SIM GLOBAL EDUCATION

UNIVERSITY OF WOLLONGONG AUSTRALIA

# CSCI203 - ALGORITHMS AND DATA STRUCTURES

## Tree (2)

Sionggo Japit
sjapit@uow.edu.au

2 January 2023

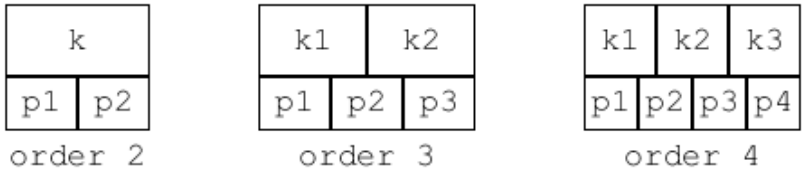
# Outline

- M-way Tree
- 2-4 Trees
- B* Trees
- Treaps

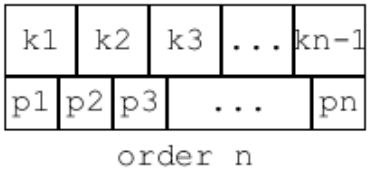
# m-way Tree

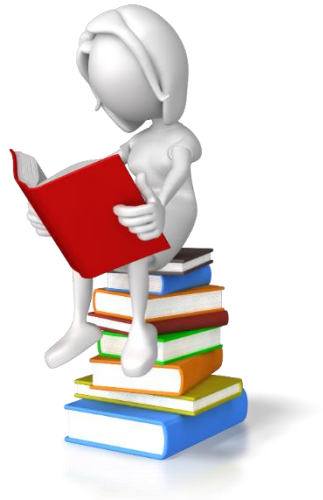
# m-Way (multi-way) Tree

- An $m-way$ tree is a search tree in which each node can have from $0$ to $m$ sub-trees, where $m$ is defined as the order of the tree.




Order $n$ tree, has up to $n-1$ keys in each node.



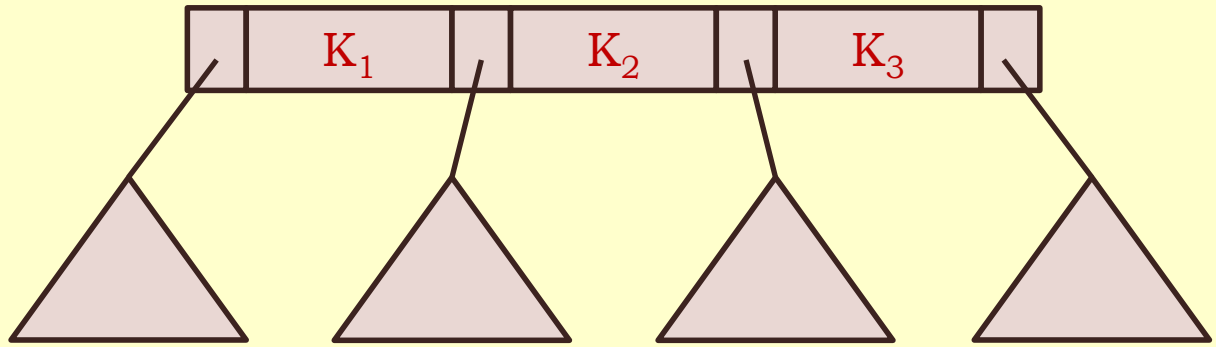
# m-Way Tree

Properties:

1. Each node has $0$ to $m$ sub-trees
2. A node with $k < m$ sub-trees contains $k$ sub-tree pointers, some of which may be $null$, and $k - 1$ data entries (keys)
3. The key values in the first sub-tree are all less than the key value in the first entry: the key values in the other sub-trees are all greater than or equal to the key value in their parent entry
4. The keys of the data entries are ordered $key_1 \leq key_2 \leq \cdots \leq key_k$
5. All sub-trees are themselves multiway trees

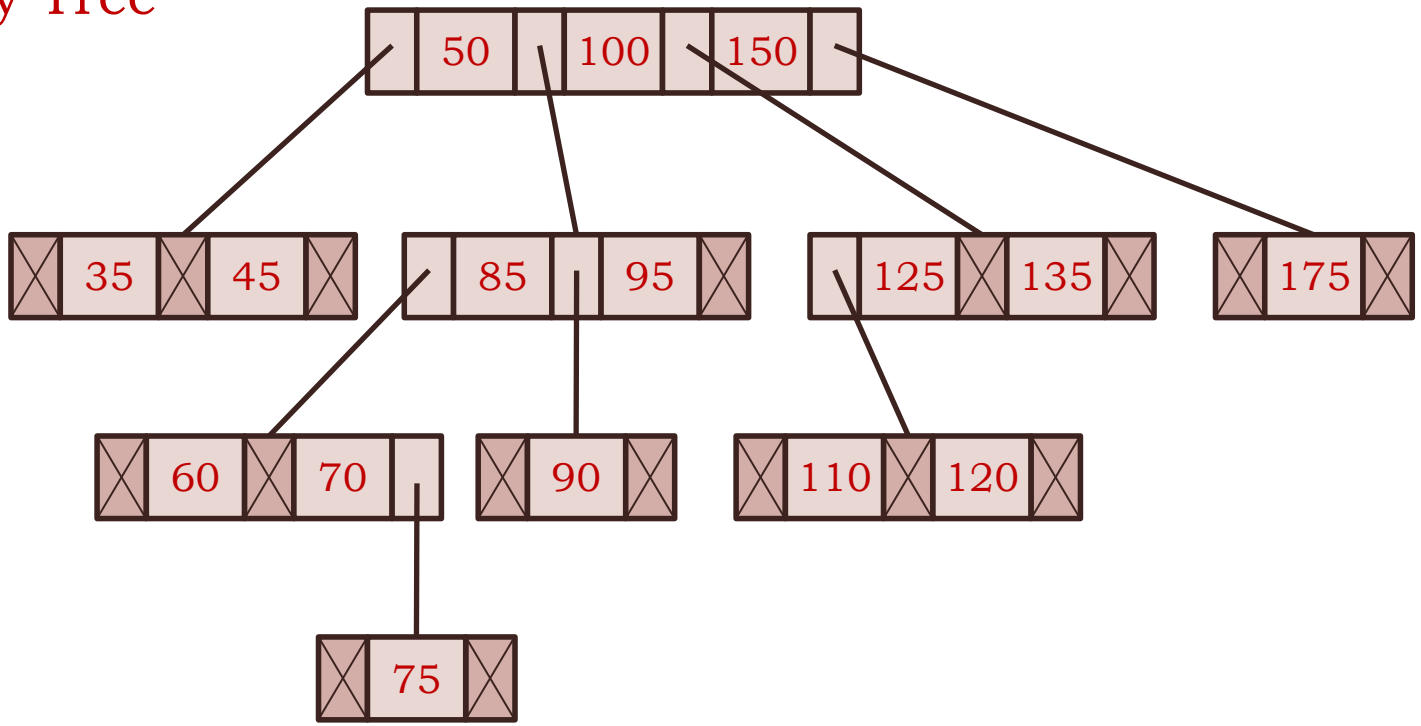
# m-Way Tree

m-way tree of order 4



# m-Way Tree

Four-way Tree



# m-Way Tree

- $m-way$ search tree has the same structure as the binary search tree:
  - sub-trees to the left of an entry contain data with keys that are less than the key of the entry, and
  - sub-trees to the right of an entry contain data with keys that are greater than or equal to the entry's key


Balance Tree
(B-Tree)

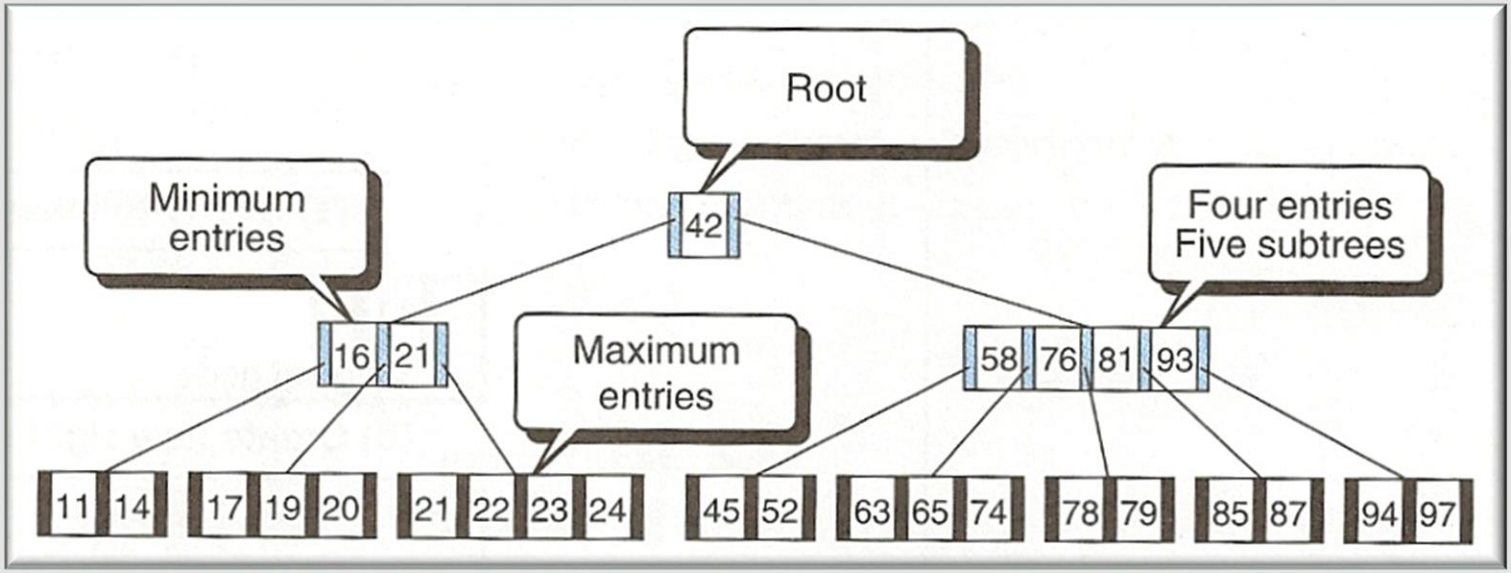
# B-Tree

- A B-tree is an $m-way$ search tree with the following additional properties:
  1. The root is either a leaf or it has $2..m$ sub-trees.
  2. All internal nodes have at least $\lceil m/2 \rceil$ non-null sub-trees and at most $m$ non-null sub-trees.
  3. All leaf nodes are at the same level, that is, the tree is perfectly balanced.
  4. A leaf node has at least $\lceil m/2 \rceil - 1$ and at most $m-1$ entries (keys).

# B-Tree

| Order | Number of sub-trees | |
|---|---|---|
| | Minimum | Maximum |
| 3 | 2 | 3 |
| 4 | 2 | 4 |
| 5 | 3 | 5 |
| 6 | 3 | 6 |
| … | … | … |
| $m$ | $\lceil m/2 \rceil$ | $m$ |

# B-Tree


Root
Minimum entries
Four entries Five subtrees
Maximum entries
42
16 21
58 76 81 93
11 14
17 19 20
21 22 23 24
45 52
63 65 74
78 79
85 87
94 97

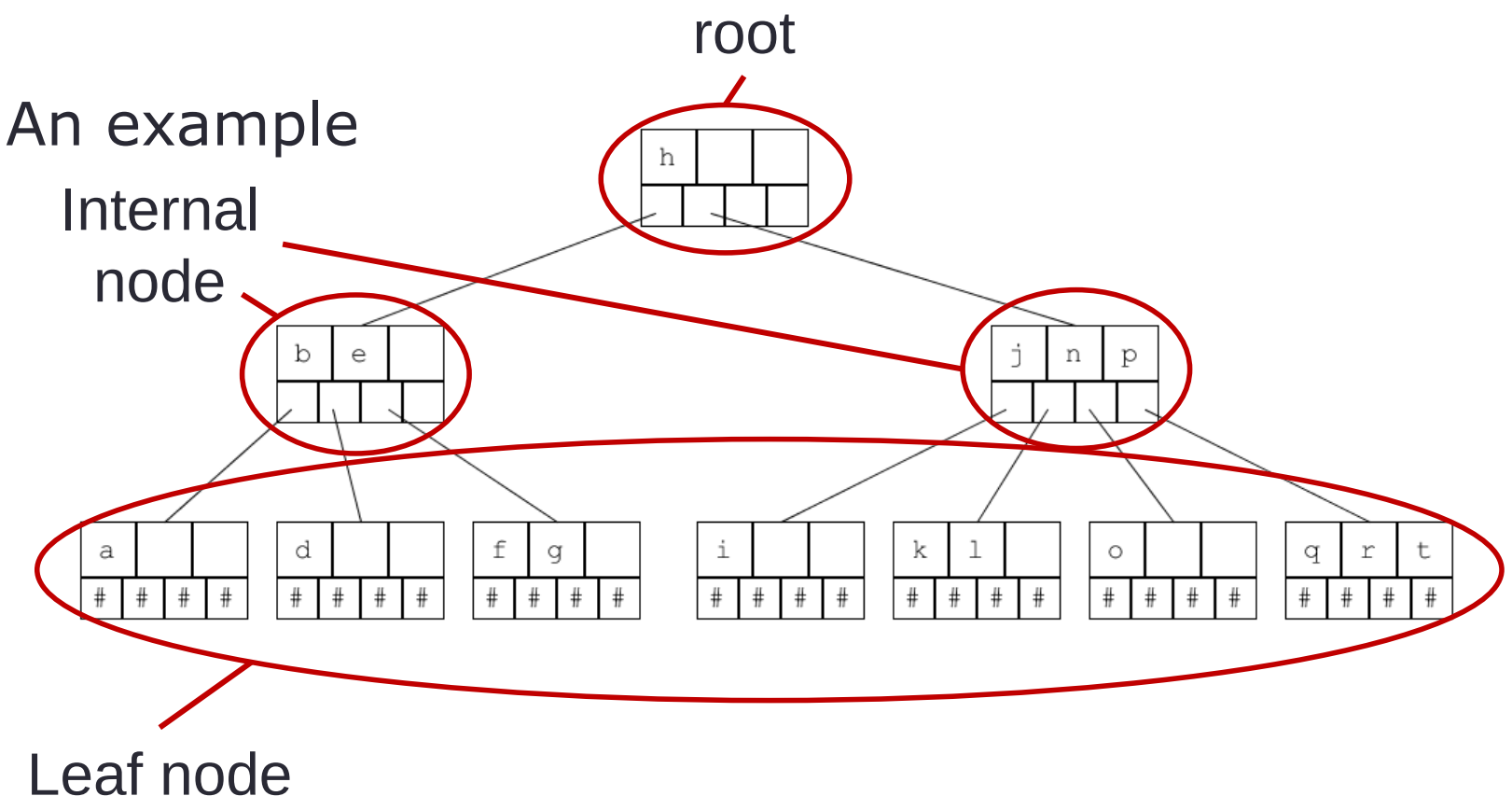

2-4 Tree

# 2-4 Tree

Properties

- Every internal node (except possibly the root) has between two and four children.
- All leaves of the tree are at the same depth.
- Items are stored in the leaves of the tree.
- The items in the leaves appear in order from left to right.

# 2-4 Tree


root
An example
Internal node
h
b e
j n p
a
d
f g
i
k l
o
q r t
Leaf node

# 2-4 Tree

- 2-4 Trees: Searching
  - Similar to searching a binary tree
  - At each level,
    - Go to the left child with search key ≤ node (item) value.
    - Go to the right child with search key > node (item) value.

NOTE: The operations ‘go to left child with search key < node’ and ‘go to right child with search key ≥ node’ are done according to how the implementation (definition) of the 2-4 tree structure.

# 2-4 Tree

## 2-4 Trees: Insertion

- Like B-tree, insertion is done at the leaf node.
- Find where the item is to be inserted
- Insert the item
- Update the node
  1. Insertion into a 1-item-node => 2-item-node
  2. Insertion into a 2-item-node => 3-item-node
  3. Insertion into a 3-item-node => 4-item-node

# 2-4 Tree

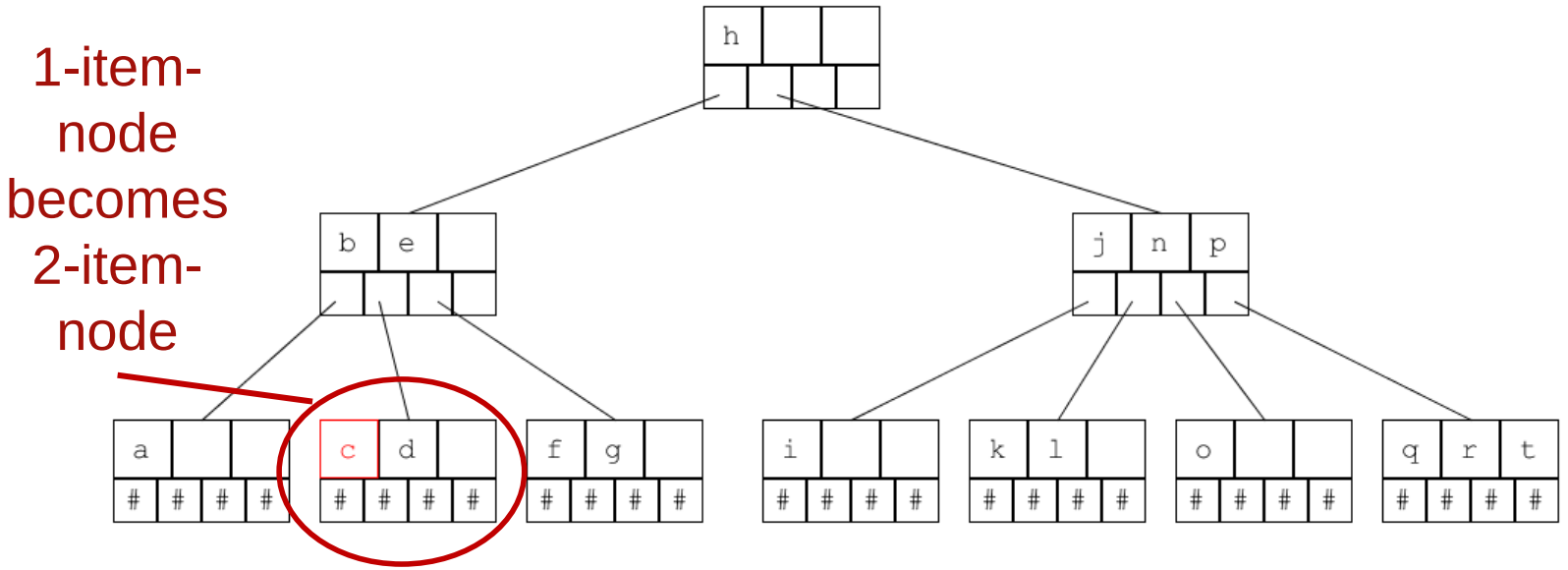
2-4 Trees: Insertion (…continue)

- (if node is overflowed), split this node and update its parent.
  1. Repeat with the parent if necessary.
  2. Create a new root layer if necessary.

# 2-4 Tree

## 2-4 Trees: Insertion

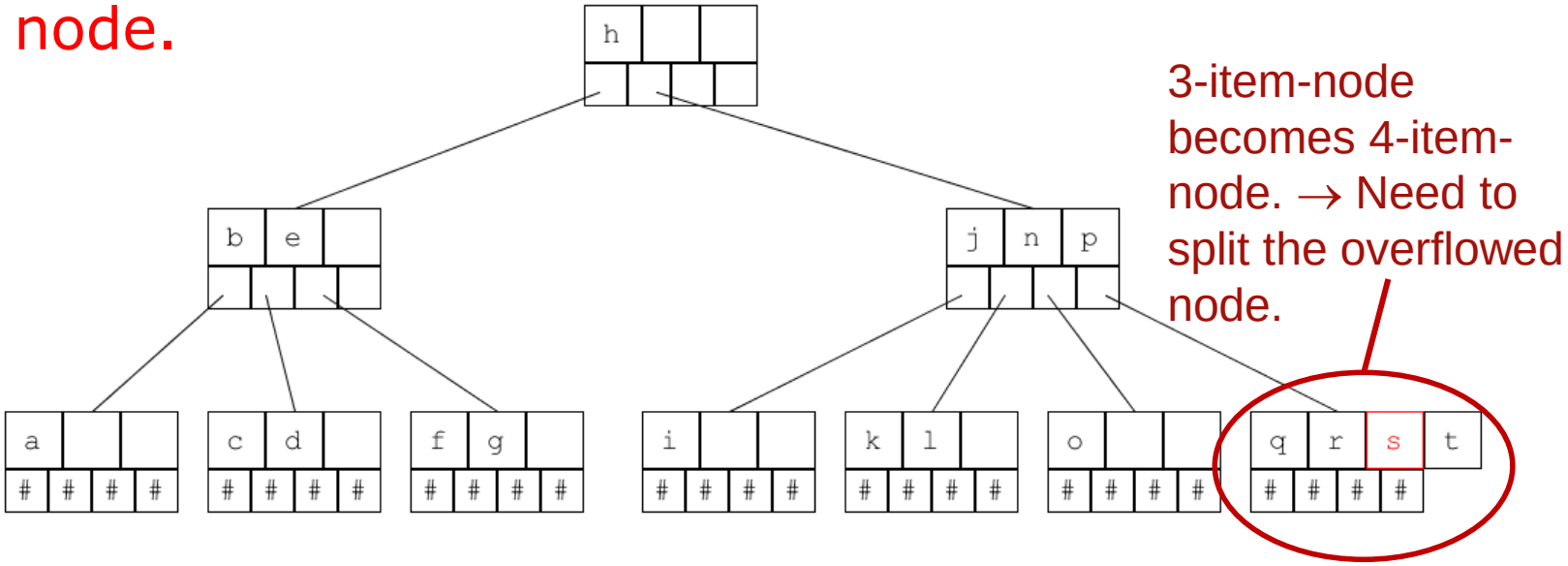
E.g. insert “c”: 1-item-node becomes 2-item-node



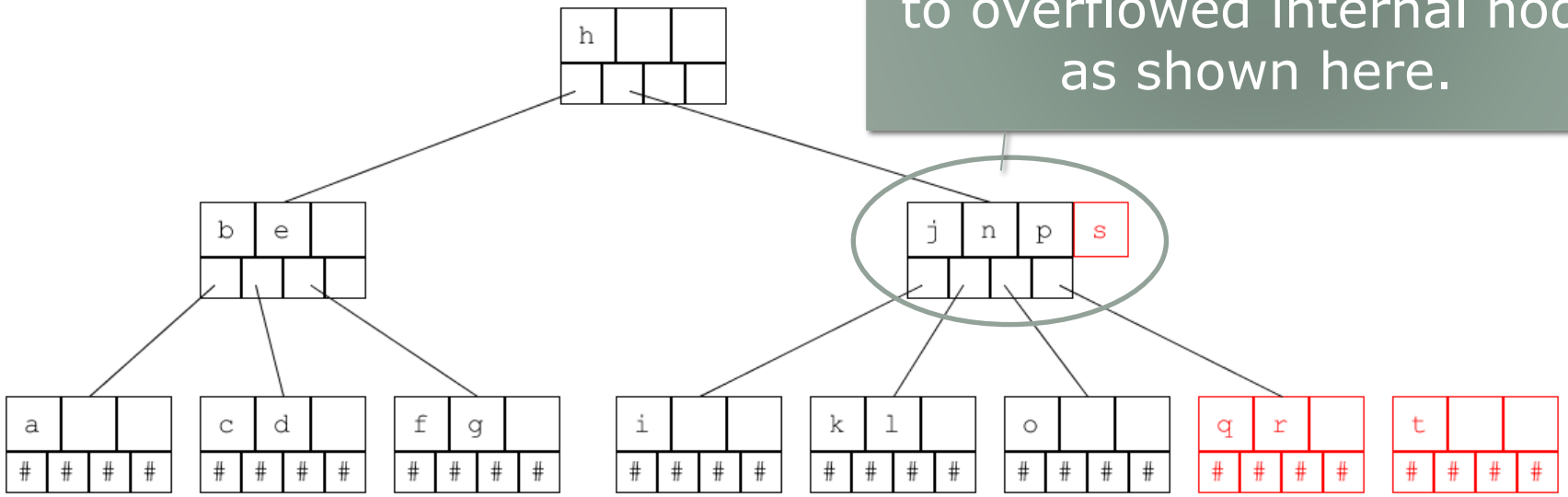
# 2-4 Tree

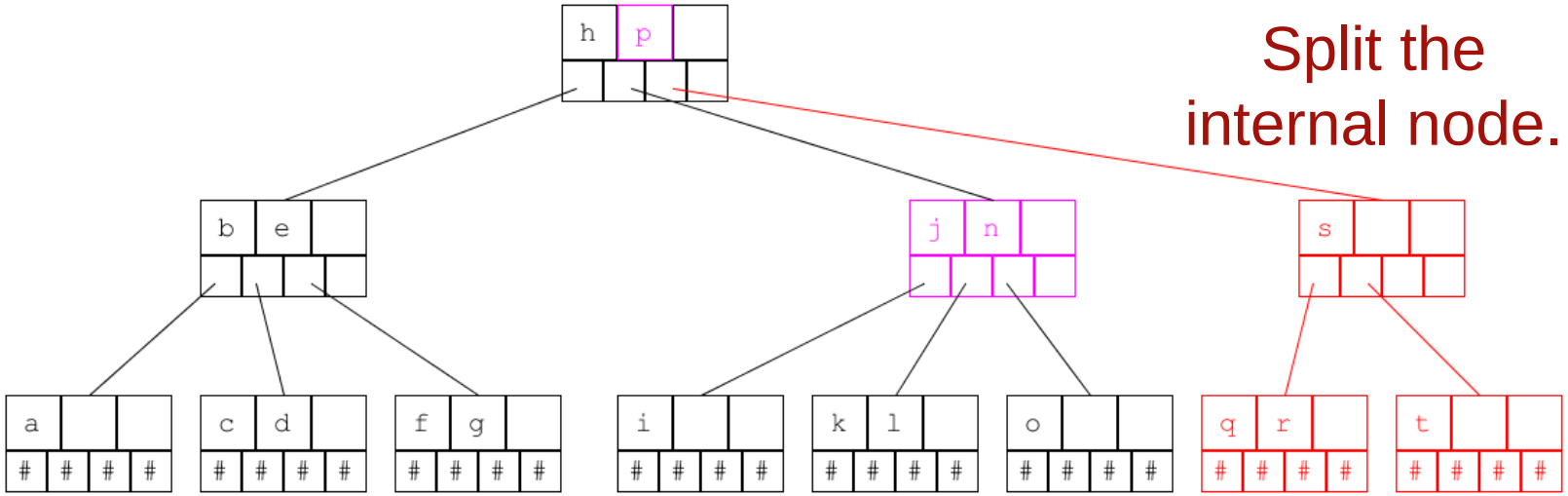
## 2-4 Trees: Insertion

E.g. insert "s": 3-item-node becomes 4-item-node. This is an overflowed situation. Need to split the node.



Unfortunately, in this instance, splitting of the overflowed leaf node leads to overflowed internal node as shown here.



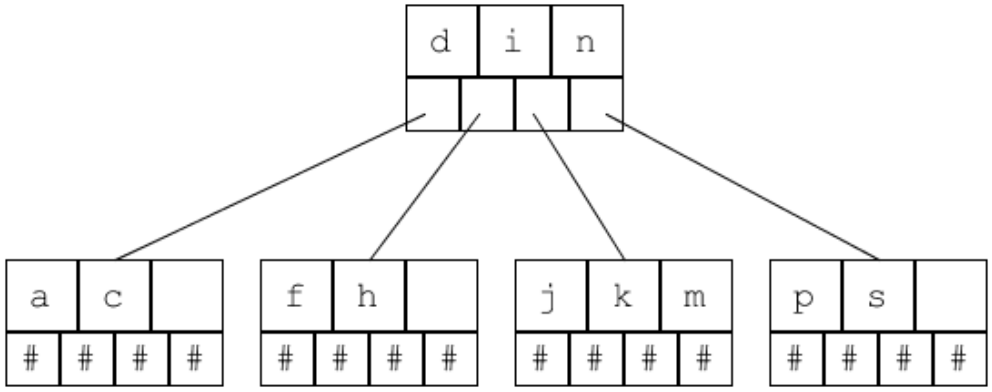


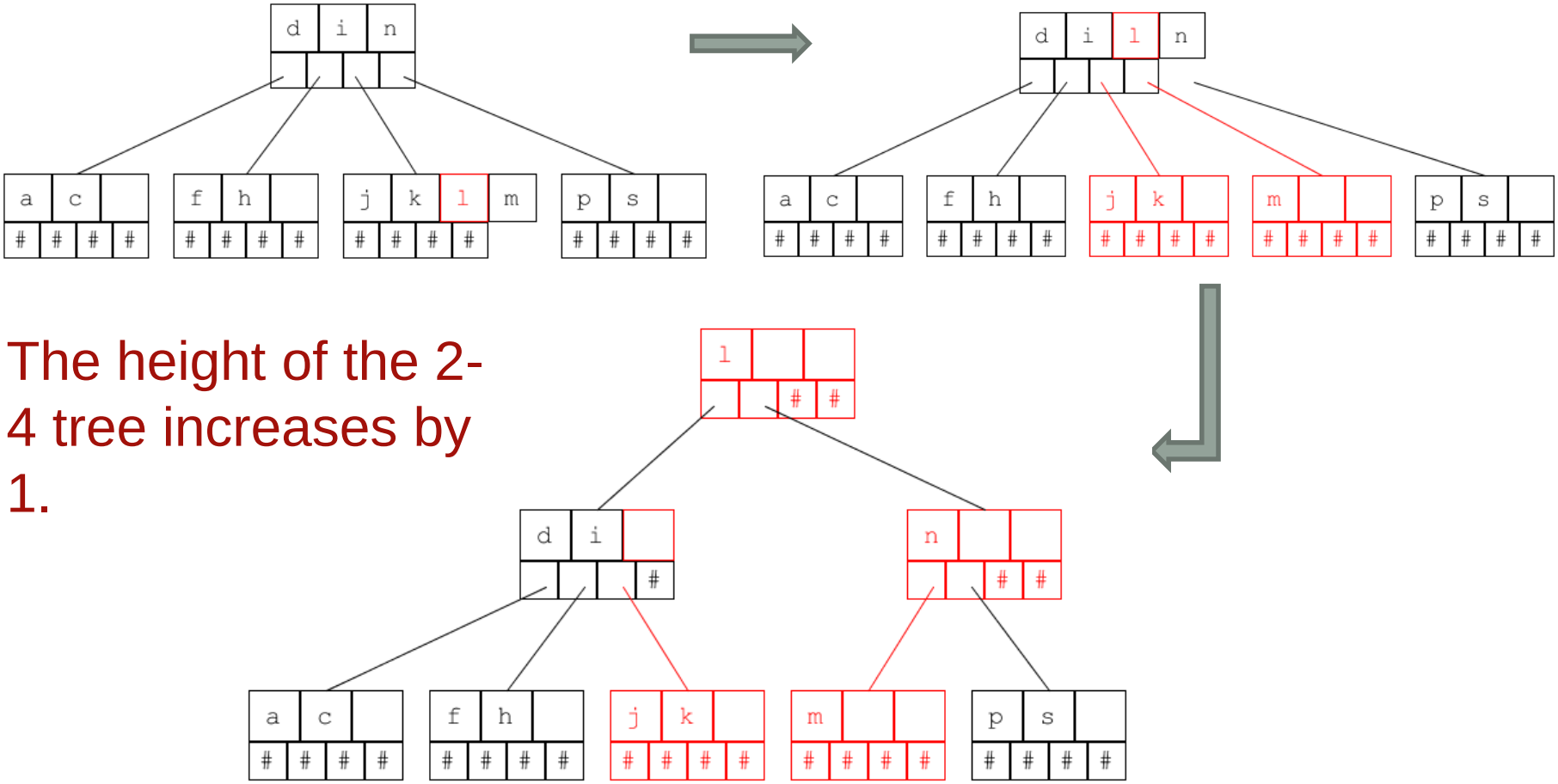

Note the different in the splitting of an internal node as compare to the splitting of a leaf node. In the splitting of an internal node, the largest element of the left node (‘p’ in this example,) moves up to the next level.

# 2-4 Tree

## 2-4 Trees: Insertion

- If the root node becomes a 4-item-node, split the root node and create a new root
- This increases the height of the tree by one.
- For example:






The height of the 2-4 tree increases by 1.

# 2-4 Tree

## 2-4 Trees: Deletion

- Find where the item to be deleted is located
- Delete the item
- Update the node
  1. Deletion from a 3-item-node => 2-item-node
  2. Deletion from a 2-item-node => 1-item-node
  3. Deletion from a 1-item-node => 0-item-node
     - If the 1-item-node has a sibling with more than two children redistribute the children. Redistribution is done via left or right rotation.

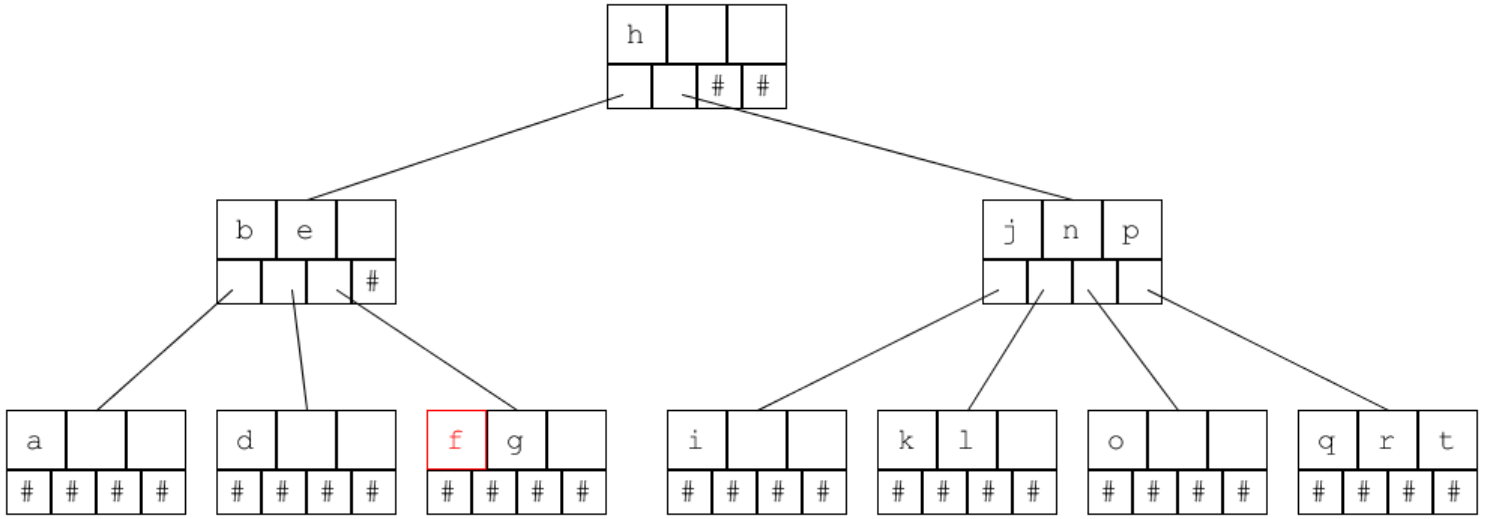
# 2-4 Tree

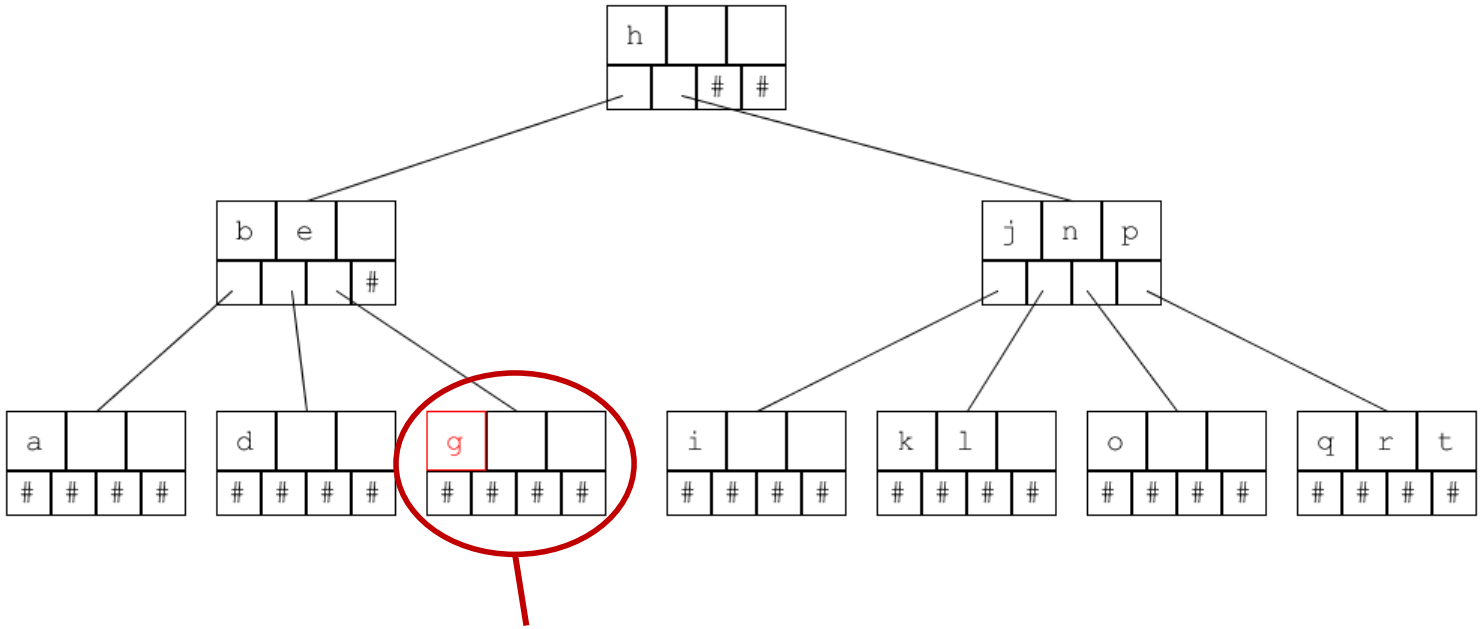
- If not, collapse the 1-item-node together with its sibling and its parent key into a 2-item-node and update the parent
- This may cause the parent to become a 1-item-node
  - If it does, fix this recursively

# 2-4 Tree

## 2-4 Trees: Deletion

E.g. delete “f”: 2-item-node becomes 1-item-node



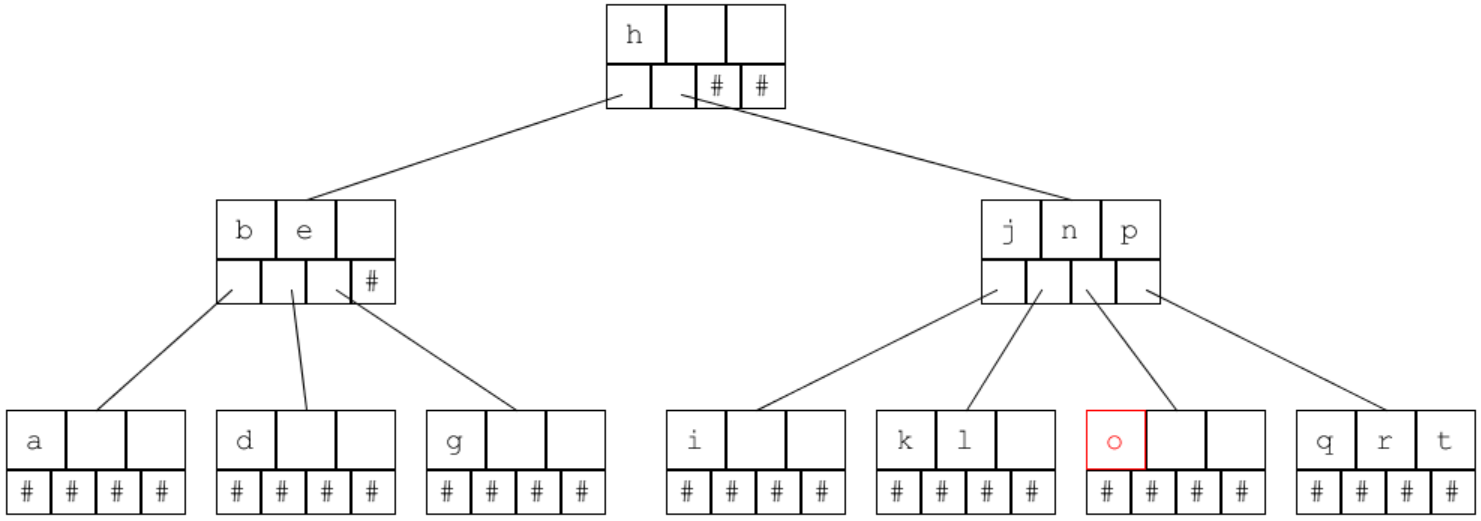


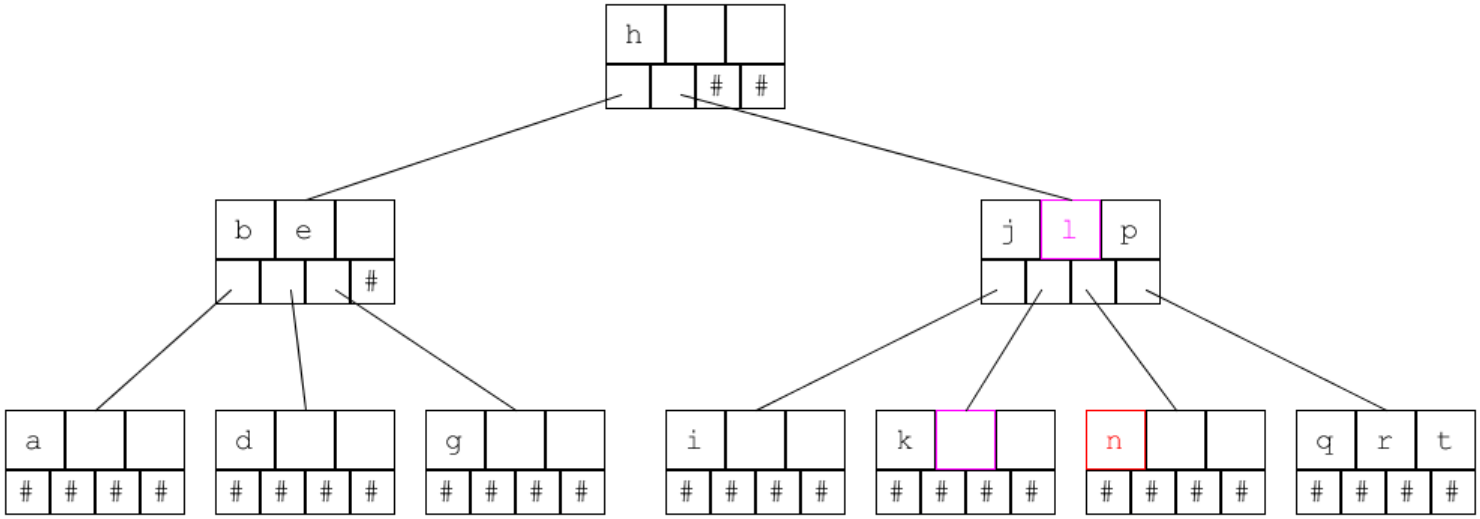

2-item-node becomes 1-item-node.

# 2-4 Tree

## 2-4 Trees: Deletion

E.g. delete "0": 1-item-node becomes 0-item-node



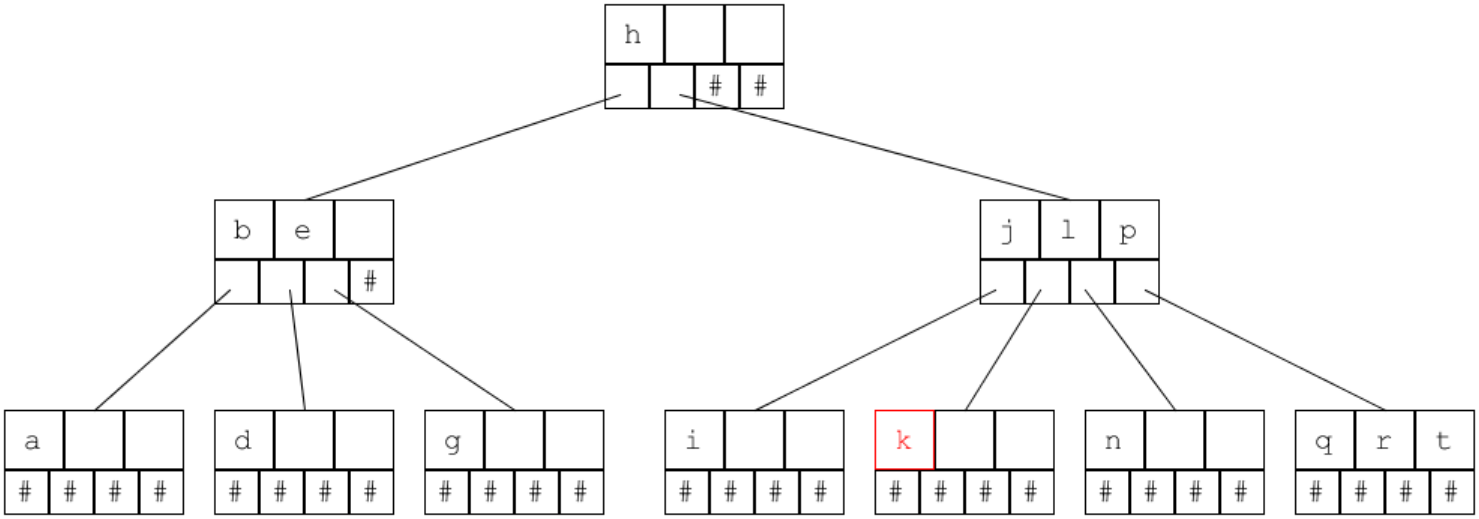


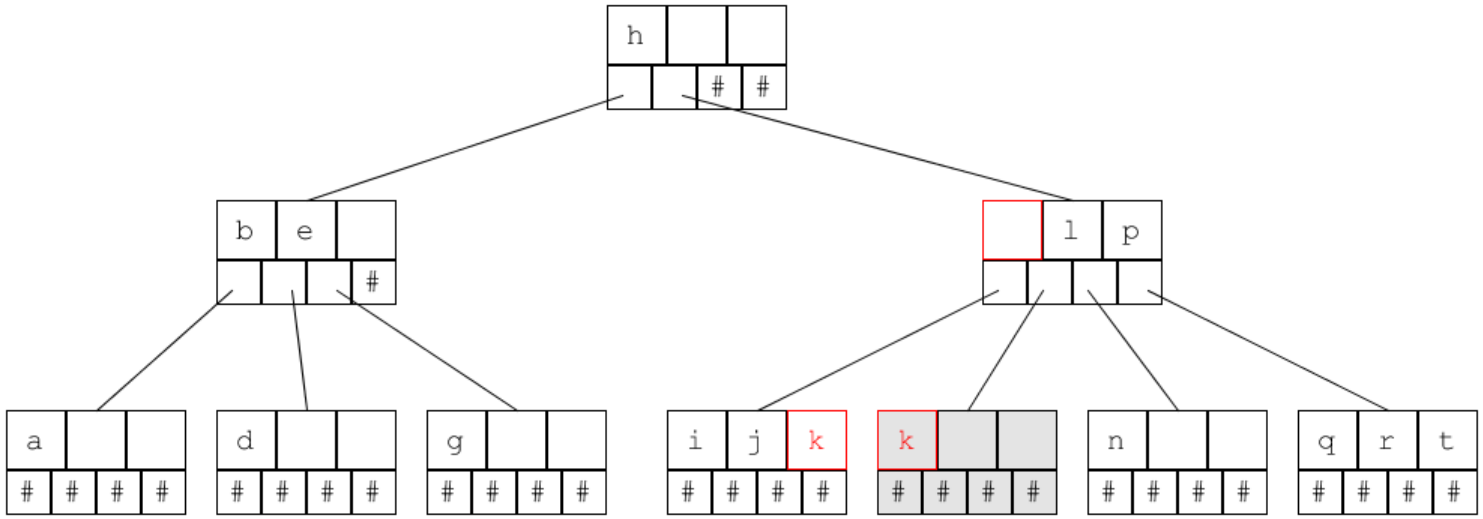

Left sibling has more than one item, re-distribute one item from left sibling. Item ‘l’ moves up to the parent, and item ‘n’ moves down to right sub-tree. In other words, perform a right-rotation on the item ‘l’ through the parent node.

# 2-4 Tree

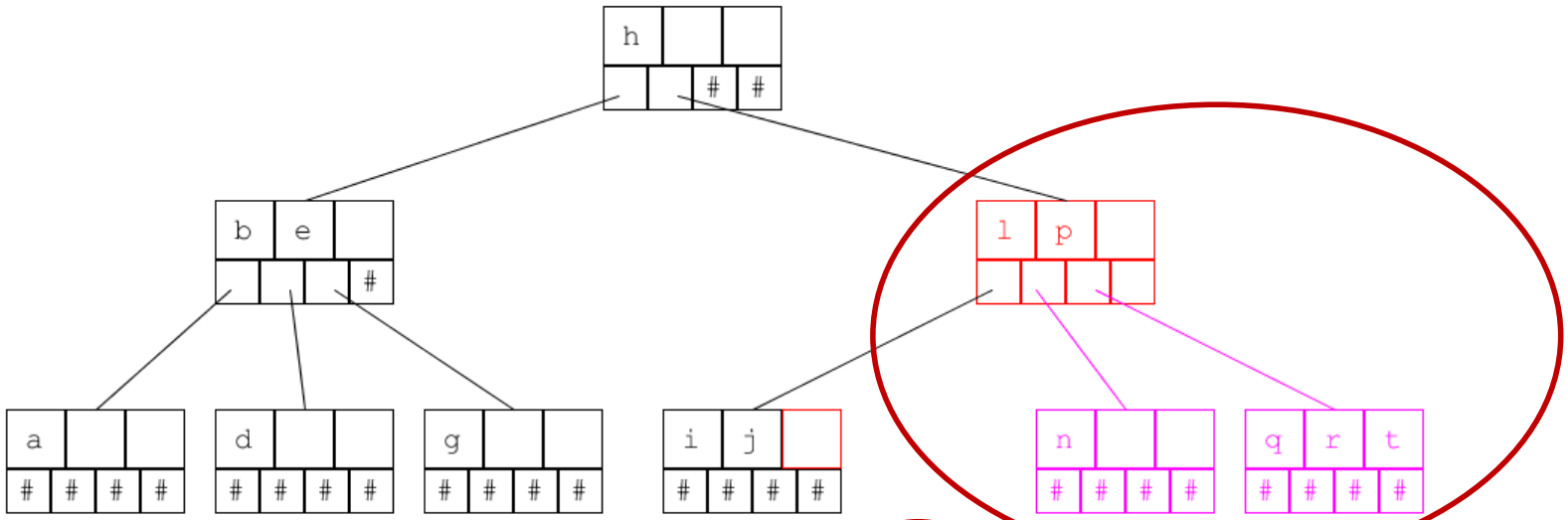
## 2-4 Trees: Deletion

- E.g. delete “k”: 1-item-node becomes 0-item-

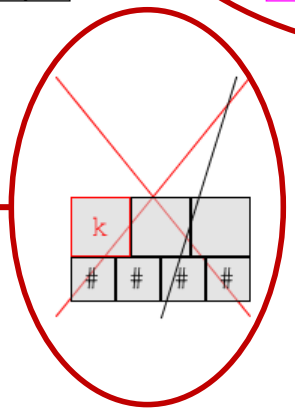




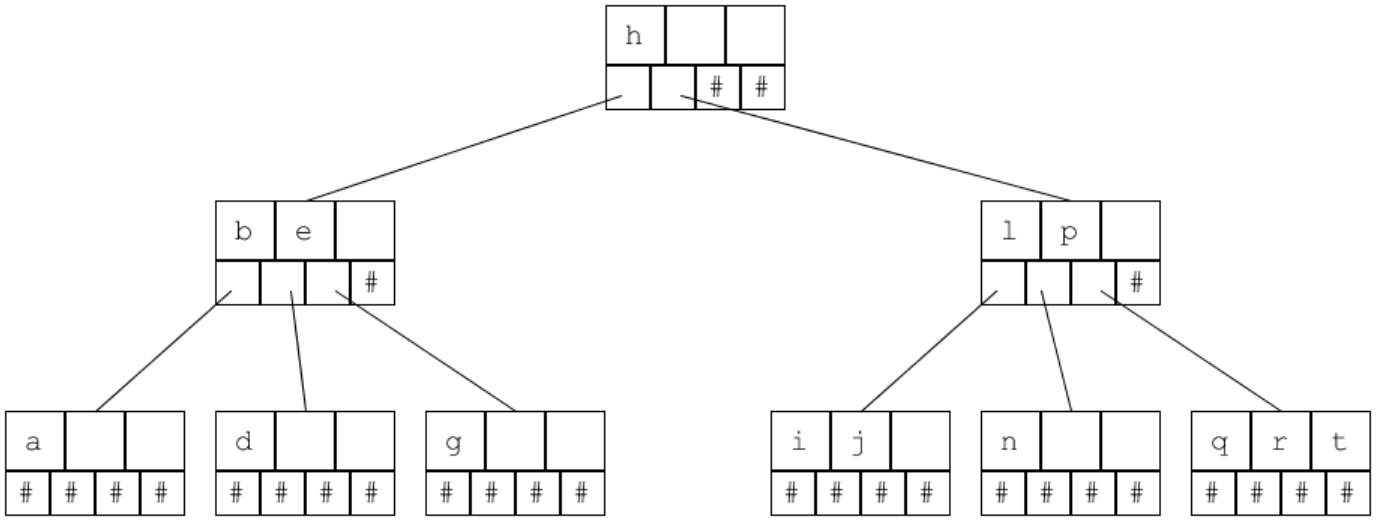

Sibling nodes (both left or right) do not have enough item to distribute, hence, need to collapse the 3 items (left ‘i’, root ‘j’, and right ‘k’) into one node. Remove the original node containing the item ‘k’, and make adjustment to the items in the parent node, that is, items ‘l’ and ‘p’.




Drop the original node containing the item ‘k’.




Adjust the items ‘l’ and ‘p’ together with their respective children.

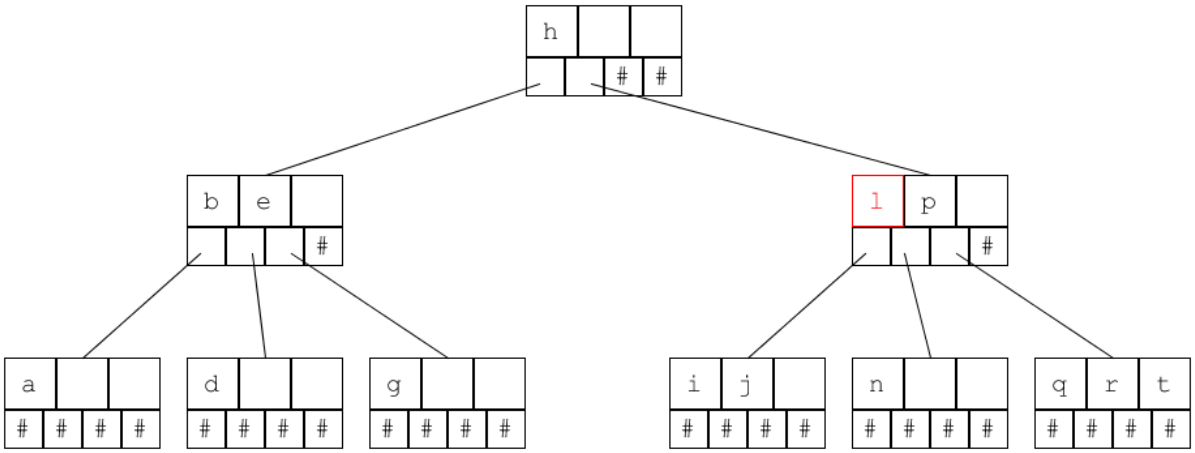



2-4 tree after deleting item ‘k’.

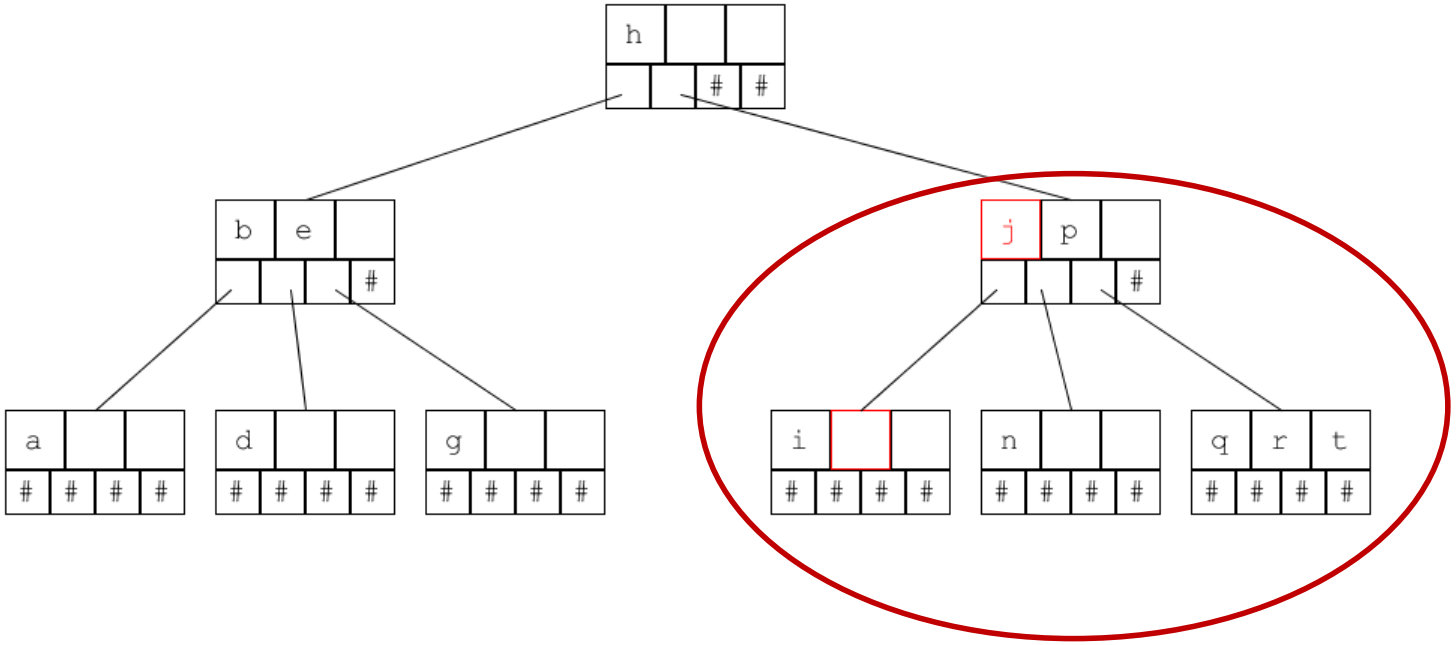
# 2-4 Tree

## 2-4 Trees: Deletion

- E.g. delete “l”: 2-item-node becomes 1-item-node.




Item ‘l’ is located in internal node; Left child contains more than 1 items, take the highest value-item ‘j’ left child to replace the item to be deleted from the parent node, that is, replace ‘l’ with ‘j’.

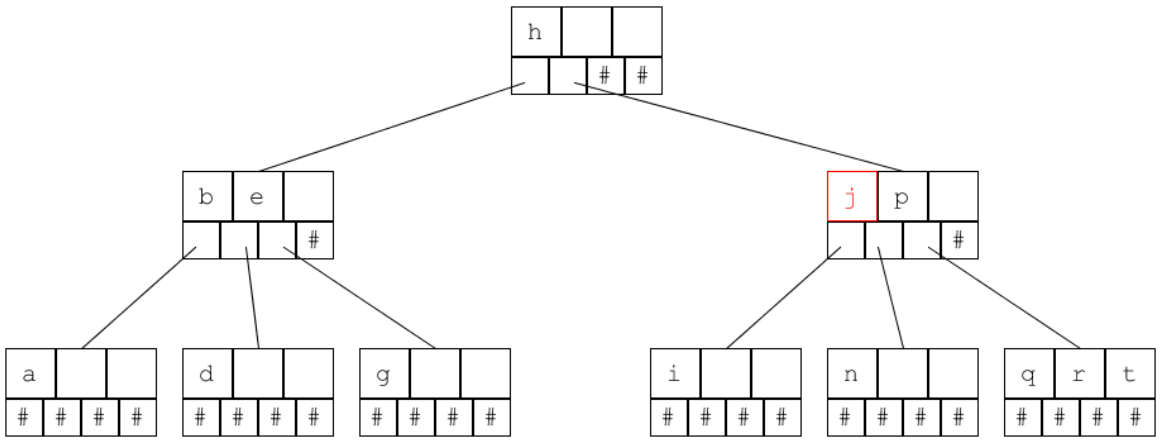



2-4 tree after item ‘l’ is removed.

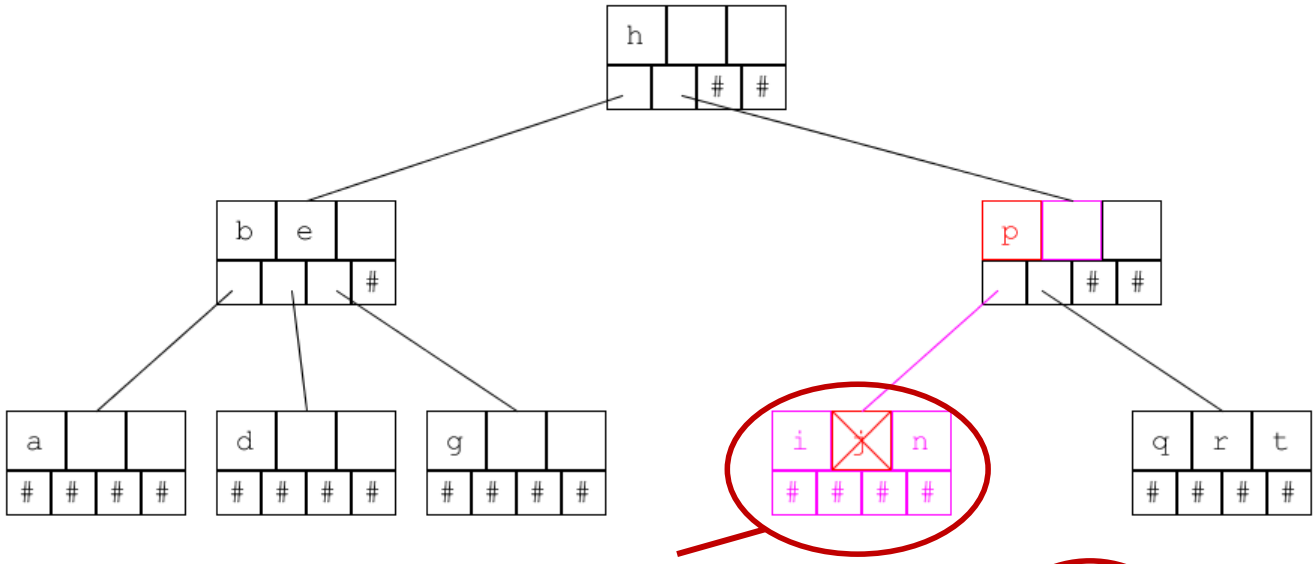
# 2-4 Tree

## 2-4 Trees: Deletion

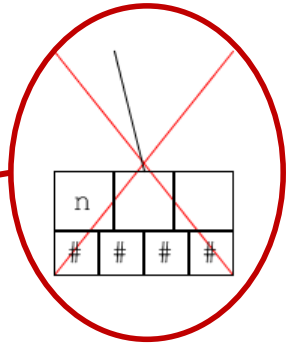
E.g. delete "j": 2-item-node becomes 1-item-node.




Item 'j' is located in internal node; But both children contain only 1 item each. Need to collapse the 3 items (left 'i', root 'j', and right 'n') into one node. Remove the item 'j' and the original node containing the item 'n', and make adjustment to the item in the parent node, that is, item 'p'.



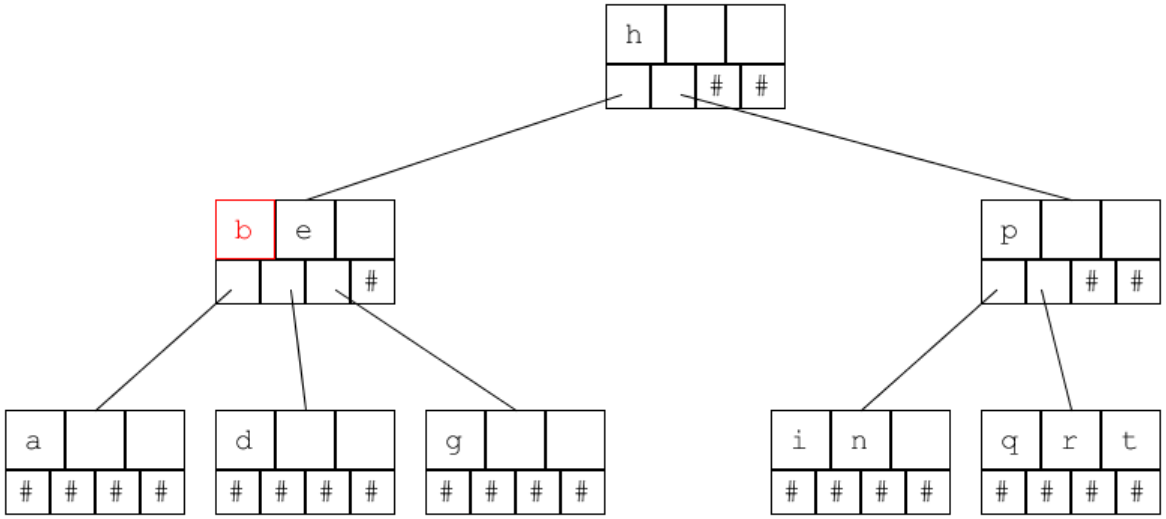

Remove the original node containing the item ‘n’.



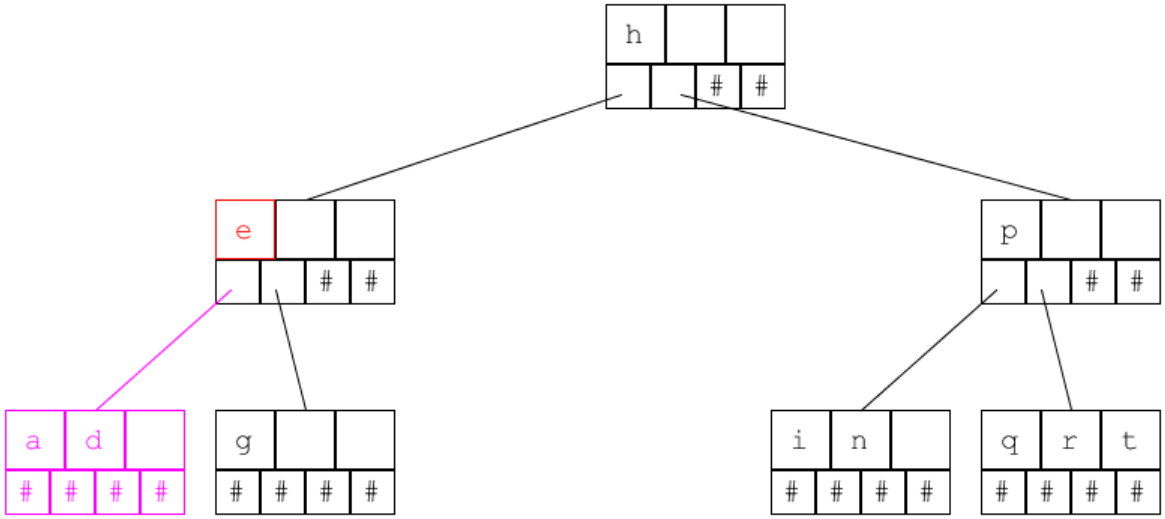
# 2-4 Tree

2-4 Trees: Deletion

E.g. delete “b”: 2-item-node becomes 1-item-node.




This example is similar to the previous example.

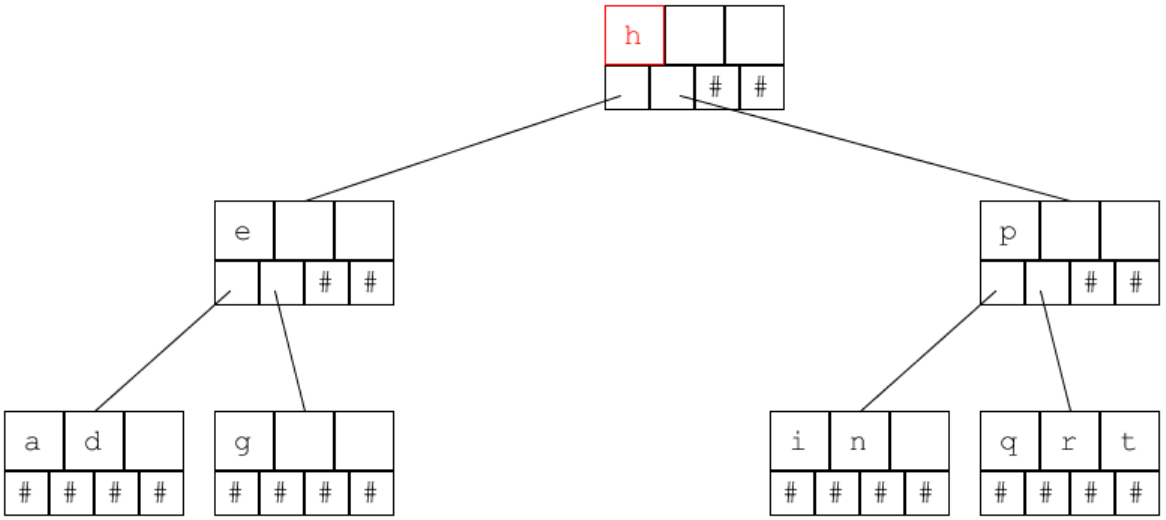



2-4 tree after item ‘b’ is removed.

# 2-4 Tree

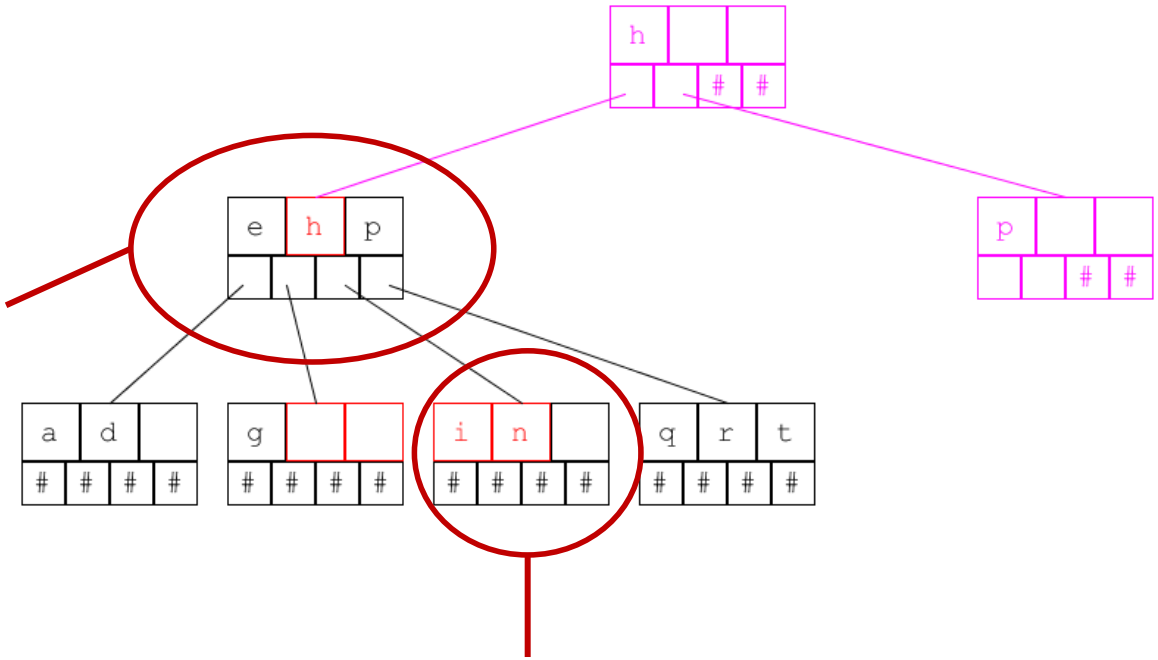
## 2-4 Trees: Deletion

E.g. delete “h”: 1-item-node becomes 0-item-node

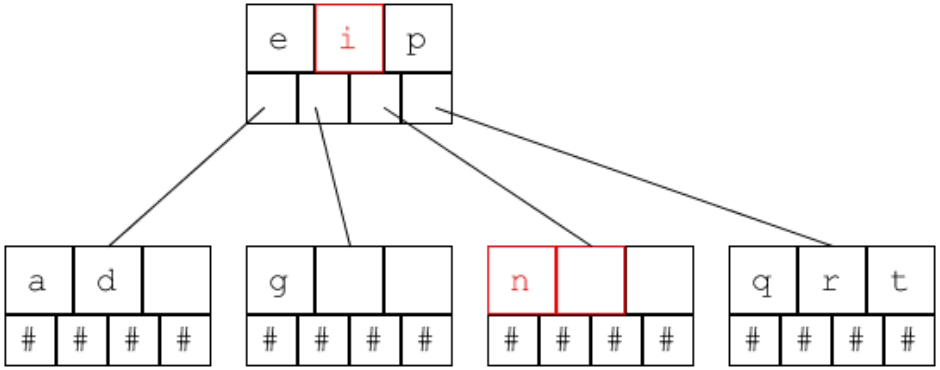



Item ‘h’ is located in root, and since there is only one item in root, the deletion will collapse the 2-4 tree by one level.

Both children of the root contain only one item each, hence, need to collapse the three nodes into one node containing the items of the three nodes, that is, item ‘e’ (left child), item ‘h’ (root), and item ‘p’ (right child).




After the nodes are collapse into, the left sub-tree of item ‘p’ becomes the right sub-tree of item ‘h’ as shown above.




Delete ‘h’ from the root, and now since the right-child contains more than one items, we replace ‘h’ with ‘i’, the smallest value-item from the right child.

# 2-4 Tree

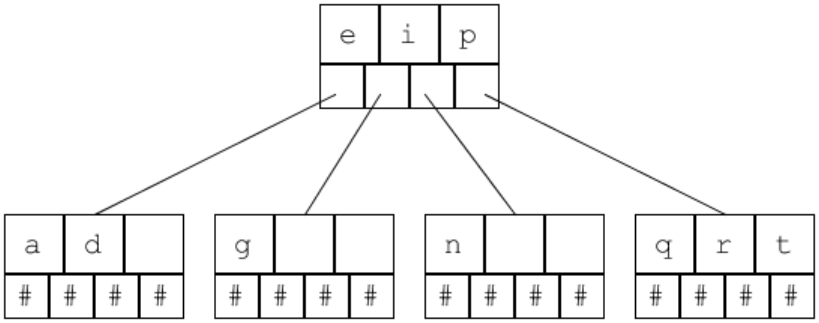
Note: I use $\log_2 n$ or $\lg n$ to represent logarithm to the base 2. When $\log n$ is used, the base is then depend in the context.

- 2-4 Trees: Efficiency
  - Height of tree is $O(\log_4 n)$
  - Searching:
    - Each node checked takes $O(1)$
    - Search takes $O(\log_4 n)$
  - Insertion:
    - Split takes $O(1)$ at each level
    - At most $\log_4 n$ splits (1 per level)
    - Insertion takes $O(\log_4 n)$

# 2-4 Tree

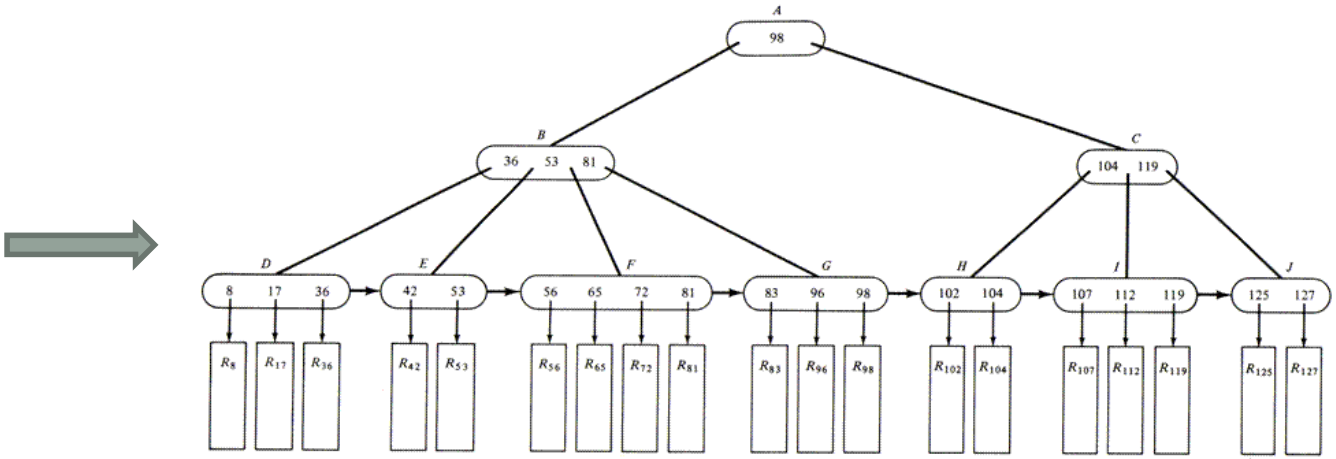
- Deletion
  - Merge takes $O(1)$
  - At most $\log_4 n$ merges
  - Deletion takes $O(\log_4 n)$


B*-Tree




2-4 tree

B* – tree



# 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 $q$ and at least $\lceil (2 \times (q-1))/3 \rceil$ children, where $q$ is the order of the tree and 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 $q$ is as follows:

1. Each internal node is of the form

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

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

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

# B*-tree

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

3. For all search field values $X$ in the sub-tree pointed at by $Pr_i$, we have $K_{i-1} < X \leq K_i$ for $1 < i < q; X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.
4. Each internal node, except the root, has at most $\lceil (2 \times (q-1))/3 \rceil$ tree pointers.

# B*-tree

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

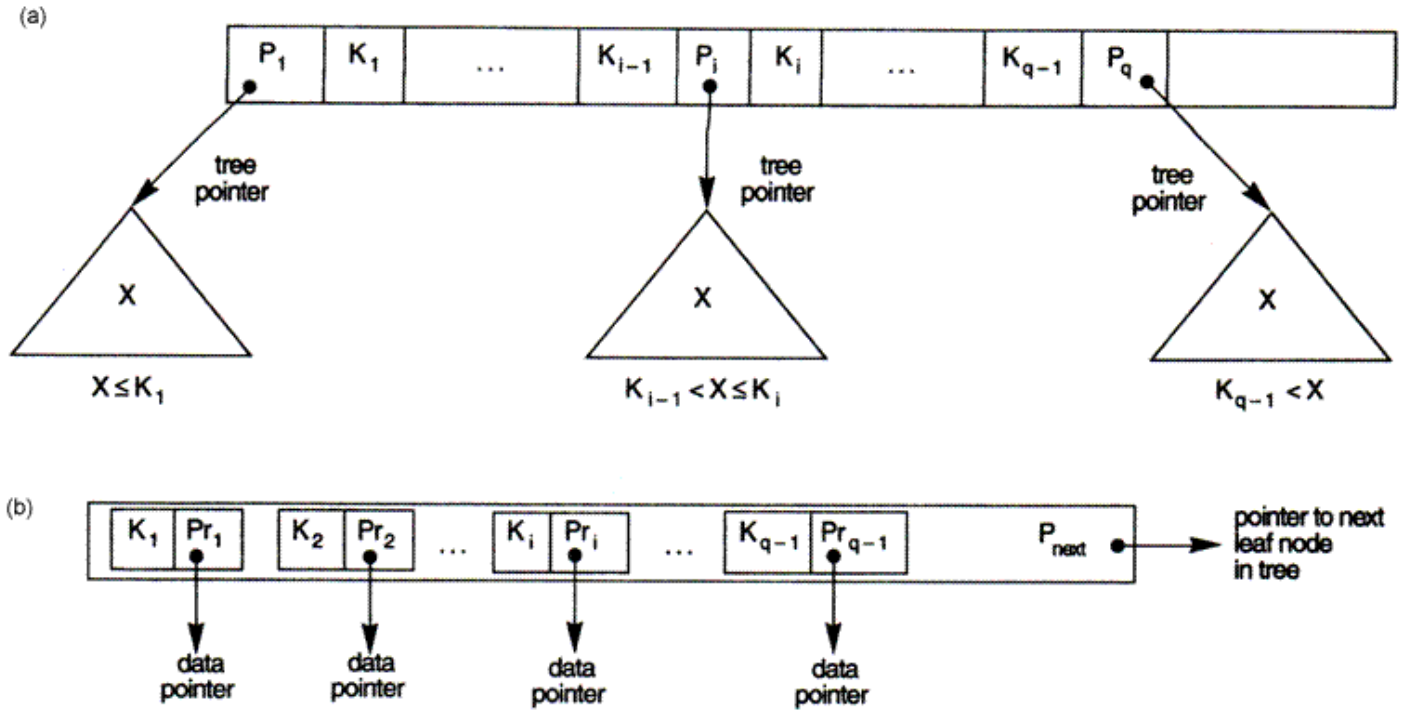
5. The root node has at least two and at most $2 \times \lfloor (2 \times (q-2))/3 \rfloor + 1$ tree pointers.

# B*-tree

The structure of the **internal nodes** of a B*-tree of order $q$ 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 (guideline only, may vary in implementation), listed from left to right accordingly with the positions of the nodes in the tree:
   - $\lfloor (2 \times (q-2))/3 \rfloor$ $(left\ node)$
   - $\lfloor (2 \times (q-1))/3 \rfloor$ $(middle\ node)$
   - $\lfloor (2 \times q)/3 \rfloor$ $(right\ node)$

# B*-tree


(a)
P1
K1
...
Ki-1
Pi
Ki
...
Kq-1
Pq
tree pointer
tree pointer
tree pointer
X
X
X
X≤K1
Ki-1<X≤Ki
Kq-1<X
(b)
K1
Pr1
K2
Pr2
...
Ki
Pri
...
Kq-1
Prq-1
Pnext
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 $q$ is as follows:

1. Each leaf node is of the form
   $<< K_1, \mathrm{Pr}_1 >, < K_2, \mathrm{Pr}_2 >, \ldots , < K_{n-1}, Pr_{n-1} >, P_{next} >$
   Where $n < q$ and each $Pr_i$ is a data pointer, and $P_{next}$ points to the next leaf node of the B*-tree.

2. Within each leaf node, $K_1 < K_2 < \cdots < K_{n-1}$, where $n < q$.

# B*-tree

The structure of the **leaf-node** of a B*-tree of order $q$ 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.
4. 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 $P_r$ = 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 $p_{leaf}$ for the leaf nodes can be calculated as follows:

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

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

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

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.

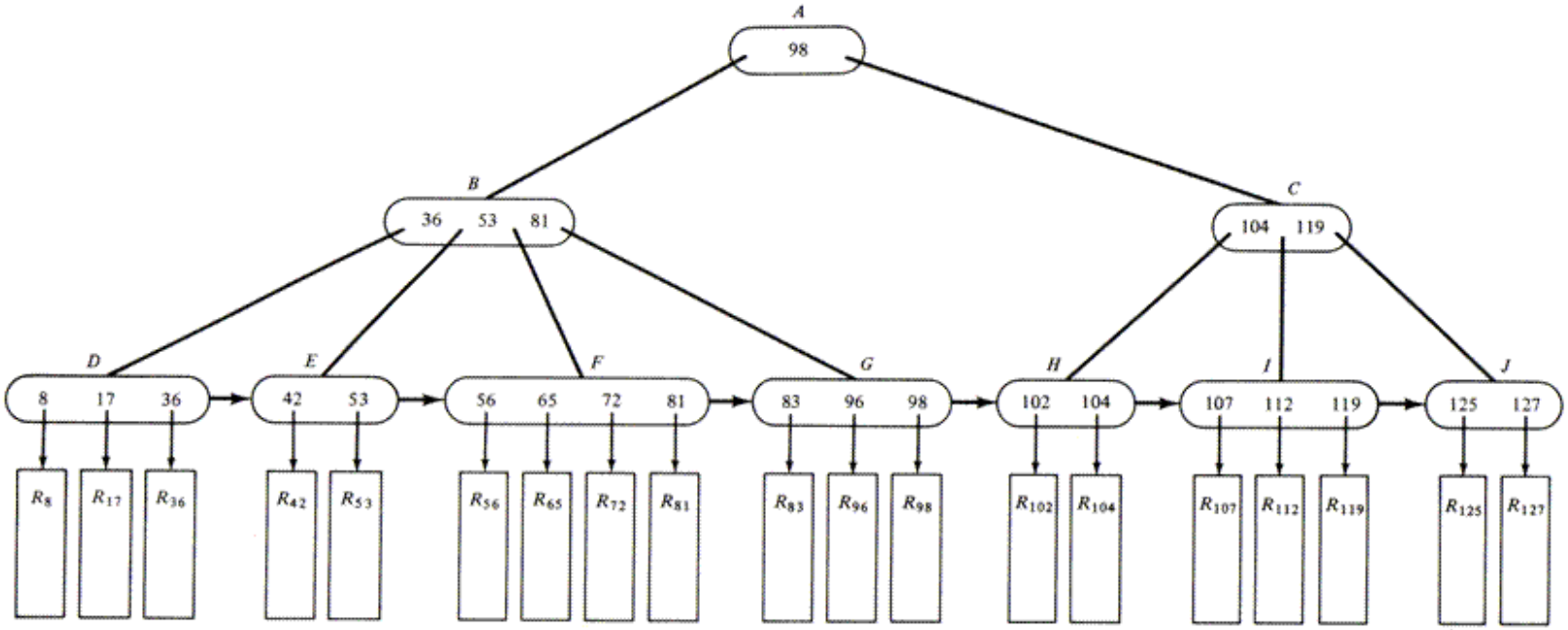
# B*-tree

**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
R8 R17 R36
R42 R53
R56 R65 R72 R81
R83 R96 R98
R102 R104
R107 R112 R119
R125 R127

# 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.

# Insertion into B*-tree

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.
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.

# Insertion into B*-tree

**procedure for insert into B*-tree**

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.

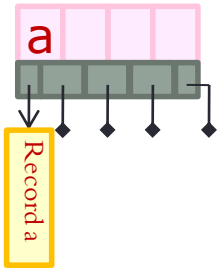
# Insertion into B*-tree

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.

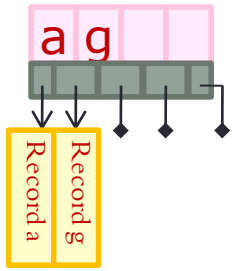
# Insertion into B*-tree

Insert: a, g, f, b

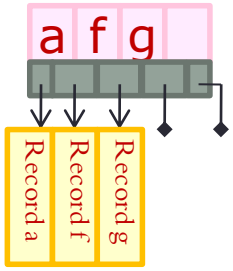
Insert: a



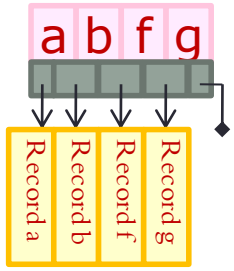

Insert: g




Insert: f

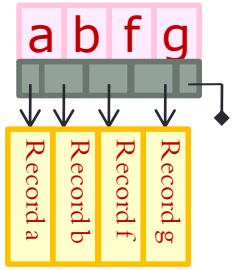



Insert: g



# Insertion into B*-tree

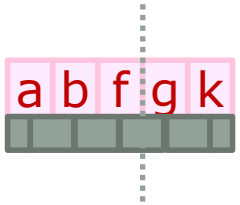
B*-tree after the insertion of a, g, f, and b.

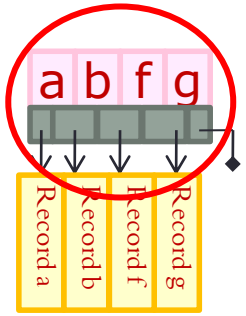


# Insertion into B*-tree

Insert: **k**

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

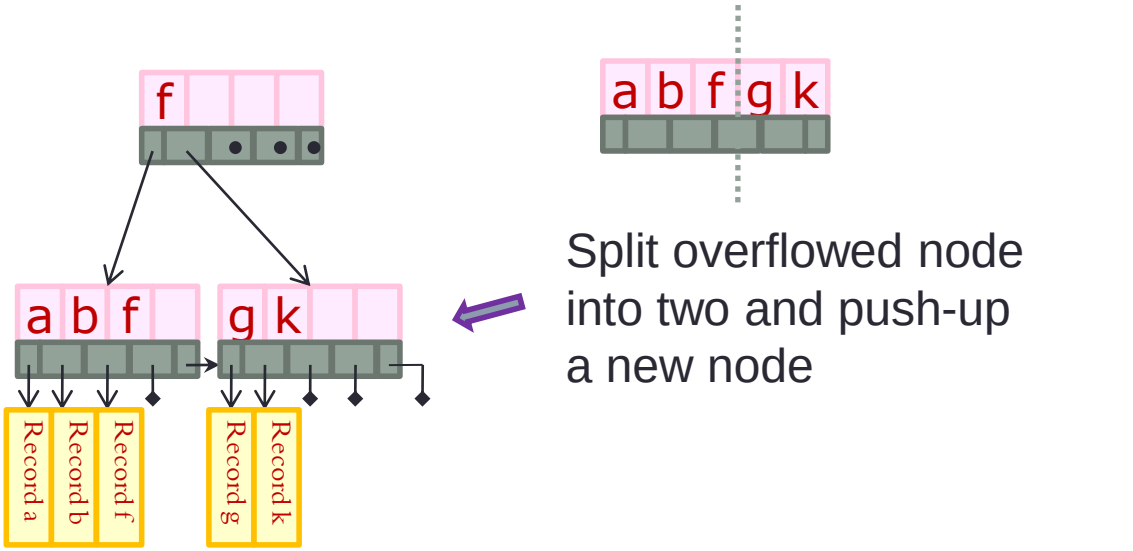


# Insertion into B*-tree

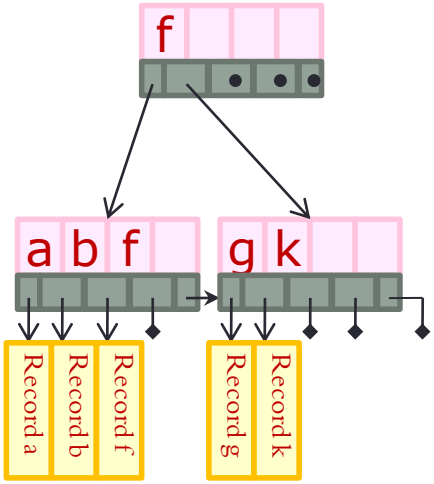
Insert: **k**

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



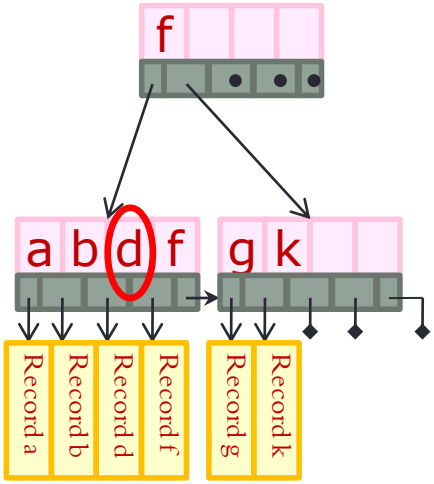
# Insertion into B*-tree

B*-tree after the insertion of k.



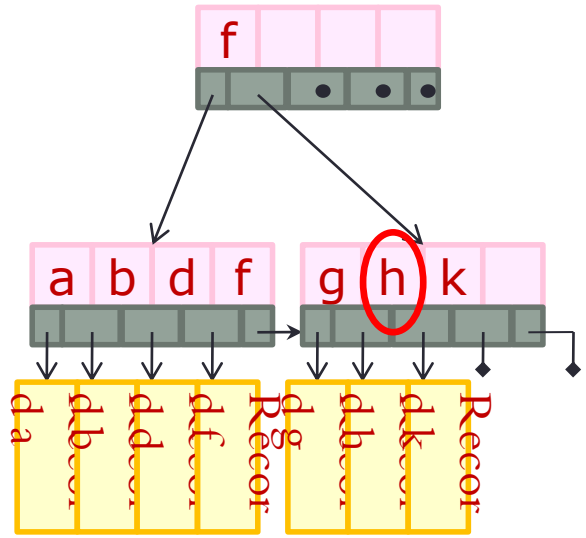
# Insertion into B*-tree

Insert: **d**, h, m



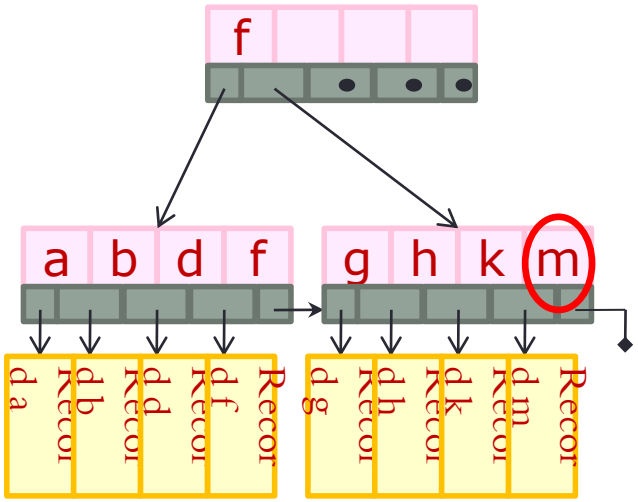
# Insertion into B*-tree

Insert: d, **h**, m



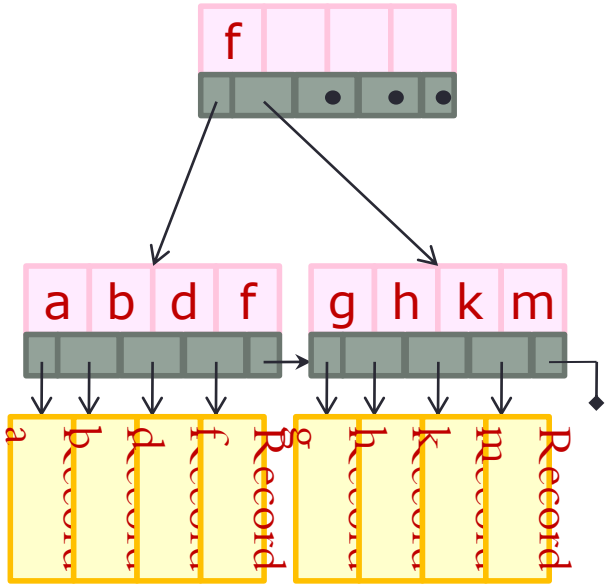
# Insertion into B*-tree

Insert: d, h, **m**



# Insertion into B*-tree

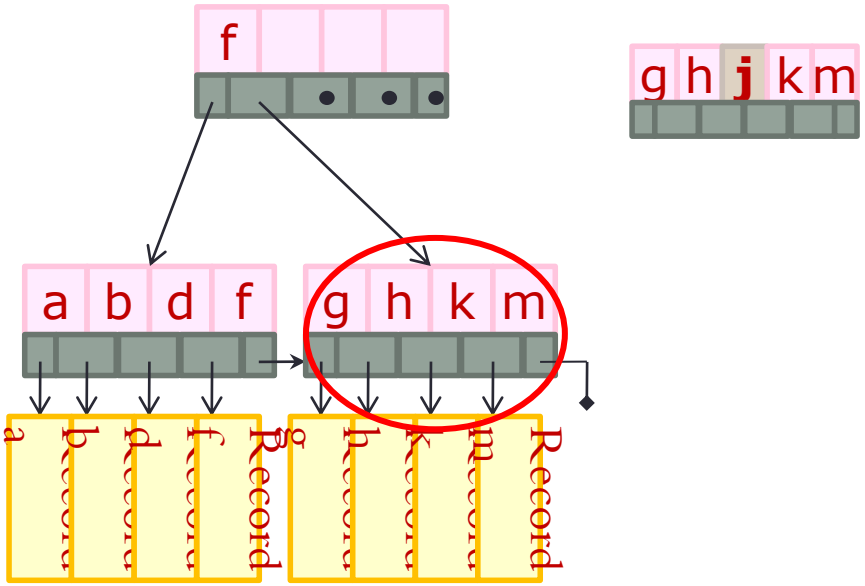
B*-tree after the insertion of d, h, and m.



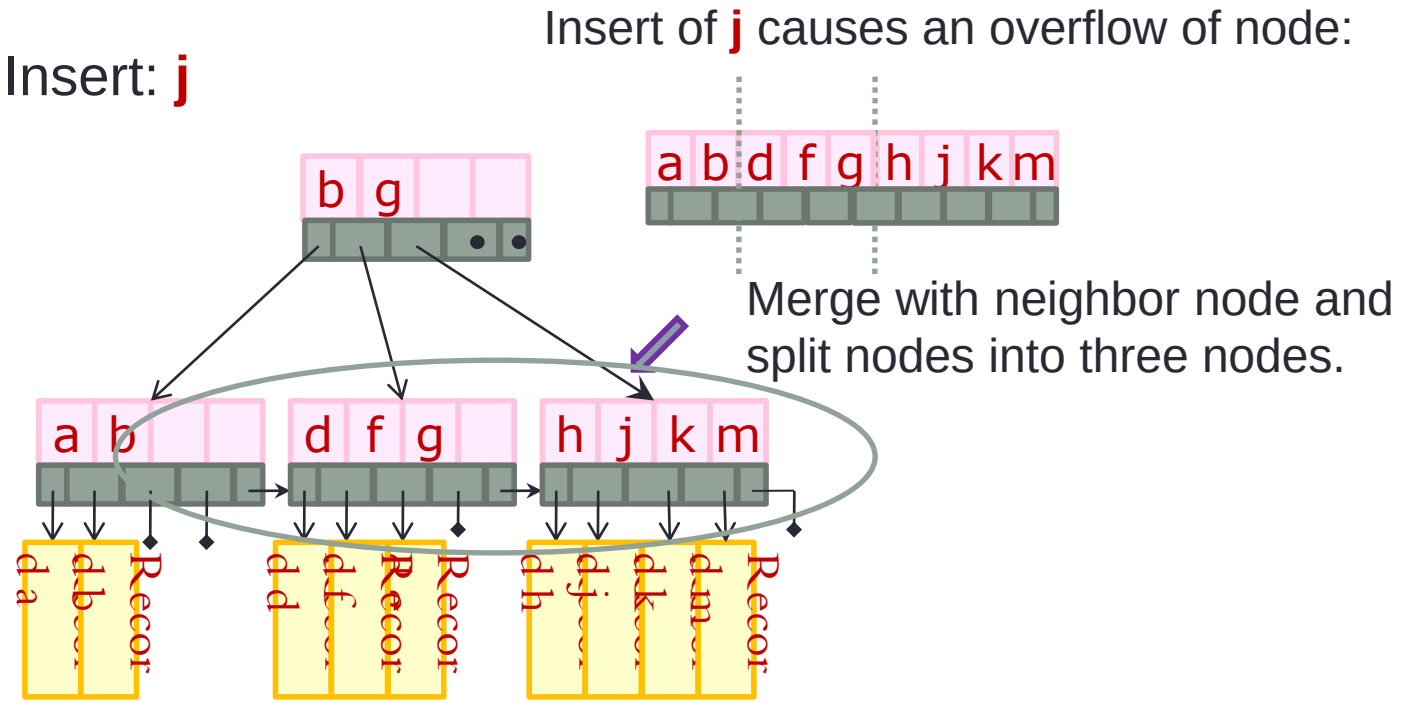
# Insertion into B*-tree

Insert: **j**

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

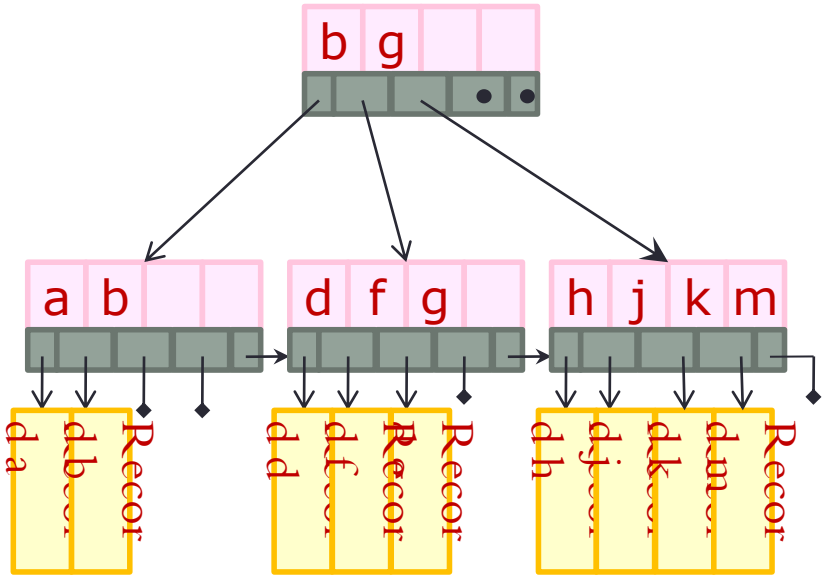


# Insertion into B*-tree



# Insertion into B*-tree

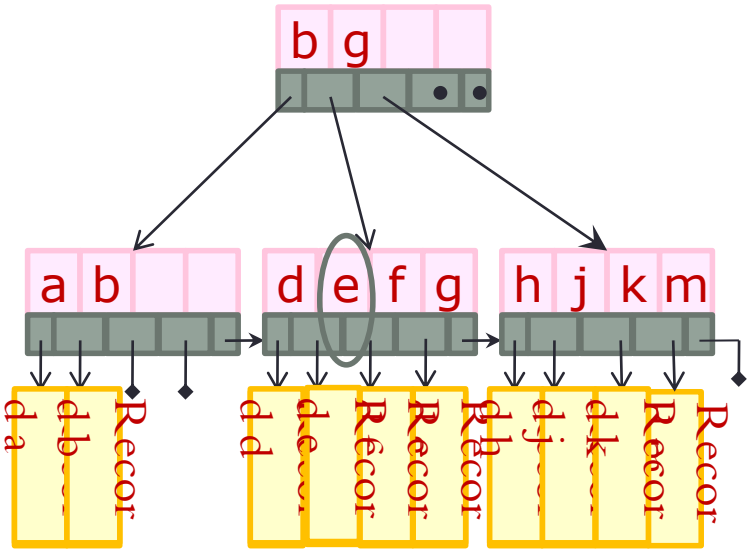
B*-tree after the insertion of j.



# Insertion into B*-tree

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

Insert: **e**

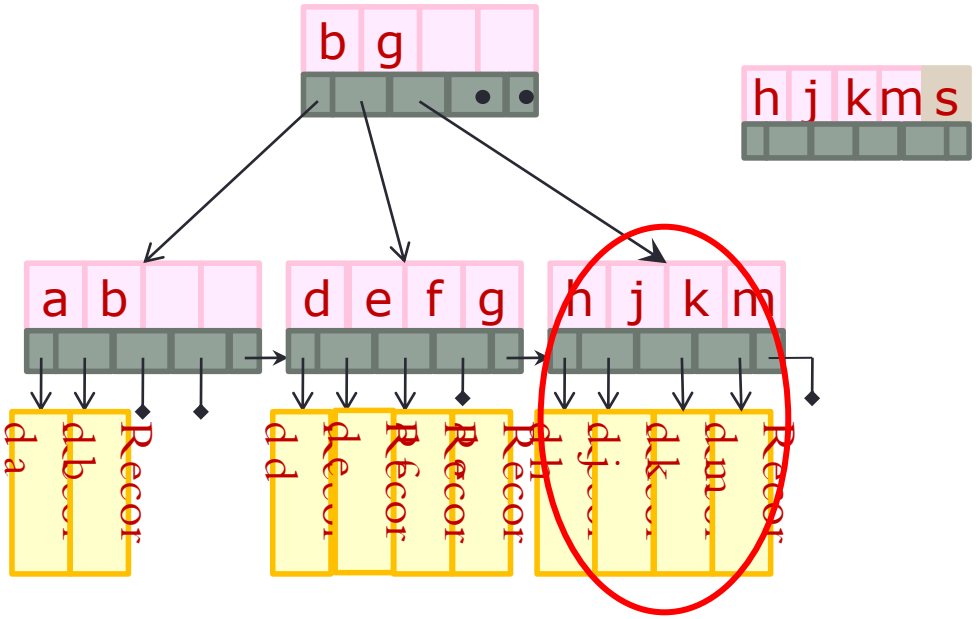


# Insertion into B*-tree

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

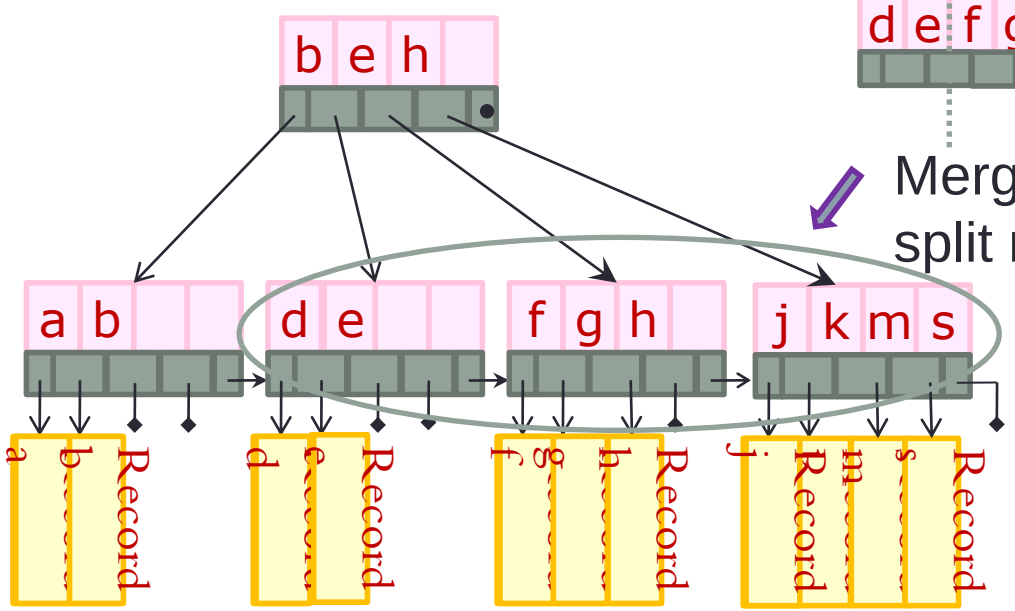
Insert **s.**

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

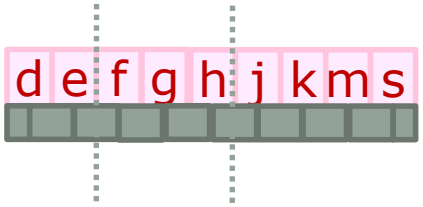


# Insertion into B*-tree

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




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

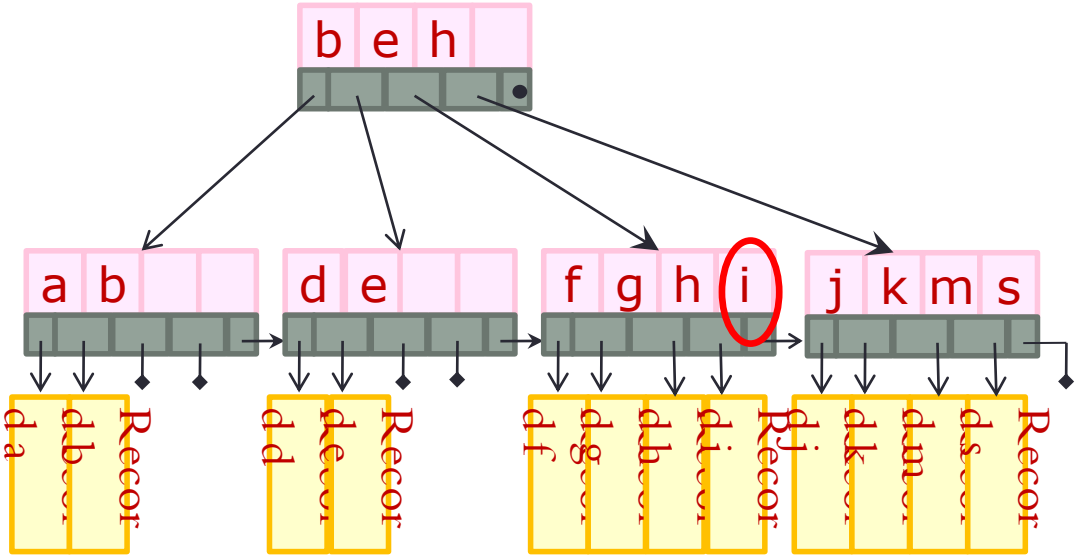



Merge with neighbor node and split nodes into three nodes.

# Insertion into B*-tree

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

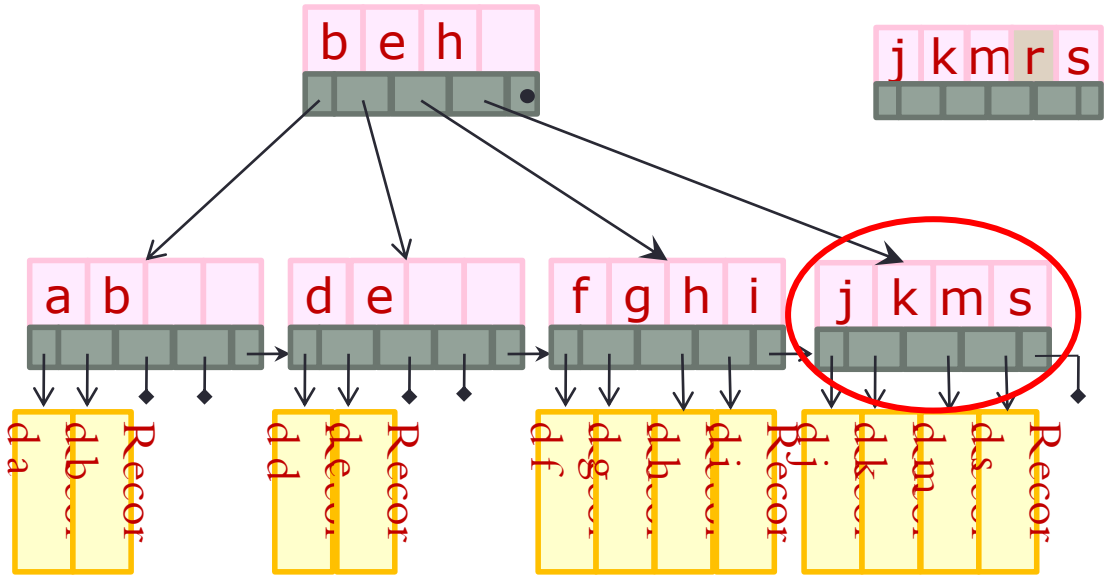
Insert **i.**



# Insertion into B*-tree

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

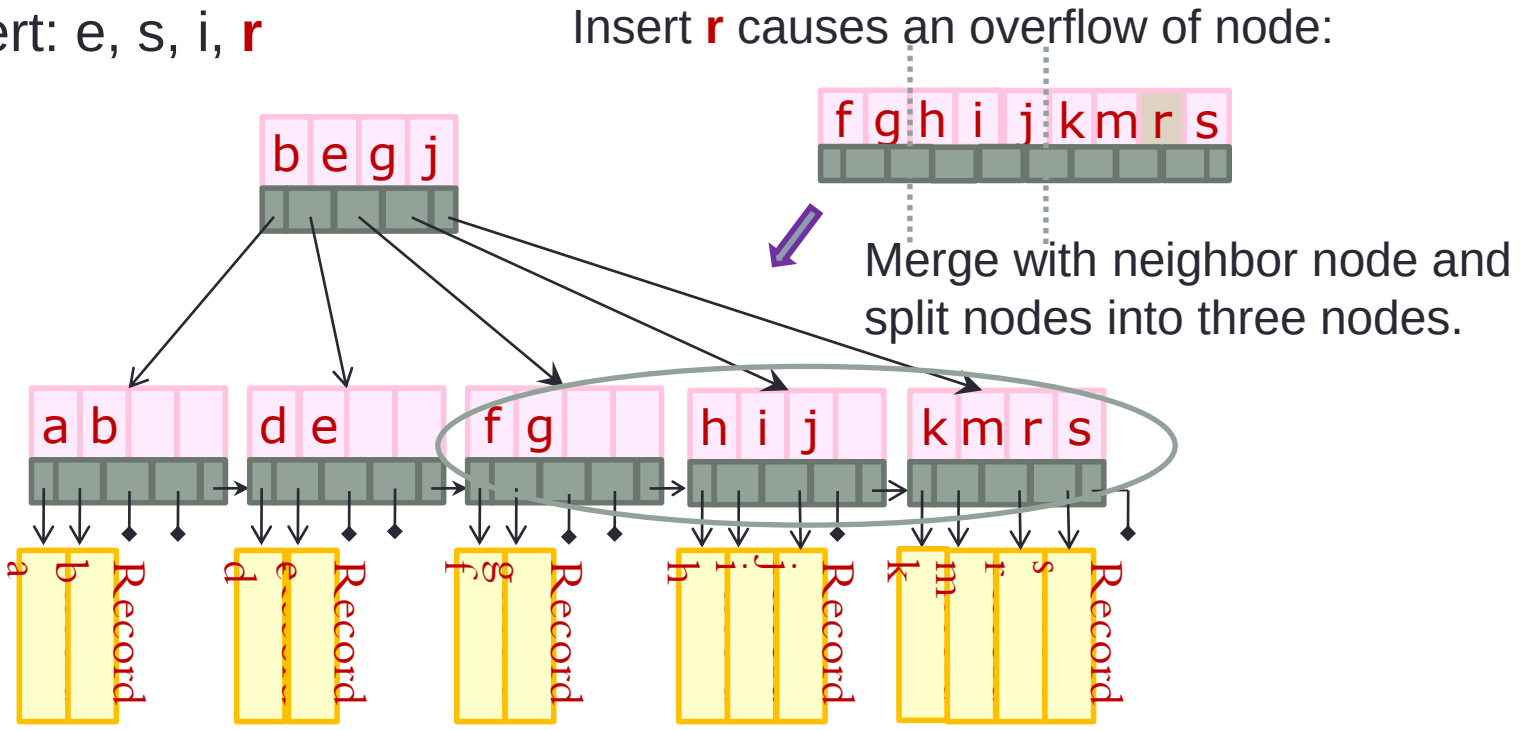
Insert **r.**




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

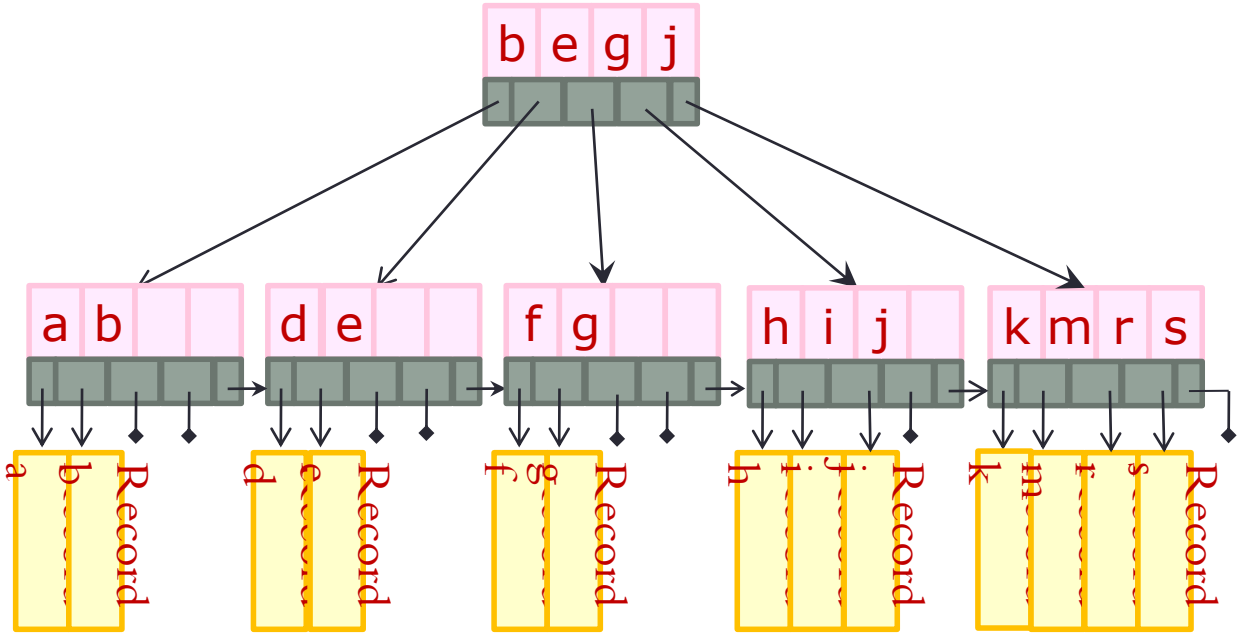
# Insertion into B*-tree

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

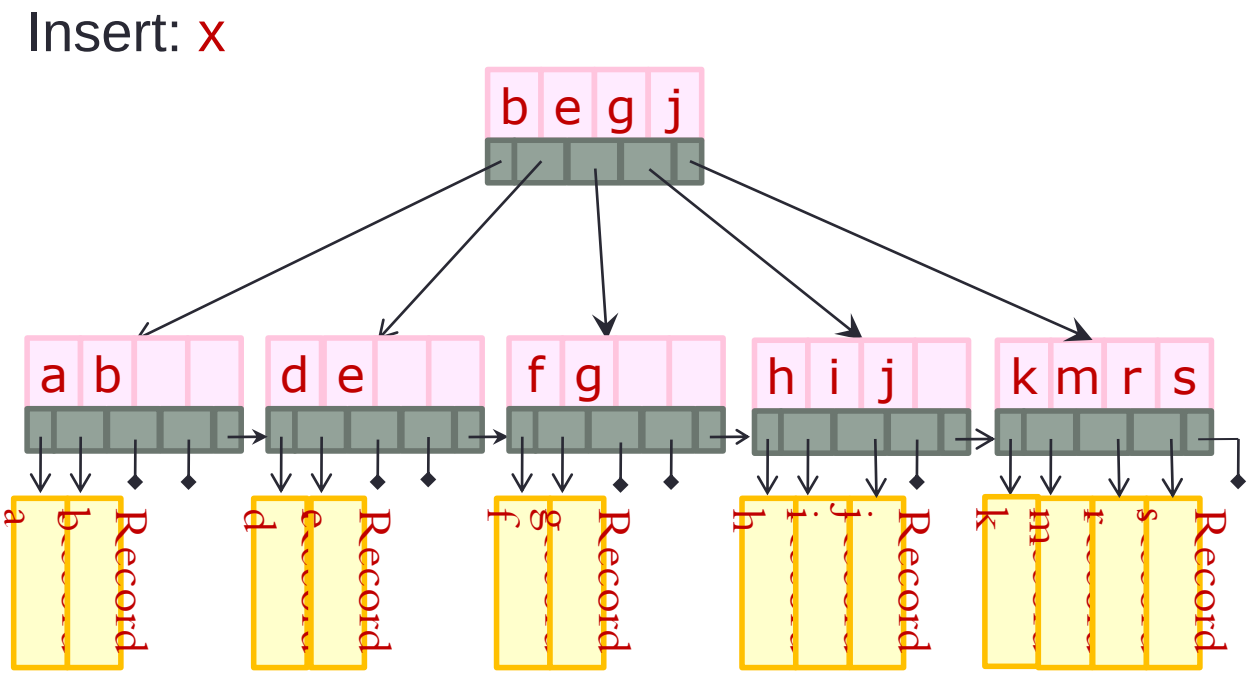


# Insertion into B*-tree

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



# Insertion into B*-tree

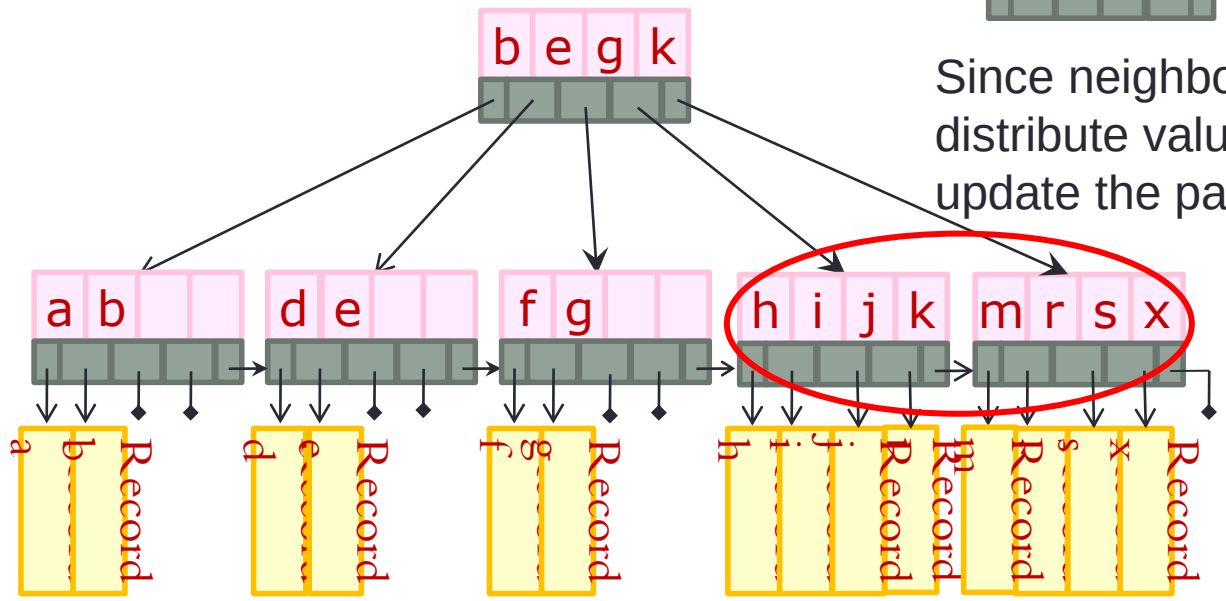


# Insertion into B*-tree

Insert: x

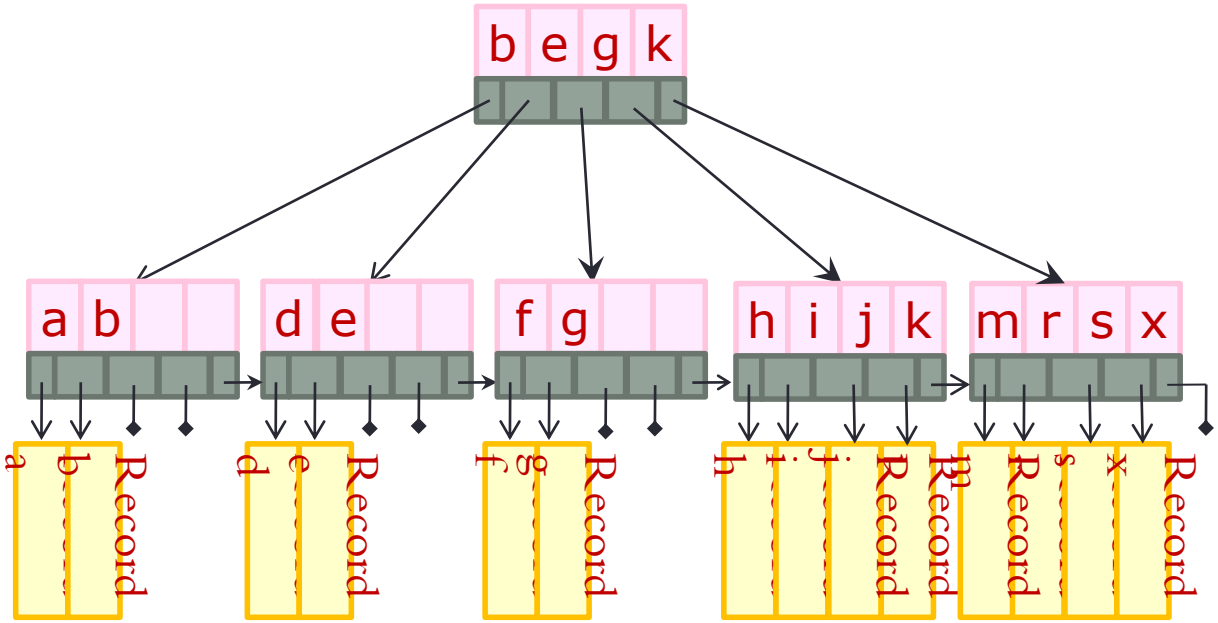
Insert of x causes an overflow of node:


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



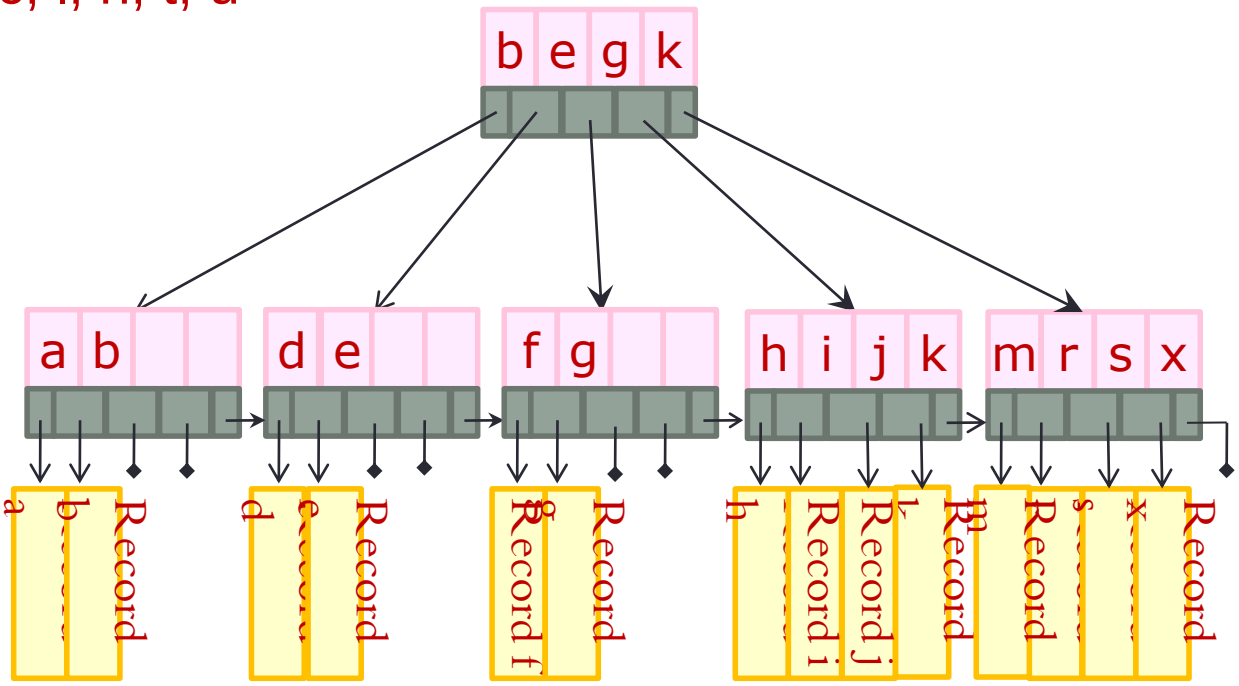
# Insertion into B*-tree

B*-tree after the insertion of x.



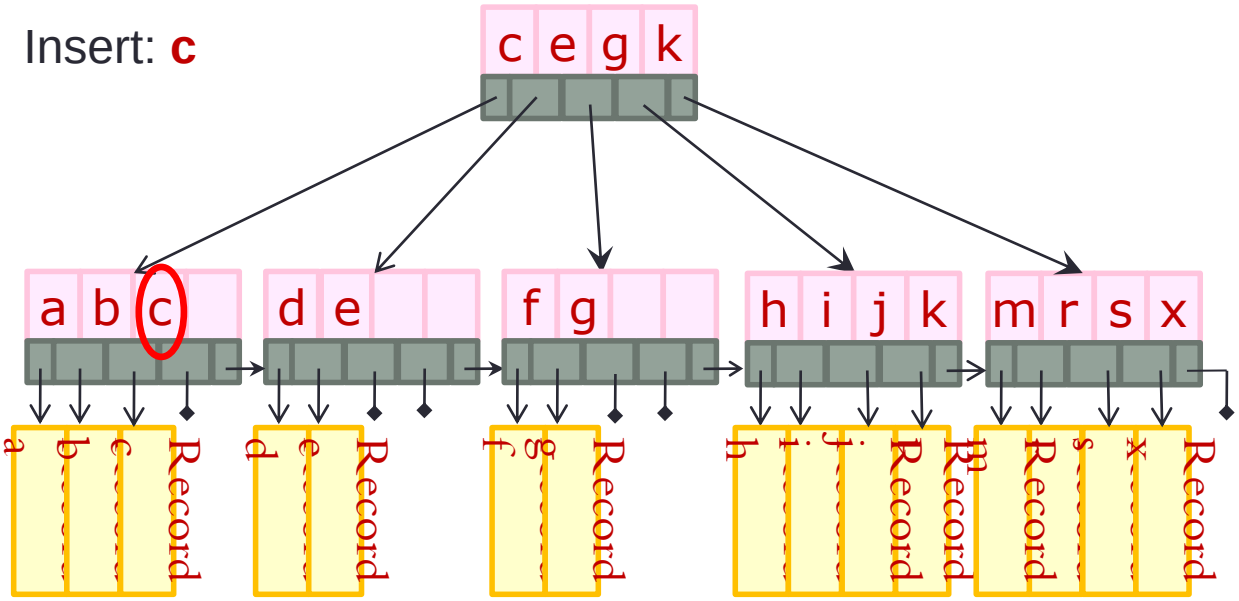
# Insertion into B*-tree

Insert: c, l, n, t, u



# Insertion into B*-tree

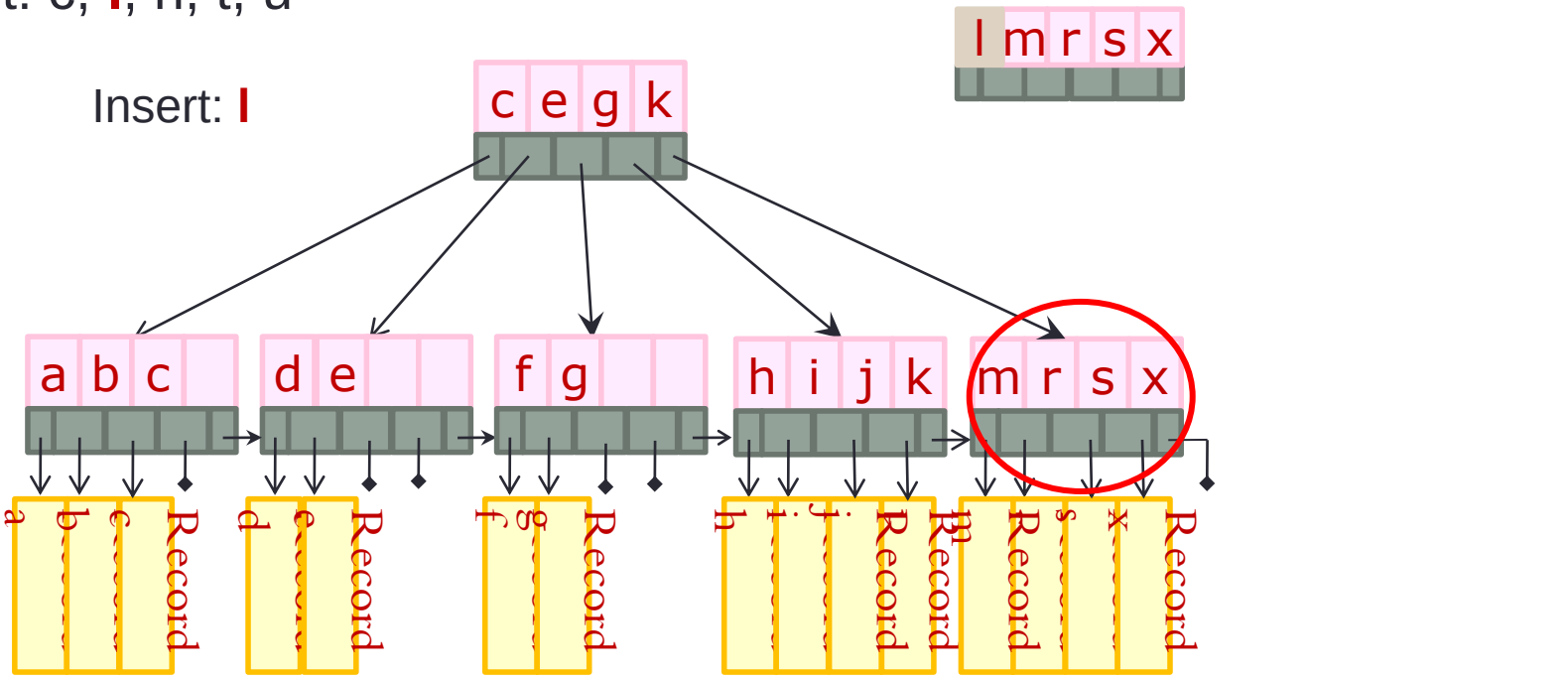
Insert: **c**, l, n, t, u



# Insertion into B*-tree

Insert: c, l, n, t, u

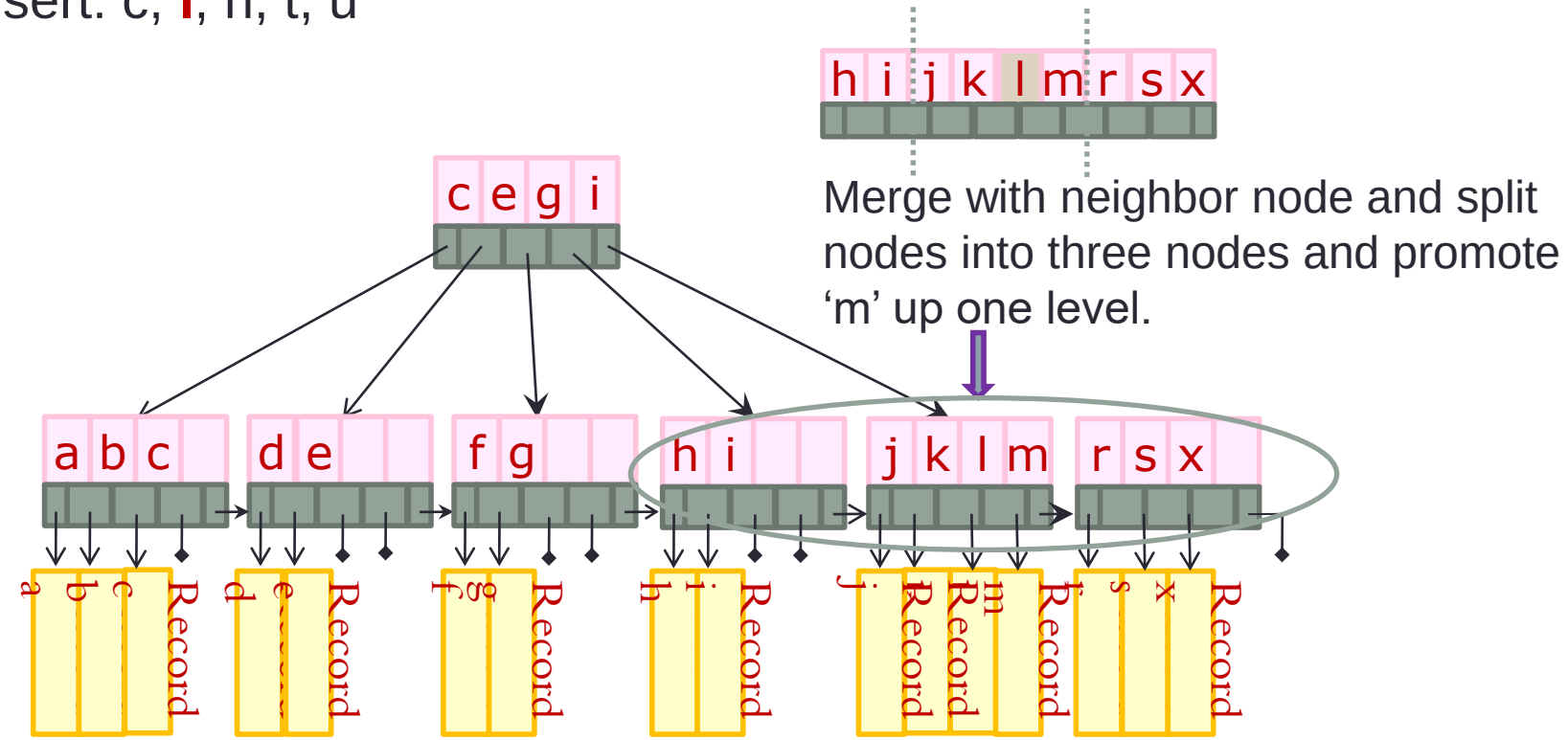
Insert of l causes an overflow of node:



# Insertion into B*-tree

Insert: c, l, n, t, u

Insert l causes an overflow of node:

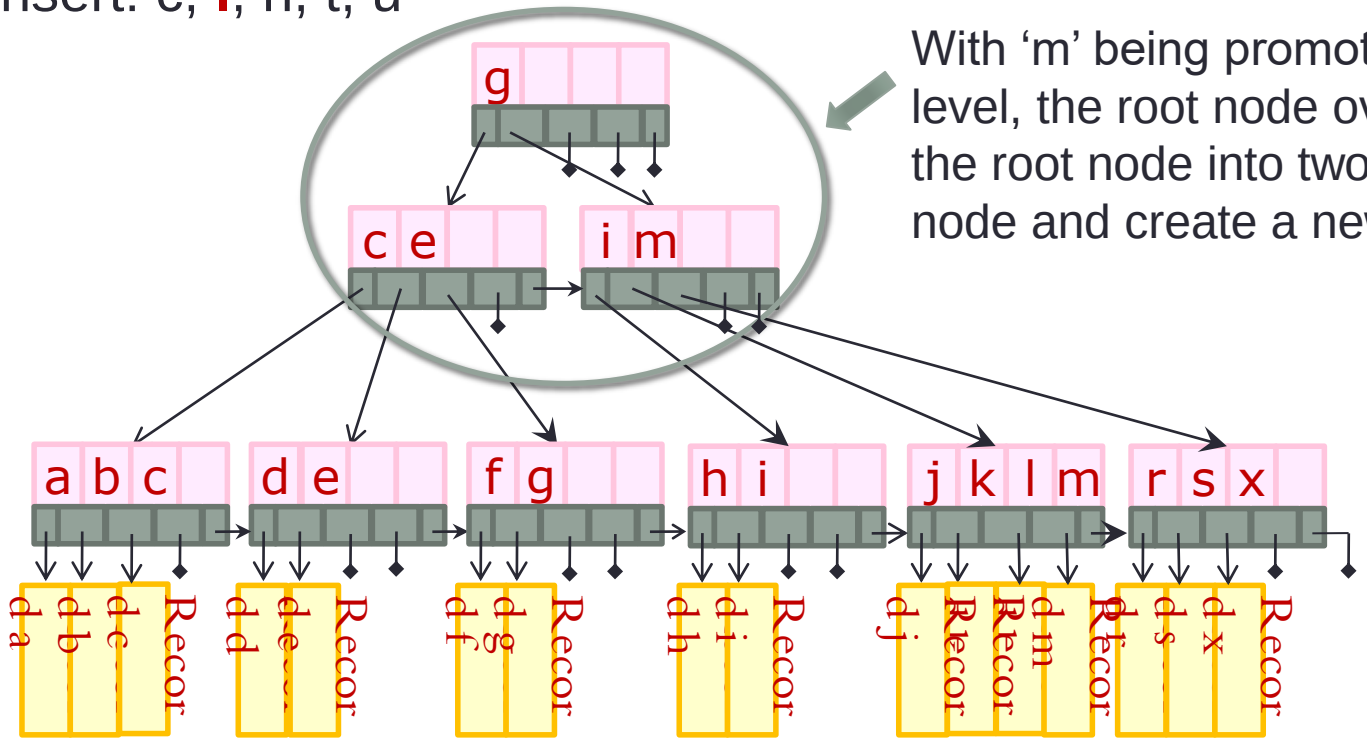


# Insertion into B*-tree

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.

# Deletion from B*-Tree

**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**

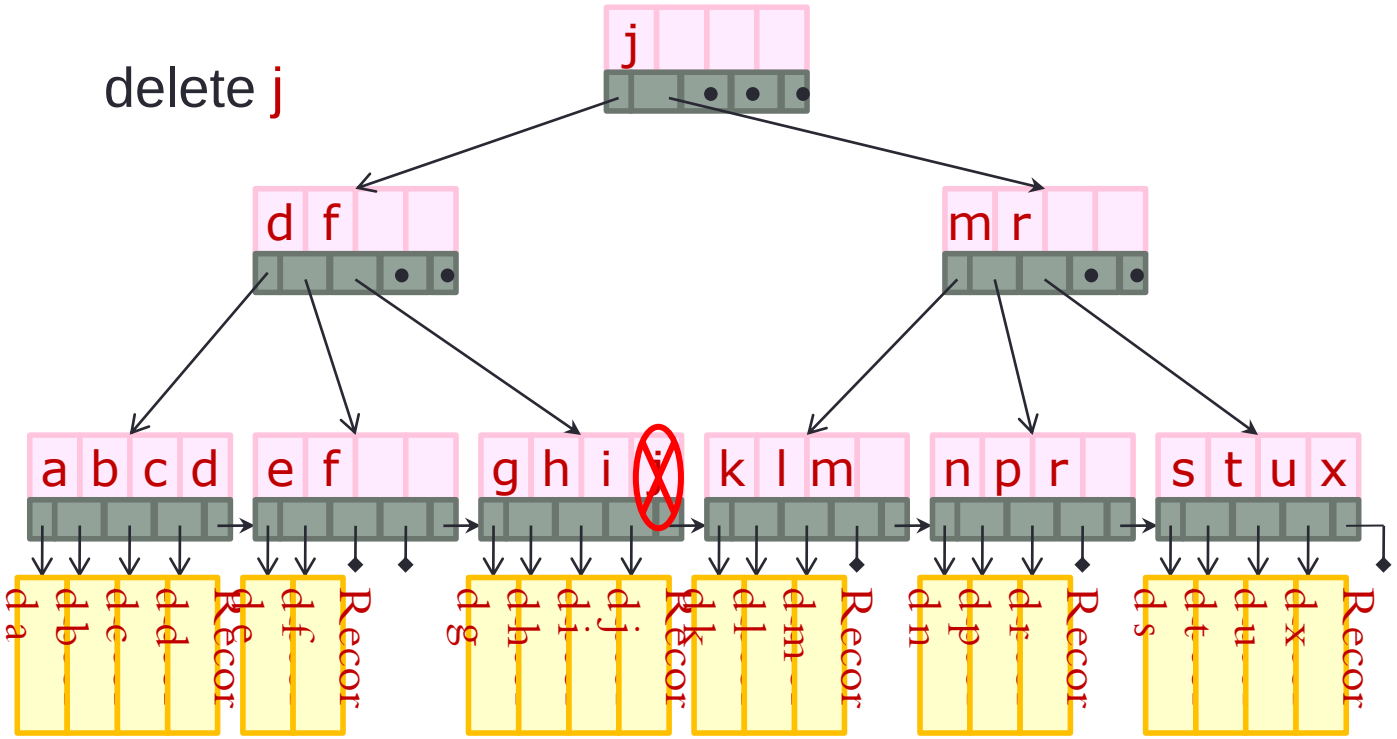
- Through parent
- Right node not less than left node

**Merge (contain too few entries)**

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

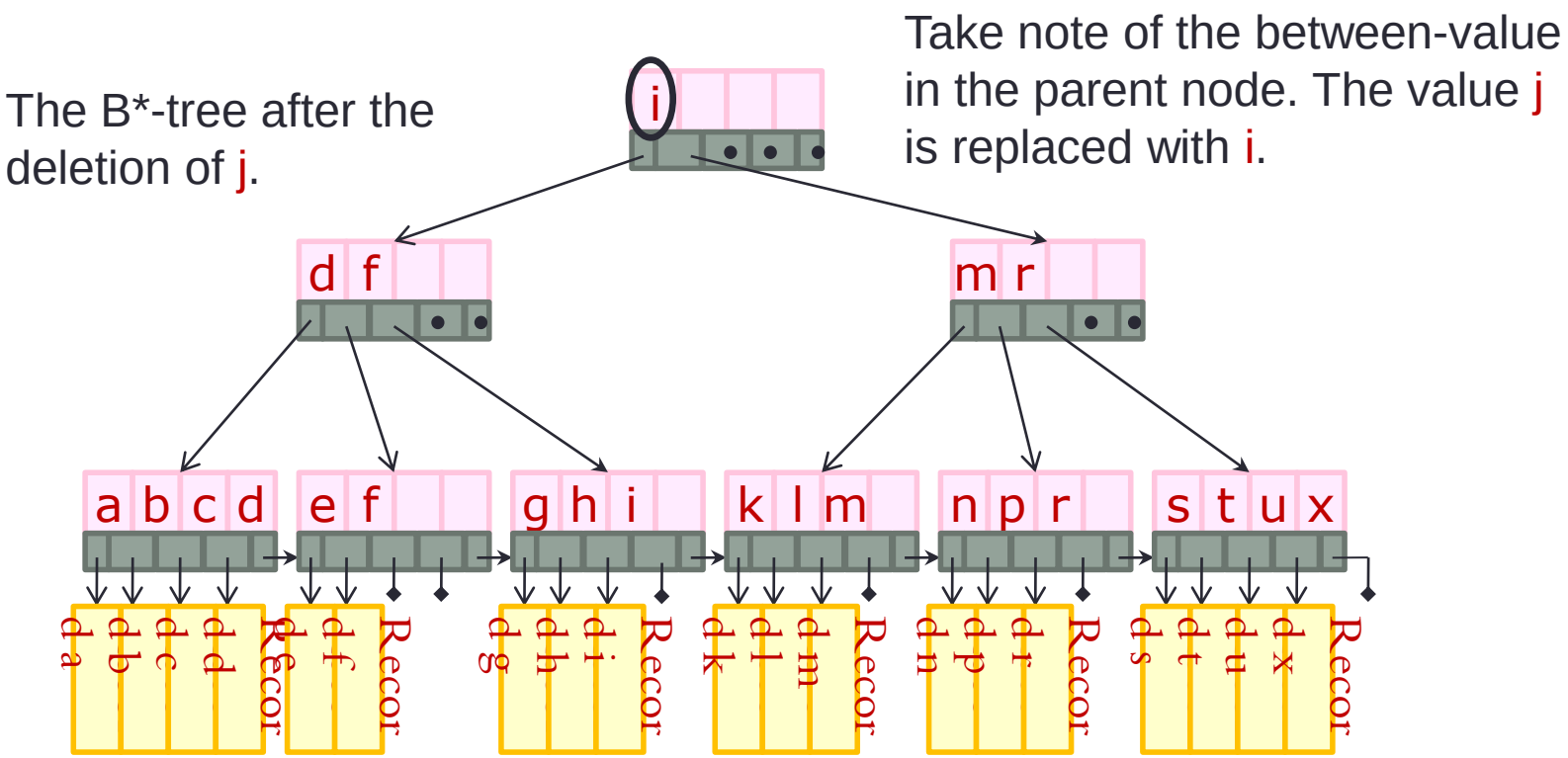
# Deletion from B*-Tree

- For example:



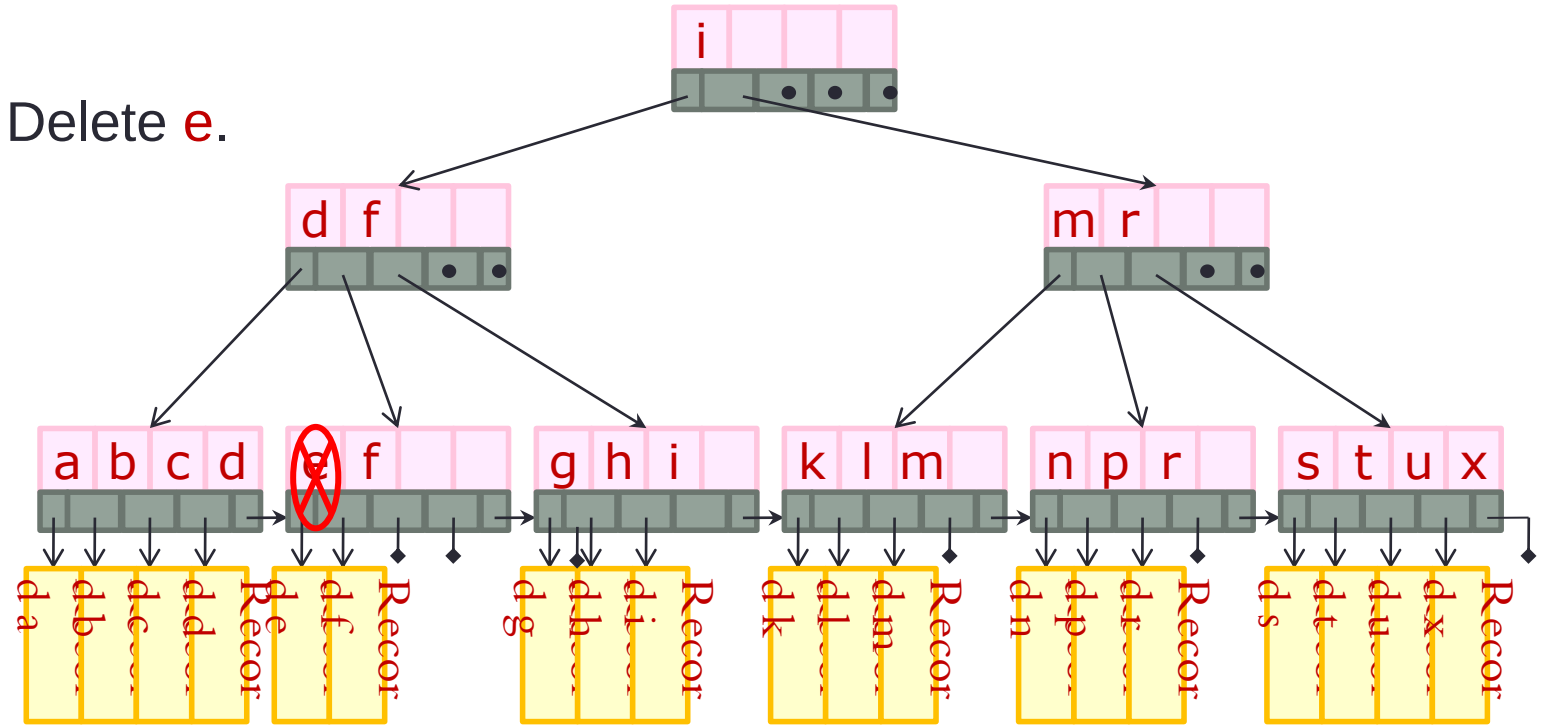
# Deletion from B*-Tree

- For example:



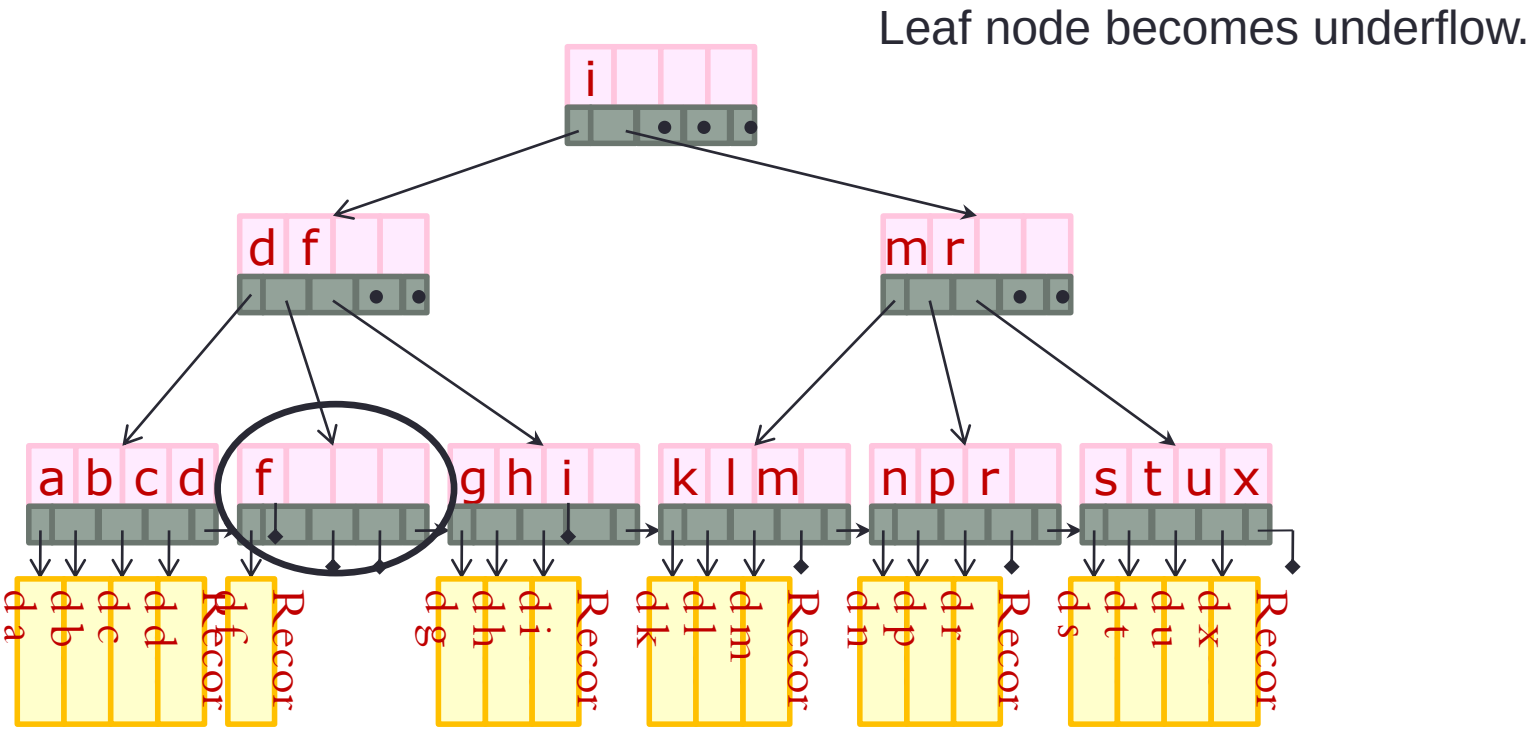
# Deletion from B*-Tree

- For example:



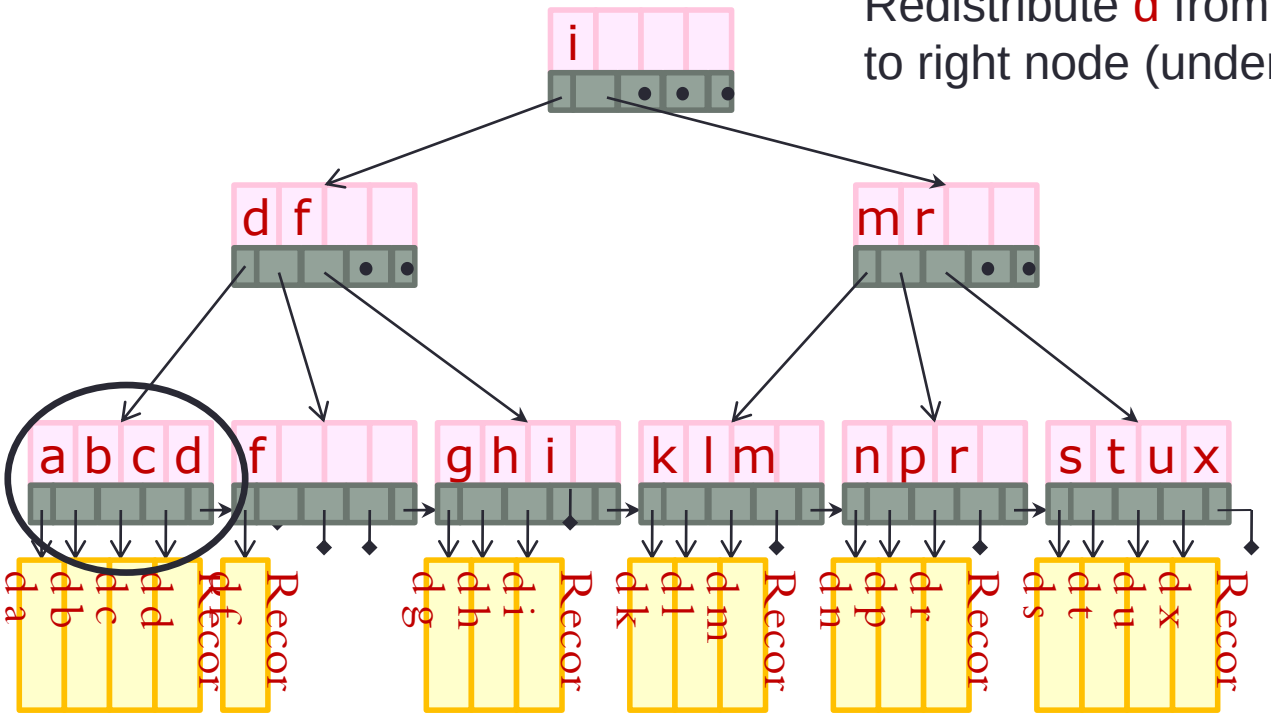
# Deletion from B*-Tree

- For example:



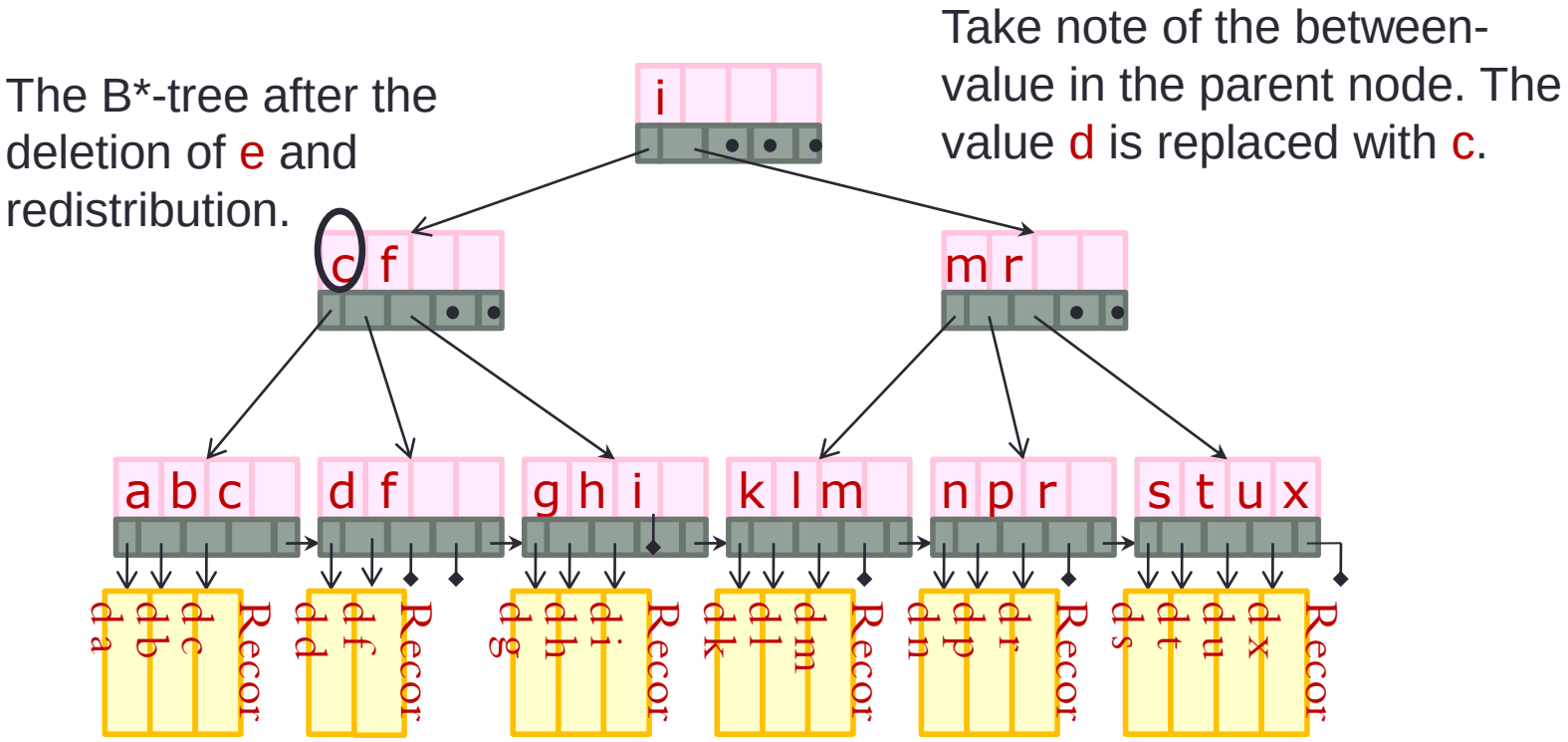
# Deletion from B*-Tree

- For example:



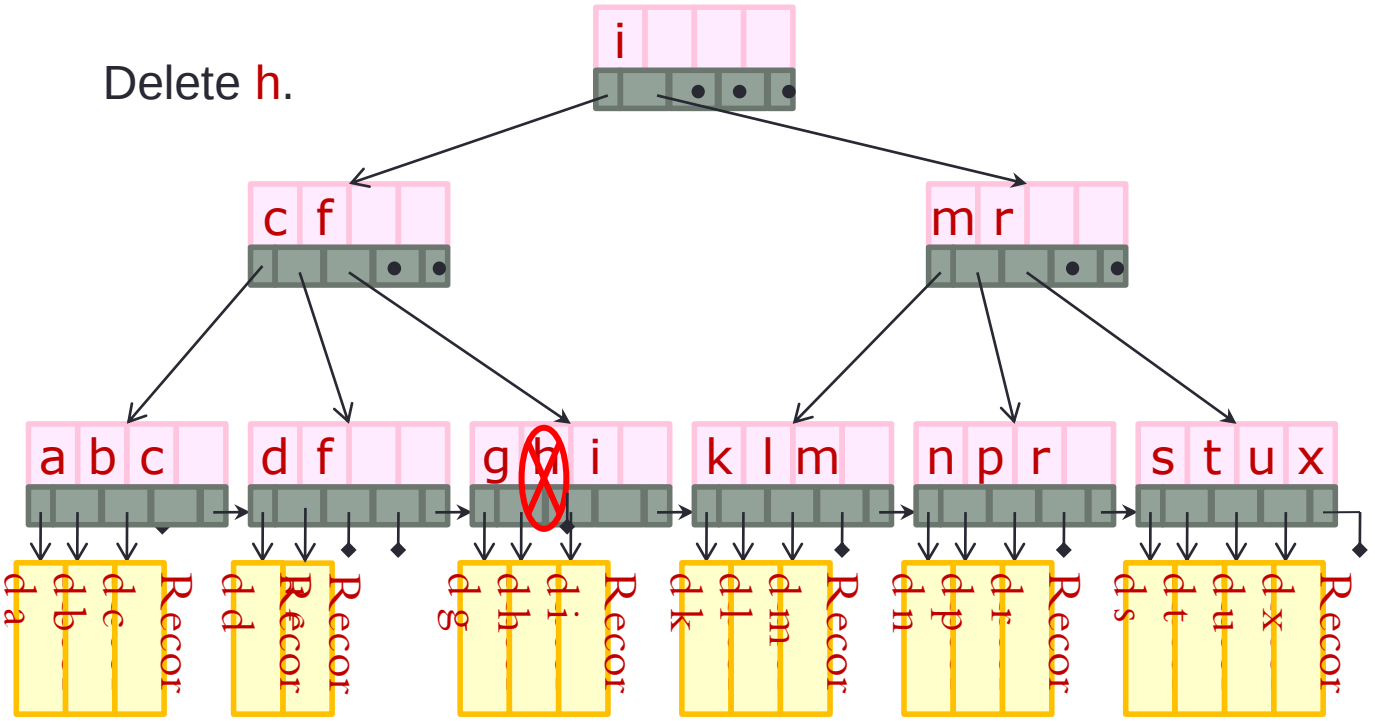
# Deletion from B*-Tree

- For example:



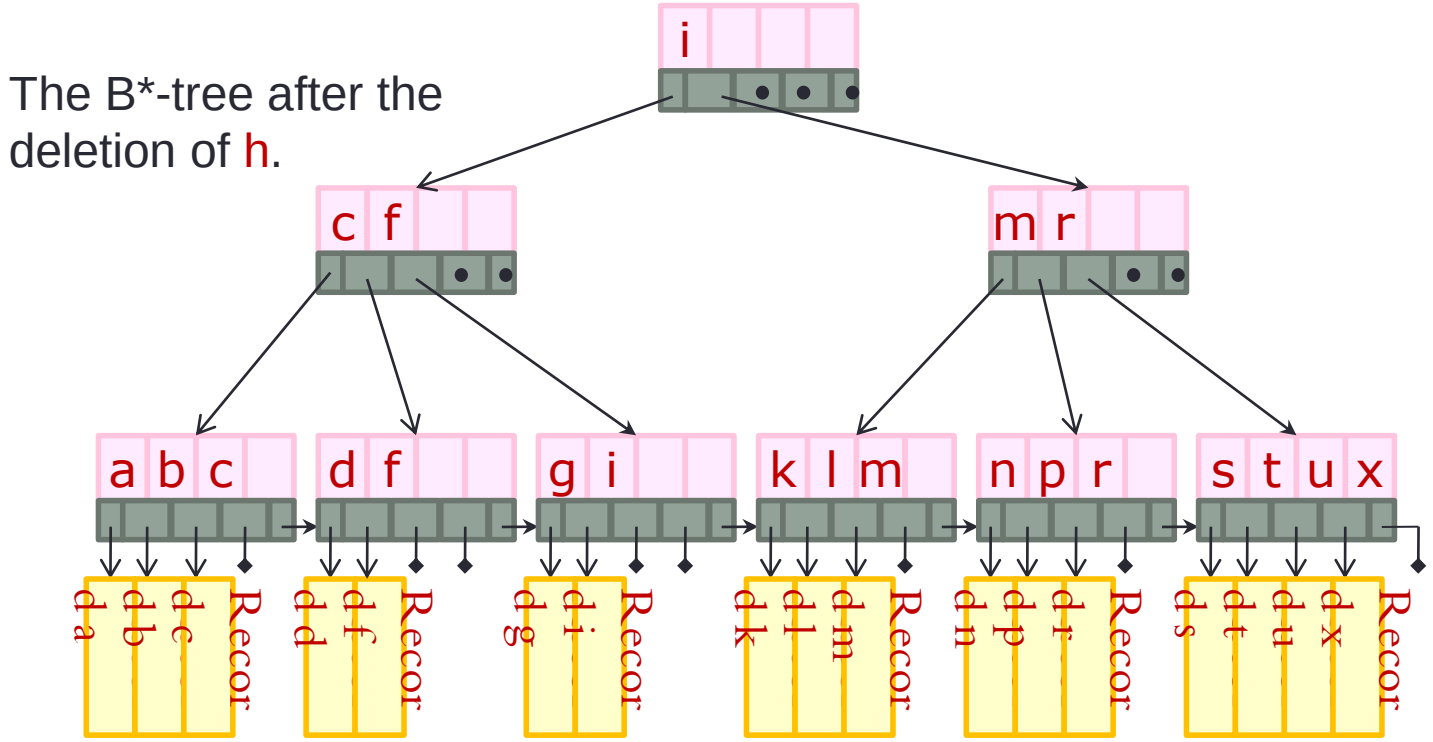
# Deletion from B*-Tree

- For example:



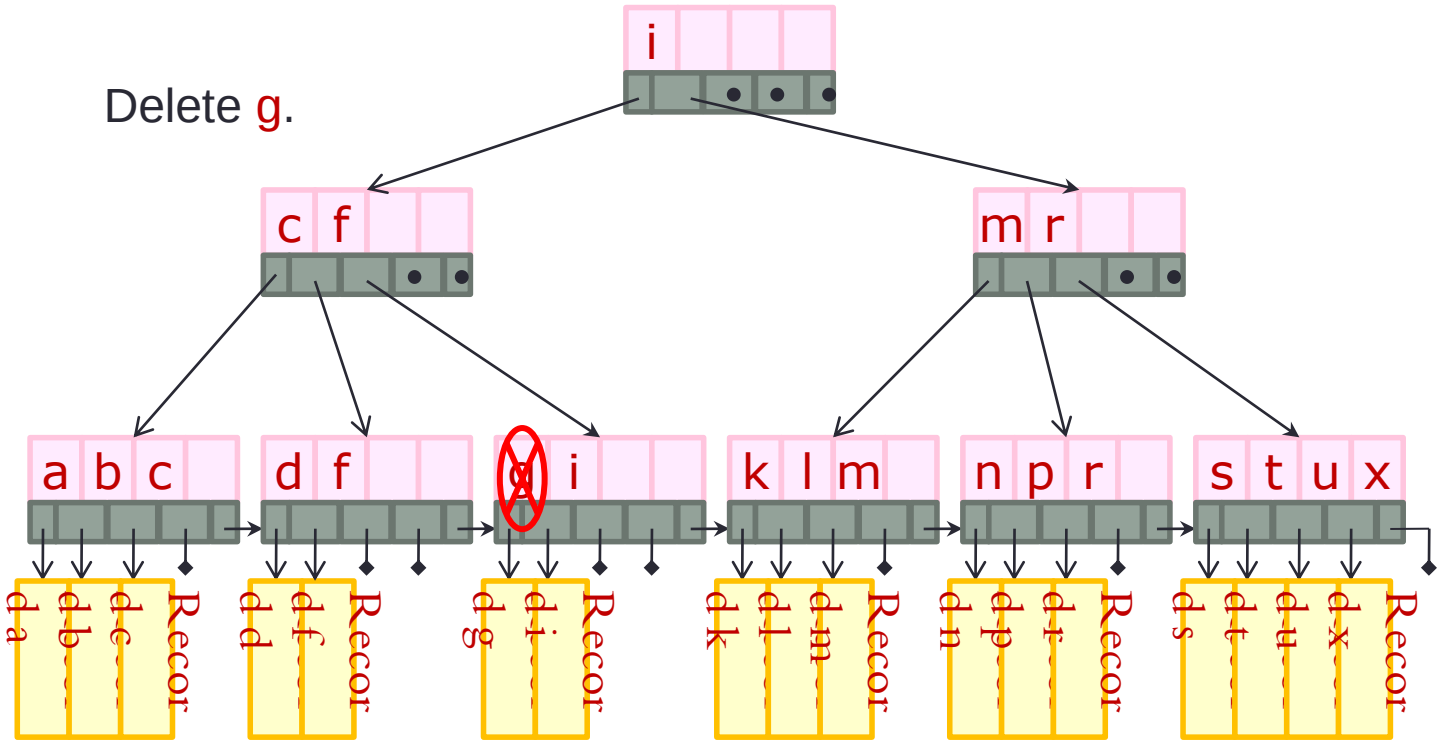
# Deletion from B*-Tree

- For example:



# Deletion from B*-Tree

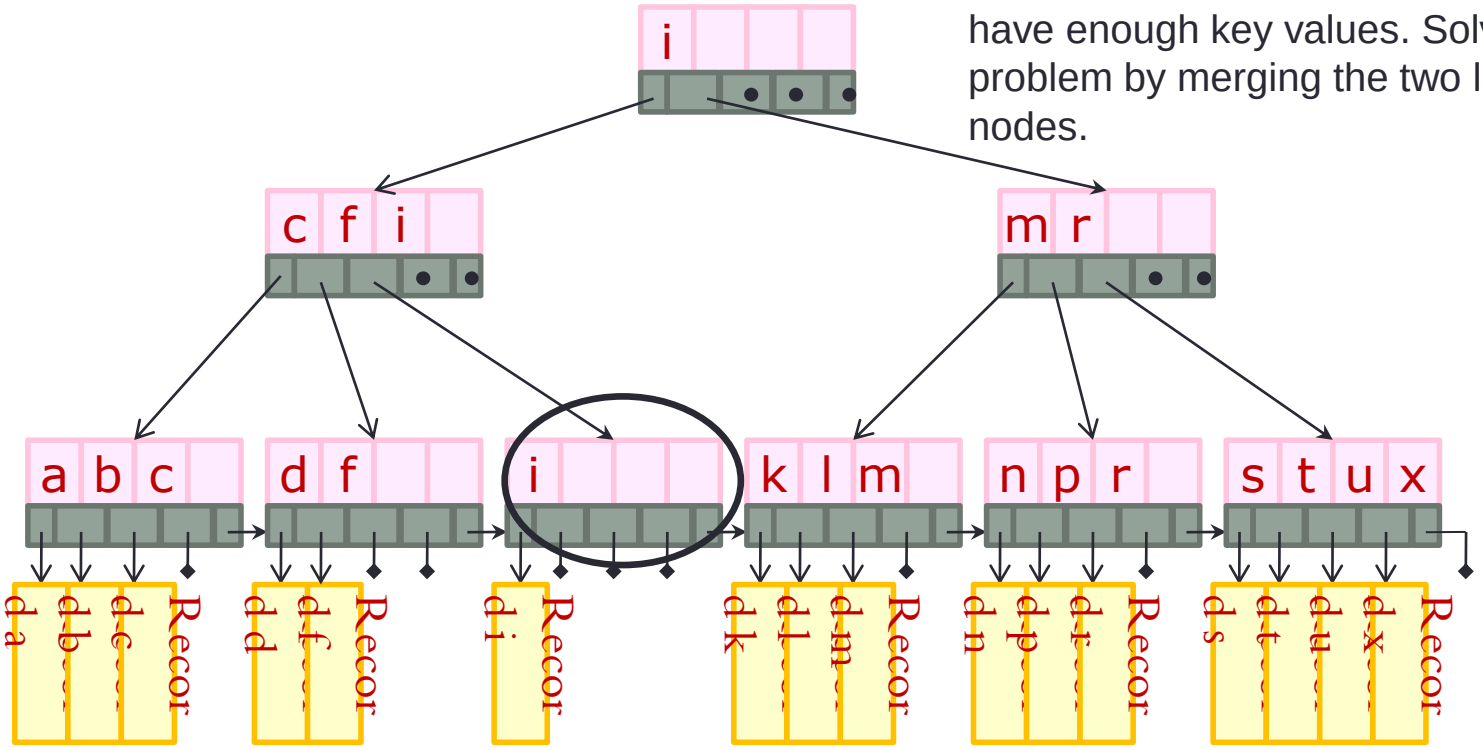
- For example:



# 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.

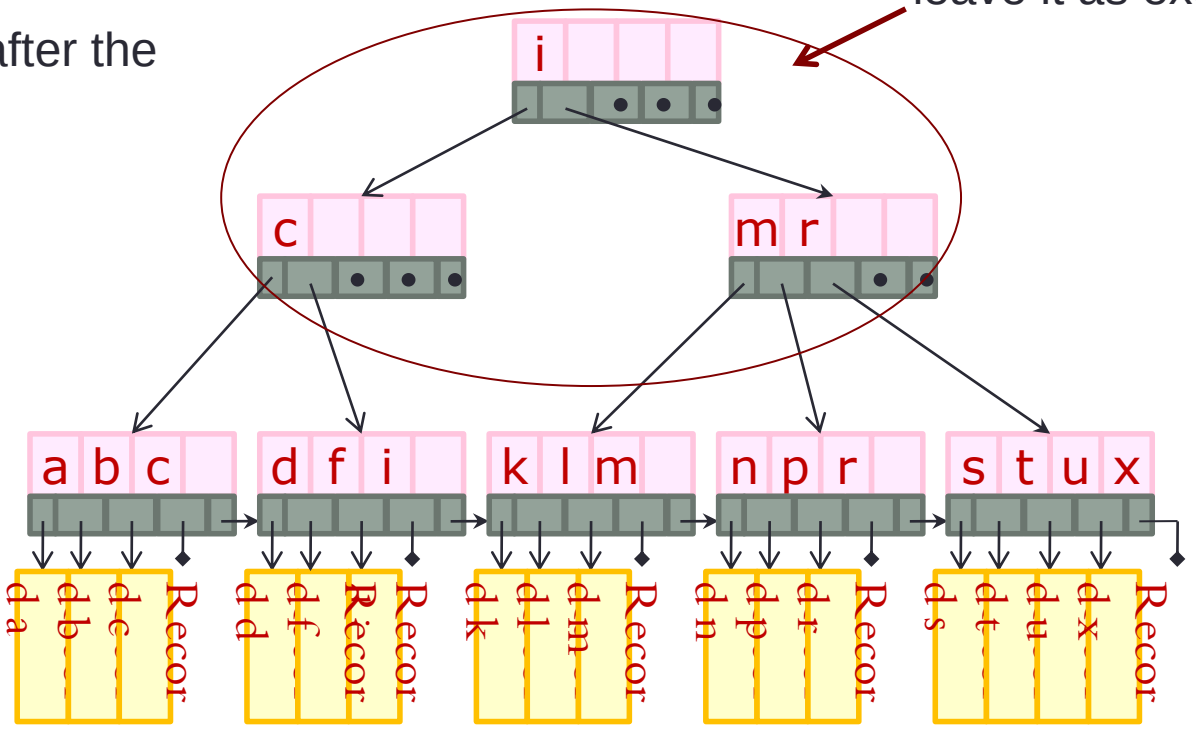


# Deletion from B*-Tree

- For example:

The B*-tree after the deletion of g.

Continue to merge and remove the root node. I leave it as exercise.



# TREAPS

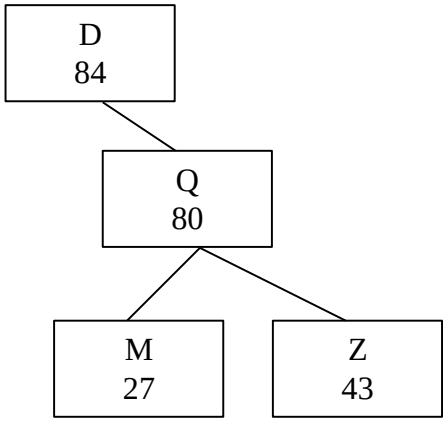
# Treaps

- Treaps
  - A hybrid data structure combining some of the properties of a binary search tree and a heap
  - Each node contains a value, a left and right child pointer and a (random) priority
  - AVL style single rotations are used to maintain the heap property and the binary search tree property at the same time
  - Every “empty” branch has a sentinel node attached.

# Treaps

- Treaps: sentinel nodes

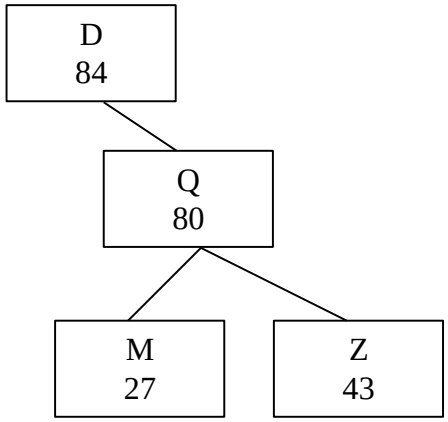
  This… is really……..

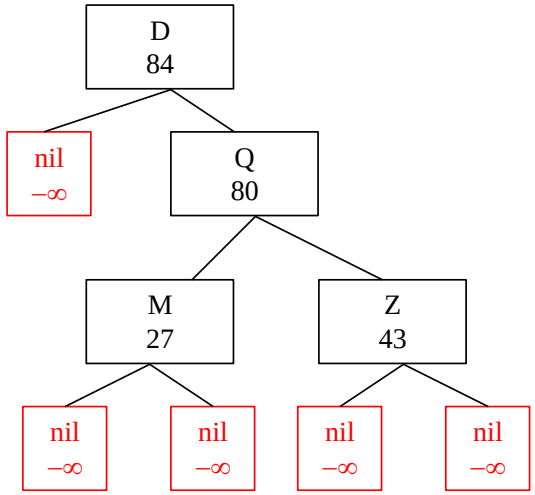


# Treaps

- Treaps: sentinel nodes

  This… is really…….. this



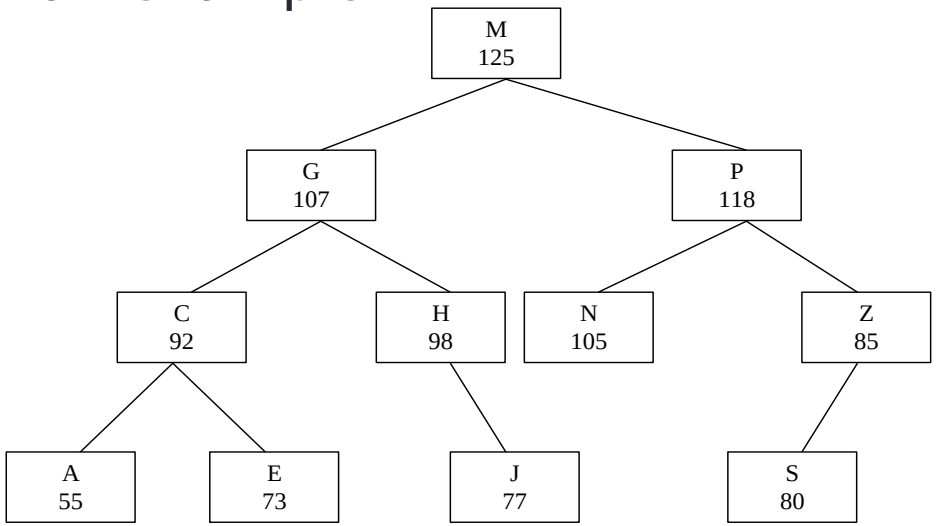

The sentinel nodes are needed for deletions

# Treaps

- Implementation
  - treap: ^treap_node
  - type treap_node: record
        value: stuff
        priority: integer
        left_child, right_child: ^treap_node

# Treaps

- Treaps: an example




  - Binary search tree on letters
  - Heap on priorities

# Treaps

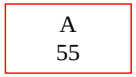
- Treaps: Insertion
  - function treap_insert(key, treap)

```
if treap = nil then
    priority = random_integer()
    treap = new treap_node(key, priority,nil,nil)
else if key < treap^.value then
    treap^.left=treap_insert(key, treap^.left)
    if (treap^.left).priority > treap^.priority then
        rotate_left_child(treap)
else if treap^.value < key then
    treap^.right=treap_insert(key, treap^.right)
    if (treap^.right).priority > treap^.priority then
        rotate_right_child(treap)
else
    ; // duplicate, do nothing
return treap;
```

# Treaps

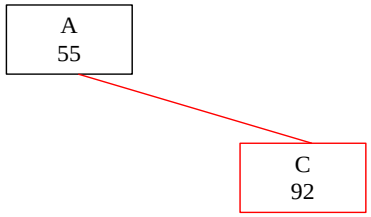
- Building a treap:
  - Insert “A”

in alphabetical order!



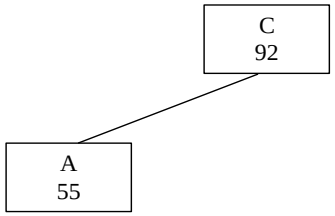
# Treaps

- Building a treap:
  - Insert “C”
  - Not a heap



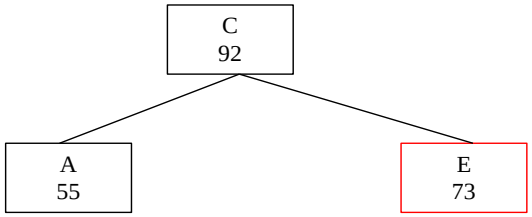
# Treaps

- Building a treap:
  - Rotate left



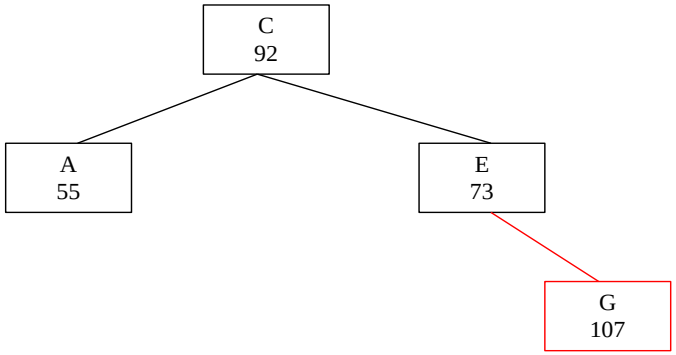
# Treaps

- Building a treap:
  - Insert “E”



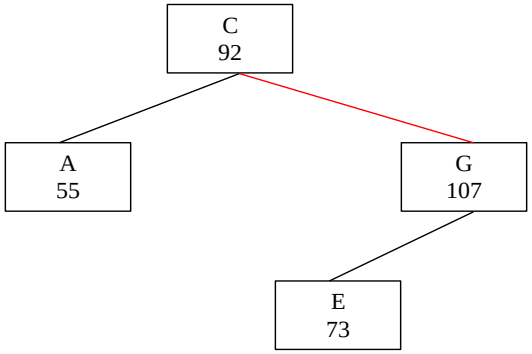
# Treaps

- Building a treap:
  - Insert “G”
  - Not a heap



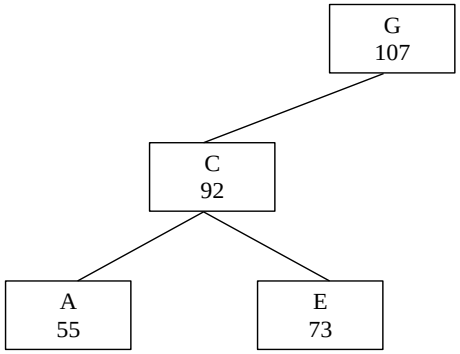
# Treaps

- Building a treap:
  - Rotate left
  - Still not a heap



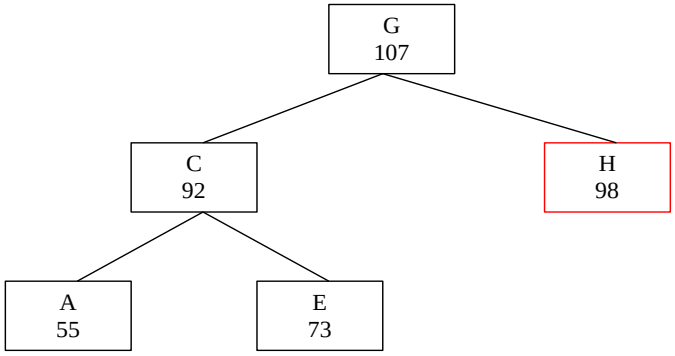
# Treaps

- Building a treap:
  - Rotate left



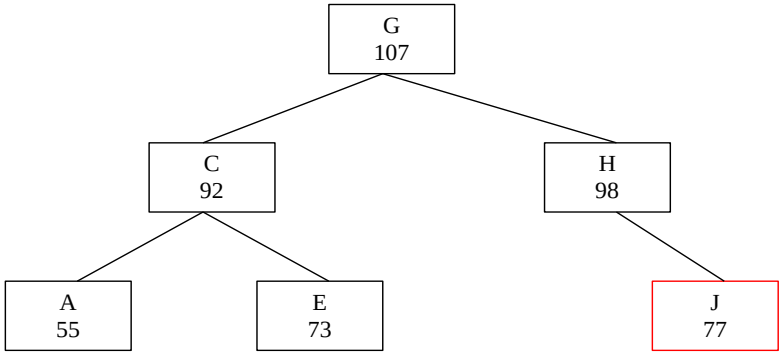
# Treaps

- Building a treap:
  - Insert “H”



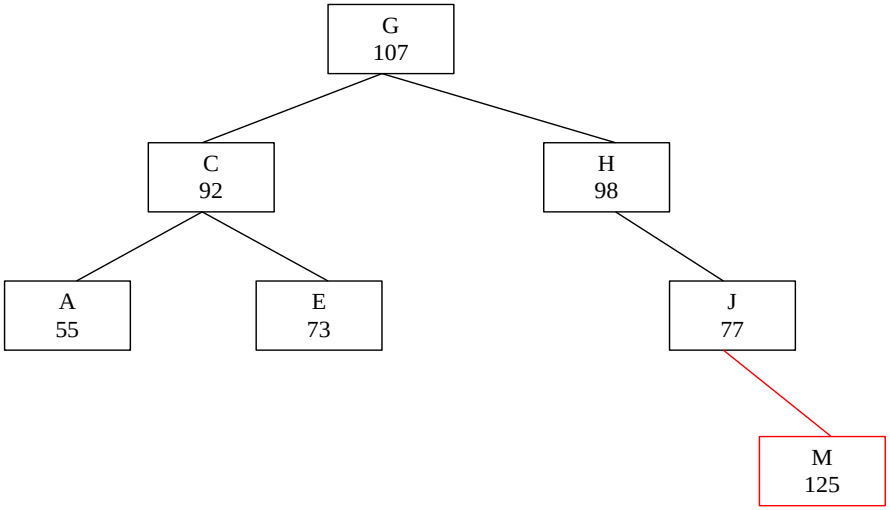
# Treaps

- Building a treap:
  - Insert "J"



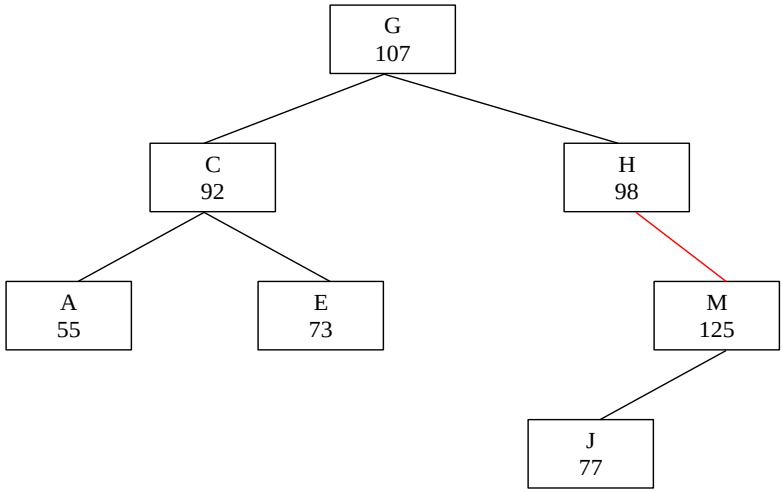
# Treaps

- Building a treap:
  - Insert “M”
  - Not a heap



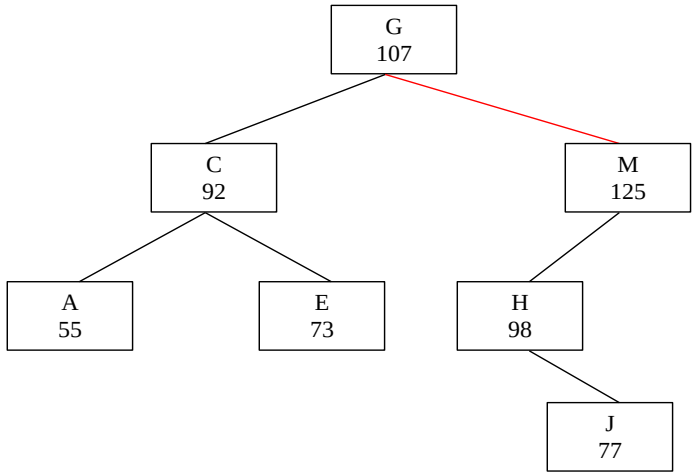
# Treaps

- Building a treap:
  - Rotate left
  - Still not a heap



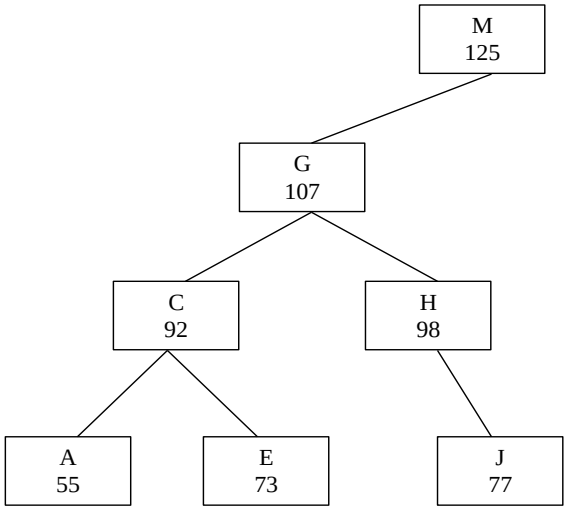
# Treaps

- Building a treap:
  - Rotate left
  - Still not a heap



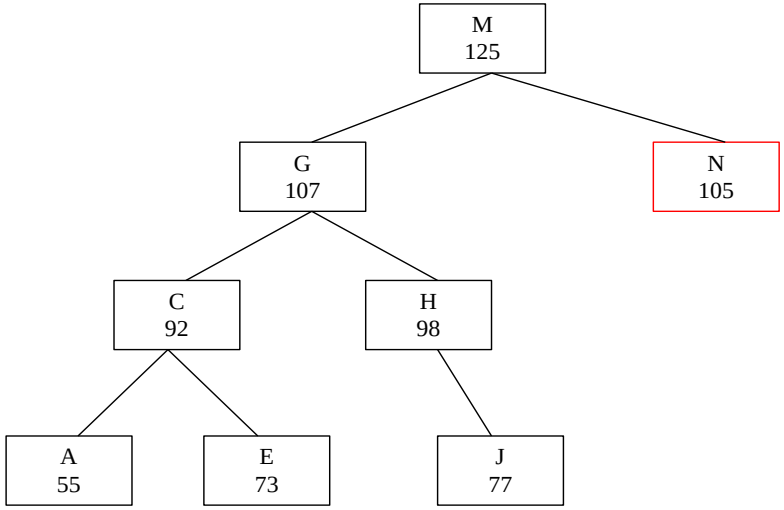
# Treaps

- Building a treap:
  - Rotate left



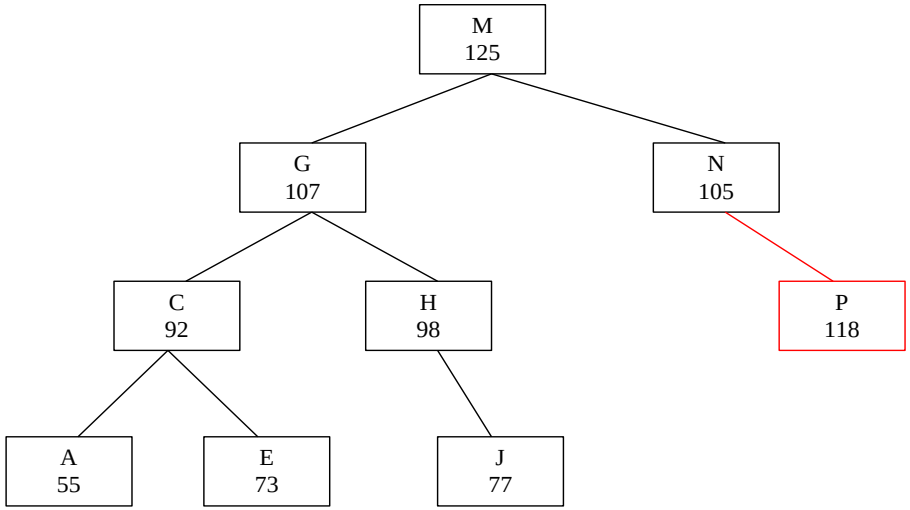
# Treaps

- Building a treap:
  - Insert “N”



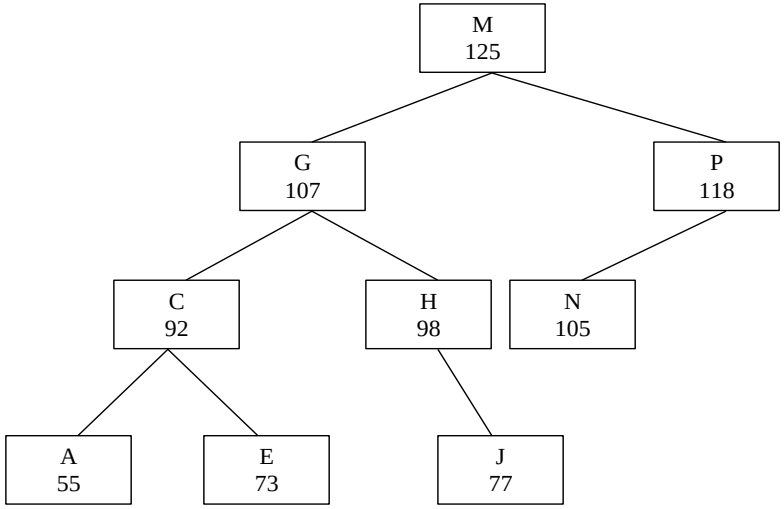
# Treaps

- Building a treap:
  - Insert “P”
  - Not a heap



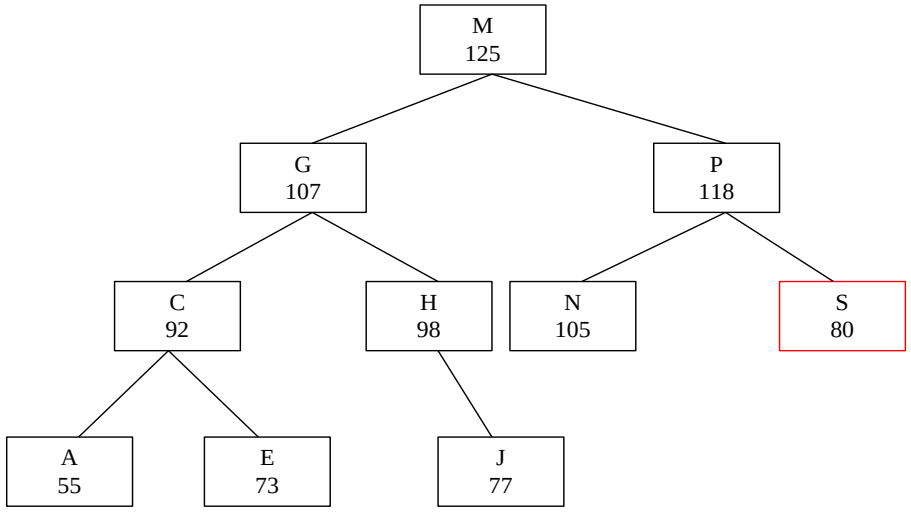
# Treaps

- Building a treap:
  - Rotate left



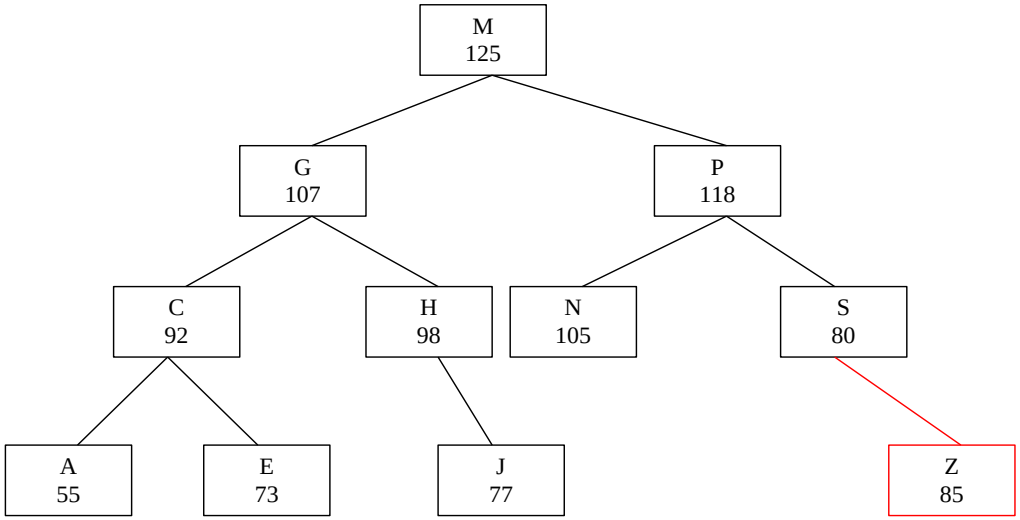
# Treaps

- Building a treap:
  - Insert “S”



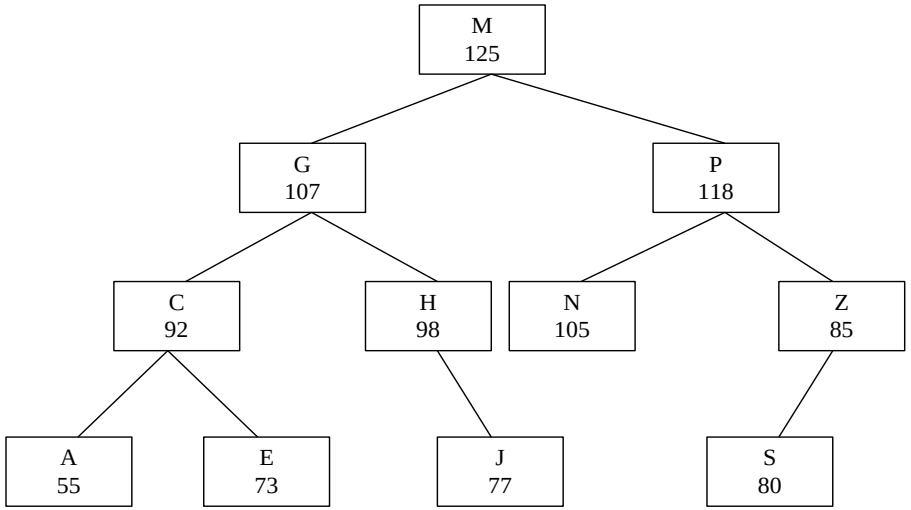
# Treaps

- Building a treap:
  - Insert “Z”
  - Not a heap



# Treaps

- Building a treap:
  - Rotate left



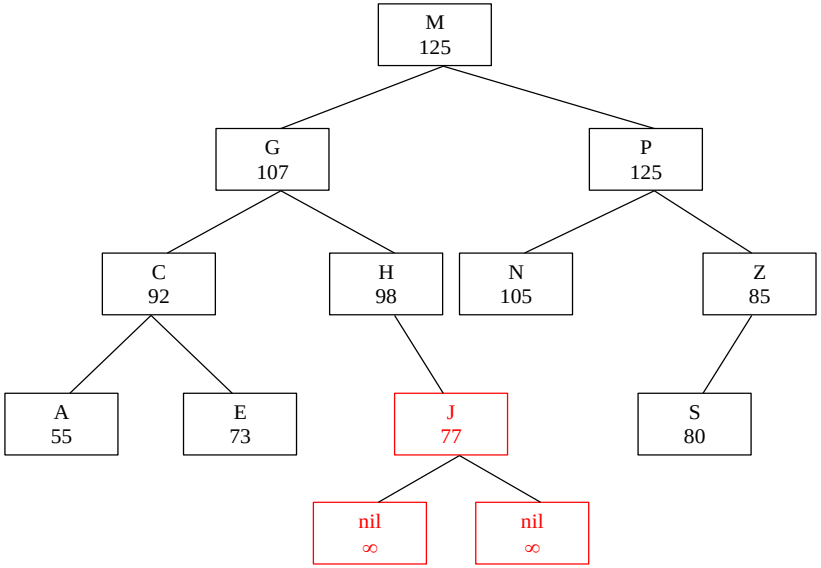
# Treaps

- Treaps: Deletion
  - function treap_delete(key, treap)

```
if treap ≠ nil then
    if key < treap^.value then
        treap^.left=treap_delete(key, treap^.left)
    else if treap^.value < key then
        treap^.right=treap_delete(key, treap^.right)
    else
        if  (treap^.left)^.priority > (treap^.right)^.priority then
            rotate_left_child(treap)
        else
            rotate_right_child(treap)
        if treap ≠ nil then
            treap=treap_delete(key, treap)
        else
            delete treap^.left
            treap^.left = nil
    return treap
```

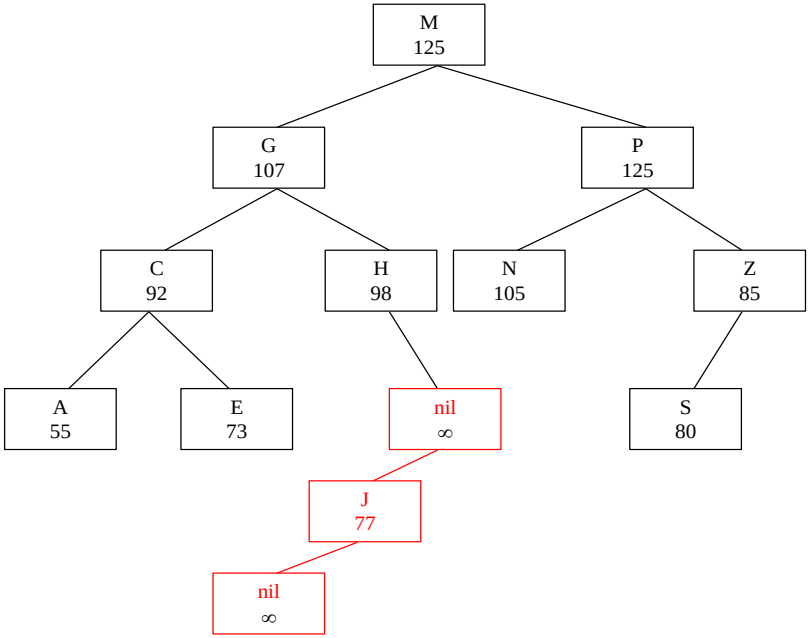
# Treaps

- Treap deletion: an example
  - Delete "J"



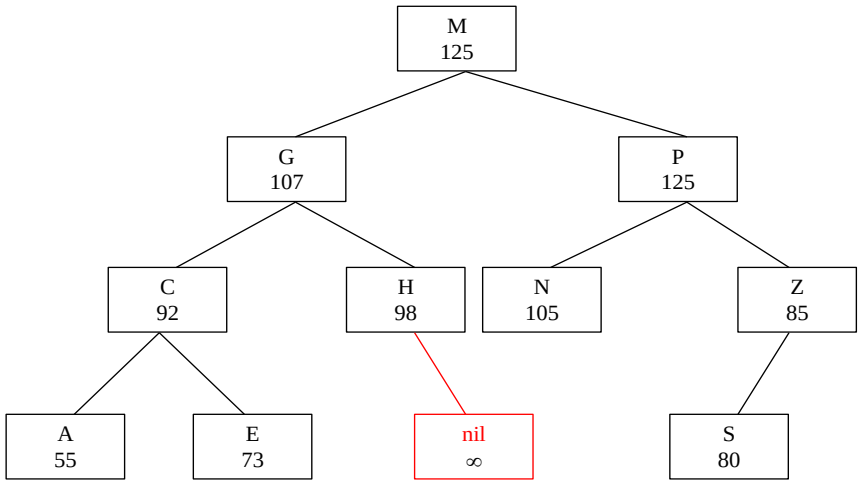
# Treaps

- Treap deletion: an example
  - Rotate right



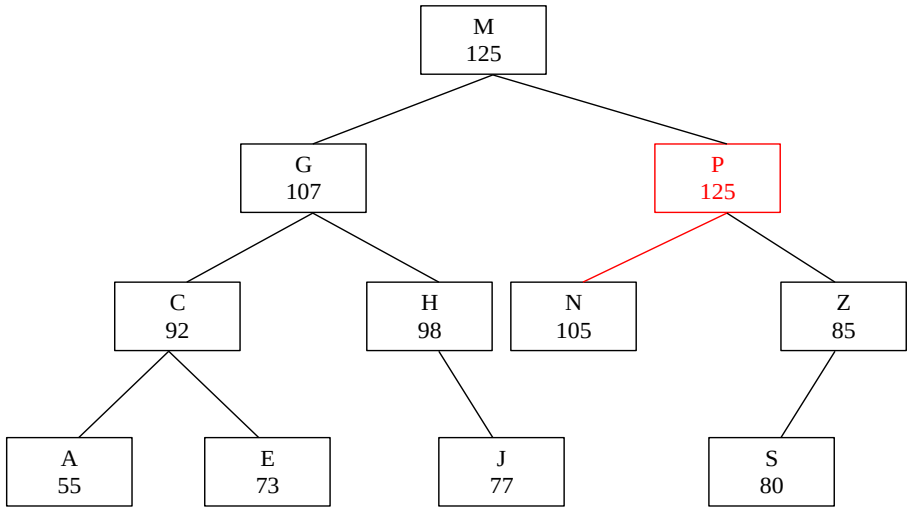
# Treaps

- Treap deletion: an example
  - Delete left sub tree



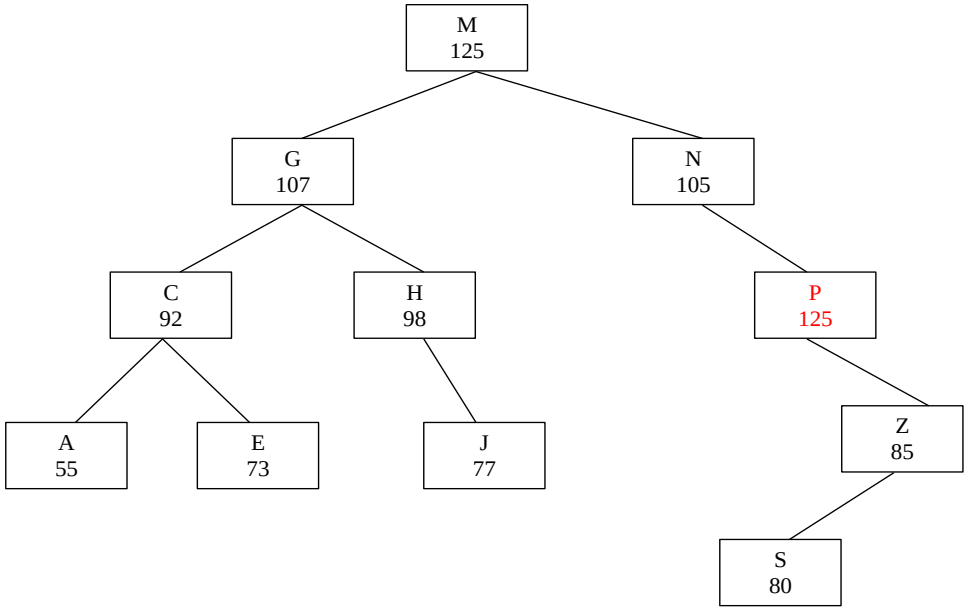
# Treaps

- Treap deletion: an example
  - Delete “P”



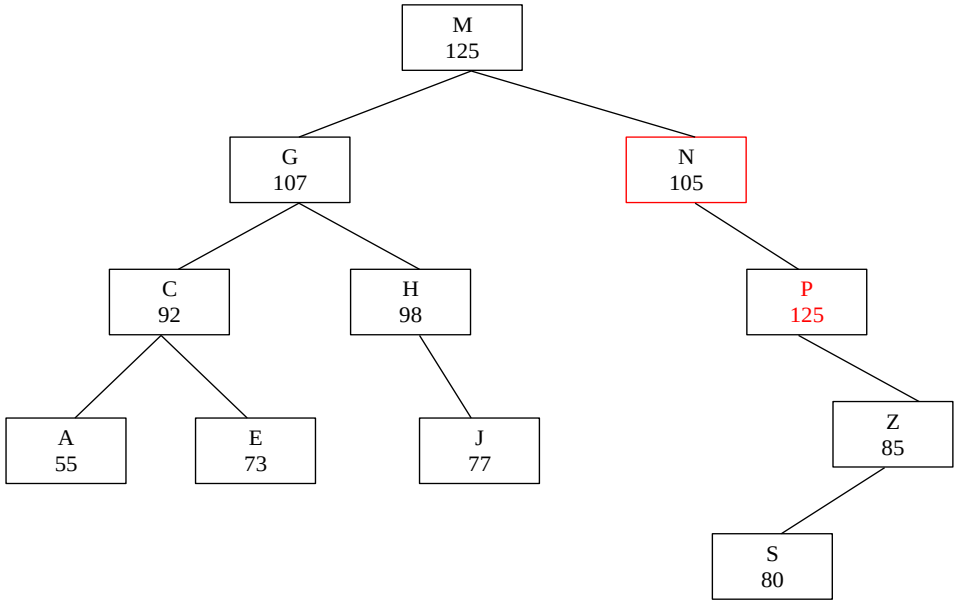
# Treaps

- Treap deletion: an example
  - Rotate right



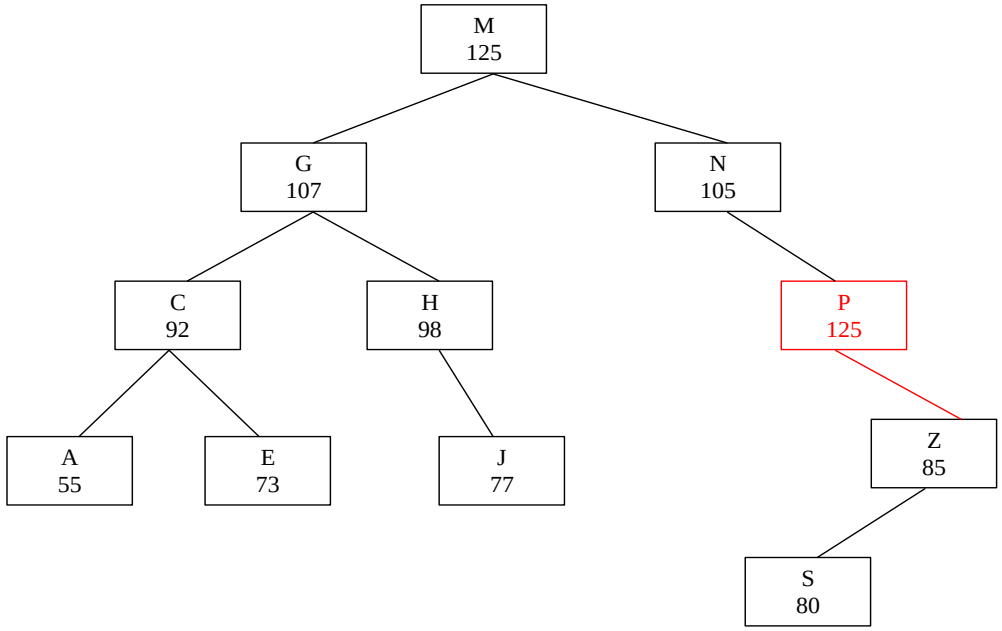
# Treaps

- Treap deletion: an example
  - Repeat from here



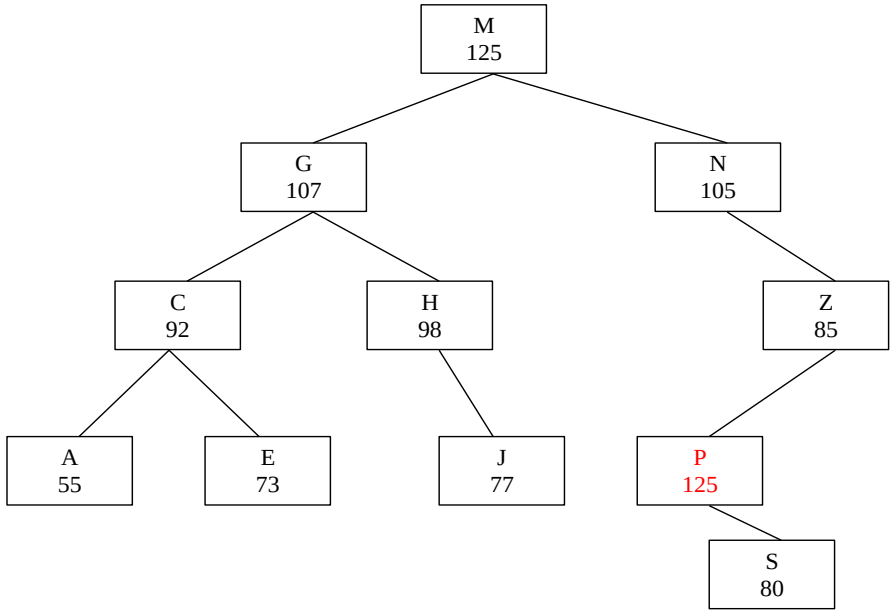
# Treaps

- Treap deletion: an example
  - Delete “P”



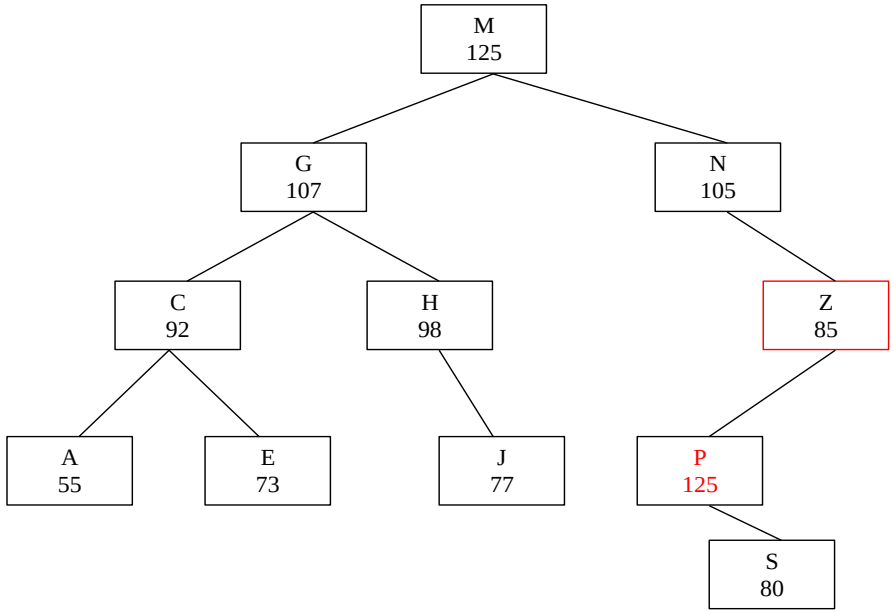
# Treaps

- Treap deletion: an example
  - Rotate left



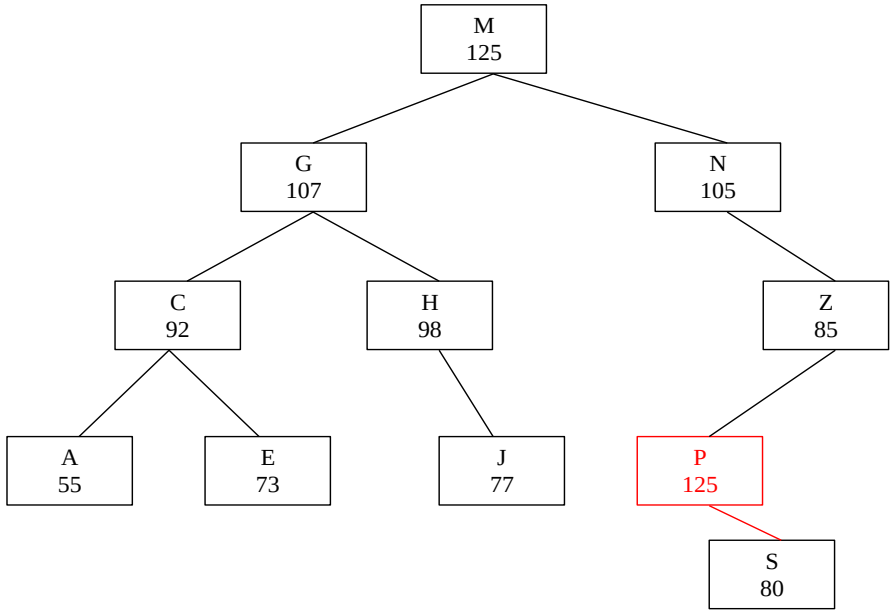
# Treaps

- Treap deletion: an example
  - Repeat from here



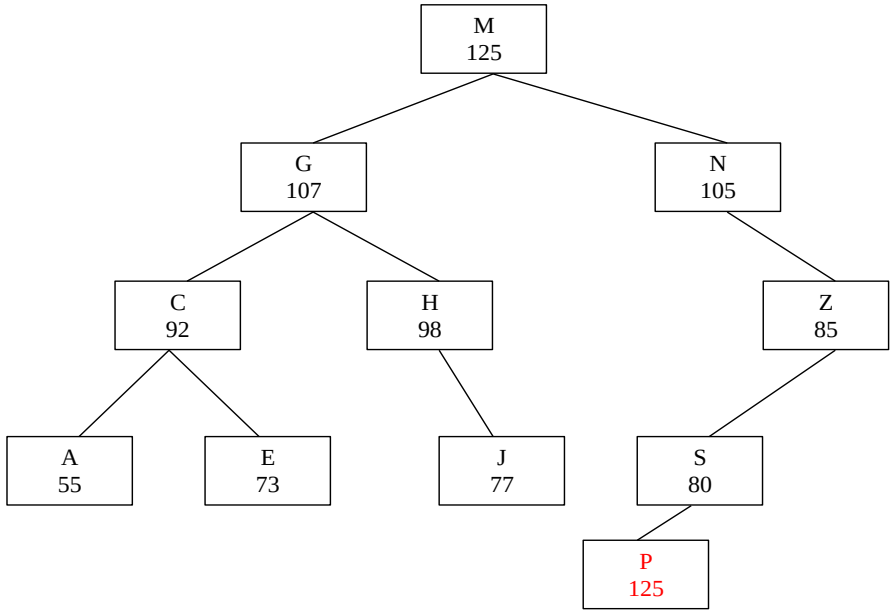
# Treaps

- Treap deletion: an example
  - Delete "P"



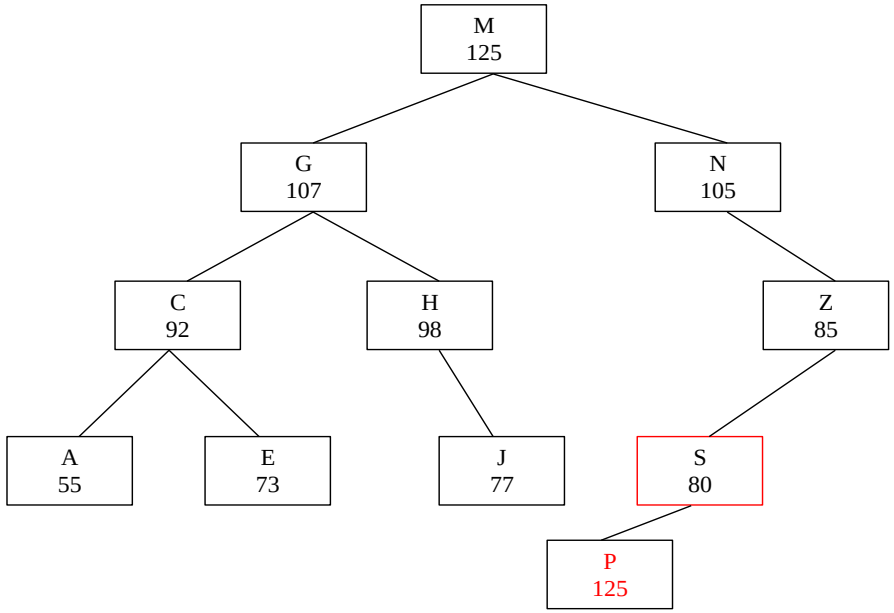
# Treaps

- Treap deletion: an example
  - Rotate left



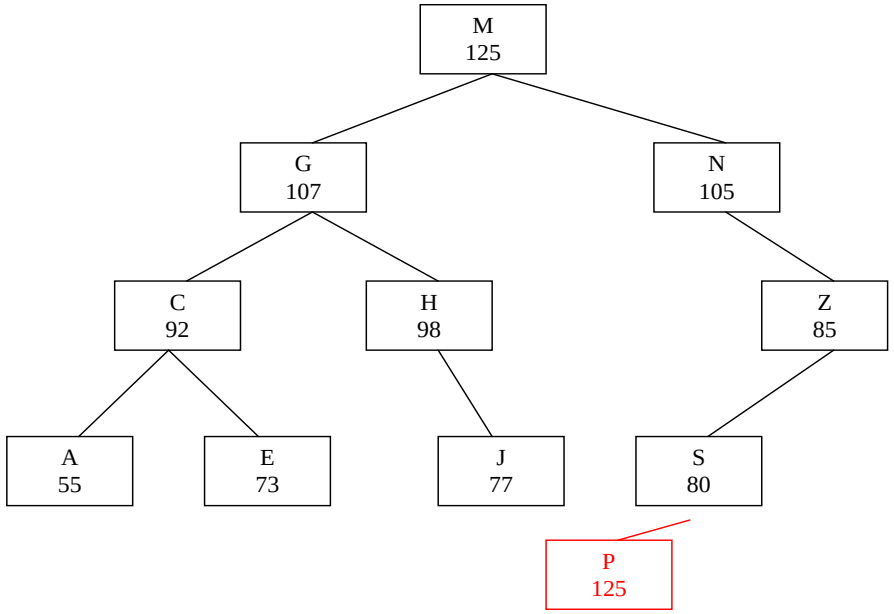
# Treaps

- Treap deletion: an example
  - Repeat from here



# Treaps

- Treap deletion: an example
  - Drop the leaf



# Treaps

- Treaps: Efficiency
  - Height of tree is O(lg *n*) at best.
  - Searching:
    - Each node checked takes O(1)
    - Search takes O(lg *n*)
  - Insertion:
    - Search for insertion point takes O(lg *n*)
    - Insertion takes O(1) with possible O(1) rotation
    - Insertion takes O(lg *n*)
  - Deletion
    - Search for node takes O(lg *n*)
    - Rotation of node down to leaf takes O(lg *n*)
    - Deletion takes O(lg *n*)