Succinct Minimum Range Queries

Simo Salmirinne

University of Helsinki

20.03.2018

Outline

- Introduction
- Preliminaries
- Range minimum queries
- Conclusions

Range Minimum Query

Let $A[1, n]$ be an array of comparable elements, such as integers. $\text{RMQ}_A(i, j)$ asks for the position of the minimum value in $A[i, j]$.

Formally, $\text{RMQ}_A(i, j) = \text{argmin}_{i \leq k \leq j} A[k]$.

Range Minimum Query

Could we just store the result of $\mathrm{RMQ}_A(i, j)$ for all possible pairs $i < j$ in a hashtable? We could, if A is small.

Instead, we build a succinct data structure to answer queries on-line. A succinct data structure is a data structure that uses space close to the information-theoretic lower bound while still supporting operations efficiently.

Where is this problem useful?
Text indexing, pattern matching, text compression, to name a few.

Range Minimum Query

The succinct data structure implementation of Ferrada and Navarro (2016) takes only $2n + o(n)$ bits.

Slightly different implementation of Heun and Fischer (2011) that also takes $2n + o(n)$ bits and answers queries in constant time.

In this presentation, we stick to the implementation of Ferrada and Navarro.

Lowest Common Ancestor

Close relation to Lowest Common Ancestor (LCA). Given an ordinal tree and nodes i and j, LCA asks for the lowest node, that is an ancestor of both i and j.

We actually reduce our RMQ problem to a LCA problem.

Procedure

The data structure that is presented here is constructed as follows

- Input array A
- Cartesian tree
- Generalized ordinal tree
- Balanced parentheses
- Range min-max tree
- ...profit?

What are all these steps? How do they support minimum range queries? Are they necessary?

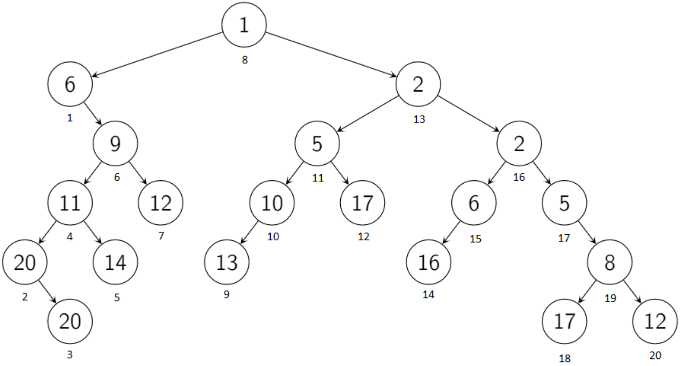
Cartesian tree

A Cartesian tree is defined recursively as follows: set $r = \min A[1, n]$ as root of the CT, then recursively set $r_{\text{left}} = \min A[1, r-1]$ as the left child of r and $r_{\text{right}} = \min A[r+1, n]$ as the right child of r.

Let us construct a CT from the following array A

A =	**6**	**20**	**20**	**11**	**14**	**9**	**12**	**1**	**13**	**10**	**5**	**17**	**2**	**16**	**6**	**2**	**5**	**17**	**8**	**12**
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Cartesian tree



A =	**6**	**20**	**20**	**11**	**14**	**9**	**12**	**1**	**13**	**10**	**5**	**17**	**2**	**16**	**6**	**2**	**5**	**17**	**8**	**12**
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

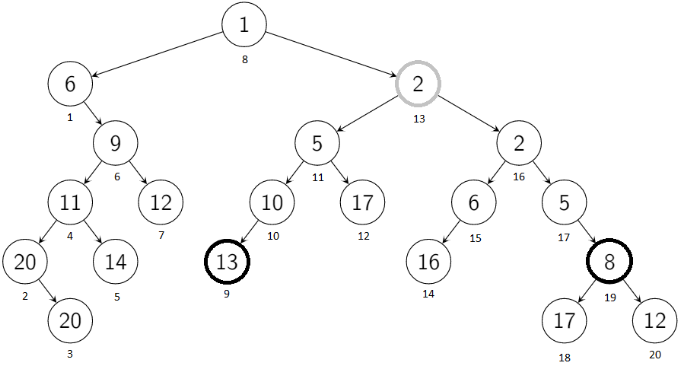
Cartesian tree

Key observation: a node with position i in A has in-order position i in the CT.

It turns out that RMQ_A and LCA_{CT} has the following relation $\text{RMQ}_A(i, j) = \text{inorder}(\text{LCA}_{\text{CT}}(\text{inorder}(i), \text{inorder}(j)))$.

RMQ - LCA

Example $\mathrm{RMQ}_A(9, 19) = \mathrm{inorder}(\mathrm{LCA}_{CT}(\mathrm{inorder}(9), \mathrm{inorder}(19)))$



A =	**6**	**20**	**20**	**11**	**14**	**9**	**12**	**1**	**13**	**10**	**5**	**17**	**2**	**16**	**6**	**2**	**5**	**17**	**8**	**12**
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

RMQ - LCA

What now? We represent the structure of the CT in a succinct form and formulate the LCA to work on that.

We can represent any ordinal tree structure in just $2n$ bits using Balanced Parentheses.

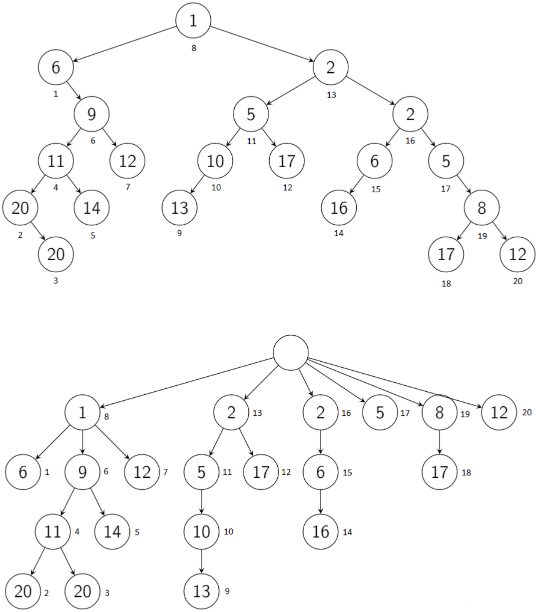
However, we have a problem, because in the BP representation we can not distinguish a left child from the right child in the CT (actually we can, but it takes an unnecessary amount of extra space).

Generalized ordinal tree

To make the distinction, we use a bijection between CTs and generalized ordinal trees, called the rightmost-path mapping.

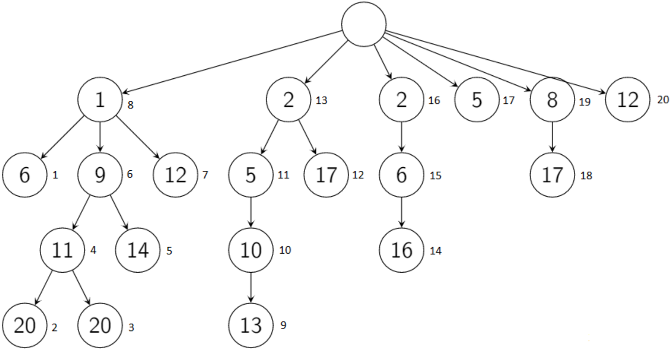
For this, we have the following procedure:
We append an empty node r_G as the root of G. We then set the root r_{CT} as the leftmost child of r_G. Now, we recursively traverse the rest of the nodes in the CT starting from the children of r_{CT}. If a node is the left child of its parent, it remains the left child of its parent in G. If a node is the right child of its parent, it becomes the rightmost child of its parents parent in G, hence the rightmost-path mapping.

Bijection of CT and G



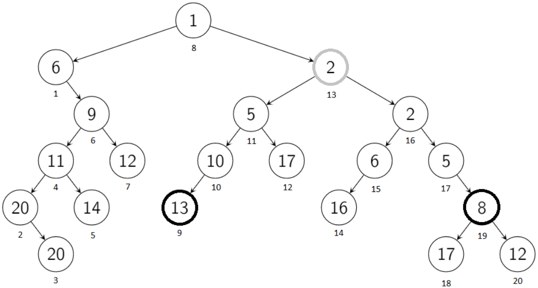
Generalized ordinal tree

Key observation: a node with position i in A has post-order position i in G.

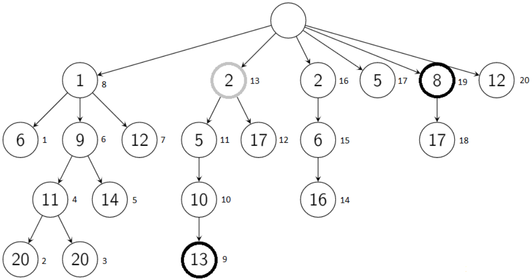


A =	**6**	**20**	**20**	**11**	**14**	**9**	**12**	**1**	**13**	**10**	**5**	**17**	**2**	**16**	**6**	**2**	**5**	**17**	**8**	**12**
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

LCA on G



1
8
6
1
2
13
9
6
5
11
2
16
11
4
12
7
10
10
17
12
6
15
5
17
20
2
14
5
13
9
16
14
8
19
20
3
17
18
12
20



1
8
2
13
2
16
5
17
8
19
12
20
6
1
9
6
12
7
5
11
17
12
6
15
17
18
11
4
14
5
10
10
16
14
20
2
20
3
13
9

Balanced Parentheses

Now, we can represent the generalized ordinal tree G in just $2n$ bits using Balanced Parentheses.

Then, we adjust the LCA formula on CTs to account for the bijection between the trees and the transformation into a BP. We end up with an elegant formula that operates on bit sequences and does exactly what we want.

First, what is Balanced Parentheses?

Balanced Parentheses

A Balanced Parentheses (BP) is a bit sequence of $B[1, 2n]$ bits, that has an equal number of opening parentheses '(' and closing parentheses ')'.

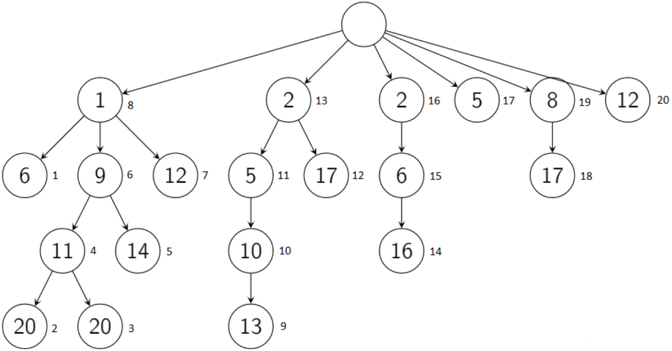
Every opening parentheses $B[i]$ is associated with a closing parentheses $B[j]$, such that $j > i$. Such associations $B[i, j]$ form a range called a *segment*. In a BP, any two segments are either disjoint or one is contained in the other.

Any subtree starting from a node v is a contiguous segment in the bit sequence $B[i, j]$ contained by v. This is important!

Balanced Parentheses

Construction: we traverse G in depth-first order starting from the root. The first time we arrive at a node v, we output an opening parentheses '(' and then we recursively traverse the sub-tree starting from the left-most child of v. After the recursion, when we finally leave node v, we output the closing parentheses ')'. The recursion completes when we finally arrive back at the root outputting a ')'.

Balanced Parentheses



(	(	(	)	(	(	(	)	(	)	)	(	)	)	(	)	)	(	(	(	(	)	)	)	(	)	)	(	(	(	)	)	)	(	)	(	(	)	)	(	)	)
1	1	1	0	1	1	1	0	1	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	0	1	1	0	0	1	0	0
1	2	3	2	3	4	5	4	5	4	3	4	3	2	3	2	1	2	3	4	5	4	3	2	3	2	1	2	3	4	3	2	1	2	1	2	3	2	1	2	1	0

Range minimum queries

It turns out, our original RMQ_A and BP has the following relation:
$\mathrm{RMQ}_A(i, j) = \mathrm{rank}_0(\mathrm{RMQ}_D(\mathrm{select}_0(i), \mathrm{select}_0(j)))$.

First, lets define operations access, select, rank and excess on bit sequences.

Range minimum queries

access(i) — return the bit at index i

select$_b$(i) — return the position of the ith b bit

rank$_b$(i) — return the number of occurrences of bit b up to position i

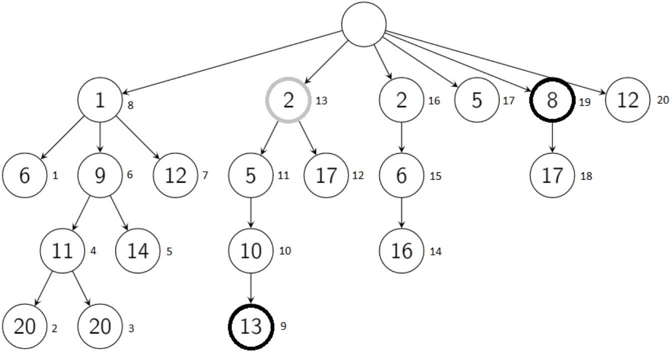
excess(i) — return the number of occurrences of 1s minus the number of occurrences of 0s up to position i

Operation select$_0$(10) returns 23, operation rank$_0$(10) returns 3 and excess(10) returns 4.

(	(	(	)	(	(	(	)	(	)	)	(	)	)	(	)	)	(	(	(	(	)	)	)	(	)	)	(	(	(	)	)	)	(	)	(	(	)	)	(	)	)
1	1	1	0	1	1	1	0	1	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	0	1	1	0	0	1	0	0
1	2	3	2	3	4	5	4	5	4	3	4	3	2	3	2	1	2	3	4	5	4	3	2	3	2	1	2	3	4	3	2	1	2	1	2	3	2	1	2	1	0

Range minimum queries

$$\mathrm{RMQ}_A(i,j) = \mathrm{rank}_0(\mathrm{RMQ}_D(\mathrm{select}_0(i), \mathrm{select}_0(j)))$$



(	(	(	)	(	(	(	)	(	)	)	(	)	)	(	)	)	(	(	(	(	)	)	)	(	)	)	(	(	(	)	)	)	(	)	(	(	)	)	(	)	)
1	1	1	0	1	1	1	0	1	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	0	1	1	0	0	1	0	0
1	2	3	2	3	4	5	4	5	4	3	4	3	2	3	2	1	2	3	4	5	4	3	2	3	2	1	2	3	4	3	2	1	2	1	2	3	2	1	2	1	0

Range min-max tree

Range min-max tree is a heap structure built on top of BP to support operations on the bit sequence efficiently.

This is the $o(n)$ part of the space consumption.

Here we divide the BP into blocks of size $b = 8$ (in practice, the blocks are much larger, say $b = 512$). These blocks are the leaves of the heap. The internal nodes cover multiple blocks of the BP.

In total, there are $2\lceil(2n+2)/b\rceil - 1$ blocks.

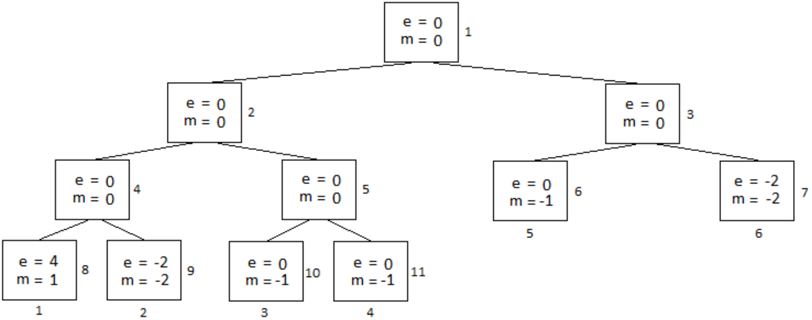
Range min-max tree

Each block contains the following information

- *e* - local excess of the block
- *m* - local minimum excess of the block.

Typically, these values are rather small (unless the tree is really skewed at some range).

Range min-max tree



(	(	(	)	(	(	(	)	(	)	)	(	)	)	(	)	)	(	(	(	(	)	)	)	(	)	)	(	(	(	)	)	)	(	)	(	(	)	)	(	)	)
1	1	1	0	1	1	1	0	1	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	0	1	1	0	0	1	0	0
1	2	3	2	3	4	5	4	5	4	3	4	3	2	3	2	1	2	3	4	5	4	3	2	3	2	1	2	3	4	3	2	1	2	1	2	3	2	1	2	1	0

Range min-max tree

Example

- A has $n = 1000000$ elements
- Set block length $b = 512$
- rmM-tree has $2\lceil(2n+2)/b\rceil - 1 \approx 8000$ blocks
- Assume $\max(\max|e|, \max|m|) = 25$
- Each e and m can be represented in 6 bits
- Total space usage is then $2 \cdot 6 \cdot 8000 = 96000$ bits
- Sublinear with respect to n.

Conclusions

Fastest and most compact succinct data structure for RMQs to date.