



Theory of algorithms (7th lecture)

Pál Pusztai
pusztai@sze.hu

Outline

- B-trees
 - Definition
 - Basic operations
- Exercises



B-trees

- B-trees are balanced search trees designed to work well on disks or other direct-access secondary storage devices.
- B-trees are similar to red-black trees, but they are better at minimizing disk I/O operations. Many database systems use B-trees, or variants of B-trees, to store information.
- B-tree nodes may have many children, from a few to thousands. That is, the “branching factor” of a B-tree can be quite large, although it usually depends on characteristics of the disk unit used.
- Every n -node B-tree has height $O(\lg n)$, so dynamic-set operations that use B-trees run in $O(\lg n)$ time too.
- In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The B-tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed.



B-trees

The typical pattern for working with an object is as follows:

- 1 x = a pointer to some object
- 2 DISK-READ (x) // if x is already in main memory it is a “no-op” (no operation).
- 3 operations that access and/or modify the attributes of x
- 4 DISK-WRITE(x) // omitted if no attributes of x were changed
- 5 other operations that access but do not modify attributes of x

Remarks:

- A B-tree node is usually as large as a whole disk page, and this size limits the number of children a B-tree node can have.
- We shall look separately at the two principal components of the running time:
 - the number of disk accesses, and
 - the CPU (computing) time.



B-trees

A **B-tree** T is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node x has the following attributes:
 - $x.n$, the number of keys currently stored in node x ,
 - the $x.n$ keys themselves, $x.key_1, x.key_2, \dots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \dots \leq x.key_{x.n}$,
 - $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $x.n+1$ pointers $x.c_1, x.c_2, \dots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \dots \leq x.key_{x.n} \leq k_{x.n+1}$.
4. All leaves have the same depth, which is the tree's height h .
5. Nodes have lower and upper bounds on the number of keys they can contain. These bounds are expressed with a fixed integer $t \geq 2$ called the **minimum degree** of the B-tree:
 - 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.
 - Every node may contain at most $2t-1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is **full** if it contains exactly $2t-1$ keys.



B-trees

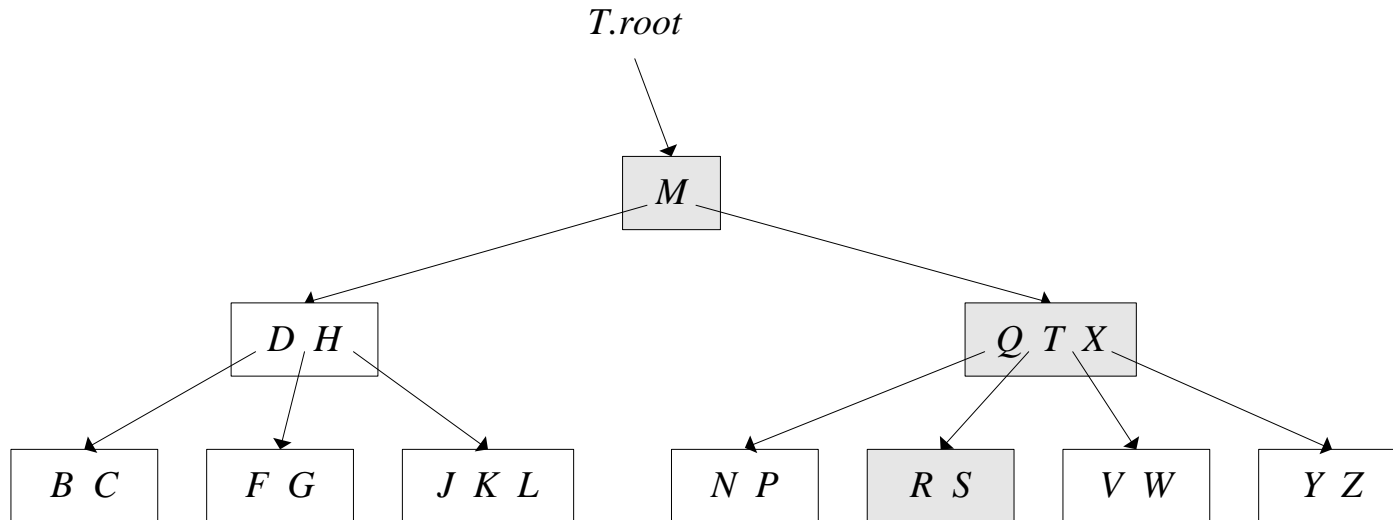
Remarks: The simplest B-tree occurs when $t=2$. Every internal node has either 2, 3, or 4 children so its name is **2-3-4 tree**. In practice much larger values of t is used because of the smaller height.

Theorem: If $n \geq 1$, then for any n -key B-tree T of height h and minimum degree $t \geq 2$, $h \leq \log_t ((n+1)/2)$.

Corollary: B-trees save a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations. Because we usually have to access the disk to examine an arbitrary node in a tree, B-trees avoid a substantial number of disk accesses.



B-trees



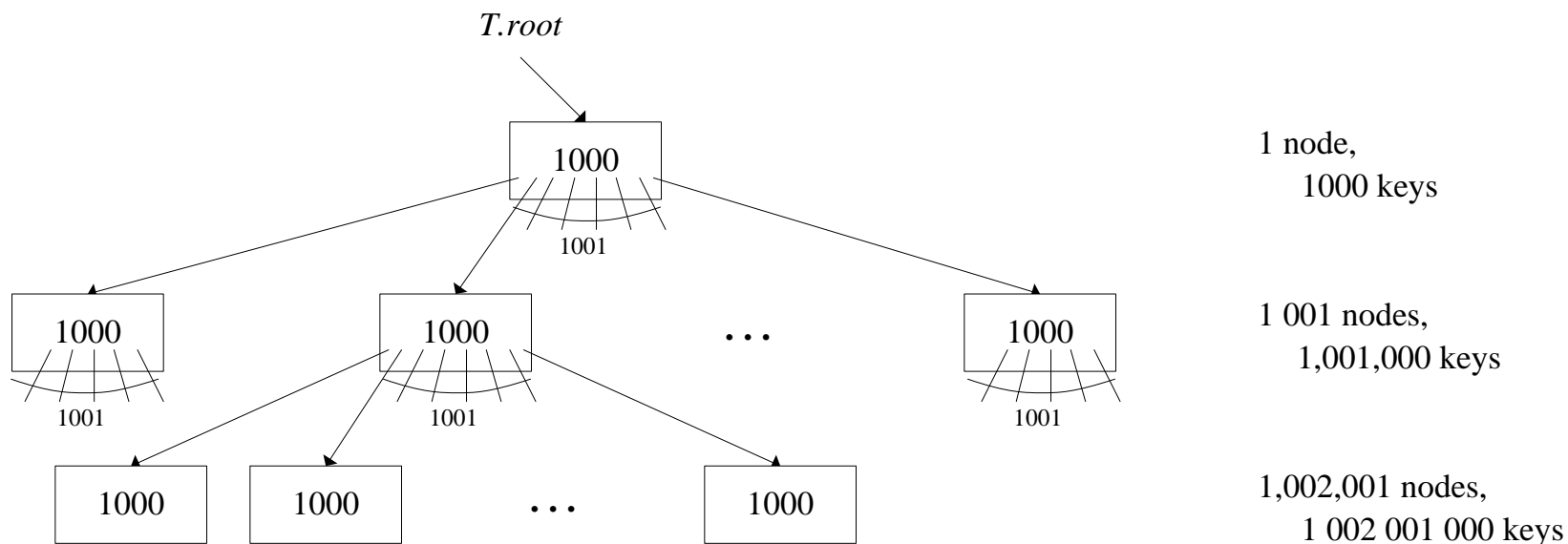
A B-tree

Exercises

- For what values of t is the tree on the previous slide a legal B-tree?
- Why the minimum degree can not be 1?
- Give all legal B-trees of minimum degree 2 that represent $\{1, 2, 3, 4, 5\}$ keys.
- What does the B-tree of height 3 look like that contains minimum number of keys? The height of root node is 0.

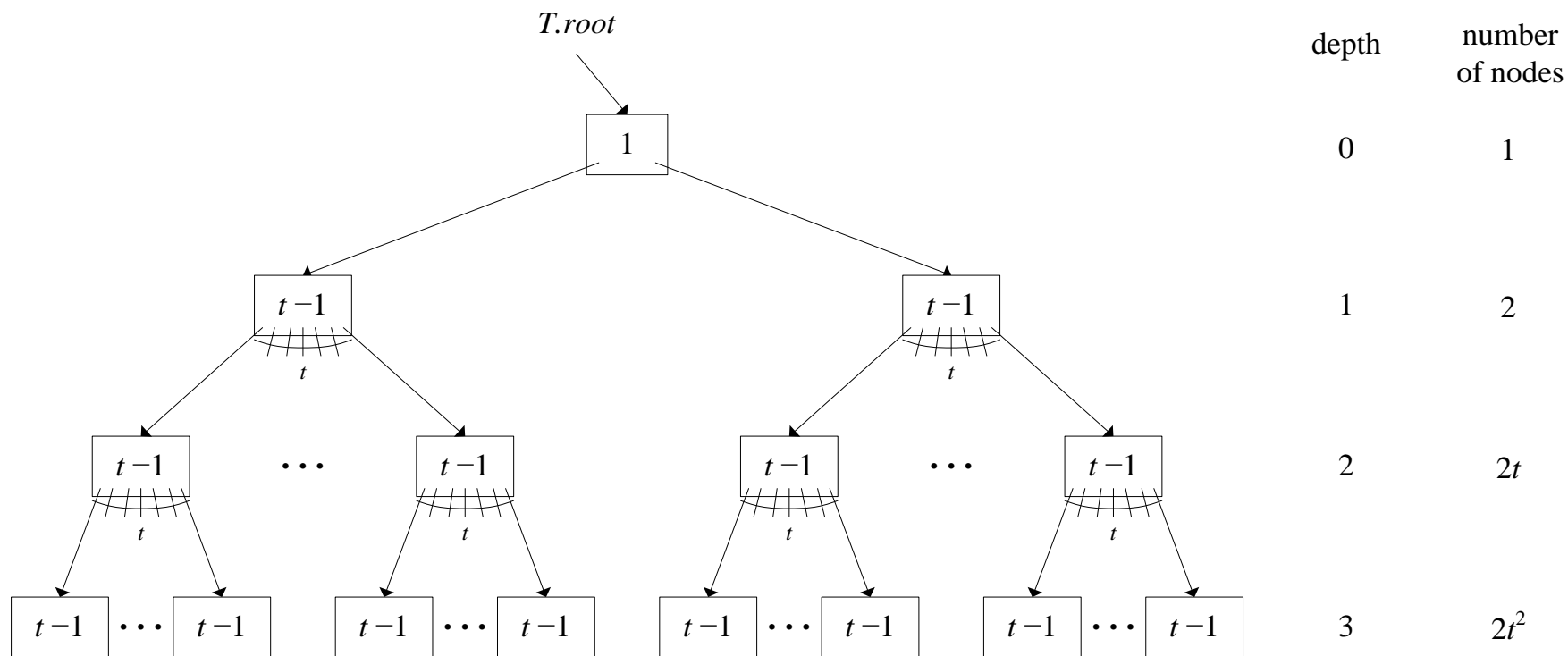


B-trees



A B-tree of height 2 containing over one billion keys

B-trees



A B-tree of height 3 containing a minimum possible number of keys

Basic operations

In the basic operations of B-tree (B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT) it is assumed that:

- The root of the B-tree is always in main memory, so that we never need to perform a DISK-READ on the root, however a DISK-WRITE of the root has to be performed 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 auxiliary procedure ALLOCATE-NODE allocates one disk page to be used as a new node in $O(1)$ time. It requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.



Basic operations

B-TREE-SEARCH(x, k)

```
1   $i = 1$ 
2  while  $i \leq x.n$  and  $k > x.key_i$ 
3       $i = i + 1$ 
4  if  $i \leq x.n$  and  $k = x.key_i$ 
5      return ( $x, i$ )
6  if  $x.leaf$ 
7      return NIL
8  else
9      DISK-READ( $x.c_i$ )
10     return B-TREE-SEARCH( $x.c_i, k$ )
```

Efficiency: It requires $O(h)$ disk operations and $O(th)$ CPU time, where $h = \log_t n$ is the height of the B-tree and n is the number of keys in the B-tree.



Exercises

- How can the minimum key be searched in a B-tree?



Basic operations

B-TREE-CREATE(T)

- 1 $x = \text{ALLOCATE-NODE}()$
- 2 $x.\text{leaf} = \text{TRUE}$
- 3 $x.n = 0$
- 4 $\text{DISK-WRITE}(x)$
- 5 $T.\text{root} = x$

Efficiency: It requires $O(1)$ disk operations and $O(1)$ CPU time.



Basic operations

B-TREE-SPLIT-CHILD(x, i)

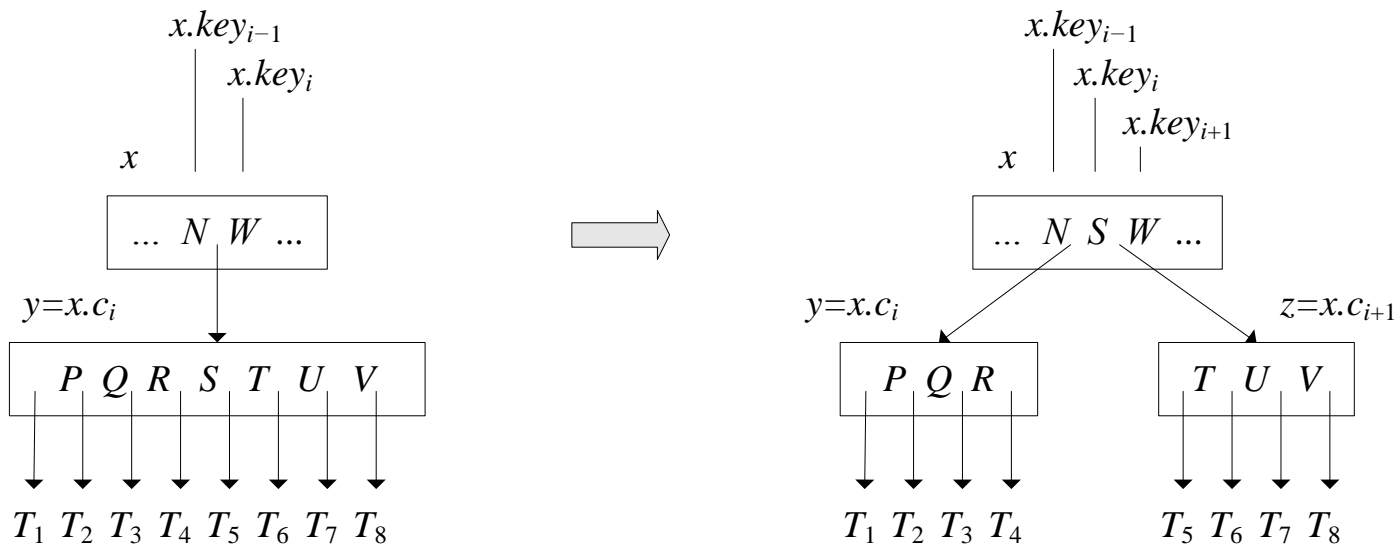
```
1   $z = \text{ALLOCATE-NODE}()$ 
2   $y = x.c_i$ 
3   $z.\text{leaf} = y.\text{leaf}$ 
4   $z.n = t - 1$ 
5  for  $j = 1$  to  $t - 1$ 
6       $z.\text{key}_j = y.\text{key}_{j+t}$ 
7  if not  $y.\text{leaf}$ 
8      for  $j = 1$  to  $t$ 
9           $z.c_j = y.c_{j+t}$ 
10  $y.n = t - 1$ 
11 for  $j = x.n + 1$  downto  $i + 1$ 
12      $x.c_{j+1} = x.c_j$ 
13  $x.c_{i+1} = z$ 
14 for  $j = x.n$  downto  $i$ 
15      $x.\text{key}_{j+1} = x.\text{key}_j$ 
16  $x.\text{key}_i = y.\text{key}_t$ 
17  $x.n = x.n + 1$ 
18 DISK-WRITE( $y$ )
19 DISK-WRITE ( $z$ )
20 DISK-WRITE ( $x$ )
```

Efficiency: It requires $O(1)$ disk operations and $\Theta(t)$ CPU time.

Criteria: The internal node x is not full and node y (the i th child of node x) is full.



Basic operations



Splitting a node ($t = 4$)

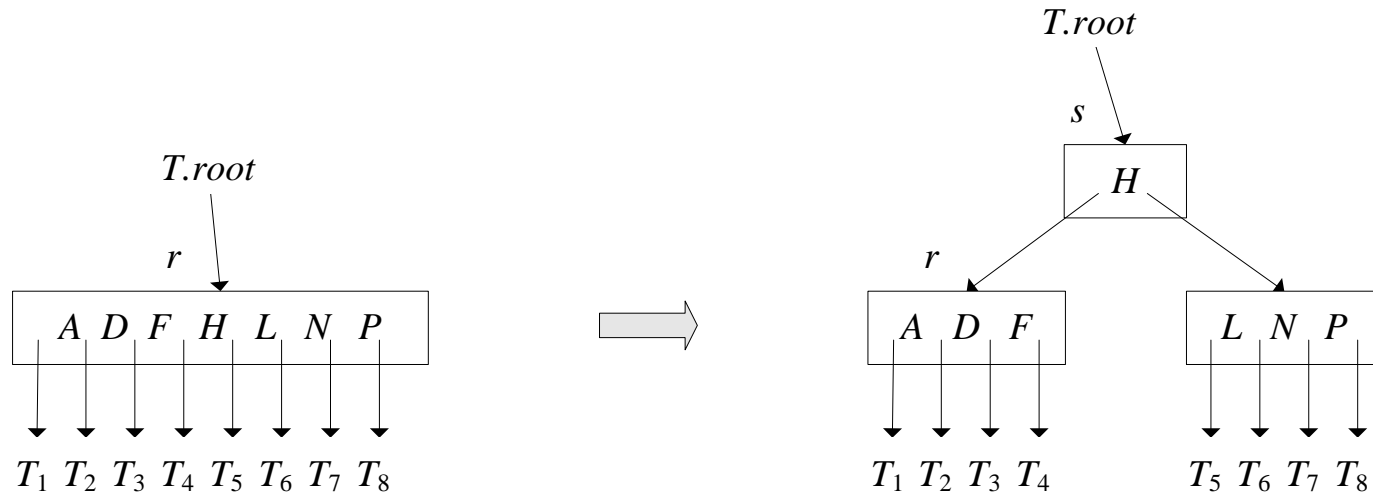
Basic operations

B-TREE-INSERT(T, k)

```
1   $r = T.root$ 
2  if  $r.n == 2t - 1$ 
3       $s = \text{ALLOCATE-NODE}()$ 
4       $T.root = s$ 
5       $s.leaf = \text{FALSE}$ 
6       $s.n = 0$ 
7       $s.c_1 = r$ 
8      B-TREE-SPLIT-CHILD( $s, 1$ )
9      B-TREE-INSERT-NONFULL( $s, k$ )
10 else
11     B-TREE-INSERT-NONFULL( $r, k$ )
```

Efficiency: They are given by the called procedures (B-TREE-SPLIT-CHILD, B-TREE-INSERT-NONFULL) because they are increased only with $O(1)$ disk operations and $O(1)$ CPU time.

Basic operations



Splitting the root ($t = 4$)

Basic operations

B-TREE-INSERT-NONFULL(x, k)

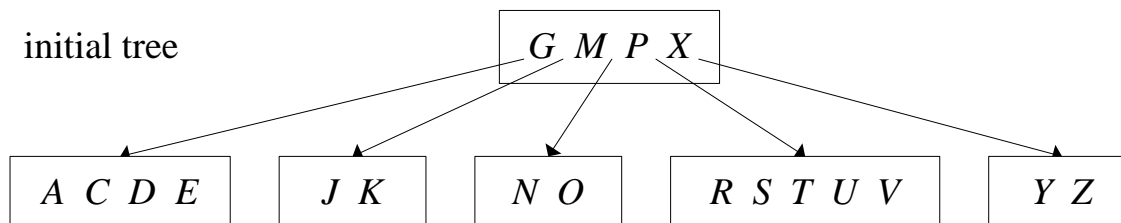
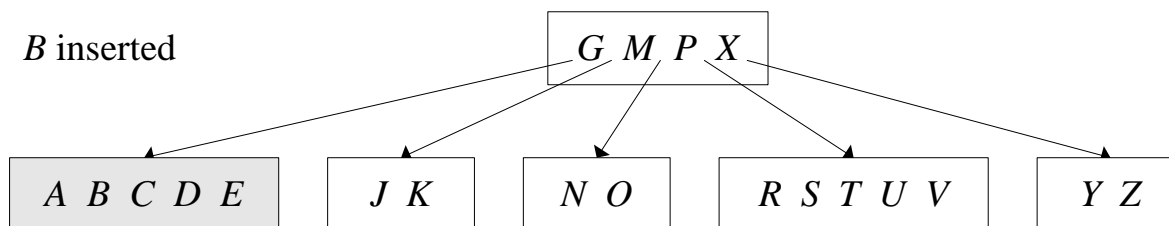
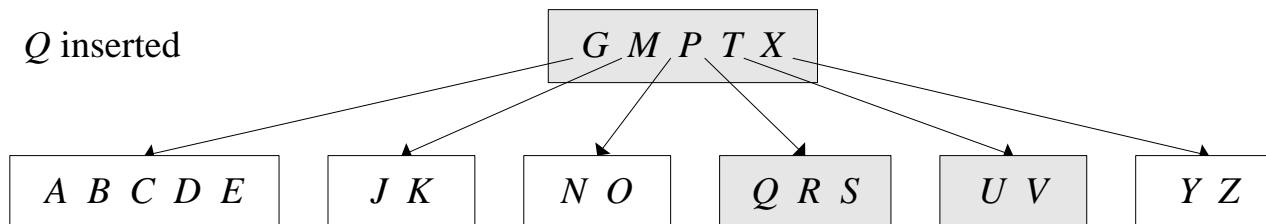
```
1   $i = x.n$ 
2  if  $x.leaf$ 
3      while  $i \geq 1$  and  $k < x.key_i$ 
4           $x.key_{i+1} = x.key_i$ 
5           $i = i - 1$ 
6       $x.key_{i+1} = k$ 
7       $x.n = x.n + 1$ 
8      DISK-WRITE( $x$ )
9  else
10     while  $i \geq 1$  and  $k < x.key_i$ 
11          $i = i - 1$ 
12      $i = i + 1$ 
13     DISK-READ( $x.c_i$ )
14     if  $x.c_i.n == 2t - 1$ 
15         B-TREE-SPLIT-CHILD( $x, i$ )
16         if  $k > x.key_i$ 
17              $i = i + 1$ 
18     B-TREE-INSERT-NONFULL( $x.c_i, k$ )
```

Efficiency: It requires $O(h)$ disk operations and $O(th)$ CPU time.

Criteria: The node x is not full.

Basic operations

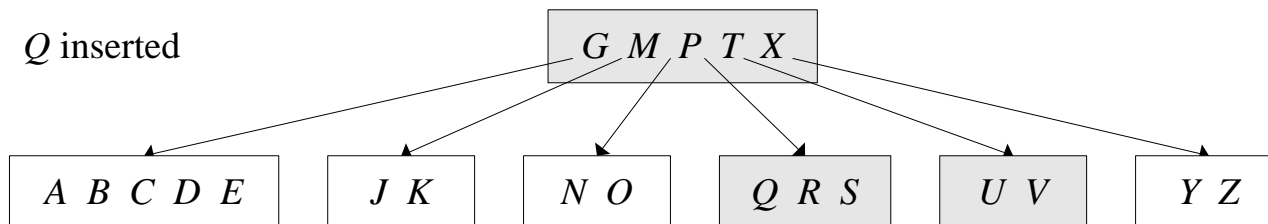
initial tree

 B inserted Q inserted

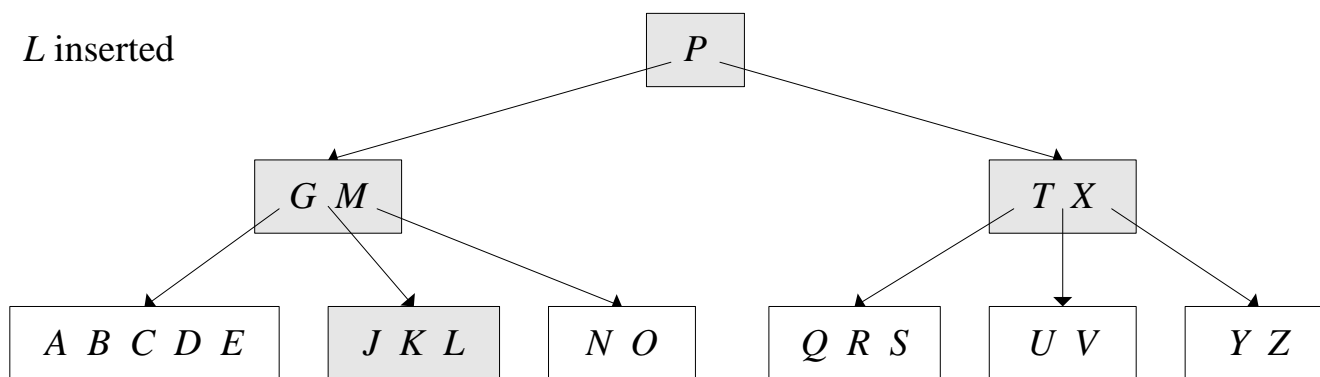
Inserting keys into a B-tree ($t = 3$)

Basic operations

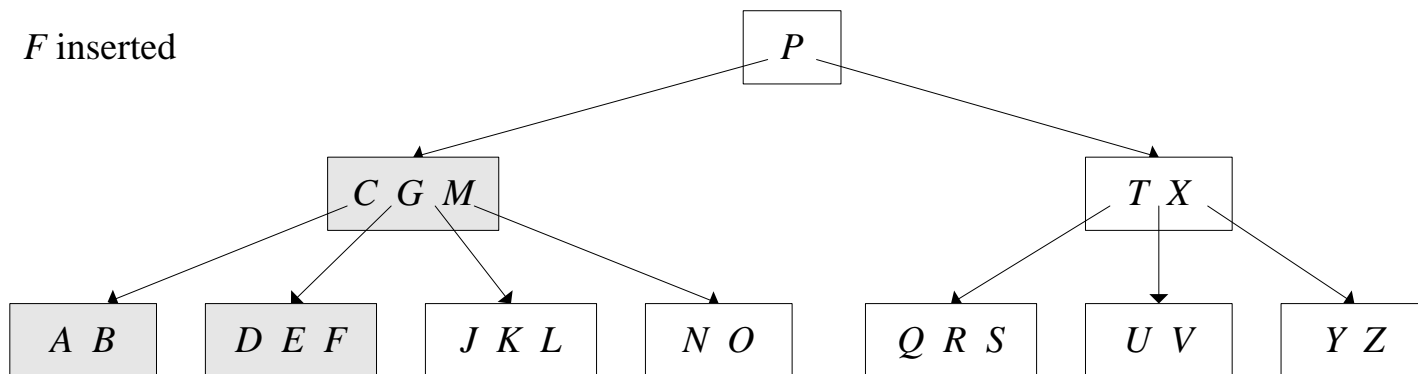
Q inserted



L inserted



F inserted



Inserting keys into a B-tree ($t = 3$)



Exercises

- What B-tree is built with inserting below given keys into an initially empty tree? The minimum degree $t=3$.
 - $F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$
- What is the result if the minimum degree $t=2$?

