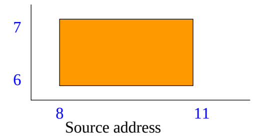
Segment Trees

- Basic data structure in computational geometry.
- Computational geometry.
  - Computations with geometric objects.
  - Points in 1-, 2-, 3-, d-space.
    - Closest pair of points.
    - Nearest neighbor of given point.
  - Lines in 1-, 2-, 3-, d-space.
    - Machine busy intervals.
    - IP router-table filters (10*, [20, 60]).

Segment Trees

- Rectangles or more general polygons in 2-space.
  - VLSI mask verification.
  - Sentry location.
  - 2-D firewall filter.
    - (source address, destination address)
    - (10*, 011*)
    - When addresses are 4 bits long this filter matches addresses in the rectangle ([8,11], [6,7])



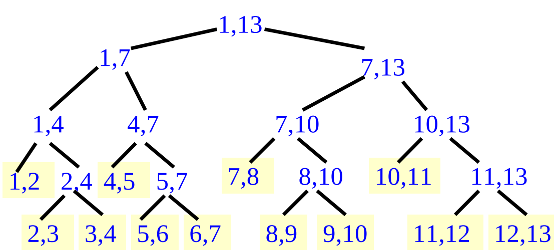
Segment Tree Application

- Store intervals of the form [i,j], i < j, i and j are integers.
  - [i,j] may, for example represent the fact that a machine is busy from time i to time j.
- Answer queries of the form: which intervals intersect/overlap with a given unit interval [a,a+1].
  - List all machines that are busy from 2 to 3.

Segment Tree – Definition

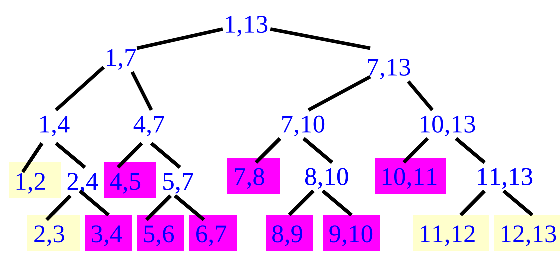
- Binary tree.
- Each node, v, represents a closed interval.
  - s(v) = start of v's range.
  - e(v) = end of v's range.
  - s(v) < e(v).
  - s(v) and e(v) are integers.
  - Root range = [1,n].
- e(v) = s(v) + 1 => v is a leaf node (unit interval).
- e(v) > s(v) + 1 =>
  - Left child range is [s(v), (s(v) + e(v))/2].
  - Right child range is [(s(v) + e(v))/2, e(v)].

Example – Root range = [1,13]



Cream colored boxes are leaves/unit intervals.

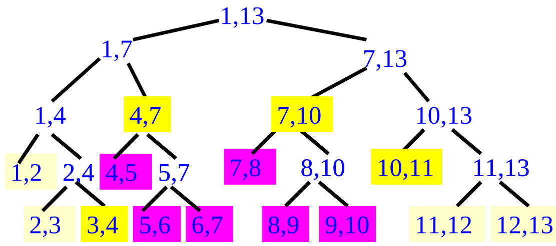
Store Interval [3,11]



Unit intervals of [3,11] highlighted.

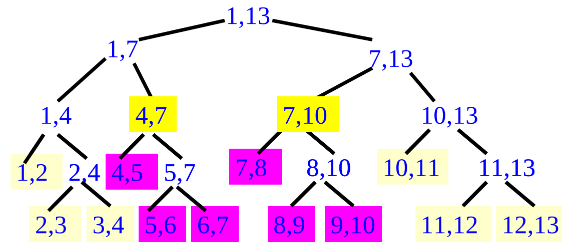
Each interval [i,j], i < j, is stored in one or more nodes of the segment tree.

Store Interval [3,11]



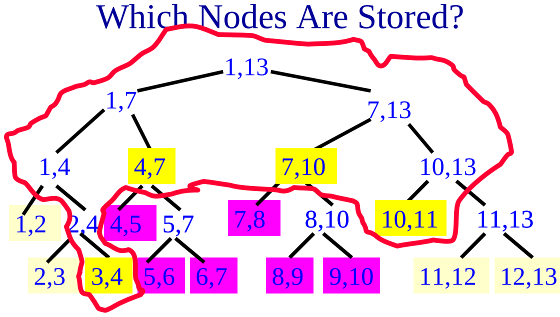
[3,11] is stored in node p iff all leaves in subtree rooted at p are highlighted and no ancestor of p satisfies this property.

Store Interval [4,10]



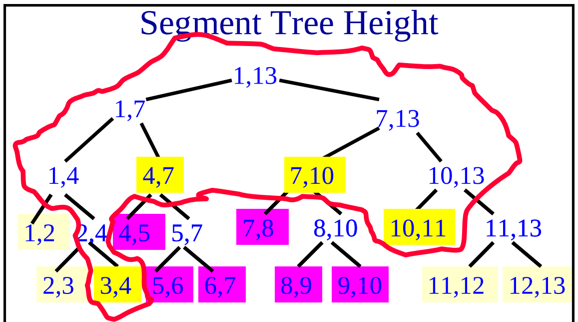
Each node of a segment tree contains 0 or more intervals.

Which Nodes Are Stored?



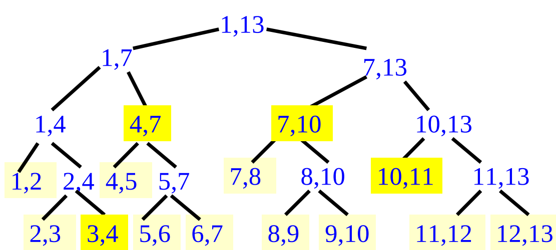
Need to store only those nodes that contain intervals plus the ancestors of these nodes.

Segment Tree Height



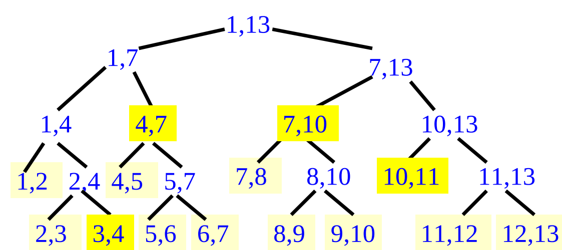
Range = [1,n] => Height <= ceil($\log_2$ (n-1)) + 1.

Properties



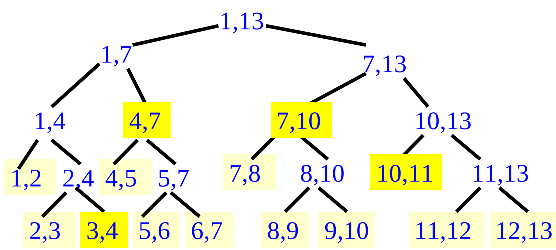
[i,j] in node v => [i,j] not in any ancestor of v .

Properties



[i,j] in node v => [i,j] not in sibling of v .

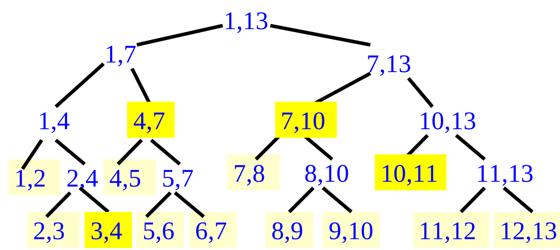
Properties



[i,j] may be in at most 2 nodes at any level.

Each interval is in O(log n) nodes.

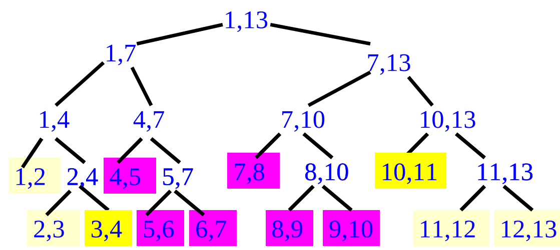
Top-Down Insert — [3,11]



Top-Down Insert

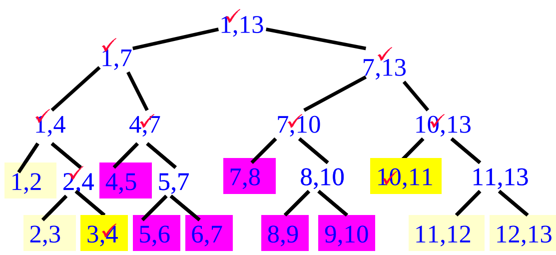
```
insert(s, e, v)
{// insert [s,e] into subtree rooted at v
   if (s <= s(v) && e(v) <= e)
      add [s,e] to v; // interval spans node range
   else {
          if (s < (s(v) + e(v))/2)
             insert(s,e,v.leftChild);
          if (e > (s(v) + e(v))/2)
             insert(s,e,v.rightChild);
        }
}
```

Complexity Of Insert



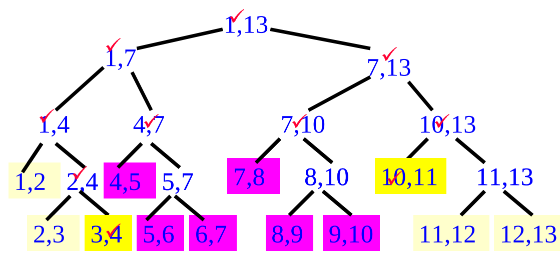
Let L and R, respectively, be the leaves for [s,s+1] and [e – 1,e].

Complexity Of Insert



In the worst-case, L, R, all ancestors of L and R, and possibly the other child of each of these ancestors are visited.

Complexity Of Insert



Complexity is O(log n).

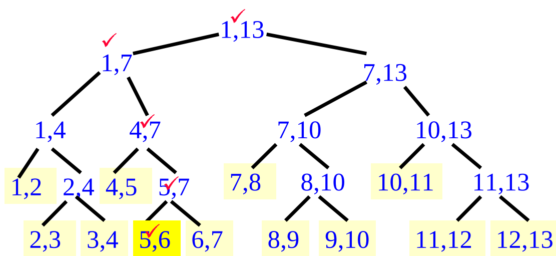
Top-Down Delete

```
delete(s, e, v)
{// delete [s,e] from subtree rooted at v
    if (s <= s(v) && e(v) <= e)
        delete [s,e] from v; // interval spans node range
    else {
            if (s < (s(v) + e(v))/2)
                delete(s,e,v.leftChild);
            if (e > (s(v) + e(v))/2)
                delete(s,e,v.rightChild);
         }
}
```

Search – [a,a+1]

- Follow the unique path from the root to the leaf node for the interval [a,a+1].
- Report all segments stored in the nodes on this unique path.
- No segment is reported twice, because no segment is stored in both a node and the ancestor of this node.

Search – [5,6]



O(log n + s), where s is the # of segments in the answer.