CSE373: Data Structures & Algorithms
Lecture 15: B-Trees

Linda Shapiro

Winter 2015

Announcements

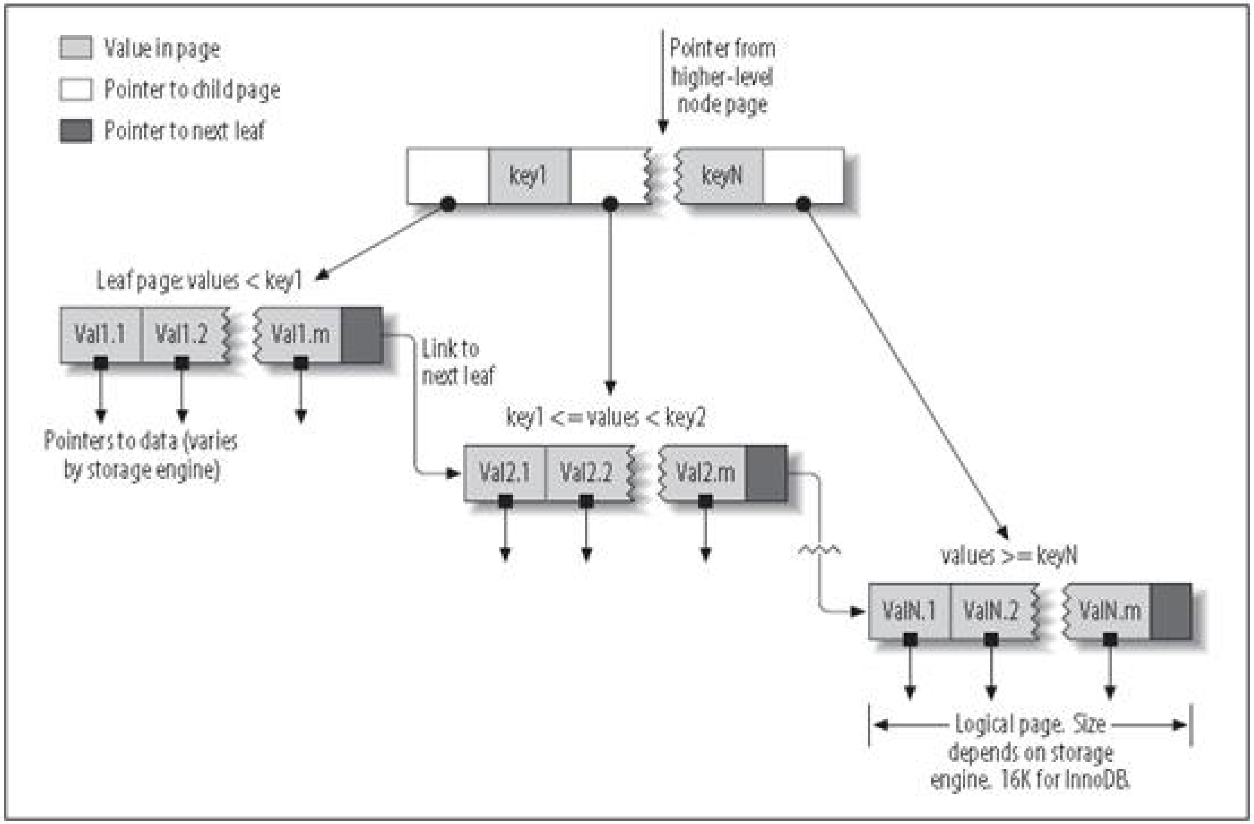
- HW05 will be out soon on hash tables. Evan is improving it.

- I am grading exams.

B-Trees Introduction

- B-Trees (and B+-Trees) are used heavily in databases.
- They are NOT binary trees.
- They are multi-way search trees that are kept somewhat shallow to limit disk accesses.

Example (Just the Idea)



Relational Databases

- A relational database is conceptually a set of 2D tables.
- The columns of a table are called attributes; they are the keys.
- Each table has at least one primary key by which it can be accessed rapidly.
- The rows are the different data records, each having a unique primary key.
- B+ trees are one very common implementation for these tables. In B+ trees, the data are stored only in the leaf nodes.

Creating a table in SQL

```
create table Company
    (cname varchar(20) primary key,
    country varchar(20),
    no_employees int,
    for_profit char(1));

insert into Company values ('GizmoWorks', 'USA', 20000,'y');
insert into Company values ('Canon', 'Japan', 50000,'y');
insert into Company values ('Hitachi', 'Japan', 30000,'y');
insert into Company values('Charity', 'Canada', 500,'n');
```

Company Table

primary key

cname	country	no_employees	for_profit
GizmoWorks	USA	20000	y
Canon	Japan	50000	y
Hitachi	Japan	30000	y
Charity	Canada	500	n

```
create table Company
    (cname varchar(20) primary key,
    country varchar(20),
    no_employees int,
    for_profit char(1));
```

Queries

- select * from Company;
- select cname from Company
  where no_employees = 500;
- select cname, country from Company
  where no_employees > 20000 AND
            no_employees < 50000;

B+-Trees

B+-Trees are multi-way search trees commonly used in database systems or other applications where data is stored externally on disks and keeping the tree shallow is important.

A B+-Tree of order M has the following properties:

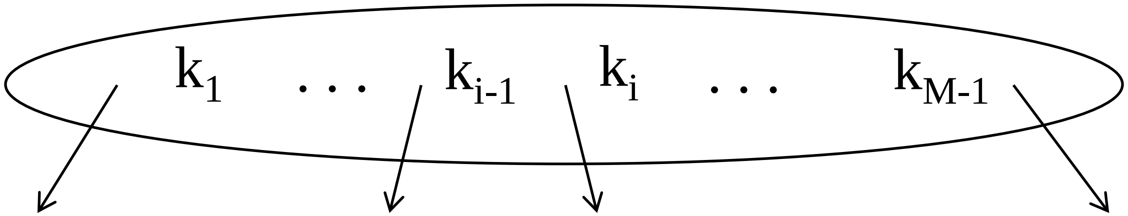
1. The root is either a leaf or has between 2 and M children.
2. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
3. All leaves are at the same depth.

All data records are stored at the **leaves**.
Internal nodes have "keys" guiding to the leaves.
Leaves store between $\lceil L/2 \rceil$ and L data records,
where L can be equal to M (default) or can be different.

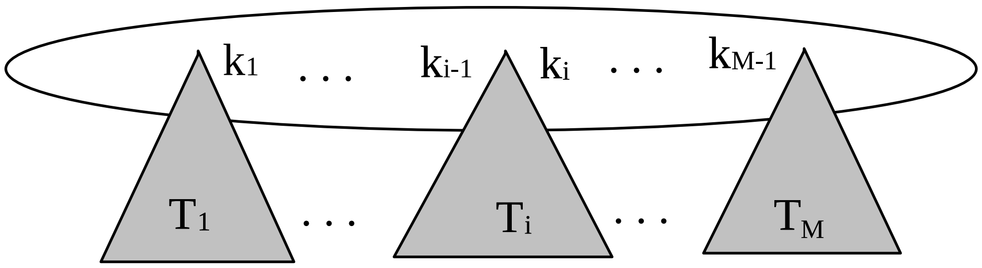
B+-Tree Details

Each (non-leaf) internal node of a B-tree has:

- Between $\lceil M/2 \rceil$ and M children.
- up to M-1 keys $k_1 < k_2 < ... < k_{M-1}$



Properties of B+-Trees



Children of each internal node are "between" the keys in that node.
Suppose subtree T_i is the *i*th child of the node:

all keys in T_i must be between keys k_{i-1} and k_i

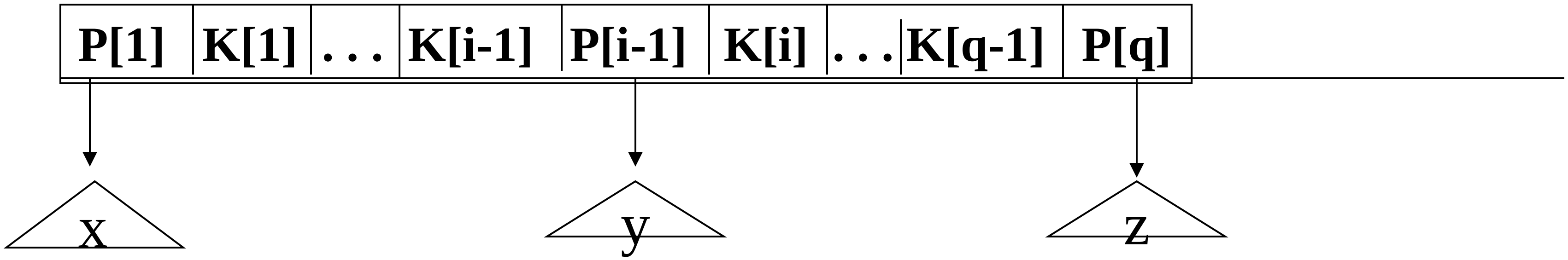
i.e. $k_{i-1} \le T_i < k_i$

k_{i-1} is the smallest key in T_i

All keys in first subtree $T_1 < k_1$

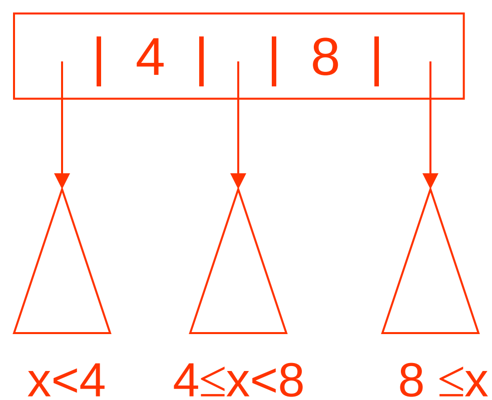
All keys in last subtree $T_M \ge k_{M-1}$

B-Tree Nonleaf Node in More Detail



x < K[1]   K[i-1]≤y<K[i]   K[q-1] ≤ z

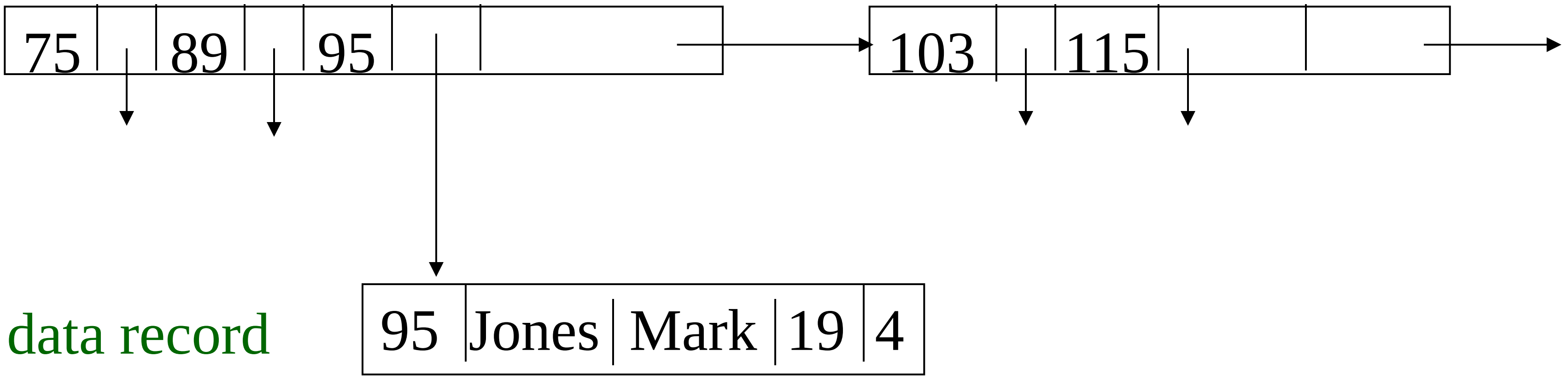
- The Ks are keys
- The Ps are pointers to subtrees.



Detailed Leaf Node Structure (B+ Tree)

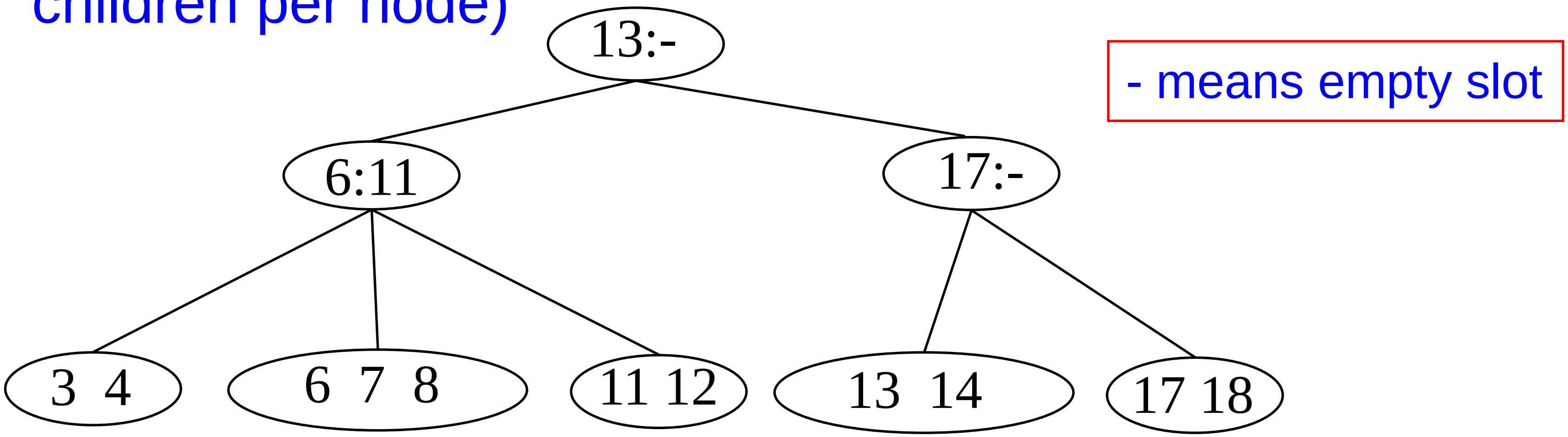
K[1]	R[1]	. . .	K[q-1]	R[q-1]	Next

- The Ks are keys (assume unique).
- The Rs are pointers to **records** with those keys.
- The **Next** link points to the next leaf in key order (B+-tree).



Searching in B-trees

- B-tree of order 3: also known as 2-3 tree (2 to 3 children per node)



- Examples: Search for 9, 14, 12

Searching a B-Tree T for a Key Value K
(from a database book)

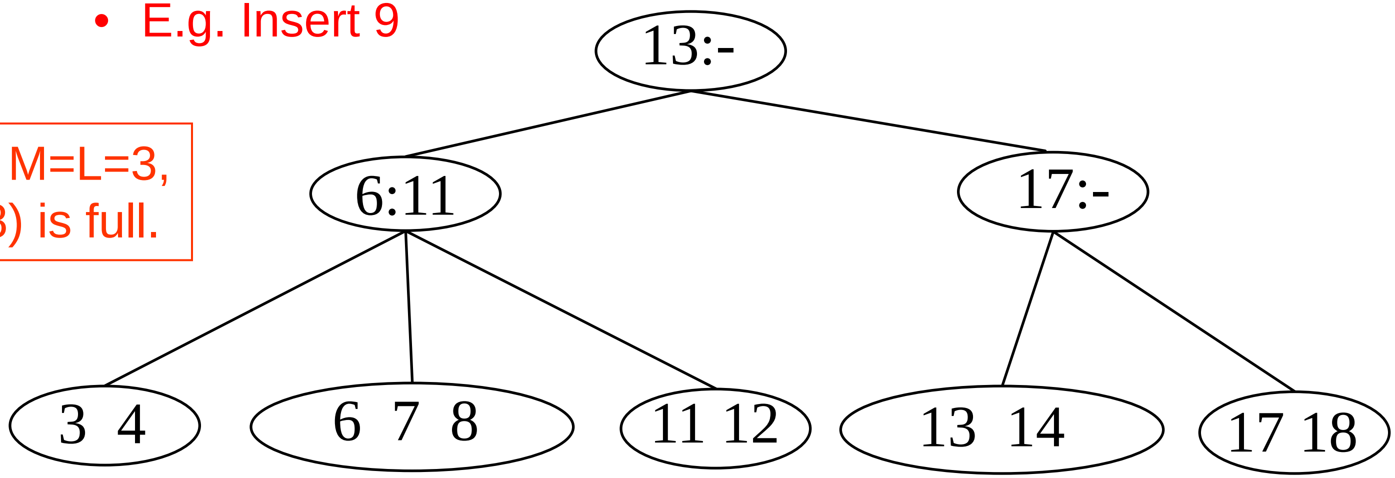
```
Find(ElementType K, Btree T) {
B = T;
while (B is not a leaf)
   {
   find the Pi in node B that points to
     the proper subtree that K will be in;

   B = Pi;
   }

/* Now we're at a leaf */
if key K is the jth key in  leaf B,
   use the jth record pointer to find the
   associated record;
else /* K is not in leaf B */ report failure;
}
```

How would you search for a key in a node?

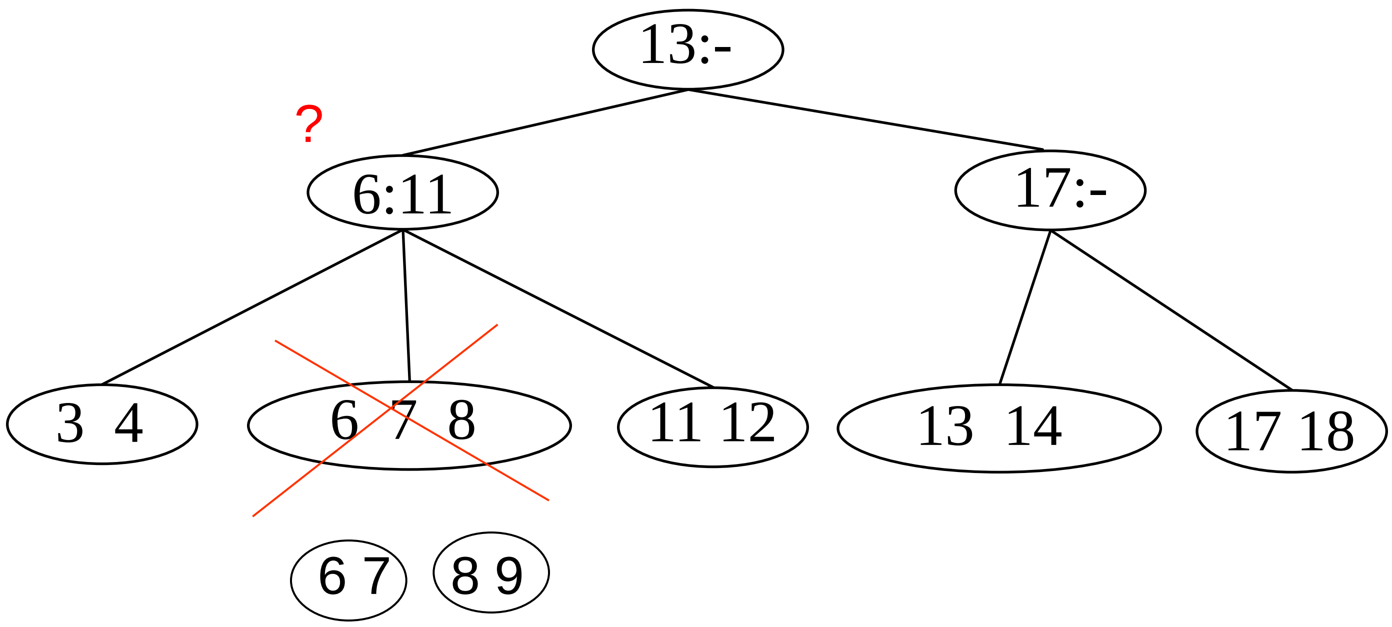
Inserting into B-Trees
The Idea

- Insert X: Do a Find on X and find appropriate leaf node
  - › If leaf node is not full, fill in empty slot with X
    - E.g. Insert 5
  - › If leaf node is full, split leaf node and adjust parents up to root node
    - E.g. Insert 9

Assume M=L=3,
so (6 7 8) is full.



Inserting into B-Trees The Idea



Inserting a New Key in a B-Tree of Order M (and L=M)

from database book

```
Insert(ElementType K, Btree B)  {
 find the leaf node LB of B in which K belongs;
 if notfull(LB) insert K into LB;
 else
    {
    split LB into two nodes LB and LB2 with
     j = ⌊(M+1)/2⌋ keys in LB and the rest in LB2;
```

LB LB2

K[1] R[1] . . . K[j] R[j] → K[j+1] R[j+1] . . . K[M+1] R[M+1] →

```
      if ( IsNull(Parent(LB)) )
         CreateNewRoot(LB, K[j+1], LB2);
      else
         InsertInternal(Parent(LB), K[j+1], LB2);
     } }
```

Inserting a (Key,Ptr) Pair into an Internal Node

If the node is not full, insert them in the proper place and return.

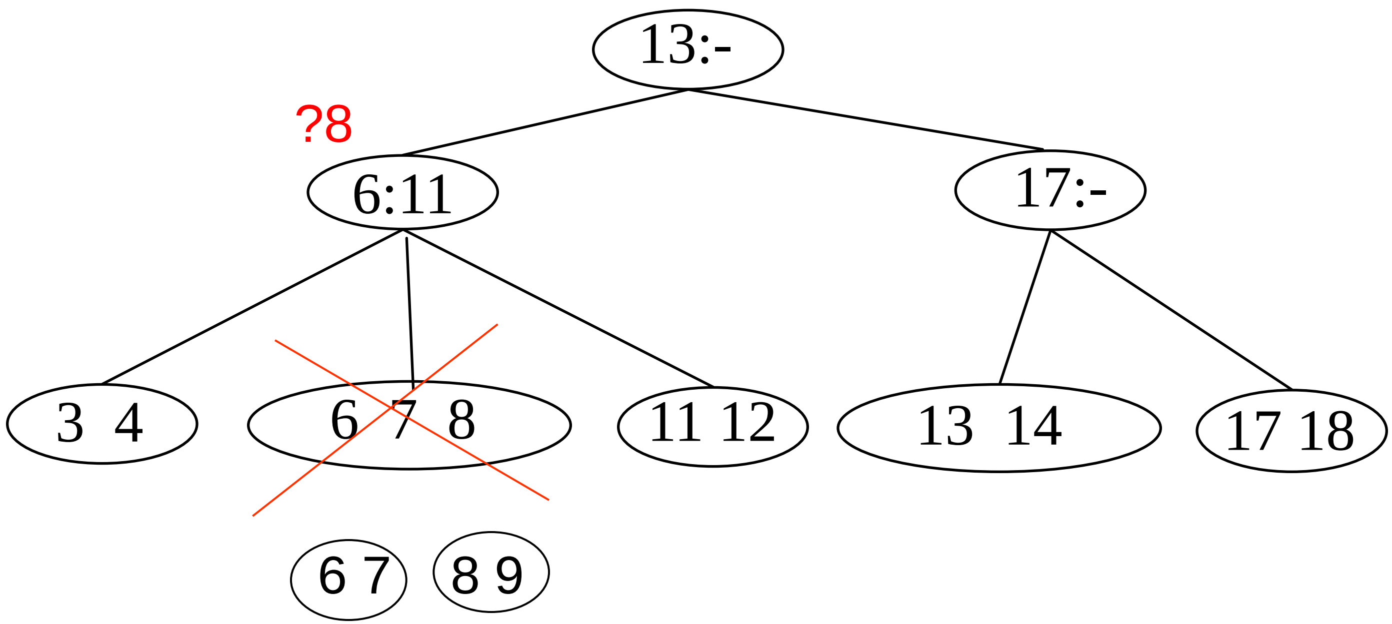
If the node is already full (M pointers, M-1 keys), find the place for the new pair and split the adjusted (Key,Ptr) sequence into two internal nodes with

$j = \lfloor (M+1)/2 \rfloor$ pointers and j-1 keys in the first,

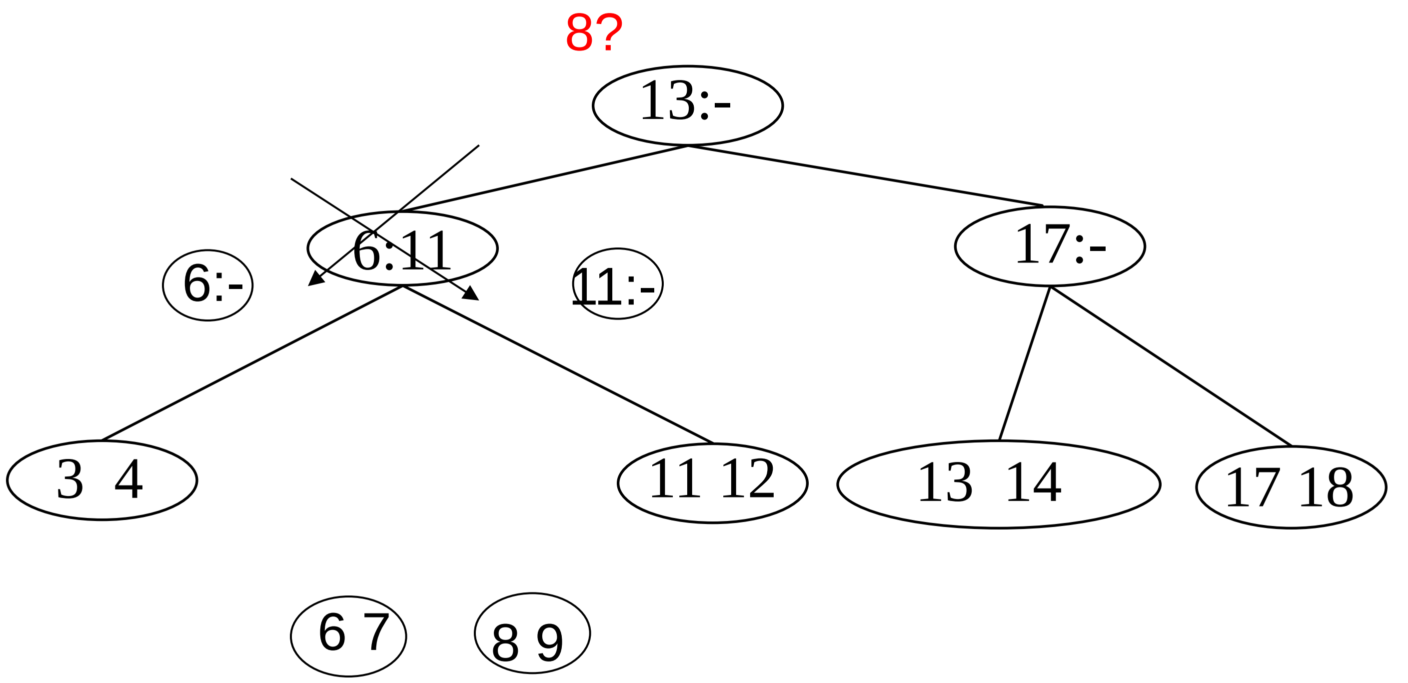
the next key is inserted in the node's parent,

and the rest in the second of the new pair.

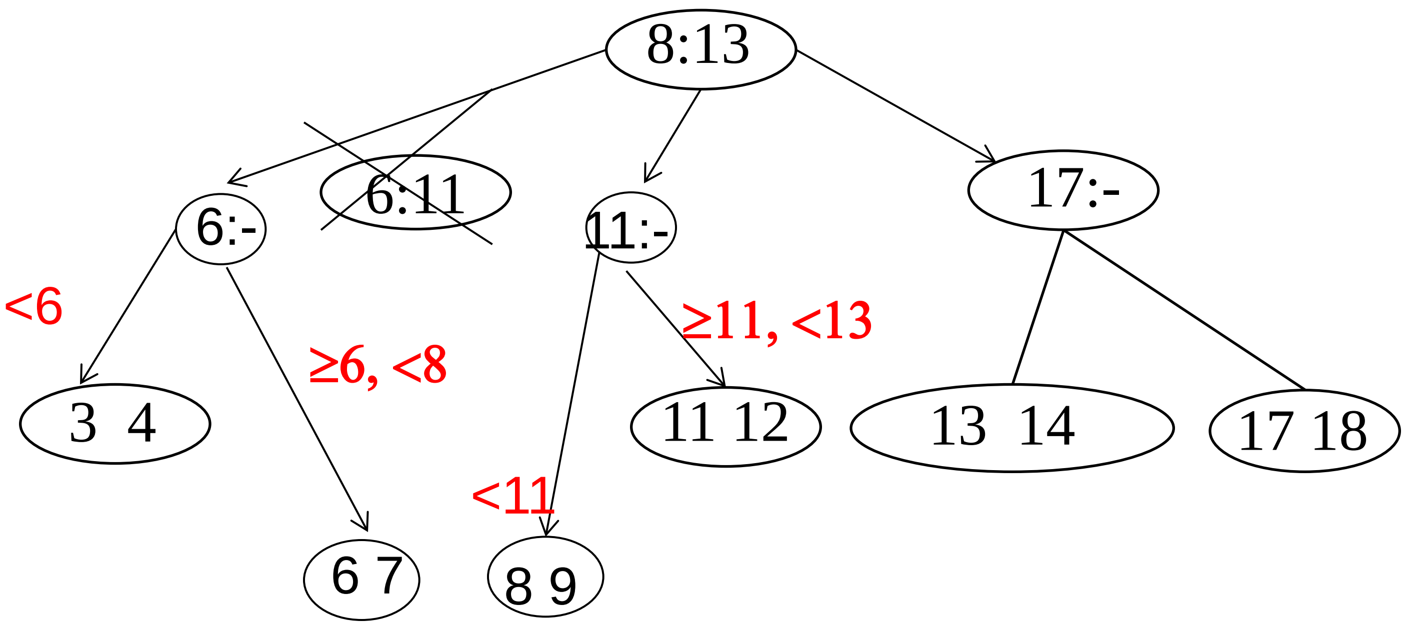
Inserting into B-Trees
The Idea



Inserting into B-Trees
The Idea

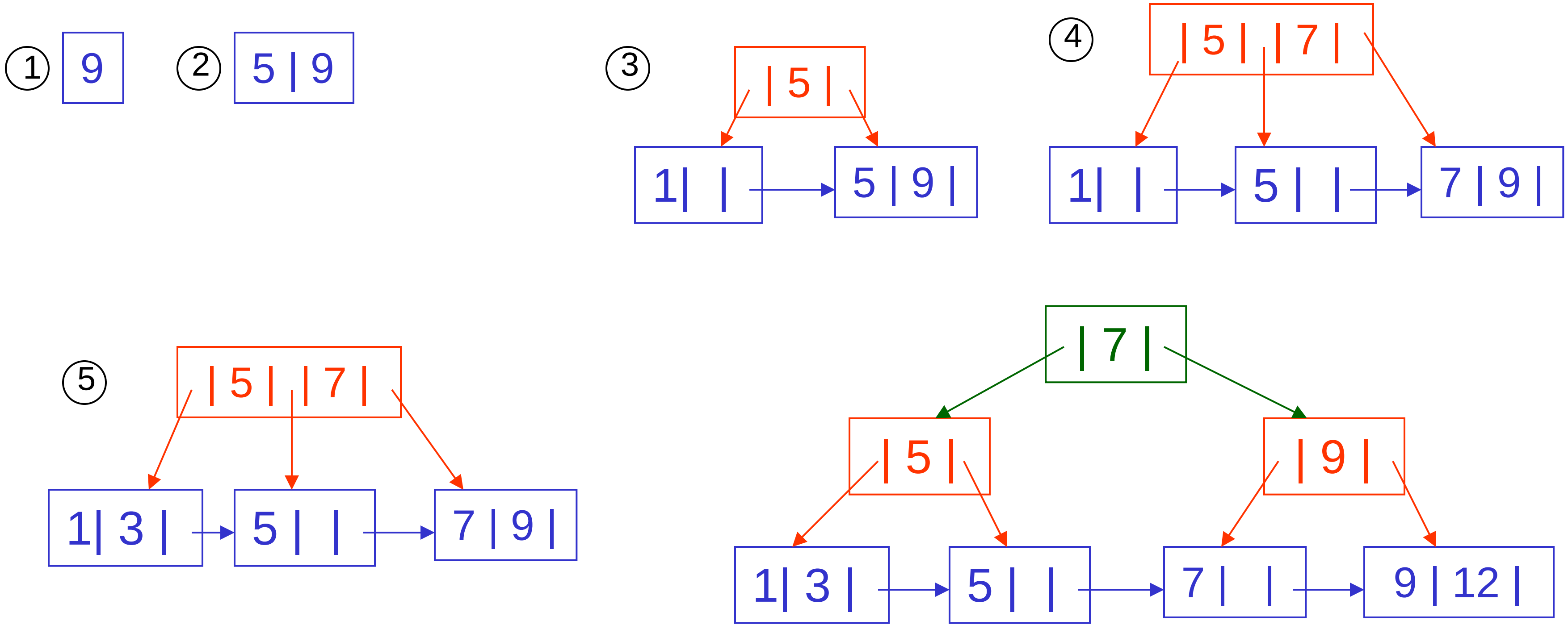


Inserting into B-Trees
The Idea



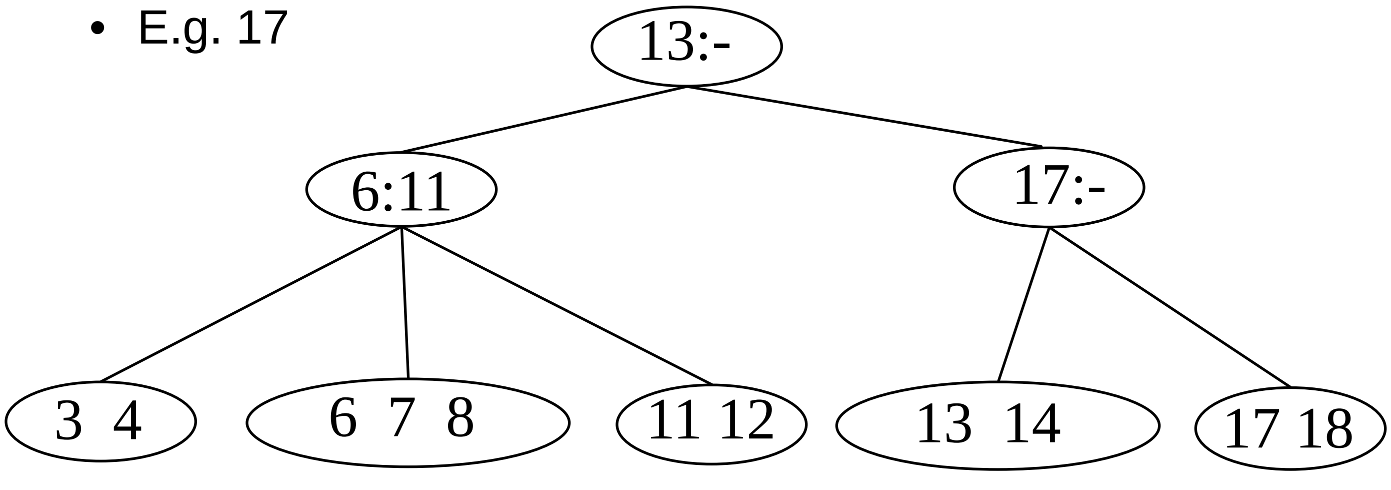
Example of Insertions into a B+ tree with M=3, L=2

Insertion Sequence: 9, 5, 1, 7, 3,12



Deleting From B-Trees
(NOT THE FULL ALGORITHM)

- Delete X : Do a find and remove from leaf
  - Leaf underflows – borrow from a neighbor
    - E.g. 11
  - Leaf underflows and can't borrow – merge nodes, delete parent
    - E.g. 17



Run Time Analysis of B-Tree Operations

- For a B-Tree of order M
  - Each internal node has up to M-1 keys to search
  - Each internal node has between $\lceil M/2 \rceil$ and M children
  - Depth of B-Tree storing N items is $O(\log_{\lceil M/2 \rceil} N)$
- Find: Run time is:
  - O(log M) to binary search which branch to take at each node. But M is small compared to N.
  - Total time to find an item is O(depth*log M) = O(log N)

How Do We Select the Order M?

- In internal memory, small orders, like 3 or 4 are fine.

- **On disk**, we have to worry about the number of disk accesses to search the index and get to the proper leaf.

Rule: Choose the largest M so that an internal node can fit into one physical block of the disk.

This leads to typical M's between 32 and 256
And keeps the trees as shallow as possible.

Summary of B+-Trees

- Problem with Binary Search Trees: Must keep tree balanced to allow fast access to stored items

- Multi-way search trees (e.g. B-Trees and B+-Trees):
  - More than two children per node allows shallow trees; all leaves are at the same depth.
  - Keeping tree balanced at all times.
  - Excellent for indexes in database systems.