B-Trees

External Searching

- 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 **hard disk**.
- **Examples**: the database of a bank, a database of satellite images, a database of videos etc.

External Searching (cont'd)

- 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.

B-Trees

- 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 $b = m$.
- We choose m such that the m children references and the $m - 1$ keys stored at a node can all fit into a **single block**.

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:
$$\frac{1}{\log b}\log(n+1) \le h \le \frac{1}{\log a}\log\frac{n+1}{2}+1$$
- 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 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 inequalities.
- The upper bound we want to prove is also easy to see as follows:

$$h \log a - \log a + 1 \leq \log(n + 1)$$

$$h \log a \leq \log(n + 1) + \log a - 1$$

$$h \leq \frac{1}{\log a} \log \frac{n + 1}{2} + 1$$

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 corresponding result we proved for (a, b) trees.
- Because the number of external nodes is one plus the number of entries of the tree, from this result we have $n \geq 2d^{h-1} - 1$.
- To prove (2), rewrite the inequalities and then take logarithms with bases m and d for the respective terms.

Fact

- From the previous proposition, we have that the height of a B-tree is $\boldsymbol{O(\log_d n)}$ as we would like it for a balanced search tree.

Declarations

- To implement B-trees in C, we can start with the following declarations:

```
#define MAX 4          /* maximum number of keys in node */
#define MIN 2          /* minimum number of keys in node */

typedef int Key;

typedef struct {
  Key key;
  int value;         /* values can be of arbitrary type */
} Treeentry;

typedef struct treenode Treenode;
struct treenode {
  int count;           /* number of keys in node */
  Treeentry entry[MAX+1];
  Treenode *branch[MAX+1];
};
```

Declarations (cont’d)

- The constant MAX=$m - 1$. The constant MIN=$\left\lceil \frac{m}{2} \right\rceil - 1$.
- The entries at each node are kept in an array `entry` and the pointers in an array `branch`.
- The variable `count` gives us the number of keys at a node.

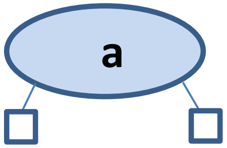
Insertion into a B-tree

- 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** 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.

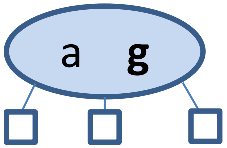
Example

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

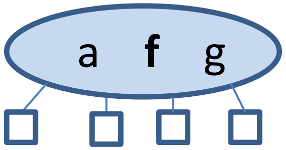
Insert a



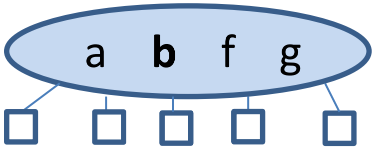
Insert g



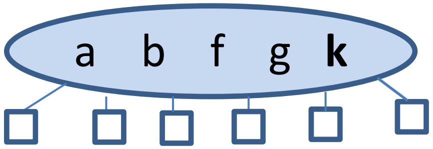
Insert f



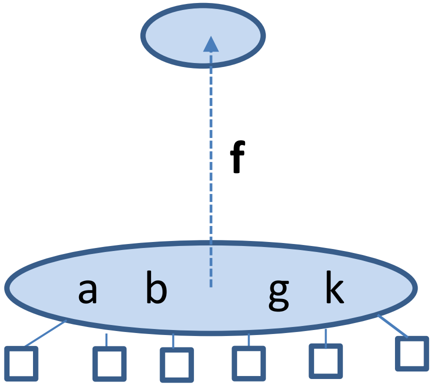
Insert b



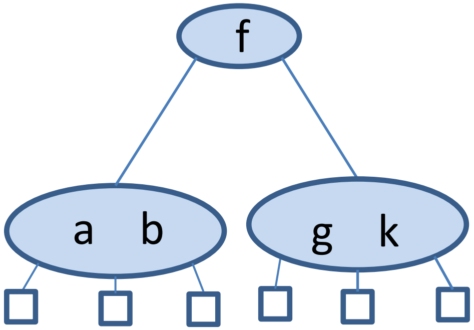
Insert k - Overflow



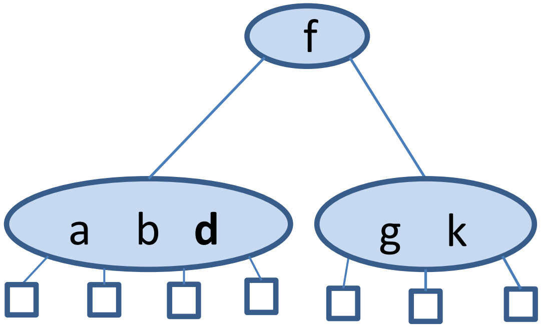
Creation of a New Root Node



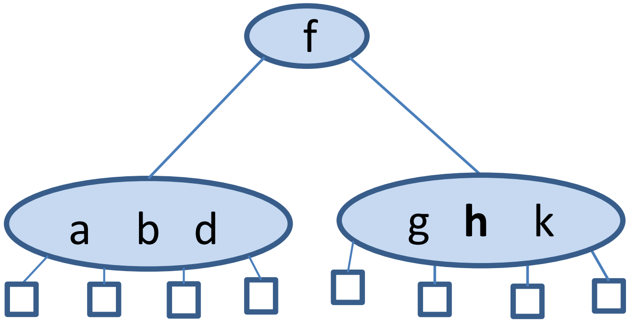
Split



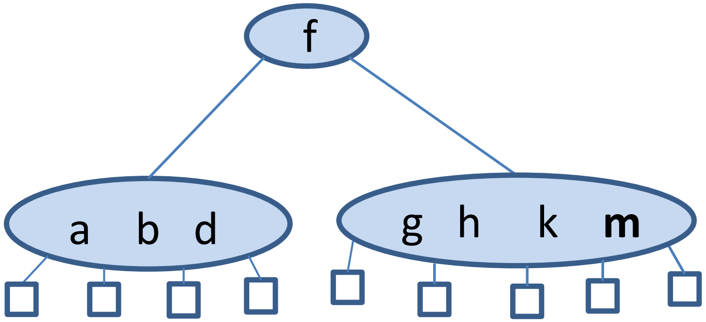
Insert d



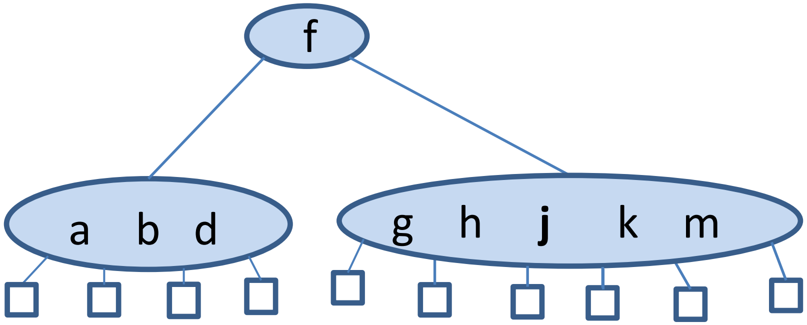
Insert h



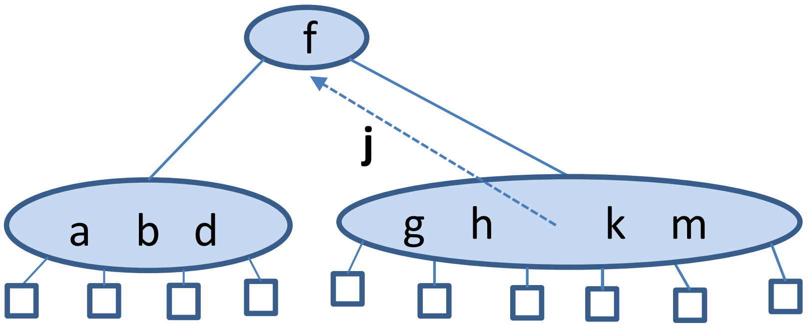
Insert m



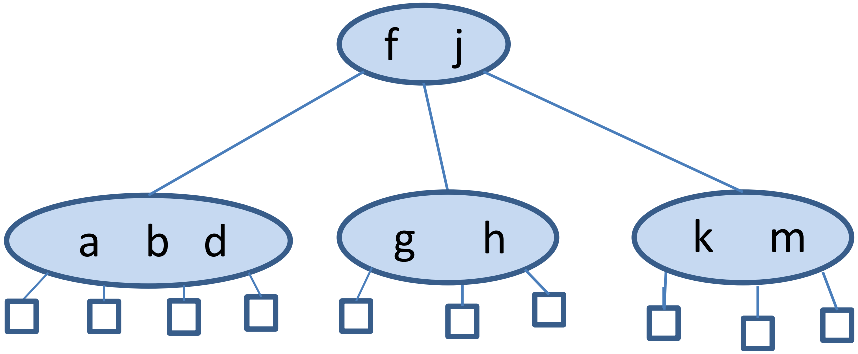
Insert j - Overflow



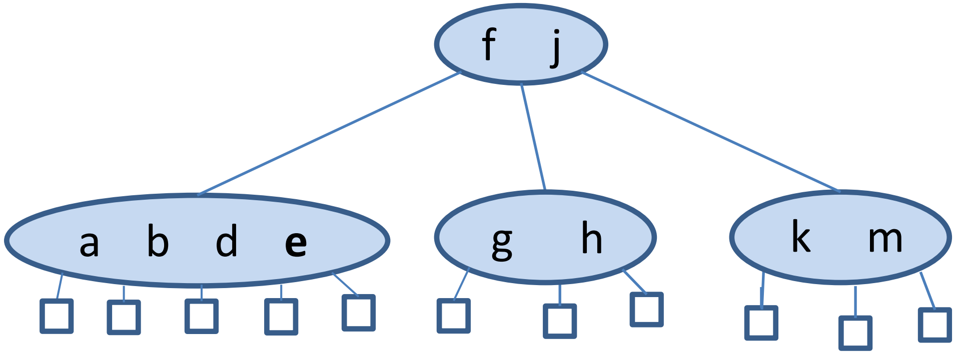
Sent j to the Parent Node



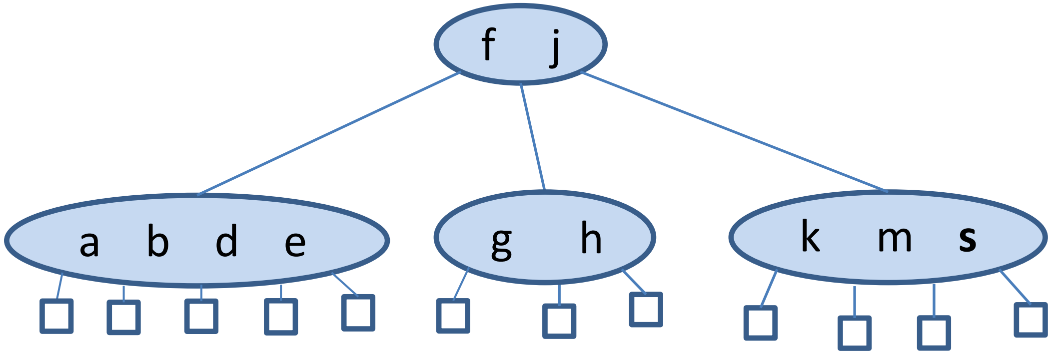
Split



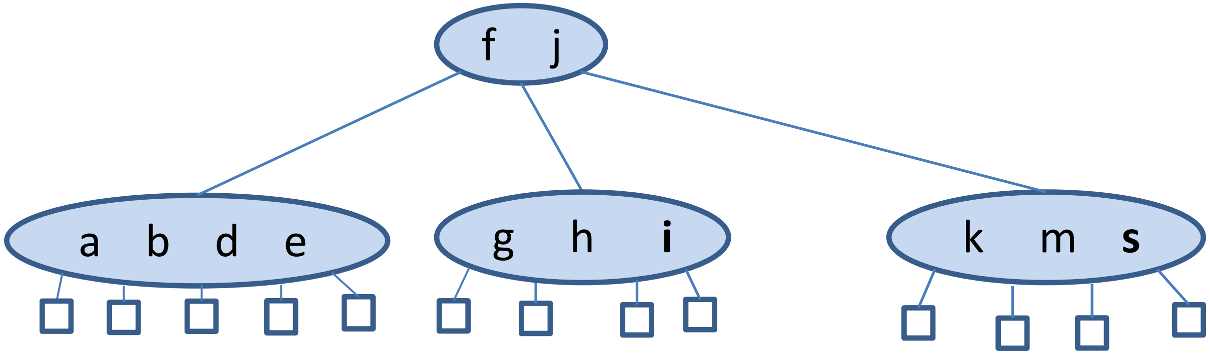
Insert e



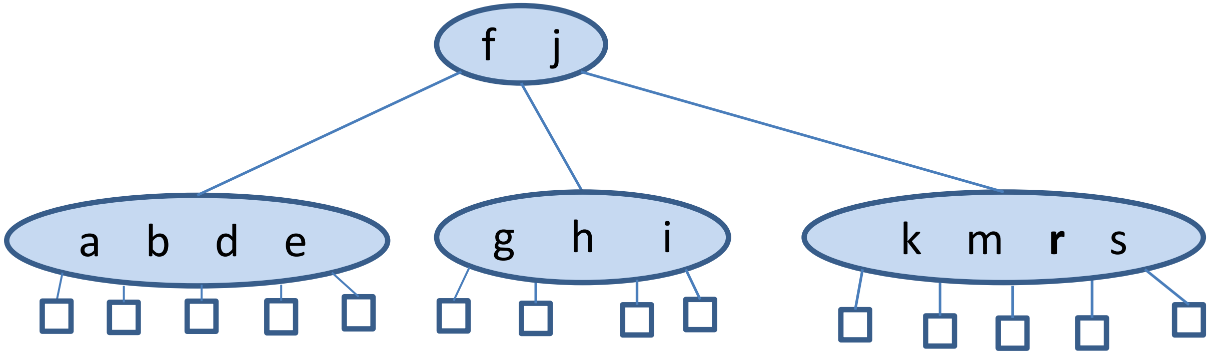
Insert s



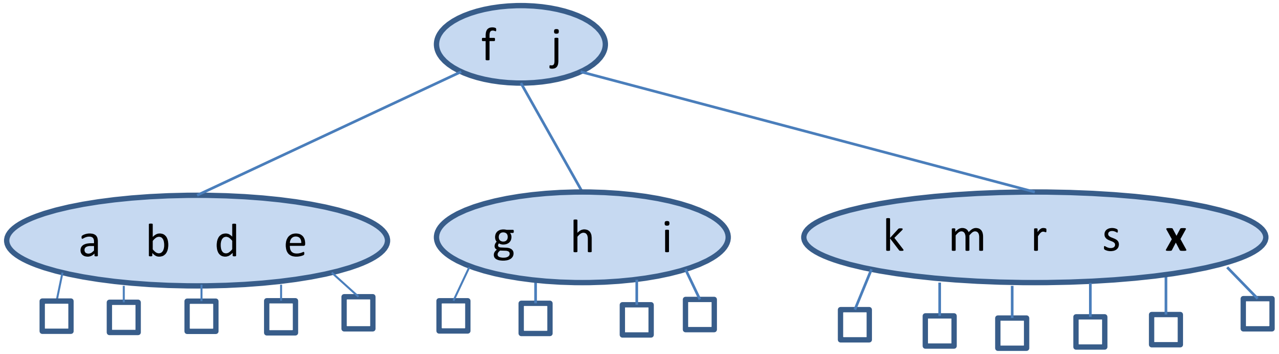
Insert i



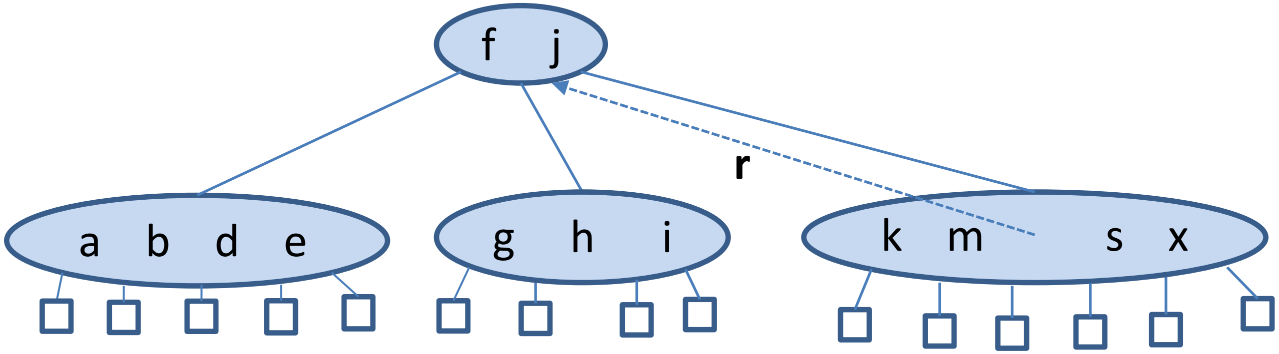
Insert r



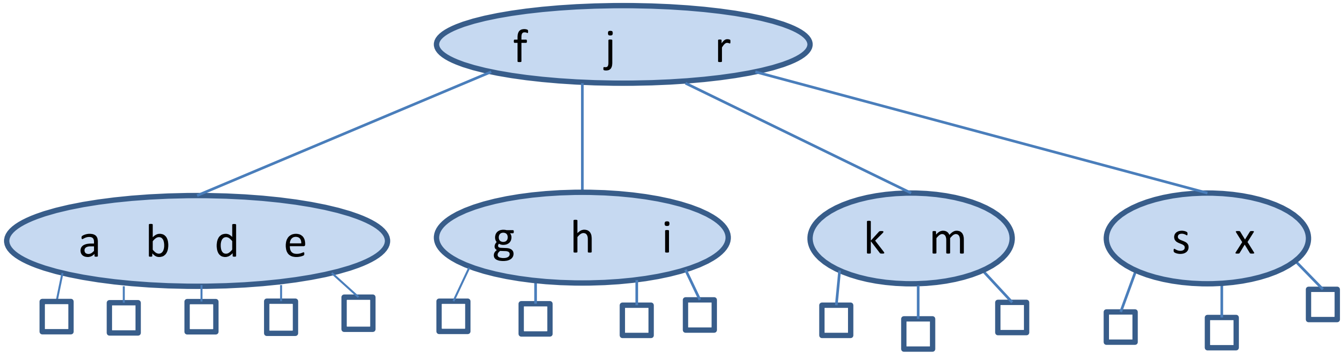
Insert x - Overflow



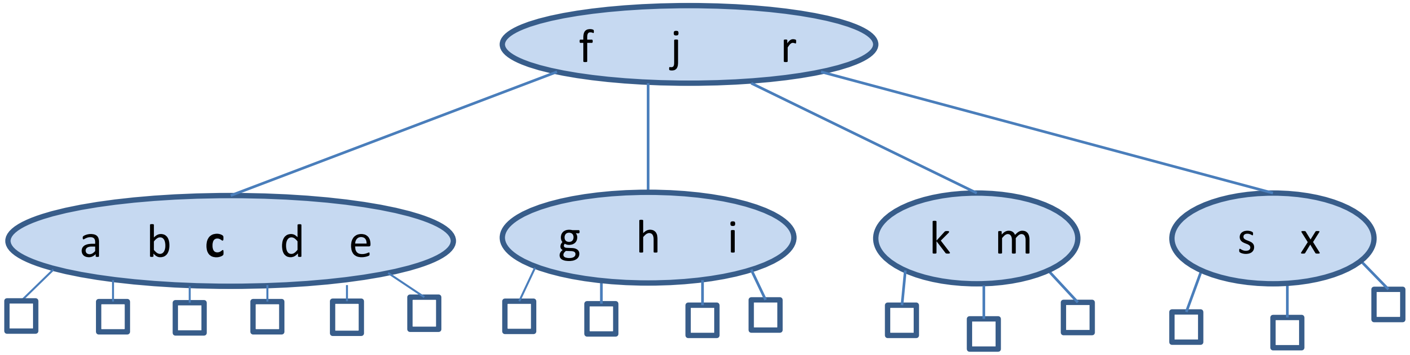
r is Sent to the Parent Node



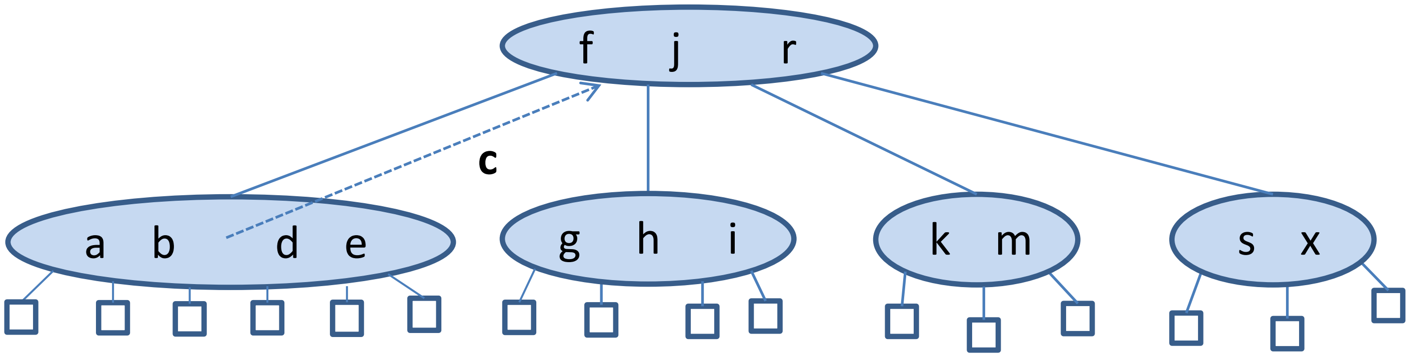
Split



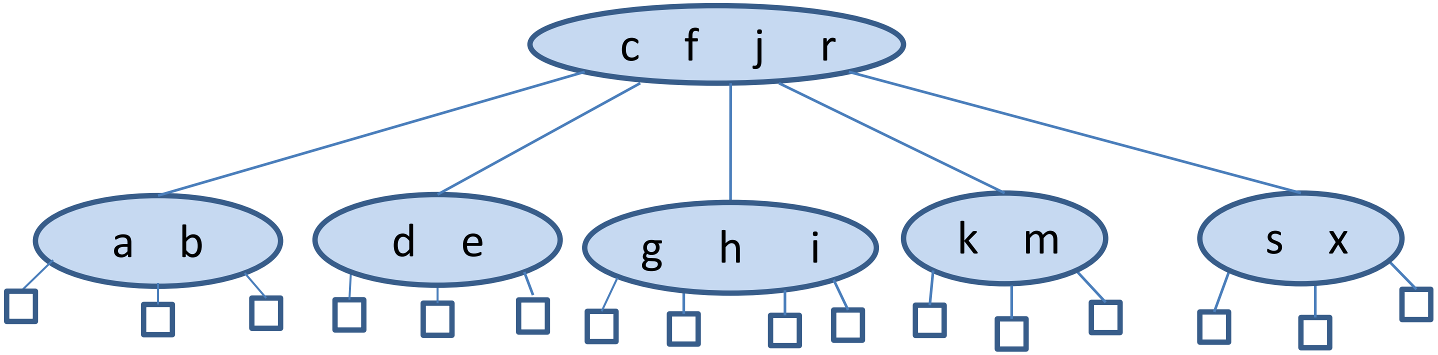
Insert c - Overflow



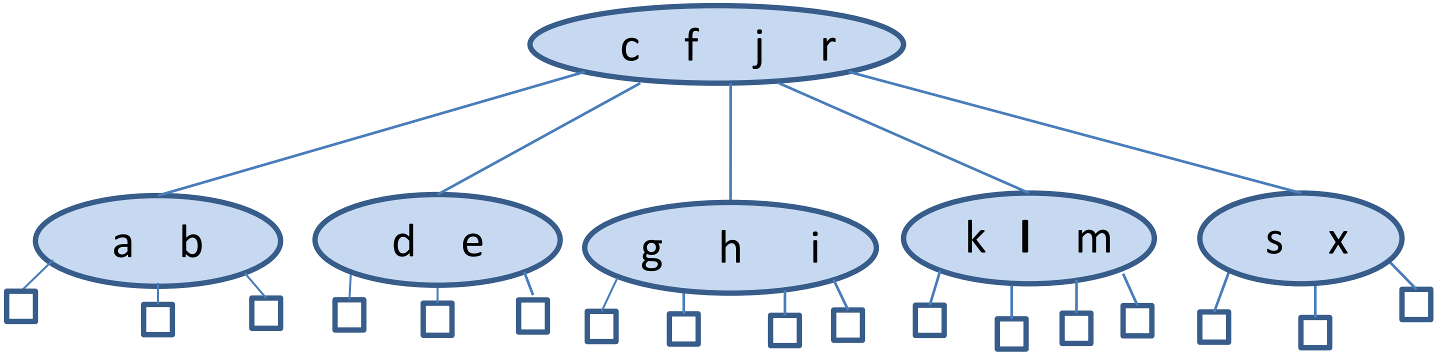
c is Sent to the Parent



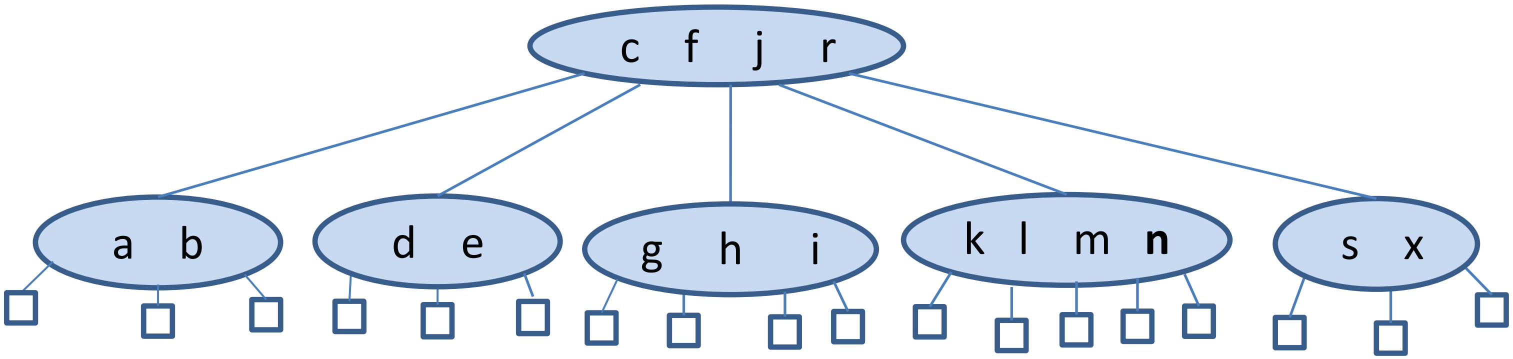
Split



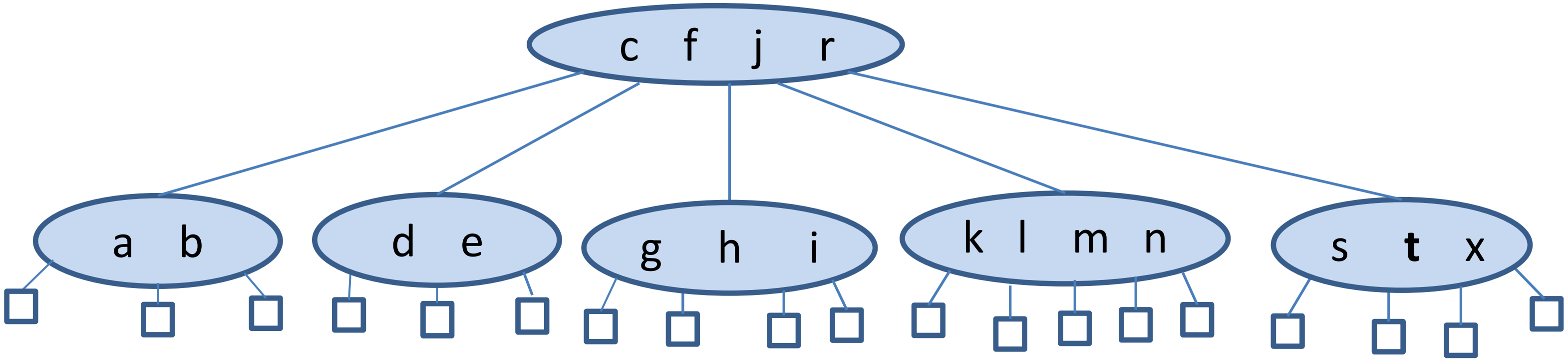
Insert l



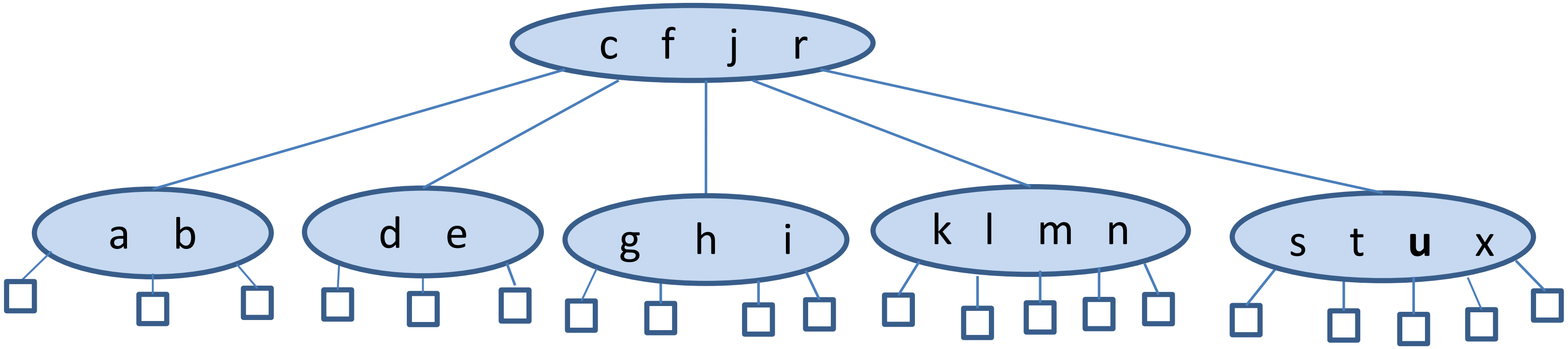
Insert n



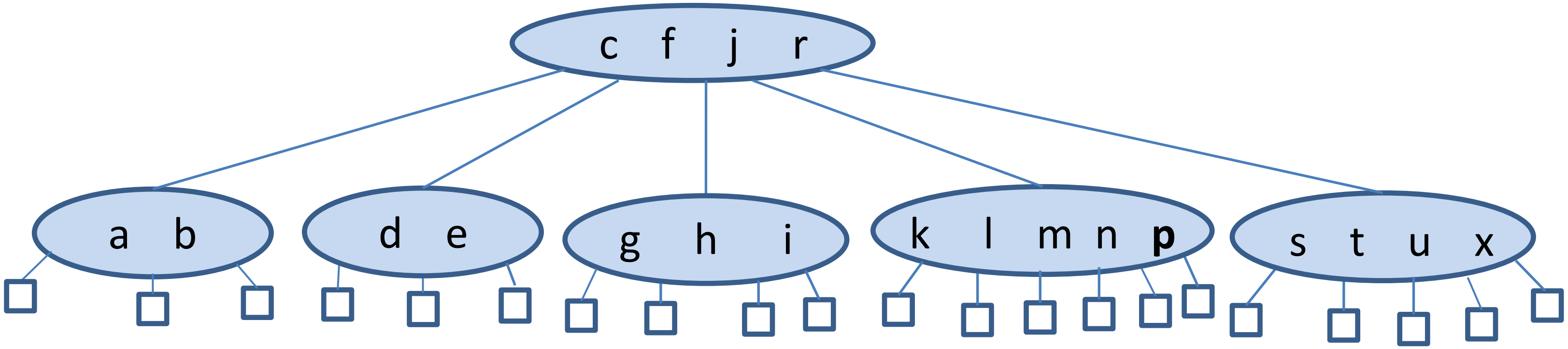
Insert t



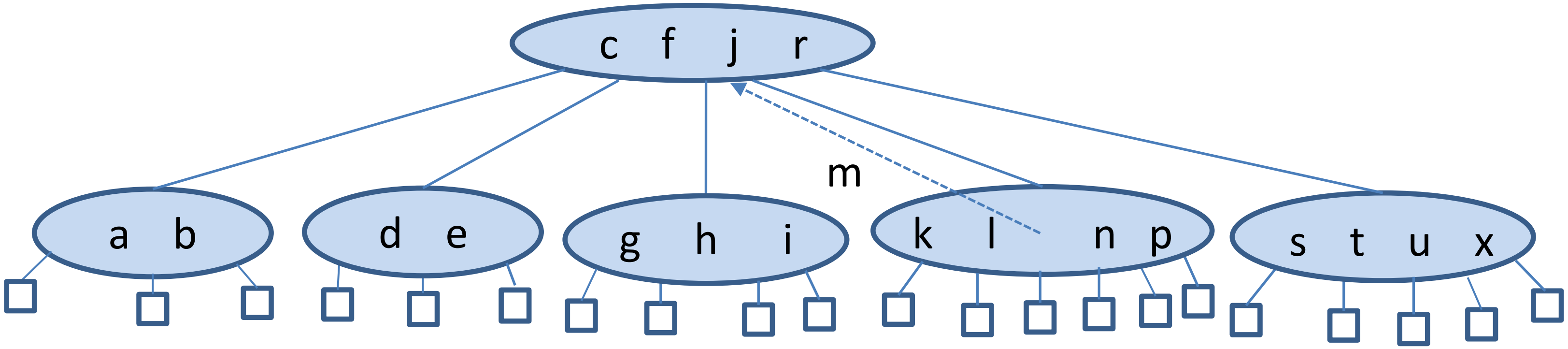
Insert u



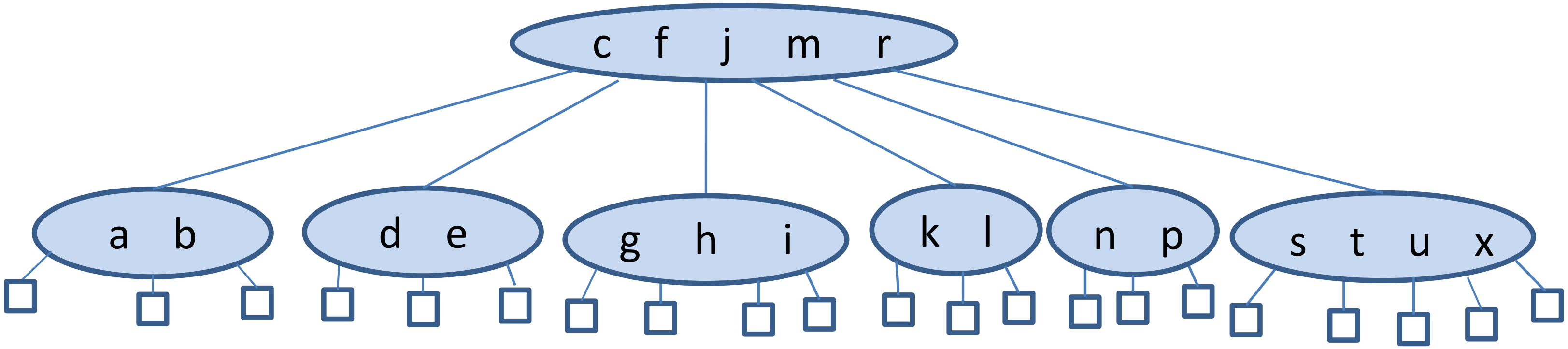
Insert p - Overflow



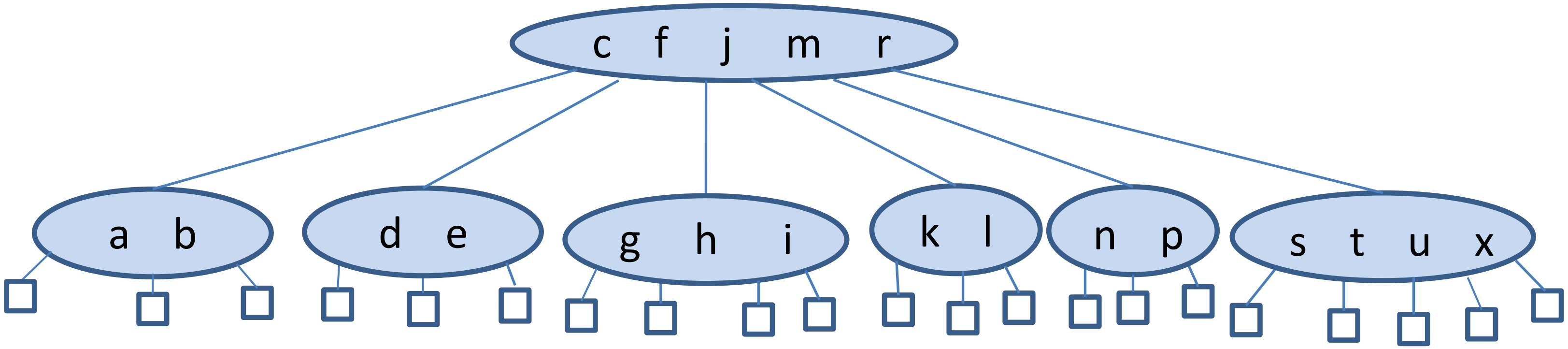
m is Sent to the Parent Node



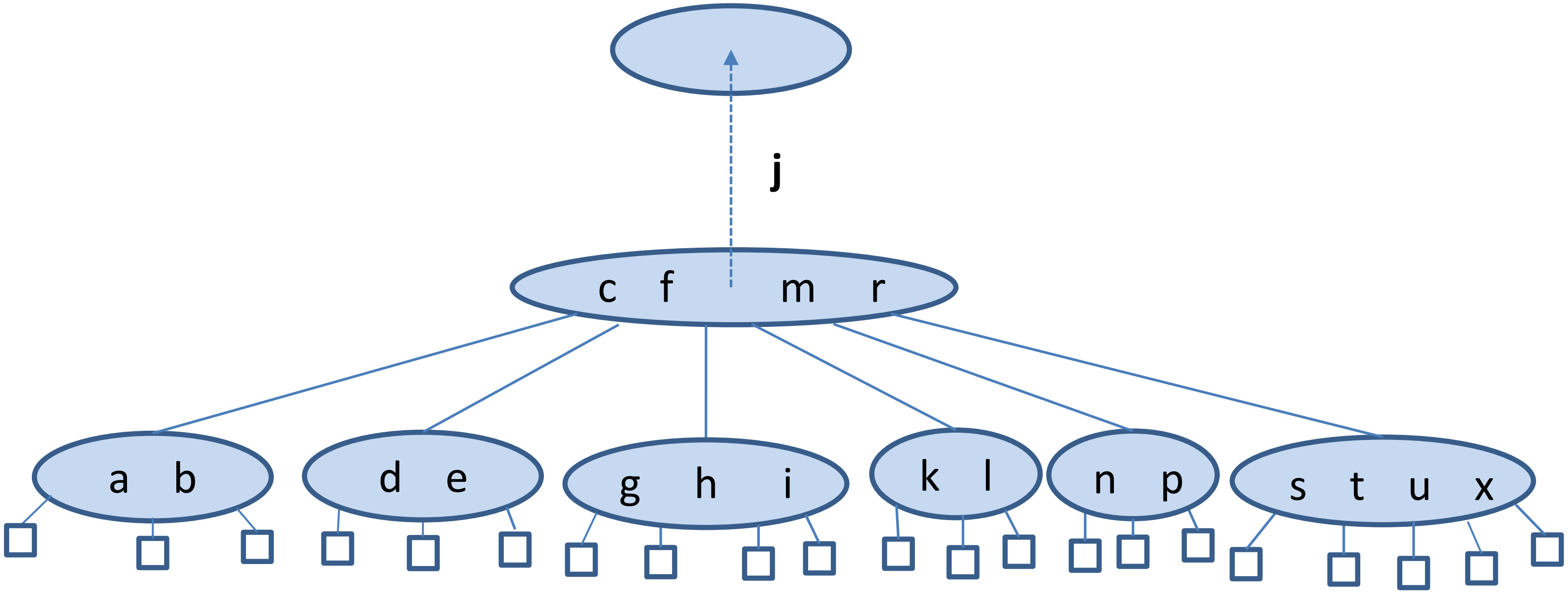
Split



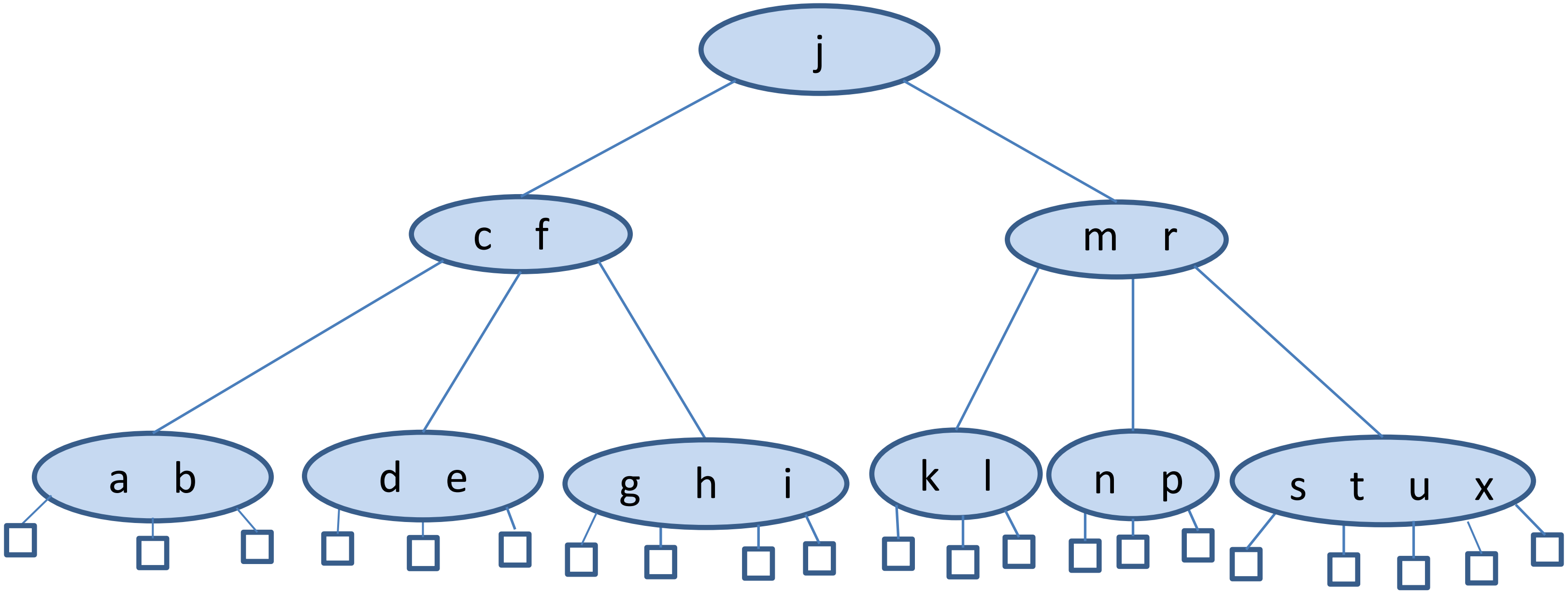
Overflow at the Root



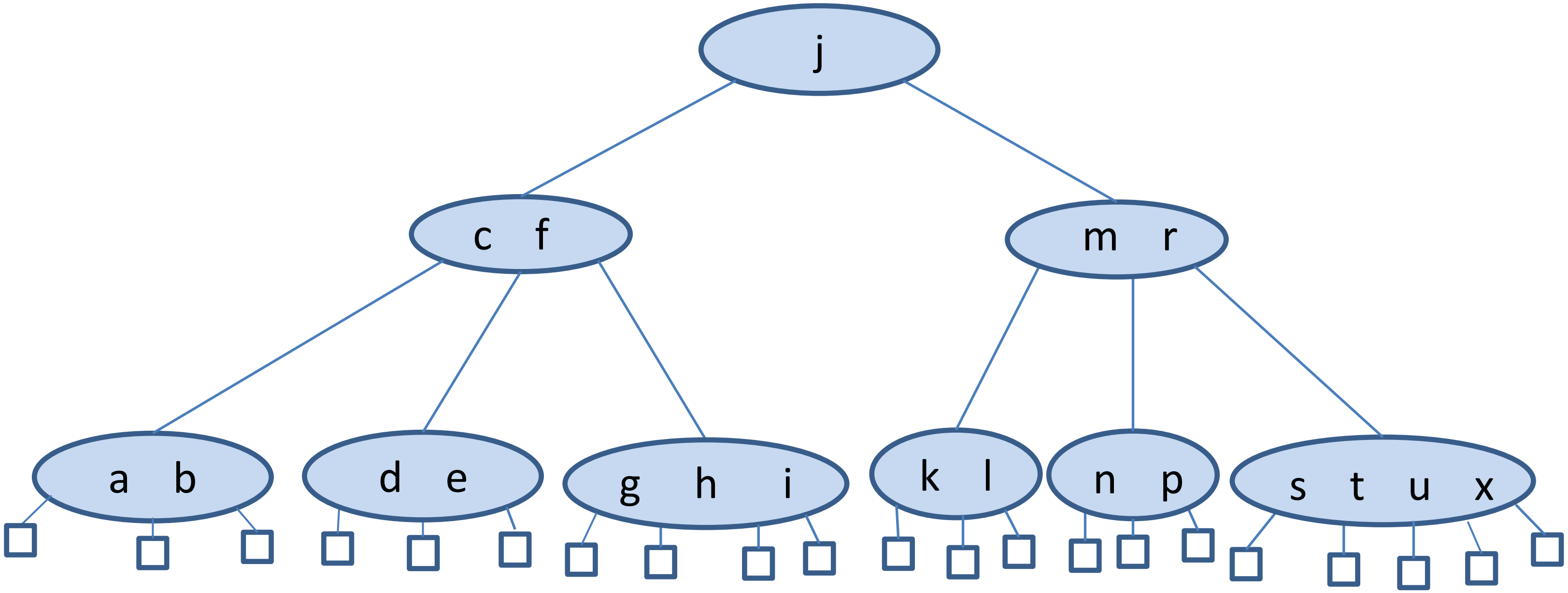
j is Sent up to a New Root



Split



Final Tree



Insertion into a B-tree

- We will write a recursive function for inserting a key into a B-tree.
- The recursion is started by the function `InsertTree` which calls the recursive function `PushDown`. If the outermost call to function `PushDown` returns `TRUE`, then there is a key to be placed in a new root and the height of the tree increases.

Insertion into a B-tree (cont'd)

```
/* InsertTree: Inserts entry into the B-tree.
   Pre: The B-tree to which root points has been created, and no entry in the B-tree
   has key equal to newentry key.
   Post: newentry has been inserted into the B-tree, the root is returned.
   Uses: PushDown */

Treenode *InsertTree(Treeentry newentry, Treenode *root)
{
  Treeentry medentry;   /* node to be reinserted as new root */
  Treenode *medright;   /* subtree on right of medentry     */
  Treenode *newroot;   /* used when the height of the tree increases */

  if (PushDown(newentry, root, &medentry, &medright)){
    /* Tree grows in height. Make a new root */
    newroot=(Treenode *)malloc(sizeof(Treenode));
    newroot->count=1;
    newroot->entry[1]=medentry;
    newroot->branch[0]=root;
    newroot->branch[1]=medright;
    return newroot;
  }
  return root;
}
```

Recursive Insertion into a Subtree

- The recursive function `PushDown` recursively moves down the tree looking for a position for `newentry`.
- We continue searching until we hit an empty subtree. Then we return `TRUE` and send the key (now called `medentry`) back up for insertion.
- The parameter `current` points to the root of the subtree being searched.
- `medentry` is the median key sent up to a parent.
- When a recursive call returns `TRUE`, we attempt to insert the key `medentry` in the current node. If there is room, we are finished.
- Otherwise, the current node `*current` splits into `*current` and `*medright` and a (possibly different) median key `medentry` is sent up the tree.

Recursive Insertion into a Subtree (cont'd)

```
/* PushDown: recursively move down tree searching for newentry.
   Pre: newentry belongs in the subtree to which current points.
   Post: newentry has been inserted into the subtree to which current points; if TRUE
   is returned, then the height of the subtree has grown, and medentry needs
   to be reinserted higher in the tree, with subtree medright on its right.
   Uses: PushDown recursively, SearchNode, Split, PushIn. */

Boolean PushDown(Treeentry newentry, Treenode *current, Treeentry *medentry, Treenode **medright)
{
  int pos;              /*branch on which to continues the search */

  if (current==NULL){  /* cannot insert into empty tree; terminates */
    *medentry=newentry;
    *medright=NULL;
    return TRUE;
  } else {             /* Search the current node */
    if (SearchNode(newentry.key, current, &pos))
      printf("Inserting duplicate key into B-tree");
    if (PushDown(newentry, current->branch[pos], medentry, medright))
      if (current->count < MAX){    /*Reinsert median key.  */
        PushIn(*medentry, *medright, current, pos);
        return FALSE;
      } else {
        Split(*medentry, *medright, current, pos, medentry, medright);
        return TRUE;
      }
    return FALSE;
  }
}
```

Searching a Node

- The Boolean function `SearchNode` determines if the target key `target` is in the current node and, if not, finds which of the `count+1` branches will contain the target key.
- The position of the target or the branch to continue searching is returned in variable `pos`.
- The branch 0 is considered separately. For the rest of the entries, the function `SearchNode` uses sequential search starting at the end of the array `entry`.
- If nodes of the tree contain many keys, we might want to use binary search.

Searching a Node (cont’d)

```
/* SearchNode: searches keys in node for target.
   Pre: target is a key and current points to a node of a B-tree.
   Post: Searches keys in node for target; returns location pos of
   target, or branch on which to continue search.*/

Boolean SearchNode(Key target, Treenode *current, int *pos)
{
  if (target < current->entry[1].key){  /* Take the leftmost branch */
    *pos=0;
    return FALSE;
  } else {           /* Start a sequential search through the keys */
    for (*pos=current->count; target < current->entry[*pos].key &&
*pos > 1; (*pos)--)
      ;
    return (target == current->entry[*pos].key);
  }
}
```

Insertion of a Key into a Node

- The function `PushIn` inserts the key `medentry` and its right-hand pointer `medright` into node `*current` at position `pos` provided that there is space.

Insertion of a Key into a Node (cont’d)

```
/* PushIn: inserts a key into a node.
   Pre: medentry belongs at index pos in node *current, which has room
   for it.
   Post: Inserts key medentry and pointer medright into *current at
   index pos. */

void PushIn(Treeentry medentry, Treenode *medright, Treenode *current, int pos)
{
  int i;        /* index to move keys to make room for medentry */

  for (i=current->count; i>pos; i--){
    /* Shift all keys and branches to the right */
    current->entry[i+1]=current->entry[i];
    current->branch[i+1]=current->branch[i];
  }
  current->entry[pos+1]=medentry;
  current->branch[pos+1]=medright;
  current->count++;
}
```

Splitting a Full Node

- The function `Split` inserts the key `medentry` with subtree pointer `medright` into the full node `*current`, splits the right half off as new node `*newright`, and sends the median key `newmedian` upward for reinsertion later.

Splitting a Full Node

```
/* Split: splits a full node.
   Pre: medentry belongs at index pos of node *current which is full.
   Post: Splits node *current with entry medentry and pointer medright at index pos
         into nodes *current and *newright with median entry newmedian.
   Uses: PushIn */

void Split(Treeentry medentry, Treenode *medright, Treenode *current, int pos, Treeentry *newmedian, Treenode **newright)
{
  int i;             /* used for copying from *current to new node */
  int median;        /* median position in the combined, overfull node */

  if (pos<=MIN)     /* Find splitting point. Determine if new key goes to left or right half */
    median=MIN;
  else
    median=MIN+1;

  /* Get a new node and put it on the right */
  *newright=(Treenode *)malloc(sizeof(Treenode));
  for (i=median+1; i<=MAX; i++){    /* Move half the keys to the right node */
    (*newright)->entry[i-median]=current->entry[i];
    (*newright)->branch[i-median]=current->branch[i];
  }
  (*newright)->count=MAX-median;
  current->count=median;

  if (pos <= MIN)          /* Push in the new key */
    PushIn(medentry, medright, current, pos);
  else
    PushIn(medentry, medright, *newright, pos-median);
  *newmedian=current->entry[current->count];
  (*newright)->branch[0]=current->branch[current->count];
  current->count--;
}
```

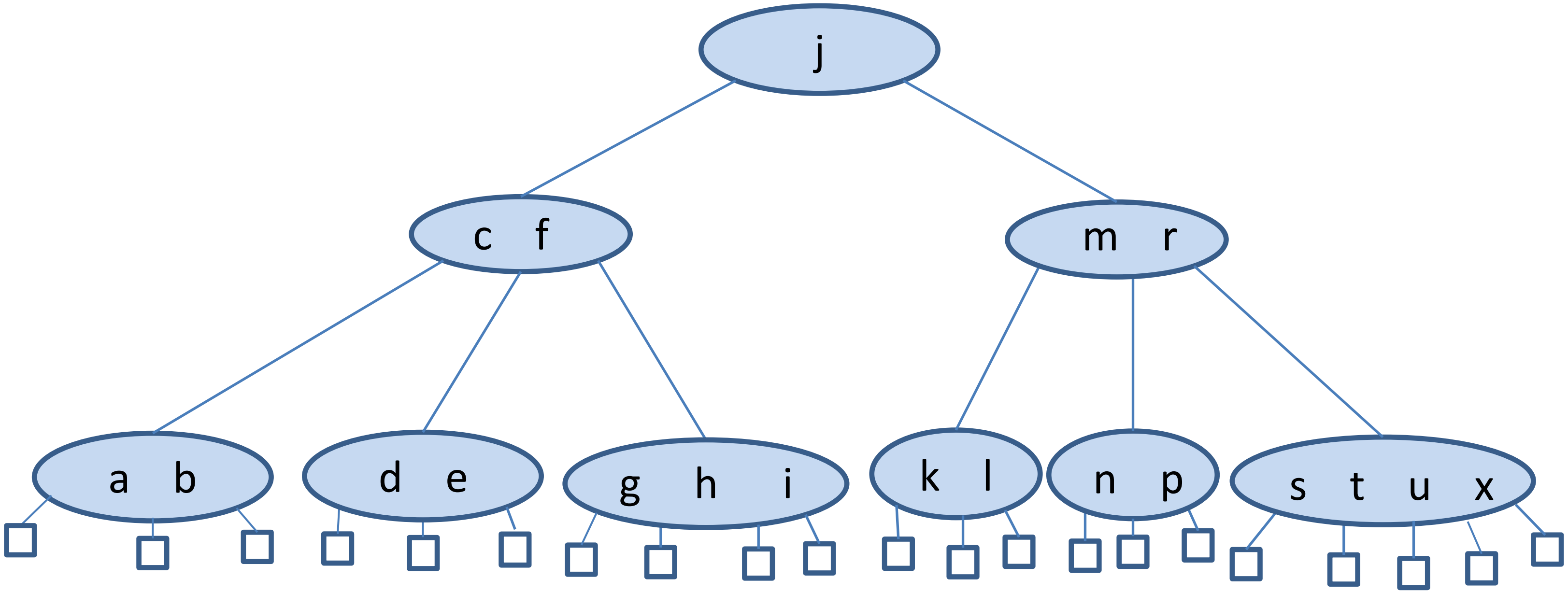
Deletion from a B-tree

- 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. (Proof?)
- 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.

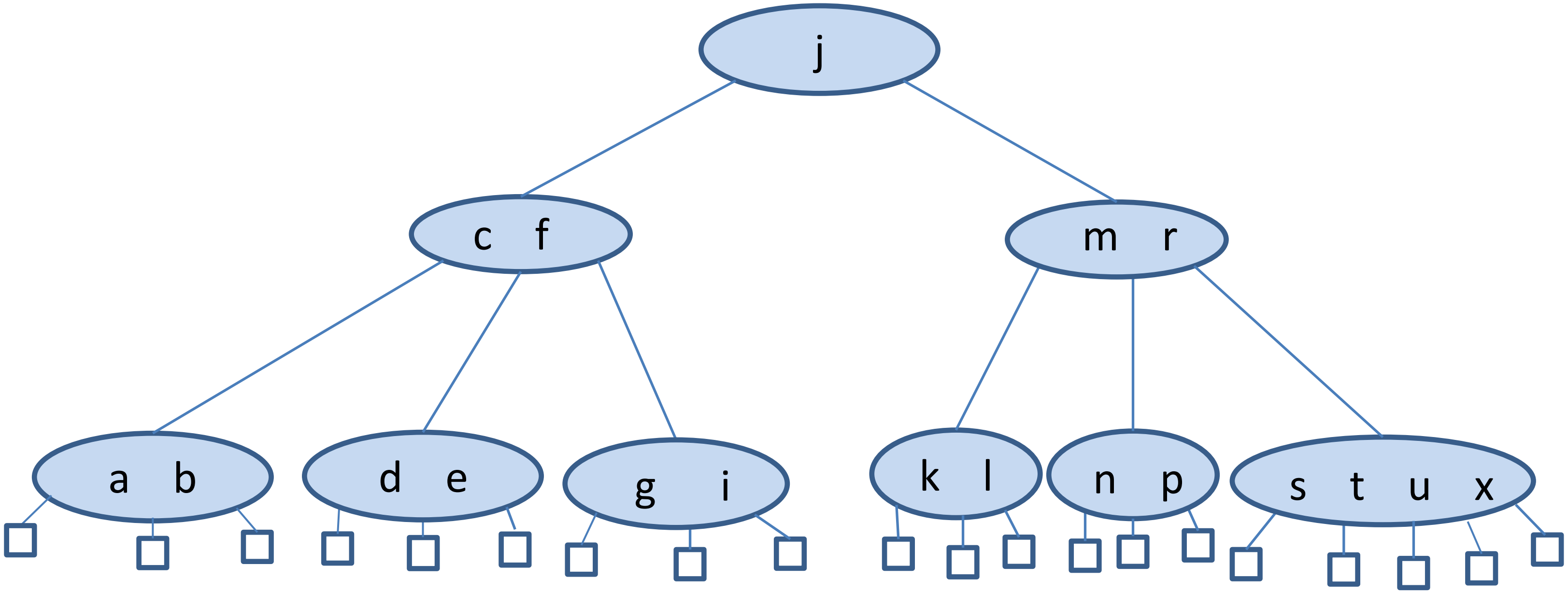
Deletion from a B-tree (cont’d)

- 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 step leaves the parent with too few entries, the process **propagates upward.**

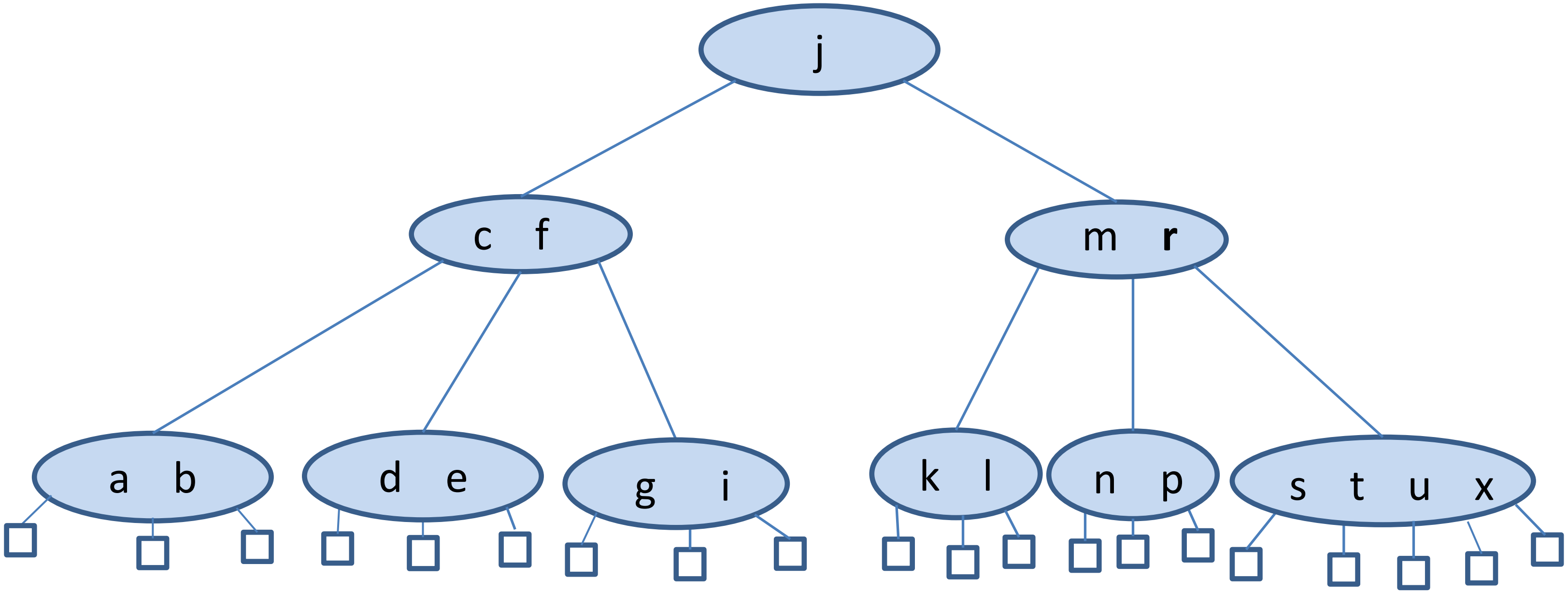
Example



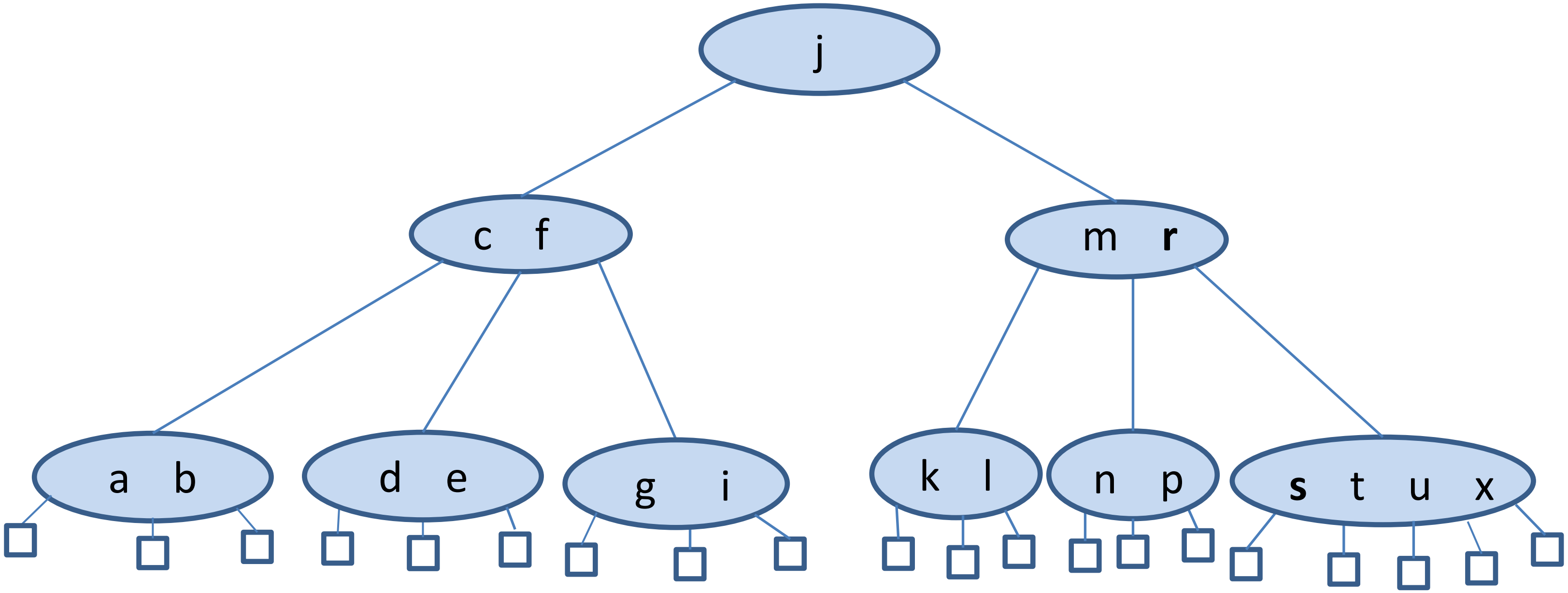
Delete h



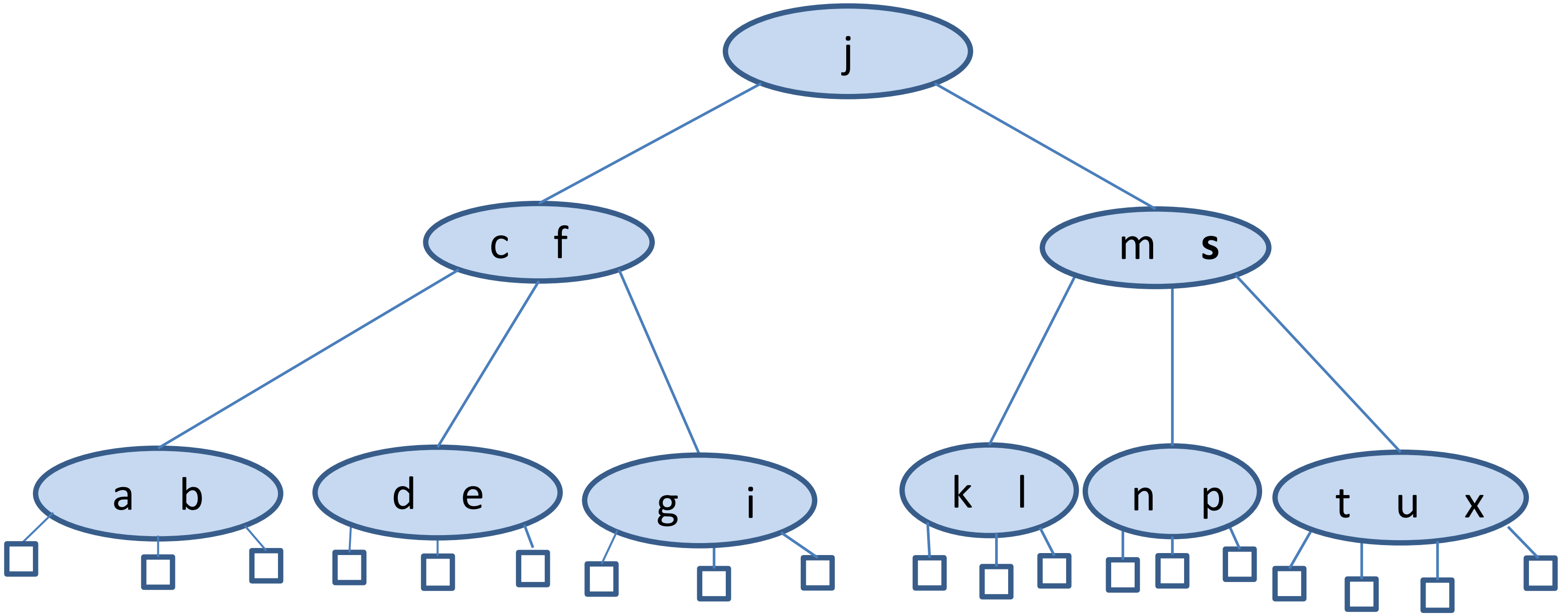
Delete r



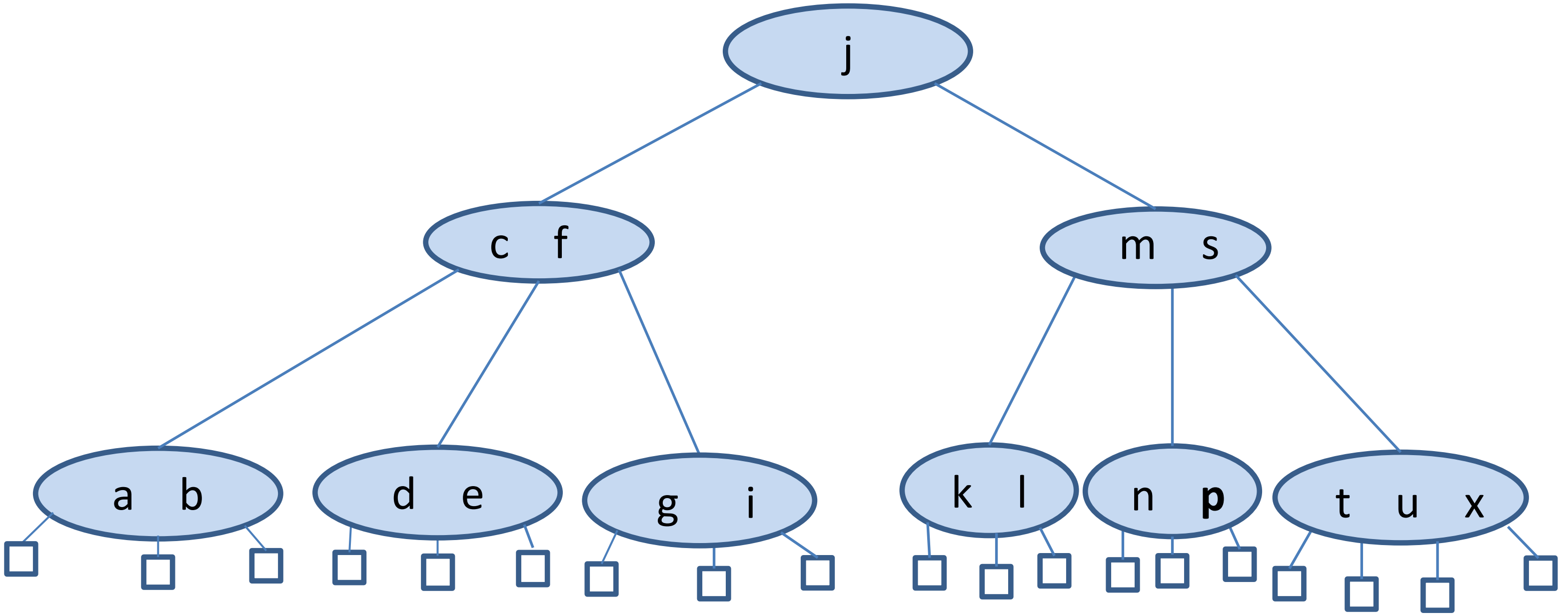
Find the Successor of r



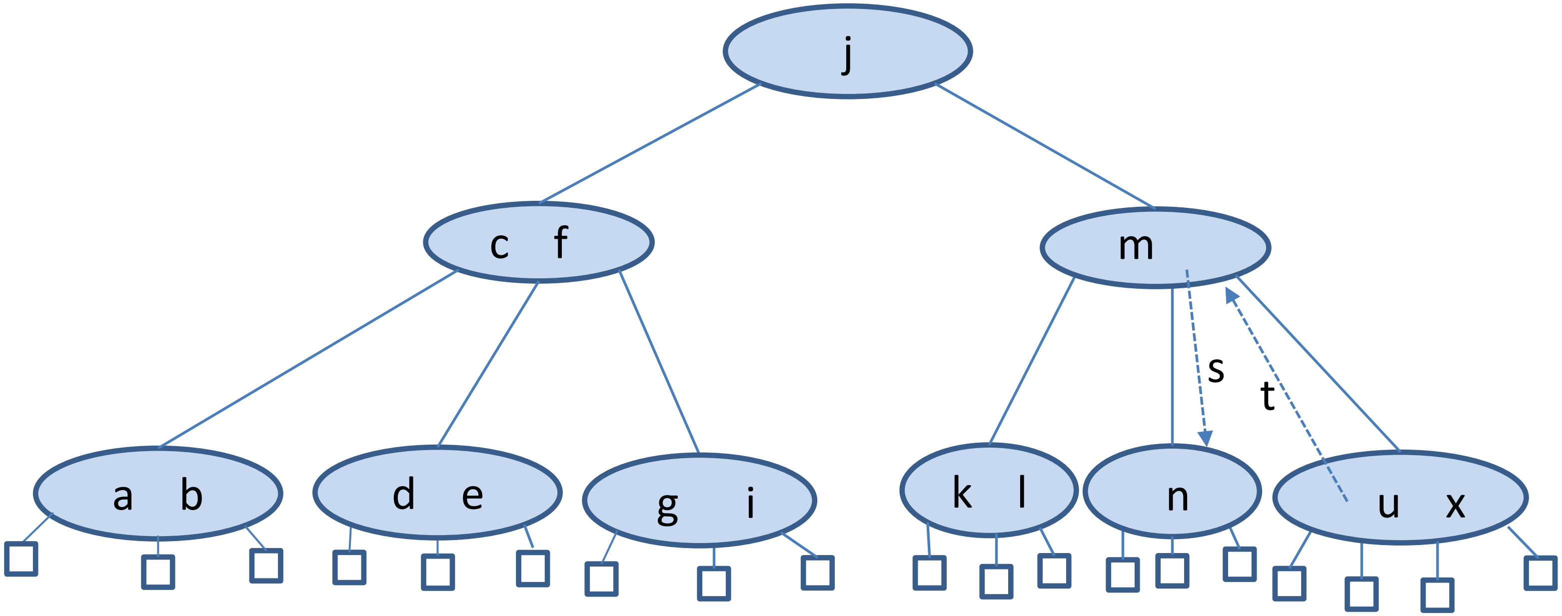
Promote the Successor of r – Delete the Successor from its Place



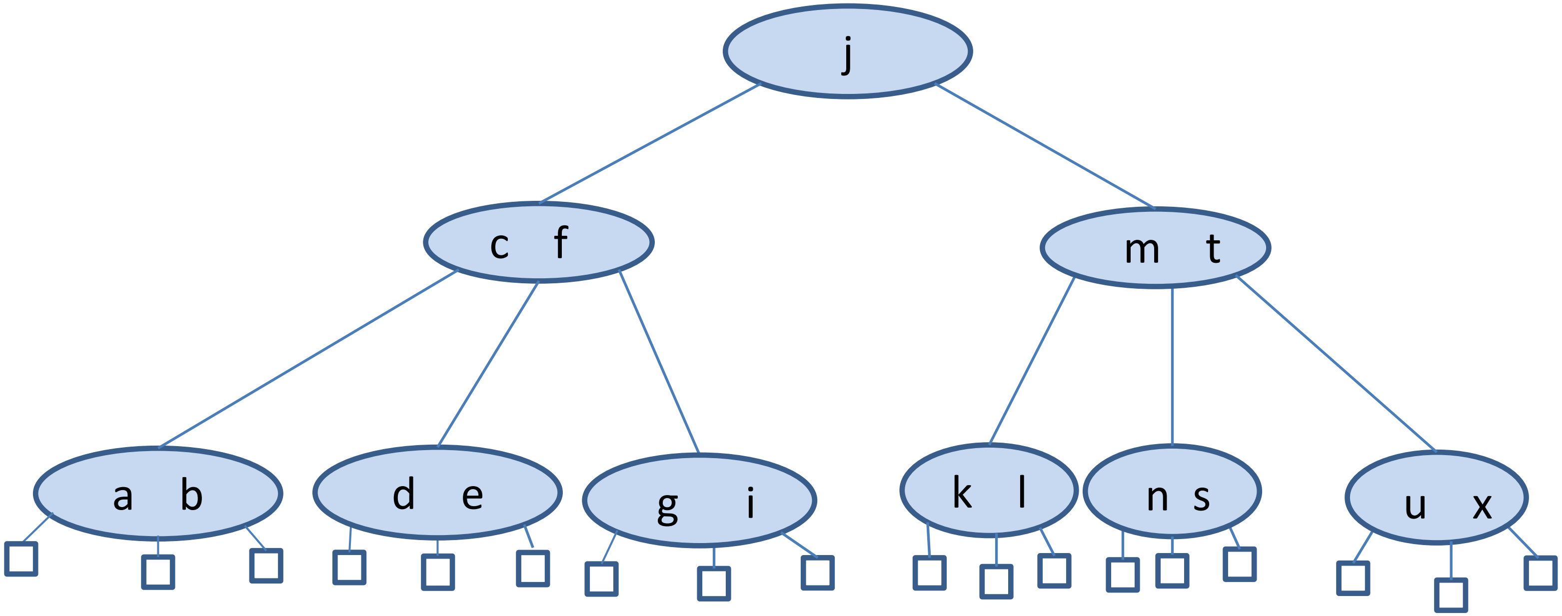
Delete p



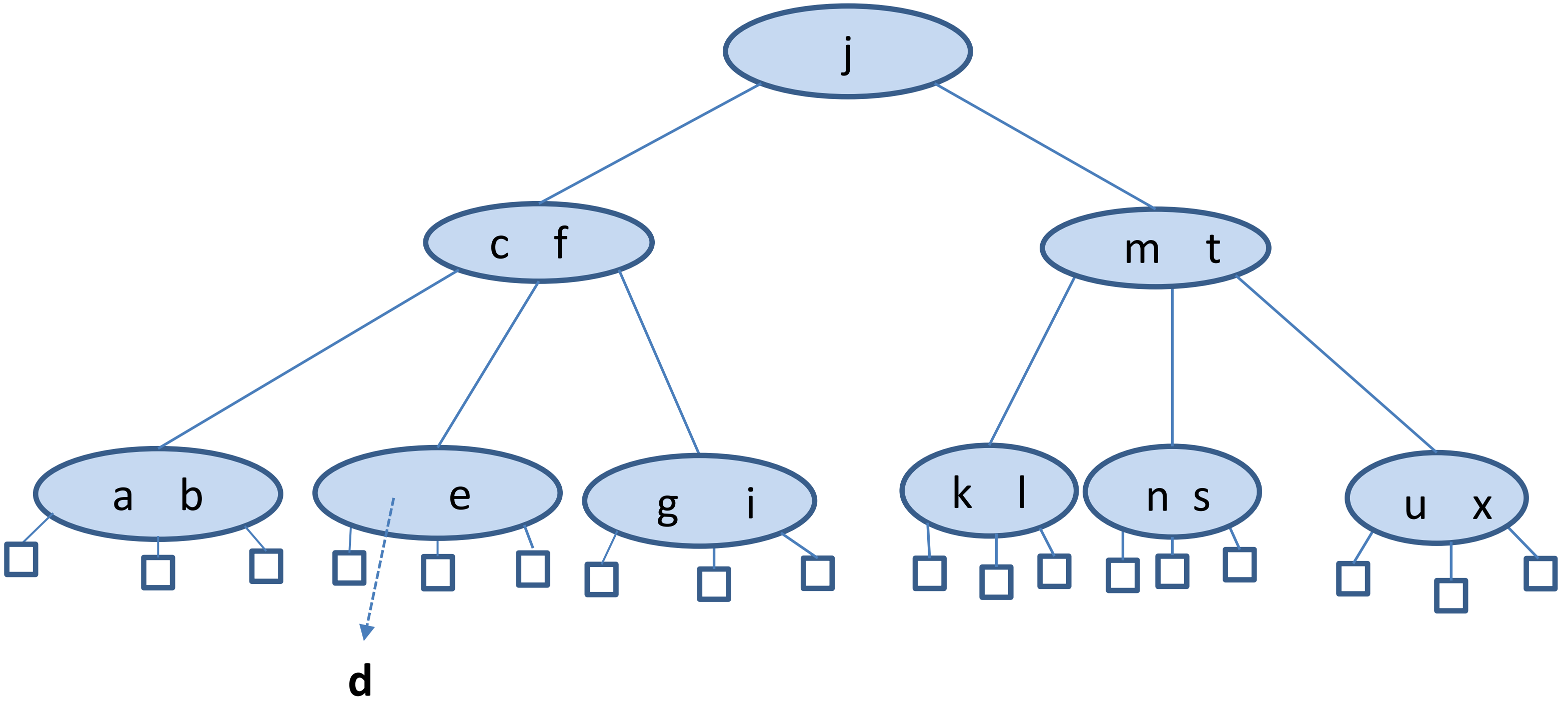
Transfer



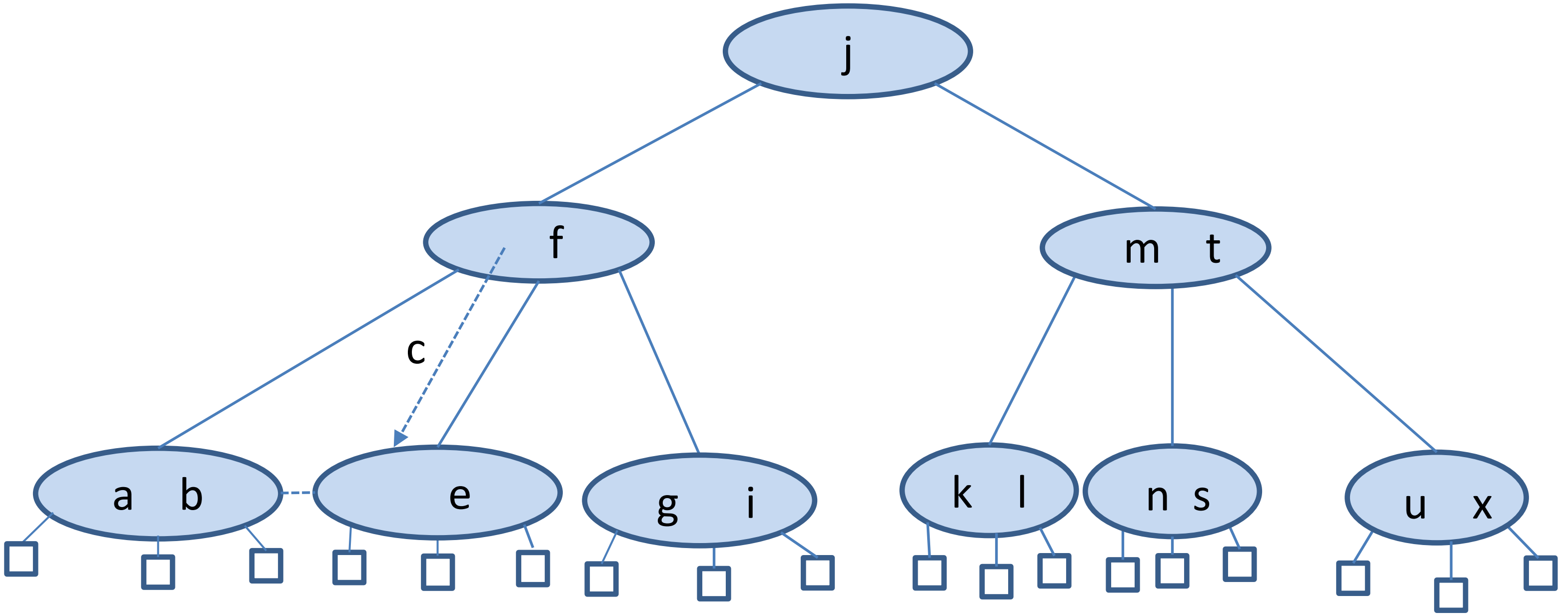
After the Transfer



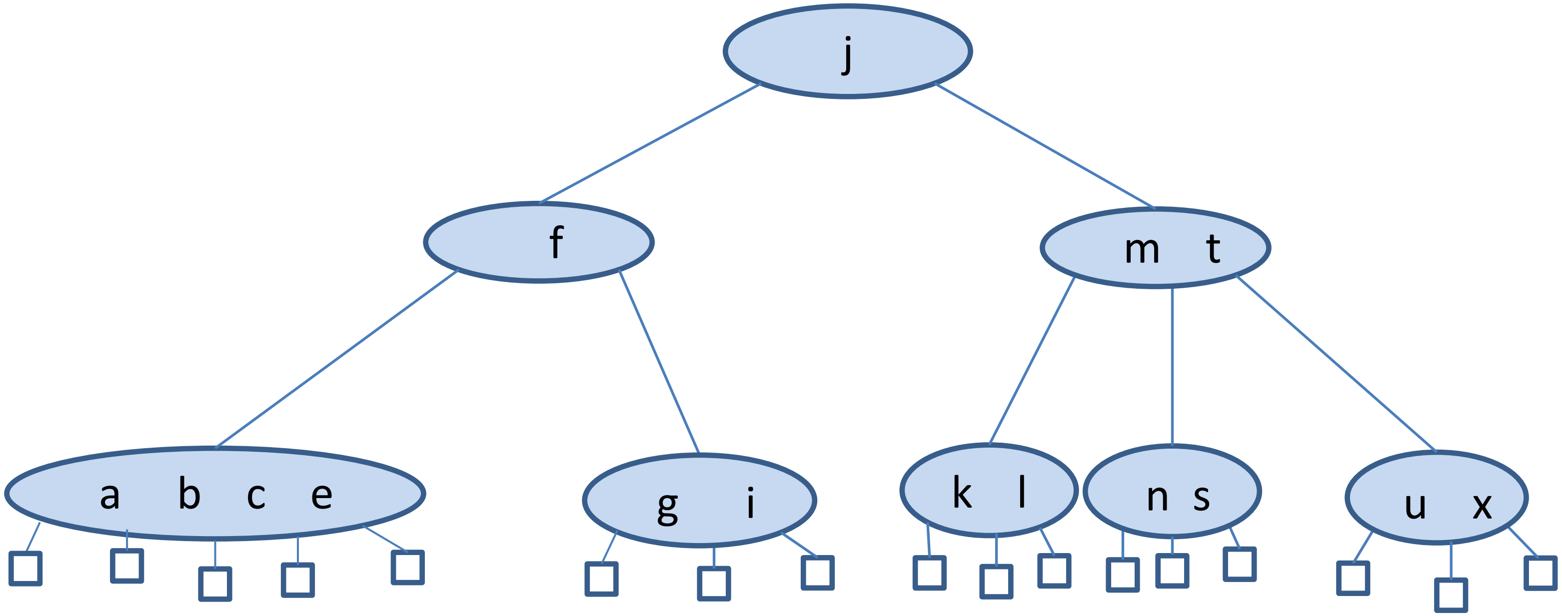
Delete d



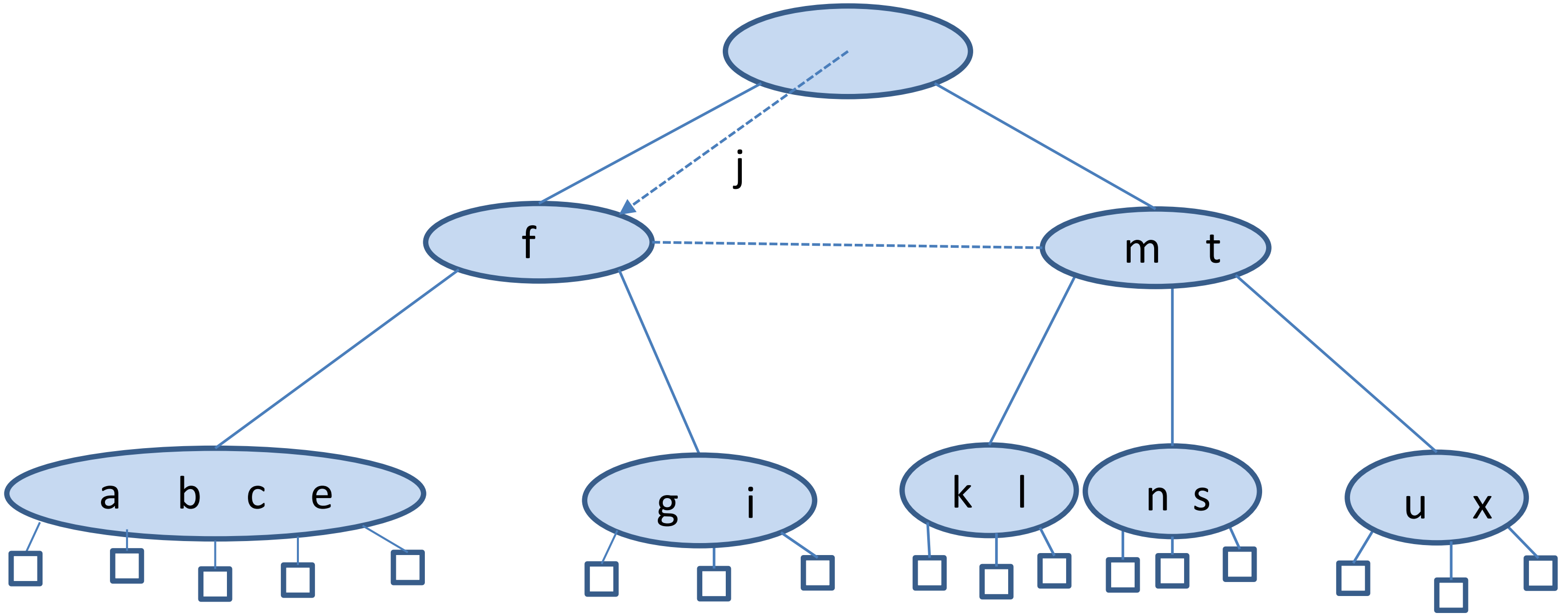
Fusion



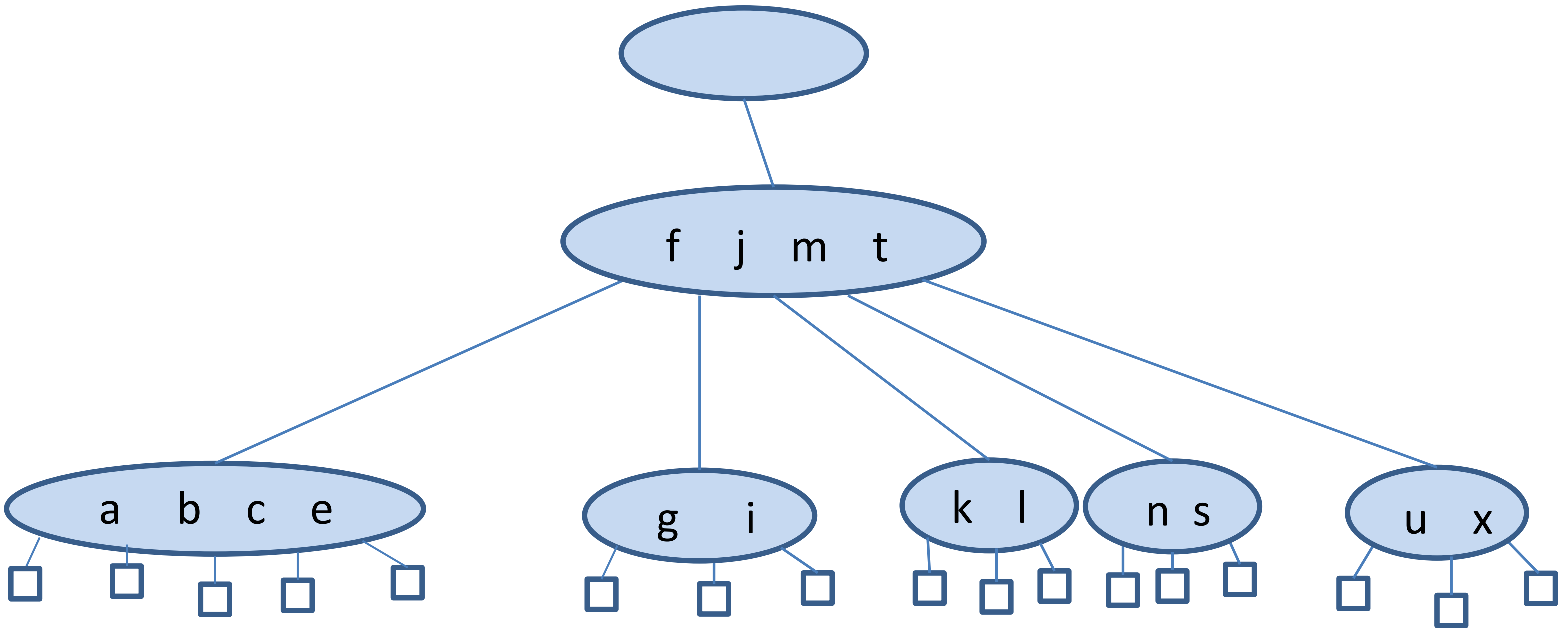
After the Fusion – Underflow at f



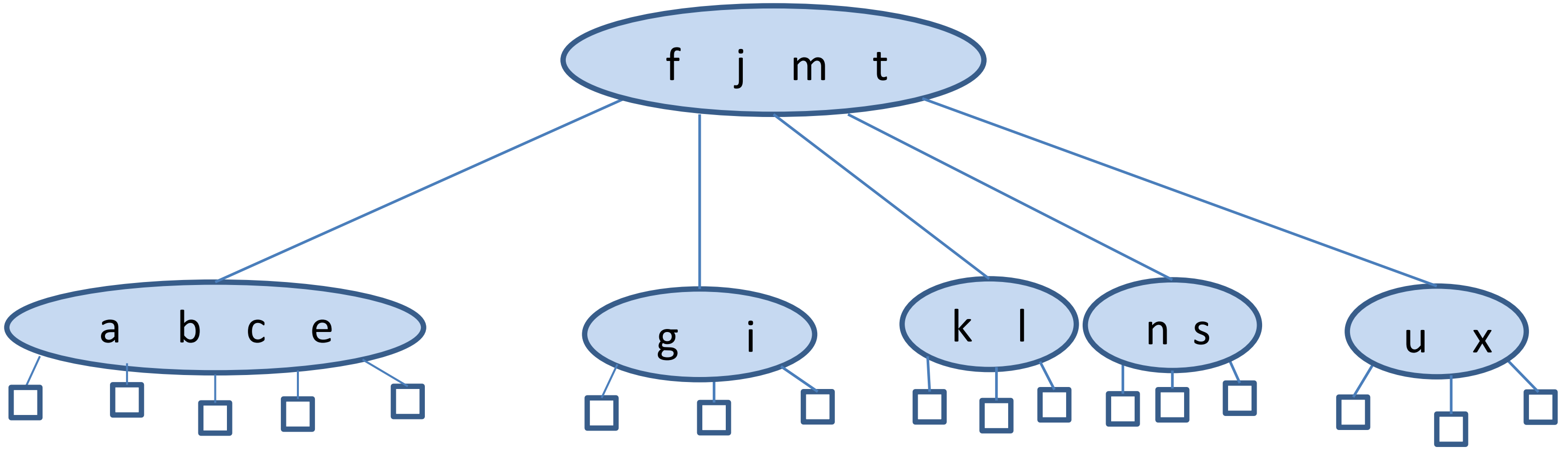
Fusion



After the Fusion – Delete Root



Final Tree



Deletion Function

```
/* DeleteTree: deletes target from the B-tree.
   Pre: target is the key of some entry in the B-tree to which root
   points.
   Post: This entry has been deleted from the B-tree.
   Uses: RecDeleteTree */

Treenode *DeleteTree(Key target, Treenode *root)
{
  Treenode *oldroot;      /* used to dispose of an empty root */

  RecDeleteTree(target, root);
  if (root->count==0){    /* root is empty */
    oldroot=root;
    root=root->branch[0];
    free(oldroot);
  }
  return root;
}
```

Recursive Deletion

- Most of the work is done in the recursive function `RecDeleteTree`.
- This function first searches the current node for the target. If it is found and the node has only internal-node children, then the immediate successor of the key is found and is placed in the current node, and the successor is deleted.
- Deletion from a node with only external-node children is straightforward.
- Otherwise, the function continues recursively. When a recursive call returns, the function checks to see if enough entries remain in the appropriate node, and, if not, moves entries as required.

Recursive Deletion (cont’d)

```
/* RecDeleteTree: look for target to delete.
   Pre: target is the key of some entry in the subtree of a B-tree to which current points.
   Post: This entry has been deleted from the B-tree.
   Uses: RecDeleteTree recursively, SearchNode, Successor, Remove, Restore */

void RecDeleteTree(Key target, Treenode *current)
{
  int pos;          /* location of target or of branch on which to search */
  if (!current){
    printf("Target was not in the B-tree");
    return;         /* Hitting an empty tree is an error */
  } else {
    if (SearchNode(target, current, &pos))
      if (current->branch[pos-1]){
        Successor(current, pos);  /* replaces entry[pos] by its successor */
        RecDeleteTree(current->entry[pos].key, current->branch[pos]);
      } else
        Remove(current, pos);  /* removes key from pos of *current */
    else                    /* Target was not found in the current node */
      RecDeleteTree(target, current->branch[pos]);
    if (current->branch[pos])
      if (current->branch[pos]->count < MIN)
        Restore(current, pos);
  }
}
```

Auxiliary Functions

```
/* Remove: delete an entry and the branch to its right.
   Pre: current points to a node in a B-tree with an entry
   in index pos.
   Post: This entry and the branch to its right are
   removed from *current */


void Remove(Treenode *current, int pos)
{
  int i;        /* index for moving entries */
  for (i=pos+1; i<=current->count; i++){
    current->entry[i-1]=current->entry[i];
    current->branch[i-1]=current->branch[i];
  }
  current->count--;
}
```

Auxiliary Functions (cont’d)

```
/* Successor: replaces an entry by its
   immediate successor.
   Pre: current points to a node in a B-tree with an
   entry in index pos.
   Post: This entry is replaced by its immediate
   successor under order of keys. */

void Successor(Treenode *current, int pos)
{

/* The code is left as exercise */


}
```

Auxiliary Functions (cont’d)

- The function `Restore` implements the transfer or fusion operation required when we have an underflow.
- The transfer operation is implemented by functions `MoveLeft` and `MoveRight`.
- The fusion operation is implemented by function `Combine`.

Auxiliary Functions (cont'd)

```
/* Restore: restore the minimum number of entries.
Pre: current points to a node in a B-tree with an entry in index pos; the branch to
the right of pos has one too few entries.
Post: An entry taken from elsewhere is used to restore the minimum number of entries by
entering it at current->branch[pos].
Uses: MoveLeft, MoveRight, Combine */


void Restore(Treenode *current, int pos)
{
if (pos==0)       /* case: leftmost key */
  if (current->branch[1]->count > MIN)
    MoveLeft(current, 1);
  else
    Combine(current, 1);
  else if (pos == current->count) /*case: rightmost key */
    if (current->branch[pos-1]->count > MIN)
      MoveRight(current, pos);
    else
      Combine(current, pos);
  else if (current->branch[pos-1]->count > MIN)
    MoveRight(current, pos);
  else if (current->branch[pos+1]->count > MIN)
    MoveLeft(current, pos+1);
  else
    Combine(current, pos);
}
```

Auxiliary Functions (cont'd)

```
/* MoveRight: move a key to the right.
   Pre: current points to a node in a B-tree with entries in the branches pos and
   pos-1, with too few entries in branch pos.
   Post: The rightmost entry from branch pos-1 has moved into *current, which has
   sent an entry into the branch pos */


void MoveRight(Treenode *current, int pos)
{
  int c;
  Treenode *t;
  t=current->branch[pos];
  for (c=t->count; c>0; c--){
  /* shift all keys in the right node one position */
    t->entry[c+1]=t->entry[c];
    t->branch[c+1]=t->branch[c];
  }
  t->branch[1]=t->branch[0];   /* move key from parent to right node */
  t->count++;
  t->entry[1]=current->entry[pos];
  t=current->branch[pos-1];   /* move last key of left node into parent */
  current->entry[pos]=t->entry[t->count];
  current->branch[pos]->branch[0]=t->branch[t->count];
  t->count--;
}
```

Auxiliary Functions (cont’d)

```
/* MoveLeft: move a key to the left.
Pre: current points to a node in a B-tree with entries in the branches pos and
pos-1, with too few in branch pos-1.
Post: The leftmost entry from branch pos has moved into *current, which has sent
an entry into the branch pos-1 */

void MoveLeft(Treenode *current, int pos)
{
  int c;
  Treenode *t;
  t=current->branch[pos-1];   /* move key from parent into left node */
  t->count++;
  t->entry[t->count]=current->entry[pos];
  t->branch[t->count]=current->branch[pos]->branch[0];
  t=current->branch[pos];  /* Move first key from right node into parent */
  current->entry[pos]=t->entry[1];
  t->branch[0]=t->branch[1];
  t->count--;
  for (c=1; c<=t->count; c++){
    /* shift all keys in the right node one position leftward */
    t->entry[c]=t->entry[c+1];
    t->branch[c]=t->branch[c+1];
  }
}
```

Auxiliary Functions

```
/* Combine: combine adjacent nodes.
   Pre: current points to a node in a B-tree with entries in the branches pos and
   pos-1, with too few to move entries.
   Post: The nodes at branches pos-1 and pos have been combined into one node,
   which also includes the entry formerly in *current at index pos. */

void Combine(Treenode *current, int pos)
{
  int c;
  Treenode *right;
  Treenode *left;
  right=current->branch[pos];
  left=current->branch[pos-1];    /* work with the left node */
  left->count++;                  /* insert the key from the parent */
  left->entry[left->count]=current->entry[pos];
  left->branch[left->count]=right->branch[0];
  for (c=1; c<=right->count; c++){ /* insert all keys from right node */
    left->count++;
    left->entry[left->count]=right->entry[c];
    left->branch[left->count]=right->branch[c];
  }
  for (c=pos; c< current->count; c++){  /* delete key from parent node */
    current->entry[c]=current->entry[c+1];
    current->branch[c]=current->branch[c+1];
  }
  current->count--;
  free(right);           /* dispose of the empty right node */
}
```

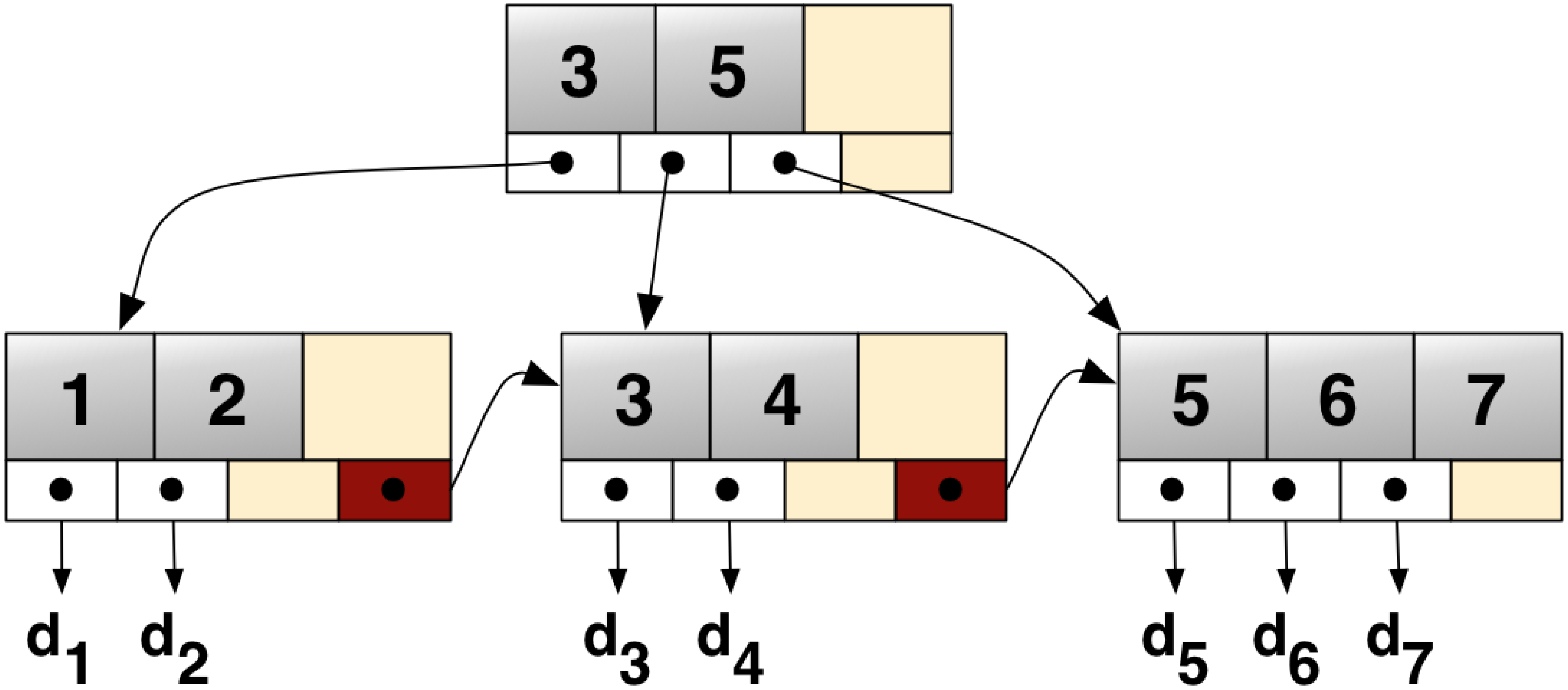
Complexity of Operations in a B-tree

- 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



Readings

- The code we presented is from the following book:
  - R. Kruse, C. L. Tondo and B. Leung. Data Structures and Program Design in C.
    - Chapter 10
- The theoretical results are from the following books:
  - M. T. Goodrich, R. Tamassia and D. Mount. Data Structures and Algorithms in C++. 2nd edition. John Wiley.
  - Sartaj Sahni. Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++. Εκδόσεις Τζιόλα.
- R. Sedgewick. Αλγόριθμοι σε C.
  - Κεφ. 16.3