RE-ENGINEERING A B-TREE IMPLEMENTATION USING DESIGN PATTERNS

Steven Li

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

August 1998

0-612-39487-5

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By:	**Steven Li**
Entitled:	**Re-Engineering A B-Tree Implementation Using Design Patterns**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________________ Chair
______________________________ Examiner
______________________________ Examiner
______________________________ Supervisor

Approved ______________________________
Chair of Department or Graduate Program Director

______________ 19 ______ ______________________________
Dr Nabil Esmail, Dean
Faculty of Engineering and Computer Science

Abstract

Re-Engineering A B-Tree Implementation Using Design Patterns

Steven Li

Software design is a difficult creative task learnt from long experience. Reusable object-oriented design aims to describe and classify designs and design fragments so that designers may learn from other peoples' experience. Thus, it provides leverage for the design process.

The evolution of a B-tree indexing system, which is a component in a database management system, is presented to illustrate how design patterns clarify the design and make it more understandable. This re-engineering is a necessary step towards a reusable design and implementation for multi-dimensional indexes. It also demonstrates the reuse of the design knowledge captured in design patterns.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Gregory Butler. His guidance and encouragement make my thesis work a pleasant and extremely educational experience.

I would like to thank several undergraduate students, Piotr Przybylski, Richard Hopkirk, and Khanh Tuan Vu. Piotr Przybylski was responsible for introducing B^+-tree; Richard Hopkirk assisted on the implementation of Pointer; Khanh Tuan Vu through his implementation of data structures for multi-key retrieval kept us striving for flexibility.

I would like to thank my wife YiJia Shen. That is no doubt that without her firm support and encouragement, I would not have made it through this program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Re-Engineering Project

The B-tree and B^+-tree are well-known and well-discussed data structures. A B-tree is a data structure for the organization and maintenance of an index for a dynamic random access file, and was proposed by R. Bayer and E. McCreight in 1971. A B-tree of order k is defined as follows [7]:

- Each path from the root to any leaf has the same length.
- Each node except the root and the leaves has at least k+1 children. The root is either a leaf or has at least two children.
- Each node has at most 2k+1 children.

As a variant of a B-tree, a B^+-tree is introduced to improve sequential access [27]. A B-tree may not perform well in a sequential access environment — it causes storage and access overhead because data is stored in the B-tree as well as all keys. 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. Leaf nodes are linked together left-to-right to facilitate sequential access.

Re-engineering [24, 86] is the examination and alteration of a subject system to reconstitute it in a new form, and the subsequent implementation of the new form. Re-engineering generally includes some form of reverse engineering followed by some form of restructuring.

Reverse engineering is the process of analyzing a subject system to

- identify the components of the subject system, and their interrelationships,
- create representations of the system in another form or at a higher level of abstraction.

Restructuring is the transformation from one representation to another at the same relative abstraction level, while preserving the external behavior of the subject system.

Re-engineering may include modifications with respect to new requirements not met by the original subject system.

The implementation of a B-tree, which is presented by Ladd [53], is analyzed as a starting point of the re-engineering. The re-engineering is to revise the B-tree implementation into a B^+-tree, and focuses on access efficiency, flexibility and extensibility of query processing, and caching techniques. Object-oriented design techniques are adopted to provides solutions to problems in the re-engineering to secure the quality of the improvement.

The aim of the re-engineering project is to adopt new requirements — multiple user environment, multi-dimensional indexes and range query, and to improve reusability by providing more general and abstract interface. This allows the design of B^+-tree can be reused directly or by specialization.

1.2 Reusable Object-Oriented Design

Software reuse [76, 11, 46] is often cited as one of the major goals of object-oriented software design. Object-oriented programming permits the reuse of design as well as programs [46] because it provides modularity and information hiding.

Parnas' writings [68, 69, 70] have lead to the consensus that "design for change" is an overarching principle for software design [37, p.65], and that information hiding is an essential step to achieving design for change. The object-oriented paradigm stresses information hiding through the use of private state variables for objects and loose coupling through message passing [51], so it is well-suited to design for change.

Meyer [62] makes the argument that object-oriented design is better for reuse than functional design. He uses small examples to illustrate the argument that by focussing on data rather than process there are more reusable components: the objects are common across applications whereas functionality is not commonly reused.

The aim of reuse is to improve the software quality and productivity and reduce production and maintenance costs.

The drive for productivity in the software industry is forcing major changes in the ways that software development and maintenance are being done. Traditionally, over 80% of total expenditure is directed to *software maintenance* (including corrective, adaptive, and perfective maintenance), and about 60-70% of the effort of maintenance is directed towards *understanding* the software (requirements, architecture, design, and code). Hence, lowering the effort and costs of maintenance and understanding are the primary means to increased productivity. Software reuse [11, 31, 76, 72] amortizes the cost of developing a component across many projects and supports better quality of products through more effort in inspections and testing.

Reuse applies not only to source-code fragments, but to all the intermediate products generated during software development, including requirements, documents, system specifications, and designs. Systematic reuse of existing software components to construct new systems has been successfully practiced by organizations since at least 1979 [57, 58].

Object-oriented programming is not a panacea [47]. Program components must be designed for reusability. However, design activity is comprehensive and creative, even very expensive.

Design is the activity to produce a description of *how* to perform a task which meets the customers' requirements and any other constraints imposed by the context in which the task is to be performed. That is, software design produces a mechanism to perform the task, and that mechanism is later realized in a programming language. There are many methodologies for designing software [13, 19, 30, 43, 44, 48, 74, 85, 88, 89]. They chiefly address designing a system from scratch rather than design with reuse. Design is an example of a "wicked problem" [19, pp.19-21]. In particular, the requirements for a system may not be fully understood until the design is complete; there is no right answer to the problem, just a vague distinction between good and bad designs; and there is no way to converge to a good design, one postulates a tentative design and by analyzing its merits and deficiencies may refine it to a complete working design.

Experienced programmers reuse design. Reusing proven existing design techniques is the way to get around the design activity, and improve design quality and productivity for some complex design issues, and make the software more robust.

Reusing an existing design still is hard, because existing design notations focus on communicating the *what* of designs, but almost ignore the *why* [9]. We need a way of describing designs that communicate the reasons for the design decisions, not just the results. A contemporary trend of software engineering is to develop tools and techniques to assist design reuse. The major problem with reusing design information is how to capture and express the design.

Software design is a difficult creative task learnt from long experience [19]. Reusable object-oriented design aims to describe and classify designs and design fragments so that designers may learn from other peoples' experience. Thus, it provides leverage for the design process.

The reusable design artifacts are

- software architectures, which describe the structure of systems at the level of gross organization and global control;
- application frameworks, which are actual implementations of those architectural components common to a family of applications, including the design and implementation of global control and the global division of responsibilities;
- design patterns, which are micro-architectures or mechanisms that resolve one issue in design; and
- class libraries, which are collections of classes and incorporate the design of interfaces and class hierarchies.

Every designer can benefit from a knowledge of software architectures and design patterns, and from familiarity with class libraries for domain-independent components such as data structures and user interfaces. However, more systematic reuse of domain-dependent design artifacts brings more benefits.

1.3 Layout of the Thesis

Some background on reusable object-oriented design is presented in Chapter 2. Then the B-tree re-engineering process and the highlighted design details are presented in Chapter 3. Some application of design patterns to the B-tree re-engineering is discussed in Chapter 4 to illustrate how design patterns clarify the design and make it more understandable. Finally, a conclusion is made in Chapter 5.

It is assumed that the reader is familiar with the object-oriented modeling methods, and with the features of the draft ANSI standard C++ programming language. Meanwhile, it is assumed that B-tree and B^+-tree are well-known data structures, so that the algorithms of the operations on B-tree and B^+-tree are not covered in the thesis.

1.4 Contribution of the Thesis

The work described here involves the design and implementation of a suite of C++ classes for a B^+-tree. The design and implementation is original, though it grew out of the work of Ladd [53].

The major concerns are:

- adopting B^+-tree to improve sequential access.
- introducing Page abstract class and hierarchy.
- using Smart Pointer to load a page on demand, and to maintain the reference to the loaded page.
- Cursor to take control of operations on B^+-tree, to iterate over query results.
- FileFactory to control instantiation of B^+-tree, and locking protocols on pages with Cursor.

The re-engineering improves reusability and extensibility of the B^+-tree implementation: for example, extensions for multi-user access, exact match query processing and range query processing.

In the thesis, most of Chapter 2 is the joint material from a survey paper "*Reusable Object-Oriented Design*" written by Dr. G. Butler, Dr. I.A. Tjandra, and Steven Li.

Several undergraduate students have participated in the re-engineering project. Piotr Przybylski was responsible for introducing B^+-tree; Richard Hopkirk assisted on the implementation of Pointer; Khanh Tuan Vu through his implementation of data structures for multi-key retrieval kept us striving for flexibility.

Chapter 2

Background

2.1 Context for Reusable Object-Oriented Design

2.1.1 Reuse

Reuse applies not only to source-code fragments, but to all the intermediate products generated during software development, including requirements, documents, system specifications, and design: indeed any information that the developer needs to create software [11, 31, 49, 76, 72]. The reuse of domain-independent software components for an organization usually involves features common to all application systems. These include common data structures, graphical user interfaces, interfaces to databases, and networks. The components may be purchased as a commercial library, or developed in-house, and then reused in new applications.

It is important to separate the two sides of the reuse process: development of components *for* reuse and development of new applications *with* reuse of existing components. The first side produces reusable components, and the other side consumes them during the development of new applications. The main steps in development *for* reuse are

- to *re-engineer* components from existing systems to make them more general and reusable,
- to *qualify* the components to ensure they are of acceptable quality for the library, and

- to *classify* the components as to purpose and dependencies in order to facilitate subsequent retrieval from the library.

The main steps in development *with* reuse are

- to *retrieve* components from the library according to some need of the application under development,
- to *evaluate* the quality and appropriateness of the candidate components, and
- to *adapt* a component if it cannot be used as-is.

Tracz's 3C Model [82] of reusable software components indicates that at least the following information must be stored about each component, and used in its classification: *concept*, the abstraction captured in a component; *content*, the implementation of that abstraction; and *context*, the environment in which component is designed to operate. The quality factors of a component which affect reusability are the general ones of reliability, testability, and maintainability, and some which are more reuse specific quality factors: flexibility, portability, understandability, and confidence [78]. Confidence differs from reliability in that confidence is an estimate of the probability of error in unforeseen contexts, while reliability estimates the probability of error in a fixed context.

In development *with* reuse, there may be *direct* reuse of a component if it precisely matches the needs of the application. If the component must be adapted, then the following forms of adaptation might be possible:

- reuse by *instantiation*, where arguments to a parameterized component are supplied;
- reuse by *specialization*, where a subclass of a component is derived; or
- reuse by *adaptation*, where the textual source of a component may be freely edited.

Reuse by *composition* of the adapted components is the norm for domain-independent components. A typical example is using pipes to compose commands in Unix shell programs. Reuse in mature domains or organizations may be reuse by *generation*, where the component is generated automatically from a description of the desired solution.

The description of the solution may use a script language, such as a fourth-generation language (4GL) for relational databases, or a visual metaphor as in graphical user interface builders.

2.1.2 Scope of Reuse

Three different scopes of reuse have been identified [73]:

- *General reuse* is independent of the domain of application. It almost entirely reuses components such as common data structures and graphical user interfaces, that are found in most applications.
- *Reuse within an application domain* restricts its focus to a single domain, such as insurance, or missile guidance. It is important to specify the common aspects of application systems for this domain, and to also specify the range of variability found in the domain. This is one role of *domain analysis*.
- *Reuse within a product family* focuses on one product line within an application domain. A generic architecture is developed for this product family, so that there is a well-defined role for each component.

Reuse could be carried out within an organization, or be inter-organizational through the marketing of reusable components. Reuse within an organization has been profitable at each of the three scopes. Reuse is most successful when it is systematic and focuses on a narrow, well-understood application domain, or on a product family [31, 38].

2.1.3 Domain Analysis

Reuse which is specific to an application domain relies on an analysis of the domain to discover the common aspects of application systems for this domain, and to also discover the range of variability found in the domain. Domain analysis is

> "a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems." [81]

The domain is analyzed from the user's and customer's perspective (as in normal systems analysis), and also analyzed from the developer's perspective to discover common software components, interfaces, and architectures [4, 81]. The sources of domain knowledge are the technical literature of the domain, requirements documents for current application software, surveys of customer needs, the architecture and components of existing systems, and domain experts. The results of domain analysis may include a taxonomy of the concepts in the domain and their relationships, which may be used to aid classification of components; standards concerning data formats, interfaces, etc; models of system architectures or components; and languages for describing domain entities. The last two outputs play significant roles in product-specific reuse.

The domain analysis identifies domain concepts, constructs, and code. Domain concepts may be reused during analysis, as the concepts capture the user- and customer-level perspective of the domain. Domain constructs are design artifacts such as architectural descriptions, abstract classes, or module interfaces, some of which are traceable to user-level concepts. The domain constructs may be reused during design. Domain code is actual code extracted from existing implementations, or specifically developed for reuse. Source code is the implementation of a module, or class, and may be reused during the implementation phase. Each concept might correspond to several constructs, and each construct might correspond to several code fragments, which implement the construct or concept.

Domain analysis may identify common architectural features, and may develop a language to describe the variable aspects of the product family. A generic architecture is developed for this product family, so that there is a well-defined role for each component. The architecture, and its realization, is variously called a *reference architecture*, a *domain specific software architecture* (DSSA), or an *application framework* [17, 87]. The role of a component within the framework may also form part of the domain taxonomy and be used in classification and retrieval.

For mature architectures and product families, one can build an *application generator* [25]. This is reuse by generation, where the components or system are generated automatically from a description of the desired solution. The description of the solution may use a script language, such as a fourth-generation language (4GL) for relational databases, or a visual metaphor as in graphical user interface builders.

Not all domains lend themselves to reuse. For example, in a domain where difficult real-time constraints are very important, these performance considerations may always demand monolithic applications. Hence there is no possibility of separating out reusable components.

2.1.4 Design

Design [19] is the activity to produce a description of *how* to perform a task in such a way that it satisfies the customers' requirements and also satisfies any constraints imposed by the environment in which the system is to be used. Design methodologies [19, 30, 48] typically have three principal components:

- a *representation* of the design using one or more notations;
- a *process* for developing or transforming the representation; and
- a set of *heuristics* maybe guiding the order of steps, or the selection of issues that are to be addressed.

Heuristics are necessary since there is no pre-determined way to converge to a good design. Typically one postulates a tentative design and, by analyzing its merits and deficiencies, the design is refined to a complete working design. Generally, an architectural design is developed before a detailed design. The architectural design describes the main subsystems and modules, their responsibilities, the division of control, and the interface or protocol of each module. The detailed design describes the internal mechanism of each module.

Design is an example of a "wicked problem" [19, p.19-21]. Often, when pursuing the resolution of an issue, the designer discovers other related issues of which the designer was not aware. The problem becomes more complicated as you approach a solution. Thus, the requirements for a system may not be fully understood until the design is complete. Design processes may be modelled [71] as an exploration of issues, the arguments for and against a position on an issue, and the design steps taken because a position has been adopted. Beck and Johnson [9] use this approach to describe an architecture by the issues encountered and the design patterns chosen to address each issue.

The *representation* or *model* of the design forms the basis of communication, analysis, and documentation. A model and the views of a model are described using notations. Budgen [19] classifies design notations into three categories:

- *diagrammatic* notations based on a visual graphical representation;
- *textual* notations written in pseudo-code or natural language; and
- *mathematical* notations based on logic, algebra, set theory, or other mathematical notations.

There are generally multiple views of a system. Each view focuses on a single aspect of the system in an attempt to reduce complexity. Object-oriented design methodologies, for example, commonly use the following viewpoints or perspectives [13, 19, 74]:

- *structural* viewpoints of the static aspects of a system, which may include the organization of subsystems and modules at the architectural level;
- *functional* viewpoints which seek to describe what the system does in terms of its tasks;
- *behavioral* viewpoints describing the dynamic or causal nature of events and system responses during execution; and
- *data-modeling* schemas, or class diagrams, concerned with the data used within the system and the relationships between these.

The notations used for these viewpoints also support *decomposition* or *hierarchies* to further assist humans to control the complexity of even a single aspect of a system. For example, a developer might use nested modules to describe the structural view of a system; hierarchical data flow diagrams to describe functionality; statecharts, which provide nesting, abstraction, and decomposition, to model behavior; and a class inheritance hierarchy to model data [19]. Each viewpoint should be derived from a consistent underlying model. Harel [40] is optimistic that developers can exercise greater intellectual control over complexity by using models throughout the development process. Harel is a supporter of multiple models, which have both a visual representation and a solid semantic foundation. A visual model aids human cognitive skills; a semantic foundation allows analysis of the models, such as providing answers to "what-if" questions.

2.2 Software Architectures

Software architectures provide an abstract description of the organizational and structural decisions that are evident in a software system.

> "Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.
> This is the *software architecture* level of design." [36]

One important aspect of this area is compiling a *catalogue* of existing software architectures. The primary reference is the work of Mary Shaw and her colleagues [36, 77, 5]. Their initial list of architectural styles [36] follows.

- *Pipe-and-Filters* architectures, like that supported by the Unix shell, connect filters in a linear fashion. Each filter has one stream of inputs and one stream of outputs.

- *Data Abstraction* and *Object-Oriented Organization* architectures promote the decomposition of the system into entities (data type variables or objects) that encapsulate their implementation details and present an interface that completely describes their behaviour or functionality.

- *Event-based, Implicit Invocation* architectures are based on components that register an interest in (a class of) events. The components are invoked in response to the occurrence of an event implicitly rather than being called directly by another component.

- *Layered System* architectures are organized as a hierarchy of layers, each layer providing services to the layers above it and each layer being a client for the services provided by layers below it.

- *Repository* architectures have distinct components for a central data store (the repository) and those components which operate upon the repository.

- *Table Driven Interpreter* architectures implement a software virtual machine (the interpreter) by separating the interpretation engine from a table that describes the machine behaviour.

- *Heterogeneous* architectures combine architectural styles.

The development of a catalogue of architectures would be greatly assisted if corporations released details of their system architectures: information which is now often proprietary.

The ultimate aim of their research is a *taxonomy* of architectures to organize our knowledge by describing common and distinguishing features of the architectures. This may help designers to appreciate the breadth of choices and trade-offs, and may guide the discovery or invention of new artifacts. It may also assist the development of design notations and languages, which could be incorporated in development environments specific to individual architectural styles [35].

The book on OMT [74] also contains very useful sections describing architectures and their use in system design. There is good general agreement of their list below with the above list of architectures.

- *Batch Transformation* architectures transform the entire input data set once.

- *Continuous Transformation* architectures transform the input data continuously in response to incremental changes in the input.

- *Interactive Interface* architectures are dominated by external interactions.

- *Dynamic Simulation* architectures simulate evolving real-world objects.

- *Real-time System* architectures are dominated by strict timing constraints.

- *Transaction Manager* architectures process transactions on data stores that are being accessed in a concurrent and distributed fashion.

- *Hybrid systems* combine architectural styles.

Buschmann et al [20] describe architectural patterns, which provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them. The following is the list of those architectural patterns:

- *Layers* architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

- *Pipes and Filters* architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

- *Blackboard* architectural pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

- *Broker* architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

- *Model-View-Controller* architectural pattern (MVC) divides an interactive application into three components. The Model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

- *Presentation-Abstraction-Control* architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

- *Microkernel* architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

- *Reflection* architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about the selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

2.2.1 Reusing Software Architectures

Rumbaugh et al [74] provide a methodology for applying an architecture that is based on

- *characterizing* the kinds of systems to which the architecture is applicable;
- presenting important *design principles* which must be observed when applying the architecture; and
- providing a sequence of *detailed steps* to follow when developing a design based on the architecture.

Another approach to utilizing architectural styles is to encode a description of the components and constraints of an architecture into an software development environment. Thus the environment forces the designer to conform to the specified style. Garlan et al [1, 35] describe an architecture in terms of its components, connectors, configurations, ports, and roles. Hence a description of an architectural style provides

- a *vocabulary* of the basic design elements (components and connector types),
- a set of *configuration rules* which constrain how components and connectors may be configured,

- a *semantic interpretation* which defines when suitably configured designs have a well-defined meaning as an architecture, and
- *analyses* that may be performed on well-defined designs.

From such a description, their system, Aesop, can generate a development environment that is tailored to the given architectural style.

2.3 Application Frameworks

For a given application domain, the process of domain analysis can draw from existing software applications to identify common architectural features, and to describe their variable aspects. A generic architecture may be developed for a product family or application domain, so that there is a well-defined role for each component. A realization of the architecture is an *application framework*:

> "a collection of abstract classes, and their associated algorithms, constitute a kind of *framework* into which particular applications can insert their own specialized code by constructing concrete subclasses that work together. The framework consists of the abstract classes, the operations they implement, and the expectations placed on the concrete subclasses" [29, p.92]

> "A framework is an abstract design for a particular kind of application, and usually consists of a number of classes. These classes can be taken from a class library, or can be application-specific." [47, p.25]

A framework is usually designed by experts in a particular domain and it is used by non-experts. It allows the user to reuse abstract designs, and pre-fabricated components in order to develop a system in the domain. A user may also customize existing components by subclassing. The design of the framework incorporates decisions about the distribution of control and responsibility, the protocols followed by components when communicating, and implementations for each of the major algorithms. Often the implementations are template methods that embody the overall structure of a computation and that call user's classes to perform sub-steps of the algorithm. Default implementations of each user class may be provided, and the user will subclass in order to override or specialize the operation which implements the sub-step.

A good characterization of the relationship between a framework and user's classes is "Don't call us, we'll call you." So the classes defined in the framework call the user's code, whereas the traditional use of class libraries is for the user's code to call the library classes.

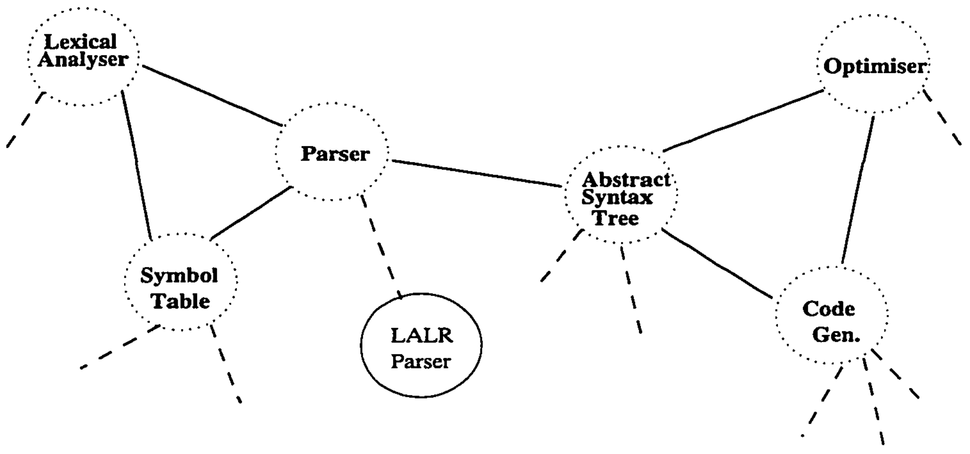


Figure 1: Compiler Framework

For example, a framework for program translation or compilation would provide an abstract class for each of the roles identified: Lexical Analyzer, Parser, Symbol Table, Abstract Syntax Tree, Optimizer and Code Generator shown in Figure 1. Furthermore, there would be a class or main program orchestrating the overall communication and control of the compilation process. Concrete subclasses, such as LALR Parser would implement specific algorithms and/or representations for a role. A reuser of the framework would select a concrete subclass for each role, if a suitable one existed in the associated class library, or create their own subclass (usually by specializing an existing one. Once the selected configuration is compiled, a new instance of the application is ready for use.

Early examples of application frameworks were for *graphical user interfaces* (GUI), including MACAPP [3], and INTERVIEWS [21]. There is now an abundance of GUI frameworks that have been used successfully on many platforms ranging from DOS to UNIX, such as MACAPP for MacIntosh, OWL-WINDOWS for DOS/WINDOWS, and MOTIF for UNIX. Frameworks now exist for a broad range of application domains such as ET++, an *editor* toolkit [61] which has recently been used in MET++ which is a framework for *multimedia* applications; RTL framework [56] for code optimization in compilers; CHOICES for object-oriented operating systems [22]; BEE++ for analyzing and monitoring distributed programs [18]; and others for network management and telecommunications [10], and financial engineering [12]. Don Batory[6] has developed the GENESIS framework for relational database systems, where a database is a composition of functional layers (or realms), and the framework consists of the realms, their type constraints as functions, and the alternative implementations.

2.3.1 Developing a Framework

An application framework evolves in response to feedback from reusers. An initial framework is based on past experience or by careful construction of one or two applications, keeping in mind the need for flexibility, reusability and clarity of concepts. Each consequent reuse points out shortfalls in these qualities in the existing framework as one stretches the architecture to accommodate the new application. By addressing the issues raised, the framework evolves, gaining flexibility, coverage of domain concepts, and clarity of the concepts and the dimensions along which they vary.

The major steps in developing an application framework can be summarized as follows [47, 80]:

- Identify and analyze the application domain and identify the framework. If the application domain is large, it should be decomposed into a set of possible frameworks that can be used to build a solution. Analyze existing software solutions to identify their commonalities and the differences.

- Identify the primary abstractions. Clarify the role and responsibility of each abstraction. Design the main communication protocols between the primary abstractions. Document them clearly and precisely.

- Design how a user interacts with the framework. Provide concrete examples of the user interaction, and provide a main program illustrating how the abstract classes are related to each other and to the classes for user interaction.

- Implement, test, and maintain the design.

- Iterate with new applications in the same domain.

The design and implementation of frameworks relies heavily on abstract classes, polymorphism (both parametric and inclusion polymorphism), and inheritance.

2.3.2 Describing Frameworks

Only a small amount of work[41, 42, 45] has been done on documenting, specifying, and reasoning about frameworks. The frameworks under consideration are chosen from toolkits for user interfaces and drawing programs. Only the Contracts paper[41] considers verifying correctness, but the authors offer no evidence of actual reuse which has benefited from their contracts. On the other hand, patterns[45] have been an important aspect of much actual reuse. There the emphasis is on documentation rather than specification, and certainly there is no concern for verification of correctness.

The documentation of a framework is very important to both the re-user of the framework, and to the developer/maintainer of the framework. These two audiences have different requirements:

- *Documentation for reuse* illustrates how to customize the framework, often through examples, for typical reuses. It discusses which classes to subclass, and which methods to override, and whether combinations of classes and methods need to be specialized in unison to maintain a protocol of collaboration amongst the classes. It is prescriptive.

  Johnson [45] introduced an informal *pattern language* that can be used for documenting a framework in a natural language. The documentation of a framework consists of a set of patterns where each pattern describes a problem that occurs repeatedly in the problem domain of the framework, and also describes how to solve that problem. Each pattern possesses the same format. The elements of a pattern are: description of its purposes, explanation of how to use it, description of its design, and some examples.

- *Specification for reuse* is generally more descriptive than prescriptive: the re-user is left to figure out the implications of the specification in terms of the desired customization. The main concerns are to clearly specify the obligations on a user-defined subclass, any protocols which the user can customize, and the collaborations amongst classes that must be supported by the user-defined subclasses.

- *Documentation and specification for general understanding* primarily assists the evolution of the framework. Traditional techniques for modules, such as the Larch family of interface languages [39], can be used for describing class interfaces and extended to include the obligations on subclasses [23]. The EIFFEL language [63] supports design-by-contract through the declaration of assertions, preconditions, and postconditions.

Contracts [41, 42] are a high-level (abstract) construct for explicitly specifying behavioral composition, the obligations on participating objects, and interactions among groups of objects. A contract specifies a set of communicating participants and their contractual obligations which extend the signature of types and functions used in a class to include constraints on behavior that capture the behavioral dependencies between objects. A contract specifies preconditions on participants required to establish the contract, and the invariant to be maintained by these participants.

Two constructs for modifying a contract are provided: *refinement* and *inclusion*. Refinement and inclusion are used for deriving a class based on particular subclasses. This concerns the aspect of inclusion polymorphism that is necessary in building an application framework.

Contracts are refined by either specializing the type of a participant, extending its actions, or deriving a new invariant which implies the old. Consequently, the refinement of a contract specifies a more specialized behavioral composition.

The behavior of a subset of the participants in a large composition can be specified by means of simpler *sub-contracts*. These sub-contracts can simply be included in a large contract. The invariant clauses of a contract with sub-contracts are formed from the union of the corresponding statements from the sub-contracts.

2.3.3 Application Generators

For mature architectures and product families, one can build an *application generator* [25]. This is reuse by generation, where the components or system are generated automatically from a description of the desired solution. The description of the solution may use a script language, such as a fourth-generation language (4GL) for relational databases, or a visual metaphor as in graphical user interface builders.

> "toolkit is a collection of high level tools that allow a user to interact with an application framework to configure and construct new applications." [47, p.26]

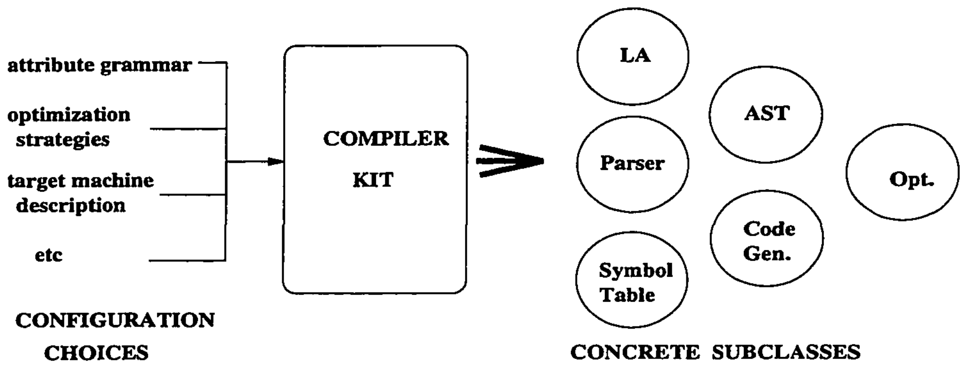


Figure 2: Toolkit for the Compiler Framework

Figure 2 shows possible inputs and outputs — the concrete subclasses for the compiler framework — of a compiler generator.

2.4 Design Patterns

A pattern is a solution to a problem in a context [26, p.93]. A pattern is an abstract model of the problem and its solution. So the pattern expresses the relationship among the elements in a problem, and describes the context: it lets the designer focus on the abstract level (building block) when thinking about the solution rather than focusing on the low concrete level (bricks) [2, 59]. The pattern mechanism also describes quality, impact and alternatives.

> "Design patterns identify, name, and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design. They identify classes, instances, their roles, collaborations, and the distribution of responsibilities." [33, p.407]

> "Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [34, p.3].

Design patterns capture design experience at the micro-architecture level, by specifying the relationship between classes and objects involved in a particular design problem. By documenting, cataloging and classifying this experience, design patterns allow it to be reused. Since design patterns only record proven reusable design techniques, they offer improved reusability and reliability for developers of new systems. Furthermore, design patterns can improve the documentation and maintenance of existing systems through the reuse of the explicit specification of class and object interactions and their underlying intent [34]. So design patterns constitute a reusable base of experience for building reusable software.

Design patterns tend to be independent of the application domain since they are smaller and less specialized than frameworks and architectures. They are abstract since they describe a pattern (of structure or collaboration) and not a concrete instance of the pattern. While concrete examples might form part of the documentation of a design pattern, the cataloguer has already abstracted a higher level description from these examples.

Design patterns provide a common vocabulary for designers to communicate, document, and explore design alternatives. The vocabulary of patterns — their names, the roles played by participants, and the names of collaborations — enrich the vocabulary of object-oriented designers forming a part of their language when communicating with colleagues and when presenting their work to others.

A design pattern provides an abstraction above the level of classes and objects. This allows a designer to work at a more abstract level and to reduce system complexity. Each design pattern identifies a design problem, constraints, solution to that problem and other alternatives, meanwhile it encapsulates a well-defined problem/solution [59, p.42] as a building block for constructing more complex designs.

Design patterns help reduce the required learning time for a library when the library uses the common vocabulary of roles in a design pattern. The library user is then familiar with the terminology in the documentation and knows from the roles how the class should be reused.

A catalogue of design patterns offers examples of good design, and provides proven solutions to design problems in a well-organized form. This can help a novice learn design skills more quickly and to perform more like an expert.

Design patterns provide a target for the reorganization or refactoring of class hierarchies.

2.4.1 Documenting Design Patterns

The purpose of design patterns is to reuse design. Clearly the problem, the context, and the solution are documented. However, besides the description of the structural aspects of a pattern, it is necessary to also record the information pertinent to the critical issues considered during design work, such as design decisions, alternatives and trade-offs. In general, a design pattern has four essential elements:

- Name(s) of Pattern — to identify a design problem, and to produce terminology for thinking and communicating.
- Problem Explanation — to describe a set of pre-conditions which should be met to apply the pattern.
- Solution Description — to describe the relationships, responsibilities and collaborations among the elements involved in the pattern.
- Consequences Analysis — to present the results and trade-offs of applying the pattern.

Typically this information is provided in a template, though some aspects of Contracts [41] for documenting frameworks actually document patterns: the connection is elaborated in work of Lajoie [54, 55].

The "standard" way to document a design pattern is the template [34, p.6] shown in Figure 3. A template provides a consistent format for documentation which makes design patterns easier to learn, compare and use.

Design patterns vary in their granularity and level of abstraction, as well as in what they actually do. Design patterns are **classified** by two criteria [34, Table 1.1, p.10]: **purpose** which can be *creational*, *structural*, or *behavioral*; and **scope** which can be *class*, *object* or *compound*. Purpose indicates the outcome of applying the pattern: Does it organize how classes or objects are created? Does it provide a useful structure amongst classes or objects? Does it provide a useful dynamic behavior? Scope indicates the granularity; that is, whether the pattern mainly works at the level of a class, an object, or a number of objects.

These are early days for design patterns, and much work is being done on ways of documenting and classifying patterns [83].

Design Pattern Name and Classification
The name should convey the pattern's essence succinctly. A good name is vital, as it will become part of the design vocabulary.

Intent
What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As
Other well-known names for the pattern, if any.

Motivation
A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help the reader understand the more abstract description of the pattern that follows.

Applicability
What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can one recognize these situations?

Structure
A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [74]. The interaction diagrams [13] is applied to illustrate sequences of requests and collaborations between objects.

Participants
Describe the classes and/or objects participating in the design pattern and their responsibilities.

Collaborations
Describe how the participants collaborate to carry out their responsibilities.

Consequences
How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What does the design pattern objectify? What aspect of system structure does it allow to be varied independently?

Implementation
What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there language-specific issues?

Sample Code
Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses
This section presents examples from real systems. We try to include at least two examples from different domains.

Related Patterns
What design patterns have closely related intent? What are the important differences? With which other patterns should this one be used?

Figure 3: Documentation Template for Design Patterns

2.4.2 Reusing Design Patterns

Design patterns support issue-based design by providing a range of solutions for certain issues that commonly arise in object-oriented design. Generally these are issues of providing flexibility so that some aspect of the design can change easily. Knowing what varies, one can access related design patterns [34, Table 1.2, p.30]. It is then a matter of understanding the abstraction of the design pattern and matching components of the actual design to the abstract participants of the pattern.

Beck and Johnson [9] illustrate the use of design patterns in developing the design for HOTDRAW, a framework for drawing editors. This is actual a re-documentation of an existing design [45], but it very clearly shows the nature of issue-based design.

In [34, pp.33–77], a case study is presented in the design of a "What-You-See-Is-What-You-Get" document editor called LEXI. The document can mix text and graphics freely in variety of formatting styles, with pull-down menus and scroll bars, and a collection of page icons for jumping to a particular page in the document. Eight design patterns are illustrated in the case study.

The use of a set of design patterns is facilitated by organising them into a *pattern language* [28, 84, 60] which provides an incremental approach to issue-driven design. The set of design issues is arranged in an appropriate order, and a pattern (or patterns) is described to address each issue.

2.5 Class Libraries

There have been many class libraries developed, and their design has been extensively documented in the literature [47, 52, 64, 65, 79].

Object-oriented class libraries embody

- the design of interfaces,
- the design of a class hierarchy, and
- the design of policies or conventions, such as those on reporting errors in calls to library classes.

The most effective libraries are domain-specific and often complement an application framework by providing a choice of concrete classes to instantiate the abstract classes in the framework [47, p.25]. Korson and McGregor [52] list the following necessary attributes of a reusable object-oriented class library:

- *complete* general model for each concept covered by the library;
- *consistent* definitions and naming throughout the library;
- *easy-to-learn* because of the organization of the documentation, design, and implementation;
- *easy-to-use* because the necessary classes are easy to find, understand, and use;
- *efficient* implementations of each algorithm, and the selection of appropriate algorithms;
- *extendable* classes via specialization or composition;
- *integrable* with other libraries;
- *intuitive* to those familiar with the standard practice for the domain covered by the library;
- *robust* classes that deal reasonably with erroneous arguments to calls and with erroneous sequences of calls; and
- *support* from the vendor.

The rules of Johnson and Foote [47] provide some guidance as to the evolution of a reusable class for a library, and Meyer [65] illustrates clarity of concepts, the precise style of Eiffel interface contracts, and just how few arguments the methods in a reusable class should have. The design of reusable class libraries is an iterative process requiring a clear vision of the domain of the library and also requiring much feedback from actual reusers about the classes, their interfaces, and the documentation.

Object-oriented languages generally have a standard library providing basic mathematical objects, I/O files and streams, and utility classes such as String.

Class libraries for common data structures are specific to the domain of data structures, and are the result of a long history of studying abstract data types and collection classes. The development of the Booch Components, first in Ada [14] then in C++ [15, 16], clearly uses a domain analysis.

The Standard Template Library (STL) provides a set of well structured generic C++ components that work together in a seamless way. The STL contains five main kinds of components:

- *algorithm* defines a computational procedure.
- *container* manages a set of memory locations.
- *iterator* provides a means for an algorithm to traverse through a container.
- *function object* encapsulates a function in an object for use by other components.
- *adaptor* adapts a component to provide a different interface.

Utility class libraries for persistence, threads, and distributed computing follow a set of conventions, as do classes for I/O files and streams.

User interface libraries complement the corresponding application framework and must be consistent with the conventions and division of responsibility and control of that framework.

Chapter 3

The Re-Engineering

This chapter describes the actual re-engineering of the design and code. The purpose of the improvement is to achieve file access efficiency, and to make the implementation more reusable and understandable.

The initial step of the re-engineering activity is to understand the design of Ladd's implementation, and then an original model is deduced as shown in Figure 4. Some issues arose while the original model was being analyzed. Appropriate solutions are adopted using object-oriented techniques.

The second step focuses on sequential access efficiency, and the adoption of the B^+-tree.

A further step addresses the required flexibility and extensibility of query processing and caching techniques for the multi-dimensional index data structures. The main decisions are the adoption of smart pointers, and cursors [66, 67], as well as overloading the indexing operator. The current version is shown in Figure 5.

Inheritance and polymorphism [47] are emphasized in the re-engineering. Essentially, polymorphism introduces a case-free programming style and leads to software design that is much easier to reuse [47]. Inheritance promotes code reuse through programming-by-difference. Also, inheritance encourages the development of the standard protocols to make polymorphism more useful, and allows extensions to be made to a class while leaving the original code intact [47].

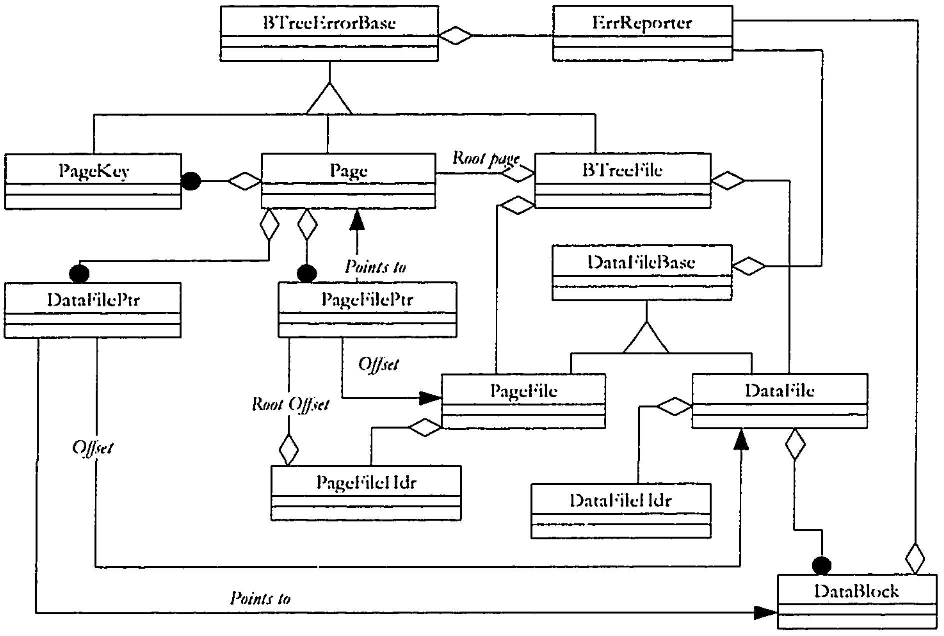


Figure 4: Model of Ladd's B-tree Implementation

3.1 The Original Implementation

As a starting point, the implementation of a B-tree data structure by Ladd [53] is analyzed, and the design of the implementation is deduced as shown in Figure 4.

ErrReporter is the basic error handling class. All classes contain an ErrReporter attribute to deal with errors. Each Page consists of 2k+1 child PageFilePtrs and 2k+1 data DataFilePtrs, and 2k PageKeys, where k is the order of the B-tree. The PageFilePtr represents the offset of a Page in a PageFile, and DataFilePtr represents the offset of a DataBlock in a DataFile. Also, each Page has a PageFilePtr to its parent. Each page accesses its child pages or its parent by the given PageFilePtr, which is the offset in the PageFile. A page does not have any links to other pages in memory.

BTreeFile actually is an abstract interface of the model, and provides the interface to perform operations on the B-tree, such as `Write`, `Read`, `Delete`. The BTreeFile consists of PageFile and DataFile.

`Search` is declared as a private function of the BTreeFile, and is used by other operations — `Write`, `Read`, `Delete`.

```
Boolean Search(const   Page & pg,
                   const PageKey & searchkey,
                          Page & keypage,
                        size_t & pos);
```

The Search starts from the given page pg. Normally, the Search starts from the root of B-tree. The Search compares each key in the page pg with the given searchkey. If the searchkey is found, the found page is returned as keypage. When the searchkey is greater than the key at a particular position in the page pg, the Search reads the child page at the position, and continues recursively.

Write is declared as a public member function of the BTreeFile.

```
Boolean Write(const PageKey & key,
              const DataBlock & db);
```

The function writes the given data to the DataFile, and then inserts the given key and the offset of the data in the DataFile into a particular page. Thus insertion would cause a re-organization of the B-tree if the particular page overflowed.

Read and Delete are declared as public member functions of the BTreeFile.

```
DataBlock Read(const PageKey & key);
Boolean Delete(const PageKey & key);
```

The Read and Delete functions initiate a Search for a page which contains the given key. The Search starts from the root of the B-tree. The Search returns the page in which the given key is found, and the position in the page where the given key resides. Given the position, Read function finds the offset of the data block in the DataFile, and reads the data block. Delete function finds the offset of the data block in the DataFile, and deletes the data block from the DataFile. Then the Delete function replaces the key and the offset of the data block with those of immediate successor. Refer to [53] for the details about the algorithms of Write, Read and Delete.

3.2 The Issues Encountered

This section lists the issues encountered during the re-engineering process.

Issue 1 Information hiding and encapsulation are violated in the original implementation since Page is declared as a `struct`, which is a class without any access restrictions. It may cause high-coupling with other classes, and becomes hard to maintain.

A Page class is created by tracking the uses of the page data structure, and deducing the basic operations from the code fragments as its interface, so as to encapsulate and hide the page data structure.

Issue 2 "Unfortunately, a B-tree may not do well in a sequential processing environment" [27].

To improve the sequential access, a variant of the B-tree, the B^+-tree, is adopted.

Issue 3 The demands of the problem domain, for features such as range queries as well as exact matches, are not fully provided in the interface. Similarly, the facility to iterate over a set of solutions to a query is not provided.

The indexing operator `[]` is overloaded so that the argument type distinguishes between the various kinds of queries. The basic exact match query has a single key as the argument. The operator returns a cursor, which is the mechanism usually used in database systems to traverse a solution set or table of records.

Issue 4 The access of pages on disk is not efficient. A page only consists of the offsets of its child pages and its parent. The offset is the address of the page in a PageFile.

Only the root of a B-tree is loaded at initial time, and resides in memory until the B-tree is destroyed. The life span of a page, except the root, in memory is fairly short, as it is loaded only on demand during the course of an operation. Each loaded page does not contain pointers to the memory location of any parent or child pages that may be in memory at the time: this information is passed as procedure arguments.

Each access to a non-root page causes a page loading no matter whether the page has been loaded into memory or not, because there is no reference maintained to a loaded page, nor a mechanism to know if the page has been loaded or not. This limits the use of caching techniques.

Smart pointers to pages are adopted. A smart pointer is an application of *Proxy* pattern. It is a replacement for a bare pointer that performs additional actions when an object is accessed. The smart pointer is responsible for loading a page into memory when it is first referenced. The smart pointer maintains a reference to the loaded page. For any further access to the loaded page, the smart pointer invokes a simple memory access.

Issue 5 The search and iteration operations on a page require a common abstract interface that is independent of the data structure used to store the keys in the page. The interface should be able to accommodate the planned extension to multi-dimensional indexes. The indexing operator [] is overloaded, and cursors are applied.

Issue 6 The creation and modification of multiple instances of the B-tree files must be controlled.

A *Singleton* FileFactory is adopted to act as a name server and to control the access to the files on disk.

3.3 The Current Implementation

B^+-tree, a variant of B-tree, is adopted to improve the sequential access [27]. The design is shown in Figure 5 and Figure 7, where Page is modeled as an abstract class. IndexPage and LeafPage are derived classes from Page to present different behaviors. Respectively, PageFile is modeled as an abstract class, and IndexPageFile and LeafPageFile are derived classes from PageFile.

An IndexPage contains 2k PageKeys and 2k+1 Pointers referring to its child pages, and a LeafPage maintains 2k PageKeys and 2k DataBlobs, where k is the order of the B^+-tree. Two sibling Pointers are implemented to link the left and right siblings of a LeafPage, in order to support sequential access.

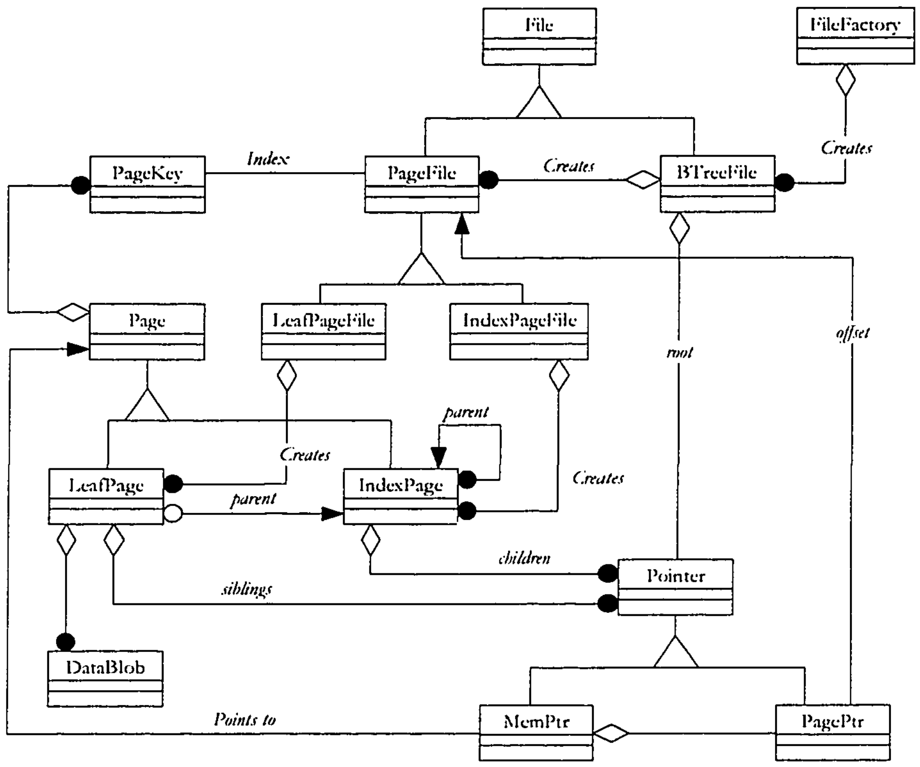


Figure 5: Current B$^+$-Tree Model

Inheritance leads to Page being an abstract class, and provides a potential use of polymorphism. "Polymorphism caused by late-binding of procedure calls and inheritance. Polymorphism leads to the idea of using the set of messages that an object understands as its type, and inheritance leads to the idea of an abstract class " [47]. It becomes feasible to reduce the code complexity and size by reusing code and getting rid of `if, case, switch` statements. It makes the implementation flexible for further maintenance and reuse.

IndexPage and LeafPage have different algorithms to perform operations, such as the `search`, `insert`, `delete`, and `update`, and have different functionality as well. IndexPage deals with rapid random location of the index and key parts. LeafPage handles sequential access from leftmost to rightmost. The interfaces of each object are minimized by separating tasks or behaviors of each object so that reuse can be achieved [34].

Pointer is an implementation of the *Smart Pointer* design pattern, and is modeled as an abstract class. A Pointer is responsible for loading a page when it is first

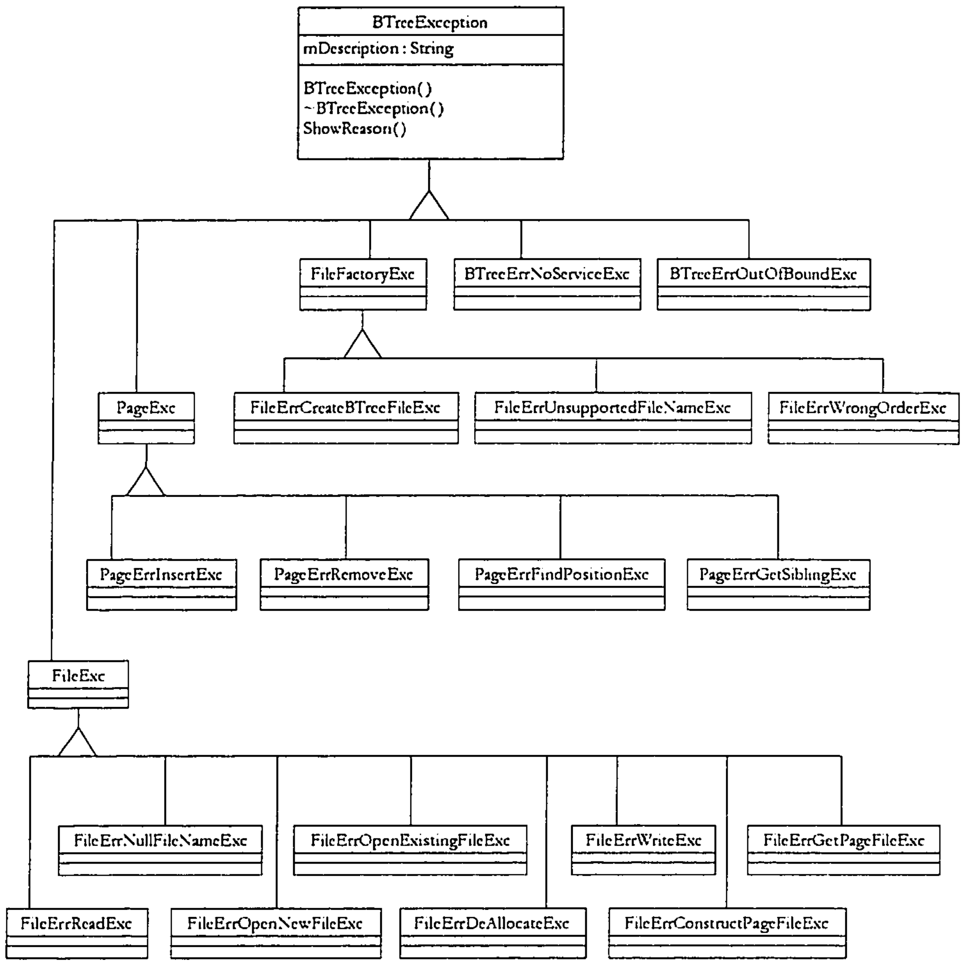


Figure 6: B^+-tree Exception Class Hierarchy.

referenced, and is able to maintain the reference to the loaded page for any further accesses. To separate the two tasks, there are two derived classes from the **Pointer** — **MemPtr** and **PagePtr**. Each **PagePtr** has a reference to a **PageFile** in which the referenced page resides, and has an offset of the page in the **PageFile**. Initially, a page has **PagePtrs** to its child pages. **MemPtr**, dealing with memory referencing, maintains a reference to a loaded page and has a reference to a **PagePtr** which holds all information about reading and writing the loaded page.

When a **PagePtr** is de-referenced, the **PagePtr** loads the referenced page from the particular **PageFile**, and returns a **MemPtr**. When a **MemPtr** is de-referenced, the **MemPtr** returns itself. By de-referencing a **Pointer**, both tasks can be achieved.

The error handling is factored out from the base of the class hierarchy, and is replaced by exceptions. Exceptions introduce a flexible method to signal the occurrence of unusual or unanticipated program events. Judicious use of exception handling makes the program easier to maintain, and more readable. Due to the nature of exception handling and the extra overhead involved, exceptions should not be over-used.

There is an exception hierarchy defined to signal some unusual program events, as shown in the class hierarchy of Figure 6. There are three sets of exceptions, FileExc, FileFactoryExc, and PageExc. FileExc handles with all errors of file I/O operations. The concrete derived classes from FileExc are described in Table 1.

Name	*Description*
FileErrNullFileNameExc	Specified file name is null.
FileErrReadExc	Failed to read data from a file.
FileErrWriteExc	Failed to write data to a file.
FileErrOpenExistingFileExc	Failed to open an existing file.
FileErrOpenNewFileExc	Failed to open an newly-created file.
FileErrDeAllocateExc	Failed to discard a record in a file.
FileErrGetPageFileExc	Failed to get a page file.
FileErrConstructPageFileExc	Failed to construct a page file.

Table 1: Description of FileExc Exceptions

FileFactoryExc deals with errors of creating B$^+$-tree. The concrete derived classes from FileFactoryExc are described in Table 2.

Name	*Description*
FileErrUnsupportedFileNameExc	Specified file name is not supported.
FileErrCreateBTreeFileExc	Failed to create a btree file.
FileErrWrongOrderExc	Specified btree file has different order.

Table 2: Description of FileFactoryExc Exceptions

PageExc copes with errors of operations on Page. The concrete derived classes from PageExc are described in Table 3.

The declaration of each exception class is made by using a macro:

```
DEFINE_EXCEPTION (ExcClassName, ExcBaseClassName, ExcDescription)
```

where `ExeClassName` specifies the name of the exception class, `ExcBaseClassName` specifies the name of the base exception class of the exception class, and `ExcDescription`

Name	*Description*
PageErrInsertExc	Failed to insert a key and a link to a page.
PageErrRemoveExc	Failed to remove a key and a link from a page.
PageErrFindPositionExc	Failed to find a position of the given key in a page.
PageErrGetSiblingExc	Failed to get the sibling page.

Table 3: Description of PageExc Exceptions

indicates the description of the reason for the exception. The description is a string that is output when the `ShowReason()` method is called.

For example, the declaration of PageErrInvalidPageOrderExc is declared using the macro as a derived class of PageExc and with an initial description message "Invalid page order" as follows:

```
DEFINE_EXCEPTION (PageErrInvalidPageOrderExc, PageExc,
                  "Invalid page order")
```

The macro declaration of PageErrInvalidPageOrderExc is equivalent to the following class declaration:

```
class PageErrInvalidPageOrderExc : public PageExc {
public:
  PageErrInvalidPageOrderExc() {mDescription = ExcDescription;}
  virtual ~ PageErrInvalidPageOrderExc() {}
};
```

There is another macro to declare a family of exceptions by declaring an abstract class that is the root class for the family:

```
DEFINE_EXCEPTION_FAMILY (ExcClassName, ExcBaseClassName).
```

where `ExcBaseClassName` specifies the name of the base exception class, and `ExcClassName` specifies the name of the root class of a family of exception classes.

For example, the family of exceptions rooted at PageExc is declared as a derived class of BTreeException as follows:

```
DEFINE_EXCEPTION_FAMILY (PageExc, BTreeException)
```

The macro declaration of the PageExc is equivalent to the following class declaration:

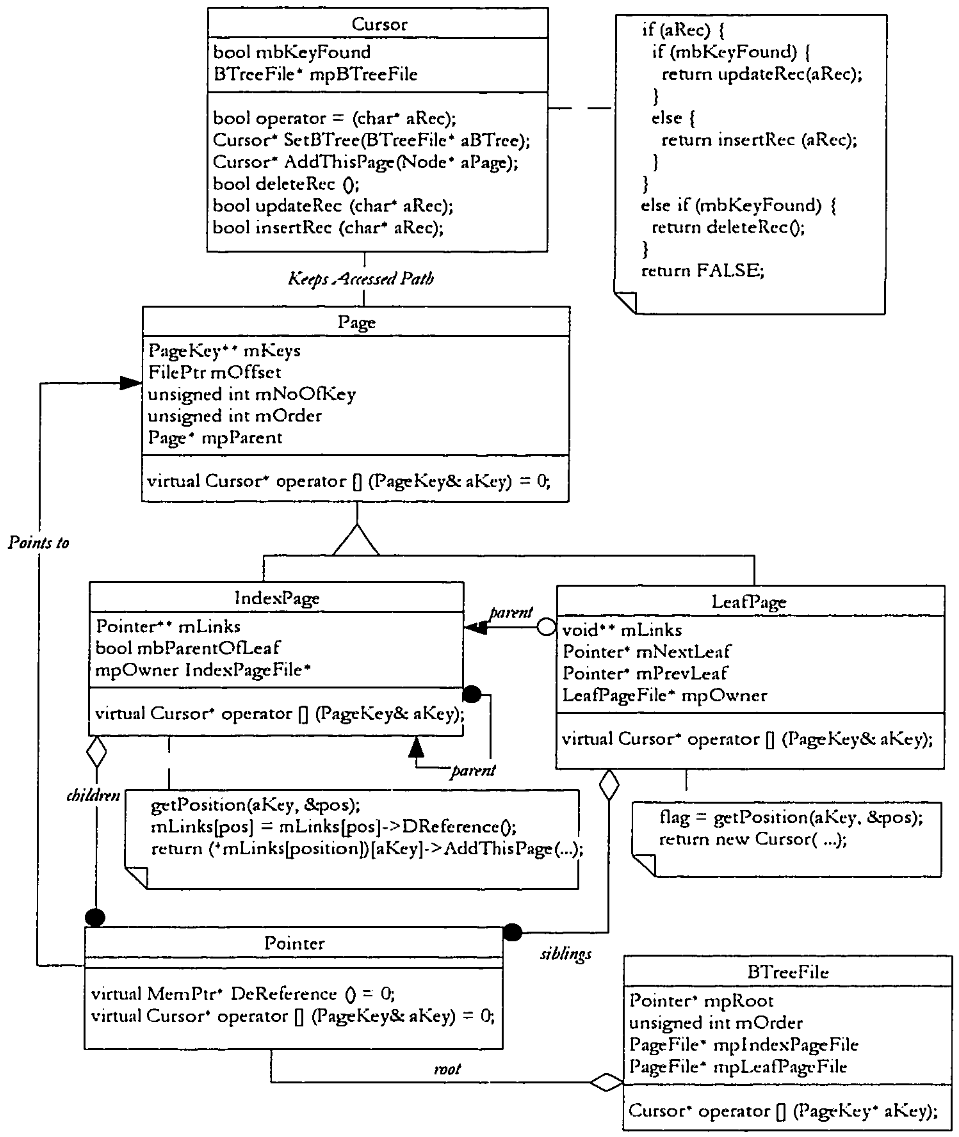
```
class PageExc : public BTreeException {
public:
   PageExc() {}
   virtual ~ PageExc() = 0;
};
```

All operations on the B$^+$-tree, such as insert, delete, update, must invoke a search from the root down to a leaf. The course from the root down to the leaf is the search path. For an exact match query, the result of the query essentially is the search path. For a range query, the result of the query is a search path, and the information about the beginning leaf and the end leaf of the specified range. Cursor (see Figure 7 and Figure 9) is introduced to maintain the search result, including the search path and the matched range, and to iterate all leafs in the matched range. The range query is not covered in the current implementation.

In some cases, an operation on a page would affect its parent. An insertion into an overflow page would create a new sibling page, so that the link to the new sibling page and the key to the new sibling page must be inserted into the parent of the overflow page. A deletion from a underflow page would cause a merge with its sibling, so that the link to the underflow page and the key to the page must be removed from its parent. By keeping the search path, a cursor collects all necessary information and invokes an operation on the parent. This avoids an extra search for the link position in the parent.

Cursor is a sort of *Iterator*. A cursor iterates over pages in the search path. Furthermore, a cursor is an indirect access reference [66, 67], and monitors operations on each Page in the search path. Based on the state of an operation on a page, the cursor decides which operation, if any, is required on the parent of the page. In that case, the cursor carries information for the operations on the parent of each page. It terminates the upward iteration if the operation on a page is complete.

The Cursor simplifies responsibilities of BTreeFile by taking charge of iterating over search path, and monitoring the operations on each page. The BTreeFile becomes an interface class which only holds knowledge about the root page of the B$^+$-tree. All details about insert, delete, and update are hidden from the BTreeFile.



```
Cursor
bool mbKeyFound
BTreeFile* mpBTreeFile

bool operator = (char* aRec);
Cursor* SetBTree(BTreeFile* aBTree);
Cursor* AddThisPage(Node* aPage);
bool deleteRec ();
bool updateRec (char* aRec);
bool insertRec (char* aRec);
```

```
if (aRec) {
  if (mbKeyFound) {
    return updateRec(aRec);
  }
  else {
    return insertRec (aRec);
  }
}
else if (mbKeyFound) {
  return deleteRec();
}
return FALSE;
```

```
Page
PageKey** mKeys
FilePtr mOffset
unsigned int mNoOfKey
unsigned int mOrder
Page* mpParent

virtual Cursor* operator [] (PageKey& aKey) = 0;
```

```
IndexPage
Pointer** mLinks
bool mbParentOfLeaf
mpOwner IndexPageFile*

virtual Cursor* operator [] (PageKey& aKey);
```

```
LeafPage
void** mLinks
Pointer* mNextLeaf
Pointer* mPrevLeaf
LeafPageFile* mpOwner

virtual Cursor* operator [] (PageKey& aKey);
```

```
getPosition(aKey, &pos);
mLinks[pos] = mLinks[pos]->DReference();
return (*mLinks[position])[aKey]->AddThisPage(...);
```

```
flag = getPosition(aKey, &pos);
return new Cursor( ...);
```

```
Pointer

virtual MemPtr* DeReference () = 0;
virtual Cursor* operator [] (PageKey& aKey) = 0;
```

```
BTreeFile
Pointer* mpRoot
unsigned int mOrder
PageFile* mpIndexPageFile
PageFile* mpLeafPageFile

Cursor* operator [] (PageKey* aKey);
```

Figure 7: A Detailed Data Model of Page and Cursor.

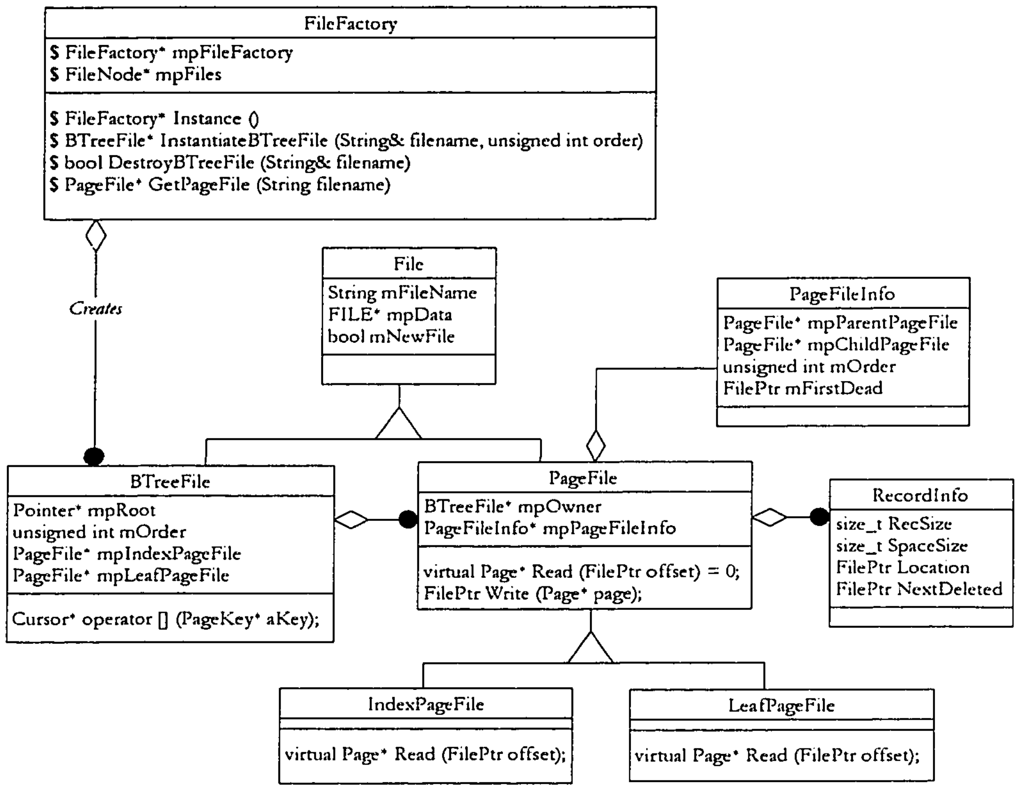

FileFactory
$ FileFactory* mpFileFactory
$ FileNode* mpFiles
$ FileFactory* Instance ()
$ BTreeFile* InstantiateBTreeFile (String& filename, unsigned int order)
$ bool DestroyBTreeFile (String& filename)
$ PageFile* GetPageFile (String filename)
Creates
File
String mFileName
FILE* mpData
bool mNewFile
PageFileInfo
PageFile* mpParentPageFile
PageFile* mpChildPageFile
unsigned int mOrder
FilePtr mFirstDead
BTreeFile
Pointer* mpRoot
unsigned int mOrder
PageFile* mpIndexPageFile
PageFile* mpLeafPageFile
Cursor* operator [] (PageKey* aKey);
PageFile
BTreeFile* mpOwner
PageFileInfo* mpPageFileInfo
virtual Page* Read (FilePtr offset) = 0;
FilePtr Write (Page* page);
RecordInfo
size_t RecSize
size_t SpaceSize
FilePtr Location
FilePtr NextDeleted
IndexPageFile
virtual Page* Read (FilePtr offset);
LeafPageFile
virtual Page* Read (FilePtr offset);

Figure 8: A Detailed Data Model of File and FileFactory

File is introduced as an abstract class, deals with the basic file I/O operations. BTreeFile is modeled as a persistent class and derived from File class, as shown in Figure 5 and Figure 8. BTreeFile holds a Pointer to the root page of a B^+-tree, and provides an interface to upper layers of the multi-index system.

The BTreeFile provides a search method, which is implemented by overloading the indexing operator []. A cursor is returned as the result of the search. Currently, exact match query with a single key as the argument is implemented. For a range query, the operator [] would take a query object, which specifies the key range of the query, and returns a cursor as a result.

```
Cursor* operator [] (PageKey&);
```

Another service provided by the BTreeFile is traversing all data records in sequential order of the keys.

```
BTreeIterator* CreateIterator() const;
```

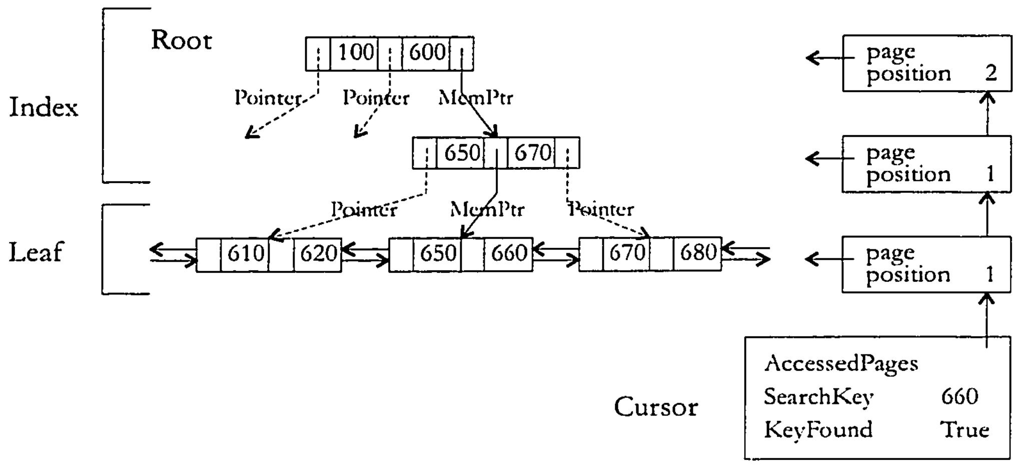


Figure 9: An Illustration of a Random Access with Key Value = 660.

Creating more than one instance of BTreeFile with the same name is eliminated by introducing a *Singleton* FileFactory as shown in Figure 5 and Figure 8. An instance of BTreeFile may only be instantiated by the FileFactory. The FileFactory maintains the knowledge about whether the specified BTreeFile is instantiated or not, so that the specific BTreeFile is not duplicated.

As an illustration of using Pointer and Cursor, a random access with a key value = 660 is shown in Figure 9, and the message trace of the random access is shown in Figure 10. The corresponding code segments are presented and explained in the following text.

```
...
FileFactory* factory = FileFactory::Instance();
BTreeFile* btree = factory->InstantiateBTreeFile("BTREE.btf", 2);
Cursor* cursor = (*btree)[PageKey(660)];
...
```

The code above is an illustration of a random access. BTreeFile starts a search for a data blob associated with a given key by invoking `BTreeFile::operator []`.

```
Cursor* BTreeFile::operator [] (PageKey& apPageKey)
{
  mpRoot = mpRoot->DeReference();
  return (*mpRoot)[apPageKey];
}
```

The BTreeFile de-references the `mpRoot`, and passes the given key to `Pointer::operator[]`.

```
Cursor* MemPtr::operator [] (PageKey& aPageKey)
{
  return (*mpPage)[aPageKey];
}
```

A `MemPtr::operator []` will pass the given key to the Page to which the Pointer points.

```
Cursor* IndexPage::operator [] (PageKey& aKey)
{
  int position;
  getPosition(aKey, &position);
  mLinks[position] = mLinks[position]->DeReference();
  return (*mLinks[position])[aKey].AddPage(*(new Node(*this, position)));
}
```

The `IndexPage::operator []` finds the position of the given key, and de-references the Pointer at the position. It calls `MemPtr::operator []` to recursively load the child page.

```
Cursor* LeafPage::operator [] (PageKey& aKey)
{
  int  iPosition = 0;
  Bool bFlag = getPosition(aKey, &iPosition);
  return new Cursor(bFlag, *(new Node(*this, iPosition)), aKey);
}
```

The `LeafPage::operator []` finds the position of the given key, checks whether the given key is found in the LeafPage, and then returns a cursor. The key at the position is expected to match the given key. If the given key is matched in the LeafPage, the `LeafPage::GetPosition ()` returns true, otherwise returns false.

The Cursor performs operations along the search path, such as insert a DataBlob, delete a DataBlob and update a DataBlob with a given key. The only interface provided to the client of the B$^+$-tree is `Cursor::operator = (void* datablob)`. As shown in Figure 7, all these three operations are interpreted and performed by the operator internally. The Cursor helps BTreeFile simplify the interface and hide internal services from the client of the B$^+$-tree.

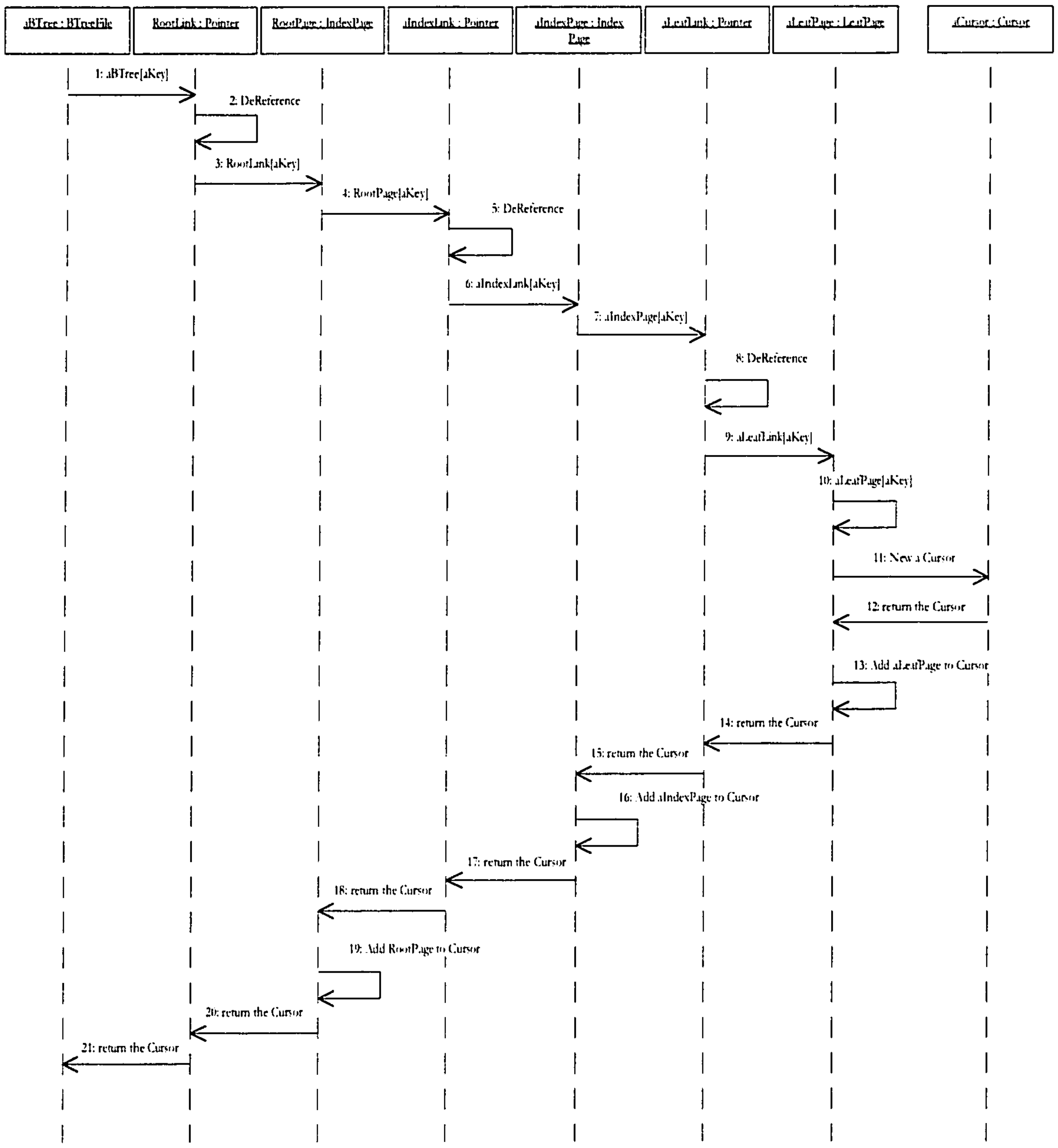


Figure 10: An Scenario of Search for a DataBlob with Given Key 660

Chapter 4

Application of Design Patterns

This chapter presents the application of the *Proxy*, *Iterator*, and *Singleton* design patterns to the re-engineering of the B-tree implementation. This chapter draws heavily from the design pattern book [34].

4.1 Proxy

Proxy provides a surrogate or placeholder for another object to control access to it.

There are some major reasons for controlling access to an object:

- While loading an expensive object, such as an image or a data bucket, there is a need to defer the full cost of the creation and initialization of such object until it is used — *an object is created on demand.*

- To simplify the interface and responsibility of an object. It is required to separate the responsibilities from an object, which has reference to those expensive objects, such as accessing, creating and deleting an object, remote accessing and protected accessing.

Figure 11 illustrates the general *Proxy* design pattern, while Figure 12 shows its use in the B^+-tree design. There are several forms of *Proxy*:

- remote proxy — provides a local representative for an object in a remote location.

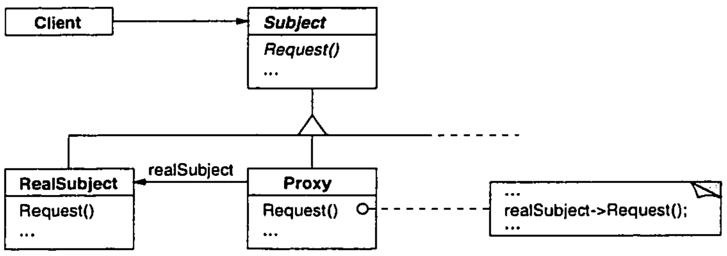


Figure 11: Proxy Design Pattern — Object Structural.

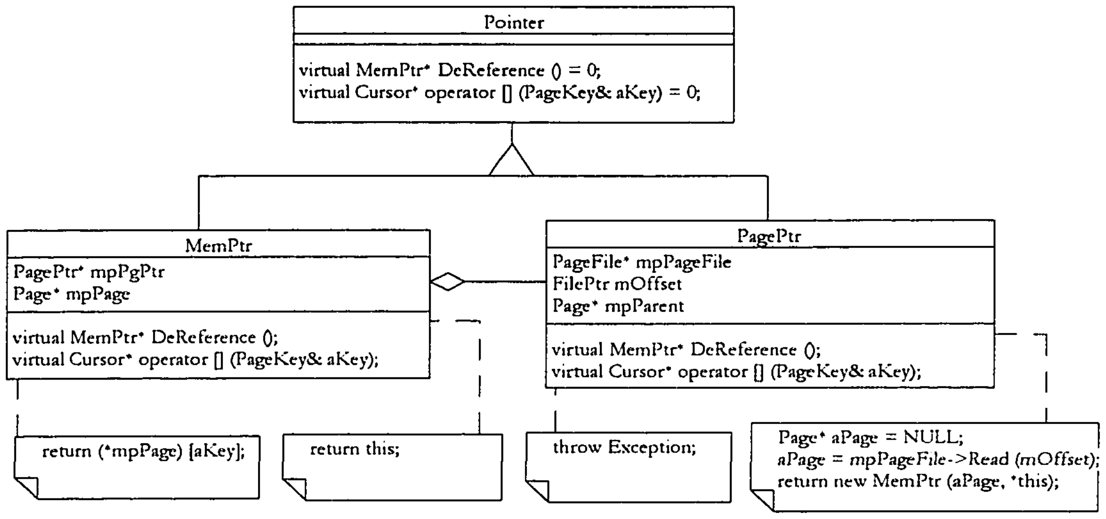
- virtual proxy — creates expensive objects on demand.
- protection proxy — controls access to the original object.
- smart reference (smart pointer)
  - access logging
  - loading a persistent object into memory
  - access locking

In the B$^+$-tree model, as shown in Figure 5 and Figure 12, Pointer is modeled as a smart pointer, to cope with loading a persistent object page, and to defer the full cost of loading until the page is used. Also a Pointer maintains a reference to a loaded page.

In the original implementation, as discussed in chapter 3, there is no mechanism to maintain a reference to a loaded page. A page only holds the offsets of child pages and its parent page. Whenever a page is accessed, the page must be loaded into memory. The page vanishes if an operation on the page is finished or the page is not passed as an argument to other operations. Any further access to the page must reload the page. A file access costs much more than a memory access.

By introducing Pointer, a memory representative for a page in persistent storage, the file access can be eliminated as much as possible. There are two derived concrete classes from the abstract Pointer class, MemPtr and PagePtr. PagePtr holds an offset to a page in a PageFile, and MemPtr maintains a reference to a loaded page in memory. A PagePtr loads the page and returns a MemPtr by de-referencing the PagePtr. A de-referencing of MemPtr returns the MemPtr itself.

If a page is to be accessed, the Pointer determines whether the page must be loaded. Actually, the need of loading a page is determined by de-referencing the


Pointer
virtual MemPtr* DeReference () = 0;
virtual Cursor* operator [] (PageKey& aKey) = 0;
MemPtr
PagePtr* mpPgPtr
Page* mpPage
virtual MemPtr* DeReference ();
virtual Cursor* operator [] (PageKey& aKey);
PagePtr
PageFile* mpPageFile
FilePtr mOffset
Page* mpParent
virtual MemPtr* DeReference ();
virtual Cursor* operator [] (PageKey& aKey);

```
return (*mpPage) [aKey];
```

```
return this;
```

```
throw Exception;
```

```
Page* aPage = NULL;
aPage = mpPageFile->Read (mOffset);
return new MemPtr (aPage, *this);
```

Figure 12: Pointer – a Smart Reference Pointer.

Pointer. If the page is loaded already, the Pointer is a MemPtr. A de-referencing of MemPtr costs a memory access only. If the page is not loaded yet, the Pointer is a PagePtr. A de-referencing of PagePtr costs a file access to load the page, and returns a MemPtr. The MemPtr keeps the loaded page in memory until the page is deleted or the BTreeFile is destroyed.

By using Pointer, a page is only loaded on demand. There is some overhead introduced by the de-referencing of MemPtr. However what the mechanism of de-referencing can achieve is more valuable than the overhead.

4.2 Iterator

Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The reasons to introduce *Iterator* design pattern are:

- To simplify the interface and responsibility of a list object. An iterator takes the responsibility for access and traversal of a list object, and for keeping track of the current element.

- To separate the traversal mechanism from the list object. Iterators are defined

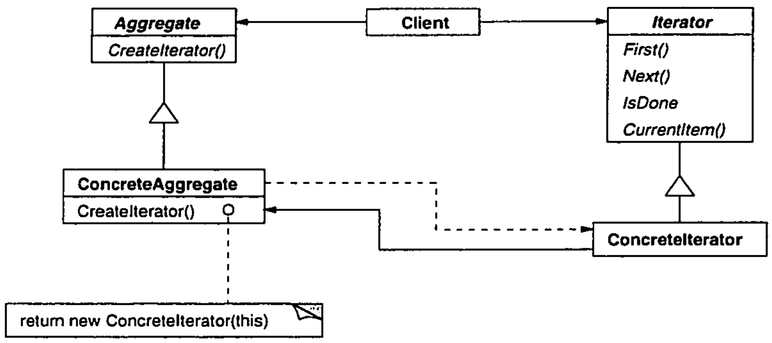


Figure 13: Iterator Design Pattern — Object Behavioral.

for different traversal policies without enumerating them in the list interface.

Figure 13 illustrates the design pattern for an iterator to access an aggregate object's contents without exposing its internal representation. The iterator provides the interface `First()`, `Next()`, `IsDone()`, and `CurrentItem()`, and hides the internal representation of an aggregate object from clients.

Figure 14 represents the iterators and their interface for the B$^+$-tree. A B$^+$-tree data structure is considered as an aggregate object, which consists of IndexPages and LeafPages. For the sequential traversal of a B$^+$-tree, in order to access all DataBlobs of the B$^+$-tree without exposing internal representation, a BTreeIterator is introduced. The BTreeIterator hides the internal implementation of the LeafPage and the sibling links between LeafPages. To encapsulate implementation details of IndexPage and LeafPage, IndexPageIterator and LeafPageIterator are applied respectively. To achieve polymorphic iteration, an abstract class Iterator provides the virtual uniform interface, `First()`, `Next()`, `IsDone()`, `blob()` and `page()`.

Basically, the BTreeIterator uses LeafPageIterator to browse a single LeafPage. When the end of a LeafPage has been reached, the BTreeIterator takes charge of searching for the next sibling, and initializes the iteration over the next sibling.

4.3 Singleton

Singleton ensures a class only has one instance, and provide a global point of access to it. 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, or when the sole instance

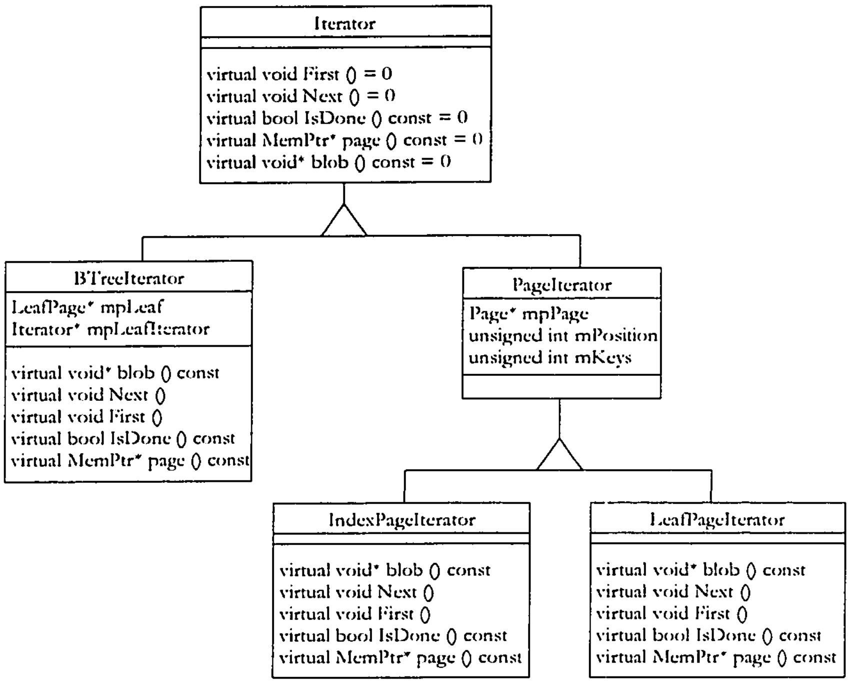

Iterator
virtual void First () = 0
virtual void Next () = 0
virtual bool IsDone () const = 0
virtual MemPtr* page () const = 0
virtual void* blob () const = 0
BTreeIterator
LeafPage* mpLeaf
Iterator* mpLeafIterator
virtual void* blob () const
virtual void Next ()
virtual void First ()
virtual bool IsDone () const
virtual MemPtr* page () const
PageIterator
Page* mpPage
unsigned int mPosition
unsigned int mKeys
IndexPageIterator
virtual void* blob () const
virtual void Next ()
virtual void First ()
virtual bool IsDone () const
virtual MemPtr* page () const
LeafPageIterator
virtual void* blob () const
virtual void Next ()
virtual void First ()
virtual bool IsDone () const
virtual MemPtr* page () const

Figure 14: Iterators in B$^+$-Tree Model

should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

FileFactory is a *Singleton* to provide a sole global point to access BTreeFile instances, as shown in Figure 8. The client of the B$^+$-tree must send a request to the sole instance of FileFactory to instantiate a BTreeFile instance with a specified name. If the BTreeFile instance has not been instantiated with the supplied name, the sole instance of FileFactory creates an instance of BTreeFile, and meanwhile, BTreeFile internally creates an IndexPageFile instance and a LeafPageFile instance. Otherwise, the sole instance of FileFactory only returns the instantiated BTreeFile instance having the supplied name.

Chapter 5

Conclusion

This thesis presents an illustration of how to use design patterns to re-engineer a B-tree implementation for reuse. 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. During the re-engineering process, we made use of several design patterns such as *Proxy*, *Iterator*, and *Singleton*. They provided a model of how to solve our design issues, many of which dealt with introducing flexibility into the design in order to make it more reusable.

Models, such as design patterns, play several important roles.

- Models and notations are an aid to *communication*.
- Models and notations improve *understanding* through precision, conciseness, and visual cues.
- A model or documentation template often provides a *checklist* of the information needed to fully understand or reuse a design artifact. For example, these may include:
    - responsibilities of each participant;
    - collaborations amongst participants;
    - purpose of the artifact; and
    - preconditions for applicability.

Furthermore, in re-engineering, they provide a target or goal to aim towards as one re-structures the code.

5.1 Contributions

We hope our main contribution was to provide a cleaner, more reusable, and easy to understand design and implementation of a B^+-tree. We followed a re-engineering process that is an evolution and iteration of design activities. Each iteration brought us closer to the current design of B^+-tree. The major improvements are:

- Adopting a B^+-tree to improve sequential access.
- Encapsulation through the introduction of the Page abstract class and hierarchy.
- Use of Pointer, to ensure that a page is loaded on demand, and to reduce file access to the minimum.
- Cursor, as an indirect access control proxy and iterator, makes B^+-tree more abstract and reusable. The Cursor provides a mechanism that prevents the B^+-tree from being implementation dependent. Cursor controls use of the operations on B^+-tree to iterate over query results.
- Overloading the indexing operator [] to allow a variety of types of query.
- Using a singleton FileFactory to control instantiation of the B^+-tree, and prepare the way for true multi-user access.

The re-engineering provides the flexibility to extend the functionality of B^+-tree, such as multi-user access, and other kinds of query. These are future work.

5.2 Future Direction

Our ultimate goal is to build a template for a B^+-tree indexing component, to handle arbitrary key and data classes. The query handling can be easily extended; for example, we are considering a range query object and a range query cursor as a means of supporting range queries. Then we will reuse the design and implementation to cover multi-dimensional indexing data structures based on trees.

Other potential directions are the use of the *State* pattern to simplify the use of smart pointers; and to make the B^+-tree work in a concurrent multi-user environment.

5.2.1 *State* Pattern

The *State* pattern may be exploited to simplify Pointer de-referencing. The *State* pattern allows an object to alter its behavior when its internal state changes. Actually, an object of Pointer has two states which are presented by MemPtr and PagePtr. The behavior of Pointer depends on its states. A possible design modification is to introduce a Link class to wrap the Pointer as its internal state attribute. The Pointer de-referencing becomes into an internal operation of Link.

The Link class simplifies the Pointer de-referencing. Each IndexPage contains 2k+1 Links referring to its child pages, where k is the order of B^+-tree. De-referencing a Link only changes the internal state of the Link. The change of state is transparent to the client of the Link, and the details about MemPtr and PagePtr are encapsulated.

In Section 3.3, the search operation of IndexPage, first de-references the Pointer, `mLinks[position]`, before the search operation of Pointer is invoked. Both operations must always be performed in this order. By introducing the Link class, the de-referencing is done internally by the Link.

The search operation of BTreeFile can be simplified as follows:

```
Cursor* BTreeFile::operator [] (PageKey& apPageKey)
{
  return (*mpRoot)[apPageKey];
}
```

5.2.2 Multi-User Environment

A common requirement of the database domain is multi-user access. An immediate need is to control the creation and modification of multiple instances of the B-tree files that form the indexes to the database. A *Singleton* FileFactory is adopted to act as a name server and to control the access to the index files on disk in one thread. Each thread has a *Singleton* instance of FileFactory, and each instance of FileFactory has a list of BTreeFiles that are identical and are being accessed by the thread. FileFactory ensures that there is only one identical BTreeFile instance within one thread.

A simple-minded scheme for multi-user access is to control the number of threads with write permission. One may allow many read-only threads simultaneously without problems, but if there is a thread with write permission then it must be the only active thread. This scheme can be easily introduced into FileFactory. FileFactory detects whether a given BTreeFile is being accessed by other threads, and detects whether an access mode is allowed for the given BTreeFile. There are two types of locks on a BTreeFile: a shared read lock and an exclusive write lock. Table 4 shows the compatibility rule of the two types of locks.

Current Locks	Requested Lock	
	read lock	write lock
no locks	OK	OK
one or more read locks	OK	denied
one write lock	denied	denied

Table 4: Compatibility between different lock types

There are implications to the Cursor and Page classes as well. Some of these arise even with single threads of control.

Robust Iterator Problem Consider a single user, a single thread of control, but with many cursors such as in a nested loop:

```
Cursor c1 = db[ query1 ];
loop over c1 results
{
    Cursor c2 = db[ query2 ];    //maybe query2 based on c1
    loop over c2 results
}
```

in the context where the database db may be updated (or new inserts and deletes). A cursor depends on the path through the tree actually existing and being consistent with the state of the cursor, so cursor c2 cannot alter the database (and hence the index) while the cursor c1 is active. The changes must be delayed until the transaction is committed.

Here you have one transaction, maybe with nested subtransactions, and integrity can be synchronized at the end of that transaction. This is essentially the robust iterator problem, as solved in ET++ [50].

General Concurrency Genuine multi-user concurrency is even more complicated than the robust iterator problem. There may be multiple, unrelated, transactions on the database at the same time, and there are concurrency control issues. One definitely needs a locking protocol at a level below that of the whole index (which is the level of our simple scheme above for FileFactory). Locking protocols need to be adopted for pages, and Cursor must have the capability to lock and unlock pages in the search path from the root down to a leaf.

These issues have been treated before for B-trees. Samadi [75], Bayer and Schkolnick [8] show various ideas to handle concurrent access in a multi-user environment.

5.3 Conclusion

This paper presents an illustration of how to use design patterns to re-engineer a B-tree implementation for reuse. 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.

Cursor, as an indirect access reference and iterator, makes the B^+-tree more general and reusable. The Cursor provides a mechanism that prevents the BTreeFile from being application dependent. Some applications of Cursor are still being considered, such as range query cursor. It is possible to load a Page on demand by introducing Pointer. FileFactory controls access to each instance of BTreeFile, and supports multiple user access.

The re-engineering provides the additional functionality of B^+-tree, such as sequential access, and the flexibility to extend the primitive query handling. The final intention is to build a template for a B^+-tree indexing component, to handle arbitrary key and data classes. Then we will reuse the design and implementation to cover multi-dimensional indexing data structures based on trees.

A major limitation at the moment is the lack of robustness of Cursor. When more than one cursor, such as in a nested query, may be actively traversing the B^+-tree and altering the underlying data records through insertions, deletions, and updates, then the structure of the index will change. This can corrupt the state of a cursor since a cursor depends on a given path through the tree. Our design works for single cursors, and for multiple read-only cursors. We need to adopt the robust iterator design of Kofler [50] to be able to handled nested cursors within a single thread of control. For multiple threads, and true multi-user access, then additionally we need to adopt a locking protocol such as those suggested by Samadi [75], and Bayer and Schkolnick [8] and to decide on a policy for handling the exceptions to insert/delete/update operations due to locked pages.

Bibliography

[1] G. Abowd, R. Allen, and D. Garlan. Using style to give meaning to software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering, ACM Software Engineering Notes*, volume 18 of *3*, pages 9–20, December 1993.

[2] C. Alexander. *A Timeless Way of Building.* Oxford University Press, 1979.

[3] Apple Computer. *Macapp 2.0 General Reference Manual*, 1990.

[4] G. Arango and R. Prieto-Diaz. Domain analysis: Concepts and research directions. In R. Prieto-Diaz and G. Arango, editors, *Domain Analysis: Acquisition of Reusable Information for Software Construction.* Computer Society Press Tutorial, May 1989.

[5] Len Bass, Paul Clements, Rick Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.

[6] D.S. Batory and S.W. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. on Software Engineering and Methodology*, 1(4):355–398, 1992.

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[8] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Inf.*, 9(1):1–21, 1977.

[9] K. Beck and R. Johnson. Patterns generate architecture. 1994.

[10] Roger P. Beck, Satish R. Desai, Doris R. Ryan, Ronald W. Tower, Dennis Q. Vroom, and Linda Mayer Wood. Architecture for large-scale reuse. *AT&T Technical Journal*, 71(6):34–45, 1992.

[11] Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability.* ACM Press/Addison-Wesley, 1989.

[12] A. Birrer and T. Eggenschwiler. Frameworks in the financial engineering domain: An experience report. In O.M. Nierstrasz, editor, *ECOOP'93 — Object-Oriented Programming*, pages 21–35, Berlin, 1993. Springer-Verlag.

[13] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, 2nd edition, 1993.

[14] Grady Booch. *Software Components with Ada — Structures, Tools, and Subsystems.* Benjamin/Cummings, Menlo Park, CA, 1987.

[15] Grady Booch and Michael Vilot. The design of the C++ Booch components, ECOOP/OOPSLA'90. *ACM SIGPLAN Notices*, 25(10):1–11, 1990.

[16] Grady Booch and Michael Vilot. Simplifying the Booch components. *The C++ Report*, pages 41–52, June 1993.

[17] Christine L. Braun. Reuse. In John Marciniak, editor, *Encyclopedia of Software Engineering*, pages 1055–1069. John Wiley and Sons, 1994.

[18] B. Bruegge, T. Gottschalk, B. Luo. A framework for dynamic program analyzers. In *OOPSLA '93*, pages 65–82, 1993.

[19] David Budgen. *Software Design.* Addison-Wesley, 1993.

[20] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* John Wiley & Sons, 1996.

[21] P.R. Calder, M.A. Linton, J.M. Vlissides. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, February 1989.

[22] Roy H. Campbell, Nayeem Islam, David Raila, Peter Madany. Designing and implementing CHOICES: An object-oriented system in C++. *Communications ACM*, 36(9):117–126, September 1993.

[23] Yoonsik Cheon and Gary Leavens. A quick overview of Larch/C++. Technical Report 93-18, Dept of Computer Science, Iowa State University, June 1993. 34 pages.

[24] Elliot Chikofsky, James Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13 - 17, January 1990.

[25] J. Craig Cleaveland. Building application generators. *IEEE Software*, 5:25–33, July 1988.

[26] P. Coad. Patterns (Workshop), OOPSLA'92. *OOPS Messenger*, 4(2):93–96, October 1993.

[27] Douglas Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–138, June 1979.

[28] James O. Coplien and Douglas C. Schmidt (editors). *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[29] L. Peter Deutsch. Reusability in the Smalltalk-80 programming system. In *ITT Proceedings of the Workshop on Reusability in Programming 1983*, pages 72–76, 1983. In [32, pp.91–95].

[30] R.G. Fichman and C.F. Kemerer. Object-oriented and conventional analysis and design methodologies. *IEEE Computer*, 25(20):22–39, 1992.

[31] W.B. Frakes and S. Isoda. Success factors of systematic reuse. *IEEE Software*, pages 15–19, September 1994.

[32] Peter Freeman. Tutorial: Software Reusability. IEEE Computer Society Press, 1987.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. In O.M. Nierstrasz, editor, *Proceedings of ECOOP'93*, Lecture Notes in Computer Science **707**, pages 406–431. Springer-Verlag, Berlin, 1993.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[35] D. Garlan, R. Allen, J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering, ACM Software Engineering Notes 19*, pages 175–188, 12 1994.

[36] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company, 1993.

[37] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

[38] M.L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, 32(4):548–566, 1993.

[39] John V. Guttag and et al Jim J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, 1993.

[40] David Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, 25(1):8–20, January 1992.

[41] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *ECOOP/OOPSLA'90*, pages 169–180, 1990.

[42] Ian M. Holland. Specifying reusable components with contracts. In *ECOOP'92*, Lecture Notes in Computer Science **615**, pages 287–308. Springer-Verlag, 1992.

[43] Michael A. Jackson. *Principles of Program Design*. Academic Press, New York, 1975.

[44] Michael A. Jackson. *System Development*. Prentice-Hall, 1983.

[45] R. Johnson. Documenting frameworks using patterns, OOPSLA'92. *ACM SIGPLAN Notices*, 27(10):63–76, October 1992.

[46] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS 91-1696, University of Illinois, May 1991.

[47] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, 1988.

[48] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 1986.

[49] Even-André Karlsson (editor). *Software Reuse: A Holistic Approach.* John Wiley and Sons, 1995.

[50] T. Kofler. Robust iterators for ET++. *Structured Programming*, 14(2):62–85, 1993.

[51] Tim Korson and John D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–60, September 1990.

[52] Tim Korson and John D. McGregor. Technical criteria for the specification and evaluation of object-oriented libraries. *Software Engineering Journal*, pages 85–94, March 1992.

[53] Scott Robert Ladd. *C++ Components and Algorithms.* M&T Books, 1992.

[54] Richard Lajoie. Using, reusing and describing object-oriented frameworks. Master's thesis, McGill University, Montreal, July 1993.

[55] Richard Lajoie and Rudolf K. Keller. Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert. In *Colloquium on Object-Orientation in Databases and Software Engineering, 62nd Congress of ACFAS, Montreal, May 16–17*, 1994.

[56] J.M. Lake, R.E. Johnson, C. McConnell. The RTL system: A framework for code optimization. In R. Giegerich and S.L. Graham, editors, *Code Generation — Concepts, Tools, techniques*, pages 255–274. Springer-Verlag, London, 1991.

[57] R.G. Lanergan and C.A. Grasso. Software engineering with reusable designs and code. *IEEE Trans. SE*, SE-10(5):498–501, September 1984.

[58] R.G. Lanergan and B.A. Poynton. Reusable code: The application development technique of the future. In *Proceedings of the IBM SHARE/GUIDE Software Symposium*, pages 127–136, IBM, Monterey, CA, October 1979.

[59] D. Lea. Christopher Alexander: An introduction for object-oriented designers. *Software Engineering Notes*, 19(1):39–46, January 1994.

[60] Robert Martin, Dirk Riehle, Frank Buschmann (editors). *Pattern Languages of Program Design 3.* Addison-Wesley, 1998.

[61] R. Marty, A. Weinand, E. Gamma. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[62] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–64, March 1987.

[63] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[64] Betrand Meyer. Lessons from the design of the Eiffel libraries. *CACM*, 33(3):68–84, 1991.

[65] Betrand Meyer. *Reusable Software — The Base Object-Oriented Component Libraries.* Prentice Hall, 1994.

[66] Carl Nelson. How to stop worrying and start loving C++, part i. *Computer*, pages 92–94, June 1994.

[67] Carl Nelson. How to stop worrying and start loving C++, part ii. *Computer*, pages 104–106, July 1994.

[68] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[69] D. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, March 1979.

[70] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Trans. Software Eng.*, SE-12(2):251–257, 1986.

[71] C. Potts and G. Bruns. Recording the reasons for design decisions. In *Proceedings of the 10th International Conference on Software Engineering*, pages 418–427, Los Alamitos, CA, 1988. IEEE Computer Society Press.

[72] Jeff Poulin and Will Tracz. WISR'93: 6th annual workshop on software reuse — Summary and workshop group reports. *ACM SIGSOFT Software Engineering Notes*, 19(1):55–71, January 1994.

[73] Danielle Ribot, Blandine Bongard, Björn Grönquist. Impact of reuse on organizations. In *Proc. Reuse'93*, pages 24–26, Lucca, Italy, March 1993. IEEE CS Press.

[74] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

[75] B. Samadi. B-trees in a system with multiple views. *Inf. Process. Lett.*, 5(4):107–112, 10 1976.

[76] Wilhelm Schäffer, Rubén Prieto-díaz, and Masao Matsumoto. *Software Reusability.* Ellis Horwood, 1993.

[77] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Inc. 1996.

[78] G. Sindre, E.-A, Karlsson, and T. Staalhane. A method for software reuse through large component libraries. In Osman Abou-Rabia, Carl K. Chang, and Waldemar W. Koczkodaj, editors, *Proceedings ICCI'93*, pages 464–468, Sudbury, Ontario, Canada, May 27–29 1993. IEEE Computer Society Press, Los Alamitos, CA.

[79] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[80] Taligent Inc. Building object-oriented frameworks. A Taligent White Paper, 1994.

[81] W. Tracz. Domain analysis working group report. In *First International Workshop on Software Reusability*, Dortmund, Germany, July 3–5 1991.

[82] W. Tracz. The three cons of software reuse. In *Proceedings of the Fourth Annual Workshop on Reuse*, Syracuse, NY, 1991.

[83] Panu Viljamaa. The patterns business: Impressions from PLoP-94. *ACM Software Engineering Notes*, 20(1):74–78, January 1995.

[84] John Vlissides, James O. Coplien and Norman L. Kerth (editors). *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[85] J.D. Warnier. *Logical Construction of Programs*. Van Nostrand, 1980.

[86] Richard Waters, Elliot Chikofsky. Reverse engineering. *Communications of the ACM*, 37(5):22 - 24, May 1994.

[87] Rebecca Wirfs-Brock and Ralph Johnson. Surveying current research in object-oriented design. *CACM*, 33(9):104–124, September 1990.

[88] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[89] E. Yourdon and L.L. Constantine. *Structured Design*. Prentice-Hall, 1979.

Appendix A

Definition and Implementation of Class Page

A.1 Definition of Class Page

```
/*
** Page.h
**
** Page Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   - Initial implemention
**
**   03-July-1997 Steven Li
**   - Used BLOBMaker and BLOBScanner to do page
**     buffer read and write
**   - disabled ctor(), cctor(), operator = ().
**     enfored that a page only can be created by a page file.
**   - made dtor() private and added Free() to destroy Page explicitly
**   - checked memory leaking.
**   - took of mOffset attribute to avoid the duplication in PagePtr.
*/

#ifndef PAGE_H
#define PAGE_H

/*
** forward declaration
*/
class Cursor;
class PageKey;
class Pointer;
class State;
class PageFile;
class PagePtr;
class LeafPage;
class IndexPage;
class BLOBMaker;
class BLOBScanner;

/*
** include files
*/
#include "Boolean.h"
#include "File.h"
#include "BTreeException.h"
#include "Iterator.h"

#define min(a, b)  (((a) < (b)) ? (a) : (b))

/*
** Macro to define Exception classes
*/
DEFINE_EXCEPTION_FAMILY (PageExc, BTreeException)
DEFINE_EXCEPTION (PageErrInvalidPageOrderExc, PageExc,
                  "Invalid page order")
DEFINE_EXCEPTION (PageErrInsertExc, PageExc,
                  "Failed to insert a PageKey and a Link into a Page")
DEFINE_EXCEPTION (PageErrRemoveExc, PageExc,
                  "Failed to remove a PageKey and a Link into a Page")
DEFINE_EXCEPTION (PageErrWriteRootExc, PageExc,
                  "Failed to write a BTREE-ROOT back")
DEFINE_EXCEPTION (PageErrWriteSiblingExc, PageExc,
                  "Failed to write a sibling back")
DEFINE_EXCEPTION (PageErrWritePageExc, PageExc,
                  "Failed to write the Page back")
DEFINE_EXCEPTION (PageErrWriteParentExc, PageExc,
                  "Failed to write the parent page back")
DEFINE_EXCEPTION (PageErrFindPositionExc, PageExc,
                  "Failed to find a position according to the given key")
DEFINE_EXCEPTION (PageErrWriteChildPageExc, PageExc,
                  "Failed to write the child page back")
DEFINE_EXCEPTION (PageErrGetSiblingExc, PageExc,
                  "Failed to get the sibling page")

/*
** Definition of Page class
*/

class Page {
public:
  virtual void Free() = 0;        // destroy a sub-BTree
  Bool IsFull() const;
  Bool IsRoot() const;

  Bool SetParent(const IndexPage* parent);
  IndexPage* GetParent() const;

  int GetNoOfKey() const;
  void SetNoOfKey(int aNum);
  int GetOrder() const;

  void SetKey(const int aPosition, const PageKey* apPageKey);
  const PageKey* GetKey(const int aPosition) const;

  void SetOffset(const FilePtr aOffset);
  FilePtr GetOffset() const;

  virtual PageFile* GetPageFile() const = 0;

  virtual unsigned int GetSize() const;
```

```
   virtual Bool BufferRead(BLOBScanner& blob);
   virtual Bool BufferWrite(BLOBMaker& blob) const;

   virtual Cursor* operator [] (PageKey& aPageKey) = 0;   //index

   virtual void SetLink(const int, Pointer* apPtr) = 0;
   virtual void SetLink(const int, const String* apPtr) = 0;
   virtual const void* GetLink(const int) const = 0;
   virtual Bool GetFirst(Page* &) = 0;
   virtual Bool getPosition (const PageKey& key, int* aPos) = 0;

   virtual State* Insert(const PageKey& key, const String* ptr) = 0;
   virtual State* Insert(const PageKey& key, Pointer* ptr) = 0;
   virtual State* Remove(const int keyPos, const int pointerPos,
                         const int parentPos) = 0;

   virtual Iterator* CreateIterator() = 0;

   virtual Bool GetPosition(Page& leaf, int& position) = 0;

#ifdef _DEBUG
  virtual void print();
#endif

protected:
  int mOrder;            //Page Order                -- transient
  IndexPage* mpParent;   //parent Pointer            -- transient
  FilePtr mOffset;       //Page Offset in a PageFile -- transient
  PageKey** mKeys;       //array of PageKey
  int mNoOfKey;          //current number of PageKeys
protected:
  Page(const PageFile& owner); // constructor
  virtual ~Page();            //destructor
  Bool doExpand(const Page&);
  void KeyBoundCheck(const int position) const;
private:
  Page();                          // NOT DEFINED
  Page(const Page&);               // NOT DEFINED
  Page& operator = (const Page&); // NOT DEFINED
};



/*
** Definition of PageIterator class
*/

class PageIterator : public Iterator {
public:
  virtual void First () = 0;
  virtual void Next () = 0;
  virtual Bool IsDone () = 0;
  virtual Pointer* page () const = 0;
  virtual String* blob () const = 0;
protected:
  Page* mpPage;
  int   mPosition;
  int   mKeys;
protected:
PageIterator(Page* apPage);
  virtual ~PageIterator();
private:
  PageIterator();
  PageIterator(const PageIterator&);
  PageIterator& operator = (const PageIterator&);
};
```

```
#endif
```

A.2 Implementation of Class Page

```
/*
** Page.cxx
**
** Page Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   - Initial implemention
**
**   03-July-1997 Steven Li
**   - Used BLOBMaker and BLOBScanner to do page
**     buffer read and write
**   - disabled ctor(), cctor(), operator = ().
**     enfored that a page only can be created by a page file.
**   - checked memory leaking.
*/


/*
** include files
*/
#include <stdlib.h>
#include "Page.h"
#include "MemPtr.h"
#include "PageKey.h"
#include "PagePtr.h"
#include "PageFile.h"
#include "Blob.h"
#include "IndexPage.h"

// --- implementation of class Page

/*
** Service: Page::Page
** Description:
**   The only valid constructor of the page. A page only can be
**   constructed by an instance of PageFile class, and the instance
**   of the PageFile class is the owner of the created object of
**   Page
*/
Page::Page(const PageFile& pagefile)
  : mNoOfKey(0), mpParent(NULL), mOrder(pagefile.GetPageOrder()),
    mOffset(FP_UNKNOWN)
{
  //initialize mKeys
  mKeys = new PageKey* [mOrder];
  for (int scan = 0; scan < mOrder; scan++) {
    mKeys[scan] = NULL;
  }
}


/*
** Service: Page::~Page
** Description:
**   Destructor.
*/
Page::~Page()
{
  if(mKeys) {
    for (int scan = 0; scan < mNoOfKey; scan++) {
      delete mKeys[scan];
    }
  }
}


//
// Service: Page::doExpand
// Description:
//   Increment the order of page, and expand
//   pagekey list one more cell.
//
Bool Page::doExpand (const Page& page)
{
  mOrder++;
  mNoOfKey = page.mNoOfKey;
  mOffset  = page.mOffset;
  mpParent = page.mpParent;

  delete [] mKeys;
  mKeys = new PageKey* [mOrder];
  for (int scan = 0; scan < page.mOrder; scan++) {
    mKeys[scan] = new PageKey(*page.mKeys[scan]);
  }
  mKeys[scan] = NULL;
  return TRUE;
}


/*
** Service: Page::IsFull
** Description:
**   Examines whether the Page is full.
*/
Bool Page::IsFull() const
{
  return (mNoOfKey < mOrder) ? FALSE : TRUE;
}


/*
** Service: Page::IsRoot
** Description:
**   Examines whether the Page is the Root of BTree.
*/
Bool Page::IsRoot() const
{
  return (mpParent) ? FALSE : TRUE;
}


/*
** Service: Page::SetParent
** Description:
**   Sets the parent of the Page. It only does a shollow copy.
*/
Bool Page::SetParent(const IndexPage* parent)
{
  mpParent = (IndexPage*)parent;
  return TRUE;
}


/*
** Service: Page::GetParent
** Description:
**   Returns the parent of the Page. It is not allowed
**   to modify the returned parent Pointer.
*/
IndexPage* Page::GetParent() const
{
  return mpParent;
}


/*
** Service: Page::GetOrder
** Description:
**   Returns the order of Page.
*/
int Page::GetOrder() const
{
  return mOrder;
}


/*
** Service: Page::GetNoOfKey
** Description:
**   Returns the number of PageKeys in the Page.
*/
int Page::GetNoOfKey() const
{
  return mNoOfKey;
}


//
// Service: Page::SetNoOfKey
// Description:
//
void Page::SetNoOfKey(int aNum)
{
  mNoOfKey = aNum;
}


/*
** Service: Page::GetKey
** Description:
**   Returns a pointer to a PageKey at the specified position.
**   It is not allowed to modified the returned PageKey.
** Exceptions:
**   BTreeErrOutOfBoundExc
*/
const PageKey* Page::GetKey(const int aPosition) const
{
  KeyBoundCheck(aPosition);
  return mKeys[aPosition];
}


/*
** Service: Page::SetKey
** Description:
```

```
**   deep assignment of PageKey
** Exceptions:
**   BTreeErrOutOfBoundExc
*/
void Page::SetKey(const int aPosition, const PageKey* apPageKey)
{
  KeyBoundCheck(aPosition);
  assert(apPageKey);
  if (mKeys[aPosition]) {
    delete mKeys[aPosition];
  }
  mKeys[aPosition] = new PageKey (*apPageKey);
}


/*
** Service: Page::SetOffset
** Description:
**   Sets transient offset mOffset attribute.
*/
void Page::SetOffset(const FilePtr aOffset)
{
  mOffset = aOffset;
}


/*
** Service: Page::GetOffset
** Description:
**   Returns the offset of Page in a PageFile
*/
FilePtr Page::GetOffset() const
{
  return mOffset;
}


/*
** Service: Page::GetSize
** Description:
**   Returns the size of Page. It only considers
**   the persistent attributes in the Page.
*/
unsigned int Page::GetSize() const
{
  unsigned int size = 0;
  size += (PageKey::GetSize())*mOrder;  //mKeys
  size += sizeof(int);                  //mNoOfKeys
  return size;
}


/*
** Service: Page::BufferRead
** Description:
**   Updates the page base on the supplied raw blob.
**   The parent of the page is set by the pageptr of
**   the page.
*/
Bool Page::BufferRead(BLOBScanner& blob)
{
  mNoOfKey = (int) blob;

  // set up mKeys
  for (int scan = 0; scan < mNoOfKey; scan++) {
    mKeys[scan] = PageKey::Make(blob);
  }
  return TRUE;
}


//
// Service: Page::BufferWrite
// Description:
//   Appends the page to the supplied raw blob.
//
Bool Page::BufferWrite(BLOBMaker& blob) const {
  blob += (int) mNoOfKey;

  for (int scan = 0; scan < mNoOfKey; scan++) {
    mKeys[scan]->BufferWrite(blob);
  }
  return TRUE;
}


//
// Service: Page::KeyBoundCheck
// Description:
//
void Page::KeyBoundCheck (const int aPosition) const
{
  if(aPosition < 0 || aPosition >= mOrder) {
    throw BTreeErrOutOfBoundExc();
  }
}


#ifdef _DEBUG
//
// Service: Page::print
// Description:
//   Print out the content of the page
//
void Page::print()
{
  cout << "  < P a g e > " << endl;
  cout << "    mOffset   = " << mOffset  << endl;
  cout << "    mOrder    = " << mOrder   << endl;
  cout << "    mNoOfKey  = " << mNoOfKey << endl;

  if (mNoOfKey > 0) {
    cout << "    mKeys     = " << endl;
    for (int scan = 0; scan < mOrder; scan++) {
      if (mKeys[scan] == NULL) {
        cout << "      NULL " << endl;
      }
      else {
        cout << "      " << mKeys[scan]->GetKey() << endl;
      }
    }
  }
  cout << "    mpParent  = ";
  if (mpParent == NULL) {
    cout << "NULL" << endl;
  }
  else {
    cout << mpParent->GetOffset() << endl;
  }
}
#endif
```

```
//
// Implementation of PageIterator class
//


//
// Service: PageIterator::PageIterator
// Description:
//   Constructor
//
PageIterator::PageIterator(Page* apPage)
{
  mpPage    = apPage;
  mPosition = 0;
  mKeys     = apPage->GetNoOfKey();
}


/*
** Service: PageIterator::~PageIterator
** Title:   Default Constructor
** Description:
**
*/
PageIterator::~PageIterator()
{}
```

Appendix B

Definition and Implementation of Class IndexPage

B.1 Definition of Class IndexPage

```
/*
** IndexPage.h
**
** IndexPage Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**   - Used BLOBMaker and BLOBScanner to do page
**     buffer read and write
**   - disabled ctor(), cctor(), operator = ().
**     enfored that a page only can be created by a page file.
**   - made dtor() private and added Free() to destroy Page explicitly
**   - checked memory leaking.
*/


#ifndef INDEXPAGE_H
#define INDEXPAGE_H

/*
** forward declaration
*/
class Page;
class Pointer;
class MemPtr;
class Cursor;
class Bundle;
class Iterator;
class DataPage;
class String;
class IndexPageFile;
```

```
/*
** include files
*/
#include "Page.h"
#include "State.h"
#include "Boolean.h"


/*
** Definition of IndexPage class
*/

class IndexPage : public Page {
  friend class IndexPageFile;
public:
  void Free();

  virtual Cursor* operator [](PageKey& aPageKey);   //index

  virtual void SetLink(const int, Pointer* apPtr);
  virtual void SetLink(const int, const String*);
  virtual const void* GetLink(const int) const;

  virtual Bool GetFirst(Page* &);
  virtual Bool getPosition(const PageKey& key, int* aPos);

  virtual State* Insert(const PageKey& key, const String* ptr);
  virtual State* Insert(const PageKey& key, Pointer* ptr);
  virtual State* Remove(const int keyPos,
                        const int pointerPos,
                        const int parentPos);

  Bool IsParentOfLeaf(void) const;
  void SetParentOfLeaf(Bool aFlag);
  void LinkBoundCheck(const int pos) const;

  virtual PageFile* GetPageFile() const;

  virtual unsigned int GetSize() const;
  virtual Bool BufferRead(BLOBScanner& blob);
  virtual Bool BufferWrite(BLOBMaker& blob) const;

  virtual Iterator* CreateIterator();

  Bool GetPosition(Page& leaf, int& position);

#ifdef _DEBUG
  virtual void print();
#endif

private:
  Pointer** mLinks;         //array of Pointer
  Bool mbParentOfLeaf;
  IndexPageFile* mpOwner;   //the owner of the page  -- transient
private:
  IndexPage (IndexPageFile& owner);
  virtual ~IndexPage();
  IndexPage* Expand ();
  Bool doExpand (const IndexPage&);
  Bool doDelete(int, int);
  Bool doInsert(const PageKey& aKey, Pointer* ptr);
  IndexPage& getSibling(State::R_L*, const int);
  void reDistribute(IndexPage&, const int, State::R_L);
  IndexPage& merge(IndexPage&, const int, State::R_L);
  MemPtr* split(IndexPage& aIndexPage);
private:
  IndexPage();                                        //NOT DEFINED
  IndexPage(const IndexPage& aIndexPage);             //NOT DEFINED
  IndexPage& operator = (const IndexPage& aIndexpage); //NOT DEFINED
};



//
// Definition of IndexPageIterator class
//

class IndexPageIterator : public PageIterator{
public:
  IndexPageIterator(IndexPage* apPage);
  virtual ~IndexPageIterator();
  virtual void First ();
  virtual void Next ();
  virtual Bool IsDone ();
  virtual Pointer* page () const;
  virtual String* blob () const;
private:
  IndexPageIterator();                                      // NOT DEFINED
  IndexPageIterator(const IndexPageIterator&);              // NOT DEFINED
  IndexPageIterator& operator = (const IndexPageIterator&); // NOT DEFINED
};

#endif
```

B.2 Implementation of Class IndexPage

```
/*
** IndexPage.cxx
**
** IndexPage and IndexPageIterator Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**   - Used BLOBMaker and BLOBScanner to do page
**     buffer read and write
**   - disabled ctor(), cctor(), operator = ().
**     enfored that a page only can be created by a page file.
**   - made dtor() private and added Free() to destroy Page explicitly
**   - checked memory leaking.
*/


/*
** include files
*/
#include <stdlib.h>
#include <iostream.h>

#include "IndexPage.h"
#include "State.h"
#include "PageKey.h"
```

```
#include "MemPtr.h"
#include "PagePtr.h"
#include "Cursor.h"
#include "PageFile.h"
#include "BTreeException.h"
#include "Blob.h"


//
// Implementation of IndexPage class
//


//
// Service: IndexPage::IndexPage
// Description:
//   The only valid constructor
//
IndexPage::IndexPage(IndexPageFile& owner)
  : Page(owner), mbParentOfLeaf(FALSE), mpOwner(&owner)
{
  mLinks = new Pointer* [mOrder + 1];
  for (int scan = 0; scan <= mOrder; scan++) {
    mLinks[scan] = NULL;
  }
}


//
// Service: IndexPage::Expand
// Description:
//   Expands the indexpage so that let the indexpage have one
//   extra pagekey and link.
//
IndexPage* IndexPage::Expand ()
{
  IndexPage* index = new IndexPage (*mpOwner);
  if (index->doExpand(*this)) {
    return index;
  }
  return NULL;
}


//
// Service: IndexPage::doExpand
// Description:
//   Expands the indexpage so that let the indexpage have one
//   extra link. mOrder is incrmented in Page::doExpand().
//
Bool IndexPage::doExpand (const IndexPage& page)
{
  if (!Page::doExpand(page)) {
    return FALSE;
  }

  // create a pointer link array with one more element
  delete [] mLinks;
  mLinks = new Pointer* [mOrder+1];

  for (int scan = 0; scan <= page.mOrder; scan++) {
    mLinks[scan] = page.mLinks[scan];
  }
  mLinks[scan] = NULL;
  mbParentOfLeaf = page.mbParentOfLeaf;
  return TRUE;
}


//
// Service: IndexPage::~IndexPage
// Description:
//
IndexPage::~IndexPage()
{
  if(mLinks) {
    delete [] mLinks;
  }
}


//
// Service: IndexPage::Free
// Description:
//   Free Memory Space, destroy the sub-tree
//
void IndexPage::Free()
{
  for (int scan = 0; scan <= mNoOfKey; scan++) {
    mLinks[scan]->Free();
  }
  delete this;
}


//
// Service: IndexPage::operator []
// Description:
//   indexing
// Exceptions:
//   PageErrFindPositionExc
//
Cursor* IndexPage::operator [] (PageKey& aKey)
{
  int position;

  if(!getPosition(aKey, &position)) {
    throw PageErrFindPositionExc();
  }

  mLinks[position] = mLinks[position]->DeReference();
  return (*mLinks[position])[aKey]->AddThisPage(*(new Node(*this, position)));
}


//
// Service: IndexPage::SetLink
// Title:   Set Link at given position
// Description:
//   the Link would be downgraded to PagePtr
//
void IndexPage::SetLink(const int aPosition, Pointer* aPtr)
{
  assert(aPtr);
  LinkBoundCheck(aPosition);

  mLinks[aPosition] = aPtr;
  aPtr->SetParent(this);
}


//
```

```
// Service: IndexPage::SetLink
// Description:
// Exceptions:
//   BTreeErrNoServiceExc
//
void IndexPage::SetLink(const int, const String*)
{
  throw BTreeErrNoServiceExc();
}


//
// Service: IndexPage::GetLink
// Description:
// Exceptions:
//   BTreeErrOutOfBoundExc
//
const void* IndexPage::GetLink(const int aPosition) const
{
  LinkBoundCheck(aPosition);
  return (Pointer*)(mLinks[aPosition]);
}


//
// Service: IndexPage::GetPageFile
// Description:
//
PageFile* IndexPage::GetPageFile() const
{
  return mpOwner;
}


//
// Service: IndexPage::Insert
// Description:
//   inserts a PageKey and a pointer into IndexPage.
// Return:
//   an Bundle to hold all information about the insertion
// Arguments:
//   PageKey* key (in) - a pointer to a PageKey
//   void* ptr (in) - a pointer to a blob
// Note:
//   ptr != NULL
// Exceptions:
//   PageErrInsertExc
//   PageErrWriteRootExc
//   PageErrWriteSiblingExc
//   PageErrWritePageExc
//
State* IndexPage::Insert(const PageKey& aKey, Pointer* ptr)
{
  assert(ptr);

  //check if page is full
  if (IsFull()){
    //make a temp page, with one more space for Link and PageKey
    IndexPage* temppage = Expand();
    if (!temppage) {
      throw PageErrInsertExc();
    }

    //Insert key and ptr into the temp page
    if (!temppage->doInsert(aKey, ptr)) {
      throw PageErrInsertExc();
      }

      //split the temp page into this page, and a new sibling page
      MemPtr* sibling_mptr = split(*temppage);
      assert(sibling_mptr);

      //check if the page is a root of BTree
      if (IsRoot()) {
        //new an IndexPage as a new Root of BTree
        IndexPage* newRoot = new IndexPage(*mpOwner);

        //set newRoot links and set parent of the indexpage and
        //the new splitted sibling indexpage.
        newRoot->SetLink(0, (new MemPtr(*mpOwner, mOffset, *this)));
        newRoot->SetLink(1, sibling_mptr);

        //set newRoot pagekey
        newRoot->SetKey(0, temppage->GetKey(mOrder/2));
        newRoot->mNoOfKey = 1;

        //set newRoot is not a ParentOfLeaf
        newRoot->SetParentOfLeaf(FALSE);

        //write the newRoot page back
        FilePtr offset = mpOwner->Write(newRoot, FP_UNKNOWN);
        if (offset == FP_UNKNOWN) {
          throw PageErrWriteRootExc();
        }
        newRoot->SetOffset(offset);
        MemPtr* newRootPtr = new MemPtr(*mpOwner, offset, *newRoot);

        // write both indexpages
        // the offset of indexpage will not be changed
        offset = sibling_mptr->GetOffset();
        if (mpOwner->Write(sibling_mptr->GetPage(),
                           offset) != offset) {
          throw PageErrWriteSiblingExc();
        }

        if (mpOwner->Write(this, mOffset) != mOffset) {
throw PageErrWritePageExc();
        }
        return (new State(*newRootPtr, State::NEWROOT));
      }//_if_isroot_

      //resovle the child and parent relationship-transient
      sibling_mptr->SetParent(mpParent);
      return new State(*(temppage->GetKey(mOrder/2)),
                       *sibling_mptr, State::SPLIT);
    }//_if_isfull_
    else {
      if (!doInsert(aKey, ptr)) {
        throw PageErrInsertExc();
      }

      //write the indexpage, offset will not be changed
      if (mpOwner->Write(this, mOffset) != mOffset) {
        throw PageErrWritePageExc();
      }
      return (new State(State::COMPLETE));
    }
  }


  //
  // Service: IndexPage::Insert
// Title:   insert a pair of PageKey and pointer into IndexPage
// Description:
//   inserts a PageKey and a pointer into IndexPage.
// Exceptions:
//   BTreeErrNoServiceExc
//
State* IndexPage::Insert (const PageKey&, const String*){
  throw BTreeErrNoServiceExc();
  return (new State(State::ERROR));
}


//
// Service: IndexPage::doInsert
// Description:
//   the routine provides a facility to deal with
//   inserting a pair of pagekey and pointer into
//   a non-full indexpage.
//
Bool IndexPage::doInsert (const PageKey& aKey, Pointer* ptr)
{
  assert(ptr);
  int insert_pos = 0;

  getPosition(aKey, &insert_pos);

  //free a space at the insert_pos to insert the key and ptr
  for (int scan = (mNoOfKey - 1); scan >= insert_pos; scan--){
    SetKey(scan+1, mKeys[scan]);
    SetLink(scan+2, mLinks[scan+1]);
  }

  SetLink(insert_pos+1, ptr);
  SetKey(insert_pos, &aKey);
  mNoOfKey++;
  return TRUE;
}


//
// Service: IndexPage::split
// Title:   split a temp page into two pages
// Description:
// Exceptions:
//   PageErrWriteChildPageExc
//   PageErrWriteSiblingExc
//   PageErrWritePageExc
//
MemPtr* IndexPage::split(IndexPage& aIndexPage)
{
  MemPtr* child_mptr = NULL;
  FilePtr offset     = FP_UNKNOWN;
  mNoOfKey           = 0;

  // shift first half to the indexpage
  for (int scan = 0; scan < (mOrder/2); scan++) {
    mNoOfKey++;
    SetKey(scan, aIndexPage.mKeys[scan]);
    SetLink(scan, aIndexPage.mLinks[scan]);
  }
  SetLink(scan, aIndexPage.mLinks[scan]);

  while (scan < mOrder){
    mKeys[scan] = NULL;
    mLinks[++scan] = NULL;
  }
```

```
    // create a new sibling IndexPage
    IndexPage* newpage = new IndexPage(*mpOwner);

    // shift second half to the new sibling
    for (scan = 0; scan < (mOrder/2); scan++){
      newpage->mNoOfKey++;
      newpage->SetKey(scan, aIndexPage.mKeys[scan+mNoOfKey+1]);
      newpage->SetLink(scan, aIndexPage.mLinks[scan+mNoOfKey+1]);
    }
    newpage->SetLink(scan, aIndexPage.mLinks[scan+mNoOfKey+1]);

    while(scan < mOrder){
      newpage->mKeys[scan] = NULL;
      newpage->mLinks[++scan] = NULL;
    }
    newpage->mbParentOfLeaf = mbParentOfLeaf;

    // write the new sibling to indexpagefile
    offset = mpOwner->Write(newpage, FP_UNKNOWN);
    if (offset == FP_UNKNOWN) {
      throw PageErrWriteSiblingExc();
    }
    newpage->SetOffset(offset);

    // write back the IndexPage
    if(mpOwner->Write(this, mOffset) != mOffset) {
      throw PageErrWritePageExc();
    }
    return new MemPtr(*mpOwner, newpage->GetOffset(), *newpage);
}


//
// Service: IndexPage::Remove
// Description:
//   Remove a Link Pointer and a PageKey from the page
// Exceptions:
//   PageErrWritePageExc
//   PageErrWriteRootExc
//
State* IndexPage::Remove(const int aKeyPos,
                         const int aLinkPos,
                         const int aParentPos)
{
  // NOTE:
  // remove key and link -- MUST also discard the child page
  // space in child pagefile

  if (!doDelete(aKeyPos, aLinkPos)) {
    throw PageErrRemoveExc();
    return (new State(State::ERROR));
  }

  if ((mNoOfKey >= (mOrder/2)) || (IsRoot() && mNoOfKey > 0)) {
    //write the IndexPage back
    if(mpOwner->Write(this, mOffset) == FP_UNKNOWN) {
      throw PageErrWritePageExc();
    }
    return (new State(State::COMPLETE));
  }

  if (IsRoot() && mNoOfKey == 0) {
    // discard disk space of the old root in the indexpagefile.
    mpOwner->Discard(mOffset);
    // set the mpParent of the new root as NULL
    mLinks[0]->SetParent(NULL);
    return (new State(*((mLinks[0])->DeReference()), State::NEWROOT));
  }

  State::R_L aRL;

  //using pointer to avoid operator = making a copy of the sibling
  IndexPage* sibling = &(getSibling(&aRL, aParentPos));

  if(sibling->GetNoOfKey() > (mOrder/2)){
    reDistribute(*sibling, aParentPos, aRL);
    return (new State(State::COMPLETE));
  }

  merge(*sibling, aParentPos, aRL);
  return (new State(aRL, State::MERGE));
}


//
// Service: IndexPage::doDelete
// Description:
//     Remove a Link and a PageKey from a IndexPage, Also discard
//     the child page space in the child pagefile
//
Bool IndexPage::doDelete(int aKeyPos,
                         int aLinkPos)
{
  KeyBoundCheck(aKeyPos);
  LinkBoundCheck(aLinkPos);

  delete mLinks[aLinkPos];  // explicitly delete the sub-tree -- the page associated
                            // with the link is already be freed at child level

  while((mNoOfKey - 1 - aKeyPos) != 0) {
    SetKey(aKeyPos, mKeys[aKeyPos+1]);
    SetLink(aLinkPos, mLinks[aLinkPos+1]);
    aKeyPos++;
    aLinkPos++;
  }
  mKeys[aKeyPos] = NULL;

  if(aKeyPos < aLinkPos) {
    mLinks[aLinkPos] = NULL;
  }
  else {
    SetLink(aLinkPos, mLinks[aLinkPos+1]);
    aLinkPos++;
    mLinks[aLinkPos] = NULL;
  }
  mNoOfKey--;
  return TRUE;
}


//
// Service: IndexPage::merge
// Description:
// Exceptions:
//   PageErrWriteChildPageExc
//   PageErrWriteSiblingExc
//
IndexPage& IndexPage::merge(IndexPage& aSibling,
                            const int pPos, State::R_L aRL)
{
  int insert_pos;
  MemPtr* child_mptr = NULL;
  FilePtr offset = FP_UNKNOWN;

  if (aRL == State::LEFT) {
    insert_pos = aSibling.GetNoOfKey();
    aSibling.SetKey(insert_pos, mpParent->GetKey(pPos-1));
    aSibling.mNoOfKey++;
    insert_pos++;

    for (int scan = 0; scan < mNoOfKey; scan++) {
      aSibling.SetLink(insert_pos, mLinks[scan]);
      aSibling.SetKey(insert_pos, mKeys[scan]);
      aSibling.mNoOfKey++;
      insert_pos++;
    }
    aSibling.SetLink(insert_pos, mLinks[scan]);
  }
  else {
    insert_pos = mNoOfKey + 1 + aSibling.GetNoOfKey();

    aSibling.mNoOfKey = insert_pos;
    aSibling.SetLink(insert_pos, aSibling.mLinks[insert_pos-mNoOfKey-1]);

    for (int scan = mNoOfKey+1; scan > 0; scan--) {
      aSibling.SetLink(insert_pos-1, aSibling.mLinks[insert_pos - mNoOfKey -1-1]);
      aSibling.SetKey(insert_pos-1, aSibling.mKeys[insert_pos - mNoOfKey -1-1]);
      insert_pos--;
    }

    aSibling.SetKey(insert_pos-1, mpParent->GetKey(pPos));
    aSibling.SetLink(insert_pos-1, mLinks[mNoOfKey]);

    for (scan = 0; scan < mNoOfKey; scan++) {
      aSibling.SetLink(scan, mLinks[scan]);
      aSibling.SetKey(scan, mKeys[scan]);
    }
  }

  //write the sibling back to pagefile
  offset = aSibling.GetOffset();
  if(mpOwner->Write(&aSibling, offset) != offset) {
    throw PageErrWriteSiblingExc();
  }

  //deallocate the space of this page from pagefile
  mpOwner->Discard(mOffset);
  delete this;
  return aSibling;
}


//
// Service: IndexPage::reDistribute
// Description:
// Exceptions:
//   PageErrWriteSiblingExc
//   PageErrWritePageExc
//   PageErrWriteParentExc
//
void IndexPage::reDistribute(IndexPage& aSibling,
                             const int pPos, State::R_L aRL)
{
  int sib_NoOfKey = aSibling.GetNoOfKey();
  int scan = 0;

  //redistribute from LEFT sibling
```

```
    if (aRL == State::LEFT) {
      SetLink(mNoOfKey+1, mLinks[mNoOfKey]);
      for(scan = (mNoOfKey - 1); scan >= 0; scan--){
        SetKey(scan+1, mKeys[scan]);
        SetLink(scan+1, mLinks[scan]);
      }
      SetLink(0, aSibling.mLinks[sib_NoOfKey]);
      aSibling.mLinks[aSibling.GetNoOfKey()] = NULL;
      SetKey(0, mpParent->GetKey(pPos-1));

      mpParent->SetKey(pPos-1, aSibling.mKeys[sib_NoOfKey-1]);
      aSibling.mKeys[sib_NoOfKey-1] = NULL;
    }
    else {
      SetLink(mNoOfKey+1, aSibling.mLinks[0]);
      SetKey(mNoOfKey, mpParent->GetKey(pPos));
      mpParent->SetKey(pPos, aSibling.mKeys[0]);

      for (scan = 0; scan < sib_NoOfKey-1; scan++) {
        aSibling.SetLink(scan, aSibling.mLinks[scan+1]);
        aSibling.SetKey(scan, aSibling.mKeys[scan+1]);
      }
      aSibling.SetLink(scan, aSibling.mLinks[scan+1]);
      aSibling.mKeys[scan] = NULL;
      aSibling.mLinks[scan+1] = NULL;
    }
    mNoOfKey++;
    aSibling.mNoOfKey--;

    // write the parent page back
    PageFile* ppfile = mpOwner->GetParentPageFile();
    assert(ppfile);

    FilePtr offset = mpParent->GetOffset();
    if (ppfile->Write(mpParent, offset) == FP_MARKER) {
      throw PageErrWriteParentExc();
    }

    // write the IndexPage, sibling page back to pagefile
    offset = aSibling.GetOffset();
    if (mpOwner->Write(&aSibling, offset) == FP_MARKER) {
      throw PageErrWriteSiblingExc();
    }

    offset = GetOffset();
    if (mpOwner->Write(this, offset) == FP_MARKER) {
      throw PageErrWritePageExc();
    }
}


//
// Service: IndexPage::getSibling
// Description:
//
IndexPage& IndexPage::getSibling(State::R_L *aRL, const int pPos)
{
  assert(mpParent);

  if (pPos == 0) {
    *aRL = State::RIGHT;
    MemPtr* mptr = (MemPtr*)*((Pointer*)(mpParent->GetLink(pPos+1)));
    return *((IndexPage*)mptr->GetPage());
  }

  if (pPos == (mpParent->GetNoOfKey())) {
    *aRL = State::LEFT;
    MemPtr* mptr = (MemPtr*)*((Pointer*)(mpParent->GetLink(pPos-1)));
    return *(IndexPage*)(mptr->GetPage());
  }

  IndexPage *left, *right;
  MemPtr    *left_mptr, *right_mptr;

  left_mptr = (MemPtr*)*((Pointer*)(mpParent->GetLink(pPos-1)));
  right_mptr = (MemPtr*)*((Pointer*)(mpParent->GetLink(pPos+1)));

  assert(left_mptr && right_mptr);

  left = (IndexPage*)(left_mptr->GetPage());
  right = (IndexPage*)(right_mptr->GetPage());

  assert(left && right);

  if(right->GetNoOfKey() > left->GetNoOfKey()){
    *aRL = State::RIGHT;
    return *right;
  }
  *aRL = State::LEFT;
  return *left;
}


//
// Service: IndexPage::GetFirst
// Description:
//
Bool IndexPage::GetFirst(Page* & first)
{
  mLinks[0] = mLinks[0]->DeReference();
  first = (mLinks[0])->GetPage();
  return TRUE;
}


#ifdef _DEBUG
//
// Service: IndexPage::print
// Description:
//   Print out the content of the page
//
void IndexPage::print()
{
  Page::print();

  cout << "  < I n d e x P a g e > " << endl;
  cout << "    mbParentOfLeaf : ";
  if (mbParentOfLeaf) {
    cout << "TRUE" << endl;
  }
  else {
    cout << "FALSE" << endl;
  }

  cout << "    mpOwner : " << (mpOwner->GetFileName()) << endl;

  if (mNoOfKey > 0) {
    cout << "    mLinks  : " << endl;
    for (int scan = 0; scan <= mOrder; scan++) {
      if (mLinks[scan] == NULL) {
        cout << "      NULL" << endl;
      }
      else {
        mLinks[scan]->print();
      }
    }
  }
}
#endif


//
// Service: IndexPage::GetSize
// Description:
//   Calculate the actual size of IndexPage.
// Return:
//   Return an size of IndexPage.
// Arguments: None
// Note:
//   get the MAX fixed size of IndexPage. The size of
//   an indexpage wonot be changed.
//   Persistent attributes:
//     mLinks, mbParentOfLeaf
//
unsigned int IndexPage::GetSize() const
{
  unsigned int size = Page::GetSize();

  // mLinks pointers size
  size += sizeof(FilePtr) * (mOrder+1);

  // mbParentOfLeaf
  size += sizeof(Bool);
  return size;
}


//
// Service: IndexPage::GetPosition
// Description:
//   Return TRUE always.
//
Bool IndexPage::getPosition(const PageKey& aKey, int* aPos)
{
  for (*aPos = 0; *aPos < mNoOfKey; (*aPos)++) {
    if (*mKeys[*aPos] > aKey) {
      return TRUE;
    }
  }
  return TRUE;
}


//
// Service: IndexPage::LinkBoundCheck
// Description:
//
void IndexPage::LinkBoundCheck(const int aPos) const
{
  if(aPos < 0 || aPos > mOrder+1) {
    throw BTreeErrOutOfBoundExc();
  }
}


//
// Service: IndexPage::BufferWrite
// Description:
```

```
//   Write a IndexPage into a blob, the format:
//     Page:
//       int mNoOfKey
//       key[1]  key[2]  ... key[mNoOfKey]
//     IndexPage:
//       Boolean  mbParentOfLeaf
//       link[1] link[2] ... link[mNoOfKey+1]
// Return:
//   TRUE on success; otherwise, return FALSE.
// Note:
//   the size of buffer is the size of IndexPage
//
Bool IndexPage::BufferWrite(BLOBMaker& blob) const
{
  if (!Page::BufferWrite(blob)) {
    return FALSE;
  }

  //write mbParentOfLeaf, mLinks
  blob += (Bool)mbParentOfLeaf;

  FilePtr offset = FP_UNKNOWN;
  for (int scan = 0; scan <= mNoOfKey; scan++) {
    offset = mLinks[scan]->GetOffset();
    blob += (FilePtr)offset;
  }
  return TRUE;
}


//
// Service: IndexPage::BufferRead
// Description:
//   Read a buffer, to get the Page infomation and build the page.
// Return:
//   TRUE on success, otherwise FALSE.
// Note:
//
Bool IndexPage::BufferRead(BLOBScanner& blob)
{
  if (!Page::BufferRead(blob)) {
    return FALSE;
  }

  //read mbParentOfLeaf, mLinks
  mbParentOfLeaf = (Bool)blob;

  PageFile* child_pagefile = NULL;
  if (mbParentOfLeaf) {
    child_pagefile = mpOwner->GetChildPageFile();
  }
  else {
    child_pagefile = mpOwner;
  }

  if (!child_pagefile) {
    return FALSE;
  }

  //restore mLinks
  FilePtr offset;
  for (int scan = 0; scan <= mNoOfKey; scan++) {
    offset = (FilePtr)blob;
    SetLink(scan, new PagePtr(*child_pagefile, offset));
  }
  return TRUE;
}


//
// Service: IndexPage::CreateIterator
// Description:
//
Iterator* IndexPage::CreateIterator()
{
  return new IndexPageIterator(this);
}


//
// Service: IndexPage::IsParentOfLeaf
// Description:
//
Bool IndexPage::IsParentOfLeaf (void) const
{
  return mbParentOfLeaf;
}


//
// Service: IndexPage::SetParentOfLeaf
// Description:
//
void IndexPage::SetParentOfLeaf(Bool aFlag)
{
  mbParentOfLeaf = aFlag;
}


Bool IndexPage::GetPosition(Page& leaf, int& position)
{
  for (IndexPageIterator iter (this); !iter.IsDone(); iter.Next(), ++position) {
    try {
      if ((iter.page())->GetPage() == &leaf) {
        return TRUE;
      }
    }
    catch (BTreeErrNoServiceExc&) {
      // ... ignore it
    }
  }
  return FALSE;
}


//
// Implementation of IndexPageIterator class
//


//
// Service: IndexPageIterator::IndexPageIterator
// Description:
//   Constructor
//
IndexPageIterator::IndexPageIterator(IndexPage* apPage)
  : PageIterator(apPage)
{}


//
// Service: IndexPageIterator::~IndexPageIterator
// Description:
//   Destructor
//
IndexPageIterator::~IndexPageIterator()
{}


//
// Service: IndexPageIterator::First
// Description:
//
void IndexPageIterator::First ()
{
  mPosition = 0;
}


//
// Service: IndexPageIterator::Next
// Description:
//
void IndexPageIterator::Next ()
{
  mPosition++;
}


//
// Service: IndexPageIterator::IsDone
// Description:
//
Bool IndexPageIterator::IsDone ()
{
  return (mPosition > mKeys) ? TRUE : FALSE;
}


//
// Service: IndexPageIterator::page
// Description:
//
Pointer* IndexPageIterator::page () const
{
  return ((Pointer*)(mpPage->GetLink(mPosition)));
}


//
// Service: IndexPageIterator::blob
// Description:
//
String* IndexPageIterator::blob () const
{
  throw BTreeErrNoServiceExc();
  return NULL;
}
```

Appendix C

Definition and Implementation of Class LeafPage

C.1 Definition of Class LeafPage

```
/*
** LeafPage.h
**
** LeafPage Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**   - merged LeafPage and DataPage together
**   - Used BLOBMaker and BLOBScanner to do page
**     buffer read and write
**   - disabled ctor(), cctor(), operator = ().
**     enfored that a page only can be created by a page file.
**   - made dtor() private and added Free() to destroy Page explicitly
**   - checked memory leaking.
*/


#ifndef LEAFPAGE_H
#define LEAFPAGE_H


//
// forward declaration
//
class Iterator;
class IndexPage;
class String;


//
// include files
//
```

```
#include "Page.h"
#include "State.h"
#include "File.h"
#include "Boolean.h"


//
// Definition of LeafPage class
//

class LeafPage : public Page {
  friend class LeafPageFile;
public:
  LeafPage::~LeafPage();
  void Free();
  virtual Cursor* operator [](PageKey& aPageKey);   //index

  virtual void SetLink(const int aPosition, Pointer* apPtr);
  virtual void SetLink(const int aPosition, const String* apPtr);
  virtual const void* GetLink(const int aPosition) const;

  virtual Bool GetFirst(Page* &);
  virtual Bool getPosition(const PageKey& aPageKey, int* apPosition);

  virtual State* Insert(const PageKey& key, const String* ptr);
  virtual State* Insert(const PageKey& key, Pointer* ptr);
  virtual State* Remove(const int keyPos,
                        const int pointerPos,
                        const int parentPos);

  Pointer* GetNextLeaf();
  Pointer* GetPrevLeaf();
  void ResolveLinks ();
  virtual PageFile* GetPageFile() const;

  virtual unsigned int GetSize() const;
  virtual Bool BufferRead(BLOBScanner& blob);
  virtual Bool BufferWrite(BLOBMaker& blob) const;

  virtual Iterator* CreateIterator();
  Page* GetLoadedSibling(State::R_L aRL);
  Page* FindLoadedSibling(IndexPage& aIndex, State::R_L aRL);
  Bool GetPosition(Page& leaf, int& position);  //no service

#ifdef _DEBUG
  virtual void print();
#endif

private:
  String** mLinks;
  Pointer* mNextLeaf;
  Pointer* mPrevLeaf;
  LeafPageFile* mpOwner;
  Bool     mbNextSiblingResolved;
  Bool     mbPrevSiblingResolved;
private:
  LeafPage(LeafPageFile& owner);
  void LinkBoundCheck(const int pos) const;

  LeafPage* Expand ();
  Bool doExpand (const LeafPage&);
  Bool doInsert(const PageKey& aKey, const String* ptr);
  MemPtr* split(LeafPage& apLeafPage);

  Bool doDelete(int, int);
  LeafPage& getSibling(State::R_L*, const int aPosition);
  void reDistribute(LeafPage&, const int, State::R_L);
  LeafPage& merge(LeafPage& aSibling,
                  const int aParentPos,
                  State::R_L aRL);
private:
  LeafPage();                          // NOT DEFINED
  LeafPage& operator = (const LeafPage&); // NOT DEFINED
  LeafPage(const LeafPage& aLeafPage);    // NOT DEFINED
};


//
// Definition of LeafPageIterator
//

class LeafPageIterator : public PageIterator {
public:
  LeafPageIterator(Page* apPage);
  virtual ~LeafPageIterator();
  virtual void First ();
  virtual void Next ();
  virtual Bool IsDone ();
  virtual Pointer* page () const;
  virtual String* blob () const;
private:
  LeafPageIterator();                                      // NOT DEFINED
  LeafPageIterator(const LeafPageIterator&);               // NOT DEFINED
  LeafPageIterator& operator = (const LeafPageIterator&); // NOT DEFINED
};

#endif
```

C.2 Implementation of Class LeafPage

```
/*
** LeafPage.cxx
**
** LeafPage Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**   - merged LeafPage and DataPage together
**   - Used BLOBMaker and BLOBScanner to do page
**     buffer read and write
**   - disabled ctor(), cctor(), operator = ().
**     enfored that a page only can be created by a page file.
**   - made dtor() private and added Free() to destroy Page explicitly
**   - checked memory leaking.
*/


//
// include files
//

#include <stdlib.h>
```

```
#include <iostream.h>

#include "State.h"
#include "LeafPage.h"
#include "MemPtr.h"
#include "PagePtr.h"
#include "PageKey.h"
#include "Cursor.h"
#include "IndexPage.h"
#include "PageFile.h"
#include "BTreeException.h"
#include "BTreeFile.h"
#include "Blob.h"


//
// Implementation of LeafPage class
//


//
// Service: LeafPage::LeafPage
// Description:
//   Constructor
//
LeafPage::LeafPage(LeafPageFile& owner)
  : Page(owner), mNextLeaf(NULL), mPrevLeaf(NULL), mpOwner(&owner)
{
  mLinks = new String* [mOrder];
  for (int scan = 0; scan < mOrder; scan++) {
    mLinks[scan] = NULL;
  }
  mbNextSiblingResolved = FALSE;
  mbPrevSiblingResolved = FALSE;
}


//
// Service: LeafPage::Expand
// Description:
//   Expands the leafpage so that let the leafpage have one
//   extra pagekey and link.
//
LeafPage* LeafPage::Expand ()
{
  LeafPage* leaf = new LeafPage (*mpOwner);
  if (leaf->doExpand(*this)) {
    return leaf;
  }
  return NULL;
}


//
// Service: LeafPage::doExpand
// Description:
//   Expands the leafpage so that let the leafpage have one
//   extra link. mOrder is incrmented in Page::doExpand().
//
Bool LeafPage::doExpand (const LeafPage& page)
{
  if (!Page::doExpand(page)) {
    return FALSE;
  }

  // create a pointer link array with one more element
```

```
    delete [] mLinks;
    mLinks = new String* [mOrder];

    for (int scan = 0; scan < page.mOrder; scan++) {
      mLinks[scan] = new String(*page.mLinks[scan]);
    }
    mLinks[scan] = NULL;   // the last new element NULL

    mNextLeaf = page.mNextLeaf;
    mPrevLeaf = page.mPrevLeaf;
    return TRUE;
  }


  //
  // Service: LeafPage::~LeafPage
  // Description:
  //   Destructor
  //
  LeafPage::~LeafPage()
  {
    if (mLinks) {
      delete [] mLinks;
    }
  }


  //
  // Service: LeafPage::Free
  // Description:
  //   Free Memory Space
  //
  void LeafPage::Free()
  {
    for (int scan = 0; scan < mNoOfKey; scan++) {
      delete mLinks[scan];
    }

    // resolve the NEXT and PREV linkages
    ResolveLinks();
    delete this;
  }


  //
  // Service: LeafPage::operator []
  // Description:
  // Exceptions:
  //   PageErrFindPositionExc
  //
  Cursor* LeafPage::operator [] (PageKey& aKey)
  {
    int  iPosition = 0;
    Bool ibFlag = getPosition(aKey, &iPosition);

    return (new Cursor(ibFlag,
       *(new Node(*this, iPosition)), aKey));
  }


  //
  // Service: LeafPage::SetLink
  // Description:
  // Exceptions:
  //   BTreeErrNoServiceExc
  //
```

```
void LeafPage::SetLink(const int, Pointer*)
{
  throw BTreeErrNoServiceExc();
}


//
// Service: LeafPage::SetLink
// Description:
//
void LeafPage::SetLink(const int aPosition, const String* record)
{
  assert(record);
  LinkBoundCheck(aPosition);

  if (mLinks[aPosition]) {
    delete mLinks[aPosition];
    mLinks[aPosition] = NULL;
  }

  mLinks[aPosition] = new String ();
  *mLinks[aPosition] = *((String*)record);
}


//
// Service: LeafPage::GetLink
// Description:
// Exceptions:
//   BTreeErrOutOfBoundExc
//
const void* LeafPage::GetLink(const int aPosition) const
{
  LinkBoundCheck(aPosition);
  return (String*)(mLinks[aPosition]);
}


//
// Service: LeafPage::GetNextLeaf
// Description:
//   If the loaded next sibling already found and been pointed by
//   the mNextLeaf, the mbSiblingLinkResolved should be set
//   TRUE, and this routine simple returns the mNextLeaf (MemPtr).
//
//   If the mbSiblingLinkResolved is set as FALSE, GetNextLeaf searches
//   for a loaded next sibling first.
//
//   If the loaded next sibling found, sets the mbSiblingLinkResolved
//   to TRUE, and the NextLeaf is replaced by a MemPtr which points to
//   the found loaded next sibling. Otherwise, keep the mbSiblingLinkResolved
//   FALSE, and DeReference the mNextLeaf to MemPtr.
//
//   The idea here is to prevent duplicated inconsistent copies of a leaf page
//   from btree. There are two Pointers refer to one identical leaf page.

Pointer* LeafPage::GetNextLeaf()
{
  if (mbNextSiblingResolved || !mNextLeaf) {
    return mNextLeaf;
  }

  // searches for loaded next sibling
  LeafPage* next = (LeafPage*)GetLoadedSibling(State::RIGHT);

  if (!next) {
    mbNextSiblingResolved = FALSE;
    mNextLeaf = mNextLeaf->DeReference();
    return mNextLeaf;
  }

  delete mNextLeaf;
  mNextLeaf = new MemPtr (*mpOwner, next->GetOffset(), *next);
  mbNextSiblingResolved = TRUE;
  return mNextLeaf;
}


//
// Service: LeafPage::GetPrevLeaf
// Description:
//   See LeafPage::GetNextLeaf ()
//
Pointer* LeafPage::GetPrevLeaf()
{
  if (mbPrevSiblingResolved || !mPrevLeaf) {
    return mPrevLeaf;
  }

  // searches for loaded next sibling
  LeafPage* prev = (LeafPage*)GetLoadedSibling(State::LEFT);

  if (!prev) {
    mbPrevSiblingResolved = FALSE;
    mPrevLeaf = mPrevLeaf->DeReference();
    return mPrevLeaf;
  }

  delete mPrevLeaf;
  mPrevLeaf = new MemPtr (*mpOwner, prev->GetOffset(), *prev);
  mbPrevSiblingResolved = TRUE;
  return mPrevLeaf;
}


//
// Service: LeafPage::GetPageFile
// Description:
//
PageFile* LeafPage::GetPageFile() const
{
  return mpOwner;
}


//
// Service: LeafPage::ResolveLinks
// Description:
//
void LeafPage::ResolveLinks ()
{
  if (mPrevLeaf) {
    mPrevLeaf = GetPrevLeaf();
    LeafPage* prev = (LeafPage*)(((MemPtr*)(*mPrevLeaf))->GetPage());

    prev->mNextLeaf = mNextLeaf;
    prev->mbNextSiblingResolved = mbNextSiblingResolved;

    FilePtr offset = prev->GetOffset();
    if (mpOwner->Write(prev, offset) != offset) {
      throw PageErrWritePageExc();
    }
  }
  if (mNextLeaf) {
    mNextLeaf = GetNextLeaf();
    LeafPage* next = (LeafPage*)(((MemPtr*)(*mNextLeaf))->GetPage());

    next->mPrevLeaf = mPrevLeaf;
    next->mbPrevSiblingResolved = mbPrevSiblingResolved;

    FilePtr offset = next->GetOffset();
    if (mpOwner->Write(next, offset) != offset) {
      throw PageErrWritePageExc();
    }
  }
}


//
// Service: LeafPage::Insert
// Description:
// Exceptions:
//   PageErrInsertExc
//   PageErrWriteRootExc
//   PageErrWriteSiblingExc
//   PageErrWritePageExc
//
State* LeafPage::Insert(const PageKey& aKey, const String* rec)
{
  assert(rec);

  //check if the LeafPage is Full
  if(IsFull()){
    //make a temp page with one more space for Link and PageKey
    LeafPage* temppage = Expand();
    if (!temppage) {
      throw PageErrInsertExc();
    }

    //insert key and rec into the temp page
    if (!temppage->doInsert(aKey, (String*)rec)) {
      throw PageErrInsertExc();
    }

    //split the temp page into this page and a new sibling page
    MemPtr* sibling_mptr = split(*temppage);
    assert(sibling_mptr);

    //check if this page is a BTree root page
    if (IsRoot()) {
      //new a IndexPage as a new Root of BTree
      IndexPageFile* indexfile = mpOwner->GetOwner()->GetIndexPageFile();
      IndexPage* newroot = indexfile->MakeIndexPage();

      //set newroot links and set parent of the leafpage and
      //the new splitted sibling leafpage.
      newroot->SetLink(0, (new MemPtr(*mpOwner, mOffset, *this)));
      newroot->SetLink(1, sibling_mptr);

      //set newroot pagekey
      newroot->SetKey(0, sibling_mptr->GetPage()->GetKey(0));
      newroot->SetNoOfKey(1);

      //set newroot is a ParentOfLeaf
      newroot->SetParentOfLeaf(TRUE);

      //write the newroot page back
      FilePtr offset = indexfile->Write(newroot, FP_UNKNOWN);
      if (offset == FP_UNKNOWN) {
        throw PageErrWriteRootExc();
      }
      newroot->SetOffset(offset);
      MemPtr* newRootPtr = new MemPtr(*indexfile, offset, *newroot);

      //write both leafpages
      //the offset of leafpage will not be changed.
      offset = sibling_mptr->GetOffset();
      if (mpOwner->Write(sibling_mptr->GetPage(),
                         offset) != offset) {
        throw PageErrWriteSiblingExc();
      }

      if (mpOwner->Write(this, mOffset) != mOffset) {
throw PageErrWritePageExc();
      }
      return (new State(*newRootPtr, State::NEWROOT));
    } //_if_isroot_

    //resolve the child and parent relationship-transient
    sibling_mptr->SetParent(mpParent);
    return (new State(*(temppage->GetKey(mOrder/2)),
                      *sibling_mptr, State::SPLIT));
    //not finished yet, insert into parent page
  } //_if_isfull_
  else {
    if (!doInsert(aKey, (String*)rec)) {
      throw PageErrInsertExc();
    }

    //write the leafpage back, offset change allowed
    //if offset changed, update its parent link.
    //and if the page is a root, so it is considered as
    //a root update.

    FilePtr offset = FP_UNKNOWN;
    offset = mpOwner->Write(this, mOffset);

    if (offset == FP_UNKNOWN) {
      throw PageErrWritePageExc();
    }
    if (offset != mOffset) {
      mOffset = offset;

      if (IsRoot()) {
        return (new State(*(new MemPtr(*mpOwner, offset, *this)),
                          State::UPDATEROOT));
      }

      // update page pointer in parent page
      int aLinkPos = 0;
      mpParent->getPosition(aKey, &aLinkPos);
      ((Pointer*)(mpParent->GetLink(aLinkPos)))->SetOffset(mOffset);

      // write the parent page back to its pagefile
      PageFile* indexfile = mpOwner->GetParentPageFile();
      offset = mpParent->GetOffset();
      if (indexfile->Write(mpParent, offset) != offset) {
        throw PageErrWriteParentExc();
      }
    }
    return (new State(State::COMPLETE));
    //finish the insertion
  }
}
```

```
//
// Service: LeafPage::Insert
// Description:
//
State* LeafPage::Insert(const PageKey&, Pointer*)
{
  throw BTreeErrNoServiceExc();
  return (new State(State::ERROR));
}


//
// Service: LeafPage::split
// Description:
//   split a temp page into two pages
// Exceptions:
//   PageErrWriteChildPageExc
//   PageErrWriteSiblingExc
//   PageErrWritePageExc
//
MemPtr* LeafPage::split(LeafPage& aLeafPage)
{
  MemPtr* child_mptr = NULL;
  FilePtr offset     = FP_UNKNOWN;
  mNoOfKey           = 0;

  // shift first half to the leafpage
  for (int scan = 0; scan < (mOrder/2); scan++) {
    mNoOfKey++;
    SetKey(scan, aLeafPage.mKeys[scan]);
    SetLink(scan, aLeafPage.mLinks[scan]);
  }

  while (scan < mOrder){
    mKeys[scan] = NULL;
    mLinks[scan] = NULL;
    scan++;
  }

  // create a new sibling LeafPage
  LeafPage* newpage = new LeafPage(*mpOwner);

  // shift second half to the new sibling
  for (scan = 0; scan < (mOrder/2)+1; scan++){
    newpage->mNoOfKey++;
    newpage->SetKey(scan, aLeafPage.mKeys[scan+mNoOfKey]);
    newpage->SetLink(scan, aLeafPage.mLinks[scan+mNoOfKey]);
  }

  while(scan < mOrder){
    newpage->mKeys[scan] = NULL;
    newpage->mLinks[scan] = NULL;
    scan++;
  }

  // write the new sibling to leafpagefile
  offset = mpOwner->Write(newpage, FP_UNKNOWN);
  if (offset == FP_UNKNOWN) {
    throw PageErrWriteSiblingExc();
  }
  newpage->SetOffset(offset);

  //link the new sibling with Prev and Next links
  if (mNextLeaf) {
    // check if the next sibling is loaded by its parent
    mNextLeaf = GetNextLeaf();
    newpage->mNextLeaf = mNextLeaf;
    newpage->mbNextSiblingResolved = mbNextSiblingResolved;

    mNextLeaf = (MemPtr*)(*mNextLeaf);
    LeafPage* leafpage = (LeafPage*)(mNextLeaf->GetPage());

    delete leafpage->mPrevLeaf;
    leafpage->mPrevLeaf = new MemPtr(*mpOwner, newpage->GetOffset(),
     *newpage);
    leafpage->mbPrevSiblingResolved = TRUE;

    offset = leafpage->GetOffset();
    if (mpOwner->Write(leafpage, offset) != offset) {
      throw PageErrWriteSiblingExc();
    }
  }

  mNextLeaf = new MemPtr(*mpOwner, newpage->GetOffset(), *newpage);
  mbNextSiblingResolved = TRUE;

  newpage->mPrevLeaf = new MemPtr(*mpOwner, mOffset, *this);
  newpage->mbPrevSiblingResolved = TRUE;

  //write back this LeafPage and the new sibling
  if (mpOwner->Write(this, mOffset) != mOffset) {
    throw PageErrWritePageExc();
  }

  //second writing the new sibling
  offset = newpage->GetOffset();
  if (mpOwner->Write(newpage, offset) != offset) {
    throw PageErrWriteSiblingExc();
  }
  return new MemPtr(*mpOwner, offset, *newpage);
}


//
// Service: LeafPage::Remove
// Description:
//   Remove a Link and a PageKey from a LeafPage
// Exceptions:
//   PageErrWritePageExc
//   PageErrWriteRootExc
//
State* LeafPage::Remove(const int aKeyPos,
                         const int aLinkPos,
                         const int aParentPos)
{
  // NOTE:
  // remove key and link -- MUST also discard the child page
  // space in child pagefile

  if(!doDelete(aKeyPos, aLinkPos)) {
    throw PageErrRemoveExc();
    return (new State(State::ERROR));
  }
  if((mNoOfKey >= (mOrder/2)) || IsRoot()){
    //write the LeafPage back
    if(mpOwner->Write(this, mOffset) == FP_UNKNOWN) {
      throw PageErrWritePageExc();
    }
    return (new State(State::COMPLETE));
  }
  //re-distribute with its sibling which hold more Links
  State::R_L aRL;

  //using pointer to avoid operator = making a copy of the sibling
  LeafPage* sibling = &(getSibling(&aRL, aParentPos));

  if (sibling->GetNoOfKey() > (mOrder/2)) {
    reDistribute(*sibling, aParentPos, aRL);
    return (new State(State::COMPLETE));
  }

  //with its sibling if none of them
  //could be re-distributed
  merge(*sibling, aParentPos, aRL);
  return (new State(aRL, State::MERGE));
}


//
// Service: LeafPage::doDelete
// Description:
//     Remove a Link and a PageKey from a LeafPage.
//
Bool LeafPage::doDelete(int aKeyPos, int aLinkPos)
{
  KeyBoundCheck(aKeyPos);
  LinkBoundCheck(aLinkPos);

  while((mNoOfKey - 1 - aKeyPos) != 0) {
    SetKey(aKeyPos, mKeys[aKeyPos+1]);
    SetLink(aLinkPos, mLinks[aLinkPos+1]);
    aKeyPos++;
    aLinkPos++;
  }

  delete mKeys[aKeyPos];   // delete the page key
  delete mLinks[aLinkPos]; // delete the content blob
  mKeys[aKeyPos] = NULL;
  mLinks[aLinkPos] = NULL;
  mNoOfKey--;
  return TRUE;
}


//
// Service: LeafPage::getSibling
// Description:
//
LeafPage& LeafPage::getSibling(State::R_L *aRL, const int aParentPos)
{
  assert(mpParent);

  if(aParentPos == 0) {
    *aRL = State::RIGHT;
    MemPtr* mptr = (MemPtr*)*((Pointer*)(mpParent->GetLink(aParentPos+1)));
    return *((LeafPage*)mptr->GetPage());
  }

  if(aParentPos == (mpParent->GetNoOfKey())){
    *aRL = State::LEFT;
    MemPtr* mptr = (MemPtr*)*((Pointer*)(mpParent->GetLink(aParentPos-1)));
    return *(LeafPage*)(mptr->GetPage());
  }

  LeafPage *left, *right;
  MemPtr   *left_mptr, *right_mptr;

  left_mptr = (MemPtr*)*((Pointer*)(mpParent->GetLink(aParentPos-1)));
  right_mptr = (MemPtr*)*((Pointer*)(mpParent->GetLink(aParentPos+1)));

  assert(left_mptr && right_mptr);

  left = (LeafPage*)(left_mptr->GetPage());
  right = (LeafPage*)(right_mptr->GetPage());

  assert(left && right);

  if(right->GetNoOfKey() > left->GetNoOfKey()){
    *aRL = State::RIGHT;
    return *right;
  }

  *aRL = State::LEFT;
  return *left;
}


//
// Service: LeafPage::merge
// Description:
//   Moving all conent in the Page into aSibling Page,
//   and Writing aSibling Page into pagefile.
//   The offset of the sibling page would be changed,
//   if so, the parent of the sibling page must be updated.
//   Parent of the sibling will not write back at this
//   moment, because a pair of a pointer and pagekey will be
//   removed from parent
// Exceptions:
//   PageErrWriteChildPageExc
//   PageErrGetSiblingExc
//   PageErrWriteSiblingExc
//
LeafPage& LeafPage::merge(LeafPage& aSibling,
                          const int aParentPos, State::R_L aRL)
{
  int insert_pos;

  if(aRL == State::LEFT){
    //merge with a LEFT sibling
    insert_pos = aSibling.GetNoOfKey();
    for (int scan = 0; scan < mNoOfKey; scan++) {
      aSibling.SetLink(insert_pos, mLinks[scan]);
      aSibling.SetKey(insert_pos, mKeys[scan]);
      insert_pos++;
    }
  } //_aRL_LEFT_
  else {
    //merge with a RIGHT sibling
    if (mNoOfKey > 0) {
      insert_pos = mNoOfKey - 1 + aSibling.GetNoOfKey();

      for (int scan = aSibling.GetNoOfKey(); scan > 0; scan--) {
        aSibling.SetLink(insert_pos, aSibling.mLinks[insert_pos - mNoOfKey]);
        aSibling.SetKey(insert_pos, aSibling.mKeys[insert_pos - mNoOfKey]);
        insert_pos--;
      }

      for (scan = 0; scan < mNoOfKey; scan++) {
        aSibling.SetLink(scan, mLinks[scan]);
        aSibling.SetKey(scan, mKeys[scan]);
      }
    }
  }

  aSibling.mNoOfKey += mNoOfKey;

  //write the sibling back to pagefile
  FilePtr offset = aSibling.GetOffset();
  FilePtr new_offset = mpOwner->Write(&aSibling, offset);

  if (new_offset == FP_UNKNOWN) {
    throw PageErrWriteSiblingExc();
  }

  if (new_offset != offset) {
    aSibling.SetOffset(new_offset);

    //update parent Link Pointer
    //get the sibling link position in parent
    int link_pos = 0;
    if (aRL == State::LEFT) {
      link_pos = aParentPos - 1;
      mPrevLeaf->SetOffset(new_offset);
    }
    else {
      link_pos = aParentPos + 1;
      mNextLeaf->SetOffset(new_offset);
    }

    ((Pointer*)(mpParent->GetLink(link_pos)))->SetOffset(new_offset);
  }

  //resolve LeafPage prev and next links
  ResolveLinks();

  //deallocate the space of this page from pagefile
  mpOwner->Discard(mOffset);
  delete this;
  return aSibling;
}


//
// Service: LeafPage::reDistribute
// Description:
// Exceptions:
//   PageErrWritePageExc
//   PageErrWriteParentExc
//   PageErrWriteSiblingExc
//
void LeafPage::reDistribute(LeafPage& aSibling,
                            const int aParentPos, State::R_L aRL)
{
  int sib_NoOfKey = aSibling.GetNoOfKey();
  int scan = 0;

  //redistribute from LEFT sibling
  if (aRL == State::LEFT) {
    for (scan = (mNoOfKey - 1); scan >= 0; scan--) {
      SetKey(scan+1, mKeys[scan]);
      SetLink(scan+1, mLinks[scan]);
    }

    SetLink(0, aSibling.mLinks[sib_NoOfKey-1]);
    SetKey(0, aSibling.mKeys[sib_NoOfKey-1]);
    mpParent->SetKey(aParentPos-1, mKeys[0]);
    aSibling.mLinks[sib_NoOfKey-1] = NULL;
```

```
      aSibling.mKeys[sib_NoOfKey-1] = NULL;
    }
    else {
      SetLink(mNoOfKey, aSibling.mLinks[0]);
      SetKey(mNoOfKey, aSibling.mKeys[0]);

      for (scan = 0; scan < sib_NoOfKey-1; scan++) {
        aSibling.SetLink(scan, aSibling.mLinks[scan+1]);
        aSibling.SetKey(scan, aSibling.mKeys[scan+1]);
      }

      aSibling.mKeys[scan] = NULL;
      aSibling.mLinks[scan] = NULL;
      mpParent->SetKey(aParentPos, aSibling.mKeys[0]);
    }
    mNoOfKey++;
    aSibling.mNoOfKey--;

    //write this page, sibling page and parent page back to pagefile
    FilePtr offset = aSibling.GetOffset();
    FilePtr new_offset = mpOwner->Write(&aSibling, offset);
    if (new_offset == FP_UNKNOWN) {
      throw PageErrWriteSiblingExc();
    }
    if (new_offset != offset) {
      aSibling.SetOffset(new_offset);

      //update parent Link Pointer
      //get the sibling link position in parent
      int link_pos;
      if (aRL == State::LEFT) {
        link_pos = aParentPos - 1;
      }
      else {
        link_pos = aParentPos + 1;
      }
      ((Pointer*)(mpParent->GetLink(link_pos)))->SetOffset(new_offset);
    }

    //then write this page back to pagefile
    offset = GetOffset();
    new_offset = mpOwner->Write(this, offset);

    if (new_offset == FP_UNKNOWN) {
      throw PageErrWritePageExc();
    }

    if (new_offset != offset) {
      SetOffset(new_offset);

      //update parent Link Pointer
      ((Pointer*)(mpParent->GetLink(aParentPos)))->SetOffset(new_offset);
    }

    //and write the parent back to pagefile
    offset = mpParent->GetOffset();
    PageFile* indexfile = mpOwner->GetParentPageFile();
    if(indexfile->Write(mpParent, offset) != offset) {
      throw PageErrWriteParentExc();
    }
}


//
// Service: LeafPage::GetFirst
// Description:
```

```
//   Get First Link Page
Bool LeafPage::GetFirst(Page* &)
{
  return FALSE;
}


//
// Service: LeafPage::GetLoadedSibling
// Description:
//   Get loaded leaf sibling. If the parent is a null pointer, it returns a
//   null pointer to indicate that there is no leaf(s) been loaded by the
//   null parent.
//   Note: a leaf can be de-referenced by its siblings. in this case, the
//         parent pointer of the leaf is a null pointer.
Page* LeafPage::GetLoadedSibling(State::R_L aRL)
{
  // check if this LeafPage is the Root
  if (mpParent == NULL) {
      return NULL;
  }

  // check if this LeafPage is the First LeafPage
  if ((aRL == State::LEFT) && (mPrevLeaf == NULL)) {
    return NULL;
  }

  // check if this LeafPage is the Last LeafPage
  if ((aRL == State::RIGHT) && (mNextLeaf == NULL)) {
    return NULL;
  }

  // check out the position of this LeafPage in its parent
  int position = 0;

  if (!mpParent->GetPosition (*this, position)) {
    throw PageErrGetSiblingExc();
  }

  // find a Left sibling
  if (aRL == State::LEFT) {
    if (position > 0) {
      try {
        return ((Pointer*)(mpParent->GetLink(position-1)))->GetPage();
      }
      catch (BTreeErrNoServiceExc&) {
        return NULL;    // the sibling is not loaded
      }
    }
    else {
      return FindLoadedSibling(*mpParent, aRL); // from different parent
    }
  }

  // find a Right sibling
  else {
    if (position < mpParent->GetNoOfKey()) {
      try {
        return ((Pointer*)(mpParent->GetLink(position+1)))->GetPage();
      }
      catch (BTreeErrNoServiceExc&) {
        return NULL;    // the sibling is not loaded
      }
    }
    else {
      return FindLoadedSibling(*mpParent, aRL); // from different parent
    }
  }
  return NULL;
}


Page* LeafPage::FindLoadedSibling(IndexPage& aIndex, State::R_L aRL)
{
  Page*      pPage   = NULL;
  IndexPage* pParent = &aIndex;
  PageKey*   pKey    = (PageKey*)(aIndex.GetKey(0));
  int iPosition = 0;

  while (( pParent = pParent->GetParent()) != NULL) {

    pParent->getPosition(*pKey, &iPosition);
    if (aRL == State::LEFT) {
      if (iPosition > 0) { // found the turn point
        try {
          pParent = (IndexPage*)((Pointer*)(pParent->GetLink(iPosition-1)))->GetPage();
          while (!pParent->IsParentOfLeaf()) {
            pParent = (IndexPage*)((Pointer*)(pParent->GetLink(pParent->GetNoOfKey())))
      ->GetPage();
          }
          return ((Pointer*)(pParent->GetLink(pParent->GetNoOfKey())))->GetPage();
        }
        catch (BTreeErrNoServiceExc&) {
          return NULL;  // the sibling is not loaded
        }
      }
    }
    else {
      if (iPosition < pParent->GetNoOfKey()) { // found the turn point
        try {
          pParent = (IndexPage*)((Pointer*)(pParent->GetLink(iPosition+1)))->GetPage();
          while (!pParent->IsParentOfLeaf()) {
            pParent = (IndexPage*)((Pointer*)(pParent->GetLink(0)))->GetPage();
          }
          return ((Pointer*)(pParent->GetLink(0)))->GetPage();
        }
        catch (BTreeErrNoServiceExc&) {
          return NULL;  // the sibling is not loaded
        }
      }
    }
    pKey = (PageKey*)(pParent->GetKey(0));
  }
  return NULL;
}


#ifdef _DEBUG
//
// Service: LeafPage::print
// Description:
//   Print out the content of the page
//
void LeafPage::print()
{
  Page::print();

  cout << "  < L e a f P a g e > " << endl;
  cout << "    mpOwner : " << (mpOwner->GetFileName()) << endl;

  if (mNoOfKey > 0) {
    cout << "    mLinks    : " << endl;
    for (int scan = 0; scan < mOrder; scan++) {
      if (mLinks[scan] == NULL) {
        cout << "      NULL" << endl;
      }
      else {
        cout << "      " << *mLinks[scan] << endl;
      }
    }
  }

  cout << "    mNextLeaf : " << endl;
  if (mNextLeaf == NULL) {
    cout << "      NULL " << endl;
  }
  else {
    mNextLeaf->print();
  }

  cout << "    mPrevLeaf : " << endl;
  if (mPrevLeaf == NULL) {
    cout << "      NULL " << endl;
  }
  else {
    mPrevLeaf->print();
  }
}
#endif


//
// Service: LeafPage::GetSize
// Description:
//   Calculate the actual size of LeafPage.
// Return:
//   return an size of LeafPage.
// Arguments: None
// Persistent attributes:
//   mLinks, mNextLeaf, mPrevLeaf
//
unsigned int LeafPage::GetSize() const
{
  unsigned int size = Page::GetSize();

  //size of left and right pointers
  size += sizeof(FilePtr) * (2);

  //get size of data blob(s)
  for (int scan = 0; scan < mNoOfKey; scan++) {
    size += mLinks[scan]->GetLen() + 1;
  }
  return size;
}


//
// Service: LeafPage::getPosition
// Description:
//   Get Indexing Position
//
Bool LeafPage::getPosition(const PageKey& aKey, int* aPosition)
{
  for (*aPosition = 0; *aPosition < mNoOfKey; ++(*aPosition)) {
    if (*mKeys[*aPosition] == aKey) {
      return TRUE;
    }
    else if (*mKeys[*aPosition] > aKey) {
      return FALSE;
    }
  }
  return FALSE;
}


//
// Service: LeafPage::LinkBoundCheck
// Description:
//
void LeafPage::LinkBoundCheck(const int aPos) const
{
  if(aPos < 0 || aPos > mOrder) {
    throw BTreeErrOutOfBoundExc();
  }
}


//
// Service: LeafPage::BufferWrite()
// Description:
//   Write a LeafPage into a blob, the LeafPage format:
//     Page:
//       int mNoOfKey
//       key[1] key[2]  ... key[mNoOfKey]
//     LeafPage:
//       FilePtr offset of leftpage
//       FilePtr offset of rightpage
//       link[1] link[2] ... link[mNoOfKey]
// Return:
//   TRUE on success; otherwise, return FALSE.
// Note:
//   the size of buffer is the size of LeafPage
//
Bool LeafPage::BufferWrite(BLOBMaker& blob) const
{
  if (!Page::BufferWrite(blob)) {
    return FALSE;
  }

  //write Offsets of Left and Right Leaf page
  if (mPrevLeaf) {
    blob += (FilePtr)(mPrevLeaf->GetOffset());
  }
  else {
    blob += (FilePtr)FP_UNKNOWN;
  }

  if (mNextLeaf) {
    blob += (FilePtr)(mNextLeaf->GetOffset());
  }
  else {
    blob += (FilePtr)FP_UNKNOWN;
  }

  //write data blobs into blob buffer
  const char* content = NULL;
  for (int scan = 0; scan < mNoOfKey; scan++) {
    content = (const char*)(*mLinks[scan]);
    blob += (char*) content;
  }
  return TRUE;
}
```

```
//
// Service: LeafPage::BufferRead
// Description:
//   Read a buffer blob, to get the Page infomation and
//   build the page
// Return:
//   TRUE on success, otherwise FALSE.
// Note:
//
Bool LeafPage::BufferRead(BLOBScanner& blob)
{
  if (!Page::BufferRead (blob)) {
    return FALSE;
  }

  FilePtr offset = FP_UNKNOWN;

  //read Pointers of left and right siblings
  offset = (FilePtr)blob;
  if (offset == FP_UNKNOWN) {
    mPrevLeaf = NULL;
  }
  else {
    mPrevLeaf = new PagePtr(*mpOwner, offset);
  }

  offset = (FilePtr)blob;
  if (offset == FP_UNKNOWN) {
    mNextLeaf = NULL;
  }
  else {
    mNextLeaf = new PagePtr(*mpOwner, offset);
  }

  //restore content blobs
  //read mLinks from buffer
  char* tstring = NULL;
  for (int scan = 0; scan < mNoOfKey; scan++) {
    tstring = (char*)blob;
    SetLink(scan, new String(tstring));
    delete [] tstring;
  }
  return TRUE;
}


//
// Service: LeafPage::doInsert
// Description:
//   Do the simply insertion into a LeafPage
//
Bool LeafPage::doInsert(const PageKey& aKey, const String* aRec)
{
  assert(aRec);
  int insert_pos = 0;

  getPosition(aKey, &insert_pos);

  //free a space at the insert_pos to insert the key and rec
  if (mNoOfKey > 0) {
    for (int scan = (mNoOfKey - 1); scan >= insert_pos; scan--) {
      SetKey(scan+1, mKeys[scan]);
      SetLink(scan+1, mLinks[scan]);
    }
  }
  SetLink(insert_pos, aRec);
  SetKey(insert_pos, &aKey);
  mNoOfKey++;
  return TRUE;
}


//
// Service: LeafPage::CreateIterator
// Description:
//
Iterator* LeafPage::CreateIterator()
{
  return new LeafPageIterator(this);
}


Bool LeafPage::GetPosition(Page& , int&)
{
  throw BTreeErrNoServiceExc();
  return FALSE;
}


//
// Implementation of LeafPageIterator class
//


//
// Service: LeafPageIterator::LeafPageIterator
// Description:
//   Constructor
//
LeafPageIterator::LeafPageIterator(Page* apPage)
  : PageIterator(apPage)
{}


/*
** Service: LeafPageIterator::~LeafPageIterator
** Description:
**   Destructor
*/
LeafPageIterator::~LeafPageIterator()
{}


//
// Service: LeafPageIterator::First
// Description:
//
void LeafPageIterator::First ()
{
  mPosition = 0;
}


//
// Service: LeafPageIterator::Next
// Description:
//
void LeafPageIterator::Next ()
{
  mPosition++;
}
```

```
//
// Service: LeafPageIterator::IsDone
// Description:
//
Bool LeafPageIterator::IsDone ()
{
  return (mPosition >= mKeys) ? TRUE : FALSE;
}


//
// Service: LeafPageIterator::page
// Description:
//
Pointer* LeafPageIterator::page () const
{
  throw BTreeErrNoServiceExc();
  return NULL;
}


//
// Service: LeafPageIterator::blob
// Description:
//
String* LeafPageIterator::blob () const
{
  return (String*)(mpPage->GetLink(mPosition));
}
```

Appendix D

Definition and Implementation of Class Pointer

D.1 Definition of Class Pointer

```
/*
** Pointer.h
**
** Pointer Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**
*/


#ifndef POINTER_H
#define POINTER_H

/*
** forward declaration
*/
class MemPtr;
class PagePtr;
class Bundle;
class Page;
class PageKey;
class Cursor;
class PageFile;
class IndexPage;

/*
** include file
*/
#include "File.h"

class Pointer{
```

```
public:
  virtual ~Pointer() = 0;
  virtual Cursor* operator [] (PageKey& aPageKey) = 0;
  virtual MemPtr* DeReference() = 0;
  virtual operator MemPtr* () = 0;
  virtual operator PagePtr* () = 0;
  virtual void Free() = 0;

  virtual FilePtr GetOffset() const = 0;
  virtual void SetOffset(const FilePtr aOffset) = 0;

  virtual Page* Read() = 0;
  virtual const FilePtr Write(Page* apPage, FilePtr aOffset) const = 0;

  virtual Page* GetPage() const = 0;
  virtual PageFile& GetPageFile() const = 0;
  virtual String& GetPageFileName() const = 0;
  virtual Bool SetParent (const IndexPage* parent) = 0;

#ifdef _DEBUG
  virtual void print() const = 0;
#endif

protected:
  Pointer();

private:
  Pointer (const Pointer&);
  Pointer& operator = (const Pointer&);
};

#endif
```

D.2 Implementation of Class Pointer

```
/*
** Pointer.cxx
**
** Pointer Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**
*/


/*
** include file
*/
#include "Pointer.h"


//
// Service: Pointer::Pointer
// Description:
//   Constructor
```

```
//
Pointer::~Pointer()
{}


//
// Service: Pointer::~Pointer
// Description:
//   Destructor
//
Pointer::Pointer()
{}
```

Appendix E

Definition and Implementation of Class PagePtr

E.1 Definition of Class PagePtr

```
/*
** PagePtr.h
**
** PagePtr Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** March, 1996
**
*/


#ifndef PAGEPTR_H
#define PAGEPTR_H

/*
** forward declaration
*/
class PageFile;
class Bundle;


/*
** include file
*/
#include "Pointer.h"

class PagePtr : public virtual Pointer{
public:
  PagePtr(PageFile&, FilePtr);
  ~PagePtr();
  PagePtr(const PagePtr& pageptr);
  PagePtr& operator = (const PagePtr& pageptr);
  void Free();

  virtual Cursor* operator [] (PageKey& aPageKey);
  virtual MemPtr* DeReference();
  virtual operator MemPtr* ();
  virtual operator PagePtr* ();

  virtual FilePtr GetOffset() const;
  virtual void SetOffset(const FilePtr aOffset);

  virtual Page* Read();
  virtual const FilePtr Write(Page* apPage, FilePtr aOffset) const;

  virtual Page* GetPage() const;
  virtual PageFile& GetPageFile() const;
  virtual String& GetPageFileName() const;
  virtual Bool SetParent (const IndexPage* parent);

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  PageFile* mpPageFile;
  FilePtr mOffset;
  IndexPage* mpParent;
private:
  PagePtr();
};

#endif
```

E.2 Implementation of Class PagePtr

```
/*
** PagePtr.cxx
**
** PagePtr Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** March, 1996
**
*/

//
// include files
//

#include <stdlib.h>

#include "PagePtr.h"
#include "MemPtr.h"
#include "Pointer.h"
#include "PageFile.h"
#include "Cursor.h"
#include "State.h"
#include "Page.h"
#include "PageKey.h"


//
// Service: PagePtr::PagePtr
// Description:
//
```

```
PagePtr::PagePtr(PageFile& pagefile, FilePtr offset)
  : mpPageFile(&pagefile), mOffset(offset), mpParent(NULL)
{}


//
// Service: PagePtr::~PagePtr
// Description:
//
PagePtr::~PagePtr()
{}


//
// Service: PagePtr::PagePtr(const PagePtr& pageptr)
// Description:
//
PagePtr::PagePtr(const PagePtr& pageptr)
  : mpParent(pageptr.mpParent), mpPageFile(pageptr.mpPageFile),
    mOffset(pageptr.mOffset)
{}


//
// Service: PagePtr& PagePtr::operator =
// Description:
//
PagePtr& PagePtr::operator = (const PagePtr& pageptr)
{
  mpParent = pageptr.mpParent;
  mpPageFile = pageptr.mpPageFile;
  mOffset = pageptr.mOffset;
  return *this;
}


//
// Service: PagePtr::Free
// Description:
//   Free Memory Space
//
void PagePtr::Free()
{
  delete this;
}


//
// Service: PagePtr::DeReference
// Description:
//   DeReference
//
MemPtr* PagePtr::DeReference()
{
  Page* apPage = NULL;
  assert(mOffset != FP_UNKNOWN);

  if((apPage = Read()) != NULL){
    return (new MemPtr(apPage, *this));
  }
  return NULL;
}


//
// Service: PagePtr::operator MemPtr*
// Description:
//
PagePtr::operator MemPtr* ()
{
  return DeReference();
}


//
// Service: PagePtr::operator PagePtr*
// Description:
//
PagePtr::operator PagePtr* ()
{
  return this;
}


//
// Service: PagePtr::GetPageFileName
// Description:
//
String& PagePtr::GetPageFileName() const
{
  return mpPageFile->GetFileName();
}


//
// Service: PagePtr::GetOffset
// Description:
//   Get Page Offset in a PageFile
//
FilePtr PagePtr::GetOffset() const
{
  return mOffset;
}


//
// Service: PagePtr::SetOffset
// Description:
//   Set Page Offset in a PageFile
//
void PagePtr::SetOffset(const FilePtr aOffset)
{
  mOffset = aOffset;
}


//
// Service: PagePtr::Read
// Description:
//
Page* PagePtr::Read()
{
  Page* page = mpPageFile->Read(mOffset);
  if (page) {
    page->SetParent(mpParent);
    page->SetOffset(mOffset);
  }
  return page;
}


//
```

```
// Service: PagePtr::Write
// Description:
//
const FilePtr PagePtr::Write(Page* apPage, FilePtr aOffset) const
{
  assert(apPage);
  return mpPageFile->Write(apPage, aOffset);
}


//
// Service: PagePtr::operator []
// Description:
//
Cursor* PagePtr::operator [] (PageKey&)
{
  throw BTreeErrNoServiceExc();
  // return a dumy cursor for the comilation error
  return (new Cursor(FALSE, *(new Node()), *(new PageKey(0))));
}


//
// Service: PagePtr::GetPage
// Description:
///
Page* PagePtr::GetPage() const
{
  throw BTreeErrNoServiceExc();
  return NULL;
}


#ifdef _DEBUG
//
// Service:
// Description:
//
void PagePtr::print() const
{
  cout << "      < P a g e P t r > " << endl;
  cout << "        mpPageFile : " << GetPageFileName() << endl;
  cout << "        mOffset    : " << GetOffset() << endl;
}
#endif


//
// Service: PagePtr::GetPageFile
// Description:
//
PageFile& PagePtr::GetPageFile() const
{
  return *mpPageFile;
}


//
// Service: PagePtr::SetParent
// Description:
//
Bool PagePtr::SetParent (const IndexPage* parent)
{
  mpParent = (IndexPage*)parent;
  return TRUE;
}
```

Appendix F

Definition and Implementation of Class MemPtr

F.1 Definition of Class MemPtr

```
/*
** MemPtr.h
**
** MemPtr Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** March, 1996
**
*/


#ifndef MEMPTR_H
#define MEMPTR_H

/*
** forward declaration
*/
class PagePtr;
class Bundle;

/*
** include file
*/
#include "Pointer.h"


class MemPtr : public virtual Pointer {
public:
  MemPtr(Page* apPage, PagePtr& aPagePtr);
  MemPtr(PageFile&, FilePtr, Page& page);
  ~MemPtr();
  MemPtr(const MemPtr& aMemPtr);
  MemPtr& operator = (const MemPtr& aMemPtr);
  void Free();

  virtual Cursor* operator [] (PageKey& aPageKey);
  Page* GetPage() const;

  virtual MemPtr* DeReference();
  virtual operator MemPtr* ();
  virtual operator PagePtr* ();

  virtual FilePtr GetOffset() const;
  virtual void SetOffset(const FilePtr aOffset);

  virtual Page* Read();
  virtual const FilePtr Write(Page* apPage, FilePtr aOffset) const;

  virtual PageFile& GetPageFile() const;
  virtual String& GetPageFileName() const;
  virtual Bool SetParent (const IndexPage* parent);

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  PagePtr* mPgPtr;
  Page* mpPage;
private:
  MemPtr();
};
#endif
```

F.2 Implementation of Class MemPtr

```
/*
** MemPtr.cxx
**
** MemPtr Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** March, 1996
**
*/


//
// include files
//

#include <stdlib.h>
#include <iostream.h>

#include "MemPtr.h"
#include "PagePtr.h"
#include "Page.h"
#include "Cursor.h"

//
// Service: MemPtr::MemPtr
// Description:
//
MemPtr::MemPtr(Page* apPage, PagePtr& aPagePtr)
  : mPgPtr(aPagePtr), mpPage(apPage)
{
}


//
// Service: MemPtr::MemPtr
// Description:
//
MemPtr::MemPtr(PageFile& pagefile, FilePtr offset, Page& page)
  : mpPage(&page)
{
  mPgPtr = new PagePtr(pagefile, offset);
}


//
// Service: MemPtr::~MemPtr
// Description:
//
MemPtr::~MemPtr()
{}


//
// Service: MemPtr::MemPtr
// Description:
//
MemPtr::MemPtr(const MemPtr& aMemPtr)
  : mPgPtr(aMemPtr.mPgPtr), mpPage(aMemPtr.mpPage)
{}


//
// Service: MemPtr::MemPtr
// Description:
//
MemPtr& MemPtr::operator = (const MemPtr& aMemPtr)
{
  mPgPtr = aMemPtr.mPgPtr;
  mpPage = aMemPtr.mpPage;
  return *this;
}


//
// Service: MemPtr::Free
// Description:
//   Free Memory Space
//
void MemPtr::Free(){
  mpPage->Free();
  delete this;
}


//
// Service: MemPtr::DeReference
// Description:
//
MemPtr* MemPtr::DeReference()
{
  return this;
}


//
// Service: MemPtr::operator MemPtr*
// Description
//
MemPtr::operator MemPtr* ()
{
  return DeReference();
}


//
// Service: MemPtr::operator PagePtr*
// Description
//
MemPtr::operator PagePtr* ()
{
  return mPgPtr;
}


//
// Service: MemPtr::operator []
// Description:
//
Cursor* MemPtr::operator [] (PageKey& aPageKey)
{
  return (*mpPage)[aPageKey];
}


//
// Service: MemPtr::GetPage
// Description:
//
Page* MemPtr::GetPage() const
{
  return mpPage;
}


#ifdef _DEBUG
//
// Service: MemPtr::print
// Description:
//    for debugging purpose
//
void MemPtr::print() const
{
  cout << "      < M e m P t r > " << endl;
  cout << "        mpPageFile : " << GetPageFileName() << endl;
  cout << "        mOffset    : " << GetOffset() << endl;
}
#endif


//
// Service: MemPtr::GetPageFileName
// Description:
//
String& MemPtr::GetPageFileName() const
{
  return mPgPtr->GetPageFileName();
}


//
// Service: MemPtr::GetOffset
// Description:
```

```
//
FilePtr MemPtr::GetOffset() const
{
  return mPgPtr->GetOffset();
}


//
// Service: MemPtr::SetOffset
// Description:
//
void MemPtr::SetOffset(const FilePtr aOffset)
{
  mPgPtr->SetOffset(aOffset);
  mpPage->SetOffset(aOffset);
}


//
// Service: MemPtr::Read
// Description:
//
Page* MemPtr::Read()
{
  return mPgPtr->Read();
}


//
// Service: MemPtr::Write
// Description:
//
const FilePtr MemPtr::Write(Page* apPage, FilePtr aOffset) const
{
  return mPgPtr->Write(apPage, aOffset);
}


//
// Service: MemPtr::GetPageFile
// Description:
//   Get PageFile
//
PageFile& MemPtr::GetPageFile() const
{
  return mPgPtr->GetPageFile();
}


//
// Service: MemPtr::SetParent
// Description:
//
Bool MemPtr::SetParent (const IndexPage* parent)
{
  mPgPtr->SetParent(parent);
  mpPage->SetParent(parent);
  return TRUE;
}
```

Appendix G

Definition and Implementation of Class File

G.1 Definition of Class File

```
/*
** File.h
**
** File Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-March-1996 Steven Li
**   Initial implemention
**
**   31-July-1997 Steven Li
**   disabled ctor()
**   added FILE* as the attribute of File.
*/


#ifndef FILE_H
#define FILE_H

/*
** define
*/
#define FP_MARKER        -1
#define FP_UNKNOWN       FP_MARKER
#define FP_EOF           -2
#define EXISTING         1
#define NONEXISTING      2
#define BTREE_EXT        "btf"
#define INDEX_EXT        "idx"
#define LEAF_EXT         "lef"
#define MAX_PAGEFILENAME 33   // extra 1 for NULL terminator

typedef long int FilePtr;

/*
** forward declaration
```

```
*/
class BLOBScanner;
class BLOBMaker;

/*
** include files
*/
#include <stdio.h>
#include "BTString.h"
#include "BTreeException.h"

/*
** Macro to define Exception classes
*/
DEFINE_EXCEPTION_FAMILY (FileExc, BTreeException)
DEFINE_EXCEPTION (FileErrNullFileNameExc, FileExc,
                  "Specified filename is NULL")
DEFINE_EXCEPTION (FileErrReadExc, FileExc,
                  "Failed to read data from a File")
DEFINE_EXCEPTION (FileErrWriteExc, FileExc,
                  "Failed to write data to a File")
DEFINE_EXCEPTION (FileErrOpenExistingFileExc, FileExc,
                  "Failed to open an existing file")
DEFINE_EXCEPTION (FileErrOpenNewFileExc, FileExc,
                  "Failed to open a new file")
DEFINE_EXCEPTION (FileErrAlreadyExistingExc, FileExc,
                  "Specified file already existing")
DEFINE_EXCEPTION (FileErrSeekSetExc, FileExc,
                  "Failed to seek the specified offset in the file")
DEFINE_EXCEPTION (FileErrNotExistingExc, FileExc,
                  "Specified file not existing")
DEFINE_EXCEPTION (FileErrLocateExc, FileExc,
                  "Failed to locate a position to write a record")
DEFINE_EXCEPTION (FileErrDeAllocateExc, FileExc,
                  "Failed to discard a record in a file")
DEFINE_EXCEPTION (FileErrReadBlobExc, FileExc,
                  "Failed to read a blob")
DEFINE_EXCEPTION (FileErrWriteBlobExc, FileExc,
                  "Failed to write a blob")
DEFINE_EXCEPTION (FileErrGetPageFileExc, FileExc,
                  "Failed to get the page file")
DEFINE_EXCEPTION (FileErrConstructPageFileExc, FileExc,
                  "Failed to construct the page file")


/*
** Definition of class File
*/

class File {
public:
  String& GetFileName() const;
  virtual unsigned int GetSize() const = 0;
  virtual Bool BufferRead (BLOBScanner& blob) = 0;
  virtual Bool BufferWrite (BLOBMaker& blob) const = 0;

#ifdef _DEBUG
  virtual void print() const;
#endif

protected:
  String mFileName;    // File name
  FILE* mpData;        // File descriptor
  Bool   mNewFile;     // indicator file existing or not
protected:
  File(const String aFileName);
  virtual ~File() = 0;
 private:
  File();                          // NOT DEFINED
  File(const File&);               // NOT DEFINED
  File& operator = (const File&); // NOT DEFINED
};

#endif
```

G.2 Implementation of Class File

```
/*
** File.cxx
**
** File Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions
**   23-March-1996 Steven Li
**   Inital version
**
**   25-July-1997 Steven Li
**   disabled ctor()
**   added FILE* as the attribute of File.
*/

/*
** include files
*/
#include <stdlib.h>
#include <sys/stat.h>

#include "File.h"
#include "Blob.h"

/*
** Implementation of class File
*/

/*
** Service: File::File
** Title:   Constructor
** Description:
** Exceptions:
**   FileErrNullFileNameExc
*/
File::File (const String aFileName)
{
  const char* filename = (const char*)aFileName;
  if(!filename) {
    throw FileErrNullFileNameExc();
  }

  struct stat iFileInfo;         // stat info struct
  int    iRet = -1;

  iRet = stat(filename, &iFileInfo);

  // check if the FileName existing
  if(iRet != 0){
    if((mpData = fopen(filename, "w+b")) == NULL){
      throw FileErrOpenNewFileExc();
    }
    mNewFile = TRUE;
  }
  else {
    if ((mpData = fopen(filename, "r+b")) == NULL) {
      throw FileErrOpenExistingFileExc();
    }
    mNewFile = FALSE;
  }
  mFileName = aFileName;
}


/*
** Service: File::~File
** Description:
**   Destructor
*/
File::~File()
{
  fflush (mpData);
  fclose (mpData);
}


/*
** Service: File::GetFileName
** Description:
*/
String& File::GetFileName() const
{
  return (String&)mFileName;
}


#ifdef _DEBUG
//
// Service: File::print
// Description:   print the content of File
//
void File::print() const
{
  cout << "  < F i l e > " << endl;

  cout << "    mFileName = " << mFileName << endl;
  if (mpData) {
    cout << "    mpData is opened " << endl;
  }
  else {
    cout << "    mpData is not opened yet " << endl;
  }
  if (mNewFile) {
    cout << "    mNewFile is TRUE " << endl;
  }
  else {
    cout << "    mNewFile is FALSE " << endl;
  }
}
#endif
```

Appendix H

Definition and Implementation of Class BTreeFile

H.1 Definition of Class BTreeFile

```
/*
** BTreeFile.h
**
** BTreeFile Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**   Added BufferRead() BufferWrite()
**   Removed BTreeFileHdr
**   Added GetSize(), GetIndexPageFile(), GetLeafPageFile()
**   Added mOrder, mpIndexPageFile, mpLeafPageFile
**   Updated BTreeIterator
*/


#ifndef BTREEFILE_H
#define BTREEFILE_H



/*
** forward declaration
*/
class Cursor;
class Pointer;
class PageKey;
class BTreeIterator;
class LeafPage;
class LeafPageIterator;
class IndexPageFile;
class LeafPageFile;
```

```
/*
** include files
*/
#include "File.h"
#include "Boolean.h"
#include "Iterator.h"
#include "LeafPage.h"
#include "BTreeException.h"


//
// Definition of BTreeFile class
//

class BTreeFile : public File {
  friend class FileFactory;

public:
  BTreeFile(const String aFileName, const int aPageOrder);
  virtual ~BTreeFile();

  void SetRoot(Pointer* root);           // updata Root
  Pointer* GetRoot() const;

  Cursor* operator [] (PageKey&);        // search
  BTreeIterator* CreateIterator() const; // create a iterator

  int GetOrder() const;
  IndexPageFile* GetIndexPageFile () const;
  LeafPageFile*  GetLeafPageFile () const;

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  Pointer* mpRoot;
  int mOrder;
  IndexPageFile* mpIndexPageFile;
  LeafPageFile* mpLeafPageFile;
private:
  virtual unsigned int GetSize() const;
  virtual Bool BufferRead (BLOBScanner& blob);
  virtual Bool BufferWrite (BLOBMaker& blob) const;
  Bool Flush ();
private:
  BTreeFile ();                                   // NOT DEFINED
  BTreeFile (BTreeFile& btreefile);               // NOT DEFINED
  BTreeFile& operator = (BTreeFile& btreefile);  // NOT DEFINED
};

/*
** Definition of BTreeIterator Class
*/

class BTreeIterator : public virtual Iterator{
public:
  BTreeIterator(Pointer* aRoot);
  ~BTreeIterator();
  virtual void First ();
  virtual void Next ();
  virtual Bool IsDone ();
  virtual Pointer* page () const;
  virtual String* blob () const;
private:
  LeafPage* mpLeaf;
  Iterator* mpLeafIterator;
private:
  BTreeIterator();
  BTreeIterator(const BTreeIterator&);
  BTreeIterator& operator = (const BTreeIterator&);
};

#endif
```

H.2 Implementation of Class BTreeFile

```
/*
** BTreeFile.cxx
**
** BTreeFile and BTreeIterator Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**   Added BufferRead() BufferWrite()
**   Removed BTreeFileHdr
**   Added GetSize(), GetIndexPageFile(), GetLeafPageFile()
**   Added mOrder, mpIndexPageFile, mpLeafPageFile
**   Updated BTreeIterator
*/

/*
** include files
**/
#include <stdlib.h>
#include <iostream.h>
#include <assert.h>
#include <sys/stat.h>

#include "BTreeFile.h"
#include "FileFactory.h"
#include "MemPtr.h"
#include "PagePtr.h"
#include "PageFile.h"
#include "State.h"
#include "Cursor.h"
#include "Pointer.h"
#include "Blob.h"

/*
** Implementation of BTreeFile Class
*/

/*
** Service: BTreeFile::BTreeFile
** Title:   Constructor
** Description:
**    Construct BTreeFile from non-existing files
** Exceptions:
**   FileErrOpenNewFileExc
**   FileErrAlreadyExistingExc
**   FileErrWriteExc
```

```
*/
BTreeFile::BTreeFile(const String aBTreeFileName, const int aPageOrder)
  : File(aBTreeFileName), mOrder(aPageOrder) {

  // Make page filenames for indexpage file and leafpage file
  FileFactory* factory = FileFactory::Instance();
  String* index_name = factory->MakePageFileName(aBTreeFileName, INDEX_EXT);
  String* leaf_name  = factory->MakePageFileName(aBTreeFileName, LEAF_EXT);

  // creates PageInfoInfos for indexpagefile and leafpagefile.
  PageFileInfo* index_info = new PageFileInfo(*index_name, *leaf_name, mOrder);
  PageFileInfo* leaf_info  = new PageFileInfo(*index_name, *leaf_name, mOrder);

  mpIndexPageFile = new IndexPageFile(*this, *index_name, *index_info);
  mpLeafPageFile  = new LeafPageFile(*this, *leaf_name, *leaf_info);

  delete index_info;
  delete leaf_info;

  /*
  ** create a new BTreeFile file
  */

  // build the initial root page, and save root page and page order
  if (mNewFile) {
    LeafPage* leafpage = mpLeafPageFile->MakeLeafPage();

    // append leafpage to the end of leaf pagefile
    FilePtr iOffset = mpLeafPageFile->Write(leafpage, FP_UNKNOWN);
    if (iOffset == FP_MARKER) {
      throw FileErrWriteExc();
    }
    // set the offset of the new leaf page (Root)
    leafpage->SetOffset(iOffset);

    // build the Root
    mpRoot = new MemPtr(*mpLeafPageFile, iOffset, *leafpage);

    // write new Root and page order and the pagefie name
    // in which the root page resides.
    if (!Flush()) {
      throw FileErrWriteExc();
    }
  }

  /*
  ** read from an existing BTreeFile file
  */

  else {

    // get the size of blob which stored in the BTreeFile file
    unsigned int size = GetSize();
    char* rawblob = new char [size];

    if (fseek(mpData, 0, SEEK_SET)){
      throw FileErrReadExc();
    }
    if (!(fread(rawblob, size, 1, mpData))){
      throw FileErrReadExc();
    }

    // read/validate page order and build root pointer
    BLOBScanner blob(rawblob);
    if (!BufferRead(blob)) {
      throw FileErrReadBlobExc();
    }
    delete [] rawblob;
  }
}


/*
** Service: BTreeFile::~BTreeFile
** Title:   Destructor
** Description:
*/
BTreeFile::~BTreeFile(){
  if (!Flush()) {
    throw FileErrWriteExc();
  }
  delete mpIndexPageFile;
  delete mpLeafPageFile;

  if(mpRoot) {
    mpRoot->Free();
  }
}


/*
** Service: BTreeFile::BufferRead
** Title:   Read information from BTreeFile blob
** Description:
**    Read from BTreeFile
** Exceptions:
**   FileErrReadExc
**   FileErrOpenExistingFileExc
*/
Bool BTreeFile::BufferRead(BLOBScanner& blob)
{
  // validate page order
  if (mOrder != (int)blob) {
    throw FileErrWrongOrderExc();
    return FALSE;
  }

  // restore root pointer
  char* filename = (char*)blob;
  FilePtr offset = (FilePtr)blob;

  if (mpIndexPageFile->GetFileName() == filename) {
    mpRoot = new PagePtr(*mpIndexPageFile, offset);
  }
  else if (mpLeafPageFile->GetFileName() == filename) {
    mpRoot = new PagePtr(*mpLeafPageFile, offset);
  }
  else {
    delete [] filename;
    throw FileErrUnsupportedFileNameExc();
    return FALSE;
  }

  delete [] filename;
  return TRUE;
}


/*
** Service: BTreeFile::BufferWrite
** Description:
**   Write BTreeFile into BTreeFile file
** Exceptions:
**   FileErrWriteExc
**   FileErrOpenExistingFileExc
*/
Bool BTreeFile::BufferWrite(BLOBMaker& blob) const {
  blob += mOrder;
  blob += (const char*)(mpRoot->GetPageFileName());
  blob += mpRoot->GetOffset();
  return TRUE;
}


/*
** Service: BTreeFile::SetRoot
** Title:   Update mpRoot
** Description:
**
*/
void BTreeFile::SetRoot(Pointer* apNewRoot)
{
  assert(apNewRoot);
  mpRoot = apNewRoot;

  // write new Root and page order and the pagefie name
  // in which the root page resides.
  if (!Flush()) {
    throw FileErrWriteExc();
  }
}


/*
** Service: BTreeFile::operator []
** Title:   Search BTree, given a Pagekey
** Description:
**
*/
Cursor* BTreeFile::operator [] (PageKey& apPageKey)
{
  mpRoot = mpRoot->DeReference();
  return (*mpRoot)[apPageKey]->SetBtree(*this);
}


/*
** Service: BTreeFile::GetRoot
** Title:   Get BTreeFile Root Page
** Description:
**
*/
Pointer* BTreeFile::GetRoot() const{
  return mpRoot;
}


/*
** Service: BTreeFile::CreateIterator
** Description:
**   Create a Iterator on BTree
*/
BTreeIterator* BTreeFile::CreateIterator() const
{
  return (new BTreeIterator(mpRoot));
}
```

```
//
// Service: BTreeFile::GetSize
// Description:
//   Gets the size of persistent blob of BTreeFile
// Note:
//   BTreeFileName, IndexPageFileName and LeafPageFileName have
//   the same length, respectly end with .btf, .idx, .lef
//   The user provides only the btree file name, filefactory
//   validates the the filename and creates .idx and .lef.
//
//   The size of BTreeFile is the sum of
//     size of integer (order), sizeof of FilePtr (offset) and
//     the size of pagefilename
//
unsigned int BTreeFile::GetSize() const
{
  return (sizeof(FilePtr) + GetFileName().GetLen() + 1 +
          sizeof(int));
}


//
// Service: BTreeFile::GetOrder
// Description:
//   Gets the order
//
int BTreeFile::GetOrder() const
{
  return mOrder;
}


/*
** Service: BTreeFile::GetIndexPageFile
** Description:
*/
IndexPageFile* BTreeFile::GetIndexPageFile () const
{
  return mpIndexPageFile;
}


/*
** Service: BTreeFile::GetIndexPageFile
** Description:
*/
LeafPageFile* BTreeFile::GetLeafPageFile () const
{
  return mpLeafPageFile;
}


/*
** Service: BTreeFile::Flush
** Description:
**
*/
Bool BTreeFile::Flush() {
  unsigned int size = GetSize();
  BLOBMaker blob(size);

  if (!BufferWrite(blob)) {
    return FALSE;
  }

  // write BTreeFile blob to BTreeFile file
  if (fseek(mpData, 0, SEEK_SET)){
    return FALSE;
  }
  if (!(fwrite(blob.hasBLOB(), size, 1, mpData))) {
    return FALSE;
  }
  fflush (mpData);
  return TRUE;
}


#ifdef _DEBUG
//
// Service: BTreeFile::print
// Description:
//
void BTreeFile::print() const
{
  File::print();
  cout << "  < B T r e e F i l e > " << endl;

  cout << "    mOrder : " << mOrder << endl;
  cout << "    mRoot  : " << endl;
  mpRoot->print();
  cout << "    mpIndexPageFile : " << endl;
  mpIndexPageFile->print();
  cout << "    mpLeafPageFile  : " << endl;
  mpLeafPageFile->print();
}
#endif


//
// Implementation of BTreeIterator class
//


//
// Service: BTreeIterator::BTreeIterator
// Description:
//
BTreeIterator::BTreeIterator(Pointer* root)
{
  root = root->DeReference();
  Page* aPage = (root)->GetPage();
  assert(aPage);

  // get first LeafPage
  while (aPage->GetFirst(aPage));
  mpLeaf = (LeafPage*)aPage;
  mpLeafIterator = mpLeaf->CreateIterator();
}


//
// Service: BTreeIterator::~BTreeIterator
// Description:
//   Destructor
BTreeIterator::~BTreeIterator()
{
  delete mpLeafIterator;
}


//
// Service:
```

```
// Description:
//
void BTreeIterator::First ()
{
  // get first leafpage
  assert(mpLeaf);
  Pointer* prev = NULL;
  while ((prev = mpLeaf->GetPrevLeaf()) != NULL) {
    prev = prev->DeReference();
    mpLeaf = (LeafPage*)(prev->GetPage());
  }
  mpLeafIterator = mpLeaf->CreateIterator();
}


//
// Service:
// Description:
//
void BTreeIterator::Next ()
{
  mpLeafIterator->Next();
}


//
// Service:
// Description:
//
Bool BTreeIterator::IsDone ()
{
  Pointer* next = NULL;
  if (mpLeafIterator->IsDone()) {

#ifdef _DEBUG
    cout << endl;
#endif

    if ((next = mpLeaf->GetNextLeaf()) != NULL) {
      next = next->DeReference();
      // delete mpLeaf; you never can delete the page, only Pointer can do it !
      delete mpLeafIterator;
      mpLeaf = (LeafPage*)(next->GetPage());
      mpLeafIterator = mpLeaf->CreateIterator();
      return FALSE;
    }
    return TRUE;
  }
  return FALSE;
}


//
// Service:
// Description:
//
Pointer* BTreeIterator::page () const
{
  throw BTreeErrNoServiceExc();
  return NULL;
}


//
// Service:
// Description:
```

```
//
String* BTreeIterator::blob () const
{
  return mpLeafIterator->blob();
}
```

Appendix I

Definition and Implementation of Class PageFile

I.1 Definition of Class PageFile

```
/*
** PageFile.h
**
** PageFile Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-August-1997 Steven Li
**   Added PageFileInfo, RecordInfo
**   Added IndexPageFile, LeafPageFile
*/


#ifndef PAGEFILE_H
#define PAGEFILE_H

/*
** forward declaration
*/
class Page;
class BLOBScanner;
class BLOBMaker;
class PageFileFactory;
class BTreeFile;
class PageFile;
class IndexPage;
class LeafPage;

/*
** include
*/
#include "File.h"
```

```
/*
** definition of class RecordInfo
*/

class RecordInfo {
public:
  static RecordInfo* make (BLOBScanner& blob);
  RecordInfo (FilePtr location, size_t space_size, FilePtr next_deleted);
  virtual ~RecordInfo ();

  Bool SetRecSize (const size_t rec_size);
  size_t GetRecSize () const;

  size_t GetSpaceSize () const;
  FilePtr GetLocation () const;

  Bool SetNextDeleted (const FilePtr next);
  FilePtr GetNextDeleted () const;

  static unsigned int GetSize();
  virtual Bool BufferRead (BLOBScanner& blob);
  virtual Bool BufferWrite (BLOBMaker& blob) const;

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  size_t  RecSize;          //record actual size
  size_t  SpaceSize;        //space size for storing the record
  FilePtr Location;         //address of the record
  FilePtr NextDeleted;      //address of the next deleted record
private:
  RecordInfo ();
private:
  RecordInfo (const RecordInfo&);              // NOT DEFINED
  RecordInfo& operator = (const RecordInfo&); // NOT DEFINED
};



/*
** definition of class PageFileInfo
*/

class PageFileInfo {
public:
  static PageFileInfo* make (BLOBScanner& blob);
  PageFileInfo (const String& ParentName, const String& ChildName,
int order);
  virtual ~PageFileInfo ();
  PageFileInfo (const PageFileInfo&);
  PageFileInfo& operator = (const PageFileInfo&);

  int GetPageOrder () const;
  FilePtr GetFirstDead () const;
  Bool SetFirstDead (const FilePtr offset);
  PageFile* GetParentPageFile ();
  PageFile* GetChildPageFile ();
  Bool operator == (const PageFileInfo&);

  unsigned int GetSize();
  virtual Bool BufferRead(BLOBScanner& blob);
  virtual Bool BufferWrite(BLOBMaker& blob) const;
#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  String mChildPageFileName;
  String mParentPageFileName;
  int mPageOrder;             // Order of Page
  FilePtr mFirstDead;         // the address of the first deleted record
  PageFile* mpParentPageFile;
  PageFile* mpChildPageFile;
private:
  PageFileInfo ();
};



/*
** definition of class PageFile
*/

class PageFile : public File {
  friend class BTreeFile;
public:
  virtual Page* Read(const FilePtr aOffset) = 0;
  virtual const FilePtr Write(Page*, const FilePtr);

  virtual BTreeFile* GetOwner () const;
  virtual PageFile* GetParentPageFile () const;
  virtual PageFile* GetChildPageFile () const;

  virtual void Discard(const FilePtr aPosition);
  virtual int GetPageOrder () const;

  virtual unsigned int GetSize () const;
  virtual Bool BufferRead (BLOBScanner& blob);
  virtual Bool BufferWrite (BLOBMaker& blob) const;
  virtual Bool Flush ();

#ifdef _DEBUG
  virtual void print () const;
  virtual void ShowDiscardedSpace ();
#endif

protected:
  BTreeFile*    mpOwner;
  PageFileInfo* mpPageFileInfo;
protected:
  PageFile (BTreeFile&, String&, PageFileInfo&);
  virtual ~PageFile () = 0;
  virtual FilePtr locate (const FilePtr aPosition,
  const size_t aPageSize);
  virtual RecordInfo* ReadRecordInfo (const FilePtr aOffset);
  virtual Bool WriteRecordInfo (const RecordInfo& info);
private:
  PageFile();                                  // NOT DEFINED
  PageFile(PageFile& aPageFile);               // NOT DEFINED
  PageFile& operator = (PageFile& aPageFile); // NOT DEFINED
};



/*
** definition of class IndexPageFile
*/
```

```
class IndexPageFile : public PageFile {
  friend class BTreeFile;
public:
  virtual Page* Read(const FilePtr aOffset);
  virtual PageFile* GetParentPageFile () const;
  virtual PageFile* GetChildPageFile () const;

  virtual unsigned int GetSize () const;
  virtual Bool BufferRead (BLOBScanner& blob);
  virtual Bool BufferWrite (BLOBMaker& blob) const;
  IndexPage* MakeIndexPage ();

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  IndexPageFile(BTreeFile&, String&, PageFileInfo&);
  virtual ~IndexPageFile();
private:
  IndexPageFile();                                     // NOT DEFINED
  IndexPageFile(IndexPageFile& aPageFile);             // NOT DEFINED
  IndexPageFile& operator = (IndexPageFile& aPageFile); // NOT DEFINED
};



/*
** definition of class LeafPageFile
*/

class LeafPageFile : public PageFile {
  friend class BTreeFile;
public:
  virtual Page* Read(const FilePtr aOffset);
  virtual PageFile* GetParentPageFile () const;
  virtual PageFile* GetChildPageFile () const;

  virtual unsigned int GetSize () const;
  virtual Bool BufferRead (BLOBScanner& blob);
  virtual Bool BufferWrite (BLOBMaker& blob) const;
  LeafPage* MakeLeafPage();

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  LeafPageFile(BTreeFile&, String&, PageFileInfo&);
  virtual ~LeafPageFile();
private:
  LeafPageFile();                                      // NOT DEFINED
  LeafPageFile(LeafPageFile& aPageFile);               // NOT DEFINED
  LeafPageFile& operator = (LeafPageFile& aPageFile);  // NOT DEFINED
};

#endif
```

I.2 Implementation of Class PageFile

```
/*
** PageFile.cxx
**
** PageFile Class Implementation File
**
```

```
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-July-1997 Steven Li
**   Added PageFileInfo, RecordInfo
**   Added IndexPageFile, LeafPageFile
*/

/*
** include files
*/
#include <stdlib.h>
#include <iostream.h>
#include <assert.h>
#include <string.h>
#include <sys/stat.h>

#include "PageFile.h"
#include "Page.h"
#include "IndexPage.h"
#include "LeafPage.h"
#include "PagePtr.h"
#include "BTString.h"
#include "Blob.h"
#include "FileFactory.h"
#include "BTreeFile.h"


/*
** Service: RecordInfo::RecordInfo
** Description:
**   default constructor
*/
RecordInfo::RecordInfo ()
  : Location(FP_UNKNOWN), SpaceSize(0),
    RecSize(0), NextDeleted(FP_UNKNOWN)
{}


/*
** Service: RecordInfo::RecordInfo
** Description:
**   constructor
*/
RecordInfo::RecordInfo (FilePtr location, size_t space_size,
FilePtr next_deleted)
  : Location(location), SpaceSize(space_size),
    RecSize(space_size), NextDeleted(next_deleted) {
}


/*
** Service: RecordInfo::~RecordInfo
** Description:
**   destructor
*/
RecordInfo::~RecordInfo ()
{}
```

```
/*
** Service: RecordInfo::~RecordInfo
** Description:
**   destructor
*/
RecordInfo* RecordInfo::make (BLOBScanner& blob) {
  RecordInfo* info = new RecordInfo ();
  if (info->BufferRead (blob)) {
    return info;
  }
  return NULL;
}


/*
** Service: RecordInfo::SetRecSize
** Description:
**
*/
Bool RecordInfo::SetRecSize (const size_t rec_size) {
  RecSize = rec_size;
  return TRUE;
}


/*
** Service: RecordInfo::GetRecSize
** Description:
**
*/
size_t RecordInfo::GetRecSize () const {
  return RecSize;
}


/*
** Service: RecordInfo::GetSpaceSize
** Description:
**
*/
size_t RecordInfo::GetSpaceSize () const {
  return SpaceSize;
}


/*
** Service: RecordInfo::GetLocation
** Description:
**
*/
FilePtr RecordInfo::GetLocation () const {
  return Location;
}


/*
** Service: RecordInfo::SetNextDeleted
** Description:
**
*/
Bool RecordInfo::SetNextDeleted (const FilePtr next) {
  NextDeleted = next;
  return TRUE;
}
```

```
/*
** Service: RecordInfo::GetNextDeleted
** Description:
**
*/
FilePtr RecordInfo::GetNextDeleted () const {
  return NextDeleted;
}


/*
** Service: RecordInfo::SetSize
** Description:
**
*/
unsigned int RecordInfo::GetSize() {
  return (sizeof(size_t)*2 + sizeof(FilePtr)*2);
}


/*
** Service: RecordInfo::BufferRead
** Description:
**
*/
Bool RecordInfo::BufferRead (BLOBScanner& blob) {
  Location    = (FilePtr) blob;
  NextDeleted = (FilePtr) blob;
  RecSize     = (size_t)  blob;
  SpaceSize   = (size_t)  blob;
  return TRUE;
}


/*
** Service: RecordInfo::BufferWrite
** Description:
**
*/
Bool RecordInfo::BufferWrite (BLOBMaker& blob) const
{
  blob += (FilePtr) Location;
  blob += (FilePtr) NextDeleted;
  blob += (size_t) RecSize;
  blob += (size_t) SpaceSize;
  return TRUE;
}


#ifdef _DEBUG
//
// Service: RecordInfo::print
// Description:
//
void RecordInfo::print() const
{
  cout << "  < R e c o r d I n f o > "        << endl;
  cout << "    SpaceSize   : " << SpaceSize   << endl;
  cout << "    RecSize     : " << RecSize     << endl;
  cout << "    Location    : " << Location    << endl;
  cout << "    NextDeleted : " << NextDeleted << endl;
  cout << endl;
}
#endif
```

```
//
// implementation of class PageFileInfo
//

/*
** Service: PageFileInfo::
** Description:
**
*/
PageFileInfo::PageFileInfo ()
  : mPageOrder(0), mFirstDead(FP_UNKNOWN),
    mpParentPageFile(NULL), mpChildPageFile(NULL)
{}


/*
** Service: PageFileInfo::
** Description:
**
*/
PageFileInfo::PageFileInfo (const String& ParentName,
    const String& ChildName, int order)
  : mParentPageFileName(ParentName), mChildPageFileName(ChildName),
    mPageOrder(order), mFirstDead(FP_UNKNOWN),
    mpParentPageFile(NULL), mpChildPageFile(NULL)
{}


/*
** Service: PageFileInfo::~PageFileInfo
** Description:
**
*/
PageFileInfo::~PageFileInfo ()
{}


/*
** Service: PageFileInfo::PageFileInfo
** Description:
**
*/
PageFileInfo::PageFileInfo (const PageFileInfo& info)
  : mParentPageFileName(info.mParentPageFileName),
    mChildPageFileName(info.mChildPageFileName),
    mPageOrder(info.mPageOrder),
    mFirstDead(info.mFirstDead),
    mpParentPageFile(info.mpParentPageFile),
    mpChildPageFile(info.mpChildPageFile)
{}


/*
** Service: PageFileInfo::operator =
** Description:
**
*/
PageFileInfo& PageFileInfo::operator = (const PageFileInfo& info)
{
  mParentPageFileName = info.mParentPageFileName;
  mChildPageFileName = info.mChildPageFileName;
  mPageOrder = info.mPageOrder;
  mFirstDead = info.mFirstDead;
  mpParentPageFile = info.mpParentPageFile;
  mpChildPageFile = info.mpChildPageFile;
  return *this;
}


/*
** Service: PageFileInfo::make
** Description:
**
*/
PageFileInfo* PageFileInfo::make (BLOBScanner& blob)
{
  PageFileInfo* info = new PageFileInfo ();
  if (info->BufferRead (blob)) {
    return info;
  }
  return NULL;
}


/*
** Service: PageFileInfo::GetPageOrder
** Description:
**
*/
int PageFileInfo::GetPageOrder () const
{
  return mPageOrder;
}


/*
** Service: PageFileInfo::GetFirstDead
** Description:
**
*/
FilePtr PageFileInfo::GetFirstDead () const
{
  return mFirstDead;
}


/*
** Service: PageFileInfo::SetFirstDead
** Description:
**
*/
Bool PageFileInfo::SetFirstDead (const FilePtr offset)
{
  mFirstDead = offset;
  return TRUE;
}


/*
** Service: PageFileInfo::GetParentPageFile
** Description:
**
*/
PageFile* PageFileInfo::GetParentPageFile ()
{
  if (!mpParentPageFile) {
    FileFactory* factory = FileFactory::Instance();
    mpParentPageFile = factory->GetPageFile(mParentPageFileName);
    if (!mpParentPageFile) {
      throw FileErrGetPageFileExc();
    }
  }
  return mpParentPageFile;
}


/*
** Service: PageFileInfo::GetChildPageFile
** Description:
**
*/
PageFile* PageFileInfo::GetChildPageFile ()
{
  if (!mpChildPageFile) {
    FileFactory* factory = FileFactory::Instance();
    mpChildPageFile = factory->GetPageFile(mChildPageFileName);
    if (!mpChildPageFile) {
      throw FileErrGetPageFileExc();
    }
  }
  return mpChildPageFile;
}


/*
** Service: PageFileInfo::operator ==
** Description:
**
*/
Bool PageFileInfo::operator == (const PageFileInfo& info)
{
  if ((mPageOrder == info.mPageOrder) &&
      //(mFirstDead == info.mFirstDead) &&
      (mParentPageFileName = info.mParentPageFileName) &&
      (mChildPageFileName  = info.mChildPageFileName)) {
    return TRUE;
  }
  return FALSE;
}


/*
** Service: PageFileInfo::GetSize
** Description:
**
*/
unsigned int PageFileInfo::GetSize()
{
  return (mParentPageFileName.GetLen() + 1 +
          mChildPageFileName.GetLen()  + 1 +
          sizeof(int) + sizeof(FilePtr));
}


/*
** Service: PageFileInfo::BufferRead
** Description:
**
*/
Bool PageFileInfo::BufferRead(BLOBScanner& blob)
{
  mPageOrder = (int)blob;
  mFirstDead  = (FilePtr)blob;
  char* filename = (char*)blob;
  if (!filename) {
    return FALSE;
  }
  mParentPageFileName = filename;
  delete [] filename;

  filename = (char*)blob;
  if (!filename) {
    return FALSE;
  }
  mChildPageFileName  = filename;
  delete [] filename;
  return TRUE;
}


/*
** Service: PageFileInfo::BufferWrite
** Description:
**
*/
Bool PageFileInfo::BufferWrite(BLOBMaker& blob) const
{
  blob += (int)mPageOrder;
  blob += (FilePtr)mFirstDead;
  blob += (char*)((const char*)(mParentPageFileName));
  blob += (char*)((const char*)(mChildPageFileName));
  return TRUE;
}


#ifdef _DEBUG
//
// Service: PageFileInfo::print
// Description:
//
void PageFileInfo::print() const
{
  cout << "  < P a g e F i l e I n f o > " << endl;
  cout << "    mPageOrder : " << mPageOrder << endl;
  cout << "    mFirstDead : " << mFirstDead << endl;
  cout << "    mParentPageFileName : " << mParentPageFileName
       << endl;
  cout << "    mChildPageFileName  : " << mChildPageFileName
       << endl;
}
#endif



/*
** implementation of class PageFile
*/

/*
** Service: PageFile::PageFile
** Title:   Constructor
** Description:
**    construct PageFile from nonexisting file
** Exceptions:
**   FileErrOpenNewFileExc
**   FileErrAlreadyExistingExc
**   FileErrSeekSetExc
*/
PageFile::PageFile(BTreeFile& btreefile, String& filename,
                   PageFileInfo& info)
  : File(filename), mpOwner(&btreefile)
{
  mpPageFileInfo = new PageFileInfo(info);

  if(mNewFile) {
    if (!Flush()) {
      throw FileErrWriteExc();
    }
  }
  // check exisiting data in the supplied info
  // with the read pagefile info
  else {
    // get the size of blob of PageFile persistent info
    unsigned int size = GetSize();
    char* rawblob = new char [size];

    if (fseek(mpData, 0, SEEK_SET)) {
      throw FileErrReadExc();
    }
    if (!(fread(rawblob, size, 1, mpData))) {
      throw FileErrReadExc();
    }

    BLOBScanner blob(rawblob);
    PageFileInfo* original = mpPageFileInfo;
    if (!BufferRead(blob)) {
      throw FileErrReadBlobExc();
    }
    delete [] rawblob;

    //validate pagefile persistent data
    if (!(*mpPageFileInfo == *original)) {
      throw FileErrConstructPageFileExc();
    }
  }
}


/*
** Service: PageFile::~PageFile
** Title:   Destructor
** Description:
*/
PageFile::~PageFile() {
  if (!Flush()) {
    throw FileErrWriteExc();
  }
  delete mpPageFileInfo;
}


/*
** Service: PageFile::Flush
** Description:
*/
Bool PageFile::Flush () {
  unsigned int size = GetSize();
  BLOBMaker blob(size);

  if (!BufferWrite(blob)) {
    return FALSE;
  }

  // write PageFile blob to PageFile file
  if (fseek(mpData, 0, SEEK_SET)){
    return FALSE;
  }
  if (!(fwrite(blob.hasBLOB(), size, 1, mpData))) {
    return FALSE;
  }
  fflush (mpData);
  return TRUE;
}


/*
** Service: PageFile::ReadRecordInfo
** Description:
**   Read a Page from PageFile
*/
RecordInfo* PageFile::ReadRecordInfo (const FilePtr aOffset)
{
  if(fseek(mpData, aOffset, SEEK_SET)) {
    return NULL;
  }

  unsigned int blob_size = RecordInfo::GetSize();
  char* rawblob = new char [blob_size];
  ::memset(rawblob, 0, blob_size);

  if(!(fread((void*)rawblob, blob_size, 1, mpData))) {
    if (ferror(mpData)) {
      return NULL;
    }
  }

  BLOBScanner infoblob((void*)rawblob);
  RecordInfo* info = RecordInfo::make(infoblob);
  delete [] rawblob;

  return info;
}


/*
** Service: PageFile::WriteRecordInfo
** Description:
**   Write a RecordInfo into PageFile
*/
Bool PageFile::WriteRecordInfo (const RecordInfo& info)
{
  unsigned int size = RecordInfo::GetSize();
  BLOBMaker blob(size);

  if (!info.BufferWrite(blob)) {
    return FALSE;
  }

  // write PageFile blob to PageFile file
  if (fseek(mpData, info.GetLocation(), SEEK_SET)){
    return FALSE;
  }
  if (!(fwrite(blob.hasBLOB(), size, 1, mpData))) {
    return FALSE;
  }
  fflush (mpData);
  return TRUE;
}


/*
** Service: PageFile::Write
** Title:   Write a Page into PageFile
** Description:
```

```
**    write Page to PageFile at given offset, return the offset
**    on success, otherwise return FP_MARKER on error.
** Exceptions:
**   FileErrOpenExisitingFileExc
**   FileErrLocateExc
*/
const FilePtr PageFile::Write(Page* apPage, const FilePtr aOffset)
{
  assert(apPage);

  // get size of page
  unsigned int pagesize = apPage->GetSize();

  // find a location to write
  FilePtr offset  = FP_MARKER;
  if((offset = locate(aOffset, pagesize)) == FP_MARKER){
    return FP_MARKER;
  }

  // setup Record Header Info
  RecordInfo* info = NULL;
  if (offset == FP_EOF) {
    if (fseek(mpData, 0, SEEK_END)) {
      return FP_MARKER;
    }
    offset = ftell(mpData);
    info = new RecordInfo (offset, pagesize, FP_MARKER);
  }
  else {
    info = ReadRecordInfo (offset);
    if (!info) {
      return FP_MARKER;
    }
    info->SetRecSize(pagesize);
  }
  // write Record Header into PageFile
  if (!WriteRecordInfo(*info)) {
    return FP_MARKER;
  }

  // write page into buffer
  BLOBMaker blob (pagesize);
  if (!apPage->BufferWrite(blob)) {
    return FP_MARKER;
  }

  // write buffer into PageFile
  if (!(fwrite(blob.hasBLOB(), pagesize, 1, mpData))) {
    return FP_MARKER;
  }

  fflush(mpData);
  return offset;       // return offset on success;
}


/*
** Service: PageFile::locate
** Title:   Locate a writing position in PageFile
** Description:
**    locate the page storage space, FP_MARKER on errors
**    return a position at which the page can be stored on success.
**    if the page is appent to end of PageFile, returns FP_EOF
*/
FilePtr PageFile::locate(const FilePtr aPosition, const size_t aPageSize)
{
  RecordInfo* curr_info = NULL;
  RecordInfo* prev_info = NULL;

  //for an existing page, check the existing space size
  if(aPosition != FP_UNKNOWN) {
    curr_info = ReadRecordInfo(aPosition);
    if (!curr_info) {
      return FP_MARKER;
    }
    if(curr_info->GetSpaceSize() >= aPageSize) {
      delete curr_info;
      return aPosition;
    }
    delete curr_info;
    curr_info = NULL;
  }

  //for aPosition != FP_UNKNOWN, de-Allocate the disk space
  //upon a sufficient space is found

  //find discarded space for writing the page
  FilePtr iPtr  = mpPageFileInfo->GetFirstDead();
  while (iPtr != FP_MARKER) {
    curr_info = ReadRecordInfo(iPtr);
    if (!curr_info) {
      return FP_MARKER;
    }

    if (curr_info->GetSpaceSize() >= aPageSize) {
      if (!prev_info) {
        //update FirstDead in fileheader
mpPageFileInfo->SetFirstDead (curr_info->GetNextDeleted());
delete curr_info;
        curr_info = NULL;
        if (!Flush()) {
          return FP_MARKER;
        }
      }
      else {
        prev_info->SetNextDeleted (curr_info->GetNextDeleted());
        if (!WriteRecordInfo(*prev_info)) {
          return FP_MARKER;
        }
      }

      //get the position to write the page
      if (aPosition != FP_UNKNOWN) {
Discard(aPosition);
      }

      if (prev_info) {
        delete prev_info;
        prev_info = NULL;
      }
      if (curr_info) {
        delete curr_info;
        curr_info = NULL;
      }
      return iPtr;
    }
    else {
      iPtr = curr_info->GetNextDeleted();
      if (prev_info) {
        delete prev_info;
        prev_info = NULL;
      }
      prev_info = curr_info;
    }
  }

  //no discarded page space can fit the page, append to EOF
  return FP_EOF;
}


/*
** Service: PageFile::Discard
** Title:   DeAllocate disk space in a PageFile
** Description:
** Exceptions:
**    FileErrDeAllocateExc
*/
void PageFile::Discard(const FilePtr aPosition)
{
  RecordInfo* info = ReadRecordInfo(aPosition);
  if (!info) {
    throw FileErrDeAllocateExc();
  }

  info->SetNextDeleted (mpPageFileInfo->GetFirstDead());
  if (!WriteRecordInfo(*info)) {
    throw FileErrDeAllocateExc();
  }

  mpPageFileInfo->SetFirstDead(info->GetLocation());
  delete info;
  if (!Flush()) {
    throw FileErrDeAllocateExc();
  }
}


/*
** Service: PageFile::GetOwner
** Description:
*/
BTreeFile* PageFile::GetOwner () const
{
  return mpOwner;
}


/*
** Service: PageFile::GetParentPageFile
** Description:
*/
PageFile* PageFile::GetParentPageFile () const
{
  return mpPageFileInfo->GetParentPageFile();
}


/*
** Service: PageFile::GetChildPageFile
** Description:
*/
PageFile* PageFile::GetChildPageFile () const
{
  return mpPageFileInfo->GetChildPageFile();
}
```

```
/*
** Service: PageFile::GetPageOrder
** Description:
*/
int PageFile::GetPageOrder () const
{
  return mpPageFileInfo->GetPageOrder();
}


/*
** Service: PageFile::GetSize
** Description:
*/
unsigned int PageFile::GetSize () const
{
  return (mpPageFileInfo->GetSize());
}


/*
** Service: PageFile::BufferRead
** Description:
*/
Bool PageFile::BufferRead (BLOBScanner& blob)
{
  mpPageFileInfo = PageFileInfo::make(blob);
  if (!mpPageFileInfo) {
    return FALSE;
  }
  return TRUE;
}


/*
** Service: PageFile::BufferWrite
** Description:
*/
Bool PageFile::BufferWrite (BLOBMaker& blob) const
{
  if (!mpPageFileInfo->BufferWrite(blob)) {
    return FALSE;
  }
  return TRUE;
}


#ifdef _DEBUG
//
// Service: PageFile::print
// Description:
//
void PageFile::print() const
{
  File::print();

  cout << "  < P a g e F i l e > " << endl;
  cout << "    mpOwner : " << mpOwner->GetFileName() << endl;
  cout << "    mpPageFileInfo : " << endl;
  mpPageFileInfo->print();
}

//
// Service: PageFile::ShowDiscardedSpace
// Description :
//
```

```
void PageFile::ShowDiscardedSpace ()
{
  RecordInfo* info = NULL;
  FilePtr     offset = FP_UNKNOWN;
  unsigned int count = 0;

  cout << "  < D i s c a r d e d   s p a c e > " << endl;

  offset = mpPageFileInfo->GetFirstDead();
  while (offset != FP_UNKNOWN) {
    info = ReadRecordInfo (offset);
    info->print ();
    offset = info->GetNextDeleted ();
    delete info;
  }
  cout << endl;
}

#endif



/*
** implementation of class IndexPageFile
*/

/*
** Service: IndexPageFile::IndexPageFile
** Description:
*/
IndexPageFile::IndexPageFile(BTreeFile& btree, String& filename,
                             PageFileInfo& info)
  : PageFile (btree, filename, info)
{}


/*
** Service: IndexPageFile::~IndexPageFile
** Description:
*/
IndexPageFile::~IndexPageFile()
{}


/*
** Service: IndexPageFile::Read
** Description:
*/
Page* IndexPageFile::Read(const FilePtr aOffset)
{
  RecordInfo* info = ReadRecordInfo(aOffset);
  if (!info) {
    return NULL;
  }

  unsigned int blob_size = info->GetRecSize();
  char* rawblob = new char [blob_size];
  delete info;

  if(!(fread(rawblob, blob_size, 1, mpData))) {
    return NULL;
  }

  IndexPage* page = new IndexPage (*this);
  BLOBScanner pageblob((void*)rawblob);
  Bool result = page->BufferRead (pageblob);
  delete [] rawblob;

  if (result) {
    //set transient page offset
    page->SetOffset(aOffset);
    return page;
  }
  return NULL;
}


/*
** Service: IndexPageFile::MakeIndexPage
** Description:
*/
IndexPage* IndexPageFile::MakeIndexPage ()
{
  return new IndexPage (*this);
}

/*
** Service: IndexPageFile::GetSize
** Description:
*/
unsigned int IndexPageFile::GetSize () const
{
  return PageFile::GetSize();
}


/*
** Service: IndexPageFile::BufferRead
** Description:
*/
Bool IndexPageFile::BufferRead (BLOBScanner& blob)
{
  return PageFile::BufferRead(blob);
}


/*
** Service: IndexPageFile::BufferWrite
** Description:
*/
Bool IndexPageFile::BufferWrite (BLOBMaker& blob) const
{
  return PageFile::BufferWrite(blob);
}


/*
** Service: IndexPageFile::GetParentPageFile
** Description:
*/
PageFile* IndexPageFile::GetParentPageFile () const
{
  return (IndexPageFile*)this;
}


/*
** Service: IndexPageFile::GetChildPageFile
** Description:
*/
PageFile* IndexPageFile::GetChildPageFile () const
```

```
{
  return PageFile::GetChildPageFile();
}


#ifdef _DEBUG
//
// Service: IndexPageFile::print
// Description:
//
void IndexPageFile::print() const
{
  PageFile::print();
  cout << "  < I n d e x P a g e F i l e > " << endl;
}
#endif



/*
** implementation of class LeafPageFile
*/

/*
** Service: LeafPageFile::LeafPageFile
** Description:
*/
LeafPageFile::LeafPageFile(BTreeFile& btree, String& filename,
                           PageFileInfo& info)
  : PageFile (btree, filename, info)
{}


/*
** Service: LeafPageFile::~LeafPageFile
** Description:
*/
LeafPageFile::~LeafPageFile()
{}


/*
** Service: LeafPageFile::Read
** Description:
*/
Page* LeafPageFile::Read(const FilePtr aOffset)
{
  RecordInfo* info = ReadRecordInfo(aOffset);
  if (!info) {
    return NULL;
  }

  unsigned int blob_size = info->GetRecSize();
  char* rawblob = new char [blob_size];
  delete info;

  if(!(fread(rawblob, blob_size, 1, mpData))) {
    return NULL;
  }

  LeafPage* page = new LeafPage (*this);
  BLOBScanner pageblob((void*)rawblob);

  Bool result = page->BufferRead (pageblob);
  delete [] rawblob;
  if (result) {
    return page;
  }
  return NULL;
}


/*
** Service: LeafPageFile::MakeLeafPage
** Description:
*/
LeafPage* LeafPageFile::MakeLeafPage ()
{
  return new LeafPage (*this);
}


/*
** Service: LeafPageFile::GetSize
** Description:
*/
unsigned int LeafPageFile::GetSize () const
{
  return PageFile::GetSize();
}


/*
** Service: LeafPageFile::BufferRead
** Description:
*/
Bool LeafPageFile::BufferRead (BLOBScanner& blob)
{
  return PageFile::BufferRead(blob);
}


/*
** Service: LeafPageFile::BufferWrite
** Description:
*/
Bool LeafPageFile::BufferWrite (BLOBMaker& blob) const
{
  return PageFile::BufferWrite(blob);
}


/*
** Service: LeafPageFile::GetParentPageFile
** Description:
*/
PageFile* LeafPageFile::GetParentPageFile () const
{
  return PageFile::GetParentPageFile();
}


/*
** Service: LeafPageFile::GetChildPageFile
** Description:
*/
PageFile* LeafPageFile::GetChildPageFile () const
{
  return (LeafPageFile*)this;
}
```

```
#ifdef _DEBUG
//
// Service: LeafPageFile::print
// Description:
//
void LeafPageFile::print() const
{
  PageFile::print();
  cout << "  < L e a f P a g e F i l e > " << endl;
}
#endif
```

Appendix J

Definition and Implementation of Class Cursor

J.1 Definition of Class Cursor

```
/*
** Cursor.h
**
** Cursor Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-March-1996 Steven Li
**   Initial implemention
**
**   31-July-1997 Steven Li
**   disabled ctor, cctor, operator =
**
*/


#ifndef CURSOR_H
#define CURSOR_H


//
// forward declaration
//
class BTreeFile;
class Page;
class PageKey;
class Pointer;

//
// include files
//
#include "Boolean.h"

//
// Definition of class Node
//
```

```
class Node {
  friend class Cursor;
public:
  Node();
  Node(Page& apPage, int aPosition);
  virtual ~Node();

  void SetUpperNode(Node& aUpperNode);
  Node* GetUpperNode() const;

  int GetPosition() const;

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  Page* mpPage;
  int mPosition;
  Node* mpUpperNode;
private:
  Node(const Node&);
  Node& operator = (const Node&);
};



//
// Definition of class Cursor
//
class Cursor {
public:
  Cursor(const Bool aFound, Node& aNode, PageKey& aKey);
  virtual ~Cursor();

  Bool operator = (char* rec);
  Cursor* AddThisPage(Node& aNode);
  Cursor* SetBtree(BTreeFile& apBTreeFile);

#ifdef _DEBUG
  void print();
#endif

private:
  Bool mbKeyFound;
  PageKey& mKey;
  Node* mpCurrentPage;
  Node* mpNodes;
  BTreeFile* mpBTreeFile;
private:
  Bool updateRec(char* rec);
  Bool insertRec(char* rec);
  Bool deleteRec();
private:
  Cursor();
  Cursor(const Cursor&);
  Cursor& operator = (const Cursor&);
};

#endif
```

J.2 Implementation of Class Cursor

```
/*
** Cursor.cxx
```

```
**
** Cursor Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-March-1996 Steven Li
**   Initial implemention
**
**   31-July-1997 Steven Li
**   disabled ctor, cctor, operator =
**   memory leaking
*/

/*
** include files
*/
#include <stdlib.h>
#include <iostream.h>
#include "Cursor.h"
#include "MemPtr.h"
#include "Page.h"
#include "PageKey.h"
#include "State.h"
#include "BTreeFile.h"
#include "PagePtr.h"
#include "PageFile.h"


//
// Implementation of class Node
//

//
// Service: Node::Node
// Description:
//
Node::Node()
  : mpPage(NULL), mPosition(0), mpUpperNode(NULL)
{}


//
// Service: Node::Node
// Description:
//
Node::Node(Page& aPage, int aPosition)
  : mpPage(&aPage), mPosition(aPosition), mpUpperNode(NULL)
{}


//
// Service: Node::~Node
// Description:
//
Node::~Node()
{}


//
// Service: Node::GetPosition
// Description:
```

```
//
int Node::GetPosition() const
{
  return mPosition;
}


//
// Service: Node::SetUpperNode
// Description:
//
void Node::SetUpperNode(Node& aUpper)
{
  mpUpperNode = &aUpper;
}


//
// Service: Node::GetUpperNode
// Description:
//
Node* Node::GetUpperNode() const
{
  return mpUpperNode;
}


#ifdef _DEBUG
//
// Service: Node::print
// Description: for debugging
//
void Node::print() const
{
  cout << "    < N o d e > " << endl;
  cout << "      mPosition : " << mPosition << endl;
  cout << "      mpPage    : " << endl;
  mpPage->print();
}
#endif



//
// Implementation of class Cursor
//

//
// Service: Cursor::Cursor
// Description:
//
Cursor::Cursor(const Bool found, Node& node, PageKey& pkey)
   : mKey (pkey), mbKeyFound(found),
     mpCurrentPage(&node), mpNodes(&node)
{}


//
// Service: Cursor::~Cursor
// Description:
//
Cursor::~Cursor()
{
  delete mpNodes;
}
```

```
//
// Service: Cursor::AddThisPage
// Description:
//   Add this page into a node list
//
Cursor* Cursor::AddThisPage(Node& node)
{
  assert (mpNodes);
  mpCurrentPage->SetUpperNode(node);
  mpCurrentPage = &node;
  return this;
}


//
// Service: Cursor::SetBtree
// Title:   Set BTree pointer
// Description:
//
Cursor* Cursor::SetBtree(BTreeFile& bf)
{
  mpBTreeFile = &bf;
  return this;
}


//
// Service: Cursor::operator =
// Description:
//
Bool Cursor::operator = (char* aRecord)
{
  mpCurrentPage = mpNodes;
  if (aRecord) {
    if (mbKeyFound) {
      return updateRec(aRecord);
    }
    else {
      return insertRec(aRecord);
    }
  }
  else if (mbKeyFound) {
    return deleteRec();
  }
  else {
    return FALSE;
  }
}


//
// Service: Cursor::updateRec
// Description:
//   Update a Record in a BTree, especially, updata a DataPage
//
Bool Cursor::updateRec(char* rec)
{
  Page* aPage = mpCurrentPage->mpPage;
  aPage->SetLink(mpCurrentPage->mPosition, new String(rec));

  FilePtr offset, pageoffset;
  pageoffset = aPage->GetOffset();
  PageFile* pagefile = aPage->GetPageFile();

  if ((offset = pagefile->Write(aPage, pageoffset)) == FP_UNKNOWN) {
      return FALSE;
    }

    if (offset != pageoffset) {
      aPage->SetOffset(offset);

      if (aPage->IsRoot()) {
        mpBTreeFile->SetRoot(new MemPtr(*pagefile, pageoffset, *aPage));
        return TRUE;
      }

      Page* aParentPage = mpCurrentPage->mpUpperNode->mpPage;
      int aParentLinkPos = mpCurrentPage->mpUpperNode->mPosition;

      ((Pointer*)(aParentPage->GetLink(aParentLinkPos)))->SetOffset(pageoffset);

      pageoffset = aParentPage->GetOffset();
      pagefile = pagefile->GetParentPageFile();
      if (pagefile->Write(aParentPage, pageoffset) != pageoffset) {
        return FALSE;
      }
    }
    return TRUE;
}


//
// Service: Cursor::insertRec
// Description:
//   Insert a Record into BTree
//
Bool Cursor::insertRec(char* rec)
{
  String aRecord(rec);
  State::Status aStatus;

  State* aState = mpCurrentPage->mpPage->Insert(mKey, &aRecord);
  while ((aStatus = aState->GetStatus()) != State::COMPLETE) {
    if(aStatus == State::ERROR) {
      delete aState;
      return FALSE;
    }
    if (aStatus == State::NEWROOT) {
      mpCurrentPage->SetUpperNode(*(new Node(*(aState->GetMemPtr()->GetPage()),
                                              0)));

      //update mpBTreeFile Root Pointer in BTreeFile
      mpBTreeFile->SetRoot(aState->GetMemPtr());
      delete aState;
      return TRUE;
    }

    if (aStatus == State::UPDATEROOT) {
      //update mpBTreeFile Root Pointer in BTreeFile
      mpBTreeFile->SetRoot(aState->GetMemPtr());
      delete aState;
      return TRUE;
    }

    //get the parent page of the current one
    mpCurrentPage = mpCurrentPage->mpUpperNode;

    // problem delete aState -- memory leaking here

    aState = mpCurrentPage->mpPage->Insert(*(aState->GetKey()),
   aState->GetMemPtr());
  }
  delete aState;
  return TRUE;
}


//
// Service: Cursor::deleteRec
// Description:
//   Delete a Record from BTree
//
Bool Cursor::deleteRec ()
{
  int aParentPos;
  Node* aUpperNode = mpCurrentPage->GetUpperNode();

  if (aUpperNode) {
    aParentPos = aUpperNode->GetPosition();
  }

  int aPosition = mpCurrentPage->GetPosition();
  State* aState = mpCurrentPage->mpPage->Remove(aPosition, aPosition,
aParentPos);

  State::Status aStatus;
  while ((aStatus = aState->GetStatus()) != State::COMPLETE) {
    if (aStatus == State::ERROR) {
      delete aState;
      return FALSE;
    }
    if (aStatus == State::NEWROOT) {
      mpBTreeFile->SetRoot(aState->GetMemPtr());
      delete aState;
      return TRUE;
    }
    if (aStatus == State::MERGE) {
      //move to upper page layer
      mpCurrentPage = mpCurrentPage->mpUpperNode;
      if ((aUpperNode = mpCurrentPage->GetUpperNode()) != NULL) {
aParentPos = aUpperNode->GetPosition();
      }
      aPosition = mpCurrentPage->GetPosition();
      if(aState->GetRL() == State::LEFT) {
        delete aState;
aState = mpCurrentPage->mpPage->Remove(aPosition-1,
       aPosition,
       aParentPos);
      }
      else {
        delete aState;
aState = mpCurrentPage->mpPage->Remove(aPosition,
                                               aPosition,
                                               aParentPos);
      }
    }
  }
  delete aState;
  return TRUE;
}


#ifdef _DEBUG
//
// Service: Cursor::print
// Title:   print the content of cursor
// Description:
```

```
//
void Cursor::print()
{
  cout << "  < C u r s o r >" << endl;
  if(mbKeyFound) {
    cout << "    mbKeyfound : TRUE " << endl;
  }
  else {
    cout << "    mbKeyfound : FALSE" << endl;
  }
  cout << "    mKey         : " << mKey.GetKey() << endl;

  if (mpNodes) {
    cout << "    mpNodes      : " << endl;
    Node* node = mpNodes;
    while (node) {
      node->print();
      node = node->GetUpperNode();
      cout << endl;
    }
  }
}
#endif
```

Appendix K

Definition and Implementation of Class FileFactory

K.1 Definition of Class FileFactory

```
/*
** FileFactory.h
**
** FileFactory Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-July-1997 Steven Li
**   Initial implemention
*/


#ifndef FILEFACTORY_H
#define FILEFACTORY_H

/*
** forward declaration
*/

class BTreeFile;
class PageFile;
class IndexPageFile;
class LeafPageFile;
class String;

/*
** include files
*/


/*
** Macro to define Exception classes
*/
DEFINE_EXCEPTION_FAMILY (FileFactoryExc, BTreeException)
DEFINE_EXCEPTION (FileErrUnsupportedFileNameExc, FileFactoryExc,
                  "Specified filename is not supported")
```

```
DEFINE_EXCEPTION (FileErrCreateBTreeFileExc, FileFactoryExc,
                  "cannot create a btree file")
DEFINE_EXCEPTION (FileErrWrongOrderExc, FileFactoryExc,
                  "Specified BTree File has different order")

/*
** Definition of class FileNode
*/

class FileNode {
  friend class FileFactory;
private:
  String      mBTreeFileName;
  BTreeFile*  mBTreeFile;
  FileNode*   mNext;
private:
  FileNode(const String&, BTreeFile*);
  virtual ~FileNode();

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  FileNode();                              //NOT DEFINED
  FileNode(const FileNode&);               //NOT DEFINED
  FileNode& operator = (const FileNode&); //NOT DEFINED
};


/*
** Definition of class FileFactory
*/

class FileFactory {
public:
  static FileFactory* Instance();
  BTreeFile* InstantiateBTreeFile(const String& filename,
                                  const int order);
  PageFile* GetPageFile(const String& filename);
  Bool DestroyBTreeFile(const String& filename);
  String* MakePageFileName(String aPageFileName, String aExt);

#ifdef _DEBUG
  static void print();
#endif

private:
  static FileFactory* mpFileFactory;
  static FileNode* mFiles;
  enum PAGEFILE_TYPE {INDEX_FILE, LEAF_FILE, BTREE_FILE, UNKNOWN};
private:
  FileFactory();
  ~FileFactory();
  static PAGEFILE_TYPE GetFileType(const String filename);
  static Bool AddFile(FileNode& file);
private:
  FileFactory(const FileFactory& factory);        // NOT DEFINED
  FileFactory& operator = (FileFactory& factory); // NOT DEFINED
};

#endif
```

K.2 Implementation of Class FileFactory

```
/*
** FileFactory.cxx
**
** FileFactory Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions
**   25-July-1997 Steven Li
**   Initial version
*/


/*
** include files
*/
#include <stdlib.h>
#include "File.h"
#include "BTString.h"
#include "BTreeFile.h"
#include "PageFile.h"
#include "FileFactory.h"


FileFactory* FileFactory::mpFileFactory = NULL;
FileNode* FileFactory::mFiles = NULL;

/*
** Implementation of class FileNode
*/


/*
** Service: FileNode::FileNode
** Description:
**   Constructor.
*/
FileNode::FileNode(const String& name, BTreeFile* btreefile)
  : mNext (NULL) {
  mBTreeFileName = name;
  mBTreeFile     = btreefile;
}


/*
** Service: FileNode::~FileNode
** Description:
**   Destructor.
*/
FileNode::~FileNode() {
  delete mBTreeFile;
}


#ifdef _DEBUG
//
// Service: FileNode::~FileNode
// Description:
//
void FileNode::print () const
{
  cout << "    < F i l e N o d e > " << endl;
```

```
   cout << "      mBTreeFileName  : " << mBTreeFileName << endl;
   cout << "      mBTreeFile      : " << endl;
  mBTreeFile->print();
}
#endif



//
// Implementation of class FileFactory
//

/*
** Service: FileFactory::FileFactory
** Description:
**   Default constructor.
*/
FileFactory::FileFactory() {
}


/*
** Service: FileFactory::~FileFactory
** Description:
**   Destructor.
*/
FileFactory::~FileFactory() {
  FileNode* node = mFiles;
  while (node) {
    node = node->mNext;
    delete mFiles;
    mFiles = node;
  }
}


/*
** Service: FileFactory::Instance()
** Description:
**   Makes a static instance of FileFactory
*/
FileFactory* FileFactory::Instance() {
  if (!mpFileFactory) {
    mpFileFactory = new FileFactory();
  }
  return mpFileFactory;
}


/*
** Service: PageFileFactory::InstantiateBTreeFile
** Description:
**   Instantiates a BTreeFile. The supplied filename must follow
**   the rules below:
**     BTreeFile     btree_name + . + file_extension (btf)
**     IndexPageFile btree_name + . + file_extension (idx)
**     LeafPageFile  btree_name + . + file_extension (lef)
**
**   the supplied filename must be:
**     btree_name.btf
*/
BTreeFile* FileFactory::InstantiateBTreeFile(const String& filename,
                                             const int order) {
  if (GetFileType(filename) != BTREE_FILE) {
    throw FileErrUnsupportedFileNameExc();
  }
```

```
    //find the btree file in the mFiles list
    FileNode* node = mFiles;
    while(node) {
      if (filename == node->mBTreeFileName) {
        if (order != node->mBTreeFile->GetOrder()) {
          return node->mBTreeFile;
        }
        throw FileErrWrongOrderExc();
      }
      node = node->mNext;
    }

    //instantiate a new BTreeFile
    BTreeFile* btreefile = new BTreeFile(filename, order);

    node = new FileNode(filename, btreefile);
    if (!AddFile(*node)) {
      throw FileErrCreateBTreeFileExc();
    }
    return btreefile;
  }


  /*
  ** Service: PageFileFactory::InstantiatePageFile
  ** Description:
  **   Instantiates a PageFile. The supplied filename must follow
  **   the rules below:
  **     BTreeFile     btree_name + . + file_extension (btf)
  **     IndexPageFile btree_name + . + file_extension (idx)
  **     LeafPageFile  btree_name + . + file_extension (lef)
  */
  PageFile* FileFactory::GetPageFile(const String& filename) {

    String* btreefile_name = MakePageFileName(filename, BTREE_EXT);

    //find the btree file in the mFiles list
    FileNode* node = mFiles;
    while(node) {
      if (*btreefile_name == node->mBTreeFileName) {
        if (GetFileType(filename) == INDEX_FILE) {
          return node->mBTreeFile->GetIndexPageFile();
        }
        else if (GetFileType(filename) == LEAF_FILE) {
          return node->mBTreeFile->GetLeafPageFile();
        }
        throw FileErrUnsupportedFileNameExc();
      }
      node = node->mNext;
    }
    return NULL;
  }


  /*
  ** Service: FileFactory::DestroyBTreeFile
  ** Description:
  */
  Bool FileFactory::DestroyBTreeFile(const String& filename)
  {
    //find the btree file in the mFiles list
    FileNode* node = mFiles;
    FileNode* prevnode = node;
    while(node) {
      if (filename == node->mBTreeFileName) {
```

```
      prevnode->mNext = node->mNext;
      delete node;
      return TRUE;
    }
    prevnode = node;
    node = node->mNext;
  }
  return FALSE;
}


/*
** Service: FileFactory::GetFileType
** Description:
*/
FileFactory::PAGEFILE_TYPE FileFactory::GetFileType (const String
     filename){
  if (filename.Overlaps(INDEX_EXT)) {
    return INDEX_FILE;
  }
  else if (filename.Overlaps(LEAF_EXT)) {
    return LEAF_FILE;
  }
  else if (filename.Overlaps(BTREE_EXT)) {
    return BTREE_FILE;
  }
  throw FileErrUnsupportedFileNameExc();
  return UNKNOWN;
}


/*
** Service: FileFactory::MakePageFileName
** Description:
**   Make PageFile Name for each PageFiles (BTree, Index, Leaf)
*/
String* FileFactory::MakePageFileName(String aBTreeFileName,
      String aExt){
  String iPageFileName;

  iPageFileName = aBTreeFileName(0, (aBTreeFileName.GetLen()
     - aExt.GetLen()));
  iPageFileName += aExt;
  return (new String(iPageFileName));
}


/*
** Service: FileFactory::AddFile
** Description:
**   Make PageFile Name for each PageFiles (BTree, Index, Leaf)
*/
Bool FileFactory::AddFile(FileNode& file) {
  file.mNext = mFiles;
  mFiles = &file;
  return TRUE;
}


#ifdef _DEBUG
//
// Service: FileFactory::print
// Description:
//
void FileFactory::print()
{
  cout << "  < F i l e F a c t o r y > " << endl;
  if (mFiles) {
    cout << "  mFiles : " << endl;
    FileNode* node = mFiles;
    while (node) {
      node->print();
      node = node->mNext;
    }
  }
}
#endif
```

Appendix L

Definition and Implementation of Class PageKey

L.1 Definition of Class PageKey

```
/*
** PageKey.h
**
** PageKey Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   31-July-1997 Steven Li
**   added persistent BufferRead(), BufferWrite(), Make()
**   added operator <, !=, >=
*/


#ifndef PAGEKEY_H
#define PAGEKEY_H

/*
** forward declarations
*/

class BLOBScanner;
class BLOBMaker;

#include "Boolean.h"

class PageKey{
public:
  PageKey(const int aKeyVal);
  ~PageKey();
  PageKey(const PageKey& aPageKey);
  PageKey& operator = (const PageKey& aPageKey);

  static PageKey* Make (BLOBScanner& blob);
```

```
    Bool operator <  (const PageKey& aPageKey) const;
    Bool operator <= (const PageKey& aPageKey) const;
    Bool operator == (const PageKey& aPageKey) const;
    Bool operator != (const PageKey& aPageKey) const;
    Bool operator >  (const PageKey& aPageKey) const;
    Bool operator >= (const PageKey& aPageKey) const;

    int GetKey() const;
    void SetKey(const int aKeyVal);

    static int GetSize();
    Bool BufferRead (BLOBScanner& blob);
    Bool BufferWrite (BLOBMaker& blob) const;

#ifdef _DEBUG
  virtual void print() const;
#endif

private:
  int mKeyVal;
private:
  PageKey();
};

#endif
```

L.2 Implementation of Class PageKey

```
/*
** PageKey.cxx
**
** PageKey Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions
**   23-March-1996 Steven Li
**   Inital version
**
**   25-July-1997 Steven Li
**   made ctor() private.
**   added persistent BufferRead(), BufferWrite(), Make()
**   added operator <, !=, >=
**
*/

#include <stdlib.h>
#include <assert.h>
#include <iostream.h>
#include "Blob.h"
#include "PageKey.h"

/*
** Service: PageKey::PageKey
** Description:
**   Default Constructor. It is not allowed to use
**   the default constructor to build a PageKey.
**   It is used internally.
*/
PageKey::PageKey(){
  mKeyVal = 0;
}


/*
** Service: PageKey::PageKey
** Description:
**   Constructor
*/
PageKey::PageKey(const int aKeyVal){
  mKeyVal = aKeyVal;
}


/*
** Service: PageKey::~PageKey
** Title:   Destructor
** Description:
*/
PageKey::~PageKey(){}


/*
** Service: PageKey::PageKey
** Title:   Copy Constructor
** Description:
*/
PageKey::PageKey(const PageKey& aPageKey){
  mKeyVal = aPageKey.mKeyVal;
}


/*
** Service: PageKey::operator =
** Title:   Assignment
** Description:
*/
PageKey& PageKey::operator = (const PageKey& apPageKey){
  mKeyVal = apPageKey.mKeyVal;
  return *this;
}


/*
** Service: PageKey::operator <=
** Title:   Comparison
** Description:
*/
Bool PageKey::operator <= (const PageKey& aPageKey) const {
  return (mKeyVal <= aPageKey.mKeyVal) ? TRUE : FALSE;
}


/*
** Service: PageKey::operator >
** Title:   Comparison
** Description:
*/
Bool PageKey::operator > (const PageKey& aPageKey) const {
  return (mKeyVal > aPageKey.mKeyVal) ? TRUE : FALSE;
}


/*
** Service: PageKey::operator ==
** Title:   Comparison
** Description:
*/
Bool PageKey::operator == (const PageKey& aPageKey) const {
  return (mKeyVal == aPageKey.mKeyVal) ? TRUE : FALSE;
}

/*
** Service: PageKey::operator !=
** Title:   Comparison
** Description:
*/
Bool PageKey::operator != (const PageKey& aPageKey) const {
  return (PageKey::operator == (aPageKey)) ? FALSE : TRUE;
}


/*
** Service: PageKey::operator <
** Title:   Comparison
** Description:
*/
Bool PageKey::operator <  (const PageKey& aPageKey) const {
  return (mKeyVal < aPageKey.mKeyVal) ? TRUE : FALSE;
}


/*
** Service: PageKey::operator >=
** Title:   Comparison
** Description:
*/
Bool PageKey::operator >=  (const PageKey& aPageKey) const {
  return (mKeyVal >= aPageKey.mKeyVal) ? TRUE : FALSE;
}


/*
** Service: PageKey::GetKey
** Title:   Get KeyVal of a PageKey
** Description:
*/
int PageKey::GetKey() const {
  return mKeyVal;
}


/*
** Service: PageKey::SetKey
** Title:   Set KeyVal of a PageKey
** Description:
*/
void PageKey::SetKey(const int aKeyVal){
  mKeyVal = aKeyVal;
}


/*
** Service: PageKey::GetSize
** Title:   Get size of a PageKey
** Description:
*/
int PageKey::GetSize() {
  return sizeof(int);
}
```

```
/*
** Service: PageKey::BufferRead
** Description:
*/
Bool PageKey::BufferRead (BLOBScanner& blob) {
  mKeyVal = (int)blob;
  return TRUE;
}


/*
** Service: PageKey::BufferWrite
** Description:
*/
Bool PageKey::BufferWrite (BLOBMaker& blob) const {
  blob += (int) mKeyVal;
  return TRUE;
}


/*
** Service: PageKey::Make
** Description:
*/
PageKey* PageKey::Make (BLOBScanner& blob) {
  PageKey* akey = new PageKey ();
  if (!akey->BufferRead (blob)) {
    delete akey;
    akey = NULL;
  }
  return akey;
}


#ifdef _DEBUG
//
// Service: PageKey::print
// Description:
//
void PageKey::print () const
{
  cout << "  < P a g e K e y > " << endl;
  cout << "    mKeyVal : " << mKeyVal << endl;
}
#endif
```

Appendix M

Definition and Implementation of Class Iterator

M.1 Definition of Class Iterator

```
/*
** Iterator.h
**
** BTreeFile Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** March, 1996
**
*/

#ifndef ITERATOR_H
#define ITERATOR_H

/*
** forward declaration
*/
class MemPtr;
class String;

/*
** include files
*/
#include "Boolean.h"


class Iterator {
public:
  virtual ~Iterator() {};
  virtual void First () = 0;
  virtual void Next () = 0;
  virtual Bool IsDone () = 0;
  virtual Pointer* page () const = 0;
  virtual String* blob () const = 0;
protected:
  Iterator() {};
};
```

```
#endif
```

Appendix N

Definition and Implementation of Class Blob

N.1 Definition of Class Blob

```
/*
** Blob.h
**
** BLOBMaker and BLOBScaner class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-July-1997 Steven Li
**   Initial implementions
*/


#ifndef BLOB_H
#define BLOB_H

#include "Boolean.h"


// BLOBMaker class definition.

class BLOBMaker  {
public:
  BLOBMaker (const unsigned size);
  virtual ~BLOBMaker ( );
  BLOBMaker& operator += (const char *achar);
  BLOBMaker& operator += (const char achar);
  BLOBMaker& operator += (const double a_double);
  BLOBMaker& operator += (const float a_float);
  BLOBMaker& operator += (const int a_int);
  BLOBMaker& operator += (const long double a_double);
  BLOBMaker& operator += (const long int a_long);
  BLOBMaker& operator += (const short int a_short);
  BLOBMaker& operator += (const unsigned char achar);
  BLOBMaker& operator += (const unsigned int a_int);
  BLOBMaker& operator += (const unsigned long int a_long);
  BLOBMaker& operator += (const unsigned short int a_short);
  void* hasBLOB ( ) const;
  unsigned hasBLOBSize ( ) const;
  Bool putWildData (const void *data,
                    const unsigned data_length);
  void reset ( );
private:
  unsigned mSize;
  char*    mpIterator;
  char*    mpBlob;
private:
  BLOBMaker ( );
  BLOBMaker& operator = (const BLOBMaker& blob);
  BLOBMaker (const BLOBMaker& blob);
};


// --- BLOBScanner class definition.

class BLOBScanner  {
public:
  BLOBScanner (const void *blob);
  virtual ~BLOBScanner ( );
  Bool advance (const unsigned bytes);
  Bool copySubBLOB (const unsigned data_length, void *client_blob);
  void* toWildData (const unsigned data_length);
  operator char  ( );
  operator char* ( );
  operator double ( );
  operator float  ( );
  operator int ( );
  operator long int ( );
  operator long double ( );
  operator short int   ( );
  operator unsigned char ( );
  operator unsigned int  ( );
  operator unsigned long int  ( );
  operator unsigned short int ( );
protected:
  char  *mpIterator;
  void  *mpBlob;
private:
  BLOBScanner ( );
  BLOBScanner& operator = (const BLOBScanner& blob);
  BLOBScanner (const BLOBScanner& blob);
};

#endif
```

N.2 Implementation of Class Blob

```
/*
** Blob.cxx
**
** BLOBMaker and BLOBScaner Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-July-1997 Steven Li
**   Initial implemention
```

```
**
*/

#include <memory.h>
#include <string.h>
#include <assert.h>
#include "Blob.h"

// --- BLOBMaker class implementation.

/*
** Service: BLOBMaker::BLOBMaker
** Title:   Constructor
** Description:
*/
BLOBMaker::BLOBMaker (const unsigned size)
  : mSize(size) {
  mpBlob = new char [size];
  assert(mpBlob);
  ::memset(mpBlob, 0, size);
  mpIterator = mpBlob;
}


/*
** Service: BLOBMaker::~BLOBMaker
** Title:   Destructor
** Description:
**   destroys the blob maker. The blob is deleted.
*/
BLOBMaker::~BLOBMaker ( ) {
  delete mpBlob;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const char *achar)
** Title:   appends a charater string
** Description:
**   appends a null terminated string at the end of the blob
**   (including the null character). A null string is appended
**   to the blob as a empty string. This function returns a
**   reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const char *achar) {
  if (achar == NULL)  {
    return operator += ('\0');
  }

  size_t len = ::strlen (achar) + 1;
  ::memcpy(mpIterator, achar, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const char achar)
** Title:   appends a charater
** Description:
**   appends a char at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const char achar) {
  size_t len = sizeof(char);
  ::memcpy(mpIterator, (char*)&achar, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const double adouble)
** Title:   appends a double
** Description:
**   appends a double at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const double adouble) {
  size_t len = sizeof(double);
  ::memcpy(mpIterator, (char*)&adouble, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const float afloat)
** Title:   appends a float
** Description:
**   appends a float at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const float afloat) {
  size_t len = sizeof(float);
  ::memcpy(mpIterator, (char*)&afloat, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const int aint)
** Title:   appends an integer
** Description:
**   appends an int at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const int aint) {
  size_t len = sizeof(int);
  ::memcpy(mpIterator, (char*)&aint, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const long double adouble)
** Title:   appends a long double
** Description:
**   appends a long double at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const long double adouble) {
  size_t len = sizeof(long double);
  ::memcpy(mpIterator, (char*)&adouble, len);
  mpIterator += len;
  return *this;
}


/*
```

```
** Service: BLOBMaker& BLOBMaker::operator += (const long int along)
** Title:   appends a long integer
** Description:
**   appends a long int at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const long int along) {
  size_t len = sizeof(long int);
  ::memcpy(mpIterator, (char*) &along, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const short int ashort)
** Title:   appends a short integer
** Description:
**   appends an short int at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const short int ashort) {
  size_t len = sizeof(short int);
  ::memcpy(mpIterator, (char*)&ashort, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const unsigned char achar)
** Title:   appends an unsigned character.
** Description:
**   appends an unsigned char at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const unsigned char achar) {
  size_t len = sizeof(unsigned char);
  ::memcpy(mpIterator, (char*)&achar, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const unsigned int aint)
** Title:   appends an unsigned integer
** Description:
**   appends an unsigned int at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const unsigned int aint) {
  size_t len = sizeof(unsigned int);
  ::memcpy(mpIterator, (char*) &aint, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const unsigned long int along)
** Title:   appends an unsigned long integer
** Description:
**   appends an unsigned long int at the end of the blob.
**   This function returns a reference to this blob maker.
*/
```

```
BLOBMaker& BLOBMaker::operator += (const unsigned long int along) {
  size_t len = sizeof(unsigned long int);
  ::memcpy(mpIterator, (char*)&along, len);
  mpIterator += len;
  return *this;
}


/*
** Service: BLOBMaker& BLOBMaker::operator += (const unsigned short int ashort)
** Title:   appends an unsigned short integer
** Description:
**   appends an unsigned short int at the end of the blob.
**   This function returns a reference to this blob maker.
*/
BLOBMaker& BLOBMaker::operator += (const unsigned short int ashort) {
  size_t len = sizeof(unsigned short int);
  ::memcpy(mpIterator, (char*)&ashort, len);
  mpIterator += len;
  return *this;
}


/*
** Service: void* BLOBMaker::hasBLOB ( ) const
** Title:
** Description:
**   returns the blob build up to the call of this function.
**   This function returns a void pointer to the blob.
**   A null pointer is returned if the blob is empty.
*/
void* BLOBMaker::hasBLOB ( ) const {
  return (void*)mpBlob;
}


/*
** Service: unsigned BLOBMaker::hasBLOBSize ( ) const
** Title:
** Description:
**   returns the size, in bytes, of the blob build up to the
**   call of this function.
*/
unsigned BLOBMaker::hasBLOBSize ( ) const {
  return mSize;
}


/*
** Service: BLOBMaker::putWildData
** Title:   appends wild raw data
** Description:
**   appends a block of unknown type of data at the end
**   of the blob. The argument "data" supplies the block
**   of data while the argument "data_length" specifies
**   the size, in bytes, of "data". This function always
**   returns TRUE indicating that the insertion was successful.
*/
Bool BLOBMaker::putWildData (const void *data,
                             const unsigned data_length) {
  ::memcpy(mpIterator, (char*) data, data_length);
  mpIterator += data_length;
  return TRUE;
  }
```

```
// --- BLOBScanner class implementation.

/*
** Service: BLOBScanner::BLOBScanner (const void *blob)
** Title:   constructor
** Description:
**   constructs a blob scanner. It will scan the supplied blob
**   (generated by an object of the class BLOBMaker).
**   The scan start at the beginning of the supplied blob.
*/
BLOBScanner::BLOBScanner (const void *blob)
  : mpBlob ((void *) blob) {
  mpIterator = (char*) blob;
}


/*
** Service: BLOBScanner::~BLOBScanner ()
** Title:   destructor
** Description:
** destroys this blob scanner. It does nothing explicitly.
*/
BLOBScanner::~BLOBScanner () {}


/*
** Service: Bool BLOBScanner::advance (const unsigned bytes)
** Title:   advance bytes forward.
** Description:
**   skips the next number of bytes (supplied by the argument)
**   in the blob.
*/
Bool BLOBScanner::advance (const unsigned bytes) {
  mpIterator += bytes;
  return TRUE;
}


/*
** Service: void* BLOBScanner::toWildData (const unsigned data_len)
** Title:   extracts wild raw data
** Description:
**   converts the blob next (data_length) bytes of data to be scanned
**   as a block of unknown type of data. This function returns a void
**   pointer to memory containing the data. The client is responsible
**   returning the allocated memory back to the operating system.
*/
void* BLOBScanner::toWildData (const unsigned data_length) {
  void *data = (void *) new char [data_length];
  ::memcpy (data, (void *) mpIterator, (size_t) data_length);
  mpIterator += data_length;
  return data;
}


/*
** Service: BLOBScanner::operator char ()
** Title:   extract a charater
** Description:
**   converts the blob next bytes of data to be scanned as a char.
**   This function returns a char.
*/
BLOBScanner::operator char () {
  char achar;
  size_t len = sizeof(char);
  ::memcpy ((void *) &achar, (void *) mpIterator, len);
  mpIterator += len;
  return achar;
}


/*
** Service: BLOBScanner::operator char* ()
** Title:   extracts a charater string
** Description:
**   converts the blob next bytes of data to be scanned as a null
**   terminated string. This function returns a pointer to a null
**   terminated string. A null pointer is returned if the string
**   is empty (length of 0 bytes). The client is responsible returning
**   the allocated memory back to the operating system.
*/
BLOBScanner::operator char* () {
  if (*mpIterator == '\0')  {
    mpIterator += sizeof(char);
    return NULL;
  }
  size_t length = ::strlen (mpIterator) +1;
  char *string = new char [length];
  ::strcpy (string, mpIterator);
  mpIterator += length;
  return string;
}


/*
** Service: BLOBScanner::operator double ()
** Title:   extracts a double
** Description:
**   converts the blob next bytes of data to be scanned as a double.
**   This function returns a double.
*/
BLOBScanner::operator double () {
  double a_double;
  size_t len = sizeof(double);
  ::memcpy ((void *) &a_double, (void *) mpIterator, len);
  mpIterator += len;
  return a_double;
}


/*
** Service: BLOBScanner::operator float ()
** Title:   extracts a float
** Description:
**   converts the blob next bytes of data to be scanned as a float.
**   This function returns a float.
*/
BLOBScanner::operator float () {
  float a_float;
  size_t len = sizeof(float);
  ::memcpy ((void *) &a_float, (void *) mpIterator, len);
  mpIterator += len;
  return a_float;
}


/*
** Service: BLOBScanner::operator int ()
** Title:   extracts an integer
** Description:
** converts the blob next bytes of data to be scanned as an int.
** This function returns an int.
```

```
*/
BLOBScanner::operator int () {
  int a_int;
  size_t len = sizeof(int);
  ::memcpy ((void *) &a_int, (void *) mpIterator, len);
  mpIterator += len;
  return a_int;
}


/*
** Service: BLOBScanner::operator long int ()
** Title:   extracts a long integer
** Description:
**   converts the blob next bytes of data to be scanned as a long int.
**   This function returns a long int.
*/
BLOBScanner::operator long int () {
  long int a_long;
  size_t len = sizeof(long int);
  ::memcpy ((void *) &a_long, (void *) mpIterator, len);
  mpIterator += len;
  return a_long;
}


/*
** Service: BLOBScanner::operator long double ()
** Title:   extracts a long double
** Description:
**   converts the blob next bytes of data to be scanned as a long
**   double. This function returns a long double.
*/
BLOBScanner::operator long double () {
  long double a_double;
  size_t len = sizeof(long double);
  ::memcpy ((void *) &a_double, (void *) mpIterator, len);
  mpIterator += len;
  return a_double;
}


/*
** Service: BLOBScanner::operator short int ()
** Title:   extracts a short integer
** Description:
**   converts the blob next bytes of data to be scanned as a short int.
**   This function returns a short int.
*/
BLOBScanner::operator short int () {
  short int a_short;
  size_t len = sizeof(short int);
  ::memcpy ((void *) &a_short, (void *) mpIterator, len);
  mpIterator += len;
  return a_short;
}


/*
** Service: BLOBScanner::operator unsigned char ()
** Title:   extracts an unsigned charater
** Description:
**   converts the blob next bytes of data to be scanned as an
**   unsigned char. This function returns an unsigned char.
*/
BLOBScanner::operator unsigned char () {
  unsigned char achar;
  size_t len = sizeof(unsigned char);
  ::memcpy ((void *) &achar, (void *) mpIterator, len);
  mpIterator += len;
  return achar;
}


/*
** Service: BLOBScanner::operator unsigned int ()
** Title:   extracts an unsigned integer
** Description:
**   converts the blob next bytes of data to be scanned as an
**   unsigned int. This function returns an unsigned int.
*/
BLOBScanner::operator unsigned int () {
  unsigned int a_int;
  size_t len = sizeof(unsigned int);
  ::memcpy ((void *) &a_int, (void *) mpIterator, len);
  mpIterator += len;
  return a_int;
}


/*
** Service: BLOBScanner::operator unsigned long int ()
** Title:   extracts a unsigned long integer
** Description:
**   converts the blob next bytes of data to be scanned as an
**   unsigned long int. This function returns an unsigned long int.
*/
BLOBScanner::operator unsigned long int () {
  unsigned long int a_long;
  size_t len = sizeof(unsigned long int);
  ::memcpy ((void *) &a_long, (void *) mpIterator, len);
  mpIterator += len;
  return a_long;
}


/*
** Service: BLOBScanner::operator unsigned short int ()
** Title:   extracts a unsigned short integer
** Description:
**   converts the blob next bytes of data to be scanned as an unsigned
**   short int. This function returns an unsigned short int.
*/
BLOBScanner::operator unsigned short int () {
  unsigned short int a_short;
  size_t len = sizeof(unsigned short int);
  ::memcpy ((void *) &a_short, (void *) mpIterator, len);
  mpIterator += len;
  return a_short;
}
```

Appendix O

Definition and Implementation of Class State

O.1 Definition of Class State

```
/*
** State.h
**
** State Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
*/


#ifndef STATE_H
#define STATE_H

/*
** include files
*/
#include "File.h"


/*
** forward declaration
*/
class PageKey;
class Pointer;
class MemPtr;
class Page;


//
// Definition of class State
//

class State {
public:
  //the intermediate status value for deleting and inserting
  enum Status { SPLIT,   MERGE,
                ERROR,   COMPLETE,
NEWROOT, UPDATEROOT };

  // Right or Left sibling Page indicator
  enum R_L { RIGHT, LEFT };

public:
  State (const PageKey& aKey, MemPtr& aMemPtr, Status aStatus);
  State (R_L aSibling, Status aStatus);
  State (Status aStatus);
  State (MemPtr& aMemPtr, Status aStatus);

  ~State();
  Status GetStatus() const;
  void SetStatus(Status aStatus);

  Pointer* GetMemPtr() const;
  void SetMemPtr(Pointer& aMemPtr);

  PageKey* GetKey() const;
  void SetKey(PageKey& apKey);

  R_L GetRL() const;
  void SetRL(R_L aRL);

private:
  Pointer* mMPtr;     //MemPtr to a new sibling/root page
  PageKey* mKey;      //key to distinguish the new page
  Status   mStatus;   //operation status
  R_L      mSibling;  //right or left sibling indicator
private:
  State ();                                   // NOT DEFINED
  State (const State& aState);                // NOT DEFINED
  State& operator = (const State& aState);    // NOT DEFINED
};

#endif
```

O.2 Implementation of Class State

```
/*
** State.cxx
**
** State Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
*/


//
// include files
//
#include <stdlib.h>
```

```
#include "State.h"
#include "MemPtr.h"
#include "PageKey.h"
#include "Page.h"


/*
** Service: State::State
** Title:   Constructor
** Description:
*/
State::State(const PageKey& aKey, MemPtr& aMemPtr, Status aStatus)
{
  mKey = (PageKey*)(&aKey);
  mMPtr = &aMemPtr;
  mStatus = aStatus;
  mSibling = LEFT;  //by default
}


/*
** Service: State::State
** Title:   Constructor
** Description:
*/
State::State(Status aStatus){
  mKey = NULL;
  mMPtr = NULL;
  mStatus = aStatus;
  mSibling = LEFT;  //by default
}


/*
** Service: State::State
** Title:   Constructor
** Description:
*/
State::State(MemPtr& aMemPtr, Status aStatus)
{
  mKey = NULL;
  mMPtr = aMemPtr;
  mStatus = aStatus;
  mSibling = LEFT;
}


/*
** Service: State::State
** Title:   Constructor
** Description:
*/
State::State(R_L aRL, Status aStatus){
  mKey = NULL;
  mMPtr = NULL;
  mStatus = aStatus;
  mSibling = aRL;
}


/*
** Service: State::GetStatus
** Title:   Get operation Status
** Description:
*/
State::Status State::GetStatus() const{
  return mStatus;
}


/*
** Service: State::~State
** Title:   Destructor
** Description:
*/
State::~State()
{
  if (mMPtr) {
    delete mMPtr;
  }
  if (mKey) {
    delete mKey;
  }
}


/*
** Service: State::GetMemPtr
** Title:   Get MemPtr
** Description:
*/
Pointer* State::GetMemPtr() const{
  return new MemPtr(*(mMPtr->DeReference()));
}


/*
** Service: State::GetKey
** Title:   Get PageKey
** Description:
*/
PageKey* State::GetKey() const{
  return new PageKey(*mKey);
}


/*
** Service: State::GetRL
** Title:   Get RL direction
** Description:
*/
State::R_L State::GetRL() const{
  return  mSibling;
}
```

Appendix P

Definition and Implementation of Class BTString

P.1 Definition of Class BTString

```
/*
** BTString.h
**
** Pointer Class Definition File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** March, 1996
**
** Description:
**    Consider Pointer as an Interface virtual class.
*/

#ifndef BTSTRING_H
#define BTSTRING_H

/*
** include files
*/
#include <iostream.h>
#include <assert.h>
#include <string.h>

#include "Boolean.h"

class String {

  friend String operator + (String&, String&);

public:

  String(const char*);
  String(const String&);
  String();
  ~String() { delete [] str; }

  String operator () (int, int);
  char&  operator [] (int);

  String& operator = (const char*);
  String& operator = (const String&);
  String& operator += (const String&);
  Bool operator == (const String &s);
  Bool operator == (const char *s);
  Bool operator !();

  unsigned int GetLen() { return len; }
  Bool operator < (String& s);
  Bool operator > (String& s);
  Bool Overlaps (const char*) const;
  void Set(char*);

  operator const char* () const;

private:
  int len;
  char *str;
};

#endif
```

P.2 Implementation of Class BTString

```
/*
** String.cxx
**
** String Class Implementation File
**
** BTree Project
** Steven Li
** Computer Science
** Concordia University
** Supervised by Dr. Greg Butler
**
** Revisions:
**   12-Jan-1997 Steven Li
**   Initial implemention
**
**   03-Mar-1997 Steven Li
**
*/

/*
** include files
*/
#include <iostream.h>
#include <string.h>

#include "Boolean.h"
#include "BTString.h"
#include "BTreeException.h"


/*
** Service: String::String
** Title:   Default Constructor
** Description:
*/
String::String(){
  len = 0;
  str = 0;
}
```

```
/*
** Service: String::String
** Title:   Constructor
** Description:
*/
String::String(const char *s) {
  if (!s) {
    len = 0;
    str = 0;
  }
  else {
    len = ::strlen(s);
    str = new char[len + 1];
    ::strcpy(str, s);
  }
}


/*
** Service: String::String
** Title:   Copy Constructor
** Description:
*/
String::String(const String& s){
  len = s.len;
  if ((str = s.str) == 0)
    return;

  str = new char[len + 1];
  ::strcpy(str, s.str);
}


/*
** Service: String::operator =
** Title:   String Assignment
** Description:
*/
String& String::operator = (const String& s){
  if (this == &s) return *this;
  if(str)
    delete [] str;

  len = s.len;
  if ((str = s.str) == 0)  return *this;

  str = new char[len + 1];
  ::strcpy(str, s.str);
  return *this;
}


/*
** Service: String::operator =
** Title:   String Assignment
** Description:
*/
String& String::operator = (const char *s){
  if (s == 0) {
    len = 0;
    str = 0;
    return *this;
  }
  len = ::strlen(s);
  delete [] str;
  str = new char[len + 1];
  ::strcpy(str, s);
  return *this;
}


/*
** Service: String::operator ==
** Title:   String Comparison
** Description:
*/
Bool String::operator == (const String &s){
  if (len != s.len) return FALSE;
  return(::strcmp(str,s.str) == 0) ? TRUE : FALSE;
}


/*
** Service: String::operator ==
** Title:   String Comparison
** Description:
*/
Bool String::operator == (const char *s) {
  return (::strcmp(str,s)==0) ? TRUE : FALSE;
}


/*
** Service: String::operator !()
** Title:   String IsNull
** Description:
*/
Bool String::operator !() {
  return (len == 0) ? TRUE : FALSE;
}


/*
** Service: String::operator []
** Title:   String Index
** Description:
*/
char& String::operator [] (int elem){
  return str[elem];
}


/*
** Service: String::operator ()
** Title:   Extract a sub-string
** Description:
*/
String String::operator()(int pos, int cnt) {
  assert (pos >= 0 && pos < len && cnt >= 0);

  if (cnt + pos - 1 > len) {
    throw BTreeErrOutOfBoundExc();
  }

  char *ps = new char[cnt + 1];
  assert(ps);

  for (int i = pos, j = 0; j < cnt; ++i, ++j)
    ps[j] = str[i];
  ps[cnt] = '\0';
  return String(ps);
}


/*
** Service: String::operator +=
** Title:   String Catenate
** Description:
*/
String& String::operator += ( const String &s ){
  assert(str);

  len += s.len;
  if ( !len )
    return *this;

  char *p = new char[len+1];
  assert(p);

  ::strcpy(p, str);
  ::strcat(p, s.str);
  delete [] str;
  str = p;
  return *this;
}


/*
** Service: String::operator +
** Title:   String Catenate
** Description:
*/
String operator + (const String &s1, const String &s2 ){
  String result = s1;
  result += s2;
  return result;
}


/*
** Service: String::operator <
** Title:   String Comparison
** Description:
*/
Bool String::operator < (String& s) {
  if (s.len == 0) return FALSE;
  if (len == 0) return TRUE;
  return(::strcmp(str,s.str) < 0) ? TRUE : FALSE;
}


/*
** Service: String::operator >
** Title:   String Comparison
** Description:
*/
Bool String::operator > (String& s) {
  if (len == 0) return FALSE;
  if (s.len == 0) return TRUE;
  return(::strcmp(str,s.str) > 0) ? TRUE : FALSE;
}


/*
** Service: String::operator const char*
** Title:   Get const char*
** Description:
*/
Bool String::Overlaps (const char* s) const {
  assert (s);
  return (::strstr(str, s) == NULL) ? FALSE : TRUE;
}


/*
** Service: String::operator const char*
** Title:   Get const char*
** Description:
*/
String::operator const char* () const {
  return str;
}
```