Can you use a hashtable to implement skipTo()?

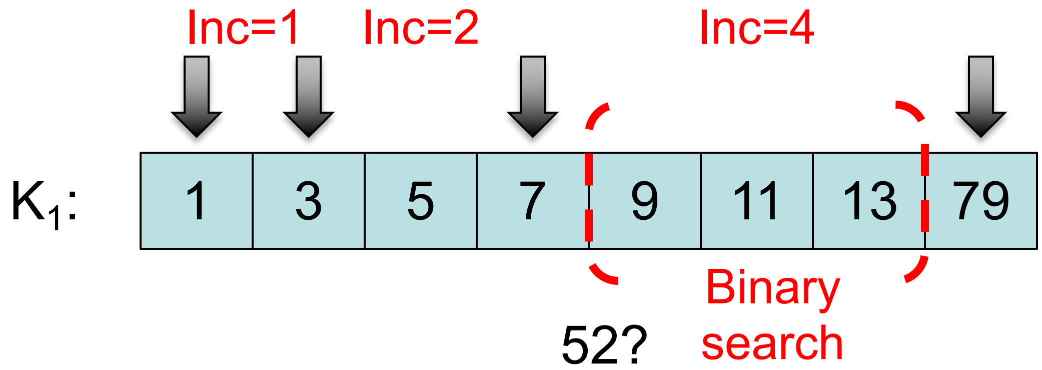
Better than next()

- What's the worst case for sequential merge-based intersection?
- {52, 1} ➔ move k_2's cursor
  - To the position whose id is at least 52 ➔ **skipTo**(52)
  - Essentially, asking the first i, such that $K_2[i] >= 52$ (K2's list is sorted).
  - Takes many sequential call of next()
  - Could use binary search in the rest of the list
  - Cost: $\lceil \log_2(N_{remainder}) \rceil$

K_2:	1	3	5	…	…	…	…	79
K_1:	52	54	56	58				

skipTo(id)

- Galloping search (gambler's strategy)
  - [Stage 1] Doubling the search range until you overshoot
  - [Stage 2] Perform binary search in the last range
- Performance analysis (worst case)
  - Let the destination position be n positions away.
  - $\approx \log_2 n$ probes in Stage 1 + $\approx \log_2 n$ probes in Stage 2
  - Total = $2\lceil \log_2 (n+1) \rceil = O(\log_2 n)$



Total Cost

- Galloping search (gambler's strategy)
  - Cost of the i-th probe: $\approx 2\log_2(n_i)$
  - Total cost of K_1 probes: $\approx 2\log_2(\prod_1^{|K_1|} n_i)$

  $\leq 2\log_2\left(\left(\left(\sum_1^{|K_2|} n_i\right) / |K_1| \right)^{|K_1|} \right) \leq 2|K_1| * \log_2(|K_2|/|K_1|)$
- Asymptotically, resembles linear merge when $|K_2|/|K_1| = O(1)$, resembles binary search when $|K_1| = O(1)$

What about list intersection using binary search?

Multiple Term Conjunctive Queries

- K_1 AND K_2 AND … AND K_n
- SvS does not perform well if none of the associated lists are short
- In addition, it is blocking
- Can you design non-blocking multiple sorted array intersection algorithm?

Generalization

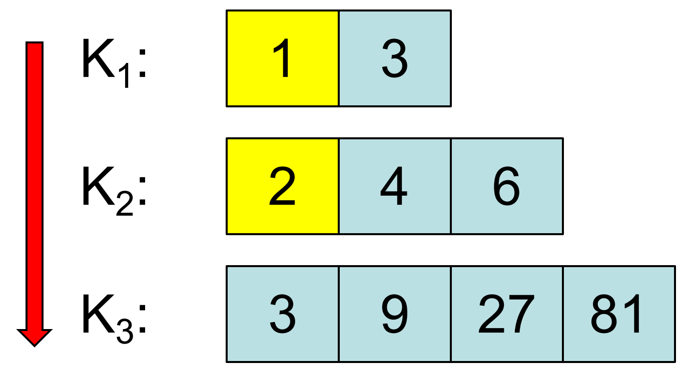
- Generalize the 2-way intersection algorithm
- 2-way:
  - {1, 2} ➔ move k_1's cursor
  - skipTo(2)
- 3-way:
  - {1, 2, 3} ➔ move k_1,k_2's cursor
  - skipTo(3)

K_1:	1	3		
K_2:	2	4	6	
K_3:	3	9	27	81

eliminator = $Max_{1<=i<=n}(k_i.cursor)$

Optimization

- Mismatch found even before accessing K_3's cursor
- Choice 1: continue to get cursors of other list
- Choice 2: settle the dispute within the first two lists ➔ max algorithm [Culpepper & Moffat, 2010]
  - Better locality of access ➔ fewer cache misses
  - Similar to SvS



Pseudo-Code for the Max Algorithm (Wrong)

- Input: $K_1, K_2, \ldots, K_n$ in increasing size

$x := K_1[1]$; *startAt* := 2 `//x is the eliminator`

while x is defined **do**

    for i = *startAt* **to** n **do**

        $y := K_i$.skipTo(x)

        if $y > x$ **then** `//mismatch`

            $x := K_1$.next() `//restart_1` `//restart_2`

            if $y > x$ **then** *startAt* := 1; $x := y$ **else** *startAt* := 2 **end if**

            break `//match in all  lists`

        elsif $i = n$ **then** `//y = x`

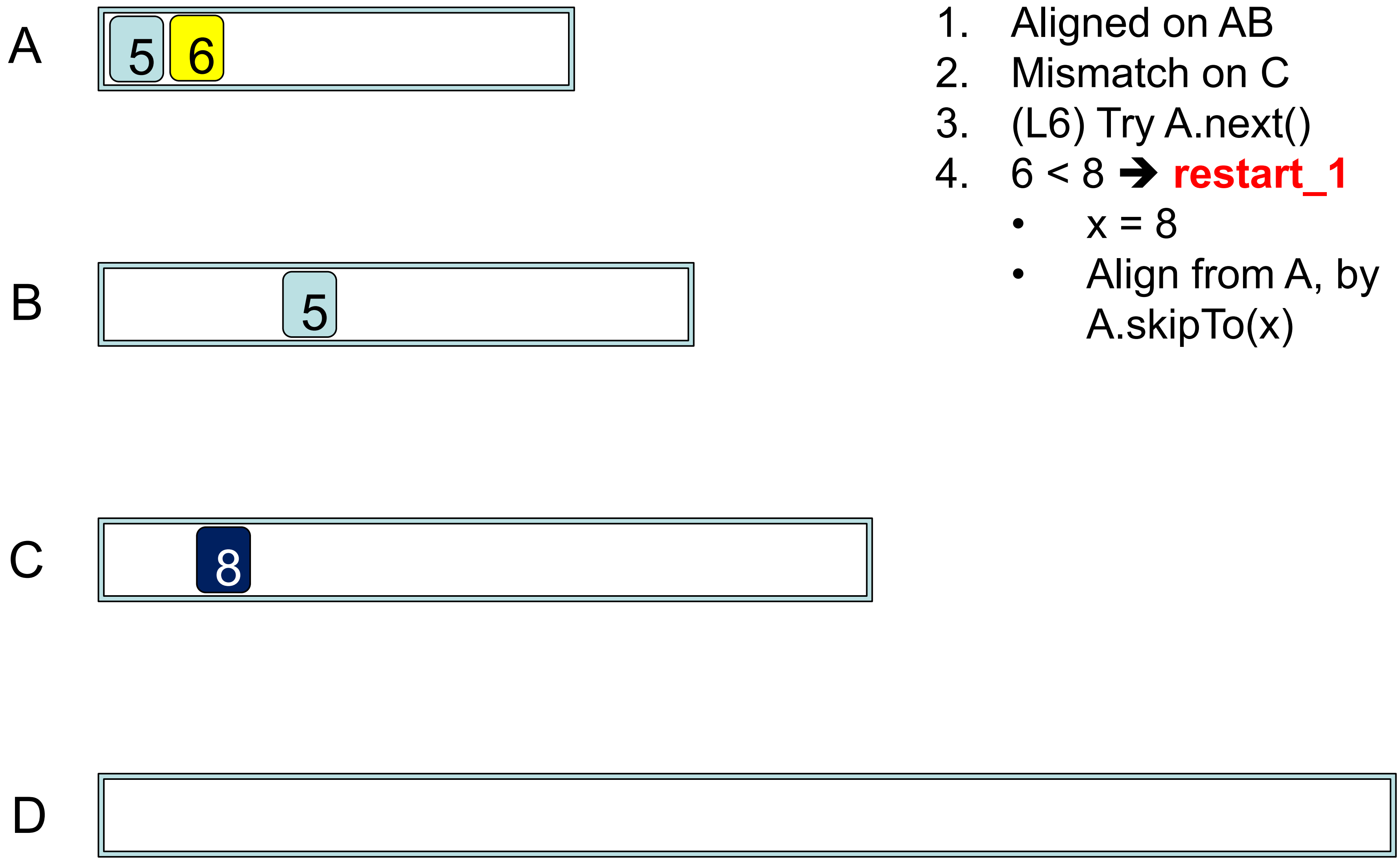
            Output x

            $x := K_1$.next()

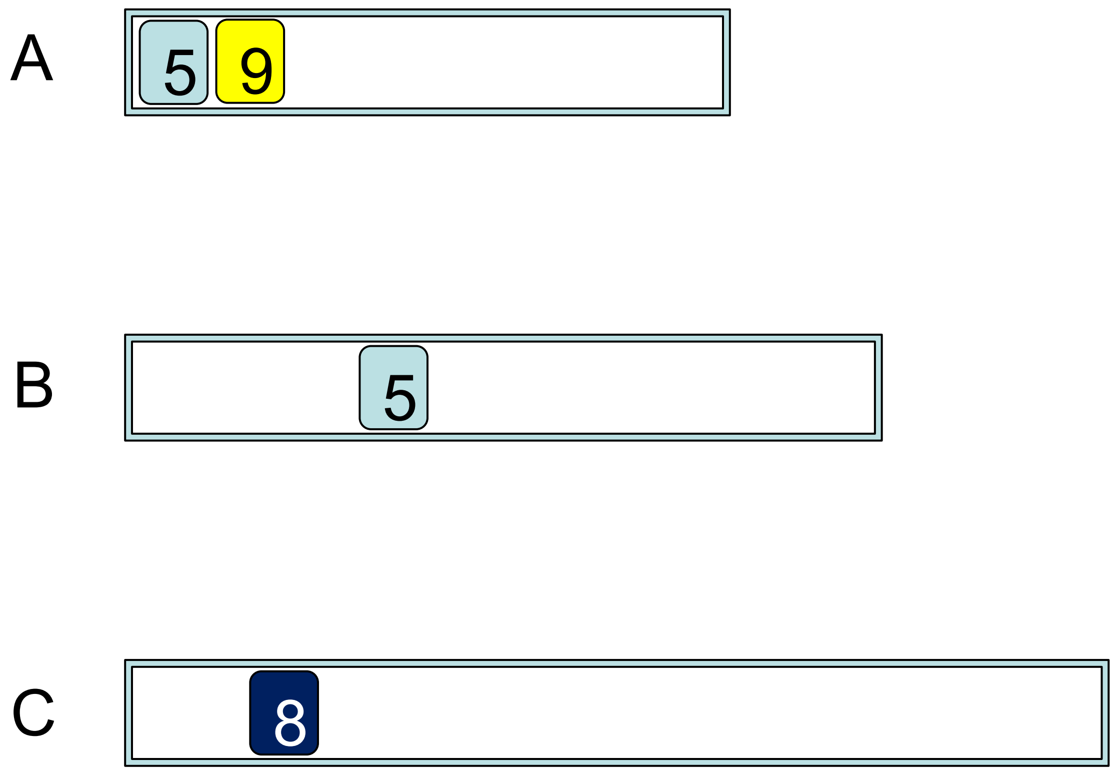
        end if

    end for

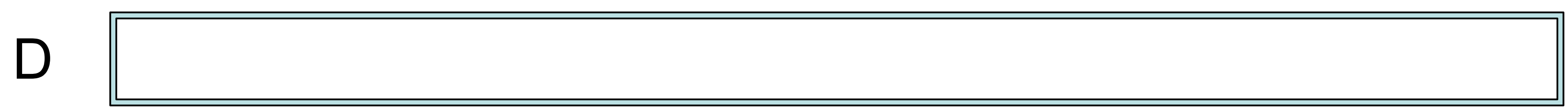
end while



1. Aligned on AB
2. Mismatch on C
3. (L6) Try A.next()
4. 6 < 8 ➔ **restart_1**
   - x = 8
   - Align from A, by A.skipTo(x)



1. Aligned on AB
2. Mismatch on C
3. (L6) Try A.next()
4. 9 < 8 ➔ **restart_2**
   - x = 9
   - Align from B, by B.skipTo(x)



The original code has a bug when in restart_1 cases

Pseudo-Code for the Max Algorithm (Fixed)

- Input: $K_1, K_2, \ldots, K_n$ in increasing size

```
(1)  x := K1[1]; startAt := 2
(2)  while x is defined do
(3)      for i = startAt to n do
(4)          y := Ki.skipTo(x)
(5)          if y > x then
(6)              x := K1.next()
(7)              if y > x then startAt := 1; x := y else startAt := 2 end if
(8)              break
(9)          elsif i = n then
(10)             Output x
(11)             x := K1.next()
(12)         end if
(13)     end for
(14) end while
```

```
(4.1) if i = 1 then
(4.2)     if y > x then
(4.3)         x := y
(4.4)         break
(4.5)     end if
(4.6) end if
```

References

- J. Shane Culpepper, Alistair Moffat: Efficient set intersection for inverted indexing. ACM Trans. Inf. Syst. 29(1): 1 (2010)
- F.K. Hwang and S. Lin, A simple algorithm for merging two disjoint linearly ordered sets. SIAM J. Comput. 1 1 (1972), pp. 31–39.
- Stefan Buettcher, Charles L. A. Clarke, Gordon V. Cormack, Information Retrieval: Implementing and Evaluating Search Engines, 2010 [Chapter 5]