Range Minimum Query Problem

Sridhar Alagar

RMQ Problem

Input: An array A and two integers i and j

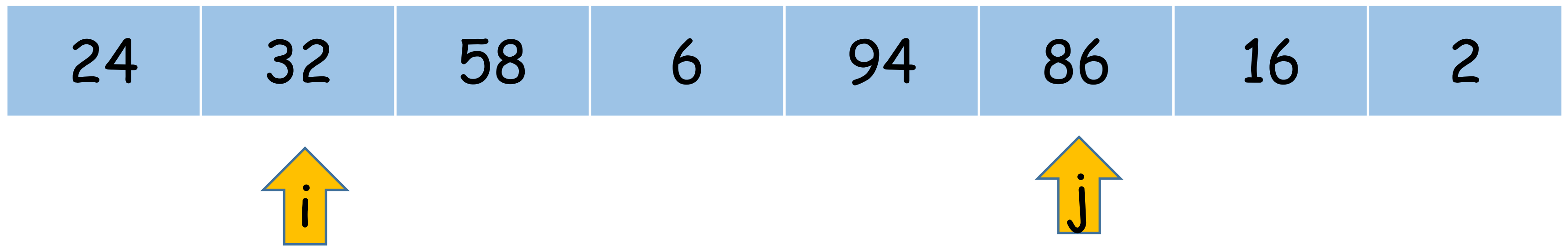
Output: Find the smallest element among A[i]…A[j]

24	32	58	6	94	86	16	2

RMQ Problem

Input: An array A and two integers i and j

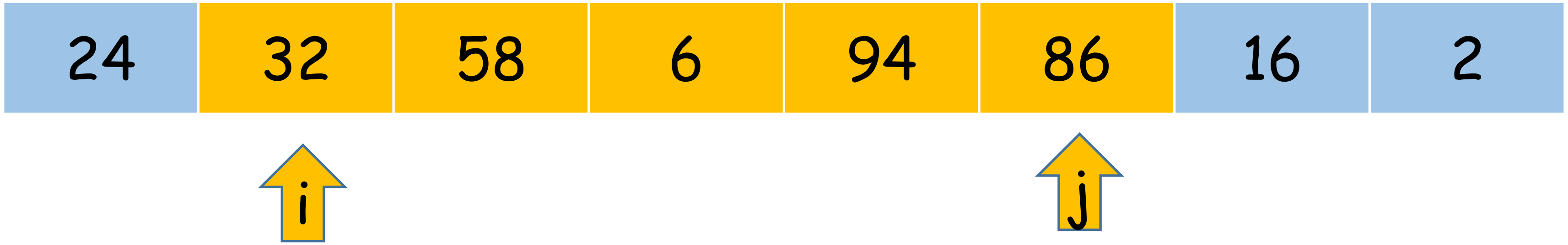
Output: Find the smallest element among A[i]…A[j]



RMQ Problem

Input: An array A and two integers i and j

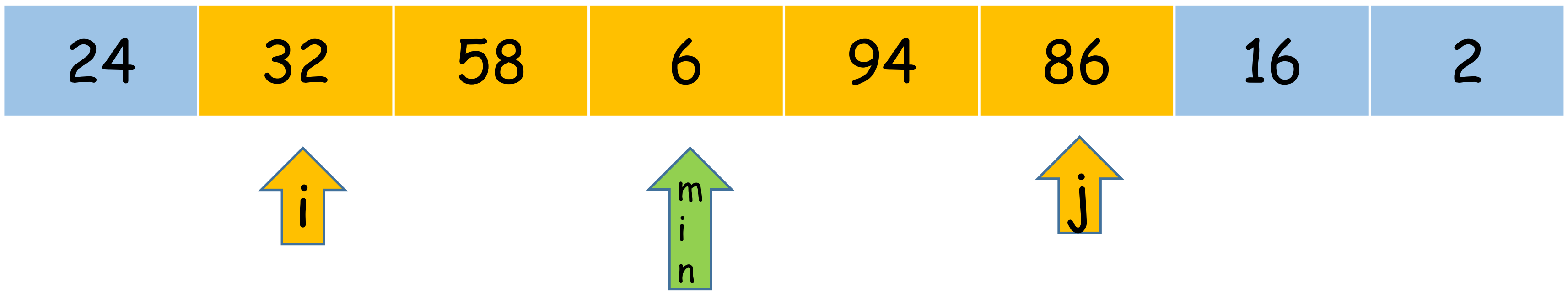
Output: Find the smallest element among A[i]…A[j]



RMQ Problem

Input: An array A and two integers i and j

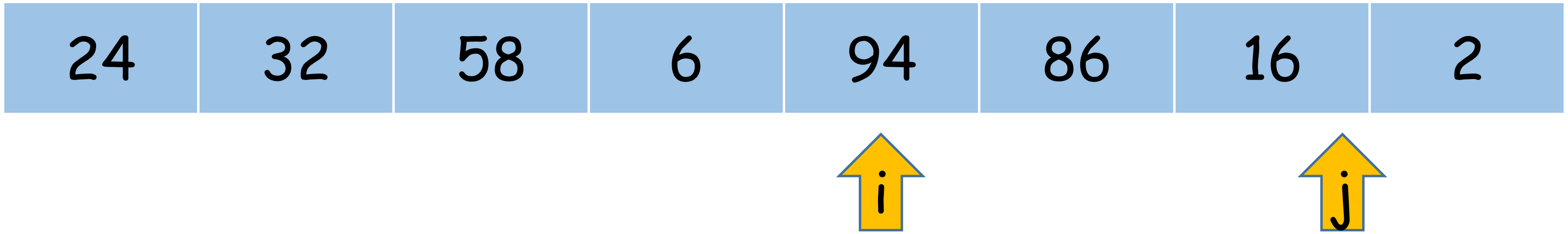
Output: Find the smallest element among A[i]…A[j]



RMQ Problem

Input: An array A and two integers i and j

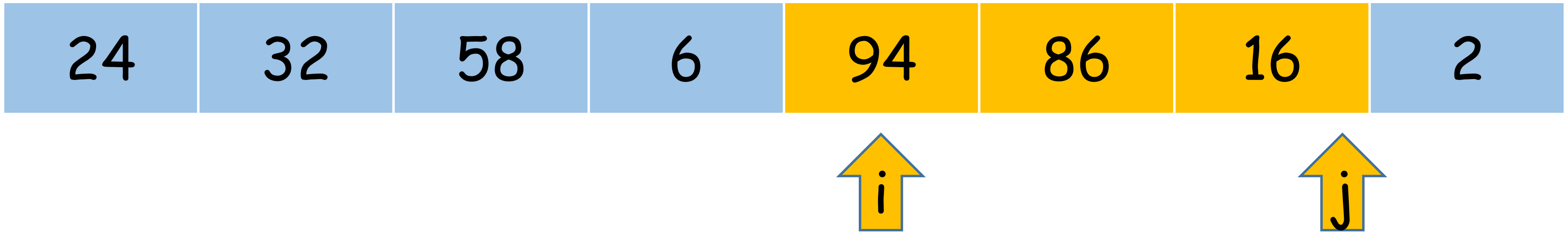
Output: Find the smallest element among A[i]…A[j]



RMQ Problem

<u>Input:</u> An array A and two integers i and j

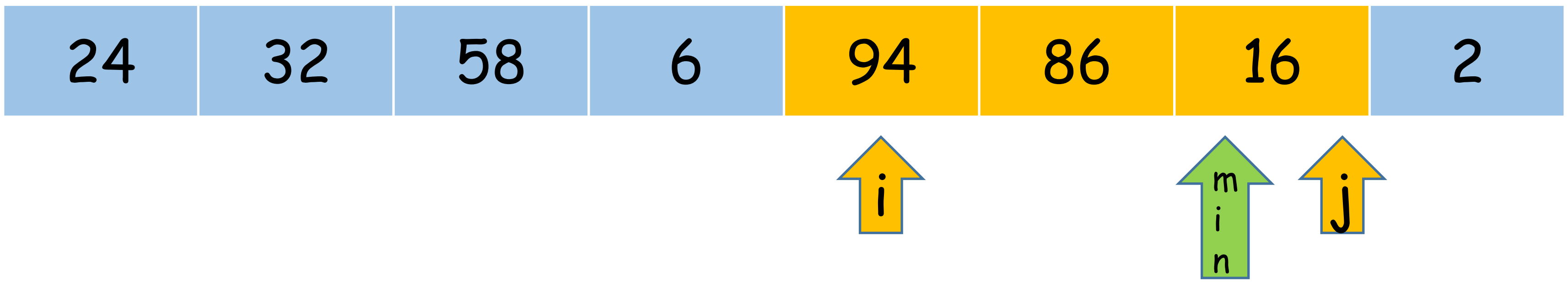
<u>Output:</u> Find the smallest element among A[i]…A[j]



RMQ Problem

Input: An array A and two integers i and j

Output: Find the smallest element among A[i]…A[j]



Straight forward solution

- Iterate from A[i] through A[j] and find the minimum
- Suppose A is fixed, and there will be many different queries
- Can we do better than the above O(n) solution?

How many distinct queries?

- i: 0 to n-1 and j: 0 to n-1
- Maximum number of distinct queries = n^2
- Preprocess these queries and store it in a table

Preprocess the all distinct queries

24	32	58	6
	i		j

	0	1	2	3
0				
1				6
2				
3				

Preprocess the all possible queries

- How to build the table?
- For each entry in the table, find the minimum over the range
- RT?
  - $O(n^3)$
- Is there a better way to build?

24	32	58	6

	0	1	2	3
0	24	24	24	6
1		32	32	6
2			58	6
3				6

Is there a better way to build the table?

- Yes
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

24	32	58	6

	0	1	2	3
0				
1				
2				
3				

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

24	32	58	6

	0	1	2	3
0	24	?		
1		32		
2			58	
3				6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

24	32	58	6

	0	1	2	3
0	24	?		
1		32		
2			58	
3				6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

24	32	58	6

	0	1	2	3
0	24	24		
1		32		
2			58	
3				6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

	0	1	2	3
0	24	24		
1		32	?	
2			58	
3				6

24	32	58	6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

24	32	58	6

	0	1	2	3
0	24	24		
1		32	?	
2			58	
3				6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

24	32	58	6

	0	1	2	3
0	24	24		
1		32	32	
2			58	
3				6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

	0	1	2	3
0	24	24	?	
1		32	32	
2			58	6
3				6

24	32	58	6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

	0	1	2	3
0	24	24	?	
1		32	32	
2			58	6
3				6

24	32	58	6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

24	32	58	6

	0	1	2	3
0	24	24	24	
1		32	32	
2			58	6
3				6

Is there a better way to build the table?

- Yes.
- In $O(n^2)$ using Dynamic Programming
- Start with the diagonal and then fill the adjacent entries of already filled entries

24	32	58	6

	0	1	2	3
0	24	24	24	6
1		32	32	6
2			58	6
3				6

So far we have…

- Two approaches
- Denote the complexity of an RMQ data structure by <p(n), q(n)>
  - p(n) is pre-processing time
  - q(n) is query time

- Our menu of structures for RMQ:
  - $\langle O(1), O(n)\rangle$ with no preprocessing
  - $\langle O(n^2), O(1)\rangle$ with preprocessing

- These are two ends of the spectrum
- Is there a trade-off or a better solution?

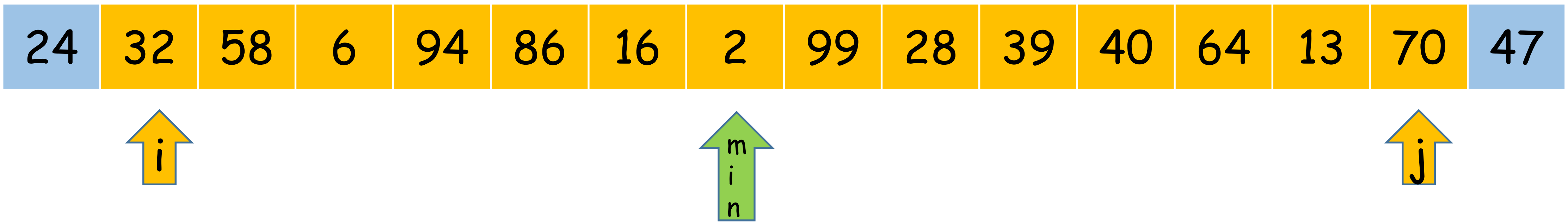
Look at the problem again…

24	32	58	6	94	86	16	2	99	28	39	40	64	13	70	47
	i													j	

Look at the problem again…

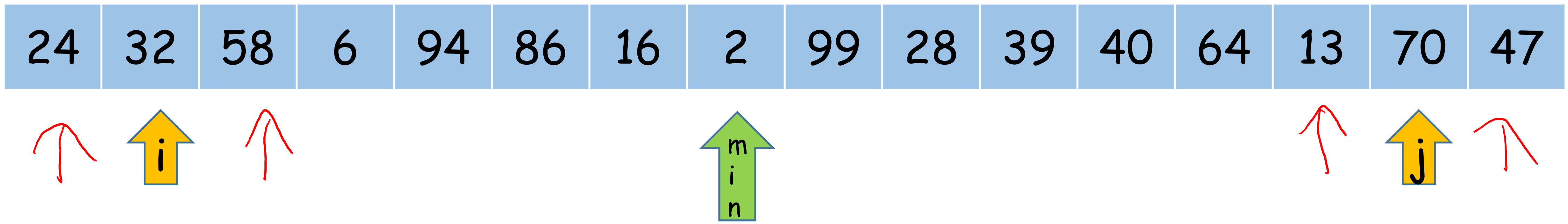
24	32	58	6	94	86	16	2	99	28	39	40	64	13	70	47
	i													j	

Look at the problem again…



Need another approach

We don't want to search all the elements…



Block partition approach

Partition the array into blocks of b elements

24	32	58	6	94	86	16	2	99	28	39	40	64	13	70	47

b = 4

Block partition approach

Partition the array into blocks of b elements

Find the min of each block and store in another array (size = n/b)

6				2				28				13			
6	32	58	24	94	86	16	2	99	28	39	40	64	47	70	13

b = 4

Block partition approach

Partition the array into blocks of b elements

Find the min of each block and store it an another array (size = n/b)

RMQ(i,j) = ?

6				2				28				13			
6	32	58	24	94	86	16	2	99	28	39	40	64	47	70	13
	i													j	

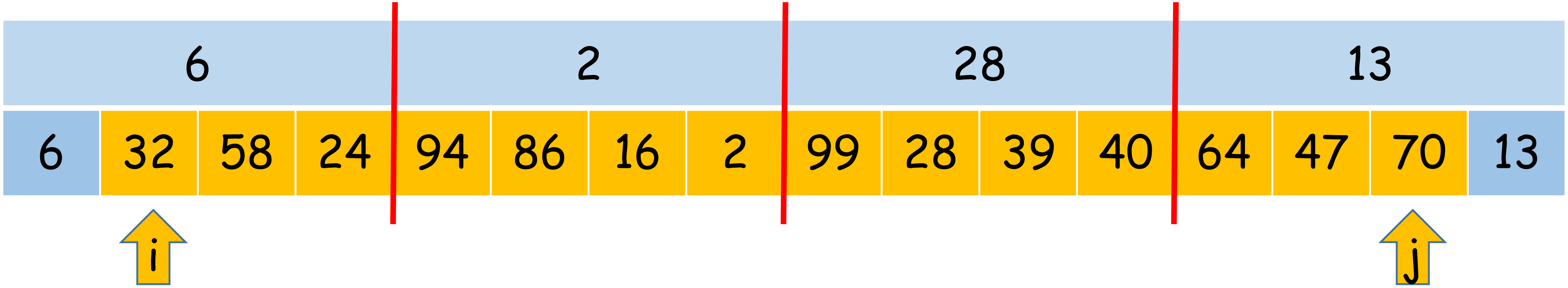
b = 4

Block partition approach

Partition the array into blocks of b elements

Find the min of each block and store it an another array (size = n/b)

RMQ(i,j) = ?



b = 4

Block partition approach

Partition the array into blocks of b elements

Find the min of each block and store it in another array (size = n/b)

RMQ(i,j) = ?

6				2				28				13			
6	32	58	24	94	86	16	2	99	28	39	40	64	47	70	13
	i													j	

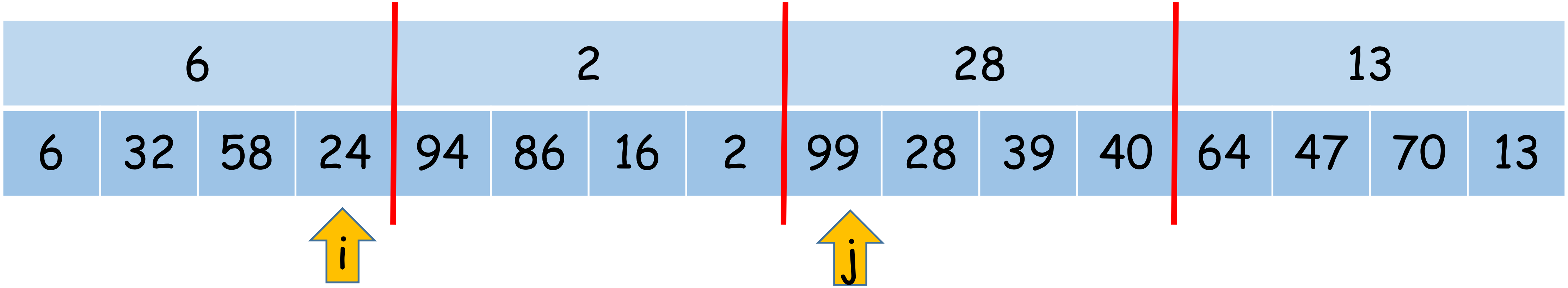
b = 4

Block partition approach

Partition the array into blocks of b elements

Find the min of each block and store it an another array (size = n/b)

RMQ(i,j) = ?



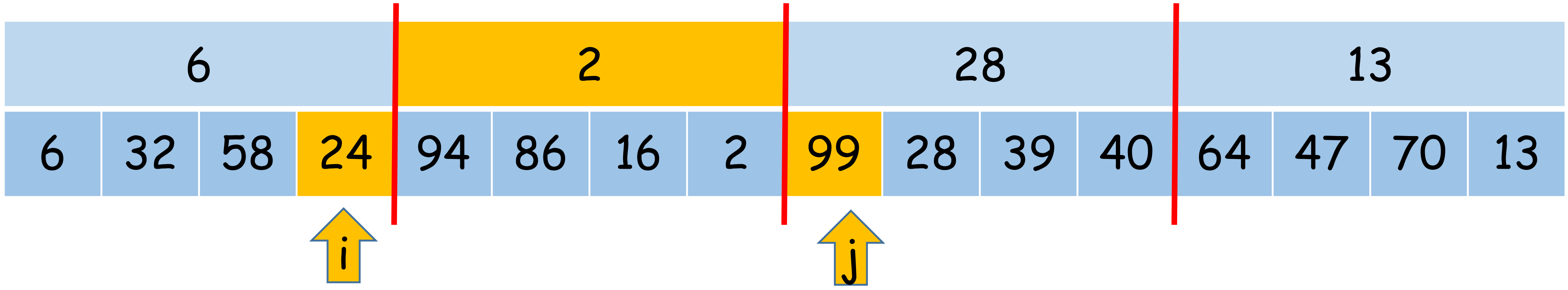
b = 4

Block partition approach

Partition the array into blocks of b elements

Find the min of each block and store it an another array (size = n/b)

RMQ(i,j) = ?



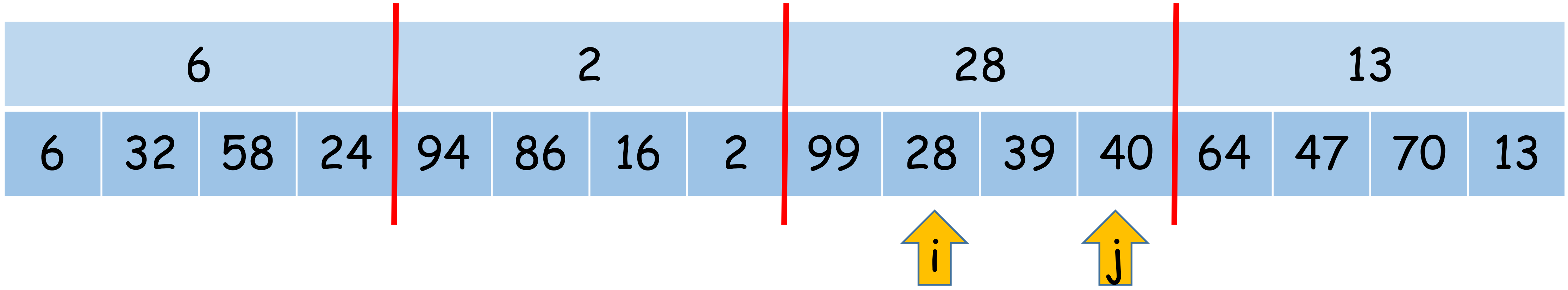
b = 4

Block partition approach

Partition the array into blocks of b elements

Find the min of each block and store it an another array (size = n/b)

RMQ(i,j) = ?



b = 4

Block partition approach

Partition the array into blocks of b elements

Find the min of each block and store it an another array (size = n/b)

RMQ(i,j) = ?

6				2				28				13			
6	32	58	24	94	86	16	2	99	28	39	40	64	47	70	13
									i		j				

b = 4

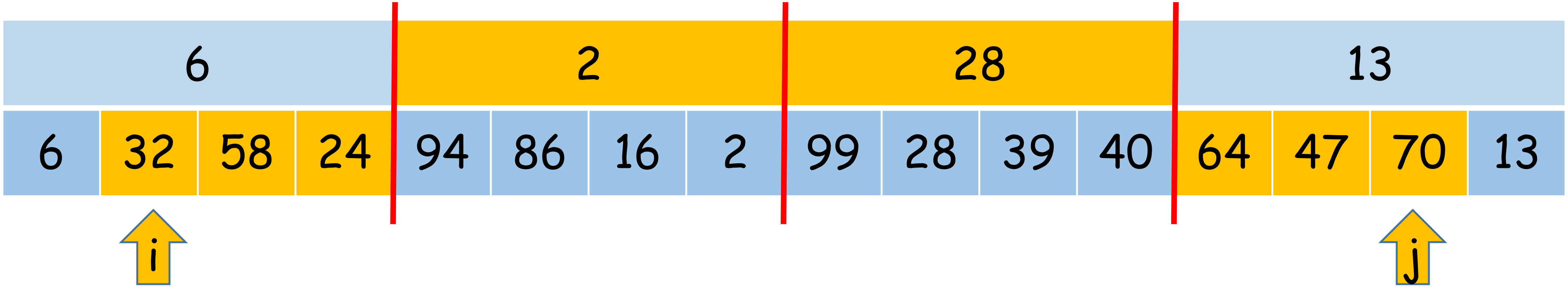
Analysis of block partition approach

Preprocessing time p(n):

- O(b) to find minimum on each of (n/b) blocks
- $p(n) = O(n)$

Query time q(n)

- O(1) to find blocks of i and j
- O(b) to scan inside blocks of i and j
- O(n/b) to scan minima of each block between blocks of i and j
- $q(n) = O(b + n/b)$



b = 4

Analyze query time $O(b + n/b)$

- If $b = 1$ or $b = n$, then no preprocessing requires
- Choose b to minimize $b + n/b$
- Optimal value of $b = \sqrt{n}$
- $q(n) = O(\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$

Our Menu

- No preprocessing: $\langle O(1), O(n)\rangle$
- Block partition: $\langle O(n), O(\sqrt{n})\rangle$
- Full preprocessing: $\langle O(n^2), O(1)\rangle$
- Can we add something better?

Revisit preprocessing full table approach

- Is it necessary to preprocess all the ranges of the given array?
- Goal: preprocess less number of ranges yet query in $O(1)$ time

Revisit preprocessing full table

- Is it necessary to preprocess all the ranges of the given array?

24	32	58	6	94	86	16	20

	0	1	2	3	4	5	6	7
0	24	24	24	6	6	6	6	6
1		32	32	6	6	6	6	6
2			58	6	6	6	6	6
3				6	6	6	6	6
4					94	86	16	16
5						86	16	16
6							16	16
7								20

Revisit preprocessing full table

- Is it necessary to preprocess all the ranges of the given array?

24	32	58	6	94	86	16	20

	0	1	2	3	4	5	6	7
0	24	24	24	6	6	6	6	6
1		32	32	6	6	6	~~6~~	?
2			58	6	6	6	6	~~6~~
3				6	6	6	6	6
4					94	86	16	16
5						86	16	16
6							16	16
7								20

Revisit preprocessing full table

- Is it necessary to preprocess all the ranges of the given array?

24	32	58	6	94	86	16	20

	0	1	2	3	4	5	6	7
0	24	24	24	6				6
1		32	32	6	6			?
2			58	6	6	6		
3				6	6	6	6	
4					94	86	16	16
5						86	16	16
6							16	16
7								20

Revisit preprocessing full table

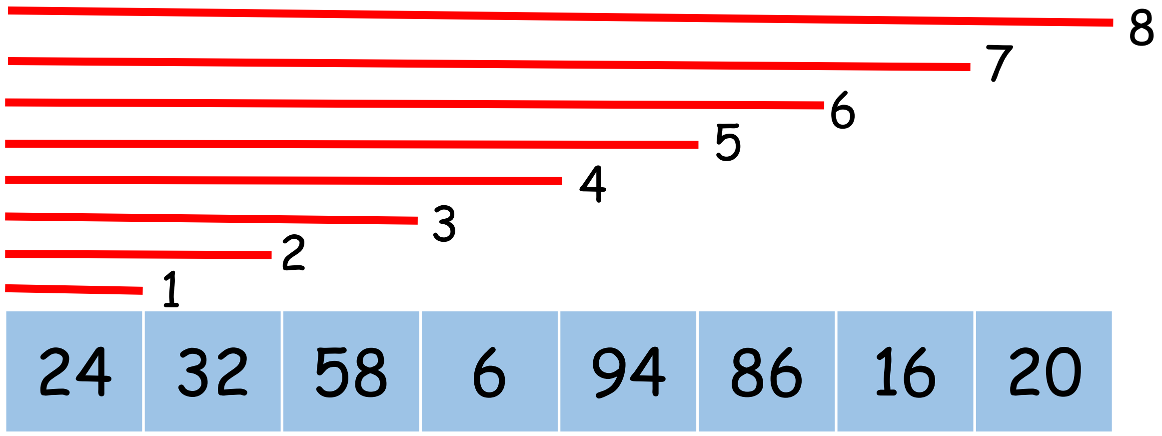
- Is it necessary to preprocess all the ranges of the given array?

24	32	58	6	94	86	16	20

	0	1	2	3	4	5	6	7
0	24	24	24	6				6
1		32	32	6	6			?
2			58	6	6	6		
3				6	6	6	6	
4					94	86	16	16
5						86	16	16
6							16	16
7								20

Revisit preprocessing full table

- Is it necessary to preprocess all the ranges of the given array?
- Which ranges to skip?
- Need to skip enough numbers to get an asymptotically lower bound than $O(n^2)$

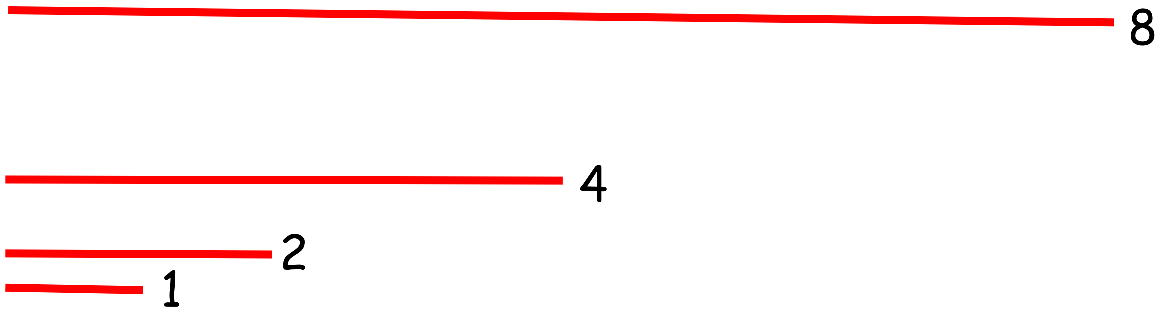


	0	1	2	3	4	5	6	7
0	24	24	24	6	6	6	6	6
1		32	32	6	6	6	6	6
2			58	6	6	6	6	6
3				6	6	6	6	6
4					94	86	16	16
5						86	16	16
6							16	16
7								20

Revisit preprocessing full table

- Is it necessary to preprocess all the ranges of the given array?
- Which ranges to skip?
- Need to skip enough numbers to get an asymptotically lower bound than $O(n^2)$

Big Idea:
For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$ starting from that index



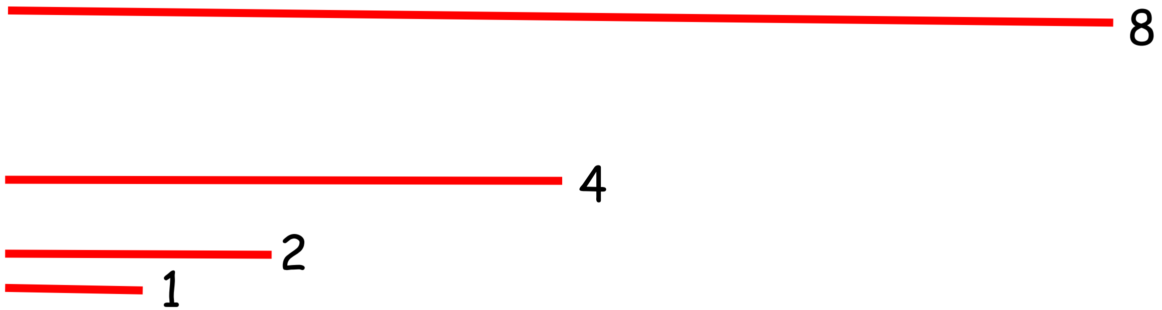
24	32	58	6	94	86	16	20

	0	1	2	3	4	5	6	7
0	24	24	24	6	6	6	6	6
1		32	32	6	6	6	6	6
2			58	6	6	6	6	6
3				6	6	6	6	6
4					94	86	16	16
5						86	16	16
6							16	16
7								20

Revisit preprocessing full table

- Is it necessary to preprocess all the ranges of the given array?
- Which ranges to skip?
- Need to skip enough numbers to get an asymptotically lower bound than $O(n^2)$

Big Idea:
For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$ starting from that index



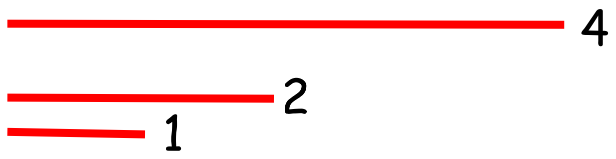
24	32	58	6	94	86	16	20

	0	1	2	3	4	5	6	7
0	24	24		6				6
1		32	32	6	6	6	6	6
2			58	6	6	6	6	6
3				6	6	6	6	6
4					94	86	16	16
5						86	16	16
6							16	16
7								20

Revisit preprocessing full table

- Is it necessary to preprocess all the ranges of the given array?
- Which ranges to skip?
- Need to skip enough numbers to get an asymptotically lower bound than $O(n^2)$

Big Idea:
For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$ starting from that index

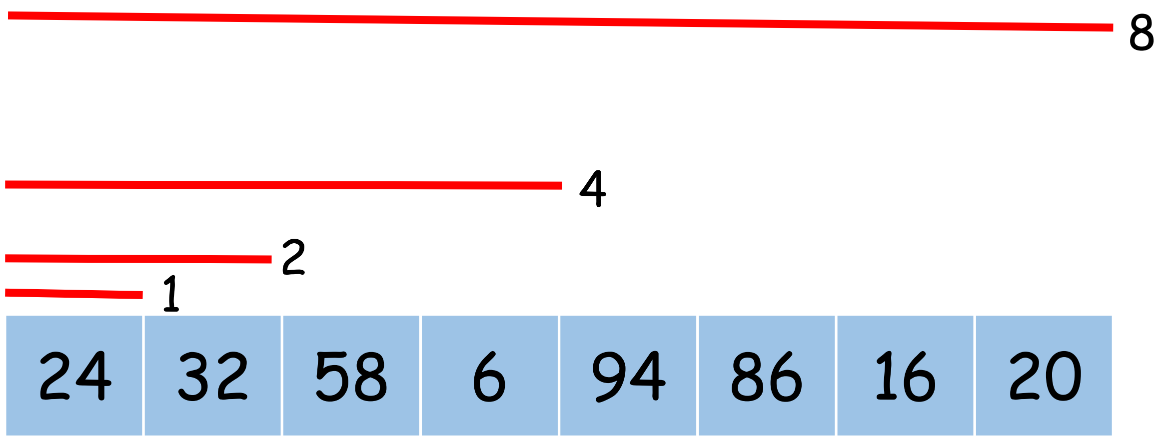


24	32	58	6	94	86	16	20

	0	1	2	3	4	5	6	7
0	24	24		6				6
1		32	32		6			
2			58	6		6		
3				6	6		6	
4					94	86		16
5						86	16	
6							16	16
7								20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \dots, 2^k$



Range size

Range start index	2^0	2^1	2^2	2^3
0				
1				
2				
3				
4				
5				
6				
7				

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \dots, 2^k$
- Fill table using DP

Range size

Range start index	2^0	2^1	2^2	2^3
0	24			
1	32			
2	58			
3	6			
4	94			
5	86			
6	16			
7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \dots, 2^k$

- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index		2^0	2^1	2^2	2^3
	0	24	?		
	1	32			
	2	58			
	3	6			
	4	94			
	5	86			
	6	16			
	7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \dots, 2^k$
- Fill table using DP

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	?		
1	32			
2	58			
3	6			
4	94			
5	86			
6	16			
7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24		
1	32			
2	58			
3	6			
4	94			
5	86			
6	16			
7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24		
1	32	?		
2	58			
3	6			
4	94			
5	86			
6	16			
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index		2^0	2^1	2^2	2^3
	0	24	24		
	1	32	?		
	2	58			
	3	6			
	4	94			
	5	86			
	6	16			
	7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24		
1	32	32		
2	58			
3	6			
4	94			
5	86			
6	16			
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24		
1	32	32		
2	58	6		
3	6			
4	94			
5	86			
6	16			
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \dots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24		
1	32	32		
2	58	6		
3	6	6		
4	94	86		
5	86	16		
6	16	16		
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	?	
1	32	32		
2	58	6		
3	6	6		
4	94	86		
5	86	16		
6	16	16		
7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

3

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	?	
1	32	32		
2	58	6		
3	6	6		
4	94	86		
5	86	16		
6	16	16		
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

Range size

Range start index		2^0	2^1	2^2	2^3
	0	24	24	?	
	1	32	32		
	2	58	6		
	3	6	6		
	4	94	86		
	5	86	16		
	6	16	16		
	7	20			

(0, 1) (2, 3)

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

Range start index	Range size 2^0	2^1	2^2	2^3
0	24	24	6	
1	32	32		
2	58	6		
3	6	6		
4	94	86		
5	86	16		
6	16	16		
7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	6	
1	32	32	?	
2	58	6		
3	6	6		
4	94	86		
5	86	16		
6	16	16		
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	6	
1	32	32	?	
2	58	6		
3	6	6		
4	94	86		
5	86	16		
6	16	16		
7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	6	
1	32	32	?	
2	58	6		
3	6	6		
4	94	86		
5	86	16		
6	16	16		
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

Range size

Range start index		2^0	2^1	2^2	2^3
	0	24	24	6	
	1	32	32	6	
	2	58	6		
	3	6	6		
	4	94	86		
	5	86	16		
	6	16	16		
	7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	6	
1	32	32	6	
2	58	6	6	
3	6	6	6	
4	94	86	16	
5	86	16		
6	16	16		
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

Range size

Range start index		2^0	2^1	2^2	2^3
	0	24	24	6	?
	1	32	32	6	
	2	58	6	6	
	3	6	6	6	
	4	94	86	16	
	5	86	16		
	6	16	16		
	7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

24	32	58	6	94	86	16	20

Range size

Range start index

	2^0	2^1	2^2	2^3
0	24	24	6	?
1	32	32	6	
2	58	6	6	
3	6	6	6	
4	94	86	16	
5	86	16		
6	16	16		
7	20			

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \dots, 2^k$
- Fill table using DP

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	6	?
1	32	32	6	
2	58	6	6	
3	6	6	6	
4	94	86	16	
5	86	16		
6	16	16		
7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	6	6
1	32	32	6	
2	58	6	6	
3	6	6	6	
4	94	86	16	
5	86	16		
6	16	16		
7	20			

24	32	58	6	94	86	16	20

Preprocess sparse table

- For every index preprocess ranges of size $2^0, 2^1, \ldots, 2^k$
- Fill table using DP
- $p(n) = O(n \log n)$

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	6	6
1	32	32	6	
2	58	6	6	
3	6	6	6	
4	94	86	16	
5	86	16		
6	16	16		
7	20			

24	32	58	6	94	86	16	20

Querying

- RMQ(2, 7)

24	32	58	6	94	86	16	20

	Range size			
Range start index	2^0	2^1	2^2	2^3
0	24	24	6	6
1	32	32	6	
2	58	6	6	
3	6	6	6	
4	94	86	16	
5	86	16		
6	16	16		
7	20			

Querying

- RMQ(2, 7)

24	32	58	6	94	86	16	20

Range size

Range start index		2^0	2^1	2^2	2^3
	0	24	24	6	6
	1	32	32	6	
	2	58	6	6	
	3	6	6	6	
	4	94	86	16	
	5	86	16		
	6	16	16		
	7	20			

Querying

- RMQ(2, 7) = min(6, 16) = 6

24	32	58	6	94	86	16	20

Range size

Range start index	2^0	2^1	2^2	2^3
0	24	24	6	6
1	32	32	6	
2	58	6	6	
3	6	6	6	
4	94	86	16	
5	86	16		
6	16	16		
7	20			

Querying

$[2, 5]$ $k = 2$ $2^k \le 7 - 2 + 1$

$[2, 2 + 2^2 - 1]$ $2^k \le 6$

- RMQ(i, j)
  - Find largest k such that $2^k \le j - i + 1$
  - Range [i, j] can be formed with two overlapping ranges $[i, i + 2^k - 1]$ and $[j - 2^k + 1, j]$
  - Look up the values of these two ranges in the sparse table and find min
  - $q(n) = O(1)$

$[7 - 4 + 1, 7]$

$[4, 7]$

$i, i + 2^k - 1$

24	32	58	6	94	86	16	20

2 7

Range size

	2^0	2^1	2^2	2^3
0	24	24	6	6
1	32	32	6	
2	58	6	6	
3	6	6	6	
4	94	86	16	
5	86	16		
6	16	16		
7	20			

Our Menu

Approaches to RMQ(i, j)

- Full preprocessing: $\langle O(n^2), O(1)\rangle$
- Sparse table: $\langle O(n \log n), O(1)\rangle$
- Block partition: $\langle O(n), O(\sqrt{n})\rangle$
- No preprocessing: $\langle O(1), O(n)\rangle$

- Can we add something better?
  - Yes. Hybrid strategies

Block partition revisited

RMQ(i,j) = ?

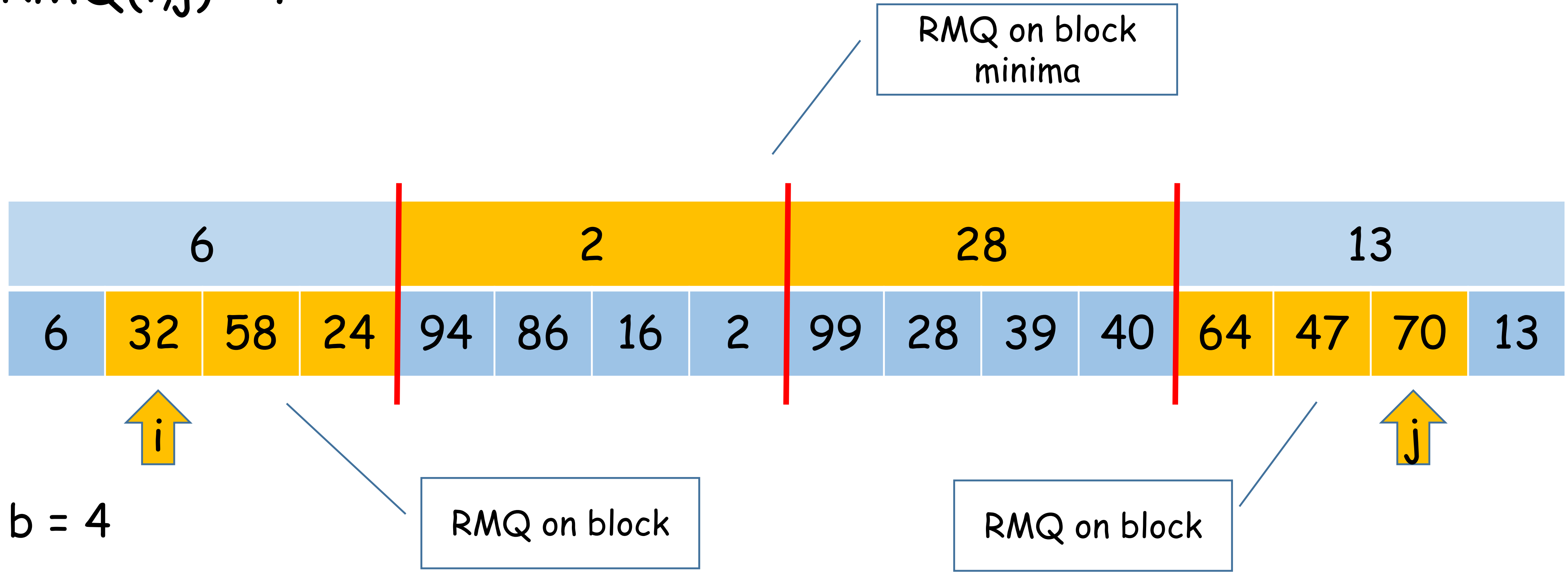
6	2	28	13

6	32	58	24	94	86	16	2	99	28	39	40	64	47	70	13
	i													j	

b = 4

Block partition revisited

RMQ(i,j) = ?



b = 4

Hybrid Approach

Split the given array into n/b blocks of size b

6	32	58	24	94	86	16	2	99	28	39	40	64	47	70	13

Hybrid Approach

Split the given array into n/b blocks of size b

Build an array of block minima

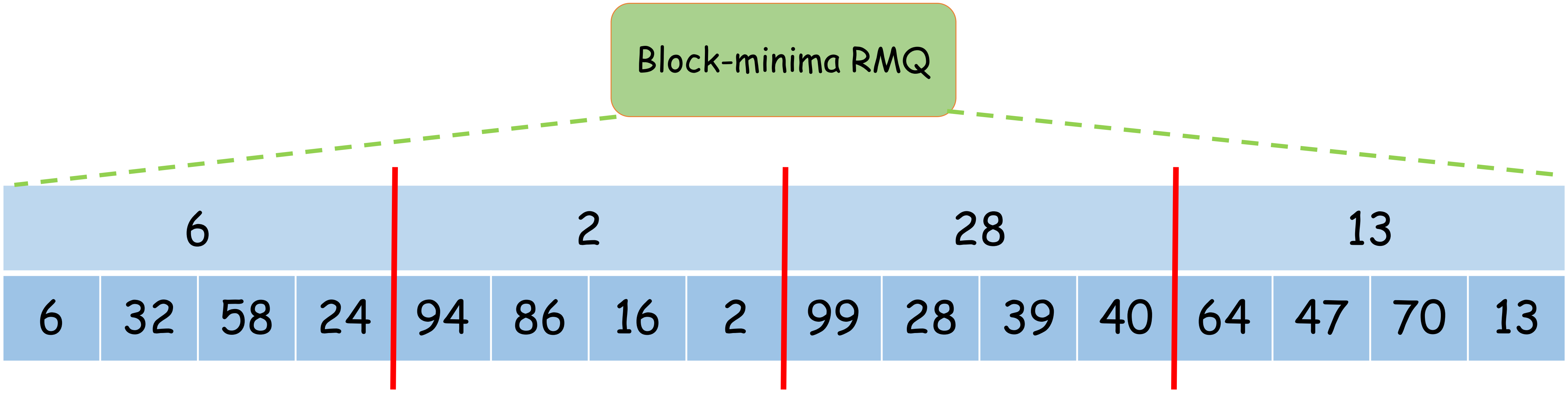
6				2				28				13			
6	32	58	24	94	86	16	2	99	28	39	40	64	47	70	13

Hybrid Approach

Split the given array into n/b blocks of size b

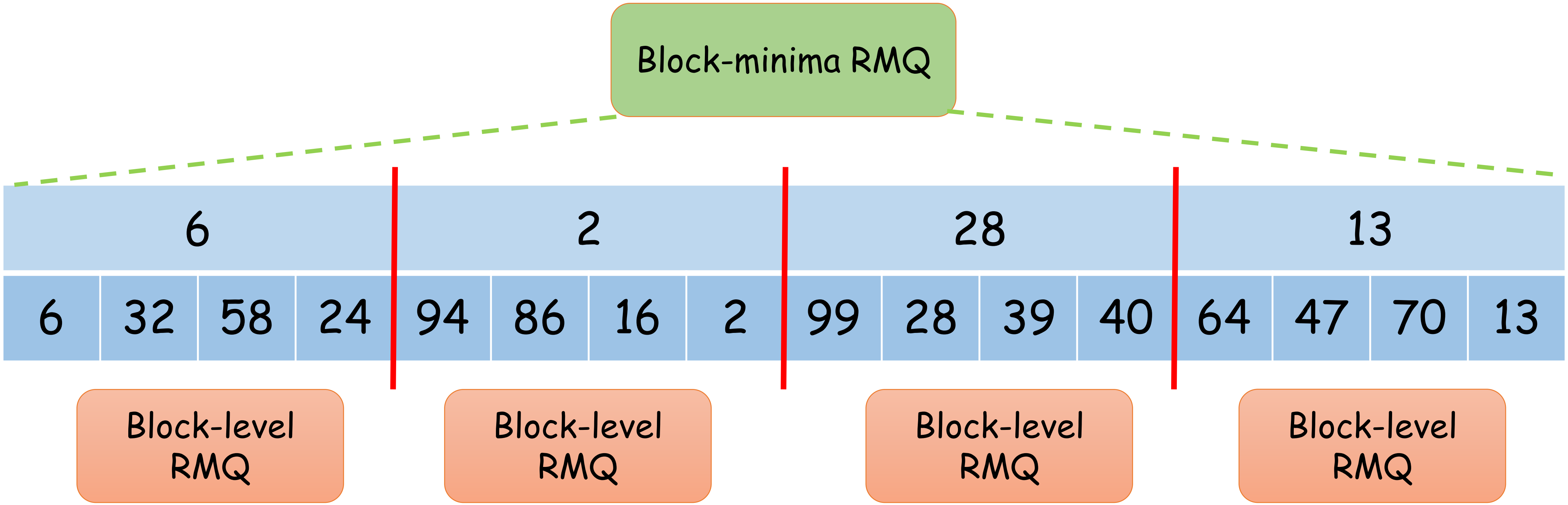
Build an array of block minima

Build block-minima RMQ structure over the array of block minima



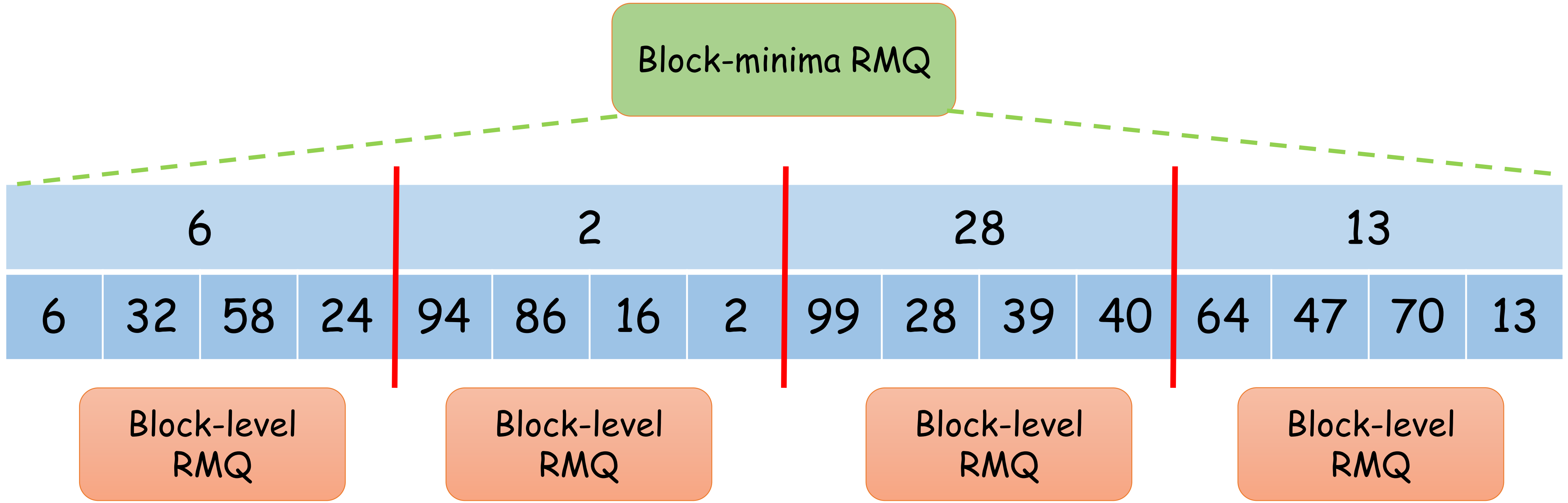
Hybrid Approach

Split the given array into n/b blocks of size b
Build an array of block minima
Build an RMQ structure over the array of block minima
Build an RMQ structure for each block



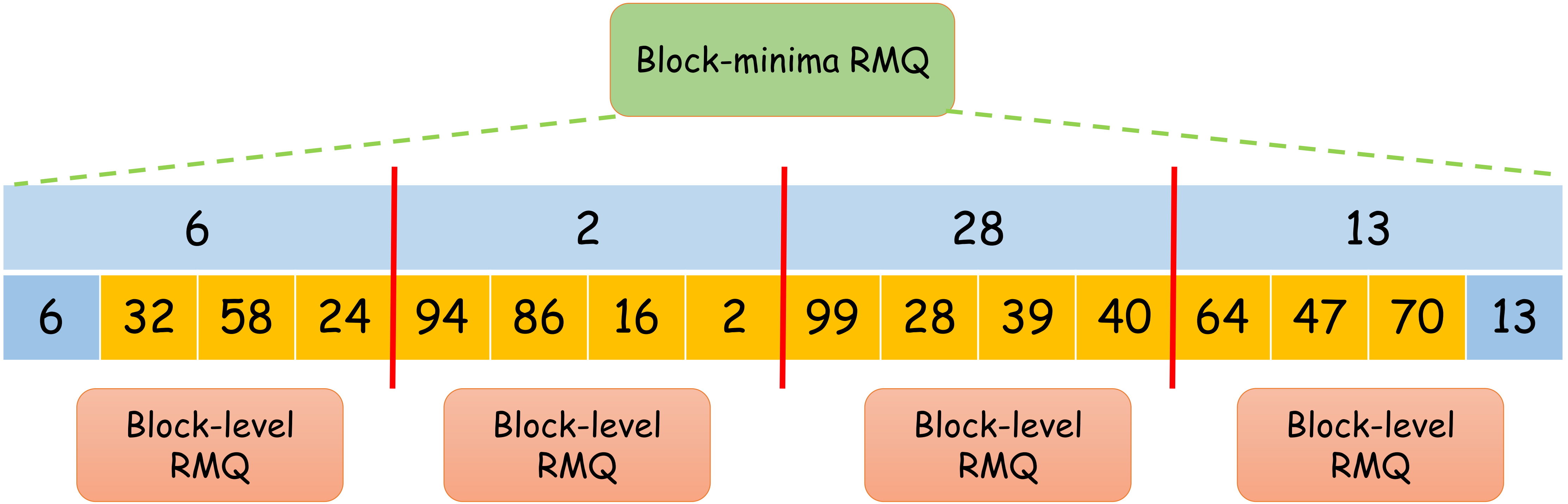
Hybrid Approach

Split the given array into n/b blocks of size b
Build an array of block minima
Build an RMQ structure over the array of block minima
Build an RMQ structure for each block
Find min of results of all RMQ



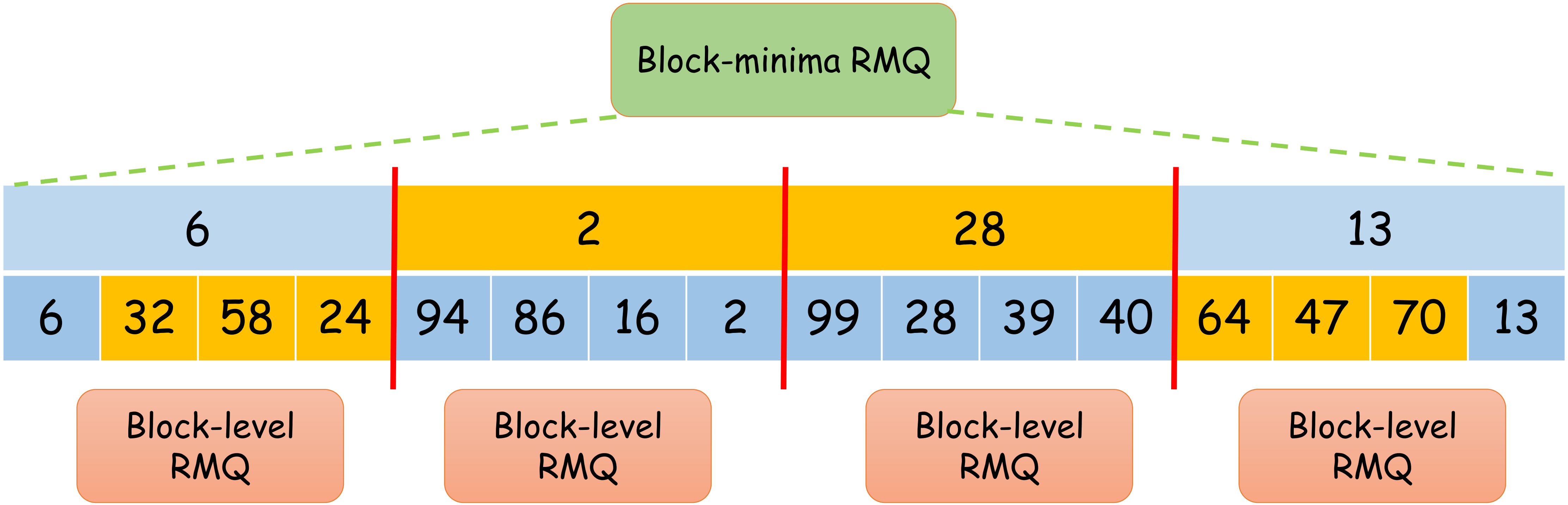
Hybrid Approach

Split the given array into n/b blocks of size b
Build an array of block minima
Build an RMQ structure over the array of block minima
Build an RMQ structure for each block
Find min of results of all RMQ



Hybrid Approach

Split the given array into n/b blocks of size b
Build an array of block minima
Build an RMQ structure over the array of block minima
Build an RMQ structure for each block
Find min of results of all RMQ

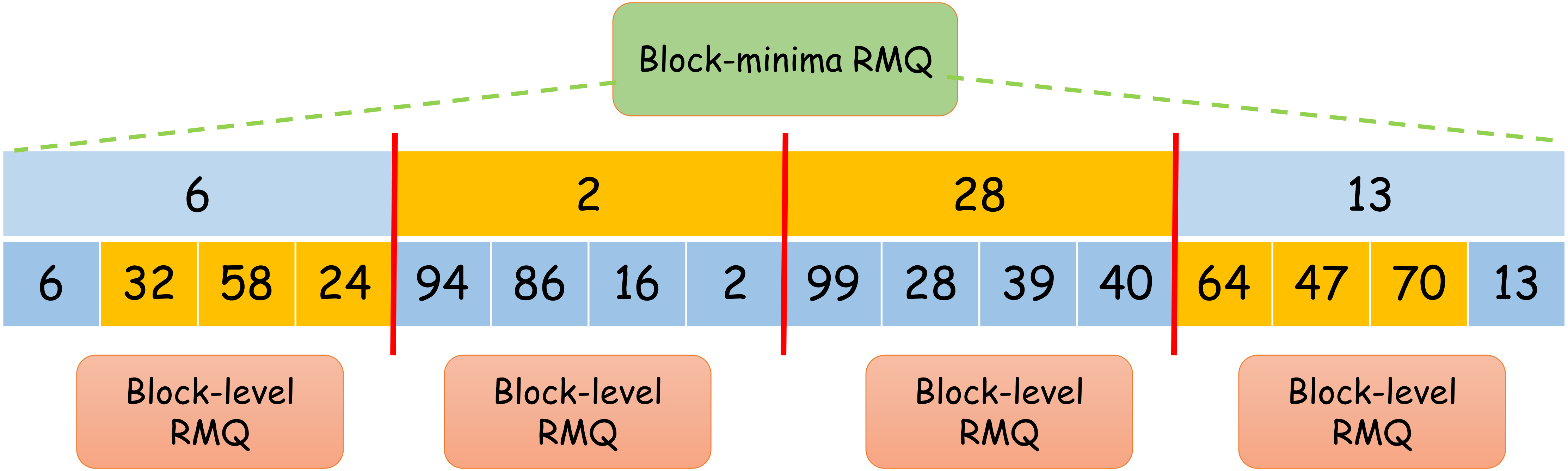


Analysis

- Assume that $\langle p_1(n), q_1(n)\rangle$ time RMQ structure is used for block minima and $\langle p_2(n), q_2(n)\rangle$ time RMQ structure is used at the block level

- Preprocessing time $p(n)$ for this hybrid structure includes:
  - $O(n)$ time to construct array of block minima
  - $O(p_1(n/b))$ time to construct block-minima RMQ structure
  - $O((n/b)\, p_2(b))$ time to construct block-level RMQ structures

- $p(n) = O(\, n + p_1(n/b) + (n/b)\, p_2(b)\,)$

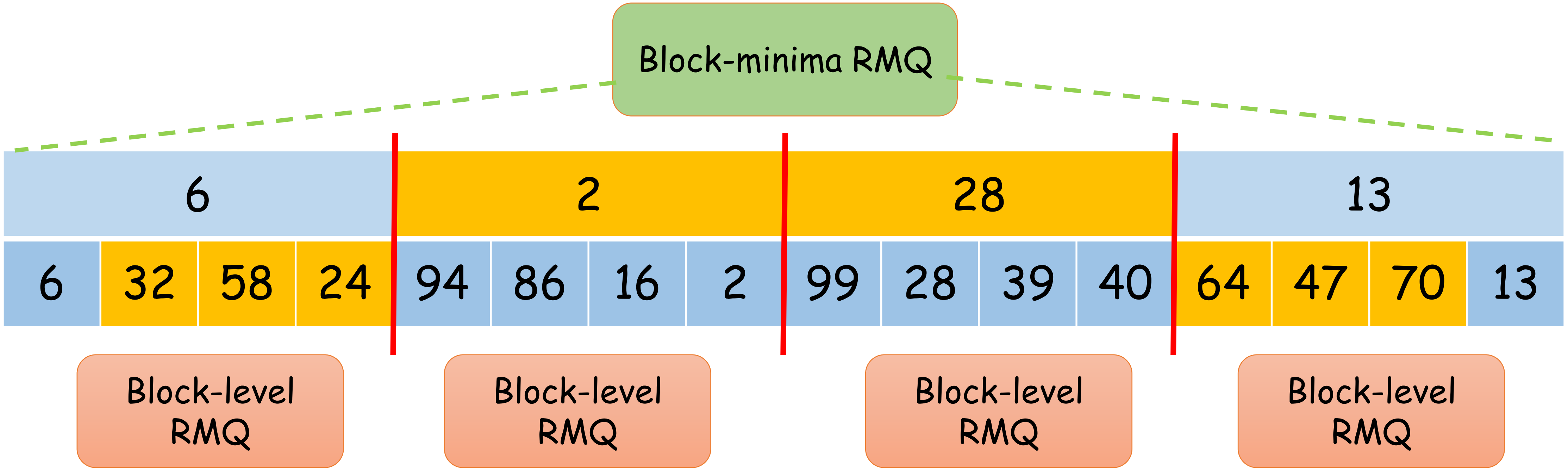


Analysis

- Assume that $\langle p_1(n), q_1(n)\rangle$ time RMQ is used for block minima and $\langle p_2(n), q_2(n)\rangle$ time RMQ is used at the block level

- Query time $q(n)$ for this hybrid structure includes:
  - $O(q_1(n/b))$ time to query block-minima RMQ structure
  - $O(q_2(b))$ time to query block-level RMQ structures

- $q(n) = O(\, q_1(n/b) + q_2(b)\,)$



Summary

- Preprocessing time $p(n) = O(\ n + p_1(n/b) + (n/b\ )\ p_2(b)\ )$

- Query time $q(n) = O(q_1(n/b)) + O(q_2(b))$

- Several choices of RMQ structure

- Time complexity depends on the RMQ structures chosen

Hybrid Approach One

- Preprocessing time $p(n) = O(n + p_1(n/b)) + (n/b\ )\ p_2(b))$
- Query time $q(n) = O(q_1(n/b)) + q_2(b))$
- Use <O(n log n), O(1)> sparse table RMQ structure for block minima
- Time to construct sparse table over block minima is $O((n/b) \log (n/b))$
- Choose $b = \log (n)$
  - $O((n/b) \log(n/b)) = O((n/\log n) \log (n/\log n)) = O((n/\log n) \log n) = O(n)$
- Use no preprocessing <O(1), O(n)> for block-level RMQ structure
- $p(n) = O(n + n + n/\log n) = O(n)$
- $q(n) = O(1 + \log n) = O(\log n)$
- A <O(n), O(log n)> solution

Hybrid Approach two

- Preprocessing time $p(n) = O(n + p_1(n/b) + (n/b\ )\ p_2(b))$
- Query time $q(n) = O(q_1(n/b) + q_2(b))$
- Use <O(n log n), O(1)> sparse table RMQ structure for both block-minima and block-level RMQ structures and block size b = log n
- Time to construct sparse table over block minima is $O((n/b) \log (n/b))$

  $O((n/b) \log(n/b)) = O((n/\log n) \log (n/\log n)) = O((n/\log n) \log n) = O(n)$
- Time to construct all block-level RMQ structures

  $O((n/\log n) (\log n \log \log n) = O(n \log \log n)$
- $p(n) = O(n + n + n \log \log n) = O(n \log \log n)$
- $q(n) = O(1 + 1) = O(1)$
- A <O(n log log n), O(1)> solution

Hybrid Approach three

- Preprocessing time $p(n) = O(n + p_1(n/b) + (n/b\ )\ p_2(b))$
- Query time $q(n) = O(q_1(n/b) + q_2(b))$
- Use $\langle O(n \log n), O(1)\rangle$ sparse table RMQ structure for block-minima
- Use hybrid two $\langle O(n \log \log n), O(1)\rangle$ RMQ structure for block-level RMQ
- Block size $b = \log n$
- A $\langle O(n), O(\log \log n)\rangle$ solution

Our Menu

Approaches to RMQ(i, j)

- Full preprocessing: $\langle O(n^2), O(1) \rangle$
- Sparse table: $\langle O(n \log n), O(1) \rangle$
- Block partition: $\langle O(n), O(\sqrt{n}) \rangle$
- No preprocessing: $\langle O(1), O(n) \rangle$
- Hybrid one: $\langle O(n), O(\log n) \rangle$
- Hybrid two: $\langle O(n \log \log n), O(1) \rangle$
- Hybrid three: $\langle O(n), O(\log \log n) \rangle$

Is there a $\langle O(n), O(1) \rangle$ solution?

How to get <O(n), O(1)> complexity?

- Relook at the hybrid structure complexity

- Preprocessing time $p(n) = O((n) + p_1(n/b) + (n/b)\, p_2(b))$

- Query time $q(n) = O(q_1(n/b) + q_2(b))$

This term needs to be O(n)

This term needs to be O(1)

Use sparse table for RMQ on block minima

This term becomes O(1)

This term becomes O(n)

Sparse table: <O(n log n), O(1)>

A hunch

We need to reduce the number of bottom RMQ structures

If there are blocks that are "similar" =>

need to construct only one RMQ structures for these blocks

24	32	58	6

49	65	78	12

A hunch

We need to reduce the number of bottom RMQ structures

If there are blocks that are "similar" =>

need to construct only one RMQ structures for these blocks

24	32	58	6

min

49	65	78	12

min

A hunch

We need to reduce the number of bottom RMQ structures

If there are blocks that are “similar” =>

need to construct only one RMQ structures for these blocks

A hunch

We need to reduce the number of bottom RMQ structures

If there are blocks that are "similar" =>

need to construct only one RMQ structures for these blocks

A new definition for RMQ

RMQ(i, j) is the index of the minimum value in the range i through j

24	32	58	6

49	65	78	12

A new definition for RMQ

RMQ(i, j) is the index of the minimum value in the range i through j

24	32	58	6

	0	1	2	3
0	24	24	24	6
1		32	32	6
2			58	6
3				6

Lookup tables based on the old definition

49	65	78	12

	0	1	2	3
0	49	49	49	12
1		65	65	12
2			78	12
3				12

A new definition for RMQ

RMQ(i, j) is the index of the minimum value in the range i through j

24	32	58	6

	0	1	2	3
0	0	0	0	3
1		1	1	3
2			2	3
3				3

Lookup tables based on the new definition

identical

49	65	78	12

	0	1	2	3
0	0	0	0	3
1		1	1	3
2			2	3
3				3

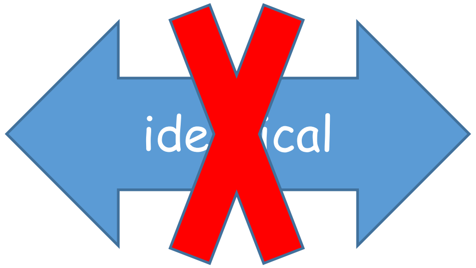
A new definition for RMQ

RMQ(i, j) is the index of the minimum value in the range i through j

24	32	58	6

	0	1	2	3
0	0	0	0	3
1		1	1	3
2			2	3
3				3

Lookup tables based on the new definition



47	45	68	52

	0	1	2	3
0	0	1	1	1
1		1	1	1
2			2	3
3				3

One more definition

Two blocks X and Y are of the same type, denoted as X ~ Y,

iff

$$RMQ_X(i, j) = RMQ_Y(i, j) \text{ where } 0 \le i \le j < b$$

Takeaway
If two blocks are of the same type, then only one RMQ structure need to be precomputed

How to efficiently find that X ~ Y?

How many types are there?

Properties of same type blocks

Property 1: If X ~ Y, then minimum of both X and Y must occur at the same position

24	32	5	16	84

14	12	80	120	124
149	165	78	120	124
49	56	10	20	14
14	12	8	120	124

Properties of same type blocks

Property 1: If X ~ Y, then minimum of both X and Y must occur at the same position

24	32	5	16	84

14	12	80	120	124
149	165	78	120	124
49	56	10	20	14
14	12	8	120	124

Properties of same type blocks

Property 1: If X ~ Y, then minimum of both X and Y must occur at the same position

Property 2: The above property must hold true for both the subarrays to the left and right of the minimum

24	32	5	16	84

149	165	78	120	124
49	56	10	20	14
14	12	8	120	124

Properties of same type blocks

Property 1: If X ~ Y, then minimum of both X and Y must occur at the same position

Property 2: The above property must hold true for both the subarrays to the left and right of the minimum

24	32	5	16	84

149	165	78	120	124
49	56	10	20	14
14	12	8	120	124

Properties of same type blocks

Property 1: If X ~ Y, then minimum of both X and Y must occur at the same position

Property 2: The above property must hold true for both the subarrays to the left and right of the minimum

24	32	5	16	84

149	165	78	120	124
49	56	10	20	14

Properties of same type blocks

Property 1: If X ~ Y, then minimum of both X and Y must occur at the same position

Property 2: The above property must hold true for both the subarrays to the left and right of the minimum

24	32	5	16	84

149	165	78	120	124
49	56	10	20	14

Properties of same type blocks

Property 1: If X ~ Y, then minimum of both X and Y must occur at the same position

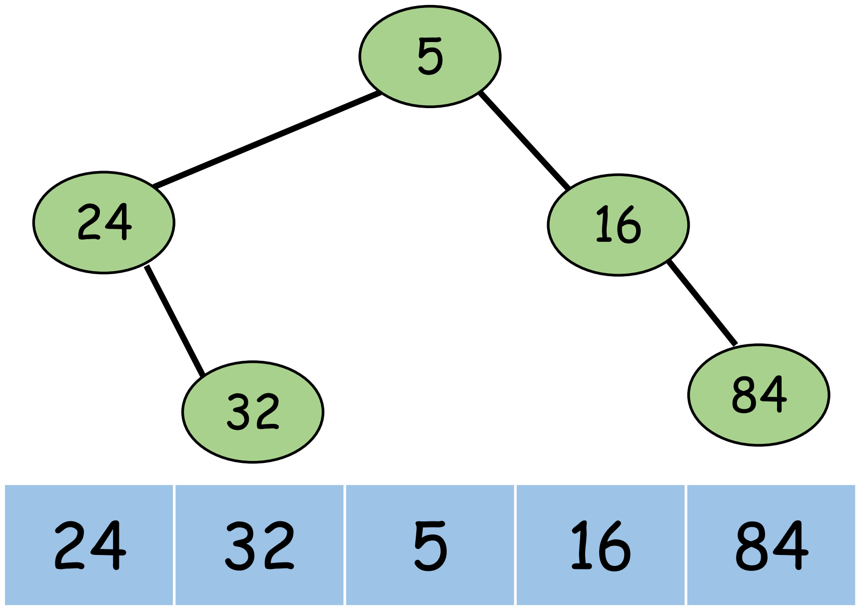
Property 2: The above property must hold true for both the subarrays to the left and right of the minimum

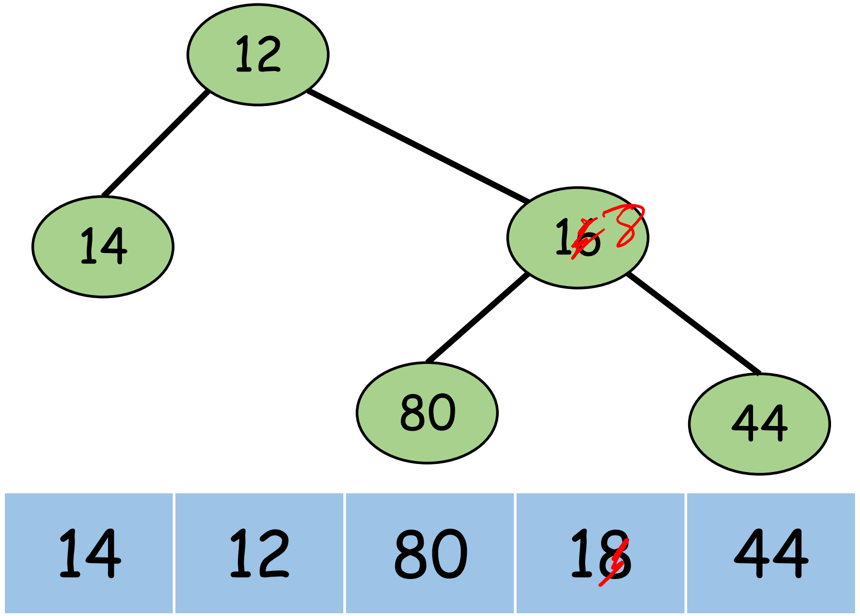
24	32	5	16	84	~	149	165	78	120	124

Cartesian Tree

A Cartesian tree of an array is a binary tree defined as follows:

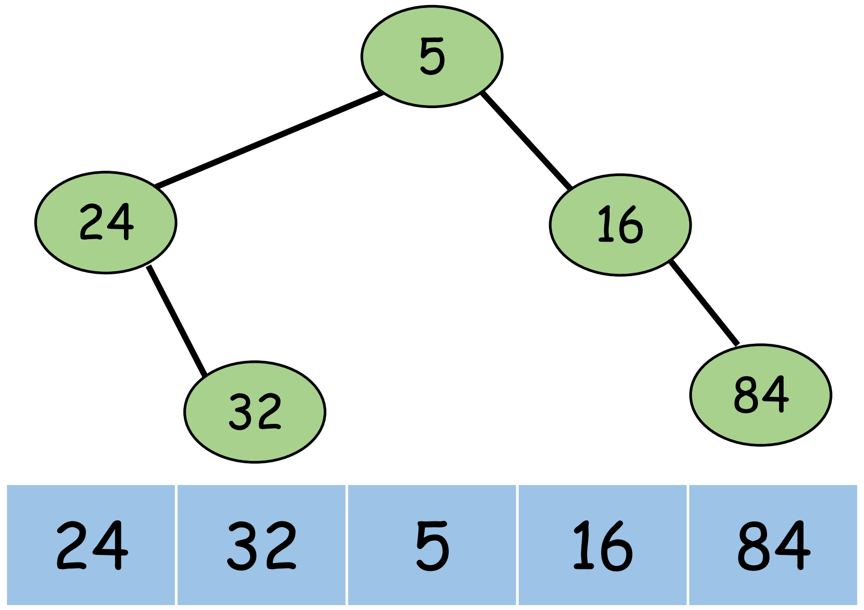
- An empty array has empty Cartesian tree
- For a non-empty array, the root stores the minimum value of the array. The left (right) child of the root is the Cartesian tree of the subarray to the left (right) of the minimum

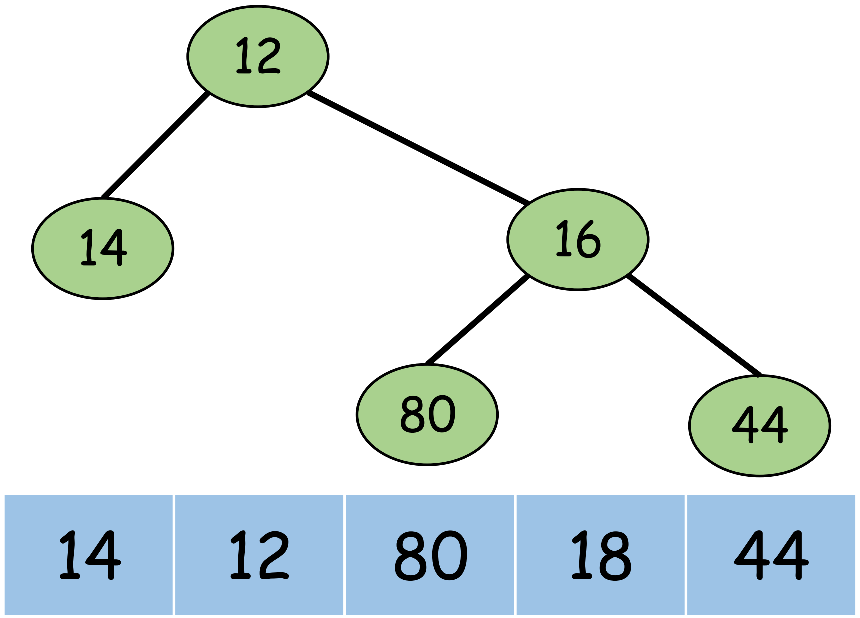




Another definition for Cartesian Tree

A Cartesian tree of an array is a binary tree that obeys the min-heap property, and the in-order traversal of the tree gives the array

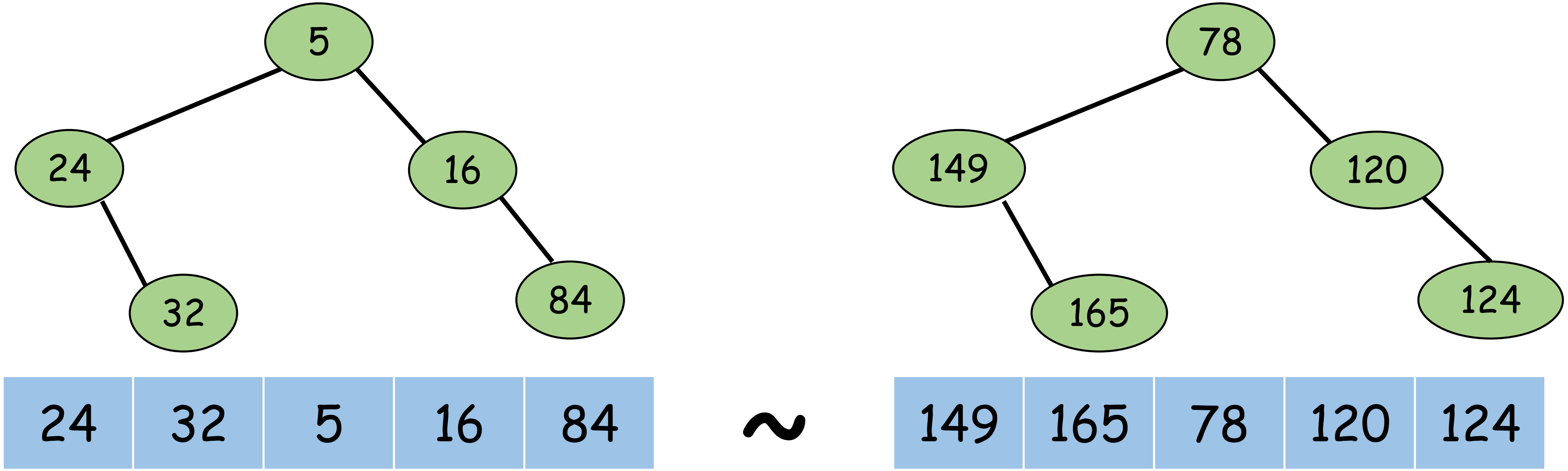




Why Cartesian Tree?

Theorem: Let X and Y be two blocks of size b. X ~ Y iff the Cartesian tree of X is isomorphic to the Cartesian tree of Y

X and Y can share one RMQ structure if their Cartesian trees are isomorphic



Questions we were pondering…

If X ~ Y, then they can share one RMQ structure

How to efficiently find that X ~ Y?

How many types are there?

Questions we were pondering…

If X ~ Y, then they can share the same RMQ structure

How to efficiently find that X ~ Y?

How many types are there?

How to efficiently build a Cartesian tree?

How to efficiently check whether two Cartesian trees are isomorphic?

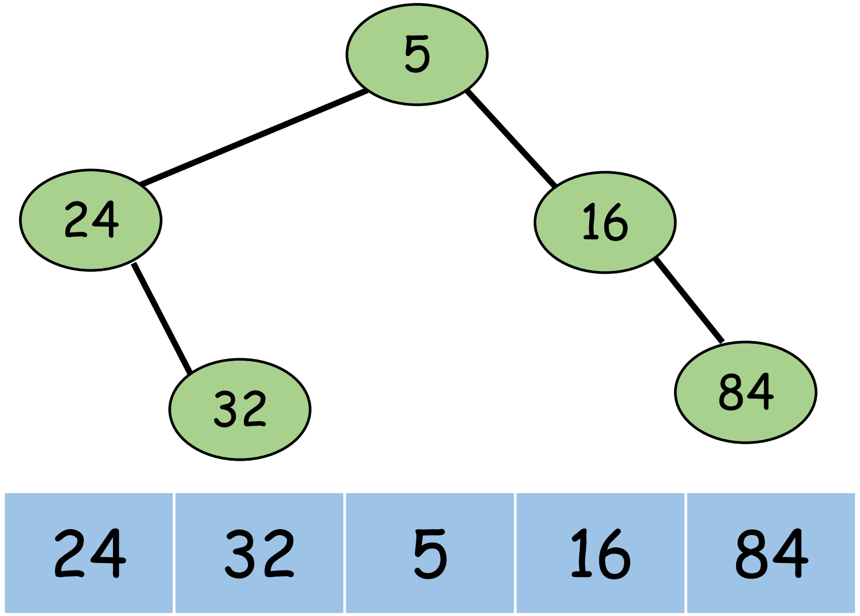
How to efficiently build a Cartesian Tree?

24	32	5	16	84

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Assume the tree has been built for the first five entries

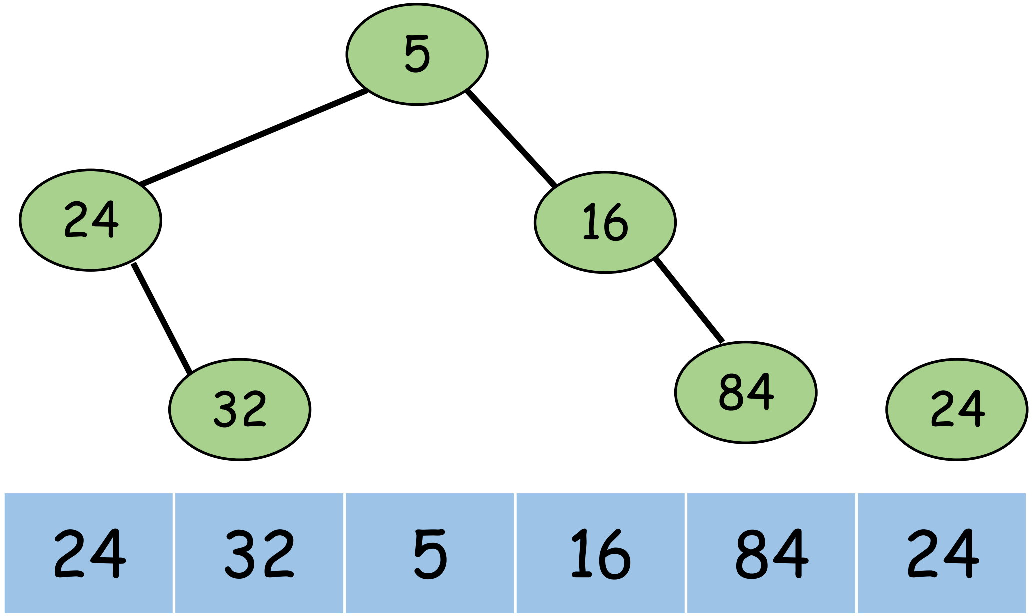


How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately

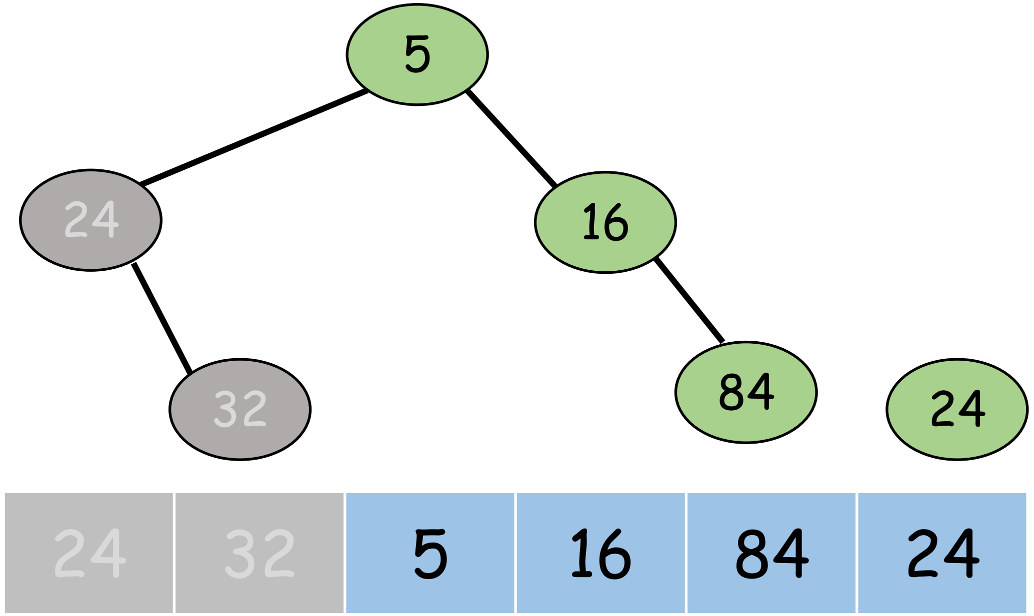
In-order traversal should produce the array. New node should be the right most node on the right spine



How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately

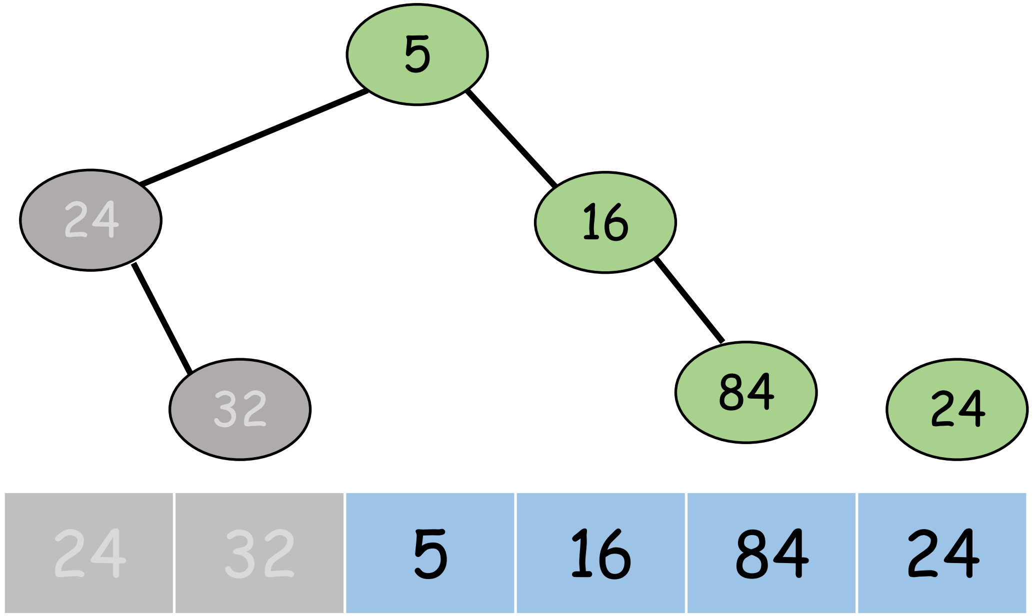


In-order traversal should produce the array. New node should be the right most node on the right spine

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately



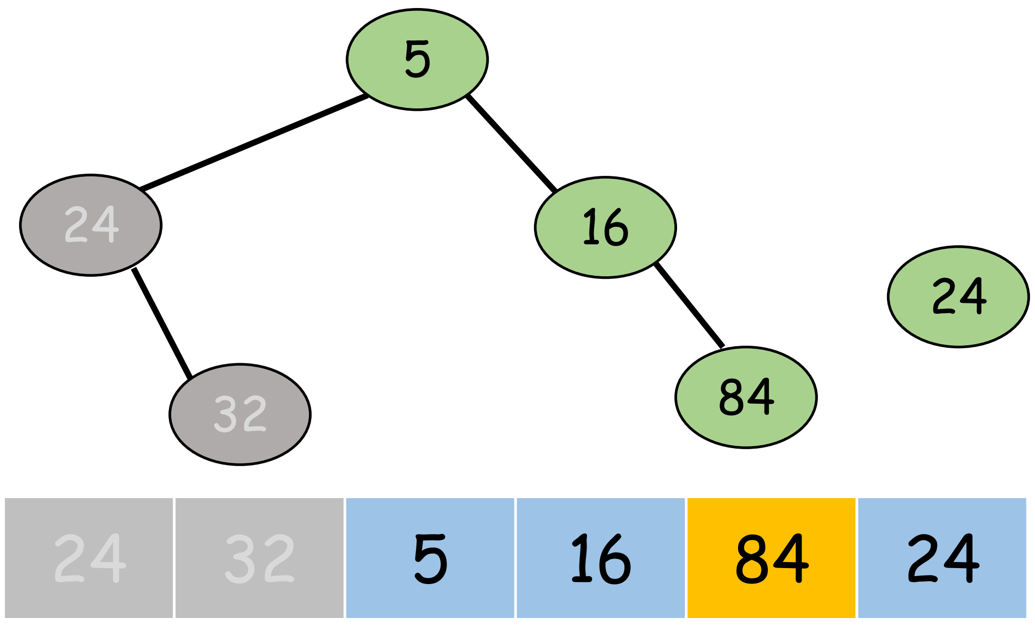
In-order traversal should produce the array. New node should be the right most node on the right spine

Min-heap property means parent of new node cannot be greater than the new node

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately



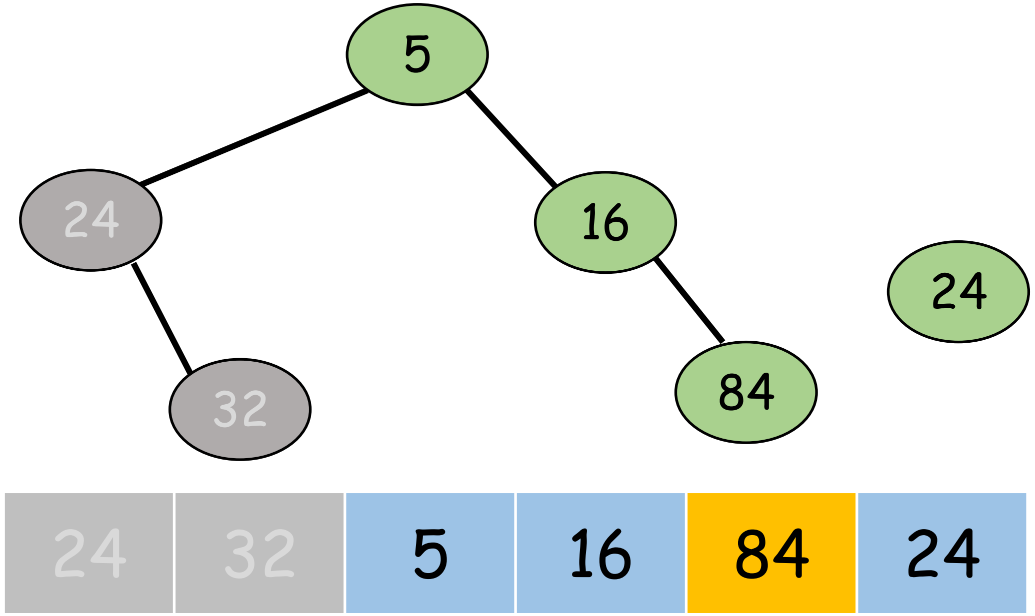
In-order traversal should produce the array. New node should be the right most node on the right spine

Min-heap property means parent of new node cannot be greater than the new node

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately



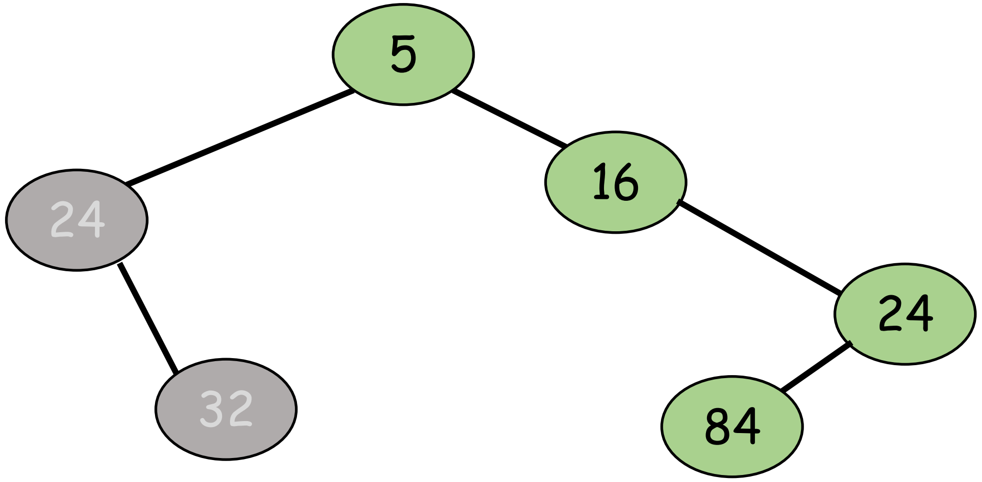
In-order traversal should produce the array. New node should be the right most node on the right spine

Min-heap property means parent of new node cannot be greater than the new node

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately



24	32	5	16	84	24

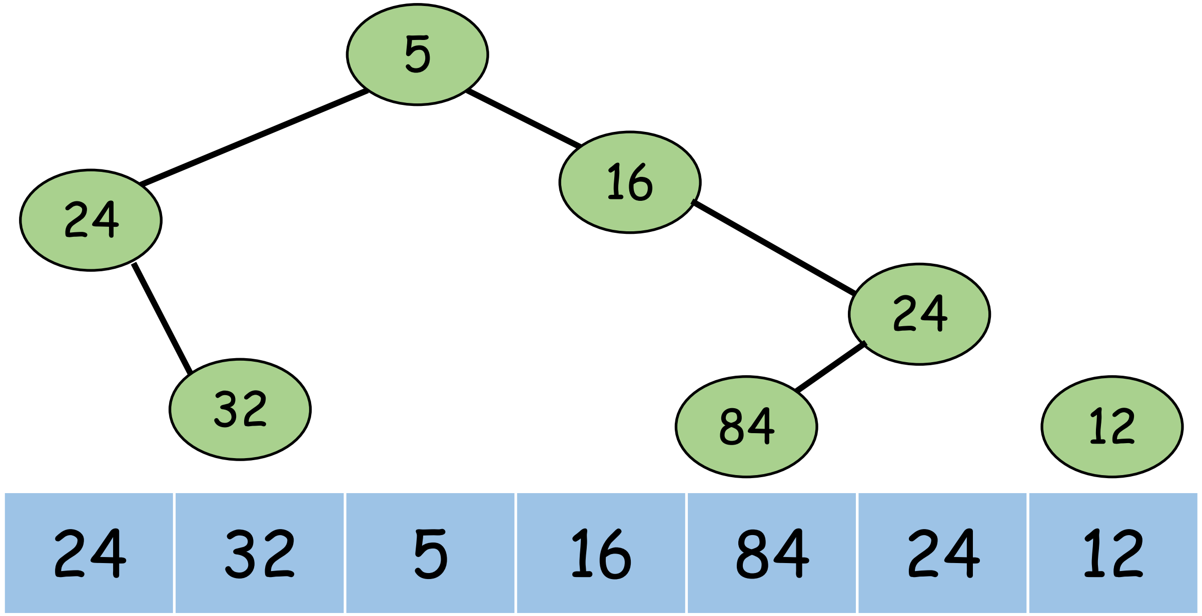
In-order traversal should produce the array. New node should be the right most node on the right spine

Min-heap property means parent of new node cannot be greater than the new node

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately

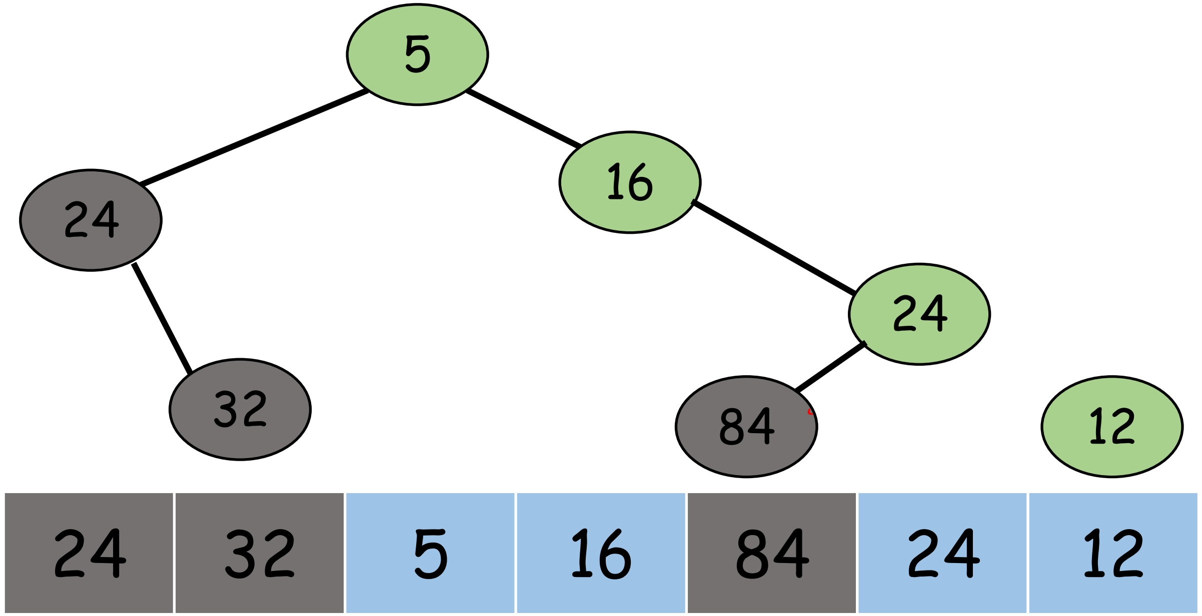


In-order traversal should produce the array. New node should be the right most node on the right spine

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately



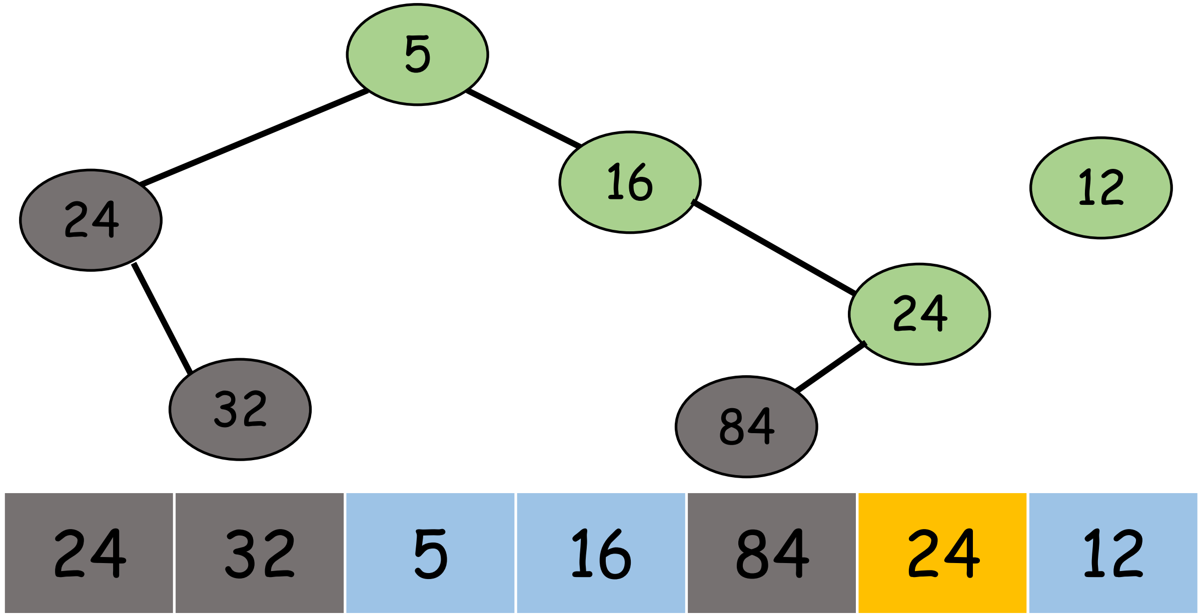
In-order traversal should produce the array. New node should be the right most node on the right spine

Min-heap property means parent of new node cannot be greater than the new node

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately



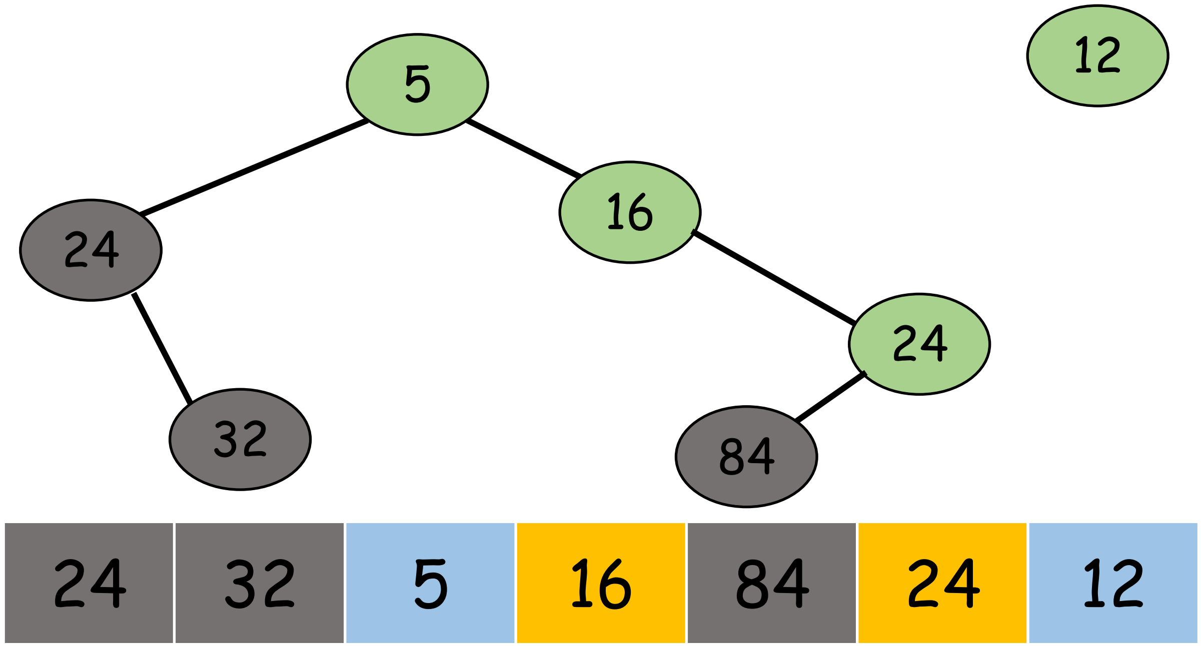
In-order traversal should produce the array. New node should be the right most node on the right spine

Min-heap property means parent of new node cannot be greater than the new node

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately



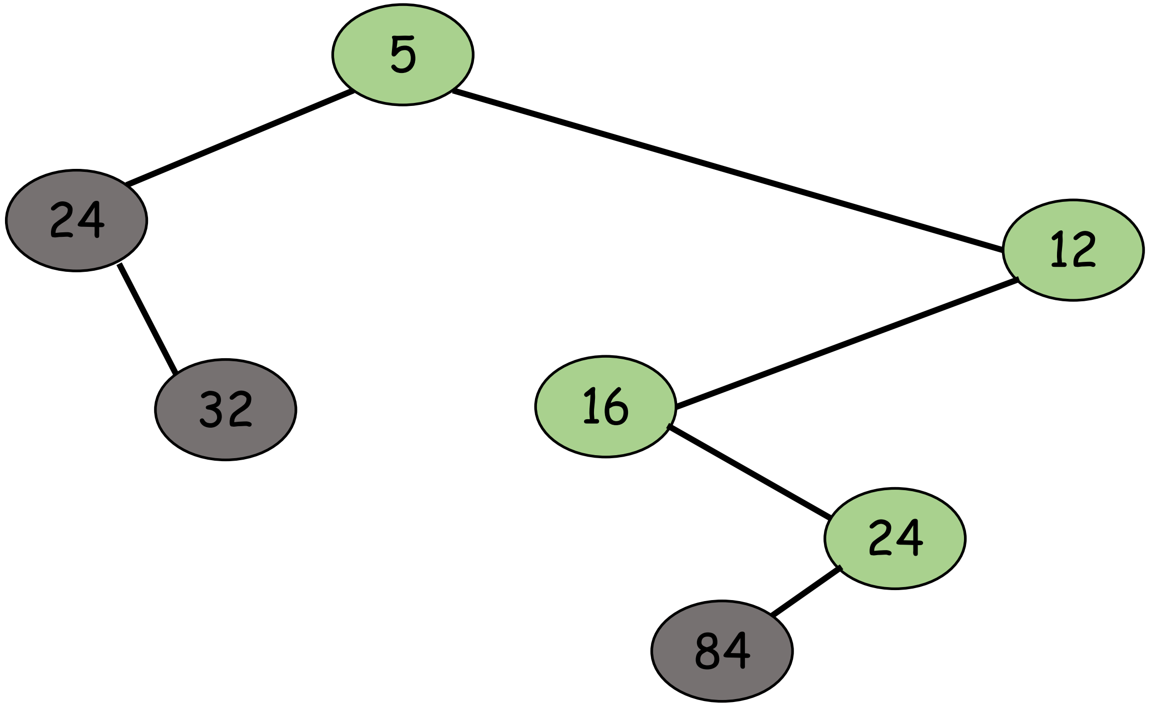
In-order traversal should produce the array. New node should be the right most node on the right spine

Min-heap property means parent of new node cannot be greater than the new node

How to efficiently build a Cartesian Tree?

Start with the first entry of the array and build it incrementally

Add the node for the next entry to the tree appropriately



24	32	5	16	84	24	12

In-order traversal should produce the array. New node should be the right most node on the right spine

Min-heap property means parent of new node cannot be greater than the new node

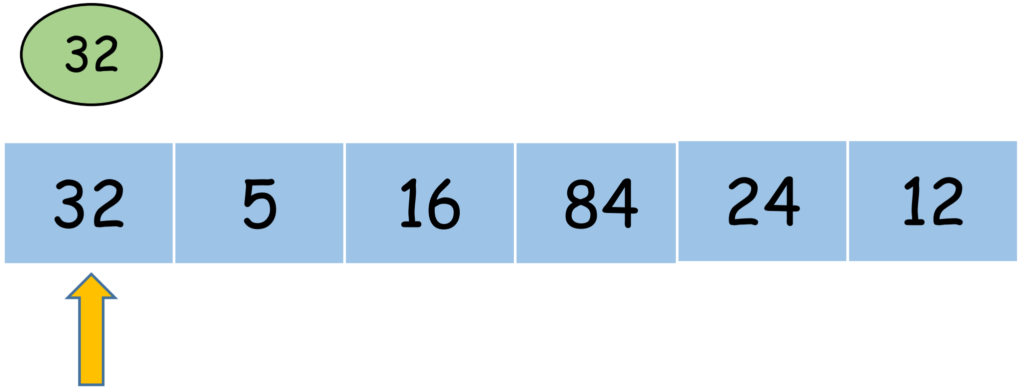
A stack-based algorithm

- Keep the nodes on the right spine in a stack

32	5	16	84	24	12

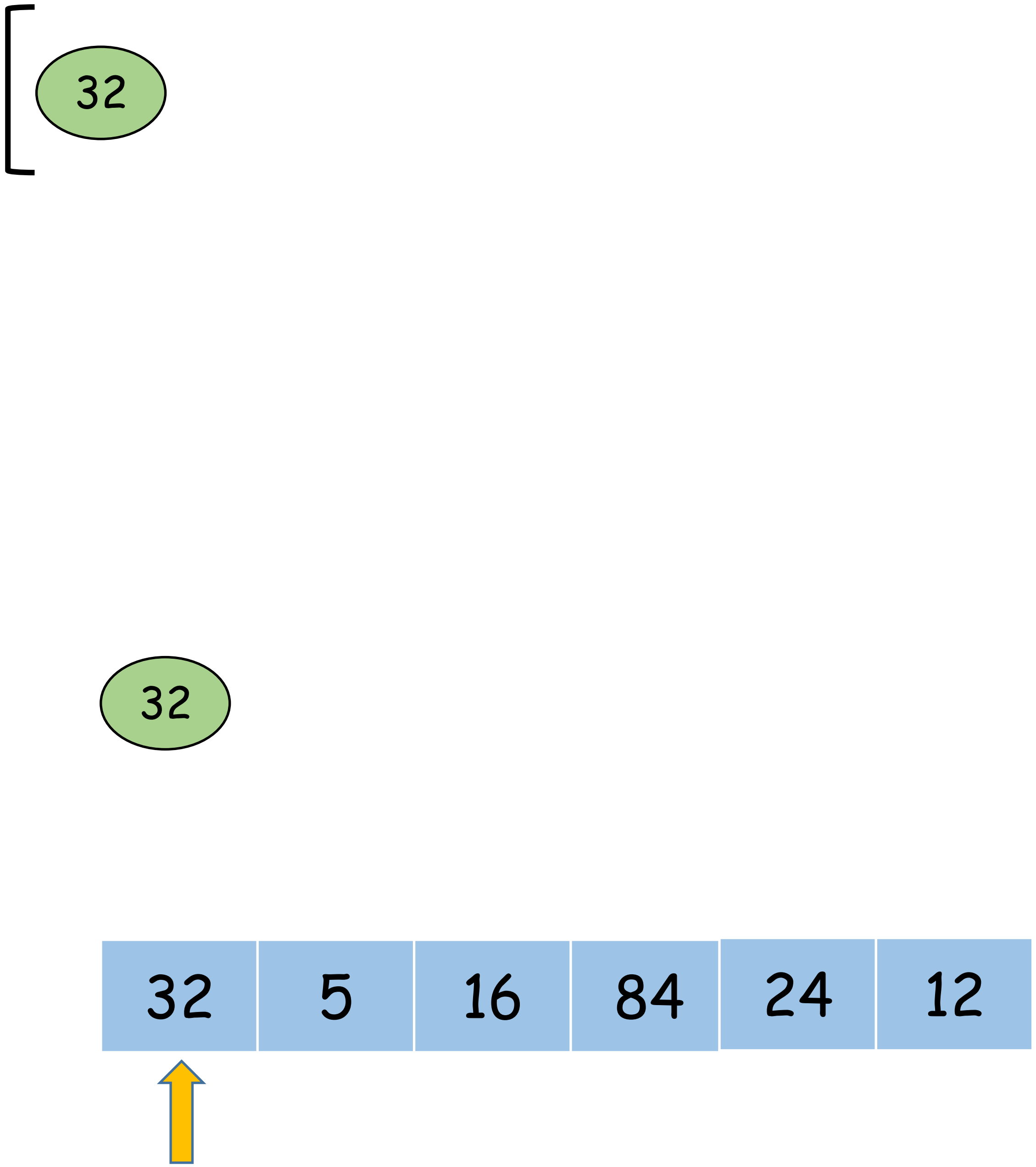
A stack-based algorithm

- Keep the nodes on the right spine in a stack



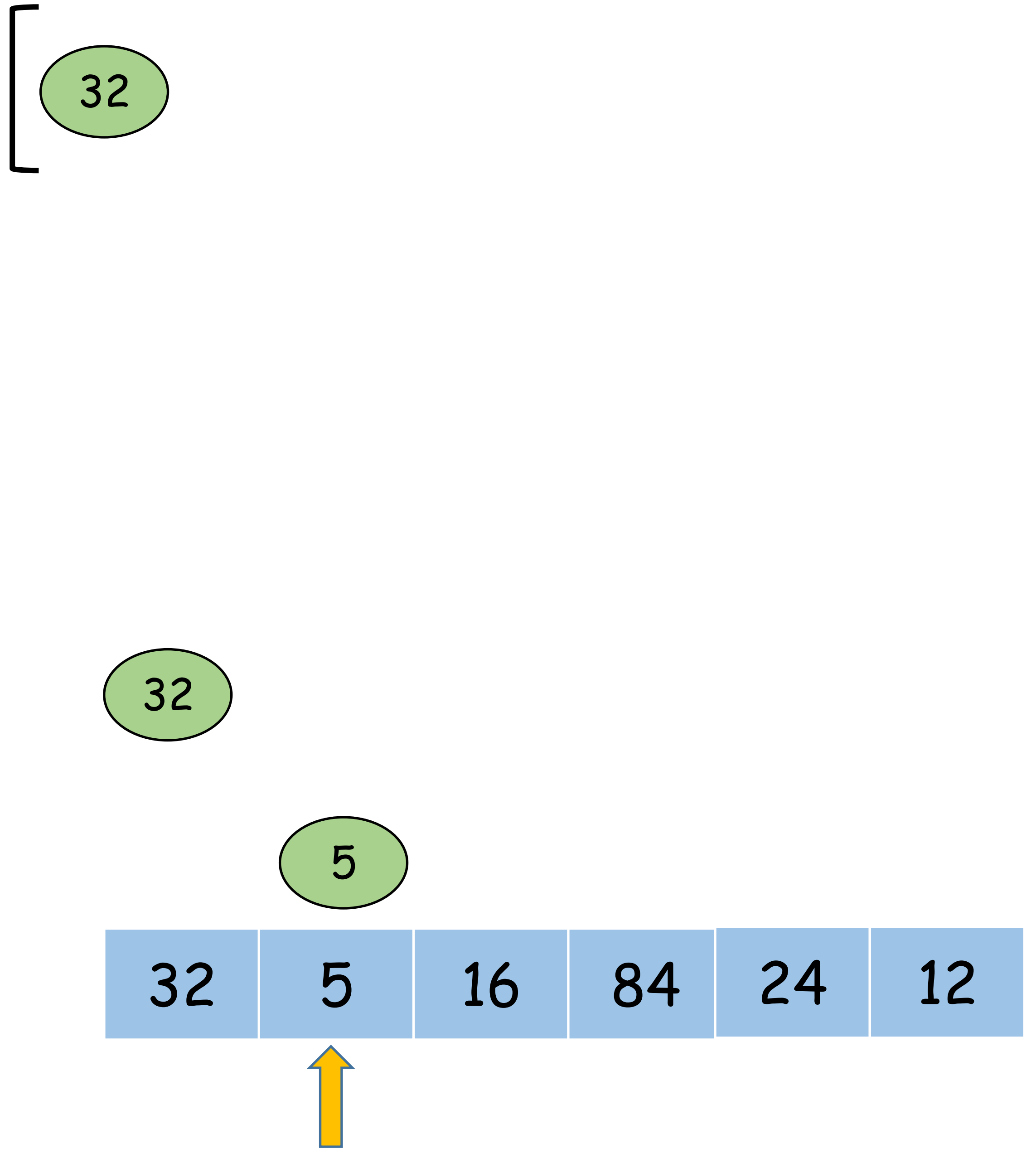
A stack-based algorithm

- Keep the nodes on the right spine in a stack



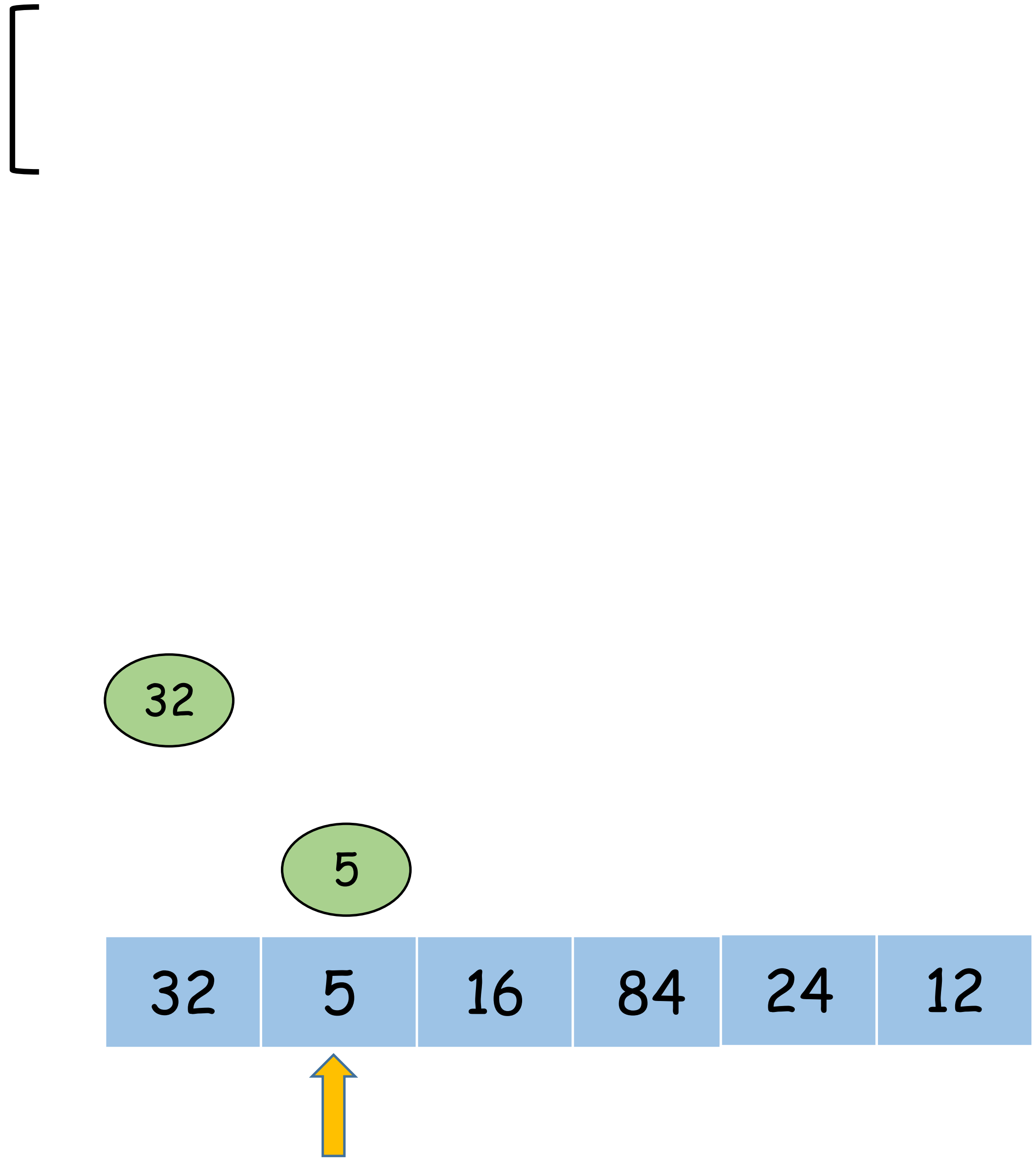
A stack-based algorithm

- Keep the nodes on the right spine in a stack



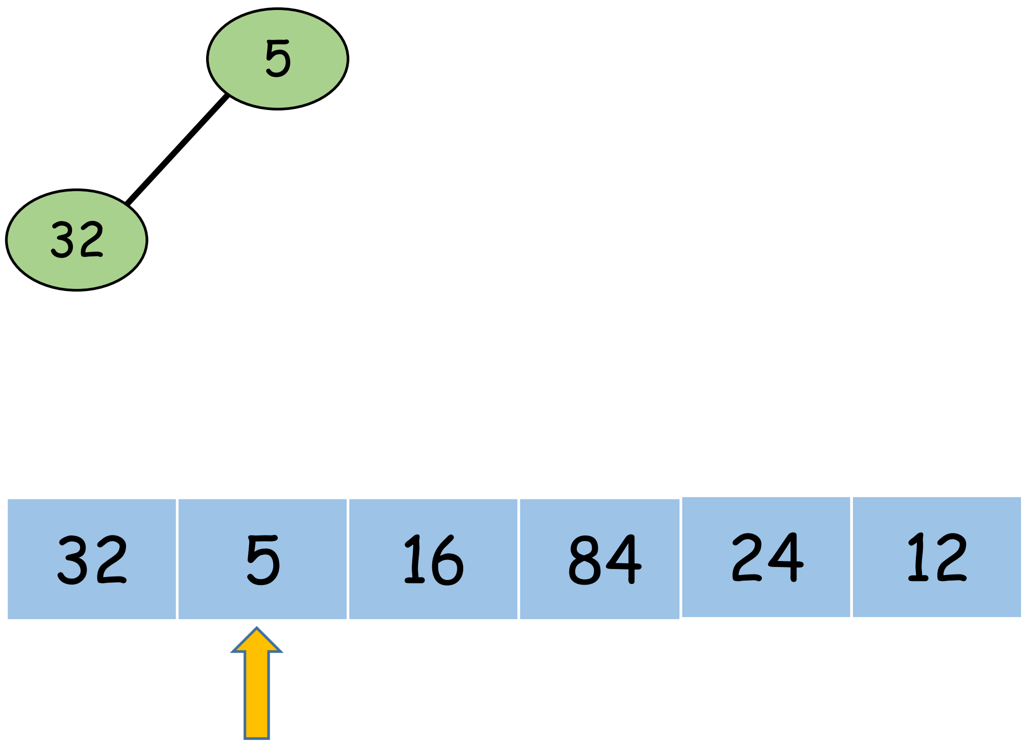
A stack-based algorithm

- Keep the nodes on the right spine in a stack



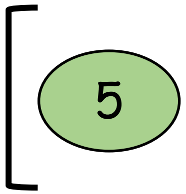
A stack-based algorithm

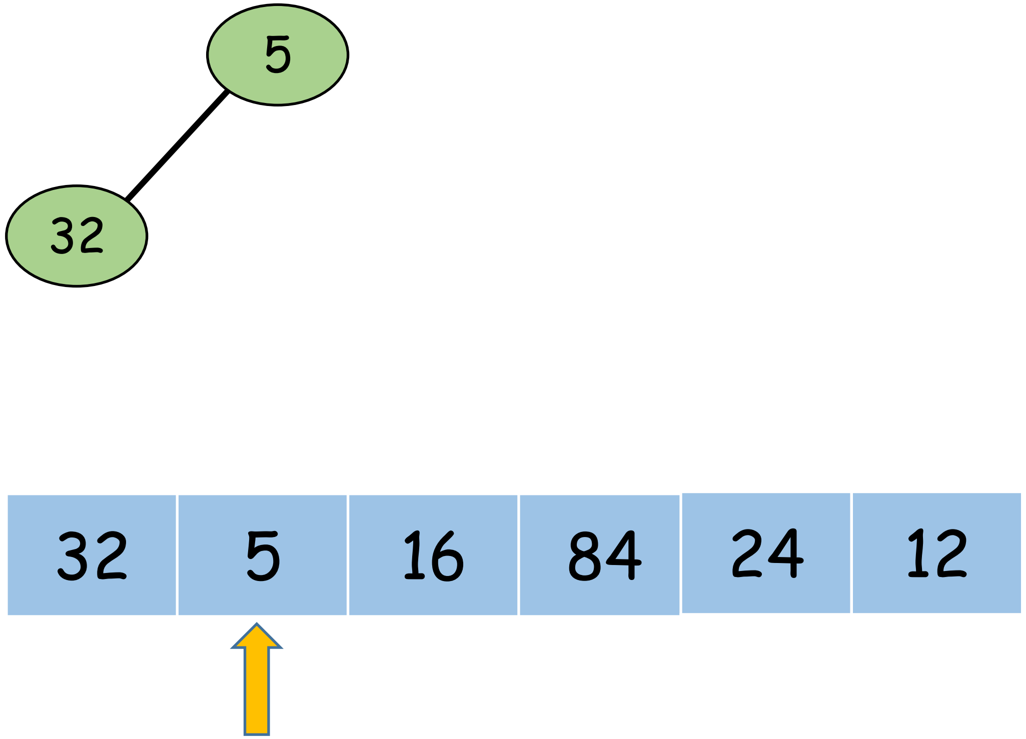
- Keep the nodes on the right spine in a stack



A stack-based algorithm

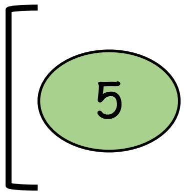
- Keep the nodes on the right spine in a stack

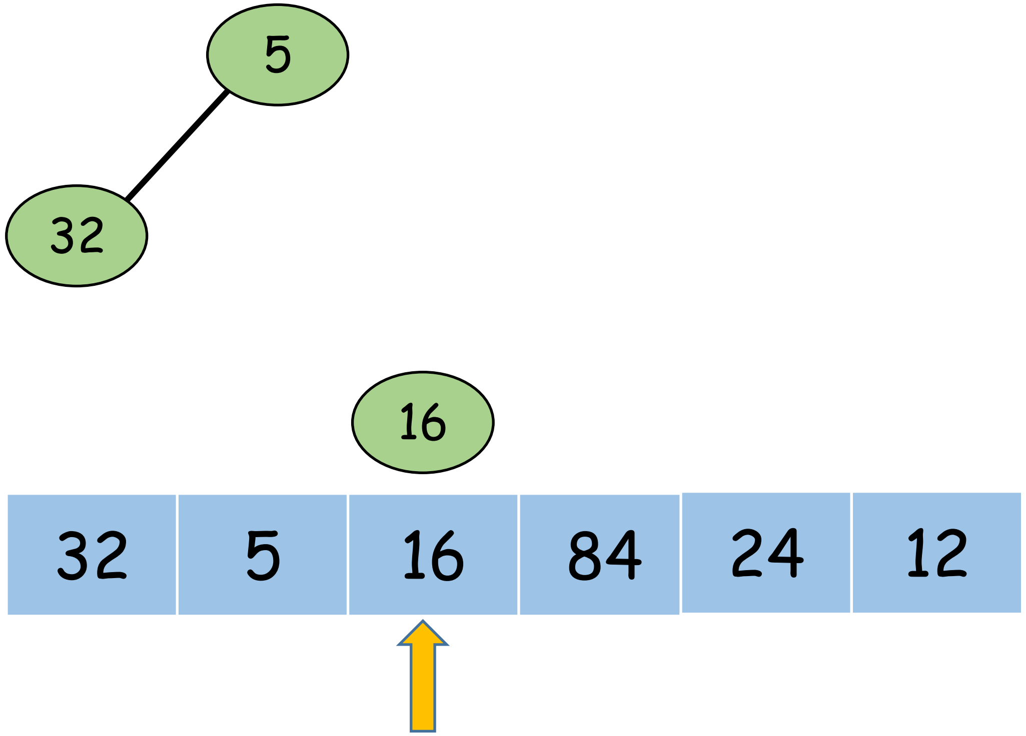




A stack-based algorithm

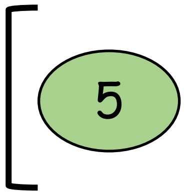
- Keep the nodes on the right spine in a stack

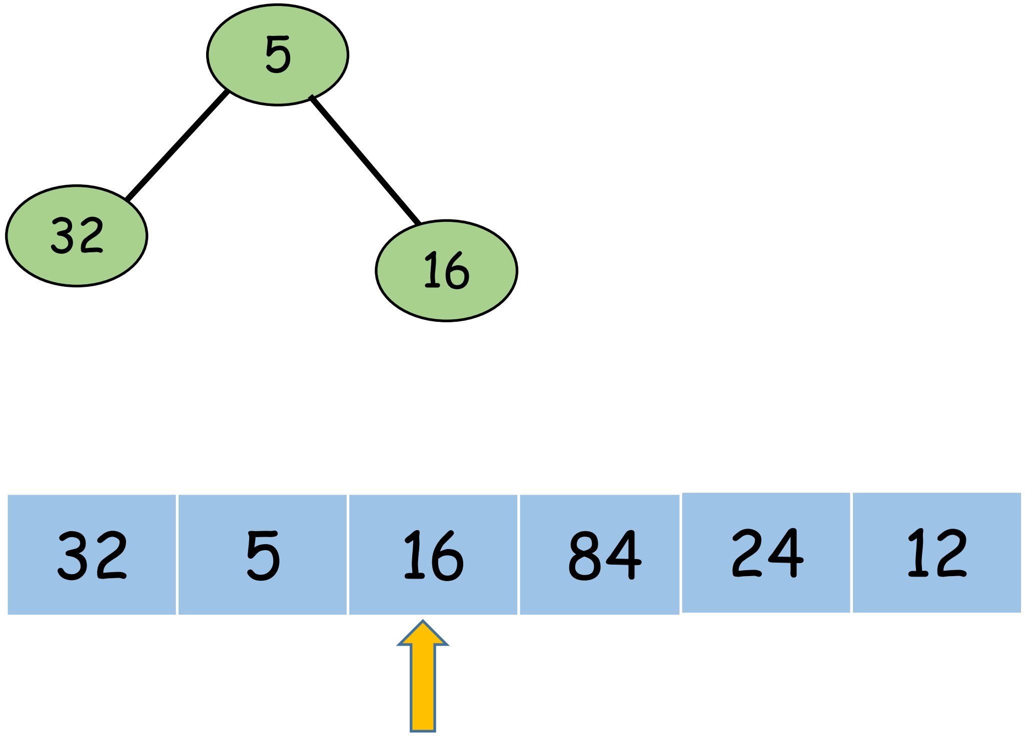




A stack-based algorithm

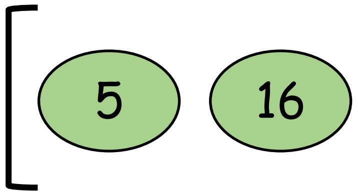
- Keep the nodes on the right spine in a stack

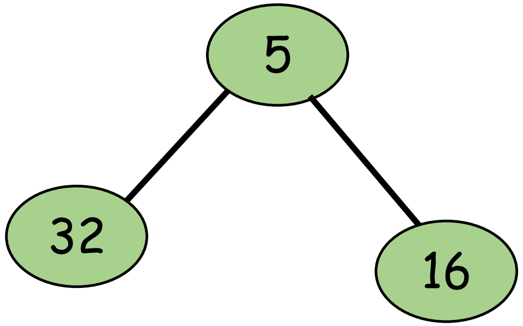




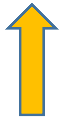
A stack-based algorithm

- Keep the nodes on the right spine in a stack



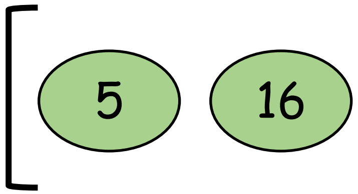


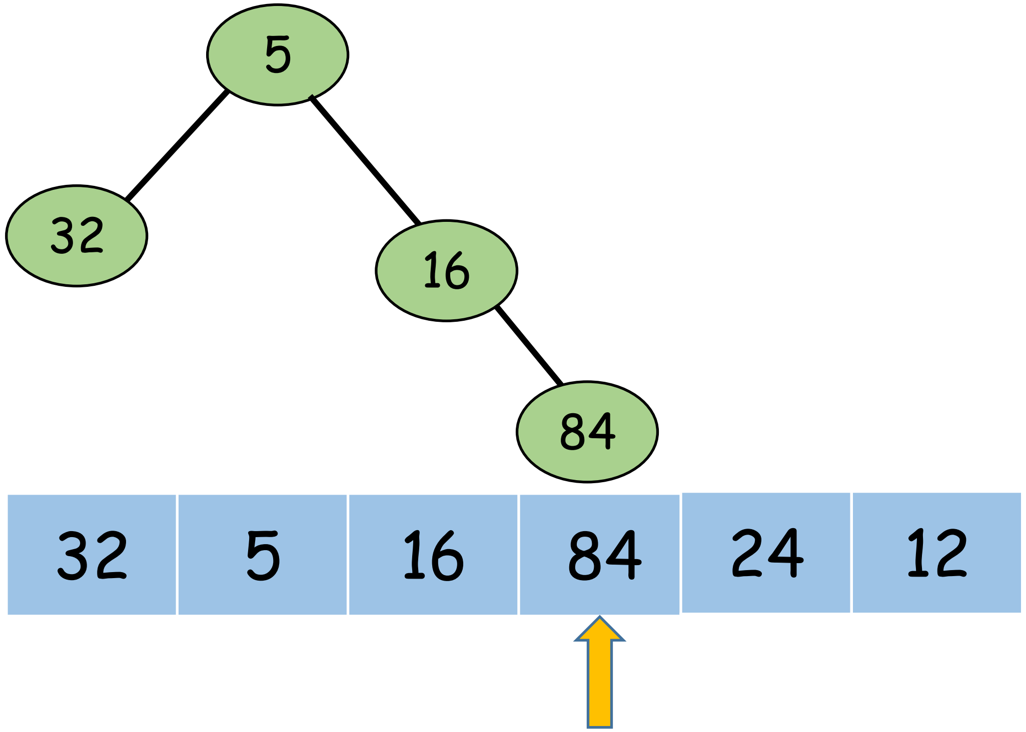
32	5	16	84	24	12



A stack-based algorithm

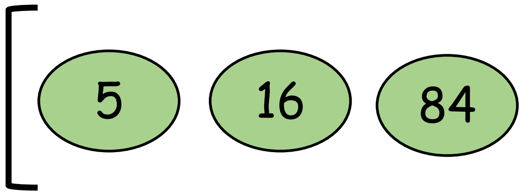
- Keep the nodes on the right spine in a stack

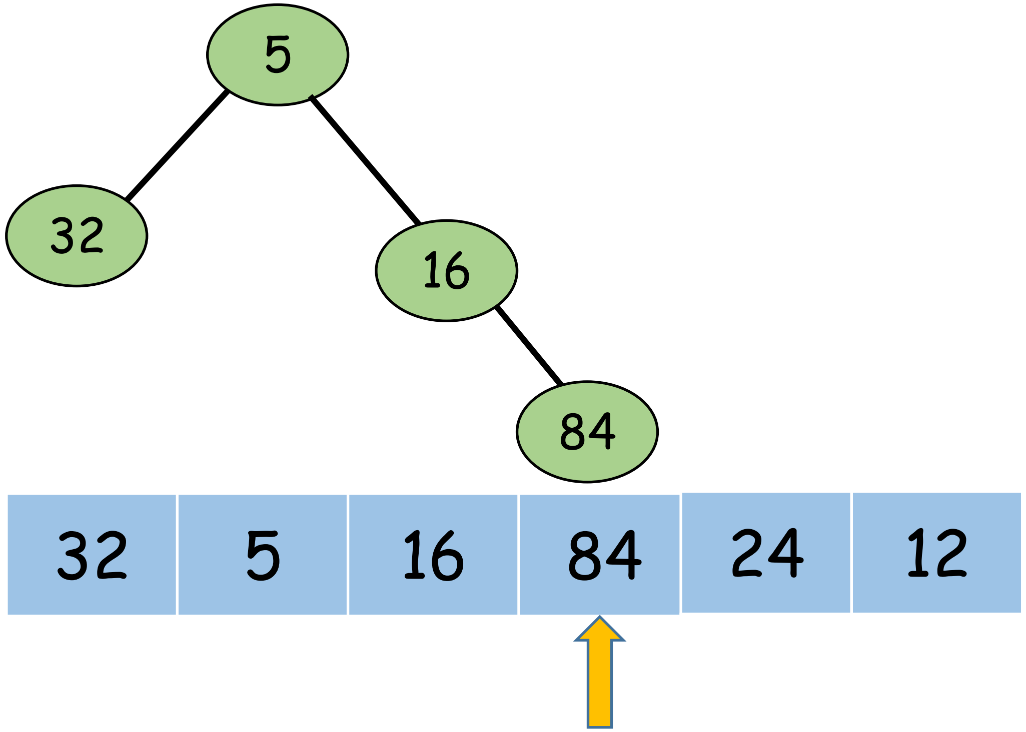




A stack-based algorithm

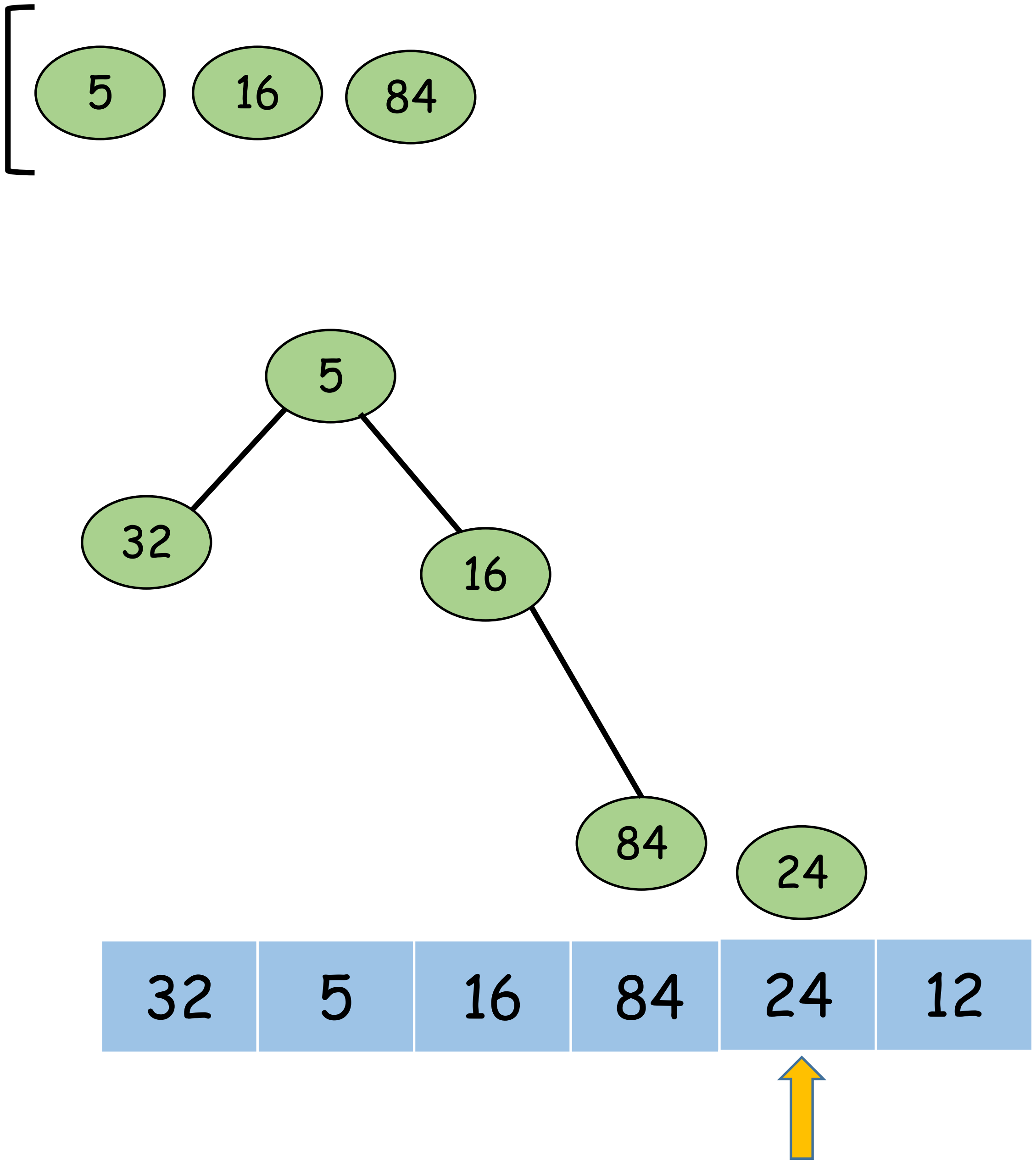
- Keep the nodes on the right spine in a stack





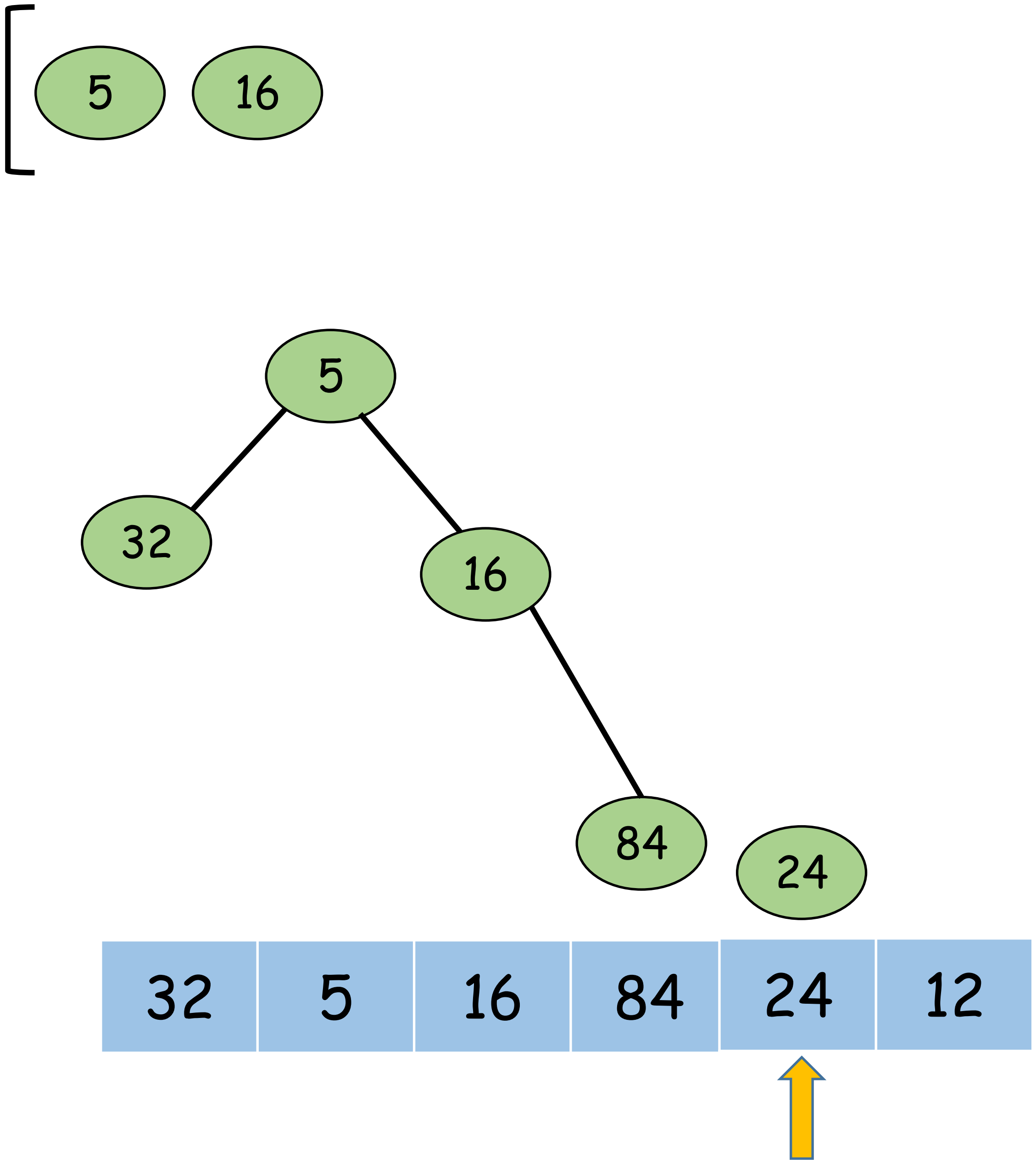
A stack-based algorithm

- Keep the nodes on the right spine in a stack



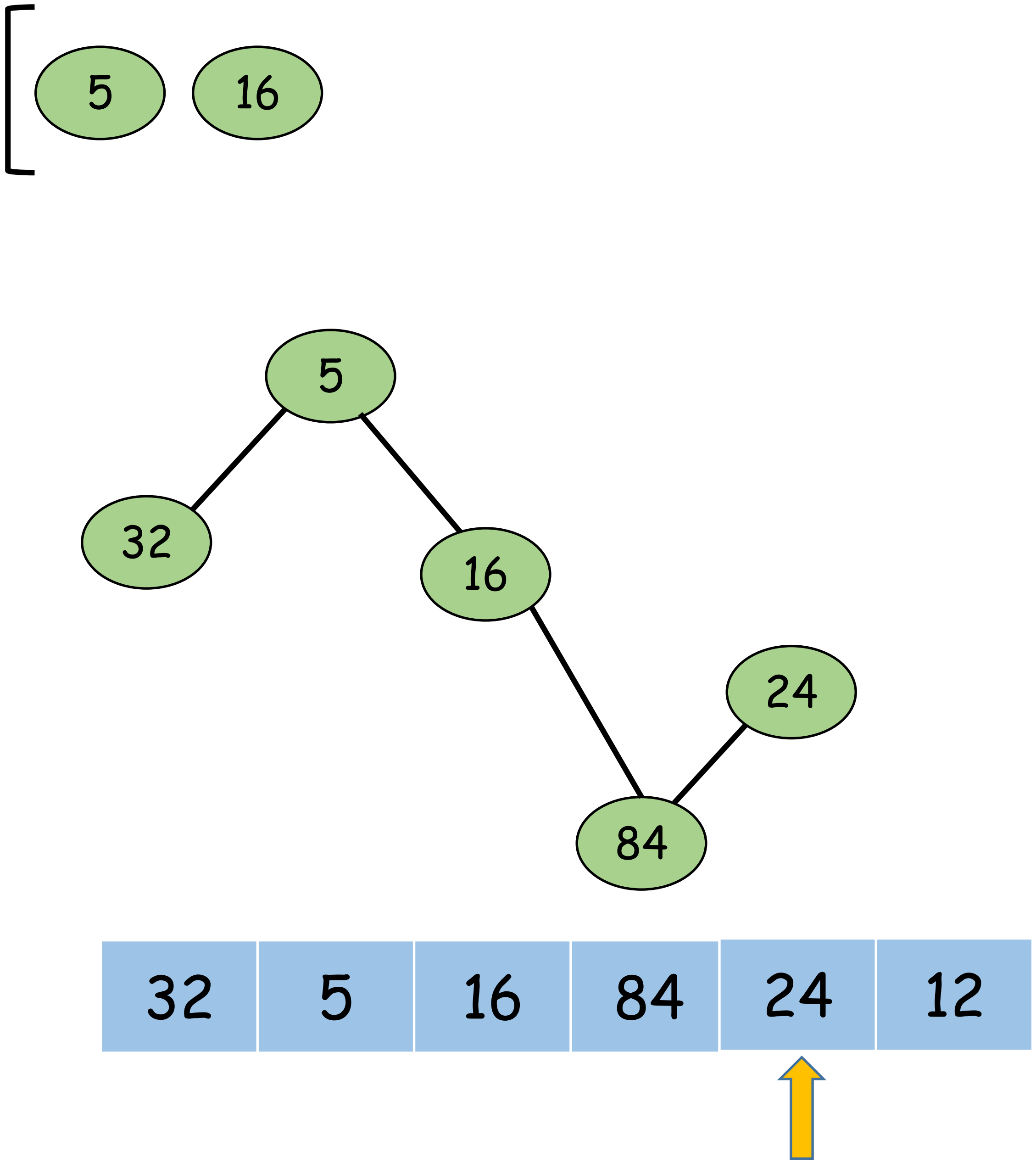
A stack-based algorithm

- Keep the nodes on the right spine in a stack



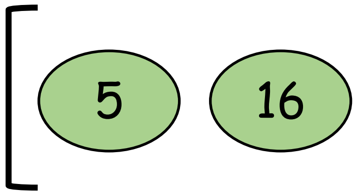
A stack-based algorithm

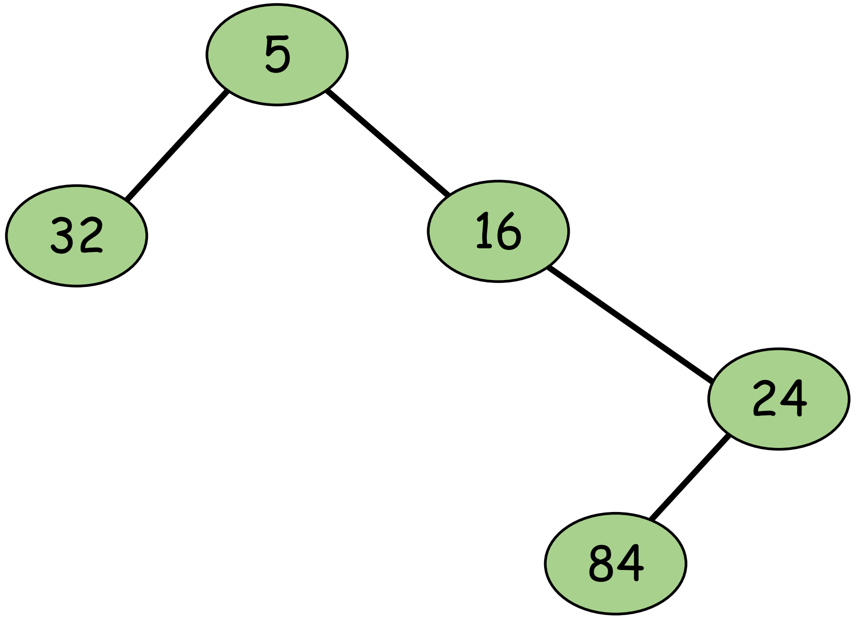
- Keep the nodes on the right spine in a stack



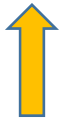
A stack-based algorithm

- Keep the nodes on the right spine in a stack



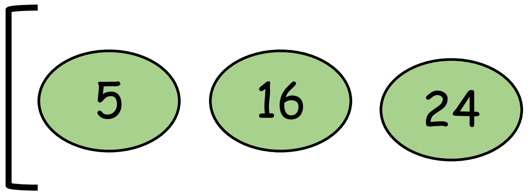


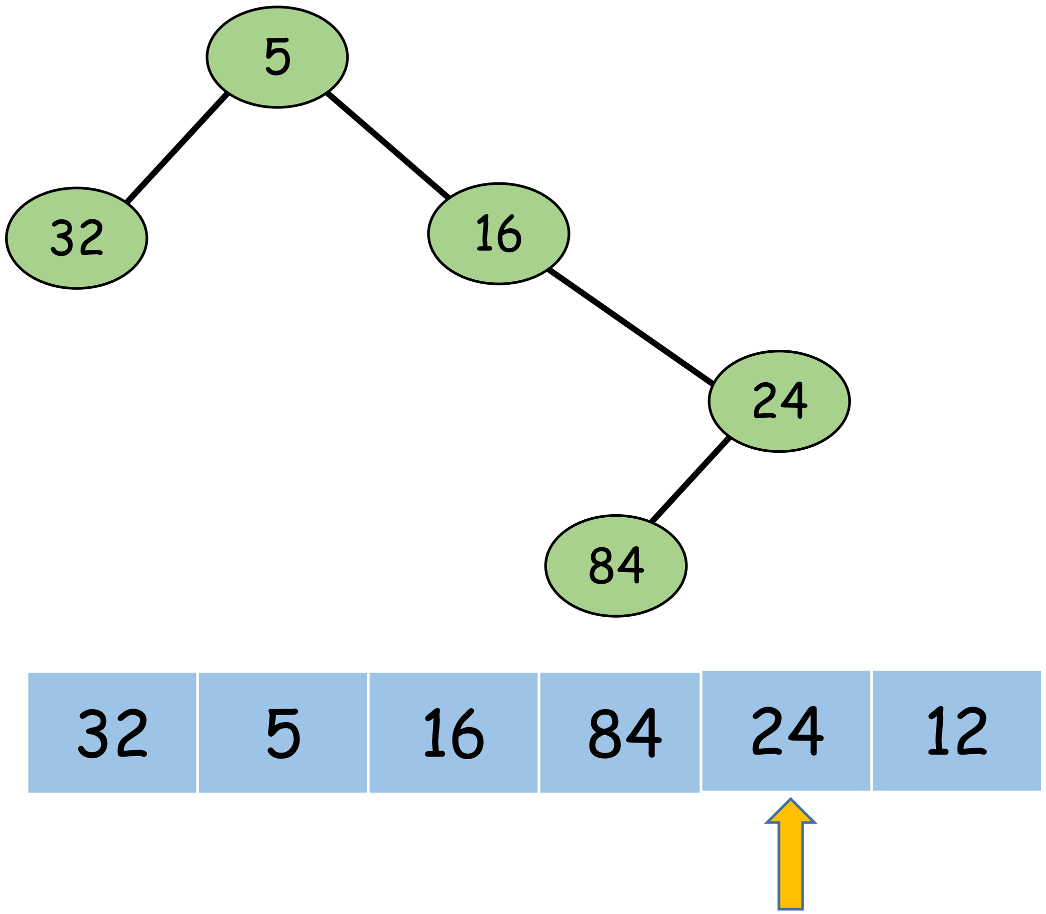
32	5	16	84	24	12



A stack-based algorithm

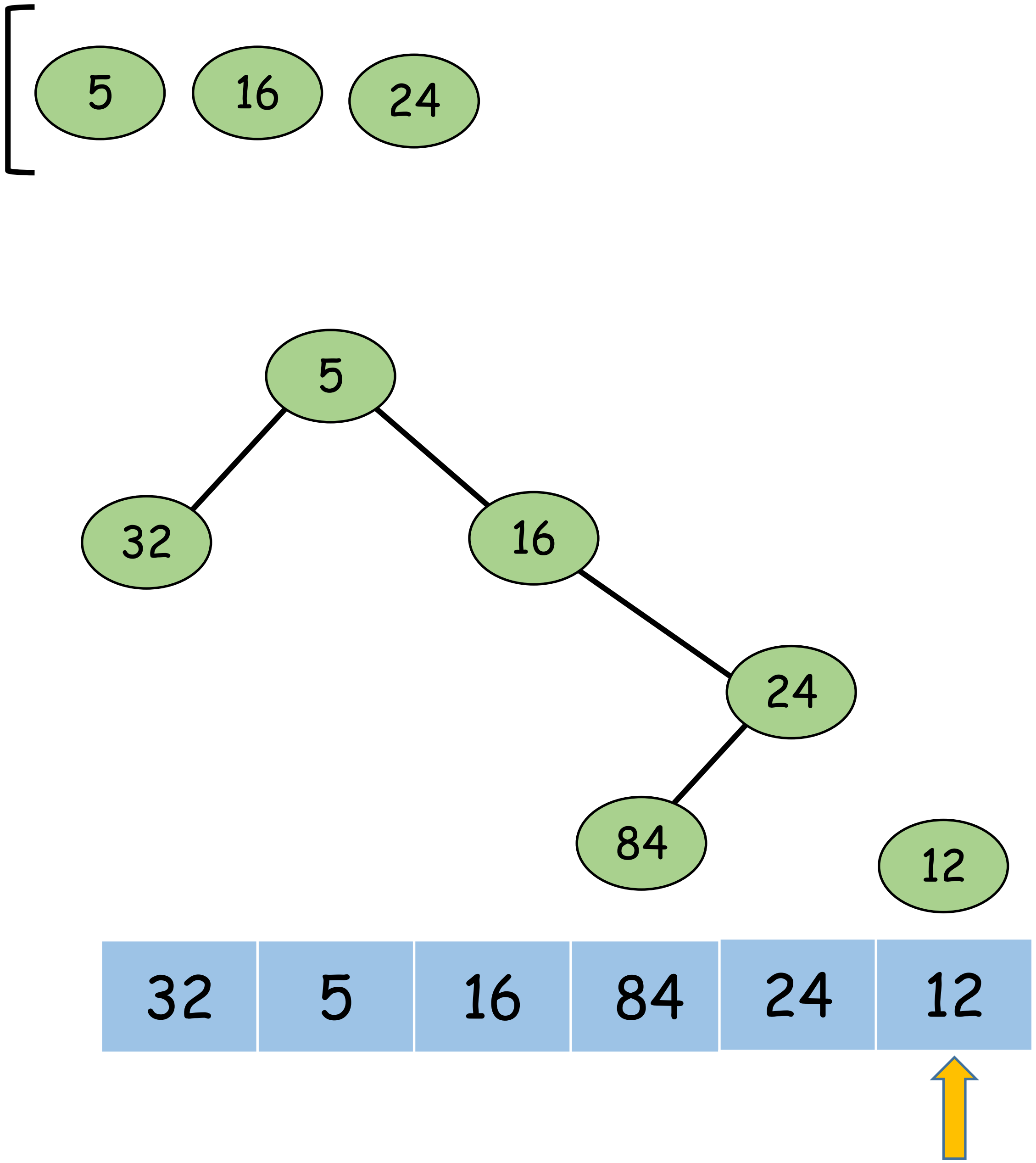
- Keep the nodes on the right spine in a stack





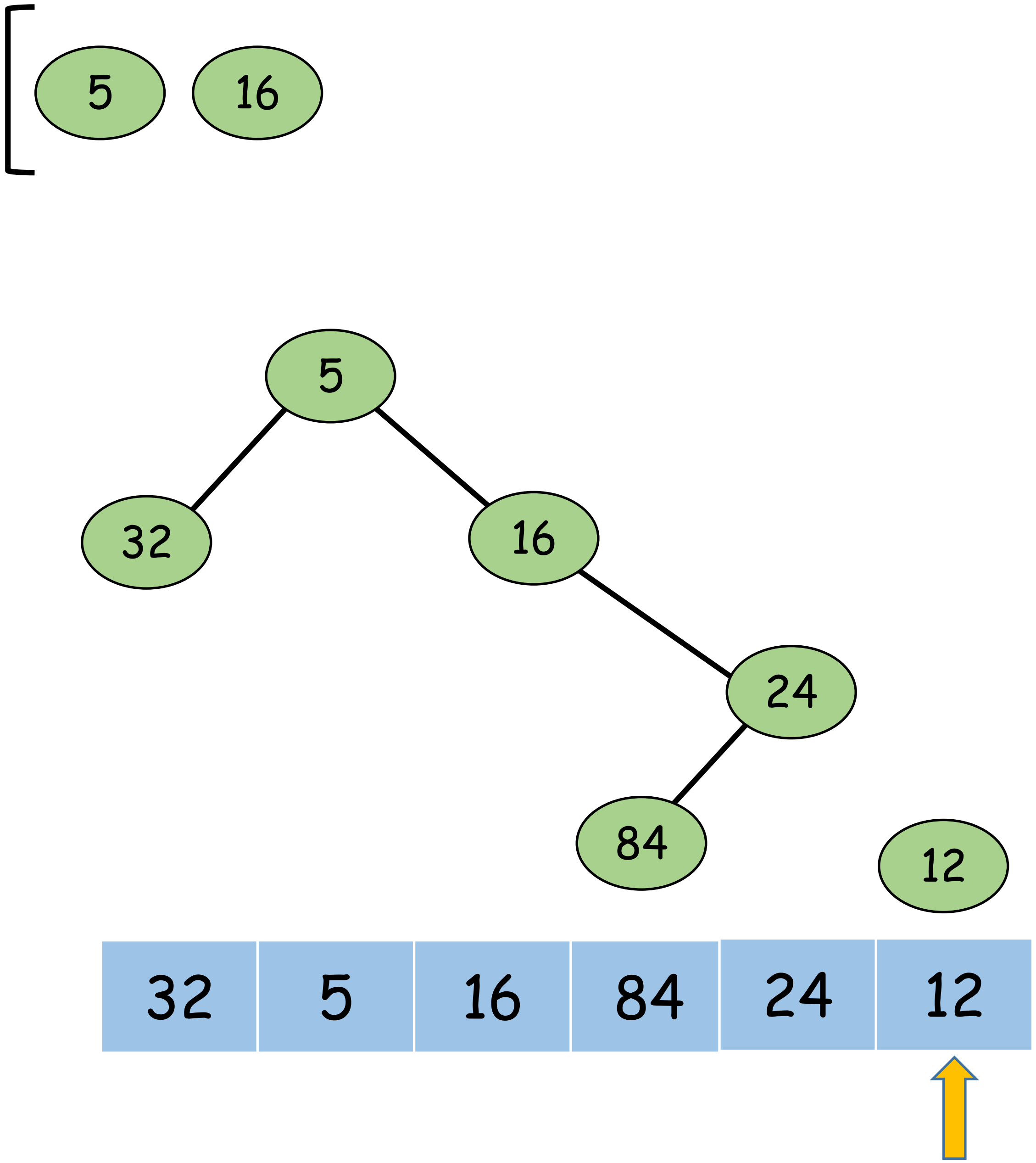
A stack-based algorithm

- Keep the nodes on the right spine in a stack



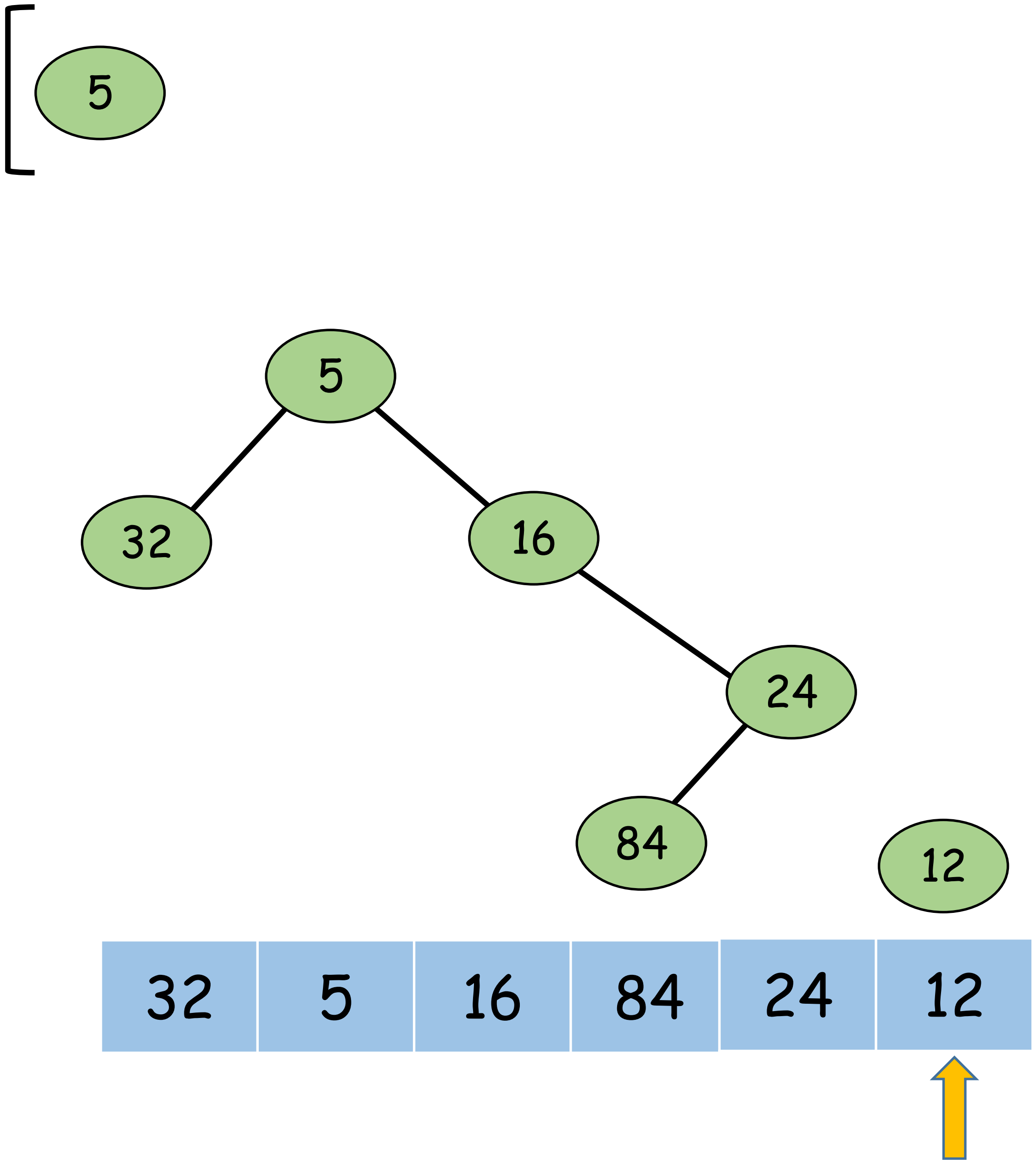
A stack-based algorithm

- Keep the nodes on the right spine in a stack



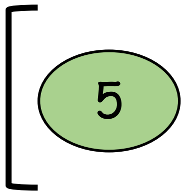
A stack-based algorithm

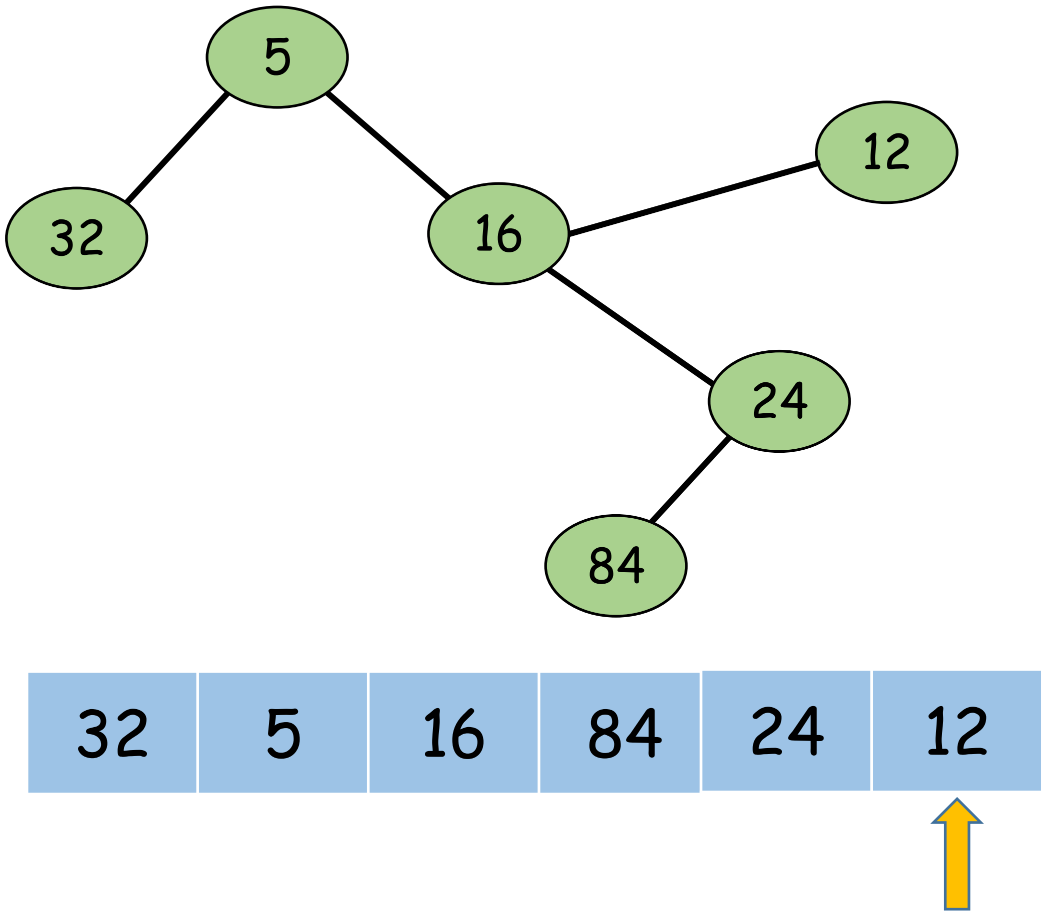
- Keep the nodes on the right spine in a stack



A stack-based algorithm

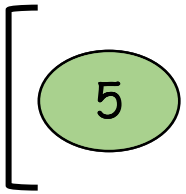
- Keep the nodes on the right spine in a stack

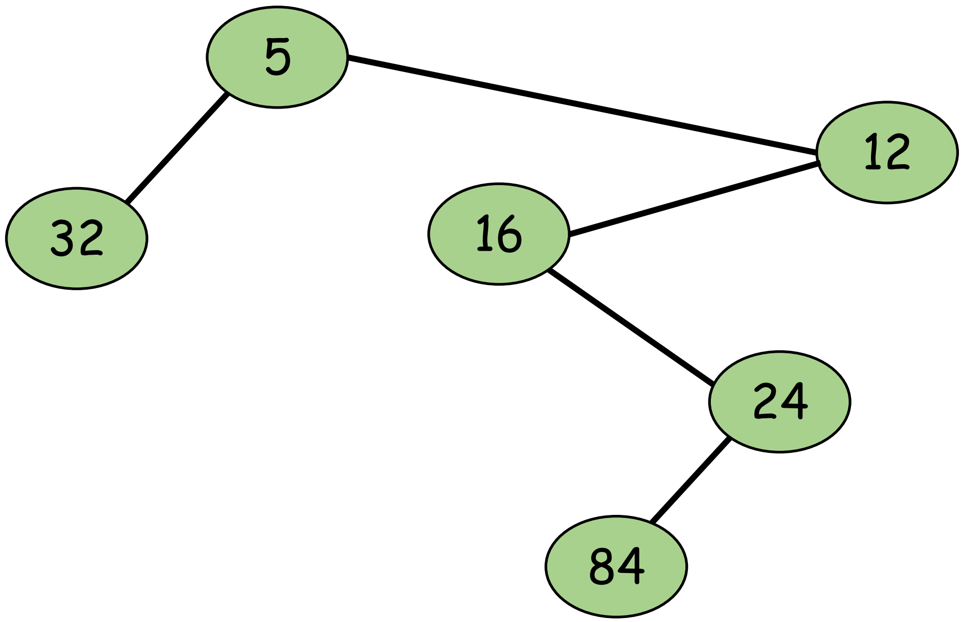




A stack-based algorithm

- Keep the nodes on the right spine in a stack

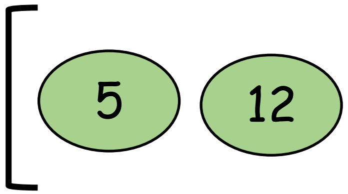


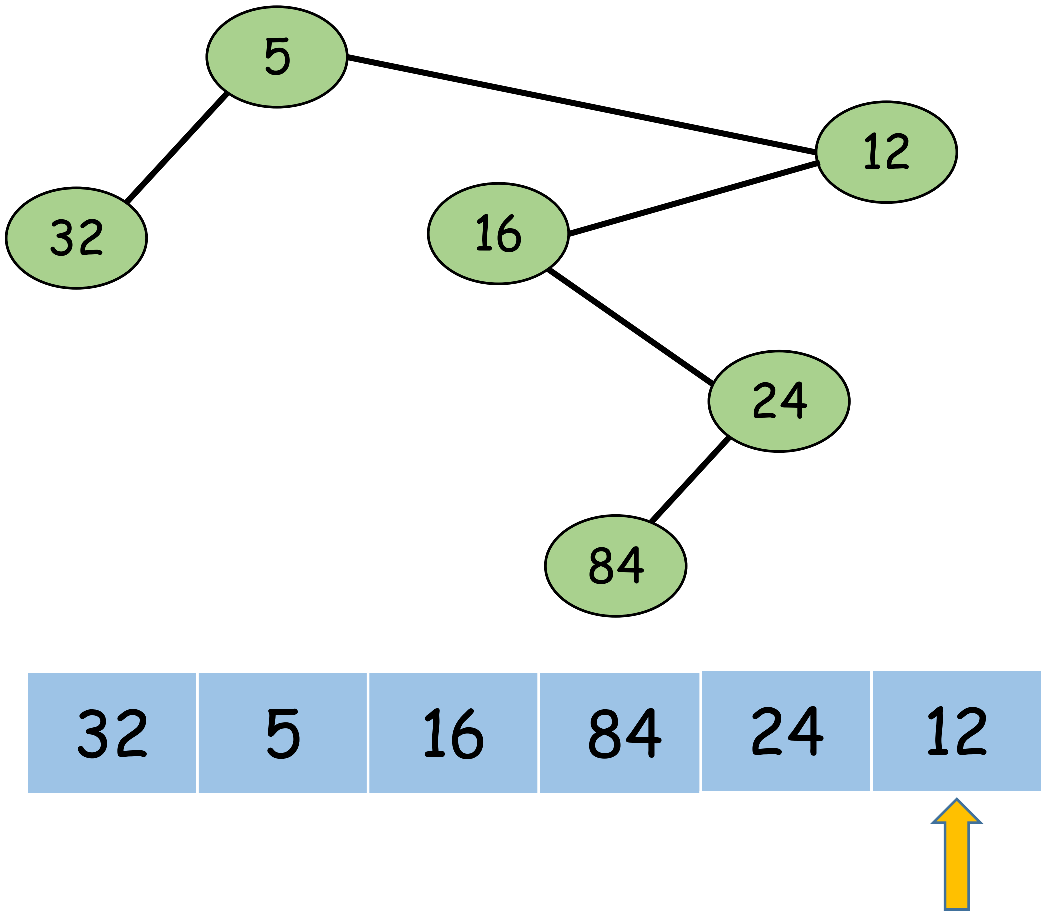


32	5	16	84	24	12

A stack-based algorithm

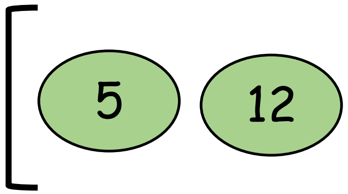
- Keep the nodes on the right spine in a stack

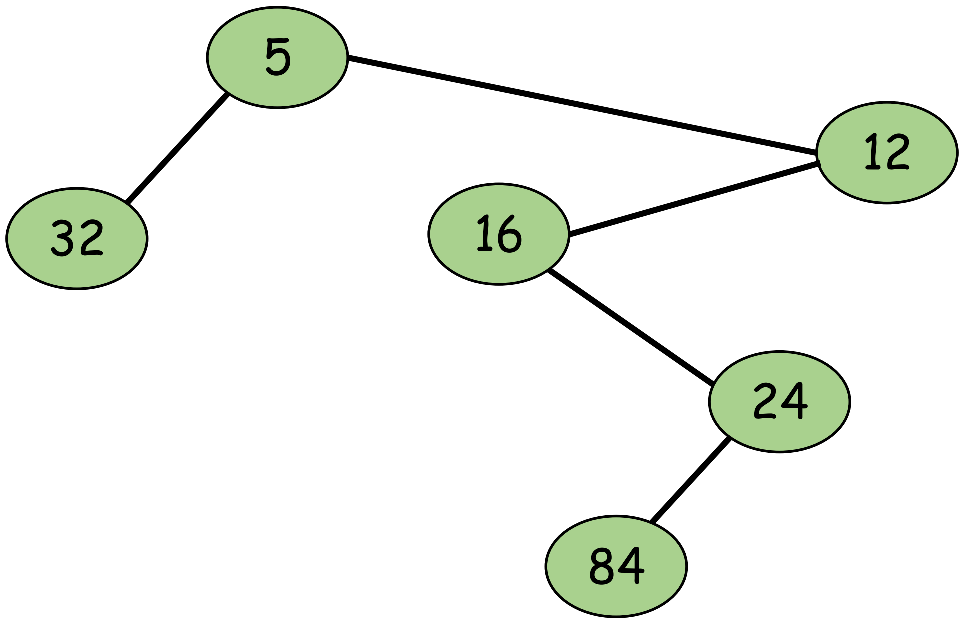


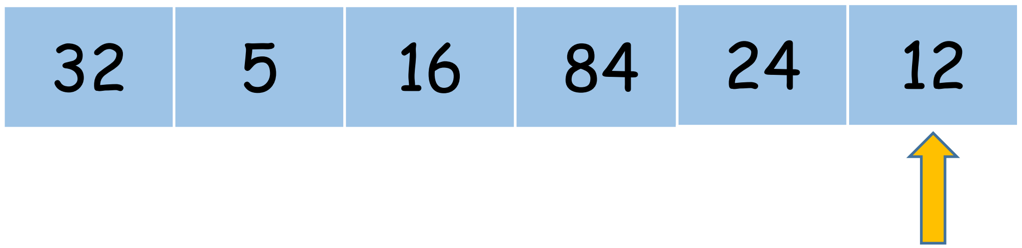


A stack-based algorithm

- Keep the nodes on the right spine in a stack
- Iterate over the array
  - Create a new node
  - Pop the stack till it is empty or top node is less than new node
  - Make the last node popped the left child of new node
  - If top is not null, make the new node the right child of the top
  - Push the new node into the stack

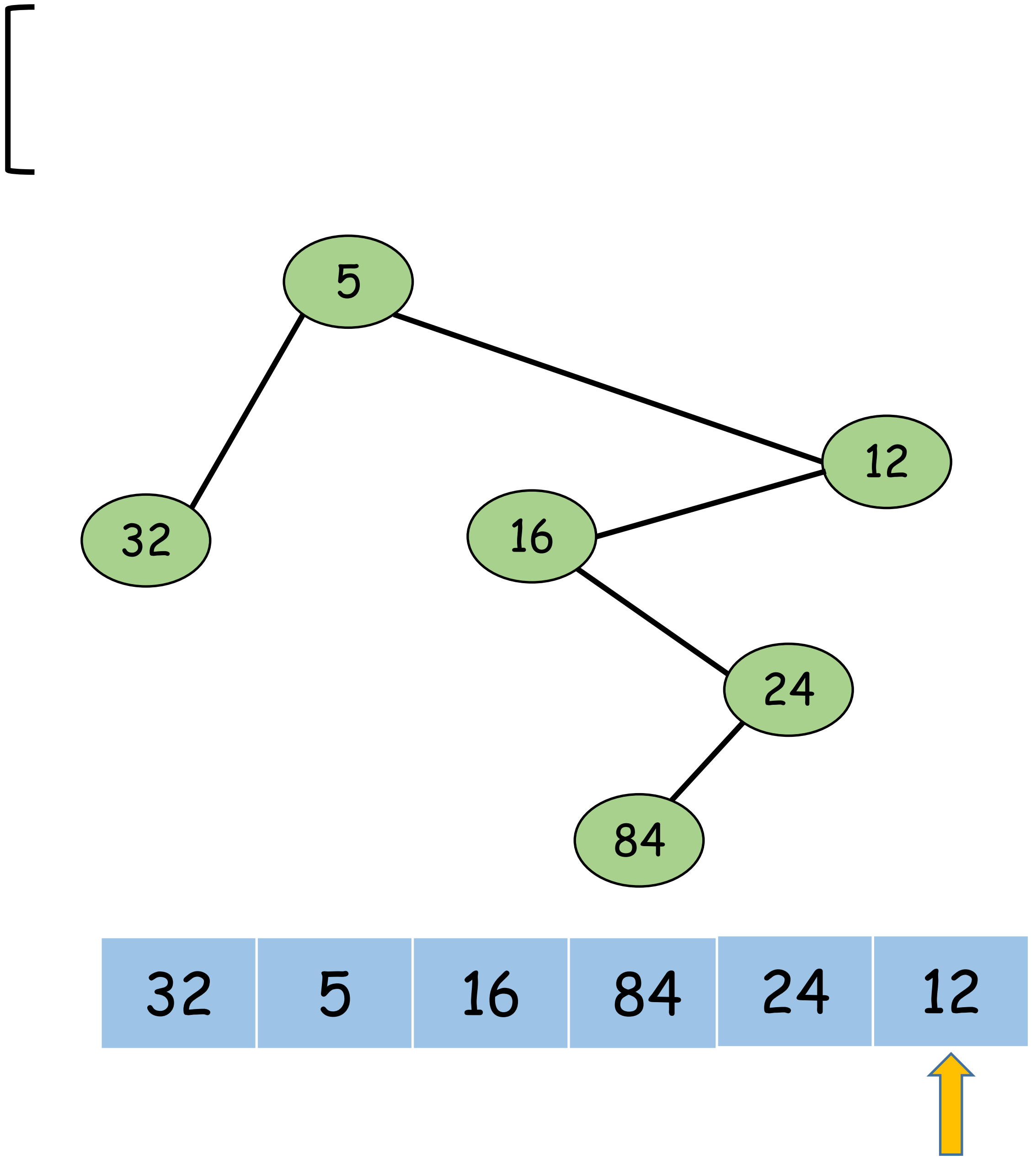






A stack-based algorithm

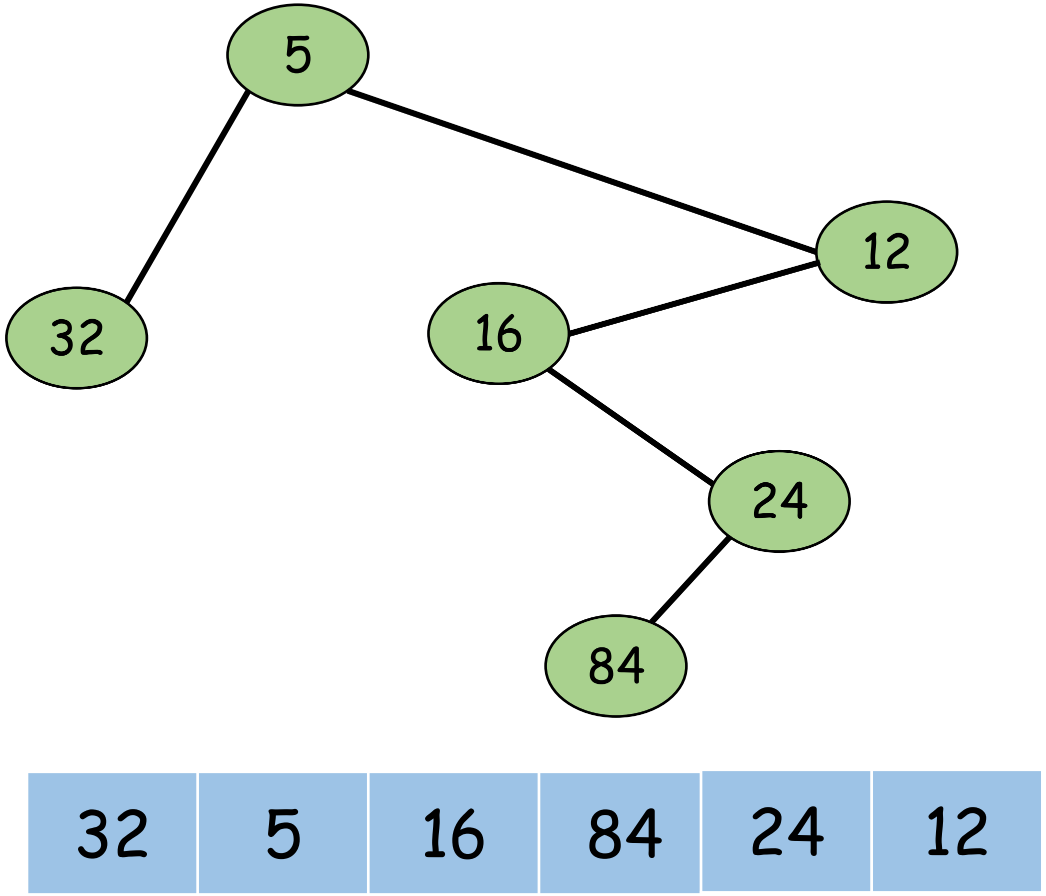
- Keep the nodes on the right spine in a stack
- Iterate over the array
  - Create a new node
  - Pop the stack till it is empty or top node is less than new node
  - Make the last node popped the left child of new node
  - If top is not null, make the new node the right child of the top
  - Push the new node into the stack
- Pop the stack till it is empty



A stack-based algorithm

- Keep the nodes on the right spine in a stack
- Iterate over the array
  - Create a new node
  - Pop the stack till it is empty or top node is less than new node
  - Make the last node popped the left child of new node
  - If top is not null, make the new node the right child of the top
  - Push the new node into the stack
- Pop the stack till it is empty

[



Analysis

- Keep the nodes on the right spine in a stack
- Iterate over the array
  - Create a new node
  - Pop the stack till it is empty or top node is less than new node
  - Make the last node popped the left child of new node
  - If top is not null, make the new node the right child of the top
  - Push the new node into the stack
- Pop the stack till it is empty

- Every node gets pushed and popped once
- So, number of stack operations per node is constant
- All other operation per iteration takes constant time
- RT: O(n)

Questions we were pondering…

If X ~ Y, then they can share the same RMQ structure

How to efficiently find that X ~ Y?

How many types are there?

How to efficiently build a Cartesian tree?

How to efficiently check whether two Cartesian trees are isomorphic?

Cartesian Number

- Encode the operations on the stack by the algorithm
- Every push is encoded as 1 and every pop is encoded a 0
- As there are 2b operations on the stack, we need 2b bits
- Construction of a tree by the algorithm is encoded by this 2b bit number. This number is called Cartesian number
- If two runs of the algorithm on two different block produces the same Cartesian number, then the two blocks can share a RMQ structure

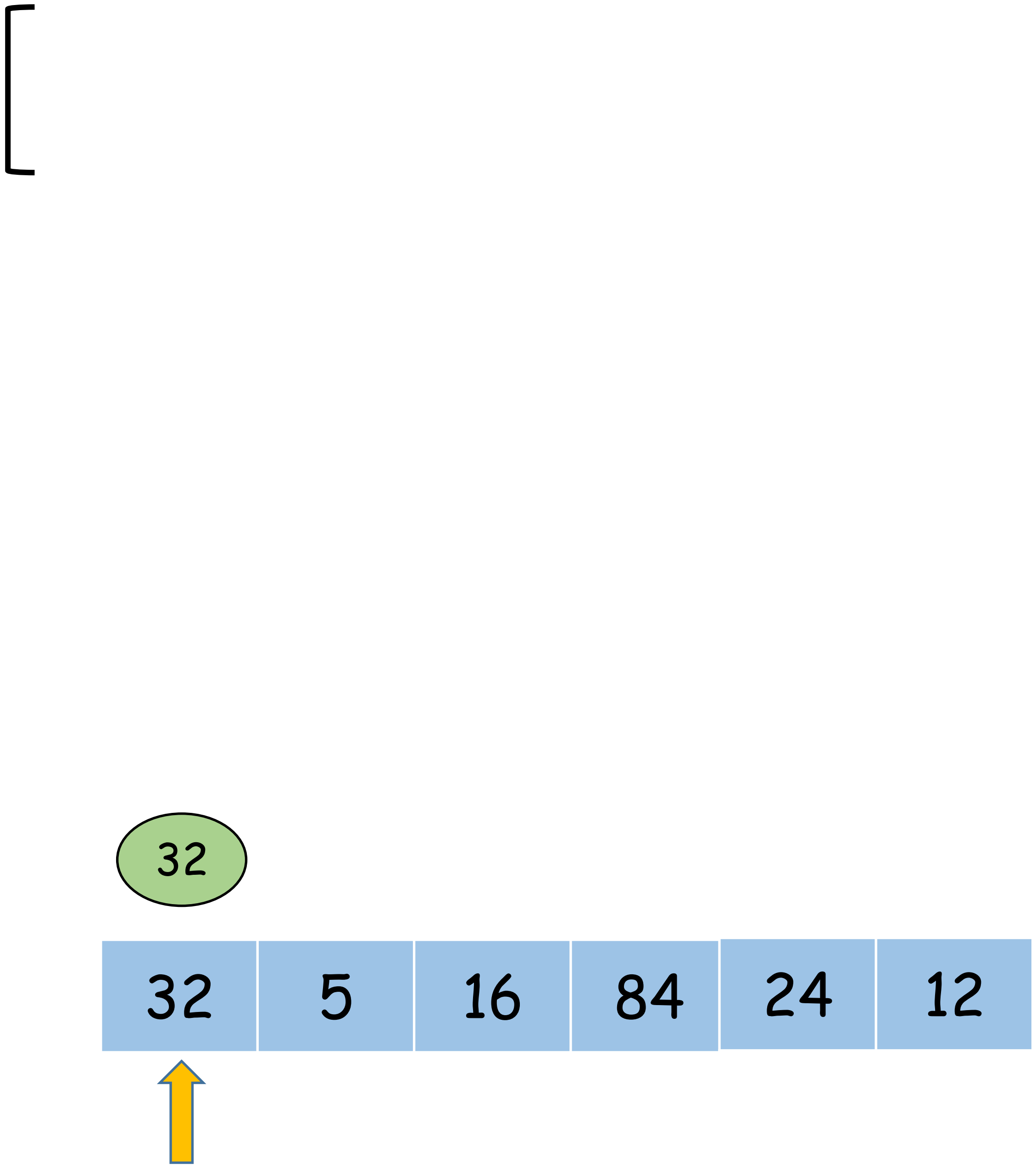
A stack-based algorithm

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

32	5	16	84	24	12

A stack-based algorithm [redacted]

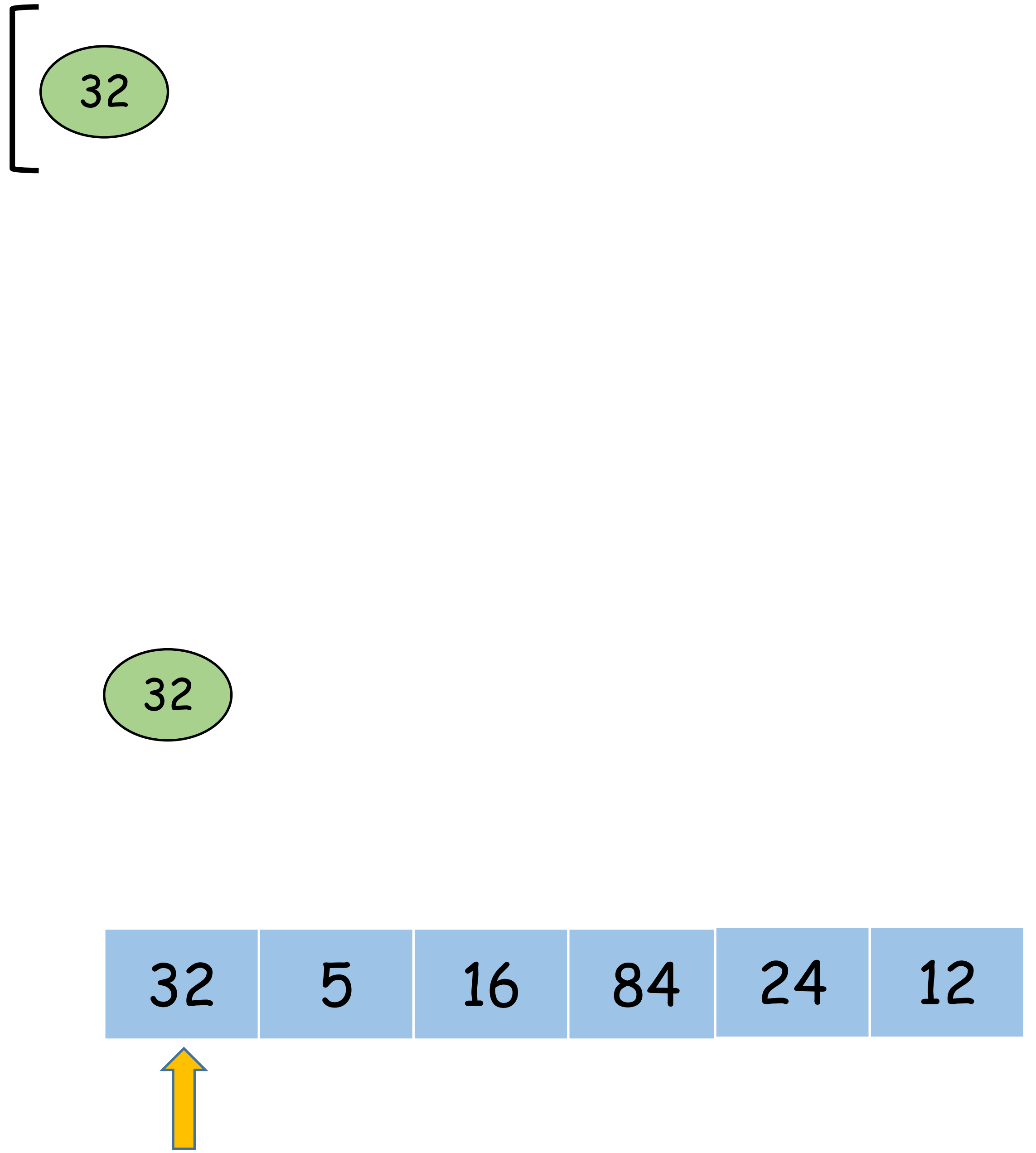
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



A stack-based algorithm

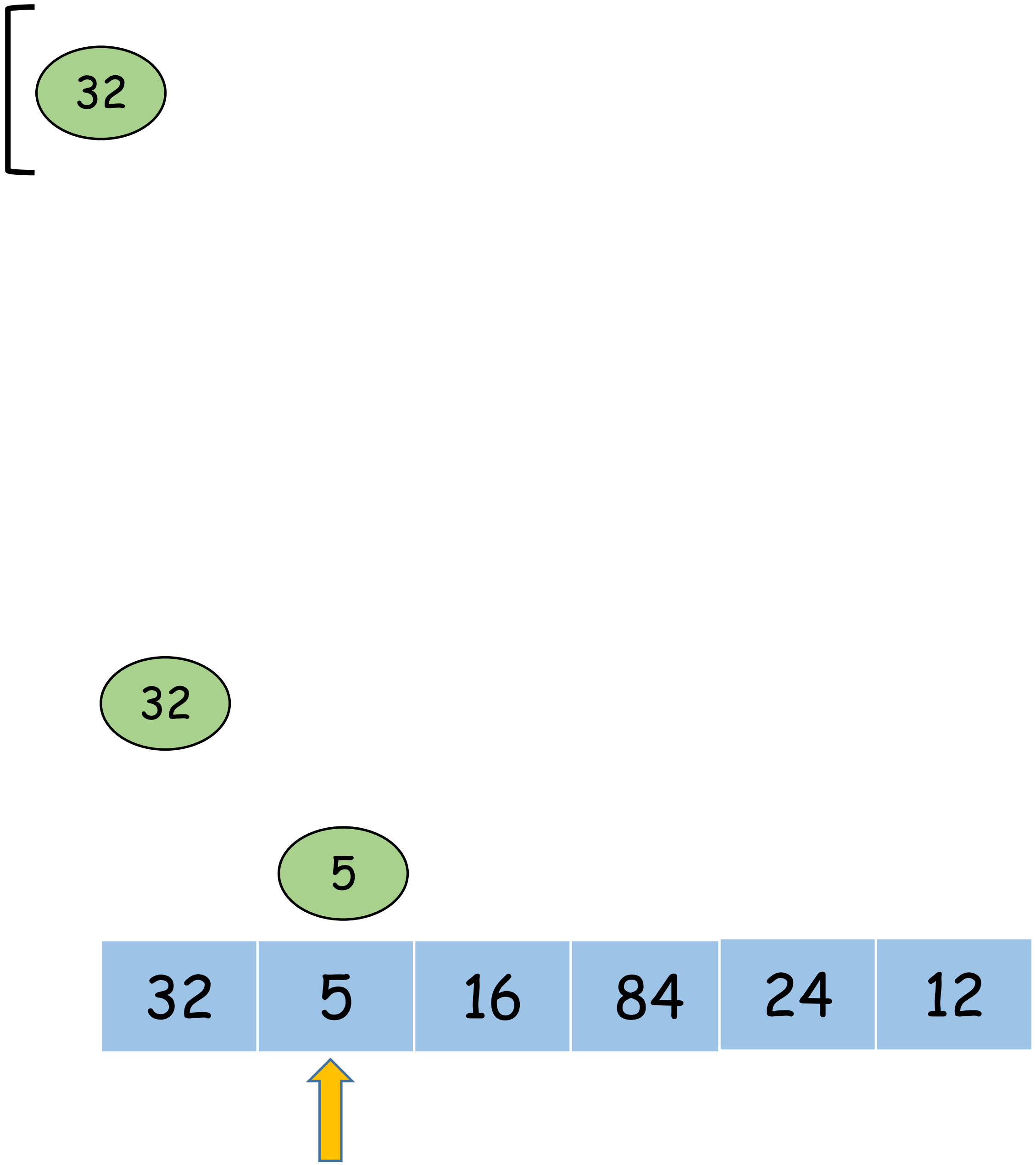
1

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



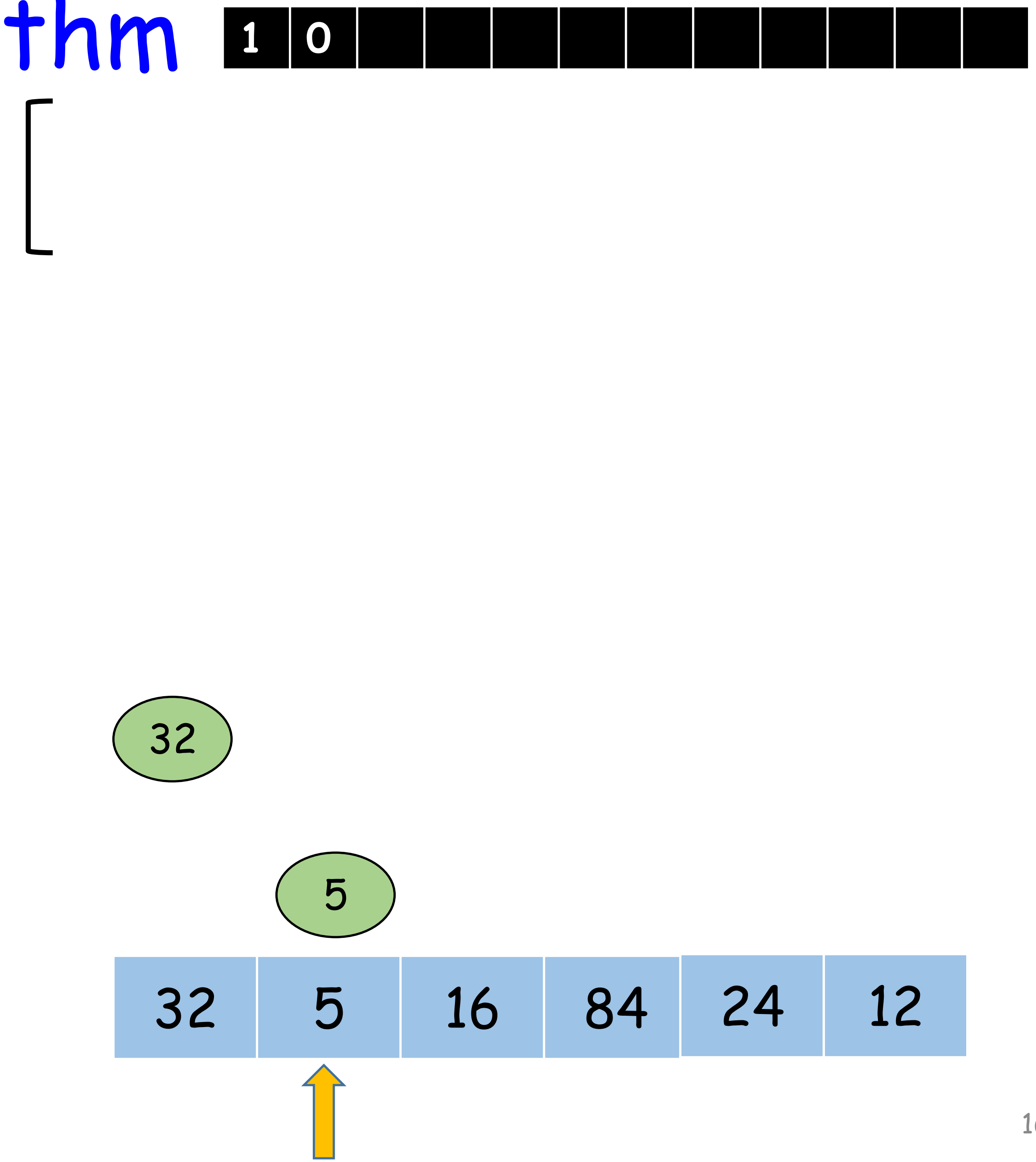
A stack-based algorithm

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

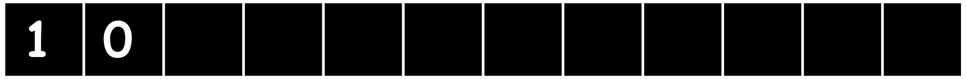


A stack-based algorithm

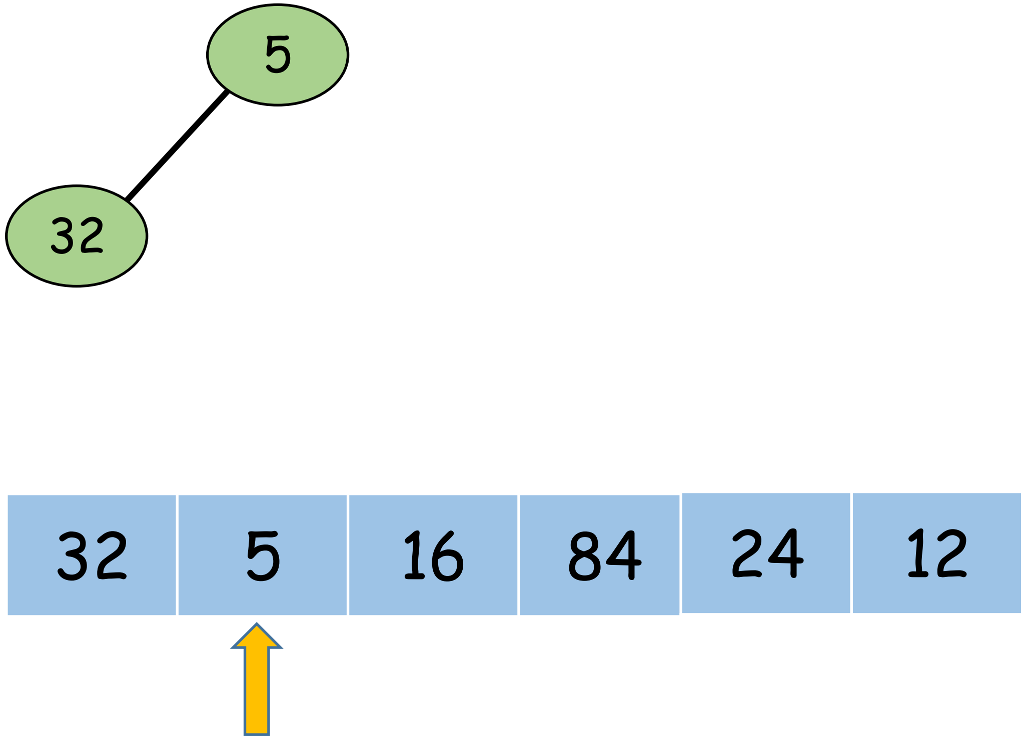
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



A stack-based algorithm



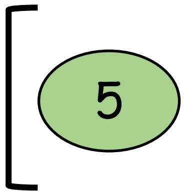
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

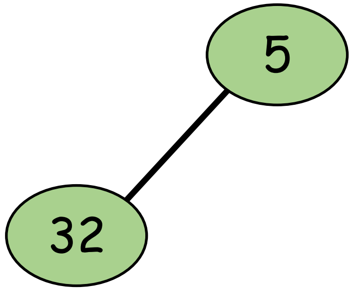


A stack-based algorithm

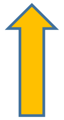
1 0 1

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack





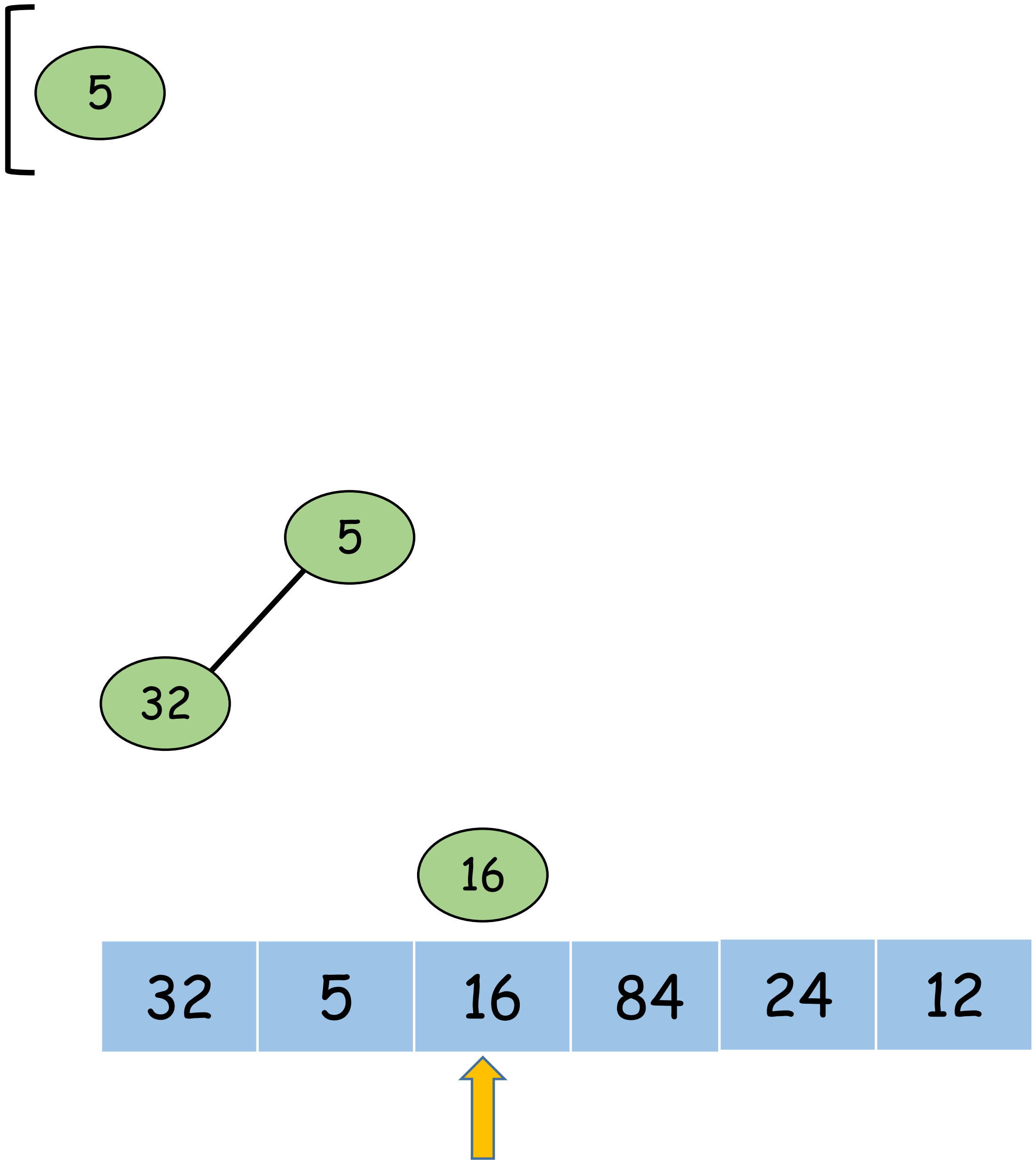
32	5	16	84	24	12



A stack-based algorithm

1 0 1

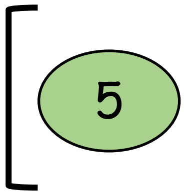
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

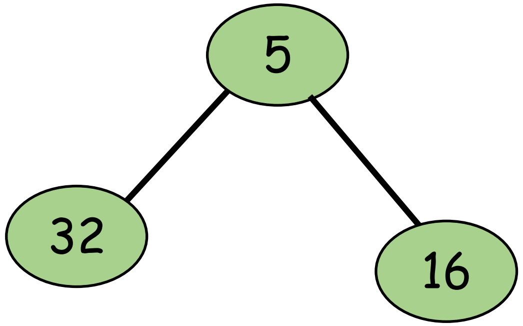


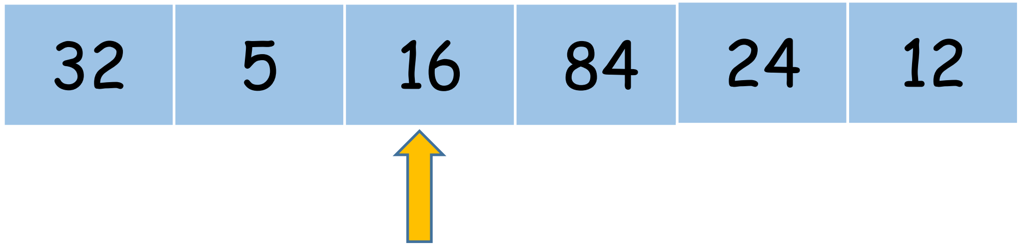
A stack-based algorithm

1 0 1

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



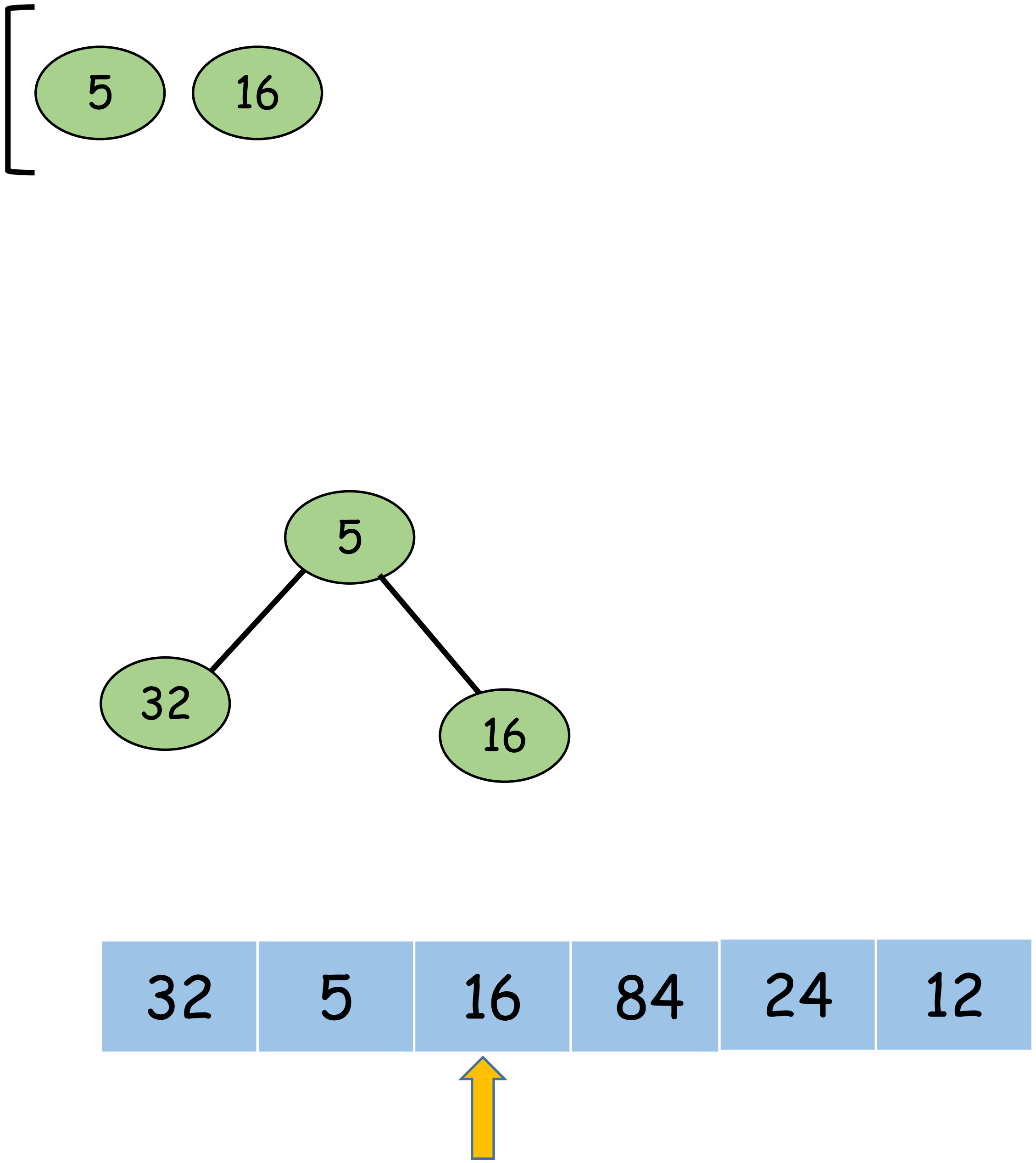




A stack-based algorithm

1	0	1	1								

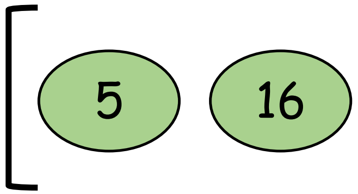
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

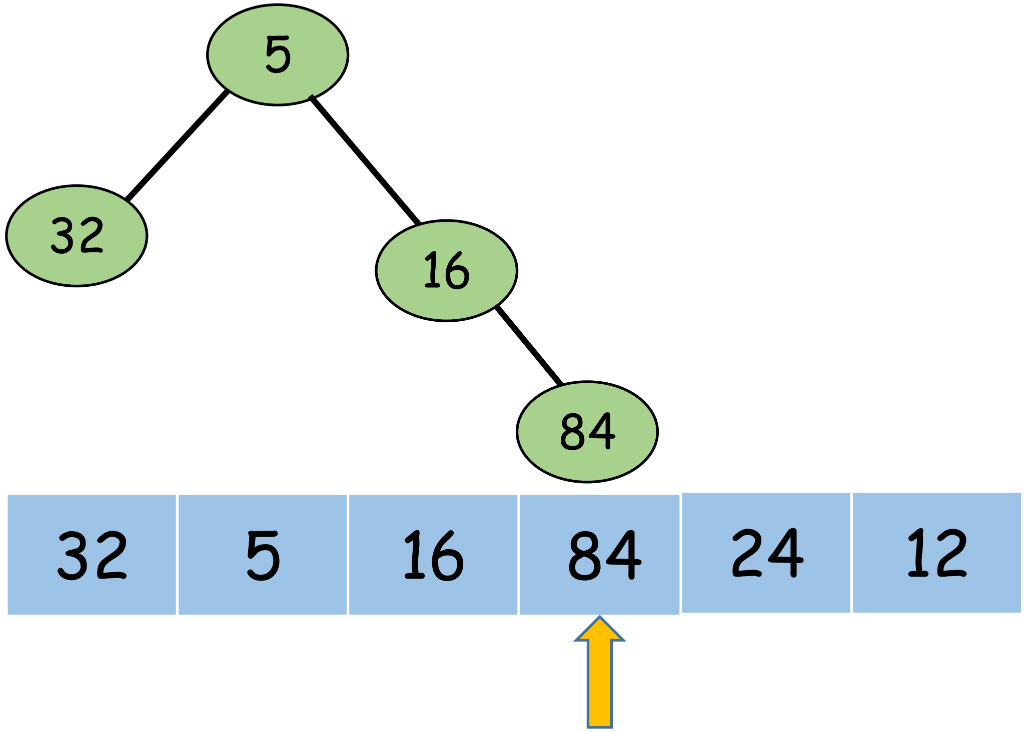


A stack-based algorithm

1 0 1 1

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

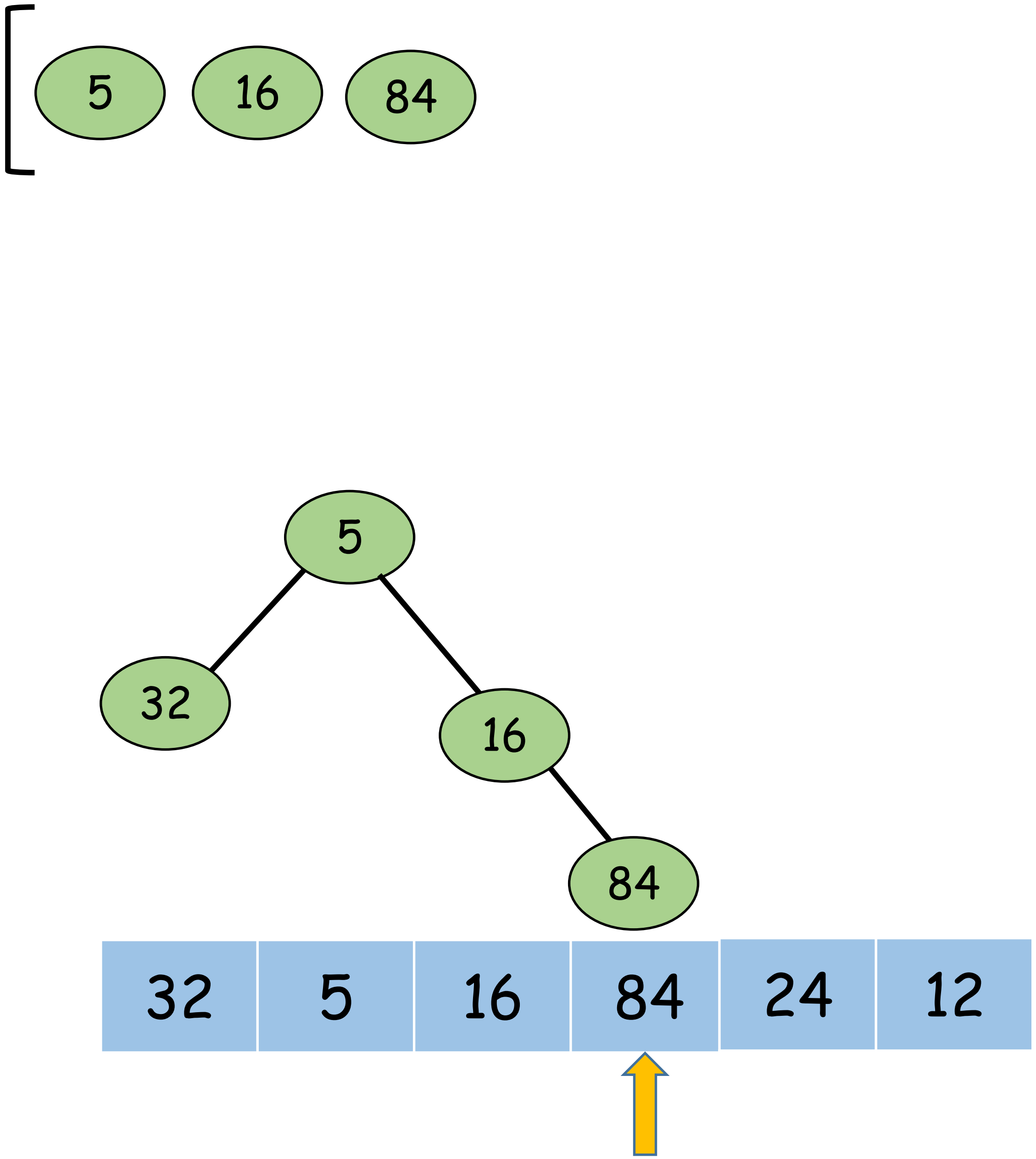




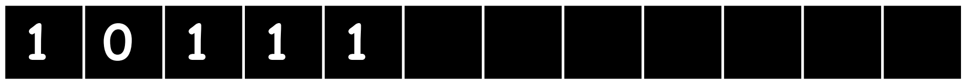
A stack-based algorithm

1 0 1 1 1

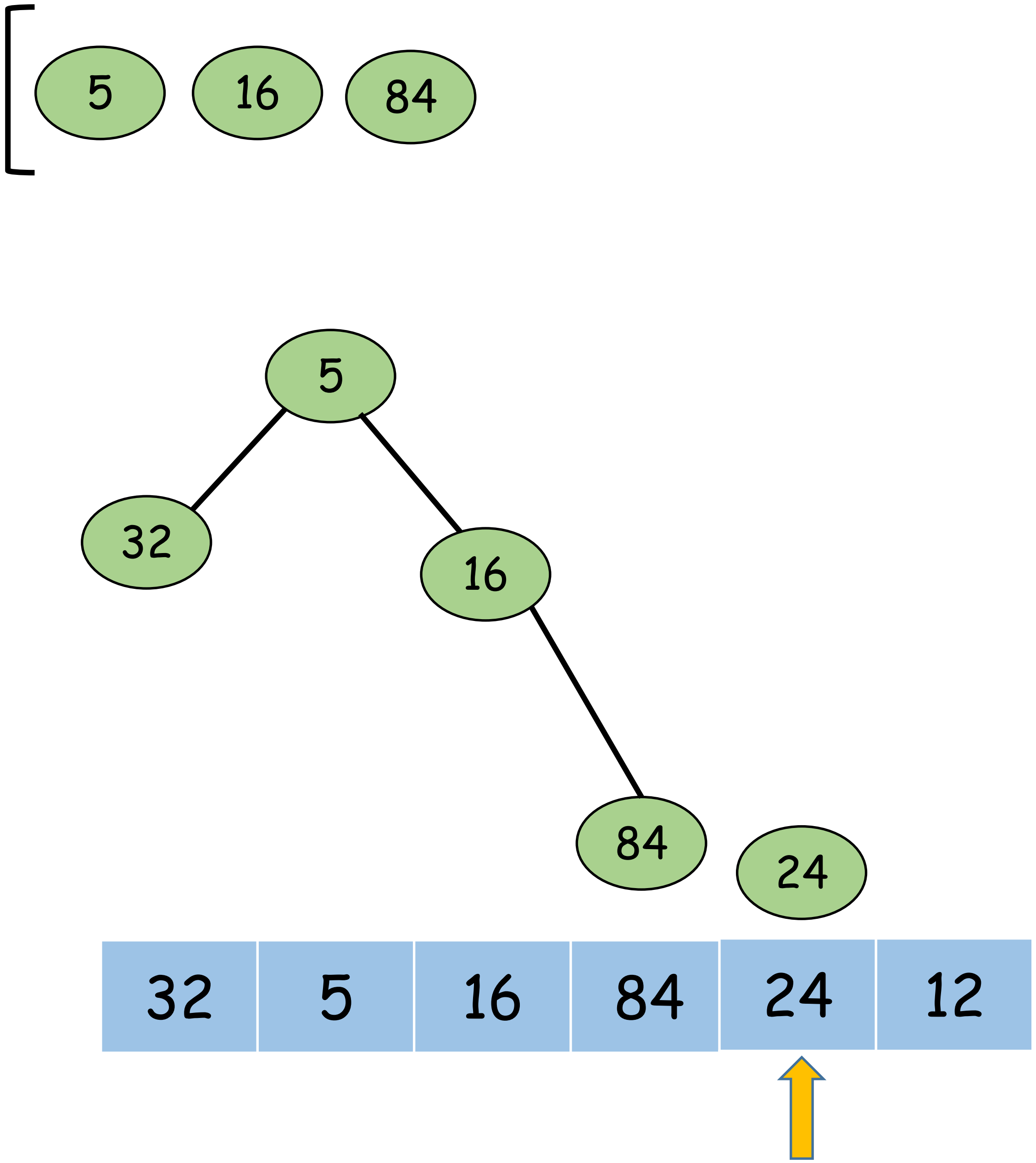
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



A stack-based algorithm



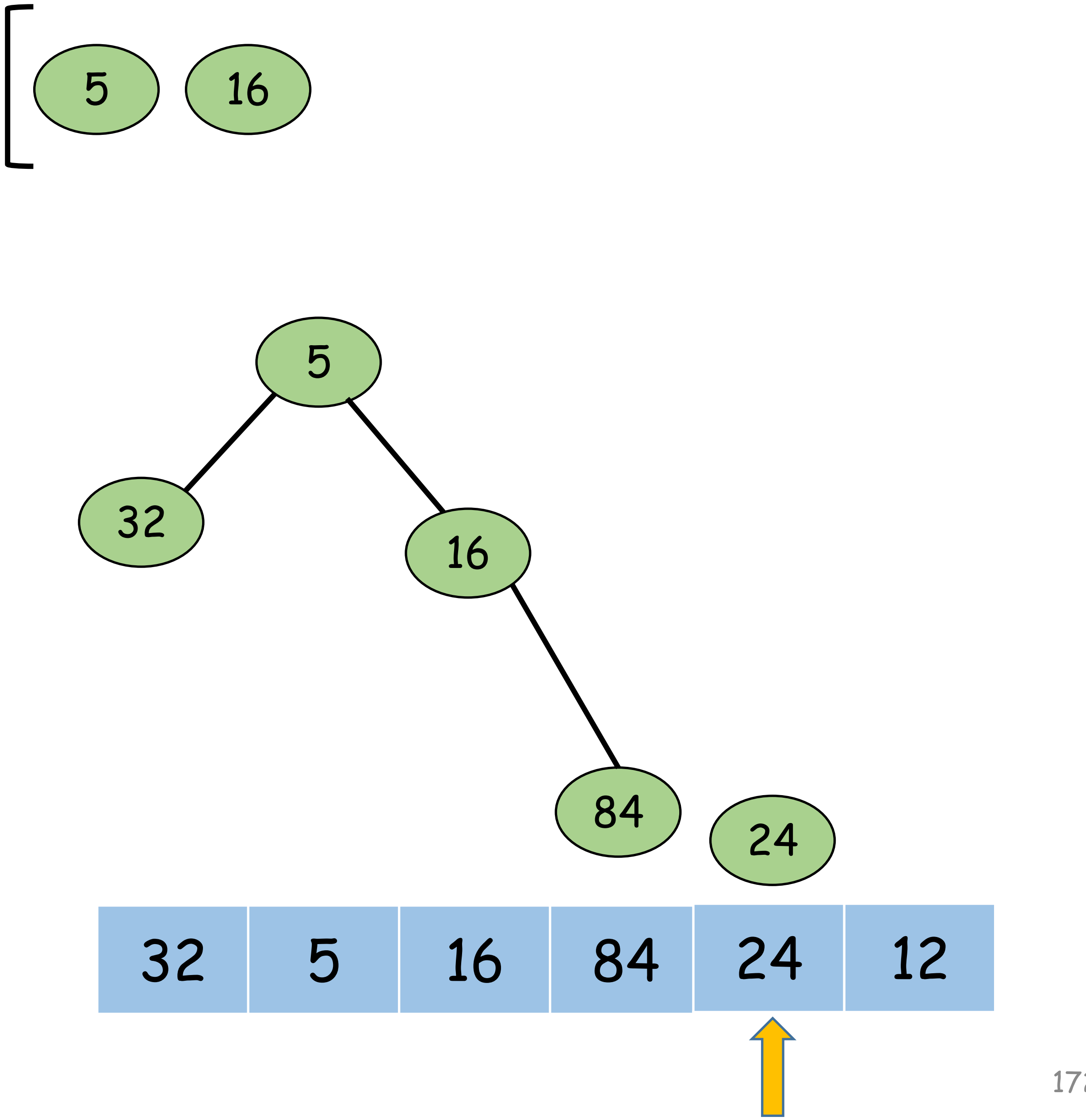
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



A stack-based algorithm

1 0 1 1 1 0

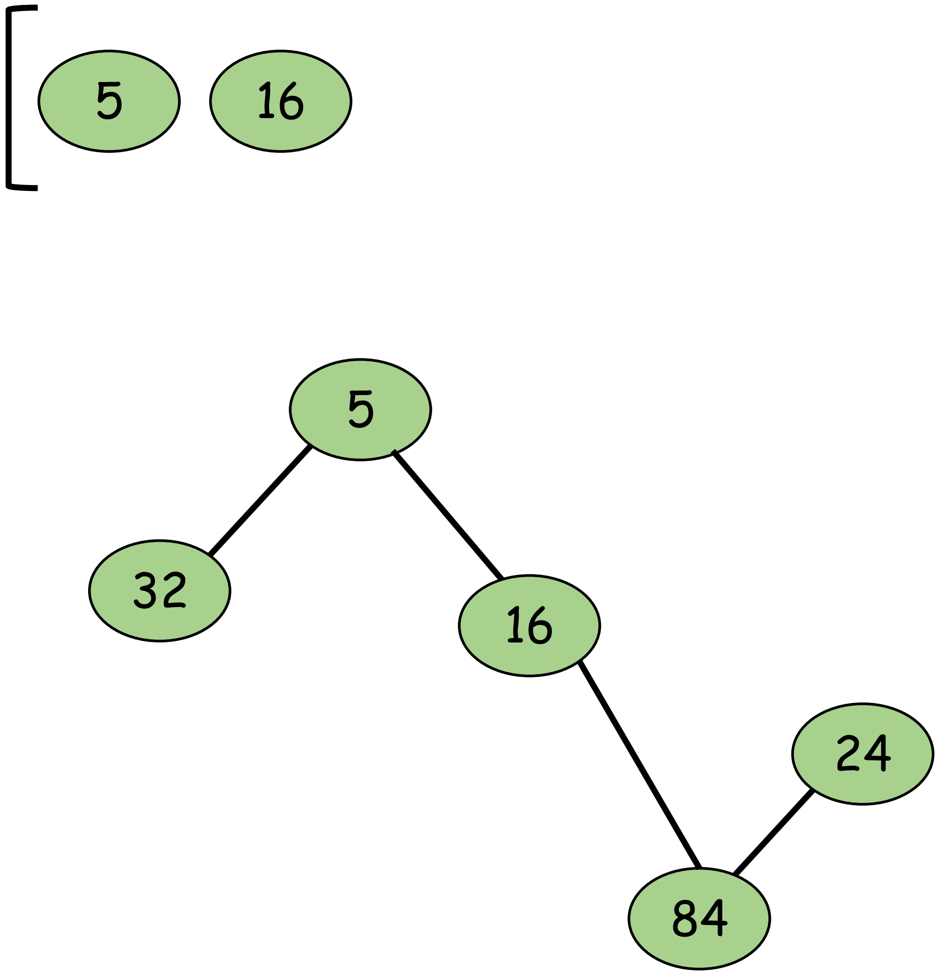
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



A stack-based algorithm

1	0	1	1	1	0						

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

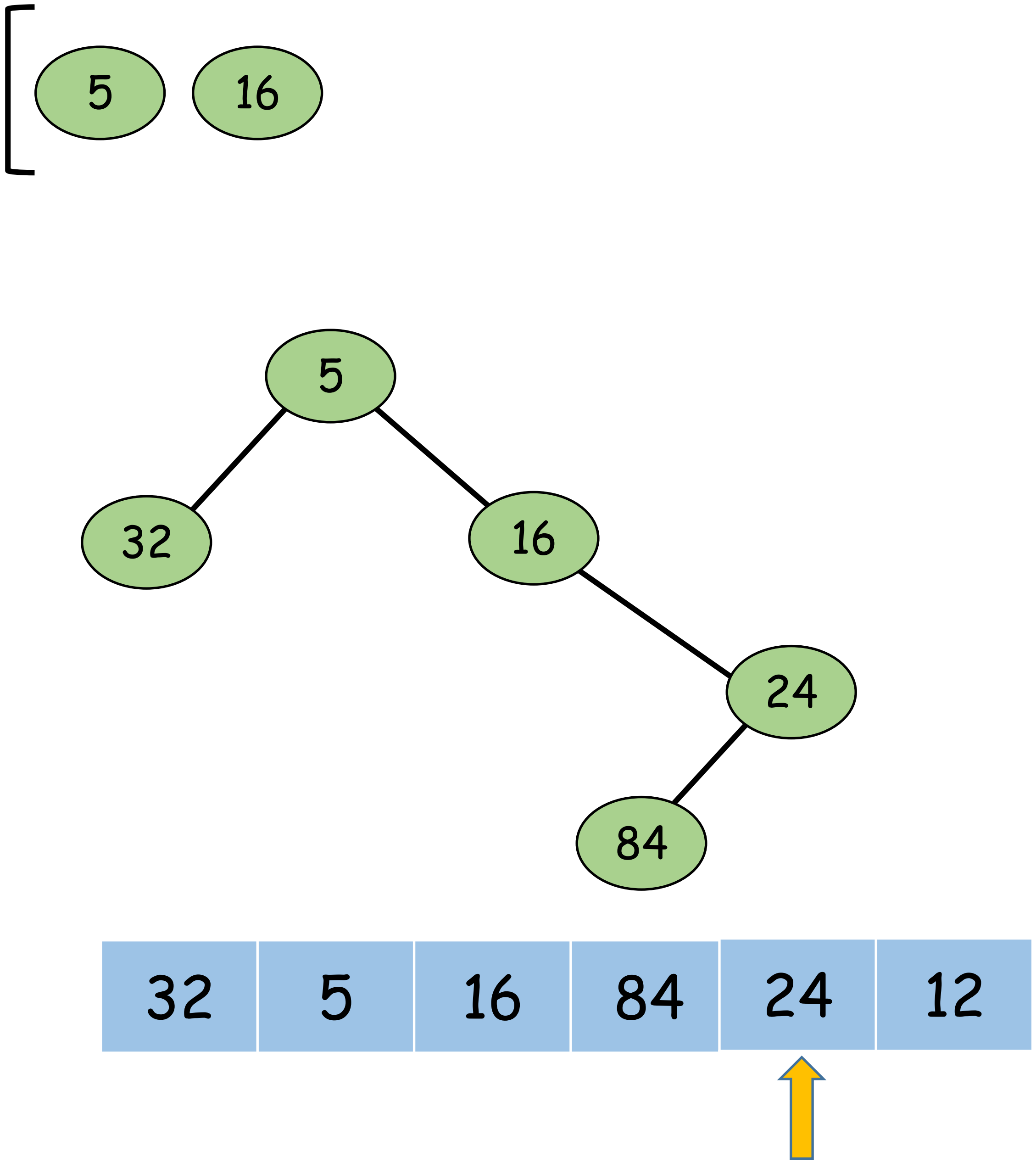


32	5	16	84	24	12

A stack-based algorithm

1	0	1	1	1	0						

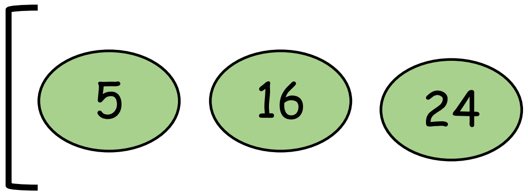
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

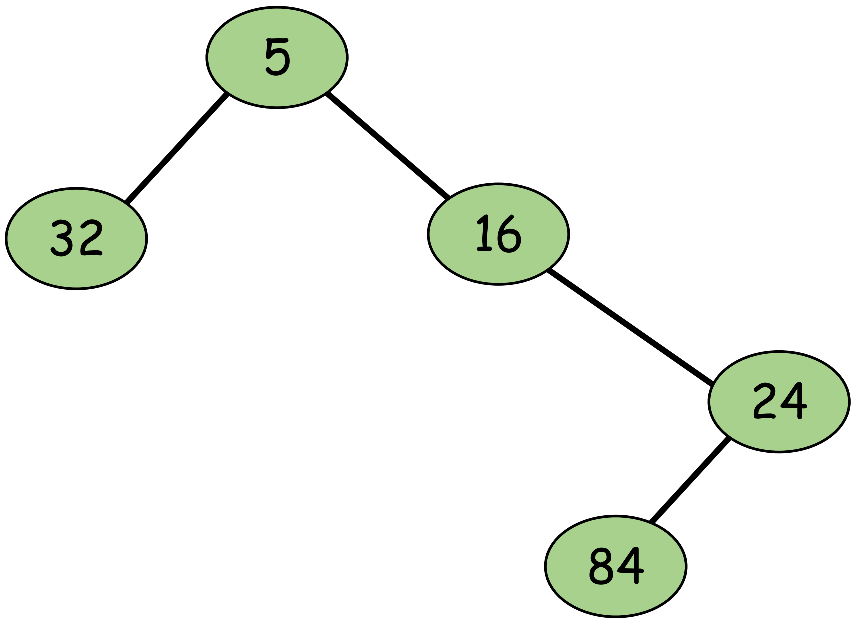


A stack-based algorithm

1	0	1	1	1	0	1					

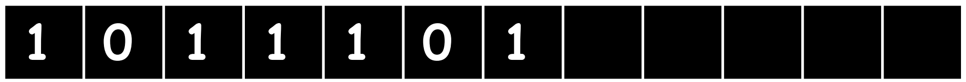
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



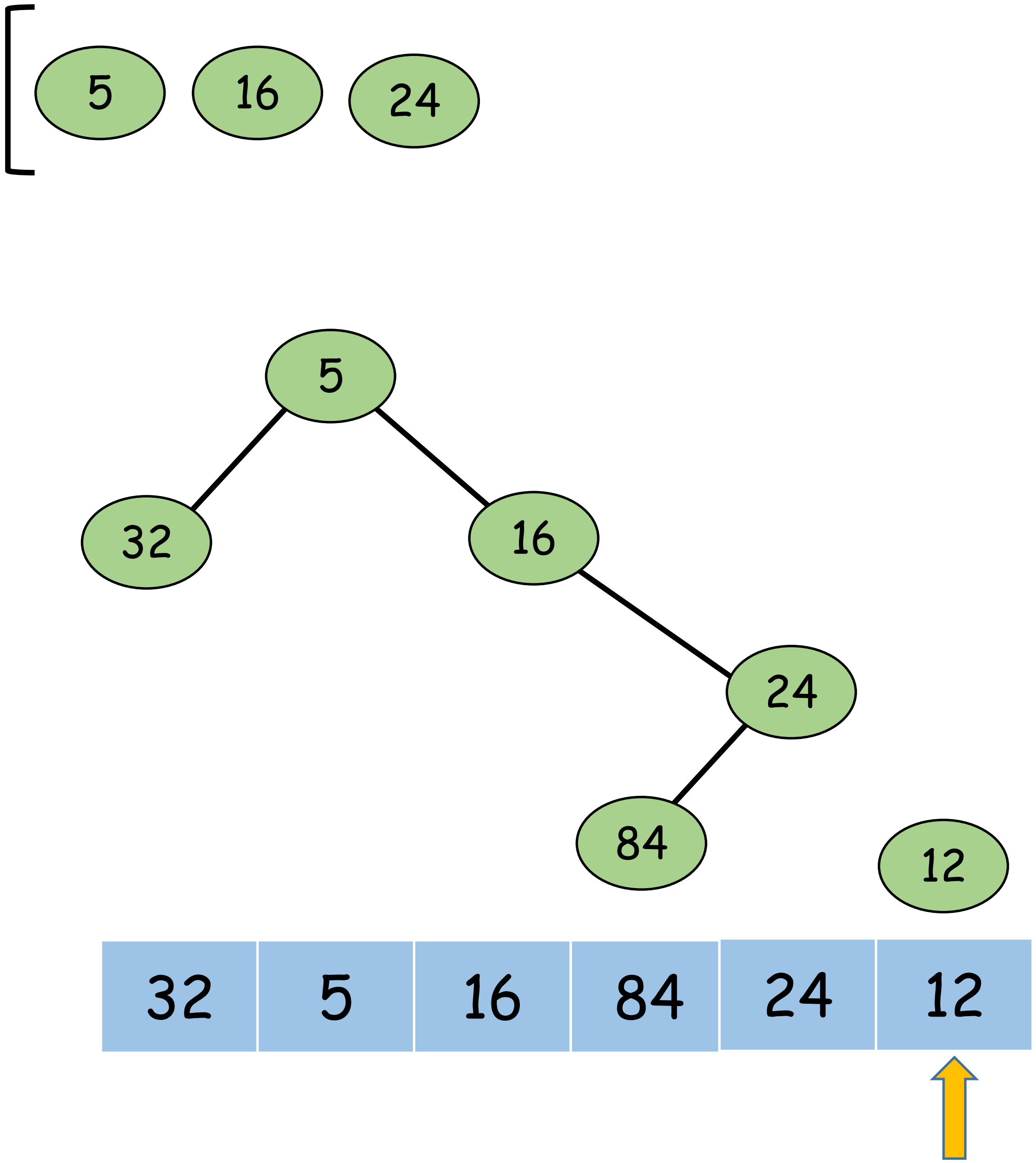


32	5	16	84	24	12

A stack-based algorithm



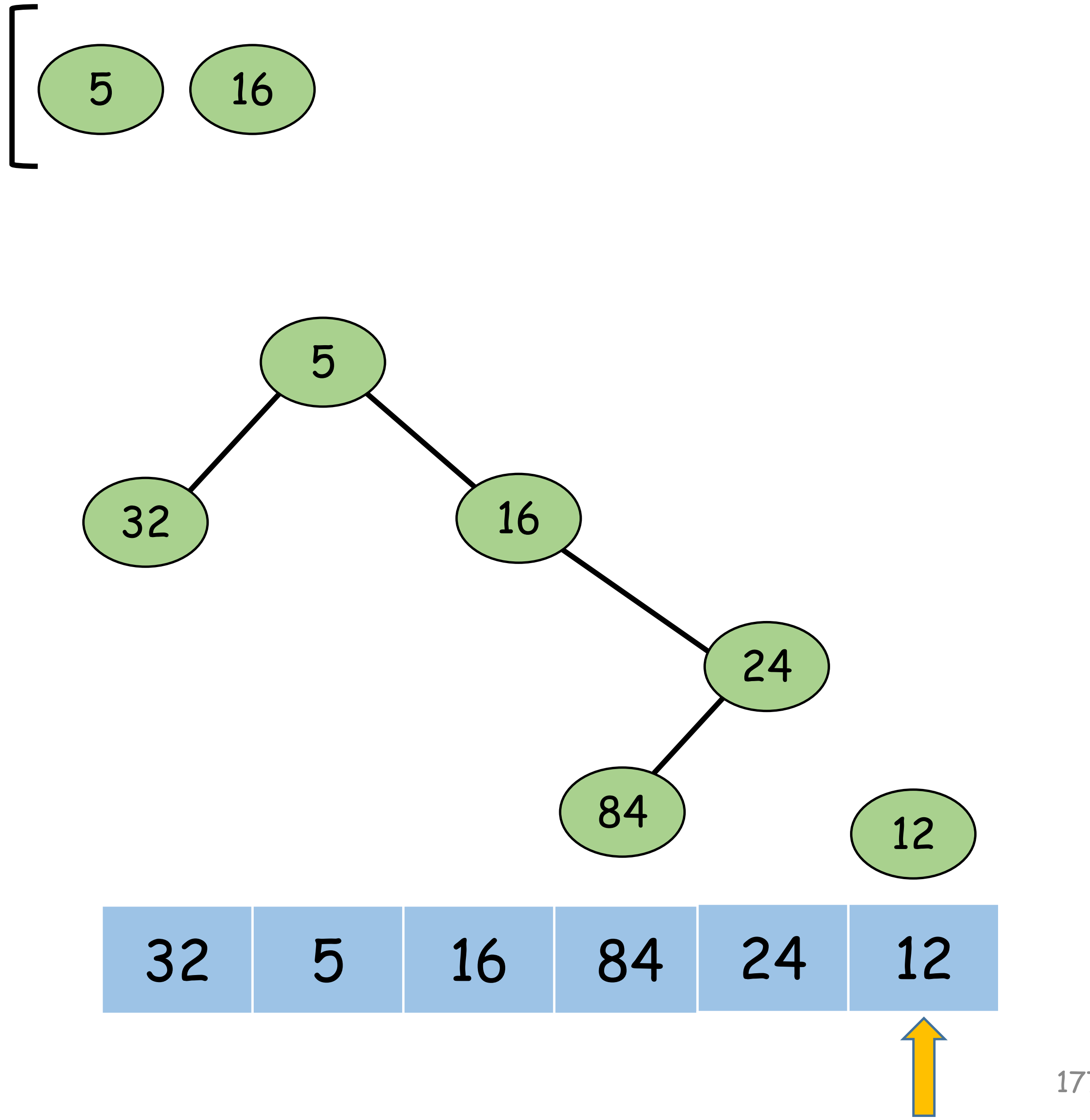
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



A stack-based algorithm

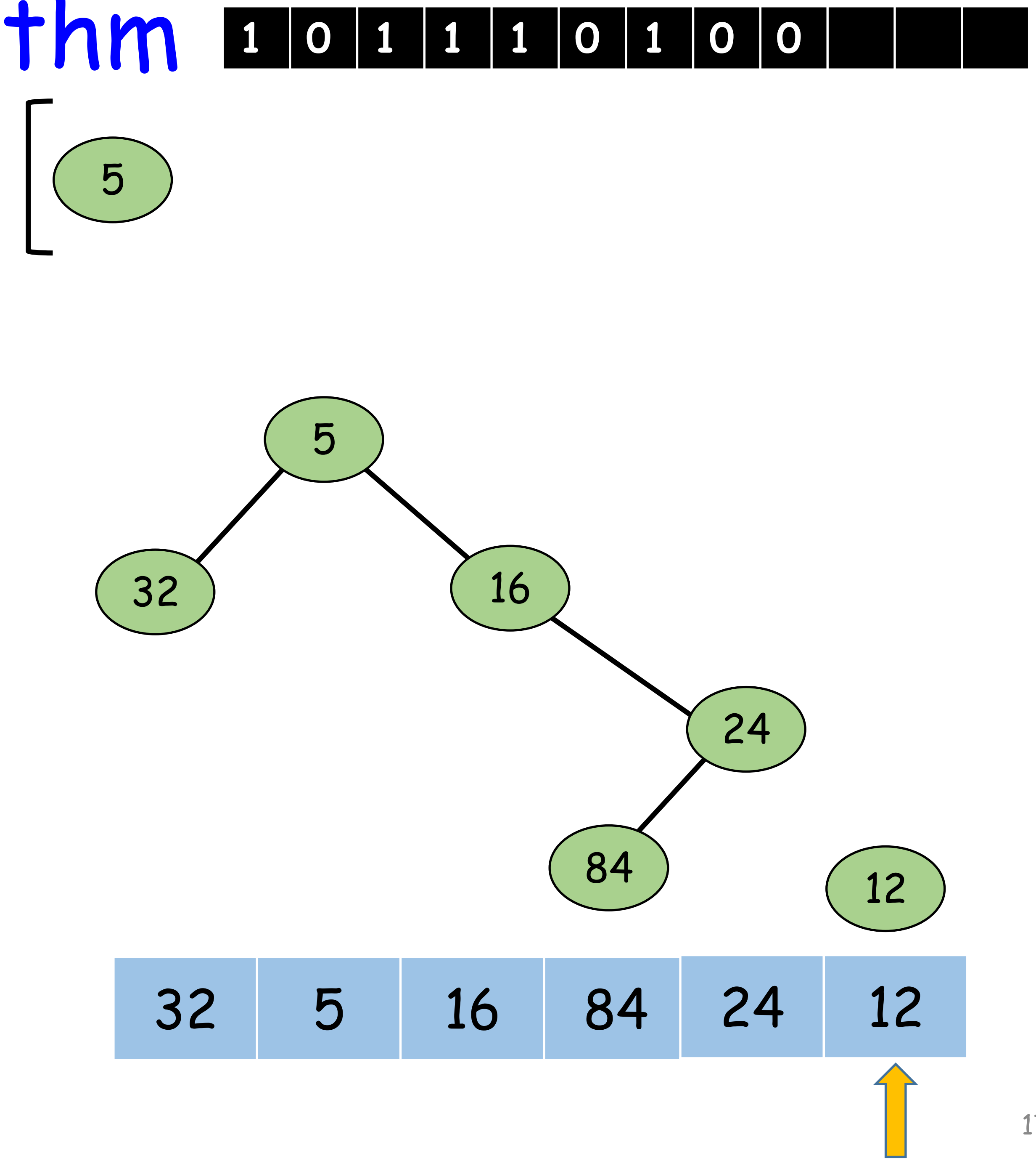
1 0 1 1 1 0 1 0

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



A stack-based algorithm

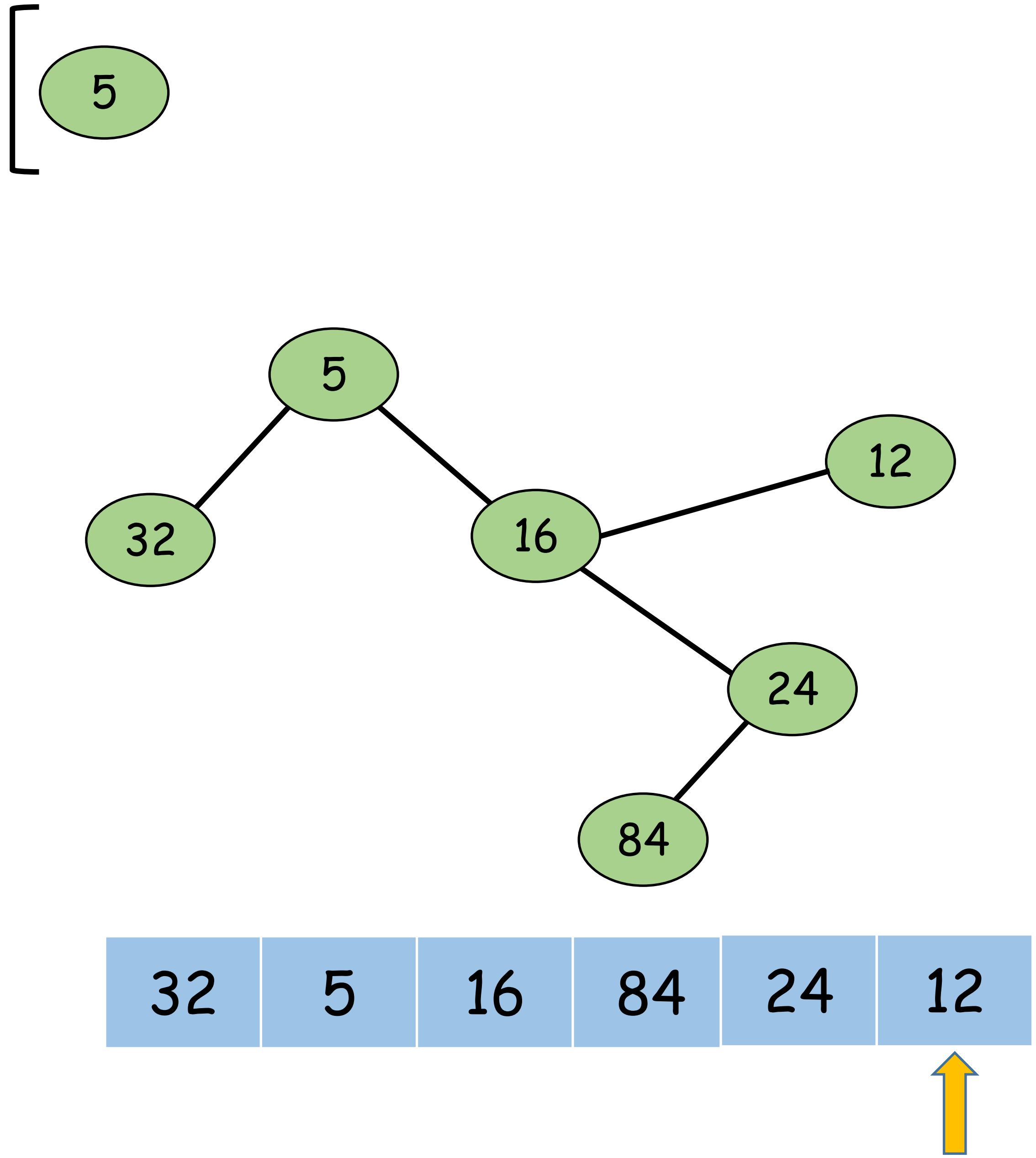
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



A stack-based algorithm

1	0	1	1	1	0	1	0	0			

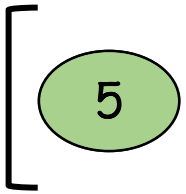
- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

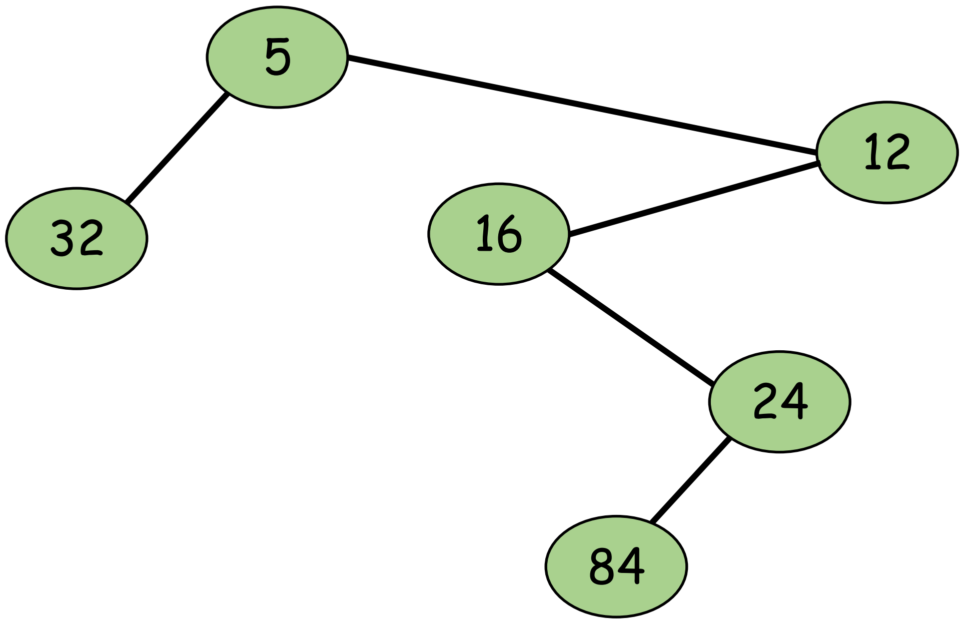


A stack-based algorithm

1	0	1	1	1	0	1	0	0			

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack



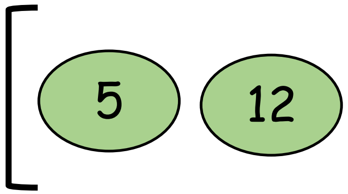


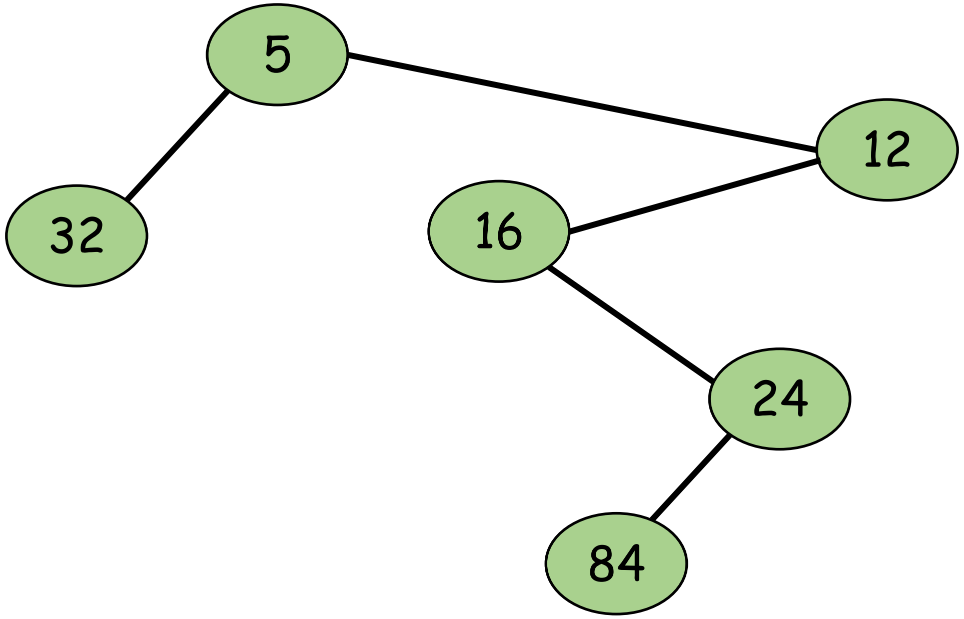
32	5	16	84	24	12

A stack-based algorithm

1	0	1	1	1	0	1	0	0	1		

- Keep the nodes on the right spine in a stack
- Iterate over the array
- Create a new node
- Pop the stack till it is empty or top node is less than new node
- Make the last node popped the left child of new node
- If top is not null, make the new node the right child of the top
- Push the new node into the stack

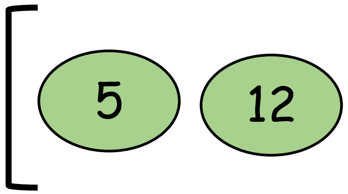




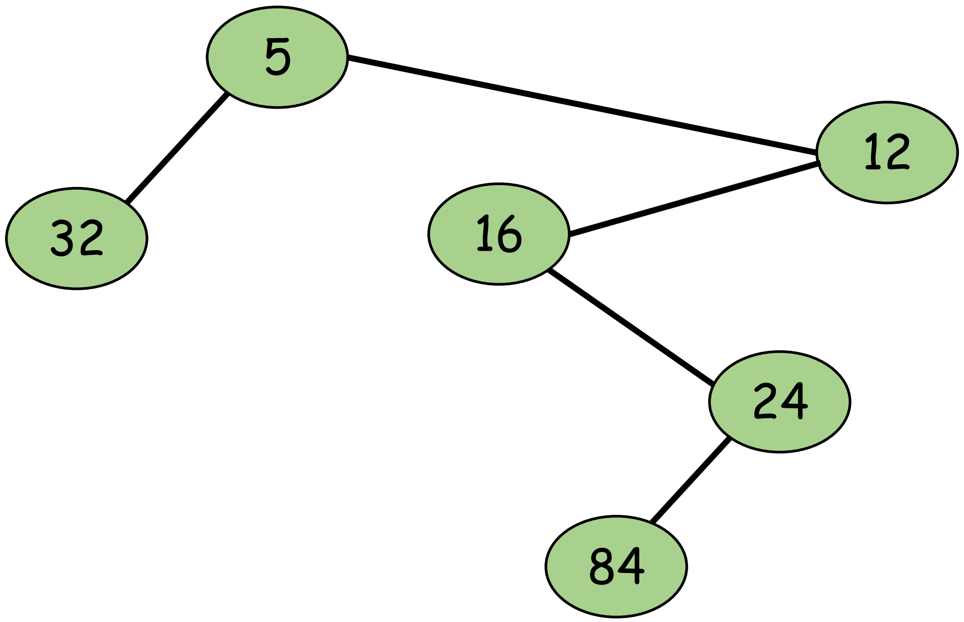
32	5	16	84	24	12

A stack-based algorithm

1	0	1	1	1	0	1	0	0	1		



- Keep the nodes on the right spine in a stack
- Iterate over the array
  - Create a new node
  - Pop the stack till it is empty or top node is less than new node
  - Make the last node popped the left child of new node
  - If top is not null, make the new node the right child of the top
  - Push the new node into the stack
- Pop the stack till it is empty

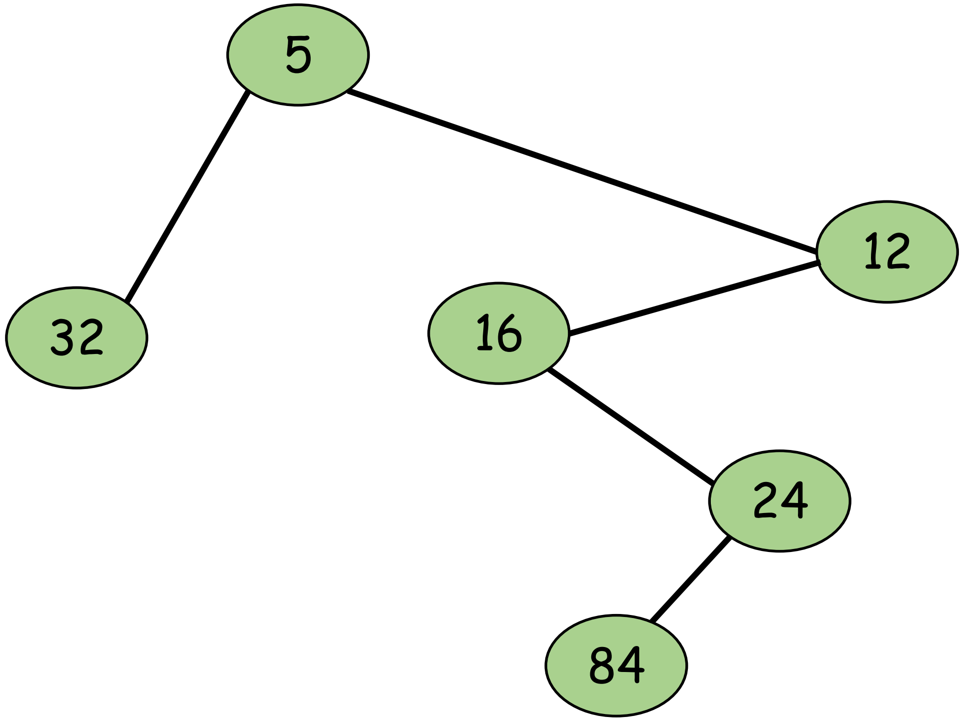


32	5	16	84	24	12

A stack-based algorithm

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

- Keep the nodes on the right spine in a stack
- Iterate over the array
  - Create a new node
  - Pop the stack till it is empty or top node is less than new node
  - Make the last node popped the left child of new node
  - If top is not null, make the new node the right child of the top
  - Push the new node into the stack
- Pop the stack till it is empty



32	5	16	84	24	12

A stack-based algorithm

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

- Keep the nodes on the right spine in a stack

[

- Iterate over the array
  - Create a new node
  - Pop the stack till it is empty or top node is less than new node
  - ~~Make the last node popped the left child of new node~~
  - ~~If top is not null, make the new node the right child of the top~~
  - Push the new node into the stack
- Pop the stack till it is empty

32	5	16	84	24	12

Questions we were pondering…

If X ~ Y, then they can share the same RMQ structure

How to efficiently find that X ~ Y?

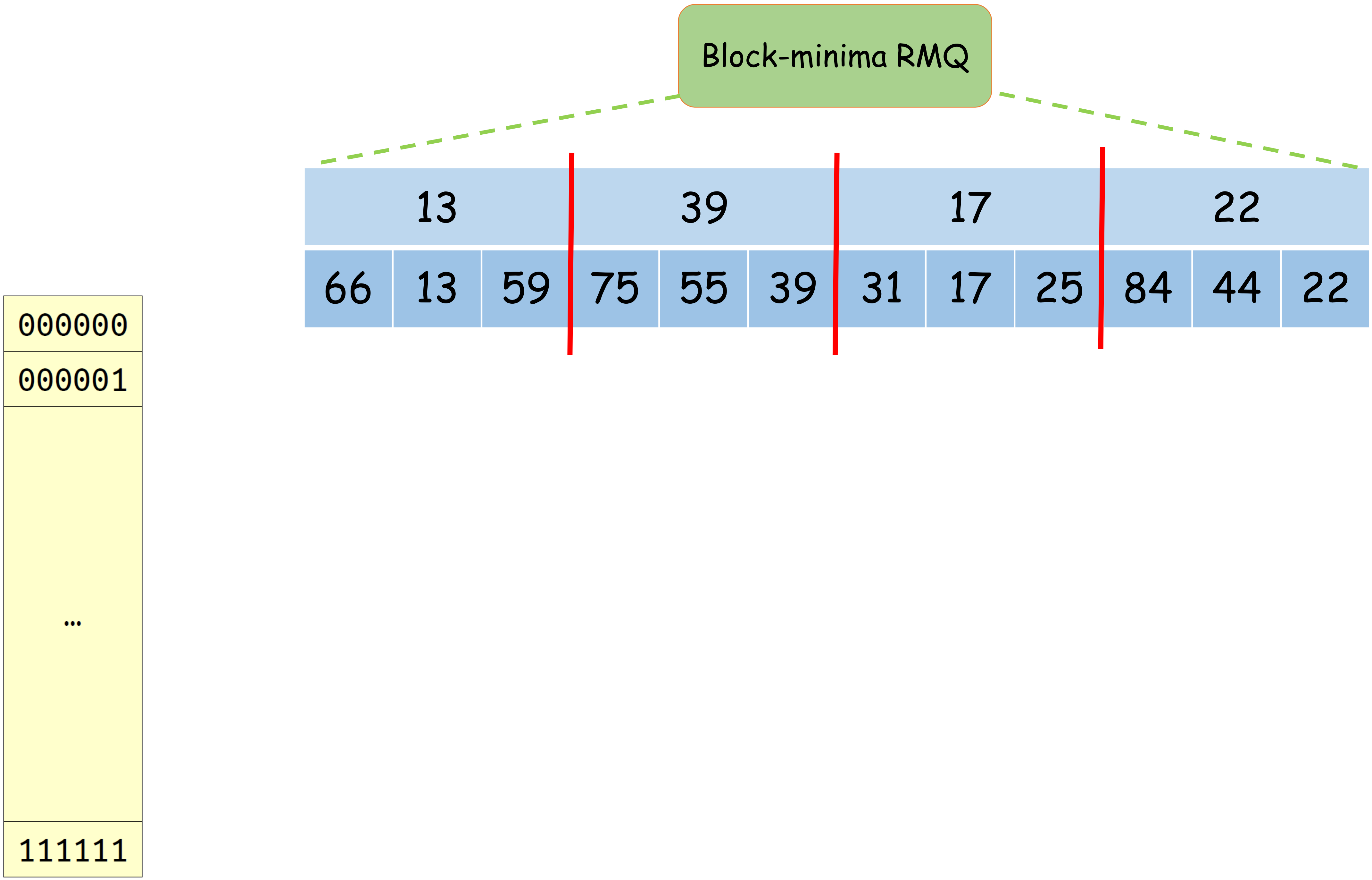
How many types are there?

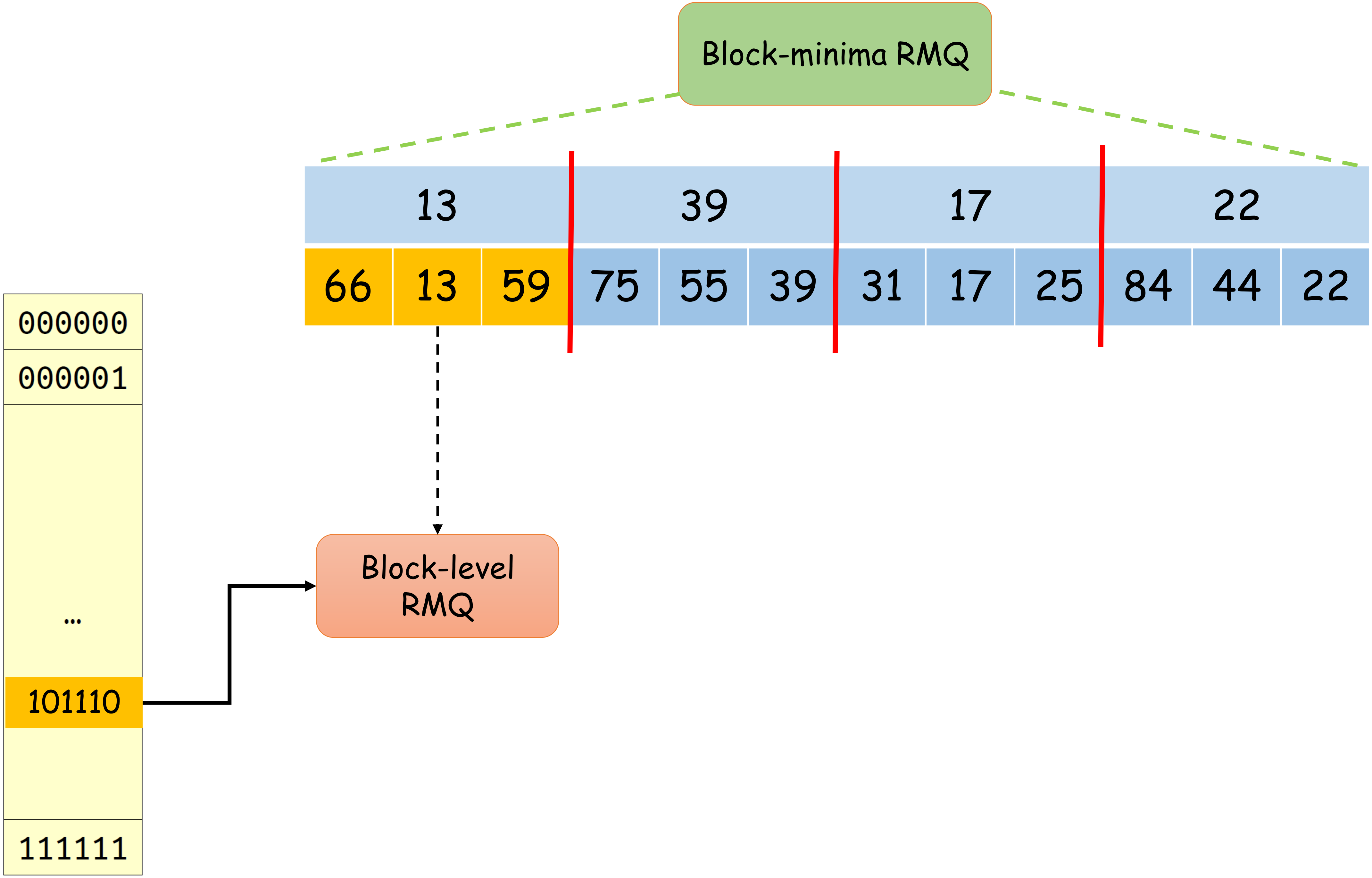
How to efficiently build a Cartesian tree?

How to efficiently check whether two Cartesian trees are isomorphic?

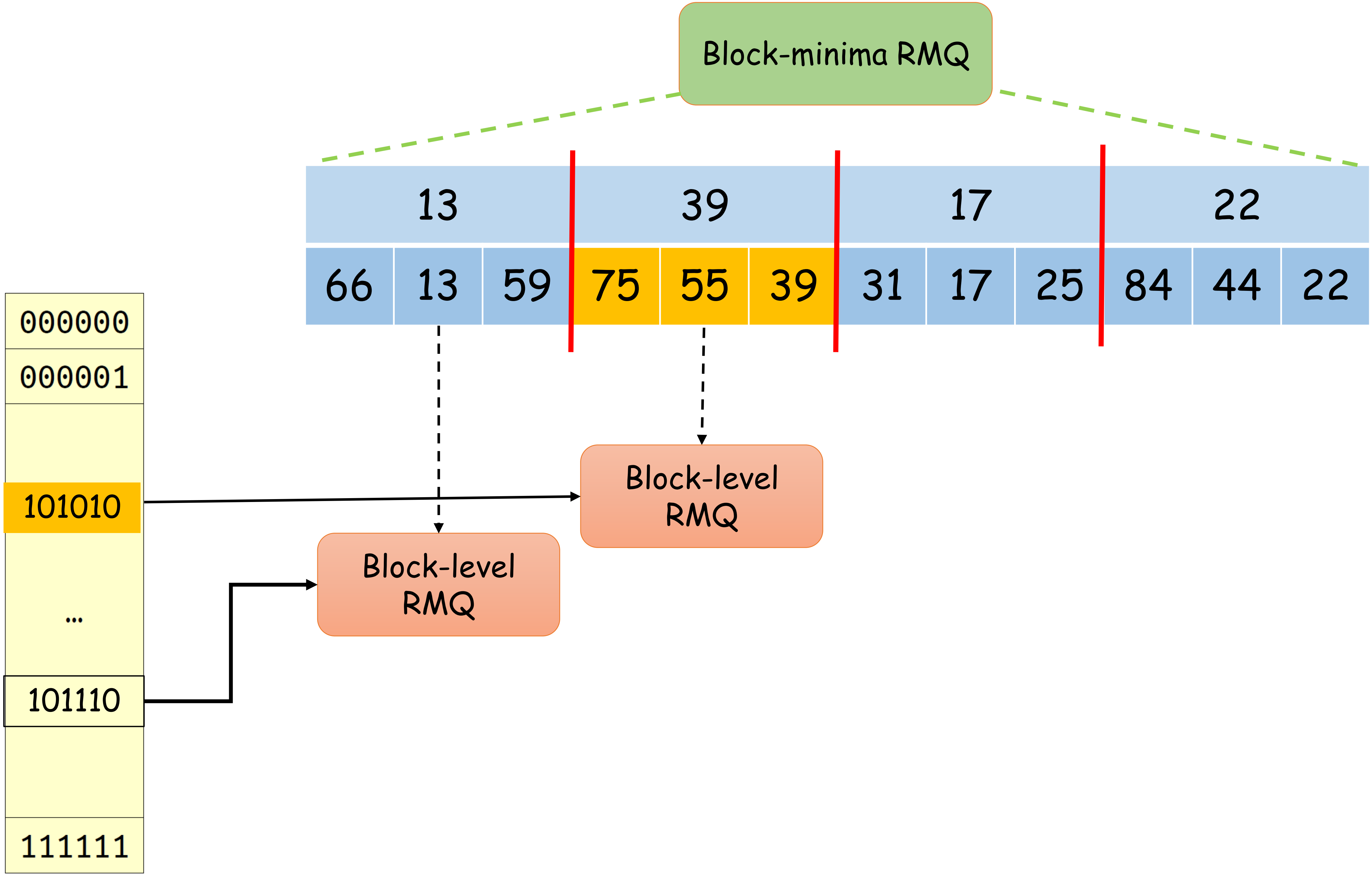
How many types of blocks are there?

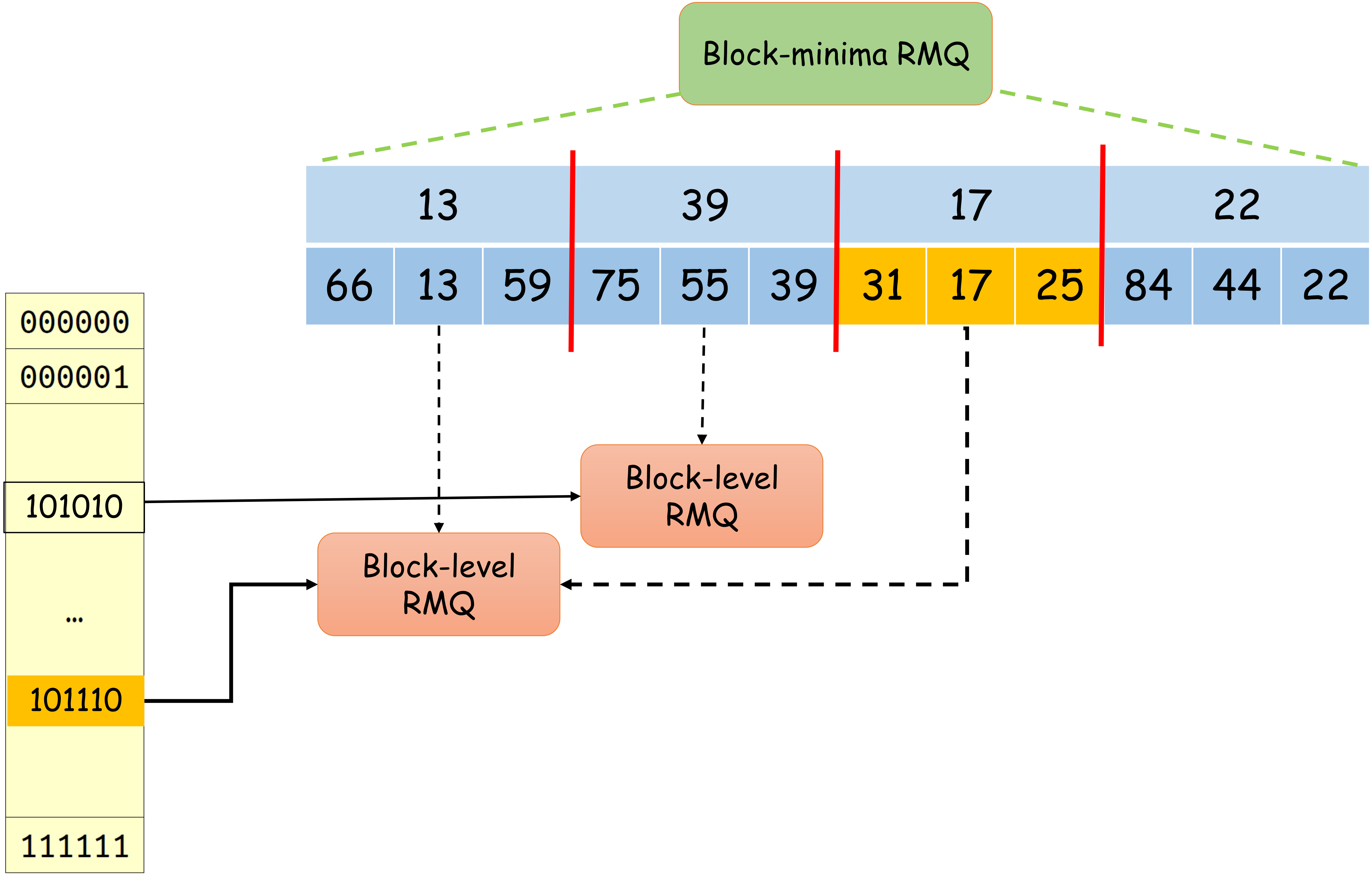
- Given block size b
- Number of block types = Number of Cartesian numbers
- Length of Cartesian number is 2b bits
- Number of Cartesian numbers = $2^{2b} = 4^b$

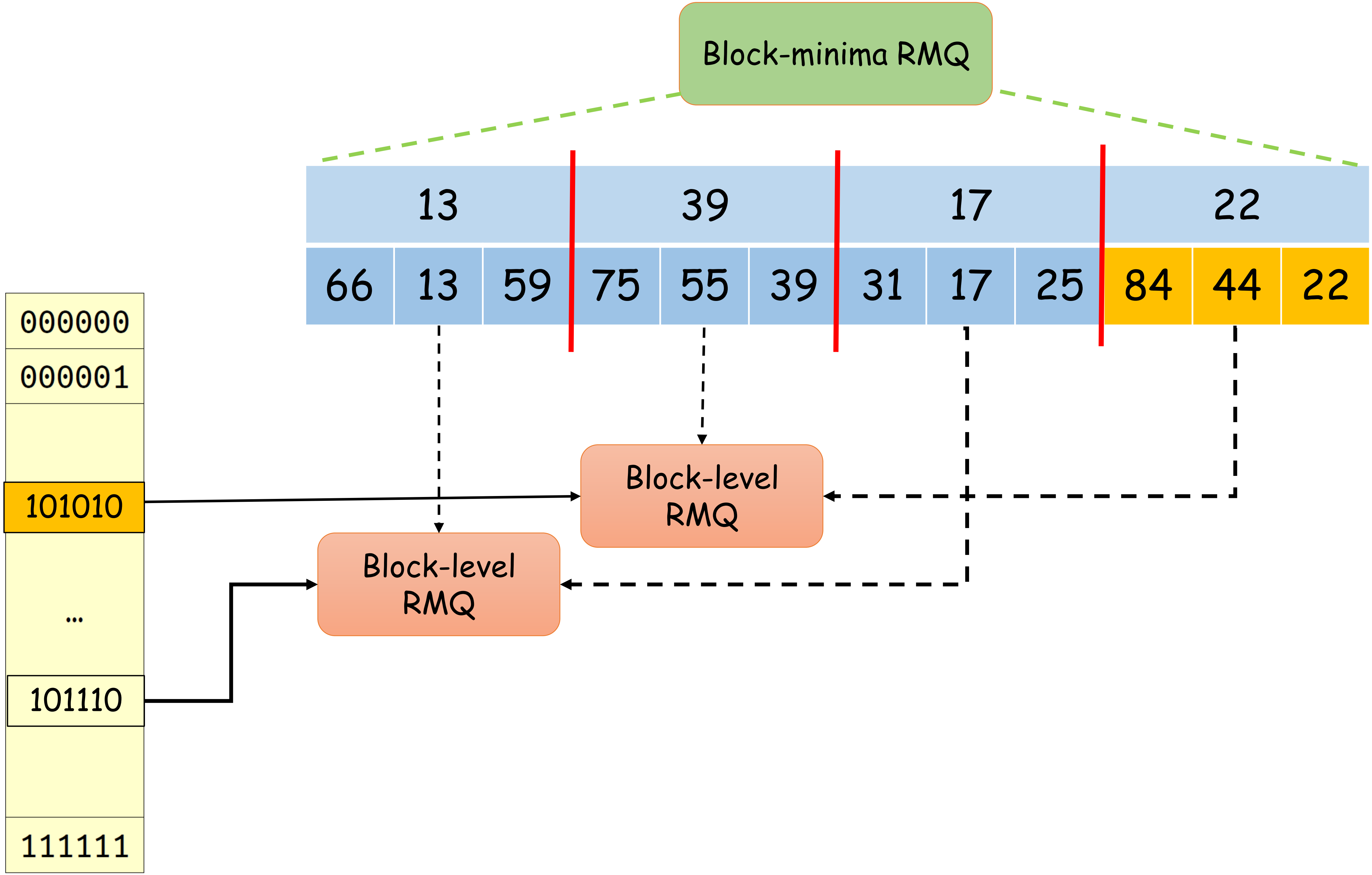

Block-minima RMQ
13
39
17
22
66
13
59
75
55
39
31
17
25
84
44
22
000000
000001
...
111111



Block-minima RMQ
13
39
17
22
66
13
59
75
55
39
31
17
25
84
44
22
000000
000001
...
101110
111111
Block-level RMQ


Block-minima RMQ
13
39
17
22
66
13
59
75
55
39
31
17
25
84
44
22
000000
000001
101010
...
101110
111111
Block-level RMQ
Block-level RMQ


Block-minima RMQ
13
39
17
22
66
13
59
75
55
39
31
17
25
84
44
22
000000
000001
101010
…
101110
111111
Block-level RMQ
Block-level RMQ


Block-minima RMQ
13
39
17
22
66
13
59
75
55
39
31
17
25
84
44
22
000000
000001
101010
...
101110
111111
Block-level RMQ
Block-level RMQ

Did we get <O(n), O(1)> complexity?

- Preprocessing time $p(n) = O(n + p_1(n/b) + (n/b)\, p_2(b))$
- Query time $q(n) = O(q_1(n/b) + q_2(b))$

This term needs to be O(n)

This term needs to be O(1)

Use sparse table for block minima

This term becomes O(1)

This term becomes O(n)

Did we get <O(n), O(1)> complexity?

- Preprocessing time $p(n) = O(n + n + (n/b)\, p_2(b))$
- Query time $q(n) = O(1 + q_2(b))$

This term needs to be O(n)

This term needs to be O(1)

This should be number of distinct types

Did we get ‹O(n), O(1)› complexity?

- Preprocessing time $p(n) = O(n + n + (4^b)\, p_2(b)\ )$
- Query time $q(n) = O(\ 1 + q_2(b)\ )$

This term needs to be O(n)

This term needs to be O(1)

Use full preprocessing for block level RMQ

Did we get $\langle O(n), O(1) \rangle$ complexity?

- Preprocessing time $p(n) = O(n + n + (4^b)\, b^2\,)$
- Query time $q(n) = O(\,1 + 1\,)$

Choose $b = \frac{1}{2} \log_4 n$

Did we get <O(n), O(1)> complexity?

- Preprocessing time $p(n) = O(n + n + n)$

- Query time $q(n) = O(\,1 + 1\,)$

Choose $b = \frac{1}{2}\log_4 n$

Fischer-Heun's <O(n), O(1)> Algorithm

- Choose $b = \frac{1}{2} \log_4 n$
- Split the input into blocks of size b
- Find the minimum of each block and store in an array
- Build a sparse table RMQ structure on the array of minima
- For each block:
  - Find the Cartesian number and check whether block-level RMQ structure is already built for its type
  - If not, build block-level RMQ structure and map it to its Cartesian number
- Make queries using hybrid approach

Practical Considerations

- Fischer-Heun is optimal and fast in practice
- In their 2005 paper, they reported that a simpler hybrid $\langle O(n), O(\log n) \rangle$ solution performs better than their solution
- Constants can be dominant in practice
- n should be very large; they ran experiments with 10M numbers
- Still an active field of study

Reference

- Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., , Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms 57(2) (2005) 75–94
- Johannes Fischer and Volker Heun:Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE
- https://web.stanford.edu/class/cs166/