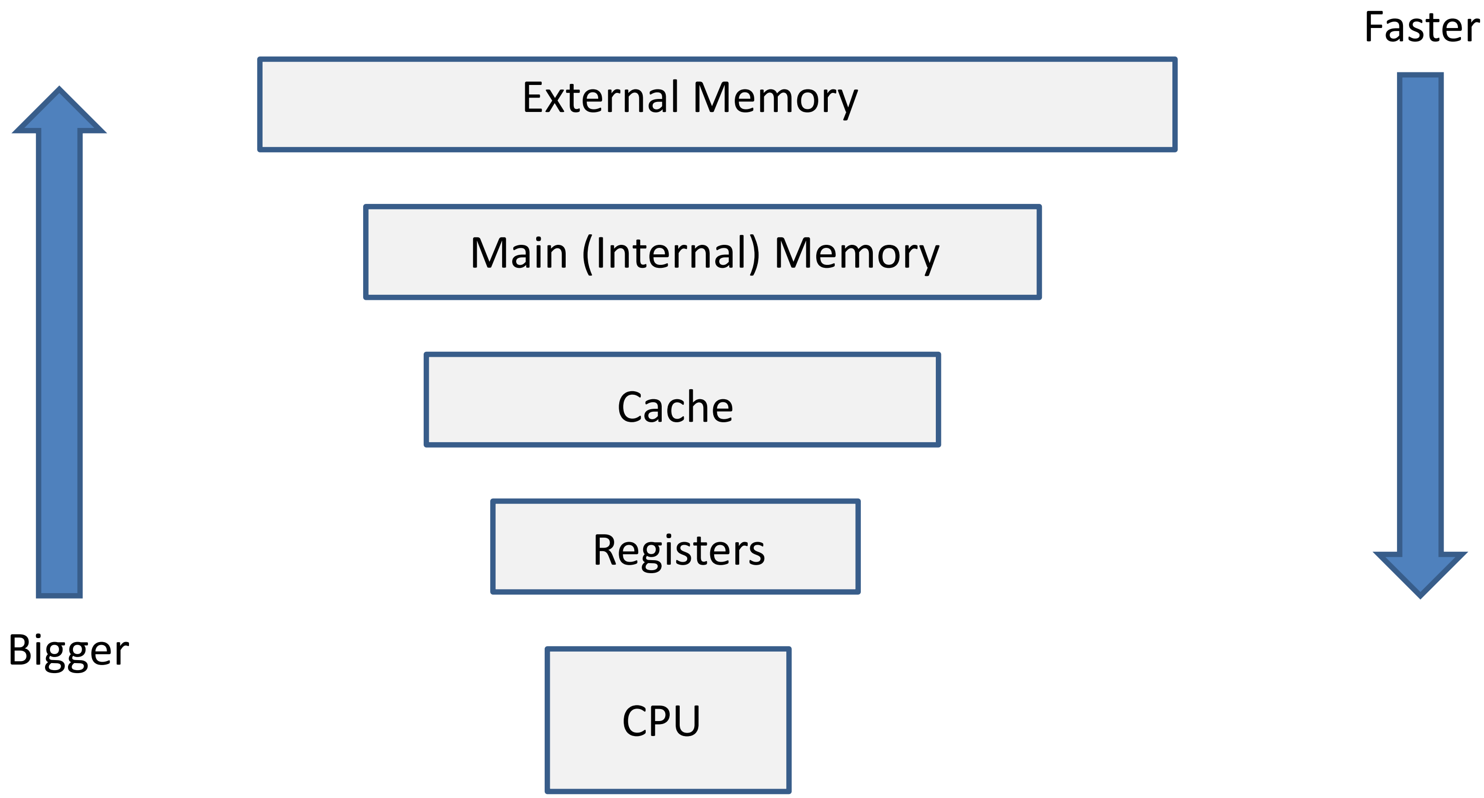
B-Trees

Manolis Koubarakis

The Memory Hierarchy



External Memory

- So far we have assumed that our data structures are stored in main memory. However, if the size of a data structure is too big then it will be stored on **external memory** e.g., on a **hard disk**.
- **Examples**: the database of a bank, a database of images, a database of videos etc.

External Searching

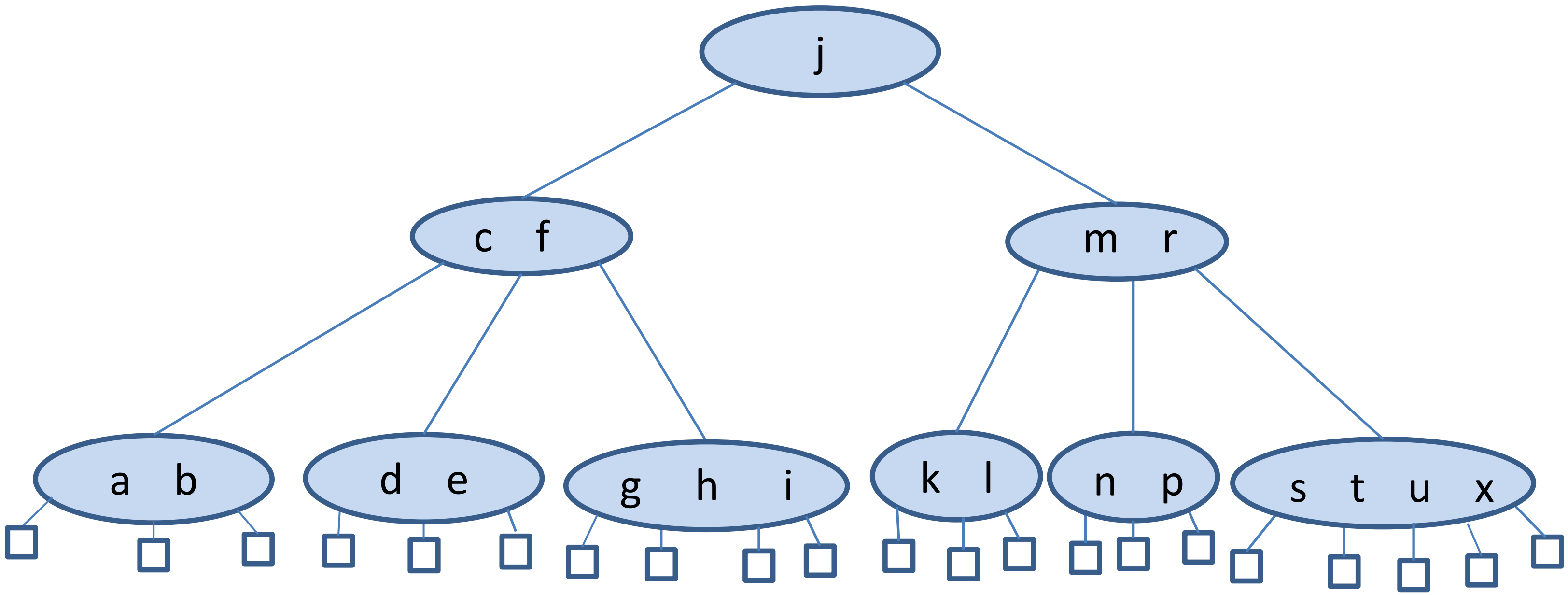
- When we access data on a disk or another external memory device, we perform **external searching.**
- A **disk access** can be at least 100,000 to 1,000,000 times longer than a main memory access.
- Thus, for data structures residing on disk, we want to **minimize disk accesses**.

(a, b) Trees

- An $(\boldsymbol{a}, \boldsymbol{b})$ **tree**, where a and b are integers, such that $2 \leq a \leq \frac{(b+1)}{2}$, is a multi-way search tree T with the following additional restrictions:
  - **Size property**: Each internal node has at least a children, unless it is the root, and at most b children. The root can have as few as 2 children.
  - **Depth property**: All external nodes have the same depth.

- A (2,4) tree is an (a, b) tree with $a = 2$ and $b = 4$.

Example (3,5) Tree



Proposition

- The height of an (a, b) tree storing n entries is $O\left(\frac{\log n}{\log a}\right)$.
- Proof?

Proof

- Let T be an (a, b) tree storing n entries and let h be the height of T. We justify the proposition by proving the following bounds on h:

$$\frac{1}{\log b}\log(n+1) \leq h < \frac{1}{\log a}\log\frac{n+1}{2}+1$$

- By the size and depth properties, the number n'' of external nodes of T is **at least $2a^{h-1}$** and **at most b^h**.
- To see the **upper bound**, consider that we can have 1 node at level 0, at most b nodes at level 1, at most b^2 nodes at level 2 etc. and at most b^h at level h (these are the external nodes).
- To see the **lower bound**, consider that we can have 1 node at level 0, 2 nodes at level 1, at least $2a$ nodes at level 2, at least $2a^2$ at level 3 etc. and at least $2a^{h-1}$ nodes at level h.

Proof (cont'd)

- By an earlier proposition for multi-way trees, we have that $n'' = n + 1$ therefore
$$2a^{h-1} \leq n + 1 \leq b^h$$
- Taking the logarithm of base 2 of each term, we get
$$(h-1)\log a + 1 \leq \log(n+1) \leq h \log b$$
- The lower bound we want to prove is obvious from the above right inequality.
- The upper bound we want to prove is also easy to see using the left inequality from above:
$$h \log a - \log a + 1 \leq \log(n+1)$$
$$h \log a \leq \log(n+1) + \log a - 1$$
$$h \leq \frac{1}{\log a} \log \frac{n+1}{2} + 1 - \frac{1}{\log a}$$
$$h < \frac{1}{\log a} \log \frac{n+1}{2} + 1$$

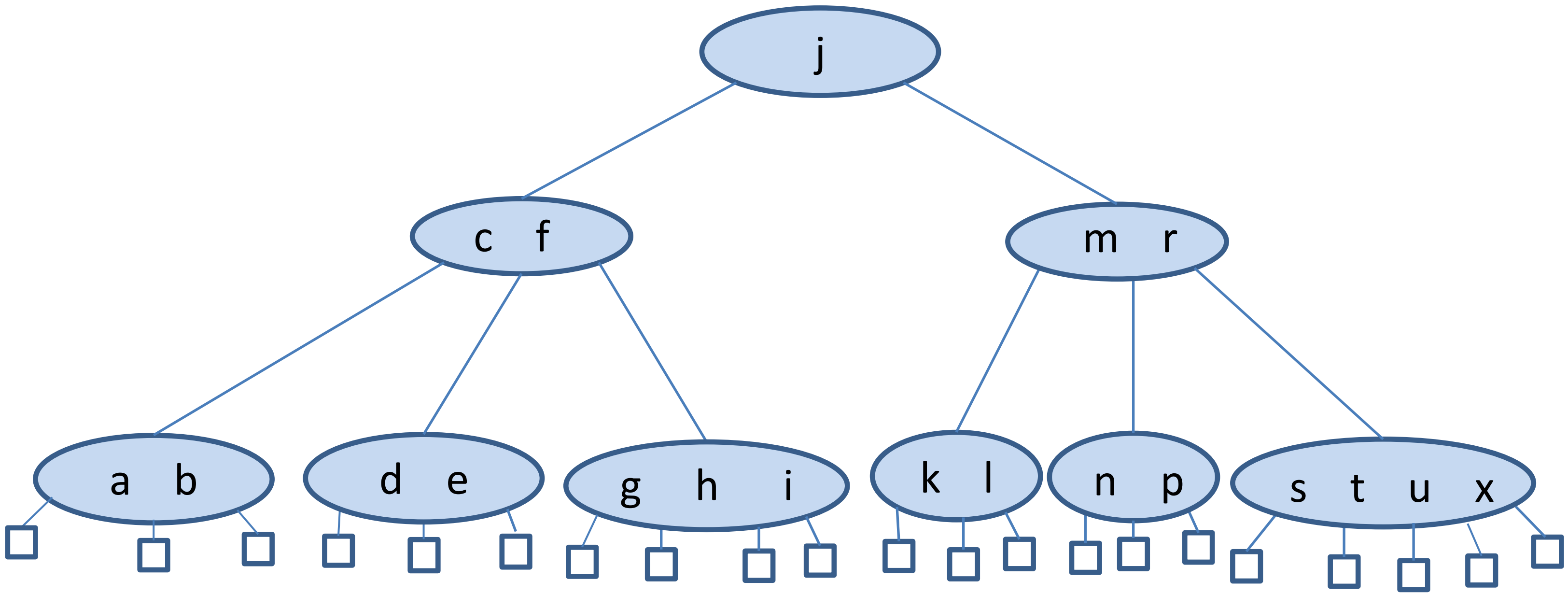
B-Trees

- In an (a, b) tree , we can select the parameters a and b so that each tree node occupies a **single disk block** or **page**.
- This gives rise to a well-known external memory data structure called the B-tree.

- A **B-tree of order m** is an (a, b) tree with $a = \lceil \frac{m}{2} \rceil$ and $b = m$.

- B-trees are used for **indexing** data stored on external memory.
- When we implement a B-tree, we choose the order m so that the (at most) m children references and the (at most) $m - 1$ keys stored at a node can all fit into a **single block**.
- Nodes are **at least half-full** all the time due to the value of a.

Example B-Tree of Order $m = 5$



Proposition

- Let T be a B-tree of order m and height h. Let $d = \lceil\frac{m}{2}\rceil$ and n the number of entries in the tree. Then, the following inequalities hold:
  1. $2d^{h-1} - 1 \leq n \leq m^h - 1$
  2. $\log_m(n+1) \leq h \leq \log_d \frac{(n+1)}{2} + 1$
- Proof?

Proof

- Let us prove (1) first.
- The upper bound follows from the fact that a B-tree of order m is a multi-way tree and the respective proposition we proved for multi-way trees.
- The lower bound follows from an inequality we used in the proof of the previous proposition that for (a, b) trees.
- To prove (2), rewrite the inequalities of (1) and then take logarithms with bases m and d for the respective terms.

Result

- From the right inequality of (2) in the previous proposition, we have that the height of a B-tree is $\boldsymbol{O(\log_d n)}$ where $d = \lceil \frac{m}{2} \rceil$, as we would like it for a balanced search tree.

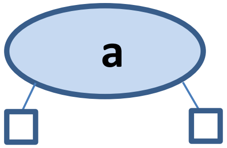
Insertion into a B-tree

- The general method for insertion in a B-tree is as follows. First, a search is made to see if the new key is in the tree. This search (if the tree is truly new) will terminate in failure at a leaf.
- The new key is then added to the parent of the leaf node. If the node was not previously full, then the insertion is finished.
- When a key is added to a full node, we have an **overflow**. Then this node **splits** into two nodes on the same level, except that the **median key** at position $\lceil \frac{m}{2} \rceil$ is not put into either of the two new nodes, but is instead sent up to the tree to be inserted into the parent node.
- When a search is later made through the tree, a comparison with the median key will serve to direct the search into the proper subtree.

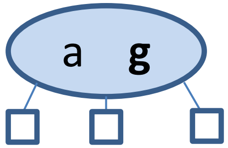
Example

- Let us see an example of insertions into an initially empty B-tree of **order 5**.

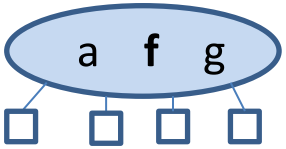
Insert a



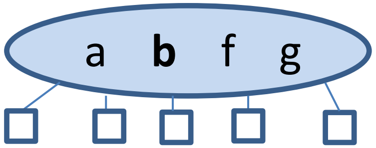
Insert g



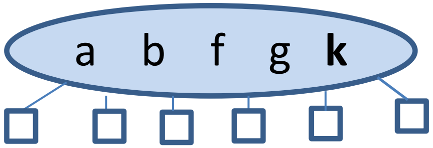
Insert f



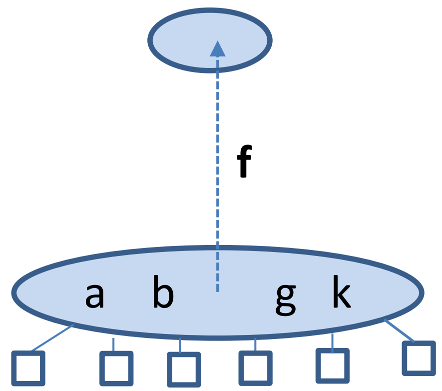
Insert b



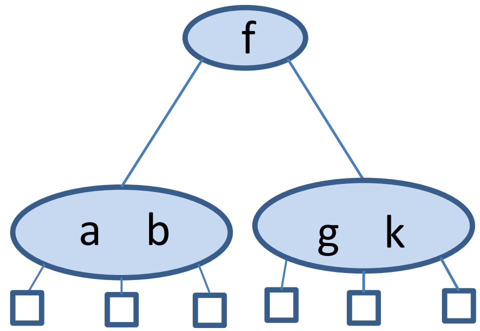
Insert k - Overflow



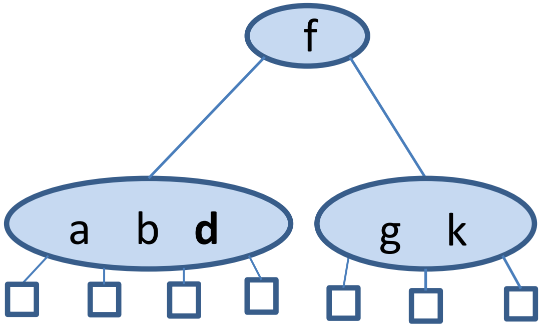
Creation of a New Root Node



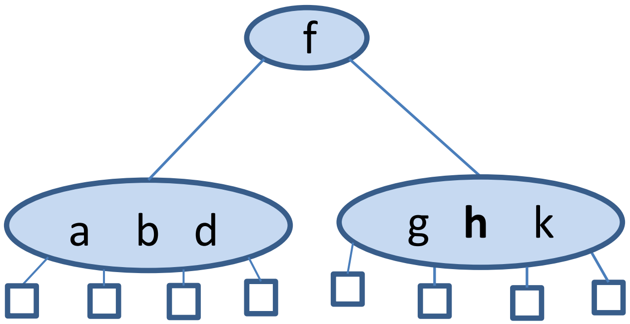
Split



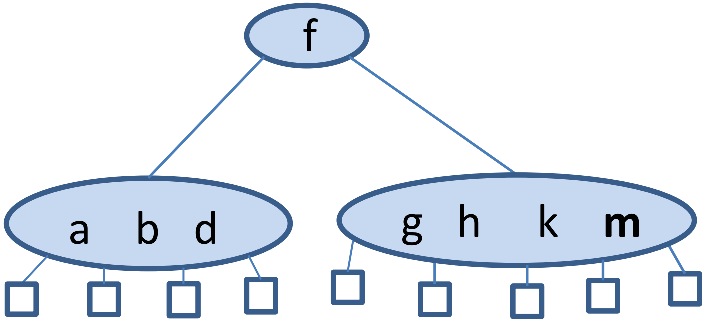
Insert d



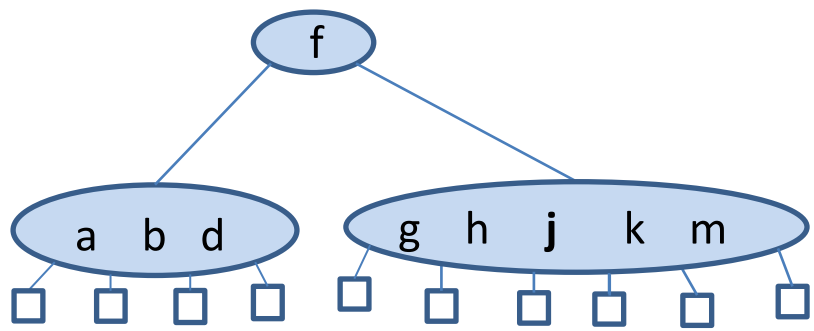
Insert h



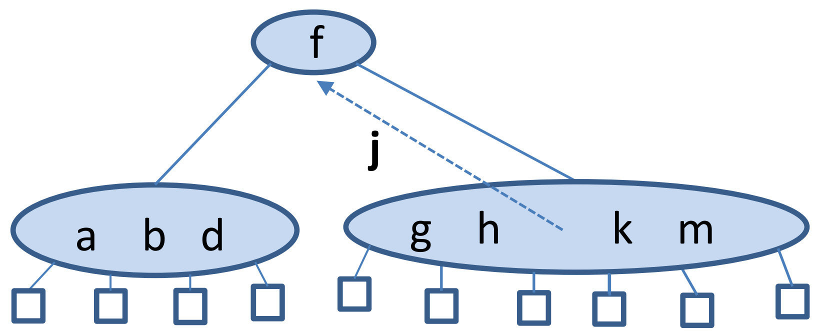
Insert m



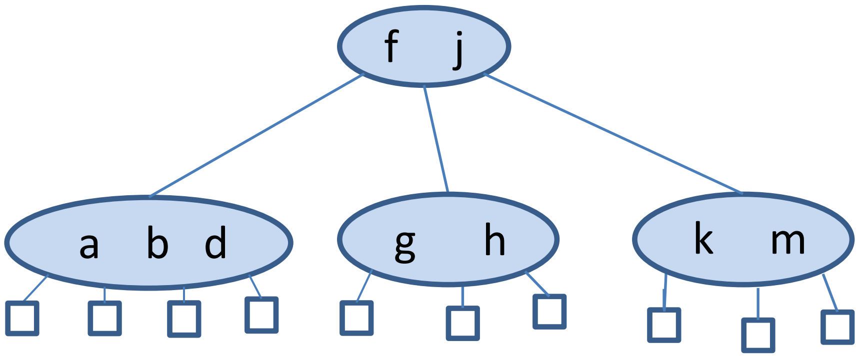
Insert j - Overflow



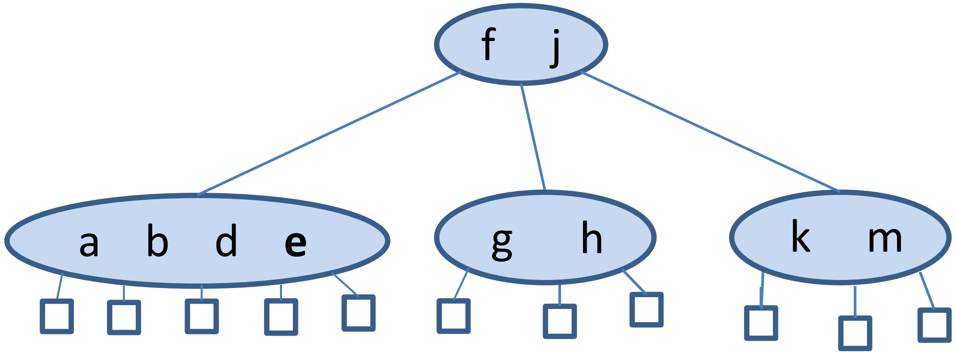
Sent j to the Parent Node



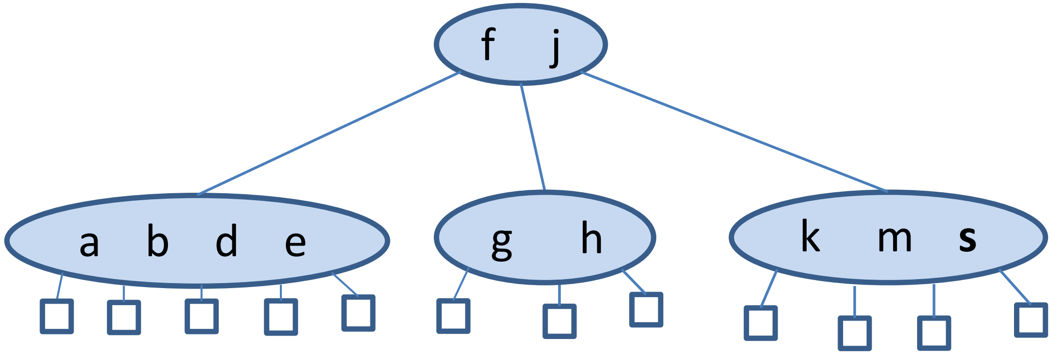
Split



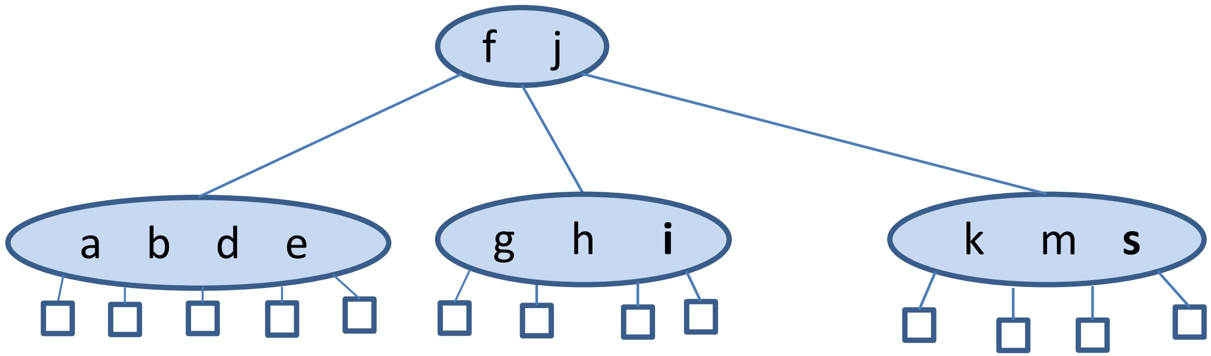
Insert e



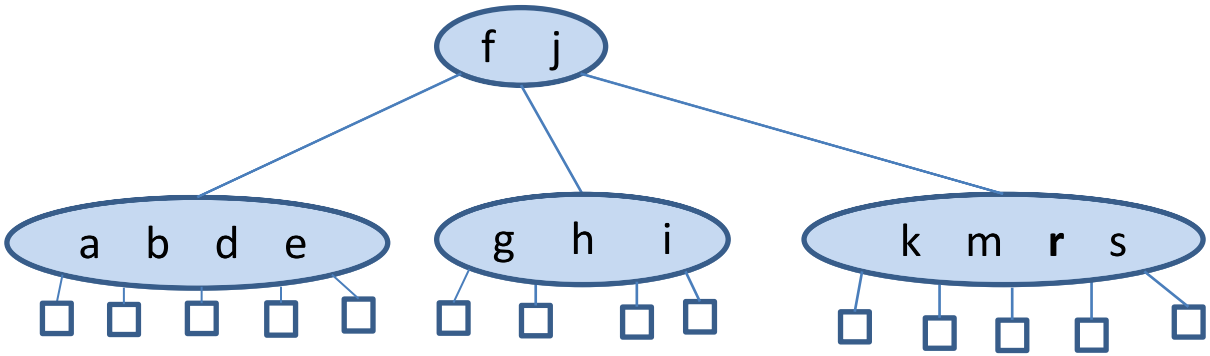
Insert s



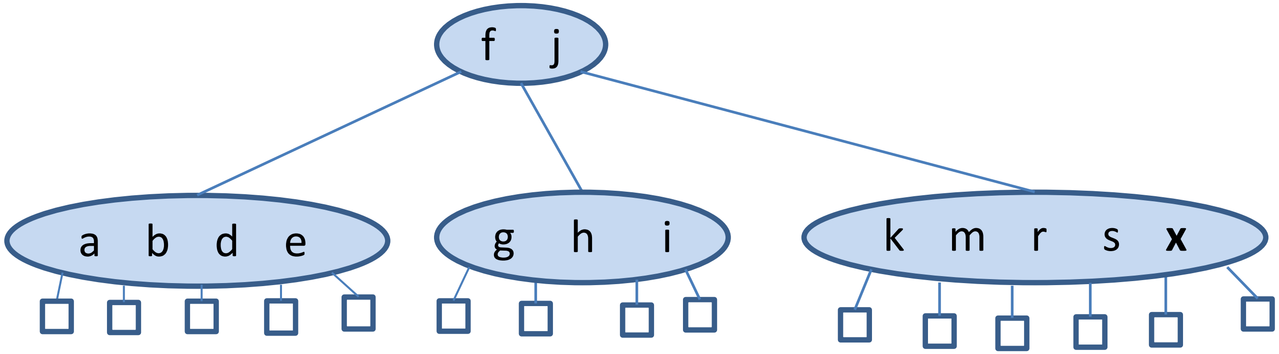
Insert i



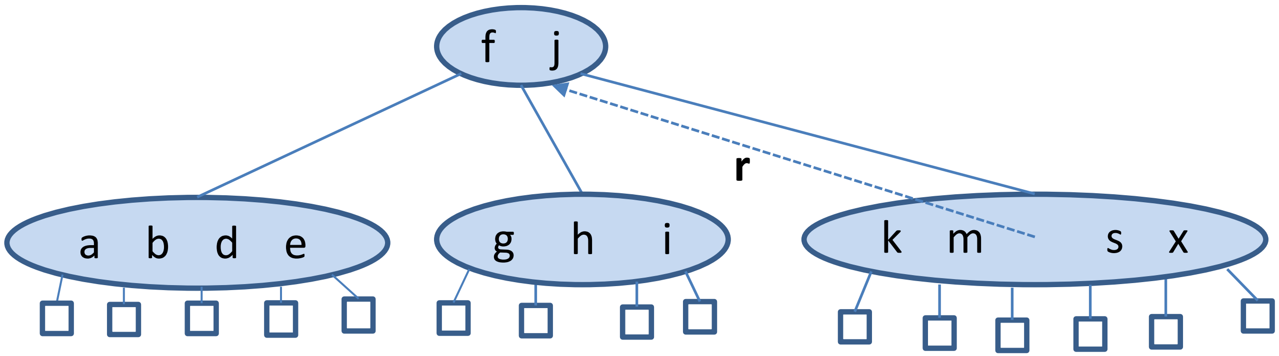
Insert r



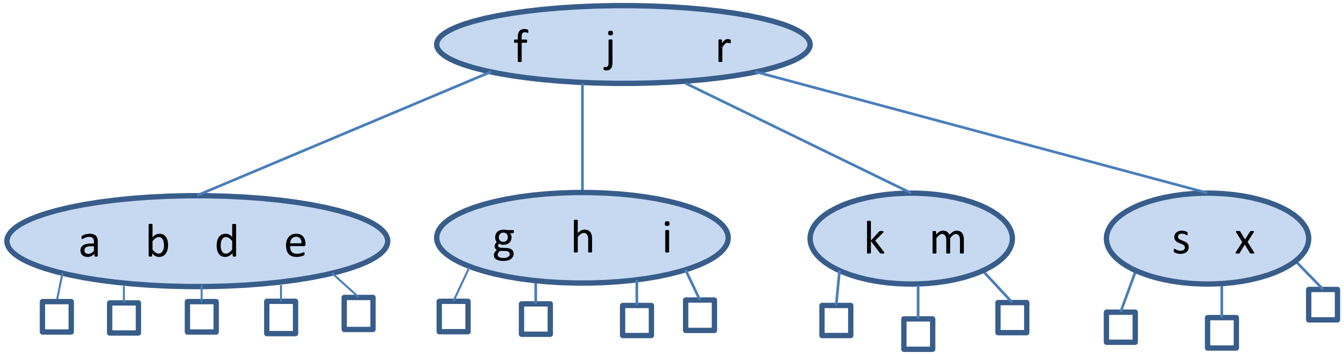
Insert x - Overflow



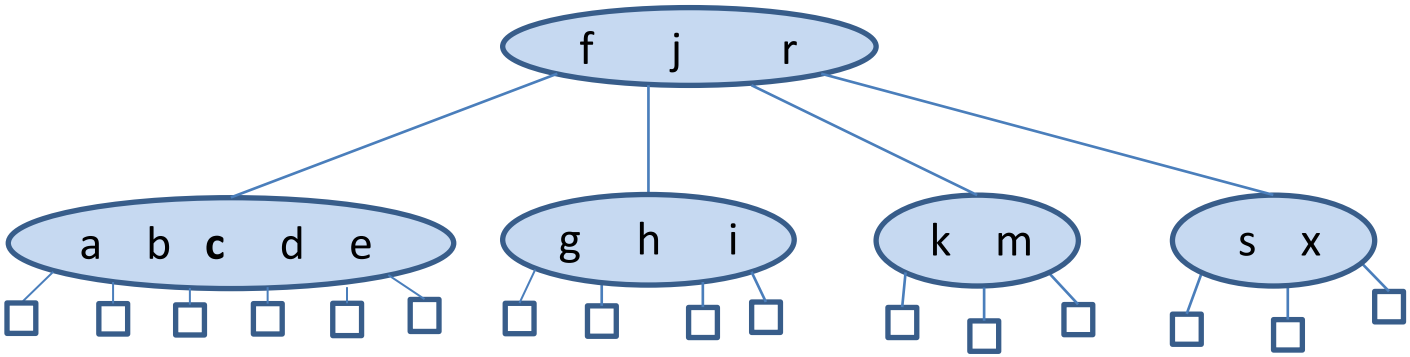
r is Sent to the Parent Node



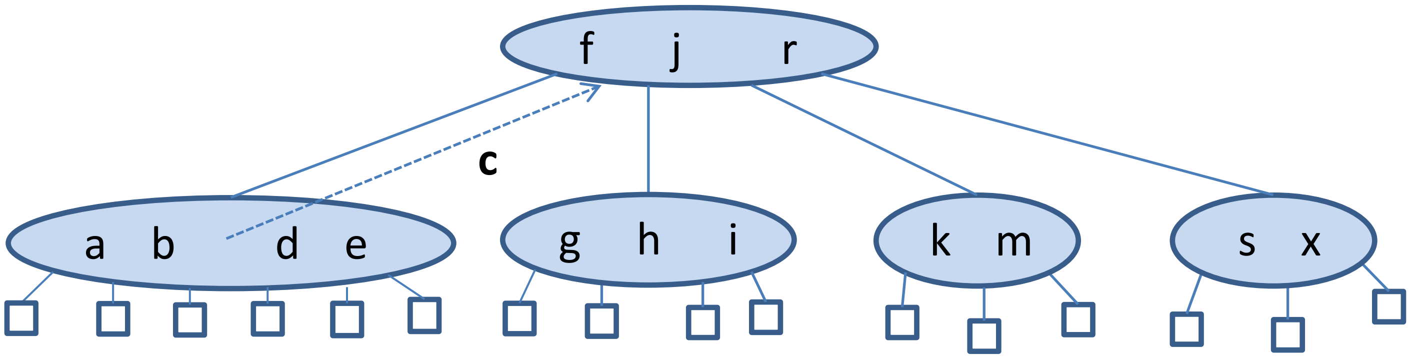
Split



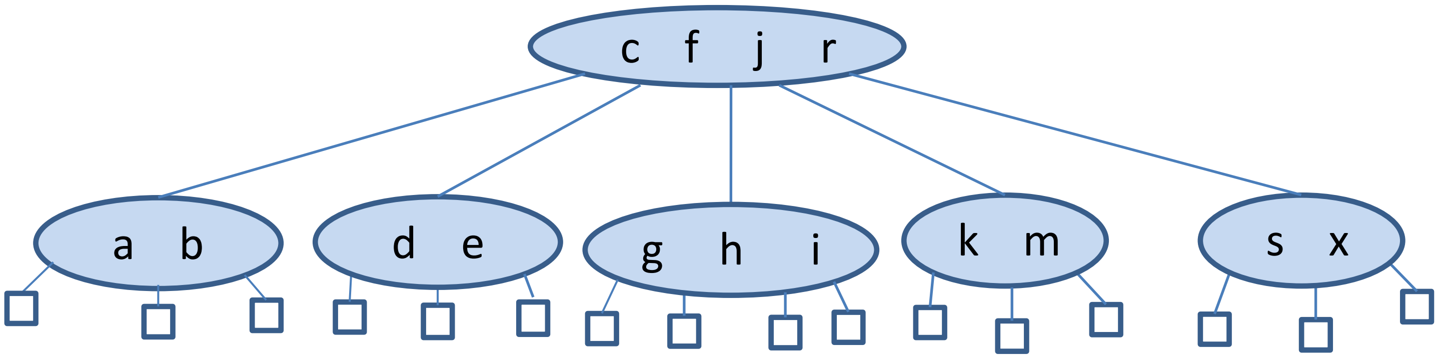
Insert c - Overflow



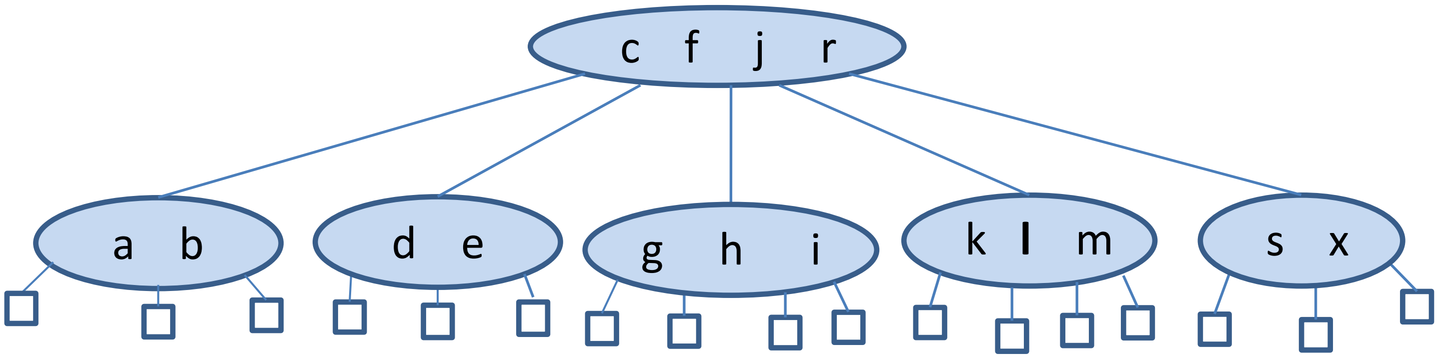
c is Sent to the Parent



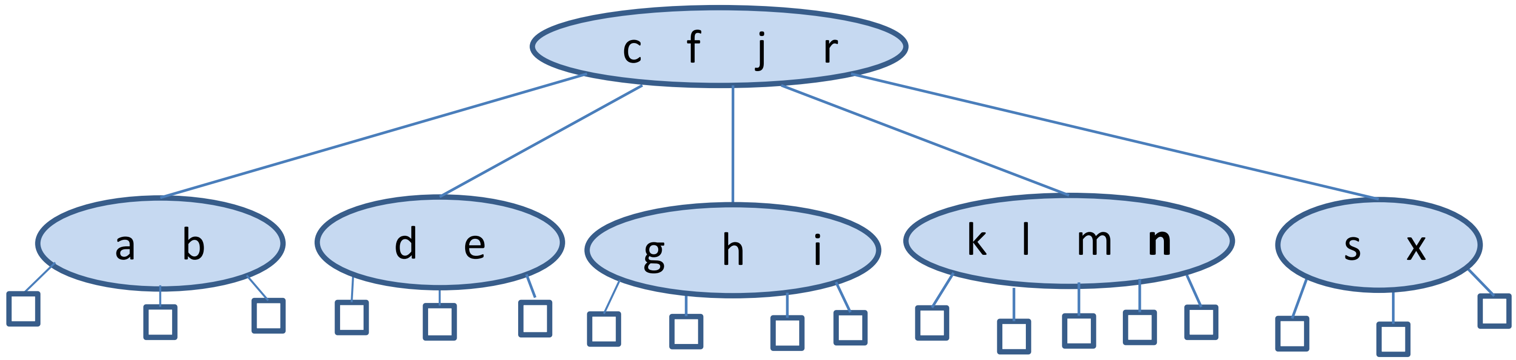
Split



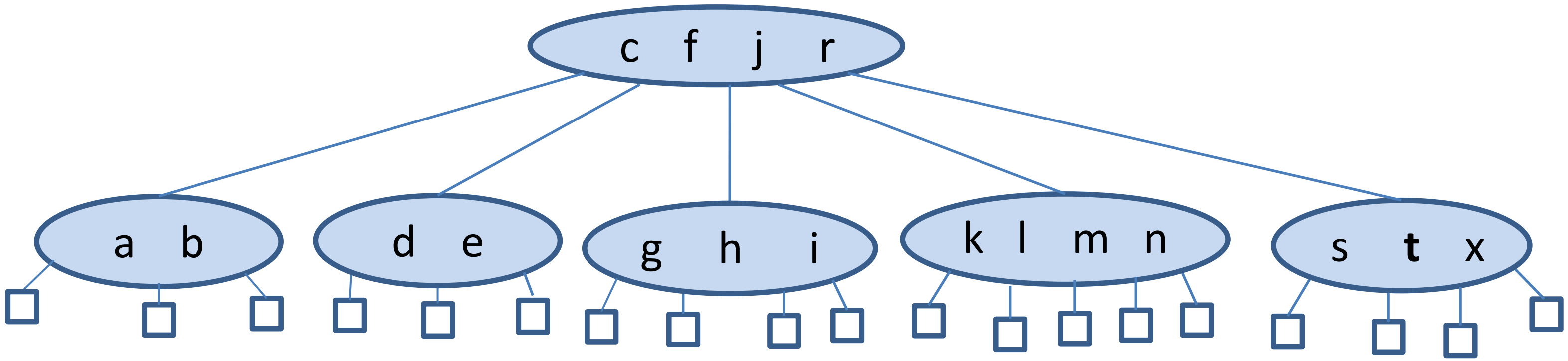
Insert l



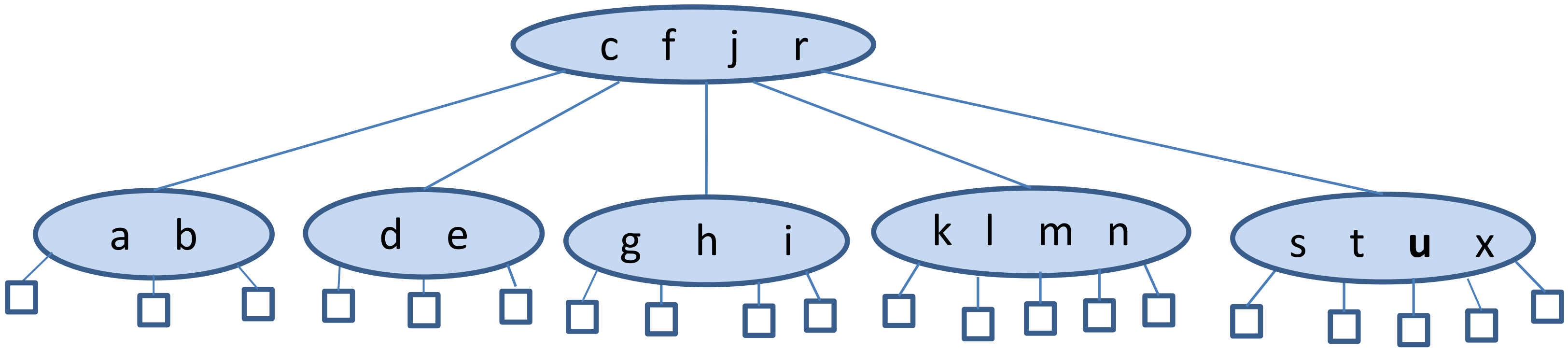
Insert n



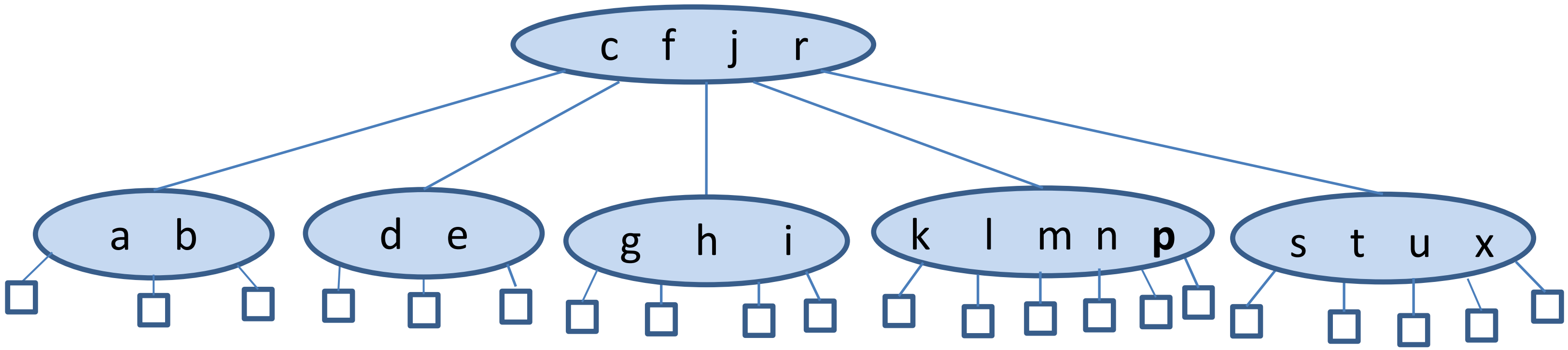
Insert t



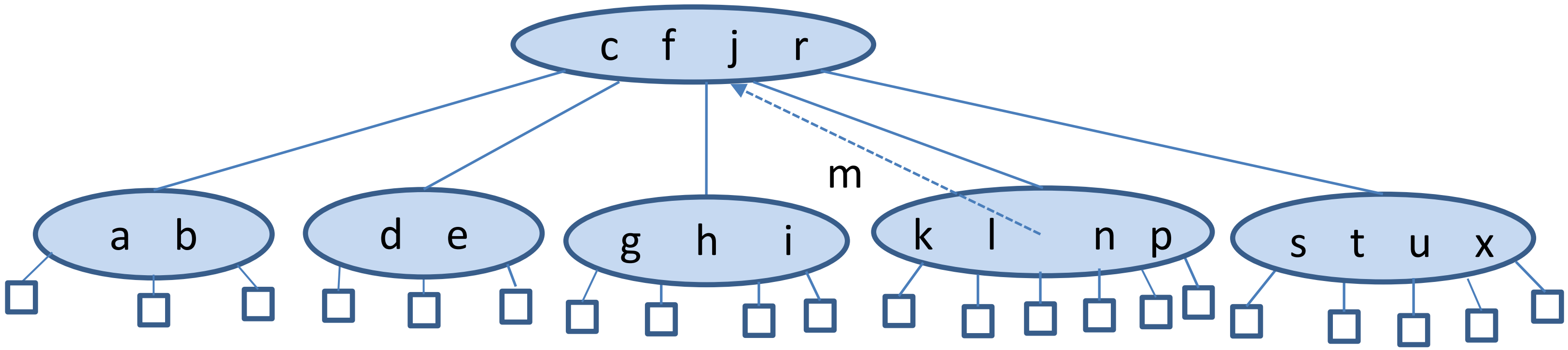
Insert u



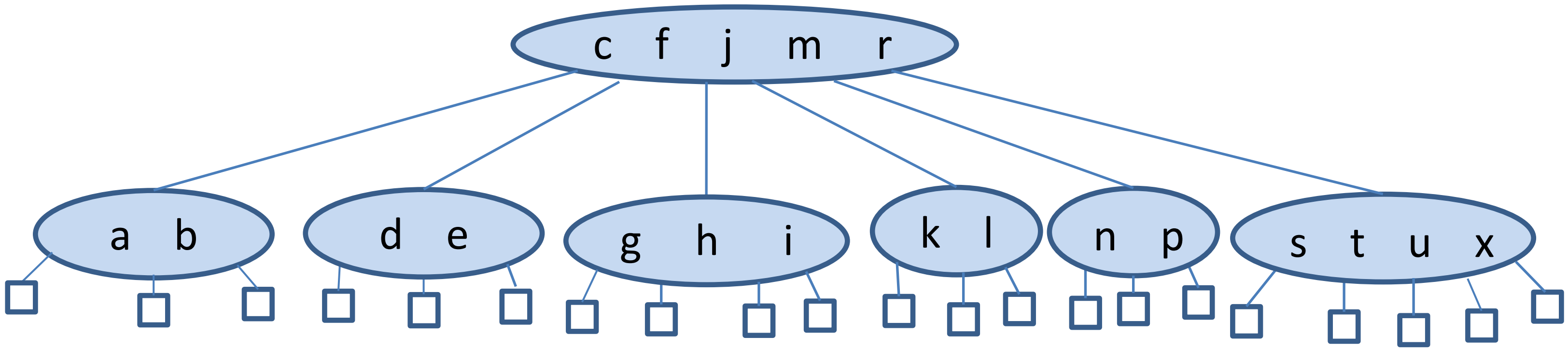
Insert p - Overflow



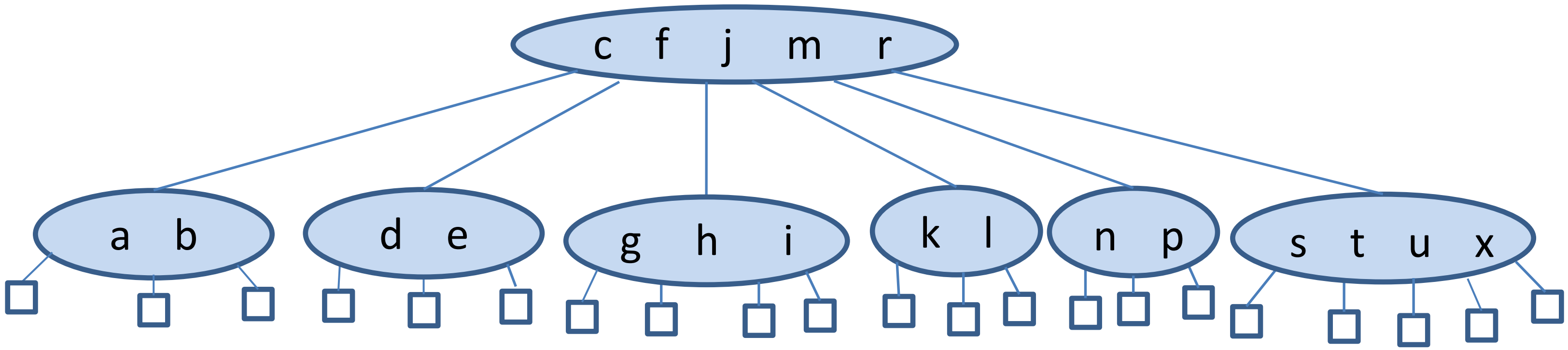
m is Sent to the Parent Node



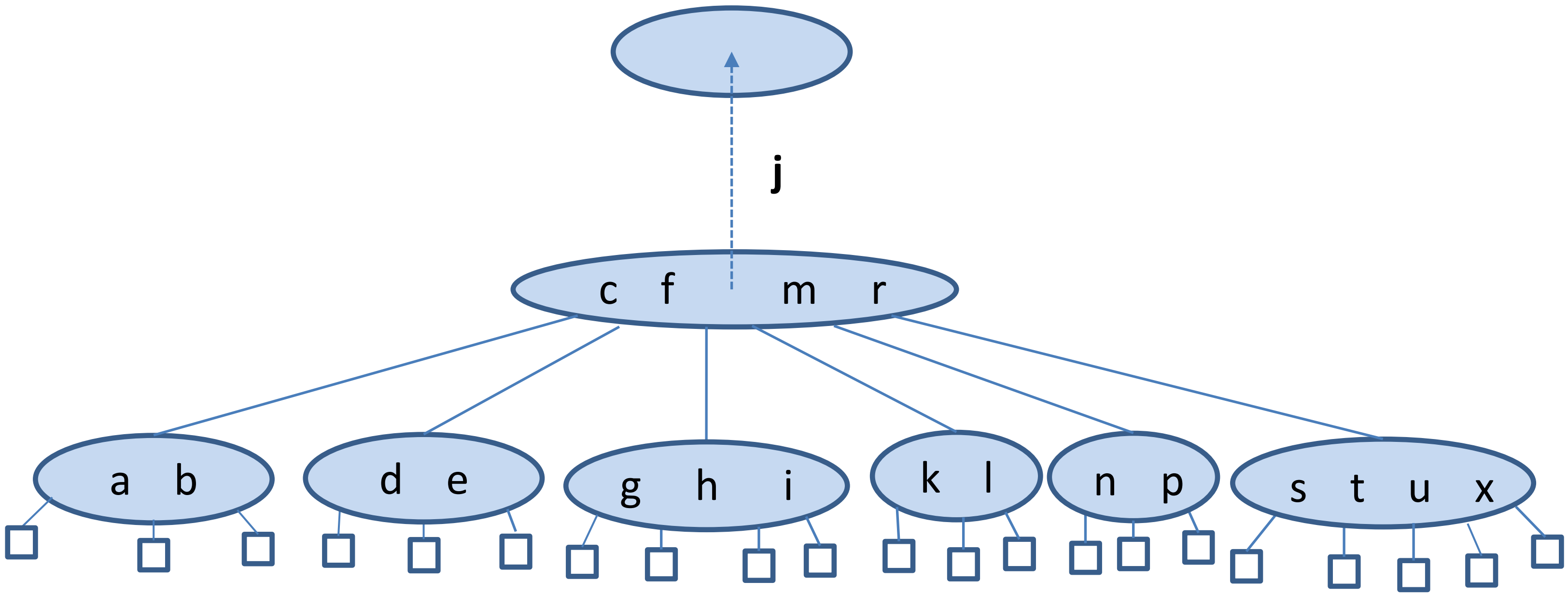
Split



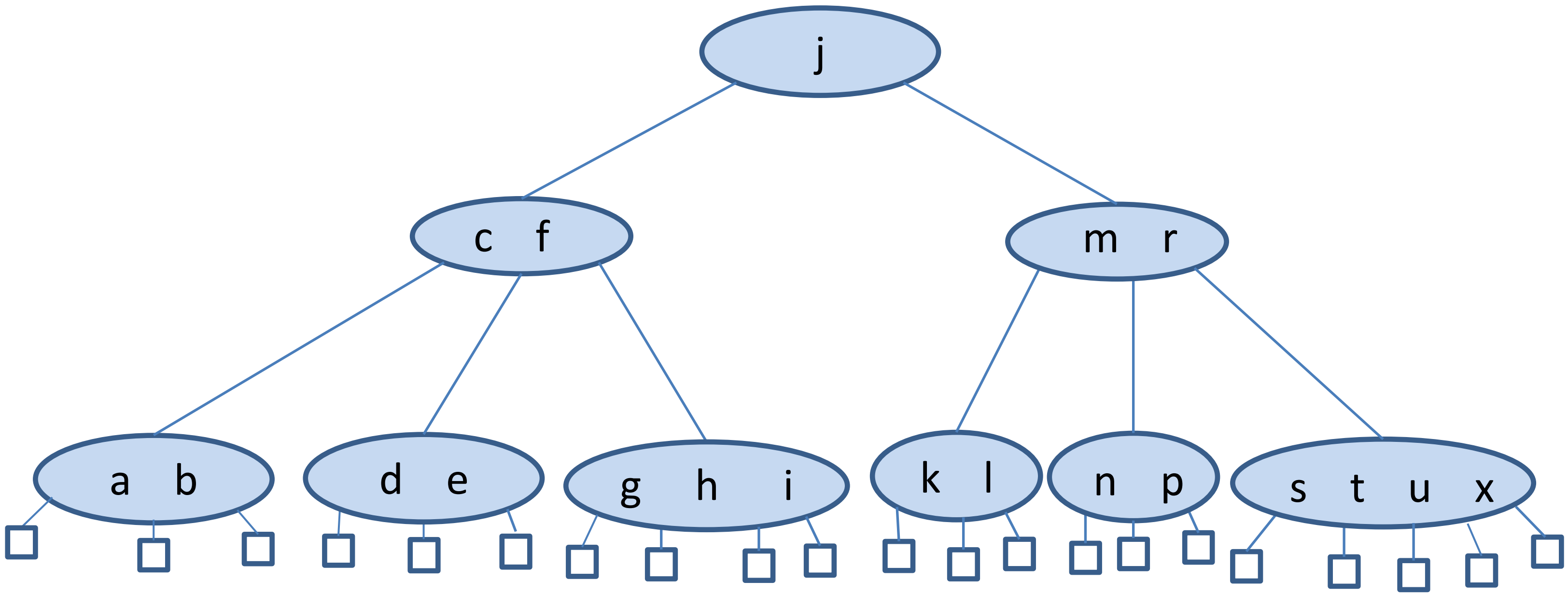
Overflow at the Root



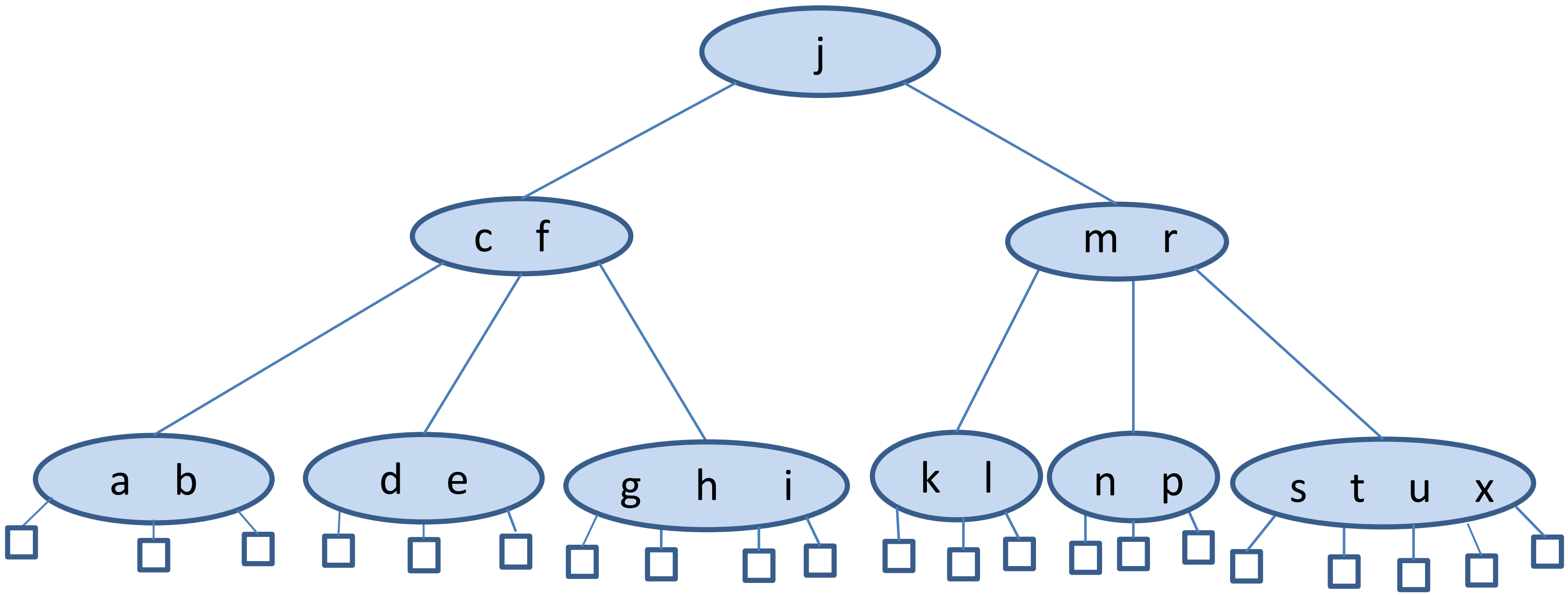
j is Sent up to a New Root



Split



Final Tree



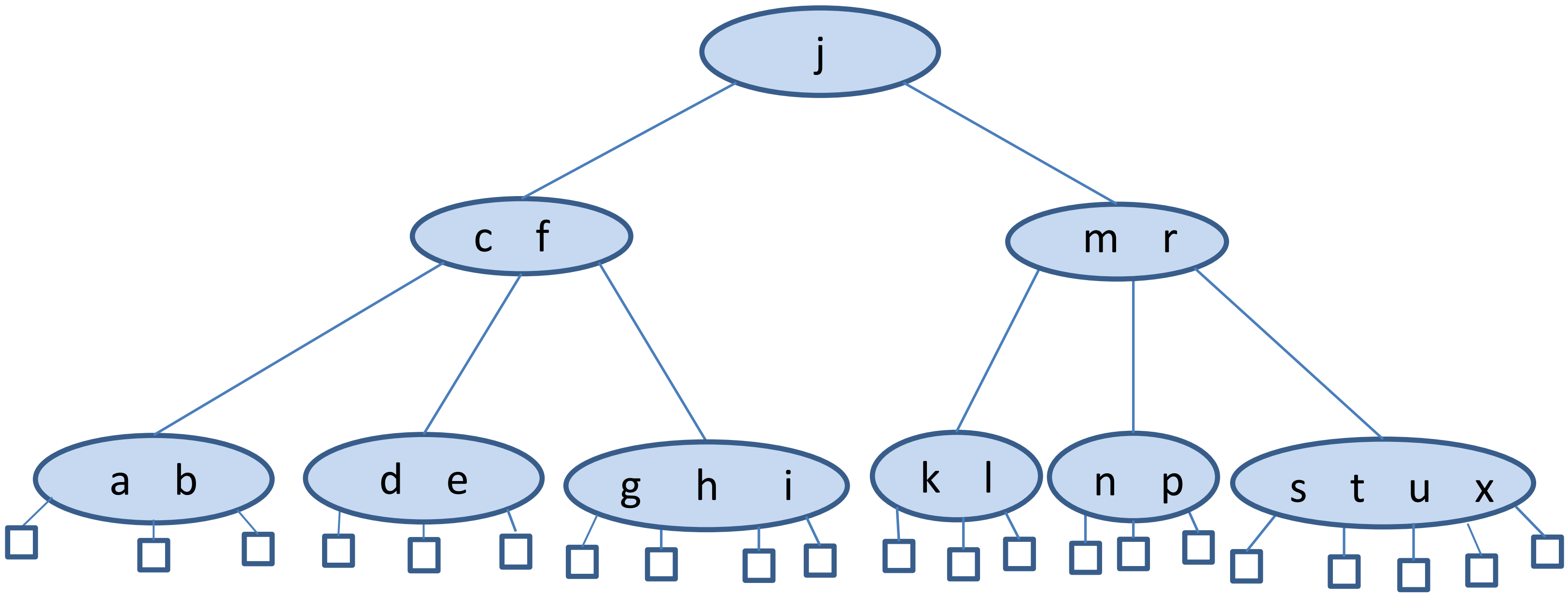
Deletion from a B-tree

- Let us now see how we **delete a key** from a B-tree.
- If the key to be deleted is in a node with only external nodes as children, then it can be deleted immediately.
- If the key to be deleted is in an internal node with only internal nodes as children, then its **immediate predecessor** (or **successor**) under the natural order of keys is guaranteed to be in a node with only external-node children.
- Hence, we can **promote** the immediate predecessor or successor into the position occupied by the key to be deleted, and delete the key from the node with only external-node children.

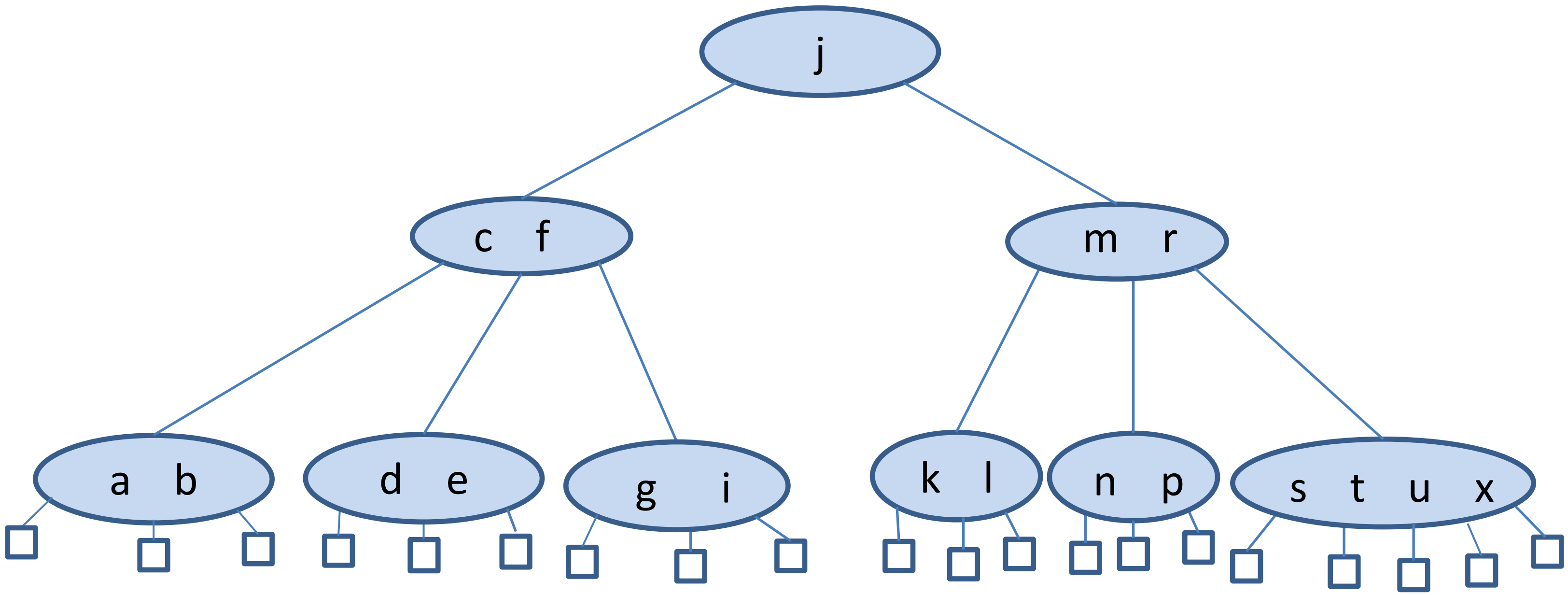
Deletion from a B-tree (cont'd)

- If the node where the deletion takes place contains **more than the minimum number of keys**, then one can be deleted with no further action.
- If the node contains the **minimum number**, then we first **look at its two immediate siblings** (or in the case of a node on the outside, one sibling).
- If one of these has more than the minimum number for entries, then we can do a **transfer** operation: one child of the sibling is moved to the node where the deletion takes place, one of the keys of the sibling is moved into the parent node, and a key from the parent node is moved into the node where the deletion takes place.
- If the immediate sibling has only the minimum number of keys then we perform a **fusion** operation: the current node and its sibling are merged into a new node and a key is moved from the parent into this new node.
- If this fusion step leaves the parent with too few entries, the process **propagates upward.**

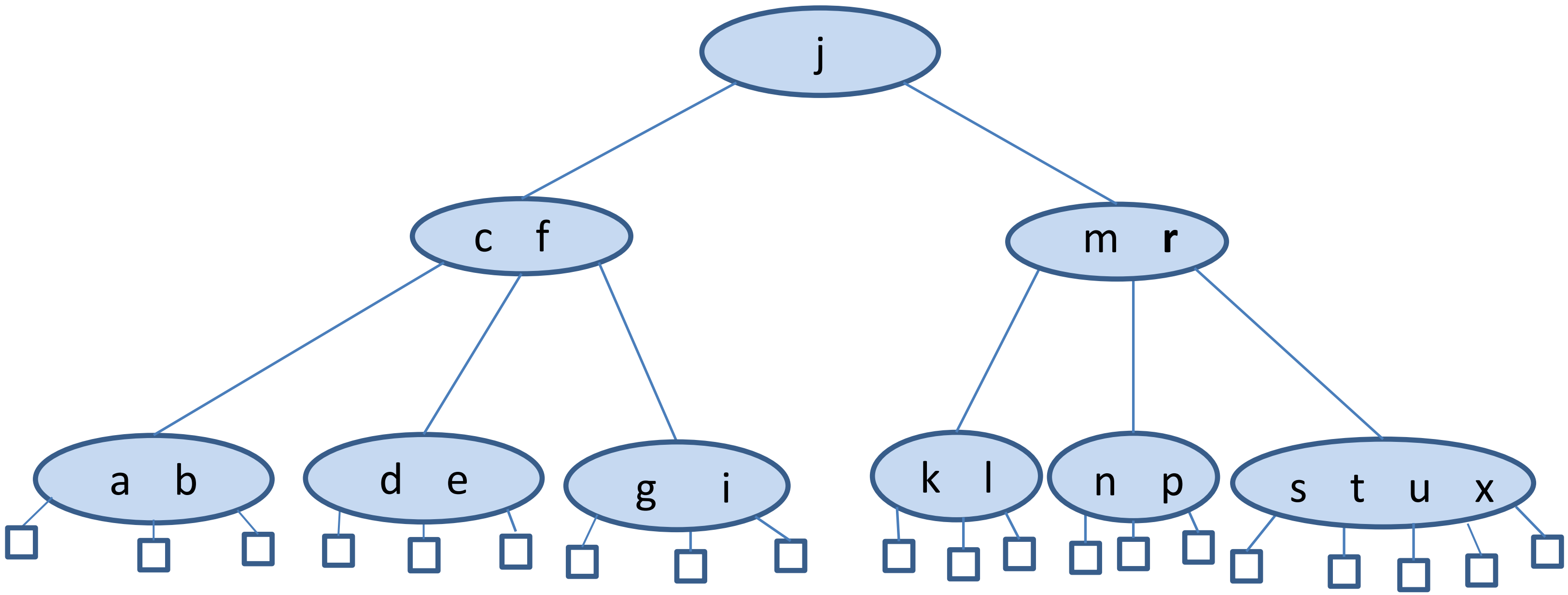
Example



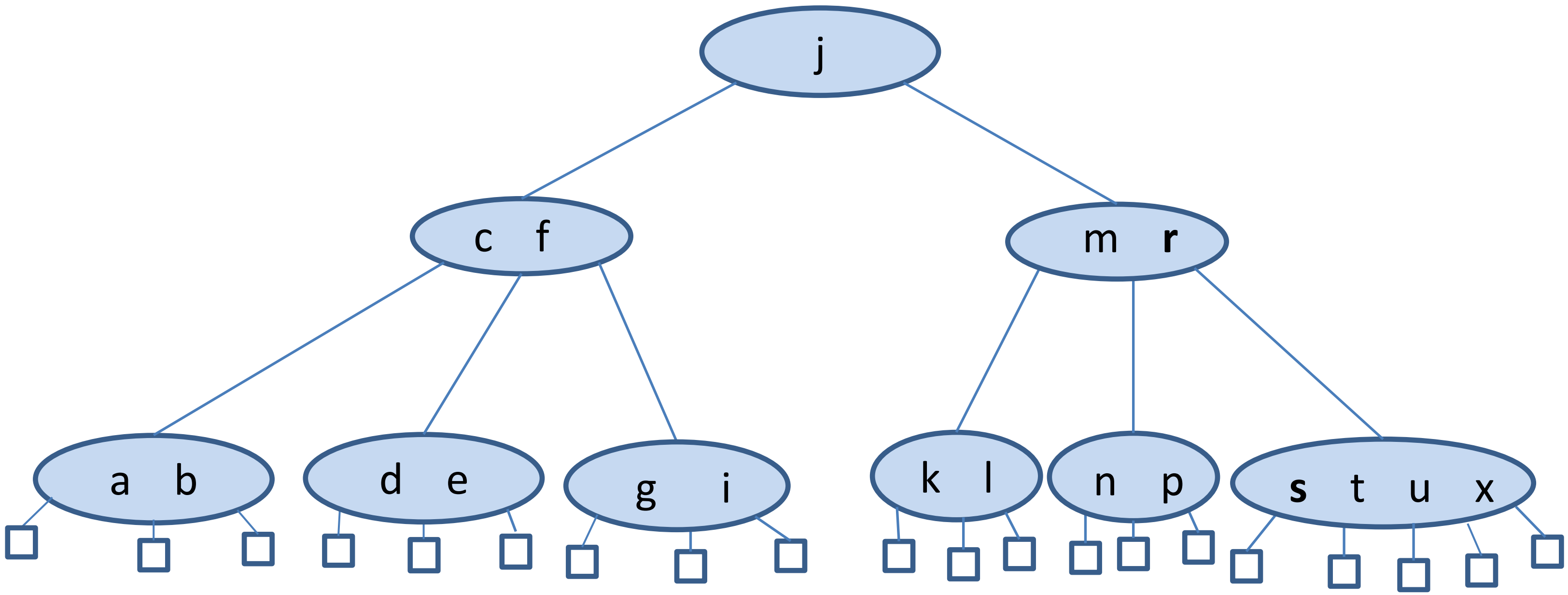
Delete h



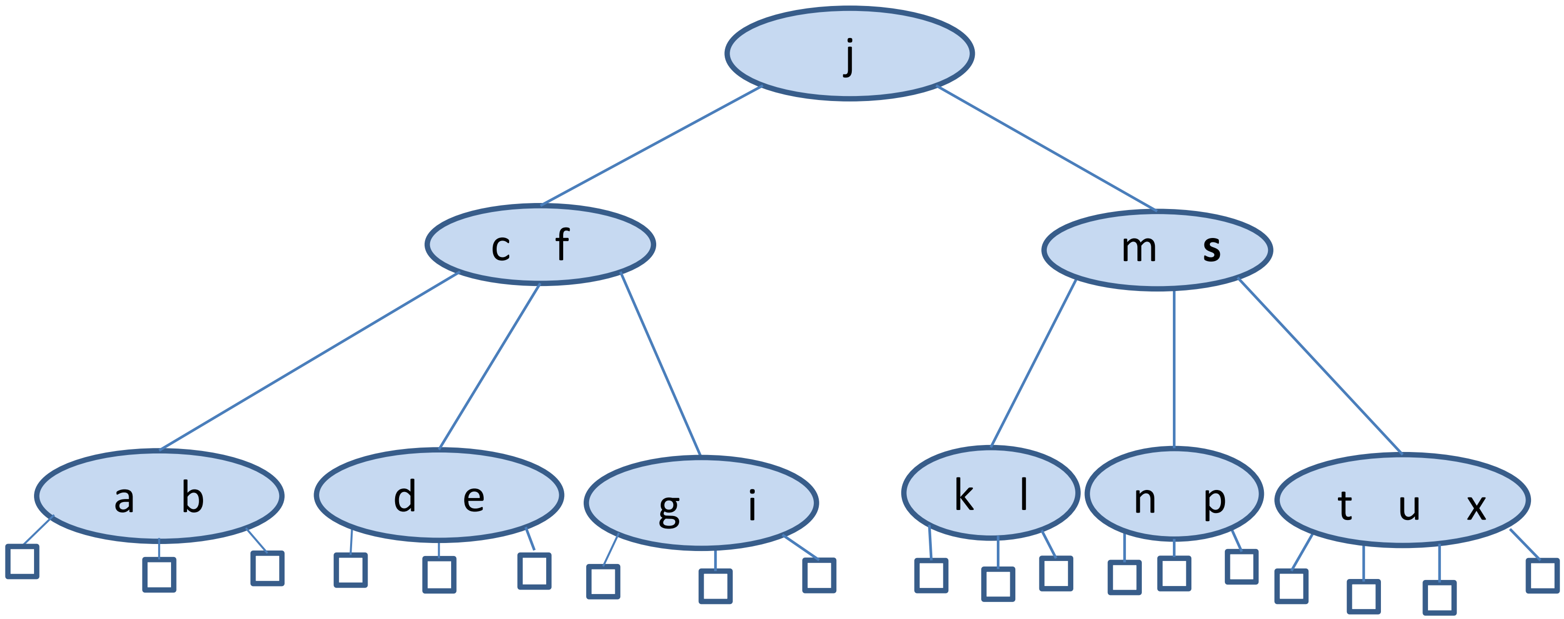
Delete r



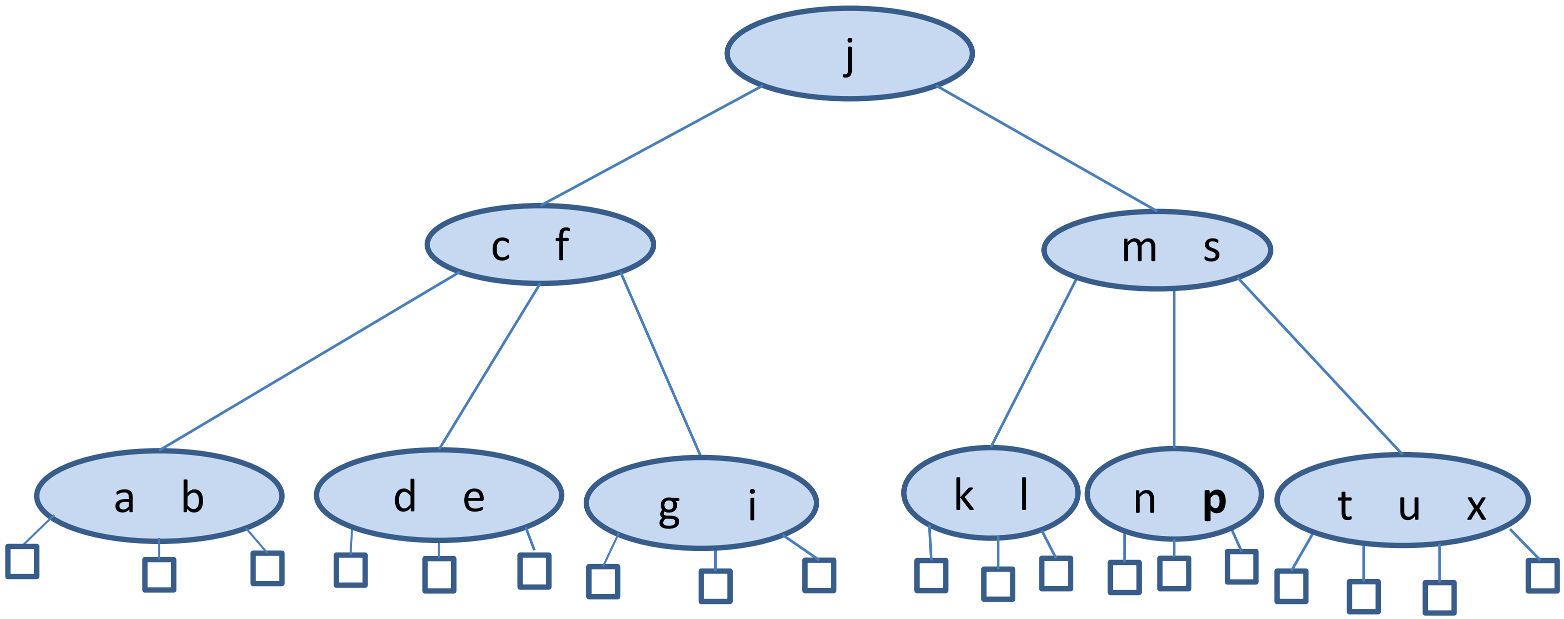
Find the Successor of r



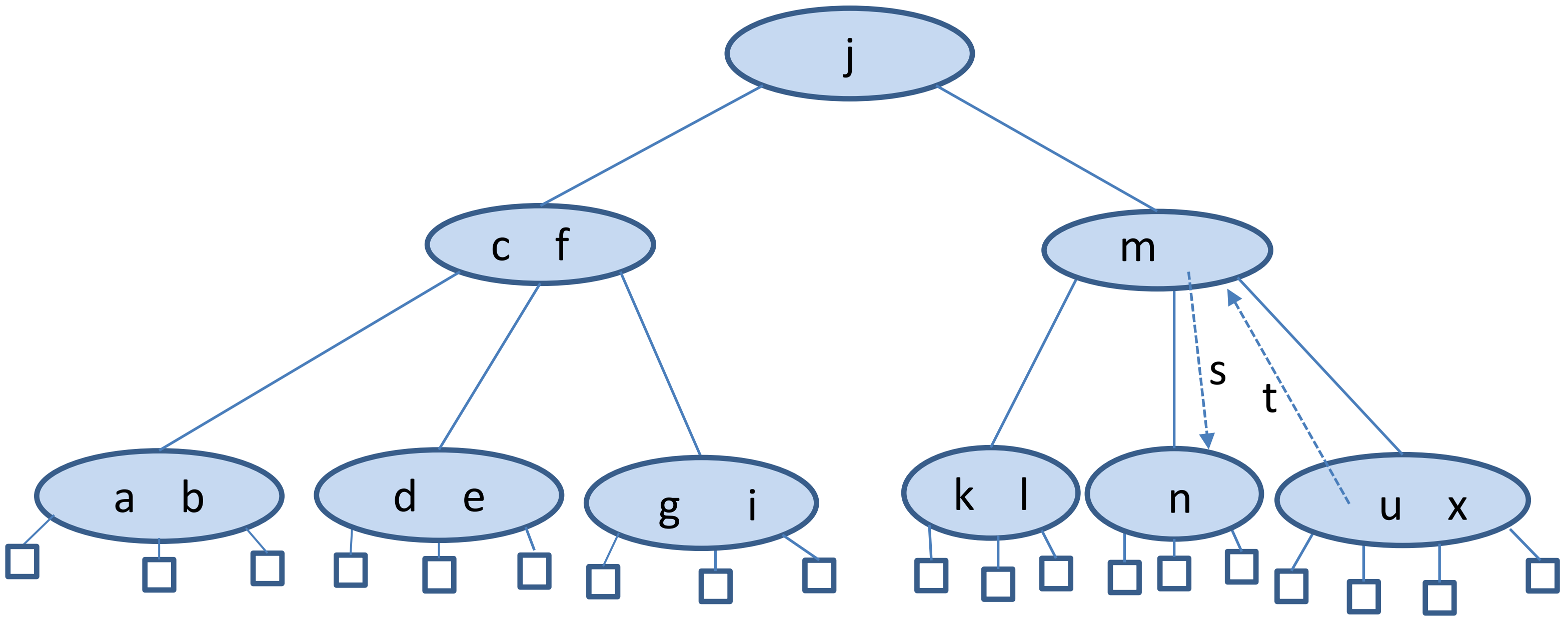
Promote the Successor of r – Delete the Successor from its Place



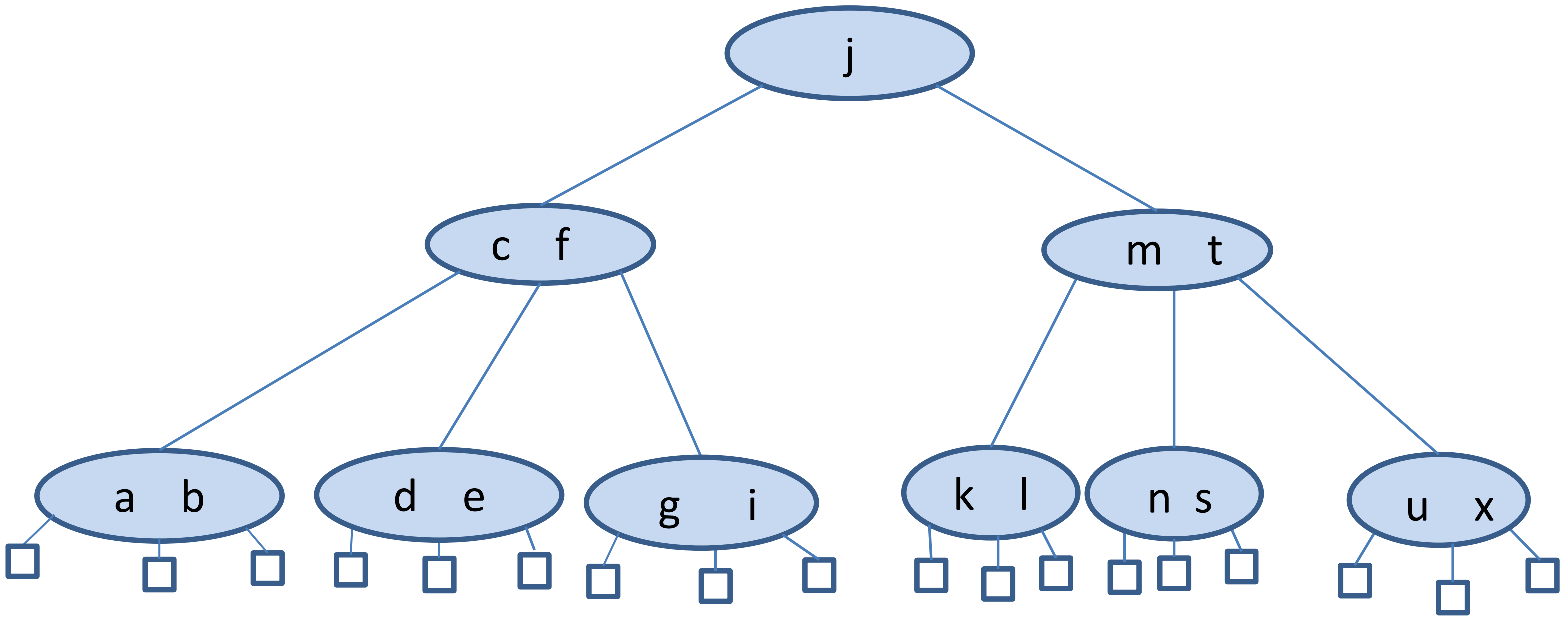
Delete p



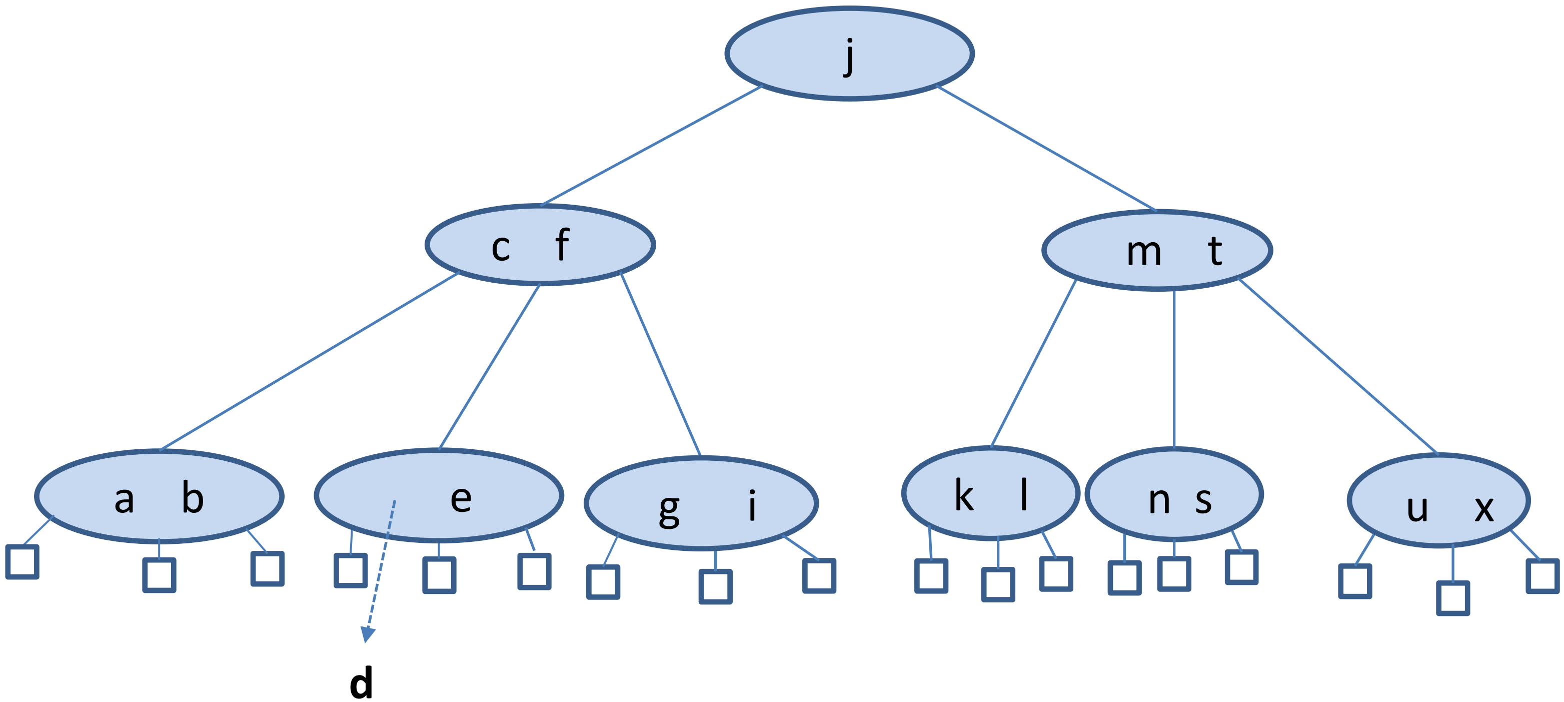
Transfer



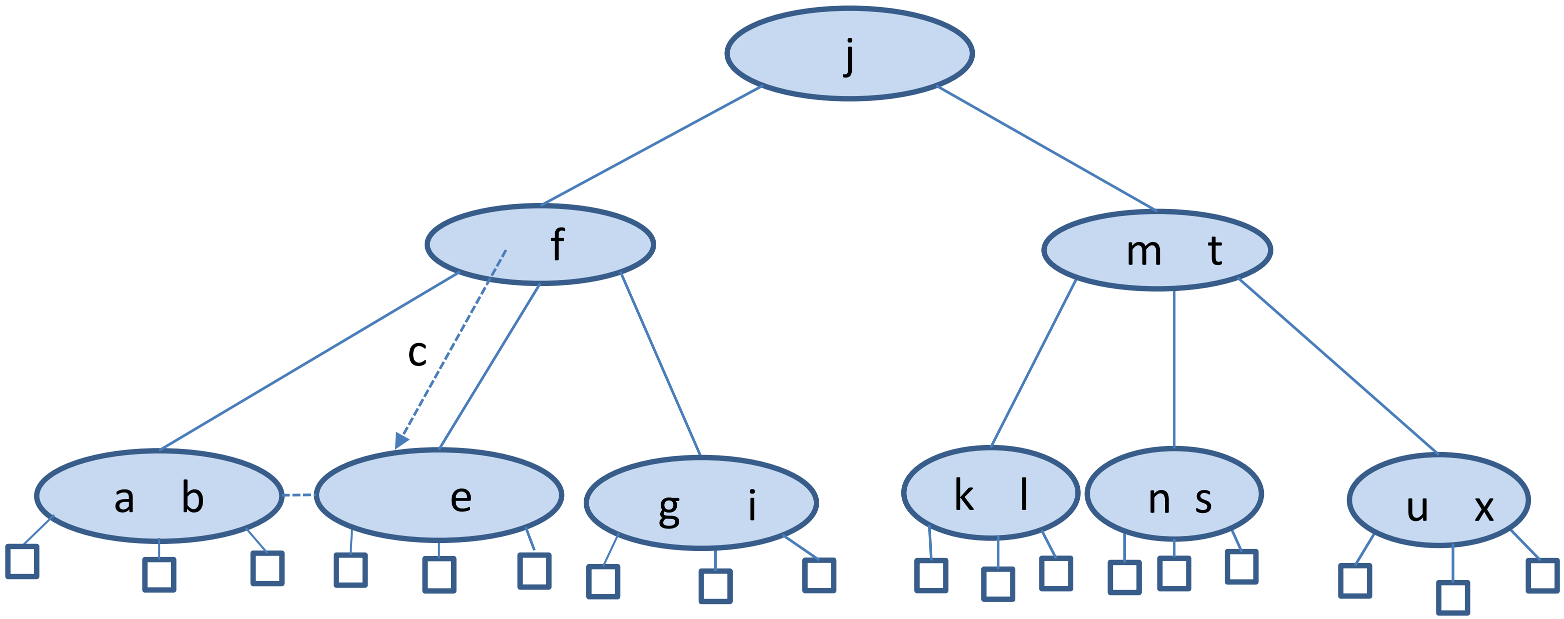
After the Transfer



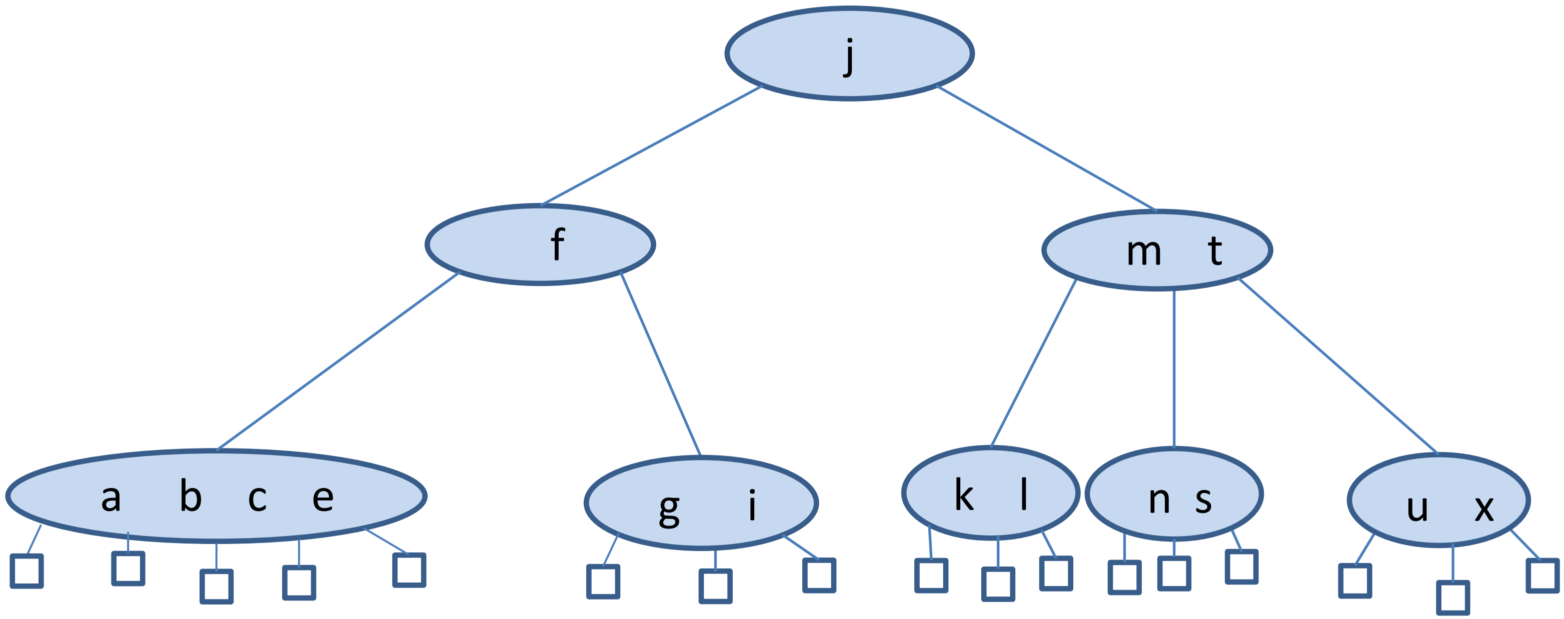
Delete d



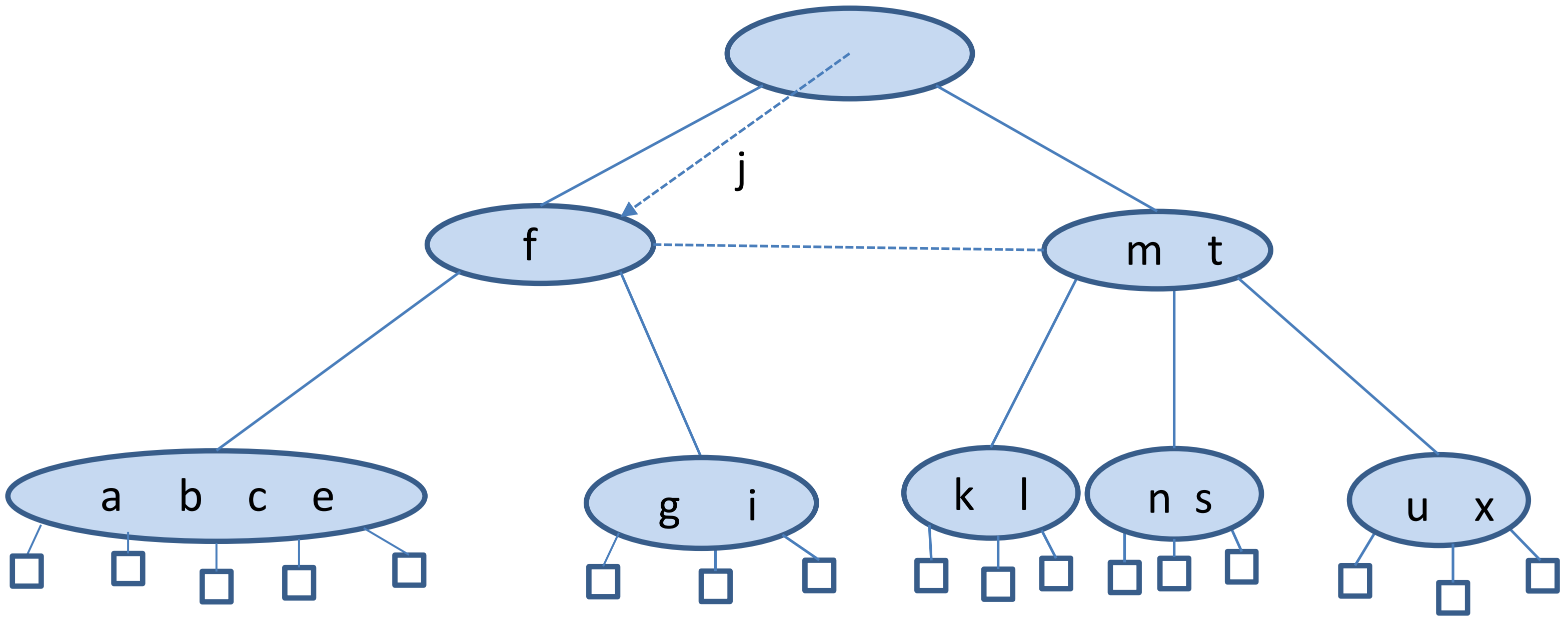
Fusion



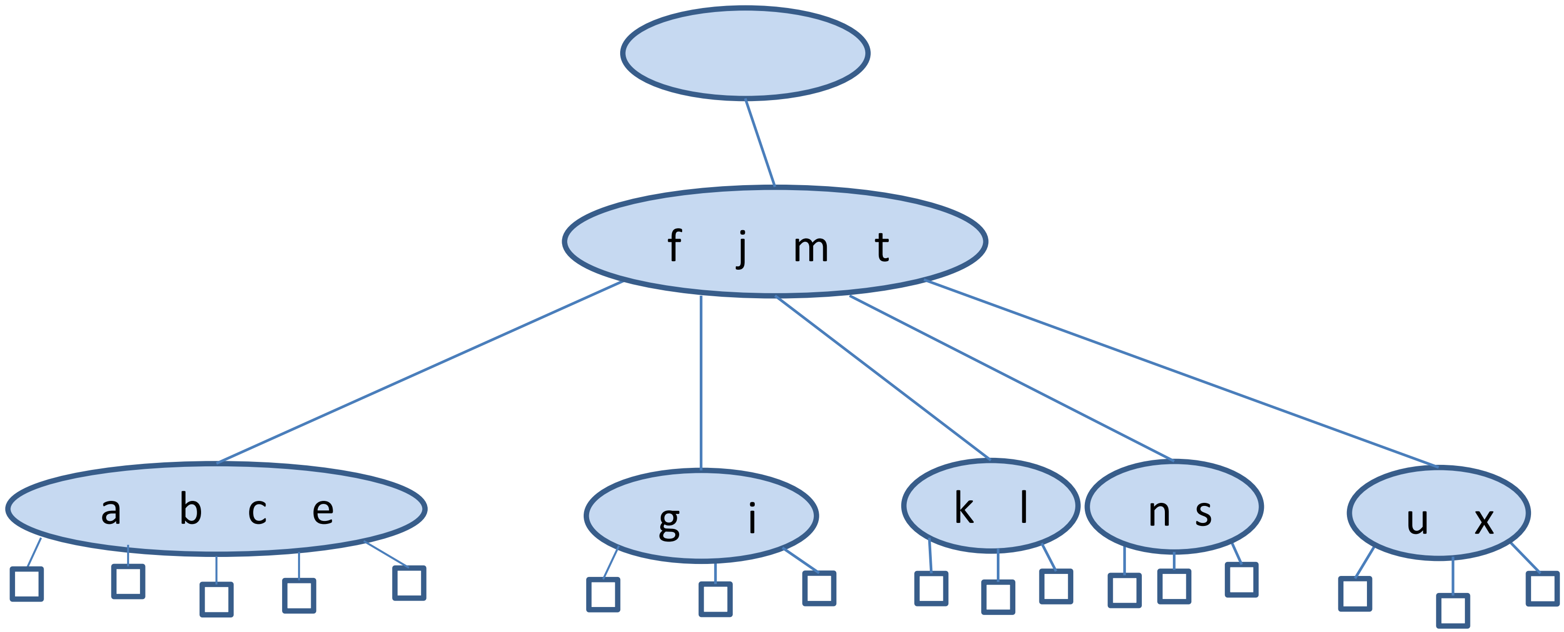
After the Fusion – Underflow at f



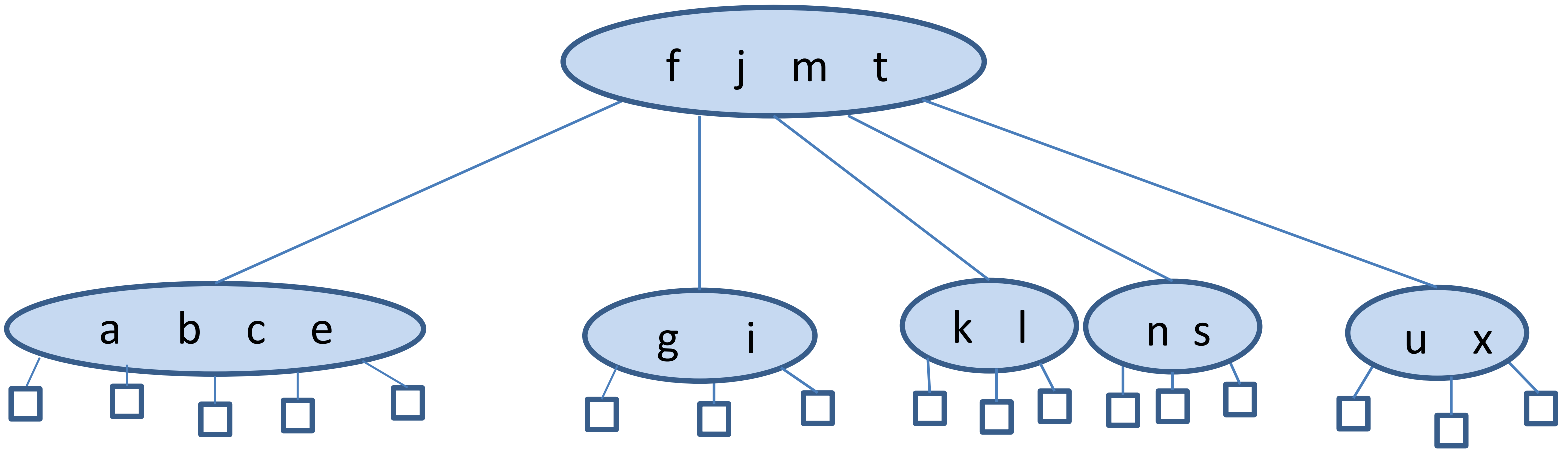
Fusion



After the Fusion – Delete Root



Final Tree



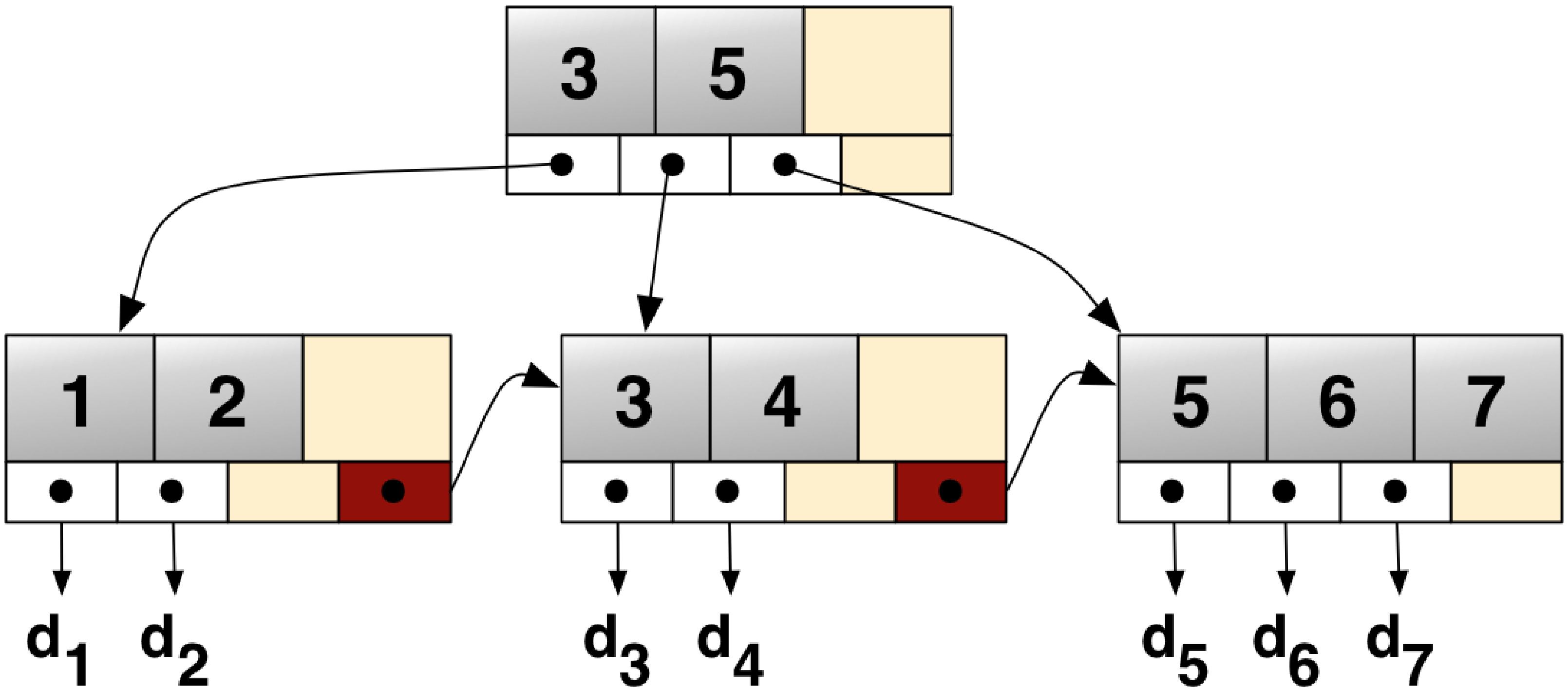
Complexity of Operations in a B-tree

- As we have shown for multi-way trees, the complexity of search, insertion and deletion in a B-tree of order m is $O(ht)$ where $O(t)$ is the time it takes to implement split, transfer or fusion using the data structure implementing each node of the tree.
- If we **count only disk block operations** then $O(t) = O(1)$. Therefore, the complexity of each operation is $\boldsymbol{O(h) = O(\log_{\lceil \frac{m}{2} \rceil} n)}$.

B^+-trees

- A variation of B-trees called **B^+-trees** is one of the most important indexing structures used in today's **file systems and relational database management systems**.

B+-tree Example



B^+-trees (cont'd)

- B^+-trees are similar to B-trees. But in B^+-trees, **internal nodes store only keys** while **external nodes at the bottom layer store keys and pointers to values** (the d_i's in the previous slide).
- The external nodes in the bottom layer are **ordered and linked** so that, not only **equality queries** (e.g., find employees with salary 10,000), but also **range queries** can be answered effectively (e.g., find employees with salary between 10,000 and 20,000 euros).

Readings

- M. T. Goodrich, R. Tamassia and D. Mount. Data Structures and Algorithms in C++. 2nd edition. John Wiley.
- M. T. Goodrich, R. Tamassia. *Δομές Δεδομένων και Αλγόριθμοι σε Java*. 5η έκδοση. Εκδόσεις Δίαυλος.
  - Κεφ. 10.5
- Sartaj Sahni. Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++. Εκδόσεις Τζιόλα.

Readings (cont'd)

- You can also see the following chapter but notice that the data structure called B-tree there is essentially a B^+-tree (but without the linking of the external nodes on the bottom layer):
  - R. Sedgewick. Αλγόριθμοι σε C.
    - Κεφ. 16.3