CS 224: Advanced Algorithms	Spring 2023
Lecture 4 — January 23rd, 2023	
Prof. Prashant Pandey	*Scribe: Joe Rodman*

1 Overview

In the last lecture we learned about x-fast and y-fast trees.

In this lecture we learn about Succint Data Structures.

Some Logistics:

- *Paper Report* – The paper report is due today, January 23rd 2023
- *Scribing* – Everyone must sign up to scribe for a lecture. Email the TA to choose a date. The schedule on the class website shows the available dates. You must sign up for a lecture by the start of next class (Wed january 25th) or you will randomly be assigned a date
- *Assignment 1* – Assignment one is due next week. Start early as there are plenty of edge cases to cover. The hope is that the assignment will help you better understand VEB Trees
- *Guest Lectures* – We will have three guest lectures this semester - one on ANN (approximate nearest neighbor problem), one on ESH, and one on Succinct Data Structures

2 Succinct Data Structures

GOAL: Store data as compactly as possible. Store N items in a ”small space” (often a static space). We try to get as close to the theoretical optimum (OPT) as possible. In more simple terms, instead of using O(n) words to store the data, succinct data structures try to store the data in O(n) bits

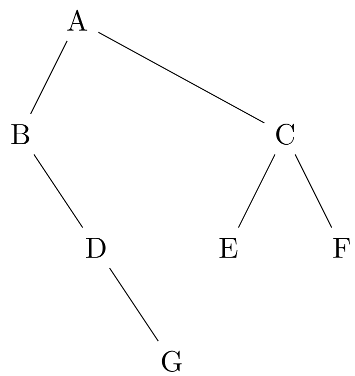
** o(OPT) means less than or equal to OPT

The three types of data structures we will go over are:

- *Implicit* – Space = OPT + O(1). An example of an implicit data structure is a sorted array representation of a binary search tree
- *Succinct* – Space = OPT + o(OPT). In other words at most 2*OPT
- *Compact* – Space = O(OPT). A Binary Search Tree is an example

2.1 Level Order Traversal of Binary Tries

GOAL: Succinctly represent tries



Method: Iterate through nodes in level order, and for each one, we write down 2 bits. 11 represents a node with a left and a right child, 10 represents just a left child, 01 represents just a right child, and 00 represents a leaf node.

Reminder: The level order traversal of the above tree is ABCDEFG

Result: 11011101000000

We add a leading 1 bit to represent the parent node, and the result uses 2N + 1 bits: 111011101000000

Question: How can we find the index of the leftIndex and rightIndex nodes of a node?

Answer: We use the equations leftIndex = 2i and rightIndex = 2i+1

Example: Take node D, with index 4 (in the level order traversal). The left node of D is index 2*4 = 8, and the right node is at index 2*4 + 1 = 9. Looking at the binary representation, index 8 is a 0 and index 9 is a 1, which represents the node G

2.2 Rank and Select

Rank(i): the number of 1's at or before index i

Rank Ex: In the previous binary representation of the trie, Rank(9) = 7

Select(j): the index of the jth bit set to 1

Select Ex: In the previous binary representation of the trie, Select(4) = 5

Assumption: If we can do these operations in constant time on an n-bit string, we could represent a binary trie that supports the three important operations: leftNode, rightNode, and parentNode using the following equations.

- *leftChild* = 2rank(i)
- *rightChild* = 2rank(i) + 1
- *parent* = select(i/2)

The Rank Algorithm Note that this is taken directly from the textbook because I didn't fully understand what was discussed in class.

Rank This algorithm was developed by Jacobsen, in 1989 [2]. It uses many of the same ideas as RMQ. The basic idea is that we use a constant number of recursions until we get down to sub-problems of size k = log(n)/2. Note that there are only 2k = n possible strings of size k, so we will just store a lookup table for all possible bit strings of size k. For each such string we have k = O(log(n)) possible queries, and it takes log(k) bits to store the solution of each query (the rank of that element). Nonetheless, this is still only O(2k · k log k) = O(n log(n) log log(n)) = o(n) bits.

First Attempt We will split the bit string into n/ log2(n) chunks of size log2(n). To find rank(i), we need to find (rank of i in its chunk) + (number of 1's in all preceding chunks). We will show how to find rank(i) within a chunk. But we also need, for each chunk, the total number of 1's among all of the preceding chunks. There are n/ log2(n) chunks, and for each of them we have to store a number (with log(n) bits). So we can store all the data using O(n/ log n) bits which we can afford.

Second Attempt Now we have chunks of size log2 n. The solution is to use one more level of recursion. We will split into 2n/ log(n) subchunks of size log(n)/2. The rank within the subchunks can be found using the lookup table. The problem is to find the number of one bits in the preceding subchunks. Note that we have 2n/ log n subchunks. But the number of ones in the preceding subchunks is not more than log2 n because we are within a chunk of size log2 n. So we can store each of these 2n/ log n numbers by O(log log(n)) bits. So the total space of this part is o(n) as well.

References

[1] G. Franseschini, R.Grossi *Optimal Worst-case Operations for Implicit Cache-Oblivious Search Trees*, Prooceding of the 8th International Workshop on Algorithms and Data Structures (WADS), 114-126, 2003

[2] G.Jacobson *Succinct Static Data Structures*, PHD.Thesis, Carnegie Mellon University, 1989. J. Comput. Syst. Sci., 58(1):137–147, 1999.