Data Structuring Problems in the Bit Probe Model

by

Mohammad Ziaur Rahman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Mohammad Ziaur Rahman

Abstract

We study two data structuring problems under the bit probe model: the dynamic predecessor problem and integer representation in a manner supporting basic updates in as few bit operations as possible. The model of computation considered in this paper is the bit probe model. In this model, the complexity measure counts only the bitwise accesses to the data structure. The model ignores the cost of computation. As a result, the bit probe complexity of a data structuring problem can be considered as a fundamental measure of the problem. Lower bounds derived by this model are valid as lower bounds for any realistic, sequential model of computation. Furthermore, some of the problems are more suitable for study in this model as they can be solved using less than w bit probes where w is the size of a computer word.

The predecessor problem is one of the fundamental problems in computer science with numerous applications and has been studied for several decades. We study the colored predecessor problem, a variation of the predecessor problem, in which each element is associated with a symbol from a finite alphabet or color. The problem is to store a subset S of size n, from a finite universe U so that to support efficient insertion, deletion and queries to determine the color of the largest value in S which is not larger than x, for a given $x \in U$. We present a data structure for the problem that requires $O(k\sqrt[k]{\frac{\log U}{\log\log U}})$ bit probes for the query and $O(k^2\frac{\log U}{\log\log U})$ bit probes for the update operations, where U is the universe size and k is positive constant. We also show that the results on the colored predecessor problem can be used to solve some other related problems such as existential range query, dynamic prefix sum, segment representative, connectivity problems, etc.

The second structure considered is for integer representation. We examine the problem of integer representation in a nearly minimal number of bits so that increment and decrement (and indeed addition and subtraction) can be performed using few bit inspections and fewer bit changes. In particular, we prove a new lower bound of $\Omega(\sqrt{n})$ for the increment and decrement operation, where n is the minimum number of bits required to represent the number. We present several efficient data structures to represent integers that use a logarithmic number of bit inspections and a constant number of bit changes per operation.

Acknowledgements

I wish to thank my excellent supervisor Ian Munro for guiding my research throughout my studies. His encouragement, friendship and financial support are invaluable for me. The results in this thesis also came from the joint work with him.

I like to thank the readers of my thesis, Jérémy Barbay, and Alejandro López-Ortiz, for their valuable comments, suggestions and corrections. I also like to thank Professor Jeffrey Shallit for his suggestions, corrections and comments.

Finally, I would like to express my gratitude to the Cheriton School of Computer Science for providing such a beautiful environment.

Contents

1 Introduction **1**

2 The Colored Predecessor Problem **4**

2.1 The Predecessor Problem 4

2.2 The Colored Predecessor Problem 5

2.2.1 A Naive Solution 6

2.3 Colored Predecessor in $O(\frac{\log U}{\log \log U})$ Bit Probes 7

2.3.1 Find the Largest Element 9

2.3.2 Change the Color of the Largest Element 10

2.3.3 Existence of a Smaller Element in the Set 10

2.4 Colored Predecessor in $O(\sqrt{\frac{\log U}{\log \log U}})$ Bit Probes 10

2.4.1 The Data Structure 11

2.4.2 Colored Predecessor Query 11

2.4.3 Insertion 12

2.4.4 Deletion 13

2.5 Further Improving the Query Time 13

2.5.1 Colored Predecessor Query 14

2.5.2 Insertion 15

2.5.3 Deletion 16

2.6 Faster Queries with Slower Updates 17

2.6.1 The Data Structure 18

2.6.2 Colored Predecessor Query 19

2.6.3 Insertion 19

2.6.4 Deletion 20

3 Application of the Colored Predecessor Problem 21
3.1 Segment Representative Problem 21
3.2 Existential Range Queries 23
3.3 Dynamic Prefix Sum Problem 24
3.4 Dynamic Connectivity Problem 25

4 Integers and Their Representations 27
4.1 Preliminaries 27
4.2 Standard Binary Representation 28
4.3 Binary Reflected Gray Code 29
4.4 The Tree Representation 30
4.4.1 Increment Operation in a Logarithmic number of Bit Probes 31
4.4.2 Increment Using a Constant Number of Bit Changes 34

5 Integers and Counting 38
5.1 The Lower Bound 38
5.2 Properties of Binary Reflected Gray Code Sequence 40
5.3 Efficient Increment and Decrement 41
5.3.1 The Data Structure 41
5.3.2 The Increment Operation 43
5.3.3 The Decrement Operation 45
5.4 Increment and Decrement Using Fewer Bit Inspections 46
5.4.1 The Data Structure 47
5.4.2 Increment and Decrement 48
5.5 Supporting Addition and Subtraction 50
5.5.1 Gray Code Addition 50
5.5.2 Addition with Different Size Operands 50
5.5.3 The Data Structure 53
5.5.4 Addition and Subtraction 54

6 Conclusion 56

A Increment Using a Constant Number of Bit Changes **58**

B Efficient Increment Algorithm **62**

List of Figures

2.1 A naive solution of the colored predecessor problem. 7
2.2 A $O(\frac{\log U}{\log \log U})$ solution of the colored predecessor problem. 8
2.3 A multi-level structure for the solution of the colored predecessor problem. 11

4.1 Increment and addition using standard binary representation. 29
4.2 The Binary Reflected Gray Code (BRGC) sequence of dimension 5. 30
4.3 Tree Representation of an Integer 32
4.4 Increment operations using logarithmic number of bit changes. 33
4.5 A valid labelling of the tree representation of 1000011111111111. 34
4.6 Increment operations using a constant number of bit changes. 36

5.1 Increment operations using a constant number of bit changes. 44
5.2 Properties of the BRGC sequence 47
5.3 Efficient data structure for integer 48
5.4 Data structure supporting efficient addition/subtraction 53

Chapter 1

Introduction

To optimize algorithm efficiency, it is important to understand the complexity of the problem at hand and to design efficient data structures. We study two data structuring problems under the *bit probe model*: the *dynamic predecessor problem* and *integer representation* in a manner supporting basic updates in as few bit operations as possible.

Efficient algorithms tend to require basic building blocks and one of them is the predecessor problem. The *predecessor problem* is to store a subset S of size n, from a finite universe U so that to support efficient insertion, deletion and queries to determine the largest value in S which is smaller than x, for a given $x \in U$. The predecessor problem is one of the fundamental problems in computer science with numerous applications such as locating the *nearest neighbour*, finding the *rank* of an element etc. and has been studied for several decades [1, 5, 7, 18, 38, 52, 55, 56]. Along with a variation of the predecessor problem, we study some other related data structure problems in this thesis. The problems considered are mainly searching problems such as the *colored predecessor problem* [44, 48], the *existential range query problem* [2, 40, 44], the *segment representative problem* [48], the *prefix sum problem* [25, 38, 48], and graph problems such as the *connectivity problem* [6, 19, 30, 41, 54] and the *reachability problem* [32, 53]. All these problems are extensively studied, mainly in the cell probe model and the word RAM model, and we want to shed some light on these problems from a different perspective by studying them in the bit probe model.

The second structure considered is the *integer representation*. The data type integer

is fundamental in any computer system and any programming language. It is hard to imagine any computer program without integers and operations on them. Therefore, the representation of integers and operations on them are fundamental issues. We are interested in representing integers in the range $[0, 2^n-1]$ in near minimal number of bits, where n is the minimum number of bits required to represent an element of this set, so that increment and decrement (and indeed addition and subtraction) can be performed using few bit inspections and fewer bit changes.

The model of computation plays a central role in analyzing a problem. In this thesis, we consider all the problems in the *bit probe model*. The bit probe model was introduced by Minsky and Papert in the book *Perceptrons* [42], where they discussed the average-case bit probe complexity of the *Membership problem*. In this model, the complexity measure counts the bitwise accesses to the data structure, and ignores the other costs of computation. As a result, the bit probe complexity of a data structure problem can be considered as a fundamental measure of the problem, which focuses on a practical bottleneck, as random memory access in real hardware are becoming orders of magnitude slower than computational instructions. Lower bounds derived in this model are also lower bounds in any realistic, sequential model of computation. Power consumption by processor and memory chips are becoming serious concern with the advance of the semiconductor technology, as more transistors are fitted in smaller area. As fewer bit accesses require less power, the solutions with fewer bit probes has gained interest. Furthermore, some of the problems are more suitable for study in the model as they can be solved using fewer bit probes than the size of a computer word.

Yao generalized the bit probe model into the *cell probe model* [57], where the accesses to the memory cells (words) are counted as a complexity measure. Although the cell probe model is used more frequently for the data structures, the bit probe complexity measure has been a persistent object of study in theoretical computer science. Now we present a brief literature review of the bit probe model. Following the work of Minsky and Papert [42], Elias et al. [16, 17] analyzed the worst case complexity of the membership problem in the bit probe model. Fredman used the decision assignment tree to obtain the bit probe lower bound to generate a quasi-gray code sequence [24]. The bit probe model is appealing because of its architecture independence and overall cleanliness. Furthermore,

it gives a more accurate idea of the complexity of a problem at the bit level such as the number of bits of the data structure probed. In recent years, the bit probe model has gained a lot of interest. Miltersen et al. analyzed the static data structure problems for some explicitly and implicitly defined functions in the bit probe model [37]. Buhrman et al. [10] considered the static membership problem in the bit probe model, for which they were the first to apply the Monte Carlo style randomization. Radhakrishnan et al. [49] presented some constructive schemes for the static membership problem. Miltersen analyzed the redundancies of the rank and select type index structures [39]. Pătraşcu et al. [48] considered predecessor like problems, partial sum problem and graph connectivity problem in the bit probe model.

The thesis is organized as follows. In Chapter 2, we study the *colored predecessor* problem, a variation of the predecessor problem, in which each element is associated with a symbol from a finite alphabet or color. For each query only the color of the predecessor needs to be returned. The key aspect of this variation is that, the answer no longer requires $\log U$ bits and so sublogarithmic methods are explored in the bit probe model. Chapter 3 includes some applications of our solution of the colored predecessor problem. We show that the result of the colored predecessor problem can be used to efficiently solve other related problems such as *existential range query*, *segment representative*, *dynamic prefix sum*, *graph connectivity*. These problems can be solved using fewer number of bit probes than the size of a computer word. In Chapter 4, we study different representations of integer and some basic operations on it. We present a new lower bound for the increment and decrement operation in Chapter 5, and several efficient data structures to represent integer that use a logarithmic number of bit inspections and a constant number of bit changes per operation. Finally, the thesis ends with our conclusion and comments on potential directions for future research.

Chapter 2

The Colored Predecessor Problem

2.1 The Predecessor Problem

The predecessor problem is a basic and heavily studied problem. It has numerous applications in other problems such as searching, sorting, range reporting, dictionary problem etc.

> *The Predecessor Problem*: The predecessor problem is to store a subset S of size n, from a finite universe U under insertion and deletion of elements, in order to support efficient queries to determine the largest value in S which is at most x for $x \in U$. While the term *predecessor* implies a value strictly less that the input parameter, it is notationally much more convenient to consider an element to be a predecessor of itself.

The obvious solution is to use a balanced binary tree that requires $O(n)$ words of space and $O(\log n)$ time per query. Van Emde Boas et al. [18] presented a solution over the universe $[U]$, i.e. with word size $\log U$ bits, with time complexity of $O(\log \log U)$. The basic form of the algorithm uses $\Theta(U)$ bits; but, using the y-fast trie of Willard [55], this can be reduced to the optimal $O(n)$ words. Along the same direction, Andersson [4] improved the time complexity of queries to $O(\sqrt{\log n})$. Miltersen generalized Ajtai's technique of probability amplification in product spaces [1] to get a lower bound of $\Omega(\sqrt{\log \log U})$ on query time [40]. Beame and Fich [7] improved the lower bound to show that the predecessor

problem has a lower bound of time complexity $\Omega(\min(\frac{\log\log U}{\log\log\log U}, \sqrt{\frac{\log n}{\log\log n}}))$ with space requirement $n^{O(1)}$ words in the cell probe model with the cell size $O(\log U)$. They presented a matching upper bound data structure that uses polynomial space in the word RAM model. The same lower bound was obtained independently by Xiao [56]. Sen [52] proved the same lower bound using a different technique. Andersson and Thorup [5] presented a structure that requires $O(\min(\frac{\log\log U}{\log\log\log U}\log\log n, \sqrt{\frac{\log n}{\log\log n}}))$ time per operation using $O(n)$ words of space. The *Q-heap* structure of Fredman and Willard [28] offers a constant time solution to the dynamic predecessor problem with $n \leq (\log U)^{1/4}$ and word size $O(\log U)$. The scheme requires a precomputed table of size $O(U)$ words. The rank/select structure solves the static case in $O(1)$ time using $U + o(U)$ bits [13, 45, 46, 47, 50, 51] under the word RAM model.

2.2 The Colored Predecessor Problem

The predecessor problem has an inherent $\Omega(\log U)$ lower bound under the bit probe model, as $\log U$ bits are required for the output. But some constant information about the predecessor (a representative or a color) can be retrieved in $o(\log U)$ time in this model. We consider the *colored predecessor problem*, where every element in the set has a color chosen from a constant size pool of colors. For a given number x, the color of x's predecessor has to be returned. The bit probe lower bound on time for the problem is $\Omega(\frac{\log U}{\log\log U})$ as shown by Fredman [25]. Pătraşcu and Pătraşcu [48] achieved $O(\frac{\log U}{\log\log U})$ upper bounds for several dynamic predecessor-type problems in the bit probe model. Mortensen et al. [44] have shown a lower bound trade-off of $(\Omega(\log\log U), \Omega(\frac{\log U}{\log\log U}))$ between the query time and the update time of a simpler problem called the *greater than problem*, where the set contains only one element and a query returns whether the element is greater than the query element.

> *The Colored Predecessor Problem*: The colored predecessor problem is to maintain a subset S of n elements from a finite universe U under insertion and deletion of elements, in order to support efficient queries to determine the color of the largest value in S which is equal or smaller than x for $x \in U$.

The insert operation specifies a value from $[U]$ together with a color from a fixed constant sized universe. If the value is already in the set the insert operation is taken to mean that the color is changed. The known trade-off lower bounds in the bit probe model for the *greater than problem* (also known *which-side problem*) are $t_q = \Omega(\log\log U), t_u = \Omega(\frac{\log U}{\log\log U})$ and $t_q = \Omega(\frac{\log U}{\log\log U}), t_u = \Omega(\log\log U)$, where t_q and t_u are the query and update times respectively. Clearly the bounds hold for the dynamic predecessor problem as well.

We first consider a naive solution of the colored predecessor problem that requires $O(\log U)$ bit probes for query and updates. Then we explore several improvements to support queries in $O(k^2\sqrt[k]{\frac{\log U}{\log\log U}})$ bit probes and updates in $O(\frac{k\log U}{\log\log U})$ bit probes and other trade-off between query time and update time, where k is a positive integer.

2.2.1 A Naive Solution

The set S is represented by a complete binary tree on the whole universe with U leaves. For each element of the universe there is a leaf. An element of S is encoded by coloring the corresponding leaf and all its ancestors. When a node has descendants of different colors, it takes the color of the descendant with the largest value.

Consider first a query for the predecessor of $x \in U$. If x is in the set S, we simply look up its color. Otherwise, consider the leaf-to-root path from the leaf corresponding to x. If S contains at least one element, this path contains at least one colored node. Let u be the lowest colored node on this path such that x is in the right branch of u. The answer to the query is the color of the left child of u.

Updates proceed in a similar way. To insert an element x with color c, consider the path from the leaf corresponding to x to the root. Let w be the lowest colored node on the path such that x is in the left branch of w and the right child of w is colored. Set the colors of the nodes on the path from the leaf x to the left child of w to color c. We have a special case when x becomes the largest in S. In that case, there is no w satisfying the condition. Set the colors of all nodes on the path, including the root, with the color of x.

To delete an element $x \in S$, consider the path from the leaf corresponding to x to the root. Let u be the lowest node on the path such that x is in the right branch and the left child of u has a color. Also, let w be the lowest node on the path such that x is in the left branch and the right child of w has a color. If there is no u satisfying the condition or w is

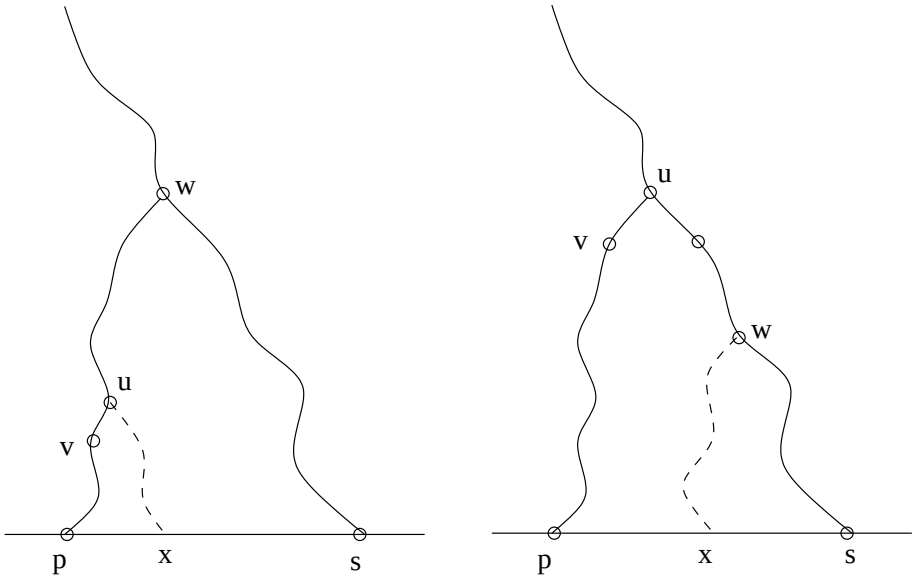

Figure 2.1: A naive solution of the colored predecessor problem.

a descendant of u then, remove the colors of the node on the path from x to the left child of w. On the other hand, if there is no w satisfying the condition or u is a descendant of w then, let v be the left child of u. Remove the colors of all nodes on the path from x to the right child of u and set the color of all nodes from u to the left child of w to the color of v. We have a special case when x is the only element in S. In that case, there is no u or w satisfying the conditions. To delete the element remove the colors of all nodes on the path. This gives an $O(\log U)$ bit probe solution.

2.3 Colored Predecessor in $O(\frac{\log U}{\log \log U})$ Bit Probes

We now describe the solution provided by Pătraşcu and Pătraşcu [48]. Instead of a binary tree, the set S is represented by a trie with branching factor $b = \Theta(\frac{\log U}{\log \log U})$. So, the height of the trie is $h = O(\frac{\log U}{\log \log U})$. As before, an element of S is encoded by coloring the corresponding leaf and all its ancestors. An internal node is colored with the color of the descendant with the largest value. Each node is associated with a bit vector of size b,

one bit per child. The bit is set to '1' if the corresponding child is the smallest among the colored siblings. Similarly, another bit vector marks the largest among the colored siblings.

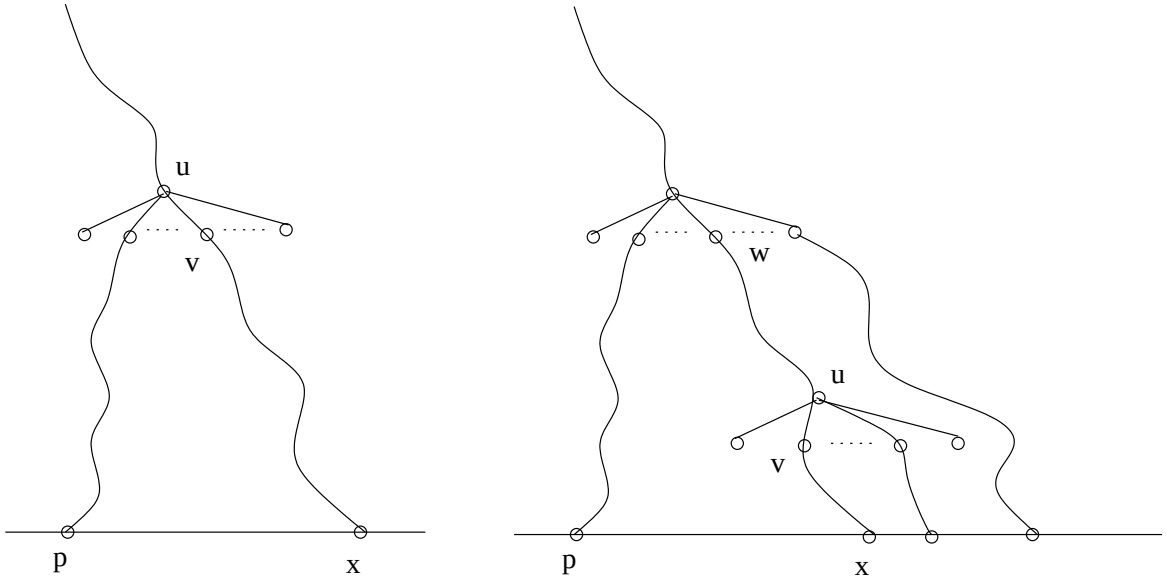

Figure 2.2: A $O(\frac{\log U}{\log \log U})$ solution of the colored predecessor problem.

Consider a query for the predecessor of $x \in U$. If x is in the set S, we simply look up its color. Otherwise, consider the leaf-to-root path from the leaf corresponding to x. If S contains at least one element, this path contains at least one colored node. Let u be the lowest colored node on this path and v be the child of u on the path. Check whether or not there is any colored child of u to the left of the path by inspecting the colors of the children of u. If there is no colored child of u to the left of the path, find w the lowest colored node on the path which is not the smallest among the colored siblings. Since the smallest colored sibling is marked by a bit vector, it takes one bit probe to determine whether or not the current node is the smallest colored node among its siblings. The color of the largest colored sibling to the left of the path is the answer. If x is smaller than any element present in S then, there is no w satisfying the condition and we reach the root without locating w on the path. Climbing the path requires $O(h)$ bit probes. The colors of the siblings are scanned at two levels at most and this requires $O(b)$ bit probes. So, the

query time is $O(h+b) = O(\frac{\log U}{\log \log U})$.

To insert an element $x \in [U]$ with color c, consider the path from the leaf corresponding to x to the root. First consider an insertion of x not in the set S. Let u be the lowest colored node on the path and v be the child of u on the path. Color each node from leaf x up to and including v with color c. Also, set each node as the largest and the smallest among the colored siblings. Scan the colors of the siblings of v to update the largest and the smallest markings if necessary. The color of the ancestors of v might be changed due to the insertion. Let w be the lowest ancestor of v such that w is not the largest among the colored siblings. Change the color of each node from u, the parent of v, to the child of w on the path to color c. On the other hand, consider the insertion of an element $x \in S$. In that case, find w the lowest node on the path from x to the root such that w is not the largest among the colored siblings. Change the color of each node on the path from x to the root to the color of x. Traversing towards the root requires $O(h)$ bit probes. Note that we have to scan the color of the nodes at one level only. Scanning the colors of all siblings at one level takes $O(b)$ time. Changing the color of a node requires constant time. So, the total insertion time is $O(h+b) = O(\frac{\log U}{\log \log U})$. Deletion can be done in a similar way.

We now describe several useful operations on the trie with branching factor b and height h. These operations are useful in designing our data structures described in the following sections.

2.3.1 Find the Largest Element

We have to find the largest element in the set (not only the color). Consider the path from the root to the leaf corresponding to the largest element of the set. All nodes along the path are colored with the color of the largest element. Also the nodes are marked as the largest among the colored siblings. Start from root towards the leaf. At each node, scan the children to find the largest colored child and move to that child. Scanning the nodes at a level requires $O(b)$ time. Total time to reach the leaf corresponding to the largest element is $O(hb)$ bit probes. In a similar way, the smallest element can be determined in $O(hb)$ bit probes.

2.3.2 Change the Color of the Largest Element

To change the color of the largest element to a new color c, consider the leaf to root path from the leaf corresponding to the largest element. All nodes on the path are colored with the color of the largest element. Also, all nodes on the path are marked as the largest among the colored siblings. Set the color of all nodes on the path to c. No other change in the data structure required. So, this operation performs $O(h)$ bit probes.

2.3.3 Existence of a Smaller Element in the Set

For a given element x, the query is asking whether or not there exists an element smaller than x. Consider the path from the leaf corresponding to x to the root. Let u be the lowest colored internal node on the path and v be the child of u on the path. Scan the siblings of v to find the colored left sibling of v if one exists. Otherwise, find the lowest ancestor of v which is not the smallest among the colored siblings using the bit vector used to mark the smallest colored sibling. There is a smaller element if one such node exists. Scanning the siblings is done at one level only. Checking whether a node is the smallest among the colored sibling requires constant bit probes. So, total time for the query is $O(h + b)$ bit probes.

2.4 Colored Predecessor in $O(\sqrt{\frac{\log U}{\log \log U}})$ Bit Probes

We prove the following theorem in this section.

Theorem 2.4.1. *There exists a data structure for the colored predecessor problem with complexity $t_q = O(\sqrt{\frac{\log U}{\log \log U}})$ and $t_u = O(\frac{\log U}{\log \log U})$, where t_q and t_u are the number of bit probes for the query and the update respectively.*

We present a constructive proof for the above theorem. First, we present a data structure for the problem and then study the complexities of the operations on it.

2.4.1 The Data Structure

The set S is represented by a trie $\mathbf{D}$ on the whole universe with U leaves, branching factor $B = \Theta(2^{\sqrt{\log U \log\log U}})$ and height $H = O(\sqrt{\frac{\log U}{\log\log U}})$. As before, an element of S is encoded by coloring the corresponding leaf and all its ancestors. An internal node takes the color of the descendant with the largest value. Each node is associated with a bit vector of size B, one bit per child. A bit is set if the corresponding child is the smallest among the colored siblings. Similarly, another bit vector marks the largest among the colored siblings.

Each internal node u is associated with a supporting data structure and denoted by $\mathbf{D}'(u)$. The structure $\mathbf{D}'(u)$ is a trie with B leaves, branching factor $b = \Theta(\frac{\log B}{\log\log B})$ and height $h = O(\frac{\log B}{\log\log B})$. For each child of $u \in \mathbf{D}$, there is a corresponding leaf in $\mathbf{D}'(u)$.

2.4.2 Colored Predecessor Query

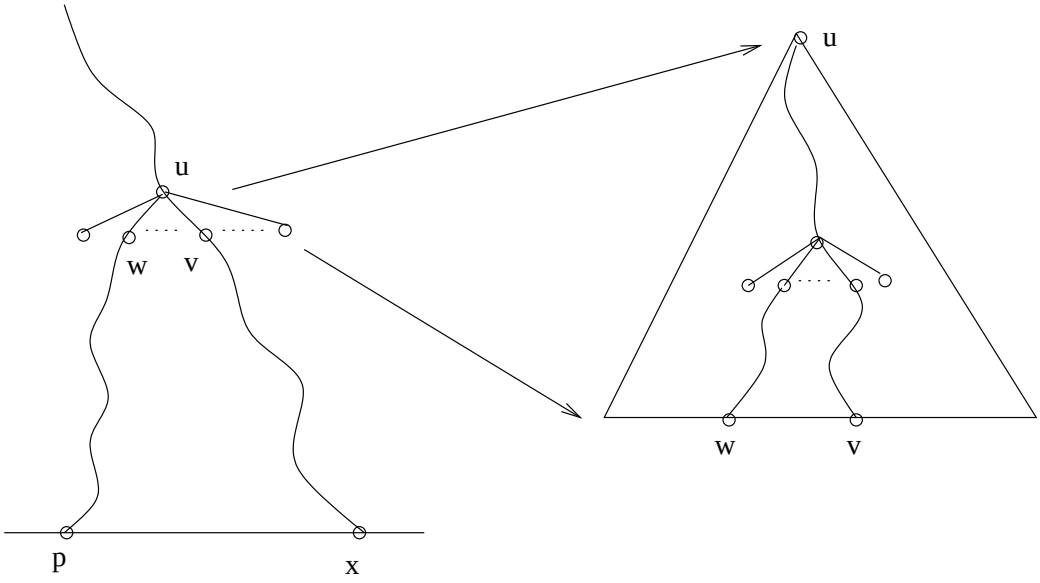

Figure 2.3: A multi-level structure for the solution of the colored predecessor problem.

Consider a query for the predecessor of $x \in U$. If x is in the set S, we simply look up its color. Otherwise, consider the path from the leaf corresponding to x to the root in

the main trie $\mathbf{D}$. Let u be the lowest colored node on the path and v be the child of u on this path. We need to find the left colored sibling of v if one exists. This itself is a predecessor problem with small universe B. Unlike the solution mentioned in the previous section, scanning exhaustively the colors of the siblings is prohibitively expensive. Instead, the supporting data structure is used.

Once the lowest colored node u on the path from leaf x to the root in $\mathbf{D}$ is found, a colored predecessor query of v is performed in the supporting structure $\mathbf{D}'(u)$, where v is the child of u on the path in $\mathbf{D}$. If there is a left colored sibling of v in $\mathbf{D}$, the color returned by the query using the supporting structure $\mathbf{D}'(u)$ is the answer. Otherwise, the query in the supporting structure returns no color. In that case, we have to find w, the lowest ancestor of v in $\mathbf{D}$, which is not the smallest among the colored siblings. A colored predecessor query in the supporting data structure provides the answer. So, the query time is $O(H + h + b) = O(H + \frac{\log B}{\log \log B}) = O(\sqrt{\frac{\log U}{\log \log U}})$.

2.4.3 Insertion

To insert an element $x \in [U]$ with color c, consider the path from the leaf corresponding to x to the root in the main trie $\mathbf{D}$. Let u be the lowest colored internal node on the path and v be the child of u on the path. Color each node from x to v with c and mark each node except v as both the smallest and the largest among the siblings. In addition to this, insert an element in the empty supporting structures associated with each of these nodes. Check whether or not v becomes the new smallest or the largest colored node among the siblings. In that case, the markings for the largest or the smallest must be updated. Using $\mathbf{D}'(u)$, the largest (smallest) colored child of u in $\mathbf{D}$ can be determined. The color of the ancestors of v in $\mathbf{D}$ might be changed due to the insertion when v becomes the new largest node among the colored siblings. Let w be the lowest ancestor of v such that w is not the largest among the colored siblings. Set the colors of the nodes on the path from u, the parent of v, to w with the color of v. Each of these nodes, except w, is the largest among their colored siblings. This results in changing the colors of the largest elements of the corresponding supporting structures.

Inserting elements in $O(H)$ supporting structures requires $O((h+b)H)$ time. Updating the largest or the smallest markings at one level requires $O(hb)$ time. Finally, changing

the color of the ancestors takes $O(H + Hh) = O(Hh)$ time. In total, insertion requires $O(Hh + hb + Hb) = O(\frac{\log U}{\log \log U})$ bit probes.

2.4.4 Deletion

To delete an element $x \in S$, consider the path from leaf x to the root in the main trie $\mathbf{D}$. Note that all nodes on the path are colored. Let u be the lowest node on the path that has more than one colored child. Let v be the child of u on the path. All nodes on the path from x to v except v are both the smallest and the largest among their siblings. The nodes u and v can be determined easily by using the associated bit vectors that mark the smallest and the largest child of its parent node. Erase the colors of all nodes on the path from x to v. Also, erase their markings as the smallest and the largest among the siblings. For all internal node a on the path from x to u, delete corresponding element from the supporting structures $\mathbf{D}'(\mathbf{a})$.

If v is neither the smallest nor the largest child of u, then we are done. If v is the smallest child of u then, use the structure $\mathbf{D}'(\mathbf{u})$ to find the new smallest child and update the vector used to mark the smallest child of u. On the other hand, if v is the largest child of u, use $\mathbf{D}'(\mathbf{u})$ to find the new largest child of u and update the markings accordingly. Let w be the lowest ancestor of v which is not the largest among the siblings. Change the color of all nodes on the path from u to w. Also, change the color of the largest element in $\mathbf{D}'(\mathbf{a})$, for all nodes a on the path from u to w. So the time requires for deletion is $O(Hh + hb + Hb) = O(\frac{\log U}{\log \log U})$ bit probes.

2.5 Further Improving the Query Time

Next we modify our data structure to support faster query while keeping the same asymptotic update time. We generalize the two-level structure, described in the previous section, into a k level structure for some positive integer k. At the i-th level of recursion of the data structure, we have a collection of tries with branching factor b_i and height h such that each internal node of a trie is associated with a level-$(i+1)$ trie with b_i leaves. As the height of the level-$(i+1)$ trie is h, the branching factor is, $b_{i+1} = {b_i}^{1/h}$. Then, the

condition ${b_k}^{(h^k)} \geq U$ must hold. The internal nodes of all the tries are supported with bit vectors to mark the largest and the smallest colored child.

2.5.1 Colored Predecessor Query

For a given colored predecessor query of $x \in [U]$, we proceed as in the previous section. If x is in the set then return the color of it. Otherwise, start from the level-1 trie. While searching in a level-i trie, start from the respective search leaf. Consider the path from the search leaf to the root of the trie. First find the lowest colored node u on this path. Let v be the child of u on the path. Check whether or not there is any left colored sibling of v. If not, continue towards the root to find the lowest node which is not the smallest colored node among its siblings. Then perform a query in the corresponding level-$(i+1)$ trie. Searching in a level-k trie is similar to the procedure mentioned in section 2.3. The searching time in a level-i trie, for $k > i \geq 1$, is given by the following equation

$$t_q(i) = O(h) + t_{l?}(i+1) + t_q(i+1)$$

$$\begin{aligned} \text{where } t_q(i) &= \text{time for a colored predecessor search in a level-}i\text{ trie} \\ t_{l?}(i) &= \text{time to check the existence of a smaller colored leaf} \\ &\quad \text{of a given leaf in a level-}i\text{ trie} \end{aligned}$$

Existence of a smaller colored leaf can be determined by following the path from the given leaf to the root to find a the lowest colored node on the path. Recursive searching in the next level trie associated with a determines whether there is a colored sibling to the left of the path. If not, continue upward along the path until the current node is not the smallest among siblings. Such a node exists if and only if there exists a smaller colored leaf. Note that

$$\begin{aligned} t_q(k) &= O(h + b_k) \\ t_{l?}(i) &= h + t_{l?}(i+1) \\ t_{l?}(k) &= O(h + b_k). \end{aligned}$$

Total query time is $t_q = t_q(1)$. Solving the recurrence, we get $t_q = O(hk^2 + kb_k)$.

2.5.2 Insertion

To insert an element $x \in [U]$ with color c, start from the level-1 trie. In general, while inserting an element in a level-i trie, start from the respective leaf of the trie. Consider the path from this leaf to the root. Find the lowest colored internal node u on the path. Let v be the child of u on the path. Set the color of each node to c on the path from the leaf to v. Also, mark the nodes, except v, as both the smallest and the largest among the colored siblings. In addition to this, insert the relevant element in each of the corresponding level-$(i+1)$ tries associated with each internal node on the path from the leaf to v. Note that these level-$(i+1)$ tries were empty and we have to insert a single element in them. In the level-i trie, we may have to update the smallest and the largest colored child of u in case v becomes the new smallest or the largest colored child of u. Note that, the smallest (largest) colored child of a node can be determined recursively and is the corresponding leaf found by the recursive searching in the level-$(i+1)$ trie associated with the node. Update the markings of the smallest or the largest colored child of u if necessary. The color of the ancestors of v might be changed due to the insertion, if v becomes the new largest node among the colored siblings. From node v continue upward traversal towards the root. If the node ascended from is the largest among the colored siblings, set the color of the node to c. Update the color of the largest element recursively in the level-$(i+1)$ trie associated with the node. To do this, in the corresponding level-$(i+1)$ trie follow the path from the leaf, which is also the largest colored leaf, to the root and update the colors of the nodes along the way as well as recursively update the colors in the associated trie. The complexity of this operation is the same as insertion of an element in an empty trie. Continue ascending along the path in the level-i trie. The procedure is done when either we ascend from a child which is not the largest among the colored siblings or when we reach the root in the level-i trie. Time for an insertion in a level-i trie is given by the following equation.

$$t_u(i) \;=\; O(h) + ht_u^0(i+1) + t_u(i+1) + t_m(i+1)$$

$$\begin{aligned}
\text{where } t_u(i) &= \text{time to insert an element in a level-}i\text{ trie} \\
&\quad \text{that contains at least one element} \\
t_u^0(i) &= \text{time to insert the first element in a level-}i\text{ trie} \\
t_m(i) &= \text{time to find the maximum (minimum) element of a level-}(i)\text{ trie}
\end{aligned}$$

Note that

$$\begin{aligned}
t_u^0(i) &= h + ht_u^0(i+1) \\
t_m(i) &= ht_m(i+1) \\
t_u^0(k) &= O(h) \\
t_u(k) &= O(h + b_k) \\
t_m(k) &= O(hb_k)
\end{aligned}$$

The total update time is $t_u = t_u(1)$. Solving the recurrence, we have $t_u = O(h^k + b_k h^{k-1})$.

For a constant k, let $b_k = \sqrt[k]{\frac{\log U}{\log\log U}}$ and $h = \sqrt[k]{k\frac{\log U}{\log\log U}}$. Note that the condition ${b_k}^{(h^k)} \geq U$ is satisfied with our choice of h and b_k. So, the query time is $t_q = O(hk^2 + kb_k) = O(k^2\sqrt[k]{\frac{\log U}{\log\log U}})$, for some positive integer k, and the update time is $t_u = O(h^k) = O(\frac{k\log U}{\log\log U})$. In particular, we have a trade-off between the query and update time of $t_q = O(k^2\sqrt[k]{\frac{\log U}{\log\log U}})$ and $t_u = O(k\frac{\log U}{\log\log U})$. The query time decreases while the update time increases with the larger value of k .

2.5.3 Deletion

To delete an element, start deletion from the level-1 trie. While deleting an element in a level-i trie, consider the path from the corresponding leaf to the root. Note that all the nodes on the path are colored. Let u be the lowest internal node on the path that has more than one colored child. Let v be the child of u on the path. All nodes on the path from x to v except v are both the smallest and the largest among their siblings. The nodes u and v can be determined easily by using the associated bit vectors that mark the smallest and the largest child of its parent node. Erase the colors of all nodes on the path from x to v.

Also, erase their markings as the smallest and the largest among the siblings. In addition to that, also delete recursively in the corresponding level-$(i+1)$ trie associated with each internal node on the path from x to u.

If v is neither the smallest nor the largest colored child of u, then we are done. If v is the smallest colored child of u, recursively find the new smallest child of u using the level-$(i+1)$ trie associated with u. Update the vector used to mark the smallest child of u. On the other hand, if v is the largest child of u, find the new largest child of u recursively and update the markings accordingly. Let w be the lowest ancestor of v which is not the largest among the siblings. Change the color of all nodes on the path from u to w. Also, change the color of the largest element recursively in the level-$(i+1)$ trie associated with all nodes on the path from u to w. So deletion in the level-i trie uses $t_d(i)$ bit probes and is denoted by the following equation.

$$t_d(i) \;=\; O(h) + ht_d^1(i+1) + t_d(i+1) + t_m(i+1)$$

$$\begin{aligned} \text{where } t_d(i) &= \text{time to delete an element in a level-}i\text{ trie} \\ &\quad\; \text{that contains more than one element} \\ t_d^1(i) &= \text{time to delete the only element in a level-}i\text{ trie} \\ t_m(i) &= \text{time to find the maximum (minimum) element of a level-}i\text{ trie} \end{aligned}$$

Note that $t_d^1(i) = t_u^0(i)$ and hence $t_d(i) = t_u(i)$. So, total time for the deletion is $t_d = t_d(1) = O(\frac{\log U}{\log\log U})$ bit probes. Hence we have the following theorem.

Theorem 2.5.1. *The colored predecessor problem can be solved with the trade-off time complexity* $t_q = O(k^2\sqrt[k]{\frac{\log U}{\log\log U}})$ *and* $t_u = O(k\frac{\log U}{\log\log U})$, *where* t_q *and* t_u *are the number of bit probes for the query and the update respectively, for some positive integer* k.

2.6 Faster Queries with Slower Updates

Next we describe a scheme to improve the query time exponentially to achieve $t_q = O(\log\log U)$ at the expense of a slight increase in the update time $t_u = O(\log U)$. We

use the properties of *split tagged tree* [9] to achieve the time bounds. The set is represented by a complete binary trie with U leaves, where each element of S is encoded by coloring the corresponding leaf and all its ancestors. A node has the color of its largest colored descendant. A *splitting node* is a colored node whose both left and right children are colored. A splitting node u is a *right splitting node* of v if v is in the right subtree of u. The *left splitting node* is defined similarly.

Lemma 2.6.1. *For a given element x, let $p \in S$ be the predecessor of x. The lowest common ancestor of x and p is either (a) the lowest colored node on the path from the leaf corresponding to x to the root or (b) the lowest right splitting node of x.*

Proof. Let v be the lowest common ancestor of x and p. As the leaf corresponding to p and all its ancestors are colored, v is colored. We have x in its right subtree and p in its left subtree. The left child of v, which is an ancestor of the leaf p, is colored. Let u be the lowest colored node on the path from the leaf corresponding to x to the root. Either u and v are the same or v is an ancestor of u. The lemma follows when u and v are the same. Let v be an ancestor of u. Then u must be in the right subtree of v. As u is colored, the right child of v is also colored. So, v is a splitting node. Since, x is in the right subtree of v, it is the right splitting node of x. Now, v must be the lowest right splitting node of x. Otherwise, let w be the lowest right splitting node of x. Then there must be an element y in the left subtree of w such that $p < y < x$. This contradicts our assumption that p is the predecessor of x. □

The main result of this section is the following theorem. We present a data structure for the problem and show that the query and the update operations require the time bound mentioned in the theorem.

Theorem 2.6.2. *There exists a data structure for the colored predecessor problem with time complexities $t_q = O(\log \log U)$ and $t_u = O(\log U)$, where t_q and t_u are the query and update time in the bit probe model respectively.*

2.6.1 The Data Structure

The set S is represented by the complete binary tree mentioned earlier in this chapter. The complete binary tree is enhanced with jump nodes to find quickly the lowest right

splitting node of a given node. A *jump node* is an internal node at level $i\lceil \log\log U \rceil$ for any positive integer i. A jump node is associated with a jump pointer. A *jump pointer* at a colored jump node u, points to the lowest right splitting node of u. Note that a jump pointer points to an ancestor and that it stores the distance of the right splitting node in $O(\log\log U)$ bits. A jump pointer at a jump node without color contains no meaningful information.

2.6.2 Colored Predecessor Query

For a given query x, the aim is to find the lowest common ancestor of x and its predecessor. First find the lowest colored node on the path from the leaf corresponding to x to the root using binary search along the path. As the tree is complete, the location of the i-th ancestor of x is $\frac{x}{2^i}$. So, this requires $O(\log\log U)$ time. Check whether this node is the lowest common ancestor of x and its predecessor. If not, find the lowest right splitting node of x.

From the lowest colored node u on the path, climb up the tree at most $\lceil \log\log U \rceil$ levels. We reach either the lowest right splitting node or the lowest jump node. A node is a right splitting node if we ascend from right branch and its both children are colored. It requires a constant number of bits to be inspected. Otherwise, we reach a jump node and follow the jump pointer to get the lowest right splitting node. The color of the left child of this node is the answer. In total, the query needs $O(\log\log U)$ bit probes.

2.6.3 Insertion

To insert an element x with color c, consider the path from the leaf corresponding to x to the root. Let u be the lowest colored node on this path. Color each node from leaf x to the child of u, which is an ancestor of x, with color c. If x is in the right branch of u, we have to update the colors of u and its ancestors as done in the previous section. In addition to that, we have to update the jump pointers on the path from x to u. They now should point to u. On the other hand, if x is in the left branch of u, find v the lowest right splitting node of u. Update the jump pointers on the path from x to u. They all now point to v. Also, the jump pointers on the path from the leaf corresponding to the successor of x to u should

be updated to point to u. Moving up or down along the path requires $O(\log U)$ bit probes. There can be $O(\frac{\log U}{\log \log U})$ jump nodes on a path and each pointer contains $O(\log \log U)$ bits. Updating jump pointers requires $O(\log U)$ bit probes. The total insertion complexity is $O(\log U)$.

2.6.4 Deletion

To delete an element x, consider the path from the leaf corresponding to x to the root. All nodes on the path are colored. Let u be the lowest splitting node on this path. Let v be the child of u on this path. Erase the color of each node on the path from x to v. If x is in the right branch of u, then set the color of u to the color of its left child. On the other hand, if x is in the left branch, then follow the path from u to the leaf that corresponds to the successor of x. Along the path update all the jump pointers. All these jump pointers should point to the lowest right splitting node of u. Moving up or down along the path requires $O(\log U)$ time. There can be $O(\frac{\log U}{\log \log U})$ jump nodes on a path and each pointer contains $O(\log \log U)$ bits. Updating jump pointers requires $O(\log U)$ bit probes. So total complexity for deletion is $O(\log U)$ bit probes.

Chapter 3

Application of the Colored Predecessor Problem

The colored predecessor problem has numerous applications, especially in searching and graph problems. In this chapter, we present some applications by showing that the solution for the colored predecessor problem can be used to solve some other related problems efficiently. The problems considered are the segment representative problem [48], existential range query problem [2, 40, 44], dynamic prefix sum problem [25, 38, 48] and dynamic connectivity problem [6, 19, 30, 41, 54].

3.1 Segment Representative Problem

The segment representative problem is a variation of the predecessor problem. A segment is formed by two consecutive elements of S. The elements of S divide the universe into $n + 1$ segments. The representative of a segment can be considered as a point inside the segment.

> *Segment Representative Problem*: The segment representative problem is to store a dynamic set S of n elements from a universe U in order to support queries to determine a representative of the segment containing the query element, where the representative depends on the leading matching bits of the boundary

elements of that segment and is independent of the query point.

The representative is not written down explicitly as it requires $\log U$ bits to write down the segment representative completely. Rather, the query determines the matching leading bits of the two boundary elements of the segment and indicates a point inside the segment that does not depend on the query point. So, we provide more information than simply returning the color of the predecessor of a query point. Insertion or Deletion of elements splits or merges segments respectively. The representatives of the affected segments can be changed without affecting the representatives of other segments.

We first define a canonical representative for each segment. The set is represented by a complete tree with branching factor b and U leaves. A leaf corresponds to an element denoted by the labels on the path from the root to this leaf. The elements of S are encoded by coloring the corresponding leaf and all its ancestors as in the case of the colored predecessor problem. Let $a, b \in S$ be two consecutive elements. Then they form a segment. Let u be the lowest common ancestor of a and b. Let v and w be two child of u such that v and w are the ancestors of a and b respectively. The segment representative is formed by the bits labelled on the path from root to u, followed by $\log b$ bits that distinguishes between v and w, and finally padded with trailing '0's. Once we know u, v and w we can construct the canonical representative in no time as the computations are free in the bit probe model.

The data structure for the colored predecessor problem can be used to solve the segment representative problem. The query and update becomes a little bit more complicated than those of the colored predecessor problem since the output depends not only on the predecessor but also on the successor of the query element.

The query proceeds in a similar way as in the case of the colored predecessor problem. Find u the lowest colored node on the path from leaf x to the root. Let v be the child of u on the path. Check whether or not v has a left colored sibling as well as a right colored sibling. If there is no left (right) colored sibling, find the lowest ancestor of v which is not the smallest (largest) among the colored siblings. The parent of this node is the lowest common ancestor of the predecessor and the successor. Update is same as in the colored predecessor problem. Hence, we have the following results.

Corollary 3.1.1. *The segment representative problem can be solved with the trade-of time*

complexity of $O(k^2 \sqrt[k]{\frac{\log U}{\log \log U}})$ *bit probes for query and* $O(k\frac{\log U}{\log \log U})$ *bit probes for insertion or deletion of an element, for some positive integer* k.

3.2 Existential Range Queries

The range query problem is to store a set S of n elements from a finite universe U such that the query $report(a, b)$ returns all elements $\{x \in S | a \leq x \leq b\}$. One variation of the problem is *range counting problem* that returns the number of elements within the range (a, b) for the given query $count(a, b)$. All these problems have inherent $\log U$ lower bound under the bit probe model. To obtain a sub-logarithmic result, we considered another variation of the range query problem, called existential range query problem.

> *Existential Range Query Problem*: The existential range query problem is to maintain a dynamic set S of n elements from universe U so that to determine the presence of an element within a given range.

Husfeldt, Rauhe and others [3, 32, 33] studied the problem along with other problems using refinement and extension of the chronogram method. They have shown a lower bound of $\Omega(\frac{\log n}{\log \log n})$ for the dynamic existential range query problem in the cell probe model. Miltersen et al [40] presented a data structure for the static version with space $O(n \log U)$ words with constant query time. Later Alstrup et al [2] presented an optimal solution for the static case with linear space and constant query time in RAM model. For the dynamic case, Van Emde Boas [18] structure requires $O(\log \log U)$ time per operation in RAM model. Mortensen et al [44] presented a linear space structure that requires $O(\log \log \log U)$ for query and $O(\log \log U)$ for updates in RAM model using bloomier filters.

We first describe the general technique used to solve the problem. The set is represented by a complete binary tree with U leaves as mentioned in earlier sections. Given a query $exists(a, b)$, asking whether or not there exists an element within the range (a, b), first determine u, the lowest common ancestor of a and b. The query range can be divided into two sub-ranges $(a, t]$ and (t, b), where the ranges $(a, t]$ and (t, b) are under the left and right subtree of u respectively. Start from the leaf corresponding to b to find whether or not

the predecessor of b lies within the range (t, b). Let v be the lowest common ancestor of b and its predecessor. According to the lemma 2.6.1, v is either the lowest active node on the leaf-to-root path of b or the lowest right splitting node of b. Note that both u and v are ancestors of b. Based on the relation of u and v, we can answer whether or not the predecessor of b lies within (t, b). If v is a descendant of u then the answer is 'yes' and we are done. Otherwise, start from the leaf corresponding to a and determine whether or not the successor of a lies within the range $(a, t]$. It can be answered using similar technique. We can generalize the notion for a trie and get the following lemma.

Lemma 3.2.1. *The lowest common ancestor of x and its predecessor is either the lowest colored node on the leaf to root path from x or the lowest colored node on the path such that it is not the smallest among the colored siblings. The lowest common ancestor of x and its successor is either the lowest colored node on the leaf to root path from x or the lowest colored node on the path such that it is not the largest among the colored siblings.*

Let u be the lowest common ancestor of a and b. Let v and w be two child of u such that v and w are the ancestors of a and b respectively. The range (a, b) is divided into subranges $(a, t_1], (t_1, t_2)$, and $[t_2, b)$, where $(a, t_1]$ and $[t_2, b)$ are the ranges under the subtree rooted at v and w respectively. Using lemma 3.2.1, find the lowest common ancestor of b and its predecessor and determine whether or not the predecessor lies within the range $[t_2, b)$. If not, determine whether or not the successor of a lies within the range $(a, t_1]$. If not, find recursively existence of an element within range (t_1, t_2) using next level structure associated with u. At each level of the structure, we have to traverse at most two leaf to root paths and one recursive range queries. As a result we have the following corollary.

Corollary 3.2.2. *The existential range query problem can be solved with trade-off time complexity of $O(k^2 \sqrt[k]{\frac{\log U}{\log\log U}})$ bit probes for the queries and $O(k\frac{\log U}{\log\log U})$ bit probes for the updates, where k is a positive integer.*

3.3 Dynamic Prefix Sum Problem

Let M be a fixed finite monoid with an associative operator $\bigoplus$ and the identity operator.

Dynamic Prefix Sum Problem: The dynamic prefix problem associated with a monoid M is to maintain $x \in M^n$ so that to support queries $prefix(i)$ asking to return $\bigoplus_{j=1}^{i} x_j$ and $change(i, a)$ changing x_i to $a \in M$.

As M is fixed there is an easy upper bound of $O(\log n)$ in the bit probe model and $O(\frac{\log n}{\log \log n})$ probes in the cell probe model with cell size $O(\log n)$ [38].

Fredman [25] studied the dynamic problem with monoid integer *mod* 2. He showed a $\Omega(\frac{\log n}{\log \log n})$ bit probe lower bound using decision assignment tree technique. Fredman and Saks [27] have shown that the same number of probes is required in the cell probe model with cell size $O(\log n)$ using chronogram method. Pătraşcu and Pătraşcu [48] recently proved a lower bound of $\Omega(\frac{\log n}{\log \log \log n})$ for the problem with monoid integer *mod* 2. A classification of the complexity of the dynamic word and prefix problem based on the algebraic properties of M was done by Frandsen et al [23]. They have presented several upper and lower bound structures. Based on their work Pătraşcu and Pătraşcu [48] derived an upper bound of $O(\frac{\log n}{\log \log n})$ for a group-free monoid in the bit probe model. The tools used in their solution is the Krohn-Rhodes decomposition [34] and a data structure for for the colored predecessor problem. A group-free monoid is decomposed under Krohn-Rhodes decomposition and the prefix sum in each of these sub categories are solved using the predecessor structure. The complexity of the solution depends on the efficiency of the colored predecessor data structures. As a result, replacing the colored predecessor structure in their solution by the structures developed in the previous sections, give us the following corollary.

Corollary 3.3.1. *The dynamic prefix sum problem can be solved in* $O(k^2 \sqrt[k]{\frac{\log U}{\log \log U}})$ *bit probes for the queries and in* $O(\frac{k \log U}{\log \log U})$ *bit probes for the updates, where* k *is a positive constant.*

3.4 Dynamic Connectivity Problem

Dynamic Connectivity Problem: The dynamic graph connectivity problem is to maintain a directed or undirected graph with set of vertices V of size n under operations $insert(u, v)$ - inserting an edge between u and v in the

graph, $delete(u, v)$ - deleting an edge between u and v and query operation $connected(u, v)$ that answers whether or not u and v are connected.

Henzinger and King presented the best worst case solution is $O(n^{1/3})$ time per operation for the general graph [30]. Holm et al. [31] presented an amortized $O((\log n)^2)$ time per operation solution. Thorup [54] later improved it to $O(\log n \log\log n)$ amortized time. Fredman and Henzinger [26] and Miltersen et al. [41] proved a lower bound of $\Omega(\frac{\log n}{\log\log n})$ using a reduction from a dynamic prefix problem. Eppstein [19] achieved a worst case time complexity of $O(\log n)$ for the dynamic connectivity problem in plane graph with fixed embedding. The lower bound of $\Omega(\frac{\log n}{\log\log n})$ holds even for a grid graph where V forms a grid and all edges must be grid edges. Mix Barrington et al. [6] presented upper bound solution of $O(\log\log n)$ time per operation on a word RAM with word size $O(\log n)$. They also proved a lower bound $\Omega(\frac{\log\log n}{\log\log\log n})$ in the cell probe model with cell size $O(\log n)$. Husfeldt and Rauhe [32] show that the query time grows linearly with the graph width.

Other graph problems of interest are *directed dynamic graph reachability problem*, *dynamic planarity testing*, etc. Tamassia and Preparata [53] have shown that the dynamic reachability can be solved in time $O(\log n)$. Husfeldt et al. [32, 33] show a lower bound of $\Omega(\frac{\log n}{\log\log n})$ for the reachability problem using chronogram method. Husfeldt and Rauhe [32] have shown a lower bound of $\Omega(\frac{\log n}{\log\log n})$ for the planarity testing problem.

We consider a restricted version of the dynamic graph connectivity problem, where V forms a constant width grid and the edges must be grid edges. Mix Barrington et al. [6] have shown that the problem can be reduced to dynamic prefix problem and achieved upper bound $O(\log\log n)$ and lower bound $\Omega(\frac{\log\log n}{\log\log\log n})$ in the cell probe model.

We consider the dynamic connectivity problem in a grid graph with length n and a constant width c in the bit probe model. As mentioned earlier, the problem can be reduced to dynamic prefix sum problem. Using the data structure for the dynamic prefix sum problem, we can present a data structure for the dynamic connectivity problem on a constant width grid graph.

Corollary 3.4.1. *The dynamic connectivity problem on a constant width grid graph can be solved in* $O(k^2 \sqrt[k]{\frac{\log U}{\log\log U}})$ *bit probes for the queries and in* $O(\frac{k \log U}{\log\log U})$ *bit probes for the updates, where* k *is a positive constant.*

Chapter 4

Integers and Their Representations

The data type *integer* is fundamental to any computer and any programming language. Therefore, the representation of integers and operations on them are fundamental issues. We study the problem of integer representation using a nearly minimal number of bits so that basic operations can be done efficiently. The operations include *increment*, *decrement*, *addition* and *subtraction*. In this chapter, we explore some alternatives for integer representations and their efficiencies based on the number of bit probes required to perform the basic operations on integers.

4.1 Preliminaries

We are interested in representing the integers in the range $[0, 2^n - 1]$, where n is the minimum number of bits required to represent an element of this set, and we call n the dimension of an integer in this range. Let x be an integer of dimension n. The increment operation yields a representation of the value $(x + 1) \bmod 2^n$. Similarly, the decrement operation can be defined. The addition operation considered is of the form $x \leftarrow x + y$, where x has a larger dimension than y. The number x is replaced with the sum. The subtraction operation can be defined similarly.

The trivial way to represent integers is to use the standard binary number system that uses n bits to represent an integer of dimension n. It requires n worst case bit probes per operation. Frank Gray invented the Binary Reflected Gray Code (BRGC) while converting

analog signals into digital signals [29]. A Gray code sequence of dimension n contains a sequence of 2^n elements, where each element of the sequence is a n-bit code, such that two consecutive codes in the sequence differ only in one bit. In cyclic gray code, the first and the last element in the sequence also differ in one bit. The BRGC is a cyclic gray code. The BRGC uses n bits and the algorithm to generate the next code of the BRGC sequence requires n bits inspections. But only one bit needs to be changed to get the next code.

The redundant binary system supports constant amortized time per operation [11, 12]. The scheme doubles the space usage. But the model considered in these schemes is not the bit probe model. Frandsen et al. [23] gave a representation that requires $O(\log n)$ bit probes for increment and decrement. Using the same time, the number can be tested to be equal to a number chosen from a fixed set. Fredman used the decision assignment tree to obtain the bit probe lower bound of $\Omega(\log n)$ to generate a quasi-gray code sequence [24]. He used a tree representation for redundant binary numbers that uses three times the minimum space required while permitting constant bit changes for the increment operation.

4.2 Standard Binary Representation

The standard binary representation of an integer of dimension n requires n bits with no additional bits. Let X be the n-bit integer number that contains a value $x \in [0, 2^n - 1]$ and is denoted by $X = x_n x_{n-1} \cdots x_1$, where $x_i \in [0, 1]$. Each bit of the representation is associated with a weight. The i-th bit has a weight of 2^{i-1}. So, the standard representation satisfies the equation $x = \sum_{i=1}^{n} 2^{i-1} x_i$.

To increment, start scanning the bits from the right until the rightmost '0'. Flip this bit to '1' and flip all previous bits to '0's. The decrement operation is similar. So, the standard binary representation requires n worst case bit probes per operation. An example is shown in figure 4.1.

The addition of two integers x and y of dimensions n and m, respectively, proceeds from the rightmost bits of x and y. Add the corresponding bits with the proper propagation of carry. Clearly, the operation requires $n + m$ bit inspections and $n + 1$ bit changes in the worst case, where $n > m$. We omit the details here as the details can be found in any elementary book on number systems.

(a)

X = 0 0 1 1 0 1 0 0	(52)
Y = 0 1 0 1 0 1	(21)
S = 0 1 0 0 1 0 0 1	(73)
C = 0 0 1 1 0 1 0 0	

(b)

Figure 4.1: Increment and addition using standard binary representation.

4.3 Binary Reflected Gray Code

In 1953, Gray patented his Gray code [29]. Since then, many authors confirmed the importance of the Gray code and its variants. For detailed survey on the Gray code, the readers are referred to the paper by Doran [15]. In this section, we briefly review the Gray code and its properties and the increment/decrement and the addition/subtraction operations on them.

> *Binary Reflected Gray Code*: The empty string is the sequence $G(0)$, the Binary Reflected Gray Code (BRGC) sequence of dimension 0. Let $G(n-1)$ be the BRGC sequence of dimension $n-1$. Then the following sequence: $G(n) = 0.[G(n-1)], 1.[G(n-1)]^R$ is the BRGC sequence of dimension n, where $[G(n-1)]^R$ is the BRGC sequence of dimension $n-1$ in the reverse order and 0 or 1 in the front denotes the concatenation of a bit '0' or '1' in front of each code of the sequence.

A BRGC sequence of dimension 5 is shown in Figure 4.2. The generation of BRGC has been studied extensively [14, 22, 36]. To get the next code, for an even parity code, flip the rightmost bit. For an odd parity code, find the rightmost '1' and flip the bit to

00000	01100	11000	10100
00001	01101	11001	10101
00011	01111	11011	10111
00010	01110	11010	10110
00110	01010	11110	10010
00111	01011	11111	10011
00101	01001	11101	10001
00100	01000	11100	10000

Figure 4.2: The Binary Reflected Gray Code (BRGC) sequence of dimension 5.

its left. The only special case is when the last bit x_n is the only '1' in the code. This is also the last code in the BRGC sequence. Note that the parity is odd. But in this case, it is sufficient to flip bit x_n to go back to the first code of the BRGC. Boothroyd's variant [8] calculates the parity and finds the last '1' bit by a scan of the bits. With no additional space, the algorithm to generate the next code of the BRGC sequence requires n bits inspections, as it has to determine the parity of the code. But only one bit needs to be changed to get the next code. One extra bit, to store the parity of the code [43], improves the performance significantly on average. Misra [43] keeps the parity separately and maintains a stack of indices of bits that are '1's to get a very fast algorithm. Er [20] presented some improvements on Misra's [43] algorithm. None of the algorithms mentioned above considers the problem in the bit probe model and these algorithms need to change a logarithmic number of bits in the worst case.

Lucal proposed a modified Gray code where the parity bit is integrated into the code [35]. It still requires n bit inspections in the worst case. To get the next code in the sequence, we have to change two bits now — one bit of the code and the parity bit which is also a part of the code.

4.4 The Tree Representation

Fredman showed that at least $\log n$ bits need to be inspected to determine which bit to flip to get the next element in a quasi-Gray code sequence [24]. He presented a data structure

that supports the increment operation using $2 \log n$ bit inspections and 7 bit changes. The data structure uses $5n - 2$ bits in total, where n bits are used to store the Gray code explicitly. But the structure does not efficiently support the decrement operation.

We extend and modify Fredman's data structure in order to support both increment and decrement operations. We first describe a representation that supports increment and decrement using a logarithmic number of bit inspections and changes. Then we modify the structure to support the operations using a constant number of bit changes.

Without loss of generality, we can assume that $n = 2^k$ for some positive integer k. If not, we can always add leading 0's to make it a power of 2. An integer is represented by a complete binary tree with n leaves such that the bits of the standard binary representation of this number are associated with the leaves of the tree. The key trick is to associate a '0' or a '1' with an internal node to indicate that its entire subtree implicitly has that value. More formally, the leaves and the internal nodes are labelled as follows:

- A node has label '1' if this is the highest node such that all bits corresponding to the leaves in its subtree are '1's.
- A node has label '0' if this is the highest node such that all bits corresponding to the leaves in its subtree are '0's.
- All other nodes are labelled '?' denoting a mix of '0's and '1's in the bits corresponding to the leaves in its subtree, or that its value is given implicitly by an ancestor.

The Gray code is not stored explicitly. As a result the data structure uses $4n - 2$ bits to store an integer of dimension n. See Figure 4.3 for an example. Note that, if the label of a node is '1' then, the label of its sibling can not be '1'. Similarly, if a node has label '0' then, its sibling can not have a label '0'. Moreover, if a node is labelled '1' or '0' then all of its ancestors are labelled '?'. By inspecting the label of the root, one can test if the number is 0, $2^n - 1$ or neither.

4.4.1 Increment Operation in a Logarithmic number of Bit Probes

Consider the increment in standard binary representation. The increment operation there finds the rightmost '0'. It flips this rightmost '0' into '1' and the trailing '1's into '0's. Our

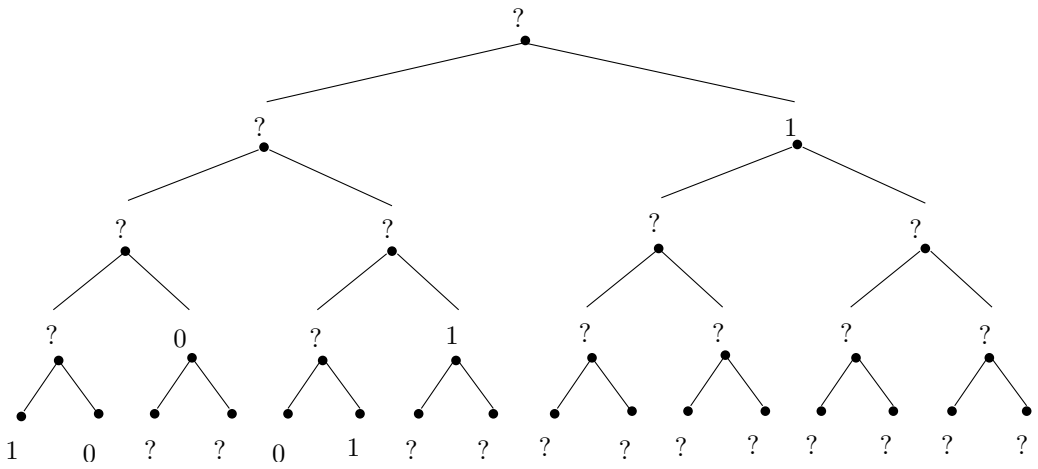

Figure 4.3: The tree representation of an integer with standard binary representation 1000011111111111.

increment algorithm proceeds along the same line.

1. Start from the root and follow the rightmost path down the tree until finding a node with a non-'?' label. If the label is '1' then, change it into '0' and move to its left sibling. Follow the rightmost path down the tree possible from that node. Continue the process until a node with label '0' is reached.

2. There can be two cases based on the position of this node with label '0'.

   (a) If the node is an internal node then, the bits in its subtree are all '0's. Only the rightmost '0' needs to be changed into '1'. Change the label of this node to '?'. Follow the rightmost path down the subtree from this node. Change the labels of all siblings of the nodes on the path to '0's. Finally change the label of the leaf to '1'.

   (b) Otherwise, the node is a leaf and corresponds to a bit that needs to be flipped from '0' to '1'. If this leaf is a left child of its parent, simply change the label to '1'. Otherwise, check whether or not the corresponding bit is going to be an isolated '1'. Traverse upward the tree, check whether the label of the left sibling of the current node is '1' and change its label from '1' to '?'. Continue until

either we reach a node which is the left child of its parent or its left sibling has a non-'1' label. Change the label of this node to '1'.

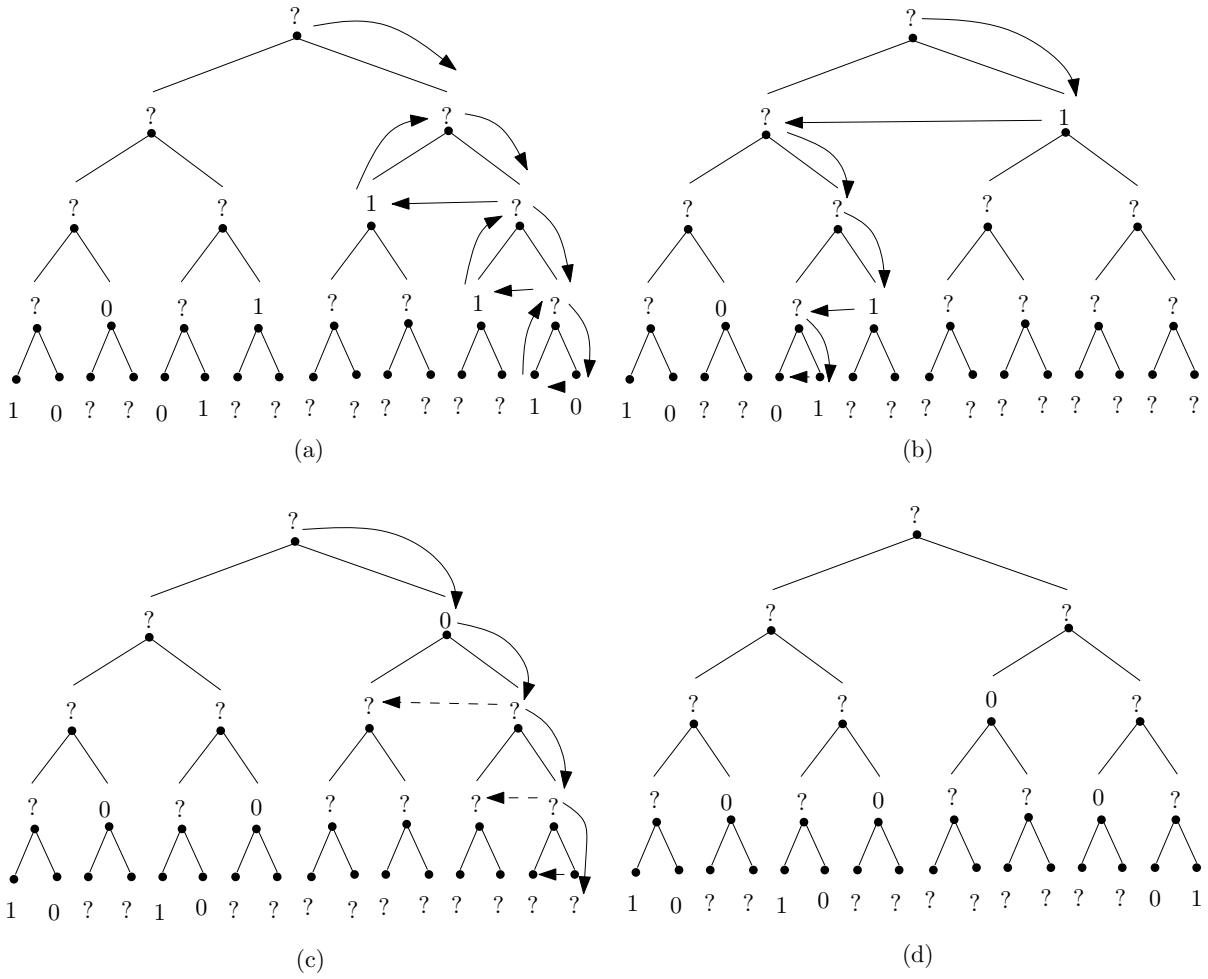

Figure 4.4: Increment operations using logarithmic number of bit changes.

Figure 4.4 shows an example of the algorithm mentioned above. Arrow signs show the steps in increment operation. Figure 4.4(a) shows the tree representation of an integer with standard binary representation 1000011111111110. An increment leads to Figure 4.4(b), the tree representation of 1000011111111111. Subsequent increments leads to Figure 4.4(c)

and Figure 4.4(d), the tree representations of 1000100000000000 and 1000100000000001 respectively. The decrement operation is similar.

4.4.2 Increment Using a Constant Number of Bit Changes

Next we modify the tree representation so that increment and decrement operations can be performed using $O(\log n)$ bit inspections with a constant number of bit changes. First we analyze the steps of the algorithm mentioned in the previous subsection. There are three sequences of label changes in steps 1, 2(a), and 2(b) of the increment algorithm in the preceding subsection. Steps 2(a) and 2(b) are mutually exclusive as only one takes place during an increment operation. Instead of changing the labels of all nodes in a sequence, a new label symbol 'σ' is used to denote that the changes in the subsequent nodes in a sequence are pending. We call this type of update a *lazy update*. Since the representation uses only four symbols, the labels uses two bits per node. As a result the space usage remains the same. The following paragraphs describe the changes needed in the increment algorithm to accommodate this new symbol so that increment (and decrement) operation ca! n be performed using a constant number of bit changes.

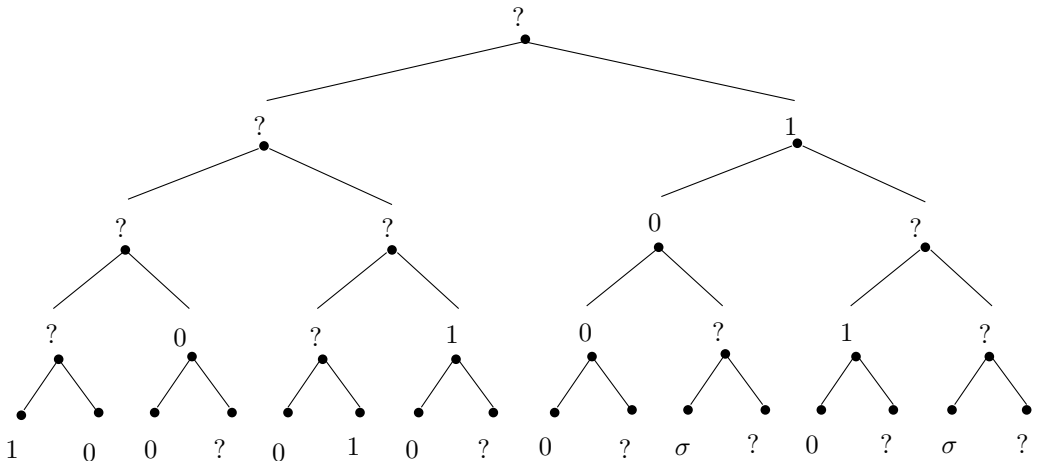

Figure 4.5: A valid labelling of the tree representation of 1000011111111111.

Let $n_1, n_2, \cdots, n_k$ be the nodes considered in step 1 (of the increment algorithm in the

preceding subsection) that get new label '0'. Note that each of these nodes is the right child of its parent and previously labelled '1'. For a node n_i, the next node in the sequence n_{i+1} is in the subtree rooted at the left sibling of n_i and can be reached following the rightmost path possible down the tree from that left sibling. Instead of changing the labels of all nodes in the sequence, the node n_1 is labelled 'σ' and the node n_k is labelled '0'. The label 'σ' implies that this node and following nodes in the sequence should be labelled '0'. A subsequent decrement operation restores the labelling. On the other hand, when an increment operation encounters the label 'σ', it propagates 'σ' to the next node in the sequence.

Next consider the series of label changing performed in step 2(a). Let $n_1, n_2, \cdots, n_k$ be the sequence of nodes in step 2(a) that get the new label '0'. Each of these nodes is the left child of its parent. For a node n_i, the next node n_{i+1} in the sequence is the left child of the right sibling of n_i. Instead of changing the labels of all nodes in the sequence, the node n_k gets the label 'σ' to indicate pending changes in the sequence and the node n_1 gets the label '0' to mark the end of the sequence. Note that, the 'σ' in this sequence denotes that the corresponding node should be labelled '0' and it moves to the previous node in the sequence during a later increment operation.

Finally, let $n_1, n_2, \cdots, n_k$ be the sequence of nodes in step 2(b) that get the new label '0'. Each of these nodes is the left child of its parent. For a node n_i, the next node n_{i+1} is the left sibling of its parent. Like the earlier two cases, the node n_1 gets label 'σ' and the node n_k gets the label '0'. This label 'σ' moves upward to the next node in the sequence during a later increment operation. In all these sequences, the label 'σ' denotes the start of a lazy update. The end of a sequence is denoted by a node with label '0'. A decrement operation undo the changes done by increment operation.

The lazy version of label updating considered in steps 1 and 2(a) are consistent with the older version. But for step 2(b), the changes are slightly different. In the older version, the nodes in the sequence got a new label '?'. But the modified algorithm suggests that the nodes should be labelled '0'. Using lazy update, only the last node in the sequence is labelled '0' and the first node is labelled 'σ' to denote the pending changes. We now show that the lazy update scheme leads to a consistent labelling for step 2(b) also.

Let $n_1, n_2, \cdots, n_k$ be the sequence of nodes considered in step 2(b) and u be the parent

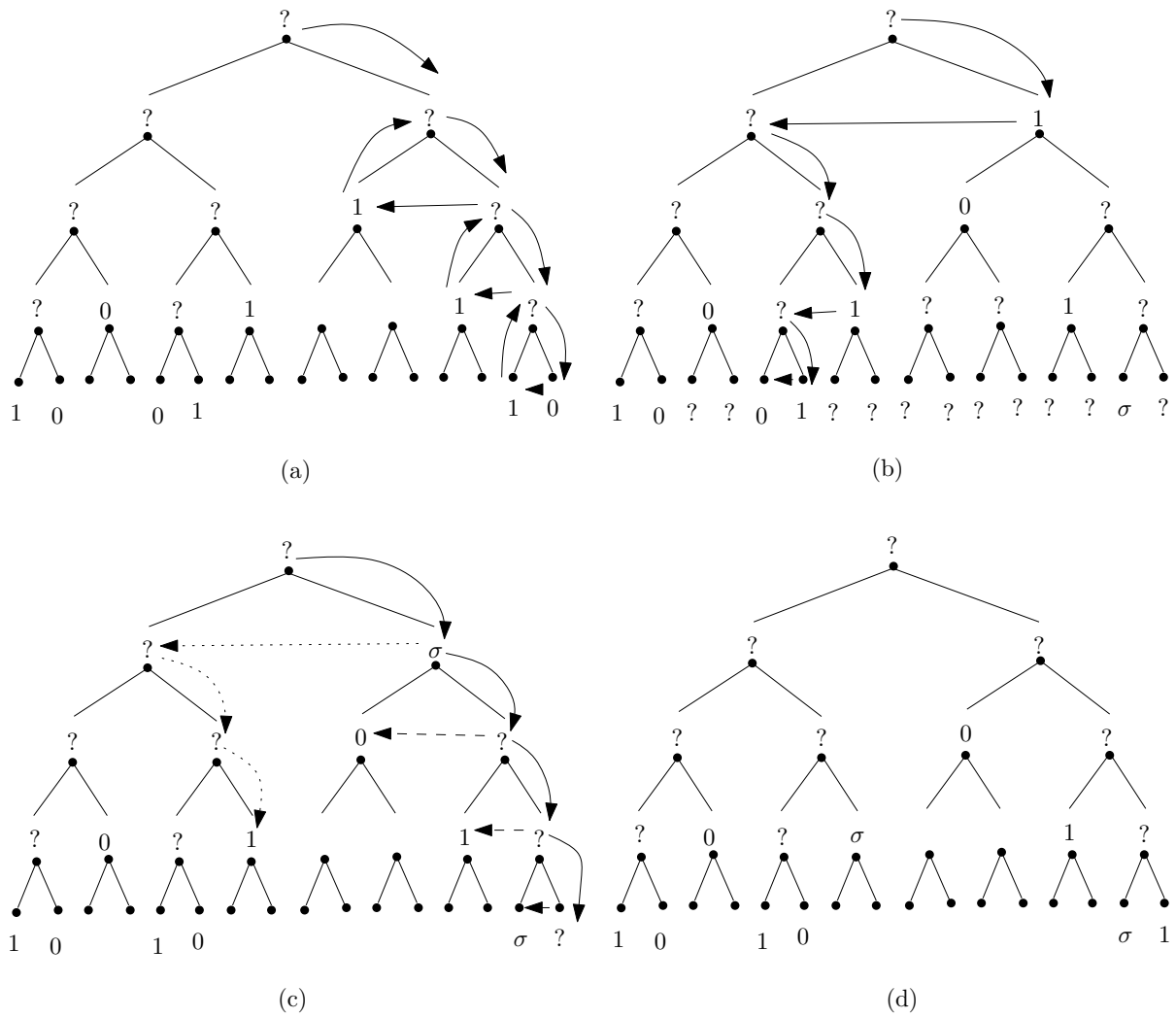

Figure 4.6: Increment operations using a constant number of bit changes.

of n_k. After lazy update, we have $label(n_1) =$ 'σ', $label(n_k) =$ '0' and $label(u) =$ '1'. A decrement restores the labelling. So, consider the next increment that will affect these nodes. The bits in the binary representation corresponding to the leaves in the subtree rooted at u are all '1's now. After that later increment, all these bits should be '0's. So, only the label of u will be changed into '0' or 'σ'. All other labels remains unchanged. Another later increment will find that u satisfies the conditions in step 2(a) and labels will be changed as: $label(u) =$ '?', $label(n_1) =$ 'σ' and $label(n_k) =$ '0'. Note that the nodes in concern already have these labels, and these new assignments overwrite the same labels on these nodes. Thus, the lazy update leads to consistent labelling in step 2(b).

The label 'σ' moves up or down while incrementing. The upward or downward movement of 'σ' depends on the type of sequence to which it belongs. The 'σ' in a type 1 sequence (sequence in step 1 of the increment algorithm in the preceding subsection) moves downward to the next node in the sequence. On the other hand, label 'σ' in type 2(a) or 2(b) sequence moves upward. It can be easily identified whether a 'σ' should move up or down: if the corresponding node is the right child of its parent, the 'σ' is part of a type 1 sequence and should move downward. The 'σ' moves up when the corresponding node is the left child of its parent and hence is part of a sequence of type 2(a) or 2(b). The decrement algorithm mainly undo the changes done during a previous increment operation that lead to the current value. Figure 4.6 shows an example of the increment operation where arrow signs show the steps of the operation. The labels of the relevant nodes are shown in the figure. The tree representation of an integer with standard binary representation 1000011111111110 is shown in Figure 4.6(a). An increment leads to Figure 4.6(b), the tree representation of 1000011111111111. Subsequent increments lead to Figure 4.6(c) and Figure 4.6(d), the tree representations of 1000100000000000 and 1000100000000001 respectively.

Chapter 5

Integers and Counting

We first present a new lower bound result for the increment operation of an integer with dimension n that is represented with only n bits. Then we present several upper bound solutions to the problem that use more space. The first solution requires $n + \log n + 3$ bits of space. Using this representation, the increment and decrement operations require at most $2 \log n + 4$ bit inspections and at most 4 bit changes per operation.

Our second structure uses $n + \log n + O(\log \log n)$ bits. The increment and decrement operations require $\log n + O(\log \log n)$ bit inspections and a constant number of bit changes. But these two structures do not support efficient addition or subtraction operations.

In the third solution, the data structure uses $n + O((\log n)^2)$ bits to store an integer number of dimension n and supports increment/decrement operations as well as addition/subtraction operations. The increment and decrement operations require $O(\log n)$ bit inspections and at most 5 bit changes. On the other hand, the addition or subtraction between two integers of dimensions n and m with $n > m$ requires $O(m + \log n)$ bit inspections and $O(m)$ bit changes.

5.1 The Lower Bound

In this section, we present a new lower bound for the increment operation of an integer with dimension n. The Sunflower lemma is used to prove the lower bound. A *sunflower* with p petals is a collection $S_1, S_2, \cdots, S_p$ of sets so that the intersection $S_i \cup S_j$ is the

same for each pair of distinct indices i and j. The intersection is called the center of the sunflower. The following well-known lemma is due to Erdös and Rado [21].

Lemma 5.1.1. *(Sunflower Lemma) Let $S_1, S_2, \cdots, S_m$ be a system of sets each of cardinality at most l. If $m > (p-1)^{l+1}l!$, then the collection contains as a subcollection a sunflower with p petals.*

Let M be a memory of size n bits. There can be $m = 2^n$ distinct memory configurations. So, the memory M can contain an integer of dimension n. A counting sequence $s = c_0c_1 \cdots c_{m-1}c_0$ is a sequence of distinct memory configurations such that for any two memory configurations c_i and c_j, if $i \neq j$ then $c_i \neq c_j$. From the initial configuration c_0, the sequence can be generated by a sequence of increment operations. The increment algorithm is denoted by a decision assignment tree where at each internal node a single bit is inspected and at each leaf one or more bits are flipped, so that the algorithm is ignorant of the current memory configuration.

Theorem 5.1.2. *Consider an integer of dimension n represented in exactly n bits using any representation of the values in the range. The increment operation on this representation requires $\Omega(\sqrt{n})$ bit inspections in the worst case.*

Proof. Fix a counting sequence $s = c_0c_1 \cdots c_{m-1}c_0$. Let S_i be the set of bits inspected while switching from configuration c_i to configuration $c_{(i+1) \bmod m}$, for all i such that $0 \leq i < m$. Build a decision assignment tree for the sequence s. For a memory configuration c_i, follow the root-to-leaf path and flip the bits mentioned at the leaf to get the memory configuration c_{i+1}. Let l be the height of the longest path of the decision assignment tree. Therefore, $|S_i| \leq l$ for all i such that $0 \leq i \leq m-1$.

Let p' be the largest integer satisfying the inequality in the sunflower lemma. Therefore, we can find a sunflower $S_{i_1} \cdots S_{i_{p'}}$ with p' petals for all j, such that $0 \leq i_j < m$ and $1 \leq j \leq p'$. Let C be the center of the sunflower. Thus C is a set of bits.

The inequality in the sunflower lemma is not satisfied for the value $p'+1$, as p' is the largest integer that satisfies the inequality.

$$\begin{aligned} \text{So, we have, } m &\leq \{(p'+1)-1\}^{l+1} l! \\ &= (p')^{l+1} l! \\ \Rightarrow \log p' &\geq \frac{\log m}{l+1} - O(\log l) \end{aligned}$$

Consider two sets S_i and S_j such that they correspond to two petals of the sunflower. The sets S_i and S_j corresponds to two root-to-leaf paths in the decision assignment tree. Consider the bit inspected at the lowest common node of these two paths. The bit is in C as it is a common bit of S_i and S_j. Furthermore, the value of the bit must differ for S_i and S_j. Therefore, the content of C must uniquely identify a petal of the sunflower. Thus, $|C| \geq \log p'$.

$$\begin{aligned} \text{Therefore, } |C| &\geq \log p' \\ &\geq \frac{\log m}{l+1} - O(\log l) \end{aligned}$$

But, we know $|C| \leq l$.

$$\begin{aligned} \text{Therefore, } l &\geq \frac{\log m}{l+1} - O(\log l) \\ \Rightarrow l &= \Omega(\sqrt{\log m}) \end{aligned}$$

Since, $m = 2^n$, we have $l = \Omega(\sqrt{n})$. □

5.2 Properties of Binary Reflected Gray Code Sequence

We examine some properties of the BRGC. These properties are used to design efficient data structure for integer representation. Let X be the BRGC of dimension n that contains a value $x \in [0, 2^n - 1]$ and is denoted by $X = x_n x_{n-1} \cdots x_1$, where $x_i \in [0, 1]$. The definition of the BRGC sequence leads to the following observations.

Observation: In the BRGC sequence of dimension n, the i-th bit is flipped 2^{n-i} times, for all i such that $1 \leq i < n$. The n-th bit is flipped 2 times.

Observation: In the BRGC sequence of dimension n, a transition from a code in the sequence to the next code causes one bit flip. Consider the flip of bit x_i.

- In each of the next $2^{i-1} - 1$ transitions, a bit x_j is flipped, where $i > j$.
- A bit x_k is flipped in the 2^{i-1}-th transition, where $i < k$.
- The distance (number of transitions) between two flips of x_i is 2^i.

The observations can be verified easily from the Figure 4.2. The distance between two codes in the sequence, where the 3rd bit is flipped, is $2^3 = 8$. Consider the code where the 3rd bit is flipped from the previous code. For the next $2^{3-1} - 1 = 3$ transitions, the bits flipped are the bits numbered 1, and 2 (< 3). A code at distance $2^2 = 4$, corresponds to a code where a bit numbered > 3 is flipped.

5.3 Efficient Increment and Decrement

Our first data structure for efficient integer representation uses the properties of the BRGC. The data structure uses $n+\log n+3$ bits to represent an integer of dimension n and performs increment or decrement operations in $2\log n + 4$ bit inspections and at most 4 bit changes.

5.3.1 The Data Structure

Store the BRGC of dimension n explicitly. Also store a parity bit for the gray code — the parity bit is '1' when the current code has even number of '1's. The n bits of the code are divided into two blocks — (i) the most active block: the rightmost $\log n$ bits of the code denoted by $X_A = x_{\log n} \cdots x_1$, and (ii) the less active block: the remaining $(n - \log n)$ bits of the code denoted by $X_L = x_n \cdots x_{\log n+1}$. A pointer P, requiring $\log n$ bits, points to the rightmost '1' in X_L, when the pointer is ready. If there is no '1' in X_L then, P points to x_n after scanning all bits of X_L. Let the pointer be denoted by the bit string $P = p_{\log n} \cdots p_1$. The pointer P becomes invalid, when the position of the rightmost '1' of X_L is changed.

Update P in the background by erasing the previous content of P and by checking one bit of X_L at a time. We use two status bits:

1. R : the ready bit, denotes whether the pointer P is pointing to the rightmost '1' of X_L, and
2. E : the erase bit, denotes whether P is being initialized by erasing the previous content.

So, the data structure requires $n + \log n + 3$ bits, where n bits are used to store the BRGC explicitly. To initialize the structure, write down n bits of the code explicitly and hence the position of the rightmost '1' of X_L is known. Initialize pointer P accordingly. If X_L contains no '1', then P points to bit x_n.

Lemma 5.3.1. *The position of the rightmost '1' of X_L is changed on an increment or decrement operation only when the rightmost bit of X_L is flipped, except for the special case, where bit x_n is flipped from '1' to '0' ('0' to '1') in an increment (decrement) operation.*

Proof. Let x_i be the bit of X_L to be flipped next by an increment (decrement) operation in general. We have, $i > \log n$. From the increment (decrement) algorithm of the BRGC, we know that, x_i is flipped only when the parity of the code is odd (even) and x_{i-1} is the rightmost '1' in the code. If $i > \log n + 1$, then x_{i-1} remains the rightmost '1' of X_L. Now, consider $i = \log n + 1$. If the bit x_i is flipped from '0' to '1' then, it becomes the new rightmost '1' of X_L. Let x_i is flipped from '1' to '0'. Before the flip, x_i was the rightmost '1' of X_L. As the bit is changed to '0', the position of the rightmost '1' of X_L is also changed. □

The pointer P is updated when the rightmost '1' of X_L is changed. By Lemma 5.3.1, this happens only when the rightmost bit of X_L is flipped. From the observations mentioned earlier, we know that in a BRGC sequence, there are $2^{\log n} = n$ transitions between two consecutive flips of two bits of X_L. We use this property to update the pointer P in the background.

A valid pointer P contains the distance of the rightmost '1' of X_L from $x_{\log n+1}$, the rightmost bit of X_L. If $x_{\log n+1}$ the rightmost bit of X_L is flipped from '0' to '1', then P

should point to it and hence P should contain all 0s. This is done by erasing the content of P. To do that, reset all bits of P to '0's, one bit at a time in the background. Consider the case when the bit $x_{\log n+1}$ is flipped from '1' to '0'. Before this flip, the pointer P was pointing to $x_{\log n+1}$, the rightmost bit of X_L and hence all the bits of P were '0's. So, there is no need to initialize P. After the flip, update P by inspecting the bits of X_L in the background, one bit at a time, starting from bit the rightmost bit of X_L. Note that, either we have to *erase* the previous content of P or *construct* P by advancing the pointer one bit at a time, not both. So, the update of the pointer P requires at most $n - \log n$ steps. As a result, there are sufficient transitions available to update P before another bit from X_L is flipped. We now describe the increment operation in detail.

5.3.2 The Increment Operation

The increment algorithm for our data structure is similar to the usual increment algorithm of the BRGC. First check the parity bit. If the parity is even, then flip the rightmost bit of X and the parity bit. Otherwise, find the rightmost '1' of X. Let x_s be the rightmost '1' of the code. Read all bits of X_A. If there is no '1' in X_A, then use the pointer P to find x_s, the rightmost '1' of X_L. Flip bit x_{s+1} and the parity bit.

Consider the special case when bit x_n is the only '1' in the code. The pointer P is ready and points to x_n. The increment algorithm reads all bits of X_A and then follows P to find the rightmost '1' of X_L. Flip bit x_n from '1' to '0' and reset bit R to '0'. No other changes are required. Note that after the changes take place, there is no '1' in X_L and P is still pointing to x_n with $R =$ '0'. So pointer P can be considered at the end of the construction mode.

The update procedure for P starts when $X_A = x_{\log n} \cdots x_1 = 10 \cdots 0$ and bit $x_{\log n+1}$, the rightmost bit of x_L, is flipped to increment the value. To update P, either we have to erase the content of P or construct P, but not both. Reset status bit R to '0' to start the procedure. If bit $x_{\log n+1}$ is flipped from '0' to '1' then, set the E to '1' to enter the *erase* mode. Otherwise, bit $x_{\log n+1}$ is flipped from '1' to '0' and pointer P enters *construction* mode. The procedure runs in the background and takes at most $n - \log n$ steps.

In the *erase* mode, the content of P is erased one bit at a time. The bit of P to be erased is determined by the BRGC denoted by the bits $x_{\log \log n} \cdots x_1$, the rightmost

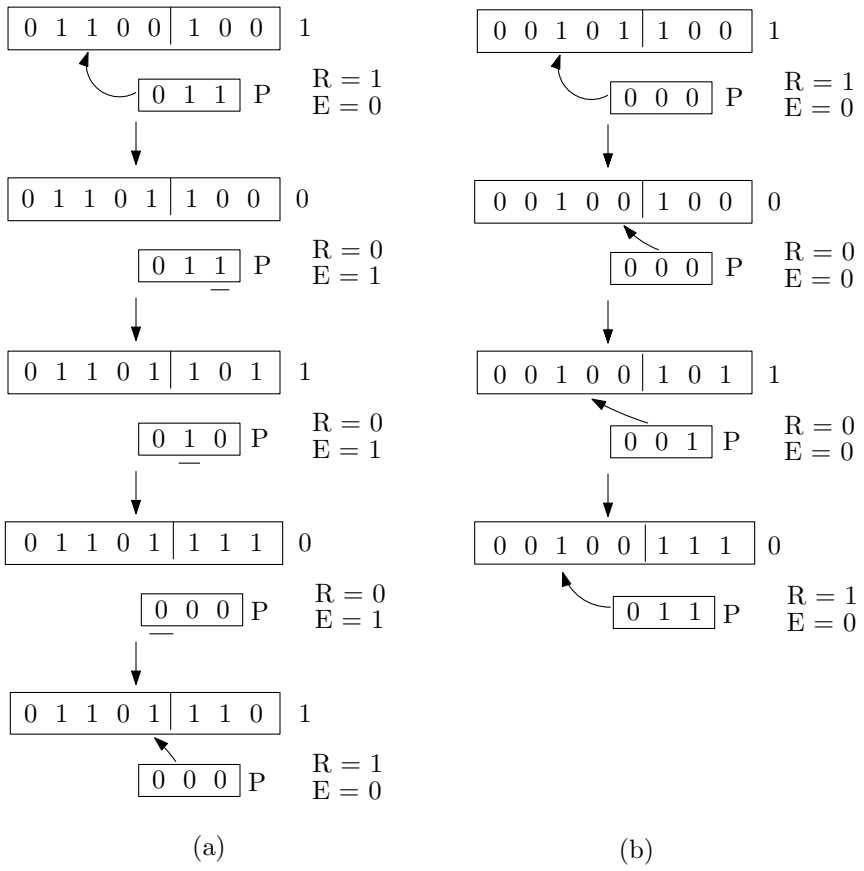

Figure 5.1: Increment operations using a constant number of bit changes.

$\log\log n$ bits of X_A. When P enters the *erase* mode, we have $x_{\log\log n}\cdots x_1 = 0\cdots 0$. By the definition of the BRGC, in the next $\log n$ transitions the contents of $x_{\log\log n}\cdots x_1$ go through the sequence from $00\cdots 0$ to $10\cdots 0$, the BRGC sequence of dimension $\log\log n$ and of length $\log n$. The status bit E is set to '0' and bit R is set to '1', once all bits of P are erased.

In the *construction* mode, the content of X_L is checked to locate the rightmost '1' by inspecting one bit at a time starting from bit $x_{\log n+1}$. The pointer P points to the bit of X_L to be inspected next, if it is not ready yet. If the bit of the code is '1', the rightmost '1' of X_L is found and the status bit R is set to '1'. Otherwise, P is incremented. So, the pointer P is a counter and the BRGC of dimension $\log n$ is used for P. As we read all bits of P, the increment of P requires only a single bit change.

Figure 5.1 shows some examples of increment operations. Figure 5.1(a) shows a sequence of increment operations where P is in erase mode and Figure 5.1(b) shows a sequence of increment operations where P is in construction mode. A detailed algorithm is included in appendix.

It is clear from above discussion that the increment operation inspects $2\log n + 4$ bits and changes at most 4 bits. The pointer P can be updated in at most $n - \log n$ steps, while we have n steps available to update P. As a result, a BRGC of dimension $n + \log n$ can be maintained using $\log n + 3$ additional bits so that the increment operation requires $2\log n + 4$ bit inspections and at most 4 bit changes.

5.3.3 The Decrement Operation

The decrement algorithm for the BRGC is similar to the increment algorithm. First, inspect the parity of the code. If the parity is odd, flip the rightmost bit of the code. Otherwise, find the rightmost '1' and flip the bit to its left. The algorithm is similar to the increment algorithm except the role of the parity bit is reversed.

The special case is when all bits of the code are '0's, the first code of the BRGC sequence. There is no '1' in X_L and pointer P points to bit x_n. In this case, flip bit x_n and set bit R to '1' as P is ready without entering erase or construction mode.

Now consider the situation that supports both increment and decrement operation. The complication arises when the pointer P enters the *erase* mode. In the *erase* mode,

the bits $x_{\log\log n}\cdots x_1$ are used to denote which bit of the pointer P to be erased. When only the increment or only the decrement is supported, these bits go through a counting sequence of length $\log n$ and can be used to point to the bit of P to be erased.

Even though this counting sequence does not exist during a mix of increment and decrement operations, the increment or decrement operation remains unaffected. Consider the situation, when we start from a BRGC code that is obtained by the flip of bit $x_{\log n+1}$ from the previous code and the pointer P enters the *erase* mode. Let we have a mix of increment and decrement operations. Due to the reflected property of BRGC, bit p_i is erased before bit p_{i+1}, for all i such that $\log n > i \geq 1$. The pointer P remains in the *erase* mode as long as bit $p_{\log n}$ doesn't get a chance to be erased. But, in between only bit x_i of the code is flipped each time, for some i such that $\log\log n > i$. The pointer P needs to be ready only when a bit in X_L is to be flipped. By that time, the content of $x_{\log\log n}\cdots x_1$ go through all possible combinations to erase P completely.

So the same data structure supports both increment and decrement operations using the same bit probe complexity. The result is summarized in the following theorem.

Theorem 5.3.2. *An integer of dimension n can be represented by a data structure that uses $n + \log n + 3$ bits so that the increment and the decrement operations require at most $2\log n + 4$ bit inspections and at most 4 bit changes per operation.*

5.4 Increment and Decrement Using Fewer Bit Inspections

Next we modify our data structure to support the increment and decrement operations using fewer bit inspections. Our new data structure reduces the number of bit inspections from $2\log n + 4$ to $\log n + O(\log\log n)$. The following property of BRGC is important in designing the data structure.

> *Observation*: Let $G(n)$ be the BRGC sequence of dimension n. Let $G_i(n)$ be the subset of $G(n)$ such that the initial code of $G(n)$ and the codes of $G(n)$ that are obtained by flipping the j-th bit from the previous code are in $G_i(n)$,

for all i and j, such that $j \geq i \geq 1$. The sequence $G_i(n)$ is the BRGC sequence $G(n-i+1)$, if the rightmost $(i-1)$ bits from each code are discarded.

0 0 0 0	**1 1 0 0**		
0 0 0 1	1 1 0 1		
0 0 1 1	1 1 1 1		
0 0 1 0	1 1 1 0		
0 1 1 0	**1 0 1 0**	**0 0** 0 0	0 0
0 1 1 1	1 0 1 1	**0 1** 1 0	0 1
0 1 0 1	1 0 0 1	**1 1** 0 0	1 1
0 1 0 0	1 0 0 0	**1 0** 1 0	1 0
(a)		(b)	(c)

Figure 5.2: (a) The BRGC sequence $G(4)$. (b) The subset $G_3(4)$. (c) The BRGC sequence $G(2)$.

An example is shown in Figure 5.2, where Figure 5.2(a) shows the BRGC sequence $G(4)$ with the codes in $G_3(4)$ in bold. Discarding the rightmost 2 bits from $G_3(4)$ gives us the BRGC sequence $G(2)$ as shown in Figure 5.2(c).

5.4.1 The Data Structure

The dimension n BRGC code is divided into k blocks and denoted by $X = X_k \cdots X_1$. The block X_i contains n_i bits and denoted by $X_i = x_{l_i} \cdots x_{l_{i-1}+1}$, where $l_0 = 0$ and $l_i = \sum_{j=1}^{i} n_j$, for all i such that $k \geq i \geq 1$. The rightmost block X_1 contains a constant number of bits. The size of the blocks are related by the equation $n_i = 2^{n_{i-1}}$, for all i such that $k \geq i > 1$. Since the dimension of the code is n, we have $\sum_{i=1}^{k} n_i = n$. From the recurrence, it is clear that $k = \log^* n$.

A pointer P_i is associated with block X_i along with two status bits E_i and R_i, for all i such that $k \geq i > 1$. Store the parity of X explicitly. When a pointer P_i is ready, it points to the rightmost '1' of X_i. If there is no '1' in X_i then, pointer P_i points to x_{l_i} after scanning all bits of X_i.

So the size of a pointer P_i is $|P_i| \geq \log n_i = n_{i-1}$. Total space usage by the pointers is

given by the following equation.

$$\begin{aligned}
\sum_{i=2}^{k} |P_i| &= \sum_{i=2}^{k} n_{i-1} \\
&= \sum_{i=1}^{k-1} n_i \\
&= n_{k-1} + n_{k-2} + \sum_{i=1}^{k-3} n_i \\
&\leq \log n + \log\log n + \sum_{i=1}^{k-3} n_i \\
&= \log n + O(\log\log n)
\end{aligned}$$

The status bits require $2(k-1) = O(\log^* n)$ bits. So the total size of the data structure is $n + \log n + O(\log\log n)$ bits.

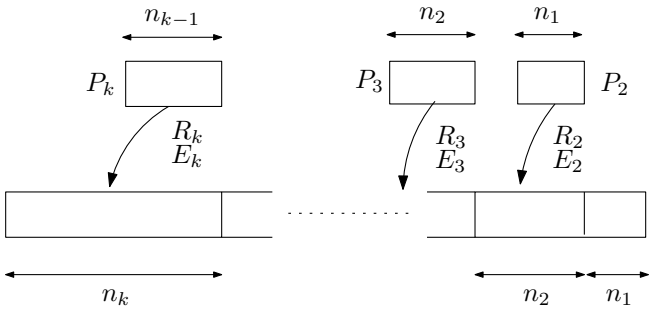

Figure 5.3: Data structure for integer representation that uses fewer bit inspections per operations.

5.4.2 Increment and Decrement

During a sequence of increment operations, consider the transitions that changes the bits only in X_i and X_{i-1}, for all i such that $k \geq i > 1$. From the observations mentioned earlier, it is clear that, there are $2^{n_{i-1}}$ transitions between two bit changes in X_i, where bits in X_{i-1}

are flipped. So, when a pointer P_i becomes invalid, we can use these $2^{n_{i-1}}$ transitions to reconstruct P_i in the background. Using a similar argument, as used for the data structure mentioned in the previous section, it can be shown that a pointer P_i is always ready when it is needed during a mix of increment and decrement operations.

To increment, first read the parity bit. If the parity is even, the rightmost bit x_1 is to be flipped. Otherwise, read all the bits of X_1. If there is at least one '1' in X_1, then we found x_s, the rightmost '1' of X. Otherwise, there is no '1' in X_1. In that case, find the rightmost block X_i that contains at least a '1'. To do that, read the status bits E_j and R_j, for all j such that $i \geq j > 1$. Note that, there is no '1' in X_j and none of the pointers P_j is in the *erase* mode. If the pointer P_j is not ready, it points to bit x_{l_j}, the leftmost bit of the block X_j. Read the pointer P_j and bit x_{l_j} to confirm that the block X_j contains no '1'. By the properties mentioned earlier, the pointer P_i either must be ready and points to the rightmost '1' of X_i or points to the leftmost bit of X_i and the lower order bits of X_i are all '0's. Read the pointer P_i to get the rightmost '1' bit x_s.

For the special case x_s is the bit x_n. Flip bit x_n and set bit R_k to '0'. Otherwise, bit x_{s+1} needs to be flipped to complete the operation. Let $x_{s+1} \in X_t$, where $k \geq t \geq 1$. Before flipping bit x_{s+1}, process pointer P_{t+1}, if it is not ready and $t < k$.. Read the status bits E_{t+1} and R_{t+1} associated with the pointer P_{t+1} (only when $t < k$). Continue the reconstruction work of P_{t+1}, if necessary, by erasing a bit of P_{t+1} (in *erase* mode), or by advancing the pointer one bit (in *construction* mode). To erase a bit of P_{t+1}, read the rightmost $\log n_t$ bits of X_t. Note that, we have to read only the corresponding bit of P_{t+1} to erase that bit. Finally, flip the parity bit and bit x_{s+1}. If x_{s+1} is the rightmost bit of X_t then, switch the pointer P_t into *erase* mode or *construction* mode depending on the value of x_{s+1} by changing the status bits E_t and R_t appropriately.

In the worst case, we might have to read all the pointers. In total, we have to read at most $\sum_{i=2}^{k} n_{i-1} + O(k) = \log n + O(\log \log n)$ bits. The number of bit changes required in the worst case is 6. It happens when P_{t+1} switches from *erase* mode to *ready* mode and at the same time P_t enters *erase* mode. The decrement operation is similar. The results can be summarized into the following theorem.

Theorem 5.4.1. *An integer of dimension n can be represented by a data structure that uses $n + \log n + O(\log \log n)$ bits so that the increment and the decrement operations require*

at most $\log n + O(\log \log n)$ *bit inspections and at most 6 bit changes per operation.*

5.5 Supporting Addition and Subtraction

The natural extensions of increment and decrement operations are addition and subtraction operations. In this section, we consider the addition and subtraction of two integer numbers. The addition operation considered has the form $X \leftarrow (X + Y) \bmod 2^n$, where X and Y are BRGCs of dimension n and m respectively such that $n > m$. The value of X is replaced by the summation. First we review the serial addition algorithm for the BRGC by Lucal [35].

5.5.1 Gray Code Addition

Let $X = x_n \cdots x_1$, $Y = y_n \cdots y_1$, and $S = s_n \cdots s_1$ be three BRGC of dimension n. The following algorithm performs the addition of the form $S \leftarrow (X + Y) \bmod 2^n$ [35].

Addition

$E_0 \leftarrow$ parity of X
$F_0 \leftarrow$ parity of Y
for $i := 1$ to n do
  $s_i \leftarrow (E_{i-1} \wedge F_{i-1}) \oplus x_i \oplus y_i$
  $E_i \leftarrow (E_{i-1} \wedge \neg F_{i-1}) \oplus x_i$
  $F_i \leftarrow (\neg E_{i-1} \wedge F_{i-1}) \oplus y_i$
end for

At the end of the loop, we must have $E_n = 0$ and $F_n = 0$ to get a valid addition.

5.5.2 Addition with Different Size Operands

Let X and Y have dimensions n and m respectively with $n > m$. The code Y can be padded with '0's to make it a code of length n. In that case $y_i = 0$, for all i such that $n \geq i > m$.

The addition can be done in two steps. In the first step, $x_m \cdots x_1$ is added with $y_m \cdots y_1$ using the serial addition algorithm 5.5.1. In the second step, $x_n \cdots x_{m+1}$ is added with $y_n \cdots y_{m+1}$ quickly. We rewrite the formulae for the summation and carry bits as follows:

$$\left.\begin{array}{rcl} s_i & = & (E_{i-1} \wedge F_{i-1}) \oplus x_i \\ E_i & = & (E_{i-1} \wedge \neg F_{i-1}) \oplus x_i \\ F_i & = & \neg E_{i-1} \wedge F_{i-1} \end{array}\right\}, \text{ for all } i \text{ such that } n \geq i > m$$

At the end of the step 1, we have the carry bits E_m and F_m. There can be three different cases possible based on the values of E_m and F_m.

Case 1 — $F_m = 0$.

We have

$$\left.\begin{array}{rcl} F_i & = & 0, \\ s_i & = & x_i \end{array}\right\}, \text{ for all } \ i \text{ such that } \ n \geq i > m$$

In other words, if $F_m = 0$ then, the remaining $n - m$ bits of the summation are same as those bits of X. Note that once the carry bit F_i becomes 0 at i-th iteration, for some i such that $i \geq m$, it remains 0 in the later iterations.

Case 2 — $F_m = 1$ and $E_m = 1$:

We have the following formula for the summation and the carry bits

$$\begin{array}{rcl} s_{m+1} & = & \neg x_{m+1} \\ E_{m+1} & = & x_{m+1} \\ F_{m+1} & = & 0 \end{array}$$

As the carry bit F_{m+1} becomes '0', we know that F_i remains '0' in the later iterations. So, we have the following formulation for the summation bit.

$$\left.\begin{array}{rcl} F_i & = & 0 \\ s_i & = & x_i \end{array}\right\} \text{ for all } \ i \text{ such that } n \geq i > m+1$$

In other words, if $F_m = 1$ and $E_m = 1$ then, the summation bit s_{m+1} is the opposite of the bit x_{m+1}. The other higher order bits of the summation are same as those bits of X.

Case 3 — $F_m = 1$ and $E_m = 0$.

Let x_{m+j} be the rightmost '1' of $x_n \cdots x_{m+1}$. We have the following formula for the summation and carry bits:

$$\begin{aligned} F_{m+1} &= \neg E_m \wedge F_m = 1 \\ E_{m+1} &= (E_m \wedge \neg F_m) \oplus x_{m+1} = x_{m+1} \\ s_{m+1} &= (E_m \wedge F_m) \oplus x_{m+1} = x_{m+1} \end{aligned}$$

Proceeding in that way we can derive the following generalized formula:

$$\left.\begin{aligned} F_{m+i} &= 1 \\ E_{m+i} &= x_{m+i} = 0 \\ s_{m+i} &= x_{m+1} \end{aligned}\right\}, \text{ for all } i \text{ such that } j > i \geq 1$$

Also, we have

$$\begin{aligned} F_{m+j} &= 1 \\ E_{m+j} &= x_{m+j} = 1 \\ s_{m+j} &= x_{m+j} \end{aligned}$$

This is similar to case 2. As a result we get the following formula

$$\begin{aligned} s_{m+j+1} &= \neg x_{m+j} \\ s_{m+j+t} &= x_{m+j+t}, \text{ for } t > 1 \end{aligned}$$

Overflow occurs when $m + j = n$. In that case, we have $s_n = \neg x_n$. Combining all the formulae, we have the following complete formulae to compute the summation bits for the case $E_m = 0, F_m = 1$, where bit x_{m+j} is the rightmost '1' of $x_n \cdots x_{m+1}$.

$$\begin{aligned} s_{m+i} &= x_{m+i}, \text{ for } j \geq i \geq 1 \\ s_{m+j+1} &= \neg x_{m+j+1} \\ s_{m+j+t} &= x_{m+j+t}, \text{ for } t > 1 \end{aligned}$$

In other words, when $E_m = 0$ and $F_m = 1$, copy the bits $x_n \cdots x_{m+1}$ into $s_n \cdots s_{m+1}$, find the rightmost '1' of $s_n \cdots s_{m+1}$ and flip the bit to its left. If s_n is that '1' then, flip s_n.

5.5.3 The Data Structure

The data structure used in our first solution is not suitable to perform an addition operation of the form $X \leftarrow (X + Y) \bmod 2^n$ efficiently, where n and m are the dimensions of X and Y respectively with $n > m$. The pointer P points to x_j, the rightmost '1' of X_L. If $j > m$ then, addition can be done efficiently, as P points to the rightmost '1' of $x_n \cdots x_{m+1}$. Add the lower m bits serially using the algorithm 5.5.1. Based on the contents of the carry bits, use the pointer P to get the rightmost '1' of $x_n \cdots x_{m+1}$ and flip the appropriate bit if necessary. On the other hand, if $j \leq m$ then, pointer P can not help in finding the rightmost '1' of $x_n \cdots x_{m+1}$. The worst case may end up in inspecting all bits of X_L. So, the addition requires $O(n + m)$ bits inspections and $O(m)$ bit changes. For a similar reason, the data structure used in the previous section does not support efficient addition / subtraction operation.

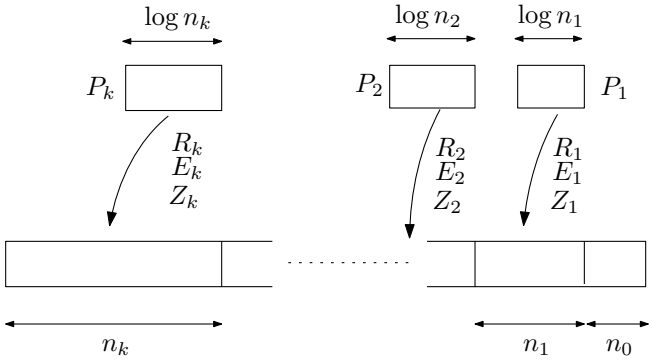

Figure 5.4: Data structure for integer representation that supports efficient addition and subtraction.

We now present our data structure and first show that the increment and decrement operations have the same complexity as the other two solutions. The n-bit BRGC number $X = x_n \cdots x_1$ is divided into $k + 1$ blocks denoted by $X_k, \cdots, X_1, X_0$ such that $X = X_k \cdots X_1 X_0$. Let n_i be the number of bits in the block X_i and is denoted by

$X_i = x_{l_i} \cdots x_{l_{i-1}+1}$, where $l_{-1} = 0$ and $l_i = \sum_{j=0}^{i} n_j$, for all i such that $k \geq i \geq 0$. The lengths of the blocks are chosen according to the following recursive formula.

$$\begin{aligned} n_0 &= \log n, \\ n_1 &= \log n, \\ n_i &= 2n_{i-1}, \text{ for all } i, \text{ such that } k \geq i > 1 \end{aligned}$$

We know that $\sum_{i=0}^{k} n_i = n$. From the recursion, it is clear that $k = \log n - \log\log n$. With each block X_i, for all i such that $k \geq i \geq 1$, there is a pointer P_i that points to the rightmost '1' of X_i, when it is ready. If there is no '1' in X_i, then P_i points to x_{l_i} after scanning all bits of X_i. In addition to E_i and R_i, each block X_i is associated with one more status bit Z_i that denotes whether or not there is a '1' in X_i. Total space used by the data structure is $n + 1 + \sum_{i=1}^{k} \log n_i + 3k = n + O((\log n)^2)$ bits. The increment and decrement operations are performed in the same manner as mentioned in the pervious section. It can be shown that increment/decrement operations require $3\log n - 2\log\log n + O(1) = O(\log n)$ bit inspections and at most 5 bit changes.

5.5.4 Addition and Subtraction

Let t be the smallest integer such that $l_t \geq m > l_{t-1}$. Addition proceeds in three steps.

1. In step 1, add $x_m \cdots x_1$ with $y_m \cdots y_1$ using the serial addition algorithm 5.5.1. During this process construct the pointers P_i and status bits E_i, R_i and Z_i, for all the i, such that $t \geq i \geq 1$.

2. Based on the value of the carry bits, in the second step, determine whether there is a carry to forward. If not, we are done. Otherwise, follow step 3.

3. In step 3, find the rightmost '1' of $x_n \cdots x_{m+1}$, denoted by x_s, and flip bit x_{s+1}.

To find x_s, the rightmost '1' of $x_n \cdots x_{m+1}$, read bits $x_{l_t} \cdots x_{m+1}$ first. If there is any '1', we are done. Otherwise, skip all blocks with no '1' bits. This is done by checking the zero status bits of these blocks. Let X_j be the rightmost block with some '1' bits such that $k \geq j > t$.

We claim that the pointer P_j is valid and points to the rightmost '1' of X_j. The pointer P_j becomes invalid when the rightmost bit of X_j is flipped. It might take at least n_j changes in X_{j-1} to complete the update of P_j. The problem might arise when there is an addition operation before the update phase of P_j is completed. But, when P_j becomes invalid, bit $x_{l_{j-1}}$, the leftmost bit of X_{j-1}, is '1'. It requires at least $2^{n_{j-1}-1} > n_j$ transitions to get all '0's in X_{j-1}, unless the first step of the addition/subtraction affects X_{j-1}. In that case, the situation is handled by reading all bits of X_j, without increasing the asymptotic complexity of the operation.

Let bit x_{s+1} be in X_r for some integer $r \geq t$. Before the flip of bit x_{s+1} takes place, check the status of pointer P_{r+1}. Perform the maintenance operations (in *erase* or *construction* mode) of pointer P_{r+1} as mentioned in the earlier sections. Finally, flip bit x_{s+1}.

The only remaining concern is the rebuilding of a pointer in the background. There are enough steps to rebuild an invalid pointer when the data structure supports only increment or decrement operations. We have to make sure that even with the addition operation, there remain enough steps to rebuild a pointer. Note that only pointer P_{t+1} might be affected adversely after an addition operation. This can be solved easily by reading all the bits of X_{t+1}, in case of an invalid pointer P_{t+1}, and rebuild it from scratch. So, the addition operation requires $O(m + \log n)$ bit probes. The subtraction operation is similar except that F_0 bit is set to the inverse of the parity of Y during the initialization of the algorithm 5.5.1. Hence, we have our final result:

Theorem 5.5.1. *An integer of dimension n can be represented by a data structure that uses $n + O((\log n)^2)$ bits so that the increment and the decrement operations require at most $O(\log n)$ bit inspections and at most 5 bit changes per operation. The addition and the subtraction between two integers of dimension n and m respectively with $n > m$ requires $O(m + \log n)$ bit probes per operation.*

Chapter 6

Conclusion

The cell probe model is the most well known and widely used model of computation for the lower bound study of a problem. The bit probe model is a variation of the cell probe model. Even though, the bit probe model is less widely used, the use of the model yields some quite interesting results. As a result, the model was never out of scene and comes to the face from time to time.

We have considered several problems in the bit probe model. In the bit probe model, we can achieve results on some problems that can break the barrier of the word size. In other words, the problems can be solved using less bit probes than the number of bits in a computer word. The predecessor problem is one of the fundamental problems in computer science. We consider a variation of the problem called the colored predecessor problem. We present a data structure for the problem that requires $O(k^2\sqrt[k]{\frac{\log U}{\log\log U}})$ bit probes for the query and $O(\frac{k\log U}{\log\log U})$ bit probes for the update operations, where U is the universe size and k is positive constant. This improves the results of [48] for the query time, while retaining the same update time. We also have shown that the results on the colored predecessor problem can be used to solve some other related problems such as existential range query, dynamic prefix sum, segment representative, connectivity problems etc.

There is still a gap between the lower bound result of [44] and the upper bound results. In [44], the authors proved a trade-off lower bound of $(\Omega(\log\log U), \Omega(\frac{\log U}{\log\log U}))$ for the query and update time in the bit probe model. For a $O(\frac{\log U}{\log\log U})$ update time, we are able to reduce the gap for the query time.

We have examined the problem of integer representation using near minimum space with few bit inspections and changes for some basic operations. The problem is only suitable for study in the bit probe model, as in the cell probe model we get trivial constant time solutions. We proved a new lower bound of $\Omega(\sqrt{n})$ for the increment operation. That means, in a counting sequence of dimension n, there exists at least one transition where at least $\sqrt{n}$ bits need to be inspected. With no additional space to store an integer of dimension n, our conjecture on the lower bound for increment is $\Omega(n)$. It can be shown that the lower bound holds for a small n by searching exhaustively. Future goal is to find a lower bound of $\Omega(n)$ for the increment operation of a counting sequence of dimension n using no additional space.

The addition of a moderate amount of extra space speeds up the operations. We present several data structures that uses little extra space for efficient increment / decrement and addition / subtraction operations. Our first solution uses $\log n + 3$ extra bits that requires $2 \log n + 4$ bit inspections and at most 4 bit changes for the increment and decrement operations. The second structure uses $\log n + O(\log \log n)$ extra bits that uses $\log n + O(\log \log n)$ bit inspections and 6 bit changes for the increment and decrement operation. Although these two structures support efficient increment / decrement operation, they are not suitable for efficient addition / subtraction. Our third data structure requires $O((\log n)^2)$ additional bits to represent an integer of dimension n. The increment and decrement operations on this structure have the same asymptotic complexity. The addition or the subtraction between two numbers of dimensions n and m with $n > m$, performs $O(m + \log n)$ bit probes. All these data structures use exponentially less space, compared to the best known previous results, while improving or retaining the same time complexity for the operations on them.

Appendix A

Increment Using a Constant Number of Bit Changes

A detailed algorithm for increment operation using tree representation that requires a constant number of bit changes is shown below:

Increment Algorithm - Tree representation

$currentNode \leftarrow root$
/* check for overflow. */
if $label(currentNode) =$ '1' then
    $label(currentNode) \leftarrow$ '0'
end if
$first \leftarrow true$
/* find the rightmost run of 1s */
while $label(currentNode) \neq$ '0' and $label(currentNode) \neq$ 'σ'
    if $label(currentNode) =$ '?' then
        /* follow the rightmost path */
        $currentNode \leftarrow rightChild(currentNode)$
    else
        /* we have a sequence of type 1 */

```
            /* mark the first node of the sequence */
            if first = true then
                fnode ← currentNode
                first ← false
            end if
            /* keep track of the nodes of the sequence */
            lnode ← currentNode
            currentNode ← leftSibling(currentNode)
        end if
    end while
    /* label the first node 'σ' */
    if fnode ≠ lnode then label(fnode) ← 'σ'
    /* label the last node '0' - end of sequence */
    label(lnode) ← '0'
    /* found the rightmost '0'- may be part of a run */
    /* we have to do extra work for 'σ' */
    if label(currentNode) = 'σ' then
        /* we have to move 'σ' up or down. */
        if currentNode is the left child of its parent then
        /* 'σ' belongs to 2(a) or 2(b)type sequence */
        /* move 'σ' upward */
            if label(leftSibling(parent(currentNode))) ≠ '0' then
                label(leftSibling(parent(currentNode))) ← 'σ'
            end if
        else
            /* 'σ' belongs to type 1 sequence. Move */
            /* 'σ' downward to next node in the sequence */
            u ← leftSibling(currentNode)
            /* find the next node in type 1 sequence */
            while label(u) = '?'
                u ← rightChild(u)
```

```
            end while
            /* label the next node with 'σ' */
            if label(u) = '1' then label(u) ← 'σ'
        end if
    end if
    if currentNode is not a leaf then
        /* we have a run of 0s. */
        /* change only the rightmost 0 into 1. */
        label(currentNode) ← '?'
        /* we have a sequence of type 2(a) */
        /* mark the first node in the sequence */
        fnode ← leftChild(currentNode)
        /* keep track of the nodes in the sequence */
        while currentNode is not a leaf
            currentNode ← rightChild(currentNode)
            lnode ← leftSibling(currentNode)
        end while
        /* mark the start of a sequence with 'σ' */
        if fnode ≠ lnode then
            label(lnode) ← 'σ'
        end if
        /* mark the end of sequence with label '0' */
        label(fnode) ← '0'
    end if
    /* the label can be '1' or '?' depending on */
    /* whether or not if becomes part of a run of 1s */
    /* currentNode is a leaf now */
    if currentNode is the right child of its parent then
        /* determine whether it is a part of a run of 1s */
        /* in that case, we have a type 2(b) sequence */
        first ← true
```

```
        while currentNode is the right child and
                label(leftSibling(currentNode)) = '1'
            /* mark the first node of the sequence */
            if first = true then
                fnode ← leftSibling(currentNode)
                first ← false
                label(currentNode) ← '?'
            end if
            /* keep track of the nodes in the sequence */
            lnode ← leftSibling(currentNode)
            currentNode ← parent(currentNode)
        end while
        /* mark the last node in the sequence */
        if lnode not null then label(lnode) ← '0'
        /* label the start of sequence with 'σ' */
        if fnode ≠ lnode then
            label(fnode) ← 'σ'
        end if
    end if
    label(currentNode) ← '1'
```

Appendix B

Efficient Increment Algorithm

We now describe the increment algorithm of section 5.3.2 in detail.

Increment Algorithm

Determine the state of P by reading the status bits R and E
if P is in *erase* mode then
    reset the bit p_{i+1} to '0', where i is the BRGC number
    denoted by the bits $x_{\log\log n} \cdots x_1$
    if $i = \log n - 1$ then
        switch P to *ready* mode by setting E to '0' and R to '1'
    end if
else if P is in *construction* mode then
    read the pointer $P = p_{\log n} \cdots p_1$
    let j be the BRGC number denoted by P
    read the bit $x_{\log n+1+j}$ from X_L
    if $x_{\log n+1+j}$ = '1' then
        switch P to *ready* mode by setting R to '1'
    else if $j < n - 1 - \log n$ then
        increment P
    end if
else Let the rightmost '1' of the code is denoted by x_s
end if

```
if the parity is even then
    s ← 0
else
    read the bits X_A = x_{log n} x_{log n−1} ··· x_1
    if X_A = 10 ··· 0 then
        s ← log n
    else if X_A = x_{log n} ··· x_1 = 0 ··· 0 then
        pointer P is ready and points to the rightmost '1' of X_L
        s ← log n + 1 + j, where j
        is the BRGC number denoted by P
    else
        We have 1 ≤ s < log n, and
        s can be determined from the bits of X_A already read
    end if
end if
flip the parity bit
if s = n then
    flip bit x_n
    reset R to '0'
else
    flip bit x_{s+1}
    if s = log n then
        reset R to '0'
        if x_{log n+1} is '1' then
            switch P to erase mode by setting E to '1'
        end if
    end if
end if
```

Bibliography

[1] M. Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8:235–247, 1988.

[2] S. Alstrup, G. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proceedings of 33rd Annual Symposium on Theory of Computing*, pages 476–482, 2001.

[3] S. Alstrup, T. Husfeldt, T. Rauhe, and S. Skyum. Marked ancestor problems. In *Proceedings of 39th Annual Symposium on Foundatons of Computer Science*, pages 534–543, 1998.

[4] A. Andersson. Sublogarithimic searching without multiplications. In *Proceedings of 36th Annual Symposium on Foundations of Computer Science*, pages 655–665, 1995.

[5] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Anjual ACM Symposium on Theory of Computing*, pages 335–342, 2000.

[6] D. A. Mix Barrington, C. J. Lu, P. B. Miltersen, and S. Skyum. Searching constant width mazes captures the AC^0 hierarchy. In *STACS 1998, Lecture Notes in Computer Science*, volume 1372, pages 73–83, 1998.

[7] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[8] J. Boothroyd. Algorithm 246 Graycode. *Communications of the ACM*, 7(12):701, 1964.

[9] A. Brodnik, S. Carlsson, M. L. Fredman, J. Karlsson, and J. I. Munro. Worst case constant time priority queue. In *Proceedings of the 12th Annual ACM/SIAM Symposium On Discrete Algorithms*, pages 523–528, 2001.

[10] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? *SIAM Journal on Computing*, 31(6):1723–1744, 2002.

[11] S. Carlsson, J. I. Munro, and P. V. Poblete. An implicit priority queue with constant insertion time. In *Proceedings of 1st Scandinavian Workshop on Algorithm Theory*, pages 1–13, 1988.

[12] M. J. Clancy and D. E. Knuth. A programming and problem-solving seminar. *Tech Report, Computer Science Dept, School of Humanities and Science, Stanford University*, STAN-CS-77-606, 1977.

[13] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th ACM/SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[14] M. Cohn and S. Even. A Gray code counter. *IEEE Transcations on Computers*, pages 662–664, 1969.

[15] M. W. Doran. The Gray code. *CDMTCS Research Report, www.cs.auckland.ac.nz/CDMTCS//researchreports/304bob.pdf*, 2007.

[16] P. Elias. Efficinent storage retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

[17] P. Elias and R. A. Flower. The complexity of some simple retrieval problems. *Journal of the ACM*, 22:367–379, 1975.

[18] P. Van emde boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[19] D. Eppstein. Dynamic connectivity in digital images. *Tech Report 96-13, Univ of California, Irvine, Dept of Info and Comp Sci*, 1996.

[20] M. C. Er. Remark on algorithm 246 (Gray code). *ACM Transactions on Mathematical Software*, 11(4):441–443, 1985.

[21] P. Erdös and R. Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, 35:85–90, 1960.

[22] A. F. Fischman. A Gray code counter. *IRE Transactions on Electronic Computers*, EC-6:120, 1957.

[23] G. S. Frandsen, P. B. Miltersen, and S. Skyum. Dynamic word problems. *Journal of the ACM*, 44:257–271, 1997.

[24] M. L. Fredman. Observations on the complexity of generating quasi-Gray codes. *SIAM Journal on Computing*, 7:134–146, 1978.

[25] M. L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, 1982.

[26] M. L. Fredman and M. R. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[27] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[28] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Compuer Systems Science*, 48:533–551, 1994.

[29] F. Gray. Pulse code communications. In *U.S. Patent 2632058*, 1953.

[30] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *ICALP 1997, Lecture Notes in Computer Science*, volume 1256, pages 594–604, 1997.

[31] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of 30th Annual ACM Symposium on Theory of Computing*, pages 342–349, 1998.

[32] T. Husfeldt and T. Rauhe. Hardness results for dynamic problems by extension of fredman and saks' chronogram method. In *ICALP 98, Lecture Notes in Computer Science*, volume 1443, pages 67–77, 1998.

[33] T. Husfeldt, T. Rauhe, and S. Skyum. Lower bounds for dynamic transitive closure, planar point location, and parenthesis matching. In *SWAT 96, Lecture Notes in Computer Science*, volume 1097, pages 198–211, 1996.

[34] K. Krohn and J. Rhodes. Algebraic theory of machines I. Prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965.

[35] H. Lucal. Arithmetic operations for digital computers using a modified reflected binary code. *IEEE Transactions on Computers*, pages 449–458, 1959.

[36] J. C. Majithia. Simple design for up/down Gray-code counters. *Electornis Letters*, 7(22):658–659, 1971.

[37] P. B. Miltersen. The bit probe complexity measure revisited. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 662–671, 1993.

[38] P. B. Miltersen. Cell probe complexity - a survey. In *Pre-Conference Workshop on Advances in Data Structures at the 19th Conference on the Foundations of Software Technology and Theoretical Computer Science*, 1999.

[39] P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM/SIAM Symposium On Discrete Algorithms*, pages 11–12, 2005.

[40] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. *journal of computer and system sciences*, 57:37–49, 1998.

[41] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130:203–236, 1994.

[42] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, 1969.

[43] J. Misra. Remark on algorithm 246:Graycode[Z]. *ACM Transactions on Mathematical Software*, 1(3):285, 1975.

[44] C. W. Mortensen, R. Pagh, and M. Patrascu. On dynamic range reporting in one dimension. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 104–111, 2005.

[45] J. I. Munro. Tables. In *FSTTCS 1996, Lecture Notes in Computer Science*, volume 1180, pages 37–42, 1996.

[46] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[47] R. Pagh. Low redundancy in static dictionaries with $O(1)$ lookup time. In *ICALP 1999, Lecture Notes in Computer Science*, volume 1644, pages 595–604, 1999.

[48] C. E. Pătraşcu and M. Pătraşcu. On dynamic bit-probe complexity. In *ICALP 2005*, pages 969–981, 2005.

[49] J. Radhakrishnan, V. Raman, and S S. Rao. Explicit deterministic constructions for membership in the bitprobe model. In *ESA 2001, Lecture Notes in Computer Science, 2161*, pages 290–299, 2001.

[50] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[51] V. Raman and S. S. Rao. Static dictionaries supporting rank. In *ISAAC 99, Lecture Notes in Computer Science*, volume 1741, pages 18–26, 1999.

[52] P. Sen. Lower bounds for predecessor searching in the cell probe model. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity*, pages 73–83, 2003.

[53] R. Tamassia and F. P. Preparata. Dynamic maintenance of planar digraphs with applications. *Algorithmica*, 5:509–527, 1990.

[54] M Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 343–350, 2000.

[55] D. E. Willard. Log-logarithmic worst case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17:81–84, 1983.

[56] B. Xiao. New bounds in cell probe model. *PhD Thesis*, 1992.

[57] A. C. C. Yao. Should tables be sorted? *Journal of The ACM*, 28:615–628, 1981.