An Efficient Algorithm for Fetching and Processing of Disk based Data Structures

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Sahil Singla
(Roll No. 801031026)

Under the supervision of:
Mr. Ravinder Kumar

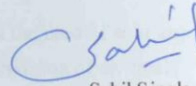
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004
June 2012

Certificate

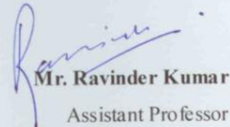
I hereby certify that the work which is being presented in the thesis entitled, **"An Efficient Algorithm for Fetching and Processing of Disk based Data Structures "**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Ravinder Kumar* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.



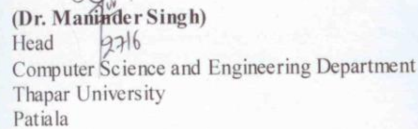
Sahil Singla

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

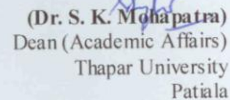


Mr. Ravinder Kumar
Assistant Professor
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by



(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala



(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

i

Acknowledgment

I express my sincere and deep gratitude to my guide Mr. Ravinder Kumar, Assistant Professor in Computer Science & Engineering Department, Thapar University Patiala, for the invaluable guidance, support and encouragement. He provided me all resource and guidance throughout thesis work.

I am heartfelt thankful to Dr. Maninder Singh, Head of Computer Science & Engineering department Thapar University Patiala, for providing us adequate environment, facility for carrying out thesis work.

I would like to thank to all staff members who were always there at the need of hour and provided with all the help and facilities, which I required for the completion of my thesis. At last but not the least I would like to thank God and mine parents for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Sahil Singla
801031026

Abstract

Data sets are often too immense to fit completely inside the computer's main memory and must instead reside on disk. If data set will be kept in main memory it will be very costly. A computer must retrieve required data and place it in internal memory to process it. Efficient data structures, like b-tree, b+ tree, are used to process large datasets. Nodes of these data structures are buffered in memory using data structures like arrays, linked list. In this thesis an algorithm is proposed which gives better results in case of disk based data structure and can be used for those disk based data structures which uses the concept of immediate modification i.e. when modification will be done in buffered structure (present in memory), an immediate modification will be done on disk . Although in case of disk based data structures the processing time is much less but if we try to reduce the processing time of CPU it will also gives beneficial results. Number of maximum disk accesses depends on height of b-tree which is $O(\log_{\text{blocksize}/2} n)$ [1], there is no change in the number of disk accesses. But the CPU processing time in case of search, insert, delete operation is modified, reduced to $O(\log n)$ from $O(n)$. The modified algorithms of b-tree are also provided in this thesis.

Table of Contents

Certificate……………………………………………………………………………………..
Error! Bookmark not defined.

Acknowledgment……………………………………………………………………ii

Abstract…………………………………………………………………………...iii

Table of Contents…………………………………………………………………iv

List of Figures……………………………………………………………………… vi

List of Tables……………………………………………………………………… vii

Chapter1 Introduction……………………………………………………………….. 1

1.1. Overview…………………………………………………………………...1

1.2. Data Structures……………………………………………………………..1

1.2.1. Primitive Data Structures…………………………………………………...2

1.2.2. Non-primitive Data Structures……………………………………………...2

1.3. Classification of data structure on basis of memory………………………………3

1.3.1. Memory based data structures………………………………………………... 3

1.3.2. Disk based data structures…………………………………………………. 3

1.3.2.1. Need of disk based data structure……………………………………... 3

1.3.2.2. Some of the disk based data structures……………………………….... 4

1.3.2.3. Disk's structure………………………………………………………....4

Chapter 2 Literature Survey…………………………………………………………7

2.1. B-tree……………………………………………………………………………7

2.1.1. Definition of B-tree..……………………………………………………..7

2.1.2. Lemmas of B-tree……………………………………………………….. 8

2.1.3. Operations of B-tree……………………………………………………….. 9

2.1.3.1. Creating an empty B-tree……………………………………………….. 10

2.1.3.2. Searching a key into a B-tree…………………………………………… 11

2.1.3.3. Inserting a key into a B-tree……………………………………………..12

2.1.3.4. Splitting a node in a B-tree…………………………………………….... 14

2.1.4. Illustrations of B-tree……………………………………………………... 16

2.1.4.1. Searching in B-tree…………………………………………………….... 16

2.1.4.2. Insertion in non-full………………………………………………….. 16

2.1.4.3. Insertion with Splitting……………………………………………………. 17
2.1.4.4. Deletion in leaf node……………………………………………………….18
2.1.4.5. Deletion in internal node…………………………………………………...19
2.2. B-trie……………………………………………………………………………...20
2.2.1. Structure of B-trie…………………………………………………………21
2.2.1.1 Trie…………………………………………………………………………21
2.2.1.2. Bucket……………………………………………………………………...22
2.2.2. Algorithm for B-tries……………………………………………………... 23
2.2.2.1. Search………………………………………………………………………23
2.2.2.2. Insert……………………………………………………………………… 25
2.2.2.3. Pure Split…………………………………………………………………...27
2.2.2.4. Hybrid split……………………………………………………………... 28
2.2.2.5. Delete……………………………………………………………………… 29
2.2.3. Example of insertion……………………………………………………… 31
2.2.3.1. Initialization of B-Trie…………………………………………………….. 31
2.2.3.2. Simple insertion…………………………………………………………32
2.2.3.3. Splitting of hybrid bucket………………………………………………. 32
2.2.3.3. Creation of pure bucket………………………………………………….33
2.2.3.4. Splitting of pure bucket………………………………………………….33
2.2.3.5. Adding short strings…………………………………………………… 34
2.2.3.6. Deletion in B-trie………………………………………………………. 35
Chapter 4 Proposed Structure………………………………………………………37
4.1. Structure…………………………………………………………………………..38
4.2. Algorithms for Structure……………………………………………………………38
4.3. Illustrations………………………………………………………………………. 42
4.3.1. Creating a List (pds)……………………………………………………… 43
4.3.2. Searching………………………………………………………………… 43
4.3.3. Insert……………………………………………………………………… 43
4.3.4. Delete…………………………………………………………………….. 44
4.4. Comparison of various data structures……………………………………………44
Chapter 5 Proposed B-tree Algorithms…………………………………………………. 45
5.1. Proposed algorithm of search in B-tree………………………………………….. 45
5.2. Proposed algorithm of insert in B-tree……………………………………………46
5.3. Proposed algorithm of split in B-tree……………………………………………..48
Chapter 6 Conclusion and Future work…………………………………………………51

References.. 52

List of Figures

Figure No.	Figure Title	Page No.
Figure 1.1	Structure of a disk	5
Figure 2.1	Sample B-tree	7
Figure 2.2	2-5 B-tree	16
Figure 2.3	Before insertion of 21	17
Figure 2.4	After insertion of 21	17
Figure 2.5	Before insertion of 37	18
Figure 2.6	After insertion of 37	18
Figure 2.7	Before deletion of 4	18
Figure 2.8	After deletion of 4	19
Figure 2.9	After deletion of 18	19
Figure 2.10	Sample B-trie	20
Figure 2.11	A sample trie	21
Figure 2.12	A sample bucket	22
Figure 2.13	Types of bucket	23
Figure 2.14	Searching in B-trie	25
Figure 2.15	Initialization of B-trie	31
Figure 2.16	Insertion of string “cat”	32
Figure 2.17	Insertion of string “algorithm”	32
Figure 2.18	Insertion of string “computer”	33
Figure 2.19	Creation of pure bucket	33
Figure 2.20	Splitting the pure bucket	34
Figure 2.21	Adding Short string “c”	34
Figure 2.22	Deletion of strings “desktop” and “practice”	35
Figure 4.1	After buffering the block in pds	41
Figure 4.2	After Insertion of α_6	42
Figure 4.3	After Deletion of α_2	42

List of Tables

Table No.	Table Title	Page No.
Table 4.1	Comparison of various data structures	42

Chapter1
Introduction

1.1. Overview

In the present age *computer* is a mandatory part of every possible business infrastructure. The dependency is increasing almost exponentially. Since the involvement of computer is increasing so does the problems related with it. One of the problems is Data Management. It is essential in almost every I.T. based organization to store the current and the historical data figures. These data figures act as very important part in strategic business development. All the risk management and market prediction is done on the basis of the pattern developed by using the stored data. Since the importance of stored data and the increasing number of users, data management becomes an essential problem to deal with. Data management stands for operations like insertion, deletion and searching. Also the security of the stored data is part of data management. Security is further subdivided in authorization and disaster management. It getting costlier day by day to perform these operations because of continues increase in data. The solution to these problems lies in the way of storing the data.

Let us take simple life example; suppose we have to put clothes in a Cupboard. If all the clothes are put in a random manner then finding a particular shirt will be very difficult and time consuming process. To make search easy, structuring of clothes should be done like all shirts in one column and all pants in another column and so on. Now, to search a particular shirt is very easy then. So, there is a need of structure to store in efficient way. To store data some structures are available which are called *Data Structures*.

1.2. Data Structures

A data structure is a mathematical or logical way of organizing data in the memory that consider not only the items stored but also the relationship to each other and also it is characterized by accessing functions. [16]

They are of two types

1) Primitive
2) Non Primitive

1.2.1. Primitive Data Structures

Primitive data structures are those data structures which are directly operated upon by machine level instructions like int, float and char. The primitive data structure can be either of the following:

- A basic type is a data type provided by programming language as basic building block.
- A built-in type is a data type for which programming languages provide built in support.

The actual range of primitive data types depends on the programming language which we are using like in case of c or c++ strings are composite (i.e. strings are array of character) but it is built in but in case of modern programming languages like java or c# strings are both basic and built-in.

1.2.2. Non-primitive Data Structures

Non Primitive data structures are derived structures from primitive like array, structure, union and class. These are much more complex in nature. They emphasize on formation of sets of similar and different data elements. Non – primitive data structure can be of two types:

- Linear
- Hierarchical

Linear data structures are data structure in which while traversing sequentially, we can reach only one element directly from another like array. Also a data structure in which there is contiguous memory allocation is called linear data structure.

Hierarchical data structure is data structure in which we can reach two or more element while traversing sequentially like B-tree. Also data structure in which there is non-contiguous memory allocation is called non-linear data structure.

1.3. Classification of data structure on basis of memory

The on basis of memory (where data structure can be saved) data structure can be of two types:

1) Main Memory data structures.
2) Secondary Memory data structures (disk based data structure).

1.3.1. Memory based data structures

Memory based data structures are data structures which resides in memory like array, link list. Data of these structures is present in memory also processing is done in memory itself. These data structures are used for the small sets of data. It's processing is fast as there is no need to transfer of data. As main memory is volatile so as these data structures. These data structures are used for temporary filling and processing of data.

1.3.2. Disk based data structures

Disk based data structure are the data structures, such as b-trees, which are stored and accessed from disk but processed only in memory. Updates are only present for an indefinite amount of time in memory. Whenever a change is done it is immediately forwarded to disk. The b-trees [5] and b+-trees [9, 19] are used in the implementation of databases like spatial databases, geographical data bases. "Nikolas Askitis" proposed b-trie [3], a data structure for disk based string management which gives better results than b-tree. The need of disk based data structures is increasing because of the growth in the size of datasets due to which the whole dataset cannot put into the main memory.

1.3.2.1. Need of disk based data structure

As explained above there are two types of memory primary and secondary. The main memory or primary memory typically consists of silicon memory chips, each of which can hold 1 million bits of data [13]. Working of this (primary) memory is fast but it is expensive as compared to secondary memory. Therefore in a computer primary memory is kept small but secondary memory is large. Sometimes data sets are so large like in case

of dictionary that they exceed amount of primary memory. Also to store large data sets on primary memory will be more costly then to store on secondary memory. Moreover, primary memory is volatile is dataset will be stored on primary memory it will lost as power shuts down, but if it will kept in secondary memory it will not lost even on power failure. So there is need to store datasets on secondary memory. Whenever processing of these data sets is required, these data sets are moved to primary memory and again moved back to secondary memory after processing.

1.3.2.2. Some of the disk based data structures

The B-tree [18] is considered to be the most efficient data structure for maintaining sorted data on disk [2]. A key characteristic is the use of a balanced tree structure, which guarantees worst-case $O(log_B N)$ performance for search of N keys with a branching factor (node fan-out i.e. number of children) of B, doesn't matter how the distribution of data is. In practice, due to its high branching factor, typically the total volume of internal nodes is very small and traversal requires only a single disk access [3].

External hash tables are also efficient data structures there access time is constant to n, but cannot guarantee a bounded worst-case cost, nor can they maintain strings in sort order, sorting is required for efficient range search. In addition to its widespread use in standard database systems, the B-tree has many applications, including databases, information retrieval, and genomic databases.

The B-trie proposed by Szpankowski [10] is simply a static trie indexing a set of buckets that store up to b keys. Askitis propose new algorithms for the insertion, deletion, and equality search of variable-length strings in a disk-based B-trie, for use in common string processing tasks such as vocabulary accumulation and dictionary management.

1.3.2.3. Disk's structure

A Disk's structure consists of disk, which is made up of cylinders. Each cylinder is divided into tracks and each track is further divided into sectors. A sector size can be 512 bytes to 16 kilobytes. A disk is accessed in a similar manner as an array. Sector 0 is the first sector of the upper track on the outermost cylinder [1]. Ordering of sectors is done

from sector 0 of the upper track of the outermost cylinder to lowest sector of the same track then numbering is given to sectors of the next track until uppermost cylinder is not finished. Similarly the numbering of sectors is proceeded for inner cylinders. Disk performance depends upon two parameters, first is seek time and second is rotational latency. Seek time is the time to move the disk arm to the proper cylinder. Rotational latency is the time to rotate the disk to required sector. Data is written on the sectors of the disk. A sector/block is the unit used to transfer data between memory and disk.

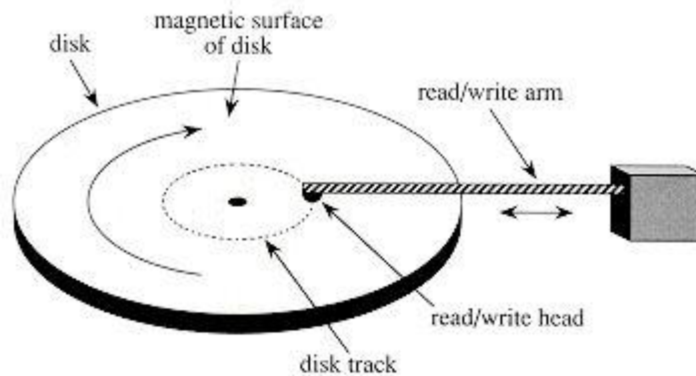


Figure 1.1: Structure of a disk [13]

The surface of disk is enclosed with magnetizable material. Then read/write head can read or write data using this magnetic material on the rotating disk face. The read/write arm can place the head at different positions on the surface of disk i.e. distance of head from center can vary. The concentric circles on disk surface is called track i.e. if head is kept at fixed position and disk surface is rotated then the surface passes under the head is called track. The track is divided into sectors. The access time is the time between the placing to head and to reach head at proper sector. This time can vary as it depends on, how far is the sector, where data is present and also on initial state of the disk. The time to access a sector and read it from a disk is large as compared to time taken by computer to examine all the information read. Due to this running time is divided in two parts:

- the number of disk accesses.
- the CPU (computing) time.

The number of disk accesses is means number of times we need to access a disk. Number of disk accesses will depend on number sectors of information that are needed to read or write. The CPU time is time required by CPU to process the data i.e. to read or write the data.

This thesis consists of 6 chapters. Chapter 2nd is literature survey which contains the explanation of b-tree and b-trie with algorithms and illustrations. Chapter 3rd is problem statement, in which motive of thesis is defined. Chapter 4th is Proposed Structure in which explanation of proposed structure is explained with the help of algorithm and illustrations. Chapter 5th is proposed b-tree algorithms. Chapter 6th is conclusion.

Chapter 2

Literature Survey

2.1. B-tree

B-trees are balanced search trees designed to work well on access secondary storage devices such as Hard disk. B-trees are better at minimizing disk I/O operations by reducing the height of the tree. B-tree is used in many database systems use B-tree to store information. In a B-tree each node may contain a large number of keys [17].

B-trees have become, de facto, a standard for file organization. File indexes of users, dedicated database systems, and general-purpose access methods have all been proposed and implemented using B-trees.[5]

B-trees are generalized binary search trees. An internal B-tree node x contains one less the number of keys from children. The b-tree is a multi-way tree in which keys in node x are used to divide the range of keys handled by x into n[x] + 1 sub-divisions, each division is handled by one child of x. When searching for a key in a B-tree, we make an (n[x] + 1)-way decision based on comparisons with the n[x] keys stored at node x.[13]. A sample B-tree of intergers is shown in Figure 2.1.

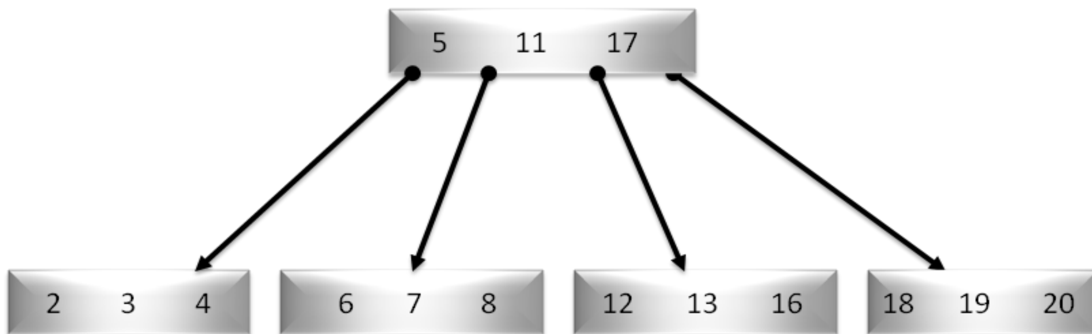


Figure 2.1: Sample B-tree

2.1.1. Definition of B-tree

A ***B-tree*** *T* is a rooted tree (with root *root*[*T*]) having the following properties[13].

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: $key_1[x] \leq key_2[x] \leq - - - - - \leq key_n[x]$,and

c. $leaf[x]$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. If x is an internal node, it also contains $n[x] + 1$ pointers $c_1[x]$, $c_2[x]$, . . . , $c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

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

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq - - - - - - \leq key_n[x] \leq key_{n[x]+1}$$

4. Every leaf has 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 $t \geq 2$ called the minimum degree of the B-tree:

a. Every node other than the root 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.

The simplest B-tree occurs when $t = 2$. Every internal node then has 2, 3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values of t are typically used.

2.1.2. Lemmas of B-tree

Lemma 2.1.2.1. : Maximum number of nodes at level L when the node size is n is n^L .

Proof 2.1.2.1. :

Basis $d = 0$;

$n^d = 1$, Satisfied for trivially for the root.

Induction: Suppose that the property holds for some d, The number of nodes in that level is n^d nodes at depth d,

then for depth $d + 1$, each of the nodes at depth d can have at most n children.

Thus the maximum possible nodes at $d + 1$ is $n * n^d$, but this is n^{d+1}

Thus the property holds for a depth of $d + 1$.

Lemma 2.1.2.2. : if node size is n and L is level. Then total of elements $(n^{L+1} - 1)/n - 1$.

Proof 2.1.2.2. :

First by definition the number of nodes is the sum of nodes at each level.

Basis $h = 0$

$(n^{h+1} - 1)/(n - 1)$, but the root is the only node in a height 0 tree thus the basis is satisfied.

Induction:

Suppose that the property holds for some h, then

The number of nodes in the tree is $(n^{h+1} - 1)/(n - 1)$

then by inductive hypothesis there are

$(n^{h+1} - 1)/(n - 1)$ trees in the first h levels and by Lemma 2.1.2.1. no more than n^{h+1} in the next level of the tree.

so the total number of nodes at depth $h + 1$ is

$n^{h+1} + (n^{h+1} - 1)/(n - 1)$

$(n^{h+2} - n^{h+1} + n^{h+1} - 1)/(n - 1)$

$(n^{h+2} - 1)/(n - 1)$ therefore, the property holds for a depth of $h + 1$.

2.1.3. Operations of B-tree

The algorithms for the *search create*, and *insert* operations are single pass means they do not traverse back the tree. Some simpler double pass approaches are necessary to fix the violations. Since b-tree is to minimize the number of disk accesses and the nodes are usually stored on secondary memory, the single-pass approach will reduce the node visits and as well as the number of disk accesses. The algorithms of search, insert, create are given below [11]

Because all the nodes are stored on secondary storage (disk) rather than primary storage (main memory), To read a given node the operation is denoted by Disk-Read. Similarly, once a buffered node is modified and it is no longer needed in main memory, it must be written on secondary storage with an operation Disk-Write. The algorithms below assume that all nodes referenced in parameters have already had a corresponding Disk-Read operation. Allocate-Node procedure is used to create new nodes. The implementation details of the Disk-Read, Disk-Write, and Allocate-Node functions are operating system and implementation dependent [11].

2.1.3.1. Creating an empty B-tree

To Create a B-tree T, Firstly, B-TREE-CREATE procedure create an empty node which act as a root node and then call B-TREE-INSERT procedure to add keys. Both of these procedures use a supplementary procedure ALLOCATE-NODE, which allocates one disk page/block to be used as a new node in constant time. There is no need of DISK-READ operation on root node because there is no useful information is present in it initially.

Algorithm 1: Create B-tree

$B_Tree_Create()$

Output: A node is created

1. $Set\ x \leftarrow Allocate_Node(\,);$
2. $leaf[x] \leftarrow true;$
3. $n[x] \leftarrow 0;$
4. $Disk_write(x);$
5. $root[T] \leftarrow x;$
6. Exit

2.1.3.2. Searching a key into a B-tree

The search operation on a b-tree is similar to a search on a binary tree. In binary tree the choice we make is two way but in a b-tree search is n-way where n is the number of elements in node. A linear search is performed on the values of a node to find the correct child. After finding the value superior than or identical to the required value, then the left child pointer of that value is followed. If last value of node is less than the required value, the rightmost pointer is followed. The time complexity of search is only depends upon the height of the b-tree.

Algorithm 2: Search in B-tree

$B_Tree_Search(x, k)$

Input: A node and a element to search.

Output: Location of element.

1. $Set\ i \leftarrow 1;$
2. $Repeat\ for\ i\ while\ i \leq n[x]\ and\ key_i[x]$

   $Let\ i \leftarrow i + 1;$
3. $If\ i \leq n[x]\ and\ key_i[x]\ then$

   $return(x, i);$
4. $If\ leaf[x]\ then$

   $return\ NIL;$
5. $Otherwise, Disk_read(c_i[x]);$
6. $return\ B_Tree_Search(c_i[x], k);$

2.1.3.3. Inserting a key into a B-tree

Insertion is one of the basic operations of a data structure. Inserting a key into a B-tree is bit more complicated than inserting a key into a BST (binary search tree). In binary search trees, we search for the position at which to insert the key and simply create a node there to place the new key. In case of B-tree, it is not so simple. We cannot simply create a new node and insert it because it violates the rule of b-trees. The new key should be inserted into an existing node. As a node becomes full we cannot insert a key into that leaf node. So, firstly that node should be split into two nodes with an operation named Split-node. It splits a full node y around its median key into two nodes having t - 1 keys in each [13]. The median key moves up into parent node to become the dividing point between the two new sub-trees. But if parent node is also full, it must be split further.

In case of binary search tree, insertion can be done into a B-tree in a single pass down the tree from the root node to a leaf node. There is no need to split any node because only one element is present at one node. So, no need to check whether parent is full or not. In b-tree we have to split each full node coming in path to insert the new key.

The maximum number of disk accesses to insert a key depends upon the height of b-tree. Suppose if b-tree T has height h then it requires h number of disk accesses. The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD procedure to guarantee that the recursion never descends to a full node.

Algorithm 3: Insert in B-tree

$B_Tree_insert(T, k)$

Input: A B-tree and an element to insert.

Output: A B-tree with element inserted in it.

1. $Set\ r \leftarrow root[T];$

2. $If\ n[r] == 2t - 1\ then$

    $Set\ s \leftarrow Allocate_Node(\ );$

    $Set\ root[T] \leftarrow s;$

$Set\ leaf[s] \leftarrow false;$

$Set\ n[s] \leftarrow 0;$

$c_1 \leftarrow r;$

$B_Tree_Split_Child(s, 1, r);$

$B_Tree_Insert_nonfull(s, k);$

3. $Otherwise\ B_Tree_Insert_nonfull;$.

Algorithm 4: B-Tree-Insert-non full (x, k)

$B_Tree_Insert_Nonfull(x, k)$

Input: A B-tree node and an element to insert.

Output: A B-tree with element inserted in it.

1. $Set\ i \leftarrow n[x];$
2. $If\ leaf[x]\ then$

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

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

$i \leftarrow i - 1;$

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

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

$Disk_write(x);$

$Otherwise,$

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

$Set\ i \leftarrow i - 1;$

$Set\ i \leftarrow i + 1;$

$Disk_read(c_i[x]);$

$If\ n[c_i[x]] == 2t - 1\ then,$

$B_Tree_Split_Child(x, i, c_i[x]);$

$If\ k > key_i[x] then,$

$Set\ i \leftarrow i + 1;$

$B_Tree_Insert_Nonfull(c_i[x], k);$

2.1.3.4. Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD takes 3 arguments as inputs, a non-full internal node x an index i, and a node y such that y is a child of x and i denotes the index of y in x. x and y must be present in memory. The procedure then split y in two by creating a new node z and adjust the median key of y in x. (There is one special case of splitting root because of no parent node present. In that case, we will first make the root as a child of a new empty node which now becomes the root node, so that we can use B-TREE-SPLIT-CHILD. The tree can grow in height by splitting child nodes. Splitting is the only way by which tree may grow. The median key of y is inserted into its parent node x. Those keys in y which are greater than the median key of y node are placed in a new node z in sorted order. Then z is made the child of x.

Algorithm 5: B-Tree-Split-Child(x, i, y)

$\mathrm{B - Tree - Split - Child(x, i, y)}$

Input: Parent node (x), loc (i) and i^{th} child node (y).

Output: Successfully written on disk

1. $Set\ z \leftarrow Allocate - Node(\ );$

2. $Set\ leaf[z] \leftarrow leaf[y];$

3. $Set\ n[z] \leftarrow t-1;$

4. $for\ j\ \leftarrow\ 1:\ t-1$

5. $\quad key_j[z] \leftarrow key_{j+t}[y];$

6. $If\ leaf[y] == false\ then,$

   $\quad for\ j\ \leftarrow\ 1:\ t$

   $\quad\quad c_j[z] \leftarrow c_{j+t}[y];$

7. $n[y] \leftarrow t-1;$

8. $for\ j\ \leftarrow\ n[x]+1:\ i+1$

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

9. $Set\ c_{i+1} \leftarrow z;$

10. $for\ j\ \leftarrow n[x]:\ i$

11. $\quad key_{j+1}[x] \leftarrow key_j[x].$

12. $key_i[x] \leftarrow key_t[y].$

13. $Set\ n[x] \leftarrow n[x]+1.$

14. $Disk_write(y).$

15. $Disk_write(z).$

16. $Disk_write(x).$

2.1.4. Illustrations of B-tree

A simple example of B-Tree in which a node can contain 2 to 5 number of elements is shown in Figure 2.2.

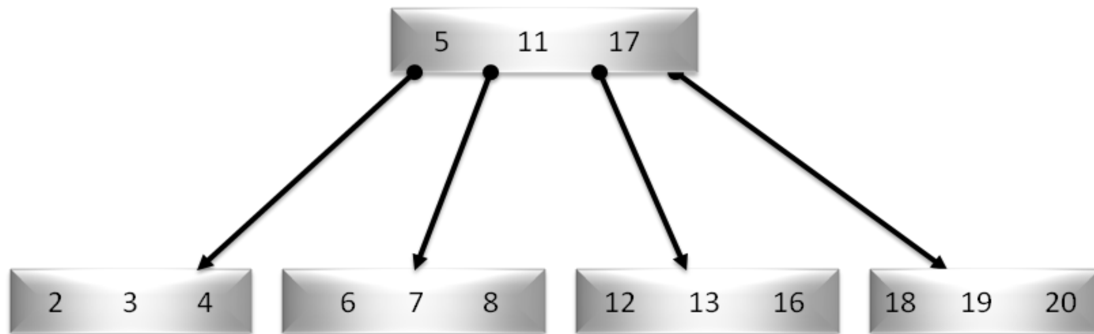


Figure 2.2: 2-5 B-tree

2.1.4.1. Searching in B-tree

(i) Searching of 21 in given sample b-tree returns null.

(ii) Searching of 5 returns (x, 1) according to algorithm where x means the node is root node and 1 means the required element is at 1st position of root node.

2.1.4.2. Insertion in non-full

The non-full B-tree is shown in Figure 2.3. When element 21 is inserted in this tree it will search for proper location which is at child node root element 17. The element will be simply inserted at child node of root element 17 as it is non-full so there will not be need of any split or addition of new child node. The B-Tree after inserting element 21 is shown in Figure 2.4.

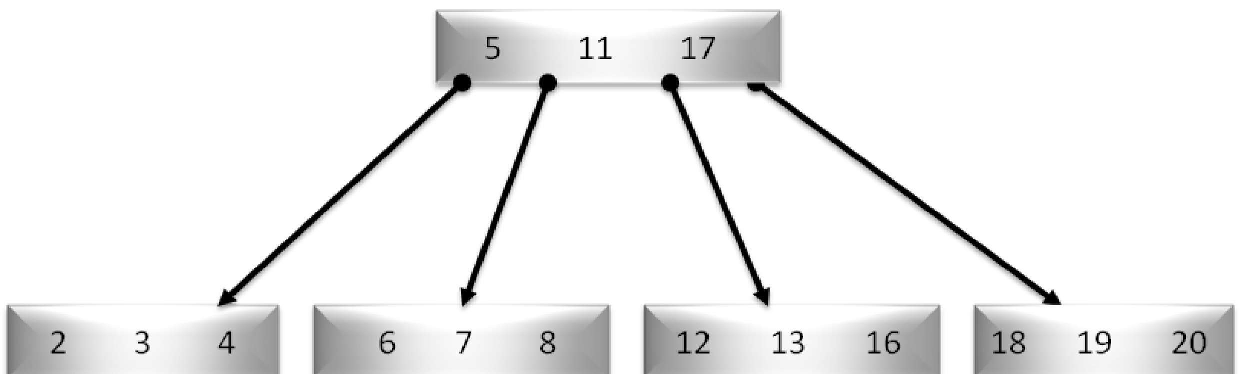


Figure 2.3: Before insertion of 21

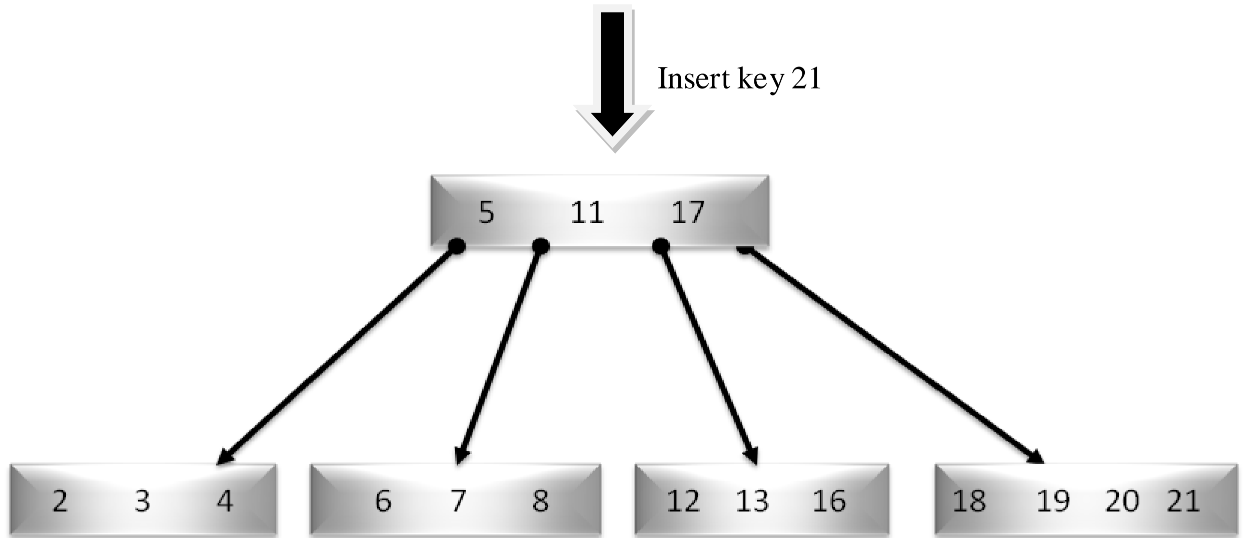


Figure 2.4: After insertion of 21

2.1.4.3. Insertion with Splitting

Now, if element is inserted in a B-Tree which is full then there will be splitting of node. A full B-Tree is shown in Figure 2.5. Consider to insert element 37. When location is searched for insertion of 37 it is child node of element 17 of root node. Now, child node of element 17 already contains 5 elements i.e. no more elements can come in this node. Then mid element of this node i.e. element 20 is added to root node. The filled node is divided into two node one containing elements of node which is smaller than 20 and other contains elements of node which are greater than 20. The B-Tree after splitting is shown is Figure 2.6.

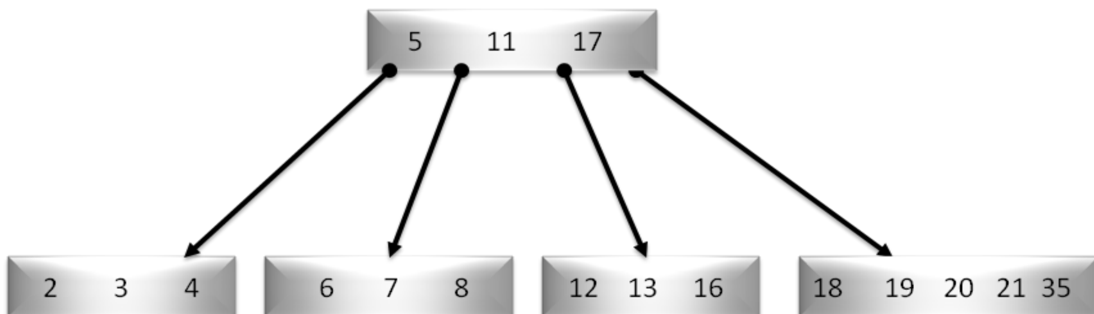


Figure 2.5: Before insertion of 37

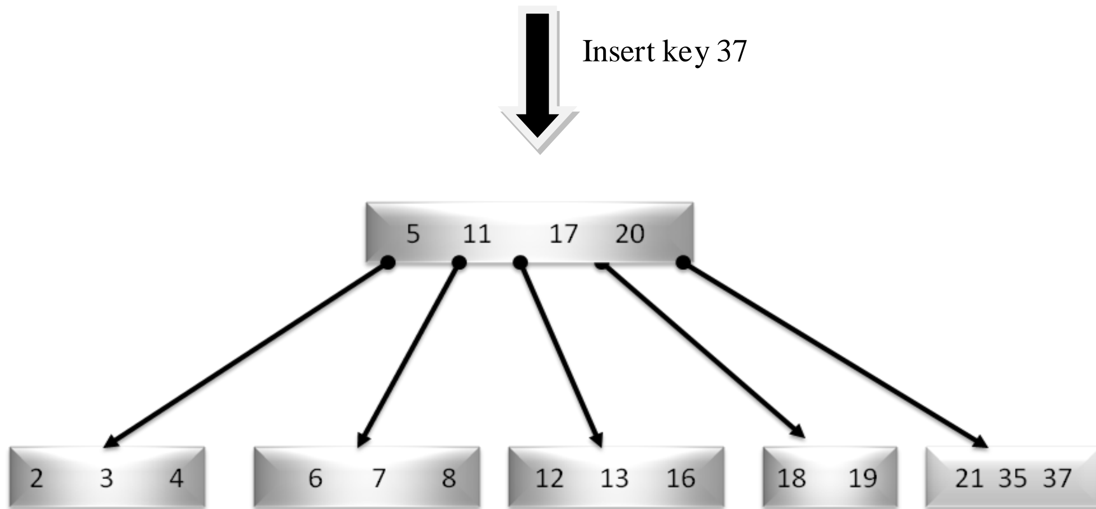


Figure 2.6: After insertion of 37

2.1.4.4. Deletion in leaf node

When an element is deleted from leaf node simply the element is removed. A B-tree before deletion of element 4 is shown in Figure 2.7. Element 4 is present in leaf node so it is simply removed. A B-tree after deletion of element 4 is shown in Figure 2.8.

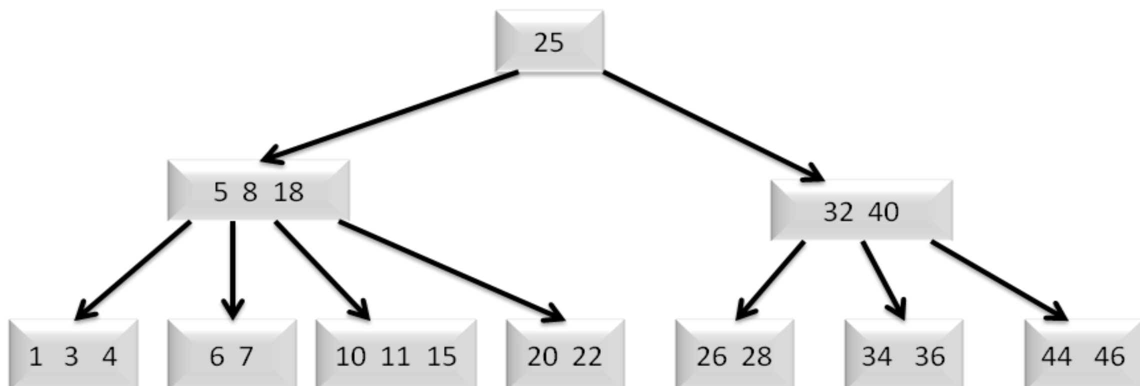


Figure 2.7: Before deletion of 4

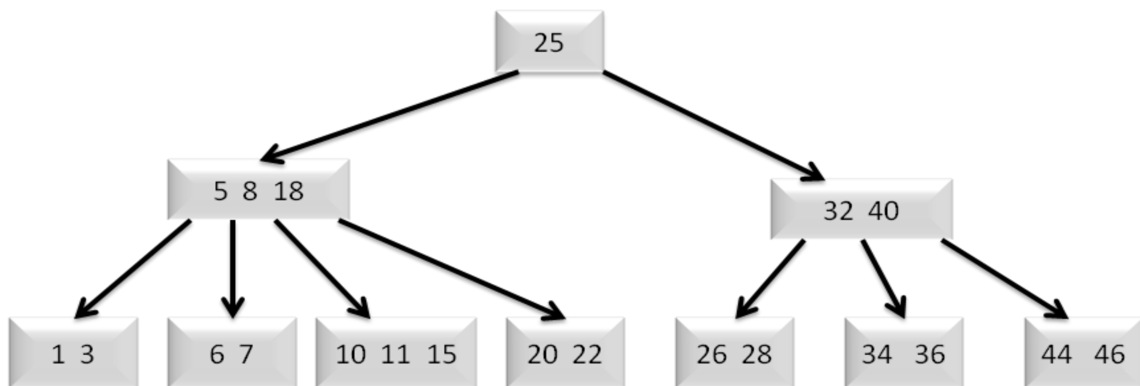


Figure 2.8: After deletion of 4

2.1.4.5. Deletion in internal node

Now, if element to delete is present in internal node (a node which contains at least one child) then there is a need of to fill that empty space. If we have to remove element 18 from B-tree shown in Figure 2.8 then node 15 is picked to fill the empty space of element 18 because element 15 is the largest element of child node pointed by pointer just before element 18. The B-tree after deletion of element 18 is shown in Figure 2.9.

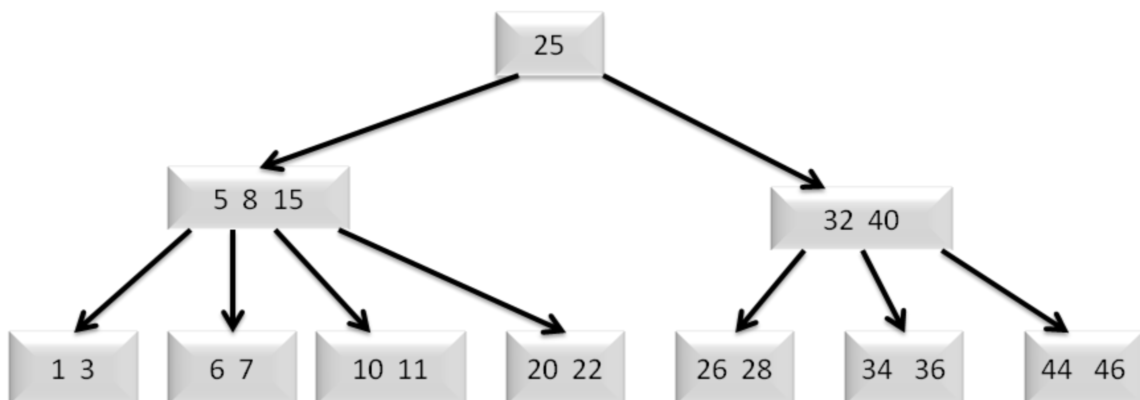


Figure 2.9: After deletion of 18

2.2. B-trie

The B-trie is a trie based data structure that can store and retrieve variable length strings efficiently on disk. The B-trie was compared against several high-performance variants of B-tree that were designed for string keys. It was shown to offer superior performance, particularly under many repeated searches. It is currently a leading choice for maintaining a string dictionary on disk, along with other disk-based tasks, such as maintaining an index to a string database or for accumulating the vocabulary of a large text collection. The B-trie is an unbalanced multi-way disk-based trie structure, designed to sort and cluster strings that share common prefixes.

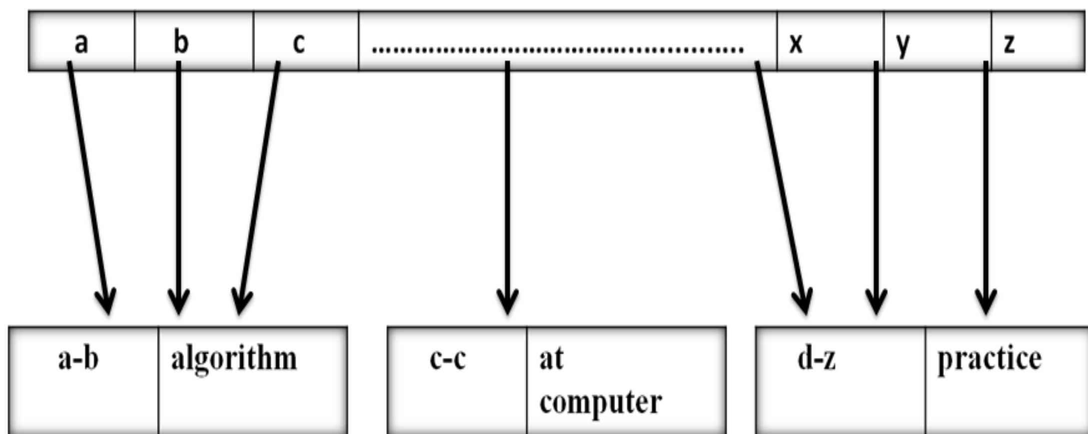


Figure 2.10: Sample B-trie

A major input in b-trie is the development of an suitable approach to bucket splitting. This allows the B-trie to exist in efficiently on disk, by reducing the number of buckets created and also reduced the randomization during the process of splitting. Classification of buckets is either hybrid or pure, which differ in the manner of range field. The B-trie is an unbalanced structure but b-tree is a balanced structure. B-trie uses more memory but it is faster, scalable and requires less disk space [3]. Structure and algorithms of B-trie are explained further with illustrations.

2.2.1. Structure of B-trie

B-trie is a data structure which is used for efficient storage and retrieval of strings on disk. We can do fast search using B-trie. It is made from:

1. Trie
2. Bucket
    1. Pure
    2. Hybrid

2.2.1.1 Trie

The name trie comes from its use for retrieval. It is pronounced like "try" by some, like "tree" (the pronunciation of "trie" in "retrieval") by others. It is an array of pointers one pointer per character. Generally it consists of 128 pointers as in ASCII table. Starting from the root node, by following pointers corresponding to letters we can trace a word in the target word. The leading character of a string is used as an offset and is discarded once a new trie node or a pure bucket is acquired.

In our example we have used array of 26 pointers pointing to each word from a-z.

Structure of trie node is

```
struct trienode
{
 void *pointers[26];
 int  type;
 };
Type is kept 1 to distinguish it from bucket
```

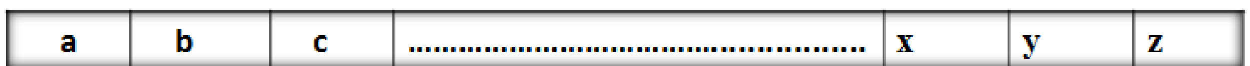


Figure 2.11: A sample trie

2.2.1.2. Bucket

It contains 2D array to store strings. It is somewhat similar to leaf nodes used in B+ tree. However, the lead character of each string in the bucket must be within its character range

Structure of bucket is:

```
struct Bucket
{
char a[Max][100];
char start_offset;
char end_offset;
int  count;
int  type;
};
```

In this structure a is 2D array of bucket having maximum size Max or we can say Max is the maximum number of strings contained in a bucket.

Start_offset is start of the range of initial character of those strings those can contained in a particular bucket.

End_offset is end of the range of initial character of those strings those can contained in a particular bucket.

Count is number of total number of strings present in bucket at any time. Type is kept 0 to distinguish it from trie node.

Figure 2.12: A sample bucket

Pure Buckets

These are the buckets having same start offset and end offset. When pure bucket is formed initial offset of that string is removed. For example if we have string named Laptop in pure bucket L then we will only save aptop (removing L).

Hybrid buckets

These are buckets having different start and end_offset. In below diagram bucket highlighted with blue color is pure bucket and buckets highlighted with red color are hybrid bucket.

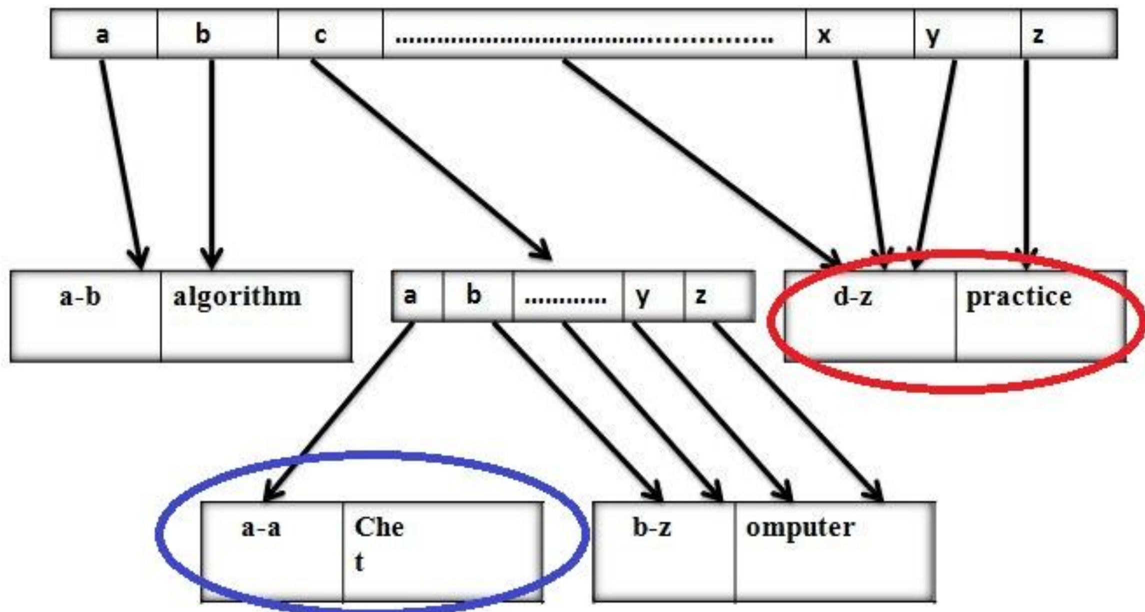


Figure 2.13: Types of bucket

2.2.2. Algorithm for B-tries

2.2.2.1. Search

It takes a string Q as input. Starting from initial character in a string it moves down the B-trie. While moving down the B-trie if a pointer is pointing towards another trie node remove the initial of remaining string and go further. if pointer pointing towards a bucket, then check the type of bucket that whether it is a pure bucket or a hybrid bucket. If it is pure bucket we will leave the initial of the remaining string and search a string with binary search in 2D array of bucket. If bucket is hybrid we will search the whole remaining string. If the complete string is consumed before reaching the bucket then searches that string in hash table (Hash table is used for storing short strings).

Algorithm 6: Search in B-trie

B_Trie_Search (Q, T)

Input: Q is a string; T is the address of first trie.

Output: String search / failure

1. $char\ c = Extract\ the\ image\ of\ initial\ character\ from\ the\ string;$
2. $while(ptr\ is\ pointing\ to\ trie\ node)$

   $Remove\ the\ initial\ char\ from\ string\ Q;$

   $Repeat\ from\ step\ 1;$
3. $If\ ptr == Null$

   $return\ null;$
4. $If\ ptr\ is\ pointing\ to\ bucket$

   $if\ bucket\ is\ pure$

   $\quad Remove\ the\ initial\ char\ from\ string;$

   $if\ remaining_string\ found\ in\ bucket$

   $\quad print \leftarrow found;$

   $else$

   $\quad print \leftarrow not\ found;$
5. $if\ \ becomes\ null\ before\ reaching\ bucket$

   $search\ the\ whole\ string\ in\ Hash\ table;$
6. Exit.

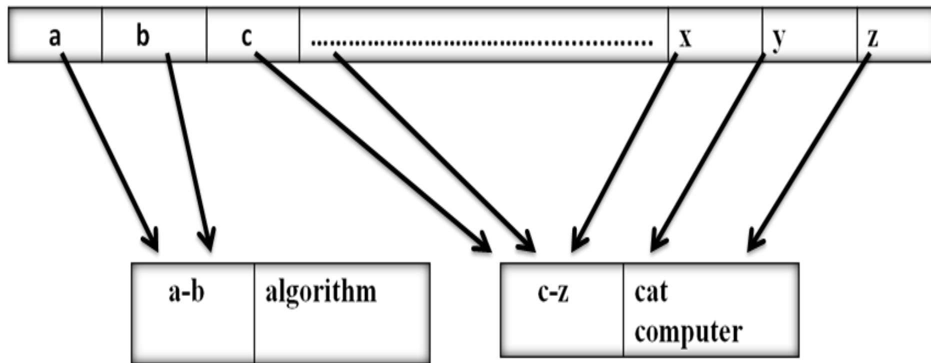


Figure 2.14: Searching in B-trie

Search for string "cat" is 'YES' but for aeroplane returns 'NO'.

2.2.2.2. Insert

Search for a string Q to be inserted. If string is found then we need not to insert that string and program prints "already present". if it is not then starting from initial character of a string Q moves down to the B-trie. While moving down the B-trie if a pointer is pointing towards another trie node removes the initial of remaining string. If null pointer is encountered new bucket is generated to store the remaining string. The new bucket has range that engulfs all neighboring null pointers in parent trie and break. When pointer pointing towards bucket, then check if bucket is full or not. If bucket is not full then, check type of bucket, if it is pure bucket we will leave the initials of the remaining string and insert a string with binary search in 2D array of bucket in sorted order or if it is a hybrid bucket we will simply insert the remaining string with binary search in 2D array of bucket in sorted order. If bucket is full then check the type of bucket, if it is a pure bucket call pure split or if it is a hybrid bucket call hybrid split

Algorithm 7: Insert in B-trie

B_Trie_Insert(Q, T)

Input: Q is a string, T is the address of first trie.

Output: String inserted / failure

```
1.  if element is already present
        print ← alreaady present;
2.  else
a.      char c = Extract the image of initial character from the string;
b.      while(ptr is pointing to trie node)
            Remove the initial char from string Q;
            Repeat from step a;
c.      If ptr == Null
            new bucket is allocated to store remaning_string;
            start_offset ← start of neighbour null pointers;
            end_offset ← end of neighbour null pointers;
d.      If ptr is pointing to bucket
            if bucket is not full
                if bucket is pure
                    Remove the initial char from string;
                    Insert remaining_string into bucket;
                if bucket is hybrid
                    Insert the remaining_string bucket;
5.          else
            if bucket is pure
                call to pure_split;
            if bucket is hybrid
```

$call\ to\ hybrid_split;$

6. $if\ string\ becomes\ null\ before\ reaching\ bucket$

   $Insert\ the\ whole\ string\ in\ Hash\ table;$

7. Exit.

2.2.2.3. Pure Split

To split a pure bucket we will add a new trie node and give its address to the parent trie (which contains the address of pure bucket which we are going to split). There is a need to decide split point d based on the initial character of the strings which is present in pure bucket such that strings got almost half in both buckets. Pure bucket is not destroyed and used further either in form of pure or hybrid bucket and one new bucket will be created. Start_offset of the previous bucket is set to 'a' and end offset of previous bucket will be split point d. Start offset of newly created bucket will be d'(=split point d+1) and end_offset of the newly created bucket is set to 'z'

Algorithm 8: Pure Split in B-trie

B_Trie_Pure_Split (T, B)

Input: B is the address of bucket; T is the address of parent trie.

Output: Split Successfully.

1. $Create\ a\ new\ bucket\ B1\ and\ a\ new\ trie\ node\ N;$
2. $Overwrite\ the\ address\ of\ B\ in\ parent\ trie\ with\ the\ address\ of\ N;$
3. $Decide\ a\ Split\ point\ d;$
   $//to\ divide\ the\ number\ of\ strings\ equally.$
4. $Shift\ those\ strings\ from\ B\ to\ B1\ those\ have\ initial\ char > d;$
5. $for\ i \leftarrow 1{:}d$

$N[i] \leftarrow address\ of\ B;$

6. $for\ i \leftarrow d + 1 : N.length$

$N[i] \leftarrow address\ of\ B1;$

7. $Set\ the\ end_offset\ of\ bucket\ B\ is\ according\ to\ split\ point\ d.$

8. $Set\ the\ start_offset\ of\ bucket\ B1\ is\ initial\ char\ of\ trie.$

9. $Set\ the\ end_offset\ of\ bucket\ B1\ is\ last\ char\ of\ trie.$

10. Exit

2.2.2.4. Hybrid split

Decide split point d based on the initial character of the strings which is present in hybrid bucket (ranging from Start_offset to End_offset) such that strings got almost half in both buckets. Hybrid bucket is not destroyed and used further either in form of pure or hybrid bucket and one new bucket will be created. Start_offset of the previous bucket is not changed and end offset of previous bucket will be split point d. Start offset of newly created bucket will be d+1 and end_offset of the newly created bucket is set to End_offset of previous bucket.

Algorithm 9: Hybrid Split in B-trie

B_Trie_Hybrid_Split (T, B)

Input: B is the address of bucket; T is the address of parent trie.

Output: Split Successfully

1. $Create\ a\ new\ bucket\ B1;$

2. $Decide\ a\ Split\ point\ d;$ //
   to divide the number of strings equally.

4. $Shift\ those\ strings\ from\ B\ to\ B1\ those\ have\ initial\ char > d;$

---

5. $for\ i \leftarrow Start_offset\ of\ B{:}d$

$N[i] \leftarrow address\ of\ B;$

6. $for\ i \leftarrow d+1 : End_offset\ of\ b$

$N[i] \leftarrow address\ of\ B1;$

7. $Set\ the\ end_offset\ of\ bucket\ B1\ is\ the\ end_offset\ of\ bucket\ B.$

8. $Set\ the\ end_offset\ of\ bucket\ B\ is\ according\ to\ split\ point\ d.$

9. $Set\ the\ start - offset\ of\ bucket\ B1\ is\ according\ to\ d+1.$

10. Exit

2.2.2.5. Delete

We search for the required string Q. If Q is fully devoted during traversal, then clear the end-of-string flag in the acquired pure bucket or trie node, and delete the string Q from the hash table.

If pointer encounters a null pointer then we delete Q from the hash table if string exists, for the completion of deletion process.

If pointer points to hybrid bucket, binary searched for the string Q or if pointer points to pure bucket, binary searched for the string after removing initial character. If the string is found, then remove the string and the bucket is within restructured to avoid inner fragmentation. The cost of reorganize a bucket is small and directly proportional to the size of the bucket. if the bucket is empty, all of the outgoing pointers from the parent trie node P to that bucket are null, and the address of file is placed in an address pool for reuse[3]. If a trie node had all of its pointers null, then it is also deleted by making its parent pointer null, and place the address in pool for reuse.

Otherwise, more space efficient deletion scheme [9] for B+-tree can be applied. Whenever bucket becomes empty, immediately its neighbors are checked to determine if some strings can be transferred. Transference of data is initiated only if neighbor is hybrid bucket and not splitting it will not become empty. (A pure can be used, but it can complicate matters because, the lead characters of its strings are needed to restore on

splitting). On satisfying above two conditions, the neighboring hybrid bucket is splitted and strings are distributed between itself and the empty bucket. According to it the parent trie node is updated.

Otherwise, if any of above condition is not satisfied i.e. either if immediate neighbors are not hybrid or if split will cause neighbor to become empty, then merging of empty bucket is not possible and it will be deleted. Another way to delete bucket is apply lazy deletion [9] as described above, and simply flag the bucket as having been deleted.

Algorithm 10: Delete in B-trie

B_Trie_Delete(Q,T)

Input: Q is a string, T is the address of first trie.

Output: String deleted / not present.

1. $if\ string\ is\ not\ present$

   $print\ \leftarrow not\ present;$

2. $else$

a. $char\ c = Extract\ the\ image\ of\ initial\ character\ from\ the\ string;$

b. $while(ptr\ is\ pointing\ to\ trie\ node)$

   $Remove\ the\ initial\ char\ from\ string\ Q;$

   $Repeat\ from\ step\ a;$

c. $If\ ptr == Null$

   $Delete\ Q\ from\ hash\ t;$

d. $If\ ptr\ is\ pointing\ to\ bucket$

   $if\ bucket\ is\ pure$

   $Remove\ the\ initial\ char\ from\ string;$

   $Search\ and\ remaining_string\ from\ bucket;$

$if\ bucket\ is\ hybrid$

$Delete\ the\ remaining_string\ from\ bucket;$

5. $if\ string\ becomes\ null\ before\ reaching\ bucket$

$Clear\ the\ end_of_string\ flag\ from\ trie\ or\ pure\ bucket;$

$Delete\ Q\ from\ hash\ table;$

6. Exit.

2.2.3. Example of insertion

First of all there is a need to initialize the B-trie.

2.2.3.1. Initialization of B-Trie

Initially B-Trie consists of a trie node and a bucket. i.e. one array of pointers and a bucket ranging from 'a' to 'z'. And all pointers of Trie are Null.

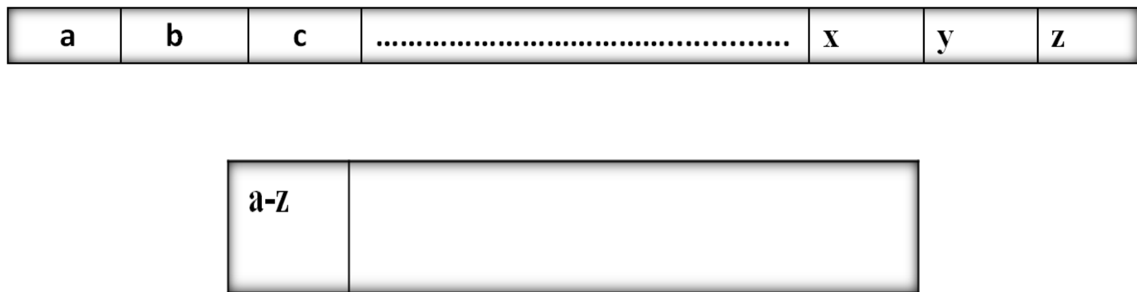


Figure 2.15: Initialization of B-trie

Consider we have to insert these strings in B-Trie:

cat

algorithm

computer

practice

caches

c

2.2.3.2. Simple insertion

Consider a initialized B-Trie and to insert a string “cat”. It will simply inserted into the first bucket. As it was hybrid bucket so complete string will get inserted into 2D array of bucket.

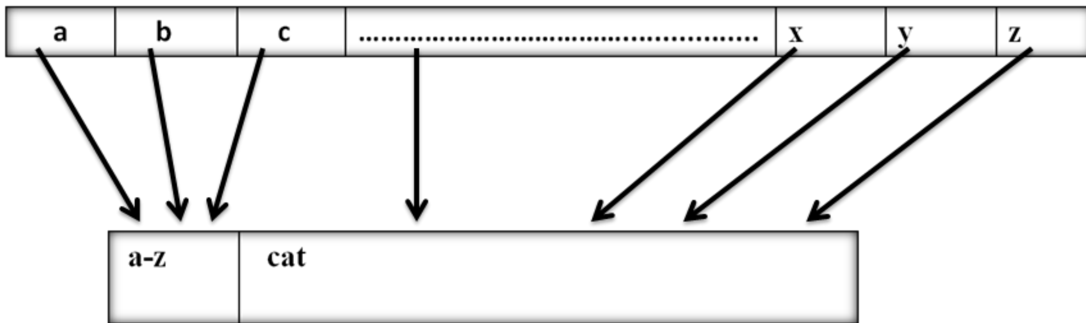


Figure 2.16: Insertion of string “cat”

Now, consider to insert string “algorithm” it will be checked if bucket is full as bucket is not full yet so string will simply inserted into 2D array of bucket.

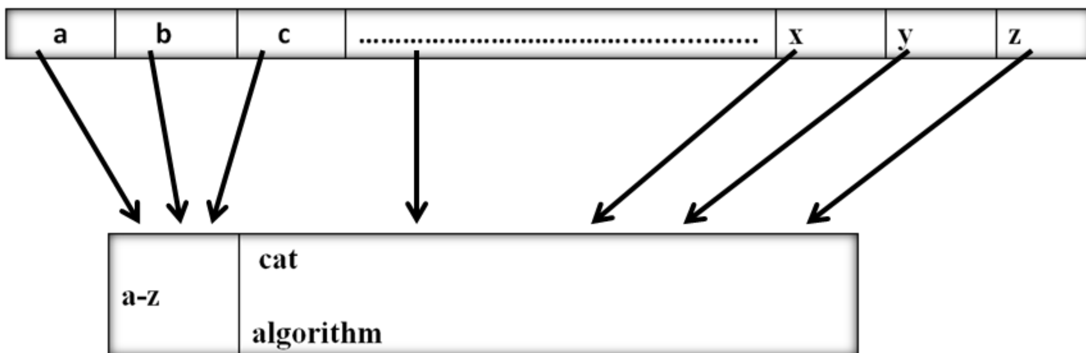


Figure 2.17: Insertion of string “algorithm”

2.2.3.3. Splitting of hybrid bucket

Now, when string “Computer” is to insert it is found that the bucket in which string “Computer” has to insert is full, so, there is need to split that bucket. For that split point is to found out. Split point is b in this case. So two buckets will be formed, one from a-b and second from c-z.

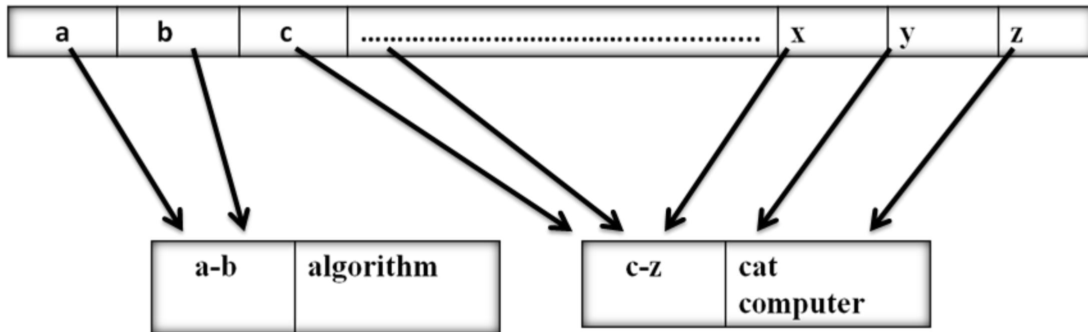


Figure 2.18: Insertion of string “computer”

2.2.3.4. Creation of pure bucket

When string “practice” has to insert it is found that bucket c-z is again full. Now, when split point will found out it will come c in this case i.e. pure bucket (a bucket in which all strings have same initials) c is needed to create. As it is a pure bucket initials of strings will be removed and strings in pure bucket c-c is at and omputer.

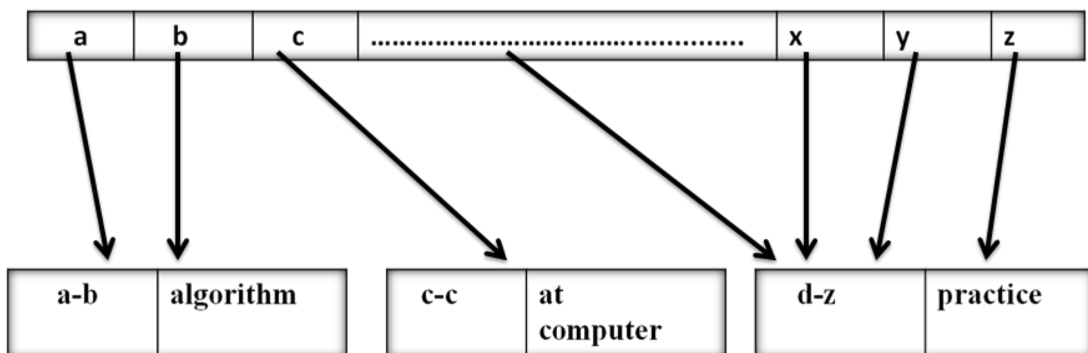


Figure 2.19: Creation of pure bucket

2.2.3.5. Splitting of pure bucket

When string “cache” is to insert, it’s location to insert is in pure bucket c-c. But c-c is already full. So, there is need to split pure bucket. While splitting pure bucket a trie node is added. Like in this example a trie node is added and split point is a again a pure bucket. Initials of remaining string will be removed i.e. ‘a’ will be removed. Now strings in pure bucket a-a are che and t. And a hybrid bucket b-z is created containing string omputer.

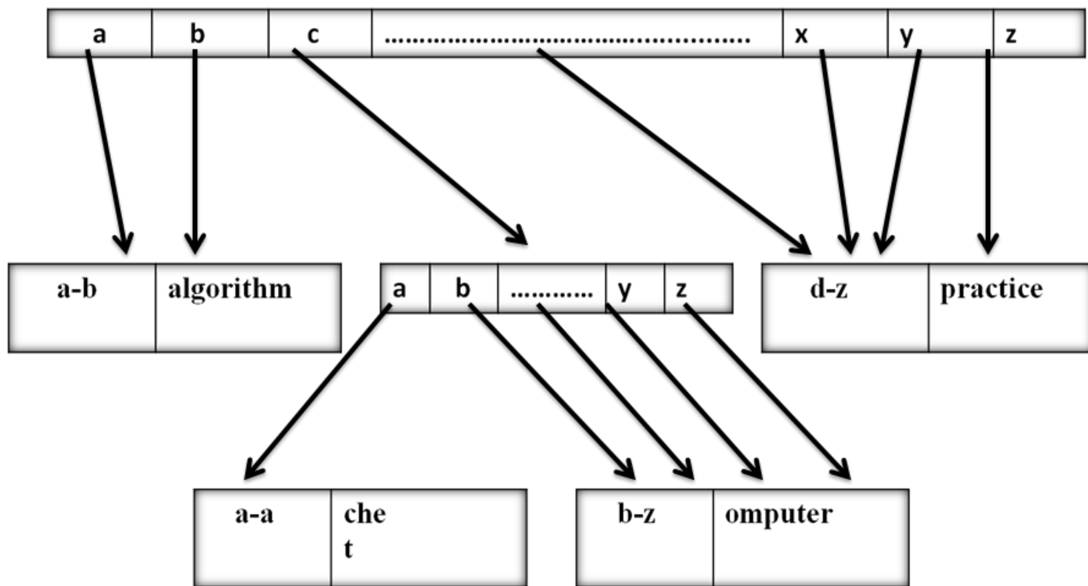


Figure 2.20: Splitting the pure bucket

2.2.3.6. Adding short strings

When short string “c” has to add then we will start moving down along B-Trie. But before reaching to any hybrid or pure bucket our string will get completed then this string will be added to hash table.

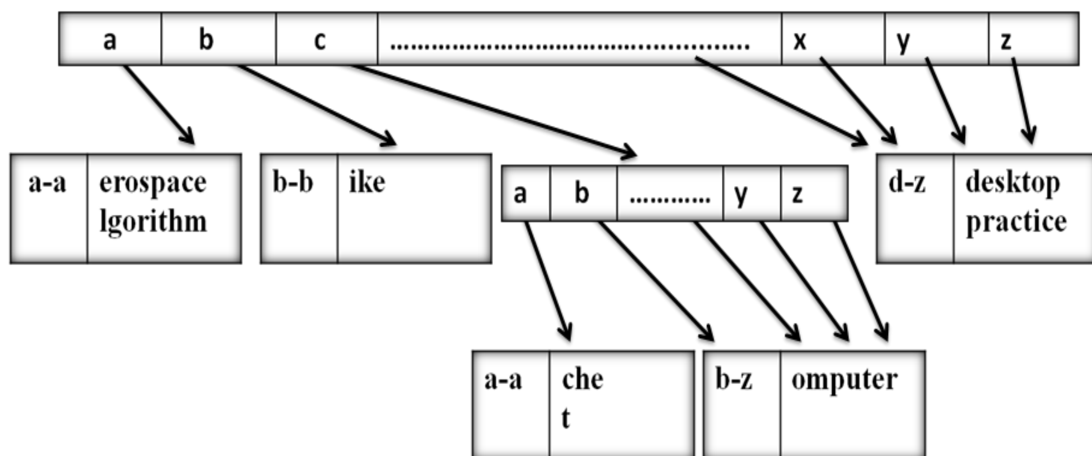


Figure 2.21: Adding Short string “c”

2.2.3.7. Deletion in B-trie

After deleting two strings desktop and practice

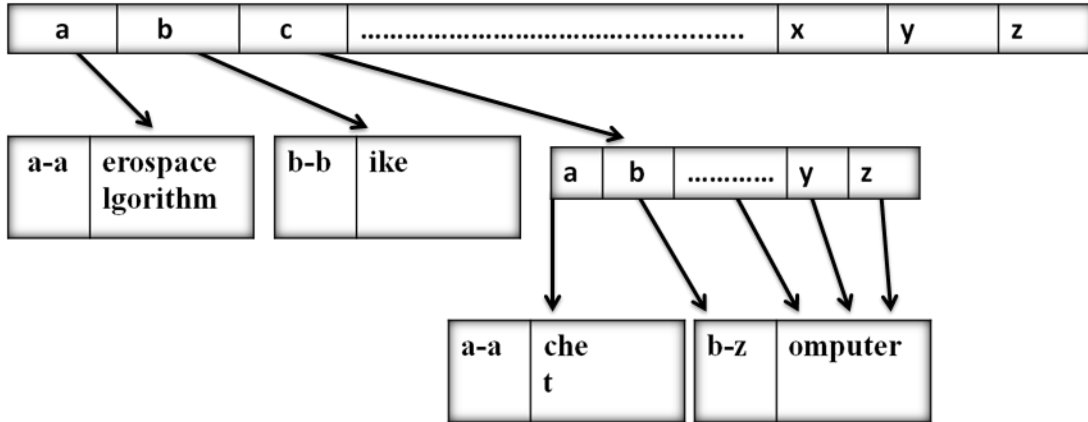


Figure 2.22: Deletion of strings “desktop” and “practice”

The pointers of d to z are null.

Chapter 3

Problem Statement

Data structure works best in main memory may not be successful on disk because of the difference in properties of main memory and secondary memory. The main two differences are, one is access time of the secondary memory is slower then access time of main memory and second is main memory is costlier than secondary memory. For example burst tries [8] and Hat tries [4] works better in main memory then on disk because of the creation of large number of buckets on every bucket split. The majority of trees and tries that are efficient in memory but cannot be directly mapped to disk without incurring high costs means updating cost in memory and then map the data in memory to disk [14].

Disk based data structure are the data structures, such as b-trees, which are stored and accessed from disk but processed only in memory. Updates are only present for an indefinite amount of time in memory. Whenever a change is done it is immediately forwarded to disk. It is not feasible to pick whole data structure in the memory because it will take a lot of time and also it is not possible to keep large structures in memory. Nodes of disk based data structures are needed to transfer to memory so that processing on these nodes can be done. In b-tree, nodes are needed to transfer to memory so that processing on these nodes can be done. So, there is a need of memory based data structure to store the data that is retrieved from disk.

Time complexity of disk based data structures depend upon number of disk accesses and processing time of CPU. Usually the prime focus is given on reducing the number of disk accesses but if the focus is transferred to reducing the processing time of CPU, it will also provide with beneficial results. To prevent the data loss from any kind of failure immediate modification is performed. Immediate modification of block to disk means, immediately writing the whole block to disk from memory whenever a change is done. It is necessary to protect the data in case of any failure.

Some of the data structures used till now are array, linked list, binary tree or b+-tree. These data structures have their own properties like Ordered Array allows binary search

[12] on itself but insertion and deletion on array leads to the shifting of the half of the elements in array. Searching in a Linked list can only be done in linear, but insertion and deletion is fast and mostly insertion and deletion is done after searching. Binary tree or B+ -tree takes logarithmic time for all the three basic operations but creating a b+-tree every time is more than linear.

There may be a structure which can hold the properties of array and linked list for one modification so that binary search is possible and there is no need to shift the array elements forward or backward because after modification block will be written to disk immediately.

Chapter 4
Proposed Structure

4.1. Structure

Structure consists of array of nodes. Each node consists of three parts value or data, child address and nextindex. Value or data contains the value of node and nextindex is index of next node. In this nextindex is taken as integer rather than pointer to save the space in some compilers.

```
struct node
{
	int value;
	int nextindex;
	long pointer*c; // c may be null pointer/disk
pointer
}pds[blocksize - 1];
```

Size of pds is equal to blocksize which is equal to 2t. A pds contains 2t pointers and 2t-1 values because 1st value is always null.

The algorithms of creating, searching, insertion, deletion and writing to disk are explained. Data is sorted in the nodes of b-tree so binary search can be used for searching. After modification, the list of structure doesn't look like a sorted array, but it holds the properties of linked list.

4.2. Algorithms for Structure

List (pds) is loaded from the values taken from the buffer. As all values are sorted so nextindex of a node will be just next adjacent location. In creation of list value and nextindex of all nodes are filled till buffer is not empty.

Algorithm 11: Creation of List (pds)

$Disk_read(pds, buffer)$

Input: Buffer containing values from disk, empty List.

Output: Filled List pds with values taken from buffer.

1. $Set\ i \leftarrow 1;$
2. $pds[0].c \leftarrow buffer.firstpointer\ ;\ pds[0].nextindex \leftarrow 1;$
3. $pds[i].value \leftarrow buffer.nextitem;$
4. $pds[i].pointer \leftarrow buffer.nextpointer;$
5. $pds[i].nextindex \leftarrow i + 1;$
6. $i + +;$
7. $if\ (buffer! = empty)$

   $Repeat\ from\ step\ 2;$
8. $pds[i - 1].nextindex \leftarrow -1;$
9. Exit

List is the array of nodes in sorted order, so binary search[12] can be performed on the List. In this case location has to found where item is present or not. Algorithm of binary search on List (pds) is

Algorithm 12: Binary Search on structure

$Binary_Search(pds, item)$

Input: pds with n values, item

Output: Location loc where item can be inserted

1. $Set\ lb \leftarrow 1\ ; Set\ ub \leftarrow n;$
2. $Set\ mid \leftarrow lb + ub/2\ ;$

3. $if(pds[mid].value == item)$

$loc \leftarrow mid;$

$else$

$if(pds[mid].value > item)$

$ub \leftarrow mid - 1;$

$else$

$lb \leftarrow mid + 1;$

4. $If(lb == ub)$

$loc \leftarrow lb;$

$else$

$repeat\ from\ step\ 2;$

5. $if(pds[loc].value == item)$

$print \leftarrow found;$

$else$

$print \leftarrow not\ found;$

6. $Exit$

In case of insertion in pds item will be inserted at end and the nextindex of location will be changed to location where item is inserted. And nextindex of location where item is inserted will be location (loc) find out by binary search.

Algorithm 13: Insert new element

Insert(pds, item, loc)

Input: pds with n values, item to insert, location where to insert item

Output: pds with inserted value

1. $loc = Binary\ Search(pds, item);$
2. $pds[n[pds] + 1].value \leftarrow item;$
3. $pds[n[pds] + 1].nextindex \leftarrow loc;$
4. $pds[loc].nextindex \leftarrow n[pds] + 1;$
5. $n[pds] \leftarrow n[pds] + 1;$
6. Exit

In case of deletion in pds from location loc (find out by binary search algorithm given above) just that node is deleted softly i.e. as pds is array of nodes, but there is no need of shifting whole array, just changes to nextindex of previous element nextindex to nextindex of loc.

Algorithm 14: Deletion

Delete(pds, loc)

Input: pds with n values, location loc to delete.

Output: modified pds.

1. $loc = Binary\ Search(pds, item);$
2. $Set\ pds[loc - 1].nextindex \leftarrow pds[loc].nextindex;$
3. $n[pds] \leftarrow n[pds] - 1;$
4. Exit

After modification there is a need to put the whole block to disk again

---

Algorithm 15: Writing List(pds) on disk

Disk_write (pds)

Input: pds.

Output: data written to disk.

1. $Set\ i \leftarrow 1;$
2. $buffer.firstpointer \leftarrow pds[0].c;$
3. $buffer.nextitem \leftarrow pds[i].value;$
4. $buffer.pointer \leftarrow pds[i].pointer;$
5. $i \leftarrow i.nextindex;$
6. $If(i < n[pds])$

   $repeat\ from\ step\ 2;$
7. Exit

4.3. Illustrations

Let's take an illustration that a block at the disk contains the value $\boldsymbol{\alpha_2}$, $\boldsymbol{\alpha_3}$, $\boldsymbol{\alpha_4}$, $\boldsymbol{\alpha_5}$ and it has one vacant space to store an element. Suppose using the b-tree algorithm we reached at that block which is a leaf. So below are the operations on that block if we want to search, insert or delete an element. In the below illustrations the child pointer part is not shown. This example is showing the process on one node which is assumed to be leaf node.

4.3.1. Creating a List (pds)

Insertions in the pds are similar like an array. Array size is similar to the size of node of b-tree and Node size should be equal to the size of the block of disk. Iterate through a simple loop we can fill values in array of structures because the elements are in sorted order. This operation takes $O(n)$ time. It is because insertion time in array is $O(1)$ and in case of insertions in structure there are n elements to insert.

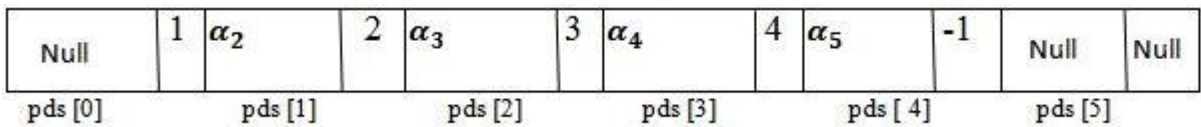


Figure 4.1: After buffering the block in pds

4.3.2. Searching

Suppose user want to search $\boldsymbol{\alpha_2}$**.** Binary search can be applied initially because it is an array. So now that the elements in the structure are 4. So lower bound =1 and upper bound=4. Then calculation of mid is comes to be 2. Now the value at location 2 is $\boldsymbol{\alpha_3}$**.**
$\boldsymbol{\alpha_2} < \boldsymbol{\alpha_3}$ then upper bound = 2. Again calculation of mid comes to be 1. The value at location 1 is $\boldsymbol{\alpha_2}$ which the required value is**.** The time complexity of searching is $O(\log n)$ because binary search is applied on pds.

4.3.3. Insert

Suppose user want to insert $\boldsymbol{\alpha_6}$. First the program checks that the element should not be already present. Position can be found by binary search because it is an array because binary search can be used to find the location to insert the element. Assume position founded is 2.

```
temp = pds[pos - 1].nextindex; // 2
pds[pos-1].nextindex = pds[n].nextindex; // 5
pds[n].nextindex = temp; // 2
```

The complexity of insert operation is $O(\log n)$because it contains search as sub operation, searching takes $O(\log n)$ time and insertion takes $O(1)$ time. Hence overall complexity becomes $O(\log n)$.

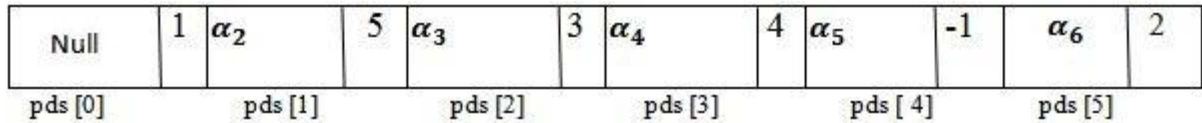


Figure 4.2: After Insertion of α_6

4.3.4. Delete

Suppose user want to delete $\boldsymbol{\alpha_2}$. First the program checks that the element should be already present. Position can be found by search method define above. Assume position founded is 1.

```
pds[position-1].nextindex = pds[position].nextindex; // 2
```

The complexity of delete operation is $O(\log n)$ because it contains search as sub operation, searching takes $O(\log n)$ time and deletion takes $O(1)$ time. Hence overall complexity becomes$O(\log n)$.

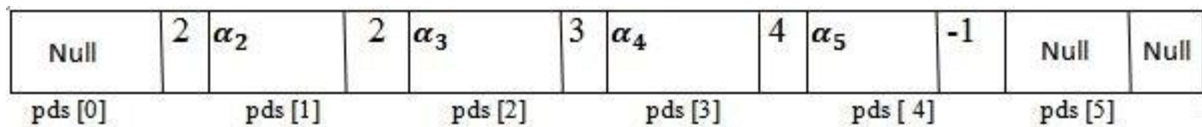


Figure 4.3: After Deletion of α_2

4.4. Comparison of various data structures

Comparison of various data structures with proposed structure for search, insert and delete are given in Table 1.

Table 4.1: Comparison of various data structures

Data structures	Search	Insert	Delete	Creation
Array	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B+ tree / Binary tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$> O(n)$
Proposed Structure	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

Chapter 5
Proposed B-tree Algorithms

5.1. Proposed algorithm of search in B-tree

Searching in B-tree means to find the location of an element in B-tree. Searching is also done in case of insertion to find proper location to insert and element.

Algorithm 16: Search in B-tree using binary search

$B_tree_Search(pds, item)$

Input: B-tree with n nodes, item to find location
Output: Location loc where item is found

1. $Set\ lb \leftarrow 1; Set\ ub \leftarrow n[pds], Set\ loc \leftarrow 1;$
2. $Set\ mid \leftarrow (lb + ub)/2;$
3. $if(pds[mid].value == item)$

    $loc \leftarrow mid;$

    $else$

    $if\ (pds[mid].value > item)$

    $ub \leftarrow mid - 1;$

    $else$

    $lb \leftarrow mid + 1;$

4. $if(lb == ub)$

    $loc \leftarrow lb;$

    $else$

    $repeat\ from\ step\ 2;$

5. $if(loc \leq n[pds] \&\&\ item < pds[loc].value)$

$return \;\; (pds, loc);$

6. $if(leaf[pds])$

$return\, nil;$

$else$

$if \;\; (pds[1].value > item)$

$Disk_read(pds[0].c);$

$else$

$Disk_read(pds[loc].c);$

$return\, B_tree_Search(pds[loc].c, item);$

In above algorithm, searching in B-Tree is done using binary search on individual nodes. Root node is taken first and binary search is applied on elements in root node when loc find the child node of that location is read and preceded this way until exact location is found or reached at leaf node. Therefore, if node size is 100 than previous algorithm may takes 100 operations which is of the order $O(n)$but the modified version will take $O(\log n)$ comparisons.

5.2. Proposed algorithm of insert in B-tree

Insertion in B-tree varies with number of elements in node i.e. whether node where element has to insert is full or non- full. In both cases there is difference of algorithms which is provided in algorithms given below.

Algorithm 17: Insert in B-tree

$B_Tree_Insert_Nonfull(pds, item)$

Input: pds, item.
Output: pds with inserted node.

```
1. Set loc ← n[pds].
2. Set lb ← 1; Set ub ← n[pds];
3. Set mid ← (lb + ub)/2;
4. if (pds[mid].value = item)
       loc ← mid;
   else
       if (pds[mid].value > item)
           ub ← mid − 1;
       else
           lb ← mid + 1;
   if(lb == ub)
       loc ← lb;
   else
       repeat from step 3;
5. if leaf[pds]
       Set pds[n[pds] + 1].value ← item;
       Set pds[n[pds] + 1].nextindex ← loc;
       Set pds[loc].nextindex ← n[pds] + 1;
       Disk_write(pds);
   else
       if(pds[1].value > item)
           pds1 = Disk_read(pds[0].c);
```

$else$

$pds1 = Disk_read(pds[loc].c);$

$if(n[pds1] == blocksize - 1)$

$B_Tree_Split_Child(pds, loc, pds1);$

$if(pds[1].value > item)$

$B_Tree_Insert_Nonfull(pds[0].c, item);$

$else$

$B_Tree_Insert_Nonfull(pds[loc].c, item);$

Above algorithm of insertion is for non full B-tree means node can consume one more node. In this algorithm first it is checked if present pds is at leaf node or internal node. If pds is at leaf node then simply binary search is implemented on values of leaf nodes and item in inserted at proper location. Otherwise if pds is an internal node then location is found to get proper child node and similarly proceeding in same way until leaf node is find.

So, if the node size is 100 than previous algorithm may take 100 comparisons and 100 shifting operations to insert when node is non full but modified version will take $O(\log n)$ comparisons and no shifting process.

5.3. Proposed algorithm of split in B-tree

The splitting of node in B-tree is done when node element has to insert is full. In that case the node is spilt to make two nodes so that there becomes space to insert element.

Algorithm 18: Split in B-tree

$B_Tree_Split_Child(pds_parent, loc, pds_child)$

Input: $pds - parent, loc, pds - child.$
Output: Successfully written on disk.

---

1. $pds_newchild \leftarrow Allocate_Node()$;

2. $Set\ leaf[pds_newchild] \leftarrow leaf[pds_child]$;

3. $Set\ n[pds_newchild] \leftarrow blocksize/2 - 1$;

4. $Set\ pds_newchild[0].c \leftarrow pds_child[blocksize/2].c$;

5. $Set\ pds_newchild[0].nextindex \leftarrow 1$;

6 $for\ i \leftarrow 1{:}blocksize/2 - 1$

   $Set\ pds_newchild[i].value \leftarrow pds_child[blocksize/2 + i + 1].value$;

   $if(leaf[pds_child])$

      $Set\ pds_newchild[i].c \leftarrow pds_child[blocksize/2 + i + 1].c$;

   $pds_newchild[i].nextindex \leftarrow i + 1$;

7. $n[pd_child] \leftarrow blocksize/2 - 1$;

8. $pds_parent[n[pds_parent] + 1].c \leftarrow pds_newchild$;

9. $pds_parent[n[pds_parent] + 1].value \leftarrow pds_newchild[blocksize/2].value$;

10. $pds_parent[n[pds_parent] + 1].nextindex \leftarrow loc$;

11. $pds_parent[loc].nextindex \leftarrow n[pds_parent] + 1$;

12. $n[pds_parent] \leftarrow n[pds_parent] + 1$;

13. $Disk_write(pds_child)$;

14. $Disk_write(pds_newchild)$;

15. $Disk_write(pds_parent)$;

16. Exit

The necessary condition of split is node should be full there is no empty space in node. Split operation splits the values of pds-child into two nodes. 1st is same as pds-child and

another is pds-newchild. And the split operation moves the median key of pds-child to its parent which is pds-parent in above algorithm. The above algorithm reduced the shifting in parent node.

Suppose, if parent node contains 100 elements then previous algorithm will shift 50 on an average. In above algorithm the node is inserted at the end only nextindex field is replaced.

Chapter 6
Conclusion and Future work

An algorithm is attempted to propose for those disk based data structures which uses the concept of immediate modification. As the time complexity of disk based data structures depends upon number of disk accesses and CPU processing time. In this thesis, CPU processing time is tried to reduce. For single search, insert and delete operation, array takes $O(n)$, linked list takes $O(n)$ but proposed algorithm using defined structure as shown in Figure 4.1 will take $O(\log n)$. Also the b-tree formation of a node takes $O(\log n)$ but to create a b-tree is more than $O(n)$. Hence proposed algorithm shows better results in case of immediate modification of block on disk. In future we can try to reduce the complexity in terms of reducing number of disk accesses.

References

[1] Amir Y., “Operating Systems 600.418 File System (cont.) and Disk Management”, http://www.cs.jhu.edu/~yairamir/cs418/os8/tsld019.html.

[2] Arge, L., “External memory data structures”, *Handbook of Massive Data Sets*, pp. 313–357, 2002.

[3] Askitis N. and Zobel J., “B-tries for disk-based string management”, *J. of The VLDB Journal*, vol. 18, pp.157–179, 2009.

[4] Askitis N. , Sinha R., “HAT-trie: A Cache-conscious Trie-based Data Structure for Strings” *ACSC '07 Proceedings of the thirtieth Australasian conference on Computer science,* vol. 62 pp. 97-105,*2007.*

[5] Comer D., “The Ubiquitous B-tree”, *ACM Computer Survey*, vol. 11, no.2, pp. 121-137, 1979.

[6] Dinda P., “Project C: B-tree”, http://www.cs.northwestern.edu/~pdinda/db-f04/projectc.pdf.

[7] Ferragina P., Grossi R.,” The String B-Tree: A New Data Structure for String Search in External Memory and its Applications” *J. ACM* , vol. 46 no. 2, pp. 236-280,1998.

[8] Heinz S., Zobel J. and Hugh E. Williams, “Burst tries: A fast, efficient data structure for string keys”, *ACM Trans. Inf. Systems*, vol. 20, no. 2, pp.192–223, 2002.

[9] Jannink J., “Implementing deletion in B+-trees”. *Proc. ACM SIGMOD Int. Conf. Manag. Data*, vol. 24, no. 1, pp.33–38, 1995.

[10] Knessl C. and Szpankowski, W., “A note on the asymptotic behavior of the height in b-tries for b large”. *The Electron. Journal of Combinatorics*, vol. 7, R39, 2000.

[11] Neubauer P.,”B-Trees: Balanced Tree Data Structures”, http://www.bluerwhite.org/btree/

[12] Pfaff B., “An Introduction to Binary Search Trees and Balanced Trees”, *Libavl Binary Search Tree Library*, vol.1: Source Code Ver. 2.0.2, pp. 19-20, 2004.

[13] Rivest R. L., Cormen T. H. and Leiserson C. E. , *Introduction to Algorithms,* 2nd ed. England: The MIT Press, McGraw- Hill, 2001.

[14] Sahni, S. "Handbook of DATA STRUCTURES and APPLICATIONS", *Chapman & Hall/CRC Computer And Information Science Series*, ch.28 pp.1-15, 2005.

[15] Vitter J.S, "External memory algorithms and data structures", *Foundations and Trends® in Theoretical Computer Science*, vol. 2 no. 4, pp. 305-474, 2008.

[16] http://www.slideshare.net/grifinder/datastructure

[17] http://cis.stvincent.edu/carlsond/swdesign/btree/btree.html

[18] http://slady.net/java/bt/view.php?w=800&h=600

[19] http://www.seanster.com/BplusTree/BplusTree.html

List of Publications

Communicated

[1] Sahil Singla, Ravinder Kumar. "An Efficient Algorithm for Storage and Retrieval of Disk based Data Structures"; Journal of Computer Science and Technology. (Impact factor: 0.656)