Index Structure For Metadata Extracted From Large Hypertext Collections

Dissertation

Submitted to the

Central Department of Computer Science and Information Technology

Tribhuvan University

In partial fulfillment of the requirements for the degree of

Masters Degree in Computer Science and Information Technology

By

Achyut Pd. Pathak

Dissertation Supervisor

Prof. Dr. Srinath Srinivasa

International Institute of Information Technology

Bangalore (**IIIT-B**), INDIA

July, 2008

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology
Kirtipur, Kathmandu
Nepal

LETTER OF CERTIFICATE

This is to certify that the dissertation work entitled **"Index Structure For Metadata Extracted From Large Hypertext Collections"**, submitted by Achyut Pd. Pathak has carried out under my supervision and guidance. In my best knowledge this is an original work in computer science and no part of this dissertation has been published or submitted for the award of any degree else where in the past.

.....

Prof. Dr. Srinath Srinivasa
International Institute of Information Technology
Bangalore (IIIT-B), INDIA
(Supervisor)

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology
Kirtipur, Kathmandu
Nepal

LETTER OF APPROVAL

We certify that we have read this dissertation and in our opinion it is satisfactory in the scope and quality as a dissertation in the partial fulfillment for the requirement of Masters Degree in Computer Science and Information Technology.

Evaluation Committee

Head, Central Department of Computer Science and Information Technology Tribhuvan University	Prof. Dr. Srinath Srinivasa International Institute of Information Technology, Bangalore (IIIT-B) (Supervisor)
(External Examiner)	(Internal Examiner)

Date:_____

ACKNOWLEDGEMENTS

It is a great pleasure for me to acknowledge the contributions of a large number of individuals to this work. First of all, I would like to thank my advisor **Prof. Dr. Srinath Srinivasa** (IIIT-B) for giving me an opportunity to work under his supervision and for providing me guidance and support through out this work. I would like to thank **Mr. Mandar M. Mutalikdesai** (IIIT-B) for his valuable comments and suggestions on my work.

I would like to express my sincere gratitude to **Prof. Dr. Devi Dutta Paudyal** (Former Head, Central Department of Computer Science and Information Technology) for his inspiration and encouragement during two years study of my Master Degree.

I would like to express my gratitude to the respected teachers **Prof. Dr. Shashidhar Ram Joshi**, **Prof. Sudarshan Karanjeet**, **Asst. Prof. Arun Timilsina, Dr. Tanka Dhamala** (Head, CDCSIT), **Prof. Dr. Laxmi P. Gewali** (University of Nevada, Las Vegas, USA), **Asst. Prof. Min B. Khati**, **Hemanta B. G.C.** and all other teachers who have taught us in our Master Degree.

Finally, I am in debt to my friends **Ananda Agrawal, Lalita Sthapit, Rashmi Shrestha, Kamal Bista and Dinesh Khadka** for their fruitful discussions. Last but not least, I would like to thank my family members for their constant support and encouragement.

Achyut Pd. Pathak

ABSTRACT

Growing amount of hypertext data can be found in various contexts like weblogs and online journals, intranet webs, the World Wide Web (WWW), online communities, intra-organizational wikis and other collaborative content management platforms. In such collections, the combination of content and hyperlink structures reflect several interesting information about various phenomena like existence of cyber communities, the documents similar to a given document, the popularity and importance of documents, the probability of reaching a document from any other document by following a sequence of hyperlinks etc. These can all be determined by analyzing a hypertext web. So, different kinds of analysis can be done on hypertext collections. Doing analysis requires locating and finding some information in hypertext collection. To locate information in hypertext database requires the use of an index. Since hypertext database is large in size, we need an efficient index structure to locate information in hypertext collection. Keys are used to construct the index and to search information in the index. Urls of web pages are used as keys to construct the index for hypertext collections. Since Urls of pages are variable in length, index that supports variable length keys is needed. To achieve these, a multilevel index supporting variable length key has been constructed as an index for hypertext collections.

LIST OF FIGURES

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Hypertext

Hypertext can be defined as "non-sequential reading and writing". Hypertext systems contain frames of text, pictures, sound and animation that are organized non-linearly in a network of linked frames. From any frame, users can access a variety of other frames containing text or other media [3]. Users follow various sequences of frames and links to retrieve the information they require or add new frames and connections between them. Hypertext systems with non-textual contents are called "hypermedia" to denote the fact that hypertext can not only include text but also non-textual data such as sound, images, animation and audio-visual data in digitized forms [10].

The origin of hypertext goes back to 1940s when Bush outlined his vision of Memex, which was to have some hypertextual features such as linking and associating different parts of documents in a way that would facilitate what he called associative thinking [4]. Twenty years later, inspired by Bush's idea, Nelson coined the terms hypertext and hypermedia to describe the associative linking of information into large networks [5].

Today, various hypertext systems, though most of them are experimental, have been developed in different areas such as education and publishing. Systems developed tend to be small in size with not-so-sophisticated information retrieval capabilities, because hypertext systems are relatively new phenomena.

The ever-increasing universe of electronic information, for example as found on the World Wide Web, competes for the effectively fixed and limited attention of people. Both consumers and producers of information want to understand what kinds of information are available, how desirable it is, and how its content and use change through time. The linkage structure of hypertext exhibits different interesting phenomena. Several different kinds of analysis can be done on hypertext collections. One kind of analysis can be done by retrieving a particular document from a large hypertext database and do further analysis on that document. To retrieve a document from a large hypertext

database or to locate information in the WWW by any means other than naive hyperlink traversal requires the use of an index.

1.2 Index

Indexes are structures that are used for searching data. Specially, they are used to speed up the retrieval of records in response to certain search conditions. The index structures typically provide alternative ways of accessing the records without affecting the physical placement of records on disk. They enable efficient access to records based on the indexing fields that are used to construct the index. Basically, any field of the file can be used to create an index and multiple indexes on different fields can be constructed on the same file. A variety of indexes are possible, each of them uses a particular data structure to speed up the search. To find a record based on some indexing field, we have to initially access the index which points to block in the file where desired data are located. The most prevalent types of indexes are based on ordered files (single-level index) and tree data structures (multilevel indexes). Indexes can also be constructed based on hashing or other search data structures.

1.3 Single-Level Ordered Indexes

Ordered index access structures are similar to the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book.

For a file with a given record structure consisting of several fields, an index access structure is usually defined on a single field of a file, called an indexing field. The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are ordered so that we can do a binary search on the index. The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient.

1.3.1 Primary Indexes

A primary index is defined on primary key field of the data file. Each index entry has two fields. The first field is of the same data type as the ordering key field-called the primary key-of the data file, and the second field is a pointer to a disk block. There is one index entry in the index file for each block in the data file. Each index entry has the value of the primary key field for the first record in a block and a pointer to that block as its second field values. The total number of entries in the index is the same as the number of disk blocks in the ordered data file. Primary index is a sparse index, since it does not contain entry for every search key value in the data file.

Since there are fewer index entries than records in the data file and each index entry is smaller in size that a data record, primary index file needs fewer blocks and more entries can fit in one block. A binary search on the index file hence requires fewer block accesses than a binary search on the data file. If the primary index file contains b_i blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record.

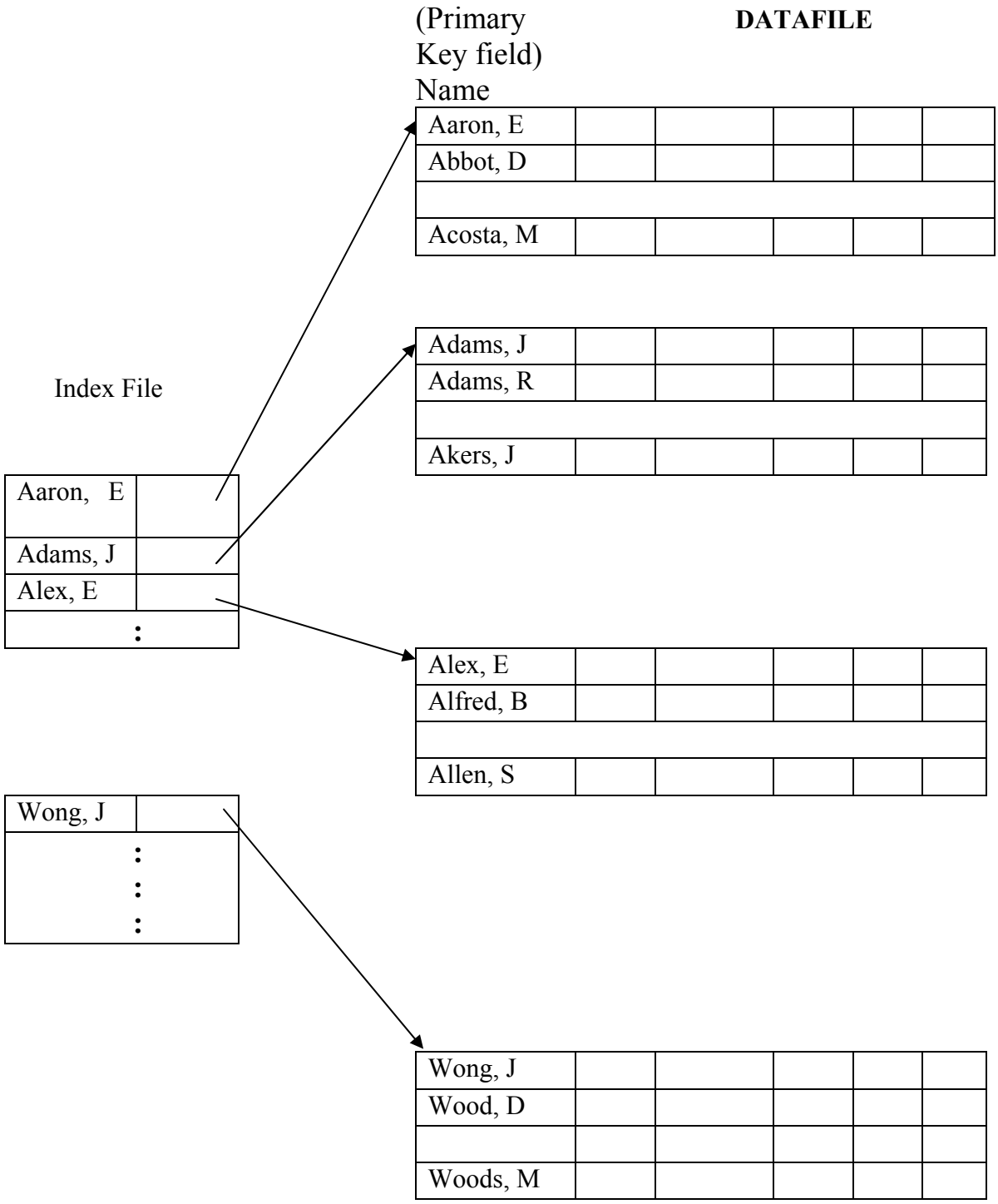


Fig 1.1 Primary index

1.3.2 Clustering Indexes

If records of a file are physically ordered on a nonkey field which does not have a distinct value for each record then that field is called the clustering field. Clustering index is defined on clustering field to speed up retrieval of records that have the same value for the clustering index. This differs from a primary index, which requires that the ordering field of the data file have a distinct value for each record.

Each entry in the clustering index has two fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer. There is one entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the first block in the data file that has a record with that value for its clustering field.

A clustering index is a sparse index, because it has an entry for every distinct value of the indexing field which is a nonkey by definition and hence has duplicate values rather than for every record in the file.

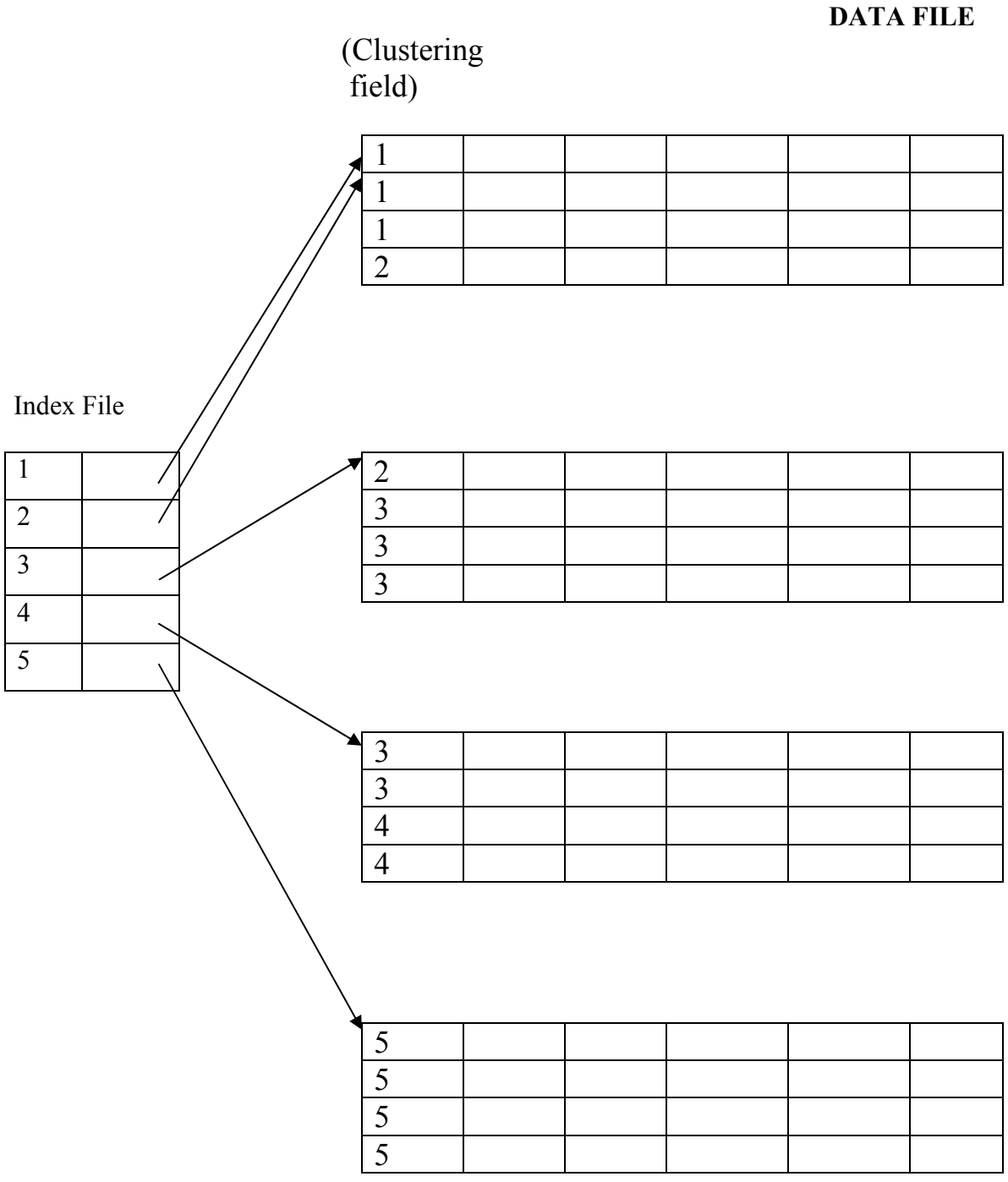


Fig 1.2 Clustering index

1.3.3 Secondary Indexes

A secondary index may be on a field which is a candidate key and has a unique value in every record, or a nonkey with duplicate values. The index is an ordered file with two fields: the first field is of the same data type as the indexing field. The second field is a block or a record pointer. There can be many secondary indexes for the same file. If secondary index access structure is constructed on a key field that has a distinct value for every record then such field is called a secondary key. In this case there is one index entry for each record in the data file, which contains the value of the secondary key for the record and a pointer to the record block or to the record itself. Hence, such an index is dense.

Two fields of secondary index can be referred as <K(i), P(i)>. The entries are ordered by value of K(i), So binary search can be performed on the index file. Block anchors can not be used because the records of the data file are not physically ordered by values of the secondary key field. That is why an index entry is created for each record in the data file, rather than for each block, as in the case of primary index.

A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However the improvement in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a linear search on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main file, even if the index did not exist.

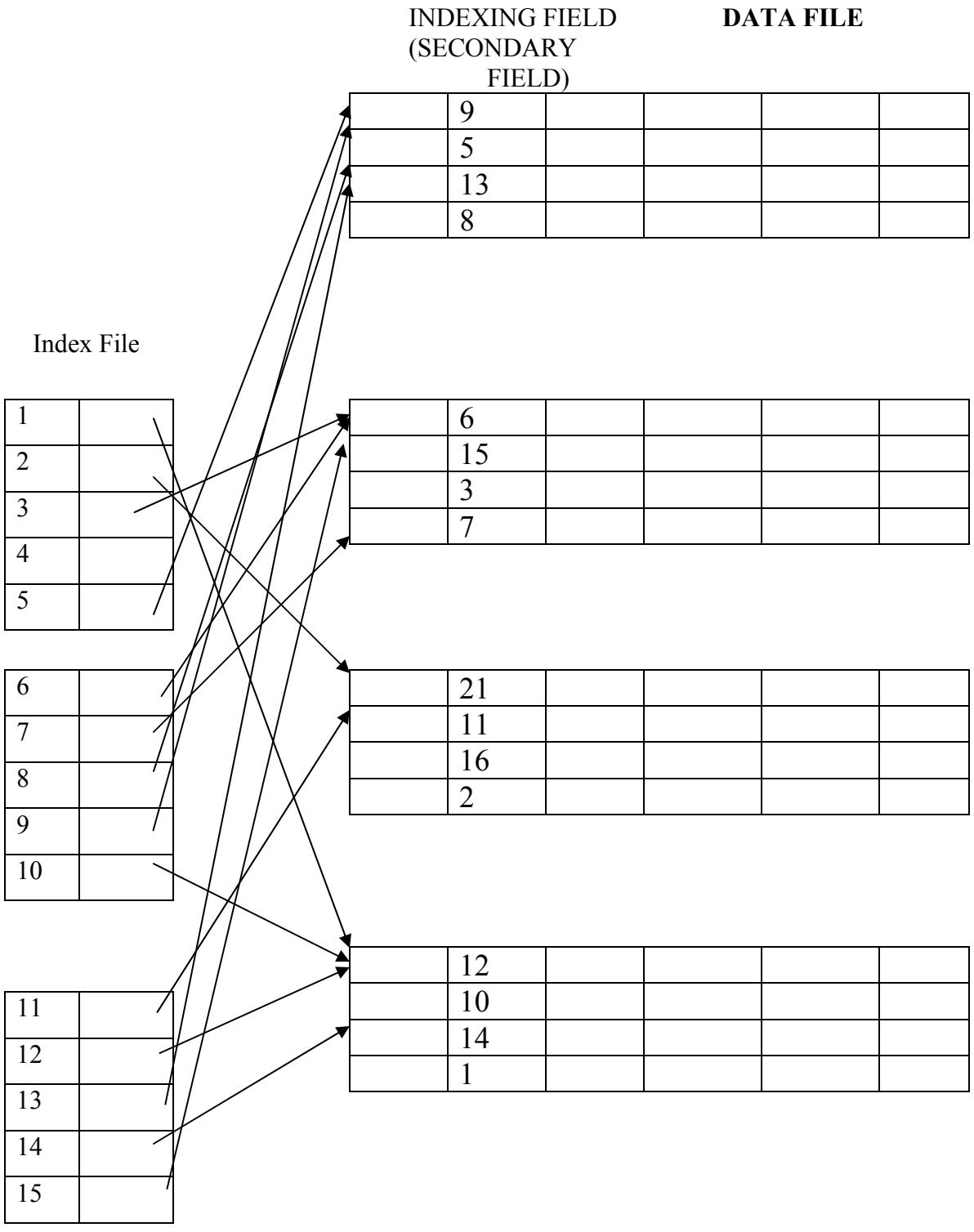


Fig 1.3 A dense secondary index

1.4 Hash Indexing

Index structure can be created based on hashing. When hashing is used, the index entries <K, Pr> can be organized as a dynamically expandable hash file. When a key is to be searched, a hash search function on K is used. Once an entry is found, the pointer Pr is used to locate the corresponding record in the data file. Hash based indexing is excellent for equality selections.

1.4.1 Static hashing

When a key is to be searched, a hash function is applied to identify the bucket to which it belongs and then search this bucket. Data entries are maintained in sorted order by search key value to speed up the bucket search. To insert a data entry, the hash function is used to identify the correct bucket and then put the data entry there. If there is no space in the bucket for this data entry, we allocate a new overflow page, put the data entry on this page and add the page to the overflow chain of the bucket. When a data entry is to be deleted, we use the same hash function to identify the bucket, locate the data entry by searching the bucket, and then remove it. If this data entry is the last in an overflow page, the overflow page is removed from the overflow chain of the bucket and added to a list of free pages.

The number of buckets is fixed in static hashing. If a file shrinks greatly, a lot of space is wasted. Similarly, if a file grows a lot, a long overflow chains develop, resulting in poor performance. This is the main limitation of static hashing.

1.4.2 Dynamic Hashing

Dynamic hashing techniques allow the number of buckets to be modified dynamically. It deals with inserts and deletes gracefully. Two most commonly used dynamic hashing techniques are extendible hashing and linear hashing.

1.4.2.1 Extendible hashing

When a new data entry is to be inserted into a full bucket, we reorganize the file by doubling the number of buckets and re-distributing the entries across the new set of

buckets. This solution has one major drawback- the entire file has to be read, and twice as many pages have to be written to achieve the reorganization. This problem can be overcome by using a directory of pointers to buckets and double the size of the number of buckets by doubling just the directory and splitting only the bucket that overflowed.

1.4.2.2 Linear Hashing

Linear hashing is a dynamic hashing technique, like extendible hashing, adjusting gracefully to inserts and deletes. In contrast to extendible hashing, it does not require a directory, deals naturally with collisions and offer a lot of flexibility with respect to the timing of bucket splits. If the data distribution is very skewed, however, overflow chains could cause linear hashing performance to be worse than that of extendible hashing.

Hash-based indexing techniques cannot support range searches, unfortunately. Tree-based multilevel indexing can support range searches efficiently and are almost as good as hash-based indexing for equality selections. Thus, many systems choose to support only tree based multilevel indexing.

1.5 Multilevel Indexes

In ordered index file, a binary search is applied to the index to locate pointers to a disk block or to a record in the file having a specific index field value. A binary search requires approximately ($\log_2 b_i$) block accesses for an index with b_i blocks, because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2.

The idea behind a multilevel index is to reduce the part of the index file that we continue to search by bfr_i or fo, the blocking factor for the index, which is larger than 2. Hence the search space is reduced much faster. The value bfr_i is called the fan-out of the multilevel index. Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses, which is a small number than for binary search if the fan-out is larger than 2.

A multilevel index considers the index file, which we refer to as the first level of a multilevel index, as an ordered file with a distinct value for each k (i). Hence we can

create a primary index for the first level; this index to the first level is called the second level of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for each block of the first level. The blocking factor for the second level and for all subsequent levels is the same as that for the first-level index, because all index entries are the same size-each has one field value and one block address. If the first level has r1 entries, and the blocking factor (which is also the fan-out) for the index is f0, then first level needs $\lceil(r1/f0)\rceil$ blocks, which is therefore the number of entries r2 needed at the second level of the index. We only require a second level only if the first level needs more than one block of disk storage, and we require a third level only if the second level requires more than one block as well. We can repeat this process until all the entries of some index level t fit in a single block. This block at the t^{th} level is called the top index level. Each level reduces the number of entries at the previous level by a factor of f0 (the index fan-out).

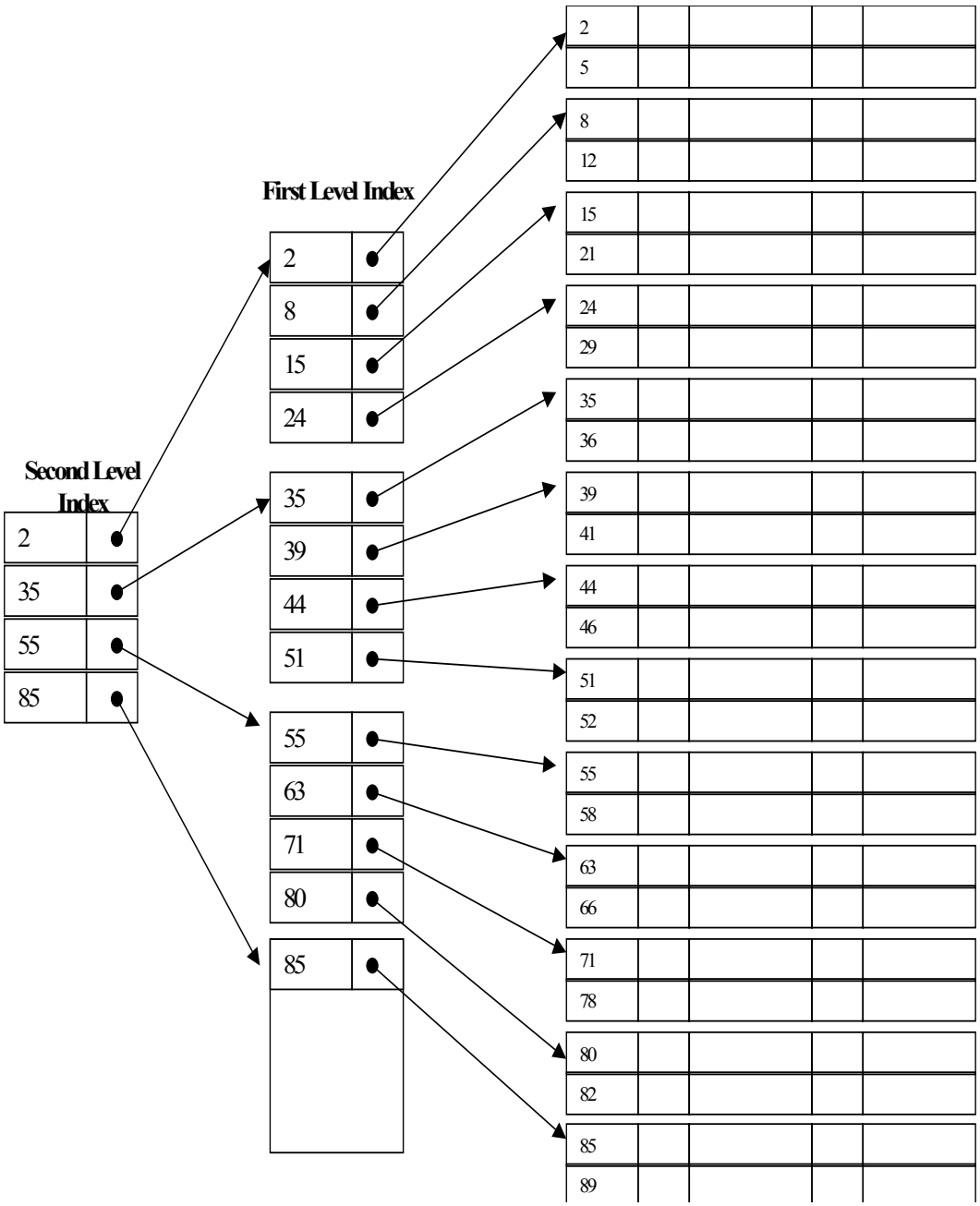


Fig 1.4 Multilevel index

CHAPTER 2: BACKGROUND AND PROBLEM DEFINITION

2.1 Background

Growing amount of hypertext data can be found in various contexts like weblogs and online journals, intranet webs, the World Wide Web (WWW), online communities, intra-organizational wikis and other collaborative content management platforms [1]. i.e. Hypertext collections abound in various contexts. In such collections, the combination of content and link structures reflects several interesting phenomena. Hypertext collections are in the form of graphs. The combination of content and linkage structure of a hypertext collection encloses interesting information about various phenomena [6]. For example, the existence of cyber communities [7], the hierarchical structure of an organization, the documents similar to a given document [8], the popularity and importance of documents [9], the probability of reaching a document from any other document by following a sequence of hyperlinks etc. can all be determined by analyzing a hypertext web. Graph theoretic analysis yields several useful insights into the dynamics of hypertext webs.

However, separate experiments need to be scripted for each individual analysis presently. It is cumbersome to write and manage a large number of standalone scripts every time a hypertext collection is analyzed for some phenomenon. Moreover, many of these analyses require huge amounts of time owing to the complexity of operations as well as size of the underlying hypertext collection. Hence a unified framework for online analytical processing (OLAP) [1] for large hypertext collection has been proposed.

Using the proposed OLAP framework, the user will be able to perform various kinds of analyses by executing them as queries. The user can query not only the entire hypertext collection, but also desired subsets of it. Such user-defined abstractions of search spaces reflect domain knowledge. Hence, queries that are answered within such a context are likely to be more useful than those that blindly search the entire data set.

In order to realize the proposed OLAP model for hypertext collections in a seamless fashion, [1] followings are identified as major challenges: (1) Designing a data model that supports handling of user-defined search-spaces in the form of views, (2) Designing a storage model that supports quick creation, updation, storage and retrieval of various views of the underlying data. (3) Designing a query algebra and language for constructing complex analytical queries, and (4) Designing a query processing engine for online execution of analytical queries using indexes, materialized views and pre-computed summaries.

2.2 Problem Formulation

Index structure is needed to perform a query. The index structure can be used to retrieve documents from large hypertext collection. When index is used to retrieve documents of web page from hypertext collection, URLs of the pages are used as keys to construct the index. So, when a URL of a page is given as an input, document of that page is retrieved from data file using index. URLs of pages are different in size. That is, keys that are used to construct the index are of variable length. So, index structure that supports variable length keys is needed. After retrieving a document, we can do several further analyses on that document. To retrieve a document from a large data set, we need to know information like name of the document, where that document lies in the data set, what is the size of the document and so on. All these kind of information should be incorporated in the index.

For large collection of records, the index itself is so voluminous that only rather small parts of it can be kept in memory at one time. Thus the bulk of the index must be kept on some backup store. Since the data file itself changes, it must be possible not only to search the index and to retrieve elements, but also to delete and to insert keys. Multilevel indexes can be used to solve this problem.

There are different types of data structures that can be used as multilevel indexes. Choosing the appropriate data structure depends on the type of problem at hand. In multilevel indexes keys are stored in disk pages called nodes. When we construct an

index to retrieve documents from a hypertext collection, we store keys (URL of the document) and satellite information associated with the key in the index. Here, the satellite information associated with the key is the starting point of the document in the data file and size of the document. When a document is to be retrieved from the data file, first, a search is performed on the index and document is retrieved from the data file using the satellite information. Where to store the satellite information in the index is crucial one - store along with the key in the same node or store in separate node. If we store this information in separate node, then the key node contains pointer to the node containing satellite information. Since keys and its associated information are stored separately, more keys can be stored in key nodes. However, searching process requires more index access. First, we search for the key in the index. When a key is found in the node, we then follow the pointer to the node containing the satellite information. This takes more time in searching. If we store the satellite information along with the key, we do not need extra node access to find the associated information. Since satellite information is stored with the key, it is found in the node where the key is stored. This results in fast search. However, there is another drawback also. Since satellite information is stored in the same node as the key, few keys can be stored in the node which ultimately increases the number of nodes in the index.

If the satellite information is large in size, storing it with key is not feasible. This is because only few keys can be stored in a node. In this case, it is better to store associated information in separate node and store the pointer to this node in key node. But, if size of satellite information is very small in comparison to the size of key, it is better to store the information along with the key in the same node. So far as the associated information relating to the document retrieval system for large hypertext collection is concerned, it is the starting point of the document in the data file and its size in bytes. The size of this information is small compared to the size of the URL of the document. So, in this case, it is good to store this information in the same node as the key (URL). This information associated with a key travels with the key whenever the key is moved from node to node. To achieve these features, B-tree is the good multilevel index to choose. B-tree is a variation of search tree, which is a kind of tree data structure.

2.3 Tree Data Structure

A tree is a widely used data structure that emulates a tree structure with a set of linked nodes. Some terminologies used in tree data structure are

Nodes: A node may contain a value or a condition or represents a separate data structure or a tree of its own. Each node in a tree has zero or more **child nodes**, which are below it in the tree. A node that has a child is called the child's **parent node** (or ancestor node, or superior). A node has at most one parent. The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its root path).

Root Nodes: The topmost node in a tree is called the **root node**. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin. All other nodes can be reached from it by following edges or links. In diagrams, it is typically drawn at the top. Every node in a tree can be seen as the root node of the subtree rooted at that node.

Leaf Nodes: Nodes at the bottommost level of the tree are called leaf nodes. Since they are at the bottommost level, they do not have any children.

Internal Nodes: An internal node is any node of a tree that has child nodes and is thus not a leaf node.

Subtrees: A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below it, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

2.4 Search Trees

A search tree is a special type of tree data structure that is used to guide the search for a record, given the value of one of the record's fields. A search tree of order P is a tree such that each node contains at most p-1 search values and p pointers in the order < P_1, K_1, P_2, K_2,……., P_{q-1}, K_{q-1}, P_q >, where q <= p, each P_i is a pointer to a child node (or a NULL pointer); and each K_i is a search value from some ordered set of values. All search values are assumed to be unique. The two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.
2. For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.

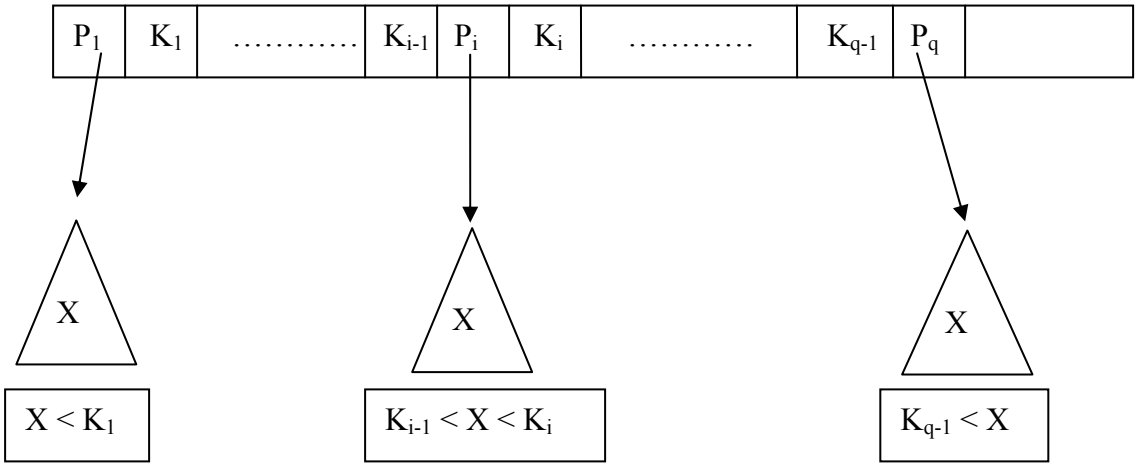


Fig 2.1 A Search tree

Whenever we search for a value X, we follow the appropriate pointer P_i . Some of the pointers P_i in a node may be null pointers. The values in the tree can be the values of one of the fields of the file, called the search field. Each key value in the tree is associated

with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. When a new record is inserted, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

In general, algorithms for inserting and deleting search values into and from the search tree do not guarantee that a search tree is balanced, meaning that all of its leaf nodes are at the same level. Keeping a search tree balanced is important because it yeilds a uniform search speed regardless of the value of the search key. Another problem with search trees is that record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

2.5 B-tree

A B-tree **T** is a rooted tree (whose root is *root* [T]) having the following properties:

1. Every node x has the following fields:

   a. n[x], the number of keys currently stored in node x,
   b. the n[x] keys themselves, stored in non-decreasing order, so that $key_1[x] <= key_2[x] <= \ldots\ldots. <= key_{n[x]}[x]$,
   c. *leaf* [x], a Boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains n[x] +1 pointers $c_1[x]$, c2[x]… $cn_{[x]+1}[x]$ to its children

3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

$$k_1 <= key_1[x] <= k_2 <= key_2[x] <= \ldots.. <= key_{n[x]}[x] <= k_{n[x]+1}$$

4. All leaves have the same depth, which is the tree's height h

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer t >=2 called the minimum degree of the B-tree:

a. Every node other than the root node must have at least t-1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
b. Every node can contain at most 2t-1 keys. Therefore, an internal node can have at most 2t children. We say that a node is full if it contains exactly 2t-1 keys.

Height of a B-tree

For *n* greater than or equal to one, the height of an *n*-key b-tree T of height *h* with a minimum degree *t* greater than or equal to 2,

$$h \leq \log_t \frac{n+1}{2}$$

The worst case height is *O (log n)*. Since the branchiness of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures.

2.5.1 Basic operations on B-trees

Basic operations on B-trees are ***create***, ***search*** and ***insert***. The algorithms for B-tree operations are single pass. In other words, they do not traverse back up the tree. Since b-trees strive to minimize disk accesses and the nodes are usually stored on disk, this single-pass approach will reduce the number of node visits and thus the number of disk accesses.

2.5.1.1 Search

Searching a B-tree is much similar to searching a binary search tree. In binary search tree we make a two way decision at each node. But in B-tree, we make a multi-way branching decision according to the number of the node's children. i.e. at each internal node x, we make an (n[x] +1) – way branching decision.

B-tree search is a simple procedure. It takes as input a pointer to the root node *x* of a subtree and a key *k* to be searched for in that tree. If *k* is in the B-tree, the search procedure returns the ordered pair (y, i) consisting of a node y and index i such that $key_i[y] = k$. otherwise it returns a NULL value.

SEARCH (*x*, *k*)

```
i ← 1
while i ≤ n[x] and k > key_i[x]  // compares k with the keys of x
    do i ← i + 1
if i ≤ n[x] and k = key_i[x]   // key found
    then return (x, i)     // returns the index
if leaf[x]    // if k is not found in x and x is leaf node then search is unsuccessful
    then return NIL
    else
  // if k is not found in x and x is not leaf node then search in next node
        return Search (c_i[x], k)
```

2.5.1.2 B-tree Create

To build a B-tree, we first create an empty root node and then call B-tree insert operation to add new keys. To create a new node, we use allocate node procedure which is used by both B-tree create and insert procedures. Allocate node procedure allocates a node to be used as a new node.

```
B-TREE-CREATE (T)
        x ← ALLOCATE-NODE () // Allocates a new node
        leaf[x] ← TRUE  // Node x is marked as leaf node
        n[x] ← 0  // Number of keys is zero
        Root[T] ← x  // x is assigned the root node of the tree
```

The B-TREE-CREATE operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node. Only the root node is permitted to have these properties. All other nodes must meet the criteria outlined previously.

2.5.1.3 Inserting a key

Inserting a key into a B-tree is more complicated than to insert a key into a binary search tree. We search for the leaf position at which to insert the new key. However, we can not simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree. Instead, we insert the new key into an existing leaf node. Since we can not insert a key into a leaf node that is full, we use an operation that splits a full node y around its median key $key_t[y]$ into two nodes having t-1 keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is also full, it must be split before the new key can be inserted, and thus this need to split full nodes can propagate all the way up the tree.

```
INSERT (T, k)

r ← root [T]

if n[r] = 2t – 1

   then s ← ALLOCATE-NODE()

            root [T] ← s

  leaf [s] ← FALSE

  n [s] ← 0

  c1 [s] ← r
```

SPLIT-CHILD (s, 1, r)

INSERT-NONFULL (s, k)

else INSERT-NONFULL (r, k)

If the root node is full, we split the root node and a new node s becomes the root. Splitting the root causes the height of B-tree to be increased by one. The procedure finishes by calling INSERT-NONFULL to perform the insertion of key k in the tree rooted at the nonfull root node. INSERT-NONFULL recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling SPLIT-CHILD as necessary. The procedure INSERT-NONFULL inserts key k into node x, which is assumed to be nonfull when the procedure is called.

INSERT-NONFULL (x, k)
$i \leftarrow n[x]$

if leaf[x]

then while $i \geq 1$ and $k < key_i[x]$

do $key_{i+1}[x] \leftarrow key_i[x]$

$i \leftarrow i - 1$

$key_{i+1}[x] \leftarrow k$

$n[x] \leftarrow n[x] + 1$

else while $i \geq 1$ and $k < key_i[x]$

do $i \leftarrow i - 1$

$i \leftarrow i + 1$

if $n[c_i[x]] = 2t - 1$

then SPLIT-CHILD (x, i, $c_i[x]$)

if k > $key_i[x]$

then i ← i + 1

INSERT-NONFULL ($c_i[x]$, k)

When we go to insert a key into b-tree, we first find the appropriate node for the key and try to insert the key into that node. If the node is not full prior to the insertion, we simply insert the key into the node. However, if the node is full, the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node.

2.5.1.4 Splitting a node

When a new key is to be inserted in a full node, we split the node into two nodes. The median key goes up to the parent node and rest of the keys are evenly distributed to the two nodes. To split a full root, we first make the root a child of a new empty root node. When the root node is split, height of tree grows by one.

SPLIT-CHILD (x, i, y)

// x is non full internal node and i is the index such that $y=c_i[x]$ is a full child of x. y is the node being split

z ← ALLOCATE-NODE ()
leaf[z] ← leaf[y]
n[z] ← t – 1 // sets the number of keys in new node z
for j ← 1 to t – 1 // moves keys
do $key_j[z] \leftarrow key_{j+t}[y]$
if not leaf[y]
then for j ← 1 to t // readjusts pointers
do $c_j[z] \leftarrow c_{j+t}[y]$
n[y] ← t – 1 // sets the number of keys in node y
// readjusts the pointers of root node to make node z child of x
for j ← n[x] + 1 **downto** i + 1

do $c_{j+1}[x] \leftarrow c_j[x]$

$c_{i+1} \leftarrow z$

for j ← n[x] **downto** i

do $key_{j+1}[x]$ <- $key_j[x]$

$key_i[x] \leftarrow key_t[y]$ // inserts the middle key of y into the root node x

n[x] ← n[x] + 1 // resets the numer of keys in the root node

Here, y is the *i*th child of x and is the node being split. Since node y is full, it originally has 2t-1 keys but is reduced to t-1 keys by this operation. Largest (t-1) keys goes into the node z, and z becomes a new child of x. The median key of y moves up to become the key in x that separates y and z.

2.5.1.5 Deleting a key

Deleting a key from a B-tree is more complicated than inserting a key into a B-tree. Keys can be deleted from any node. This can be a leaf node or an internal node. When a key is deleted from an internal node, we need to rearrange the children of this node. As in insertion, we must guard against deletion producing a tree whose structure violates the B-tree properties. We must ensure that a node does not get too small during deletion. Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node along the path to where the key is to be deleted has the minimum number of keys.

Deletion can be performed by finding the key to be deleted and removing it. If this key is one of the only t-1 keys in a node, then its removal violates the B-tree properties. We can fix this by combining this node with a sibling. If the sibling has t keys, we can take one and have both nodes with t-1 keys. If the sibling has only t-1 keys, we combine the two nodes into a single node with 2t-1 keys. The parent of this node now loses a child. So, we percolate this strategy all the way to the top. If the root loses its second child, then the root is also deleted and the tree becomes one level shallower. As we combine nodes, we must update the information kept at the internal nodes.

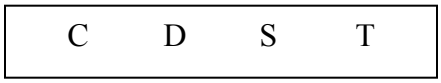
2.5.2 Creating a B-tree with a sample key sequence

Here is an example of creating a B-tree for the given key sequence. The sequence is:

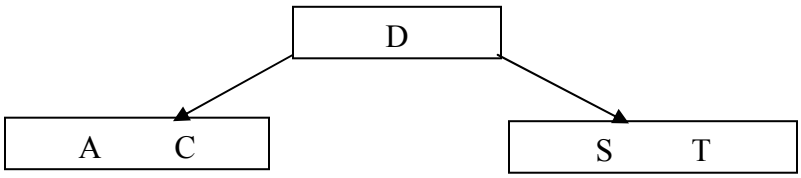
C S D T A M P I B W N G U R K E H O L J

Here, we are assuming a B-tree of order 4 (maximum of four key-reference pairs per node).

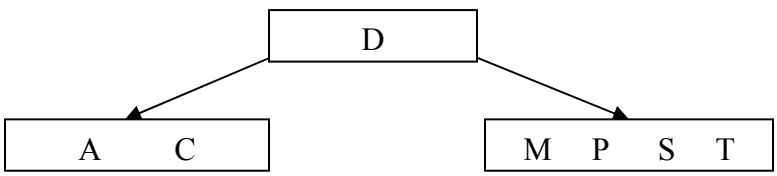
a) Inserting first four keys into an initially empty B-tree



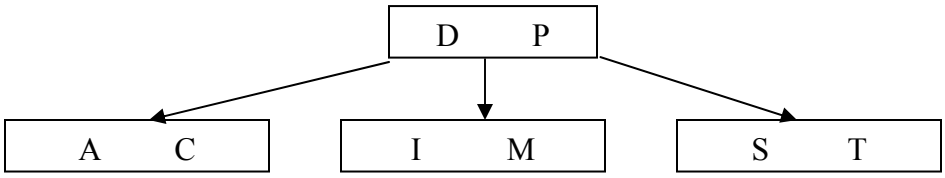
b) Insertion of **A** causes the root node to split and height of the tree grows by one



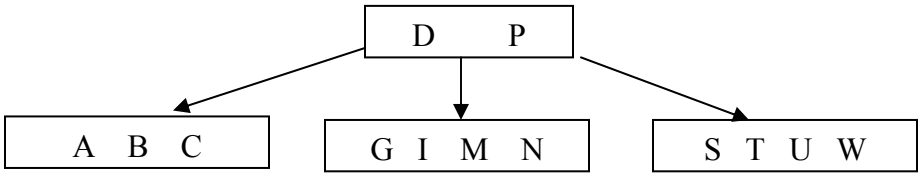
c) Inserting keys **M** and **P** in the above tree



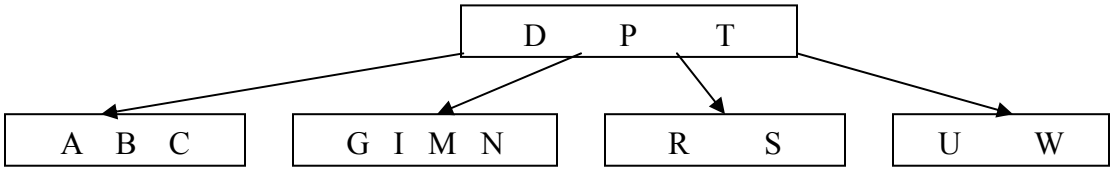
d) Insertion of **I** in the above tree causes the leaf node to split



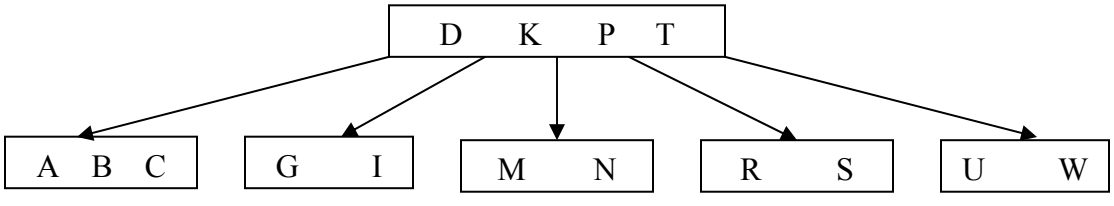
e) Inserting keys **B W N G U** in the above tree



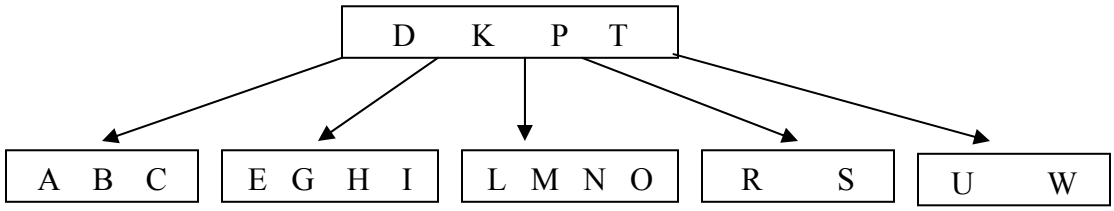
f) Insertion of key **R** in the above tree causes the leaf node to split



g) Insertion of key **K** in the above tree causes the leaf node to split



h) Inserting keys **E H O** and **L** in the above tree



i) Insertion of **J** in the above tree causes the leaf node to split which in turn causes the root node to split and height of the tree grows by one

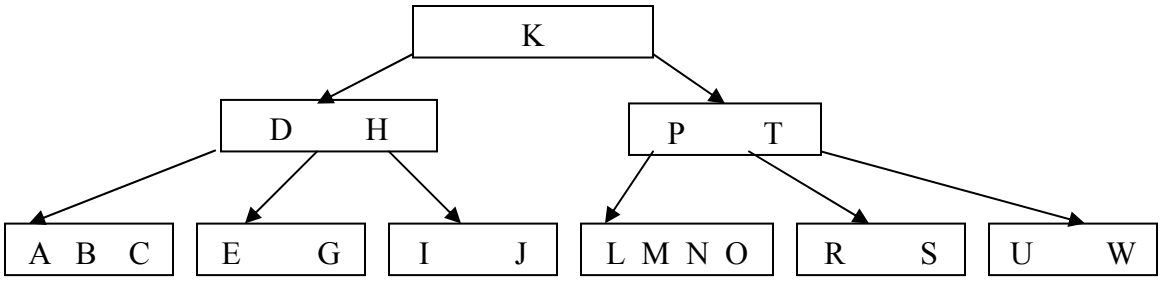


Fig 2.2 Inserting keys into Btree

2.6 Record format

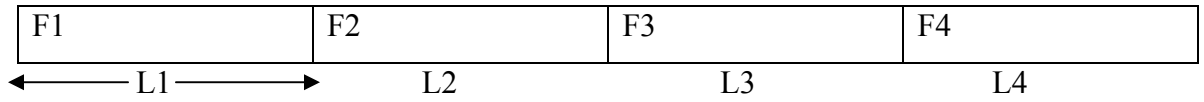
A record can be defined as a set of fields that belong together when the file is viewed in terms of a higher level of organization. There are two types of record formats

a) Fixed length records and
b) Variable length records

2.6.1 Fixed length records

In a fixed length record, each field has a fixed length (that is, the value in this field is of the same length in all records) and the number of fields is also fixed. The fields of such a

record can be stored consecutively, and, given the address of the record, the address of a particular field can be calculated using information about the lengths of preceding fields.



Fi = Field i

Li = length of field i

Fig 2.3 Fixed length record format

2.6.2 Variable length records

If the number of fields is fixed, a record is of variable length only because some of its fields are of variable length. Two commonly used methods for handling variable length records are:

Using array of field offsets

A simple method is to reserve some space at the beginning of a record for use as an array of integer offsets. The i^{th} integer in this array is the starting address of the i^{th} field value relative to the start of the record. An offset to the end of the record is also stored here.

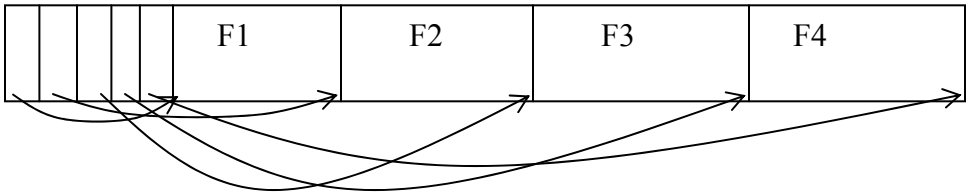


Fig 2.4 Variable length record format - Array of field offsets

Place a delimiter at the end of each record

Another method to deal with variable length records is to store fields consecutively, separated by delimiters. Delimiters are special characters that do not appear in the data

itself and must not get in the way of processing. This organization requires a scan of the record to locate a desired field.

F1	$	F2	$	F3	$	F4	$

Fig 2.5 Variable length record format - fields delimited by $

2.7 B tree with variable length keys

In many applications information associated with a key varies in length. One way to handle this variability is to place the associated information in a separate, variable length record file. The B-tree would contain a reference to the information in this other file. Another approach is to allow a variable number of keys and records in a B-tree page.
The definition of B-tree assumes the tree having some order m. Each page has a fixed maximum and minimum number of keys that it can legally hold. The algorithms for B-tree operations discussed above assume a B-tree of some order t. The maximum number of keys a node can contain is fixed. When a new key is to be inserted into a full node, it needs to be split. The condition for splitting a node depends on the number of keys on the node. That is, splitting a node is performed by counting the number of keys in the node.

The notion of a variable-length record and, therefore, a variable number of keys per page is a significant departure from the definition of B-tree we are using. A B-tree with a variable number of keys per page clearly has no single, fixed order. Using variable length key, we might choose to use only as much space as required for a key rather than allocate a fixed size field for each key. Implementing a structure with variable length fields allows us to put many more keys in a given amount of space. If we can put more keys in a page, then we have a larger number of descendents from a page and very probably a tree with fewer levels. Accommodating this variability in key length means using a different kind of criterion for deciding when a page is full and when it is an underflow condition. Rather

than use a maximum and minimum number of keys per page, we need to use a maximum and minimum number of bytes.

2.7.1 Algorithm Design

Designing algorithm for variable length key Btree requires a slight departure from the general definition of Btree. In a Btree, condition to split a node is already defined on the number of keys a node can hold. So, when a new key is to be inserted in a node, we count the number of keys in a node. If it is equal to the maximum number of keys a node can hold, we split the node into two and insert the key into the appropriate one. If it is not equal to the maximum number of keys, we insert the key into this node in appropriate position. This method works fine if all the keys are of the same size.

If keys are variable in length, it is better to check the free available space in a node before a key is inserted into the node. If there is enough space in the node, we insert the key into the node otherwise we split the node. To do this, every time when a new key is to be inserted into a node we need to compute the size of the key and the available free space in the node. Hence condition to split a node is based on the availability of the free space for new key rather than on the number of keys. Algorithms for inserting a key and splitting a node are outlined below.

2.7.1.1 Insertion algorithm

Let **NodeKeySize** be the maximum size of keys in a node and **nkey** is the number of keys in the node. **NodeKeySize** is same for all the nodes in the tree. **K** is the key to be inserted in node **N**. **KeySize** is the size of all the keys in a node. The insertion algorithm is as:

```
KeySize = 0
if (nkey == 0)
        insert K in N and increase nkey by 1
else
        for (i = 1; i <= nkey; i++)
                // key1, key2…are keys in node N
```

KeySize = **KeySize** + sizeof (key_i)

if (sizeof (**k**) + **KeySize** > **NodeKeySize**)

Split **N**

else

insert **K** in **N** in proper position by comparing keys and increase **nkey** by 1

2.7.1.2 Splitting a node

A node **N** is split if there is not enough space to hold the new key. When a node is full, we split the node into two. The middle key goes into the parent node. With fixed length key Btree, nodes always have odd number of keys when nodes are full. So, keys are evenly distributed into two new nodes when a node is split. However, in variable length key Btree a node can have even or odd number of keys when a node is full. Therefore, we need a different kind of method to distribute the number of keys into two new nodes when a node is split. Algorithm to split a node is as:

if (parent == **NULL**)

// algorithm for root node

// allocate two new nodes

LeftNode = AllocateNewNode ()

RightNode = AllocateNewNode ()

midposition = nkey/2

move all the keys lower than key in the midposition into LeftNode

move all the keys higher than key in the midposition into RightNode

// sets the number of keys in the left node

nkey (LeftNode) = midposition

// sets the number of keys in the right node

if (nkey%2 == 0)

nkey (RightNode) = midposition – 1

else

```
            nkey (RightNode) = midposition
        // resets the number of key in the current node which is new root node
        nkey = 1
        // sets the parent of left and right nodes
        parent (LeftNode) = N
        parent (RightNode) = N
else
        // algorithm for non-root nodes
        midposition = nkey/2
        newkey = key in the midposition of N
        parentsize = size of all the keys in the parent
        if (sizeof (newkey) + parentsize > NodeKeySize of parent)
                split parent
        insert newkey in the parent in proper position by comparing keys.
        RightNode = AllocateNewNode ()
        move all the keys higher than newkey into the RightNode
        // sets the number of keys in the newly allocated right node
        if (nkey%2 == 0)
                nkey (RightNode) = midposition – 1
        else
                nkey (RightNode) = midposition

        // resets the number of keys in node N
        nkey = midposition
        //sets the parent of newly created right node which is same as the parent of N
        parent (RightNode) = parent
```

CHAPTER 3: IMPLEMENTATION

The implementation of variable length key BTree is done using Java, there are several class APIs created to represent the data structure and BTree operations. We can set the total key size of the node. Before inserting a key into a node, program checks the available space in the node to test whether the node can accommodate the key. We can give large values for the total key size of node. If we give large value for total key size of the node, more keys can be packed into the node and thus reduces the height of the tree. A brief description of classes and methods are given below:

3.1 Interfaces

Comparator: This interface defines a method called compareTo, any key object which is supposed to be inserted in the btree must implement this interface and provide implementation, and BTree methods ensure that the Keys can be compared using compareTo method.

3.2 Class Description

Class StringComparator: Implements the Comparator interface and provides details for compareTo, the compareTo method compares the string (property of this class) using the String class's compareTo method. Objects of this class are used as keys to be inserted in the BTree.

Class KeyObject: This is a simple class which represents the data object for a key. KeyObject class has attributes like URL, start position of the record in the content (input) file and size of that record. Object of this class is inserted along with the key in a btree insert operation.

Class SearchResult: Objects of this class represent a search result, it contains a reference to a Node and an index of the key found in that node.

Class BTree: This is the main class representing the BTree, it contains a reference to the root node of the btree and methods for inserting a node and retrieving the keys. This class can be initialized using its one argument constructor which takes the maximum size of all the keys that can be inserted in a node. The size is the character length of a key. The method insertObj finds a leaf node in the tree where the new key can be inserted traversing from its reference to the root node. After inserting a node in the tree this method checks available space in the root node. If it cannot accommodate new key, it splits the root node. Keys are always inserted into the node so the methods for inserting key is implemented in a different class called BTNode.

- **insertObj** takes 2 arguments a Comparator (key) and an Object (data for that key), the Comparator object used in this project is StringComparator and associated data object is KeyObject, any type of object can be used as the data for a key. It finds the leaf node in the tree where the new key can be inserted. After inserting the key, this method checks the available space in the root node. If the root does not have enough space to hold new key, it splits the parent.

- **searchObj** is a recursive method which starts searching for a key in a btree from the root node, the key is compared to the every key nodes in a given node, if the search key is less or more in order with a key in the current node, searchObj is called recursively using the left or right node pointer of the current node. If the key is equal to any key in the node the search is success. If the node is a leaf node and the search key is greater than the last key in that node the search fails. This method takes 2 arguments as BTNode and Comparator, it returns an object of SearchResult Class which might contain a Node and a key index of the search key in that node if the result is a success else it contains a null and index of -1.

Class BTNode: This class represents node in a btree, it has members like current number of keys in the node and arrays to hold the Keys and References to the Child Nodes. isLeaf and parent properties in this node represent if the node is a leaf node and a reference to the parent node of the node respectively. If the node is a root node, parent references to a

null value. **BTNode** class has methods for finding the size of all the keys in a node, inserting a key in a node, splitting a node and shifting nodes in a given node.

- **getCurrentSize method** computes the total key length of a node N. It does this by calculating the size of all the keys in the node and adding this. When a new key is to be inserted into node N, the value returned by this method is used to check whether the node N can accommodate the new key or not.

- **The insert method** checks whether a new key can be inserted by checking the current total key length in that node and splits the node if it cannot be accommodated. It compares the new key with all the keys in the node (where the new key is to be inserted) to find the insertion point. It then computes the total length of keys in the node. This length is added with the length of new key. If the total size is less than the node size, it inserts the new key in proper position in the node. If the total size is greater than the node size, it splits the node by allocating a new node as the right node of the parent. The new key is inserted into the new node or the current node (which becomes left node of the parent)

- **Split method** splits a given node into two nodes. This can only be done if the node is full. It splits a node by finding the middle positioned key and creating a new node as its parent's right node. The middle positioned key goes up into the parent, the left ones of them rest in this node and the right ones go into the new node. If the node is a root node, 2 new child nodes (left and right) are created. The middle positioned key remains in this node. All the keys to the left of the mid key go into the left node and the keys to the right of the mid key go into the right node. This node now becomes the root node. There is a check for this condition in this method. This method is called recursively if the parent node in context also requires split before adding the middle key to it.

- **Shifting** is done when a key is required to be inserted between the existing keys, the new key is inserted at the correct position after shifting all the keys in the right side of that position to the right direction by one position.

Class KeyNode: Object of this class represents key nodes in a btree node; it contains 2 members as a Key and a KeyObject. The type of Key is Comparator and type of KeyObject is Object. We are using **StringComparator** as a Comparator and **KeyObject** as an Object.

Class BTreeEnumeration: This class creates an enumeration of btree keys. It implements Enumeration interface and has a method called getNext which returns the next key in the btree sorted by key.

- Constructor **BTreeEnumeration** takes two arguments as BTree and boolean getKeys. It sets the current node of enumeration to root node of btree passed and calls getNext method which prepare the first element to be served in nextElem.

- **getNext method** gives the next element (key) in btree maintaining a currentNode and a currentItem (position in the node). When this method is called for the first time, it goes all the way to the left most leaf to get the first element and returns it, and then increases the currentItem position in that node. The currentNode is changed to this node (initially the current node was set to root node). It returns the next element from the currentNode until it reaches the end of it. After then

    - the currentNode is added to a vector called nodesSeen
    - the currentNode is assigned the parent node of it
    - all the Childs of the currentNode (which is the parent of previous node) are examined whether they are already added to nodesSeen, which gives the location of the key in the node to be returned. Now the next unseen node is assigned as the currentNode.

The method always checks for the next element and assigns false to moreElements variable if the current element returned was the last element. This is set if the currentNode is root and the position (currentItem) is greater than the keys in the root node.

- **hasMoreElements method** checks if there is more elements in the tree.

- **nextElement method** returns a previously retrieved next element and sets a new next element

class Main: This class drives the program and is executed from command line; this class takes some command line parameters like input file and the operation to be performed, the operations are, 1=search a key , 2=scan btree in sorted order of keys, 3=prints the nodes of the btree. Methods used in this class are:

- **getContentLength method** finds the size of the content of the URL from within a string passed to it. Size of document of every page is given in the data file. The method just extracts this value from the input file (data file). This value is used to read the contents of the page (URL) from the data file at the time of search.

- **getContentStartPosOff method** computes the start position offset of every page (URL) in the data file. This value is used to read the content of the page from the data file at the time of search.

- **getURL method** strips the URL from the string passed to it. The URL is the key and index is constructed based on this field. When a search is to be performed, URL (key) is given as a search string. Keys are variable in length.

- **usedMemory method** calculates the total memory allocated. It first computes the memory allocated by different objects. It then runs garbage

collector to free the memory used by the objects which do not have references. Finally it calculates the total actual memory allocated by taking difference of total memory and free memory. This function is called two times in the program to find the size of the Btree index in memory - before and after the construction of Btree.

- **trimSizes method** is a recursive function that recreates every node of Btree by copying all the keys and the pointers into new node. This function is used to calculate the actual size of the index in memory. Initially, key array and pointer array of all the nodes are initialized to same value. But during index construction, number of keys and pointers in different nodes may be different. So, to calculate the actual index size, we need to trim the key array and pointer array. To do this, this method makes use of two different functions **trimArray** and **trimArray1**.

- **trimArray method** starts from root and is called recursively until leaf node is found. It takes a node and number of pointers in the node as input and returns a new pointer array by copying all the pointers into this new array.

- **trimArray1 method** starts from root and is called recursively until leaf node is found. It takes a node and number of keys in the node as input and returns a new key array by copying all the keys into this new array.

- **printNodes method** prints the node of btree using preorder traversal and the method **printNode** prints the content of the node, its key, level and id.

- **printContent method** grabs the record content from the input file using information from key object passed to it. This makes use of the values returned by the methods **getContentLength** and **getContentStartPosOff**.

- **searchKey method** reads the input key to search from command line and uses Btree's **get** method to perform search. It returns null if key is not found in the tree. If key is found, it returns a KeyObject.

- **createBtree method** creates a btree from an input file. It uses random access file to seek the content and makes use of getContentLength, getContentStartPosOff and getURL methods. It inserts keys into btree as they are read from input file and returns a btree object.

3.3 Input File

The input file is **Ask.com** data file. The file contains URLs of web pages along with their documents (HTML code). The single file contains documents of more than 100 web pages. The input file taken for the testing of this program is "bin0.Crawler0-.13.00000013.dat". This file contains 796 web pages with their HTML code.

There is some information regarding each web page in the input file. Before HTML code of every web page, there is indication of start of the page, size of the page in bytes, time of crawl of the page and the URL of the page. For example-

<>RECORD<> 6134 2006-01-30/17:09:06 http://www.graystonelearning.com/
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>…….
…….
</HTML>

Here, **<>RECORD<>** indicates start of the page, the value 6134 is size of the page in bytes, 2006-01-30/17:09:06 is time of crawl of the page and "http://www.graystonelearning.com/" is the URL of the web page. After this page there is again this type of information for the next page in the input file. For this work, the time of

crawl is of no interest. The implementation of this work makes use of rest of the information. i.e. starting point, size in bytes and URL.

3.4 Program Flow

The program starts by reading the input file specified; it uses RandomAccessFile to seek the contents. Information about all the records such as key name, start position of the content and size of the content are stripped from the file and they are used to create Key and KeyObject objects. An instance of BTree class is created and keys (along with KeyObject) are inserted into BTree as they are read from the input file.

The command line option (1=search, 2=scan, 3=print nodes) is checked and related operation is performed. Search makes use of searchObj method, scan uses an Enumeration implementation of Keys, and Print nodes prints all the nodes in btree in preorder along with their keys, the nodes are printed with a node id tag, their level in btree, and node id of the parent node.

CHAPTER 4: TESTING AND OUTPUT

The program is tested with **Ask.com** data file. The file used for testing is "bin0.Crawler0-.13.00000013.dat". There are 796 web pages (URLs) in this file. The program constructs the index of this file. Keys used to create the index are the URLs of pages. When URL of webpage is given as a search key to the program, it searches the key in the index. If the key is found, it displays the contents of the page. The program also counts the number of nodes accessed to find a key. Program reads the contents of the page from the data file. To test the correctness of the program, various searches with different URLs have done and it always produces the correct output. Some output from the program are:

// Test for successful search

```
C:\btree>java Main input.dat 1
Memory Heap=1688 KB
Enter key to search:www.hogarmrsannicolas.org.ar/
**<>RECORD<>** 704 2006-01-30/17:09:54 http://www.hogarmrsannicolas.org.ar/
<html>
<head>
<title>Asociaci ├ │ n Civil Mar ├¡a del Rosario de San Nicol ├ís</title>
</head>
<frameset cols="210,*" framespacing="0" border="0" frameborder="0">
  <frame name="contents" target="main" src="contents.htm" marginwidth="0" margin
height="0" scrolling="no" noresize>
  <frame name="main" src="main.htm" marginwidth="0" marginheight="0" scrolling="
auto" noresize>
  <noframes>
  <body topmargin="20" leftmargin="20">
  <p><img border="0" src="iamges/hogar-grande.gif"></p>
  <p><font color="#FF0000"><b>Esta p ├ígina soporta frames, pero su Navegador no
los soporta.</b></font></p>
  </body>
```

```
  </noframes>
</frameset>
</html>

Found at level 3

C:\btree>java Main input.dat 1
Memory Heap=1688 KB
Enter key to search:www.graywest.com/
**<>RECORD<>** 383 2006-01-30/17:09:47 http://www.graywest.com/
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
      <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <TITLE>graywest.com</TITLE>
</HEAD>
<FRAMESET rows="100%,*" border=0 frameborder=0 framespacing=0>
      <FRAME name=top src="http://12.203.219.126/gray/index.htm" noresize>
</FRAMESET>
</HTML>

Found at level 3

C:\btree>java Main input.dat 1
Memory Heap=1688 KB
Enter key to search:global.proximity.on.ca/
**<>RECORD<>** 537 2006-01-30/17:09:27 http://global.proximity.on.ca/
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE html PUBLIC
      '-//W3C//DTD XHTML 1.0 Transitional//EN'
      'DTD/xhtml1-transitional.dtd'>
```

```
<html>
    <head>
        <title>
            Global Proximity Corporation
        </title>
    </head>
    <body bgcolor="WHITE">
        <center>
            <img src="graphics/gp_globe.gif" alt="" />
            <br clear="all" />
            <p>
            <font face="Helvetica,Arial,Swiss">
            Global Proximity Corporation<br />
            info (at) global.proximity.on.ca
            </font>
            </p>
        </center>
    </body>
</html>
```

Found at level 2

// Test for unsuccessful search

```
C:\btree>java Main input.dat 1
Memory Heap=1688 KB
Enter key to search:www.iiitb.ac.in
Key Not Found
```

// Test for unsuccessful search

```
C:\btree>java Main input.dat 1
Memory Heap=1688 KB
```

Enter key to search:www.nepalnews.com

Key Not Found

// Tree scan. It shows all the keys in the tree with their starting offset and size in the data file. Output shown here is only a portion of the complete output.

C:\btree>java Main input.dat 2 > tscan.doc

```
Memory Heap=1688 KB
7741622:    6038: fsu.campusrec.com/crenshaw.html
4373628:    32871:      gallery.aut.ac.nz/album05
1845592:    38937:      gallery.baz.org/album45
4054440:    11293:      gallery.derosia.org/Awards
6146051:    16446:      gallery.ittefaq.com/nation/pic9
4949725:    28279:      gallery.johny.sk/bp
1912996:    19290:      gallery.kinfolk.org/album96
4416382:    50253:      gallery.peimic.com/ebay
9453039:    35107:      garage.docsearls.com/node/view/503
6906378:    14850:      garage.docsearls.com/node/view/504
4611752:    16524:      garage.docsearls.com/node/view/505
2448219:    17142:      garage.docsearls.com/node/view/508
41004:      16137:      garage.docsearls.com/node/view/509
913832:     2100:
      genealogy.bgwoodruff.com/reunionweb/ps01/ps01_184.html
3118685:    4729: genealogy.harmonicflight.net/legacy/1310.htm
6008685:    3959: genealogy.harmonicflight.net/legacy/1818.htm
3947293:    3691: genealogy.harmonicflight.net/legacy/2044.htm
7245636:    4766: genealogy.harmonicflight.net/legacy/4530.htm
6728891:    56570:      ghana.info-pagina.com/
6531961:    3557: ghettosrise-records.com/
6991517:    34038:      ghostwhowalksinside.org/
5783728:    52361:      giants.realfootball365.com/
5875622:    18785:      gillian.simplenet.com/
7799374:    427:  gimel.esc.cam.ac.uk/
5741246:    21536:      girl.tornpaperbag.com/
4256201:    5679: glass.mallettantiques.com/
4133968:    33489:      glayzed_honey.paxed.com/
3752263:    2745: glibersat.linux62.org/
6922381:    13756:      global.finland.fi/rekry/linkit.htm
3390252:    556:  global.proximity.on.ca/
5588649:    712:  globaldomain.saax.com/
4336752:    36876:      globefund.manulife.ca/
3224615:    16403:      glossa.fltr.ucl.ac.be/

3162480:    409:  gmso.linux-sevenler.org/
2778151:    8475: goldenservicesllc.com/
2228092:    7245: goldsgym.pretzelman.com/
167232:     25514:      golf.gfccnet.org/
2465361:    6632: goodenoughsprings.org/
1911264:    1634: gopublicity.cognigen.org/
2090793:    18786:      gossamer.simplenet.com/
6187547:    9579: gotigers.obu.edu/baseball/Stats/Obuac1.htm
3134963:    27517:      gr.fipu.krasnoyarsk.edu/
241361:     3305: greatercommunities.com/
```

```
459381:     1283: grefo.ishavingamassage.com/
468755:     11977:      gremio.ccdc.cam.ac.uk/
752588:     16837:      griffel.vabhosting.com/
2335431:    1390: groenlinks.westbrabant.net/
526549:     1180: groothuizen.babbelt.nl/
716856:     35732:      groups.montrosetravel.com/
958780:     5339: gruposdefe.marianistas.org/
509999:     1622: grzegorz.cognigen.org/
2474435:    24724:      guiden.guildportal.com/
9703922:    1541: gunslinger.codenamenetwork.com/
7605919:    77:   guy.forclark.com/
9619328:    13012:      hagens-berman.vabhosting.com/
10028115:   12765:      hairfertilizer.qcommerce.com/
10317877:   45900:      handbag.propertyfinder.co.uk/
5323152:    11920:      happyblue.net/gallery/1996
9078533:    639:  happyhome.ownanewbusiness.com/
468675:     80:   hardly.forclark.com/
10048731:   5402: hastingtrailers.yorke.net.au/
3957280:    80:   hawker.forclark.com/
9655806:    14855:      healthinfo.health-first.org/advantage/DTFAP.htm

7988522:    10627:      help-desk-analyst.info-all.org/
9198553:    21045:      helpdesk.readysetconnect.com/
8628997:    17908:      heroesblood.guildportal.com/
7261066:    11128:      higher-education.info-all.org/
8614663:    14233:      hitachilabparts.jmscience.com/
7185171:    21669:      hkretirement.axarosenberg.com/
7976253:    12269:      home.config.com/
6921340:    639:  homebased.ownanewbusiness.com/
7160982:    24189:      homeschoolresourceexchange.com/
2280316:    14584:
      www.answers.com/related%253Aactionfigures.about.com/library/press
/blebayholiday.htm
3552243:    16677:
      www.answers.com/related%253Achristianmusic.about.com/library/bl_g
eorgehuff_cmas.htm
2653701:    18493:
      www.answers.com/related%253Aentertaining.about.com/cs/etiquette/a
/thankyou.htm
1279271:    18511:
      www.answers.com/related%253Aentertaining.about.com/cs/etiquette/a
/thankyou_5.htm
1829828:    15764:
      www.answers.com/related%253Afamilycrafts.about.com/library/bdspin
s/blbdspinxmas.htm
10067148:   13640:
      www.answers.com/related%253Afootball.about.com/cs/football101/g/g
l_playbook.htm
5762878:    20753:
      www.answers.com/related%253Afootball.ballparks.com/NFL/OaklandRai
ders/oldindex.htm
9058955:    19578:
      www.answers.com/related%253Agoflorida.about.com/library/bls/bl_ph
oto_tampa14.htm
3171342:    19809:
      www.answers.com/related%253Ahumanresources.about.com/cs/recruitin
g/a/intjobapp.htm
```

5461562: 18793:
www.answers.com/related%253Alibrary.humboldt.edu/infoservices/quickref/music.htm
4713488: 15580:
www.answers.com/related%253Anascar.about.com/cs/schedulesraces/a/2005sched.htm
57141: 19807:
www.answers.com/related%253Anovi.lib.mi.us/aboutlib/calendar/bracelets.htm
9236103: 15213:
www.answers.com/related%253Awww.chem.latech.edu/%7Ededdy/Lectnote/Chap1Moore.html
4023503: 13115:
www.answers.com/related%253Awww.chipublib.org/008subject/005genref/giswedding.html
10257559: 13106:
www.answers.com/related%253Awww.ldonline.org/ld_indepth/math_skills/coopmath.html
7314997: 16280:
www.answers.com/related%253Awww.letsgodigital.org/en/news/articles/story_2175.html
664600: 13104:
www.answers.com/related%253Awww.mathcats.com/grownupcats/mathcatsnewsissue12.html
2884637: 20459:
www.answers.com/related%253Awww.nascar.com/races/wc/2003/data/schedule.html
5038296: 13140:
www.answers.com/related%253Awww.powercheerleading.com/UWO0405/UWOCheerMediaLinks.html
7939497: 19298:
www.answers.com/related%253Awww.sandbox.com/fantasysportsgames/aflsalarycapfootball.html
6936137: 16722:
www.answers.com/related%253Awww.scarletknights.com/football/history/hof-robeson.htm
8286304: 18654:
www.answers.com/related%253Awww.unc.edu/depts/nccsi/SurveyofFootballInjuries.htm
1606017: 13167:
www.answers.com/related%253Awww.usafootball.com/features/playerfeature_conindex.html
9903114: 13410:
www.answers.com/related%253Awww.vikings.com/news_detail_OBJECTNAME_2005DraftPC42405.html
6347843: 13162:
www.answers.com/related%253Awww.westarkchurchofchrist.org/benjamin/2004/041128am.htm
3080947: 11609:
www.arabicnews.com/BasicFacts/QATAR/government.html

// Prints keys in the node wise fashion along with their level and parent ID. It also shows number of keys in every node. Output shown here is only a portion of complete output.

Total key size of Node = 300 character length. With this size Number of nodes created = 130 and Level of tree = 3

C:\btree>java Main input.dat 3 >tree.doc

```
Memory Heap=1688 KB
============================================================
Node Id: 1(root) Parent Node Id:0 ,  #Keys:2     Level=0
Start=1606017      Size=13167
      Key=www.answers.com/related%253Awww.usafootball.com/features/play
erfeature_conindex.html
Start=374203       Size=42329
      Key=www.freedownloadscenter.com/Search/ocr.html
============================================================
Node Id: 2(internal) Parent Node Id:1 ,    #Keys:2     Level=1
Start=2280316      Size=14584
      Key=www.answers.com/related%253Aactionfigures.about.com/library/p
ress/blebayholiday.htm
Start=57141  Size=19807
      Key=www.answers.com/related%253Anovi.lib.mi.us/aboutlib/calendar/
bracelets.htm
============================================================
Node Id: 3(internal) Parent Node Id:2 ,    #Keys:9     Level=2

Start=1912996      Size=19290 Key=gallery.kinfolk.org/album96
Start=913832       Size=2100
      Key=genealogy.bgwoodruff.com/reunionweb/ps01/ps01_184.html
Start=5875622      Size=18785 Key=gillian.simplenet.com/
Start=3390252      Size=556   Key=global.proximity.on.ca/
Start=167232       Size=25514 Key=golf.gfccnet.org/
Start=468755       Size=11977 Key=gremio.ccdc.cam.ac.uk/
Start=958780       Size=5339  Key=gruposdefe.marianistas.org/
Start=10028115     Size=12765 Key=hairfertilizer.qcommerce.com/
Start=7988522      Size=10627 Key=help-desk-analyst.info-all.org/
============================================================
Node Id: 4(leaf) Parent Node Id:3 ,  #Keys:6     Level=3
Start=7741622      Size=6038  Key=fsu.campusrec.com/crenshaw.html
Start=4373628      Size=32871 Key=gallery.aut.ac.nz/album05
Start=1845592      Size=38937 Key=gallery.baz.org/album45
Start=4054440      Size=11293 Key=gallery.derosia.org/Awards
Start=6146051      Size=16446 Key=gallery.ittefaq.com/nation/pic9
Start=4949725      Size=28279 Key=gallery.johny.sk/bp
============================================================
Node Id: 5(leaf) Parent Node Id:3 ,  #Keys:6     Level=3
Start=4416382      Size=50253 Key=gallery.peimic.com/ebay
Start=9453039      Size=35107 Key=garage.docsearls.com/node/view/503
Start=6906378      Size=14850 Key=garage.docsearls.com/node/view/504
Start=4611752      Size=16524 Key=garage.docsearls.com/node/view/505
Start=2448219      Size=17142 Key=garage.docsearls.com/node/view/508
Start=41004  Size=16137 Key=garage.docsearls.com/node/view/509
============================================================
```

```
Node Id: 6(leaf) Parent Node Id:3 ,  #Keys:8     Level=3
Start=3118685      Size=4729
      Key=genealogy.harmonicflight.net/legacy/1310.htm
Start=6008685      Size=3959
      Key=genealogy.harmonicflight.net/legacy/1818.htm
Start=3947293      Size=3691
      Key=genealogy.harmonicflight.net/legacy/2044.htm
Start=7245636      Size=4766
      Key=genealogy.harmonicflight.net/legacy/4530.htm
Start=6728891      Size=56570 Key=ghana.info-pagina.com/
Start=6531961      Size=3557  Key=ghettosrise-records.com/
Start=6991517      Size=34038 Key=ghostwhowalksinside.org/
Start=5783728      Size=52361 Key=giants.realfootball365.com/
============================================================
Node Id: 7(leaf) Parent Node Id:3 ,  #Keys:6     Level=3
Start=7799374      Size=427   Key=gimel.esc.cam.ac.uk/
Start=5741246      Size=21536 Key=girl.tornpaperbag.com/
Start=4256201      Size=5679  Key=glass.mallettantiques.com/
Start=4133968      Size=33489 Key=glayzed_honey.paxed.com/
Start=3752263      Size=2745  Key=glibersat.linux62.org/
Start=6922381      Size=13756 Key=global.finland.fi/rekry/linkit.htm
============================================================
Node Id: 8(leaf) Parent Node Id:3 ,  #Keys:6     Level=3
Start=5588649      Size=712   Key=globaldomain.saax.com/
Start=4336752      Size=36876 Key=globefund.manulife.ca/
Start=3224615      Size=16403 Key=glossa.fltr.ucl.ac.be/
Start=3162480      Size=409   Key=gmso.linux-sevenler.org/
Start=2778151      Size=8475  Key=goldenservicesllc.com/
Start=2228092      Size=7245  Key=goldsgym.pretzelman.com/
============================================================
Node Id: 9(leaf) Parent Node Id:3 ,  #Keys:7     Level=3
Start=2465361      Size=6632  Key=goodenoughsprings.org/
Start=1911264      Size=1634  Key=gopublicity.cognigen.org/
Start=2090793      Size=18786 Key=gossamer.simplenet.com/
Start=6187547      Size=9579
      Key=gotigers.obu.edu/baseball/Stats/Obuac1.htm
Start=3134963      Size=27517 Key=gr.fipu.krasnoyarsk.edu/
Start=241361       Size=3305  Key=greatercommunities.com/
Start=459381       Size=1283  Key=grefo.ishavingamassage.com/
============================================================
Node Id: 10(leaf) Parent Node Id:3 ,       #Keys:4     Level=3
Start=752588       Size=16837 Key=griffel.vabhosting.com/
Start=2335431      Size=1390  Key=groenlinks.westbrabant.net/
Start=526549       Size=1180  Key=groothuizen.babbelt.nl/
Start=716856       Size=35732 Key=groups.montrosetravel.com/
============================================================
Node Id: 11(leaf) Parent Node Id:3 ,       #Keys:5     Level=3
Start=509999       Size=1622  Key=grzegorz.cognigen.org/
Start=2474435      Size=24724 Key=guiden.guildportal.com/
Start=9703922      Size=1541  Key=gunslinger.codenamenetwork.com/
Start=7605919      Size=77    Key=guy.forclark.com/
Start=9619328      Size=13012 Key=hagens-berman.vabhosting.com/
============================================================
Node Id: 12(leaf) Parent Node Id:3 ,       #Keys:7     Level=3
Start=10317877     Size=45900 Key=handbag.propertyfinder.co.uk/
Start=5323152      Size=11920 Key=happyblue.net/gallery/1996
Start=9078533      Size=639   Key=happyhome.ownanewbusiness.com/
```

```
Start=468675       Size=80    Key=hardly.forclark.com/
Start=10048731     Size=5402  Key=hastingtrailers.yorke.net.au/
Start=3957280      Size=80    Key=hawker.forclark.com/
Start=9655806      Size=14855 Key=healthinfo.health-
first.org/advantage/DTFAP.htm
============================================================
Node Id: 13(leaf) Parent Node Id:3 ,       #Keys:8     Level=3
Start=9198553      Size=21045 Key=helpdesk.readysetconnect.com/
Start=8628997      Size=17908 Key=heroesblood.guildportal.com/
Start=7261066      Size=11128 Key=higher-education.info-all.org/
Start=8614663      Size=14233 Key=hitachilabparts.jmscience.com/
Start=7185171      Size=21669 Key=hkretirement.axarosenberg.com/
Start=7976253      Size=12269 Key=home.config.com/
Start=6921340      Size=639   Key=homebased.ownanewbusiness.com/
Start=7160982      Size=24189 Key=homeschoolresourceexchange.com/
============================================================
Node Id: 14(internal) Parent Node Id:2 ,   #Keys:3     Level=2
Start=1279271      Size=18511
      Key=www.answers.com/related%253Aentertaining.about.com/cs/etiquet
te/a/thankyou_5.htm
Start=5762878      Size=20753
      Key=www.answers.com/related%253Afootball.ballparks.com/NFL/Oaklan
dRaiders/oldindex.htm
Start=3171342      Size=19809
      Key=www.answers.com/related%253Ahumanresources.about.com/cs/recru
iting/a/intjobapp.htm
============================================================
Node Id: 15(leaf) Parent Node Id:14 ,      #Keys:2     Level=3
Start=3552243      Size=16677
      Key=www.answers.com/related%253Achristianmusic.about.com/library/
bl_georgehuff_cmas.htm
Start=2653701      Size=18493
      Key=www.answers.com/related%253Aentertaining.about.com/cs/etiquet
te/a/thankyou.htm
============================================================
Node Id: 16(leaf) Parent Node Id:14 ,      #Keys:2     Level=3
Start=1829828      Size=15764
      Key=www.answers.com/related%253Afamilycrafts.about.com/library/bd
spins/blbdspinxmas.htm
Start=10067148     Size=13640
      Key=www.answers.com/related%253Afootball.about.com/cs/football101
/g/gl_playbook.htm
============================================================
Node Id: 17(leaf) Parent Node Id:14 ,      #Keys:1     Level=3
Start=9058955      Size=19578
      Key=www.answers.com/related%253Agoflorida.about.com/library/bls/b
l_photo_tampa14.htm
============================================================
Node Id: 18(leaf) Parent Node Id:14 ,      #Keys:2     Level=3
Start=5461562      Size=18793
      Key=www.answers.com/related%253Alibrary.humboldt.edu/infoservices
/quickref/music.htm
Start=4713488      Size=15580
      Key=www.answers.com/related%253Anascar.about.com/cs/schedulesrace
s/a/2005sched.htm
============================================================
Node Id: 19(internal) Parent Node Id:2 ,   #Keys:3     Level=2
```

```
Start=4023503      Size=13115
      Key=www.answers.com/related%253Awww.chipublib.org/008subject/005g
enref/giswedding.html
Start=664600       Size=13104
      Key=www.answers.com/related%253Awww.mathcats.com/grownupcats/math
catsnewsissue12.html
Start=5038296      Size=13140
      Key=www.answers.com/related%253Awww.powercheerleading.com/UWO0405
/UWOCheerMediaLinks.html
============================================================
Node Id: 20(leaf) Parent Node Id:19 ,      #Keys:1     Level=3
Start=9236103      Size=15213
      Key=www.answers.com/related%253Awww.chem.latech.edu/%7Ededdy/Lect
note/Chap1Moore.html
============================================================
Node Id: 21(leaf) Parent Node Id:19 ,      #Keys:2     Level=3
Start=10257559     Size=13106
      Key=www.answers.com/related%253Awww.ldonline.org/ld_indepth/math_
skills/coopmath.html
Start=7314997      Size=16280
      Key=www.answers.com/related%253Awww.letsgodigital.org/en/news/art
icles/story_2175.html
============================================================
Node Id: 22(leaf) Parent Node Id:19 ,      #Keys:1     Level=3
Start=2884637      Size=20459
      Key=www.answers.com/related%253Awww.nascar.com/races/wc/2003/data
/schedule.html
============================================================
```

Total key size of Node = 600 character length. With this size Number of nodes created = 62 and Level of tree = 2

C:\btree>java Main input.dat 3 >tree.doc

```
Memory Heap=1683 KB
============================================================
Node Id: 1(root) Parent Node Id:0 ,  #Keys:4     Level=0
Start=1606017      Size=13167
      Key=www.answers.com/related%253Awww.usafootball.com/features/play
erfeature_conindex.html
Start=2640091      Size=13610
      Key=www.forrelease.com/D20050914/1264638.html
Start=1742833      Size=2833  Key=www.goshenstorage.com/
Start=8914101      Size=10038 Key=www.hanna-policygovernance.com/
============================================================
Node Id: 2(internal) Parent Node Id:1 ,    #Keys:9     Level=1
Start=3118685      Size=4729
      Key=genealogy.harmonicflight.net/legacy/1310.htm
Start=3224615      Size=16403 Key=glossa.fltr.ucl.ac.be/
Start=468755       Size=11977 Key=gremio.ccdc.cam.ac.uk/
Start=2474435      Size=24724 Key=guiden.guildportal.com/
Start=8628997      Size=17908 Key=heroesblood.guildportal.com/
Start=2653701      Size=18493
      Key=www.answers.com/related%253Aentertaining.about.com/cs/etiquet
te/a/thankyou.htm
```

```
Start=9058955      Size=19578
      Key=www.answers.com/related%253Agoflorida.about.com/library/bls/b
l_photo_tampa14.htm
Start=57141  Size=19807
      Key=www.answers.com/related%253Anovi.lib.mi.us/aboutlib/calendar/
bracelets.htm
Start=2884637      Size=20459
      Key=www.answers.com/related%253Awww.nascar.com/races/wc/2003/data
/schedule.html
============================================================
Node Id: 3(leaf) Parent Node Id:2 ,  #Keys:14    Level=2
Start=7741622      Size=6038  Key=fsu.campusrec.com/crenshaw.html
Start=4373628      Size=32871 Key=gallery.aut.ac.nz/album05
Start=1845592      Size=38937 Key=gallery.baz.org/album45
Start=4054440      Size=11293 Key=gallery.derosia.org/Awards
Start=6146051      Size=16446 Key=gallery.ittefaq.com/nation/pic9
Start=4949725      Size=28279 Key=gallery.johny.sk/bp
Start=1912996      Size=19290 Key=gallery.kinfolk.org/album96
Start=4416382      Size=50253 Key=gallery.peimic.com/ebay
Start=9453039      Size=35107 Key=garage.docsearls.com/node/view/503
Start=6906378      Size=14850 Key=garage.docsearls.com/node/view/504
Start=4611752      Size=16524 Key=garage.docsearls.com/node/view/505
Start=2448219      Size=17142 Key=garage.docsearls.com/node/view/508
Start=41004  Size=16137 Key=garage.docsearls.com/node/view/509
Start=913832       Size=2100
      Key=genealogy.bgwoodruff.com/reunionweb/ps01/ps01_184.html
============================================================
Node Id: 4(leaf) Parent Node Id:2 ,  #Keys:17    Level=2
Start=6008685      Size=3959
      Key=genealogy.harmonicflight.net/legacy/1818.htm
Start=3947293      Size=3691
      Key=genealogy.harmonicflight.net/legacy/2044.htm
Start=7245636      Size=4766
      Key=genealogy.harmonicflight.net/legacy/4530.htm
Start=6728891      Size=56570 Key=ghana.info-pagina.com/
Start=6531961      Size=3557  Key=ghettosrise-records.com/
Start=6991517      Size=34038 Key=ghostwhowalksinside.org/
Start=5783728      Size=52361 Key=giants.realfootball365.com/
Start=5875622      Size=18785 Key=gillian.simplenet.com/
Start=7799374      Size=427   Key=gimel.esc.cam.ac.uk/
Start=5741246      Size=21536 Key=girl.tornpaperbag.com/
Start=4256201      Size=5679  Key=glass.mallettantiques.com/
Start=4133968      Size=33489 Key=glayzed_honey.paxed.com/
Start=3752263      Size=2745  Key=glibersat.linux62.org/
Start=6922381      Size=13756 Key=global.finland.fi/rekry/linkit.htm
Start=3390252      Size=556   Key=global.proximity.on.ca/
Start=5588649      Size=712   Key=globaldomain.saax.com/
Start=4336752      Size=36876 Key=globefund.manulife.ca/
============================================================
Node Id: 5(leaf) Parent Node Id:2 ,  #Keys:11    Level=2
Start=3162480      Size=409   Key=gmso.linux-sevenler.org/
Start=2778151      Size=8475  Key=goldenservicesllc.com/
Start=2228092      Size=7245  Key=goldsgym.pretzelman.com/
Start=167232       Size=25514 Key=golf.gfccnet.org/
Start=2465361      Size=6632  Key=goodenoughsprings.org/
Start=1911264      Size=1634  Key=gopublicity.cognigen.org/
Start=2090793      Size=18786 Key=gossamer.simplenet.com/
```

```
Start=6187547      Size=9579
      Key=gotigers.obu.edu/baseball/Stats/Obuac1.htm
Start=3134963      Size=27517 Key=gr.fipu.krasnoyarsk.edu/
Start=241361       Size=3305  Key=greatercommunities.com/
Start=459381       Size=1283  Key=grefo.ishavingamassage.com/
============================================================
Node Id: 6(leaf) Parent Node Id:2 ,  #Keys:6     Level=2
Start=752588       Size=16837 Key=griffel.vabhosting.com/
Start=2335431      Size=1390  Key=groenlinks.westbrabant.net/
Start=526549       Size=1180  Key=groothuizen.babbelt.nl/
Start=716856       Size=35732 Key=groups.montrosetravel.com/
Start=958780       Size=5339  Key=gruposdefe.marianistas.org/
Start=509999       Size=1622  Key=grzegorz.cognigen.org/
============================================================
Node Id: 7(leaf) Parent Node Id:2 ,  #Keys:13    Level=2
Start=9703922      Size=1541  Key=gunslinger.codenamenetwork.com/
Start=7605919      Size=77    Key=guy.forclark.com/
Start=9619328      Size=13012 Key=hagens-berman.vabhosting.com/
Start=10028115     Size=12765 Key=hairfertilizer.qcommerce.com/
Start=10317877     Size=45900 Key=handbag.propertyfinder.co.uk/
Start=5323152      Size=11920 Key=happyblue.net/gallery/1996
Start=9078533      Size=639   Key=happyhome.ownanewbusiness.com/
Start=468675       Size=80    Key=hardly.forclark.com/
Start=10048731     Size=5402  Key=hastingtrailers.yorke.net.au/
Start=3957280      Size=80    Key=hawker.forclark.com/
Start=9655806      Size=14855 Key=healthinfo.health-
first.org/advantage/DTFAP.htm
Start=7988522      Size=10627 Key=help-desk-analyst.info-all.org/
Start=9198553      Size=21045 Key=helpdesk.readysetconnect.com/
============================================================
Node Id: 8(leaf) Parent Node Id:2 ,  #Keys:8     Level=2
Start=7261066      Size=11128 Key=higher-education.info-all.org/
Start=8614663      Size=14233 Key=hitachilabparts.jmscience.com/
Start=7185171      Size=21669 Key=hkretirement.axarosenberg.com/
Start=7976253      Size=12269 Key=home.config.com/
Start=6921340      Size=639   Key=homebased.ownanewbusiness.com/
Start=7160982      Size=24189 Key=homeschoolresourceexchange.com/
Start=2280316      Size=14584
      Key=www.answers.com/related%253Aactionfigures.about.com/library/p
ress/blebayholiday.htm
Start=3552243      Size=16677
      Key=www.answers.com/related%253Achristianmusic.about.com/library/
bl_georgehuff_cmas.htm
============================================================
Node Id: 9(leaf) Parent Node Id:2 ,  #Keys:4     Level=2
Start=1279271      Size=18511
      Key=www.answers.com/related%253Aentertaining.about.com/cs/etiquet
te/a/thankyou_5.htm
Start=1829828      Size=15764
      Key=www.answers.com/related%253Afamilycrafts.about.com/library/bd
spins/blbdspinxmas.htm
Start=10067148     Size=13640
      Key=www.answers.com/related%253Afootball.about.com/cs/football101
/g/gl_playbook.htm
Start=5762878      Size=20753
      Key=www.answers.com/related%253Afootball.ballparks.com/NFL/Oaklan
dRaiders/oldindex.htm
```

CHAPTER 5: CONCLUSION AND FUTURE WORK

The basic idea behind this dissertation work is to construct a multilevel index that supports variable length keys. This index is mainly constructed to retrieve information from large hypertext collections. Because of the characteristics of the B-tree, it is chosen as data structure to implement multilevel index for this problem. Definition of B tree assumes the tree with some fixed order. The maximum number of keys a node can hold is fixed and is already known in advance. In this work, this property of B tree has been changed. There is no fixed order of the tree and the number of keys a node can hold depends on the size of the keys. The maximum number of keys in different nodes may be different. The condition to split a node depends on free available space in the node rather than the number of keys in the node. The program has been implemented quite differently than the algorithms for basic operations of B-tree.

There are three basic operations on B-tree - searching a key, inserting a key and deleting a key. This implementation has support for searching a key and inserting keys. It also has option for printing the entire index with keys in node wise fashion. There is no provision for deleting a key from the index. Designing algorithm for deleting key from index is left as future work. The constructed algorithm will support variable length record. After incorporating deletion, this index can be used in any real application not only in hypertext mining applications.

REFERENCES

[1] Srinath Srinivasa, Mandar M. Mutalikdesai “An Online Analytical Processing Framework For Large Hypertext Collection”, 2006

[2] Carl Franklin, "Hypertext Defined and Applied," Online 13 (May 1989)

[3] Carolyn L. Foss, "Tools for Reading and Browsing Hypertext," Information Processing & Management 25 (1989)

[4] Vannevar Bush, "As We May Think," Atlantic Monthly 176 (July 1945)

[5] Ray R. Larson, "Hypertext and Information Retrieval: Towards the Next Generation of Information Systems," ASIS '88 Proceedings of the 51st ASIS Annual Meeting Atlanta, Georgia October 23-27, 1988. Ed. by Christine L. Borgman and Edward Y.H. Pai. (Medford, New Jersey: ASIS, 1988)

[6] S. Chakrabarti. Mining the Web: Discovering Knowledge from Hypertext Data. Morgan Kaufmann, 2002

[7] R. Kumar, P. Raghavan, S. Rajagopalan and A. Tomkins. Trawling Emerging Cyber Communities Automatically. Proc. of the 8th International World Wide Web Conference, 1999

[8] J. Dean and M. Henzinger. Finding Related Web Pages in the World Wide Web. Proc. of the 8th International World Wide Web Conference, 1999

[9] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. Proc. of the 7th International World Wide Web Conference, 1998

[10] Yasar Tonta. “Indexing in Hypertext Databases”. School of Library and Information Studies, University of California at Berkeley, CA 94720

[11] Thijs Westerveld, Wessel Kraaij, and Djoerd Hiemstra “Retrieving Web Pages using Content, Links, URLs and Anchors”

[12] David Lomet “The Evolution of Effective B-tree Page Organization and Techniques: A Personal Account”, Microsoft research, WA 98052

[13] Jan Jannink, “Implementing Deletion in B+-Trees”, Stanford university

[14] Edward M. McCreight, “Pagination of B*-Tree with Variable Length Records”

[15] Linda Farmer, "Hypertext: Links, Nodes and Associations," Canadian Library Journal 46 (August 1989).

[16] J. Conklin, "Hypertext: An Introduction and Survey," IEEE Computer 20 (1987).

[17] M. Thelwall, Link Analysis: An information Science Approach. Elsevier, 2004

[18] Larson, "Hypertext."

[19] Larson, "Indexing."

BIBLIOGRAPHY

1. File Structures – An Object-Oriented Approach with C++, Michael J. Folk, Bill Zoellick, Greg Riccardi

2. Fundamentals of Database Systems, Fourth Edition, Remez Elmasri, Shamkant B. Navathe

3. Introduction to Algorithms, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

4. Database Management Systems, Third Edition, Raghu Ramakrishnan and Johannes Gehrke

5. Data Structures and Algorithm Analysis in C, Second Edition, Mark Allen Weiss

6. Java ™ 2: The Complete Reference, Fourth Edition, Herbert Schildt