CSCI317 – Database Performance Tuning

Indexing

Sionggo Japit
sjapit@uow.edu.au

B*-TREE

- The B*-tree index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.
- A B*-tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.
- Each nonleaf node in the tree has at most *n* and at least $\lceil (2 \times n - 1)/3 \rceil$ children, where *n* is fixed for a particular tree.

B*-TREE

- In a B*-tree, data pointers are stored only at the leaf nodes.
- The leaf nodes have an entry for every value of the search field, along with a data pointer to the record or to the block that contains this record if the search field is a key field.
- For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection. (Implementation dependent)

B*-TREE

- The leaf nodes of the B*-tree are usually linked together to provide ordered access on the search field to the records.
- These leaf nodes are similar to the first level of an index.
- Internal nodes of the B*-tree correspond to the other levels of a multilevel index.
- Some search field values from the leaf nodes are repeated in the internal nodes of the B*-tree to guide the search.

B*-TREE

The structure of the **internal nodes** of a B*-tree of order p is as follows:

1. Each internal node is of the form

   $<Pr_1, K_1, Pr_2, K_2, \ldots Pr_i, K_i, \ldots, Pr_{p-1}, K_{p-1}, Pr_p>$

   Where $i < p$ and each Pr_i is a node pointer.

2. Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$.

B*-TREE

The structure of the **internal nodes** of a B*-tree of order p is as follows: (continue…)

3. For all search field values X in the subtree pointed at by Pr_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < p$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = p$.

4. Each internal node, except the root, has at most $\lceil (2 \times p - 1)/3 \rceil$ tree pointers.

B*-TREE

The structure of the **internal nodes** of a B*-tree of order p is as follows:

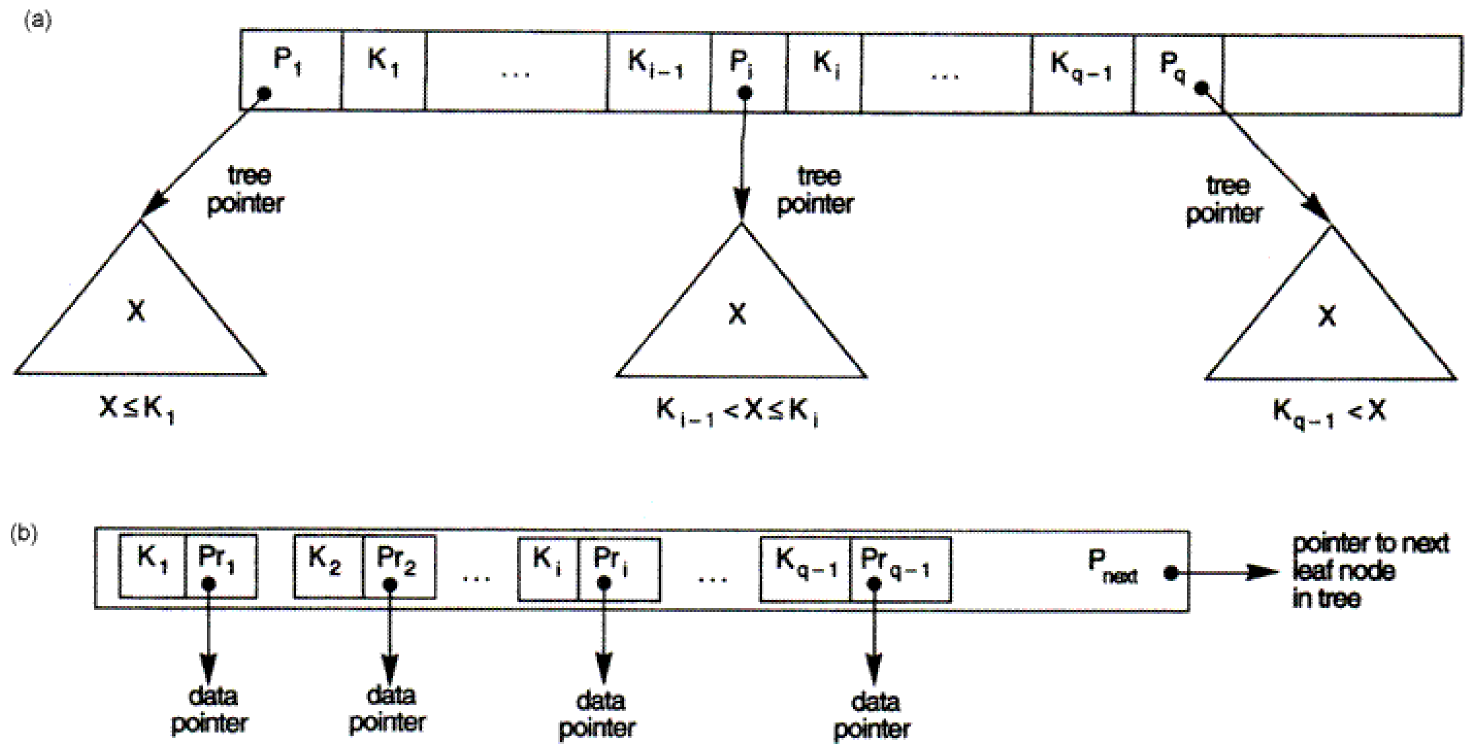
5. The root node has at least two and at most $2 \times \lfloor (2 \times p - 2) / 3 \rfloor + 1$ tree pointers.

B*-TREE

The structure of the **internal nodes** of a B*-tree of order p is as follows:

6. Non-root node splits involve 2 full nodes splitting into 3 nodes. The resulting 3 nodes have the following number of entries, listed from left to right accordingly with the positions of the nodes in the tree:
   - $\lfloor (2 \times p - 2) / 3 \rfloor$ $(\textit{left node})$
   - $\lfloor (2 \times p - 1) / 3 \rfloor$ $(\textit{middle node})$
   - $\lfloor (2 \times p) / 3 \rfloor$ $(\textit{right node})$

B*-TREE


(a)
P_1
K_1
...
K_{i-1}
P_i
K_i
...
K_{q-1}
P_q
tree pointer
tree pointer
tree pointer
X
X
X
$X \le K_1$
$K_{i-1} < X \le K_i$
$K_{q-1} < X$
(b)
K_1 Pr_1
K_2 Pr_2
K_i Pr_i
K_{q-1} Pr_{q-1}
P_{next}
pointer to next leaf node in tree
data pointer
data pointer
data pointer
data pointer

B*-TREE

The structure of the **leaf-node** of a B*-tree of order p is as follows:

1. Each leaf node is of the form

   $<<K_1,Pr_1>,<K_2,Pr_2>, \ldots , <K_{q-1},Pr_{q-1}>, P_{next}>$

   where $q < p$ and each Pr_i is a data pointer, and P_{next} points to the next leaf node of the B*-tree.

1. Within each leaf node, $K_1 < K_2 < \ldots < K_{q-1}$, where $q < p$.

B*-TREE

The structure of the **leaf-node** of a B*-tree of order p is as follows:

3. Each Pr_i is a data pointer that points to:
   - the record whose search field value is K_i or
   - to a file block containing the record or
   - to a block of record pointers that point to records whose search field value is K_i if the search field is not a key.

3. All leaf nodes are at the same level.

B*-TREE

EXAMPLE 4: To calculate the order of a B*-tree, suppose that the search key field is V = 9 bytes long, the block size is B = 512, a record pointer is Pr = 7 bytes, and a block pointer is P = 6 bytes as in previous examples. An internal node of the B*-tree can have up to p tree pointers and p-1 search field values; these must fit into a single block. Hence, we have:

$$(p * P) + ((p-1) * V) \leq B$$

$$(p * 6) + ((p-1) * 9) \leq 512$$

$$(15 * p) \leq 521$$

we can choose p to be the largest value satisfying the above inequality, which gives p = 34.

B*-TREE

The leaf nodes of the B*-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order pleaf for the leaf nodes can be calculated as follows:

$$(p_{leaf} * (Pr + V)) + P \leq B$$

$$(p_{leaf} * (7 + 9)) + 6 \leq 512$$

$$(16 * p_{leaf}) \leq 506$$

It follows that each leaf node can hold up to p_{leaf} = 31 key value/data pointer combinations, assuming that the data pointers are record pointers.

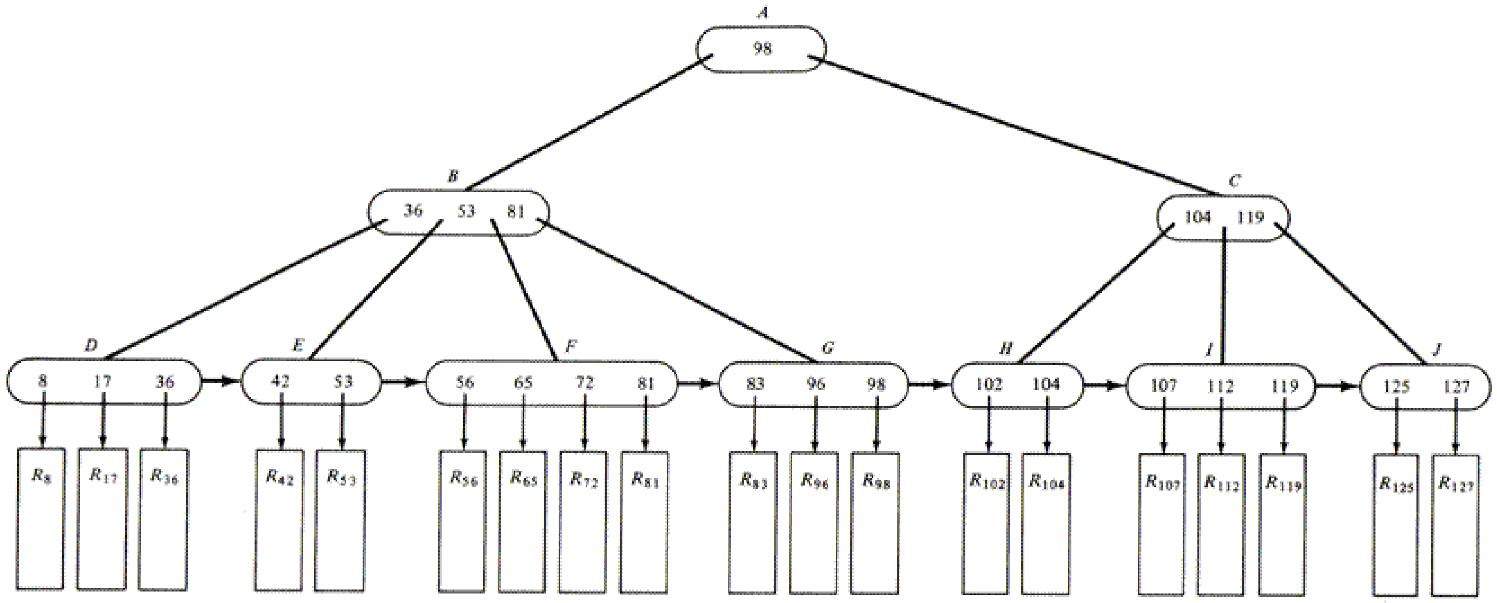
B*-TREE

EXAMPLE 5: Suppose that we construct a B*-tree on the filed of Example 4. To calculate the approximate number of entries of the B*-tree, we assume that each node is 69 percent full. On the average, each internal node will have 34 * 0.69 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold 0.69 * pleaf = 0.69 * 31 or approximately 21 data record pointers. A B*-tree will have the following average number of entries at each level:

Root:	1 node	22 entries	23 pointers
Level 1:	23 nodes	506 entries	529 pointers
Level 2:	529 nodes	11,638 entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 record pointers	

For the block size, pointer size, and search field size given above, a three-level B*-tree holds up to 255,507 record pointers, on the average.

B*-TREE



A
98
B
36 53 81
C
104 119
D
8 17 36
E
42 53
F
56 65 72 81
G
83 96 98
H
102 104
I
107 112 119
J
125 127
R_8
R_{17}
R_{36}
R_{42}
R_{53}
R_{56}
R_{65}
R_{72}
R_{81}
R_{83}
R_{96}
R_{98}
R_{102}
R_{104}
R_{107}
R_{112}
R_{119}
R_{125}
R_{127}

Insertion into B*-tree

procedure for insert into B*-tree

1. Starting at the root, search to a leaf node.
2. If the key value is found in the leaf node, you have a duplicate. Insertion fails. Stop. (This is assuming that duplicate key values are not allowed.)
3. If the key value is not found in the leaf and if there is room in the node, put the new key value there. If the new key value is not the largest key value in the leaf node, the insertion successful. Stop. If the new key value is the largest key value in the leaf node, proceed to Step 6.
4. If the key value is not found in the leaf and If there is **no** room for insertion, copy the key values in the current leaf node plus the new key value into a temporary node and proceed to Step 5.

Insertion into B*-tree

procedure for insert into B*-tree

5. Redistribute the key values and pointers such that half the keys go into the current (old) leaf node and half into a new leaf node. If there is an odd number of keys put the extra key in the current (old) leaf node. Order the key values so that the smaller ones go into the current (old) leaf node and the larger ones go into the new leaf node. All the keys in the current (old) leaf node will be smaller than any key value in the new node.
6. Duplicate the largest key value in the current (old) leaf node into the parent non-leaf node.
7. If there is no parent node create a new root node, adjust the pointers and stop. If there is room in the parent node, adjust the pointers, stop.
8. If the parent node is full, allocate a new non-leaf node.

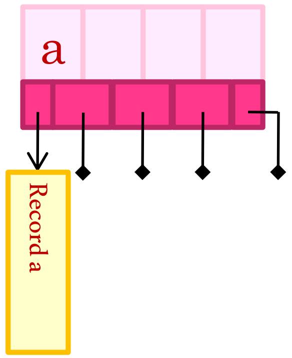
Insertion into B*-tree

9. Redistribute the key values that were in the old non-leaf node, plus
   the new key value, into the two non-leaf nodes, except the middle-valued key. Move the middle-valued key up the tree into the parent node. If there is an odd number of keys to be redistributed into the two nodes, put the extra one in the old(left) non-leaf node.
10. Go to step 7.
11. Stop.

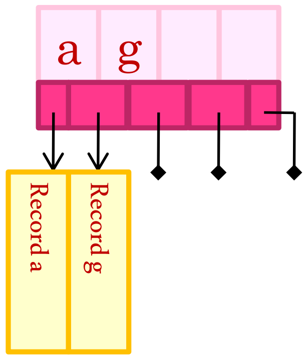
INSERTION INTO B*-TREE

Insert: a, g, f, b

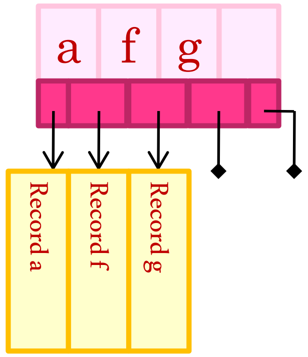
Insert: a



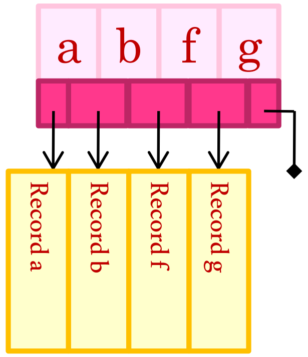
Insert: g



Insert: f

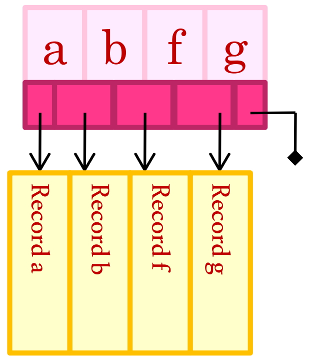


Insert: g



Insertion into B*-tree

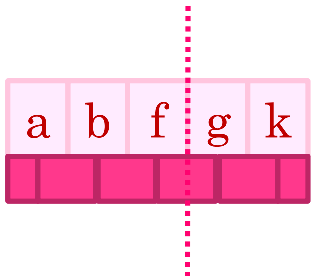
B*-tree after the insertion of a, g, f, and b.

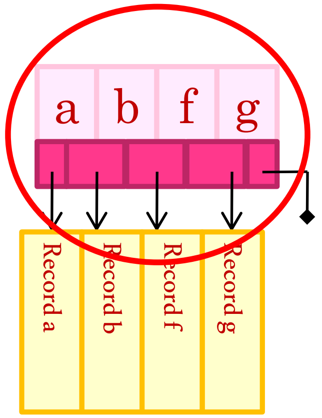


Insertion into B*-tree

Insert: **k**

Insert of **k** causes an overflow of node:

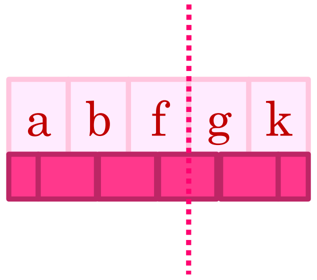


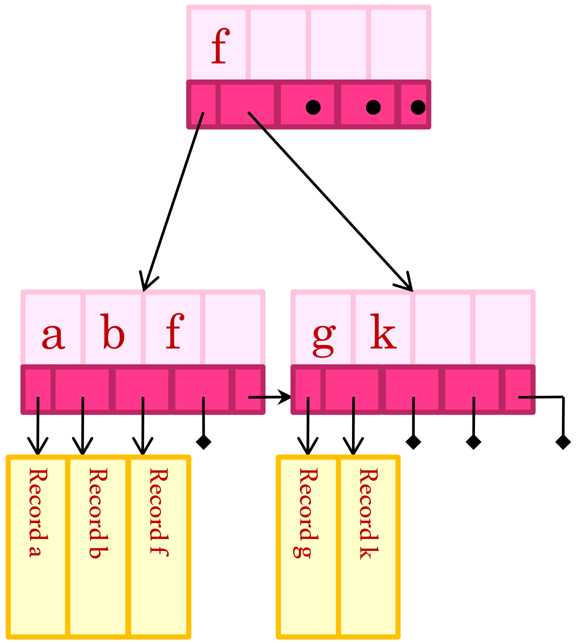


Insertion into B*-tree

Insert: **k**

Insert of **k** causes an overflow of node:

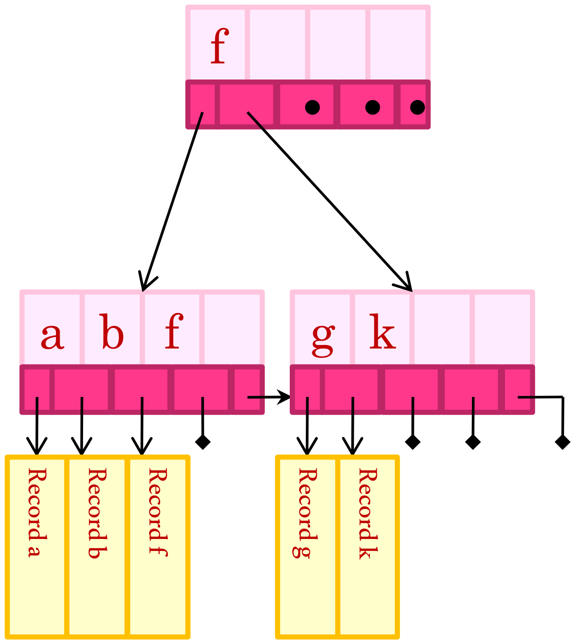




Split overflowed node into two and push-up a new node

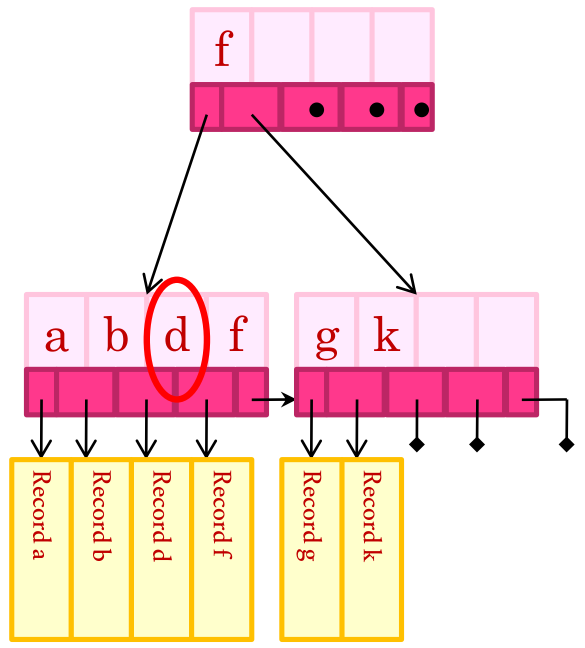
INSERTION INTO B*-TREE

B*-tree after the insertion of k.



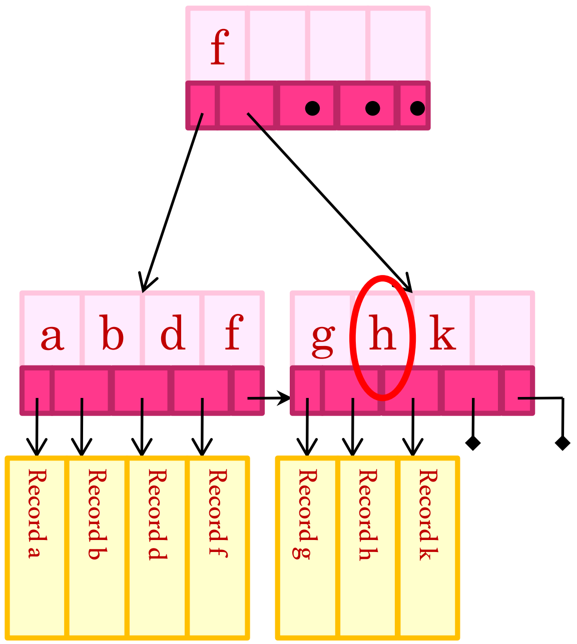
Insertion into B*-tree

Insert: **d**, h, m



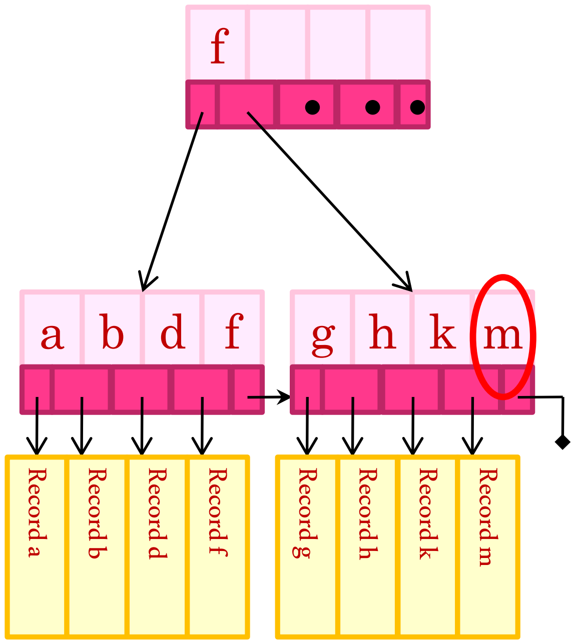
Insertion into B*-tree

Insert: d, **h**, m



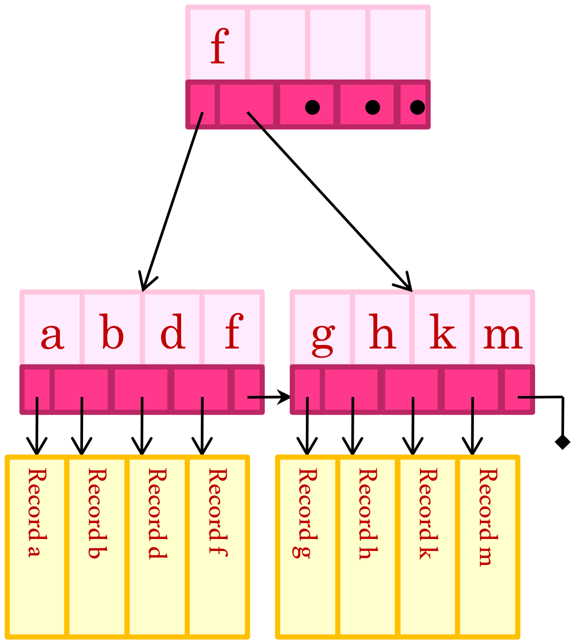
Insertion into B*-tree

Insert: d, h, **m**



Insertion into B*-tree

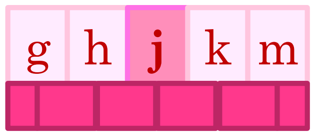
B*-tree after the insertion of d, h, and m.

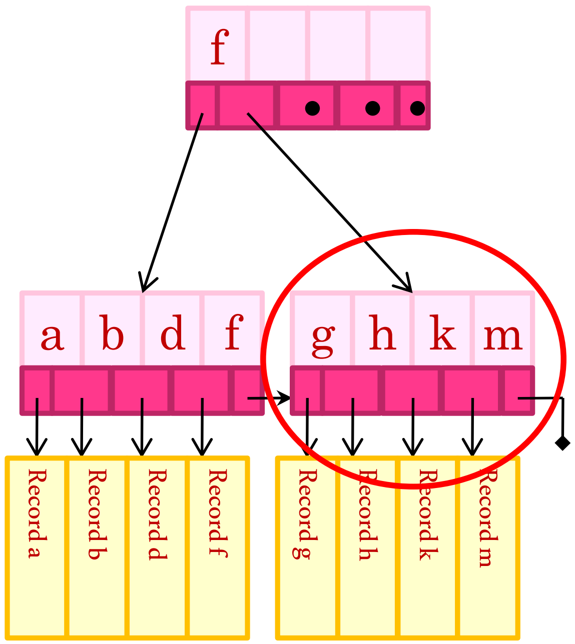


Insertion into B*-tree

Insert: **j**

Insert of **j** causes an overflow of node:

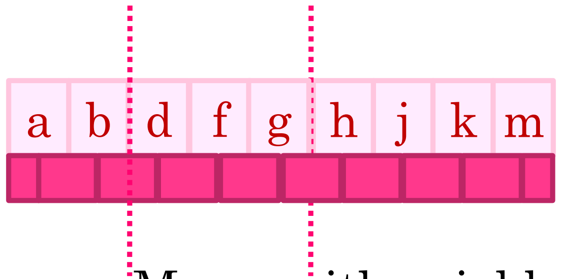




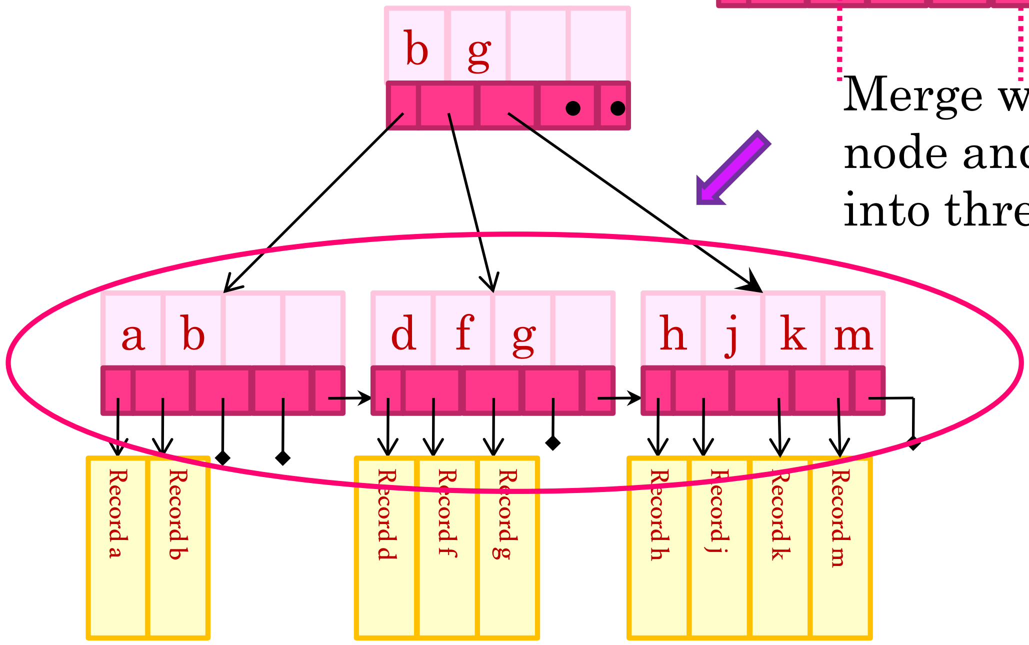
Insertion into B*-tree

Insert: **j**

Insert of **j** causes an overflow of node:

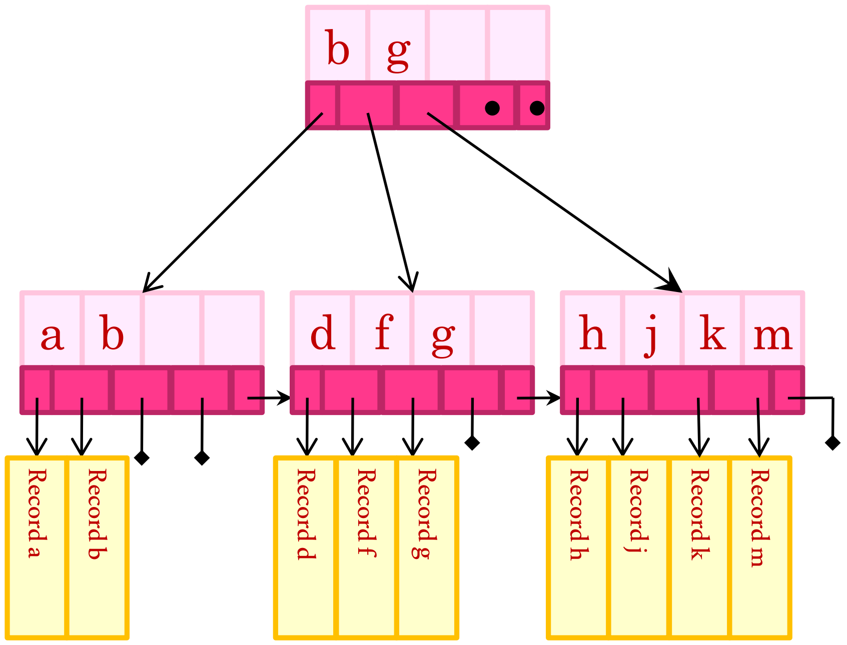


Merge with neighbor node and split nodes into three nodes.



Insertion into B*-tree

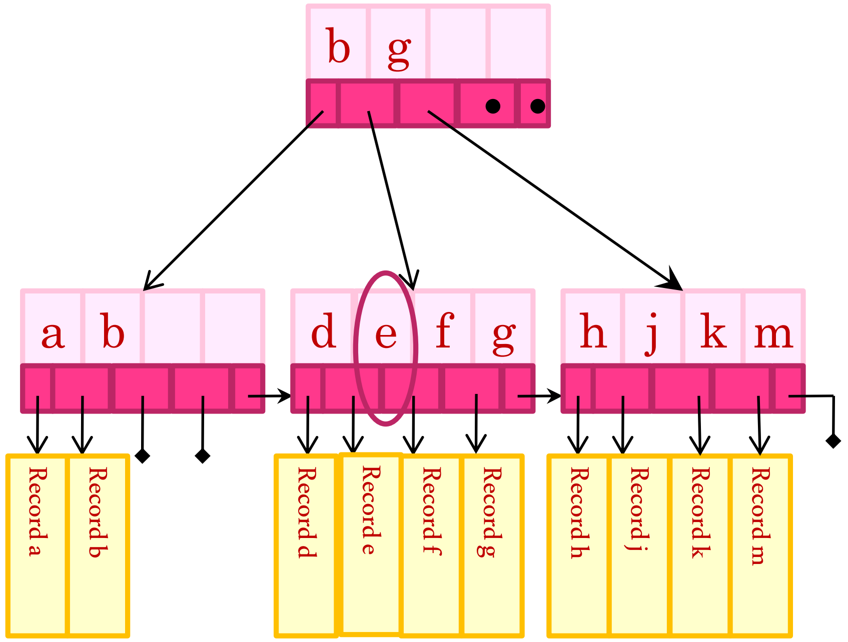
B*-tree after the insertion of j.



INSERTION INTO B*-TREE

Insert: **e**, s, i, r

Insert: **e**

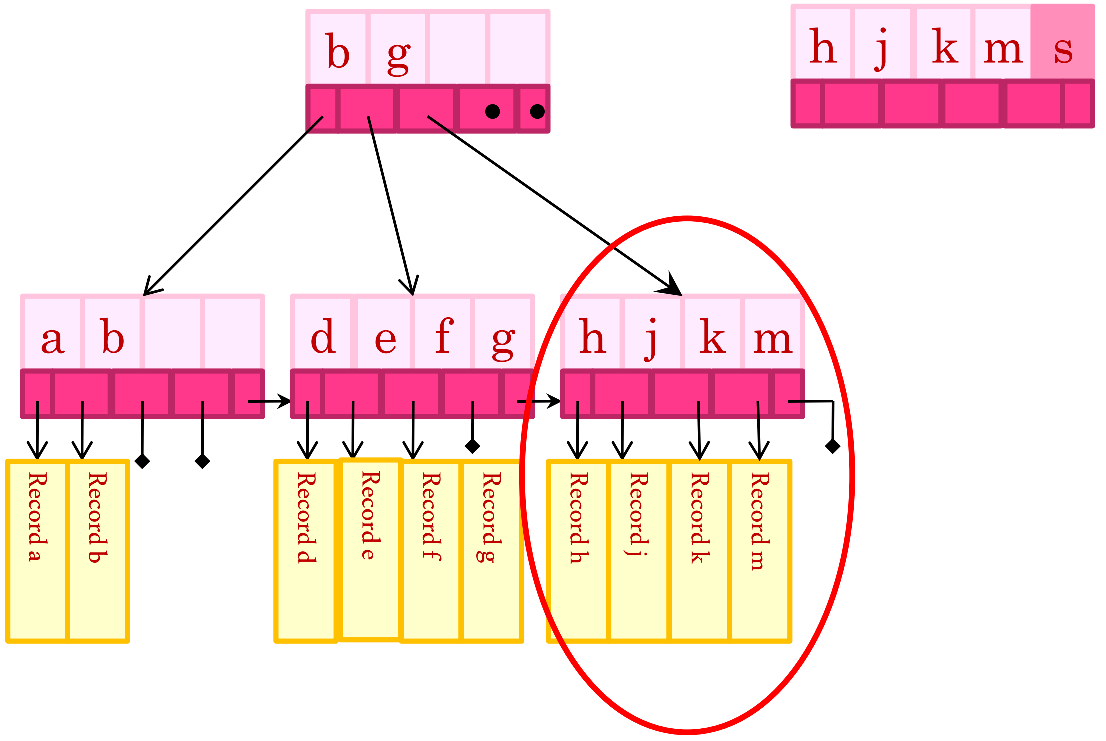


Insertion into B*-tree

Insert: e, s, i, r

Insert s.

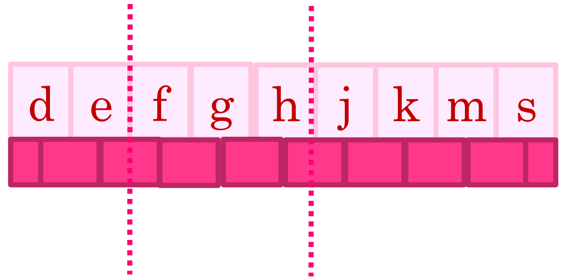
Insert of s causes an overflow of node:



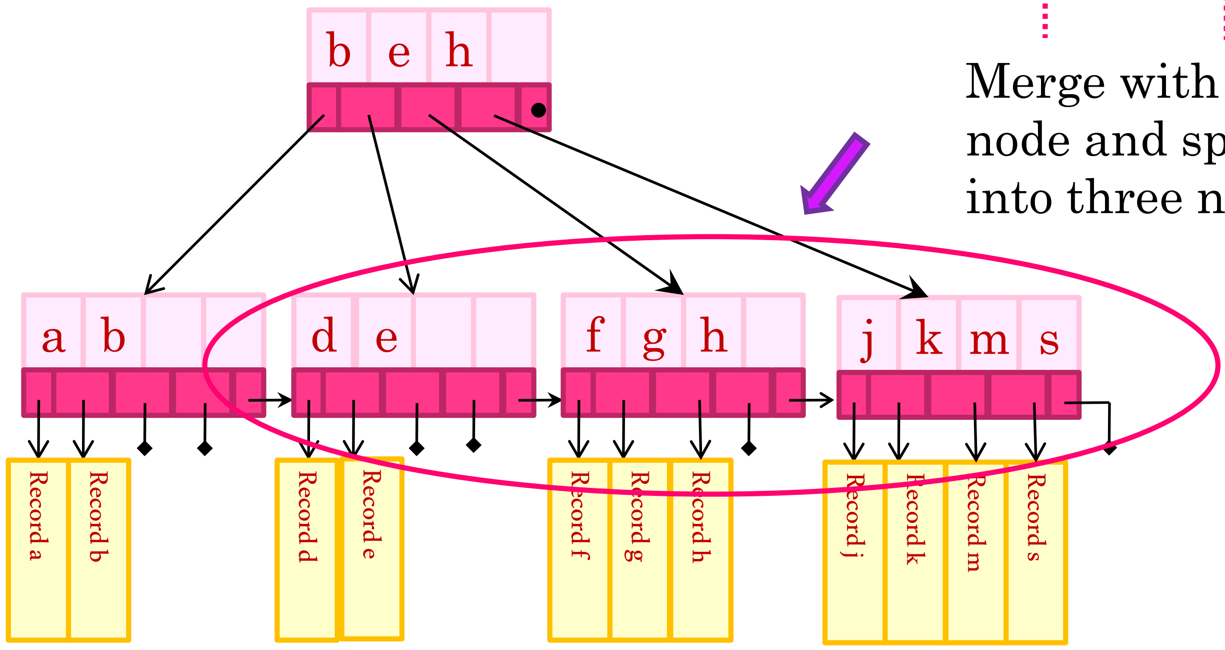
Insertion into B*-tree

Insert: e, **s**, i, r

Insert **s** causes an overflow of node:



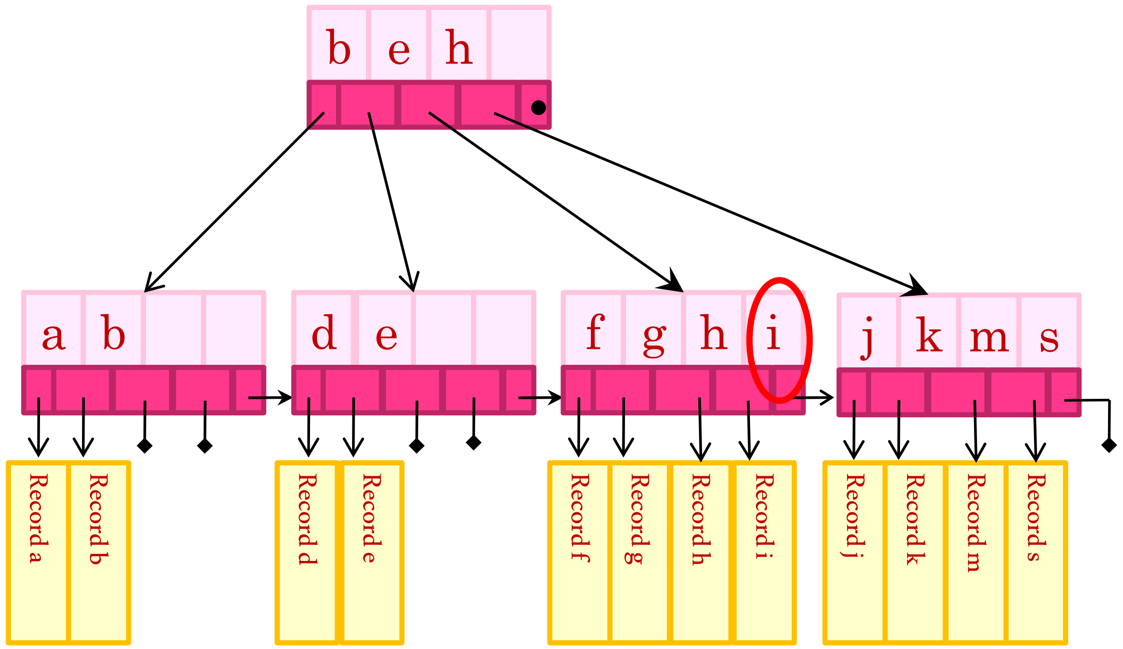
Merge with neighbor node and split nodes into three nodes.



INSERTION INTO B*-TREE

Insert: e, s, **i**, r

Insert **i.**

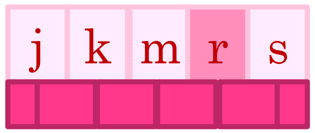


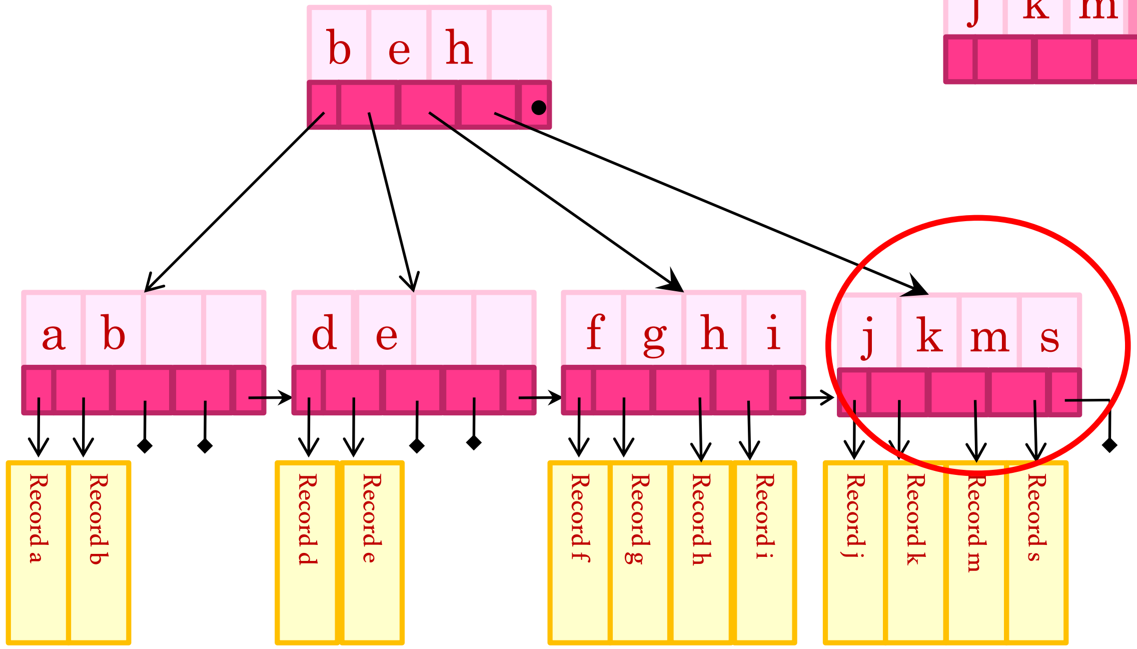
Insertion into B*-tree

Insert: e, s, i, **r**

Insert **r.**

Insert of **r** causes an overflow of node:

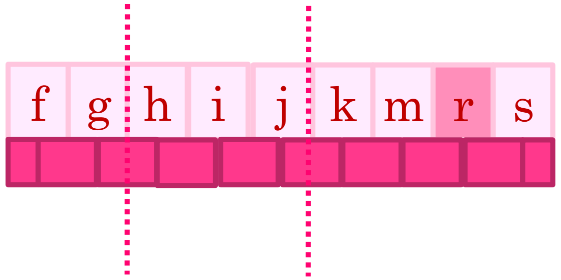




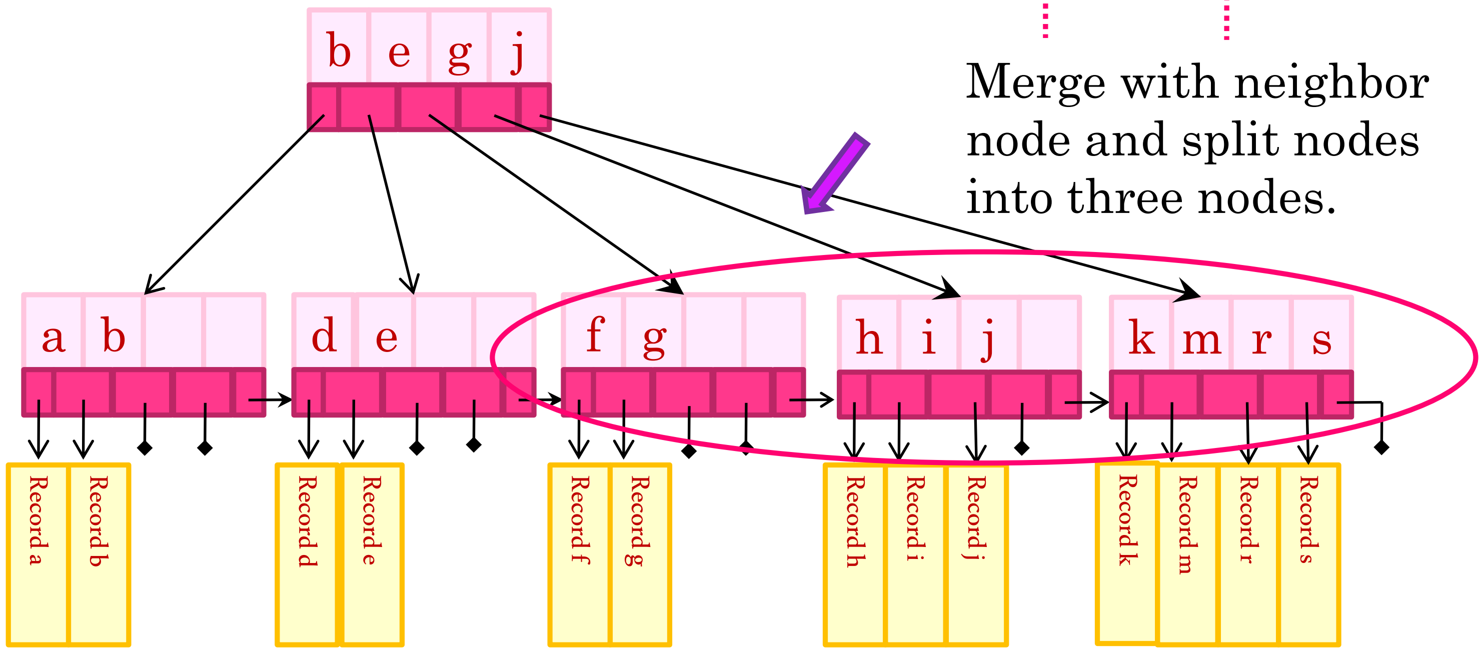
INSERTION INTO B*-TREE

Insert: e, s, i, **r**

Insert **r** causes an overflow of node:

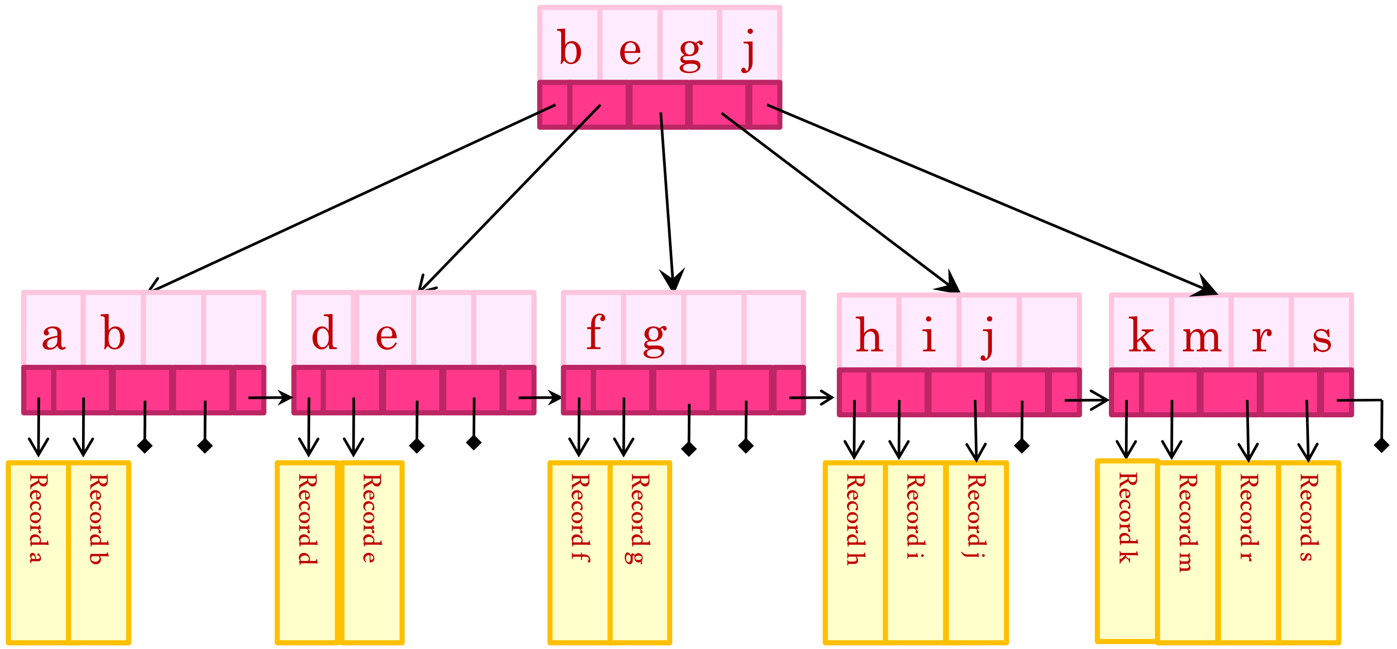


Merge with neighbor node and split nodes into three nodes.



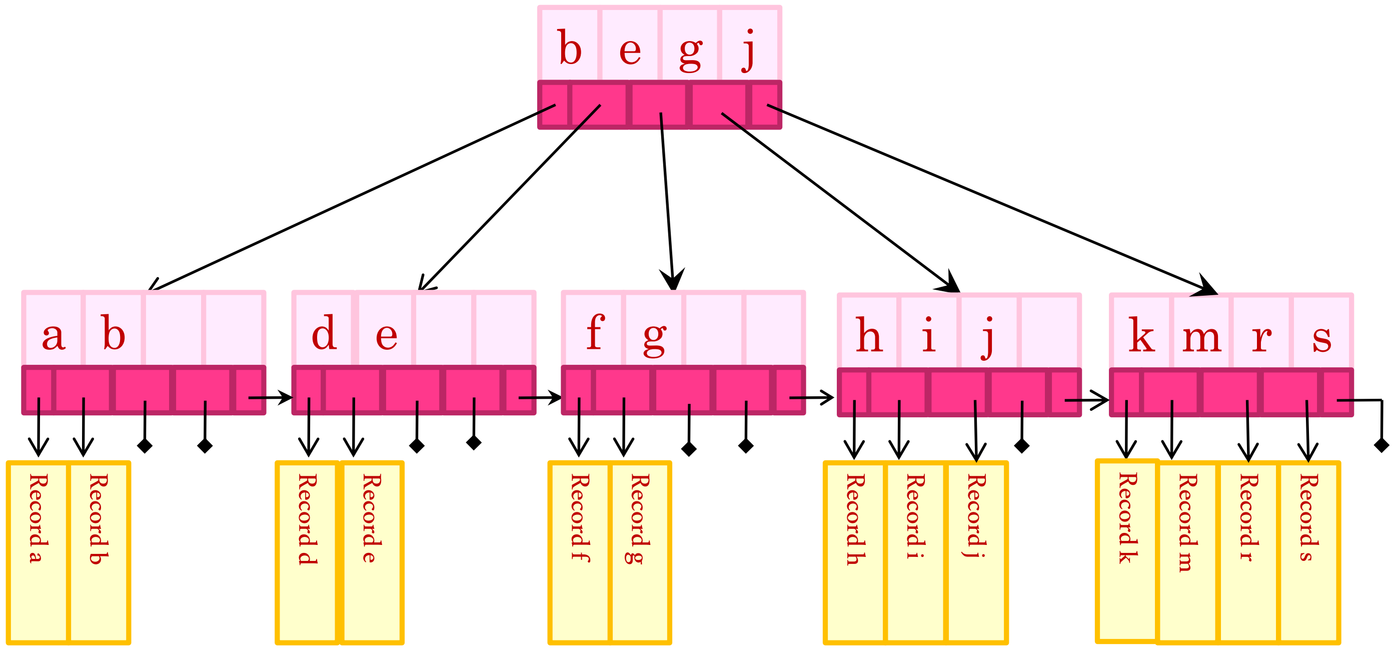
Insertion into B*-tree

B*-tree after the insertion of e, s, i, and r.



INSERTION INTO B*-TREE

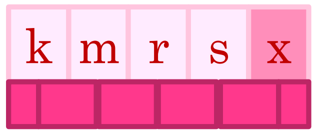
Insert: x



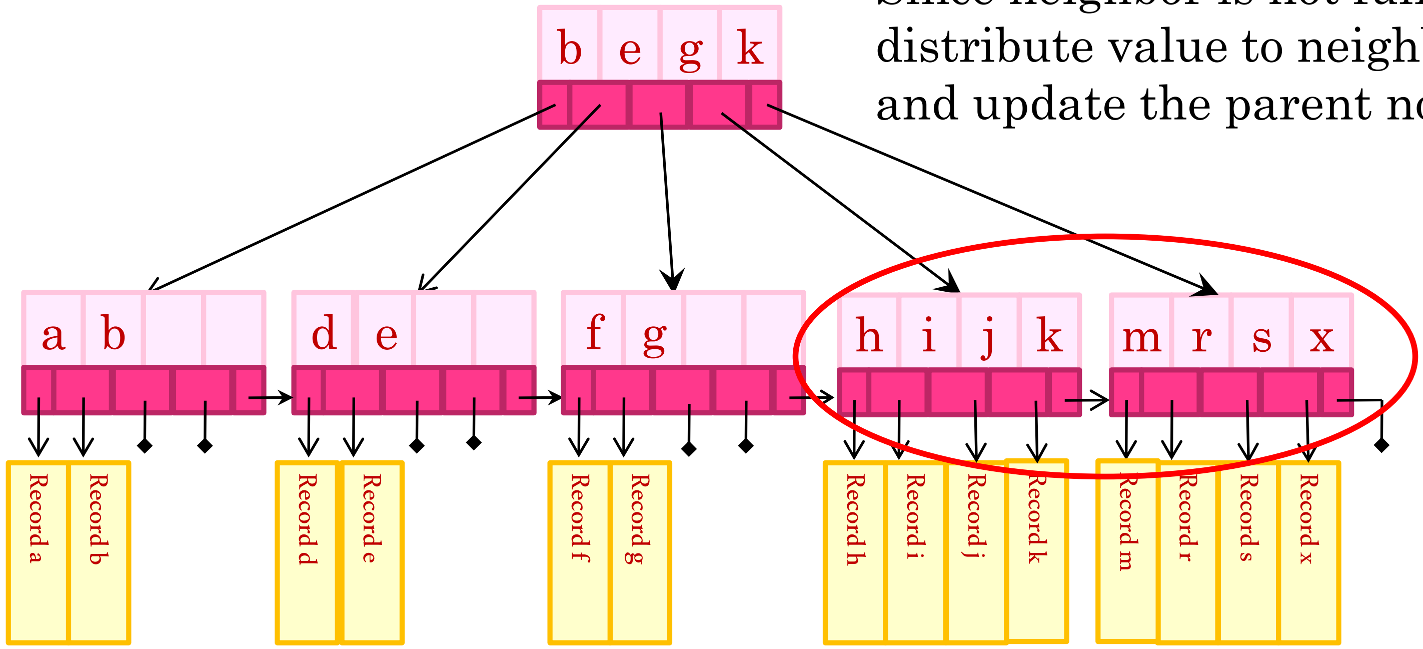
Insertion into B*-tree

Insert: x

Insert of x causes an overflow of node:

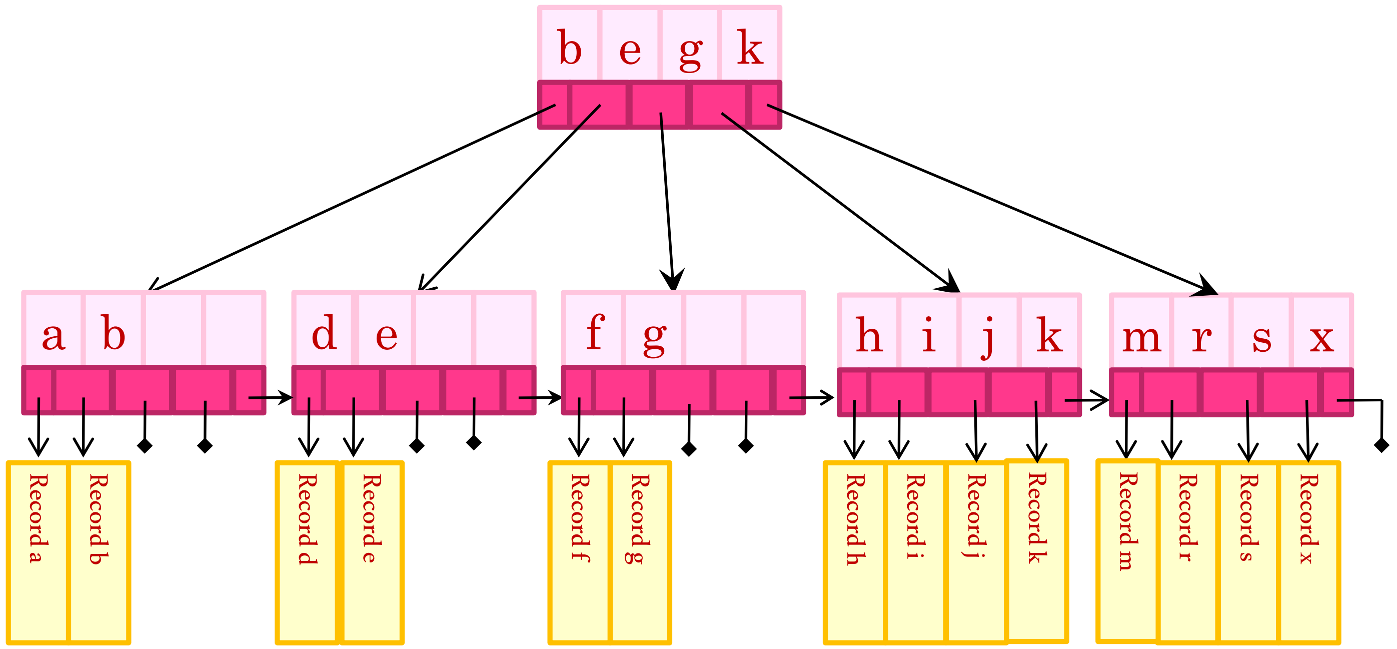


Since neighbor is not full, distribute value to neighbor and update the parent node.



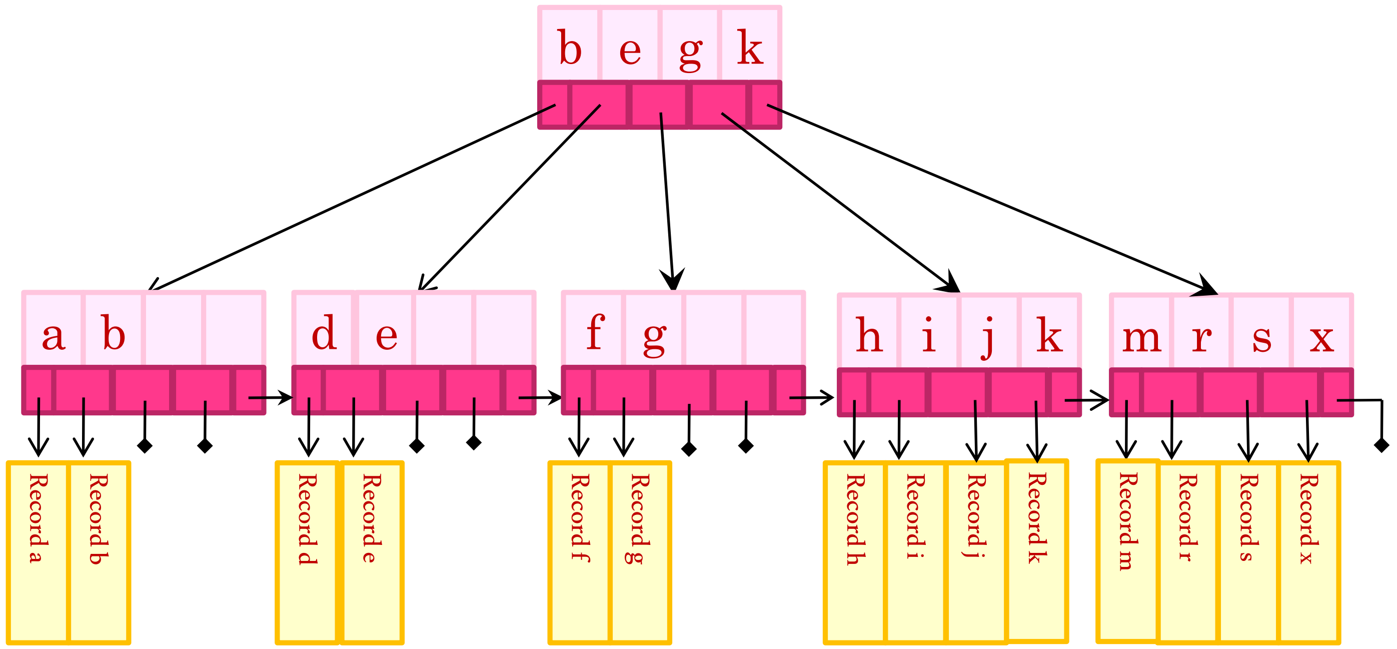
INSERTION INTO B*-TREE

B*-tree after the insertion of x.



Insertion into B*-tree

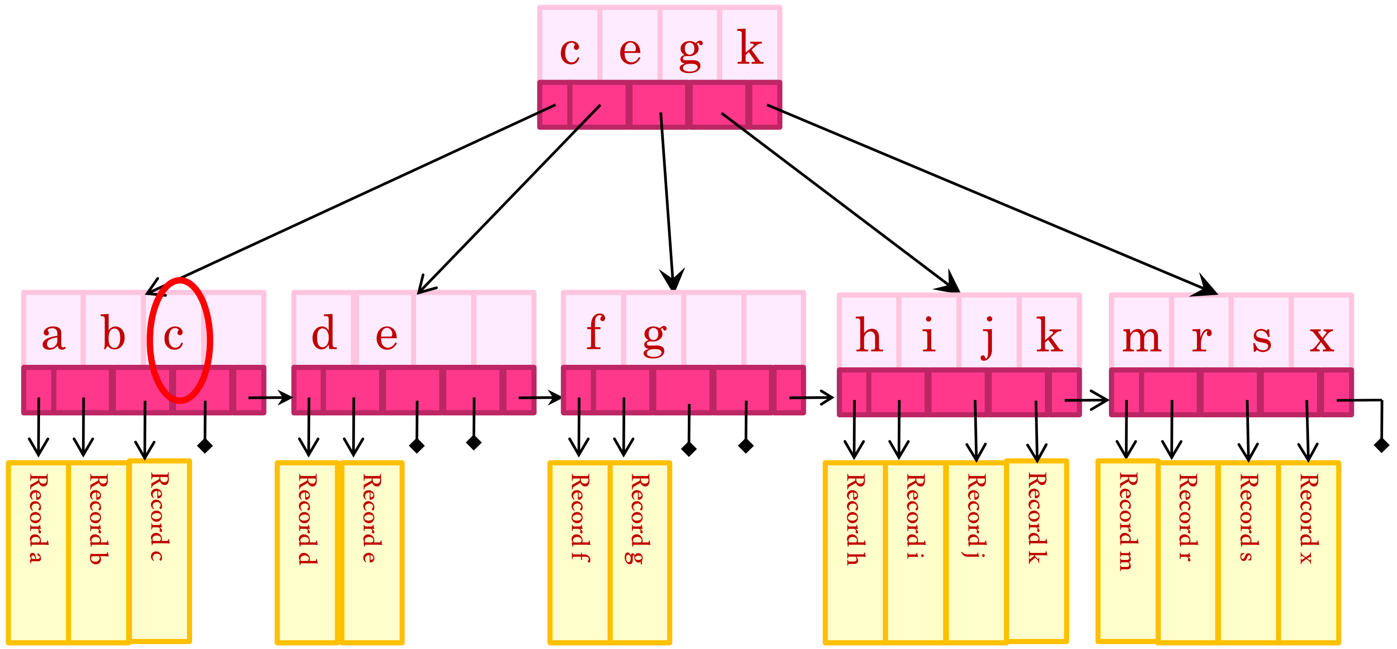
Insert: c, l, n, t, u



INSERTION INTO B*-TREE

Insert: **c**, l, n, t, u

Insert: **c**

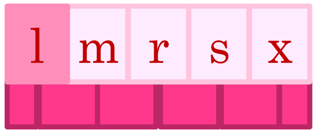


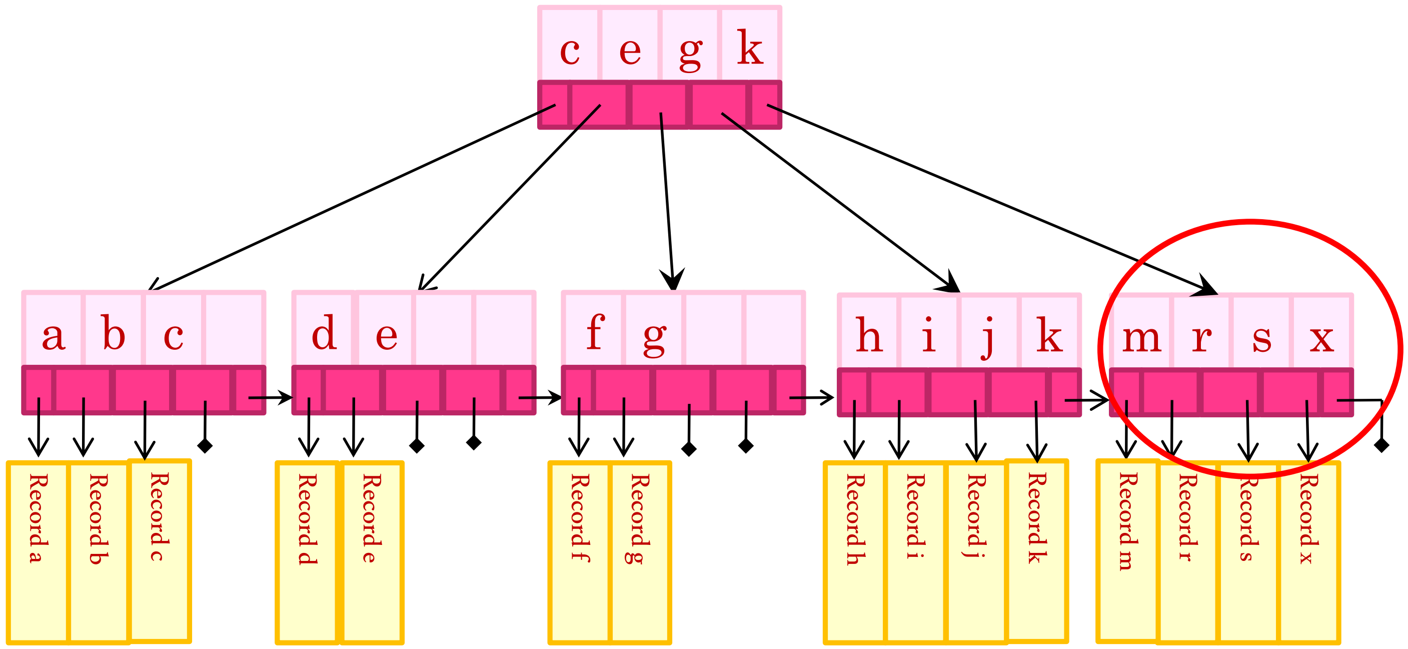
Insertion into B*-tree

Insert: c, **l**, n, t, u

Insert: **l**

Insert of **l** causes an overflow of node:

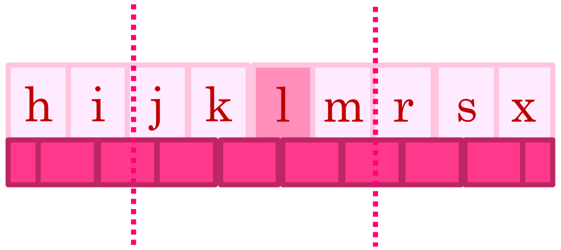




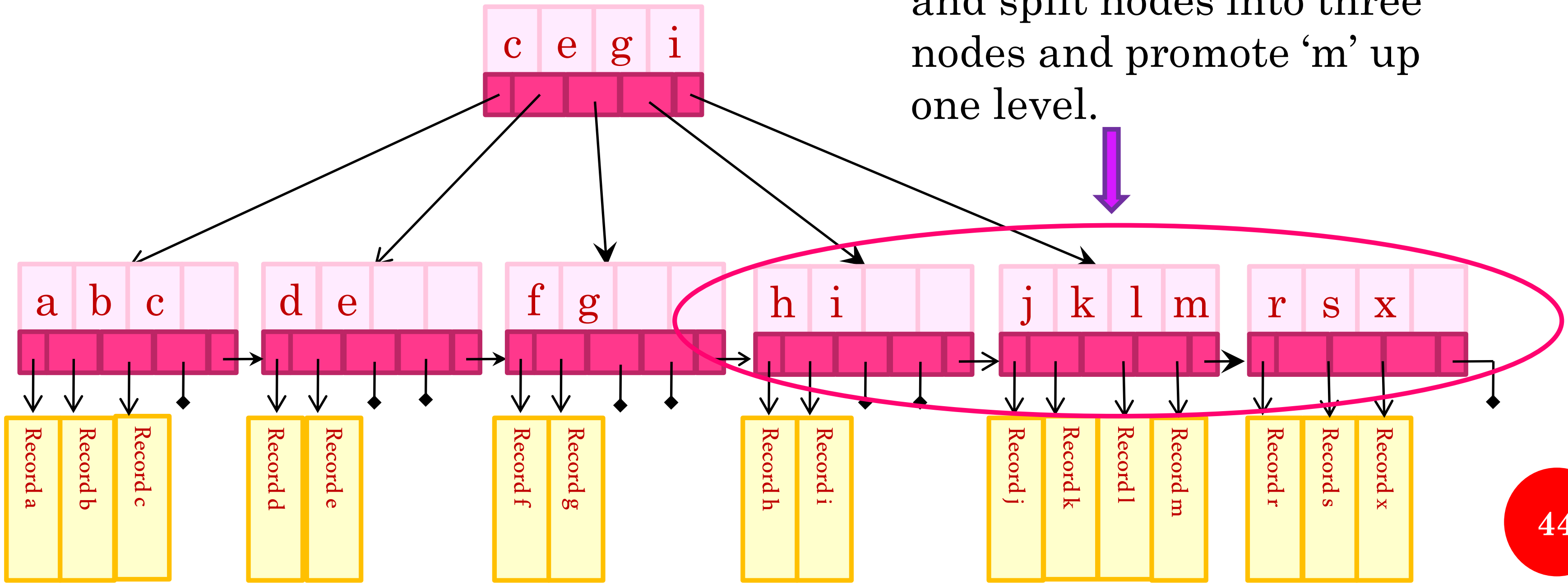
INSERTION INTO B*-TREE

Insert: c, l, n, t, u

Insert l causes an overflow of node:



Merge with neighbor node and split nodes into three nodes and promote 'm' up one level.

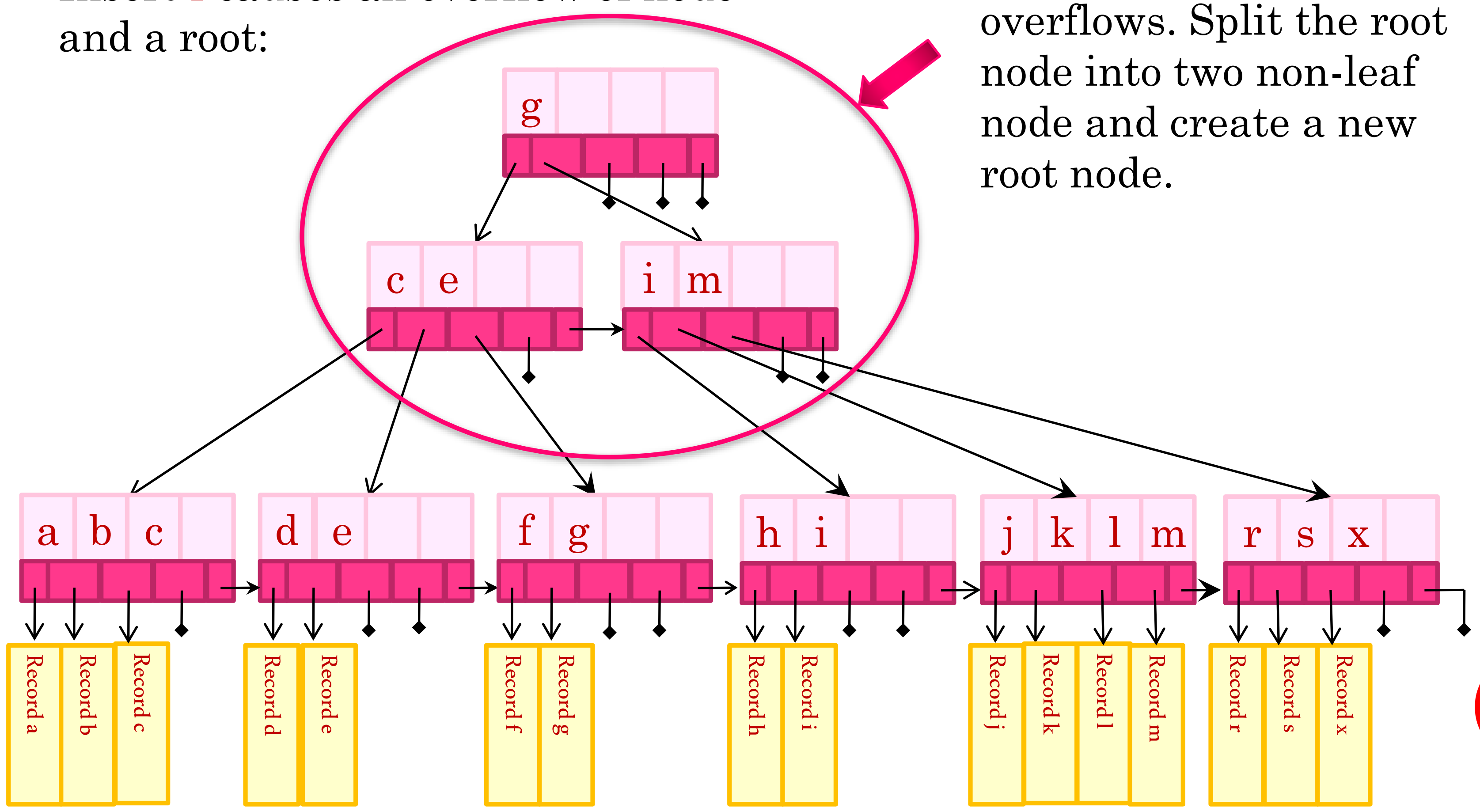


Insertion into B*-tree

Insert: c, **l**, n, t, u

Insert **l** causes an overflow of node and a root:

With 'm' being promoted up one level, the root node overflows. Split the root node into two non-leaf node and create a new root node.



Insertion into B*-tree

- Continue the rest as exercise.

Deletion from B*-Tree

- A deletion is efficient if a node does not become less than half full (known as underflow). In another words, underflow is a situation when the number of key values < n/2-1.

- If a deletion causes a node to become underflow, it must either redistribute its remaining key values to neighbouring nodes or be merged with neighbouring nodes.

Deletion from B*-Tree

Leaf Node

Redistribute key values to sibling

- If the leaf node becomes underflow, redistribute the remaining key values to sibling.
  - Right node less than left node
  - Replace the between-value in parent by their largest key value of the left node.

Merge (contain too few entries)

- Move all key values, pointers to left node or right node, but be consistent, for example, always move to the left node as priority. If left node is full, then move to the right node.
- Remove the between-value in parent

Deletion from B*-Tree

Non-Leaf Node

Redistribute key values to sibling

- Through parent
- Right node not less than left node

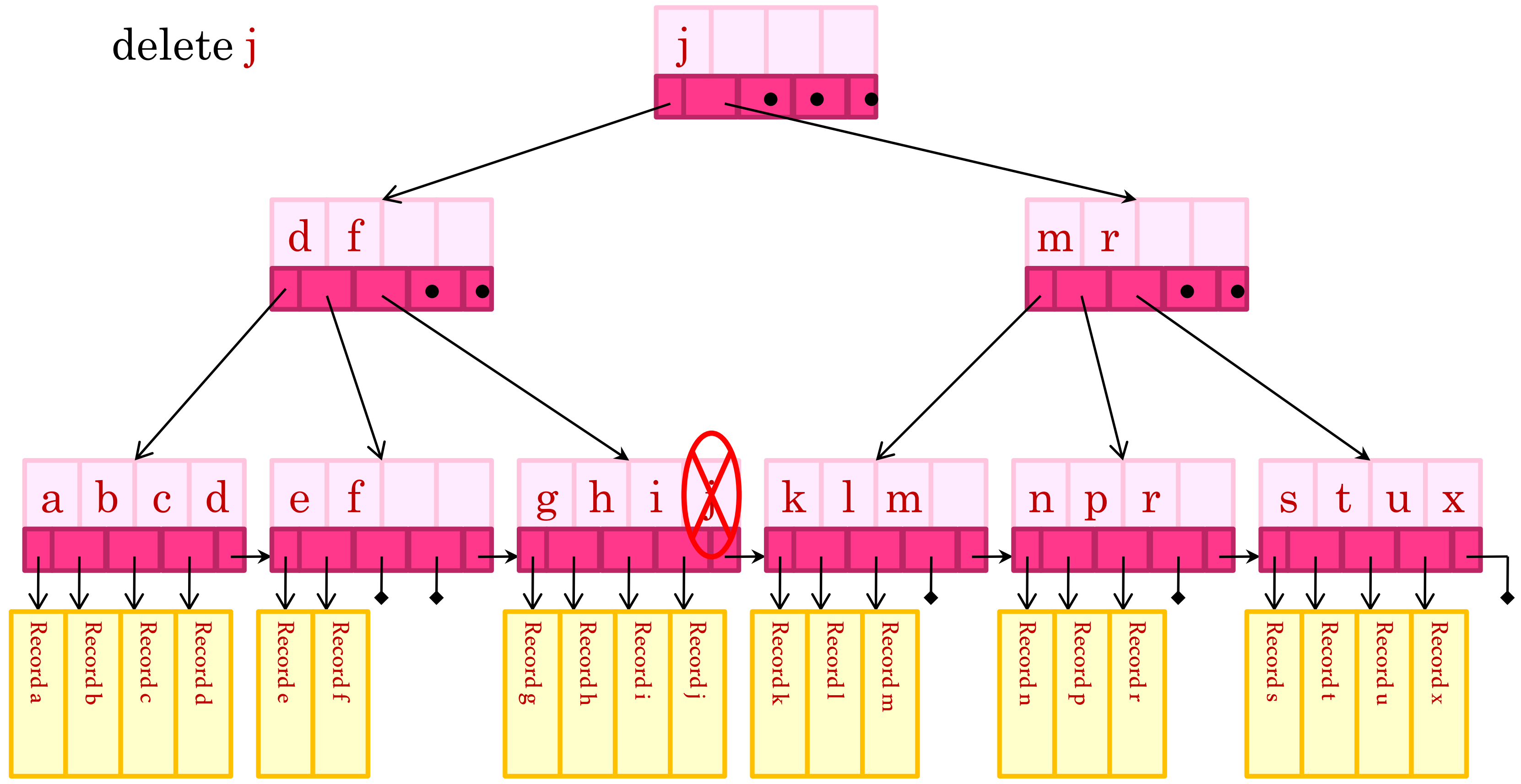
Merge (contain too few entries)

- Bring down parent
- Move all values, pointers to left node
- Delete the right node, and pointers in parent

Deletion from B*-Tree

- For example:

delete j

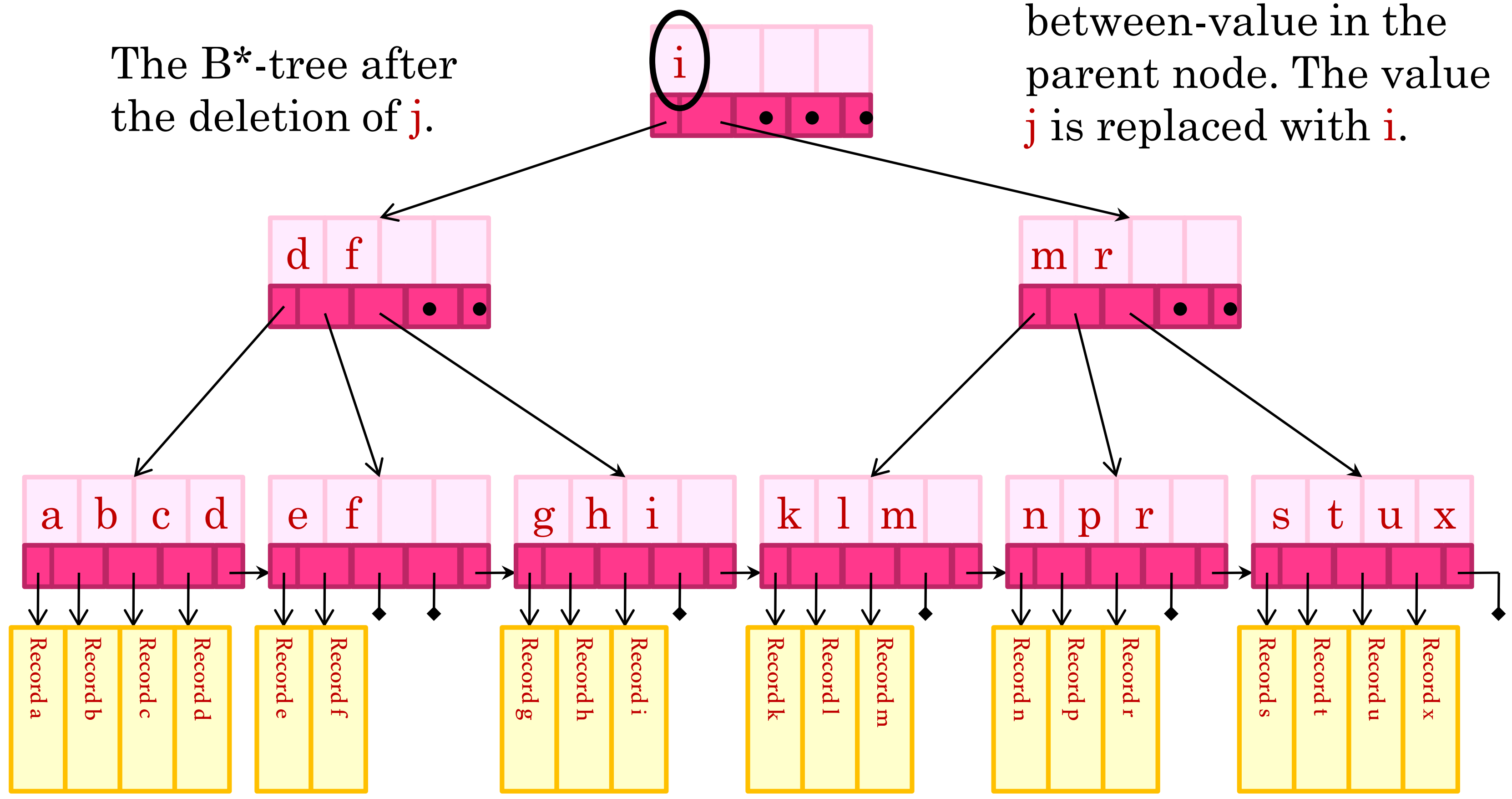


DELETION FROM B*-TREE

- For example:

The B*-tree after the deletion of j.

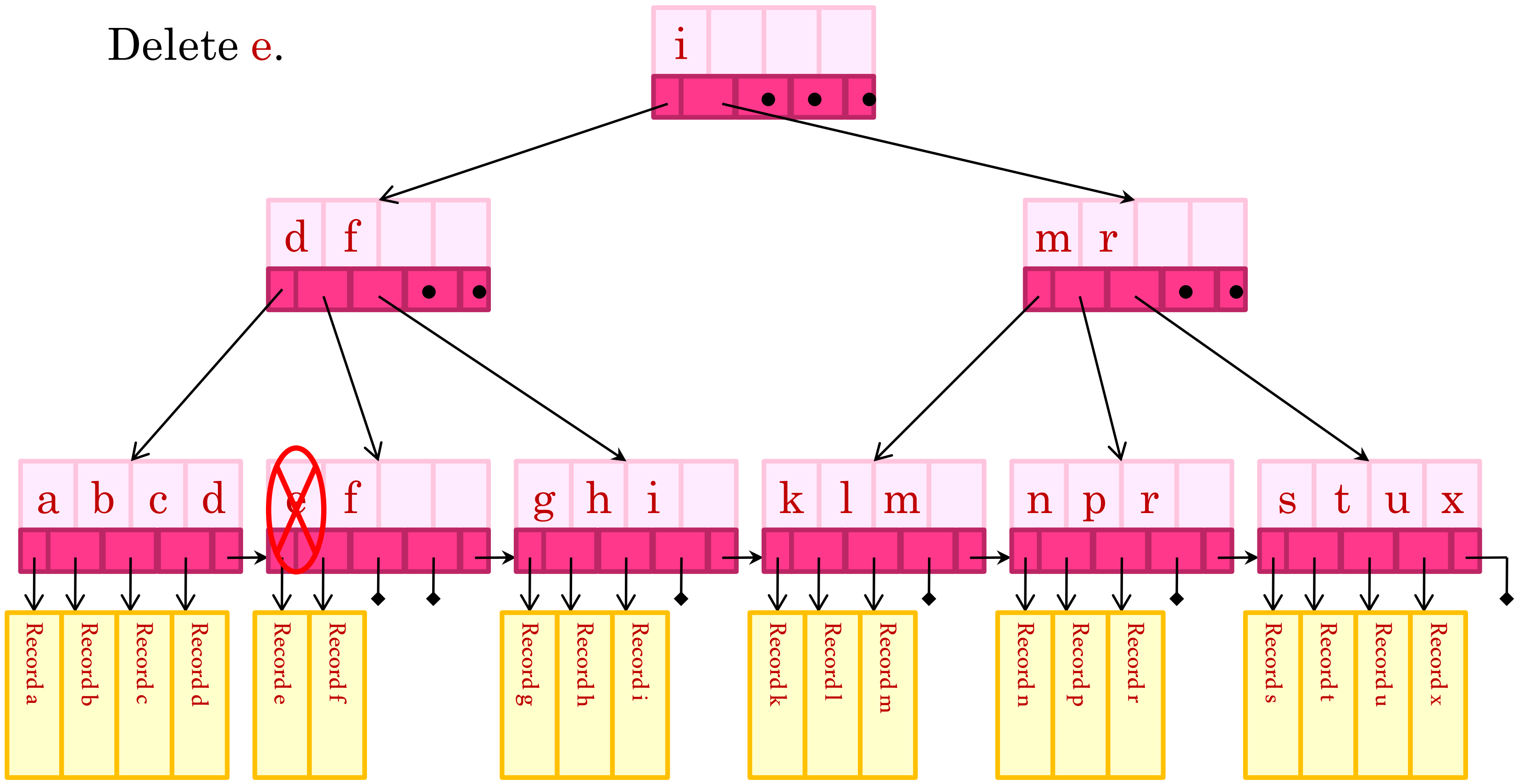
Take note of the between-value in the parent node. The value j is replaced with i.



Deletion from B*-Tree

- For example:

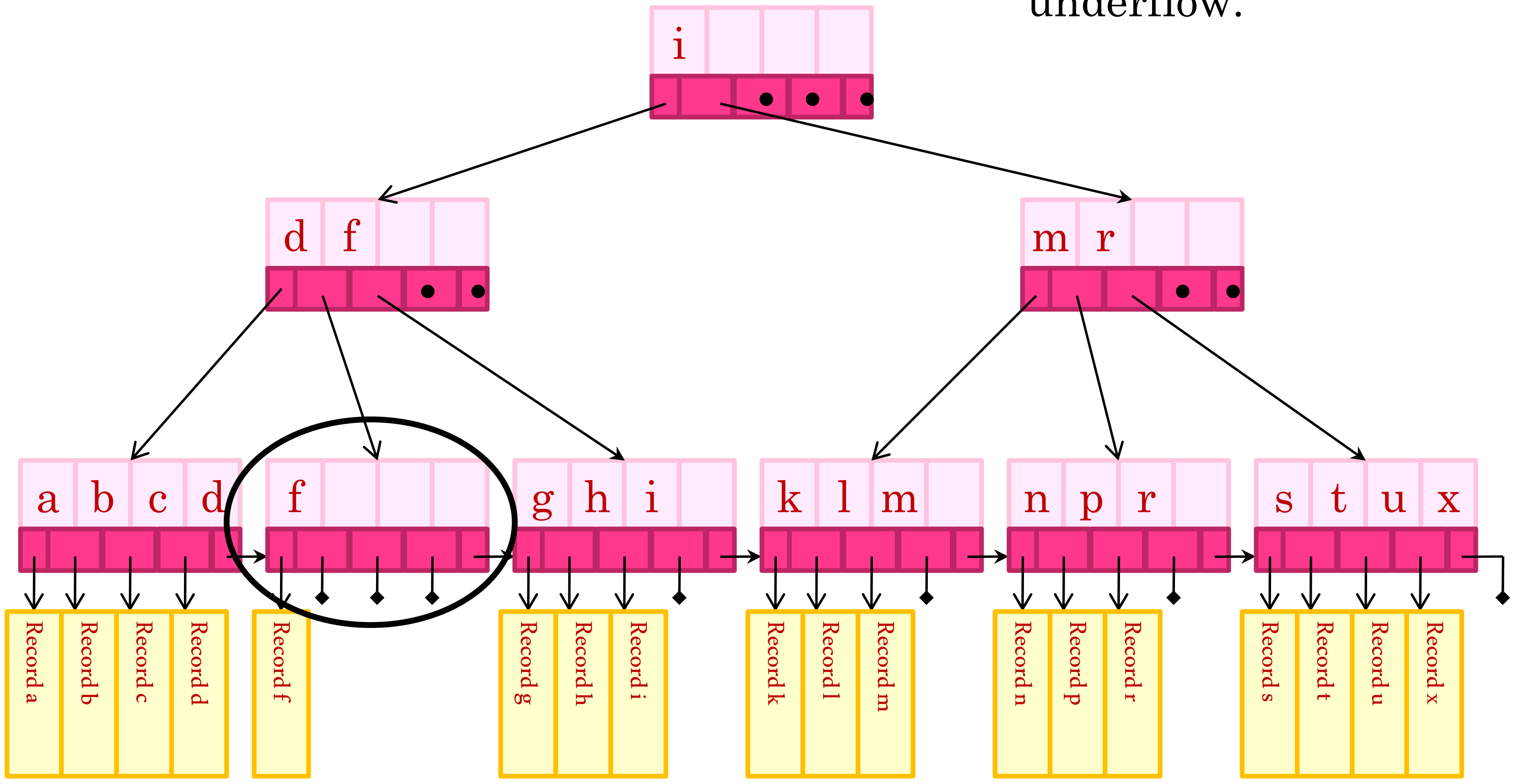
Delete e.



DELETION FROM B*-TREE

- For example:

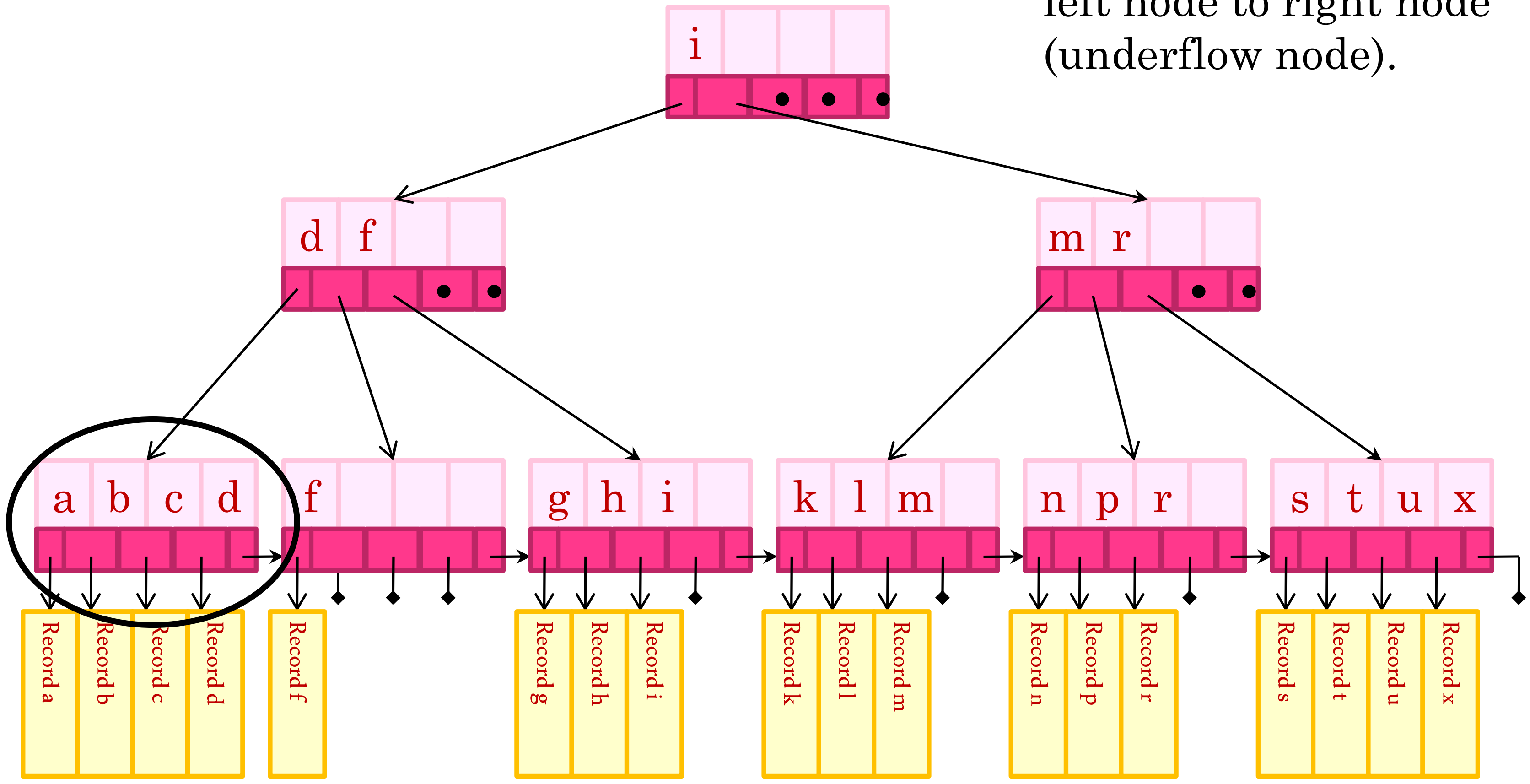
Leaf node becomes underflow.



Deletion from B*-Tree

- For example:

Redistribute d from the left node to right node (underflow node).

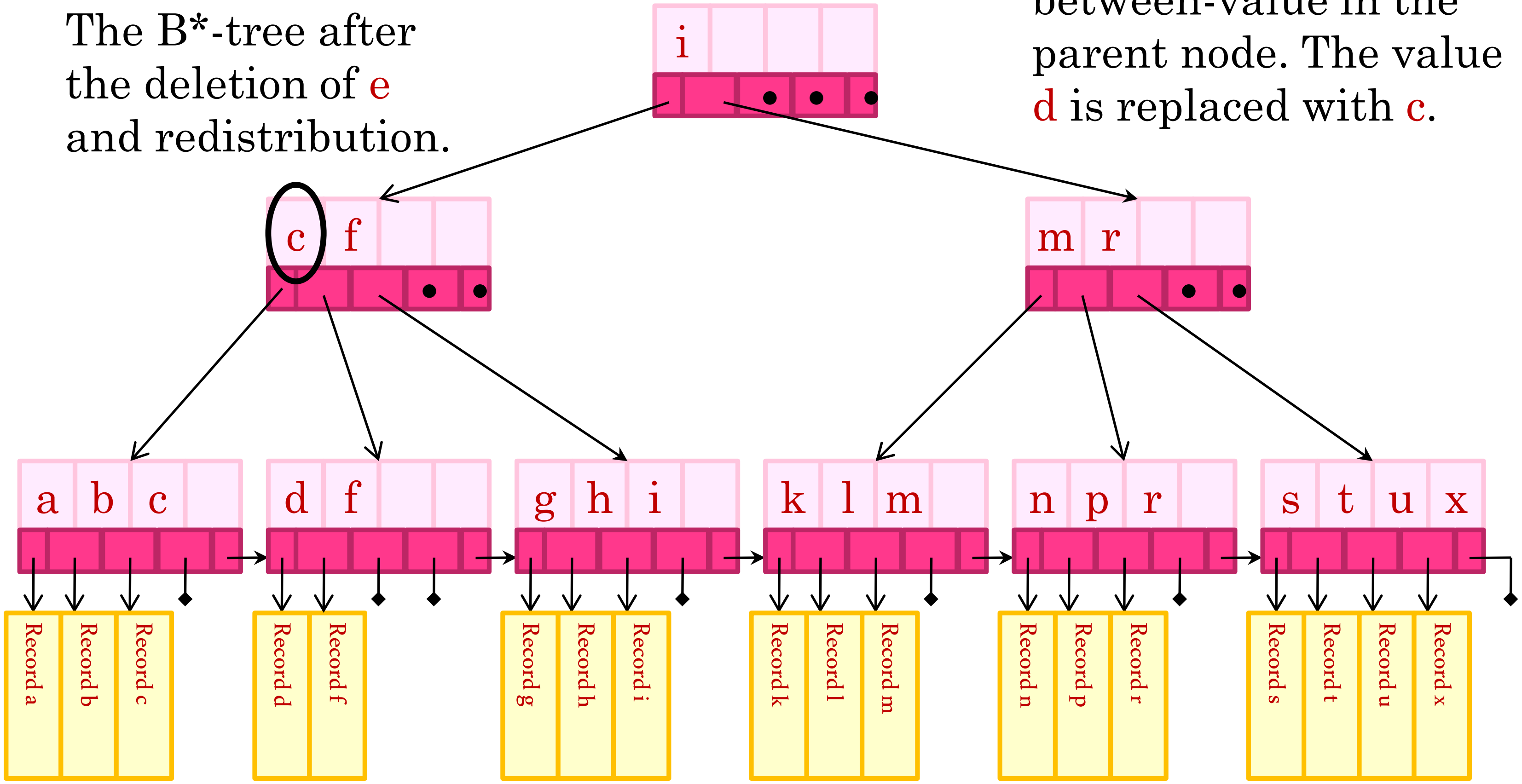


Deletion from B*-Tree

- For example:

The B*-tree after the deletion of e and redistribution.

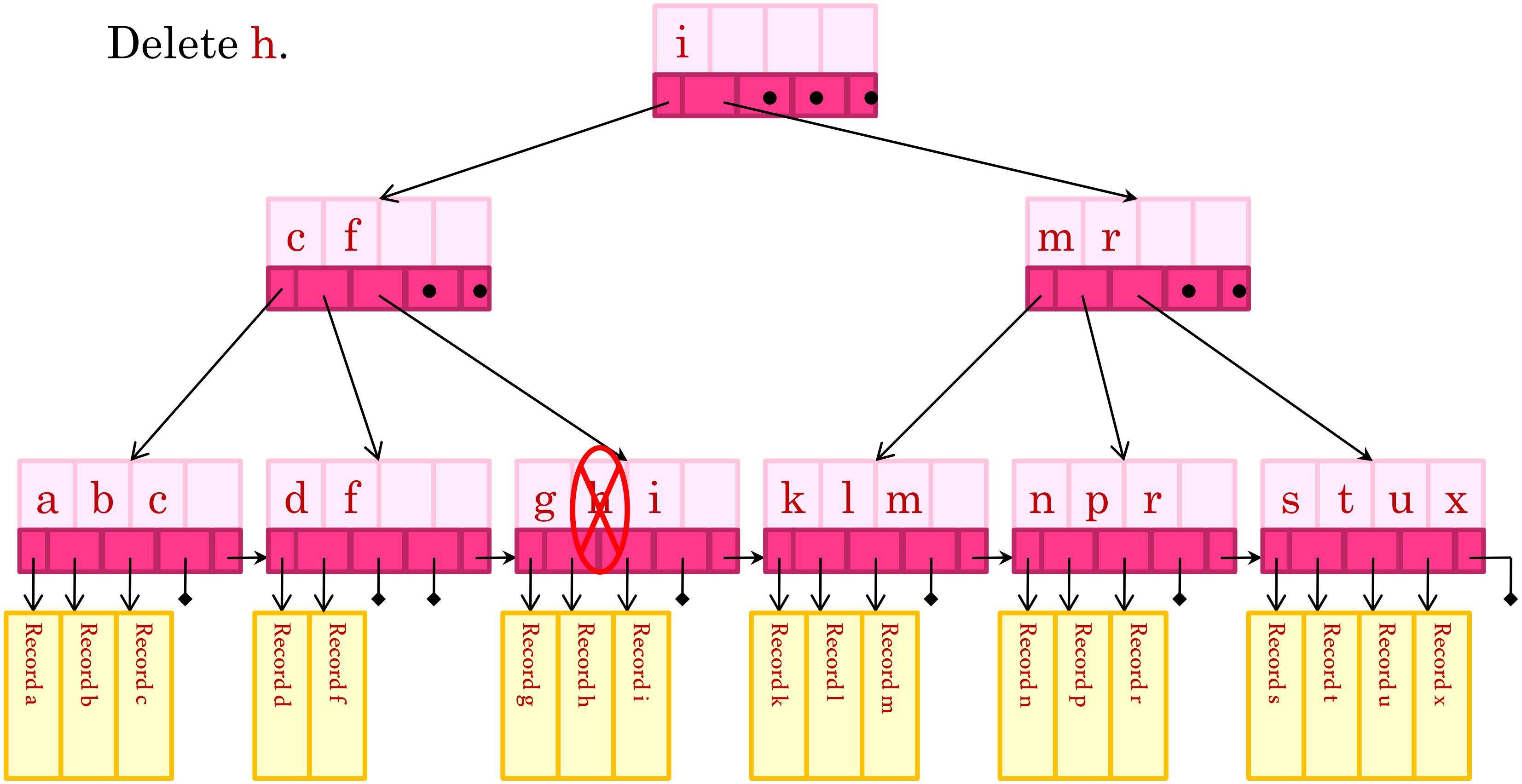
Take note of the between-value in the parent node. The value d is replaced with c.



Deletion from B*-Tree

- For example:

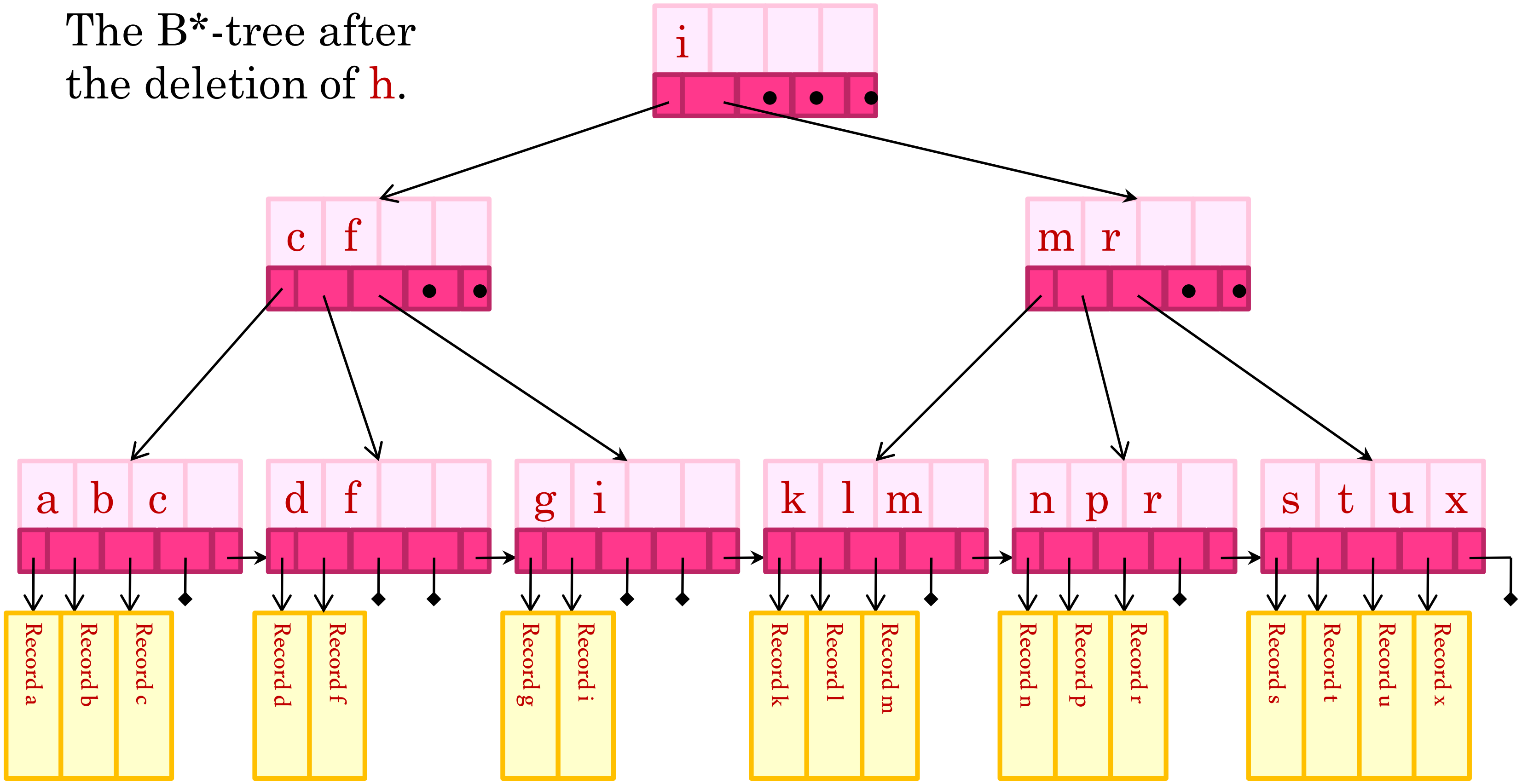
Delete h.



Deletion from B*-Tree

- For example:

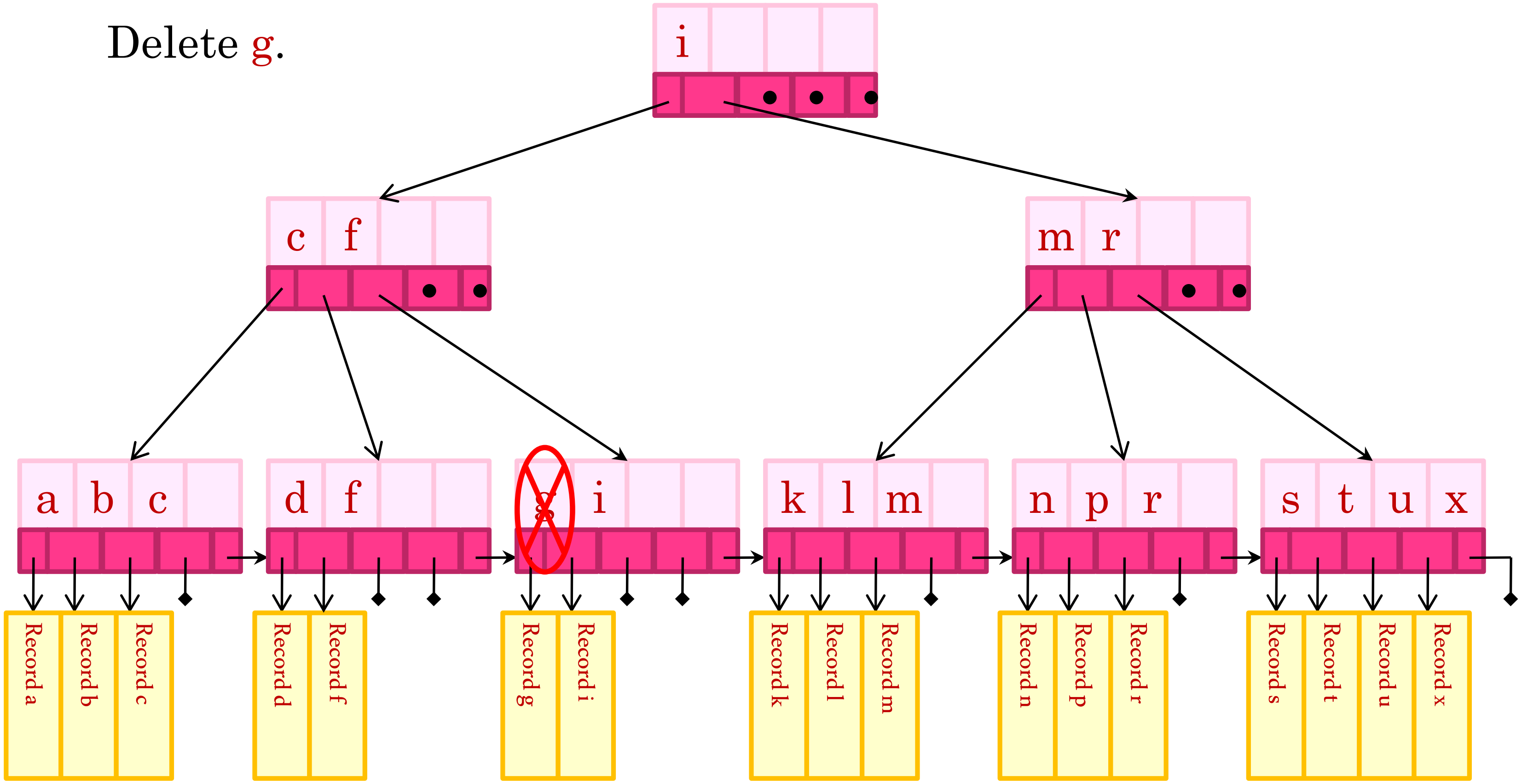
The B*-tree after the deletion of h.



Deletion from B*-Tree

- For example:

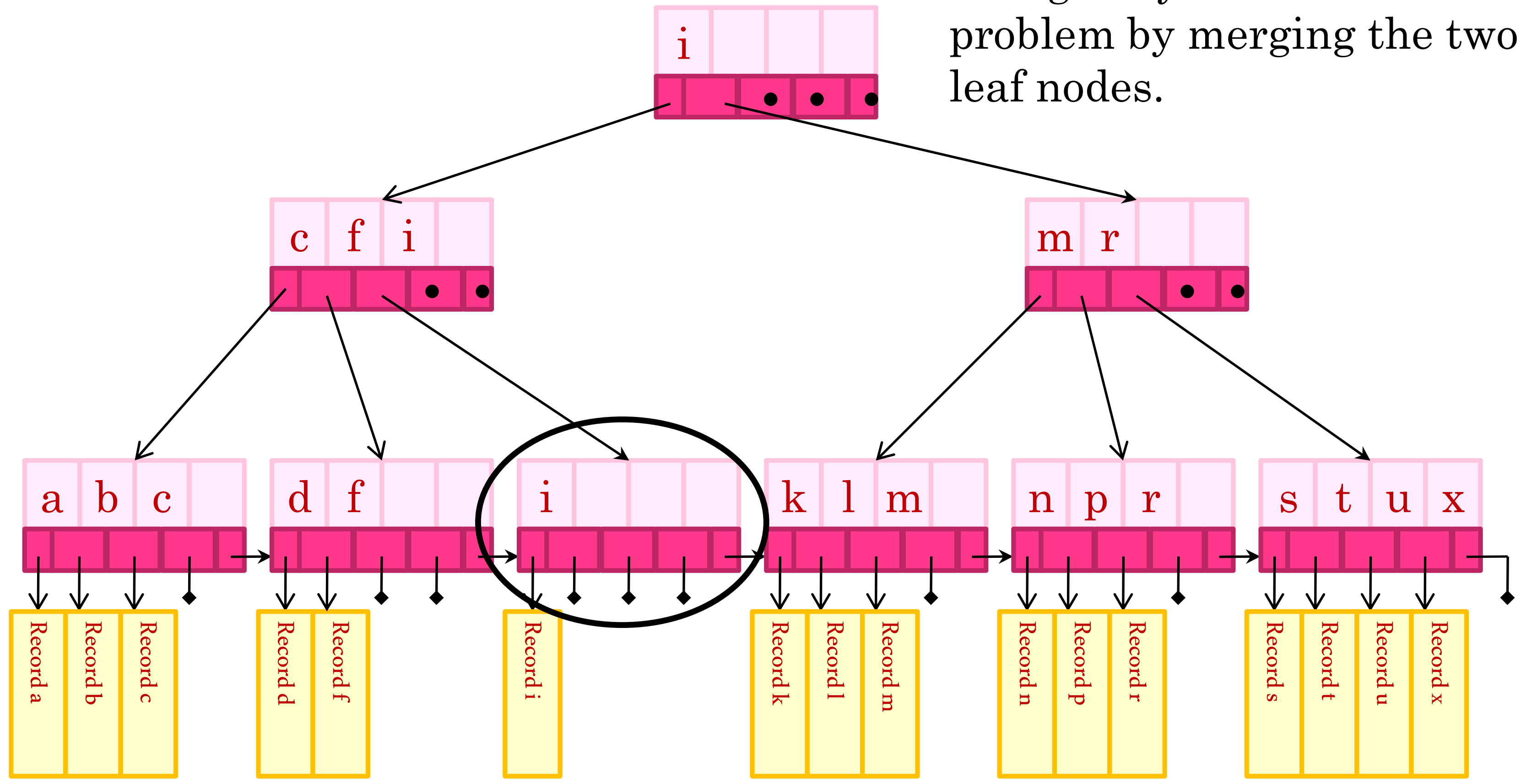
Delete g.



Deletion from B*-Tree

- For example:

Leaf node becomes underflow and cannot redistribute from the left node as well because the left node does not have enough key values. Solve the problem by merging the two leaf nodes.

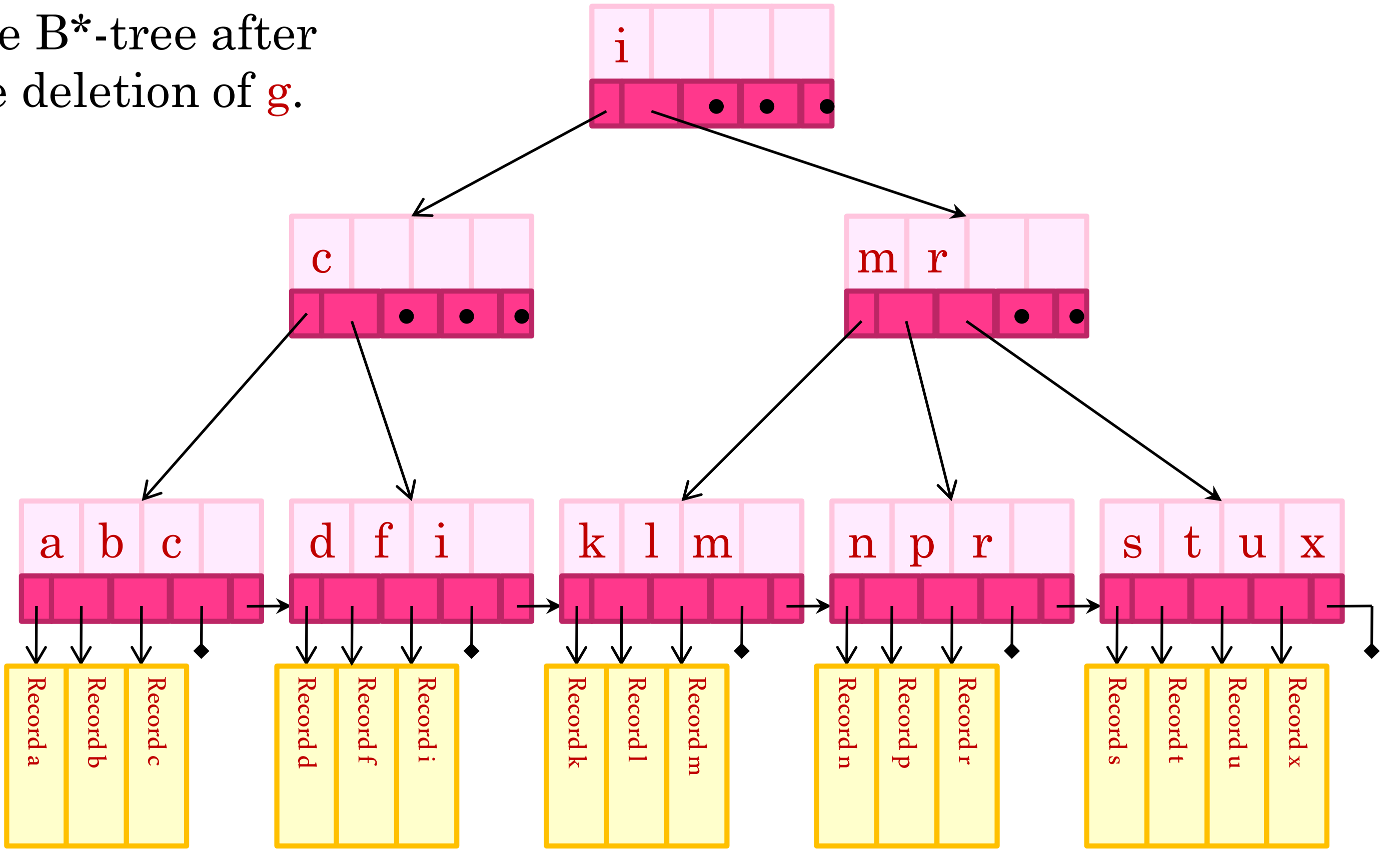


DELETION FROM B*-TREE

- For example:

a ~~g~~ f b k d ~~h~~ m ~~j~~ e s i r x c l n t u p

The B*-tree after the deletion of g.



i
c
m r
a b c
d f i
k l m
n p r
s t u x
Record a
Record b
Record c
Record d
Record f
Record i
Record k
Record l
Record m
Record n
Record p
Record r
Record s
Record t
Record u
Record x