CS 7280: Data Str & Alg Scalable Comp Spring 2025

Lecture 4

Prof. Prashant Pandey *Scribe: Rohan Jamadagni*

1 Overview

In previous lectures, we focused on reducing the time complexity of queries, insertions, and deletions from $O(\log n)$ to $O(\log\log u)$. This lecture explores methods for reducing space complexity to sublinear, specifically smaller than $O(n)$.

1.1 Recap: Speed Comparison for $n = 16$ billion (2^{34})

We compare the query times of different data structures for a dataset of $n = 2^{34}$ elements:

1. Binary Search Tree (BST): $\log_2 n = \log_2 2^{34} = 34$
2. B^+ Tree (with branching factor $b = 16$): $\log_b n = \log_{16} 2^{34} = \frac{\log_2 2^{34}}{\log_2 2^4} = \frac{34}{4} = 8.5 \approx 9$
3. Y-fast Tree: $\log\log U = \log\log 2^{64} = \log 64 = 6$ (assuming a universe size of $U = 2^{64}$)

A key advantage of Y-fast trees is that their query time remains constant as n increases, unlike BSTs and B^+ trees, whose query times grow with n.

2 Succinct Data Structures

The primary goal of succinct data structures is to construct a data structure that uses "small space" (often static) and to store n items from a universe U while supporting "fast" operations such as queries and insertions.

2.1 Notions of Space Efficiency

The information-theoretic lower bound is a fundamental concept in space efficiency. To represent n distinct items, we require at least $\lceil \log_2 n \rceil$ bits. This represents the minimum space theoretically required to distinguish between the n items. For example, to represent 20 distinct items, we need $\lceil \log_2 20 \rceil = 5$ bits.

We define OPT (optimal) as the information-theoretic lower bound. We consider three classes of space-efficient data structures:

1. **Implicit:** Uses OPT + $O(1)$ bits, meaning the overhead is constant.

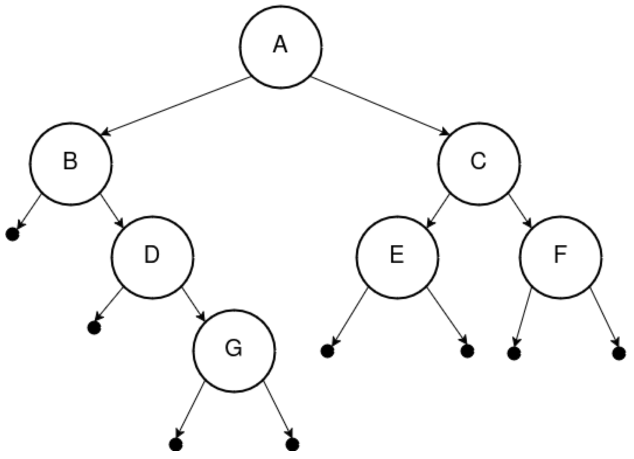
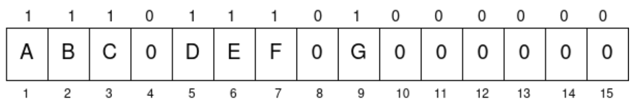


Figure 1: A simple tree with 7 elements



1	1	1	0	1	1	1	0	1	0	0	0	0	0	0
A	B	C	0	D	E	F	0	G	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 2: Level Order Representation of Fig 1

2. **Succinct:** Uses OPT + $o(\text{OPT}) < 2 \cdot OPT$ bits. The overhead is always less than OPT. An example is the level-order representation of binary trees (discussed later).

3. **Compact:** Uses $O(\text{OPT})$ bits. The space is within a constant factor of the optimal space. Suffix trees are an example.

These space bounds (implicit, succinct, and compact) are often a factor of w smaller than linear space, where w is the machine word size (64 or 32 bits).

2.2 Level-Order Representation of Binary Trees

Sorted arrays require $O(w \cdot n)$ bits but do not support efficient updates. Binary search trees (BSTs) have an extra overhead of storing left and right pointers for each node $O(2 \cdot w \cdot n)$ bits. The level-order representation offers an alternative approach.

In the level-order representation, we traverse the tree level by level and represent each node with two bits. A '1' indicates the presence of a child (left or right), and a '0' indicates its absence. The first bit corresponds to the left child, and the second to the right child. A tree with n nodes thus uses $2n$ bits. Nodes present in the tree are called *internal nodes* and are represented by '1's. Null pointers, representing missing children, are called *external nodes* and are represented by '0's.

The construction procedure involves first appending external nodes for any missing children. Then, for each node in level order, we write '0' if it is an external node and '1' if it is an internal node. Finally, we add a leading '1' to the bit string, resulting in a total of $2n + 1$ bits.

Figure 2 shows the level order representation of Figure 1

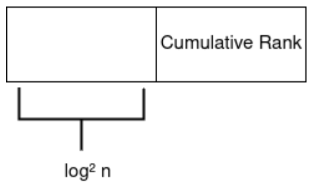


Figure 3: Space Complexity is $O(\frac{n}{\log^2 n} \cdot \log n) = O(\frac{n}{\log n})$ bits

2.3 Navigation

The left and right children of the i-th internal node are located at positions $2i$ and $2i+1$, respectively. It is important to note that i is incremented only for internal nodes. A key problem arises: to find the children of a node, we need to determine its index i as an internal node. This requires knowing the number of '0's and '1's encountered during the level-order traversal up to that node.

3 Rank and Select

To solve the aforementioned problem, the following operations are introduced:

- $\text{rank}_1(i)$ returns the number of '1's at or before position i
- $\text{select}_1(j)$ returns the position of the j-th '1'.

If these operations can be performed in $O(1)$ time, the level-order representation would use $O(2n)$ bits while supporting constant time child and parent traversals. The following formulas express tree navigation using rank and select:

- $\text{left-child}(i) = 2 \cdot \text{rank}_1(i)$
- $\text{right-child}(i) = 2 \cdot \text{rank}_1(i) + 1$
- $\text{parent}(i) = \text{select}_1(\lfloor i/2 \rfloor)$

3.1 Rank

The rank algorithm, first introduced by Jacobsen[2] in 1989, efficiently computes $\text{rank}_1(i)$. The core idea is to divide the array into smaller chunks and store the cumulative rank in lookup tables for each chunk. The algorithm proceeds as follows:

1. Use lookup tables for substrings of length $\frac{1}{2}\log n$. Space complexity is $O(\sqrt{n} \cdot \log n \cdot \log\log n)$
2. Split the array into $\log^2 n$-bit chunks. Illustration shown in Fig. 3
3. Further split each chunk into $\frac{1}{2}\log n$-bit subchunks. Illustration shown in Fig. 4

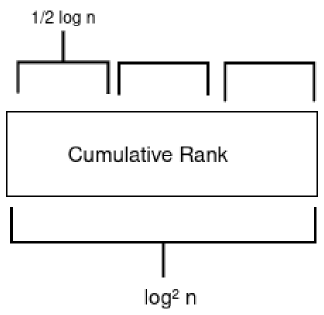


Figure 4: Space Complexity is $O(\frac{n}{\log n} \cdot \log\log n) = O(n)$ bits

The rank of an element at position i, denoted as Ranks(i), can be computed by summing three components: the rank of the containing chunk, the relative rank of the containing subchunk within that chunk, and the relative rank of the element within its subchunk. Formally:

$$\begin{aligned} \text{Rank}(i) =& \text{Rank of Chunk} \\ &+ \text{Relative rank of subchunk within the chunk} \\ &+ \text{Relative rank of element within the subchunk} \end{aligned} \tag{1}$$

This approach achieves $O(1)$ query time with an extra space overhead of $O(n \cdot \frac{\log\log n}{\log n})$ bits.

3.2 Select

The select operation, introduced by Clark and Munro[1], efficiently finds the position of the j-th '1'. While the implementation details were not covered in detail, it is important to note that the select operation can be performed in $O(1)$ time with an additional space overhead of $O(\frac{n}{\log\log n})$ bits.

3.3 SIMD Operations to Reduce Space Overhead

Single-Instruction Multiple Data (SIMD) CPU instructions can operate on entire words or even cache lines (e.g., with AVX512). These instructions enable constant-time implementations of rank and select without the additional space overhead typically required. The homework is to think about how the following SIMD instructions could be used to implement these algorithms.

- **PDEP** (Parallel Bits Deposit): This instruction gathers bits from scattered positions in a source operand and packs them into contiguous bits of a destination operand, according to a mask.
- **LZCNT** (Count Leading Zero Bits): This instruction counts the number of consecutive zero bits starting from the most significant bit of the operand.

4 Summary

This lecture focused on reducing space complexity in data structures, complementing previous discussions on time complexity. We introduced succinct data structures, aiming for minimal space usage with efficient queries. Key concepts included the information-theoretic lower bound (OPT) and the categorization of space-efficient structures (implicit, succinct, compact). We explored the level-order representation of binary trees, which requires rank and select operations for navigation. We then discussed the rank algorithm and the select operation itself, along with how SIMD instructions can offer constant-time performance and reduce space overhead for these operations.

References

[1] David R. Clark and J. Ian Munro. "Efficient suffix trees on secondary storage". In: *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '96. Atlanta, Georgia, USA: Society for Industrial and Applied Mathematics, 1996, pp. 383–391. ISBN: 0898713668.

[2] Guy Joseph Jacobson. "Succinct static data structures". AAI8918056. PhD thesis. USA, 1988.