SEARCHING IN A SORTED LINKED LIST AND SORT INTEGERS INTO A LINKED LIST

A THESIS IN
Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

By
HEMASREE KOGANTI

B. Tech(CS), GOKARAJU RANGARAJU INSTITUTE OF ENGINEERING
AND TECHNOLOGY , India, 2015

Kansas City, Missouri
2019

SEARCHING IN A SORTED LINKED LIST AND SORT INTEGERS INTO A LINKED LIST

Hemasree Koganti, Candidate for the Master of Science Degree
University of Missouri – Kansas City, 2018

ABSTRACT

The research work consists of two parts. Part one is about Searching for an integer in a sorted Linked list. A tree is constructed in O(nloglogm/p+loglogm) time with p processors based on the trie with all the given integers. Additional nodes (O(nloglogm) of them) are added to the tree. After the tree is constructed, for any given integer we can find the predecessor and successor of the integer, insert or delete the integer in O(loglogm) time. The result demonstrates for the searching purpose we need not to sort the input numbers into a sorted array for this would need at least O(logn/loglogn) time while this algorithm for constructing the tree can run in O(loglogm) time with n processors.

Part two is on sorting integers into a linked list. There are various best algorithms for sorting integers. The current research work applies the recent important results of sorting integers in Ω(logn/loglogn) time. This algorithm takes "constant time" to sort integers into a linked list with nlogm processors and O(loglogm/logt) time using nt processors on the Priority CRCW PRAM model.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of School of Computing and Engineering, have to examine the thesis titled “Searching in a Sorted Linked List and Sort Integers into a Linked List” presented by Hemasree Koganti, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yijie Han, Ph.D., Chair
School of Computing and Engineering

Ken Mitchell, Ph.D.
Department of Computer Science Electrical Engineering

Sejun Song, Ph.D.
Department of Computer Science Electrical Engineering

CONTENTS

ABSTRACT iii

ILLUSTRATIONS vi

ACKNOWLEDGEMENTS vii

CHAPTER

1. INTRODUCTION 1

2. SEARCHING IN A SORTED LINKED LIST 4

2.1 Introduction 4

2.2 Building the Search Tree based on a Trie 5

2.3 Augmenting the Tree 8

2.4 Searching 9

3. SORT INTEGERS INTO A LINKED LIST 11

3.1 Introduction 11

3.2 The Algorithm 12

3.3 Building a Tree Based on a Trie: An Example 19

4. CONCLUSION 23

REFERENCES 25

VITA 28

ILLUSTRATIONS

Figure Page

1. Example of a Trie……5
2. Trie with Step numbers ……...6
3. Final Trie……...……..10
4. Processors assigned to each Input Integer……...12
5. Example of a Trie…….....13
6. The Nodes with one child are removed……...14
7. All the Nodes are shown……..15
8. Trie with ‘a’ sections…….……..19
9. Intermediate Levels are removed…….20
10. Linked Lists are formed at the Nodes……..21
11. Parent Node connects all its child Linked Lists……...22

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank following people who have directly or indirectly helped me in academic achievements. Firstly, I would like to thank Dr. Yijie Han for the constant and endearing support which has helped me in fulfilling my thesis. He has provided me with an opportunity to realize my potential in the field of my thesis. His encouragement and inputs were elements of vital guidance in my thesis. He has been a constant source of motivation and challenged me with algorithms, deadlines, that have contributed to me acquiring inspiration and ideology. His expertise and innovative insights have been phenomenal in completing my thesis.

I sincerely thank Dr. Ken Mitchell and Dr. Sejun Song for accepting to be a part of my thesis committee and making time for me from their busy schedule. I would like to thank the University of Missouri- Kansas City for providing me with an opportunity to continue my research and supporting me in this regard.

I would like to dedicate my thesis to my parents who constantly inspired me to pursue higher studies. I would like to thank my family who stood behind me all these years during my degree. Finally, I would like to thank all my teachers, educational administrators, present, and past and all who helped me achieve this academic goal.

CHAPTER 1

INTRODUCTION

In computer science, algorithms play a crucial role. Time and space complexities are major concern in terms of CPU and memory usage, performance and efficiency. The performance of the developed system is determined by the efficient algorithm used by it. There are various algorithms available for various purposes. It is well know that n objects drawn from an arbitrary totally ordered universe can be sorted by n processors in O(log n) time, even given a very weak model of parallel computation such as the processor network of bounded degree (assuming, of course, that binary comparisons take unit time). This result is optimal in the sense that the product of the number of processors and the time used is O(n log n), to be compared with a lower time bound of Ω(n log n) for any sequential algorithm operating according to the decision-tree model. Reif obtained a partial solution by giving a probabilistic algorithm that sorts n integers in the range {1, ... n}, uses O(n/logn) processors, and terminates within O(log n) steps with high probability. Some doubt remains as to whether his algorithm is able to sort larger numbers, the question hinging on whether the sorting can be made stable. We investigate the restricted sorting problem in a deterministic setting. The algorithms we have developed has overcome the memory usage and performance issues. To achieve this I have used the recent best results of Parallel binary search with delayed read conflicts [10] by H.Meijer and S.G.Akl where search can be done in O(logn/logw) time where w is the number of bits in a word. The computation model used in both the paper's is the CRCW (Concurrent Read Concurrent Write) PRAM (Parallel Random-Access Machine) Model. On the CRCW PRAM memory is shared among processors and multiple processors

can read the same memory cell in one step and can write to the same memory cell in one step. When concurrent write happens, we use the Priority CRCW PRAM in which when multiple processors write the same cell in one step the highest indexed processor wins the write.

In the part one, we are going to construct a search tree for the sorted n input integers using a trie. Such a tree can be constructed in O(nloglogm/p+loglogm) time for n integers in {0, 1, …, m-1} using p processors on the CRCW (Concurrent Read Concurrent Write) PRAM (Parallel Random-Access Machine) [8]. We will add O(loglogm) nodes for each node in the tree to facilitate searching. After the tree is built then searching among these n input integers can be done in O(loglogm) time. The technique presented in this paper has been used to achieve an O(n/p+logloglogm) time CRCW PRAM merging algorithm [7].

This has been published as a research paper to "International Conference on Information Technology 2018, Bhubaneshwar, India". The paper was accepted to the conference with Paper Id 2.

In Algorithm 2, we show that n integers in {0, 1, …, m-1} can be sorted into a linked list in constant time using nlogm processors on the Priority CRCW PRAM model, and they can be sorted into a linked list in O(loglogm/logt) time using nt processors on the Priority CRCW PRAM model.

We use a trie of height logm to sort integers into a linked list. We use A[i][j] to represent the j-th node of the trie at level i. When i=0, A[0][j] is a node at a leaf. When i>0 then A[i-1][2j] is the left child of A[i][j] and A[i-1][2j+1] is the right child of A[i][j]. 0 is labeled on the

edge from a parent to its left child and 1 is labeled on the edge from a parent to its right child. The label reads from the root of the trie to leaf A[0][j] is the binary representation of j. Without loss of generality we assume that logm is a power of 2 as when this is not true we use the smallest power of 2 greater than logm in place of logm.

When nt processors are used the trie with height logm is divided into t section's and each number a at the leaf of the trie will use 1 of the t processors (concurrently with processors for other leaves of the trie) to write into the A[ilogm/t][a div $2^{ilogm/t}$], i=1, …, t.

CHAPTER 2

SEARCHING IN A SORTED LINKED LIST

2.1 Introduction

This chapter explains the algorithm 1 of my research. Under the guidance of Dr. Yijie Han I have submitted the paper to "The International Conference on Informational Technology, Bhubaneshwar, India". The paper was accepted and presented in the conference with Paper Id 2. In next sections lets understand our work on this topic.

Many researchers have studied the search problem [1][2][3]5][9][10]. It is well known to find the predecessor and successor of a number in a sorted array of n numbers in O(logn) time use binary search. If these numbers are integers, then faster algorithms are known. When numbers are integers search can be done faster. In [10] it is shown that search can be done in O(logn/logw) time where w is the number of bits in a word and in [3] it is shown that search can be done in $O\left(\sqrt{\frac{logn}{loglogn}}\right)$ time.

In this paper we are going to construct a search tree for the sorted n input integers using a trie. Such a tree can be constructed in O(nloglogm/p+loglogm) time for n integers in {0, 1, …, m-1} using p processors on the CRCW (Concurrent Read Concurrent Write) PRAM (Parallel Random Access Machine) [8]. We will add O(loglogm) nodes for each node in the tree to facilitate searching. After the tree is built then searching among these n input integers can be done in O(loglogm) time. The technique presented in this paper has been used to achieve an O(n/p+logloglogloglogm) time CRCW PRAM merging algorithm [7].

2.2 Building the Search Tree

We build a binary tree for the m input integers based on a trie. A trie is a full binary tree with $m=2^k$ leaves 0, 1, …, m-1. Each edge of the trie is labeled with a 0 or a 1. The labels on the edges from the root to leaf e is the binary representation of e. An internal node e of the trie is labeled by the labels reads from the root to e. Thus the trie has height k=logm. The level of a node e in the trie is the length of the path from e to one of its leaves. Integer i will be placed at bucket i in the trie. After we place all input integers at the leaves of the trie we can delete all internal nodes of the trie that has only one child (except the root), then the resulting tree is the base tree we want to construct.

For example, let m=16 and the input integers are 3, 6, 10, 11. The trie is shown in fig. 1.

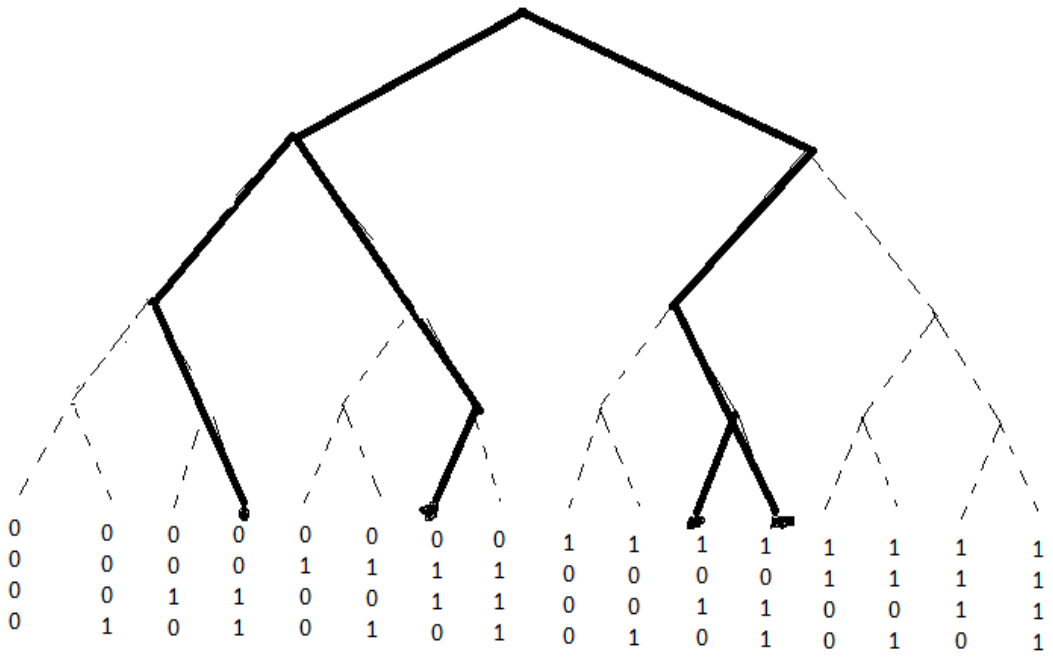

fig.1: Example of a Trie

A parallel algorithm is given in [6] to shown how to sort input integers into a linked list in O(loglogm) time using n processors on the CRCW PRAM. Sorting integers into a linked list is to link the integers in the increasing order. The basic structure of the algorithm in [6] is built on a trie. Let $m=2^k$. The algorithm in [6] use the m leaves of the trie as buckets and first drop the n input integers into buckets and thus integer i is dropped into bucket A[0][i]. One processor is associated with each input integer. Then integer i tries to write into node A[logm/2][i div 2^{(logm/2)}] in the trie. This node is in the middle level of the trie. Multiple leaves could write into the same node and concurrent write is used to allow the highest index processor that attempts to write into the node succeeds in writing and other processors fail. This partition the sorting problem in the trie into multiple partitions. The integers win the write will move up and work on the upper trie with height logm/2. The integers lose the write will move down and work on the lower tries, as shown in fig.2.

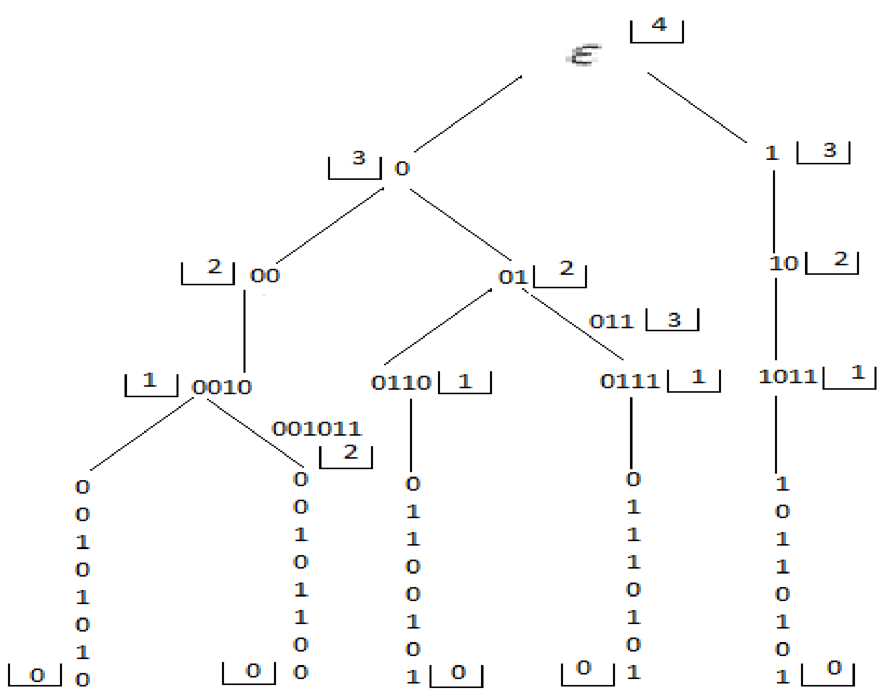

fig.2: Trie with Step numbers

Recursion is then used to solve the sorting problem in the upper trie and in the lower tries. When recursion return's we assume that nodes in the upper trie is linked into a linked list and the nodes in each lower trie also have been linked into a linked list. If there are nodes in a lower trie then one of them, integer e, went up in the concurrent write before. e is now compared with all the integers in this lower trie. Because all the remain integers in this lower trie have been linked into a linked list and therefore e can find its insertion point in the linked list. The linked list in the upper trie is then used to link the linked lists in the lower tries into one linked list. Except the recursion the time used in the previous paragraph and this paragraph is constant with n processors. Because there are loglogm levels of recursion this algorithm [6] sorts n integers into a linked list in O(loglogm) time.

To build the tree needed by us we can, in the recursion, build the tree in the upper trie and each of the lower tries. Then we have to insert the integer e that wins the write and went up in the upper trie into the tree built in the lower trie. This is done by first inserting e into the linked list in the lower trie. Then compare e with its two neighbors b_1 and b_2 in the linked list. We do c_1=e XOR b_1 and c_2=e XOR b_2, where XOR is the bitwise exclusive-or operation. If $c_1 < c_2$ we will let $c=c_1$ and $b=b_1$ else let $c=c_2$and $b=b_2$. Let the i-th bit (counting from the least significant bit starting at 0^{th} bit) of c be the most significant bit of c that is 1. Then we need to insert the node $d = A[i+1]\lfloor b/2^{i+1} \rfloor$ in the trie as the parent of e into the tree. d's other child is the ancestor of b at the highest level less than i+1.

2.3 Augmenting the Tree

The tree we built in Section 2.1 needs to be augmented for the searching purpose. Let e be a node in the tree and p(e) be its parent. Let e be at level l(e) of the trie and p(e) at level l(p(e)) of the trie. If $l(p(e))-l(e)=2^k$ and $l(e) \bmod 2^k=0$ then we do not need add any node between e and p(e). If $2^k< l(p(e))-l(e) < 2^{k+1}$ or $l(p(e))-l(e)=2^k$ and $l(e) \bmod 2^k \neq 0$ then we need to add additional nodes between e and p(e). Let a_1 be the node with $l(a_1) \bmod 2^k =0$ and a_1 is a descendent of p(e) and an ancestor of e in the trie. We add a_1 as the child of p(e) and the parent of e into the tree. Let $b=b_k b_{k-1} b_{k-2} \ldots b_1 b_0$ be the binary representation of $l(p(e))-l(a_1)$. Let b_{t1} be the most significant bit of b that is 1. We will add node a_2 with $l(a_2)=l(a_1)+2^{b_{t1}}$ and in the trie a_2 is a descendant of p(e) and an ancestor of a_1 to the tree. p(e) will become the parent of a_2 and a_1 will become the child of a_2. Let b_{t2} be the second most significant bit of b that is 1. We will add a_3 at level $l(a_1)+2^{b_{t1}}+2^{b_{t2}}$ and in the trie a_3 is a descendant of p(e) and an ancestor of a_2 into the tree. a_2 will become the child of a_3 and p(e) will become the parent of a_3, and so on. Thus if there are c bits in b that are 1's then we will added c nodes between e and p(e). (for the least significant bit in b that is 1 we do not need to add a node because this node is p(e).) We have thus added at most loglogm nodes for each node e in the tree. When we use n processors these nodes can be added in O(loglogm) time.

The fig. 3 shows an example how the tree is augmented. Each node e in the tree has to know the smallest leaf (leftmost leaf) and the largest leaf (rightmost leaf) for the subtree rooted at e. We can use two passes to gain this information. The first pass will build the linked list and in the second pass when integers in a lower trie try to write the node in the

middle level of the trie we will let the smallest leaf and the largest leaf on the linked list in the subtrie write. This will have the leftmost and rightmost leave in the subtree write the information to the root of the subtree.

2.4 Searching

After the tree is built, for a given input integer e we can find the predecessor of e, the successor of e in the tree in O(loglogm) time. We can also insert e into the tree or delete e from the tree in O(loglogm) time. To find the predecessor (successor) for e, we first visit $A[i$ $A[i+1]\lfloor e/2^{(logm)/2} \rfloor$ in the trie. That is, we visit the node that is the ancestor of e in the middle level of the trie. If there is a tree node there we then go down and visit $\lfloor e/2^{(logm)/4} \rfloor$,else we then go up andvisit$\lfloor e/2^{(3logm)/4} \rfloor$. Each time the range of the levels of trie will be cut by half. Thus, in O(loglogm) time we will find the node b in the tree that e branches out. If e branches to the right, then the rightmost leaf r of the tree rooted at b is the predecessor of e and the successor of r is the successor of e. If e branches to the left then the leftmost leaf r of the tree rooted at b is the successor of e and the predecessor of r is the predecessor of e. To delete node e, we first find e in the tree in O(loglogm) time. We then delete e. The supplemental nodes added for e should also be deleted. Now e's parent p(e) should have another descendant b and supplemental nodes added for b. These supplement nodes and p(e) need to be deleted if $2^k \leq$ l(p(e))-l(b) $<2^{k+1}$ and l(p(e)) mod $2^k \neq 0$. Now O(loglogm) nodes may need to be added between b and b's current parent. To insert a node e, we first find the predecessor p of e and the successor s of e. We then insert e in to the tree just as we insert the integer that went up and to be inserted into the lower trie.

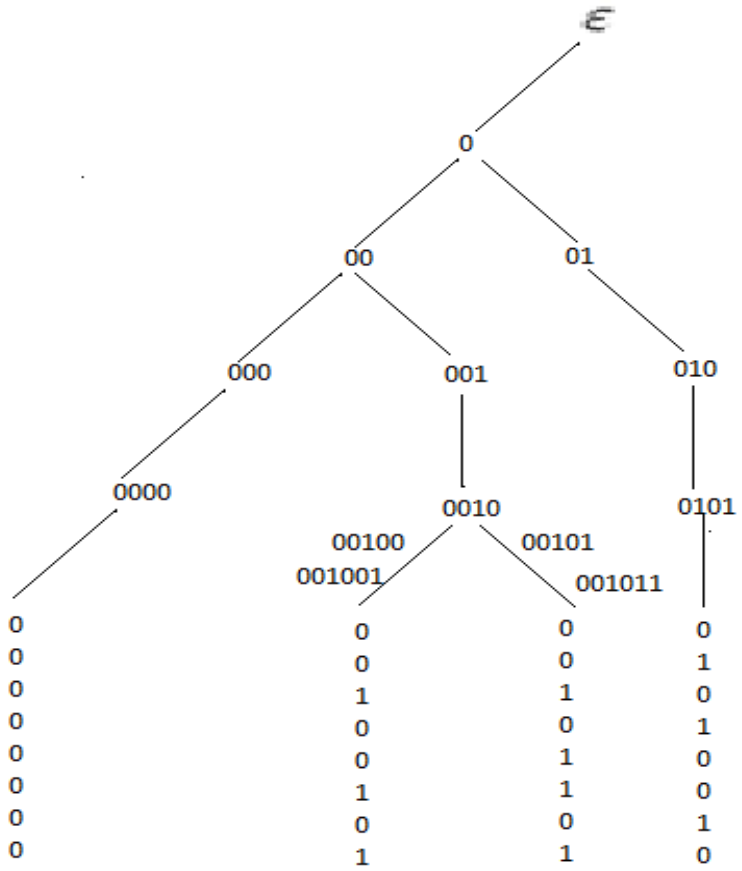

fig.3: Final Trie

CHAPTER 3

SORT INTEGERS INTO A LINKED LIST

3.1 Introduction

It is well known that Ω(logn/loglogn) is a time lower bound for sorting integers [1]. However, if we sort integers into a linked list this lower bound needs not hold. Sorting integers into a linked list is to let smaller integers precedes larger integers in the linked list. As in approximate sorting [23][24] we may allow padding when sort integers into a linked list. It is known that n 0-1's can be sorted into a linked list by chaining 0's into a linked list and 1's into another linked list. This can be done in α(n) time using n/α(n) processors [25], where α(n) is the inverse Ackermann function. Sort padded 0-1 into a linked list takes constant time with n processors. This can be done by making a dummy 0 for each 1 and a dummy 1 for each 0 and then chaining 0's and dummy 0's into a linked list and 1's and dummy 1's into another linked list. Sort integers into a linked list has resulted faster and efficient parallel algorithms for sorting integers in an array [19]. In this paper we show that n integers in {0, 1, …, m-1} can be sorted into a linked list in constant time using nlogm processors on the Priority CRCW PRAM model, and they can be sorted into a linked list in O(loglogm/logt) time using nt processors on the Priority CRCW PRAM model. The computation model used in this paper is the CRCW (Concurrent Read Concurrent Write) PRAM (Parallel Random-Access Machine) Model [8]. On the CRCW PRAM memory is shared among processors and multiple processors can read the same memory cell in one step and can write to the same memory cell in one step. When concurrent write happens, we use the Priority CRCW PRAM [6] in which when multiple processors write the same cell in one step the highest indexed processor wins the write. We use a trie of height logm to sort integers into a linked list. We use A[i][j] to represent the j-th node of the trie at

level i. When i=0, A[0][j] is a node at a leaf of the trie. When i>0 then A[i-1][2j] is the left child of A[i][j] and A[i-1][2j+1] is the right child of A[i][j]. 0 is labeled on the edge from a parent to its left child and 1 is labeled on the edge from a parent to its right child. The label reads from the root of the trie to leaf A[0][j] is the binary representation of j. Without loss of generality we assume that logm is a power of 2 as when this is not true, we use the smallest power of 2 greater than logm in place of logm. A trie of 16 leaves is shown in fig. 5. When nt processors are used the trie with height logm is divided into t section's and each number a at the leaf of the trie will use 1 of the t processors (concurrently with processors for other leaves of the trie) to write into the A[ilogm/t][a div $2^{ilogm/t}$], i=1, …, t, where div is the integer division.

3.2 The Algorithm

With nlogm processors

Let I be the input array of n integers in {0, 1, …, m-1}. I[i] is first placed in A[0][I[i]] at the leaf of the trie. We assume that all input integers are distinct for otherwise we will replace I[i] with I[i]*n+i as the input integer. The input integers and processors assigned to them are shown in fig. 4.

Integer	0	5	3	9	4	8
Processor	1	2	3	4	5	6

fig. 4: Processors assigned to each Input Integer

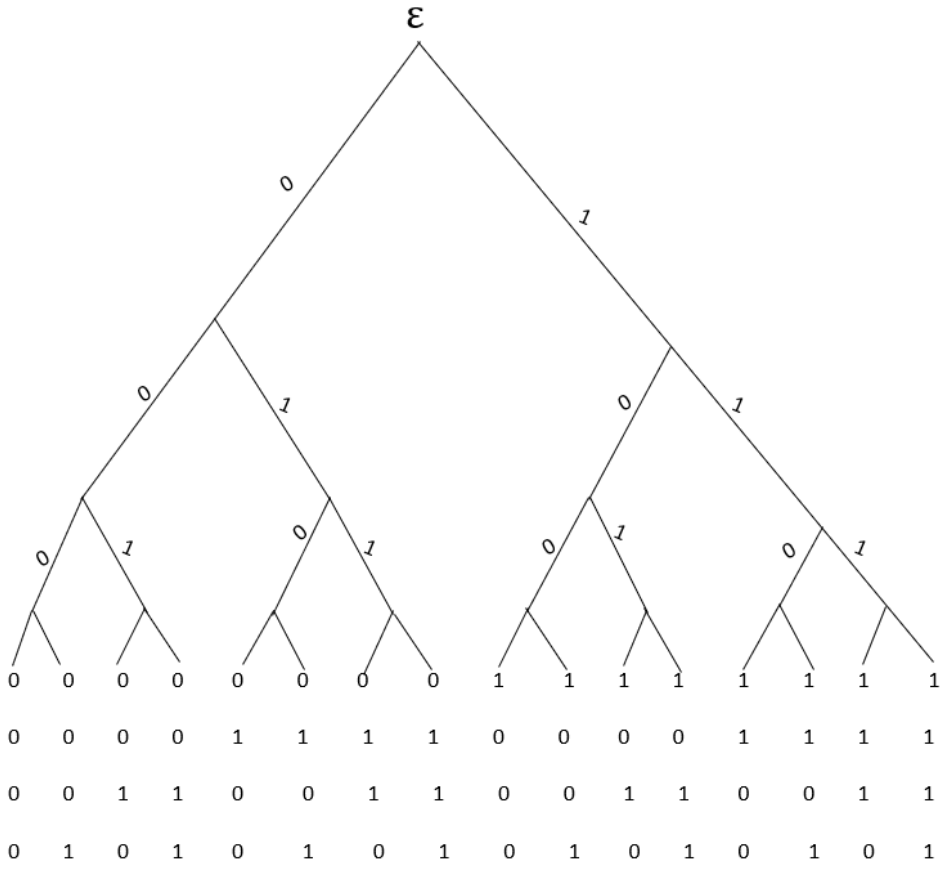

fig. 5: Example of a Trie

We will build a tree for the input integers based on the trie. An interior node of the tree is a node in trie such that the node has a left child and a right child. Node having a single child in the tree is removed. Such a tree is shown in fig 6. The reason such a tree is built is because the tree can facilitate searching and finding the predecessor and successor of an integer [26].

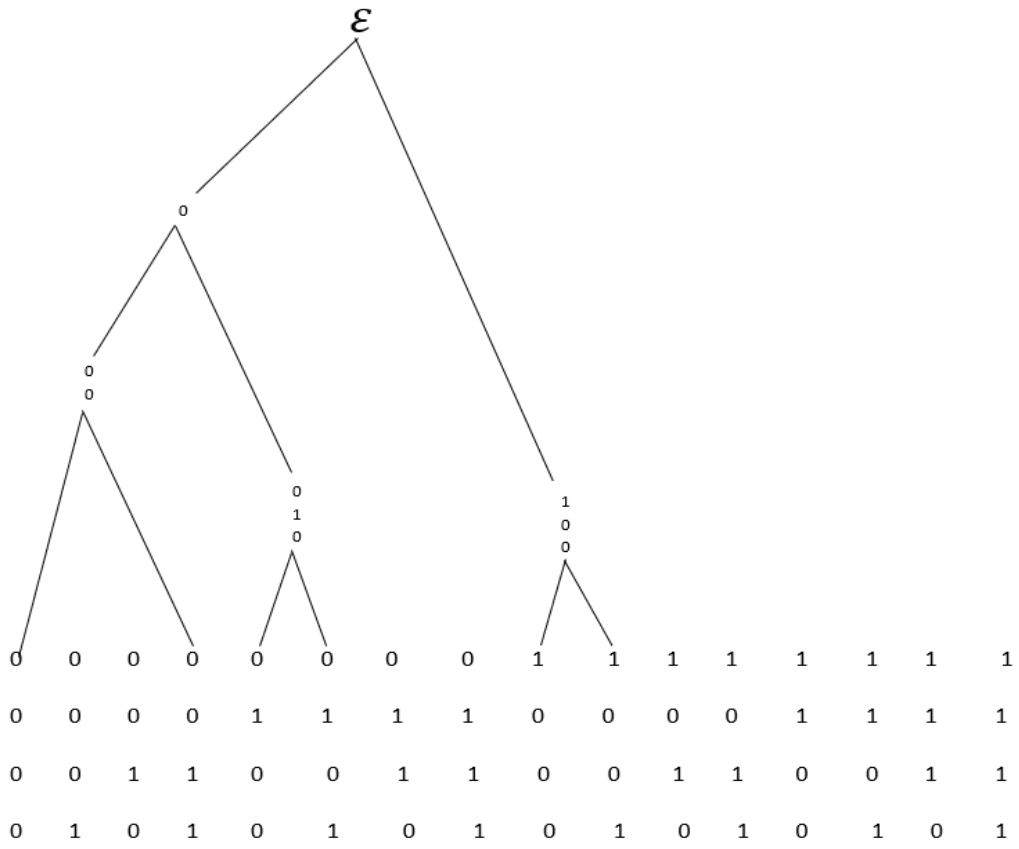

fig.6: The Nodes with one child are removed

This tree is constructed in constant time with nlogm processors on the Priority CRCW PRAM and in O(loglogn/logt) time with nt processors on the Priority CRCW PRAM. When we use nlogm processors we allocate logm processors for each input integer. I[i] will use the j-th processor, $1 \leq j \leq \log m$, at A[j][I[i] div 2^j], where a div b = $\lfloor a/b \rfloor$. Processors at A[i][j] will first use concurrent write to write it's processor id (index) into A[i][j]. Then the processor(s) at A[i][j] will check if A[i-1][2j] and A[i-1][2j+1] are written. This determines whether A[i][j] has one child or has two children. We will label A[i][j] with 1 if it has two children and label A[i][j] with 0 if it has one child. The leaves are always labeled with 1. This situation is depicted in fig. 7.

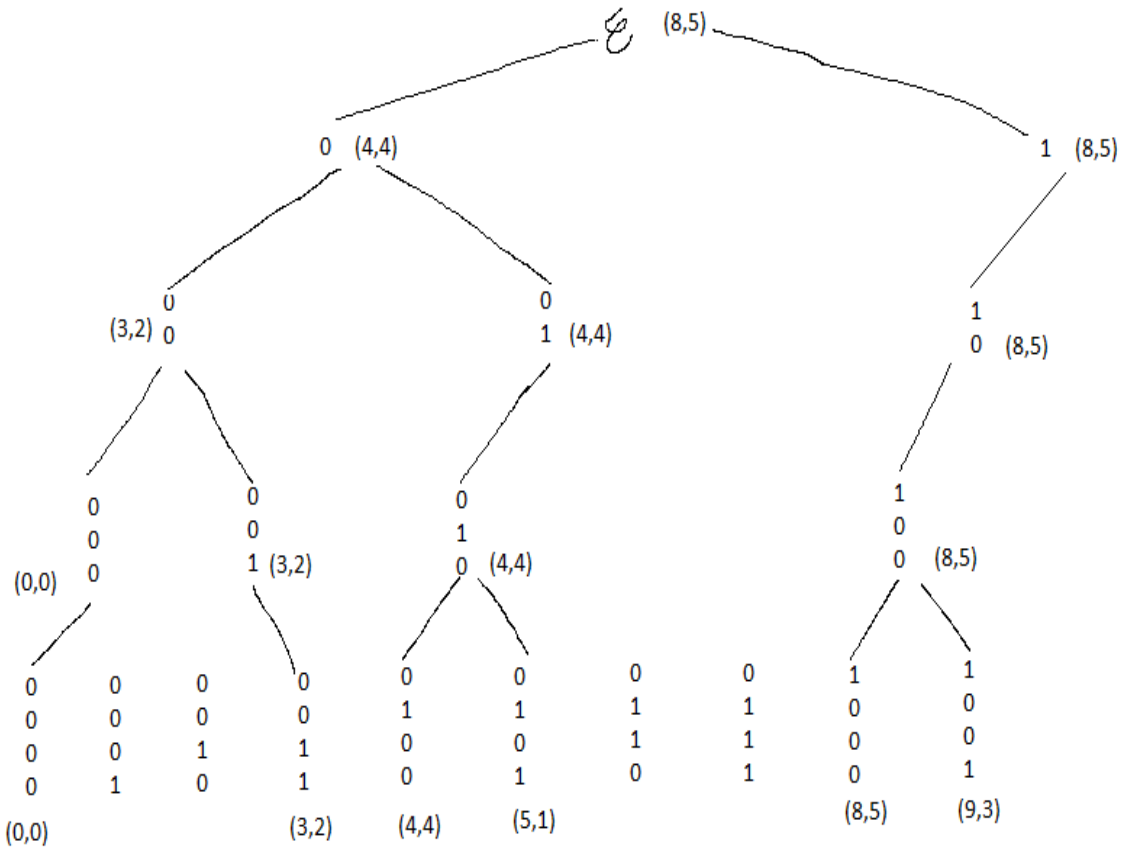

fig 7: All the Nodes are shown

Now for the root of A and each node in A that is labeled with 1 we need to find its nearest descendants that are labeled with 1. Say A[i][j] is labeled with 1 and processor p wins the concurrent write at A[i][j]. Let A[i'][j'] and A[i''][j''] are the two nearest descendants of A[i][j] that are labeled with 1's. If an integer a is a leaf of A[i'][j'] (A[i''][j'']) then because we use Priority CRCW PRAM the processors associated with a win the write at A[i'][j'] (A[i''][j'']) and all the ancestors of A[i'][j'] (A[i''][j'']) up to A[i][j]. And another integer b at the leaf of A[i''][j''] (A[I'][j']) and the processors associated with b win the write at A[i''][j''] (A[I'][j']) and all ancestors of A[i''][j''] (A[I'][j']) up to A[i][j]. If the processor associated with a (b) wins the concurrent write at A[i][j] then we will use the logm processors associated with b (a) to link A[i'][j'] and A[i''][j''] to A[i][j].

To link A[i'][j'] to A[i][j], these logm processors will form an array of size B[0..logm]. B[c], i≤c≤logm, will be set to -1. B[c], 0≤c<I, will be set to c if A[c][j'''] is labeled with 1, where A[c][j'''] is an ancestor of A[i'][j'] in the trie, it will be set to -1 if A[c][j'''] is labeled with 0. Then we use logm processors to find the maximum in array B. This takes constant time with logm processors [2]. The way to link A[i'']][j''] to A[i][j] is similar. Thus, in constant time we build the tree for the input integers. The tree built is shown in fig. 6

To chain the integers into a linked list, we need to let each leaf a in the tree find the lowest ancestor in the tree that has a left (right) child which is not an ancestor of a. For leaf a to find the lowest ancestor in the tree that has a left child which is not an ancestor of a, we will use the logm processors for a, if a's ancestor at level l in trie is not a node in the tree (i.e. it has one child) then processor l will write logm+1 into array B[l]. Processor l will write logm+1 into B[l] also if the ancestor a' of a at level l of the trie has its left child which is an ancestor of a. Otherwise the ancestor a' of a at level l of the trie has its left child which is not an ancestor of a and processor l will write l into B[l]. Then we need to find minimum in array B which takes constant time with logm processors [2]. If leaf a locates b as the lowest ancestor in the tree that has a right child which is not an ancestor of a and a' locates b as the lowest ancestor in the tree that has a left child which is not an ancestor of a'' then we link a to a'. This builds the linked list for the input integers in constant time.

Theorem 1: n integers in {0, 1, …, m-1} can be sorted into a linked list in constant time with nlogm processors on the Priority CRCW PRAM.

As we noted [6] that the tree built here can be augmented to facilitate predecessor and successor queries and insertion in O(loglogm) time.

With nt processors

When we have nt<nlogm processors, we will assign t processors to each input integer. Integer a will be dropped at A[0][a] and the i-th processor for a will write at A[ilogm/t][a div $2^{ilogm/t}$]. Then processors for a will find the highest level in the trie that they win the write. Let this level be level l. Then all the t processors allocated to a will move to A[llogm/t][a div $2^{llogm/t}$]. This cuts the trie into t sections as shown in fig. 8. Now the linked list in each subtrie is built recursively.

After we return from the recursion the linked list for each subtrie is built. We said that processors for integer a was winning at A[llogm/t][a div $2^{llogm/t}$] and now the linked list for the subtrie (with logm/t levels) rooted at A[(l-1)logm/t][a div $2^{(l-1)logm/t}$] is built. Now a uses the b-th processor and processors for the linked list at the subtrie rooted at A[blogm/t][a div $2^{blogm/t}$] to insert it into the linked list at the subtrie rooted at A[blogm/t][a div $2^{blogm/t}$]. Note that if the subtries rooted at A[blogm/t][a div $2^{blogm/t}$] and A[(b+1)logm/t][a div $2^{(b+1)logm/t}$] are empty then a will not insert into the empty linked list for the subtrie rooted at A[blogm/t][a div $2^{blogm/t}$]. Then the linked lists at t different levels will be joined into one linked list. This is done by letting (the largest (smallest) integer in) each linked list in the subtrie rooted at r' find the lowest ancestor having a leftmost right (rightmost left) child which is not an ancestor of r'. Then chaining the linked list as in the previous section. Because there are O(logm/logt) levels of recursion and thus we build the linked list for the input integer in O(logm/logt) time.

Theorem 2: n integers in {0, 1, …, m-1} can be sorted into a linked list in O(logm/logt) time with nt processors on the Priority CRCW PRAM. We can then build the tree for the linked list. This is done by assuming that when the recursion returns both linked list and the tree for the subtries are built. Because the linked list is built we can then insert the (processor associated with the) integer winning the write to the root of the subtrie back into the linked list in constant time by comparing all integers in the linked list with this winning integer. After inserting into the linked list we can compare this winning integer with its two neighboring integers to determining the lowest ancestor of this winning integer with its two neighbors. Thus this winning integer can be inserted into the tree in constant time. Then (the largest (smallest) integer in) in the tree of the subtrie rooted at r' find the lowest ancestors a' (a'') that has a leftmost right child c' (rightmost left child c'') which is not an ancestor of r'. The root of the tree in the subtrie rooted at r' will link to either a' or a'', whichever is at lower levels of the trie. This builds the tree for the trie.

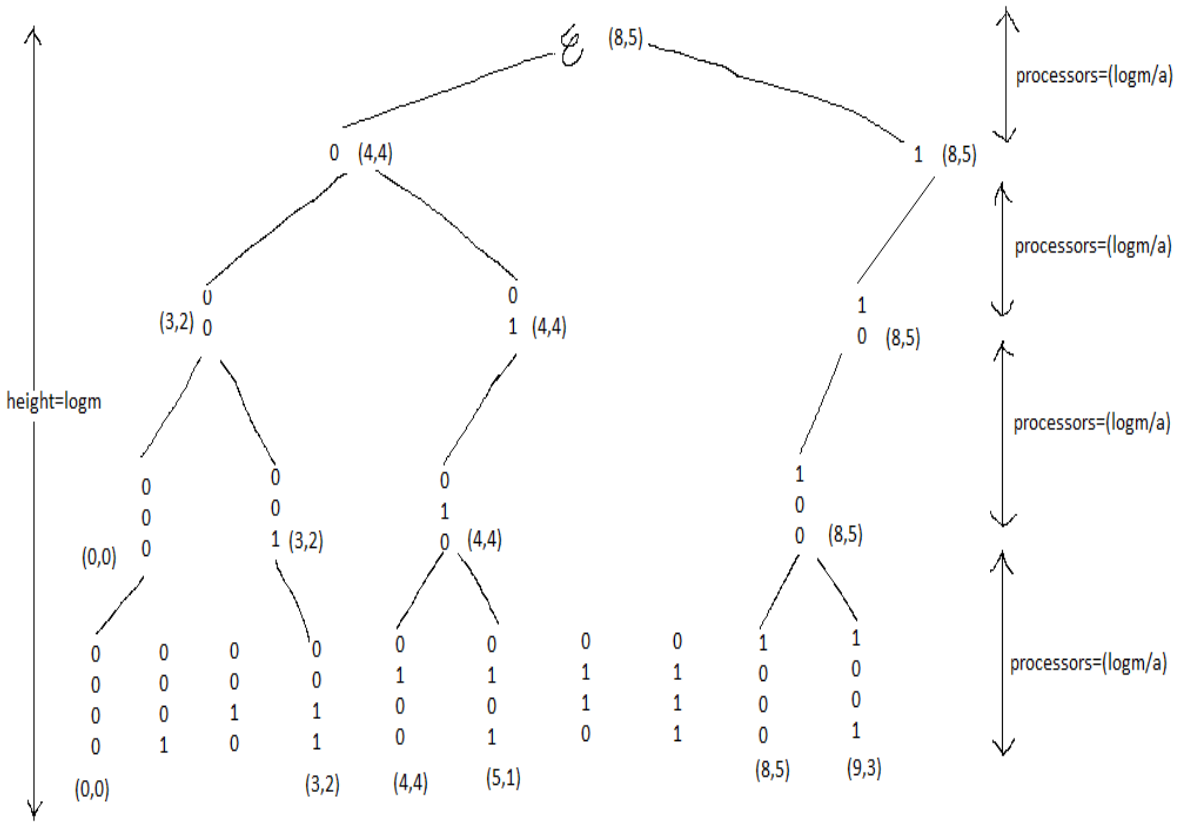

fig.8: Trie with ‘a’ sections

3.3 Building a Tree Based on a Trie: An Example

We will build a binary tree for the n input integers based on a trie. For suppose we have 6 input integers 0,5,3,9,4 and 8 we assign a processor to each integer as shown in Fig. 2. and the processor will drop the integer at it’s position as in Fig 4. Priority CRCW approach is used and the highest indexed processor will win the write and move upward. The same approach is used at every level and the highest index processor with its respective integer will reach the root like shown in Fig 2. The height of the trie will be logm. The intermediate nodes are removed and only the highest-level node will be placed in the tree as in Fig. 9.

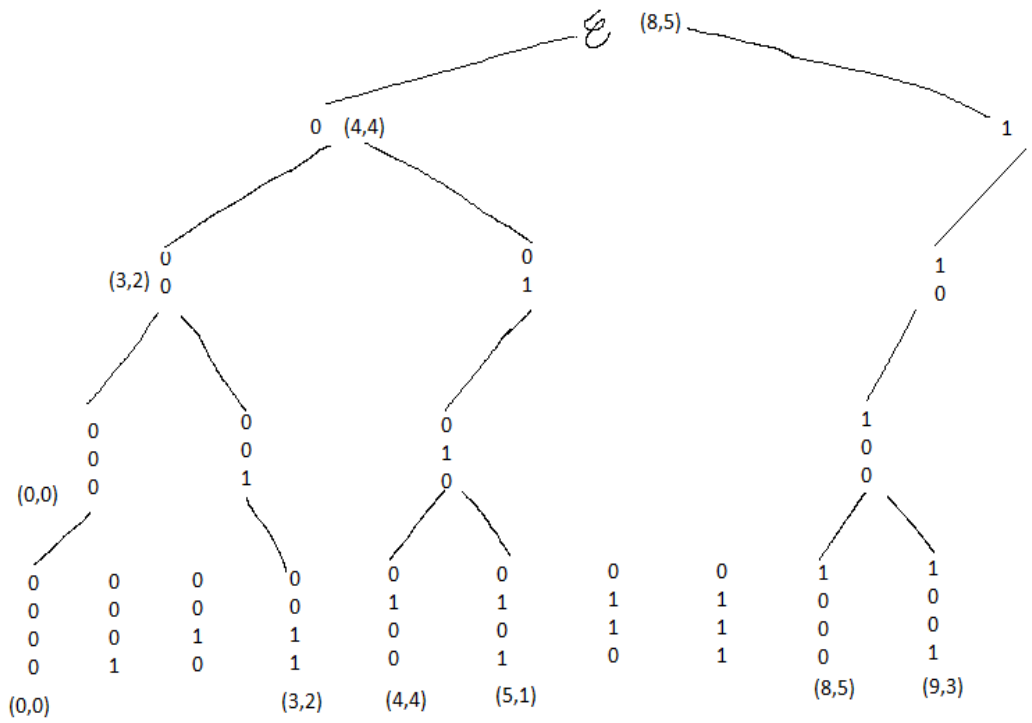

fig. 9: Intermediate levels are removed

For example (4,4) at nodes ‘010’ and ‘00’ can be removed and kept only at node ‘0’. At each level a linked list is created if there are any nodes like fig 10.

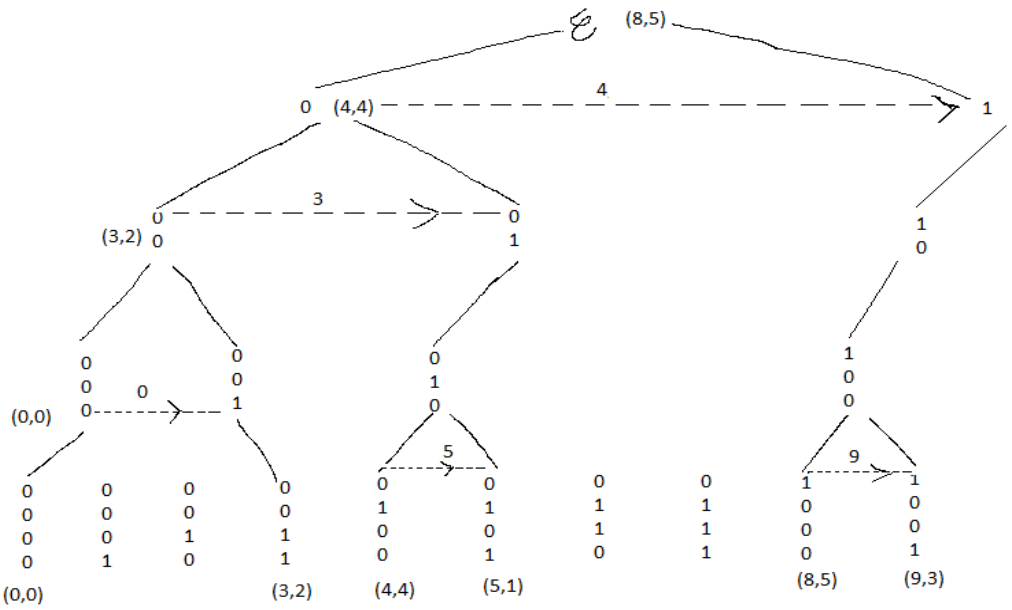

fig. 10: Linked Lists are formed at the Nodes

The root level processor is responsible for linking it's child nodes into a linked list. In the fig. 9, the root node 8 and it's processor 5 are responsible for linking the next level node (4,4) into a linked list. The parent node will be inserted in the child linked list and will connect the tail of left child with the head of right child linked list like in figure 11.

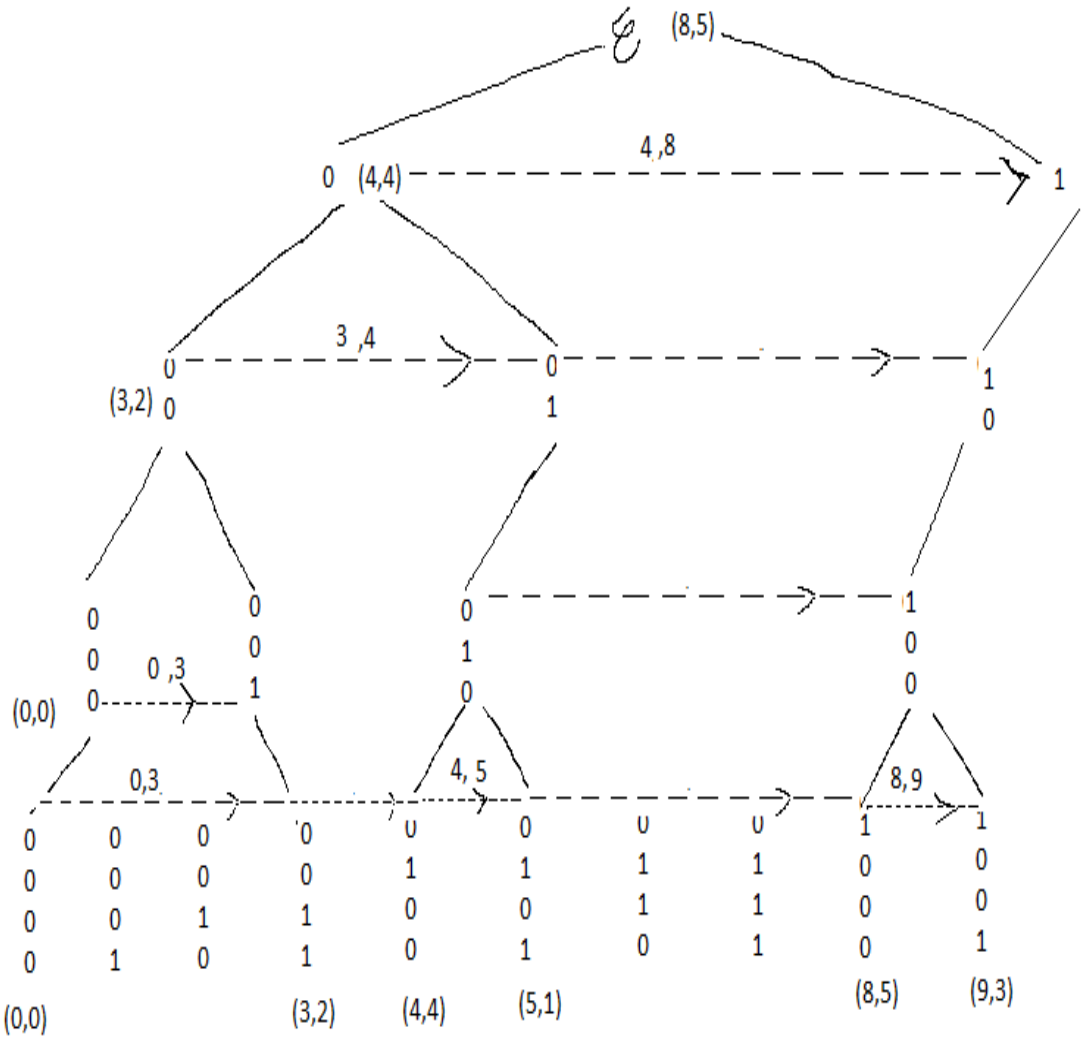

fig.11: Parent node connects all its child Linked Lists

The processors of the integers in the linked list will compare the new integer with its respective number and place the new integer in it's correct position, in that way the linked list is sorted and contain all the input integers. Using nlogm processors we can form a sorted linked list in constant time with priority CRCW approach. As another approach the trie is divided into a section's, each section has (logm/t) levels and each section is given with t processors for each integer as shown in Fig. 3. Using priority concurrent read concurrent write approach with (lglgm/lgt) time the integers can be linked to a linked list in constant time.

CHAPTER 6

CONCLUSION

In both the algorithms an important feature demonstrated by us is that for the searching purpose we need not to sort the input integers into a sorted array which need at least Ω(logn/loglogn) time. Sorting input integers into a linked list can be done in O(loglogm) time with n processors [6]. In fact, if nlogm processors are available n input integers can be sorted into a linked list in constant time. On the positive side the algorithm is simple and easy to program, it has no hidden factors and is fast in practical terms. It is uniform and robust that is for a large value of n this algorithm can efficiently search for the given integer and insert that in the tree accordingly in O(loglogm) time.

In algorithm1 for n integers a tree can be constructed in O(nloglogm/p+loglogm) time with p processors based on the trie with all the given integers. Additional nodes (O(nloglogm) of them) are added to the tree. After the tree is constructed, we can for any given integer, find the predecessor and successor of this integer, insert or delete the integer in A in O(loglogm) time. This result demonstrates for the searching purpose we need not to sort the input numbers into a sorted array for this would need at least O(logn/loglogn) time while this algorithm for constructing the tree can run in O(loglogm) time with n processors. If n processors are available, m integers can be sorted into a linked list in constant time at each level. If there is a tree node there we then go down and visit $\lfloor e/2\text{^}((logm)/4) \rfloor$, else we then go up and visit $\lfloor e/2\text{^}((3logm)/4) \rfloor$. Each time the range of the levels of trie will be cut by half. Thus in O(loglogm) time we will find the node b in the trie that e branches out.

The algorithm in part two demonstrates that if nlogm processors are available n input integers can be sorted into a linked list in constant time with priority CRCW approach. We can also divide the height of the tree into l sections and with ln processors the chaining can be done in (loglogm)/(loga) time with recursive priority CRCW approach.

REFERENCES

[1] S. G. Akl and H. Meijer. “Parallel binary search”, *IEEE funs. Parallel and Distributed Systems*, Vol. 1 No.2, 247-250(1990).

[2] A. Andersson. “Faster deterministic sorting and searching in linear space”. *Proc. 1996 IEEE Int. Conf. on Foundations of Computer Science (FOCS'1996)*,1996, pp. 135-141

[3] A. Andersson, M. Thorup. “Tight(er) worst-case bounds on dynamic searching and priority queues”. *Proc. 2000 ACM Syposium on Theory of Computing (STOC'2000) ,*2000, pp. 335-342.

[4] R. P. Brent. "The parallel evaluation of general arithmetic expressions." *J. of the ACM vol.21 (2) ,*1974, pp. 201-206. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.9361&rep=rep1&type=pdf

[5] D. Z. Chen “Efficient Parallel Binary Search on Sorted Arrays”, Department of Computer Science technical report, 90-1009, Purdue University, August 1990

[6] T. Hagerup. “Toward optimal parallel bucket sorting”. *Information and Computation, 75,* 1987, pp. 39-51.

[7] Y. Han. CS 5590PA.Class Lecture, Topic: “Merge in O(loglogloglogs) time and linear operations”, Faculty of School of Computing and Engineering, University of Missouri Kansas City, Kansas City, Missouri, Mar. 21 ,2018.

[8] R. M. Karp, V. Ramachandran. “Parallel algorithms for shared-memory machines”. In *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*, J. van Leeuwen, Ed., New York, NY: Elsevier, 1991, pp. 869-941.

[9] H. Meijer, S. G. Akl. “Parallel binary search with delayed read conflicts”. Tech. *Rep. 89·257*, Department of Computer and Information Sciences, Queen's University, Kingston,

Ont., Canada, June 1989.

[10] M. Phtra`cu, M. Thorup. “Dynamic integer sets with optimal rank, select, and predecessor search”. *Proc. 2014 IEEE Symp. on Foundations of Computer Science*, 2014, pp. 166-175.

[11] P.C.P Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, S. Saxena.” Improved deterministic parallel integer sorting”. *Information and Computation, 94, 1*, 1991, pp. 29-47.

[12] R. Cole. 1988.” Parallel merge sort”. *SIAM J. Comput., Vol. 17, No. 4,* 1988, pp. 770-785(1988). Correction: “Parallel merge sort”. *SIAM J. Comput., 22(6),* 1993, pp. 1349.

[13] T.H. Corman, C.E. Leiserson, R.L. Rivest, C. Stein. “Introduction to Algorithms”. *Third Edition, The MIT Press,* 1991.

[14] T. Hagerup. “Towards optimal parallel bucket sorting”. *Information and Computation, 73*, 1987, pp. 39-51.

[15] Y. Han. “Sort real numbers in O(n√logn) time and linear space”. *In arXiv.org with paper id 1801.00776,* 2017.

[16] Y. Han. “Deterministic sorting in O(nloglog n) time and linear space”. *Journal of Algorithms, 50*, 2004, pp. 96-105.

[17] Y. Han. “A linear time algorithm for ordered partition”. *Proc. 2015 International Frontiers in Algorithmics Workshop (FAW'15), LNCS 9130,* 2015, pp. 89-103.

[18] Y. Han, X. Shen. “Conservative algorithms for parallel and sequential integer sorting”. *Proc. 1995 International Computing and Combinatorics Conference, Lecture Notes in Computer Science 959*, 1995, pp. 324-333.

[19] Y. Han, X. Shen. “Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs”. *Proc. 1999 Tenth Annual ACM-SIAM Symposium on*

Discrete Algorithms (SODA'99), Baltimore, Maryland, 1999, pp. 419-428.

[20] C. P. Kruskal. “Searching, merging, and sorting in parallel computation”. *IEEE Trans. Comput., C-32,*1983, pp. 942-946.

[21] S. Saxena, P. Chandra, P. Bhatt, V.C. Prasad. “On parallel prefix computation”. *Parallel Processing Letters 4,* 1994, pp. 429-436.

[22] L. G. Valiant. “Parallelism in comparison problems”. *SIAM J. Comput., 4,* 1975, pp. 348-355.

[23] T. Goldberg, U. Zwick. "Optimal deterministic approximate parallel prefix sums and their applications". *Proceedings 1995 Israel Symposium on the Theory of Computing and Systems*, 1995, pp. 220-228.

[24] T. Hagerup, R. Raman. “Fast deterministic approximate and exact parallel sorting”. *In Proceedings of the 5th Annual ACM Symposium on Parallel algorithms and architectures, Velen, Germany*, 1993, pp. 346-355.

[25] P. Ragde. “The parallel simplicity of compaction and chaining”. *Journal of Algorithms, 14(3)*, 1993, pp. 371-380.

[26] Y. Han, H. Koganti. “Searching in a Sorted Linked List”. *In Proceedings of the 17th Int. Conf. on Information Technology (ICIT'2018),* 2018.

VITA

Hemasree Koganti was born on December 3rd, 1993, in Andhra Pradesh, India. She completed her bachelor's degree in Computer Science from Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad, India.

After completing her under-graduation, she worked as Software Engineer at Trianz, Hyderabad for more than 2 years. To enhance her career further she started her master's in computer science at the University of Missouri-Kansas City (UMKC) in Spring 18, specializing in Data Science emphasis. Upon completion of her requirements for the master's Program, Hemasree Koganti plans to work as a Data Scientist.