CS246
C:Text

May 3

Lab

- Find evidence that your browser really does things in parallel.
  - Hint, in chrome, look in Developer / Developer Tools
  - You can definitely do this without using developer tools, but you are a developer so …
- Send a screenshot (or picture) of something that shows parallel actions with a brief paragraph of how your screenshot shows parallel.

- Open chrome and show the console

Does a string contain a substring? Where? All Where?

- Brute Force
- Indexing
  - Inverted Indices
- Boyer Moore Algorithm
- Knuth-Morris-Pratt

Brute Force

- call string searched for “needle”
- call string searched within “haystack”

- start at the first letter, if same
    - compare next letter of each
        - …
- not same, go back to first letter of needle and the next letter of haystack …

- repeat — possibly a lot

Code

- struct to allow multiple return values
  - really should be full array list, but this way no malloc/free
  - use a “designated initializer” for struct
- Complexity Analysis
  - O() ??
  - what inputs get close to worst case?

why not return arrayList*

```
typedef struct  {
    int count;
    int array[100];
} arrayList;
void printList(arrayList* al) { /* Code not shown */ }
int compare(char *needle, char *stackpart) {
    while (*needle && *stackpart) {
        if (*needle != *stackpart) return 0;
        needle++;
        stackpart++;
    }
    return (*needle == '\0');
}
int strstrGT(char* needle, char* haystack) {
   int hayloc = 0;
   while (*haystack != '\0') {
        if (compare(needle, haystack)) return hayloc;
        haystack++;
        hayloc++;
   }
   return -1;
}
arrayList allLocsF(char* needle, char* haystack) {
    arrayList ret = (arrayList){.count = 0};
    int oloc = 0, nloc;
    while (0 <= (nloc = strstrGT(needle, haystack + oloc))) {
        ret.array[ret.count++] = nloc+oloc;
        oloc = oloc + nloc + 1;
    }
    return ret;
}
int main(int argc, char const *argv[])
{
    arrayList allLocs = allLocsF((char *)argv[1], (char *)argv[2]);
    printList(&allLocs);
    return 0;
}
```

Indexing

- Idea, preprocess the text into an easily used form
  - “bigrams” or trigrams
    - restricting to ASCII there are still 128*128 = 16384 bigrams
    - 2.048 million trigrams

- With real text might do words rather than bigrams/trigrams
  - If working with words then you might build an “Inverted Index”

Bigram index

- On ASCII could start with a table of size 2^^14 (16384)
  - Most locations would be empty
  - Only feasible for ASCII
    - unicode uses 2 bytes per char so would need 2^^30
      - again, mostly empty ..
- Or could do hashing
  - it takes time to hash and then you have to do the whole collision resolution thing …

Bigram Code

- Approach: bigrams encoded as a 128x128 array
  - then Linked List of locations
    - Yes, stylistically this code sucks
    - Yes, I could eliminate a lots of the wasted space
- Preprocess text to be searched to get all bigrams

```
typedef struct llInt {
    int loc;
    struct llInt *next;
} llInt;

llInt *bigrams[128][128];

void preprocess(char * aa) {
    int loc = 0;
    while (*(aa + 1) != '\0') {
        char aaa = *aa;
        char bbb = *(aa + 1);
        llInt *newll = malloc(1 * sizeof(llInt));
        newll->loc = loc;
        if (bigrams[aaa][bbb]!=NULL) {
            newll->next = bigrams[aaa][bbb];
        }
        bigrams[aaa][bbb] = newll;
        aa++;
        loc++;
    }
}
```

More Bigram Code

- to find, first go to the bigram table.
- then check only those locations where there is a bigram
  - best case factor of 16192 speedup
- But need to be doing repeated searches on text

```
int compare(char *needle, char *stackpart) {
    while (*needle && *stackpart)
    {
        if (*needle != *stackpart) {
            return 0;
        }

        needle++;
        stackpart++;
    }
    return (*needle == '\0');
}

llInt* findNeedle(char* needle, char* haystack) {
    llInt *rtn = NULL;
    if (needle == NULL)
        return NULL;
    if ((needle+1)== NULL)
        return NULL;
    char aaa = *needle;
    char bbb = *(needle + 1);
    llInt * candidates = bigrams[aaa][bbb];
    while (candidates!=NULL) {
        if (compare((needle+2), (haystack+candidates->loc+2))) {
            llInt * newLL = malloc(1 * sizeof(llInt));
            newLL->loc = candidates->loc;
            newLL->next = rtn;
            rtn = newLL;
        }
        candidates = candidates->next;
    }
    return rtn;
}
```

Bigrams vs Brute Force

	Brute Force	Average time	Bigrams	average time	marginal time	speedup
reps = 1 needle = 6 haystack=10000000	0.167	0.167	1.903	1.903	–	
reps = 200	6.023	0.030115	2.07	0.01035	0.000835	36.065868
reps = 1000	29.513	0.029513	3.6	0.036	0.001697	17.391278

Boyer-Moore Algorithm

- Rather than preprocess the text to be searched (the haystack) pre-process the thing you are searching for (the needle).
  - see “increment i appropriately”
- Further idea: rather than checking from the start for the needle, check from the end

```
int i = needlelen - 1 ;
while (i < haystacklen) {
        int j = needlelen - 1;
        #ifdef ONCE
        printf("C: %d %d\n", i, j);
        #endif
        while (j >= 0 && (haystack[i] == needle[j]))
        {
            #ifdef ONCE
            printf("M: %d %d\n", i, j);
            #endif
            --i;
            --j;
        }
        if (j < 0) {
            return i+1;
        }
        // increment i appropriately
}
```

B-M
Bad Character Rule

- Idea: when you get to a mismatch between needle and haystack, you can use information about the location of the mismatched chars to get information about how far ahead in in the haystack the next match to need could possibly be.

0	1	2	3	4
a	a	a	a	a

- EX: needle = “aaaaa” haystack=“aaaabaabaa”
- start with comparing n[4] to h[4]
  - No match. But ‘b’ is NOT in needle so the next possible place that needle could possibly match to haystack would be 4 + len(needle);
- So, next check is n[4] and h[9]

0	1	2	3	4	5	6	7	8	9
a	a	a	a	b	a	a	b	a	a

- bad char rule: part 1: for any character not in needle, skip ahead = length of needle.
- Best Case Complexity?

BM
Bad Char Rule — Continued

- what about characters in needle?
- skip = distance of the last occurrence of the character from the end

- Importantly: skip in haystack is from the point at which the mismatch was detected.

```
for (int i=0; i < needlelen-2; i++) {
        delta1[needle[i]] = needlelen-1 - i;
```

Needle

0	1	2	3	4
a	b	a	c	d

delta1

a	b	c	d	e	f
2	3	5	5	5	5

BM
Bad Char Rule — Issue

- suppose
  - needle = nekjq
  - haystack = aaakekjqaa

delta1

n	e	k	j	q	a
4	3	2	5	5	5

- Now align q's and compare
  - work backward all the way to n!=k
    - bad char rule says jump forward 2.
      - Note that this actually leaves you with "negative progress"
    - So now compare q in needle to k in haystack.
    - q!=k so bad char rule says jump forward 2
    - Which bring you back to where you started!!! LOOP. BAD

BM
Suffix Rule 1

- Jump ahead based on
  - whether the beginning of the needle is echoed at the end of the needle
  - how much of the suffix has been matched

- Determine “index of last prefix” by working backwards
- * — see next slide

nekjq

Position	Index of “last prefix”	len-1-pos	Skip
0	5	4	9
1	5	3	8
2	5	2	7
3	5	1	6
4	5	0	5* —— 1

bacaba

Position	Index of “last prefix”	len-1-pos	Skip
0	4	5	9
1	4	4	8
2	4	3	7
3	4	2	6
4	6	1	7* —— 3
5	6	0	6* —— 1

BM
Suffix Rule pt 2

- Since suffix_length may not be unique
  - take the minimum value,

  - ABYXCDBYX

d2: 17
d2: 16
d2: 15
d2: 14
d2: 13
d2: 8
d2: 11
d2: 10
d2: 1

d2: 17
d2: 16
d2: 15
d2: 14
d2: 13
d2: 12
d2: 11
d2: 10
d2: 9

B-M vs Brute Force

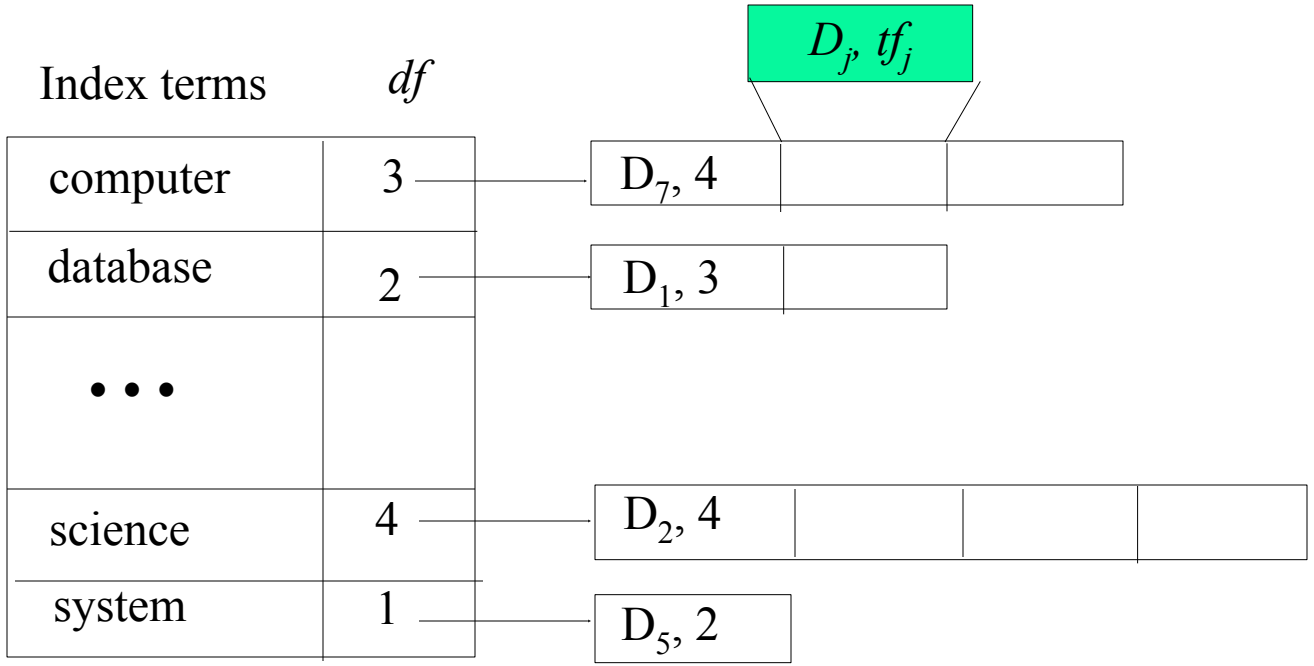
haystack=50,000,000 reps = 100	Boyer Moore	Brute Force
needle=5	11.521	15.353
10	6.414	15.136
20	4.562	
40	3.352	
80	3.237	

- Complexity: O(m+n) if pattern not in text, O(mn) if pattern is in text
- Best Case: m/n

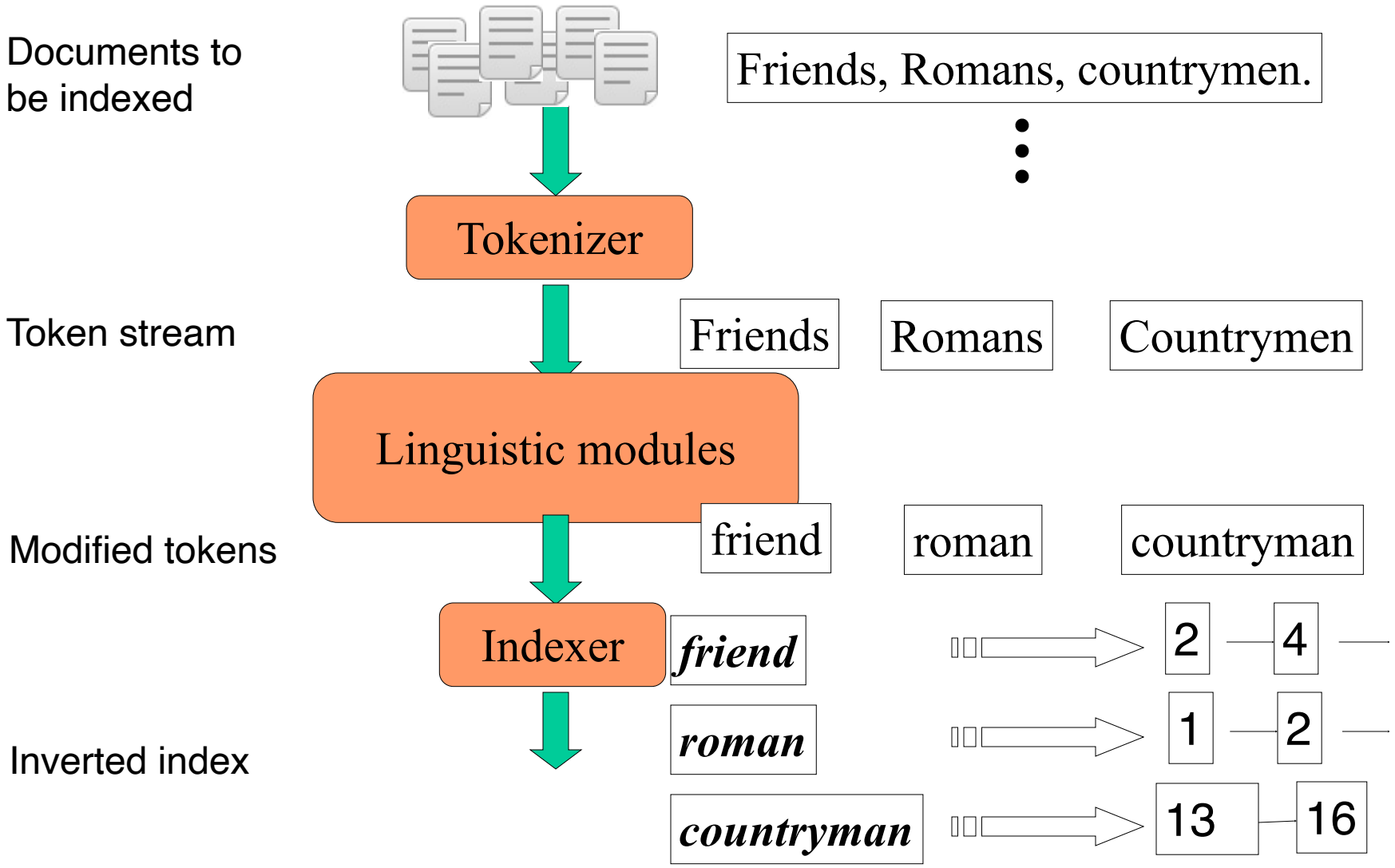
Knuth Morris Pratt

- asymptotically better than B-M
  - in practice B-M usually a better

Inverted Index



Inverted Index Construction



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

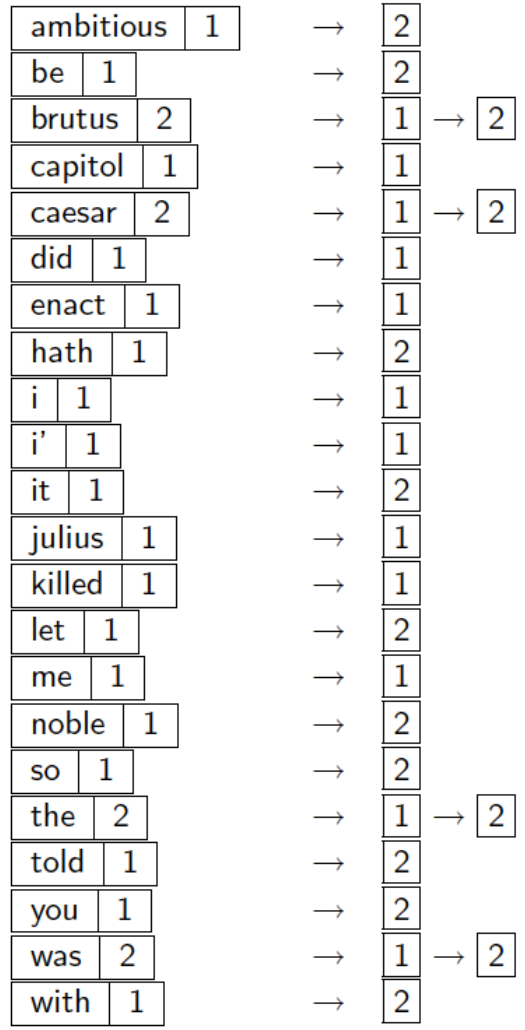
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

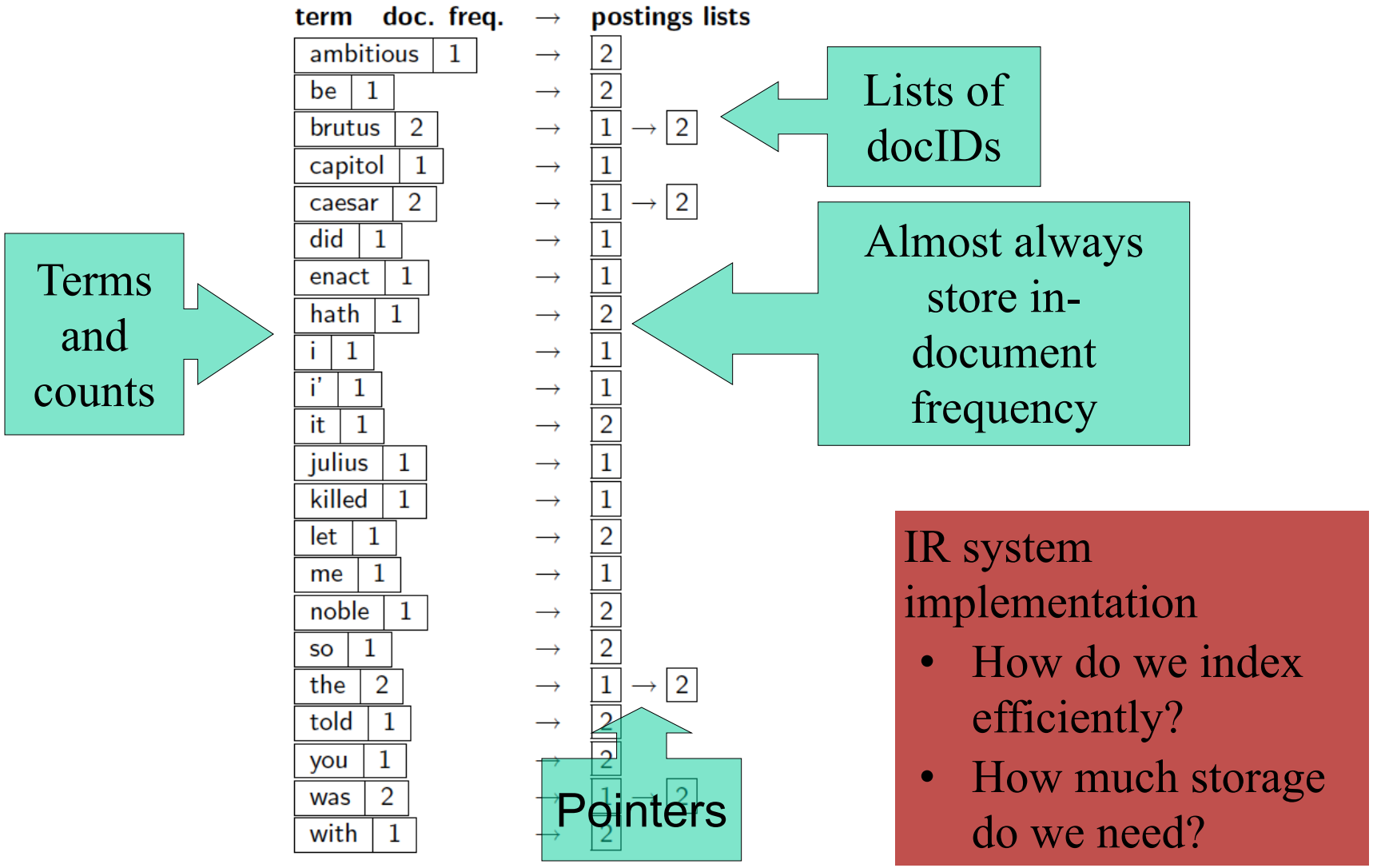
Indexer steps: Dictionary & Postings

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

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



Where do we pay in storage?



IR system implementation

- How do we index efficiently?
- How much storage do we need?