Data Structures II

• Partial Sums
• Dynamic Arrays
Data Structures II

• Partial Sums

• Dynamic Arrays
Partial Sums

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

```
 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
```
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.

• Challenge. How can solve the problem with current techniques?
Partial Sums

- Slow sum and ultra fast updates. Maintain $A$ explicitly.
  - $\text{UPDATE}(i, \Delta)$: set $A[i] = A[i] + \Delta$
- Time.
  - $O(i) = O(n)$ for $\text{SUM}$, $O(1)$ for $\text{UPDATE}$.
Partial Sums

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

• **Time.**
  - $O(1)$ for $\text{SUM}$, $O(n - i + 1) = O(n)$ for $\text{UPDATE}$.
• Fast sum and fast updates. Maintain balanced binary tree $T$ on $A$. Each node stores the sum of elements in subtree.
Partial Sums

- **Sum**.
  - **Sum(i)**: traverse path to i + 1 and sum up all off-path nodes.
- **Time.** \(O(\log n)\)
Partial Sums

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

- **UPDATE.**
  - **UPDATE**(i, Δ): add Δ to nodes on path to i.
- **Time.** O(log n)
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>$\Omega(\log n)$</td>
<td>$\Omega(\log n)$</td>
<td></td>
</tr>
</tbody>
</table>

- **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 $D$. 
Fenwick tree. Replace A by another array F.

- Recurse on the entries $A[2, 4, \ldots, n]$ until we are left with a single element.
• **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.
**Partial Sums**

- **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 rmb($i_j$) is the integer corresponding to the rightmost 1-bit in $i$. Stop when we get 0.
  - Time. $O(\log n)$

- **Sum(14)?**

  - $14 = 1110_2$
  - $12 = 1100_2$
  - $8 = 1000_2$
  - $0 = 0000_2$
Partial Sums

- Partial Sums

- \[ \text{UPDATE.} \]
  - \[ \text{UPDATE}(i, \Delta): \text{add} \ \Delta \ \text{to partial sums covering} \ i. \]
  - Indexes \( i_0, i_1, \ldots \) 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

<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>$\Omega(\log n)$</td>
<td>$\Omega(\log n)$</td>
<td></td>
</tr>
<tr>
<td>Fenwick tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>in-place</td>
</tr>
</tbody>
</table>
Data Structures II

- Partial Sums
- Dynamic Arrays
Dynamic Arrays

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

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

```plaintext
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
```
• **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></td>
</tr>
</tbody>
</table>

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

- 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.
Dynamic 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).
Dynamic Arrays

• ACCESS.
  • ACCESS\( (i) \): compute rotated array \( R_j \) and index \( k \) corresponding to \( i \). Return \( R_j[k] \).
• Time. \( O(1) \)
Dynamic 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.** \( O(\sqrt{n}) \)
Dynamic Arrays

DELETE(5)

• **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

<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>Ω(( \frac{\log n}{\log \log n} ))</td>
<td>Ω(( \frac{\log n}{\log \log n} ))</td>
<td>Ω(( \frac{\log n}{\log \log n} ))</td>
<td></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^\varepsilon ))</td>
<td>O(( n^\varepsilon ))</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Data Structures II

• Partial Sums
• Dynamic Arrays