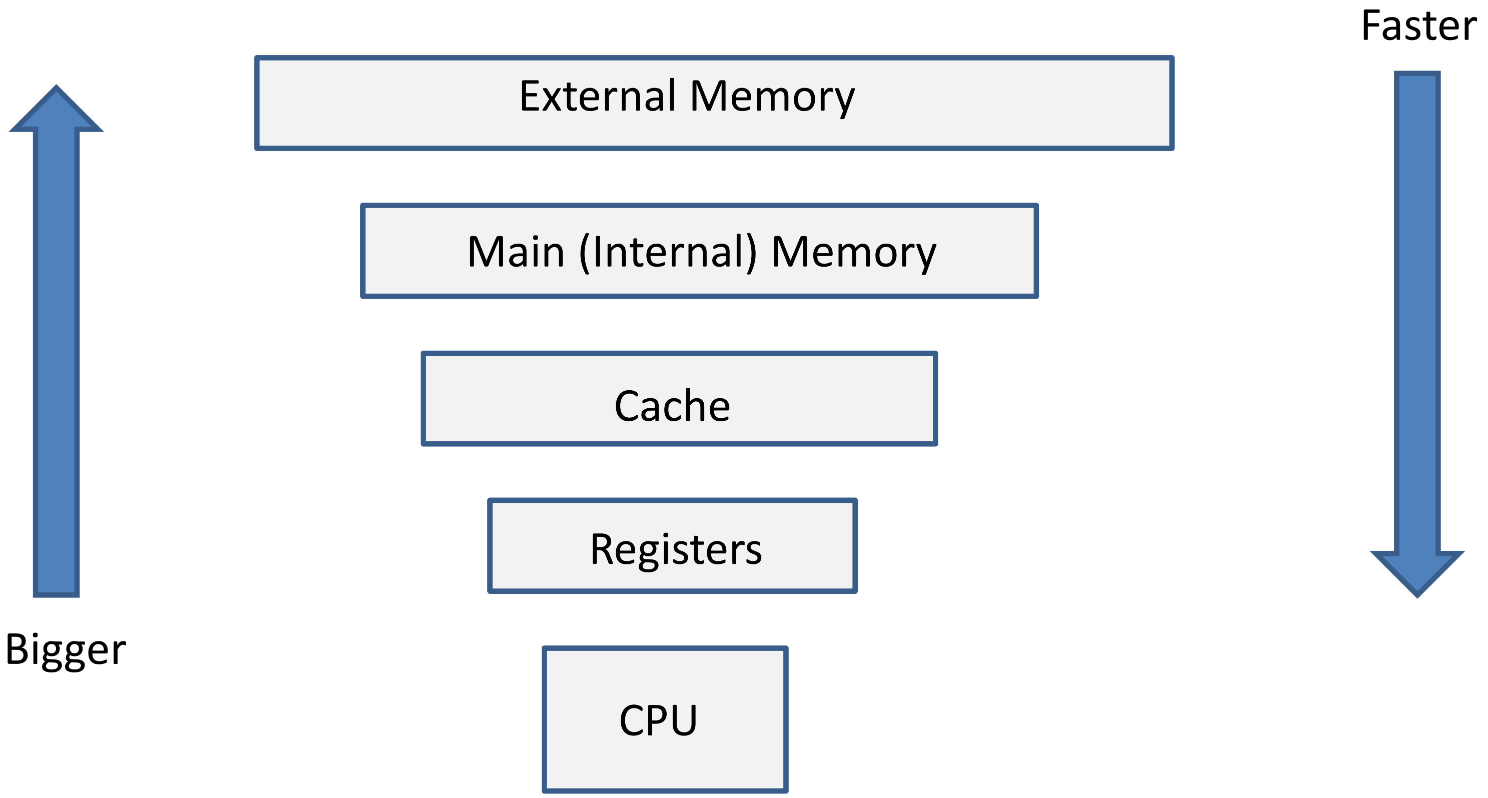
B-Trees

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

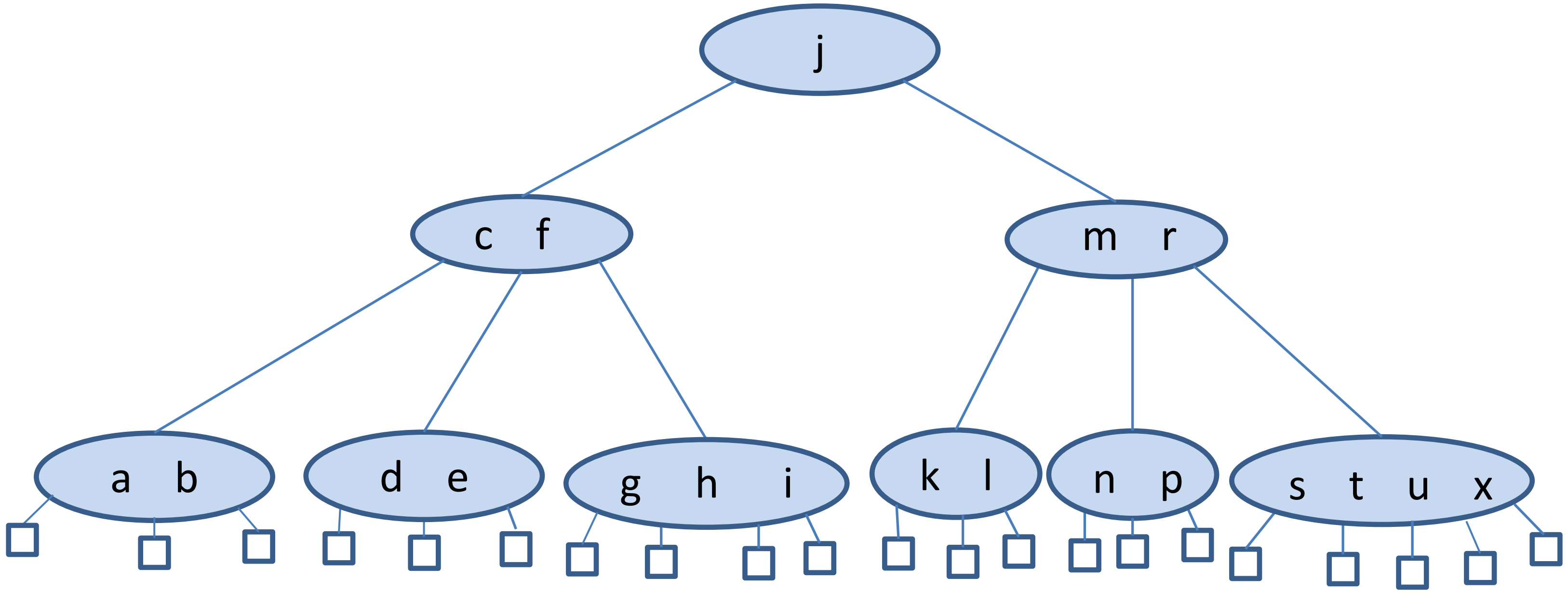
- When we access data on a disk or another external memory device, we perform **external searching (εξωτερική αναζήτηση).**
- A **disk access** can be at least 100,000 to 1,000,000 times longer than a main memory access.
- Thus, for data structures residing on disk, we want to **minimize disk accesses**.

(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, at least 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:
$$h \log a - \log a + 1 \leq \log(n+1)$$
$$h \log a \leq \log(n+1) + \log a - 1$$
$$h \leq \frac{1}{\log a} \log \frac{n+1}{2} + 1 - \frac{1}{\log a}$$
$$h < \frac{1}{\log a} \log \frac{n+1}{2} + 1$$

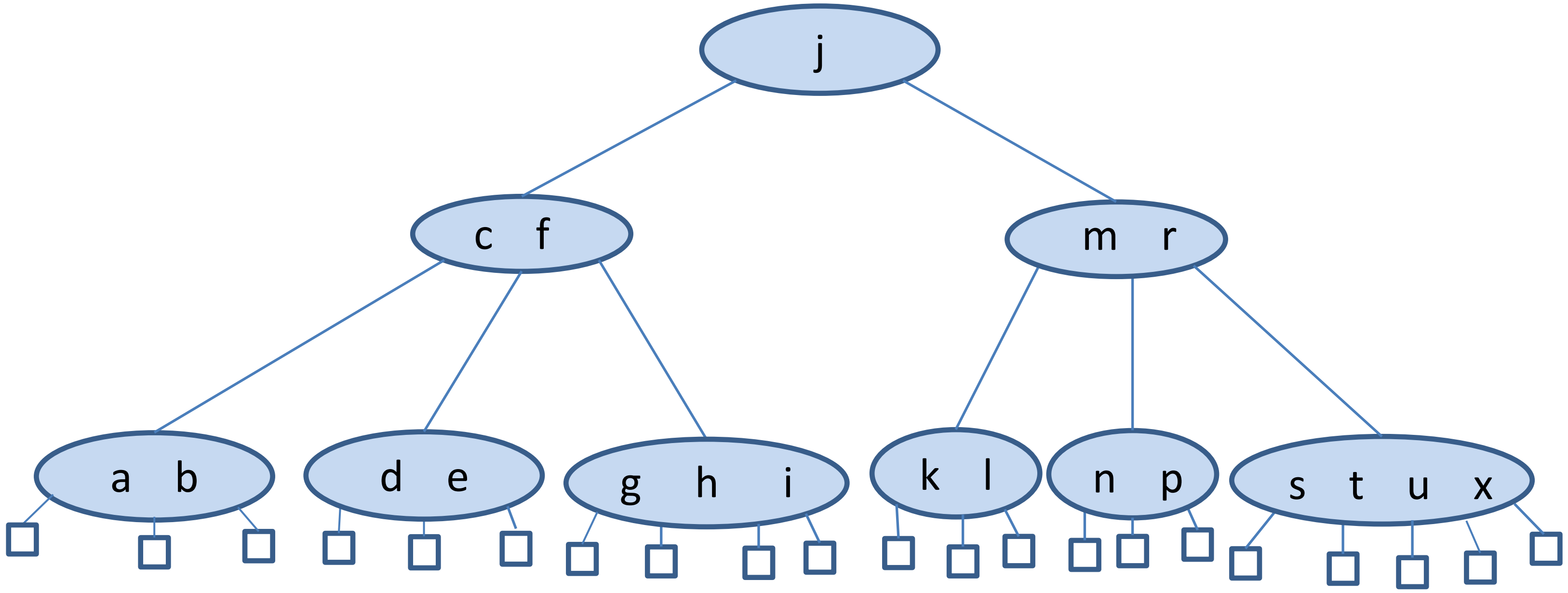
B-Trees

- 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 (B-δένδρο τάξης 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 the inequality $2a^{h-1} \leq n + 1$ which we used in the proof of the previous proposition 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.

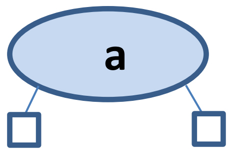
Insertion into a B-tree

- 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.

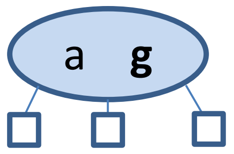
Example

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

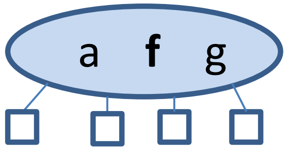
Insert a



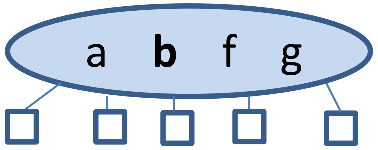
Insert g



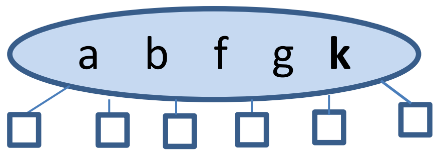
Insert f



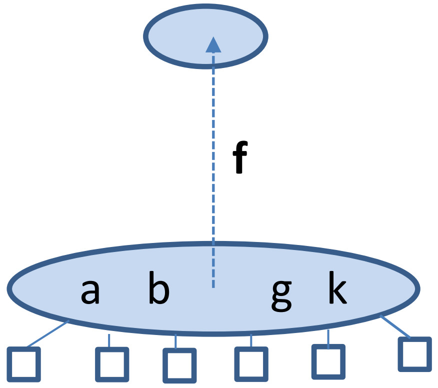
Insert b



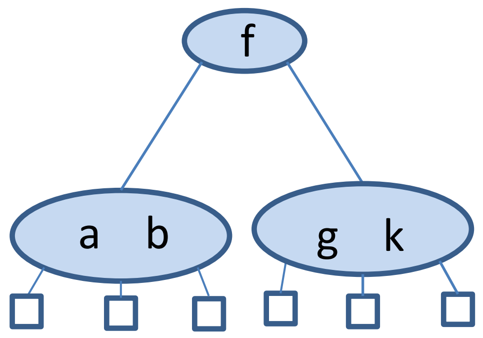
Insert k - Overflow



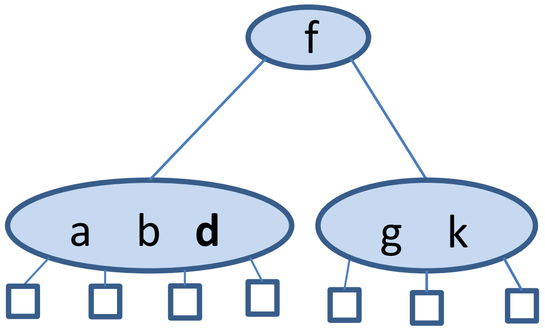
Creation of a New Root Node



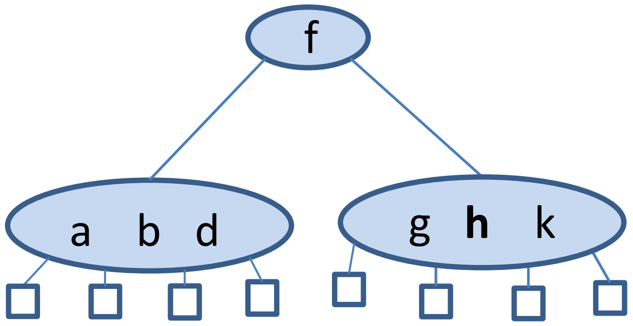
Split



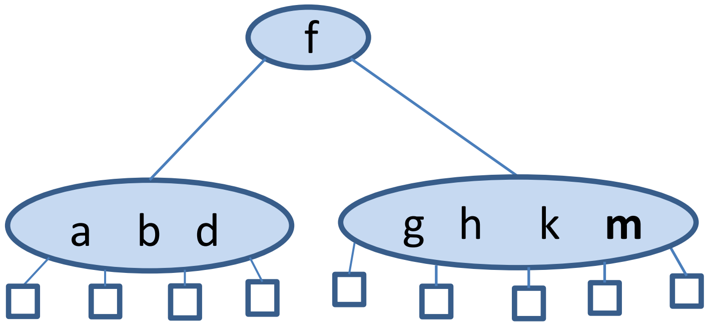
Insert d



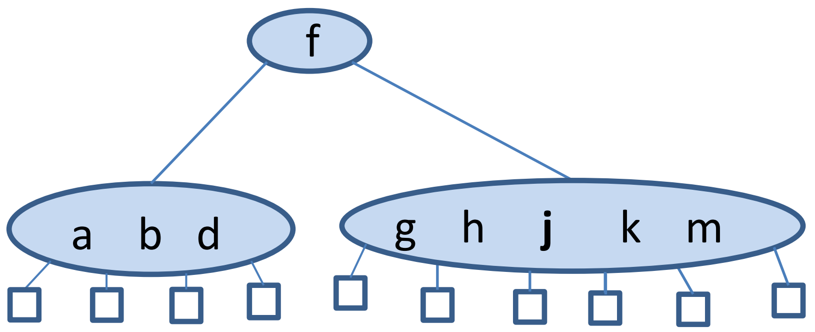
Insert h



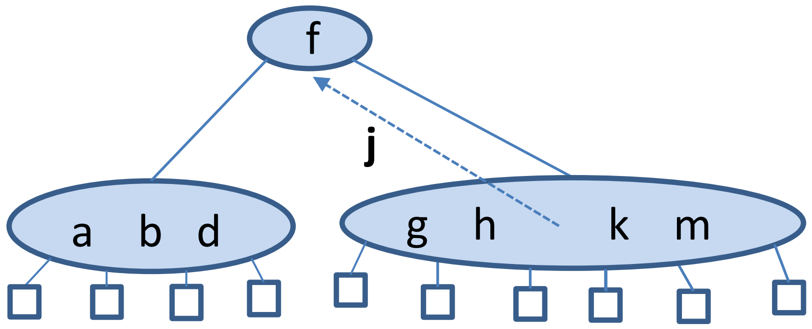
Insert m



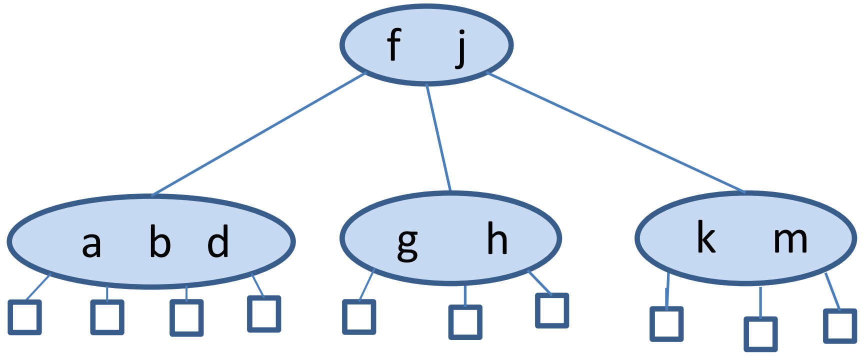
Insert j - Overflow



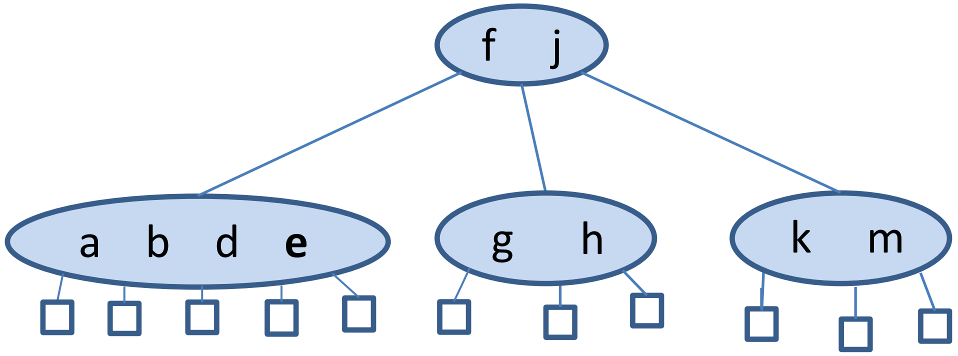
Sent j to the Parent Node



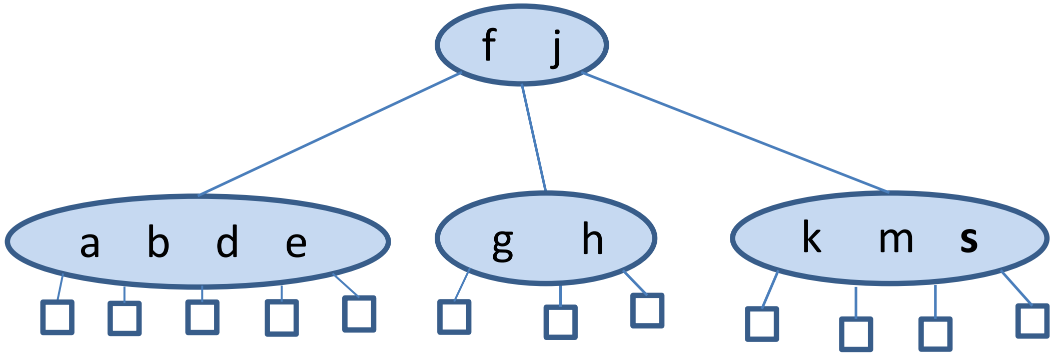
Split



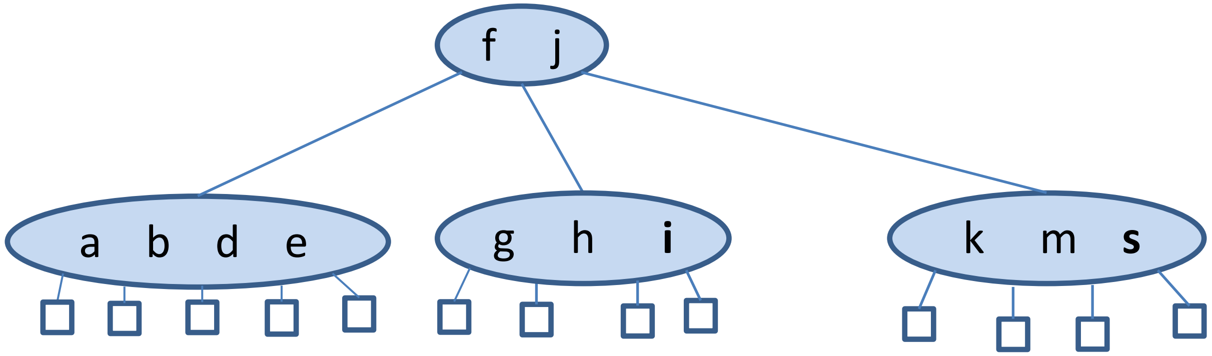
Insert e



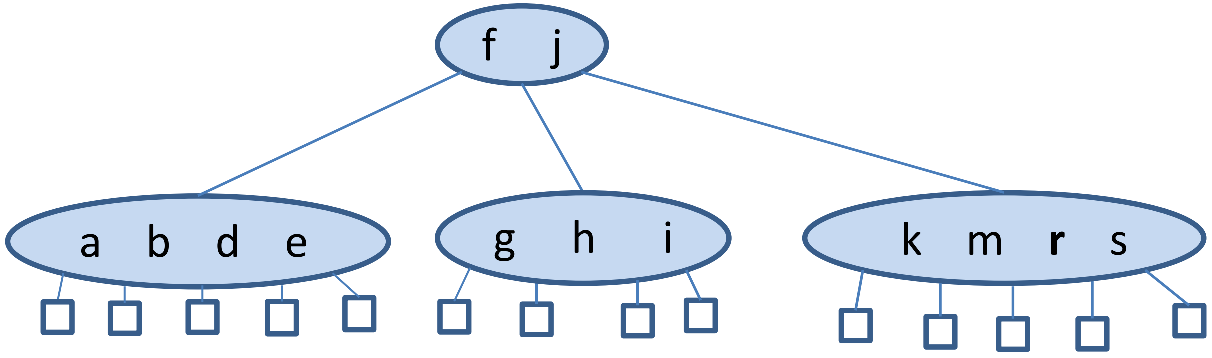
Insert s



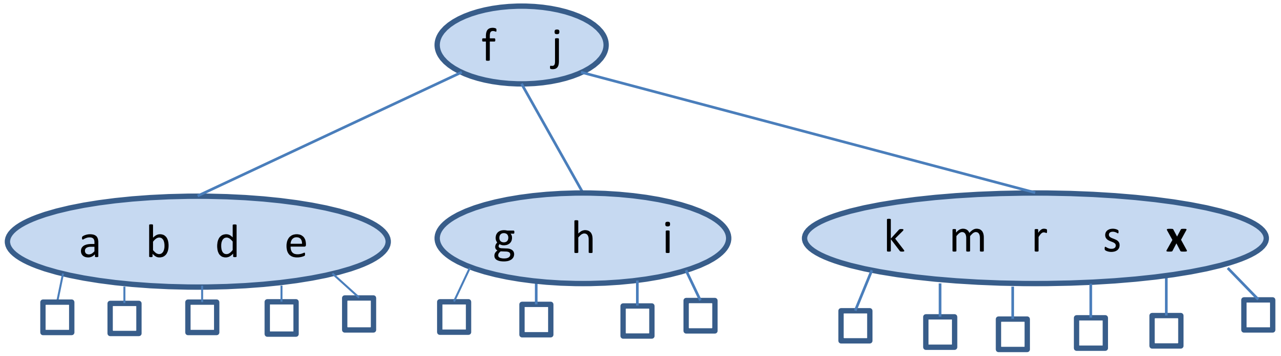
Insert i



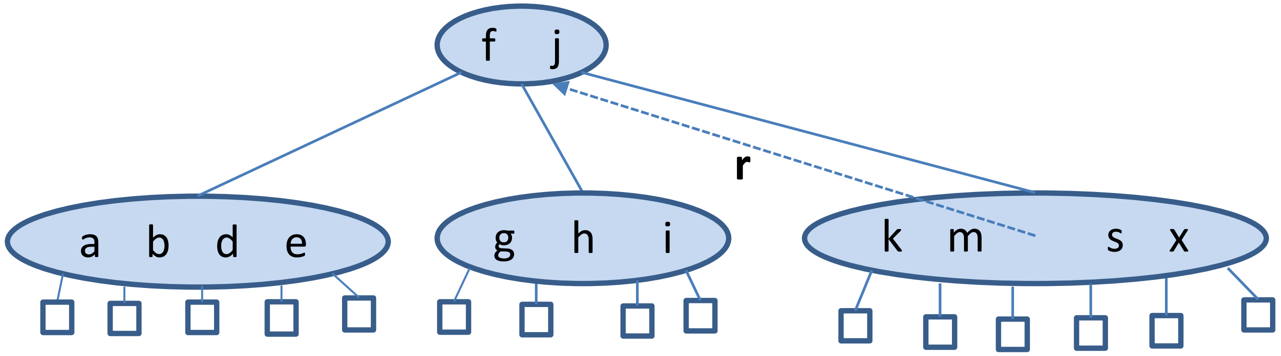
Insert r



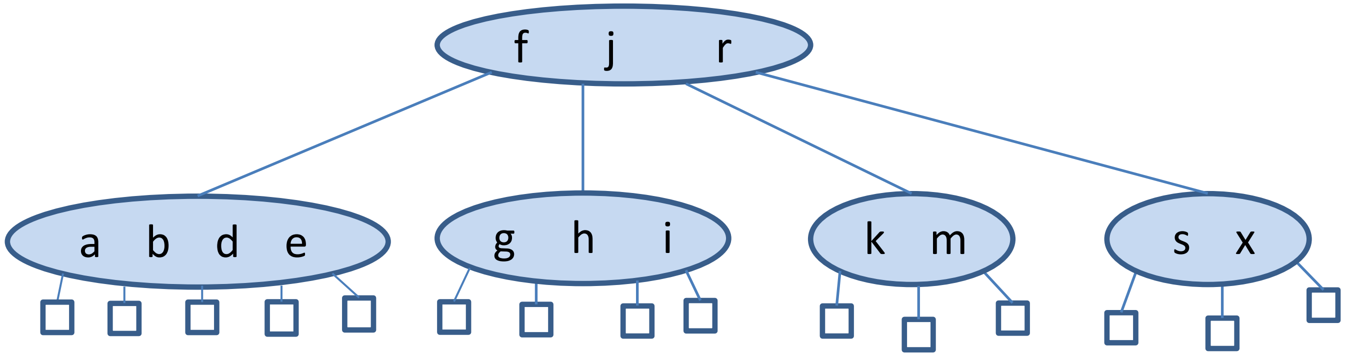
Insert x - Overflow



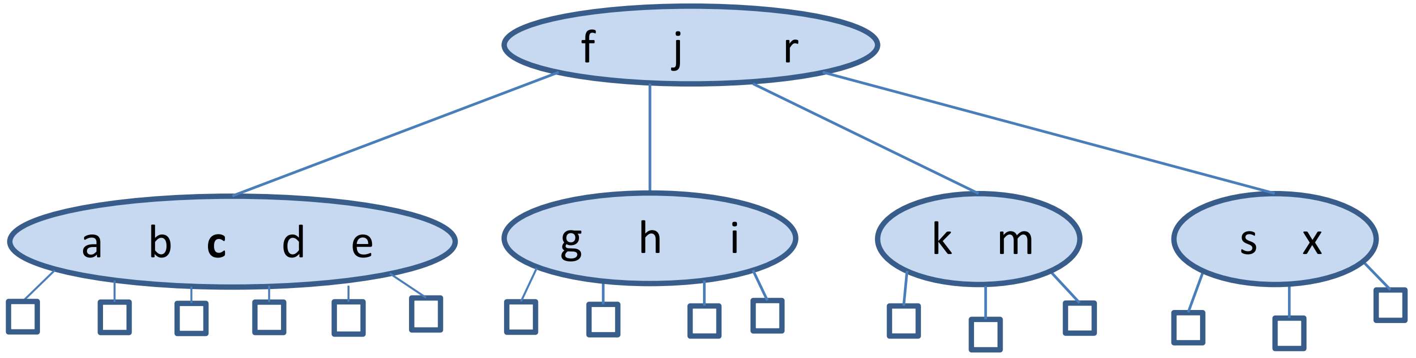
r is Sent to the Parent Node



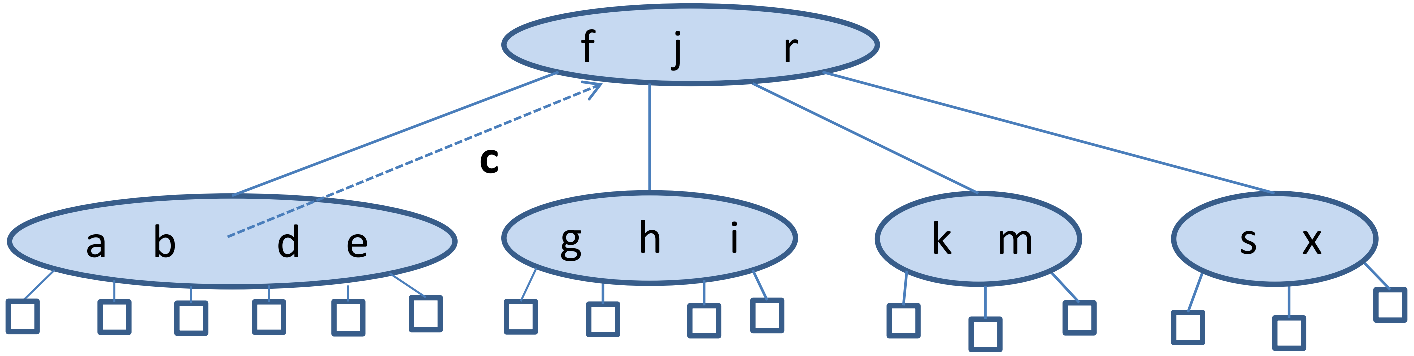
Split



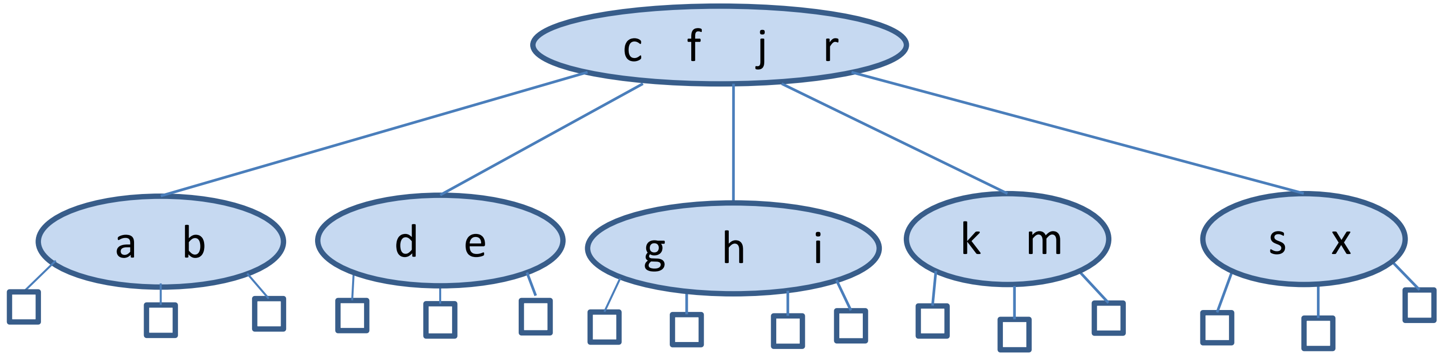
Insert c - Overflow



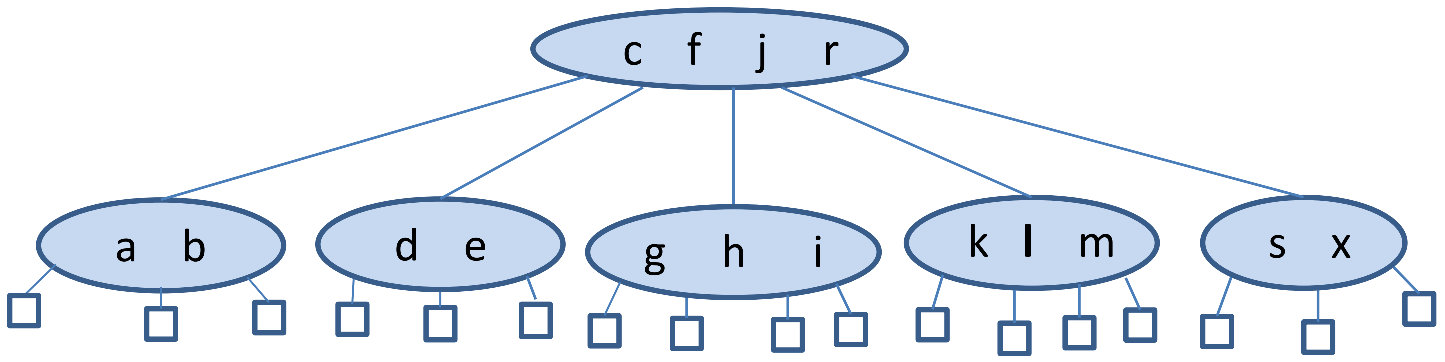
c is Sent to the Parent



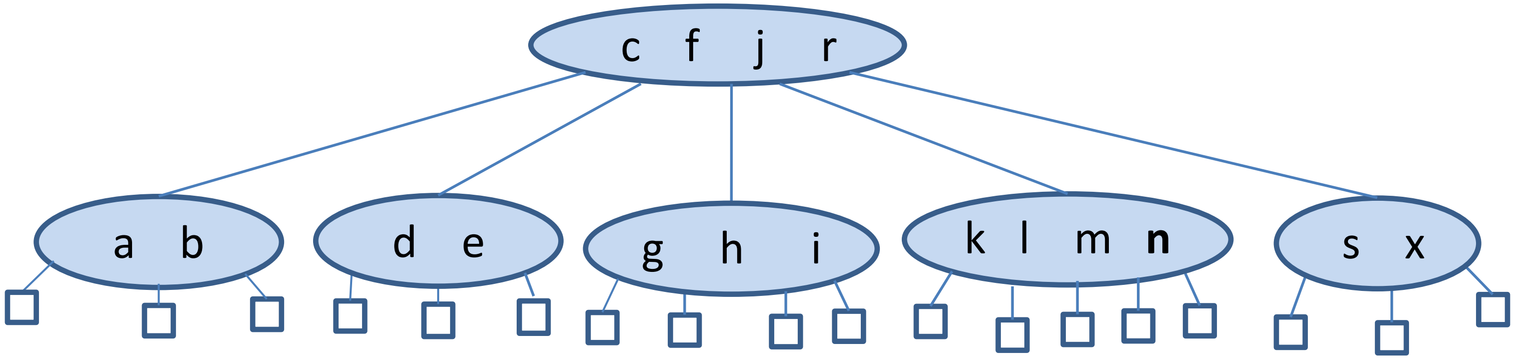
Split



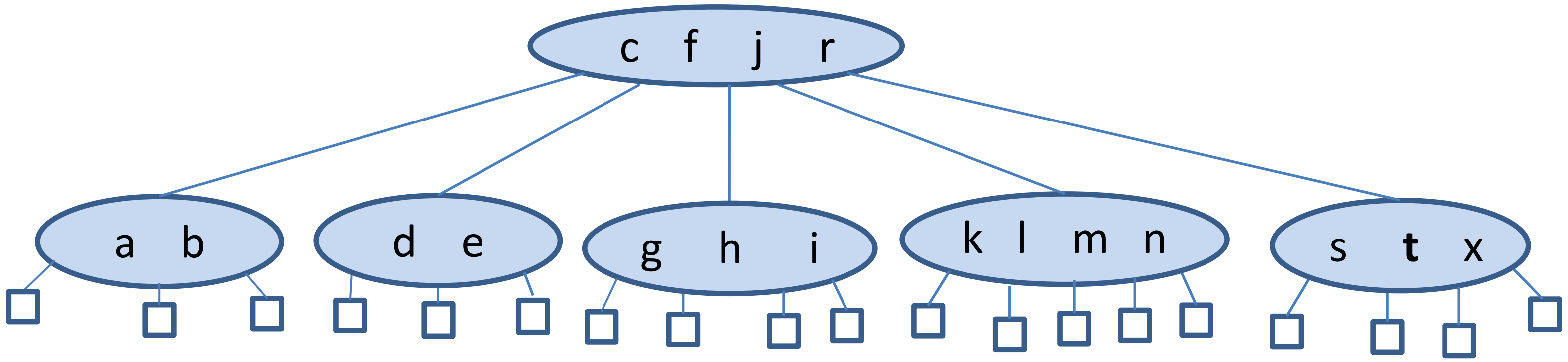
Insert l



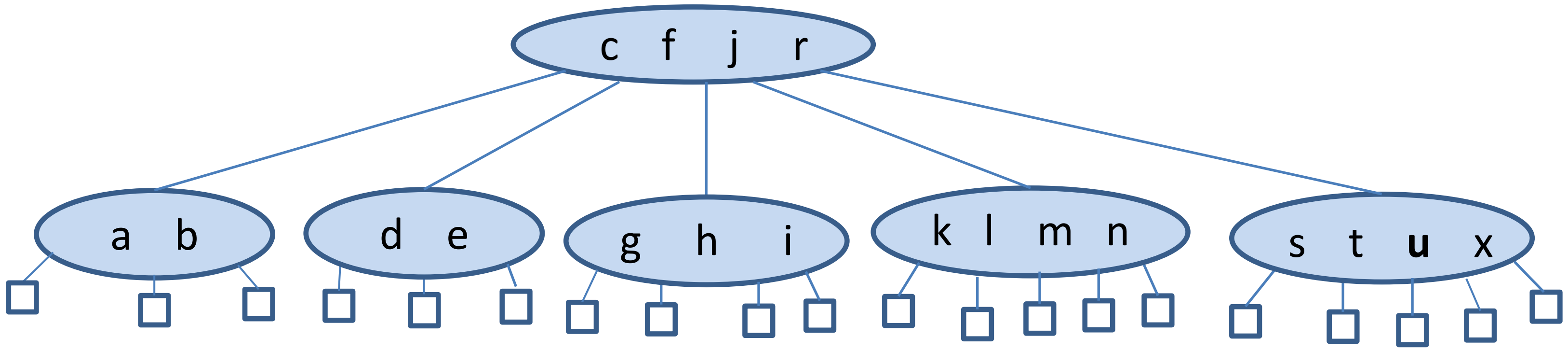
Insert n



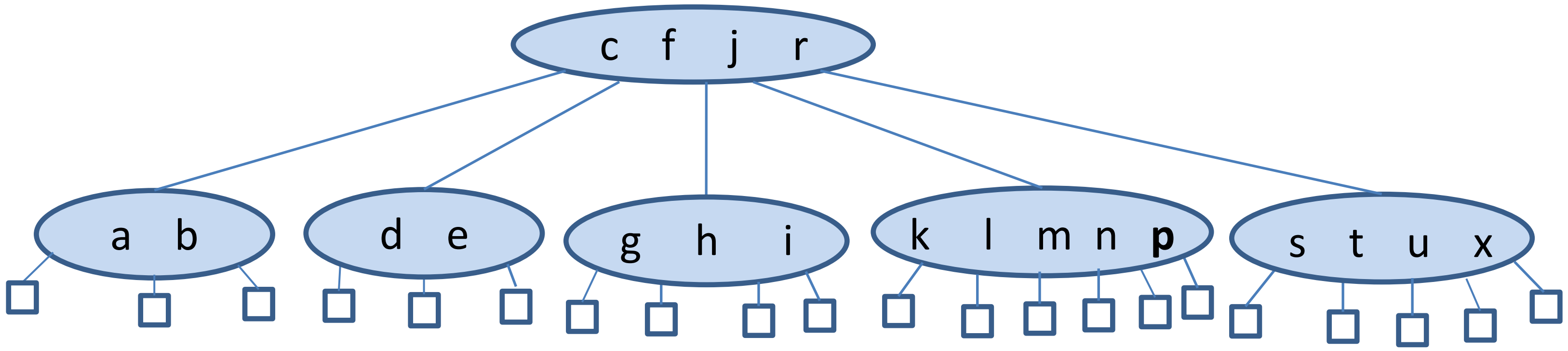
Insert t



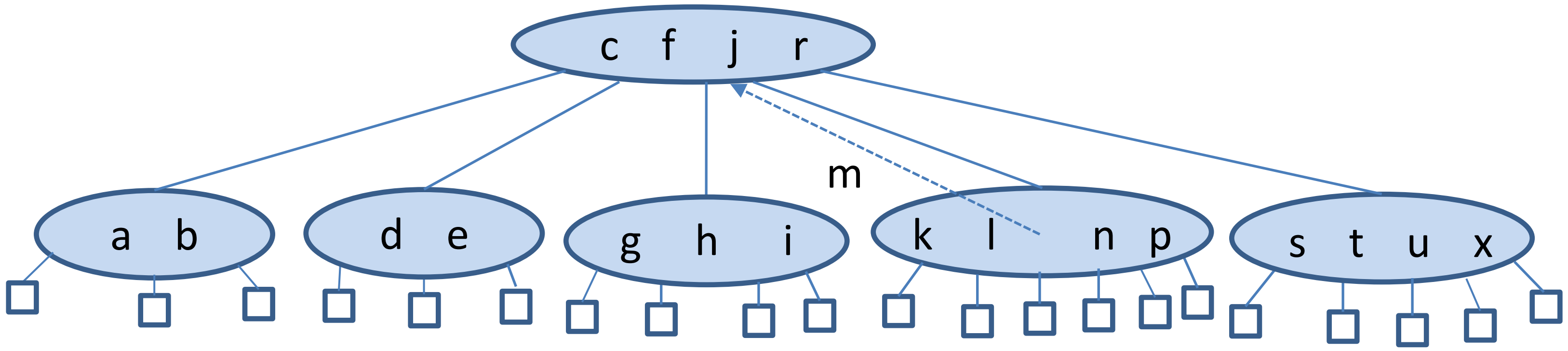
Insert u



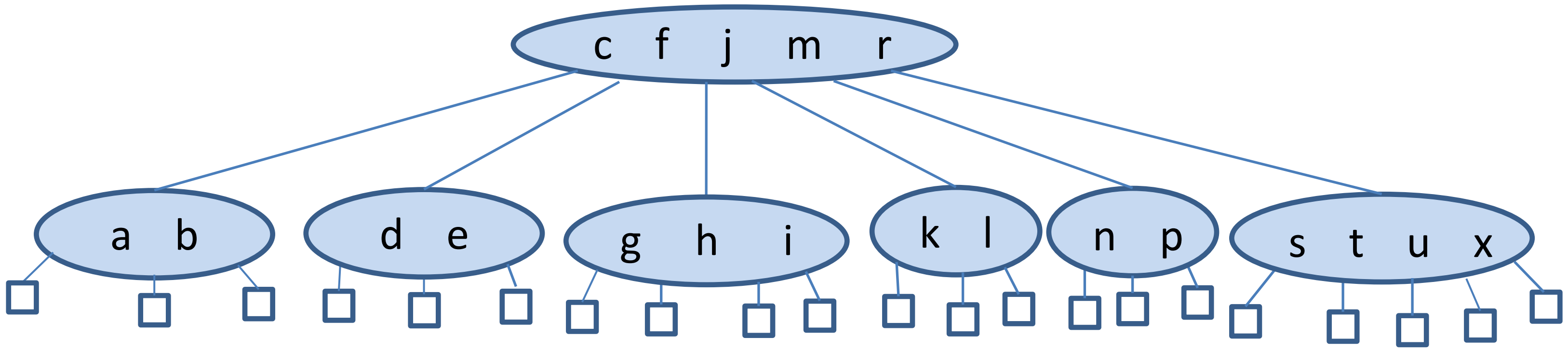
Insert p - Overflow



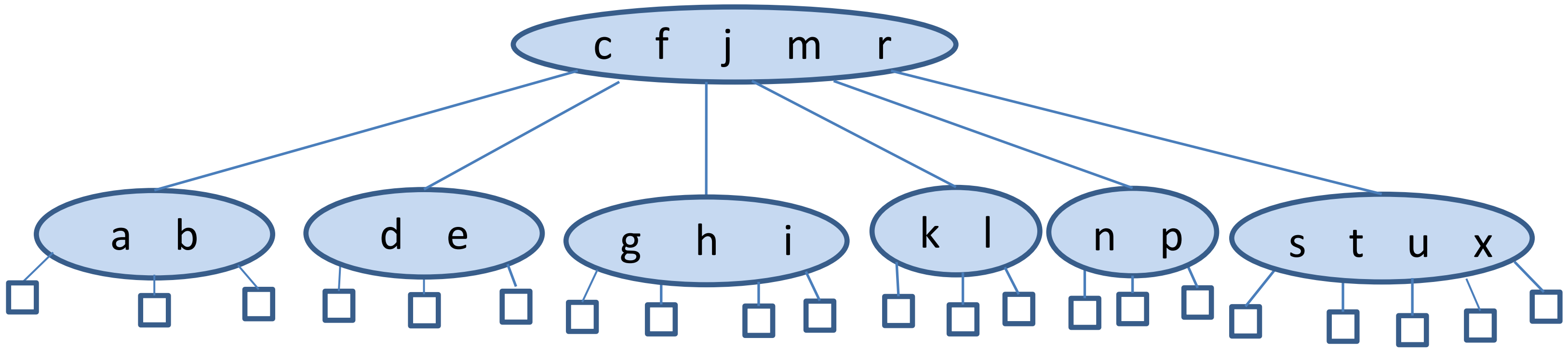
m is Sent to the Parent Node



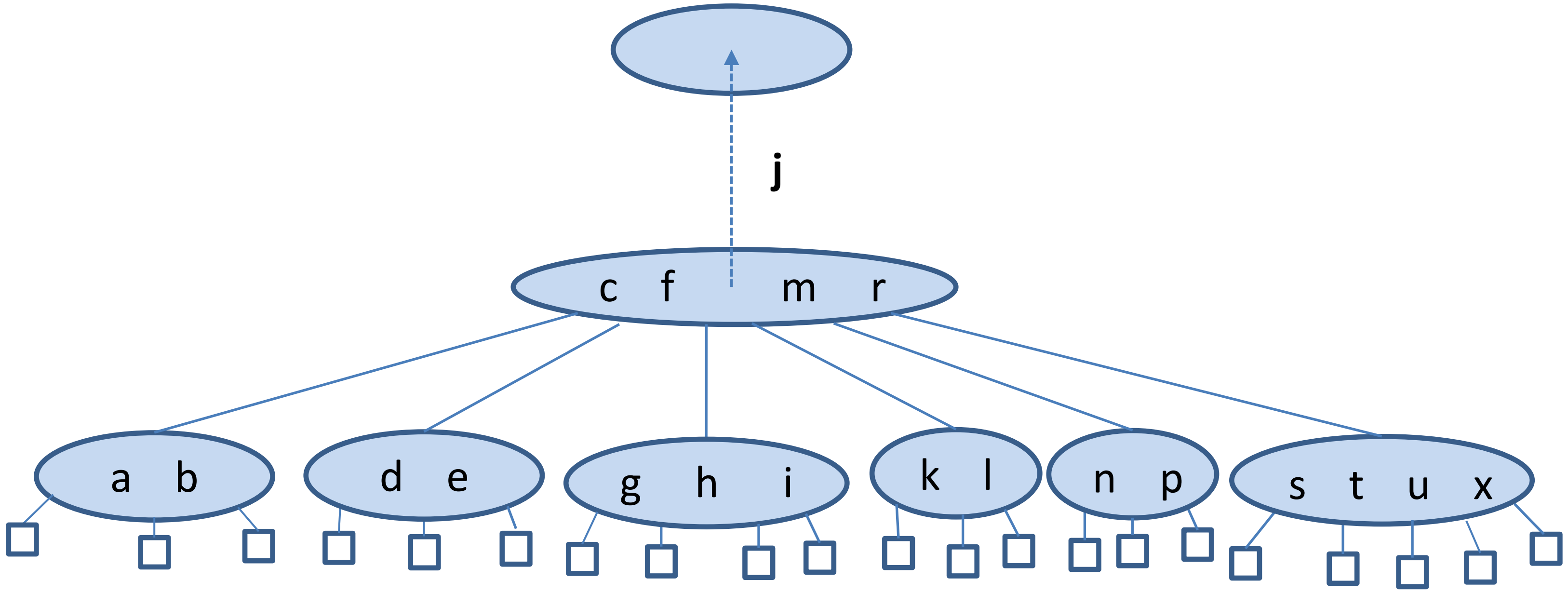
Split



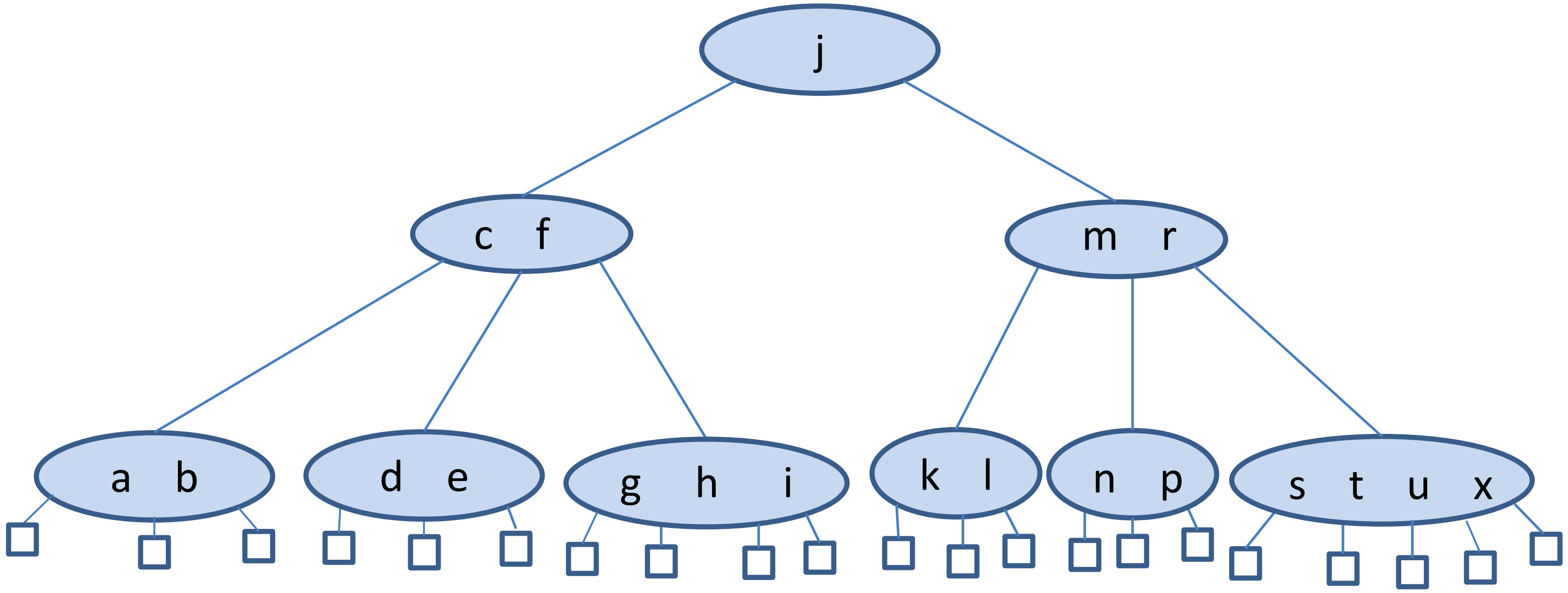
Overflow at the Root



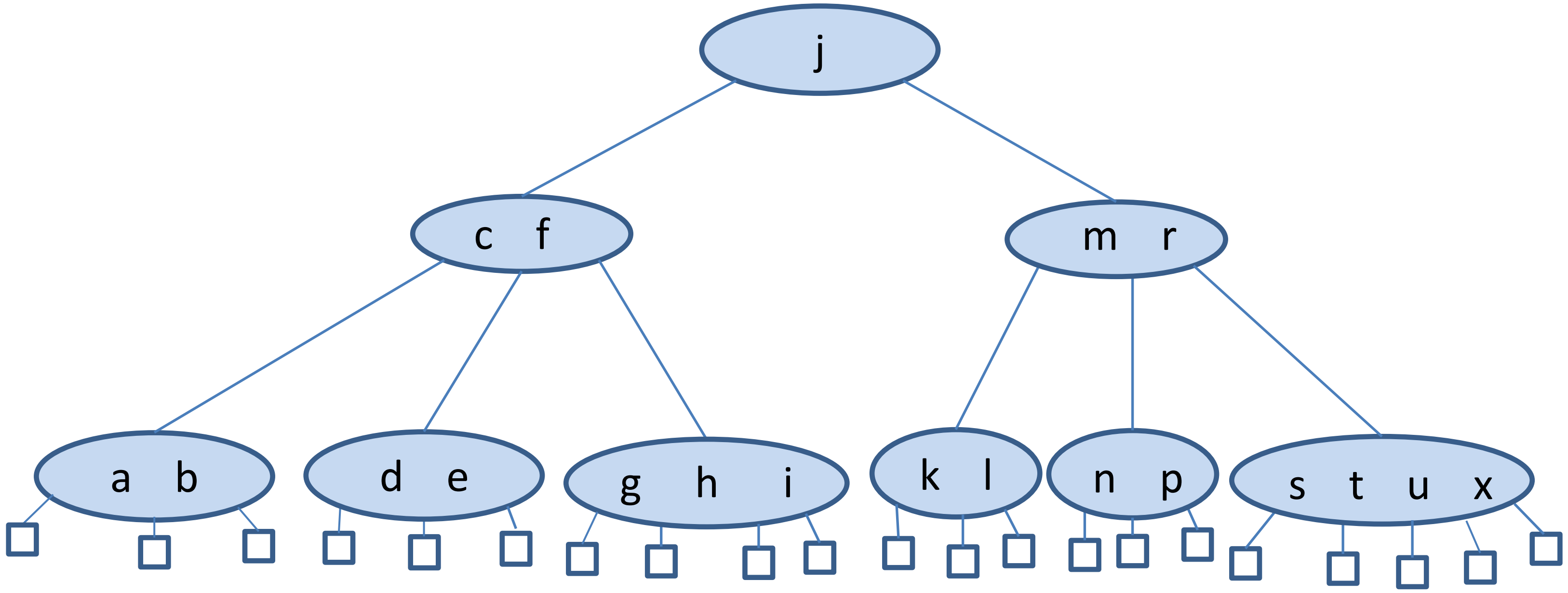
j is Sent up to a New Root



Split



Final Tree



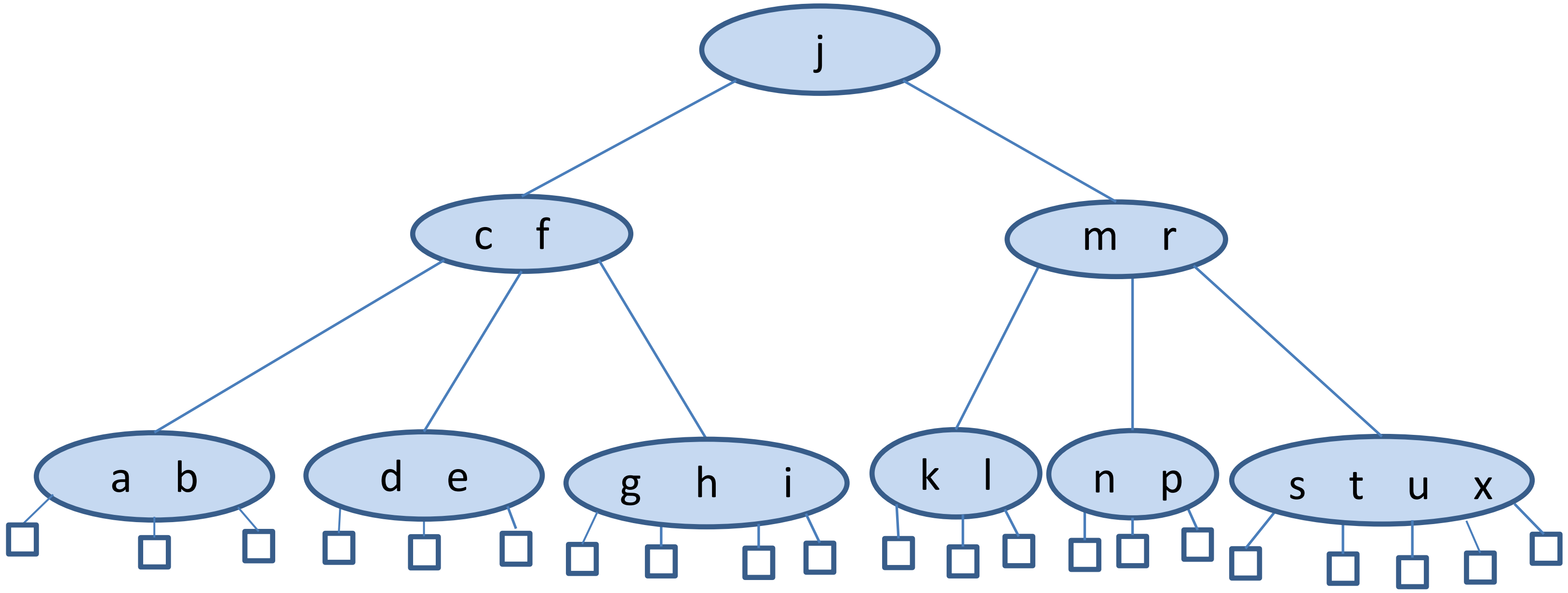
Deletion from a B-tree

- 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.

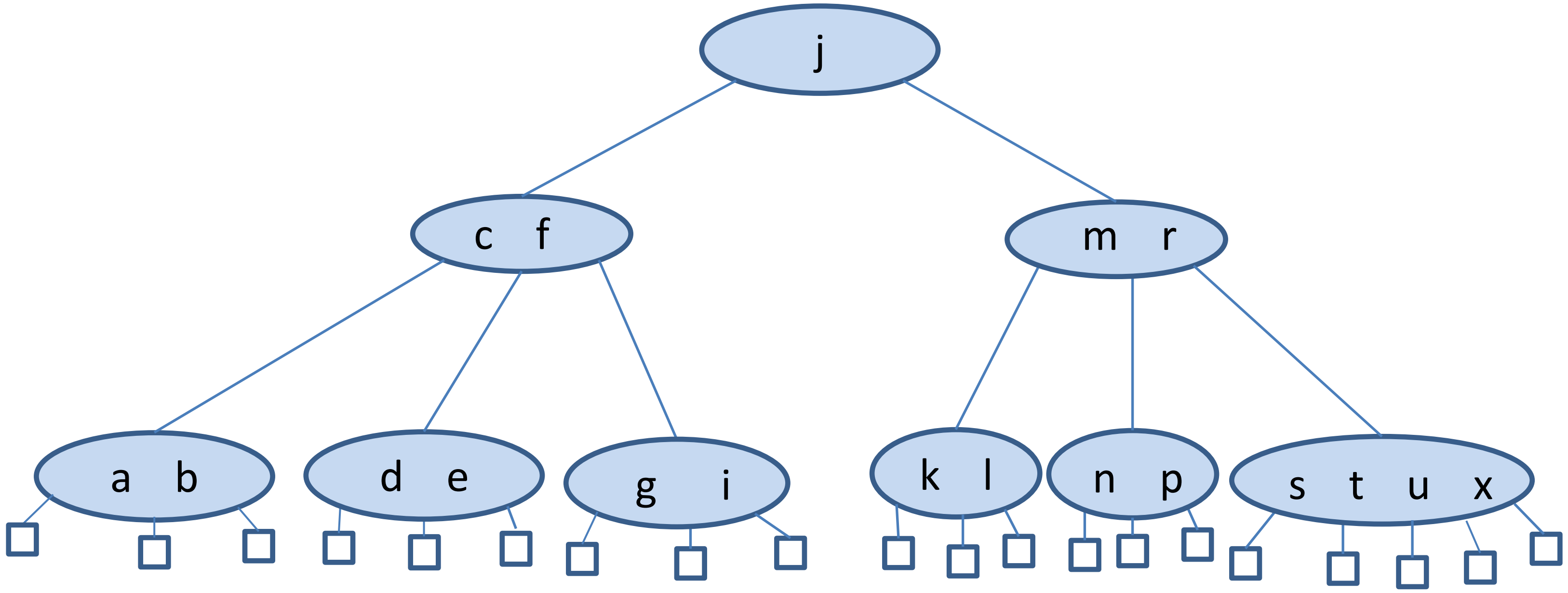
Deletion from a B-tree (cont'd)

- 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.**

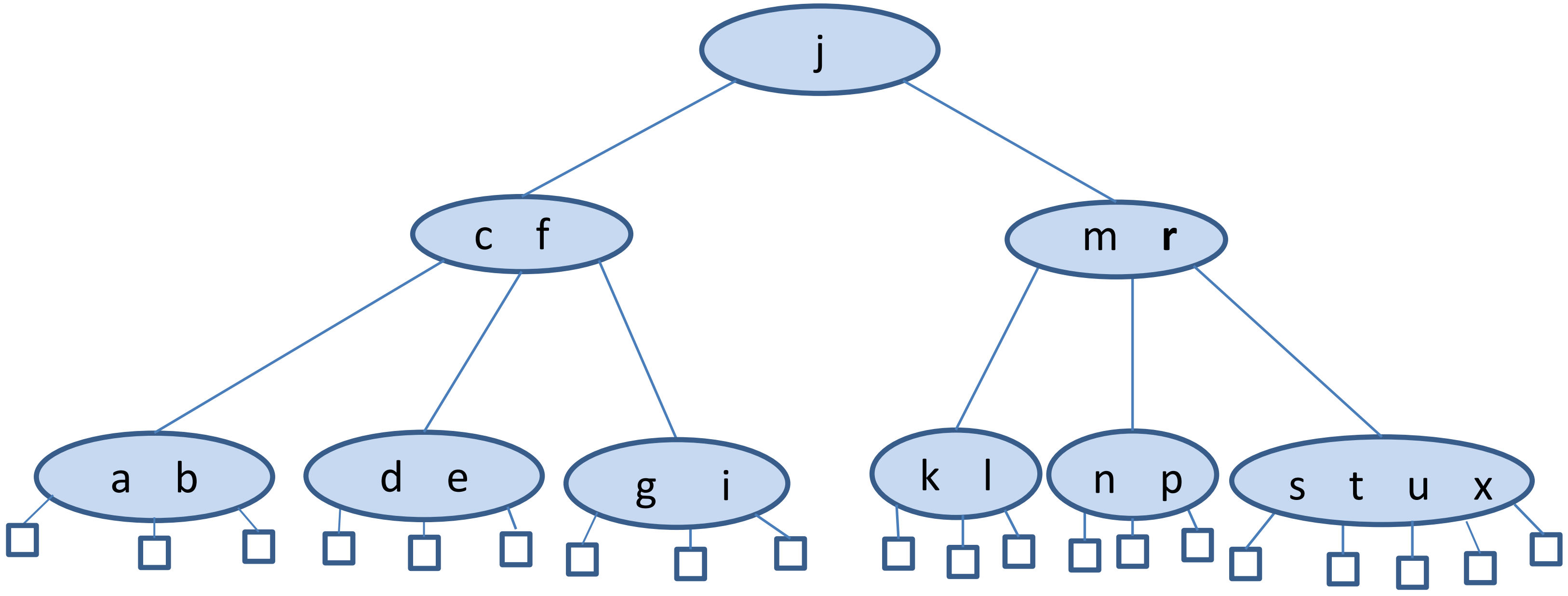
Example



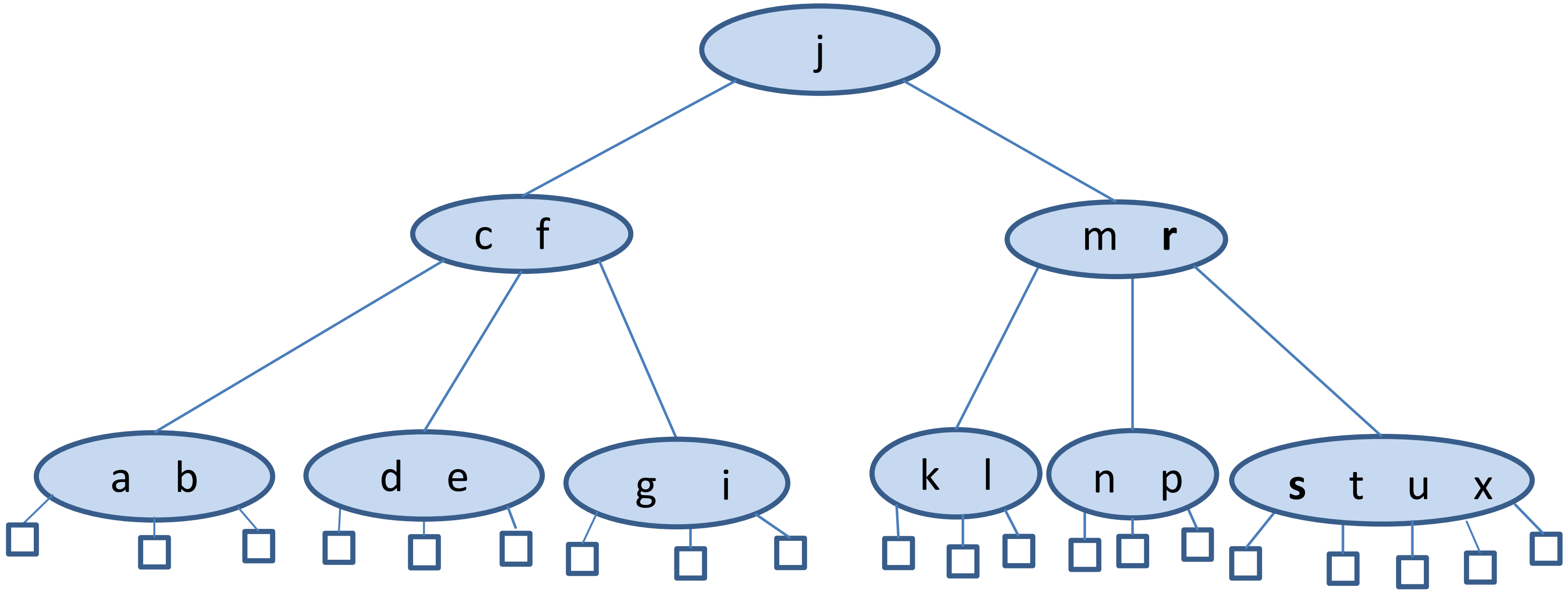
Delete h



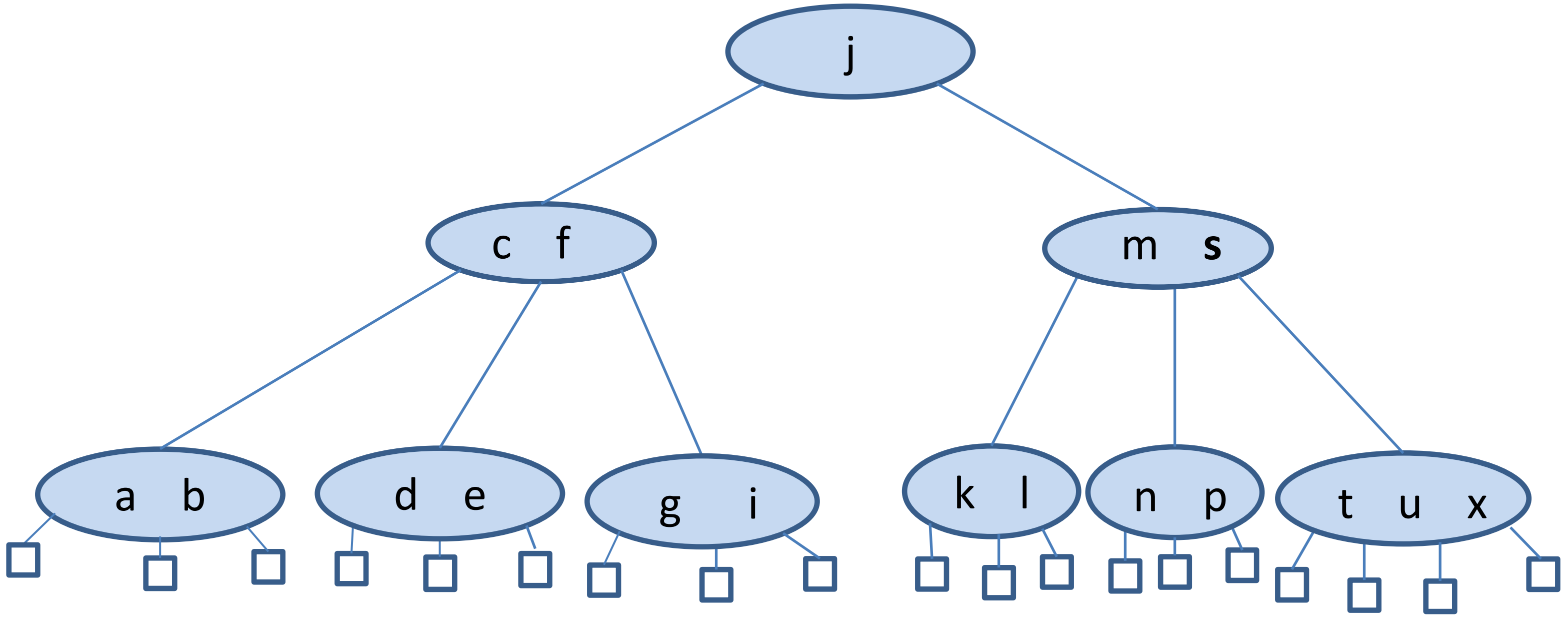
Delete r



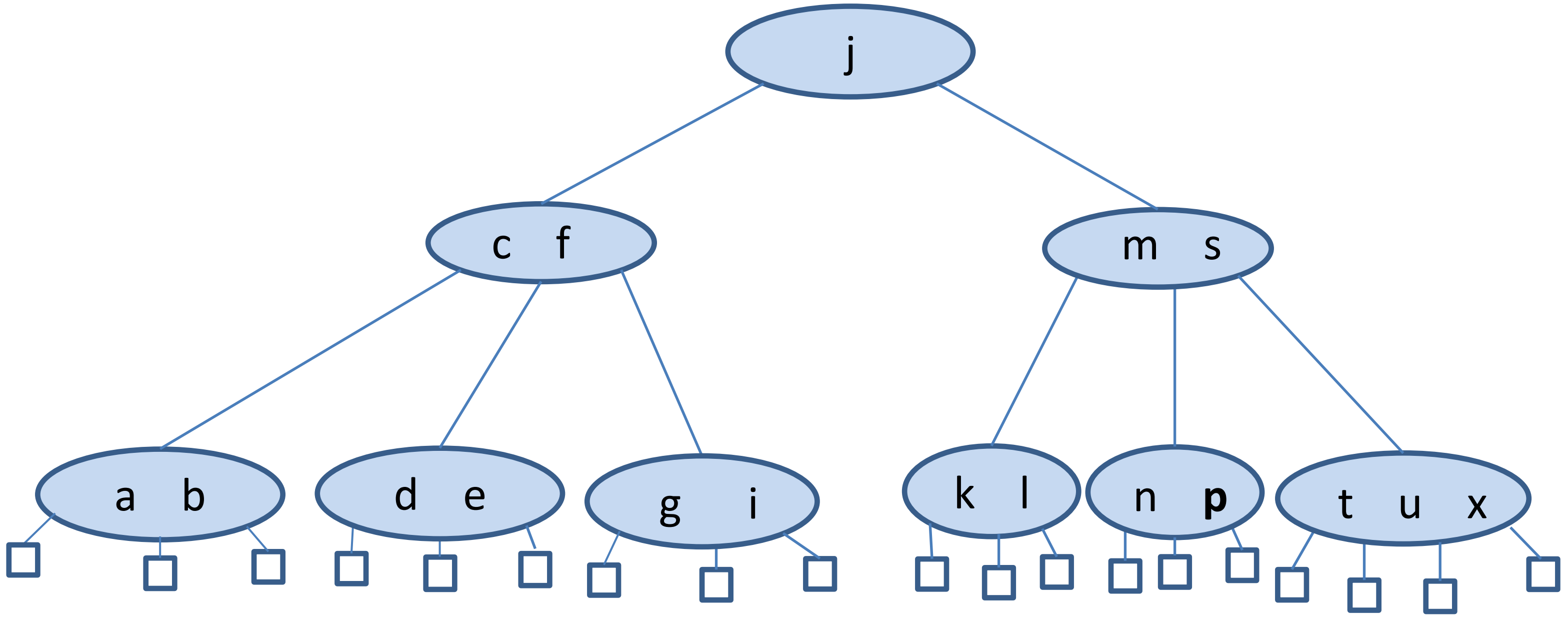
Find the Successor of r



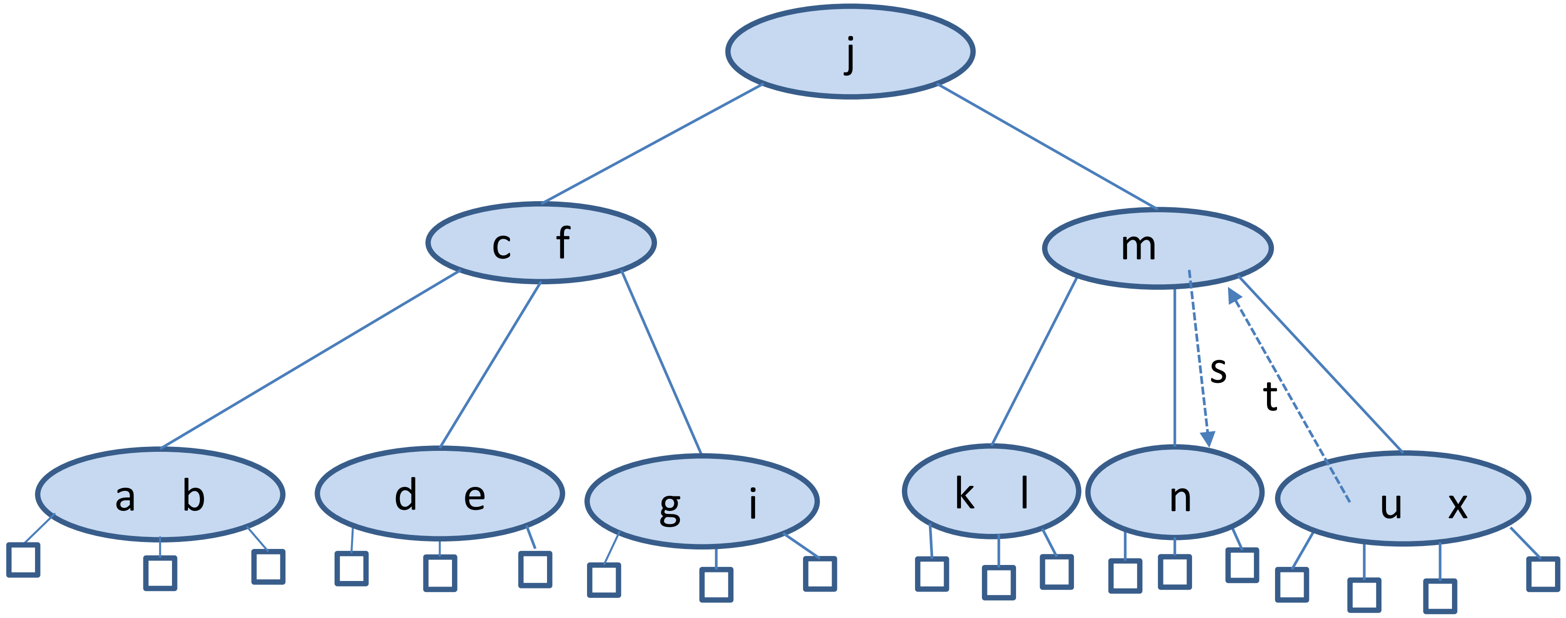
Promote the Successor of r – Delete the Successor from its Place



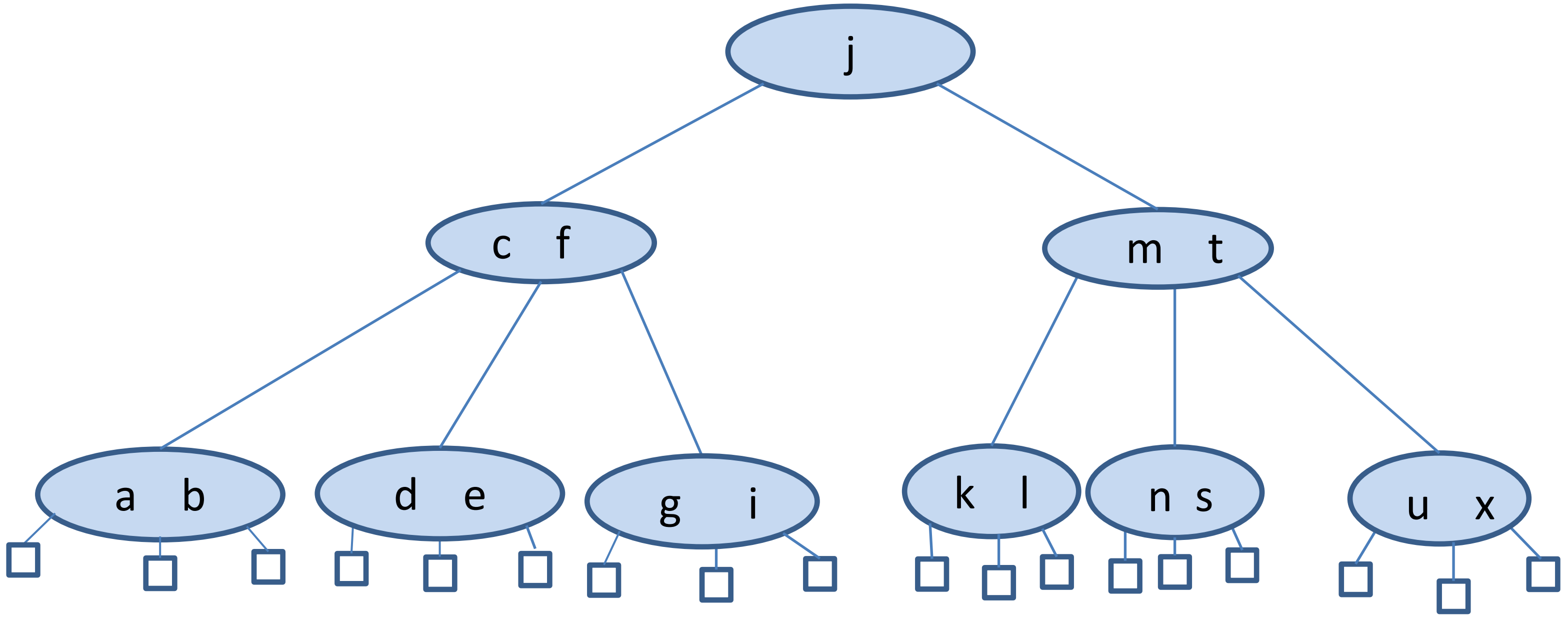
Delete p



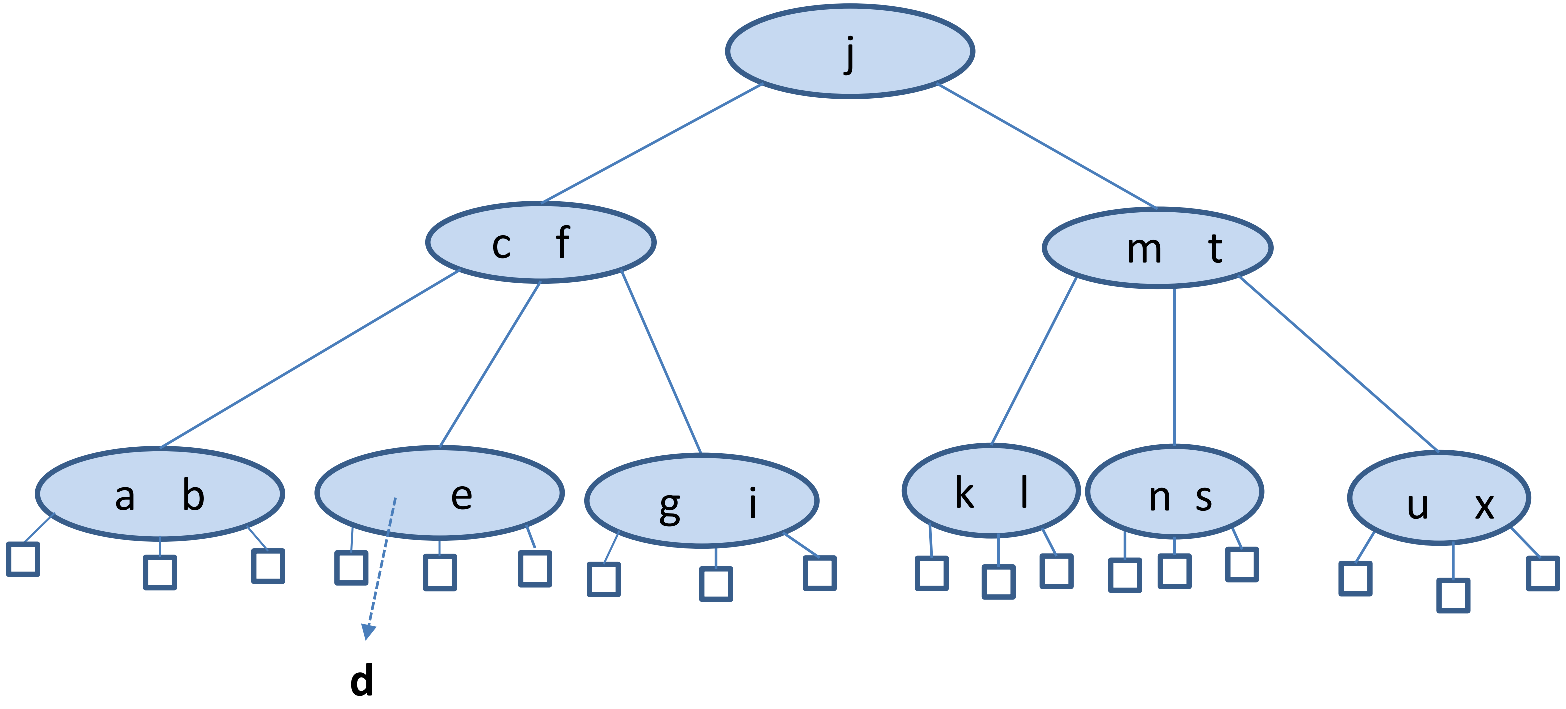
Transfer



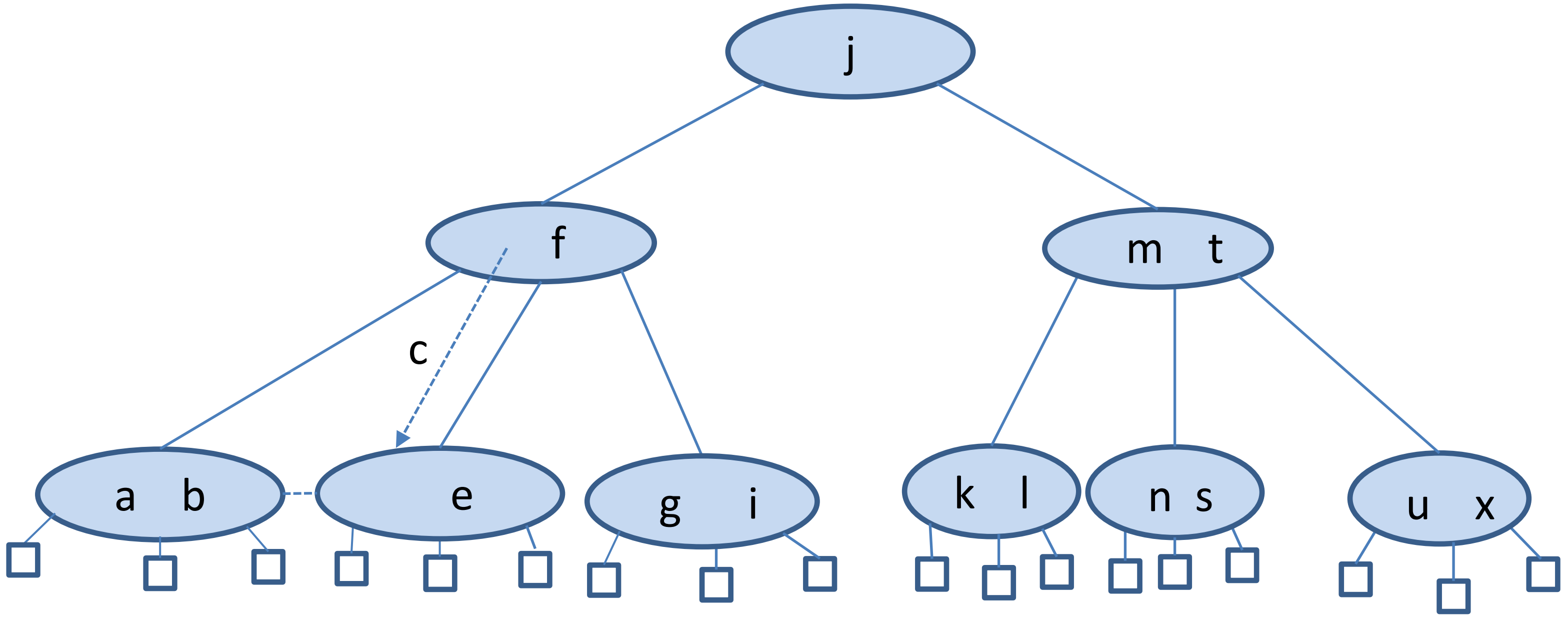
After the Transfer



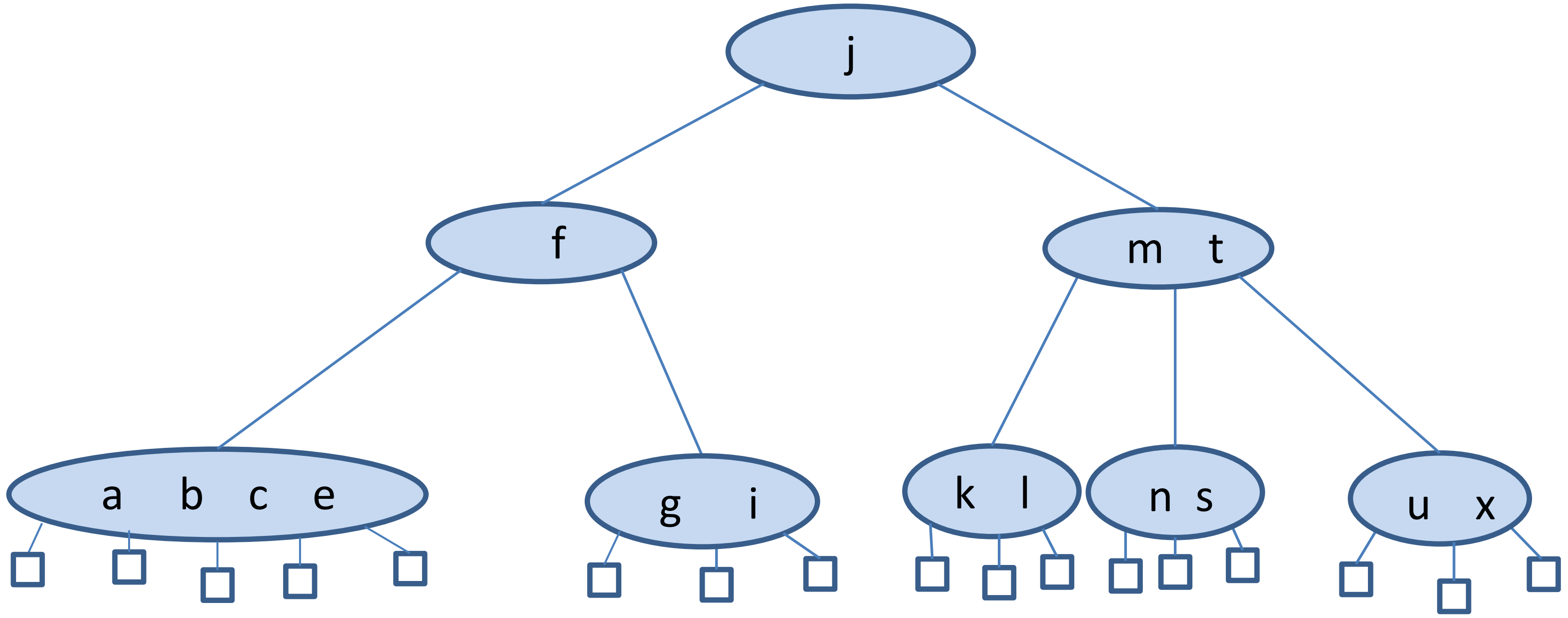
Delete d



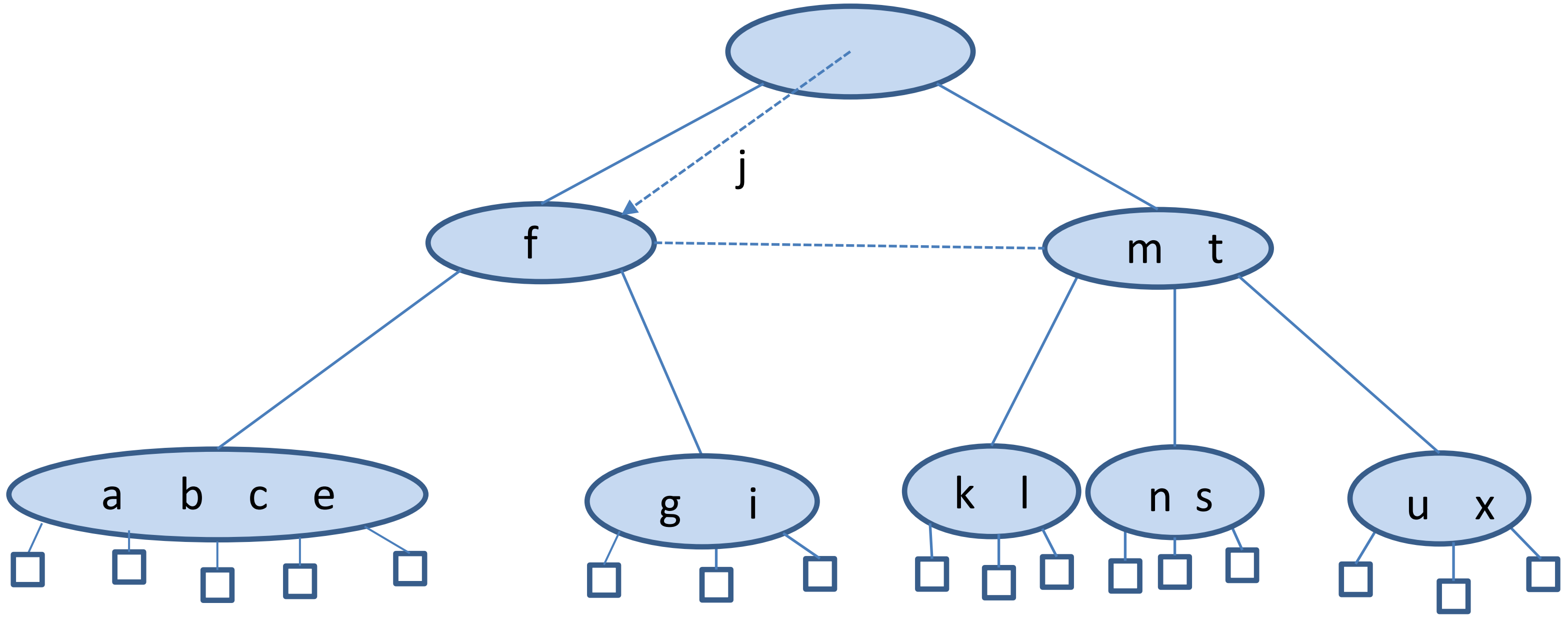
Fusion



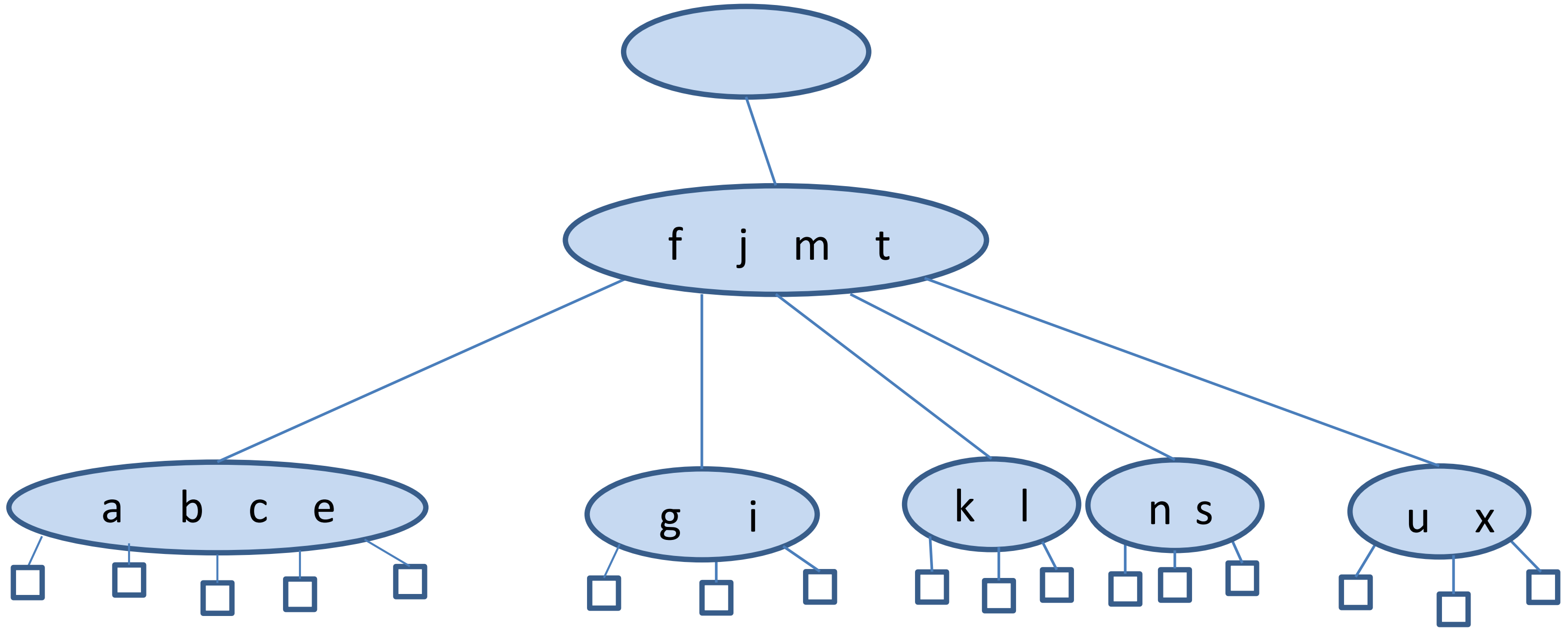
After the Fusion – Underflow at f



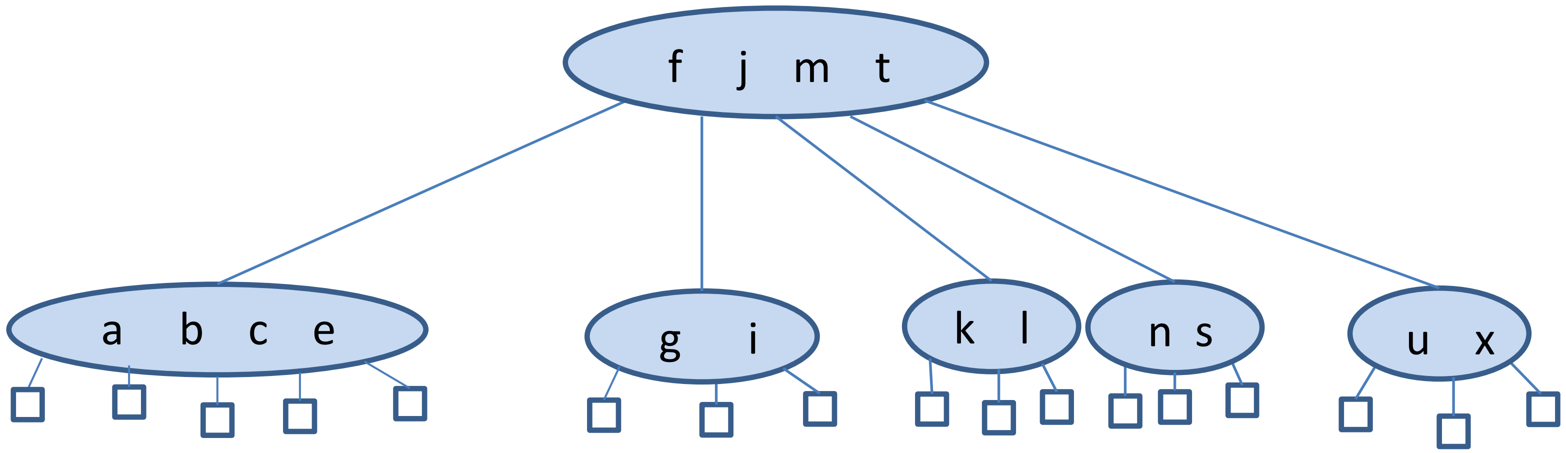
Fusion



After the Fusion – Delete Root



Final Tree



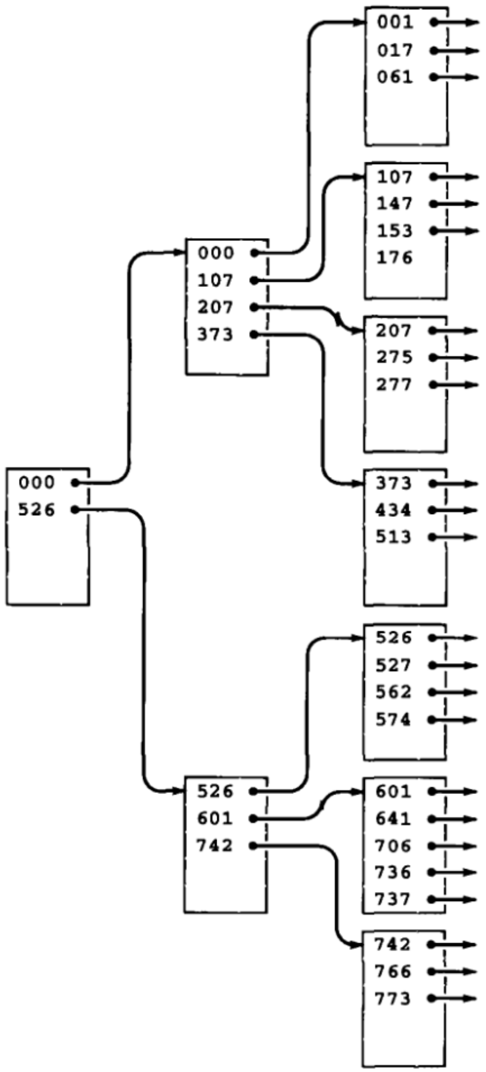
Complexity of Operations in a B-tree

- 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



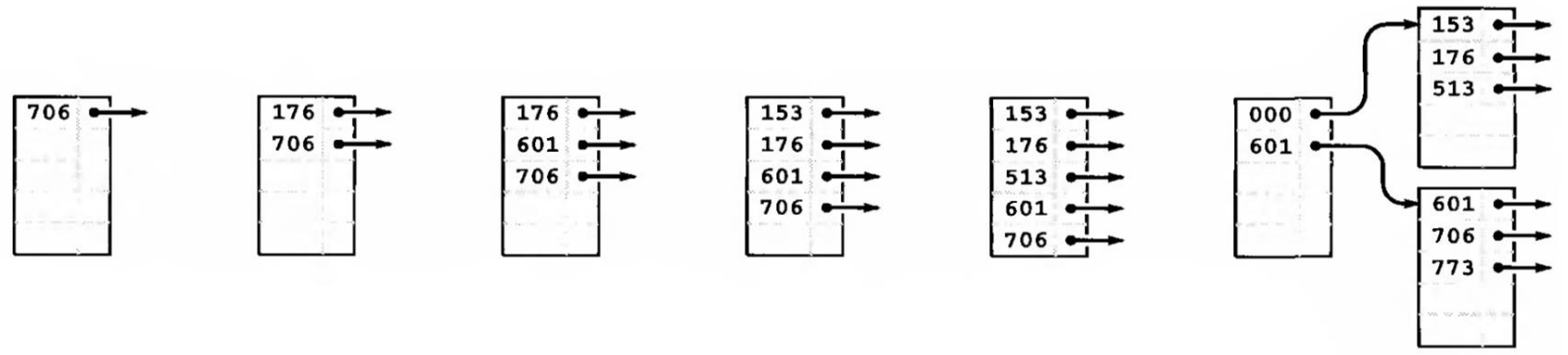
B$^+$-trees (cont'd)

- B$^+$-trees are similar to B-trees. But in B$^+$-trees, **internal nodes store only keys (they are index nodes)** while **external nodes at the bottom layer store keys and pointers to values** (the pointers are the arrows and the values are not shown).
- 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).
- The linking of external nodes is not shown in the previous figure.

B$^+$-trees (cont'd)

- Note another difference with B-trees: for every interval of keys (k, l) in an internal node of a B$^+$-tree, this interval is connected to the node in the level below which holds keys m such that $k \leq m < l$.

Examples of insertions in a B^+-tree

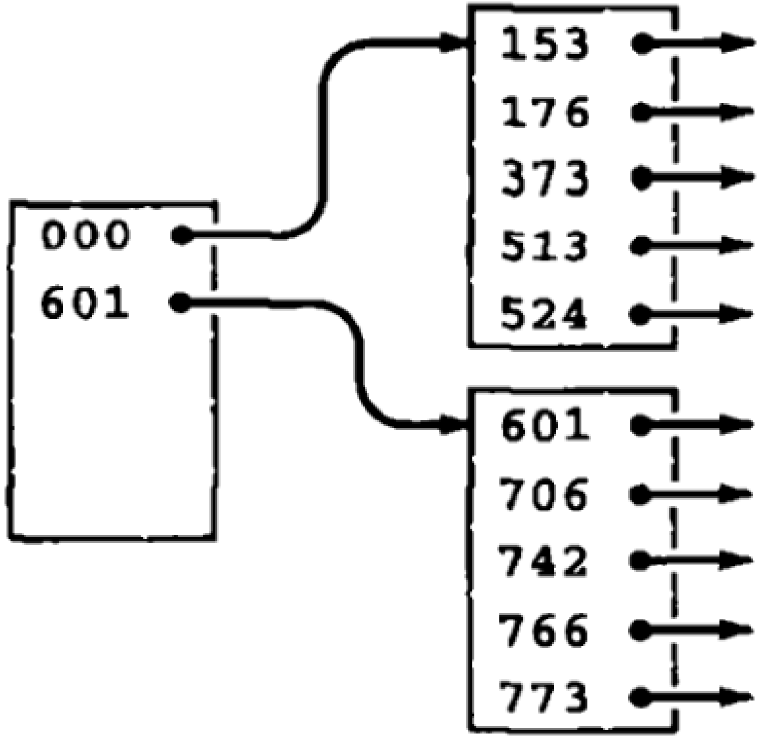


The insertion of key 773 causes a split in the root.

Notes

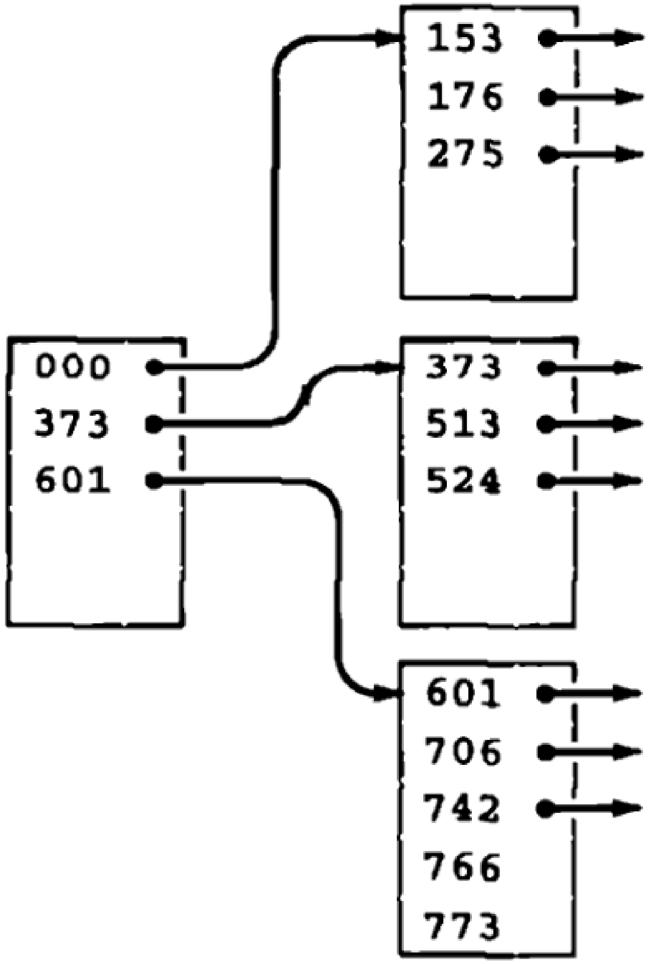
- Note the difference with B-trees: **a copy of the median key 601 goes up to form the new root.** The original key 601 remains in one of the two nodes resulting from the split (the one containing the greater keys) so that it can be in an external node and be connected to its value.
- Key 000 in the root is a **sentinel key (a special key smaller than all the others).**

Examples (cont'd)



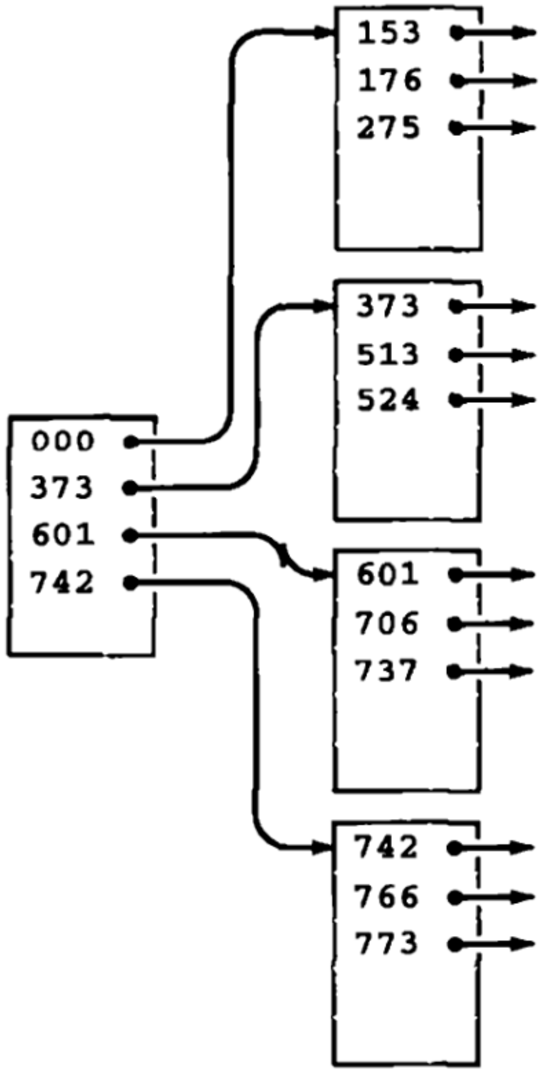
The keys 373, 524, 742 and 766 are inserted.

Examples (cont'd)



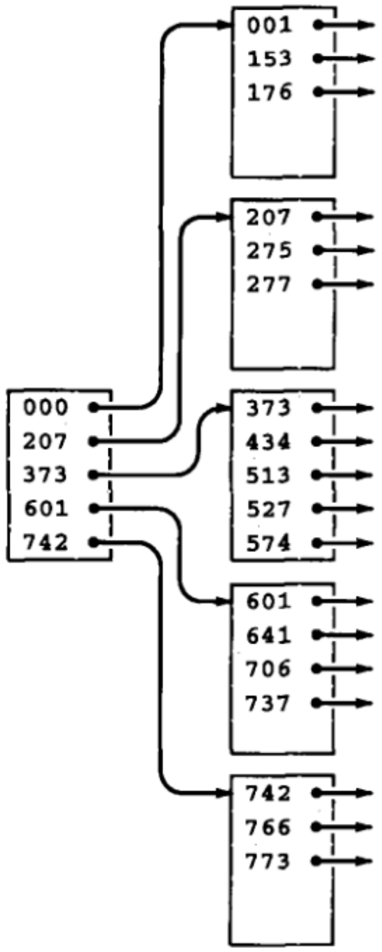
The insertion of key 275 causes a split in an external node.

Examples (cont'd)



The insertion of key 737 causes a split in an external node.

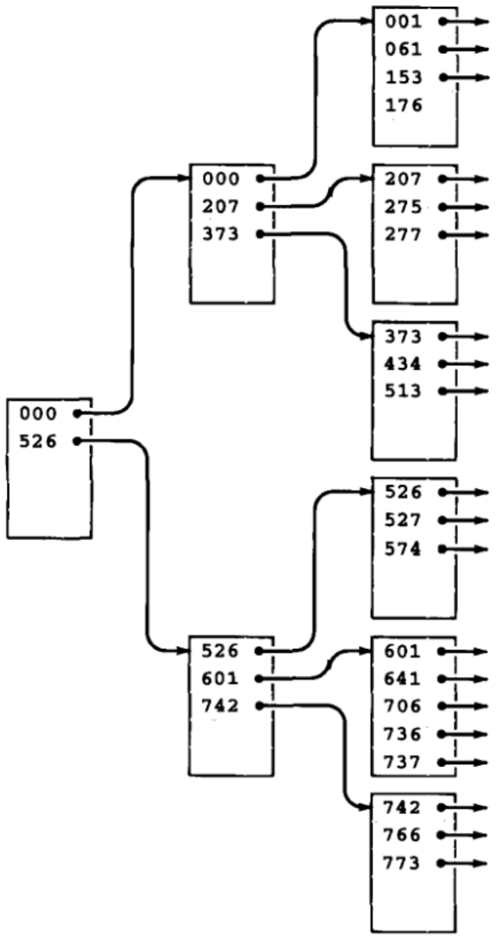
Examples (cont'd)



The 13 keys 574, 434, 641, 207, 001, 277, 061, 736, 526, 562, 017, 107, and 147 are inserted.

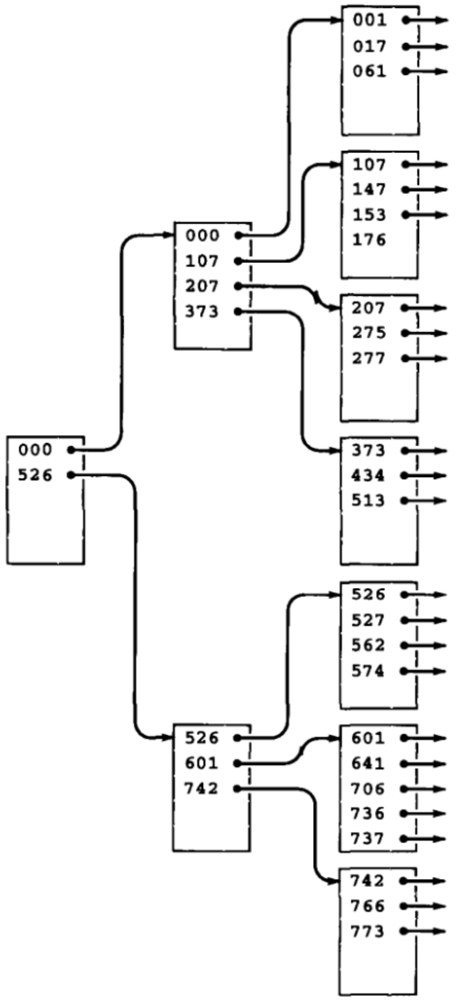
The insertion of key 277 causes a split in an external node.

Examples (cont'd)



The insertion of key 526 causes a split in an external node and in the root.

Examples (cont'd)



The insertion of key 107 causes a split in an external node.

B^+-tree type definitions

```
typedef struct STnode* link;
typedef struct
   { Key key; union { link next; Item item; } ref; }
entry;

struct STnode { entry b[M]; int m; };
```

B$^+$-tree definitions (cont'd)

- Each node in a B$^+$-tree contains an array (`b`) of size `M` (the order of the tree). It also contains a **count** (`m`) of the number of active entries in the array.
- In internal nodes, each array entry is a key and a link to a node; in external nodes, each array entry is a key and an item.
- The C `union` construct allows us to specify these two options in a single declaration.
- We are not keeping track of the ordered links between external nodes.

B$^+$-tree initialization

```
static link head;
static int H, N;

link NEW()
       {  link x = malloc(sizeof *x);
          x->m = 0;
          return x;
        }

void STinit(int maxN)
        { head = NEW(); H = 0; N = 0; }
```

B^+-tree initialization (cont'd)

- We initialize new nodes to be empty (count field `m` is set to 0).
- An **empty** B^+-tree is a link to an empty node.
- Also, we maintain variables to track the **number of items in the tree** (`N`) and the **height of the tree** (`H`).

B^+-tree search

```
Item searchR(link h, Key v, int H)
  { int j;
    if (H == 0)
      for (j = 0; j < h->m; j++)
        if (eq(v, h->b[j].key))
          return h->b[j].ref.item;
    if (H != 0)
      for (j = 0; j < h->m; j++)
        if ((j+1 == h->m) || less(v, h->b[j+1].key))
          return searchR(h->b[j].ref.next, v, H-1);
    return NULLitem;
  }


Item STsearch(Key v)
  { return searchR(head, v, H); }
```

B^+-tree search (cont'd)

- The implementation of search for B^+-trees is based on a recursive function as usual.
- For internal nodes (positive height), we scan to **find the first key larger than the search key** and do a recursive call on the subtree referenced by the previous link.
- Special consideration is given to the case when we reach the last key in the node (`j+1==h->m`). In this case, the search key is larger than the last key and search continues with a recursive call on the subtree referenced by the link `b[j]`.
- For external nodes (height 0), we scan to see whether or not **there is an item with key equal to the search key.**

B$^+$-tree insertion

```
void STinsert(Item item)
  { link t, u = insertR(head, item, H);
    if (u == NULL) return;
    t = NEW(); t->m = 2;
    t->b[0].key = head->b[0].key;
    t->b[0].ref.next = head;
    t->b[1].key =  u->b[0].key;
    t->b[1].ref.next = u;
    head = t; H++;
  }
```

B^+-tree insertion (cont'd)

- The function in the previous slide calls the recursive function `insertR` to do the insertion of the new key.
- If the returned value is `NULL`, then the insertion took place successfully and nothing more needs to be done.
- If the returned value is not `NULL`, then there has been an overflow in the previous root and a new root (variable `t`) with two children (links to `head` and `u` – `u` is returned by `insertR`) needs to be created. This is the case where the sentinel key 000 of our example is inserted in `t->b[0].key`.

B$^+$-tree insertion (cont'd)

```
link insertR(link h, Item item, int H)
  { int i, j; Key v = key(item); entry x; link u;
    x.key = v; x.ref.item = item;
    if (H == 0)
      for (j = 0; j < h->m; j++)
        if (less(v, h->b[j].key)) break;
    if (H != 0)
      for (j = 0; j < h->m; j++)
        if ((j+1 == h->m) || less(v, h->b[j+1].key))
          {
            u = insertR(h->b[j++].ref.next, v, H-1);
            if (u == NULL) return NULL;
            x.key = u->b[0].key; x.ref.next = u;
            break;
          }
    for (i = ++(h->m); (i > j) && (i != M); i--)
      h->b[i] = h->b[i-1];
    h->b[j] = x;
    if (h->m < M) return NULL; else return split(h);
  }
```

B^+-tree insertion (cont'd)

- We insert new items by moving larger items to the right by one position as in insertion sort (this is done by the last `for` loop in the function).
- If the insertion overfills the node, we call `split` to divide the node into two halves, and return the link to the new node.
- One level up in the recursion, this extra link causes a similar insertion in the parent internal node, which could also split, possibly propagating the insertion all the way up to the root.

B^+-tree insertion (cont'd)

```
link split(link h)
  { int j; link t = NEW();
    for (j = 0; j < M/2; j++)
      t->b[j] = h->b[M/2+j];
    h->m = M/2; t->m = M/2;
    return t;
  }
```

B^+-tree insertion (cont'd)

- To split a node in a B^+-tree , we create a new node, move the larger half of the keys to the new node, and then adjust counts in both nodes.
- **The code on the previous slide assumes that `M` is even.** That is, the maximum number of keys in a node is `M-1`, and when a node gets `M` keys, we split it into two nodes with `M/2` keys each.

Exercise

- Write a function `STdelete` that deletes a key from a B$^+$-tree.

Readings

- M. T. Goodrich, R. Tamassia and Michael H. Goldwasser. *Data Structures and Algorithms in Java. 6*th edition. John Wiley and Sons, 2014.
  - Section 15.3
- Sartaj Sahni. Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++. Εκδόσεις Τζιόλα.

Readings (cont’d)

- You should 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 nodes on the bottom layer):
    - R. Sedgewick. Αλγόριθμοι σε C.
        - Κεφ. 16.3

The presentation of B^+-trees in the present slides comes from that chapter.