External Memory Segment Trees

Rishav Mondal Siddharth Charan Mayank Chandak

April 2022

Outline

The problem we are trying to solve

Given a dynamically changing set of segments (specified by their endpoints, whose endpoints belong to a fixed set), and a query point, we need to find and report all segments that contain the given query point.
In other words, we can have 2 type of operations :

1. **Insert a segment**
2. **Stabbing query** : A stabbing query refers to a query in which we are given a query point and we need to find all segments that contain the query point.

Our task is to efficiently do these operations.

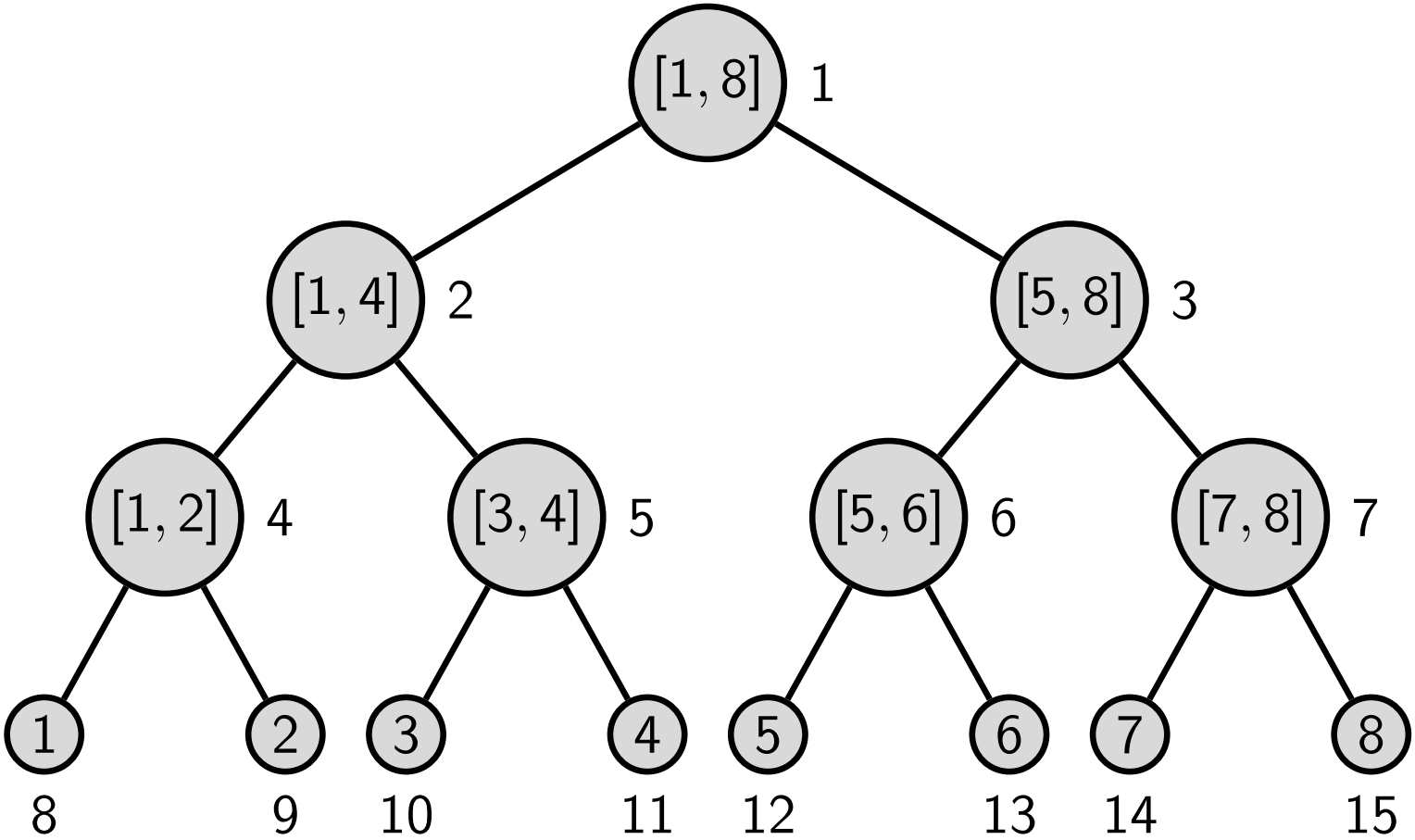
Internal memory version of segment tree

- Consists of a static binary tree (base tree) over the sorted set of endpoints.
- Every node in the tree is associated with some interval in the entire range of input points.
- A child node represents one of the sorted endpoints.
- An internal node's interval is union of it's left child and right child's interval.
- Every node also has a list associated with it.

Example

- Consider that we are given an input set of intervals $\{[1, 2], [3, 5], [4, 6], [7, 8]\}$.
- We build the segment tree and associate the interval range $[1, 8]$ with the root.

Example



How to insert a Segment

- A segment is stored in all nodes where the segment contains (spans) the interval associated with that node but does not span the interval associated with it's parent node.
- We can push the intervals that satisfy the above property to the relevant node's associated list.

Example

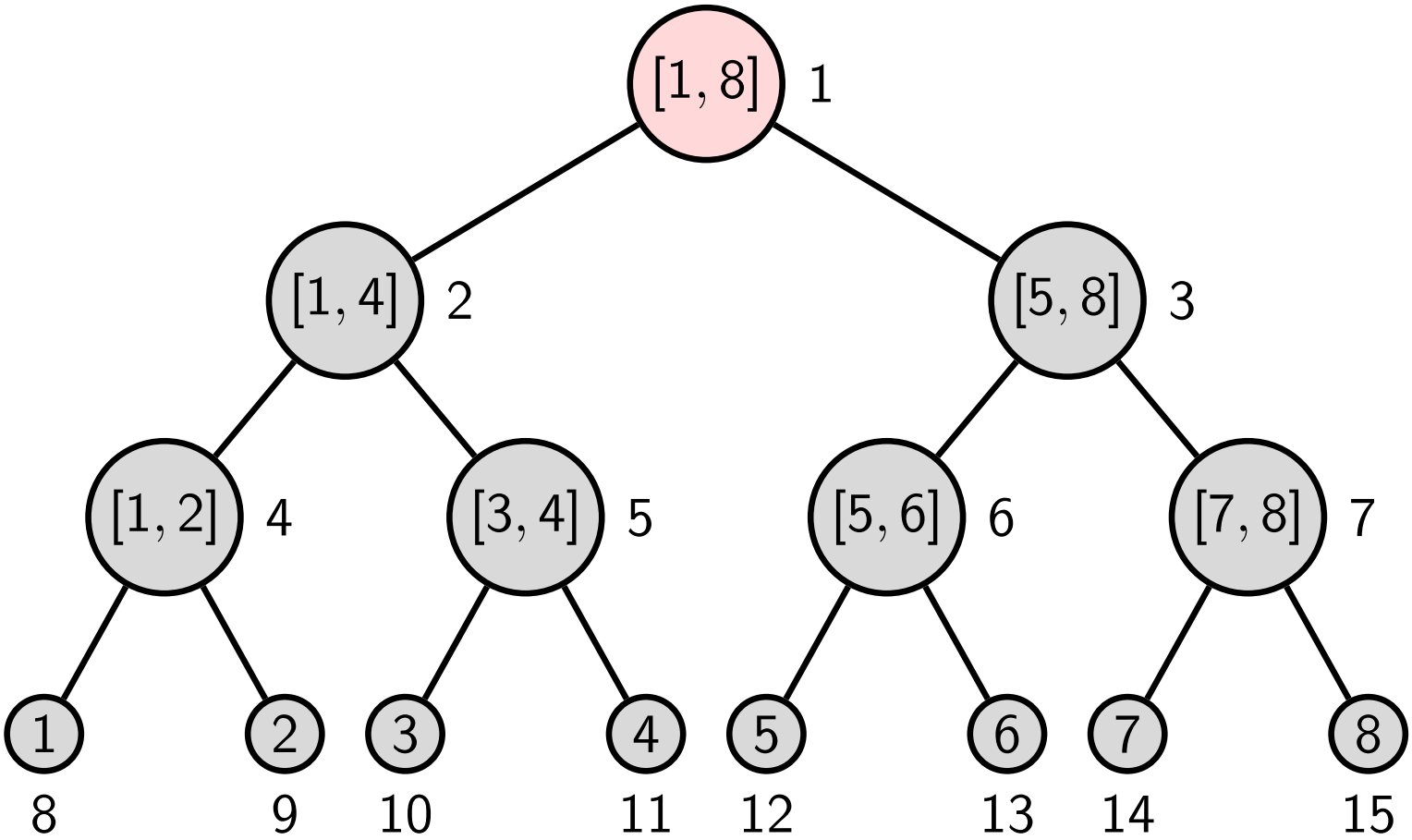
Let us consider specifically the interval $[3, 5]$ as an example. Which lists store the interval $[3, 5]$?
We start from the root [1,8]. Now as [3,5] does not span [1,8], so it is not stored in the list associated with the root. So we recurse down to it's two children. Now [3,5] does not completely [1,4], and similarly [5,8]. So we go further down. We notice that [1,2] lies outside [3,5], so we stop the recursion for that branch there. On the other hand, [3,5] completely spans [3,4] and hence we store [3,5] in list associated with node 5. For [7,8] we stop recursion because it lies completely outside [3,5], and for [5,6] we recurse further down. Finally we see that [3,5] completely spans the point 5, hence we push [3,5] in list associated with node 12. **So, [3,5] interval is stored in lists associated with node 5 and node 12.**
We can do similarly for other segments in the input list.

To answer a stabbing query(x)

- starting from the root, search down the tree for position of x among the leaves
- report all segments stored in the lists associated with the nodes encountered during the search

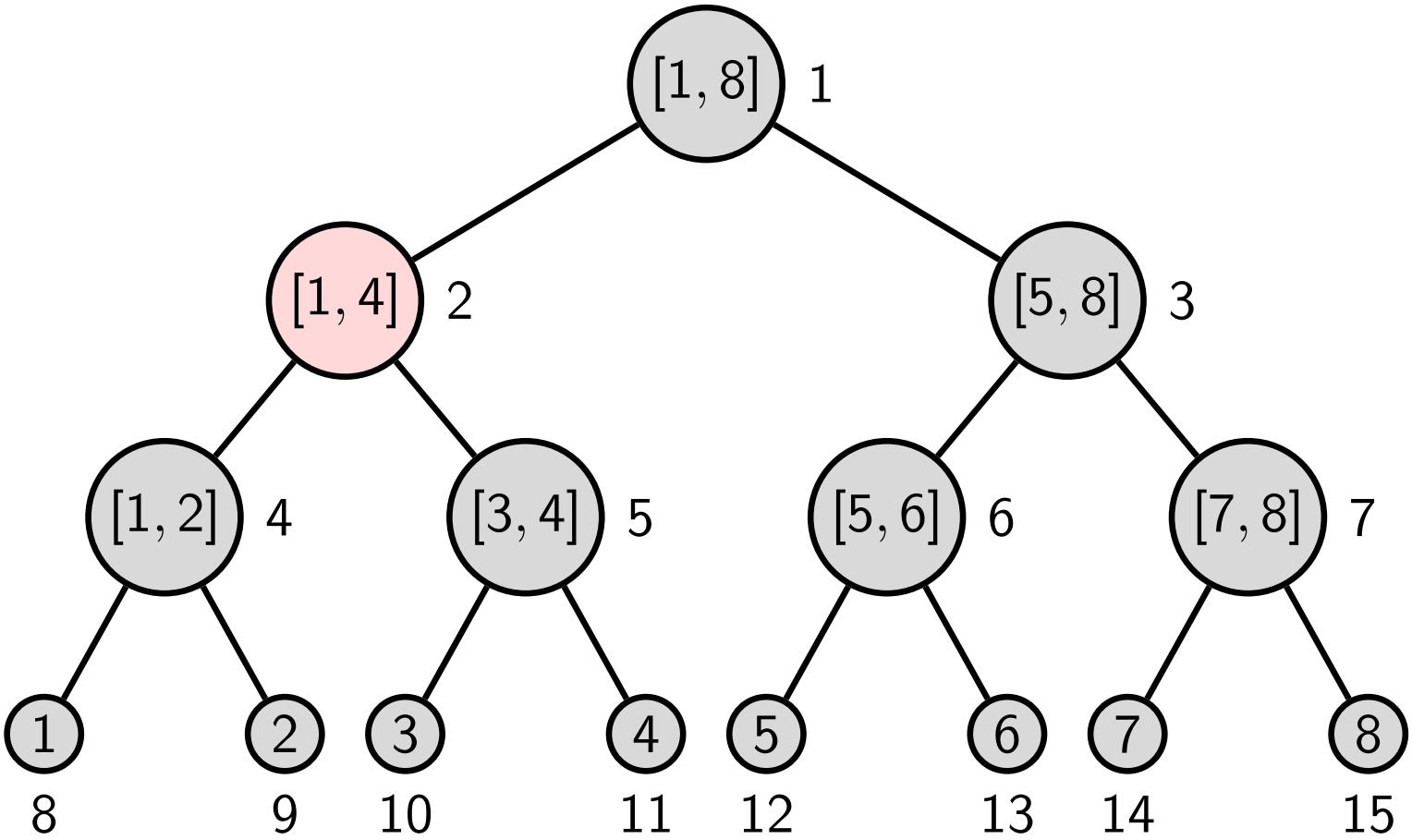
Answering stabbing query(3)

Print list associated with node 1 ([1,8]).



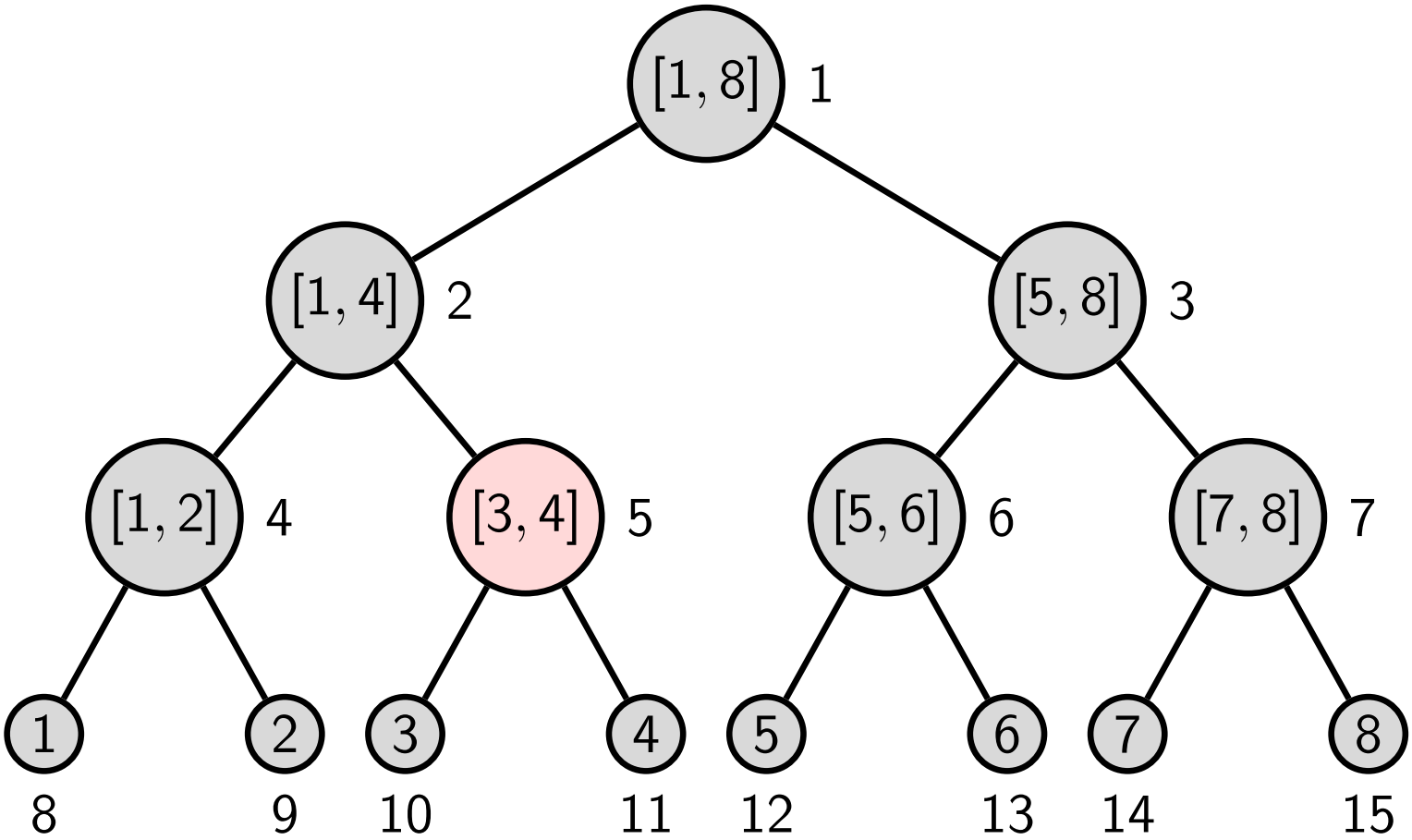
Answering stabbing query(3)

Print list associated with node 2 ([1,4]).



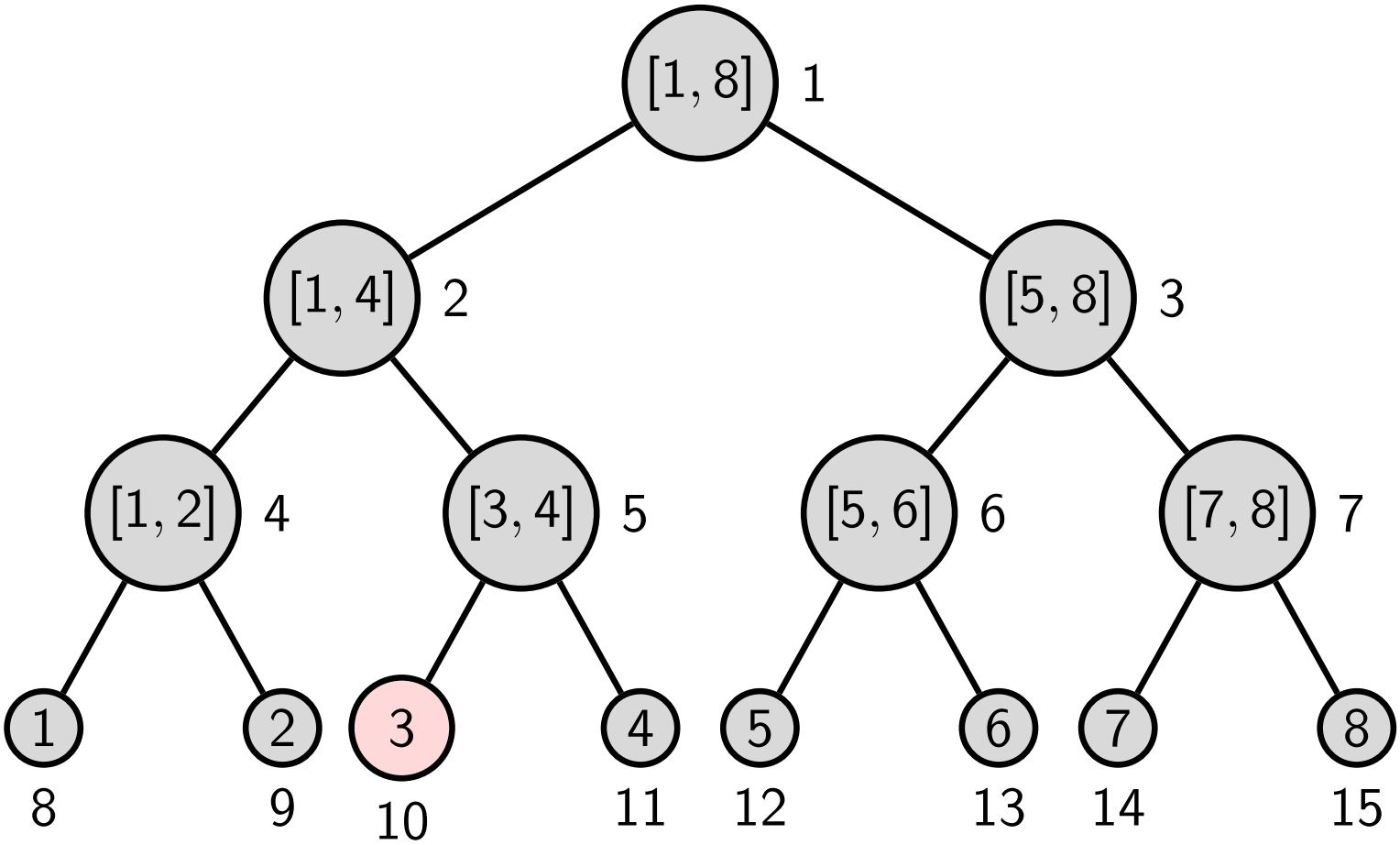
Answering stabbing query(3)

Print list associated with node 5 ([3,4]).



Answering stabbing query(3)

Print list associated with node 10 (3).



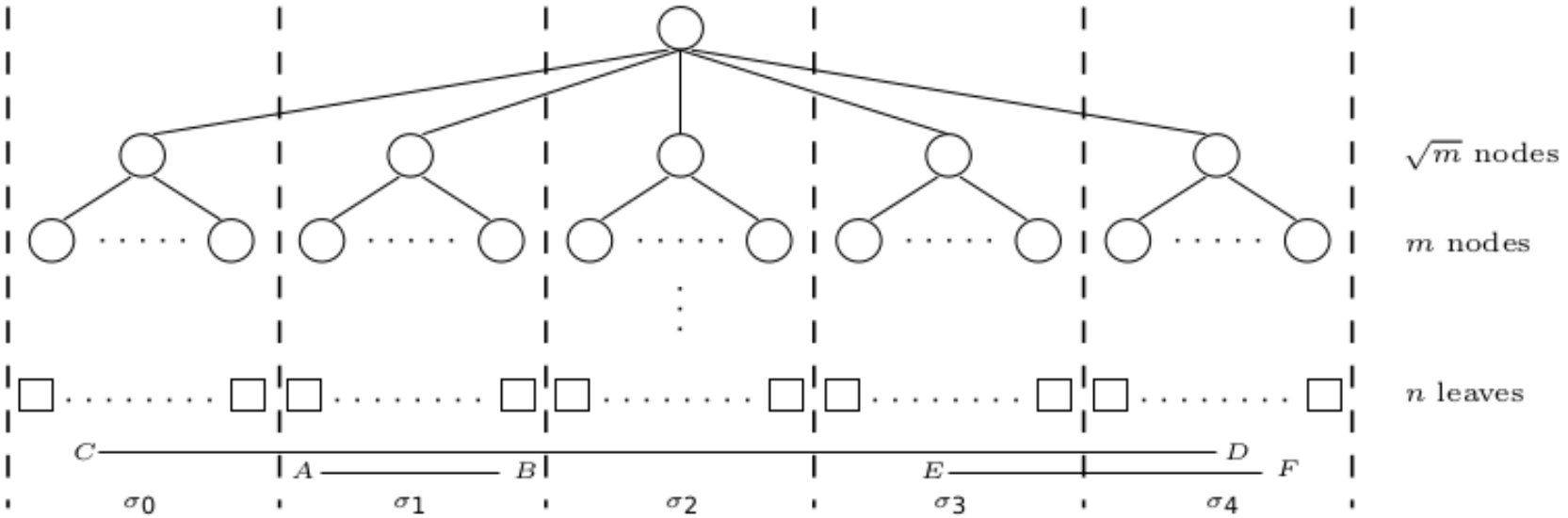
Problem with internal memory segment tree

In the worst case, this approach would require an input segment to be stored in $O(log_2 n)$ lists. Now to store a segment in a list, we need an I/O to bring the list into the main memory. Thus this approach would require too many I/Os, and therefore we need to do better, and need to redesign the entire structure of the segment tree in order to get better performance in external memory.

Design of external memory segment tree

- The base structure remains the same as before - a perfectly balanced tree over the set of endpoints.
- The branching factor of the tree is chosen to be $\sqrt{m}$; where $m = \frac{M}{B}$.
- WLOG we assume that $n = (\sqrt{m})^k$ where $n = \frac{N}{B}$ and k is some integer and N is the total number of operations. We also assume that the endpoints of the segments are all distinct.
- Each internal node has a buffer of size $\frac{m}{2}$ and and $(\frac{m}{2} - \frac{\sqrt{m}}{2})$ lists of segments associated with it (in contrast to only 1 list in the internal case).
- Each leaf node only has a list (block) of segments associated with it.

Design of external memory segment tree



Design of external memory segment tree

- The first level of the tree (the root) partitions the tree into $\sqrt{m}$ slabs. It is indicated by the dotted lines in the figure. We denote the slabs by $\sigma_0, \sigma_1, \ldots$
- The **multislabs** for the root are then defined as continous ranges of slabs. For example $[\sigma_1, \sigma_3]$. Notice that there can be a total of $\binom{\sqrt{m}}{2}$ multislabs, which is equal to $(\frac{m}{2} - \frac{\sqrt{m}}{2})$, and hence our choice for the number of multislabs associated with every node.
- We have a list associated with every multislab.

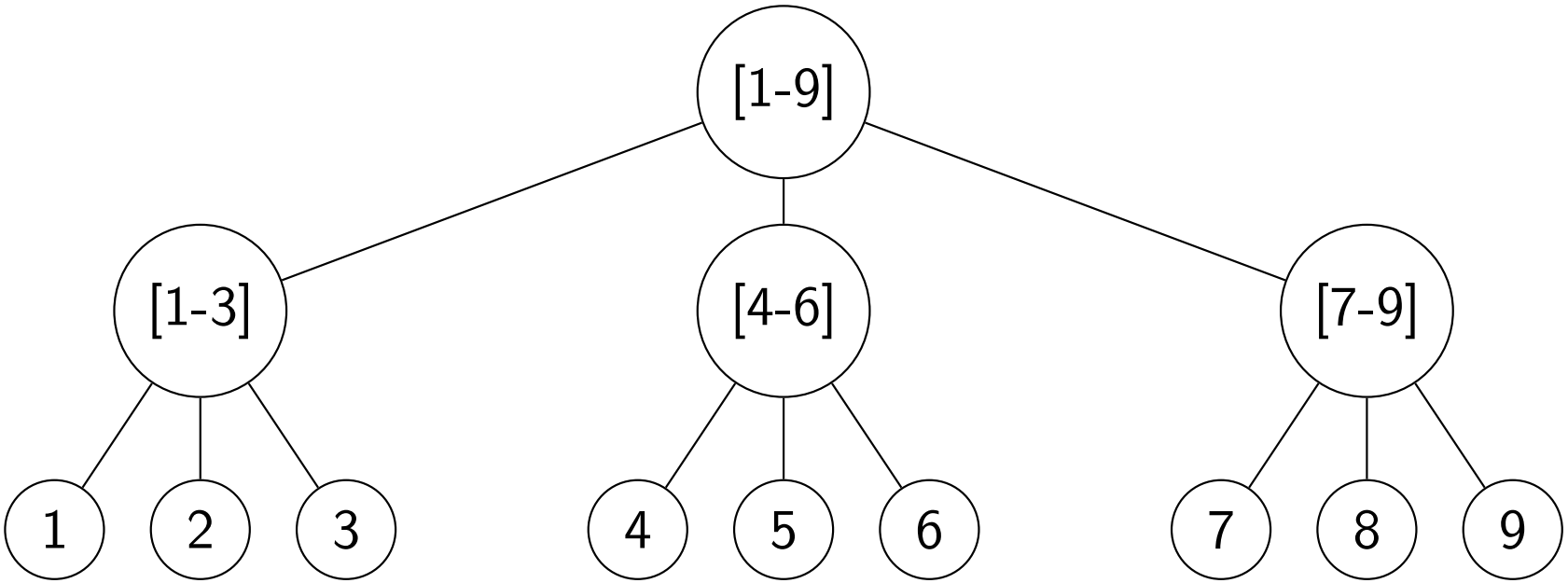
Terminology

- **Long segment**: It is a segment that completely spans greater than 1 slab. A copy of each long segment is stored in a list associated with the largest multislab it spans.
- **Short segment**: Short segments are segments that span less than or equal one slab(in other words, segments that are not long). They are not stored in any multislab list and are to be recursed down to lower levels of the tree.
- For any long segment, there can be atmost two short segments associated with it (the ones at the fringes).

Example

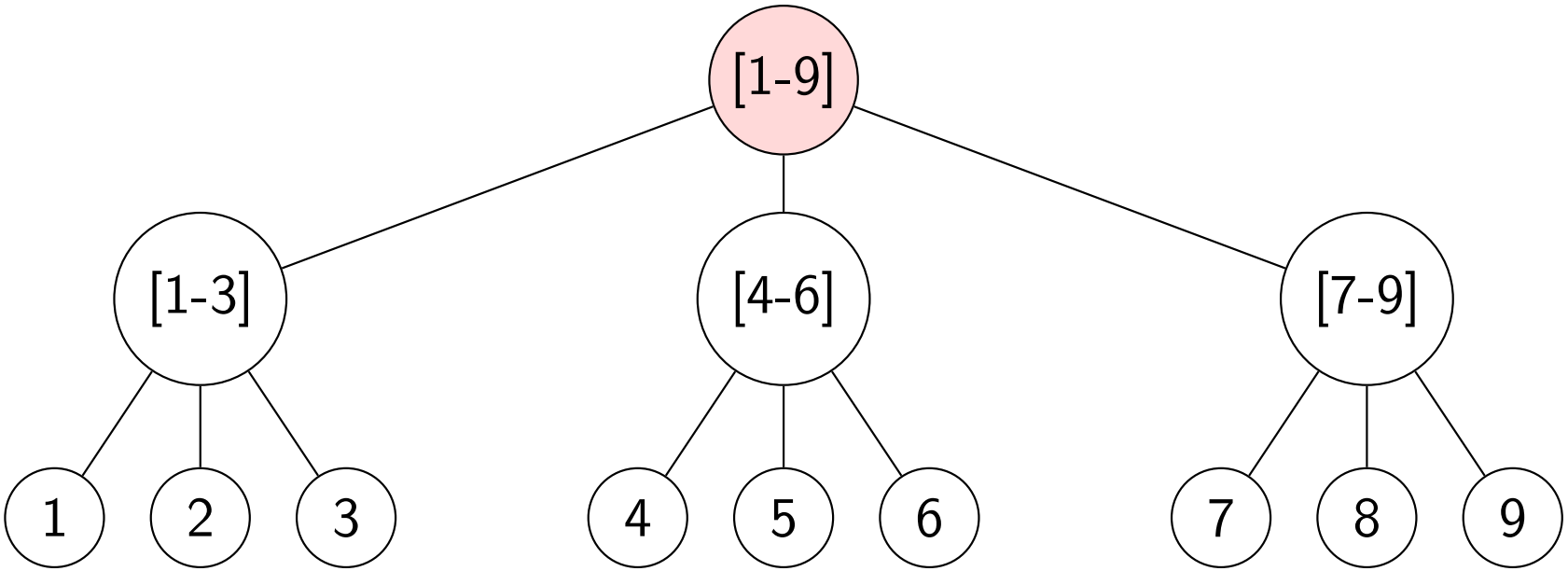
- Consider that we are given an input set of intervals $\{[1, 2], [3, 5], [4, 6], [7, 9], [8, 9]\}$.
- We build the segment tree with branching factor $k = \sqrt{m}$ and associate the interval range [1, 9] with the root. Let us assume that $\sqrt{m} = 3$ for sake of simplicity.
- Every internal node has a buffer and $(\frac{m}{2} - \frac{\sqrt{m}}{2}) = \frac{(9-3)}{2} = 3$ lists associated with it, each for every multislab.
- There are 3 multislabs associated with the root - $[\sigma_0, \sigma_1], [\sigma_1, \sigma_2], [\sigma_0, \sigma_2]$.

Example



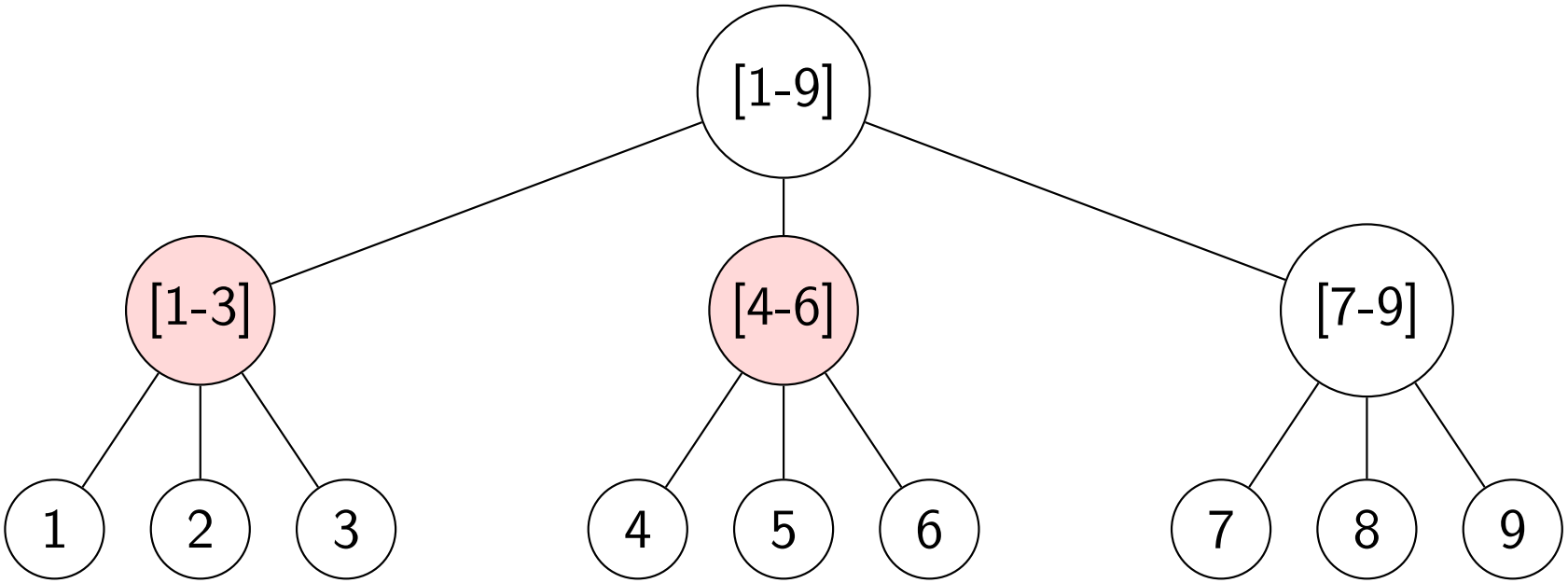
Storing example segment [4,9]

The segment is stored in list of multislab $[\sigma_1, \sigma_2]$ associated with the root since [4,9] completely spans it.



Storing example segment [2,5]

The segment is stored in list of multislab $[\sigma_1, \sigma_2]$ associated with the node [1-3] and in list of multislab $[\sigma_0, \sigma_1]$ associated with the node [4-6] .



Why to use buffers?

It can be shown that a segment is stored in at most four lists on each level of the base tree (Why?). Now given an external segment tree (with empty buffers) a stabbing query can in analogy with the internal case be answered by searching down the tree for the query value, and at every node encountered report all the long segments associated with each of the multislabs that span the query value. However, answering queries on an individual basis is of course not I/O-efficient. Hence we use a buffer of size $O(m)$ associated with each node for reasons discussed below.

Operations on external segment tree

When we use a segment tree, we talk about **insert, delete and query** operations. However a separate delete operation is not required.
Here, instead of separate delete operation, we will talk in terms of "time" when a segment should be deleted from the tree. We will require this delete-time for each segment, when they are inserted.

Preprocessing time:

1. Sort the endpoints in increasing order. This takes $O(nlog_m n)$ time.
2. Build the base tree structure.

Building base tree structure

It is easy to realize how the base tree structure can be build in $O(n)$ I/O operations given the endpoints in sorted order. First we construct the leaves by scanning through the sorted list, and then we repeatedly construct one more level of the tree by scanning through the previous level of nodes (leaves). In constructing one level we use a number of I/Os proportional to the number of nodes on the previous level, which means that we in total use $O(n)$ I/Os as this is the total number of nodes and leaves in the tree.

Algorithm for insert/query

1. We make a new element with the segment or query point in question, a time-stamp, and—if the element is an insert element—a delete time. When we have collected a block of such elements, we insert them in the buffer of the root. If the buffer of the root now contains more than $m/2$ elements we perform a buffer-emptying process on it.

Algorithm for insert/query

2. We will move the buffer to memory. Now, we need to perform 2 types of operations for each multislab:
   - **Store segments**: Collect all segments that should be stored in the node (long segments). Then for every multi-slab list in turn insert the relevant long segment in the list. At the same time replace every long segment with the small (synthetic) segments which should be stored recursively.
   - **Report stabbings**: For every multi-slab list in turn decide if the segments in the list are stabbed by any query point. If so then scan through the list and report the relevant elements while removing segments which have expired (segments for which all the relevant queries are inserted after their delete time).

Algorithm for insert/query

3. Distribute the segments and the queries to the buffers of the nodes on the next level.
4. If the buffer of any of the children now contains more than $m/2$ blocks, the buffer emptying process is recursively applied on these nodes.

Time complexity analysis

1. The first step can be easily completed by storing B queries at a time in buffer. Hence it is easy to note that storing operations in buffer takes amortized $O(\frac{1}{B})$ time for 1 query.
2. We can move buffer elements in $O(m)$ time in memory. Same, way storing 1 non-full block from each multi-slab list also costs $O(m)$. Now, we will see the time-complexity to perform 2 different type of operations.

Time complexity analysis

- **Store segments**: As we are storing a segment in 1 multi-slab only, we can directly store these in fetched multi-slab lists non-full block. If, this non-full becomes full we can fetch next block, and it can easily be seen that we need at most $O(2 \cdot m)$ operations, to perform $O(m)$ operations.
- **Report Stabbings** : The number of I/Os charged to the buffer-emptying process for queries is also $O(m)$, as this is the number of I/Os used to load non-full multi-slab list blocks. The rest of the I/Os used to scan a multi-slab list can either be charged to a stabbing reporting or to the deletion of an element. We can do so by assuming that every segment holds $O(\frac{1}{B})$ credits to pay for its own deletion. This credit can then be accounted for (deposited) when we insert a segment in a multi-slab list.

Time complexity analysis

3. Again trivially time to distribute operations to below level is linear.
4. Now, for a operation this recursive step can happen for $O(log_m n)$ times.

Hence, total time complexity $= O(nlog_m n + r)$.

Final result

Theorem

Given a sequence of insertions of segments (with delete time) intermixed with stabbing queries, such that the total number of operations is N, we can build an external-memory segment tree on the segments and perform all the operation on it in $O(n\log_m n + r)$ I/O operations (Here r stands for the number of stabbings reported so far).

References

- Efficient External-Memory Data Structures and Applications by Lars Arge (1996)
- The Buffer Tree: A New Technique for Optimal I/O Algorithms by Lars Arge (1996)