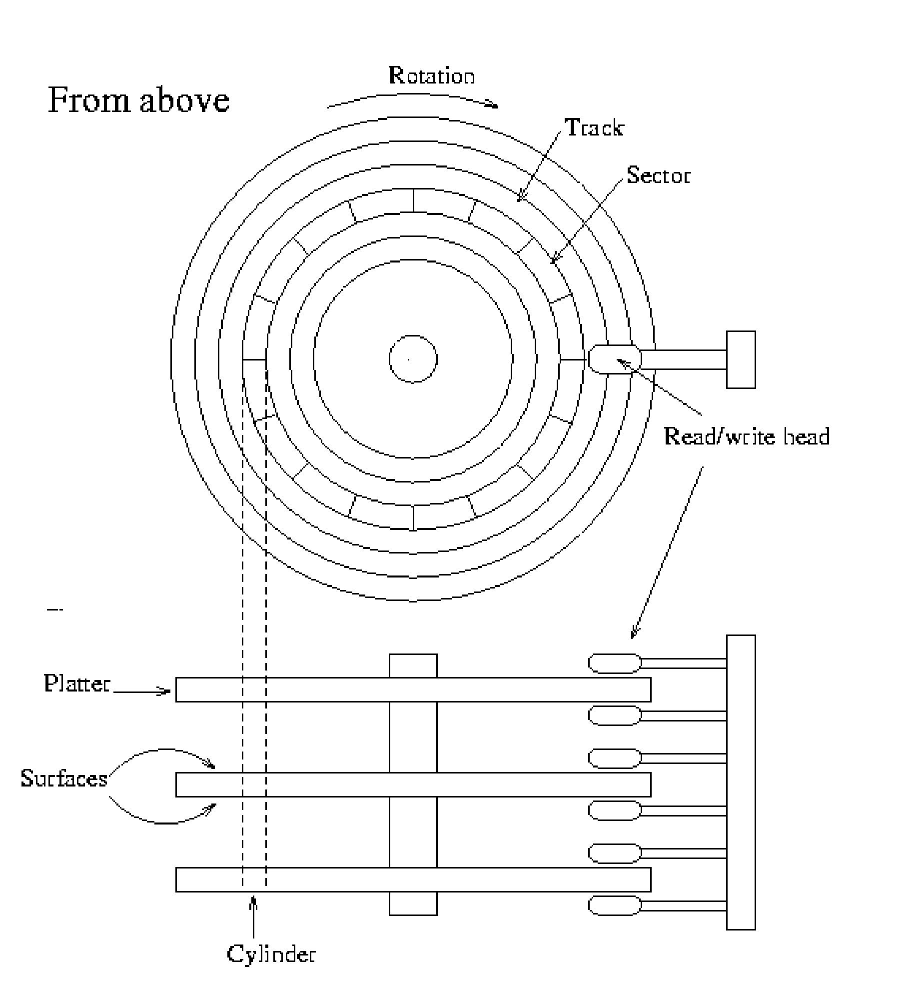
Disk Based Data Structures

- So far search trees were limited to main memory structures
  - Assumption: the dataset organized in a search tree fits in main memory (including the tree overhead)
- Counter-example: transaction data of a bank > 1 GB per day
  - use secondary storage media (punch cards, hard disks, magnetic tapes, etc.)
- Consequence: make a search tree structure secondary-storage-enabled

Hard Disks

- **Large amounts of storage, but slow access!**
- Identifying a page takes a long time (seek time plus rotational delay – 5-10ms), reading it is fast
  - It pays off to read or write data in **pages** (or blocks) of 2-16 Kb in size.



Algorithm analysis

- The running time of disk-based algorithms is measured in terms of
  - computing time (CPU)
  - number of disk accesses
    - sequential reads
    - random reads

- Regular main-memory algorithms that work one data element at a time can not be “ported” to secondary storage in a straight-forward way

Principles

- Pointers in data structures are no longer addresses in main memory but locations in files
- If *x* is a pointer to an object
  - if *x* is in main memory *key*[*x*] refers to it
  - otherwise DiskRead(*x*) reads the object from disk into main memory (DiskWrite(*x*) – writes it back to disk)

Principles (2)

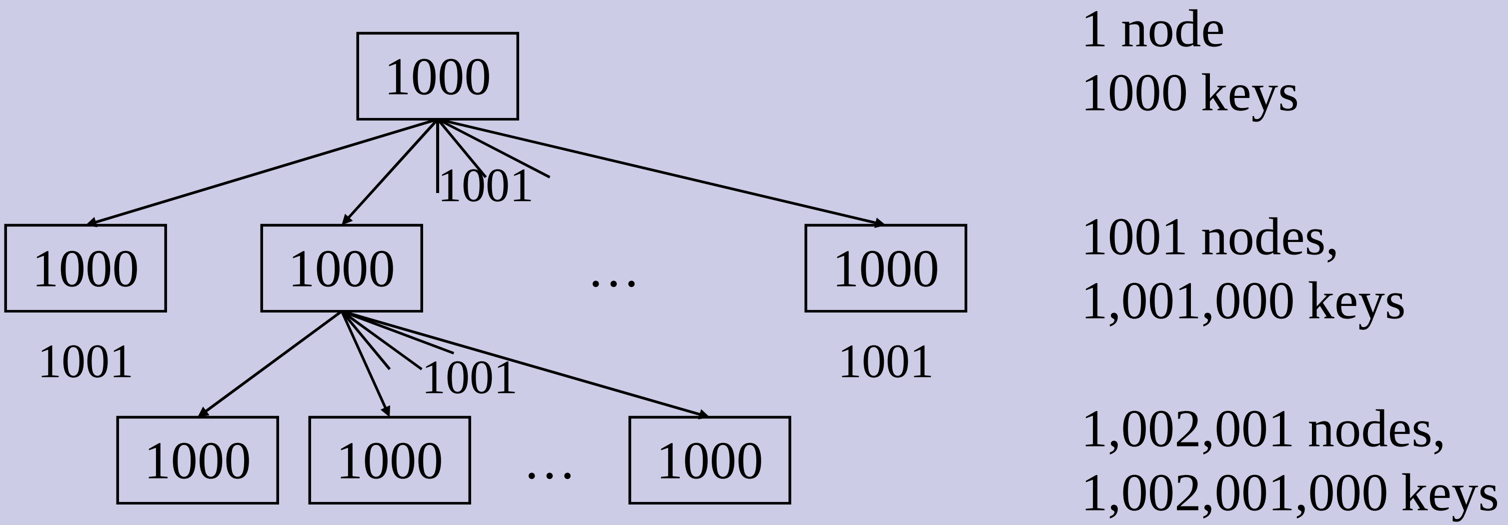
- A typical working pattern

```
…
x ← a pointer to some object
DiskRead(x)
operations that access and/or modify x
DiskWrite(x) //omitted if nothing changed
other operations, only access no modify
…
```

- Operations:
  - DiskRead(x:pointer_to_a_node)
  - DiskWrite(x:pointer_to_a_node)
  - AllocateNode():pointer_to_a_node

Binary-trees vs. B-trees

- Size of B-tree nodes is determined by the page size. One page – one node.
- A B-tree of height 2 may contain > 1 billion keys!
- Heights of Binary-tree and B-tree are logarithmic
  - B-tree: logarithm of base, e.g., 1000
  - Binary-tree: logarithm of base 2



B-tree Definitions

- Node x has fields
  - $n[x]$: the number of keys of that the node
  - $key_1[x] \leq \ldots \leq key_{n[x]}[x]$: the keys in ascending order
  - leaf[x]: true if leaf node, false if internal node
  - if internal node, then $c_1[x], \ldots, c_{n[x]+1}[x]$: pointers to children
- Keys separate the ranges of keys in the sub-trees. If k_i is an arbitrary key in the subtree $c_i[x]$ then $k_i \leq key_i[x] \leq k_{i+1}$

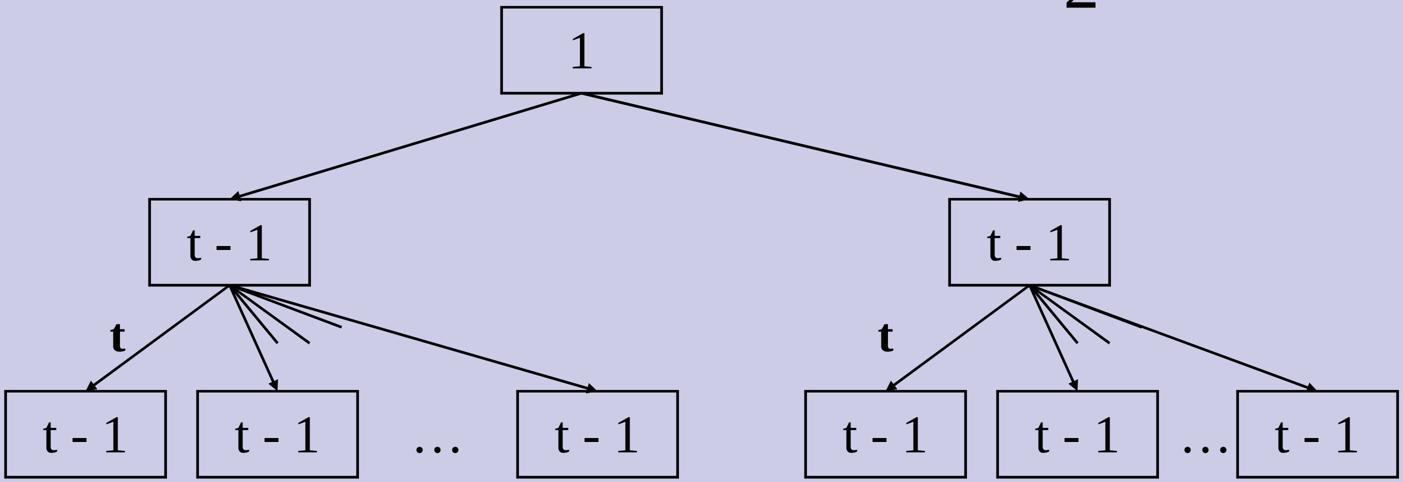
B-tree Definitions (2)

- Every leaf has the same depth
- In a B-tree of a **degree** *t* all nodes except the root node have between *t* and 2*t* children (i.e., between *t*–1 and 2*t*–1 keys).
- The root node has between 0 and 2*t* children (i.e., between 0 and 2*t*–1 keys)

Height of a B-tree

- B-tree *T* of height *h*, containing $n \geq 1$ keys and minimum degree $t \geq 2$, the following restriction on the height holds:

$$h \leq \log_t \frac{n+1}{2}$$



depth	#of nodes
0	1
1	2
2	2t

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1$$

B-tree Operations

- An implementation needs to suport the following B-tree operations
  - **Searching** (simple)
  - **Creating** an empty tree (trivial)
  - **Insertion** (complex)
  - **Deletion** (complex)

Searching

- Straightforward generalization of a binary tree search

```
BTreeSearch(x,k)
i ← 1
while i ≤ n[x] and k > key_i[x]
   i ← i+1
if i ≤ n[x] and k = key_i[x] then
   return(x,i)
if leaf[x] then
   return NIL
   else DiskRead(c_i[x])
        return BTtreeSearch(c_i[x],k)
```

Creating an Empty Tree

- Empty B-tree = create a root & write it to disk!

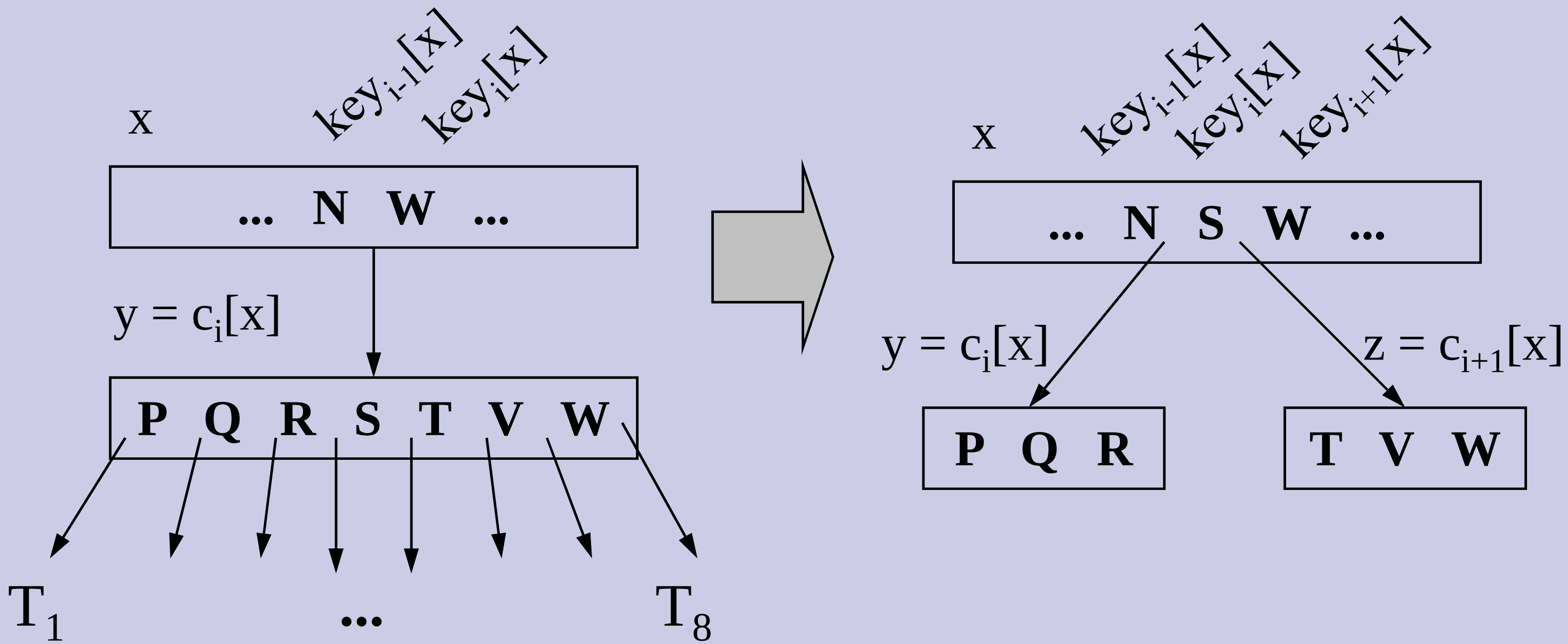
```
BTreeCreate(T)
01 x ← AllocateNode();
02 leaf[x] ← TRUE;
03 n[x] ← 0;
04 DiskWrite(x);
05 root[T] ← x
```

Splitting Nodes

- Nodes fill up and reach their maximum capacity $2t - 1$
- Before we can insert a new key, we have to “make room,” i.e., split nodes

Splitting Nodes (2)

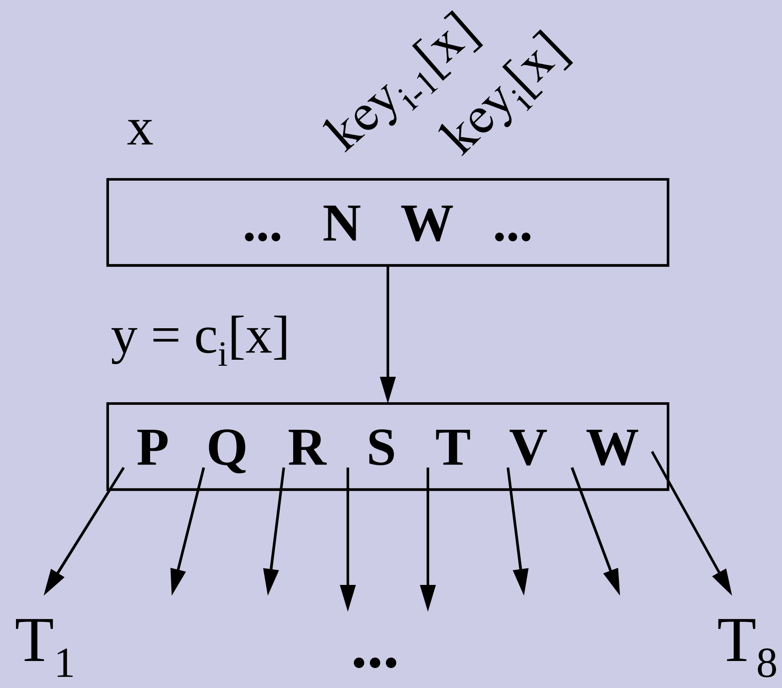
- Result: one key of *x* moves up to parent + 2 nodes with *t*-1 keys



Splitting Nodes (2)

```
BTreeSplitChild(x,i,y)
 z ← AllocateNode()
 leaf[z] ← leaf[y]
 n[z] ← t-1
 for j ← 1 to t-1
    key_j[z] ← key_j+t[y]
 if not leaf[y] then
    for j ← 1 to t
       c_j[z] ← c_j+t[y]
 n[y] ← t-1
 for j ← n[x]+1 downto i+1
    c_j+1[x] ← c_j[x]
 c_i+1[x] ← z
 for j ← n[x] downto i
    key_j+1[x] ← key_j[x]
 key_i[x] ← key_t[y]
 n[x] ← n[x]+1
 DiskWrite(y)
 DiskWrite(z)
 DiskWrite(x)
```

x: parent node
y: node to be split and child of x
i: index in x
z: new node



Split: Running Time

- A local operation that does not traverse the tree
- $\Theta(t)$ CPU-time, since two loops run t times
- 3 I/Os

Inserting Keys

- Done recursively, by starting from the root and recursively traversing down the tree to the leaf level
- Before descending to a lower level in the tree, make sure that the node contains < $2t - 1$ keys:
  - so that if we split a node in a lower level we will have space to include a new key

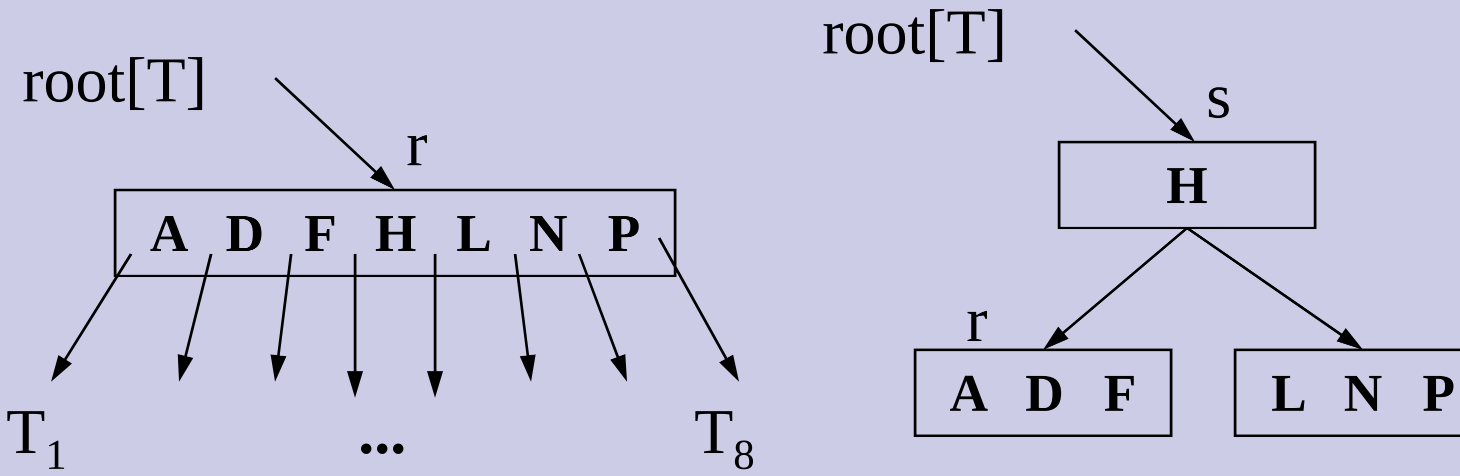
Inserting Keys (2)

- Special case: root is full (BtreeInsert)

```
BTreeInsert(T)
 r ← root[T]
 if n[r] = 2t – 1 then
    s ← AllocateNode()
    root[T] ← s
    leaf[s] ← FALSE
    n[s] ← 0
    c1[s] ← r
    BTreeSplitChild(s,1,r)
    BTreeInsertNonFull(s,k)
 else BTreeInsertNonFull(r,k)
```

Splitting the Root

- Splitting the root requires the creation of a new root



- The tree grows at the top instead of the bottom

Inserting Keys

- BtreeNonFull tries to insert a key *k* into a node *x*, which is **assumed to be non-full** when the procedure is called
- BTreeInsert and the recursion in BTreeInsertNonFull guarantee that this assumption is true!

Inserting Keys: Pseudo Code

BTreeInsertNonFull(x,k)

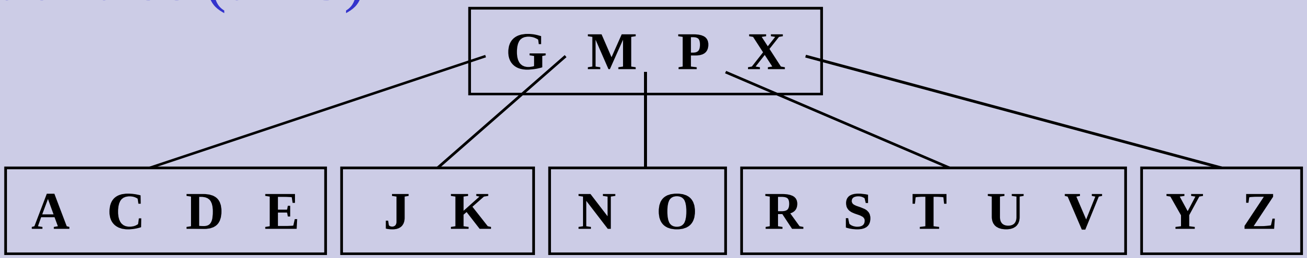
```
i ← n[x]
if leaf[x] then
   while i ≥ 1 and k < key_i[x]
      key_i+1[x] ← key_i[x]
      i ← i - 1
   key_i+1[x] ← k
   n[x] ← n[x] + 1
   DiskWrite(x)
else while i ≥ 1 and k < key_i[x]
      i ← i - 1
   i ← i + 1
   DiskRead c_i[x]
   if n[c_i[x]] = 2t – 1 then
      BTreeSplitChild(x,i,c_i[x])
      if k > key_i[x] then
         i ← i + 1
   BTreeInsertNonFull(c_i[x],k)
```

leaf insertion

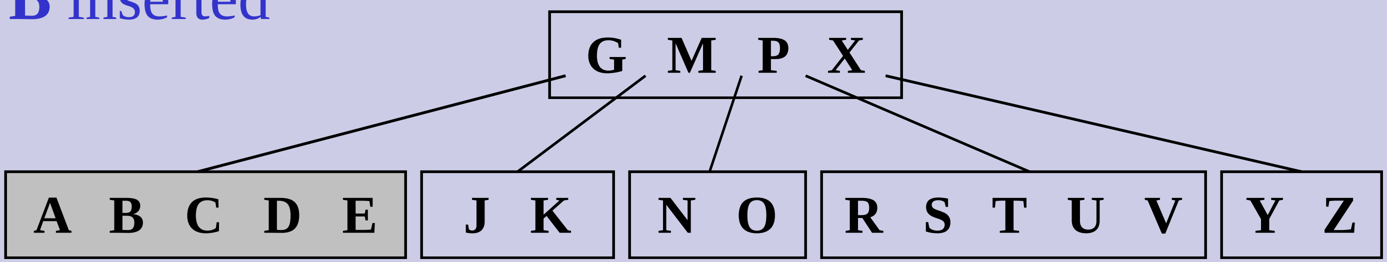
internal node: traversing tree

Insertion: Example

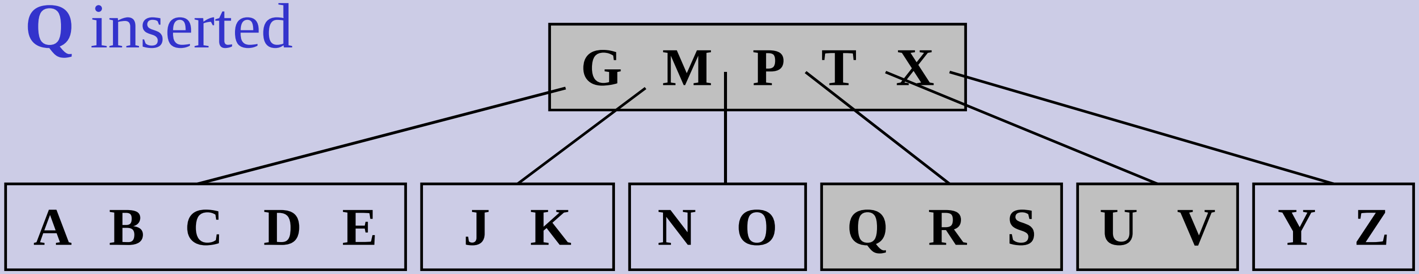
initial tree (t = 3)



B inserted

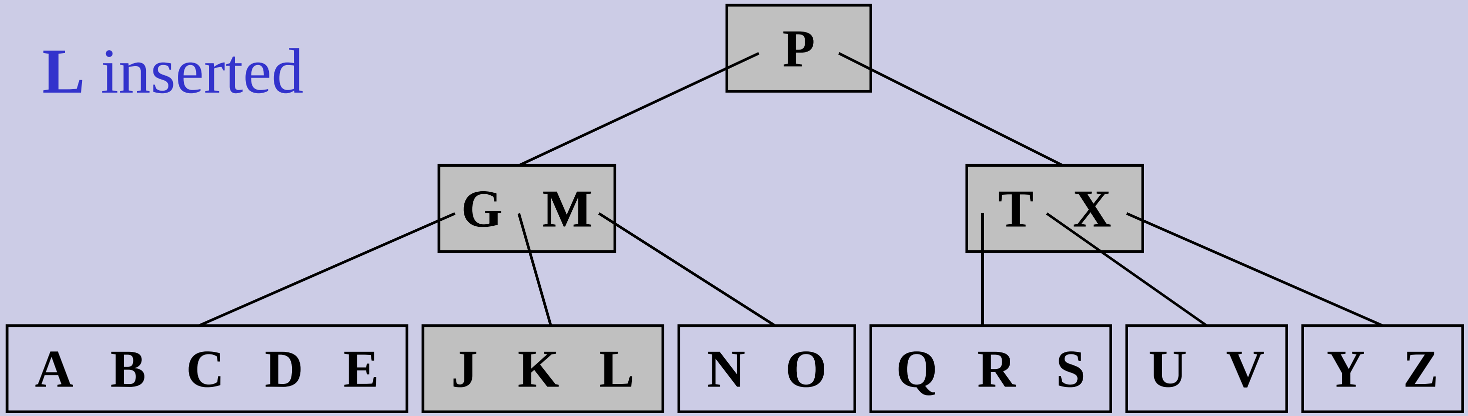


Q inserted

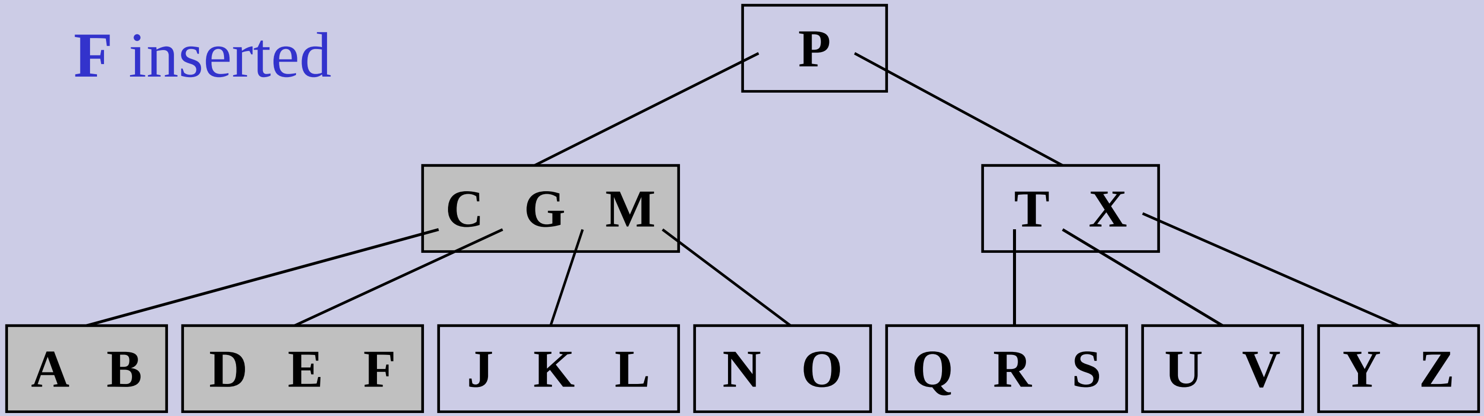


Insertion: Example (2)

L inserted



F inserted



Insertion: Running Time

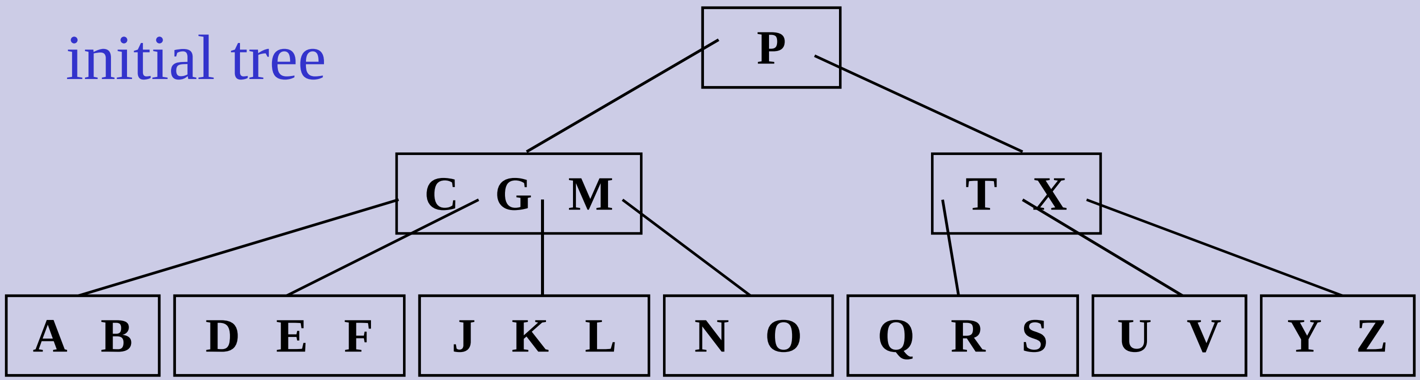
- Disk I/O: $O(h)$, since only $O(1)$ disk accesses are performed during recursive calls of BTreeInsertNonFull
- CPU: $O(th) = O(t \log_t n)$
- At any given time there are $O(1)$ number of disk pages in main memory

Deleting Keys

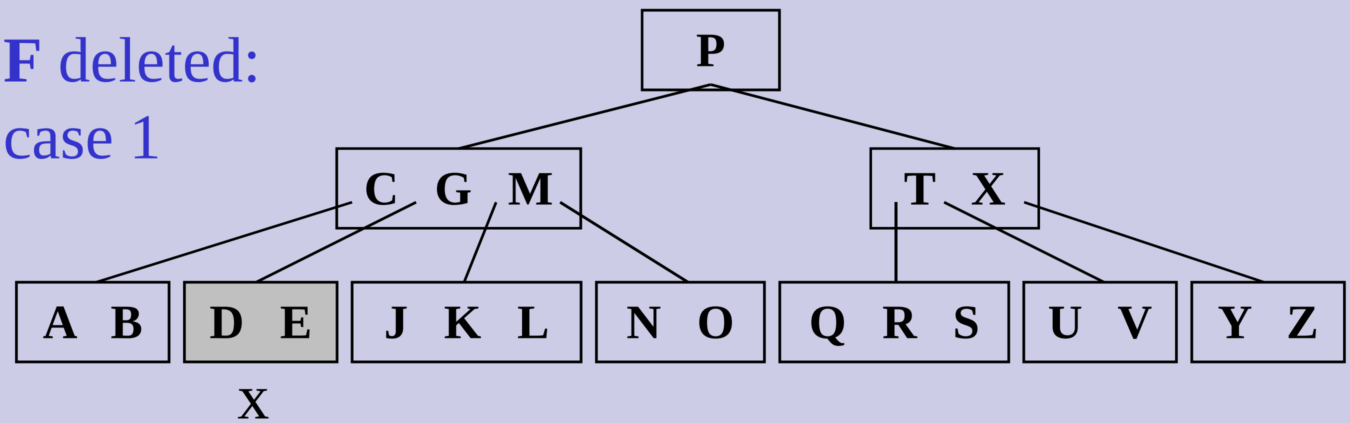
- Done recursively, by starting from the root and recursively traversing down the tree to the leaf level
- Before descending to a lower level in the tree, make sure that the node contains $\geq t$ keys (cf. insertion $< 2t - 1$ keys)
- BtreeDelete distinguishes three different stages/scenarios for deletion
  - Case 1: key *k* found in leaf node
  - Case 2: key *k* found in internal node
  - Case 3: key *k* suspected in lower level node

Deleting Keys (2)

initial tree



F deleted:
case 1

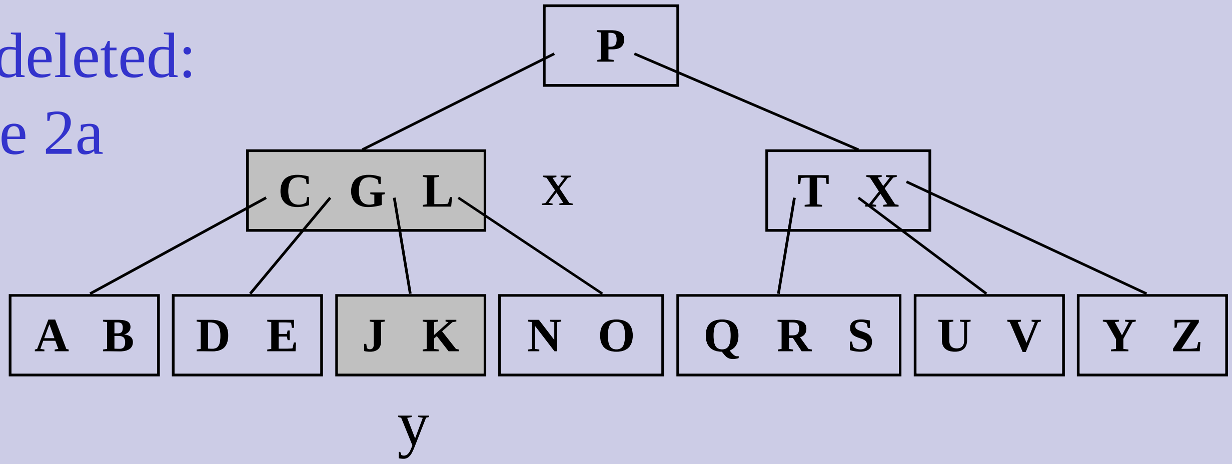


- Case 1: If the key *k* is in node *x*, and *x* is a leaf, delete *k* from *x*

Deleting Keys (3)

- Case 2: If the key *k* is in node *x*, and *x* is not a leaf, delete *k* from *x*
  - a) If the child *y* that precedes *k* in node *x* has at least *t* keys, then find the predecessor *k*' of *k* in the sub-tree rooted at *y*. Recursively delete *k*', and replace *k* with *k*' in *x*.
  - b) Symmetrically for successor node *z*

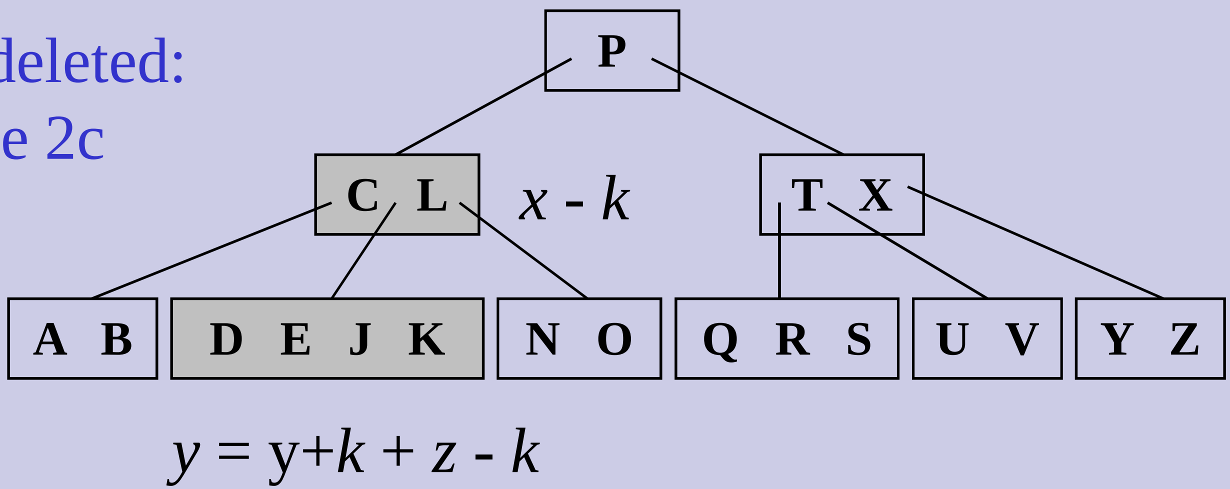
M deleted:
case 2a



Deleting Keys (4)

- If both *y* and *z* have only *t* –1 keys, ***merge*** *k* with the contents of *z* into *y*, so that *x* loses both *k* and the pointers to *z*, and *y* now contains 2*t* – 1 keys. Free *z* and recursively delete *k* from *y*.

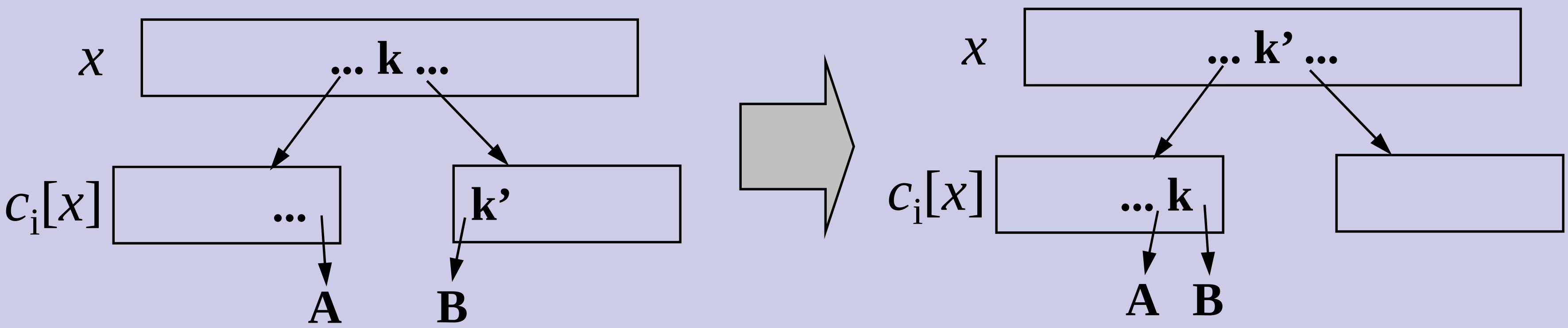
G deleted:
case 2c



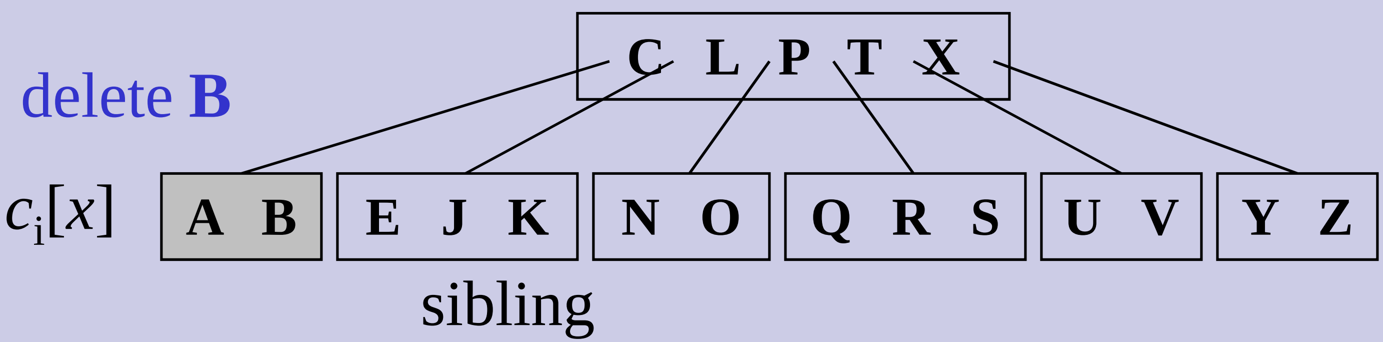
Deleting Keys - Distribution

- Descending down the tree: if *k* not found in current node *x*, find the sub-tree $c_i[x]$ that has to contain k.
- If $c_i[x]$ has only $t - 1$ keys take action to ensure that we descent to a node of size at least *t.*
- We can encounter two cases.
  - If $c_i[x]$ has only t-1 keys, but a sibling with at least *t* keys, give $c_i[x]$ an extra key by moving a key from *x* to $c_i[x]$, moving a key from $c_i[x]$'s immediate left and right sibling up into *x*, and moving the appropriate child from the sibling into $c_i[x]$ - ***distribution***

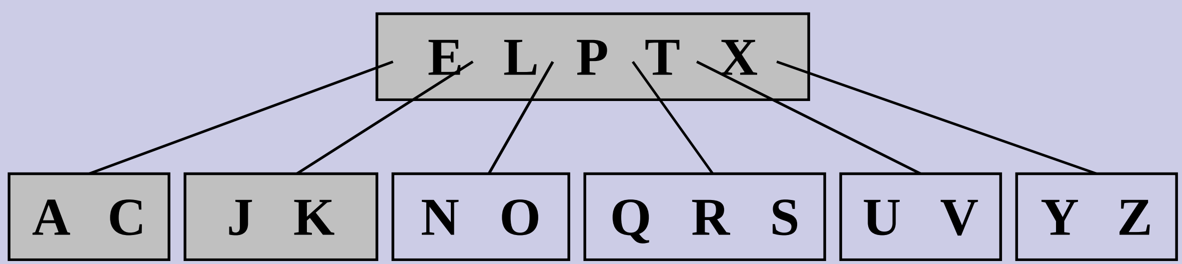
Deleting Keys – Distribution(2)



delete **B**

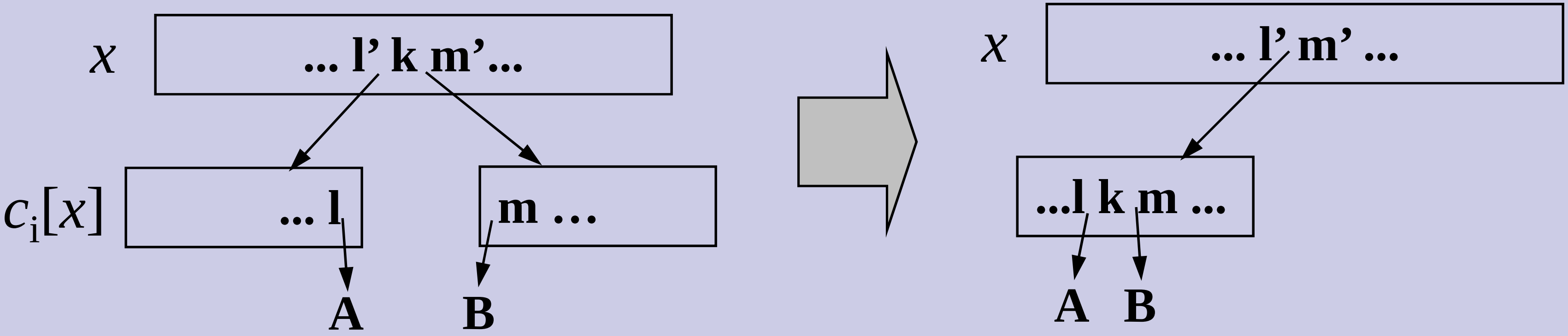


B deleted:



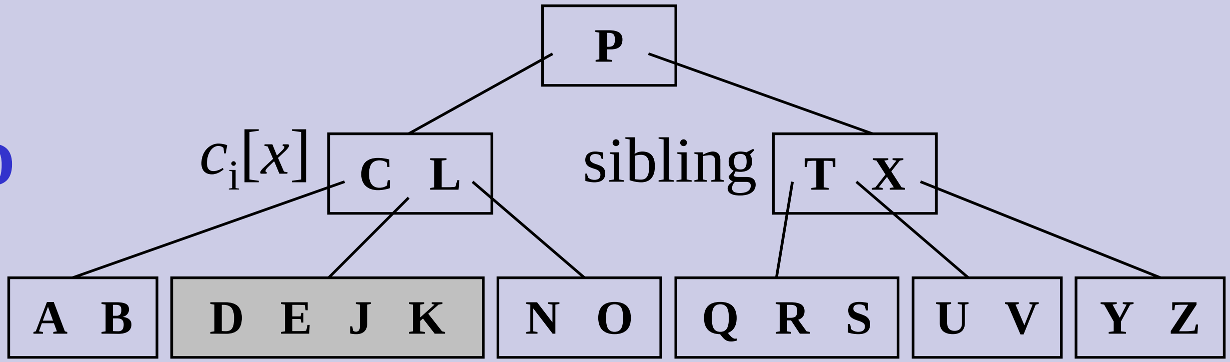
Deleting Keys - Merging

- If $c_i[x]$ and both of $c_i[x]$'s siblings have t – 1 keys, ***merge*** c_i with one sibling, which involves moving a key from *x* down into the new merged node to become the median key for that node

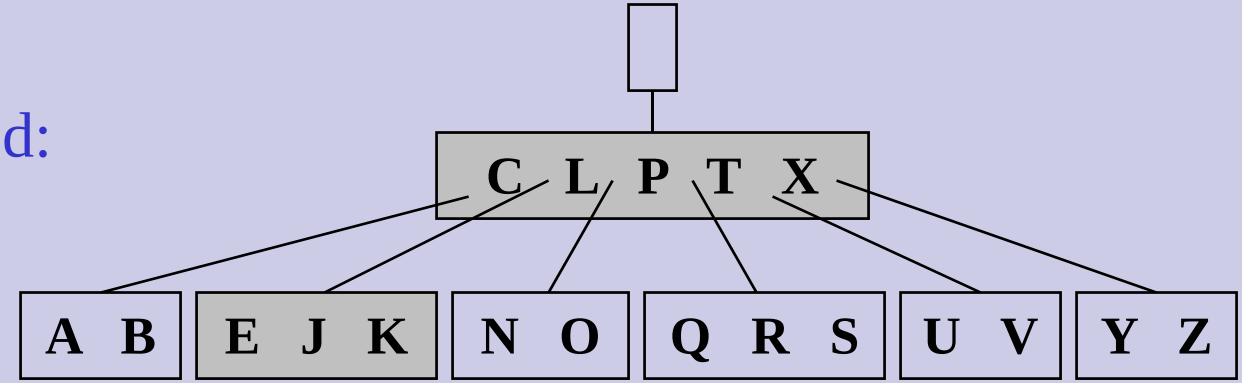


Deleting Keys – Merging (2)

delete **D**



D deleted:



tree shrinks in height

Deletion: Running Time

- Most of the keys are in the leaf, thus deletion most often occurs there!
- In this case deletion happens in one downward pass to the leaf level of the tree
- Deletion from an internal node might require "backing up" (case 2)
- Disk I/O: O(h), since only O(1) disk operations are produced during recursive calls
- CPU: $O(th) = O(t \log_t n)$

Two-pass Operations

- Simpler, practical versions of algorithms use two passes (down and up the tree):
  - *Down* – Find the node where deletion or insertion should occur
  - *Up* – If needed, split, merge, or distribute; propagate splits, merges, or distributes up the tree
- To avoid reading the same nodes twice, use a buffer of nodes