Solving the Dynamic Predecessor Problem

- Calvin Higgins, Ethan Carlson and Robert Oganesian

Problem: Implement a dynamic ordered set S_k on a bounded universe U_k.

Problem: What does that mean???

Definition: Dynamic Ordered Set

- Dynamic
  - S.insert(x)
  - S.remove(x)
- Ordered
  - S.predecessor(x)
    - Find first element of S strictly less than x
  - S.successor(x)
    - Find first element of S strictly greater than x
- Set
  - S.contains(x)

Examples: Dynamic Ordered Set

- Binary search tree
- Red-black tree
- Sorted vector
- Sorted list

Examples: Dynamic Ordered Set

	insert(x)	remove(x)	predecessor(x)	successor(x)	contains(x)	space
BST	O(N)	O(N)	O(log(N))	O(log(N))	O(N)	O(N)
RB-Tree	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(N)
Sorted Vector	O(N)	O(N)	O(log(N))	O(log(N))	O(log(N))	O(N)
Sorted List	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)
???	O(log(M))	O(log(M))	O(log(log(M)))	O(log(log(M)))	O(1)	O(N*log(M))
???	O(log(log(M)))	O(log(log(M)))	O(log(log(M)))	O(log(log(M)))	O(1)	O(N)

Definition: Bounded Universe

- Bounded universe $U_k = \{x \mid x \in Z, 0 \leq x \leq k\}$ where $k \geq 0$
- Bounded universe U_k contains all unsigned integers in the range [0, k]
- We are storing unsigned integers in some range!

Examples: Bounded Universe

- $U_0 = \{0\}$
- $U_3 = \{0, 1, 2, 3\}$
- $U_7 = \{0, 1, 2, 3, 4, 5, 6, 7\}$
- $U_{1024} = \{0, 1, \ldots, 1023, 1024\}$

Notation: We denote a dynamic ordered set on U_k as S_k.

Problem: Implement a dynamic ordered set S_k on a bounded universe U_k.

Time to start developing our data structure!

Idea: Represent a set as a vector of booleans.

- **Consider:** Set S_7 = {0, 1, 7}
  - **Recall:** S_7 is on U_7 = {0, 1, 2, 3, 4, 5, 6, 7}

Vector V_7 representing set S_7							
0	1	2	3	4	5	6	7
true	true	false	false	false	false	false	true

Example: Represent a set as an vector of booleans.

- **Consider:** S_7.contains(3) is the same as V_7[3]

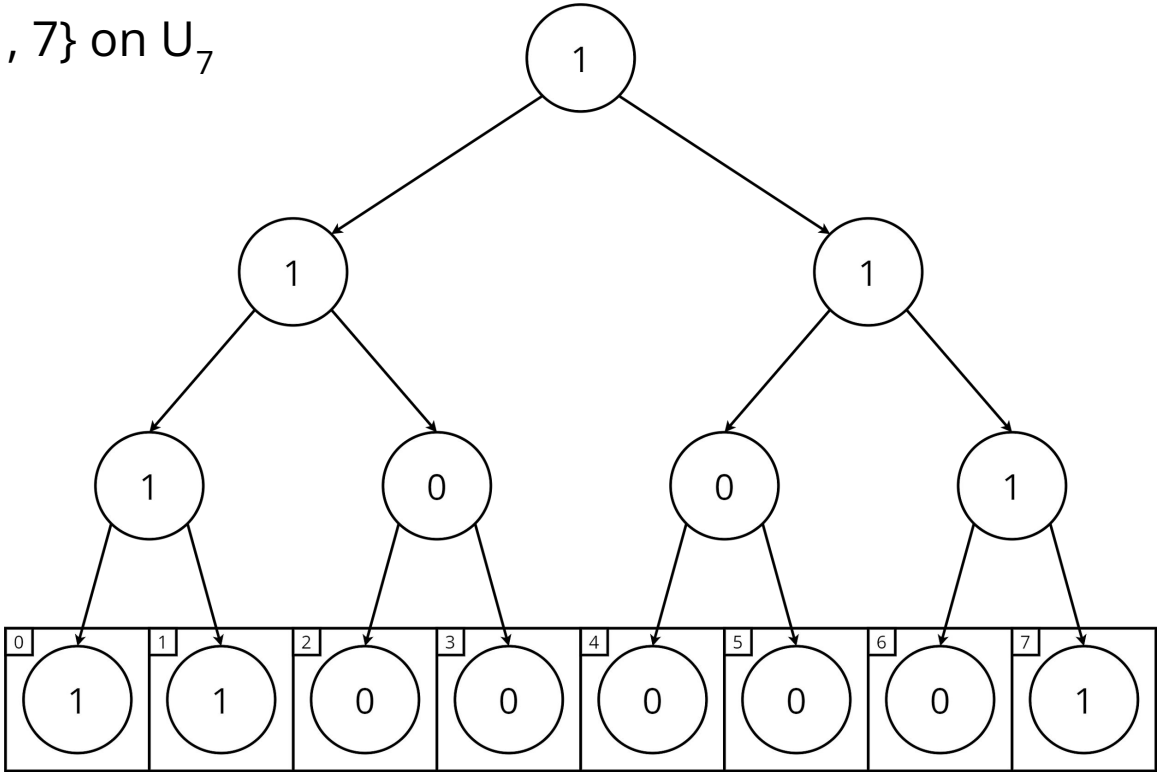
Vector V_7 representing set S_7							
0	1	2	3	4	5	6	7
true	true	false	false	false	false	false	true

Analysis: O(1) complexity for contains because we can simply index into a vector.

Problem: How to achieve sublinear complexity for predecessor and successor?

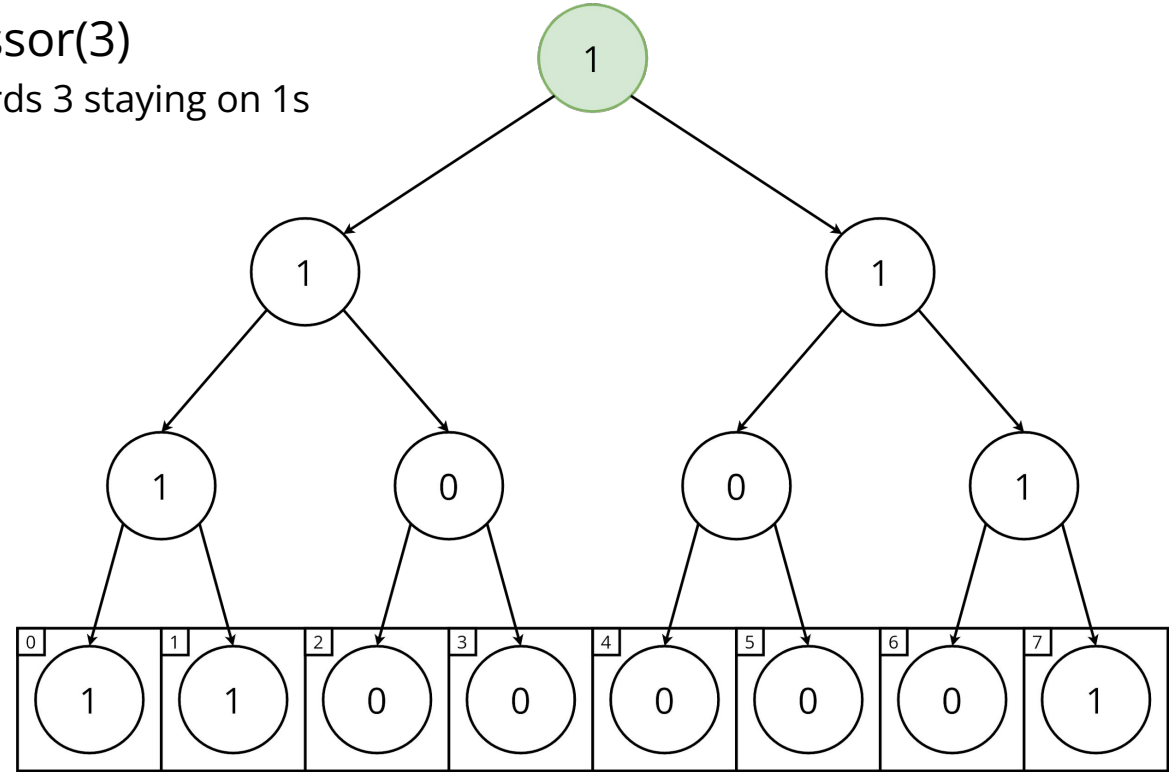
Idea: Represent array of bits as a binary tree.

- **Consider:** Set S_7 = {0, 1, 7} on U_7



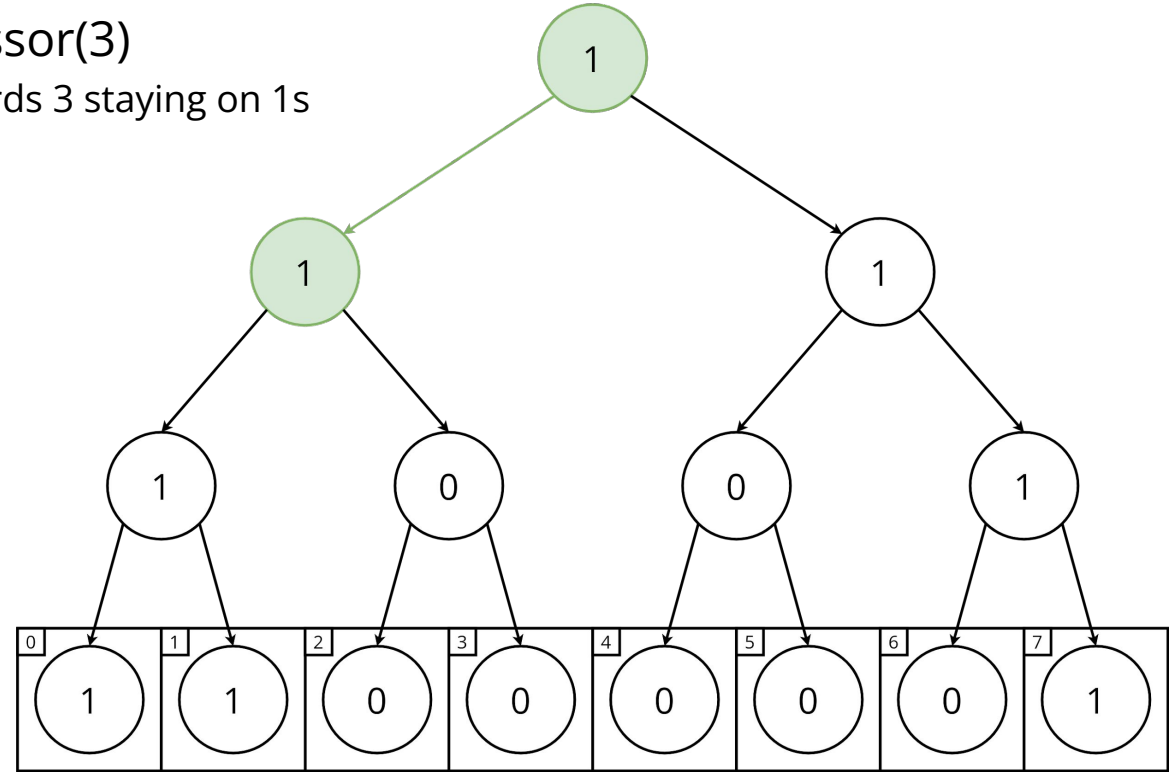
Example: Represent array of bits as a binary tree.

- **Consider:** S_7.predecessor(3)
  - **Idea:** Walk down towards 3 staying on 1s



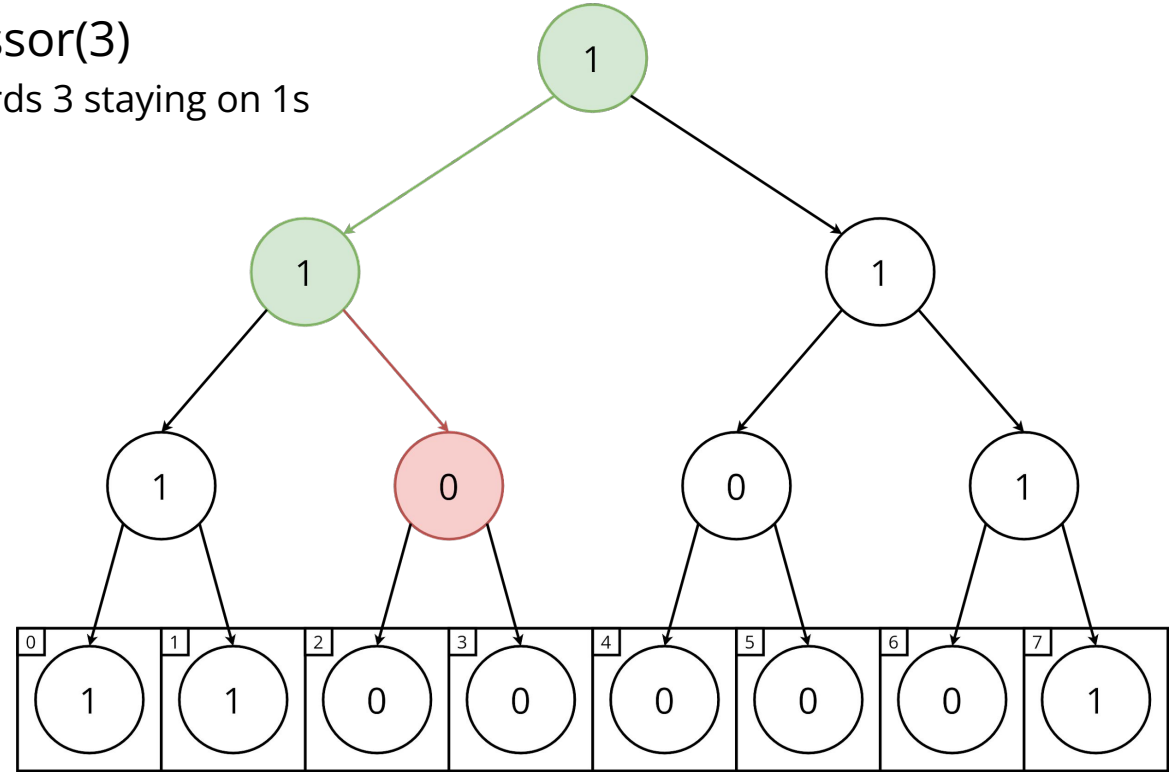
Example: Represent array of bits as a binary tree.

- **Consider:** S_7.predecessor(3)
  - **Idea:** Walk down towards 3 staying on 1s



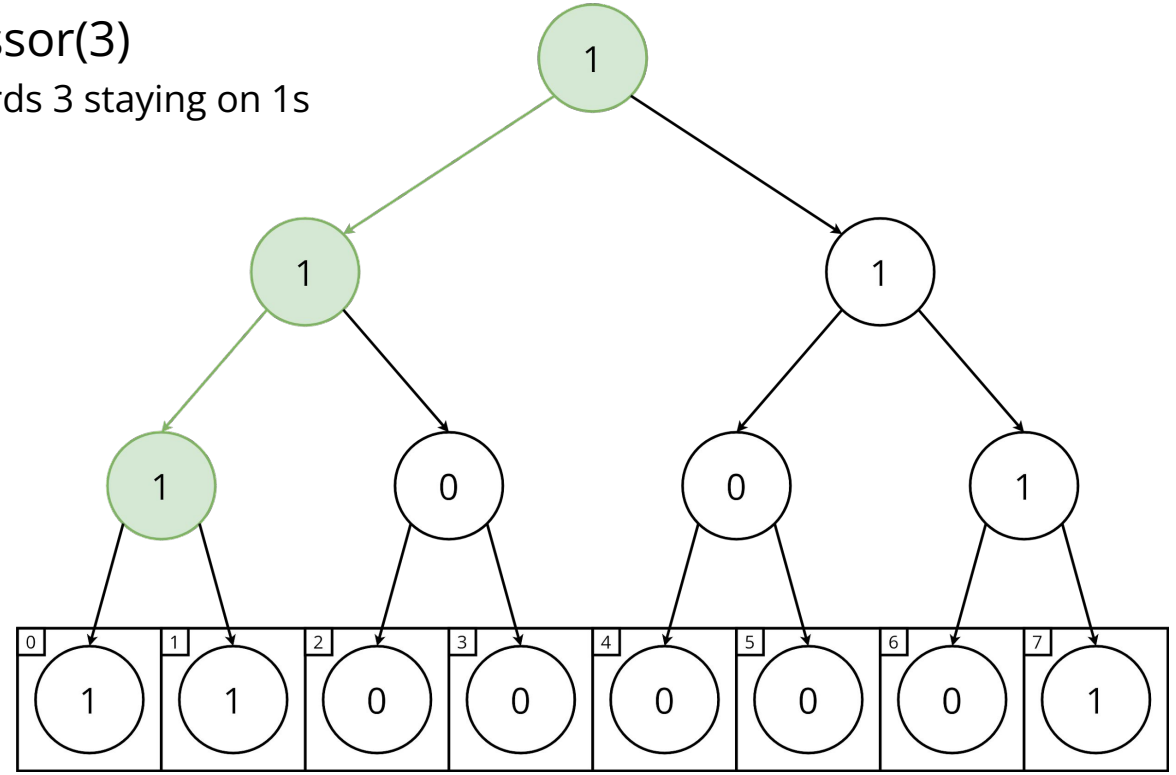
Example: Represent array of bits as a binary tree.

- **Consider:** S_7.predecessor(3)
  - **Idea:** Walk down towards 3 staying on 1s



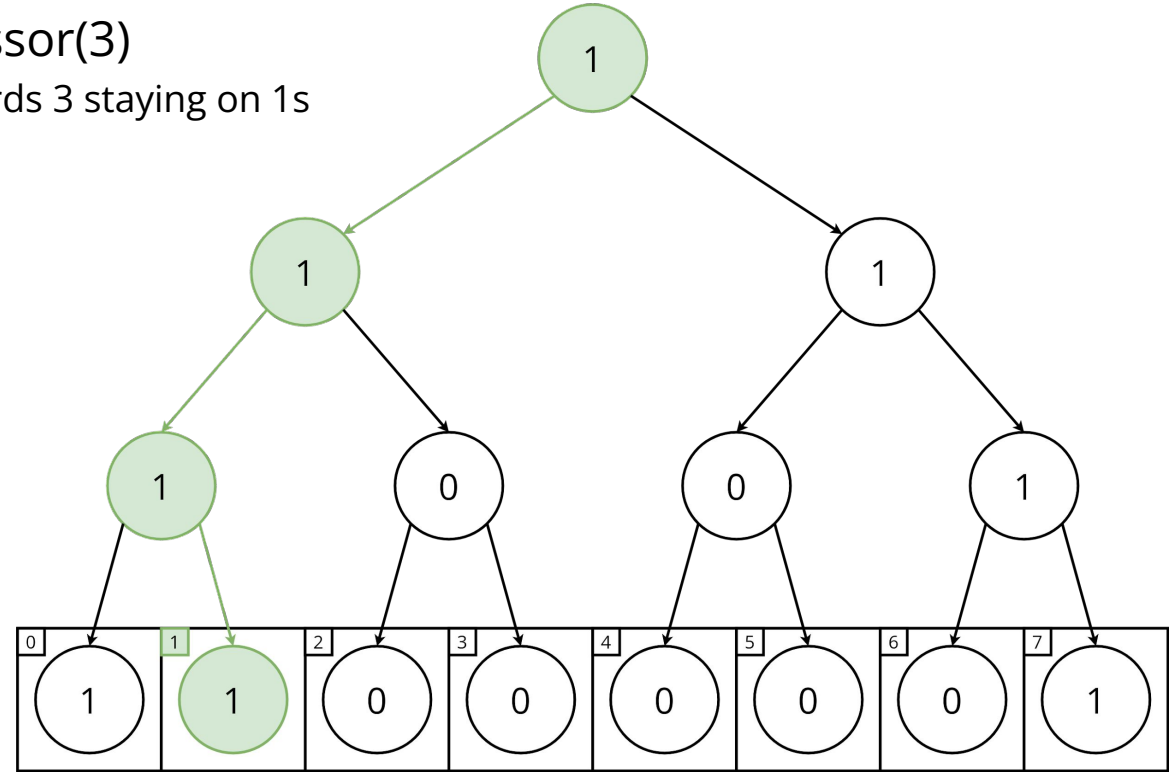
Example: Represent array of bits as a binary tree.

- **Consider:** S_7.predecessor(3)
  - **Idea:** Walk down towards 3 staying on 1s



Example: Represent array of bits as a binary tree.

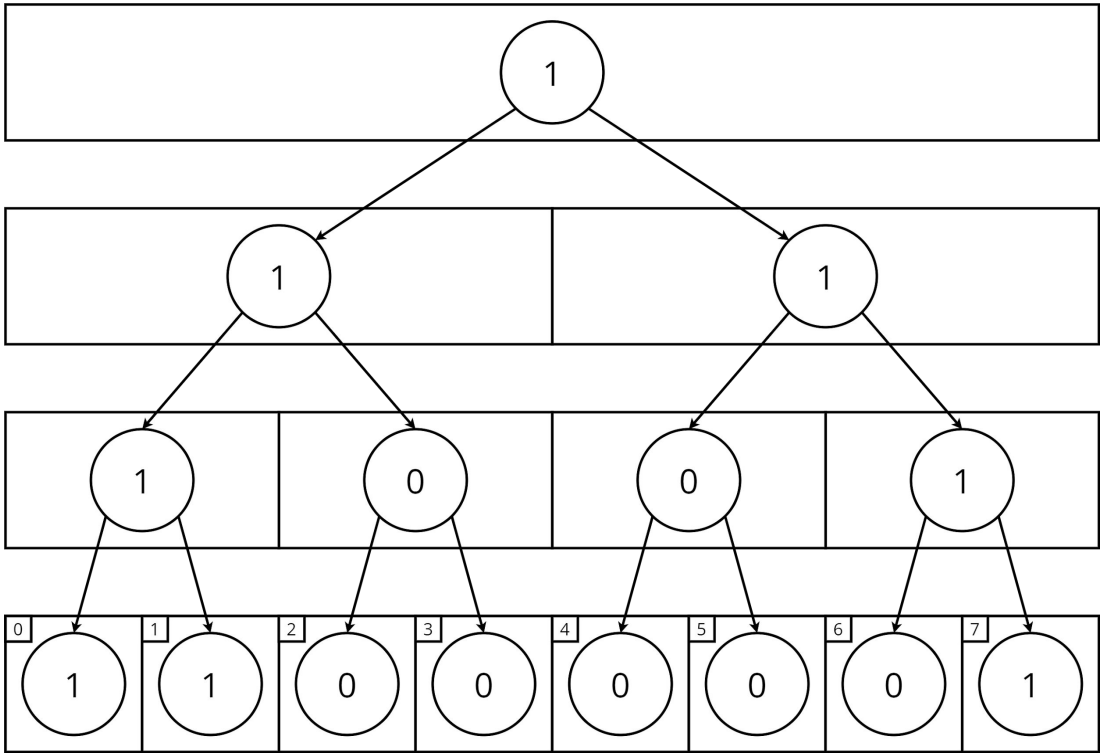
- **Consider:** S_7.predecessor(3)
  - **Idea:** Walk down towards 3 staying on 1s



Analysis: O(log(M)) complexity for predecessor ***or*** successor because we walk a height log(M) tree.

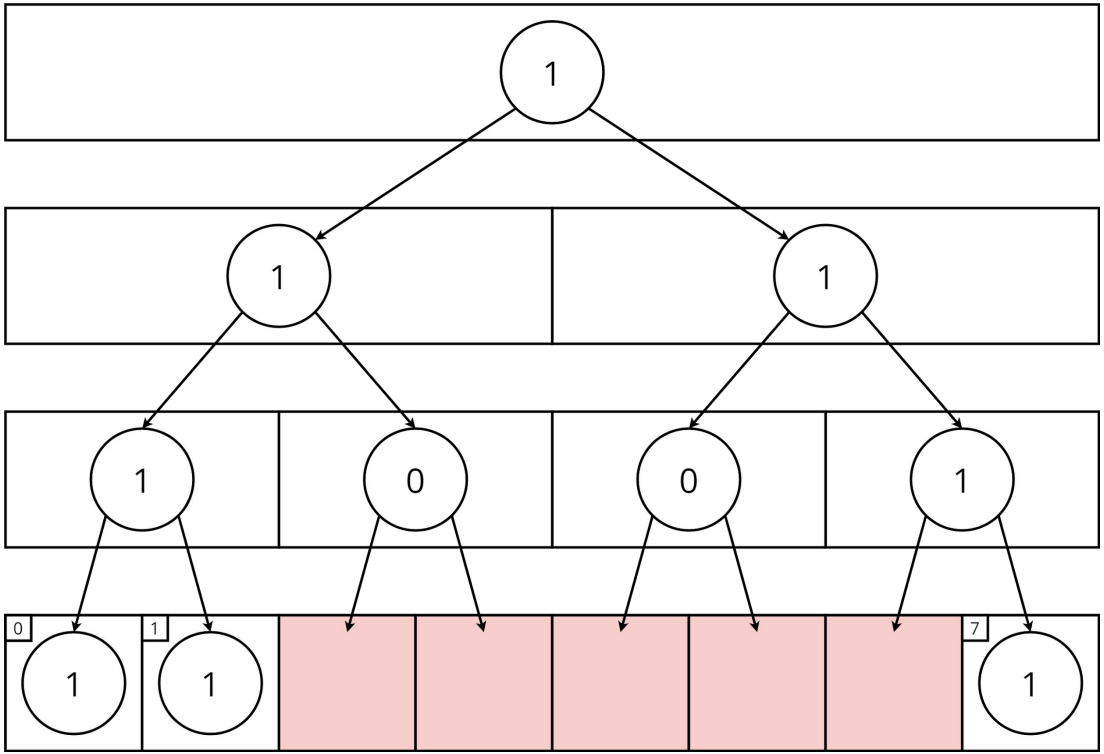
Problem: Reduce space complexity from O(M) to O(N*log(M)).

Idea: Store layers in hash tables.



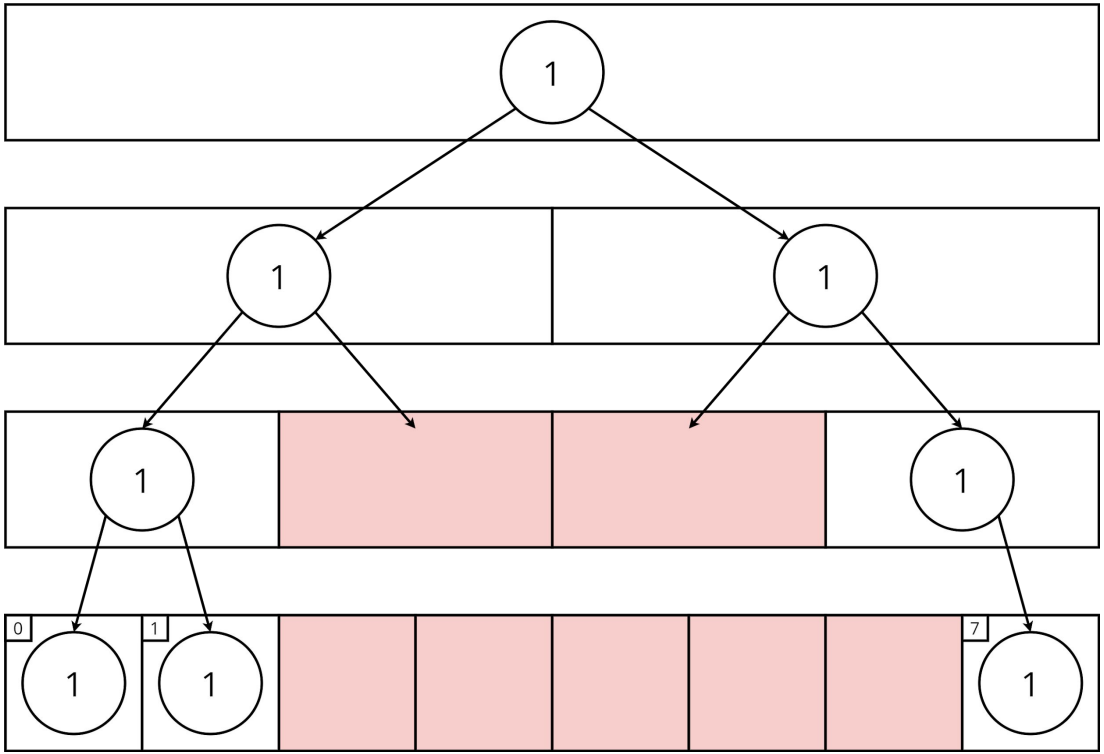
Consider: Set S_7 = {0, 1, 7} on U_7

Idea: Store layers in hash tables.



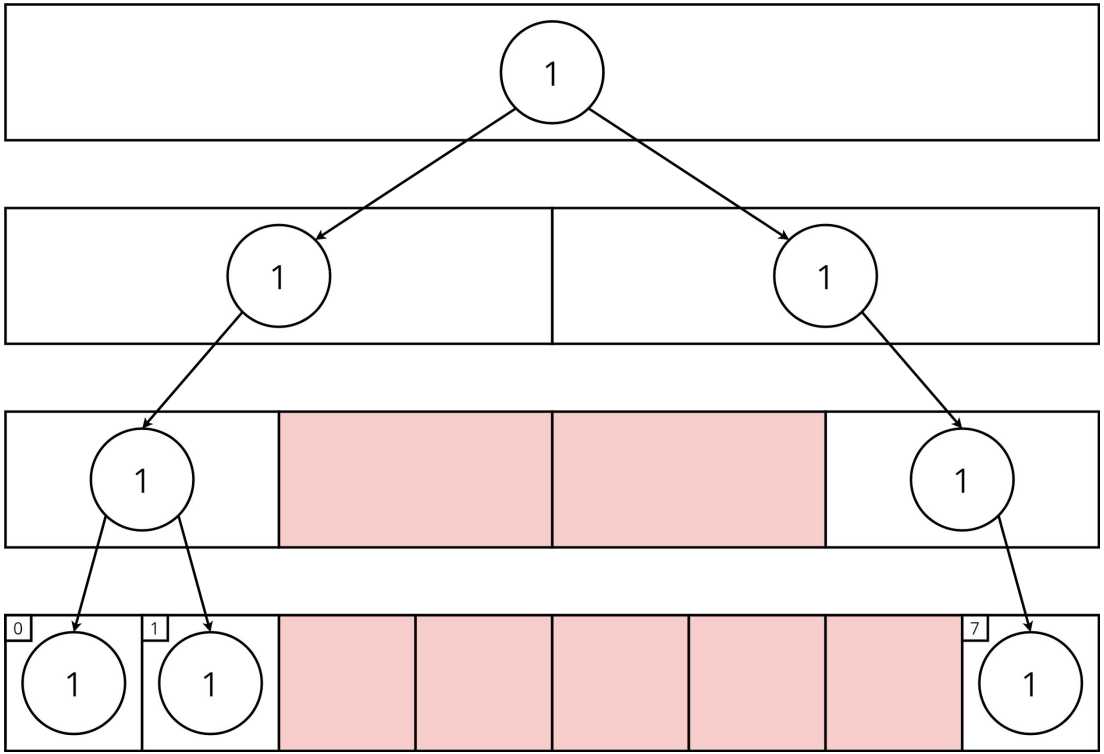
Consider: Set S_7 = {0, 1, 7} on U_7

Idea: Store layers in hash tables.



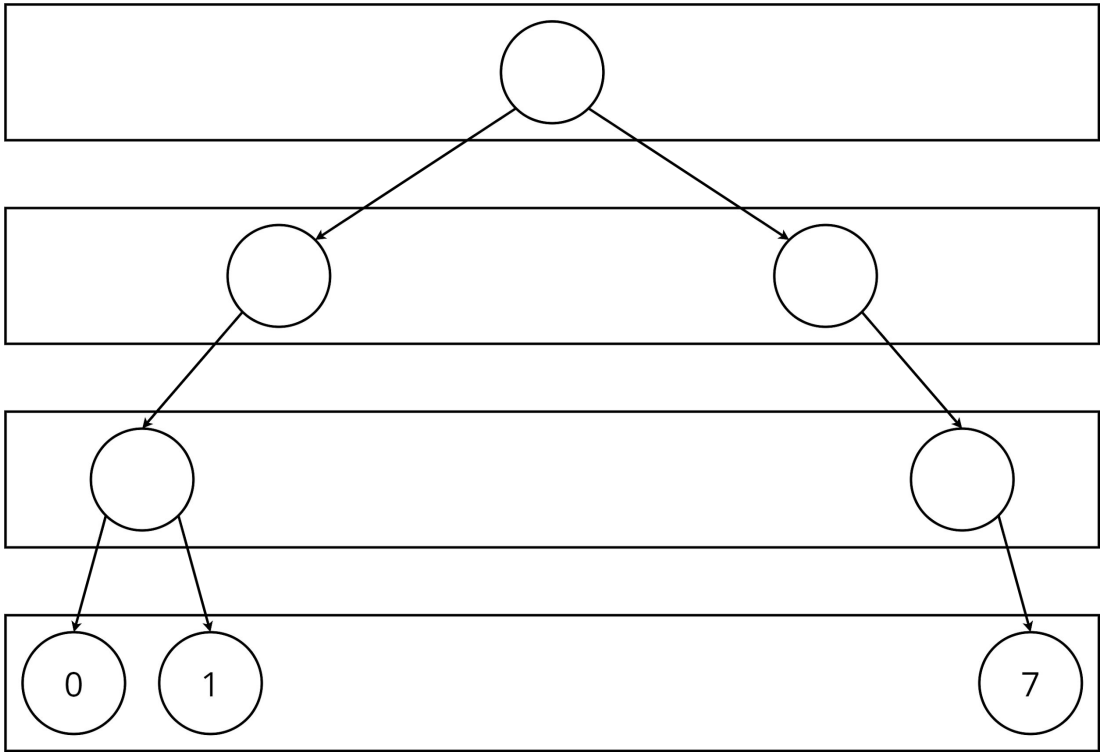
Consider: Set S_7 = {0, 1, 7} on U_7

Idea: Store layers in hash tables.



Consider: Set S_7 = {0, 1, 7} on U_7

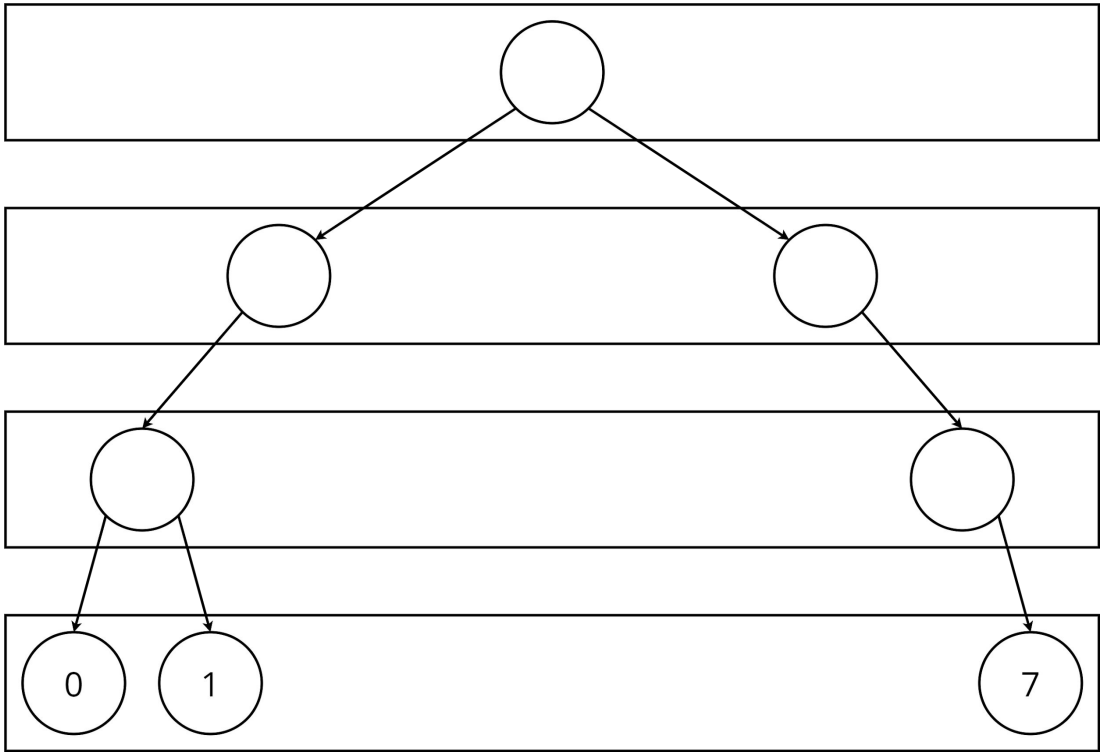
Notation: A more convenient view.



Consider: Set S_7 = {0, 1, 7} on U_7

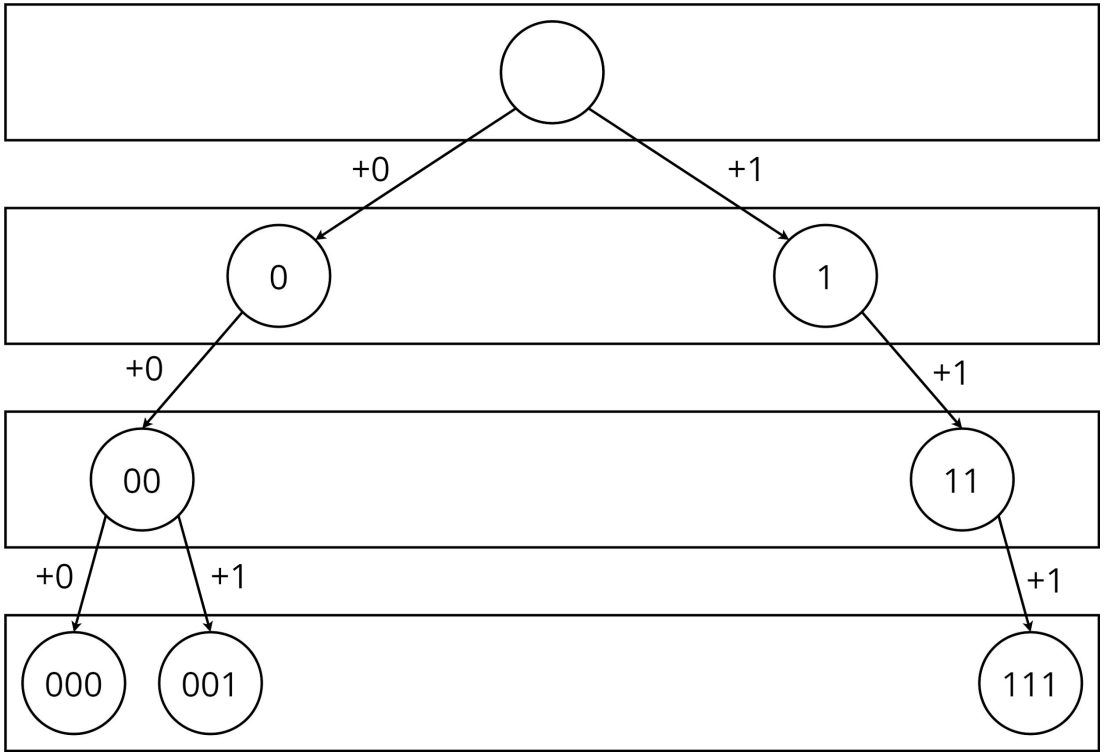
Problem: How to improve predecessor and successor to sublogarithmic with extra pointers?

Idea: Internal nodes represent prefixes.



Consider: Set S_7 = {0, 1, 7} on U_7

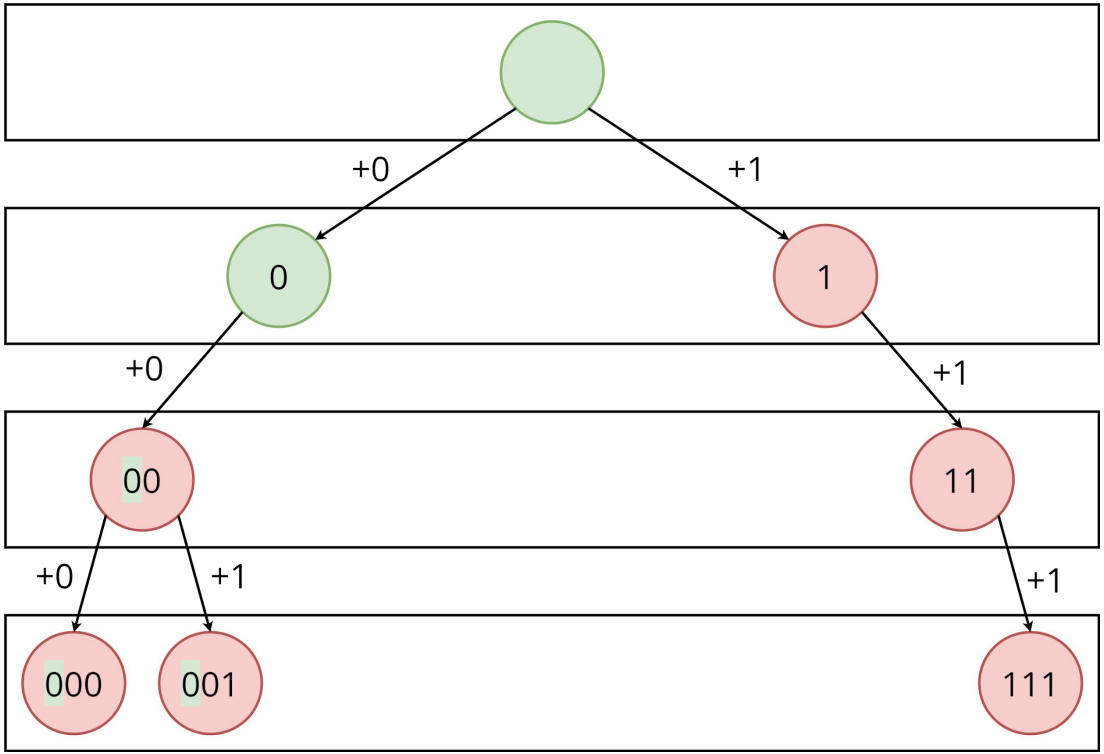
Idea: Internal nodes represent prefixes.



Consider: Set S_7 = {0, 1, 7} on U_7

Idea: Binary search levels for longest matching prefix.

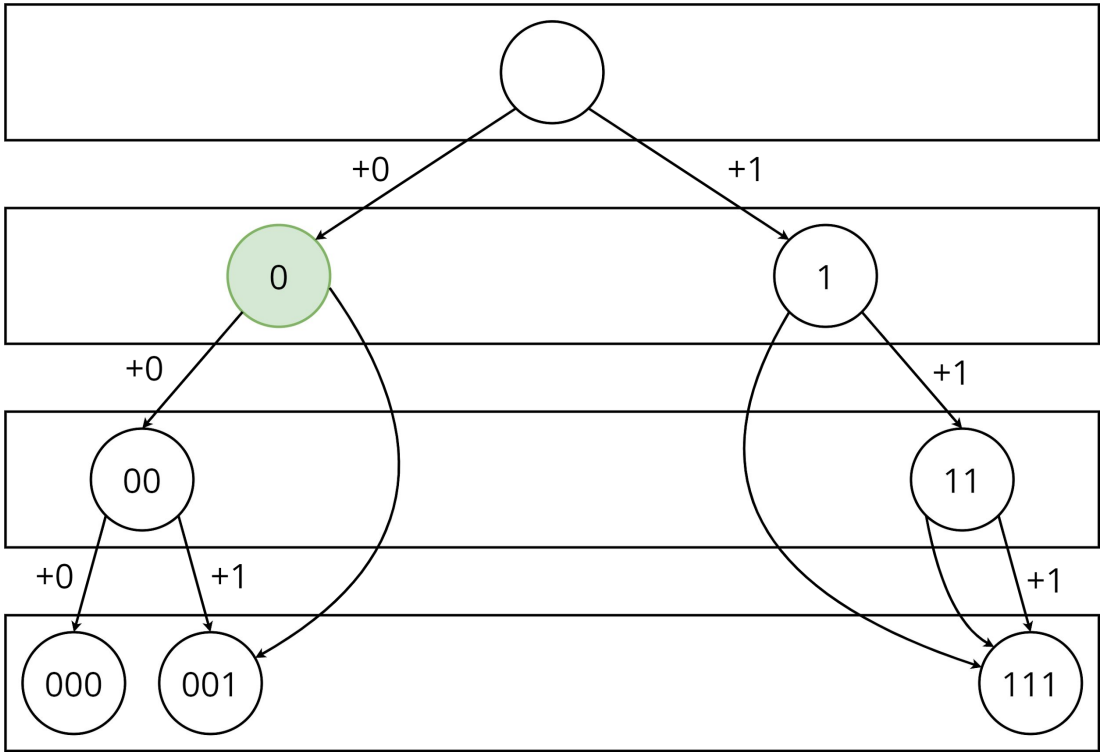
Example: Binary search levels for longest matching prefix of 011 (3).



Consider: Set S_7 = {0, 1, 7} on U_7 and S_7.predecessor(3)

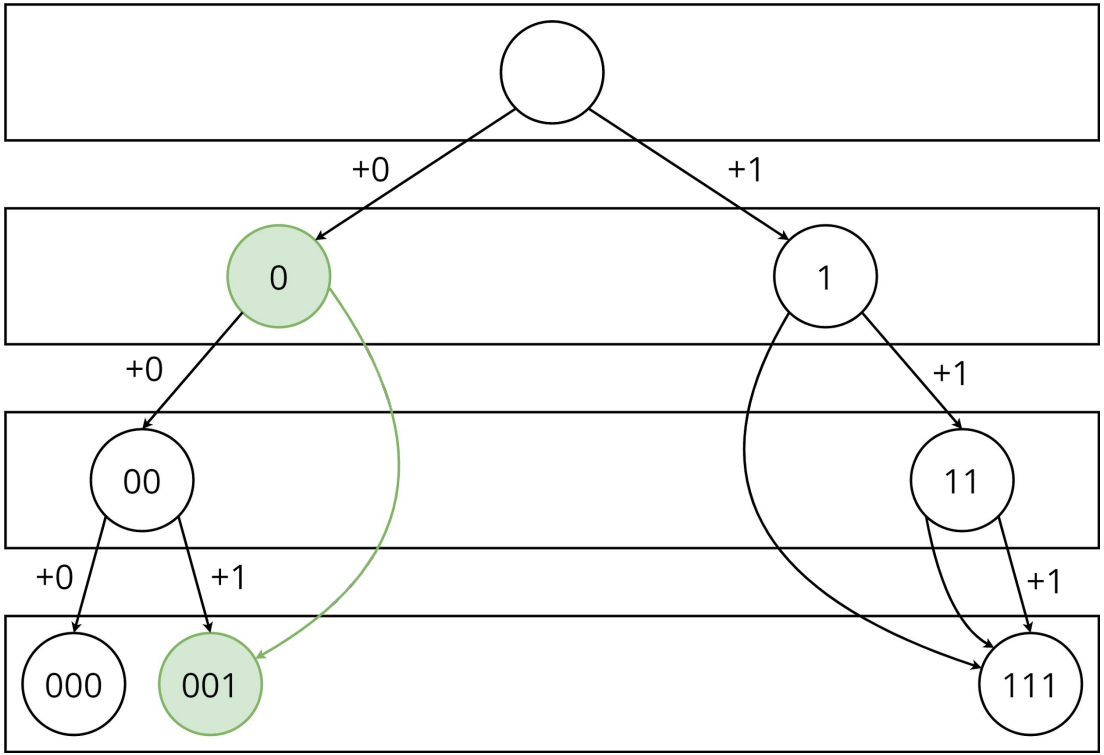
Idea: Pointers to zero are “skip links” to either the predecessor or the successor.

Example: Pointers to zero are “skip links” to pred/succ.



Consider: Set S_7 = {0, 1, 7} on U_7 and S_7.predecessor(3)

Example: Pointers to zero are “skip links” to pred/succ.

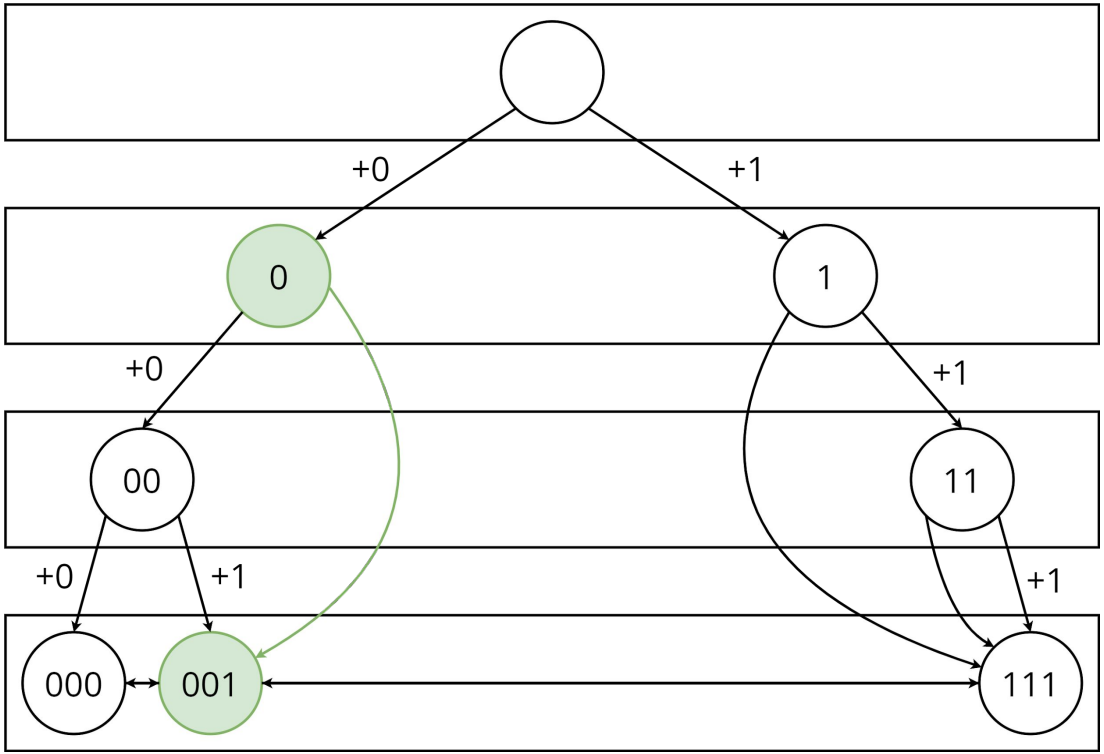


Consider: Set S_7 = {0, 1, 7} on U_7 and S_7.predecessor(3)

Problem: Only either the predecessor or the successor will be found in sublogarithmic time.

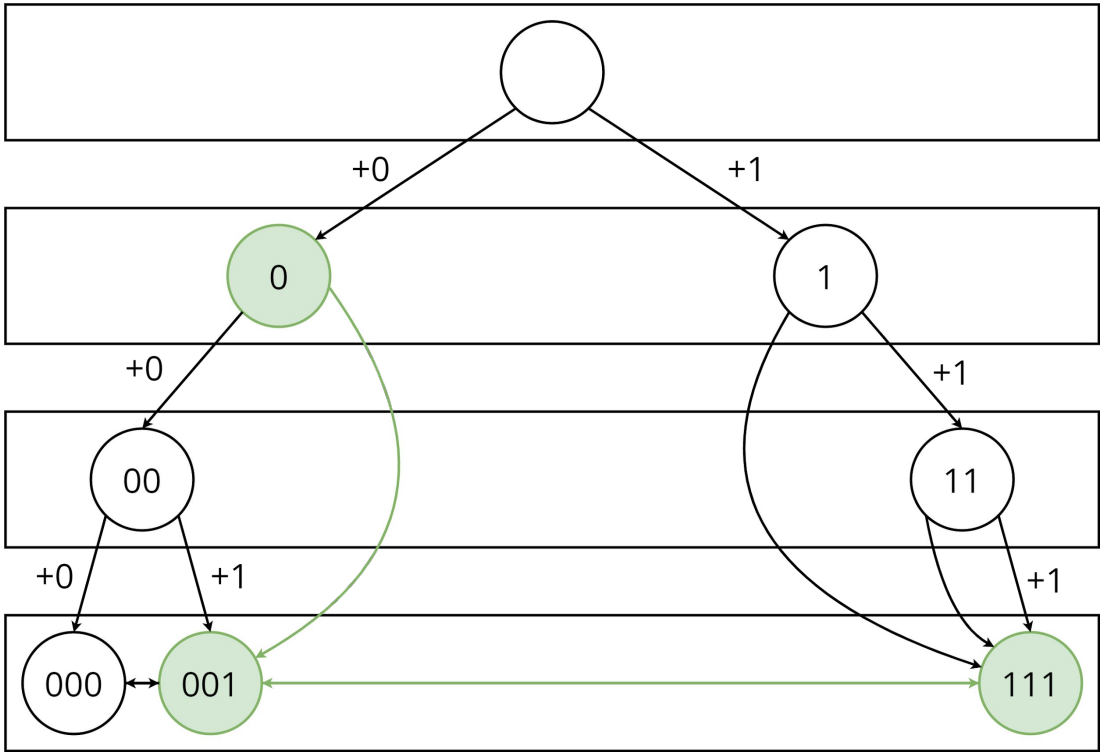
Idea: Store the bottom layer in a linked list.

Example: Store the bottom layer in a linked list.



Consider: Set S_7 = {0, 1, 7} on U_7 and S_7.successor(3)

Example: Store the bottom layer in a linked list.



Consider: Set S_7 = {0, 1, 7} on U_7 and S_7.successor(3)

Solution: That is a X-Fast Trie! Let's take a look at the complexity!

Examples: Dynamic Ordered Set

	insert(x)	remove(x)	predecessor(x)	successor(x)	contains(x)	space
BST	O(N)	O(N)	O(log(N))	O(log(N))	O(N)	O(N)
RB-Tree	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(N)
Sorted Vector	O(N)	O(N)	O(log(N))	O(log(N))	O(log(N))	O(N)
Sorted List	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)
X-Fast Trie	O(log(M))	O(log(M))	O(log(log(M)))	O(log(log(M)))	O(1)	O(N*log(M))
???	O(log(log(M)))	O(log(log(M)))	O(log(log(M)))	O(log(log(M)))	O(1)	O(N)

Problem: Improve insert and remove to sublogarithmic time.

Recall: Improved vector insert from O(N) to O(1).

- Perform expensive O(N) resize operation with O(1/N) probability
  - O(N) * O(1/N) + O(1) = O(1)
- Reduces time complexity from O(N) to O(1)
- Can we apply a similar idea to the X-Fast Trie?

Idea: Insert with O(1/log(M)) probability.

- Perform expensive O(log(M)) operation every O(1/log(M)) times
  - O(log(M)) * O(1/log(M)) + O(1) = O(1)
- Reduces time complexity from O(log(M)) to O(log(log(M)))
- What can we do about the other elements?

Problem: Provide all operations in O(log(log(M))) on subsets of the data.

Recall: Balanced BSTs have O(log(N)) complexity.

Idea: Store O(log(M)) elements in a balanced BST

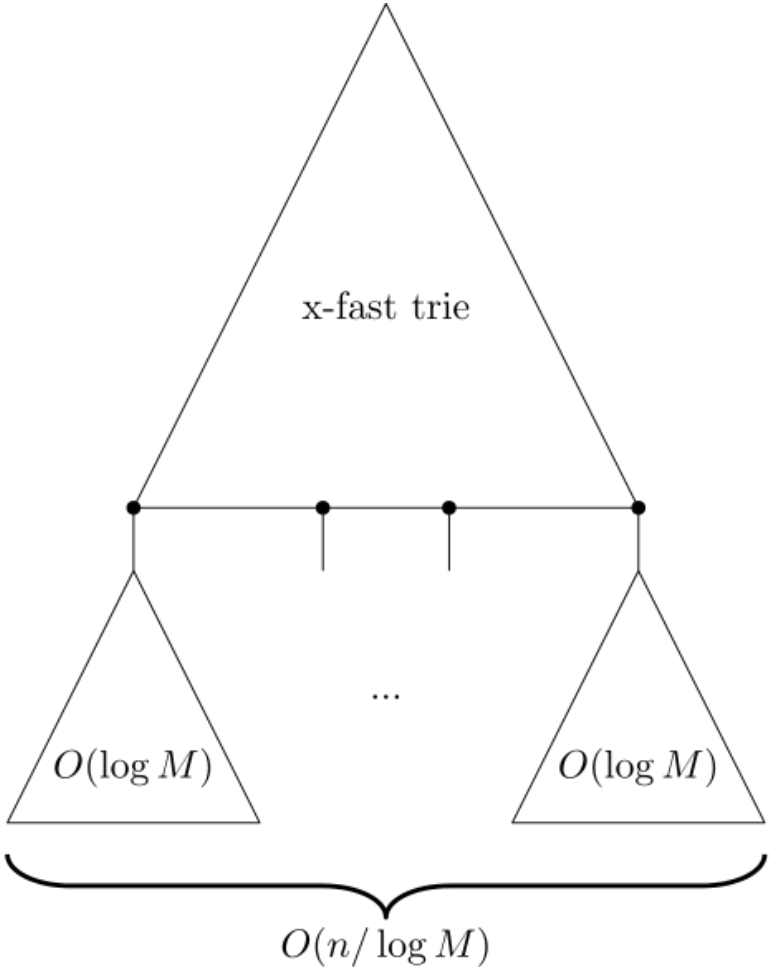
- Assume we only stored O(log(M)) elements in a balanced BST
  - O(log(N)) where M=log(M) is O(log(log(M)))
- How can we place only O(log(M)) elements in a balanced BST?

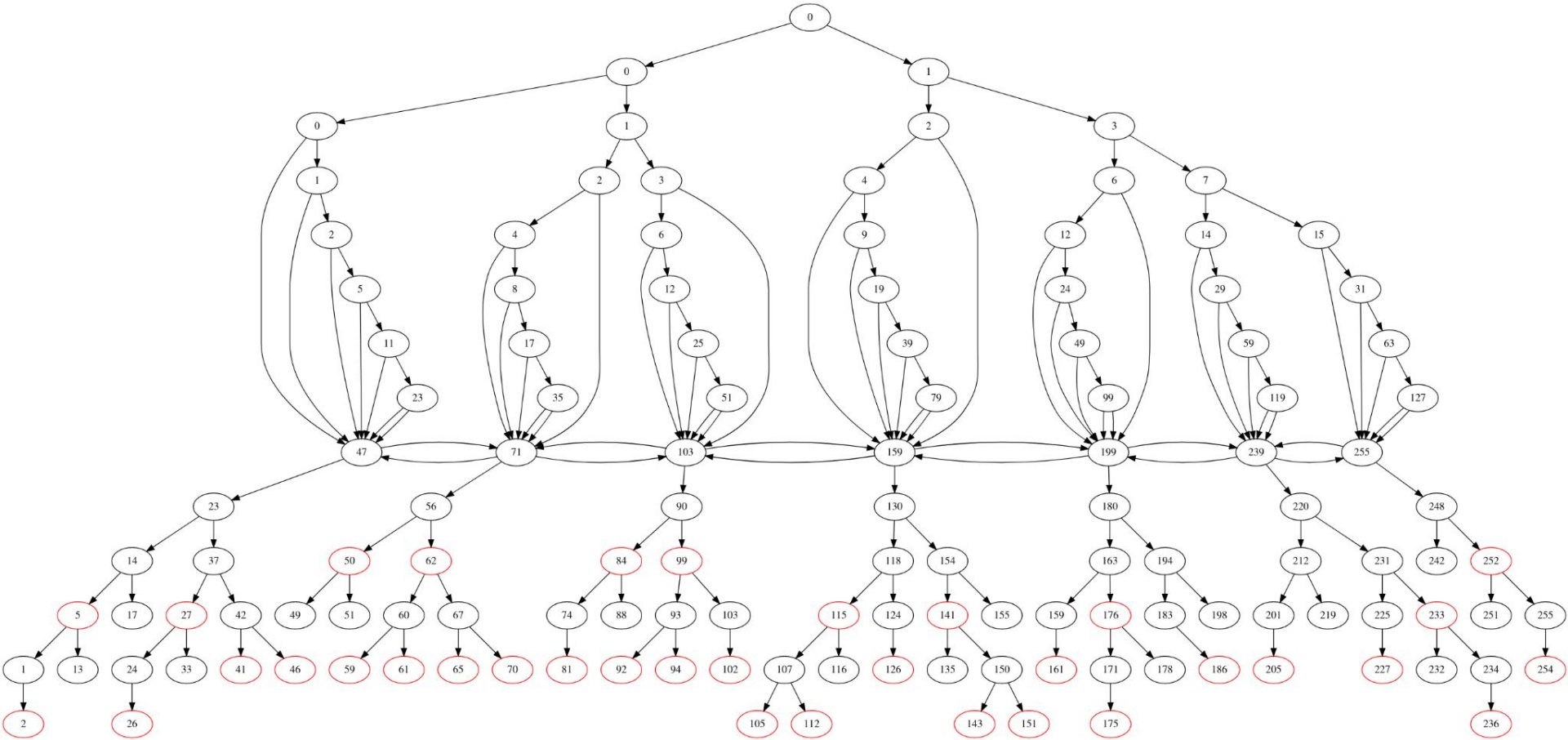
Problem: How can we partition O(N) elements into O(N/log(M)) balanced BSTs?

Idea: Choose **representatives** such that they split the data into O(log(M)) sized partitions.

- Insert the O(1/log(M)) probability **representatives** in the X-Fast Trie for O(1) average complexity
- Access the O(log(M)) sized partitions through a hash table in O(1) time
- Since the partitions are O(log(M)) size, we can perform all operations in O(log(log(M))) time

Diagram:




0
0 1
0 1 2 3
1 2 3 4 6 7
2 4 5 6 8 9 12 14 15
5 8 11 12 17 19 24 29 31
11 17 23 25 35 39 49 59 63
23 35 47 51 71 79 99 119 127
47 71 103 159 199 239 255
23 56 90 130 180 220 248
14 37 50 62 84 99 118 154 163 194 212 231 242 252
5 17 27 42 49 51 60 67 74 88 93 103 115 124 141 155 159 176 183 198 201 219 225 233 251 255
1 13 24 33 41 46 59 61 65 70 81 92 94 102 107 116 126 135 150 161 171 178 186 205 227 232 234 254
2 26 105 112 143 151 175 236

Solution: That is a Y-Fast Trie! Let's take a look at the complexity!

Examples: Dynamic Ordered Set

	insert(x)	remove(x)	predecessor(x)	successor(x)	contains(x)	space
BST	O(N)	O(N)	O(log(N))	O(log(N))	O(N)	O(N)
RB-Tree	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(N)
Sorted Vector	O(N)	O(N)	O(log(N))	O(log(N))	O(log(N))	O(N)
Sorted List	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)
X-Fast Trie	O(log(M))	O(log(M))	O(log(log(M)))	O(log(log(M)))	O(1)	O(N*log(M))
Y-Fast Trie	O(log(log(M)))	O(log(log(M)))	O(log(log(M)))	O(log(log(M)))	O(1)	O(N)

What we didn’t have time to cover:

- How to maintain skip links?
- How to choose **representatives**?
- How to implement priority queue operations?
- How to maintain correctly sized partitions?
- How to split/merge Red Black trees?
- How to implement the data structures efficiently?