Indexing

Information Retrieval
CMP-5036A
Sarah Taylor
s.l.taylor@uea.ac.uk

Overview

- Inverted indexes
- Index construction
- Index sorting
- Indexing in Python

Why do we need to Index Documents?

To Retrieve all Documents containing a Term…

- The naïve approach is to scan the entire collection
    - Typical in early (batch) retrieval systems
    - Computational and I/O costs are high: O(terms in all documents)
    - Fine for modest collections, but impractical for most

- Or, we can directly access information by indexing the collection in advance
    - Search evaluation time much lower: O(query term occurrences in collection)
    - Practical for larger collections
    - Many opportunities for optimisation

Think of the Index of a Book

Without it, you'd need to look through every page of the book until you find the information that you need

With it, you can directly access the pages that contain information related to your need

Term-Document Matrices

Term\Doc#	1	2	3	4	5
the	1	1	1	1	1
cow	1	0	0	1	0
says	0	0	1	0	1
moo	1	1	0	1	0

Term-Document Matrices

Term\Doc#	1	2	3	4	5
the					1
cow					0
says	0	0	1	0	1
moo	1	1	0	1	0

How does this
scale for large collections?

How does it Scale?

- Reminder:
    - vocabulary grows proportionate to the square root of the collection size (Heaps' Law)
    - approximately 50% of terms are singletons (Zipf's Law)

How does it Scale?

- Reminder:
  - vocabulary grows proportionate to the square root of the collection size (Heaps' Law)
  - approximately 50% of terms are singletons (Zipf's Law)

Term\Doc#	1	2	3	…	498	499	500
the	1	1	1	…	1	1	1
cow	1	0	0	…	1	0	0
says	0	0	1	…	1	0	0
moo	1	1	0	…	0	0	1
…	…	…	…	…	…	…	…
<singleton1>	0	0	0	…	1	0	0
<singleton2>	0	0	0	…	0	1	0
<singleton3>	0	0	0	…	0	0	1

Singletons have zeros in every column but one

How does it Scale?

- Reminder:
    - vocabulary grows proportionate to the square root of the collection size (Heaps’ Law)
    - approximately 50% of terms are singletons (Zipf’s Law)
- Matrix size is vocab x #documents

How does it Scale?

- Reminder:
    - vocabulary grows proportionate to the square root of the collection size (Heaps' Law)
    - approximately 50% of terms are singletons (Zipf's Law)
- Matrix size is vocab x #documents
- e.g. TREC 1GB corpus has approximately:
    - 336,000 documents containing 125 million words
    - 500,000 vocabulary

How does it Scale?

- Reminder:
  - vocabulary grows proportionate to the square root of the collection size (Heaps’ Law)
  - approximately 50% of terms are singletons (Zipf’s Law)
- Matrix size is vocab x #documents
- e.g. TREC 1GB corpus has approximately:
  - 336,000 documents containing 125 million words
  - 500,000 vocabulary
- Matrix will be very **sparse** and very **big**

Term-Document Matrices

Term\Doc#	1	2	3	4	5
the					1
cow					0
says					1
moo					0

How does this scale for large collections?

Poorly - this is an inefficient way to index a collection for IR

Inverted Indexes

~~Term\Doc#~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~
~~the~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
~~cow~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~
~~says~~	~~0~~	~~0~~	~~1~~	~~0~~	~~1~~
~~moo~~	~~1~~	~~1~~	~~0~~	~~1~~	~~0~~

Term	Postings
the	1, 2, 3, 4, 5
cow	1, 4
says	3, 5
moo	1, 2, 4

Inverted Indexes

~~Term\Doc#~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~
~~the~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
~~cow~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~
~~says~~	~~0~~	~~0~~	~~1~~	~~0~~	~~1~~
~~moo~~	~~1~~	~~1~~	~~0~~	~~1~~	~~0~~

Term	Postings
the	1, 2, 3, 4, 5
cow	1, 4
says	3, 5
moo	1, 2, 4

More efficient both in terms of memory and search speed

What Should an Index Contain?

- Database management systems have indexes on primary keys and usually other fields
    - Index provides fast access to database records
    - The more fields are indexed, the greater the memory overhead. (Trade off between memory and speed of the search)

- Can we use prior knowledge of the user’s information need to limit the index terms?
    - Usually not, and every word in a collection of documents is a potential search term

Index Contents

- How do we represent the presence/absence of a term in a document?
    - Boolean (1 or 0/present or absent)
    - Statistical (term frequency, ...)
    - Positional information (where does the term appear?)
    - (More later…)

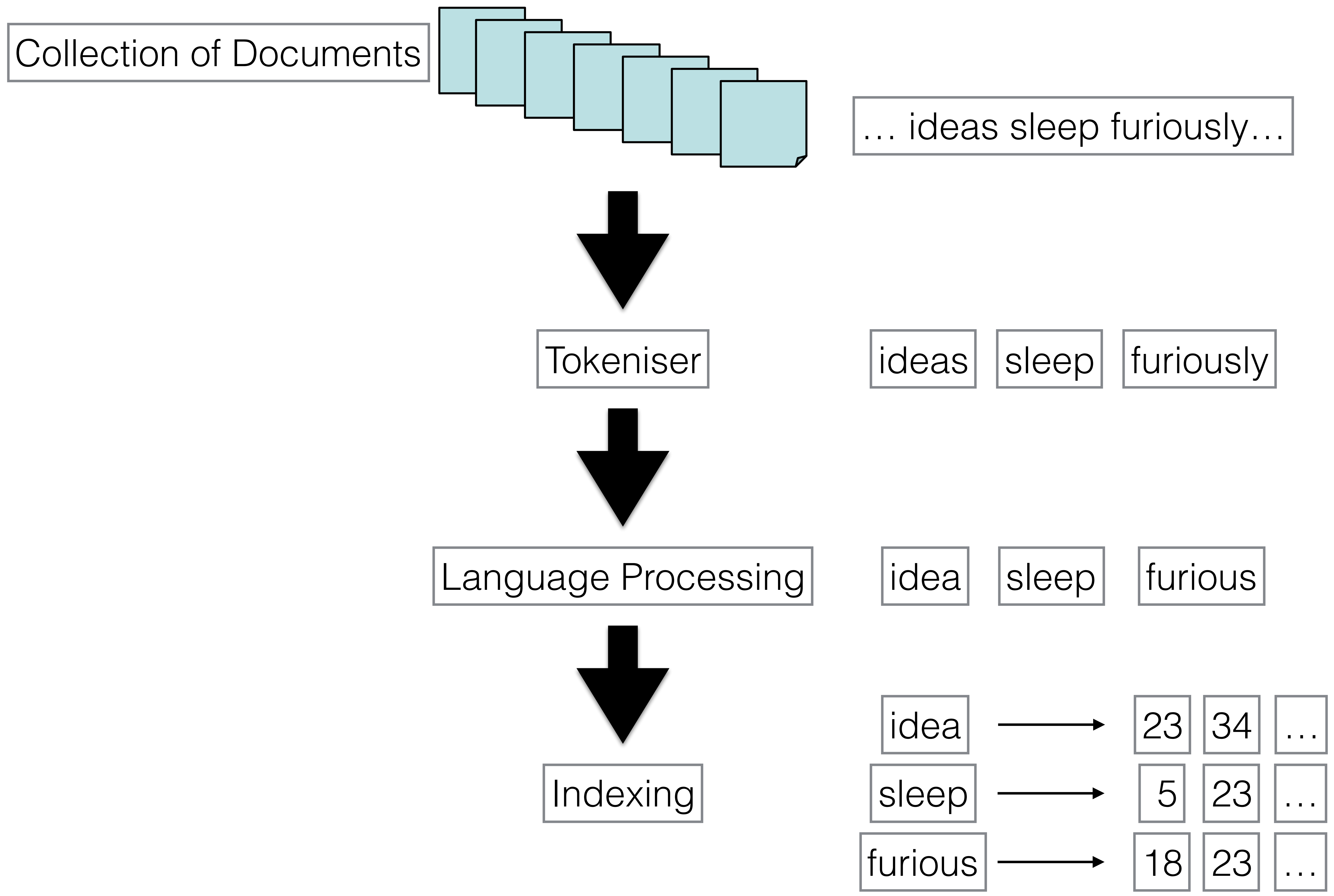
Terminology

Term	Postings
the	1, 2, 3, 4, 5
cow	1, 4
says	3, 5
moo	1, 2, 4

- Posting: *A record in the index containing the documents containing a term (and/or frequency and positional information)*

Constructing an Index

Index Construction



Example

Collection of Documents

Doc.	Text
1	Pease pudding hot, pease pudding cold
2	Pease pudding in the pot
3	Nine days old
4	Some like it hot
5	Some like it cold
6	Pease pudding in the pot
7	Nine days old

Example

Inverted index with no post-processing

Doc.	Text
1	Pease pudding hot, pease pudding cold
2	Pease pudding in the pot
3	Nine days old
4	Some like it hot
5	Some like it cold
6	Pease pudding in the pot
7	Nine days old

Term ID	Term	Postings
1	Pease	1,2,6
2	pease	1
3	pudding	1,2,6
4	hot	1,4
5	cold	1,5
6	in	2,6
7	the	2,6
8	pot	2,6
9	Nine	3,7
10	days	3,7
11	old	3,7
12	Some	4,5
13	like	4,5
14	it	4,5

Example

Inverted index with *some* post-processing

Doc.	Text
1	Pease pudding hot, pease pudding cold
2	Pease pudding in the pot
3	Nine days old
4	Some like it hot
5	Some like it cold
6	Pease pudding in the pot
7	Nine days old

$\Rightarrow$

Term ID	Term	Postings
1	pease	1,2,6
2	pudding	1,2,6
3	hot	1,4
4	cold	1,5
5	pot	2,6
6	nine	3,7
7	days	3,7
8	old	3,7
9	some	4,5
10	like	4,5

1) Case normalisation
2) Stopword removal

Index with Frequencies

Doc.	Text
1	Pease pudding hot, pease pudding cold
2	Pease pudding in the pot
3	Nine days old
4	Some like it hot
5	Some like it cold
6	Pease pudding in the pot
7	Nine days old

Term ID	Term	Postings
1	pease	(1:2),(2:1),(6:1)
2	pudding	(1:2),(2:1),(6:1)
3	hot	(1:1),(4:1)
4	cold	(1:1),(5:1)
5	pot	(2:1),(6:1)
6	nine	(3:1),(7:1)
7	days	(3:1),(7:1)
8	old	(3:1),(7:1)
9	some	(4:1),(5:1)
10	like	(4:1),(5:1)

(docID:termFreq)

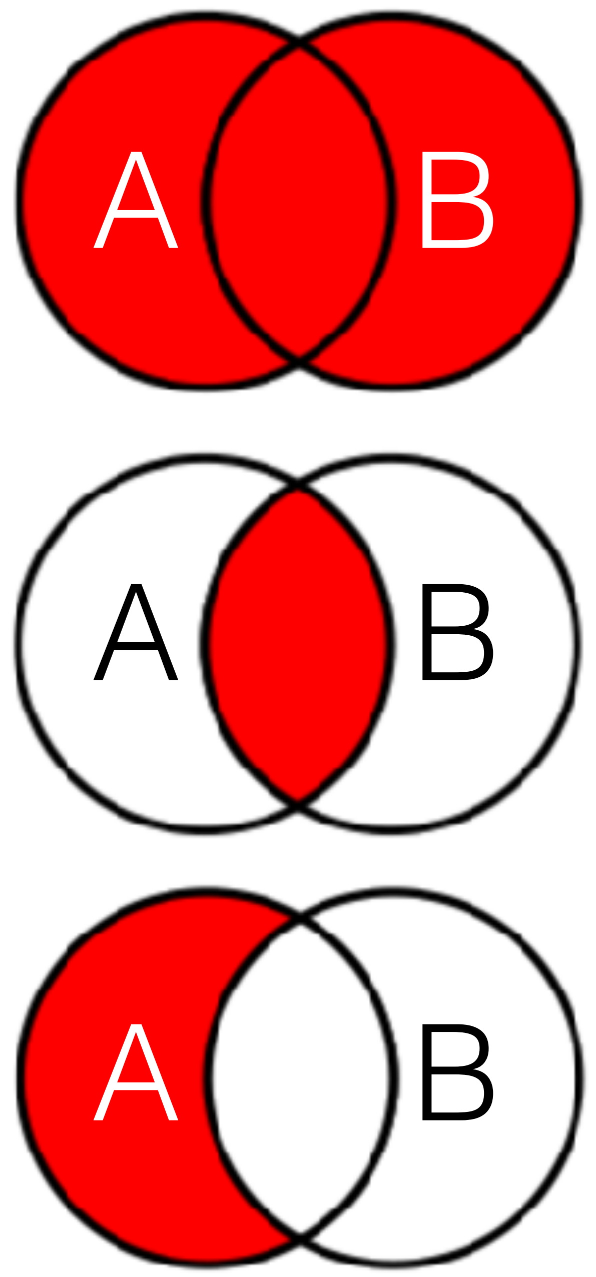
Searching an Index

Boolean Queries - Exact Matching

- Boolean queries are queries using AND, OR and NOT to join query terms
    - x AND y - both x and y must match in the result.
    - x OR y - either x or y must match in the result.
    - NOT x - x must not match in the result.
    - Can form complex Boolean expressions - x AND (y OR z)
- Views each document as a set of terms
- Exact matching - document either matches condition(s) or not
- Professional searchers (e.g., lawyers) like Boolean queries:
    - You know exactly what you’re getting
    - Easy to refine query
    - Needs expertise to do well

Boolean Searching

1. Split the query into separate words
2. Retrieve occurrence list for each word from postings table
3. Merge occurrence lists appropriately:
   - OR the union of lists
   - AND the intersection of lists
   - NOT the difference between lists
4. Retrieve documents



Query Processing: AND

Consider processing the query
Hot AND *pease*

Query Processing: AND

Consider processing the query
Hot AND *pease*

1. Locate *Hot* in the Dictionary
    - Retrieve its postings.

$$\begin{tikzcd} \boxed{2} \arrow[r] & \boxed{4} \arrow[r] & \boxed{8} \arrow[r] & \boxed{16} \arrow[r] & \boxed{32} \arrow[r] & \boxed{64} \arrow[r] & \boxed{128} \end{tikzcd}$$

Query Processing: AND

Consider processing the query
Hot AND *pease*

1. Locate *Hot* in the Dictionary
    - Retrieve its postings.

$$\begin{tikzcd} \boxed{2} \arrow[r] & \boxed{4} \arrow[r] & \boxed{8} \arrow[r] & \boxed{16} \arrow[r] & \boxed{32} \arrow[r] & \boxed{64} \arrow[r] & \boxed{128} \end{tikzcd}$$

2. Locate *pease* in the Dictionary
    - Retrieve its postings.

$$\begin{tikzcd} \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{5} \arrow[r] & \boxed{8} \arrow[r] & \boxed{13} \arrow[r] & \boxed{21} \arrow[r] & \boxed{34} \end{tikzcd}$$

Query Processing: AND

Consider processing the query
Hot AND *pease*

1. Locate *Hot* in the Dictionary
   - Retrieve its postings.

$$\begin{tikzcd}\boxed{2} \arrow[r] & \boxed{4} \arrow[r] & \boxed{8} \arrow[r] & \boxed{16} \arrow[r] & \boxed{32} \arrow[r] & \boxed{64} \arrow[r] & \boxed{128}\end{tikzcd}$$

2. Locate *pease* in the Dictionary
   - Retrieve its postings.

$$\begin{tikzcd}\boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{5} \arrow[r] & \boxed{8} \arrow[r] & \boxed{13} \arrow[r] & \boxed{21} \arrow[r] & \boxed{34}\end{tikzcd}$$

3. Merge the two postings:

$$\begin{tikzcd}\boxed{2} \arrow[r] & \boxed{8}\end{tikzcd}$$

Merging Algorithm

Posting List 1:

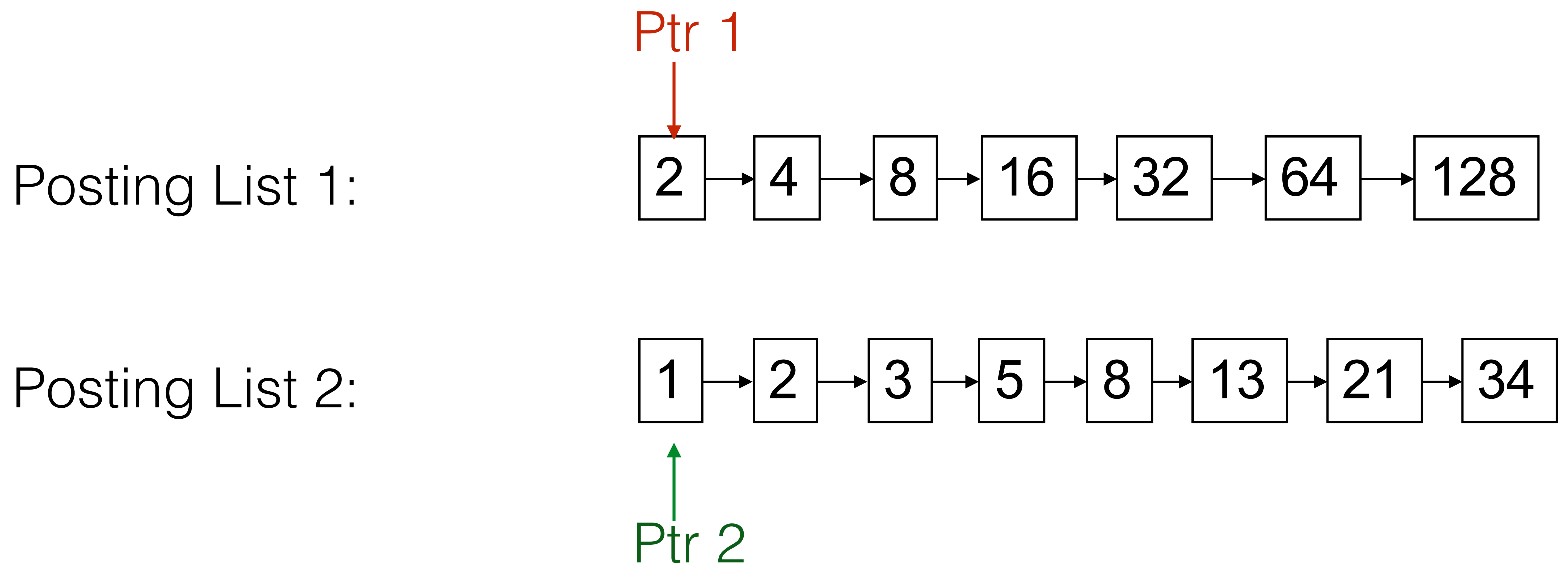
$$\begin{tikzcd}\boxed{2} \arrow[r] & \boxed{4} \arrow[r] & \boxed{8} \arrow[r] & \boxed{16} \arrow[r] & \boxed{32} \arrow[r] & \boxed{64} \arrow[r] & \boxed{128}\end{tikzcd}$$

Posting List 2:

$$\begin{tikzcd}\boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{5} \arrow[r] & \boxed{8} \arrow[r] & \boxed{13} \arrow[r] & \boxed{21} \arrow[r] & \boxed{34}\end{tikzcd}$$

Merged List:

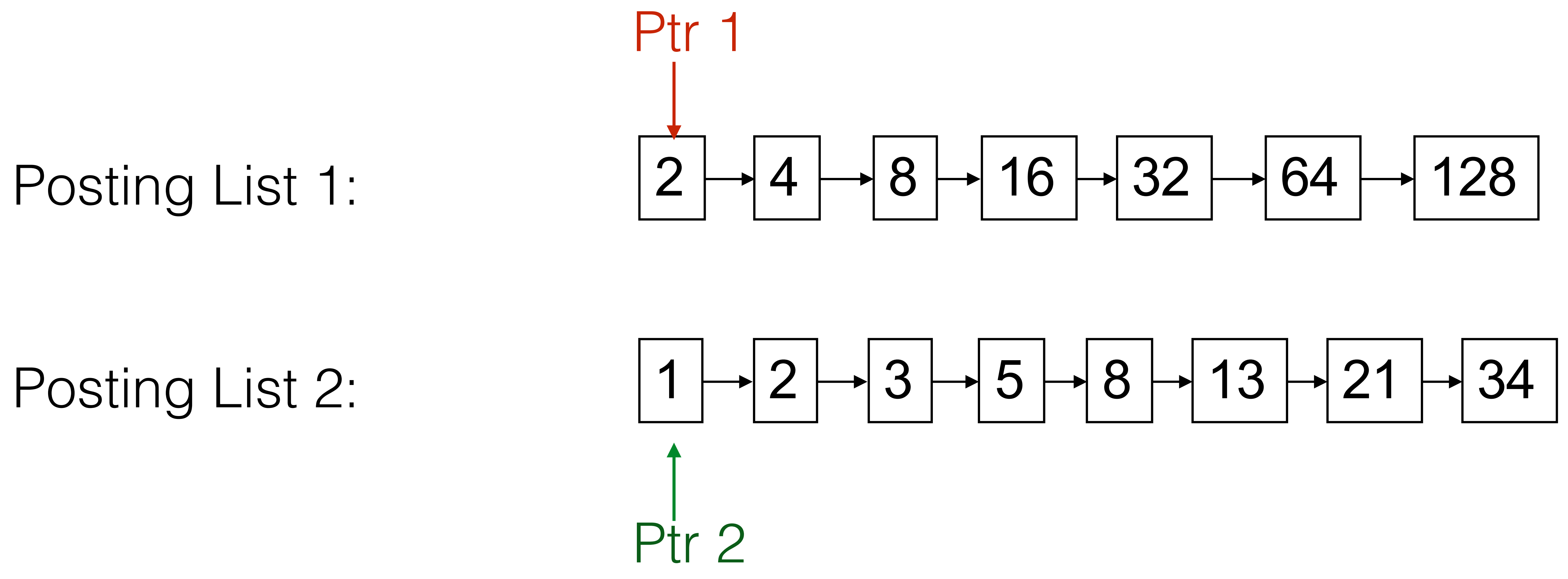
Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

Merged List:

Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

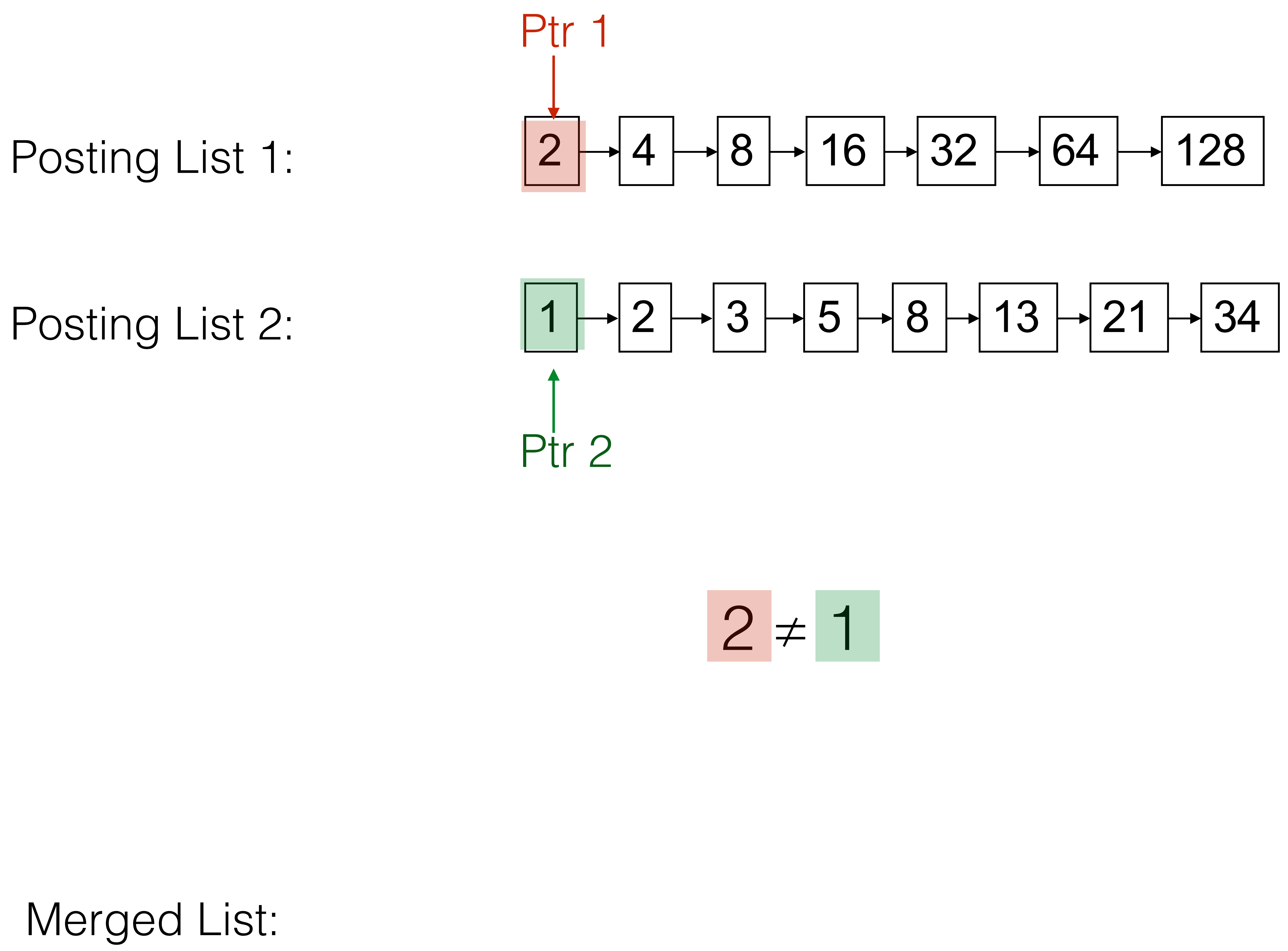
1. Do the doc ID's match?

    A. No - Advance the pointer pointing to the smaller doc ID

    B. Yes - Add doc ID to merged list and advance both pointers

Merged List:

Merging Algorithm



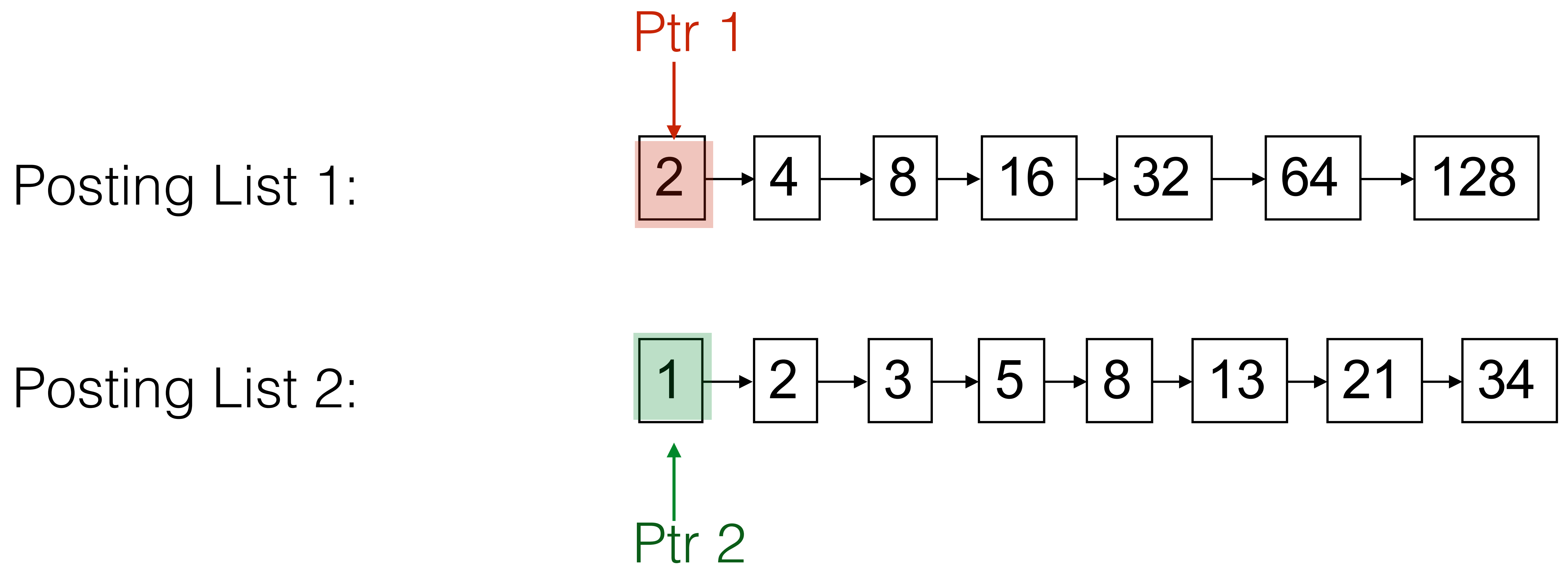
Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?

   A. No - Advance the pointer pointing to the smaller doc ID

   B. Yes - Add doc ID to merged list and advance both pointers

Merging Algorithm



$2 \neq 1$

Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?

   A. No - Advance the pointer pointing to the smaller doc ID

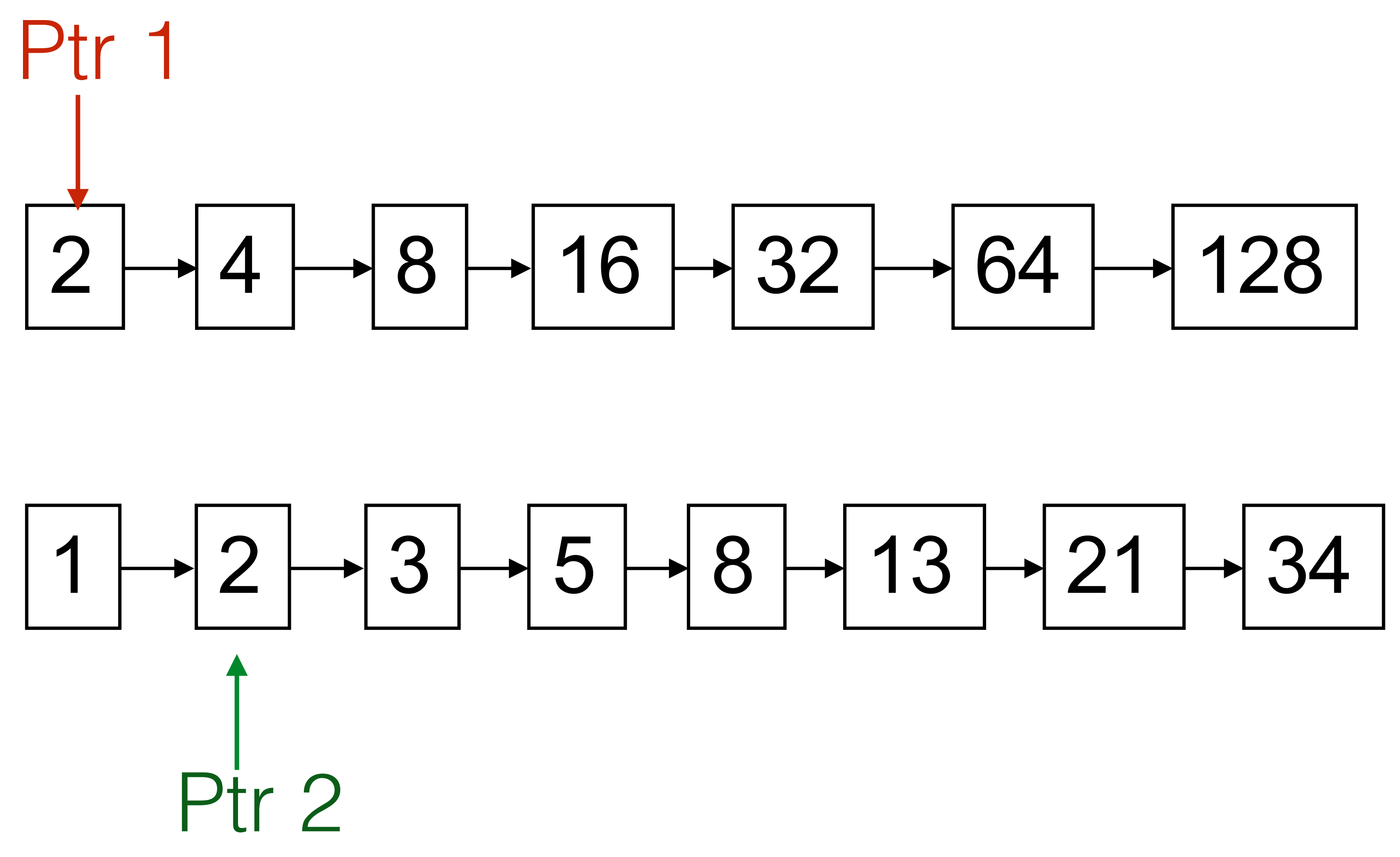
   B. Yes - Add doc ID to merged list and advance both pointers

Merged List:

Merging Algorithm

Posting List 1:

Posting List 2:



Maintain two pointers that are initialised on first posting from each list respectively

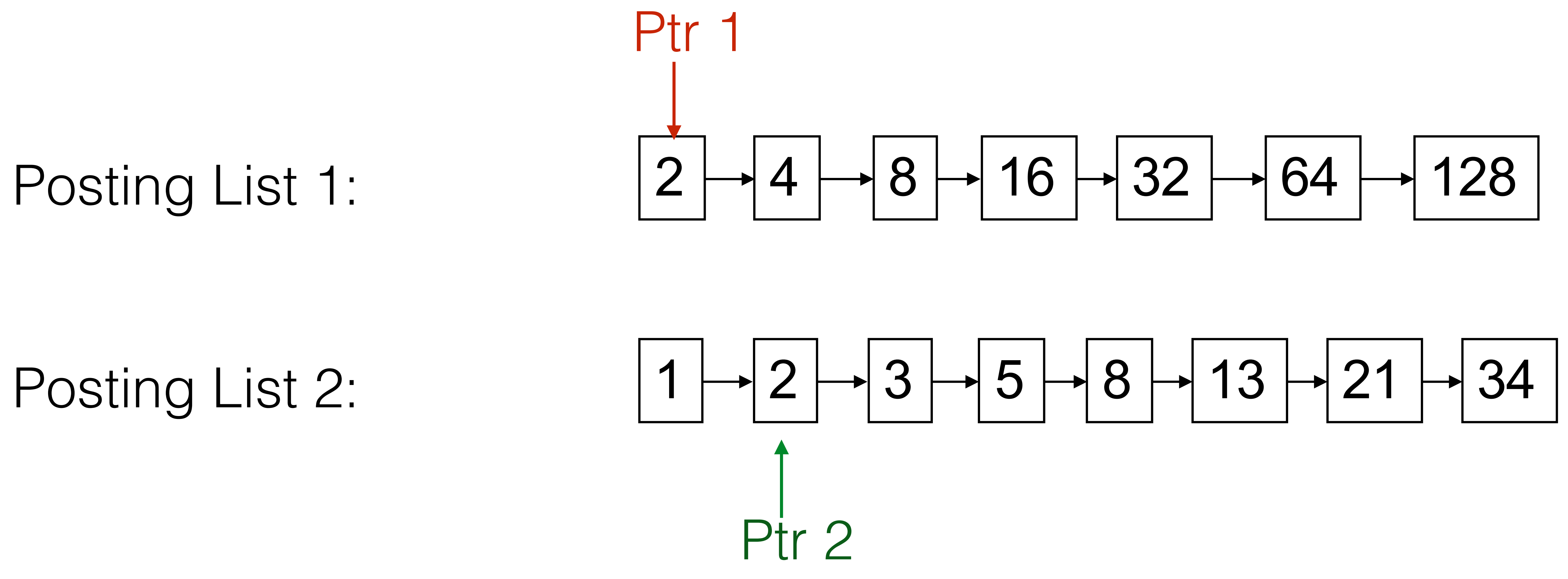
1. Do the doc ID's match?

   A. No - Advance the pointer pointing to the smaller doc ID

   B. Yes - Add doc ID to merged list and advance both pointers

Merged List:

Merging Algorithm

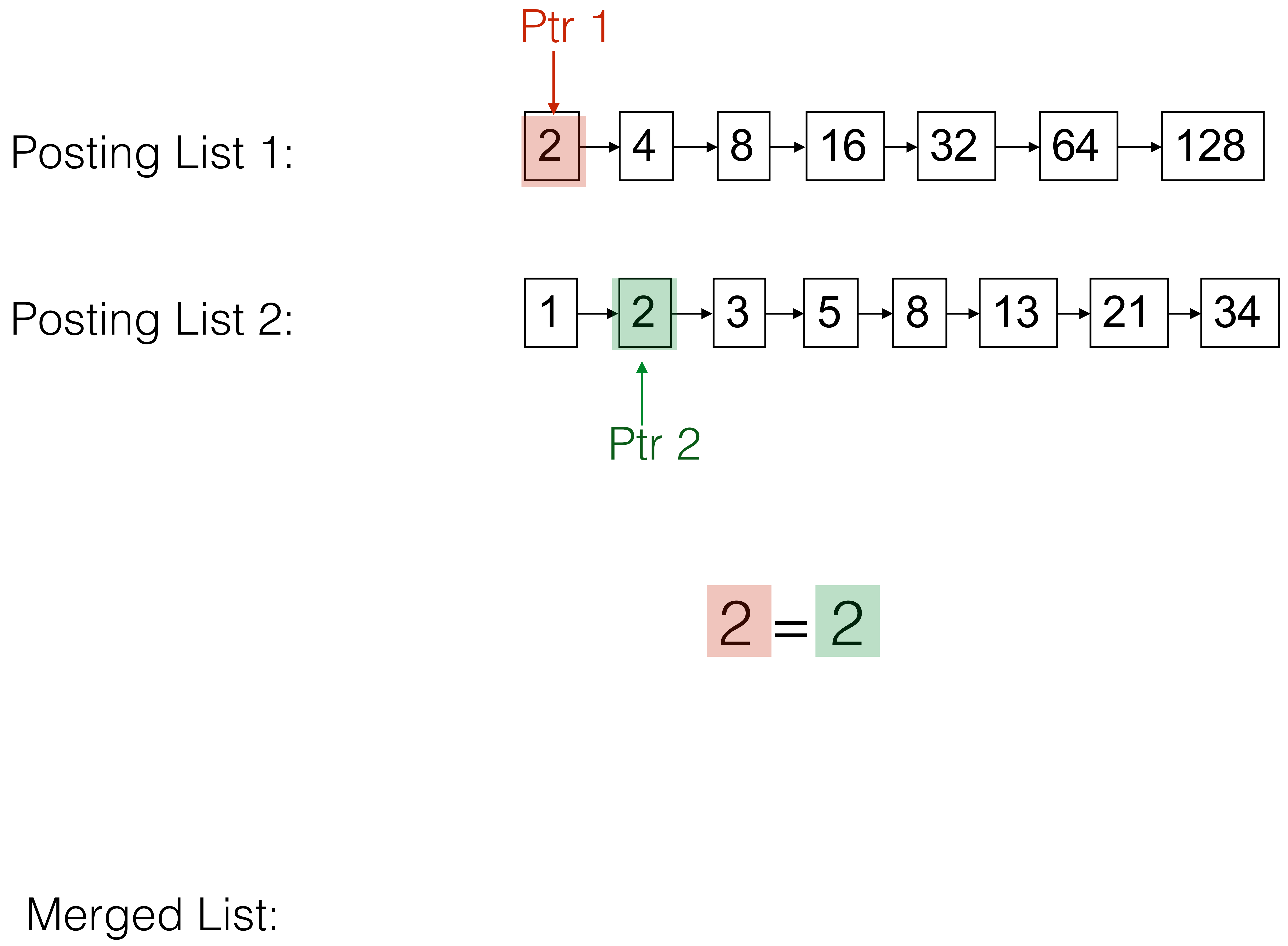


Merged List:

Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?
   A. No - Advance the pointer pointing to the smaller doc ID
   B. Yes - Add doc ID to merged list and advance both pointers
2. Repeat until end

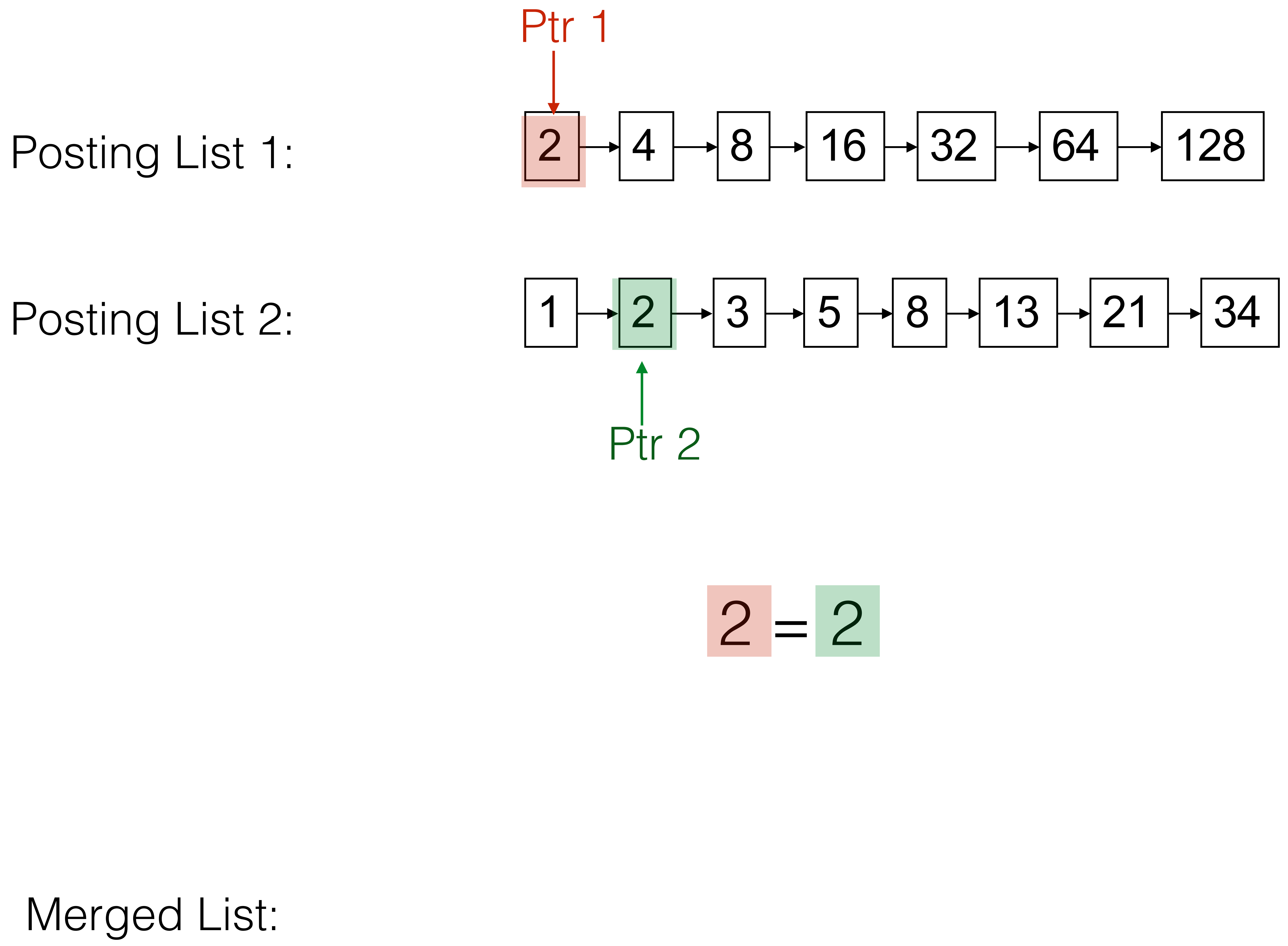
Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?
   A. No - Advance the pointer pointing to the smaller doc ID
   B. Yes - Add doc ID to merged list and advance both pointers
2. Repeat until end

Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

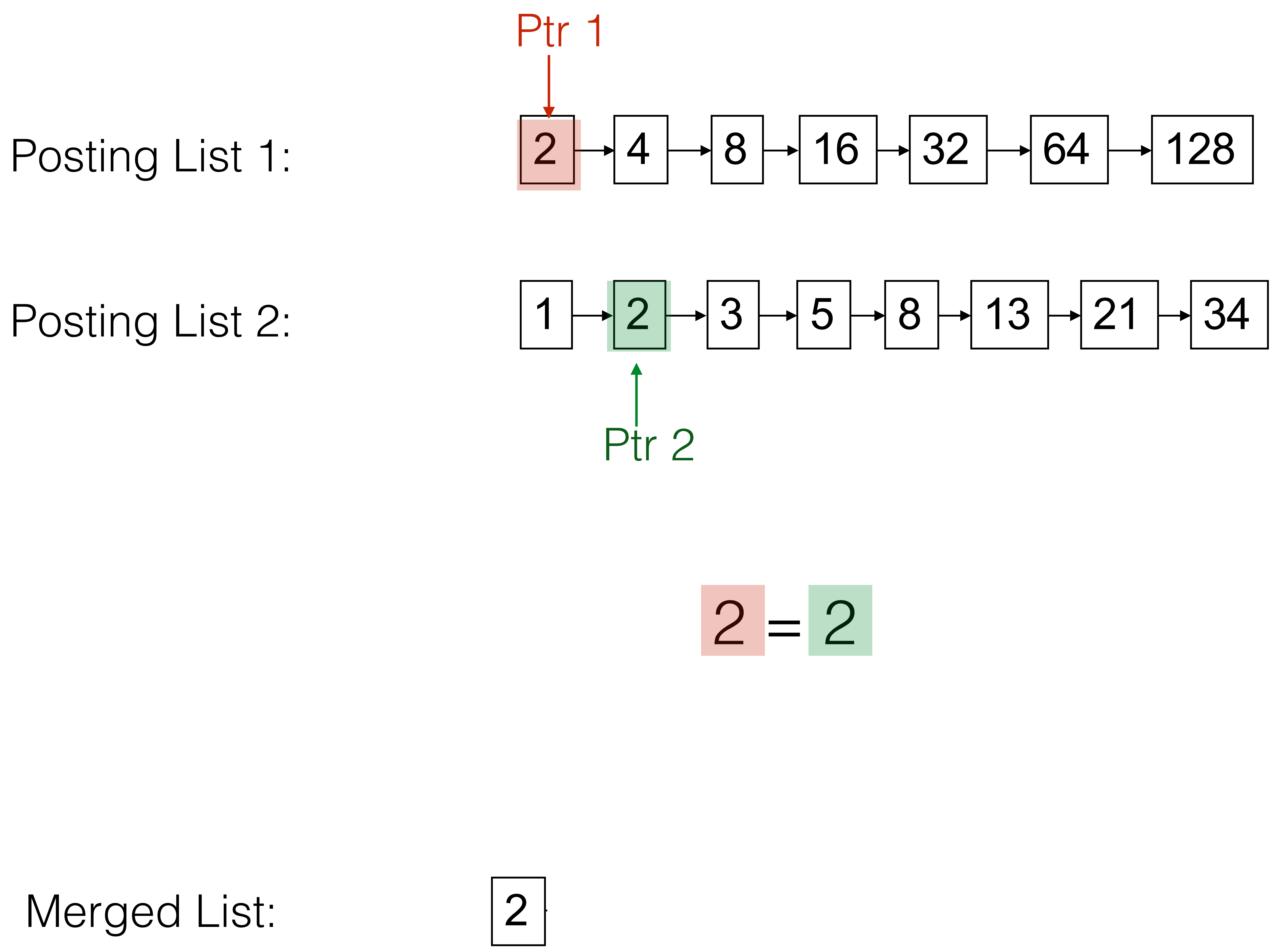
1. Do the doc ID's match?

    A. No - Advance the pointer pointing to the smaller doc ID

    B. Yes - Add doc ID to merged list and advance both pointers

2. Repeat until end

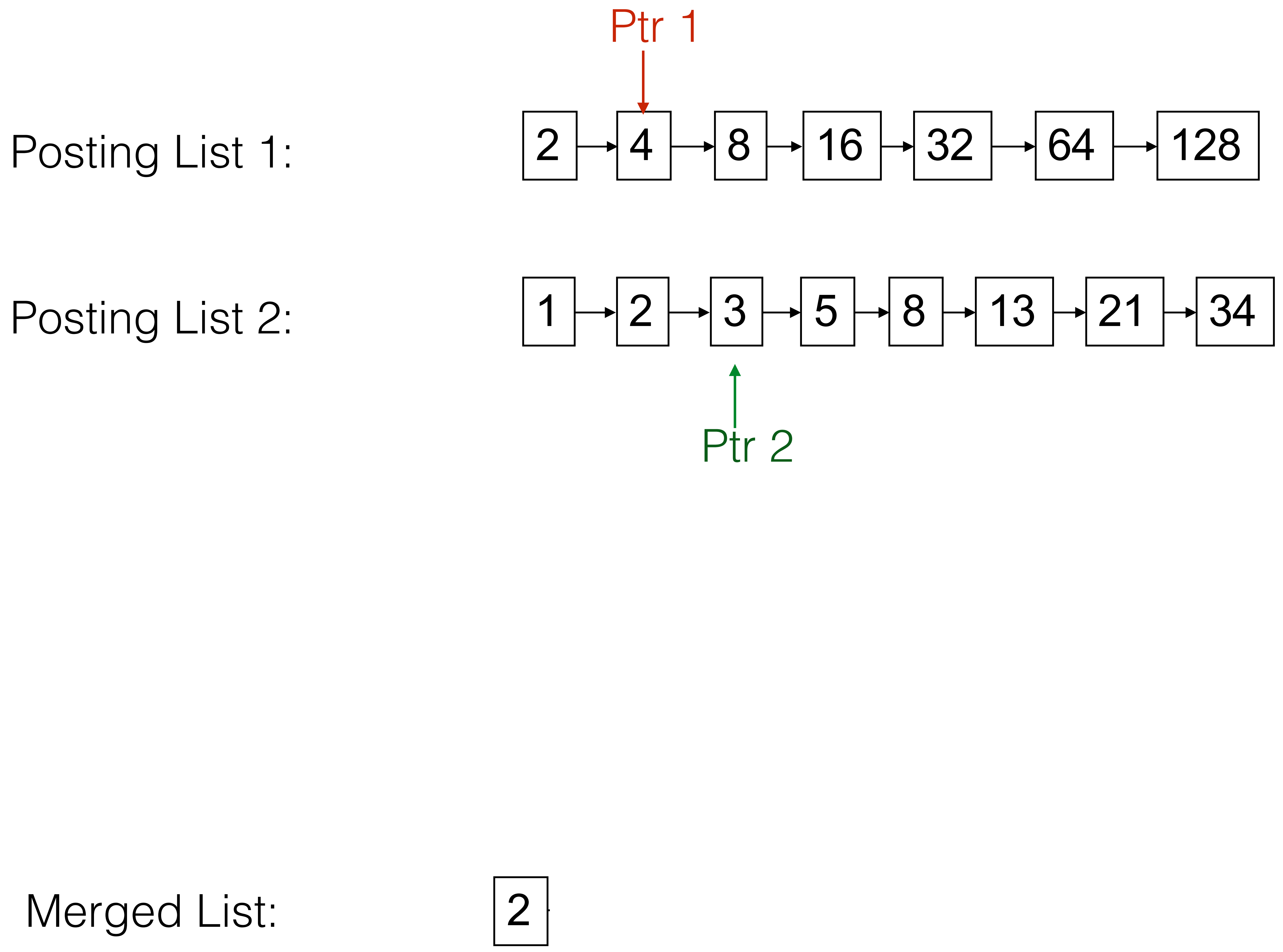
Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?
   A. No - Advance the pointer pointing to the smaller doc ID
   B. Yes - Add doc ID to merged list and advance both pointers
2. Repeat until end

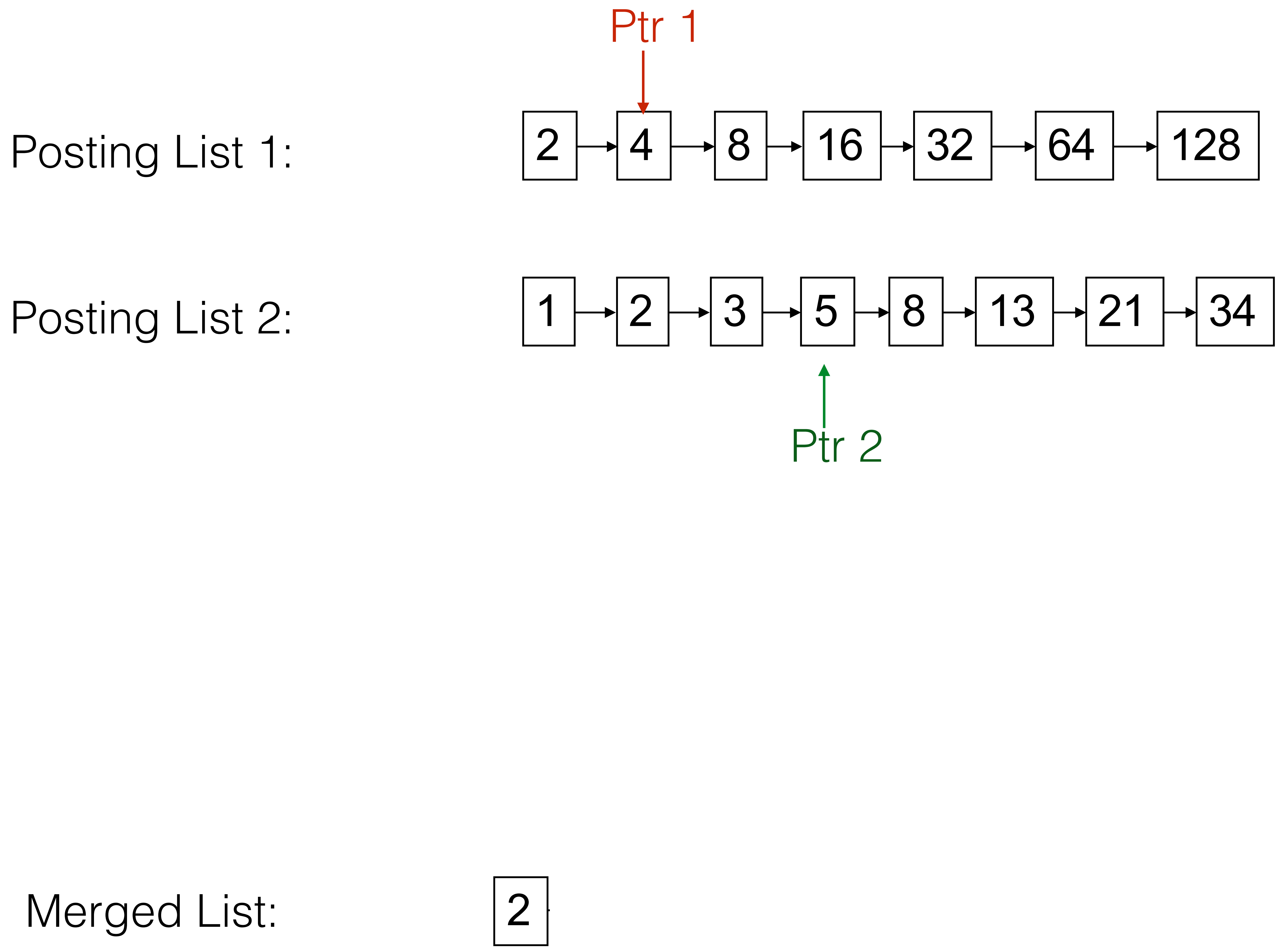
Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?
   A. No - Advance the pointer pointing to the smaller doc ID
   B. Yes - Add doc ID to merged list and advance both pointers
2. Repeat until end

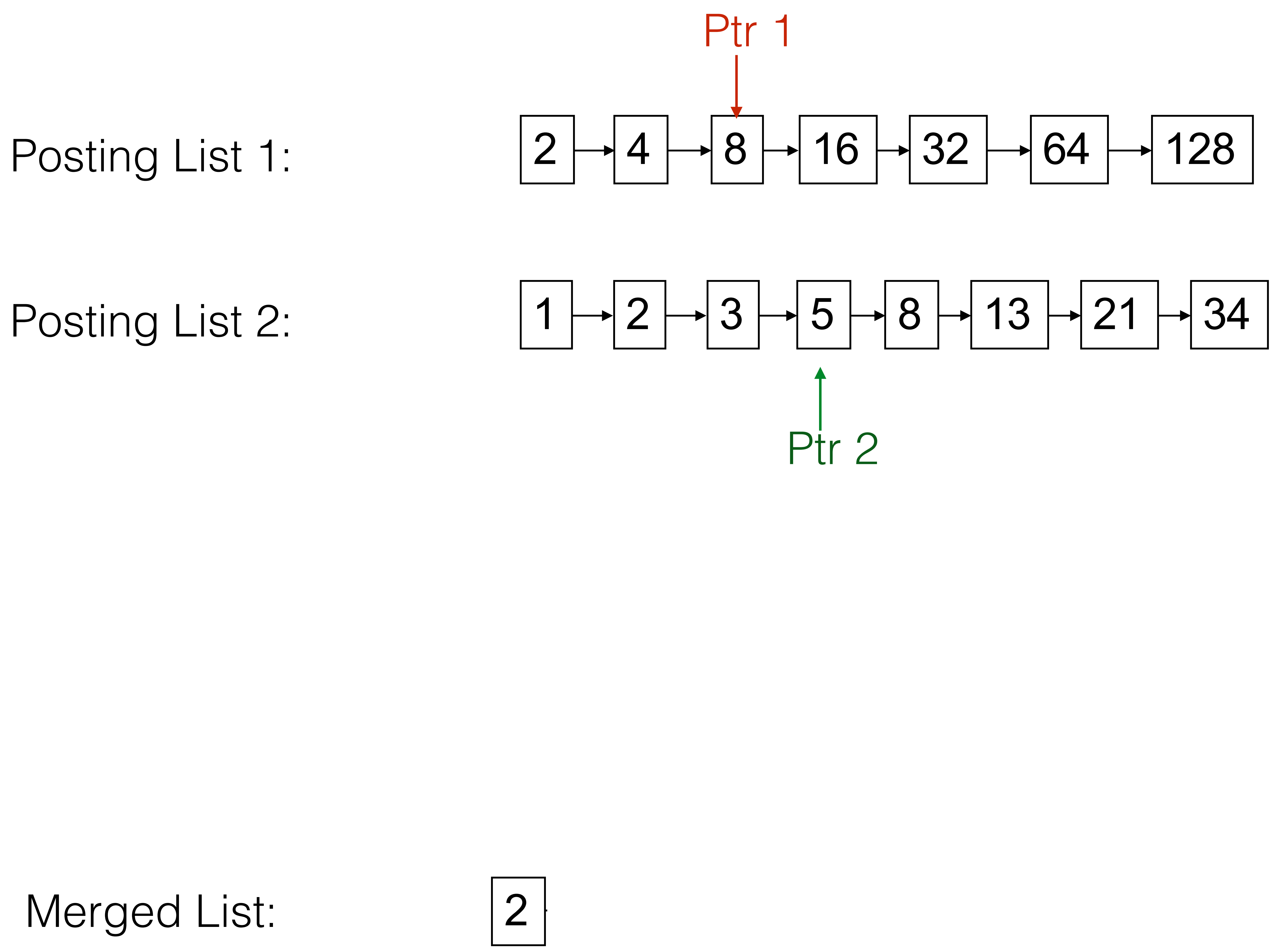
Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?
   A. No - Advance the pointer pointing to the smaller doc ID
   B. Yes - Add doc ID to merged list and advance both pointers
2. Repeat until end

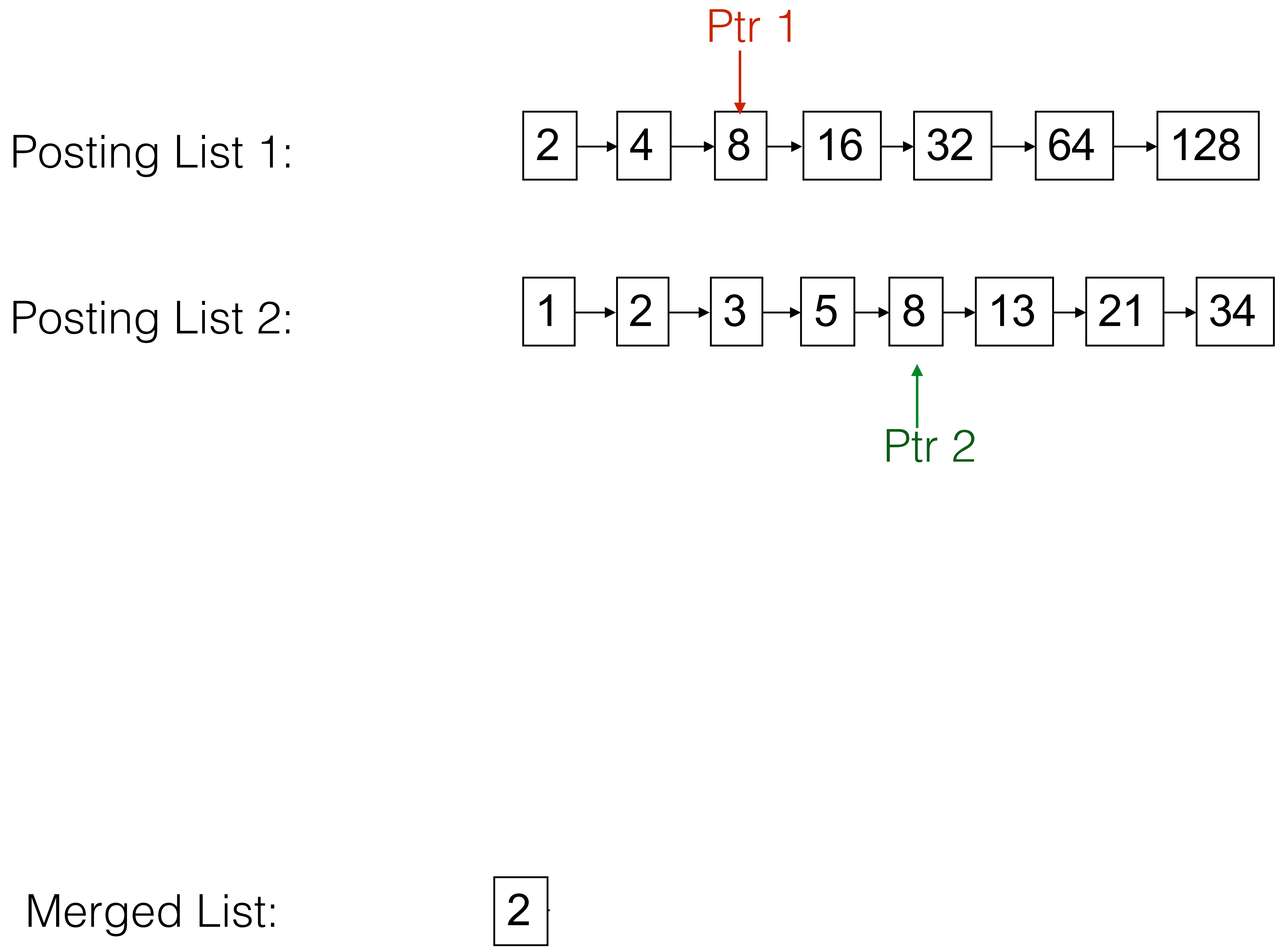
Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?
   A. No - Advance the pointer pointing to the smaller doc ID
   B. Yes - Add doc ID to merged list and advance both pointers
2. Repeat until end

Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

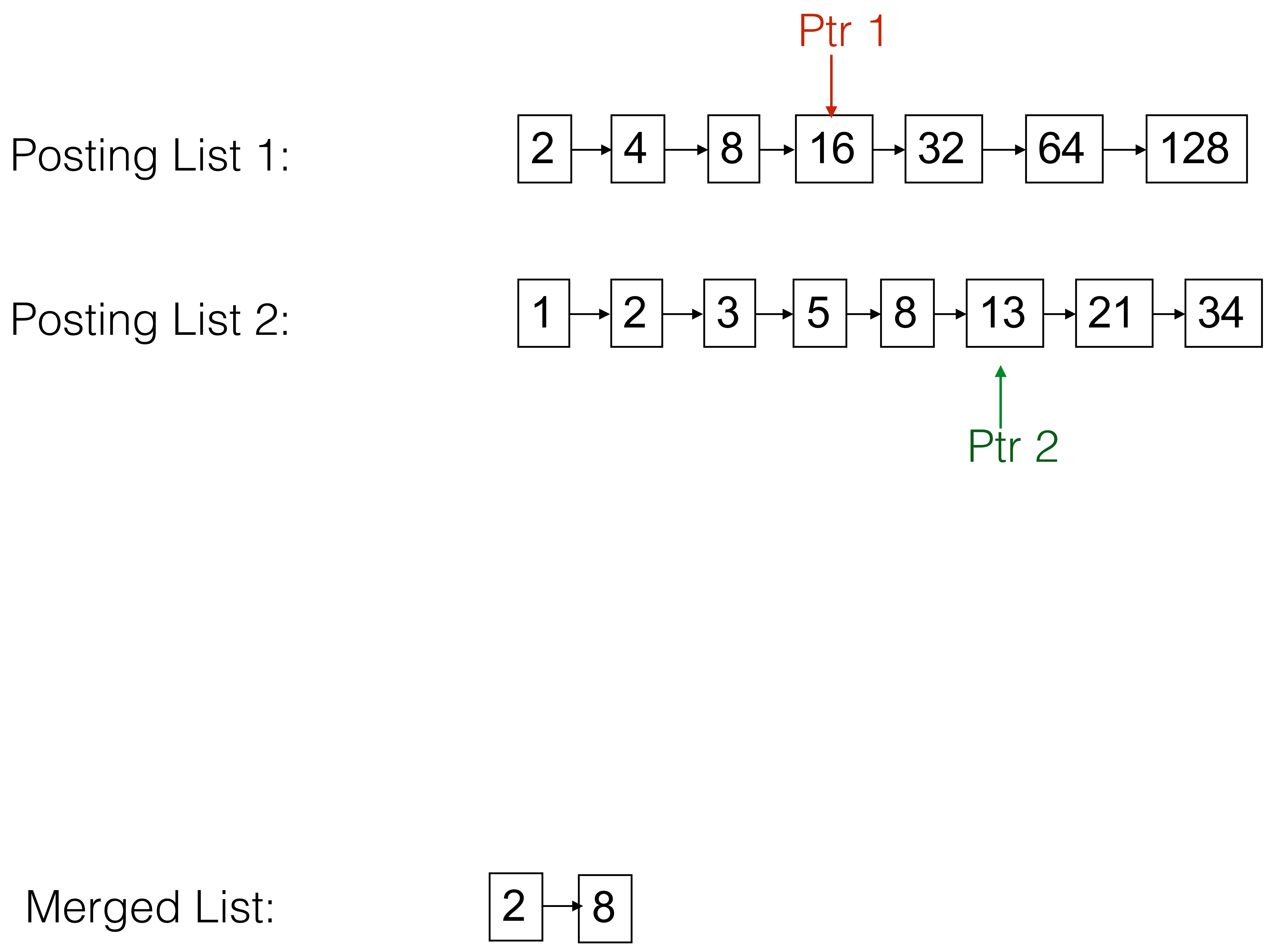
1. Do the doc ID's match?

   A. No - Advance the pointer pointing to the smaller doc ID

   B. Yes - Add doc ID to merged list and advance both pointers

2. Repeat until end

Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

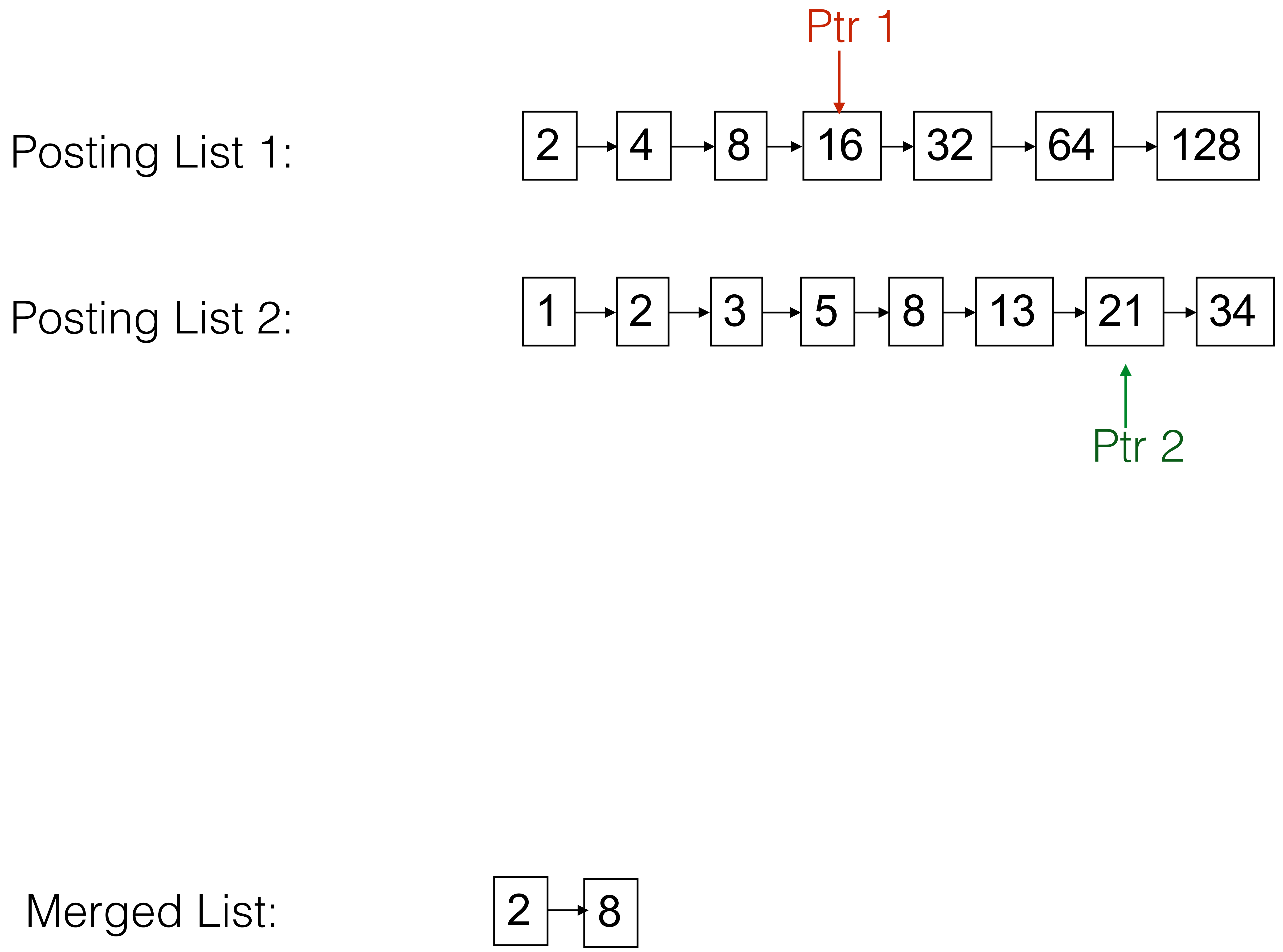
1. Do the doc ID's match?

    A. No - Advance the pointer pointing to the smaller doc ID

    B. Yes - Add doc ID to merged list and advance both pointers

2. Repeat until end

Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

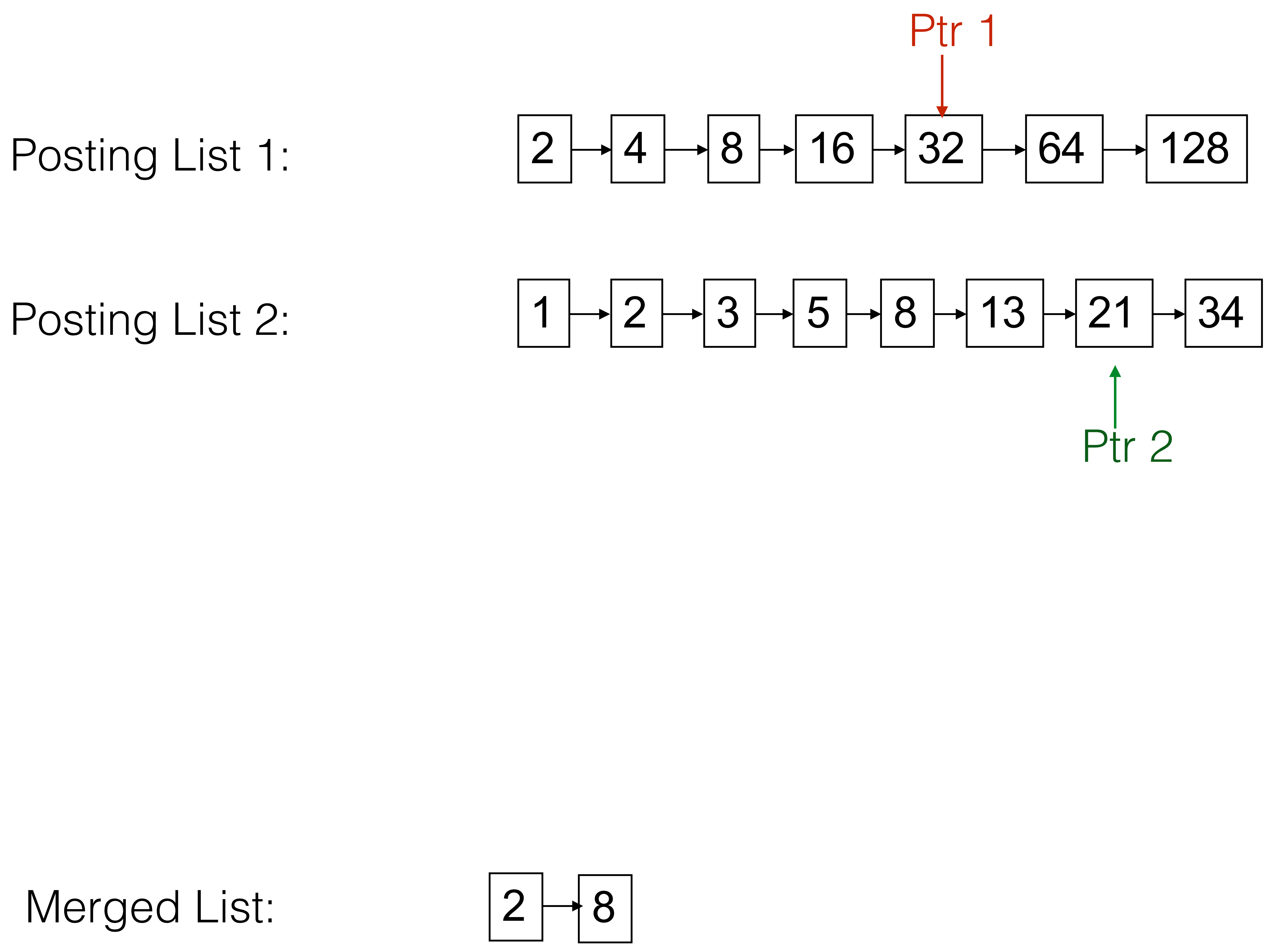
1. Do the doc ID's match?

    A. No - Advance the pointer pointing to the smaller doc ID

    B. Yes - Add doc ID to merged list and advance both pointers

2. Repeat until end

Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

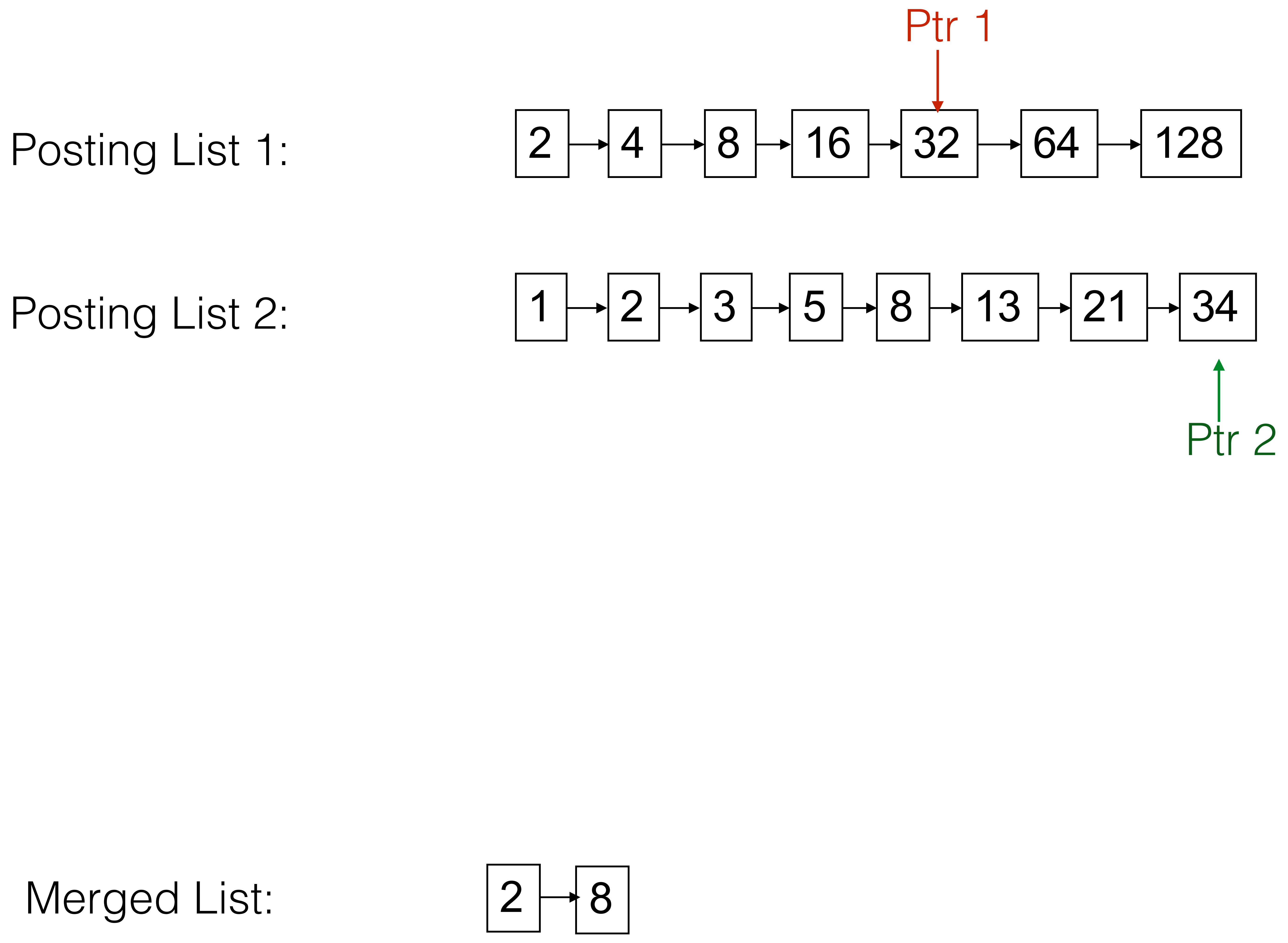
1. Do the doc ID's match?

    A. No - Advance the pointer pointing to the smaller doc ID

    B. Yes - Add doc ID to merged list and advance both pointers

2. Repeat until end

Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

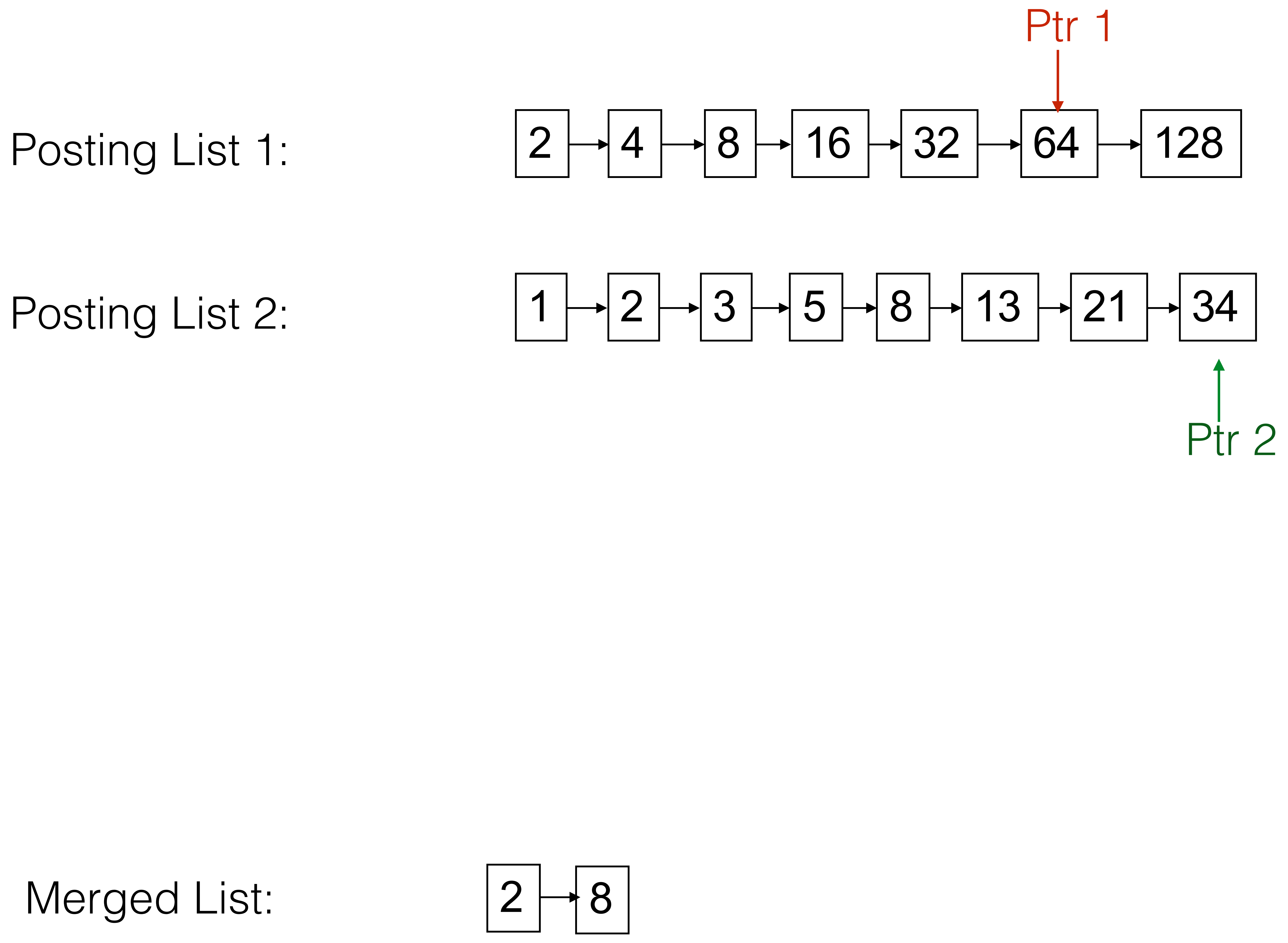
1. Do the doc ID's match?

    A. No - Advance the pointer pointing to the smaller doc ID

    B. Yes - Add doc ID to merged list and advance both pointers

2. Repeat until end

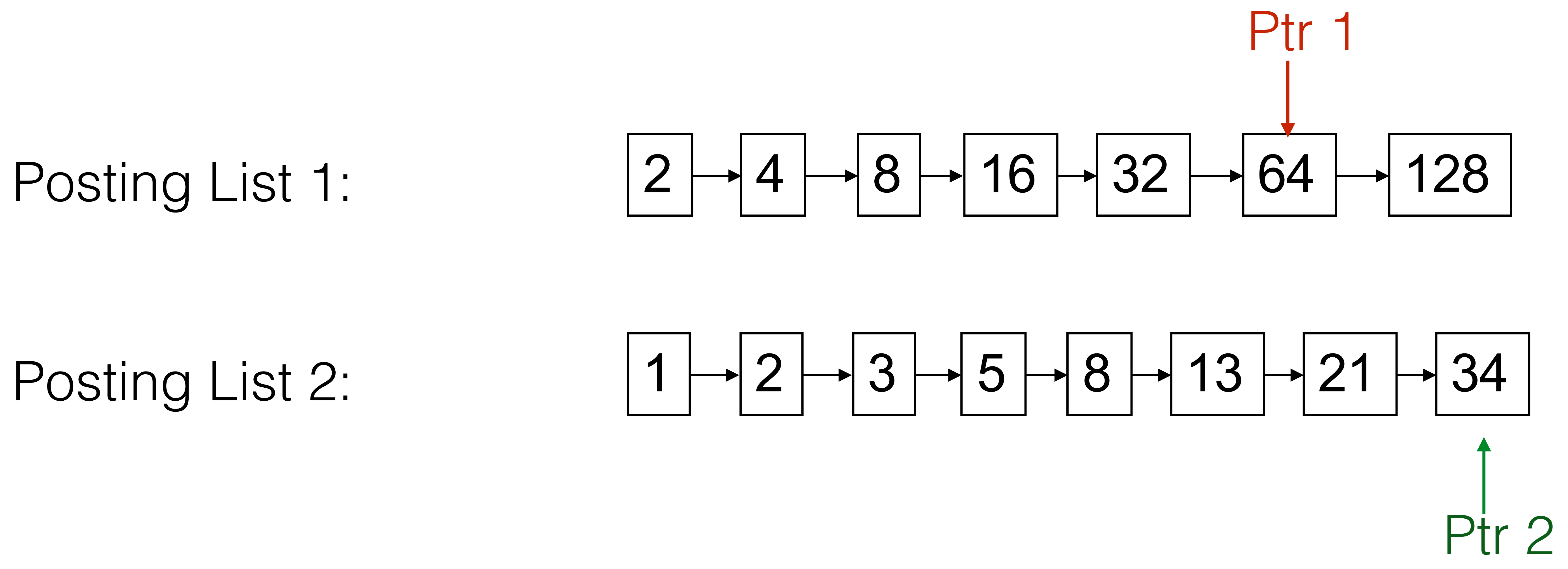
Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

1. Do the doc ID's match?
   A. No - Advance the pointer pointing to the smaller doc ID
   B. Yes - Add doc ID to merged list and advance both pointers
2. Repeat until end

Merging Algorithm



Maintain two pointers that are initialised on first posting from each list respectively

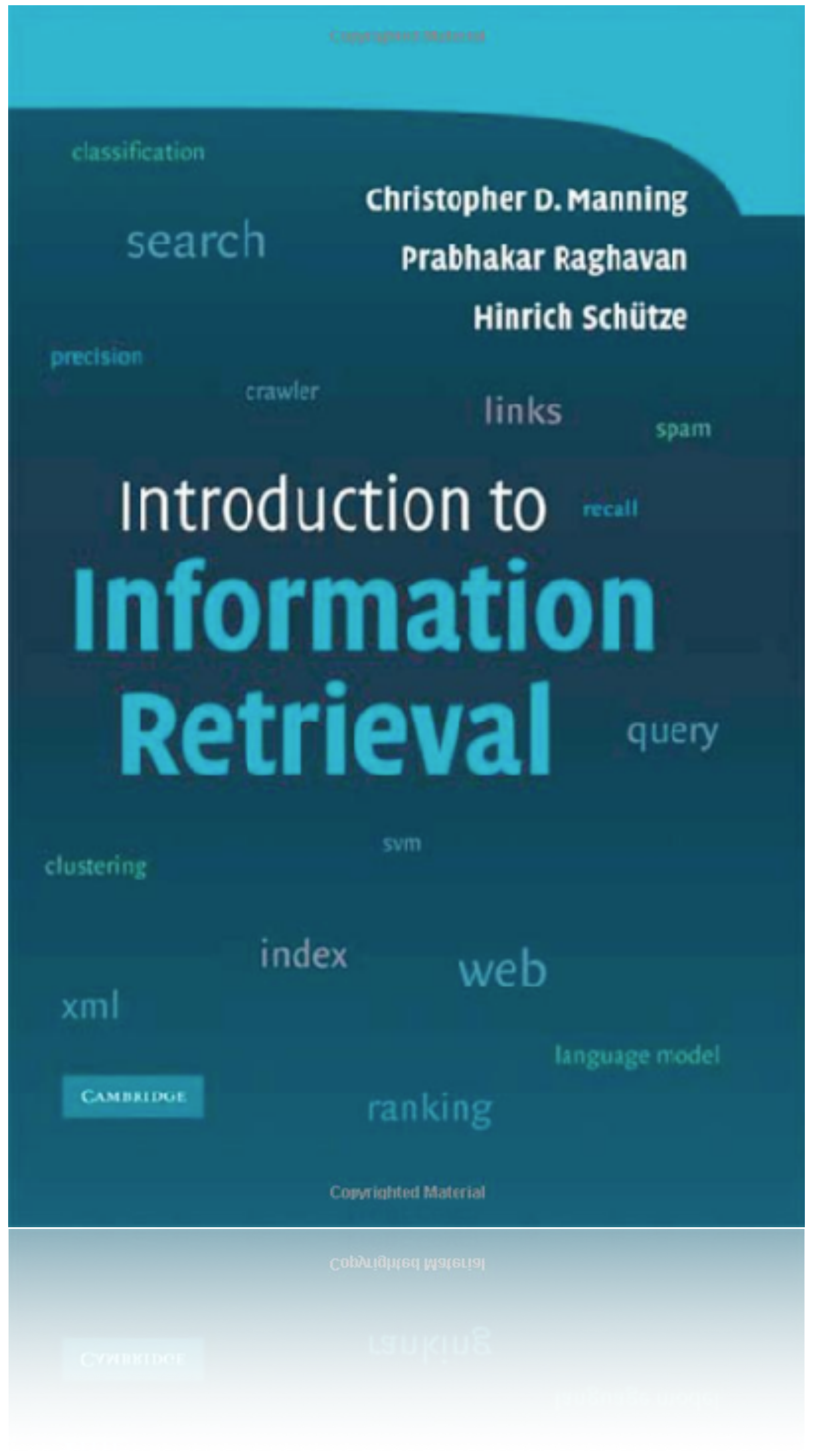
1. Do the doc ID's match?
   - A. No - Advance the pointer pointing to the smaller doc ID
   - B. Yes - Add doc ID to merged list and advance both pointers
2. Repeat until end

- If the posting list lengths are *x* and *y*, the merge takes O(*x*+*y*) operations.
- It is essential that the Document IDs are ordered.

Merged List:
2
8

Merging Algorithm

Check out Section 2.3 of the course text for a way of potentially speeding this algorithm up using *Skip lists*



Dynamic Indexing

Most collections are modified frequently with documents being added, deleted, and updated

As new documents arrive

- Update postings for terms already in dictionary
- Add new terms to dictionary

Documents may also be deleted…

Dynamic Indexing: Periodic Reconstruction

- Periodically reconstruct the index from scratch
  - A good solution if
    - changes are small and infrequent
    - document collection is relatively small
  - New documents are not indexed immediately…

Dynamic Indexing: Auxiliary Index

- Maintain a *main* index and a smaller *auxiliary* index (kept in memory) for new documents
- **Search** across both indexes and merge results
- When *auxiliary* index becomes too large, **merge** it with the *main* index
- By storing the *auxiliary* index in memory, we reduce the number of disk seeks. (We only put additional load on the disk when merging the indexes.)

Phrase (or Proximity) Queries

Example phrase query: “Lonely hearts club band”

- Not sufficient to only index a term’s presence in a document, we also need positional information
- Index every word position?
- Index larger blocks?
    - Sentence
    - Paragraph
    - …
- Cost is a big increase in index size

Index with Term Positions

Doc.	Text
1	Pease pudding hot, pease pudding cold
2	Pease pudding in the pot
3	Nine days old
4	Some like it hot
5	Some like it cold
6	Pease pudding in the pot
7	Nine days old

Term ID	Term	Postings
1	pease	(1:2{1,4}),(2:1{1}),(6:1{1})
2	pudding	(1:2{2,5}),(2:1{2}),(6:1{2})
3	hot	(1:1{3}),(4:1{4})
4	cold	(1:1{6}),(5:1{4})
5	pot	(2:1{5}),(6:1{5})
6	nine	(3:1{1}),(7:1{1})
7	days	(3:1{2}),(7:1{2})
8	old	(3:1{3}),(3:1{3})
9	some	(4:1{1}),(5:1{1})
10	like	(4:1{2}),(5:1{2})

(docID:termFreq{termPosition1,termPosition2,…})

Indexing Granularity

- The position can be represented in the index at different granularities:
    - Block within document
        - paragraph, section, … (what paragraphs/sections/… does it appear in?)
    - Word position
        - full inverted index (what positions within the document does it appear in?)
- Coarse granularities are less precise, but take less space

Index Lookup Cost vs. Granularity

- Index size
  - finer grain => larger index
  - a paragraph-level index is typically 5% the size of a word-level inverted index
- Tradeoff
  - full inverted index makes proximity and phrase searches easy
  - block addressing means blocks must be searched online (or only perform an approximate query search)
  - for big collections block indexes reduce disk accesses

Index Construction: Case Study

The Problem

- We have a collection of documents (a corpus) that contains the following:
  - Number of documents = n = 1M
  - Number of words per document = 1K
  - Number of distinct terms in collection = m = 500K
- We expect 667 million postings entries (see Chapter 4 of course text for details)
- The goal is to sort these postings. (It is essential that the Document IDs are ordered for merging.)
  - Corpus is too large to fit in memory
  - We want to minimise the number of disk seeks

System Parameters for Design

- Assume:
  - Disk access takes approx. 5 milliseconds (0.005 seconds)
  - All other operations take approx. 0.01 μseconds (1e-8 seconds)
  - E.g., to compare two postings entries and decide their merge order

The Sorting Bottleneck

- Number of documents = n = 1M
- Number of words per document = 1K
- Number of distinct terms in collection = m = 500K
- Number of postings entries = 667M

If every comparison takes 2 disk seeks, and *N* items could be sorted with $N \log_2 N$ comparisons, how long would this take?

The Sorting Bottleneck

- Number of documents = n = 1M
- Number of words per document = 1K
- Number of distinct terms in collection = m = 500K
- Number of postings entries = 667M

If every comparison takes 2 disk seeks, and *N* items could be sorted with $N \log_2 N$ comparisons, how long would this take?

9256401 seconds
(or 15 weeks)

Blocked Sort-Based Indexing Algorithm

- Split the records into blocks which will fit into memory
- Sort each block and write to disk
- Merge sorted blocks

Blocked Sort-Based Indexing Algorithm

- Create blocks of 10M postings (a number that can be loaded into memory)
    - Our case study would have 66 blocks
- Can fit multiple blocks into memory at once
- Sort each block
- Merge sorted blocks and store to disk

- Well suited to parallel processing (take a look into MapReduce architectures)

Indexing in Python

Python Dictionary as an Index

Keys (Terms)		**Values** (Doc. ID and Freq.)
‘a’	→	[[1, 2], [4, 1], [5,12]]
‘and’	→	[[1, 4], [2, 2], [3,13], [4, 3] [5, 6]]
‘arm’	→	[[4, 1], [5, 2]]
…		
‘yellow’	→	[[1, 1], [5, 1]]

- Checking for existence is very fast
- Keys can be (normalised) term, and values can be list of document ID, frequency pairs

Summary

- Inverted indexes
- Index construction
- Index sorting
- Indexing in Python