CS 224: Advanced Algorithms Spring 2026

Lecture Lecture 4: Succinct binary tries (Rank) — 01/21/2026

Prof. Prashant Pandey *Scribe: Woodward Galbriath*

1 Overview

In the previous lectures we discussed Van Emde Boas Trees X-fast trees and Y-fast trees, which are all data structured designed to speed up operations on a fixed size universe U which is sparsely populated. IE. the number of elements n is $U >> n$. Today we are developing a data structure that is optimal in terms of space for holding n keys while retraining lookup speed of time $\mathcal{O}(n)$, called a succinct binary trie. The succinct binary trie leverages the big universe little universe approach we saw previous in Y-fast trees.

2 Notation

- n number of keys we would like to store
- U universe size.
- w machine word size, typically this is $w = \mathcal{O}(n * log(U))$ bits
- x is Big O of n, $x = \mathcal{O}(y) \leq y$, that is y is the upper bound of x
- x is little o of y, ie $x = o(y) \geq y$, that is y is the lower bound of x
- The information theoretic minimum space to store n keys is $\mathcal{O}(n)$ bits = opt

3 Motivation

- Today we hope to optimize a data structure for space instead of time efficiency
- space of know data structures
  - A binary search tree (BST) will have space complexity $\mathcal{O}(n)$
  - A Van Emde Boas Tree will have space complexity $\mathcal{O}(U)$
- **Today's goal:** find a small space static data structure to store n keys.

3.1 Static Data structures

- A static data structure is a data structure for which:
  - The data structure is of fixed size
  - We know all keys before hand
  - we can not update the data structure after creation.

3.2 Space complexity

- It is easy to get a data structure that is linear in space, but this is sub-optimal.
- Call a linear space data structure a data structure with space complexity

$$\mathcal{O}(n) \text{ machine words} = \mathcal{O}(nlog(U)) \text{ bits} = \mathcal{O}(n)w \text{ bits}$$

- The information theoretic minimum space to store n keys is $\mathcal{O}(n)$ bits = opt

3.3 Ideas of a small space data structure

- There are 3 ideas of a small space data structure
  1. Implicit data structure: has space complexity opt + $\mathcal{O}(1)$ bits, typically this is just the data permuted in order. An example would be a min perfect hash.
  2. Succinct data structure: Space complexity opt $+ o(\text{opt})$ bits. This is what we are building today
  3. Compact data structure: space complexity $O(\text{opt})$ bits.
- Today we are building a succinct data structure

4 Succinct binary tries

4.1 Key idea

- Build a binary search tree with no pointers
- Think of this as an array based binary search tree

4.2 Level order representation

- Suppose we have a standard binary search tree (BST) shown in Figure 1 with:
  - each pointer being of size w that is a machine word
  - thus the total size complexity is the number of keys you need to store as well as 2 pointers for each nodes children, that is opt $+ 2nw$
- to achieve a succinct data structure we want space complexity $opt + \mathcal{O}(opt)$, note that $opt + 2n = opt + \mathcal{O}(opt)$, thus we can achieve this by eliminating the w.

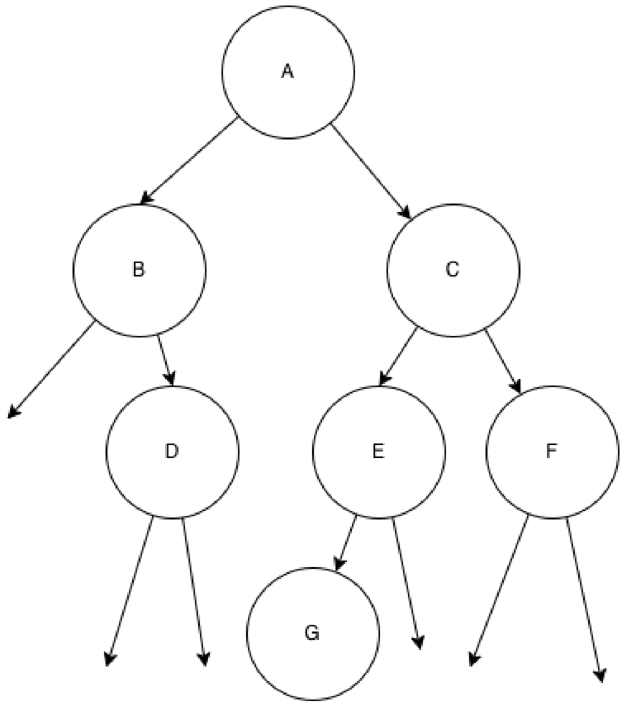


Figure 1: Binary search tree

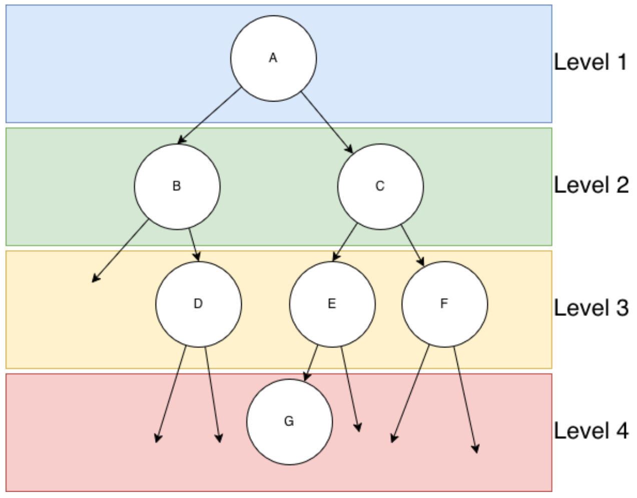


Figure 2: Caption

4.3 Data structure

- Suppose we are thinking about our tree in terms of levels as shown in Figure 2
- We can store a bit array of size $2n$ bits
- Then we add our nodes to the bit array left to right one level at a time.
- We can construct our bit vector level representation of the binary search tree using Algorithm 1.
- The result of running this procedure on Figure 1 can be seen in Equation 1

Algorithm 1 Instantiate level representation of bit vector BST

```
for node in level order do
    Add a zero to the bit array if the node has no left child otherwise add a one to the bit array
    Add a zero to the bit array if the node has no right child otherwise add a one to the bit array
end for
```

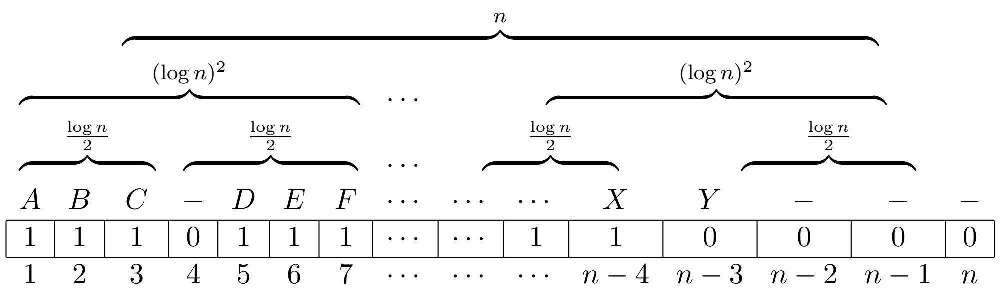
$$\begin{array}{cccccccccccccccc} A & B & C & - & D & E & F & - & - & G & - & - & - & - & - & - \\ \hline \multicolumn{1}{|c|}{1} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \end{array} \tag{1}$$

4.4 Navigation

- Given we have bit array level representation of a BST as shown in Equation 1, how can we navigate from node internal node index i to its children?
- Here internal node i means the index of the ith non-zero node in the bit vector.
- Note: Here we are assuming that the bit vector is one indexed
- We get:
  - The left descendant of node i with $2i$
  - The right descendant of node i with $2i + 1$
- this can be proven by induction but we did not go over this in class.
- as an example node internal node D has index 5, and we can see that $5 * 2 = 10$ which is the index of node G its left descendant
- Recall that our bit vector is $2n$ space, thus if we do not need to store any additional information to traverse it we will have a succinct data structure.

5 Rank and selection in bit strings

- call $rank(i)$ the number of nodes in the bit vector before the ith non-zero element.
- call $select(j)$ the index of the jth non-zero element.
- For internal node i and index j we can use these operations to get:
  - $left\_child(i)$= $2 * rank(i)$
  - $right\_child(i)$= $2 * rank(i) + 1$
  - $parent(j)$= $select(\lfloor select(\frac{j}{2}) \rfloor)$
- Example: get parent of E
  - E has index 6, this $i = 6$
  - $\lfloor \frac{6}{2} \rfloor = 3$
  - $select(3)$ gives node C


n
$(\log n)^2$
$(\log n)^2$
$\frac{\log n}{2}$
$\frac{\log n}{2}$
$\frac{\log n}{2}$
$\frac{\log n}{2}$
A B C $-$ D E F $\cdots$ $\cdots$ $\cdots$ X Y $-$ $-$ $-$
1 1 1 0 1 1 1 $\cdots$ $\cdots$ 1 1 0 0 0 0
1 2 3 4 5 6 7 $\cdots$ $\cdots$ $\cdots$ $n-4$ $n-3$ $n-2$ $n-1$ n

(2)

5.1 Rank algorithm

- This was solved by Jacobson in 1989 [1].
- the main idea is big universe little universe, so that is we want to split our problem into many smaller data structures that can be brute forced efficiently, then run some more complex algorithm on the brute force results.
- This algorithm has the following 4 key steps
  1. Develop a lookup table
     - Split the bit vector into chunks of size $\frac{log(n)}{2}$, as show in Equation 2
     - create a lookup table mapping to each chunk. So that is each binary string of length $\frac{log(n)}{2}$ will be a key in the lookup table.
     - This lookup table will have size $\sqrt{n} * log(n) * log(log(n))$
  2. split the bit vector into chunks of size $log(n)^2$, as shown in equation 2. This will have size $\mathcal{O}(\frac{n}{log(n)})$ bits
  3. split each chunk into sub-chunks of size $\frac{log(n)}{2}$ this will have size of $\mathcal{O}(n)$ bits

4. Aggregate the ranks, so can calculate the rank of i as rank of the chunks up to that point + relative rank of the sub-chunk up to that point + relative rank of the elements within that sub-chunk.
   - this will have time complexity $\mathcal{O}(1)$
   - this will have space complexity $\mathcal{O}(n * \frac{log(log(n))}{log(n)}) = opt + \mathcal{O}(opt)$bits

- note that in an actual implementation once we have completed the pre-processing to construct our rank table we no longer need to store the original bit array.

References

[1] Guy J. Jacobson Succinct static data structures 1989, CMU-CS-89-112 https://dl.acm.org/doi/10.5555/915547