A B+-tree Index for the Know-It-All Database Framework

Jingxue Zhou

in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2003

Your file Votre référence
ISBN: 0-612-83927-3
Our file Notre référence
ISBN: 0-612-83927-3

ABSTRACT

A B+-tree Index for the Know-It-All Database Framework

Jingxue Zhou

An efficient implementation of search trees is crucial for any database systems. The B+-tree is one of the most widely and studied data structures and provides an efficient index structure for databases. The Index subframework is a component of the Know-It-All database framework. It covers tree-based indexes such as B+-tree, R-tree, X-tree and SS-tree, including sequential queries, exact match queries, range queries, approximate queries, and similarity queries. Our B+-tree implementation is a proof-of-concept for the Index subframework. Our B+-tree index is designed to be a container by following the STL style in C++ and implemented by using design patterns and generic programming techniques. Therefore, the B+-tree index can adapt to different key types, data types, different queries, and different database application domains, and be easy and convenient for developers to reuse.

Acknowledgements

I would like to thank my supervisor, Dr. Butler Gregory, for his valuable guidance, encouragement, patience, support and hard work throughout my study and thesis work.

I also extend my gratitude to my groupmates, Bin Nie and MinAn Zhong. Working in the group and discussing with them are always pleasant and enjoyable.

I really would like to thank my wife, Ying Liu. It is no doubt that without her tremendous encouragement and unfailing support, I would not have made it throughout my studies.

I dedicate this thesis to my wife Ying Liu and our soon-to-be-born son.

Contents

List of Figures And Tables

Chapter 1 Introduction

Database indexes are the search engines for database management systems. An index is commonly used to enhance database performance, but an efficient and effective index will result in a better quality, and quicker responding database.

To achieve this goal, a specialized handcrafted index is a good way to support a specific database application in a specific domain using domain-specific access methods. However, these specialized access methods are usually hand-coded from scratch. As a result, a specialized index has generally better code efficiency and performance but the tradeoffs are development time and cost associated with the customized implementations. Furthermore, the effort required to implement and maintain them is high.

Another choice is to develop a framework for a family of indexes, and reuse it to develop different indexes for different applications. A framework is a skeletal group of software modules that may be tailored for building domain-specific applications, typically resulting in increased productivity and faster time-to-market. Therefore, the way to specialize an index framework can largely reduce the cost of providing a new index.

1.1 The Problem and Related Work

1.1.1 The Problem.

The Generalized Index Search Tree, GiST [HNP95] is an existing framework of a generalized index system. It has friendly hot spots that can be adapted to different key types and access methods. However, GiST has tried to satisfy all the possible needs of the future members of a family of applications at a time, so it often leads to an even more generic code that is larger and less user-friendly. In addition, the source code itself, largely influenced by the C programming language, has poor object-oriented style.

As the framework development methodologies improve, these problems are being recognized and addressed. The Know-It-All Framework [BCC+2002] is an object-oriented framework for database management systems with a good potential for improving the process of developing better quality software.

1.1.2 Related Work

The Know-It-All Project [BCC+2002] has been underway at Concordia University. This project is investigating methodologies for the development, application and evolution of frameworks. A concrete framework for database management systems is being developed as a case study for the methodology research. The aims of this project are to research methodologies and models for framework evolution, to develop a framework for database management systems, and to apply the framework to advanced database applications for bioinformatics. It is written in C++, with some Java for user interfaces, and XML for communication of data between the C++ framework and the Java tools. The user interfaces will provide a full range of query mechanisms, from icons for canned queries, to forms, to textual queries in set comprehension languages, and diagrammatic queries.

The KIA (short for Know-It-All) project, a framework for DBMS that supports a variety of data models of data and knowledge, and Case study is the integration of different paradigms and heterogeneous databases [BCC+2002]. KIA involves multiple subframeworks that are integrated together in an adaptable DBMS context. It started by supporting the traditional relational database model, and it is expanding to support other types of database applications using different data models. Eventually, it will be applied to advanced applications in bioinformatics.

The Tree Index Framework is one of KIA subprojects. It is being developed in C++ to conform to the style of the Standard Template Library (STL) design of collections and iterators. The index subframework covers tree-based indexes such as B+-tree, R-tree, X-tree and SS-tree, including multi-dimensional trees and similarity-based retrieval. It also covers sequential queries, exact match queries, range queries, approximate queries, and similarity queries.

1.2 Our Work

B+-trees are the most common dynamic index structures in database systems. The B+-tree index structure is one of the most widely used and studied data structure because it is height-balanced, multi-way, and has external file organization.

In his thesis titled "Design of A Framework for Database Indexes" [Gaf2001], Ashraf Gaffar designed the indexing framework to follow the STL style. The design

separates Index, Data, and Data Reference, uses iterators to define both positions within indexes or files, as well as to refer to a collection of information. Allocators hide the memory management issues, such as the use of buffers for IO from disk to memory for index pages and leaf pages. The Data Reference is a smart pointer or proxy that hides whether the actual location is in memory or on disk.

As we know, a framework is a procduct-line architecture, plus an implementation, plus documentation that captures the intended use of the framework for building applications. The implementation of a framework is an important part to be reused. However, when we implemented a B+-tree index to follow Ashraf Gaffar's design, we found some problems required to be addressed and improved. In his framework design, there is no base class for index pages and leaf pages. This causes too much type-casting in the implementation of the B+-tree index. In addition, allocators are too complicated because they do not only take care of memory, but also control access to disk.

In our implementation, we have redesigned the B+-tree index to follow the design of the STL containers in C++, and good object-oriented design patterns. In order to avoid too much type-casting, we add a base Page class, use design patterns and combine static polymorphism and dynamic polymorphism mechanism. Furthermore, we separate the responsibilities of allocators into two parts. Allocators are only responsible for the memory management issues, and we use Proxy to load a page from disk on demand and maintain the reference to the loaded page.

The B+-tree Index is designed to be a container that provides an iterator to its contents. The only way to interact with the container is through its iterator. Because we implement the B+-tree index by using design patterns and generic programming techniques, the B+-tree Index can be easily reused anywhere like other containers in the STL.

1.3 Contribution of the Thesis

The work described here involves the design and implementation of a suite of C++ classes, and correctness and performance testing for a B+-tree index.

The major concerns are:

- redesigning and implementing the B+-tree index to follow the STL style
- using design patterns and generic programming
- using a proxy to load a page on demand, and to maintain the reference to the loaded page.

1.4 Layout of This Thesis

Some background on reusable tree-based indexes is presented in Chapter 2. Then the B+-tree design is presented in Chapter 3. Chapter 4 describes the implementation of B+-tree and testing in details. Finally, a conclusion is made in Chapter 5.

Chapter 2 Background

In this chapter, after reviewing database index and B+-tree index, we simply introduce the relationship between static and dynamic polymorphism, the STL style, the chameleon technique, and the design patterns that will be used in our design and the implementation of the B+-tree index.

2.1 Database Index

Suppose we have a textbook lying around. If we want to find all pages relating to "database index", the best way is to look in the index that says "database index, pages 55-65". We just flip straight to the appropriate pages and get the information we need. Thus we do not have to scan the whole book, which would take a long time.

The index of a book provides a way to find a topic quickly. Such an index is a table containing a list of topics (keys) and numbers of pages where the topic can be found (data reference). Indexes in databases work the same way as the index of our book does. A database is any organized collection of information. An index is an auxiliary data structure intended to help speed up the retrieval of information in response to certain search conditions.

An index [GDW2000] is "a data structure that allows for random access to arbitrary data within a field, or a set of fields. In particular, an index lets us find a record without having to look at more than a small fraction of all possible records." From this definition, we can see that an index [BM1972] consists of "index elements which are pairs (x,a) of fixed size physically adjacent data items, namely a key x and some associated information a. The key x identifies a unique element in the index, and the associated information is typically a pointer to a record or a collection of records in a random access file." All indexes are based on the same basic concept—Key and Pointer.

There are many different data structures that serve as indexes [GDW2000]: simple indexes on sorted files, secondary indexes on unsorted files, tree-based indexes and hash tables. Many of the advantages associated with simple index and secondary index are tied to the assumption that the index file is small enough to be loaded in memory in its entirety. When a simple index is too large to be held in memory, we should consider using a hash table that is another useful and important index structure, and a tree-based index that is a commonly used way to build indexes on any file. Although a hashed

organization can improve access speed, a tree-structured index such as B-tree and B+-tree has more flexibility in both keyed access and ordered, sequential access.

The B-tree and its variant B+-tree are efficient data structures that are widely used as tree-based multilevel indexes in database systems. They had already become so widely used that Comer was able to state that "the B-tree is, de facto, the standard organization for indexes in a database system" in his survey article titled "The Ubiquitous B-Tree" [Com1979]. However, B+-trees can support true indexed sequenctial access as virtual trees, and possibly compress separators and potentially produce an even shallower tree than B-trees [FZ1992], so we choose the B+-tree index to be implemented first in the KIA framework.

2.2 B+-tree Index

Since the B-tree is the basic foundation for the B+-tree, it will be helpful to understand the B-tree first.

2.2.1 B-tree

In 1972, R. Bayer and E. McCreight announced B-trees to the world in their article titled "Organization and Maintenance of Large Ordered Indexes" [BM1972]. This article describes the theoretical properties of B-trees and includes empirical results. Then Comer's survey in 1979 [Com1979] provides an excellent overview of some important variations on the basis of B-trees.

In Bayer and McCreight's article, B-trees are designed to solve how to access and maintain efficiently an index that is too large to hold in memory, so the index itself must be external and is organized in pages that are blocks of information transferred between main memory and backup storage like hard disks.

A B-tree of order n [FZ1992] is a multi-way search tree of order n with the following properities:

1. Every page has a maximum of n children
2. Every page except the root and the leaves has at least $\lceil n/2 \rceil$.
3. The root is either a leaf or has at least two children
4. All the leaves are on the same level.
5. A nonleaf page with m children contains $m-1$ keys.
6. A leaf page contains at least $\lceil n/2 \rceil$-1 keys and no more than $n-1$ keys.

In a B-tree, one tree node can be made to correspond to a page. Non-leaf pages are called index pages here as shown in Figure 2.1, containing search keys and pointers to lower-level children. Leaf pages as shown in Figure 2.2 contain search keys and references to data items or records.

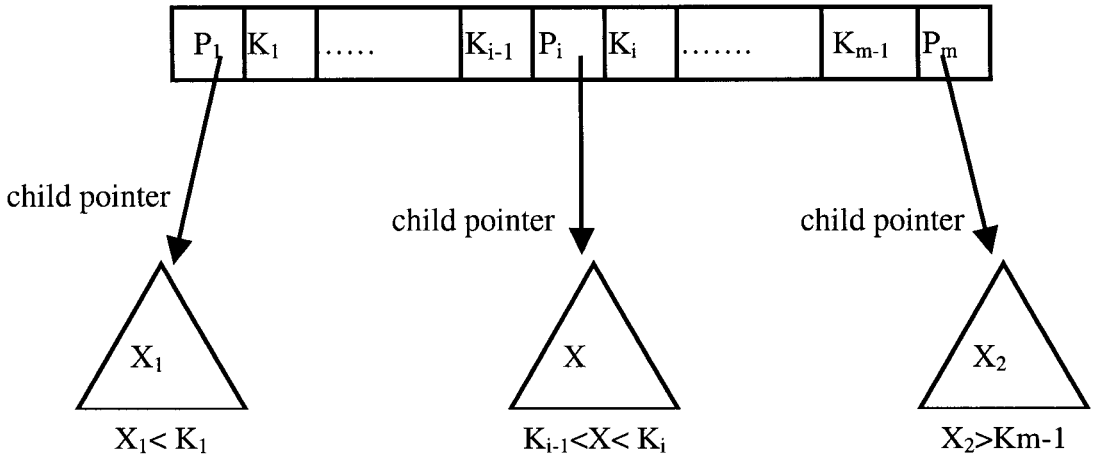

Figure 2.1 B-tree Index Page with m-1 search keys

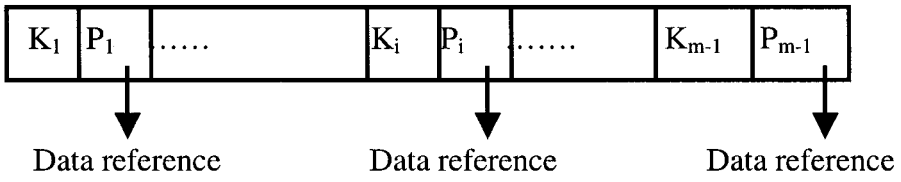

Figure 2.2 B-tree leaf page with m-1 search keys

For leaf pages in Figure 2.2, P_i points to either a file record with search key value K_i, or a bucket of pointers to records with that search key value. A bucket structure is used if the search key is not a primary key, and the file is not sorted in search key order.

For non-leaf pages, index pages form a multilevel index on leaf pages. In each index page as shown Figure 2.1 where $m<n$ (n is the order of the B-tree), Pi is a child pointer to a subtree containing search key values between K_{i-1} and K_i. Pointer P_m points to a subtree containing search key values that are greater than K_{m-1}. Pointer P_1 points to a subtree containing search key values that are less than K_1.

Each page, except the root and leaf pages, has at least $n/2$child pointers but has at most n child pointers. Within each page, keys are ordered, such that $K_1<K_2<\ldots<K_{m-1}$.

A pair (K_i, P_i) is also called an entry. The root page has at least two child pointers unless it is the only page in the tree.

The power of B-trees lies in the following significant advantages:

1. Storage utilization is guaranteed to be at least 50% and should be considerably better in the average [BM1972].
2. The balance is maintained dynamically at a relatively low cost. No overly long branches exist, and random insertions and deletions are accommodated to maintain balance [FZ1992].

2.1.2 B+-tree

One of the major drawbacks of the B-tree is the difficulty of traversing the keys sequentially. The B+-tree is designed to compensate this shortcoming. "In a B+-tree, all keys reside in the leaves. The upper levels, which are organized as a B-tree, consist only of an index, a road map to enable rapid location of the index and key parts" [Com1979].

Figure 2.3 shows the logical separation of the index and leaf parts, which allows index pages and leaf pages to have different formats or even different sizes. In particular, leaf pages are usually linked together left-to-right. To improve the reverse traversal, we allow leaf pages to be double linked together. The double linked list of leaves is referred to as the *sequence set*. The key and record information is not in the upper-level, tree-like portion of the B+-tree but contained in the sequence set. The leaves are linked together to provide a sequential path for traversing the keys in the tree. Index access to this *sequence set* is provided through a conceptually (though not necessarily physically) separate structure called the *index set*. In a B+-tree the *index set* consists of copies of the keys that represent the boundaries between *sequence set* blocks. These copies of keys are called *separators* since they separate a sequence set block from its predecessor. Keys are replicated in index pages to define paths for searching individual records.

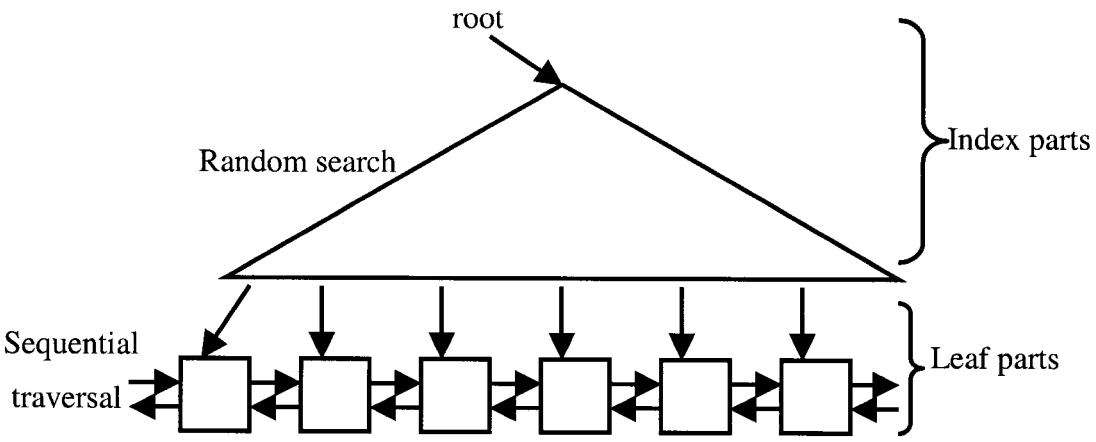

Figure 2.3 B+-tree with separate index and leaf parts.

Because of the implications of having an independent index and sequence set, and all the operations for the sequence set, the B+-tree can be designed to be a STL container. Leaf parts can be accessed through iterators but index parts are invisible for users.

Insertion and find operations in a B+-tree are processed in a way similar to insertion and search operations in a B-tree. Find operations differ from those in a B-tree in that searching does not stop if a key in the index equals the query value. A searching processes from the root of a B+-tree through the index of a leaf. Since all the keys reside in the leaves, it does not matter what values are encountered as the finding progresses as long as the path leads to the correct leaf. Insertions into B+-tree are similar to B-tree insertions except that when a page is split, the middle key is retained in the left half page as well as being promoted to the parent.

The B+-tree retains the search and insertion efficiencies of the B-tree but increases the efficiency of searching the next record in the tree from **O**(log N) to **O**(1). Instead, the nearest right or left pointer is followed, and the search proceeds all the way to a leaf.

During the deletion in a B+-tree [Com1979], the ability to leave non-key values in the index part as separators simplifies the operation. The key to be deleted must always reside in a leaf so its removal is simple. As long as the leaf remains at least half full, the index need not be changed, even if a copy of a deleted key is propagated up into it.

Queries on B+-tree

Algorithm Find finds all index records with a search key value of *k*, given a B+-tree whose root is *T*.

F1 [Find subtrees] If *T* is not a leaf, examine keys of *T* to find the smallest search key value *Ki*>*k*. For *Ki*, invoke Find on the corresponding pointer *Pi*.

F2[Find leaf node] If *T* is a leaf, check keys to find the first data reference with the key value of *k*.

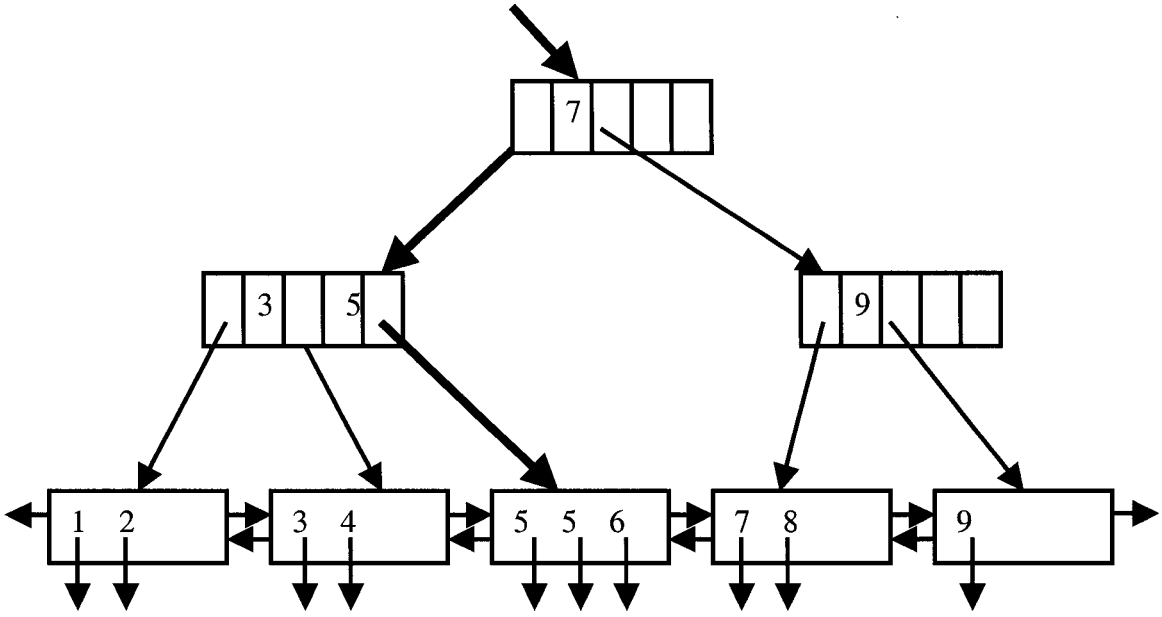

Figure 2.4 Sample B+-tree querying process: find(5)

Figure 2.4 shows a querying process with the search key 5 in a B+-tree of order 3. Finding begins with the root of the tree, and key comparisons direct to the right position of a proper leaf. Here the thicker lines represent the search path.

In processing a query, we traverse a path from the root to a leaf page. Except for the root, each page in a B+-tree has at least d=*n/2* (n is the order of the tree) children since there are between d and 2d keys per page. The root has at least 2 children. The height h of a tree with *s* total keys is therefore $h \leq log_d^{(s+1)/2}$ [Com1979]. Thus, the cost of processing a find operation grows as the logarihm of the file size. A B+-tree of order 50 which indexes a file of one million records can be searched with only 4 disk access in the worst case.

B+-tree supports not only equality queries but also range queries efficiently. Range queries use the forward/backward pointers in the leaf nodes to get all the records in the requested range.

Insertion and deletion are more complicated than searching, as they may require splitting or merging nodes to keep the tree balanced.

Insertion into B+-tree Algorithm

Algorithm Insert inserts a new entry *E* with the search key of *k* into a B+-tree.

I1 [Find position for new entry] Find a leaf page *N* where search key value *k* should appear.

I2 [Insert entry into page] If *N* has room for another entry, insert *E* into *N*. Otherwise, move half of the entries into a new page *NN*. This is called **splitting.**

I3 [Propagate changes upward] let P be the parent page of *N* and *NN* if a split was performed. Let *E* be a new entry with the pointer to *NN*. Set *N=P* if *N* is not the root,Repeat from I2.

I4 [Grow taller] If page split propagation causes the root to split, create a new root whoes children are the two resulting pages.

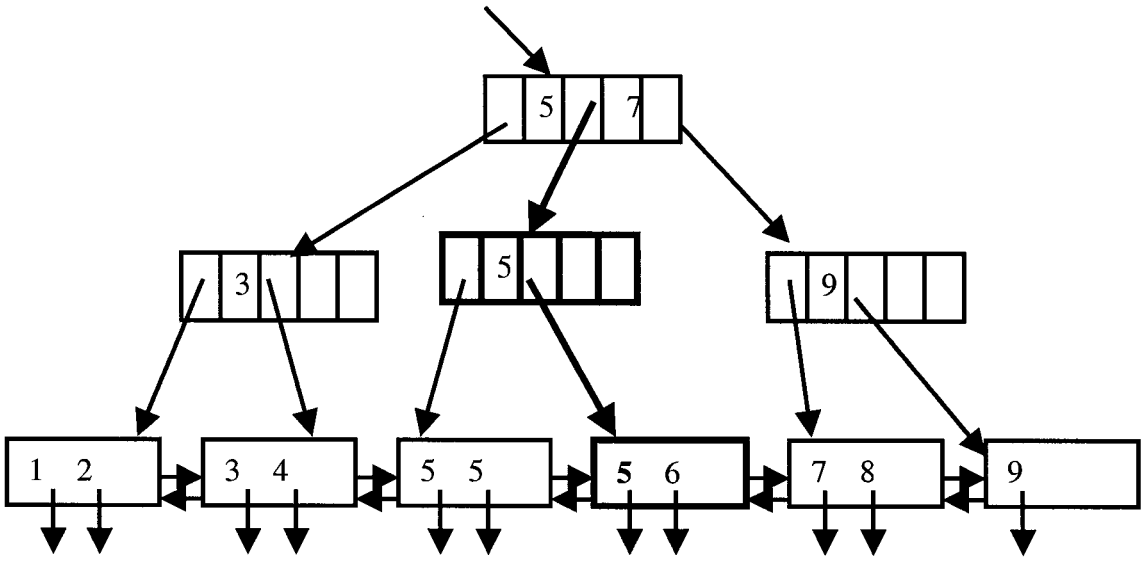

Figure 2.5 Sample B+-tree inserting process:insert(5)

During an insertion operation, first a search from the root is performed to locate the proper leaf for the insertion. Then the insertion is performed and balance is restored by a

procedure which moves from the leaf back toward the root. Referring to Figure 2.4, Figure 2.5 shows the tree after the key "5" was inserted. Because the appropriate leaf (the third leaf) is already full, a split occurs: a new page (the fourth leaf) is created and a lager half in the old page is moved into the new one. Then a copy of the middle key (the first key of the new page) is promoted to its parent where it serves as a separator. Usually the parent page will accommodate an addition key and insertion process terminates. But if the parent happens to be full too, then the splitting process is applied again. However, the primary difference between the splittings of index pages and leaf pages is that the middle key is promoted to the parent in the index page splits. Here the boxes with the thicker border represent new pages created during a splitting process. In the worst case, splitting propagates all the way to the root and the tree increase in height by one level.

Deletion from B+-tree

Algorithm Erase removes index entry *E* from a B+-tree.

D1 [Find leaf page containing *E*] Find the leaf page Ncontaining *E*. Stop if the entry was not found.

D2 [Erase entry] remove *E* from *N*.

D3 [Propagate changes] If *N* was sparse (the number of values is less then *n/2*, wher *n* is the order of the tree), check to determine whether to borrow from its right or left sibling. If yes, moving some into *N* makes it not sparse, and update the search key of its borrowing sibling in its parent. If not, merge with a sibling *NN*. Let *P* be the parent of *NN* and *E* be the entry of *NN* in *P*. Set *N=P* if *N* is not the root, repeat from D2.

D4 [Shorten tree] If the root page has only one child, make the child the new root.

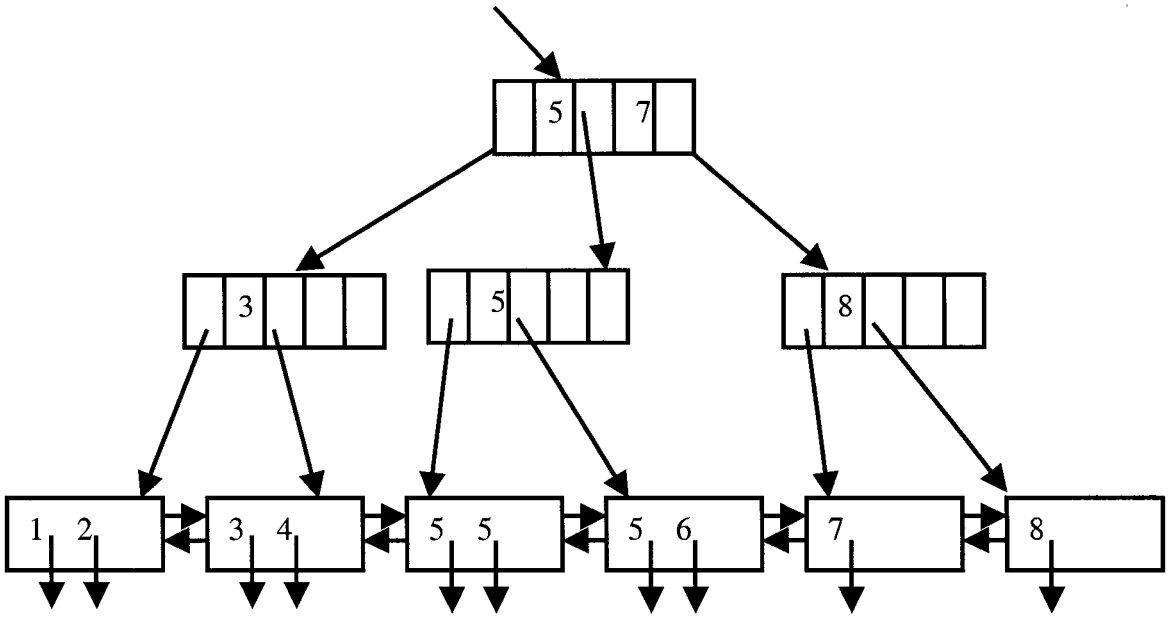

Figure 2.6 Sample B+-tree deleting process: erase(9)

Deletion also requires a find operation to identify the proper leaf. If the key to be erased resides in a leaf, it can just be removed. After that, we must check if the number of the elements in the leaf is less than half of the order. If yes, then underflow occurs and redistribution of the keys becomes necessary. Restoring balance could be obtained by borrowing from a neighboring leaf. Redistribution is shown in Figure 2.6 after the key "9" is deleted. The key "8" is borrowed from the left neighbor. But if the two neighbors are not sufficient, a merge must occur. Merge, which is the inverse of splitting, combines two pages into one, and the other is discarded. Since only one page remains, the key separatoring the two pages in the parent is no longer necessary and it will be deleted from the parent. If the parent happens to be sparse too, then the same merging process is applied again. Finally, if the children of the root are merged, they form a new root, decreasing the height of the tree by one level.

An insert or erase operation may require additional secondary storage access beyond the cost of a find operation. Overall, the costs are ast most doubled, so the height of the tree still dominates the expressions for these costs. Therefore, in a B+-tree of order n for a file of s records, insertion and erasion take time proportional to $log_{n/2}S$ [Com1979]. In all cases, the splits/merges are amortized over $n/2$ operations.

2.2 Templates and Generic Programming

Templates provide a generic way to reuse source code as opposed to inheritance and composition that provide a way to reuse object code. By using templates, we can design a single class that operates on data of many types, instead of having to create a separate class for each type. Also called *parametized types*, templates provide specifications for general-purpose classes and functions that automatically specialize themselves to new uses. In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

2.2.1 Templates

Templates are mechanisms for generating functions and classes based on type parameters (templates are sometimes called "parameterized types"). With respect to their usage, there are two basic types of templates: function templates and class templates. A class template [DWH1997] allows the compiler to generate multiple versions of a class type by using type parameters. A function template [DWH1997] allows us to define a group of functions that are the same except for the types of one or more of their arguments or objects.

Template parameters

Template parameters [VM2002] are declared in the introductory parameterization clause of a template declaration. There are three kinds of template parameters: type parameters, nontype parameters, and template template parameters.

Type parameters are by far the most commonly introduced with either the keyword typename or the keyword class (the two are entirely equivalent). Nontype parameters stand for constant values that can be determined at compile or link time. The type of such a parameter must be one of an integer type or enumeration type, a pointer type, and a reference type. For example: *template<int> class X2{}*. Template template parameters are placeholders for class templates. They are declared much like class templates, but the keywords struct and union cannot be used. For example: *template< template<class> class T > X3{};*

Template arguments

Template arguments [VM2002] are the values that are substituted for template parameters when instantiating a template. These values can be determined using several different mechanisms: explicit template arguments, injected class name, default template arguments and argument deduction.

A template name can be followed by explicit template argument values enclosed in angle brackets. The resulting name is called templated-id. For example: *X1<int, char> x;* Within the scope of a class template X with template parameters *P1, P2, ...* the name of that template(X) can be equivalent to the templated-id *X<P1, P2,...>*. Explicit template arguments can be omitted from class template instances if default template arguments are available. For example: *template<class T, class U=char > X1{...}; X1<int> x;* However, even if all template parameters have a default value, the angle brackets must be provided.

Function template arguments that are not explicitly specified may be deduced from the types of the function call arguments in a call. If all the template arguments can be deduced, no angle brackets need to be specified after the name of the function template.

Template specialization

Template specialization lets templates deal with special cases. Template specialization is a very powerful feature that approximates overloading. The template parameters are specified in the angle bracket enclosed list that immediately follows the template keyword.

For example, if the template is *SmartPtr<T>*, it can be specialized in a concrete type like *SmartPtr<int>*. This gives you good granularity in customizing behavior.

A template can also be partially specialized with multiple parameters. Partial template specialization gives us the ability to specialize a class template for only some of its arguments. For example, *template <class T, class U> class SmartPtr{...};* we can specialize *SmartPtr<T,U>* for integer and any other type, with the following syntax:
Template <class U> class SmartPtr<int, U> {...}

The compile-time and combinatorial nature of templates makes them very attractive for creating design artifacts, but it is easy to stumble on some problems that are not self-evident [VM2002]. The structure of a class (its data members) cannot be specialized by using template alone. Specialization of member functions does not scale. The functions of

a class with one template parameter can be specialized, but individual member functions for templates with multiple template parameters may cause problems.

2.2.2 Generic Programming

In the context of C++, generic programming is sometimes defined as programming with templates. In this sense, any use of templates could be thought of as an instance of generic programming. Templates can be also used to implement polymorphism (Static polymorphism).

Polymorphism [VM2002] is the ability to associate different specific behaviors with a single generic notation. The polymorphic behaviors are implemented through inheritance or templates. A polymorphism supported mainly via inheritance, which is handled at run time, is usually thought of as dynamic polymorphism. The different specific behaviors handled via templates at compile time are referred to as static polymorphism.

Dynamic polymorphism in C++ exhibits a lot of strengths such as heterogeneous collections handled elegantly and smaller executable code size potentially. In contrast, static polymorphism [VM2002] has such merits as implementing collections of built-in type easily and potentially faster generated code, etc. Static polymorphism is often regarded as type safer than dynamic polymorphism because all the bindings are checked at compile time.

Static polymorphism leads to the concept of generic programming. Although there is no one universally agreed-on definition of generic programming, generic programming is practical exactly because it relies on static polymorphism, which resolves interfaces at compile time.

According to [CE2000], Generic programming is about representing domains as collections of highly general and abstract components, which can be combined in vast numbers of ways to yield very efficient concrete programs. The term generic programming has at least four different meanings:

1. Programming with generic parameters
2. Programming by abstracting from concrete types
3. Programming with parameterized components
4. Programming method based on finding the most abstract representation of efficient algorithms.

The first meaning is the most common: generic parameters are type or value parameters. With generic parameters, code duplication can be avoided in statically typed languages.

The Standard Template Library (STL) is a general-purpose C++ library of algorithms and data structures, originated by Alexander Stepanov and Meng Lee. The STL, based on generic programming, is part of the standard ANSI/ISO C++ library. The STL is implemented by means of the C++ template mechanism and provides a powerful set of components for generic programming.

2.3 The STL Style

The Standard Template Library (STL) [SL1995], is a template-based C++ library of generic data structures and algorithms that work together in an efficient and flexible fashion. As the authors state in the STL specification: "The Standard Template Library provides a set of well-structured generic C++ components that work together in a seamless way. Special care has been taken to ensure that all the template algorithms work not only on the data structure in the library, but also on built-in C++ data structures."

There are six components as shown in Figure 2.7 in the STL organization. Three components, in particular, can be considered the core components of the library: template-based container classes, iterators and generic algorithms (template functions). The remaining three components of the STL are also fundamental to the library and contribute to its flexibility and portability: allocators, adapters and functors (function objects). Figure 2.7 shows that the relations among STL components. Generic algorithms assisted with function objects use iterators to access the elements in containers, and they can be used in different containers that use different allocators to manage and control memory.

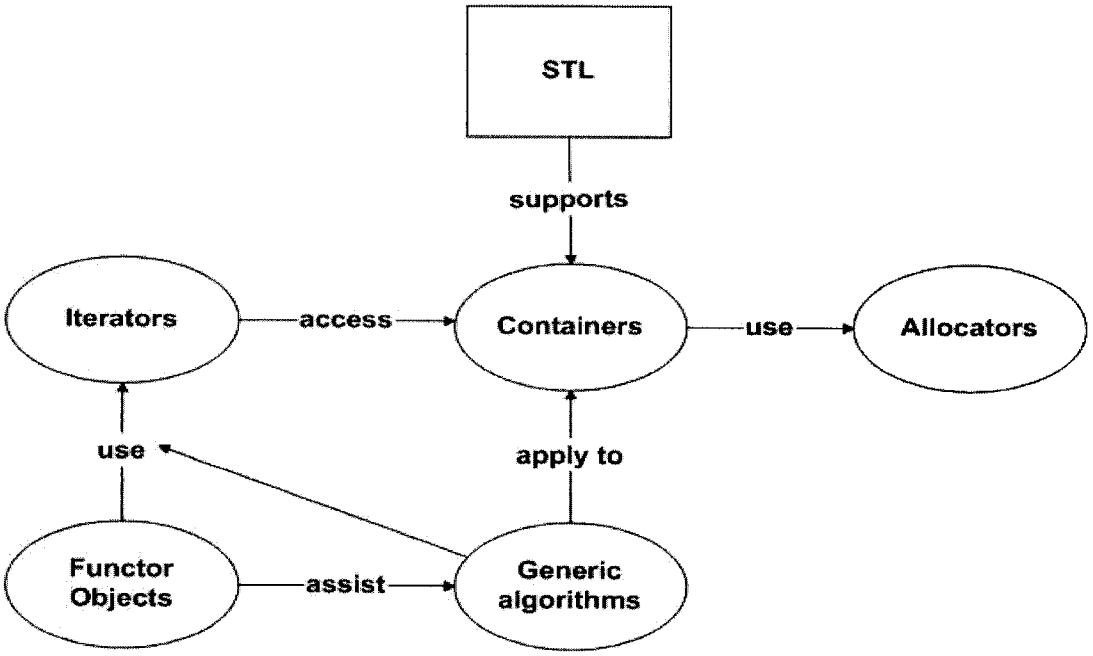

Figure 2.7 STL components

An STL data structure or container, unlike traditional ones, does not contain many member functions. STL containers contain a minimal set of operations for creating, copying, and destroying the container along with operations for adding and removing elements. Instead, algorithms such as functions for examining or sorting the elements in a container have been decoupled from the container and can only interact with a container via traversal by an iterator. This orthogonal component structure is illustrated in Figure 2.8

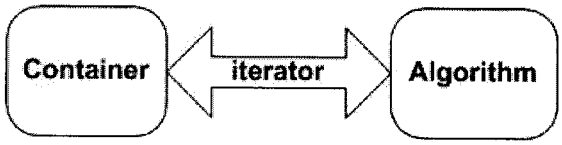

Figure 2.8 Orthogonal component structure

2.3.1 Containers

Containers [SL1995] are objects that store other objects and are responsible for the allocation and deallocation of those objects through constructors, destructors, insert and erase operations. Elements are stored in containers as whole objects; no pointers are used to access the elements in a container. This results in containers that are type safe and efficient.

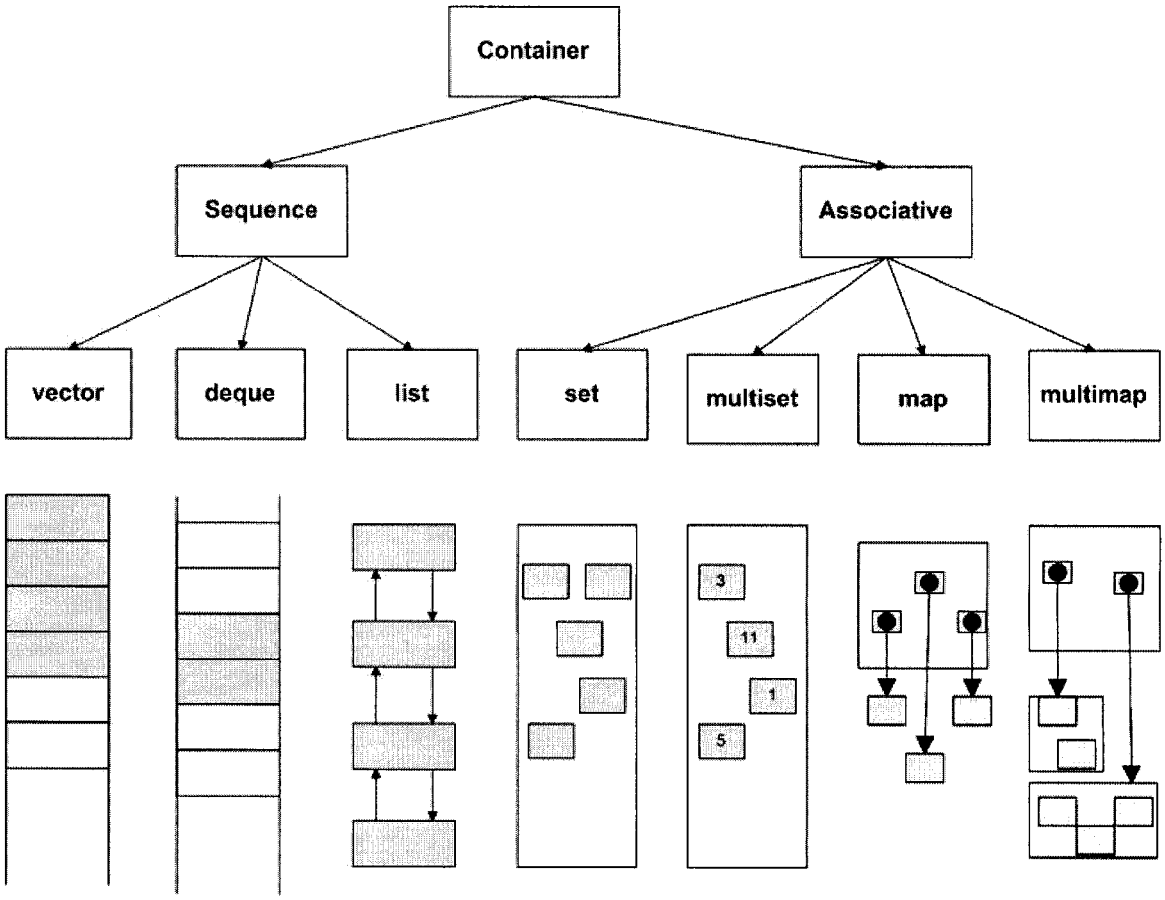

Figure2.9 STL containers

The STL provides two categories of containers as shown in Figure 2.9.

Sequence containers

Sequence containers store elements in sequential order. These containers group a finite set of elements in a linear arrangement. The STL includes class templates for vectors, lists, and deques:

vector<T> provides array-like random access to a sequence of varying length, with constant time insertions and deletions at the end.

deque<T> provides random access to a sequence of varying length, with constant time insertions and deletions at both ends.

list<T> provides linear time access to a sequence of varying length, with constant time insertions and deletions anywhere.

Associative containers

Associative containers store elements based on a key value. Implemented as red-black trees, they provide efficient retrieval of elements based on their key. The STL provides class templates for maps, multimaps, sets, and multisets. Maps and multimaps allow arbitrary data to be associated with each key. Also, maps and sets only allow elements with unique keys, whereas multimaps and multisets may contain elements with duplicate keys.

set<Key, Compare> supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves.

multiset<Key, Compare> (the number in boxes means times repeated) supports duplicate keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves.

map<Key, T, Compare> supports unique keys (contains at most one of each key value) and provides for fast retrieval of another type T based on the keys

multimap<Key, T, Compare> supports duplicate keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of another type T based on the keys.

Containers have different time and space complexities for their different operations:

	C array	vector	deque	list	set, multiset, map, multimap
Insert/erase at start	*n/a*	O(n)	O(1)	O(1)	O(logN)
Insert/erase at end	*n/a*	O(1)	O(1)	O(1)	O(logN)
Insert/erase in middle	*n/a*	O(n)	O(n)	O(1)	O(logN)
Access first element	O(1)	O(1)	O(1)	O(1)	O(logN)
Access last element	O(1)	O(1)	O(1)	O(1)	O(logN)

Access middle element	O(1)	O(1)	O(1)	O(n)	O(logN)
Overhead	none	low	medium	high	high

O(1) means constant time, O(n) means linear time, O(logN) means logarithmic time.

Table 2.1 Time and space complexities of containers

2.3.2 Iterators

Containers, by themselves, do not provide access to their elements. Instead, iterators are used to traverse the elements within a container. Iterators [SL1995] are very similar to smart pointers and have increment and dereferencing operations. By generalizing access to containers through iterators, the STL makes it possible to interact with containers in a uniform manner.

Iterators are a generalization of pointers that allow a programmer to work with different data structures (containers) in a uniform manner. An iterator can be thought of as an object, which can point to any value in the container. It may either point to an element of this container, or beyond it, using the special past-the-end value as shown in Figure 2.10.

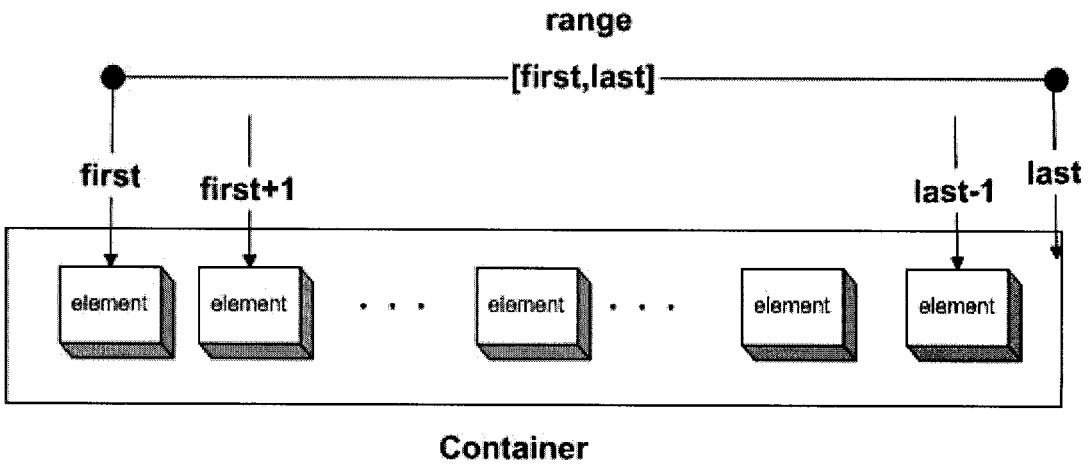

Figure 2.10 iterator activity

Iterators are the cornerstone of the STL design and give the STL most flexibility. Iterators are the glue that connects algorithms to containers. Instead of developing algorithms for a specific container, they are developed for a specific iterator category. This strategy makes it possible to use the same algorithm with a variety of different containers.

Iterators [SL1995] are classified into five categories: forward, bidirectional, random access, input, and output. The Iterator hierarchy is shown in Figure 2.11. Each category forms a set of requirements that must be met by concrete iterator types within that category. 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 in the STL must satisfy complexity requirements. These requirements ensure that algorithms written in terms of iterators will work correctly and efficiently.

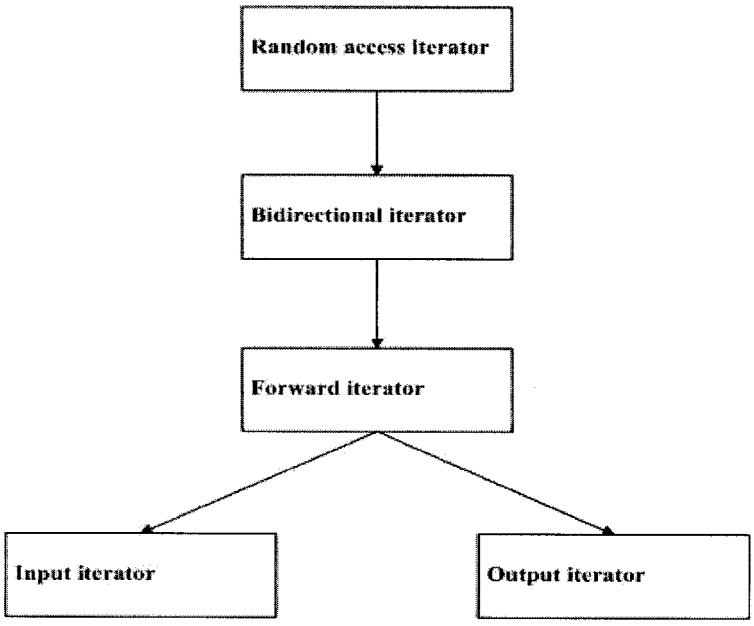

Figure 2.11 Iterators hierarchy

- A Random access iterator is the most powerful one. It can be used to store and retrieve values, provide for bidirectional traversal, and supports the following operations: ++, --, +, - . It can also be indexed via the [ ] operator.
- A Bidirectional iterator provides for traversal in both directions. It can store and retrieve values and supports ++ and --, but not + or -
- A Forward iterator provides for one-directional traversal of a sequence. It can store and retrieve values and supports ++.
- An Input iterator can obtain values but not alter them. It can move in a forward direction only and support ++.

- An Output iterator can store values, but not obtain them. It can move in a forward direction only and support ++.

The categories of iterators provided by each STL container type are:

- vector<T>::iterator and deque<T>::iterator are random access iterator types
- list<T>::iterator is a bidirectional iterator type
- All the iterator types of the associative containers are bidirectional

2.3.3 Algorithms

The STL algorithms offer functionality not provided by the containers. Container member functions and iterators between them provide the fundamental means for accessing elements of containers. More powerful operations are implemented using standard library algorithms. These are template functions that act on containers through iterators. According to [SL1995], the STL provides many fundamental algorithms are in five categories:

Categories	Operations	Examples
Non-Mutating Sequence Algorithms	Inspect rather than modify the container elements on which they operate.	count, search,min
Mutating Sequence Algorithms	Modify the container elements on which they operate.	copy, reverse, swap
Sorting Algorithms	Modify the container elements on which they operate by rearranging their positions based on sorting criteria	sort, partial_sort
Sorting related Algorithms	Operations on sorted ranges such as set and heap operations	merge, set_union, make_heap
Generalized numeric Algorithms	Modify containers that are comprised of data elements which can be modified in the same way as numbers.	accumulate, inner_product, partial_sum

Table 2.2 The STL fundamental algorithms

Because algorithms are written to work on iterators rather than components, the software development effort is drastically reduced. For example, an algorithm that uses InputIterators can be used with any container that works with InputIterators or any of the

iterator categories above it in the iterator hierarchy. The pointer-like semantics of STL algorithms guarantee that there is an efficient implementation for them, often resulting in code that is nearly as efficient as hand-written assembly code.

2.3.4 Allocators

Any package provided by the Standard C++ Library must be portable to many different machine architectures. Portability is one of the strengths of C++ and the standards process helps to ensure that all compiler vendors have a common base to address this need.

One of the main problems to address in portability is the memory model of the machine. The memory model contains information about pointer types, the type of the difference between two pointers, the size of objects and also which primitives are used to allocate and deallocate raw memory. The STL encapsulates this information in a special class called an allocator. Allocators separate the STL from the dependencies of the underlying memory model of the machine architecture.

Each container is given an allocator when it is constructed. Whenever a container inserts or removes an element, it uses its allocator to allocate and deallocate the memory for the object. The container does not know anything about the memory model of computers, so it relies on the allocator for all of its memory needs.

2.3.5 Adaptors

Adaptors are template classes that provide an existing class with a new interface (interface mapping [SL1995]). Adaptors can be used to create new interfaces for containers or iterators. The STL provides container adaptors, some which change the behavior of an underlying container, some which allow a non-STL data structure to be used in STL algorithms.

Container Adaptors

Often it is desirable to create a specialized container from an existing, more general container. The operations of the new container are implemented using the underlying operations of the existing containers. Container adaptors are used to create a new container by mapping the interface of an existing container to that of the new container. This allows new containers to be created without much additional effort, since most of the functionality is already provided by the existing containers.

The STL provides three container adaptors:

- stack<Container>;
- queue<Container>;
- priority_queue<Container> ;

iterator adaptors

Adaptors can also be used to extend the functionality of an existing iterator. The STL provides three iterator adaptors:

- Reverse iterators. By applying the reverse_iterator adaptor to either a random access iterator or a bi-directional iterator an iterator is obtained that will traverse the same container in the reverse direction.
- Insert iterators. Iterators behave a lot like pointers. Sometimes, it is desirable to be able to insert a new element at that location in the container.
- Raw storage iterators. Raw storage iterators allow algorithms to use raw, uninitialized memory during their execution. The raw_storage_iterator is used by several internal algorithms for partitioning and merging elements in a container.

2.3.6 Functors

Functors [SL1995] (function objects) are classes that define operator( ). Using function objects rather than function pointers allows the STL to generate more efficient and more flexible code. They contain no data members or constructors/destructors (except the default ones provided by the compiler). Function objects can be passed to template functions in much the same way as pointers to functions are passed to C functions. However, because there is no pointer indirection and because they can often be expanded inline, they are much more efficient.

Suppose we wish to remove all of the ints that are a multiple of 3 from our list. We can use the STL remove_if function and supply our own predicate, called three_mult, as a function object. The actual checking of the multiple goes in the body of *operator(),* which returns a bool:

```
struct three_mult {
    bool operator() (int& v) { return (v % 3 == 0);}
  };
```

A new list, *m*, is created as the predicate is applied to each element of the list by remove_copy_if:

remove_copy_if (l.begin(), l.end(), back_inserter(m), three_mult());

The STL provides many function objects that are used by STL algorithms, but may also be used by users in their code. The STL provides function objects for arithmetic operators (e.g., times, divides, and modulus), comparison operators (e.g., greater, less, and less_equal), and logical operators (e.g., logical_and, logical_or, and logical_not).

2.4 Why the C++ STL Style?

We adopt the STL style to design and implement B+-tree index because the STL supports good programming practices and addresses several problems with previous C++ container libraries in a new and innovative way. There are a number of advantages to using the STL:

- "Standard" and "template"
- Reuse
- Smaller codes
- Flexibility
- Efficiency

2.4.1 "Standard" and "Template"

The STL is not just another C++ library among many libraries available today; it is a rather important addition to the C++ programming community. One reason is in the word "Standard". Another technical reason is in the word "Template".

The STL is made up of "standard components". Each of them has a clear standard interface and a well-defined functionality. This makes all the components easy to understand and to reuse. Also new components may be added with the same look as standard ones. Because STL has a completely modular system design, the programmers modifying the system focus on the modules to be changed without the need to understand the details of the whole system. The STL is using the template mechanism of C++ and extends it to new dimensions by using interoperable components concept.

Programming with "templates" is a compiler-supported mechanism to take generic data structures, such as arrays and lists, and generic algorithms, such as sort and binary search, and write them so we can reuse them without retyping them. With the STL's tools, we can solve complex programming tasks more quickly, using less code than with other

methods. Our programs will likely run faster with the STL components than if we had written our own version because the algorithms in the STL are efficient.

2.4.2 Reuse

The STL supports the generic programming paradigm, whose goal is to design algorithms so they are fundamentally independent from the types they act upon. The STL provides reusable components to achieve code reuse based on templates, rather than class inheritance. A large number of components already exist with a complete implementation on hand. This dramatically reduces the time needed for the implementation for many large systems where a great percentage of the code is simply imported from the STL.

The STL [Nel1995] represents a significant step forward in the development of C++ because it gives C++ programmers a framework of carefully designed generic data structures and algorithms that finally bring to C++ the long-promised dream of reusable software components.

2.4.3 Smaller Codes.

The STL is easy-to-learn structure because the library is quite small owing to the high degree of generality. Some have claimed that template-based containers cause code bloat, but template-based containers will actually make codes smaller because we will not have to have an entirely different set of code for different types of lists. If we are still concerned about code bloat, use the STL containers to store pointers instead of object copies.

There are approximately 50 different algorithms in the STL, and about a dozen major data structures. The separation of algorithms and data structures has the effect of reducing the size of source code, and decreasing some of the risk that the similar activities will have dissimilar interfaces. Without this separation, each of the algorithms would have to be reimplemented in each of the different data structures, requiring several hundred more member functions than are found in the present scheme.

2.4.4 Flexibility.

The use of generic algorithms allows algorithms to be applied to many different structures. Furthermore, the STL's generic algorithms work on native C++ data structures such as strings and arrays.

All containers follow the same conventions, so if we decide that maybe a deque will provide better performance than a list, the container can easily be changed. If we use typedefs, it can be as easy as changing one line of code. We also get the advantage of dozens of predefined algorithms for searching and sorting that work for any STL container.

The STL framework has a flexible design by adopting a complete component replacement policy. No component is made mandatory to the design. In other words all the components that make up the system are replaceable. This gives the designer the freedom to isolate and replace any one or more of these components with no- or minimal impact on the system. The design can be adapted to the needs of any application by simply changing some of the components. This was made easy since the building components are highly decoupled.

The STL allows for new components to be written and put into work with the existing ones, thus emphasizing flexibility and extendibility. This makes the STL particularly adaptive to different programming contexts including algorithms, data structures, and data types.

2.4.5 Efficiency.

The STL is efficient because "Much effort has been spent to verify that every template component in the library has a generic implementation that performs within a few percentage points of the efficiency of the corresponding hand coded routine" as described by Alexander Stepanov and Meng Lee in the STL specification. STL containers are very close to the efficiency of hand-coded, type-specific containers. The STL has been already written, debugged, and tested. No guarantees, but better than starting from scratch.

All the STL building blocks used are written with the most efficient implementation possible, therefore allowing the system itself to be efficient. The efficiency, as a fundamental issue in any new system, is addressed as a design goal of the STL.

The STL in particular, and the Standard C++ Library in general, provide a low-level, "nuts and bolts" approach to developing C++ applications. This low-level approach can be useful when specific programs require an emphasis on efficient coding and speed of execution.

Therefore, The STL is a standard library of high quality and efficiency with great emphasis on code reuse. Furthermore, The STL allows for new components to be written and put into work with the existing ones, thus emphasizing flexibility and extendibility. This makes the STL particularly adaptive to different programming contexts including algorithms, data structures, and data types.

2.5 Why Design Patterns?

A pattern is a way of doing something, or a way of pursuing an intent. This idea applies to cooking, making fireworks, developing software, and to any other craft. In any craft that is mature or that is starting to mature, you can find common, effective methods for achieving aims and solving problems in various contexts. The community of people who practice a craft usually invent jargon that helps them talk about their craft. This jargon often refers to patterns, or standardized ways of achieving certain aims. Writers document these patterns, helping to standardize the jargon. Writers also ensure that the accumulated wisdom of a craft is available to future generations of practitioners.

As Christopher Alexander describes, "Each pattern describes a problem which occurs over and over again ... 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 doing it the same way twice."

"Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [GOF1994]

The purposes of design pattern are mainly to reuse solutions and establish common terminology. Patterns are an attempt to describe successful solutions to common software problems by experts in software architecture and design. By reusing already established designs, people can get a head start on their own problems and avoid gotchas. Not only do patterns teach useful techniques, they help people communicate better, and they help people reason about what they do and why. Communication and teamwork require a

common base of vocabulary and a common viewpoint of the problem. Design patterns provide a common point of reference during the analysis and design phase of a project.

We simply introduce to the following design patterns associated with our design and implementation of the B+-tree index.

2.5.1 Casting Method

The intent of Casting-method pattern [Mey1992] is to represent an operation to dynamically and quickly obtain a type-safe reference to a subclass in an inheritance hierarchy

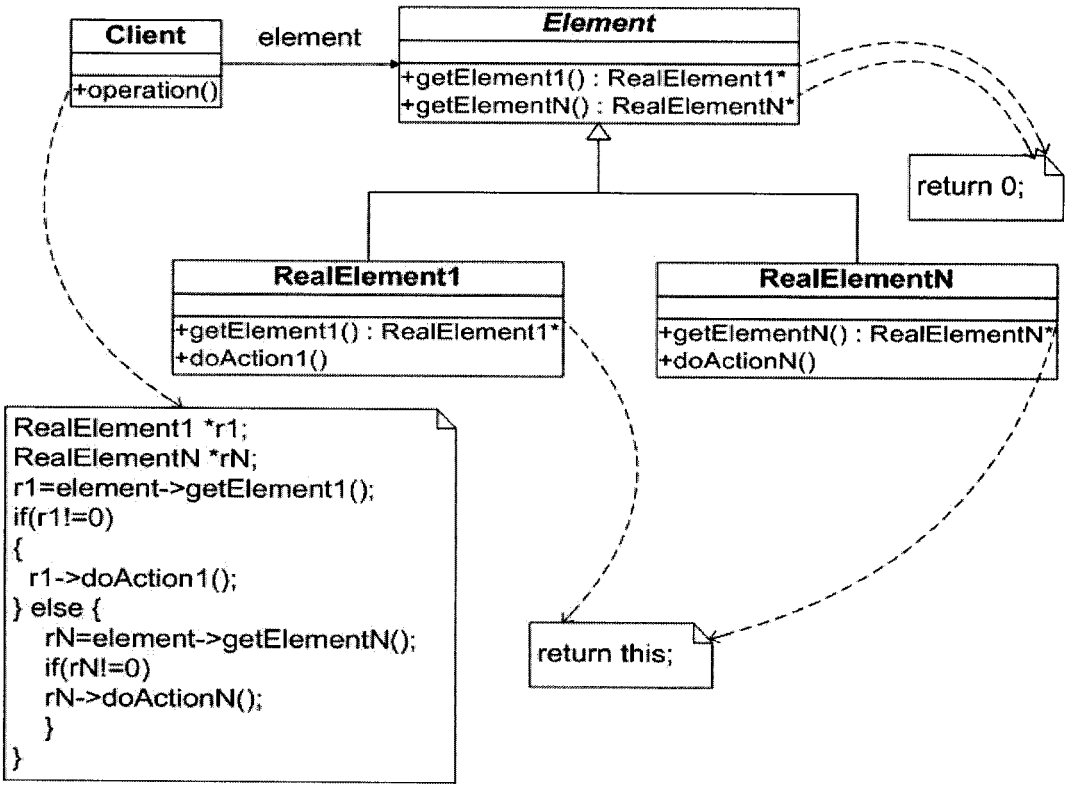

Figure 2.12 Casting method design pattern

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 downcasted

class reference from a base class and when real-time constraints require the faster and safest solution possible.

The Casting method pattern is rather simple and easy to understand. Furthermore, it has faster execution than any other technique except static cast. However, it is difficult to add new elements because every time a new element is added, the base class must be modified to add a new GetElement...() method and all users of any classes derived from the base class must be recompiled. Therefore, this pattern has better applicability for the cases that the number of subclasses is small and stable.

2.5.2 Composite Design Pattern

The intent of the composite pattern [GOF1994] in Figure 2.13 is to compose objects into tree structures to represent whole-part hierarchies in a manner that lets clients treat objects and compositions uniformly.

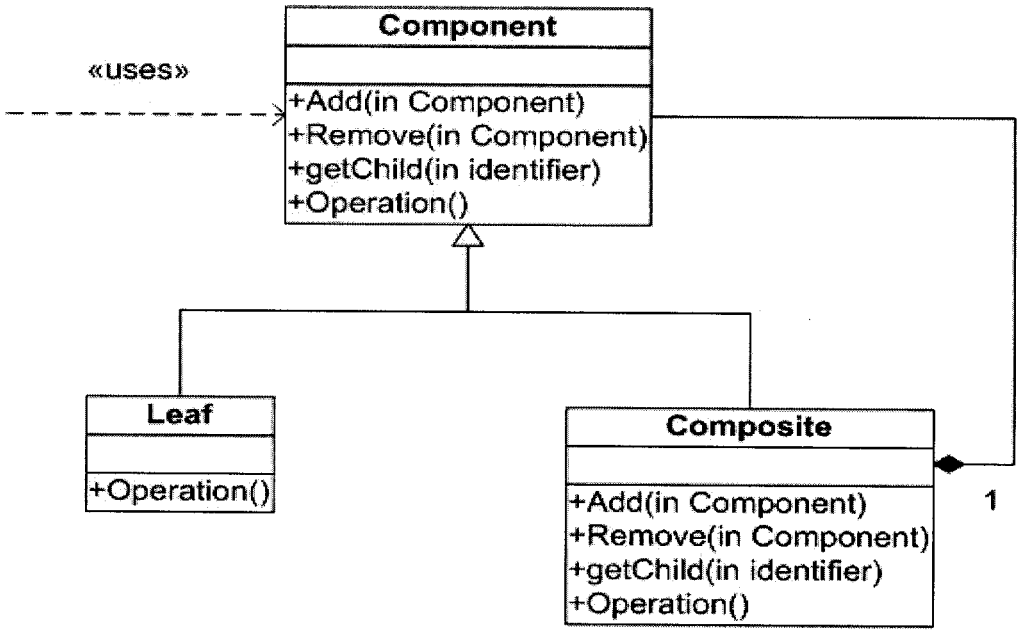

FIGURE 2.13 COMPOSITE DESIGN PATTERN

Implementing the Composite pattern is easy. Create an abstract Component class to define the interface of all tree objects and implements default common behavior. A

concrete Composite class implements behavior for storing and accessing child Component objects.

The Composite Design Pattern enables the Client to treat composite structures as well as leaf structures as if they are the same thing. As a consequence this simplifies the Client and makes changes or the additions to the component very simple. The Composite pattern makes it easier to add new types of components, newly defined Composite of Leaf subclasses will work automatically with current structures.

The composite Design Pattern may violate Liskov Substitution Rule due to possible client calls to the Add, Remove and GetChild methods for a Leaf object. If a client calls these functions in a leaf object, an error may result. Thus the client might need to implement special processing.

2.5.3 Proxy

The intent of the Proxy design pattern [GOF1994] in Figure 2.14 is to provide a surrogate or placeholder to control access to an object as shown in Figure 2.15.

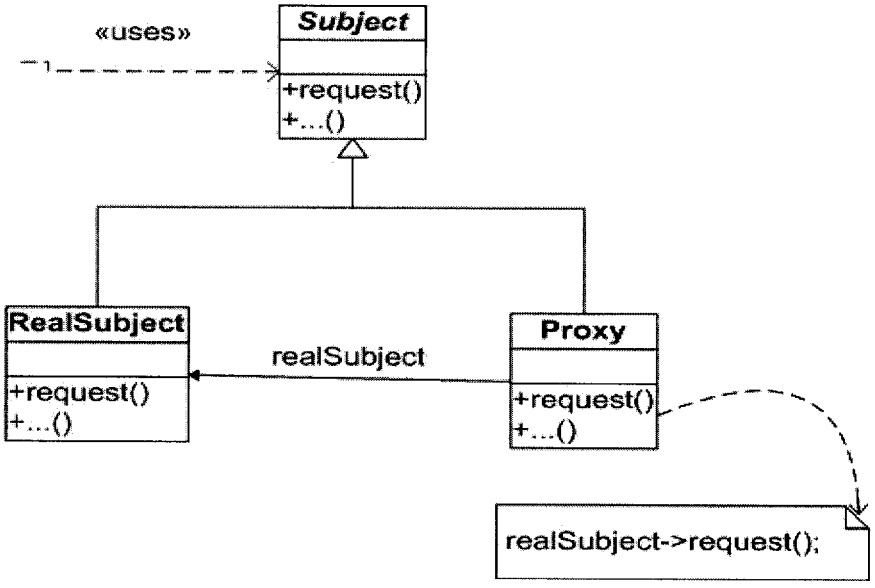

Figure 2.14 Proxy class diagram

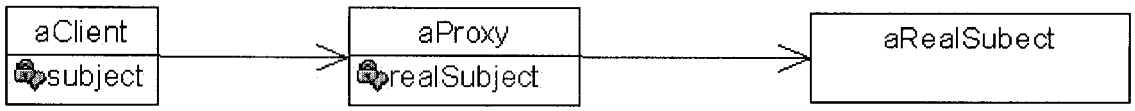

Figure 2.15 Proxy object diagram

In the Proxy design pattern, realSubject and Proxy implement the same interface, so that Proxy can handle some or all requests to the realSubject. Proxies provide a level of indirection to specific properties of objects, so they can restrict, enhance or alter these properties. Proxy is applicable whenever there is a need for a versatile or sophisticated reference to an object than a simple pointer. A proxy can be used in many ways:

- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand.
- A protection proxy controls access to the original object.
- A smart reference (smart pointer) replaces an ordinary pointer and performs additional actions when an object is accessed, e.g.: reference counting, loading a persistent object into memory when it's first referenced, and ensuring mutual exclusion

Smart pointers [Alex2001] are objects that look and feel like pointers, but are smarter. It is an application of the Proxy design pattern.

To look and feel like pointers, smart pointers need to have the same interface that pointers do: they need to support pointer operations like dereferencing (operator *) and indirection (operator ->). To be smarter than regular pointers, smart pointers need to do things that regular pointers do not. Probably the most common bugs in C++ (and C) are related to pointers and memory management: dangling pointers, memory leaks, allocation failures, locking and others.

The simplest example of a smart pointer is auto_ptr, which is included in the standard C++ library. We can find it in the header<memory>. Here is part of auto_ptr's implementation, to illustrate what it does:

```
template <class T> class auto_ptr
{
        T* ptr;
        public:
        explicit auto_ptr(T* p = 0) : ptr(p) {}
        ~auto_ptr()                 {delete ptr;}
        T& operator*()              {return *ptr;}
        T* operator->()             {return ptr;}
        // ...
};
```

Figure 2.16 Simple smart pointer: auto_ptr

As we can see in Figure 2.16, auto_ptr is a simple wrapper around a regular pointer. It forwards all meaningful operations to this pointer (dereferencing and indirection). Its smartness rests in the destructor: the destructor takes care of deleting the pointer.

2.5.4 Singleton

The intent of the Singleton design pattern [Alex2001] is to ensure a class has only one instance and provide a global point of access.

The Singleton class hides the operation that creates the instance behind a static member function. This member function, traditionally called *Instance(),* returns a pointer to the sole instance. Clients access the singleton by calling the static instance function to get a reference to the single instance and then using it to call other methods.

2.6 Chameleon technique

Chameleon is a technique for building generic, but unparameterized, classes, which are able to store arbitrary typed objects and still maintain type safety. A chameleon class [Sim2000] is a kind of new wrapper class called Value. An arbitrary variable *v* of type T can be assigned to an instance of Value, and thereafter the Value object itself can be assigned to any instance of type T, just as if 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 exception.

The class Value itself is defined as follows:

```
class Value {
private:
 enum Action { SET, GET };
 template <class T> T& value(T t = T(), Action action = GET)
  throw (Incompatible_Type_Exception&);
public:
 Value() {}  // Default constructor
 template <class T> Value(const T&) { value(t, SET); }  // Generic constructor
 template <class T> operator T() const throw (Incompatible_Type_Exception&) {
  return const_cast<Value*>(this)->template value<T>();// const_cast is safe
 }
 template <class T> T& operator=(const T &t) { return value(t, SET); }
};
```

Figure 2.17 interface of Value class

Value itself, as shown in Figure 2.17, is not a template class, but it heavily uses parameterization. All of its methods, including the constructor, are template functions. **value()** seems to contain all the magic of Value class:

```
template <class T>
T& Value::value(T t, Action action) throw (Incompatible_Type_Exception&) {
 static map<Value*, T > values;
 switch(action) {
  case SET :  {
   values[this] = t;
   return t;
  }
  case GET : {
   if (values.count(this)) return values[this];
   else throw Incompatible_Type_Exception(typeid(T).name());
  }
 }
};
```

Figure 2.18 Function value()

The function *value()* in Figure 2.18 contains the static local variable values, which is of type map<Value*, T> . Since this function is instantiated by the compiler once for every data type T we use in conjunction with Value objects, there will be a separate variable called values for each type T. Each Value object consumes one slot in the map corresponding to its actual internal type. Because these maps are static, they will be created before the start of our program, and because read/write operations to maps are guaranteed only to need logarithmic time, we can at least expect logarithmic time for our Value class, too

Chapter 3 B+-tree Index Design

The B+-tree index design is mainly based on ASHRAF GAFFAR's thesis titled "Design of A Framework for Database Indexes"[Gaf2001], but we address some problems of his design and develop it further. Finally we implement our design to follow the C++ STL style.

3.1 Use Cases

We will consider a B+-tree index as a part of the database with the following properties:

- Built as a balanced tree structure
- Be customized to suit different applications
- Be used to lookup data in the database (read-only access)
- Be used to insert or delete data in the database (read-write access)

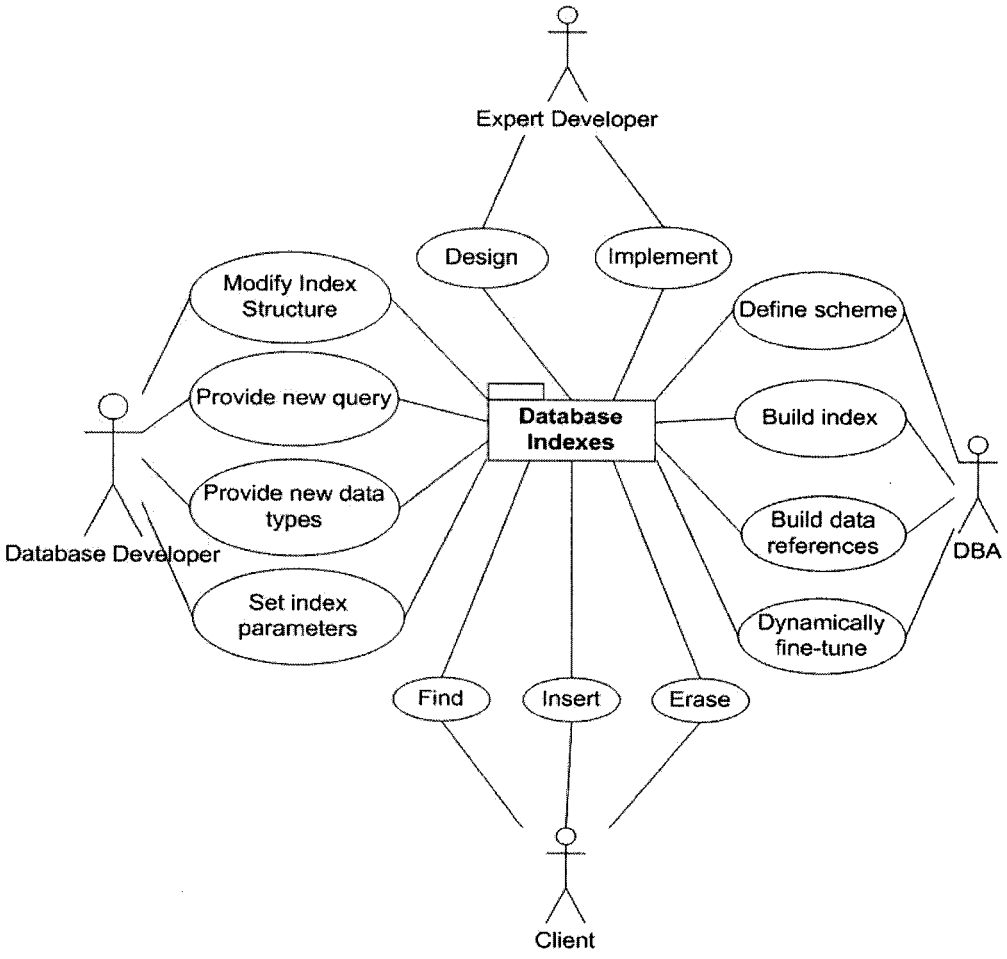

Figure 3.1 Detailed Use Cases

Use cases [BCC+2002] capture the functionality of a system. A use case defines a goal-oriented set of interactions between external actors and the system under consideration. Figure 3.1 shows actors do *what* (interaction) with the system, for what *purpose* (goal), without dealing with system internals.

3.1.1 Expert Developer

The expert developer is an experienced programmer who designs the database index system to satisfy domain experts needs and then implements the design using a programming language.

The design for a database index system needs to meet not only the functional need of database domain experts but also the nonfunctional needs like storage, retrieval issues, and platform mounting.

Implementation of the design includes all details such as the physical access details and the platform-dependent details.

3.1.2 Database Developer

The Database developer is a knowledge domain expert, responsible for customizing an existing database system by modifying some of its parts (components) or by replacing some parts with others to be able to support a new data or query type, or support a completely new index structure with new access method.

The Database developer is also responsible for setting the index parameters such as the tree order, the page minimum fill factor, and the page size, to suit the system platform and the application variables.

The database developer defines a new data type by writing an index implementation to produce a new index capable of handling the data type used by a new application. This includes defining suitable keys to describe the data partitions, and defining ways to compare them, setting a suitable layout for the pages and a page policy for adding or deleting this new data type keys inside them.

The database developer defines a new method to traverse the index by replacing some parts of an existing index in order to produce a new index that is using a new access method

3.1.3 Database Administrator

The Database administrator (DBA) is the person responsible for building the Database system, which includes determining the scheme, the physical tables and the indexes used to access them.

The DBA will build the index by insertions of the pairs of key, data reference (data's physical location) into an empty index file (bulk loading the data reference file). A successful bulk loading operation will yield a complete index to the data.

After the database system is up and running, the DBA needs to dynamically fine-tune it by adjusting its parameters to achieve the optimal performance under typical workloads.

3.1.4 Client

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

The application will connect to the database index and search for some data. An index will be used to lookup data. The index will return the matching data in the form of references to their physical locations. The references will then be used to access the data.

For an insertion, the search use case runs first to find a suitable insertion position. The position is then used to insert the element, and the index structure is adjusted if necessary to reflect the changes to the element.

For a delete use case, the search use case runs first to locate the matching element to be deleted. Then the candidate data is deleted. The index structure is adjusted if necessary.

3.2 Relation between Index and Database data

Each index is built from the physical data indirectly, through a data reference file called index file. The index file is composed of key/data reference pairs, and constitutes the data level of each index (see Figure 3.2). This separation between physical data and data references allows for building multiple indexes on the same data set (data is not included in the index) and for the ability to change physical data formats without affecting the existing indexes.

The index does not directly provide us data, but information about where the data is. This means that the data stored in the index is different from the typical database information (tables, records, etc).

Depending on some system characteristics (like access time constraints, and system volatility) different binding mechanisms are applied. The mechanism used must guarantee that changes in the record physical locations will still allow for all associated indices to access them correctly.

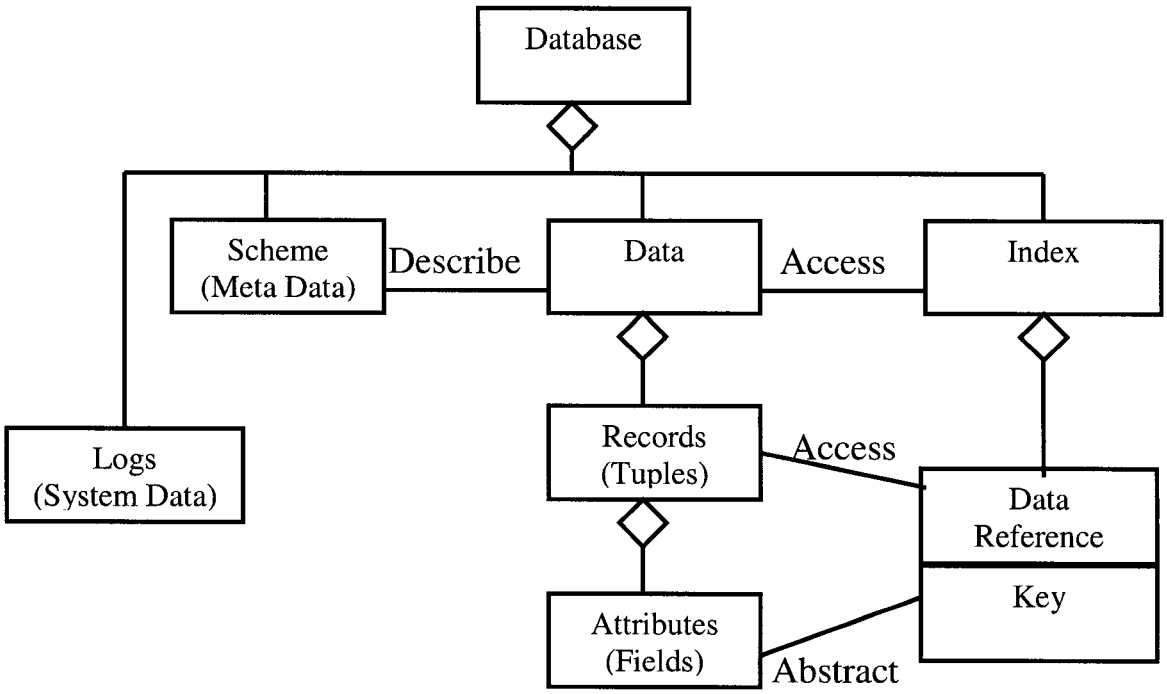

Figure 3.2 relation between the index data and the database data

From Figure3.2 we can see that index data references physical data. We start querying the index using a certain key. With some comparison criteria, we try to find our way through the index down to the index leaf level, where the index data is stored. This will bring the index responsibility to an end. We should then refer to the physical database access mechanism to get the real data we are looking for.

Implementing an index goes the opposite way. We start with physical database data that we want to build an index for, build the index file, and then build the index using the index file.

3.3 B+-tree Index Structure

3.3.1 Basic Components

The B+-tree index will be built using the STL components. These components will provide the necessary index structure, and manage the access and storage of the index.

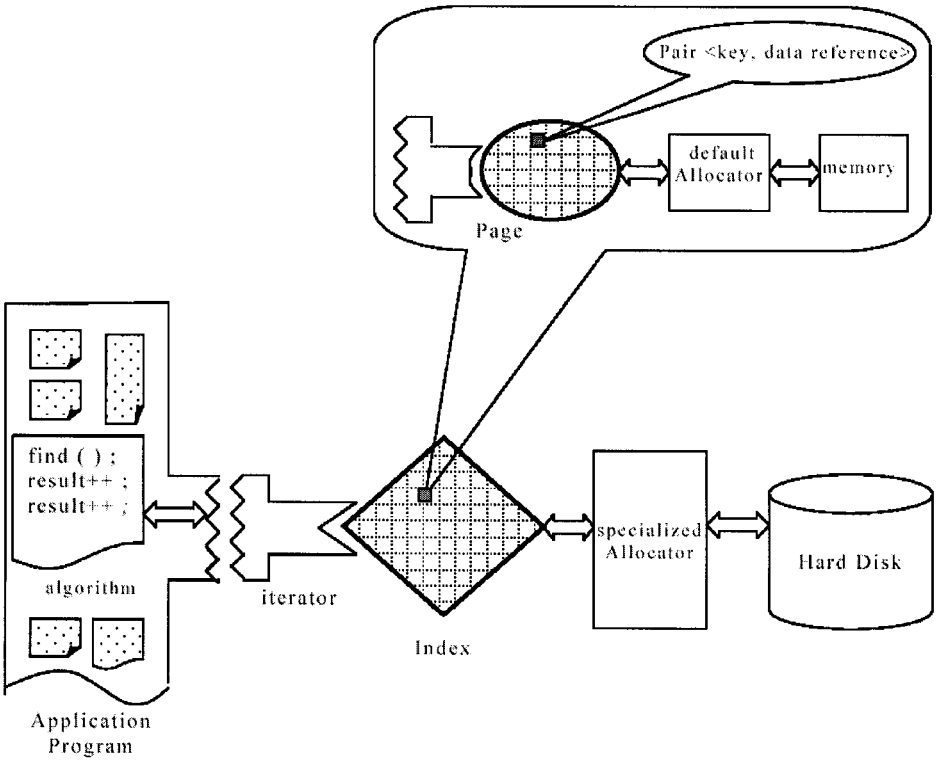

Figure 3.3 The components layout [Gaf2001]

In Gaffar's design as shown in Figure 3.3, the B+-tree Index container will be an index of pages, so the elements that the index stores are of type page. The index iterator is therefore iterating through pages, one page at a time. The Index container will use a specialized allocator that takes the responsibility of retrieving the page from the storage (like a hard disk or other mass storage media) into memory for access and controlling the different objects accessing the same page simultaneously. A default, in-memory allocator is provided for simple applications where the whole index can fit in memory at one time.

It is up to the system designer to use it or override it by providing a storage dependent specialized allocator.

However, there are several problems we found in his design. A B+-tree index contains index pages and leaf pages, but they are invisible for users. What users can operate on are not pages but pairs in pages. Therefore, the index iterator does not iterate over pages but pairs in leaf pages. In addition, we follow the STL to let allocators to take the responsibility of the management of memory, and use a Proxy to control access to pages from the storage like a hard disk or other mass media.

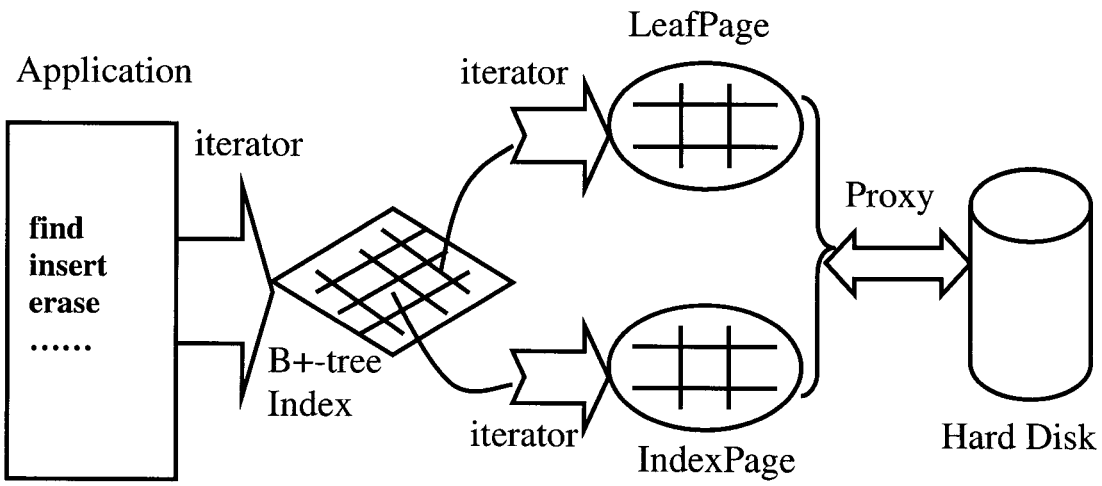

Figure 3.4 Basic components

The B+-tree index will be built on leaf and index page components shown in Figure 3.4. These components will provide the necessary index structure, and manage the access and storage of the index through a proxy mechanism.

The B+-tree index shown in Figure 3.5 is designed to be an associative container like multimap in the C++ STL. The index container will be composed of pairs (Key,data/data reference). Pairs are divided into interlinked subcontainers (leaf page containers). The Key type and data type are passed to the index as a template type. This makes the index work with any key and data type. The index provides an iterator to its contents (pairs). The only way to interact with the container is through its iterator, which provides controlled access to the elements of the container.

The index and leaf page are also containers on a smaller scale. Because the keys in them are ordered, index and leaf pages are associative containers. While a B+ index typically resides on hard disk, a page is small enough to fit in memory. Whenever a page is needed, it is retrieved from the hard disk into memory through a proxy. At this point the page can perform its tasks of searching for a key in its contents, accepting new entries, deleting some existing ones, etc. This design produces a simple system structure without affecting its complexity or extensibility.

The elements of a leaf page container shown in Figure 3.5 is a kind of pair which is entries of the form (Key, Data reference) where Key refers to a sorting key or field in the database, and Data Reference is the physical reference to a tuple in the tables.

An index page shown in Figure 3.5 is also a container that has entries of the form (separator, child-pointer) where a child-pointer is the address of a lower page and a separator provides information about the boundaries between the two pages in the sequence set of a B+-tree, so child-pointers have one more than separators. A separator may be a prefix from page key or an exact copy of the page key of the lower page that the child pointer points to. In our design, the page key of a leaf page is the first key but the page key of an index page is the page key of the leftmost leaf page if the child pointer is treated as the root of a subtree.

The B+-tree index will be built as a tree. To save disk access and storage overhead of disk, we restrict the number of elements in leaf and index page containers. However, the index is completely dynamic: inserts and deletes can be intermixed with searches, so inserting and erasing may cause pages to split and merge with their neighbor. The splitting and merging may result in reorganizing the tree.

Concurrency will be developed to control the different objects accessing the same page simultaneously. This will ensure data integrity by applying a suitable access and locking policy (pinning the page). After the pages have been modified, they also need to be updated in the physical storage (flushing the page) by the proxy

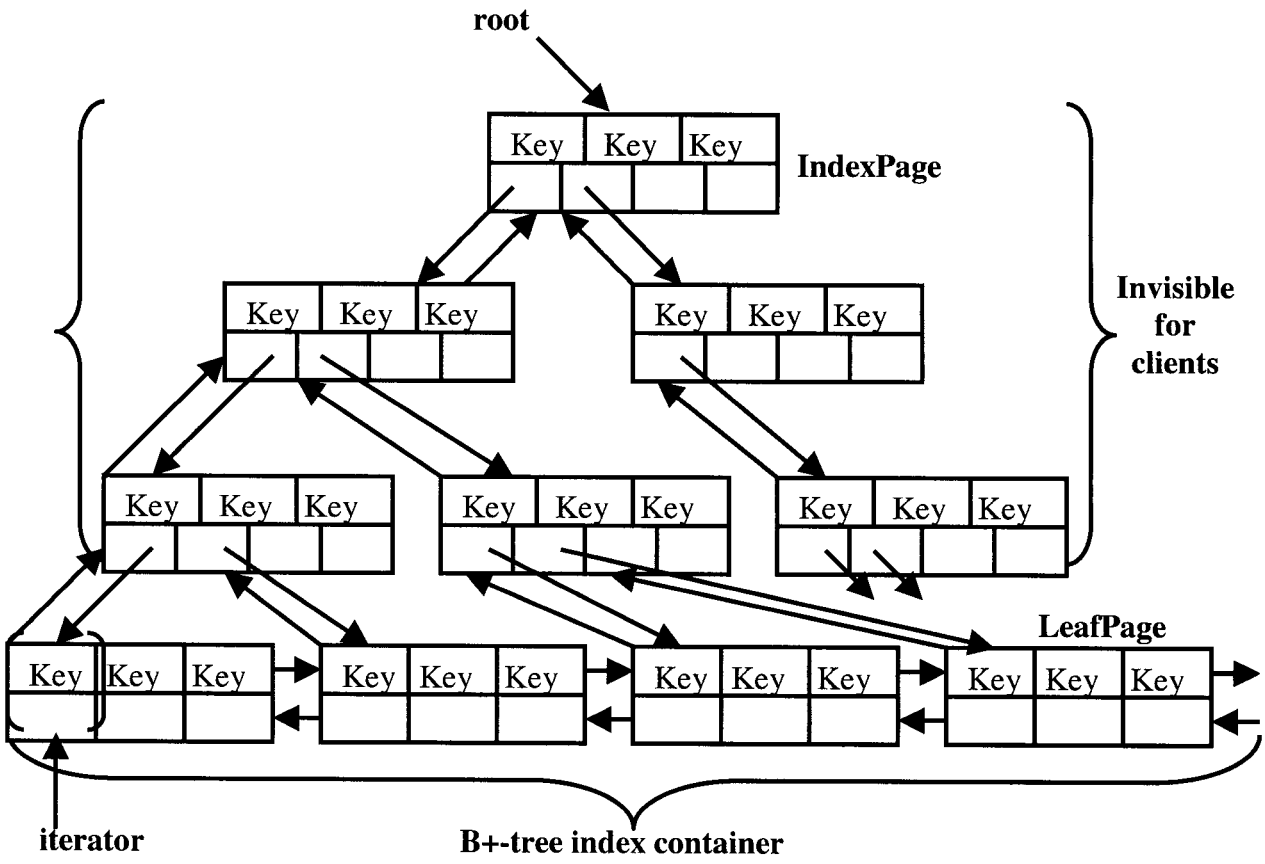

Figure 3.5 B+-tree index container

3.3.2 Class Diagram

Figure 3.6 shows a class diagram of the B+-tree index in our design. Because Bplustree, IndexPage, and LeafPage classes are designed to be STL-like containers, they must follow STL styles. To handle inhomogeneous collections of page objects and make our implementation safer and more elegant, a page class is added as a base class of IndexPage and LeafPage. In practical application the index exists permanently on non-volatile storage, like a hard disk or other mass storage media since it is normally too large to fit entirely in memory. This means that the container will use a proxy (smart pointer)

that takes the responsibility of retrieving the page from the storage into memory for access.

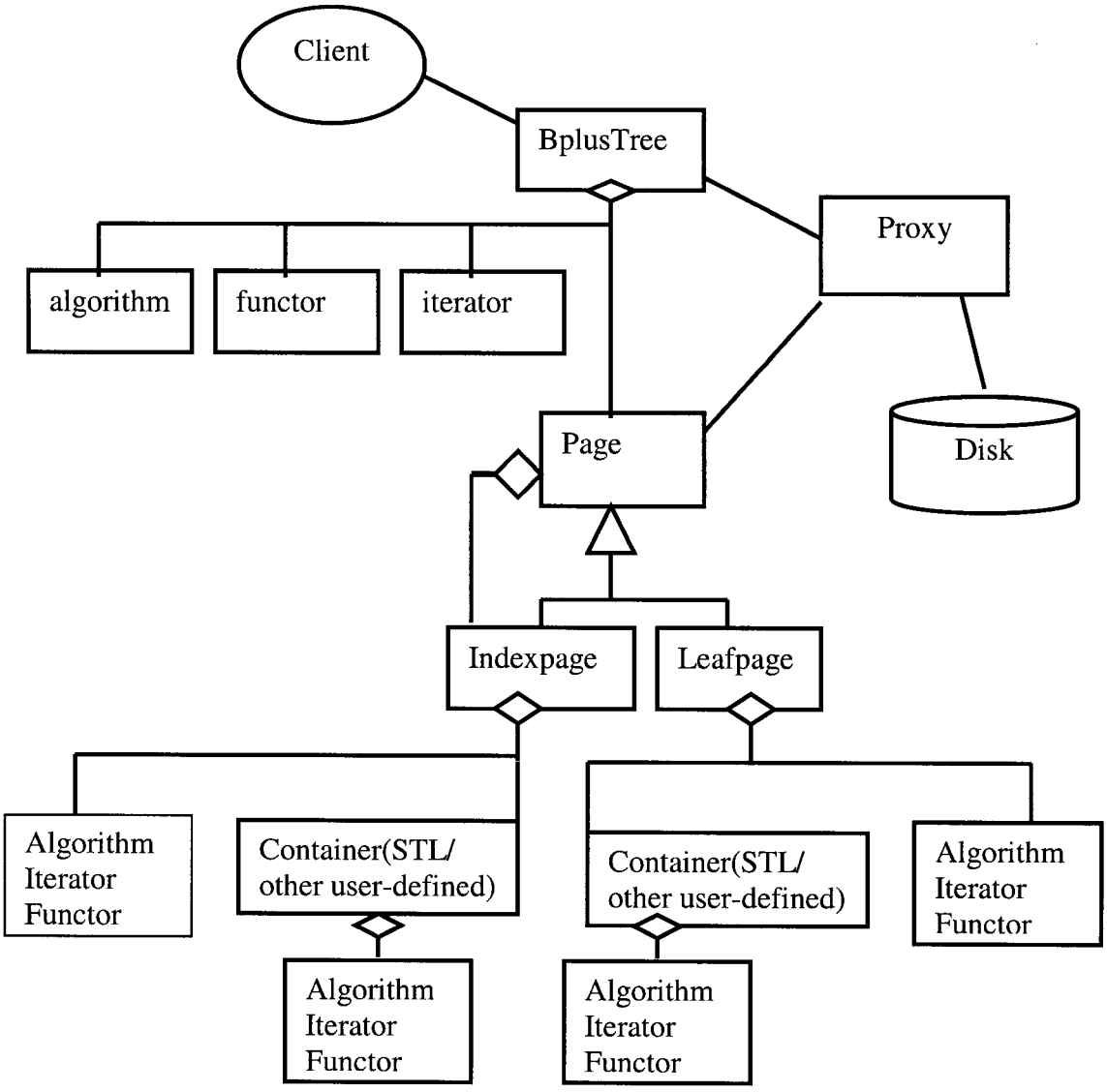

Figure 3.6 Main class diagram of the B+-tree Index

3.4 General Interfaces

The reason that the STL's components [Aus1999] are interoperable and extensible, and the reason that we can add new algorithms and new containers and can be confident that the new pieces and the old can be used together, is that all the STL components are written to conform to precisely specified requirements. An abstract concept can be thought of as a list of type requirements or a set of types. Defining abstract concepts and writing algorithms and data structures in terms of abstract concepts is the essence of generic programming.

It is important for BplusTree, Page, IndexPage, and Leafpage containers shown in Figure 3.7 to conform to a set of abstract concepts provided by the STL. Also, IndexPage and LeafPage should have some specific functions for special operations on them. In addition, these two classes are derived from a base class of Page, so they are needed to implement interfaces inherited from the base class.

To understand our design, it is necessary to recognize that inheritance, modeling and refinement are useful for different things. Inheritance is a relationship between two types, modeling is a relationship between a type and a set of types (a concept) and refinement is a relationship between two sets of types (concepts). They are completely separate relationships and any methodology that tries to make do with only one of them is incomplete. This is why we combine inheritance and generic programming in B+-tree index.

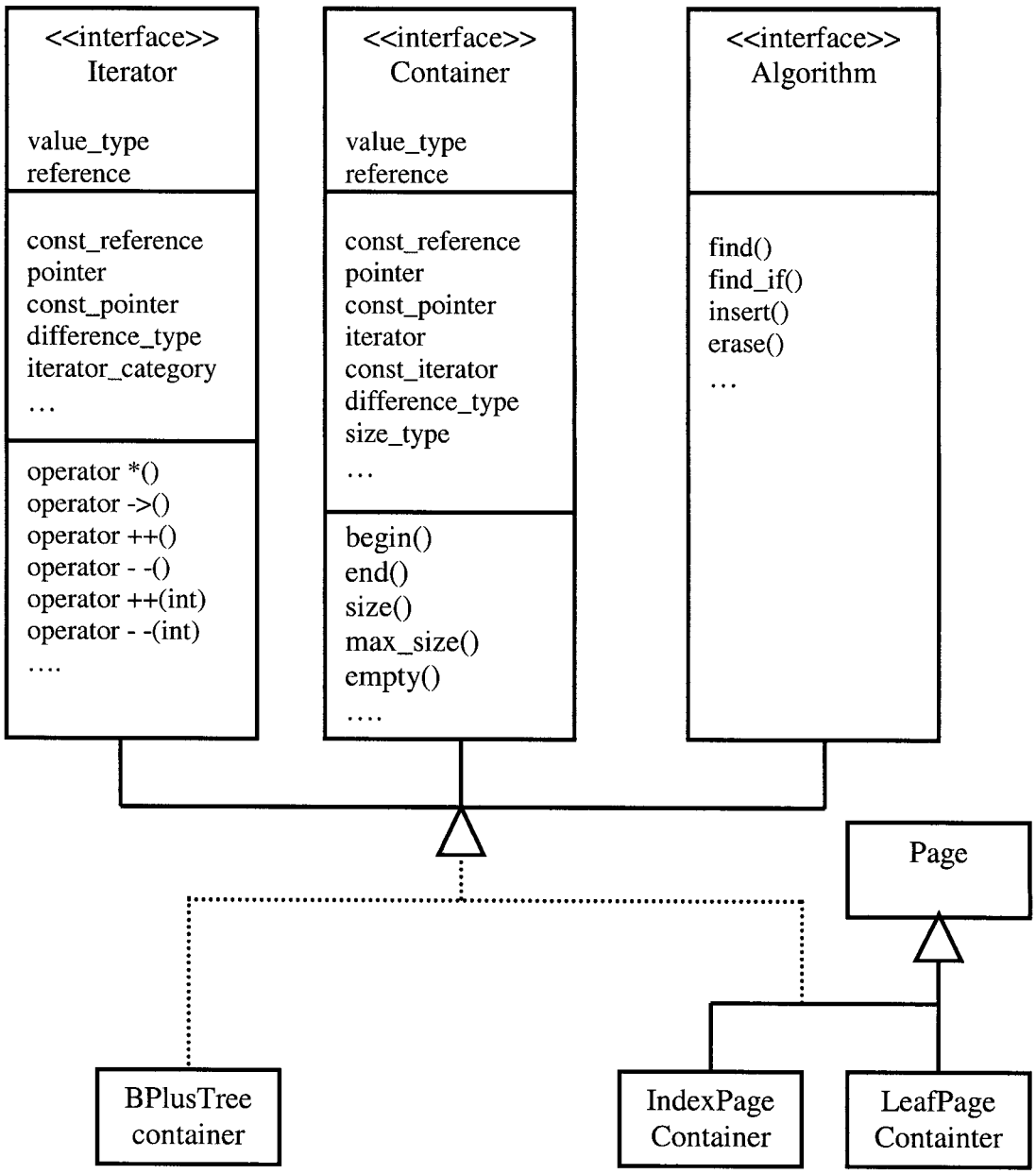

Figure 3.7 View of main interfaces

Chapter 4 B+-tree Index implementation

4.1 Issues Encountered in the Implementation

4.1.1 Static or Dynamic Polymorphism?

In principle, functions such as a STL-like approach could be implemented with dynamic polymorphism. In practice, however, it would be of limited use because the iterator concept is too lightweight compared with the virtual function call mechanism. Adding an interface layer based on virtual functions would most likely slow down our operations by an order of magnitude.

As a rule of thumb, we try to use a static model when we can, and rely on a dynamic model when needed. Therefore, sometimes we may combine both forms of inheritance in B+-tree index design.

4.1.2 Interface Realization or Implicit Container Inclusion?

The B+-tree, including index and leaf pages, is designed as a container that follows the STL style. However, containers can be implemented using one of two implementation criteria: Interface realization or Implicit container inclusion.

Interface realization is implemented by inheriting the public container interface from a STL container class and implementing the class methods completely. This will provide a more specialized code that is still conformant with the design. This is particularly useful when special time or space restrictions might apply.

For efficient implementation, the containers can be built on top of an existing STL container. They will work as a wrapper class that has a STL container class as one of its data members (an implicit container). This will allow for masking the unneeded member functions in the STL class, and will save a lot of implementation time. This will give a more generic code.

In our implementation of the B+-tree index, we always use implicit container inclusion to implement these STL-like containers.

4.1.3 No Virtual Template Function

Member function templates cannot be declared virtual in C++. This constraint is imposed because the usual implementation the virtual function call mechanism uses a fixed-size table with one entry per virtual function. However, the number of instantiations

of a member function template is not fixed until the entire program has been translated. Hence, supporting virtual member function templates would require support for a whole new kind of mechanism in C++ compilers and linkers.

In contrast, the ordinary members of class templates can be virtual because their number is fixed when a class is instantiated.

There are two common solutions: One is that if we know the number of possible data types, we need to write a specialized function for every type. The second solution is to use only a void* as data type, and cast the pointer according to the desired result. That means it may cause too many type casts. Moreover, we have no guarantee of the correctness of the cast.

In the B+-tree implementation, the Page class is designed to be a base class for Indexpage and LeafPage container classes. Although most of the functions in these two subclasses have the same conceptual interface, they have actually different signatures. We take an example of *insert(value_type&)* in both IndexPage and LeafPage classes. The parameter value_type& of insert operation has different types and meanings in the two classes. The type value_type is referred to as a pair of key and data reference in a leaf page, but is not the pair in an index page. Therefore, in the Page interface, we should add two functions like *insert(IndexPage::value_type)* and *insert(Leafpage::value_type).*

The disadvantage of this solution is the high amount of code that must be written for every derived class, even for types that class is not actually used.

We can use a chameleon technique referring to section in Background or casting-method design pattern to solve this problem.

4.2 The Solutions to B+-tree Implementation.

On the basis of our design, we had tried some solutions to implement the B+-tree index. These ways could work, but there would still be limitations about efficiency, extensibility and reusability of interfaces and compilers. From our experiments, we finally got a better way to implement the B+-tree index.

4.2.1 B+-tree Implemented by using Composite Pattern

According to the design of the B+-tree index, it is natural for us to use a composite design pattern to implement it. Figure 4.1 shows a class diagram for the B+-tree index using a composite design pattern. The Page class is an interface that declares common virtual functions. For every function in the Page interface, the IndexPage and LeafPage class implements it respectively in their own class.

The IndexPage class maintains a container of separators (keys) and page pointers. Typically, IndexPage functions are implemented by iterating over that container and invoking the appropriate function for each element in the container.

The LeafPage class is designed to be a container of pairs of key and data reference. The only way to interact with the container is through its iterator, which provides controlled access to the elements of the container.

This class diagram does illustrate one problem with the pattern: we must distinguish between leaf pages and index pages when a pointer points to a page, and we must invoke an IndexPage-specific or a LeafPage-specific function, such as *insert(value_type& v)*. We typically fulfill that requirement by adding a method, such as *isLeaf()*, to the Page class. That method returns false for IndexPage object and true for LeafPage object. Additionally, we must also cast the page pointer to an IndexPage or LeafPage pointer. Figure 4.2 shows sample codes how to cast a page pointer to a leaf page pointer.

```
...
if(page->isLeaf())
{
   LeafPage* lPage = static_cast<LeafPage*>(page);
   lPage->insert(v);
}
...
```

Figure 4.2 Sample codes to use type-casting

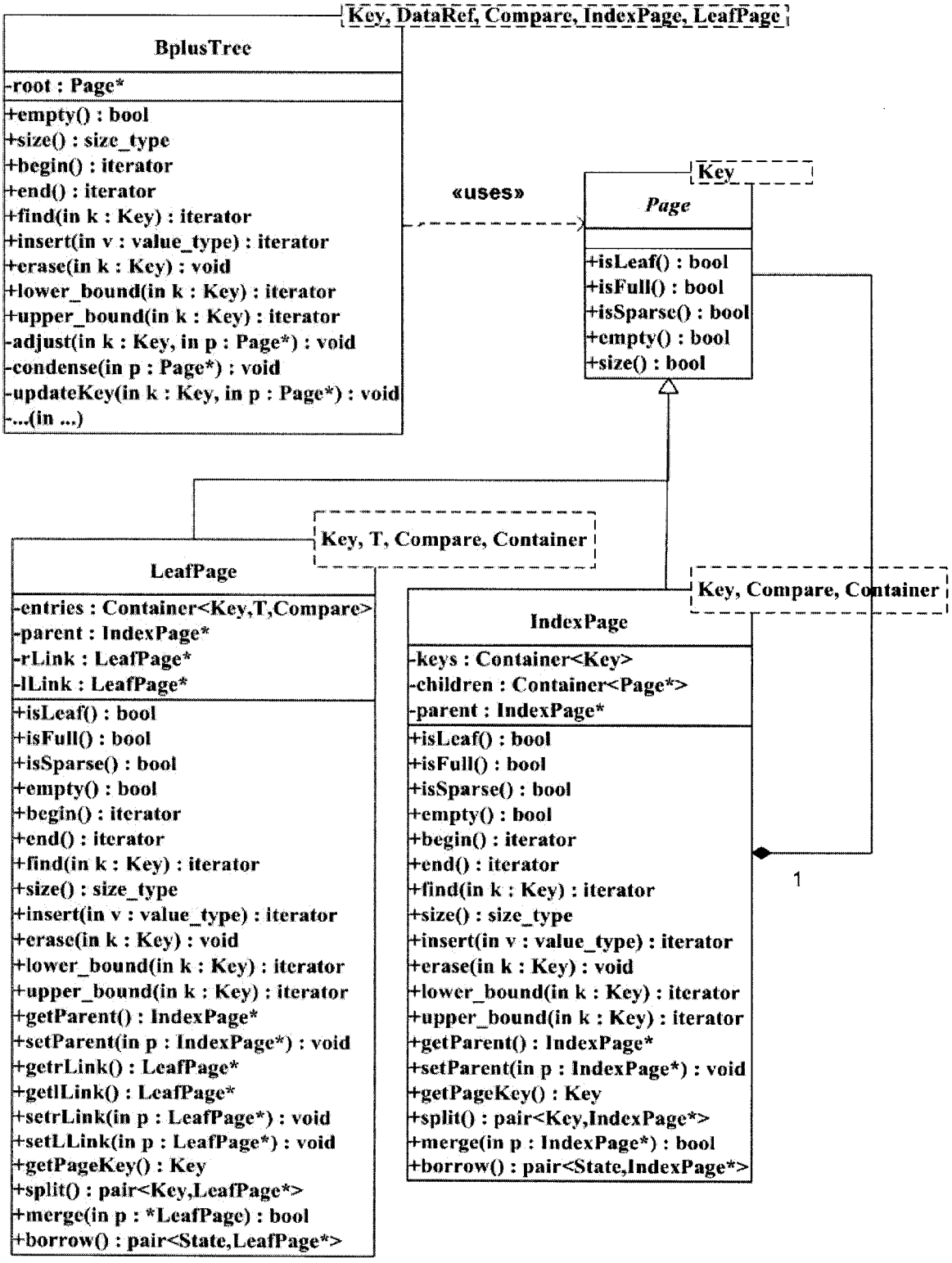

Figure 4.1 B+-tree class diagram using composite pattern

The implementation of the B+-tree by using type-casting is ugly, and results in a lot of useless code which makes the useful code hard to read. In addition, type-casting is not very safe and efficient.

4.2.2 Alternative B+-tree Implementation using Composite Pattern

If we want to avoid too much casting, another choice to implement the B+-tree using Composite design pattern is to put "everything" into the Page class. All the functions of its subclasses are declared in the Page interface. Thus it is not necessary to cast a page to an indexPage or LeafPage. However, this causes another problem: function overloading.

The functions with different return types but the same parameters cannot be overloaded in C++. Although iterators of the IndexPage and LeafPage class have the same conceptual name, they are completely different types essentially. An iterator in LeafPage points to a pair of Key and Data Deference, but an iterator in IndexPage should not. IndexPage and LeafPage classes are containers that follow the STL style, there must be a lot of functions such as IndexPage:: iterator begin() and LeafPage::iterator begin(), which are the same conceptual name but different meanings and types. If they are declared in the base class together, that would cause overloading problems. There are two ways to solve overloading ambiguity:

1. Changing function names to make them different. For example, begin() of IndexPage can be changed into index_begin(), and begin() of LeafPage can be like leaf_begin().
2. Using empty tag classes to solve overloading problems. We can define two tag classes like:

   struct Bplustree_indexPage_tag{};

   struct Bplustree_leafPage_tag{};

Because these two tag classes are empty, their cost is very little. But they help us to resolve the overloading problems in the declaration of the Page class. For example, IndexPage::begin() can be declared as begin(Bplustree_indexPage_tag) in the Page class, and LeafPage::begin() can be changed into begin(Bplustree_leafPage_tag).

Although this approach can be used to implement the B+ tree index, it breaks the STL concepts and it is not easy to understand and not convenient for a STL developer to use

them. This is mainly because IndexPage and LeafPage have different iterator types. If we can unify the iterators, the interface for Page class will become easier to use.

4.2.3 An Improved Way using Composite Pattern

Because IndexPage and LeafPage have small differences in their structure, they should be uniform if there are only small changes. In order to generalize these two pages, we have to change some of index page structures.

Generalize Index page and Leaf page

Notice that index pages of B+-trees have structures *<P1,K1, P2,K2, ..., Kn-1, Pn>*. In order to generalize the B+-tree with other tree structures, one more key, *Kn* is added at the end to pair a key and a pointer in each index node without affecting the original structure and property of a tree page. In this case, each page now has the structure *<P1,K1, P2,K2, ..., Kn-1, Pn, Kn>*. If the leaf keys are in ascending order from left to right, then each key *Ki* is greater than or equal to every key value in the subtree pointed by *Pi.* That is, every key is the biggest value of its subtree. If the leaf keys are in descending order from left to right, then each key *Ki* is the smallest value in its subtree.

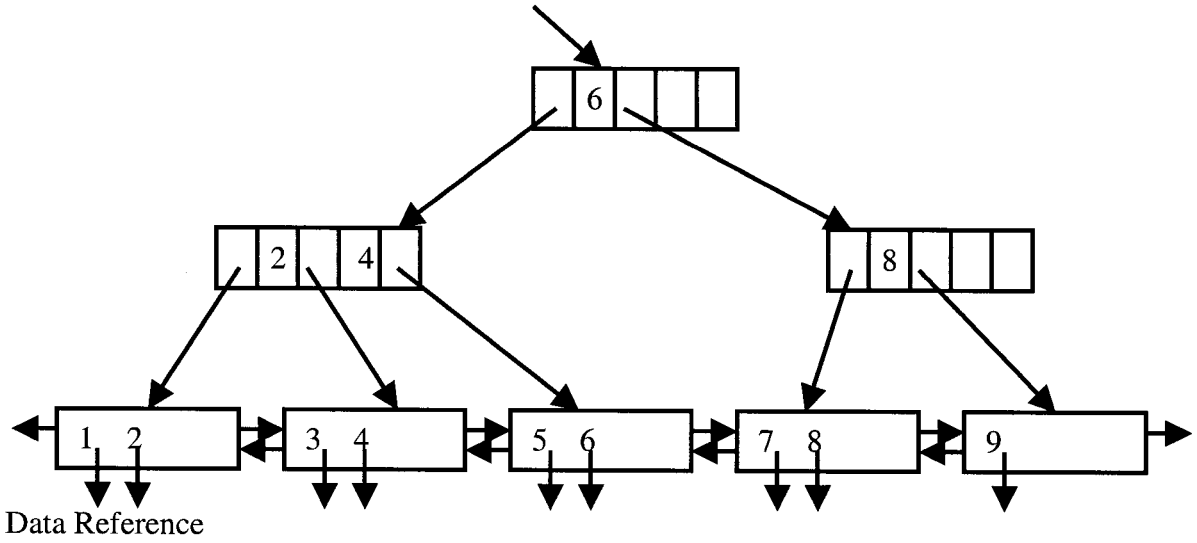

Figure 4.3 Simple B+-tree

Figure 4.3 shows a simple B+-tree with integer key values. The key values in the tree are from 1 to 9. This range is divided into two parts by the key in the root, 6. Every key value in the left subtree, including its left children and subtrees rooted at them, is less

than or equal to 6. Key values in the right subtree are greater than 6. The range in the left subtree, 1 to 6, is then subdivided by keys 2 and 4. The same happens in the right subtree. Key values in the left subtree of 2 are less than or equal to 2, while values in the left subtree of 4 are less than or equal to 4.

The simple B+-tree in Figure 4.3 is redrawn in Figure 4.4 with the modified index structure. Every index page is added the biggest key of its subtree. For example, 9 is added into the root page (an index page) because 9 is the biggest key of the tree.

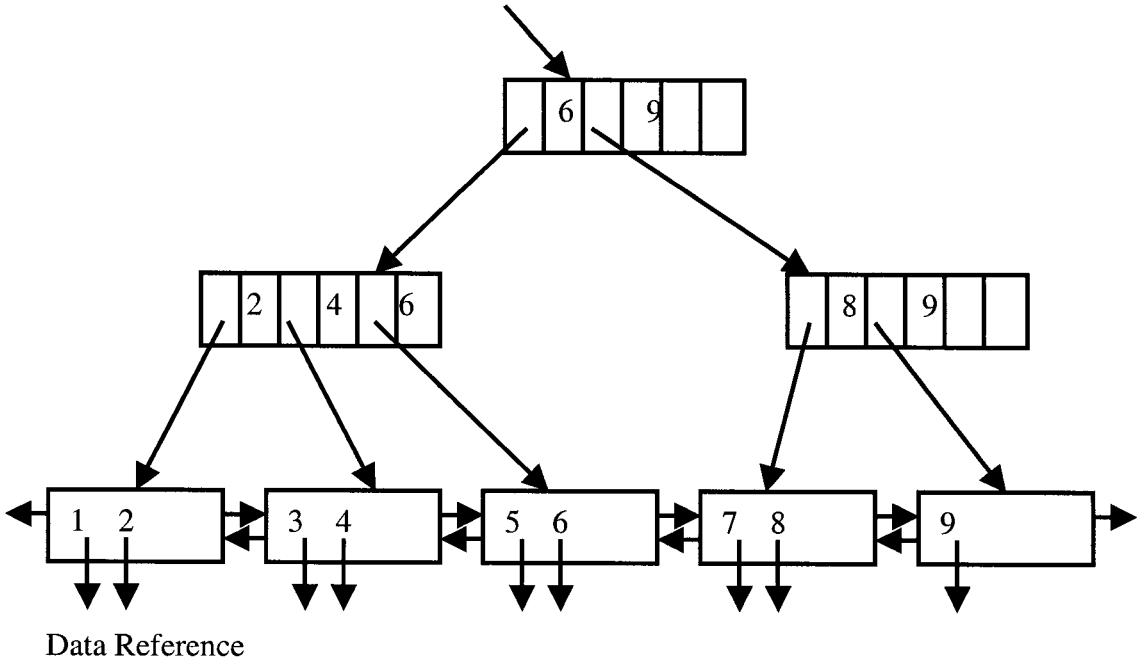

Figure 4.4 Pairing Key and Pointer for B+-tree

Through the generalization, index pages have the same number of keys as that of child pointers like leaf pages. Thus, an index page contains pairs of key and child pointer and a leaf page has pairs of key and data reference. Through a key position in an index page, we can get a child pointer responding to this key position. The same thing also happens to a leaf page. A data reference in a leaf page can be found by computing its key position. Therefore, we can use two implicit containers such as vectors to store keys and child pointers of an index page respectively. Keys and data references of a leaf page are also stored into two separate implicit containers.

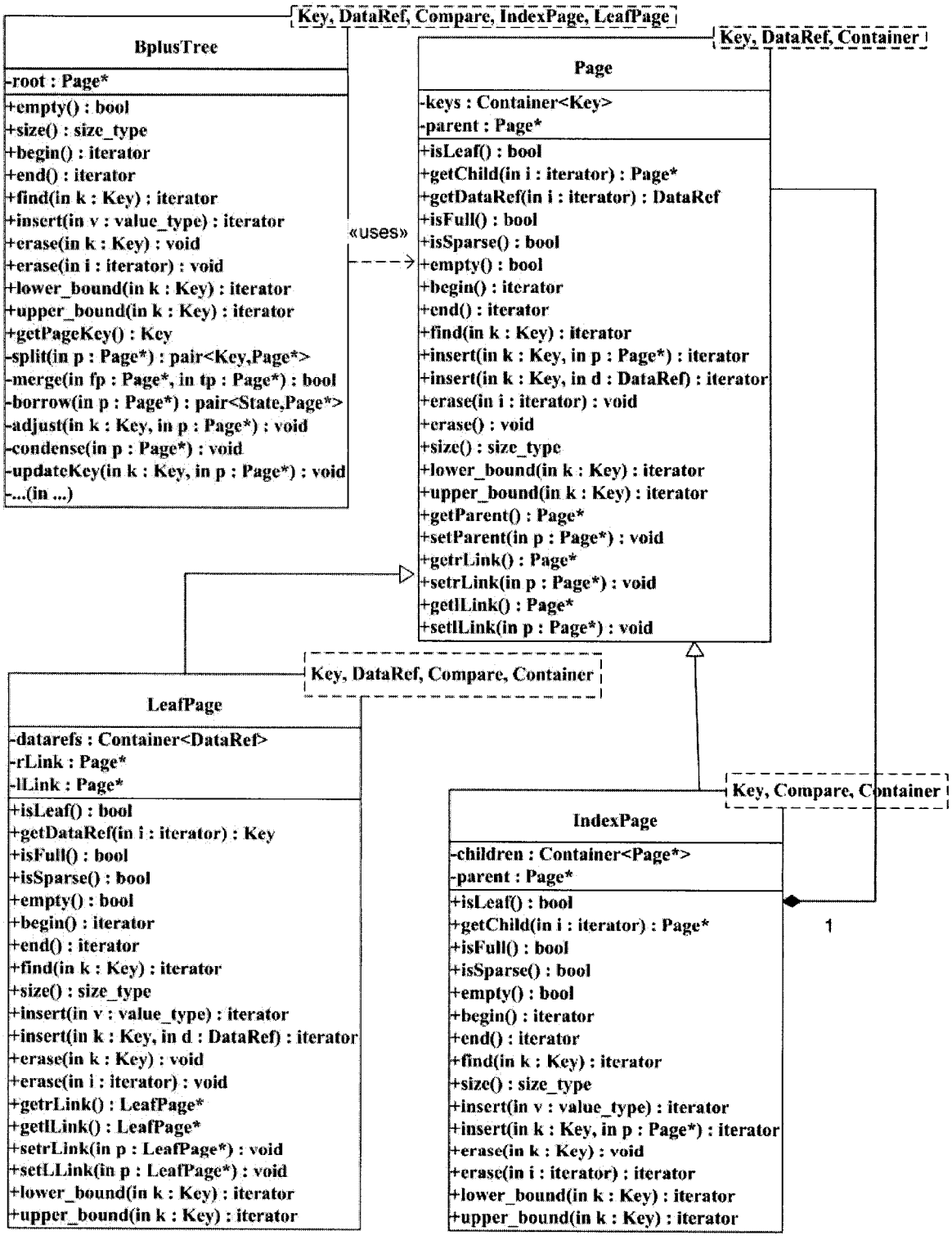

Figure 4.5 A improved way to implement B+ tree using Composite Pattern

In order to get the same iterator type, both index pages and leaf pages use the iterator provided by the containers to store keys as their external iterators. They have their own internal iterators to access their own elements. Using this way, we get the same external iterator type, so we can uniform the index page and leaf page into a common base class.

Figure 4.5 shows a class diagram for B+ tree index using Composite Pattern. Page is a base class that declares all the functions of IndexPage and LeafPage classes. Page, IndexPage, and LeafPage classes use the iterator provided by the container of keys as their external iterator, so they have the same type of iterators. This makes the interface of Page simpler and easier to use. This way reduces the number of functions declared in the base class, but all the functions are also needed to put in the Page class. All the changes in the subclasses will affect the base class. In addition, there are two special functions that must be declared in the base class and implemented in its subclasses respectively: getChild(iterator *i*) and getDataRef(iterator *i*). Given an iterator *i* in the key container, these two functions return the child pointer and the data reference responding to the position *i*.

In fact, there are many shortcomings in this way such as changing the original data structure of B+-tree, and changing the original meanings of iterators.

The ways to use Composite design pattern can work for the B+-tree implementation, but they do not follow the STL style completely or cause too much type-casting. Can we find a better way to conform to the STL concepts and avoid downcasting?

4.2.4 Using Chameleon Techniques to Uniform the Interface of Page

Another better way to implement the B+-tree is to unify index and leaf pages. Index pages and Leaf pages have different types of elements and virtual functions that cannot be parameterized. Thus uniforming would become more difficult to achieve.

However, a technique called chameleon [Sim2000] can help us to get a solution. The main idea of a chameleon is to define a simple and elegant wrapper class which can hold arbitrary data types and can be used to pass these objects between different program units while maintaining type safety.

Through the generalization above, a leaf page consists of pairs of key and data reference and an index page contains pairs of separator (key) and child pointer. The Key type of the index and leaf page is identical, but child pointer and data reference are different types. If a child pointer and data reference were the same type, the index and leaf pages would be implemented by using the same implicit container. Thus iterators should be the same type, so it would be easier to be uniformed into a common base class. Chameleon can be used to be unify the data reference and child pointer.

```
Value p(a child pointer);
Value  d(a data reference object);
Childpointer p1=p;
DataReference d1=p;
DataReference d1=d;
```

Figure 4.6 a sample using chameleon

As shown in Figure 4.6, in the first line, *p*, is initialized with a child pointer. Therefore, "childpointer *p1=p*" can work well. However, "*d1=p*" will throw an Incompatible_Type_Exception because *p* is of type child pointer at the time of the assignment. But "*d1=d*" can work well because d is of type Data reference.

Because child pointers and data references can be uniformed into one class type by using chameleon technique, index page and leaf page container have the same type of iterator. Thus they can be implemented by using the same implicit container. A class diagram using Chameleon technique and Composite Pattern is shown in Figure 4.7. An index page and a leaf page have almost common structures except that there are more two links with its right and leaf siblings in the leaf page. Therefore, it is not necessary to have an abtract common class for these two pages. Just let LeafPage class inherit from IndexPage class that functions as a base Page class.

Although the chameleon technique can be used to implement the B+-tree index very well, now it can only work on Edison Design Group for their compiler front end. Neither Microsoft VC++ nor g++ work at present time.

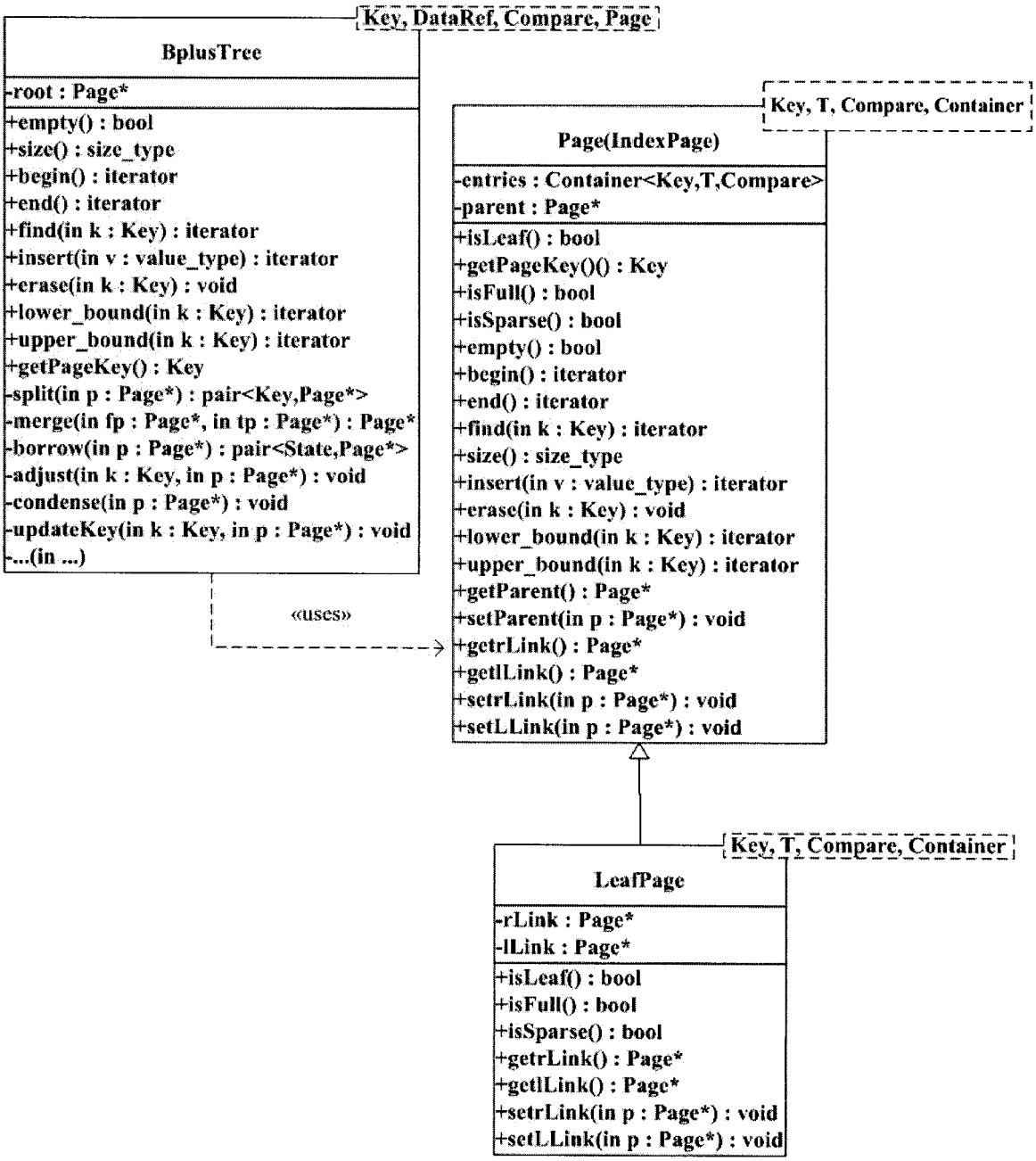

Figure 4.7 B+-tree diagram using chameleon technique

4.3 Our Implementation of B+-tree Index

Can we find a way to implement B+-tree index to follow the STL style, avoid downcasting and run with popular compilers like VC and g++?

As we know, the child pointers in index pages may point to another index page or a leaf page in its lower level. In order to avoid typecasting, we use dynamic polymorphism. Thus a Page class is added as a base class of index pages and leaf page. The index page and leaf page are designed as containers based on the STL style and generic programming. Although these two kinds of pages have almost the same interfaces, the interfaces are not programming but conceptual interfaces. If we try to unify them, that leads to the problems discussed before.

One solution will be proposed in this thesis by combining the composite pattern with the casting-method pattern. This solution does not only avoid downcasting by using casting method, but it does follow the STL style and work with any standard compiler.

Figure 4.8 shows the class diagram of the main classes of one B+-tree implementation. Inheritance leads to Page being an abstract class, and provides a potential use of polymorphism. Page defines virtual functions which act as an interface for its subclasses: IndexPage and LeafPage. This base class cannot only take on polymorphic behavior, but it can also obtain a type-safe reference to a subclass by using casting methods. As a result, BplusTree only holds a pointer to a page but it can get references to the index page and leaf page through this page, and then can invoke the class-specific functions such as *begin(), end()* and *insert()* through these references. IndexPage and LeafPage are designed to be template container classes in the spirit of the STL, so they must conform to all STL interface characteristics. All containers provide their own public functions (built-in algorithms like *find()*). They also provide public iterators and type definitions to allow for interaction with external STL algorithms like *find_if( )* or any new user defined algorithm.

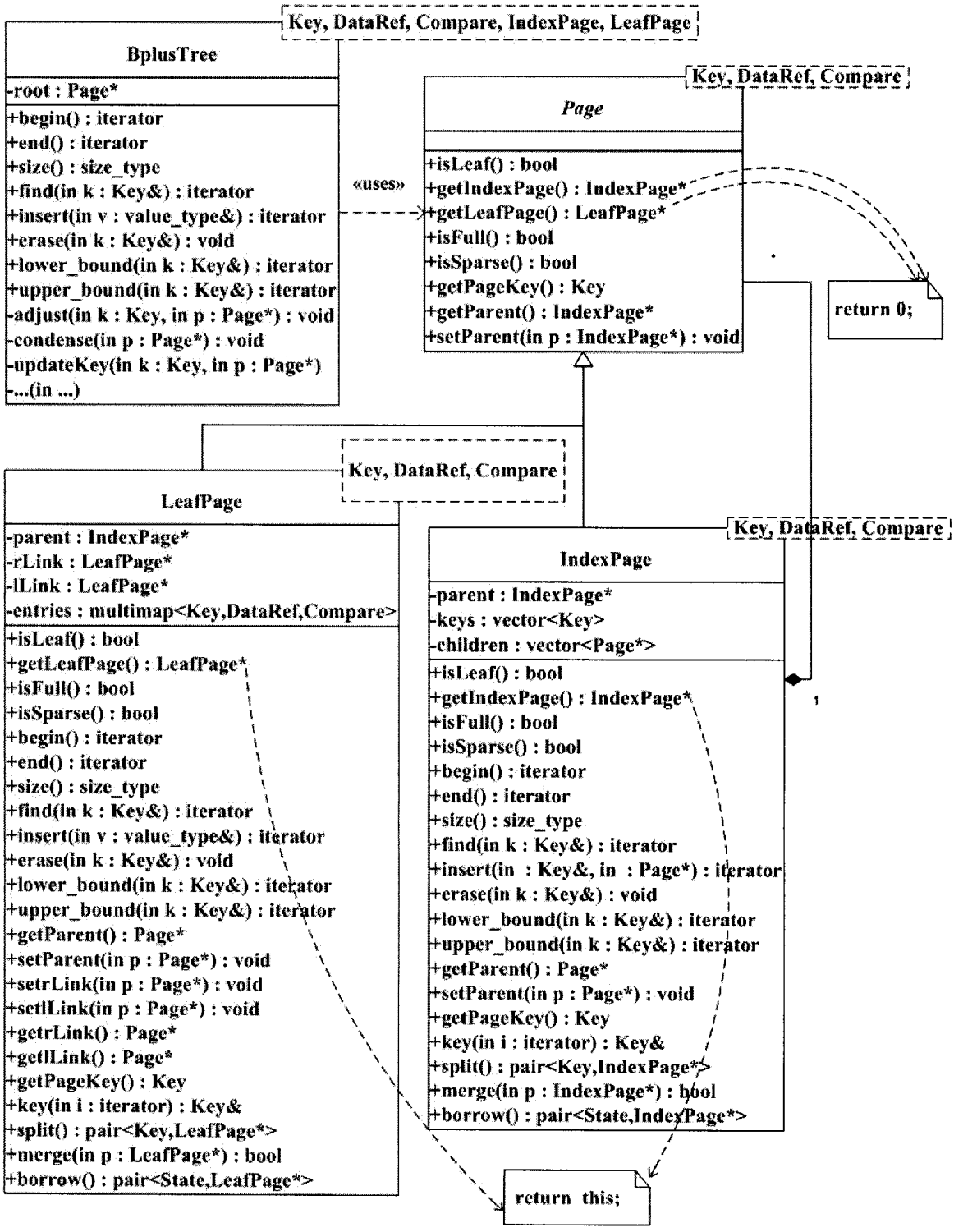

Figure 4.8 B+-tree class diagram using casting-method and composite pattern

The elements in a page are stored in a sequential order. The key part of the elements is at least Less-Than Comparable, meaning that it must be possible to compare two objects of the key type using the operator < and the operator < must define a consistent order [Aus1999], so it can be at least partially ordered. For example, for two elements A and B, we can decide which one is less than the other through their keys, and store them in their order. If we cannot decide, they are said to be equivalent and we store them in the same order as one group. Ordering here can be done only partially but at least between groups, so there exists a complete order. No two groups can be equivalent unless they are the same group.

In case of a group result, the beginning of the group is sufficient to give the necessary element. Since elements are ordered, the rest of the group can be easily located. This mandates a small addition to the generic page concept: Each leaf page should be physically connected to the next leaf page. Even pages are not necessarily physically ordered, it is easy to find the logical order of leaf pages regarding their contents. Then it is easy to link each page with the page that has the immediately next sequence of data. In the design, each leaf page will therefore have a reference to the page that follows it in the sequence. Acting like a linked list data structure, inserting a new page between two pages requires some modifications to keep the logical sequence in order. Linking each page to the previous page as well can make a further addition. It will then act like a doubly-linked list.

4.3.1 Page class

In the B+-tree, there are two kinds of pages: index pages and leaf pages. If the order of the tree is n, an index page contains n keys and $n+1$ child pointers but a leaf page has n pairs of key and data reference. Every child pointer may point to an index page or a leaf page. Therefore, there is a need for a base class or interface to implement this kind of polymorphism.

The Page class shown in Figure 4.9 is an abstract class (interface) that only declares common functions such as *isLeaf()*, *isFull()*, and *isSparse()* in both IndexPage and LeafPage classes and two special virtual functions: *getIndexPage()* and *getLeafPage()*. These two special virtual functions are declared in the Page class but implemented in

IndexPage and LeafPage respectively. The function *getIndexPage()* returns a reference to the current index page and *getLeafpage()* returns a reference to the current leaf page. From a page, we can obtain an IndexPage or LeafPage object references by calling *getIndexPage()* or *getLeafPage()*. Once we get a reference from a derived class, we can call its specific methods directly by using this reference. Thus we can avoid downcasting a page to a leaf page or an index page.

```
template<class Key,class DataRef,class Compare>
class Page
{
  public:
        typedef Key          key_type;
        typedef Compare      compare_type;
        ... ...
        typedef Page<Key,DataRef,Compare> page;
        typedef IndexPage<Key,DataRef,Compare> indexPage;
        typedef LeafPage<Key,DataRef,Compare> leafPage;
        typedef page*        PagePtr_t;
        typedef indexPage*   iPagePtr_t;
        typedef leafPage*    lPagePtr_t;
   public:
        virtual ~Page(){};
        virtual bool         isLeaf()=0;
        virtual iPagePtr_t   getIndexPage(){ return 0;}
        virtual lPagePtr_t   getLeafPage(){return 0;}
        virtual bool         isFull()=0;
        virtual bool         isSparse()=0;
        virtual key_type     getPageKey()=0;
        virtual iPagePtr_t   getParent()=0;
        virtual void         setParent(iPagePtr_t p)=0;


 }; //end of page
```

Figure 4.9 Interface for Page class

4.3.2 Implementation of LeafPage Container

The LeafPage class is implemented to be an associative container like a multimap. It can support multiple keys, which means that a leaf page may contain elements with duplicate keys.

Furthermore, associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

Like the STL multimap, the LeafPage container has at least three explicit template parameters: Key (key_type), Data Reference (mapped_type), and Compare (functor).

A data record in databases, once available, is prepared for the indexes by extracting suitable attributes to be used as keys in each index (primary / secondary) along with the corresponding physical location references (called data reference).

Compare is the functor used when searching objects to decide on the matching keys. The default values allow for simply using the built-in comparison methods as built-in queries, or adding external functors as new user-defined queries new matching criteria.

Each leaf page, as shown in Figure 4.10, stores at most n but at least $n/2$ of pairs of key and data reference if the order of the B+-tree is n. Also, a leaf page keeps a pointer to its parent page if depth first with no stack were used as traversal method or for tree maintenance purpose. In order to support sequential access, a leaf page will have pointers to both its left and right siblings as well.

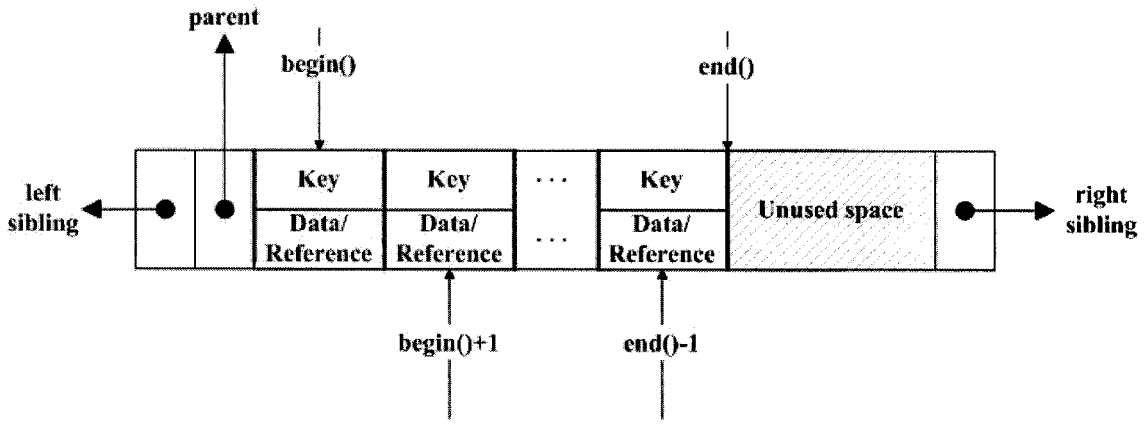

Figure 4.10 LeafPage Structure

Many STL containers can be used as an implicit container, but we consider multimap to be one of the most suitable ones for the following reasons:

1. It supports the efficient retrieval of element values based on an associated key value.
2. Its elements are ordered by key values within the container in accordance with a specified comparison function.
3. It is multiple, because its elements do not need to have a unique key, so that one key value may have many element data values associated with it.
4. It provides bidirectional iterators to access its elements.

Associative vector container is another good choice to be used as an implicit container. Originally it was written in The Loki Library by Andrei Alexandrescu [Alex2001], but it can be implemented with the unique key like std::map on the basic of vector and sort algorithm. Our implementation of the B+ tree index is based on multiple keys, so we must change that original associative vector into a syntactic drop-in replacement for std::multimap but it does not respect all multimap's guarantees. The most important things are: iterators are invalidated by insert and erase operations; the complexity of insert/erase is **O**(N) not **O**(logN) value_type is std::pair<K, V> not std::pair<const K, V>; iterators are random.

The LeafPage container uses its implicit container's iterator as its external iterator. Therefore, the iterator may be a pointer to a pair of (key, data reference). If the container is implemented by using multimap, the iterator of leaf page should be bidirectional

iterator because the iterator provided by the multimap is a bidirectional iterator. If LeafPage is realized by using associative vector, the iterator is a random access iterator.

The LeafPage class shown in Figure 4.11 provides its built-in set of functions, type definitions and iterators to its contents to interact with external algorithms and external iterators. As an associative container, Leaf page container also needs to conform to the STL interface for containers. So it should have the following standard functions:

begin() returns an iterator addressing the first pair of (key, data reference) in the leaf page container by calling its implicit container's *begin* function.

end() returns an iterator that addresses the location succeeding the last pair in a leaf page container by calling its implicit container's *end* function.

find(k) returns an iterator addressing the first location of a pair in a leaf page container that has a key equivalent to the specified key *k* by invoking its implicit container's *find(k).*

insert() inserts a pair or a range of pairs into a leaf page container by invoking its implicit container's *insert* function.

erase() removes a pair or a range of pairs in a leaf page container from specified positions or removes pairs that match a specified key by invoking its implicit container's *erase* function.

lower_bound(k) returns an iterator to the first pair in a leaf page container that with a key that is equal to or greater than a specified key *k* by invoking its implicit container's *lower_bound(k).*

upper_bound(k) returns an iterator to the first pair in a leaf page container that with a key that is greater than a specified key *k* by invoking its implicit container's *upper_bound(k)* function.

Other standard built-in functions are implemented the same way as functions above. The LeafPage class needs to implement all the functions in the base class and some important functions will be used by its own or the B+-tree index's algorithms:

getleafPage() returns a pointer to the current leaf page itself is returned.

IsLeaf() returns true.

IsFull() will returns true if the size of a leaf page is greater than the order of the tree, otherwise it will return false.

Similarly, *isSparse()* will return true if the size of a leaf page is less than half the order of the tree, otherwise it will return false.

getPageKey() returns the key of first element in the leafpage.

getParent() returns the pointer to its parent and *setparent()* assigns a pointer to be its parent.

getrLink() and *setrLink()* are used to deal with its right link in the double linked leaf pages. The same things happen to *getlLink()* and *setlLink()* but they are used to process on its left link.

merge() removes all the elements of that merged sibling and move them into the current leaf page.

split() creates a new page and move half all the elements of current page into the new one, and return a pair of the middle key of the old leaf page and the reference to the new one.

borrow() first checks if it can borrow from its left or right sibling, if yes, it returns a pair of State:SUCCESS and the reference of the borrowed sibling. Otherwise, if it has a sparse right sibling, it returns a pair of State:RIGHT and the reference of its right sibling. If not, a pair of State:LEFT and its left sibling will be returned.(State is an enum type like enum State{SUCCESS,RIGHT,LEFT})

```
template<class Key,class DataRef,class Compare=less<Key> >
class LeafPage: public Page<Key,DataRef,Compare>
{
  public:
        typedef Key          key_type;
        typedef Compare    compare_type;
        ...          ...
        typedef Page<Key,DataRef,Compare> page;
        typedef IndexPage<Key,DataRef,Compare> indexPage;
        typedef LeafPage<Key,DataRef,Compare> leafPage;
        typedef page*         PagePtr_t;
        typedef indexPage*  iPagePtr_t;
        typedef leafPage*     lPagePtr_t;
        typedef AssocVector<Key,DataRef,Compare>  Container;
        typedef typename Container::value_type          value_type;
        typedef typename Container::iterator             iterator;
        typedef typename Container::difference_type   difference_type;
        typedef typename Container::size_type           size_type;
```

```
protected:
        iPagePtr_t      parent;
        lPagePtr_t      rLink;
        lPagePtr_t      lLink;
        Container       entries;
 public:
        LeafPage;
        LeafPage(iPagePtr_t p,lPagePtr_t rl,lPagePtr_t ll,Container e);
        LeafPage(LeafPage& x) ;
        template<class InputIterator>
        LeafPage(InputIterator f,InputIterator l);
        virtual ~LeafPage(){};
        bool empty();
        size_type size();
        size_type count(const key_type& k);
        iterator find(const value_type& v);
        iterator find(const key_type& k);
        iterator begin();
        iterator end();
        iterator insert(const value_type& v);
        iterator insert(iterator pos,const value_type& v);
        template<class InputIterator>
        void insert(InputIterator f, InputIterator l);
        void erase(iterator pos);
        void erase(iterator f,iterator l);
        virtual size_type   erase(const key_type& k);
        size_type max_size() const ;
        iterator lower_bound(const key_type& k);
        iterator upper_bound(const key_type& k);
        std::pair<iterator,iterator> equal_range(const key_type& k);
        virtual bool isLeaf();
        virtual lPagePtr_t getLeafPage();
        virtual key_type& getPageKey();
        key_type&  key(iterator i);
        virtual bool isFull();
        virtual bool isSparse();
        virtual iPagePtr_t getParent();
        virtual void setParent(iPagePtr_t ip);
        void setrLink(lPagePtr_t lp);
        void setlLink(lPagePtr_t lp);
        lPagePtr_t getrLink();
        lPagePtr_t getlLink();
        lPagePtr_t getrSibling();
        lPagePtr_t getlSibling();
        bool merge(lPagePtr_t p);
        std::pair<key_type,lPagePtr_t> split();
        std::pair<State,lPagePtr> borrow() ;
        ......
}; //end of LeafPage
```

Figure 4.11 Interface for LeafPage

4.3.3 Implementation of IndexPage Container

An index page is also designed to be an associative container. The IndexPage container is invisible to applications. It is created and managed by the tree index container. Because child pointers have one more than separators (keys), the IndexPage container will be a special associative container

Like LeafPage container, the IndexPage container should have at least two explicit template parameters: Key (key_type) and Compare (functor), and another explicit parameter Data Reference is also needed here just because of inheritance. Child pointer is a reference to another index page or a leaf page. Child pointer is mapped_type for Key, but it is not necessary to be a template parameter. Key, Data reference and Compare have the same meanings as those in LeafPage container.

The IndexPage class shown in Figure 4.12 stores *n* separators (keys) and *n+1* child pointers if the order of the tree is *n*. Separators (keys) are ordered in an index page. Each index page keeps a reference to its parent page for traversal method or tree maintenance purpose.

Although many STL containers can be used an implicit container for IndexPage container, we think vector to be one of the most suitable. The vector provides its built-in set of functions that can be used easily. In addition, by reserving space as the size of the order of the tree, that eliminates reallocating cost. The iterator provided by vectors is random access iterator, so we can get better efficiency. In our implementation, two vectors are needed: one for keys (also called key container), and the other for child pointers (also called child pointer container).

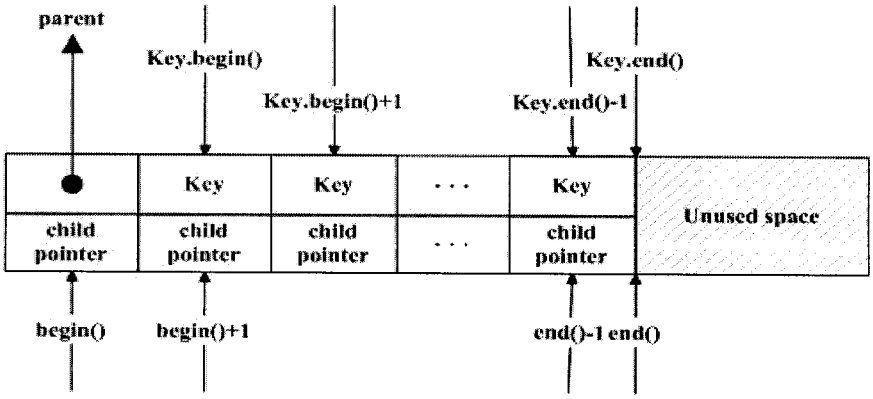

Figure 4.12 IndexPage structure

There are two kinds of iterators provided by two vectors respectively but the index page container uses the iterator of the child pointer container as its external iterator. The iterator of the key container is only used as an internal iterator. From Figure 4.12, we can see that every child has a key called its PageKey in the key container except the first child. The distance between the positions of a child pointer and its PageKey is 1. Therefore, given an iterator i to a child pointer in the child pointer container, we can get the iterator to its PageKey in the key container by computing like *Key.begin()+(std::distance(begin(),i)-1).* If the index page is implemented by using implicit containers like two vectors with a random access iterator, then this computation just takes a constant time **O**(1). The PageKey of the first child is the first key of its leafmost leaf page if the child pointer is treated as the root of a subtree, so it can be gotten by using a function of *getPageKey().*

Figure 4.13 shows the interface of the IndexPage container, including declaration of built-in standard functions, type definitions and iterators to its contents to interact with external algorithms. Following the STL style, the IndexPage container must provide the following standard functions:

begin() and *end()* return the child pointer container's *begin()* and *end().*

find(k) returns an iterator addressing the first location of a child pointer in a index page that has a page key equivalent to the specified key *k*. This function first get the iterator *i* by invoking *Key.lower_bound(k)* of the key container, and then return the iterator to the first child pointer that has PageKey *k* by computing like *begin()+(std::distance(Key.begin(),i) +1).*

insert() inserts a child pointer or a range of child pointers into the child pointer container and the page keys of the inserting child pointers into the key container in a index page. If the first one is inserted, its PageKey will be ignored.

erase() removes an child pointer or a range of child pointers in a index page from specified positions or removes child pointers that match a specified key. At the same time, it also needs to remove the keys corresponding to those erasing child pointers in the key container.

lower_bound(k) returns an iterator to the first child pointer in a index page that with a page key that is equal to or greater than a specified key *k*. It can be implemented the same way as *find(k).*

upper_bound(k) returns an iterator to the first child pointer in a index page that with a page key that is greater than a specified key *k*.

Other standard built-in functions are implemented the same way as functions above. Similar to the LeafPage, the IndexPage container also must implement all the functions in the Page class and other functions to be used by B+-tree index:

getIndexPage() returns a pointer to the current index page itself is returned.

IsLeaf() returns false.

IsFull() will return true if the size of a leaf page is greater than the order of the tree plus one, otherwise it will return false. Similarly, *isSparse()* will return true if the size of a leaf page is less than half the order of the tree plus one, otherwise it will return false.

getPageKey() returns the PageKey of first child pointer in the index page.

getParent() returns the pointer to its parent and *setparent()* assigns a pointer to be its parent.

merge() combines that merged sibling with the current index page.

split() creates a new page and move half all the elements of current page into the new one, and returns a pair of the middle key of the old index page and the reference to the new one.

borrow() is similar to the borrow function of a leaf page but returns a pair of State and the related pointer to an index pointer.

```
template<class Key,class DataRef,class Compare=less<Key> >
class IndexPage:public Page<Key,DataRef,Compare>
{
 public:
    typedef Key                          key_type;
    typedef Compare                      key_compare;
    typedef std::vector<Key>             Key_Container;
    typedef std::vector<PagePtr_t>       Container;
    typedef Container::value_type        value_type;
    typedef Key_Container::iterator      key_iterator;
    typedef Container::iterator          iterator;
    typedef Container::difference_type   difference_type;
    typedef Container::size_type         size_type;
    ... ...
 protected:
    iPagePtr_t        parent;
    Key_Container     keys;
```

```
    Container          children;
  public:
    IndexPage();
    IndexPage(iPagePtr_t p,Key_Container k,Container c);
    template<class InputIterator>
    IndexPage(InputIterator f,InputIterator l);
    virtual ~IndexPage(){};
    IndexPage& operator=(const IndexPage&);
    virtual bool empty();
    virtual size_type size();
    virtual size_type count(const key_type& k);
    iterator find(const value_type& v);
    iterator find(const key_type& k);
    iterator begin();
    iterator end();
    iterator insert(const value_type& v);
    iterator insert(key_type& k, const value_type& v);
    iterator insert(iterator pos,key_type&k, const value_type& v);
    void insert(InputIterator f, InputIterator l);
    void erase(iterator pos);
    void erase(iterator f,iterator l);
    size_type erase(const key_type& k);
    iterator lower_bound(const key_type& k);
    iterator upper_bound(const key_type& k);
    std::pair<iterator,iterator> equal_range(const key_type& k);
    virtual bool isLeaf();
    virtual iPagePtr_t getIndexPage();
    virtual key_type&  getPageKey();
    key_type&  key(iterator i);
    virtual iPagePtr_t getParent();
    virtual void setParent(iPagePtr_t ip);
    virtual bool isFull();
    iPagePtr_t getrSibling();
    iPagePtr_t getlSibling();
    bool merge(iPagePtr_t p);
    std::pair<key_tpe,iPagePtr_t> split();
    std::pair<State, iPagePtr> borrow();
  ......
}; end of Index Page
```

Figure 4.13 Interface for IndexPage

4.3.4 Implementation of B+-tree Index Container

The implementation of the B+-tree index container shown in Figure 4.14 is based on Leaf Page and Index Page containers. The B+-tree index is initialized from an empty Leafpage container. However, the index will dynamically grow or condense with insertions or deletions. The B+-tree index container is designed and implemented to be an associative container, so it supports equality and range-searches efficiently.

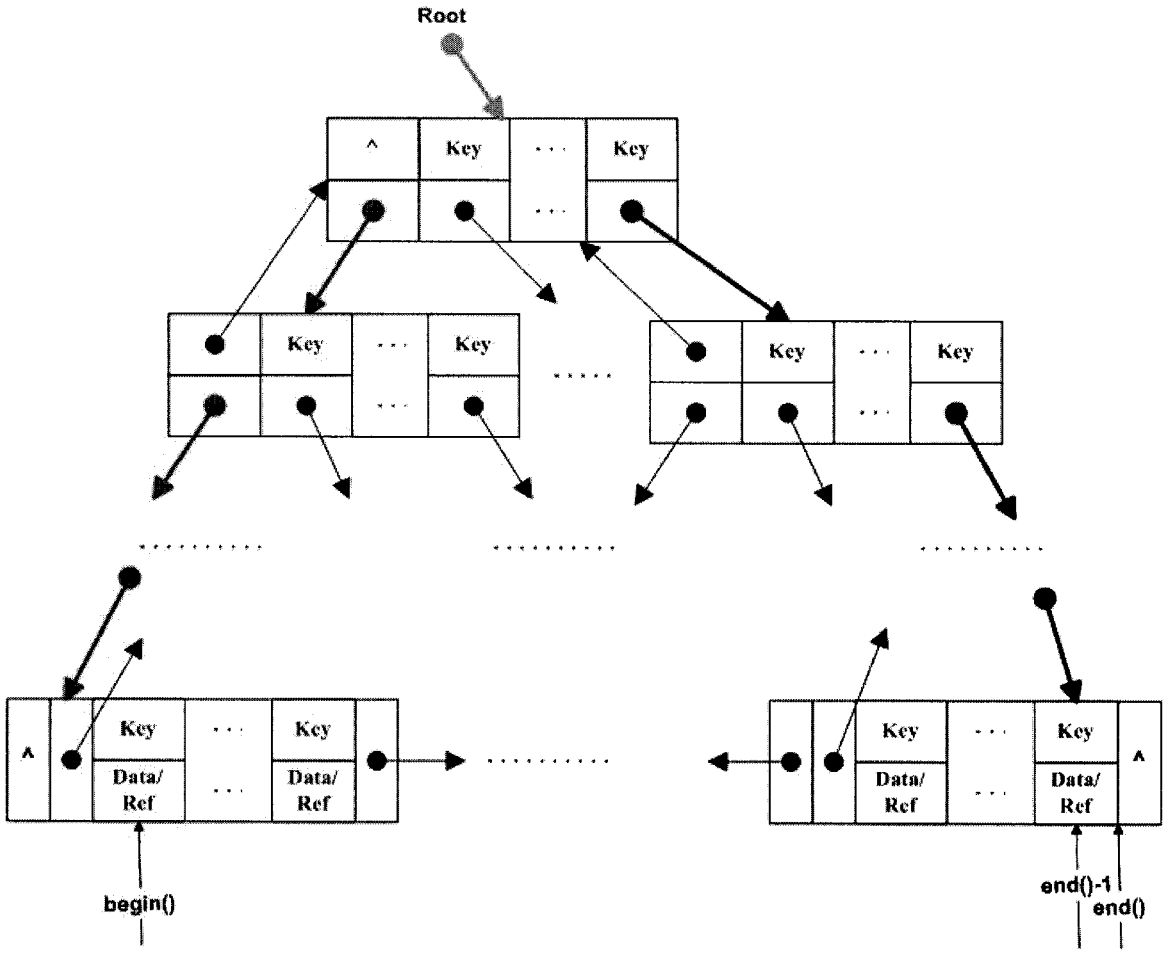

Figure 4.14 B+-tree structure

The B+-tree index container has at least five template parameters: Key(key_type), Data reference(mapped_type), Compare(functor), LeafPageContainter, and IndexPageContainer. The first three parameters are the same as those in LeafPage and IndexPage. LeafPageContainer and IndexPageContainer make B+-tree index container more flexible because they can be replaced with other or user-defined page containers.

The B+-tree index container must provide a set of standard STL types that are internally translated to the equivalent container-specific types with typedef statements. This allow for the container to be easily integrated with other STL components.

The B+-tree index container only holds a pointer to a root page. In the operations of B+-tree index, the required pages are loaded on demand through a proxy mechanism.

Roughly speaking, an iterator is a small, light-weight object, which is associated with B+-tree index container. It is the only way to allow access to elements within B+-tree index containers. The Iterator is a nested class defined within B+-tree index container class and is a friend to this container. B+-tree index container is a double link list of leaf pages, but the elements that B+-tree iterators shown in Figure 4.15 are iterating over are pairs of key and data reference. Therefore, a B+-tree iterator should point to a pair that is determined by the position of this pair. If we want the place of the pair, a leaf page pointer where the pair is stored and the leaf page iterator that points to this pair must be provided.

The iterator had to be integrated with the STL, and that's why it is derived from a standard Iterator tag class like std::iterator<std::random_access_iterator_tag, T>. Iterator tags are an ingenious mechanism that allows some STL algorithms to be tailored to the specific capabilities of each iterator type. This derivation tells the client algorithms that the class just introduced is an iterator that supports random access (i.e., indexed access, like pointers) and accesses elements of type T. In the iterator of B+-tree index container, T should be pair of (key, Data reference).

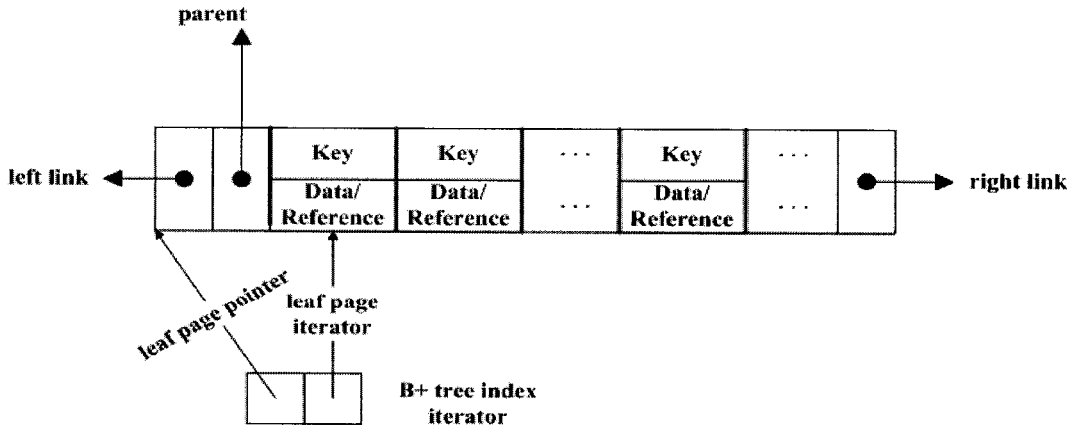

Figure 4.15 B+-tree iterator structure

The Iterator class must implement the following main operations below:

*operator** dereference the leaf page iterator (returning a reference to a pair of key and data reference).

operator++ () shown in Figure 4.16 advances the iterator to the next element using in-order traversal. This function is the heart of the class. Suppose *p* to be the current leaf pointer and *i* to be the current leaf page iterator in this leaf page.

```
iterator& operator++ ()
{
  if (i!=p->end())
  {
     if((++i)==p->end() && ((p->getrLink())!=NullPtr))
     {
         p=p->getrLink();
          i=p->begin();
     }
   }else{
     if((p->getrLink())!=NullPtr)
     {
          p=p->getrLink();
           i=p->begin();
           }
      }
   return *this;
}
```

Figure 4.16 Sample codes for operator ++()

In Figure 4.16, we can see first the current leaf iterator *i* performs a ++ operation. After that, if *i* has not reached the *end()* of the current leaf page *p*, ++ operation is completed. Otherwise, p will move to its right link and *i* also moves to the *begin()* of the page the link points to if the link is not null pointer.

operator-- () shown in Figure 4.17 advances the iterator to the next element using in-order reversal traversal. The process of -- operation is similar to ++ operation. The difference between them is that *p* goes to its leaf link and *i* goes to the position next to the *end()* of the left link when the *begin()* of the current leaf page *p* is reached

```
iterator& operator-- ()
{
    if (i!=p->begin()) --i;
   else{
        if(p->getlLink()!=NullPtr)
        {
          p=p->getlLink();
          i=--(p->end());
        }
    }
   return *this;
 };
```

Figure 4.17 Sample codes for operator –()

operator==() compares two iterators for equality, not the elements they refer to. If the leaf page pointers and leaf page iterators of these two iterators are equal, true will be returned. Otherwise, false will be returned.

operator!=() compares two iterators for inequality, not the elements they refer to. This operator is implemented by using operator = =.

If iterator class is designed to be a random access iterator, *operator+(int n)*, *operator –(int n)*, *operator <(iterator)* and other functions related to the requirements of random access iterator type are required. Iterator must also have a default constructor, copy constructor, assignment operator and destructor, but the implementation of all of these can be not very complicated.

As an associative container, the B+ tree index container also needs to conform to the STL interface for containers. Therefore it has the following main standard functions:

begin() always begins at the root page and follows the first child pointer of the page from page to page until the first leaf page is arrived. *begin()* returns an iterator addressing the first pair of (key, data reference) in the first leaf page container.

end() always begins at the root page and follows the last child pointer of the page from page to page until the last leaf page is arrived. *end()* returns an iterator that addresses the location succeeding the last pair in the last leaf page container.

find is the fundamental operation in using the index. The application uses the index to locate data by searching its contents to find where the physical data resides. As mentioned before, the index does not provide the data we are looking for, but tells us where it is stored, so the data returned by the index is typically a reference to a location.

find by a key is the most basic form. It uses the internal comparison operator (the built-in functor) inside the index to find the first data entry whose key matches the searched key.

```
iterator find(const key_type& k)
 {
          PagePtr_t  pPtr=root;
          iPagePtr_t iPtr;
          lPagePtr_t lPtr;

         // Iteratively traverse children to the leaf page
          while(pPtr!=NullPtr && !pPtr->isLeaf())
          {
             iPtr=pPtr->getIndexPage();
             pPtr=*(iPtr->lower_bound(k));
          }
         //searching in the leafpages
          if(!pPtr && pPtr->isLeaf()) lPtr=pPtr->getLeafPage();
          leaf_iterator i=lPtr->find(k);
          if(i==lPtr->end())
          {
                  if(lPtr->getrLink()!=NullPtr)
                  {
                     lPtr=lPtr->getrLink();
                      i=lPtr->find(k);
                     if(i==lPtr->end())   return end();
                  }else return end();
          }
        return iterator(lPtr,i);
 }  //end of find()
```

Figure 4.18 sample codes of find()

As shown in Figure 4.18, searching a B+-tree for a key *k* always begins at the root page. If the root page is not a leaf page, the root page will first be cast to the index page by invoking the function of *getIndexPage()*. Then the child pointer to a page in the lower level will be obtained by deferencing the index page iterator returned by *lower_bound(k)* against the searched key k in this index page. This process will be recursive till a leaf page is reached. By invoking *getLeafPage()*, a reference to this leaf page will be returned. Then call the *find(k)* of this leaf page, and a leaf page iterator *i* will be achieved. If *i* has not reached the end of this leaf page, a tree iterator that is made up of the leaf page pointer and the leaf page iterator *i*. If *i* is *end()* of this page, the right leaf page of this page has the same process if it is not null pointer. Otherwise, the *end()* of the tree will be returned.

insert(pair) (as shown in Figure 4.19) first chooses a correct leaf page and insert a pair of (key and data reference) into this leaf page. Insertion may cause splits. In a case, splits increase the width of the tree. Root split increases its height

Like the find operation, choosing the correct leaf page begins at the root page and invokes the upper_bound function against the key of the inserted pair. By dereferencing the iterator returned by the upper_bound to get the child pointer to a page in the next level, choosing function follows this process from page to page recursively until a leaf page is reached.

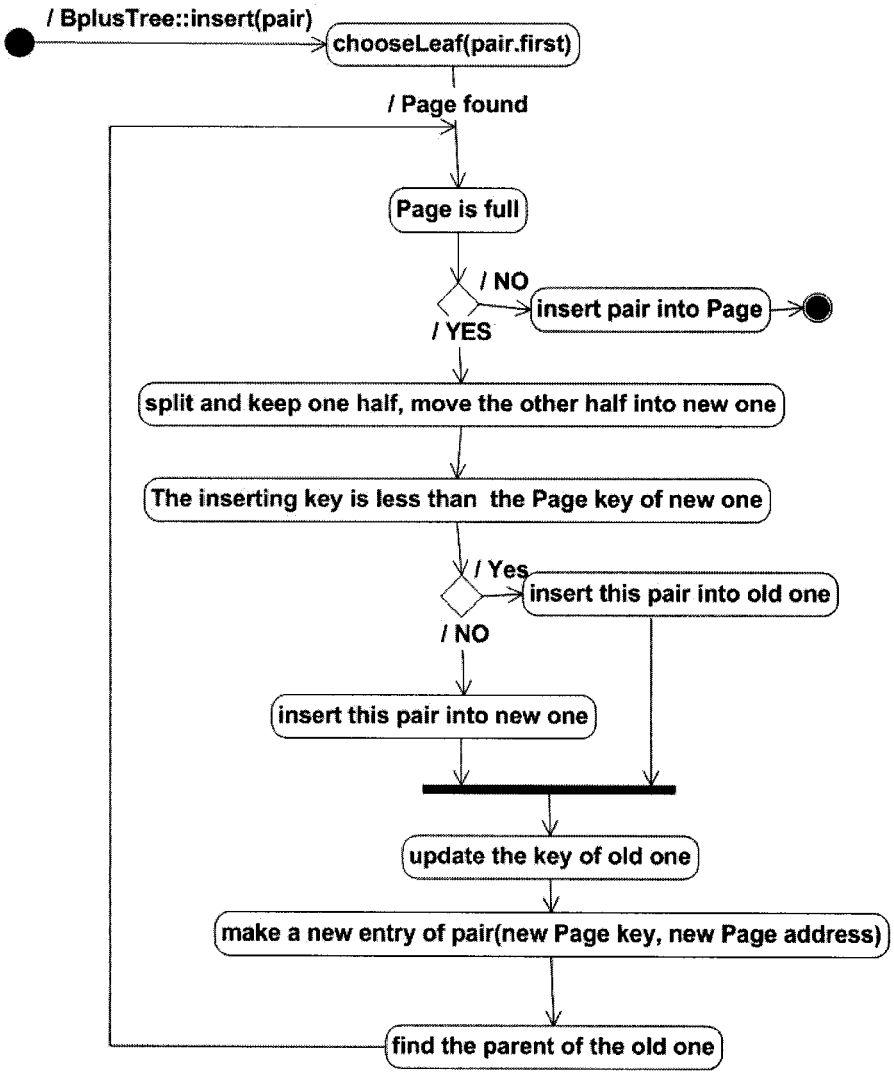

Figure 4.19 B+-tree insert activity

When we insert a pair into a B+-tree, each scenario causes a different action in the insert algorithm. The scenarios are shown in Table 4.1:

The insert algorithm for B+-tree		
Leaf Page Full	**Index Page Full**	**Action**
NO	NO	Place the pair in sorted position in the appropriate leaf page
YES	NO	1. Split the leaf page 2. Place a copy of Middle Key in the index page in sorted order. 3. Left leaf page contains pairs with keys below the middle key. 4. Right leaf page contains pairs with keys equal to or greater than the middle key.
YES	YES	1. Split the leaf page. 2. Pairs with keys < middle key go to the left leaf page. 3. Pairs with keys >= middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

Table 4.1 The insert algorithm for B+-tree

erase(iterator) (as shown in Figure 4.20) takes an iterator as a parameter and returns a void. *erase* first chooses a correct leaf page and remove the pair that the iterator points to from this leaf page. Deletion may cause borrowingss and merges. If a borrowing happened, an entry has to be moved from its borrowed sibling. If a merge occurred, one must delete the entry of its merged sibling from its parent. A merge could propagate to the root, possibly decreasing height of the tree. Borrowings and merges may cause to update PageKeys of some pages. If the PageKey of a page is changed, one entry responding to this page in its parent may be updated. If this entry is also the PageKey of its parent, the key updating will be recursive till the root is reached.

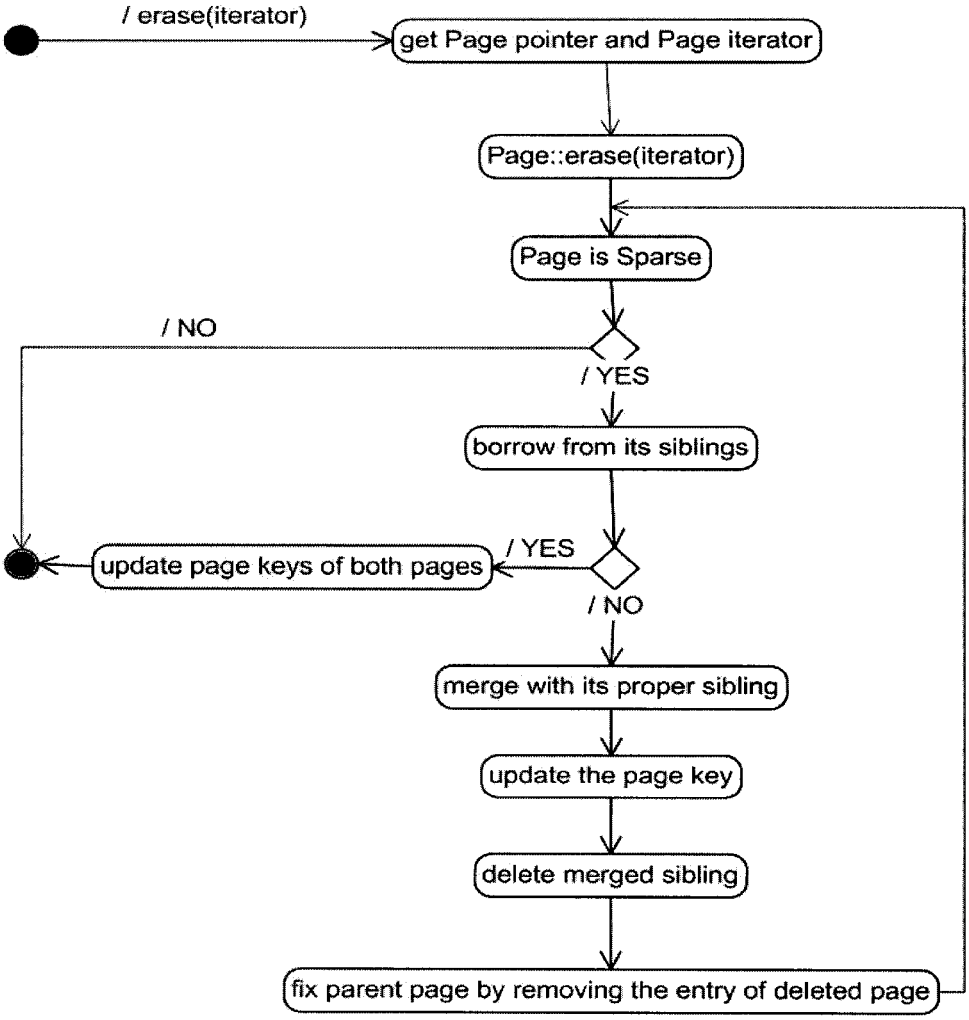

Figure 4.20 B+-tree erase activity

When we erase a record from a B+-tree, each scenario causes a different action in the erase algorithm. The scenarios are shown in Table 4.2:

The erase algorithm for B+-trees		
Leaf Page Sparse	**Index Page Sparse**	**Action**
NO	NO	delete the pairs from the leaf page.
YES	NO	1. Check if the leaf page can borrow from its right or left sibling. If yes, borrow. 2. If not, merge the leaf page with its sibling. 3. Adjust the index page to reflect the change.
YES	YES	4. Check if the leaf page can borrow from its right or left sibling. If yes, borrow. 5. If not, merge the leaf page with its sibling. 6. Adjust the index page to reflect the change. 7. Check if the index page can borrow from its right or left sibling. If yes, borrow. 8. If not, merge the index page with its sibling. 9. Continue combining index pages until you reach a page with no sparse or you reach the root page.

Table 4.2 The erase algorithm for B+-trees

```
template <class Key,
          class DataRef,
          class Compare=std::less<Key>,
          class IndexPageContainter=IndexPage<Key,DataRef,Compare>,
          class LeafPageContainer=LeafPage<Key,DataRef,Compare> >
class BplusTreeIndex
{
public:
   typedef Key                                  key_type;
   typedef Compare                              key_compare;
   typedef Page<Key,DataRef,Compare>            page;
   typedef IndexPage<Key,DataRef,Compare>       indexPage;
   typedef LeafPage<Key,DataRef,Compare>        leafPage;
   typedef page::PagePtr_t                      PagePtr_t;
   typedef page::iPagePtr_t                     iPagePtr_t;
   typedef page::lPagePtr_t                    lPagePtr_t;
   typedef leafPage::value_type                value_type;
   typedef value_type*                          pointer;
```

```
   typedef value_type&                          reference;
   typedef size_t                               size_type;
   typedef ptrdiff_t                            difference_type;
   ...                          ....                    ....
protected:
  PagePtr_t root;
public:
     BplusTreeIndex(PagePtr_t p):root(p){};
     BplusTreeIndex();
     BplusTreeIndex(BplusTreeIndex& x );
     template<class InputIterator>
     BplusTreeIndex(InputIterator f,InputIterator l);
     virtual ~BplusTreeIndex();
     class iterator:public
std::iterator<std::random_access_iterator_tag,value_type,difference_type>;
     iterator begin();
     iterator end();
     size_type size();
     iterator find(const key_type& k);
     iterator lower_bound(const key_type& k);
     iterator upper_bound(const key_type& k);
     std::pair<iterator,iterator> equal_range(const key_type& k);
     iterator insert(const value_type& aPair)  ;
     template <class InputIterator>
     void insert(InputIterator f,InputIterator l);
     void erase(iterator pos);
     void erase(iterator f,iterator l);
     size_type erase(const key_type& k);
     size_type max_size();
     void clear();
     ... ...
private:
     lPagePtr_t chooseLeaf(const key_type& k);
     void adjustTree(const key_type& k, PagePtr_t page);
     void condenseTree(PagePtr_t page);
     void updateKey(PagePtr_t p,const key_type& k);
     ... ...
}; end of Bplustree
```

Figure 4.21 Interface for B+-tree index

4.3.5 Proxy Mechanism

The Index page or Leaf page container is meant to be small enough to easily fit into memory. However, the B+-tree index container does not have to fit completely in the memory and typically it will not. The B+-tree index uses a Proxy mechanism to manage controlled access to the storage of the index.

Smart Pointer

Figure 4.22 shows the Proxy (smart pointer) activities. Only the root of a B+-tree is loaded initially, and resides in memory until the B+-tree is destroyed. Each access to a non-root page checks if the page is in memory. If yes, the Proxy returns a smart pointer to the tree algorithm. Otherwise, the Proxy will check if Cache has a reference to the page.

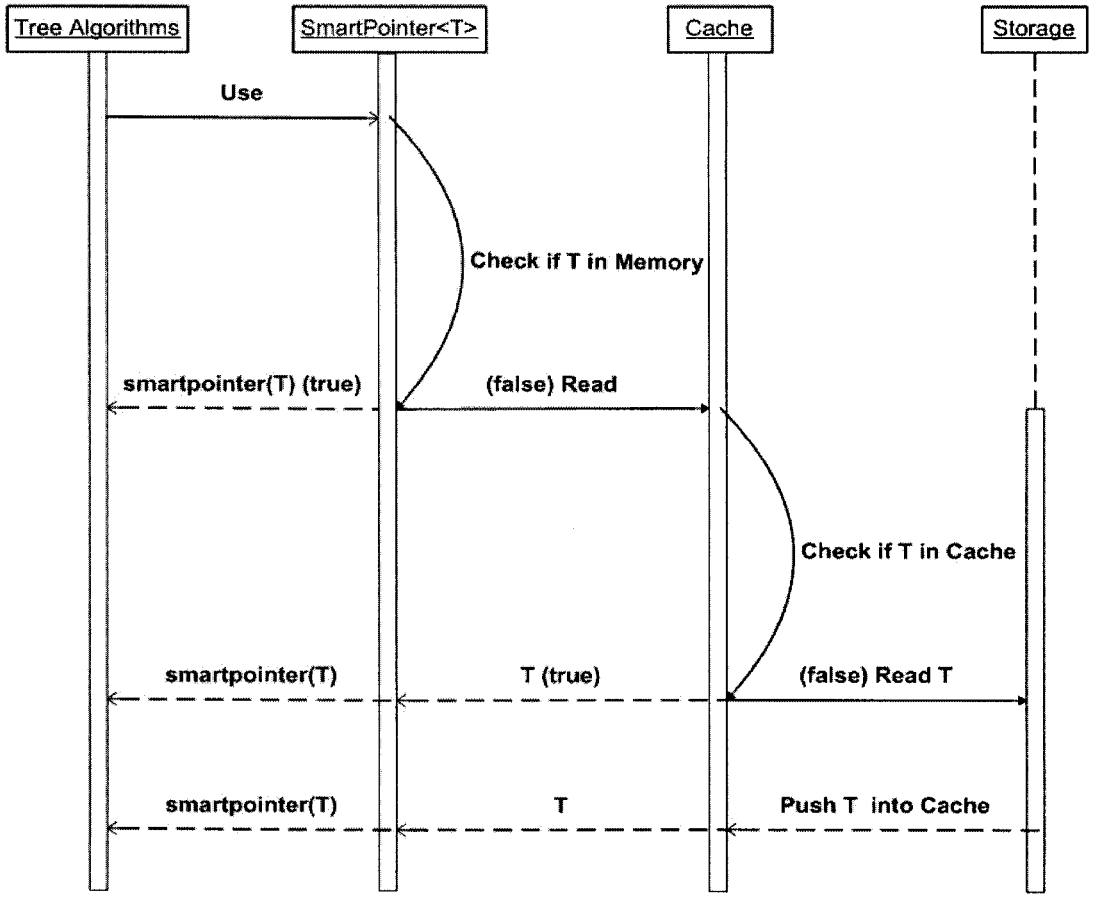

Figure 4.22 Proxy(smart pointer) activity

If the page reference is in Cache, the tree algorithm also can quickly get a smart pointer to the page. If not, the Proxy needs to read the page object from the storage.

In the B+-tree implementation, a smart pointer can be used for many purposes such as garbage collection, exchanges between memory pointers and disk pointers, and locking mechanism. A smart pointer is like a bridge between memory and physical storage. It is important for STL-like containers to consider the specific garbage collection scheme used. A Non-intrusive reference counting smart pointer in Figure 4.23 is suited for the B+-tree implementation with the STL style. The number of object references is stored in a counter that maintains a count of the smart pointers that point to the same object. The smart pointer will delete the object when this count becomes zero. The diskPointer holds an offset to a page in the index files, and a memoryPointer maintains a reference to a loaded page in memory. Reference counting smart pointer requires using locks if the pointers are used by more than one thread of execution.

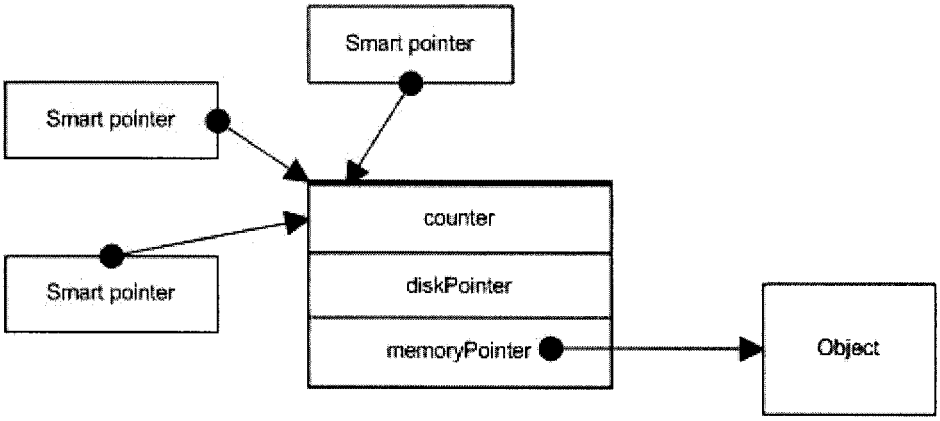

Figure 4.23 Structure of non-intrusive reference counting smart pointer

Figure 4.24 shows the interface of SmartPointer class. Template parameters T and Cache stand for a reference object type and cache type. By using smart pointers, a page is only loaded on demand. Actually, the need of loading a page is determined by using * and -> operations with a smart pointer. If the page is loaded already in the cache, these operations just cost a memory access only. If not, the cache manager will be responsible for loading the page from the storage using a read operation and put it into the cache. If the page is changed and become dirty, it will be written back to the storage by invoking write function. It will take a disk access time.

```
struct RefCounter
{
   RefCounter () : totalRefs_ (0), strongRefs_ (0),dirty_(false){ }
   bool dirty_;
   long totalRefs_;
   long strongRefs_;
};

template<class T, class Cache>
class SmartPointer
{
 public:
    SmartPointer();
    SmartPointer(const T* x);
     ~SmartPointer();
    template<class U>
    SmartPointer(const SmartPointer<U,Cache>& x);
    template<class U>
    SmartPointer& operator=(SmartPointer<U,Cache>& x);
    template<class U>
    SmartPointer(const WeakPointer<U,Cache>& x);
    template<class U>
    SmartPointer& operator=(WeakPointer<U,Cache>& x);
    T& operator*() const
    T* operator->() const
     RefCounter& getCounter();
     T& get();
     setID();
     getID();
     void read(id);
     void write(id);
     bool dirty();
      ......
  private:
    RefCounter counter;
    T* pRefered;
     long id;
     void release();
}
```

Figure 4.24 Interface for SmartPointer

However, the problem with smart pointers is the use of circular references. If object A has a smart pointer that points at object B which has a smart pointer that points back at A, then neither object will ever be destroyed since their reference counters will never reach zero. It can be difficult to prevent such circular references. One approach to fixing this problem is to use another smart pointer type often used for a weak pointer that does not

increment or decrement the reference counter. This weak pointer, as shown in Figure 4.25, can cooperate with the smart pointer so that when the object is deleted, the weak pointer will be automatically set to Null pointer, thus preventing dangling pointers.

```
template<class T, class Cache>
class WeakPointer
{
 public:
    WeakPointer();
    WeakPointer(const T* x);
     ~WeakPointer();
    template<class U>
    WeakPointer(const WeakPointer<U,Cache>& x);
    template<class U>
    WeakPointer& operator=(SmartPointer<U,Cache>& x);
    template<class U>
    WeakPointer(const WeakPointer<U,Cache>& x);
    template<class U>
    WeakPointer& operator=(SmartPointer<U,Cache>& x);
     ... ...
    see Figure 4.24
}
```

Figure 4.25 Interface for WeakPointer

The former implementation of B+-tree index can work in memory, but if we want to let the B+ tree index connect with the storage, one thing needed to do is that all the raw pointers are simply replaced with smart pointers. Furthermore, because we use template and generic programming to implement B+-tree index, we need to change some type definitions in our programs and do not need to change other parts. We take LeafPage class in Figure 4.11 as an examples to show how to change some type definitions in the LeafPage class. If a page class has data members that are pointers to its parent, rLink, or lLink, these pointers are needed to be replaced with WeakPointers that are used to fix circular references.

```
template<class Key,class DataRef,class Compare=less<Key> >
class LeafPage: public Page<Key,DataRef,Compare>
{
  public:
        typedef Key          key_type;
        typedef Compare    compare_type;
        …          …
        typedef Page<Key,DataRef,Compare> page;
        typedef IndexPage<Key,DataRef,Compare> indexPage;
        typedef LeafPage<Key,DataRef,Compare> leafPage;
        typedef Cache<long, Storage> CacheType;
        typedef SmartPointer<page,CacheType>           PagePtr_t;
        typedef SmartPointer<indexPage,CacheType>   iPagePtr_t;
        typedef SmartPointer<leafPage,CacheType>     lPagePtr_t;

  protected:
        WeakPointer<indexPage,CacheType>    parent;
        WeakPointer<leafPage,CacheType>      rLink;
        WeakPointer<leafPage,CacheType>      lLink;
        Container          entries;
    … …
  This part is not changed, and please see Figure 4.11
```

Figure 4.26 Interface of LeafPage with SmartPointers

Cache

The use of a main-memory buffer (also called a Cache shown in Figure 4.27) is a good way to increase the speed of an index system. The global cache management consists of two processing components [HS1999]: allocation and replacement. Allocation distributes global buffer space among concurrent transaction and replacement is responsible for accessing of the global buffer and page replacement operations. Cache allocation appears more important when there is contention for the globe buffer, so we use the Singleton design pattern [Alex2001] to manage Cache instances. A Singleton is used when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point. This ensures the Cache has only one instance and provides a global point of access. If the number of the pages becomes too big to entirely fit in the cache, a replacement strategy is needed. The life span of a page, except the root, in memory depends on the replacement strategy of Cache. When a page is removed from the cache, it will be destroyed in the memory heap. In our B+-tree index, we use a Least Recently Used (LRU) [HS1999] replacement strategy. That means the least recently used

page will be replaced with a new page when the cache is full. The LRU methods keeps track of the actual requests for pages. Since the root is requested on every search, it seldom, if ever, is selected for replacement. The page to be replaced is the one that has gone the longest time without a request for use. Thus, frequently accessed parts of the tree will remain in memory and the memory protection mechanism isolates other users through demand paging technique.

```
template <class T, class Storage> class Cache
{
public:
   typedef SingletonHolder<Cache>          CacheType;
   typedef SmartPointer<T,CacheType>       SmartPtr_t;
   typedef WeakPointer<T,CacheType>        WeakPtr_t;
   typedef multimap<long,SmartPtr_t>       Buffer_t;
   typedef multimap<int,long>              Count_t;
   typedef typename container_type::iterator iterator;
   ... ...
private:
   Buffer_t container;
   long length;
   Storage storage;
   Count_t times;
public:
   Cache(long l);
   ~Cache(){ };
   iterator begin();
   iterator end();
   size_type size() ;
   bool empty() ;
   SmartPtr_t getPointer(long id);
   void insert(long id, SmartPtr_t ptr);
   SmartPtr_t createNew(PageType)
   void deleteObject(long id);
   long getRootID();
   bool isRoot(SmartPtr_t root);
   SmartPtr_t getRoot();
   void setRoot(SmartPtr_t root)
   bool hasRoot()
   void erase(iterator pos);
   void clear();
   Storage* getStorage();
   Void setStorage(Storage*);
};
```

Figure 4.27 Interface for Cache

Storage

In order to rebuild a B+-tree index from index files, we need to maintain a copy of the B+-tree structure on disk shown in Figure 4.28. In our implementation, the index page and leaf page are mingled within the same file to avoid seeking between two separate files while accessing the tree, but they have different Page Types.

The most critical feature of secondary storage devices is that they take a relatively long time to seek to a specific location, but once the read head is positioned and ready, reading or writing a stream of contiguous bytes proceeds rapidly. This combination of slow seek and fast transfer leads naturally to the notion of index paging. To reduce the number of seeks associated with any search, a B+-tree index is divided into and then each page is stored in a block of contiguous location on disk. The size of a block is usually determined by many factors such as the characteristics of the disk drive, and the amount of memory available. In general, the size of a page is that of a block

Storage class is mainly responsible for manging and controlling accesses to the index files on disk. In B+-tree index, a block is the basic unit for I/O operation. When a new page object is needed to wirte into the index files, Storage allocates a block for it on the physical storage. When a page object is deleted from the files, Storage collects the block used by the page and reallocate it.

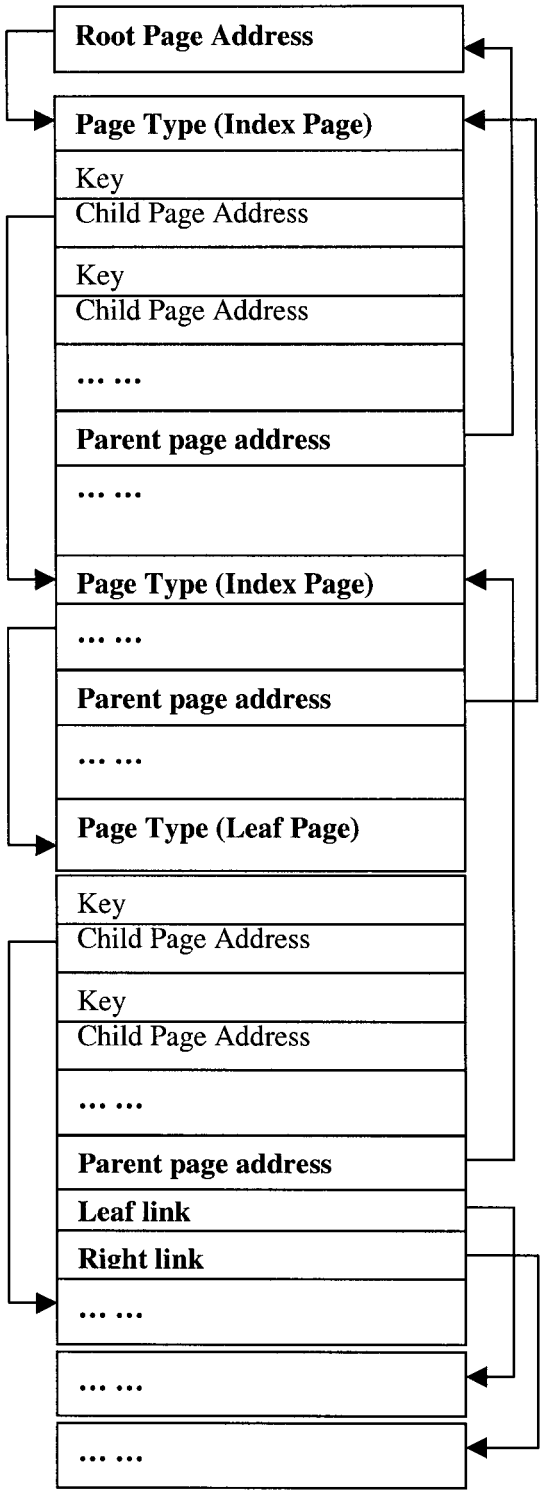

Figure 4.28 Physical Storage Structure

Serialization

In our B+-tree index, serialization is used to read or write a page to or from the index files. The basic idea of serialization is that a page should be able to write its current state, usually indicated by the value of its member variables, to persistent storage. Later, the page object can be re-created by reading, or deserializing, the object's state from the disk. Thus, for a page class to be serializable, it must implement the basic serialization operations in the page.

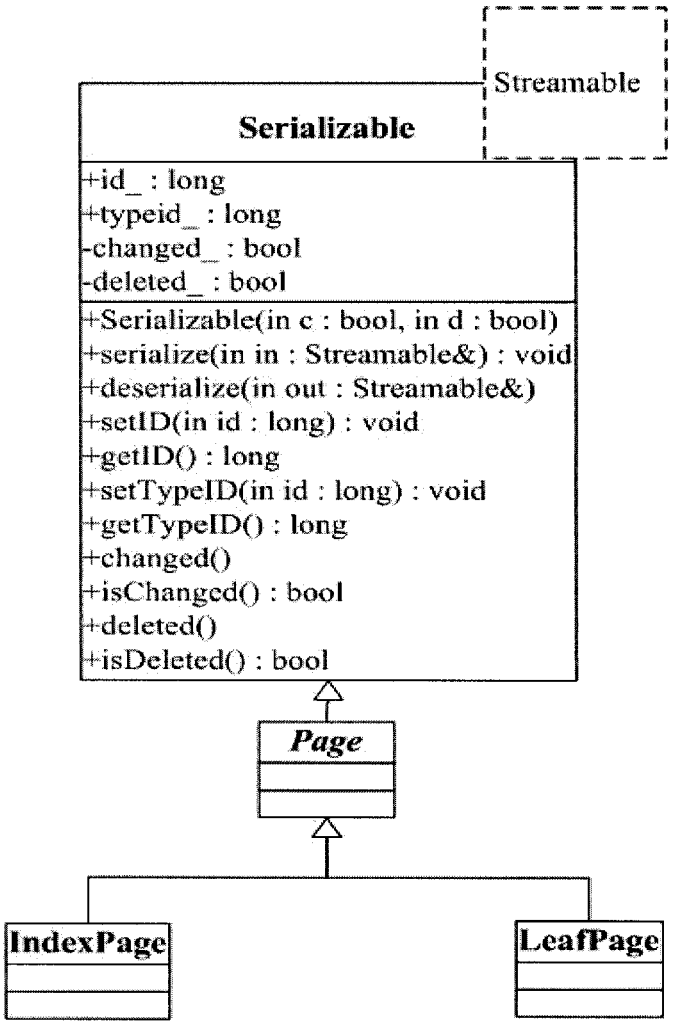

Figure 4.29 a class diagram related to serialization

Figure 4.29 shows a class diagram related to serialization. In order to add this functionality to B+-tree index, we let the base class Page inherit from the Serializable

class that declares serialize, deserialize and other related methods. Template parameter in Serializable class provides an efficient conduit to persistent storage. Instead of directly reading and writing the file, we serialize data to and from Streamable that is hooked up to the index files. Streamable uses overloaded insertion (<<) and extraction (>>) operators to perform writing and reading operations. IndexPage and LeafPage must implement these two methods: *serialize(Streamable& in)* and *deserialize(Streamable& out).* We take the serialize operation shown in Figure 4.30 of LeafPage as example:

```
void serialize(Streamable& out>
  {
   Serializable<Streamable>::serialize(out);
   out<<parent;
   out<<rLink;
   out<<lLink;
   out<<entries;
   }
```

Figure 4.30 serialization method

4.3.6 Use B+-tree Index

The B+-tree index will be used in tree-based index framework shown in Figure 4.31. Because the B+-tree index is implemented to be a STL-like container, it can be easily replaced with other search trees such as R-tree and SS-tree in the framework. When the B+-tree index as a template parameter is passed to the framework, the TreeIndex can invoke all the functions provided by the B+-tree container through its reference. The iterator of the B+-tree index container provides controlled access to its elements.

```
template<
class Key, class DataRef, class  Compare=less<Key> >
class SearchTree = BplusTree<Key, DataRef,Compare>
>
class TreeIndex
{
  public:
      typedef Key                          key_type;
      typedef SearchTree::iterator         iterator;
      typedef SearchTree::value_type  value_type;
       ... ...
      iterator find(key_type& k)
      {
         return searchtree.find(k);
       }
      iterator insert(const value_type& aPair)
      {
         return searchtree.insert(aPair);
      };
      void erase(key_type& k)
      {
        return searchtree.erase(k);
       }
      ......
 private:
      SearchTree searchtree;
      ..... ..
}; //end of TreeIndex
```

Figure 4.31 Interface for index framework

4.4 Testing

Testing is a major consideration in the development and maintenance of B+-tree index. "Testing is the process of executing a program with the intent of finding errors." [Mye1979] and it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results [Het1988]. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well.

Figure 4.32 shows a test pattern. Tests are performed and all outcomes considered, test results are compared with expected results. When erroneous data is identified error is implied, debugging begins. Debugging is performed heavily to find out design defects by programmers. The imperfection of human nature makes it almost impossible to make a moderately complex program correct the first time. Finding the problems and getting them fixed [Kan1988], is the purpose of debugging in programming phase. An "error" that indicates a discrepancy of 0.01 percent between the expected and the actual results can take hours, days or months to identify and correct. It is the uncertainty in debugging that causes testing to be difficult to schedule reliability. The debugging procedure is the most unpredictable element of the testing procedure.

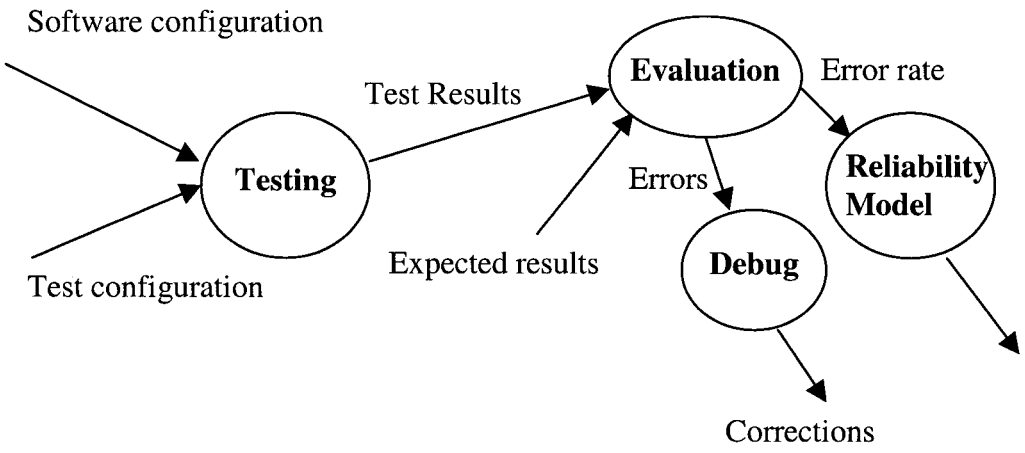

Figure 4.32 testing pattern

Correctness testing and performance testing are two major areas of our testing.

4.4.1 Correctness Testing

Correctness testing is to test whether B+-tree index does what it is supposed to do. Correctness is the minimum requirement of software or system, the essential purpose of testing. Being correct, the minimum requirement of quality, means performing as required under specified circumstances.

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

Black-box testing

The black-box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure [Per1992]. It is also termed data-driven, input/output driven [Mye1979], or requirements-based [Het1988] testing. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing [How1987] which is a testing method emphasized on executing the functions and examination of their input and output data. In this type of test, the tester knows the inputs and what the expected outcomes should be, but not necessarily how the program arrived at them. The test cases for black box testing are normally devised as soon as the program specifications are complete. The test cases are based on equivalence classes. Black-box testing treats the system as a "black-box", so it doesn't explicitly use knowledge of the internal structure and no implementation details of the code are considered.

The research in black-box testing mainly focuses on how to maximize the effectiveness of testing with minimum cost, usually the number of test cases. It is not possible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. In our B+-tree index testing, the lower-level components are first built and tested by using a 'test harness' with the selected values. After that, the higher-level components are tested on the basis of the lower-level components and by using the same way as before.

For the B+-tree index testing, we should mainly focus on testing insert, erase and find operations. We use Tcl for creating test harnesses to test our programs because Tcl

can be easier quickly create a user interface that can run on multiple platforms. In fact, there are many test suites such as Berkeley DB [OBM1999] test suite that is currently available. Berkeley DB test suite is a complete test suite, written in Tcl. It allows users who download and build the software to be sure that it is operating correctly. In this test suite, there are some existing test cases for B-tree index testing. We can take most of them as our test cases to test our B+-tree index. In addition, we also create some special test cases such as illegal inputs, large inputs, and with values smaller or larger than the specified range.

A good testing plan will not only contain black-box testing, but also white-box approaches, and combinations of the two.

White-box testing

Contrary to black-box testing, White-box testing requires the intimate knowledge of program internals, while black box testing is based solely on the knowledge of the system requirements. System is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the system under test are visible to the tester. Testing plans are made according to the details of the system implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White-box testing is also called glass-box testing, logic-driven testing [Mye1979] or design-based testing [Het1988].

In white-box testing, test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may discover unnecessary "dead" code -- code that is of no use, or never get executed at all, which cannot be discovered by functional testing. Control-flow testing, loop testing, and data-flow testing, all maps the corresponding flow structure of the software into a directed graph.

The B+-tree index is implemented by using the STL style. The STL components are meant to be largely independent of each other. They have different types, and a component of one type can connect and work correctly with other component of the same- or different types. The B+-tree index follows the STL style, so we can have unit tests easily. In our white-box tests, we mainly test a function or a code segment is tested individually. The lower-level components such as index page container, leaf page container, smart pointer, cache, and storage are tested independently. That guarantees

these components can work well. For every component, we plan some test cases to try all boundary conditions. For example, a leaf page is a STL-like container, so all the standard built-in functions of the container such as insert, erase, and find operations should be tested on different template parameters and sizes. At the same time, testing the functions for the higher-level components, eg. split, merge, etc. is necessary. After finishing all the tests of the lower-level components, we move to the tests for the higher-level components in the same ways.

4.4.2 Performance Testing

Performance has always been a great concern about Database Index systems. Performance evaluation of an index system usually includes the following:

1. Access Types -- types of access that are supported efficiently, e.g., value-based search or range search.
2. Search Time -- time to find a particular data item or set of items.
3. Insertion Time -- time taken to insert a new data item or set of items (includes time to find the right place to insert).
4. Deletion Time -- time to delete an item or set of items (includes time taken to find item, as well as to update the index structure).
5. Space Overhead -- additional space occupied by an index structure.

The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc. The typical method of doing performance testing is using a benchmark -- a program, workload or trace designed to be representative of the typical system usage. [VW1998]

We did the performance testing on a computer, with the operating system of Solaris 9, a 4G memory, 2 UltraSparc-III+ CPUs and the compiler of GNU g++ 3.2. We used a dataset provided by Gist. The dataset contains 10,000 integers as keys that are random. To do the test, we first set the size of a page to be 8kb that is the size of a block of the testing computer and the buffer (cache) can hold at most 16 pages, and then recompiled Gist v1.0 and our KIA B+ tree index system. Therefore, a page will contain at least 500 keys and at most 1000 keys if Data Reference is treated as an integer. The B+ tree should have at least 21 pages in two levels. The following operations were involved in the test:

1. Insert all the integers of the dataset as keys into the B+ trees
2. Find the first position with key>=20000
3. Delete all the element where key<20000

Table 4.3 shows the test results that are the average time (microseconds) of 10 tests under the same conditions. From the table, we can see our B+ tree index has gotten better performance in the insertion, access, and deletion than Gist B+ tree.

Operation Time	KIA B+ tree index	Gist B+ tree index	Comparison(KIA/Gist)
Insertion Time	260,385.9	626,468.4	0.4
Search Time	25.3	64.5	0.4
Deletion Time	734.4	2,389.1	0.3

Table 4.3 Performance Testing Results using Gist Dataset

To test further, we use a larger dataset that contains 100,000 integers as keys that are generated randomly in the range between 0 and 32767. At the same time, we set a buffer that can hold at most 128 pages. A page has the same size as before, but the B+-tree has at least 201 pages. Table 4.4 shows the test results that the average time (microseconds) of 10 tests under the same conditions when all the operations are performed as before. From the table, we can also see our B+-tree index has gotter much better performance than Gist B+-tree in the insertion and searching, but Gist B+-tree has better deletion performance than that of KIA B+-tree.

Operation Time	KIA B+ tree index	Gist B+ tree index	Comparison(KIA/Gist)
Insertion Time	2,250,000	223,362,386	0.1
Search Time	35	4,095	0.008
Deletion Time	2,270,000	297,000	7

Table 4.4 Performance Testing Results using a larger Dataset

Our B+-tree index can adapt to more access types easily than Gist because it uses a template mechanism but Gist extends its access types by means of inheritance.

In addition, the buffer size is another important factor that affects the performance of B+ trees. Figure 4.33 shows the relationship between the buffer size and Insertion Time when 10,000 integers as keys are inserted. Inserting performance of Gist B+-tree is not affected signifcantly by the buffer size. However, the slowdown of KIA B+ tree at roughtly 16 happens as the buffer size is decreased.

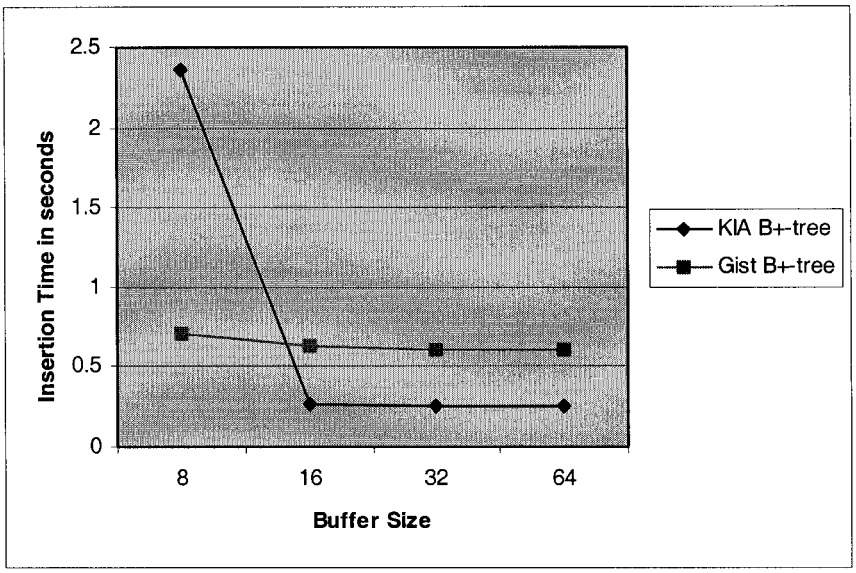

Figure 4.33 Buffer size and Insertion Time

Figure 4.34 shows the relationship between the buffer size and Deletion Time when all the elements where key<=20000 are deleted after the insertion of 10,000 elements. As the buffer size increases, the performance of deletion in both Gist and KIA B+-tree is improved but our B+-tree index can get faster and greater than that of Gist.

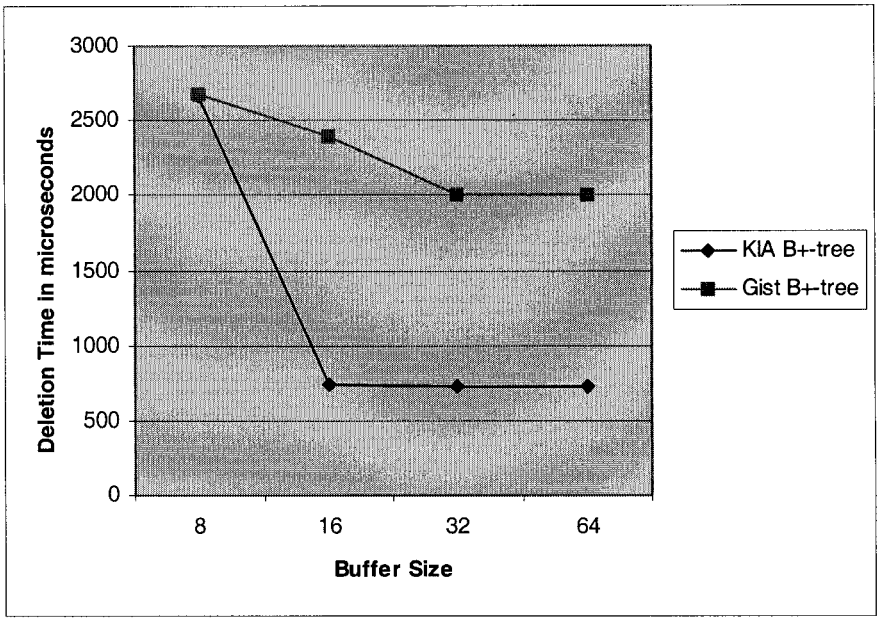

Figure 4.34 Buffer size and Deletion Time

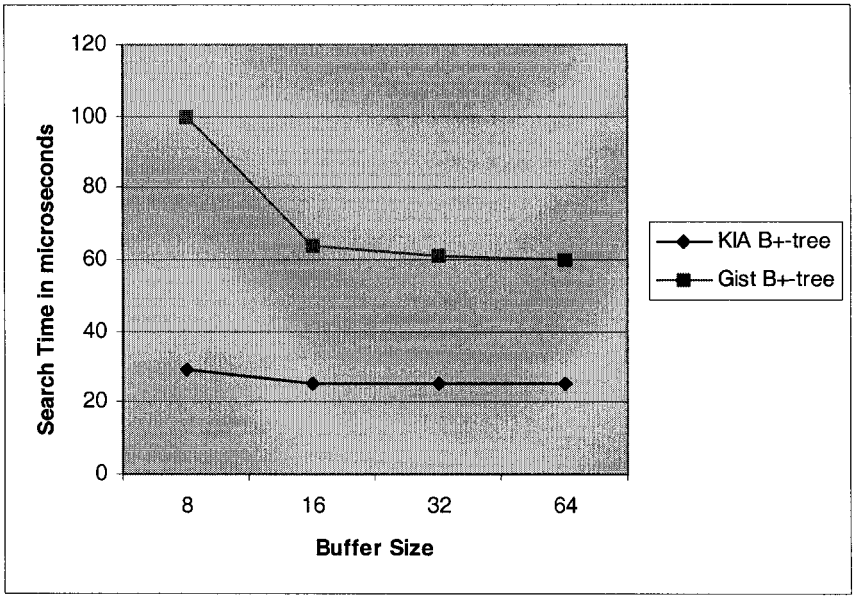

Figure 4.35 Buffer size and Search Time

Figure 4.35 show the relationship between the buffer size and Search Time when a search with key=20,000 is performed after the insertion of 10,000 integers as keys. The searching time of Gist and KIA B+-tree is not greatly influenced by the buffer size.

Testing is also a trade-off between budget, time and quality. To guarantee correctness, all the possible values need to be tested and verified, but complete testing is infeasible. The optimistic stopping rule is to stop testing when either reliability meets the requirement, or the benefit from continuing testing cannot justify the testing cost.

Chapter 5. Conclusion

In this thesis we describe how to build a template for a B+-tree indexing component that can easily handle arbitrary key and data references by using the STL style and design patterns. We investigated several ways to implement our B+-tree index, but we chose the best one among them as our solution to implement the B+-tree index. Our B+-tree implementation is a proof of concept for the Index subframework of the Know-It-All project.

The adoption of STL approach promotes code reusing, increases readability and user friendliness, and reduces time and money overheads incurred during the application development process. Design patterns can simplify the design complexity by separating design concerns at the micro-architecture level, and constitute a reusable base of experience for building reusable software. The combination of the STL style and design patterns makes our B+ tree index more general and reusable. In our design and implementation, we always tried to go with a static model when we can, and relied on a dynamic model when we must in the whole implementation. Therefore, we combined both forms of static and dynamic model in B+-tree index design. Several design patterns such as Composite, Casting method, Proxy, and Singleton were used because they provided a model of how to solve our design issues, many of which dealt with introducing extensibility into the design in order to make it more reusable

Our B+-tree index is designed to be a container built with index pages and leaf pages that are invisible for users. The only way to interact with elements in the B+-tree index container is through its iterator. Because the B+-tree index container conforms to the standard interfaces of STL components, it is easy to be reused. An index page and leaf page are also containers that are small enough to fit in memory. However, the B+-tree index container does not have to fit completely in the memory. Therefore, a copy of the B+-tree structure is maintained on disk. The B+-tree index uses a proxy mechanism (smart pointer) to manage and control accesses to the physical storage. Only the root of the B+-tree is loaded initially, and resides in memory until the B+-tree is destroyed. Through the proxy mechanism, a non-root page is loaded into memory on demand. This improves the efficiency of the tree greatly.

To guarantee correctness and improve quality and reliability of our index system, we paid a great deal of attention to testing in the development and maintenance of B+-tree index.

In the whole development of the index system, we followed the STL style and design patterns, so our B+ tree index should have the following characteristics:

1. Codes are reused easily, readable, and user-friendly.
2. Implementation is based on independent components.
3. Efficiency, flexibility, and extensibility are sustained.

Future Work

Our goal is to build an index system to handle any type of data, key, query or access method in a real database application. One important issue in database is concurrency and locking mechanisms to allow for multiple accesses to the same piece of information while guaranteeing data integrity. To monitor the size and performance of indexes in order to help people design indexes for new applications, an index Mixin may be required. This issues need to be addressed according to the classes suggested in the design.

The design and implementation started with B+-tree indexes and we see that it has a potential to handle other index types such as R-tree index, SS-tree index, and other indexes. These types of indexes deserve further investigations, and we hope to see the design extended to cover them as well.

Bibliography

[Alex2001] Andrei Alexandrescu. **Modern C++ Design: Generic Programming and Design Patterns Applied.** Addison Wesley Professional, 2001.

[Aok98] P. M. Aoki. **Generalizing "Search" in Generalized Search Trees**. Proceedings of the 14th IEEE International Conference on Data Engineering, Orlando, FL, Feb. 1998, pp. 380-389.

[Aus1999] Matthew H. Austern. **Generic Programming and the STL**. Addison-Wesley, 1999.

[BCC+2002] 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.

[BM1972] R.Bayer, E.McCreight **Organization and Maintenance of Large Ordered Indexes.** Acta Informatica, Vol. 1, Fasc. 3, 1972, pp. 173-189.

[CE2000] Krzysztof Czarnecki, Ulrich W. Eisenecker. **Generative Programming: Methods, Tools, and Applications.** Addison-Wesley, 2000.

[Com1979] Douglas Comer. **The Ubiquitous B-Tree.** Computing Surveys, Vol. 11, No. 2, 1979, pp. 121-137.

[DWH1997] Nell Dale, Chip Weems, Mark Headington. **Programming and problem solving with C++ .** Sudbury, Mass. : Jones and Bartlett, 1997.

[FZ1992] Michael J. Folk, Bill Zoellick **File Structures.** Addison Wesley, 1992.

[Gaf2001] Ashraf Gaffar. **Design of a framework for database indexes**. Master Thesis, Department of Computer Science, Concordia University, 2001.

[GDW2000] Hector Garcia-Molina, Jeffrey D.Ullman, Jennifer Widom. **Database system implementation.** Prentice-Hall, 2000

[GOF1994] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software.** Addison-Wesley, 1994.

[Het1988] William C. Hetzel. **The Complete Guide to Software Testing.** Wellesley, 1988.

[HNP1995] Joseph M. Hellerstein, Jeffrey F. Naughton and Avi Pfeffer. **Generalized Search Trees for Database Systems**. Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, September, 1995.

[How1987] William E. Howden. **Functional program Testing and Analysis**. McGraw-Hill, 1987.

[HS1990] Jiandong Huang, John A. Stankovic. **Buffer Management in Real-Time Databases.** COINS technical Report 90-65, University of Massachusetts, 1990.

[Kan1988] Cem Kaner. **Testing Computer Software**. TAB BOOKS Inc. 1988.

[Li1998] Steven Li. **Reengineering a B-tree Implementation Using Design Patterns.** Master Thesis, Department of Computer Science, Concordia University, 1998.

[Mey1992] Scott Meyers. **Effective C++.** Addison-Wesley, 1992.

[Mye1979] Glenford J. Myers. **The art of software testing.** Wiley, New York 1979.

[Nel1995] Mark Nelson. **C++ Programmer's Guide to the Standard Template Library.** IDG Books Worldwide, Foster City, CA, 1995.

[OBM1999] Michael A. Olson, Keith Bostic, and Margo Seltze. **Berkeley DB.** Proceedings of the FREENIX Track:1999 USENIX Annual Technical Conference Monterey, California, USA, 1999

[Per1992] William E. Perry. **A standard for testing application software.** Auerbach Publishers, Boston 1992.

[Sim2000] Volker Simonis. **Chameleon Objects, or how to write a generic type safe wrapper class.** C++ Report Jan. 2000, SIGS Publications.

[SL1995] Alexander Stepanov, Meng Lee. **The Standard Template Library.** Hewlett-Packard Company, Palo Alto, 1995.

[VM2002] David Vandevoorde, Nicolai M.Josuttis. **C++ templates: the complete guide.** Pearson Education, 2002.

[VW1998] Filippos I.Vokolos, Elaine J.Weyuker. **Performance testing of software. systems.** Proceedings of the first international workshop on Software and performance, 1998, Pages 80 - 87