Generalizing Database Access Methods

by

Ming Zhou

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 1999

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Efficient implementation of access methods is crucial for many database systems. Today, database systems are increasingly being employed to support new applications involving multi-dimensional data. A large variety of specialized access methods have been developed to solve specific problems. These specialized access methods are usually hand-coded from scratch. The effort required to implement and maintain such data structures is high. As an alternative to developing new data structures from scratch, some researchers have tried to simplify search tree technology by generalization.

This thesis generalizes access methods at multiple levels, starting with hierarchical search-tree access methods. The basic logic and commonalities among different database search trees are explored. A generalized search tree is built and a generalized class library is created including a small set of specialized components. Specific search trees can be built significantly faster than using the traditional approach, using the generalized model and selecting components from the library. This work attempts to start a taxonomy for secondary-memory data structures, which is used both to classify functionality and reuse implementation. It is similar to the taxonomy provided by Leda/STL in C++ for primary-memory data structures. Example tree file structures are also developed and performance experiments on these file structures demonstrate the feasibility of the generalization approach to achieve efficient implementation of search-tree access methods.

Acknowledgements

I would like to thank my supervisor, Dr. Peter Buhr, for his valuable guidance, encouragement, mentoring, patience and hard work throughout my study and thesis work. His sense of humor always made my work enjoyable.

I also extend my gratitude to Dr. Naomi Nishimura and Dr. Frank Tompa who carefully read the final draft of the thesis and gave valuable suggestions.

I thank Anil Goel for providing information on μDatabase and taking time to attend our meetings and give good suggestions.

Furthermore, thanks to my friends and lab-mates Oliver Schuster, Ashif Harji, Dorota Zak and Tom Legrady for their help. Working in the Programming Language Lab with them is always pleasant and enjoyable. Thanks also extend to my officemate Curtis Cartmill, friends Tayfun Umman, Zhe Liu, Haihong and many other friends at Waterloo.

I thank my husband, Zhongbin Zhang, and my parents for their tremendous encouragement and unfailing support throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efficient data access is one of the properties provided by a database management system. The most common mechanism to achieve this goal is to associate an index with a large, randomly accessed data file on secondary storage. The index is an auxiliary data structure intended to help speed up the retrieval of records in response to certain search conditions. An index, like the labels on the drawers and folders in a filing cabinet, speeds up retrieval by directing the searcher to the small part of the file containing the desired item. For disk files, an index allows the number of disk accesses to be reduced. An index may be physically integrated with the file, like the labels on employee folders, or physically separate, like the labels on the drawers. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on. Organization techniques, or data structures, for index files are called access methods.

Many techniques for organizing database access methods have been proposed. Some of the most common one-dimensional structures include hashing and its variants, such as linear hashing [Lit80, Lar80] and extendible hashing [FNPS79], and the B-tree [BM72] and its variants [Com79].

Hierarchical access methods such as the B+-tree are scalable and behave well in the case of skewed input; they are nearly independent of the distribution of the input data. This property is not necessarily true for hashing techniques, where performance may degenerate depending on the given input data. This problem is aggravated by the use of order-preserving hash functions that try to preserve neighborhood relationships among data items, in order to support range queries. As a result, highly skewed data causes accumulation at a few selected locations in the hash table. For most data, on the other hand, hashing techniques usually outperform hierarchical methods on average.

In traditional relational systems, one-dimensional access methods such as the B+-tree and extendible hashing are sufficient for the sorts of queries posed on the usual set of alphanumeric data types. However, they are not suitable for indexing multi-dimensional data. A query such as , "Find all employees who are between 30 and 35 years old and earn a salary between 40k and 60k," is an example of a two-dimensional range query. One way of answering such a query is by maintaining one-dimensional indexes on age and salary, and using one or both indexes to narrow down the set of relevant employee records. Another way is to maintain a composite index on age and salary. But neither of these solutions is efficient, and therefore, specialized structures are required to handle multi-dimensional range queries.

Today, database systems are increasingly being employed to support new applications such as geographic information systems, multimedia systems, CAD tools, document libraries, DNA sequence databases, fingerprint identification systems, biochemical databases, and so on. These data management applications rely heavily on multi-dimensional data. For example, in a geographic information system, points, lines, line groups and polygons are used as basic data types. Retrieval and update of spatial data is usually based not only on the value of certain alphanumeric attributes, but also on the spatial location of a data object. A retrieval query against a spatial database often requires the fast execution of a geometric search operation such as a point query (find all objects that contain a given search point) or a region query (find all objects that overlap a given search region). To support such search operations, special multi-dimensional access methods are needed.

Volker Gaede and Oliver Gunther [GG98] list the following requirements that multi-dimensional access methods should meet based on the properties of spatial data and their applications. These requirements are applicable to one-dimensional access methods, too.

(1) Dynamics. As data objects are inserted and deleted from a database in any given order, access methods should continuously keep track of the changes.

(2) Secondary/tertiary storage management. Despite growing main memories, it is often impossible to hold a complete database in main memory. Access methods, therefore, need to integrate secondary and tertiary storage in a seamless manner.

(3) Broad range of supported operations. Access methods should not support just one particular type of operation (such as retrieval) at the expense of other tasks (such as deletion).

(4) Independence of the input data and insertion order. Access methods should maintain their efficiency even when the input data is highly skewed. This point is especially important for data that is distributed differently along the various dimensions. Related to this point is a good average storage utilization.

(5) Simplicity. Intricate access methods with special cases are often error-prone to implement, and thus not sufficiently robust.

(6) Scalability. Access methods should adapt well to growth in the underlying database.

(7) Time efficiency. Because spatial searches are usually CPU- and I/O-intensive, an access method should try to minimize both the CPU utilization and the number of disk accesses.

(8) Space efficiency. An index should be small in size compared to the data to be addressed, and therefore, guarantee a certain storage utilization.

1.1 Problem

A large variety of specialized access methods have been developed to solve specific problems. Among them are those based on the hierarchical approach such as B-tree [BM72], B*-tree [Knu73], K-D-B-tree [SRF87], LSD-tree [HSW89], hB-tree [LS89, LS90], R-tree [Gut84], R+-tree [SSH86, SRF87], Cell-tree [Gun88, Gun89], GBD-tree [OS90], and so on, those based on hashing techniques such as Grid file [NHS84], BANG file [Fre87], R-file [HSW90], Z-hashing [HSW88], and so on, and those based on heuristic solutions such as Space-Filling Curves [OM84]. While some of this work has had significant impact in particular domains, the approach of developing domain-specific access methods is problematic. These specialized access methods are usually hand-coded from scratch, normally requiring substantial knowledge of the underlying file system to build correctly and efficiently. As a result, the effort required to implement and maintain such data structures is high. Furthermore, there is every reason to assume this trend will continue. As new database applications need to be supported, new access methods will be developed to deal with them efficiently.

1.2 Solution

As an alternative to developing new data structures from scratch, Stonebraker [Sto86] proposed to generalize existing data structures such as B+-trees and R-trees to support user-defined data types by revising procedures of these access methods to work on new data types. Hellerstein et al. [HNP95] proposed a way to further generalize search trees at the tree functionality level by introducing a generalized search tree called GiST. The GiST provides abstraction of common tree operations, such as search, insertion and deletion. Other tree structures, such as B+-trees and R-trees, can be built as extensions of the GiST. These do not add generalization covering alternatives such as the internal structure of a B-tree's node, nor several other aspects of search tree design.

This thesis intends to generalize access methods starting with database search trees. By generalizing the parts that search-tree developers possibly need to specialize, a developer is not required to write all the components from scratch so that new search trees can be built more easily. By discovering the core, reusable components of search trees along a number of different dimensions, a generalized search tree can be built and different search trees can be developed from the generalized components significantly faster than working from scratch. This generalization can be done by taking advantage of reuse capabilities in modern programming languages such as inheritance and generic programming; for example, by deriving a new object's implementation from an existing object, it is possible to take advantage of previously written and tested code, which can substantially reduce the time to compose and debug a new object and increase its robustness.

The goal of this work parallels that for primary-memory data structure libraries such as the Standard Template Library [SL95] and Leda [Nah90] for data structures in C++. These libraries attempt to provide a taxonomy among primary-memory data structures, which is used both to classify functionality and reuse implementation. This work attempts to start a similar taxonomy but for access methods manipulating data on secondary storage. The difference in performance and capability of secondary storage from primary storage requires a separate solution. This thesis does not intend to construct new access methods, but is an engineering exercise, combining a number of well known ideas in a way that provides a new and interesting solution to this important problem.

1.3 Implementation

The generalization and resulting taxonomy are largely system independent. Yet, a further goal of this work is to demonstrate a partial implementation. The system chosen for the implementation is μDatabase. μDatabase is a toolkit for constructing memory mapped databases [BGW92]. The generalization approach should apply equally well in traditional non-memory mapped environments.

1.4 Overview

The next chapter is an overview of related work. Three categories of specialized search trees and their basic properties are discussed. Then two generalization attempts for search trees are presented.

Chapter 3 gives background in design and implementation of a new generalization of search trees. First, the generalizing techniques provided by C++ are introduced. The search-tree generalization depends on these language abstraction facilities. Second, the background of μDatabase and the μDatabase view of storage are discussed.

Chapter 4 discusses the multiple levels of generalization in detail. Then, the hierarchy of persistent tree classes and components of the generalization library are presented.

Chapter 5 gives some example search tree structures developed from the generalized tree and shows some experimental results on the measurement of the cost for generalization.

Chapter 6 is the conclusion and suggests some future work.

Chapter 2

Related Work

To understand how to generalize search tree access methods, it is necessary to understand various access methods sufficiently to find the common aspects. This chapter reviews database search trees, identifies points of commonality, and looks at prior attempts to generalize search trees.

2.1 Specialized database search trees

A large variety of search trees have been developed to solve specific problems. They can be viewed in three categories, one-dimensional search trees, point search trees and spatial search trees. The basic structure and representative search trees of each category are discussed in the next three subsections.

2.1.1 One-Dimensional Access Methods

Classical one-dimensional access methods are an important foundation for the design of multi-dimensional access methods. Hierarchical access methods, e.g., the B-tree and its variants are efficient data structures for indexing one-dimensional data.

B-tree

The B-tree [BM72] is a balanced tree corresponding to a nesting of intervals. Each node v corresponds to a disk page $D(v)$ and an interval $I(v)$. If v is an interior node, then the intervals $I(v_i)$, corresponding to the immediate descendants of v, are mutually disjoint subsets of $I(v)$. Figure 2.1 shows an example node in a B-tree with $n-1$ search values. Leaf nodes contain pointers to data items; depending on the type of the B-tree, interior nodes may do so as well.

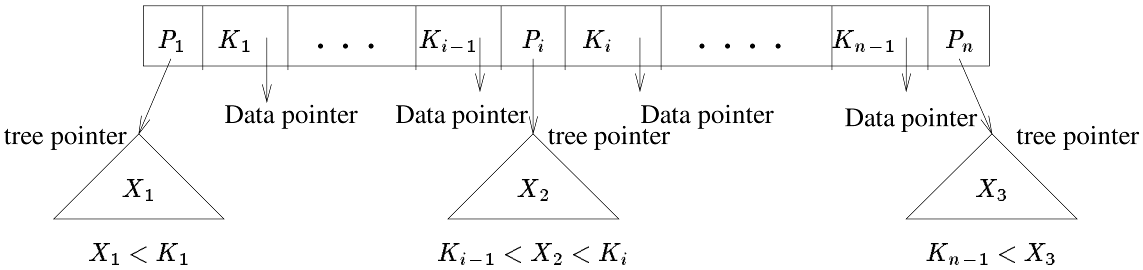


Figure 2.1: B-tree node with n-1 search values

Formally, a B-tree, when used as an access method on a key field to search for records in a data file, can be defined as follows (q is the *order* of the B-tree):

(1) In each interior node as shown in Figure 2.1 where $n \leq q$, P_i is a tree pointer — a pointer to another node in the B-tree. With each key K_i, there is a data pointer — a pointer to the record whose search key field value is equal to K_i.

(2) Within each node, keys are ordered, such that $K_1 < K_2 < \cdots < K_{n-1}$.

(3) For all search key field values X in the subtree pointed at by P_i, $K_{i-1} < X < K_i$ for $1 < i < n$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = n$.

(4) Each node has at most q tree pointers.

(5) Each node, except the root and leaf nodes, has at least $q/2$ tree pointers. So each node is at least 1/2 full. The root node has at least two tree pointers unless it is the only node in the tree.

(6) All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except they have no tree pointers.

B*-tree

Knuth [Knu73] defines a B*-tree to be a B-tree in which each node is at least 2/3 full (instead of just 1/2 full). B*-tree insertion uses a local redistribution scheme to delay splitting until two sibling nodes are full. Then the two nodes are divided into three with each 2/3 full. This scheme guarantees that storage utilization is at least 66%, while requiring only moderate adjustment of the maintenance algorithms. Increasing storage utilization can speed up the search since the height of the resulting tree is smaller.

B+-tree

For a B+-tree, proposed by D. Comer [Com79], all keys reside in the leaves. The upper levels, which are organized as a B-tree, consist only of an index, a road map to enable rapid location of the index and leaves. Figure 2.2 shows the logical separation of the index and leaves, which allows interior nodes (also called *index nodes*) and leaf nodes (also called *data nodes*) to have different formats or even different sizes. In particular, leaf nodes are usually linked together left-to-right, as shown in Figure 2.2. The linked list of leaves is referred to as the *sequence set*. Sequence set links allow easy sequential processing.

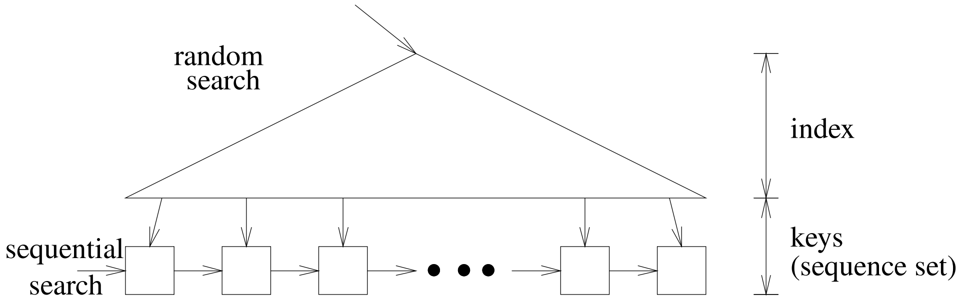


Figure 2.2: B+-tree with separate index and key parts.

Insertion and search operations in a B+-tree are processed in a way similar to insertion and search operations in a B-tree. When a leaf splits in two, instead of promoting the middle key, the algorithm promotes a copy of the key, retaining the actual key in the right or left leaf. The resulting node structure, as shown in Figure 2.3, is different from that in a B-tree. Search operations differ from those in a B-tree in that searching does not stop if a key in the index equals the query value. Instead, the nearest right or left pointer is followed, and the search proceeds all the way to a leaf.

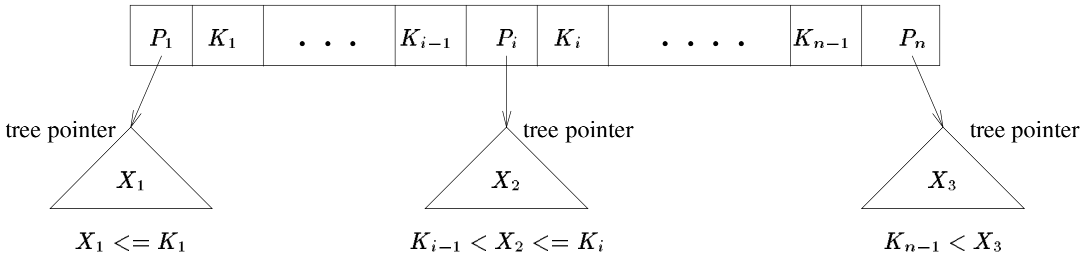


Figure 2.3: B+-tree Node

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

2.1.2 Point Access Methods

A point is defined to be an element of $(domain_1 \times domain_2 \times \cdots \times domain_k)$, and a rectilinear region is defined to be the set of all points $(x_1, x_2, \cdots, x_k)$ satisfying $min_i < x_i < max_i$, $1 < i < k$, for some collection of min_i, max_i of $domain_i$. Points can be represented most simply by storing the x_i's, and regions by storing the min_i's and max_i's.

Point access methods (PAMs) have been designed primarily to perform spatial searches on point databases. The points may have two or more dimensions, but they do not have a spatial extension. Usually, the points in the database are organized in a number of buckets, each of which corresponds to a disk page and a subspace of the universe. The subspaces (often referred to as *bucket regions* or simply *regions*, even though their dimension may be greater than two) need not be rectilinear, although they often are.

Tree-based PAMs are usually a generalization of the B+-tree to higher dimensions. A leaf node of the tree contains those points that are located in the corresponding bucket region. The index nodes of the tree are used to guide the search; each of them typically corresponds to a larger subspace of the universe that contains all bucket regions in the subtree below. A search operation is then performed by a top-down tree traversal.

Differences among PAM tree structures are mainly based on the characteristics of the regions. In most PAMs, the regions at the same tree level form a partitioning of the universe, i.e., they are mutually disjoint, with their union being the complete space. Examples of tree-based point access methods are K-D-B-trees [Rob81], LSD-trees [HSW89], hB-trees [LS89, LS90], and so on. In the following, the K-D-B-tree and the hB-tree are introduced as representatives of this category.

K-D-B-tree

The K-D-B-tree [Rob81] combines the properties of the adaptive K-D-tree [Ben75] and the B+-tree [Com79] to handle multidimensional points. Like the B+-tree, the K-D-B-tree consists of a collection of nodes. There are two types of nodes:

1. Region nodes: region nodes contain a collection of (region, child) pairs, where *child* is a pointer to a node.
2. Point nodes: point nodes contain a collection of (point, location) pairs, where *location* gives the location of a database record.

A K-D-B-tree structure has to satisfy the following properties. The algorithm for range queries depends only on these properties, and the algorithms for insertions and deletions are designed to preserve these properties.

(1) No region node contains a null pointer, and no region page is empty, so point nodes are the leaves of the tree.

(2) Leaf nodes are on the same level from the root (have the same depth).

(3) In every region node, the regions in the node are disjoint, and their union is a region.

(4) If the root node is a region node, the union of its regions is the universe.

(5) If (region, child) occurs in a region node r, and the child points to another region node, then the union of the regions in the child node is the region in r.

(6) If (region, child) occurs in a region node r, and the child points to a point node, then all the points in the point node must be in the region in node r.

The K-D-B-tree is a perfectly balanced tree that adapts well to the distribution of data. Unlike B-trees, the K-D-B-tree does not have a minimum space utilization requirement. Figure 2.4 illustrates an example 2-D-B-tree where p's are points and c's represent central points of spatial objects.

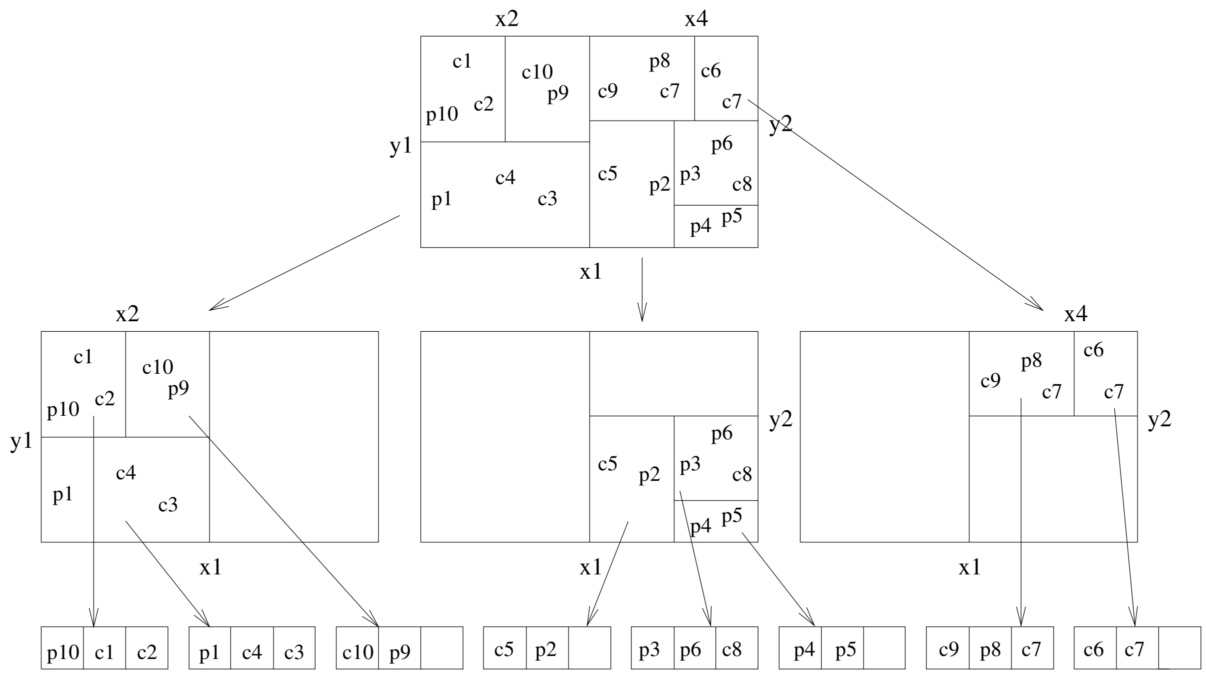


Figure 2.4: Example 2-D-B-tree

To insert a new data point, a point search is performed to locate the correct bucket. If this point node is not full, the entry is inserted. Otherwise, it is split and about half of the entries are shifted to the new point node. In order to find a good split, various heuristics are available [Rob81]. If the parent region node does not have enough space left to accommodate the new entries, a new region node is allocated and the region node is split by a hyperplane. The entries are distributed between the two nodes, depending on their position relative to the splitting hyperplane, and the split is propagated up the tree. The split of the index node may also affect regions at lower levels of the tree, which have to be split by this hyperplane as well. Because of this forced split effect, it is impossible to guarantee a minimum storage utilization.

Deletion is straightforward. After performing an exact match query, the entry is removed. If the number of entries drops below a given threshold, the point node may be merged with a sibling point node as long as the union remains a d-dimensional interval.

hB-tree

The hB-tree [LS89] is a relative of the K-D-B-tree. One of the distinguishing features of hB-trees is that the index nodes are organized as K-D-trees.

In hB-trees, the nodes represent regions from which smaller regions have been removed. These regions are called *holey bricks* and the hB-tree is also called a *holey brick* tree. Each index node represents a holey brick and refers to smaller holey bricks on a lower level of the index. The K-D-tree [Ben75] is used to organize the internal structure of index nodes of the hB-tree. The K-D-tree is a binary search tree that represents the recursive subdivision of the universe into subspaces by means of $d-1$ dimensional hyperplanes. Figure 2.5 illustrates a holey brick represented via a K-D-tree. The region is first divided into two parts, left of $x1$ and right of $x1$. The right part is subdivided into two regions, above $y1$ which is A and below $y1$ which is part of B.

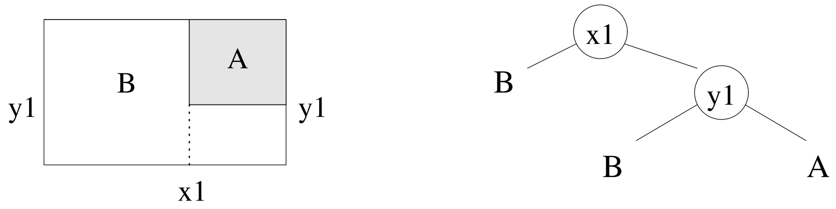


Figure 2.5: Holey brick represented via a K-D-tree

Figure 2.6 shows a hB-tree with the root node containing two pointers to its child u. The region u is the union of two rectangles: the one to the left of $x1$ and the one above $y1$. The remaining space (C, D and F) is excluded from u, which is made explicit by the entry ext in the K-D-tree representing region u.

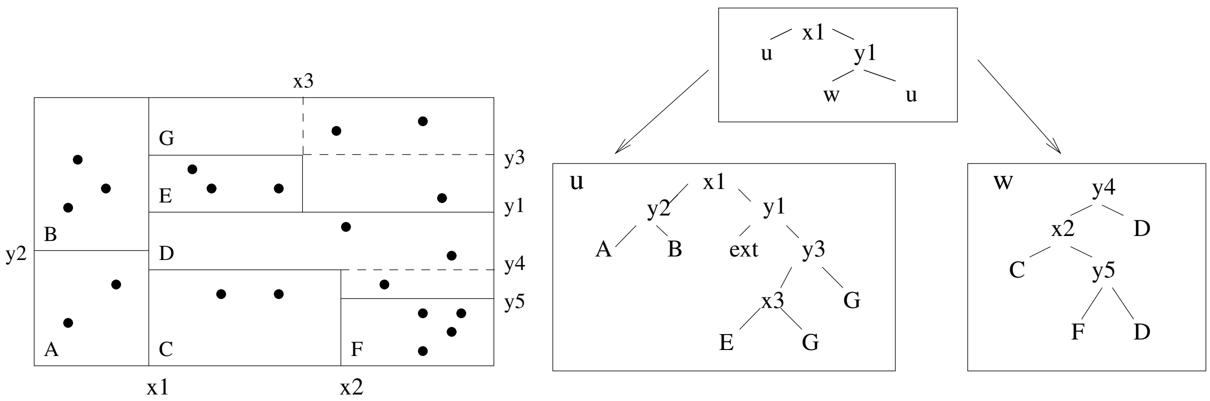


Figure 2.6: Example hB-tree

2.1.3 Spatial Access Methods

Spatial access methods are applicable to databases containing objects with a spatial extension. Typical examples include geographic databases, containing mostly polygons, or mechanical CAD data, consisting of three-dimensional polyhedra. In order to handle such extended objects, point access methods have been modified using one of the following three techniques [SK88]:

- Transformation (Object Mapping) : The basic idea of transformation schemes is to represent minimal bounding rectangles of multidimensional spatial objects by higher dimensional points.

- Overlapping Regions (Object Bounding) : The key idea of the overlapping regions technique is to allow different data buckets in an access method to correspond to mutually overlapping subspaces. A well-known example is the R-tree [Guttman84, Greene89].

- Clipping (Object Duplication) : Clipping-based schemes do not allow any overlaps among bucket regions; they have to be mutually disjoint. A typical example is the R+-tree [SSH86, SRF87], a variant of the R-tree that allows no overlap among regions corresponding to nodes at the same tree level.

R-tree

An R-tree [Gut84] is a height-balanced tree similar to a B+-tree with index records in its leaf nodes. Nodes correspond to disk pages and the structure is designed so that a spatial search requires visiting only a small number of nodes.

A spatial database consists of a collection of tuples representing spatial objects, and each tuple has a unique identifier, which can be used to retrieve it. Leaf nodes in an R-tree contain entries of the form *(I, tuple-identifier)* where *tuple-identifier* refers to a tuple in the database and I is an n-dimensional rectangle which is the bounding box of the spatial object indexed: $I = (I_0, I_1, \cdots, I_{n-1})$ where n is the number of dimensions and I_i is a closed bounded interval $[a, b]$ describing the extent of the object along dimension i. Non-leaf nodes contain entries of the form *(I, child-pointer)* where *child-pointer* is the address of a lower node in the R-tree and I covers all rectangles in the lower node's entries. Figure 2.7 shows some rectangles organized into an R-tree.

An R-tree satisfies the following properties:

(1) Every node except the root contains between min and max entries where $min \leq max/2$.

(2) For each leaf entry, key I is the smallest rectangle, also called minimum bounding box, that spatially contains the n-dimensional data object represented by the indicated tuple.

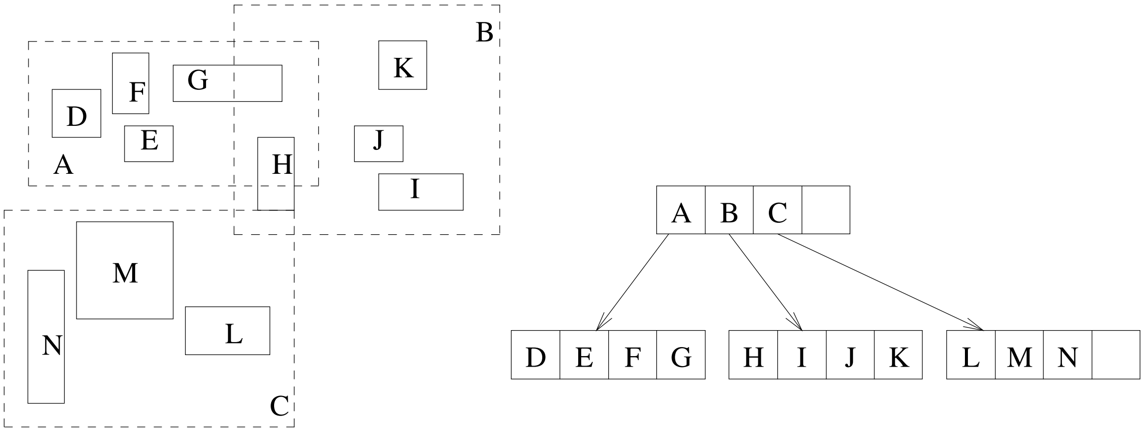


Figure 2.7: Rectangles organized into an R-tree

(3) For each non-leaf entry, key I is the smallest rectangle that spatially contains the rectangles in the child node. (For example in Figure 2.7, the actual scope of the X-dimension in the rectangle labeled B extends from the minimum value in H to the maximum value in I.)

(4) The root node has at least two children unless it is a leaf.

(5) All leaves appear on the same level.

The search algorithm descends the tree from the root in a manner similar to a B+-tree. However, more than one subtree under a visited node may need to be searched because the bounding boxes at the same tree level may overlap. Even for point queries, there may be several intervals that intersect the search point.

To insert a new object, the minimum bounding box and the object reference are inserted into the tree. In contrast to searching, only a single path from the root to the leaf is traversed. At each level, a child node is chosen so that its corresponding bounding box needs the least enlargement to enclose the data object's bounding box. If several intervals satisfy this criterion, one descendant is chosen at random. This guarantees that the object is inserted only once, i.e., the object is not dispersed over several buckets. Once the leaf level is reached, the object is ready to be inserted. If this requires an enlargement of the corresponding bucket region, adjustment is done appropriately and the change is propagated upwards. If there is not enough space left in the leaf, the leaf is split and the entries are distributed between the old and the new page. Then each of the new bounding box needs to be adjusted accordingly and the split is propagated up the tree.

Similarly for deletion, first, an exact match query is performed for the object to be deleted. If the object is found, it is deleted. If the deletion causes no underflow, a check is performed whether the bounding box can be reduced in size. If so, this adjustment is made and the change is propagated upwards. On the other hand, if the deletion causes the node capacity to drop below 50%, the node contents are copied into a temporary node and the original node is removed from the tree. All bounding boxes have to be adjusted. Afterwards all orphaned entries of the temporary node are reinserted into the tree. In his original paper, Guttman [Guttman84] discussed various policies to minimize the overlap during insertion. For node splitting, for example, Guttman suggests several algorithms, including a simpler one with linear time complexity and a more elaborate one with quadratic complexity. More sophisticated policies can be seen in the *packed R-tree* [RL85], the *sphere tree* [Oo90] and the *Hilbert R-tree* [KF94].

R+-tree

To overcome the problems associated with overlapping regions, Sellis et al. introduced a close relative of the R-tree, called an R+-tree [SRF87]. The R+-tree uses clipping, i.e., there is no overlap among index intervals on the same tree level. Figure 2.8 shows how the rectangles in Figure 2.7 are grouped into an R+-tree. Objects that intersect more than one index interval have to be stored on several different pages, such as the rectangle G in Figure 2.8. As a result of this policy, point searches in a R+-tree correspond to single-path tree traversals from the root to one of the leaves. Therefore, they tend to be faster than the corresponding R-tree operation. An important difference for node splitting is that splits may propagate not only up the tree, but also down the tree. The resulting forced split of the nodes below may lead to several complications, including a further fragmentation of the data intervals.

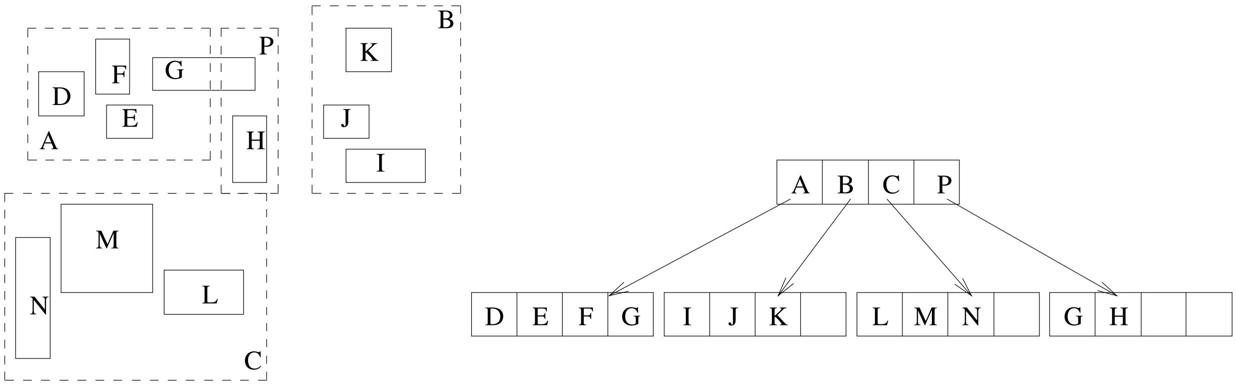


Figure 2.8: Rectangles organized into an R+-tree

R*-tree

The R*-tree is another variant of the R-tree. Based on a careful study of the R-tree behaviour under different data distributions, Beckmann et al. [BKSS90] identified several weaknesses of the original algorithms. In particular, the insertion phase is claimed to be critical for search performance. The design of the R*-tree introduces a policy called forced reinsert: if a node overflows, it does not split right away. Instead, about 30% of the maximal number of entries are removed from the node first and then reinserted into the tree. In addition, to solve the problem of choosing an appropriate insertion path, the R*-tree not only takes the area parameter into consideration, but tests the parameters' area, margin and overlap in different combinations. The R*-tree differs from the R-tree mainly in the insertion algorithms; deletion and searching are essentially unchanged.

2.2 Search trees with extensible data types

Instead of developing new access methods for new data types, for example, polygons, Stonebraker [Sto86] proposes that existing data structures such as B+-trees and R-trees can be made extensible in the data types they support.

In general, an access method is a collection of procedures that retrieve and update records. A generic abstraction for an access method could be the following:

open (file-name) : returns a descriptor for the file representing the relation

close (descriptor) : terminates access

get-first (descriptor, OPR, value) : returns the first record with key satisfying "key OPR value"

get-next (descriptor, OPR, value) : returns the next record

insert (descriptor, value) : insert a record into indicated relation

delete (descriptor, value) : delete a record from the indicated relation

search (descriptor, value) : search for the indicated record

update (descriptor, value, new-value) : replace the indicated record with the new one

The basic idea of this approach is to replace these procedures by others, which operate on a different data type and allow the access method to work for the new type. For example, consider a B+-tree and the generic query: select ... from ... where key OPR value, and OPR is one of $=, <, \leq, \geq, >$. A B+-tree includes appropriate procedures to support these operators for a data type. For example, to search for a record matching a specific key value, one need only descend the

B+-tree at each level searching for the minimum key whose value exceeds or equals the indicated key. Only calls on the operator "$\leq$" are required with a final call or calls to the operator "=". These comparison operators can be overloaded for user defined types. The procedures implementing these operators can be replaced by any collection of procedures for overloaded operators and the B+-tree "works" correctly. For example, Stonebraker shows three operators on a box data type can be added to the B+-tree. They are AE (two box areas equal), AL (one box area less than the other box area) and AG (one box area greater than the other box area).

This approach provides generalization in the data that can be indexed. For example, B+-trees can be used to index any data with a linear ordering. However, this generalization is rather simple and limited. Regardless of the type of data stored in a B+-tree, the only queries which can benefit from the tree are those containing equality and linear range predicates. Similar in an R-tree, the only queries that can use the tree are those containing equality, overlap or containment predicates. No unified view of different search trees is achieved by this approach.

2.3 GiST

Hellerstein et al. [HNP95] introduce an index structure, called a Generalized Search Tree (GiST), which is a generalized form of an R-tree [Gut84]. The GiST allows new data types to be indexed and supports an extensible set of queries. In addition, the authors claim that the GiST unifies previously disparate structures used for currently common data types. For example, both B-trees and R-trees can be implemented as extensions of the GiST.

In detail, a GiST is a balanced tree structure with tree nodes containing (p, ptr) pairs, where p is a predicate that is used as a search key, and ptr is the identifier of some tuple in the database for a leaf node and a pointer to another tree node for a non-leaf node. A GiST has the following properties:

(1) Every node contains between min and max index entries unless it is the root.

(2) For each index entry (p, ptr) in a leaf node, p is true when instantiated with the values from the indicated tuple (i.e., p holds for the tuple).

(3) For each index entry (p, ptr) in a non-leaf node, p is true when instantiated with the values of any tuple reachable from ptr.

(4) The root has at least two children unless it is a leaf.

(5) All leaves appear on the same level.

In principle, the keys of a GiST may be any arbitrary predicates that hold for each datum below the key. In practice, the keys come from a user-defined object class, which provides a particular set of methods required by the GiST. To adapt the GiST for different uses, users are required to register the following set of six methods:

- **Consistent(E,q):** given an entry $E = (p, ptr)$, and a query predicate q, returns false if $p \wedge q$ can be guaranteed unsatisfiable, and true otherwise.

- **Union(P):** given a set P of entries (p_1,ptr_1), ... (p_n,ptr_n), returns some predicate r that holds for all tuples stored below ptr_1 through ptr_n.

- **Compress(E):** given an entry $E = (p, ptr)$, returns an entry (pp, ptr) where pp is a compressed representation of p.

- **Decompress(E):** given a compressed representation $E = (pp, ptr)$, returns an entry (r, ptr) where r is a decompressed representation of pp.

- **Penalty($E1$,$E2$):** given two entries $E1 = (p_1, ptr_1)$, $E2 = (p_2, ptr_2)$, returns a domain-specific penalty for inserting $E2$ into the subtree rooted at $E1$, which is used to aid the splitting process of the insertion operation.

- **PickSplit(P):** given a set P of $M + 1$ entries (p, ptr), splits P into two sets of entries $P1$ and $P2$, each of size kM, where k is the minimum fill factor.

The tree methods of GiST provide algorithms for search, insertion and deletion operations:

- The search algorithm can be used to search any dataset with any query predicate by traversing as much of the tree as necessary to satisfy the query. This approach is the most general search technique, analogous to that of the R-tree. But it is not efficient in all situations, for example, in a linear ordered domain of a B+-tree. So GiST provides another set of methods to perform search operation when the domain to be indexed has a linear ordering, and queries are typically equality or range-containment predicates. The general search algorithm depends on the user-specified Consistent method to check if the predicate is satisfiable or not. The same Consistent method applies to both index node and leaf node. But usually the satisfiable conditions are different for the index node and leaf node, no matter if it is an exact matching search or a range/window query. So further checking is needed after leaf entries are fetched according to the algorithm. Another restriction is that only one type of query can be conducted in one program because function Consistent is used in the search instead of a pointer to a function.

- The insertion routine guarantees that the GiST remains balanced. It is very similar to the insertion routine of R-trees, which generalizes the simpler insertion routine for B+-trees. User defined key method Penalty is used for choosing a subtree to insert; method PickSplit is used for the node splitting algorithm; method Union is used for propagating changes upward to maintain the tree properties.

- The deletion algorithm attempts to keep keys as specific as possible and also maintain the balance of the tree. For underflow, it uses the B+-tree "borrow or coalesce" technique if there is a linear order, otherwise it uses the R-tree reinsertion technique.

Hellerstein et al. [HNP95] also describe implementations of key classes used to make the GiST behave like a B+-tree, an R-tree and an RD-tree, a new R-tree-like index over set-valued data. However, each GiST key class for a B+-tree is a pair of integers instead of one integer value as in a traditional B+-tree. The pair represents the interval contained below the key; particularly, a key (a, b) represents the predicate $\mathsf{Contains}([a, b], v)$ with variable v. Because more space is used for keys, the Compress method is applied before each key is placed in a node and the Decompression method is used when a key is read from a node. The Penalty method is used to choose an appropriate subtree for insertion:

```
Penalty( E = ([x1, y1), ptr1), F = ([x2, y2), ptr2) )
    if E is the leftmost pointer of its node
        return MAX( y2 - y1, 0 )
    else if E is the rightmost pointer of its node
        return MAX( x1 - x2, 0 )
    else
        return MAX( y2 - y1, 0 ) + MAX( x1 - x2, 0 )
```

Penalty is calculated for every entry in the node and the entry with minimum penalty is chosen. But keys in a B+-tree are in order, so only comparisons to find the correct range are needed. Therefore, the Penalty method should be unnecessary for a B+-tree.

While the GiST attempts to generalize all tree access methods, the approach is skewed towards spatial access methods. As a result, simpler access methods require more complex implementations than would otherwise be expected and the complexity and cost of these implementations is questionable.

2.4 Summary

A large variety of domain-specific search trees have been developed. The representative structures and their properties in each category, one-dimensional, point and spatial, are presented in detail as background material for the design of the generalization of search trees. These search trees have many points of commonality in their structure, such as one or more indices; the structure of the index and data; the kinds, number and location of keys. As well, the operations on these search trees is a strong common point. Two attempts at extending search tree technology by generalization have been discussed. Both approaches provide some ideas for this thesis work.

Chapter 3

Generalizing Techniques and μDatabase

The C++ reuse facilities used in the design and implementation of the generalized persistent search tree are discussed first in this chapter, followed by the system chosen for implementation, μDatabase, and facilities in μDatabase for managing storage.

3.1 Language Abstraction Mechanism

The inheritance and polymorphism capabilities of C++ can be applied to abstract commonalities from many different search tree file structures and provide a generalized structure that can be easily adapted to a specific tree access method.

3.1.1 Inheritance

> Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviours and embellishing these with capabilities the new classes require. Polymorphism enables us to write programs

in a general fashion to process a wide variety of existing and yet-to-be-specified related classes. [DD94, p.730]

Inheritance provides a powerful way of representing hierarchical relationships among classes, that is, expressing commonality among classes. A base class specifies common properties and derived classes inherit these properties from their base and add data members and member functions of their own. For example, the concepts of a faculty member and a staff member are related in that they are both university employees; that is, they have the concept of an employee in common. Therefore a base class Employee can be used to capture common information for all employees such as name, age and salary. Class Faculty and class Staff can include this information by inheriting from class Employee and add more specific information for a faculty member and a staff member respectively:

```
class Employee {
  public:
     int age;                  // common information for all employees
     char *name;
     float salary;
     ...
};
class Faculty : public Employee {
  public:
     char *research_area;      // additional information for faculty members
     ....
};
class Staff : public Employee {
  public:
     char *position;           // additional information for staff members
     ...
};
```

The strength of inheritance comes from the abilities to abstract commonality in the base class and to define in the derived class additions, replacements, or refinements to the features inherited from the base class. Each derived class itself can be a base class so a set of related classes forms a class hierarchy.

Inheritance contains two concepts: implementation and type sharing. Implementation inheritance allows one object to reuse existing declarations and code to build another object, whereas type inheritance means that objects of the derived type can behave just like objects of the base type. C++ supports these two concepts when deriving a class from a base class; the base class may be inherited as either public, protected, or private:

```
class derived1 : public base { ... };
class derived2 : protected base { ... };
class derived3 : private base { ... };
```

Public inheritance provides both implementation and type sharing. The "public" can be read as "is derived from" and "is a subtype of". This notion is the most common form of derivation. Protected and private inheritance are used to represent implementation details that are not reflected in the type of the derived class. The "protected" and "private" can be read as "is derived from" or "implements". Protected inheritance is useful in class hierarchies in which further derivation is the norm. Private inheritance is used when defining a class by restricting the interface to a base so that stronger guarantees can be provided [Str97]. Finally, C++ provides no direct mechanism to inherit just type without implementation, but this can be accomplished indirectly through an empty base class called an abstract class.

3.1.2 Polymorphism and Template

Polymorphism allows an algorithm to be expressed once and applied to a variety of types. C++ provides two mechanisms to accomplish this: virtual functions provide run-time polymorphism, whereas templates offer compile-time polymorphism.

Through virtual functions, objects of different classes related by inheritance respond differently to the same member function call depending on the type of the object. For example, an *earnings* function call applies generically to all employees. But the way each person's earnings are calculated depends on the type of the employee, e.g., a salaried or an hourly-paid employee. Hence the *earnings* function is declared virtual in base class Employee, and appropriate implementations of *earnings* are provided for each of the derived classes, Salaried or HourlyWorker:

```
class Employee {                          // abstract class
  public:
     virtual float earnings() = 0;        // pure virtual
};
class Salaried : public Employee {
  private:
     float weeklySalary;
  public:
     virtual float earnings() { return weeklySalary; }
     // other methods;
};
class HourlyWorker : public Employee {
  private:
     float wagePerHour;
     float hours;
  public:
     virtual float earnings() { return wagePerHour * hours; }
};
```

Templates provide direct support for generic programming, that is, programming using types as parameters. The C++ template mechanism allows a type to be a parameter in the definition of a function or a class. A template depends only on the properties that it actually uses from its parameter types and does not require different types used as arguments to be explicitly related. In particular, the argument types used for a template need not be from an inheritance hierarchy. Code implementing the templates is identical for all parameter types. As mentioned, both functions and classes can be generalized with templates.

The following template function declares a single formal parameter T as the type of two variables being compared and the type of the return value:

```
template<class T> T max( T x, T y ) {
    return ( x > y ? x : y );
}
```

It can be used to compare two variables of any type, such as integers, floating point numbers or user-defined types that define the operator “$>$”. Based on the argument types provided in calls to this function, C++ automatically generates separate object-code functions to handle each type of call appropriately. Function templates provide a compact solution like macros in C, but enable full type-checking.

The following is an example template for a stack class:

```
template<class T> class stack {
    T elems[10];
    int size;
  public:
    stack();
    void push( T e ) { elems[size] = e; size += 1; }
    T pop() { size -= 1; return elems[size]; }
};
```

On the basis of the template stack class, many stack classes can be created, such as:

```
stack<float> floatStack;                // stack of float
stack<int> intStack;                    // stack of int
stack< stack<int> > intStackStack;      // stack of stack of int
```

Besides generic programming, templates serve another need, that is policy parameterization. Consider sorting an array of elements. The element can be a structure like:

```
struct {
    char *firstname;
    char *lastname;
    int age;
} Employee;
```

Three concepts are involved: the element type, the container holding the elements (e.g., an array) and the criteria used by the sort algorithm for comparing array elements. The sorting criteria can not be hard-coded as part of the container because the container should not (in general) impose its needs on the element type. The sorting criteria can not be hard-coded as part of the element type, because there are many different ways of sorting elements. Consequently, the sorting criteria is not built into the container nor into the element type, instead, it must be supplied on a specific need basis. For example, to sort an array of structures like Employee defined above, the criteria used for a comparison depends on different needs. The employees can be sorted by lastname/firstname or by age. Neither a general element type nor a general sort algorithm should know about the conventions for sorting such an array. Therefore, any general solution requires the sorting algorithm be expressed in general terms that can be defined not just for a specific type but also for a specific use of a specific type.

For example, the template parameter class C in the following sorting function represents the possible comparison criteria:

```
template<class T, class C>
void sort( const T t[ ], int arraysize ) {
    . . .
    if( C::eq( t[i], t[j] ) )
    . . .
    if( C::lt( t[i], t[j] ) )
    . . .
}
```

Different criteria for sort() can be achieved by defining suitable C::eq() and C::lt() routines. This technique allows any sorting or other algorithms that can be described in terms of the operations supplied by the "C-operations". Different element comparison pairs are needed for the different kinds of comparison. For example, name comparison is more complex than age comparison, requiring different comparison templates for each:

```
template<class T> class Cmp1 {
  public:
    static bool eq( T a, T b ) { ... }          // compare on lastname/firstname
    static bool lt( T a, T b ) { ... }          // compare on lastname/firstname
};
```

```
template<class T> class Cmp2 {
  public:
     static bool eq( T a, T b ) { ... }          // compare on age
     static bool lt( T a, T b ) { ... }          // compare on age
};
```

The rules for comparison can be chosen by explicit specification of the template arguments:

```
void f( Employee e[], int size ) {
     sort< Employee, Cmp1<Employee> >( e, size );      // sort by name
     sort< Employee, Cmp2<Employee> >( e, size );      // sort by age
}
```

Passing the comparison operations as a template parameter has significant benefits compared to alternatives such as passing pointers to functions. Several operations can be passed as a single argument with no run-time cost [Str97]. The technique of supplying a policy through a template argument is widely used in the C++ standard library.

3.1.3 Inheritance vs. Containment

To examine the design choices involving template and inheritance, it is necessary to look at the difference between inheritance and containment. When a class D is derived from another class B, it is said that a D is a B:

```
class B { ... };
class D : public B { ... };          // D is a kind of B
```

which means inheritance is an *is-a* relationship. On the other hand, a class D that has a member of another class B is often said to *have* a B or *contain* a B:

```
class D {          // D contains a B
     B b;
};
```

which means containment is a *has-a* relationship. For given classes B and D, to choose between inheritance and containment depends on what kind of relationship is necessary between the two classes. For example, the relationship between a Car and an Engine should be containment instead of inheritance. A Car is not an Engine; it has an Engine. In general, inheritance is used to provide an *is-a* relationship and a template a *has-a* relationship.

3.2 μDatabase

A database programmer is faced with the problem of dealing with two different views of structured data: transient data and persistent data. Transient data is in primary storage and ceases to exist after the creating process terminates, whereas persistent data is stored in secondary storage and outlives the programs that create and manipulate it. Traditional programming languages provide facilities for the manipulation of transient data. If data is required to be persistent, explicit use of a file system or a database management system is needed. Data structures in primary storage are usually organized using pointers, which are used directly by the processor's instructions. However, it is generally impossible to store and retrieve data structures containing pointers to/from disk without converting at least the pointers and at worst the entire data structure into a different format. Atkinson et al. [ABC+83] claim that typical programs devote significant amounts of code (up to 30%) transferring data to and from the file system or DBMS as a result. Therefore, significant time and space is taken up by code to perform translations between the structured data in primary storage and the data stored on secondary storage, especially for complex data structures. Hence, the powerful and flexible data structuring capabilities of modern programming languages are not directly available for building data structures on secondary storage.

μDatabase is a toolkit that provides a uniform view of data between primary and secondary storage and a methodology for efficiently constructing low-level database tools, such as access methods. The uniform view of data gives the illusion that data on secondary storage is accessible in the same way as data in primary storage so that the need for complex and expensive execution-time conversions of structured data between primary and secondary storage can be eliminated. The software provided by μDatabase allows normal programming pointers to be stored directly onto secondary storage, and subsequently retrieved and manipulated by other programs without having to modify the pointers or the code that manipulates them. File structures for a database, such as a B-tree, built in μDatabase are significantly simpler to build, test, and maintain than traditional file structures. All access to the file structures is statically type-safe. In addition, multiple file structures may be simultaneously accessible by an application. Finally, μDatabase allows all the generalization facilities of C++ (templates, inheritance, overloading) to be applied directly in this project.

3.2.1 μDatabase Storage

In μDatabase, memory is divided into three major levels for storage management [Goe96] (see Figure 3.1):

address space is a set of addresses from 0 to N used to refer to bytes or words of memory. This memory is conceptually contiguous from the address-space user's perspective, although it might be implemented with non-contiguous pages. The address space is supported by the hardware and managed by the operating system.

segment is a contiguous portion of memory. There may be a one-to-one correspondence between an address space and a segment, or an address space may be subdivided into multiple segments. In μDatabase, a segment or group of segments is mapped onto a portion of the secondary storage. The segment is supported by the hardware and managed by an address-space storage manager.

heap is a contiguous portion of a segment whose internal management is independent of the storage management of other heaps in a segment, but heaps at a particular storage level do interact. The heap is *not* supported by the hardware and is managed by its containing segment storage manager.

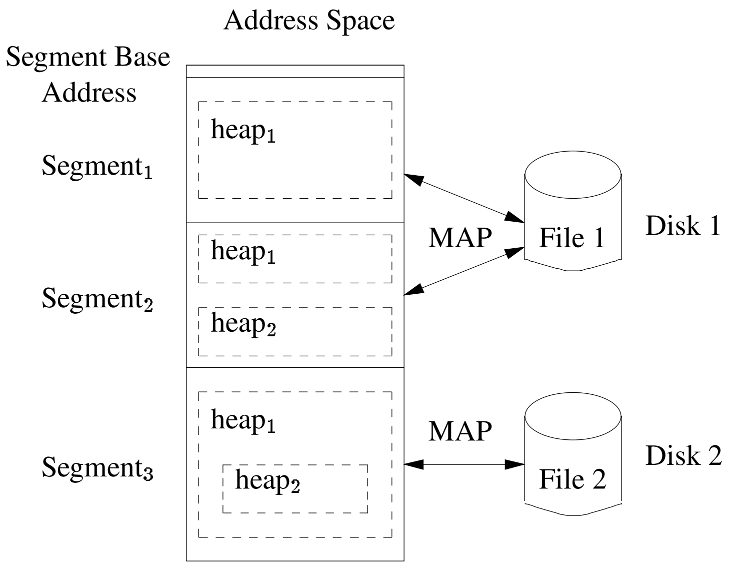


Figure 3.1: Multiple Segments of a File Structure

The difference between a segment and a heap is that only segments provide the units for mapping primary storage to secondary storage.

There are two major differences between the heap area provided by the language runtime system for dynamically allocating variables (i.e., the area where `new` allocates storage) and a heap associated with a μDatabase segment:

1. If there is only one heap, any space allocated must come from that heap. When multiple heaps can be present at the same time, a particular heap must be specified each time a memory management request occurs.

2. The language heap is a general purpose storage area. On the other hand, a segment heap is almost always dedicated to a particular data structure, e.g., a B-Tree. Therefore, there is an opportunity for optimizing the storage management scheme based on the contained data structure. In addition, many data structures require special actions to be taken when overflow and underflow occur. The storage management facility has to be able to accommodate application specific actions for these cases.

μDatabase provides tools to create, manage and destroy segments and heaps in an address space. Furthermore, flexible capabilities are provided for mapping different segments onto different disks. For example, a file structure can be partitioned into a list of segments mapped into a single UNIX file or separate UNIX files which may reside on different disks as depicted in Figure 3.1. The drawback of this scheme is that the size of each segment is restricted. In Section 4.1.7, a new scheme is introduced so that a file structure composed of multiple segments can still allow each segment to grow separately.

It is also possible for an application in μDatabase to have multiple file structures accessible simultaneously because each file structure is mapped into its own private address space. Figure 3.2 [BGW92] shows the memory organization of an application using three file structures simultaneously. The addresses in each address-space's segments are reused so only one segment can be active at a time. The shared memory is used to communicate data from one address-space's segment to another, but addresses are not shared, since they are specific to each segment.

Finally, memory manager classes are used to manage memory allocation and deallocation for a segment and its heaps. Memory manager objects instantiated from these classes are self-contained units capable of managing a contiguous piece of storage of arbitrary size, starting at an arbitrary

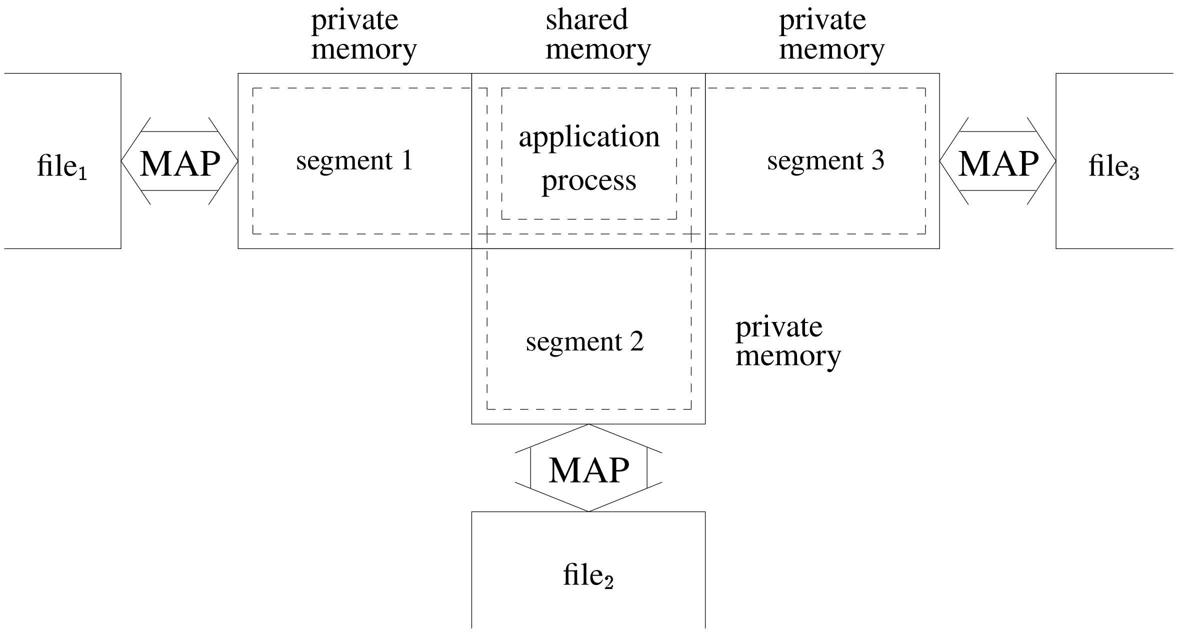


Figure 3.2: Accessing Multiple File Structures

address. If a segment is managed by a given memory manager object, invoking member routines within that object implicitly performs the desired operation on that segment/heap. The following storage management schemes are provided in μDatabase.

uniform has fixed allocation size. The size is specified during the creation of the memory manager object and cannot be changed afterwards. Uniform memory management is often used to divide a segment into fixed sized heaps or a heap into fixed sized nodes (e.g., B-Tree fixed-sized nodes).

variable has variable allocation size. The size is specified on a per allocation basis but once allocated, cannot be changed. Variable memory management is a general purpose scheme very similar to C's `malloc` and `free` routines.

dynamic has variable allocation size. The size is specified on a per allocation basis and can be expanded and contracted any time as long as the area remains allocated. Because of this property, the location of allocated blocks are not guaranteed to be fixed. Therefore, an allocation returns an *object descriptor* instead of an absolute address. An allocated block does not have an absolute address and must be accessed indirectly through its descriptor. Because of this indirection, it is possible to perform compaction on the managed space (i.e., garbage collection). Therefore, fragmentation can be dealt with in an application independent manner.

3.3 Summary

This chapter presents the main forms of reuse available in C++: inheritance and templates. Inheritance defines a class hierarchy, establishing implementation and/or type sharing. Templates define a powerful macro-like facility for constructing new types based on other types.

As well, the implementation vehicle, μDatabase, is introduced. μDatabase unifies primary and secondary storage using memory mapping, allowing normal primary memory capabilities to be used on secondary storage; in particular, the reuse capabilities of C++. μDatabase provides complex storage management facilities to organize access-method data and distribute the data appropriately onto secondary storage.

Chapter 4

Generalizing Persistent Search Trees

This chapter discusses the design methodology to achieve generalization at multiple levels and presents the structure and components of the generalized persistent search tree in detail.

4.1 Generalization of Search Trees

The language abstraction mechanisms discussed in Section 3.1 and the storage management mechanisms discussed in Section 3.2.1 are used to develop a generalized search tree that provides the basis for common tree access methods used in database systems. The commonalities of different tree file structures are explored to achieve generalization at multiple levels, including search tree functionality, behaviour, structure, index and storage management.

4.1.1 Search Trees and Search keys

The generalization of database search trees is based on the general notion of search keys and the essential nature of tree structures as observed in the GiST paper [HNP95].

A database search tree is a balanced tree with high fanout. Figure 4.1 shows the basic structure of such a tree. There are two major components: Index and Leaf. Both Index and Leaf are further composed of tree nodes, which usually have a fixed size (usually the page size). Within each tree node is a series of entries and empty space if the node is not full. The interior nodes, also called index nodes, are used as a directory to guide the search down the tree. The leaf nodes, also called data nodes, contain data entries or pointers to the actual data. The leaves may be further connected via a linked list to allow for partial or complete scanning. Figure 4.2 shows the typical structure of leaf nodes.

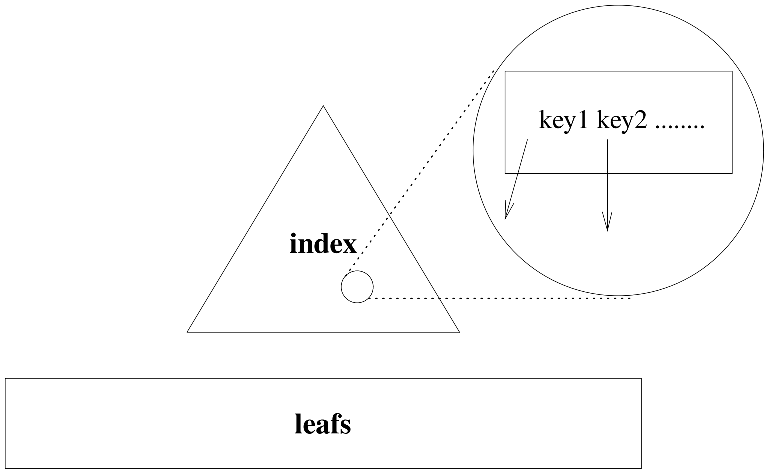


Figure 4.1: Sketch of a Database Search Tree

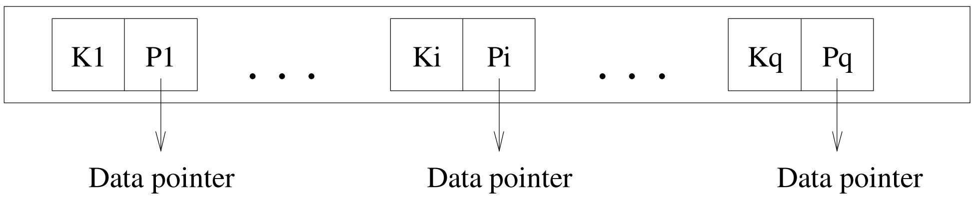


Figure 4.2: Typical Structure of Leaf Node

From the review of tree access methods, it is possible to deduce that tree nodes of most database search trees, from one-dimensional B+-tree to multi-dimensional point and spatial search trees, satisfy the following common properties:

(1) Each index node contains between min and max entries unless it is the root.

(2) Each entry in a leaf node contains a data object or a pointer to a data object and a key uniquely identifying the object.

(3) Each entry in an index node contains a key and a pointer to a child node.

(4) The root has at least two children unless it is a leaf.

(5) All leaves appear on the same level.

Different search trees are used to solve specific problems in different domains, and therefore, have different structures for the keys in properties (2) and (3). Leaf keys are used to uniquely identify the data objects in the database. Examples of key structures include integer values for data in a B-tree, coordinate values for points in a K-D-B-tree and rectangles for regions in an R-tree. Index key structures may be different from the leaf key depending on the kind of tree. For example, in a K-D-B-tree, leaf keys represent points and index keys represent regions. Behind all these different structures, there is a common notion for search keys. Basically, a tree structure is used to classify information by breaking a whole into parts, and repeatedly breaking the parts into subparts. In this sense, an index key logically matches all data stored in its subparts.

For example, keys in the B+-tree logically delineate a range in which the data below a pointer is contained. Figure 4.3 shows a simple B+-tree with integer key values. The key values in the tree are from 2 to 16. This range is divided into two parts by the key in the root, 10. Every key value in the left subtree, including its left children and subtrees rooted at them, is less than or equal to 10. Key values in the right subtree are greater than 10. The range in the left subtree, 2 to 10, is then subdivided by keys 3 and 6. The same happens in the right subtree. Key values in the left subtree of 3 are less than or equal to 3, while values in the left subtree of 6 are less than or equal to 6.

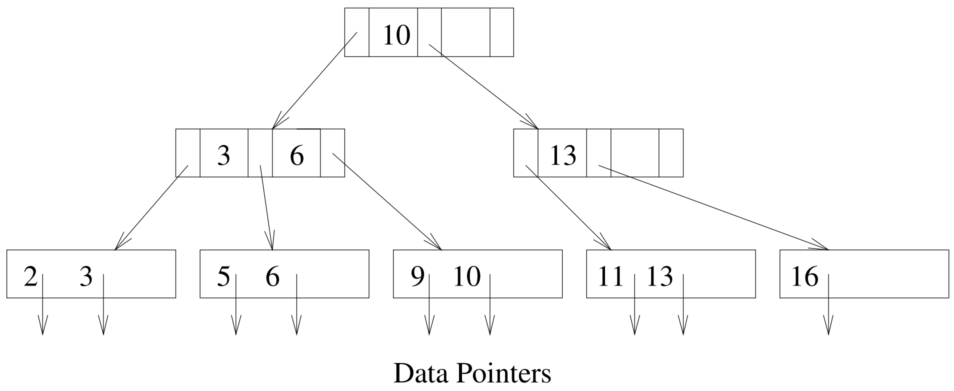


Figure 4.3: Simple B+-tree

Notice that index nodes of B+-trees have structures $< P_1, K_1, P_2, K_2, ...K_{n-1}, P_n >$. In order to generalize the B+-tree with other tree structures, one more key, K_n is added at the end to pair a key and a pointer in each index node without affecting the original structure and property of a tree node. In this case, each node now has the structure $< P_1, K_1, P_2, K_2, ...P_n, K_n >$. If the leaf keys are in ascending order from left to right, then each key K_i is greater than or equal to every key value in the subtree pointed by P_i. That is, every key is the biggest value of its subtree. If the leaf keys are in descending order from left to right, then each key K_i is the smallest value in its subtree. The simple B+-tree in Figure 4.3 is redrawn in Figure 4.4 with the modified index structure.

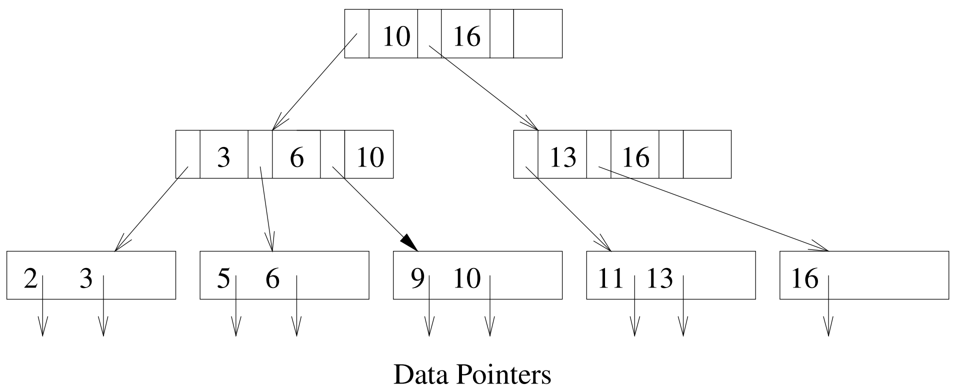


Figure 4.4: Pairing key and pointer for B+-tree

While the ranges represented by B-tree keys are disjoint, in the R-tree, the regions represented by rectangle keys may overlap. Figure 4.5 shows a region containing some rectangles and points that are leaf keys of the corresponding R-tree. Index keys are all rectangles. The region is divided into two parts represented by keys R1 and R2. R1 is subdivided into R3 and R4, and so on. For index key R1, its children R3 and R4 are contained in R1. Children of R3 and R4, rectangle A, B, G, F and point P, are contained in R3 and R4, respectively. R-tree keys delineate the bounding box in which the data below a pointer is contained.

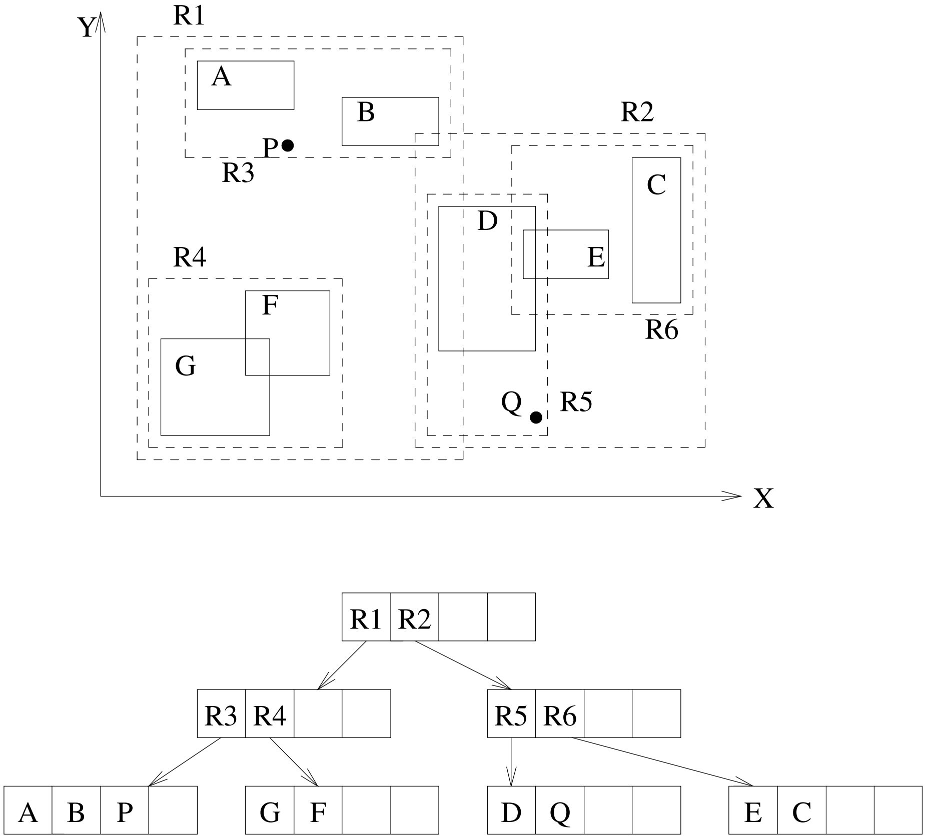


Figure 4.5: Simple R-tree

To satisfy the search tree properties, the restriction placed on a key is that it must logically match each datum stored below it. In B-trees, if the leaf keys are in ascending order from left to right, the matching condition is greater than (or equal to). Key values are greater than (or equal to) any value in its subtree. In R-trees, the matching condition is to cover. Starting from the root, every rectangle covers all rectangles or points in its child.

By generalizing the notion of a search key, it is possible to capture the essential nature of a database search tree: it is a hierarchy of categorizations, in which each categorization holds for all data stored under it in the hierarchy. Based on this idea, it is possible to generalize basic tree operations. Furthermore, in the same search tree, keys in index nodes can be different from those in leaf nodes. Therefore, to generalize the search key further, two categories of keys, *IndexKey* and *LeafKey*, are used.

4.1.2 Tree Functionality

Different kinds of search trees are related in that they share the common concept of a search tree, that is, they have certain common operations such as insertion, deletion and search. Other possible functions include returning the size of the tree, creating an empty tree or clearing the contents of the tree, and so on. This commonality at the functional level is captured in a generalized search tree class, GTree:

```
class GTree {
  public:
     int Insert( LeafKey &, Data * );
     int Delete( LeafKey & );
     bool Search( LeafKey &, Data * );
     int Size();
     bool Empty();
     void Clear();
     ...
};
```

Specific search trees, such as B+-trees and R-trees, inherit publicly from GTree so that more operations or operations with a revised algorithm can replace existing ones.

```
class BTree : public GTree {
  public:
     // add or replace GTree operations
};

class RTree : public GTree {
  public:
     // add or replace GTree operations
};
```

A B+-tree may have a different deletion algorithm and an R-tree may use a different node splitting algorithm. It is now possible to write a general function like print(GTree t) that can print the contents of either a B-tree or an R-tree. A specific search tree may inherit from specific ones, forming an inheritance hierarchy, for example:

```
class R*Tree : public RTree {
  public:
     // add or replace RTree operations
};
```

What algorithms should be used for the operations in the GTree? They should be general enough to apply to most search trees and should be independent of any data types and any implementations of inner structures. But they cannot be too general. The GTree class is intended to provide common functionalities of search trees, not just interfaces of these operations (i.e., an implementation class rather than an abstract class). In addition, the search tree properties must remain unchanged after the algorithms are applied.

The following algorithms are designed to capture the common parts in search, insertion and deletion operations and stub routines are used within them so that the specific parts related to a particular tree structure can be user specified.

Search

Recursively descend all possible paths in the tree whose keys match with the search key.

S1. [Search subtrees] if the root is a leaf, go to step S2, else check possible subtrees that match the search condition

S2. [Search leaf] check the entries in the leaf for a match with the search key

This search algorithm basically can be used to search any search tree with any query predicate. The search condition can be an exact match (equality), range query, window query or any possible query predicate corresponding to the key type. To achieve this, the search condition is provided as a stub routine in the algorithm, called Consistent. This routine is provided by the user depending on the individual search tree implementation.

Insertion

An entry for a new key, with or without a new data object, is added to the leaf level. A leaf that overflows is split, and splits propagate up the tree.

I1. [Find leaf L for the new record] if the root is a leaf, root is L, else choose a subtree where the record should go until a leaf is reached

I2. [Add record to the leaf L] if the leaf has room for another entry, add the record to the leaf, else split the old records in L and the new record into both L and a new leaf LL

I3. [Propagate changes upward] adjust the tree according to new L to preserve search tree properties, passing LL if a split is performed

I4. [Grow tree taller] if split propagation caused the root to split, create a new root

Choosing a subtree in step I1 and splitting a tree node in step I2 are application dependent. In steps I3 and I4, where changes propagate upward, to calculate the new parent key according to its new subtree is also dependent on the specific application.

Deletion

Remove an entry from its leaf node. If this causes underflow, adjust tree accordingly by updating key in parent nodes to preserve search tree properties.

D1. [Find node containing entry] invoke search to locate the leaf node L containing the entry, stop if the entry is not found

D2. [Delete entry] remove the entry from leaf L

D3. [Propagate changes] adjust the tree according to new L to preserve search tree properties

D4. [Shorten tree] If the root node has only one child after the tree has been adjusted, make the child the new root

The above algorithms are used for tree operations in GTree. A user specifies the necessary stub routines for use within these GTree routines, which depend on the individual search tree and the specific implementation. Collectively, these stub routines are called tree behaviour routines and are discussed in the next section.

4.1.3 Structure

The basic structure of a database search tree is a hierarchy of tree nodes, from the root node down to the leaf nodes. Each node can be divided into a fixed or variable sized block with each holding a (key,pointer) pair or data record. These entries can also be linked as a list inside a node. Other internal node structures are possible, for example, the entries in an index node of an hB-tree are organized as a K-D-tree. The generalized search tree has to be independent of the implementation of tree nodes to accommodate all situations. The solution is to generalize tree nodes as containers and generalize entries inside the containers as another level of containers, which are the nodes of the previous containers. Therefore, the structure of a search tree is abstracted as a hierarchy of nested containers.

Generally, a container is an object that holds other objects. Examples are lists, vectors, and associative arrays. Each container in the generalized search tree is an area corresponding to one tree node. It contains a storage manager and a number of nodes organized as an array, linked as a list, or arranged as other types of data structure. Figure 4.6 shows a container with an array of nodes and a container with a linked list of nodes. Nodes can be inserted into and removed from the container. The use of containers hides the implementation details of the inner container structure from the tree classes, for example, what type of fields each node has and how nodes are organized.

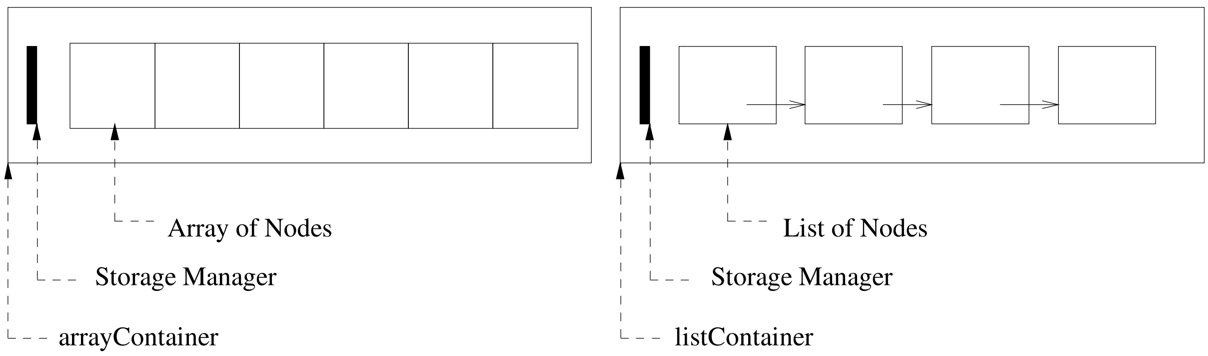


Figure 4.6: Structures of an array and a list container

A search tree holds containers, and therefore, the container type is a template parameter for tree class GTree. A container holds nodes, and therefore, the node type is a template parameter for the container class. A node becomes the container for the next level of nesting and may hold a pointer, a key or a data object. Finally, the key type and the data type are template arguments of the node class. For example:

```
template<class Container> class GTree { ... };
template<class Node> class Container { ... };
template<class key> class IndexNode { ... };
template<class key, class Data> class LeafNode { ... };
```

Container design must meet two criteria: to provide the maximum freedom in the design of an individual container, while at the same time requiring containers to present a common interface. To provide a common interface for all containers, an abstract class (type inheritance only) is provided, which specifies the common operations on containers:

```
template<class Node> class Container {
  protected:
     int count;                    // number of nodes in the container
     int level;                    // level of the container in the tree
  public:
     Container() :  count(0), level(-1) {}
     virtual Node &operator[ ]( int i ) = 0;
     virtual Node *&first() = 0;   // first node in the container
     virtual Node *&last() = 0;    // last node in the container
     virtual void AddNode( Node n ) = 0;
     virtual void DeleteNode( Node n ) = 0;
     virtual void printContainer();
};
```

The two data fields are common to all containers: count represents the current number of nodes in the container; level is the level in the tree for the container, with level = 0 for leaves and increasing by 1 each level upward in the tree hierarchy. The virtual functions provide an interface for operations like subscripting, returning the first or the last node of the container, adding a new node to the container, deleting a node from the container and printing the contents of the container (usually for debugging).

A specific container implementation, such as an array container, publicly inherits from the abstract container class:

```
template<class Node> class ArrayContainer : public Container<Node> {
    // more data fields
  public:
    // define abstract operations
};
```

More data fields can be defined in the specific container class. Containers with different implementation have different data fields. Containers with the same implementation may have slightly different structures for different search trees. For example, the leaf containers of a B+-tree are connected together for easy sequential searching while this is not the case for an R-tree.

4.1.4 Tree Behavior

GTree provides the common parts of the tree operation algorithms and requires a user to give specific tree behaviour routines to complete the operations of individual search trees. The following tree behaviour routines are defined and are derived from the GiST [HNP95] work, but have been modified:

- **Consistent** : A search key matches a certain condition with a key inside a tree node. This routine is used by the search and deletion algorithms.
- **Union** : A parent key is updated when changes are made in its subtree. This routine is used for adjusting the tree in insertion and deletion algorithms to maintain the tree property.
- **Choose-Subtree** : A subtree is chosen to insert a new record. This routine is used by the insertion algorithm.
- **Split** : A tree node is split into two nodes when some criterion is reached. This routine is used by the insertion algorithm.

These routines specify particular behaviour and implementation of a search tree.

How are these routines accessed by GTree? The first option is that GTree contains four methods as virtual functions:

```
class GTree {
  public:
     ...
     virtual bool Consistent(...) = 0;
     ...
     virtual void Split(...) = 0;
};
```

A specific search tree inheriting from GTree, for example, R-Tree, gives the implementation of the four methods:

```
class RTree : public GTree {
  public:
     ...
     bool Consistent(...) { ... }
     ...
     void Split(...) { ... }
};
```

But these four routines are only part of the implementation of a tree operation. Tree behaviour routines cannot be called directly, as can tree operations, only indirectly from within tree operations. A solution is to include the four virtual functions as protected methods of GTree and the implementations are protected methods of the specific tree. But more importantly, the actual implementations of the four methods depend on the implementation of the containers. For one specific search tree, such as a B+-tree, the four methods associated with array containers are different from those associated with list containers. Furthermore, different implementations may be chosen for index containers and leaf containers of the same tree. For example, in a B+-tree, it is good to choose an array implementation for index containers holding fixed-size key, pointer and a list implementation for leaf containers holding variable-size data objects. It is impossible to hard-code the four methods in a specific tree when the type of containers are decided through template parameters. Furthermore, the four methods are not part of a container. One type of container can apply to many different trees but the four methods are different for different trees.

The second choice is to use template functions for the four methods. For example, different versions of Union are defined as:

```
template<class Key> Key Union( ... ) { ... }
```

Inside template class GTree, the correct version of Union is chosen according to the correct Key type by the compiler's unification algorithm. However, the different implementations of the same method cannot be included in the same file. To generalize search trees, it must be possible to build a tree with different structures and implementations for index and leaf parts.

The third approach is to bind the four methods in a class called GIST and let GTree inherit from GIST so that GTree can use methods of its superclass:

```
class GTree : public GIST {
    ...
};
```

But the inheritance relationship between GIST and GTree is inappropriate. GIST is not a GTree; GTree contains a GIST as an implementation policy. One way of seeing this is to think about the GTree having different implementations, such as the tree node splitting algorithm. This situation is like a car that can have different engines. Therefore, containment is appropriate rather than inheritance.

The approach used is to first make class GIST a template parameter of class GTree to specify the implementation policy of the specific tree so that the four routines can be passed as a single argument with no run-time cost, e.g.:

```
template<class GIST> class GTree {
    ...
};
```

and second, to make the different operations static members of the class, for example, the method Split is called using:

```
GIST::Split( ... );
```

in the Insert method. Static methods are independent of the class's object, and therefore, the class acts like a module in this case packaging the GIST routines under a unique name while still allowing the compiler's unification algorithm to select the appropriate routine. For example, given a BTreeGIST class with four static methods:

```
class BTreeGIST {
  public:
     static bool Consistent ( ... );
     static Key Union ( ... );
     static Node ChooseSubtree ( ... );
     static void Split ( ... );
};
```

then a GTree is adapted to a B-tree by registering behaviour methods:

```
class BTree : public GTree<BTreeGIST>
```

In this way, the general tree functionality provided by GTree is specialized for each specific search tree.

4.1.5 Generator

A generator (or iterator) is an abstraction of the notion of a pointer to an element of a sequence, such as a vector, an array, a linked list or a tree [Str97]. Generators provide access to the elements of these data structures without having to use or access the particular data structure's implementation. Iterating through these data structures by a generator is a very flexible and powerful technique. Depending on the data structure, there may be multiple generators that iterate over the data structure in different ways. For example, a doubly linked list may have two generators with one iterating over the list in the forward direction and the other one iterating in the reverse direction. A generator can be used to traverse the entire data structure or just part of it, for example, the first 10 nodes of a tree or until a predicate is satisfied. Moreover, multiple generators can be used at the same time to iterate over different parts of the same structure. A generator can also be used to perform any action during iteration and different actions can be performed based on the data in the current node, especially when the condition for deciding on which action to perform depends on state information local to the traversal context.

Generators can be used to perform general queries on search tree structures, while providing a clean interface for these queries. For example, generators providing range queries on data in an ordered domain, such as a B+-tree, can include the following (where `tr` represents the tree file structure):

- retrieve all records (sequential scan)

  `RangeGen1 gen(tr);`

- sequential scan from `Key1` to `Key2`

  `RangeGen2 gen(tr,Key1,Key2);`

- sequential scan from `Key` for `Cnt` amount

  `RangeGen3 gen(tr,Key,Cnt);`

- sequential scan for `Cnt1` amount before `Key` and `Cnt2` amount after

  `RangeGen4 gen(tr,Key,Cnt1,Cnt2);`

Examples of generators for window queries, with a similar interface as that for range queries, include the following:

- Intersection : given an object O, find all objects with at least one point in common with O

  `IntersectionGen gen(tr, O);`

- Enclosure : given an object O, find all objects enclosing O

  `EnclosureGen gen(tr, O);`

- Containment : given an object O, find all objects enclosed by O

  `ContainmentGen gen(tr, O);`

- Adjacency : given an object O, find all objects adjacent to O

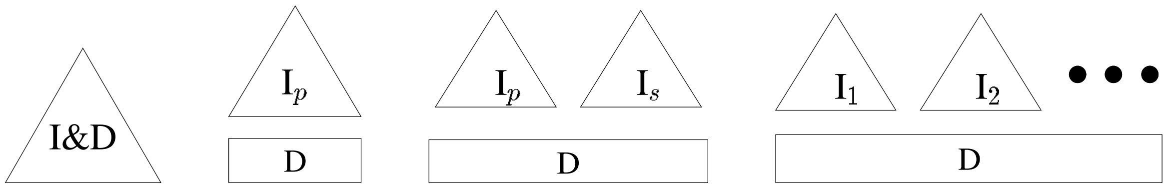
  `AdjacencyGen gen(tr, O);`

A file structure designer should be able to easily provide the above generators. These generators iterate over tree structures in a similar way but perform different query actions. Therefore, a generalized generator class, GTreeGen, is designed to provide a common interface and a complete iteration through all possible branches of the tree structure. All specific generators inherit from GTreeGen providing specific query criteria and possibly overriding member routines of GTreeGen. For example, all possible branches of the tree need to be checked in a window query, so the specific window query generators only need to inherit from GTreeGen and pass in different query routines. A range query generator may need to override member routines of GTreeGen to take advantage of sequential scan instead of iterating through all branches of the tree. The implementation details of GTreeGen and specific tree generators are discussed in later sections.

4.1.6 Index

The number of indices, as well as the different organizations between data and the indices, affects the interface and structure of the GTree and results in different GTree types.

Data records may be stored in leaves of the index tree structure, as shown in Figure 4.7(a). More commonly, index and data records are stored in separate areas or files, as shown in Figure 4.7(b), with one index tree indexed on the primary key of data records in the database. Besides the primary index, there may be an index on a non-primary key attribute, called a secondary index, as shown in Figure 4.7(c). In general, there may be multiple indices, as shown in Figure 4.7(d).



(a) (b) (c) (d)

Figure 4.7: Different Indices

To achieve this level of generalization, an address space of a file structure must be able to be divided into several segments with each segment accommodating one index tree or data part. Furthermore, these segments can be different sizes and the file structure user can specify the initial size of each segment. A difficulty occurs when one segment becomes full; expansion should be possible so that more storage can be allocated to that segment. Often the last segment is used for data because it can grow downwards in the address space easily, whereas interior segments cannot expand past their initial sizes. As mentioned in Section 3.2.1, the initial fixed partition of an address space into multiple segments is not enough and a new memory management scheme is needed to achieve this goal. This leads to a nested storage management scheme.

4.1.7 Nested Storage Management

One of the most complex parts of any data structure is efficient storage management. Usually a file structure designer spends significant time in organizing data in primary memory and on secondary storage. For memory mapped file structures, organizing data in primary memory indirectly organizes the data on secondary storage.

In order to manage storage for multiple indices, a variable storage manager (VSM) is used to allocate and free each segment of the address space. The total size this VSM can manage is the address space size. In each index tree segment, a storage manager, SM, is used to manage memory for containers. For example, a uniform storage manager (USM) allocates fixed sized blocks of the container size when a new container is needed and frees the storage when a container is deleted. The total size this SM can manage is the segment size. Inside each container, a uniform or variable storage manager is used to manage memory for nodes, depending on whether the nodes are fixed-sized or variable-sized. The total size this SM can manage is the container size. In the data segment, a USM or VSM is also used to allocate and free storage for the data objects. Figure 4.8 shows a picture of the nested storage managers for an address space with one primary index segment and a data segment.

Storage managers are linked to support expansion. An inner storage manager relies on its containing storage manager to expand. Associated with each storage manager, there is a generalized expansion object which supports expansion. In each container, if the storage manager cannot allocate more space for a new node, the index segment storage manager is used to allocate storage for a new container to hold more nodes. If there is no space for a new container, the address space storage manager allocates more storage for this index structure if there is free memory in the address

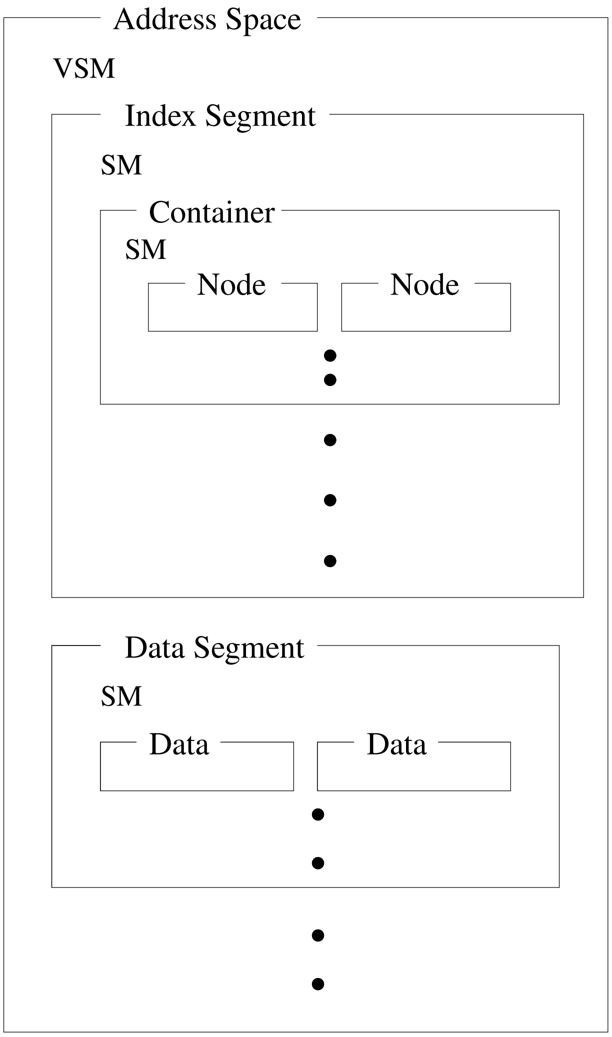


Figure 4.8: Nested Storage Managers

space.

4.2 Hierarchy of Persistent Search Tree Classes

Currently, there are three levels in defining a persistent search tree class: persistent, generic and specialized. A class PTree is used to provide storage management for different segments in an address space. The generic tree class, GTree, inherits the storage management implementation from the persistent tree class, PTree, and provides common tree operations:

```
class GTree : private PTree {
    ...
};
```

*S*Tree is used to represent any specialized tree, such as a B+-tree or an R-tree. The specialized tree class, *S*Tree, inherits both type and implementation from GTree:

```
class STree : public GTree {
    ...
};
```

In parallel with the inheritance hierarchy of tree classes, there is an inheritance hierarchy for three additional data structures, administration, access and wrapper classes, which are needed for constructing a persistent file structure in μDatabase and discussed in detail in the following sections:

```
class GTreeAdmin : private PTreeAdmin {
    ...
};
class STreeAdmin : public GTreeAdmin {
    ...
};
class GTreeAccess : private PTreeAccess {
    ...
};
class STreeAccess : public GTreeAccess {
    ...
};
class GTreeWrapper : private PTreeWrapper {
    ...
};
class STreeWrapper : public GTreeWrapper {
    ...
};
```

4.2.1 PTree

The PTree class and PTreeAdmin, PTreeAccess, PTreeWrapper classes provide a storage manager for all segments and a cover for μDatabase implementation details.

```
class PTree {
  public:
     PTree();
};
```

In the prototype implementation, the PTree is empty and only acts to name the address space containing the segments.

PTreeAdmin

The PTreeAdmin is the first data structure allocated at the start of the address space to administer the address space storage. PTreeAdmin contains the storage manager for the segments in the address space. Currently, the storage manager is restricted to be a variable storage manager.

```
class PTreeAdmin : private RepAdmin {
     friend class PTreeExpType;
  public:
     PTreeExpType exp;
     uVariable vsm;
     PTreeAdmin( int FileSize );
}; // PTreeAdmin
```

The constructor parameter is the initial file structure size, FileSize. The address space initializes a pointer to itself at the beginning of the persistent area. This pointer can be accessed from subclasses of RepAdmin[1] through the protected variable rep and is the reason for the convention requiring the administrative class to inherit from RepAdmin and for the administrative object to be stored at the beginning of the persistent area.

[1] All objects prefixed with Rep are part of the μDatabase implementation details. To simplify the explanations in this thesis, the details of these objects are omitted.

```
class PTreeAccess {
    friend class PTreeWrapper;

    PTreeAccess( const PTreeAccess & );              // prevent copying
    PTreeAccess &operator=( const PTreeAccess );
  protected:
    RepAccess<Rep> repacc;                           // access class for representative
  public:
    PTreeAccess( char *name ) : repacc( name ) {
    } // PTreeAccess::PTreeAccess
};
```

Figure 4.9: PTreeAccess definition

```
class PTreeWrapper {
    RepWrapper wrapper;                          // migrate to file segment

    PTreeWrapper( const PTreeWrapper & );    // prevent copying
    PTreeWrapper &operator=( const PTreeWrapper );
  public:
    PTreeWrapper( const PTreeAccess &tr ) : wrapper( tr.repacc ) {
    } // PTreeWrapper::PTreeWrapper
};
```

Figure 4.10: PTreeWrapper definition

PTreeAccess

An access class defines the duration for which a file-structure address space is accessible. The constructor parameter of PTreeAccess in Figure 4.9 is the UNIX file name. A file structure mapping is established by creating a RepAccess object.

PTreeWrapper

A pointer in the file-structure address space can not be used directly in an application program because the persistent area is not directly accessible in the application. A wrapper class is used to make the file-structure's address space accessible. The PTreeWrapper definition in Figure 4.10 is a cover for declaring a RepWrapper for the specified persistent area.

4.2.2 GTree

In the tree class hierarchy, GTree is the core component, which is a template class with the following parameters:

```
template<class IndexKey,       // key type for index containers
      class IndexNode,         // node type for index containers
      class IndexContainer,    // index container type
      class Indexgist,         // GIST methods for index containers
      class LeafKey,           // key type for leaf containers
      class LeafNode,          // node type for leaf containers
      class LeafContainer,     // leaf container type
      class Leafgist,          // GIST methods for leaf containers
      class Data,              // data type
      class SM                 // storage manager type
      > class GTree { ... };
```

Details of the template parameters are presented in Section 4.3.

As discussed, GTree performs the basic operations of a search tree: insertion, deletion and search using the methods provided through the GIST. The constructor of GTree takes one parameter, the container size, which is specified by the file structure designer. Figure 4.11 is the definition of a GTree class where GTreeType and GTreeArgs are macro-identifiers replaced in the program with the corresponding text defined in the #define statements.

GTreeAdmin

Figure 4.12 shows an example definition of GTreeAdmin, which inherits from PTreeAdmin privately in order to link with the storage manager for the address space for possible expansion. The administrative class contains a pointer, Root, to the root node of the persistent search tree. In this example, the persistent area (address space) is divided into two segments, one for the primary index tree and the other for data objects. Therefore, two expansion objects and two storage managers are present. The storage manager for the index tree, which deals with fixed-size containers, is built

```
#define GTreeType
        class IndexKey, class IndexNode,
        class IndexContainer, class Indexgist,
        class LeafNode, class LeafKey,
        class LeafContainer, class Leafgist,
        class Data, class SM
#define GTreeArgs
        IndexNode, IndexKey, IndexContainer, Indexgist,
        LeafNode, LeafKey, LeafContainer, Leafgist,
        Data, SM

template<GTreeType> class GTree : private PTree {
    GTree( const GTree & );                    // prevent copying
    GTree &operator=( const GTree );
    int sizeOfContainer;

    int Insert( LeafKey &r, Data *data, int size );
    int Delete( LeafKey &r, Data *data );
    LeafNode *Search( LeafKey &r );
  protected:
    GTreeAdmin<GTreeArgs> *SegZero;
  public:
    GTree();
    GTree( int containerSize );
};
```

Figure 4.11: GTree definition

in as type uUniform and the storage manager for the data part is passed in as a template parameter type so that an appropriate type can be chosen for fixed-size or variable-size data objects.

The constructor parameters are the initial file structure size, each segment size and the container size. The constructor initializes both expansion objects and storage managers, and then sets the root pointer to NULL, indicating an empty tree.

GTreeAccess

GTreeAccess inherits from PTreeAccess and defines the duration for which GTree is accessible. The class definition GTreeAccess in Figure 4.13 provides routines to operate on the tree. It is the

```
template<GTreeType> class GTreeAdmin : private PTreeAdmin {
  public:
    IndexContainer *Root;
    GTreeExpType<GTreeArgs> expobj;
    GTreeDataExpType<GTreeArgs> expobj_data;
    uUniform usm;
    SM sm;

    GTreeAdmin( int FileSize, int indexSize, int dataSize, int containerSize );
}; // GTreeAdmin<GTreeArgs>
```

Figure 4.12: GTreeAdmin definition

only way for application code to access the file structure contents. The constructor parameters are a reference to a GTree class object and the UNIX file name for the file structure, which is passed to PTreeAccess. The constructor retains the reference for subsequent access to the corresponding tree routines. The member routines Insert, Delete and Search are called by application programs to perform tree operations. These members cover the corresponding members in the tree object. Each routine makes the address space accessible by creating a GTreeWrapper object before performing an operation on the tree. Routines get and set are also needed to directly access the data segment. Given a pointer to a leaf node in the tree, the purpose of get is to get the data value from the database and assign it out to the shared memory of the application program. For example, the Search routine first calls GTree::Search, which returns a pointer to a leaf node, and then calls get to obtain the data value into a variable record, similar to an embedded SQL statement: exec sql select ... into : host variables. The routine set performs an update in the database directly given a pointer to the leaf node and the desired data value, similar to an SQL update statement: update table set column = ... where ....

```
template<GTreeType> class GTreeAccess {
    GTree<GTreeArgs> &tree;
    RepAccess<Rep> repacc;                      // access class for representative

    GTreeAccess( const GTreeAccess & );         // prevent copying
    GTreeAccess &operator=( const GTreeAccess );
  public:
    GTreeAccess( GTree<GTreeArgs> &tree, char *name );

    int Insert( LeafKey &r, Data *data, int size );
    int Delete(  Key &r, Data *data );
    bool Search( LeafKey &r, Data &record );
    void get( const LeafNode *p, Data &value );
    void set( LeafNode *p, const Data &value );
};
```

Figure 4.13: GTreeAccess definition

GTreeWrapper

The GTreeWrapper inherits from PTreeWrapper and makes the GTree directly accessible. The following is an example definition of GTreeWrapper:

```
template<GTreeType> class GTreeWrapper : private PTreeWrapper {
    GTreeWrapper( const GTreeWrapper & );      // prevent copying
    GTreeWrapper &operator=( const GTreeWrapper );
  public:
    GTreeWrapper( const GTreeAccess<GTreeArgs> &tr ) : PTreeWrapper( tr ) {
    } // GTreeWrapper<GTreeArgs>::GTreeWrapper
};
```

4.2.3 *S*Tree

An *S*Tree is the highest level class in the tree class hierarchy representing a user specialized persistent search tree. It inherits from GTree and can be a B+-tree, an R-tree or other database search trees. A file structure designer can choose an appropriate Key type, Data type, storage manager and the specific implementation and gets a new persistent file structure in a significantly shorter time than writing from scratch. In addition, insert, delete or search methods of the GTree can always

```
template<GTreeType> class BTree : public GTree<GTreeArgs> {
    friend class BTreeAccess<GTreeArgs>;

    char *fileName;
    BTreeAdmin<GTreeArgs> *admin;
  public:
    BTree( char *Name, int initSize, int indexSize, int dataSize, int containerSize );
    ~BTree();
};
```

Figure 4.14: B+-tree definition

be overridden in *S*Tree for new versions of tree operations. Figure 4.14 shows an example B+-tree definition.

Each *S*Tree instance generated from a generic *S*Tree type has several common parameters: the name of the UNIX file that contains the persistent file structure, the initial space allocated for the address space in bytes, as well as the initial size for each segment and the size of the container used in the index.

The *S*Tree contains a pointer to *S*TreeAdmin, which inherits from GTreeAdmin, and encapsulates all persistent administrative information at the beginning of the address space. The constructor of an *S*Tree makes a copy of the UNIX file name in shared memory, establishes a mapping to the file, makes the resulting address space accessible, obtains a pointer to the beginning of the address space to use as the location of the administrative object, and checks to see if the file is created on access. If the file has been newly created, the address space is extended to the specified size and the administrative object is created at the beginning, which initializes itself through its constructor, creating an empty tree. Otherwise, the file has been created before and the constructor checks if the file is the correct type by performing a dynamic type-check.

*S*TreeAdmin

*S*TreeAdmin inherits from GTreeAdmin. The constructor parameters are the initial file structure, segment and container sizes. The constructor passes them to GTreeAdmin, copies the file's type into the file as a "magic cookie", which is a type identifier used for dynamic type checking when the tree is made accessible. The member routine CheckType performs the dynamic type checking for the file. The following is an example B+-tree definition:

```
template<GTreeType> class BTreeAdmin : public GTreeAdmin<GTreeArgs> {
    char typeName[0];
  public:
    BTreeAdmin( int initSize, int indexSize, int dataSize, int containerSize );
    int CheckType( char *filename );
};
```

*S*TreeAccess

*S*TreeAccess inherits from GTreeAccess, and defines the duration for which the *S*Tree/GTree is accessible. If more specific operations are defined in *S*Tree, *S*TreeAccess needs to provide corresponding cover routines to operate on the tree. The following is an example B+-tree definition:

```
template<GTreeType> class BTreeAccess : public GTreeAccess<GTreeArgs> {
    friend class RangeGen<GTreeType>;
  public:
    BTreeAccess( BTree<GTreeArgs> &tree )
        : GTreeAccess<GTreeArgs>( tree, tree.fileName ) {}
    ...      // possible more routines
};
```

*S*TreeWrapper

*S*TreeWrapper inherits from GTreeWrapper and is used to make *S*Tree's address space accessible. The following is an example B+-tree definition:

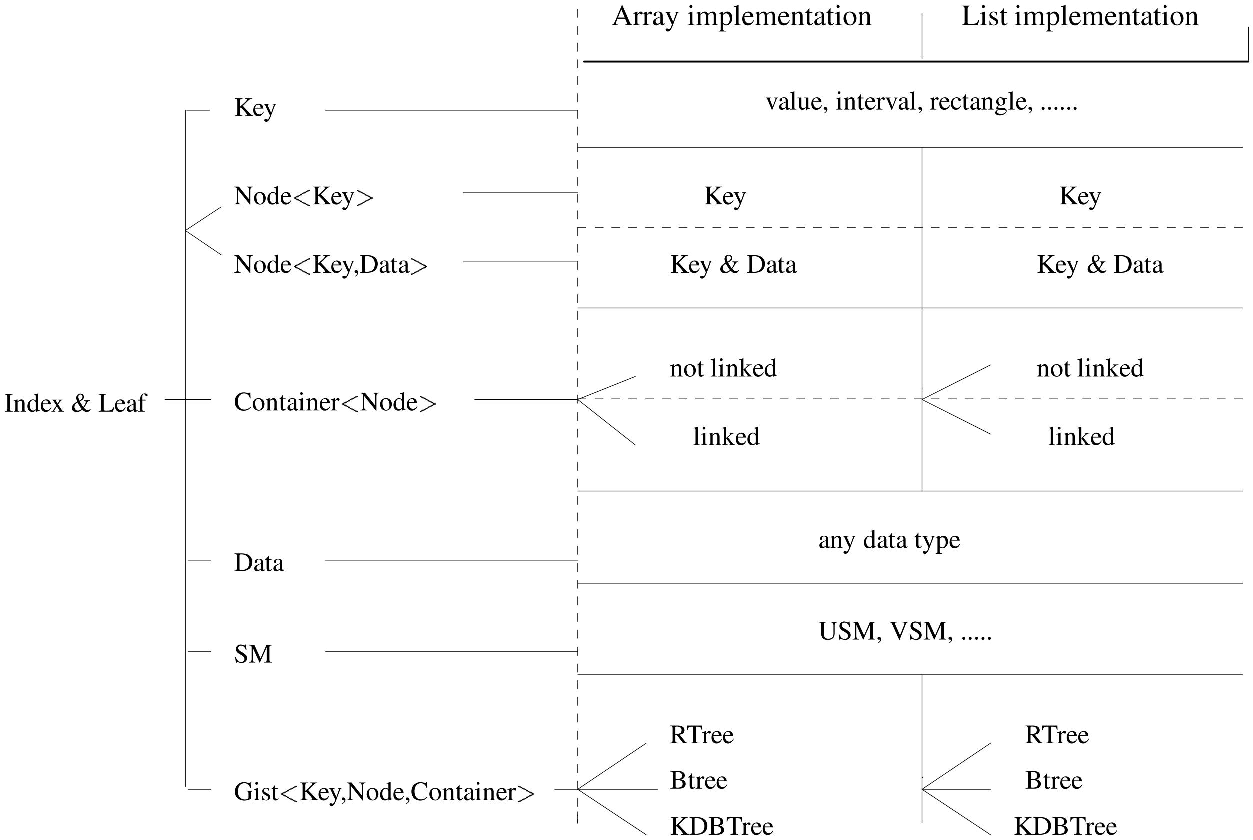


Figure 4.15: Components of Generalization Library

```
template<GTreeType> class BTreeWrapper : public GTreeWrapper<GTreeArgs> {
  public:
     BTreeWrapper( const BTreeAccess<GTreeArgs> &tr )
          : GTreeWrapper<GTreeArgs>( tr ) {}
};
```

4.3 Tree Class Library

Section 4.2.2 lists the template type parameters of the GTree class. This section discusses the details of each parameter and a small set of specialization components for these types available in the generalization library, including two common implementations, array and linked list for both index and leaf container structures and other various implementations. The components of the generalization library are showed in Figure 4.15.

4.3.1 Key

The Key class provides the type of the key in each tree node and the basic operations on keys. Examples of key structures include integers for data as in B+-trees and bounding boxes for regions as in R-trees. The index key type and the leaf key type can be the same, as in B+-trees and R-trees, but they can be different as in KDB-trees and other point-tree access methods. Therefore, there are two parameter types for keys: IndexKey and LeafKey. The key type is provided by the file structure designer.

4.3.2 Node<Key> and Node<Key,Data>

The Node class provides all fields needed for an entry in a container. It is a template class itself with Key type and possible Data type as parameters depending on where data is stored. In the GTree definition, the IndexNode is the type used for index nodes and LeafNode is the type used for leaf nodes in the tree. For different organizations of nodes inside each container, a node has different fields. For a container with array of nodes, an index node contains a (key, child pointer) pair. Figure 4.16 shows such a node class named arrayNode. The variable keyValue is the key of type Key. The variable child is the pointer to its child. In the general node class, child is a void pointer. The reason for this untyped pointer is that depending on the level in which the node is in the search tree, child can point to an index container, a leaf container or a data object. If a union type is used to define child pointer, the container types must be passed to arrayNode class, which is impossible because the node type is a template parameter for container classes, as discussed in Section 4.3.3. The arrayNode class provides an overloaded assignment operator and a print routine for easy manipulation of the container and for debugging.

```
template<class Key> class arrayNode {
  public:
     Key keyValue;
     void *child;

     arrayNode() : child(NULL) {}
     arrayNode( Key &r, void *ptr ) : keyValue(r), child(ptr) {}
     arrayNode &operator=( const arrayNode &e );
     void print();
};
```

Figure 4.16: Array Node Example

```
template<class Key> class listNode {
  public:
     Key keyValue;
     void *child;
     listNode *next;

     listNode() : child(NULL), next(NULL) {}
     listNode( Key &r, void *ptr ) : keyValue(r), child(ptr) {}
     listNode &operator=( const listNode &e );
     void print();
};
```

Figure 4.17: List Node Example

Figure 4.17 shows a possible implementation of a node in a container with linked list nodes. In addition to the key value and child pointer, each node has a link field, a pointer *next, used to construct the linked list.

A leaf node can contain data objects or a pointer to data objects. Figure 4.18 and 4.19 are example definitions of such nodes with array and linked list implementations, respectively.

The above commonly used node class implementations are provided in the library so that possible combinations can be chosen for the index and leaf nodes. File structure designers can always implement their own node classes.

```
template<class Key, class Data> class arrayDataNode {
  public:
     Key keyValue;
     Data data;         // or Data *data

     arrayDataNode();
     arrayDataNode( Key &r, Data &data);
     arrayDataNode &operator=( const arrayDataNode &e );
     void print();
};
```

Figure 4.18: Array Node with Data

```
template<class Key, class Data> class listDataNode {
  public:
     Key keyValue;
     Data data;         // or Data *data
     listDataNode *next;

     listDataNode();
     listDataNode( Key &r, Data &data );
     listDataNode &operator=( const listDataNode &e );
     void print();
};
```

Figure 4.19: List Node with Data

4.3.3 Container<Node>

Containers have nodes inside, so the Container type is a template class with Node as a type parameter. As discussed in Section 4.1.3, a Container class is used to provide common data variables and the interface for operations that an individual container needs to specify, like subscripting, returning the first/last node of the container and adding/deleting nodes to/from the container. The container can be an array or a linked list. Figure 4.20 and 4.21 list two versions of Container type definitions. numEntry is the maximum number of entries that the container can hold. Other variants may support a linear ordering among the nodes; leaf containers may be connected for easy sequential retrieval.

Associated with each container, there is a Cursor class, which is a generator for the container.

```
template<class Node> class Array : public Container<Node> {
  public:
     int numEntry;              // number of nodes in this container
  private:
     Node *firstNodePtr, *lastNodePtr;
     Node node[1];              // the first node
  public:
     class Cursor {             // iterator for the container
          int i;
          Array<Node> *array;
       public:
          Cursor( Array<Node> *array );

          Node *start();
          Node *succ();
          Node *pred();
     };

     Array( int size );                   // constructor
     Node &operator[ ]( int i );          // subscripting operator
     Node *&first();                      // first node
     Node *&last();                       // last node
     void printContainer();               // print content of container
     void AddNode( Node &node );          // add a node to container
     void DeleteNode( Node &node );       // delete a node from container
}
```

Figure 4.20: Array Container Example

No matter what kind of structure and implementation a container has, it must provide the Cursor class with the following common interface:

```
class Cursor {
     // . . .
  public:
     Node *start();     // the first node
     Node *succ();      // the next node
     Node *pred();      // the previous node
};
```

The Cursor class provides a common interface for iterating through containers so that the internal structure and implementation of a container is transparent.

```
template<class Node> class List : public Container<Node> {
  public:
     int numEntry;                            // number of nodes in this container
  private:
     ListExpType<Node> expobj;
     uUniform usm;
     Node *firstNodePtr, *lastNodePtr;        // pointers to the first/last node
  public:
     class Cursor {                           // iterator for the container
          Node *prev;
          Node *curr;
          List<Node> *list;
       public:
          Cursor( List<Node> *list );

          Node *start();
          Node *succ();
          Node *pred();
     };

     List( int size );                   // constructor
     Node *alloc( Node &e );             // allocate storage for node
     void free( Node *p );               // free storage
     Node *&first();                     // first node
     Node *&last();                      // last node
     void printContainer();              // print content of container
     void AddNode( Node &node );         // add a node to container
     void DeleteNode( Node &node );      // delete a node from container
}
```

Figure 4.21: List Container Example

4.3.4 Data

Data stored in a database may be a basic type, such as integer or float or complex, such as array or structure, and finally, a user-defined class. A database search tree should be able to index on all different kinds of data. Thus, the notion of "data" should be represented with minimal dependence on a special type. The search tree is generalized on the data type by taking the data type as a type parameter.

Each indexed datum in the database can be an arbitrary data object, fixed or variable sized. The data can be stored in the index nodes, but most commonly in the leaves of the search tree. Usually if the data objects are large, in order to maximize the fanout of the tree, only pointers (tuple ids) to the actual locations of data objects in the database are stored in leaves of the tree.

4.3.5 Storage Management

Similarly to the key type and the data type, the storage manager type is generalized, and is a type parameter of the search tree class. The type of the storage manager can be chosen from the following depending on the type of the data it manages:

- uniform: for storage of fixed size, which is specified with the creation of the storage manager
- variable: for storage of variable size, which is specified on a per allocation basis but can not be changed later on
- dynamic: for storage of variable size, which is specified on a per allocation basis and can be subsequently expanded or contracted

4.3.6 GIST<Key, Node, Container>

As discussed in Section 4.1.4, a set of *static* methods in class GIST provide behaviour routines for an individual search tree corresponding to a particular implementation. GIST is a template class with Key, Node and Container type as its type parameters. The following shows the definition of a GIST class. The file structure designer provides an implementation of a GIST class for each kind of access method.

```
template<class Key, class Node, class Container> class GIST {
  public:
     static bool Consistent( Key &key1, Key &key2 );
     static Key Union( Container *container );
     static Node *ChooseSubtree( Container *container, Key &r );
     static void Split( Container *container, Node &e, Container **newContainer );
};
```

Consistent: Given two nodes, returns true if they match a certain condition, false otherwise. The matching condition depends on the particular tree structure.

Union: Given a pointer to a container containing nodes $(k_1, ptr_1), ...(k_n, ptr_n)$, returns a key k that covers all nodes in the container, $k = k_1 \vee ... \vee k_n$.

ChooseSubtree: Given a pointer to a container and a key to insert, a node is selected to be the root of the subtree for insertion depending on the desired property of the tree. For a B-tree with ordered records in leaves, the node chosen leads to the correct leaf to insert the key according to the ordering. For an R-tree, the node chosen is the rectangle that needs the least enlargement to include the key r.

Split: Given a pointer to a full container and a node to be inserted, a new container is created and the nodes from the old container and the new node are distributed between the two containers. Different search trees usually have different algorithms on container splitting. The same search tree may also have different container splitting algorithms. For example, a quadratic and a linear algorithm can both be applied to an R-tree.

4.4 Tree Generators

The definition of GTreeGen is shown in Figure 4.22. GTreeGen is implemented using coroutines. Retaining both data and execution state is crucial for creating a generator to traverse a tree structure. The coroutine allows a generator to return control back to the caller after each node is extracted but retain its location in the tree structure [Buh95]. Specific tree generators are also coroutines inheriting from GTreeGen, for example:

```
template<GTreeType> uCoroutine RangeGen : public GTreeGen<GTreeArgs> { ... }
template<GTreeType> uCoroutine ContainmentGen : public GTreeGen<GTreeArgs> { ... }
```

The member routine uOver resets the generator to the root of the tree. Operator >> resumes the coroutine main to invoke member nextNode, which performs the recursive traversal of the tree, suspending back to operator >> for each node satisfying the query. The resume in operator >> restarts the coroutine in nextNode, where it has suspended. When a query finishes, the generator returns to the call in GTreeGen::main, which completes the traversal by setting root to NULL. Member GTreeGen::main then suspends back to operator >>, which returns a NULL data pointer and false. GTreeGen has two constructors. The first constructor allows the specification of a tree access object and is employed when the generator is going to be used only once for one particular tree object, as in:

```
for( gen( tree ); gen >> p ) {
    p->...    // reference data in the node
}
```

The second constructor is employed to create a generator that is subsequently initialized to work with any associated tree object. The association occurs through the uOver member routine.

In order to accommodate diverse queries, a specific query routine is passed to GTreeGen as a pointer to a function. For one query, the criteria used on index keys and leaf keys may be different.

For example, a containment window query needs to test the overlap relation between the query key and index keys and to test the containment relation between the query key and leaf keys. Hence, two pointers to query function are used in GTreeGen with Consistent applying to index keys and Query applying to leaf keys.

A counter (variable count) is used to control traversal if only a certain number of records are needed. For example, a query such as "get the first 10 records in the range from Key1 to Key2" can be performed by a generator declared as (where tr represents the tree file structure):

```
RangeGen<...> gen(tr,Key1,Key2,10);
```

The default value for the counter is -1, which means the query is conducted on the entire tree structure.

4.5 Summary

The essential nature of a database search tree and the general notion of a search key provide the basis for generalizing tree operations. Multiple levels of generalization are shown. The levels include tree functionality, container structure, tree behavior, tree generator, index and storage management. Finally, the implementation of the generalization is presented, including the hierarchy of persistent tree classes and the components of the generalized tree class library.

```
template<GTreeType> uCoroutine GTreeGen {
  protected:
    const GTreeAccess<GTreeArgs> *ga;
    IndexContainer *root;
    LeafNode *curr;
    LeafKey testkey;
    bool (*Consistent)( IndexKey key1, LeafKey key2 );
    bool (*Query)( LeafKey key1, LeafKey key2 );
    LeafContainer *parentPtr;
    int count;

    void nextNode( IndexContainer *c );

    void main() {
        nextNode( root );
        root = NULL;
        curr = NULL;
        uSuspend;
    }
  public:
    GTreeGen( const GTreeAccess<GTreeArgs> &ga,
          bool (*Consistent)(IndexKey, LeafKey),
          bool (*Query)(LeafKey, LeafKey),
          LeafKey testkey, int count = -1 );

    GTreeGen( bool (*Consistent)(IndexKey, LeafKey),
          bool (*Query)(LeafKey, LeafKey),
          LeafKey testkey, int count = -1 );

    void uOver( const GTreeAccess<GTreeArgs> &ga ) {
        RepWrapper wrapper( ga.repacc );

        GTreeGen::ga = &ga;
        root = ga.tree.SegZero->Root;
    }

    bool operator>>( LeafNode *&tp ) {
        RepWrapper wrapper( ga->repacc );
        if ( root != NULL ) uResume;
        tp = curr;
        return root != NULL;
    }
};
```

Figure 4.22: GTree Generator

Chapter 5

Example File Structures and Experiments

After constructing the generalized search tree and developing the specialization component library, the next goal is to demonstrate the feasibility of the generalization idea on database search trees; that is, to show that the overhead of generalization does not affect the performance of search trees significantly. The most effective way of doing so is to design and construct illustrative search tree structures and run experiments on them. B+-tree, R-tree and R*-tree file structures are developed from the generalized search tree toolkit.

5.1 B+-tree Example

Section 4.2.3 showed the definition of a B+-tree. A B+-tree can take advantage of the existing operations provided in GTree. The key class contains a value of the same type as the indexed data. For keys in ascending order, the implementation of the GIST methods are:

- **bool Consistent(Key &key1, Key &key2):** if key1 $<$ key2, then return true, otherwise return false

- **Key Union(Container *container):** return the last key, which is the biggest value, in the container
- **Node *ChooseSubtree(Container *container, Key &r):** return the address of the first node in the container whose key is greater than key r. If r is greater than all keys in the container, the last node is returned
- **void Split(Container *container, Node &e, Container **newContainer):** first half of old container and the new node go into the old container and the second half go into the new container

The queries supported by the B+-tree are exact match and range queries. To achieve efficient range queries, B+-tree range generators are developed on the basis of GTreeGen, taking full advantage of sequential scan. The overloaded coroutine main is shown in Figure 5.1, where querykey1 and querykey2 represent the range of the query and variable count is used to control the number of records needed for partial retrieval.

Several versions of B+-tree with different index or leaf structures can be developed very easily by specifying appropriate template parameters in the B+-tree definition. For the experiment, the B+-tree has an array of (key, pointer) pairs in its index containers and a linked-list with (key, data) in its leaf containers. To declare an instance of such a B+-tree, the following statement is used:

```
template<GTreeType> uCoroutine RangeGen : public GTreeGen<GTreeType> {
    const BTreeAccess<GTreeArgs> *ba;
    LeafKey querykey2;                          // the end of the range
    int count;

    void main() {
        nextNode( root );                       // get the first record in the range
        if( count > 0 ) count--;

        LeafContainer::Cursor cursor( parentPtr );
        LeafNode *start = cursor.current( curr->next );
        for( ; ; ) {        // sequential through rest leaves till reach the end of the range
            for( curr = start; curr; curr = cursor.succ() ) {
                if( Less( curr->rect, querykey2 ) ) root = NULL;
                uSuspend;
                if( count > 0 ) count--;        // -1 => infinite maximum number
                if( count == 0 ) break;
            } // get nodes out of the first leaf in the range

            if( count == 0 ) break;

            parentPtr = parentPtr->nextPtr;     // get the next leaf
            if( !parentPtr ) {                  // exhaust all nodes till last leaf
                root = NULL;
                uSuspend;
            } // if
            cursor.over( parentPtr );
            start = cursor.start();
        } // for

        root = NULL;
        uSuspend;
    }
  public:
    RangeGen( LeafKey querykey1, LeafKey querykey2, int count = -1 );
    RangeGen( const BTreeAccess<GTreeArgs> &ba, LeafKey querykey1,
        LeafKey querykey2, int count = -1 );

    void uOver( const BTreeAccess<GTreeArgs> &ba );
    bool operator>>( LeafNode *&tp );
};
```

Figure 5.1: BTree Range Generator Example

```
BTree< Bkey, arrayNode<Bkey>, Array< arrayNode<Bkey> >,
     arrayGist< arrayNode<Bkey>, Bkey, Array< arrayNode<Bkey> > >,
     Bkey, listDataNode<Bkey,Data>, List< listDataNode<Bkey,Data> >,
     listGist< listDataNode<Bkey,Data>, Bkey, List<listDataNode<Bkey,Data> > >,
      > btree( "testdb",           // UNIX file name
          // creation only arguments
          10000*1024,              // initial size of segment
          10000*1024,              // index size
          1024                     // container size
        );
```

The node types and container types can be chosen from the generalization library. Only the key type and GIST methods need to be specified. Two different implementations for the same set of GIST method are used here, arrayGist and listGist. The UNIX file name for this B+-tree file structure is testdb with an initial size of 10M bytes. The container size chosen is 1K.

5.2 R-tree Example

Similar to the B+-tree, an R-tree is defined as:

```
template<GTreeType> class RTree : public GTree<GTreeArgs> {
     friend class RTreeAccess<GTreeArgs>;

     char *fileName;
     RTreeAdmin<GTreeArgs> *admin;
  public:
     RTree( char *Name, int initSize, int indexSize, int dataSize, int containerSize );
     ~RTree();
};
```

In the 2-dimensional domain, the keys in the tree are 4 numbers of type float, representing the upper-left and lower-right corners of rectilinear bounding rectangles for 2d-objects. The following is the implementation of the GIST methods:

- **bool Consistent(Key &key1, Key &key2):** if key1 is contained in key2, then return true, otherwise return false

- **Key Union(Container *container):** return a rectangle having the smallest lower-right corner and the biggest upper-left of all rectangles in the container

- **Node *ChooseSubtree(Container *container, Key &r):** return the address of the node in the container whose rectangle needs the least enlargement to include the key r (If several nodes qualify, one node is chosen randomly.)

- **void Split(Container *container, Node &e, Container **newContainer):** splitting uses Guttman's quadratic splitting algorithm

Besides exact matching, the queries supported are point queries and window queries including containment, enclosure and intersection queries. The generator classes for these queries simply inherit from GTreeGen class and pass the query routines to the constructor of GTreeGen. For example, the following is the generator for containment queries, where the query routines used are Overlap, for querying on index containers, and Contained, for querying on leaf containers:

```
template<GTreeType> uCoroutine ContainmentGen : public GTreeGen<GTreeType> {
  public:
     ContainmentGen( const RTreeAccess<GTreeArgs> &tr, LeafKey querykey,
                      int count = -1 ) :
          GTreeGen<GTreeType>( &Overlap, &Contained, testkey, count ) { ... }
};
```

An R-tree file structure with separate index and data parts is built. To declare an R-tree instance with backing-store file name testdb and initial file size 2500K is:

```
RTree< Rect, indexNode<Rect>, Array<indexNode<Rect> >,
     arrayGist<indexNode<Rect>,Rect,Array<indexNode<Rect> > >,
     Rect, indexNode<Rect>, Array<indexNode<Rect> >,
     arrayGist<indexNode<Rect>,Rect,Array<indexNode<Rect> > >,
     Data, uUniform
      > rtree( "testdb",           // UNIX file name
       // creation only arguments
       2500*1024,             // initial size of segment
       1000*1024,             // index size
       1000*1024,             // data size
       1024                   // container size
     );
```

5.3 R*-tree Example

As introduced in Section 2.1.3, R*-trees differ from R-trees mainly in the insertion algorithm. An R*-tree inherits from the R-tree and uses an overloaded insertion routine to implement the forced reinsertion policy:

```
template<GTreeType> class RStarTree : public RTree<GTreeArgs> {
     friend class RStarTreeAccess<GTreeArgs>;
  public:
     RStarTree( char *Name, int initSize, int indexSize, int dataSize, int containerSize );
     ~RStarTree();

     int Insert( LeafKey &r, Data *data, int size );          // overloaded routine
};
```

To achieve dynamic reorganizations of old rectangles, the R*-tree forces nodes to be reinserted during the insertion routine. Whenever an overflow occurs, an algorithm called OverflowTreatment is used:

OverflowTreatment: if the container is not the root and this is the first call of OverflowTreatment at this container level during the insertion of one data rectangle, then invoke ReInsert, else invoke Split as in the R-tree.

ReInsert: for all $MAX+1$ nodes of a container C, compute the distance between the centers of their rectangles and the center of the bounding rectangle for C; sort the nodes in decreasing order of their distances; remove the first 30% nodes from C and adjust rectangle of C; invoke Insert for the nodes removed.

Declaring the R*-tree is similar to that of the R-tree, except that for template parameter GIST, the ChooseSubtree method uses an alternative algorithm for testing a rectangle's area, margin and overlap in different combination in order to minimize overlaps.

5.4 Experiments

For each experiment, the performance gathered is the elapsed time, which is the real clock time from the beginning to the end of a test run. All experiments show that the search trees developed from the generalized model have competitive performances comparing to traditional search trees developed from scratch.

5.4.1 B+-trees

Query performances are compared for the B+-tree inheriting from GTree and a B+-tree developed from scratch, both using memory mapping. For each of the B+-trees, 100,000 uniformly distributed records are generated with keys taken from the unit interval. The records are inserted into the B+-tree. For the resulting B+-tree, two kinds of queries are conducted. First, for exact matches, query files with three different distributions, normal, uniform and random, are generated with 10, 100, 1000 query keys in each file for all distributions. For each query file, the experiment searches for the specified key in the B+-tree and retrieved the corresponding data record. Second, for multiple-response searches, four query files are generated with each requiring 10,000 records to be read in total in response to a collection of multiple-response queries of a given size. An individual query in each file is specified by a key based on a uniform distribution and a fixed number of records (the size of the range query) to be read sequentially starting from the specified key. Each of the query files is described by a tuple <n,m> where n is the total number of queries in the file and m is the size of each query. For example, <10,1000> implies 10 queries of size 1,000 records each and the query file consists of 10 keys from a uniform distribution. For each key, the experiment searches for the key in the B+-tree and reads up to 1,000 data records sequentially by following the leaf links of the B+-tree. Tables 5.1 and 5.2 show the experiment results for exact matching queries and multiple-response queries respectively. The results are essentially identical, given that the experiments were run with other users on the computer.

Query Distribution	number keys	Elapsed Time (in secs) of Each Run Generalized memory mapped B+-tree	memory mapped B+-tree
normal	10	0.14	0.16
	100	0.15	0.17
	1000	0.33	0.33
uniform	10	0.15	0.16
	100	0.16	0.17
	1000	0.35	0.34
random	10	0.15	0.16
	100	0.16	0.17
	1000	0.36	0.35

Table 5.1: Experiment Results of Exact Matching Queries on B+-trees

The Elapsed Time (in secs) of Each Run

Query Distribution	Generalized memory mapped B+-tree	memory mapped B+-tree
<1,10000>	0.58	0.58
<10,1000>	0.66	0.61
<100,100>	0.84	0.77
<1000,10>	1.85	1.84

Table 5.2: Experiment Results of Multi-response Queries on B+-trees

5.4.2 R-trees and R*-tree

The experiment is conducted on an R-tree developed using traditional I/O with a generalized R-tree and R*-tree using memory mapping. Each tree is populated with data obtained from a standardized testbed [BKSS90]. The data consists of 100,000 2-dimensional rectangles where each rectangle is assumed to be in the unit square $[0, 1]^2$. The centers of the rectangles follow a 2-dimensional independent uniform distribution. These rectangles are the minimum bounding rectangles of elevation lines from real cartography data. The query files used for the experiment are also taken from the same testbed and consist of 1000 point queries and 400 each of the containment, enclosure and intersection window queries. The experiment results are shown in Table 5.3. The generalized R-trees are slightly slower. It is difficult to determine if the slow down is the result of the generalization or the difference between traditional I/O and memory mapping I/O. Since performance of the two memory mapping B+-trees is identical, the cause of the difference is probably the I/O.

5.5 Summary

This chapter demonstrated the feasibility and viability of the generalization on database search trees. Example B+-tree, R-tree and R*-tree file structures were easily built to illustrate different

The Elapsed Time (in secs) of Each Run

Query Type	Generalized memory mapped R-tree	Generalized memory mapped R*-tree	Traditional I/O R-tree
containment	0.18	0.16	0.17
enclosure	0.15	0.15	0.11
intersection	0.22	0.22	0.18
point	0.48	0.47	0.34

Table 5.3: Experiment Results on R-trees and R*-tree

capabilities of the generalization and for running experiments. The experimental results show that the generalization does not affect the performance of search trees significantly.

Chapter 6

Conclusion

The generalized search tree provides all the basic search tree logic required to build most tree based access methods. It unifies distinct structures, such as B-trees and R-trees. Specific search trees can be built from the generalized model significantly faster and more reliably than hand-coding. As well, flexibility is provided for search tree developers to choose the kind and number of indices, the internal representation and implementation of index and leaf, as well as the specific algorithms for particular operations. The tree generators are able to support an extensible set of queries and provide a unified interface for the queries.

In summary, the generalizations for search tree file structures are:

- Functionality: basic search tree operations are captured in a base tree class.
- Structure: container structure makes internal representation and implementation transparent.
- Behaviour: parameterization of behaviour routines allow different algorithms and implementation.

- Generator: extensible set of queries can be built easily and have unified interface.
- Index: flexibility is provided in the kind and number of indices.
- Storage management: nested storage managers provide efficient memory management at all levels.

A generalized class library is created including a small set of specialized components so that a search tree file structure developer is not required to write all the components from scratch. By carefully selecting components from the library and only specializing components needed for a new algorithm/data-structure, a new tree access method can be created, and subsequently tested, significantly faster than via the traditional approach, as experienced in developing the example tree file structures in this thesis.

Performance experiments demonstrate the feasibility and viability of the generalization idea to achieve efficient implementation of tree access methods.

6.1 Future Work

This thesis concentrates on the generalization on database search trees. More work is needed to explore the idea on other database access methods, such as hashing techniques. Currently, the generalization of tree access methods is only pursued in a memory mapped environment, but the ideas are also applicable in a traditional environment and possibly on more complex data structures.

Bibliography

[ABC^{+}83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BGW92] Peter A. Buhr, Anil K. Goel, and Anderson Wai. μDatabase : a toolkit for constructing memory mapped databases. *Persistent Object Systems*, pages 166–185, 1992.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Information*, 1(3):173–189, 1972.

[Buh95] Peter A. Buhr. *Understanding Control Flow with Concurrent Programming Using μC++*. http://www.uwaterloo.ca/ cs342, 1995.

[Com79] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–138, 1979.

[DD94] H. M. Deitel and P. J. Deitel. *C How to Program*. Prentice Hall, 2 edition, 1994.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and R. Strong. Extendible hashing: a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[Fre87] M. Freeston. The bang file: a new kind of grid file. In *ACM SIGMOD International Conference on Management of Data*, pages 260–269, 1987.

[GG98] Volker Gaede and Oliver Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):171–231, 1998.

[Goe96] Anil K. Goel. *Exact Positioning of Data Approach to Memory Mapped Persistent Stores : Design, Analysis and Modelling*. PhD thesis, University of Waterloo, 1996.

[Gun88] O. Gunther. Efficient structures for geometric data management. In *LNCS*, number 337. Springer-Verlag, 1988.

[Gun89] O. Gunther. The cell tree: an objected-oriented index structure for geometric databases. In *Proceedings of 5th IEEE International Conference on Data Engineering*, 1989.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data*, pages 47–54, 1984.

[HNP95] J. M. Hellerstein, J. F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, 1995.

[HSW88] A. Hutflesz, H. W. Six, and P. Widmayer. Globally order preserving multidimensional linear hashing. In *Proceedings of 6th IEEE International Conference on Data Engin eering*, pages 572–579, 1988.

[HSW89] Andreas Henrich, Hans-Werner Six, and Peter Widmayer. The lsd tree : spatial access to multidimensional point and non point objects. In Peter M.G. Apers and Gio Wiederhold, editors, *Proceedings of the International Conference on Very Large Data Bases*, Amsterdam, Netherlands, 1989.

[HSW90] A. Hutflesz, H. W. Six, and P. Widmayer. The r-file: an efficient access structure for proximity queries. In *Proceedings of 6th IEEE International Conference on Data Engin eering*, pages 372–379, 1990.

[Knu73] D. Knuth. sorting and searching. In *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[Lar80] P. A. Larson. Linear hashing with partial expansions. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 224–232, 1980.

[Lit80] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of the Sixth International Conference on Very Larg e Data Bases*, pages 212–223, 1980.

[LS89] David B. Lomet and Betty Salzberg. A robust multi-attribute search structure. *IEEE Conference on Data Engineering*, pages 296–304, 1989.

[LS90] David B. Lomet and Betty Salzberg. The hb-tree: a multiattribute search structure. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.

[Nah90] Stefan Naher. Leda, 1990.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[OM84] J. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190, 1984.

[OS90] Y. Ohsawa and M. Sakauchi. A new tree type data structure with homogeneous node suitable for a large spatial database. In *Proceedings of 6th IEEE International Conference on Data Engin eering*, pages 296–303, 1990.

[Rob81] J. T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[SK88] B. Seeger and H. P. Kriegel. Techniques for design and implementation of spatial access methods. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 360–371, 1988.

[SL95] Alexander Stepanov and Meng Lee. *The Standard Template Library*, 1995.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In Peter M Stocker and William Kent, editors, *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 507–518, Brighton, England, 1987.

[SSH86] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in database systems. In *Proceedings of 1st International Conference on Expert Database Systems*, 1986.

[Sto86] Michael Stonebraker. Inclusion of new types in relational database systems. In *Proceedings of 4th IEEE International Conference on Data Engin eering*, pages 262–269, Washington, D.C., 1986.

[Str97] Bjarne Stroustrup. *C++ Programming Language*. addison wesley, 1997.