Mechano-Informatics Special Lecture II,
University of Tokyo,
May 1st, 2024

Succinct Data Structure for Scalable Knowledge Discoveries

Yasuo Tabei

Team Leader at

Succinct Information Processing Team,

RIKEN-AIP

Announcements

- Please click the “raise hand” button if you can see the slides and hear me clearly.
- The presentation slides are available for download from the website.
- The presentation will last approximately 80-90 minutes.
- Feel free to ask any questions using the chat window during the presentation.

Overview

- Topic: Compressed Data Structure for Enhancing Scalability in Machine Learning and Data Mining
- Contents: Introduction to Succinct Data Structures (SDS)
- Prerequisites: Basic knowledge of data structures, algorithms, and programming is recommended.
- Agenda:
  - First Half: Introduction and exploration of Succinct Data Structures (SDS)
  - Second Half: Applications of SDS in Data Mining and Machine Learning

Big data analysis

- Huge amounts of data, known as "big data," are everywhere in research and business today.

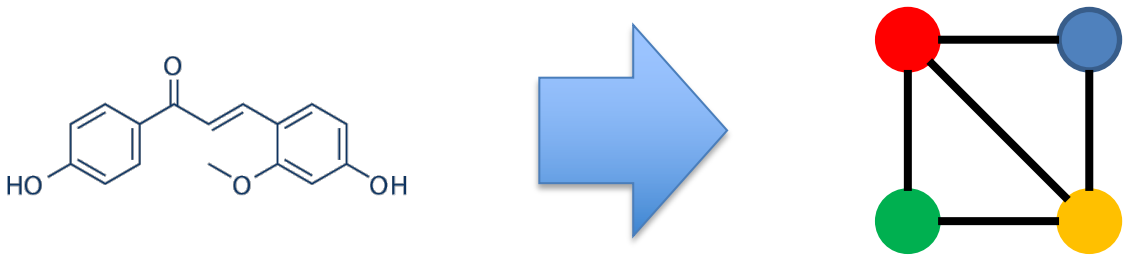
- Three Main Features of Big Data:
  1. Size: The huge amount of data collected.
  2. Speed: The fast pace at which new data comes in and needs to be used.
  3. Types: The wide range of different kinds of data and sources.

- Challenges:
  - Working with these big sets of data is a tough problem in many areas of study.

- Need for New Tools:
  - There is a strong and growing need to create new and better tools to make good use of big data.

New Drug Discovery from Big Chemical Databases

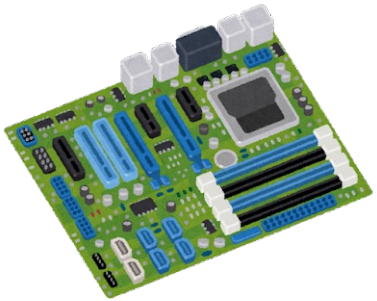
- Scientists have made big databases that keep information on lots of different chemical compounds.
  - For example, there's one called NCBI PubChem that has details on 70 million compounds.
- It's thought that there can be as many as 10 to the power of 60 different ways these compounds can vary.
- These databases are getting bigger all the time, with more compounds being added.
- How Compounds are Shown:
  - Each chemical compound can be shown as a picture that looks like a web or network, which we call a graph.
- The Need for Special Tools:
  - To find useful information in these huge databases with lots of graphs, we need special tools that can sort through and understand all that data.



Databases of Human Genomes

- A genome is like a long string of letters, with each letter standing for a part of DNA.
  - Ex: ATGCGCATAGATGC
- Scientists keep huge collections of human genomes to study them.
  - Ex: The "100,000 genomes project" has a database that's 72 terabytes big.
- When you compare the genomes from two people, they are almost exactly the same—99.9% the same, in fact.
- The database of human genomes is like a big collection of very similar strings of letters—so there's a lot of repetition.
- Scientists have come up with new ways to handle these long, repetitive strings to learn more from them.

Computer memory hierarchy



- Access Time Gap:

  Different memory devices have big differences in how quickly they can be accessed.

- Efficient Data Handling:

  To speed up data processing, it is important to store data in a compact form at the higher, faster levels of memory.

Fast

Operation	Access (sec)	Size
CPU register	~10^{-10}	~10 words
Cache memory	~10^{-9}	~256KB
RAM	~10^{-7}	~1TB
Hardisk	~10^{-2}	~10TB

Small

Slow

https://en.wikipedia.org/wiki/Memory_hierarchy

Large

Big Data Analysis Essentials: Data Structures & Algorithms

- Key Features Required:

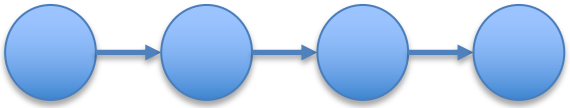
1. Compactness: Efficient space usage.
2. Operational Diversity: Ability to support operations on various data types.
3. Dynamic Updating: Capability to modify data structures as needed.

- Progress Note:

Significant advancements in compressed data structures enable a wide range of data operations in recent decades.

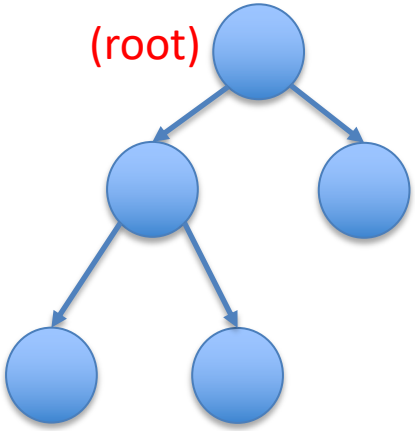
Representative data structures (recap)

String (a.k.a text)



(node) (edge)

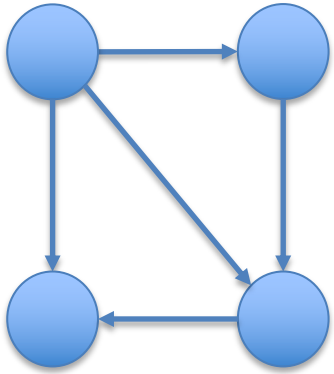
Tree (e.g., XML, glycan)



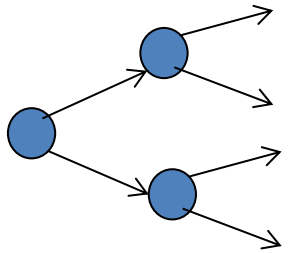
(root)

(leaf)

Graph (e.g., social network, chemical compound)



Succinct Data Structures (SDS) for Efficient Storage



- Standard Implementations:

Trees and graphs typically use pointers, taking O(n log n) space, where 'n' is the number of nodes.

For billions of nodes, the log n factor (20-30) adds substantial size.

- SDS Approach:

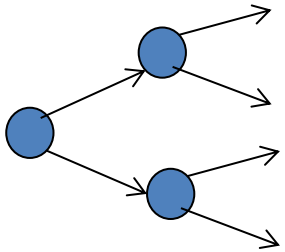
SDS compactly represents a tree with a bit array, using just O(n) space.

Most operations with SDS are quick, performed in O(1) time.

- Background:
  - Originated from Jacobson's 1989 PhD thesis at CMU.
  - Built on early works by Elias (1975), Tarjan/Yao (1979), Chazelle (1985), and others.
  - Evolved with RRR in 2002 and further with Sadakane/Navarro in 2010.
- Key Benefit:

Allows big data to be indexed in memory, enhancing speed and efficiency.

Succinct Data Structure (SDS)



- Many different types of SDS have been proposed
  - For every traditional data structure, there is a corresponding succinct version.

Data structure	SDS
Hash	Bloom Filter
Integer array	Wavelet tree
Tree	LOUDS
Graph	K^2-Tree
Suffix tree	FM-index, CSA

Today's Lecture: Exploring Succinct Data Structures

- Agenda:

1. Rank/Select Dictionary
2. Succinct Tree
3. Wavelet Tree
4. Optional: Application of Wavelet Tree to Graph Similarity Search

Today's Lecture: Exploring Succinct Data Structures

- Agenda:

1. Rank/Select Dictionary
2. Succinct Tree
3. Wavelet Tree
4. Optional: Application of Wavelet Tree to Graph Similarity Search

Rank/select (RS) dictionary : A SDS for binary vectors

- Fundamental data structure for implementing other SDSs
- It supports the rank and select operations on binary vector B
  - $\mathrm{Rank}_c(B,i)$: return the count of $c \in \{0,1\}$ in B[0…i]
  - $\mathrm{Select}_c(B,i)$: return the position of i-th occurrence of $c \in \{0,1\}$ in B

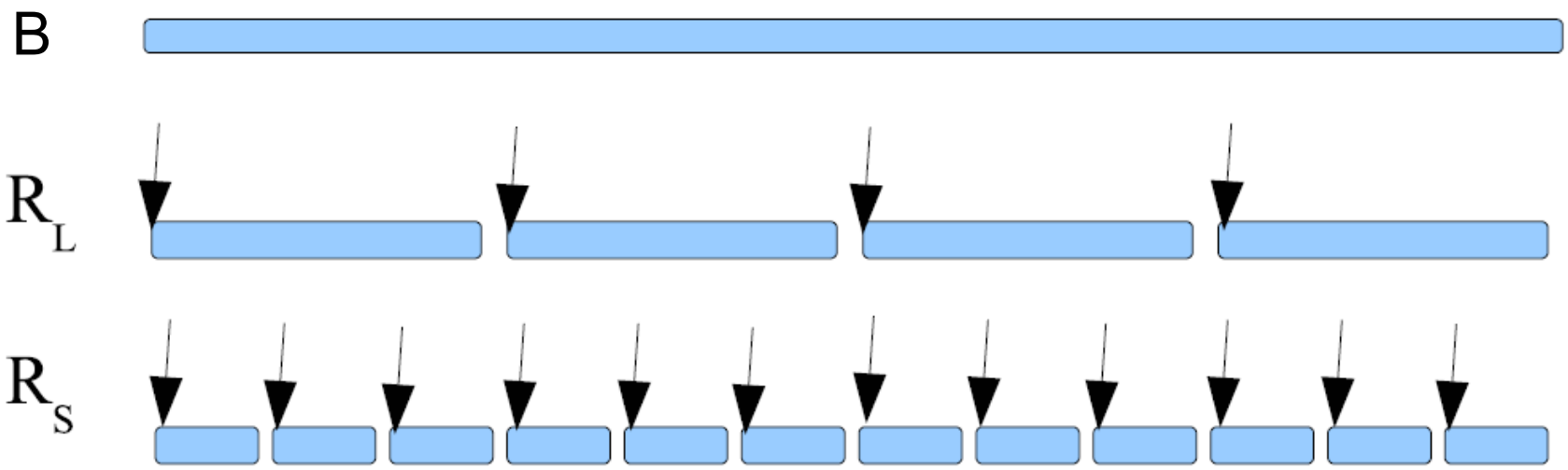
Ex: B=011010100

	i	0	1	2	3	4	5	6	7	8
$\mathrm{Rank}_1(B,5)=3$	B	0	1	1	0	1	0	1	0	0
$\mathrm{Select}_1(B,3)=4$	B	0	1	1	0	1	0	1	0	0

Rank/select (RS) dictionary : A SDS for bit vectors

- Three types of implementation according to density of bit vector B
  - Density = # of 1s

1. Dense bit vector (explained in the next slide)
2. Sparse bit vector [D.Okanohara and K.Sadakane'07]
   - Utilizes Elias-Fano encoding [Elias'74 and Fano'71]
3. Bit vector with any distribution
   - Introduced by RRR (Raman et al., 2002), suitable for bit vectors with any pattern of ones.



- Divide bit vector B into large blocks, each of length $\ell=\log^2 n$
  - R_L = Counts the number of 1s in each large block
- Divide each large block into blocks, each of length $s=(\log n)/2$
  - R_S = Counts the number of 1s in the small block relative to its large block

$$rank_1(B,i)=R_L[i/\ell]+R_s[i/s]+(\text{remaining ranks})$$

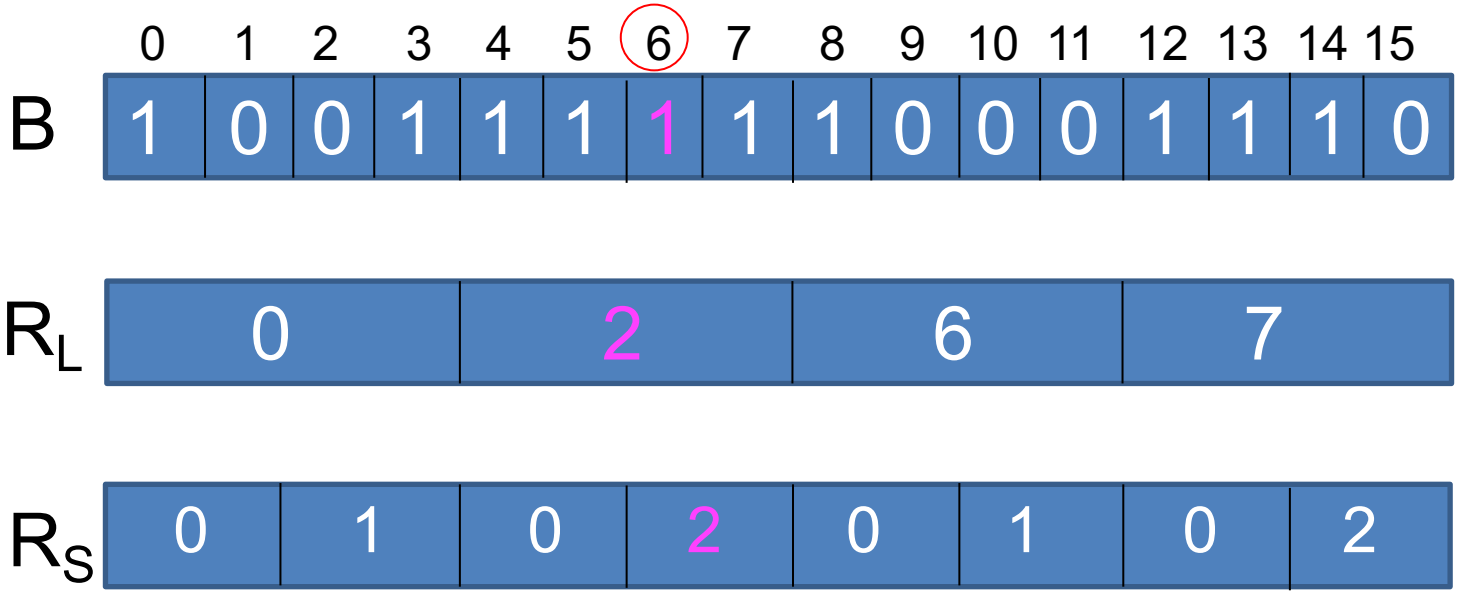
O(1) time using n + o(n) bits of space

Computation of remaining rank : Counting the number of 1s by Position

1. Bitwise Operator Implementation:
   - Complex to implement.
   - Considered the standard method
2. Table-based Counting:
   - Stores counts of 1s by position in a precomputed table.
3. CPU 'popcount' Instruction:
   - Direct CPU support for counting 1s.
   - Specific to CPU architecture.

```
int numofbits5(long bits)
{
    bits = (bits & 0x55555555) + (bits >> 1 & 0x55555555);
    bits = (bits & 0x33333333) + (bits >> 2 & 0x33333333);
    bits = (bits & 0x0f0f0f0f) + (bits >> 4 & 0x0f0f0f0f);
    bits = (bits & 0x00ff00ff) + (bits >> 8 & 0x00ff00ff);
    return (bits & 0x0000ffff) + (bits >>16 & 0x0000ffff);
}
```

	0	1	2
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1



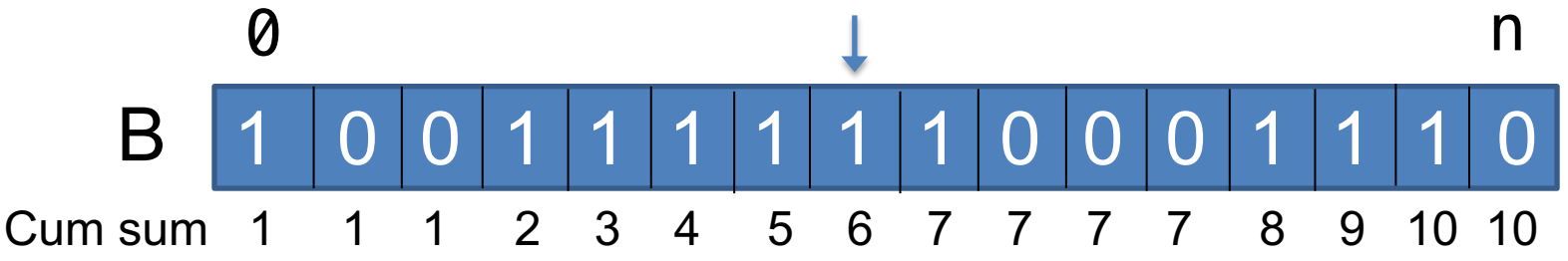
$$\text{Rank}_1(B, 6) = R_L[6/4] + R_S[6/2] + 1$$
$$= R_L[1] + R_S[3] + 1$$
$$= 2 + 2 + 1 = 5$$

Each rank operation can be computed in O(1) time (two accesses to R_L and R_S)

Understanding $Select_c(B,i)$

- Fast Algorithms: Proposals exist for O(1)-time algorithms, though they are complex.
- Key Property: The total count of 1s in B grows in a steady, one-directional pattern.
- Common Implementation: Practically, it's often implemented with a binary search that uses rank operations, taking O(log n) time.

$i < rank_c(B,n/2)?$



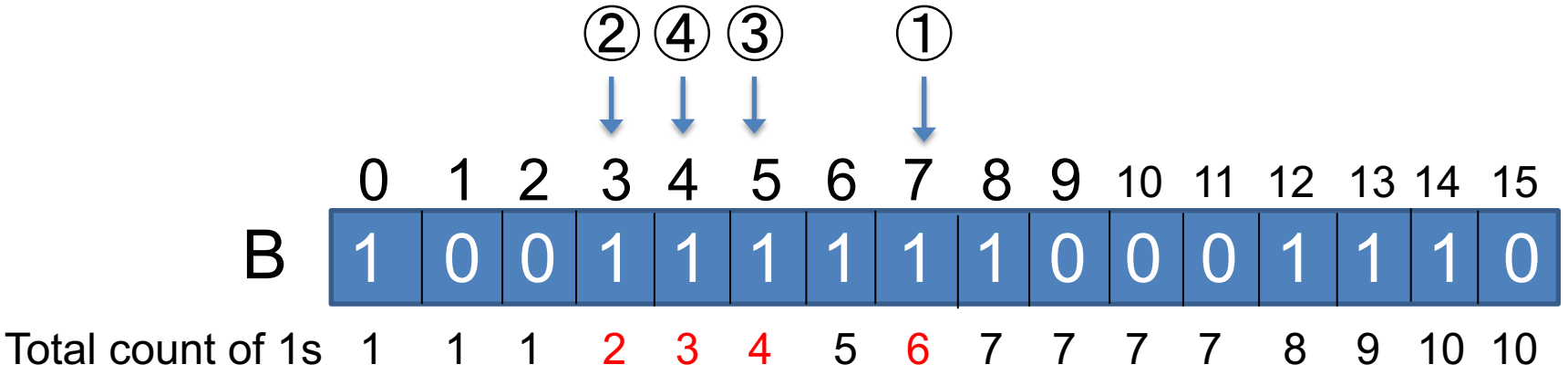
Steps to computation ‘$select_1(B, 3)$’ by binary search

① Check position 7: $rank_1(B,7)=6$
- It suggests the target is left of position 7.

② Check position 3: $rank_1(B,3)=2$
- It indicates the target is right of position 3.

③ Check position 5: $rank_1(B,5)=4$
- Again shifts the search left, to before position 5.

④ Check position 4: $rank_1(B,4)=3$
- It confirms that position 4 is the solution.



RS Dictionary Properties

- RS operations relationship:

  - $\text{rank}_c(B, \text{select}_c(B,i))$ equals i.

  - $\text{select}_c(B,\text{rank}_c(B,i))$ is less than or equal to i.

- Rank relationship:

  - $\text{rank}_1(B,i) + \text{rank}_0(B,i) = i$

  - $\text{rank}_0(B,i) = i - \text{rank}_1(B,i)$

Application1: fast set operations with RS dictionaries

- Set representation:
  - A set V within a universe U={1,2,…,n} is shown as bit vector B, where B[i]=1 if $i \in V$.
- Supported fast operations:
  - Count elements $\leqq i$ in V: Use $rank_1(B,i)$
  - Find i-th smallest element: Use $select_1(B, i)$
  - Max value $\leqq i$ in V: Apply $select_1(B, rank_1(B,i))$

- Example:
  - For V={1,3,5,6}, B=0101011
  - Elements $\leqq 4$ in V: $rank_1(B, 4) = 2$
  - Max value $\leqq 4$ in V: $select_1(B, rank_1(B,4))=3$

Application 2: Computing partial sums of integer arrays with RS dictionary

- Integer array representation:
  - Encode each element of integer array into unary code, which turns each integer t into a series of t zeros followed by 1.

  Ex: Unary codes encode t into 00…001
  t 0s
- Bit vector representation:
  - Join these unary codes end-to-end to form bit vector B
- Partial sum Calculations:
  - The partial sum par(S,i) for S[0,i] is calculated as $select_1$(B, i+1) – 1, representing the total count of zeros in B[0, i]
- Example:
  - Given S = 1, 3, 2, 3, 4, B = 0100010010001000 01
  - The partial sum up to position 2, par(S, 2) is $select_1$(B,3)-2=8-2=6

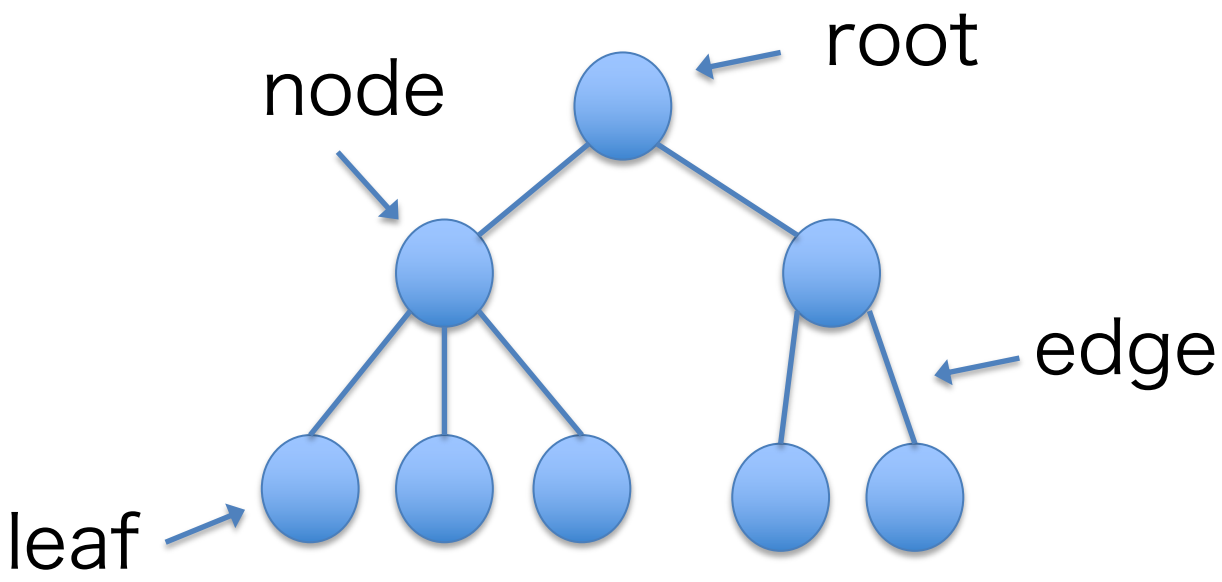
Today's Lecture: Exploring Succinct Data Structures

- Agenda:

1. Rank/Select Dictionary
2. Succinct Tree
3. Wavelet Tree
4. Optional: Application of Wavelet Tree to Graph Similarity Search

Ordered tree: An overview

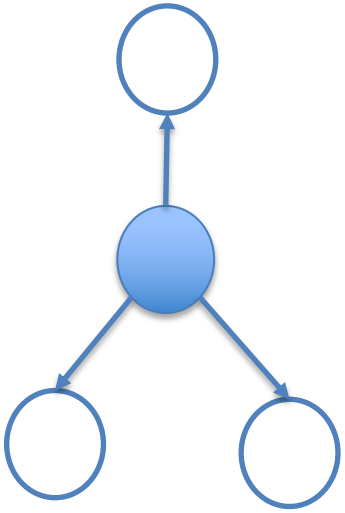
- Structure: Made up of nodes connected by edges.
- Hierarchy: Each node typically has one parent and possibly multiple children.
- Root node: The one node without a parent.
- Leaf nodes: Nodes that do not have any children.



Tree Implementation with Pointers

- Space requirement: Takes up O(n log n) bits, where n is the number of nodes.
- Example: A binary tree with 100 million nodes requires approximately 2.4GB of space, assuming 192 bits per node.

```
struct Node {
   Node *parent;
   Node *left;
   Node *right;
};
```



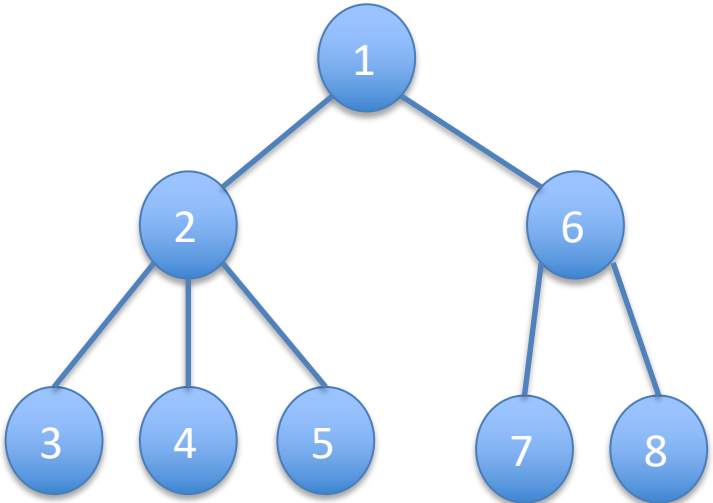
Succinct Ordered Tree

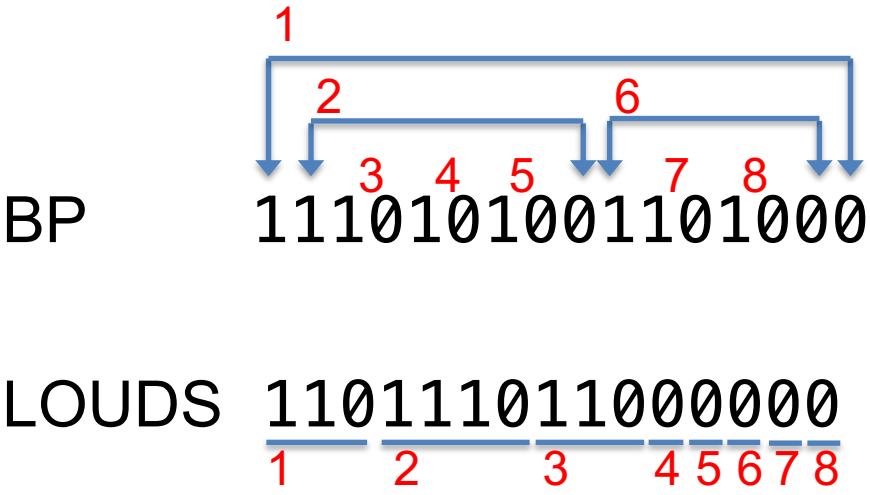
- Purpose: Efficiently compresses the representation of an ordered tree to save space.
- Representations: Three compression forms:

  (i) Balanced Parenthesis (BP)

  (ii) Depth First Unary Degree Sequence (DFUDS)

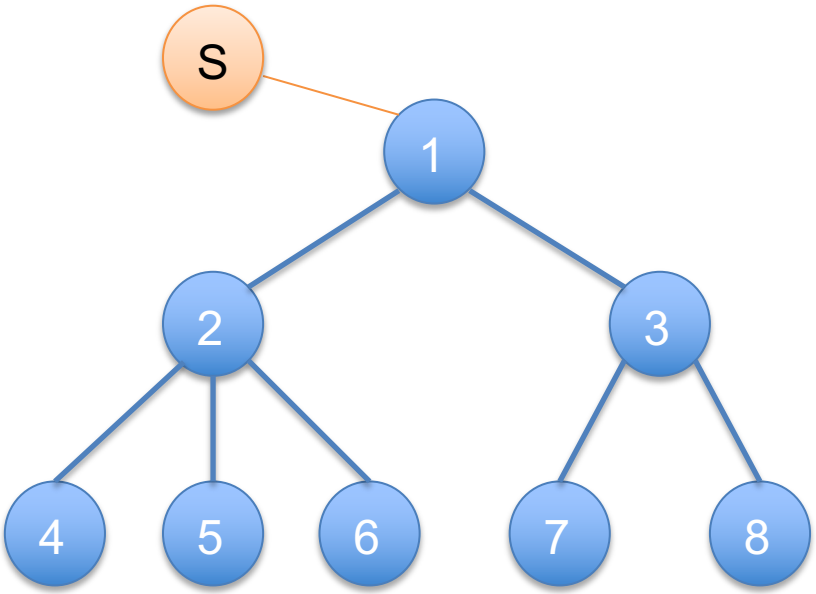
  (iii) Leveled Order Unary Degree Sequence (LOUDS)
- Requires just 2 bits per node, drastically reducing the space to 0.3MB for a tree with 100 million nodes, far less than the 2.4GB needed with standard pointers.





Level Ordered Unary Degree Sequences (LOUDS)

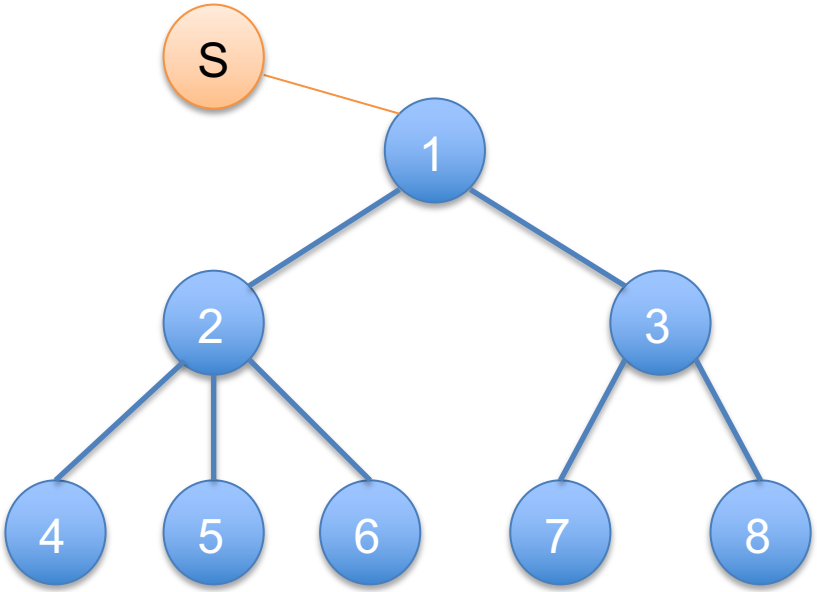
- Represent an ordered tree as bit vector B
- Place ’10’ at the beginning of B (as a dummy node S)
- Traverse the tree in a breadth-first manner, place k ‘1’s followed by ‘0’ for each k degree node on B
- 2n+1 bits for representing a tree with n nodes



S 1 2 3 4 5 6 7 8

B:1011011101100000

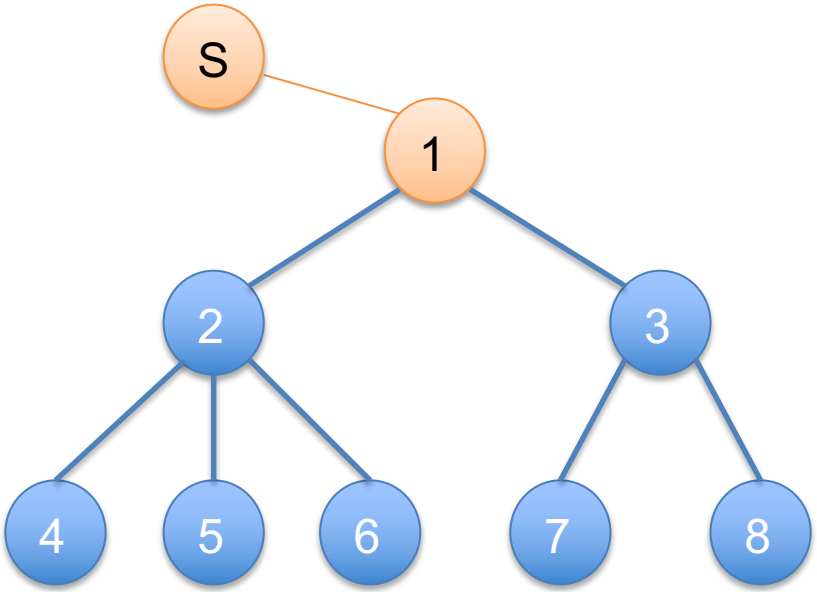
Level Ordered Unary Degree Sequences (LOUDS)



S

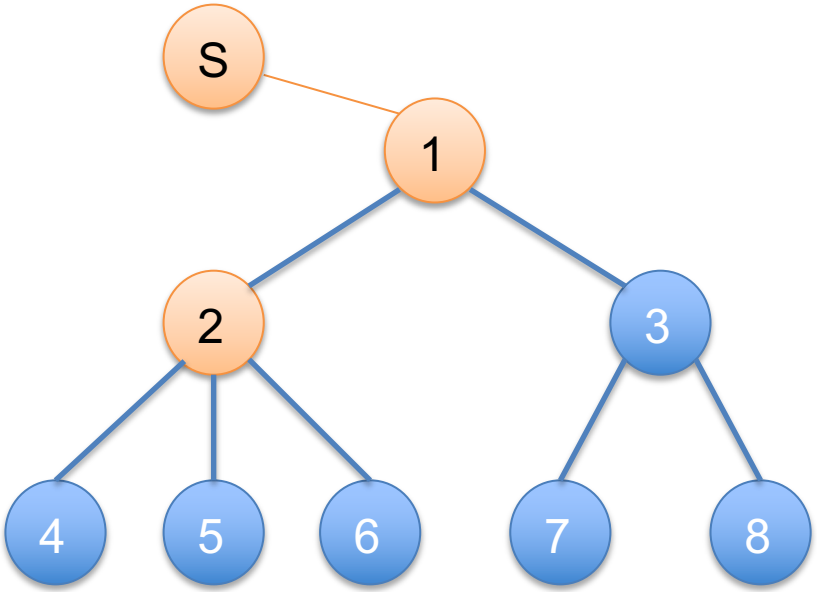
B:10

Level Ordered Unary Degree Sequences (LOUDS)



B: S 1
10110

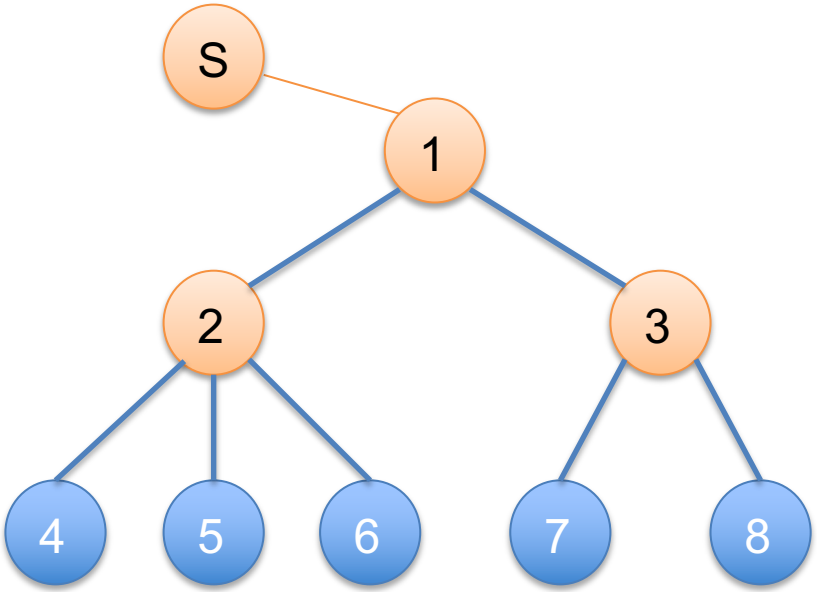
Level Ordered Unary Degree Sequences (LOUDS)



S 1 2

B:101101110

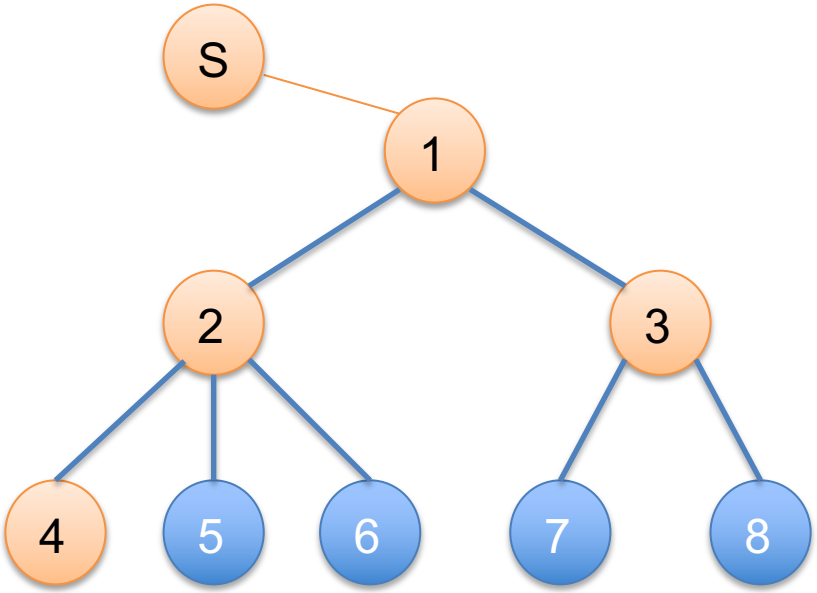
Level Ordered Unary Degree Sequences (LOUDS)



S 1 2 3

B:10110111110110

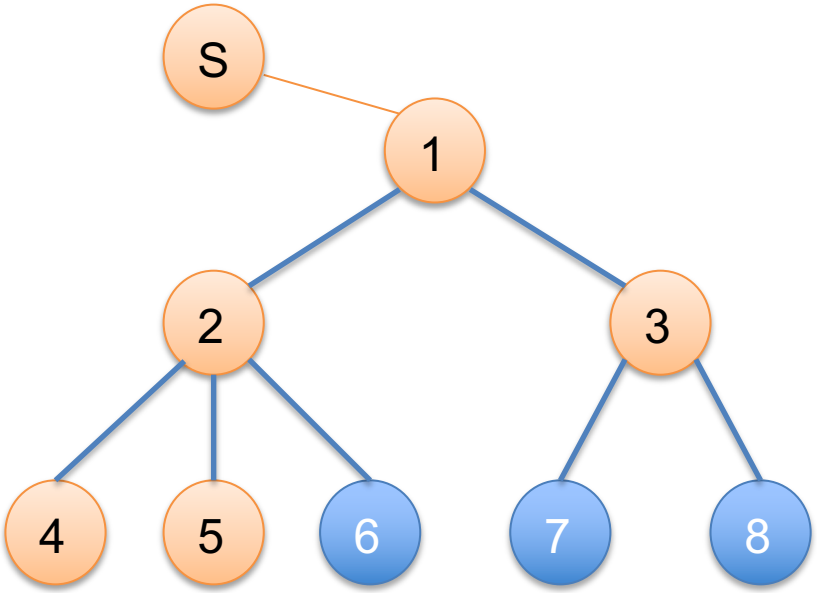
Level Ordered Unary Degree Sequences (LOUDS)



S 1 2 3 4

B:10110111011100

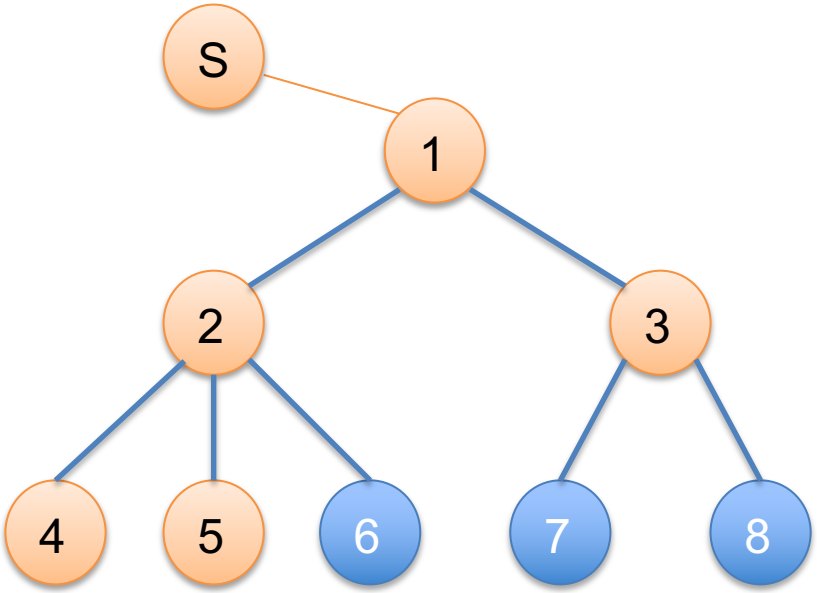
Level Ordered Unary Degree Sequences (LOUDS)



S 1 2 3 4 5

B:10110111011000

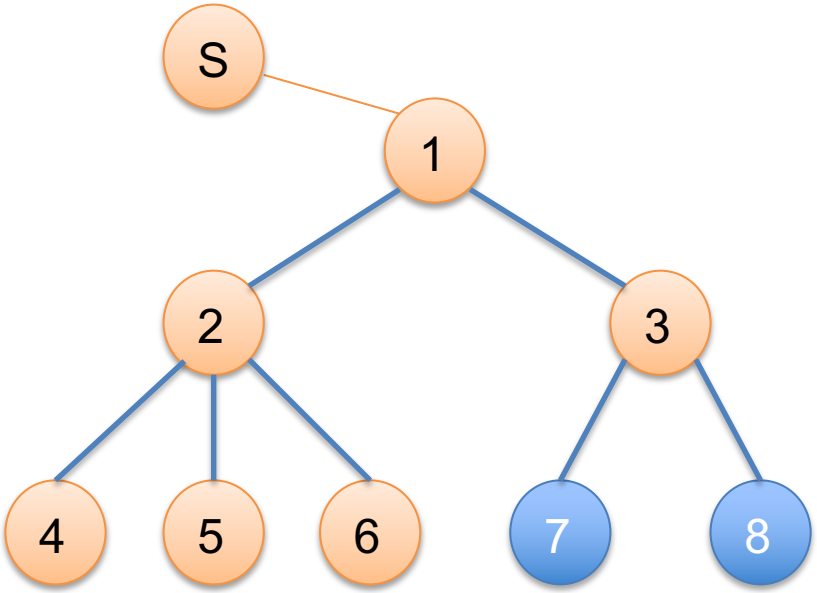
Level Ordered Unary Degree Sequences (LOUDS)



S 1 2 3 4 5

B:10110111011000

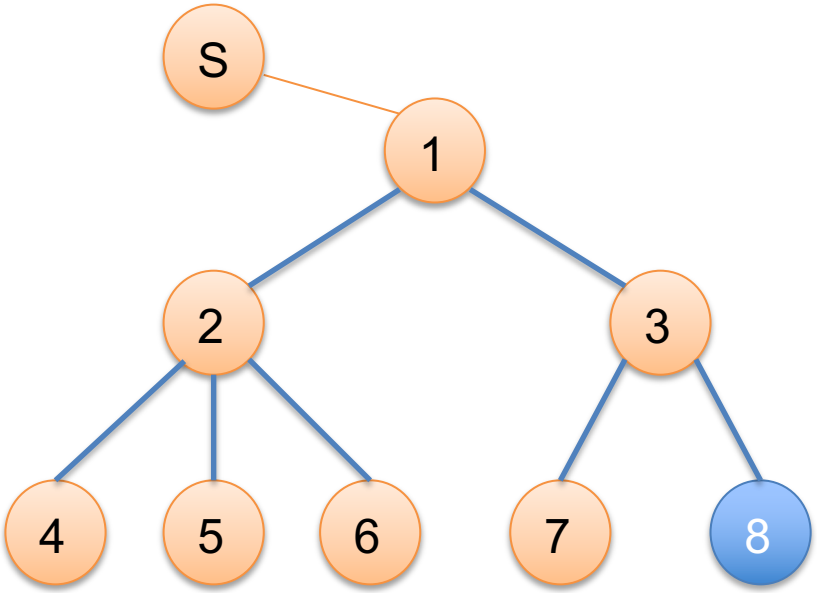
Level Ordered Unary Degree Sequences (LOUDS)



S 1 2 3 4 5 6

B:10110111011000

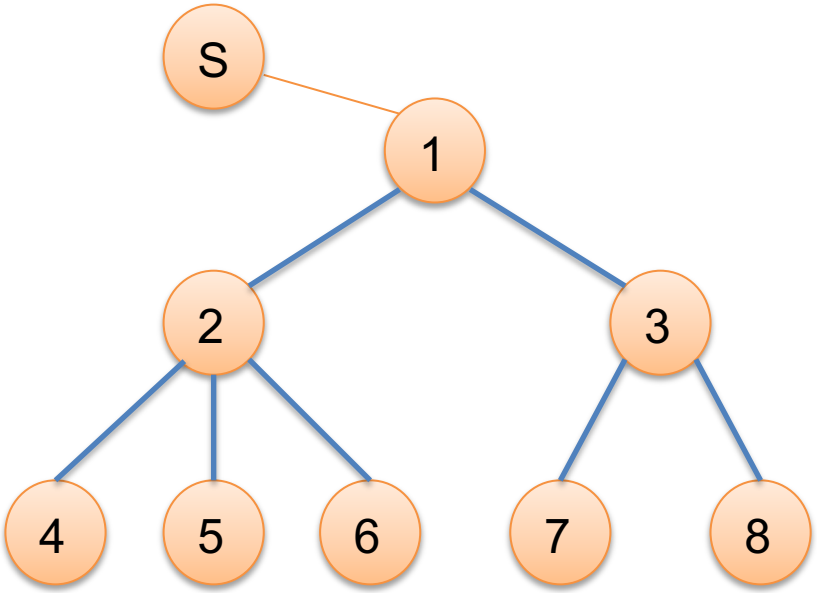
Level Ordered Unary Degree Sequences (LOUDS)



S 1 2 3 4 5 6 7

B:10110111011000000

Level Ordered Unary Degree Sequences (LOUDS)

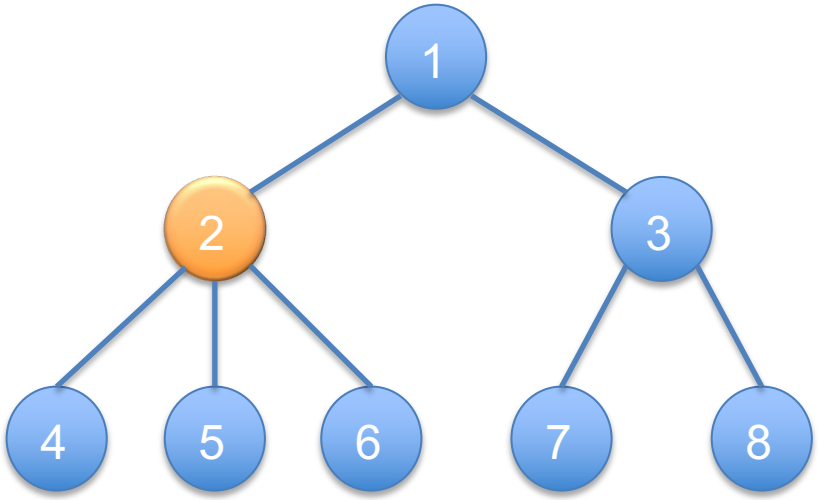


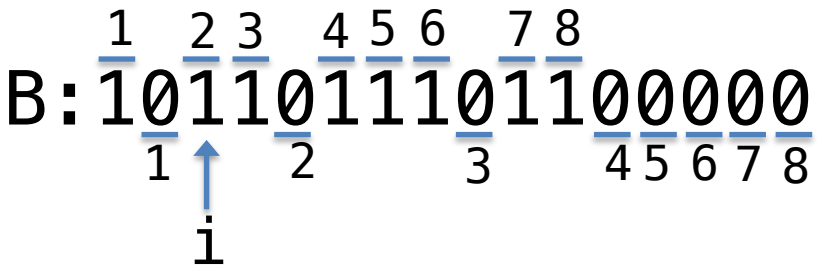
S 1 2 3 4 5 6 7 8

B:10110111011000000

Operations with LOUDS

- Property: Each node corresponds to 1 and 0 on B, respectively
- Allows determination of the first child, last child, next child, and the parent of a given node i.



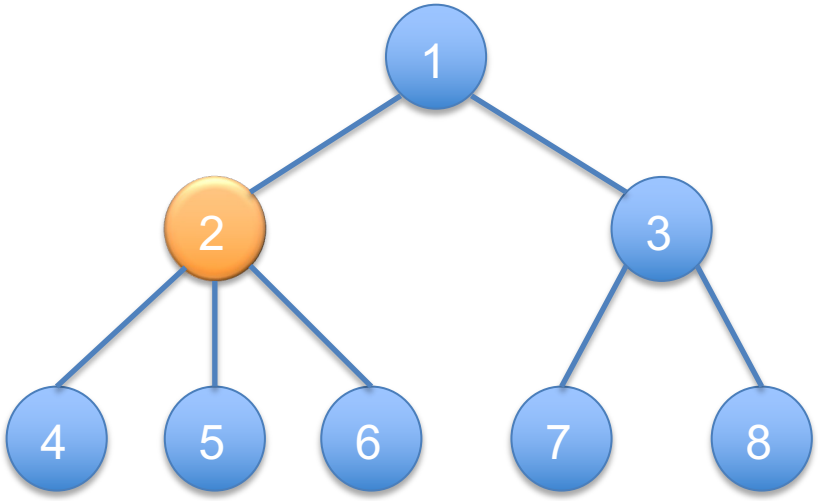


Computing the first child of a node

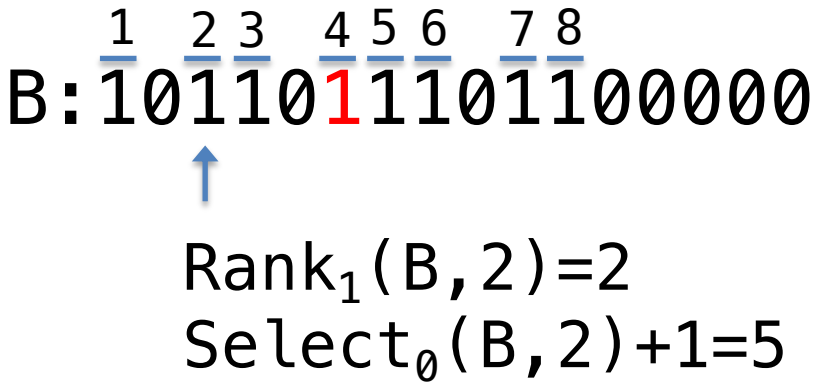
- The first child for i on B can be computed as

  $select_0(B, Rank_1(B, i))+1$

- Step 1: Identify the node. Use $rank_1(B, i)$ to find the position of node i in B, let's call this k.
- Step 2: Locate the first child. Then, $select_0(B, k)$ gives the position of the zero that marks the end of the k-th node's children. Add 1 to get the position of the first child.

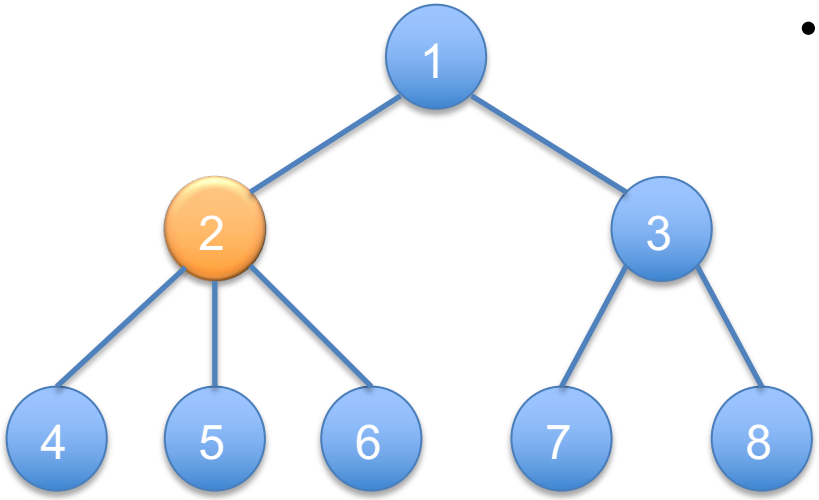


- Compute the first child 4 of node 2



Computing the last child of a node

- Last child for position i on B can be computed as
  $\texttt{select}_0(B, \texttt{rank}_1(B, i)+1)-1$
- Step 1: Find the node's position with $\text{rank}_1(B, i)$; let's call this number k.
- Step 2: To get the position of the last child of node i; use $\text{select}_0(B, k+1)$ and subtract 1.



- Compute the last child 6 of node 2

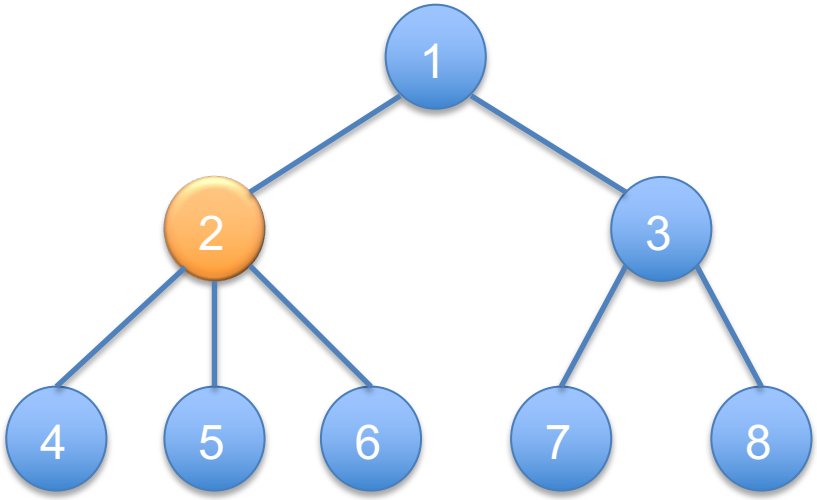
```
  1  2 3  4 5 6  7 8
B:10110111011100000
     ↑
     i
```

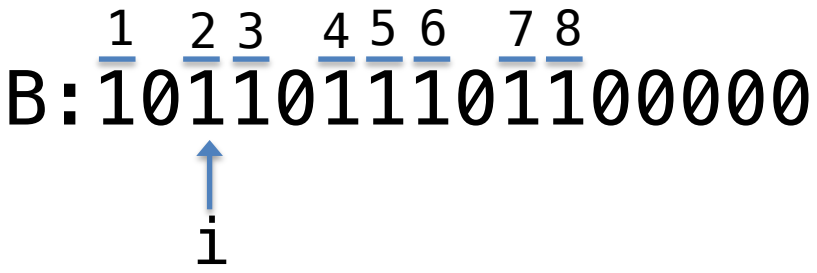
$\text{Rank}_1(B,2)=2$

$\text{Select}_0(B,2+1)-1=7$

Computing the next sibling node of a node

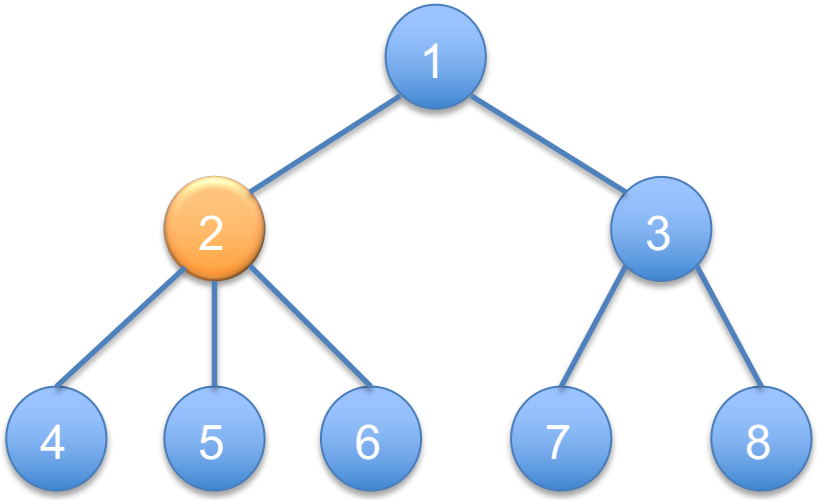
- Sibling nodes in LOUNDS: Sibling nodes are represented by consecutive ‘1’s in B.
- Finding the next sibling: the next sibling for a node at position is simply at position i + 1





Computing the parent node for a node

- Parent node for position i on B can be computed as $select_1(B, rank_0(B,i))$
- Step 1: Locate zero: Use $rank_0(B, i)$ to calculate the position of the '0' corresponding to node i, denoted as k.
- Step 2: Identify parent: The parent's position is then found by $select_1(B, k)$, which locates the '1' that corresponds to the parent of node i.



- Compute parent 1 for node 2

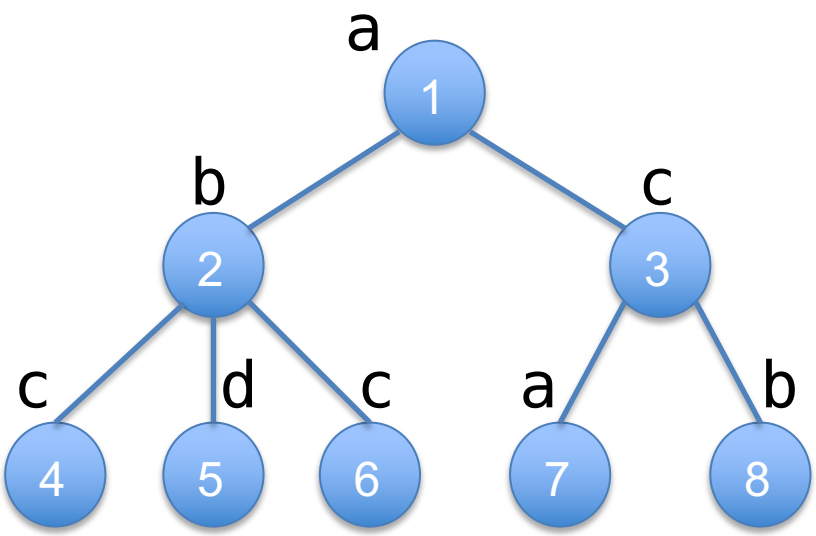
```
   1  2 3  4 5 6  7 8
B: 1 0 1 1 0 1 1 1 0 1 1 0 0 0 0 0
   1   ↑   2       3       4 5 6 7 8
       i
```

$Rank_0(B,2)=1$

$Select_1(B,1)=0$

Node labels on LOUDS

- LOUDS represents the tree structure, not any associated data.
- Create an array A where each entory corresponds to a node label.
- Assign the label for node i to the i-th element in array A.
- Retrieve the label of node i using A[$rank_1$(B, i) - 1].



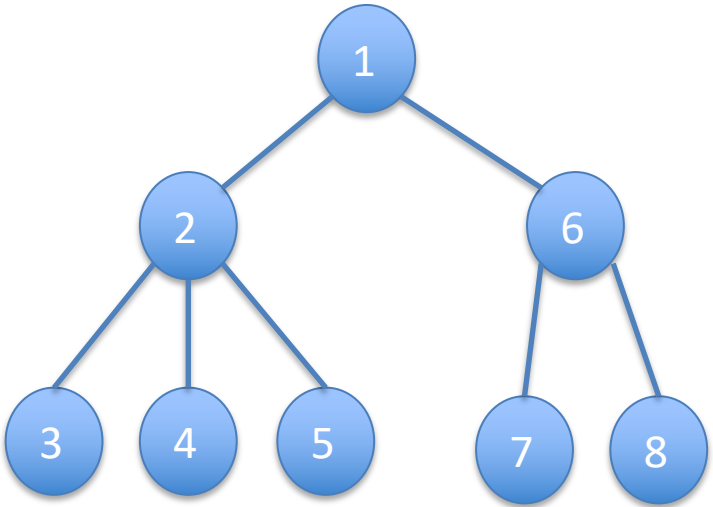
1 2 3 4 5 6 7 8

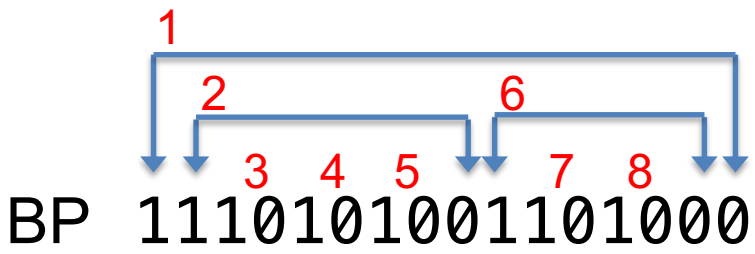
B:10110111011000000

A:abccdcab

Fully functional succinct tree
[G.Navarro and K.Sadakane, 2014]

- Supports the complete set of operations for tree data structures.
- Theoretical operation time is O(1), while practical performance is typically O(loglog n).
- Uses only 2n + o(n) bits, optimizing storage requirements.
- Built on Balanced Parenthesis (BP) representation.





Operations supported by fully-functional succinct tree

operation	description	time complexity
$inspect(i)$	$P[i]$	$\mathcal{O}(\log n/\log\log n)$
$findclose(i)/findopen(i)$	position of parenthesis matching $P[i]$	$\mathcal{O}(\log n/\log\log n)$
$enclose(i)$	position of tightest open parent. enclosing node i	$\mathcal{O}(\log n/\log\log n)$
$rank_(i)/rank_)(i)$	number of open/close parentheses in $P[1,i]$	$\mathcal{O}(\log n/\log\log n)$
$select_(i)/select_)(i)$	position of i-th open/close parenthesis	$\mathcal{O}(\log n/\log\log n)$
$rmqi(i,j)/RMQi(i,j)$	position of min/max excess value in range $[i,j]$	$\mathcal{O}(\log n/\log\log n)$
$pre\text{-}rank(i)/post\text{-}rank(i)$	preorder/postorder rank of node i	$\mathcal{O}(\log n/\log\log n)$
$pre\text{-}select(i)/post\text{-}select(i)$	the node with preorder/postorder i	$\mathcal{O}(\log n/\log\log n)$
$isleaf(i)$	whether $P[i]$ is a leaf	$\mathcal{O}(\log n/\log\log n)$
$isancestor(i,j)$	whether i is an ancestor of j	$\mathcal{O}(\log n/\log\log n)$
$depth(i)$	depth of node i	$\mathcal{O}(\log n/\log\log n)$
$parent(i)$	parent of node i	$\mathcal{O}(\log n/\log\log n)$
$first\text{-}child(i)/last\text{-}child(i)$	first/last child of node i	$\mathcal{O}(\log n/\log\log n)$
$next\text{-}sibling(i)/prev\text{-}sibling(i)$	next/previous sibling of node i	$\mathcal{O}(\log n/\log\log n)$
$subtree\text{-}size(i)$	number of nodes in the subtree of node i	$\mathcal{O}(\log n/\log\log n)$
$level\text{-}ancestor(i,d)$	ancestor j of i such that $depth(j) = depth(i) - d$	$\mathcal{O}(\log n)$ or $\mathcal{O}(d + \log n/\log\log n)$
$level\text{-}next(i)/level\text{-}prev(i)$	next/previous node of i in BFS order	$\mathcal{O}(\log n/\log\log n)$
$level\text{-}lmost(d)/level\text{-}rmost(d)$	leftmost/rightmost node with depth d	$\mathcal{O}(\log n)$ or $\mathcal{O}(d + \log n/\log\log n)$
$lca(i,j)$	the lowest common ancestor of two nodes i,j	$\mathcal{O}(\log n/\log\log n)$
$deepest\text{-}node(i)$	the (first) deepest node in the subtree of i	$\mathcal{O}(\log n/\log\log n)$
$degree(i)$	number of children of node i	$\mathcal{O}(\log n/\log\log n)$
$child(i,q)$	q-th child of node i	$\mathcal{O}(\log n/\log\log n)$
$child\text{-}rank(i)$	number of siblings to the left of node i	$\mathcal{O}(\log n/\log\log n)$
$in\text{-}rank(i)$	inorder of node i	$\mathcal{O}(\log n/\log\log n)$
$in\text{-}select(i)$	node with inorder i	$\mathcal{O}(\log n/\log\log n)$
$leaf\text{-}rank(i)$	number of leaves to the left of leaf i	$\mathcal{O}(\log n/\log\log n)$
$leaf\text{-}select(i)$	i-th leaf	$\mathcal{O}(\log n/\log\log n)$
$lmost\text{-}leaf(i)/rmost\text{-}leaf(i)$	leftmost/rightmost leaf of node i	$\mathcal{O}(\log n/\log\log n)$
$insert(i,j)$	insert node given by matching parent. at i and j	$\mathcal{O}(\log n/\log\log n)$
$delete(i)$	delete node i	$\mathcal{O}(\log n/\log\log n)$

Summary of succinct tree

- Introduces Level Ordered Unary Degree Sequence (LOUDS) as a method to represent trees efficiently.
- Details several operations possible with LOUDS, such as computing parent, child, and sibling nodes.

- Key Application: Trie
  - Purpose: A data structure designed for storing sets of words, commonly used in input method editors (IMEs).
  - Functionality: Efficiently retrieves and suggests words based on partial inputs.

- Potential in Data Mining:
  - Expanding Use: Succinct trees hold potential for various data mining applications, offering efficient storage and rapid operations on large data sets.

Today's Lecture: Exploring Succinct Data Structures

- Agenda:

1. Rank/Select Dictionary
2. Succinct Tree
3. Wavelet Tree
4. Optional: Application of Wavelet Tree to Graph Similarity Search

Wavelet tree overview



- A succinct data structure designed for efficiently handling integer arrays.
- Origins: Introduced by R. Grossi, A. Gupta, and J.S. Vitter at SODA 2003, initially it did not gain widespread attention.
- Rising Importance:
  - Revival: Gained recognition as a versatile tool for data operations, as highlighted by T. Gagie, G. Navarro, and S.J. Puglisi in 2012.
  - Functionalities: Supports a variety of operations on integer arrays, making it suitable for diverse applications.
- Applications:
  - Diverse Uses: Employed in graph search algorithms, compressed indices for document storage, and sequence alignment in bioinformatics, among others.

Wavelet Tree Capabilities

- Fast operations on integer arrays:
  - Range queries: Quickly find maximum, minimum, and specific intermediate values within an interval.
  - Distribution: Efficiently compute the distribution of integers across any given interval.
  - Commonality: Compute common integers among multiple intervals.
  - Integrity: Retrieve all integers within a specific interval.
- Performance Efficiency:
  - Independent computation time: The time required to perform operations is consistent and does not depend on the array's length.

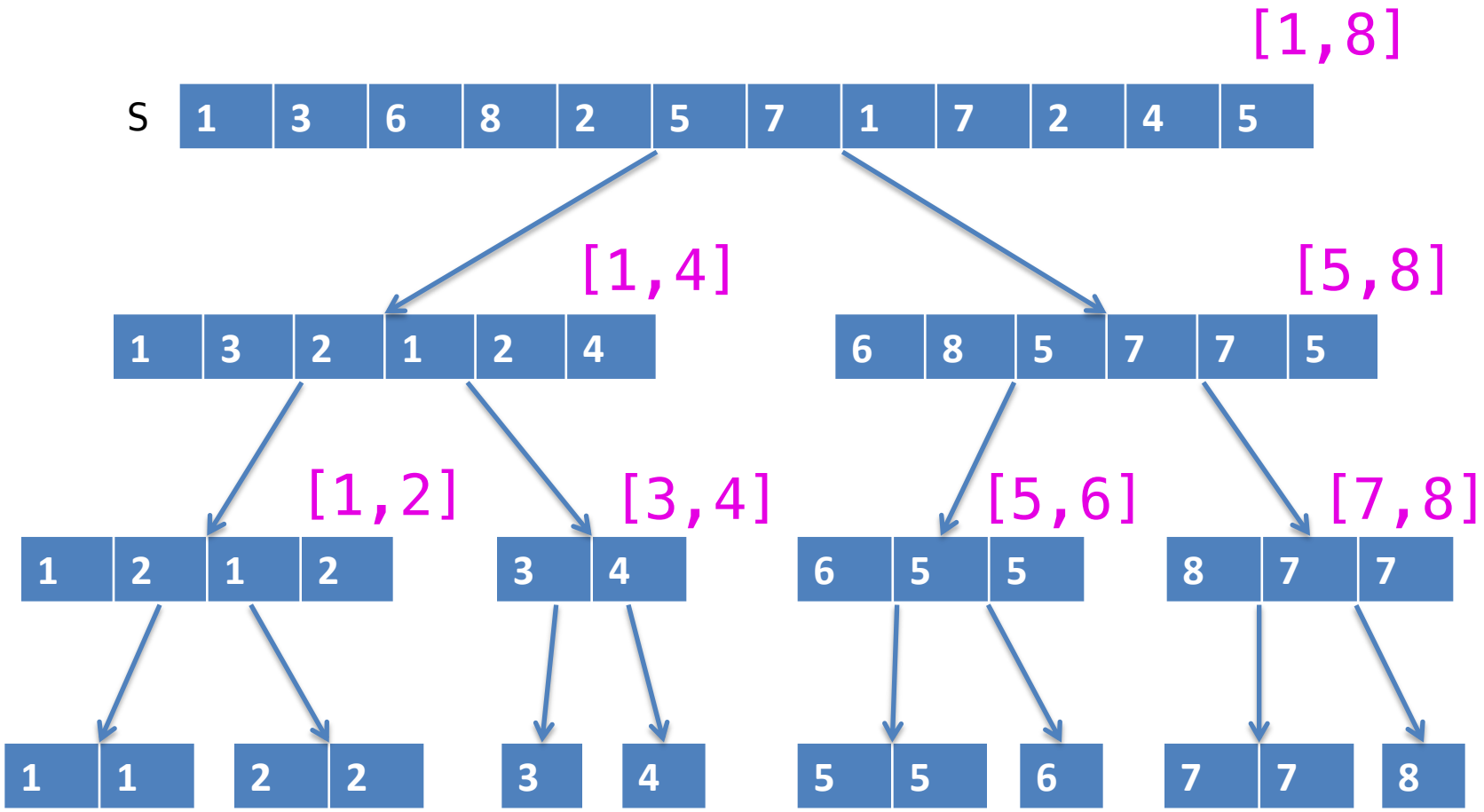
Advantage of operations on wavelet tree

Ex: Computing the maximum value within a speficic interval

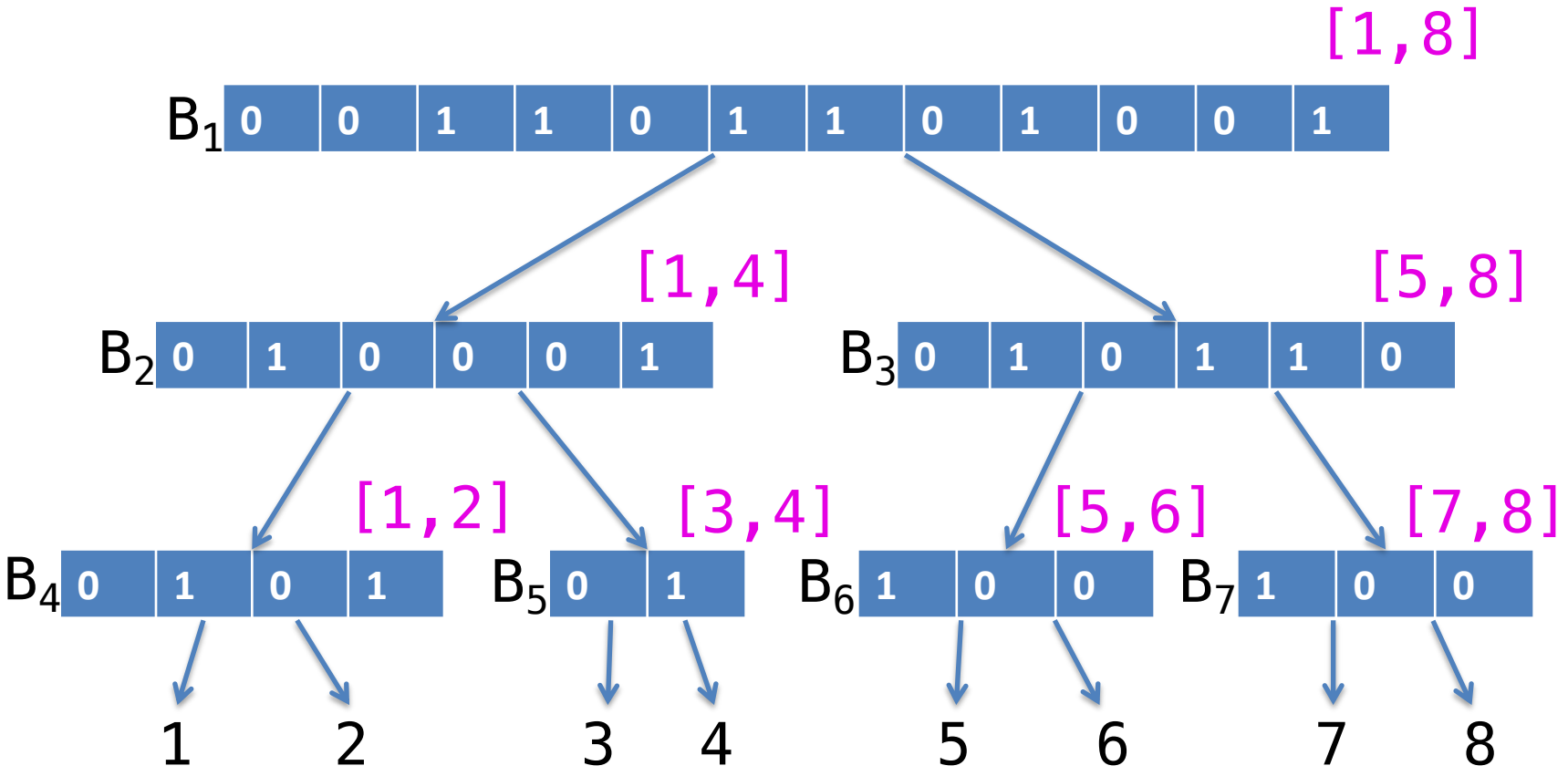
- Naïve method: Takes O(ℓ) time (ℓ : length of interval)
- Wavelet tree: Takes O(log m) time (m: maximum value on integer array)
  - It does not depend on the interval length.

- Definition of wavelet trees

Tree of subarrays:
Lower half = left, Higher half=right



Remember if each element is either in lower half (0) or higher half (1)



Rank/select (RS) dictionary : A SDS for binary vectors

- Fundamental data structure for implementing other SDSs
- It supports the rank and select operations on binary vector B
  - $\mathrm{Rank}_c(B,i)$: return the count of $c \in \{0,1\}$ in B[0…i]
  - $\mathrm{Select}_c(B,i)$: return the position of i-th occurrence of $c \in \{0,1\}$ in B

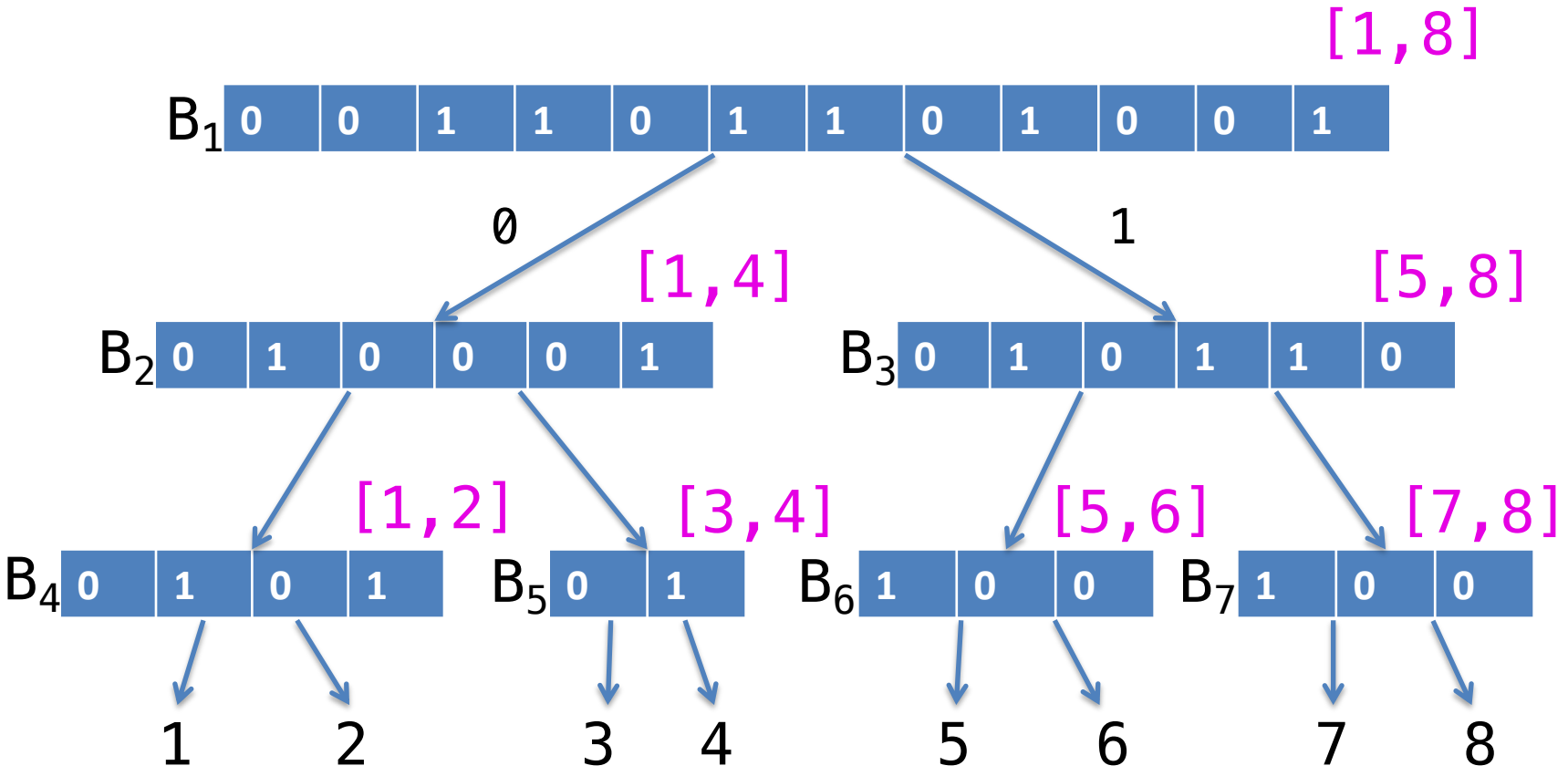
Ex: B=011010100

$\mathrm{Rank}_1(B,5)=3$

$\mathrm{Select}_1(B,3)=4$

i	0	1	2	3	4	5	6	7	8
B	0	1	1	0	1	0	1	0	0
B	0	1	1	0	1	0	1	0	0

Wavelet Tree = Collection of bit arrays indexed by rank/select dictionaries



Memory Usage

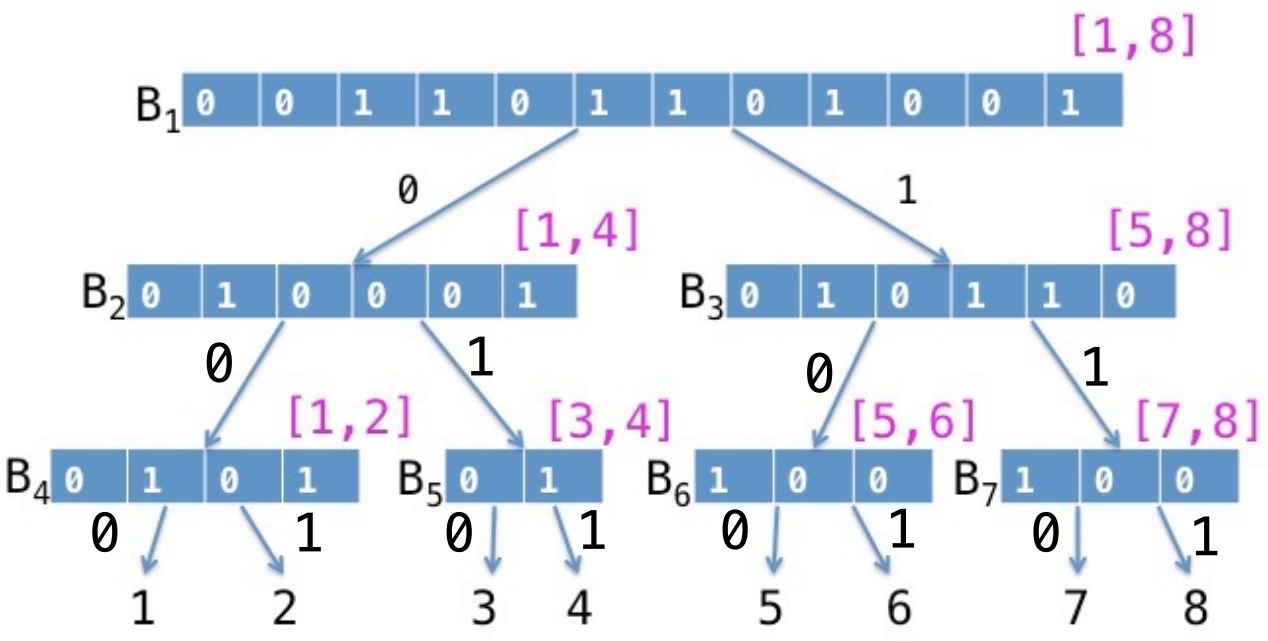
- $(1+\gamma)\ n \log m$ bits
  - n: Number of all integers in the database
  - m: Maximum integer
  - γ: Overhead for rank dictionary (around 0.6)
  - The memory required by a wavelet tree is compatible to simply storing the integer array using n log m bits.
  - This shows that the wavelet tree manages to offer its advanced querying capabilities without significant additional memory costs compared to basic storage.

Operations on wavelet tree

Operation	Description
Access(i)	S[i]
$Rank_\sigma(S,i)$	Number of σs in S[1,i]
$Select_\sigma(S,i)$	Position of i-th occurrence of σ
MaxElem(S,sp,ep)	Maximum value in S[sp,ep]
MinElem(S,sp,ep)	Minimum value in S[sp,ep]
RangeQuantile(S,p,sp,ep)	p-th largest value in S[sp,ep]
EnumElem(S,sp,ep,l,h)	All the values with at least l and at most h in S[sp,ep]
MaxRange(S,sp,ep,l,h)	The most frequently appearing value between at least l and at most h in S[sp,ep]
RangeInt(S,sp1,ep1,sp2,ep2)	All the common value in S[sp1,ep1] and S[sp2,ep2]

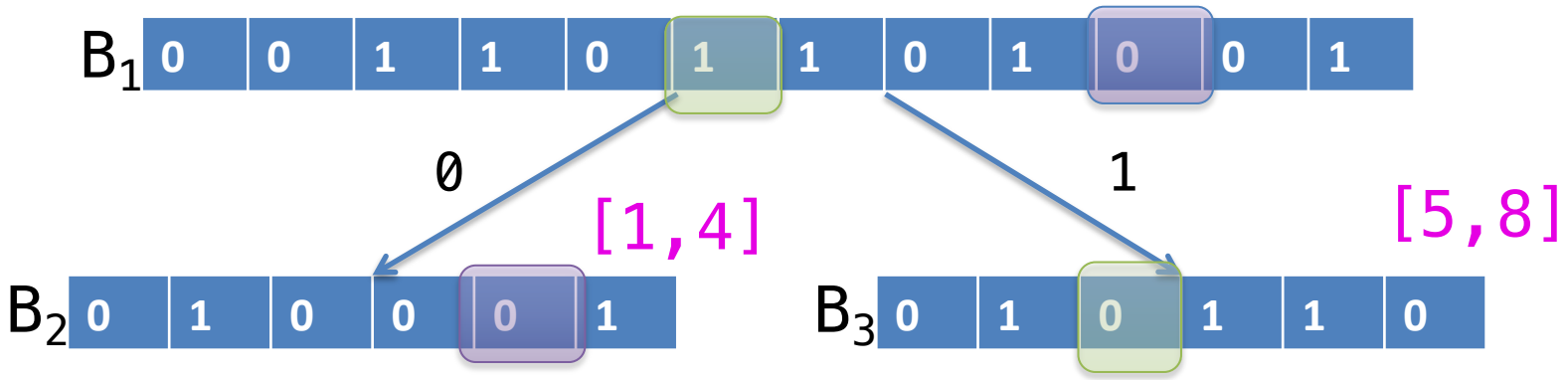
Property (i) of Wavelet Tree: Traversal and Value Recovery

- Implementation Mechanism:
  - Operations on the wavelet tree involve traversing from the root down to the leaves.
  - Each leaf in the wavelet tree represents a unique value from the integer array.
- Value Recovery:
  - The specific value represented by a leaf can be determined by the path taken from the root to that leaf. This path encodes the binary representation of the value, allowing it to be reconstructed accurately.



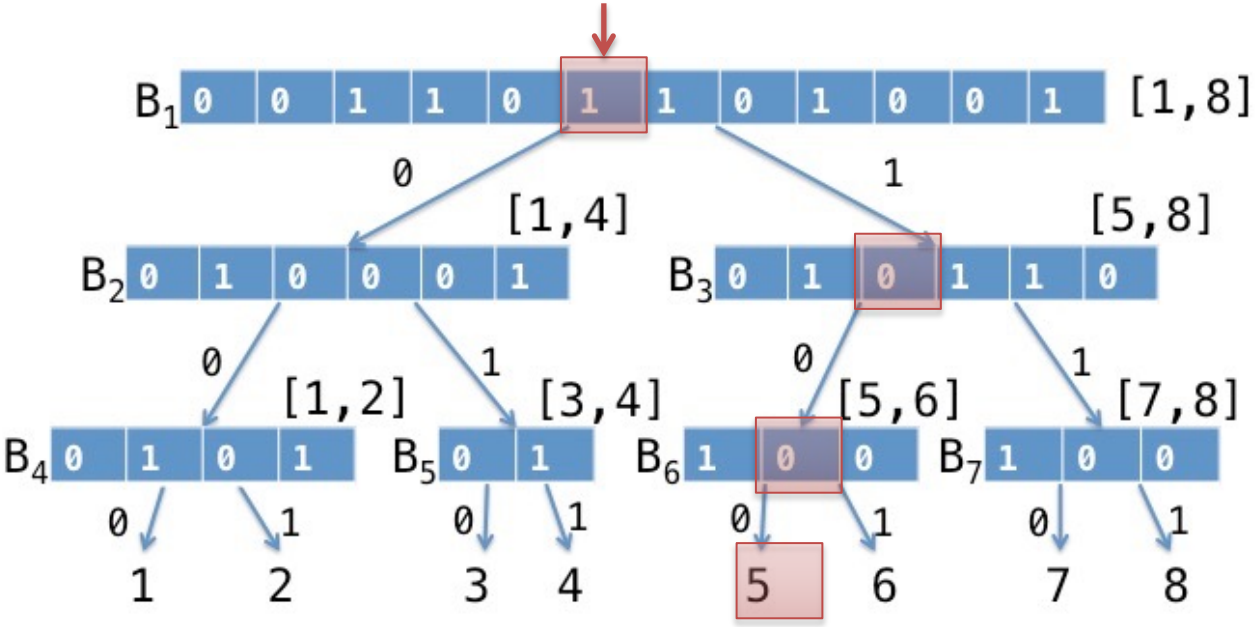
Property (ii) of Wavelet Tree: Fast Child Position Computation

- Using rank operation, a position of a child can be computed in O(1)-time
  - Left child: Determined by $rank_0(B, i)$
  - Right child: Determined by $rank_1(B, i)$
- Comparison to linear search: O(n) time



Access(i) : return S[i]

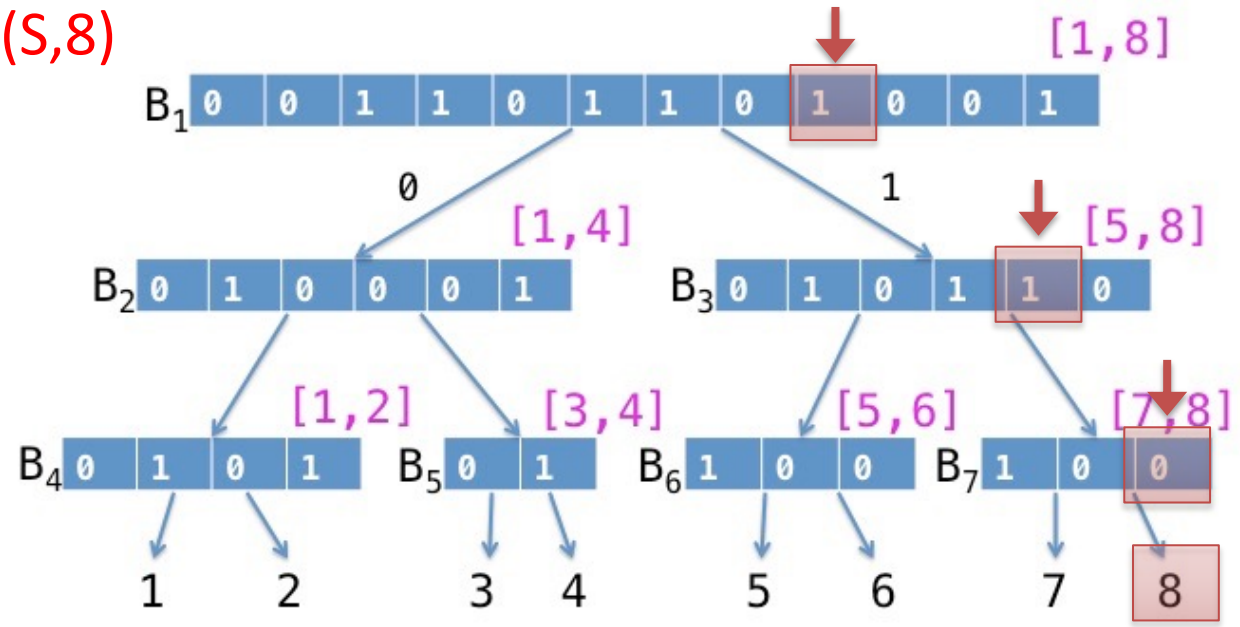
- Start at the root: Begin at the i-th position ($B_1[i]$) at the root
- Traverse using rank operations: Depending on whether the bit at the current position is a 0 or a 1:
  - Use $rank_0$ to follow the path to the left child for a 0 bit.
  - Use $rank_1$ to follow the path to the right child for a 1 bit.
- Recover Value: The sequence of bits obtained from the root to a leaf represents the binary form of S[i].



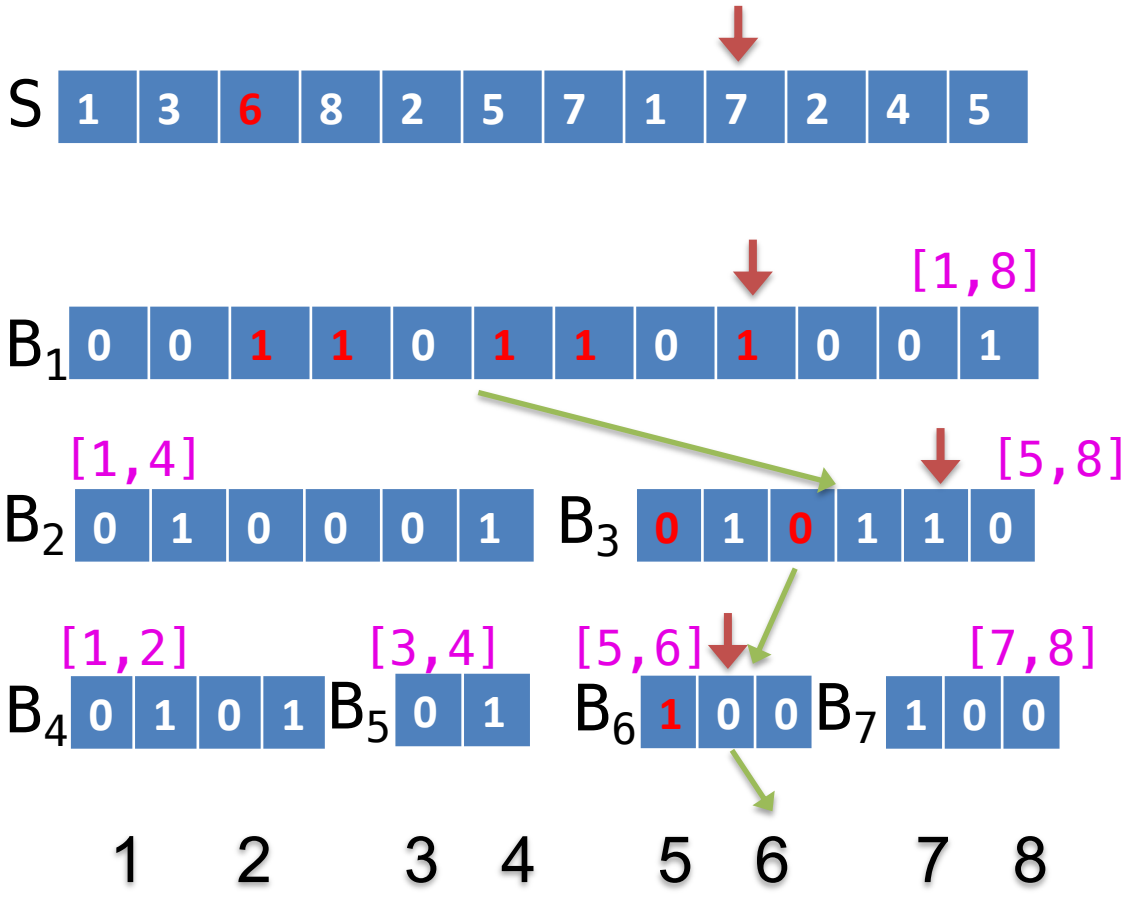
$Rank_\sigma(S,i)$: return the number of σs in S[1,i]

1. Trace symbol σ: Start at the root of the wavelet tree and trace the positions of σ downward to the corresponding leaf. This involves:
   – Following paths marked by bits that represent σ, using $rank_0$ or $rank_1$ operations as appropriate for each bit.
2. Calculate occurrence: Upon reaching the leaf, count the number of 0s or 1s, which corresponds to the number of time σ appears in S[1,i].

Ex: $rank_7(S,8)$



$Rank_6(S,9)$



1. 6 belongs to the higher half in interval [1,8] in B_1

   $Rank_1(B_1,9)=5$

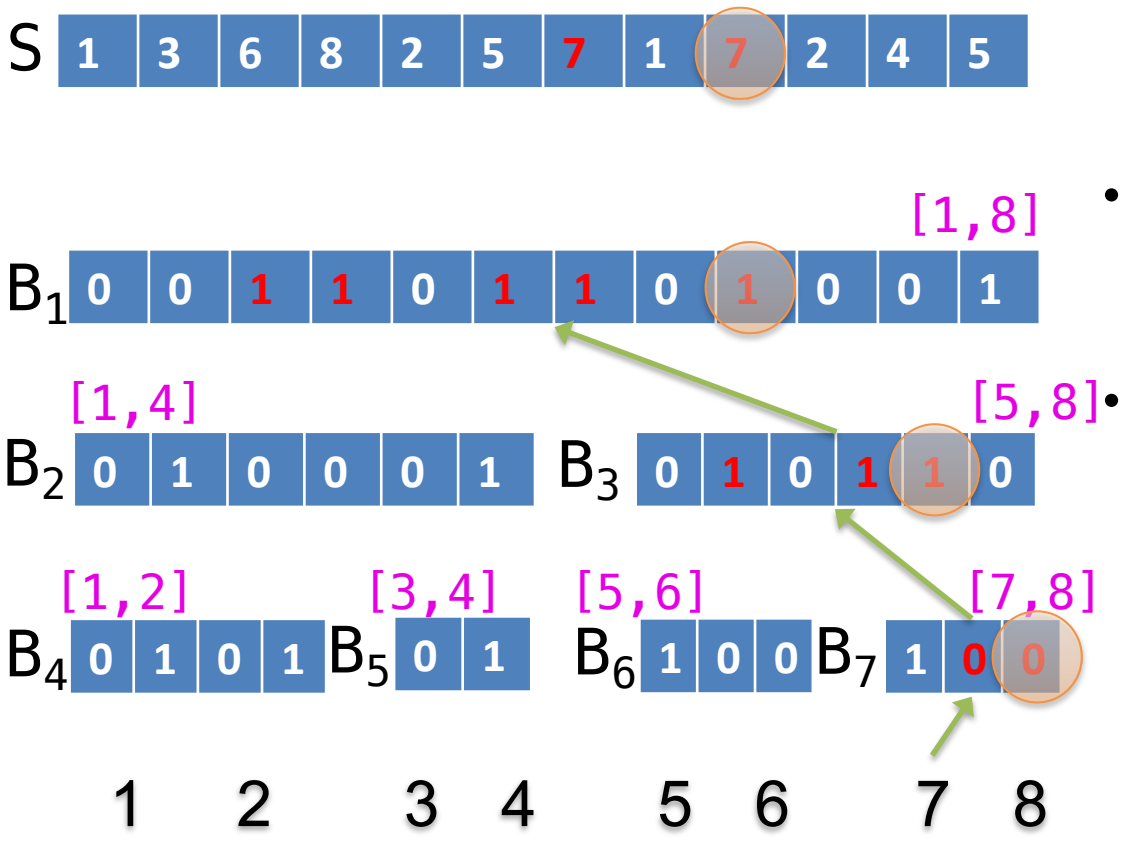
2. 6 belongs to the lower half in interval [5,8] in B_3

   $Rank_0(B_3,5)=2$

3. 6 belongs to the higher half in interval [7,8] in B_7

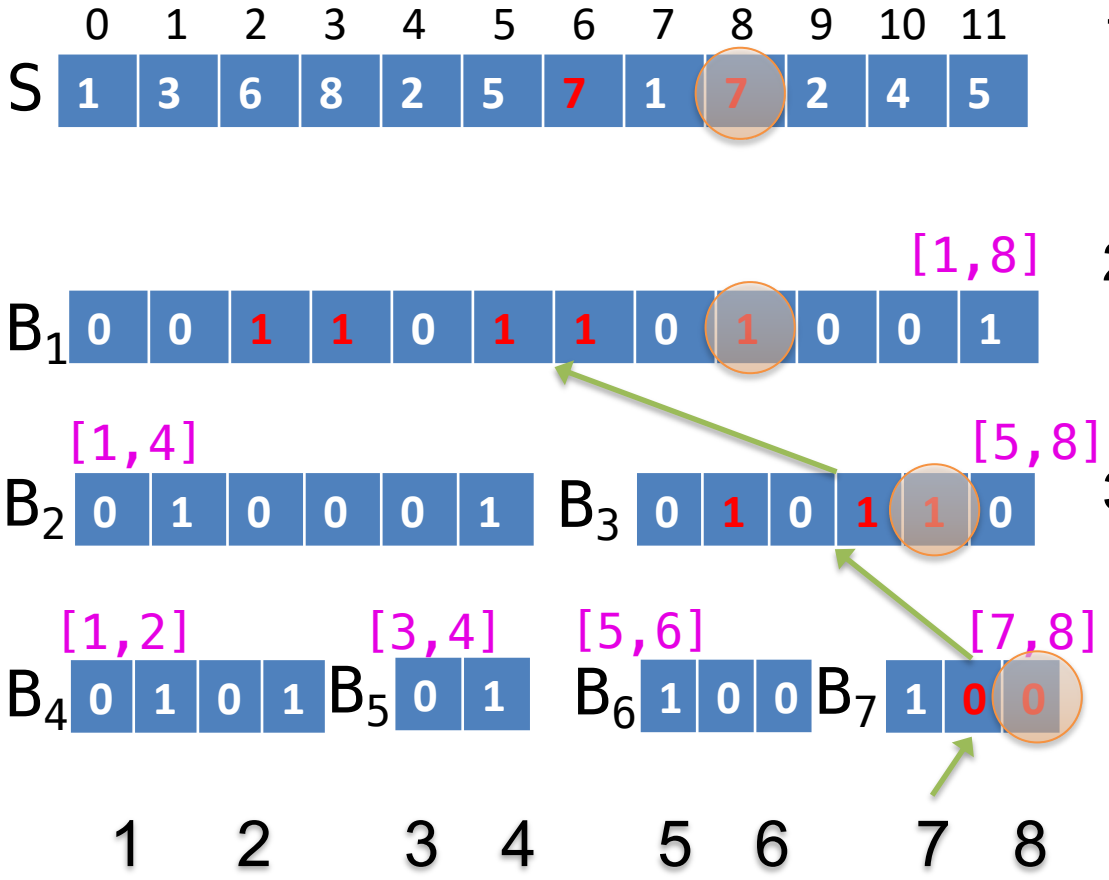
   $Rank_1(B_6,2)=1$ (solution)

$Select_\sigma(S,i)$: return the position of i-th occurrence of σ in B



- Bottom up traversal from the leaf corresponding to σ to the root
- Chase positions on each B_i corresponding to σ using select operations
- Compute $select_0$ for lower half and $select_1$ for higher half at each B_i

$select_7(S,2)$

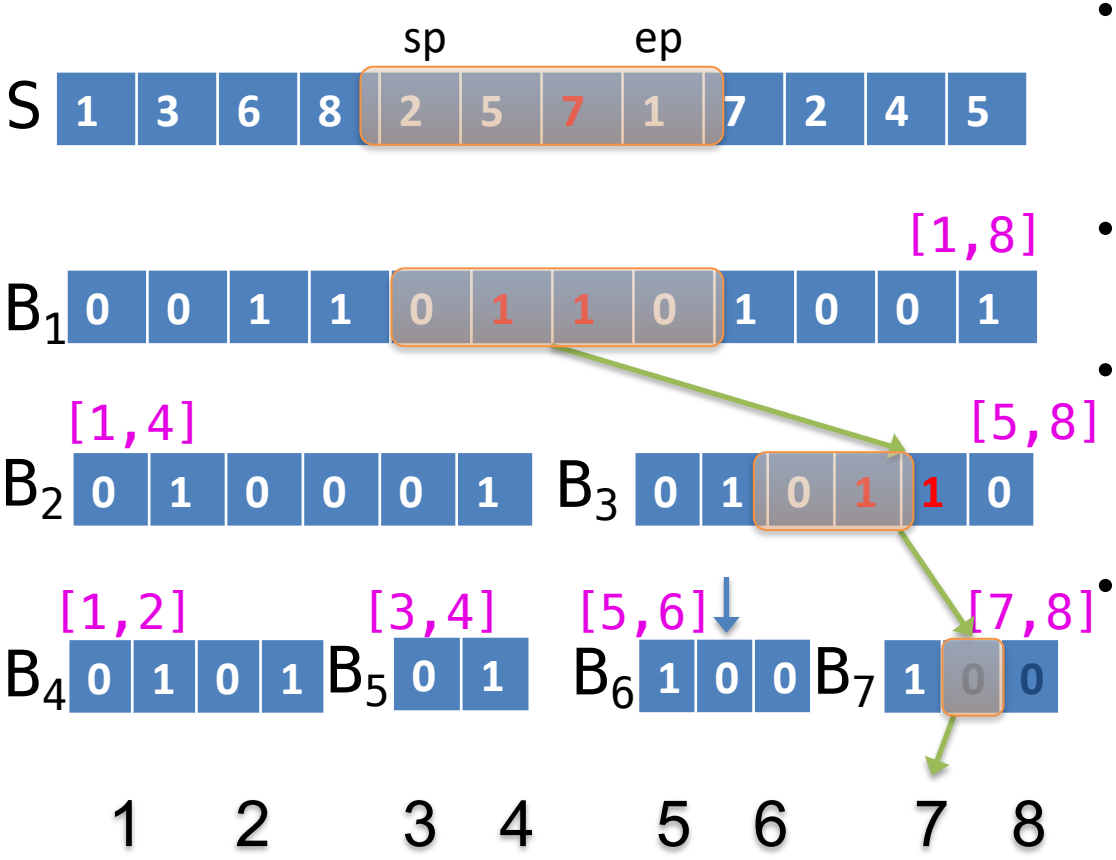


1. 7 belongs to lower half
   $select_0(B_7,2)=3$

2. 7 belongs to higher half
   $select_1(B_3,3)=5$

3. 7 belongs to higher half
   $select_1(B_1,5)=8$

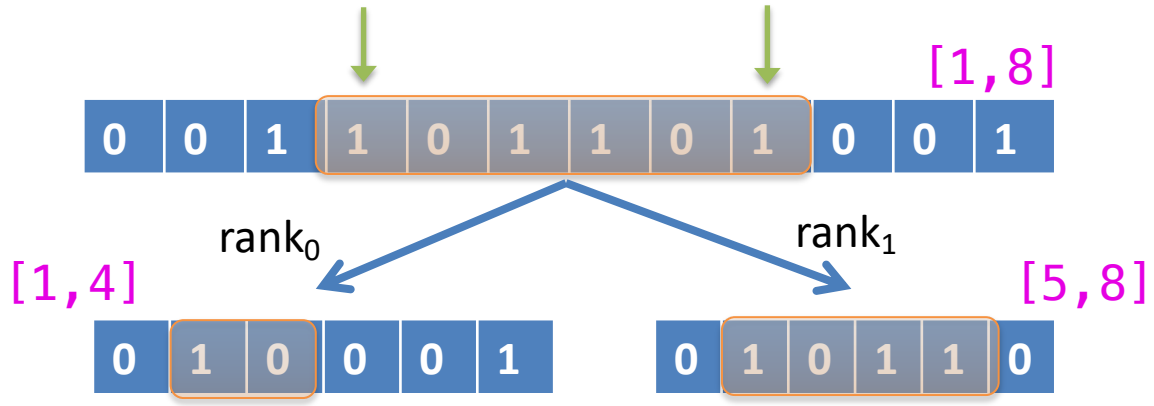
MaxElem(S,sp,ep) : returns the maximum value in S[sp,ep]



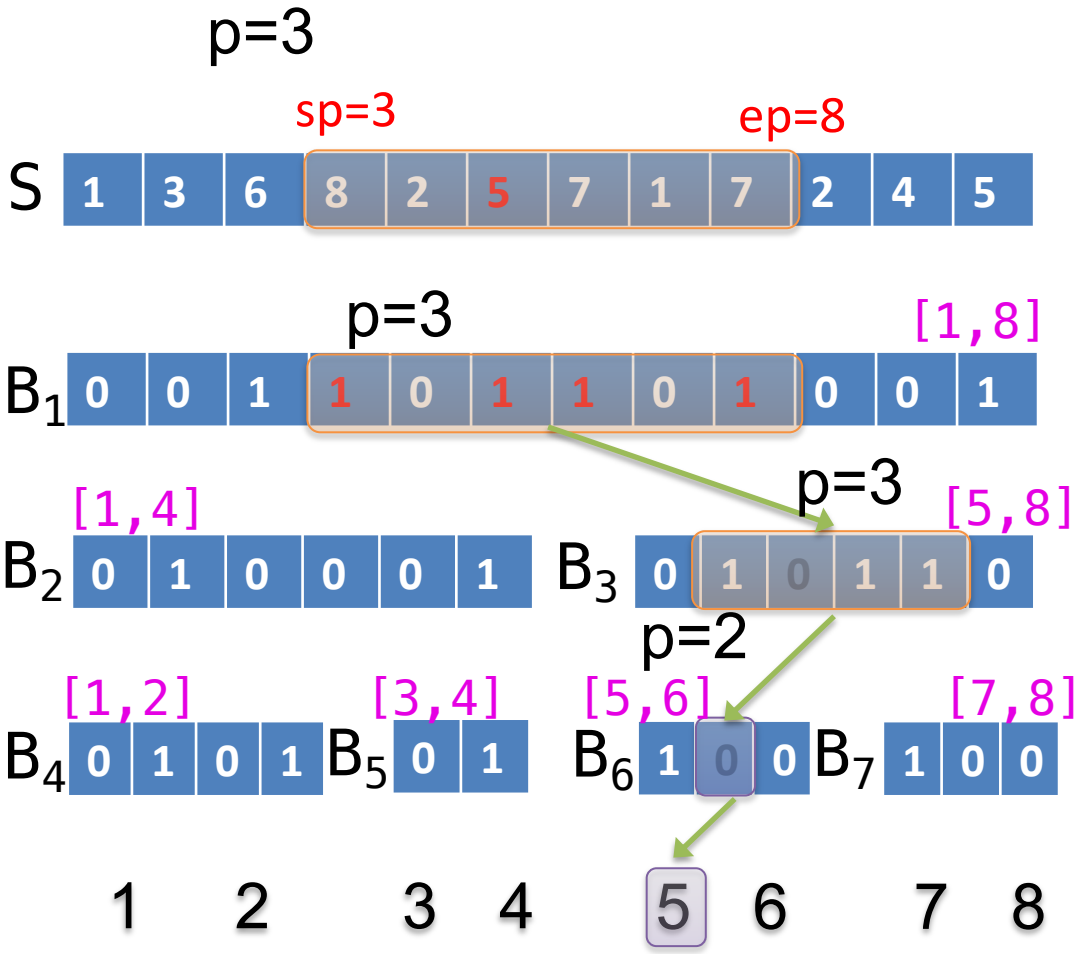
- Start from the root B_1 with interval [sp, ep]
- If an interval includes 1, go to the right child
- Otherwise, go to the left child
- Maximum value is found at the arrived leaf

Remark: O(1)-time division of an interval using rank operation

- Rank operations: Utilize $rank_0$ and $rank_1$ to determine how to split an interval between the left and right children efficiently:
- Left division ($rank_0$): Determines how many elements in the interval fall to the left child, representing lower values.
- Right division ($rank_1$): Determines how many elements in the interval go to the right child, representing higher values.
- Execution time: These operations allow the interval to be divided in constant (O(1)) time, regardless of the size of the interval.



RangeQuantile(S,p,sp,ep)
: p-th largest value in S[sp,ep]

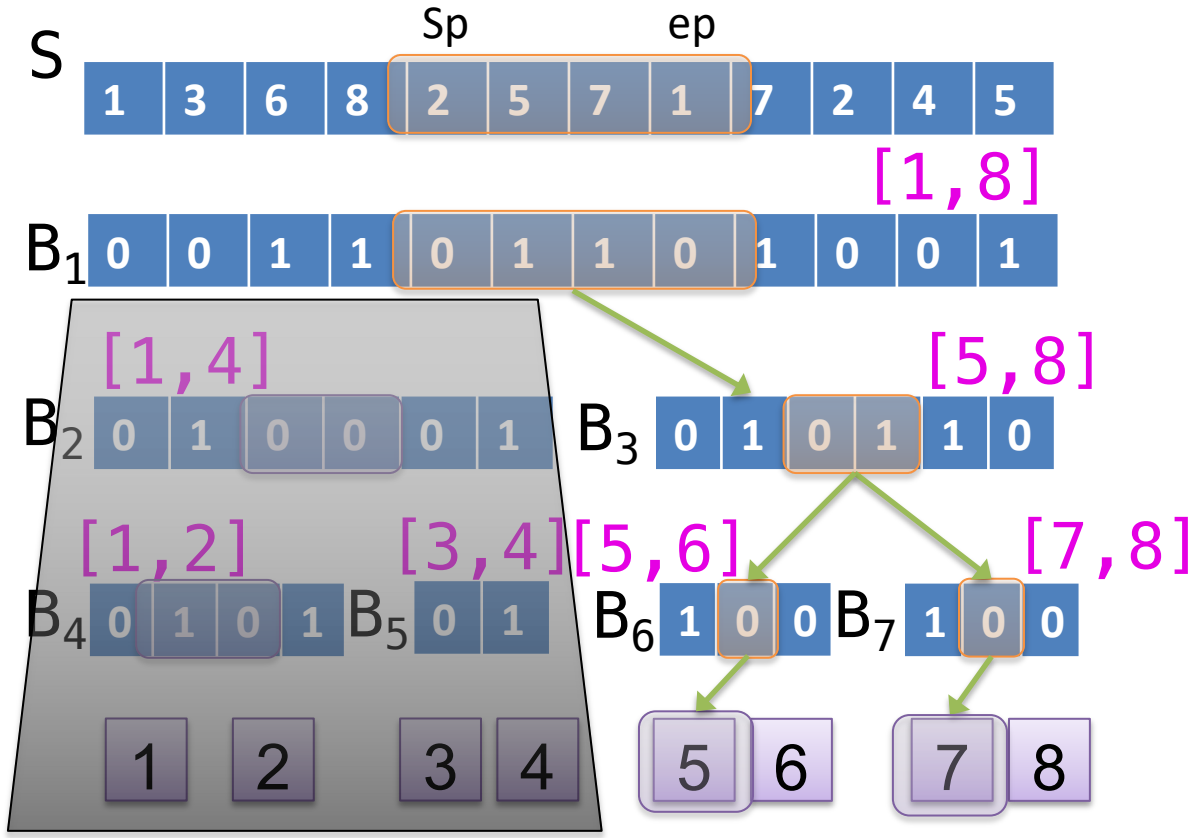


- Start from the root with interval [sp,ep]
- If the number of 0s is no less than p, go to the right child
- Otherwise, set p as (p - #0s) and go to the left child
- p-th largest value is the value corresponding to the arrived leaf

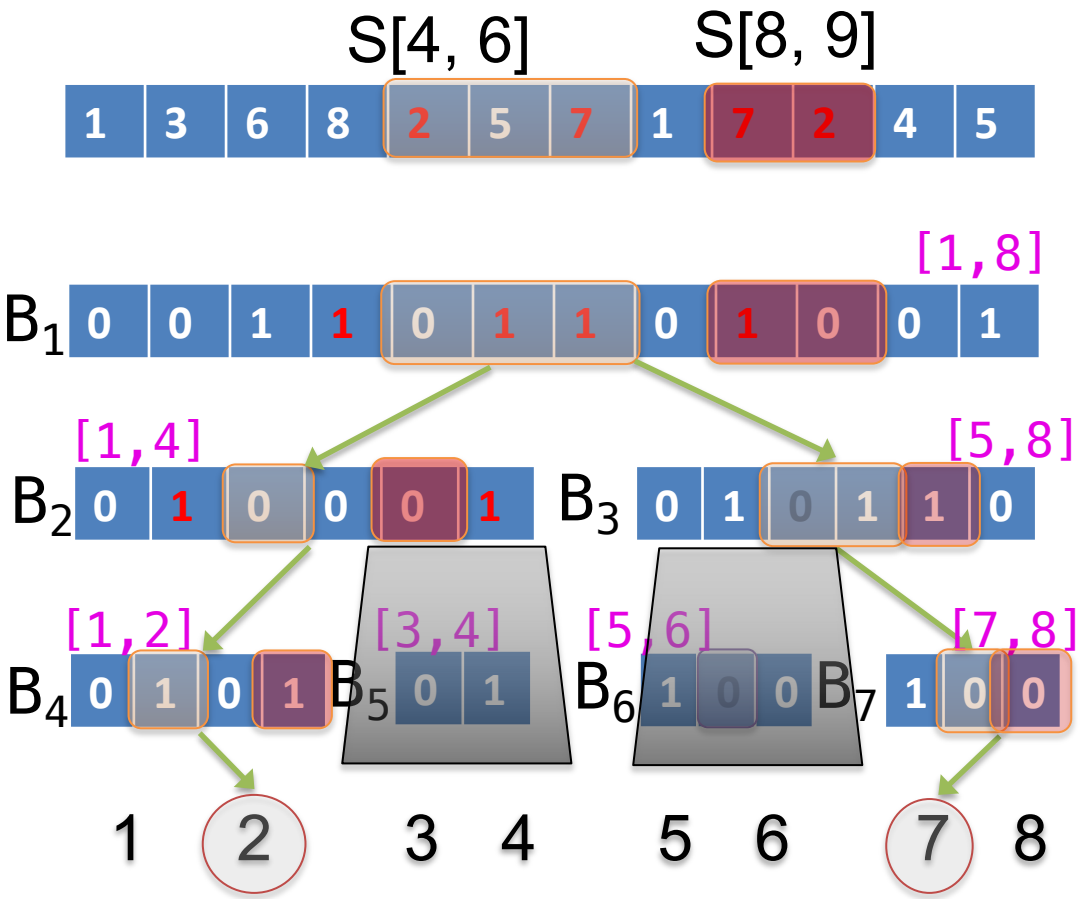
EnumElem(S,sp,ep,min,max): Compute all the values of at least min and at most max in S[sp,ep]

- Traverse all the nodes with intervals overlapped with range [min, max]
- Solutions : elements in all the arrived leaves

[min,max]=[5,7]



RangeInt(S,sp1,ep1,sp2,ep2) : Compute all the common integers in two intervals [sp1,ep1] and [sp2,ep2] on S



- Initiate at root: Begin with two interval [sp1, ep1] and [sp2, ep2], at the root
- Traverse all the nodes with two intervals
- Solutions : elements in all the arrived leaves

Operations on wavelet tree

Operation	Description
Access(i)	S[i]
$Rank_{\sigma}(S,i)$	Number of σs in S[1,i]
$Select_{\sigma}(S,i)$	Position of ith occurrence of σ
MaxElem(S,sp,ep)	Maximum value in S[sp,ep]
MinElem(S,sp,ep)	Minimum value in S[sp,ep]
RangeQuantile(S,p,sp,ep)	pth largest value in S[sp,ep]
EnumElem(S,sp,ep,l,h)	All the values with at least l and at most h in S[sp,ep]
MaxRange(S,sp,ep,l,h)	The most frequently appearing integer with at least l and at most h in S[sp,ep]
RangeInt(S,sp1,ep1,sp2,ep2)	All the common value in S[sp1,ep1] and S[sp2,ep2]

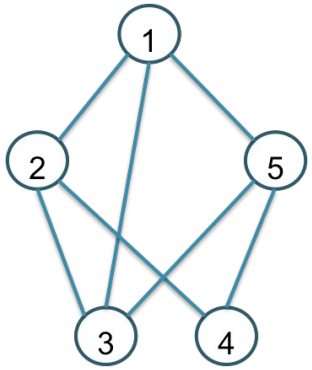
Recommendation system using WT: Identifying node similarity

- In social networks, each node represents a user, and the similarity metric is based on the number of common friends (common adjacent nodes).

1. Create an adjacency list: Each user's list of friends is compiled
2. Concatenate lists: all rows from the adjacenty list into a single array S
3. Build wavelet tree: construct a WT from S

- Similarity between = # of common friends
  two users = RangeInt(S,sp_1,ep_1,sp_2,ep_2)

Social graph



Adjacency lists

Node			
1	2	3	5
2	1	3	4
3	1	2	5
4	2	5	
5	1	3	4

Concatenate

S 2 3 5 1 3 4 1 2 5 2 5 1 3 4

Wavelet Tree

Implementation remarks

- Wavelet matrix [F.Claude et al., 2011]
  - Implementation is easy
  - Has the same operations as wavelet tree

- Dynamic version [G.Navarro, Y.Nekrich, SODA, 2013]
  - Support insertion and deletion of symbols