Partial Sums

- Partial sums. Maintain array $A[0, 1, \ldots, n]$ of integers support the following operations.
  - $\text{UPDATE}(i, \Delta)$: set $A[i] = A[i] + \Delta$

- 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.

- Challenge. How can solve the problem with current techniques?
• **Slow sum and ultra fast updates.** Maintain A explicitly.
  • SUM(i): compute $\text{A}[0] + \ldots + \text{A}[i]$.
  • UPDATE(i, $\Delta$): set $\text{A}[i] = \text{A}[i] + \Delta$.
  • **Time.**
  • $O(i)$ for SUM, $O(1)$ for UPDATE.

• **Ultra fast sum and slow updates.** Maintain partial sum $P$ of A.
  • SUM(i): return $P[i]$.
  • UPDATE(i, $\Delta$): add $\Delta$ to $P[i]$, $P[i+1]$, ..., $P[n-1]$.
  • **Time.**
  • $O(1)$ for SUM, $O(n - i + 1) = O(n)$ for UPDATE.

• **Fast sum and fast updates.** Maintain balanced binary tree $T$ on A. Each node stores the sum of elements in subtree.
  • SUM(i): traverse path to $i + 1$ and sum up all off-path nodes.
  • **Time.** $O(\log n)$
**Update**.
- **Update(i, Δ)**: add Δ to nodes on path to i.

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

### Partial Sums

<table>
<thead>
<tr>
<th>Data structure</th>
<th>SUM</th>
<th>UPDATE</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit array</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>explicit partial sum</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>balanced binary tree</td>
<td>O(\log n)</td>
<td>O(\log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>lower bound</td>
<td>O(\log n)</td>
<td>O(\log n)</td>
<td></td>
</tr>
</tbody>
</table>

### Partial Sums

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

- **Time.** O(\log n)
**Partial Sums**

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

**Update**

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

---

**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 | \( O(\log n) \) | \( O(\log n) \) | \( O(n) \)
Fenwick tree | \( O(\log n) \) | \( O(\log n) \) | in-place
Data Structures II

- Partial Sums
- Dynamic Arrays

Dynamic Arrays

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

- **Challenge.** How can solve the problem with current techniques?

- **Dynamic Arrays.** Maintain array $A[0, ..., n-1]$ of integers support the following operations.
  - $\text{ACCESS}(i)$: return $A[i]$.
  - $\text{INSERT}(i, x)$: insert a new entry with value $x$ immediately to the right of entry $i$.
  - $\text{DELETE}(i)$: Remove entry $i$.

  ![Dynamic Arrays example](image)

- **Very fast access and slow updates.** Maintain $A$ explicitly.
  - $\text{ACCESS}(i)$: return $A[i]$.
  - $\text{INSERT}(i, x)$: set $A[i] = x$. Shift all elements to the right of entry $i$ to the right by 1.
  - $\text{DELETE}(i)$: shift all elements to the right of entry $i$ to the left by 1.

- **Time.**
  - $O(1)$ for $\text{ACCESS}$ and $O(n-i+1) = O(n)$ for $\text{INSERT}$ and $\text{DELETE}$. 

  ![Dynamic Arrays example](image)
Fast access and fast updates. 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.

### Dynamic Arrays

<table>
<thead>
<tr>
<th>Data structure</th>
<th>ACCESS</th>
<th>INSERT</th>
<th>DELETE</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>balanced binary tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>lower bound</td>
<td>$\Omega(\log n/\log \log n)$</td>
<td>$\Omega(\log n/\log \log n)$</td>
<td>$\Omega(\log n/\log \log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

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

### Rotated array

- Circular shift of array by an offset.
- **Idea.** By moving offset we can delete and insert at endpoints in $O(1)$ time.
- Lead to underflow or overflow.

### 2-level rotated arrays

- Store $\sqrt{n}$ rotated arrays $R_0, \ldots, R_{\sqrt{n}-1}$ with capacity $\sqrt{n}$ (last may have smaller capacity).
• **ACCESS**.
  • ACCESS(i): compute rotated array $R_i$ and index $k$ corresponding to $i$. Return $R_i[k]$.
  • Time. $O(1)$

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

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

<table>
<thead>
<tr>
<th>Data structure</th>
<th>ACCESS</th>
<th>INSERT</th>
<th>DELETE</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>balanced binary tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>lower bound</td>
<td>$O(\log n / \log \log n)$</td>
<td>$O(\log n / \log \log n)$</td>
<td>$O(\log n / \log \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>2-level rotated array</td>
<td>$O(1)$</td>
<td>$O(\sqrt{n})$</td>
<td>$O(\sqrt{n})$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>O(1)-level rotated array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Data Structures II

- Partial Sums
- Dynamic Arrays