Stemming & Lemmatization:

Root form of a words
↳ stem / lemma.

~~Lemmatization~~ Lemmatization
↳ proper reduction of the word to its lemma (usually by matching the headword from a dictionary).

am, are, is → be

car, car's, cars, cars' → car

inflections.
(morphology → surface form of the word).

Stemming
↳ chops the end of a word and assumes that the stem has been obtained.
→ heuristics.

① car (N) → cars (N)
② automate (V) → automation (N)

derivational morphology

→ inflectional morphology.

Text processing for IR systems-

Stop words & ~~their~~ their removal.

→ words (do not have ~~their~~ a meaning of their own)

→ Render meaning to other words in a query / document.

Articles, prepositions, conjunction, pronouns, auxiliary verbs ← stop words.

{a, an, the, will, was, were, has, he, in, ...

→ very frequent in a document

Some of the most frequent words in documents.

Design policy: not separately index these words.

Caveats: phrase queries "Joan of Arc"

automation, automatic .... → automat

Algo that does stemming → stemmer

(Porter Stemmer)

Stage 1

✓ sses → ss (caresses → caress)

ies → i (ponies → poni)

ss → ss (caress → caress)

✓ s → ϕ (cats → cat)

caresses ← Longest match from the end first.

Stage 2.

vowel.

(*v*) ing → ϕ [talking → walk, king → king]

(*v*) ed → ϕ [played → play]

Stage 3

only 3 among many rules

- ational → ate (relational → relate)
- izer → ize (digitizer → digitize)
- ator → ate (operator → operate)

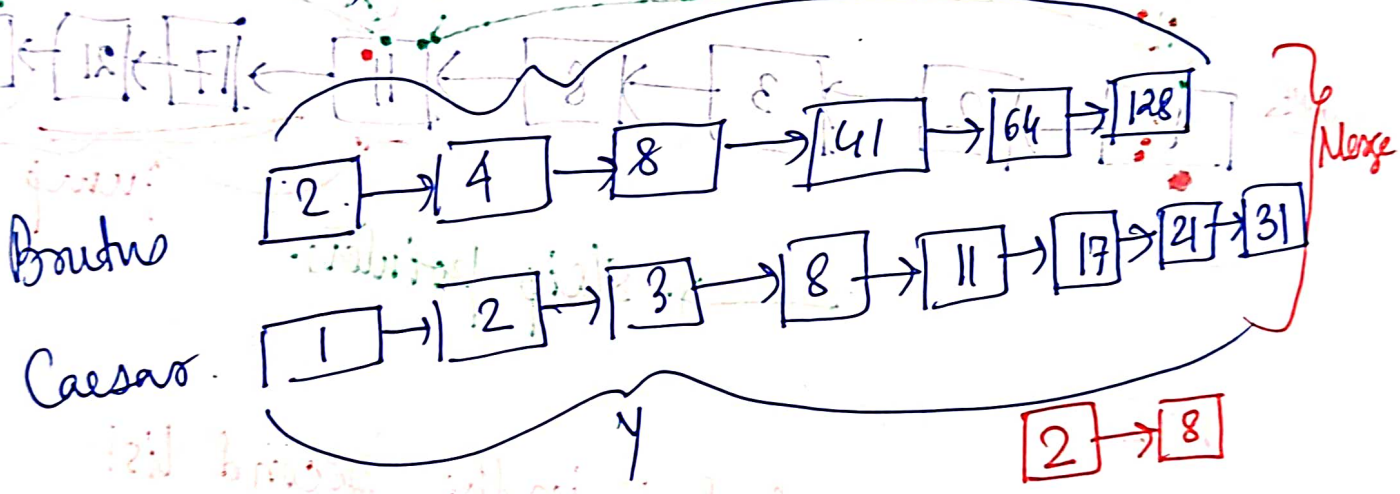
Stage 4.

- al → ϕ (revival → reviv).
- able → ϕ (adjustable → adjust)
- ate → ϕ (activate → activ).

automate → automat (N)

Making postings list operations more efficient → skip pointers

Phrase queries → Positional indexing



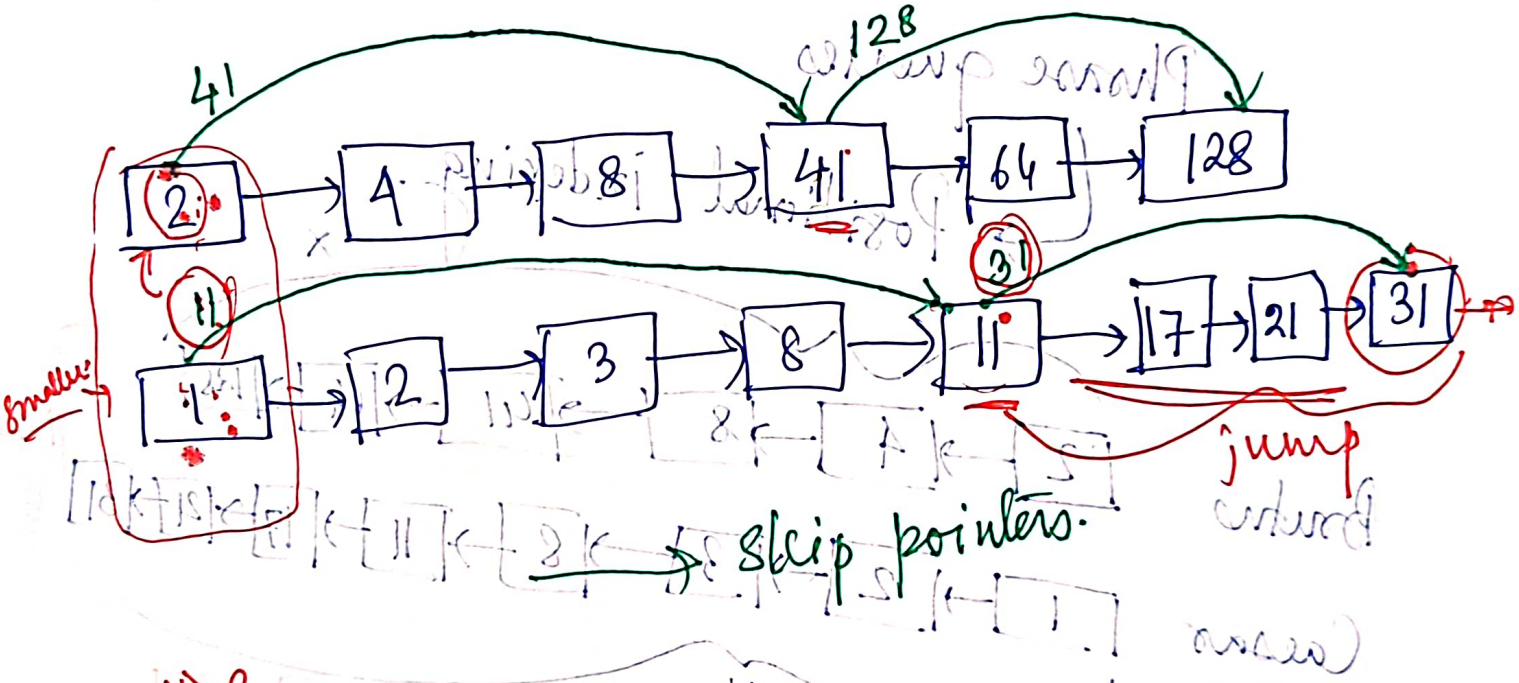
$O(X+Y)$

Can we make it any better?

Augment the postings list → Skip pointers

Postings lists are not changing too fast.

Skip those docIDs which will not feature in the search.



11 > 2

Till we have reached 11 in the second list

31 < 41

INTERSECT WITH SKIP (p_1, p_2)

1. answer ← < >

while $p_1 \neq$ NIL && $p_2 \neq$ NIL

do if docID(p_1) = docID(p_2)

then ADD(answer, docID(p_2))

$p_1 \leftarrow$ next(p_1)

$p_2 \leftarrow$ next(p_2)

else if docID(p_1) < docID(p_2)

then if hasSkip(p_1) && (docID(skip(p_1)) ≤ docID(p_2))

then while hasSkip(p_1) &&

(docID(skip(p_1)) ≤ docID(p_2))

do $p_1 \leftarrow$ skip(p_1)

else $p_1 \leftarrow$ next(p_1)

else if hasSkip(p_2) && (docID(skip(p_2)) ≤ docID(p_1))

then while hasSkip(p_2) && (docID(skip(p_2))

≤ docID(p_1))

do $p_2 \leftarrow$ skip(p_2)

else $p_2 \leftarrow$ next(p_2)

return answer

① How many skip pointers?] IR designer will have restrictions.
② Where to place them?

Tradeoff for Q. ①

More skips. — skip spans will be shorter. hardly miss anything
Many pointer comparisons.

Less skips. — Less pointer comparisons.
Unsuccessful frequently.

② If the list size is L then

[Moffat & Zobel 1996]

Keeping $\sqrt{L}$ skip pointers evenly placed empirically gives the best performance.
Easy → L is not growing frequently.

More pointers means more memory.

I/O of bringing skip pointer laden postings list from disk to memory can surpass the benefit of merging time saved by these additional pointers.

Phrase queries.

[Stanford university] → together as an unit.

will return such cases: Stanford. AND University

The inventor Stanford Ovshinsky never went to university

10% of web queries → phrase queries.

→ biword indexing.

→ positional indexing.

Biword indexing → index as biphrase. → "Friends, Romans, Countrymen"

"friends romans", "romans countrymen" → Each of these biwords → added to the dictionary.

"stanford university palo alto" → stanford university AND palo alto

Caveat. "abolition of slavery"

Extended biwords. What is the part-of-speech of each word in query / document? → POS-tagger.

Bucket all the terms that are nouns (N) and articles/prepositions (X).

Extended biword: NX*N

Catcher	in	the	rye
N	X	X	N

King	of	Denmark
N	X	N

New vocabulary → all the extend biwords of the form NX*N

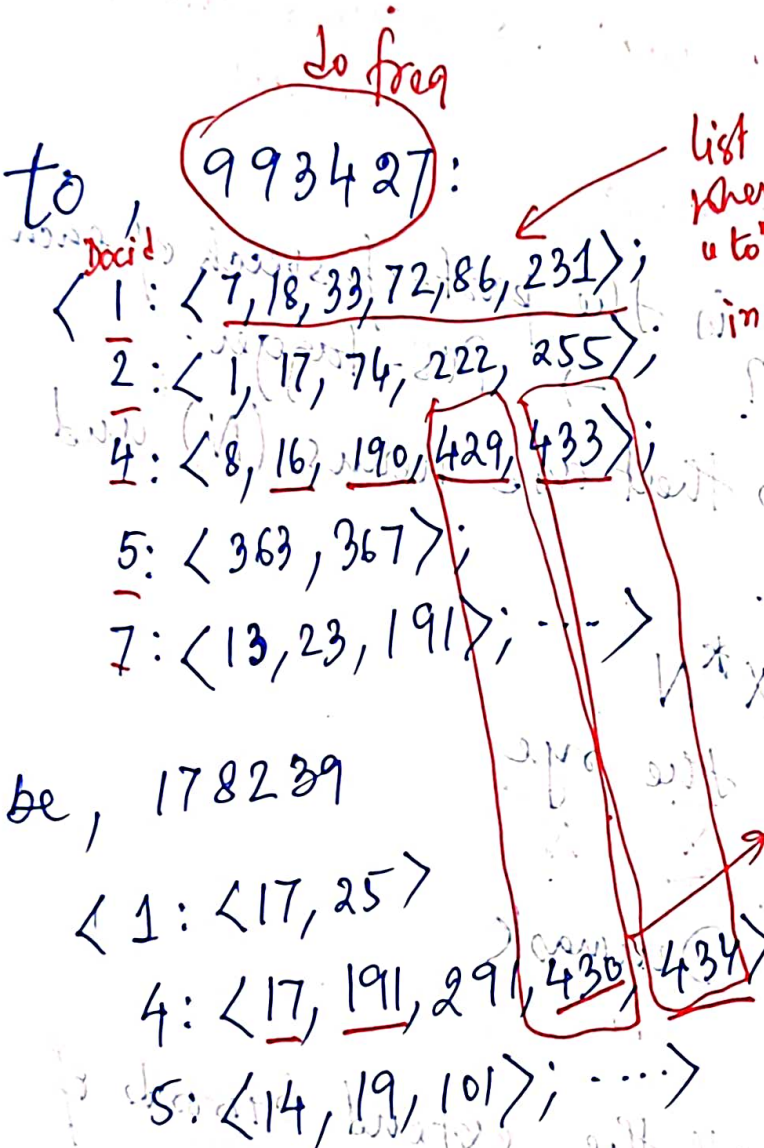
Caveat: Could easily blow up.

Positional indexes. (reconstruction of the postings list).

Postings list in a nonpositional index: each posting is just a docID

Postings list in a positional index: each posting is a docID + a list of positions.

"(to)1 be2 or3 not4 (to)5 be6"



① Get the postings list of to, be, or, not.

② Progressive intersect the postings list starting with to & be

docID = 2 ← removed easily

NOT EQUALITY ← merge the positional merge checks for difference of k=1

merge within merge