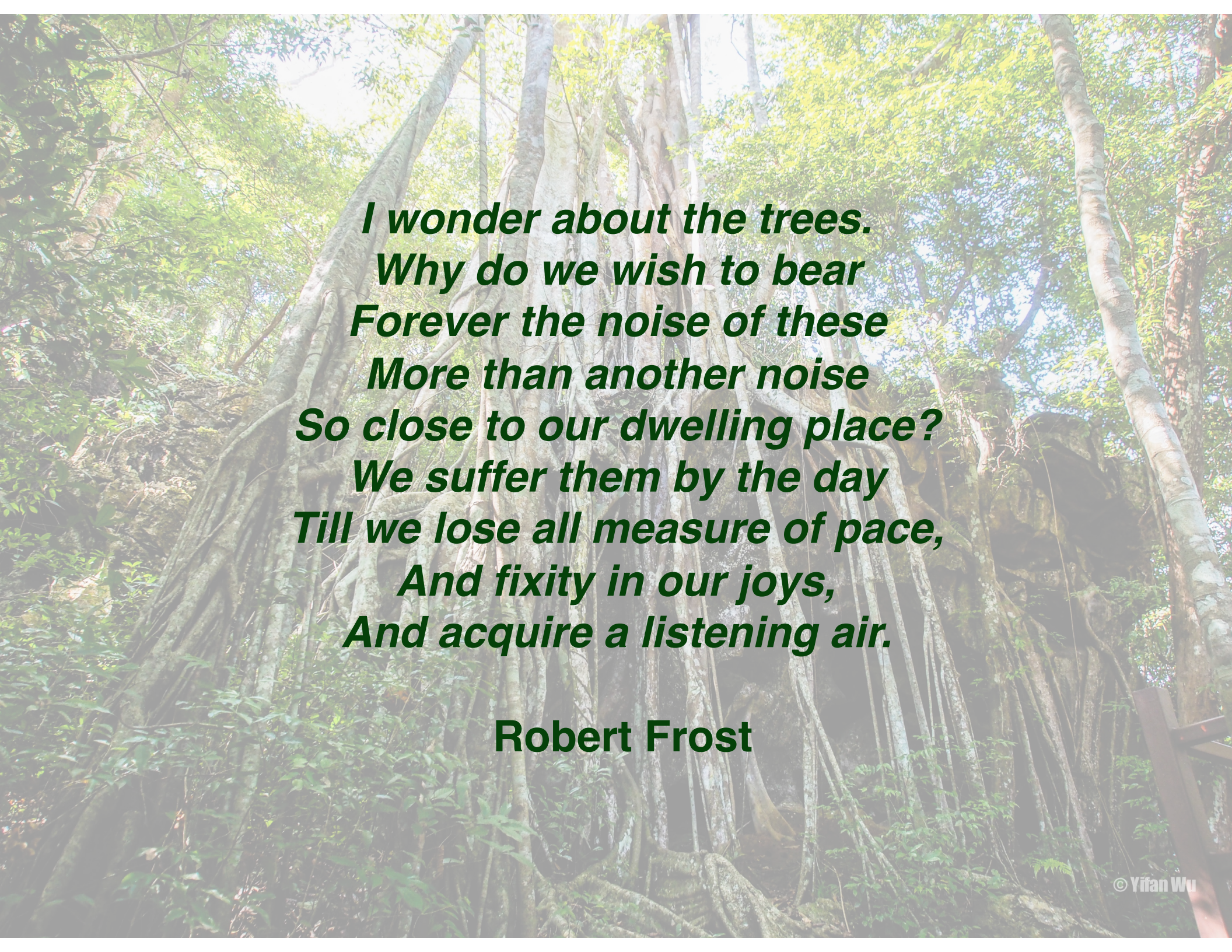

I wonder about the trees.
Why do we wish to bear
Forever the noise of these
More than another noise
So close to our dwelling place?
We suffer them by the day
Till we lose all measure of pace,
And fixity in our joys,
And acquire a listening air.
Robert Frost
© Yifan Wu

Discussion 4

Indices

note: these slides were designed to assist discussion and might not be as informative as notes.

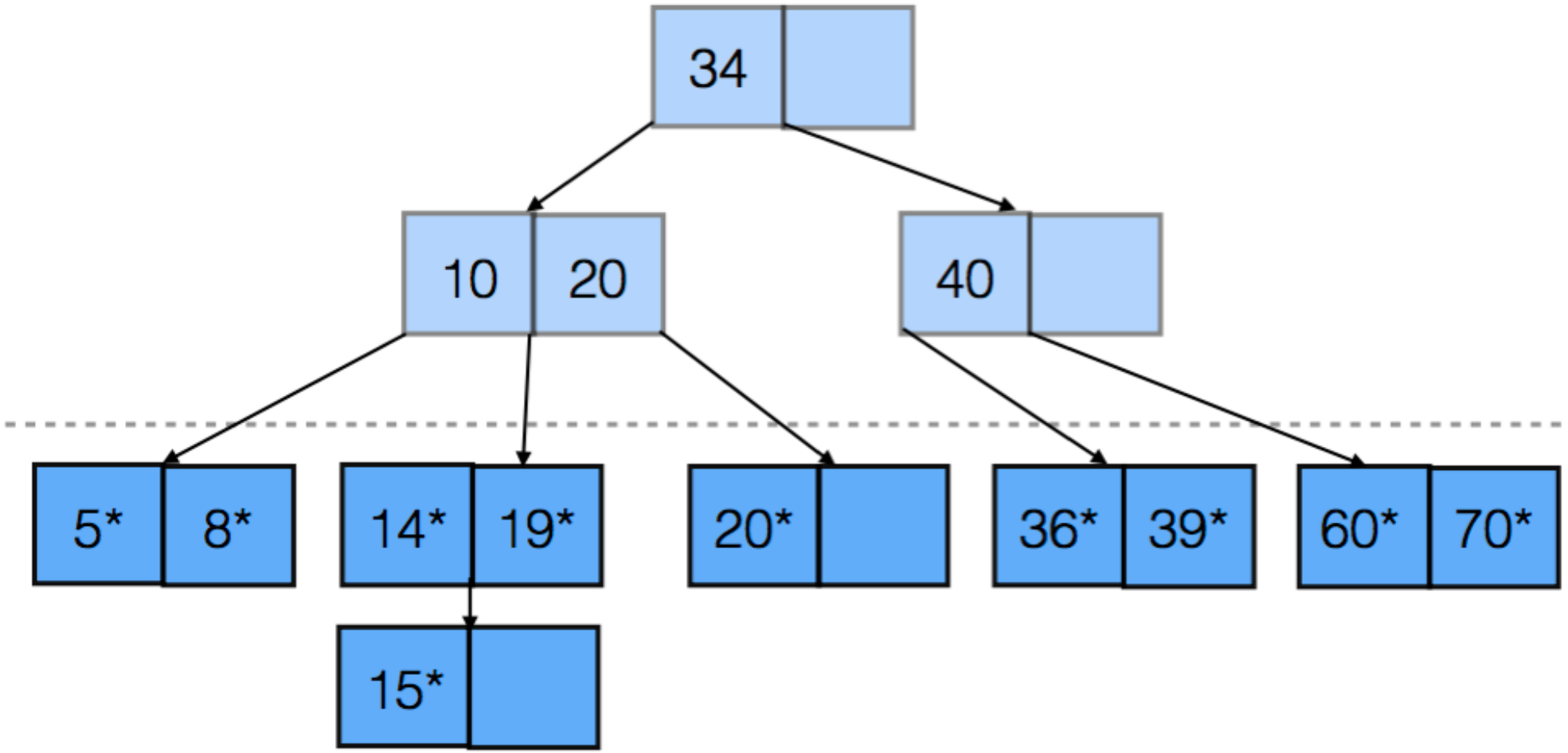
Many pages adapted from
https://github.com/aimichelle/cs186

Types of Indices

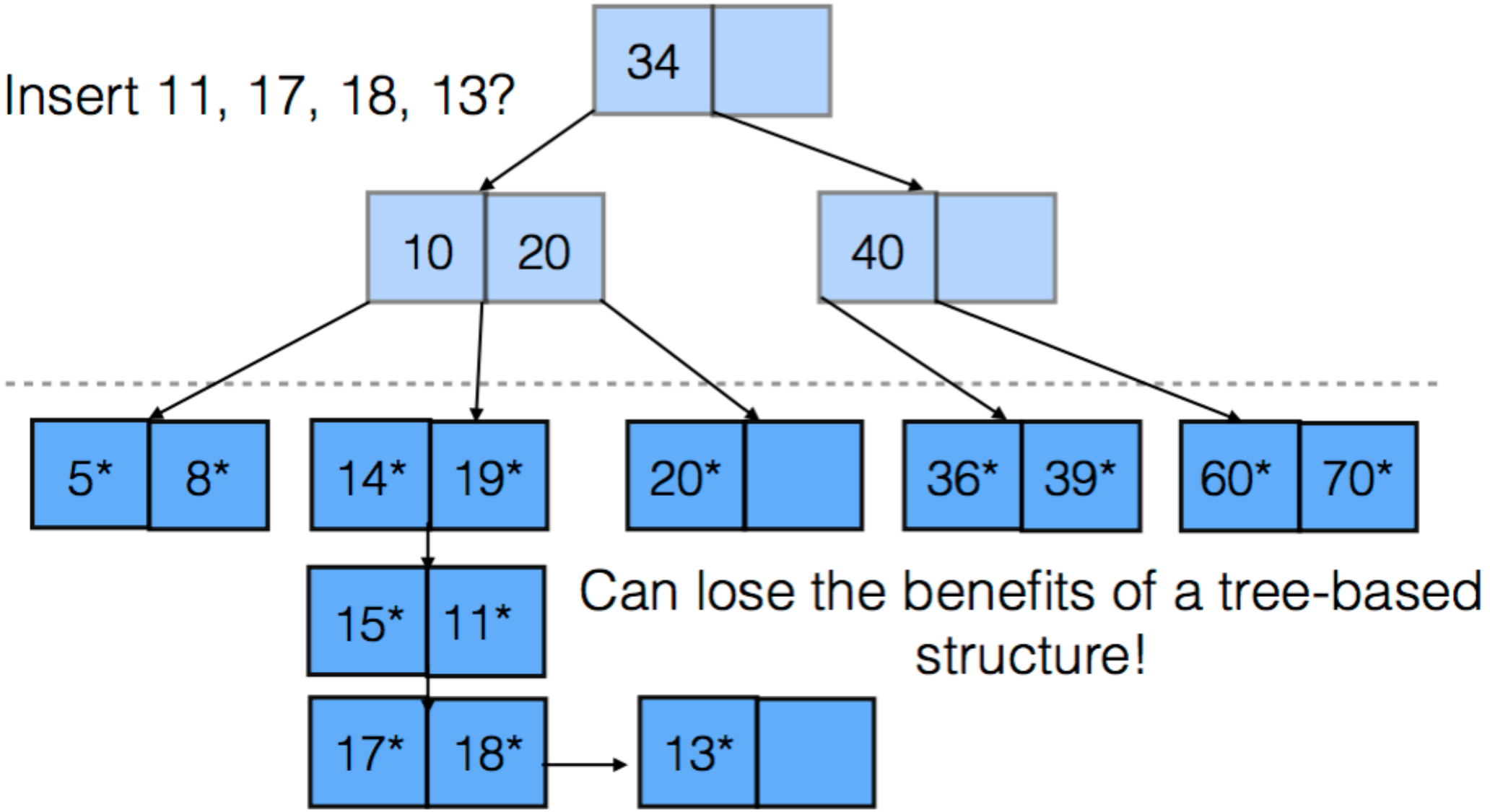
- Hash
- Tree

- Clustered
- Unclustered

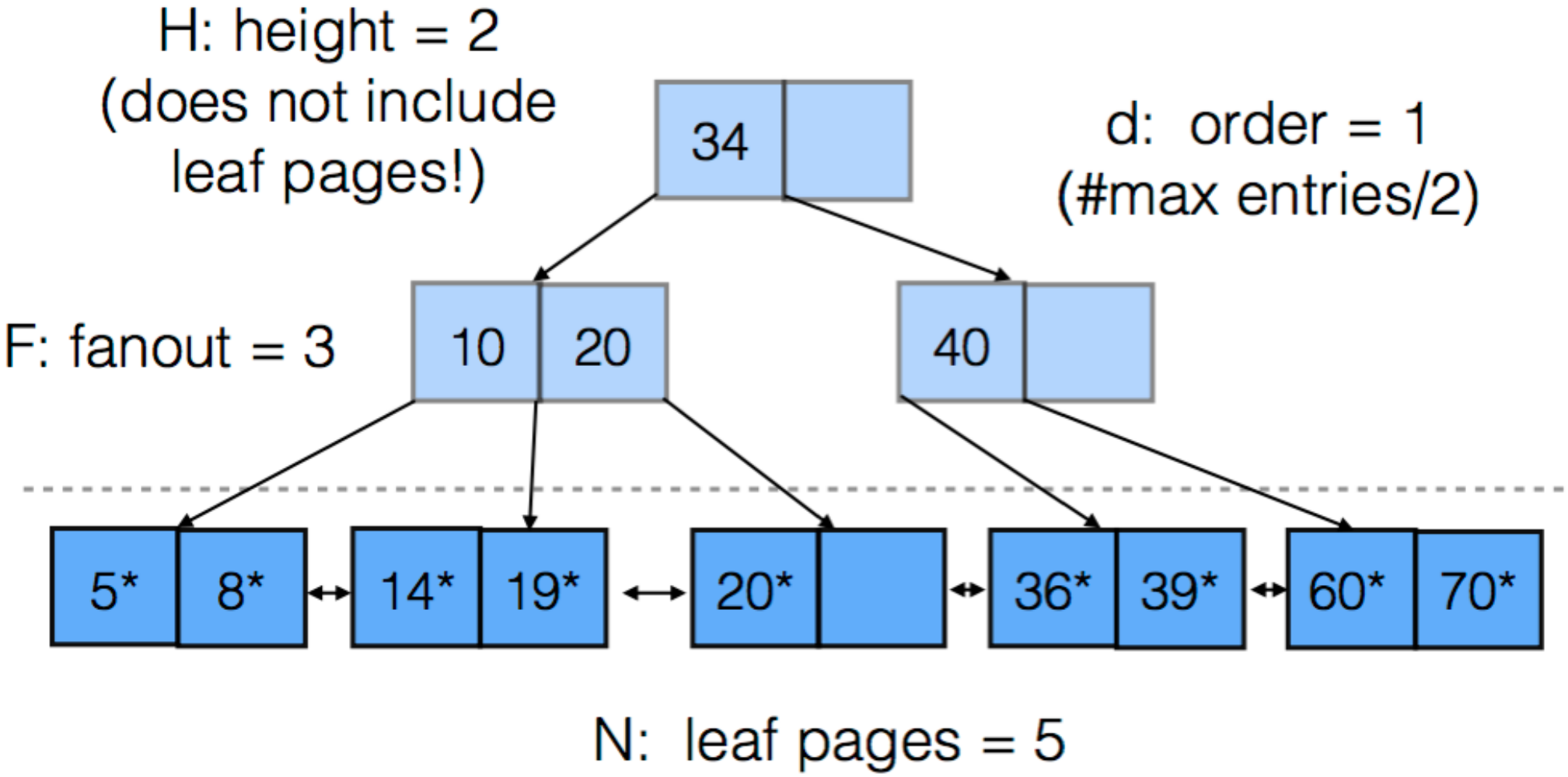
ISAM



ISAM


Insert 11, 17, 18, 13?
34
10
20
40
5*
8*
14*
19*
20*
36*
39*
60*
70*
15*
11*
17*
18*
13*
Can lose the benefits of a tree-based structure!

B+ Trees


H: height = 2
(does not include
leaf pages!)
d: order = 1
(#max entries/2)
F: fanout = 3
34
10
20
40
5*
8*
14*
19*
20*
36*
39*
60*
70*
N: leaf pages = 5

B+ Trees - Insert X

- Find correct leaf L
- Put X in L
  - If not enough space in L:
    - Split L into L and L2
    - **Copy** up middle key to parent
    - If not enough space in parent:
      - Apply algorithm recursively, except **push** up middle key

More Indices?

Making B$^+$-Trees Cache Conscious in Main Memory

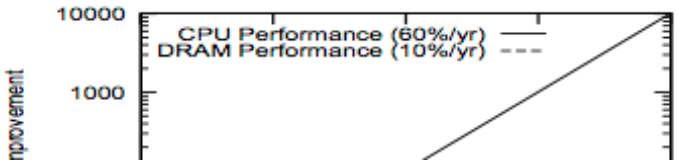
Jun Rao
Columbia University
junr@cs.columbia.edu

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

Abstract

Previous research has shown that cache behavior is important for main memory index structures. Cache conscious index structures such as Cache Sensitive Search Trees (CSS-Trees) perform lookups much faster than binary search and T-Trees. However, CSS-Trees are designed for decision support workloads with relatively static data. Although B$^+$-Trees are more cache conscious than binary search and T-Trees, their utilization of a cache line is low since half of the space is used CSB$^+$-Trees preallocate space for the full node group and thus reduce the split cost. Our performance studies show that CSB$^+$-Trees are useful for a wide range of applications.

1 Introduction



Database Cracking

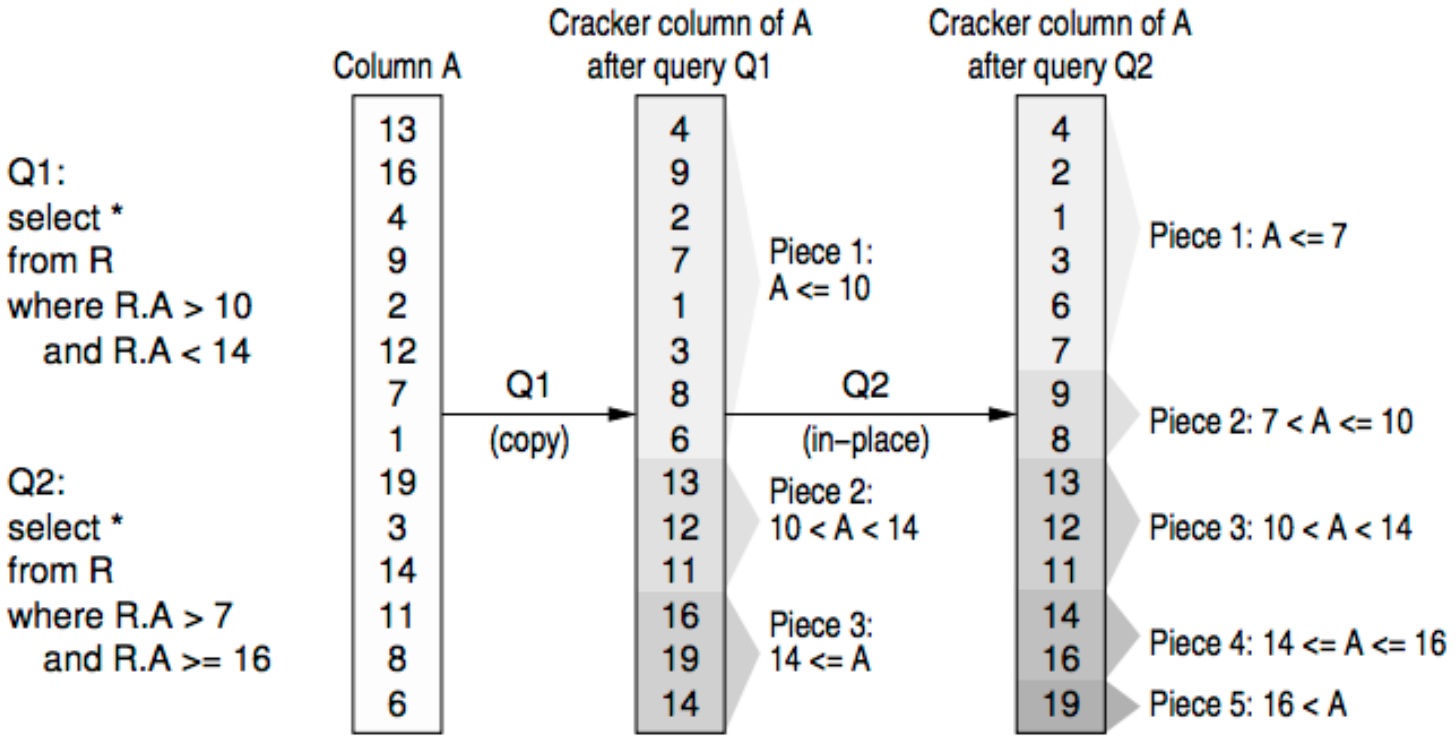


Figure 1: Cracking a column

Idreos et al

homework tips

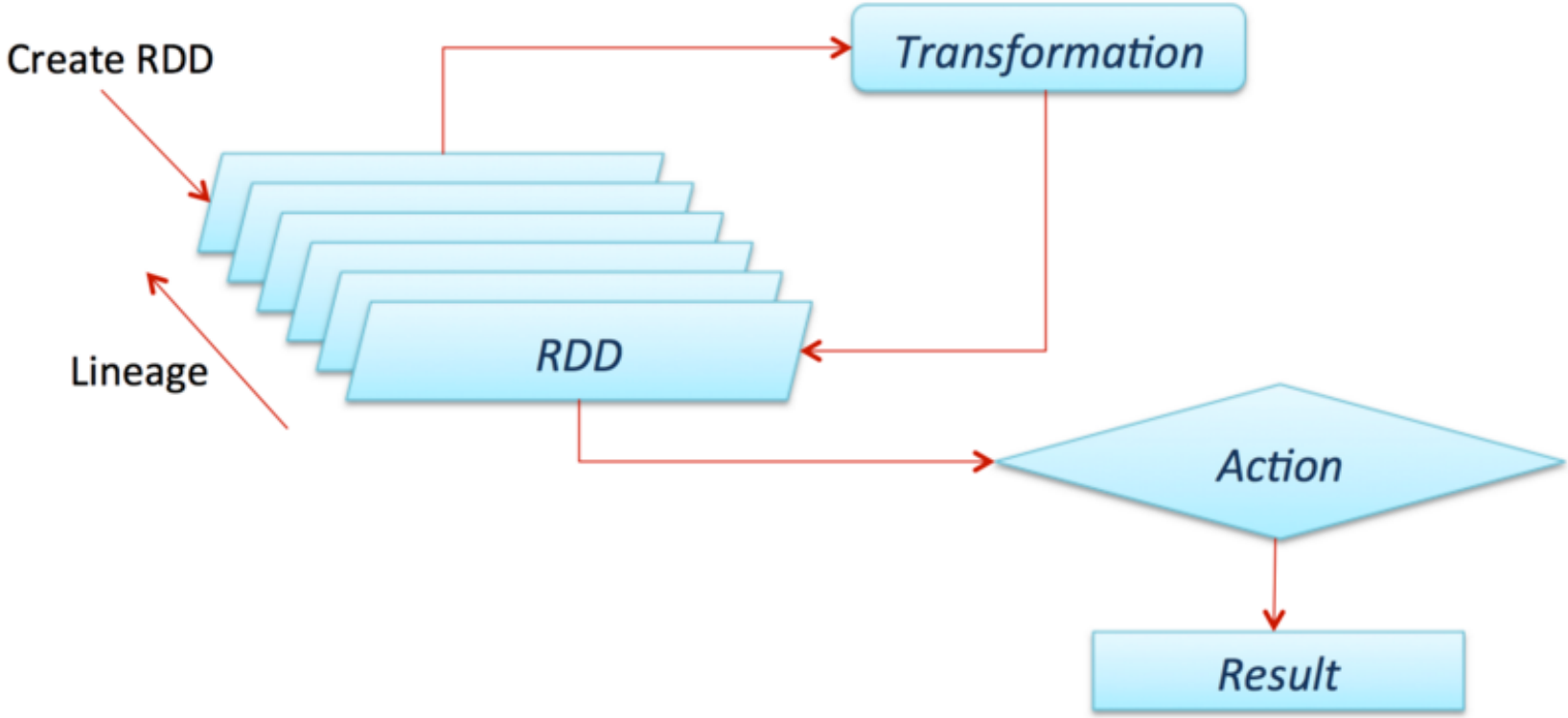
Method Chaining

```
sc = SparkContext(conf=conf)

distFile = sc.textFile('hdfs://{0}:9000/tmp/enron/*/*.txt'.format(HDFS_MASTER))

nonempty_lines = distFile.filter(lambda x: len(x) > 0)
print 'Nonempty lines', nonempty_lines.count()

words = nonempty_lines.flatMap(lambda x: x.split(' '))

wordcounts = words.map(lambda x: (x, 1)) \
                  .reduceByKey(lambda x, y: x+y) \
                  .map(lambda x: (x[1], x[0])).sortByKey(False)
```

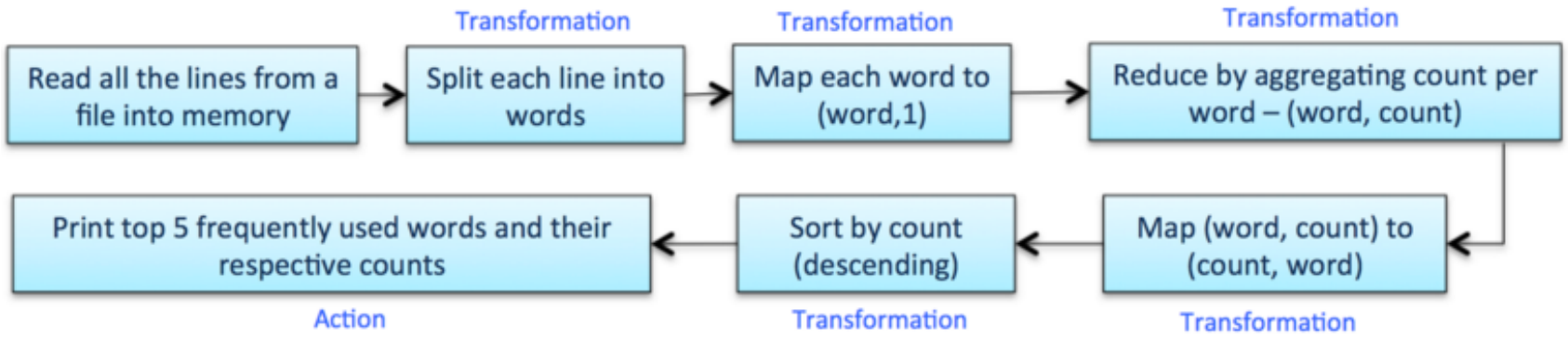
http://docs.continuum.io/anaconda-cluster/examples/spark-wordcount

Spark is LAZY

- lazy evaluation is key for performance, why?
  - the system could optimize based on all the tasks (and could piggy back functions to run together to avoid multiple iterations)
  - also no overwrite means it will be very expensive to materialize every change



Let's understand this conceptually by using the following example: Say we want to find the 5 most commonly used words in a text file. A possible solution is depicted in Figure 10.



https://dzone.com/refcardz/apache-spark

worksheet