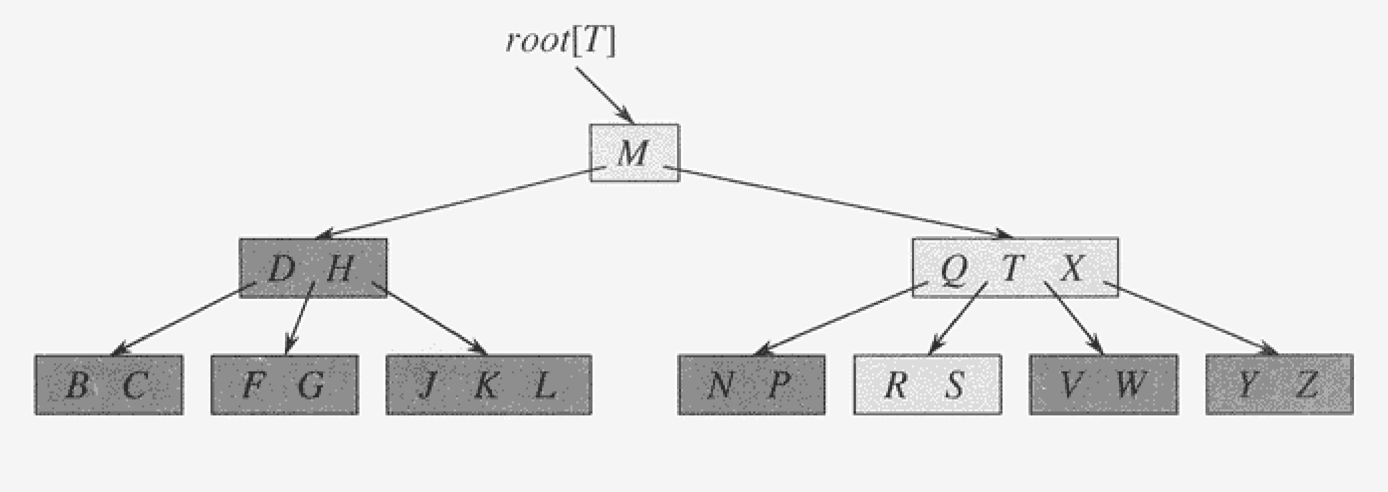
B-TREES

- B-trees are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices.
- B-tree nodes may have many children, from a handful to thousands. That is called the "branching factor" of a B-tree.
- Every n-node B-tree has height $O(\lg n)$, therefore, B-trees can be used to implement many dynamic-set operations in time $O(\lg n)$.
- B-trees generalize binary search trees in a natural manner.
  - If a B-tree node x contains $n[x]$ keys, then x has $n[x] + 1$ children.
  - The keys in node x are used as dividing points separating the range of keys handled by x into $n[x] + 1$ subranges, each handled by one child of x.
  - When searching for a key in a B-tree, we make an $(n[x] + 1)$-way decision based on comparisons with the $n[x]$ keys stored at node x.

B-tree example

- A B-tree whose keys are the consonants of English.
- An internal node x containing $n[x]$ keys has $n[x] + 1$ children.
- All leaves are at the same depth in the tree.
- The lightly shaded nodes are examined in a search for the letter R.



Definition of B-trees

A ***B-tree*** T is a rooted tree (with root $root[T]$) having the following properties.

1. Every node x has the following fields:
   a. $n[x]$, the number of keys currently stored in node x,
   b. the $n[x]$ keys themselves, stored in nondecreasing order: $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$, and
   c. $leaf[x]$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. If x is an internal node, it also contains $n[x] + 1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq \mathrm{key}_{n[x]}[x] \leq k_{n[x]+1}$$

4. Every leaf has the same depth, which is the tree's height h.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \geq 2$ called the ***minimum degree*** of the B-tree:
   a. Every node other than the root must have at least t - 1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
   b. Every node can contain at most 2t - 1 keys. Therefore, an internal node can have at most 2t children. We say that a node is ***full*** if it contains exactly 2t - 1 keys.

The height of a B-tree

Theorem

- If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

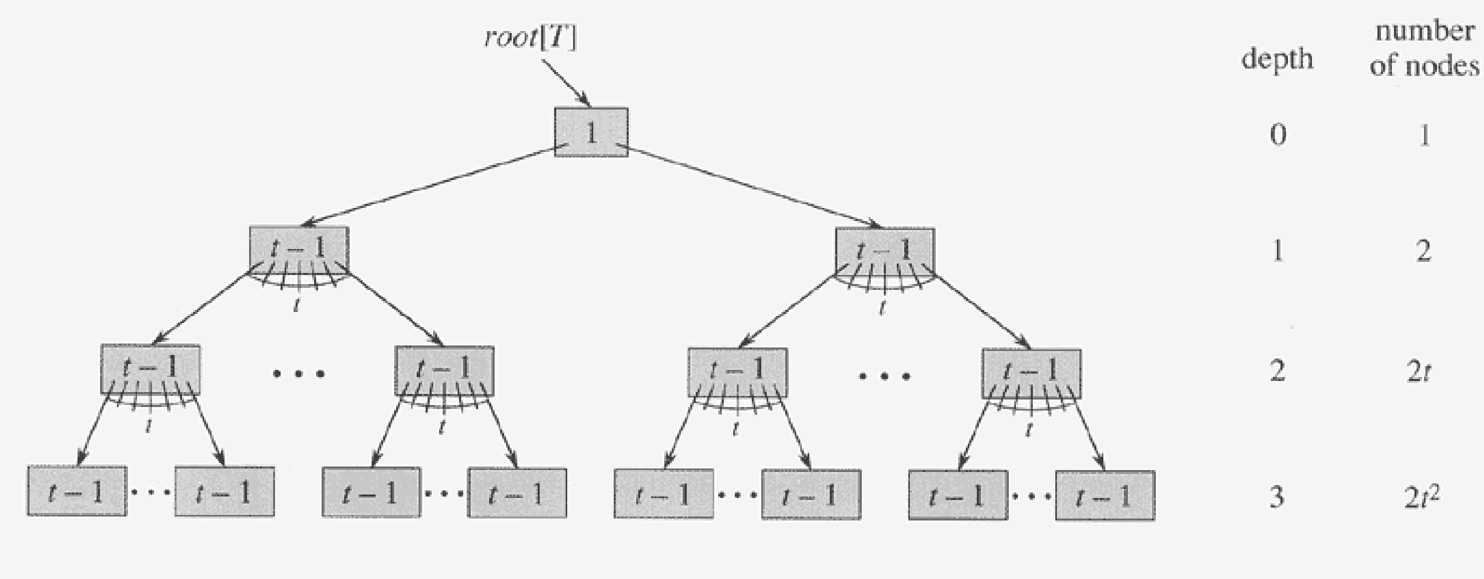
$$h \leq \log_i \frac{n+1}{2}$$

Proof

- If a B-tree has height h, the number of its nodes is minimized when the root contains one key and all other nodes contain t - 1 keys.
- In this case, there are 2 nodes at depth 1, 2t nodes at depth 2, $2t^2$ nodes at depth 3, and so on, until at depth h there are $2t^{h-1}$ nodes.
- Thus, the number n of keys satisfies the inequality

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

  which implies the theorem.



A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is n[x].

Basic operations on B-trees

Conventions:

- The root of the B-tree is always in main memory, so that a DISK-READ on the root is never required; a DISK-WRITE of the root is required, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.
- The procedures are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.

Searching a B-tree

```
B-TREE-SEARCH(x, k)
i ← 1
while i ≤ n[x] and k ≥ key_i[x]
      do i ← i + 1
if i ≤ n[x] and k = key_i[x]
      then return (x, i)
if leaf [x]
```

```
7          then return NIL
8          else DISK-READ(c_i[x])
9                return B-TREE-SEARCH(c_i[x], k)
```

- Using a linear-search procedure, lines 1-3 find the smallest i such that $k \le key_i[x]$, or else they set i to $n[x] + 1$.
- Lines 4-5 check to see if we have now discovered the key, returning if we have.
- Lines 6-9 either terminate the search unsuccessfully (if x is a leaf) or recurse to search the appropriate subtree of x, after performing the necessary DISK-READ on that child.

- Searching a B-tree is much like searching a binary search tree, except that instead of making a binary branching decision at each node, at each internal node x, we make an $(n[x] + 1)$-way branching decision.
- B-TREE-SEARCH takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree.
- The top-level call is thus of the form B-TREE-SEARCH(*root*[T], k). If k is in the B-tree, B-TREE-SEARCH returns the ordered pair (y,i) consisting of a node y and an index i such that $key_i[y] = k$. Otherwise, the value NIL is returned.

Complexity of B-TREE-SEARCH

- The nodes encountered during the recursion form a path downward from the root of the tree.
- The number of disk pages accessed by B-TREE-SEARCH is therefore $\Theta(h) = \Theta(\log_t n)$, where h is the height of the B-tree and n is the number of keys in the B-tree.
- Since $n[x] < 2t$, the time taken by the **while** loop of lines 2-3 within each node is $O(t)$, and the total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

- To build a B-tree T, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys.
- Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in $O(1)$ time.
- We can assume that a node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.

```
B-TREE-CREATE(T)
1 x ← ALLOCATE-NODE()
2 leaf[x] ← TRUE
3 n[x] ← 0
4 DISK-WRITE(x)
5 root[T] ← x
```

- B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

Splitting a node in a B-tree

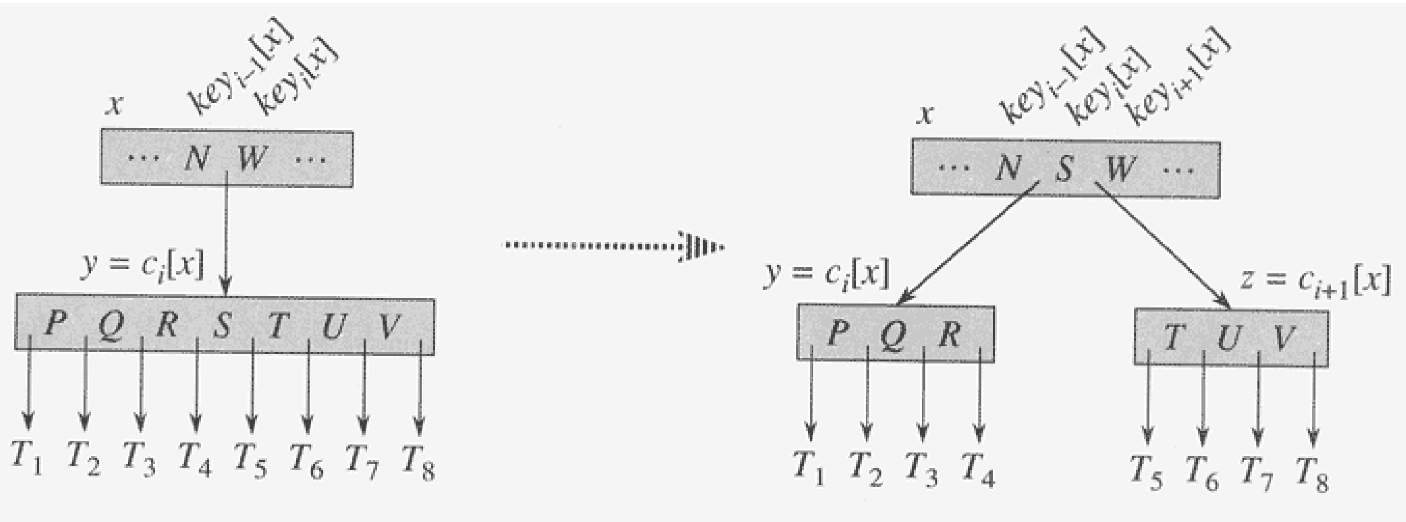
- Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree.
- A fundamental operation used during insertion is the ***splitting*** of a full node y (having $2t$ - 1 keys) around its ***median key*** $key_t[y]$ into two nodes having t - 1 keys each.
- The median key moves up into y's parent - which must be nonfull prior to the splitting of y - to identify the dividing point between the two new trees; if y has no parent, then the tree grows in height by one.
- Splitting, then, is the means by which the tree grows.

- The procedure B-TREE-SPLIT-CHILD takes as input a *nonfull* internal node x (assumed to be in main memory), an index i, and a node y such that $y = c_i[x]$ is a *full* child of x.
- The procedure then splits this child in two and adjusts x so that it now has an additional child.

```
B-TREE-SPLIT-CHILD(x,i,y)
z ← ALLOCATE-NODE()
leaf[z] ← leaf[y]
n[z] ← t - 1
for j ← 1 to t - 1
      do keyj[z] ← keyj+t[y]
if not leaf [y]
     then for j ← 1 to t
       do cj[z] ← cj+t[y]
n[y] ← t - 1
for j ← n[x] + 1 downto i + 1
      do cj+1[x] ← cj[x]
ci+1[x] ← z
for j ← n[x] downto i
      do keyj+1[x] ← keyj[x]
keyi[x] ← keyt[y]
n[x] ← n[x] + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

- B-TREE-SPLIT-CHILD works by straightforward "cutting and pasting. "
- Here, y is the ith child of x and is the node being split.

- Node y originally has $2t$ - 1 children but is reduced to t - 1 children by this operation.
- Node z "adopts" the t - 1 largest children of y, and z becomes a new child of x, positioned just after y in x's table of children.
- The median key of y moves up to become the key in x that separates y and z.
- Lines 1-8 create node z and give it the larger t - 1 keys and corresponding t children of y.
- Line 9 adjusts the key count for y.
- Finally, lines 10-16 insert z as a child of x, move the median key from y up to x in order to separate y from z, and adjust x's key count.
- Lines 17-19 write out all modified disk pages.
- The CPU time used by B-TREE-SPLIT-CHILD is $\Theta(t)$, due to the loops on lines 4-5 and 7-8.
- Splitting a node with $t = 4$.
- Node y is split into two nodes, y and z, and the median key S of y is moved up into y's parent.



Inserting a key into a B-tree

- Inserting a key k into a B-tree T of height h is done in a single pass down the tree, requiring $O(h)$ disk accesses.

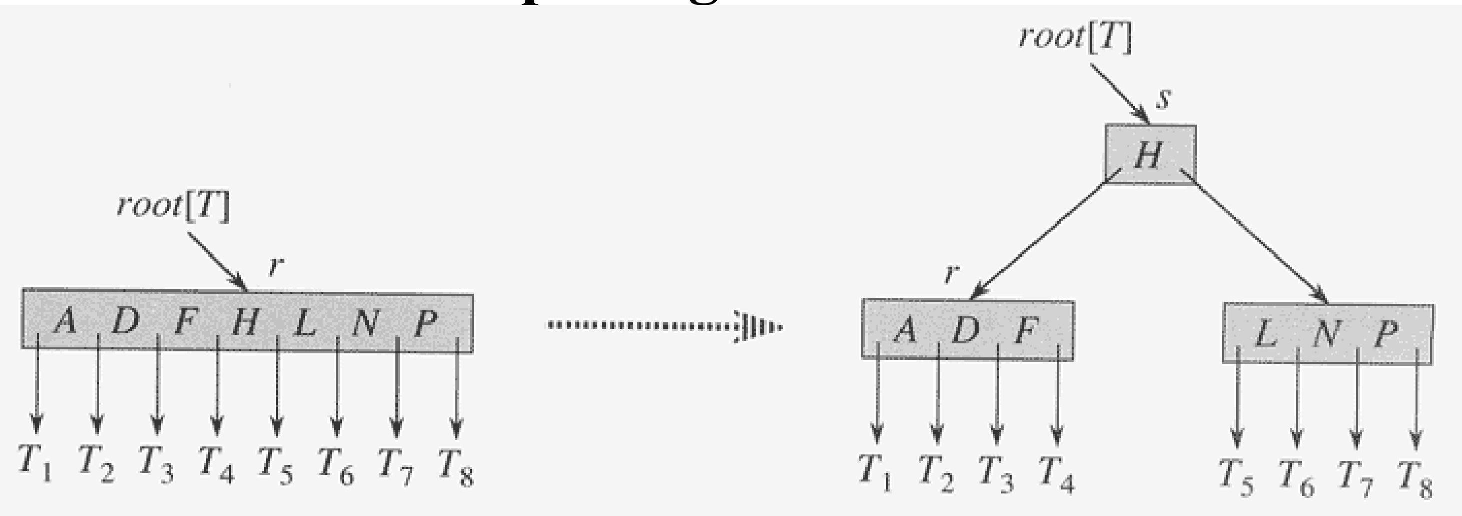
- The CPU time required is $O(th) = O(t \log_t n)$.
- The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.

```
B-TREE-INSERT(T,k)
1 r ← root[T]
2 if n[r] = 2t - 1
3   then s ← ALLOCATE-NODE()
4     root[T]    s
5     leaf[s]    FALSE
6     n[s] ← 0
7     c1[s] ← r
8     B-TREE-SPLIT-CHILD(s,1,r)
9     B-TREE-INSERT-NONFULL(s,k)
10 else B-TREE-INSERT-NONFULL(r,k)
```

Lines 3-9 handle the case in which the root node *r* is full:

- the root is split and a new node *s* (having two children) becomes the root.
- Splitting the root is the only way to increase the height of a B-tree.
- Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom.
- The procedure finishes by calling B-TREE-INSERT-NONFULL to perform the insertion of key *k* in the tree rooted at the nonfull root node.
- B-TREE-INSERT-NONFULL recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling B-TREE-SPLIT-CHILD as necessary.

Splitting the root



- Splitting the root with t = 4.
- Root node *r* is split in two, and a new root node s is created.
- The new root contains the median key of *r* and has the two halves of *r* as children.
- The B-tree grows in height by one when the root is split.

B-TREE-INSERT-NONFULL

The auxiliary recursive procedure B-TREE-INSERT-NONFULL inserts key *k* into node x, which is assumed to be nonfull when the procedure is called.

```
B-TREE-INSERT-NONFULL(x,k)
i ← n[x]
if leaf[x]
        then while i ≥ 1 and k < key_i[x]
                do key_{i+1} [x] ← key_i[x]
                    i ← i - 1
            key_{i+1}[x] ← k
            n[x] ← n[x] + 1
            DISK-WRITE(x)
else while i ≥ 1 and k < key_i[x]
```

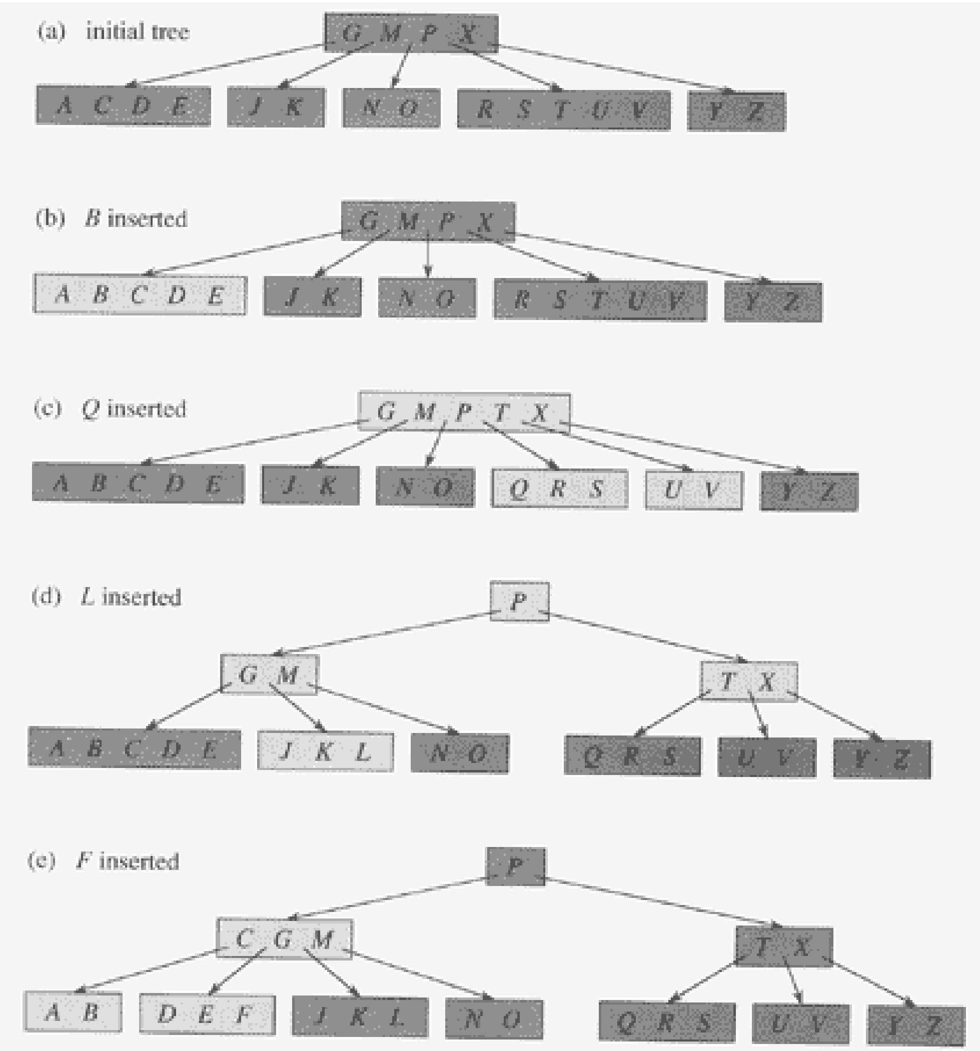
```
10                do i ← i - 1
11        i ← i + 1
12        DISK-READ(ci[x])
13        if n[ci[x]] = 2t - 1
14          then B-TREE-SPLIT-CHILD(x,i,ci[x])
15            if k > keyi[x]
16               then i ← i + 1
17        B-TREE-INSERT-NONFULL(ci[x],k)
```

- The B-TREE-INSERT-NONFULL procedure works as follows.
  - Lines 3-8 handle the case in which *x* is a leaf node by inserting key *k* into *x*.
  - If *x* is not a leaf node, then we must insert *k* into the appropriate leaf node in the subtree rooted at internal node *x*.
  - In this case, lines 9-11 determine the child of *x* to which the recursion descends.
  - Line 13 detects whether the recursion would descend to a full child, in which case line 14 uses B-TREE-SPLIT-CHILD to split that child into two nonfull children, and lines 15-16 determine which of the two children is now the correct one to descend to.
  - The net effect of lines 13-16 is thus to guarantee that the procedure never recurses to a full node.
  - Line 17 then recurses to insert *k* into the appropriate subtree.

Complexity of B-TREE-INSERT

- The number of disk accesses performed by B-TREE-INSERT is $O(h)$ for a B-tree of height h, since only $O(1)$ DISK-READ and DISK-WRITE operations are performed between calls to B-TREE-INSERT-NONFULL.
- The total CPU time used is $O(th) = O(t\log_t n)$.

- Since B-TREE-INSERT-NONFULL is tail-recursive, it can be alternatively implemented as a **while** loop, demonstrating that the number of pages that need to be in main memory at any time is $O(1)$.



a) The initial tree for this example.

(b) The result of inserting B into the initial tree; this is a simple insertion into a leaf node.

(c) The result of inserting Q into the previous tree. The node RSTUV is split into two nodes containing RS and UV, the key T is moved up to the root, and Q is inserted in the leftmost of the two halves (the RS node).

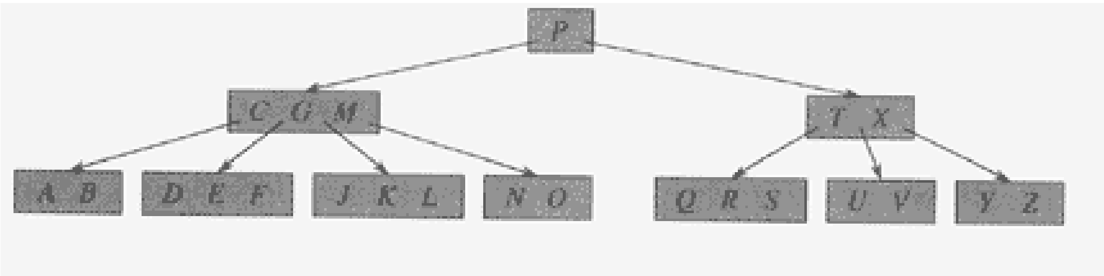
(d) The result of inserting L into the previous tree. The root is split right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK.

(e) The result of inserting F into the previous tree. The node ABCDE is split before F is inserted into the rightmost of the two halves (the DE node).

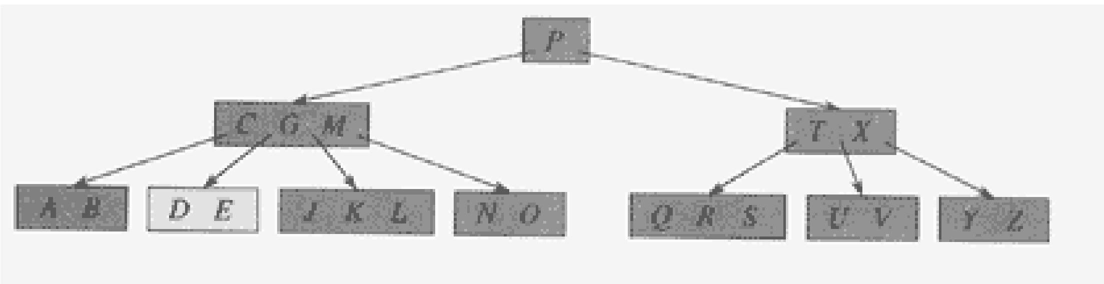
Deleting a key from a B-tree

- Assume that procedure B-TREE-DELETE is asked to delete the key *k* from the subtree rooted at *x*.
  - This procedure is structured to guarantee that whenever B-TREE-DELETE is called recursively on a node *x*, the number of keys in *x* is at least the minimum degree *t*.
  - Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child.
  - This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain).

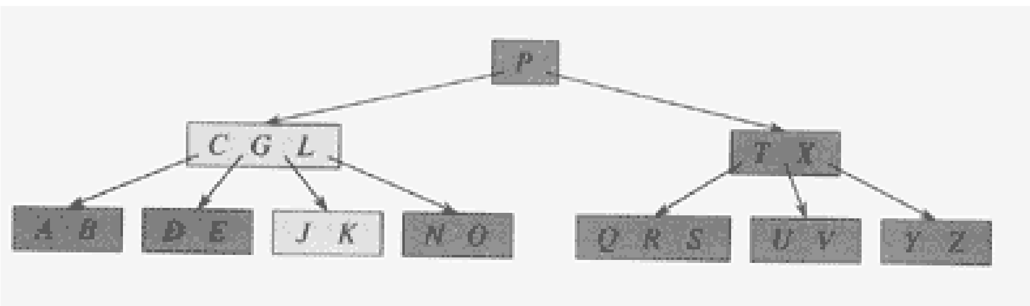
Initial tree



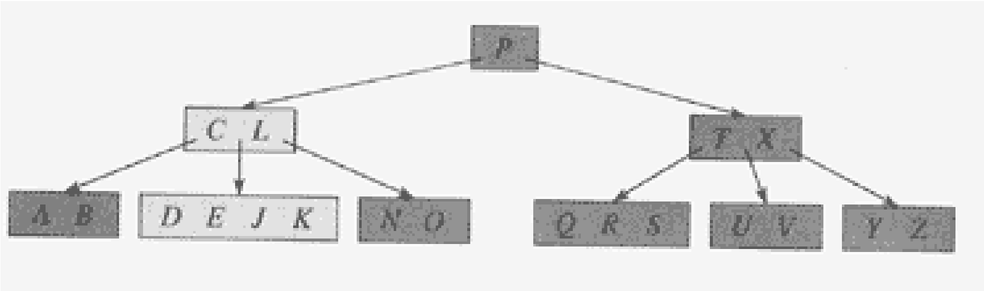
Case 1: If the key k is in node x and x is a leaf, delete the key k from x.
Example: Deletetion of F.



Case 2: If the key *k* is in node *x* and *x* is an internal node, do the following.



Deletion of M. The predecessor L of M is moved up to take M's position.



Deletion of G. G is pushed down to make node DEGJK, and then G is deleted from this leaf

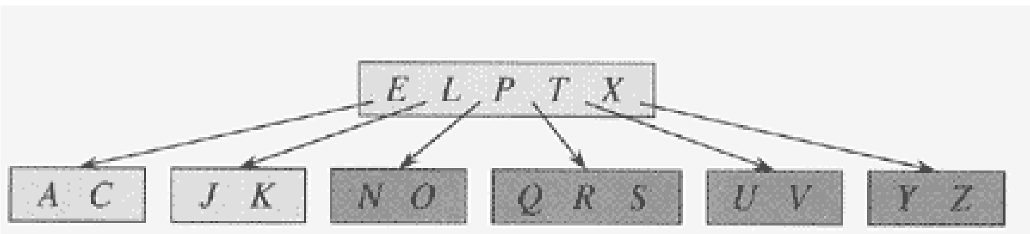
If the child *y* that precedes *k* in node *x* has at least *t* keys, then find the predecessor *k*' of *k* in the subtree rooted at *y*. Recursively delete *k*', and replace *k* by *k*' in *x*.

Symmetrically, if the child *z* that follows *k* in node *x* has at least *t* keys, then find the successor *k*' of *k* in the subtree rooted at *z*. Recursively delete *k*', and replace *k* by *k*' in *x*.

Otherwise, if both *y* and *z* have only *t*- 1 keys, merge *k* and all of *z* into *y*, so that *x* loses both *k* and the pointer to *z*, and *y* now contains 2*t* - 1 keys. Then, free *z* and recursively delete *k* from *y*.

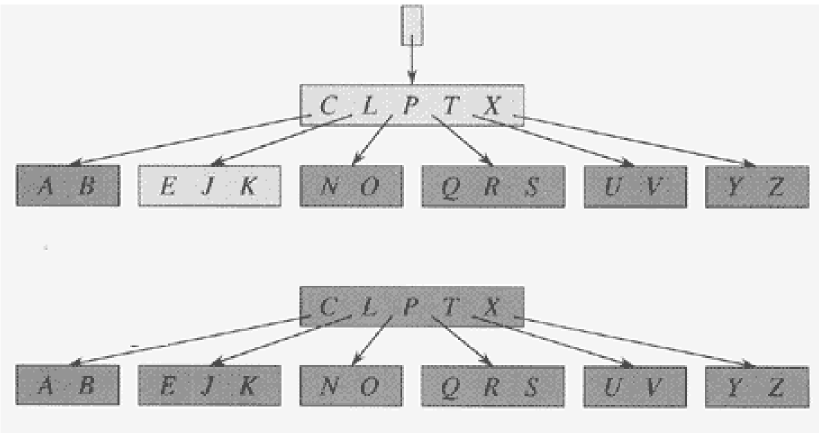
- Case 3: If the key *k* is not present in internal node *x*, determine the root $c_i[x]$ of the appropriate subtree that must contain *k*, if *k* is in the tree at all.
- If $c_i[x]$ has only *t* - 1 keys, execute step **3a** or **3b** as necessary to guarantee that we descend to a node containing at least *t* keys.
- Then, finish by recursing on the appropriate child of *x*.

a. If $c_i[x]$ has only *t* - 1 keys but has a sibling with *t* keys, give $c_i[x]$ an extra key by moving a key from *x* down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into *x*, and moving the appropriate child from the sibling into $c_i[x]$.



Deletion of B. C is moved to fill B's position and E is moved to fill C's position.

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



Deletion of D. The recursion can't descend to node CL because it has only 2 keys, so P is pushed down and merged with CL and TX to form CLPTX; then, D is deleted from a leaf After this the root is deleted and the tree shrinks in height by one.