Indexes Part 2 and Query Execution

Instructor: Matei Zaharia

cs245.stanford.edu

From Last Time: Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing

Conventional Indexes

Pros:

- Simple
- Index is sequential file (good for scans or binary search)

Cons:

- Inserts expensive, and/or
- Lose sequentiality & balance

B-Trees

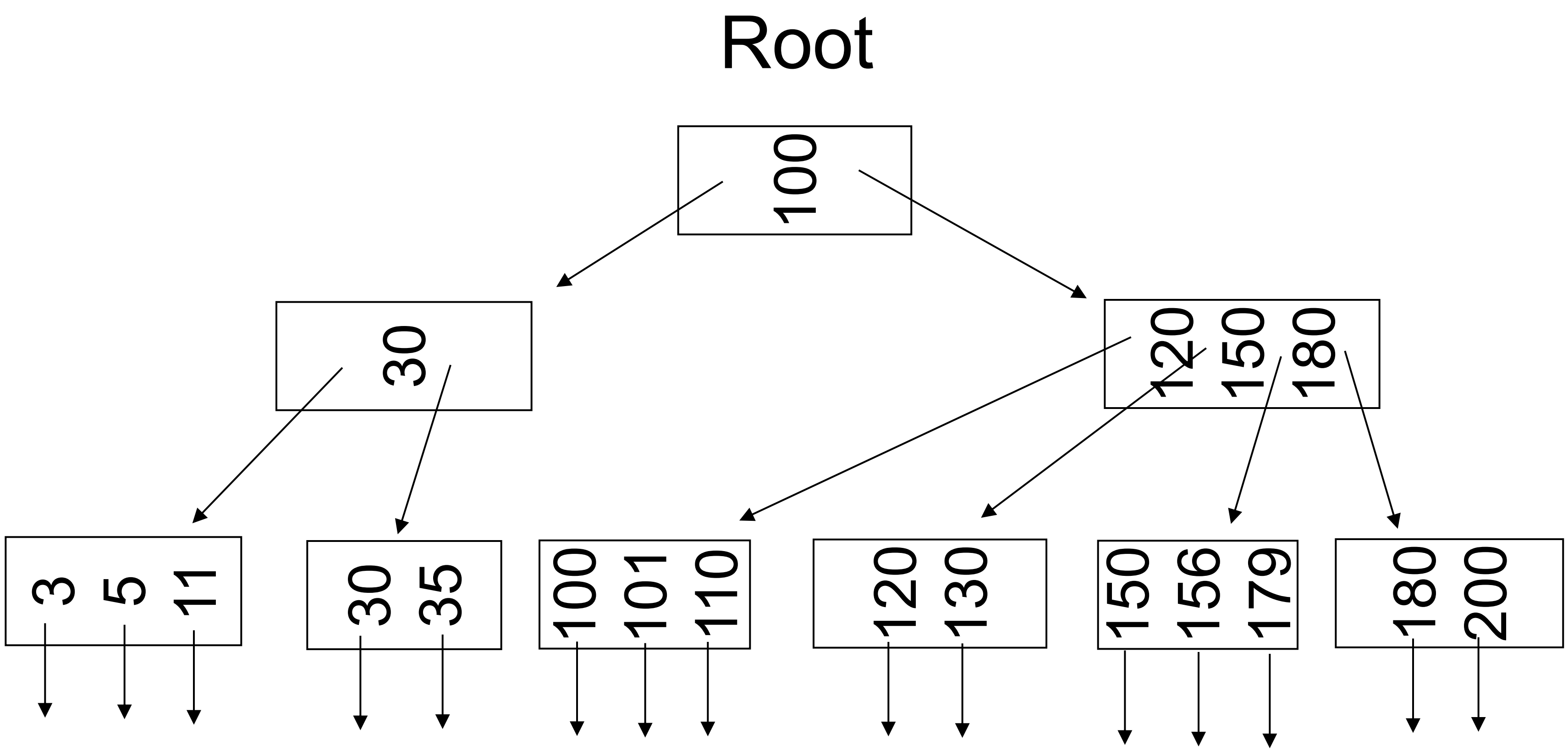
Another type of index

» Give up on sequentiality of index

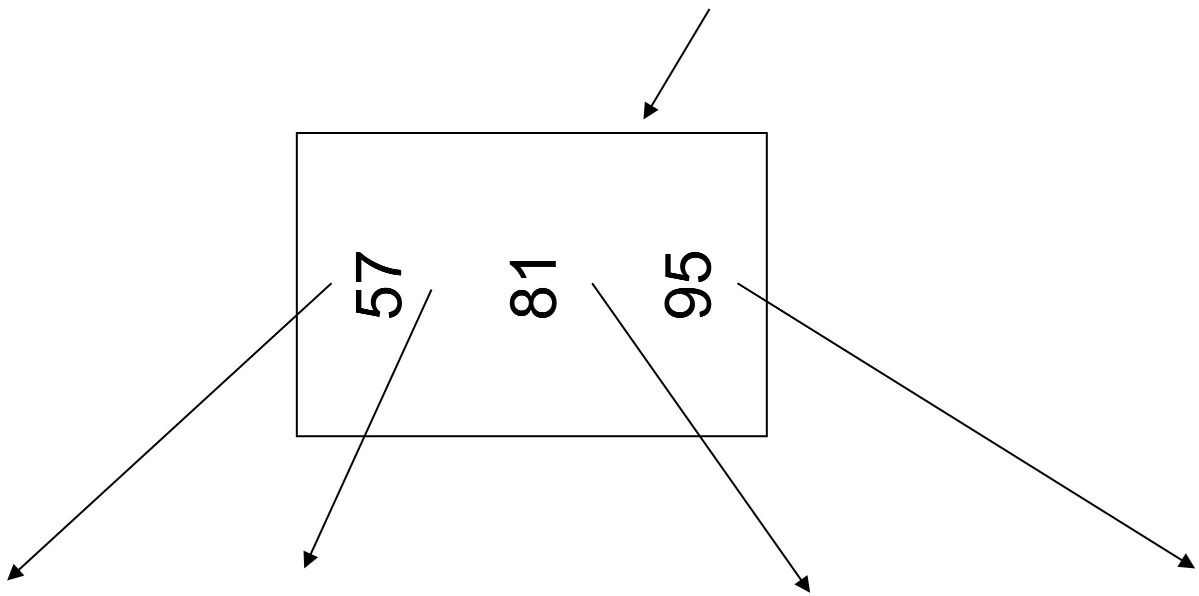
» Try to get “balance”

Note: the exact data structure we’ll look at is a **B+ tree**, but plain old “B-trees” are similar

B+ Tree Example (n = 3)

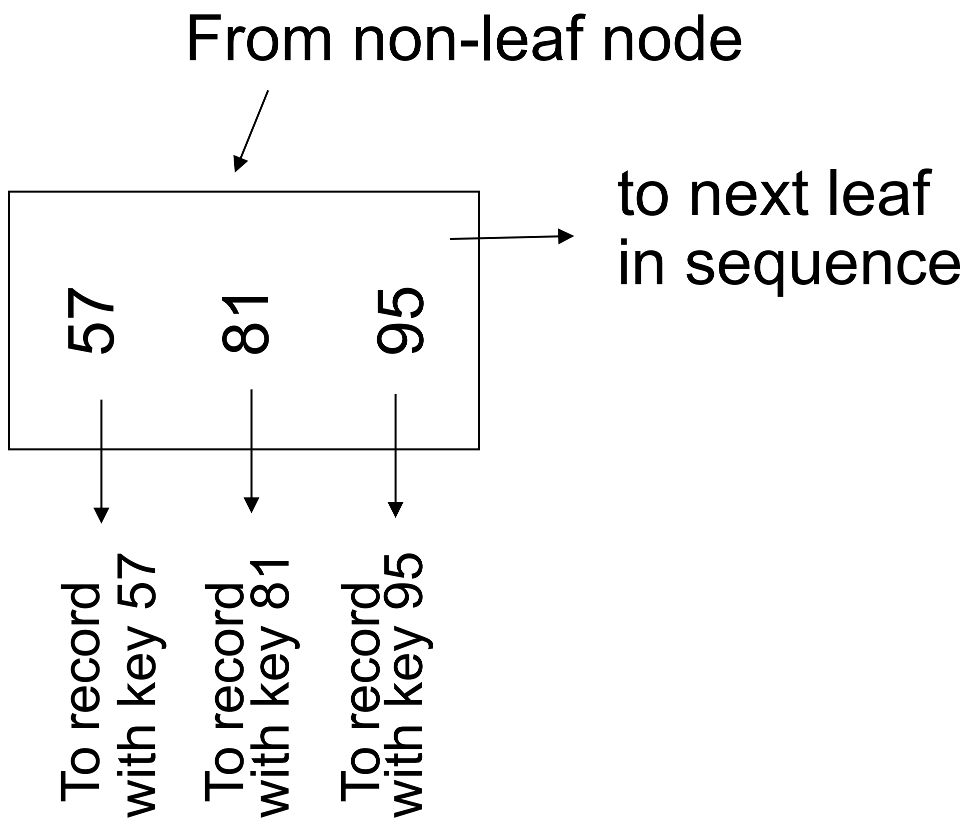


Sample Non-Leaf



to keys	to keys	to keys	to keys
< 57	$57 \leq k < 81$	$81 \leq k < 95$	≥ 95

Sample Leaf Node



Size of Nodes on Disk

$\begin{cases} \text{n + 1 pointers} \\ \text{n keys} \end{cases}$

(Fixed size nodes)

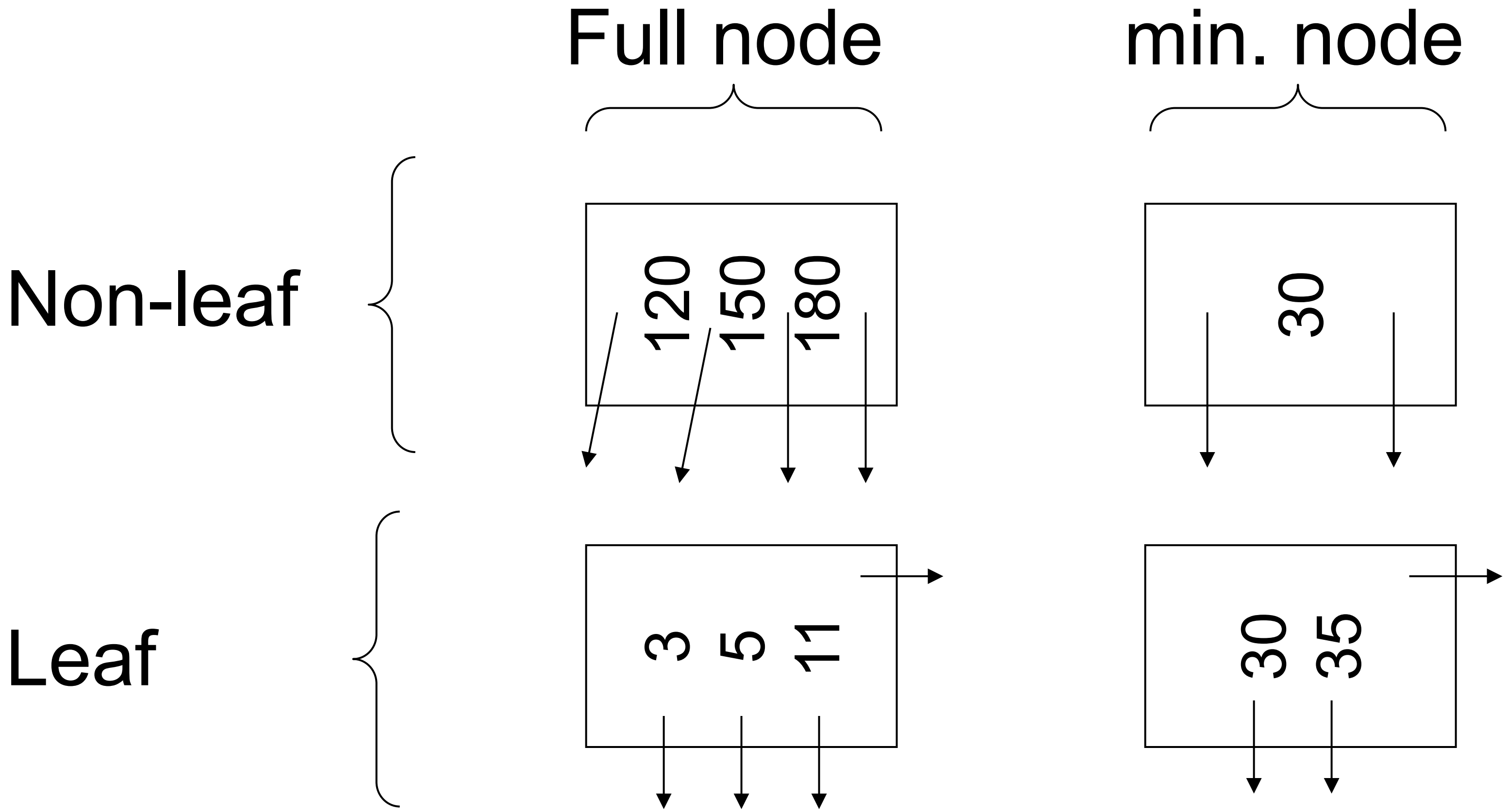
Don’t Want Nodes to be Too Empty

Use at least

Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

Example: n = 3



B+ Tree Rules (tree of order n)

1. All leaves are at same lowest level (balanced tree)

2. Pointers in leaves point to records, except for “sequence pointer”

B+ Tree Rules (tree of order n)

(3) Number of pointers/keys for B+ tree:

	Max ptrs	Max keys	Min ptrs→data	Min keys
Non-leaf (non-root)	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil$-1
Leaf (non-root)	n+1	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	2*	1

* When there is only one record in the B+ tree, min pointers in the root is 1 (the other pointers are null)

Insertion Into B+ Tree

(a) simple case: have space in leaf

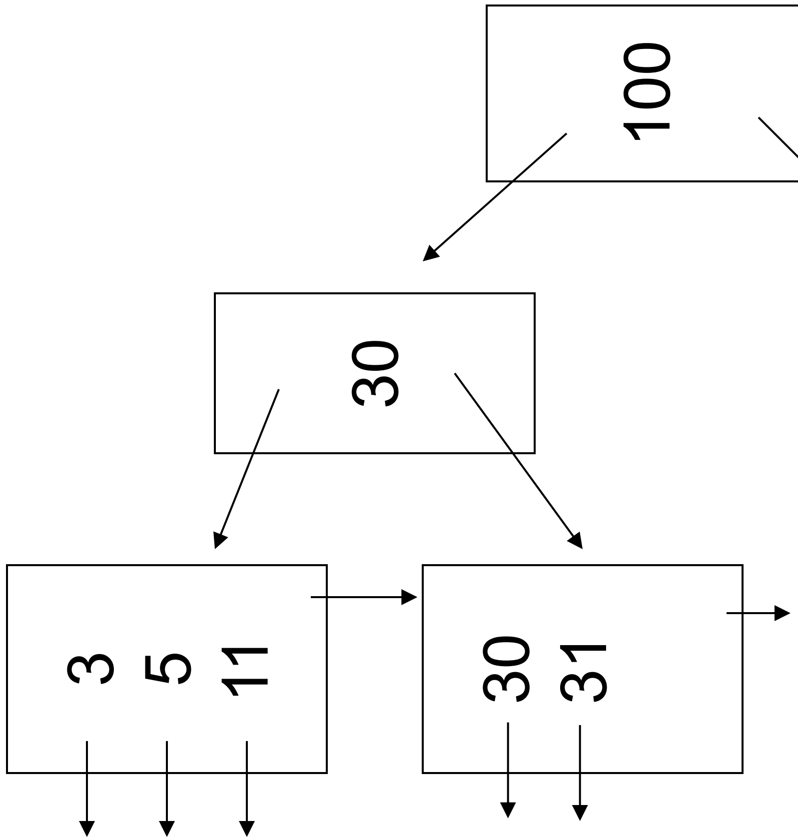
(b) leaf overflow

(c) non-leaf overflow

(d) new root

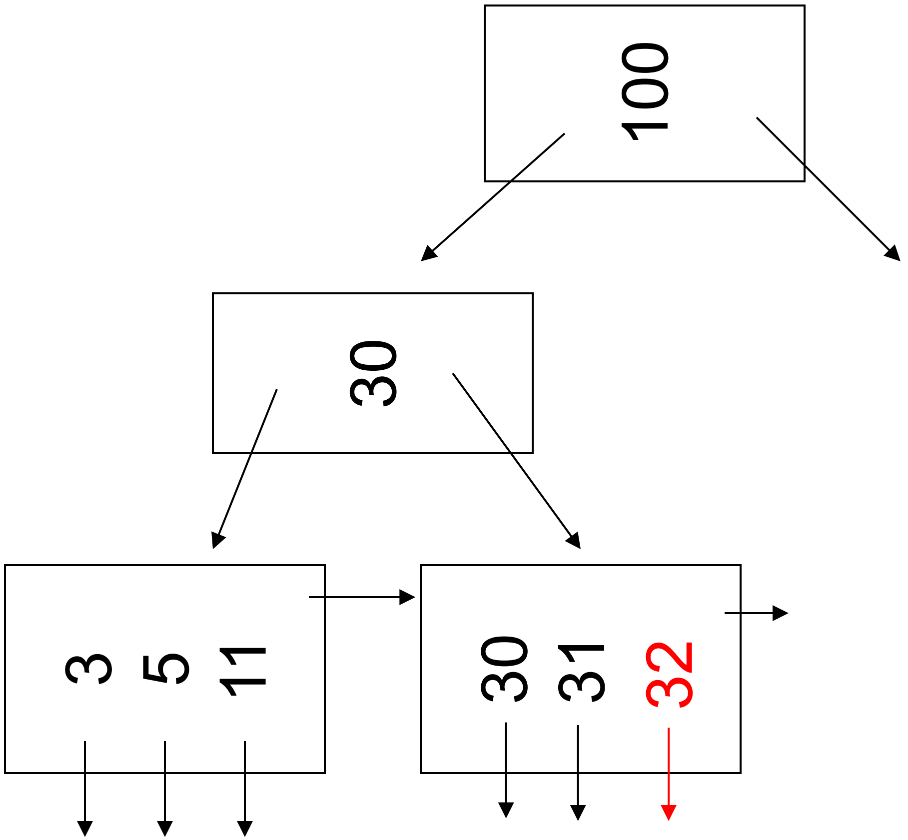
(a) Insert key = 32

n=3



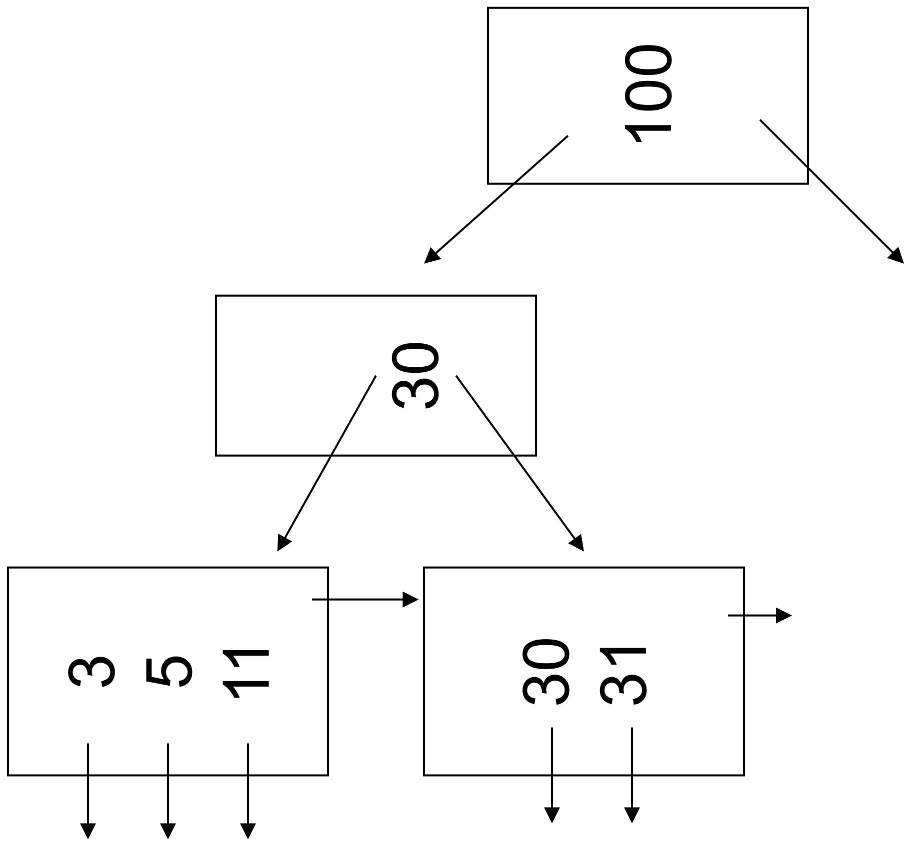
(a) Insert key = 32

n=3



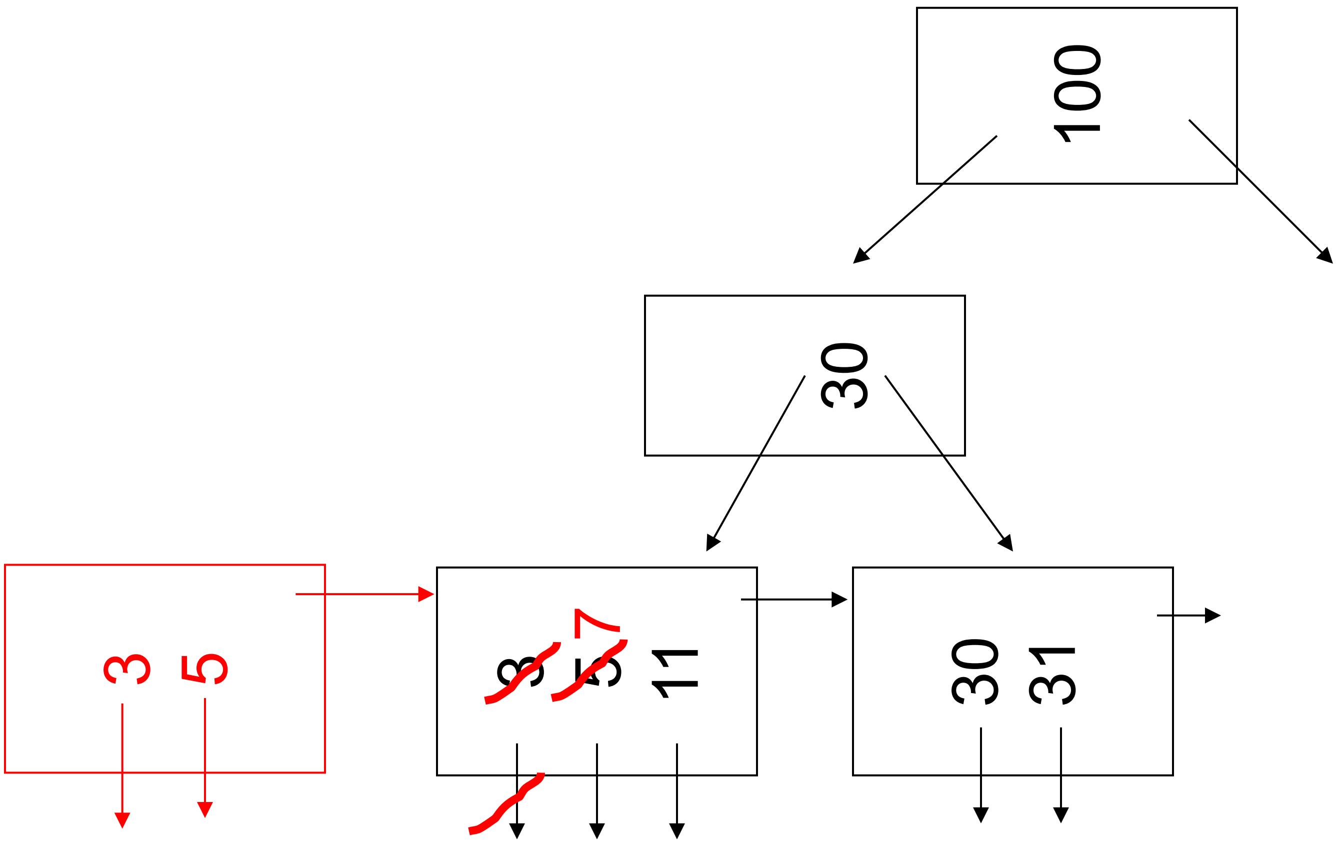
(b) Insert key = 7

n=3



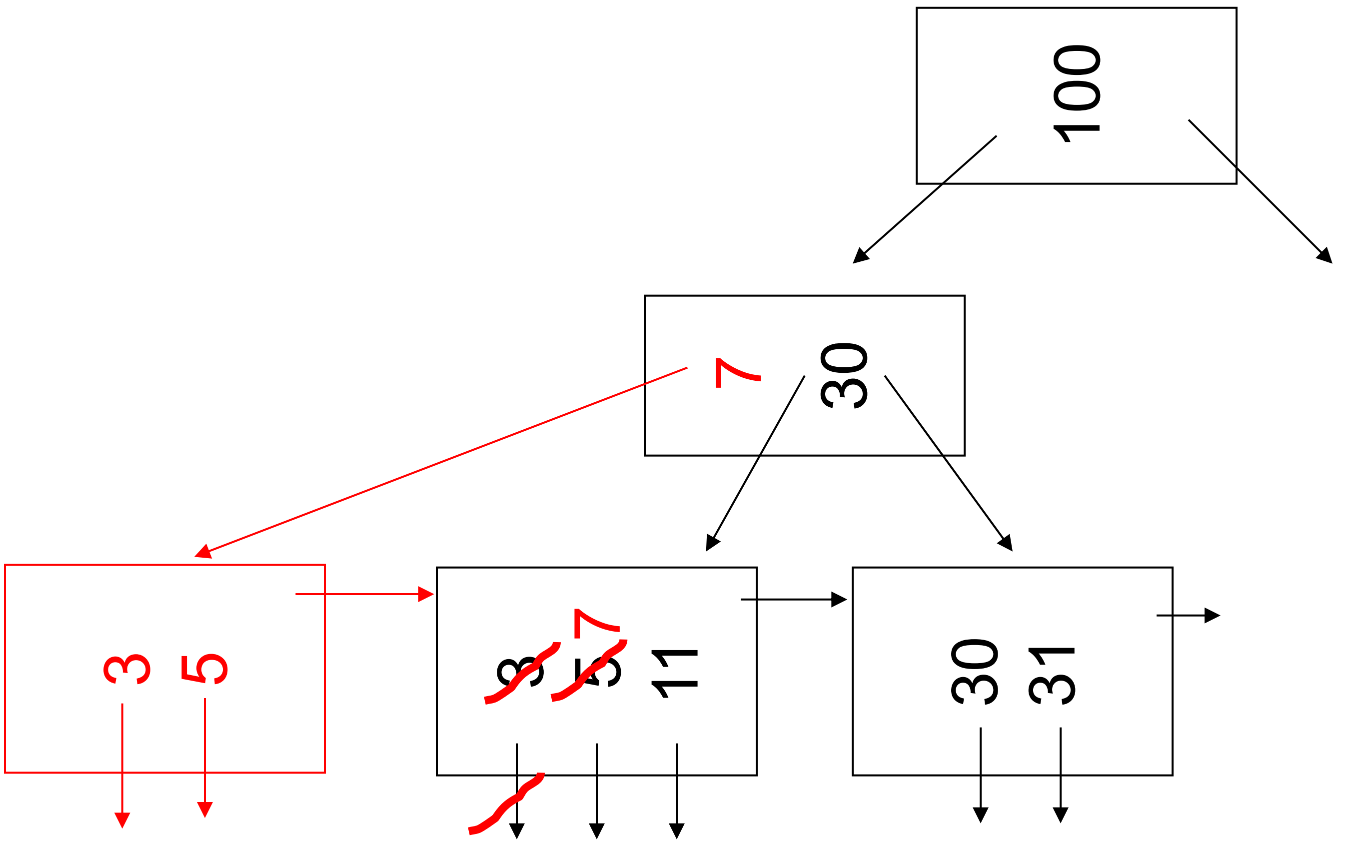
(b) Insert key = 7

n=3



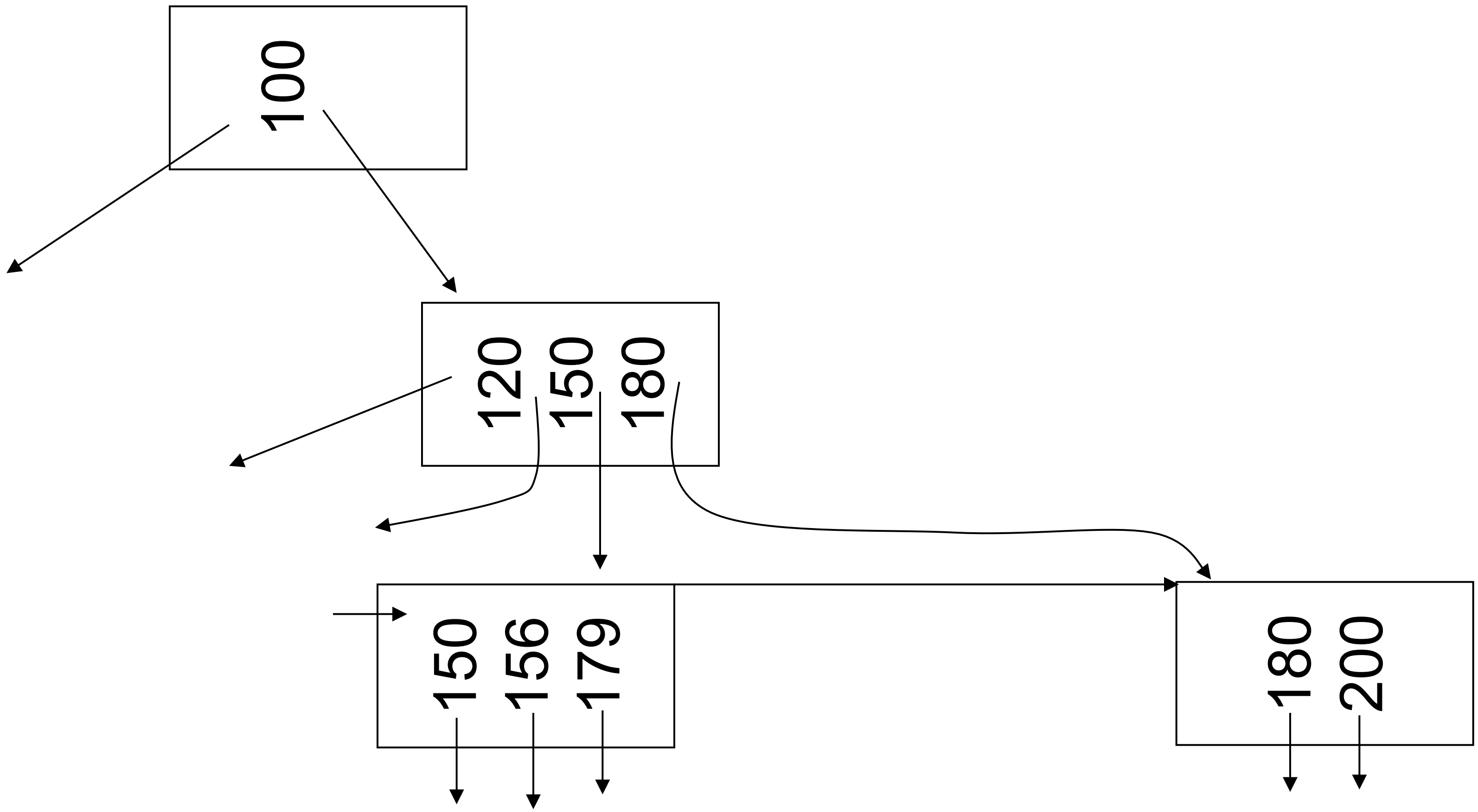
(a) Insert key = 7

n=3



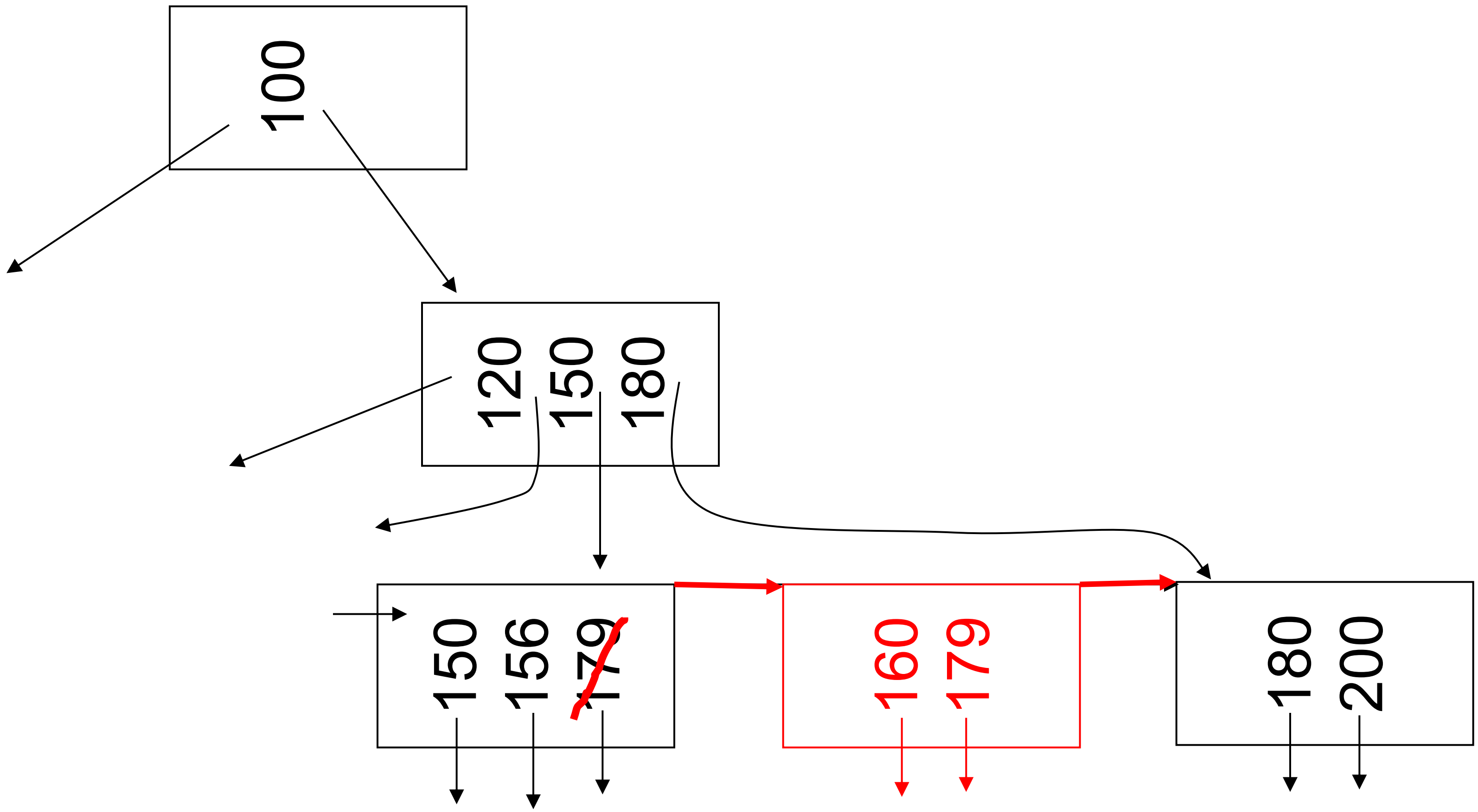
(c) Insert key = 160

n=3



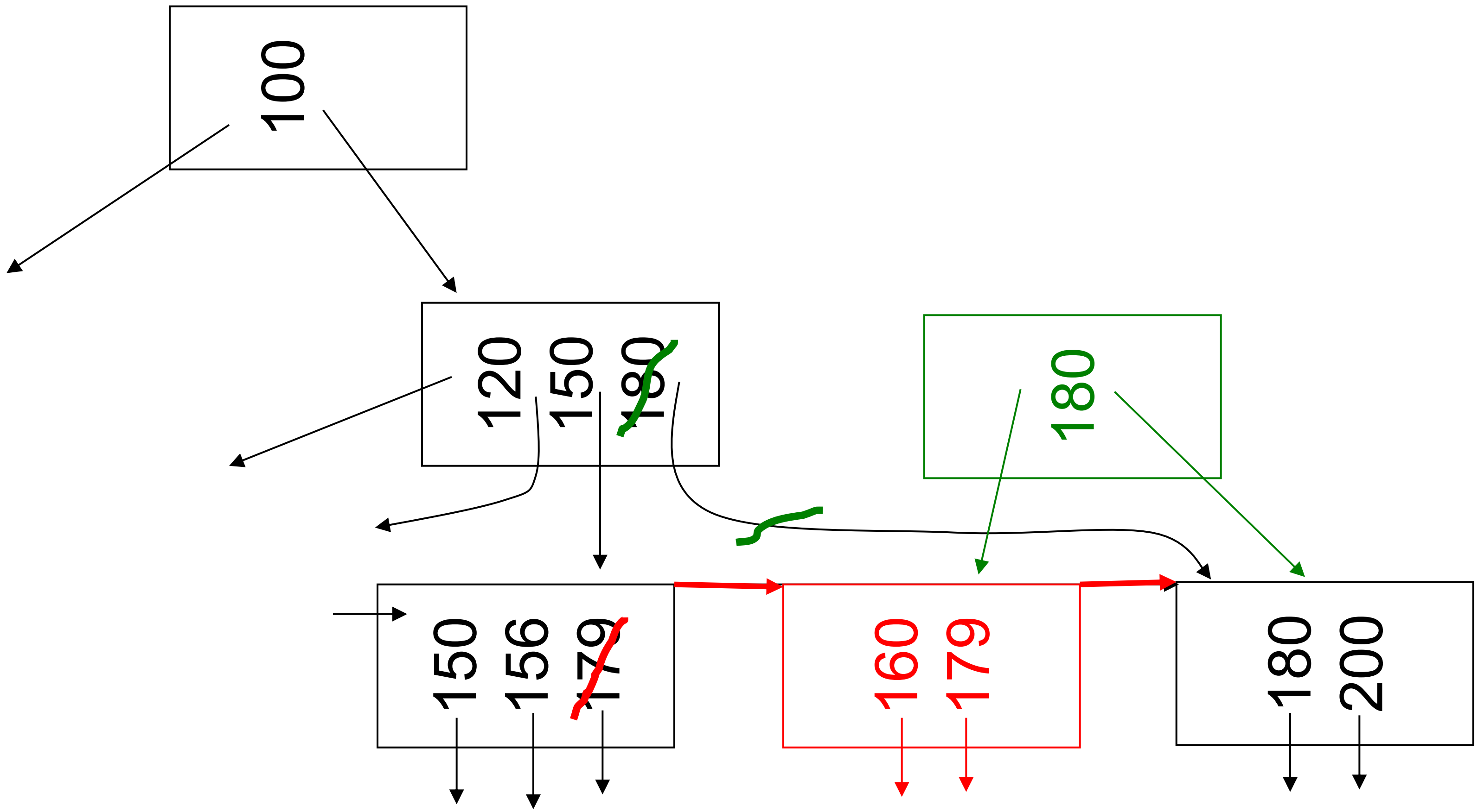
(c) Insert key = 160

n=3



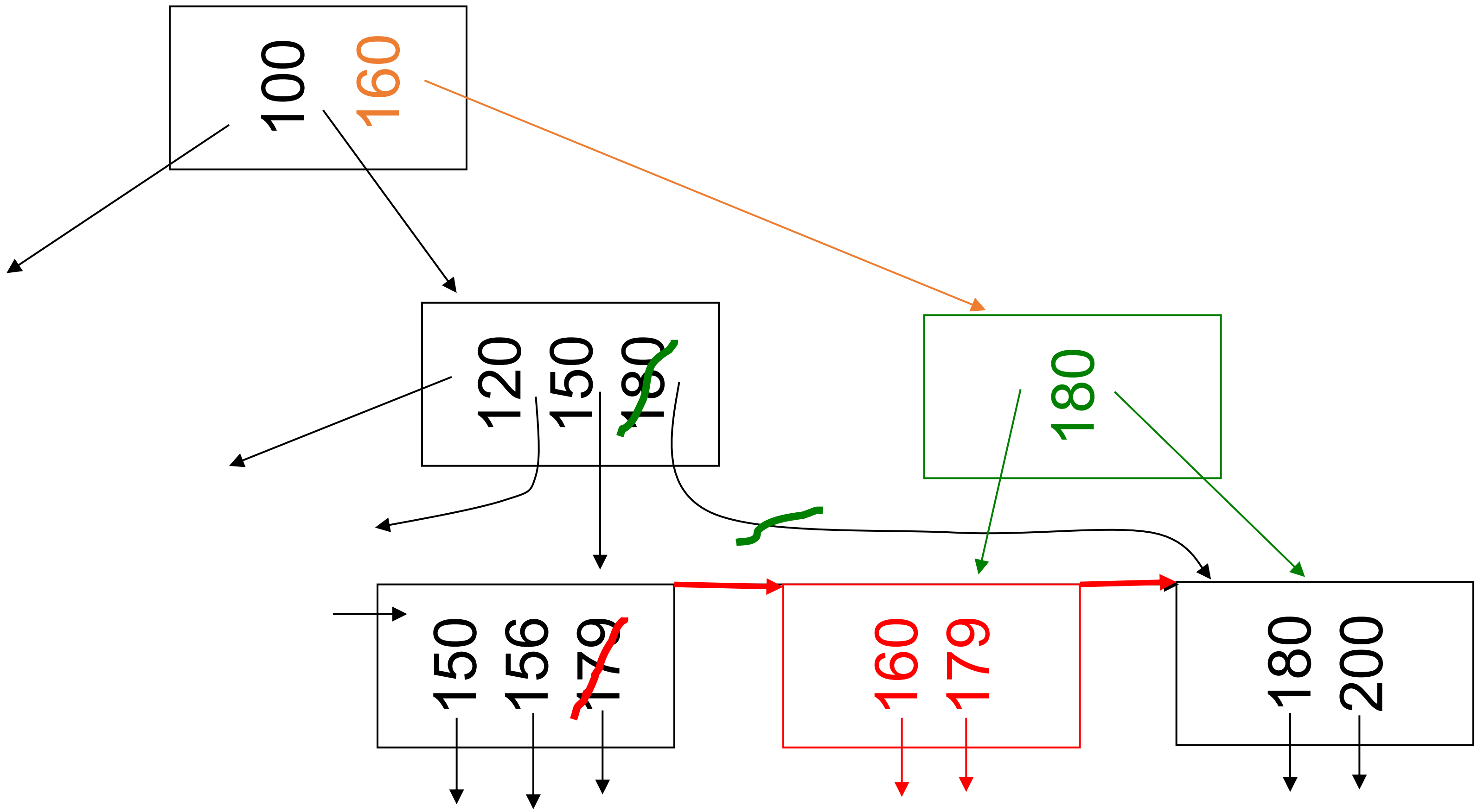
(c) Insert key = 160

n=3



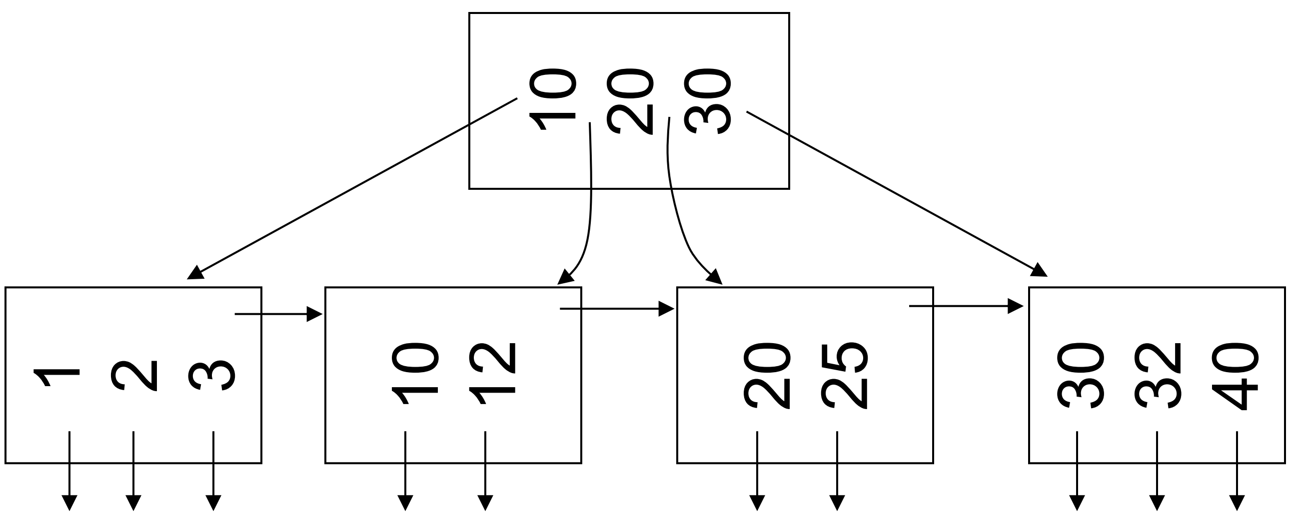
(c) Insert key = 160

n=3



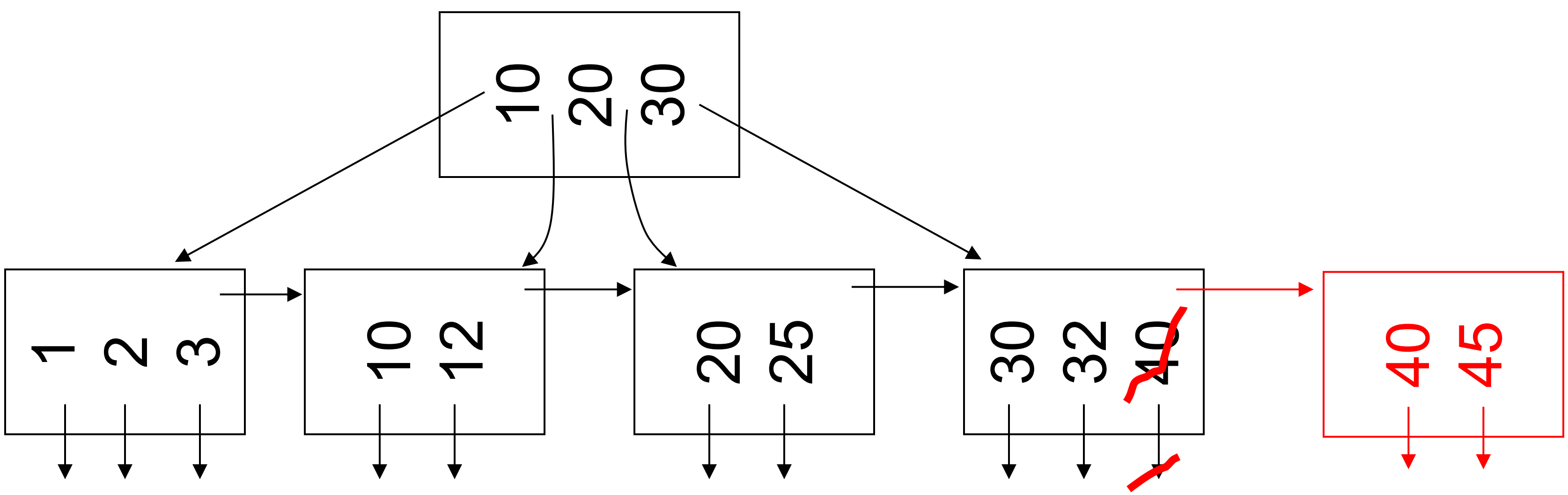
(d) New root, insert 45

n=3



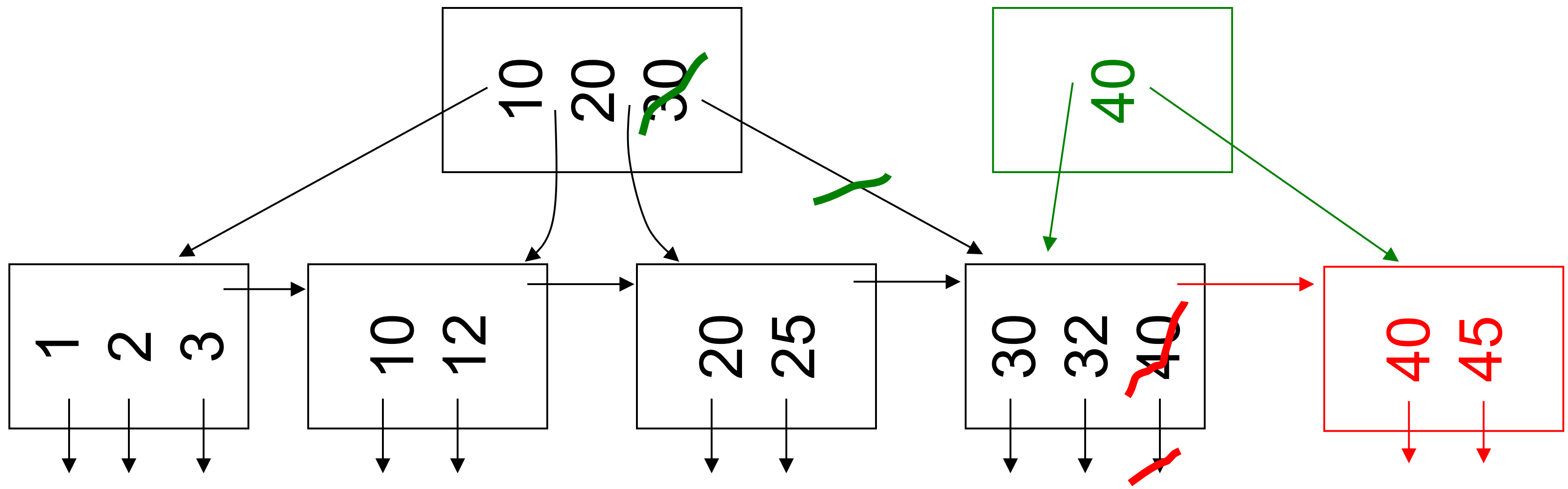
(d) New root, insert 45

n=3



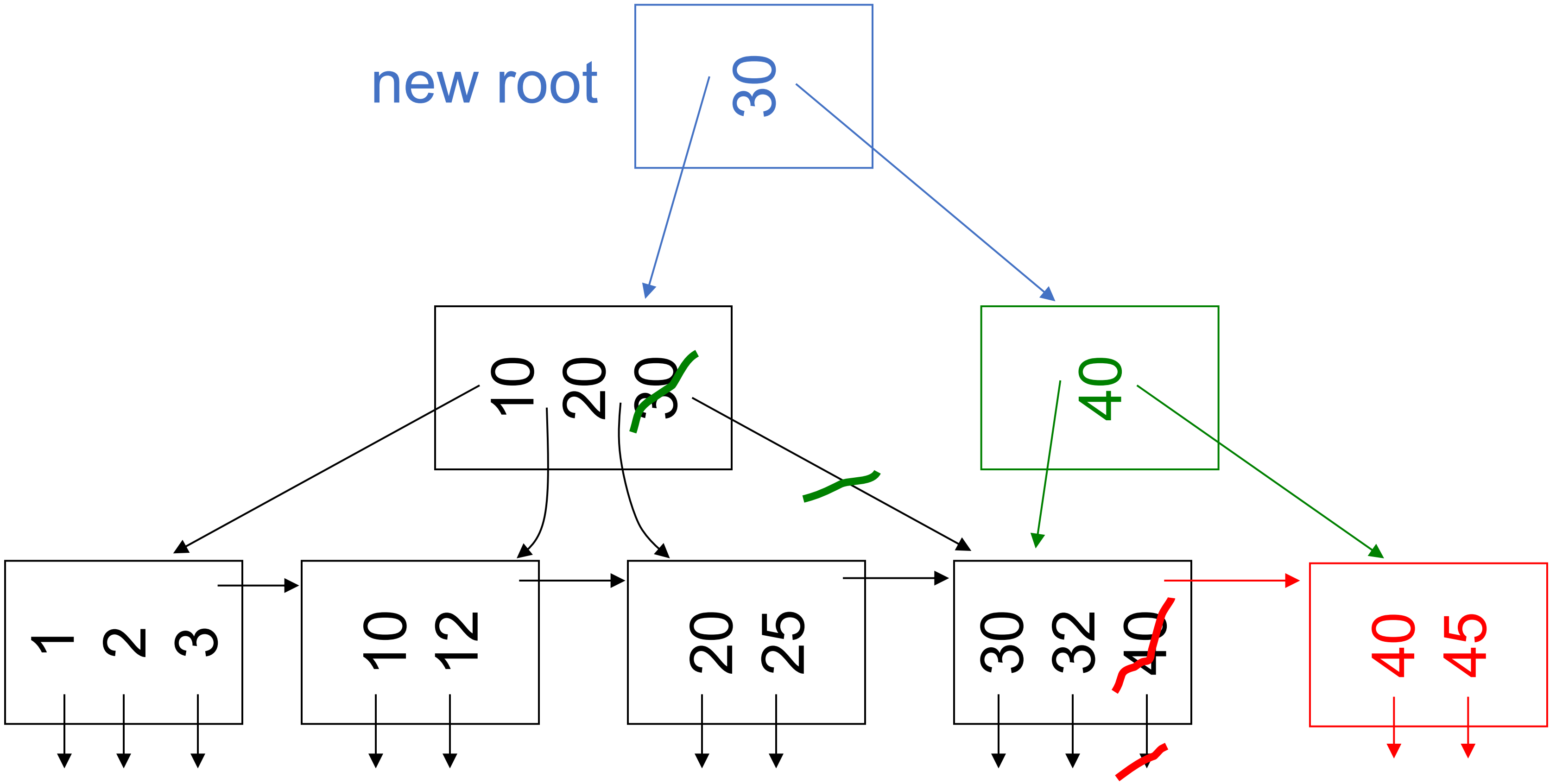
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3



Deletion from B+ Tree

(a) Simple case: no example

(b) Coalesce with neighbor (sibling)

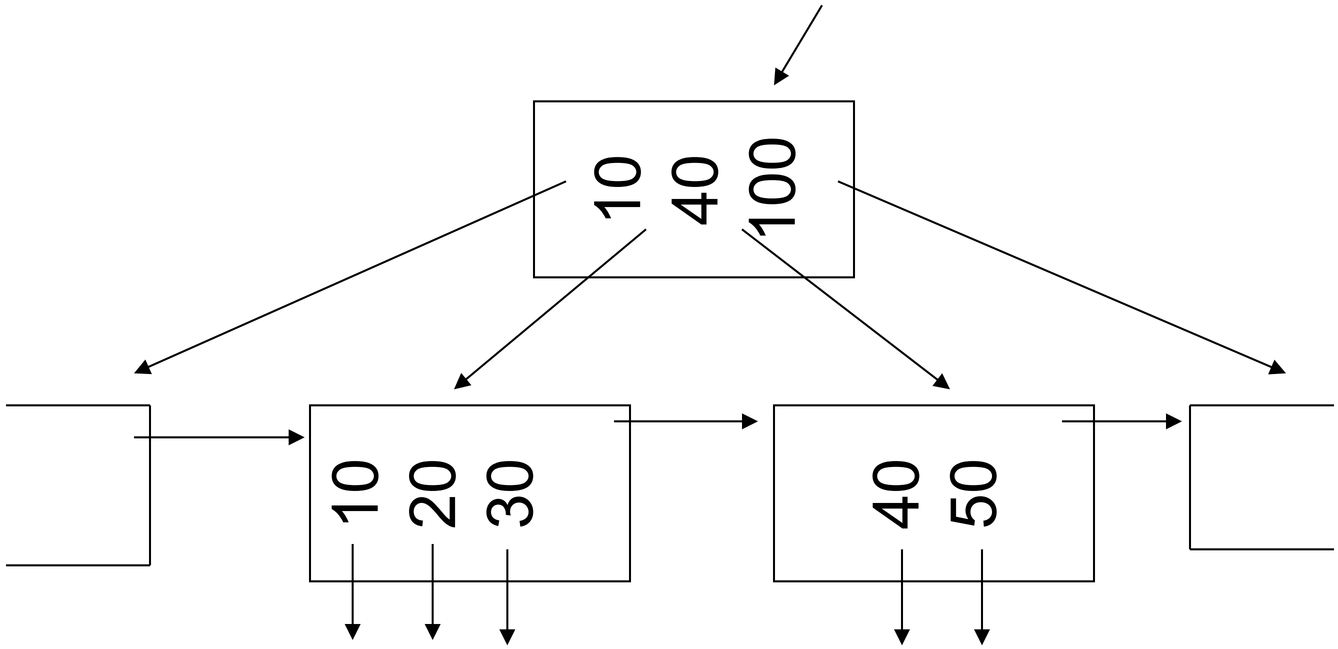
(c) Re-distribute keys

(d) Cases (b) or (c) at non-leaf

(b) Coalesce with sibling

» Delete 50

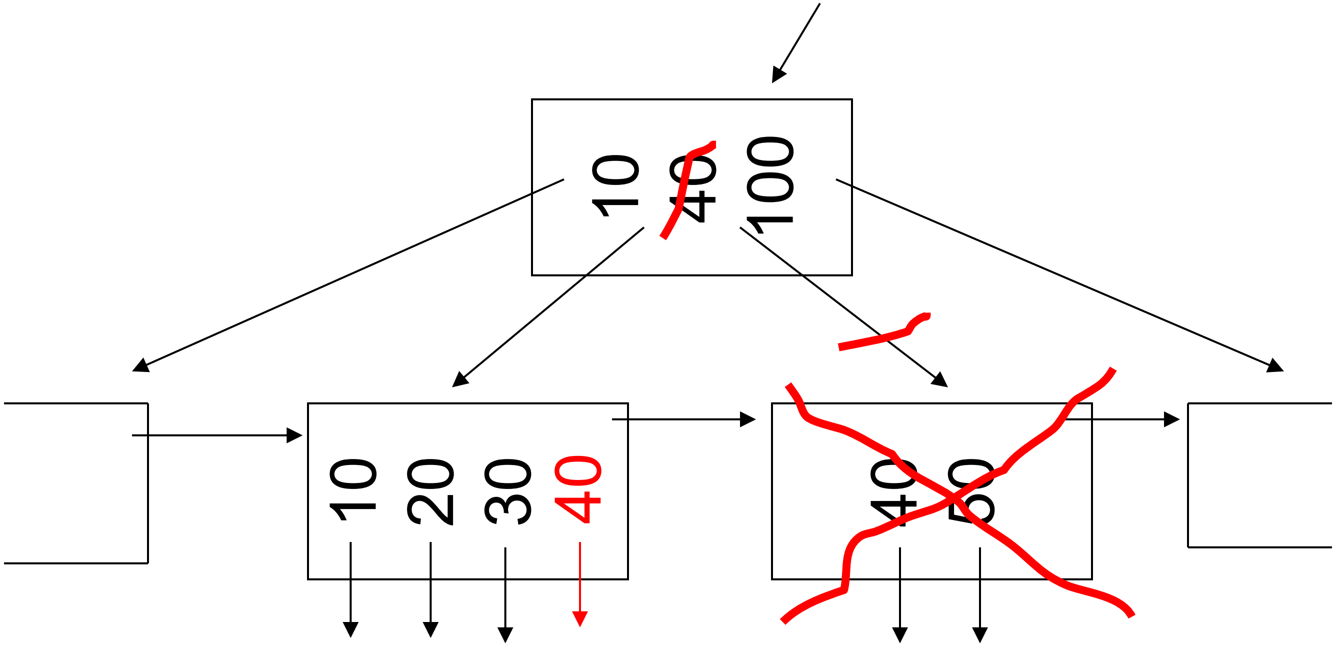
n=4



(b) Coalesce with sibling

» Delete 50

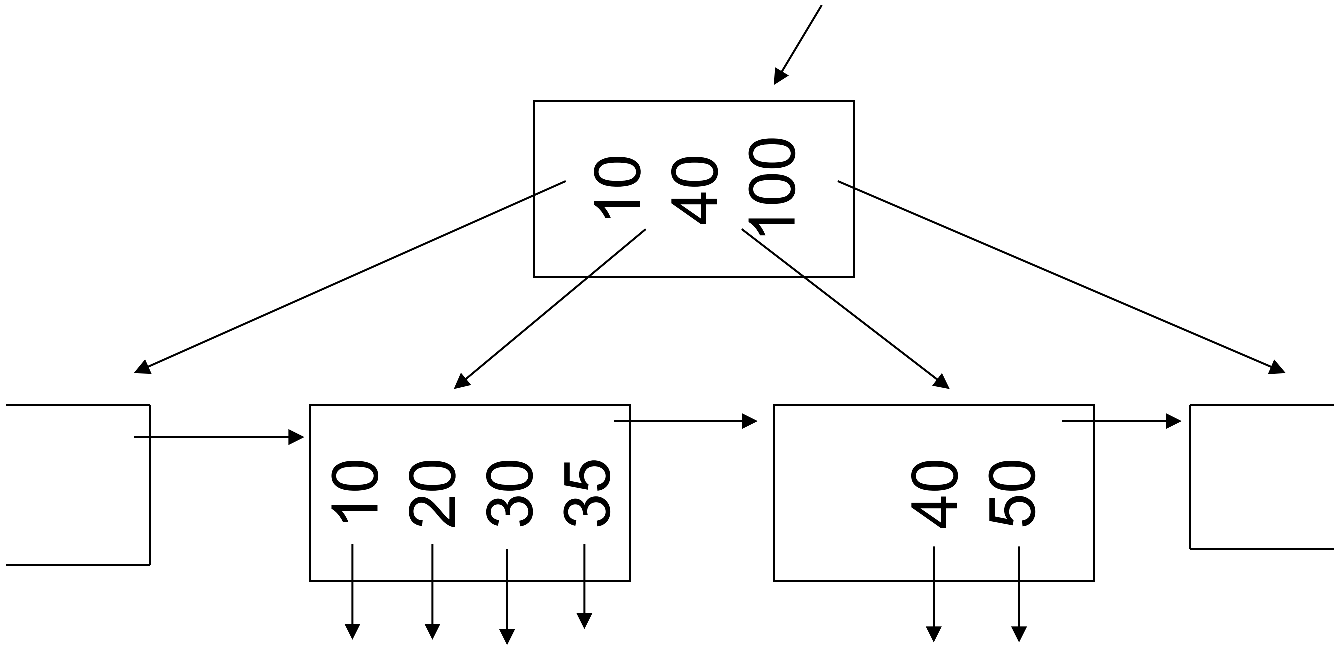
n=4



(c) Redistribute keys

» Delete 50

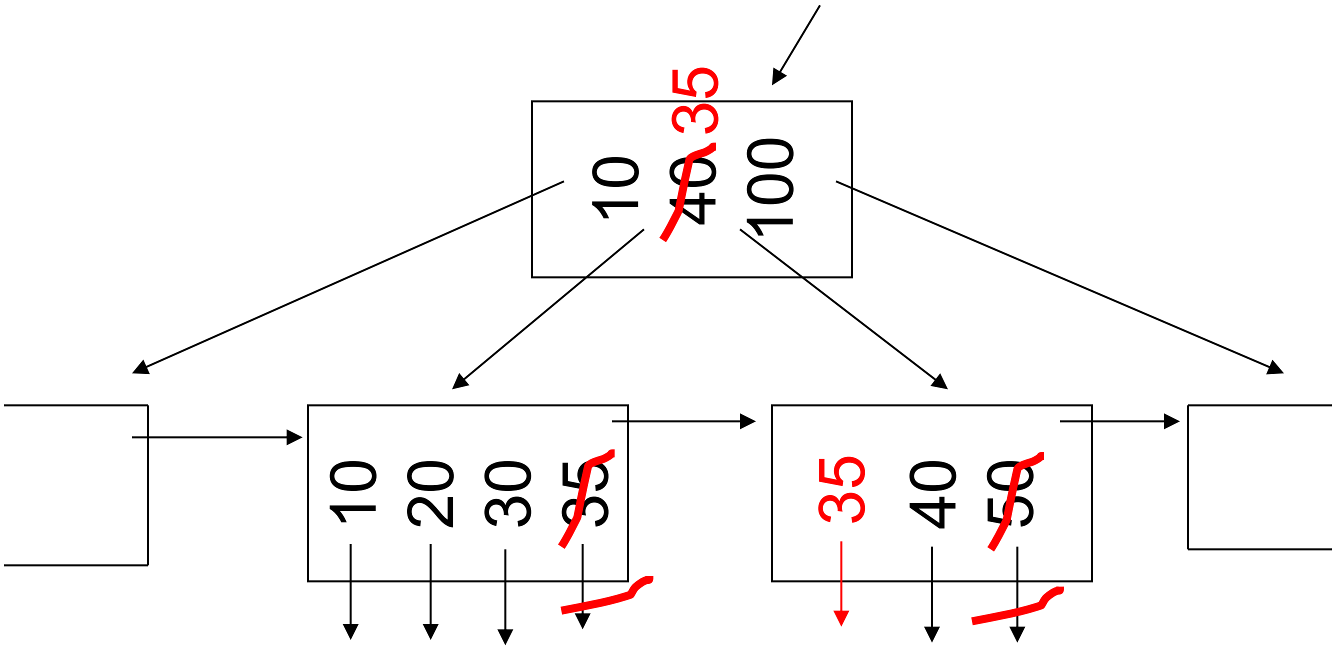
n=4



(c) Redistribute keys

» Delete 50

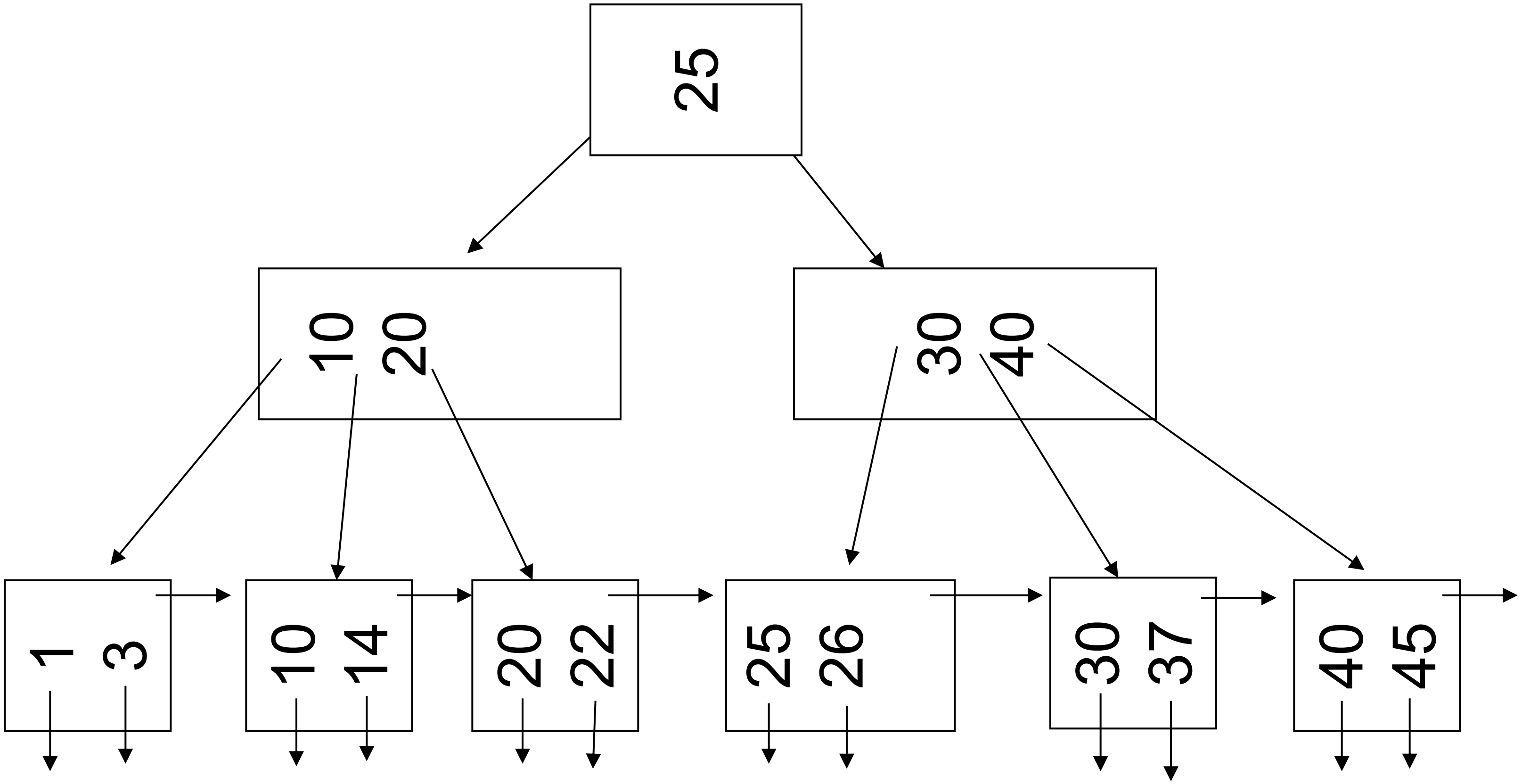
n=4



(d) Non-leaf coalesce

– Delete 37

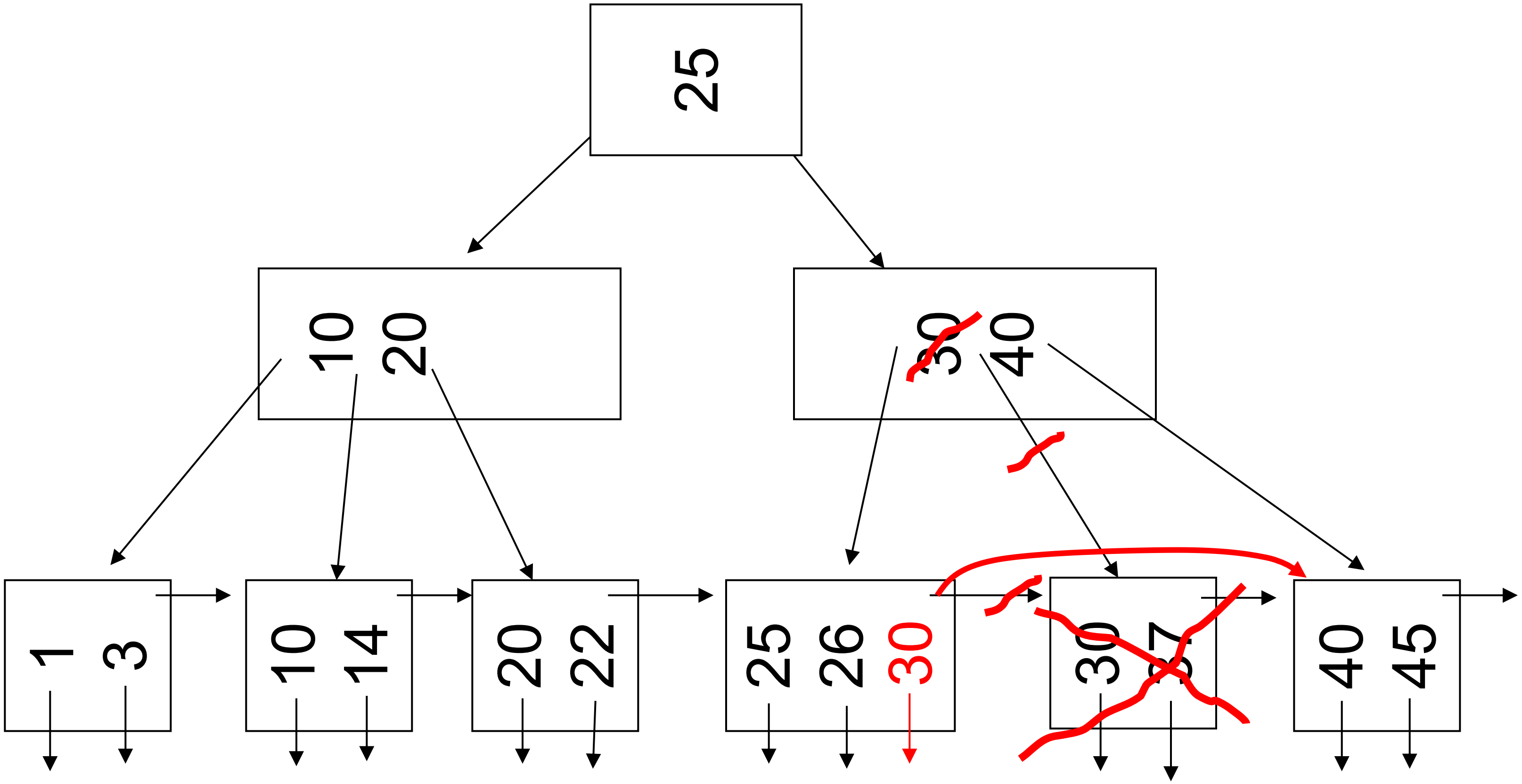
n=4



(d) Non-leaf coalesce

n=4

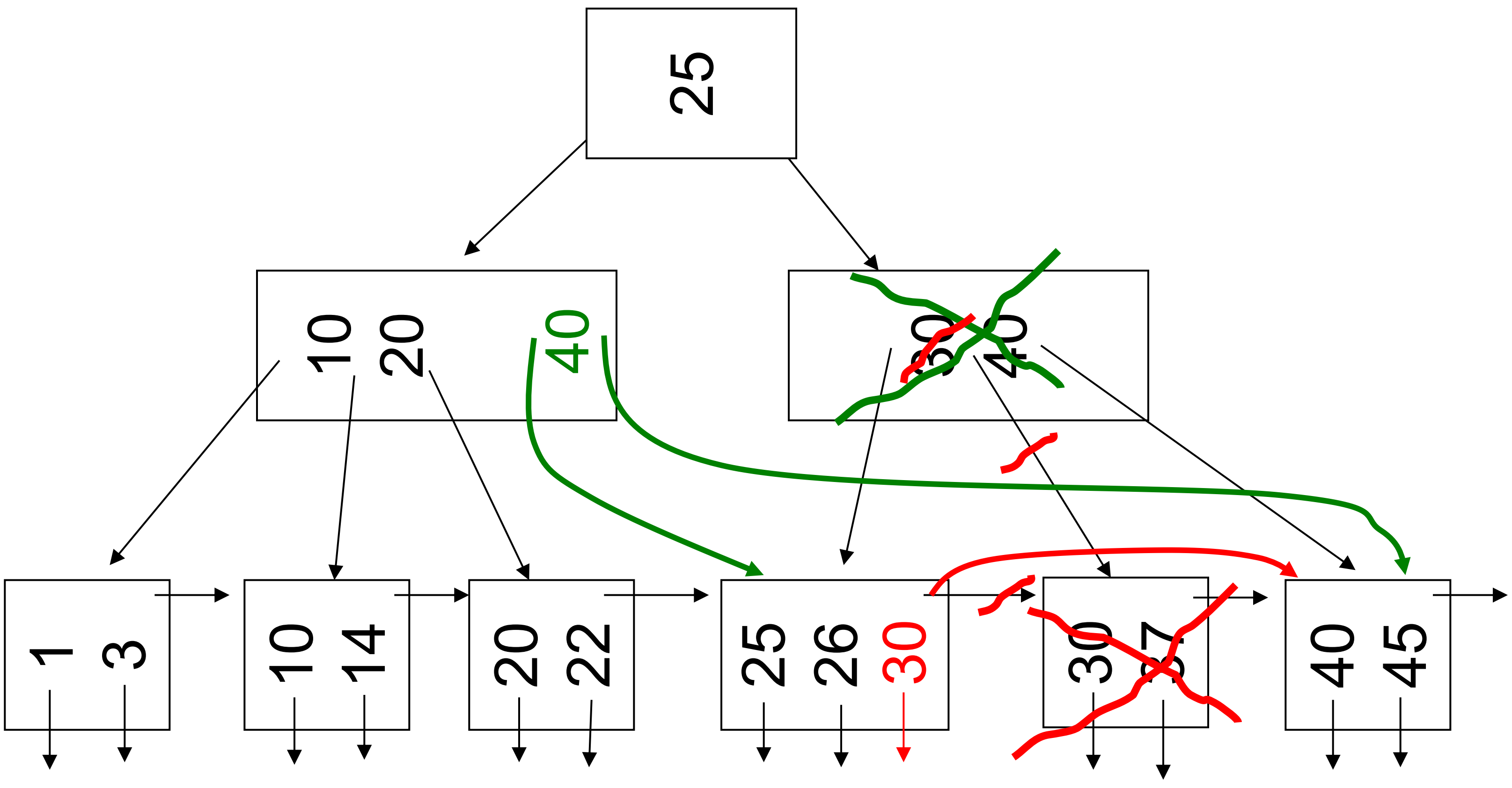
– Delete 37



(d) Non-leaf coalesce

– Delete 37

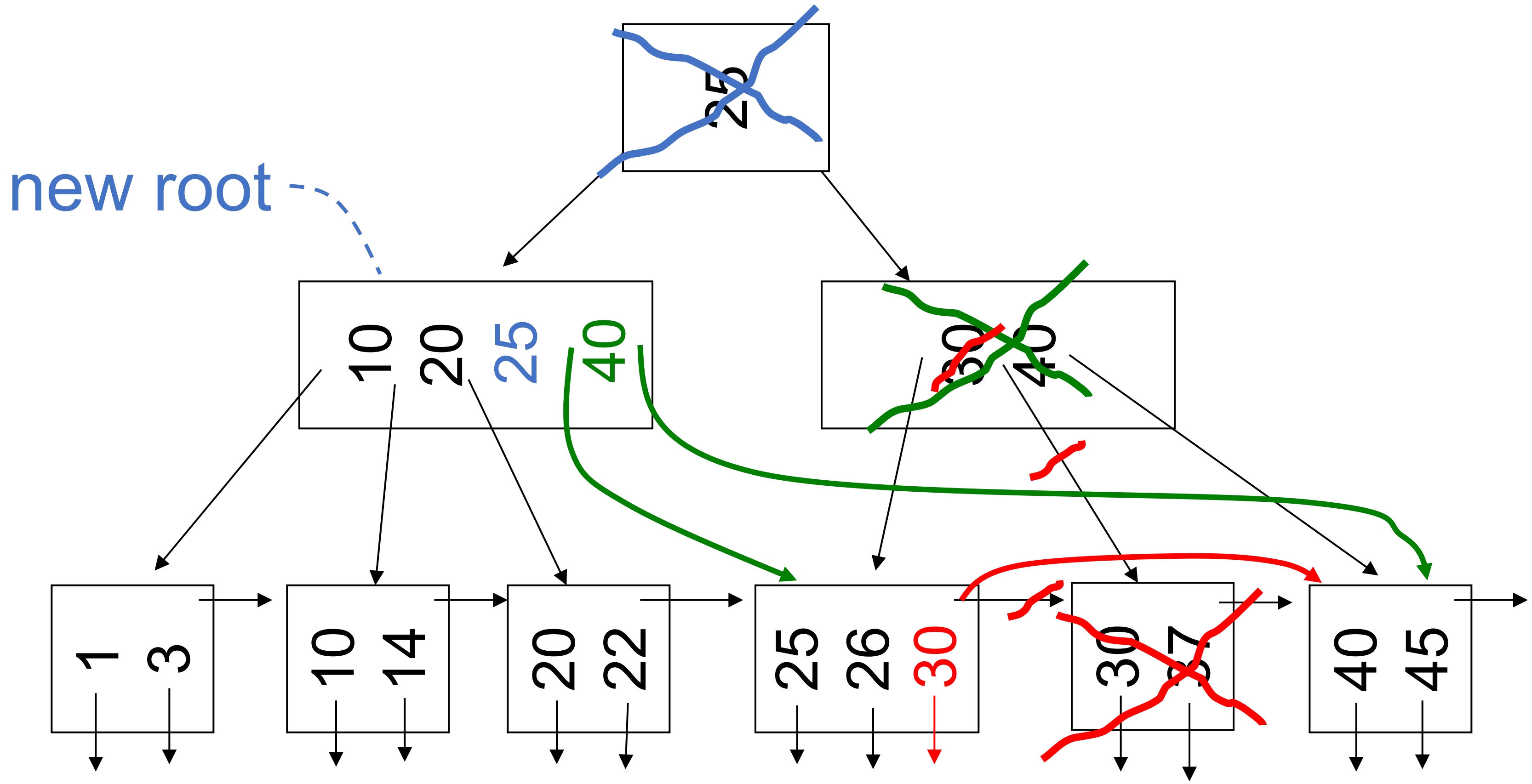
n=4



(d) Non-leaf coalesce

– Delete 37

n=4



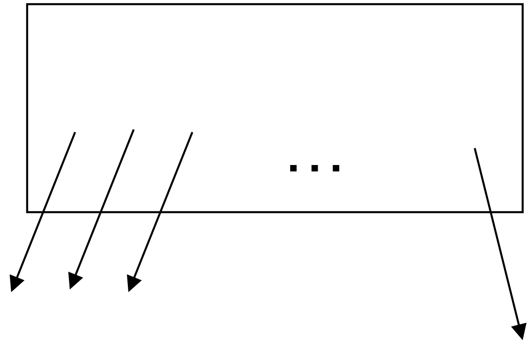
B+ Tree Deletion in Practice

Often, coalescing is not implemented

» Too hard and not worth it! (Most datasets generally grow in size over time.)

Interesting Problem:

For B+ tree, how large should n be?



n is number of keys / node

Sample Assumptions:

(1) Time to read node from disk is
(S + Tn) msec.

Sample Assumptions:

(1) Time to read node from disk is
$(S + Tn)$ msec.

(2) Once block in memory, use binary search to locate key:
$(a + b \log_2 n)$ msec.

For some constants a, b; Assume $a << S$

Sample Assumptions:

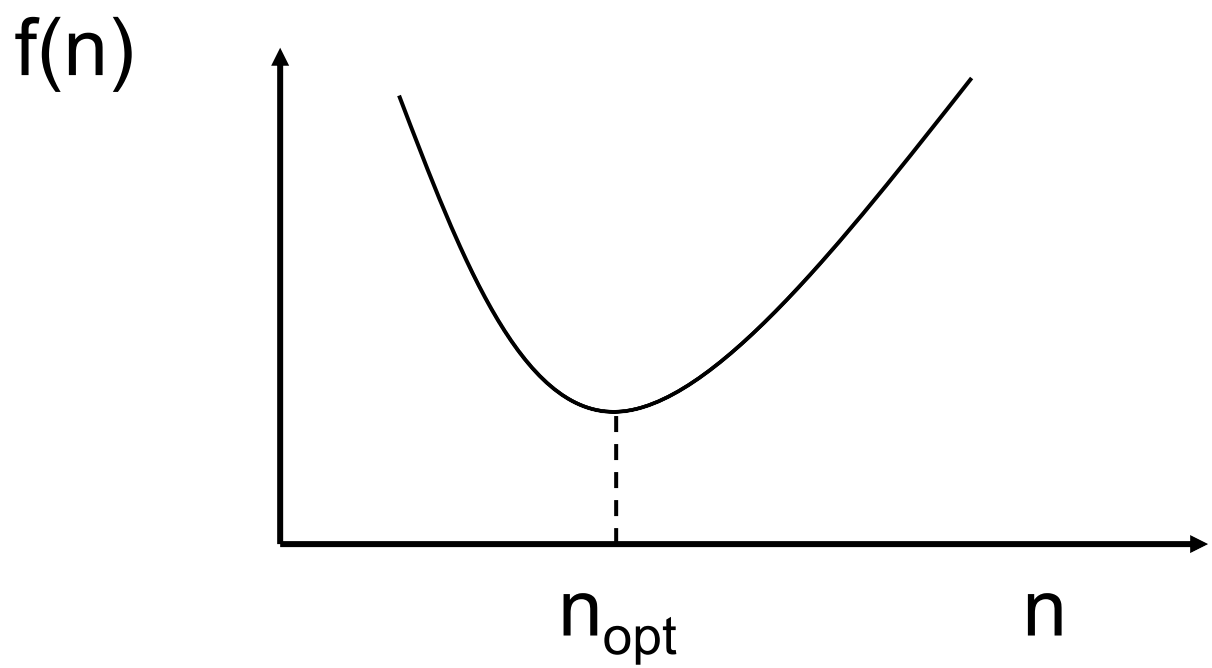
(1) Time to read node from disk is
 $(S + Tn)$ msec.

(2) Once block in memory, use binary search to locate key:
 $(a + b \log_2 n)$ msec.

 For some constants a, b; Assume $a \ll S$

(3) Assume B+tree is full, i.e., # nodes to examine is $\log_n N$ where N = # records

Can Get:
f(n) = time to find a record



Find n_{opt} by setting f'(n) = 0

Answer is n_{opt} = "a few hundred" in practice

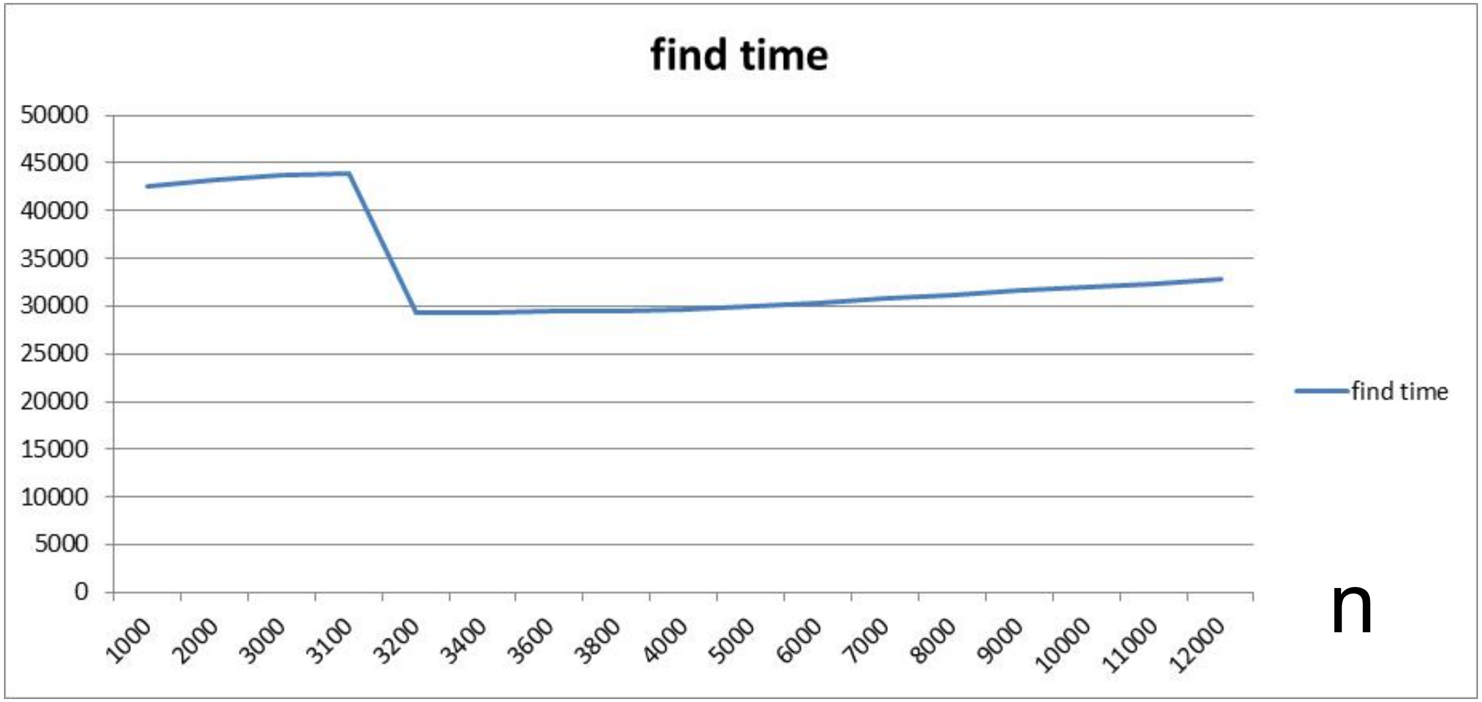
Exercise

$S =$	$14000\ \mu s$
$T =$	$0.2\ \mu s$
$b =$	$0.002\ \mu s$
$a =$	$0\ \mu s$
$N =$	10,000,000

$$f(n) = \log_n N * (S + T\,n + a + b \log_2 n)$$

N = 10 Million Records

S=	14000
T=	0.2
b=	0.002
a=	0
N=	10,000,000

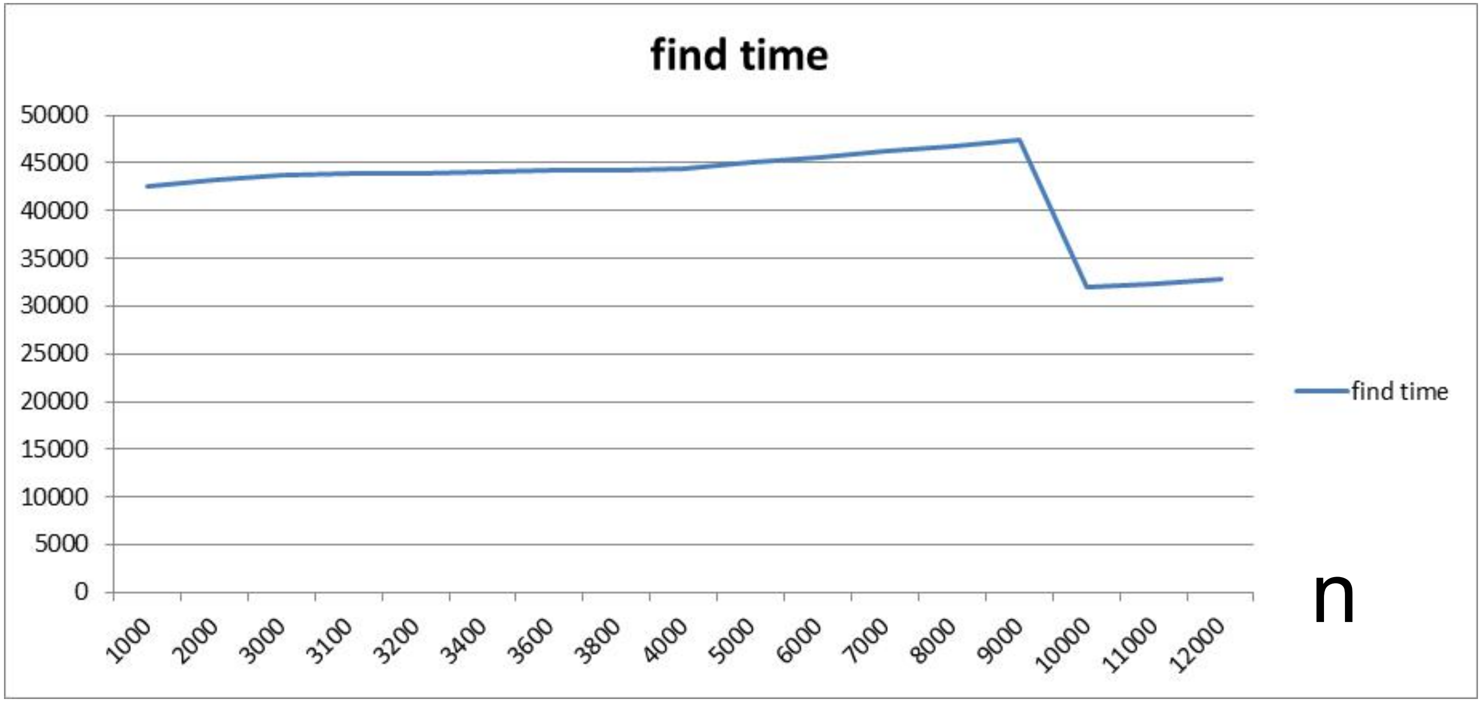


n

times in microseconds

N = 100 Million Records

S=	14000
T=	0.2
b=	0.002
a=	0
N=	100,000,000

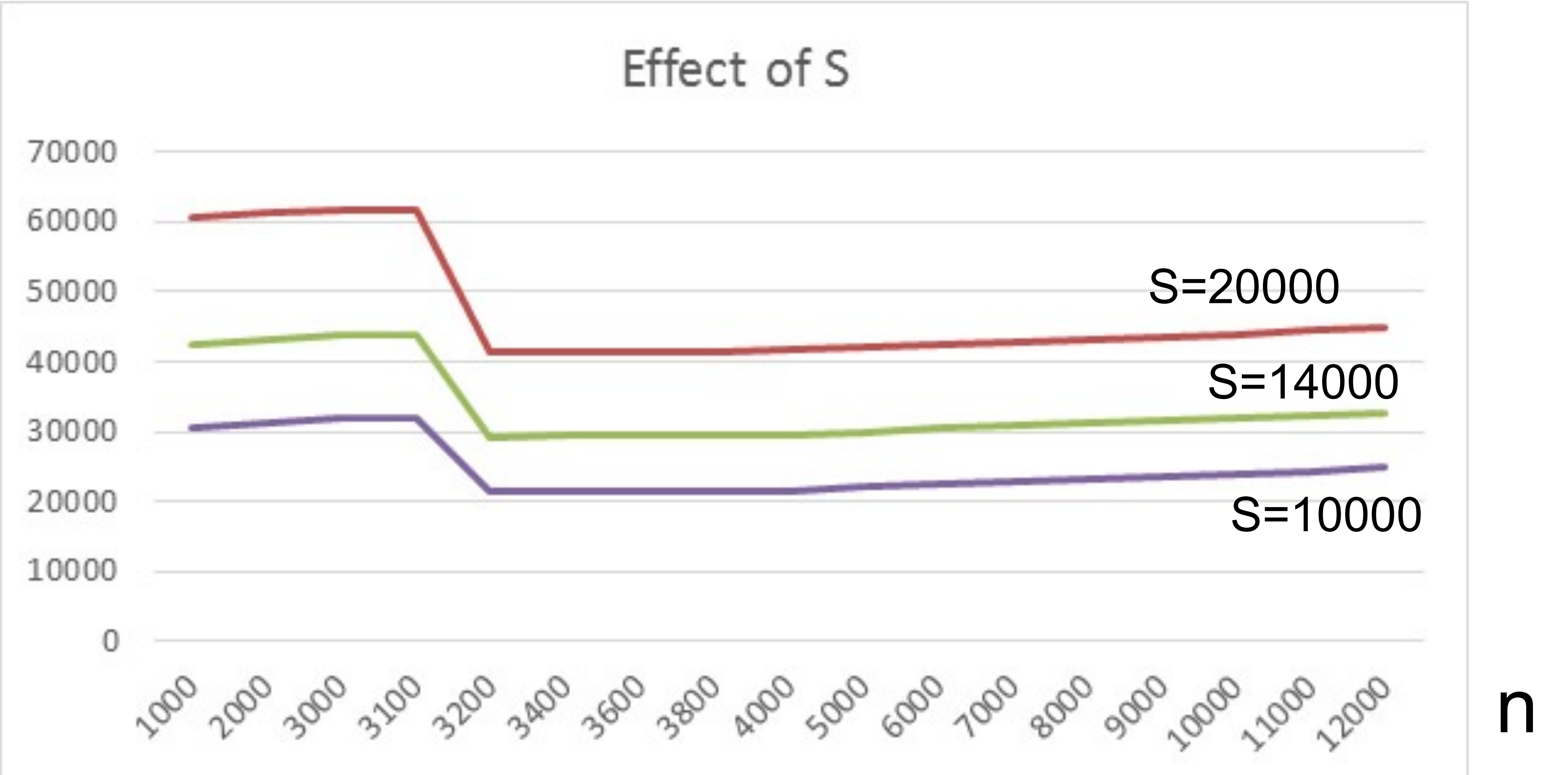


n

times in microseconds

N = 10 Million Records

S=	varies
T=	0.2
b=	0.002
a=	0
N=	10,000,000



times in microseconds

Some Types of Indexes

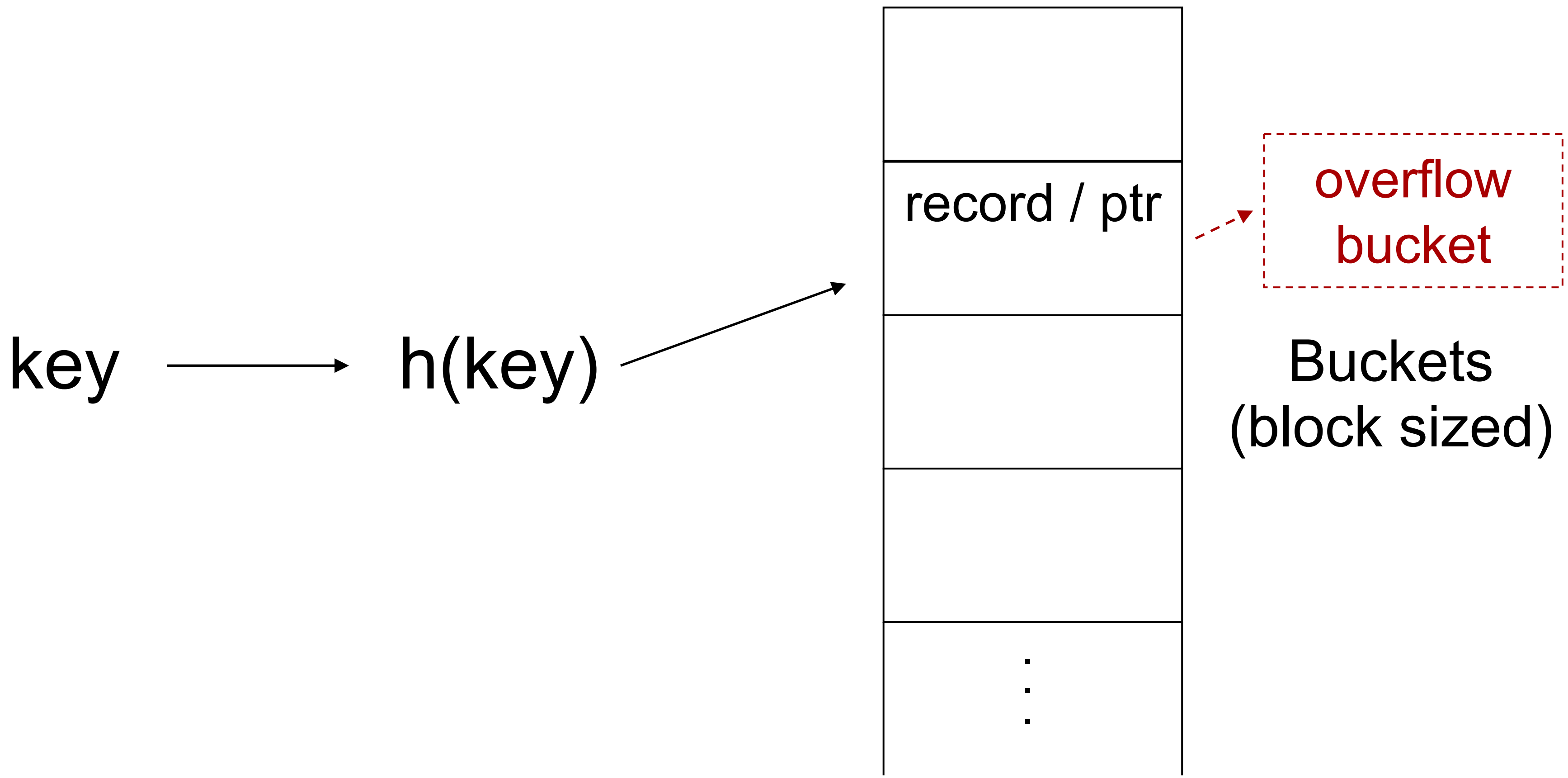
Conventional indexes

B-trees

Hash indexes

Multi-key indexing

Hash Indexes



Chaining is used to handle bucket overflow

Hash vs Tree Indexes

+ O(1) instead of O(log N) disk accesses

– Can’t efficiently do range queries

Challenge: Resizing

Hash tables try to keep occupancy in a fixed range (50-80%) and slow down beyond that

» Too much chaining

How to resize the table when this happens?

» **In memory:** just move everything, amortized cost is pretty low

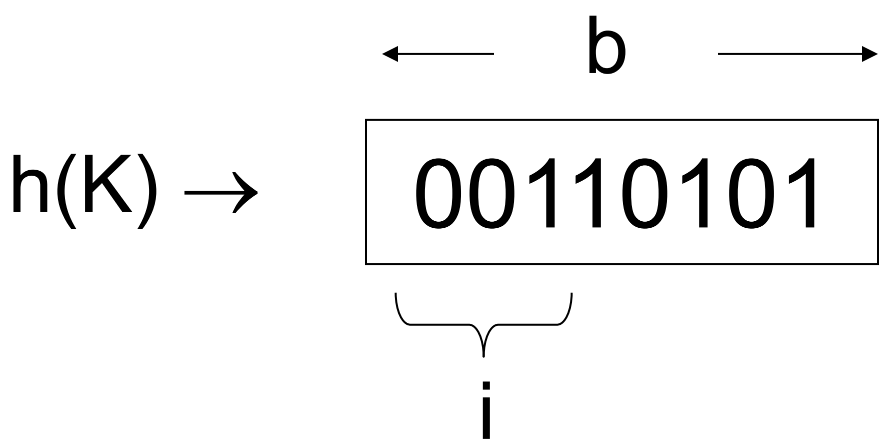
» **On disk:** moving everything is expensive!

Extendible Hashing

Tree-like design for hash tables that allows cheap resizing while requiring 2 IOs / access

Extendible Hashing: 2 Ideas

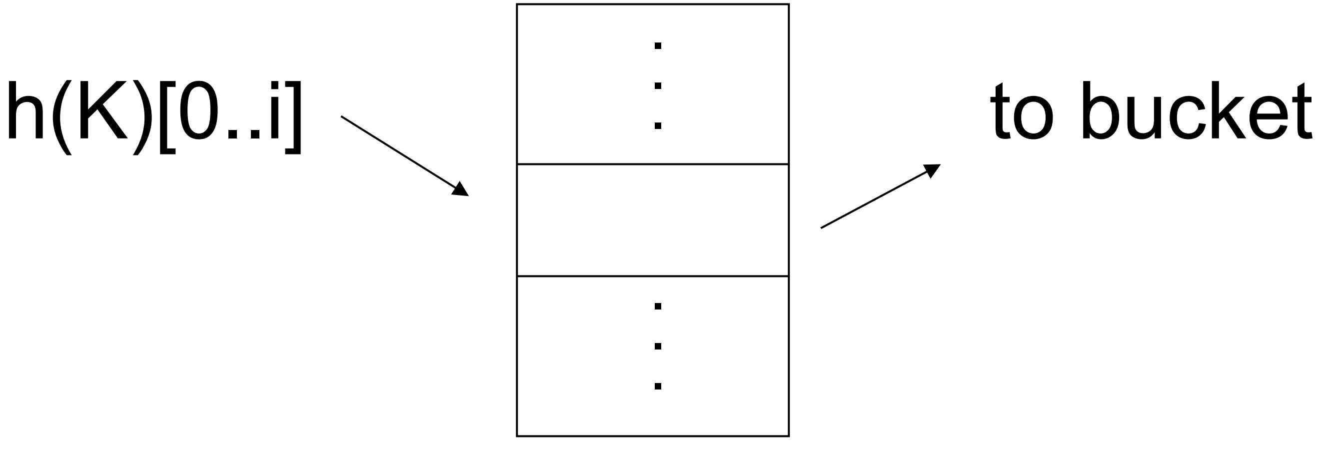
(a) Use i of b bits output by hash function



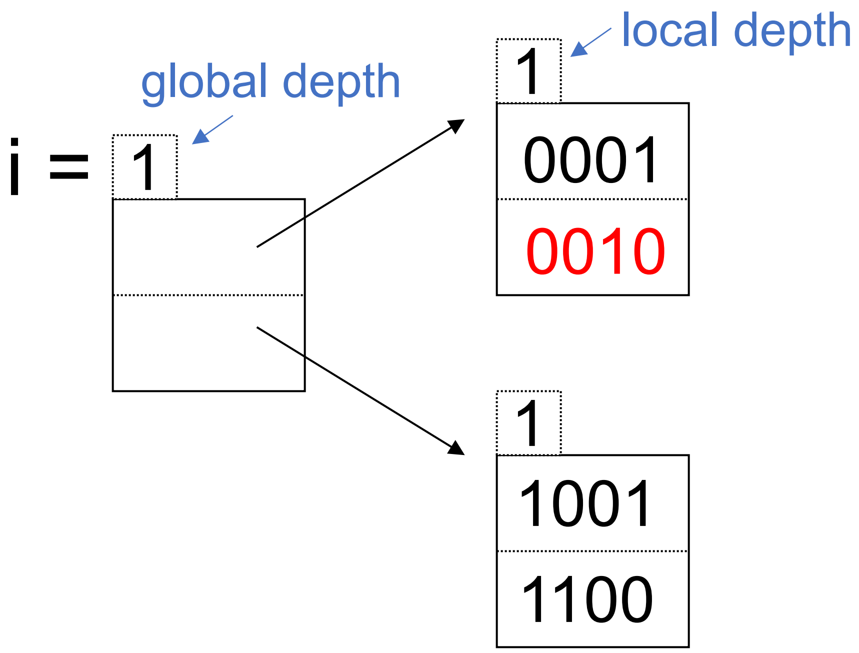
i will grow over time; the first i bits of each key’s hash are used to map it to a bucket

Extendible Hashing: 2 Ideas

(b) Use a directory with pointers to buckets

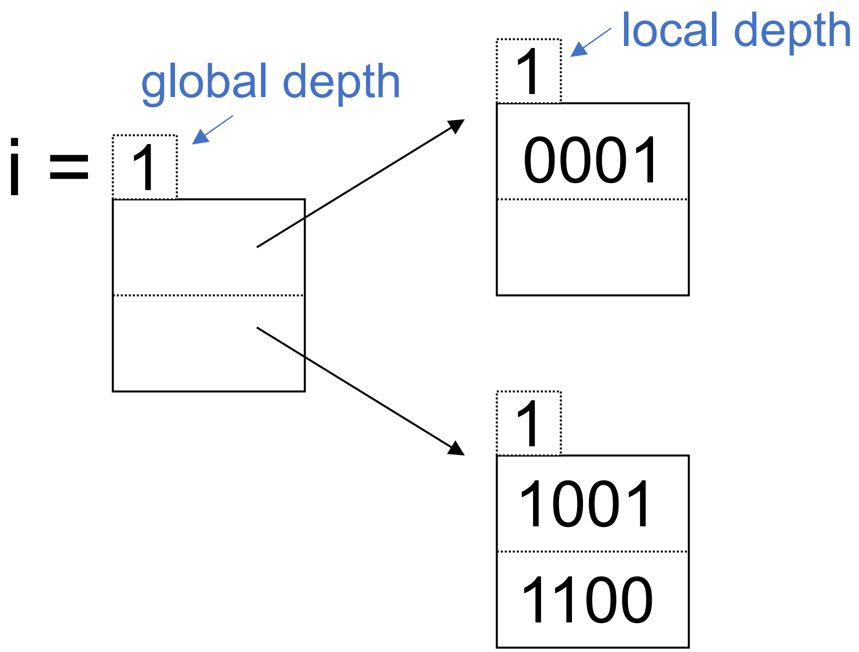


Example: 4-bit h(K), 2 keys/bucket



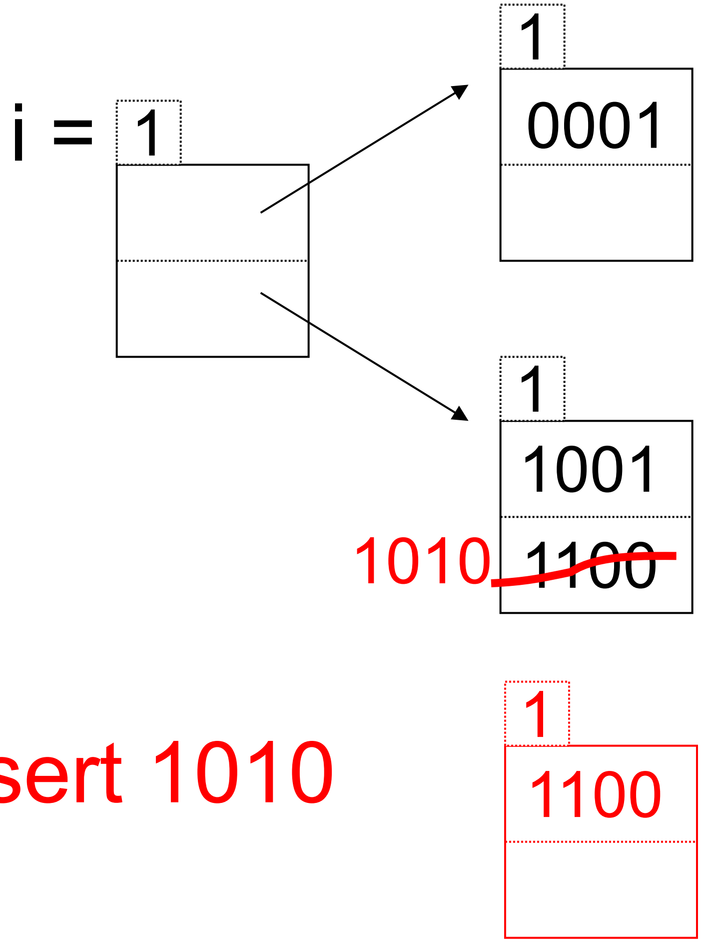
Insert 0010

Example: 4-bit h(K), 2 keys/bucket



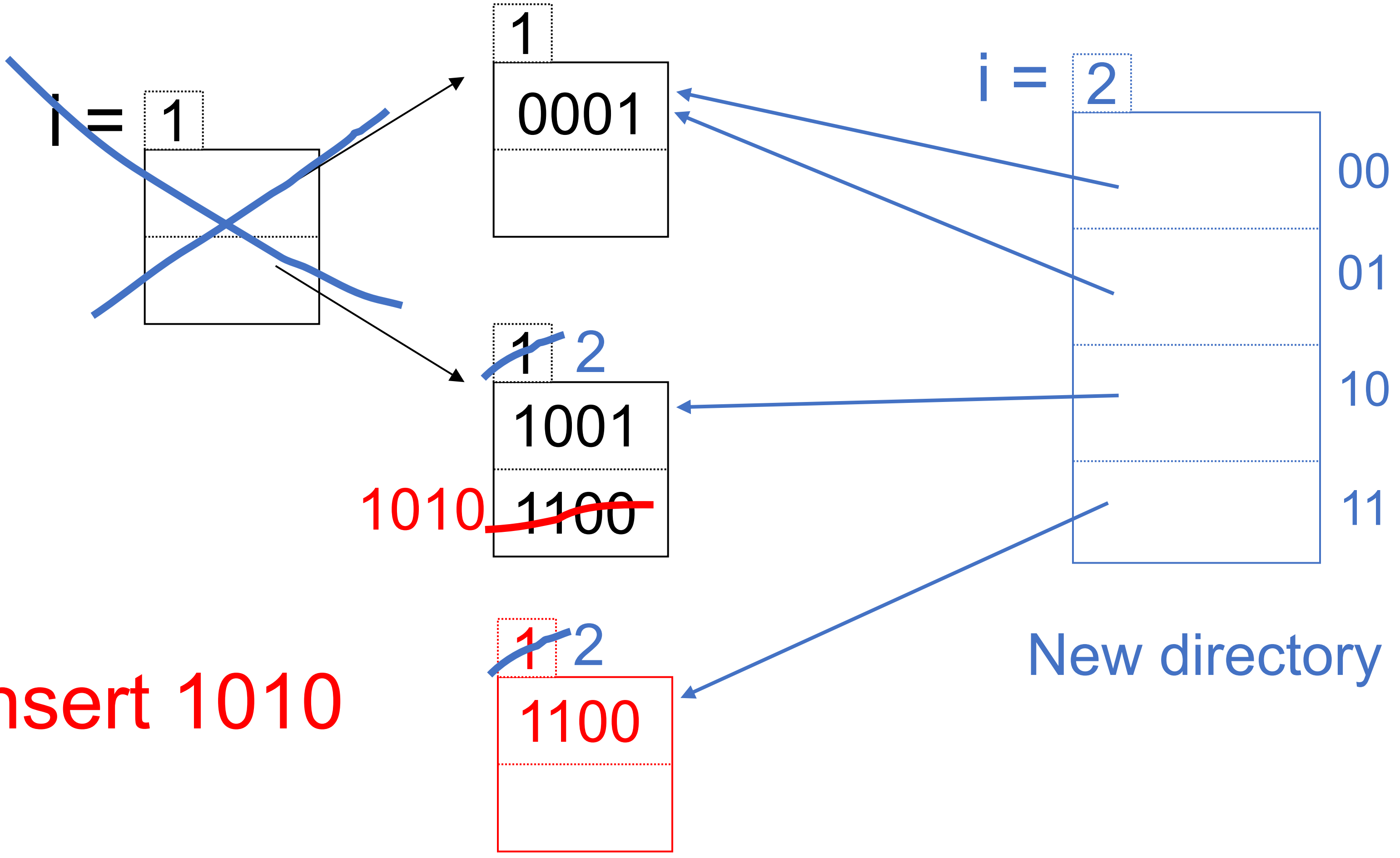
Insert 1010

Example: 4-bit h(K), 2 keys/bucket



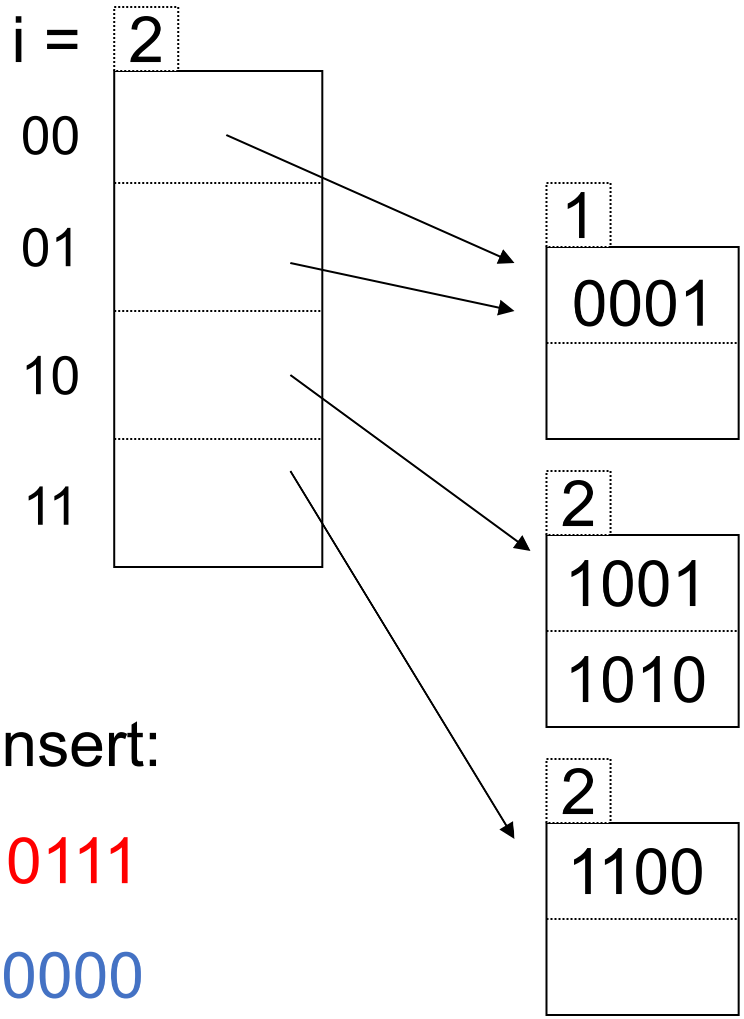
Insert 1010

Example: 4-bit h(K), 2 keys/bucket



Insert 1010

Example

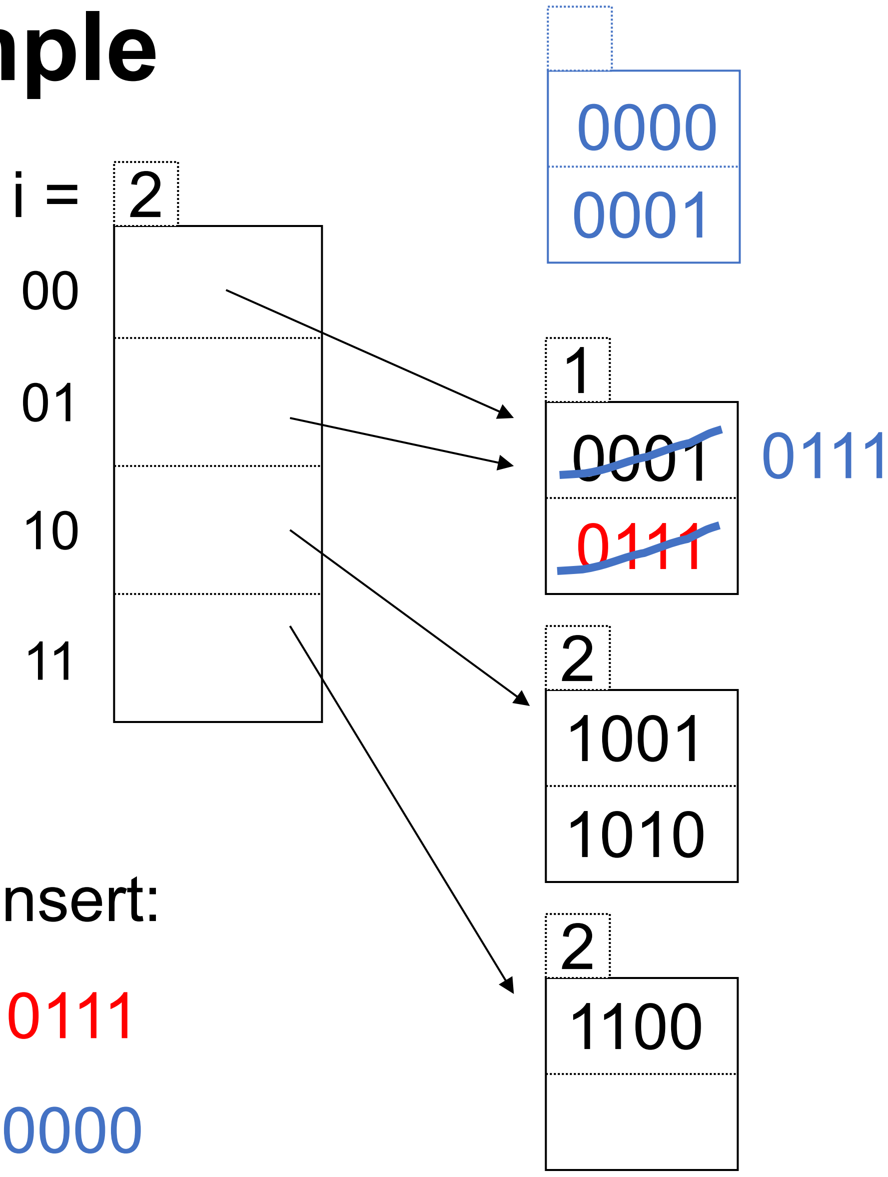


Insert:

0111

0000

Example

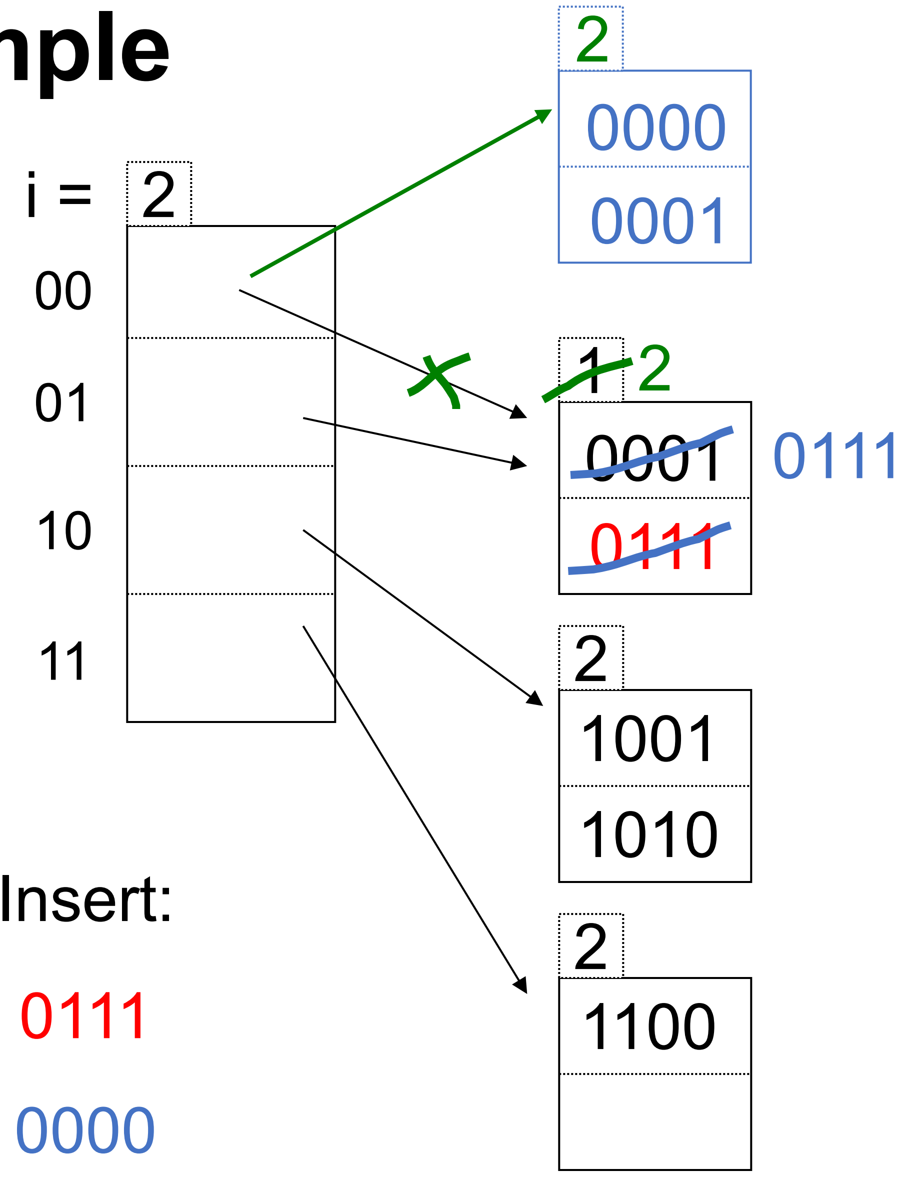


Insert:

0111

0000

Example

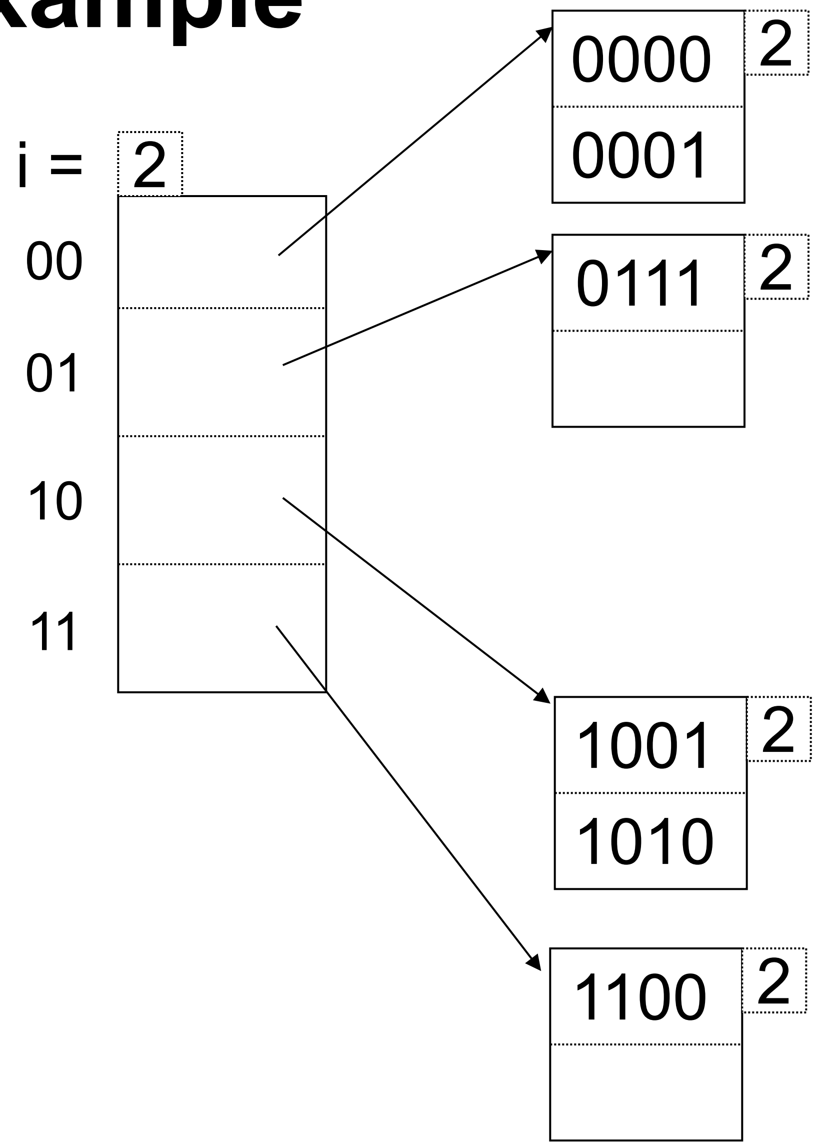


Insert:

0111

0000

Example



Note: still need chaining if values of h(K) repeat and fill a bucket

Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing

Motivation

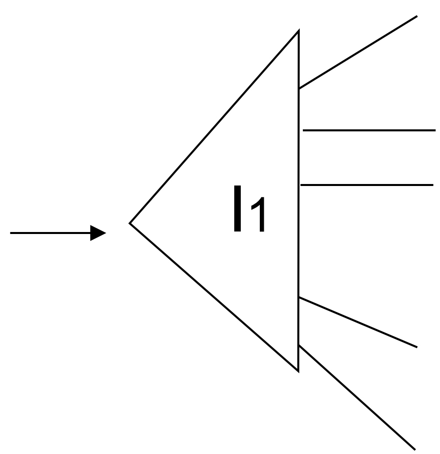
Example: find records where

DEPT = “Toy” AND SALARY > 50k

Strategy I:

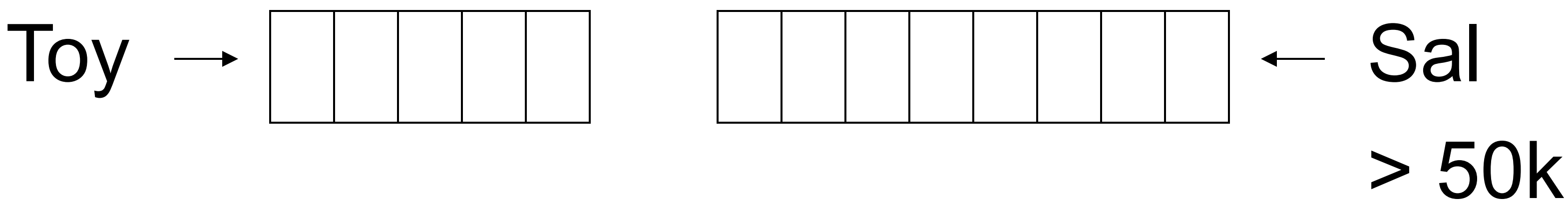
Use one index, say Dept.

Get all Dept = “Toy” records
and check their salary



Strategy II:

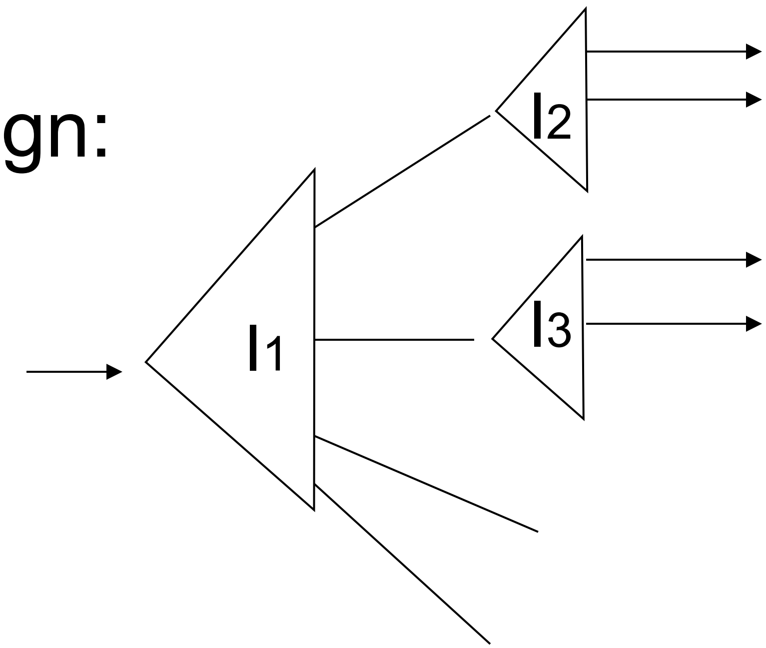
Use 2 indexes; intersect lists of pointers



Strategy III:

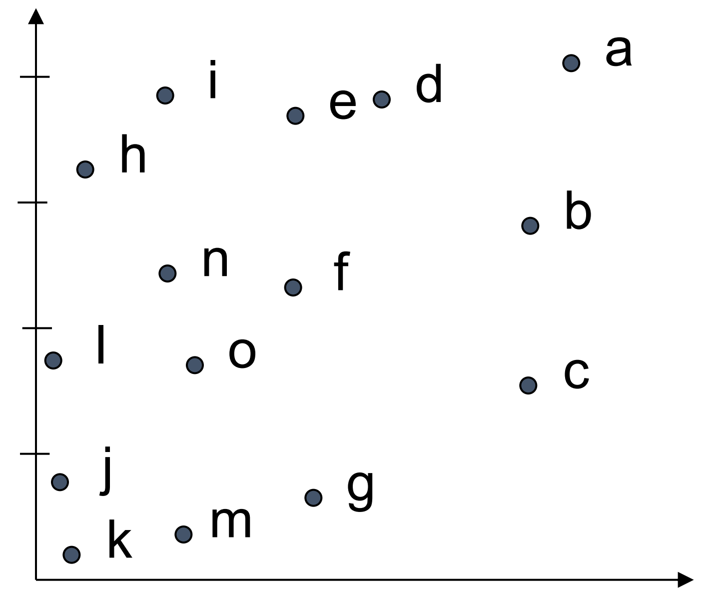
Multi-key index

One design:

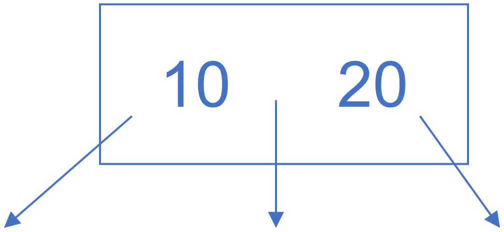


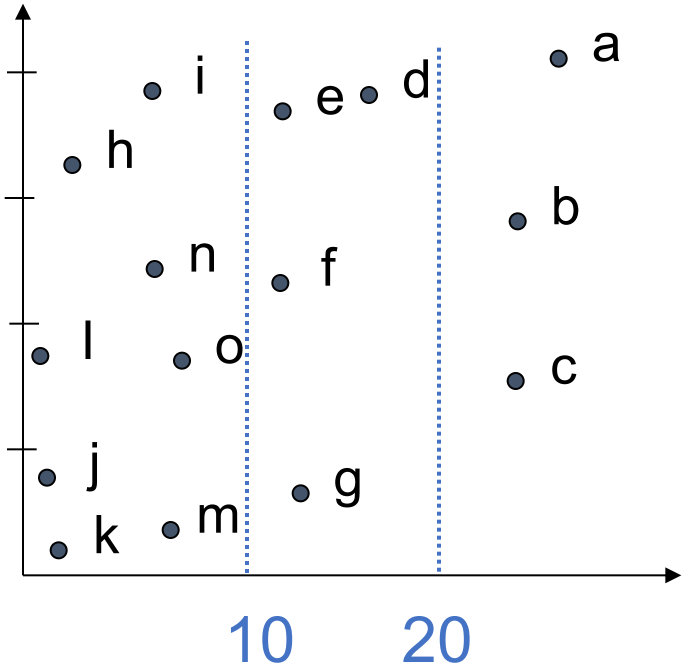
k-d Trees

Split dimensions in any order to hold k-dimensional data

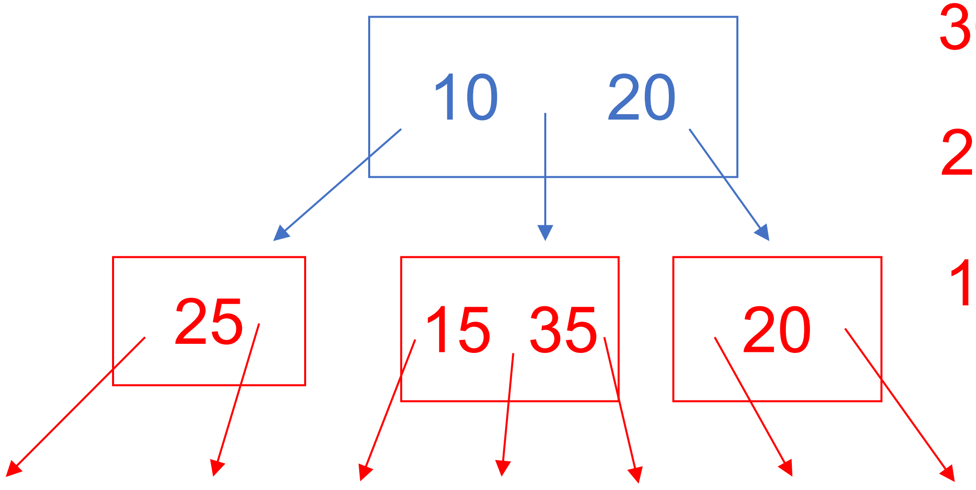


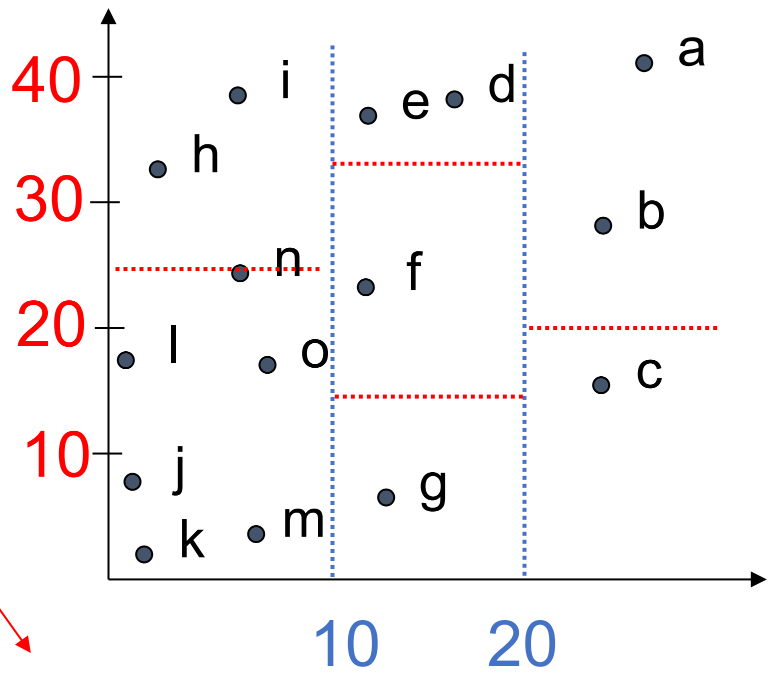
k-d Trees



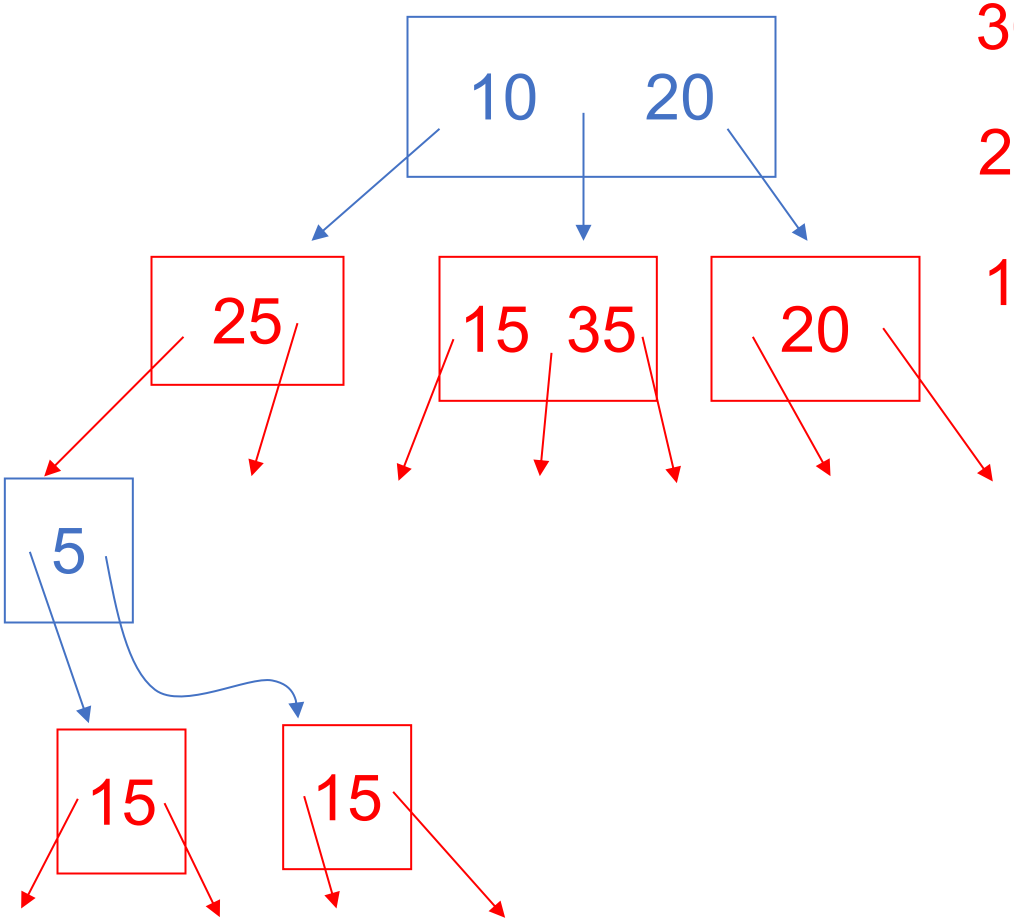


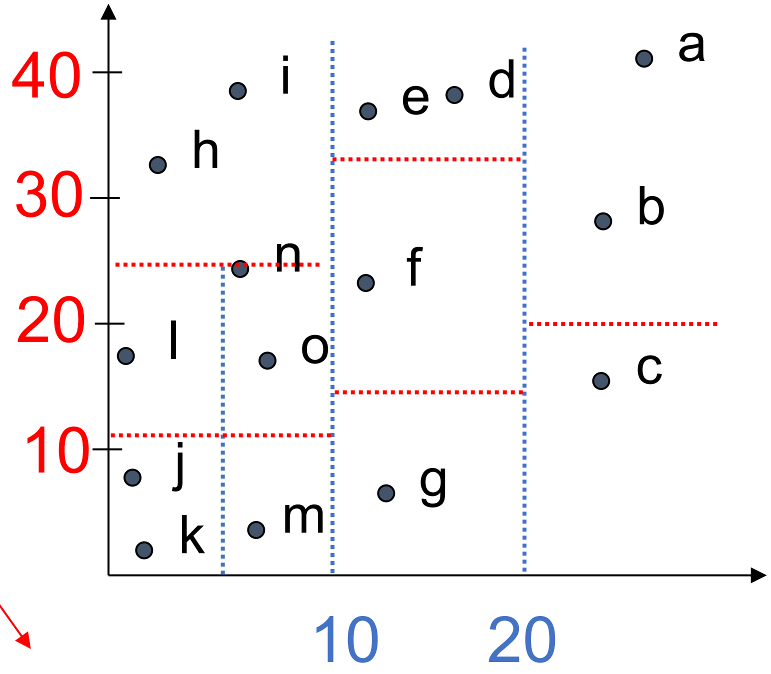
k-d Trees



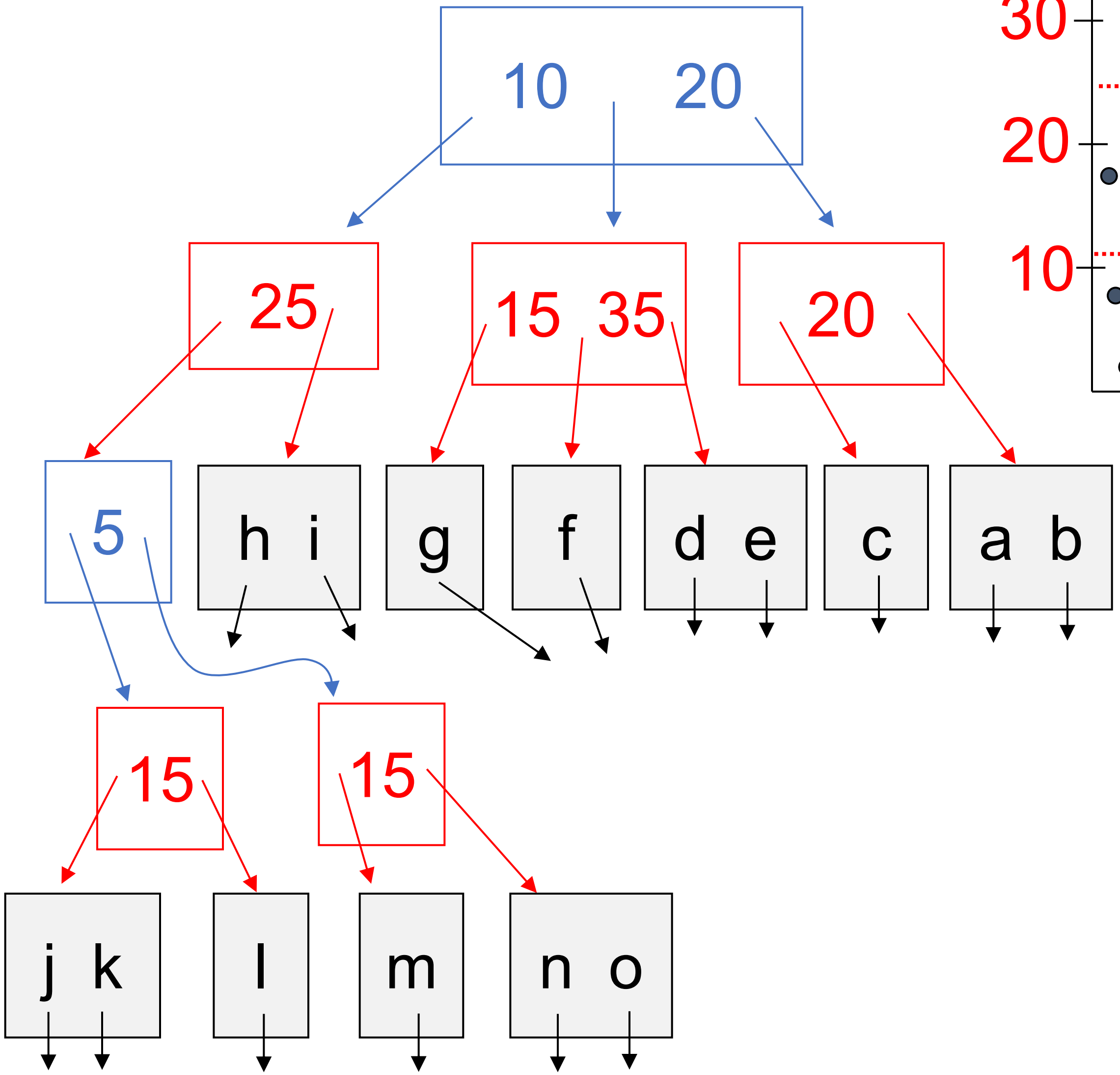


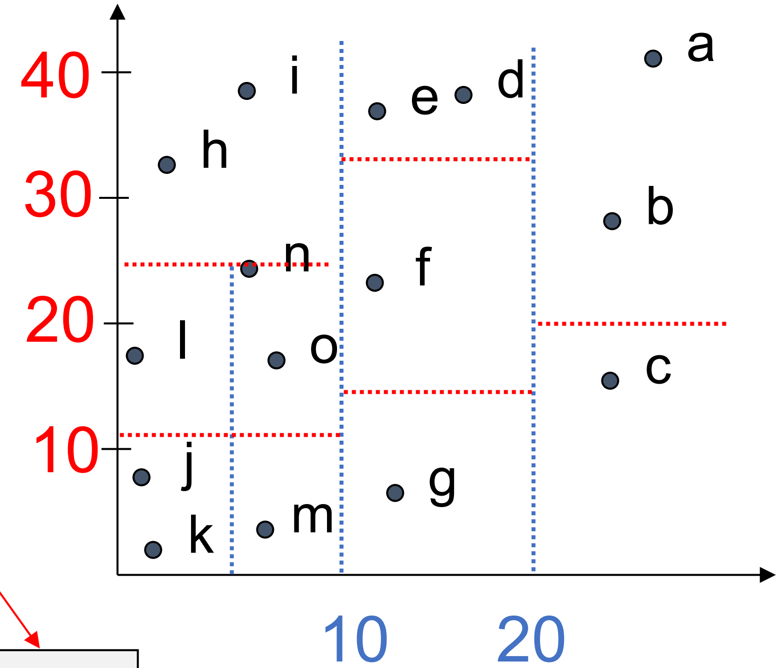
k-d Trees





k-d Trees





Efficient range queries in both dimensions

Storage System Examples

MySQL: transactional DBMS

» Row-oriented storage with 16 KB pages

» Variable length records with headers, overflow

» Index types:

- B-tree
- Hash (in memory only)
- R-tree (spatial data)
- Inverted lists for full text search

» Can compress pages with Lempel-Ziv

Storage System Examples

Apache Parquet + Hive: analytical data lake

» Column-oriented storage as set of ~1 GB files (each file has a slice of all columns)

» Various compression and encoding schemes at the level of pages in a file

- Special scheme for nested fields (Dremel)

» Header with statistics at the start of each file

- Min/max of columns, nulls, Bloom filter

» Files partitioned into directories by one key

Query Execution

Overview

Relational operators

Execution methods

Query Execution Overview

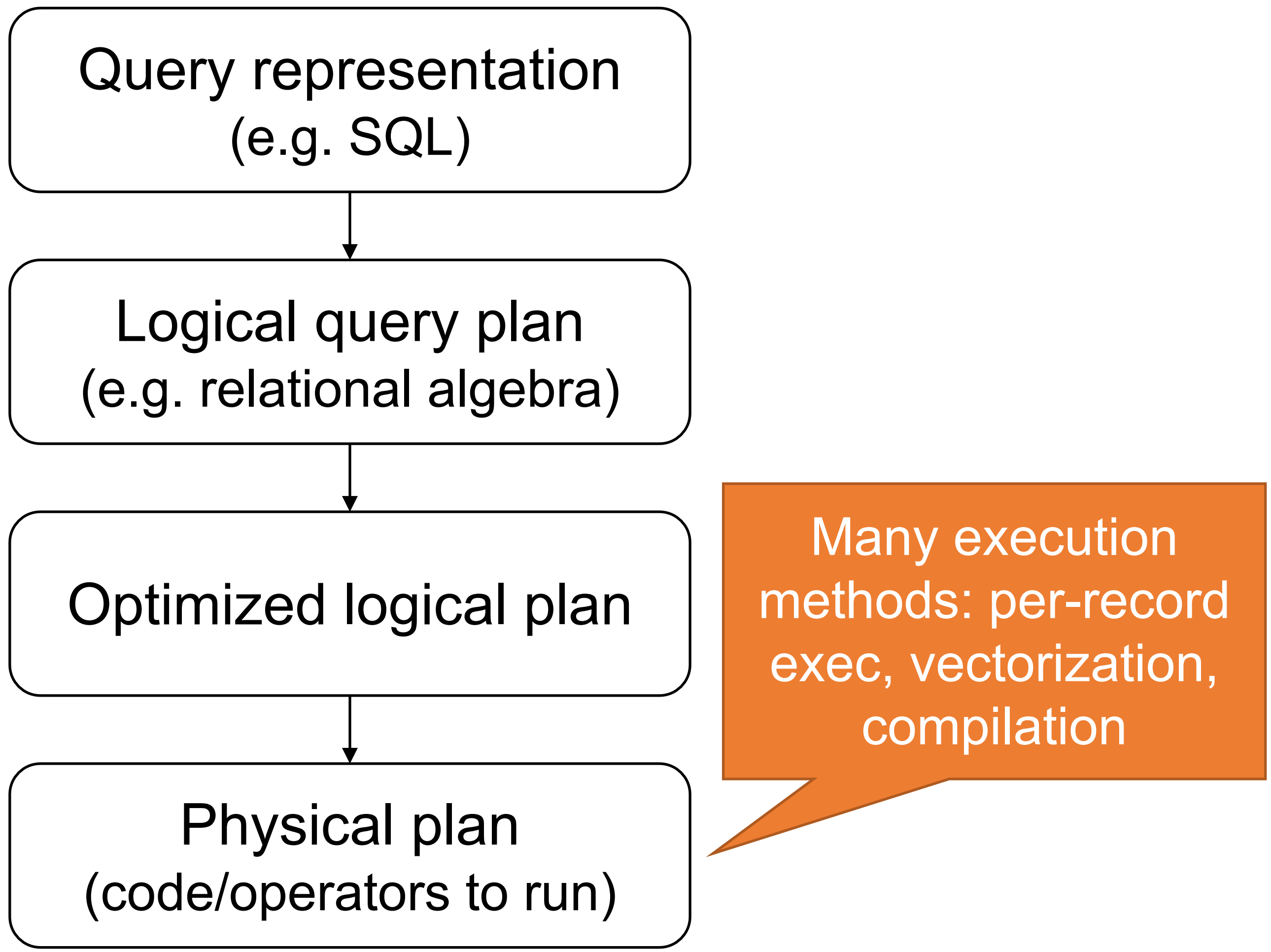
Recall that one of our key principles in data intensive systems was **declarative APIs**

» Specify what you want to compute, not how

We saw how these can translate into many storage strategies

How to execute queries in a declarative API?

Query Execution Overview



Plan Optimization Methods

Rule-based: systematically replace some expressions with other expressions

» Replace X OR TRUE with TRUE

» Replace M*A + M*B with M*(A+B) for matrices

Cost-based: propose several execution plans and pick best based on a **cost model**

Adaptive: update execution plan at runtime

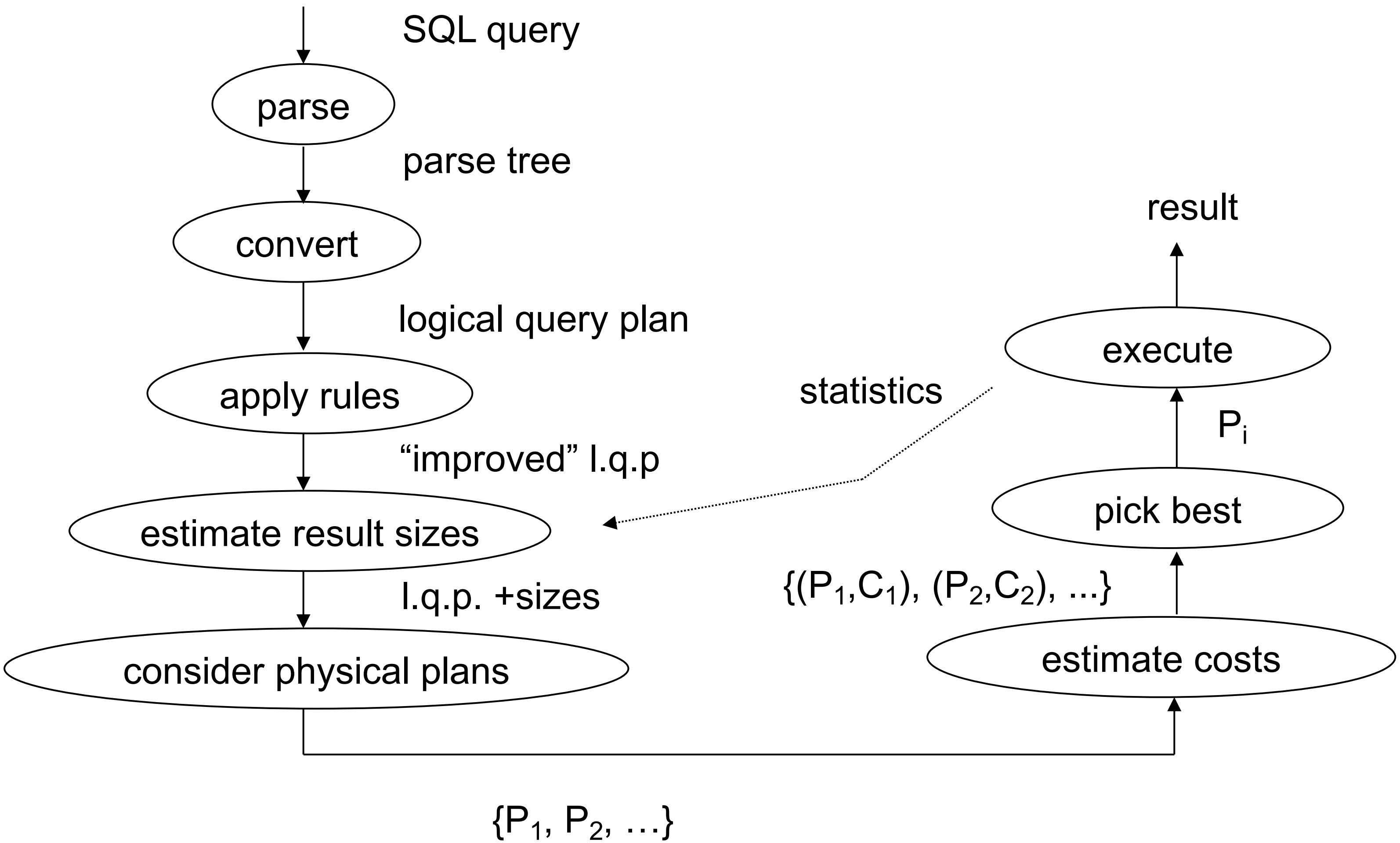
Execution Methods

Interpretation: walk through query plan operators for each record

Vectorization: walk through in batches

Compilation: generate code (like System R)

Typical RDBMS Execution



Query Execution

Overview

Relational operators

Execution methods

The Relational Algebra

Collection of operators over tables (relations)

» Each table has named attributes (fields)

Codd's original RA: tables are **sets of tuples** (unordered and tuples cannot repeat)

SQL's RA: tables are **bags (multisets) of tuples**; unordered but each tuple may repeat

Relational Algebra Operators

Basic set operators:

Intersection: $R \cap S$

Union: $R \cup S$ — for tables with same schema

Difference: $R - S$

Cartesian Product: $R \times S$ $\{ (r, s) \mid r \in R, s \in S \}$

Relational Algebra Operators

Basic set operators:

Intersection: $R \cap S$

Union: $R \cup S$ ← consider both distinct (set union) and non-distinct (bag union)

Difference: $R - S$

Cartesian Product: $R \times S$

Relational Algebra Operators

Special query processing operators:

Selection: $\sigma_{condition}(R)$ { r ∈ R | condition(r) is true }

Projection: $\Pi_{expressions}(R)$ { expressions(r) | r ∈ R }

Natural Join: $R \bowtie S$ { (r, s) ∈ R × S) | r.key = s.key }
where key is the common fields

Relational Algebra Operators

Special query processing operators:

Aggregation: ${}_{keys}G_{agg(attr)}(R)$

SELECT agg(attr)
FROM R
GROUP BY keys

Examples: ${}_{department}G_{Max(salary)}(Employees)$

$G_{Max(salary)}(Employees)$

Algebraic Properties

Many properties about which combinations of operators are equivalent

» That's why it's called an algebra!

Properties: Unions, Products and Joins

$R \cup S = S \cup R$

Tuple order in a relation doesn't matter (unordered)

$R \cup (S \cup T) = (R \cup S) \cup T$

$R \times S = S \times R$

Attribute order in a relation doesn't matter either

$(R \times S) \times T = R \times (S \times T)$

$R \bowtie S = S \bowtie R$

$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

Properties: Selects

$\sigma_{p \wedge q}(R) =$

$\sigma_{p \vee q}(R) =$

Properties: Selects

$$\sigma_{p \wedge q}(R) = \sigma_p(\sigma_q(R))$$

$$\sigma_{p \vee q}(R) = \sigma_p(R) \cup \sigma_q(R)$$

careful with repeated elements

Bags vs. Sets

$R = \{a,a,b,b,b,c\}$

$S = \{b,b,c,c,d\}$

$R \cup S = ?$

Bags vs. Sets

R = {a,a,b,b,b,c}

S = {b,b,c,c,d}

R ∪ S = ?

- **Option 1:** SUM of counts
  R ∪ S = {a,a,b,b,b,b,b,c,c,c,d}
- **Option 2:** MAX of counts
  R ∪ S = {a,a,b,b,b,c,c,d}

Executive Decision

Use “SUM” option for bag unions

Some rules that work for set unions cannot be used for bags

Properties: Project

Let: X = set of attributes

Y = set of attributes

$\Pi_{X \cup Y}$ (R) =

Properties: Project

Let: X = set of attributes

Y = set of attributes

$$\Pi_{X \cup Y} (R) = \Pi_X(\Pi_Y(R))$$

Properties: Project

Let: X = set of attributes

Y = set of attributes

$$\Pi_{X \cup Y}(R) = \Pi_X(\Pi_Y(R))$$

Properties: $\sigma + \bowtie$

Let p = predicate with only R attribs

q = predicate with only S attribs

m = predicate with only R, S attribs

$\sigma_p(R \bowtie S) =$

$\sigma_q(R \bowtie S) =$

Properties: $\sigma + \bowtie$

Let p = predicate with only R attribs

q = predicate with only S attribs

m = predicate with only R, S attribs

$\sigma_p(R \bowtie S) = \sigma_p(R) \bowtie S$

$\sigma_q(R \bowtie S) = R \bowtie \sigma_q(S)$

Properties: $\sigma + \bowtie$

Some rules can be derived:

$\sigma_{p \wedge q}(R \bowtie S) =$

$\sigma_{p \wedge q \wedge m}(R \bowtie S) =$

$\sigma_{p \vee q}(R \bowtie S) =$

Properties: $\sigma + \bowtie$

Some rules can be derived:

$\sigma_{p \wedge q}(R \bowtie S) = \sigma_p(R) \bowtie \sigma_q(S)$

$\sigma_{p \wedge q \wedge m}(R \bowtie S) = \sigma_m(\sigma_p(R) \bowtie \sigma_q(S))$

$\sigma_{p \vee q}(R \bowtie S) = (\sigma_p(R) \bowtie S) \cup (R \bowtie \sigma_q(S))$

Prove One, Others for Practice

$$\sigma_{p \wedge q}(R \bowtie S) = \sigma_p(\sigma_q(R \bowtie S))$$

$$= \sigma_p(R \bowtie \sigma_q(S))$$

$$= \sigma_p(R) \bowtie \sigma_q(S)$$

Properties: Π + σ

Let x = subset of R attributes

z = attributes in predicate p
(subset of R attributes)

$\Pi_x(\sigma_p (R))$ =

Properties: Π + σ

Let x = subset of R attributes

z = attributes in predicate p (subset of R attributes)

$$\Pi_x(\sigma_p (R)) = \sigma_p(\Pi_x(R))$$

Properties: Π + σ

Let x = subset of R attributes

z = attributes in predicate p
(subset of R attributes)

$$\Pi_x(\sigma_p (R)) = \Pi_x(\sigma_p(\Pi_{x \cup z}(R)))$$

Properties: Π + $\bowtie$

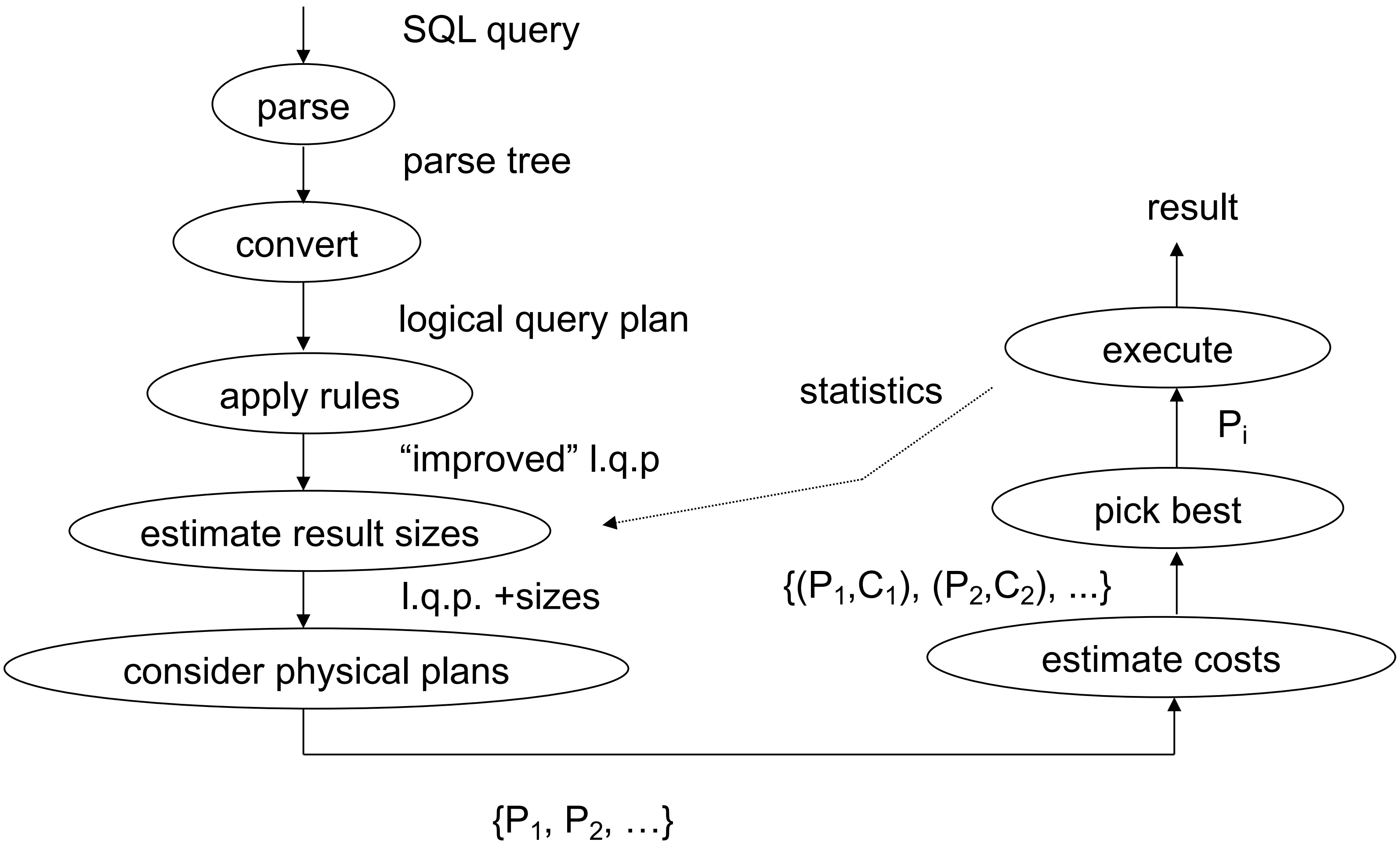
Let x = subset of R attributes

y = subset of S attributes

z = intersection of R,S attributes

$$\Pi_{x \cup y}(R \bowtie S) = \Pi_{x \cup y}((\Pi_{x \cup z}(R)) \bowtie (\Pi_{y \cup z}(S)))$$

Typical RDBMS Execution

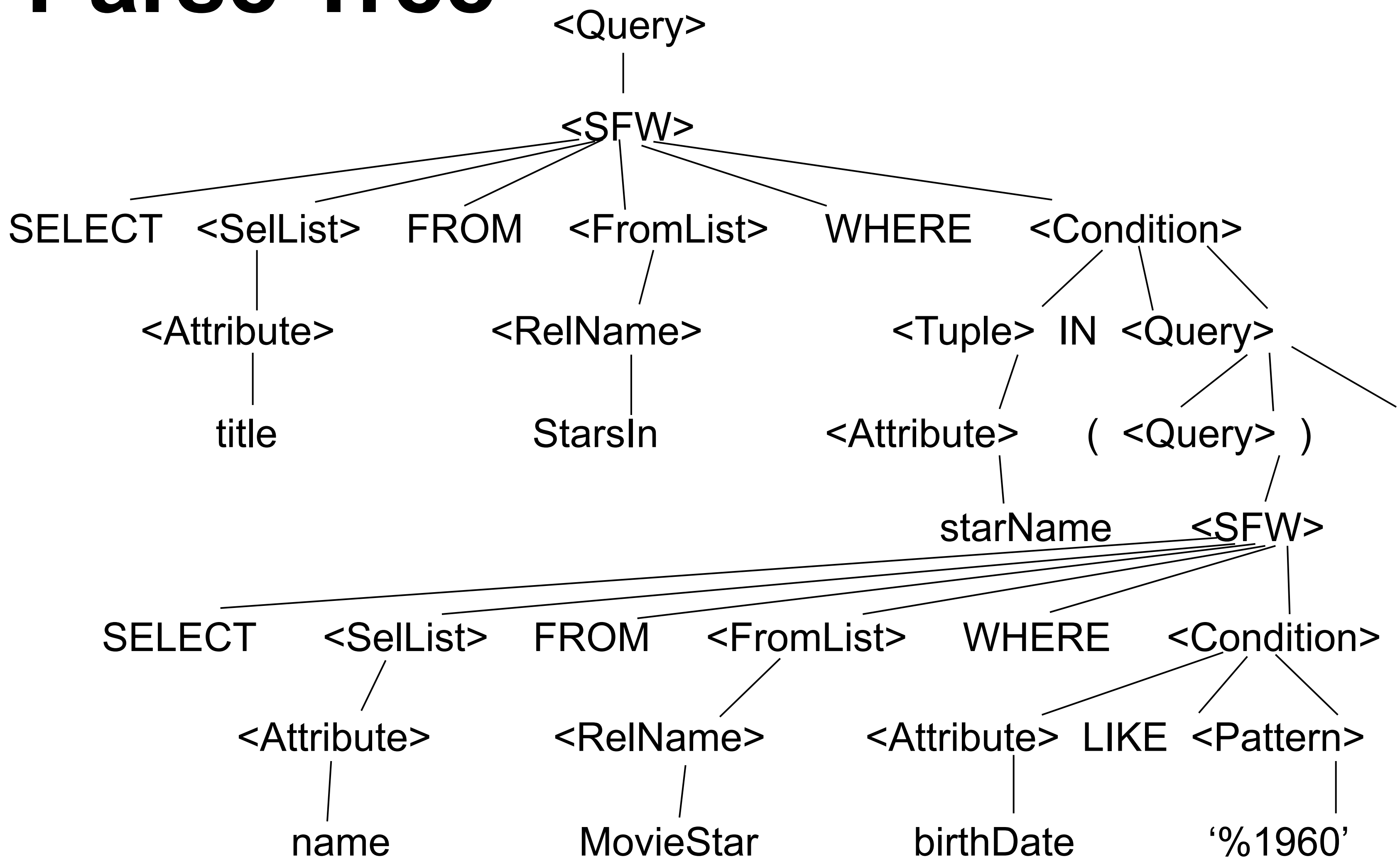


Example SQL Query

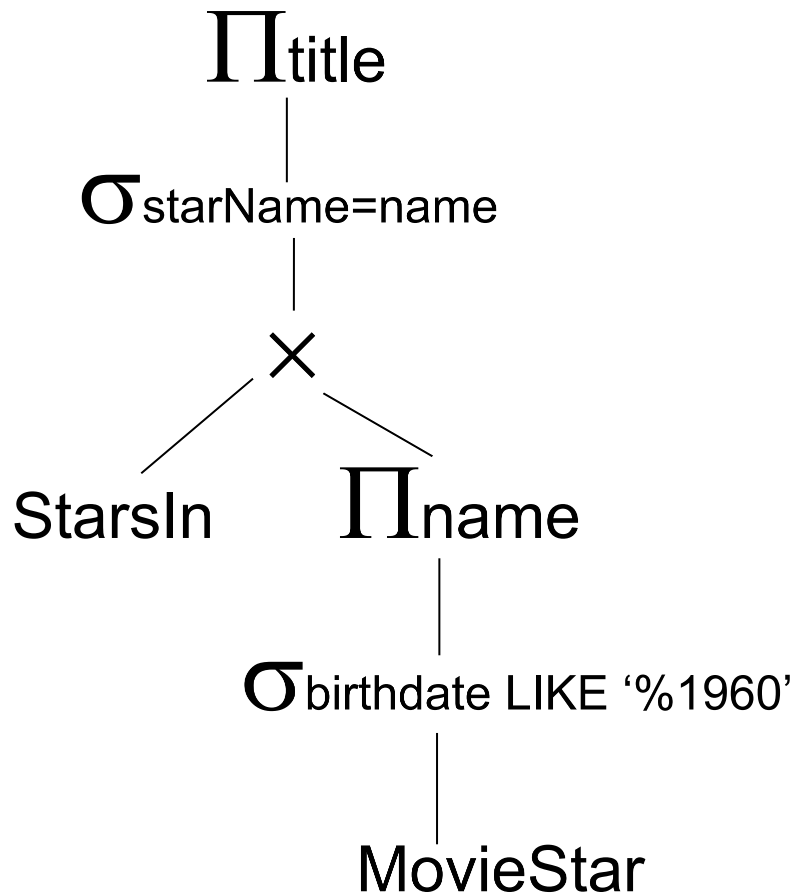
```
SELECT title
FROM StarsIn
WHERE starName IN (
      SELECT name
      FROM MovieStar
      WHERE birthdate LIKE ‘%1960’
);
```

(Find the movies with stars born in 1960)

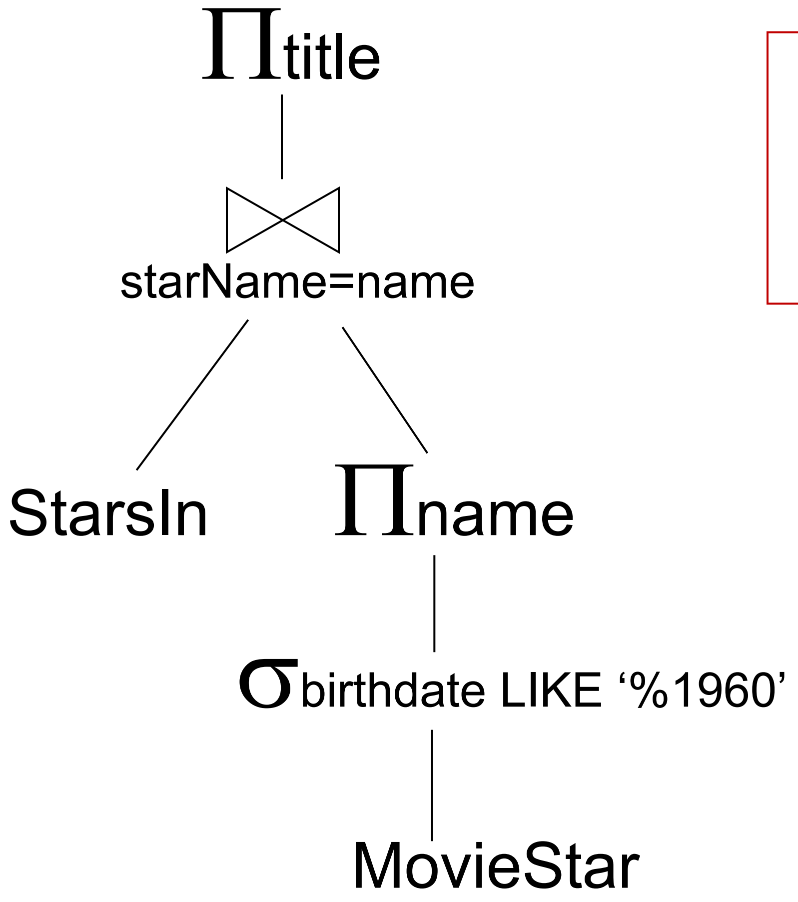
Parse Tree


<Query>
<SFW>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attribute>
<RelName>
<Tuple>
IN
<Query>
title
StarsIn
<Attribute>
(
<Query>
)
starName
<SFW>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attribute>
<RelName>
<Attribute>
LIKE
<Pattern>
name
MovieStar
birthDate
‘%1960’

Logical Query Plan

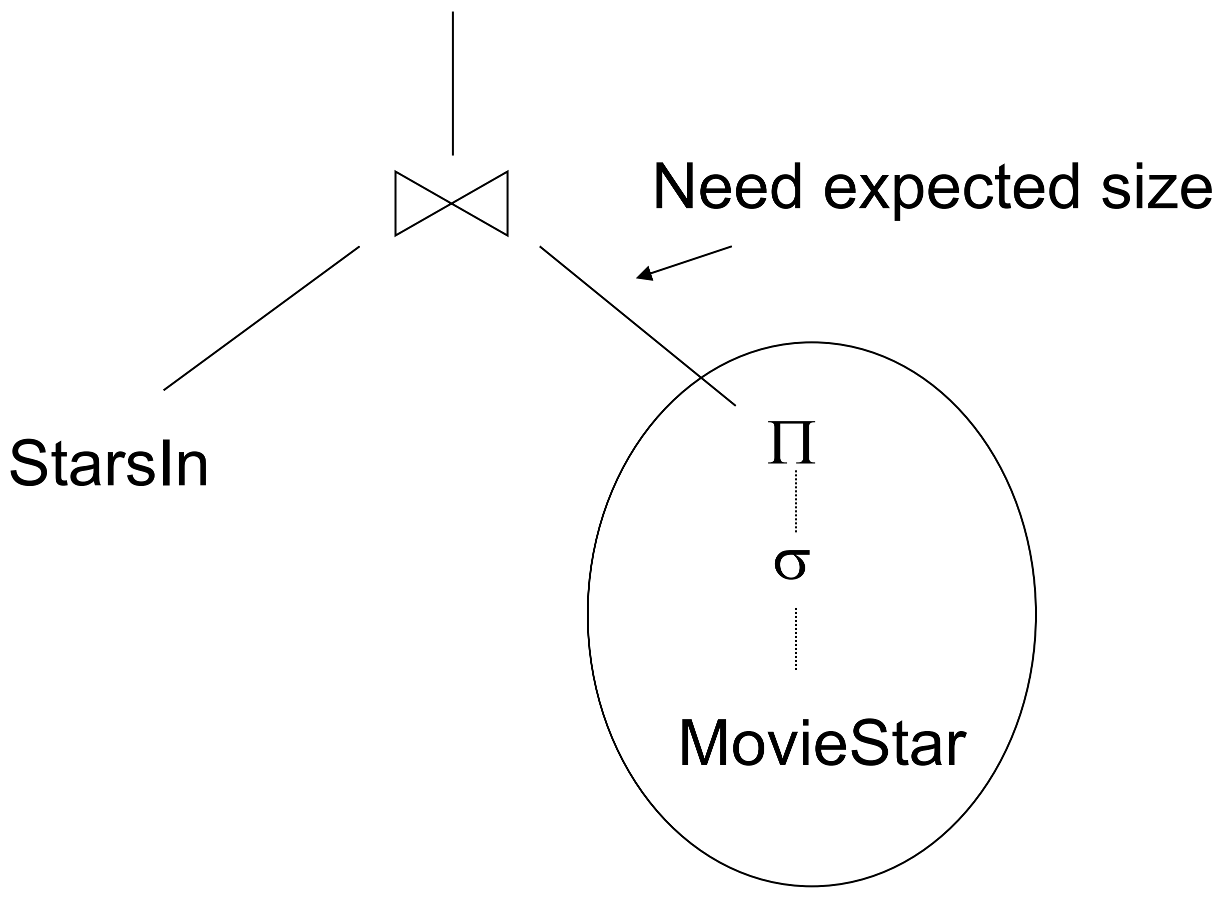


Improved Logical Query Plan

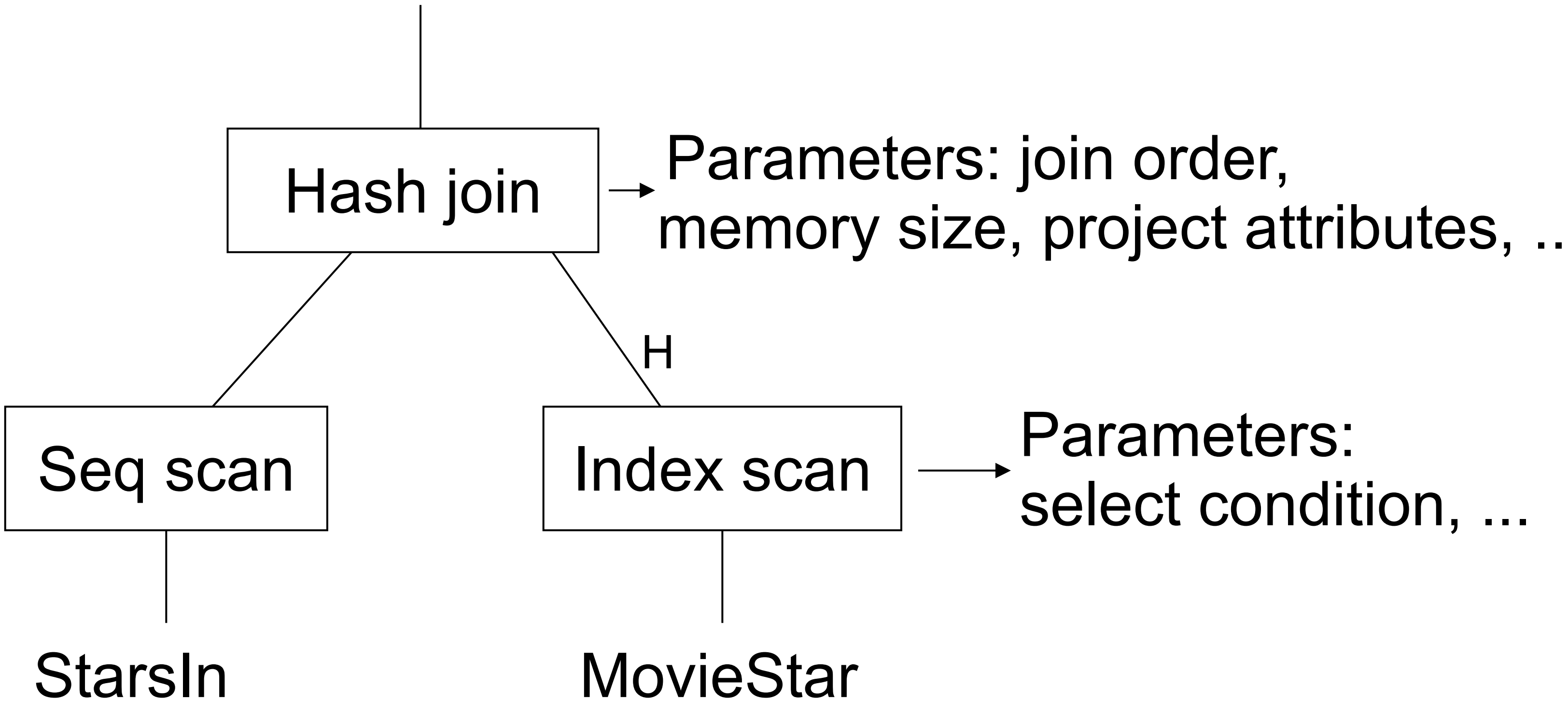


Question:
Push Π_{title} to StarsIn?

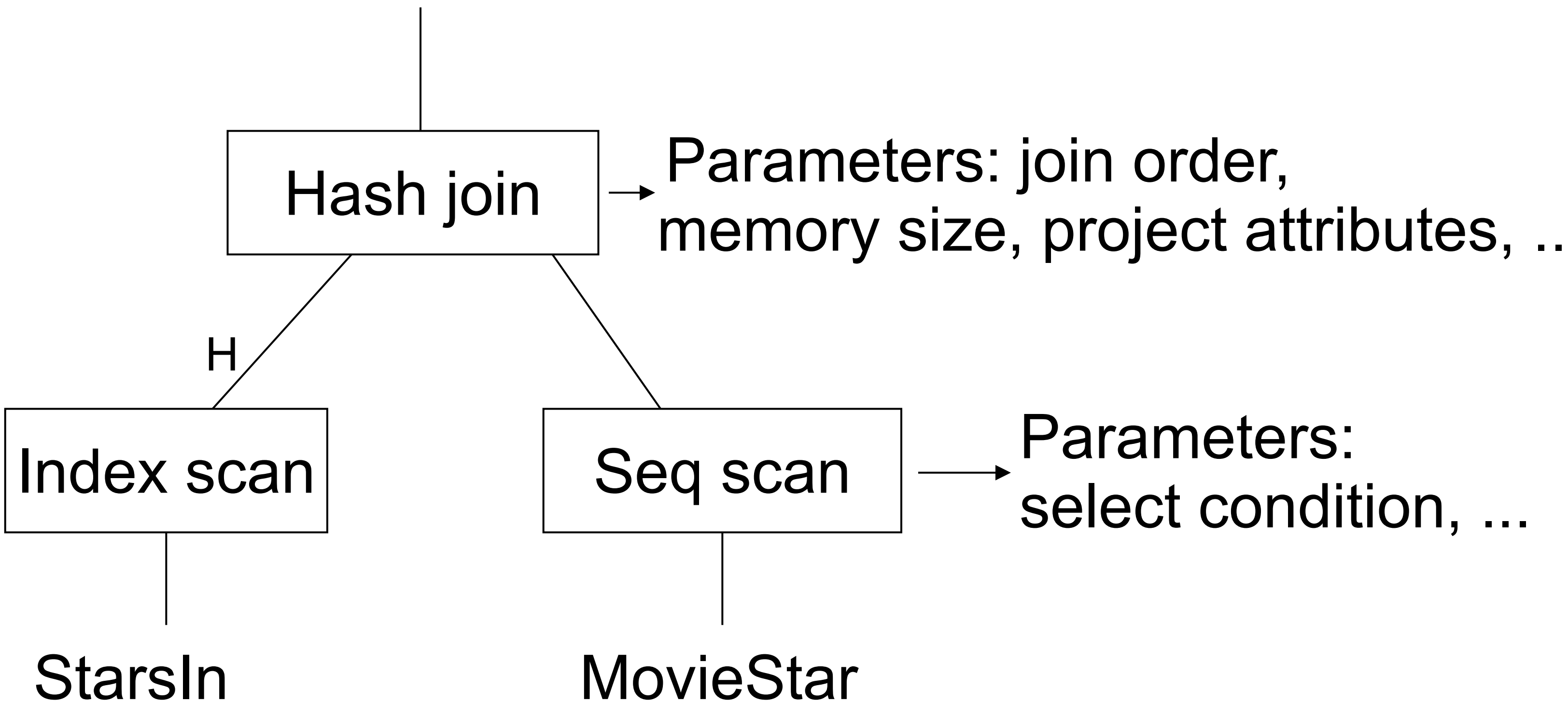
Estimate Result Sizes



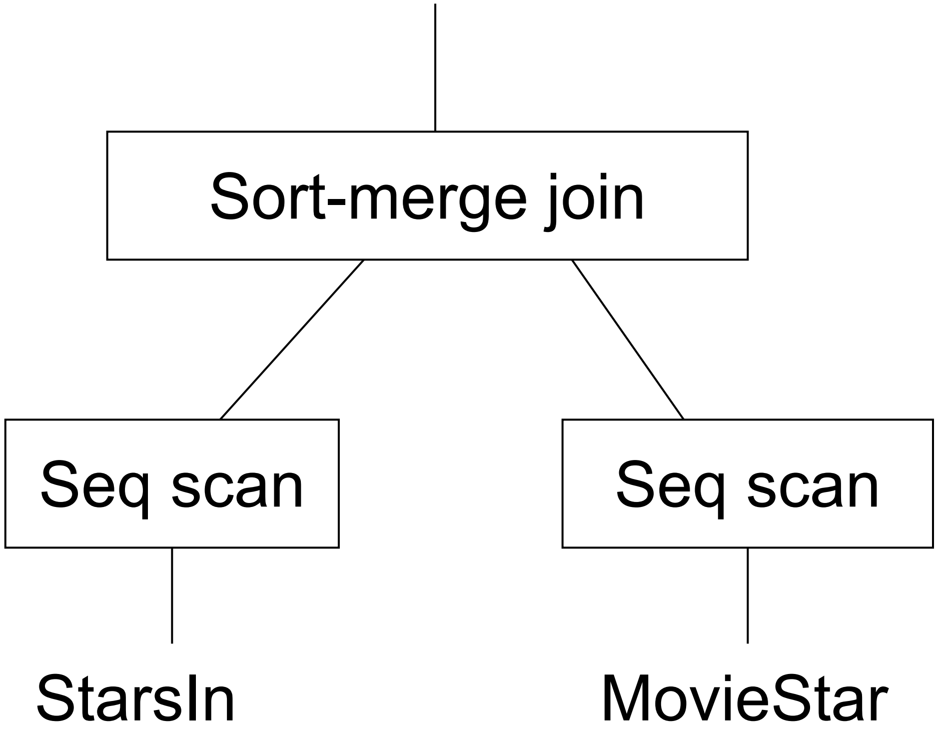
One Physical Plan



Another Physical Plan

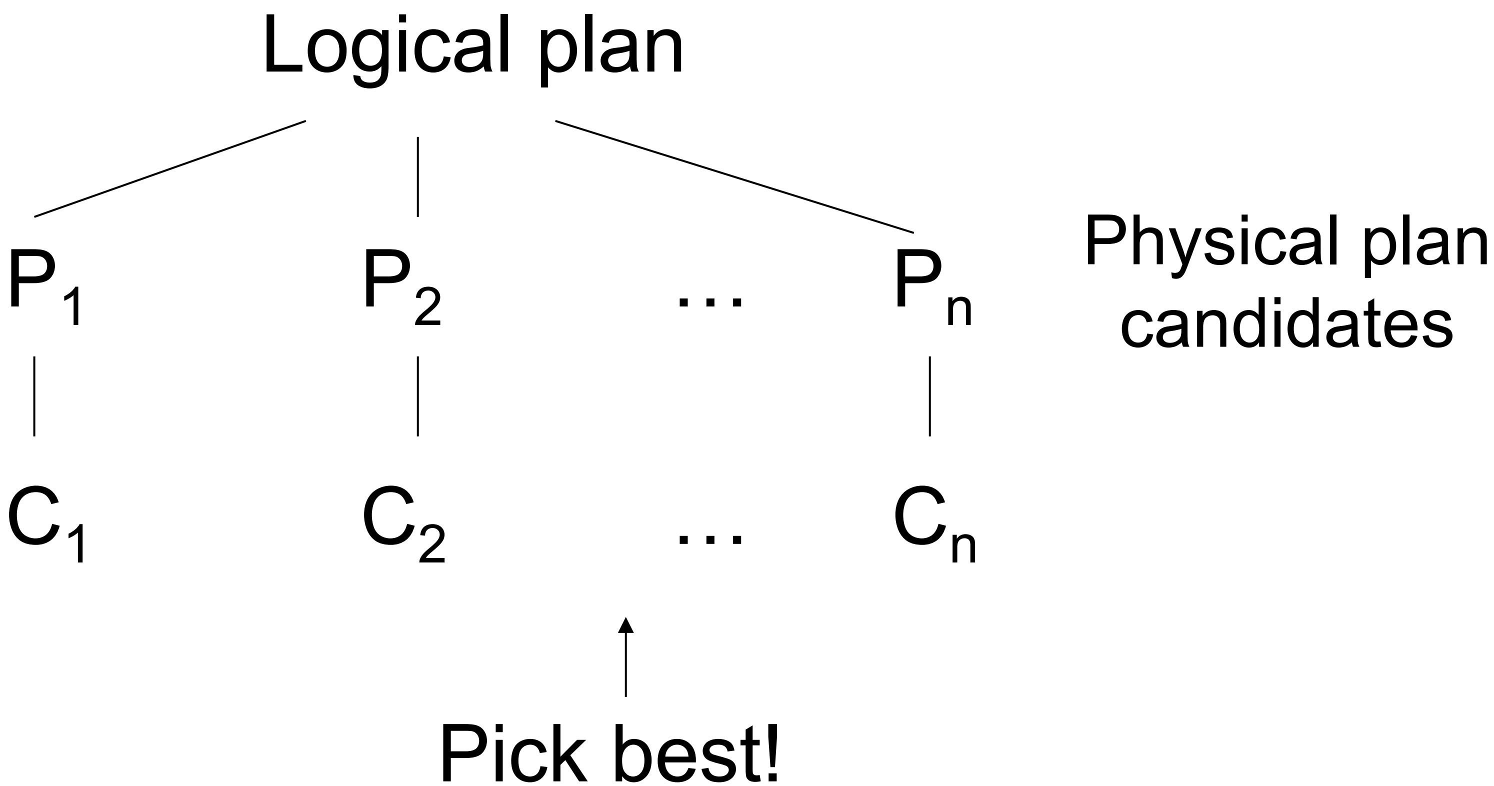


Another Physical Plan



Which plan is likely to be better?

Estimating Plan Costs



Covered in next few lectures!

Query Execution

Overview

Relational operators

Execution methods

Now That We Have a Plan, How Do We Run it?

Several different options that trade between complexity, setup time & performance

Example: Simple Query

```
SELECT quantity * price
  FROM orders
 WHERE productId = 75
```

$$\Pi_{\text{quanity*price}}(\sigma_{\text{productId=75}}(\text{orders}))$$

Method 1: Interpretation

```
interface Operator {
  Tuple next();
}

class TableScan: Operator {
  String tableName;
}

class Select: Operator {
  Operator parent;
  Expression condition;
}

class Project: Operator {
  Operator parent;
  Expression[] exprs;
}
```

```
interface Expression {
  Value compute(Tuple in);
}

class Attribute: Expression {
  String name;
}

class Times: Expression {
  Expression left, right;
}

class Equals: Expression {
  Expression left, right;
}
```

Example Expression Classes

```
class Attribute: Expression {
  String name;

  Value compute(Tuple in) {
    return in.getField(name);
  }
}

class Times: Expression {
  Expression left, right;

  Value compute(Tuple in) {
    return left.compute(in) * right.compute(in);
  }
}
```

probably better to use a numeric field ID instead

Example Operator Classes

```
class TableScan: Operator {
  String tableName;

  Tuple next() {
    // read & return next record from file
  }
}

class Project: Operator {
  Operator parent;
  Expression[] exprs;

  Tuple next() {
    tuple = parent.next();
    fields = [expr.compute(tuple) for expr in exprs];
    return new Tuple(fields);
  }
}
```

Running Our Query with Interpretation

```
ops = Project(
        expr = Times(Attr(“quantity”), Attr(“price”)),
        parent = Select(
          expr = Equals(Attr(“productId”), Literal(75)),
          parent = TableScan(“orders”)
        )
      );

while(true) {
  Tuple t = ops.next();
  if (t != null) {
    out.write(t);
  } else {
    break;
  }
}
```

recursively calls Operator.next() and Expression.compute()

Pros & cons of this approach?

Method 2: Vectorization

Interpreting query plans one record at a time is simple, but it's too slow

» Lots of virtual function calls and branches for each record (recall Jeff Dean's numbers)

Keep recursive interpretation, but make Operators and Expressions run on **batches**

Implementing Vectorization

```
class TupleBatch {
  // Efficient storage, e.g.
  // schema + column arrays
}

interface Operator {
  TupleBatch next();
}

class Select: Operator {
  Operator parent;
  Expression condition;
}

...
```

```
class ValueBatch {
  // Efficient storage
}

interface Expression {
  ValueBatch compute(
    TupleBatch in);
}

class Times: Expression {
  Expression left, right;
}

...
```

Typical Implementation

Values stored in columnar arrays (e.g. int[]) with a separate bit array to mark nulls

Tuple batches fit in L1 or L2 cache

Operators use SIMD instructions to update both values and null fields without branching

Pros & Cons of Vectorization

+ Faster than record-at-a-time if the query processes many records

+ Relatively simple to implement

– Lots of nulls in batches if query is selective

– Data travels between CPU & cache a lot

Method 3: Compilation

Turn the query into executable code

Compilation Example

$\Pi_{\text{quanity*price}} (\sigma_{\text{productId=75}} (\text{orders}))$

generated class with the right field types for orders table

```
class MyQuery {
  void run() {
    Iterator<OrdersTuple> in = openTable("orders");
    for(OrdersTuple t: in) {
      if (t.productId == 75) {
        out.write(Tuple(t.quantity * t.price));
      }
    }
  }
}
```

Can also theoretically generate vectorized code

Pros & Cons of Compilation

+ Potential to get fastest possible execution

+ Leverage existing work in compilers

– Complex to implement

– Compilation takes time

– Generated code may not match hand-written

What’s Used Today?

Depends on context & other bottlenecks

Transactional databases (e.g. MySQL): mostly record-at-a-time interpretation

Analytical systems (Vertica, Spark SQL): vectorization, sometimes compilation

ML libs (TensorFlow): mostly vectorization (the records *are* vectors!), some compilation