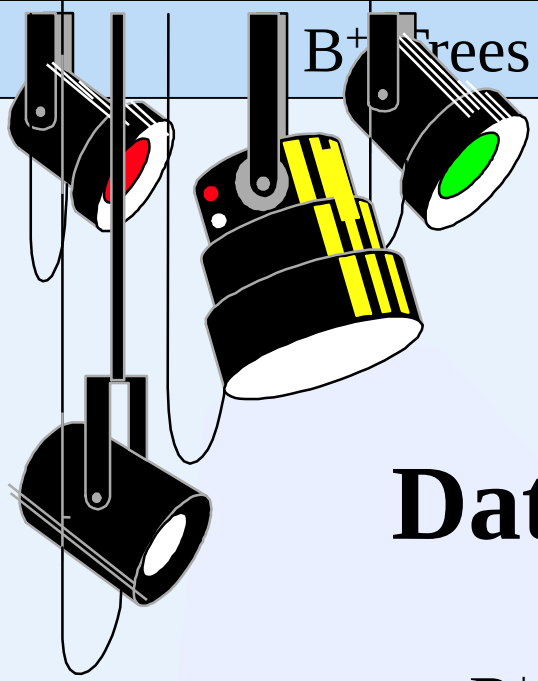
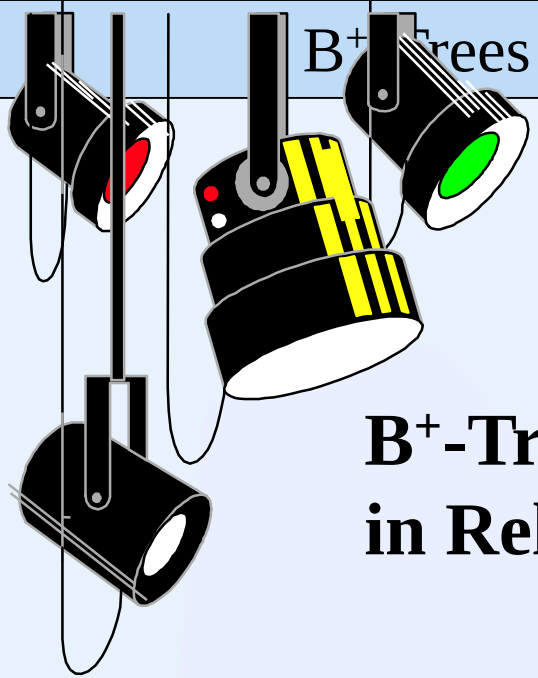


Database Index Techniques

- B^+ - tree
- Multiple-key indexes
- kd – tree
- Quad - tree
- R – tree
- Bitmap
- Inverted files

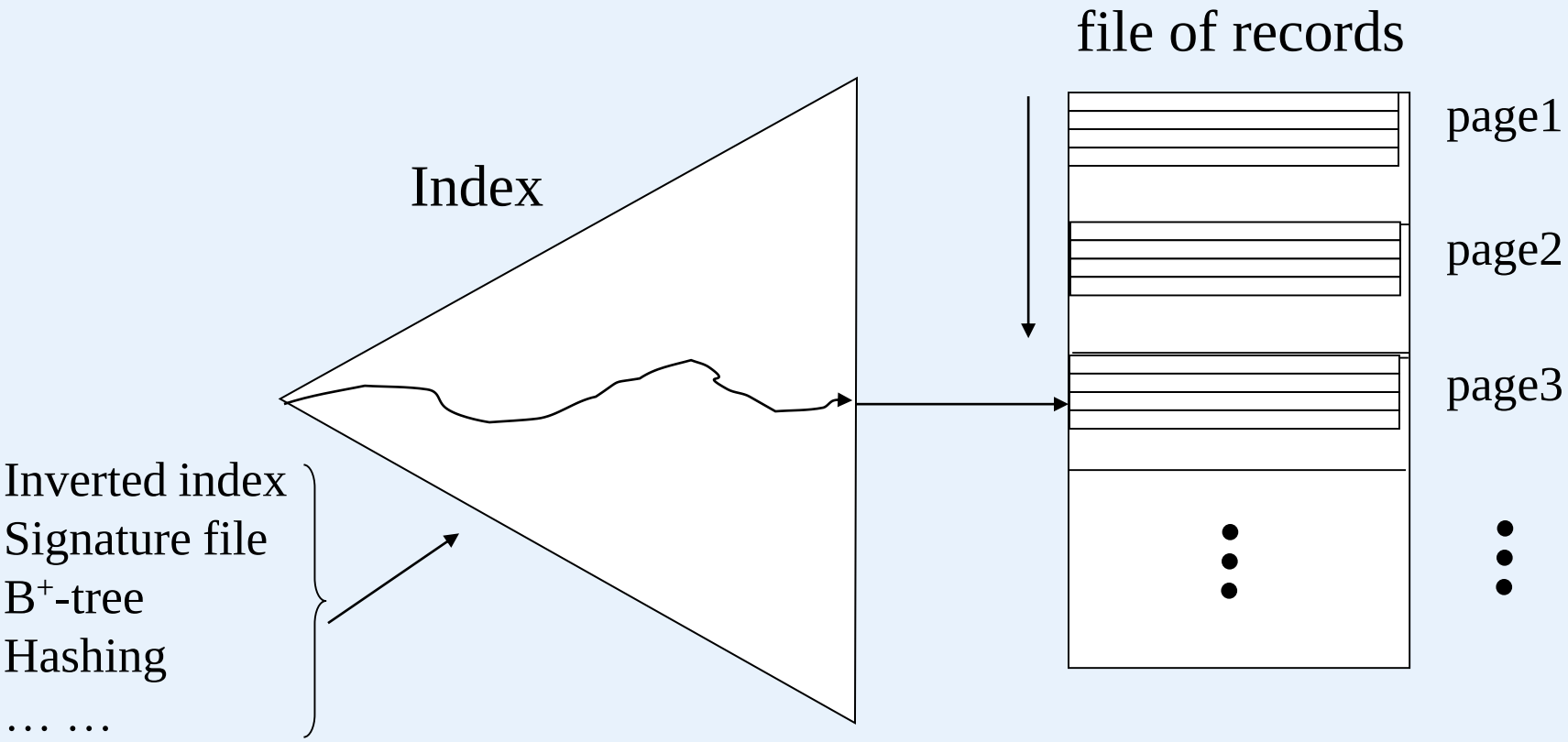


B+-Tree Construction and Record Searching in Relational DBs

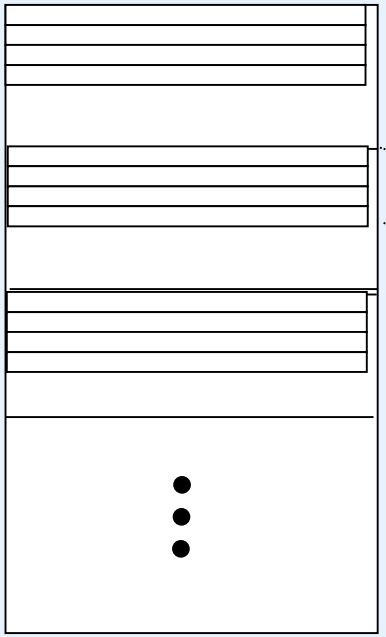
- Motivation
- What is a B+-tree?
- Construction of a B+-tree
- Search with a B+-tree
- B+-tree Maintenance

Motivation

- Scanning a file is time consuming.
- B+-tree provides a short access path.



file of records

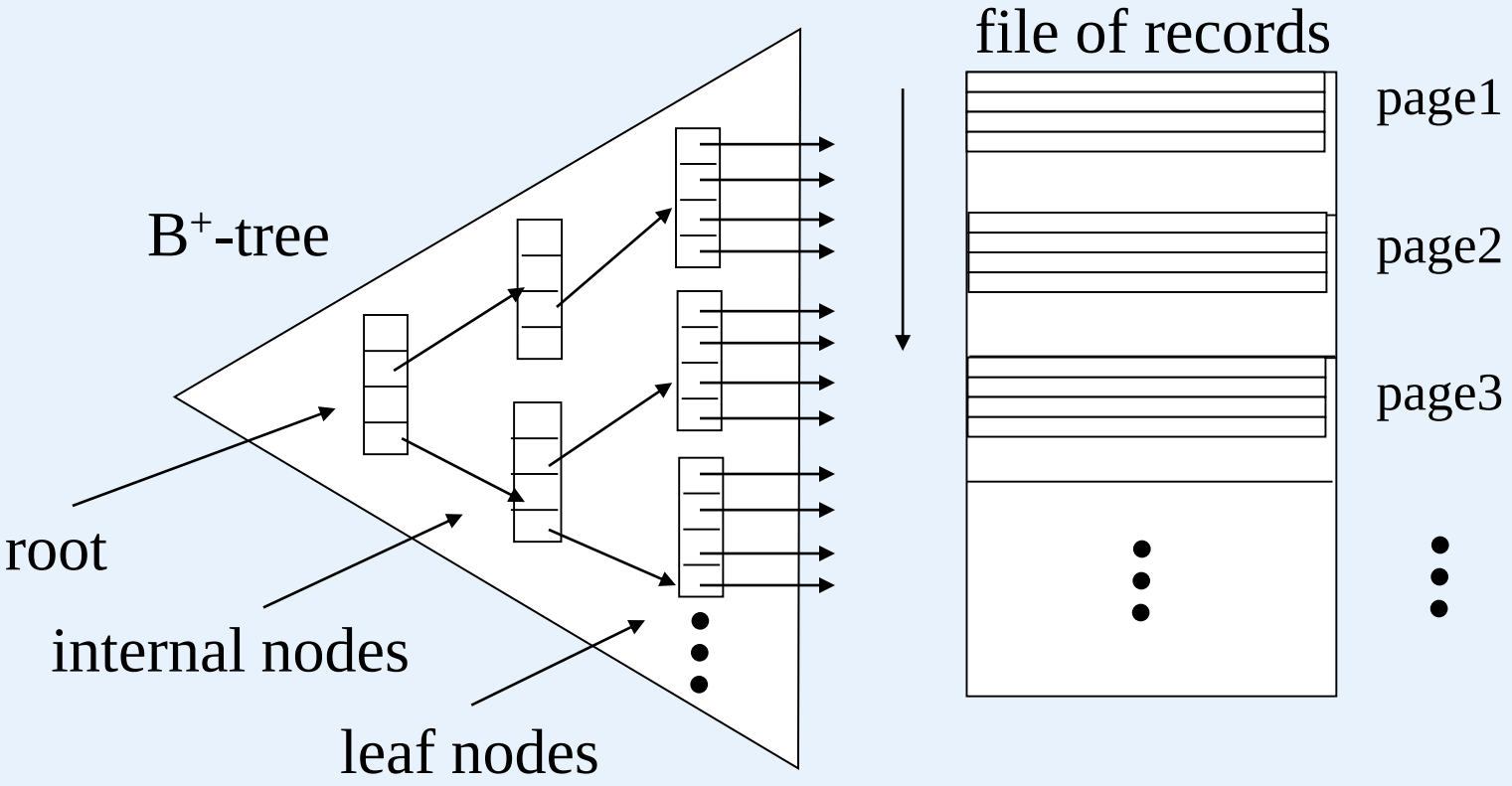


Employee

ename	ssn	bdate	address	dnumber
Aaron, Ed				
Abbott, Diane				
Adams, John				
Adams, Robin				

Motivation

- A B+-tree is a tree, in which each node is a page.
- The B+-tree for a file is stored in a separate file.



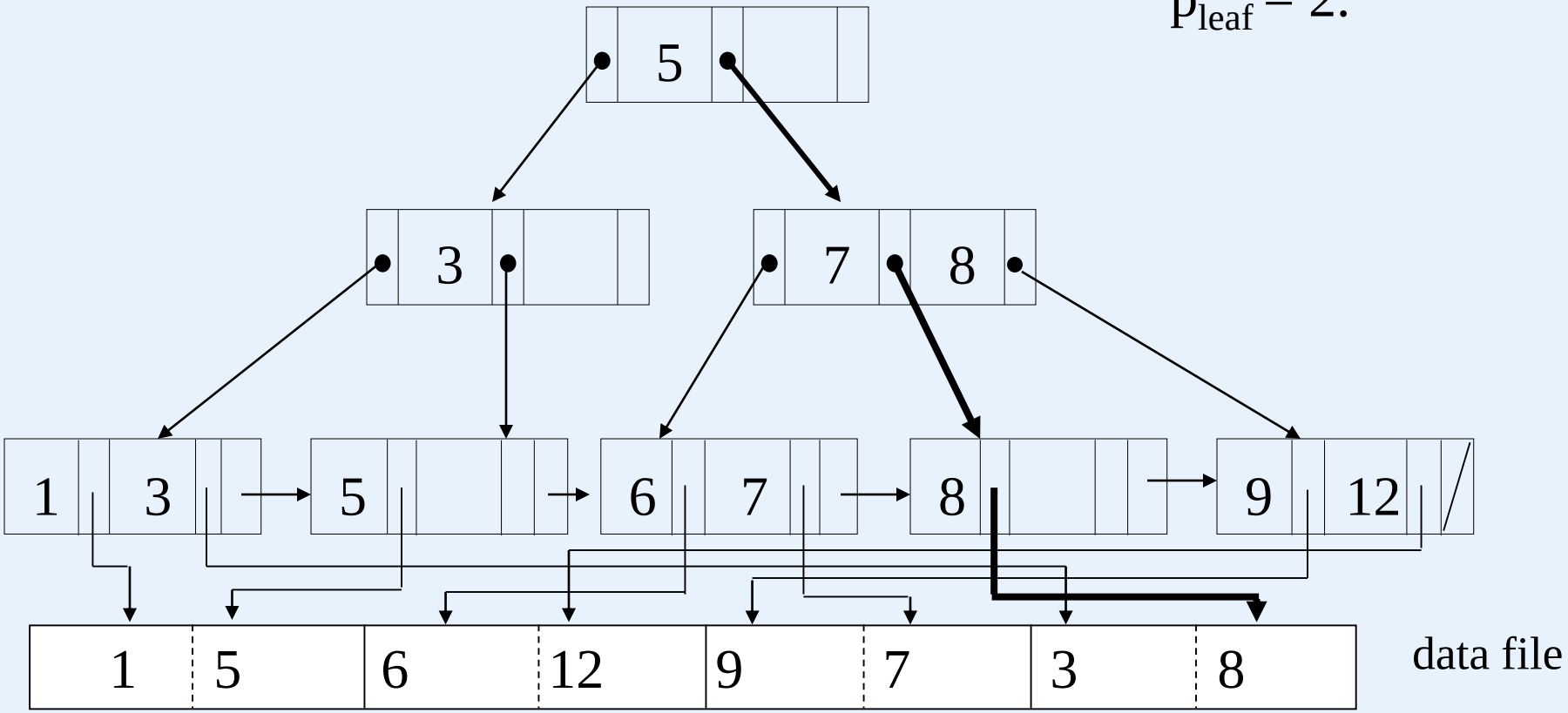
B+-tree Structure

non-leaf node (internal node or a root)

- $< P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q >$ $(q \le p_{internal})$
- $K_1 < K_2 < ... < K_{q-1}$ (i.e. it's an ordered set)
- For any key value, X, in the subtree pointed to by P_i
  - $K_{i-1} < X \le K_i$ for $1 < i < q$
  - $X \le K_1$ for $i = 1$
  - $K_{q-1} < X$ for $i = q$
- Each internal node has at most $p_{internal}$ pointers.
- Each node except root must have at least $\lceil p_{internal}/2 \rceil$ pointers.
- The root, if it has some children, must have at least 2 pointers.

A B+-tree

$p_{internal} = 3$,
$p_{leaf} = 2$.



B+-tree Structure

leaf node (terminal node)

- $< (K_1, Pr_1), (K_2, Pr_2), \ldots, (K_{q-1}, Pr_{q-1}), P_{next} >$
- $K_1 < K_2 < ... < K_{q-1}$
- Pr_i points to a record with key value K_i, or Pr_i points to a page containing a record with key value K_i.
- Maximum of p_{leaf} key/pointer pairs.
- Each leaf has at least $\lceil p_{leaf}/2 \rceil$ keys.
- All leaves are at the same level (balanced).
- P_{next} points to the next leaf node for key sequencing.

B+-tree Construction

- **Inserting key values into nodes**
- **Node splitting**

  - Leaf node splitting

  - Internal node splitting

  - Node generation

B+-tree Construction

- **Inserting key values into nodes**

  Example:

  Diane, Cory, Ramon, Amy, Miranda, Ahmed,

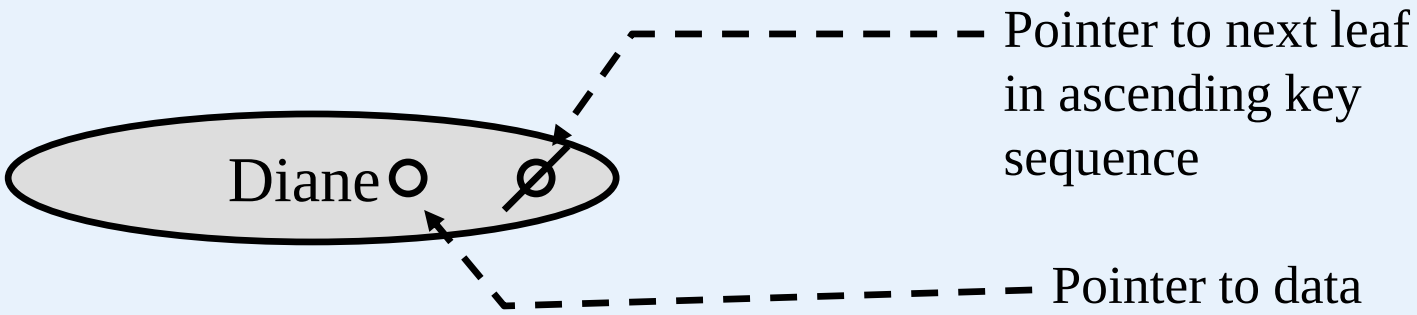
  Marshall, Zena, Rhonda, Vincent, Mary

  B^+-tree with $p_{internal} = p_{leaf} = 3$.

  Internal node will have minimum 2 pointers and maximum 3 pointers - inserting a fourth will cause a split.

  Leaf can have at least 2 key/pointer pairs and a maximum of 3 key/pointer pairs - inserting a fourth will cause a split.

insert Diane

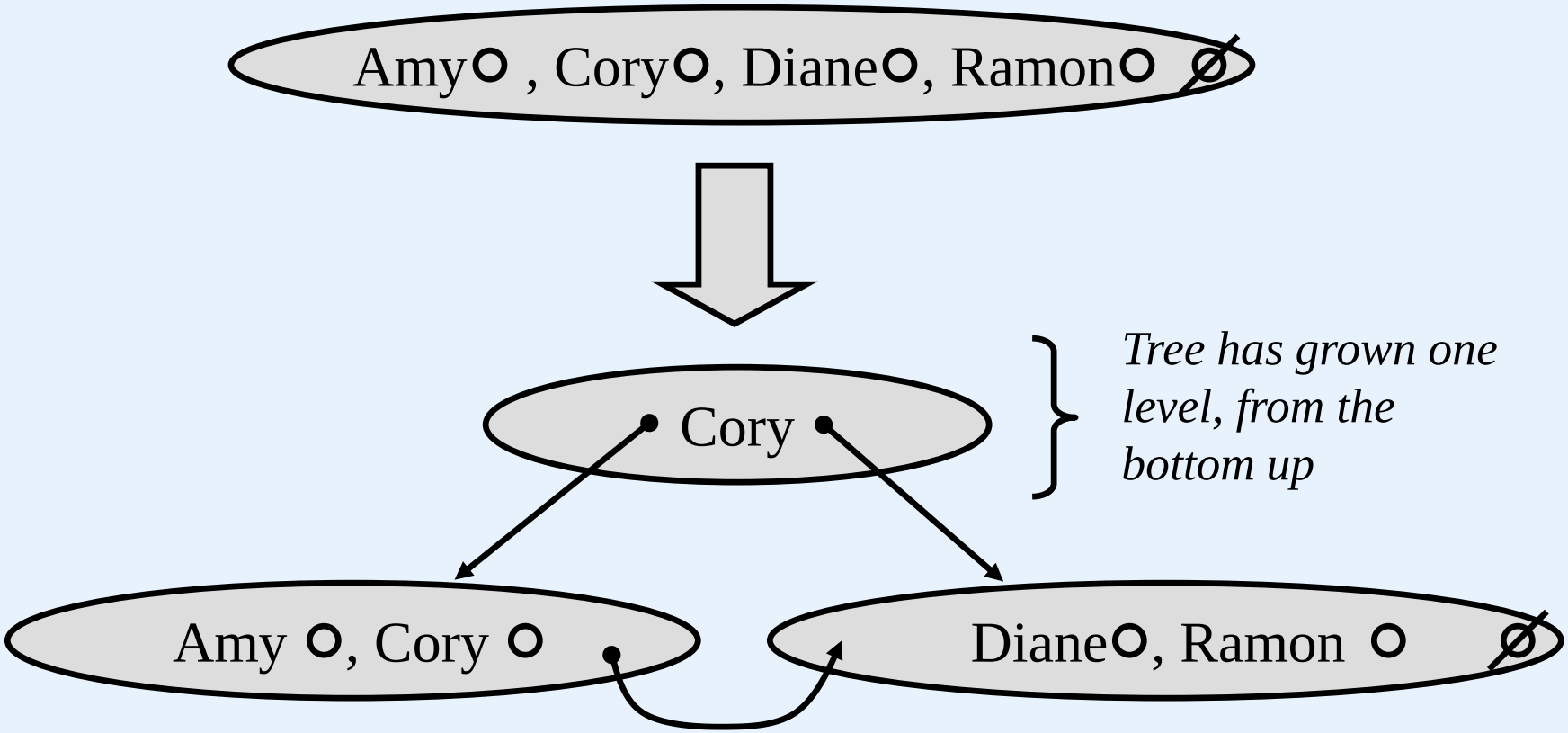


insert Cory

insert Ramon

inserting Amy will cause the node to overflow:

Continuing with insertion of Amy - split the node and promote a key value upwards (this must be Cory because it's the highest key value in the left subtree)



•**Splitting Nodes**

There are three situations to be concerned with:

- a leaf node splits,
- an internal node splits, and
- a new root is generated.

When splitting, any value being promoted upwards will come from the node that is splitting.

- When a leaf node splits, a ‘copy’ of a key value is promoted.
- When an internal node splits, the middle key value ‘moves’ from a child to its parent node.

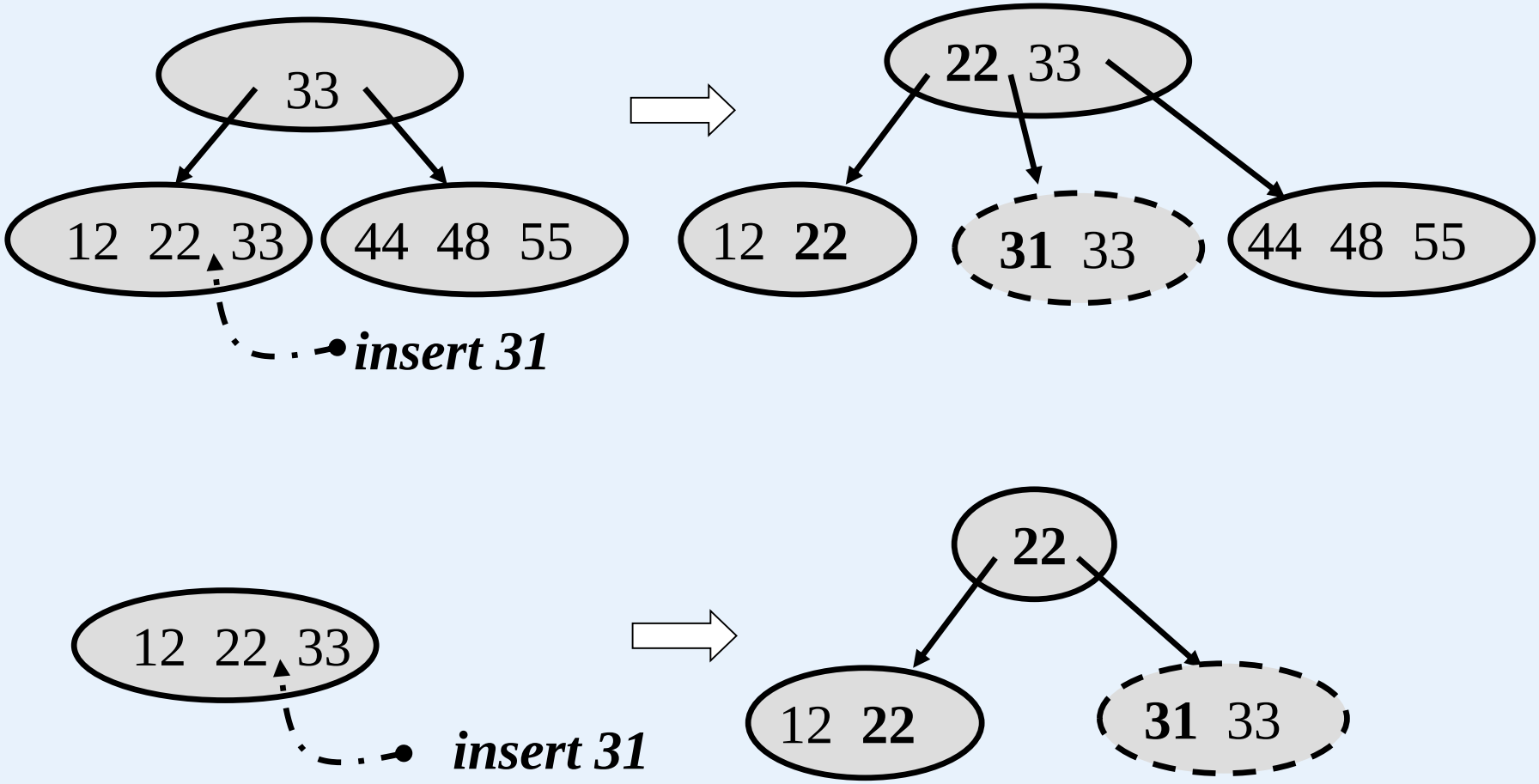
•**Leaf Node Splitting**

When a leaf node splits, a new leaf is allocated:

- the original leaf is the left sibling, the new one is the right sibling,
- key and pointer pairs are redistributed: the left sibling will have smaller keys than the right sibling,
- a 'copy' of the key value which is the largest of the keys in the left sibling is promoted to the parent.

Two situations arise: the parent exists or not.

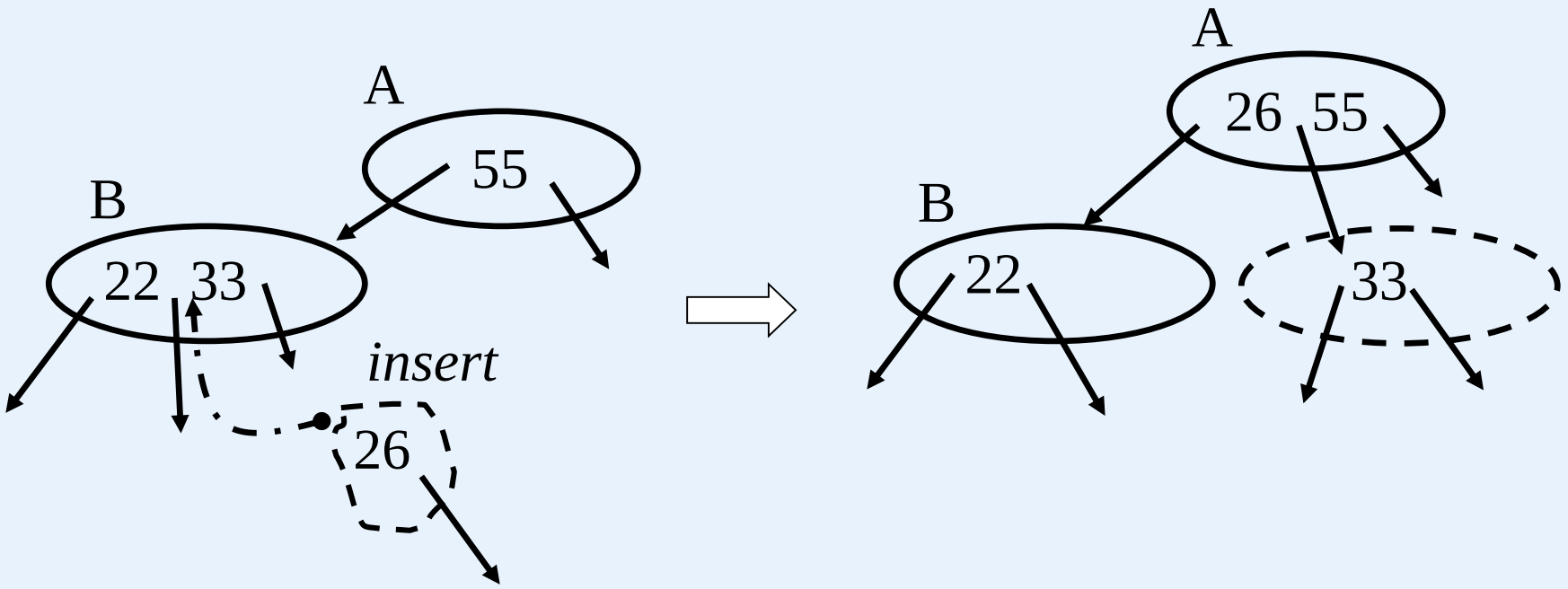
- If the parent exists, then a copy of the key value (just mentioned) and the pointer to the right sibling are promoted upwards.
- Otherwise, the B+-tree is just beginning to grow.


33
12 22 33
44 48 55
insert 31
22 33
12 22
31 33
44 48 55
12 22 33
insert 31
22
12 22
31 33

Internal Node splitting

If an internal node splits and it is **not the root**,

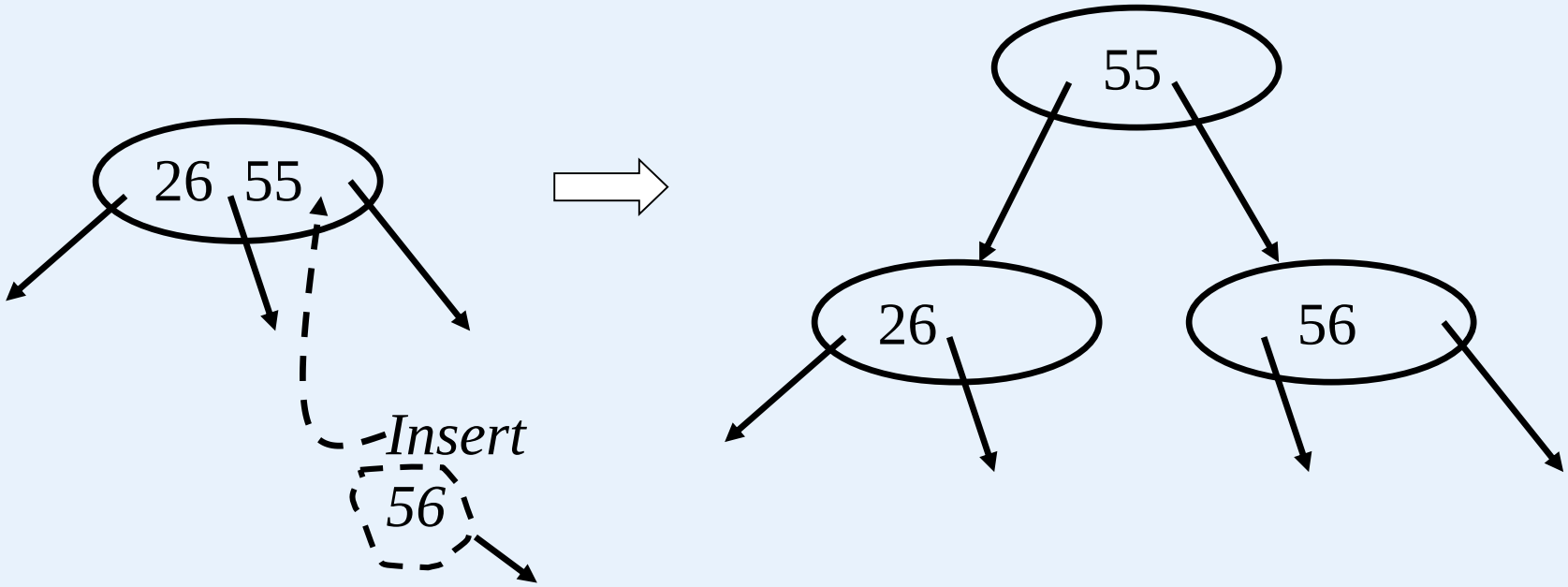
- insert the key and pointer and then determine the middle key,
- a new 'right' sibling is allocated,
- everything to its left stays in the left sibling,
- everything to its right goes into the right sibling,
- the middle key value along with the pointer to the new right sibling is promoted to the parent (the middle key value 'moves' to the parent to become the discriminator between the left and right sibling)



Note that '26' does not remain in B. This is different from the leaf node splitting.

Internal node splitting

When a **new root** is formed, a key value and two pointers must be placed into it.



B+-trees:

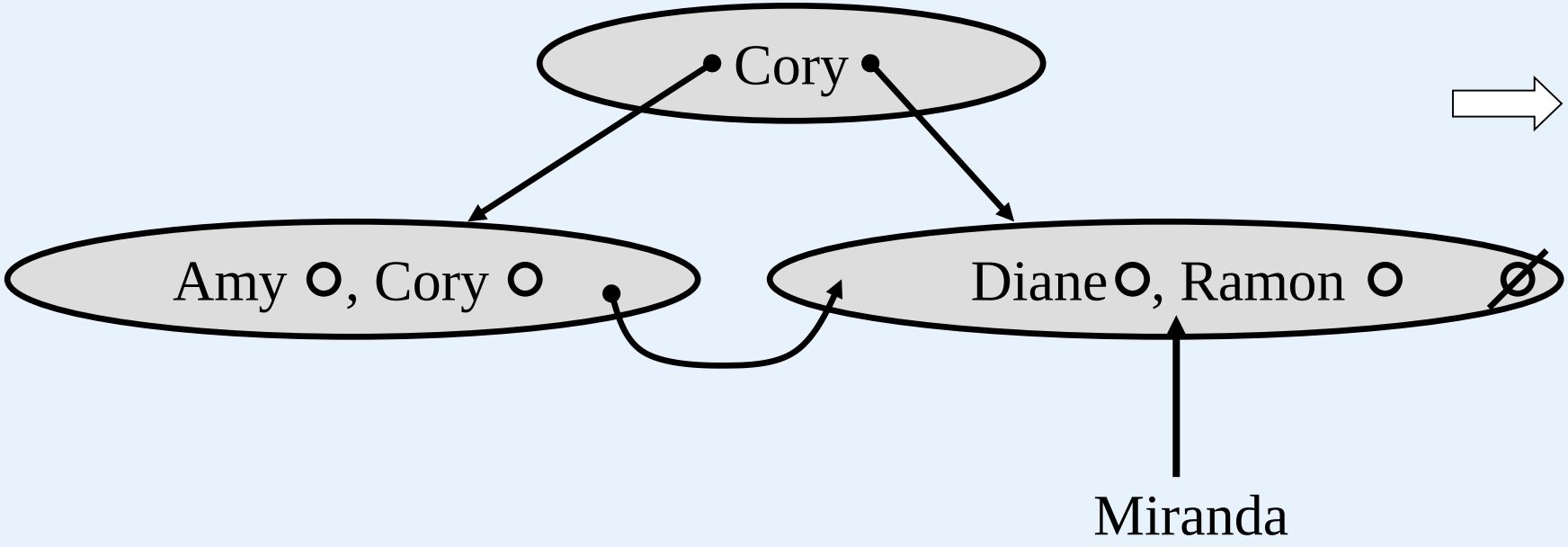
1. Data structure of an internal node is different from that of a leaf.
2. The meaning of $p_{internal}$ is different from p_{leaf}.
3. Splitting an internal node is different from splitting a leaf.
4. A new key value to be inserted into a leaf comes from the data file.
5. A key value to be inserted into an internal node comes from a node at a lower lever.

A sample trace

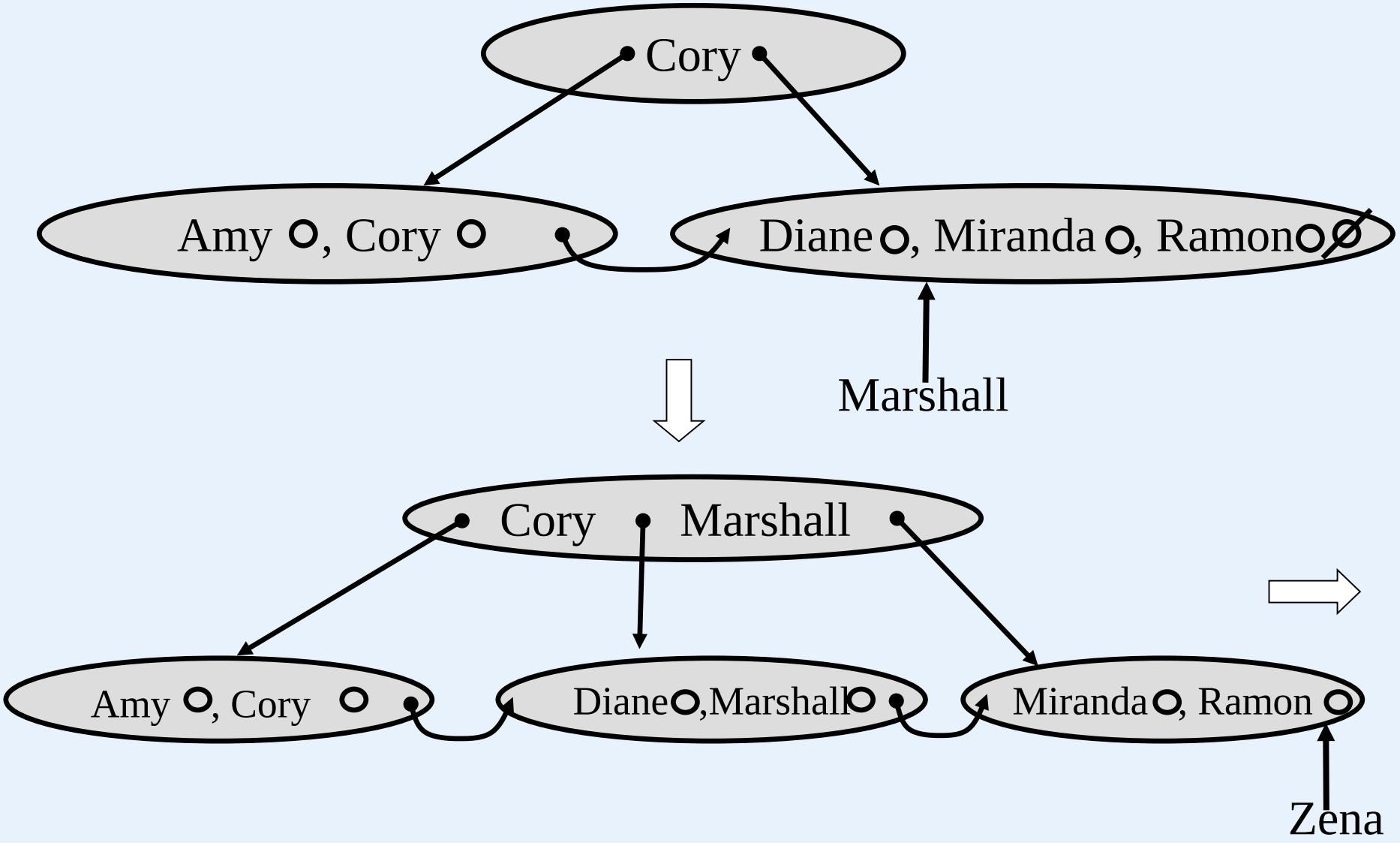
Diane, Cory, Ramon, Amy, Miranda,

Marshall, Zena, Rhonda, Vincent, Simon, mary

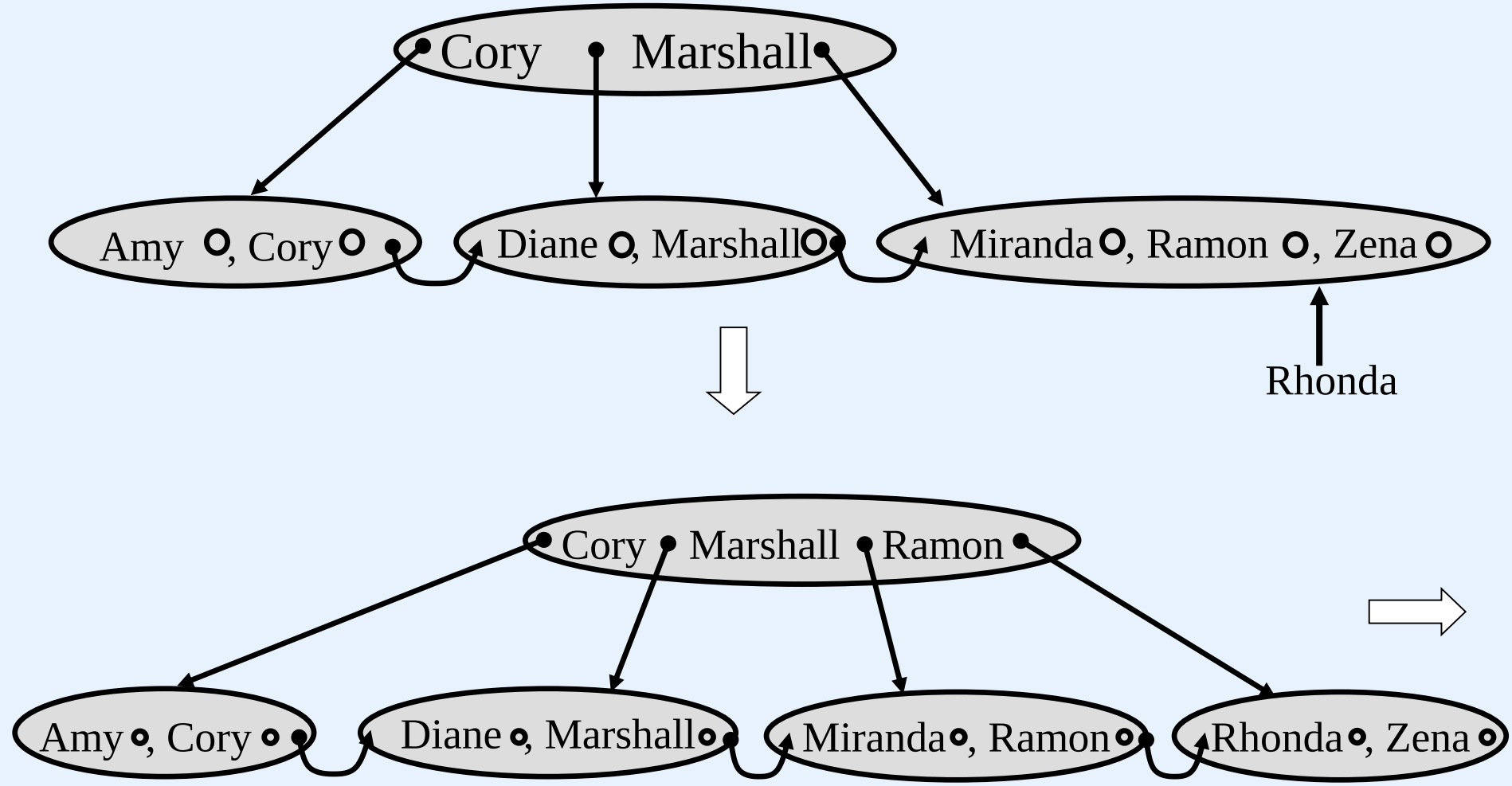
into a b^+-tree with $p_{internal} = p_{leaf} = 3$.



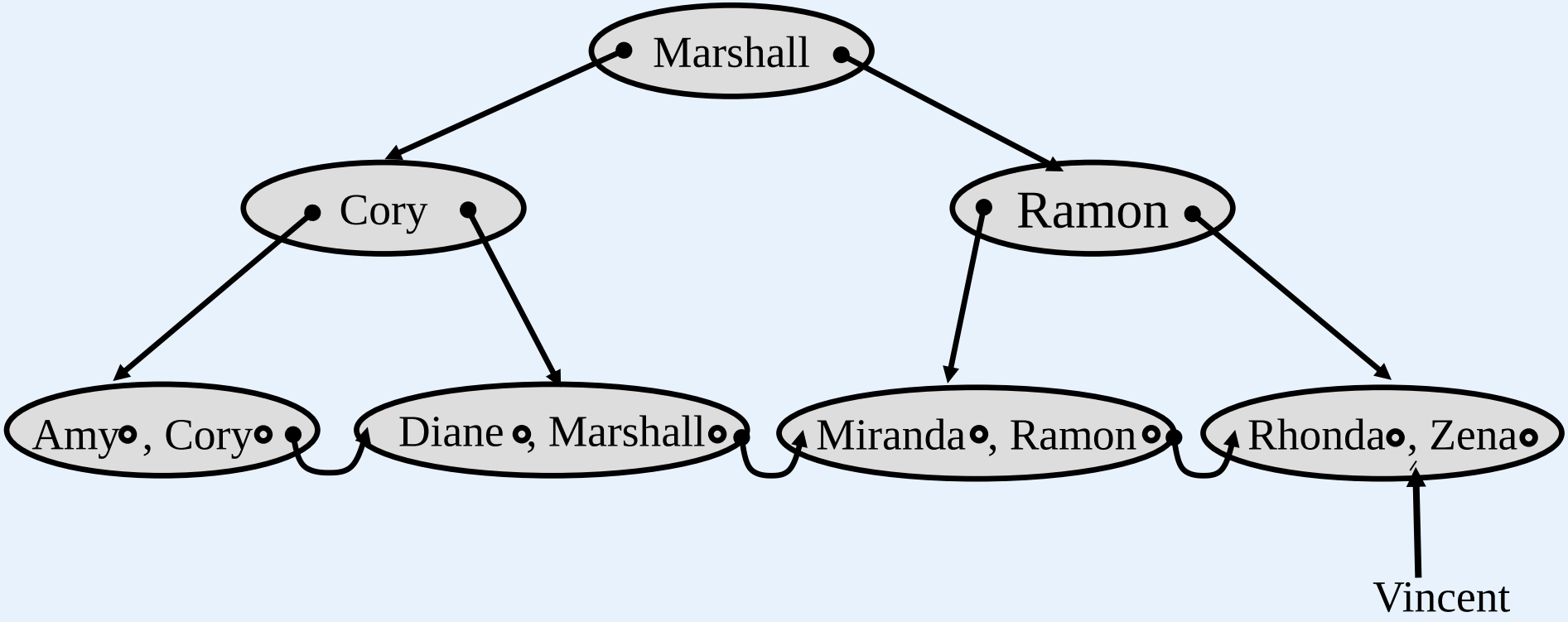
B+-Trees and Trees for Multidimensional Data



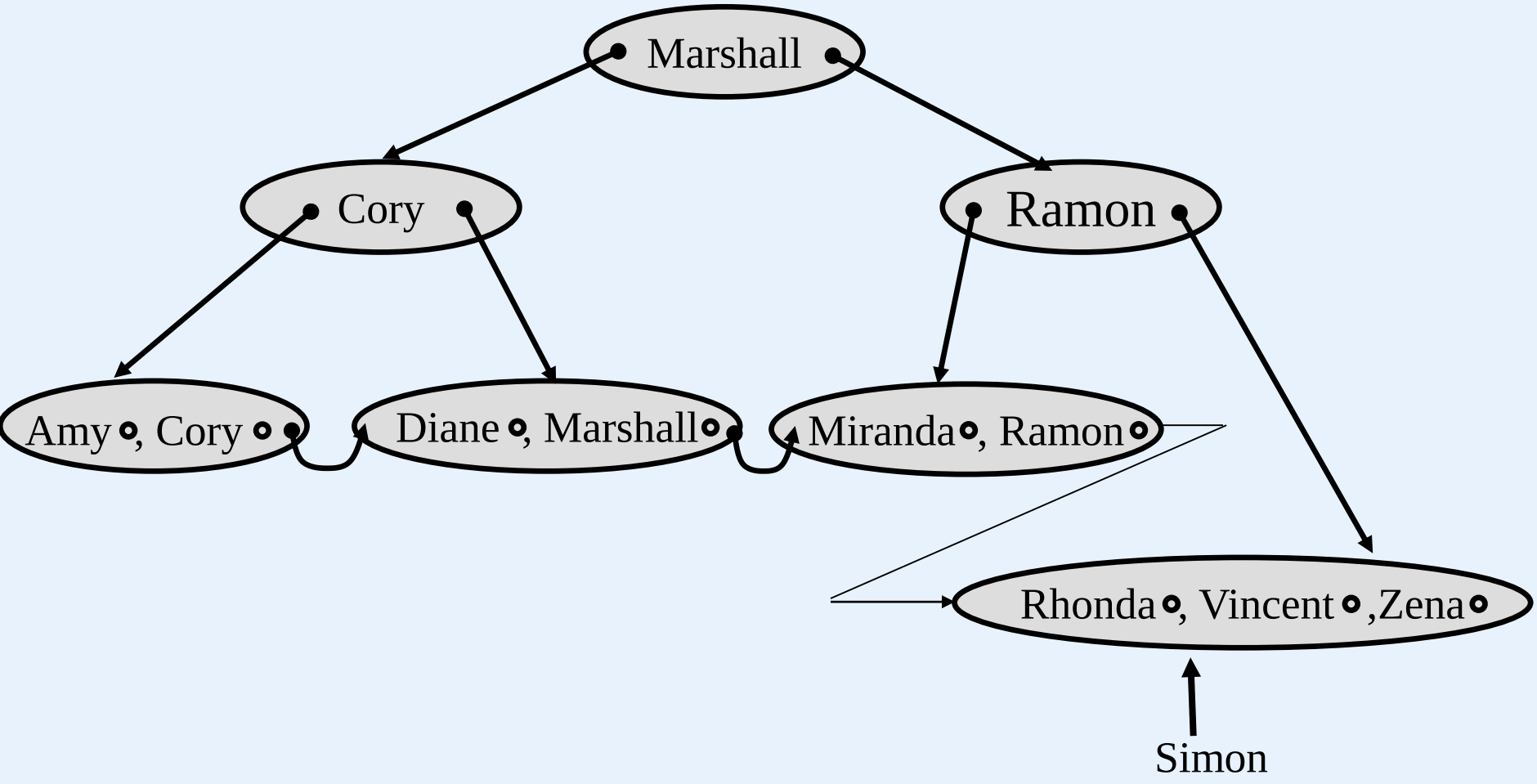
B+-Trees and Trees for Multidimensional Data



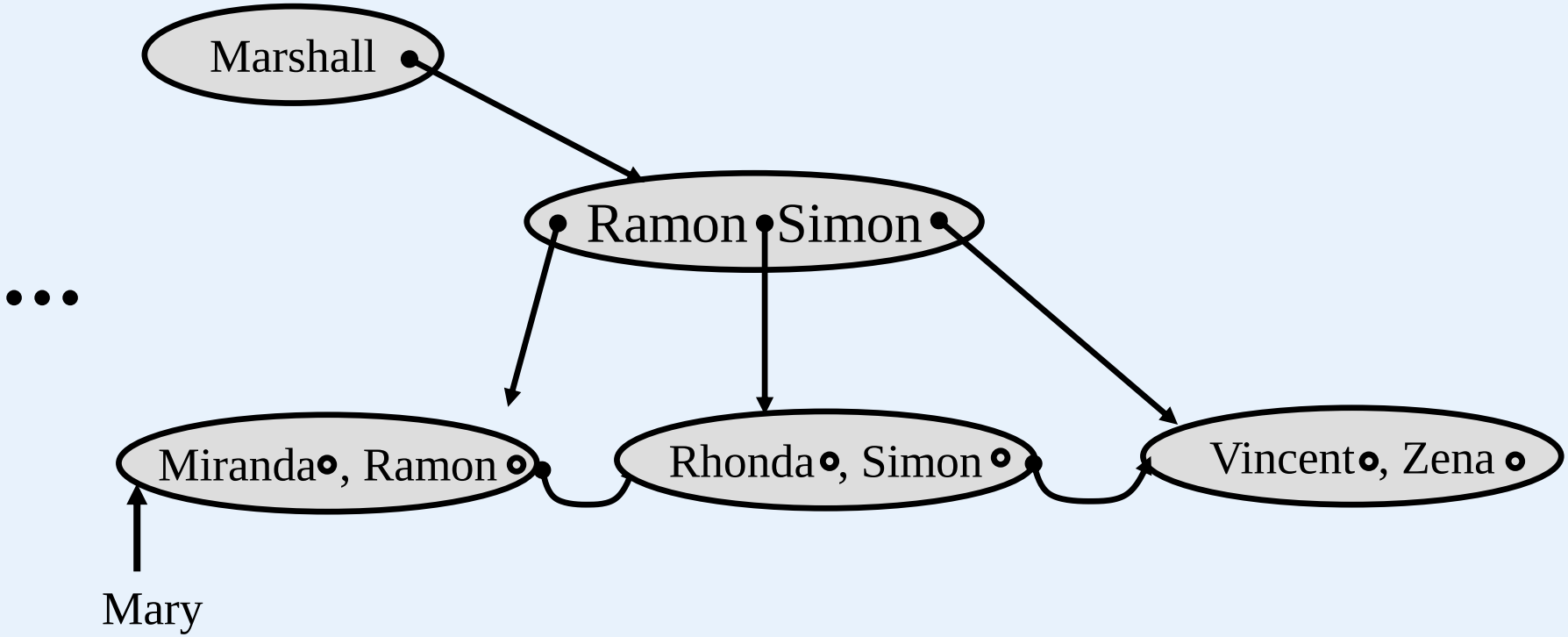
B+-Trees and Trees for Multidimensional Data



B+-Trees and Trees for Multidimensional Data



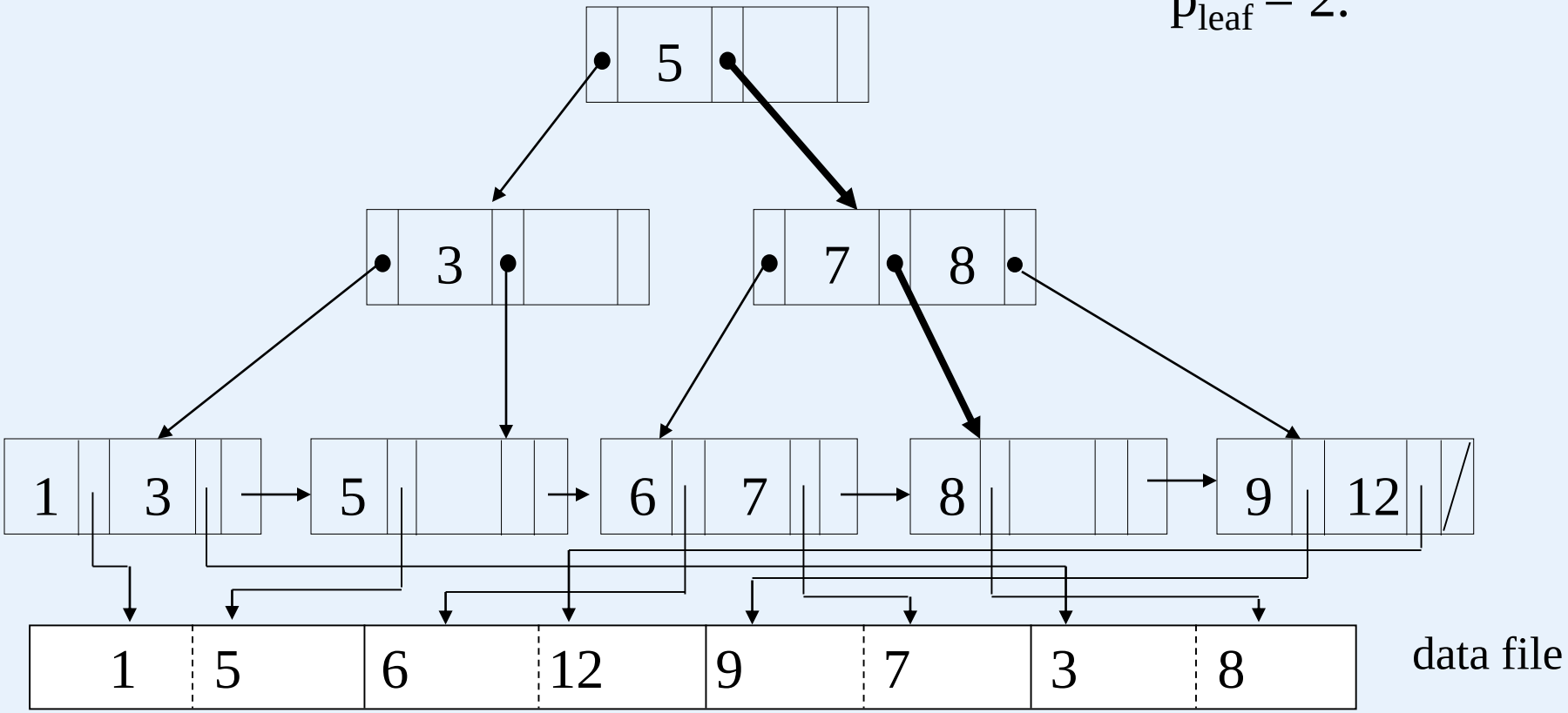
B+-Trees and Trees for Multidimensional Data



Searching a B+-tree

- searching a record with key = 8:

$p_{internal} = 3$,
$p_{leaf} = 2$.



B+-tree Maintenance

- Inserting a key into a B+-tree

  (Same as discussed on B+-tree construction)

- Deleting a key from a B+-tree

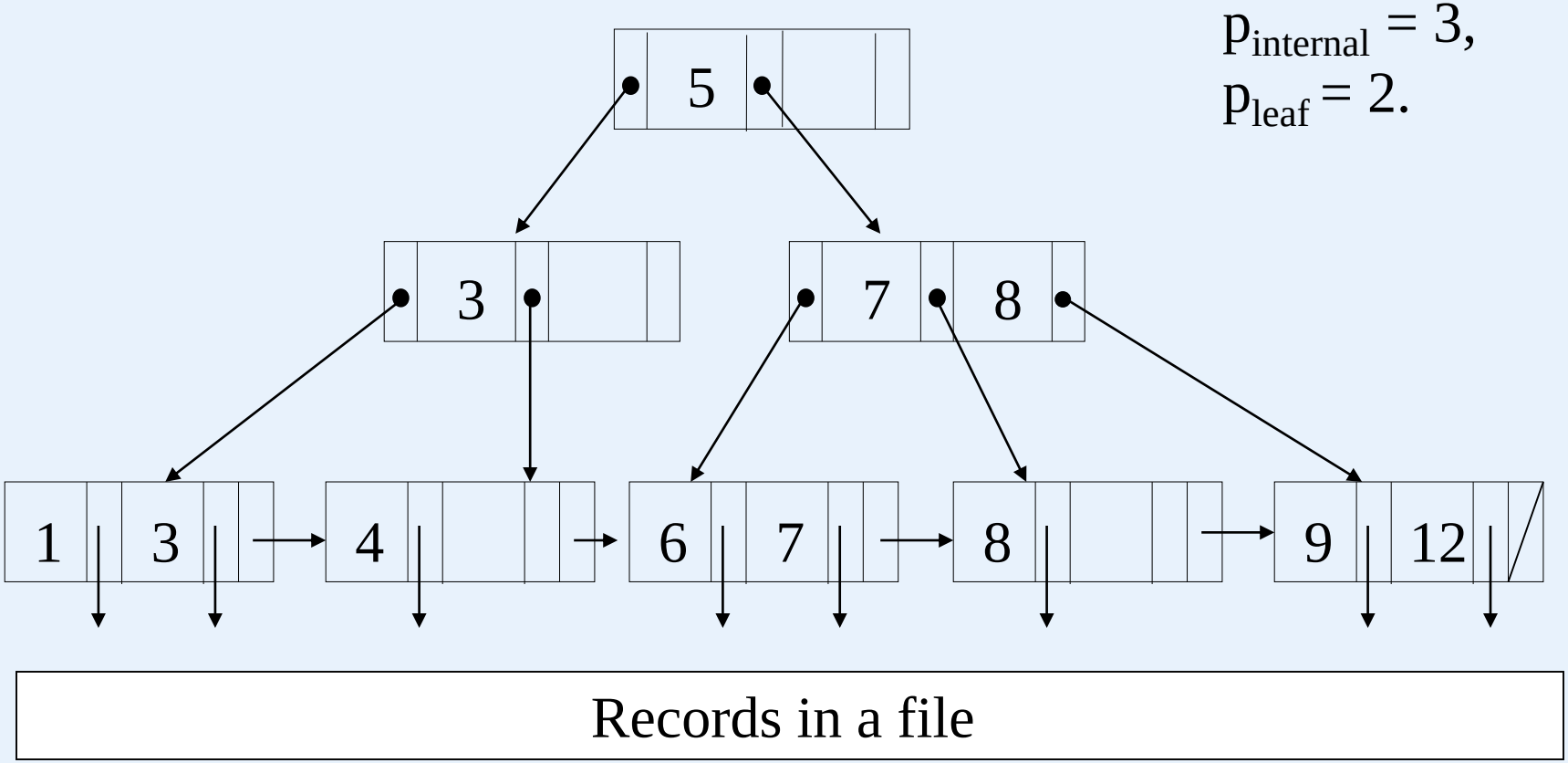
  i) Find the leaf node containing the key to be removed and delete it from the leaf node.

  ii) If *underflow*, redistribute the leaf node and one of its siblings (left or right) so that both are at least half full.

  iii) Otherwise, the node is merged with its siblings and the number of leaf nodes is reduced.

Entry deletion

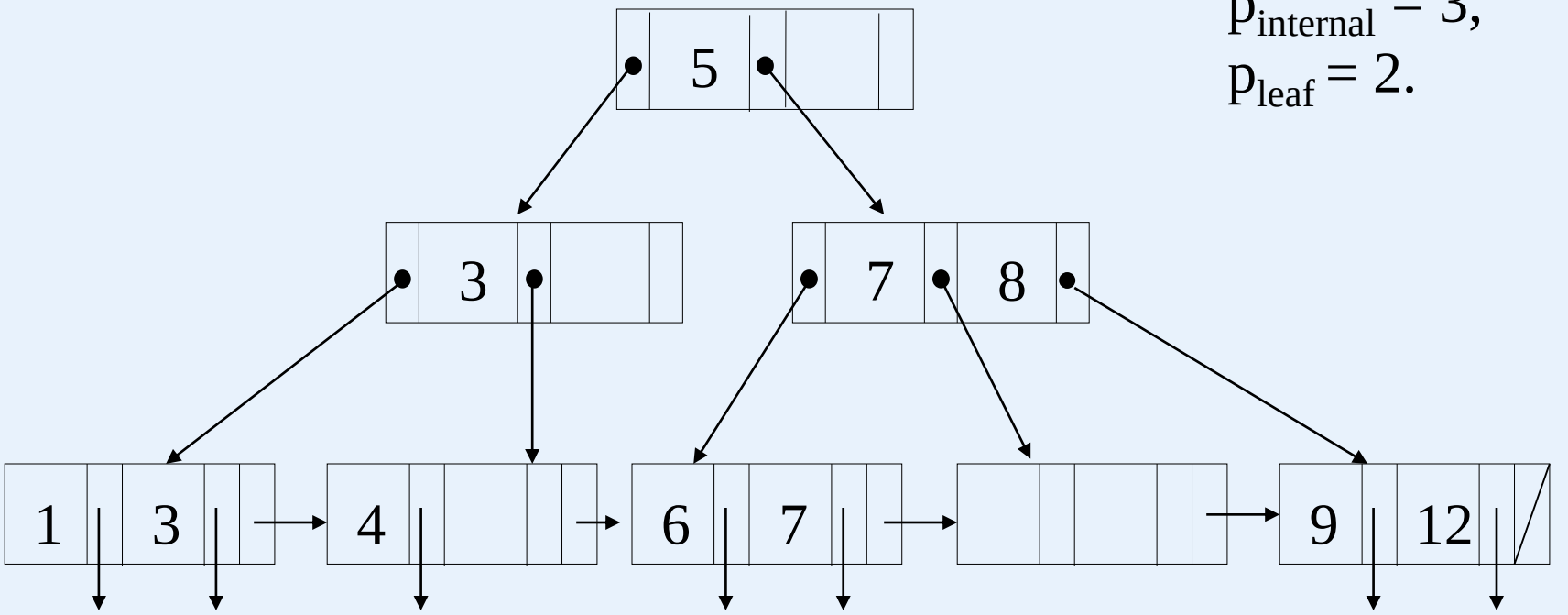
- deletion sequence: 8, 12, 9, 7



Entry deletion

- deletion sequence: 8, 12, 9, 7

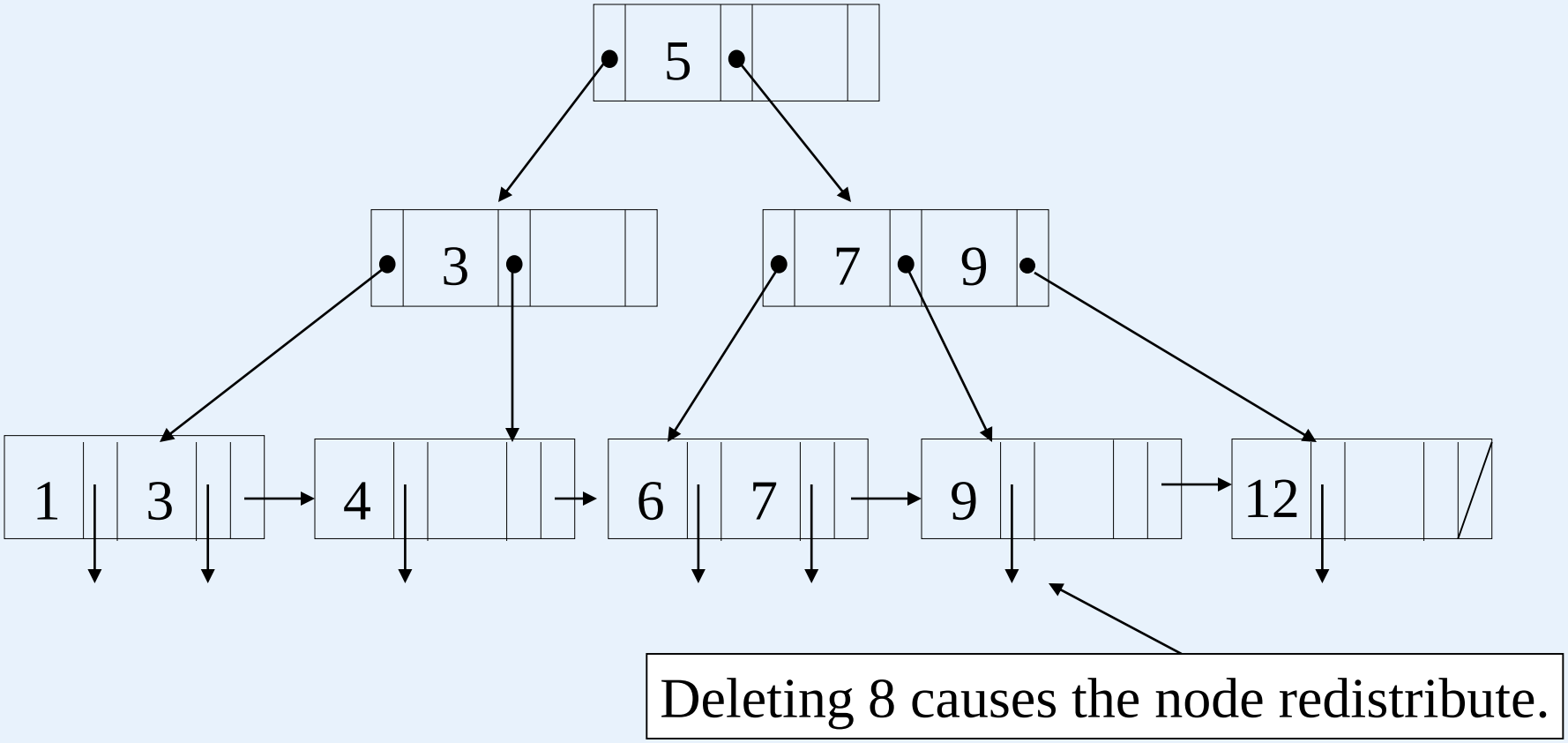
$p_{internal} = 3$,
$p_{leaf} = 2$.



Records in a file

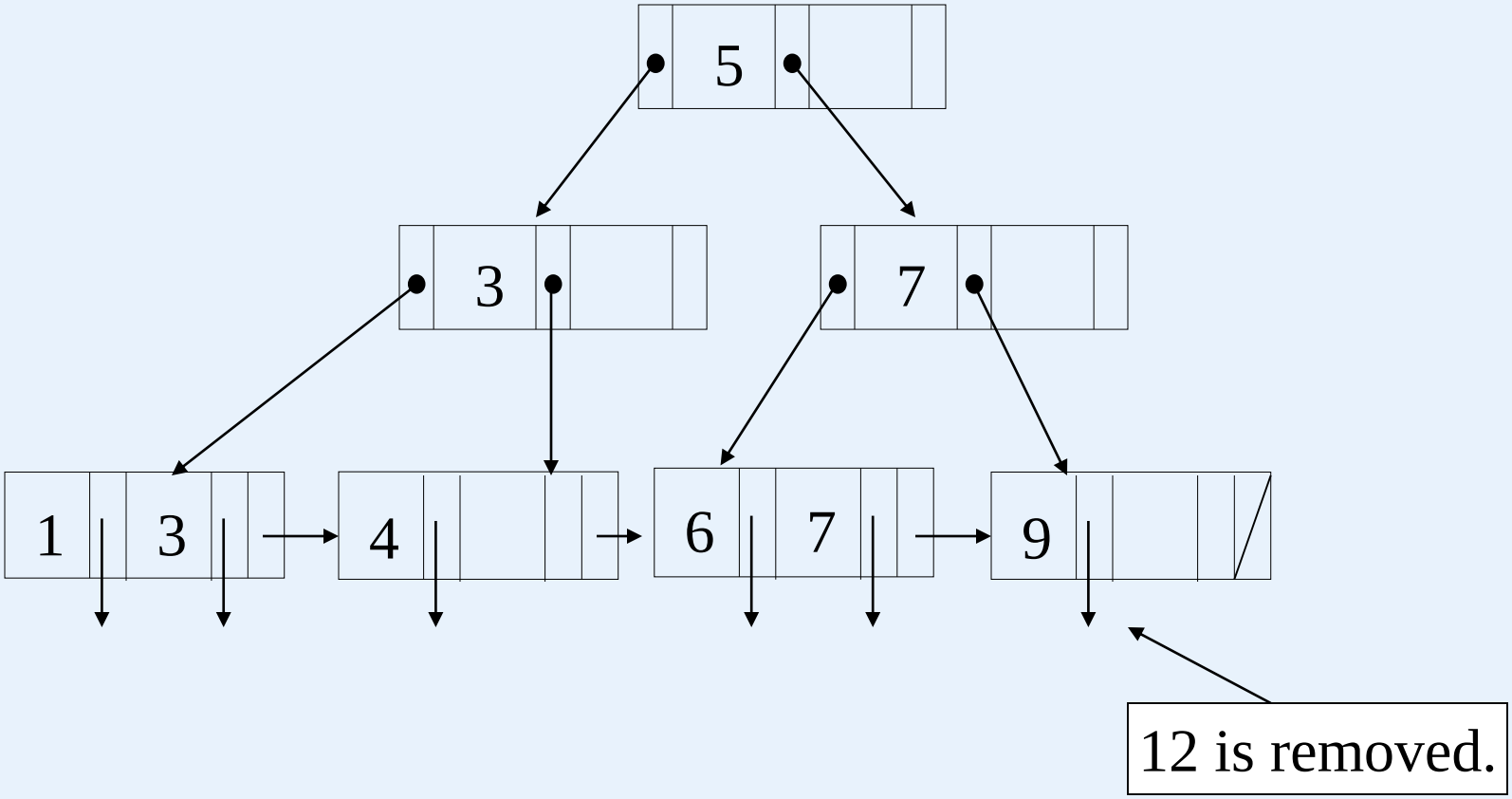
Entry deletion

- deletion sequence: 8, 12, 9, 7



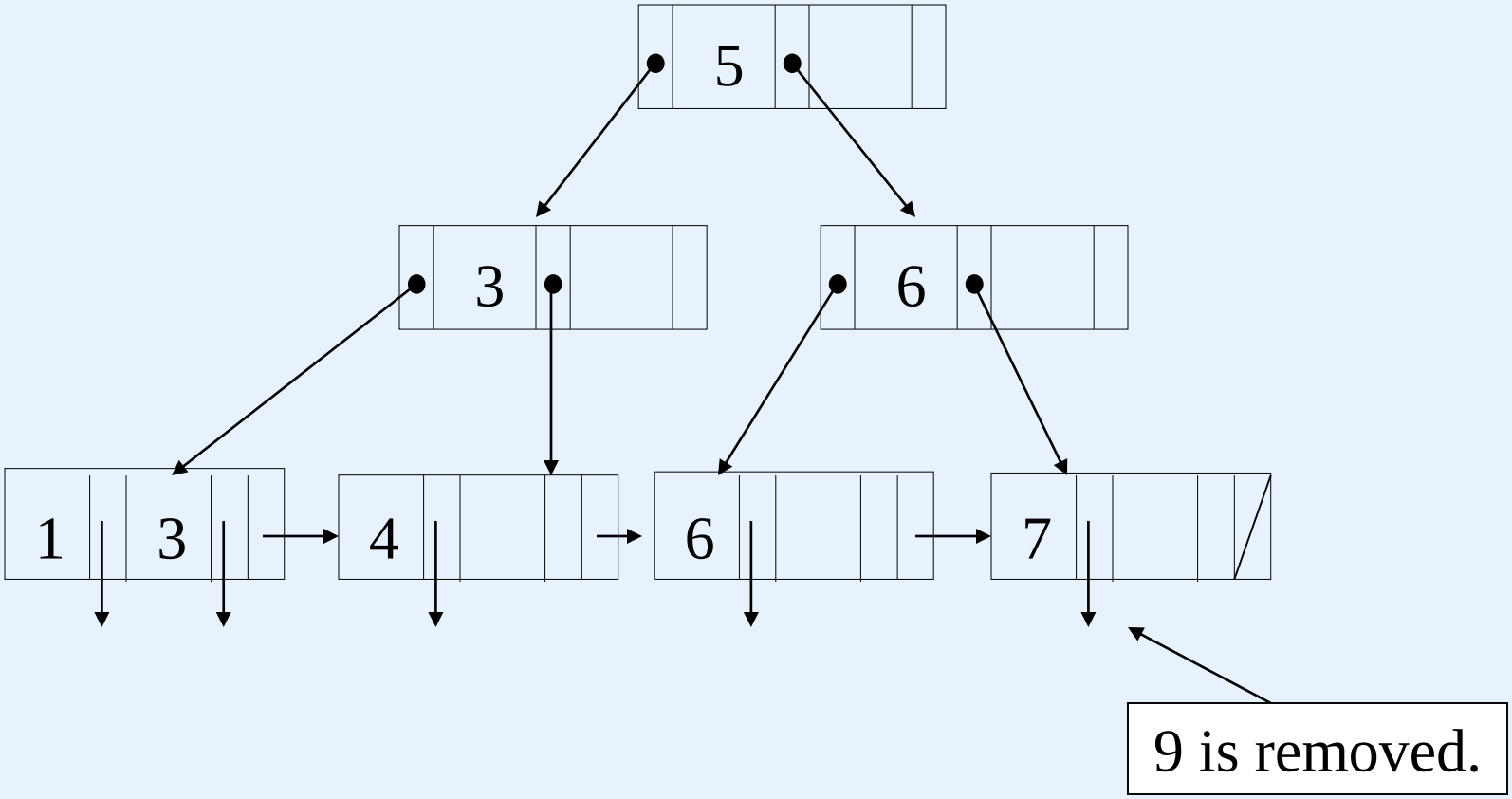
Entry deletion

- deletion sequence: 8, 12, 9, 7



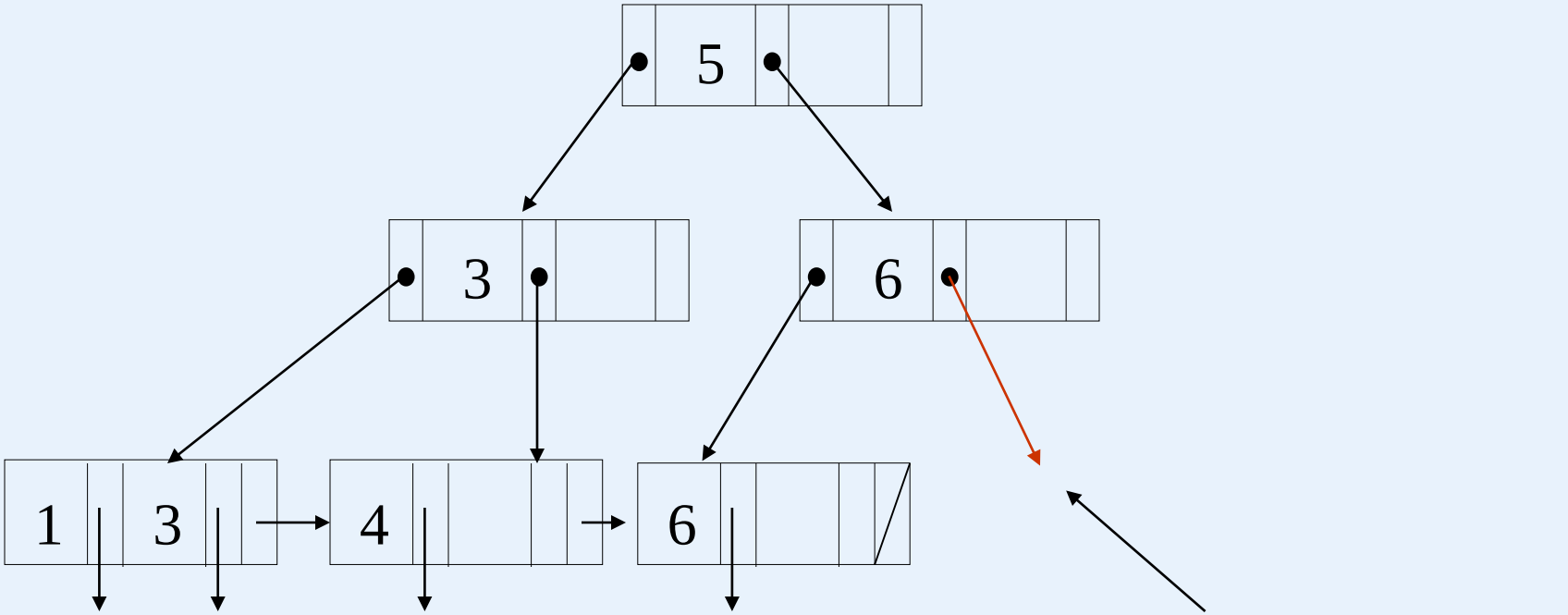
Entry deletion

- deletion sequence: 8, 12, 9, 7



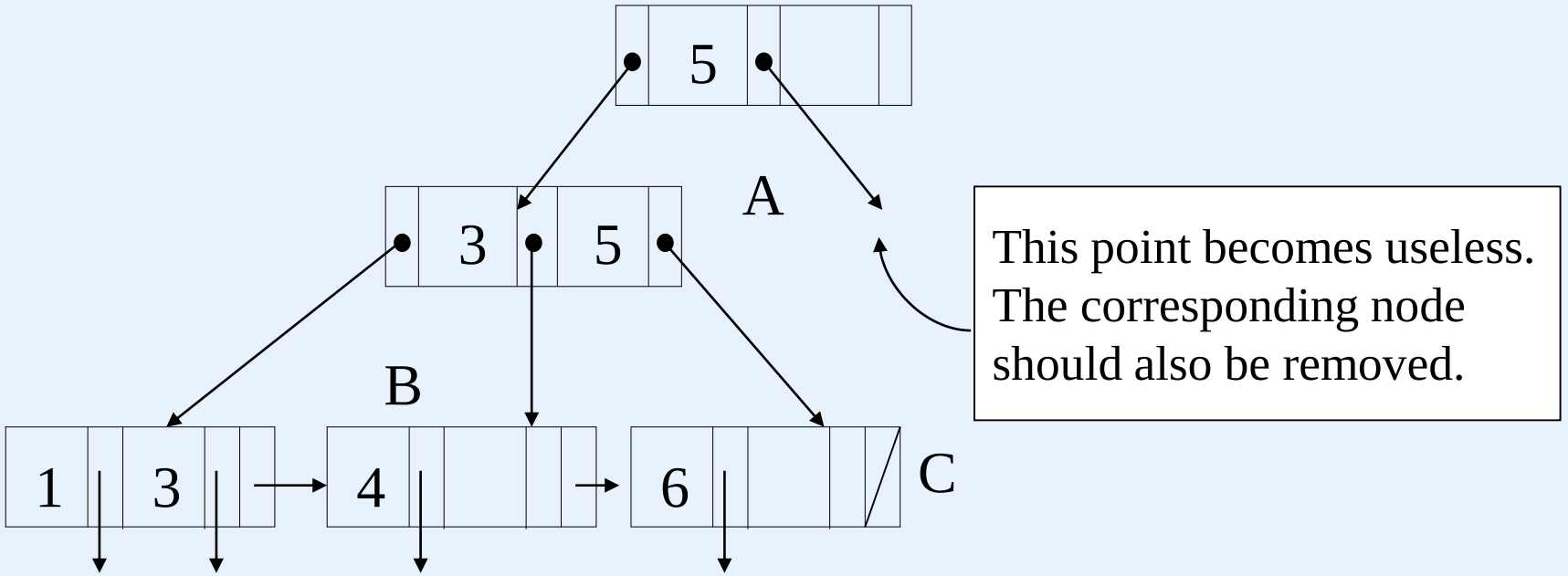
Entry deletion

- deletion sequence: 8, 12, 9, 7



Entry deletion

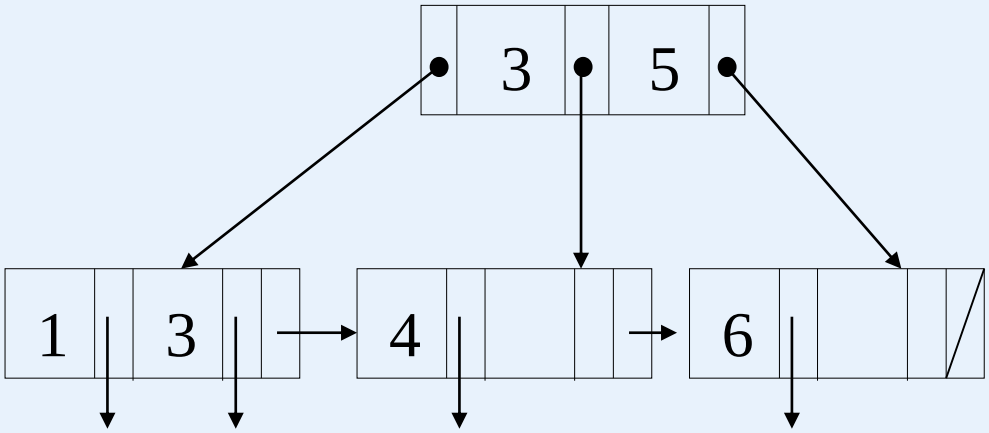
- deletion sequence: 8, 12, 9, 7



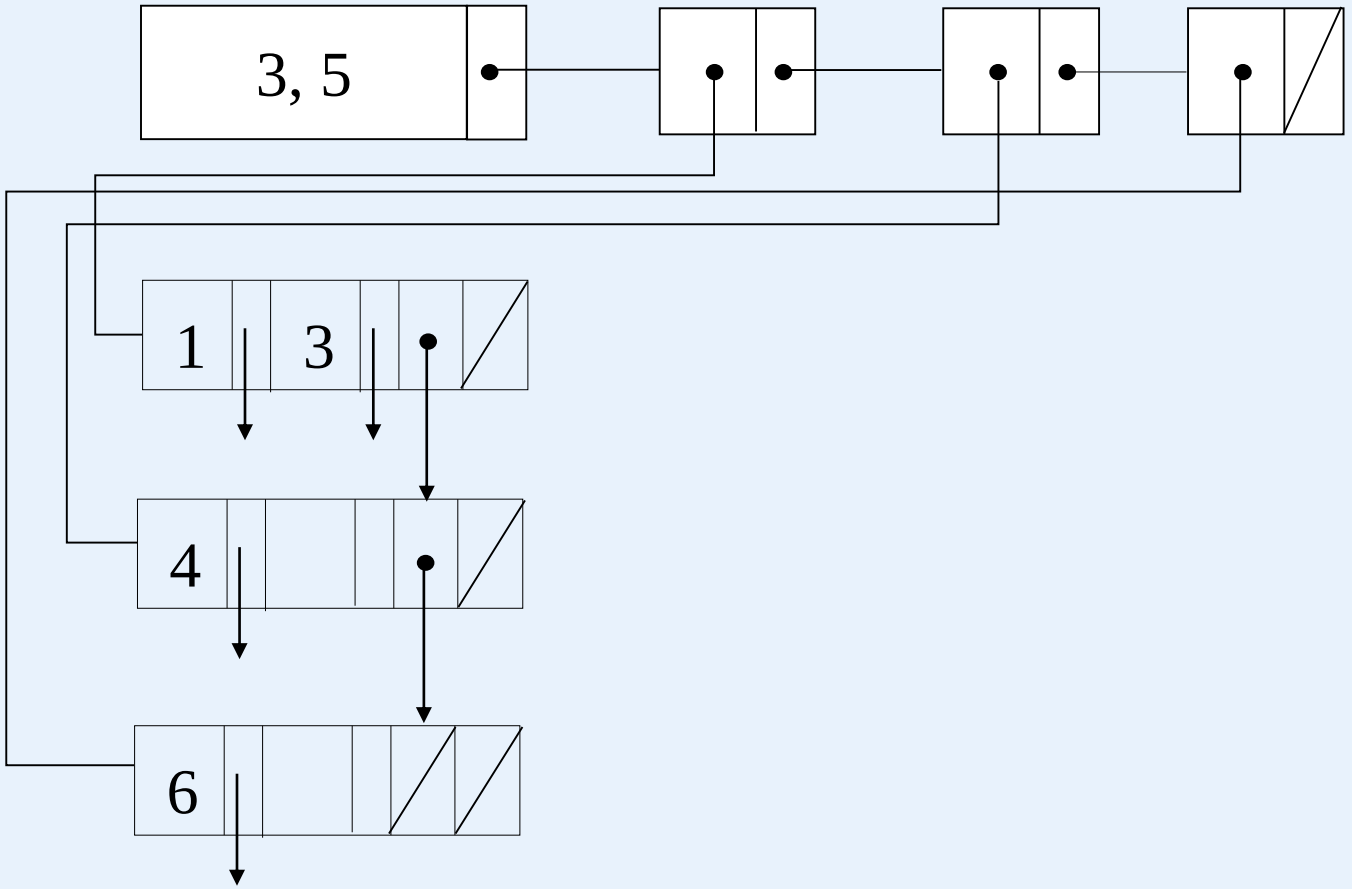
For this merge, 5 will be taken as a key value in A since any key value in B is less than or equal to 5 but any key value in C is larger than 5.

Entry deletion

- deletion sequence: 8, 12, 9, 7



A B+-tree stored in main memory as a link list:



Creating link lists in C:

1. Create data types using “struct”:

```
struct  node                    struct  edge
    {                               {
     name string[200];               link_to_node node;
     next node;                      link_to_next edge;
     link edge;                     }
    }
```

2. Allocate place for nodes:

- Using “allocating commands” to get memory place for nodes

```
x = (struct node *) calloc(1, sizeof(struct node));
```

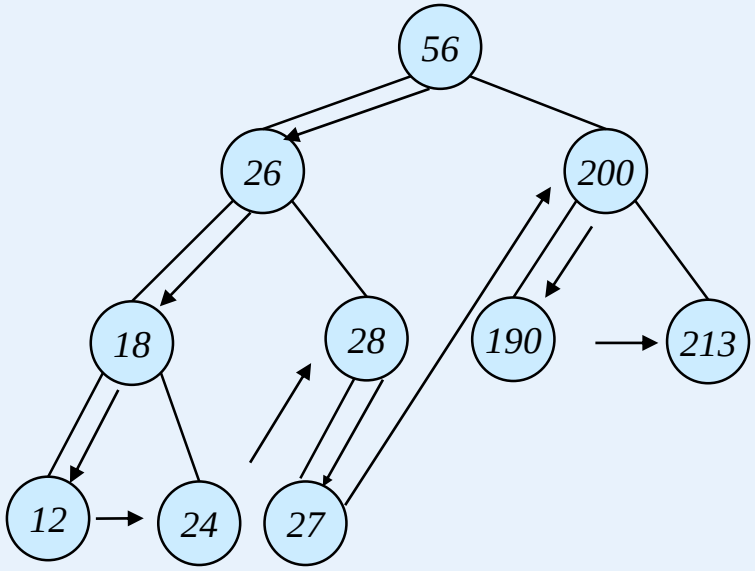
- Using fields to establish values for the nodes

```
x.name = “company”;
y = (struct edge *) calloc(1, sizeof(struct edge));
x.link = y;
```

Store a B+-tree on hard disk

Depth-first-search:

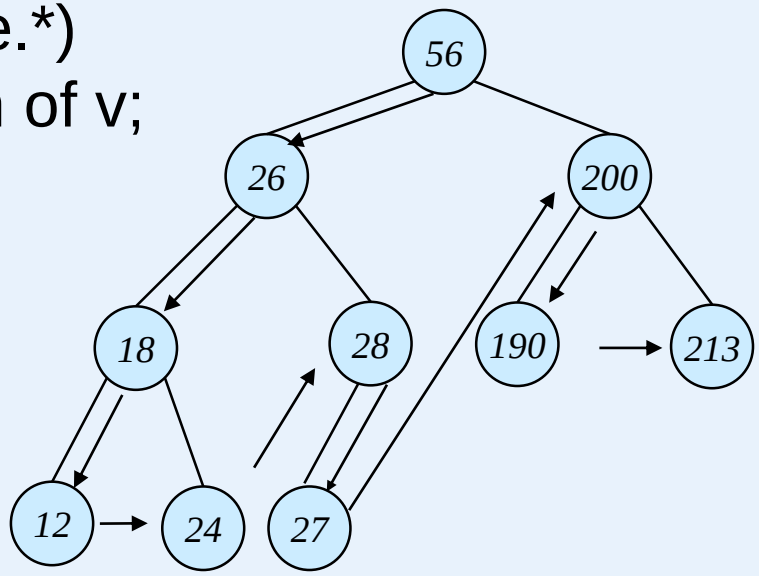
```
DFS(v)          (*recursive strategy*)
Begin
    print(v); (*or store v in a file.*)
    let v1, …, vk be the children of v;
    for (i = 1 to k ) {DFS(vi);}
end
```



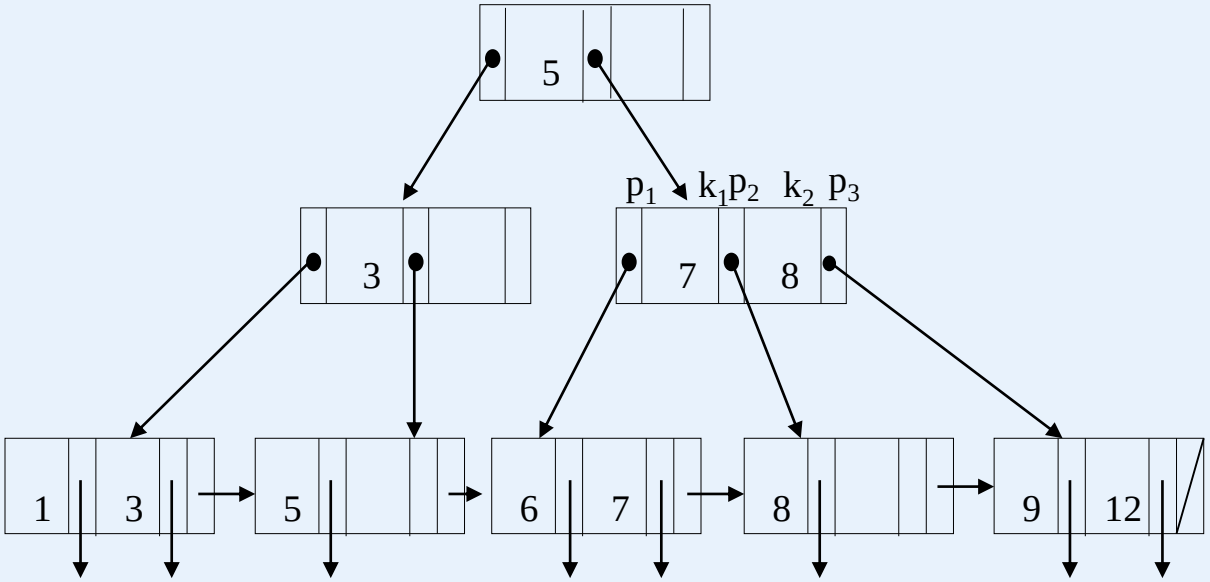
Store a B+-tree on hard disk

Depth-first-search:

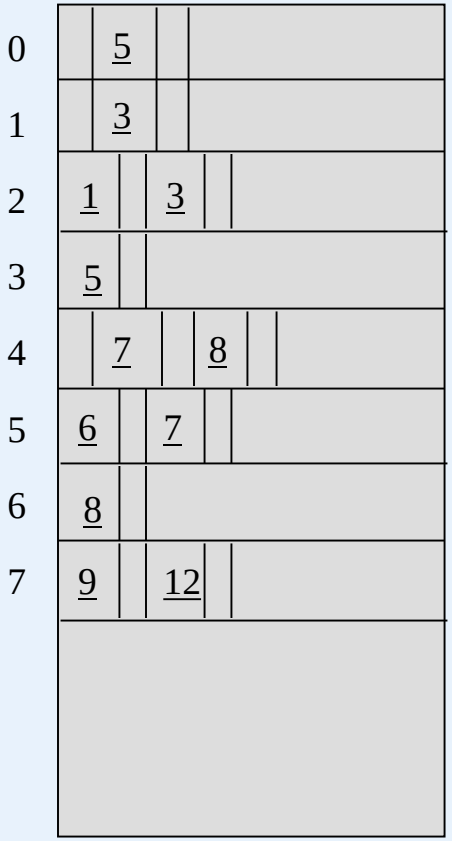
```
(*non-recursive strategy*)
push(root);    (*push the root into stack.*)
while (stack is not empty) do
{    v := pop( );
     print(v); (*or store v in a file.*)
     let v1, …, vk be the children of v;
     for (i = k to 1) {push(vi)};
}
```



B+-Trees and Trees for Multidimensional Data



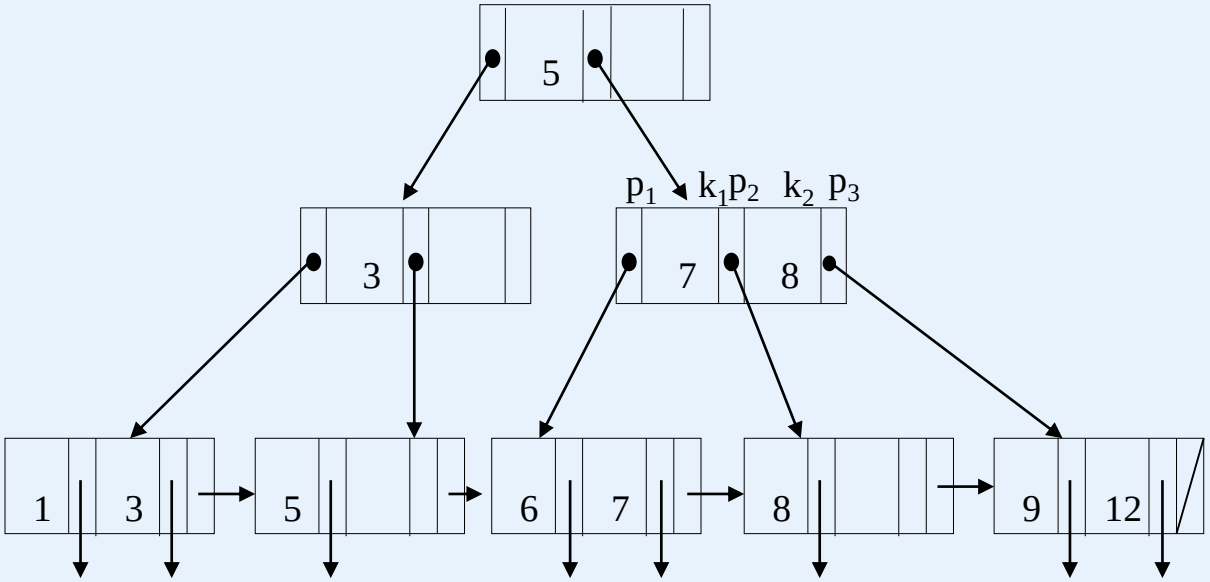
B+-tree stored in a file:



Data file:

0 1 2 3

B+-Trees and Trees for Multidimensional Data



B+-tree stored in a file:

0	1	**5**	4					
1	2	**3**	3					
2	**1**	0	**3**	3				
3	**5**	0						
4	5	**7**	6	**8**	7			
5	**6**	1	**7**	2				
6	**8**	3						
7	**9**	2	**12**	1				

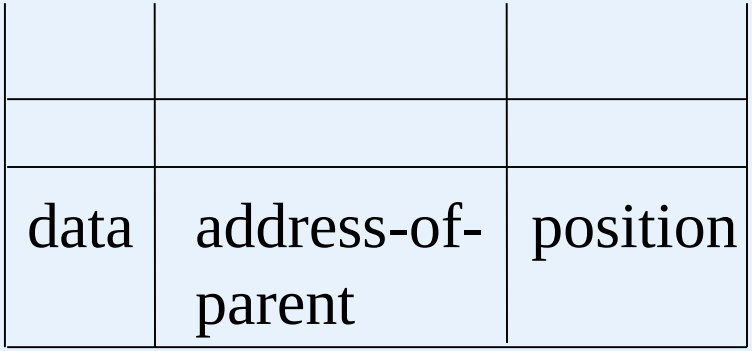
Data file:

0	1	2	3
1 5	6 12	9 7	3 8

Store a B+-tree on hard disk

Algorithm:

```
push(root, -1, -1);
while (S is not empty) do
{     x := pop( );
      store x.data in file F;
      assume that the address of x in F is ad;
      if x.address-of-parent ≠ -1 then {
            y := x.address-of-parent;
            z := x.position;
            write ad in page y at position z in F;
      }
      let x1, …, xk be the children of x;
      for (i = k to 1) {push(xi, ad, i)};
}
```

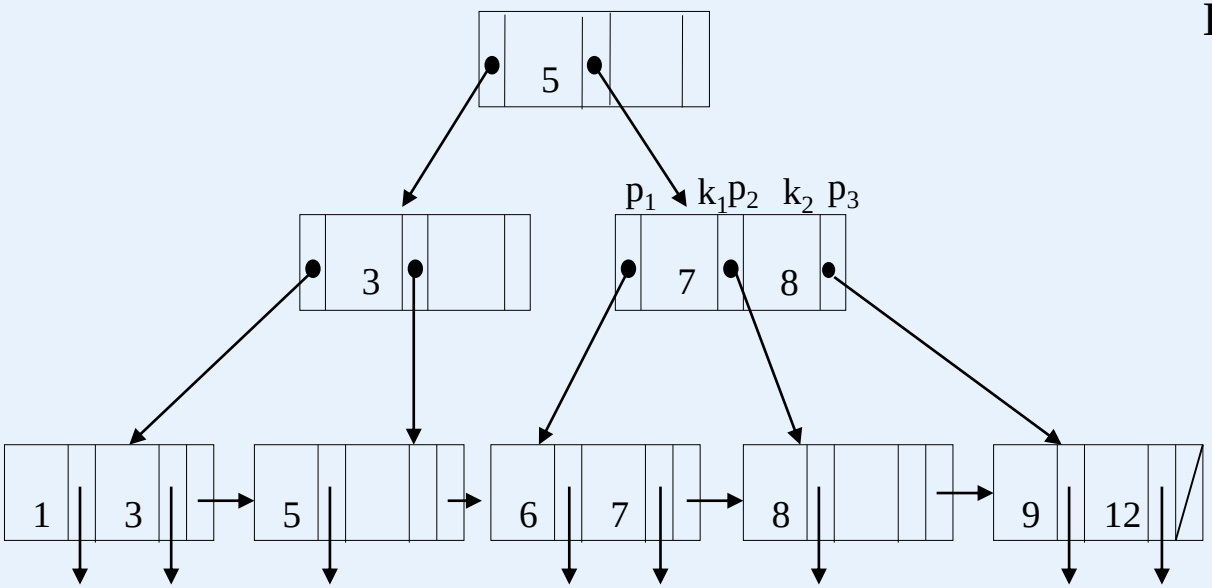


stack: *S*

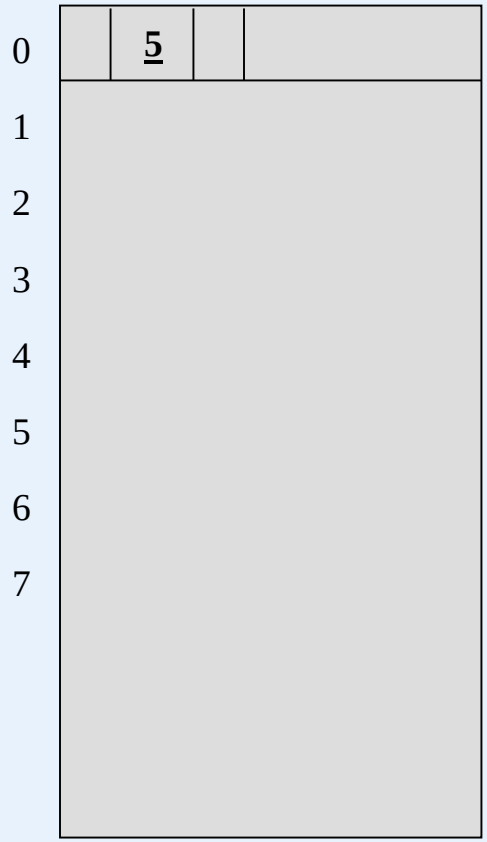
data: all the key values in a node

address-of-parent: a page number in the file *F*, where the parent of the node is stored.

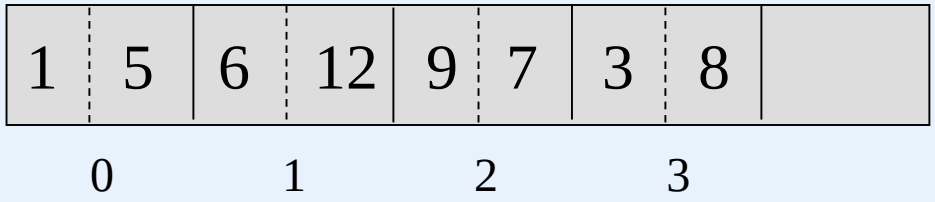
position: a number indicating what is the ranking of a child. That is, whether it is the first, second, …, child of its parent.



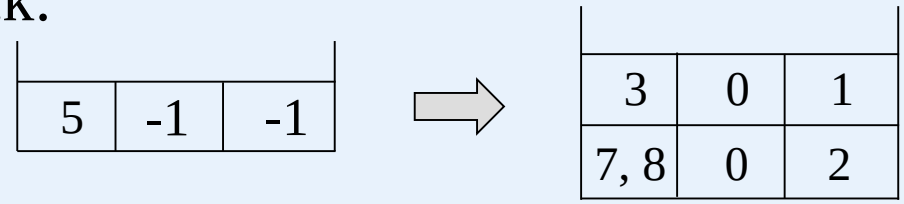
B+-tree stored in a file:



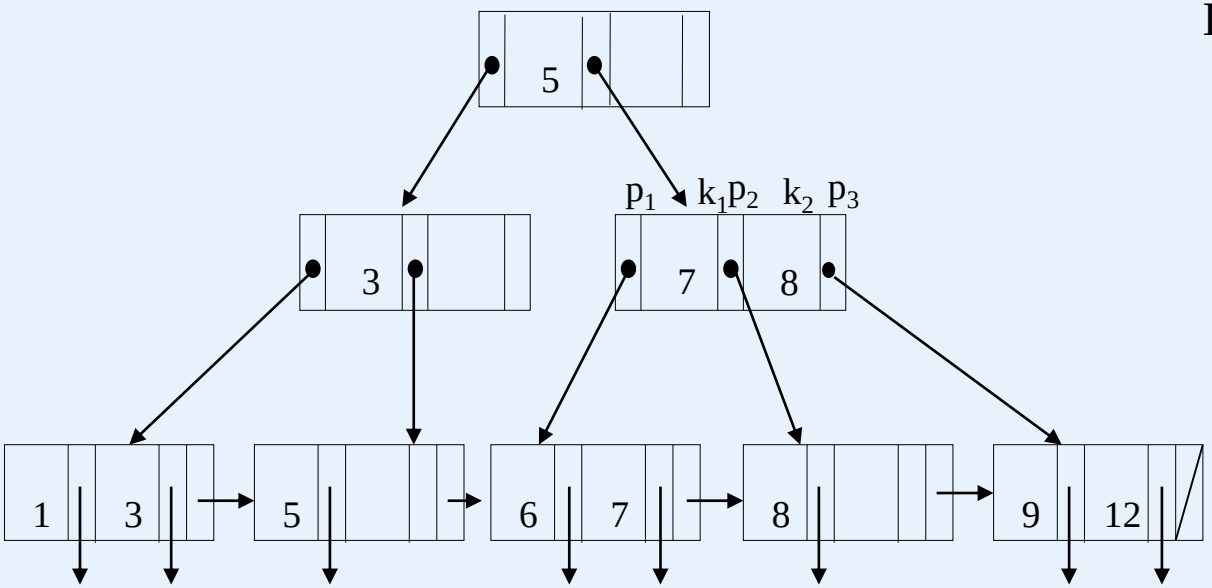
Data file:



Stack:



B+-Trees and Trees for Multidimensional Data



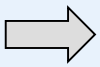
B+-tree stored in a file:

0	1	**5**		
1		**3**		
2				
3				
4				
5				
6				
7				

Data file:

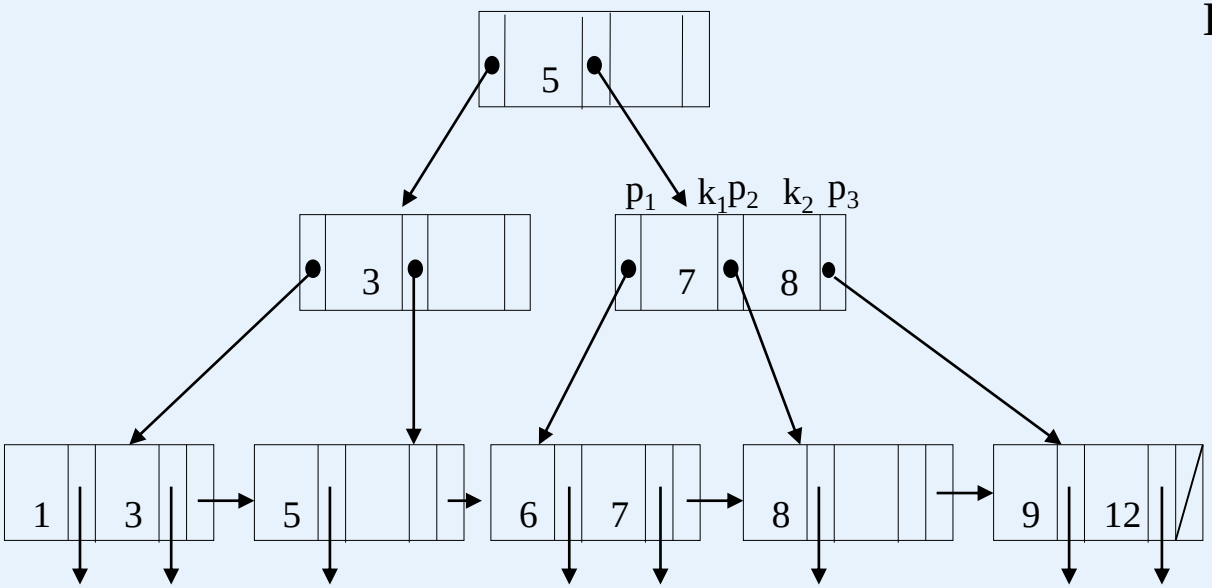
1	5	6	12	9	7	3	8	

	0	1
3	0	1
7, 8	0	2



	2	3
1, 3	1	1
5	1	2
7, 8	0	2

B+-Trees and Trees for Multidimensional Data



B+-tree stored in a file:

0	1	**5**		
1	2	**3**		
2	**1**	0	**3**	3
3				
4				
5				
6				
7				

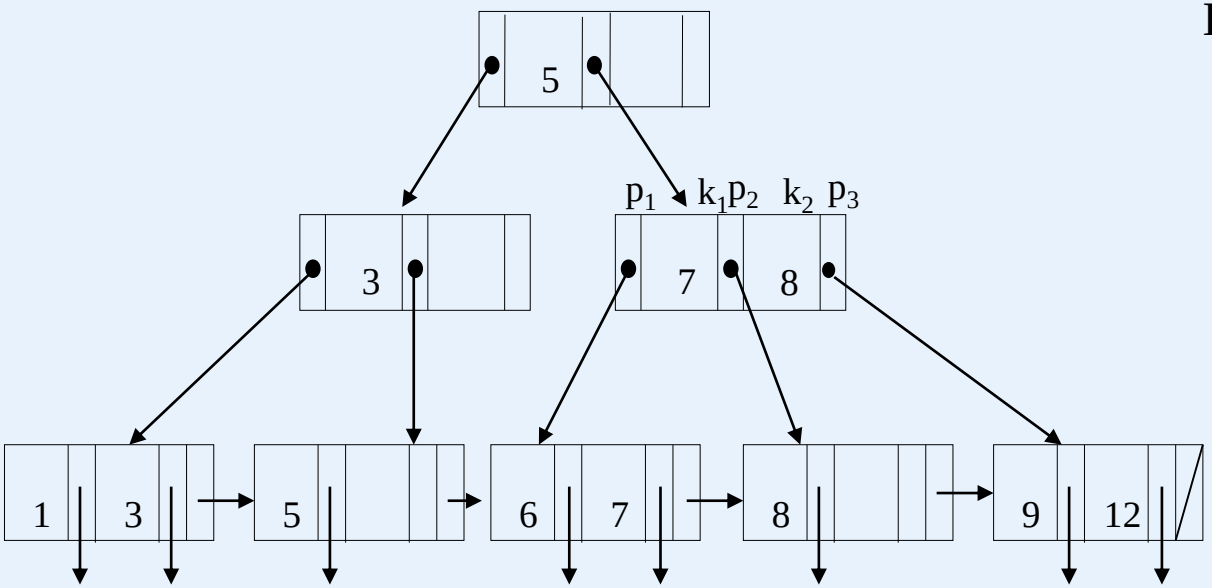
Data file:

1	5	6	12	9	7	3	8	

	0	1
1, 3	1	1
5	1	2
7, 8	0	2

	2	3
5	1	2
7, 8	0	2

B+-Trees and Trees for Multidimensional Data



B+-tree stored in a file:

0	1	**5**		
1	2	**3**	3	
2	**1**	0	**3**	3
3	**5**	0		
4				
5				
6				
7				

Data file:

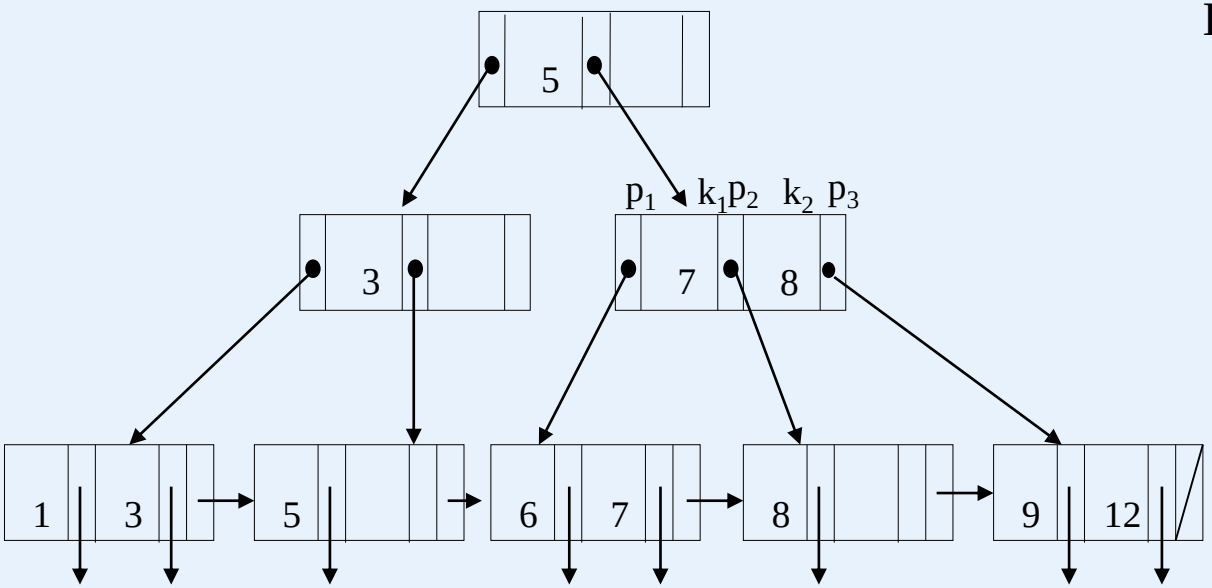
0	1	2	3
1	5	6	12
9	7	3	8

5	1	2
7, 8	0	2

⇒

7, 8	0	2

B+-Trees and Trees for Multidimensional Data



B+-tree stored in a file:

0	1	**5**	4	
1	2	**3**	3	
2	**1**	0	**3**	3
3	**5**	0		
4		**7**		**8**
5				
6				
7				

Data file:

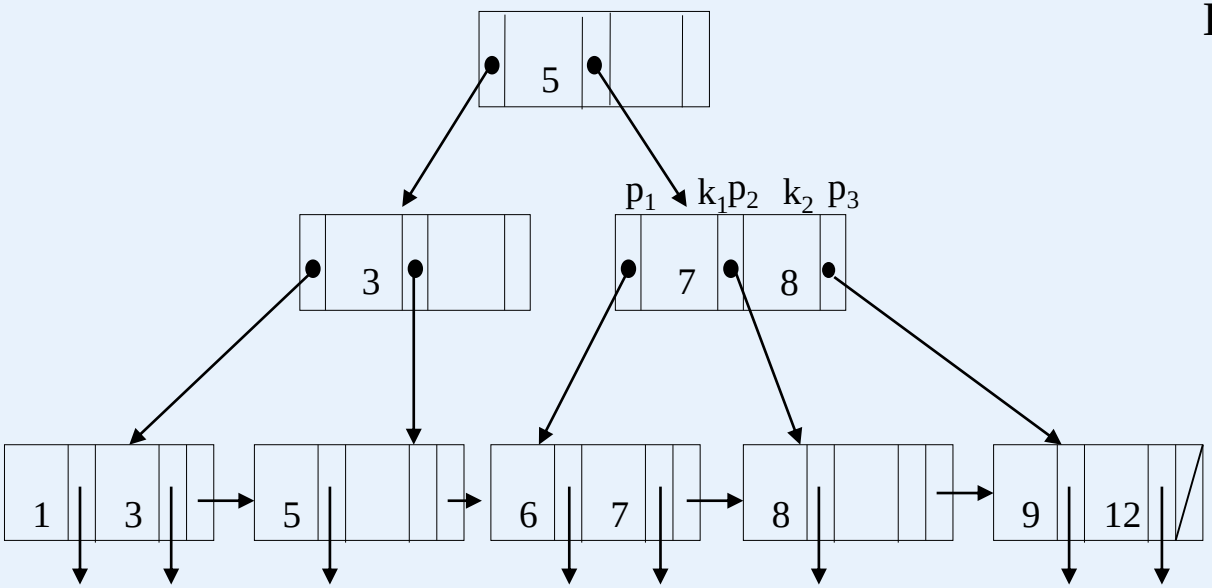
0	1	2	3
1	5	6	12
9	7	3	8

7, 8	0	2

⇒

6, 7	4	1
8	4	2
9,12	4	3

B+-Trees and Trees for Multidimensional Data



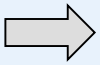
B+-tree stored in a file:

0	1	**5**	4		
1	2	**3**	3		
2	**1**	0	**3**	3	
3	**5**	0			
4	5	**7**		**8**	
5	**6**	1	**7**	2	
6					
7					

Data file:

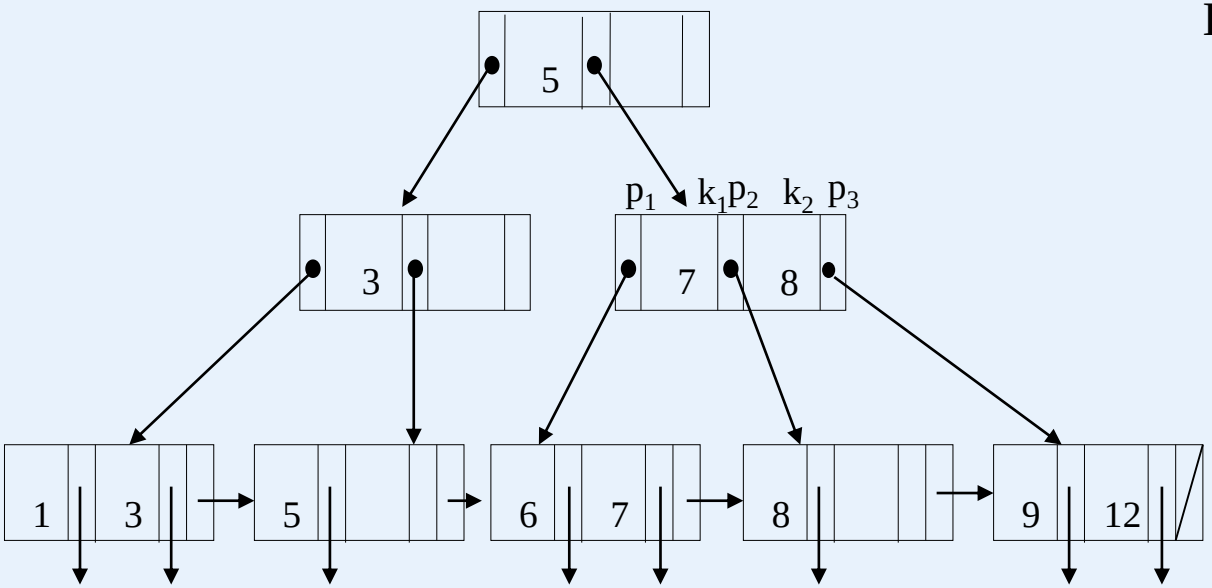
1	5	6	12	9	7	3	8	
0		1		2		3		

6, 7	4	1
8	4	2
9,12	4	3



8	4	2
9,12	4	3

B+-Trees and Trees for Multidimensional Data



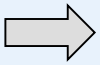
B+-tree stored in a file:

0	1	**5**	4		
1	2	**3**	3		
2	**1**	0	**3**	3	
3	**5**	0			
4	5	**7**	6	**8**	
5	**6**	1	**7**	2	
6	**8**	3			
7					

Data file:

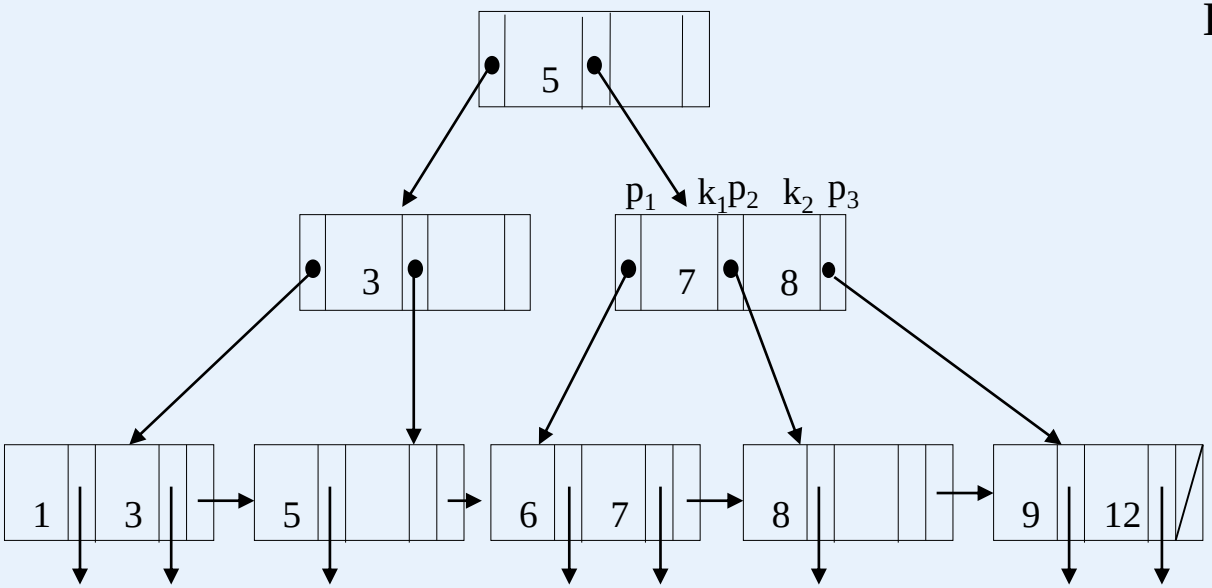
1	5	6	12	9	7	3	8	
0		1		2		3		

8	4	2
9,12	4	3



9,12	4	3

B+-Trees and Trees for Multidimensional Data

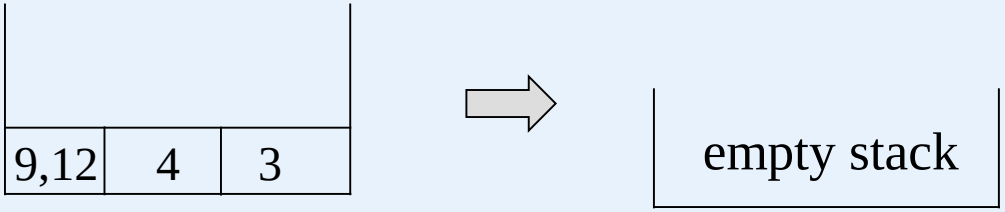


B+-tree stored in a file:

0	1	**5**	4			
1	2	**3**	3			
2	**1**	0	**3**	3		
3	**5**	0				
4	5	**7**	6	**8**	7	
5	**6**	1	**7**	2		
6	**8**	3				
7	**9**	2	**12**	1		

Data file:

1	5	6	12	9	7	3	8	
0		1		2		3		



Summary

- **B+-tree structure**
- **B+-tree construction**

  A process of key insertion into a B+-tree data structure
- **B+-tree maintenance**

  Deletion of keys from a B+-tree:

    Redistribution of nodes

    Merging of nodes

B+-tree operations

- search - always the same search length - tree height
- retrieval - sequential access is facilitated - how?
- insert - may cause overflow - tree may grow
- delete - may cause underflow - tree may shrink

What do you expect for storage utilization?

Index Structures for Multidimensional Data

- **Multiple-key indexes**
- ***kd*-trees**
- **Quad trees**
- **R-trees**
- **Bit map**

Indexes over texts

- **Inverted files**

Multiple-key indexes

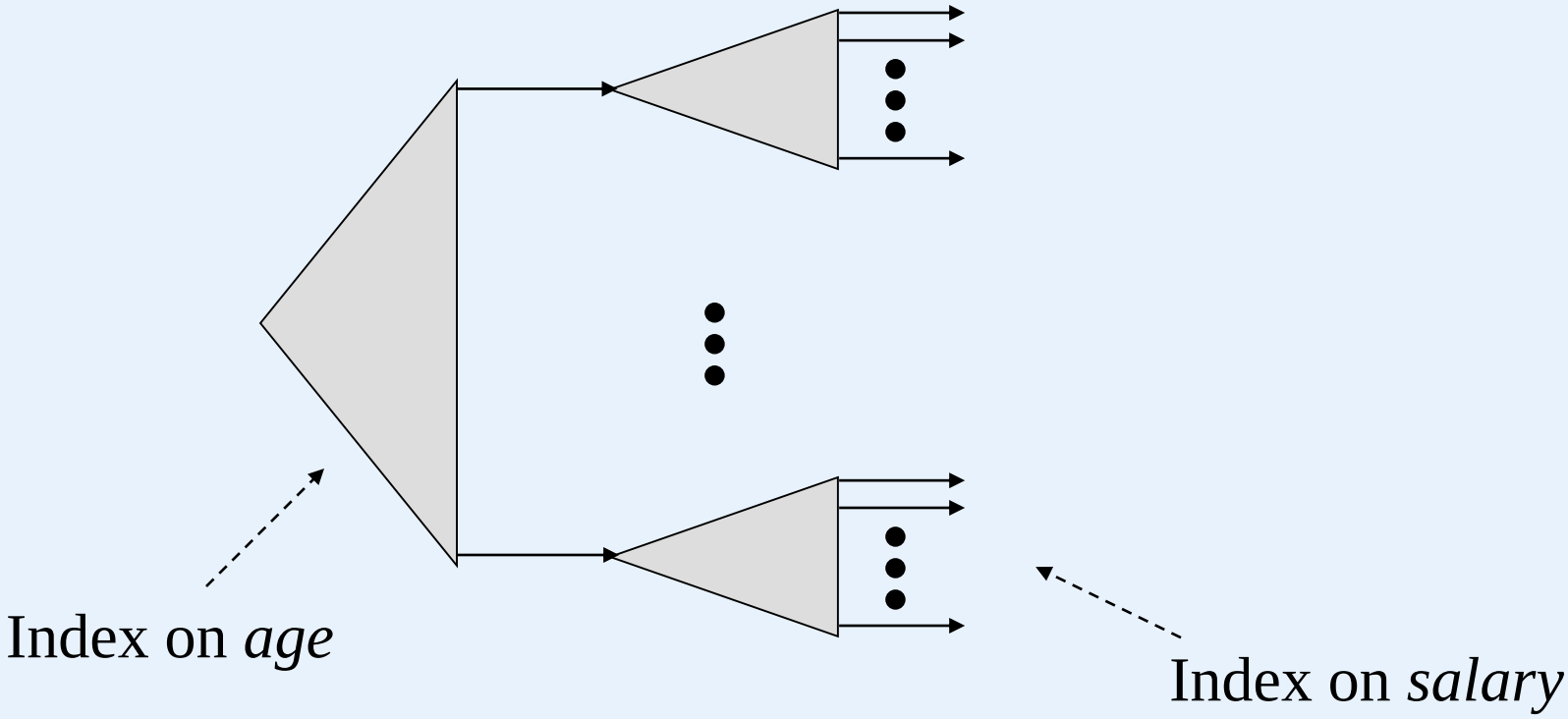
(Indexes over more than one attributes)

Employee

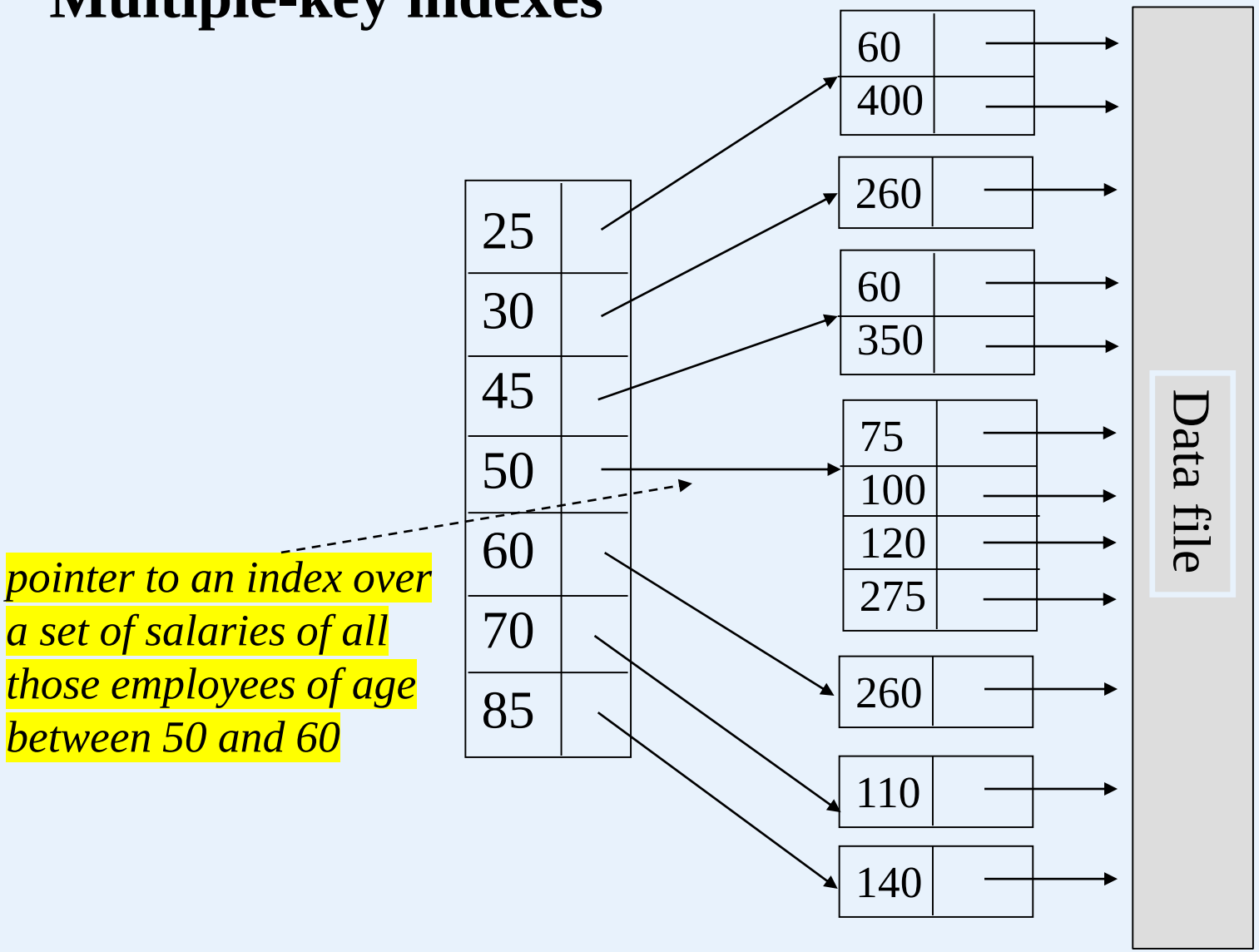
ename	ssn	age	salary	dnumber
Aaron, Ed				
Abbott, Diane				
Adams, John				
Adams, Robin				

Multiple-key indexes

(Indexes over more than one attributes)



Multiple-key indexes

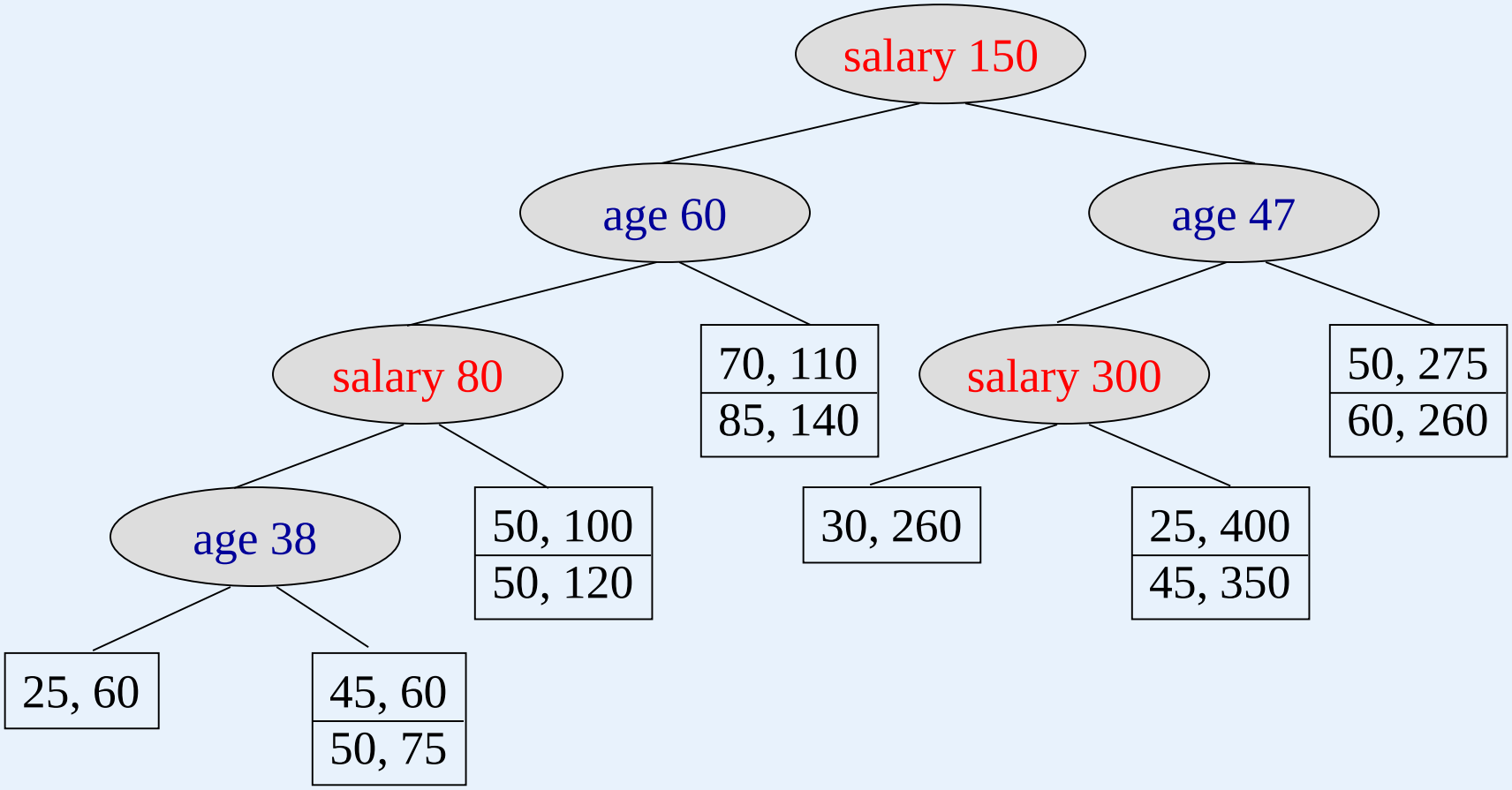


kd-Trees

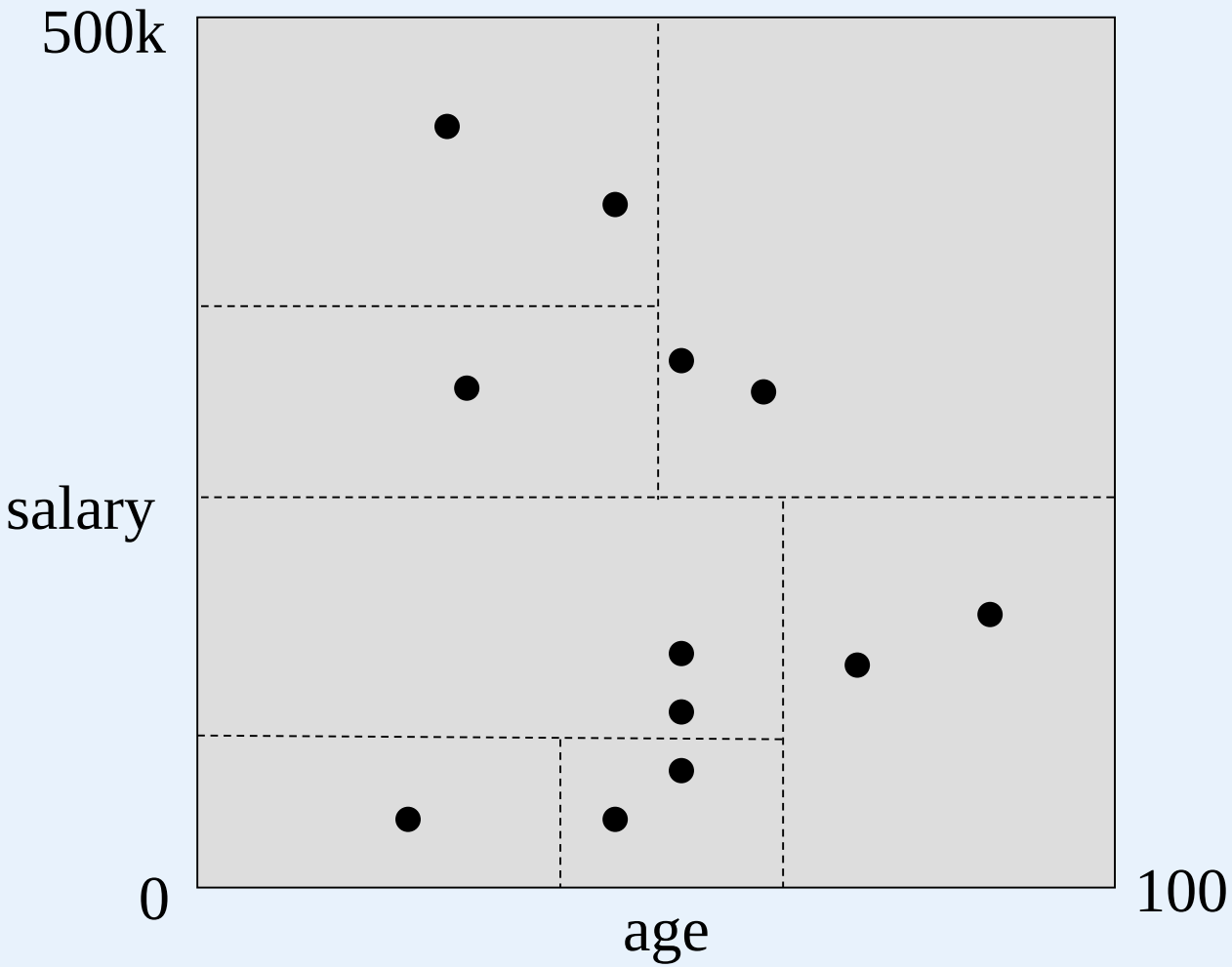
(A generalization of binary search trees)

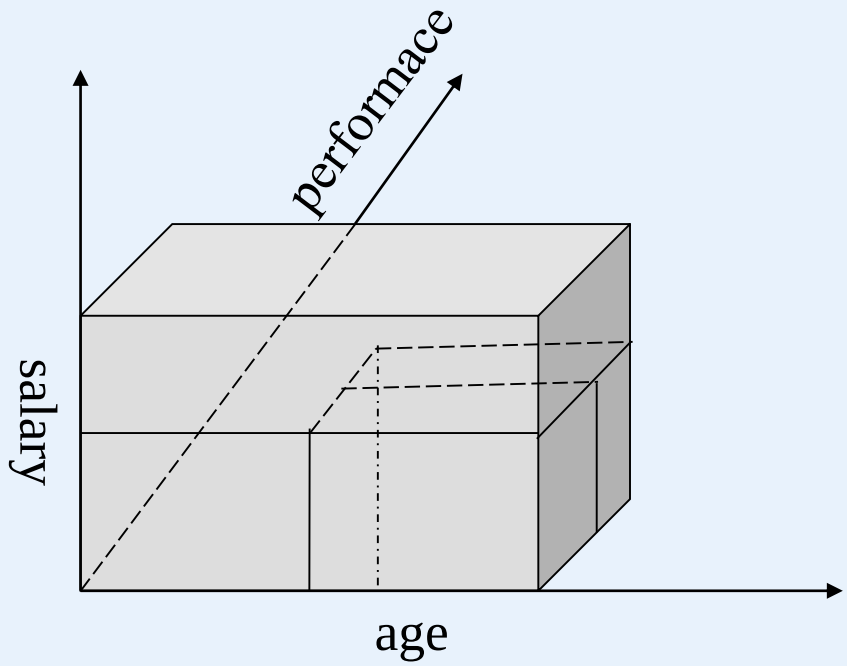
A *kd*-tree is a binary tree in which interior nodes have an associated attribute *a* and a value *v* that splits the data points into two parts: those with *a*-value less than *v* and those with *a*-value equal to or larger than *v*.

kd-Trees

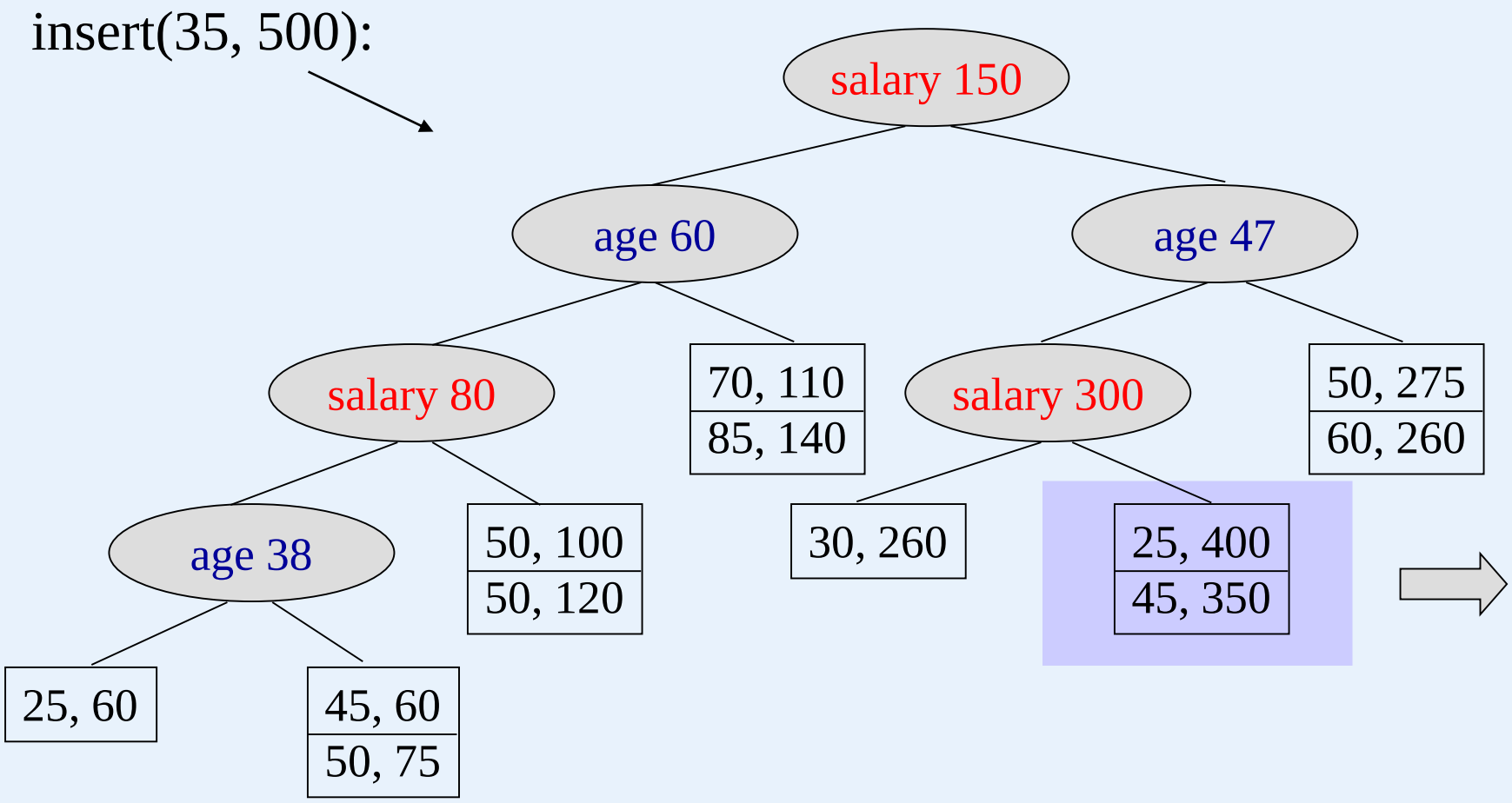

salary 150
age 60
age 47
salary 80
70, 110
85, 140
salary 300
50, 275
60, 260
age 38
50, 100
50, 120
30, 260
25, 400
45, 350
25, 60
45, 60
50, 75

kd-trees

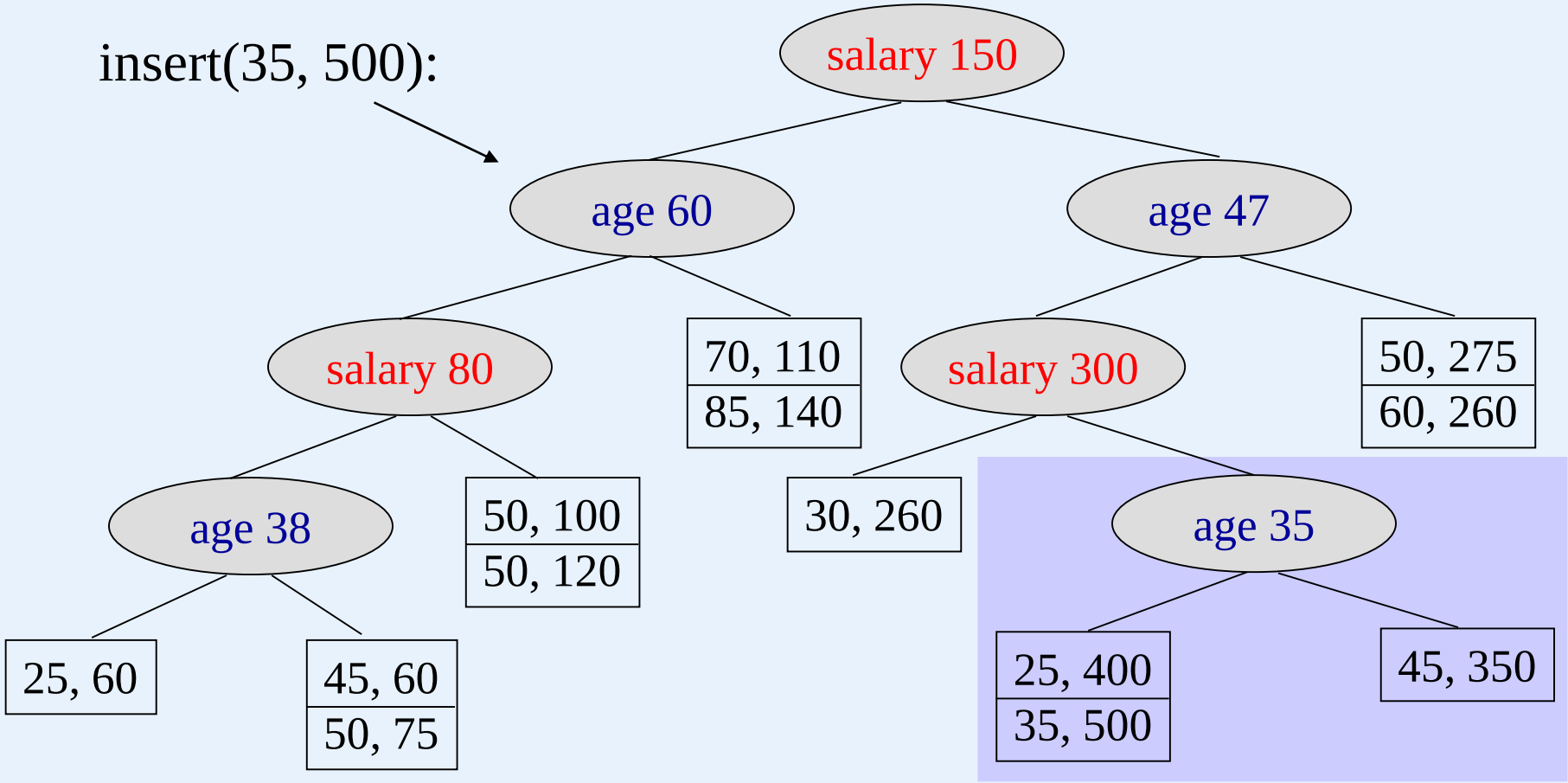



performace
salary
age

Insert a new entry into a *kd*-tree:



Insert a new entry into a *kd*-tree:



Quad-trees

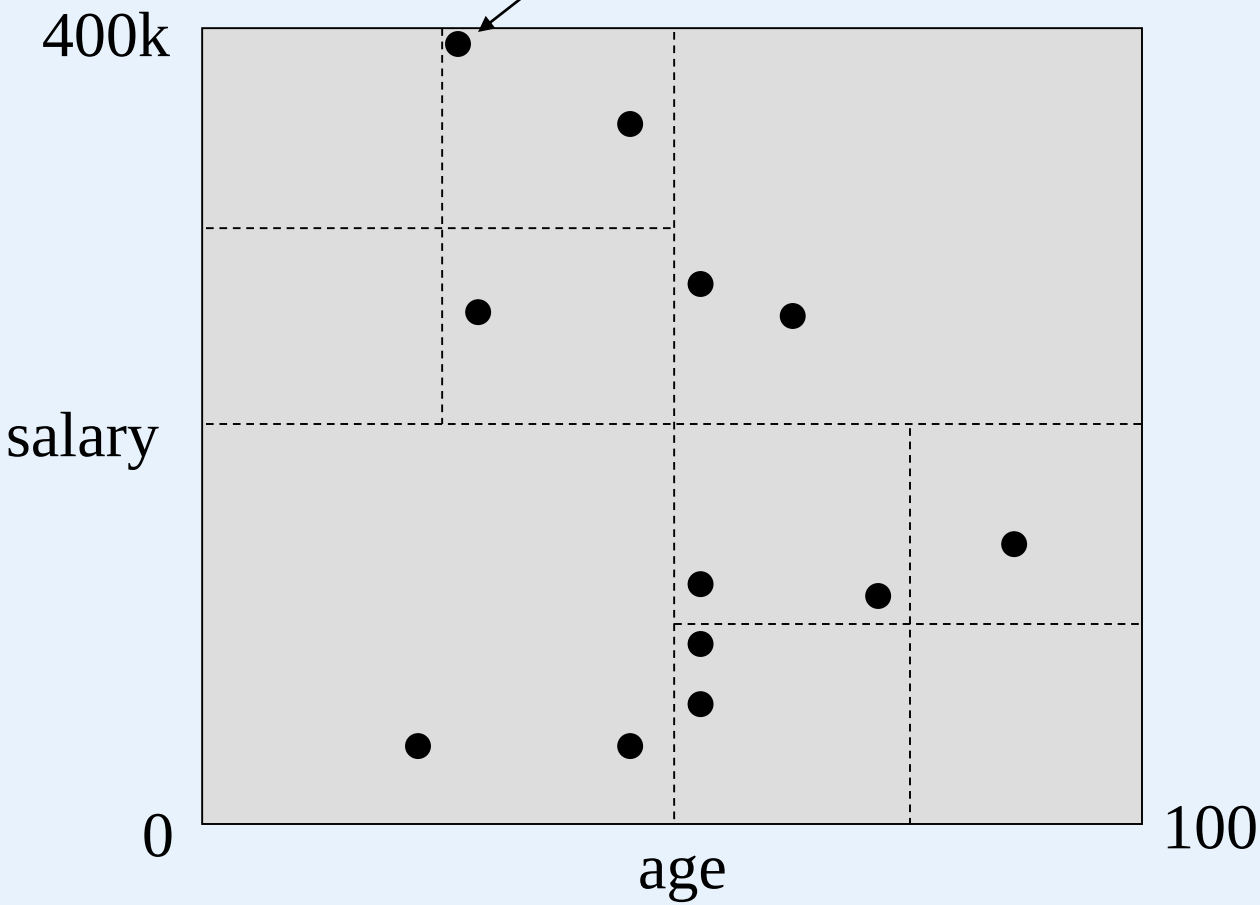
In a Quad-tree, each node corresponds to a square region in two dimensions, or to a *k*-dimensional cube in *k* dimensions.

- If the number of data entries in a square is not larger than what will fit in a block, then we can think of this square as a leaf node.

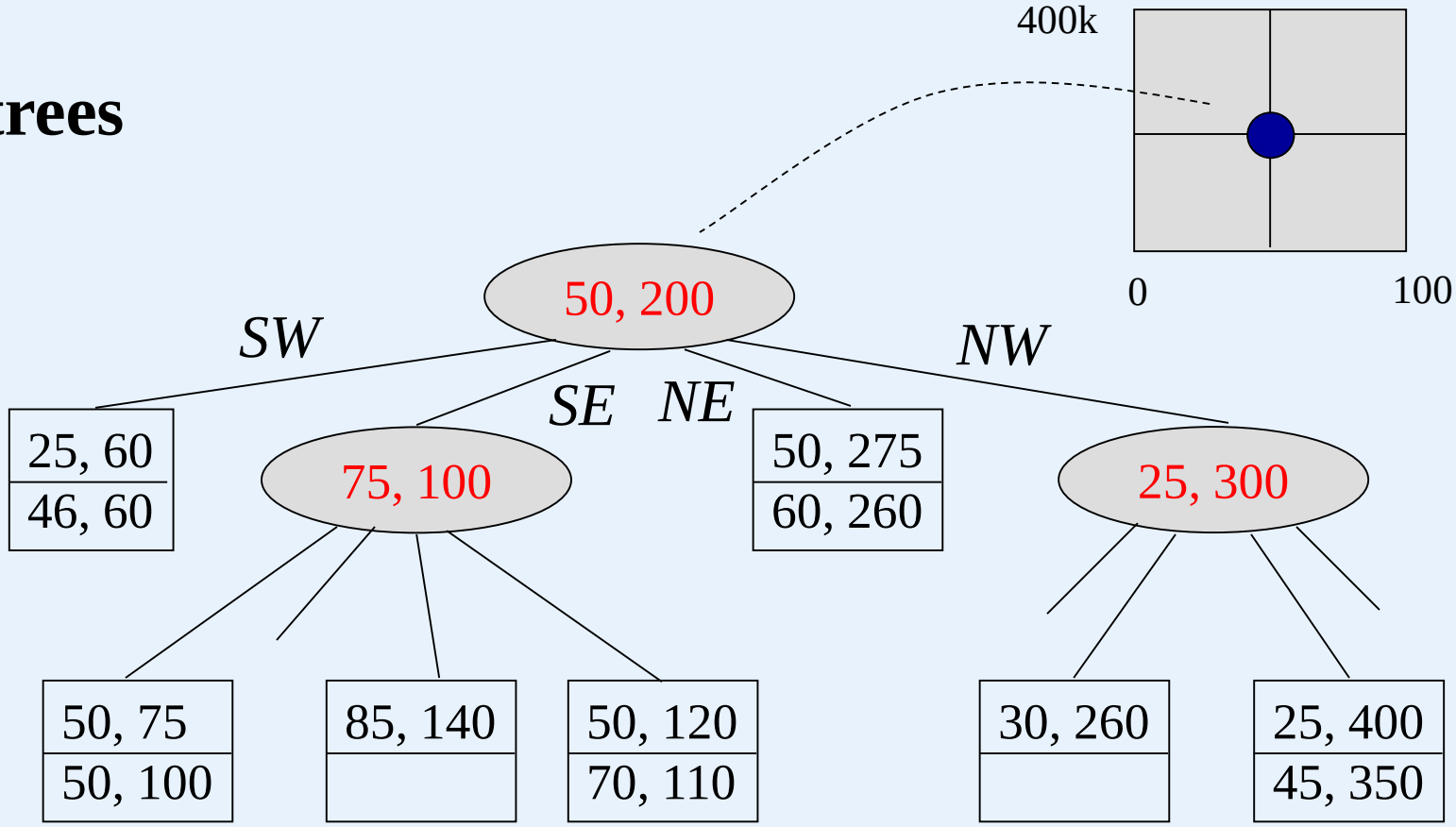
- If there are too many data entries to fit in one block, then we treat the square as an interior node, whose children correspond to its four quadrants.

Quad-trees

name	age	…	salary	…
…	25	…	400	…



Quad-trees



NW – north-west
NE – north-east
SW – south-west
SE – south-east

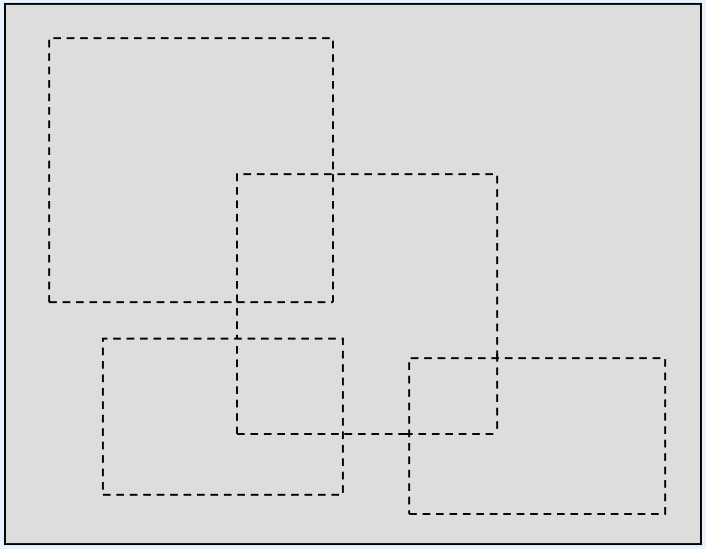
R-trees

An R-tree is an extension of B-trees for multidimensional data.

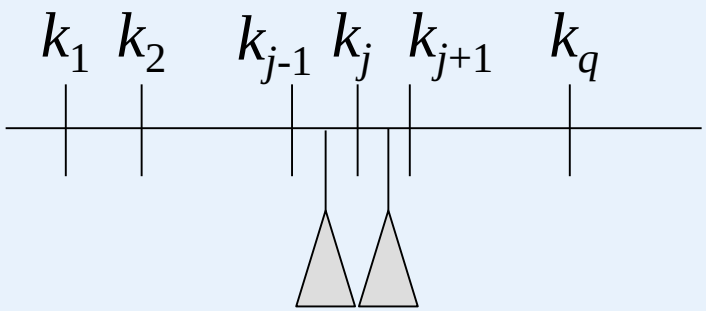
- An R-tree corresponds to a whole area (a rectangle for two-dimensional data.)
- In an R-tree, any interior node corresponds to some interior regions, or just regions, which are usually a rectangle
- Each region *x* in an interior node *n* is associated with a link to a child of *n*, which corresponds to all the subregions within *x*.

R-trees

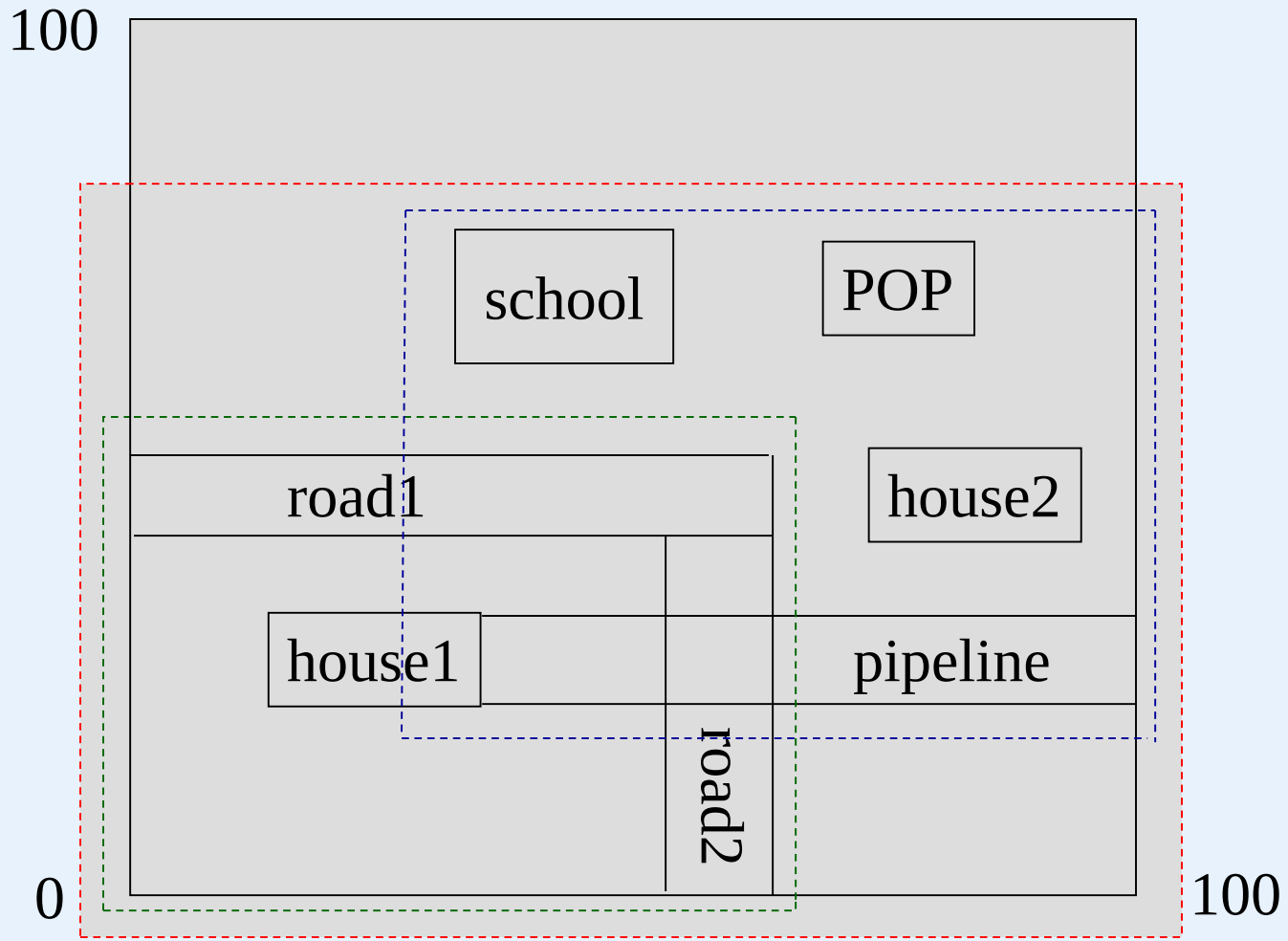
In an R-tree, each interior node contains several subregions.



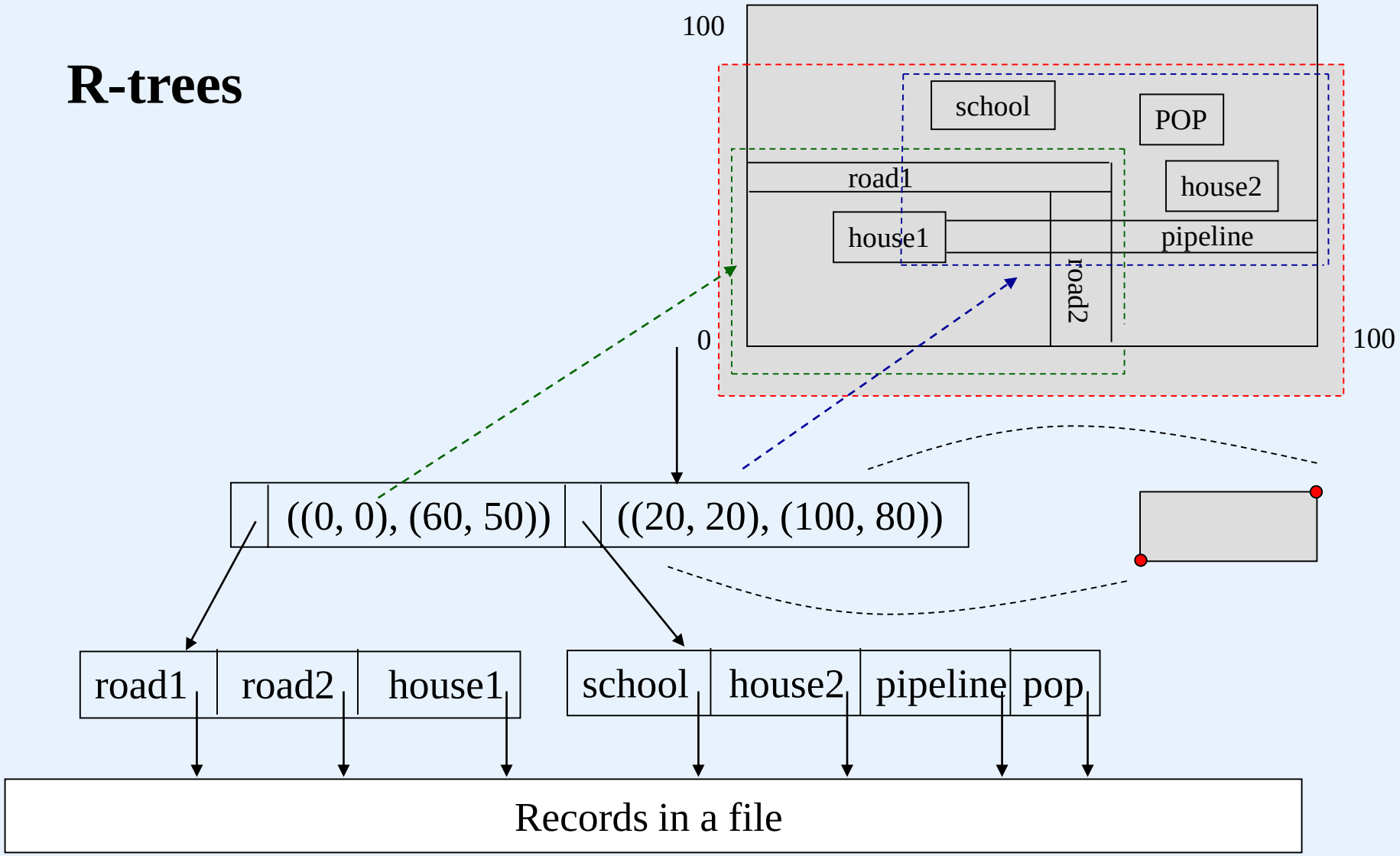
In a B+-tree, each interior node contains a set of keys that divides a line into segments.



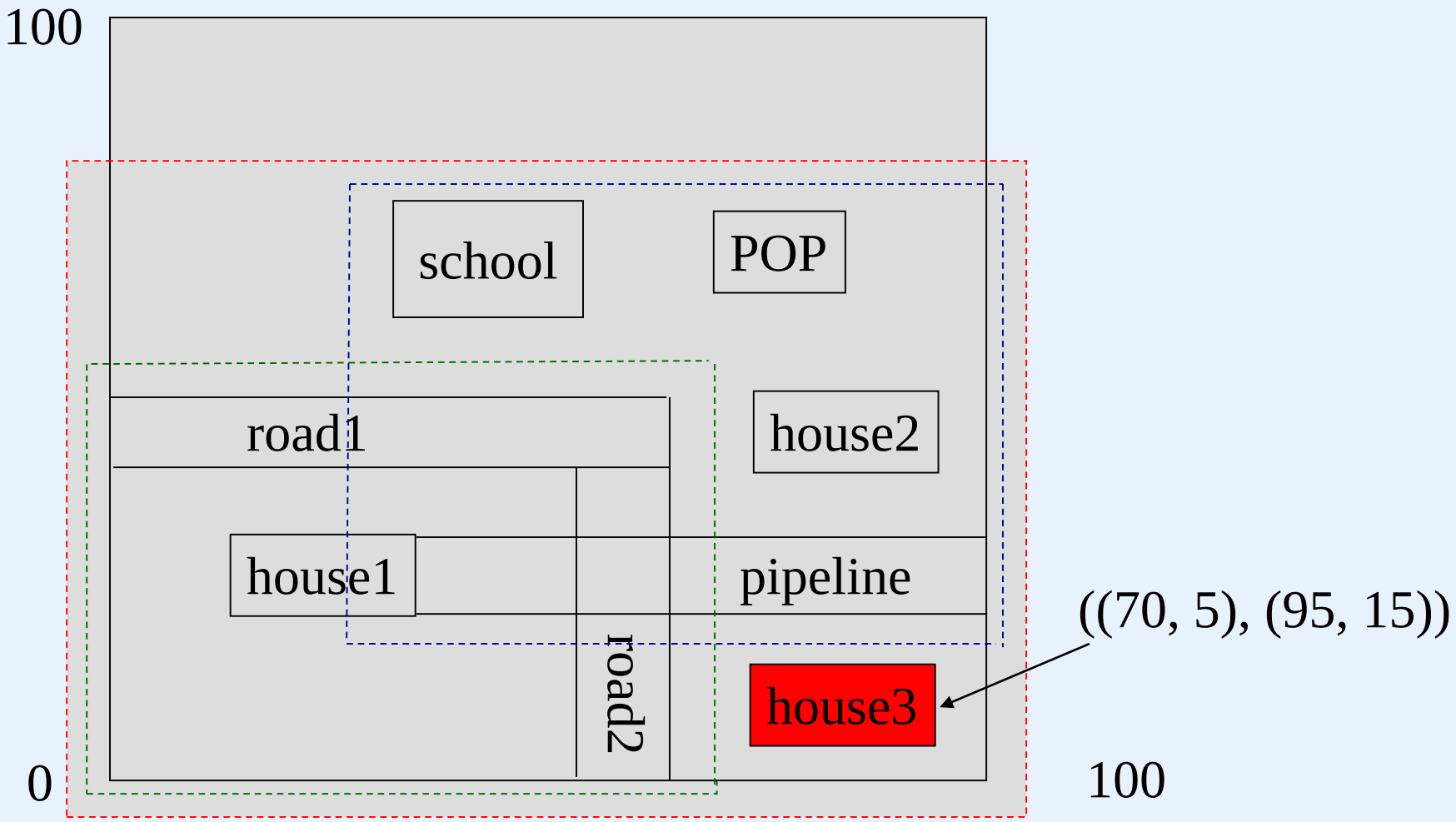
Suppose that the local cellular phone company adds a POP (point of presence, or base station) at the position shown below.



R-trees

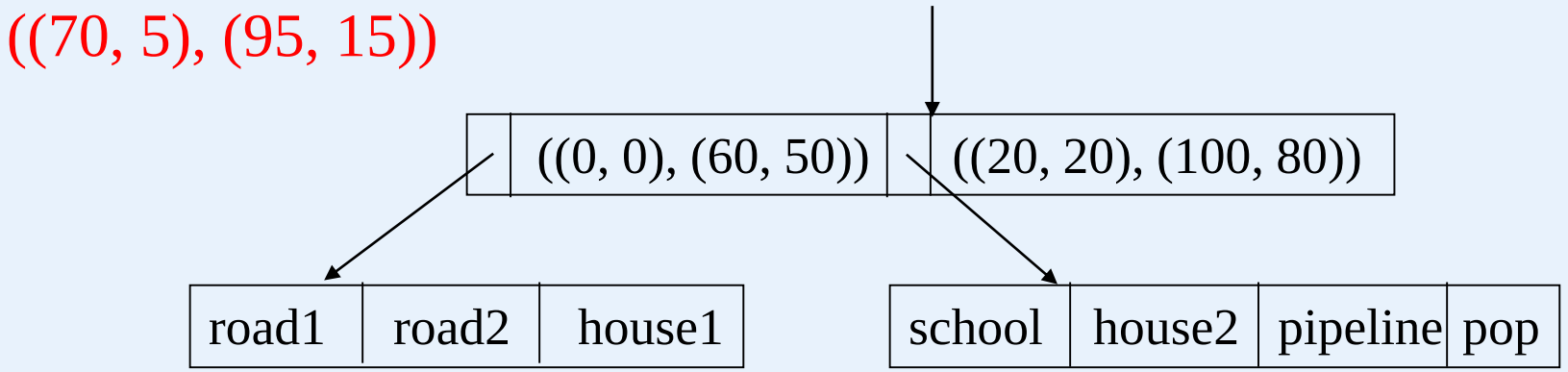

100
school
POP
road1
house2
house1
pipeline
road2
0
100
((0, 0), (60, 50))
((20, 20), (100, 80))
road1
road2
house1
school
house2
pipeline
pop
Records in a file

Insert a new region *r* into an R-tree.



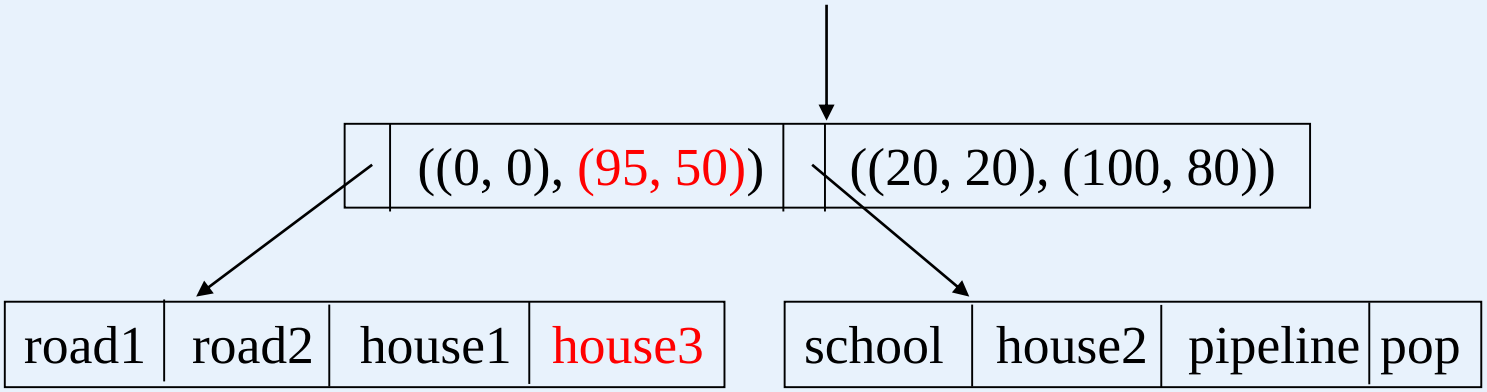
Insert a new region *r* into an R-tree.

1. Search the *R*-tree, starting at the root.
2. If the encountered node is internal, find a subregion into which *r* fits.
   - If there is more than one such region, pick one and go to its corresponding child.
   - If there is no subregion that contains *r*, choose any subregion such that it needs to be expanded as little as possible to contain *r*.

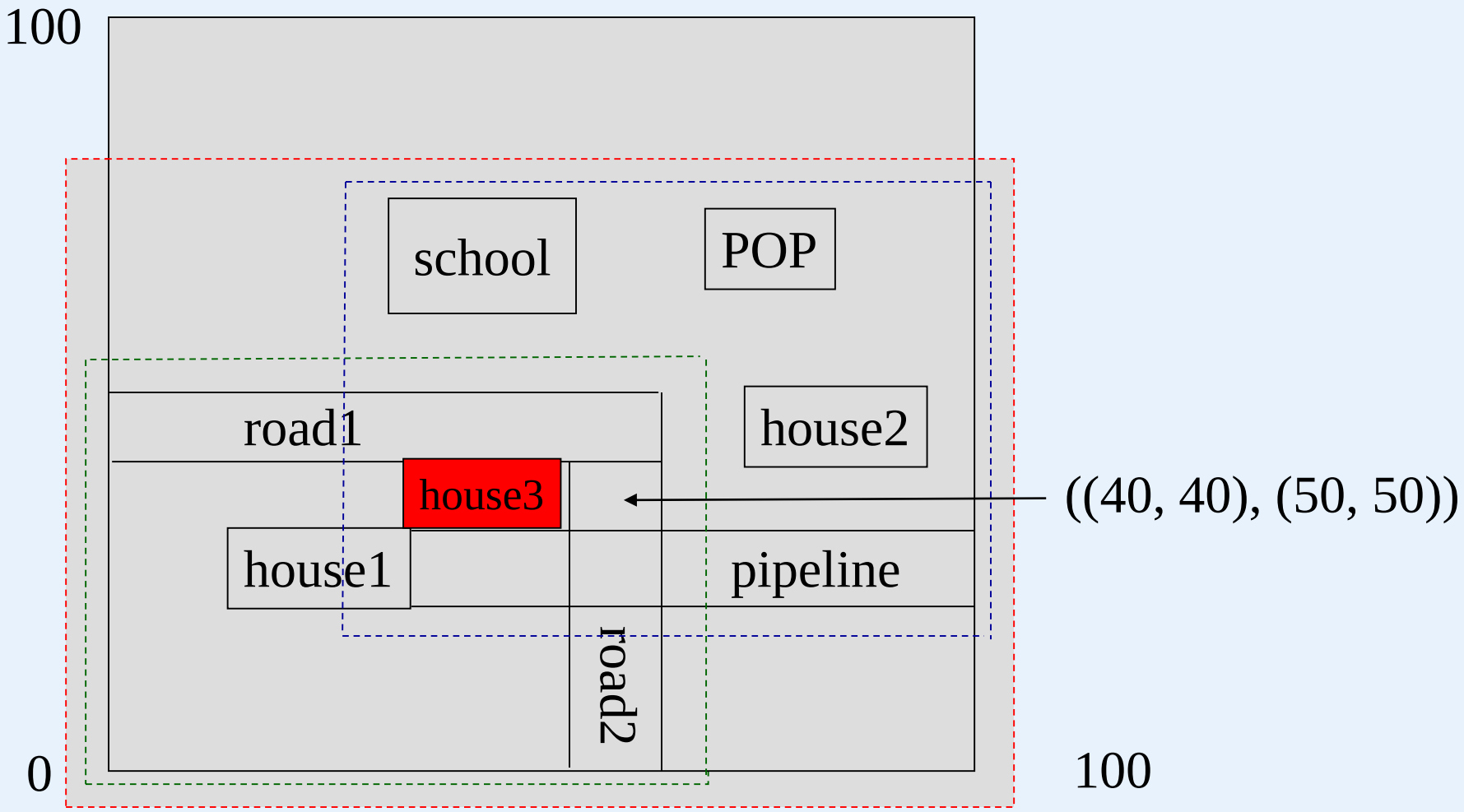


Two choices:

- If we expand the lower subregion, corresponding to the first leaf, then we add 1050 square units to the region.
- If we extend the other subregion by lowering its bottom by 15 units, then we add 1200 square units.



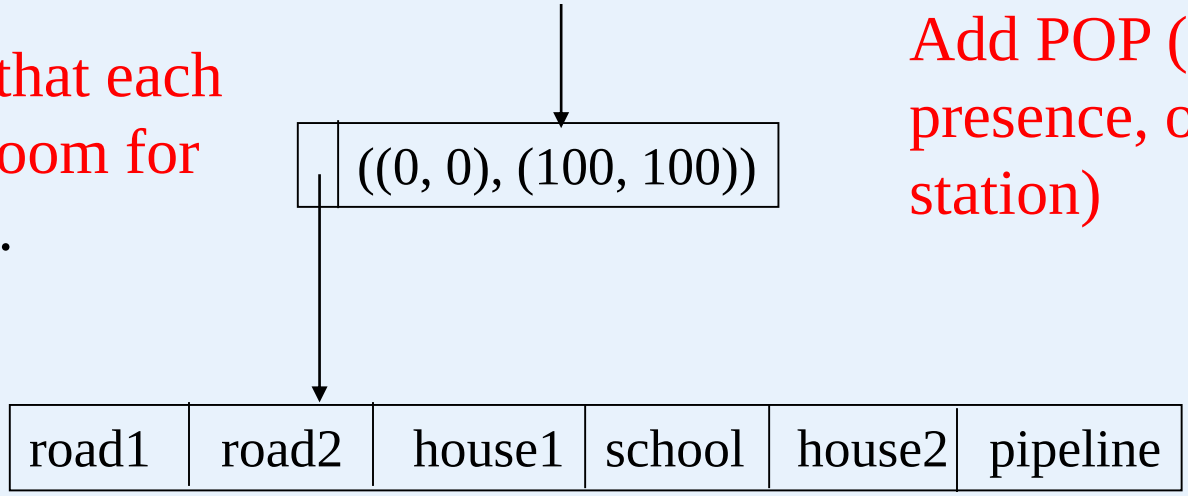
Insert a new region *r* into an R-tree.



Insert a new region *r* into an R-tree.

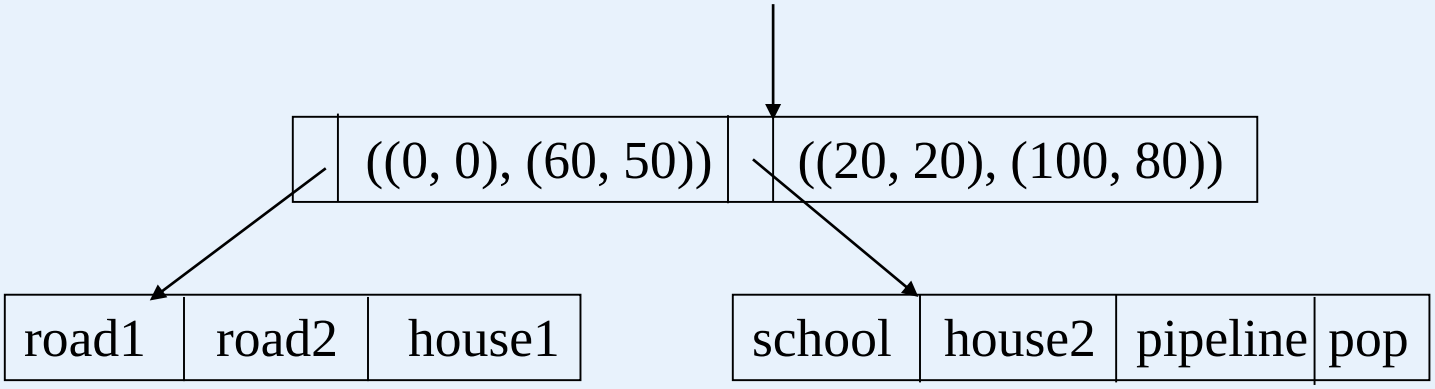
3. If the encountered node *v* is a leaf, insert *r* into it. If there is no room for *r*, split the leaf into two and distribute all subregions in them as evenly as possible. Calculate the 'parent' regions for the new leaf nodes and insert them into *v*'s parent. If there is the room at *v*'s parent, we are done. Otherwise, we recursively split nodes going up the tree.

Suppose that each leaf has room for 6 regions.



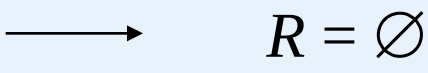
Add POP (point of presence, or base station)

- Split the leaf into two and distribute all the regions evenly.
- Calculate two new regions each covering a leaf.

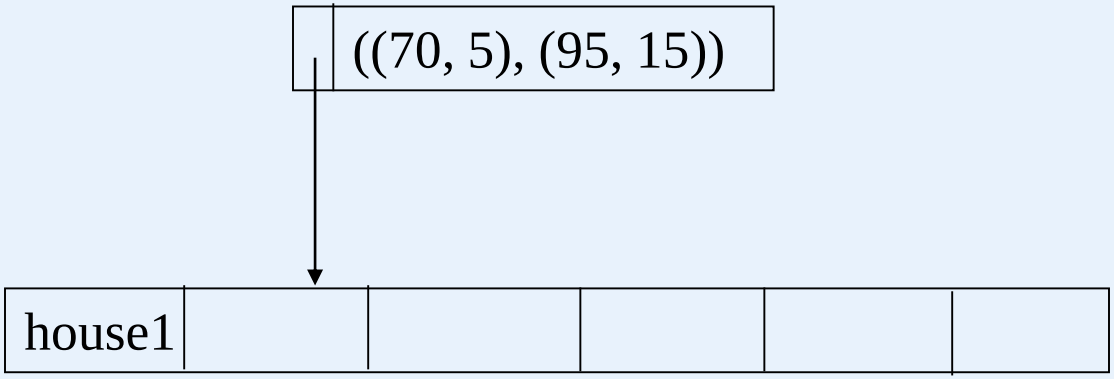


Insert the first object into an R-tree:

house1
((70, 5), (95, 15))



$R = \varnothing$



Bit map

1. Imagine that the records of a file are numbered 1, …, *n*.
2. A bitmap for a data field *F* is a collection of bit-vectors of length *n*, one for each possible value that may appear in the field *F*.
3. The vector for a specific value *v* has 1 in position *i* if the *i*th record has *v* in the field *F*, and it has 0 there if not.

Example

Employee

ename	ssn	age	salary	dnumber
Aaron, Ed		30	60	
Abbott, Diane		30	60	
Adams, John		40	75	
Adams, Robin		50	75	
Brian, Robin		55	78	
Brian, Mary		55	80	
Widom, Jones		60	100	

Bit maps for *age*:

30: 1100000
40: 0010000
50: 0001000
55: 0000110
60: 0000001

Bit maps for *salary*:

60: 1100000
75: 0011000
78: 0000100
80: 0000010
100: 0000001

Query evaluation

```
Select ename
From Employee
Where age = 55 and salary = 80
```

In order to evaluate this query, we intersect the vectors for *age* = 55 and *salary* = 80.

	0000110	---------- vector for *age* = 55
∧	0000010	---------- vector for *salary* = 80
	0000010	

This indicates the 6th tuple is the answer.

Range query evaluation

```
Select ename
From Employee
Where 40 ≤ age ≤ 50 and 50 ≤ salary ≤ 78
```

We first find the bit-vectors for the age values in (30, 50); there are only two: 0010000 and 0001000 for 40 and 50, respectively.

Take their bitwise OR: $0010000 \vee 0001000 = 0011000$.

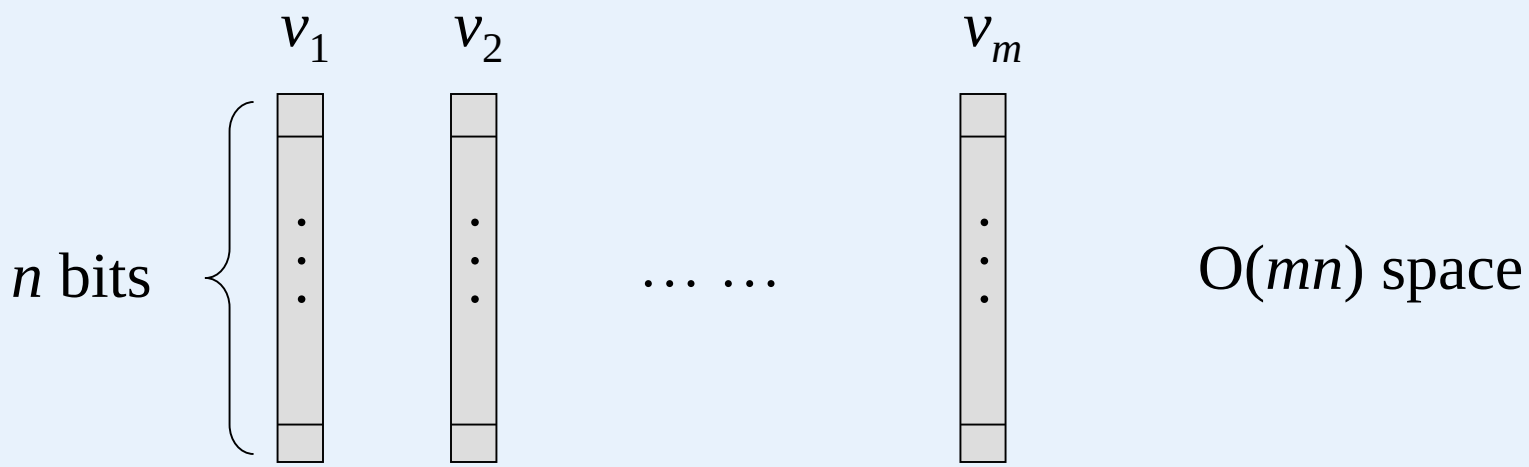
Next find the bit-vectors for the salary values in (50, 78) and take their bitwise OR: $1100000 \vee 0011000 \vee 0000100 = 1111100$.

$$\begin{array}{rr} & 0011000 \\ \wedge & 1111100 \\ \hline & 0011000 \end{array}$$

The 3rd and 4th tuples are the answer.

Compression of bitmaps

Suppose we have a bitmap index on field *F* of a file with *n* records, and there are *m* different values for field *F* that appear in the file.



O(*mn*) space

Compression of bitmaps

Run-length encoding:

Run in a bit vector: a sequence of *i* 0's followed by a 1.

000000010001 ← This bit vector contains two runs.

Run compression: a run *r* is represented as another bit string *r'* composed of two parts.

part 1: *i* expressed as a binary number, denoted as $b_1(i)$.
part 2: Assume that $b_1(i)$ is *j* bits long. Then, part 2 is a sequence of $(j - 1)$ 1's followed by a 0, denoted as $b_2(i)$.

$r' = b_2(i)b_1(i)$.

Compression of bitmaps

Run-length encoding:

Run in a bit vector *s*: a sequence of *i* 0's followed by a 1.

000000010001 ← This bit vector contains two runs.

$r' = b_2(i)b_1(i)$.

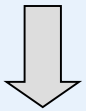
$r_1 = 00000001$

$b_{11} = 7 = 111$, $b_{12} = 110$ ⇒ $r_1' = 110111$

$r_2 = 0001$

$b_{11} = 3 = 11$, $b_{12} = 10$ ⇒ $r_2' = 1011$

000000010001



$r_1'r_2' = 1101111011$

Starting at the beginning, find the first 0 at the 3rd bit, so $j = 3$. The next 3 bits are 111, so we determine that the first integer is 7. In the same way, we can decode1011.

Decoding a compressed sequence *s*:

1. Scan *s* from the beginning to find the first 0.
2. Let the first 0 appears at position *j*. Check the next *j* bits. The corresponding value is a run.
3. Remove all these bits from *s*. Go to (1).

Uncompression:

$r_1'r_2' = 1101111011$

$r_1 = 00000001$

$r_2' = 1011$

$r_1r_2 = 000000010001$

$r_2 = 0001$

Question:

We can put all the compressed bit vectors together to get a bit sequence:

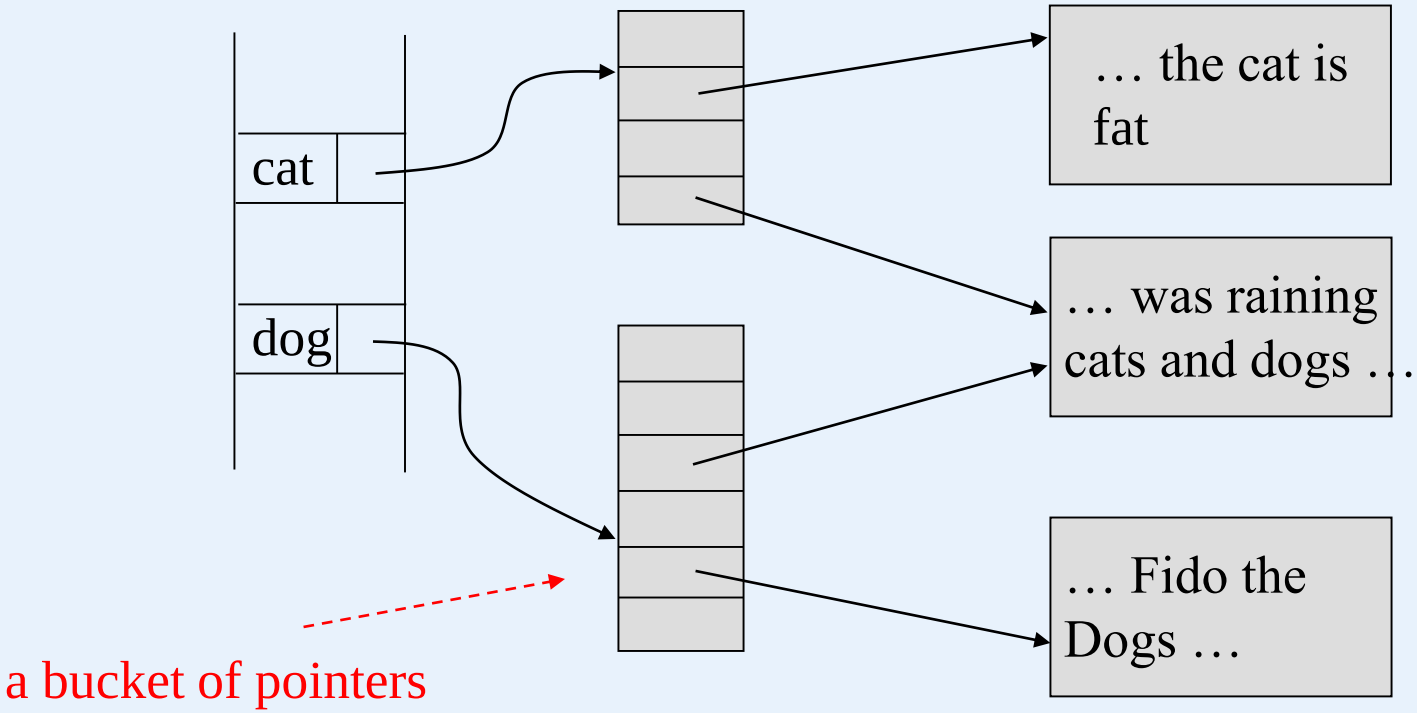
$$\boldsymbol{s} = s_1 s_2 \ldots s_m,$$

where s_i is the compressed bit string for the ith bit vector.

When decoding a certain s_j, how to differentiate between consecutive bit vectors?

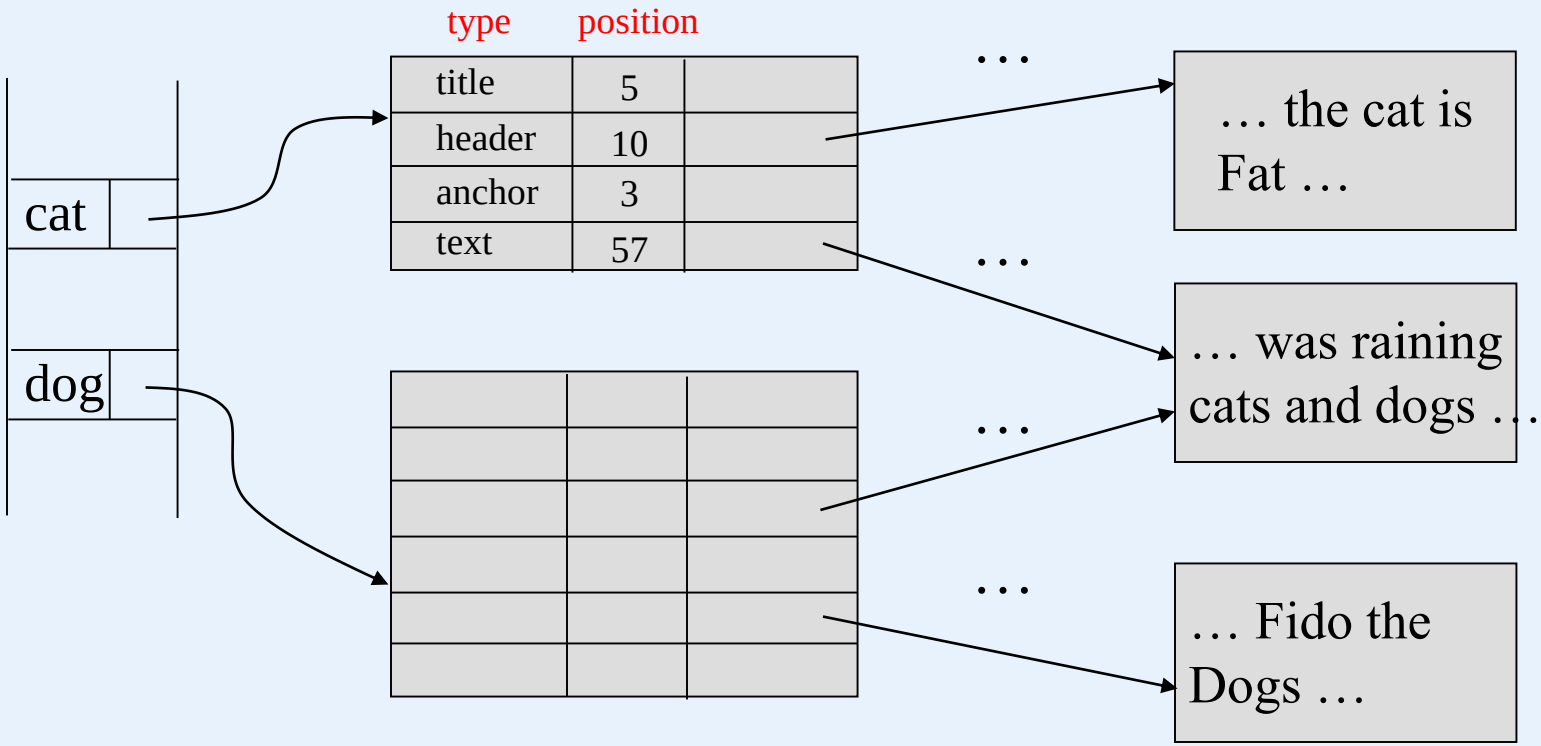
Inverted files

An inverted file - A list of pairs of the form: <key word, pointer>



$L(\text{cat}) = \{1, 3, 5\}$ $L(\text{dog}) = \{3, 5, 8, 9\}$

$L(\text{cat} \wedge \text{dog}) = \{1, 3, 5\} \cap \{3, 5, 8, 9\} = \{3, 5\}$

Inverted files

When we use “buckets” of pointers to occurrences of each word, we may extend the idea to include in the bucket array some information about each occurrence.