# Partial Sums and Dynamic Arrays

- Partial Sums
- Dynamic Arrays

Philip Bille

# Partial Sums and Dynamic Arrays

- Partial Sums
- Dynamic Arrays

# Partial Sums

- Partial sums. Maintain array A[0,1,..., n] of integers support the following operations.
  - SUM(i): return A[1] + A[2] + ... + A[i]
  - UPDATE(i, Δ): set A[i] = A[i] + Δ

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | 2 | 1 | 1 | 0 | 2 | 3 | 1 | 0 | 1 | 3 | 4 | 1 | 1 | 1 | 2 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

# Partial Sums

- Applications.
  - Dynamic lists and arrays (random access into changing lists)
  - Arithmetic coding.
  - Succinct data structures.
  - Lower bounds and cell probe complexity.
  - Basic component in many data structures.

# Partial Sums

- Goal. Partial sums data structure with SUM and UPDATE in optimal O(log n) time and no extra space(!)
- Solution in 4 steps.
  - Explicit array: slow sum and ultra fast update.
  - Explicity partial sum: ultra fast sum and slow update.
  - Balanced binary tree: fast sum and fast update.
  - Fenwick tree: fast sum and fast update and no extra space.

# Solution 1: Explicit Array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | 2 | 1 | 1 | 0 | 2 | 3 | 1 | 0 | 1 | 3 | 4 | 1 | 1 | 1 | 2 |

- Slow sum and ultra fast update. Maintain A explicitly.
  - SUM(i): compute A[0] + .. + A[i].
  - UPDATE(i, Δ): set A[i] = A[i] + Δ
- Time.
  - O(i) = O(n) for SUM, O(1) for UPDATE.

# Solution 2: Explicit Partial Sum

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | 3 | 4 | 5 | 5 | 7 | 10 | 11 | 11 | 12 | 15 | 19 | 20 | 21 | 22 | 24 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

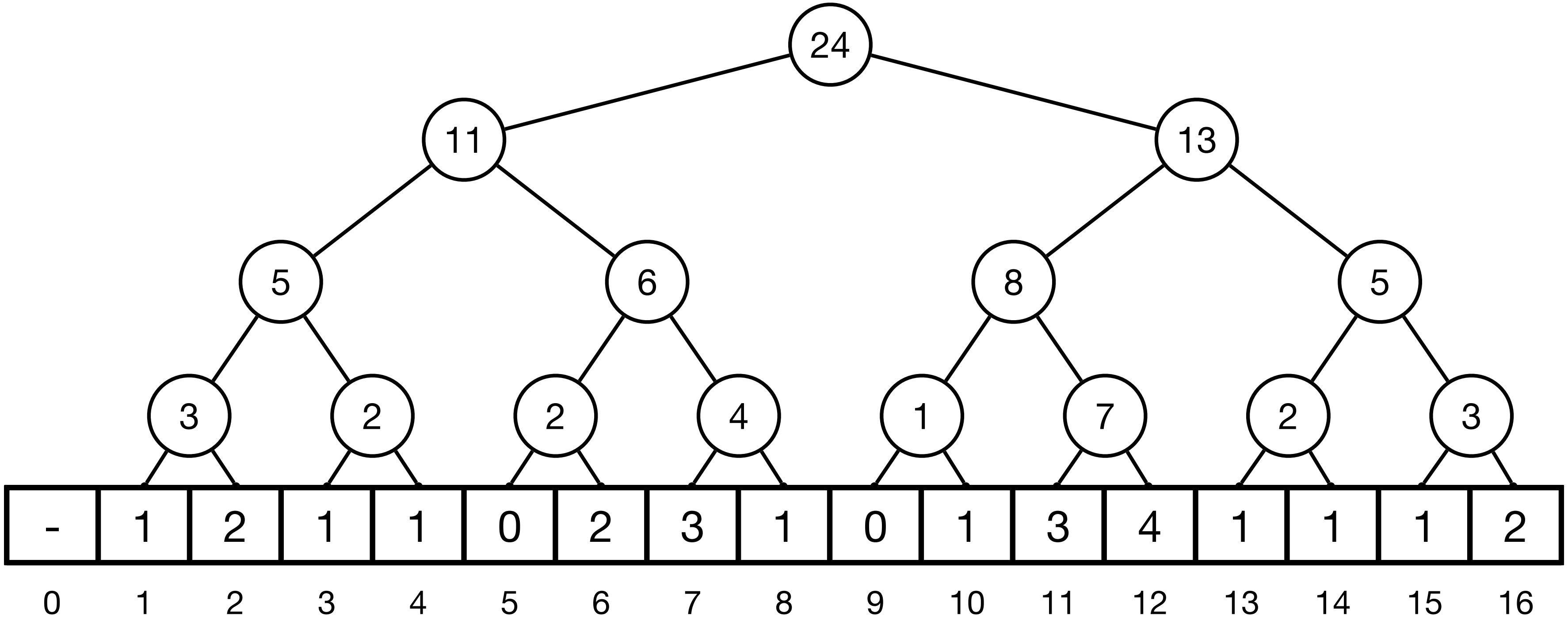
| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | 2 | 1 | 1 | 0 | 2 | 3 | 1 | 0 | 1 | 3 | 4 | 1 | 1 | 1 | 2 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

- Ultra fast sum and slow update. Maintain partial sum P of A.
  - SUM(i): return P[i].
  - UPDATE(i, Δ): add Δ to P[i], P[i+1], ..., P[n].
- Time.
  - O(1) for SUM, O(n - i + 1) = O(n) for UPDATE.

# Partial Sums

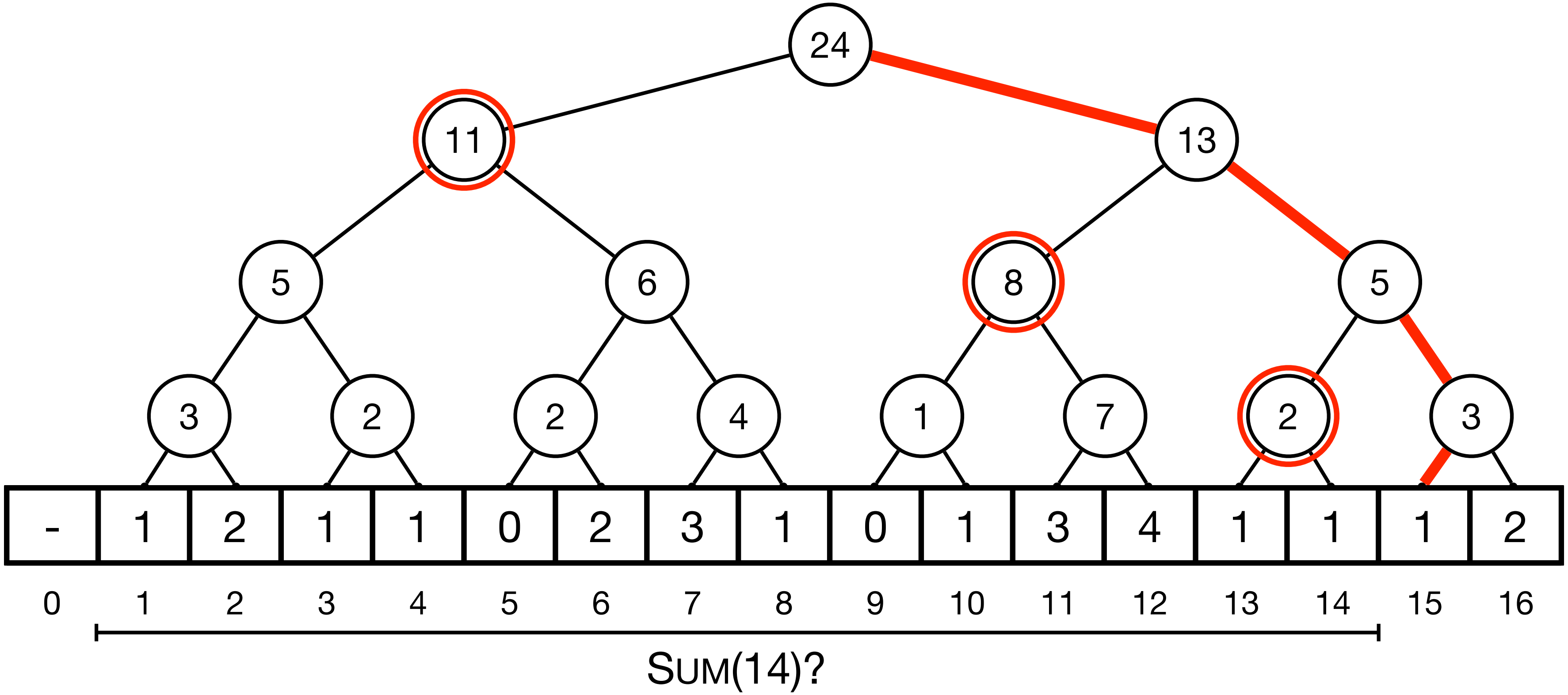
| Data structure | SUM | UPDATE | Space |
|---|---|---|---|
| explicit array | O(n) | O(1) | O(n) |
| explicit partial sum | O(1) | O(n) | O(n) |

# Solution 3: Balanced Binary Tree



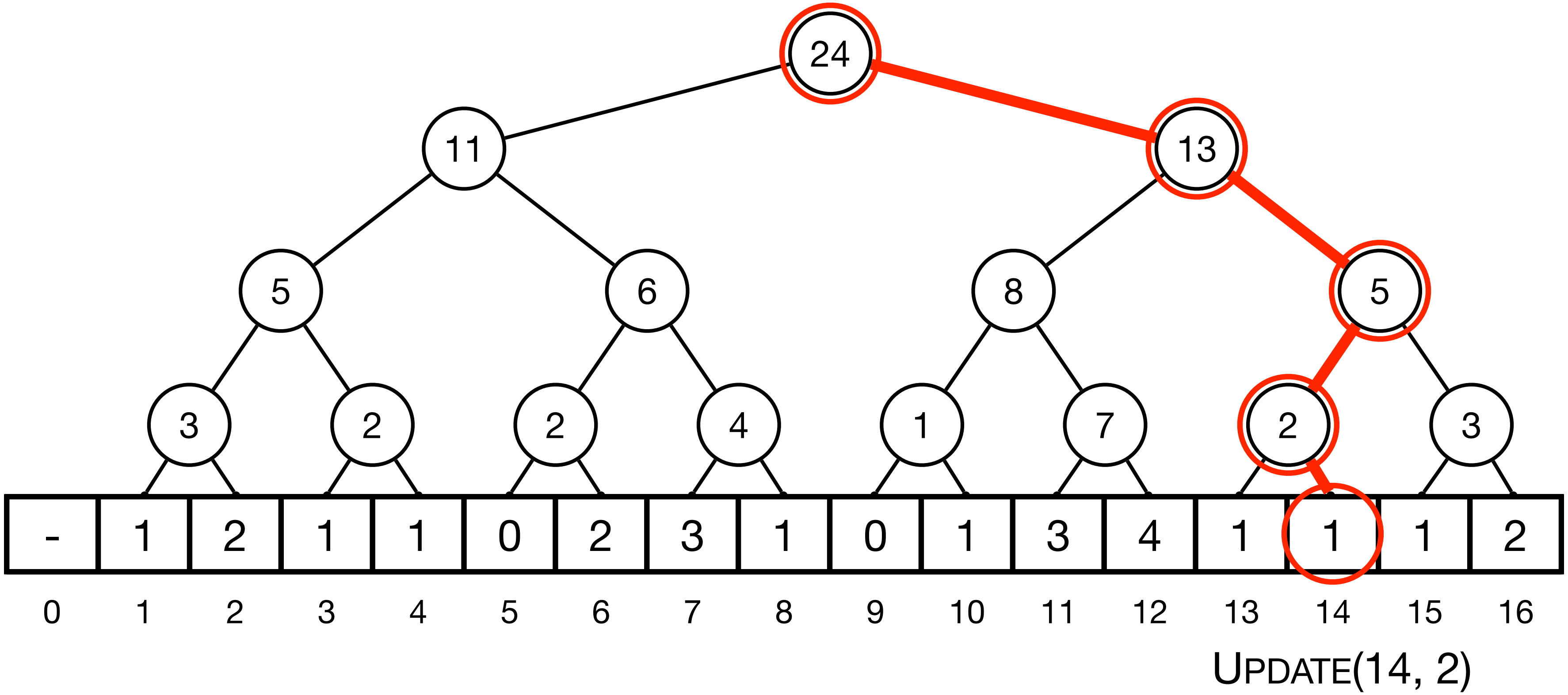

- Fast sum and fast update. Maintain balanced binary tree T on A. Each node stores the sum of elements in subtree.

# Solution 3: Balanced Binary Tree



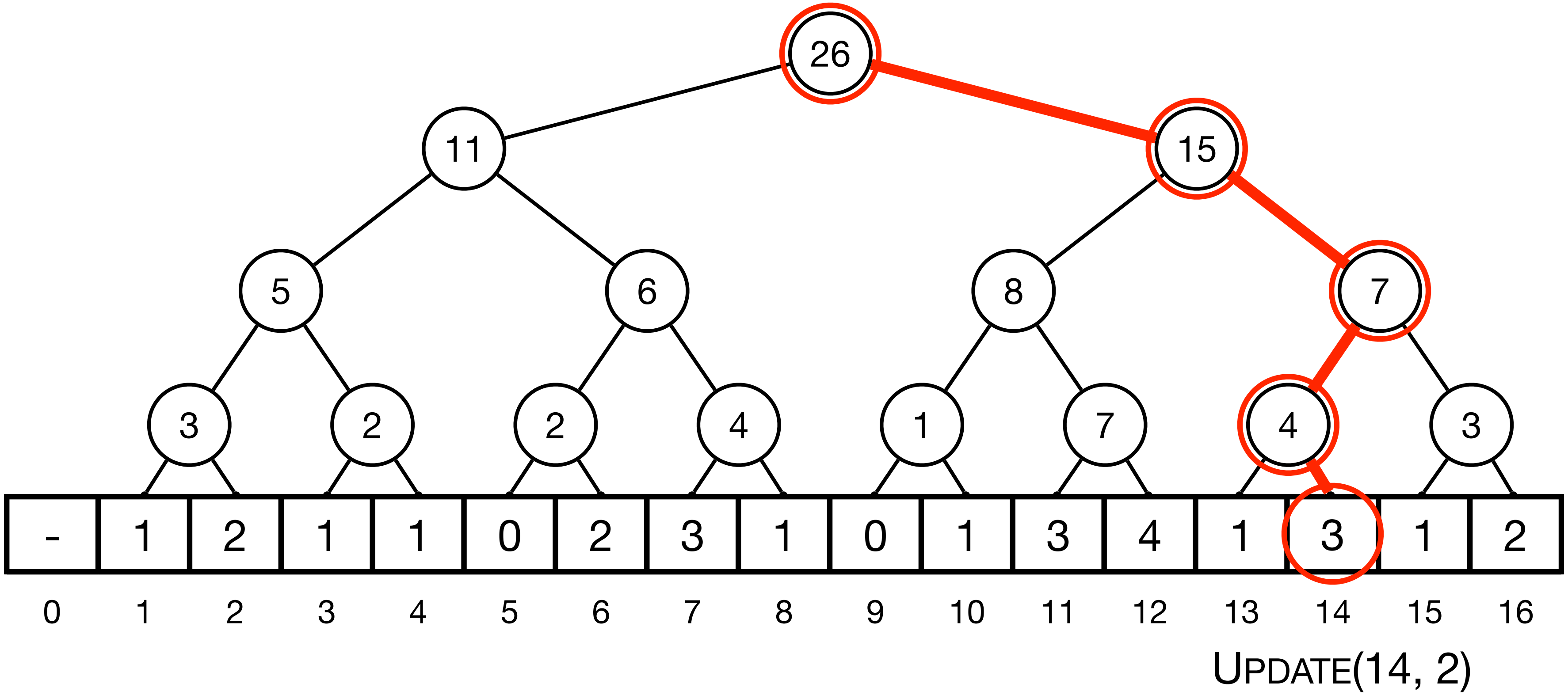

- SUM.
  - SUM(i): traverse path to i + 1 and sum up all off-path nodes.
- Time. O(log n)

# Solution 3: Balanced Binary Tree




- UPDATE.
  - UPDATE(i, Δ): add Δ to nodes on path to i.

# Solution 3: Balanced Binary Tree




- UPDATE.
  - UPDATE(i, Δ): add Δ to nodes on path to i.
- Time. O(log n)

# Partial Sums

| Data structure | SUM | UPDATE | Space |
|---|---|---|---|
| explicit array | O(n) | O(1) | O(n) |
| explicit partial sum | O(1) | O(n) | O(n) |
| balanced binary tree | O(log n) | O(log n) | O(n) |
| lower bound | Ω(log n) | Ω(log n) | |

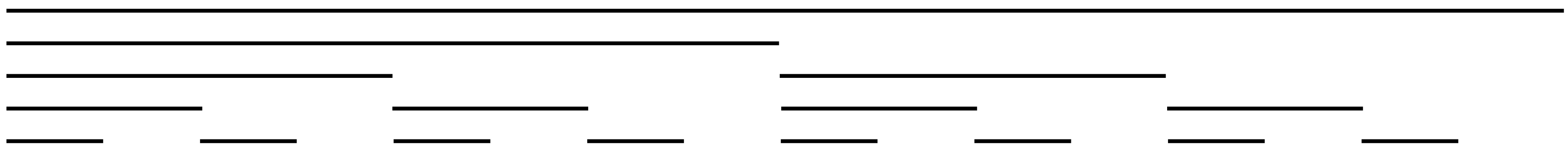
- Challenge. How can we improve?
- In-place data structure.
  - Replace input array A with data structure of exactly same size.
  - Use only O(1) extra space.

# Solution 4: Fenwick Tree

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | 3 | 1 | 5 | 0 | 2 | 3 | 11 | 0 | 1 | 3 | 8 | 1 | 2 | 1 | 24 |
| - | 1 | 3 | 1 | 5 | 0 | 2 | 3 | 11 | 0 | 1 | 3 | 8 | 1 | 2 | 1 | 13 |
| - | 1 | 3 | 1 | 5 | 0 | 2 | 3 | 6 | 0 | 1 | 3 | 8 | 1 | 2 | 1 | 5 |
| - | 1 | 3 | 1 | 2 | 0 | 2 | 3 | 4 | 0 | 1 | 3 | 7 | 1 | 2 | 1 | 3 |
| - | 1 | 2 | 1 | 1 | 0 | 2 | 3 | 1 | 0 | 1 | 3 | 4 | 1 | 1 | 1 | 2 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

- Fenwick tree. Replace A by another array F.
  - Replace all even entries A[2i] by A[2i - 1] + A[2i].
  - Recurse on the entries A[2, 4, .., n] until we are left with a single element.

# Solution 4: Fenwick Tree

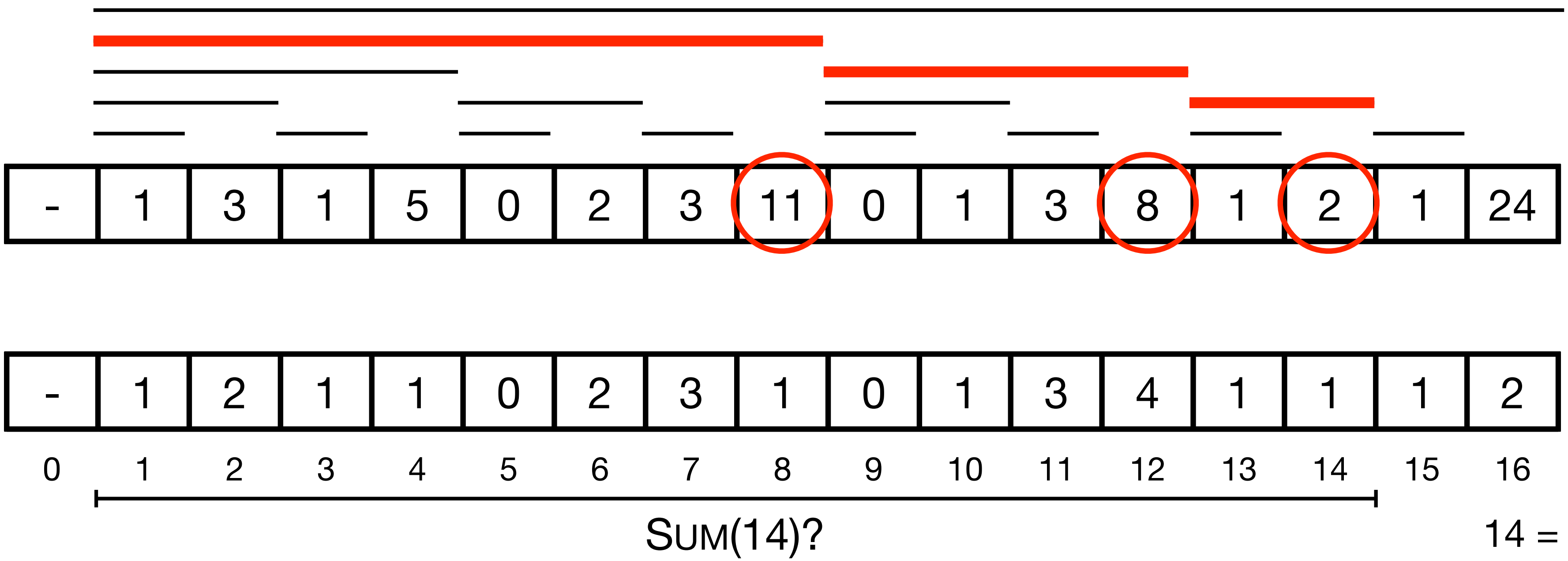


| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | 3 | 1 | 5 | 0 | 2 | 3 | 11 | 0 | 1 | 3 | 8 | 1 | 2 | 1 | 24 |

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | 2 | 1 | 1 | 0 | 2 | 3 | 1 | 0 | 1 | 3 | 4 | 1 | 1 | 1 | 2 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

- Fenwick tree. Replace A by another array F.
  - Replace all even entries A[2i] by A[2i - 1] + A[2i].
  - Recurse on the entries A[2, 4, .., n] until we are left with a single element.
- Space.
  - In-place. No extra space.

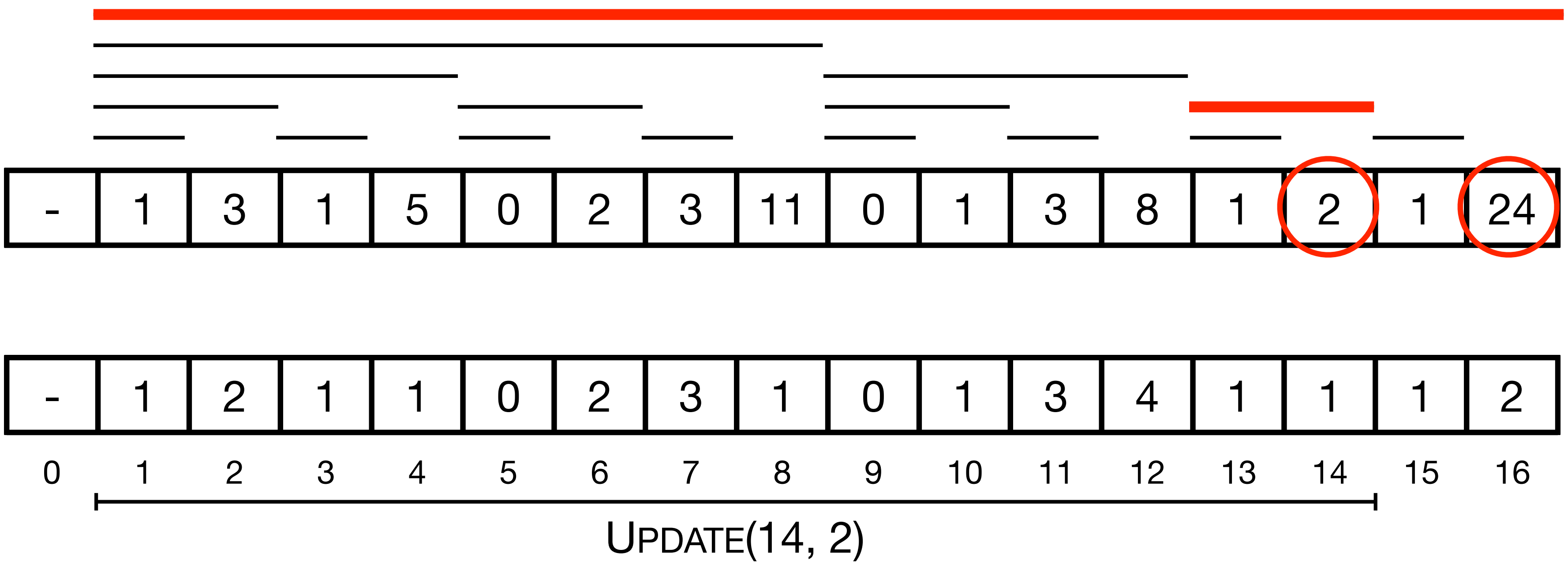
# Solution 4: Fenwick Tree




$14 = 1110_2$
$12 = 1100_2$
$8 = 1000_2$
$0 = 0000_2$

- SUM.
  - SUM(i): add largest partial sums covering [1,..,i].
  - Indexes $i_0$, $i_1$, .. in F given by $i_0 = i$ and $i_{j+1} = i_j - \text{rmb}(i_j)$, where $\text{rmb}(i_j)$ is the integer corresponding to the rightmost 1-bit in i. Stop when we get 0.
- Time. O(log n)

# Solution 4: Fenwick Tree




$14 = 1110_2$
$16 = 10000_2$

- UPDATE.
  - UPDATE(i, Δ): add Δ to partial sums covering i.
  - Indexes $i_0, i_1, ..$ in F given by $i_0 = i$ and $i_{j+1} = i_j + \text{rmb}(i_j)$. Stop when we get n.
- Time. O(log n)

# Partial Sums

| Data structure | SUM | UPDATE | Space |
| --- | --- | --- | --- |
| explicit array | O(n) | O(1) | O(n) |
| explicit partial sum | O(1) | O(n) | O(n) |
| balanced binary tree | O(log n) | O(log n) | O(n) |
| lower bound | Ω(log n) | Ω(log n) | |
| Fenwick tree | O(log n) | O(log n) | in-place |

- Practical? Fenwick trees for competitive programming.

# Partial Sums and Dynamic Arrays

- Partial Sums
- Dynamic Arrays

# Dynamic Arrays

- Dynamic arrays. Maintain array A[0,..., n-1] of integers and support the following operations.
  - ACCESS(i): return A[i].
  - INSERT(i, x): insert a new entry with value x immediately to the left of entry i.
  - DELETE(i): Remove entry i.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 1 | 0 | 2 | 3 | 1 | 0 | 1 | 3 | 4 | 1 | 1 | 1 | 2 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

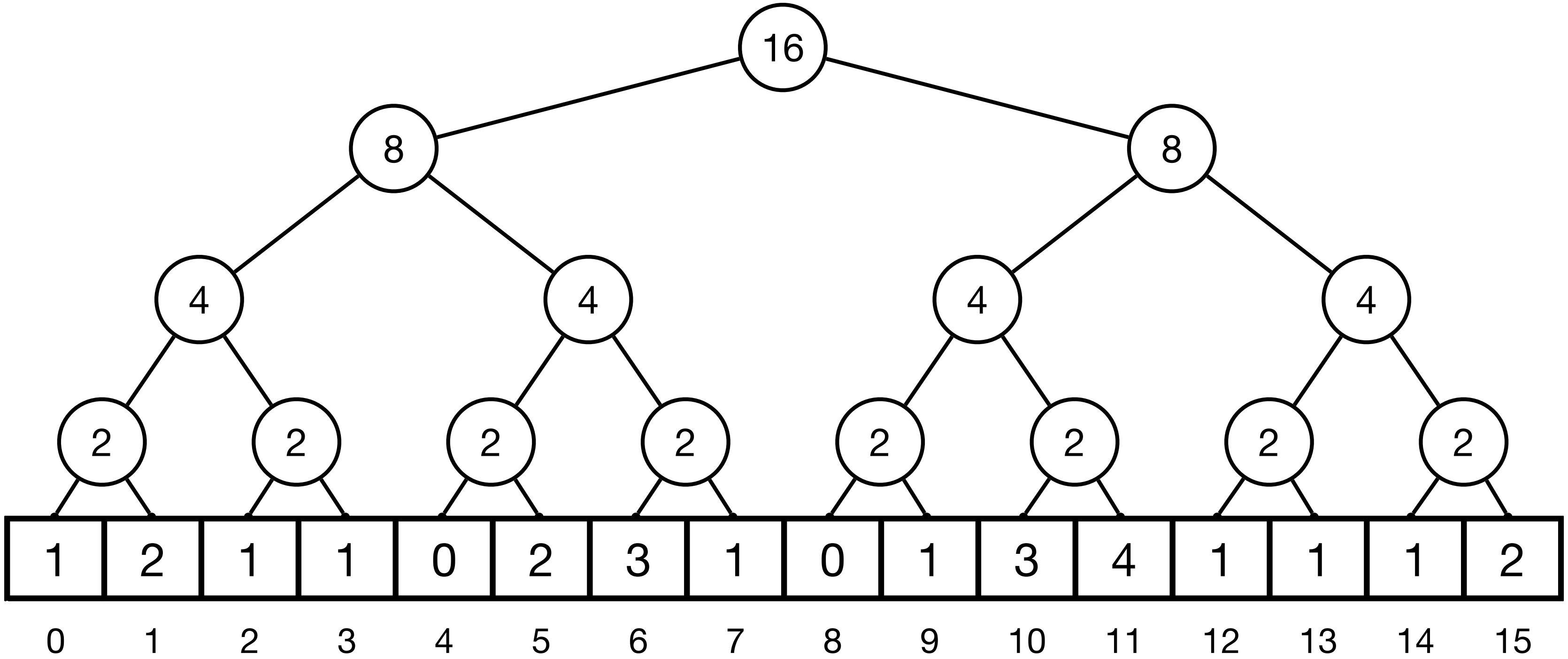
# Dynamic Arrays

- Applications.
    - Dynamic lists and arrays (random access into changing lists)
    - Basic component in many data structures.

# Dynamic Arrays

- Goal. Dynamic array data structure with linear space that supports ACCESS in O(1) time and INSERT and DELETE in $O(\sqrt{n})$ time.
- Solution in 4 steps.
  - Balanced binary tree: fast access and fast update.
  - Explicit array: ultra fast access and slow update.
  - Rotated array: ultra fast access and slow update (but with ultrafast updates at endpoints).
  - 2-level rotated array: ultra fast access and faster update.

# Solution 1: Balanced Binary Tree




- Fast access and fast update. Maintain balanced binary tree T on A. Each node stores the number of elements in subtree.
  - ACCESS(i): traverse path to leaf j.
  - INSERT(i, x): insert new leaf and update tree.
  - DELETE(i): delete new leaf and update tree.
- Time. O(log n) for ACCESS, INSERT, and DELETE.

# Solution 2: Explicit Array

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 1 | 0 | 2 | 3 | 1 | 0 | 1 | 3 | 4 | 1 | 1 | 1 | 2 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

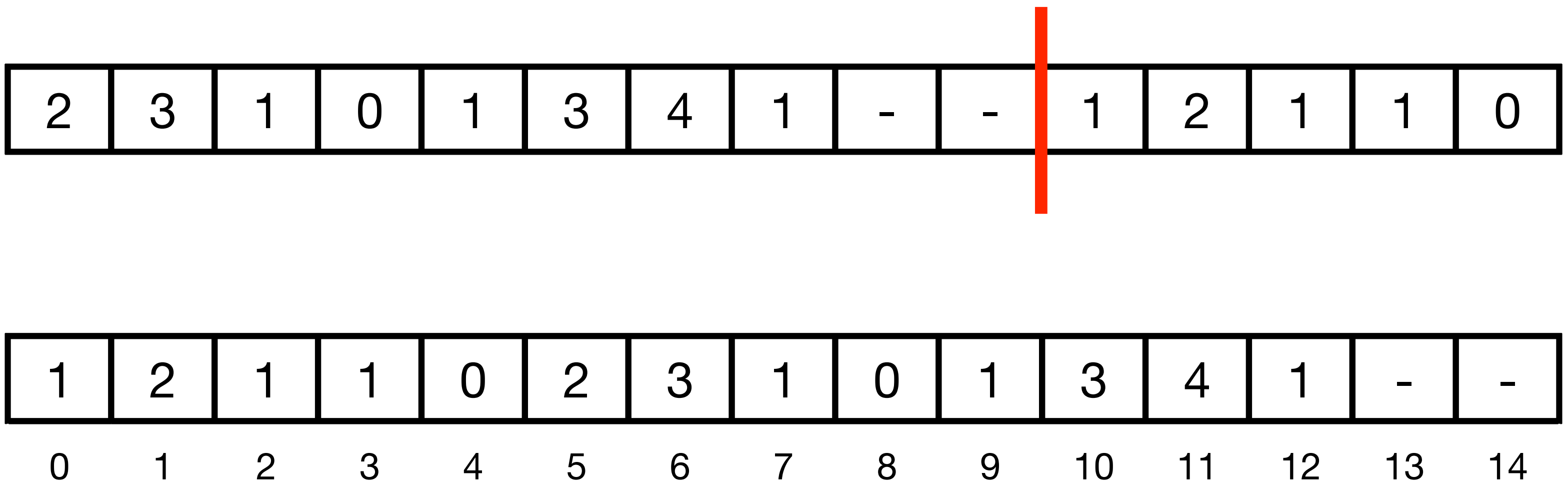
- Ultra fast access and slow update. Maintain A explicitly.
    - ACCESS(i): return A[i].
    - INSERT(i, x): Shift all elements from i to n by 1 to the right. Set A[i] = x.
    - DELETE(i): shift all elements to the right of entry i to the left by 1.
- Time.
    - O(1) for ACCESS and O(n-i+1) = O(n) for INSERT and DELETE.

# Dynamic Arrays

| Data structure | ACCESS | INSERT | DELETE | Space |
|---|---|---|---|---|
| balanced binary tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(n)$ |
| explicit array | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| lower bound | $\Omega(\log n/\log\log n)$ | $\Omega(\log n/\log\log n)$ | $\Omega(\log n/\log\log n)$ | |

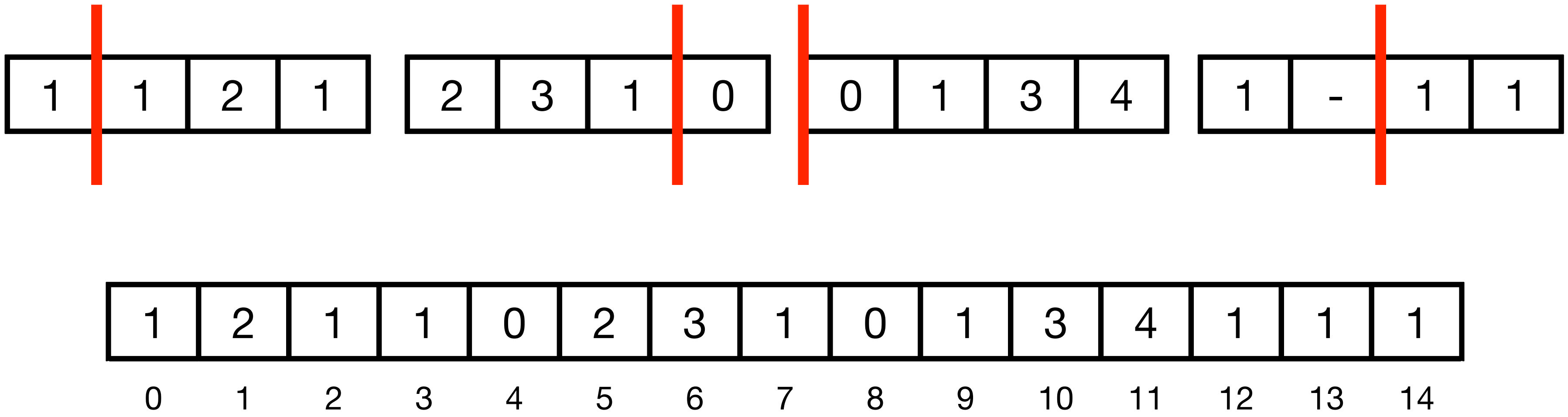
- Challenge. What can we get if we insist on constant time ACCESS?

# Solution 3: Rotated array



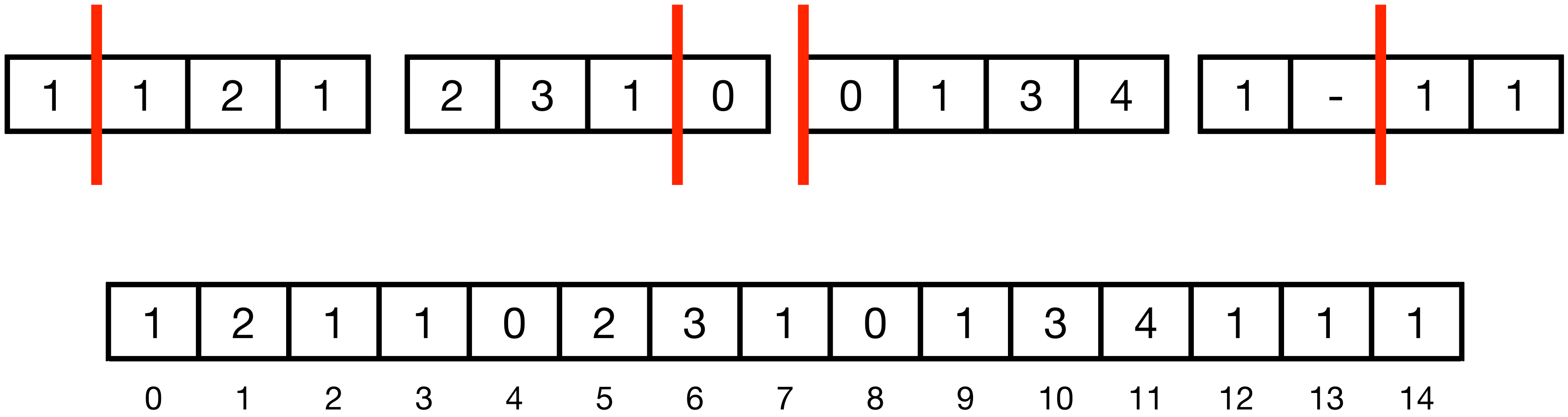

- Rotated array with capacity N. Maintain a circular shift of array in an array with capacity N. Store offset to mark start of array.
  - ACCESS, INSERT, and DELETE as in solution 1 but shifted by offset.
- Time. O(1) for ACCESS and O(n) for INSERT and DELETE.
- INSERT and DELETE at endpoints in O(1) time.

# Solution 4: 2-level Rotated Arrays



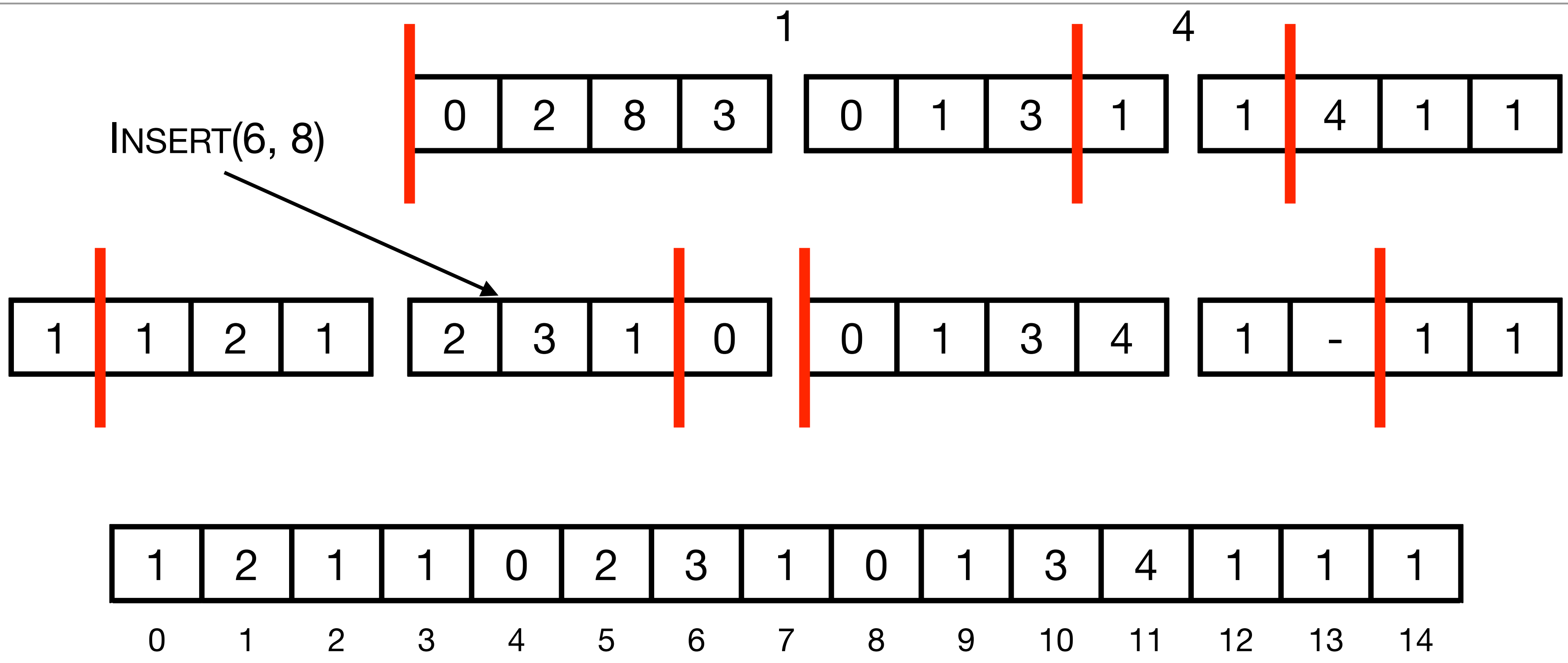

- 2-level rotated arrays.
  - Store $\sqrt{n}$ rotated arrays $R_0$, ..., $R_{\sqrt{n}-1}$ with capacity $\sqrt{n}$ (last may have smaller capacity).

# Solution 4: 2-level Rotated Arrays



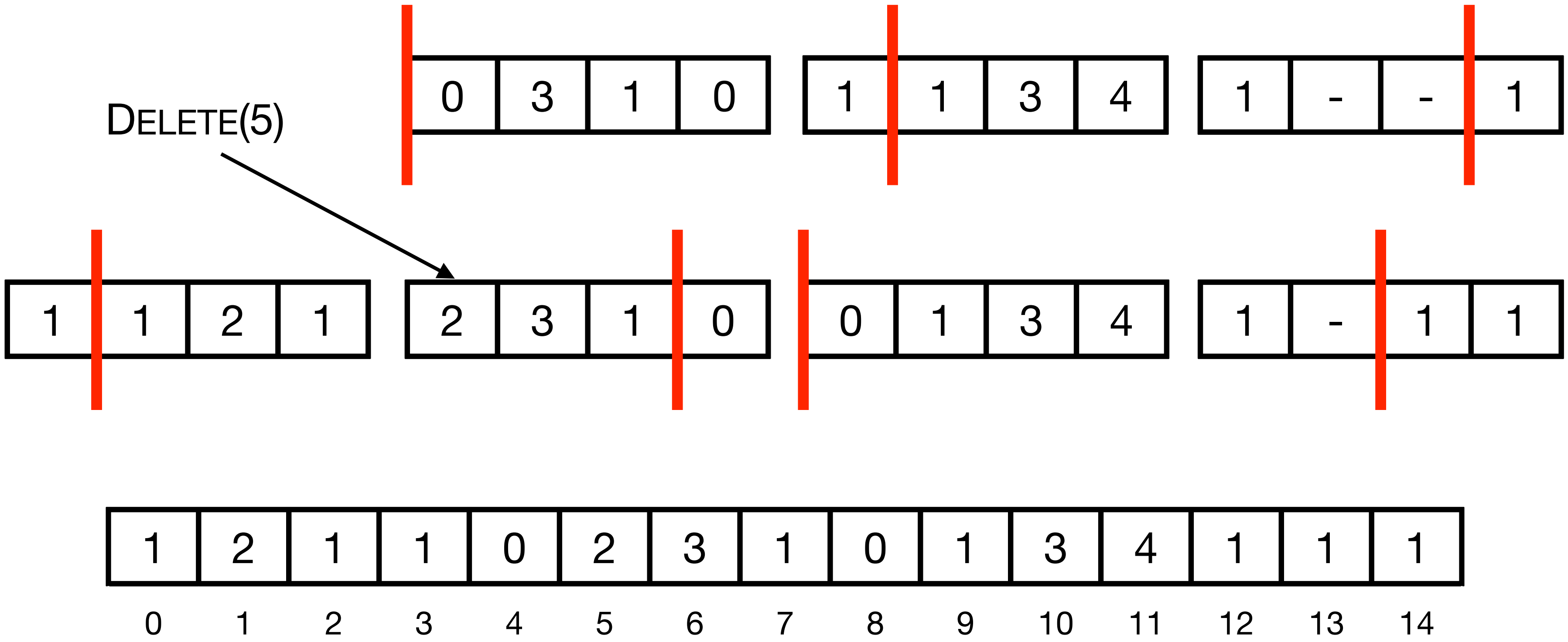

- Access.
  - Access(i): compute rotated array $R_j$ and index k corresponding to i. Return $R_j[k]$.
- Time. O(1)

# Solution 4: 2-level Rotated Arrays




- INSERT.
  - INSERT(i, x): find $R_j$ and k as in ACCESS.
    - Rebuild $R_j$ with new entry inserted.
    - Propagate overflow to $R_{j+1}$ recursively.
- Time. $O(\sqrt{n})$

# Solution 4: 2-level Rotated Arrays




- DELETE.
  - DELETE(i): find $R_j$ and k as in ACCESS.
    - Rebuild $R_j$ with entry i deleted.
    - Propagate underflow to $R_{j+1}$ recursively.
- Time. $O(\sqrt{n})$

# Dynamic Arrays

| Data structure | ACCESS | INSERT | DELETE | Space |
|---|---|---|---|---|
| balanced binary tree | O(log n) | O(log n) | O(log n) | O(n) |
| explicit array | O(1) | O(n) | O(n) | O(n) |
| lower bound | Ω(log n/log log n) | Ω(log n/log log n) | Ω(log n/log log n) | |
| 2-level rotated array | O(1) | $O(\sqrt{n})$ | $O(\sqrt{n})$ | O(n) |
| O(1)-level rotated array | O(1) | $O(n^{\varepsilon})$ | $O(n^{\varepsilon})$ | O(n) |

# Partial Sums and Dynamic Arrays

- Partial Sums
- Dynamic Arrays