
Succinct Data Structures: Towards Optimal Representation of Massive Data

S. Srinivasa Rao

School of CSE
Seoul National University

Outline

□ Succinct data structures

- Introduction
- Examples and applications

□ Tree representations

- Motivation
- Heap-like representation for binary trees
- Parenthesis representation for ordered trees
- Applications

□ Random access to compressed data

□ Range searching problems

Succinct Data Structures

Succinct data structures

- Goal: represent the data in close to **optimal space**, while supporting **fast queries**.

(optimal -- information-theoretic lower bound)

Introduced by [Jacobson, FOCS '89]

- An “extension” of data compression.

(Data compression:

- Achieve close to optimal space
- Queries need **not** be supported efficiently)

Applications

- Succinct data structures are most useful when there is:
 - **Massive amounts of data** to be stored: DNA sequences, geographical/astronomical data, search engines etc.
 - **Limited amount of memory** available: small memory devices like PDAs, mobile phones etc.

By making the size of the data structure small, we may be able to keep it in higher levels of memory hierarchy and access it much faster.

Applications

□ Web Graph

- Eg.: uk-union-2006-06-2007-05
- #Nodes: 133,633,040
- #Edges: 5,507,679,822
- A plain representation requires 22GB!
- A succinct representation takes <2GB.

□ XML data [Delpratt et al. 2006]

- Data: XML trees with 57K to 160M nodes
- Result: 3.12 to 3.81 bits per node
- The space cost is merely a tiny percentage of an explicit representation!
- Fast queries at the same time.

Examples of SDS

- Trees, Graphs
- Bit vectors, Sets
- Dynamic arrays
- Permutations, Functions

- Text indexes
 - suffix trees/suffix arrays etc.
- XML documents, File systems (labeled, multi-labeled trees)
- DAGs and BDDs
- ...

Example: Text Indexing

- A text string T of length n over an alphabet Σ can be represented using
 - $n \log |\Sigma| + o(n \log |\Sigma|)$ bits,
(or the even the k -th order entropy of T)

to support the following pattern matching queries (given a pattern P of length m):

- count the # occurrences of P in T ,
- report all the occurrences of P in T ,
- output a substring of T of given length

in almost optimal time.

Better space and functionality than **inverted indices**.

Example: Compressed Suffix Trees

- A string T of length n over an alphabet Σ can be stored using $O(n \log |\Sigma|)$ bits, to support all the operations supported by a standard suffix tree such as pattern matching queries, suffix links, string depths, lowest common ancestors etc. with slight slowdown.
- Note that standard suffix trees use $O(n \log n)$ bits.

Succinct Tree Representations

Motivation

Trees are used to represent:

- Directories (Unix, all the rest)
- Search trees (B-trees, binary search trees, digital trees or tries)
- Graph structures (we do a tree based search)
- Search indexes for text (including DNA)
 - Suffix trees
- XML documents
- ...

Drawbacks of standard representations

- Standard representations of trees support very few operations. To support other useful queries, they require a large amount of extra space.
- In various applications, one would like to support operations like “subtree size” of a node, “least common ancestor” of two nodes, “height”, “depth” of a node, “ancestor” of a node at a given level etc.

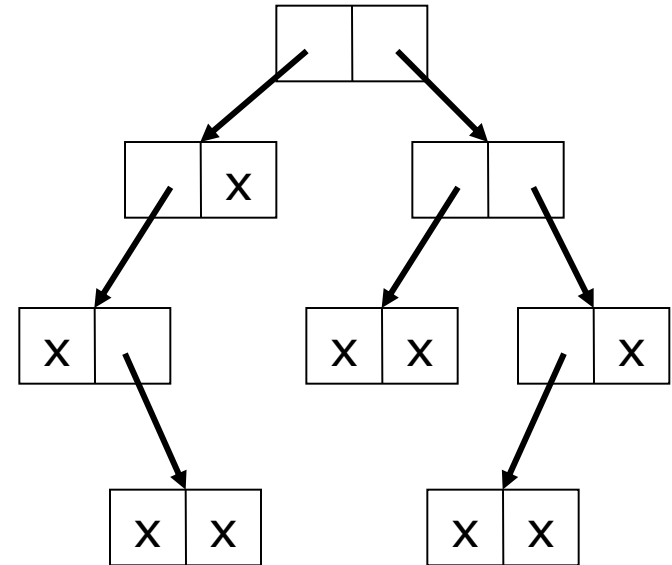
Drawbacks of standard representations

- The space used by the tree structure could be the dominating factor in some applications.
 - **Eg.** More than half of the space used by a standard **suffix tree** representation is used to store the tree structure.
- “A pointer-based implementation of a suffix tree requires more than **20n** bytes. A more sophisticated solution uses at least **12n** bytes in the worst case, and about **8n** bytes in the average. For example, **a suffix tree built upon 700Mb of DNA sequences may take 40Gb of space.**”
 - Handbook of Computational Molecular Biology, 2006

Standard representation of (binary) trees

Binary tree:
each node has two
pointers to its left
and right children

An n -node tree takes
 $2n$ pointers or $2n \lg n$ bits
(can be easily reduced to
 $n \lg n + O(n)$ bits).



Supports finding **left child** or **right child** of a node
(in constant time).

For each extra operation (eg. **parent**, **subtree size**)
we have to pay, roughly, an additional $n \lg n$ bits.

Can we improve the space bound?

- There are less than 2^{2n} distinct binary trees on n nodes.
- $2n$ bits are enough to distinguish between any two different binary trees.
- Can we represent an n node binary tree using $2n$ bits?

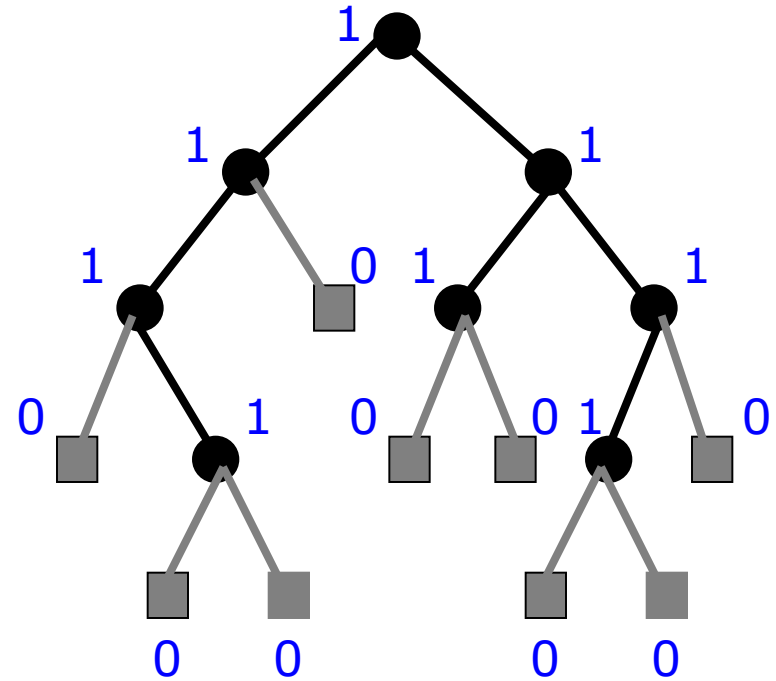
Heap-like notation for a binary tree

Add external nodes

Label internal nodes with a 1
and external nodes with a 0

Write the labels in level order

1 1 1 1 0 1 1 0 1 0 0 1 0 0 0 0 0



One can reconstruct the tree from this sequence

An n node binary tree can be represented in $2n+1$ bits.

What about the operations?

Heap-like notation for a binary tree

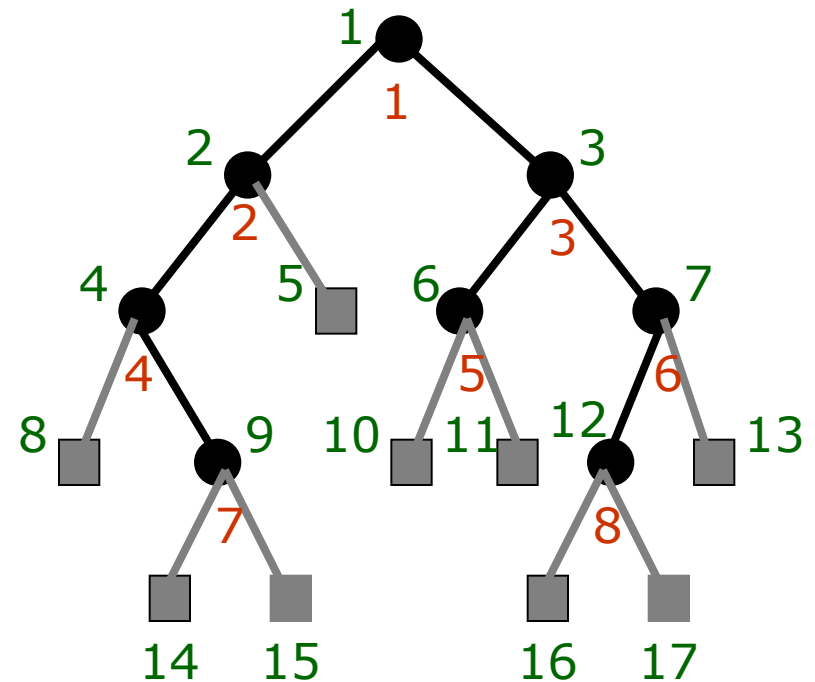
left child(x) = $[2x]$

right child(x) = $[2x+1]$

parent(x) = $[\lfloor x/2 \rfloor]$

$x \rightarrow x$: # 1's up to x

$x \rightarrow x$: position of x -th 1



1	2	3	4	5	6	7	8										
1	1	1	1	0	1	1	0	1	0	0	1	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	

Rank/Select on a bit vector

Given a bit vector B

$\text{rank}_1(i) = \#$ 1's up to position i in B

$\text{select}_1(i) =$ position of the i -th 1 in B

(similarly rank_0 and select_0)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
B: 0 1 1 0 1 0 0 0 1 1 0 1 1 1 1

Given a bit vector of length n , by storing an additional $o(n)$ -bit structure, we can support all four operations in $O(1)$ time.

$\text{rank}_1(5) = 3$
 $\text{select}_1(4) = 9$
 $\text{rank}_0(5) = 2$
 $\text{select}_0(4) = 7$

An important substructure in most succinct data structures.

Implementations: [Kim et al.], [Gonzalez et al.], ...

Binary tree representation

□ A binary tree on n nodes can be represented using $2n+o(n)$ bits to support:

- parent
- left child
- right child

in constant time.

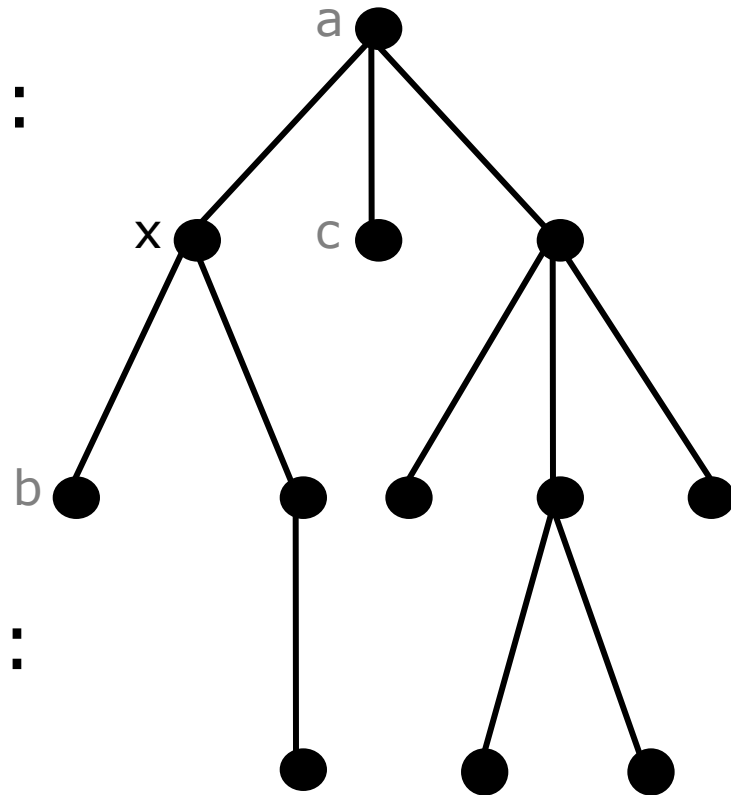
[Jacobson '89]

Ordered trees

A rooted ordered tree (on n nodes):

Navigational operations:

- $\text{parent}(x) = a$
- $\text{first child}(x) = b$
- $\text{next sibling}(x) = c$



Other useful operations:

- $\text{degree}(x) = 2$
- $\text{subtree size}(x) = 4$

Parenthesis representation

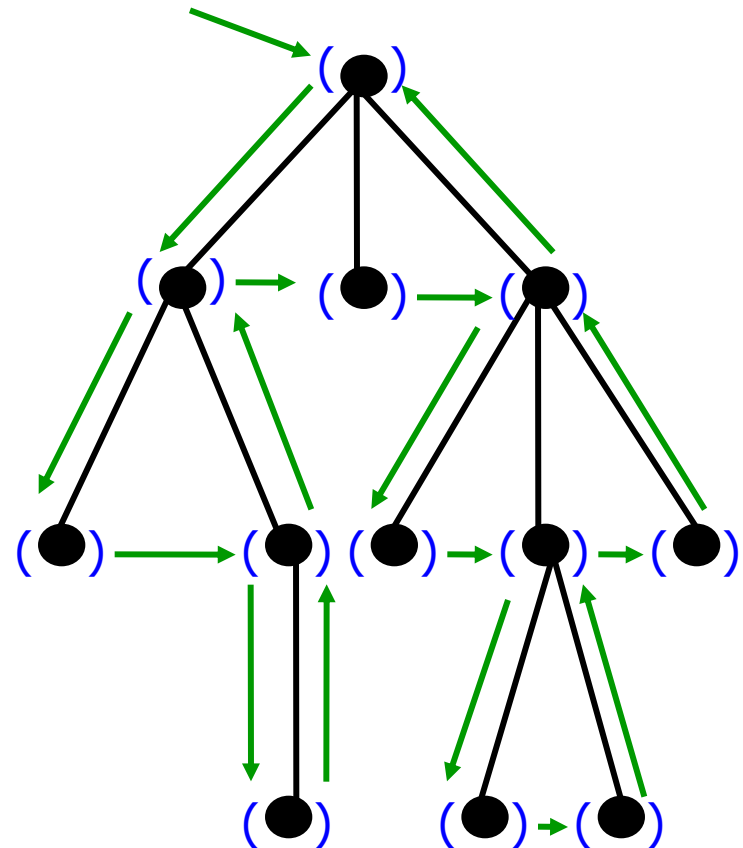
Associate an open-close parenthesis-pair with each node

Visit the nodes in pre-order, writing the parentheses

length: $2n$

space: $2n$ bits

One can reconstruct the tree from this sequence



((() (())) () (() (() ()) ()))

Operations

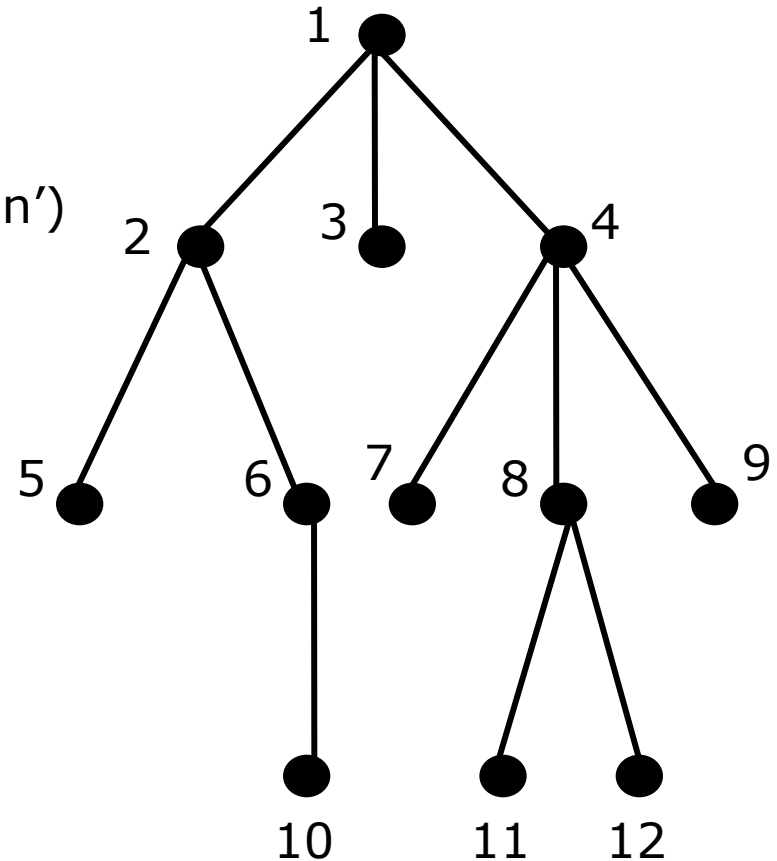
parent – enclosing parenthesis

first child – next parenthesis (if 'open')

next sibling – open parenthesis following the matching closing parenthesis (if exists)

subtree size – half the number of parentheses between the pair

with $O(n)$ extra bits, all these can be supported in constant time



((() (())) () (() (() ()) ()))
1 2 5 6 10 6 2 3 4 7 8 11 12 8 9 4 1

Parenthesis representation

- Space: $2n+o(n)$ bits

- Supports:

- parent

- first child

- next sibling

- subtree size

- degree

- depth

- height

- level ancestor

- LCA

- leftmost/rightmost leaf

- number of leaves in the subtree

- next node in the level

- pre/post order number

- i-th child

in constant time.

[Munro-Raman '97] [Munro et al. '01] [Sadakane '03] [Lu-Yeh '08]

Implementation: [Geary et al. '04]

Other methods

- Space: $2n+o(n)$ bits

- Supports:

- parent
- first child
- next sibling
- subtree size
- degree
- depth
- height
- level ancestor
- LCA
- leftmost/rightmost leaf
- number of leaves in the subtree
- next node in the level
- pre/post order number
- i-th child

in constant time.

Tree covering: [Geary et al. '04] [He et al. '07] [Farzan-Munro '08]

DFUDS: [Demaine et al. '05] [Jansson et al. '07]

Universal: [Farzan-Raman-Rao, '09]

Fully-Functional: [Sadakane-Navarro '10]

Applications

□ Representing

- suffix trees
- XML documents (supporting XPath queries)
- file systems (searching and Path queries)
- representing BDDs
- ...

Random access to compressed data

Random Access to Compressed Strings



- What is the i^{th} character?
- What is the substring at $[i,j]$?
- Does pattern P appear in text? (perhaps with k errors?)

Random Access to Grammar Compressed Strings

AGTAGTAG

$N = 8$

$X_7 \rightarrow X_6X_3$

$X_6 \rightarrow X_5X_5$

$X_5 \rightarrow X_3X_4$

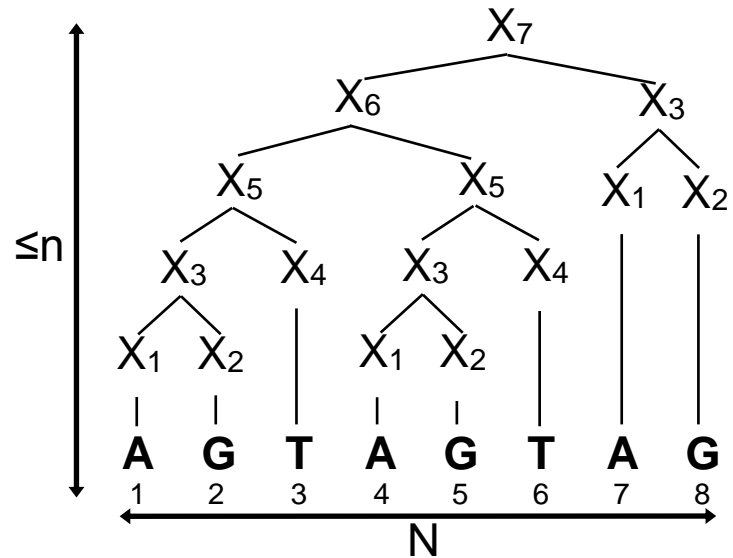
$X_4 \rightarrow \mathbf{T}$

$X_3 \rightarrow X_1X_2$

$X_2 \rightarrow \mathbf{G}$

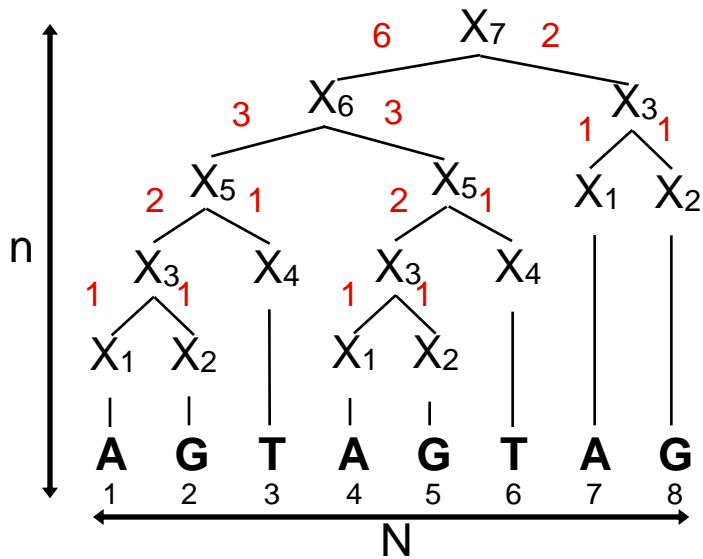
$X_1 \rightarrow \mathbf{A}$

$n = 7$



- Grammar based compression captures many popular compression schemes with no or little blowup in space [Charikar et al. 2002, Rytter 2003].
- Lempel-Ziv family, Sequitur, Run-Length Encoding, Re-Pair, ...

Tradeoffs and Results



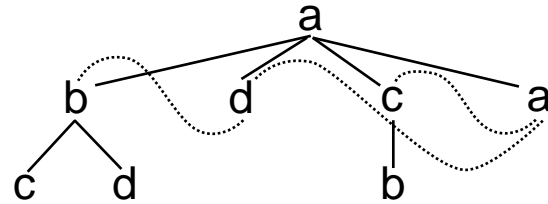
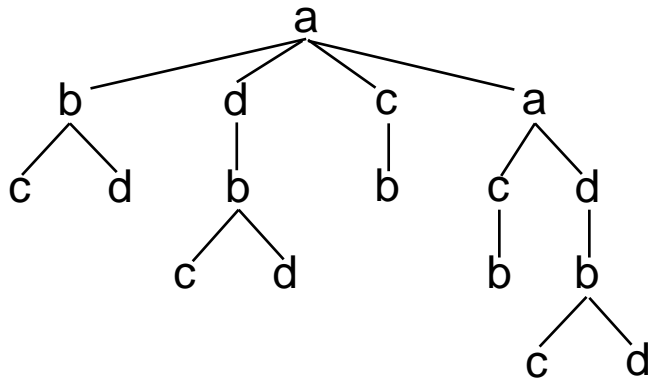
- What is the i^{th} character?

$O(N)$ space	$O(n)$ space	$O(n)$ space
$O(1)$ query	$O(n)$ query	$O(\log N)$ query

- What is the substring at $[i, j]$?

$O(n)$ space
$O(\log N + j - i)$ query

Extension: Compressed Trees

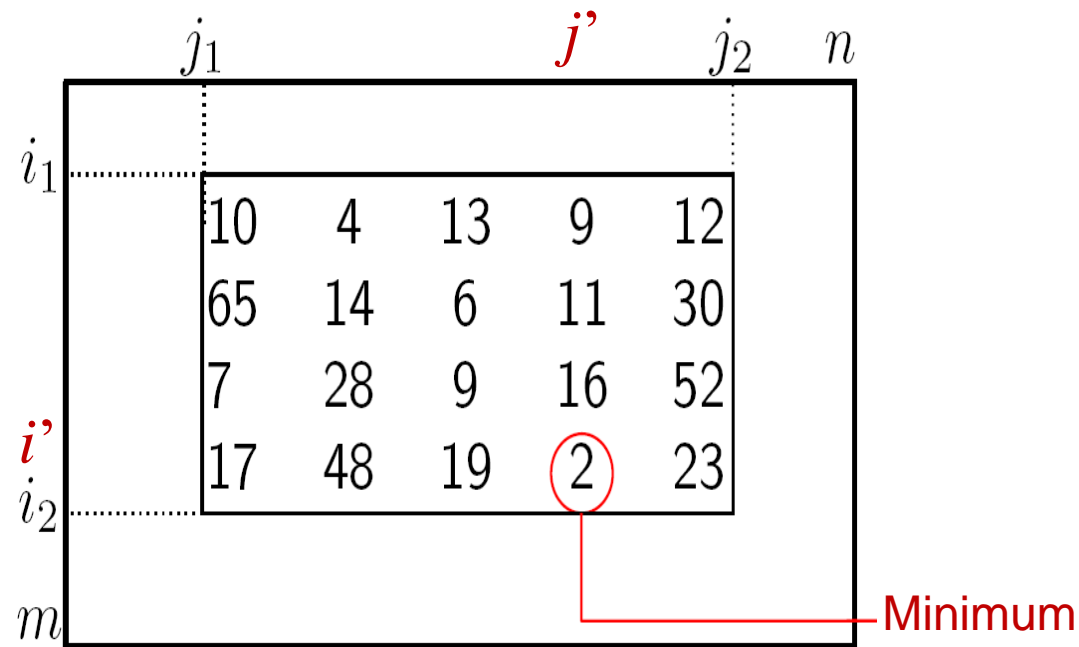


- Linear space in compressed tree.
- Fast navigation operations (`select`, `access`, `parent`, `depth`, `height`, `subtree_size`, `first_child`, `next_sibling`, `level_ancestor`, `lca`): $O(\log N)$ time.

Range Searching Problems

The 2D Range Minimum Problem

Introduced by Amir et al. (2007) as a generalization of the classic 1D RMQ problem.



- Input: an $m \times n$ -matrix of size $N = m \cdot n$, $m \leq n$.
- **Preprocess** the matrix s.t. range minimum queries are efficiently supported.

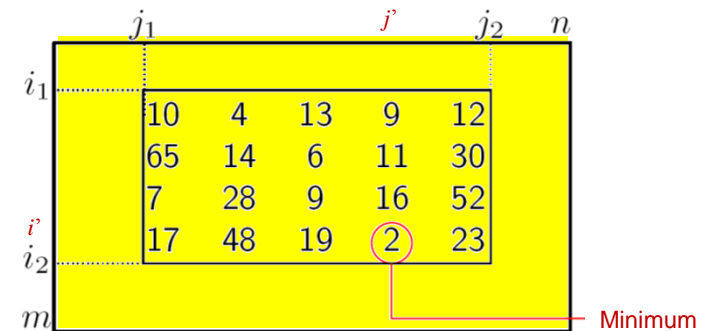
Models

Encoding model

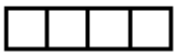
Queries can access **data structure** but not input matrix

Indexing model

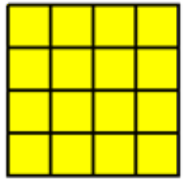
Queries can access **data structure** and read **input matrix**



1D Range Minimum Queries

	Indexing	Upper Bound	$\left\{ \begin{array}{l} \text{Time} = O(1) \\ \text{Space} = 2n + o(n) + A \text{ bits} \end{array} \right.$
		Lower Bound	$\left\{ \begin{array}{l} \text{Time} = O(c) \\ \text{Space} = O(N/c) + A \text{ bits} \end{array} \right.$
	Encoding	Upper Bound	$\left\{ \begin{array}{l} \text{Time} = O(1) \\ \text{Space} = 2n + o(n) \text{ bits} \end{array} \right.$
		Lower Bound:	$\text{Space} = 2n - \Theta(\log n) \text{ bits}$

2D Range Minimum Queries



Indexing

Upper Bound

$$\text{Time} = O(1)$$

$$\text{Space} = O(N) + |A| \text{ bits}$$

$$\text{Time} = O(c \log^2 c)$$

$$\text{Space} = O(N/c) + |A| \text{ bits}$$

Lower Bound

$$\text{Time} = \Omega(c)$$

$$\text{Space} = O(N/c) + |A| \text{ bits}$$

Encoding

Upper Bound

$$\text{Time} = O(1)$$

$$\text{Space} = O(N \log n) \text{ bits}$$

Lower Bound:

$$\text{Space} = \Omega(N \log m) \text{ bits}$$

1D Range Top-k Problem

- Input: an array A of size n , and a parameter $k < n$.
- **Preprocess** the array s.t. **range-top-k queries** or **k^{th} -pos** are efficiently supported.
- **range-top-k(i,j)** returns the positions of the k largest values in $A[i,j]$
- **k^{th} -pos(i,j)** returns the position of the k^{th} largest value in $A[i,j]$

Eg.:

15	12	9	21	18	25	14	29
----	----	---	----	----	----	----	----

- **range-top-2(3,7)** returns the positions $\{4, 6\}$.
- **4^{th} -pos(2,8)** returns 5

1D Range Top-k results

Problem	Variant	Lower bound space (bits)	Upper bound space (bits)	Upper bound time
kth-pos	optimal space	$n \lg k - O(n)$	$n \lg k + o(n \lg k) + n$	any $\omega(1)$
	optimal time	$n \lg k - O(n)$	$(1 + \epsilon)n \lg k$	$O(1/\epsilon)$
top-k-pos	one-sided	$n \lg k - O(n)$	$n \lg k + o(n \lg k) + n$	$O(k)$
	two-sided	$n \lg k - O(n)$	$O(n \lg k)$	$O(k)$

- All one-sided queries can be answered with optimal space and time.
- For two-sided k^{th} -pos queries, we achieve close to optimal trade-offs.

Current and future work

- Making the succinct structures dynamic (there are some existing results)
- Implementations of SDSs
- Indexing compressed XML (labeled trees) (two different approaches supporting different sets of operations)
- Other memory models
 - External memory model (a few recent results)
 - Flash memory model
 - (So far mostly RAM model)

Thank You