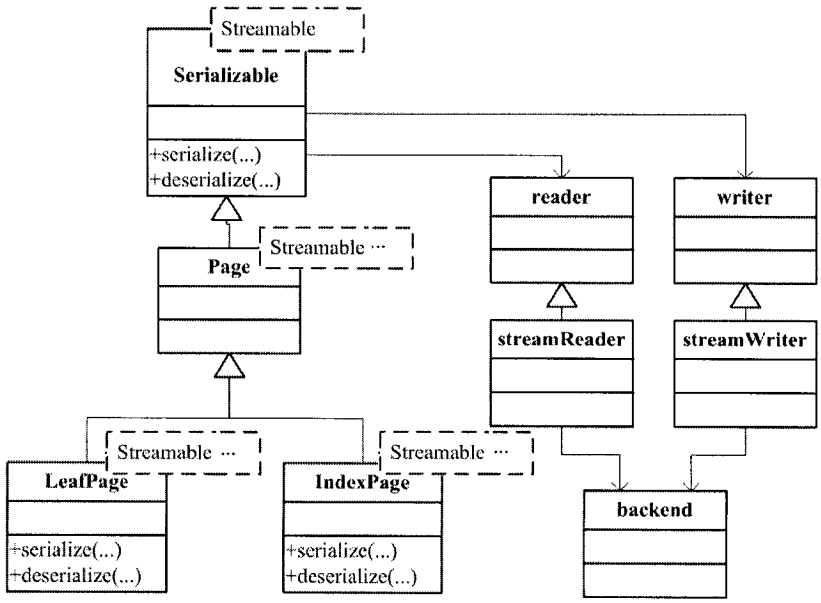



Figure 3-23 Using Serializer Pattern

```
void serialize(writer<Streamable>& out){
        serializable<Streamable>::serialize(out);
        out << mLevel;
        out << mParent;
        out << mEntries;
}

void deserialize(reader<Streamable>& in){
        serializable<Streamable>::deserialize(in);
        in >> mLevel;
        in >> mParent;
        in >> mEntries;
}
```

Figure 3-24 Serialize and Deserialize Operations of IndexPage

### 3.5.3.9 Queries

A search process will find all the entries on leaf pages that are qualified with the query demand and make the predicate return true. We design a generalized search method in the RTree container. Since keys can overlap, the query will descend multiple subtrees within the tree structure. The underlying data structure is a STL std::stack, which is used to remember which pages still have to be visited. The process

starts by initially pushing the root pointer on the stack. A page that has not yet been examined is popped off the stack and all entries in the page that qualify for the query demand are in turn pushed onto the stack. If a leaf page is popped off the stack, its qualified entries are inserted into the cursor, which holds all query results. The whole process is repeated until the stack becomes empty. Figure 3-25 describes the process.

```
template<typename Predicate>
void find_if(Predicate& pred, Cursor& aCursor)
{
    stack<page_pointer> path;
    path.push(root);
    while(!path.empty()){
        page_pointer p = path.top();
        path.pop();
        if (p is index page)
        {
            for(IndexPage::iterator it = p->begin(); it !=p->end(); ++it)
                if(pred(*it)) path.push((*it).second);
        }else
        {
            for(LeafPage::iterator it = p->begin();it != p->end(); ++it)
                if(pred(*it)) aCursor.insert(iterator(p, it));
        }
    }
}
```

Figure 3-25 Function RTree::find_if

An R-tree index should support multiple queries such as exact match query, range query, similar query, and so on. A STL-like algorithm usually receives a single predicate object and performs on a container. Instead of writing specific code as usual, we use a generic search function that takes one predicate and then returns the searching results.

Different queries use different predicates on index page and leaf page. Take exact match query as an example. The basic predicate used for index page levels is *contain* while *equal* is used for the leaf page level. Based on our binder, the class Predicate,

we are able to handle predicates of several kinds such as equal, contain, overlap, and within. The exact match query could be like:

```
void find(const Key& k, Cursor& aCursor)
{
    Predicate<contain<...>, equal<...>, Key> pred(contain<...>(), equal<...>(), k);
    find_if(pred, aCursor);
}
```

Table 3:1 lists some common queries on the RTree container by using predicate binder and invoking the function find_if.

| Queries | Predicates used on | |
|---|---|---|
| | IndexPage | LeafPage |
| Exact Match Query: Given a point or rectangle *S*, find the data point or rectangle that exactly equals S. | contain | equal |
| Point Query: Given a point P, find all data rectangles that contain P. | contain | contain |
| Rectangle enclosure query: Given a rectangle R, find all data rectangles that completely contain R. | contain | contain |
| Rectangle Intersection query: Given a rectangle S, find all data rectangles that intersect with S. | overlap | overlap |

Table 3:1 Different Queries Using Predicates

### 3.5.3.10 Activity Diagram of Insertion

After an insertion, the page may become full. In this case the R-tree index will split the page into two pages. Figure 3-26 shows the activity diagram for insertion.

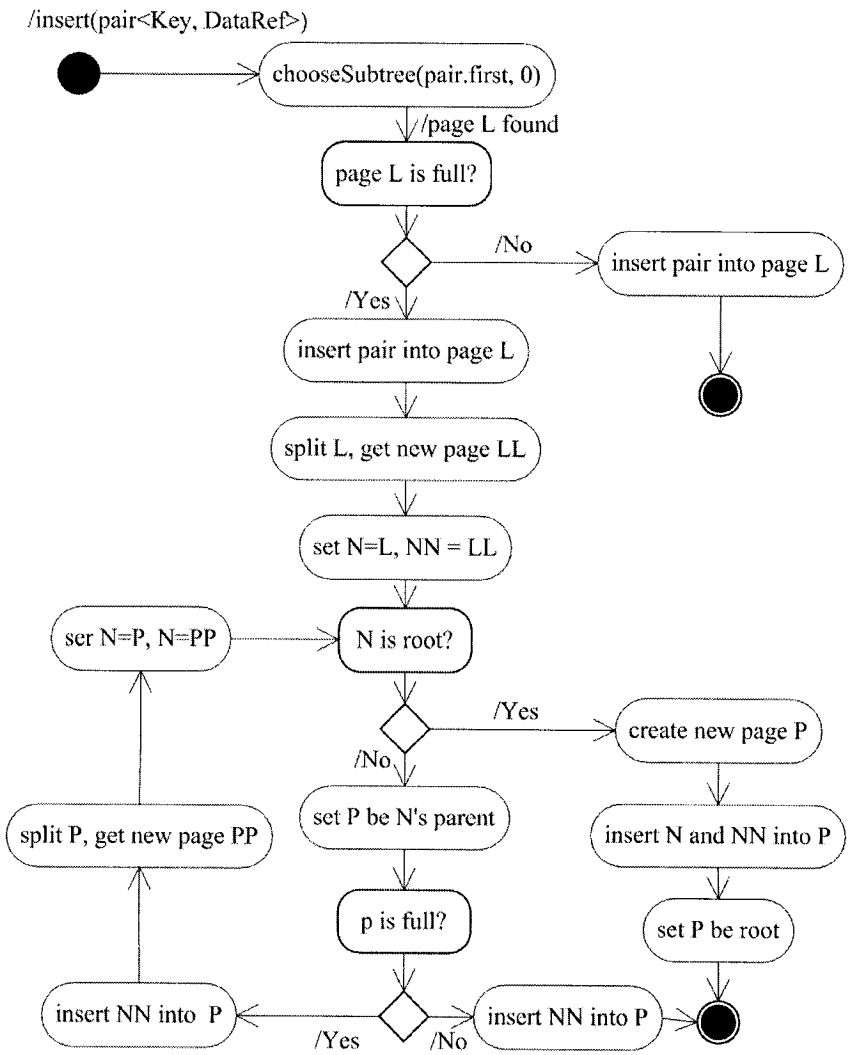



Figure 3-26 Insert Activity

### 3.5.3.11 Activity Diagram for Deletion

After a deletion, the page may become underflow. In this case, the index will take out all the entries of the underflowed page, and reinsert these entries into the index. Figure 3-27 shows the activity diagram for deletion.

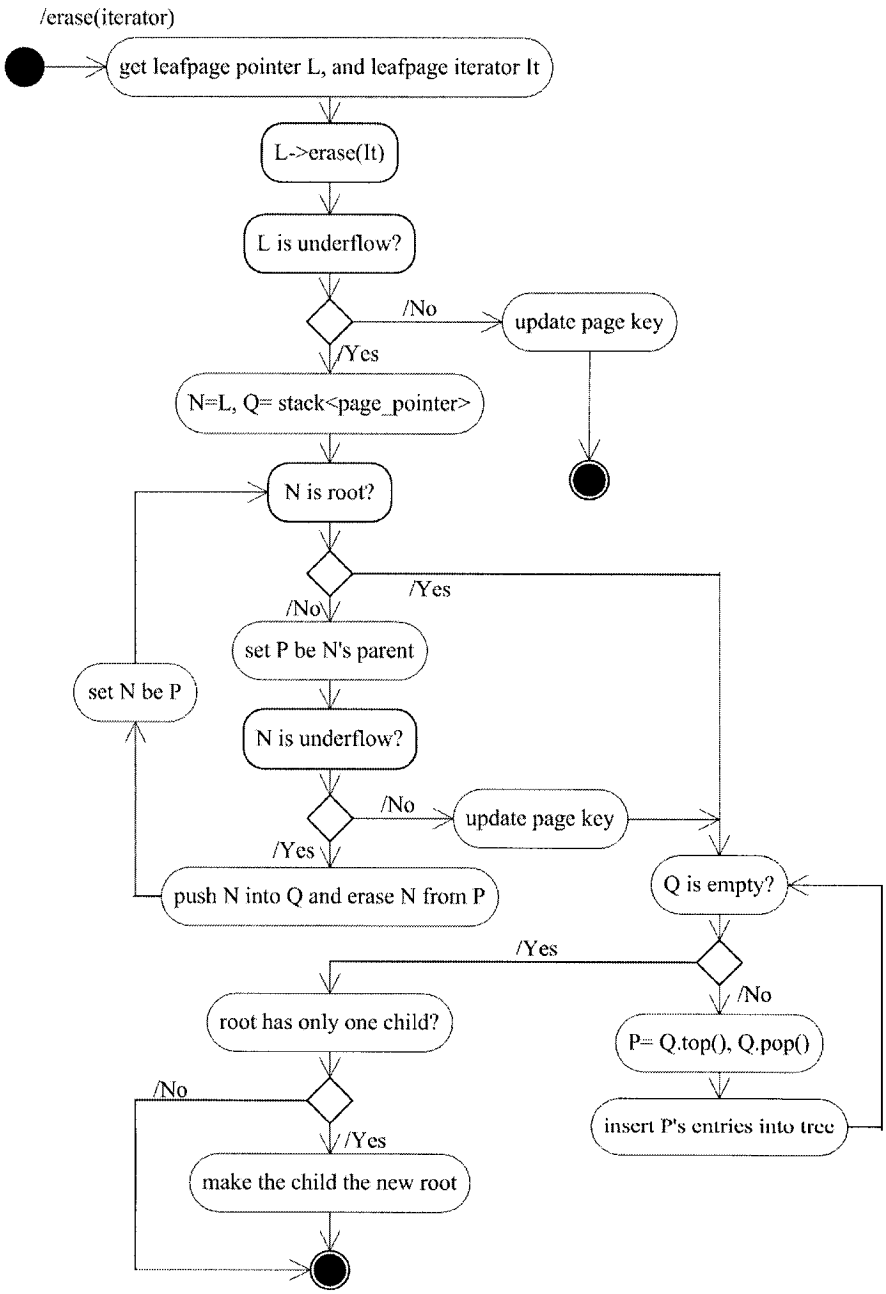



Figure 3-27 Erase Activity

### 3.5.3.12 Issue about Split Algorithm

Previous work on the R-trees [Garc98, BSK90] has shown that the split procedure is perhaps the most critical part during the dynamic R-tree construction and it significantly affects the index performance. The quadratic split algorithm proposed by Antonun Guttman [Gutt84] has a quadratic cost. The pickSeed algorithm calculates the inefficiency of grouping enries and choose the most wasteful pair as the seeds. The criteria used here is area enlargement which has to invoke Rectangle's funciton *joint*( ) to create a temperary MBR and *area*( ) to calculate the area. To improve this, we apply the Euclidean distance between two point objects as the split criteria for the leafpages whenever the key type for leaf page is Point.

The GiST uses a different algorithm which is much faster than the quadratic split algorithm. After picking seeds, this algorithm only simply assigns the remaining entries to the group with less area enlargement. We apply the algorithm (Table 3:2) to our R-tree implementaion.

| | Point Entry | Rectangle Entry |
|---|---|---|
| [Pick Seeds] | For each pair of entries $E_1$ and $E_2$, calculate their distance d. | For each pair of entries $E_1$ and $E_2$, compose a rectangle J including $E_1.I$ and $E_2.I$. Calculate d = area(J) – are($E_1.I$) – area($E_2.I$). |
| | Choose the pair with the largest d to be the two seeds. Assign one seed to Group1 and another to Group2. | |
| [Assign Entries ] | For each remaining entries E, Calculate $d_1$ = area increase required in the covering rectangle of Group1 to include E.I. Calculate $d_2$ similarly for Group2. Assign E to the group with less area enlargement (d) | |

Table 3:2 Split Algorithms from GiST

# Chapter 4 Implementation and Evaluation

## 4.1 Implementation

### 4.1.1 Implementing R-tree Index

The R-tree index is implemented according to the design discribed in chapter 3. Appendix A~I are interfaces of the main classes.

### 4.1.2 Using R-tree Index

The R-tree design and implementation is only one of the subprojects of search tree framework in the Know-It-All project. The optimizer will choose an apporpriate index structure according to the actual application. Because our R-trees is implemented to be a STL-like container so that it can be replaced with other search trees such as B+tree, SS-tree, SR-tree, and X-tree.

A search tree framework, TreeIndex, simply takes the tree as a template parameter so that the TreeIndex can invoke all the functions provided by the R-tree container through its reference (Figure 4-1).

## 4.2 Testing

Testing is the process of executing a program or system with the intent of finding errors [Myer79]. Its purposes are quality assurance, verification and validation, or reliability estimation. For the testing of the R-tree implementation, the correctness and performance testing are two major areas of testing.

### 4.2.1 Correctness Testing

Correctness is the minimum requirement of software, the essential purpose of testing.

```
tempate<typename T,  typename DataRef,
        typename LeafPageKey  = Point<T>,  typename IndexPageKey= Rectangle<T>,
        typename PageType= Page<LeafPageKey, IndexPageKey, DataRef>
        typename SearchTreeType = RTree<PageType>  >
class Index {
public:
    typedef  LeafPageKey                    key_type;
    typedef  SearchTreeType::iterator       iterator;
    typedef  SearchTreeType::Cursor         Cursor;
    ...
private:
    SearchTreeType  theTree;
public:
    ...
    iterator find(const LeafPageKey & k)  {   return theTree.find(k);  }
    void ExactMatchQuery(const LeafPageKey &k, Cursor& aCursor)  {theTree.find(k, aCursor); }

    Cursor RangeQuery (const Rectangle& s)  {              // or point
        typedef SearchTreeType::Predecate<Contain, Contain, Rectangle>  Predicate;
        Predicate  pred(Contain(), Contain(), s);
        Cursor aCursor;
        theTree.find_if(pred, aCursor);
        return aCursor;
    }
    ...
```

Figure 4-1 Interface for search index framework

Correctness testing will need some type of oracle to tell the right behavior from the wrong one [Myer79]. The tester may or may not know the inside details of the software module under test, e.g. control or data flow. Therefore, either a white-box or black-box point of view can be taken in our testing.

#### 4.2.1.1 Black-box Testing

In black-box testing, test data are derived from the specified functional requirements without regard to the final program structure [Myer79]. We treats the R-tree implementation under test as a black box -- only the inputs, outputs and specification are visible, and the functionality is determined by observing the outputs to corresponding inputs. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. No implementation details of the code are considered.

It is obvious that the more we have covered in the input space, the more problems we will find, and therefore we will be more confident about the quality of our implementation. However, exhaustively testing the combinations of valid inputs is impossible. We only take some cases to test our R-tree index. In addition, some special test cases such as illegal inputs, large inputs, and with values smaller or larger than the specified ranges.

We mainly focus on testing insertion, deletion and find operations. After the lower-level components are tested with the selected values, the higher-level components are tested on the basis of the lower-level ones.

#### 4.2.1.1 White-box Testing

A good testing plan not only contains black-box testing, but also white-box approaches. In white-box testing, software is viewed as a white-box, in which the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles [Myer79].

Since our R-tree index is implemented using the STL components that are meant to be largely independent of each other, we can do the unit tests easily. We can test a function or a code segment individually. The lower-level components such as Point, Rectangle, IndexPage container, LeafPage container, smart pointer, cache, and storage are tested independently, which guarantees they work well. For each component, test cases are planed for both simple cases and all boundary conditions. These testing will be conducted first on the low-level components and then high-level components. For our R-tree implementation, class Point and Rectangle are created and tested in the first step. When testing the LeafPage or IndexPage container, class Point and Rectangle

are passed as template parameters. For the containers, it is necessary to test all the built-in functions such as insert, erase and find. Based on the testing on these containers, we are able to do the tests on the RTree container for the function insert, erase, find, and find_if.

### 4.2.2 Performance Testing

Performance has always been a great concern about indexing structures of database. The goal can be performance bottleneck identification, performance comparison and evaluation. Performance evaluation of an indexing structure usually includes:

- Access Types – access types that are supported efficiently. Examples: exact match query or range query;
- Search Time – time to find a specified data item or set of items;
- Insertion Time – time to insert a new data item or set of items
- Deletion Time – time to delete an item or set of items;
- File Utilization ratio – the ratio of the amount of free nodes by the amount of allocated nodes.

Our performance testing mainly contains insertion, search (exact query and range query), and deletion.

#### 4.2.2.1 Experimental Environment

All experiments have been conducted on a SUN SunFire 280R server, 2 UltraSparc-III+ processors, 4 Gb main memory, and one hard disk with 36 Gb, running the Solaris 9 operating system. The compiler used is GNU g++ 3.2.

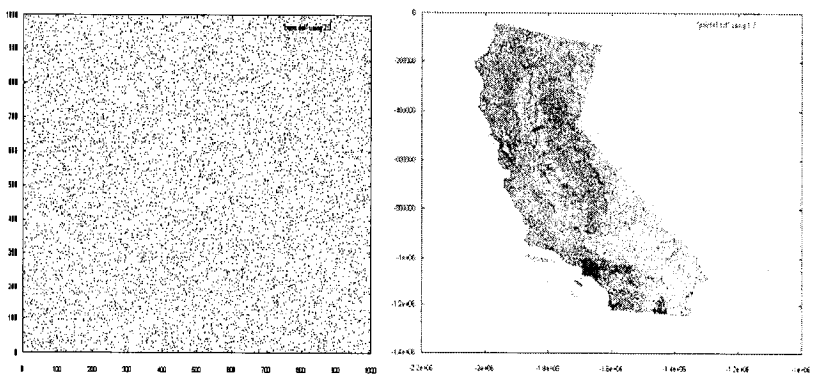
The standard timing functions of C++ do not measure I/O time. They are not suitable for benchmark in a multi-threaded environment [Rijk99]. Therefore, we use the real time clock (wall time).

The size of a page is set to 8192 bytes to meet with the disk block size of the operating system. Except where noted, all benchmark were done with the page size of 8,192 bytes and the cache size of 512 that is the number of pages the cache can holds.

#### 4.2.2.2 Experiment Datasets

We conducted experiments on both the synthetic and real point datasets (Figure 4-2):

1. Synthetic 2-d point dataset: This dataset is included in the GiST package (version 1.0). It contains 10,000 2-d points that are uniformly distributed with the range for each dimension being (0, 1,000).
2. Real 2-d point dataset: This is the 2-d point dataset of the Sequoia 2,000 benchmark. It contains locations of 62,556 California places extracted from the US Geological survey's Geographic Names Information system (GNIS). The points are geographically distributed over a 1,046km by 1,317km area.



(a) Synthetic Dataset (b) Real Dataset

Figure 4-2 Dataset Used for Experiments

#### 4.2.2.3 Testing Procedure

A test program is written to perform our tests that are arranged sequentially. Starting from an empty tree, we construct a tree by inserting all the data, perform different queries on the tree, and then delete some entries from the tree.

#### 4.2.2.4 Experimental Results

##### 4.2.2.4.1. Efficiency of Different Key Implementations

As discussed in Chapter 3, vector-based and array-based keys (Point and Rectangle) have different sizes and hence a page container can hold different number of keys. We conduct an experiment to evaluate their effects on the performance of the R-tree. This will help us in choosing an appropriate key implementation.

The synthetic 2d point dataset is used to perform our test. Each time we replace the different key implementations, compile and run the testing program. The tests include:

- Tree construction: create a tree by inserting 10,000 points
- Exact match query: for each of the randomly chosen 100 points, find the entries that exactly match.
- Deletion: for each of the randomly chosen 1,000 points, erase the entries from the tree

The test program is executed ten times, over which the results are averaged. Therefore, each final result is an average over dozens or hundred of test points. The results are summarized in Figure 4-3. Here, the vector-based class Point is used as the key of LeafPage; vector-based (1) and vector-based (2) uses one-vector-based and two-vector-based class Rectangle as the key of IndexPage respectively.

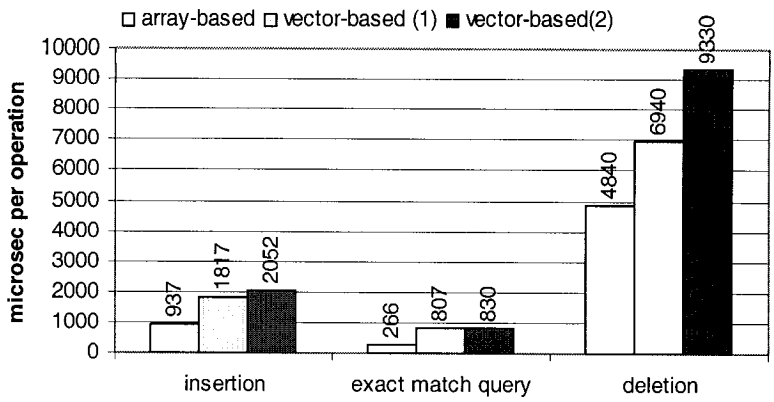



Figure 4-3 Effects of Different Key Implementations on Tree Performance

The measurement has demonstrated that using array-based implementation can greatly improve the tree performance. The cause is that array-based keys do not import the overhead from vectors so that the pages can hold many more entries than those of using vector-based keys. This possibly improves the tree fan-out. In addition, since the STL container copies the object you pass, the construction of a vector for vector-based keys costs time and greatly contributes to performance degradation. As a result, we prefer the array-based key implementation, which has been used in the final R-tree implementation.

#### 4.2.2.4.2. Test on Different Page Size

The synthetic dataset from the GiST is comparatively small. To make the R-tree of significant size (at least 3 levels), we choose different smaller page sizes and test the performances. Fixing the cache size at 512 pages, we adjust the page size to 1 kb, 2kb, 4 kb, and 8 kb, then compile and run the test program. The GiST is also tested at the same condition on the same synthetic dataset. Figure 4-4 illustrates the test result.

| | | Response time per operation (microseconds) | | | |
|---|---|---|---|---|---|
| Page Size | | 1 k | 2 k | 4 k | 8 k |
| insertion | KIA | 157 | 213 | 330 | 484 |
| | GiST | 180 | 225 | 376 | 502 |
| exact match query | KIA | 125 | 145 | 290 | 289 |
| | GiST | 498 | 296 | 386 | 486 |
| deletion | KIA | 413 | 756 | 1610 | 1870 |
| | GiST | 644 | 312 | 370 | 513 |

Table 4:1 Performance Comparisons at Different Page Size

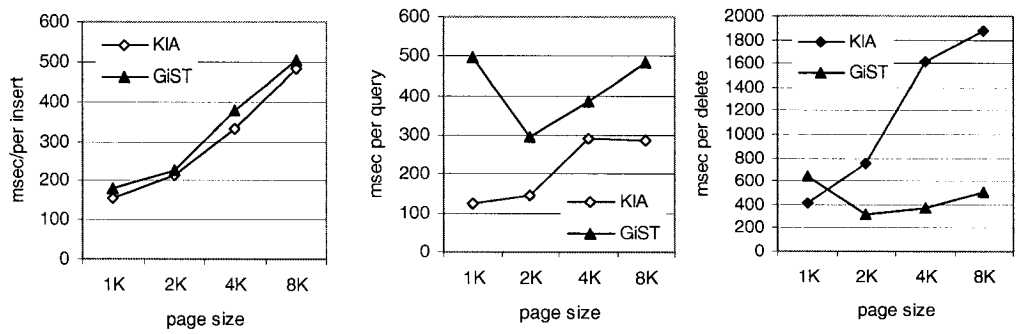



Figure 4-4 Performance Comparisons at Different Page Size

The KIA R-tree is slightly faster than the GiST R-tree for insertion. The exact match query of KIA R-tree is also faster than that of the GiST R-tree. The reason is that KIA R-tree keeps lots of the index pages and some leaf pages in the memory that accelerate the query, whereas the GiST R-tree only keeps a path in the memory. In terms of the deletion, the GiST R-tree has much better performance than ours. We will discuss the cause later in this chapter.

#### 4.2.2.4.3. Test on Real Dataset

Tests on point dataset of Sequoia 2000 benchmark are performed.

- Tree construction: create a tree by inserting 62,556 points;

- Exact match query: for each of the randomly chosen 100 points, find the entries that exactly match;
- Range query: for each of the randomly chosen 100 points with coordinate of (x, y), create a query rectangle (square) whose left bottom point is (x, y) and right upper point is (x + l, y + l) where l is the side length of the square. The area of the query rectangles varies from 0.001%, 0.01%, 0.1%, and 1% relatively to the area of the data space.
- Deletion: for each of the 6,256 points (10%), erase the entry.

The test program is executed ten times and averages over them are taken as the results which are summarized in Table 4:2 and Figure 4-5.

| microsec | insertion | exact match query | range query | | | | deletion |
|---|---|---|---|---|---|---|---|
| | | | 0.001% | 0.01% | 0.1% | 1% | |
| KIA | 654 | 657 | 891 | 1138 | 2321 | 7339 | 3620 |
| GiST | 743 | 1256 | 1474 | 1893 | 3926 | 13013 | 1275 |

Table 4:2 Test Results of Real Dataset

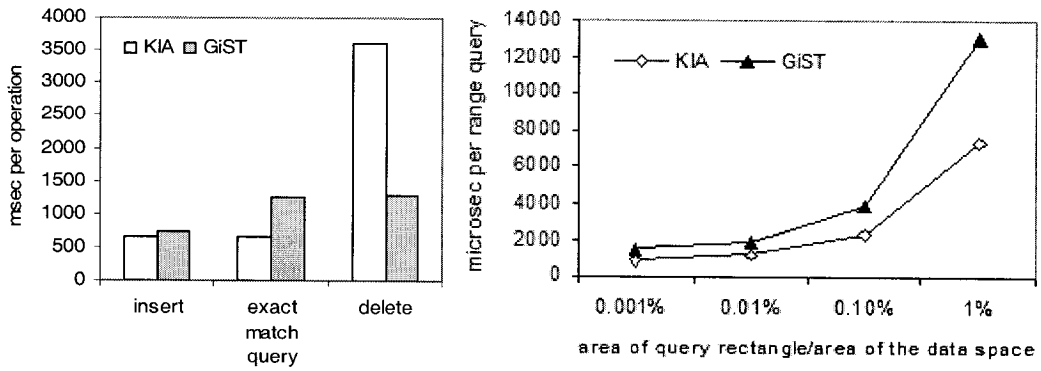



Figure 4-5 Performance Comparisons between KIA and the GiST R-tree

The KIA R-tree inserts records slightly faster than the GiST R-tree and the exact match query is performed much faster than the GiST R-tree. However, in terms of the

deletion, the GiST R-tree runs about 2.5 times quicker than ours. The cause is that in the GiST:

1. "Node deletion is not implemented. Instead, empty nodes are simply left in the tree" [Libgist]
2. "Bounding predicates are not shrunk when leaf items are deleted".[Libgist]

The GiST's code also proves this. We tested our R-tree under the condition that the condense function is not used. Then the average time of single point deletion is 557 microseconds for the KIA R-tree and 1,271 for the GiST R-tree.

# Chapter 5 Conclusions

As a proof of concept for the index sub-framework of the Know-It-All project, the R-tree index design and implementation is developed using STL generic programming and design patterns. This thesis starts with a review and analysis of the original design of the index framework proposed by [Gaff01]. After investigating many possible approaches, one approach is selected for our final design and implementation.

The R-tree in this thesis utilizes the casting method pattern to present the composite relationship between the abstract interface (class Page) and sub-classes i.e. class LeafPage and IndexPage. The class IndexPage, LeafPage, and RTree are designed to be STL style containers so that they conform to the standard interface of STL containers. To make the index structure persistent, a proxy mechanism is used to load a page from the physical storage on demand and write a page to disk as needed. This R-tree is user friendly, easy to be used, and can holds both point and rectangle datasets. Four basic predicates and a predicate binder are provided so that our R-tree supports many queries such as exact match, range, and other user-defined queries.

The experiments demonstrate that our R-tree implementation is comparable with that of the GiST. The insertion is slightly faster and the exact match query is faster than the GiST. However, the deletion needs more time than that of the GiST since the GiST does not erase the empty node, nor adjust the bounding rectangle, which both cost much time.

The measurements also show that it is better to use the C-style array in the class Point and Rectangle since the std::vector<> has some space overhead. The construction and deconstruction of vector, and the space overhead will degrade the performance of the R-tree.

## Appendix A Definition of Class Point

1.Using one vector

```
template<typename T>
class Point
{
public:
        typedef Point Point;
        typedef size_t size_type;
        typedef T value_type;
        typedef std::vector<value_type> container;
        typedef typename container::iterator iterator;
        typedef typename container::const_iterator const_iterator;
        typedef typename container::reverse_iterator reverse_iterator;
        typedef typename container::const_reverse_iterator const_reverse_iterator;
        typedef typename container::reference reference;
        typedef typename container::const_reference const_reference;
private:
        container mCoords;
public:
        Point();
        Point(const size_type)
        Point(const size_type, const T*)
        Point(const Point&);
        virtual ~Point();
        Point& operator = (const Point&);
        bool operator ==(const Point&) const;
        bool operator != (const Point&) const;
        T coord(size_type) const;
        void coord(const size_type, T);
        reference operator[](const size_type);
        const_reference operator[](const size_type) const;
        size_type size() const;
        bool empty() const;
        void clear();
        iterator begin();
        iterator end();
        reverse_iterator rbegin();
        reverse_iterator rend();
};
```

2.Using array

```
template<typename T>
class Point
```

```
{
public:
        typedef CPoint Point;
        typedef size_t size_type;
        typedef T       value_type;
        typedef T*      iterator;
        typedef const T*        const_iterator;
        typedef std::reverse_iterator<iterator>         reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
        typedef T&              reference;
        typedef const T&        const_reference;
        typedef std::size_t     size_type;
        typedef std::ptrdiff_t  difference_type;
private:
        value_type* mCoords;
public:
        Point();
        Point(const size_type)
        Point(const size_type, const T*)
        Point(const Point&);
        virtual ~Point();
        Point& operator = (const Point&);
        bool operator ==(const Point&) const;
        bool operator != (const Point&) const;
        T coord(size_type) const;
        void coord(const size_type, T);
        reference operator[](const size_type);
        const_reference operator[](const size_type) const;
        size_type size() const;
        bool empty() const;
        void clear();
        iterator begin();
        iterator end();
        reverse_iterator rbegin();
        reverse_iterator rend();
};
```

## Appendix B Definition of Class Rectangle

1. Using one vector

```
template<typename T>
class Rectangle
{
public:
        typedef Point<T> Point;
        typedef size_t size_type;
        typedef std::pair<T,T> value_type;
        typedef std::vector<value_type> container;
        typedef typename container::iterator iterator;
        typedef typename container::const_iterator const_iterator;
        typedef typename container::reverse_iterator reverse_iterator;
        typedef typename container::const_reverse_iterator const_reverse_iterator;
        typedef typename container::reference reference;
        typedef typename container::const_reference const_reference;
private:
        container  mCoords;
public:
        Rectangle();
        Rectangle(const size_type);
        Rectangle(const Point&, const Point&);
        Rectangle(const Rectangle&);
        virtual ~Rectangle(){};
        size_type size() const;
        bool empty() const;
        T low (const size_type) const;
        T high(const size_type) const;
        void low (const size_type, const T&);
        void high(const size_type, const T&) ;
        Rectangle& operator=(const Rectangle&);
        bool operator==(const Rectangle&) const;
        bool operator!= (const Rectangle&) const;
        void clear();
        bool equal   (const Rectangle&) const;
        bool contain(const Rectangle&)const;
        bool within  (const Rectangle&) const;
        bool overlap(const Rectangle&) const;
        double overlapArea(const Rectangle&) const ;
        void joint(const Rectangle&);
        Point center() const;
        double area() const;
```

```
        double margin() const;
        iterator begin();
        iterator end();
        reverse_iterator rbegin();
        reverse_iterator rend();
};

2.Using  two vector
template<typename T>
class Rectangle
{
public:
        typedef Point<T> Point;
        typedef size_t size_type;
        typedef T value_type;
        typedef std::vector<value_type> container;
private:
        container  mLow;
        container  mHigh;
public:
        Rectangle();
        Rectangle(const size_type);
        Rectangle(const Point&, const Point&);
        Rectangle(const Rectangle&);
        virtual ~Rectangle(){ };
        size_type size() const;
        bool empty() const;
        T low (const size_type) const;
        T high(const size_type) const;
        void low (const size_type, const T&);
        void high(const size_type, const T&) ;
        Rectangle& operator=(const Rectangle&);
        bool operator==(const Rectangle&) const;
        bool operator!= (const Rectangle&) const;
        void clear();
        bool equal   (const Rectangle&) const;
        bool contain(const Rectangle&)const;
        bool within  (const Rectangle&) const;
        bool overlap(const Rectangle&) const
        double overlapArea(const Rectangle&) const ;
        void joint(const Rectangle&);
        Point center() const;
        double area() const;
        double margin() const;
```

```
};

3.Using  array
template<typename T>
class Rectangle
{
public:
        typedef Point<T> Point;
        typedef size_t size_type;
        typedef T value_type;
private:
        size_type mDimension;
        T*  mLow;
        T*  mHigh;
public:
        Rectangle();
        Rectangle(const size_type);
        Rectangle(const Point&, const Point&);
        Rectangle(const Rectangle&);
        virtual ~Rectangle(){ };
        size_type size() const;
        bool empty() const;
        T low (const size_type) const;
        T high(const size_type) const;
        void low (const size_type, const T&);
        void high(const size_type, const T&) ;
        Rectangle& operator=(const Rectangle&);
        bool operator==(const Rectangle&) const;
        bool operator!= (const Rectangle&) const;
        void clear();
        bool equal   (const Rectangle&) const;
        bool contain(const Rectangle&)const;
        bool within  (const Rectangle&) const;
        bool overlap(const Rectangle&) const;
        double overlapArea(const Rectangle&) const ;
        void joint(const Rectangle&);
        Point center() const;
        double area() const;
        double margin() const;
};
```

## Appendix C Definition of Class Page

```
template<typename DataRef,
         typename LeafPageKey,
         typename IndexPageKey>
class CPage
{
public:
    typedef size_t size_type;
    typedef CPage<LeafPageKey, IndexPageKey, DataRef>       page_type;
    typedef CIndexPage<LeafPageKey, IndexPageKey, DataRef> indexpage_type;
    typedef CLeafPage<LeafPageKey, IndexPageKey, DataRef>  leafpage_type;
    typedef page_type*          page_pointer;
    typedef indexpage_type*     indexpage_pointer;
    typedef leafpage_type *     leafpage_pointer;
public:
    CPage() {};
    virtual ~CPage(){};
    virtual  size_type level() const;
    virtual indexpage_pointer getIndexPage();
    virtual leafpage_pointer   getLeafPage();
    virtual  indexpage_pointer parent() const;
    virtual  void parent(indexpage_pointer);
    virtual IndexPageKey  pageKey()
};
```

## Appendix D Definition of Class LeafPage

```
template<typename DataRef,
         typename LeafPageKey,
         typename IndexPageKey
        >
class LeafPage: public Page< DataRef, LeafPageKey, IndexPageKey >
{
public:
        typedef LeafPageKey key_type;
        typedef size_t size_type;
        typedef CPage<LeafPageKey, IndexPageKey, DataRef>        page_type;
        typedef CIndexPage<LeafPageKey, IndexPageKey, DataRef> indexpage_type;
        typedef CLeafPage<LeafPageKey, IndexPageKey, DataRef>  leafpage_type;
        typedef page_type*              page_pointer;
        typedef indexpage_type*         indexpage_pointer;
        typedef leafpage_type *         leafpage_pointer;
        typedef std::pair<key_type, DataRef> value_type;
        typedef std::vector<value_type> container;
        typedef typename container::iterator iterator;
        typedef typename container::const_iterator const_iterator;
        typedef typename container::reverse_iterator reverse_iterator;
        typedef typename container::const_reverse_iterator const_reverse_iterator;
        typedef typename container::reference reference;
        typedef typename container::const_reference const_reference;
private:
        IndexPage* mParent;
        Container   mEntries;
public:
        LeafPage();
        LeafPage(const LeafPage&);
        virtual ~LeafPage();
        virtual size_type level() const;
        virtual LeafPage* getLeafPage();
        virtual Page* parent() const;
        virtual void   parent(Page*);
        iterator insert(const value_type& x);
        iterator insert(iterator pos, const value_type& x);
        template<typename InputInterater>
        void    insert(InputInterater first, InputInterater last);
        iterator find(const value_type&);
        iterator find(const key_type&);
        iterator find(const DataRef&);
        template<typename Predicate>
```

```
        iterator find(iterator first, iterator last, const Predicate& pre)
        iterator erase(iterator) ;
        void     erase(iterator, iterator);
        void clear();
        DataRef find(iterator pos) const;
        key_type pageKey();
        size_type size() const;
        bool empty() const;
        bool underflow() const;
        bool full() const;
        iterator begin();
        iterator end();
        reference front();
        reference back();
};
```

## Appendix E Definition of Class IndexPage

```
template<typename DataRef,
          typename LeafPageKey,
          typename IndexPageKey
         >
class IndexPage: public Page<DataRef, LeafKey, IndexPageKey, >
{
public:
        typedef std::vector< std::pair<Key, Page*> > Container;
        typedef Container::value_type value_type;
private:
        IndexPage* mParent;
        Size_type   mLevel;
        Container   mEntries;
public:
        IndexPage()
        IndexPage(indexPage*, const unsigned short)
        IndexPage(const IndexPage&)
        virtual ~IndexPage();
        virtual size_type level() const;
        virtual void level(const size_type);
        indexPage* getIndexPage();
        virtual Page* parent() const;
        virtual void   parent(IndexPage*);
        iterator insert(value_type&);
        iterator insert(iterator pos, value_type&);
        template<typename InputInterater>
        void    insert(InputInterater, InputInterater);
        iterator find(const value_type&) const;
        iterator find(const Key&);
        iterator find(Page*);
        template<typename Predicate>
        iterator find(iterator, iterator, const Predicate&);
        Page*   find(iterator) const;
        iterator    erase(iterator);
        size_type erase(value_type&);
        void        erase(iterator, iterator);
        void clear();
        Key pageKey();
        size_type size() const;
        bool empty() const;
        bool underflow() const;
        bool full() const;
```

```
        iterator begin();
        iterator end();
        reference front();
        reference back();
        iterator findLeastEnlarge(const Key&);
        iterator findLeastOverlap(const Key&);
};
```

## Appendix F Definition of Class RTree

```
template<typename DataRef,
         typename LeafPageKey,
         typename IndexPageKey,
         typename PageType = Page<DataRef, LeafPageKey, IndexPageKey>
        >
class CRTree
{
public:
        typedef PageType page_type;
        typedef typename page_type::leafpage_type  leafpage_type;
        typedef typename page_type::indexpage_type indexpage_type;
        typedef typename page_type::DataReference DataRef;
        typedef typename ikey_type::Point Point;
        typedef lkey_type key_type;
        typedef page_type*          page_pointer;
        typedef indexpage_type*     indexpage_pointer;
        typedef leafpage_type*      leafpage_pointer;
        typedef typename indexpage_type::iterator         indexpage_iterator;
        typedef typename indexpage_type::const_iterator indexpage_const_iterator;
        typedef typename indexpage_type::reverse_iterator   indexpage_reverse_iterator;
        typedef typename indexpage_type::value_type   indexpage_value_type;
        typedef typename leafpage_type::iterator          leafpage_iterator;
        typedef typename leafpage_type::const_iterator   leafpage_const_iterator;
        typedef typename leafpage_type::reverse_iterator leafpage_reverse_iterator;
        typedef typename leafpage_type::value_type       leafpage_value_type;
        typedef leafpage_value_type value_type;
        typedef size_t size_type;
        typedef typename kia::contain<indexpage_value_type, leafpage_value_type> Contain;
        typedef typename kia::equal <indexpage_value_type, leafpage_value_type> Equal;
        typedef typename kia::overlap<indexpage_value_type, leafpage_value_type> Overlap;
        typedef typename kia::within <indexpage_value_type, leafpage_value_type> Within;
private:
        page_pointer mRoot;
        size_type mSize;
public:
        CRTree();
        ~CRTree();
        class iterator;
        class Cursor;
        iterator begin();
        iterator end();
        size_type size() const;
```

```
        bool empty();
        size_type height();
        void insert(const value_type&);
        void find(const key_type&, Cursor&);
        iterator find(const key_type&);
        void find(const value_type&, Cursor&);
        template<typename Predicate>
        void find_if(Predicate&, Cursor&);
        void erase(iterator);
        void erase(const key_type&);
private:
        page_pointer chooseSubtree(const IndexPageKey&, const size_type level);
        template<typename KeyType>
        void adjustKey(const KeyType&, page_pointer);
        void adjustKeyErase(page_pointer N);
        template<typename KeyType>
        void adjustTree(const KeyType&, page_pointer, IndexPageKey&,
                                        page_pointer, IndexPageKey&);
        void condenseTree(page_pointer);
        void insert(indexpage_value_type&);
        double dist(const IndexPageKey&, const IndexPageKey&)
        double dist(const Point&, const Point&);
        template<typename PointerType>
        void split(PointerType, IndexPageKey&,   PointerType, IndexPageKey &);
};
```

## Appendix G Definition of Basic Predicate and Class Predicate

```
template<typename IndexEntry, typename LeafEntry>
struct equal : public std::binary_function<IndexEntry, LeafEntry, bool>
{
        typedef typename LeafEntry::first_type  LeafKey;
        typedef typename IndexEntry::first_type IndexKey;

        template<typename EntryType, typename KeyType>
        bool operator()(const EntryType&, const KeyType&)  const;

        bool operator()(const LeafEntry&,  const IndexKey&)  const;
};

template<typename IndexEntry, typename LeafEntry>
struct contain : public std::binary_function<IndexEntry, LeafEntry, bool>
{
        typedef typename LeafEntry::first_type  LeafKey;
        typedef typename IndexEntry::first_type IndexKey;
        template<typename EntryType, typename KeyType>
        bool operator()(const EntryType&, const KeyType&)  const;
        bool operator()(const LeafEntry&,  const IndexKey&)  const;
};

template<typename IndexEntry, typename LeafEntry>
struct overlap : public std::binary_function<IndexEntry, LeafEntry, bool>
{
        typedef typename LeafEntry::first_type  LeafKey;
        typedef typename IndexEntry::first_type IndexKey;

        template<typename EntryType, typename KeyType>
        bool operator()(const EntryType&, const KeyType&)  const;

        bool operator()(const LeafEntry&,  const IndexKey&)  const;
};

template<typename IndexEntry, typename LeafEntry>
struct within : public std::binary_function<IndexEntry, LeafEntry, bool>
{
        typedef typename LeafEntry::first_type  LeafKey;
        typedef typename IndexEntry::first_type IndexKey;

        template<typename EntryType, typename KeyType>
        bool operator()(const EntryType&, const KeyType&)  const;
```

```
        bool operator()(const LeafEntry&,  const IndexKey&)  const;
};

template<typename  BinFun1, typename  BinFun2, typename  KeyType>
class Predicate{
public:
        typedef IndexPage::value_type IndexValueType;
        typedef LeafPage::value_type  LeafValueType;
private:
        BinaryFun1 mPredHandle1;            //** for indexpage
        BinaryFun2 mPredHandle2;            //** for leafpage
        KeyType mKey;
public:
        Predicate(const BinFun1&, const BinFun2&, const KeyType&);
        Predicate(const Predicate&);
        ~Predicate();
        bool operator=(const Predicate&);
        bool operator()(const IndexValueType&) const;
        bool operator()(const LeafValueType &) const;
};
```

## Appendix H Definition of Class iterator

```
class iterator
{
private:
        leafpage_pointer mPtr;
        size_type mSlot;
public:
        iterator();
        iterator(leafpage_pointer, leafpage_iterator);
        DataRef operator * ()  const ;
        bool operator ==(const iterator&) const ;
        bool operator !=(const iterator&) const ;
        iterator& operator=(iterator&);
        iterator& operator++ ();
        iterator operator++ (int) ;
        iterator& operator-- () ;
        iterator operator-- (int) ;
        iterator operator+(int n);
        iterator operator-(int n);
        leafpage_pointer  getPointer() ;
        leafpage_iterator  getIterator();
};
```

## Appendix I Definition of Class Cursor

```
class Cursor {
public:
        typedef CRTree::iterator treeIterator;
        typedef vector<TreeIterator> Container;
        typedef Container::iterator iterator;
private:
        int mCurrent;
        Container mResults;
public:
        Cursor();
        Cursor(const Cursor&);
        ~Cursor(){}
        size_type size() const;
        void insert(LeafPage*, LeafIterator*);
        void insert(treeIterator);
        void erase(iterator);
        bool empty() const;
        iterator begin();
        iterator end();
        bool isBegin() ;
        bool isEnd();
        Cursor& operator=(const Cursor&);
        bool operator==(const Cursor&) const;
        bool operator!=(const Cursor&) const;
        void clear();
        LeafPage* getLeafPtr();
        LeafIterator* getLeafIter();
        treeIterator operator* () const ;
        Cursor& operator++() ;
        Cursor operator++(int) ;
        Cursor& operator--();
        Cursor operator--(int);
};
```

# Bibliography

[Abra01] David Abrahams, "*Generic Programming Techniques*", http:// www.boost.org/more/generic_programming.html, 2001.

[Alex01] Andrei Alexandrescu, "*Modern C++ Design: Generic Programming and Design Pattern Applied*", Addison-Wesley, 2001.

[ANSI97] ANSI/ISO, "*Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*", American National Standards Institute (ANSI), Nov. 1997.

[Beck90] N. Beckmann, H. P Kriegel, R. Schneider, B. Seeger, "*The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*", *Proc. of the ACM SIGMOD Conf., 1990*, 322-331.

[Bent75] J.L. Bentley, "*Multidimensional Binary Search Trees Used for Associative Searching*", CACM 18, 9(1975), 505-517.

[Berc96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel, "*The X-tree: An Index Structure for High-Dimensional Data*", Proc. of the 22nd Int. Conf. on Very Large Databases, pages 28-39, 1996.

[Brey02] Ulrich Breymann, "*Designing components with C++ STL: A New Approach to Programming*", Addison Wesley, 2002.

[Butler02] Greg Butler, Ling Chen, Xuede Chen, Ashraf Gaffar, Jinmiao Li, Lugang Xu, "*The Know-It-All Project: A Case Study in Framework Development and Evolution, Domain Oriented Systems Development: Perspectives and Practices*", Kiyoshi Itoh, Satoshi Kumagai, T. Hirota (eds), Taylor and Francis Publishers, UK, 2002.

[Chak99] Kaushik Chakrabarti, *"Supporting Spatial Index Structures as Access Methods in a DataBase System"*, (Master Thesis), University of Illinois, 1999.

[Doel02] Mario Doeller, Harald Kosch, *"Enhancement of Oracle's Indexing Capacities through GiST-implemented Access Methods"*, Institute of Information Technology, 2002.

[Fink74] R.A. Finkel, J.L. Bentley, *"Quad Trees, a data structure for retrieval on composite keys"*, Acta Informatica, 4(1974), 1-9.

[Gaff01] Ashraf Gaffar, *"Design of a Framework for Database Indexes"* (Master Thesis), Department of Computer Science, Concordia University, 2001.

[Gamm95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley, 1995.

[Garc98] Y.J. Garcia, M.A. Lopez, S.T. Leutenegger, *"On Optimal Node Splitting for R-trees"*, In Proc. of the 24th Int. Conf. on Very Large Databases, 334-344, 1998.

[Grand02] Mark Grand, *"Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML"*, (2nd Edition), Wiley, 2002.

[Gsch01] Thomas Gschwind, *"PSTL- A C++ Persistent Standard Template Library"*, 6th USENIX Conference on Object-Oriented Technologies and Systems, page 147-158, Jan. 29-Feb. 2, 2001, San Antonio, TX.

[Gutt84] Antonin Guttman *"R-Trees: A Dynamic Index Structure for Spatial Searching"*, Proc ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984.

[Heller95] Joseph. M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer, *"Generialized Search Tree for Database System"*, Proc. 21st Int. Conf. Very Large Data Bases, page 562-573 1995.

[HypDic] Hyperdictionary: http://www.hyperdictionary.com.

[John88] Ralph E. Johnson, Brian Foote, "*Design Reusable Classes*", Journal of Object-Oriented Programming, 1(2), pp 22-35, 1988.

[Kata97] Norio Katayama, Shin'ichi Satoh, "*The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries*", In Proceedings of ACM SIGMOD International conference on Management of Data, page 369-380, 5 1997.

[Kern98] Brian W. Kernighan, "*STL Tutorial and Reference Guide*", Addison-Wesley, 1998.

[Libgist] libgist v1.0/doc/libgist v.1.0 Release Notes, Berkeley.

[Meyers92] Scott Meyers, *"Effective C++"*, Addison-Wesley Publishing, 1992. ISBN 0-201-56364-9.

[Meyers95] Scott Meyers, *"More Effective C++"*, Addison-Wesley Publishing, 1995.

[Meyers01] Scott Meyers, "*Effective STL*", Addison-Wesley Publishing, 2001

[Muss96] David R. Musser, Gillmer J. Derge, and Atul Saini, "*STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*", Addison-Wesley, 1996.

[Muss03] David R. Musser, "*Generic Programming*", Rensselaer Polytechnic Institute, http://www.cs.rpi.edu/~musser/gp/, May 19, 2003.

[Myer79] Myers, Glenford J., *"The art of software testing"*, Wiley, c1979. ISBN: 0471043281

[Rieh98] Dirk Riehle, Frank Buschmann, *"Pattern Languages of Program Design 3"*, Addison-Wesley, 1998.

[Rijk99] Amaut de Rijk, "*Improving the R*-tree Storage Allocation Algorithm*" (Master thesis), Delft Universiy of Technology, 1999.

[Rob81] J. T. Robinson, "*The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes*", In Proc. ACM SIGMOD Int. Conf. on Management of Data, 10 – 18, 1981.

[RtreePortal] www.rtreeportal.org

[Sanc00] Arturo Sanchéz-Ruéz, "*Standard Template Library*", University of Massachusetts Dartmouth, http://vega.cocse.unf.edu/~asanchez/ms-stl/web, 2000.

[Sim99] Volker Simonis, "*Chameleon Objects, or how to write a generic, type safe wrapper class*", Willhelm-Schickard-Institute fur Informatik, 22 Nov 1999

[Step95] Alexander Stepanov, Meng Lee. "*The Standard Template Library*", Hewlett-Packard Company, Palo Alto, 1995.

[Strou97] Bjarne Stroustrup, "*The C++ Programming Laguage*". Addison-Welsey, 3rd edition, 1997.

[UCalifornia] http://www.cs.ucr.edu/~marioh/spatialindex/readme.html

[Van02] David Vandevoorde, Nicolai M. Josuttis "*C++ Templates: A Complete Guide*", Addison Wesley, Nov12, 2002.

[White96] D. A. White, Ramesh Jain, "*Similarity Indexing with the SS-tree*", In International Conference on Data Engineering (ICDE), page 516-523, New Orleans, LA, March 1996.

[Wise95] G. Bowden Wise, "*An Overview of the Standard Template Library*", http://www.cs.rpi.edu/~wiseb/xrds, 1995.