Part 2: Boolean Retrieval

Francesco Ricci

Most of these slides comes from the course:
Information Retrieval and Web Search,
Christopher Manning and Prabhatkar Raghavan
Content

- Term document matrix
- Information needs and evaluation of IR
- Inverted index
- Processing Boolean queries
- The merge algorithm
- Query optimization
- Skip pointers
- Dictionary data structures
  - Hash tables
  - Binary trees
### Term-document incidence

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1 if play contains word, 0 otherwise

**Brutus AND Caesar BUT NOT Calpurnia**
Incidence vectors

- So we have a 0/1 vector for each term
- To answer query:
  - Brutus, Caesar and NOT Calpurnia
  - take the vectors for
    - Brutus 110100
    - Caesar 110111
    - Calpurnia (complemented) 101111
  - Bitwise AND
    - 110100 AND 110111 AND 101111 = 100100
Antony and Cleopatra, Act III, Scene ii

_Agrippa_ [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
When Antony found Julius _Caesar_ dead,
He cried almost to roaring; and he wept
When at Philippi he found _Brutus_ slain.

Hamlet, Act III, Scene ii

_Lord Polonius_: I did enact Julius _Caesar_ I was killed i' the Capitol; _Brutus_ killed me.

http://www.rhymezone.com/shakespeare/
Basic assumptions of IR

- **Collection**: fixed set of documents
- **Goal**: retrieve documents with information that is relevant to the user’s information need and helps the user complete a task
- Using the **Boolean Retrieval Model** means that the information need must be translated into a **Boolean expression**:
  - terms combined with AND, OR, and NOT operators
- We want to support **ad hoc retrieval**: provide documents relevant to an arbitrary user information need.
How good are the retrieved docs?

- **Precision**: Fraction of retrieved docs that are relevant to user’s information need
- **Recall**: Fraction of relevant docs in collection that are retrieved

More precise definitions and measurements to follow in another lecture on evaluation.
Relevance

- Relevance is the core concept in IR, but nobody has a good definition
  - Relevance = useful
  - Relevance = topically related
  - Relevance = new
  - Relevance = interesting
  - Relevance = ???

- Relevance is very dynamic – it depends on the needs of a person at a specific point in time

- *The same result for the same query may be relevant for a user and not relevant for another*
Assumption: A document is relevant to the information need expressed by a query if it satisfies the Boolean expression of the query.

Question: Is it always true?

No: consider for instance a collection of documents dated before 2014, and the query is "oscar AND 2014". Would the documents retrieved by this query relevant?
Relevance and Retrieved documents

Information need

relevant

not relevant

Ex: "lincoln"

Documents

TP

FP

not retrieved

FN

TN

retrieved

Query and system

Precision $P = \frac{tp}{(tp + fp)}$

$= \frac{tp}{\text{retrieved}}$

Recall $R = \frac{tp}{(tp + fn)}$

$= \frac{tp}{\text{relevant}}$
### Term-document incidence

<table>
<thead>
<tr>
<th>Term</th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Brutus AND Caesar BUT NOT Calpurnia

1 if play contains word, 0 otherwise
Bigger collections

- Consider a more realistic case
- 1M (million) documents, each with about 1000 words
- Avg 6 bytes/word including spaces/punctuation
  - 6GB of data in the documents
- Say there are 500K *distinct* terms among these
- 500K x 1M matrix has half-a-trillion 0’s and 1’s
- But it has no more than one billion 1’s
  - matrix is extremely sparse
- What’s a better representation?
  - We only record the positions of the 1's.
Inverted index

For each term $t$, we must store a list of all documents that contain $t$

- Identify each by a **docID**, a document serial number

- Can we use fixed-size arrays for this?

<table>
<thead>
<tr>
<th>Brutus</th>
<th>$\rightarrow$</th>
<th>1 2 4 11 31 45 173 174</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caesar</td>
<td>$\rightarrow$</td>
<td>1 2 4 5 6 16 57 132</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>$\rightarrow$</td>
<td>2 31 54 101</td>
</tr>
</tbody>
</table>

What happens if the word *Caesar* is added to document 14?
Inverted index

- We need variable-size postings lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion

```
<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>1 2 4 11 31 45 173 174</td>
</tr>
<tr>
<td>Caesar</td>
<td>1 2 4 5 6 16 57 132</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2 31 54 101</td>
</tr>
</tbody>
</table>
```

Sorted by docID (more later on why)
Inverted index construction

Documents to be indexed

Token stream.

More on these later.

Modified tokens

Linguistic modules

Indexer

Inverted index

Friends, Romans, countrymen.

Friends

Romans

Countrymen

friend

roman

countryman

friend

roman

countryman

2 → 4

1 → 2

13 → 16

More on these later.
Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

```
Doc 1
I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious
```

<table>
<thead>
<tr>
<th>Term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>
# Indexer steps: Sort

- Sort by terms
  - And then docID

## Core indexing step

<table>
<thead>
<tr>
<th>Term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>'t</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>1</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>'t</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
</tbody>
</table>
Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged
- Split into Dictionary and Postings
- Doc. frequency information is added.

Why frequency? Will discuss later
Where do we pay in storage?

Terms and counts

Lists of docIDs

Later in the course:
• How do we index efficiently?
• How much storage do we need?
Exercise

- How many bytes do we need to store the inverted index if there are:
  - \( N = 1 \) million documents, each with about 1000 words
  - Say there are \( M = 500K \) distinct terms among these
  - *We need to store: term IDs, doc frequencies, pointers to postings lists, list of doc IDs (postings).*
Exercise Solution

- \( \log_2(500,000) = 19 \) bits are required for representing the terms and the pointers to their postings lists
  - Hence 3 bytes (= 24 bits, representing 16.7M of alternatives) are enough for each term and pointer
- 3 bytes for each term frequency (the largest term frequency is 1M = \# of docs)
  - Hence \( 9 \times 500,000 = 4.5 \times 10^6 \)
- We have at most 1 billion postings (\# of tokens in documents), hence 3 bytes for each posting (docid) = \( 3 \times 10^9 \)
- In total \( 3,004,500,000 \sim 3 \) GB
The index we just built

- How do we process a query?
- Later - what kinds of queries can we process?
Consider processing the query:

\textit{Brutus AND Caesar}

- Locate \textit{Brutus} in the Dictionary
  - Retrieve its postings
- Locate \textit{Caesar} in the Dictionary
  - Retrieve its postings
- “Merge” the two postings

How we can merge?
The idea

brutus  nn  02  nn  04  nn  nn  nn  nn  nn  nn  nn  nn  16
cesar   01  02  nn  nn  05  nn  nn  08  nn  nn  nn  13  nn  nn  nn
position 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16

- If we have the incidence vectors we scan in parallel the entries of the two vectors – starting from the first position (here I wrote the doc id, e.g., "08", instead of 1 and "nn" instead of 0)
- Try to replicate this idea but imagine that in these two arrays you removed the "nn" entries ...
- Keep a pointer to each list, advance the pointer to the smallest docID and check if now the pointers refer to the same docID.
The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries.

If the list lengths are $x$ and $y$, the merge takes $O(x+y)$ operations.

**Crucial:** postings sorted by docID.
Intersecting two postings lists (a “merge” algorithm)

\[
\text{INTERSECT}(p_1, p_2)
\]

1. \( \text{answer} \leftarrow \langle \rangle \)
2. while \( p_1 \neq \text{NIL} \) and \( p_2 \neq \text{NIL} \)
3. do if \( \text{docID}(p_1) = \text{docID}(p_2) \)
4. \hspace{1em} then \( \text{ADD}(\text{answer}, \text{docID}(p_1)) \)
5. \hspace{1em} \( p_1 \leftarrow \text{next}(p_1) \)
6. \hspace{1em} \( p_2 \leftarrow \text{next}(p_2) \)
7. else if \( \text{docID}(p_1) < \text{docID}(p_2) \)
8. \hspace{1em} then \( p_1 \leftarrow \text{next}(p_1) \)
9. \hspace{1em} else \( p_2 \leftarrow \text{next}(p_2) \)
10. return \( \text{answer} \)
Boolean queries: Exact match

- The **Boolean retrieval model** is being able to ask a query that is a Boolean expression:
  - Boolean Queries are queries using *AND*, *OR* and *NOT* to join query terms
    - Views each document as a set of words
    - Is precise: document matches condition or not.
  - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades
- Many search systems you still use are Boolean:
  - Email, library catalog, Mac OS X Spotlight.
Example: WestLaw  
http://www.westlaw.com/

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- Tens of terabytes of data; 700,000 users
- Majority of users *still* use boolean queries
- Example query:
  - *What is the statute of limitations in cases involving the federal tort claims act?*
  - LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM
    - /3 = within 3 words, /S = in the same sentence
More general merges

- **Exercise**: Adapt the merge for the queries:
  
  *Brutus AND NOT Caesar*
  
  *Brutus OR NOT Caesar*

Can we still run through the merge in time $O(x+y)$?
What can we achieve?
Merging

What about an arbitrary Boolean formula? 
\((\text{Brutus OR Caesar}) \text{ AND NOT } (\text{Antony OR Cleopatra})\)

- Can we always merge in “linear” time?
  - Linear in what?
- Can we do better?
Query optimization

- What is the best order for query processing?
- Consider a query that is an AND of $n$ terms
- For each of the $n$ terms, get its postings, then AND them together

Query: *Brutus* AND *Calpurnia* AND *Caesar*
Query optimization example

- Process in order of increasing term freq, i.e., posting list length:
  - start with smallest set, then keep cutting further.

  This is why we kept document freq. in dictionary

<table>
<thead>
<tr>
<th>Brutus</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caesar</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>13</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execute the query as (Calpurnia AND Brutus) AND Caesar.
More general optimization

- e.g., \( \text{madding OR crowd} \) AND \( \text{ignoble OR strife} \)
- Get doc. freq.’s for all terms
- Estimate the size of each OR by the sum of its doc. freq.’s (conservative)
- Process in increasing order of OR sizes.
Algorithm for conjunctive queries

**INTERSECT**($\langle t_1, \ldots, t_n \rangle$)

1. $\text{terms} \leftarrow \text{SORTBYINCREASINGFREQUENCY}(\langle t_1, \ldots, t_n \rangle)$
2. $\text{result} \leftarrow \text{postings}(\text{first(terms)})$
3. $\text{terms} \leftarrow \text{rest(terms)}$
4. **while** $\text{terms} \neq \text{NIL}$ and $\text{result} \neq \text{NIL}$
5. **do** $\text{result} \leftarrow \text{INTERSECT}($result, postings(first(terms))$)$
6. $\text{terms} \leftarrow \text{rest(terms)}$
7. **return** $\text{result}$

- The intermediate result is in memory
- The list is being intersected with is read from disk
- *The intermediate result is always shorter and shorter*
Exercise

- Recommend a query processing order for

\[(\text{tangerine OR trees}) \land (\text{marmalade OR skies}) \land (\text{kaleidoscope OR eyes})\]

<table>
<thead>
<tr>
<th>Term</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>eyes</td>
<td>213312</td>
</tr>
<tr>
<td>kaleidoscope</td>
<td>87009</td>
</tr>
<tr>
<td>marmalade</td>
<td>107913</td>
</tr>
<tr>
<td>skies</td>
<td>271658</td>
</tr>
<tr>
<td>tangerine</td>
<td>46653</td>
</tr>
<tr>
<td>trees</td>
<td>316812</td>
</tr>
</tbody>
</table>
What’s ahead in IR? Beyond term search

- What about phrases?
  - Stanford University
- Proximity: Find Gates NEAR Microsoft.
  - Need index to capture position information in docs.
- Zones in documents: Find documents with (author = Ullman) AND (text contains automata).
Evidence accumulation

- 1 vs. 0 occurrence of a search term
  - 2 vs. 1 occurrence
  - 3 vs. 2 occurrences, etc.
  - Usually more seems better
- Need term frequency information in docs
FASTER POSTINGS MERGES:
SKIP POINTERS/SKIP LISTS
Recall basic merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries

If the list lengths are $m$ and $n$, the merge takes $O(m+n)$ operations.

Can we do better?
Yes (if index isn’t changing too fast).
Augment postings with **skip pointers** (at indexing time)

- **Why?**
- **To skip postings that will not figure in the search results.**
- **How?**
- **Where do we place skip pointers?**
Query processing with skip pointers

Suppose we’ve stepped through the lists until we process 8 on each list. We match it and advance.

We then have 41 and 11 on the lower. 11 is smaller.

But instead to advance to 17 the skip successor of 11 on the lower list is 31, and it is smaller than 41, so we can skip ahead.
Intersect with skip pointers

\[
\text{INTERSECTWITHSKIPS}(p_1, p_2)
\]

1. \(\text{answer} \leftarrow \emptyset\)
2. \(\text{while } p_1 \neq \text{NIL} \text{ and } p_2 \neq \text{NIL}\)
3. \(\text{do if } \text{docID}(p_1) = \text{docID}(p_2)\)
   \(\quad\text{then } \text{ADD}(\text{answer}, \text{docID}(p_1))\)
   \(\quad p_1 \leftarrow \text{next}(p_1)\)
   \(\quad p_2 \leftarrow \text{next}(p_2)\)
4. \(\text{else if } \text{docID}(p_1) < \text{docID}(p_2)\)
   \(\quad\text{then if } \text{hasSkip}(p_1) \text{ and } (\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))\)
   \(\quad\quad\text{then while } \text{hasSkip}(p_1) \text{ and } (\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))\)
   \(\quad\quad\quad\text{do } p_1 \leftarrow \text{skip}(p_1)\)
   \(\quad\quad\text{else } p_1 \leftarrow \text{next}(p_1)\)
   \(\text{else if } \text{hasSkip}(p_2) \text{ and } (\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))\)
   \(\quad\text{then while } \text{hasSkip}(p_2) \text{ and } (\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))\)
   \(\quad\quad\text{do } p_2 \leftarrow \text{skip}(p_2)\)
   \(\quad\text{else } p_2 \leftarrow \text{next}(p_2)\)
5. \(\text{return } \text{answer}\)
Where do we place skips?

- Tradeoff:
  - More skips $\rightarrow$ shorter skip spans $\rightarrow$ more likely to skip. But lots of comparisons to skip pointers.

- Fewer skips $\rightarrow$ few pointer comparison, but then long skip spans $\rightarrow$ few successful skips.
Placing skips

- Simple heuristic: for postings of length $L$, use $\sqrt{L}$ evenly-spaced skip pointers.
- This takes into account the distribution of query terms in a simple way – the larger the doc frequency of a term the larger the number of skip pointers.
- Easy if the index is relatively static; harder if postings keep changing because of updates.

- This definitely used to help; with modern hardware it may not (Bahle et al. 2002) unless you’re memory-based:
  - because the I/O cost of loading a bigger index structure can outweigh the gains from quicker in memory merging!
Dictionary data structures for inverted indexes

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?

<table>
<thead>
<tr>
<th>Term</th>
<th>Pointer to Postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>1 2 4 11 31 45 173 174</td>
</tr>
<tr>
<td>Caesar</td>
<td>1 2 4 5 6 16 57 132 …</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2 31 54 101</td>
</tr>
</tbody>
</table>

...
A naïve dictionary

- An array of struct:

<table>
<thead>
<tr>
<th>term</th>
<th>document frequency</th>
<th>pointer to postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>656,265</td>
<td>→</td>
</tr>
<tr>
<td>aachen</td>
<td>65</td>
<td>→</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>zulu</td>
<td>221</td>
<td>→</td>
</tr>
</tbody>
</table>

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?
Dictionary data structures

- Two main choices:
  - Hash table
  - Tree
- Some IR systems use hashes, some trees
Hashes

- Each vocabulary term is hashed to an integer
  - (We assume you’ve seen hashtables before)
- **Pros:**
  - Lookup is faster than for a tree: $O(1)$
- **Cons:**
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search ("bar*") [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*
Tree: binary tree
Definition: Every internal node has a number of children in the interval \([a,b]\) where \(a, b\) are appropriate natural numbers, e.g., \([2,4]\).
Trees

- *Simplest*: binary tree
- *More usual*: B-trees

Trees require a standard ordering of characters and hence strings ... but we have one – lexicographic

- Unless we are dealing with Chinese (no unique ordering)

- **Pros:**
  - Solves the prefix problem (terms starting with 'hyp')

- **Cons:**
  - Slower: $O(\log M)$ [and this requires balanced tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem.
Reading Material

- Chapter 1
- Section 2.3: Faster postings list intersection via skip pointers
- Section 3.1: Search structures for dictionaries