Algorithms for Generating

and

Coding B-Trees

By

Mounir BELBARAKA

A thesis submitted to
the School of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
Department of Computer Science
University of Ottawa
Ottawa, Ontario

ISBN 0-612-16401-2

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

The undersigned recommend to the School of Graduate Studies

and Research acceptance of the thesis

"ALGORITHMS FOR GENERATING AND CODING OF B-TREES"

submitted by Mounir Belbaraka

in partial fulfillment of the requirements for

the degree of Master of Computer Science

Thesis Supervisor (Dr Ivan Stojmenovic)

Department of Computer Science

University of Ottawa

Submitted: December 1995

Abstract

Gupta, Lee and Wong described algorithms for generating 2-3 trees and B-trees with a given number of nodes and left as open problems whether algorithms exist that generate them in lexicographic order, and whether it is possible to generate 2-3 trees [GLW] or B-Trees [GLW1] in constant average delay, exclusive of the output. In this thesis, we propose solutions to the open problems in both [GLW] and [GLW1].

The main results of this thesis are: introducing a new notation of B-Trees which provides lexicographic order and a proof that [GLW] and [GLW1] algorithms do have a constant average delay (thus solving two open problems posed by Gupta, Lee and Wong).

A new algorithm for generating 2-3 trees, with a given number of nodes, in lexicographic order is also presented. This algorithm is an improvement over [GLW] in terms of time complexity and storage. An algorithm for lexicographic generation of B-Trees with a given number of leaves is described. Finally new algorithms for coding and decoding B-Trees sequentially are described.

Acknowledgments

I wish to acknowledge several persons who directly or indirectly contributed to this thesis. First, I would like to thank my supervisor, Dr Ivan Stojmenovic, for having given me the chance to do my research with him and whose supervision, while very strict, was very helpful and friendly. Dr Ivan Stojmenovic was always open to suggestions and was a great source of encouragement and inspiration for me during the writing of this thesis. His patience and flexibility to work with me at the most outrageous hours, has helped me combine the pursuit of a full time job with the writing of this thesis. I thank him for understanding my frequent switch of priorities.

I would like to thank the Department of Computer Science of the University of Ottawa for giving me the opportunity to do this research as well as the administrative and technical backup crucial to any research.

Last, but certainly not least, I would like to thank my parents (whose occasional financial support made university life bearable) and my wife Carole, who not only had to accept my frequent week ends and evenings spent away from home in the computer science lab, but also encouraged me in doing so, understanding that education is not a chore but the most important privilege a person can get.

TABLE OF CONTENTS

Figures

Chapter I

Introduction

1.1 Definitions

Trees are a type of data structure that are widely used in many fields of modern science and technology. They can also be found in diverse areas of the computer science field and a number of different kinds of trees is studied in the literature. Examples of studied trees are: binary trees and t-ary trees, AVL trees and B-Trees.

This research will be restricted to 2-3 trees, B-Trees and their generation and coding.

1.1.1 B-Trees.

B-Trees are a special type of trees whose properties make them useful for storing and retrieving information. In one aspect, the longest path between the root and a leaf will always be $O(\log_m n)$ (where m is the order of the B-Tree and n is the number of nodes).

A B-tree of *order* m is defined as a tree satisfying the following properties[BM]:

1. all leaves are on the same level;
2. the root has k descendants $2 \leq k \leq m$;
3. other internal nodes have k' descendants $\lceil m/2 \rceil \leq k' \leq m$

   where $\lceil x \rceil$ is the **smallest integer** $\geq x$ (i.e. $\lceil 5/2 \rceil = 3$).

A B-Tree of order 4 is shown in Figure 1.

In the tree family, B-Trees remain unique as insertion (and deletion) algorithms are different from those used for an AVL tree and unlike binary trees, each node may contain many elements and may have up to m children. Such a characteristic implies that searching for a node will always be done by using very short path lengths and more nodes can be accessed.

As an example, in a B-Tree of order 4, each node (except maybe the root) will contain between 2 and 4 elements. Therefore, between 2 and 4 elements can be accessed, which is an improvement over a binary or AVL tree which can access only one element per node.

A B-Tree of order 4 where the number sequence 10 20 30 40 50 60 70 80 has been inserted is shown in Figure 1 (in fact it is an example of a search B-Tree, a search B-Tree being a B-Tree whose nodes contain data and where the data is inserted following an insertion algorithm that compares data values).

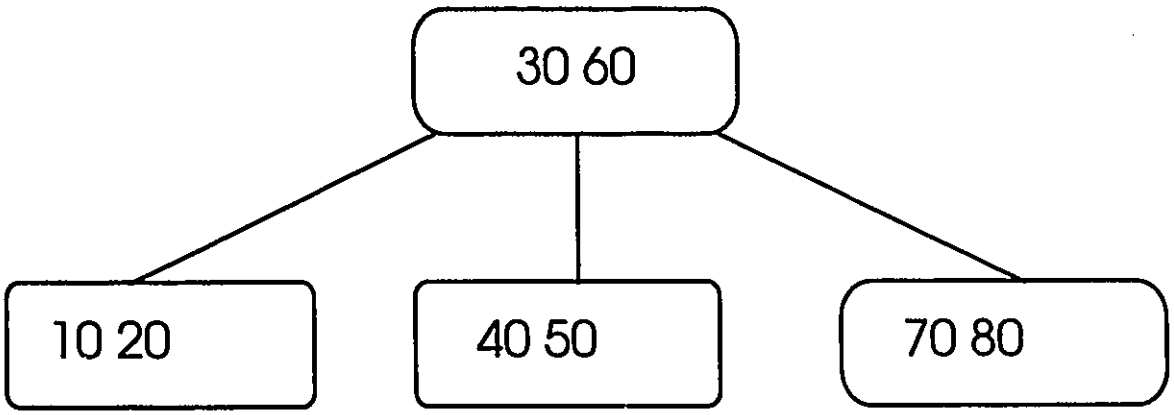

Figure 1

1.1.2 2-3 trees.

The attractive concept of 2-3 trees was first introduced by J. Hopcroft in the mid-seventies. A 2-3 tree is a special case of a B-Tree (a B-Tree of order 3). All of its internal nodes (i.e.: non leaf nodes) have either 2 or 3 children. The leaf nodes have no children and, as any other node, own one or two data items. As all the leaves are at the same level, every path from the root to a leaf is of the same depth.

Because the number of data items that is contained in a 2-3 tree of height h ranges from 2^{h-1} to 3^{h-1} (the height being defined as the number of levels) can be used either as a heap or as a search tree [AHU], 2-3 trees are an easy type of tree to implement and an excellent tool for storing and retrieving information for the following reasons:

- re-balancing is not necessary;
- logarithmic accessing time is guaranteed;
- accessing the 2-3 tree is easier than for a B-Tree of higher order;
- contrary to AVL and balanced trees, they are free of additional information for implementation (such as keeping track of the difference of height between the left and the right sib-tree...);

(For more information on 2-3 trees see [AHU].)

1.2 B-Tree representations

In this thesis, two types of B-Tree representations, defined in the next paragraph, are used in the algorithms in the fifth chapter.

- **B-Tree Parent-Child representation**

A B-Tree can be represented as a pointer to a node. Such node is a record that contains data, pointers to children nodes and a pointer to the parent node. It can be represented with arrays holding the information about the parent and children for each node.

For example, this pointer to a node is defined in the following Pascal-like statements:

```
const order = 6;
    maxvalues = 7      { order+1 };
    maxsubtrees = 8    {maxvalues + 1 };
type
   btree = ^node;
   valuelist = array [1..maxvalues] of integer;
   btreelist = array [1..maxsubtrees] of btree;
   node = record
                nvalues: integer;
                values: valuelist;
                subtrees: btreelist;
          end.
```

A B-Tree is graphically represented as a tree where all paths from the root to the leaves are of the same height, and where the nodes are numbered as in [GLW1]: starting from the lowest level (or bottom level), each node is listed in an increasing order, level by level ending with the root. Within the same level the listing is done from left to right.
An example of node numbering is shown in Figure 2.

Note: Figure 2 shows the node numbering, not the node value as the contents of the nodes are irrelevant in this research; only the data giving node indices or pointers to parent and children nodes are considered.

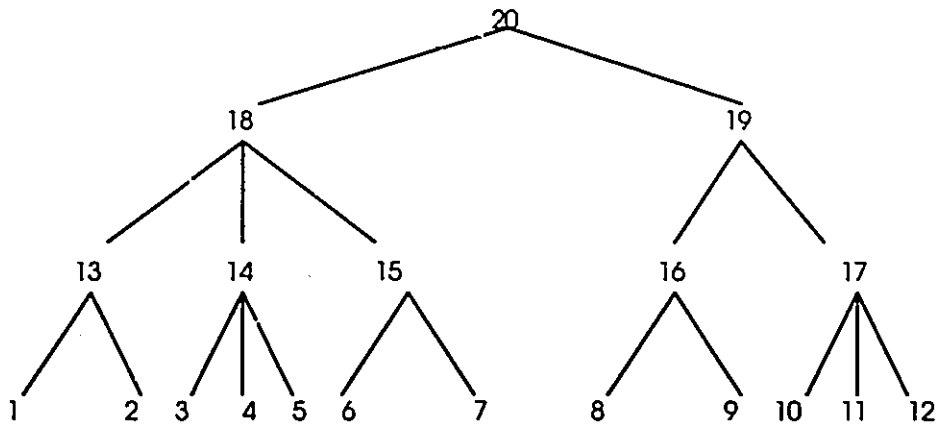

Figure 2

The next figure, Figure 3 , shows the corresponding B-Tree Parent-Child representation.

Node#	Parent	child1	child2	child3
1	13	0	0	0
2	13	0	0	0
3	14	0	0	0
4	14	0	0	0
5	14	0	0	0
6	15	0	0	0
7	15	0	0	0
8	16	0	0	0
9	16	0	0	0
10	17	0	0	0
11	17	0	0	0
12	17	0	0	0
13	18	1	2	0
14	18	3	4	5
15	18	6	7	0
16	18	8	9	0
17	19	10	11	12
18	20	13	14	15
19	20	16	17	0
20	0	18	19	0

Figure 3

Such numbering is important as the purpose of chapter V of this thesis is to establish a relationship between parent and children nodes.

- **Sequence representation**

The same B-Tree notation as [GLW1] is used to explain sequence representation; any B-Tree is represented by a sequence $a_1a_2...a_k$ obtained as follows:

starting with the lowest level of internal nodes, the number of descendants of each node is listed from left to right, level by level, ending with the root.

For example, the sequence corresponding to the tree in Figure 2, according to our representation, is 2 3 2 2 3 3 2 2 = $a_1a_2a_3a_4a_5a_6a_7a_8$ (denoted as a1a2a3...a8 in Figure 4). An illustration is given in Figure 4 for that same sequence <a> where $a_1a_2a_3a_4a_5a_6a_7a_8$ is the sequence 2 3 2 2 3 3 2 2 .

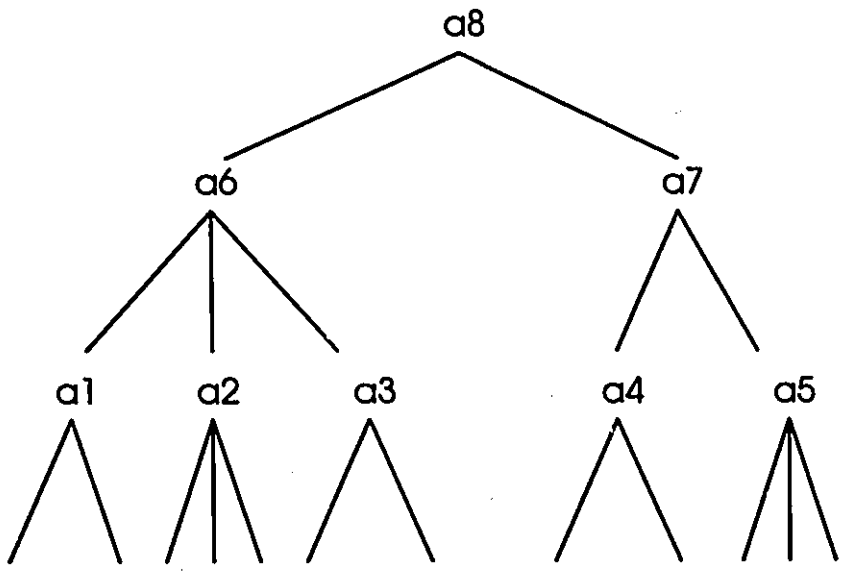

Figure 4

The total number of nodes, n, consisting of both the leaves and the internal nodes is given by:

$$n = (\sum_{i=1}^{k} a_i) + 1$$

Note: While Figure 2 and Figure 4 might be identical in shape, there is a difference between the two that resides in the numbering fashion. Figure 2 gives the node numbering (or node indices) while Figure 4 can also be seen as the internal nodes numbering (every a_i gives the number of children of parent i as will be shown in Chapters 3 and 5).

• B-Trees sequence representation.

As in [GLW1], a B-Tree of n leaves (and hence n-1 keys) will be represented by an a-sequence $a_1a_2...a_k$. (defined below). This a-sequence represents the reading, at every level, of the number of children in the B-Tree, from left to right, starting from the lowest level of internal nodes up to the root.

The sequence $a_1...a_k$ representing a B-Tree with n leaves must have the following properties [GLW]:

1. $a_1 + a_2 + ... + a_k + 1 = n + k$;
2. $a_i \in \{\lceil m/2 \rceil, \lceil m/2 \rceil + 1, ..., m\}$, i=1,2,...,k; $\quad 2 \le a_k \le m$
3. if $k > 1$ then there exists $l_0, l_1, ..., l_r$ such that
   - i) $0 = l_0 < l_1 < ... < l_{r-1} = k-1$, $\quad$ with $l_r = k$
   - ii) $\sum_{i=1}^{b} a_i = a_1 + ... + a_{l_1} = n$ (where $b = l_1$ = number of nodes at level 1)

iii) $l_i - l_{i-1} = a_{l_i+1} + \ldots + a_{l_{i+1}}$ for i=1,2,...,r-1 (defines l_{i+1}).

Note: As 2-3 trees are B-trees of order 3, the previous properties also apply to them.

- An **a-sequence** is a sequence that satisfies all of the above three properties (in which case it is said to be *feasible*), and in [GLW], it is shown that a-sequences $a_1 a_2 \ldots a_k$ are in one to one correspondence to B-Trees with n leaves (for $n = a_1 + a_2 + \ldots + a_k - k - 1$).

The first and the second properties previously defined deal with the number of nodes and the valid integer values of the a-sequence. The third property says that every a-sequence should be partitionable (i.e. can be decomposed) into levels such that the number of nodes on a particular level is equal to the sum of the number of nodes at the next higher level, and the highest level (the root) has exactly one node (i.e:Figure 5).

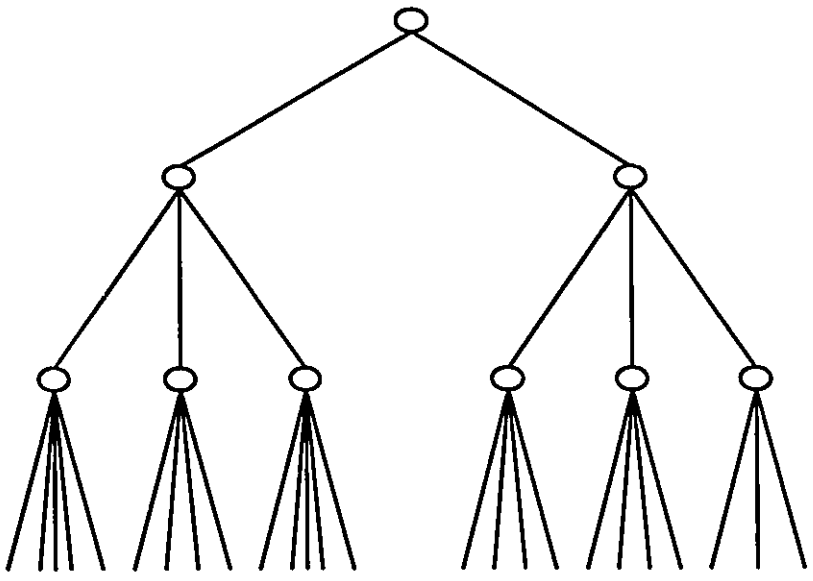

Figure 5

The a-sequence 5 4 5 4 4 3 3 3 2 in Figure 5 is partitionable as follows:

5 4 5 4 4 3 are at the first non-leaf level (6 elements), 3 3 (3 + 3 = 6, the 6 elements at

the previous level) at the second level (and make 2 elements), ...s being the root.

- An **l-sequence** $l_0 l_1 ... l_r$ is defined as a sequence of level sizes where $l_0 = 0$, l_1 is the number of internal nodes at the lowest level, l_2 the total number of internal nodes at the lowest two levels, etc.

In fact, in the sequence $l_0 l_1 ... l_r$, l_1 is the number of internal nodes at the lowest level (level 1), l_2 is the total number of internal nodes at the second lowest level and so on. Level i is a subsequence $a(l_{i-1} + 1)$, ..., $a(l_i)$ and the number $u_i = l_i - l_{i-1}$ will be referred to at its size (i.e. the number of nodes at level i). The size of a given level i is determined by

$$\lceil u_{i-1}/m \rceil \leq u_i \leq \lfloor (u_{i-1}) / (\lceil m/2 \rceil) \rfloor$$

(where $\lceil x \rceil$ represents the smallest integer $\geq x$ and $\lfloor x \rfloor$ the greatest integer $\leq x$)

For example, in the a-sequence 3553333432 which corresponds to a B-tree of order 5 with 25 leaves (because 3553333432 is decomposed by levels as 3553333 43 2 : in other words 3553333 at the first level, 43 at the second level, 2 at the root), the l-sequence is determined as follows:

$l_1 = 7$ since $u_1 = l_1 - l_0 = 7$: the 7 first elements have a sum of 25 (3+5+5+3+3+3+3) which is the number of leaves,

$l_2 = 9$ and since $u_2 = l_2 - l_1 = 9 - 7 = 2$ then the number of elements at that level is equal to 2 and those two elements have a sum of 7 (3 + 4) which is the number of internal nodes at that level,

$l_3 = 10$ and since $u_3 = 10 - 9 = 1$ then there is only one element at that level equal to 2.

- A ***normalized subsequence*** $b(l_i+1),...,b(l_{i+1})$ is associated with each ***a-subsequence*** of the form $a(l_i+1),...,a(l_{i+1})$, for j=0,...,r-1, such that b's are permutations of a's and are arranged in a non increasing order , i.e. $b(l_i+1)\geq...\geq b(l_{i+1})$. Such normalized sequence is referred to as **b-subsequence** in [GLW] and [GLW1].

For example, let 3543433342 be an a-sequence, then the corresponding b-sequence will be 5443333432 (3543433342 is partitionable as 3543433 34 2, therefore the b-subsequence for 3543433 is 5443333, the b-subsequence for 34 is 43 and the b-subsequence for 2 is 2, which gives the above b-sequence 5443333432 when all b-subsequences are concatenated).

Finally, a brief definition of the terminology used in all remaining chapters of this thesis is given as follows:

- **Generating** sequences of n nodes of order m means producing all feasible sequences for that specific order (without necessarily **listing** them i.e. without **ouputting** every sequence). It is of some use in theoretical computing when this generation is done in a lexicographic order as a list of all shapes of trees of a given type might be used to search for a counter-example to some conjecture, or to test or analyze an algorithm for its correctness or computational complexity. Also the order is relatively "natural" and easier to follow.
- The **output time** is defined as the time it takes to output either a sequence or a result on screen or in an array (storage).
- The **delay** between two consecutive sequences is defined as the time required to

generate a new sequence from the previous existing one, and delay is **constant** if this time is constant (exclusive of output time) in the worst case.

- The **average delay** is defined as the ratio of the total time to generate all sequences to the total number of sequences generated.
- An algorithm has **constant average delay** if the ratio is less than a constant for any n, again, exclusive of output time.
- The B-Tree **coding** is defined as finding the sequence representation of a B-Tree given its Parent-Child representation.
- The B-Tree **decoding** is defined as the opposite of the B-Tree coding, namely finding the corresponding Parent-Child representation of a B-Tree of a B-Tree given its sequence representation.
- A **lexicographic order** is defined as follows: Let A and B be two sequences such that $A=(a_1, a_2, ...,a_n)$ and $B=(b_1, b_2, ...,b_n)$. We say that A precedes B in a lexicographic order if and only if for some $j \geq 1$, $a_i=b_i$ when $i<j$, and $a_j< b_j$.

• Some B-Trees applications.

B-Trees play a fundamental role in database applications. They allow for a fast retrieval of data and are the basis for the virtual sequential access method (VSAM) used in databases (for the sake of information, Hash trees are an alternative to B-Trees for the organization of large files of data and are discussed and compared by Bell and Deen[1984]). In order to appreciate the storing/retrieving capabilities of a B-Tree, it would be helpful to give an idea of the number of nodes, as well as the height of a B-Tree, for certain numbers.

The number of nodes and elements at a level k in a B-Tree of order m are in the following ranges:

$$2(m+1)^{k-2} \leq \text{Number of nodes} \leq 2m+1)^{k-1}$$

$$2m(m+1)^{k-2} \leq \text{Number of elements} \leq 2m(2m+1)^{k-1}$$

A numerical example of the number of elements and nodes that can be accommodated in a B-Tree of order m=100 and k = 4 is given below:

$$20\,402 \leq \text{Number of nodes} \leq 8\,120\,601$$

$$2\,040\,200 \leq \text{Number of elements} \leq 1\,624\,000\,000$$

For the sake of information, [MAW] claims that the real use of B-Trees lies in database systems, where the tree is kept on a physical disk instead of main memory. Accessing a disk is typically several orders of magnitude slower than any main memory operation. If a B-Tree of order m is used, then the number of disk accesses is $O(\log_m n)$. Although each disk access carries the overhead of O(log m) to determine the direction to branch, the time to perform this computation is typically much smaller than the time needed to read a block of memory and can thus be considered inconsequential (as long as m is chosen reasonably). Even if updates are performed and O(m) computing time is required at each node, this too is generally not significant. The value of m is then chosen to be the largest value that still allows an interior node to fit into one disk block. It is typically in the range of $32 \leq m \leq 256$.

The maximum number of elements that are stored in a leaf is chosen so that if the leaf is full, the leaf fits in one block. This means that a record can always be found in very few

disk accesses since a typical B-Tree will have a depth of only 2 or 3, and the root can be kept in main memory.

1.3 Research Problems

Gupta, Lee and Wong, in their articles on 2-3 trees and B-trees ([GLW] and [GLW1]), presented two algorithms for 2-3 trees and B-trees. They left as an open problem whether algorithms existed that could generate B-Trees and 2-3 trees in a lexicographic order, and whether it was possible to generate them in constant average delay, exclusive of the output. Studying the existing algorithms as well as the open problems related to them, it was decided to direct this thesis research in that specific path.

While working on the topic, we came across many related topics. We decided to extend the subject to coding and decoding of B-Trees. As we worked on sequential algorithms, we found out that some of them were good candidates for parallel implementation (which will be the topic of a separate paper to be published in mid 1996).

We were less fortunate when it came to parallel coding of B-Trees, which was left as an open problem.

1.4 Contributions

The contributions of this thesis could be summarized as follows:

1. A new algorithm, G23T, which generates 2-3 trees in a lexicographic order with constant average delay, thus solving the open problem raised in [GLW].
2. A new algorithm, G1BT, which generates B-Trees in a lexicographic order with

constant average delay.

3. A proof that B-Tree generation in [GLW1] is lexicographically ordered (if a suitable notation, that we introduce in Chapter 3 is used) thus solving the open problem raised in [GLW1].
4. A proof that both algorithms, G23T and G1BT, as well as the algorithm in [GLW1] have a constant average delay property, thus solving the open problem raised in [GLW1].
5. Two algorithms, DECOD1 (based on a B-Tree sequence) and DECOD2, (based on the number of leaves) which create a parent-child representation of a B-Tree given its sequence representation (B-Trees decoding).
6. An algorithm, COD1, which, given the parent-child representation of a B-Tree, finds its corresponding sequence representation (B-Trees coding).

Items 3 and 4 were accomplished jointly with Professor Ivan Stojmenovic (University of Ottawa) and published in the Information Processing Letters. All the remaining items have been completed by the candidate.

Organization.

The remainder of this thesis is organized as follows:

- Chapter II defines a lexicographic order and presents a new and more efficient algorithm than [GLW] for generating all 2-3 tree sequences in our defined lexicographic order in constant average delay time (thus solving the open problem in [GLW]).

- Chapter III introduces [GLW1], shows that B-Trees generation in [GLW1] is indeed lexicographic if a suitable notation is used and presents a modified algorithm for lexicographic B-Tree generation.
- Chapter IV is the proof that [GLW] and [GLW1] had constant average delay properties.
- Chapter V presents new sequential algorithms for coding and decoding of B-Trees as well as a parallel algorithm for decoding of B-Trees.
- A conclusion and open problems section
- References used in this research
- Relevant algorithms for this thesis

Chapter II

Generation of 2-3 trees

2.1 Generation of 2-3 trees by Gupta, Lee and Wong [GLW]

Gupta, Lee and Wong described algorithms for generating 2-3 trees and B-trees and left as open problems whether algorithms exist that generate them in lexicographic order, and whether it is possible to generate 2-3 trees [GLW] or B-Trees [GLW1] in constant average delay, exclusive of the output. In this chapter, [GLW] is presented, the lexicographic order of the 2-3 trees is defined and the algorithm for generating them in a constant average delay time is presented, thus solving the open problems that were raised in [GLW].

The representation of a list of items in a 2-3 tree can lead to more than one tree representation as illustrated in Figure 6 and will help explain the concept. Figure 6 shows the different 2-3 trees that can be generated with 8 leaves (i.e.: 7 keys, the content of the nodes is irrelevant in this study) and every tree has its own sequence "name". The sequence naming is based on the number of children for internal nodes and follows a simple pattern. As in [GLW1], we shall represent a B-Tree (here a B-Tree of order 3) with n leaves and hence n-1 keys) by a sequence $a_1a_2...a_k$ that is obtained as explained in the first chapter (Chapter 1).

Figure 6 shows an example of sequence naming (i.e.: naming consists in starting from the

bottom left to the right, counting the number of children and repeating the same pattern at the next level up to the root). Figure 6.a has sequence 3323, Figure 6.b has sequence 2333, Figure 6.c has sequence 3233 and Figure 6.d has sequence 2222222).

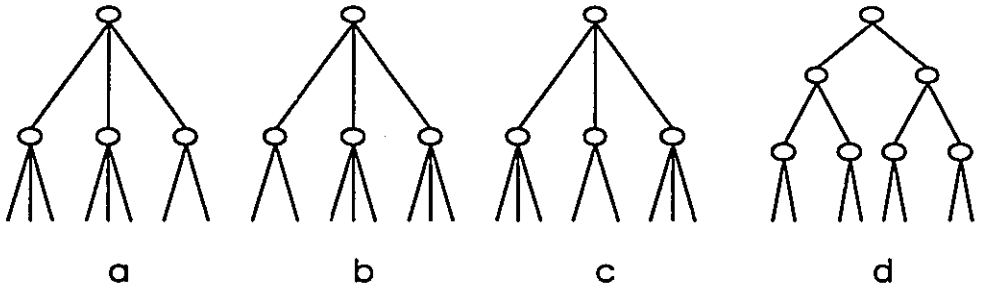

Figure 6

All internal nodes contain one or two keys depending on whether they have 2 or 3 children respectively. Since we are only interested in the node structure of the tree (i.e.: the data giving node indices or pointers to parent and children nodes) the key values as well as the nodes values are immaterial.

The algorithm presented in this chapter, algorithm G23T, is new, different from [GLW] and will, given a number n of leaves, generate all possible 2-3 tree sequences having n as their number of leaves. It is also more efficient than the one presented in [GLW].

We shall first introduce the [GLW] 2-3 tree generating algorithm before introducing ours and comparing it to [GLW].

[GLW] use ranking (given a sequence, find the position of this sequence in the lexicographic order) and unranking (given the position, find the corresponding sequence) procedures to generate 2-3 trees. First a search graph is constructed with the help of

which ranking and unranking can be done in O(n) time. This search graph is defined as a weighted directed acyclic graph with nodes identified as labels and it is constructed in two passes, the forward and the backward pass. At the end of the forward pass, we have a graph with all its nodes and edges and weights on the edges. How the search graph is obtained is shown in the next paragraph.

The algorithm for ranking, unranking and generating 2-3 trees is carried out in the following two major steps:

- **Step 1 Find the Search Graph**

A search graph is defined as a graph used to either rank or unrank 2-3 trees with n keys in O(n) time. It is used in concordance with the different number of permutations with repetitions for each set of sequences (shown in Figure 7 for a number of nodes equal to 20) and is done using 2 passes: a forward pass and a backward pass.

Partition	size of partition	Number of permutations with repetitions
2222222222	10	10!/10!=1
22222233	9	9!/2!7!=36
22223333	8	8!/4!4!=70
2333333	7	7!/6!=7

Figure 7

The forward pass construction, shown in Figure 8, consists of putting the size of the next partition in the node, and the number of permutations for that node in the edge. For example, let n=20 and let Figure 7 be the corresponding partition table. For n=20, according to Figure 7, there exists one sequence of ten 2's (total being 20), 36

permutations of size 9, and so on (the weight of the edges that originate from node 20 are given by Figure 7 and correspond to the number of permutations with repetitions). We recursively carry on with node 10. It has only 1 (one) permutation of size 5 (i.e. 22222) and 6 of size 4 (3322, 3232, 3223, 2323, 2233, 2332). Node 5 has 2 permutations of size 2 (namely 2 3, 3 2). Finally, Node 2 has only 1 permutation of size 1.

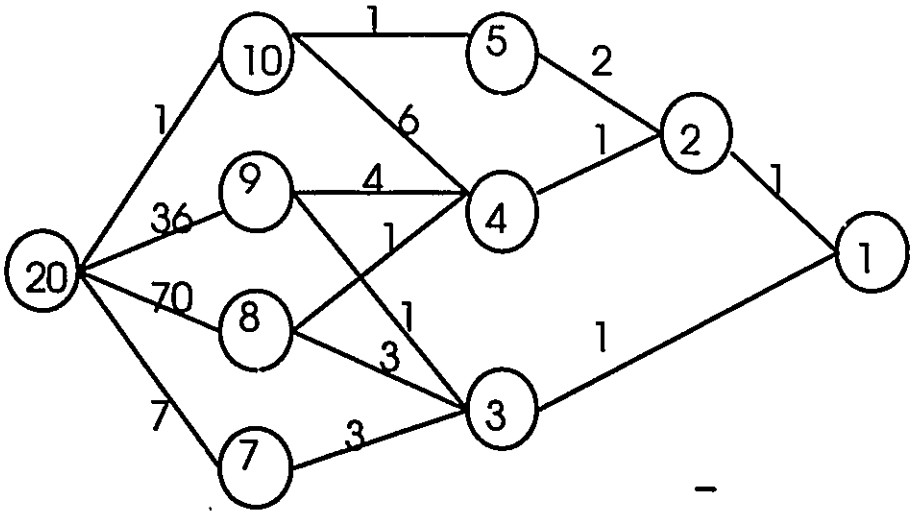

Figure 8

The number of nodes is bounded by $1+\lfloor(n+1)/2\rfloor$ (i.e. O(n)), the number of edges by $O(n^2)$ and the out-degree of node p by $\lfloor p/2\rfloor - \lceil p/3\rceil + 1$. The edge (p,q) with weight w(p,q) denotes that every 2-3 tree with q leaves and height h can be extended to w(p,q) distinct 2-3 trees with p leaves and height (h+1). If we have a table of the factorial function for arguments $\{1,2,\ldots, \lfloor(n+1)/2\rfloor\}$ then clearly the time complexity for the forward pass is $O(n^2)$.

The backward pass is illustrated in the following figure:

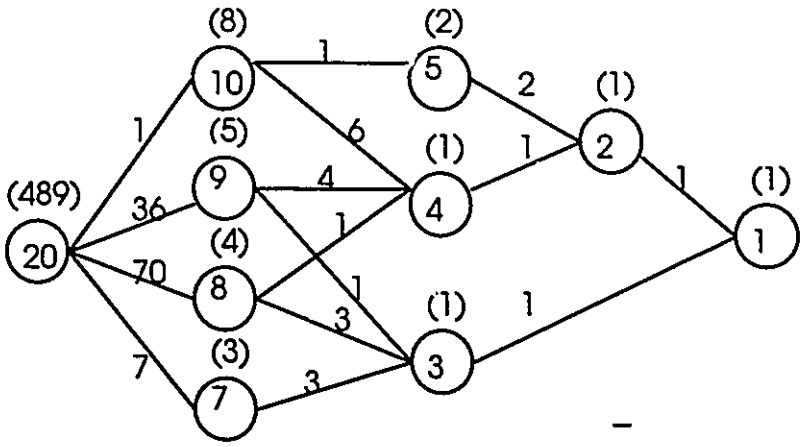

Figure 9

The backward pass, shown in Figure 9, consists in starting at node 1 and in a backward manner successively attaching weights to the nodes of the graph using the following formula to get to the next node:

$$W(1) = 1, \quad W(p) = \Sigma\, W(q) * w(p,q)$$

(where W(p) is the total number of distinct 2-3 trees with p leaves, (p,q) is the edge between node p and node q and w(p,q) the weight associated with that edge).

To illustrate the previous formula, let us look at the first path (20 --> 10 --> 5 --> 1), starting from the last node (node 1):

- (1) * 1 = 1 (result at node 2);
- (1) * 2 = 2 (result at node 5);
- [(2) * 1](from node 5) + [(1) * 6](from node 4)=(8) at node 10 and so on...;
- the backward pass involves looking at each edge once, therefore the time complexity is $O(n^2)$ (as there are n nodes looking at a maximum of n-1 edges); and therefore

- the whole graph can be constructed in $O(n^2)$ and needs $O(n^2)$ storage.

- **Step 2 Ranking and unranking**

[GLW] use ranking and unranking procedures. Those procedures are mentioned in this chapter for the sole purpose of showing that our algorithm does not require such procedures and is therefore faster.

The ranking procedure computes the rank (or position) of a given 2-3 tree represented as an n-feasible sequence in a level by level manner. As generation of 2-3 trees is done using unranking, the ranking procedure is not relevant in this thesis and therefore will not be presented.

The unranking procedure computes an n-feasible sequence given the rank of the n-feasible sequence. It computes the number q of internal nodes at the lowest level and then the permutation of the number of sons of these q internal nodes. The process is repeated at every level.

Initially, there are p = n + 1 leaves and q is obtained as follows:

$$\sum_{q'=(p/3)}^{q} w(p,q') * W(q') \leq \text{rank}.$$

(i.e.: given the rank, find a value q such that the previous formula applies).

After q has been determined, the number r of 2's and the rank u of the permutation of the number of sons of these q internal nodes are computed. With the values q, r and u the function unrankperm(q,r,u) is invoked to obtain the permutation of r 2's and (q-r) 3's whose rank is u (because we want to show that we are not using any unrankperm function, the unrankperm function referenced in [GLW] will not be defined or discussed in

this thesis).

The rank and the number p of leaves (which is set to be equal to q) are then updated. The process repeats until the root of the tree is reached; i.e.: p=1).

For example, to determine the 231st tree with 19 keys, it is necessary to first determine the number of nodes at the lowest internal level. Since $231 > w(20,7) * W(7) = 7 * 3 = 21$ and $231 \leq w(20,7) * W(7) + w(20,8) * W(8) = 21 + 280 = 301$ then that number must be 8. Also since $\lfloor (231-21)/w(20,8) \rfloor = \lfloor (210/4) \rfloor = 52$, then this corresponds to the 53rd permutation of 4 2's and 4 3's which happens to be 32332232.

In both algorithms (ranking and unranking), [GLW] use some values of n which are useless (i.e.: they do not appear in the search graph: node #6 does not appear in table 1 but is nevertheless used). This has the effect of "eating up" unnecessary time in the algorithm. Once the search graph is constructed, [GLW] claim that ranking and unranking can be done in O(n) time. Since decoding can be done in O(n) time and the output size is also O(n), [GLW] also claim time and output size to be optimal. Unfortunately, [GLW] do not take into account that the rank of a 2-3 tree can be a very large integer since the number of 2-3 trees is exponential in n. The representation of the rank of a tree may require O(n) space and O(n) time for any arithmetic with it, thus causing an $O(n^2)$ delay in producing 2-3 trees rather than linear as claimed.

In this presentation, we generate all 2-3 trees by simply "unranking" (not in the same manner as [GLW]) 1st, 2nd, 3rd,... kth 2-3 trees until all of them are encountered, thus bypassing the inconvenience of creating a search graph. This is done with a time

complexity of O(n) time between any two 2-3 trees. Since the algorithm generates all 2-3 trees while dealing with smaller integers and uses a worst case linear delay per 2-3 tree, it can be proved that the generation of 2-3 trees is done in constant expected time. [GLW] not only claimed that such generation is proportional to the size of the output, but also left a constant delay average as an open problem.

2.2 Sequence representation and ordering of 2-3 trees.

As mentioned in chapter I, the third property applying to 2-3 trees said that every a-sequence should be partitionable into levels. The levels are defined as

- $l_0 = 0$ represents the leaf level (or zero level).
- l_1 represents the first level (starting from the bottom) such that $a_1 + a_2 + ... + a_{l_1} = n$;
- $l_2,...,l_r$ are determined easily from property 3.

The value for n is derived from the sequence using a backtracking method as follows:

- start with the last element of the sequence, backtrack a number of times equal to its value and add up the encountered numbers.
- repeat until there are no more numbers to backtrack
- the last number is the value of n

For example, let 233322223323 be an a-sequence. The last number is 3.

Backtracking and accumulating the 3 numbers before it, 233322223323 adds up to 8.

Backtracking and accumulating the 8 numbers before them 233322223323 adds up to 19.

As there are no more elements to backtrack, n = 19.

The graphic representation of the sequence 233322223323 is shown in Figure 10.

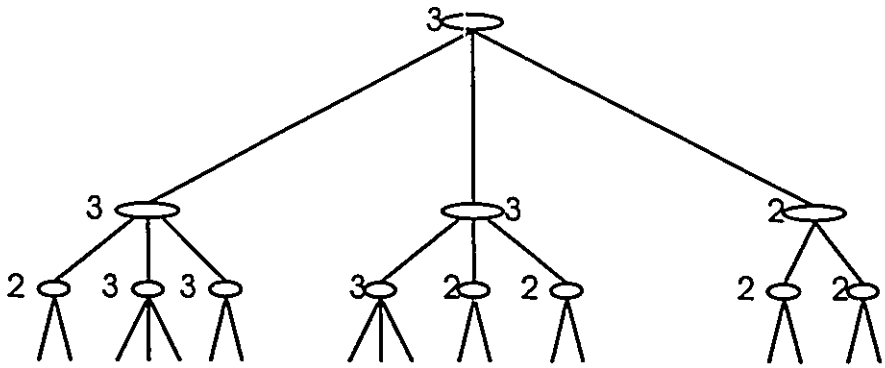

Figure 10

The level sequence for Figure 10 is 0 8 11 12: ($l_0 = 0$, $l_1 = 8$ the number of elements at the first level) and $l_2 = 11$ (the sum of the number of elements at the first and second level) and $l_3 = 12$ (the sum of the number of elements at all levels).

[GLW] generated 2-3 trees using unranking. We noticed that this generation of 2-3 trees could be a generation in lexicographic order if a suitable notation was used. In this chapter, we introduce this notation and generate 2-3 trees in that defined lexicographic order. In the literature, a **lexicographic order** is defined as follows:

Let A and B be two sequences such that $A=(a_1, a_2, ...,a_n)$ and $B=(b_1, b_2, ...,b_n)$. We say that A precedes B in a lexicographic order if and only if for some $j \geq 1$, $a_i=b_i$ when $i<j$, and $a_j< b_j$.

We shall introduce our 2-3 tree lexicographic order notation as being as follows:

l_1 (sequence at level 1) l_2 (sequence at level 2) and so on .

For the above sample sequence, in Figure 10 the lexicographic sequence would be as shown in Figure 11.

8	2-3-3-3-2-2-2-2	11	3-3-2	12	3
l_1	sequence at level 1	l_2	sequence at level 2	l_3	sequence at level 3

Figure 11

thus giving a lexicographic sequence of

8 23332222 11 332 12 3

(which when compared to 8 23232222 11 332 12 3 is lexicographically later because the third character, bolded for clarity, is alphabetically (or lexicographically) larger).

For a complete view of a lexicographic order, Figure 6 which represents all the possible 2-3 trees that can be generated will therefore be lexicographically ordered as follows:

8 2222 10 22 11 2 ($l_1 = 8, l_2 = 10, l_3 = 11$)

8 233 9 3 ($l_1 = 8, l_2 = 9$)

8 323 9 3 ($l_1 = 8, l_2 = 9$)

8 332 9 3 ($l_1 = 8, l_2 = 9$)

Note: it is important to remind that the previous l-sequence, and for that matter the whole sequence, consist of integers (i.e.: array of integers) and not alpha numeric characters (strings). In case characters are used, a problem occurs as 11 is lexicographically before 2, therefore our lexicographic order would apply only to l-sequences with any $l_i < 10$.

2.3 The sequential algorithm

The sequential algorithm presented in this section uses a backtracking search strategy. The backtracking search strategy is a systematic strategy that is used to enhance the

efficiency of exhaustive search, since the number of candidates for a solution is often exponential in input size [RND]. The algorithm finds in a backward run the **turning point** (which is an element with the greatest possible index that can be increased), while a **forward** run updates elements from the turning point to the last element to produce the next 2-3 tree.

The lexicographically first sequence for any level i of size m_i is defined by 2^p3^q (with $p+q=m_i$ and $q=l_{i-1} - l_{i-2} - 2m_i$ and where $2^p = 2...2$ (p times)) and is constructed by procedure **lfirst()**, which finds the largest number of 2's that a number can be decomposed into and which is defined as follows:

```
Procedure lfirst(i);
{       m_i ←l_i - l_i-1 ;
        q_i ←(l_i-1 - l_i-2 ) - [2 * m_i ];
        p_i ←m_i - q_i ;
        for k←(l_i-1 + 1) to (l_i-1 + p_i) do a_k ←2;
        for k←l_i-1 + p_i + 1 to l_i do    a_k ←3;          }
```

A procedure called **lexfirst()** is then used to construct the first lexicographic sequence at level i after the sequence at the previous level has been updated to the present sequence.

```
Procedure lexfirst(i);
repeat
        i← i+1;                 { Go one level up                                  }
        m_i ←l_j - l_i-1 ;
        l_i ← ⌈ (m_i )/3 ⌉;     { Take upper limit of result                       }
        l_i ← l_i + l_i-1       { Get value at new level                           }
        lfirst(i);              { Find lexicographically first sequence at this level }
until (l_i - l_i-1 = 1);        { Until the root is reached                        }
```

The following Pascal-like procedures are used in the program that generates all 2-3 trees in a lexicographic order.

(The entire program can be found in Appendix A and is called Program G23T.)

```
-- This procedure will create the very first of all sequences
Procedure start;
        {Init;          { Initializes all arrays to 0 }
        l[0]← 0;
        l[1]← ⌈n/3⌉;    { to get the smallest partition number }
        l[-1]← -n;
        i← 1;
        lfirst(i);
        lexfirst(i); }


-- This procedure creates a sequence when a turning point has been found}
procedure one;
        {tp← true;
        a[j]← 3;
        for l1← j+1 to z+j+1 do a[l1]← 2;      { z is obtained as shown in the   }
        for l1← z+j+2 to l[i] do a[l1]← 3;     { main program in the sequel}
        lexfirst(i);}

-- This procedure creates a sequence in case no turning point was found

procedure two;
        if m[i] < trunc( ( l[i-1] - l[i-2] ) / 2) then
                {
                l[i]← l[i] + 1;
                lfirst(i);
                lexfirst(i);
                tp← true;}
```

Finally, the main program that incorporates all of the previous procedures, includes a procedure called newsequence, that will simply output the first lexicographically ordered sequence (the first time the lexfirst procedure is entered) or any new updated sequence (that is generated later) is defined as follows.

```
-- This program generates all 2-3 trees given a number n of leaves

{ MAIN PROGRAM : G23T}
{
        get(n);
        start;
        repeat
        newsequence;                    -- Outputs sequence
        writeln;                        -- Now to create next 2-3 tree
        tp←false;
                repeat
                        i←i-1;          y←0;            z←0;            j←l[i];
                        while ((a[j]=2) and (j > l[i-1])) do { z←z+1;    j←j-1; }
                        while ((a[j]=3) and (j > l[i-1])) do { y←y+1;    j←j-1; }
                        if (j>l[i-1]) then one      -- Turning point is found
                            else two                -- No turning point is found
                until (tp or (i=1));
        until (tp = false);}
```

The previous main program generates all 2-3 trees with a constant average delay. The two while loops are crucial to the algorithm as they determine whether a turning point was found or not. The logic behind them is to determine whether a sub-sequence made of two 3's (equalling 6) can be expanded to the subsequence 222 in order to get the next lexicographic sequence. This is performed at every level, whereof the utility of the variable j acting as a sentinel and ensuring that this is done level by level. The proof of termination is trivial and can be done by using the formula $\sum_{i=a}^{b} i$ (where a = z (or y depending on which loop is used) and b = l[i] with i being the present level) as being the invariant and proof of that formula by induction.

Note : Formal proof of constant average delay is given in Chapter IV.

2.4 Comparison of algorithms

In order to summarize the two algorithms, it is necessary to compare the different steps

that each of them contains.

- **The GLW algorithm :**

As was mentioned above, the [GLW] algorithm contains different steps:

1. Create a search graph (for both forward and backward pass). Creating such a graph took $O(n^2)$ time and $O(n^2)$ storage space.
2. Use of ranking and unranking in order to generate the different sequences. Such ranking and unranking took only O(n) time (decoding and encoding took O(n) as well).
3. The output size was O(n).
4. The delay between 2 sequences was $O(n^2)$ rather than linear as claimed.

- **The G23T algorithm:**

5. It was not necessary to create a search graph thus saving time and storage space (O(n) for both of them).
6. Neither ranking nor unranking in order to get the sequences were used.
7. The average delay between sequences was constant (as will be shown in chapter IV).

As shown above, the algorithm presented here is far more efficient than the one presented in [GLW]. The next step would be to extend the present study to the development of an algorithm that will consider the more general generation of B-Trees with a constant average delay .

Chapter III

Lexicographic generation of B-Trees of order m

3.1 Introduction to [GLW1]

In [GLW1], the problem of generating B-trees is considered. The authors presented a generating algorithm based on a backtrack search and obtained an algorithm that produced B-trees in time proportional to the output size. The same issues with lexicographic order and constant average delay (and thus efficiency) as in [GLW] remained unsolved in [GLW1].

In this chapter, as first contribution, we modify [GLW1] and create a new, more optimal algorithm for the lexicographic generation of the B-Trees

As a second contribution, we introduce a sequence notation of B-trees under which the order in [GLW, GLW1] is lexicographic and we prove later in chapter IV that the algorithm in [GLW1] has constant average delay property.

As 2-3 trees are B-Trees of order 3, [GLW1] also applies to 2-3 trees. The algorithm in [GLW] and the one presented in chapter 2 (G23T) are both different from the one in [GLW1]. The question many people might ask when first hearing about the generation of B-Trees would probably be: what is the purpose of such a generation? The answer is simple in the sense that in theoretical computing, a list of all shapes of trees of a given type

might be used to search for a counter-example to some conjecture, or to test or analyze an algorithm for its correctness or computational complexity. This makes the generation of such lists of all shapes of trees of some specified kind more interesting. Also the order is relatively "natural" and easier to follow.

Typically, a one to one correspondence is established between a class of trees and certain integer sequences. It is then shown how these sequences can be generated in order (usually lexicographic) and how the position of a given sequence in this ordering can be determined (ranking) and vice-versa (unranking).

There are some desirable properties of algorithms that generate any kind of combinatorial objects. Those properties are:

- optimal time
- not using large numbers (i.e. 2^n)
- lexicographic order (because the order is relatively "natural" and easier to follow)
- constant average delay between any two consecutive combinatorial objects.
- constant delay between any two consecutive combinatorial objects (such property implies that there is no loop in the program that generates the next B-tree from the current one). If this property is satisfied, then obviously so is the constant average delay property; and
- minimal changes between consecutive sequences.

Two properties are used in our generating algorithm: lexicographic order and constant average delay between any two consecutive sequences.

The first property is desirable because the order is natural and the control over the order of sequences is easy to follow. The second is also desirable because it is possible to measure the performance of an algorithm either with output or without producing output. But as outputting of data applies to any algorithm, constant average delay between sequence generation is desirable (in fact, since the output is normally fixed, the rest of the algorithm is the real cost).

Also if P(n) is the number of combinatorial objects, each of them having the size O(n), then the total output size is O(nP(n)). However, in some applications, the objects to be generated do not need to be output and are merely used as the source of information for other procedures that work on combinatorial objects and check some criteria (which may be verified without looking at the whole new object). Such optimal algorithms in this sense may work in O(P(n)) time (constant time per object).

In the sequel, we present the algorithm from [GLW1] for generating B-Trees and show that it indeed generates B-Trees in a lexicographic order if a suitable sequence representation is used. We also present a new algorithm based on [GLW1] to generate all B-Trees in a lexicographic order. As partitions and compositions of integers are frequently mentioned, they are defined and explained in the following section.

3.2 Partitions and compositions of integer

Any integer n can be represented as the following sum of the series:

$$n = a_1 + a_2 + ... + a_s$$

In this thesis, **parts** are defined as the components of that sum of series and the number of parts as the largest index in that sum. For example, if we let 5 = 1+1+1+2 , then the parts are 1, 1, 1, 2 and the number of parts is 4.

Note: This definition of **parts** will remain the same throughout this thesis.

Such a representation is called a **partition,** if the order of the a_i is of no importance (i.e. two partitions of an integer are different if they differ with respect to the a_i they contain) and a **composition** if the order of the a_i is important. Therefore, the divisions of a positive integer are of two types:

- **Partitions:** which can be defined as *unordered* divisions; and
- **Compositions:** which can be defined as *ordered* divisions.

Those partitions and compositions can be also referred to as either *restricted* or *unrestricted.* The main difference between those two terms is as follows:
unrestricted (as opposed to restricted) refers to the case where there is no particular limit (or restriction) on the number of parts or size of each part that a series can contain.

Two kinds of restrictions might be observed in the literature:

- restrictions on the number of parts (implying no restriction on the size of the part, for instance 5 could be partitioned as 14, 23, 41 or 32 if number of parts is restricted to 2 and 311, 221 if the number of parts is restricted to 3); and

- restrictions on the size of each part (in the case of B-Trees for example, where the size of the part is limited by the order of the B-Tree itself. For instance, 5 can be partitioned as 23 or 32 if the order is 3 but cannot be partitioned as 14 or 41 for the same order as neither 1 nor 4 are valid number within the specified order).

For example, if we let n = 5, therefore we can say the following about 5:

a) 5 has 7 **unrestricted partitions** on both the number of parts and the size of each part which are:

5	4+1	3+2	3+1+1	2+2+1	2+1+1+1	1+1+1+1+1

for simplicity's sake, they will be noted as

5	41	32	311	221	2111	11111

b) 5 has 16 **unrestricted compositions** on both the number of parts and the size of each part which are:

5
41 14
32 23
311 131 311
221 212 122
2111 1211 1121 1112
11111

Note: a partition $n=a_1 + a_2 +...+ a_m$ of n can be defined such that $a_1 \leq a_2 \leq...\leq a_m$ if an ascending order is preferred.

Given any integer n, there are 2^{n-1} unrestricted compositions.

As an illustration, 4 has $2^{4-1} = 2^3 = 8$ unrestricted compositions which are:

4	13	31	22	112	121	211	1111

Finding the exact number of unrestricted partitions given an integer n is not known and remains an open problem. However and for the sake of information, a recursive formula RP(n,m) is proposed in [Z] that does find the exact number of unrestricted partitions of a number n using at most m parts as follows:

RP(n,m) = RP(n,m-1) + RP(n-m, min(n-m,m))

with RP(n,1)=1, RP(n,0)=0, RP(0,0)=1 and RP(n,m)=0 if m > n.

The following table shows the number of unrestricted partitions for a specific n based on the above RP(n,m) formula.

n	P(n)
1	1
2	2
3	3
4	5
5	7
6	11
7	15
8	22
9	30
10	42
15	176
150	40 853 235 548
240	105 882 249 516 398

Table 1

- **Doubly restricted partitions.**

The same two kinds of restrictions that applied to restricted partitions apply to **doubly restricted partitions** with the difference that there exist two boundaries, P1 and P2, instead of one (P1 and P2 being integer numbers). The two types of doubly restricted partitions are:

- restrictions on the *number* of parts, where the number of parts is between P1 and P2, (implying no restriction on the size of the part, for instance 5 could be partitioned as 14, 113, 122 or 23, but not as 11111 if number of parts is restricted between P1=2 and P2=3); and
- restrictions on the *size* of each part (for example in the case of B-Trees, where the size of the part is limited by the order of the B-Tree itself. For instance, 5 can be partitioned as 23 or 32 if the order is 3 but cannot be 14 or 41 for the same order as the size of the part is restricted between P1=2 and P2=3).

Those doubly restricted partitions apply to B-Trees as a B-Tree sequence contains parts whose sizes are bound by the order of the tree. They will be discussed in the next sections.

3.3 Lexicographic generation of partitions with s parts.

In this section, we present the **make_partitions(n,s,P1,P2)** algorithm. This algorithm generates all partitions of an integer n with exactly s parts, each part having a size between two integers P1 and P2.

Two arrays, p and e, are used to code the sequences (p[i] represents a number, e[i] represents its multiplicity);

therefore sequence 3332223 or $3^3 2^3 3^1$ is coded as

p[1]=3 and e[1]=3, p[2]=2 and e[2]=3, p[3]=3 and e[3]=1

i.e. all lexicographic sequences will be represented as

$$\mathbf{p[1]^{e[1]}\ p[2]^{e[2]}\ldots\ p[x]^{e[x]}}$$

The size of the parts has been defined as being between two numbers P1 and P2. How P1 and P2 are obtained is irrelevant at this stage (it will be explained in section 3.5.1).

We define a procedure **lexmin** which is used the first time to find the smallest lexicographic sequence given **n** and **s**, as defined previously, and will be used the subsequent times to update arrays p and e when called by procedure make_partitions (defined in the next paragraph). A new parameter, parameter **u**, is introduced in procedure lexmin. This parameter represents the index for both arrays (i.e. it is the starting position in arrays p and e) as [GLW1] use a backward manner to generate their sequences by updating arrays p and e starting from a certain position. How lexicographic order is obtained is shown in section 3.4.2.

The lexmin procedure is as follows:

```
procedure lexmin(n,s,u:integer; p,e:array);
 {if n>=s then
    {r←n mod s;       q←n div s;
      if r <> 0 then    {u←u+1;      p[u]←q+1;      e[u]←r; }
      if (r <> s) and (q<>0) then     {u←u+1;      p[u]←q;      e[u]←s-r;}
```

The main idea behind procedure lexmin is to partition n into s parts. The division giving q (where q = n div s) means that there are s parts of size q. The remainder giving r (where r=n mod s) means that once all the parts have been assigned a size of q, there remains r "items" to be placed in those s parts. By increasing q by 1 (p[u]=q+1) and this for the r parts (e[u]=r), we ensure that all the r items have been distributed. The position u is then

increased by 1, and by placing the remaining (s-r) items of value q, we ensure that all s parts have been assigned a value.

The following provides a trace of the lexmin procedure. Let n=31, s=6 and u is initialized to 0 at the beginning of the program.

```
{if n>=s then                                   -- (True)
    {r←n mod s;        q←n div s;               -- r = 1;  q = 5
      if r <> 0 then                            -- (True)
              {u←u+1;        p[u]←q+1;          e[u]←r; }
              -- u = 1 ;      p[1] = 6;          e[1] = 1
      if (r <> s) and (q<>0) then               -- (True)
              {u←u+1;        p[u]←q;            e[u]←s-r;}
              -- u = 2 ;      p[2] = 5;          e[2] = 5
The sequence is therefore 655555
```

A more interesting example is with n=33 and s=6. Each part is assigned a value of 5 (for a total of 30 and a remainder of 3) having a sequence of 555555. The three remaining items are distributed one by one over each part starting from the left until all items are exhausted thus giving a sequence of (5+1)(5+1)(5+1)555 which is 666555.

The functionality of procedure lexmin is two-fold:

1. it generates the first lexicographically minimum normalized partition (or sequence).
2. it provides an update of the two arrays p and q, using different parameter values (n,s,u) each time (those parameters are updated by the make_partitions procedure, presented in the sequel).

The update is used to generate the next smallest lexicographic ordered partition and procedure lexmin(n,s,u,p,e) is called from within procedure make_partitions(n,s,P1,P2), which will loop until all lexicographic sequences are exhausted.

The following procedure, procedure make_partitions(n,s,P1,P2), is used to create all

partitions of a number n with s parts, each part size being between P1 and P2 and was largely inspired from [GLW1]. In order to understand it, a high level explanation would be useful: the lexmin procedure ensures generation of the partition whose corresponding normalized sequence is lexicographically minimum (denoted lexmin(n,s) for reference).

If m is the order then clearly, the size of each part belongs to a set M, where M = {$\lceil m/2 \rceil$, $\lceil m/2 \rceil + 1$, ..., m}. The next partition is obtained from the current one as follows: suppose the current partition is $p[1]^{e[1]}\ p[2]^{e[2]} \ldots\ p[t]^{e[t]}$ where $e[x] > 0$. The last element of the sequence, $p[t]^{e[t]}$, is of special interest as whether it is equal to $\lceil m/2 \rceil$ or not will determine which action is to be taken. In effect, the idea is to look at the last element of the sequence and determine whether it is possible to diminish its size by 1 and add 1 to one of the previous elements of the sequence. The different actions to be taken are shown in the sequel followed by an example.

If $p[t] = \lceil m/2 \rceil$ **then**

* **if** $e[t-1] > 1$ **then** replace $p[t-1]^{e[t-1]}p[t]^{e[t]}$ by $(p[t-1]+1)\text{Lexmin}(n_1, s_1)$,

where $n_1 = e[t-1]p[t-1] + e[t]p[t] - (p[t-1] + 1)$ and $s_1 = e[t-1] + e[t] - 1$,

* **else if** $e[t-1] = 1$ **then** replace $p[t-2]^{e[t-2]}\ p[t-1]^{e[t-1]}\ p[t]^{e[t]}$ by $(p[t-2] + 1)\text{Lexmin}(n_2, s_2)$,

where $n_2 = e[t-2]p[t-2] + e[t-1]p[t-1] + e[t]p[t] - (p[t-2]+1)$ and $s_2 = e[t-2] + e[t-1] + e[t] - 1$.

If $p[t] \neq \lceil m/2 \rceil$ **then**

* **if** $e[t] > 1$ **then** replace $p[t]^{e[t]}$ by $(p[t] + 1]\text{Lexmin}(n_3, s_3)$,

where $n_3 = e[t]p[t] - (p[t] + 1)$ and $s_3 = e[t] - 1$,

* **if** $e[t] = 1$ then replace $p[t-1]^{e[t-1]}\ p[t]^{e[t]}$ by $(p[t] + 1)\text{Lexmin}(n_4, s_4)$,

where $n_4 = e[t-1]p[t-1] + e[t]p[t] - (p[t-1] + 1)$ and $s_4 = e[t-1] + e[t] - 1$.

If the replacing sequence Lexmin(n',s') does not exist, i.e., x > e

or the sequence to be replaced does not exist, e.g., $p[t-1]^{e[t-1]}$ and e[t-1] = 0,

then the algorithm terminates.

For example, let n=31, s=6 and order m = 7 (i.e. the only allowed numbers ∈ M={4, 5, 6 and 7}).

We start with Lexmin(31,6) which is $65^5 = p[1]^{e[1]}\ p[2]^{e[2]}$.

Since p[2] <> 4 and e[2] > 1 we replace 5^5 by 6Lexmin(19,4) = 65^3 4.

Therefore the next partition after 65^5 is $6^2 5^3 4 = p[1]^{e[1]} p[2]^{e[2]} p[3]^{e[3]}$.

Since p[3] = 4 and e[2] > 1, we replace $5^3 4$ by 6Lexmin(13,3) =654^2 : the new partition is $6^2\,654^2 = 6^3 54^2$.

Since p[3] = 4 and e[2] = 1, we replace $6^3 54^2$ by 7Lexmin(24,5) = $75^4 4$.

Following the same reasoning, the next partitions are $75^4 4$, $765^2 4^2$, $76^2 4^3$, $7^2 54^3$.

When $7^2 54^3$ has been generated, the algorithm terminates because the next partition is 8Lexmin(23,5) = $7^2 54^3$ (but the number 8 does not belong to M={4,5,6,7} and is therefore not allowed).

Procedure make_partitions(n,s,P1,P2) computes the parameters n, s and u that lexmin uses to update the two arrays p and e. It is entirely based on the previous discussion on actions to be taken and is defined as follows:

Procedure make_partitions(n,s,P1,P2)

```
{n←t;
 while (p[1] <=min) do
  {bool←false;          OUTPUT sequence p;          counter←counter+1;
 if e[i]*low=number_nodes then terminate;
 if  (u=1) and (e[1]=1) then writeln('end of generation');
 if  p[1]=1 then writeln('end of generation');

 if p[u]=low then                                  -- low corresponds to ⌈m/2⌉
       {if e[u-1]>1 then
         {n←e[u-1]*p[u-1] + e[u]*p[u] - p[u-1] -1;   -- corresponds to n1 defined above
          s←e[u-1] + e[u] -1;                        -- corresponds to s1 defined above
         p[u-1]←p[u-1]+1;  e[u-1]←1;  u←u-1;         -- corresponds to values defined above
         lexmin(n,s,u,p,e);}

 else if u<>2 then
           {n←e[u-2]*p[u-2] + e[u-1]*p[u-1];         -- corresponds to n2 defined above
            n←n+e[u]*p[u]-p[u-2] -1;
            s←e[u-2] + e[u-1] + e[u] -1;                  -- corresponds to s2 defined
above
            u←u-2;  p[u]←p[u]+1;  e[u]←1;             -- corresponds to values defined above
           lexmin(n,s,u,p,e); }

       else if e[u]>1
       then
       {n←e[u]*p[u]-p[u]-1;                          -- corresponds to n3 defined above
        s←e[u]-1;                                    -- corresponds to s3 defined above
       p[u]←p[u]+1;  e[u]←1;                         -- corresponds to values defined above
       lexmin(n,s,u,p,e);}

       else
       {n←e[u-1]*p[u-1] + e[u]*p[u] - p[u-1] -1;     -- corresponds to n4 defined above
        s←e[u-1] + e[u] -1;                          -- corresponds to s4 defined above
        p[u-1]←p[u-1]+1;                             -- corresponds to values defined above
        e[u-1]←1;  u←u-1;
        lexmin(n,s,u,p,e);}
```

The make_partitions(n,s,P1,P2) algorithm has been slightly modified from the original in [GLW1]. In fact, [GLW1] use concatenation of strings to output a new sequence while we use the lexmin procedure to do so (working with strings instead of numbers is not optimal, as strings of numbers have to be converted to numbers and vice versa, but this is

a minor detail and it is mentioned for clarification purposes only). Another reason why numbers are used instead of strings is that as mentioned in chapter 2, our lexicographic order does not apply to strings of characters.

3.3.1 Generation of permutations with repetitions.

As seen in the previous section, the make_partitions procedure generated all the partitions, with exactly n parts, the size of each part being between P1 and P2. Those partitions can be changed into compositions (or combinations) by sending each partition as an argument to a procedure that would find all the possible permutations for that partition. Because the number of parts is fixed, those partitions are most likely to be with repetitions (one element is repeated at least once, i.e.: 56666, 6 is repeated 4 times). In their algorithm, [GLW1] do not mention anything about a procedure to generate compositions. We decided to provide one, called **permute** to further improve the algorithm. To make the procedure more general, a partition can be thought of as a sequence of numbers.

The **permute** procedure (*I. Stojmenovic*, unpublished) will always take an ascending sorted sequence z as an argument, and will output all the *distinct* permutations with *repetitions* for that sequence (i.e. no permutation will be repeated). It is important to remember that this procedure generates **distinct** permutations with repetitions (as the permute procedure might look quite unusual from existing permutation procedures). In the sequel, a description of that procedure followed by an example (let the input sequence be 1123) is given. The main idea behind procedure permute is to take a sorted ascending sequence (with or without repetitions) as input and starting from the last element in that sequence, find the first element that is smaller than it and mark it as the pivot. From the

end of the sequence again, find the largest element between the pivot and the last element and swap the two numbers. Between the two positions of those numbers, swap any numbers that are not sorted in ascending order (by again starting the comparison from the last element in the sequence downto the element to the "right" of the pivot and swapping any non sorted elements).

For example, for the sequence 1123, 3 is the last element and 2 being the first smaller element becomes the pivot. There are no other elements between the two, therefore after swapping, 1132 is obtained. Reapplying the same reasoning, **1** becomes the pivot and is swapped with 2 giving 1231 (2 becoming the new pivot). Between 2 and the last element, 31 are not sorted in ascending order and are therefore eligible for swapping, giving 1213. Reapplying the procedure until all permutations are output gives all 12 permutations..

Note: The previous example was one of a sequence with repetitions and our procedure gave (4!)/(2!) = 12 permutations and not 4! = 24 (having 4! permutations would mean not only that the permutations were not unique but also that efficiency was reduced). The Procedure permute is designed as follows:

```
Procedure permute(z:array of integer);
i,j,k,l,max:integer;        -- max being the length of sequence z
bool : boolean;
{z[0]:=0;                   -- used as a sentinel for exiting while loop
i:=1;                          (thus preventing loop from running off)
while(i<>0) do
 {bool:=false;   OUTPUT z;
  i:=max-1;
  while (z[i]>=z[i+1] ) do i:=i-1; -- Find the pivot
  j:=max;
  while (z[i]>=z[j])   do j:=j-1;  -- Find the number to swap with the pivot
  swap(z[i],z[j]);
  k:=max;          l:=i+1;         -- positions of last element and the one "right" to the pivot
  while(k>l) do                    -- Does another search to verify that
  { swap(z[k],z[l]);               -- there is no other number eligible
    k:=k-1;                        -- for swapping
    l:=l+1; }
output z
```

```
}                                   -- end of while (i<>0)
}                                   -- end of procedure
```

Such procedure is called, after each new partition is generated to ensure that all compositions for that particular partition are generated.

The first line of the make_partitions(n,s,P1,P2) procedure is therefore slightly modified to include the permute procedure. This new procedure make_compositions(n,s,P1,P2) is *exactly the same* as the make_partitions(n,s,P1,P2) except that it includes the permute procedure: therefore it is defined as follows:

```
Procedure make_compositions(n,s,P1,P2)
{n←t;
 while (p[1] <=min) do
  {bool←false;
  OUTPUT sequence p;        z←p;        permute(z);
       (Remainder of procedure remains same as make_partitions;)
```

3.4 Generation of B-Trees in a lexicographic order

[GLW1] use the make_partitions(n,s,P1,P2) algorithm for the complete generation of B-Tree sequences. The following briefly explains their algorithm. We then define our lexicographic order and show that [GLW1] indeed generates B-Trees in a lexicographic order. As we present a new algorithm in the next section, a high level description of the [GLW1] is given, to show how our algorithm differs from theirs. For more details, refer to [GLW1].

3.4.1 Iterative generation of B-Trees [GLW1]

Procedure make_partitions(n,s,P1,P2) was shown to generate all partitions of a number n

with s parts. If b-subsequences are thought of as restricted partitions and a-subsequences as restricted compositions, then the same procedure can be extended to generate B-Tree b-subsequences. In fact, if n is thought of as the number of nodes, s the number of parts and P1 and P2 as $\lceil m/2 \rceil$ and m (m being the order of the B-Tree), then the make_partitions(n,s,P1,P2) procedure does generate all b-subsequences at a particular level. However, it does not generate permutations of all the b-subsequences at different levels (i.e. all a-subsequences corresponding to a given b-subsequence).

[GLW1] use the make_partitions(n,s,P1,P2) procedure to generate all B-Tree sequences from the first a-sequence as follows:

given n, generate the first a-sequence. To get to the next a-sequence, the prefix of the given a-sequence is kept intact as much as possible (as shown in section 3.3). [GLW1] therefore propose to scan the a-sequence backward, recognizing each sub-sequence.

For example, let 455445332 be an a-sequence. Starting from the last number, number 2, scan backwards two numbers, thus giving 33 as a-subsequence. Subsequence 33 adds up to 6 (i.e. 3+3). Therefore, they scan backwards the next six numbers, giving the next a-subsequence 455445 which happens to be the last subsequence.

Once the first a-sequence is obtained, [GLW1] propose the following very high level explanation on how to obtain the next a-sequences from the first a-sequence:

given n , obtain the least (first) a-sequence . To get the next a-sequence , [GLW1] keep intact the prefix of the given a-sequence as much as possible and then recognize each a-subsequence by scanning the a-sequence backward.

If the a-subsequence is not lexicographically maximum (i.e. it is not identical to its

corresponding normalized b-subsequence) then find an a'-subsequence that immediately follows the a-subsequence in lexicographical order ([GLW1] do not explain how they achieve finding this a'-subsequence), otherwise they use the make_partitions(n,s) with n being the sum of the number of the sequence , s the length of the subsequence to find the next partition.

If the next partition exists, then obtain the a'-subsequence which is lexicographically minimum based on the partition (in fact it is the reversal of the sequence corresponding to the partition).

If the next partition does not exist then increase the partition size by 1 to see if lexmin(n,s+1) is feasible. If it is feasible a new a'-subsequence which is lexicographic with respect to Lexmin(n,s+1) is obtained, else continue scanning the a-subsequence to recognize the next a-subsequence (repeat the process until no more a-subsequence can be found).

Whenever a new a'-subsequence is obtained, the remaining portion of the a-sequence is filled with the least a-sequence with respect to n', where n' is the length of the newly created a'-subsequence.

Unfortunately, no complete algorithm or pseudo-code are provided for finding the next sequence in a lexicographic order.

3.4.2 Lexicographic order.

Before pursuing with the remainder of the chapter, we define our lexicographic order in this section. [GLW1] defines an order to correspond to the above algorithm. We noted that the order is lexicographic if a suitable tree sequence representation is used. We point

out that the order used in [GLW,GLW1] becomes lexicographic if B-Trees are coded in the following way :

$$l_1 b_1 b_2 \ldots b_{l_1} a_1 a_2 \ldots a_{l_1}\, l_2 b_{l_1+1} \ldots b_{l_2} a_{l_1+1} \ldots a_{l_2} \ldots l_r b_{l_r} a_{l_r}.$$

We call this sequence a ***B-tree sequence***. Such B-tree sequence can be viewed as level x | b-subsequence at level x | a-subsequence at level x | (same at next level) etc. ... [GLW1] generates the sequences starting from the smallest number of parts up to the highest number of parts (each part size is within the order of the B-Tree).

The number of parts at the first level corresponds to l_1 and as all the B-Tree sequences that are generated start with the level value, it is therefore safe to say that all sequences that exist are already sorted by level. Unfortunately, sorting just on the level value is not enough as partitions may have the same number of parts (i.e. 5554444 and 6544444) and generating combinations (i.e. defined as a-sequences) for those partitions will result in the order being wrong (i.e. one combination for 5554444 is 4444555 which is lexicographically larger than 4444456, a combination for 6544444). One way to solve this problem is to "append" the normalized sequence after the level value, before generating any combinations, as b-sequences are generated in lexicographic order, thus guaranteeing lexicographic order the way we defined it.

The ordering on B-trees is then easily defined as lexicographic order on the corresponding B-tree sequences.

For example and without, for simplicity's sake, giving any details about the numbers at levels above l_1 , let 5554444 be a b-subsequence partition, $l_1 = 7$ (the number of parts in 5554444), $b(1)b(2)...b(l_1) = 5554444$, $a(1)a(2)...a(l_1) = 4444555$, then the following B-

Tree sequences will be ordered as follows (bold characters show where the sequence has changed):

- For b-subsequence partition 5554444
  7 5554444 4444555 $l_2b(l_1+1)...b(l_2)a(l_1+1)...a(l_2)...\ l_rb(l_r)a(l_r)$.
  7 5554444 4445455 $l_2b(l_1+1)...b(l_2)a(l_1+1)...a(l_2)...\ l_rb(l_r)a(l_r)$.
  .....
  7 5554444 5554444 $l_2b(l_1+1)...b(l_2)a(l_1+1)...a(l_2)...\ l_rb(l_r)a(l_r)$.
- For the next b-subsequence partition 6544444
  7 6544444 4444456 $l_2b(l_1+1)...b(l_2)a(l_1+1)...a(l_2)...\ l_rb(l_r)a(l_r)$.
  7 6544444 4444465 $l_2b(l_1+1)...b(l_2)a(l_1+1)...a(l_2)...\ l_rb(l_r)a(l_r)$.
  .....
  7 6544444 6544444 $l_2b(l_1+1)...b(l_2)a(l_1+1)...a(l_2)...\ l_rb(l_r)a(l_r)$.

and so on, for the number of parts = 7.

While the sequence under this lexicographic order might look unnecessarily long, it is nonetheless a very structured way of ordering sequences based on a partition. In fact, the search for a specific sequence becomes much easier under this lexicographic order, as all sequences are generated for a specified number of parts (which was not really the case in [GLW1] where the first three partitions output by their algorithm are: 5554444, 6544444, 655555, obviously missing the sequence 7444444: one reason for this omission could be that according to their algorithm, the next partition would start with an 8 which is not allowed for an order of 7, and therefore the program terminates, leaving the remaining 12 partitions unprocessed).

Because the previous B-Tree sequences are generated using the number of parts (for the first level) as a parameter, then all B-Tree sequences with the same number of parts will have the same l-sequence and the same a-subsequences after l_2 (in fact, it is necessary to just get the subsequences after l_2 for the first partition to know exactly what the

subsequences after l_2 for the next partition are, as when the algorithm is applied with the same parameters, it will output the same sequences: this result will be used in our algorithm in the next section).

For example, let the B-tree sequence be

$$7\ 5554444\ 4444555\ l_2b(l_1+1)...b(l_2)a(l_1+1)...a(l_2)...\ l_rb(l_r)a(l_r).$$

Because the order of the B-Tree is 7, the sizes of the parts belong to M={4,5,6,7}, therefore $l_1 = 7$ can only be decomposed as (3,4) or (4,3). Those represent 2 elements, therefore $l_2 = l_1 + 2 = 9$, the b-sequence at level 2 is 43 and the first a-sequence is 34, thus giving

7 *5554444* 4444555 **9** *43* 34 **11 2**
7 *5554444* 4444555 **9** *43* 43 **11 2**

7 *5554444* 4445455 **9** *43* 34 **11 2**
7 *5554444* 4445455 **9** *43* 43 **11 2**

7 *5554444* 4445545 **9** *43* 34 **11 2**
7 *5554444* 4445545 **9** *43* 43 **11 2**
.....
7 *5554444* 5554444 **9** *43* 34 **11 2**
7 *5554444* 5554444 **9** *43* 43 **11 2**

(l-sequence is in **bold**)

As can be observed in the previous example, the subsequences after l_2 ($l_2 = 9$) remain the same for every new partition as they are based on the number of parts at the leaf level (level 1). In effect, as the algorithm is being applied with the same parameters at every level, it outputs the same results (i.e.sequences) at every level.

The previous algorithm generates subsequences for the leaves level (l_1) only. To get the subsequences at the other levels, make_partitions(n,s,P1,P2) can be recursively called with s being the number of parts (or sub-sequence length) obtained at each level and the sum of

all parts at that level being the new value for n, as shown in the next paragraph.

3.5 Recursive generation of B-Trees.

In this section, we give a different approach to the lexicographic generation of B-Trees of order m. The algorithm in [GLW1] used a fixed number of leaves and had a number of parts varying between P1 and P2. [GLW1] generated B-Trees in the lexicographic order that we defined by getting the next a-sequence from the current one iteratively and in a backward manner (starting scanning from the end of the sequence) as explained in the previous section.

The proposed approach is to generate B-Trees in a lexicographic order by also using the same tree representation as in [GLW1] and a fixed number of leaves. The main difference is that it will be carried out in a forward manner and using a recursive approach.

The major steps will be as follows. All partitions for n at the leaves level are generated. The main difference with [GLW1] resides in the fact that for each partition, we generate all compositions corresponding to that partition (the same way as defined earlier). The main idea is again to think of partitions as b-subsequences and compositions as a-subsequences. The obvious result being a generation of all b-subsequences and a-subsequences at the leaf level. The algorithm is then gradually expanded to the next levels as explained later, to generate all B-Trees of order m in a lexicographic order.

3.5.1 Complete lexicographic order B-Trees generations

The make_compositions algorithm generated all partitions in a lexicographic order, with all the permutations for each partition. Again, if restricted partitions can be thought of as b-subsequences and restricted compositions as a-subsequences, then the make_compositions algorithm can be used to generate B-Trees a-subsequences and b-subsequences to eventually generate the entire B-Tree a-sequence. The transition from generation of partitions and combinations to generation of B-Trees is explained at the end of this section.

For the sake of clarity, the focus will be on the first level (leaves level) only, to then be applied at every other level: this should not be a big restriction as our proposed algorithm is recursive and therefore applied with different parameters at every level. The make_compositions algorithm generated all partitions in a lexicographic order, with all the permutations for each partition, but only with a fixed number of parts. The algorithm can be modified to produce all partitions in a lexicographic order, with all the permutations for each partition for all numbers of parts *consistent* with the order m of the B-Tree. A consistent number of parts is defined as the number for which the sum of the parts will never exceed n and for which the size of the parts are between $\lceil m/2 \rceil$ and m (except for the root whose size can be from two to m). This is an example of a doubly restricted partition as defined earlier. This consistent number of parts is between two numbers P1 and P2 given by procedure num_parts as follows:

Procedure num_parts (n,m,P1,P2)
{ P1 = $\lceil n / m \rceil$;
P2 = $\lfloor n / \lceil m/2 \rceil \rfloor$; }

For example, let n=31, m=7 : the consistent number of parts is between 5 (from $\lceil 31/7 \rceil = 5$) and 7 (from $\lfloor 31/\lceil 7/2 \rceil \rfloor = 7$). Some subsequence examples are 77764 for 5 parts and 7444444 for 7 parts : 4 parts are not consistent as 7777 only adds up to 28 and neither are 8 parts as 44444444 adds up to 32.

At a very high level, the main algorithm can be summarized as

```
num_parts (n,m,P1,P2);
For range ← P1 to P2 do
make_compositions(n,range,P1,P2);
```

This loop will ensure that for one level only (the first level or leaf level for the purpose of this discussion), all B-Tree subsequences are generated. As generating all B-Trees is the purpose of this section, the main interest is on how to use the generation of subsequences to generate all B-Tree sequences.

As mentioned earlier, the make_compositions procedure can be thought of as generating all the a-subsequences and b-subsequences at one level only. If this level is chosen to be the leaf level, then it is permitted to say that the make_compositions procedure is capable of generating part of a B-Tree sequence

level 1 | b-subsequence at level 1 | a-subsequence at level 1

In order to have the entire a-sequence, it is necessary to get the a and b-subsequences at all the remaining levels. To achieve this, it is necessary to recursively apply the same algorithm at every level (using, obviously, different parameter values). The four parameters used by the algorithm are computed as follows:

- n (the initial number of leaves) will become the number of parts at the previous level (namely s) for non leaf levels,

- s (the number of parts) is between P1 and P2, as given by procedure num_parts.

This process is repeated until the number of parts is smaller than $\lceil m/2 \rceil$ and cannot be decomposed within the permitted size of the parts (i.e.P1 and P2) at that level.

The recursive aspect of the make_compositions procedure at every level is *twice* interesting as not only it is recursive (and therefore less code is required) but also it opens the door to the use of multiprocessors as shown in what follows.

- A multi processor program can easily be designed to handle the generation by level, thus having a greater impact on the performance. The main idea is that $m_i = l_i - l_{i-1}$ would be passed to the next processor as the number of parts. The sum of those parts would be passed as the parameter n. The same algorithm, make_compositions will be applied by this processor to output all the B-Tree sub-sequences at that level. This processor can pass in its turn new values for n and s to another processor and so on until there are no more parts to process.
- Another multi processor program can easily be designed to handle each partition that is being generated. In effect, the make_partitions procedure generates all partitions for a number n of leaves. Instead of waiting for each partition to finish before processing the next partition, it is possible to first generate all the partitions (let N be the number of partitions) and have N processors generate all B-Tree sequences for those N partitions (i.e. one processor per partition).

Chapter IV

Proof of constant average delay property

We showed in Chapter 2 and Chapter 3 that the order of generating 2-3 trees and B-trees in [GLW] and [GLW1] was lexicographic if a suitable sequence representation was used. In this section, we prove that the algorithm in [GLW1] generates B-Trees in constant expected time, exclusive of the output time (i.e. it has a constant average delay property).

4.1 Constant average delay property

Let an (h,m) composition of h be any integer sequence $b_1...b_t$ such that $\lceil m/2 \rceil \leq b_i \leq m$ for each i, $1 \leq i \leq t$, and $b_1+...+b_t = h$ and let C(h,m) be the number of such distinct (h,m) compositions. In [GLW1], normalized sequences are generated using a procedure that generates restricted integer partitions, and for an obtained integer partition all corresponding integer compositions (where the order of terms becomes important) are found. In our proof we only use the fact that the algorithm generates B-trees by generating such integer compositions, where the next composition can be determined from the current one in linear time. In this sense, our proof may be applied to some other algorithms for generating B-trees (for example, one which would avoid generating integer partitions and directly update integer compositions).

We use the following lemma in our proof:

Lemma. $C(h,m) \geq \frac{5h^2}{96m^2}$ for $h \geq 4m$ and $m \geq 3$.

Proof. Let $p_1d_1 + p_2d_2 + \ldots + p_sd_s = h$ be the normalized sequence for the (h,m)-composition $b_1 + \ldots + b_t = h$ where $p_1 > p_2 > \ldots > p_s$ are all distinct parts in the given (h,m)-composition and $d_1, \ldots, d_s$ their multiplicities, $d_1 + \ldots + d_s = t$ (i.e. $22244 = 2^3 4^2$, 2 has multiplicity 3, 4 has multilicity 2). The case s=1 may occur only when $t=h/m$ ($b_1 = \ldots = b_t = m$) or $t = h/\lceil m/2 \rceil$ ($b_1 = \ldots = b_t = \lceil m/2 \rceil$).

Otherwise, for each t, $\lceil (h+1)/m \rceil \leq t \leq \lfloor (h-1)/\lceil m/2 \rceil \rfloor$, there exists at least two different parts ($s \geq 2$). We show that for each t in the given interval there exist at least t+1 (h,m)-compositions, which can be obtained as follows. Starting from

$$p_1+\ldots+p_1+p_2+\ldots+p_2+\ldots+p_s+\ldots+p_s = \sum_{i=1}^{s} p_id_i = h,$$

the last p_1 "walks" through the (h,m)-composition to generate $d_2+\ldots+d_s=t-d_1$ new (h,m)-compositions

$$p_1(d_1-1) + p_2d_2 + p_{i-1}d_{i-1} + p_id' + p_1 + p_i(d_i - d') + p_{i+1}d_{i+1} + \ldots + p_sd_s = h,$$

for $d' = 1,\ldots,d_i$ and $i=2,\ldots,s$.

Similarly, when p_s "walks" through the list $d_1+\ldots+d_{s-1}= t-d_s$ (h,m)-compositions are produced. Thus there are at least $t-d_1+t-d_s+1 \geq t+1$ (since $d_1 + d_s \leq t$) (h,m)-compositions for each t.

Let $H = \lceil (h+1)/m \rceil$ and $Z = \lfloor (h-1)/(\lceil m/2 \rceil) \rfloor$.

Therefore the number of compositions of (h,m)-compositions of h is:

$$C(h,m) \geq (H+1) + (H+2) + \ldots + (Z+1) = (Z+H+2) * (Z-H+1)^{1/2}$$

From $\lceil m/2 \rceil \leq (m+1)/2$ and $\lfloor x \rfloor > x-1$ it follows that

$$Z \geq \lfloor 2(h-1)/(m+1) \rfloor > (2(h-1)/(m+1)) - 1 .$$

For $m \geq 3$ it easily follows that $2/(m+1) \geq 3/2m$. Thus

$$(Z + H + 2)$$

$$\geq [\, 3(h-1)/2m] - 1 + [(h+1)/m] + 2$$

$$\geq (3h - 3 + 2h + 2 + 2m) / 2m$$

$$\geq 5h/2m.$$

Consider now the second expression. From $\lceil (h+1)/m \rceil < [(h+1)/m]+1$ and $\lfloor (h-1)/\lceil m/2 \rceil \rfloor > 3(h-1)/2m - 1$, it follows that

$$(Z - H + 1) > [3(h-1)/2m] - 1 - (h+1)/m - 1 + 1.$$

We now prove that $3(h-1)/2m - (h+1)/m - 1 \geq h/24m$ for $h \geq 4m$ and $m \geq 3$. The condition is (after doing obvious transformations) equivalent to $h \geq (24/11)m+(60/11)$. It easily follows from $h \geq 4m \geq (24/11)m + 60$ which is satisfied for $m \geq 3$.

From both approximations it follows that

$$C(h,m) \geq [(5h)/(2m)] * [(h/24m)] * [(1/2)] = (5h^2)/(96m^2)$$

$$\text{for } h \geq 4m \text{ and } m \geq 3$$

Although a much stronger lemma can be derived (probably with more sophisticated arguments), this weak property is sufficient to prove the constant time average delay behavior of the above algorithm.

Theorem. *The Gupta-Lee-Wong algorithm [GLW1] has constant expected delay in*

producing the next B-tree, exclusive of output.

Proof. Consider level i sequence $a(l_{i-1}+1)...a(l_i)$. The lexicographically next level sequence can be determined by a backtracking procedure that finds in a backward run an element that can be increased, and in a forward run finds the next sequence. The time for updating is linear in the number of elements of the sequence , i.e. it is bounded by cu_i, where c is a constant and $u_i=l_i-l_{i-1}$. However, the current level sequence can be completed in various ways to produce distinct B-tree sequences. During the production of these B-trees the algorithm for finding the next B-tree does not "arrive" in its search for the turning point to level i. For a fixed sequence $a(l_{i-1}+1)...a(l_i)$, the sequence at the next level $a(l_i+1)...a(l_{i+1})$ can be any (u_i, m)-composition $u_i=l_i-l_{i-1}$ into parts with sizes between $\lceil m/2 \rceil$ and m, according to the definition of tree sequences. The number of such (u_i, m)-compositions is at least $(5u_i^2)/(96m^2)$ whenever $u_i \geq 4m$ (from the lemma). Therefore, cu_i operations are performed on level i when the turning point comes back to the level (even when the turning point continues its search to other levels), and in the meanwhile at least $5u_i^2/96m^2$ B-trees are produced that contribute zero operations (level i sequence intact). Thus the average number of operations for level i is at most $cu_i/(5u_i^2/96m^2)=96m^2c/5u_i$.

The B-tree sequence $a_1...a_k$ is always partitioned into levels $l_1,...,l_r$, with sizes $u_i=l_i-l_{i-1}$, $1 \leq i \leq r$, and $u_i \geq \lceil u_i - 1/m \rceil \geq u_{i-1}/m$ is always satisfied, i.e. $u_i \leq m^{r-i}u_r + m^{r-i}$. Since $u_r=1$ and $u_{r-1} = a_k \leq m < 4m$, the condition $u_i \geq 4m$ is not satisfied for $i = r - 1$ and $i = r$. Let j be the maximal index such that $u \leq 4m$ (clearly $j \leq r - 2$). The average number of operations for all levels 1,2,...,j together is smaller than

$$(1/u_1 + 1/u_2 + \ldots + 1/u_t)\, 96m^2c/5 \leq (1/m^{r-1} + 1/m^{r-2} + \ldots + 1/m^{r-j})\, 96m^2c/5$$

$$< (96m^2c/5\, m^{r-j})\, (m/m-1)$$

$$< 40c$$

which is a constant. ■

The case which was not counted in is when $u_i < 4m$. Because each part has a size $\geq \lceil m/2 \rceil$, it means that the compositions of any such u_i has at most 7 parts ($t \leq 7$ in $b_1 + b_2 + \ldots + b_t = u_i$). Depending on m, there are between 1 and 3 parts in a composition of 7 and probably one more part of size 2 or 3, i.e.: at most 4 more elements in tree sequence. Therefore the part of the a-sequence which is not considered in our operation count contains at most the last 11 elements. Since the algorithm is a backtrack search, for each new tree, it takes at most constant additional time to pass through them in both backward and forward direction. Therefore the algorithm has overall a constant time delay.

Our proof of constant average delay property assumes only a kind of level sequence representation and their lexicographic order. More precisely, the proof requires merely that an algorithm generated all B-trees with given fixed a-subsequence (level sequence) in a block of consecutive sequences, such that the sequences in the block can be generated without any work performed on the given a-subsequence except at the beginning and end of block where linear update time applies. Therefore the proof applies, for instance, to an algorithm that generates B-trees as the only part of a B-Tree representation (i.e.: the l-sequences and normalized sequences are not part of the representation).

There exists a straightforward decoding procedure that generates the B-tree data structure (meaning that the parent-children links are established) from a given B-tree sequence. Since only the part of the sequence that is updated needs to update its parent-children links on the procedure, the algorithm can be completed to produce B-tree data structures with a constant average delay.

Chapter V

Coding and decoding of B-Trees

In this chapter, three sequential algorithms are presented. Two of them are for decoding a sequence and the third one is for coding a given array of parent-children relationship.

Decoding, as seen in chapter 1, is defined as finding the corresponding B-tree Parent-Child representation of a B-Tree given its sequence representation while coding was defined as the opposite problem: given a B-Tree Parent-Child representation, find the corresponding sequence representation. The coding and decoding of B-Trees are useful if one is interested in a one to one relationship between the two previously different B-Tree representations. The algorithms presented in this chapter will be derived from both the Parent-Child and the sequence representations (such representations have been defined in the first chapter of this thesis).

5.1 Decoding

Given the sequence representation of a particular B-Tree, we are interested in finding an array of records, where each record contains relevant information such as who is the parent node (except for the root), who are the children nodes (in the case of internal nodes) and the node number in the tree, thus creating a parent-child relationship. An example of that array is shown in Figure 3 of the first chapter.

The following terms and notations will be defined before giving a formula to derive the parent-child relationship array.

Let n be the total number of nodes in the tree, and k the maximum index in an <a>sequence $a_1a_2... a_k$ where a_m is the number of children of node (n-k+m). Clearly, if n is the total number of nodes and k is the number of internal nodes, then the number of leaves NL is given by (n-k), therefore a_m is also the number of children of node (NL + m).

We define D_m as follows:

$$D_m = \sum_{i=0}^{m} a_i$$

where $D_0 = a_0 = 0$.

The children of the node (n-k+m) are the nodes $(1 + D_{m-1})$ to (D_m).

For a better illustration of the concept, the following figures (Figure 12 and Figure 13) are tree representations of the <a> sequence 2 3 2 2 3 3 2 2. Figure 12 gives the node numbering (or node indices) while Figure 13 gives the number of children of nodes. For consistency with the node numbering, the index i of any a_i from the sequence $a_1a_2...a_8$ (denoted as a1a2 ...a8 in Figure 13) can also be viewed as the numbering of the internal nodes (i.e.: a_m corresponds to the m-th internal node).

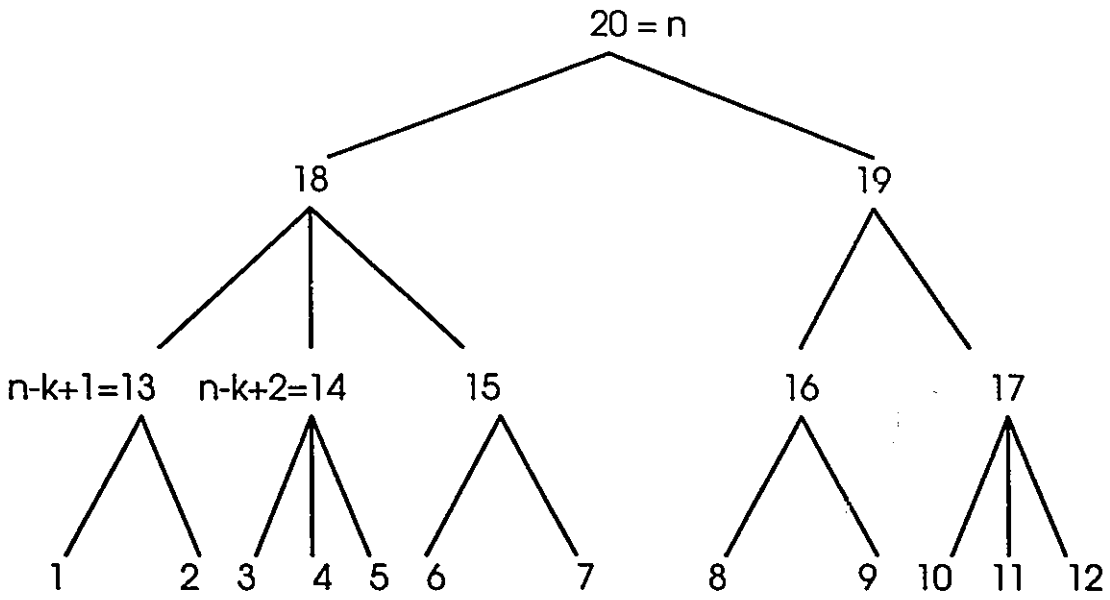

Figure 12

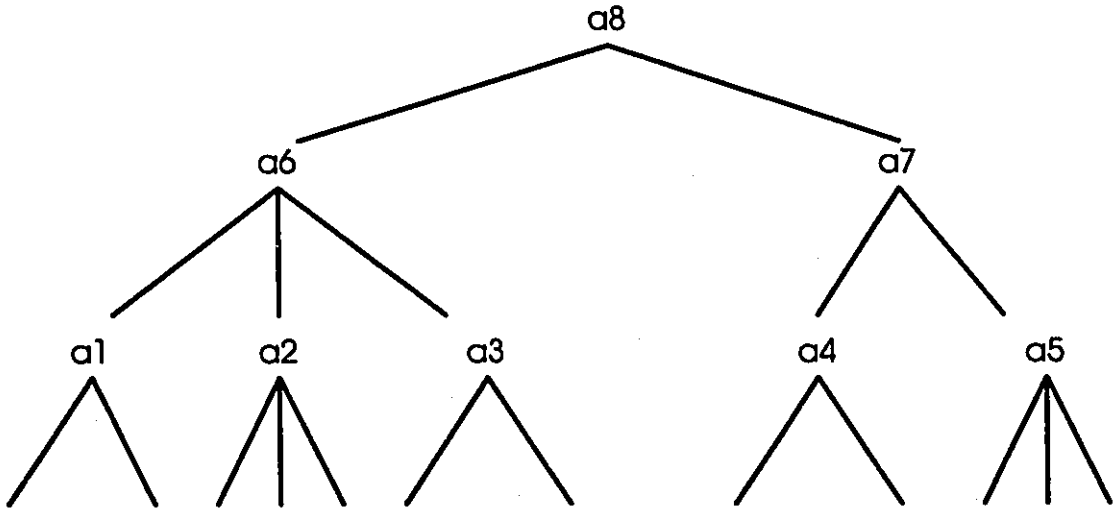

Figure 13

For example, in the following sequence, 23223322, the children of the fifth internal node (corresponding to a5 in Figure 13 and n-k+5=17 in Figure 12) are the nodes from

$1 + D_{m-1}$ to D_m. $= 1 + D_4$ to D_5.

$= 1 + (a_1 + a_2 + ... + a_4)$ to $(a_1 + a_2 + ... + a_5)$

$= 1 + (2+3+2+2)$ to $(2+3+2+2+3)$

$= 10$ to 12.

The previous formula was to define the children node numbering at a particular non-leaf node. As decoding is supposed to give a parent-children relationship, the next paragraph will define the parent of a node.

By looking at both Figure 12 and Figure 13, a first correlation was derived between node numbers in Figure 12 and their sequence index in Figure 13 as being:

Node number of the i-th element (Figure 13) = i + n - k (Figure 12)

For example, the first internal node in Figure 13 (which corresponds to a_1) has a one to one relationship with node 1+12=13 in Figure 12, the second internal node (which corresponds to a_2) with 2+12=14 up to the 8th internal a_8 with 8+12=20. Therefore, there is a mapping (one to one correspondence) between non-leaf node number and sequence index. The main idea behind getting the parent is summarized in the sequel: any a_i corresponds to the number of children of the (n-k+i)-th node. Starting from the first internal node (n-k+1 or node 13 in Figure 12), it is clear that all children nodes at (n+k+1) not only have the same parent but also have node numbers less or equal to a_1. Carrying on to the next element (n-k+2), node number 14 in Figure 12, all children nodes have the

same parent and their node numbers are between $(a_1 + 1)$ and $(a_1 + a_2)$.

In a more general scope, all children nodes at (n-k+i) have a parent whose node number is (n-k+i) and their node numbers are between

$$(a_1 + a_2 + \ldots + a_{i-1} + 1 = \sum_{r=1}^{i-1} a_r + 1) \text{ and } (a_1 + a_2 + \ldots + a_i = \sum_{r=1}^{i} a_r)$$

as defined earlier. This result is used in the next paragraph to derive algorithms for decoding a sequence.

5.2 ALGORITHMS FOR DECODING

For the same input sequence 23223322, the tree in Figure 12 shows the node numbering, while the tree in Figure 13 shows the sequence numbering of each element of the sequence representing the number of nodes it contains. By superimposition of the two figures (i.e. by placing Figure 12 over Figure 13), a one to one correspondence was derived between the index of an element of the sequence and the node number of the parent node. Such correspondence is defined as:

Parent of children at (n-k+i) = n - k + i = (number of leaves) + i

For example, and still using superimposition:

Parent of all children at 1st internal node = 1 + 12 = 13

Parent of all children at 2nd internal node = 2 + 12 = 14

Such property is very useful in defining the array in this algorithm. It is therefore possible to do the inverse operation: given a parent node number, find the children node numbers. The latter (finding children node numbers) is more complex than finding the parent node number in the sense that this relation could be **one to many** relationship, but if we restrict

ourselves to the sequence itself, we can still obtain the expected results.

5.2.1 First decoding algorithm

In this section, we present the first algorithm for decoding using a backward approach.

For the algorithms, the structure of the record could be summarized (in Pascal) as follows:

```
Record Structure
Node = Record
            Num, Parent :longint;
            Child : array[0..max] of longint;
        end
```

The algorithm is as follows: given an input sequence called <a>, in a backwards manner (i.e. starting from the last element of the sequence), find out how many children each non leaf node has and backtrack until there are no more non leaf nodes to process.

For example, let us use the sequence 2 3 2 2 3 3 2 2 from Figure 13. The last number is 2 (2322332 **2**), therefore the root has 2 children, which are nodes 18 and 19. The number of children of nodes 18 and 19 respectively are given by backtracking twice in the sequence <a> thus obtaining the sub sequence **32** (in our sequence 2 3 2 2 3 **3 2** 2), whose elements add up to 3 + 2 = 5 meaning that nodes 18 and 19 have 5 children. To be more precise, nodes 18 and 19 have 3 and 2 children respectively (which are nodes 13,14,15 and 16,17). The number of children for nodes 13, 14, 15, 16, 17 is given by the subsequence **2 3 2 2 3** (**2 3 2 2 3** 3 2 2) obtained by backtracking 5 times from the last subsequence (i.e.: the number of children for nodes 13, 14, 15, 16 and 17 are respectively 2, 3, 2, 2, 3).

Clearly, as we start with the last number, the algorithm starts from the root, whose node number = $a_1 + a_2 + ... + a_8 + 1$ which is equal to 20 in this particular case. Because we start from the root and process nodes down to the first level, the order to follow is now

from the top down and level by level from the right to the left (this explains the loops in the decod1 algorithm, presented below, going from one value **downto** another value). Based on this approach, which has the advantage of not needing to know in advance the number of leaves in the tree, it is possible to derive a parent-children relationship from the following algorithm, algorithm Decod1, which will take as input a sequence called <a> with n nodes and the sequence length, seq_length, of <a>. It will output a table of records containing the parent-child relationships. For clarification, this algorithm (Procedure DECOD1) will be accompanied by a trace (on the right hand side) for the first pass of the algorithm. The input sequence is 23223322.

Procedure DECOD1

```
seq_length: integer;                                  -- length of the sequence
child    : array[1..max_nodes,1..max_child] of integer; -- contains info about children
a        : array[1..seq_length] of integer;           -- contains the sequence
parent   : array[1..max_nodes] of integer;            -- contains info about the parents
n        : integer                                    -- the number of nodes
acc      : integer                                    -- accumulates node values
{
n←1;
INPUT seq_length ;          -- get the length of the sequence
for i ← 1 to seq_length do
{ INPUT(a[i]);              -- get sequence
 n←n+a[i];}                 -- add up number of elements to have total.
                            -- so n will contain the exact number of nodes
                            -- n=1+2+3+2+2+3+3+2+2=20
L←n;                        -- L=20
acc←0;                      -- will add node values as they are being processed
for j←seq_length downto 1 do --we go from last element of sequence to the first
{                           -- trace with tree representation
 for k←a[j] downto 1 do     -- a[j] contains the exact number of elements at index j
  {                         -->1st time in loop   |2nd time      |New loop      |2nd
 sum1←n - (k +acc);         -->=20-(2+0)=18       |20-(1+0)=19   |20-(2+2)=16   |=17
 child[L,k]←sum1;           -->=18                |=19           |=16           |=17
 parent[sum1]←L;            -->=20                |=20           |=19           |=19
  }                                                |              |              |
  acc←acc+a[j];             -->=0                 |0+2=2         |2+2=4         |
  L←L-1;                    -->=20                |20-1=19       |19-1=18       |
 }
```

for i←1 **to** n **do** Output Parent, Children Nodes

As the number of statements inside of the double loop is constant, every statement can be assigned one node (child), and each child node is mentioned exactly once in all loop entrances, then clearly the complexity of the algorithm is O(n).

5.2.2 Second algorithm (based on the number of leaves)

The previous algorithm (DECOD1) did not take into account the number of leaves in a B-Tree. Another approach to decoding is to find the parent-children relationship based on the lowest level sequence (leaf level). Once again the sequence 2 3 2 2 3 3 2 2 from Figure 12 is used as illustration.

As mentioned at the end of section 5.1, there is a mapping (one to one correspondence) between non-leaf node number and sequence index i.e.: the index represents the offset from the first level starting from the left. If the number of nodes is added, the node number would then represent the offset from the leaf level starting from the left. Obviously this only gives the parent node number. In order to also give a node number to the children, it is necessary to remind that only node numbers that have a value greater that the number of leaves have children; therefore a counter can be used (that can start adding up as soon as a non leaf node is processed) to number the children as is shown in procedure DECOD2 below. Based on that idea, the following algorithm (DECOD2) will give the relationship table between those nodes.

The first algorithm, DECOD1 was based only on the sequence. The DECOD2 algorithm is based on both the sequence and the number of leaves (both taken as input). Although it

would be easy to find out the leaf sequence, both algorithms work well and are easy to implement.

Procedure DECOD2:

```
{num_leaves:integer                          -- represents the number of leaves in the tree
read(a);                                     -- read the input sequence a
n←0
count←1                                      -- the child counter
for i←1 to length(a) do
n←n + a[i];                                  -- Gets total number of nodes
acc←0;
for i←1 to length(a) do
  {
  for j←1 to a[i] do
        {
        node[j+acc].num←j + acc;             -- assign a number to the node
        node[j+acc].parent←num_leaves+i;     -- the parent is num_leaves 'further'
                if (j+acc)> num_leaves then  --If non leaf, give a number
                                             --to child
                for k←1 to a[j+acc-num_leaves]
                do {
                Node[j+acc].child[k]←count;  -- the child is assigned a number
                 count←count + 1;
                }
        }
        acc←acc+a[i];              }
For i←1 to n do Output node[i].parent, node[i].child[k]  -- k from 1 to max_child
```

The Decod2 algorithm is simply a linear scan through the <a> sequence $a_1a_2...a_k$ and requires a constant number of operations for each element. As the number of statements inside of the double loop is constant and every statement can be assigned to one node (child or parent), each node is mentioned exactly once in all loop entrances, then clearly the complexity of the algorithm is O(n).

When executed with the sequence 23223322 as input sequence, both algorithms DECOD1 and DECOD2 produced the resulting parent-child relationship table shown in Table 2.

Node	parent	ch1	ch2	ch3
1	13	0	0	0
2	13	0	0	0
3	14	0	0	0
4	14	0	0	0
5	14	0	0	0
6	15	0	0	0
7	15	0	0	0
8	16	0	0	0
9	16	0	0	0
10	17	0	0	0
11	17	0	0	0
12	17	0	0	0
13	18	1	2	0
14	18	3	4	5
15	18	6	7	0
16	19	8	9	0
17	19	10	11	12
18	20	13	14	15
19	20	16	17	0
20	0	18	19	0

Table 2

5.3 Algorithm for coding.

This section presents an algorithm for the opposite of decoding, the coding, which will, given the parent-child relationship table, find the sequence that was used to generate it. The B-Tree in Figure 12 is used to illustrate our concept. By following the <a> sequence definition given at the beginning of this chapter, the tree representation for Figure 12 has an a-sequence equal to 2 3 2 2 3 3 2 2. The root (node 20) has 2 children (18 and 19), node 18 has 3 children (nodes 13,14,15) and so on.

The approach to finding the sequence would be to scan through each node in the array and

compare its parent with the parent of the next node; if they are the same, then they are **siblings** and a counter will tell exactly how many siblings there are.

The value of the counter also happens to be the value of the element in the sequence. The process is repeated until the root node is reached (or in other words until a node with no parent is found), always recording the counter in an array thus finishing with the whole sequence.

It is possible to derive the sequence corresponding to the information contained in Table 3 (which corresponds to the output of the two previously defined decoding algorithms) by following the algorithm proposed in the previous paragraph:

Node#	Parent	child1	child2	child3
1	12	0	0	0
2	12	0	0	0
3	12	0	0	0
4	13	0	0	0
5	13	0	0	0
6	13	0	0	0
7	14	0	0	0
8	14	0	0	0
9	14	0	0	0
10	15	0	0	0
11	15	0	0	0
12	16	1	2	3
13	16	4	5	6
14	17	7	8	9
15	17	10	11	0
16	18	12	13	0
17	18	14	15	0
18	0	16	17	0

Table 3

The following algorithm (algorithm COD1) will take as input a parent-child relationship table and output the corresponding sequence <a>. The most important field here is the

parent field as it is the one against which comparisons are made (to determine whether nodes are siblings i.e.: if nodes have the same parent, they are siblings).

The COD1 procedure is as follows:

```
Procedure COD1
current ←parent[1];                 -- the parent to compare with
i ←1;
k← 1;
j ← 0;
WHILE (parent[i] <> 0) DO           -- if parent[i]=0 then it is the root
{       IF ( parent[i] = current)   -- if same parent then increase
        THEN j ← j+1;                  the child count by 1
        ELSE                        -- else we have a node with different parent
                { a[k]← j;          -- takes the value of # of children with same parent
                  j ←1;             -- reset the child counter
                  k ← k + 1;        -- increase the index of the sequence
                  }                 -- as soon as the parent changes
        current ←parent[i];         -- the different parent becomes
        i ← i + 1;                  -- the new one to compare with
}

FOR i ← 1 TO k DO
output a[k]
```

When the COD1 algorithm was applied to the previous table (Table 3), it gave the following sequence:

3 3 3 2 2 2 2

As Procedure COD1 only requires scanning a column of n elements in a table and comparing each element to one value, then the time complexity for the COD1 algorithm is linear O(n).

Conclusion

B-Trees and their variants (not defined in this thesis, i.e.: Red-Black trees, B+-Trees etc...) are extremely useful when it comes to large storage of data. Also, because they have all the advantages of binary search trees, they simultaneously provide storage and relatively fast retrieval of data (hashing being faster). Such qualities make them interesting to research and study.and any contribution in the field of B-Trees is likely to make them even more useful tools.

The main original contributions were the new algorithm for the lexicographic generation of 2-3 trees, the proof in chapter 4 and the algorithms in chapter 5. We say original, because the new algorithms from chapter 3 were based on [GLW1] and simply improved or modified.

We defined a lexicographic order for B-Trees generation under which [GLW] and [GLW1] were indeed lexicographic and presented new algorithms ranging from lexicographic generation of 2-3 trees to decoding of B-Trees. We also proved that [GLW1] algorithm indeed generated B-Trees in constant average delay.

The algorithm presented in chapter 2 (G23T) is an improvement over [GLW] and the algorithms in chapter 3 showed that new algorithms based on both our lexicographic order and recursion could be designed to increase efficiency. The algorithms in chapter 5 are designed in a way that makes them perfect candidates for parallel algorithms.

Future work.

In this thesis, we worked mainly with the B-Tree of Bayer and McCreight, the most widely used file structure today; we did not extend the research to other B-Tree variants such as Red-Black trees, Prefix B-Trees , B+–Trees etc...even though we believe most of the work could be applied to those B-Tree variants. We shall consider extending the work to them in future papers. Also, the conversion from sequential to parallel algorithm for decoding will be the topic of a separate paper (to be published in mid 1996).

Open Problems.

The problem that can be considered, based on the chapter on B-Trees is to find an algorithm that generates B-trees in Gray Code order (i.e. minimal change order). Alternatively, it would be interesting to derive an algorithm that generates B-trees (or merely corresponding sequences) which will have constant time delay in the worst case. It also remains an open problem to generate a parallel algorithm for generating B-trees which will satisfy some desirable properties such as lexicographic order and constant average delay. Such algorithms exist for the case of binary and t-ary trees [AS].

The work on coding and decoding leads to an interesting open problem. It consists of deriving an algorithm that given the Parent-Child relationship array, will be able to insert and delete elements from a B-Tree without having to go through all the process of having to split nodes and change the pointers when a node is split into two or a node is having a new element inserted. That would be extremely efficient to retrieve, insert or delete nodes by simply changing the number of nodes and finding exactly which element of the

sequence will have to be modified in order to accommodate such changes.

It also remains an open problem to design a parallel algorithm for coding the input table to find the <a>sequence to improve on the linear complexity of the sequential algorithm.

Bibliography & References

[AHU] A.V. Aho, J.E. Hopcroft and J.D Ulman, *The Design and Analysis of computer Algorithms* (Addison-Wesley, Reading, MA, 1976).

[AS] S.G. Akl , I. Stojmenovic, "Generating binary trees in parallel", in *: Proc Allerton Conf. on Communications, Control and computing* (1992), to appear

[BC] Bing Feng and Gen-Huey Chen, "Cost optimal Parallel Algorithms for Constructing 2-3 Trees", *Journal of Parallel and Distributed Computing*

[BH] T.Beyer and S.M Hedetniemi, "Constant time generation of rooted trees", *SIAM J. Comput* 9(4) (1980) 706-712

[BM] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", *Acta Inform 1* (1972) 173-189

[D] L.E.Dickson, "History of the theory of numbers", *Vol. II, Diophantine Analysis*, Chelsea Publishing Co., New York, 1971.

[E6] M.C Er, "A simple algorithm for generating non regular trees in lexicographic order", *Comput. Journal.* 31 (1988) 61-64

[GLW] U. Gupta, D. Lee and C.K Wong, "Ranking and unranking of 2-3 Trees", *Society for Industrial and Applied mathematics*, 1982, p582-590

[GLW1] U. Gupta, D. Lee and C.K Wong, "Ranking and unranking of B-Trees", *Journal of algorithms 4*, 1983, p51-60

[Hik] T. Hikita , "Listing and counting sub-trees of equal size of a binary tree", *Inform . Process. Lett. 17* (1983) 225-229.

[Kelsen] P. Kelsen, "Ranking and unranking of trees using regular reductions", *Computer*

Science M.Sc. Thesis, University of Illinois at Urbana-Champaign, 1989.

[Kn] D.E Knuth, "Sorting and searching", *The art of computer programming Vol.3:* (Addison-Wesley, Reading, Ma,1973).

[LLW] C.C. Lee, D.T. Lee and C.K. Wong, "Generating binary treees of bounded heights" , *Acta Inform. 23 (*1986) 529-544

[Li] L. Li, "Ranking and unraning of AVL trees", *SIAM J. Comput, 15* (1986) 1025-1035.

[MBW] Mark B. Wells, *Elements of Combinatorial Computing* (Benjamin-Cummings, Ca, 1992).

[Mk] J.K.S. McKay, *Map of partitions into integers*, Alg.264, CACM, 8, 1965, p493

[MAK] Mark Allen Weiss, *Data Structures and Algorithm Analysis*, p130-137

[Pa] J.M. Pallo, "Generating trees with n nodes and m leaves", *Internat. J. Comput*, MAth. 21 (1987) 133-144.

[Pa1] J.M. Pallo, "Lexicographic generation of binary unordered trees", *Pattern Recognition Lett. 10* (1989) 217-221.

[RND] E.M. Reingold, J. Nievergelt, N. Deo, *Combinatorial algorithms*, (Prentice-Hall , N.J. Englewood Cliffs (1977) 179-182.

[Ru1] F.Ruskey, "Generating t-ary trees lexicographically", *SIAM J. Comput. 7* (1978) 492-509.

[RH] F.Ruskey and T.C. Hu, "Generating binary trees lexicographically", *SIAM J. Comput. 6* (1977) 745-758.

[VB] D.R. Van Baronaigien, "A loopless algorithm for generating binary trees sequences". *Inform. Process. Lett 39* (1991) 189-194.

[W] S.G. Williamson, "On the ordering ranking and random generation of basic

combinatorial sets", *Lecture Notes in mathematics 579* (Springer, Berlin, 1976).

[WROM] R.A. Wright, B. Richmond, A. Odlyzko and B.D. McKay, "Constant time generation of free trees", *SIAM J. Comput. 15* (1986), p 540-548.

[WCY] F. Wang, G. Chen and M.S. Yu, "Cost-Optimal Parallel algorithms for Constructing 2-3 Trees", *Journal of parallel and distributed computing 11*, 257-261 (1991)

[WCY1] F. Wang, G. Chen and M.S. Yu, "Cost-Optimal Parallel algorithms for Constructing B-Trees", *International Conference on parallel Processing*, 1991

[WS] W. Skarbek, "Generating ordered trees", *Theoret. Comput. Sci.* 57 (1988) 153-159.

[Z] A. Zoghbi, "Algorithms for generating integer partitions", *Master's thesis*, Univ. of Ottawa, 1993.

[Za] Zacks, "Generation and ranking of k-ary trees", *Inform. Process. Lett. 14* (1982) 44-48.

Appendix A: Programs

Program G23T;

```
{
 This program generates all combinations of 2-3 trees given
 a number n of leaves.
}
uses dos,crt;

type
aray=array[-2..20] of integer;

var
l1,a1,n,z,y:integer;
k,i,j,index:integer;
a,p,q,l,m:aray;
tp:boolean;
total:real;

procedure lfirst(var I:integer);
var ver:integer;
begin
 m[i]:=l[i]-l[i-1];
 q[i]:=(l[i-1] - l[i-2] -2*(l[i]-l[i-1]));
 p[i]:=m[i] - q[i];
 ver:=l[i-1]+p[i];
     for j:=(l[i-1] + 1) to  ver do
     begin
     a[j]:=2;
     end;
    for j:=(ver + 1) to (l[i] ) do
    begin
    a[j]:=3;
    end;
end;

procedure lexfirst( var i:integer);
begin
 repeat
  i:=i+1;
  l[i]:=round( (l[i-1] - l[i-2])/3  );
  if ( (l[i-1] - l[i-2])/3 > l[i]) then
     begin
     l[i]:=l[i] +1;
     end;
  l[i]:=l[i]+ l[i-1];{add li-1}
  lfirst(i);
 until (l[i]-l[i-1] = 1);
end;

procedure one1;
begin
for i:=-1 to 20 do
 begin
```

```
  a[i]:=0;  l[i]:=0;
  m[i]:=0;   p[i]:=0;
  q[i]:=0;   end;
end;

procedure newsequence;
begin
  index:=1;
  repeat
     write(l[index],' ');
     for j:=1 + l[index-1] to l[index] do
     write(a[j],' ');
     index:=index+1;
  until (index > i);
end;

procedure start;
begin
clrscr;
one1;
i:=1;  l[0]:=0;  l[1]:=round(n/3);
if ((n/3) - l[1]) > 0 then l[1]:=l[1]+1;
l[-1]:=-n;      i:=1;
lfirst(i);      lexfirst(i);
end;

procedure two;
begin
     if m[i] < trunc( ( l[i-1] - l[i-2] ) / 2) then
     begin
     l[i]:=l[i] + 1;
     lfirst(i);
     lexfirst(i);
     tp:=true;
     end;
end;

procedure one;
begin
    tp:=true;
    a[j]:=3;
    for l1:=j+1 to z+j+1 do
      a[l1]:=2;
    for l1:=z+j+2 to l[i] do
      a[l1]:=3;
    lexfirst(i);
end;

begin
clrscr;
readln(n);
start;
 repeat
  newsequence;
   writeln;      {now to create next 2-3 tree}
```

```
  tp:=false;
  repeat
      i:=i-1;
      y:=0;
      z:=0;
      j:=l[i];
      while ((a[j]=2) and (j > l[i-1])) do
          begin
          z:=z+1;
          j:=j-1;
          end;
      while ((a[j]=3) and (j > l[i-1])) do
          begin
          y:=y+1;
          j:=j-1;
          end;
      if j>l[i-1] then {We found the turning point}
          one
          else
          two
  until (tp or (i=1));
until tp = false;
end.
```

```
program Coding_Decoding;
Uses crt;
const maximo=20;
{ This program will code and decode B-Trees }
{ given the number of leaves and the order  }
{ The output can be seen in a file called testree }
TYPE  {declaration of types used in this program}

    Info= Record
    Num, Parent: longint;
    Child : Array[0..3] of longint;
    End;  {RECORD}

    Node1=array[1..20] of Info;

VAR  {declaration of variables in this program}
Order,I,J,K,N,Long   : Longint;
Node              : Node1;
Sequence            : Array[1..maximo] of Longint;
Level             : Array[0..maximo] of Longint;
Seq               : String;
ADD, NumElts, Count  : Longint;
F                 : Text;
Number,new          : Integer;

Procedure Init(ord:longint);
{ ** Initializes all nodes to zero **  }
begin
for i:=0 to maximo do
begin
Node[i].num:=0;   Node[i].Parent:=0; sequence[i]:=0;
for j:=0 to ord do Node[i].child[j]:=0;
end;
end;

Procedure convert(seq:string);
{ ** Converts a string of characters into an array of integers **
}
begin
for i:=1 to long do
sequence[i]:=ord(seq[i])-48;
end;

Procedure look;
{  **  Displays the table Parent-Children on the screen  **}
begin
writeln('Node# Parent  ch1  ch2  ch3');
for i:=1 to add+1 do
  begin
    write(Node[i].Num:3,Node[i].Parent:8);
    for j:=1 to 3 do write(Node[i].child[j]:5);
    writeln;
  end;
end;
```

```
Procedure look1;
{  **  Writes table Parent-Children to a file named f  **}
begin
assign(f,'testree');
rewrite(f);
writeln(f,'Order is: ',order,' and n=',n);
writeln(f,'Node# Parent  ch1  ch2  ch3');
for i:=1 to add+1 do
  begin
  write(f,Node[i].Num:3,Node[i].Parent:8);
  for j:=1 to 3 do
     write(f,Node[i].child[j]:5);
  writeln(f);
  end;
close(f);
end;

{ **********  MAIN PROGRAM  **********  }

Begin
Clrscr;
Order:=3;
Init(Order);
n:=7;   Order:=3;
writeln('enter n the number of nodes: 11 is optional for an example');
readln(n);
writeln('Please enter sequence of ',n,' numbers:  <<3332222>> for an example');
readln(seq);
long:=length(seq);
convert(seq);
NumElts:=0;
for i:=1 to long do
NumElts:=NumElts + sequence[i];
Add:=0;
for i:=1 to long do
 BEGIN
 for j:=1 to sequence[i] do
 begin
 Node[j+add].num:=j+add;
 Node[j+add].parent:=N+i;
 END;
 add:=add+sequence[i];
end;
count:=1;
for i:=n+1 to NumElts+1 do
   for j:=1 to sequence[i-n] do
      begin
         Node[i].Child[j]:=count;
       count:=count+1;
      end;
Node[Numelts+1].num:=Numelts+1;
Look1; Look;
Writeln('Please press enter to go back');
Readln;
new:=0;
```

```
i:=0;
REPEAT
 number:=0;
    REPEAT
    i:=i+1;
    IF node[i].parent = node[i+1].parent
    THEN number:=number+1 ELSE
      begin
      number:=number+1;
      new:=new+1;
      end;
    UNTIL (node[i].parent <> node[i+1].parent);
sequence[new]:=number;
UNTIL(i>=numelts);
for i:=1 to new do
write(' ',sequence[i]);
end.
```

Program permute;

```
{ This program will give all unique combinations with repetitions of n numbers in an increasing
  order }
uses crt;
const max=4;
type aray=array[0..max] of integer;
var z        : aray;
l,k,i,j,n    : integer;
counter      : integer;
bool         : boolean;

procedure swap(var a,b:integer);
var c:integer;
begin
c:=a;
a:=b;
b:=c;
end;

procedure print(var p:aray);
var j:integer;
begin
 for j:=1 to max do write(p[j],' ');
  counter:=counter+1;
 writeln;
end;

begin
clrscr; counter:=0; z[0]:=0;
for i:=1 to max do readln(z[i]);
i:=1;
while(i<>0) do
 begin
 bool:=false;
 print(z);

 i:=max-1;
 while (z[i]>=z[i+1] ) do i:=i-1;
 j:=max;
 while (z[i]>=z[j])   do j:=j-1;
 swap(z[i],z[j]);
 k:=max;
 l:=i+1;
 while(k>l) do
 begin
   swap(z[k],z[l]);
   k:=k-1;
   l:=l+1;
 end;
 end;
writeln(counter);

end.
```