Master Thesis
Computer Science
Thesis no: MCS-2009-23
June 2009

An Optimized Representation for Dynamic *k*-ary Cardinal Trees

Author: Venkata Sudheer Kumar Reddy Yasam

This thesis is submitted to the Department of Interaction and System Design, School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Venkata Sudheer Kumar Reddy Yasam
E-mail: sudheergid@gmail.com

External advisor(s):
Dr. Srinivasarao Satti
Aarhus University
E-mail: ssrao@cs.au.dk, ssrao@cse.snu.ac.kr, ssrao@gmail.com
Phone:
+45 89425748

University advisor(s):
Dr. Stefan J. Johansson
Department of Systems and Software Engineering
E-mail: stefan.johansson@bth.se

School of Computing
Blekinge Institute of Technology
Soft Center
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 102 45

ABSTRACT

Trees are one of the most fundamental structures in computer science. Standard pointer-based representations consume a significant amount of space while only supporting a small set of navigational operations. Succinct data structures have been developed to overcome these difficulties. A succinct data structure for an object from a given class of objects occupies space close to the information-theoretic lower-bound for representing an object from the class, while supporting the required operations on the object efficiently. In this thesis we consider representing trees succinctly. Various succinct representations have been designed for representing different classes of trees, namely, ordinal trees, cardinal trees and labelled trees. Barring a few, most of these representations are static in that they do not support inserting and deleting nodes. We consider succinct representations for cardinal trees that also support updates (insertions and deletions), i.e., dynamic cardinal trees.

A cardinal tree of degree *k*, also referred to as a *k*-ary cardinal tree or simply a *k*-ary tree is a tree where each node has place for up to *k* children with labels from 1 to *k*. The information-theoretic lower bound for representing a *k*-ary cardinal tree on n nodes is roughly $(2n + n\log k)$ bits. Representations that take $\left(2n + n\log k + o(n\log k)\right)$ bits have been designed that support basic navigations operations like finding the parent, *i*-th child, child-labeled *j*, size of a subtree etc. in constant time. But these could not support updates efficiently. The only known succinct dynamic representation was given by Diego [42], who gave a structure that still uses $\left(2n + n\log k + o(n\log k)\right)$ bits and supports basic navigational operations in $O\big((\log k + \log\log n)\big)$ time, and updates in $O((\log k + \log\log n)\left(1 + \frac{\log k}{\log(\log k + \log\log n)}\right))$ amortized time. We improve the times for the operations without increasing the space complexity, for the case when k is reasonably small compared to n. In particular, when $k = (O(\sqrt{\log n}))$ our representation supports all the navigational operations in constant time while supporting updates in $O\left(\sqrt{\log\log n}\right)$ amortized time.

Keywords: Succinct data structures, binary trees, ordinal trees, cardinal trees, rank, select, static and dynamic.

Contents

List of Figures

Chapter 2

Chapter 3

Chapter 4

Acknowledgements

First and foremost, I would like to express my sincere gratitude towards my thesis supervisors Stefan J. Johansson and Srinivasa Rao Satti for their patient guidance, invaluable comments, and support at every stage of my Master Thesis. All the work presented in this thesis was possible only because of the close and fruitful co-operation from my supervisors. Without their help and encouragement this thesis would never have been completed. My great thanks to them, not only for answering tones of my infinite questions, but also for guiding me with invaluable support.

Many thanks to my thesis examiner at BTH, Prof: Dr. Guohua Bai for his continuous guidance and for commenting the report.

My loving thanks to all of my family members. They have provided continuous encouragement and support during my work. Special thanks to my sister for many great memories and still many more to come.

Most of all, I would like to express my warmest possible thanks by dedicating this thesis to my sweet heart and my wife Mrs. Madhuri Ravipati for supporting me in such an excellent and unselfish way in harder times, especially when I was sick, as well as for sharing all the joy in the good times. I would never have been able to carry this out without you!

Finally, I would like to express my special thanks to May-Louise Andersson and the other BTH staff and the study environment for being very cooperative and for their continuous support.

1 INTRODUCTION

This chapter describes the significance, existence and the pitfalls of the conventional pointer based data structures. An aroma of succinct trees explains the problem and the research gap. It also describes how the main aim envisions the essential objectives, research questions and research methodologies.

1.1 Preface

Radically, ***computer science*** is a vast span of discipline. It is the study of identifying and solving problems mechanically through methodical processes or algorithms [33, 34]. Computer science consists of several areas. In this thesis mainly we investigate on theoretical computer science or informatics, which deals with complexity of computation problems and different models of computation. The computer memory is indispensable to store the information in the form of bits. In majority of applications the capability of storing and accessing massive amount of data plays a crucial role. ***Data structure*** is a method of storing the data in computer's memory [33-39]. Algorithms and data structures are essential parts in informatics to store and access the data efficiently. An ***algorithm*** consists of a set of rules or instructions or processes to solve the problems [34-38]. It is an initial step for solving the problems. An algorithm directs the data to store and access in the computer's memory based on the set of instructions in it.

Traditionally, the information is stored in the data structures as of arrays, linked lists, stacks, queues, double ended queues, trees, and graphs. Eventually, the phenomena of stored data should be accessible and able to perform several operations on it. In the earlier days, the data storage methods are inefficient in terms of storing and retrieving the data. The data redundancy is imperative in all real world applications and the storage cost is becoming a critical factor. For instance, data compression and data optimization are two essential methods in informatics to resolve the data redundancy. Data ***compression*** is the way of storing the data in minimum number of bits [40]. It reduces the data storage time as well as accessing time. Claude E. Shannon [40] was the father of data compression theory and information theory. In 1948, he formulated fundamental data compression techniques such as lossless compression and lossy compression in his paper "*A Mathematical Theory of Communication*" [40]. ***Lossless compression*** is a method of compression which results the exact original data into minimum amount of compression [40]. This way of compression is mainly used where the exact replication of data is essential such as data base records, spread sheets, word processing files...etc. ***WinZip*** program is the best known example for loss less compression. Shannon [40] mathematically proved entropy rate ***H*** in lossless data compression which is possible to compress the information source with effective compress rate is very close to ***H***. ***Lossy compression*** is another method of compression which generates certain loss of accuracy from the original data and results substantial to the original data but not exactly [40]. This compression method is very effective for compress digital voices, graphical images, digital videos...etc. Shannon [40] developed ***rate-distortion theory*** in lossy data compression, which showed that the amount of ***distortion D*** is tolerated with a given source and a given distortion measure, then the ***rate distortion*** function ***R(D)*** will become a best distortion method. That means Shannon [40] mathematically proved that the rate distortion function ***R(D)*** is the best possible compression rate and it is highly impossible to prove better than ***R(D)***.

In other hand, Data ***optimization*** is a process of supporting several operations on compressed data rapidly and efficiently, by consuming little bit of extra space while retaining from the memory [1, 2]. Data optimization is organized in two types [1]. They are ***concrete*** optimization and ***abstract*** optimization. ***Concrete*** optimization is a predefined representation of data structure to store the data within the least possible space [1]. In the concrete

optimization all required operations are not in developer's control, because they are predefined by optimizer. Here, ***optimizer*** refers to the designer of the optimizing compiler [1]. ***Abstract*** optimization is on the other hand, an optimizer gives some control on primitive data types and the developer can design a new format of data structure efficiently [1]. In this procedure the developer can access the data and able to perform several operations on it within the minimal space and time.

This thesis mainly focused on the abstract optimization, which comes under static data types such as trees. A ***tree*** is a hierarchical representation of collection of items or data sets in the computer's memory [1 - 3, 5, 6, 8 - 12, 14, 19, 20, 27, 29, 30, 32]. Trees are the best known data structure for storing massive amount of data such as genealogical information, astronomical data, DNA sequences and many more. Trees are many types and their detailed description is discussed in further chapters. Generally, trees contain several types of nodes such as parent, left child, right child, siblings and few more. Traditionally, trees are stored in the data structure using pointers which are linked among the nodes. Each node contains memory and it does not determine any fixed dimension or size or layout. Each node connects with one or more pointers along with their data. In pointers it is very easy to store other information within the pointer, but moving the data from one to another node is quite fuzzy, which is possible by dereferencing the pointer. The addresses of pointers are free from memory. The operations like inserting and deleting are possible by dereferencing the pointers. The procedure of dereferencing from one node to another node is possible, but the program is not allowed to embed the numerical values to the pointers. In this conventional method each node occupies a distinct block of memory. These disadvantages are affected to the optimization.

Mathematically, a tree structure of n nodes required at least $O(n)$ pointers and each pointer required at least n distinct memory locations to store their addresses. Usually, for addressing these enormous memory locations occupy $\log n$ bits per pointer. That means the standard method of pointers of n node tree required $O(n \log n)$ bits of memory [1, 2]. The dereferencing procedure is required to support other operations like searching or deleting, which does change few pointers. In this procedure each node has to be visit even once. In the worst case an average of $N/2$ nodes required to visit and it takes $O(N)$ time (in this "$\log_2 n$" denotes as logarithm of n base 2**)**. This wastage of memory and time are quite unacceptable in the data structures. To resolve these problems Jacobson [1] invented a special form of data structures. They are simply called as ***succinct*** or space efficient data structures.

Since the number of distinct binary trees with n nodes is only $C_n = \binom{2n}{n}/(n+1)$, the information-theoretic lower bound is only $\log C_n = 2n - O(\log n)$ bits [41]. Here C_n is the ***Catalan*** number. In fact there are various succinct binary tree representations that use only $2n + o(n)$ bits and support a wide range of operations [1 - 4, 6, 8 - 12, 14, 16, 21, 23, 29, 30 - 33]. The binary tree representation of Jacobson [1, 6] uses $2n + o(n)$ bits of space, and supports parent, left child and right child operations in the constant time. Later Munro and Raman [3, 13, 18], and others gave representations that use same space, and support some additional operations like sub tree size, least common ancestor of two nodes, level-ancestor of a node etc. All these representations are static which means one cannot efficiently update the tree structure. For binary trees, Munro et al. [33] gave a representation that supports insertions and deletions of nodes efficiently, while still supporting the operations in constant time. Raman and Rao [19] improved the update times of this structure. But for *k*-ary trees (where each node has *k* slots, labeled 1.., *k*) no dynamic succinct representation exists. In this thesis we implement a method for dynamic *k*-ary trees which supports queries and updates efficiently in constant time while using close to optimal space.

1.2 Problem domain

The research discipline is succinct (or space efficient) data structures. It has applications in representing tries, suffix trees which are widely used in various text processing/compression algorithms, and in bioinformatics. In the above section 1.1 envisioned the existence of conventional pointer based data structures and the essential significance of succinct data structures. This research discipline of succinct data structures is first implemented by Guy Jacobson [1] in his Ph.D thesis at Carnegie Mellon University, 1989. This area of research is quiet reluctant, because from the past two decades several researchers have implemented quiet few methods such as balanced parenthesis [8], level order unary degree sequence (LOUDS) [9], depth first unary degree sequence (DFUDS) [10] and *k*-ary trees [11]. Furthermore, the practical implementation is done only on LOUDS. There is no detailed description and literature is available in this area of succinct data structures. Here, we utilize this opportunity to describe extensive enlightenment, implementing dynamic *k*-ary trees with in the optimal space and time.

1.3 Problem and Research Gap

The non trivial suffix trees of balanced parentheses method were first implemented by Jacobson [1, 2, 6], which is required $2n + o(n)$ of linear space within the constant time. Later Munro and Raman [3, 14, 18], and Geary and Raman [8, 10] proposed level order unary degree sequence (LOUDS) and depth first unary degree sequence (DFUDS) respectively for the same linear space and less amount of time by supporting some additional tree traversal operations. The area of ***cardinal*** trees is not well studied. Heretofore, a ***cardinal k-ary tree,*** where each node has ***k*** slots and labeled from 1….*k*, and it is also known as a multi-way tree with the maximum of ***k*** children. An n node binary tree with ***k*** degree is represented as $C_n^k = \frac{\binom{kn+1}{n}}{(kn+1)}$ distinct cardinal trees [41]. Simply applying log on both sides we get $\log C_n^k = (k \log k - (k-1)\log(k-1))n - O(\log(kn))$ bits required to store ***k-ary*** cardinal tree in a theoretic lower bound. David et al. [11] given a structure for ***k-ary*** trees, which supports parent, child, label and subtree size in constant time as $(\lceil \log k \rceil + \lceil \log e \rceil)n + o(n) + (\log \log k)$ bits. But all these methods are ***static*** that means they only support accessing the data, with derived linear space and constant time. But in this thesis we implement a method for dynamic ***k*-ary** trees which supports updating operations within the optimal space and time. This significant method is enough to reinforce the research gap efficiently and effectively.

1.4 Aims and Objectives

The main objective of this thesis is to implement an optimized dynamic *k*-ary cardinal tree structure by supporting all the basic navigational operations are in constant time, and update, insert and delete operations are in $O(\sqrt{log\ log\ n})$ amortized time. To achieve this we discuss the following aims.

- The critical tradeoffs in traditional pointer based data structures
- Analyzing the succinct data structures and their needs and advantages
- Analyzing the navigational operations in succinct data structures and their effect on time and space bounds
- Analyzing several kinds of trees which are available for succinct data structures
- Understanding the available tree representation methods and procedures in succinct data structures

- Finding the tradeoffs in existed representations including with their supporting operations
- Understanding the research gap between the static and dynamic trees
- Implementing an optimized dynamic tree for cardinal trees

1.5 Research Questions

The main research question of this thesis is:

How can we implement an optimized dynamic *k*-ary cardinal tree?

In order to achieve this main question the following questions are also answered.

RQ1. What are the challenges and existing problems in dynamic cardinal trees?

RQ2. What are the challenges faced while implanting the dynamic cardinal trees?

RQ3. How can we overcome the problems and challenges (RQ1 and RQ2) while implementation?

1.6 Research Methodology

In order to answer the research questions firstly, we conducted a literature review which comes under quantitative procedures [46, 47] through research publications, journals, books and conference proceedings. This extensive knowledge is used to answer RQ1in the upcoming section 4.1.

Secondly, we used qualitative procedures [46, 47] by focusing on required concepts through observations, investigations and examined by applying several parameters on available operations. It resulted to answer RQ2 in section 4.5.

Thirdly, we used both methods of qualitative and quantitative methods [46, 47] and answered RQ3 in the sections 4.1 to 4.5. In this thesis we used mainly both methods, so it is comes under mixed method approach.

1.7 Structure of the Thesis

Chapter 1, explains introduction of the study; Chapter 2, explains a comprehensive study of succinct data structures and their supporting operations with examples; Chapter 3, exemplifies several types of trees and their supporting operations; Chapter 4, explains the implementation of an optimized dynamic *k*- ary cardinal tree.

2 Succinct Dictionaries

The proliferation of data storage and its cost is dominating in all levels of real world applications. Data compression and data optimization techniques are occupied at most of crucial stages in data structures. This amplification boosts to minimize extensive space, rapid data storage, accurate accessing and reallocate the data in all levels of hierarchy. To achieve this, Succinct or space efficient data structures are absolute. For instance, *Rank* and *Select* dictionaries are basic tools to resolve some critical operations in succinct data structures. In this chapter we exhibit an extensive research work on *Rank* and *Select* dictionaries.

2.1 Rank and Select Functions

Ordered sets contain elements which are particularly in order. They are the best example of static data types. Let us assume that "S" is a non empty subset of *m* positive integers then the subset denotes as S= {1, 2,.....m}, where m < n. The non empty subset of S occupies $\log\binom{n}{m}$ bits to store in a computer's memory, which is relatively $m\log\left(\frac{n}{m}\right)$ by Stirling's Approximation.

Jacobson [1] defined the following two basic operations on a bit vector, which form the basic building blocks in various succinct data structures. They are shown in the figure 2.1:

$rank_{\propto}(i)$: Counts the number of α's in the non empty subset of *S* including the current position up to the given position *i*.
$select_{\propto}(i)$: Finds i^{th} position of α^{th} element in the given non empty subset of *S*.

Figure 2.1: Rank and Select functions, where α denotes 0 or 1.

2.2 Rank and Select functions over a Bit String

Generally, computers store memory in the form of bits. These types of consecutive bits appear like static strings. Simply, we apply *rank* and *select* operations on a static bit string instead of a non empty subset. In later chapters we store trees in static bit strings. Let us take a simple bit vector which is shown in the Figure 2.2 with *n* number of blocks where *n* is 20, and we compute *rank* and *select* operations. Firstly, we apply the basic operations of *rank* and *select* on the following bit string.

	1	2	3	4	5	6	7	8	9	10	11
B=	1	1	0	1	0	0	1	0	1	1	0	0	0	1	1	1	0	0	1	1
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Figure 2.2: Static bit string

$rank_1(x)$ = Counts the number of 1's up to the required element *x* in the given bit string **B**.

$rank_0(x)$ = Counts the number of 0's up to the required element *x* in the given bit string **B**.

$select_1(x)$ = Counts 1^{th} position of *x* in the given bit string of **B**.

$select_0(x)$ = Counts 0^{th} position of *x* in the given bit string of **B**.

Example: 2.1:

Now we compute basic operations in the following example.

$rank_1(12)$ = 6 (Number of 1's up to the position of 12)

$rank_0\ (12)$ = 6 (Number of 0's up to the position of 12)

$select_1\ (5)$ = 9 (Position of the 5th 1)

$select_0\ (5)$ = 11 (Position of the 5th 0)

Amusingly, here *rank* and *select* operations are inverse each other. Let us observe that *rank* (*select* (*x*)) = *x*, and *select* (*rank* (*x*)) = *x*, where $1 \leq x \leq \|\mathbf{B}\|$, for $x \in \mathbf{B}$.

limit_count(*x, y*): counts the number of entities between specified intervals of *x* and *y*.

limit_count (*x, y*) = *rank* (*y*) - *rank* (*x-1*)

next (*x*): counts the smallest entity in **B** greater than *x*.

next(*x*) = *select*(*rank*(*x*) + 1).

previous (*x*): counts the largest entity in **B** and smaller than *x*.

previous (*x*) = *select*(*rank*(*x-1*)).

pass(*x, z*): prints the entity in **B's** sorted list, which appears *z* positions next to *x*.

pass(*x, z*) = *select*(*rank*(*x*) + *z*).

Figure 2.3: Supporting operations based on *rank* and *select* functions

For example $rank_1(select_1(5))$ = 5 (observe that $select_1(5)$ = 9 and $rank_1(9)$ = 5). In the same sense $select_1(rank_1(14))$ = 14 (observe that $rank_1(14)$ = 7 and $select_1(7)$ = 14).

Based on these two operations, we can perform some more additional operations. They are *limit_count*, *next*, *previous* and *pass*. Let us assume that *x, y* ≤ *n*, where *z* ∈ **B**. These operations are shown in the figure 2.3.

Example: 2.2:

Now we compute additional operations for a given bit string **B** shown in figure 2.2. Remember that we are performing the operations for 1's entropy.

(i) *limit_count*(5,*16*): counts the number of entities between specified intervals of 5 and 16.

limit_count(5, 16) = *rank* (16) - *rank* (5-*1*)

Which is *limit_count(5, 16)* = *rank* (16) - *rank* (4)

That is *limit_count(5, 16)* = 9 – 3 = 6.

(For a proof let us count number of 1's between the blocks 5 and 16 shown in the figure 2.2, which is 6).

(ii) *next* (12): counts the smallest entity in **B** greater than 12.

next(12) = *select*(*rank*(12) + 1)

Which is *next*(12) = *select*(*rank*(12) + 1)

That is *next*(12) = *select*(6+1) (Here we have to calculate *select*(7))

So *next*(12) = 14

(For a proof let us examine that the 1 is available at block 14 after the given block of 12).

(iii) *previous* (15): counts the largest entity in **B** and smaller than 15.

previous (15) = *select*(*rank*(15-*1*))

Which is *previous* (15) = *select*(*rank*(14))

That is *previous* (15) = *select*(*7*) = 14

(Let us observe that 14 is the previous block of 15th block)

(iv) *pass*(*6, 5*): prints the entity in **B's** sorted list, which appears 5 positions next to 6.

Which is *pass*(*6,5*) = *select*(*rank*(6) + 5)

That is *pass*(6,*5*) = *select*(8) = 15

(Let us observe that from 6th block to 15th block, the number of 1's are 5, which was our given entity to pass between blocks)

In this example we used a bit string of length *n* is 20 blocks, which we cannot minimize the space. But we need to perform some operations which are discussed above and they required some additional space. Let us assume that if each block of length is $\log n$ and if we need the rank of 20th block then the linear scanning procedure is required for all 20 blocks of scanning. It almost takes $n * \log n$ space and *n* time. If length of the bit string is huge then the traversal time is unacceptable. Because length of the bit string is *n*, and then it required $O(n)$ bits of space and $O(\log n)$ time to perform *rank* and *select* operations. This method is quite chaotic.

Let us assume that if we divide each block of length $\frac{1}{2}\log n$ then we get $\frac{n}{\frac{1}{2}\log n} = 2n/\log n$ blocks, and to store *n* number of cumulative 1's then the bit string is required $O(\sqrt{n}\log n \log\log n)$ bits that is $\theta(n)$ bits. Now we discuss how to overcome these tradeoffs by using two levels directory schemes in the following section.

2.3 Two Levels Directory Scheme for Ranking

Ironically, Jacobson [1] gave an auxiliary structure which is more efficient in terms of space and time. We have seen the tradeoffs between time and space over a bit string from the above section. Jacobson [1] gave a profound auxiliary structure which is based on two levels scheme in order to access *rank* and *select* functions within the constant time.

- Notionally split the bit vector fragment into blocks of size "**b",** which is first level directory**.**

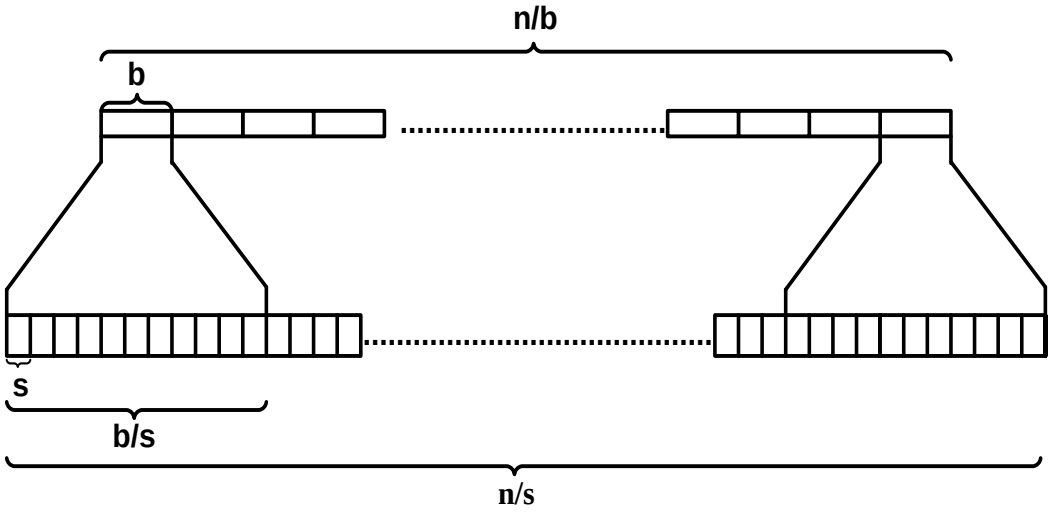

Figure 2.4: A Two-Level Directory scheme for compute *rank* and *select*

- Now split each "**b"** block into sub block size of "**s**" in second level directory. Then the block "**b"** contains "**b/s"** sub blocks**,** because we sub-divided each block.
- Let us assume that we have "1 ...n" number of blocks in the first level directory then the whole fragment in the first level directory contains "**n/b**" blocks. The second level directory contains "**n/s**" sub blocks.
- Now we compute the *rank* of each "**b"** block in the first level directory and store in a bit string.
- In the second level directory, we also compute *rank* of each sub block "**s**". In the last cell of each sub block we store null because we already stored ranks of each block in first level directory.
- We compute the *rank* of element x by finding the location of particular block of *x,* which is $\lfloor x/b \rfloor$. We already computed the *ranks* of each "**b"** and **"s"** blocks from the first and second level directories. If $x \leq (\lfloor x/b \rfloor * b)$ is false then we conduct a linear search from the $(\left(\left\lfloor \frac{x}{b} \right\rfloor * b\right) + 1)$ bit to x^{th} bit. Once we find the x^{th} element then we calculate the *rank* from $\left(\left\lfloor \frac{x}{b} \right\rfloor * b\right) + 1)$ bit to x^{th} bit by using both first and second level directories, and we add both *ranks* to get the desired value of *x.*

Theoretically, let us assume that we have *n* number of blocks in the first level directory, which are split into each block of length $\lceil \log^2 n \rceil$ (i.e. **b**). Then it takes $O(n/\log n)$ bits of space. Now store the *ranks* of each block in the first level directory. We split each block into sub blocks of size $\lceil \log n \rceil$ (i.e. **s**), then it takes $O\left(\frac{n}{(\log n)^2} \log n\right) = O(n/\log n)$ bits of space which is $o(n)$ bits. Here also we compute *ranks* of each sub block and store in the second level directory. Now we compute the time and space for *n* number of blocks. In the worst case we need *n* bits to store *n* blocks. Addition to this for computing additional operations like *rank* and *select* we need some extra space. If there are $O(\sqrt{n})$ distinct blocks and $O(\log n)$ distinct sub blocks then this structure takes $O(n \log\log n / \log n)$ bits, which is $o(n)$ bits of extra space. So, the total structure is required $n + o(n)$ bits of space and $O(\log n)$ time, which is significant improvement with the two levels directory scheme.

Now we compute *rank* and *select* operations over a bit string using two levels directory scheme. Let's take a bit vector **B** = 1101010101010001111001110010101011101100001

11101, where *n* is the number of elements in **B,** which length is 48 bits**.** Now we split the bit vector "**B"** into $\log^2 n$ bits per block, where $1 \leq x \leq \lceil \log^2 n \rceil$, which is first level directory shown in the figure 2.5. Again we split $\log^2 n$ blocks into log*n* bits which are shown in second level directory. Let us assume that $\log^2 n = 16$, then $\log n = 4$. We split our bit string into the block length of $\log^2 n$ in the first level directory, which gave 3 consecutive blocks of each big block length is 16 bits.

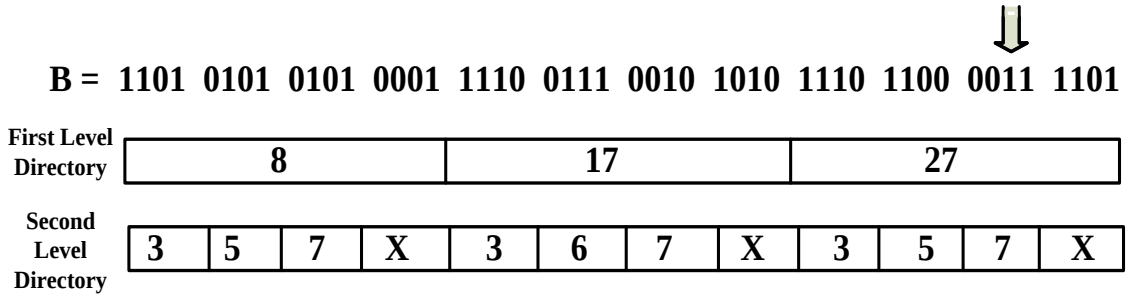

Figure 2.5: Two level directory of a bit vector B

Example: 2.3:

Firstly, we compute *ranks* of each big block in the first level directory. Secondly, we compute the *ranks* for sub divided blocks. Let us observe in the second level directory that each last bit contains null, because we use the *rank* from the first level directory. We took this example to compute the *rank* of 43rd bit.

- Fundamentally, we have to identify the accurate block number in the first level directory. In our example we have taken 16 as the size of each big block in the first level directory. To find the exact block for the required element we first calculate the block number like $\lfloor 43/16 \rfloor$ which gives the value as 2nd block. In the first level directory the *rank* shows up to second block is 17.
- Now we compute the *rank* between 33rd and 43rd bit. We already know that the size of each sub block is 4. So our desired bit is available in 3rd sub block. In the second level directory we already stored the *rank* of sub blocks. So the *rank* up to 40th bit is 5 and now we conduct a linear search to find the *rank* between 40th and 43rd bit, which is 1.
- Now we add up all the *ranks* obtained from the above steps they are 17+5+1=23.
- Let us observe from this example that the other operations like limit_count, next, pass and previous all are dependent on *rank* and *select* operations only. We compute *select* operation in the next section.
- Let us observe from Jacobson's [1] method we used 12 bits of extra space for a 48 bit string to store two levels directory scheme for supporting all the operations efficiently in $O(\log n)$ time.

Since the two levels scheme is successfully outdone and little efforts were tried to invent three levels directory schemes by several researchers. But unfortunately the outcomes were worst, because when we increase each level of directory by dividing each small block into tiny chunks then we must required vast number of bits to store their ranks. This was fundamentally gave very bad results in terms of space and time. Of course we have a successful method for *rank* by using three level directories which we discuss in the upcoming sections. Now we discuss *select* procedure by using two levels directory schemes in the following section.

2.4 Two Levels Directory Scheme for Selection

Select is a basic function to find the position of the given 0^{th} or 1^{th} bit in a given string. Practically, we discussed the *select* procedure in section 2.2. In that procedure, length of each block is $\log n$ then it required $O(\log n)$time for each operation and more additional space is required. Jacobson [1] gave a method which is based on two levels directory scheme, ranking directories and binary search. In this method if the length of block is $\log n$ then this procedure required $n(1 + O\left(\frac{\log\log n}{\log n}\right) = n(1 + o(1))$ [1-4, 7] space and $O(\log n)$ time. The detailed step by step procedure is as follows:

- Let us assume that ***x*** is the element to find the *select(x)* in the given bit string by using two levels directory scheme.
- We use binary search to find the appropriate position of the element ***x*** in the given bit string **B** which is shown in the figure: 2.4.
- In figure 2.4, we used "**b**" as size of each big block and "**s**" as size of each small block. We already knew that the *ranks* of each big block and sub blocks are available in first and second level directories. We find appropriate big block based on the *ranks* available in directories. Let us assume that first level directory contains *ranks* of each big block as $\{r_1, r_2, \ldots \ldots\ r_n\}$, such that $r_1 < r_2 \ldots < r_n$. In the same way second level directory contains *ranks* of sub block as $\{c_1, c_2, \ldots \ldots\ c_n\}$, such that $c_1 < c_2 \ldots < c_n$.
- Firstly, we use binary search by comparing the *ranks* available in first level directory with the required element of *x*. We obtain the appropriate block by finding the lower and upper bounds of *x*. Here we have two cases to find the particular block of *x*. Let us assume that if r_5 is a middle *rank* available in first level directory then first case is we compare like $x > r_5$, if the statement is true then it compares to its right side *ranks* until the statement is false. In second case, if the statements $x \leq r_5$ and $x > r_4$ are true then the required element is available in the r_5 block only, else we continue the comparison towards left side. If we do not have any option to satisfy the second ***if*** statement then the required element is available in the first block of If we have two middle blocks then we continue binary search with right side block.
- Secondly, we find a particular sub block of *x* by using binary search from the *ranks* of second level directory. Once we find the sub block of *x* then we search for particular position of *x.* The procedure is same as above but we find *x* from the second level directory. Once we find an exact position of *x* then the *select(x)* denotes the address of particular block of *x*.

Example: 2.4:

Now we find *select (14)* for the same bit string which is shown in figure: 2.5. Let us recall our block sizes from our example. There, we assumed the big block size is 16 and the sub block size is 4.

Firstly, let us find the middle big block from our first level directory in the figure: 2.5. In our example, the rank of the middle block is 17. Now compare the required element 14 with 17.

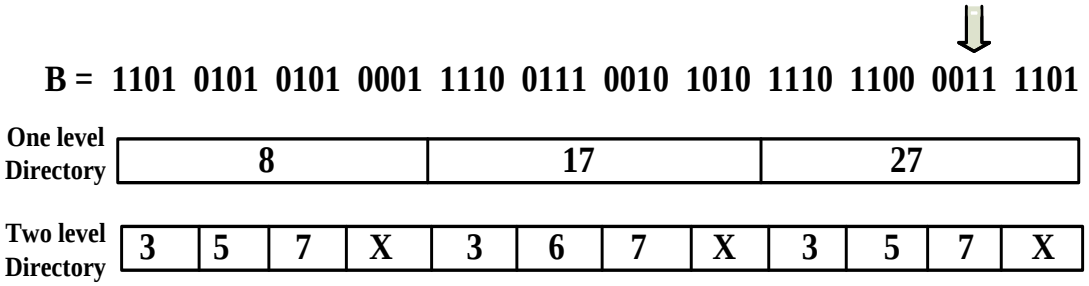

Figure: 2.6: Two level *rank* directory

Patterns	# of 1's
0000	0
0001	1
0010	1
0011	2
0100	1
0101	2
0110	2
0111	3
1000	1
1001	2
1010	2
1011	3
1100	2
1101	3
1110	3
1111	4

Figure 2.7: *rank* lookup table.

Patterns\i	1	2	3	4
0000	X	X	X	X
0001	X	X	X	4
0010	X	X	3	X
0011	X	X	3	4
0100	X	2	X	X
0101	X	2	X	4
0110	X	2	3	X
0111	X	2	3	4
1000	1	X	X	X
1001	1	X	X	4
1010	1	X	3	X
1011	1	X	3	4
1100	1	2	X	X
1101	1	2	X	4
1110	1	2	3	X
1111	1	2	3	4

Figure 2.8: Select lookup table

In our scenario, if the first case $14 > 17$ is true then we compare with right side *ranks*, but in this case the statement is false so we go to second case. In the second case we compare the *ranks* of two big blocks, they are 17 and 8. So, if the statements $14 \leq 17$ *and* $14 \geq 8$, both are true then we consider the biggest ranking possible block contains the required element of 14, that is *select(14)*. If any of the statement is false then we compare with previous *ranks* available in the first level directory.

- Secondly, we find an exact location of 14 by using binary search like comparing the middle *rank* of sub block from second level directory plus *rank* of previous big block from the first level directory. In this case, if $14 > (8 + 6)$ is true we go for right side ranks, but in our case the statement is wrong. So now we compare two statements like $14 \leq (8 + 6) and\ 14 > (8 + 3)$. Let us observe that both statements are true, so the required element *select(14)* is available in second sub block which rank is 6. Now we use linear search for the 14^{th} 1^{th} bit in the sub block and the *select(14)* is the address of the particular bit location that is 24.

In the above example we discussed the procedure to find the *select* of given element by using Jacobson's [1] method. We also discussed the tradeoffs between space and time. To resolve those tradeoffs Clark [2] proposed a method which is addition to Jacobson's [1] approach by adding lookup tables. Clark [2] achieved and proved some better performance than Jacobson [1] within the same amount of space and with constant time. We discuss the detailed procedure in the following section.

2.5 Rank and Select Functions Using Lookup Tables

Clark [2] proposed a method which is similar to Jacobson's [1] procedure which deals with implementing the rank and select lookup tables and they consume less amount of space in constant time. This procedure is used almost in every aspect of succinct data structures. We apply this procedure to our example for finding the *rank* and *select* functions. The figure 2.4 is again used as of our example and it shows the index of rankings for each of their blocks. Generally, the lookup table patterns are system default that means in our case each sub block of size is 4 bits. So the system initiates the binary values from 0000 to 1111, which are not required to specify explicitly like in our pre computed tables. They specified in figure: 2.7 for our better understandings only. In figure: 2.8, shows that positions of 1's in each pattern. In this Clark's [2] algorithm, the space occupied by the rank directories, rank lookup tables and select lookup tables which are very minimal as $o(n)$ bits.

Example 2.5:

Let us take an example to compute *rank* of 43^{rd} bit by using two level directories and lookup tables. Firstly, we implement two level directories as shown in the figure: 2.6, by dividing the x^{th} element by $\log^2 n$. In our case here *x* is 43, $\log^2 n$ is 16 and $\log n$ is 4. We split the bit vector into bit strings of length 16 in the first level directory, and again we split each big block into sub block size of $\log n$ in the second level directory which is shown in figure: 2.6. The figure: 2.7, shows that the sub block size of each bit pattern and the total number of 1's available in that each pattern of bits. The *ranks* of each sub blocks are pre computed based on the patterns available in figure 2.7.

Now we find *rank (43)* in a bit string shows in figure: 2.6. First we compute the *rank* of maximum possible sub block which *rank* is less than the required element. We obtain r*ank (40)* that is 22. The bit pattern which is available in sub block is 0011. Now we just need to compute exact position of *x* and its *rank* which is in between the bits of 41 and 43. To get exact number of 1's in this sub block we use bit wise AND operation with the inverse pattern of that particular sub block bits. In our example the inverse pattern for 0011 with respective of 3 bits that is 1110. Simply, use bit wise AND between 0011 and 1110 then we get 0010. So the number of 1's available in that particular sub block is 1. Now we add both ranks 22+1 then we get 23 as the *rank (43)*.

Now we compute the *select* of 20. Firstly, we record the address of each 1^{th} bit available in the bit string. We already knew that the *ranks* of each $\log^2 n$ block. Secondly, we compare the *ranks* of each big block with the required element's position. Obviously, we find *rank* of big block from the first level directory which is 17. Now we just required 3 positions of 1from the sub block which is available in second level directory. We use rank lookup table

and select look up table to find the number of 1's and their exact position. In our scenario we just required 3 positions of 1. So the next bit pattern is 1110 and it has three 1's, they satisfy our requirement of *select (20)*. In this case the third 1^{th} bit is *select(20)*. Now we can easily obtain the address of 20^{th} one from the stored list which is 35.Let us observe that in this two levels directory method the bit string is split into balanced consecutive blocks. When we required the *rank* or *select* operations we perform the linear search only on one particular block. Other values are yield from pre-computed lookup tables. Therefore, in Clark's [2] procedure the time consumption for rank and select operations are always constant. Let us observe that in the figure: 2.6, other than bit string the lookup tables and *ranks* of each block required less than the size of string, which is for *n* bits, then $\left(\left(\frac{n}{\log n}\right) * (\log\log n)\right) = o(n)$ of additional space. Clark's [2] algorithm is the best algorithm for *select* procedure up to now, because this is the only algorithm which supports the *select* in constant time. But the performance is varies between the size of the strings. Clark's [2] algorithm gave best results when the string is too long. In this algorithm the performance is affected while distributing the 0's and 1's especially when the bit string is small, we need large space to construct lookup tables. It also gives in sufficient results when we split sub blocks using byte based method.

Munro and Raman [14] implemented a three level directory scheme which gave better results only for *rank.* Clark's [2] algorithm is practically implemented by Dong et al. [4] and its behavior is studied which resulted by proposing two more algorithms and they are close to the theoretic lower bound.

2.6 Conclusion

In this chapter we discussed the basic operations of *rank* and *select* for succinct dictionaries. It gave a clear picture and better understanding of performance tradeoffs between space and time. We described how to support *rank* and *select* functions over a bit string efficiently. In the following chapter we describe succinct data structures for ordinal and cardinal trees. We use the ideas mentioned in both these chapters to develop our new dynamic representation for cardinal trees.

3 Trees in Succinct Data Structures

Trees are the underlying structures to store the data and very quick to access or delete. Traditionally, trees are stored using data structures and pointers which are linked among the nodes with wasteful of space and time. Thereafter, succinct or space efficient data structures are eminent to store the data very close to information theoretic lower bound. Trees are many types, and they are binary trees, ordinal trees, cardinal ary trees and many more. In this chapter, we discuss succinct representation of several trees and their supporting operations.

3.1 A Simple Binary Tree

Trees are most essential structures in computer science. A ***tree*** is a hierarchical representation of collection of items or data sets in computer's memory [1 - 3, 5, 6, 8 - 12, 14, 19, 20, 27, 29, 30, 32]. Trees are many types. But, a ***binary tree*** is a tree like structure which contains utmost two nodes and each node may or may not have right or left child but they must be always specified [1-12, 14-16, 21-22, 27, 30-39]. A simple binary tree of 10 nodes is shown in the figure: 3.1.

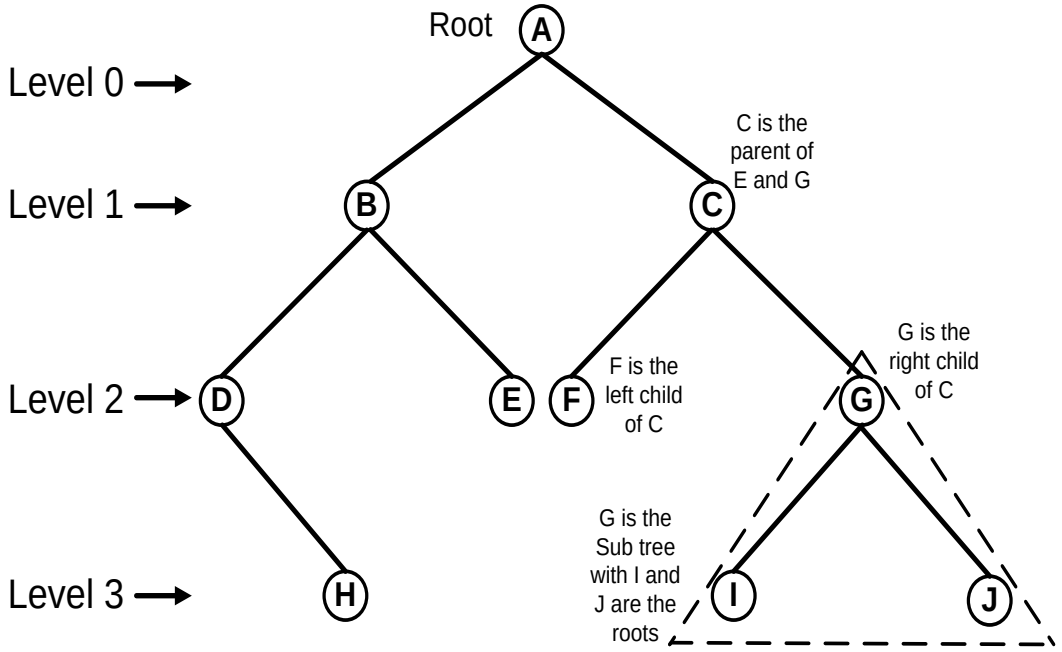

Figure: 3. 1 A Simple Binary Tree structure

Each binary tree contains a root node and maximum of two child nodes are connected by edges. In figure: 3.1 the nodes E, F, H, I, J are leaf nodes, and B, C, D, G are the internal nodes. Traditionally, trees are stored in the data structure using pointers which are linked among the nodes. Each node contains memory and it does not determine any fixed dimension or size or layout. Each node connects with one or more pointers along with their data. A simple binary tree which is shown in figure: 3.1 is represented using pointers in the following figure: 3.2.

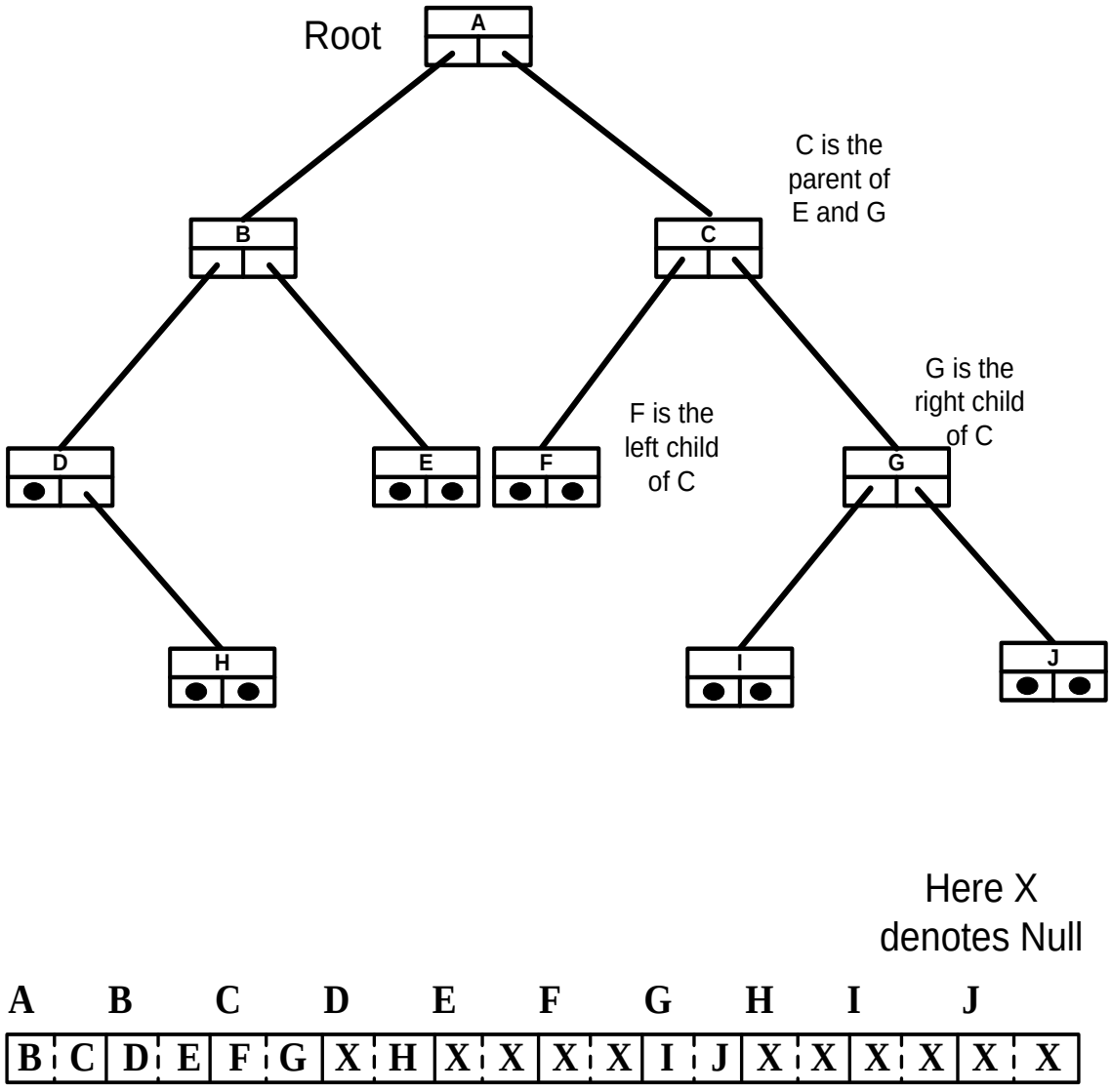

Figure: 3. 2 A Simple Binary Tree representation using Pointers.

In pointers, each root node contains a right child pointer and a left child pointer. If a node does not have any child then it denotes a null pointer. In figure: 3.2, the nodes of E, F, H, I, J are null nodes. Let us assume that if a tree has n number of nodes, then just for addressing each node it required $\lceil \log n \rceil$ bits of space. A structure of $O(n)$ pointers required to link among the nodes takes $O(n \log n)$ bits. A standard representation of binary tree of n nodes is $C_n = \frac{\binom{2n}{n}}{n+1}$, where C_n is the ***Catalan number*** [1-12, 14-16, 21-22, 27, 30-41]. In pointers it is very easy to store other information within the pointer, but moving the data from one to another node is quite fuzzy, which is possible by dereferencing the pointer. The addresses of pointers are free from memory. The operations like insert and delete are possible by dereferencing the pointers. The procedure of dereferencing from one node to another node is possible, but the program is not allowed to embed the numerical values to the pointers. In this conventional method each node occupies a distinct block of memory. Let us observe in the figure: 3.2, a 10 node simple binary tree is consumed 30 bits of space, and 10 bits of extra space is required to support other operations ($where\ n = 10,\ such\ that\ (n\log n = 10)$). As we know, trees are the best known data structure for storing massive amount of data such as genealogical information, astronomical data, DNA sequences and many more. When

it comes to the large extent of nodes then the pointer representation of tree is required hell amount of space and it destroys its optimization.

A simple binary is possible to store asymptotically close to the information theoretic lower bound. We know n nodes of binary tree is $C_n = \binom{2n}{n}/(n+1)$ [1-12, 14-16, 21-22, 27, 30-41]. Let us take $\log n$ both sides.

$$\log C_n = \log\left(\frac{\binom{2n}{n}}{n+1}\right)$$

$$= \log\frac{(2n)!}{(n!)^2} - \log\ (n+1)$$

$$= \log(2n)! - 2\log(n!) - \log(n+1)$$

$$= 2n\log(2n) - 2n\log n + \ o(n)\ (Stirlings\ Approximation)$$

$$= (2n)(1+\log n) - 2n\log n + o(n)$$

$$= 2n + o(n)$$

Here, we can store n nodes of binary tree in $2n + o(n)$ bits of space. Jacobson [1] designed a procedure to store a binary tree using ***Level Order Unary Degree Sequence*** (LOUDS) by achieving $2n + o(n)$ bits of space. We discuss Jacobson's [1] procedure in the following sections.

3.2 Level Order Representation of a Binary Tree

Jacobson [1] implanted a procedure by representing in level order to achieve better space bounds. A simple unbalanced binary tree with 10 nodes is shown in figure: 3.3. An *internal* node defines minimum of one child. An *external* node defines without any node. To represent our binary tree succinctly, we transform our binary tree into balanced binary tree by adding external nodes as shown in figure: 3.4. A complete binary tree transformation method is as follows.

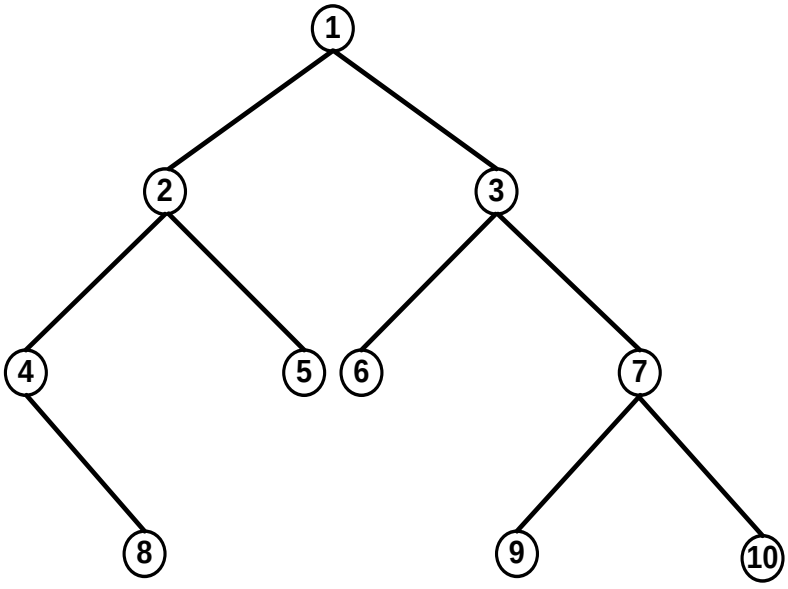

Figure: 3. 3 A Simple 10 node Binary tree

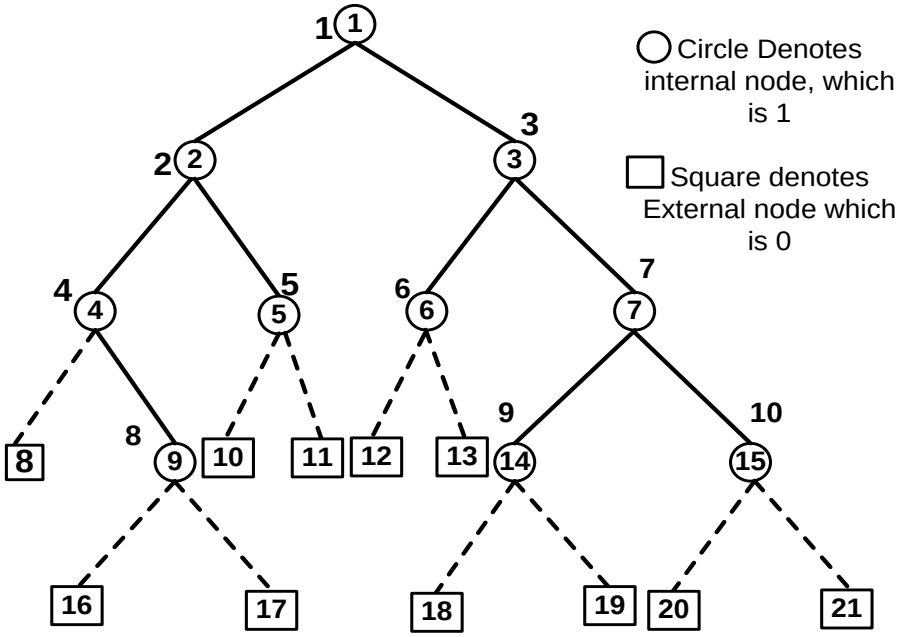

Figure: 3. 4 A Simple Balanced Binary tree

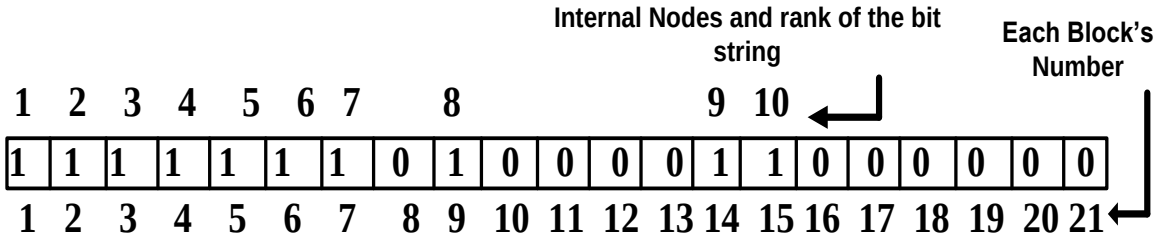

Figure: 3. 5 The Binary tree transformation into Bit String using level order sequence

- Represent all the internal nodes with 1.
- Add external nodes to the tree and represent them with 0.
- Write all the nodes in level order sequence, which is left to right and top to bottom approach. Therefore we write like **1→2→3→4→5→6→7→8→9→10**.

The figure: 3.5 shows a bit vector which represents our binary tree in a level order. Here, observe that our tree consists of 10 nodes and it took 11 extra nodes to construct as a complete binary tree. Therefore an n node tree required $n+1$ extra bits, such that $n+(n+1)=2n+1$ bits required to represent a complete binary tree.

Now, we can do tree operations like visiting the nodes, finding right child, left child and parent by using *rank* and *select* operations which are discussed in chapter 2. Here, again let us remind *rank* and *select* operations.

rank: counts number of 1's up to the position itself in a given bit sequence.

select: finding the required 1^{th} element in a given bit sequence.

Let us observe in the figure: 3.5 shows that the *ranks* of bit string in the first row and numbers of each consecutive block in the last row.

Here, we can see the following formulas for tree traversing [1-4].

- $left\ child\ (x) = 2 * rank(x)$
- $right\ child\ (x) = 2 * rank(x) + 1$
- $parent\ (x) = (\lfloor select\ x/2 \rfloor)$

We compute above operations based on the bit string available in figure: 3.5.

Example: 3.1

Now we compute right child, left child and parent of the required node using above formulas.

$left\ child\ (7) = 2 * rank(7) = 2 * 7 = 14.$

$Lets\ observe\ in\ figure\ 3.5, the\ rank(7) is\ 7. Therefore\ right\ child\ of\ 7\ is\ 14.$

$right\ child\ (4) = (2 * rank(4) + 1) = 2 * 4 + 1 = 9.$

$Lets\ observe\ in\ figure: 3.5, the\ rank(4) is\ 4. So\ the\ right\ child\ of\ 4\ is\ 9.$

$parent\ (10) = (\lfloor select(10/2) \rfloor) = (\lfloor select\ (5) \rfloor) = 5.$

$Lets\ observe\ in\ figure: 3.5, the\ 5th\ 1\ position\ is\ 5. Therefore\ the\ parent\ of\ 10\ is\ 5.$

In this section we discussed a new way of representing binary trees succinctly and their supporting operations. But we may think that why not we store an unbalanced tree directly into a bit string to achieve better space bounds. Of course, we may do that, but unfortunately we cannot perform supporting operations efficiently. Therefore we are sacrificing some space to achieve best performance. Let us observe that the bit string shows in figure: 3.5 occupied $2n + 1$ bits of space and the ranking directory occupied $o(n)$ bits, such that the total space occupied by the binary tree is $2n + o(n)$ bits. We can also observe that the bit accessing time is optimal, which is $O(\log n)$time.

In other hand, assume that if a tree has more than two nodes how we can represent them. Generally, binary trees are also one type of ordered trees. In the following section we discuss about ordered trees, a procedure to represent them succinctly, and their supporting operations.

3.3 An Ordered Tree using Level Order Unary Degree Sequence (LOUDS)

An *ordered tree* is a tree where each children of every node is ordered [1-4, 6-14]. Jacobson [1, 6] gave a method to represent rooted ordered trees by using degrees of each node. A simple rooted order tree is shown in the figure: 3.6. We discussed from the above section to represent a tree in level order. Now we use the same procedure by using degree d of each node. Here, degree d is an integer and it is represented as d number of 1's followed by 0, where 0 differentiates each node.

In the above figures all the 13 nodes except the root node are the child of other nodes. Now we write these tree degrees in unary sequence.

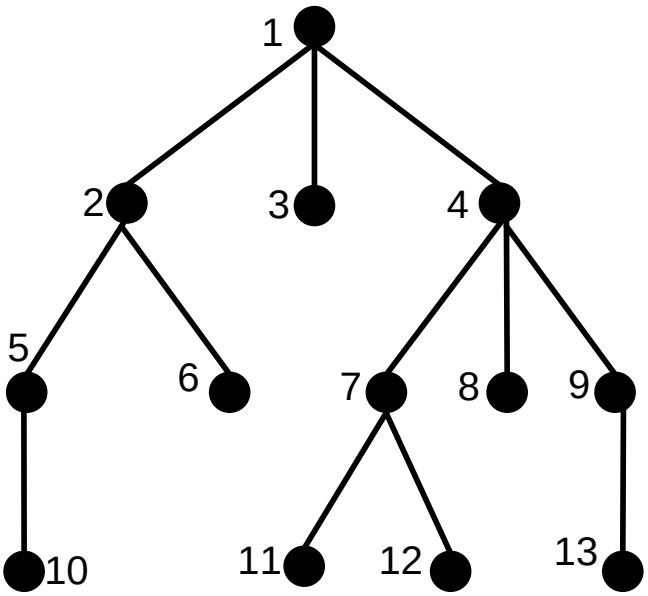

Figure: 3. 6 A Simple Rooted Order Tree

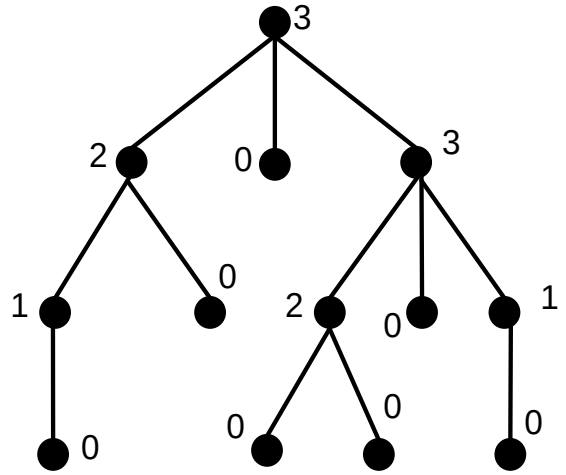

Figure: 3.7 An Ordered Tree with Degrees

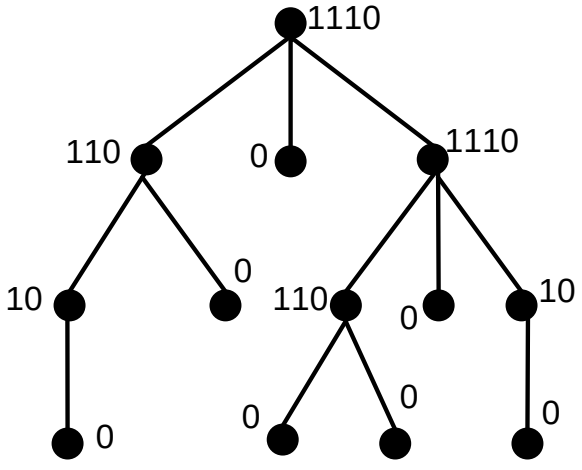

Figure: 3.8 An ordered Tree Representation in Unary Sequence

1	1	1	0	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0	1	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Figure: 3.9 Representing Rooted Ordered tree in a Bit Vector

The figure: 3.9 shows that the tree contains 12 number of 1's and 13 number of 0's. Therefore an n node tree contains $(n-1)$ number of 1's and n number of 0's, such that $n+n-1=2n-1$ bits required to store an ordered tree. If an n node tree contains $n-1$ number of 1's then it cannot support operations efficiently and it also doesn't satisfy the rule of $2n+1$ bits of space per tree. To resolve this tradeoff, Jacobson [1, 6] added a super root to the present root of the tree which is shown is figure: 3.10. Hence, the degree of super node denotes 10, which is shown in the figure 3.10. We construct a bit string for our extra rooted tree, which is shown in figure: 3.11. Now let us observe figure: 3.10 shows that the bit string of length 27, since the tree has 13 nodes. The following operations are supported in this LOUDS [1, 6, 11] representation.

- $first-child(x): select_0\big(rank(x)\big)+1$
- $next-sibling(x): x+1$
- $parent(x): select(rank_0(x))$

We compute above operations based on the bit string available in figure: 3.11.

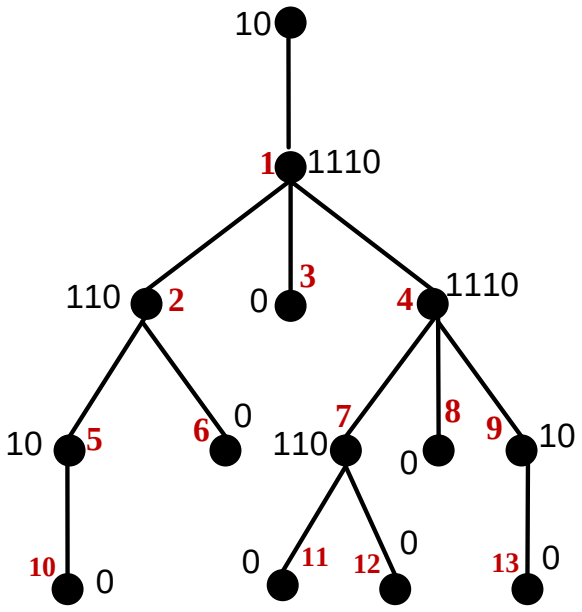

Figure: 3.10 An Ordered tree with a Super node

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	0	1	1	1	0	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0	1	0	0	0	0	0

Figure: 3.11 An Ordered Tree representation of Bit String with an extra Super Node

Example: 3.2

- $first - child(5): select_0\big(rank(5)\big) + 1 = select_0(5) + 1 = 10$

  Here, $rank(5)$ is 5 and $select_0(5)$ + next block in the bit string is 10.

- $next - sibling(6): 6 + 1 = 7$, here the next sibling after 6 is 7

- $parent(7): select\big(rank_0(7)\big) = select(4) = 4$

  Here, $rank(7)$ with respect to the zero's is 4 and the *select* (4) is 4.

Jacobson [1, 6] proved all these operations by occupying $2n + o(n)$ bits and $O(\log n)$ time.

3.4 Representing the Tree using Balanced Parentheses Method

Munro and Raman [14] gave a different approach to encode an ordinal tree using parenthesis. Instead of using 1's and 0's Munro and Raman [14] encode the tree using parenthesis representation which is shown in figure: 3.13.

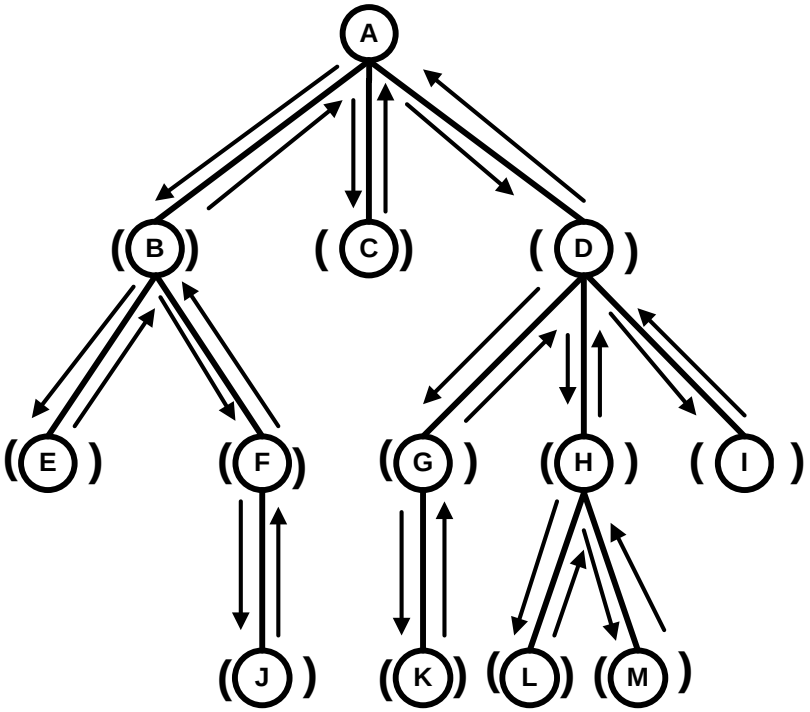

Figure: 3.12 A Simple Ordinal Tree Representation using Balanced Parenthesis method

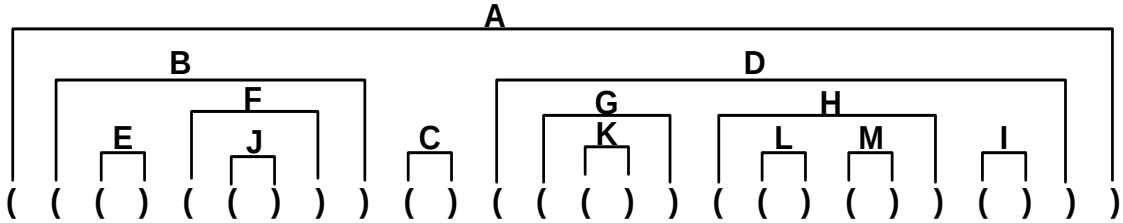

Figure: 3.13 A Parenthesis Encoding of an Ordinal Tree

In this representation an open parenthesis denotes the opening of a new node and a closing parenthesis denotes the closing of the node where the node's children end. We can encode the tree by traversing left to right unary degree sequence. In the figure: 3.12 shows that the traversal procedure with arrows. We can encode the tree using this balanced parenthesis representation as well as we can rebuild the tree using parenthesis sequence. In the figure: 3.13 shows that the parenthesis sequence which is built from the tree shown in figure: 3.12. Let us observe that in figure: 3.12, the tree consists of 13 nodes and which occupied 26 bits shown in figure: 3.13. Therefore, the balance parenthesis representation of an n node ordinal required $2n$ bits of space. In this representation the structure supports some mode additional navigational operations than previous representations. Munro and Raman [14] also proved that this representation of ordinal tree consumes $O(1)$ constant worst case time by occupying additional $o(n)$ bits which is negligible. We can also observe in figure: 3.12, the nodes of our tree stored contiguously in figure: 3.13.

In this representation we can support many operations by using open and closing parenthesis. In the previous representations we compute the *rank* and *select* using 1's and 0's, but in this representation we compute using open parenthesis which is "(" and closing parenthesis whicu is ")". Therefore, $rank_{open}(x) = rank_1(x)$, $rank_{close}(x) = rank_0(x)$, $select_{open}(x) = select_1(j)$ and $select_{close}(j) = select_0(j)$. The following operation can be performed by using this method [14].

- *findclose(x):* finds the position of the closing parenthesis matching with the open parenthesis at the given position *x*.
- *findopen(x):* finds the position of the open parenthesis that matches with the closing parenthesis at the given position *x*.
- *excess(x):* finds the difference between the number of opening and closing parenthesis before the given position of *x*.
- *enclose(x):* finds a parenthesis pair whose parenthesis position available in position *x*, return the position of the open parenthesis with respect to the closest matching parenthesis pair enclosing to the given position *x*.

When compare with the previous tree representations, here we can perform some additional operations.

- *parent:* enclosing parenthesis
- *first child:* next parenthesis, if its open
- *next sibling:* open parenthesis following with the closing parenthesis, if it is available
- *subtree size:* half of the number between a pair of parenthesis
- *lca(i,k):* least common ancestor of the given nodes. Example: from figure 3.12, *lca(K.M) = D*

The only drawback in this representation is to find the x^{th} child takes $\theta(x)$ time.

3.5 Representing the Trees using Depth-First Unary Degree Sequence Method

In the previous representations, we have seen that Jacobson's [1, 6] LOUDS representation achieved to access the x^{th} child in constant time, and Munro and Raman's [24] BP (balanced

parenthesis) representation supported the subtree size in constant time. Based on these two advantages Benoit et al. [11] gave a new and different representation by combing Jacobson [1, 6], Munro and Raman's [24] ideas together.

- Firstly, add a super root for a given tree, similarly like Jacobson's [1, 6] representation, which is shown in figure: 3.14.
- Denote the degrees for each node.
- Write the degrees in unary degree sequence that means we represent 1's and 0's. But we denote only 1 for the super node but not 10, which are shown in figure: 3.15.
- Now place an open parenthesis instead of 1 and closing parenthesis instead of 0, which is shown figure: 3.16.

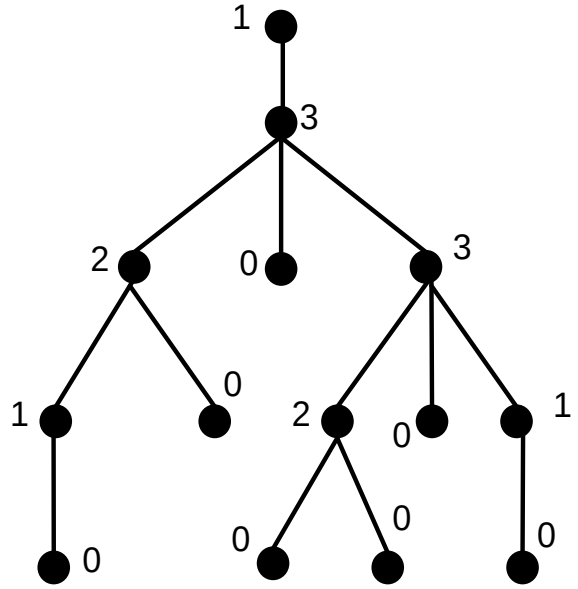

Figure: 3.14 A Simple Ordinal Tree denoted with degrees

1	1	1	1	0	1	1	0	0	1	1	1	0	1	0	0	1	1	0	0	1	0	0	0	0	0

Figure: 3.15 Bit String in Unary Sequence

(((( ) (( )) ((( ) ( )) (( )) ( )))))

Figure: 3.16 Parenthesis Representation of the tree shown in figure 3.15

In this representation our tree of 13 nodes occupied 26 bits of space. That means an *n* node tree occupied *2n* bits of space. As we discussed above when we represent a tree in depth-first order then the supporting operation like parent, child, degree and subtree are possible in constant time. Therefore, Benoit et al. [14] proved that all the supported operations are occupied $2n + o(n)$ bits with $O(1)$ time.

3.6 Cardinal Trees

A ***cardinal tree*** is a tree where each node has ***k*** slots and labeled from 1….*k*, and it is also known as a multi-way tree with the maximum of ***k*** children. An n node binary tree with ***k*** degree is represented as $C_n^k = \frac{\binom{kn+1}{n}}{(kn+1)}$ distinct cardinal trees [41]. Simply applying log on both sides we get log $C_n^k = (k \log k - (k-1)\log(k-1))n - O(\log(kn))$ bits required to store ***k-ary*** cardinal tree in a theoretic lower bound. David et al. [11] given a structure for ***k-ary*** trees, which supports parent, child, label and subtree size in constant time as $(\lceil \log k \rceil + \lceil \log e \rceil)n + o(n) + (\log \log k)$ bits. A simple cardinal tree as shown in figure: 3.17

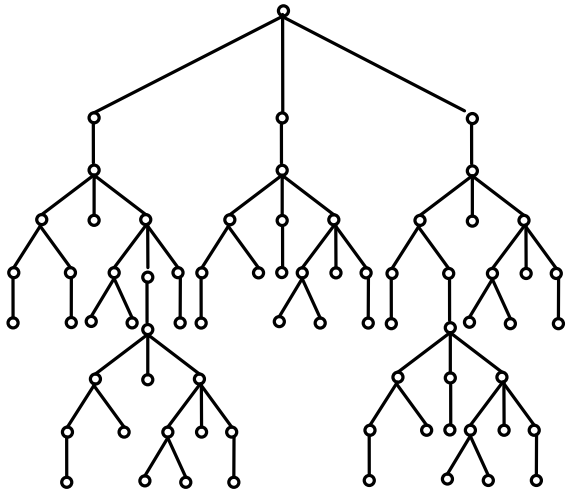

Figure 3.17 A Simple Cardinal Tree

Benoit et al. [14] also gave a representation which stores close to the information theoretic lower bound that is $((\lceil \log k \rceil + \lceil \log e \rceil)n + o(n + \log k))$ bits.

But all these methods are ***static*** that means they only support accessing the data, with derived linear space and constant time. Most recently Diego [42] implemented a representation of k-ary cardinal tree of n nodes which required $2n + n \log k + o(n \log k)$ bits of space by supporting navigational operations in $O((\log k + \log \log n)\ (1 + \frac{\log k}{\log(\log k + \log \log n)}))$ amortized time. Here k is an alphabet which is drawn from an alphabet set $\{1 \ldots . k\}$.

In chapter 4, we discuss the trade off's in Diego's [42] representation and also how we overcome those trade off's using with our new optimized structure for dynamic *k*-ary cardinal trees.

3.7 Dynamic Binary Trees

All the tree representations which are discussed in the previous sections are static, which means they can only access the data and they cannot update, insert or delete. The operations like insert, update and delete are called dynamic operations. In succinct data structures, if we want to update minor information in the trees then the whole tree structure will be changed. When we come to the large trees such as DNA sequences or XML structures then the problem will be more critical. So there is a large research gap between static and dynamic trees.

Munro et al. [32] gave a representation that occupies $2n + o(n)$ bits by supporting the operations of left child, right child, parent and subtree size in $O(1)$ time. Dynamic operations like updating tree leaf or a node requires $O(log^c n)$ time, where *c* is a constant. Later, Raman and Rao [19] improved the update time to $O((loglogn)^{1+\epsilon})$, where ϵ is a some fixed

constant. Munro et al. [32] implemented this dynamic binary tree representation by splitting the tree into some consecutive blocks, where the block sizes are fixed. Later, he used implicit pointers pre-computed tables for each block. Now, the required operations only apply on the necessary sub blocks without disturbing other sub blocks. Based on these basic ideas we implement an optimized dynamic k-ary cardinal tree in chapter 4. We do not need to discuss the complete implementation of dynamic binary trees. It will be so vast. The detailed description is available in Munro et al. [32]. Now we only consider the basic ideas to split the tree into sub blocks and connecting them with pointers.

3.8 Conclusion

In this chapter we discussed several types of trees such as binary tree, ordinal trees and cardinal trees. We also have seen their representing procedures by applying LOUDS [1, 6], balanced parenthesis [24], DFUDS [11] and dynamic binary trees [32]. In this chapter we also understood the problem definition for our thesis. Now this detailed study is useful to construct an optimized dynamic *k*-ary cardinal tree.

4 An Optimized Dynamic k-ary Cardinal Tree Representation

This chapter describes our new cardinal tree representation, which is better than Diego's [42] representation. Our representation is similar to the representation of Diego's [42], but we follow a different approach to achieve better performance. Diego's [42] representation of *k*-ary cardinal tree of n nodes required $2n + n\log k + o(n\log k)$ bits of space by supporting navigational operations in $O((\log k + \log\log n)\,(1 + \frac{\log k}{\log(\log k + \log\log n)}))$ amortized time. Here k is an alphabet which is drawn from an alphabet set $\{1 \ldots . k\}$. This alphabet set consists of sorted and labelled children of each node. Improving the time for operations in the case of small alphabets is mentioned as an interesting open problem by Diego [42]. Our main result achieves this goal. In particular, we give a *k*-ary cardinal tree representation when $k = O(\sqrt{\log n})$, that uses $2n + n\log k + o(n\log k)$ bits of space and supports navigational operations, updates such as insertions and deletions in constant amortized time. This result also improves upon the representation of Raman and Rao [19] and that of Chan et al. [25, 43].

4.1 Problem definition

In the previous chapters, we discussed succinct representation of several trees, their supporting operations and their complexities [1-4, 6-14]. All those trees are static, which means that they support only basic operations such as parent, child, degree, depth, pre order, subtree size, is-ancestor and many more. But these static structures do not support updates, i.e., inserting and deleting nodes into the tree. Later on, Munro et al. [32], Raman and Rao [19] gave succinct representations for dynamic binary trees that occupy the optimal $2n + o(n)$ bits. The structure of Raman and Rao [19] supports navigating operations in worst case constant time, and updates take $O((\log\log n)^{1+\epsilon})$ amortized time, where $\epsilon > 0$ is any fixed positive constant.

An efficient representation of succinct dynamic k-ary cardinal tree was discussed as an open problem by Munro et al. [32]. When we apply Munro et al. [32] procedure to our dynamic k-ary cardinal tree then it gives $O(k)$ worst case time which is not acceptable. When we assume k is always constant then we get better results. Diego [42] also used the same technique by keeping the k as a constant to achieve $O((\log\log n)^{1+\epsilon})$ amortized time. We use a different approach while partitioning the trees, inserting and deleting the leaves. Therefore in our case k is not a constant.

We have another approach to represent dynamic k-ary cardinal trees succinctly by using Chan et al. [43] representation of dynamic balanced parenthesis method to our *Depth First Unary Degree Sequence* (DFUDS) [11] of the tree. In this procedure the basic operations are performed in $O(\log n)$ time. Here, the time complexity is depends on n, but not on k. Therefore, we cannot use this structure efficiently to our implementation.

In this chapter, we implement an improved method for dynamic *k*-ary cardinal trees, which is motivated by the work of Diego [42] and Chan et al. [25, 43] representations. Our data structure occupies $2n + n\log k + o(n\log k)$ bits of space and supports all basic operations in $O(1)$ time. The following operations are supported by our representation of succinct dynamic k-ary cardinal tree.

- parent(*x*): parent of required node *x*
- child(*x*, *i*): i^{th} child of required node *x*
- child(*x*, *a*): child of required node *x* by label *a*

- depth(x): depth of x
- degree(x): degree of x
- subtree-size(x): gets the subtree size of x
- preorder(x): gets the pre order of x
- is-ancestor(x, y): is the node x an ancestor of node y
- insertions: insertions for leaves possible
- deletions: deletions for nodes and leaves possible

Initially, in our data structure we split the tree into small blocks. These small blocks are useful for all basic operations and mainly to modify and update. Here, we use conventional way to split the tree into small blocks. We face following problems while implementing the structure.

- Representing inside blocks.
- When defining block sizes which are connected them with pointers, we restrict them with $o(n)$ bits.
- Block overflows which are not controlled by the presently available techniques such as table lookups and text indexes.
- The value of k is not constant.

We implement a dynamic k-ary cardinal tree which supports all the above discussed operations and resolve the above problems that occur during the implementation.

4.2 Succinct searchable partial sums structure

In our data structure we split the k-ary cardinal tree into small blocks and each small block is stored and labelled with alphabet symbols. So, in this data structure we need to search for required symbols. To get better results we must use best data structure of partial sums. Therefore, in our data structures we use succinct data structures for searchable partial sums implemented by Hon at el. [44]. Let us assume an array of n integers as $A[1 \dots m]$. Hon at el.'s [44] data structure supports to find i^{th} element in A and also supports $Sum(A, i)$ by computing $\sum_{j=1}^{i} A[j]$. We can also find the required i^{th} element using search operation in the available array A, that is $Search\ (A, i)$. Based on this operation we can also find the smallest j', where $Sum(A, j') \geq i$. We can also update the array at the i^{th} position with the given information δ in A, such that $Update(A, i, \delta)$ updates $A[i] \leftarrow A[i] + \delta$. Another operation of Insert is also supporting by the given element e, at the position of i in a given array A, such that it denotes as $Insert(A, i, e)$. We can also perform delete operation very simply by giving required element of j in a given array A, such that $Delete(A, j)$.

Theorem 1.1. Suppose A is an array of m integers where each element of A[i] is between 1 and k. On a RAM with word size $\log n$, one can maintain this array using $n \log k + o(\log k)$ bits to support $Sum, Search\ and\ Update$ operations in constant time, and $insert\ and\ delete$ operations in constant amortized time, when m is $polylog(n)$, and k is $\sqrt{\log n}$.

Let $A[1..m]$ be the array which we want to store using the searchable partial sums structure. We describe the structure for the case when $m = (\sqrt{\log n})$. This can be easily generalized to the case when $m = O(\log^2 n)$. We consider the case when $k = O(\sqrt{\log n})$.

Hon at el. [44] implementation was done on a traditional red-black tree where each node is balancing by itself. Let us assume that T_A is a red-black tree and its leaves are elements of A. All the internal nodes of T_A represent a pointer for each node. In each pointer of one bit confirms available position of the parent node, total number of elements available in its left subtree and their total sum available from their leaves. Such that, we can indicate nl is

number of elements in left subtree and sl is the sum of elements available in the leaves. The total space is occupied in this pointer representation is $O(n \log n)$ bits. We can perform all the basic operations on the information available parent nodes. This structure is also supports the navigations from root to leaves. Hence the total structure consumes $O(\log n)$ time. Here, we optimize performance while computing the sum up to give position in A by computing with respect to the corresponding leaf in T_A, and eliminating unnecessary knowing position of j'. Therefore all the operations can be supported in $O(1)$ constant time.

4.3 A data structure for the Balanced Parenthesis encoding

In the previous chapter we discussed the balanced parenthesis representation and its supporting operations. Now we utilize this data structure in our sub blocks of dynamic *k*-ary cardinal tree. As we know Munro and Raman [32] implemented all necessary operations, by occupying $2n + o(n)$ bits of space, in constant time. Now we briefly discuss those supporting operation with respect to our tree.

Let us assume a balanced parenthesis sequence is P, and its length is $2n$. A basic operation is to find the closing parenthesis from the given sequence by using $findclose(P, i)$, which finds the closing parenthesis of the given position i matching with the given opening parenthesis. Another operation is to find the opening parenthesis in the given sequence by using $findopen(P, j)$, which finds the opening parenthesis of the given element j matching with the given closing parenthesis. An additional operation is to find the closest matching parenthesis by using $enclose(P, i)$, which gives a closest matching pair of the corresponding element i. One more additional operation is to find the difference between the open and close parentheses up to the given element i by using $excess(P, i)$.

When we utilize this data structure of balanced parenthesis in our *k*-ary dynamic tree, we change the space and time bounds when we insert or delete any pair of parenthesis from the available sequence. When it comes to perform the insertions or deletions there are two better data structures available. They are implemented by Chan et al. [25, 43], and Makinen and Navarro [45]. These both data structures support all static and dynamic operations in $O(\log n)$ worst case time by occupying $O(n)$ bits of space. Addition to this, Makinen and Navarro's [45] data structure supports dynamic operations for a new element is also in $O(\log n)$ worst case time. So we utilize Makinen and Navarro's [45] data structure for representing the blocks of our *k*-ary cardinal tree. Based on this representation we can minimize the time bounds while retaining the same space bounds. Therefore, from both the cases the worst case time is $O(1)$ time. In other case the following lemma is applied.

Lemma 4.1. When $N = O(\log^2 n)$ there exists a dynamic representation of balanced parentheses which is length $2N$ bits, and occupying $2N + o(N)$ bits of space for supporting all the operations such as *findopen, findclose, enclose, excess, rank, select, insert and delete* in $O(1)$ worst case time.

4.4 A data structure for the DFUDS encoding

In the previous chapter, we discussed several types of succinct representation of trees. In this section we briefly describe some useful structures to represent the tree structure of the *k*-ary cardinal trees, such as LOUDS [1, 6], DFUDS [11], balanced parenthesis [14], and ultra succinct representation of ordered trees [27]. All these representations occupy $2n + o(n)$ bits of space and each representation supports different kinds of operations in constant time.

Now we use some of the required structures in our *k*-ary cardinal tree representation with respects to our requirements. When it comes to the Diego's [42] *k*-ary cardinal tree, he used

balanced parenthesis [14] and DFUDS [11]. So their supporting operations occupy $2n + n\log k + o(n\log k)$ bits of space, $O(\log k + \log\log n)$ time for navigational operations, and $O\left((\log k + \log\log n)\left(1 + \frac{\log k}{\log(\log k + \log\log n)}\right)\right)$amortized time for insertions and deletions. But in our dynamic representation we optimize time bounds. In the previous section we discussed a method to optimize balanced parentheses. Now we optimize the performance of DFUDS [11] to utilize in our dynamic *k*-ary cardinal trees.

As we know DFUDS [11] supports all the basic operations in constant time, but they are static operations. Now we implement a structure which supports dynamic operations. The idea is basically inspired by Jacobson [1, 6] and Munro et al. [11]. Firstly, we define a tree and start traversing of each node by representing degrees. Secondly, we add an open parenthesis to our root node and we write in pre order based on their degrees. When we find the degree of each node we assume that $degree + 1$ parenthesis represents the position of the required $degree$ of node. Now we store the edges of nodes by labelling in a sequence of S. The stored labels of children are also stored in S. We use a data structure which is discussed in chapter 2, to compute *rank* and *select* operations on the sequence of S. Let us assume that D is a DFUDS sequence in S, for the node of x, with the given position p. Such that the required element in p is *rank* of available position in D. Here, we follow Diego's [42] procedure to supporting basic operations. Let us assume that $n_\propto = rank_\propto(S, p' - 1)$ and $i = select_\propto(S, n_\propto + 1)$. In this scenario if i is available between the positions of $p'$and $p' + degree(x) - 1$, then the child is $(i - p' + 1)^{\text{th}}$ child of x. Based on this approach we achieve all the basic operations, insertions and deletions in constant time.

4.5 Efficient Implementation of optimized dynamic *k*-ary cardinal tree

In this section we implement an optimized representation of dynamic *k*-ary cardinal tree. To achieve this we use our new approaches which are discussed in sections 4.4 and 4.5. If we use DFUDS [11] method in our internal tree then the insert operation moves the existed nodes, which results to reconstruct the tree. When we delete the nodes then also the structure must required reconstruction. Diego [42] adapted ideas of DFUDS [11] and dynamic binary trees [32] to implement his structure. He used a technique by keeping k as a constant and his time bound result only depends on n. His data structure gives better results when k is large and he also derived k is a large constant all the time. But, in our case we split the tree with different parameters and we represent trees by using optimized approaches discussed in sections 4.5.4 and 4.5.5. In our scenario the value of k changes according to the inner subtrees and the time bounds are constant even when k is $O(polylog(n))$.

In this section we present an optimized representation of dynamic *k*-ary cardinal tree which is better than Diego's [42] representation.

4.5.1 Basic *k*-ary Cardinal Tree Representation

An efficient way to represent a *k*-ary cardinal tree is splitting the tree into disjoint blocks. This idea is basically adapted by Munro et al. [11]. Each block consists of N nodes, which are connected by a pointer. We define the block sizes like maximum as N_{max} and minimum as N_{min}. We connect all these nodes using pointers. Hence, all these nodes are connected and the entire tree can be navigated. Let us assume that p is a block and but not a leaf then x is a leaf or x is a inter block of p.

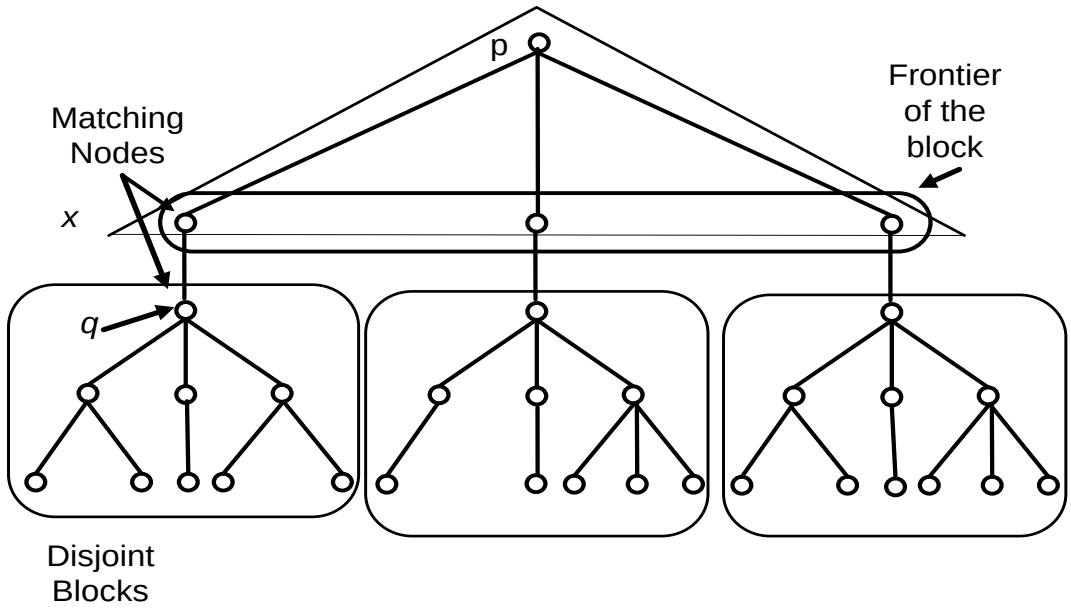

Figure: 4. 1: A Basic *k*-ary Cardinal Tree Representation

Let us observe the figure: 4.1. It shows that splitting the tree into disjoint blocks and their root of the nodes available in frontier of the block. In each frontier block the root of x is available which is connected with child block q by inter block pointers. Each disjoint block of tree should be an independent tree to support all the operations. So now we create matching nodes of x by storing dummy root of q. Now every disjoint block is a tree itself. Let us remember that we got two cases which are described below.

- Every node of x is a block of p, which means p is a leaf of x.
- Every node of x is a root node of q.

Hereafter, we should not think about the whole tree, because the tree is splitting into disjoint blocks. So now we concentrate only on disjoint block of q with respect to its root node x and super root of p. As we know the root node x is connected with q by the inter block pointer. The inter block pointer of x contains all the sibling nodes of information. This technique is useful when we insert new elements in the pointer then the pointer itself partitioned without affecting on q, which means that the whole subtree of q will not be disturb. If we insert new elements inside the blocks then we reconstruct the subtree itself without affecting on q. Now we can easily apply DFUDS [11] on each block for all static operations to maintain constant time. But, dynamic operations like insertions and deletions are complicated to maintain in constant time. So, here we apply a new technique while doing partitions of the block, which is discussed in the following section. This technique is different from Diego's [42] representation.

4.5.2 Assign Block Sizes

In the above section we took two parameters to partition the block. They are N_{max} and N_{min}. Here, we take $N_{min} = \log^2 N$ and $N_{max} = 2\log^2 N$. We partition each block into assigned sizes. We used pointers for inter block pointers which occupy $O(\log n)$ bits, and they have small space of $o(n)$ bits to store the parameters of N. When we insert or deleted the elements the value of k changes according to the operations applied on inter blocks. When the k changes we must store the information of k in the root node of x. Based on the assigned parameters, while conducting the dynamic operations, such as insert or delete then the block size may be overflows or squeezed. So these two cases are explained below.

- If the block size is reached to N_{max}, then the block overflows. When it happens we create a new block of p and it is connected with inter block pointers to its root x. We update the x with respect to the q and also the value of k changes in p. The detailed procedure is explained in buffer overflow.
- If the block size is below the N_{min}, then we try to merge this block of tree to its left side subtree. But we have two cases only. We merge the squeezed tree when these two cases are must be true. Firstly, the left side subtree must be less than N_{max}. Secondly, the left side subtree must contain the empty space, which is greater than the size of squeezed tree. Here, we may think that why we should merge the squeezed tree into left side subtree. The answer is we represent the subtree structure using DFUDS [11], where each tree represents from unary degree sequence and left to right approach. When the merging procedure is over the squeezed tree is not available anymore. But the information is available in its left subtree. Therefore, we must update the inter block pointer information with respect to q and also the value of k in p.

We have partitioned the *k*-ary cardinal tree into sub blocks, and applied insertions and deletions on sub blocks. But we must know how the information is changing in each block. So we implement a layout structure for blocks.

Now let us assume that the block p has N nodes, and it has a root node r_p and N_c child blocks. Also assume that the node q has an inter block pointer PTR_p with a set of N_c pointers. In the frontier block the root node x is connected with p using pointer r_p and the flags of p represented as F_p.

4.5.3 Managing the Tree Topology of Blocks

In this section we represent tree topology of blocks similar to Diego's [42] structure but the parameters of N varies between N_{max} and N_{min}. Here, we represent a tree structure T_p of block p, plus the edge labels S_p using appropriate succinct dynamic data structures. So they need not to be built from scratch after every update.

The structure of the tree T_p is represented using the following dynamic tree representation.

Lemma 4.2 Given a tree on $N = O(\log^2 n)$ nodes, there exist a dynamic DFUDS [11] representation that uses $2N + o(N)$ bits. On a RAM with word size $\theta(\log n)$, this representation is allow us to support the operations *parent, i^{th} child, degree, subtreesize, preorder and selectnode*, all in constant time. Insertions and deletions can be supported in $O(\sqrt{\log \log n})$ amortized time.

The main idea is used to prove the above lemma us to maintain the DFUDS [11] sequence at the leaves of a complete tree with branching factor $\log^{1/4} n$, and hence constant height. To support updates efficiently, we store them temporarily in an "overflow block", and occasionally merge them with the actual leaves. The details are similar to the proof of the next lemma. The overall space requirement of the tree structure for all the blocks is $2n + o(n)$ bits.

4.5.4 Representing the edge labels

We store the symbols labeling the edges of T_p in an array S_p. The labels are sorted according to the DFUDS [11] order. We represent the array S_p using a dynamic data structure that supports *rank* and *select* queries. The cardinal tree representation of Diego [42] uses the data structure of Gonzalez and Navarro [44] for storing the array $S_p$to support *rank* and *select*,

which is the best-known structure for general alphabets. As we only consider the case when the alphabet size k is small (in particular, $k = O(\sqrt{\log n})$), we use a different data structure for this purpose. We briefly describe this structure below.

Lemma 4.3 *Given a sequence of length $m = O(\log^2 n)$, over an alphabet of size $k = O(\sqrt{\log n})$, there is a representation that uses $m \log k + o(m \log k)$ bits, and supports rank and select queries on the sequence in constant worst-case time, and updates in $O(\sqrt{\log \log n})$ amortized time.*

Proof. Given a sequence S of length $m = (\log^2 n)$, over an alphabet of size $k = O(\sqrt{\log n})$, we store the sequence of elements at the leaves of a complete tree where the branching factor of each node is at most $b = \log^{1/4} n$. Also the branching factor of any node, except the root, is at least $b = 4$. The height of the tree can be easily shown to be $O\left(\frac{\log m}{\log \log n}\right) = O(1)$ as $m = O(\log^2 n)$. All the data in the leaf nodes are packed together and stored as a sequence of bits packed into b words of $(\log n)$ bits each. Thus for every $O(b)$ words, we might waste one word or $\log n$ bits. Thus the total wasted space due to this packing is $O\left(m \log \frac{k}{b}\right) = 0(m \log k)$ bits.

For each node that is at one level above the leaf level, we store an "overflow block" of size bl bits, for some parameter l to be chosen later. This overflow block is used to store the insertions or deletions into that node temporarily. All the overflow blocks are packed together and are stored as a sequence of words. Each insertion can be stored as pair (i, j) consisting of the position $i \in [0 \ldots b]$ at which a new leaf has to be inserted together with the character $j \in j[1..k]$ that is to be inserted. Thus each insertion is stored in the overflow word using $\log b + \log k = O(\log \log n)$ bits. A deletion can be stored as a pointer $i \in [0 \ldots b]$ to the corresponding leaf, using $O(\log \log n)$ bits. Whenever an overflow block becomes full, all the insertions stored in that block are performed on the corresponding node, and the bits of the block are reset. Performing the insertions on the representation of a node can be done in constant time using the pre computed tables. Hence the amortized cost of updates becomes $O(\log \log n / l)$.

Since we spend $O(bl)$ bits for every b leaves in the tree, the total extra space used by all the overflow blocks is $O(ml)$ bits. To make this extra space to be $o(m \log k)$ bits, we choose l to be $O(\sqrt{\log \log n})$. This gives us the following result.

We use the above lemma to represent the sequence of edge labels in each block. Since each block has size $O(\log^2 n)$, and the edge labels are in the range $[1..k]$. The total space used to store all the edge labels of all the blocks is $n \log k + o(n \log k)$ bits. Using these dynamic structures and the dynamic tree structure of the block, we can support $child(x, \alpha)$ operation within a block. This can done using the operations $child(x, i)$ on the tree structure of the block, and rank and select on the sequence of preorder edge labels of the block. All these operations can be supported in constant time on a block. Updates to the block (inserting and deleting nodes within the block) can be supported in $O(\sqrt{\log \log n})$ amortized time.

4.5.5 Representing the Frontier of a Block

The frontier of a block is represented as a conceptual sorted array $Pre[0..N_c]$ storing the preorder number, within the block, of the nodes in the frontier of the block, and with $Pre[0] = 0$. We compute an array F that stores the difference between adjacent values in the array Pre, and maintain it using a dynamic searchable partial sums structure. For this purpose, we use the following lemma. Let us observe the figure: 4.2. for easy understanding.

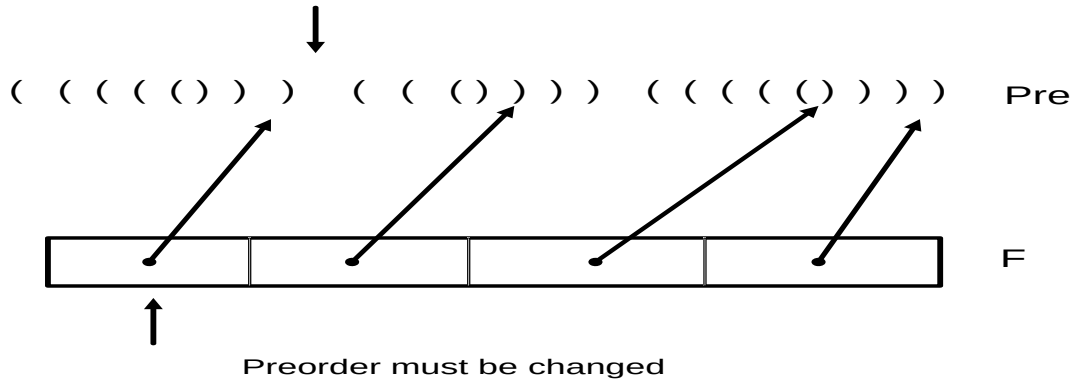

Figure: 4. 2 Frontier Block of the Tree

Lemma 4.4 Given a sequence of numbers of length $m = O(\log^2 n)$ where each number is the range $[1..k]$, one can maintain this sequence under insertions, deletions and updates using $m \log k + o(m \log k)$ bits. This dynamic structure supports sum and search queries on the sequence in constant worst case time, and insertions, deletions and updates in $O\left(\sqrt{\log \log n}\right)$ amortized time.

The proof of this lemma is very similar to the proof of the previous lemma supporting *rank* and *select* on a sequence of same length.

Using this structure, the frontier nodes of all blocks can be represented in space proportional to the number of such nodes. Since the total number of frontier nodes over the entire tree is $O(n/N)$, the space used by the all frontier nodes is $o(n)$ bits.

4.5.6 Representing Inter-block Pointers.

The following diagram shows the representation of inter-block pointers. Here, p is a sub block of tree and we store pointer information of the child in an array of $PTR_p[1 \ldots N_c]$.

This array is sorted the pre order of nodes with respect to the frontier block. All the pointers available in the pointer array are associated with frontier block elements. It also contains the information of flags and leaves of sub block pointers of tree. Each parent pointer contains the information of the parent q, and the root r_p. While doing the operations, the DFDS [11] algorithm changes updates of the parent pointer and we are unable to store the exact pointer positions.

Therefore we must spend the worst case time to store the pointer positions. So the root pointer r_p is available in frontier block. We also store the root information in flags of q with respect to its tree leaves. Based on this technique we can easily update the tree when it's needed. We can also find the absolute positions of each pointer by using information available in the tree leaves of q. Each pointer block contains the blocks of own structure. So the total number blocks are equal to the pointers available in frontier block. Therefore, the inter block pointer is $O(n/\log^2 n)$, and the total space occupied by this pointer is $o(n)$ bits.

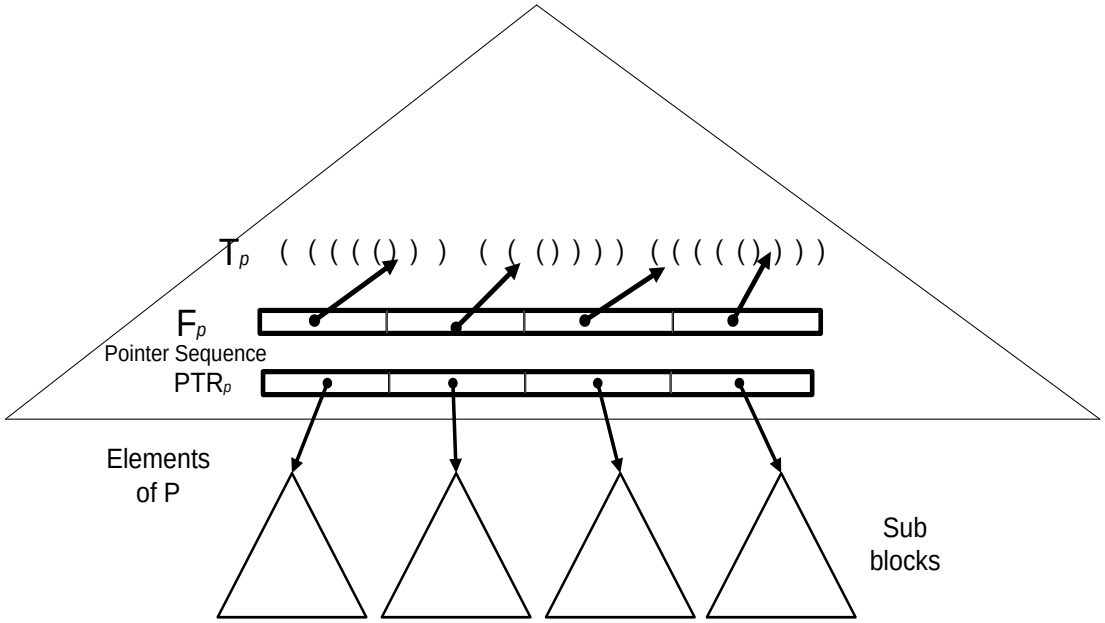

Figure: 4. 3 Inter-block Pointers Representation

4.6 Supporting Basic Operations

We discuss the basic operation for our optimized dynamic data structure as follows. For a reference we can observe the following figure, which shows a single sub-block in *k*-ary cardinal tree.

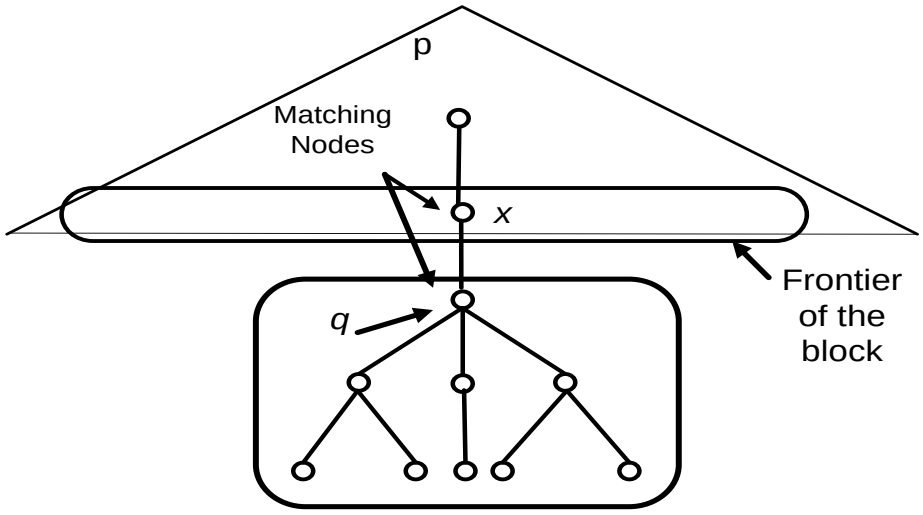

Figure: 4. 4 A Simple Sub-block of k-ary Cardinal Tree

4.6.1 Operation Child

In this operation we find the child of a given element. Here, the *child* is available in two cases. According to the figure 4.4, x does not have any leaf, but it has a child of q, and also observe that the node q has leaves. So, from the first case we can find the child of x by using $child(x, i)$ and we compute the child directly. In the second case the node q has leaves, so we use another function to compute inside the block, such that $child_p(x, i)$. We can also

compute type of the child itself. That means, a child may be a leaf of the whole tree or it may be a pointer in frontier block. Therefore, we use another operation to find the type of child by using $child(x, \alpha)$ and $child_p(x, \alpha)$. Here, we have a critical problem that how we can identify the available node is pointer or leaf of another node. This can be solved by finding the position of the element. We can find the position of the element by using $search\ (F_p, preorder(x))$, where F_p contains the all elements of x in preorder. In our case we are using a different approach rather than Diego's [42] approach. As we discussed in section 4.5.2 we create a layout structure for each frontier block. Instead of searching the whole subblock we initially find the available subblock in our layout structure. As we discussed the layout structure does not require any vast memory because we have enough memory in frontier block of pointers. Hence the child operations of $child(x, i)$ and $child(x, \alpha)$ can be supported in constant worst case time.

4.6.2 Operation Parent

We can find the parent of given element by using two cases. As we know a node which is not having its root then that node is stored in the same block. In this case we can use the operation $parent_p(x)$, and we can easily find the parent of x because the information is available in the same block. In other case we have to find the pointer node in the frontier block which is q and then we find the required position by using $selectnode_q(Sum(F_q, j))$. As we discussed earlier we are approaching better method to find the position of j. Thus, the operation parent can be supported in constant worst case time.

4.6.3 Operation Insert

When we insert a new node of x in a sub block of p the structure of the block must require an update. We can insert a new node by using operation $insert_p$ as discussed in lemma 4.2. When we insert a new node means a new edge is adding by assigning a label to the existed sequence. So the *preorder* F_p must be increased. We discussed in section: 4.5.4 that gives a best procedure to manage the edge labels. When we inserting a new node to the existing tree then we must find the position for the insertion. We can find the position using $j = search(F_p, preorder(x))$ in F_p. Again, in this operation also the time is depends on j. So the linear time is decreased in frontier block. Thus insertion can be supported in $O(\sqrt{\log \log n})$ amortized time.

Block Overflows: A buffer overflow means that the size of the tree is reached the maximum size of N_{max}. When the buffer overflow is occurred then the information must be stored in another node. But, we cannot insert a new node directly. So, we first create a duplicate node with respect to its frontier block then we update the frontier block. The procedure is as follows.

- Firstly, we split the block p into two sizes between the parameters of N_{max} and N_{min}, by creating the duplicate node z in block p. Here, the subtree of z can be reinserted as a new child of p'.
- We can perform further insertions in the blocks of $p\ and\ p'$ until the buffer overflow occurred.
- Once we create a new node of z the pointer PTR_p must be updated with respect to z.
- When we inserting a new node or element then the time is mainly consumed to find an exact location. So, to find the corresponding element the search option takes less time in our layout structure and also with the edge label mechanism which is discussed in 4.5.4. We also proved the time bounds in lemma 4.4. Addition to this we described the parameters of N must be

between $\log^2 n$ *and* $2\log^2 n$. This technique is also proved in lemma 4.2. So this result resembles and achieved in $O(\sqrt{\log \log n})$.

4.6.4 Operation Delete

When we want to delete the node x in block p by using $delete_p$, then we must update the data structure p. In another case if we want to delete some leaves of C inside p then we search the required elements of C by using the data structure $search(C_p, preorder_p(x) - 1)$. If we delete the leaves from the node p, and then if the size of the node is less than N_{min}, which is $log^2 n$ then the block overflow occurs.

If the block size is below the N_{min}, then we try to merge this block of tree to its left side subtree. But we have two cases only. We merge the squeezed tree when these two cases are must be true. Firstly, the left side subtree must be less than N_{max}. Secondly, the left side subtree must contain the empty space, which is greater than the size of squeezed tree. When the merging procedure is over the squeezed tree is not available anymore. But the information is available in its left subtree. Therefore, we must update the inter block pointer information with respect to q and also the value of k in p.

When $k = O(\sqrt{\log n})$, there exists a representation of a dynamic k-ary cardinal tree on n nodes that uses $2n + n\,logk + o(n \log k)$ bits, and supports the operations $parent, child(x, \alpha)$ in constant worst-case time, and operations insert and delete in $O(\sqrt{\log \log n})$ amortized time.

4.7 Conclusions and Future Work

We considered the problem of representing a dynamic cardinal tree in space close to the information-theoretic optimal lower bound. When $k = O(\sqrt{\log n})$, our representation uses 2n + n log k + o(n log k) bits to maintain a dynamic k-ary tree with *n* nodes, and supports all the basic navigational operations, like parent, i-th child, child labelled j, subtree size, degree etc., all in constant worst-case time, and insertions and deletions of leaf nodes in $O(\sqrt{\log \log n})$ amortized time. This improves time bounds for all the operations of the earlier known dynamic cardinal tree representation by Diego, for the case when *k* is small.

We are currently working on extending this structure to support other useful operations like *level-ancestor, least common ancestor, height, distance* etc. When the tree is static, all these operations can be supported in constant worst-case time, without increasing the space bounds. We believe the same can be done for the dynamic case.

Improving the running times for the navigational operations for the general case is still an open problem. Another interesting open problem is to improve the $O(\sqrt{\log \log n})$ amortized time bounds for the insertions and deletions of nodes into the tree.

References

[1] Guy Jacobson. Succinct Static Data Structures. Ph.D. thesis, Carnegie Mellon University, 1988.

[2] David R. Clark. Compact Pat Trees. Ph.D thesis, University of Waterloo, pages 73-101, 1996.

[3] Ian Munro, and S. Srinivasa Rao. Succinct Representation of Data Structures, 2001.

[4] Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. Efficient Implementation of Rank and Select Functions for Succinct Representation. In *Proc. of WEA*, Pages 315-327, 2005.

[5] Rodrigo Gonzalez, Szymon Grabowski, Veli Makinen and Gonzalo Navarro. Practical Implementation of Rank and Select Queries. In *Proc. of WEA 2005*.

[6] Guy Jacobson. Space Efficient Static Trees and Graphs. In *Proc. of IEEE FOCS*, pages 549-554, 1989.

[7] Daisuke Okanohara, and Kunihiko Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. of ALENEX*, 2007.

[8] Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A Simple Optimal Representation for Balanced Parentheses. In *Proc. of CPM*, 368(3): 231-246, 2006.

[9] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the LOUDS Succinct Tree Representation. In *Proc. of WEA*, pages 134-145, 2006.

[10] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct Ordinal Trees with Level-Ancestor Queries. *In Proc. of ACM Transactions on Algorithms*, 2(4):510-534, 2006.

[11] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S.Srinivasa Rao. Representing Trees of Higher Degree. *In Proc. of Algorithmica*, 43(4): 275-292, 2005.

[12] Hsueh-I Lu, and Chia-Chi Yeh. Balanced Parenthesis Strike Back. *In Proc. of ACM Transactions on Algorithms*, 2005.

[13] Kunihiko Sadakane. Succinct Representations of *lcp* Information and Improvements in the Compressed Suffix Arrays. *In Proc. of SODA*, pages 225-232, 2002.

[14] J. Ian Munro, and Venkatesh Raman. Succinct Representation of Balanced Parenthesis, Static Trees and Planar Graphs. *In Proc. of FOCS*, pages 118-126, 1997.

[15] Xin He, Ming-Yang Kao, and Hsueh-I Lu. Linear-Time Succinct Encodings of Planar Graphs via Canonical Orderings. *In Proc. of CoRR*, 2001.

[16] J. Ian Munro, and S. Srinivasa Rao. Succinct Representations of Functions. *In Proc. of ICALP*, pages 1006-1015, 2004.

[17] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/Select Operations on large Alphabets: A Tool for Text Indexing. *In Proc. of SODA*, pages 368-373, 2006.

[18] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct Representations of Permutations. *In Proc. of ICALP*, pages 345-356, 2003.

[19] Rajeev Raman, and S. Srinivasa Rao. Succinct Dynamic Dictionaries and Trees. *In Proc. of ICALP*, pages 357-368, 2003.

[20] Rajeev Raman, Venkatesh Raman, and S.Srinivasa Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. *In Proc. of SODA*, pages 233-242, 2002.

[21] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. Succinct Dynamic Data Structures. *In Proc. of WADS*, pages 426-437, 2001.

[22] J. Ian Munro, Venkatesh Raman, S. Srinivasa Rao. Space Efficient Suffix Trees. *In Proc. of J. Algorithms*, 39(2): 205-222, 2001.

[23] Venkatesh Raman, and S. Srinivasa Rao. Static Dictionaries Supporting Rank. *In Proc. of ISAAC*, pages 18-26, 1999.

[24] Kunihiko Sadakane, and Roberto Grossi. Squeezing Succinct Data Structures into entropy bounds. *In Proc. of SODA*, pages 1230-1239, 2006.

[25] Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Dynamic Dictionary Matching and Compressed Suffix Trees. *In Proc. of SODA*, pages 13-22, 2005.

[26] Wing-Kai Hon, Tak Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. Compressed Index for Dynamic Text. *In Proc. of Data Compression Conference*, pages 102-111, 2004.

[27] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-Succinct Representation of Ordered Trees. *In Proc. of ACM-SIAM SODA*, 2007.

[28] Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly Spanning Trees with Applications. *In Proc. of ACM-SIAM*, 2002.

[29] D.R. Clark, and J.I. Munro. Efficient Suffix trees on Secondary Storage. *In Proc. of 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383-391, 1996.

[30] Meng He, Ian Munro, and S. Srinivasa Rao. Succinct Ordinal Trees Based on Tree Covering. *In press*, 2007.

[31] Meng He, Ian Munro, and Srinivasa Rao. A Categorization Therom on Suffix Arrays with Applications to Space Efficient Text Indexes. *In Proc. of SODA*, 2005.

[32] Ian Munro, Venkatesh Raman, Adam J. Strom. Representing Dynamic Binary Trees Succinctly. *In Proc. of SODA*, 2001.

[33] Douglas Baldwin and Greg W. Scragg. Algorithms and Data Structures: The Science of Computing. *Charles River Media, ISBN: 1584502509, 2004.*

[34] Robert Lafore. SAMS Teach Yourself: Data Structures and Algorithms in 24 hours. *Sams Publishing, ISBN: 0-672-31633-1, 1999.*

[35] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. Data Structures and Algorithms. *Addison-Wesley, ISBN: 0-201-00023-7, 1983.*

[36] Dinesh P. Mehta and Sartaj Sahni. Hand Book of Data Structures and Applications. *Chapman & Hall/Crc Computer and Information Science Series, ISBN: 1-58488-435-5, 2005.*

[37] Niklaus Wirth. Algorithms and Data Structures. *Prentice Hall, ISBN-13: 978-0130220059 Reprinted 2004.*

[38] Jim Koegh and Ken Davidson. Data Structures Demystified. *Mc-Graw-Hill/Osborne, ISBN: 0072253592, 2004.*

[39] Ellis Horowitz and Sartaj Sahni. Fundamentals of Data Structures. *W. H. Freeman, ISBN-13: 978-0716782636, 1993.*

[40] C.E Shannon. A Mathematical Theory of Computation. *The Bell Systems Technical Journal, volume 27, pp. 379-423, 623-656, July, October, 1948.*

[41] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. Concrete Mathematics. *Addison-Wesley, Reading, MA, pages: 234-238, 1989.*

[42] A. Diego. An Improved Succinct Representation for Dynamic *k*-ary Trees. *CPM 2008, pages: 277-289, 2008.*

[43] H.L. Chan, W.K. Hon, T.W. LAM and K. Sadakane. Compressed Indexes for Dynamic text Collections. *ACM TALG 3(2), article 21, 2007.*

[44] W.K. Hon, K. Sadakane and W.K. Sung. Succinct Data Structures for Searchable Partial Sums. *ISSAC 2003, volume 2906, pages: 505-516, Springer Heidelberg, 2003.*

[45] V. Makinen, G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM TALG (to appear, 2007).*

[46] C.W. Dawson. The Essence of Computing Projects – a Student's Guide. *Prentice Hall, Harlow UK, 2000.*

[47] J.W. Creswell. Research Design - Qualitative, Quantitative and Mixed Method Approaches. *2nd ed., Sage Publications, Thousand Oaks CA, 2002.*