1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

pTrie: Paginated Trie Hashing

by

Samuel Yao Effah

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute of Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario
August, 1993

© Copyright
1993, Samuel Yao Effah

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-89820-8

Canada

Name

SAMUEL YAO EFFAH

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0291
Literature 0401
General 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0316
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0324
Anthropology 0326
Archaeology 0327
Cultural 0310
Physical 0272
Business Administration 0770
General 0454
Accounting 0338
Banking 0385
Management 0501
Marketing 0503
Canadian Studies 0505
Economics 0508
General 0509
Agricultural 0510
Commerce-Business 0511
Finance 0358
History 0366
Labor 0351
Theory 0578
Folklore 0366
Geography 0351
Gerontology 0578
History 0578
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
General 0938
Criminology and Penology 0631
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family Studies 0630
Industrial and Labor Relations 0700
Public and Social Welfare 0344
Social Structure and Development 0709
Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology 0306
Biology 0287
General 0308
Anatomy 0309
Biostatistics 0379
Botany 0329
Cell 0353
Ecology 0369
Entomology 0793
Genetics 0410
Limnology 0307
Microbiology 0317
Molecular 0416
Neuroscience 0433
Oceanography 0821
Physiology 0778
Radiation 0472
Veterinary Science 0786
Zoology 0760
Biophysics 0760
General 0425
Medical 0996

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0386
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0494
Pharmaceutical 0495
Physical 0754
Polymer 0405
Radiation 0605
Mathematics 0986
Physics 0606
General 0608
Acoustics 0748
Astronomy and Astrophysics 0607
Atmospheric Science 0798
Atomic 0759
Electronics and Electricity 0609
Elementary Particles and High Energy 0610
Fluid and Plasma 0752
Molecular 0756
Nuclear 0611
Optics 0463
Radiation 0346
Solid State 0984
Statistics 0984
Applied Sciences 0346
Applied mechanics 0984
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal 0790
System Science 0428
Geotechnology 0796
Operations Research 0795
Plastics Technology 0994
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,

pTrie: Paginated Trie Hashing

submitted by

Samuel Yao Effah

in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

August, 1993

Abstract

The thesis proposes a new method of storing a trie structure on secondary storage. The most notable data structures that have been in use and continue to be used for indexing on secondary storage are the B-tree, its variants and hashing schemes. Trie structures as file indexes, are generally maintained in main memory. They are not retained on secondary storage except for special designed schemes such as PATRICIA and compressed tries. One known such structure is the trie hashing scheme of Litwin. Since its introduction, several researchers have proposed ways of mapping the trie onto disk in order to achieve durability and recoverability in the event of system failure. Maintaining the trie on secondary storage allows its use for very large databases beyond the capacity of main memory. This thesis presents a new file structure called Paginated Trie Hashing (or pTrie) that combines the merits of the existing trie hashing structures and the dynamic and balanced characteristics of the B⁺-tree. In the work leading to implementation of the pTrie, the rbTrie was developed as an improved variant of Litwin's original trie hashing scheme. The Paginated Trie Hashing scheme is a dynamic balanced index method for both long variable length keys and fixed length keys. It maintains insertion, deletion and retrieval times that is each logarithmic in the size of the number of keys stored and has a guaranteed storage utilization of at least 50%.

Acknowledgments

I am greatly indebted to my supervisor and friend, Dr. Ekow Otoo whose invaluable criticisms and support led to the quality of this thesis. I also thank my fellow graduate students who made Carleton campus a congenial atmosphere for academic studies. I am also indebted to the government and people of the Republic of Ghana who partially supported my studies at Carleton University. I am especially grateful to my mother Akuah Nsiah, wife Margaret and sons Raphael and Jeffrey for their continual support and encouragement.

**This work is dedicated to my son Jeffrey
who had to wait for a period of nearly two years just to see a daddy
and
my mom**

Contents

Abstract	iii
Acknowledgments	iv
List of Tables	ix
List of Figures	x
List of Algorithms	xiii
1 Introduction	1
1.1 Objective	2
1.2 Research results	3
1.3 Organization	4
2 Related work	5
2.1 Binary Search Tree	5
2.1.1 Dynamic set operations	5
2.1.2 Balanced binary trees	7
2.2 B-tree and its variants	11
2.2.1 Basic B-tree	12
2.2.2 B ⁺ -tree	19
2.2.3 Prefix B-tree	21

3	Trie hashing	28
3.1	The basic trie hashing scheme	28
3.1.1	Insertion	32
3.1.2	The concept of split key	33
3.1.3	Deletion	36
3.1.4	Range query	36
3.2	Reconstruction and optimization of trie hashing	39
3.3	Locally balanced compact trie	43
3.3.1	Compact binary trie	43
3.3.2	Balanced compact trie hashing	44
3.4	Multilevel trie hashing	46
3.4.1	Choosing a split node	47
3.4.2	MLTH splitting	47
3.5	rbTrie	48
3.5.1	Insertion	49
3.5.2	Deletion	53
4	pTrie: Paginated trie hashing	56
4.1	Key search	57
4.2	Insertion	59
4.2.1	Page splitting	60
4.2.2	Further compression	61
4.3	Range query	70
4.4	Deletion	70
4.4.1	Redistribution and Concatenation	74
4.5	Implementation	78
4.5.1	Page organization	78
4.5.2	Other aspects	79

5	Performance evaluation	81
5.1	Comparison between rbTrie and other tries	81
5.1.1	rbTrie versus compact trie	82
5.1.2	rbTrie versus other trie balancing strategies	84
5.1.3	Black height of rbTrie	85
5.2	pTrie comparison with prefix B-tree	86
5.2.1	Size of partial separators	87
5.2.2	Height of pTrie	89
5.2.3	Other parameters	90
5.2.4	Remark	92
5.3	pTrie comparison with other trie hashing structures	94
6	Application of pTrie	96
6.1	The pTrie as an English spelling dictionary	96
6.2	The pTrie as a searching tool in a text editor	98
6.2.1	Semi-infinite strings	99
6.2.2	Indexing points	100
6.2.3	Implementation	100
7	Conclusion and future work	103
	References	107
A	Graphs of rbTrie and compact trie comparison	112
B	Tables of pTrie and B-trees comparison	116

List of Tables

3.1	The KWIC index of size 35.	55
4.1	The 60 frequently used English words with 3 or more digits.	64
4.2	The split information of the example pTrie.	66
5.1	Compact trie and rbTrie comparison for $c = 10$	83
5.2	Compact trie and rbTrie comparison for $c = 20$	83
5.3	The height of rbTrie, for $c = 20$	85
5.4	The sizes of the separators in the pTrie and the prefix B-tree.	88
5.5	The size of digit value of the pTrie, for $c = 20$	89
B.1	Comparison for page size of 256 bytes.	117
B.2	Comparison for page size of 512 bytes.	118
B.3	Comparison for page size of 1024 bytes.	119

List of Figures

2.1	An example of a binary search tree with 12 keys.	6
2.2	Demonstration of right rotation in a binary tree.	9
2.3	Demonstration of double rotation in a binary tree.	9
2.4	Transformation from a binary search tree to a B-tree.	13
2.5	Insertion of key “come” into a full page in the B-tree.	15
2.6	An illustration of a page split in the B-tree.	16
2.7	A B-tree constructed from randomly inserted keys.	16
2.8	An example of a B ⁺ -tree constructed from random keys.	19
2.9	(a) An example of how a page is split in the B ⁺ -tree, (b) A leaf node of a B-tree.	22
2.10	An example of a simple prefix B-tree and a prefix B-tree constructed from 21 random keys.	21
3.1	An example of a trie hashing structure.	29
3.2	An illustration of how an insertion is accomplished in the trie hashing.	33
3.3	An example of trie hashing insertion procedure.	34
3.4	An example of how the split key is selected in the trie hashing during a split.	35
3.5	An illustration of how the reconstruction algorithm produces an equivalent trie.	40

3.6	An example of an equivalent trie produced by the reconstruction algorithm.	41
3.7	An illustration of how local transformation is performed during the application of the optimization algorithm a trie.	42
3.8	An optimized trie of the trie in Figure 3.6.	42
3.9	Difference between the basic trie and the compact binary trie during a split.	44
3.10	An example of a compact binary trie.	44
3.11	Illustration of RL and LR migrations of the BCTH.	45
3.12	An example of a balance compact binary trie.	46
3.13	Case 1 of inserting into a rbTrie.	52
3.14	Case 2 of inserting into a rbTrie.	53
3.15	Cases 3 and 4 of inserting into a rbTrie.	54
3.16	An example of a rbTrie from the insertion of random keys.	55
4.1	An illustration of how the page split operation functions for a pTrie. .	62
4.2	An illustration of how further compression is performed during the insertion of an entry into a page.	63
4.3	An example of a pTrie constructed from the keys in Table 4.1.	64
4.4	The construction of the example pTrie in Figure 4.3.	65
4.5	An illustration of how page merging is performed in the pTrie.	73
4.6	An illustration of page redistribution in the pTrie.	75
5.1	Number of pages allocated comparison, random insertion.	91
5.2	Number of pages allocated comparison, ascending insertion.	91
5.3	Execution time comparison, random insertion.	93
5.4	Execution time comparison, ascending insertion.	93
6.1	The pTrie file for a searching tool.	102

A.1	Average height comparison $c = 10$, random insertion.	113
A.2	Maximum height comparison $c = 10$, random insertion.	113
A.3	Average height comparison $c = 10$, sorted insertion.	114
A.4	Average height comparison $c = 20$, random insertion.	115
A.5	Maximum height comparison $c = 20$, random insertion.	115

List of Algorithms

2.1	The search algorithm of the prefix B-tree.	27
3.1	The search algorithm of the trie hashing.	31
3.2	The range query processing algorithm for trie hashing.	38
3.3	The insertion algorithm of the rbTrie.	50
4.1	The search algorithm of the pTrie.	59
4.2	The insertion algorithm of the pTrie.	67
4.2	cont.	68
4.3	The insert-in-page() function of the pTrie insertion algorithm.	69
4.4	The deletion algorithm of the pTrie.	76
4.4	cont.	77
4.5	The delete-from-page() function of the pTrie deletion algorithm.	78

Chapter 1

Introduction

Activities carried out in the use of computers have become predominantly that of information management, i.e. the storage and use of large collections of data. Random access memory (RAM) is generally expensive, volatile and too small to hold entire databases. Secondary storage devices such as disks, can store more information than RAM and are relatively cheap compared with RAM. There are however disadvantages to using cheap secondary devices — the retrieval time is high compared with primary memory.

One of the early storage organization mechanisms required the use of sequential search to manipulate an index. Sequential search requires at most n comparisons if there are n records in the file and, on the average, requires $n/2$ comparisons. Therefore sequential search is of order $O(n)$. The divide-and-conquer search strategy, known as binary search, is the immediate improvement of the sequential search.

The binary search algorithm locates a key in a sorted file of n records in time $O(\log_2 n)$. Binary search on secondary storage has some limitations. For example, it requires many disk accesses, and as such, its search time performance in a very large file is unacceptable. The other problem of the binary search relates to keeping the index in sorted order. For a very large index, this maintenance can be very expensive.

To resolve the organization problem involved in a binary search, a binary search tree is used instead. The binary search tree has time $O(\log_2 n)$ for the search, insert and delete operations. However like its predecessors, it is memory resident and is generally not used to store very large databases. The major storage structures that enable very large amount of data to be stored on external storage devices and be manipulated efficiently are the B-tree and its variants. The search, insert and delete times of the B-tree are $O\left(\log_{\lceil \frac{m}{2} \rceil} \left(\frac{n+1}{2}\right)\right)$ for a tree with order m .

1.1 Objective

Trie hashing [31], is one of the fastest known search strategies for long alphanumeric strings. But its index is maintained in primary memory. Like all memory resident structure, it has some drawbacks. For example, as the index grows very large (corresponding to a very large database), its size might exceed memory capacity. The trie hashing structure has the propensity to degenerate if records are presented in sorted order for insertion. In addition, the structure will be lost in the event of system failure or the termination of the search/insert algorithm. Several researchers [35; 47; 51] have contributed to reducing the drawbacks of the trie structure. The balanced strategies suggested so far are either non-dynamic or not global with respect to the entire trie.

The aim of this thesis is to:

1. develop an appropriate strategy to balance the trie hashing structure while leaving the trie function invariant.
2. develop a scheme for retaining the trie on secondary storage.

Trie hashing is a dynamic bucket hashing scheme which consists of a binary trie resident in memory and a set of data buckets (or pages) resident on an external storage device. The binary trie, or simply the trie, is an index structure for the buckets during a key search. The retrieval of a key requires at most one disk access since the trie structure is memory resident and only the data bucket, that the search

(traversal) leads to, is retrieved. Not all searches in the trie however lead to buckets. The load factor of the data file is about 70%, and the trie grows by splitting buckets that overflow in a manner similar to the B-tree's. The load factor of a nonempty data page (or bucket) is the number of items (keys) stored in the data page divided by the size (number of slots) of the data page. The load factor of a data file is the average of the load factors of all data pages in the file.

In this thesis we present a scheme known as rbTrie, or red-black trie, that resolves the degeneracy problem of the trie hashing. The rbTrie balances the trie structure dynamically during key insertion in a manner analogous to the red-black balancing of a binary search tree.

The second major algorithm developed in this thesis is the pTrie or paginated trie hashing. The pTrie addresses and resolves both the degeneracy and durability problems of basic trie hashing. The pTrie is a pagination of the trie hashing structure into pages as in the B-trees. The performance of pTrie is only matched by the prefix B-tree [7], and it outperform most of the existing tree based file structures. The pTrie combines the advantages of digital searching, key compression and the B-tree's.

1.2 Research results

The implementation strategy to balance the trie hashing structure was achieved using the red-black's [23] balancing strategy, and is called rbTrie. The rbTrie's search, insert and deletion times compare with Litwin's trie when records are presented in random order. However, if the records are presented in sorted order, the search, insert and deletion performance of rbTrie is better than that of Litwin's trie.

The second objective is met by implementing the paginated trie hashing or pTrie. A page within a pTrie is organized with variable length strings. Two known one-dimensional secondary storage structures that compare with the pTrie are the prefix B-tree and PATRICIA. The pTrie combines both the structural characteristics and advantages of the B-tree and the trie hashing file structures. The pTrie solves both of the problems of the basic trie. These are that, it is balanced and it can easily be

protected against system failure. Due to its structural nature, pTrie can be used in applications where tries and B-trees perform well. For example, the B-tree cannot be used for storing a text database without undue modification, but tries (including pTrie) are good structures for text databases. The two major results achieved in this thesis form the topic of the paper Paginated Trie Hashing under preparation.

1.3 Organization

The following chapters of this thesis present a detailed discussions of the structures and results that are mentioned above. Chapter 2 is devoted to the review of some of the relevant structures whose concepts are used in the achievement of the goals of this thesis. It includes the binary search tree, and B-tree and its variants. Chapter 3 presents the Litwin's trie hashing structure and the various improvements proposed by other researchers. The rbTrie strategy is also presented in Chapter 3. The paginated trie hashing scheme (pTrie) is presented in Chapter 4. The performance of the pTrie is measured and compared with its competing file structures namely the simple prefix B-tree and the prefix B-tree in Chapter 5. Chapter 6 explores some applications areas of the pTrie structures. The conclusions of the study are stated in Chapter 7. Chapter 7 also proposes some further areas of research.

Chapter 2

Related work

2.1 Binary Search Tree

The *binary search tree* is the underlying structure of the trie hashing scheme. Knuth [27] defined binary tree as a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the *root* of the tree and the other two subsets are themselves binary trees, called the *left* and *right* subtrees of the original tree. A left or right subtree can be empty, and each element of a binary tree is called a *node* of the tree.

A binary search tree is a binary tree in which all keys stored in the left subtree of a node x are less than the key stored at x and all keys stored in the right subtree of x are greater than the key stored at x . This property, called the *binary search tree property*, holds for every node in the binary search tree including the root. An example of a binary search tree is shown in Figure 2.1. The figure is created from the insertion of the following random keys: *not*, *have*, *this*, *no*, *some*, *which*, *can*, *any*, *part*, *do*, *stand*, and *kind*.

2.1.1 Dynamic set operations

If the cost of an operation in the tree is the number of comparisons done, then the basic dynamic operations of data structures which are search, insert, delete,

Figure 2.1 An example of a binary search tree with 12 keys.

maximum and minimum briefly discussed in this section are performed in the binary search tree with a cost that is proportional to the height of the tree.

Given a pointer to a tree rooted at node x , to search for a key k , k is compared with the root item. If $k = \text{key}(x)$ then the search is successful, else if $k < \text{key}(x)$ the left subtree is chosen otherwise the right subtree is chosen for the next comparison. At the chosen subtree, the same comparison is performed. If the search ends in a *nil* node, then the search is unsuccessful. The nodes encountered during the recursion form a path from the root downwards and is bounded by $O(h)$, where h is the height of the tree.

To insert into a binary search tree, we first determine where the value say v , to be inserted logically appears with respect to the binary search tree property. Thus we perform a search and at the leaf node where the search fails (assuming v is not in the tree, otherwise an attempt is being made to insert a duplicate key) the new node z for which $\text{key}(z) = v$, $\text{right}(z) = \text{nil}$ and $\text{left}(z) = \text{nil}$ is inserted.

As usual, to delete node z from a binary search tree, we must locate the node using the search procedure. Deletion from a binary search tree considers three cases.

Case 1: If z has no child we modify its parent $p[z]$ by replacing z with *nil* as its child. For example, the deletion of key “*stand*” from Figure 2.1.

Case 2: If z has only a single child, we make a new link from $p[z]$ to z ’s only child, $\text{child}[z]$, splicing out z completely. An example is the deletion of

key “no” from Figure 2.1.

Case 3: If z has two children, we splice out z 's successor, say, y which has no left child and replaces its content with z 's content. For example the deletion of the root node from Figure 2.1 will have key “part” becoming the root's key.

The dynamic operations on the binary search tree run in time $O(h)$ since each operation travels downwards from one node to another and in the worse case from the root to a leaf node. Note that h , the height of the tree with n nodes, is $O(\log_2 n)$. If a binary search tree of n nodes is *complete* (where all nodes, except those of the lowest level have two children or all the paths from the root to leaf have exactly the same length), then no path in the tree has more than $(1 + \log_2 n)$ nodes.

Insertions into a binary search tree may not necessarily produce a balanced binary search tree. For example, inserting sorted keys will produce a linked list. In such a case, it takes $O(i)$ steps to insert the i^{th} element and

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = O(n^2)$$

steps to insert n elements. This amortizes to $O(n)$ steps per operation.

It is however proven that if records are presented in random order (where each of the $n!$ permutations of the n input keys are equally likely), insertions takes an average time $O(\log_2 n)$ [2].

2.1.2 Balanced binary trees

The dynamic set operations on the binary search tree run in time $O(h)$, where h is the height of the tree. If records inserted into a binary tree are in sorted order, then as noted above, the tree degenerates and the binary tree reduces to a sequential ordering. This is the worse case insertion and the time for the dynamic set operations on the tree will be $O(n)$, for a tree with n keys. To ensure that the running time of the binary search tree remains logarithmic (i.e. in time $O(\log_2 n)$), the tree must be balanced. Formally, the *height* (or depth) of the binary search tree is the deepest

level of its leaves. A *balanced binary tree* is a binary search tree in which the heights of the two subtrees of every node never differ by more than one. The *balance* or *balance factor* of a node in a binary search tree is defined as the height of the left subtree minus the height of the right subtree. For example, the balance factor of the subtree rooted at the node with key “*this*” in Figure 2.1 is +1 [12; 16; 20].

There are alternative ways of stating the *balance* condition for a binary search tree as there are several ways of balancing it. In the red-black tree in Subsection 2.1.2.2, the tree is said to be β -*balanced* if there are exactly β *black* nodes from the root (but excluding the root) to each leaf node, that is, β is constant for each path. A binary search tree can also be considered balanced in terms of the number of nodes within the left and right subtrees of each node. Assuming each node p holds a field $w(p)$ (the weight of p) which is the number of keys stored in the subtree rooted at p , then node p is α -*balanced* ($1/2 \leq \alpha < 1$) if

$$w(\text{left}(p)) \leq \alpha \cdot w(p) \quad \text{and} \quad w(\text{right}(p)) \leq \alpha \cdot w(p)$$

The entire tree is α -*balanced* if every node in the tree is α -*balanced* [15].

2.1.2.1 AVL-tree

AVL-tree (sometimes synonymous to height balanced binary tree) were first introduced by Adel'son-Vel'ski and Landis [1]. The idea is to keep the binary search tree balanced by performing rotations on nodes that go out of balance. A node of the AVL-tree in addition to the information contained in the ordinary binary search tree node, has a field for the balance factor.

Figures 2.2 and 2.3 demonstrate how rotations are performed. Nodes A, B and C in part (a) of the figures carry their balance factors (before the new insertion). With the insertion of the new node marked “newly inserted node”, both Figures 2.2a and 2.3a become unbalanced at node A, which takes on a balance factor of +2. A right rotation on node A is performed in the case of Figure 2.2a to yield Figure 2.2b, where the new balance factors are shown. In Figure 2.3a however, we perform two

Figure 2.2 Demonstration of right rotation in a binary tree. **Figure 2.3** Demonstration of double rotation in a binary tree.

rotations to rebalance the tree — first a left rotation on node B, then a right rotation on node A. The final resulting balanced tree is shown in Figure 2.3b.

The idea is therefore to proceed with rotations after every insertion (which renders the tree unbalanced) to rebalance the tree. The AVL-tree is essentially a tree that is close to a complete balanced tree in which all paths from the root to any leaf are exactly the same length.

2.1.2.2 Red-black tree

The balancing of AVL-trees may require as many as $\Theta(\log_2 n)$ (see the definition below) rotations after an insertion to maintain balance in an n -node tree. Another binary search tree balancing technique was introduced by Bayer [4], as the symmetric binary B-trees and later named *red-black* trees in [23]. The red-black tree performs rotations after an insertion, in a constant time, $O(1)$.

Definition: The running time of an algorithm of a particular input is the number of operations executed. The running time will be expressed in this thesis using the Θ and O notations. The Θ -notation (asymptotic tight bound) asymptotically bounds a function from above and below, while the O -notation asymptotically bounds a function only from above.

The red-black tree colors each node in the binary search tree either red or black, and satisfies the following properties.

1. Every node is either red or black.
2. Every extension or *nil* node is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Definition: The *black-height* of a node p , $bh(p)$, is the number of black nodes along the path from (but not including) node p to a leaf. The black-height of a red-black tree is the black height of its root.

The red-black tree perform rotations as in the AVL-tree's rotation described above. In place of the field for balance factor in the AVL-tree, the red-black tree has a bit field in a node for the color which is either red (0) or black (1). When a new node is inserted into the red-black tree, it is colored red. The algorithm then proceeds to rebalance the tree by recoloring and performing rotations to ensure that the tree after the insertion conforms to the red-black properties enumerated above. Unlike the AVL-tree, the difference between the heights of the left and right subtrees of a node may differ by more than one. The red-black tree is balanced if property 4 above is satisfied.

The red-black tree and the AVL-tree which are approximately balanced ensure that all the dynamic set operations on the tree run in time $O(\log_2 n)$. In both balancing techniques, insertion is performed by the basic binary search tree's insertion algorithm (see Section 2.1.1) and then the respective balancing technique is applied. The deletion algorithm in these balancing techniques also modifies the deletion algorithm of the binary search tree to maintain their corresponding properties (see Section 2.1.1). The search procedures of the AVL-tree and the red-black tree are however the same as that of the basic binary search tree.

2.2 B-tree and its variants

Binary search trees have the problem of being memory resident, and files of large size may not entirely fit in primary memory. On the other hand, a binary search tree will require many seeks to access a record if it is organized on secondary storage. In a B-tree, once the disk area is located, an entire page can be read from the file. A B-tree allows us to tap all the advantages of the binary search tree and simultaneously provides us with external storage of the index structure. B-tree [5] data structure

and maintenance algorithms as we will see shortly, are generalization of the binary search tree to multi-way trees in which two or more paths (pointers) leave a given node. Basically, B-trees are a collapsed form or the pagination of balanced binary search trees with additional properties.

As shown in Figure 2.4a, if we paginate the binary search tree such that there are three nodes per page, then we have a maximum of 4 (number of nodes or number of keys in the page plus one) pointers leaving a page, each entering a different page in the next level. We can translate a page in the binary search tree into the page structure shown in Figure 2.4b. In Figure 2.4a, any key less than key 10, will be located in page 1, a key greater than 10 but less than 15 will be located in page 2 etc. Similarly, in Figure 2.4b a search for a key less than 10 will proceed to page 1, and a search for a key greater than 10 but less than 15 will proceed to page 2, etc. The maximum number of pointers leaving a page in Figure 2.4b is 4 as in Figure 2.4a. The structure in Figure 2.4b with some additional properties is what constitutes the B-tree.

The properties of paged binary trees were extensively studied long after the introduction of B-tree. In their study in [11], Cesarini and Soda found that an AVL paged binary tree (i.e. a paged binary tree that has each page AVL balanced) and the B-tree are *in fact* similar. Their findings were supported with experimental results in [11].

2.2.1 Basic B-tree

The maximum number of pointers that can leave a page (or the maximum sons that a page can have) is termed the *order* of the tree. The B-tree is built bottom-up, and to enable a page to be at least 50% full at all times, there is the constraint that every page (except the root page), must have at least $\lceil m/2 \rceil - 1$ keys, i.e. it must have at least $\lceil m/2 \rceil$ pointers leaving a page. Naturally, the sons of any node is at most m , where m is the order of the tree. The sons (or children) of a node is equivalent to the number of pointers leaving the node. A node with m sons has $m - 1$ keys. The

Figure 2.4 Transformation from a binary search tree to a B-tree.

root must have at least one key (or two sons), when the tree is composed of only a page, and when the root has no children it is considered as a leaf.

Formally, a B-tree of order m is defined as a tree which satisfies the following properties [27]:

1. Every page has at most m sons.
2. Every page, except the root, has at least $\lceil m/2 \rceil - 1$ keys.
3. The root page has at least two sons (except when it is a leaf).
4. All terminal/leaf pages appear at the same level. That is, the path length from any page to its terminal pages is the same.
5. A non-terminal page with c children contains $c - 1$ keys, for $\lceil m/2 \rceil \leq c \leq m$.

As already noted in Section 2.1.1, a balanced binary search tree with n keys has in the worst-case a dynamic set operations cost of $O(\log_2 n)$. The base of the logarithm, two, is the *branching factor* of the nodes in the binary tree. The branching factor of a node/page is the number of pointers leaving the node/page. To generalize, the dynamic set operations cost for an m -paged (in Figure 2.4a m is 3, a binary search tree is 1-paged) balanced tree is $O(\log_{m+1} n)$. We will assume that pages are stored on secondary storage media and a page is retrieved entirely each time the page is accessed during a search operation.

A node (henceforth the words "node" and "page" will be used interchangeably in our discussion of B-trees) in a B-tree which contains j keys and $j + 1$ pointers is organized as follows:

$$P_0, K_1, P_1, K_2, P_2, \dots, P_{j-1}, K_j, P_j$$

The P_i 's are pointers to the sons and the K_i 's are the keys in the node, for $0 \leq i \leq j$. $K_s < K_t, \forall s < t$. The order of such a tree is $j + 1$.

During a search for key k , if we arrive at the node above, we search for k among the K_i , $i = 1, 2, \dots, j$. We may use a sequential or binary search within a node depending on whether j is small or large. If $k = K_s$, for some $s \in \{1, 2, \dots, j\}$ we

Figure 2.5 Insertion of key "come" into a full page in the B-tree. a. Before insertion of key *come*

b. After insertion and page split.

terminate successfully. If the search is unsuccessful because $K_s < k < K_{s+1}$, we proceed to fetch and examine, in the same manner, the page that is pointed to by the pointer P_s in the current node. If $k < K_1$, pointer P_0 is used and if $k > K_j$, pointer P_j is used. If the pointer chosen points to null, then the search terminates unsuccessfully.

2.2.1.1 Insertion

The insertion algorithm of the B-tree is straight forward. To insert, we first apply the search algorithm for the key say k , that we wish to insert. Any of the following three cases may occur during the search.

1. The key k may be located in some node in the tree. In this case, the algorithm will report an attempt to insert a duplicate key in the tree, and then terminate.

We may get to the leaf node where k is supposed to be.

2. The leaf node may contain less than $m - 1$ keys (m , as before, is the order of the tree). In this situation, we insert k into the page and terminate the insertion procedure successfully.
3. The leaf node may be full, i.e. having $m - 1$ keys. We handle this case by splitting the node into two halves (with one node containing $\lfloor m/2 \rfloor - 1$ and the other $m/2$ keys) with the middle key recursively inserted into the father of the present node. An example of the case is shown in Figure 2.5, for $m = 5$ and $k = \text{"come"}$.

Figure 2.6 An illustration of a page split in the B-tree. **Figure 2.7** A B-tree constructed from randomly inserted keys.

In general if we have a full node of order m at level l , as shown in Figure 2.6a, to insert an item into the page, we insert it at the appropriate place and then split the node into two and promote the middle key to an upper level, namely level $l - 1$ (Figure 2.6b). If the node at level $l - 1$, in which the promoted key, $K_{[m/2]}$ is due to be inserted into contains $m - 1$ keys, then a further split of that page would be required. This will also require the promotion of another middle key from level $l - 1$ to level $l - 2$ and so on. In the worse case, the promotion propagates to the root node, which may also split if it has $m - 1$ keys. The height of the B-tree is increased by one in this situation.

Figure 2.7 shows an example of a B-tree with $m = 4$, constructed from the following random keys: *do, front, then, use, some, all, way, past, none, come, have, yes, more, open, know, just, gate, low, least*.

2.2.1.2 Deletion

Deletion from a B-tree is more complicated than the insertion procedure. To delete a key, a search for the key is first performed. If the key is not found in the tree this fact is reported and the deletion is terminated. If the key is found (either in an internal node or a leaf node) one of the following cases is applied, as may be appropriate.

Case 1: If the key to be deleted is located in a leaf node with $\lceil m/2 \rceil$ or more keys, the key is simply deleted and the affected page reorganized.

Case 2: If the key to be deleted is located in an internal page, it is swapped with its immediate successor (which always resides in the leaf node). The key (now in the leaf node) is then deleted from the tree. After the deletion of the key from the leaf node, case 1 or any of the following situations of case 3 may result.

Case 3: The key to be deleted may be located in a leaf (as in case 1) or is swapped with a leaf key (as in case 2), and the number of keys in the affected page is at most $\lceil m/2 \rceil - 1$. In this situation a deletion of the key will leave the page with $\lceil m/2 \rceil - 2$ keys, which is not acceptable by the B-tree properties (see property 2 on page 14). To maintain the B-tree's properties, the following is carried out:

- redistribution

If the right brother of the affected page, has at least $\lceil m/2 \rceil$ keys, the keys are redistributed among the two pages together with their separator in their common father page. In redistribution, a new middle key is promoted to the father node, to act as the new separator for the two siblings. Two adjacent nodes at level l are said to be siblings, or brothers if they share a common separator in level $l - 1$. If however, the right page has just $\lceil m/2 \rceil - 1$ keys, or is null (no right brother), then if the left sibling exists and has at least

$\lceil m/2 \rceil$ keys, the same redistribution procedure is applied on the two brother nodes.

If a redistribution is not possible, then a concatenation (described below) is applied:

- concatenation

Concatenation is the opposite of page splitting. The keys of the page with $\lceil m/2 \rceil - 2$ keys are merged with one of its brother keys and the key acting as the separator between the merging pages. Merging frees one of the pages. Like insertion, a deletion that involves a concatenation may propagate upwards towards the root, if it finally affects a root page with one key (or two pointers) the height of the tree is reduced by unity.

A B-tree of order m with N keys, has an *upper bound* of

$$h \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2} \right) \quad (2.1)$$

The dynamic set operations search, insert and delete travel down the tree from the root to a node (which may be a leaf). Therefore the dynamic set operations on the B-tree take time $O(h)$ in the worse case or

$$O \left(\log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2} \right) \right) \quad (2.2)$$

Thus the height of the B-tree dominates the cost of its dynamic set operations.

Several variations of the B-tree have been suggested in the literature, notably in [5; 27]. The implementation strategy of these alternatives modifies the rules of the basic B-tree marginally. In their paper, Bayer and McCreight [5], suggested the alternative of redistribution as opposed to strict splitting during insertion, and concatenation during deletion. The redistribution strategy postpones splitting of overflowed pages (in the case of insertion), and the concatenation of pages that underflow (in the case of deletion). This eliminates the overhead associated with these actions. Experiments carried out by McCreight [41] have shown that a B-tree that implements this strategy performs better than the basic B-tree.

Figure 2.8 An example of a B⁺-tree constructed from random keys.

2.2.2 B⁺-tree

An important variation of the B-tree which has become popular was suggested by Knuth [27] and later named in [13] as B⁺-tree. In the B⁺-tree all the keys are stored in the leaves of the tree, with the other levels organized as the B-tree. The leaf nodes are linked together to form what is called the *sequence set*. The internal nodes which contain copies of the keys act as a roadmap to the keys in the sequence set. The internal nodes in the tree form a set called the *index set*. An example of the B⁺-tree of the B-tree in Figure 2.7 is shown in Figure 2.8.

2.2.2.1 Dynamic operations

The search procedure in the B⁺-tree progresses in the same manner as in the basic B-tree, but a location of the search key in the index set does not guarantee (or mean) the presence of the search key in the B⁺-tree. The search must continue until the leaf nodes — the sequence set. The search is successful only if the search key resides in the sequence set.

The insertion process is also the same as in the B-tree. In the B⁺-tree when a split occurs, the middle key is not promoted to the level above the leaf nodes, rather a copy of it is promoted. This procedure guarantees that all the keys reside in the sequence set, and the index set only has copies of the keys and act merely as a guide to the sequence set. In the index set however, if a page overflows, the split behaves as in the basic B-tree, that is, the middle key is promoted to the upper level, with no copy left in the splitting page.

Deletion in the B⁺-tree also takes place only in the sequence set. If the key to be deleted exists in the sequence set, it is simply removed. The copy of the key in the index set, is however not effected, it is allowed to remain to direct search to the sequence set. If the node from which the key was removed underflows, the necessary action is taken — redistribution or concatenation, just as in the B-tree. In concatenation, no key is brought from the father node to participate, rather the key that acts as the separator for the siblings in the father node, say k_f , is simply removed from the father node. This action may propagate the deletion towards the root, and may result in the decrease of the tree's height by one. During redistribution, key k_f may be replaced with a new key, which actually now separates the siblings. This discussion does not apply to actions in the index set, which strictly is the same as in the basic B-tree.

2.2.2.2 Sequential and range query

In the B⁺-tree sequential processing is quite simple because of the linking of the sequence set. To process sequentially, we first locate the leftmost node in the sequence set and proceed to list the keys in the sequence set. The cost for accessing the next item in the sequence given an item (this is termed the *next operation*) is at most one disk access. The sequential processing will require at most one page in memory at anytime and so the memory requirement is very small as compared with the same operation in the B-tree which may require $\Theta(\log_{\lceil m/2 \rceil}(N + 1)/2)$ accesses to secondary storage to process a next operation.

During a range search, the smallest range value is used to locate the node in the sequence set where processing starts till the largest range value is reached. The B⁺-tree is suitable for applications that require both sequential and random processing.

2.2.2.3 Advantages

The search, insert and deletion operations in the B⁺-tree are of order $O(\log_{\lceil m/2 \rceil}(N + 1)/2)$ as in the B-tree. As stated above, the sequence set can be processed sequentially, providing efficient access to records in-order. Unlike the

B-tree, the next operation requires at most one disk access. If the file is organized such that the nodes of the B-tree stores key/record pairs, then the B⁺-tree's index set which contains only copies of the keys (not key/record pair) will be able to accommodate more separator keys per node. It follows that, the number of descending nodes is greater. As a consequence, a B⁺-tree approach can result in a shallower tree than would the basic B-tree approach.

2.2.3 Prefix B-tree

Two variations of the B⁺-tree (motivated by the existence of long variable length keys) that will further enable more separators to be stored in the index set were introduced in [7]. These are the *Simple Prefix B-tree* and *Prefix B-tree*. It is suggested that since the index set of the B-tree serves only as a roadmap to guide searches to the sequence set where the actual keys reside, the index set need not necessarily contain copies of the keys themselves. It should rather contain enough information to guide a search to the right leaf node. Prefixes of the keys could therefore be used in the index set. Unlike the B-tree and the B⁺-tree, in the prefix B-tree the index set is organized such that separators are stored as variable length strings.

In the B⁺-tree of order four, the split presented in Figure 2.9a will result in the leaf node P_1 , upon the insertion of the key "butcher". In fact any separator s with the property $butcher < s \leq calendar$ in node P_f will direct searches to P_1 and P_2 correctly. Practically, s can be for example *buzz*, *bv*, *c*, *cal*, *calendar*. To be able to pack more separators into P_f , the shortest of these possible separators, which is chosen such that it is also a prefix of one of the keys is used. Thus in the example in Figure 2.9a, the string *c* is chosen as the separator, instead of *calendar*.

2.2.3.1 Simple Prefix B-tree

By replacing the copies of the keys in the index set of the B⁺-tree with variable length prefixes, one gets a *simple prefix B-tree* [7]. The main rationale in storing shorter prefixes in the simple prefix B-tree is to enable more separators per node in

Figure 2.9 (a) An example of how a page is split in the B^+ -tree, (b) A leaf node of a B-tree.

the index set, which interprets to a more shallow tree with reduced search, insertion and deletion times.

Not all splits at the middle of leaf nodes that overflow will provide shorter separators. The *split interval* may therefore be chosen around the middle of the leaf that overflows. Consider the leaf node in Figure 2.9b, a split at the middle point will result in a separator *commons*. Although this separator is shorter than the entire key "*commonsensical*", it does not provide enough saving. However, allowing the split point to be one key away from the middle key will yield a separator "*c*" or "*e*".

This idea of split point can also be applied to the index set in choosing shorter separators to promote to the splitting node's father. Clearly this strategy will decrease the height of the tree and may also disadvantageously decrease storage utilization if the split point is chosen far away from the middle split point.

The dynamic set operations are pretty much the same as in the B^+ -tree. In insertion, we choose a split point around the middle point that provides a shorter key prefix separator if a leaf node splits. Similarly a shorter separator is promoted if an index set node splits. This will provide shorter separators towards the root and yields a much more shallow tree.

2.2.3.2 Prefix B-tree

Prefix B-tree seeks to further place more separators in a node to increase the branching factor of the tree and decrease the height of the tree. For any arbitrary page,

P , in a simple prefix B-tree, there is a lower bound $\lambda(P)$ and upper bound $\mu(P)$ for the separators and keys within it. Such that, for any key, k , or separator, s , that is stored in the subtree rooted at P , the following is true.

$$\lambda(P) \leq k < \mu(P), \quad \lambda(P) \leq s < \mu(P). \quad (2.3)$$

$\lambda(\text{root})$ is defined as a sequence of the smallest letter in the set or space in which k and s are drawn, and $\mu(\text{root})$, a sequence of the largest letter in this set, such that, no other k or s is smaller than $\lambda(\text{root})$ or greater than $\mu(\text{root})$.

Let page P contain the following pointers (p_j) and separators (s_k):

$$p_0, s_1, p_1, \dots, s_i, p_i, s_{i+1}, \dots, s_m, p_m$$

for $j = 0, 1, 2, \dots, m$ and $k = 1, 2, \dots, m$. The λ and μ of a page referred to by the pointer p_i as defined in [7] are:

$$\lambda(P(p_i)) = \lambda(p_i) = \begin{cases} s_i & i = 1, 2, \dots, m \\ \lambda(P) & i = 0, \end{cases}$$

and

$$\mu(P(p_i)) = \mu(p_i) = \begin{cases} s_{i+1} & i = 0, 1, 2, \dots, m-1 \\ \mu(P) & i = m. \end{cases}$$

If $\bar{k}_i$ is the longest common prefix of $\lambda(p_i)$ and $\mu(p_i)$, then all separators (s) and keys (k) at the subtree rooted at $P(p_i)$ have a common prefix

$$\kappa(p_i) = \begin{cases} \bar{k}_i l_i & \text{if } \lambda(p_i) = \bar{k}_i l_i z \text{ and } \mu(p_i) = \bar{k}_i l_{i+1} y \\ & \text{where } z \text{ and } y \text{ are arbitrary strings,} \\ \bar{k}_i & \text{otherwise} \end{cases} \quad (2.4)$$

where l_i is any arbitrary digit following $\bar{k}_i$, and l_{i+1} the digit that follows l_i immediately in the key space.

In constructing a prefix B-tree, one does not store all of the common prefixes. The common prefixes $\kappa(P_i)$ are truncated before the remaining strings are stored

Figure 2.10 An example of a simple prefix B-tree and a prefix B-tree constructed from 21 random keys.

in the index set. Figure 2.10b is an example of a prefix B-tree of the simple prefix B-tree in part (a). These trees are the result of the insertion of the following 21 random keys: *has, come, like, pride, thing, can, after, main, study, where, of, kind, joke, effect, probe, substance, all, same, was, have, support*. The capacity of a bucket in this example is two.

It is however advantageous to retain full keys at the leaf nodes if the sequential link processing of records is required. That is, instead of storing “udy” in bucket B10 for “study”, it is advisable to store the full key.

Consider Figure 2.10b. Assuming that the space from which keys are drawn lies in the range $[a, z]$ (i.e. no key k can be less than a or greater than z), then the κ of page P_2 is computed as follows: The right pointer of page P_0 , say p_{01} , points to page P_2 , and therefore $\lambda(P_2) = l$ since p_{01} is not the first pointer in P_0 and the partial separator, l , is immediately to the left of the pointer. On the other hand, $\mu(P_2) = z$ since pointer p_{01} is the last in P_0 . With $\lambda(P_2) = l$ and $\mu(P_2) = z$, $\bar{k}$ of

$P2$ is *null* since there is no common prefix shared by l and z . Furthermore, z does not follow l immediately in the key sequence, and hence $\kappa(P2) = \phi$. Notice that the λ , μ and κ of $P2$ were calculated while at the page $P0$, and with information at $P0$. Thus when the pointer to be followed is chosen, the κ of the page/bucket the pointer points to is computed before arriving at that page or bucket.

As another example let us compute the κ of bucket $B6$ which is pointed to by the leftmost pointer (say $p5_0$) in page $P5$. Pointer $p5_0$ is the first pointer in $P5$, and therefore $\lambda(B6)$ is that of $P5$ which is also pointed to by $p2_0$, the first pointer in page $P2$. So $\lambda(P5) = \lambda(P2) = l$, i.e. $\lambda(B6) = l$. Now $\mu(B6) = \mu(p5_0) = m$, the first partial separator in page $P5$, or the partial separator immediately to the right of $p5_0$. Now $\lambda(B6) = l$ and $\mu(B6) = m$, therefore $\bar{k} = \phi$ for $B6$. But $\kappa(B6) = l$ since m follows l directly in the key space collating sequence.

Below are other computation of λ , μ and κ for some nodes and buckets in Figure 2.10b.

$\lambda(P0) = a$	$\mu(P0) = z$	$\bar{k} = \phi$	$\kappa(P0) = \phi$
$\lambda(P6) = s$	$\mu(P6) = t$	$\bar{k} = \phi$	$\kappa(P6) = s$
$\lambda(P7) = t$	$\mu(P7) = z$	$\bar{k} = \phi$	$\kappa(P7) = \phi$
$\lambda(B0) = a$	$\mu(B0) = c$	$\bar{k} = \phi$	$\kappa(B0) = \phi$
$\lambda(B1) = c$	$\mu(B1) = co$	$\bar{k} = c$	$\kappa(B1) = c$
$\lambda(B7) = m$	$\mu(B7) = p$	$\bar{k} = \phi$	$\kappa(B7) = \phi$
$\lambda(B9) = s$	$\mu(B9) = st$	$\bar{k} = s$	$\kappa(B9) = s$
$\lambda(B10) = st$	$\mu(B10) = su$	$\bar{k} = s$	$\kappa(B10) = st$

The λ , μ and κ for page P_i can be computed during the traversal of the tree from the root towards P_i . During a search for key k , at node P_i , $\kappa(P_i)$ is determined and pruned from k , yielding $\hat{k}$ which is then used in comparisons with the $\hat{s}$ (the partial separators) in page P_i . For example, to search for the key “same” in Figure 2.10b, one proceeds as follows:

At page $P0$, the root, $k = \hat{k} = same > l$, the separator and therefore $P2$ is to be visited, at $P0$, $\kappa(P2) = \phi$. Therefore $k = \hat{k} = same$ at page $P2$, since

$s < same < t$, page $P6$ will be visited next. At $P2$, $\kappa(P6)$ is calculated to be s , and therefore $k = \hat{k} = ame$. At $P6$ the string $ame < t$, so we visit the leaf node — bucket $B9$, which also has $\kappa = s$. Since ame exists in bucket $B9$, the search is successful. If full keys were stored in the leaf nodes, on arriving at bucket $B9$, the original search key is sought for in the node, and not its pruned version.

During insertion and deletion, if an overflow occurs, it is handled as in the B^+ -tree, where separators are replaced by their partial separators, $\hat{s}$ (separators with pruned common prefixes κ). For further discussions, we refer the reader to [7].

As can be seen in Figure 2.10, separators and partial separators are shorter than in the B^+ -tree and therefore more of them can practically fit within a page than depicted in the figure. The partial separators are a product of both rear and front compression of keys. The prefix B-tree, like digital search trees, constructs prefixes on-the-fly and in addition preserves the balanced properties of the B-tree. A Prefix B-tree therefore enjoys the combined merits of B-tree, key compression and digital search trees.

Assuming that a page $P(p)$ referenced by a pointer p is organized as follows:

$p_0, \hat{s}_1, p_1, \dots, \hat{s}_i, p_i, \hat{s}_{i+1}, \dots, \hat{s}_m, p_m$	unused
--	--------

then the search algorithm of the prefix B-tree is as presented in Algorithm 2.1 below.

An alternative implementation of prefix B-tree is to construct a simple prefix B-tree, but instead of separators, we replace them with partial separators computed such that for each page, the $\bar{k}$ is trimmed from all separators and stored within the page as part of the contents of the page. This implementation is much simpler than the previously discussed implementation.

Search algorithm of prefix B-tree**Begin**Let the tree be rooted at T , with page number t .Let k be the search key. $p = t$;while $P(p)$ is an internal page compute $\kappa(p)$ as defined in Equation 2.4; remove $\kappa(p)$ from k , to get $\hat{k}$; if $\hat{k} < \hat{s}_1$ then $q = p_0$; if $\hat{s}_i \leq \hat{k} < \hat{s}_{i+1}$ then $q = p_i$; if $\hat{s}_m < \hat{k}$ then $q = p_m$; $p = q$;

endwhile;

retrieve bucket referenced by p ;examine bucket for k ;**End**

Algorithm 2.1: The search algorithm of the prefix B-tree.

Chapter 3

Trie hashing

Trie hashing is a data structure that was proposed by Litwin in the early 1980's [31; 32]. There have been several contributions to the structure since its introduction. In the sequel, we will refer to the trie hashing structure, or simply the trie proposed by Litwin as the *basic trie*, to distinguish it from the other proposed modifications to be described later.

3.1 The basic trie hashing scheme

Trie hashing unlike other hashing schemes stores records in-order. It is an access method for storing and accessing records of dynamic files. Searching for a record requires only one disk access, since the trie itself is memory resident. The key/record pairs are stored in buckets on disk. If records are inserted randomly, the load factor of the trie file is around 70%. Trie hashing has a tree-like structure and is ordered. The structure grows by splitting and sharing of keys between buckets.

The trie hashing structure is a binary data structure with a node in the structure defined as in the following figure.

The pair (DV, DN) is referred to as DF (the digit field).

Figure 3.1 An example of a trie hashing structure.

the, and, for, that, you, this, with, not, have, are, from, can, but, will,
all, was, there, one, they, what, would, any, which, about, get, your,
use, some, then, name, like, out, when, time, other, more, only, just,
end, also, know, how, new, should, been, than.

b. Key insertion order.

about	get	use	one	for	been	should	the	you	will	and	this	more	was	time	know
all	have		only	from	but	some	then	your	with	any		name	what		like
also	how		other	end	can	that	there		would	are		new	when		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

c. The buckets after the 46 keys have been inserted

A node in a trie is either internal or external. An internal node has its pointers pointing to other nodes, while an external node has no digit field, but stores the address of a disk resident bucket. A bucket has a capacity, c , where $c > 1$. Figure 3.1 is an example of a trie, in which Figure 3.1b is the data of the first 46 most used English words which have three or more letters (ordered by frequency). The italicized words are the words whose insertion caused collision (or bucket overflow). Figure 3.1c is the buckets contents after the insertions of the data in Figure 3.1b in which the bucket capacity is set at $c = 4$. The trie structure is shown in part (a). Detailed, search, insertion and deletion algorithms of the structure are presented in subsequent sections.

If the trie is empty, then only one bucket may be allocated. Assuming that the trie is not empty, to search for a key, k , within the trie, the procedure is as follows. We maintain two strings initially set empty, say u (called the comparator string in [47]) and v . String u at any node say P , will contain the logical path [35] from the root to P . The logical path to node P , u_p , is defined recursively as follows:

If P is the root, $u_p = :$, where $:$ is the largest digit in the key space.
otherwise,

let f be the father of P ,
if P is the right child of f then $u_p = u_f$
else $u_p = (u_f)_{DN(f)-1}DV(f)$.

$(u_f)_{DN(f)-1}$ is empty if $DN(f) - 1$ is negative. The logical path from root to bucket 7 is as illustrated in Figure 3.1a by the letters on the edges of the graph.

During a search, when node P is the current node arrived at, the comparator $u = u_0u_1 \cdots u_{DN(P)-1}DV(P)$ ($u_0u_1 \cdots u_{DN(P)-1}$ is a prefix of the logical path to P) is compared with a substring of the key k , $k' = k_0k_1 \cdots k_{DN(P)}$. If u , the comparator, is less than k' , in the collating sequence, then the right pointer of P is followed, otherwise the left pointer of P is followed. Whenever the right pointer is followed, the string u computed at P is discarded and u is set back to v . When a left pointer is followed, v 's value is updated by setting it to u 's value.

The search terminates if a pointer points to an external node, or a null value. If the external node is non-null, the bucket's whose address is stored in the node is brought into memory from disk and examined if k , the search key is present. The formal search algorithm is shown in Algorithm 3.1.

A search algorithm for trie hashing**Begin**Let the trie be rooted at T .Let k be the search key. $u = v = \phi$;*Initialize u and v as empty strings.* $P = T$;while P is an internal node and $P \neq \text{null}$ if $DN(P) > 0$, then $u = u_0u_1 \cdots u_{DN(P)-1}DV(P)$;*Concatenate u^1 with $DV(P)$.*else $u = DV(P)$;*Copy $DV(P)$ into u .* $k' = k_0k_1 \cdots k_{DN(P)}$;*Compute k' .*if $u < k'$ then $P = RP(P)$; $u = v$;*Move right, discard u at P .*else $P = LP(P)$; $v = u$;*Move left, retain u .*

endwhile;

if $P = \text{null}$

return message for unsuccessful search;

No disk access in this case.

else return bucket address;

End**Algorithm 3.1:** The search algorithm of the trie hashing.

The structure of the search algorithm guarantees that any successful search takes one disk access, and an unsuccessful search at most one disk access. Suppose we are searching for the key "when" in the example trie in Figure 3.1. We would start at the root of the trie where $DN(f, 0) = 0$, therefore $u = "f"$ and $k' = "w"$. Since $"f" < "w"$, so we take the right pointer and P , the current node is now $(t, 0)$, $u = v = \phi$, empty. At $(t, 0)$, $DN = 0$ and so $u = "t"$, $k' = "w"$, and again the right pointer is followed to $P = (w, 0)$. Again u is set to the value of v which is still empty. At $(w, 0)$, $DN = 0$, implying $u = "w"$ and $k' = "w"$ giving $u = k'$, and so the left pointer is taken and v is set to the value of u , "w". At $(h, 1)$, $DN > 0$, and so $u = "wh"$, $k' = "wh"$ and once again $u = k' = "wh"$. Thus the left pointer is followed to node $(u, 0)$ where $DN = 0$, and the variable u equals constant "u", $k' = "w"$, and since $u < k'$, the right pointer is followed. The node now visited will be detected to be an external node, holding the address of bucket 13. Bucket 13 is

¹ u truncated at the $DN(P)^{\text{th}}$ digit.

then retrieved from disk and examined in memory for key “when”, which is found, hence the search is successful.

3.1.1 Insertion

To insert a key, k into the trie structure, a search must first be performed. Let the capacity of a bucket be c . On arriving at an external node P , the bucket whose address is in P , say B_s , is read into memory. Assume that the search fails, if there are less than c keys in B_s , k is inserted into B_s at its lexicographic position and the insertion terminates. However, if B_s contains exactly c keys, i.e. the bucket is full, a collision is said to have occurred. Bucket B_s will be divided into two.

A split key will be chosen among the $c+1$ keys. The split key is a prefix of one of the $c+1$ keys, which is usually referred to as the middle key. The procedure involved in choosing the split key is described in the next section. Assuming $k' = k'_0 k'_2 \cdots k'_{i-1}$, is chosen as the split key, k' is used as a search argument in the search algorithm. On arrival at node P during the second search, if u , the comparator string matches the first I digits of k' , then $i - I + 1$ nodes are created such that, $LP(P_j) = P_{j+1}$, $RP(P_j) = \text{null}$, $DV(P_j) = k'_j$ and $DN(P_j) = j$, for $j = I, I+1, \dots, i-1$. Node P is replaced by node P_I , and node P_i will have two external nodes as its sons. Let these external nodes be P_l and P_r , then $LP(P_i) = P_l$, and $RP(P_i) = P_r$. Also, the bucket addresses B_s and B_n are stored in P_l and P_r respectively, where B_n is a newly allocated bucket. All keys (of the $c+1$ keys) that are less than or equals to k' are stored in bucket B_s and all other keys (greater than k') are stored in B_n .

Figure 3.2 illustrates the splitting procedure described above. Suppose during the search to insert a key we follow a pointer that leads to a *null* node, we allocate a new bucket and create a new node to hold the bucket's address. The key to be inserted is then inserted into the new bucket.

Figure 3.3 shows an example of the insertion procedure. Assuming the bucket capacity $c = 3$, then bucket $b2$ in part (a) is full. If we attempt to insert the key “there”, on arriving at node P , we examine bucket $b2$ which is full, so we compute $k' = \text{“ther”}$. Performing a search with “ther”, on arriving at $P = b2$, $u = \text{“th”}$,

Figure 3.2 An illustration of how an insertion is accomplished in the trie hashing.

therefore we create two internal nodes. Assuming a new bucket b_5 is allocated, then the resulting trie is shown part (b) of Figure 3.3.

If the next key to be inserted into the trie (in Figure 3.3b) is “third”, the search fails on a pointer to a *null* node. In this situation, a new bucket b_6 is allocated to hold key “third” and an external node is created in place of the *null* node to hold the bucket b_6 ’s address. The resulting buckets of the file are shown in part (c) of the figure.

3.1.2 The concept of split key

Storage utilization has always been a concern of file structures. We will normally want to split a bucket into two equal sizes when an overflow occurs to guarantee a storage utilization of 50% (as in the case of B-trees). However, the trie structure may not behave well when the split key is chosen as the prefix of the middle key of the $c + 1$ keys.

An example will best illustrate this notion. In Figure 3.4 the capacity, c of a bucket is 5. On inserting the 6th key “there” into the bucket in part (a), an overflow occurs. If we choose *thes* (prefix of the middle key “these”) as the split key (assuming our second search with “thes” yielded $I = 0$), then the resulting trie is as illustrated in Figure 3.4b. Now assuming during the search for the key “those”, we arrive at the node $P_I = (e, 2)$, the comparator $u = “the” < “tho” = k'$, therefore the right pointer is followed, and $RP(P_I) = null$. The search is unsuccessful, and

Figure 3.3 An example of trie hashing insertion procedure.

the choice of split key leads to a wrongly constructed trie. Otoo in [47] addressed this problem and defined the split key as follows:

In a set of c ordered keys K_i , $0 \leq i \leq c - 1$, such that each is terminated by a blank symbol " $\sqcup$ ", the middle key $K_j = k'_0 k'_1 \dots k'_{l_j-1}$, $j < c - 1$, is one such that if the next key in the ordered sequence is $K_{j+1} = k_0 k_1 \dots k_{l_{j+1}-1}$, then for some $l' \leq \min(l_j, l_{j+1})$ we have:

1. $k'_s = k_s$ for $s < l'$, and $k_{l'} \neq k'_{l'}$.
2. l' is minimum over all j satisfying (1) above.
3. $|\lfloor c/2 \rfloor - j|$ is minimum for all j satisfying (1) and (2) above.

In our example, following the definition, the split key will be the prefix of the key "those" which is "tho". The correct splitting of bucket b_2 in Figure 3.4a is shown in Figure 3.4c.

Figure 3.4 An example of how the split key is selected in the trie hashing during a split.

3.1.3 Deletion

Deletion from the trie hashing structure is simple. To delete a key, the search algorithm is first applied, if the key to be deleted is found, it is simply removed from the bucket. We do not concern ourselves with the fact that the deleted key could have been used to form the nodes leading to it.

If after deletion, the bucket becomes empty, we deallocate the bucket and set the external node, say p , of the trie holding the bucket's address to *nil* and free memory area corresponding to it. If the sibling of p is also an external node, say q , we link the grandfather of p (which is an internal node) to q and free p 's father $f(p)$. If q is *null*, we set $f(p)$ to *nil*. In either case, the node deletion is propagated upwards. Bucket merging is also possible if both p and q are external nodes and have a total of less than c keys. We merge p and q 's buckets to the bucket referenced by the left external node, let it be p . The external node p is then made the child of its grandfather, and $f(p)$ is freed.

3.1.4 Range query

To perform a range query, the leaf nodes have to be linked as in the B-tree. To link leaf nodes, a modification is required to the trie structure *nil* values. One may replace them with empty external nodes with no bucket pointers, but like all other external nodes, with pointers to the nodes immediate before and after it. Alternatively, without any modification to the trie structure, range query can be performed by traversing the trie. Since the keys are in the buckets whose addresses are stored in the external nodes of the trie, in-order, pre-order and post-order traversal will yield the same result. To process the trie in Figure 3.1 in-order, we have to visit the buckets in the order 0, 10, 5, 4, 1, 15, 12, 3, 6, 7, 11, 14, 2, 13, 9, and 8.

The in-order trie traversal algorithm is presented in [35]. Litwin et al. [35], also suggested two methods to process a range query. One way is to use the in-order traversal algorithm and only list keys within the specified bounds. If the lower bounds is *key1* and the upper bound is *key2*, then using the in-order algorithm, we

process all records with keys lying in the range $[key1, key2]$. In this case, all buckets up to the bucket containing $key2$ will be read. To limit the number of buckets examined, a prior search for $key1$ and $key2$ can be performed so as to know which buckets will be examined during the traversal. The search for $key2$ is however not necessary since we can terminate the algorithm once a key greater than $key2$ has been accessed.

A one-pass range processing will require the left edges to be stacked during the search for $key1$. The stack maintained during this strategy will contain the same nodes as the in-order traversal algorithm's stack when it gets to the bucket with $key1$. In this second strategy, fewer nodes are examined in addition to the fewer buckets that are actually read during the query processing. For example, in performing the range query for $key1 = "fa"$ and $key2 = "oz"$, using the trie in Figure 3.1, one proceeds as follows. First a search for $"fa"$ is performed, which terminates in the external node with bucket address 4. The stack starts with $S = (f, 0)$ and we do not visit the subtrie rooted at $(a, 0)$ at all. Using the in-order traversal algorithm, bucket 4 is retrieved from disk and all key values greater or equal to $key1$ are processed. This includes keys $"for"$ and $"from"$ (note that key $"cnd"$ is ignored). By the organization of trie hashing structures, it is guaranteed that after bucket 4, all subsequent buckets have key values greater than $"from"$ and for that matter greater than $key1$. Following the in-order procedure, $(f, 0)$ is popped, and $(t, 0)$ is visited. We stack $(t, 0); (n, 0); (l, 0); (j, 0)$, visit and process keys in bucket 1 less than or equal to $key2$. Since all keys in bucket 1 are less than $key2$ they are all processed. We thus proceed till bucket 3, process all keys, pop $(o, 0)$ from $S = (t, 0); (o, 0)$ which again grows to $(t, 0); (h, 1); (a, 2)$. It is only when bucket 6 is examined that it will be realized that processing of the range query is done, since all keys in bucket 6 are greater than $key2$. The algorithm now terminates producing all keys in the file between $key1$ and $key2$ inclusive.

We present a range query algorithm, Algorithm 3.2, that further enhances the performance of the second strategy. Compared to the second strategy described above, Algorithm 3.2 stacks fewer nodes, by using the upper bound, $key2$, as well

as the lower bound, $key1$.

Range query processing algorithm

Input: Bounds of the query, $key1$ and $key2$.

Output: Keys within $key1$ and $key2$ inclusive.

Begin

```

    Let the trie be rooted at  $T$ ,
     $key1 = k_1k_2 \cdots k_i$ , and
     $key2 = l_1l_2 \cdots l_j$ .
     $u = v = \phi$ ;                                     Initialize  $u$  and  $v$  as empty strings.
     $P = T$ ;
    for (;;)                                           Do forever.
        while  $P$  is an internal node and  $P \neq null$ 
            if  $DN(P) > 0$ , then
                 $u = u_0u_1 \cdots u_{DN(P)-1}DV(P)$ ;      Concatenate  $u$  with  $DV(P)$ .
            else  $u = DV(P)$ ;                             Copy  $DV(P)$  into  $u$ .
             $k' = k_0k_1 \cdots k_{DN(P)}$ ;                 Compute  $k'$  and  $l'$ .
             $l' = l_0l_1 \cdots l_{DN(P)}$ ;
            if  $(u < k')$  then  $P = RP(P)$ ;  $u = v$ ;
            else if  $(u < l')$  then stack( $P$ );  $P = LP(P)$ ;  $v = u$ ;      Stack node
            else  $P = LP(P)$ ;  $v = u$ ;                     else don't stack node.
        endwhile;
         $Bucket = bktAddr(P)$ ;                             Assign bucket address to  $Bucket$ .
        read and process  $Bucket$  in  $[key1, key2]$ ;

    do
        if  $((q = pop(q)) = null)$  return;                Pop node from top of stack.
         $P = RP(q)$ ;                                         Follow right pointer.
        if  $(P = null)$                                      Ignore null node and
             $flag = 0$ ;                                       pop another.
        else  $flag = 1$ ;
        while  $(flag = 0)$ ;                                End of do...while loop.
    endfor;
End
```

Algorithm 3.2: The range query processing algorithm for trie hashing

Consider the example for the range query with $key1 = "fa"$ and $key2 = "oz"$. Using the refined algorithm, as in the second strategy, the subtrie rooted at $(a, 0)$ is not visited at all. In addition, $(t, 0)$ is never stacked since at $(t, 0)$, $u = "t"$ and $t >$

“oz”. This guarantees that the entire subtrie rooted at $(w, 0)$ will never be visited for the processing of this query. Node $(o, 0)$ is the last node visited, and since it will not be stacked, the stack will be empty, and the algorithm will terminate after bucket 3 is processed.

Section 3.3.1 presents a solution to *null* nodes in the trie hashing structure.

3.2 Reconstruction and optimization of trie hashing

In [51] the basic trie hashing scheme was extended to enable the mapping of the trie structure into secondary storage during the creation of the trie. This scheme makes it possible for the trie to be reconstructed in the event of loss of information due to system crash or the termination of the searching/insertion algorithm. A further algorithm in [51] provides for the optimization of the reconstructed trie which is usually a chain. The optimized trie is usually more balanced than the basic trie.

The information that is saved for the reconstruction algorithm utilizes two storage areas, namely, NS and BS (probably NS stands for “nodes set” and BS “buckets set”). NS stores a collection of strings and BS a sequence of integers. Initial values of NS and BS are set to empty, and as the insertion progresses NS and BS are updated accordingly. NS will contain all the comparators that result from the insertion algorithm from the root of the trie to each leaf node from the leftmost to the rightmost. The set BS will contain the collection of bucket address from the leftmost external node bucket address to the rightmost. All elements in NS that are prefixes of the strings that precede them in NS are removed from NS. For example, if the set NS contains the subset $\{wh, w\}$, it is reduced to $\{wh\}$.

For example after the complete insertion to the trie in Figure 3.1a the sets NS and BS are respectively,

$$NS = \{al, c, f, j, l, n, o, tha, the, u, wh\},$$

$$BS = \{0, 10, 5, 4, 1, 15, 12, 3, 6, 7, 11, 14, 2, 13, 9, 8\}.$$

The initial value of set NS is $\{al, a, c, f, j, l, n, o, tha, the, th, t, u, wh, w\}$.

Figure 3.5 An illustration of how the reconstruction algorithm produces an equivalent trie.

The reconstruction algorithm uses the sets NS and BS (also known as the formal trie in [51]) to construct an equivalent Litwin (basic) trie. The reconstruction algorithm recursively creates an internal node for each digit in the elements of the set NS and link the elements of BS to the external nodes created. In general, if $NS = \{k_0k_1k_2 \dots k_p, l_0l_1l_2 \dots l_q, \dots, m_0m_1m_2 \dots m_r\}$ and $BS = \{b_0, b_1, b_2, \dots b_s\}$ then the reconstruction algorithm will produce the equivalent Litwin trie in Figure 3.5. It is assumed that no consecutive elements in NS have common prefixes. An example of when this does occur is shown in the example in Figure 3.6 where strings "tha" and "the" are in NS. The figure is an example of the reconstructed trie of the sets NS and BS of the trie in Figure 3.1a presented above.

The sets NS and BS are stored on disk and can be reloaded by at most one disk access, on the premise that the entire trie is memory resident. If NS and BS become very large, and cannot stay in memory together with the trie one or both of NS and BS can be stored on disk.

Obviously, the trie reconstructed from the formal trie is less attractive since it is mostly skewed to the right. Torenvliet and van Emde Boas [51] have presented an additional heuristic algorithm that seeks to optimize the transformed trie. Though not formally proved, experiment results presented in [51] suggest that the optimized

Figure 3.6 An example of an equivalent trie produced by the reconstruction algorithm.

trie is more “balanced” than the basic trie.

The optimization algorithm selects from the transformed trie a new root in such a way that for the selected node r , the value $|w_l(r) - w_r(r)|$ is as small as possible, where $w(P)$ is the weight of a trie rooted at P , and $w_l(r)$ and $w_r(r)$ are the weights of the left and right subtrees of node r respectively. The weight of a trie is the number of nodes in the trie.

After selecting the new root, a local transformation is performed to distribute the nodes into the left and right subtrees of the new node. An illustration of the local transformation is given in Figure 3.7. Using the left and right subtrees, we recursively call the optimization algorithm on them. That is, a new root for the subtree is selected using the selection condition above, the subtree is then locally transformed.

Applying the optimization algorithm to Figure 3.6, we get the optimized trie shown in Figure 3.8.

Figure 3.7 An illustration of how local transformation is performed during the application of the optimization algorithm a trie.

a Before local transformation.

b After node with "k" is chosen for as root for local transformation.

Figure 3.8 An optimized trie of the trie in Figure 3.6.

3.3 Locally balanced compact trie

Another approach to resolving the basic trie hashing problem has been proposed by Otoo in [46; 47]. In [46] a balancing scheme similar to the AVL-tree's was introduced to balance the trie. A strategy to enable the trie to be reconstructed in the event of system failure was also presented.

In this section, we will consider a “less-than-or-equal-to” ($\leq$) comparison between the comparator string, u , and the prefix of the search key, k' , in Algorithm 3.1. In other words, the right pointer of the current node will be followed if $k' = u$. This will also mean discarding the comparator u computed at the current node, P , if the left pointer is followed, and retaining it when the right pointer is taken.

3.3.1 Compact binary trie

In [46], Otoo introduced a scheme called the *compact binary trie* that improved upon the performance of the basic trie hashing scheme in that it eliminates the null pointers that appear in the basic trie structures. Compact binary tries also enable the median key's prefix to be selected as a separator in the event of a split. This automatically guarantees a load factor of at least 50%. Furthermore the problem of linking buckets of the trie file is eliminated and so buckets can be linked by pointers for easier sequential processing.

In the compact binary trie, the entire separator, s , obtained after the comparator is trimmed from it, is stored as the digit value in the node p with the digit number of p being the size of the comparator. The digit number of p specifies the position at which the digit value of p is to be concatenated to the comparator computed up to p . Figure 3.9 shows the difference between the basic trie and the compact binary trie; in the figure bucket $b0$ splits and yields a separator “ th ”.

The implementation of compact binary trie will not require variable length digit value fields for a memory resident trie structure. A fixed length pointer will be stored at the digit value's field which points to the memory address of the digit value string. An example of compact trie that appear in [47] is presented in Figure 3.10, it

Figure 3.9 Difference between the basic trie and the compact binary trie during a split.

Figure 3.10 An example of a compact binary trie.

is constructed from the KWIC index (see page 54). The resulting bucket structure of the trie is shown in Figure 3.16.

In the compact trie, if the comparator u matches the first I digits of the split key $k' = k'_0 k'_1 \dots k'_{i-1}$, at node P where the split key is to be inserted (see page 32), then the $DV(P) = k'_I k'_{I+1} k'_{I+2} \dots k'_{i-1}$ and $DN(P) = I$.

3.3.2 Balanced compact trie hashing

Balanced compact trie hashing (BCTH) is a compact trie that is locally balanced with respect to nodes in the trie with common digit numbers which form (or is connected as) a subtree. The balancing strategy is the same as in the AVL-trees. Because of the trie rotation condition outlined in page 49, BCTH introduces migration into its scheme, so that rotations are carried on only nodes with common digit numbers.

Figure 3.11 Illustration of RL and LR migrations of the BCTH.

When a new node p is inserted, the BCTH algorithm ‘pushes’ the node up the trie (the node is said to migrate) towards the root until p becomes the son of a node q where, $DN(q) \leq DN(p)$. If the migration causes nodes in the trie with common digit numbers forming a subtree to be unbalanced, the BCTH balancing scheme is employed to balance the corresponding subtree using left and right rotations as in the AVL-trees (see Subsection 2.1.2.1).

The entire trie is said to be locally height balanced if for each subtree with common digit number, for every node p in the subtree, $|h_l(p) - h_r(p)| \leq 1$, where $h(P)$ is the height of a trie rooted at P , $h_l(p)$ and $h_r(p)$ are the heights of the left and right subtrees of node p respectively.

Figure 3.11 illustrates how a migration in BCTH is performed. In both parts (a) and (b), the node with value R migrates by one step. Migration guarantees that along any path from the root to a leaf node, the digit numbers occur in a non-decreasing order. An example of a BCTH of the compact trie in Figure 3.10 is shown in Figure 3.12.

If the left (right) spine of the trie has non-decreasing digit numbers, then LR-

Figure 3.12 An example of a balance compact binary trie.

migration (RL-migration) will not be required. The left (right) spine of a binary trie rooted at R is the maximal path traversed, starting from R , by following the left (right) pointers only [47]. In the basic Litwin trie where the comparison between the comparator and the insertion key's prefix uses the "less-than" sign ($<$), (see Algorithm 3.1), the right spine has non-decreasing digit numbers. If the ' $<$ ' sign is changed to ' $\leq$ ' the left spine will have non-decreasing digit numbers instead.

Migration and rotation of nodes maintains the sequence order and leaves the trie hashing function invariant. Thus the basic trie search algorithm still apply to the BC'TH structure.

3.4 Multilevel trie hashing

Multilevel trie hashing (MLTH) [35] is the pagination of the trie hashing structure as in the paged binary tree (see Section 2.2). In MLTH, the splitting of bucket is still the basis for the trie construction, with the exception that tries themselves split into subtries. These subtries are then stored as pages to overcome the memory and reliability problems faced by the basic tries.

In MLTH a *page capacity* (i.e. the number of nodes per page) is set, and if a page overflows, a node say r , within the page is chosen as the split node. The trie

is then split in a way similar to a page split in the B-tree. Node r is promoted to the father page while nodes that precede node r in in-order are left in the splitting page, T_L , and all other nodes are moved to a new (right) page, T_R . The pointers in r are set such that, $LP(r) = T_L$ and $RP(r) = T_R$. The construction of MLTH therefore involves two phases, namely:

- choosing split node r , and
- splitting the (sub)trie using r .

3.4.1 Choosing a split node

The choice of r must be such that it balances the trie after the split of the second phase. To achieve an even split, a node r must be found such that:

1. the number of internal nodes that precede r in in-order in the subtrie to split say T , is the closest to the number of nodes following r ;
2. r has no logical parent within T . This implies that the comparator computed within T up to node r (i.e. the logical path of r) is *null* for $DN(r) > 0$. A node with $DN = 0$ has no logical parent. The logical parent of node $(h, 1)$, (the son of node $(o, 0)$) in Figure 3.1 is $(t, 0)$.

A simple algorithm in [35] that utilizes the in-order traversal algorithm enables r to be selected easily.

3.4.2 MLTH splitting

During the computation of r with the algorithm of [35], a set E which is a copy of the stack S in the in-order algorithm is maintained. It is the set E that is used in a subsequent algorithm in [35], to build the MLTH.

Basically, all nodes that precede node r in in-order are left in the splitting page T_L , and all other nodes are moved into a new page T_R . Node r is moved to the parent page T_p if one exists, and if such a parent does not a new page T_p is allocated

for node r . Pages T_L and T_R are then written to disk. Node r in T_p is now modified to point to pages T_L and T_R , satisfying, $LP(r) = T_L$ and $RP(r) = T_R$.

Page T_p is now tested for overflow. If no overflow takes place this page is also written to disk. Otherwise the MLTH algorithm is invoked recursively on T_p .

Two sibling pages T_1 and T_2 (with parent T_p) may be merged as a result of a deletion that results in a fewer number of nodes in one of the pages. Page merging is possible if the total number of nodes in the two pages is less than the page capacity. In this case, pages T_1 and T_2 are merged onto T_1 together with their split node in T_p . Deletion in MLTH is therefore relatively easy. For a detailed treatment of MLTH, the reader is referred to [35].

3.5 rbTrie

We present yet another means of balancing the trie hashing structure in order to solve the possible degeneracy problem of the original Litwin trie.

Red-black tries or rbTries are the result of a balancing strategy that is similar to that of the red-black trees on the trie hashing structure. A red-black tree is one of the various ways by which a binary search tree can be balanced, and these trees were briefly introduced in Subsection 2.1.2.2. In rbTrie we will require, the following modifications to the basic trie.

1. The entire (partial) separator after the prefix is trimmed will be stored as the digit value for the concerned node. This modification yields a compact trie, (see Section 3.3.1), and prevents deep trees and null nodes. Alternatively, *null* leaf node may not be allowed, and consequently unused nodes will be created in their place. Typically, the first alternative is chosen due to its superiority.
2. In addition to the content of a node in the basic trie, the rbTrie will have two additional fields for the node's color and a backward pointer to its father node.
3. The red-black condition of black height will be modified (see Subsection 2.1.2.2, condition 4) as follows:

- Every simple path from a node to a descendant leaf contains **approximately** the same number of black nodes.

An estimate of the deviation of the black height of the rbTrie will be done in Chapter 5 using simulation.

Apart from condition 4 of the red-black trees which is modified above, all the other conditions apply to rbTrie.

Rotation in the trie structure may not involve nodes with different digit numbers. In particular, a rotation should not change the role of a node x and its parent $p(x)$ if $DN(x) > DN(p(x))$. If this *trie rotation condition* is not adhered to, the trie structure will be modified and the search algorithm will fail to locate some keys.

Thus the modification of red-black condition of black height results from the above trie rotation condition, i.e., rotations are not allowed to take place if it will lead to decreasing digit numbers for the nodes participating in the rotation. This condition is strictly checked in the insertion algorithm described presently.

3.5.1 Insertion

When constructed, the rbTrie uses the same search algorithm as the basic trie. Similarly, to insert a record into the rbTrie, we first apply the search algorithm of the basic trie to locate the bucket in which the key is to be inserted. Algorithm 3.3 is the insertion algorithm for the rbTrie which uses the same insertion algorithm as the basic trie. The algorithm then proceeds to color the inserted internal node red (the external nodes, which contains the bucket addresses are always colored black). After the coloring, Algorithm 3.3 proceeds to examine which of the conditions of the rbTrie are violated. If any of the conditions are violated, the algorithm fixes them.

Conditions 1 and 2 certainly continue to hold since every node is colored and the external nodes are all black. Modified condition 4 also holds since the newly inserted node takes the place of an external node which was black. Since we color the newly inserted internal node red, and its external nodes black, the number of black nodes on the path is unaffected.

The rbTrie Insertion Algorithm

Assumption: The algorithm invokes the `Trie-Insert()`, the insertion algorithm of the basic trie.

Begin

```

    Let the trie be rooted  $T$ ,
    and let  $k$  be the key to be inserted.
    Trie-Insert( $T, k$ );
    if bucket split does not occur, return;
    color inserted internal node  $x$  red;
    while  $x \neq \text{root}[T]$  and  $\text{color}[f(x)] = \text{red}$ 
        let  $gf(x) = f(f(x))$ ;
        if  $\text{DN}[x] > \text{DN}[f(x)]$  or
            $\text{DN}[f(x)] > \text{DN}[gf(x)]$ 
            color $[f(x)] = \text{black}$ ;
            if  $f(x) = \text{left}(gf(x))$ 
                color $[\text{right}(gf(x))] = \text{black}$ ;
            else color $[\text{left}(gf(x))] = \text{black}$ ;
            color $[gf(x)] = \text{red}$ ;
             $x = gf(x)$ ;
        else
            if  $f(x) = \text{left}(gf(x))$ 
                 $y = \text{right}(gf(x))$ ;
                if color $[y] = \text{red}$ 
                    color $[f(x)] = \text{black}$ ;
                    color $[y] = \text{black}$ ;
                    color $[gf(x)] = \text{red}$ ;
                     $x = gf(x)$ ;
                else
                    if  $x = \text{right}(f(x))$ 
                         $x = f(x)$ ;
                        Left-Rotate( $T, x$ );
                    color $[f(x)] = \text{black}$ ;
                    color $[f(f(x))] = \text{red}$ ;
                    Right-Rotate( $T, f(f(x))$ );
            else (same as the then clause with
                  "right" and "left" exchanged)
    endwhile;
    color $[\text{root}(T)] = \text{black}$ ;

```

End

Case 1.

Case 2.

Case 3.

Case 4.

Cases 5, 6, 7.

Algorithm 3.3: The insertion algorithm of the rbTrie.

The only condition that is therefore likely to be violated is condition 3, which says that a red node cannot have a red child (or a red father). Specifically, condition 3 is violated if the inserted internal node has a red parent. The *while* loop in Algorithm 3.3 iterates to move the violation of condition 3 up the trie while ensuring that condition 4 is not violated in the process. Rotations are not allowed to take place in the *rbTrie* if they will cause the trie structure to fail. So if condition 3 is violated, original condition 4 is marginally modified. While condition 3 is violated, if rotations will affect the trie structure, the nodes are recolored (hence modifying condition 4) and the iteration is allowed to continue. Although in most cases, the trie rebalances by iterations at the upper level of the trie, this is not always guaranteed. Thus the check provided by case 1 (below) inhibits full balancing in the red-black sense. All grey nodes in the figures in this section are black, and unpainted nodes are red.

The following are the discussions of the cases that are marked in the insertion algorithm, Algorithm 3.3.

Case 1: A situation where a rotation will render the trie structure in-operative is depicted in Figure 3.13. In part (a) of Figure 3.13, the trie is red-black balanced, and the insertion of “can” the new internal node $(c, 0)$ renders the trie unbalanced as shown in Figure 3.13b. Without checking for case 1, Figure 3.13c will result, in which one cannot locate keys “but”, “can”, “hate”, and “have”. This is what is termed the loss of the trie structure’s function.

To preserve the trie structure and condition 3, a recoloring as shown in Figure 3.13d is performed. Note that node $(h, 0)$ is set back to black when the algorithm’s iteration terminates. If node $(h, 0)$ were not a root node, and its parent were red, then a rebalancing may occur if the trie rotation condition is not violated again. Note that, if one attempts a migration as in BCTH [47] (see page 45) and proceeds with Algorithm 3.3, the situation in Figure 3.13c will still result. This is because the migration will be reverted by the rotation of case 3.

Figure 3.13 Case 1 of inserting into a rbTrie.

Case 2: Case 2 is executed if the inserted internal node's father and its sibling are both red. If that is the case, a recoloring identical to that of case 1, is performed, and no rotation is performed at the current iteration.

In Figure 3.14 when bucket *b0* in part (a) splits, and node (*b,0*) is formed, condition 3 is violated. Case 1 does not apply, and case 2 applies since (*b,0*)'s uncle (*t,0*) is red. Furthermore condition 4 is preserved after this recoloring (see part (c)). The *while* loop continues with (*b,0*)'s grandfather, node (*h,0*). The iteration continues only if (*h,0*)'s father is red.

Cases 3 and 4: These cases apply when the inserted node's uncle (its father's sibling) is black. Case 3 applies if the inserted internal node (say *x*) is its father's right child. A left rotation is applied to the father of *x*, *f(x)*, to transform it to a left child. Both *x* and *f(x)* are red and so the rotation does not affect the black height of nodes.

After the transformation (if any), case 4 is executed to rebalance the

Figure 3.14 Case 2 of inserting into a rbTrie.

trie. Case 4 colors $f(x)$ black, grandfather of x , $f(f(x))$ red, and right rotates $f(f(x))$. The *while* loop terminates after case 4 is executed, because there are no longer two red nodes connected as father and son.

Figure 3.15 illustrates cases 3 and 4.

Cases 5, 6, and 7 are symmetric to cases 2, 3 and 4 respectively.

3.5.2 Deletion

In most cases when the item to be deleted is located in a bucket, it is simply removed and nothing is done. If the bucket from which an item is deleted becomes empty, it is freed and the external node, say p , in the rbTrie with the freed bucket number is allowed to exist. If the sibling of node p has no freed bucket, then the deletion terminates. Otherwise node p and its sibling are freed and their father set to black (which now becomes an external node with no bucket address). The rbTrie structure guarantees that the colored father node, say $f(p)$, has a sibling. If $f(p)$ were red prior to its coloring, then no red-black condition is violated since two nodes, p and

Figure 3.15 Cases 3 and 4 of inserting into a rbTrie.

its sibling were black (remember external nodes are always black) and were deleted. A red node, $f(p)$, set to black therefore does not alter the black height of the trie.

If $f(p)$ were however black prior to the coloring, then the deletion of p and its sibling causes the black height of the subtrie with p to be one less. The deletion procedure iterates on $x = f(p)$ and moves the extra “black color” up the trie, until:

1. x points to a red node, in which case it is colored black, or
2. x points to the root, or
3. suitable rotations and recoloring can be performed to get the trie red-black balanced.

A complete rbTrie is presented in Figure 3.16, the black-height of the trie is 2. The BCTH equivalence of this rbTrie is shown in Figure 3.12. The figure is constructed from the KWIC (Key-Word-In-Content) index that appeared in [27: 47]. The split keys have been selected exactly as in the BCTH in [47] (which is presented

Figure 3.16 An example of a rbTrie from the insertion of random keys.

as Figure 3.12). Also see Section 3.3 for discussion on the search procedure on this trie. The KWIC index in the order of insertion is presented in Table 3.1. The bucket structure in Figure 3.16 is also used by structures in Figures 3.10 and 3.12.

Table 3.1 The KWIC index of size 35.

1. part	13 methods	25 problems
2. their	14. as	26. from
3. solve	15. obtaining	27. note
4. for	16. a	28. and
5. to	17. its	29. solution
6. an	18. solutions	30. which
7. some	19. use	31. with
8. certain	20. notes	32. method
9. new	21. by	33. computations
10. equation	22. computation	34. the
11. problems	23. in	35. equation
12. on	24. of	

Chapter 4

pTrie: Paginated trie hashing

A pTrie is the pagination of the trie hashing structure in a manner analogous to B-trees. It combines the structural features of trie hashing and the B-tree. Pages in the pTrie can be adjusted to any suitable size and can be stored on disk, thus the problems of degeneracy and durability that hamper the performance of the trie hashing structure are eliminated. As in the prefix B-tree, the pTrie organizes its page content as a variable length string. The pTrie is a balanced structure with a very shallow tree that has a very high branching factor.

In the pTrie, as in the prefix B-tree, the index set and sequence set are organized differently. The external pages in the pTrie point to data pages or data buckets which form the sequence set stored on disk. The usage of the word **bucket** in this thesis also refers to a data page. The pTrie is formed (and grows) by the split actions of the data pages of the sequence set. A page in the pTrie index set is organized as follows:

p_0, df_0	p_1, df_1	p_2, df_2		p_i, df_i		p_{m-1}, df_{m-1}	p_m	Unused
-------------	-------------	-------------	-------	-------------	-------	---------------------	-------	--------

In the above page, the p_i 's are index page pointers if the current page is internal, and are data page pointers if the current page is external. The df_i 's are digit fields and the bars separate consecutive (p, df) pairs. Recall from Section 3.1 that a digit field consists of the pair of digit value and digit number. In the following sections, the search, insertion, range query and deletion algorithms of the pTrie are presented with illustrative examples. The section on implementation discusses in detail the

organization of a page.

4.1 Key search

The search algorithm of the pTrie is not much different from the search in the compact trie hashing structure described in Section 3.3.1. Any search begins at the root of the pTrie. Within an internal page starting with the first *entry* (the term *entry* will be used for the pair (p_i, df_i)) a comparator string, u , is computed and so is the prefix k' , of the search key k . An entry corresponds to a node in the compact trie, and as such, the digit number specifies the position of the comparator that the digit value is to be concatenated with the currently computed comparator string.

If $k' < u$, for u computed at the entry (p_i, df_i) , for $i < m$, then pointer p_i (which leads to a new page) is followed. If $k' \geq u$, then we move to the next entry (p_{i+1}, df_{i+1}) , within the page. At entry (p_{i+1}, df_{i+1}) , a new u and k' are computed and the same decision is made as to move to the next entry within the page or to move on a pointer to another page. If the decision to move to the next entry within a page occurs on the last entry in the page, entry (p_{m-1}, df_{m-1}) , then the last pointer, p_m , of the page is followed to the page it points to. Anytime we move to the next entry within a page or follow the right pointer of the page, the comparator string computed at the current entry where the decision is made, is retained and used to compute the next comparator string. Whenever any pointer p_i (where $p_i \neq p_m$, the last pointer) of a entry (p_i, df_i) is followed, the comparator computed at that entry is discarded and is reset to its previous value.

The search within an external page is the same as that within the internal page, except that the pointers in the page point to data pages. The search for a key k is successful if the data page arrived at contains the key k ; otherwise the search is unsuccessful.

The search algorithm described above is presented in Algorithm 4.1. Consider the example pTrie in Figure 4.3 on page 64. To search for a key $k = \text{"time"}$, the following is carried out.

The root page, page 8, is the first page visited where the comparator string $u = \text{"n"}$ and the prefix of k , $k' = \text{"t"} > \text{"n"} = u$. The right pointer of page 8 is therefore followed since the current entry with DF $(n, 0)$ is the only entry in page 8. At page 7, which contains 2 entries, entry 1 yields a comparator $u = \text{"t"}$ since $\text{DN}(\text{entry } 1) = 0$. Since $k' = \text{"t"} = u$, entry 2 is examined next. At entry 2, $u = \text{"w"} > \text{"t"} = k'$, and so the (left) pointer of entry 2 is followed, which brings the search to page 4. The comparator on entering page 4 is $u = \text{"t"}$ since the current entry's comparator "w" is discarded. Page 4 is an external page — its pointers are data page addresses. There are 3 entries in page 4, the comparator at the first entry is $u = \text{"the"}$, and $k' = \text{"tim"}$, $k' > u$ and so entry 2 of page 4 is examined. At entry 2, $k' = \text{"time"} > \text{"ther"} = u$ and so entry 3 is also examined. At entry 3, $u = \text{"ti"} = k'$, and therefore the last pointer of page 4 is followed. The search thus terminates in data page 12, the last pointer of page 4.

Note that successful and unsuccessful searches take the same amount of time. Every search traverses the height of the pTrie from the root to a data page and therefore the total time spent on a search within the pTrie is of order $O(h)$, where h is the height of the pTrie.

pTrie search algorithm: pTrie-Search(k, pT)

Input: Search key k and pTrie's root, pT .

Output: Bucket address.

Begin

Let e_i be the i^{th} entry within the current page P .

Let $P(e_i)$ be the pointer of e_i .

Let $NUM(P)$ be the number of entries in P .

Initialize comparator $u = \phi$.

Initialize previous comparator $v = \phi$.

Set $P = pT$;

while (P is an internal node)

for ($i = 1; i \leq NUM(P); ++i$)

if ($DN(e_i) > 0$) then

$u = u_0 u_1 \cdots u_{DN(e_i)-1} DV(e_i)$;

else $u = DV(e_i)$;

$k' = k_0 k_1 \cdots k_{|u|-1}$;

if ($k' < u$) then

$P = P(e_i)$;

Move left.

$i = NUM(P) + 1$;

Fail for loop.

$u = v$;

Discard comparator computed at e_i .

else

$v = u$;

Retain comparator.

if ($i = NUM(P)$);

$P = P(p_m)$;

Move right on last pointer in P .

(else)

Move within P with for loop.

endelse;

endfor;

endwhile;

return data page address P ;

P now contain a data page address.

End

Algorithm 4.1: The search algorithm of the pTrie.

4.2 Insertion

Like most file structures, to insert a record $(k, info)$ with key k and detail information $info$, a search must first be performed using the key k to locate the data page in which the record is to be inserted in sequence. If during the search, the key k is

located in the pTrie then a message is issued and the insertion terminates.

If the data page into which the record is to be inserted contains less than c records, then the record is simply inserted into the page and the algorithm terminates successfully. If the data page contains exactly c keys then an overflow or a collision occurs. As in the trie structures discussed in Chapter 3, we split the page into two. Unlike the basic trie, and like the compact trie, the separator of the data pages can be chosen to be the prefix of the exact middle key of the $c+1$ keys. The first collision in the pTrie is exactly the same as in the prefix B-tree. If the separator between the two data pages, say pages b_0 and b_1 is $c_0c_1 \cdots c_t$, then the first insertion yields the following index page:

where there is one entry in the page P , say e_1 , with $DV(e_1) = c_0c_1 \cdots c_t$, $DN(e_1) = 0$, $P(e_1) = b_0$ and $RP(P) = b_1$. In subsequent collisions the separators are themselves used as search argument to the positions where they are supposed to be inserted. The comparator computed up to that position is trimmed from the separator and the remaining partial separator is stored as the digit value with its digit number being the size of the comparator trimmed (see Section 3.3.1). Let us assume that data page b_1 in the figure above splits and produces a separator $c_0c_1 \cdots c_t$. If on using $c_0c_1 \cdots c_t$ as a search argument on arrival at the position where it is to be inserted (entry 2) the comparator string computed is $c_0c_1 \cdots c_t$, then the digit field $(c_{t+1}c_{t+2} \cdots c_t, i+1)$ is stored as the second entry in P , with $P(e_2) = b_1$ and $RP(P) = b_2$, assuming the split yielded b_2 as the new data page. A page becomes full when there is not enough room for an entry presented for insertion into the page. In such a case, the page is split into two.

4.2.1 Page splitting

The splitting of a page selects the entry closest to the middle of the page (in terms of the number of bytes in the page) for promotion to an upper page. The significant

element for promotion is the comparator, u' that is computed at the entry due to be promoted, and not the digit field. Just as in the case of the insertion of a digit value into an index page, the promoted comparator (or separator) u' , is used as an argument in the search algorithm. The comparator computed up to u' is trimmed off from it and the remaining portion stored, its digit number (as before) is the size of the comparator trimmed.

For example, consider the pTrie in Figure 4.1, where data page capacity $c = 4$, in which we assume that index page 1 in part (a) is full. If key “*taught*” is presented for insertion, data page 1 splits and yields a separator “*te*”. By the data page split insertion procedure outlined above, the digit field $(e, 1)$ is supposed to be inserted as the first entry in index page 1, as in part (b). Since page 1 is full, it has to be split. Entry 3 is therefore selected for promotion to the upper page, page 2. We do not promote $DV(e_3) = “er”$, rather the full comparator string computed within entry 3, that is “*ther*”, is promoted. To insert “*ther*” in the father page, as the second entry, a search is first performed with it. At its supposed position, entry 2 in page 2, the comparator computed so far will be “*t*” and so the digit field $(her, 1)$ will be inserted as the DF for entry 2 of page 2. The split structure after the insertion is completed is shown in Figure 4.1c.

The page split and subsequent promotion to parent page may also trigger the parent page to split if it cannot accommodate the new entry to be inserted. Page splitting may propagate to the root page which may itself overflow. This will cause a new root to be generated with one entry whose $DN = 0$. The height of the pTrie grows with a split of its root page.

4.2.2 Further compression

There is the possibility that the inserted entry into an index page will enable the trimming off of some digits from the next entry’s digit value. Compression is possible if the two digit values (the inserted one and the one after it) have common digit numbers and common prefixes. In general, if the inserted digit field is $(k_0k_1k_2 \cdots k_i, dn_k)$ and the next digit field is $(l_0l_1l_2 \cdots l_j, dn_l)$ then if $dn_k = dn_l$ and k and l

Figure 4.1 An illustration of how the page split operation functions for a pTrie. a. pTrie structure before insertion of key *taught*.

b. Overflow on page 1 after bucket split.

c. pTrie after page split

Figure 4.2 An illustration of how further compression is performed during the insertion of an entry into a page.

compare equally up to the first l digits, $i \leq l < j$, and differ at position l , then the next digit field can be represented as:

$$(l_l l_{l+1} \cdots l_j, \text{dn}_l + 1)$$

without loss of the pTrie hashing function. The compression further reduces the size of bytes utilized by entries and goes to enhance the shallowness of the pTrie structure.

As an example, assume that a current page is as shown in Figure 4.2a, and that a digit field $(\text{and}, 0)$ is to be inserted as in Figure 4.2b. Since $(\text{and}, 0)$ and $(\text{ar}, 0)$ have the same digit number, and their digit values “and” and “ar” are equal up to position 0 and differ at position 1, the next digit field can be represented as in Figure 4.2c.

The insertion algorithm of pTrie is presented in Algorithm 4.2. A complete example of a pTrie insertion with the 60 frequently used English words with 3 or more digits is presented in Figure 4.3. The 60 words are shown in Table 4.1 in the order of insertion (and frequency in use). Some discussion of the construction follows.

4.2.2.1 Construction of Figure 4.3

In this example the size of a data page is set at 4. We assume, for the sake of this example, that each index page can hold a maximum of 4 entries. Any data page split moves 2 or 3 keys exactly to the new allocated data page, depending on which move will provide a shorter and convenient separator between the data pages. An index page split moves the exact middle entry to the upper page.

First data page 0 is allocated and the first c keys are hashed into it. The first

Table 4.1 The 60 frequently used English words with 3 or more digits.

1. the	16. was	31. like	46. than
2. and	17. there	32. out	47. them
3. for	18. one	33. when	48. who
4. that	19. they	34. time	49. make
5. you	20. what	35. other	50. may
6. this	21. would	36. more	51. people
7. with	22. any	37. only	52. these
8. not	23. which	38. just	53. their
9. have	24. about	39. end	54. here
10. are	25. get	40. also	55. into
11. from	26. your	41. know	56. first
12. can	27. use	42. how	57. could
13. but	28. some	43. new	58. way
14. will	29. then	44. should	59. had
15. all	30. name	45. been	60. his

Figure 4.3 An example of a pTrie constructed from the keys in Table 4.1.

a. The pTrie structure after the insertion of the 60 keys.

about	that	was	first	been	name	there	will	and	you	had
all	than	what	for	but	new	these	with	any	your	have
also		way	get		not	they	would	are		here
						this	you			
0	1	2	3	4	5	6	7	8	9	10

one	time	should	know	the	when	may	into	can	his
only	use	out	like	them	which	make	just	could	how
people		other		then	who	more		end	
		some		their					
11	12	13	14	15	16	17	18	19	20

b. The buckets after the 60 keys have been inserted

Figure 4.4 The construction of the example pTrie in Figure 4.3.

collision occurs when the key “you” is presented for insertion. This causes a split and produces a separator “t”. We allocate index page 0 and insert the the digit field (t,0) into it as shown in Figure 4.4a.

Progressively, index page 0 becomes full as shown in Figure 4.4c, when the key “one” is presented for insertion, data page 3 splits and a new data page, data page 5, is allocated. Since index page 0 is also full, it splits into pages 0 and 1. The middle entry with digit field (n,0) is promoted into the father page which does not exist, and which is consequently created as page 2 shown in the second part of Figure 4.4c. Again, as indicated in the insertion procedure, it is not the digit value “n” that is promoted, but the comparator computed within the digit field (n,0).

During the 15th split, caused by the key “them”, the separator chosen is “the”,

Table 4.2 The split information of the example pTrie.

Key	Sep	OB	NBA	OP	NPA
1. you	t	0	1		
2. with	w	1	2		
3. are	f	0	3		
4. all	b	0	4		
5. one	n	3	5	0	1, 2
6. they	the	1	6		
7. what	wh	2	7		
8. about	ab	0	8		
9. your	y	7	9	1	3
10. like	l	3	10		
11. out	o	5	11		
12. time	t	6	12		
13. only	ou	11	13	1	4
14. know	k	10	14	0	5
15. them	the	1	15		
16. who	wh	2	16		
17. may	m	14	17		
18. into	i	10	18		
19. could	c	4	19		
20. has	h	10	20	5 2	6 7, 8

Sep -- Separator

OB -- Overflow bucket

OP -- Overflow page

NBA -- New bucket allocated

NPA -- New page allocated

the comparator computed up to “the” is “t”, and hence the digit field ($he, 1$) is to be inserted into page 4, which appears in the first part of Figure 4.4b. Since “he” and “her” have common digit numbers (1) and common prefixes (he), ($her, 1$) is reduced to ($r, 3$). This is shown in page 4 of the fully inserted pTrie in Figure 4.3a. The data pages after the complete insertion are shown in Figure 4.3b.

A complete list of the insertion splitting of index pages and data pages and the separators selected during each split is shown in Table 4.2. The 20 keys are those which caused collisions. The load factor of our example is 0.714.

pTrie insertion algorithm: pTrie-Insert(k, pT)**Input:** Key to insert k and pTrie's root, pT .**Begin**

Let $buckAddr$ be the data page address returned by $pTrie\text{-}Search()$;
 L : $rtnAddr$ be the page address of the new page returned by
 $insert\text{-}in\text{-}page()$; *If a promotion occurs.*
 Let $NUM(P)$ be the number of entries in P ;

 $buckAddr = pTrie\text{-}Search(k, pT)$;
 $read\text{-}bucket(buckAddr, bucket)$;
 search $bucket$ for k ;
 if k is found. then
 report attempt to insert duplicate key;
 return;
 if $(no\text{-}of\text{-}key(bucket) < c)$ then *c is the bucket capacity.*
 insert k into $bucket$;
 write $bucket$ to disk;
 return;
 $split(bucket, separator, newAddr)$; *Function split() splits bucket.¹*
 write old and new buckets to disk;
 $overflow = 1$; *Set while to execute at least once.*
 while ($overflow$)
 $pop\text{-}stack(page)$; *Removes a page from top of stack.²*
 $promote = insert\text{-}in\text{-}page(separator, newAddr, page,$
 $promo\text{-}key, rtnAddr)$; *No promotion.*
 if (**not** $promote$) *Terminates while loop.*
 $overflow = 0$;

Algorithm 4.2: The insertion algorithm of the pTrie.¹It also returns the *separator* of the data pages and the address of the new data page *newAddr*²We assume that during the traversal of the pTrie by $pTrie\text{-}Search()$ algorithm, any page that was visited was pushed onto a stack

```

    else if (page is root page)
        NewRoot = getPage();
        DV( $e_1$ ) = promo-key;
        DN( $e_1$ ) = 0;
        pointer( $e_1$ ) = Addr(page);
        last-pointer(NewRoot) = rtnAddr;
        return;
    else
        separator = promo-key;
        newAddr = rtnAddr;
    endelse;
endwhile;
End

```

Allocate a new page.
Where e_1 is the
first entry in page NewRoot.
The address of page.
Recursively call insert-in-page().

Algorithm 4.2: cont.

**pTrie insertion algorithm's: insert-in-page(*sep*, *newAddr*, *page*,
promo-key, *rtnAddr*);**

Input: Separator to insert *sep*, address of new of page from split, *newAddr*
and the page to insert into.

Output: A separator to be promoted to an upper page, *promo-key*, and the
address of the new page, *rtnAddr*, if a page split occurred.

Begin

Let *u* be a comparator string;

Let *MaxSize* be the maximum page size allowed;

compute *u* using *sep* as argument in pTrie-Search();

Let *I* be the number of digits for which *sep* and *u* prefix digits
compare equally; *From left to right.*

trim off the first *I* digits of *sep*, let resulting partial
separator be string *s'*;

sep = *s'*;

DV(ϵ_i) = *sep*; *ϵ_i is the entry to be inserted.*

DN(ϵ_i) = *I*;

insert ϵ_i into *page*; *Footnote on next page.*

if (ϵ_i is not the last entry in *page*) then

 retrieve(*ndv*, *ndn*); *Retrieve next digit field of ϵ_{i+1} .*

 if (*sep* and *ndv* first *J* digits compare equally and *I* = *ndn*) then

 trim off the first *J* digits of *ndv*, to get *ndv'*;

ndn' = *ndn* + *J*;

 replace(*ndv'*, *ndn'*); *In (*ndv*, *ndn*)'s place in *page*.*

 if (size-used(*page*) > *MaxSize*) then *Overflow.*

 find entry closest to the middle of *page* and compute

 the comparator, *u'* at that entry;

newPage = getPage();

 split *page* between *page* and *newPage*;

 write pages to disk;

promo-key = *u'*;

rtnAddr = Addr(*newPage*);

 return (1); *Signifying a promotion to an upper page.*

 else

 write *page* to disk;

 return (0); *No promotion to upper page.*

End

Algorithm 4.3: The insert-in-page() function of the pTrie insertion algorithm.

4.3 Range query

The pTrie has all the advantages that the B⁺-tree offers. The data pages in the pTrie file are linked together by pointers, this allows sequential processing of the file to be carried out as in the B⁺-tree. To process the file sequentially, the leftmost data page must first be located. This can easily be done by starting with the root page and moving on the (left) pointers of the first entries of the pTrie pages downward to an external page. Since the external page points to data pages, we retrieve the data page that is referenced by the pointer of the first entry of the external page.

Once the first leftmost data page is located, sequential processing of the file is done by moving along the right pointers of the data pages till the last data page's record is processed. The last data page has a sentinel right pointer usually the value *nil*.

Similarly, a range query on pTrie is processed by locating the lower key value of the range within a data page. The location of the lower key (which resides in a data page) is performed by the search algorithm, Algorithm 4.1. Once the lower value is located, the range query can be processed as in the sequential query till a key with value greater than the higher range value is read.

4.4 Deletion

The deletion of a key (with its associated information) will normally take place in a data page of the pTrie file after the key has been located using the search algorithm, pTrie-Search(). After the removal of the key from a data page, the data page may underflow, the affected data page may therefore be merged with its brother data page. A data page underflow occur if the number of keys or key/record pairs within it fall below the minimum number allowed. The modification (or deletion of the digit

³P(ϵ_i) is the page that got divided, and, if ϵ_i is the last entry in *page*, then $RP(page) = newAddr$ else $P(\epsilon_{i+1}) = newAddr$.

field) of the index page in the pTrie whose pointers points to the merging sibling data pages is not as straightforward as in the B-trees.

Data page merging: If two data pages merge, the digit field between them (and one page pointer — the one freed) must be removed from their common parent page. However, before the removal of the digit field, the next digit field (to the right of it) must be partially decompressed. The decompression is the reverse of the further compression technique described in Section 4.2.2. Assume an external page, P , is organized as illustrated.

p_0	dv_k	dn_k	p_1	dv_l	dn_l	p_2	dv_m	dn_m	---	p_t
-------	--------	--------	-------	--------	--------	-------	--------	--------	-----	-------

Let $dv_l = l_0 l_1 \cdots l_i$ and $dv_m = m_0 m_1 \cdots m_j$ and $\text{bucket}(p_i)$ be the data page pointed to by pointer p_i . If data pages $\text{bucket}(p_1)$ and $\text{bucket}(p_2)$ are to be merged into $\text{bucket}(p_1)$, then the digit field (dv_l, dn_l) and pointer p_2 must be removed from P . Before the removal, however, we must partially decompress the digit value dv_m and update its digit number, dn_m . Let the modified digit field of (dv_m, dn_m) be (dv'_m, dn'_m) then if $dn_m \leq dn_l$ then no change takes place i.e., $dv'_m = dv_m$ and $dn'_m = dn_m$. However if $dn_l < dn_m$ then:

$$dv'_m = l_0 l_1 \cdots l_{(dn_m - dn_l) - 1} dv_m \quad dn'_m = dn_l \quad (4.1)$$

$l_0 l_1 \cdots l_{(dn_m - dn_l) - 1}$ is empty if $(dn_m - dn_l) - 1$ is negative.

Index page merging: The merging of index pages is similar to that of data page merging. The difference is in what digit field is decompressed as a result of the removal of an entry from the merging pages' common parent. In page merging, the first digit field in the right page of the two merging index pages is partially decompressed before the content of the two pages are merged. The partial decompression is the same as described in data page merging, the digit field used in the decompression is the digit field due to be removed from the parent page of the merging pages.

The condition for page merging is that the partial decompressed digit field continues to fit onto the concerned page.

Proposition 4.1 *A partial decompressed digit field always fits onto its page after a merge of data pages.*

Proof:

The proof follows from Equation 4.1. Consider the decompression performed as a result of merging data pages $\text{bucket}(p_1)$ and $\text{bucket}(p_2)$ under data page merging above. We consider the following two cases.

Case 1: $dn_m \leq dn_l$. In this case the entire digit field (dv_l, dn_l) and pointer (p_2) are removed from the page.

Case 2: $dn_l < dn_m$. The dv_m increases in size by $|l_0 l_1 \cdots l_{(dn_m - dn_l) - 1}| \leq |dv_l|$, the size of the digit value removed from the page. In addition the one byte pointer (p_2) is also removed from the page.

In either cases, the number of bytes within the index page does not increase as a result of the data page merging. $\square$

Consider Figure 4.5a. If a deletion from data page a causes it to underflow, then we can merge data pages a and b into a , and the digit field $(r, 1)$ must first be decompressed using the digit field $(o, 0)$ which is to be removed. Since $DN(r, 1) > DN(o, 0)$, a new digit field for $(r, 1)$ is computed. By Equation 4.1, the decompressed digit field will be computed as $(\text{"or"}, 0)$. The result of the merge is shown in Figure 4.5b.

The merging of data pages d and e onto d in Figure 4.5b will trigger index pages $P2$ and $P3$ to be merged as well. The merging of $P2$ and $P3$ into $P2$ will require removing digit field $(t, 0)$ which is also used to decompress the first digit value in $P3$. $(hc, 1)$. From Equation 4.1, $(hc, 1)$ decompresses to $(the, 0)$, the result of the merging is shown in Figure 4.5c. We assume that the size of the page is large

Figure 4.5 An illustration of how page merging is performed in the pTrie.

a. pTrie before bucket merge.

b. pTrie after merging buckets *a* and *b* onto *a*.

c. pTrie after the merge of pages P2 and P3 onto P2

enough to retain the contents of the two pages (including the additional value “ t ”, from page 1).

The term *partial decompression* is used because the decompression does not produce a complete front decompression. For example in Figure 4.5b, if data pages e and f merge, the digit field $(r, 3)$ by Equation 4.1 will be replaced by $(her, 1)$ and not the full front decompressed digit field $(ther, 0)$.

4.4.1 Redistribution and Concatenation

Redistribution of keys among data pages is also possible and it will require the modification of the digit field between the data pages whose keys are being redistributed. If the contents of two data pages are redistributed, a new separator is computed for them. The common prefix of the comparator and the new separator is chopped off from it and the remainder is stored as the new digit value. The digit number retained is the size of the comparator chopped off. Thus the insertion of the digit field into the page is the same as in the key insertion.

Redistribution of entries within index pages is also possible, and has two phases:

Phase 1: The construction of a dummy page consisting of the contents of the two merged pages and the entry that separates them in their parent page.

Phase 2: Using the procedure for page splitting, the dummy page is split. That is, the entry closest to the middle of the dummy page’s comparator is selected for promotion to the parent page, to replace the former separator.

In replacing the separator we must go through the algorithmic steps as in the insertion algorithm.

For example in Figure 4.5c if pages $P2$ and $P4$ are to be redistributed to balance the load in them, phase 1 of the redistribution will produce a dummy page as in Figure 4.6a. Assuming phase 2 chooses the entry $(i, 1)$ for promotion, we first

Figure 4.6 An illustration of page redistribution in the pTrie.

compute the comparator at $(i, 1)$, i.e. $u = "ti"$. Phase 2 then inserts the comparator $"ti"$ into page $P1$. Before its insertion however, $"ti"$ is also used as an argument in the search function as in Algorithm 4.3. The resulting redistributed pTrie of Figure 4.5c is as shown in Figure 4.6b.

One may attempt to concatenate index pages if the contents of a page falls to less than half of its allocated size. In concatenation, the separator entry of the two concatenated pages is deleted from the parent page and inserted in the single page formed. Before this action however, the separator is used to decompress its next entry as in data page merging.

An example of concatenation is illustrated by the formation of the dummy page of Figure 4.6a from pages $P2$ and $P4$ of Figure 4.6b.

The deletion algorithm of the pTrie as discussed in this section is presented in Algorithm 4.4. Some of the functions and variables used in the algorithm are as in the insertion algorithm.

pTrie deletion algorithm: pTrie-Delete(k, pT)

Input: Key to delete k and pTrie's root, pT .

Note: Procedures *redistribute()*, *merge()* and *concatenate()* for data pages and index pages are left out, reader may refer to Sections 4.4 and 4.4.1 for details.

Begin

Let *MaxSize* be the maximum page size allowed;

Let *buckAddr* be the data page address returned by pTrie-Search();

Let *NUM(P)* be the number of entries in P ;

buckAddr = pTrie-Search(k, pT);

Stack pages while searching.

read-bucket(*buckAddr*, *bucket*);

search *bucket* for k ;

if k is not found, then

 report key k not found;

 return;

remove k into *bucket*;

if (no-of-key(*bucket*) $\geq c/2$) then

c is the bucket capacity.

 write *bucket* to disk;

 return;

if left-sibling(*bucket*) exist and no-of-keys(left-sibling(*bucket*)) $> c/2$

 redistribute (left-sibling(*bucket*), *bucket*);

 return;

if right-sibling(*bucket*) exist and no-of-keys(right-sibling(*bucket*)) $> c/2$

 redistribute (*bucket*, right-sibling(*bucket*));

 return;

if left-sibling(*bucket*) exist

 merge (left-sibling(*bucket*), *bucket*);

 what-undfl = Address(left-sibling(*bucket*));

else merge (*bucket*, right-sibling(*bucket*));

 what-undfl = *buckAddr*;

pop-stack(*page*);

Removes a page from top of stack.

underflow = delete-from-page(*page*, *what-undfl*);

Algorithm 4.4: The deletion algorithm of the pTrie.

```

while (underflow)
    if page is root
        return;
    if left-sibling(page) exist and size-used(left-sibling(page)) > MaxSize/2
        redistribute (left-sibling(page), page);
        return;
    if right-sibling(page) exist and size-used(right-sibling(page)) > MaxSize/2
        redistribute (page, right-sibling(page));
        return;
    if left-sibling(page) exist
        concatenate (left-sibling(page), page);
    else concatenate (page, right-sibling(page));
    if parent(page) is root
        if (NUM(parent(page)) = 0)
            make the concatenated page the new root;
            return;
            Root page can have one entry.
    if parent(page) underflow
        underflow = TRUE;
        while loop executes again.
        page = parent(page);
    else underflow = FALSE;
        Terminate.
endwhile;
End

```

Algorithm 4.4: cont.

pTrie deletion algorithm's: delete-from-page(*page*, *what-undfl*)

Input: Page *page* and the address of bucket/page to be removed from the page.

Output: An integer, specifying if an underflow occurred or not in *page* after the removal of *what-undfl*.

Begin

Let $NUM(P)$ be the number of entries in page P ;

Let $MaxSize$ be the maximum page size allowed;

for ($i = 0$; $i < NUM(page)$; $++i$)

if ($P(\epsilon_i) = what-undfl$)

The pointer of entry ϵ_i .

if ($i \neq NUM(page) - 1$)

If ϵ_i is not the last entry.

retrieve(ndv , ndn);

Retrieve next digit field of ϵ_{i+1} .

if ($ndn > DN(\epsilon_i)$)

compute a new digit field (ndv , ndn); *See Equation 4.1.*

replace the old (ndv , ndn) with the new one;

endif;

endif;

remove ϵ_i from the page;

if ($size-used(page) \geq MaxSize/2$)

write *page* to disk;

return (0);

No underflow.

else return (1);

Signifying an underflow.

endif;

endfor;

Examine next entry in page.

End

Algorithm 4.5: The delete-from-page() function of the pTrie deletion algorithm.

4.5 Implementation

4.5.1 Page organization

Detailed algorithms of the pTrie file structure have been presented in the preceding sections in this chapter. This section will closely examine how an index page of the structure is organized. The pTrie has been implemented in C language on a Sun SparcStation running SunOS UNIX. A pTrie index page is defined in C-like syntax as follows:

```

struct ptrie_page {
    int          status;
    int          no-of-entries;
    unsigned char content[SIZE-CONTENT];
}

```

The **status** field states whether a page is internal or external. The **content** field's size, **SIZE-CONTENT**, is the number of bytes allocated for a page less $2 * \text{sizeof}(\text{int})$, (i.e., the bytes used by the **status** and **no-of-entries** fields) the field's characters are unsigned. Clearly the **no-of-entries** field states the number of entries or digit fields that are within the page. The organization of information within the content field consist of variable digit values and fixed integers. Specifically, the content field within a page with m entries may be as shown below.

Status	m	P_0	dn_0	len_0	dv_0	P_1	dn_1	len_1	dv_1	P_2	...	P_m	Unused
--------	---	-------	--------	---------	--------	-------	--------	---------	--------	-------	-----	-------	--------

where the p_i 's are pointers, dn_i 's are digit numbers, and len_i 's are the lengths of the digit values, dv_i 's. There is no actual physical separator between the quadruples (p_i, dn_i, len_i, dv_i) , since len_i gives the information on the number of bytes in the variable length dv_i .

The organization of a page in the pTrie structure is similar to that of the simple prefix B-tree or the prefix B-tree. It differs from the B-tree only with respect to what is stored in the content field of the page. Simple prefix B-tree and prefix B-tree do not need the dn_i 's, but variable size partial separators are stored in place of the dv_i 's of the pTrie's page. Prefix B-tree in addition has an area within the content field that stores the κ (see page 23) for the page if one exits.

4.5.2 Other aspects

We present some suggested modifications that can improve the pTrie's performance. The choice of split key during a split in the insertion algorithm can be chosen to be the one that provides the shortest separator for the $c + 1$ keys. This choice provides a shallower tree with a high branching factor since more separators can be packed

into a page. The overhead however is a low utilization for the data pages of the pTrie file. Likewise the promotion of the entry closest to the middle of the splitting page may be altered by selecting the entry that will provide the shortest comparator to be promoted. This has the same consequence as stated above in terms of index page utilization.

A situation in the pTrie structure where there are common prefixes among digit values in the page (irrespective of their digit numbers) is very rare. One cannot therefore trim off common prefixes as in the prefix B-tree. However the probability of having common digit numbers (irrespective of their digit values) cannot be ruled out. The page can be made to absorb more entries if the common digit number in the page is stored once instead of for each entry. The overhead incurred here is a high maintenance cost, since a new entry that is introduced into the page with a different digit number will normally cause a page reorganization and probably a page split.

One may also implement page redistribution as opposed to strict page splits during insertions. This strategy, as noted in Chapter 2, improved the B-tree's search, insert and deletion times.

Chapter 5

Performance evaluation

This chapter compares the performance parameters such as height, execution time, size of separators and page utilization of the file structures developed in this thesis against their counterparts. Specifically, the rbTrie and the compact trie have been implemented and are being compared with respect to their maximum and average heights. Some observations are also drawn from rbTrie and the other trie balancing strategies discussed in Chapter 3. The pTrie has also been implemented and is compared with the simple prefix B-tree and the prefix B-tree. The parameters used in this later comparison includes size of separators, number of pages allocated, disk accesses, height of index, page utilization, load factor and execution times for some N insertions.

The key space is the set of English alphabets (both lower and upper cases). In particular, the keys that will be used for simulation will be drawn from a dictionary consisting of about 50000 words, which we term as the key set D . The structures are implemented in C on the Sun SparcStation.

5.1 Comparison between rbTrie and other tries

The rbTrie is a locally balanced (compact) trie hashing structure using the technique for balancing in the red-black tree. The trie rotation condition however inhibits the rbTrie to be completely balanced unlike its red-black binary tree counterpart.

2 of/de 2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

In ascending ordered insertion, the rbTrie yields a trie that is red-black tree balanced. In a simulation performed, the maximum difference of the black heights in the rbTrie was 4 (see Section 5.1.3). That is, there is a path from the root to an external node that has 4 more black nodes than another. The load factor of the data pages in the rbTrie, like the basic trie hashing is about 70% for random order insertion and 50% for ascending order insertion.

5.1.1 rbTrie versus compact trie

Simulation has been performed on the implementation of the rbTrie and compact trie. Compact trie was chosen for the simulation because of its good performance over the basic Litwin trie, and since rbTrie was implemented on compact trie, it is appropriate to compare these two schemes. Various sizes of words were drawn from the key set D and inserted both in random order and in ascending order. In addition, the buckets size c , of the rbTrie and compact trie files were set at 10 and 20 to see the impact of the size of c on the performance of the rbTrie and compact trie.

The split key in each case (for $c = 10$ and $c = 20$) was selected to vary from the middle key by 1. That is, 3 separators, about the middle of the keys that are involved in the splitting, are computed during each split, the shortest (priority is given to the middle one) of which is chosen as the separator. Because the same split behavior is adapted for both the rbTrie and compact trie, their bucket utilizations compare equally. It is typical for key values chosen randomly to have a load factor of 70% and for ascending order a load factor of 50% [13]. This load factor in many situations also applies to the structures discussed in this thesis, since the same split strategy is adopted for all of them.

Other parameters measured are the longest and average external path lengths of the structures. The path lengths are calculated by traversing the trie for each key inserted. The longest path length is the longest of all paths from the root of the trie to all external nodes, and the average path length is the sum of all paths length from the root to each leaf divided by the number of keys inserted. Let λ , be an

Table 5.1 Compact trie and rbTrie comparison for $c = 10$.

#Keys (x1000)	Average height		Maximum	
	CT	rbT	CT	rbT
1	8.60	8.50	12	10
2	9.53	9.67	13	12
5	10.90	11.22	13	14
10	12.17	12.62	16	16
15	12.76	13.65	16	19
20	15.08	13.92	25	19
25	14.28	14.17	26	19
30	15.06	14.65	27	20

#Keys (x1000)	Average height		Maximum	
	CT	rbT	CT	rbT
1	87.56	8.78	172	13
2	169.63	9.73	338	15
5	426.25	11.25	851	18
10	846.92	12.25	1694	20
15	1271.10	12.95	2540	21
20	1697.49	13.25	3386	22
25	2117.36	13.62	4233	23
30	2544.69	13.96	5082	23

a. Random Insertion, utilization = 70%

b. Sorted Insertion, utilization = 59%

Table 5.2 Compact trie and rbTrie comparison for $c = 20$.

#Keys (x 1000)	Average height		Maximum	
	CT	rbT	CT	rbT
1	7.34	7.28	9	8
2	8.31	8.32	10	9
5	9.67	9.86	12	12
10	10.69	10.96	13	13
15	11.44	11.60	14	15
20	13.12	12.47	21	16
25	12.77	12.35	23	15
30	13.26	12.57	22	16

#Keys (x 1000)	Average height		Maximum	
	CT	rbT	CT	rbT
1	47.06	7.76	91	11
2	92.59	8.79	183	13
5	231.87	10.24	459	16
10	459.61	11.24	917	18
15	690.77	11.80	1377	19
20	918.40	12.24	1833	20
25	1155.20	12.87	2299	21
30	1378.88	12.80	2756	21

a. Random Insertion, utilization = 70%

b. Sorted Insertion, utilization = 55%

external path in a trie corresponding to key k_i , and N the number of keys inserted into the pTrie, then the average path length $\hat{\lambda} = \frac{1}{N} \sum_1^N \lambda_i$.

The results from the simulation are shown in Tables 5.1 and 5.2. The graphs representing the tables are shown in appendix A as Figures A.1 to A.5. Note that the scale of the y -axis of the compact trie and rbTrie for sorted keys insertion differ by 100.

As can be seen in the cases $c = 10$ and $c = 20$, compact trie has a slight advantage over rbTrie in random insertion for a small file size (about $N < 17K$). As N increases, the height of both rbTrie and compact trie increases marginally with that of compact trie exceeding rbTrie. Therefore for large files, rbTrie's search time for random insertion is better than that of compact trie.

In the case of ascending ordered insertion, considering rbTrie, the performance

matches that of the random insertion. The average case of smaller buckets (see Table 5.1) sometimes even outperforms its random counterpart. This is not the case for ordered insertions in the compact trie. The height of the compact trie (both external maximum and average path lengths) deteriorate drastically as the number of keys inserted increases, the increase averages linearly about $800/c$ per 1000 records (approximately 84 per 1000 records for $c = 10$ and 46 per 1000 records for $c = 20$). Ascending order insertion of rbTrie therefore outperforms that of the compact trie by a factor ranging from 10 to 100 as illustrated in the test cases.

5.1.2 rbTrie versus other trie balancing strategies

In comparison to the BCTH discussed in Section 3.3:

1. Balancing in rbTrie is not restricted to subtrees. BCTH balances with respect to nodes with common digit numbers identified as subtrees within the trie. In effect the height of the rbTrie will normally be shorter than that of BCTH.
2. As in their binary search tree counterparts, the rbTrie takes a constant number of rotations to balance the trie after an insertion is made. BCTH takes an average of $\Theta(\log_2 n)$ rotations to balance a subtree with n nodes.
3. The rbTrie however uses more memory space relative to BCTH, since in rbTrie an additional field is needed in a node to store a pointer to its parent. We consider the balance factor field in BCTH and the color field in rbTrie to be the same size.

The reconstruction and optimization algorithms of Torenvliet and van Emde Boas discussed in Section 3.2 even take more time to balance the standard Litwin trie. The steps involved are namely:

- creation of sets BS and NS (building a formal trie),
- reconstruction of the trie from sets BS and NS, and
- optimizing the reconstructed trie,

Table 5.3 The height of rbTrie, for $c = 20$.

#Keys (x 1K)	Black height		Maximum Path	Deviation (Max-Min)	#Keys (x 1K)	Black height		Maximum Path
	Max	Min				Max	Min	
1	4	4	8	0	1	5	5	11
2	4	3	9	1	2	6	6	13
5	6	4	12	2	5	8	8	16
10	7	5	13	2	10	9	9	18
15	7	4	15	3	15	9	9	19
20	9	5	16	4	20	10	10	20
25	8	5	15	3	25	10	10	21
30	8	5	16	3	30	10	10	20

a. Random Insertion

b. Ascending Order Insertion.

are practically inappropriate and expensive to be performed after each insertion. Therefore the steps can only be carried out at some points during file maintenance. Thus the Torenvliet and van Emde Boas algorithm is therefore not truly dynamic. The non-dynamic nature makes the algorithm unattractive for most database management systems which require dynamic file structures. In contrast rbTrie is dynamic with a single phase of "red-black" balancing after each insertion.

5.1.3 Black height of rbTrie

As shown in Table 5.3, ascending ordered insertion generates a trie that is perfectly red-black balanced where the original condition 4 of red-black holds, i.e., every simple path from a node to a descendant leaf contains the same number of black nodes. As can be seen in the table, there is no difference between the minimum and maximum black height of the trie in the case of ascending ordered insertion. This is the case because case I in the rbTrie insertion algorithm always fails, and so rotations are always possible after an internal node is inserted.

The case of random insertion produces variable black heights for some nodes in the rbTrie. The heights of the rbTries as shown in Tables 5.1 and 5.2 compare favorably with that of compact trie and hence the basic Litwin trie. In the simulation, the largest difference between the maximum and minimum black heights is 4, which

occurred for $N = 20000$.

In practise, databases are far in excess of 17K records, and rbTrie therefore has advantage over compact trie in both the random and ordered insertion. One can consider the difference in the heights of rbTrie and compact trie in the case of random insertion as negligible, but the height of rbTrie is far shorter than that of the compact trie in the case of ascending insertion.

5.2 pTrie comparison with prefix B-tree

The pTrie structure discussed in the previous chapter has been implemented together with the simple prefix B-tree and the prefix B-tree. The characteristics of the pTrie can be inferred from the discussions in this section. Various parameters of the structures were measured in a simulation experiment. These parameters are:

1. The average size of the partial separators in the pTrie and in the prefix B-tree.
2. The number of pages allocated per entire insertion of N keys.
3. The number of page accesses to disk (including read and write of both index pages and data pages). At anytime, the root page and the next level to the root are maintained in memory, and therefore their examination do not require access to disk.
4. The height of the structure after the N insertions. That is, the number of pages that are traversed by the search algorithm from the root to a leaf inclusive.
5. The utilization of the page, which is measured as the ratio of the number of bytes used (or utilized) by the pages and the number of bytes allocated.
6. The load factor of the file is also measured. This is the ratio of the number of keys in the file (the N keys inserted) and the maximum number of keys that the file can hold (i.e. the product of number of data pages allocated and the maximum number of keys per data page).

7. The time taken by each of the file structures to build an index for some N records is also measured. In measuring this parameter the UNIX execution time monitoring utility was used.

The split key is selected to vary from the middle key by one as in Section 5.1.1. Thus the load factor of pTrie and the B-trees compare equally under similar conditions. The capacity of a data page is kept constant at 20. In each of the following simulation runs, the size of the keys is set constant at 30, this is to enable more keys to be tested. We discuss the parameters measured in the following two subsection.

5.2.1 Size of partial separators

The only file structure whose properties are related to pTrie is the prefix B-tree. The numerical estimation of the partial separator's size and other parameters of the pTrie is a difficult task. The expected size of the partial separator of the prefix B-tree is even not provided. The problem of estimation becomes even complex when pTrie is involved, since the computation of the digit values is not as straight forward as that of the prefix B-tree. Based on this, simulation has been used to estimate the lengths of the digit values in the pTrie.

The size of the digit value of the pTrie within a page has been compared with the size of the prefix B-tree's partial separators in Table 5.4. In Table 5.4 the page size is varied by 256, 512 and 1024 bytes, for each page size, data size (number of keys) is varied by 5, 10, 20, 30 and 40 thousand. The table reports for both random and ascending ordered insertions.

The results that are obtained for the prefix B-tree matches that in [7]. Typical values of pTrie's digit value never exceed 2 in all the test cases, in addition the behavior of the digit value of the sorted key insertions are only slightly higher than their random counterparts. One significant characteristic of the size of the digit value of the pTrie, is that it decreases as page size increases. This feature is opposite in prefix B-tree. This is due to the fact that pTrie compression is not totally dependent on other digit values within the page. On the contrary, page compression in prefix

Table 5.4 The sizes of the separators in the pTrie and the prefix B-tree.

Page size (bytes)	Number of Keys	Size of PB partial separator		Size of pT digit value	
		random	sorted	random	sorted
256	5000	2.65	3.12	1.58	1.81
	10000	2.81	3.28	1.56	1.81
	20000	2.90	3.35	1.58	1.84
	30000	2.90	3.41	1.60	1.86
	40000	2.98	3.49	1.67	1.91
512	5000	2.80	3.18	1.51	1.77
	10000	3.06	3.47	1.52	1.77
	20000	3.13	3.60	1.53	1.81
	30000	3.15	3.64	1.54	1.83
	40000	3.28	3.72	1.61	1.88
1024	5000	2.80	3.18	1.49	1.75
	10000	3.11	3.50	1.50	1.75
	20000	3.28	3.77	1.51	1.78
	30000	3.47	3.86	1.51	1.80
	40000	3.55	3.97	1.57	1.85

B-tree becomes effective only when (partial) separators within a page share common prefixes. In both the pTrie and the prefix B-tree the increase in the separator's size is not proportional to increases in data size.

The behavior and size of the digit value of the pTrie is very much dependent on the choice of the separators during a data page split. Table 5.5 shows how the choice of split strategy affects the size of the digit values. The size of the digit value approaches 3 for a small page size where the exact middle key's prefix is chosen as a separator (Table 5.5a). The size converges to unity for a large page size as the entire keys involved in the split are scanned for the shortest separator (Table 5.5b).

The characteristic of decreasing digit value size with increasing page size is however invariant to the splitting strategy. It is unlikely that the average size of the digit value will approach 3 in the real database instance, where page sizes are chosen far in excess of 128 bytes. Exact middle key split and shortest key split are the two extreme cases.

In summary, with one more byte used for digit number for each digit value, pTrie attains more compression than prefix B-tree. The feature of decreasing digit value

Table 5.5 The size of digit value of the pTrie, for $c = 20$.

Page Size (bytes)	# Keys (x1000)	Size of DV Random	Size of DV Sorted	Page Size (bytes)	# Keys (x1000)	Size of DV Random	Size of DV Sorted
128	5	2.42	2.72	128	5	1.17	1.14
	10	2.34	2.82		10	1.19	1.14
	20	2.53	2.80		20	1.18	1.13
256	5	2.37	2.66	256	5	1.09	1.09
	10	2.27	2.75		10	1.13	1.09
	20	2.40	2.75		20	1.12	1.08
512	5	2.29	2.63	512	5	1.07	1.06
	10	2.19	2.71		10	1.08	1.06
	20	2.34	2.71		20	1.07	1.06
1024	5	2.26	2.61	1024	5	1.05	1.04
	10	2.16	2.69		10	1.05	1.04
	20	2.31	2.69		20	1.05	1.03

a. Exact middle key for split key computation.

b. Shortest split key computation.

size with increasing page size is becoming more attractive since memory sizes in system are becoming larger and disk space are cheaper.

5.2.2 Height of pTrie

Assuming a typical key size of 30 bytes for a database implemented with a B^+ -tree, we can give an estimation of the height of the pTrie that is used to implement the same database. Assuming a worse case partial separator size of 3 bytes, and a 1 byte digit number, the digit field of the pTrie will use 4 bytes in the worse case.

If the database is built with N records, then the height of the B^+ -tree will be given in the worse case as

$$O\left(\log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2}\right)\right)$$

where m is the order of the B^+ -tree for a fixed page size. The pTrie in this case will have an order of $m \times (30/4) = 7.5m$, we can optimistically set the order to $7m$.

Thus the worse case height of the pTrie will be given as

$$O\left(\log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2}\right)\right). \quad (5.1)$$

Note that as the size of the key (*size-key*) used in the index part of the database grows (say from 30 to 40 bytes), for a fixed page, *m* in the B⁺-tree will fall, but then this interprets to an increase in the ratio *size-key*/4 for the pTrie. Hence, while the height of the B⁺-tree is likely to increase due to an increase of *size-key*, that of pTrie is likely to fall since the logarithmic base of Equation 5.1 is likely to increase.

5.2.3 Other parameters

The results of the simulation for this section are in Appendix B. Figures 5.1 and 5.2 show the results of the number of pages allocated for random and ascending ordered insertions respectively when the page size is 256 bytes. For random insertion, except for the case of $N = 5000$, the number of pages allocated for pTrie is always less (i.e., better) than that of simple prefix or prefix B-trees. Consequently the page utilization of pTrie is higher than those of the prefix B-trees as shown in Appendix B. Nevertheless, the superiority of pTrie compression over the prefix B-trees is seen in the cases where the number of pages used is less. The high page utilization, above 50% for random insertion and above 40% for ascending ordered insertion, in all the structures is due to the page splitting strategy adopted. Page split transfers approximately one-half of the page's contents.

In the ascending ordered insertion however, the results shows a constant superiority of the pTrie over the prefix B-trees with respect to:

1. number of disk accesses, and
2. the number of pages used. This is reflected in a high page utilization. One result, for $N = 47800$ and a page size of 256 bytes, emphasize this by a height of 3 for the pTrie and 4 for the B-trees (see Table B.1).

The data page load factor of the structures is around 70% for the case of random insertion and about 55% for the case of ascending ordered insertion. This result is

Figure 5.1 Number of pages allocated comparison, random insertion. **Figure 5.2** Number of pages allocated comparison, ascending insertion.

similar to those attained in most experiments if the split key is selected very close to the middle key of the splitting bucket, see [33].

Another parameter which shows no significant difference among the structures is the number of disk accesses required to complete the insertion of N keys. In the ascending ordered insertion however, the number of disk accesses of the pTrie is better than those of the B-trees in all cases. The execution time measured confirms results obtained by Bayer and Unterauer in [7]. That is, the time required by the prefix B-tree to complete the insertion algorithm for some N keys, is twice as much as that for a simple prefix B-tree. The execution times measured are graphed in Figures 5.3 and 5.4 for random and ascending ordered insertion respectively. In either cases the simple prefix B-tree execute fastest, followed closely by the pTrie. The execution time of the prefix B-tree is slightly more than twice the time taken by the simple prefix B-tree and about twice that taken by the pTrie for a given N . The times were computed on a machine with only one user. The machine was also the local file system for the directory from which the programs were tested.

The similarity of the results for the simple prefix and the prefix B-trees for small N and bigger page size is expected, since the probability of having common prefixes for separators within a page under these circumstances is quite low. When no common prefixes of separators are realized within a prefix B-tree, it reduces to a simple prefix B-tree.

5.2.4 Remark

The new trie hashing structure introduced — paginated trie hashing, as compared with the most frequently used file structure — the B-trees, has a better search time performance associated with it. The B-trees never have a height advantage over the pTrie, thus the search time (which is in the order of the heights of the structures) of the B-trees cannot be better than that of the pTrie. In disk usage, (or size of the index structure) the pTrie always has an advantage over the B-trees. In particular, in the ascending ordered insertion, the pTrie far outperforms the B-trees.

Figure 5.3 Execution time comparison, random insertion. **Figure 5.4** Execution time comparison, ascending insertion.

The main drawback of the pTrie, compared with the prefix B-tree, is that a binary search cannot be performed within the index page. Binary search on the prefix B-tree is possible, but at an additional processing cost.

5.3 pTrie comparison with other trie hashing structures

The only advantage of the basic trie over the pTrie is that all searches in the pTrie either successful or unsuccessful terminate in a data page. In the basic trie some unsuccessful searches terminate in memory and do not require disk access. This advantage is however lost for very large file since *nil* nodes are rare in such a circumstance.

The pTrie can be used to create an index for a very large database whose index is too big to fit in memory. The basic trie on the other hand, cannot be used to build a very large database since its index needs to be memory resident. The pTrie structure does not require the index to be re-created in the event of system failure since it is by design resident on disk.

The idea of compact trie used by the trie balancing strategies BCTH and rbTrie, is more advantageous in pTrie. This is because in the compact trie, BCTH and rbTrie additional memory is required to store pointers to digit values which are (already) in memory. In the pTrie the exact digit values are stored and not pointers.

In comparison with multilevel trie hashing (MLTH), pTrie is a simple structure. MLTH requires two phases. Namely: the selection of a split node and the splitting of the trie using the split node. In these phases the nodes in the splitting page (subtrie) need to be traversed more than once, the time involved in this process is avoided in the pTrie. In pTrie any entry can be chosen for promotion without destroying the trie hashing property. In a very large file the advantage of having *nil* pointers within the trie such that some unsuccessful search may not require data bucket access, is rare in the MLTH [35]. Therefore any search in the MLTH as in the pTrie, will normally require access to the data buckets.

Since the pages reside on disk, there is no reason why the compact trie strategy cannot be used in the MLTH to reduce the height of the subtries (in the pages) and consequently the overall height of the MLTH.

Chapter 6

Application of pTrie

In the preceding chapters we have developed and analyzed a new file structure called pTrie. In this chapter we will present two of the areas where pTrie can be used. The pTrie by itself can be used for any conventional database as a one dimensional index to the database using the primary or secondary keys. In fact pTrie can be used wherever tries have had applications. We discuss two application areas where the use of pTries can have an impact. Namely: In English spelling dictionary, and in a text editor.

6.1 The pTrie as an English spelling dictionary

Most existing English spell-checkers are "frequency-based", for example one technique is to sort all words in a document and print those which occur infrequently. The list is then manually checked for misspellings. The disadvantage is that habitual misspelled words would escape unnoticed, while unusual words though correctly spelled, would crop up in every spelling check.

Knuth in [27] also suggested an organization of the spelling dictionary intended to reduce the storage requirement. His method removes the suffixes of the words and stores only the stems. In this way, to search for a word, its suffixes are removed and only the stem is checked. The method has the disadvantage of accepting wrong words with legal stems. For example the word "computation" whose suffix is "ion" and

stem "comput" will pass the check. In addition the implementor of the dictionary must have a good knowledge of the contents of the dictionary.

It is suggested in [14] that a dictionary with complete English words and their inflections should be maintained. In general, hashing methods do well in dictionary storage, and in particular double-hashing and hash-bucket methods [14] give fast searches. While these methods are for maintaining the dictionary in memory, it is not clear how they will perform if made disk-resident.

In Chapter 5, pTrie was used to construct a spelling dictionary of up to 47,800 words. Both the index and data pages were made disk resident. The size of the index of a dictionary of about 50,000 in a pTrie is small enough to fit in memory with other applications. Thus entire index of the pTrie's dictionary can be retrieved and brought into memory before a search is initiated. The frequency of access can also be reduced if at least the first 100 commonly used English words are stored in memory. Since the dictionary itself is disk resident its size can be as large as possible and so full words as opposed to stems pose little storage overhead.

If the dictionary's content is relatively static, the load factor of the data pages can be driven up to 100%, equally, the utilization factor of the index pages in the pTrie can also be pushed as high as 80%. In this case, we will have to pre-sort the keys/words of the dictionary in ascending order before insertion. If a collision occurs, the split key will be chosen as the last word. In this way, the first c words are left in the splitting data page, and only one word is moved into the newly allocated data page. Normally, this organization will yield a load factor of 100% and this does not affect the size of the digit field significantly. This is because the size of the digit value for random insertion and that of sorted insertion do not differ much (see Table 5.5).

Similarly, when a page overflows, instead of the normal split strategy of splitting it about the middle string of the page, we will split it such that more entries are left in the page where the overflow occurred. This page splitting strategy improves the page utilization, and hence the height of the pTrie. This split strategy can yield a page utilization of 80% if about 10% of the content of the overflowed page is moved.

6.2 The pTrie as a searching tool in a text editor

A *document* contains both text and a hierarchical structure of sentences, paragraphs, sections, chapters, etc., as well as a provision for lists, quotations, etc. It can also be conceived as a data object, usually textual, though it may also contain other types of data such as photographs, graphs etc. In this section, a document will be considered as a sequence of words. The data structure for a text editor must allow scanning of the text in the same order of this sequence of words, as well as allowing insertion into and deletion from the text, while maintaining the order. It must allow searches by substrings or phrases of the text.

In such an application the objective is to build an index for the text of a size similar to or smaller than the text's size. The traditional model of text in information retrieval is that of a *set of documents* where each document is assigned a list of keywords, that are maintained in a record. The traditional model works well in library applications. However, it still has some problems. Namely:

- The basic structure is assumed (for documents and words).
- Keywords must be extracted from the text for indexing. This is non-trivial and error prone.
- Queries are restricted to keywords.

Some indices also index all words involved in the document except for those deemed to be too common.

Two methods for the indexing of large text databases have been presented by Gonnet et al. in [22] called PAT trees and PAT arrays. PAT trees and PAT arrays are lexicographical indices (indices that are sorted) where the text is considered as a long *string* with each position within it corresponding to a semi-infinite string (or *sistring*). Sistrings start at a position within the text and extend to the end of the text. Each sistring is therefore unique. This view has the following advantages.

- No structure of the text is needed.

- No keywords are used. The queries are based on prefixes of sistrings, i.e. on any substring of the text.

The model is simple and does not restrict the query domain. Almost any file structure can be used to support this view of text. The question is which ones do this efficiently? We will adapt this model or view of text for the implementation of the searching tool using pTrie. But first some notes on PAT trees. A PAT tree is a PATRICIA tree [27; 44] constructed over all possible sistrings of a text. A PATRICIA tree is a binary digital tree where the individual bits of the key are used to decide whether to take a left or right branch.

In PAT trees external nodes are sistrings. For a text of size n , there are n external nodes and $n - 1$ internal nodes. Arbitrary prefix search in the PAT tree takes time $O(\log_2 n)$. For more benefits, PAT trees are implemented such that external nodes are grouped into buckets and the index mapped to disk using supernodes or pages. PAT arrays are an efficient implementation of PAT trees.

6.2.1 Semi-infinite strings

Under the chosen model, a text database consist of a single array of characters numbered sequentially. A sistring is a subsequence of characters from this array, taken from a starting point but going on to the end of (or possible beyond) the actual text. Sistrings are abstract data types, and so present a useful model of text. An example follows.

text retrieving a short text from a textbook ...

Position	Sistring
----------	----------

1	retrieving a short text from a textbook ...
2	etrieving a short text from a textbook ...
3	trieving a short text from a textbook ...
12	a short text from a textbook ...

20	text from a textbook ...
30	a textbook ...
32	textbook ...

The comparison of two sistrings implies the comparison between the two sistrings contents and not their relative positions. The above example of sistrings compare as follows:

$$3 > 32 > 20 > 1 > 2 > 30 > 12$$

The highest sistring of the first 40 sistrings is "xtbook ..." and the lowest is "a short ...".

6.2.2 Indexing points

For some search types, not all points within a text are necessary or even desirable to index. The requirement of having n sistrings in a text of size n might not be necessary for some types of search. Some applications may for instance be interested in searching for words, phases or sentences only, in such a case, only sistrings at the beginning of words are necessary for indexing. This accounts for about 20% of the size of a common English text [22]. What sistrings one indexes is a decision based on the application.

6.2.3 Implementation

The implementation of the searching tool using pTrie is straight forward just as in the basic implementation of pTrie for any database indexing on primary keys. Each sistring is considered as a *key* which is inserted into the pTrie. The external pages will contain pointers to data pages. The data pages contain sistrings, i.e. integer displacements specifying the beginning points of the sistrings in the text. When a search reaches a data page, the data page's content (the sistrings) are checked (or scanned) to determine if the search string is present. The search string is usually a

prefix of a sistring, and there might be more than one sistrings whose prefix match the search string.

If the data page size is large, the search within it may take a longer time. This can be alleviated by the implementation of an indirect binary search within the data pages. That is, the comparison of the middle sistring in the data page with the search string. Remember that references to, and comparison among sistrings refer to their contents within the text and not the integer displacement. When a data page overflows, the split key is selected from one of the sistrings. Its prefix is then promoted to the index part of the pTrie just as in the ordinary implementation.

As an example, Figure 6.1 shows the search tool implementation for the example text: "retrieving a short text from a textbook ...". All the sistrings minus those beginning with blank characters are represented. Clearly, it can be seen that we have achieved the goal of representing the text with an index smaller than its size. In this example, data page capacity is set at 4, and an index page holds a maximum of 4 entries. All split keys are chosen to be the prefixes of the exact middle keys. That is, 2 sistrings remain in the splitting page and 3 are transferred to the new page. The sistrings are arranged in the data pages such that a binary search can be performed on them. Binary search within a data page is significant if the page size is very large.

Signature files [19], use hashing and compression techniques to produce an index being between 10% and 20% of the text size. The storage overhead is small compared with the pTrie's text index. However, the search time in a signature file is linear and for a large text database access may be very slow. In addition, signature files sometimes produce false matches for a search. The PAT trees on the other hand have the same implementation strategy for text databases as the pTrie's design. They both use digital searching, and use disk pages to retain the nodes. The comparison of these techniques is a subject of further work.

Figure 6.1 The pTrie file for a searching tool.

Chapter 7

Conclusion and future work

Trie hashing is one of the fastest access methods for ordered dynamic files. The hashing function which is structured as a binary tree is defined in terms of a *trie*. The trie, as originally proposed, is memory resident with its data pages residing on disk. The retrieval of any record requires at most one disk access.

In this thesis we have identified some of the basic problems of the trie hashing scheme. Namely the trie degeneracy in the event of ascending ordered insertion, and the loss of the index in the event of a system crash. This thesis has also examined some of the early approaches that were offered to resolve these problems. These include the reconstruction and optimization algorithms of Torenvliet and van Emde Boas and the BCTH of Otoo. Though these solutions offered improvements to the basic trie hashing, they have shortcoming themselves. For example the reconstruction and optimization algorithms are very expensive to implement dynamically.

We introduced a new variation of the trie hashing (or compact trie) that attempts to balance the trie structure dynamically using the balancing strategy of the red-black trees. The variation was named rbTrie after its binary tree counterpart. In a performance evaluation that involved the compact trie and the rbTrie, it was observed that the height of the rbTrie in the case of insertions arriving in ascending order is far better (shorter) than that of the compact trie. However their heights are comparable in the case of random insertion. The average height of the compact trie in the ascending order insertion, for example, is about 100 times that of rbTrie for

$N = 25,000$ with a bucket size of 20. Compared to the ascending order insertion, the descending order insertion of the compact trie has improved results. The rbTrie and the compact trie have an average load factor of about 70% for random insertion and 50% for ordered insertion.

The B-tree of Bayer and McCreight, is one of the the most widely used file structure today. We have introduced a new structure called paginated trie hashing or simply pTrie that has the combined merits of the B-trees and trie hashing. The basic B-tree's performances in terms of search, insert and deletion times fall short of pTrie's. This is because the pTrie has a higher branching factor. A variation of the B-tree called prefix B-tree has performance comparable to pTrie's.

Performance evaluation has been performed to compare the pTrie, simple prefix and prefix B-trees. With respect to their variable length field, partial separator for the prefix B-tree and digit field for the pTrie, they compare equally for small page sizes. As the page size increases however, the partial separators of the prefix B-tree increase in size resulting in a lower branching factor for the pages. The digit field of the pTrie decreases in size as page size increases, and hence produces pages with higher branching factor. Thus, for large page sizes the pTrie has a shorter height than the prefix B-tree. On other parameters measured, the B-trees and the pTrie compare almost equally for random insertions. But in the case of ascending order insertions, pTrie has a better performance in terms of the number of pages allocated (with a corresponding high page utilization), and the number of accesses to disk per N insertions. The load factor of the data pages for both structures are the same since the same page splitting strategy was adopted for them. With regards to the execution time for the insertion of some N records, the simple prefix B-tree takes a shorter time than the pTrie. However, the pTrie has a better execution time than the prefix B-tree.

Some application areas of the pTrie were also discussed briefly in the thesis. Though the structure has a good search time when it is used in the indexing of primary (or secondary) keys, it can be used in other areas as well. We have shown how it can be used in the implementation of a dictionary and also in a text retrieval

tool. Experiment has shown that if the spell checking dictionary is fairly static (as in most cases), a load factor of 100% is attainable in the pTrie. In addition index page utilization can be set as high as 80%. Like other digital hashing structures, pTrie is a suitable structure for implementing a text retrieval tool for text databases.

The main drawback of the pTrie has been identified to be the inability to perform a binary search within the page. One can also perform a binary search on the prefix B-tree's page, but at a cost. A binary search on a page is necessary when the page size is very large.

Future Work

Database systems typically operate in a multi-user environment, it is therefore necessary to ensure that concurrent access to an index is not a bottleneck in the system. Concurrency controls in trie hashing and its variations (not including pTrie) are straight forward, though not much research has been done in this area. Litwin et al. addressed the concurrency control of trie hashing in [34]. The algorithm they presented locks only leaf nodes, and at most two nodes need to be locked simultaneously by any operation regardless of the number of buckets being accessed.

With regard to concurrency issues, pTrie is more a B-tree than it is a trie. Concurrency control is one aspect of the B-trees that is well addressed in the literature. Most prominent ones are the Bayer-Schkolnick algorithms [6], the B-link tree algorithms [30; 29] and the Lehman-Yao algorithms [30]. In a study in [49] it was concluded that the B-link algorithms perform better among most of the B-tree concurrency control algorithms. Several other concurrency algorithms are presented in [10; 28; 29; 43] and [48].

It is apparent that most of the concurrency mechanism of the B-trees can be applied to the pTrie directly with little or no change at all. This however does not suggest a unique pTrie-based concurrency control algorithm should not be sought for. Future investigation into the concurrency mechanism of the pTrie is required if the structure is to remain competitive with its rivals. This is particularly important since the digit fields of pTrie are very sensitive to key search.

The theory of the pTrie, as presented here, lacks a well formulated mathematical analysis. A detailed analysis is also a subject of further work.

While pTrie may be adopted for other application areas, the effectiveness of pTrie compared with the PAT tree is an interesting area to be looked into in the area of text retrieval. With the help of PAT tree, researchers at the Centre for the New Oxford English Dictionary at the University of Waterloo were able to reduce the time of probably 3.4 years of inserting 600Mb of text into a PATRICIA tree (in any form) to a period *during a weekend* [22]. The Centre is still in search of an effective text retrieval structure.

References

- [1] ADEL'SON-VELSKI, G., AND LANDIS, E. An algorithm for the organization of information. *Soviet Mathematics Doklady* 3 (1962), 1259-1263.
- [2] AHO, A., HOPCROFT, J., AND ULLMAN, J. *The design and analysis of algorithms*. Addison-Wesley, Readings, Mass., 1974.
- [3] BAEZA-YATES, R. String searching algorithms. In *Information retrieval: Data structures & algorithms* (Englewood Cliffs, N.J., 1992), Prentice-Hall, pp. 219-240.
- [4] BAYER, R. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* 1 (1972), 290-306.
- [5] BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Informatica* 1, 3 (1972), 173-189.
- [6] BAYER, R., AND SCHKOLNICK, M. Concurrency operations on B-trees. *Acta Informatica* 9 (1977), 1-21.
- [7] BAYER, R., AND UNTERAUER, K. Prefix B-trees. *ACM Transactions on Database Systems* 2, 1 (March 1977), 11-26.
- [8] BENTLEY, J., AND FRIEDMAN, J. Data structures for range searching. *ACM Computing Surveys* 11, 4 (Dec. 1979), 397-409.
- [9] BILIRIS, A. Operation specific locking in B-trees. In *Proceedings of the Sixth ACM Symposium on PODS* (San Diego, California, 1987), pp. 159-169.

- [10] CAREY, M., AND STONEBRAKER, M. The performance of concurrency control algorithms for database management systems. In *Proceedings of the Tenth VLDB Conference* (Singapore, Aug. 1984).
- [11] CESARINI, F., AND SODA, G. Binary trees paging. *Information Systems* 7, 4 (1982), 337-344.
- [12] CHANG, H., AND IYENGAR, S. Efficient algorithms to globally balance a binary search tree. *Communications of ACM* 27, 7 (July 1984).
- [13] COMER, D. The ubiquitous B-tree. *ACM Computing Surveys* 11, 2 (June 1979), 21-137.
- [14] COMER, D., AND SHEN, V. Hash-bucket search: A fast technique for searching an english spelling dictionary. *Software-Practice and experience* 12 (March 1980), 669-682.
- [15] CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction to algorithms*. MIT Press, Cambridge, Mass., 1989.
- [16] DAY, A., AND IYENGAR, S. Balancing a binary tree. *Computing Journal* 19, 4 (Nov. 1976), 360-361.
- [17] ELLIS, C. Concurrent search and insertion in 2-3 trees. *Acta Informatica* 14 (1980), 63-68.
- [18] ENBODY, R., AND DU, H. Dynamic hashing schemes. *ACM Computing Surveys* 20, 2 (June 1988), 85-113.
- [19] FALOUTSOS, C. Signature files. In *Information retrieval: Data structures & algorithms* (Englewood Cliffs, N.J., 1992), Prentice-Hall, pp. 219-240.
- [20] FOLK, M., AND ZOELLICK, B. *File structures: A conceptual toolkit*. Addison-Wesley, Readings, Mass., 1987.

- [21] FRAKES, W., AND BAEZA-YATES, R., Eds. *Information retrieval: Data structures & algorithms*. Prentice-Hall, Englewood Cliffs, N.J., 1992.
- [22] GONNET, G., BAEZA-YATES, R., AND SNIDER, T. New indices for text: PAT trees and PAT arrays. In *Information retrieval: Data structures & algorithms* (Englewood Cliffs, N.J., 1992), Prentice-Hall, pp. 66-82.
- [23] GUIBAS, L., AND SEDGEWICK, R. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science* (January 1978), pp. 8-21. IEEE Computer Society.
- [24] KARLTON, P., FULLER, S., SCROGGS, R., AND KAEHLER, E. Performance of height-balanced trees. *Communications of ACM* 19, 1 (Jan. 1976), 23-28.
- [25] KELLER, A., AND WIEDERHOLD, G. Concurrent use of B-trees with variable-length entries. *SIGMOD Record* 17, 2 (1988).
- [26] KNUTH, D. Optimum binary search trees. *Acta Informatica* 1 (1971), 14-25.
- [27] KNUTH, D. *The art of computer programming Vol. 3: Sorting and searching*. Addison-Wesley, Readings, Mass., 1973.
- [28] KWONG, Y., AND WOOD, D. A new method for concurrency in B-trees. *IEEE Transactions on Software Engineering* 8, 3 (May 1982), 211-222.
- [29] LANIN, V., AND SHASHA, D. A symmetric concurrent B-tree algorithm. In *Proceedings of the Fall Joint Computer Conference* (1986), pp. 380-389.
- [30] LEHMAN, P., AND YAO, S. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems* 6, 4 (Dec. 1981), 650-670.
- [31] LITWIN, W. Trie hashing. In *Proc. ACM SIGMOD Conf* (Ann Arbor, Michigan, May 1981), pp. 19-29.
- [32] LITWIN, W. Trie hashing: Further properties and performance. In *Intl. Conf. on Foundation of Data Organization* (Kyoto, Japan, May 1985), Plenum Press.

- [33] LITWIN, W., ROUSSOPOULOS, N., LEVY, G., AND HONG, W. Trie hashing with controlled load. *IEEE Transactions on Software Engineering* 17, 7 (July 1991), 678-691.
- [34] LITWIN, W., SAGIV, Y., AND VIDYASANKAR, K. Concurrency and trie hashing. *Acta Informatica* 26 (1989), 597-614.
- [35] LITWIN, W., ZEGOUR, D., AND LEVY, G. Multilevel trie hashing. In *Extending Database Technology* (New York, NY, March 1988), Springer-Verlag, pp. 309-335.
- [36] LOMET, D. Digital B-trees. In *Proceedings of the 7th International Conference on Very Large Databases* (Cannes, France, 1981), pp. 333-343.
- [37] LOMET, D. A high performance, universal, key associative access method. In *Proc. ACM SIGMOD Conf* (San Jose, California, 1983), pp. 120-132.
- [38] MALY, K. Compressed tries. *Communications of ACM* 19, 7 (July 1976), 410-415.
- [39] MANBER, U., AND BAEZA-YATES, R. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters* 37 (Feb. 1991), 133-136.
- [40] MAURER, W., AND LEWIS, T. Hash table methods. *ACM Computing Surveys* 7, 1 (March 1975), 5-19.
- [41] MCCREIGHT, E. Pagination of B*-trees with variable-length records. *Communications of ACM* 20, 9 (September 1977), 670-674.
- [42] MERRET, T., AND FAYERMAN, B. Dynamic patricia. In *Int. Conf. on Foundation of Data Organization* (Kyoto, Japan, May 1985), Plenum Press, pp. 19-29.
- [43] MOND, Y., AND RAZ, Y. Concurrency control in B⁺-trees databases using preparatory operations. In *Proceedings of the 11th VLDB Conference* (Stockholm, Sweden, 1985), pp. 331-334.

- [44] MORRISON, D. PATRICIA: Pratical algorithm to retrieve information coded in alphanumeric. *JACM* 15 (1968), 514-534.
- [45] NIEVERGEIT, J. Binary search trees and file organization. *ACM Computing Surveys* 6, 3 (Sept. 1974), 175-207.
- [46] OTOO, E. Searching on alphanumeric keys using local balanced trie hashing. Tech. Rep. No. 120, Sch. of Comp. Sc., Carleton U, Ottawa, August 1987.
- [47] OTOO, E. Locally balanced compact trie hashing. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases* (Jerusalem, Isreal, June 1988).
- [48] OTOO, E. Efficient concurrency control protocols for B-tree indexes. Tech. Rep. No. 166, Sch. of Comp. Sc., Carleton U, Ottawa, January 1990.
- [49] SRINIVASAN, V., AND CAREY, M. Performance of B-tree concurrency control algorithms. *SIGMOD Record* 20, 2 (June 1991), 416-425.
- [50] TENENBAUM, A., LANGSAM, Y., AND AUGENSTEIN, M. *Data structures using C*, 2nd ed. Prentice-Hall, Englewood Cliffs, N.J., 1990.
- [51] TORENVLIET L., AND VAN EMDE BOAS, P. The reconstruction and optimization of trie hashing functions. In *Proceedings of the 9th International Conference on Very Large Databases* (Florence, Italy, November 1983), pp. 142-156.

Appendix A

Graphs of rbTrie and compact trie comparison

Figure A.1 Average height comparison $c = 10$, random insertion. **Figure A.2** Maximum height comparison $c = 10$, random insertion.

Figure A.3 Average height comparison $c = 10$, sorted insertion.

Figure A.4 Average height comparison $c = 20$, random insertion. **Figure A.5** Maximum height comparison $c = 20$, random insertion.

Appendix B

Tables of pTrie and B-trees comparison

Table B.1 Comparison for page size of 256 bytes.

Method	5000 random Load Factor = 0.696				5000 ordered Load Factor = 0.545			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	16	10706	2	0.70	36	11509	3	0.42
Prefix	16	10706	2	0.69	36	11430	3	0.42
pTrie	17	10708	2	0.64	31	11042	3	0.47

Method	20000 random Load Factor = 0.704				20000 ordered Load Factor = 0.546			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	73	52847	3	0.67	156	59575	3	0.42
Prefix	71	52766	3	0.64	148	59580	3	0.42
pTrie	61	53964	3	0.70	123	58929	3	0.47

Method	47800 random Load Factor = 0.700				47800 ordered Load Factor = 0.545			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	189	141411	3	0.65	399	152087	4	0.42
Prefix	174	141358	3	0.64	363	149933	4	0.43
pTrie	160	140962	3	0.66	298	147799	3	0.47

Table B.2 Comparison for page size of 512 bytes.

Method	5000 random Load Factor = 0.696				5000 ordered Load Factor = 0.545			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	8	10690	2	0.69	17	10908	2	0.44
Prefix	8	10690	2	0.69	17	10908	2	0.44
pTrie	9	10692	2	0.60	15	10904	2	0.47

Method	20000 random Load Factor = 0.704				20000 ordered Load Factor = 0.546			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	35	42866	2	0.67	74	47403	3	0.43
Prefix	33	42862	2	0.69	72	47343	3	0.44
pTrie	31	42860	2	0.67	59	44216	3	0.48

Method	47800 random Load Factor = 0.700				47800 ordered Load Factor = 0.545			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	93	116109	3	0.65	188	137697	3	0.43
Prefix	91	113476	3	0.62	176	137389	3	0.44
pTrie	77	115337	3	0.67	142	133365	3	0.49

Table B.3 Comparison for page size of 1024 bytes.

Method	5000 random Load Factor = 0.696				5000 ordered Load Factor = 0.545			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	5	10684	2	0.55	9	10892	2	0.41
Prefix	5	10684	2	0.55	9	10892	2	0.41
pTrie	5	10684	2	0.53	8	10890	2	0.44

Method	20000 random Load Factor = 0.704				20000 ordered Load Factor = 0.546			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	17	43830	2	0.69	36	43694	2	0.44
Prefix	17	42830	2	0.68	36	43694	2	0.44
pTrie	16	42830	2	0.64	28	43678	2	0.50

Method	47800 random Load Factor = 0.700				47800 ordered Load Factor = 0.545			
	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.	1 # Pages	2 # Disk access	3 Height	4 Page Utiliz.
SimPrefix	46	102476	2	0.65	90	104504	2	0.45
Prefix	45	102474	2	0.63	86	104496	2	0.45
pTrie	36	102456	2	0.71	68	104460	2	0.50

END

1 9-07-94

FIN