

Analysis of Algorithms (Fall 2013) Istanbul Technical University Computer Eng. Dept.

Chapter 18 B-Trees



Last updated: December 16, 2009

Purpose

- Understand B-tree properties and why B-trees are important
- Understand search, insert, and delete operations on B-trees
- Learn B+ and B* tree definitions

Outline

- B-Tree
 - B-Tree Properties
 - B-Tree Search, Insert, Delete
- B* Tree
- B+ Tree

B-Trees

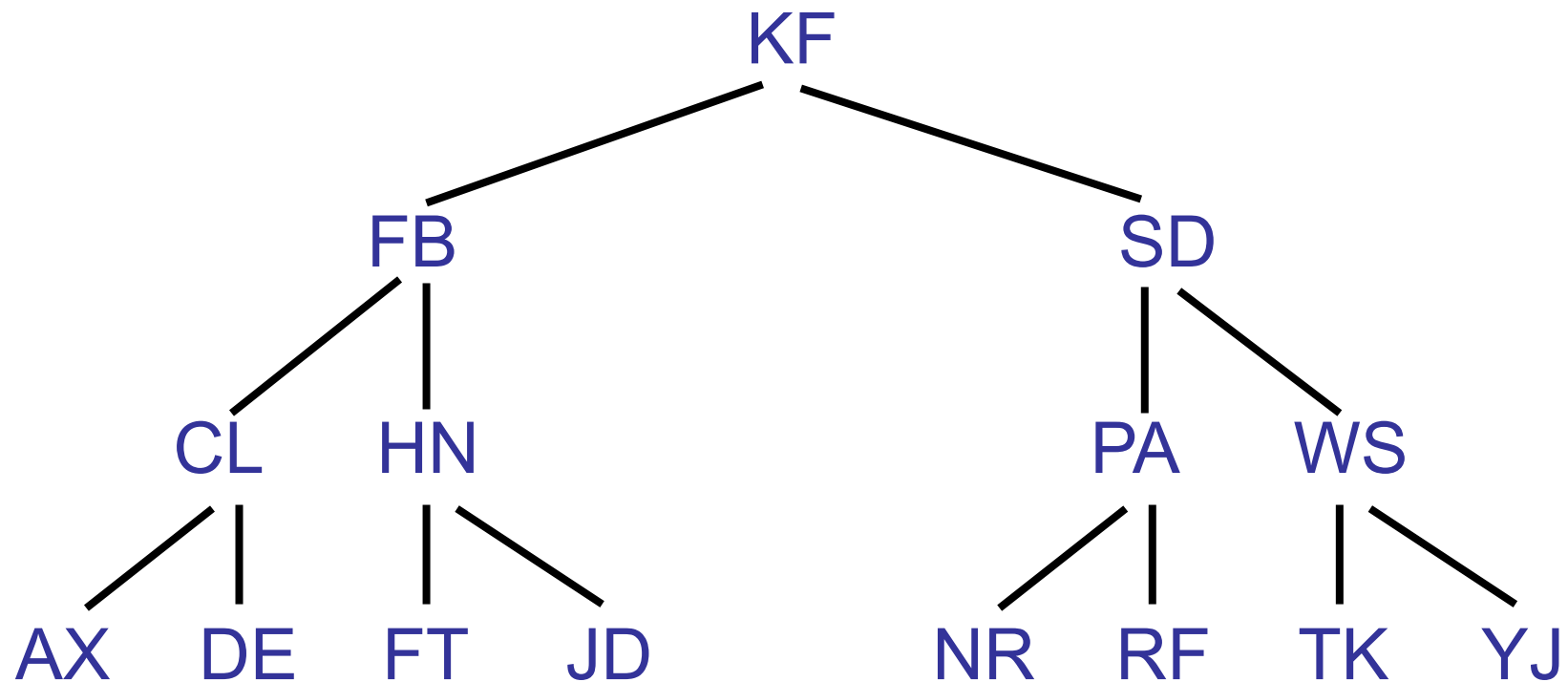
- Relatively new research area
- 1972: R. Bayer and E. McCreight (Boeing Corp.), “Organization and Maintenance of Large Ordered Indexes” first introduced B-trees
- 1979: B-trees had become “the standard organization for indexes in a database system” (D. Comer, “The Ubiquitous B-Tree”)
- Also new publications, e.g.:
 - “A practical scalable distributed B-tree,” M.K. Aguilera, W. Golab, and M.A. Shah, Proc. of VLDB Endowment, 2008

Statement of the Problem

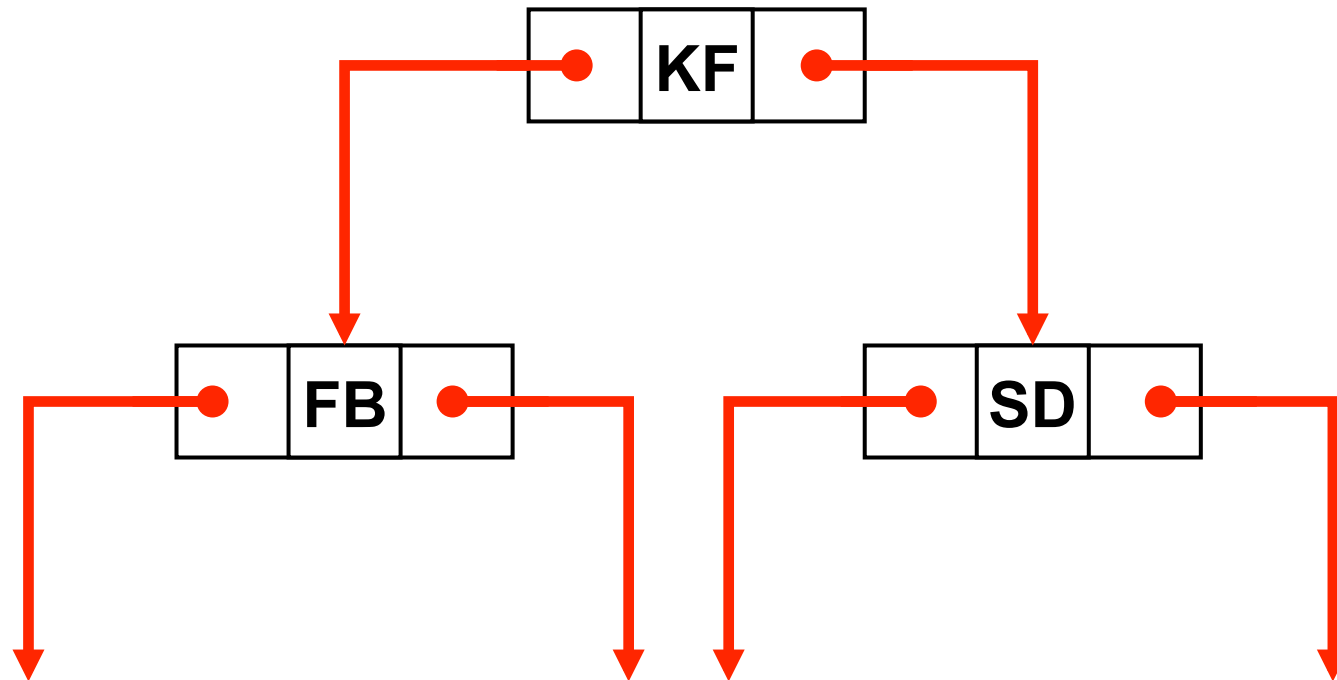
- Fundamental Problem
 - Secondary storage is slow
 - Keeping an index on secondary storage is slow, too!
- Approaches
 - Faster index searching (than binary search)
 - Fast insertion and deletion

Indexing with Binary Search Trees

AX CL DE FB FT HN JD KF NR PA RF SD TK WS YJ

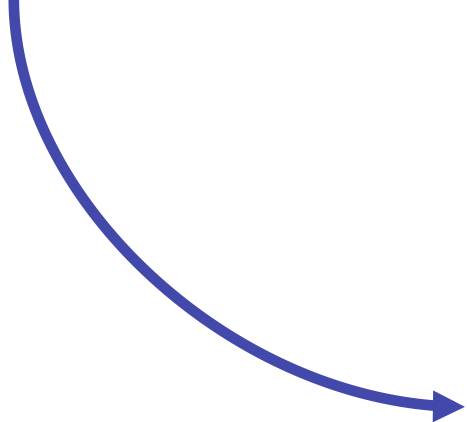


Linked List Representation of a BST



Array Representation of BST

ROOT



0	FB	10	8
1	JD		
2	RF		
3	SD	6	13
4	AX		
5	YJ		
6	PA	11	2
7	FT		
8	HN	7	1
9	KF	0	3
10	CL	4	12
11	NR		
12	DE		
13	WS	14	5
14	TK		

B-Trees

- Are balanced search trees designed to work well on magnetic disks or other direct-access
- Are similar to red-black trees but they are better at minimizing disk I/O operations
- Have height $O(\log n)$
- Can also be used to implement many dynamic-set operations in time $O(\log n)$

B-Trees

- B-Tree algorithms copy selected pages from disk into main memory as needed and write back onto disk the pages that have changed
- **Example:** B-Tree with a branching factor of 1001 and height 2
 - can store over one billion keys
 - only two disk accesses at most are required to find any key

B-Tree

B-tree, T , is a rooted tree (whose root is $\text{root}[T]$) having the following properties:

1. Every node x has the following properties
 - $n[x]$, number of keys currently stored in node x
 - $n[x]$ keys themselves in non decreasing order, so that $\text{key}_1[x] \leq \text{key}_2[x] \leq \dots \leq \text{key}_{n[x]}[x]$
 - $\text{leaf}[x]$, 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 $n[x]+1$ pointers $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$ to its children
 - Leaves have no children, hence their c_i are undefined

B-Tree (continued)

B-tree, T , is a rooted tree (whose root is $\text{root}[T]$) having the following properties:

3. Keys $\text{key}_i[x]$ separate ranges of keys stored in each subtree:

- if k_i is any key stored in subtree with root $c_i[x]$, then

$$k_1 \leq \text{key}_1[x] \leq k_2 \leq \text{key}_2[x] \leq \dots \leq \text{key}_{n[x]}[x] \leq k_{n[x]+1}$$

3. All leaves have same depth, which is tree's height h

B-Tree (continued)

B-tree, T , is a rooted tree (whose root is $\text{root}[T]$) having the following properties:

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
 - 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 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

B-Tree

- 2-3-4 tree is the simplest B-tree with $t = 2$
- Typically much larger values of t are used

Theorem:

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

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

High Capacity of B-Trees

- B-tree of height 2 can contain over 1 billion keys when each internal node and leaf contains 1000 keys!
- depth 0: 1 node 1,000 keys
- depth 1: 1001 nodes, 1,001,000 keys
- depth 2: 1,002,001 nodes, 1,002,001,000 keys

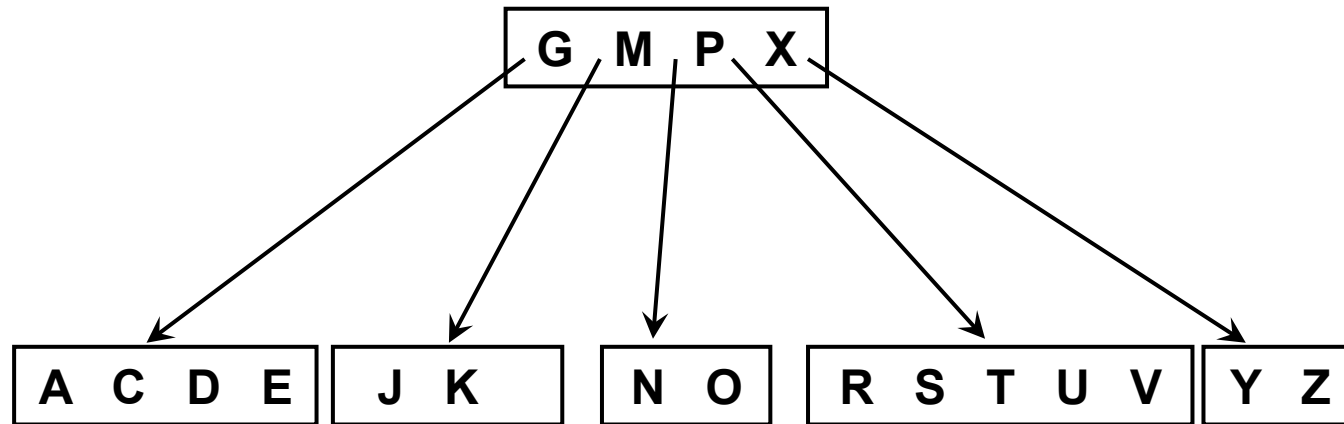
Why $t-1$ to $2t-1$ Keys?

As opposed to t keys at every node for example?

Answer: Because we can keep at least $t-1$ and at most $2t-1$ keys in a node, we

- do not have to increase the tree height right after every insertion and
- need to increase height only after inserting t more keys

Example of a B-Tree



- Minimum degree $t = 3$
- Check:
 - Max: $2t - 1 = 5$
 - Min: $t - 1 = 2$

Searching

▷ Search for key k at node x

B-TREE-SEARCH(x, k)

1 $i \leftarrow 1$

2 **while** $i \leq n[x]$ and $k > \text{key}_i[x]$

3 **do** $i \leftarrow i+1$

4 **if** $i \leq n[x]$ and $k = \text{key}_i[x]$

5 **then return** (x, i)

6 **if** $\text{leaf}[x]$

7 **then return** NIL

8 **else** DISK-READ($c_i[x]$)

9 **return** B-TREE-SEARCH($c_i[x], k$)

Searching

- Number of disk accesses is:
 $\Theta(h) = \Theta(\log_t n)$
where h : tree height
 n : number of keys in B tree
- Total CPU time:
 $O(th) = O(t \log_t n)$
- Could do better with binary search:
 $O(\log_2 t \log_t n)$

Inserting

- More complicated than inserting a key into a binary search tree
- Need to insert at a leaf node
- Can not insert into a full leaf node, hence need to split around the median if the leaf node is full

Inserting (2)

► y is the i th child of x and is the node being split

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

```
1  z ← Allocate_Node
2  leaf[z] ← leaf[y]
3  n[z] ← t - 1
4  for j ← 1 to t - 1
5      do keyj[z] ← keyj+t[y]
6  if not leaf[y]
7      then for j ← 1 to t
8          do cj[z] ← cj+t[y]
9  n[y] ← t-1
10 for j ← n[x]+1 downto i+1
11     do cj+1[x] ← cj[x]
12 ci+1[x] ← z
13 for j ← n[x] downto i
14     do keyj+1[x] ← keyj[x]
15 keyi[x] ← keyt[y]
16 n[x] ← n[x]+1
17 DISK-WRITE(y) ; DISK-WRITE(z) ; DISK-WRITE(x)
```

Inserting (3)

B-TREE-INSERT(T, k)

```
1   $r \leftarrow \text{root}[T]$ 
2  if  $n[r] = 2t - 1$   $\triangleright$  if root is full, add a new layer
3      then  $s \leftarrow \text{ALLOCATE-NODE}()$ 
4           $\text{root}[T] \leftarrow s$ 
5           $\text{leaf}[s] \leftarrow \text{FALSE}$ 
6           $n[s] \leftarrow 0$ 
7           $c_1[s] \leftarrow r$ 
8          B-TREE-SPLIT-CHILD( $s, 1, r$ )
9          B-TREE-INSERT-NONFULL( $s, k$ )
10 else B-Tree-INSERT-NONFULL( $r, k$ )
```

Inserting (4)

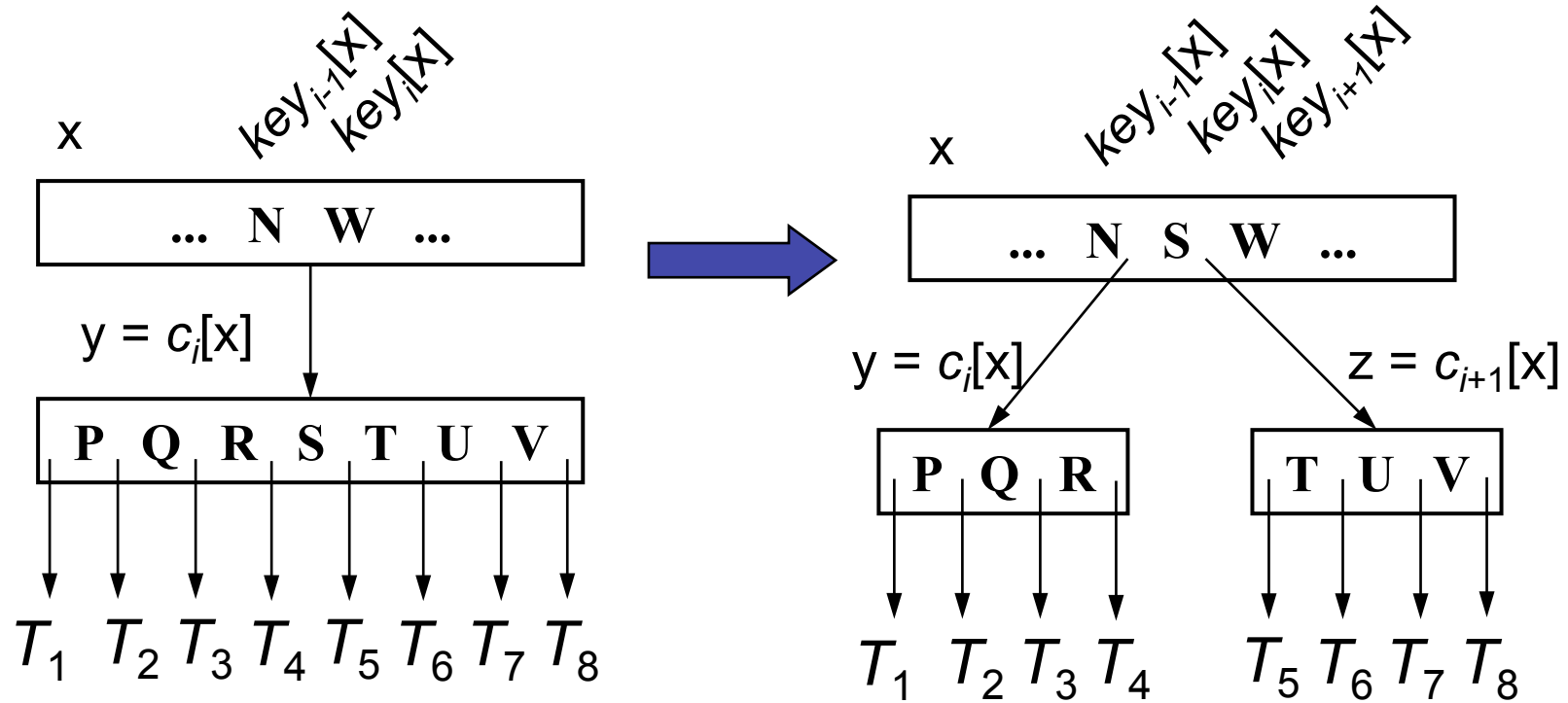
B-TREE-INSERT-NONFULL(x, k)

```
1   $i \leftarrow n[x]$ 
2  if leaf[ $x$ ]
3      then while  $i \geq 1$  and  $k < \text{key}_i[x]$ 
4          do  $\text{key}_{i+1}[x] \leftarrow \text{key}_i[x]$ 
5               $i \leftarrow i - 1$ 
6           $\text{key}_{i+1}[x] \leftarrow k$ 
7           $n[x] \leftarrow n[x] + 1$ 
8          DISK-WRITE( $x$ )
9      else while  $i \geq 1$  and  $k < \text{key}_i[x]$ 
10         do  $i \leftarrow i - 1$ 
11          $i \leftarrow i + 1$ 
12         DISK-READ( $c_i[x]$ )
13         if  $n[c_i[x]] = 2t - 1$ 
14             then B-TREE-SPLIT-CHILD( $x, i, c_i[x]$ )
15             if  $k > \text{key}_i[x]$ 
16                 then  $i \leftarrow i + 1$ 
17         B-TREE-INSERT-NONFULL( $c_i[x], k$ )
```

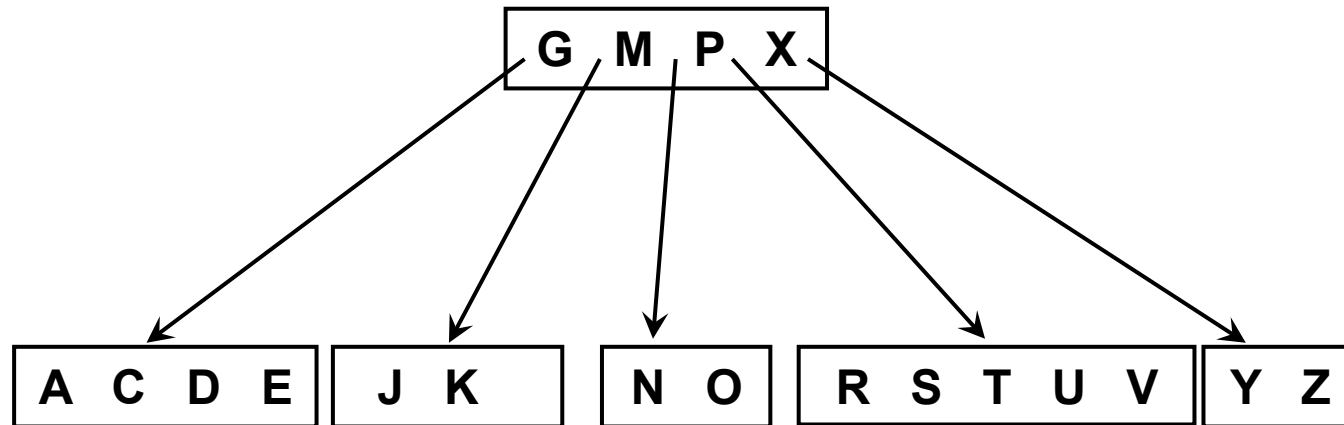
Inserting and Splitting

- Insert into a node to its limit, i.e., $2t - 1$ keys
- A node with $2t - 1$ keys, i.e. $2t$ children has to split
- After a split
 - One key (median of $2t - 1$ keys) moves up
 - Two new nodes with $t - 1$ keys
 - New item inserted into the appropriate node

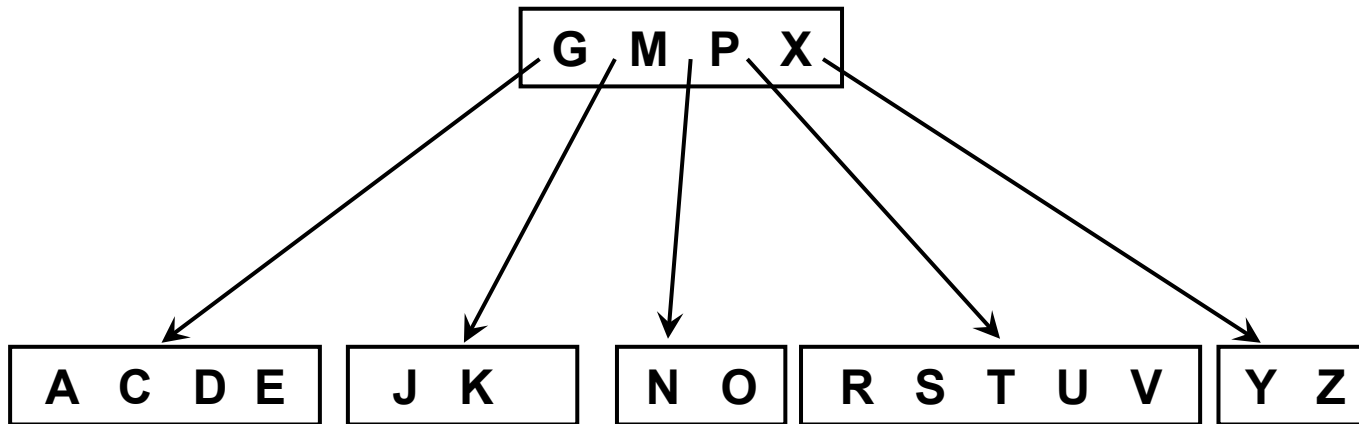
Split



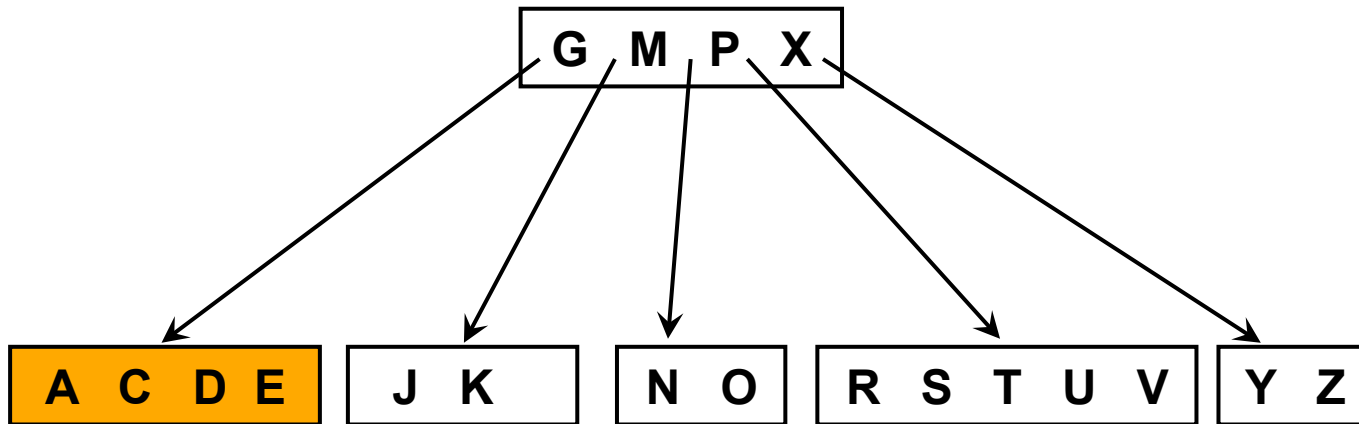
Initial Tree



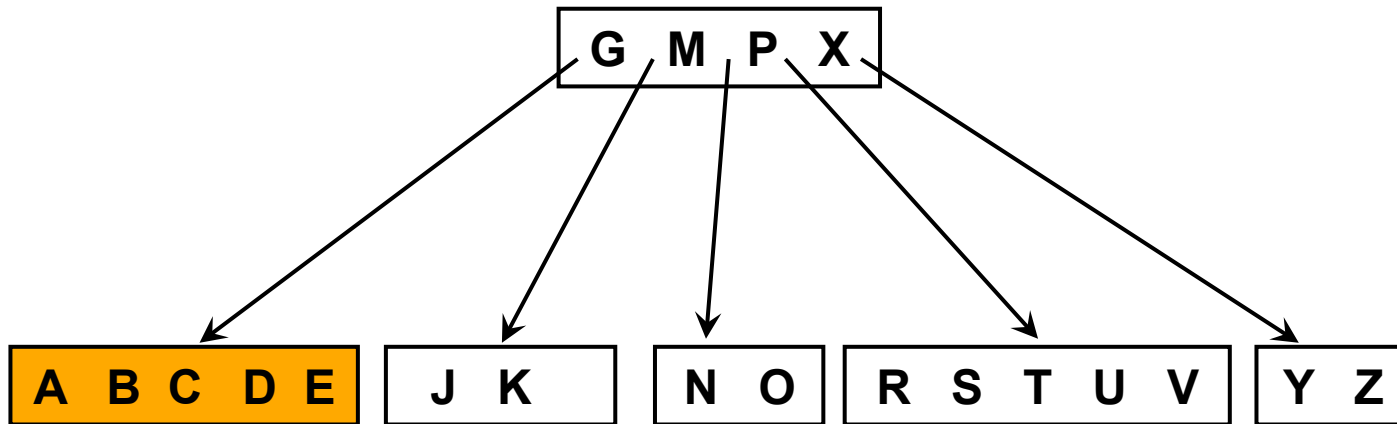
Inserting B



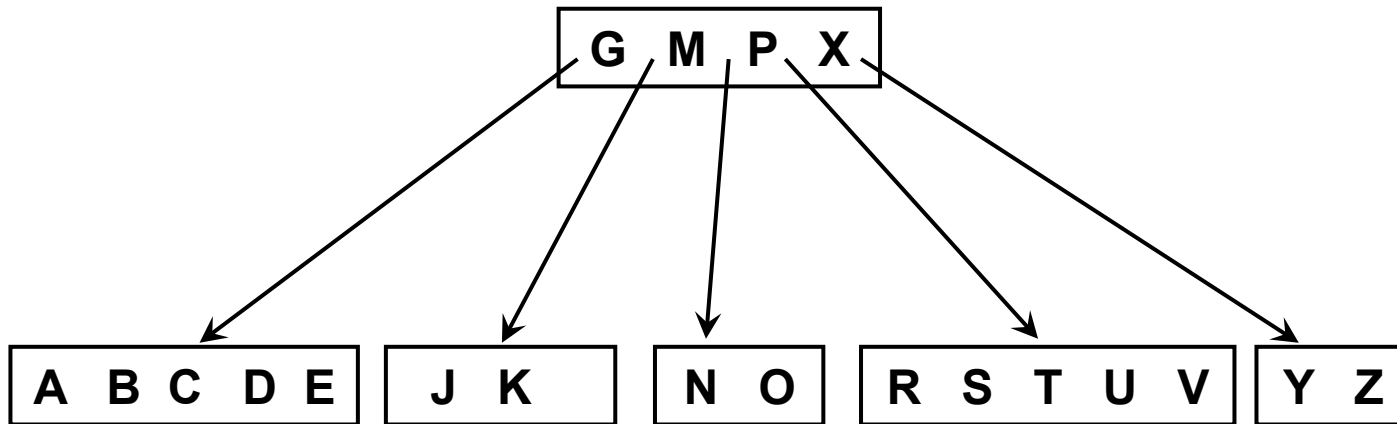
Inserting B



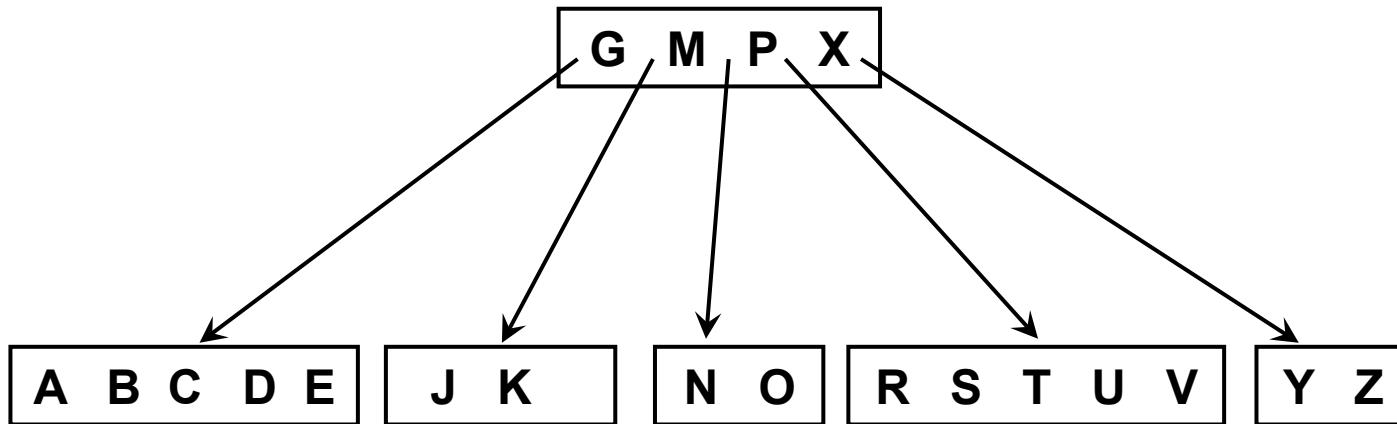
Inserting B



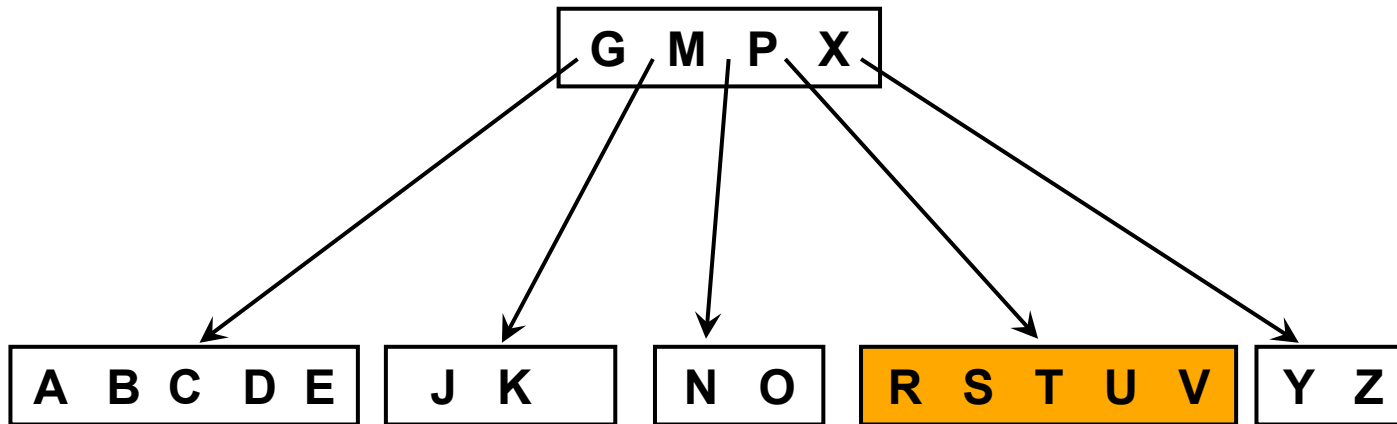
Inserting B



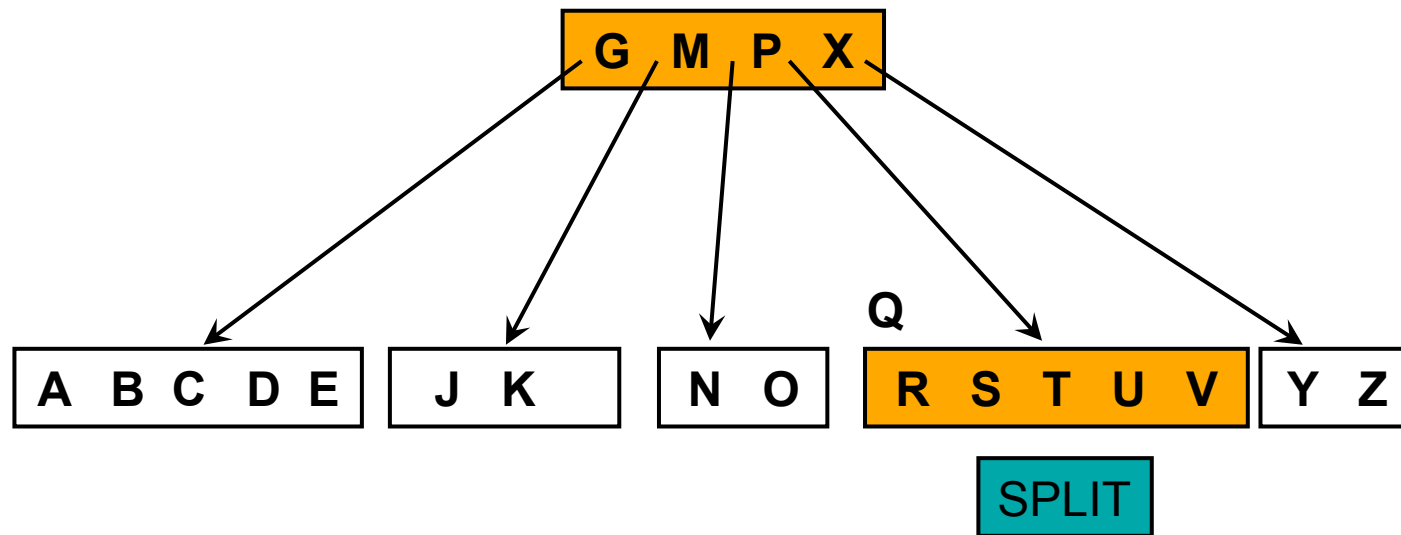
Inserting Q



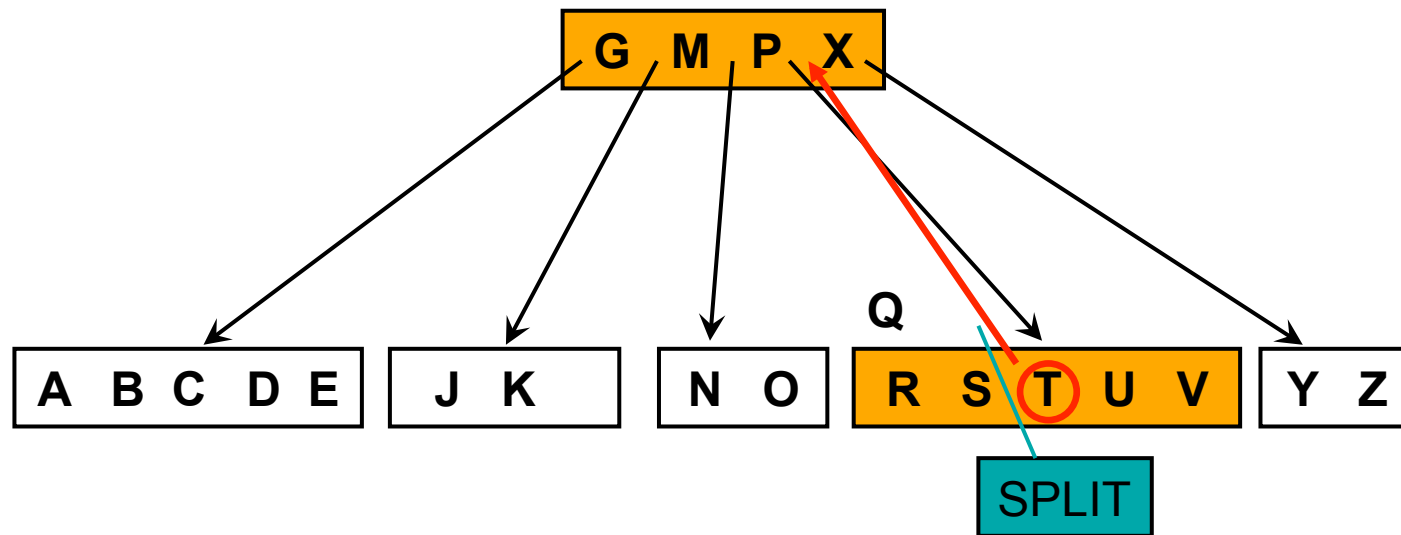
Inserting Q



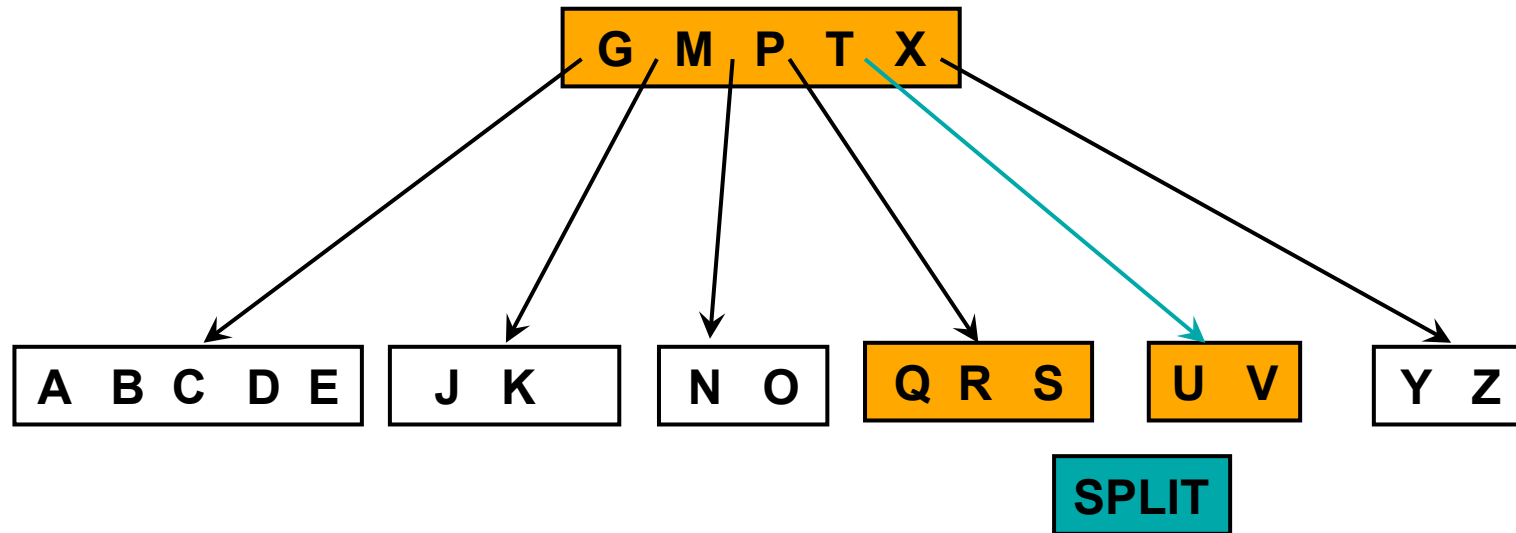
Inserting Q



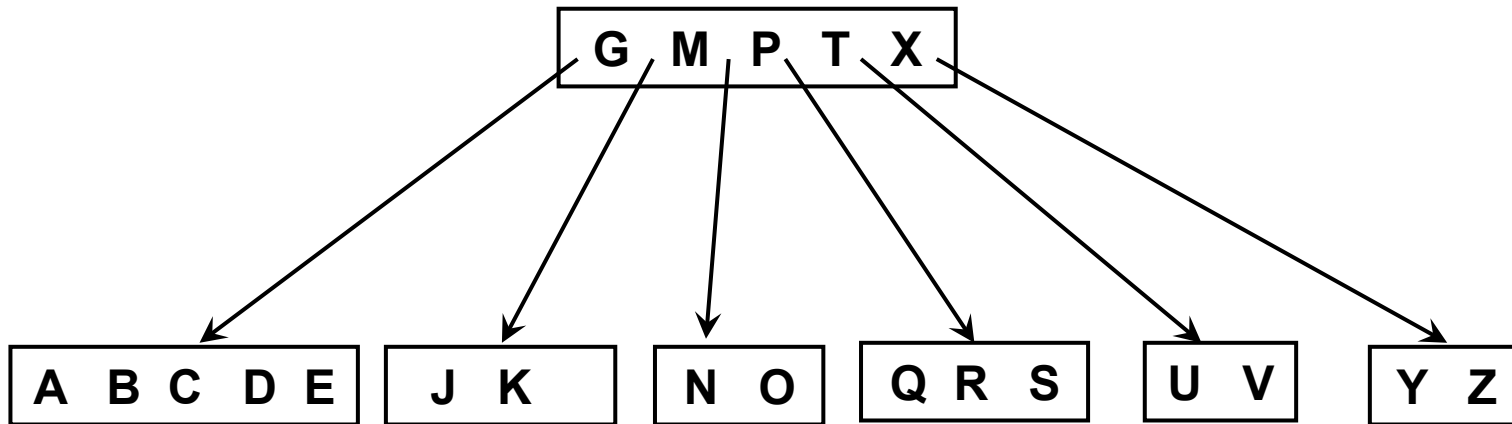
Inserting Q



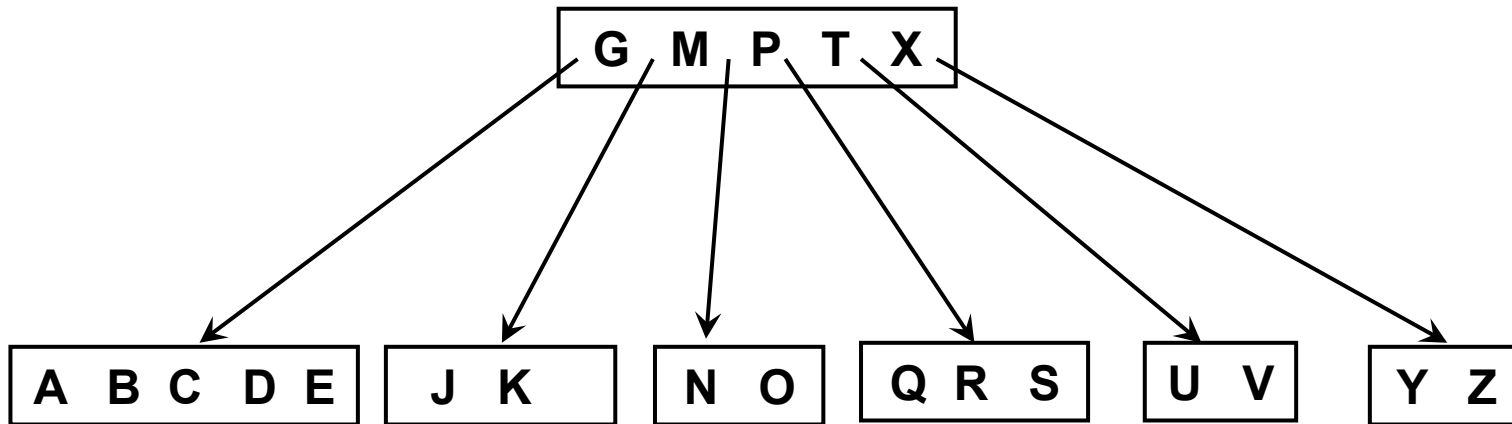
Inserting Q



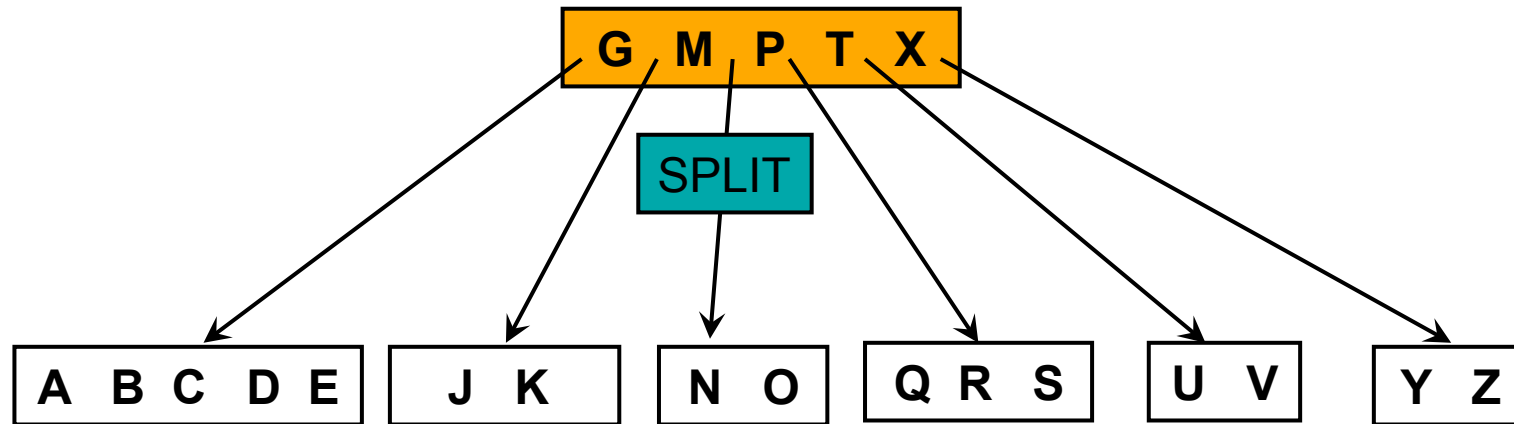
Inserting Q



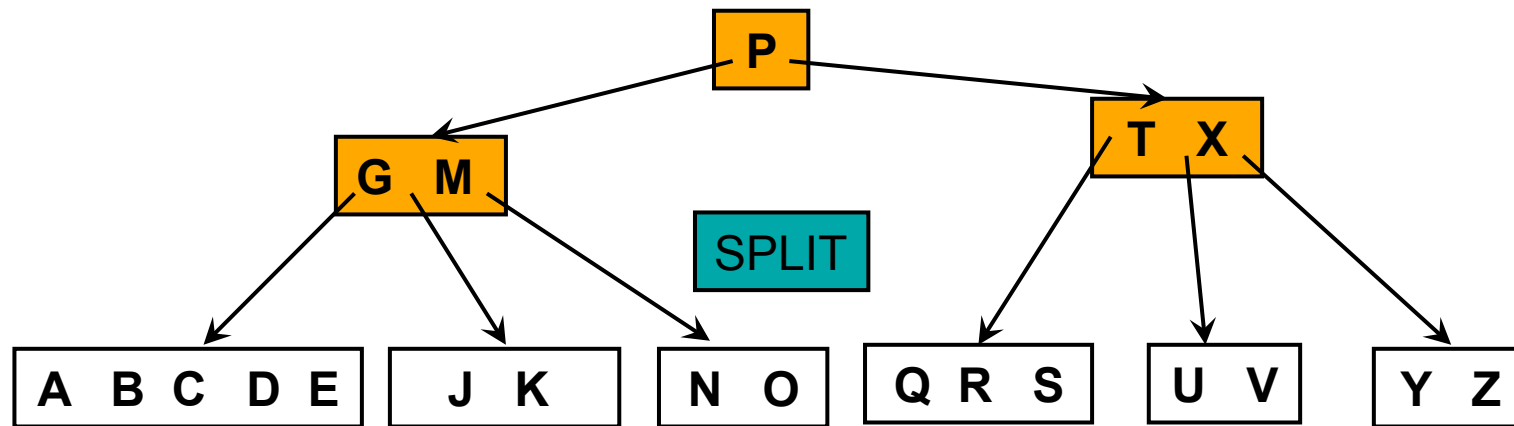
Inserting L



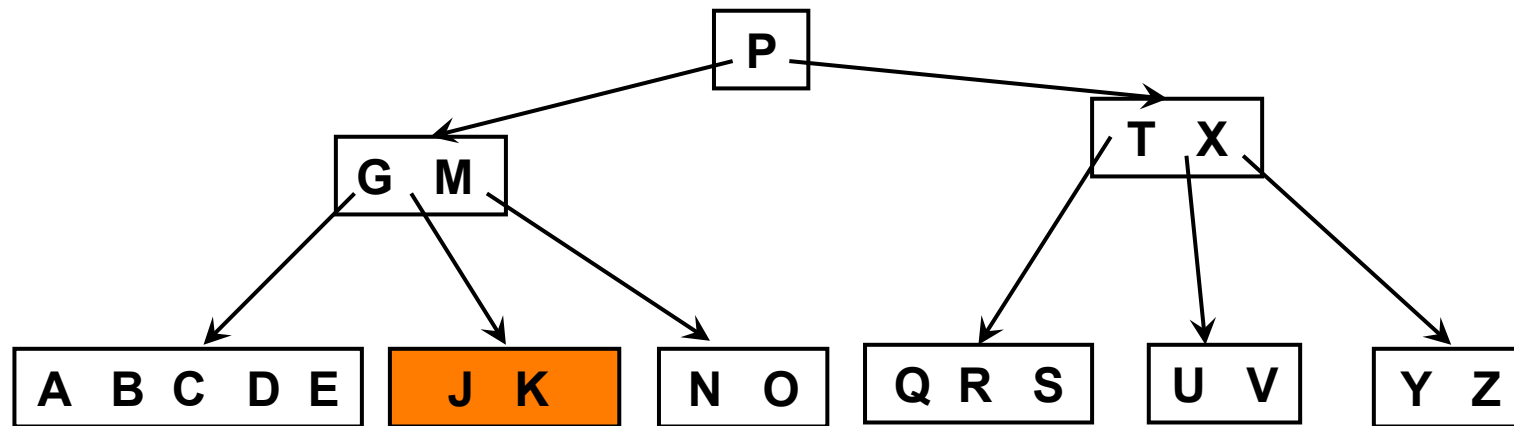
Inserting L



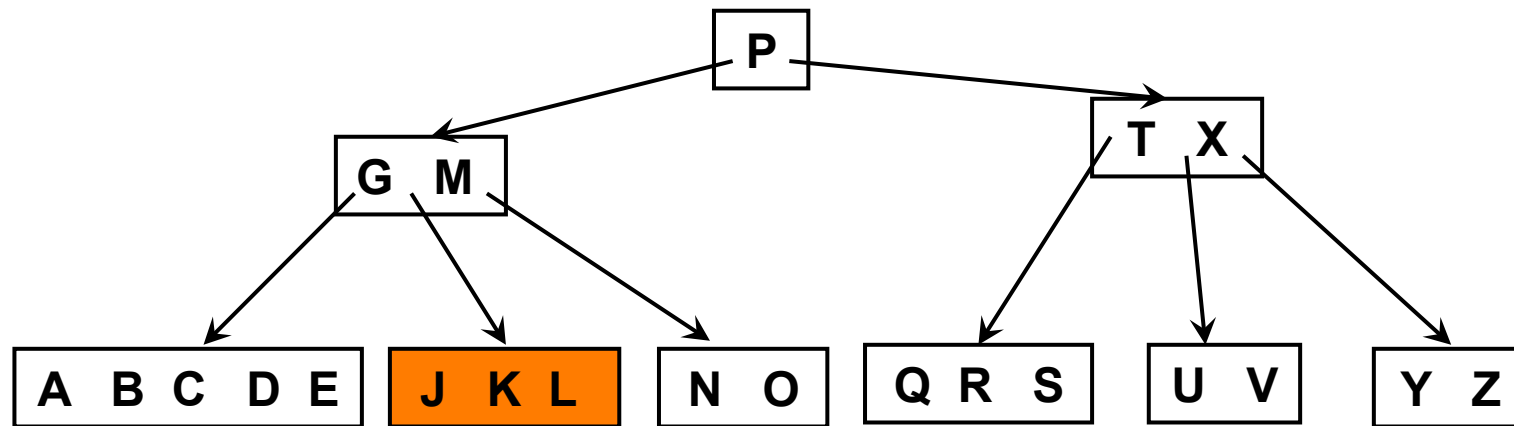
Inserting L



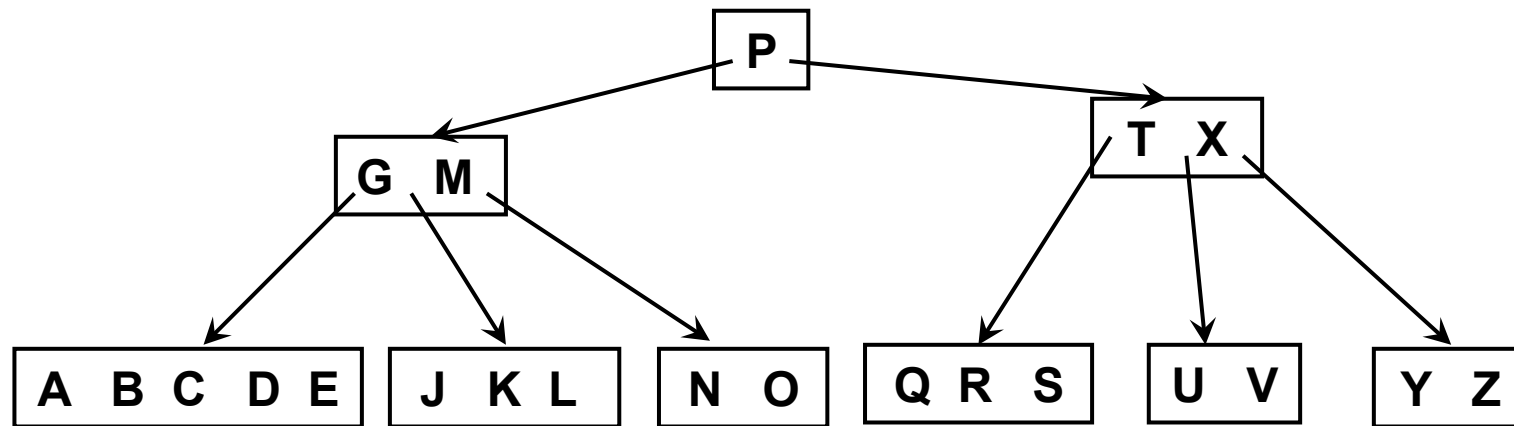
Inserting L



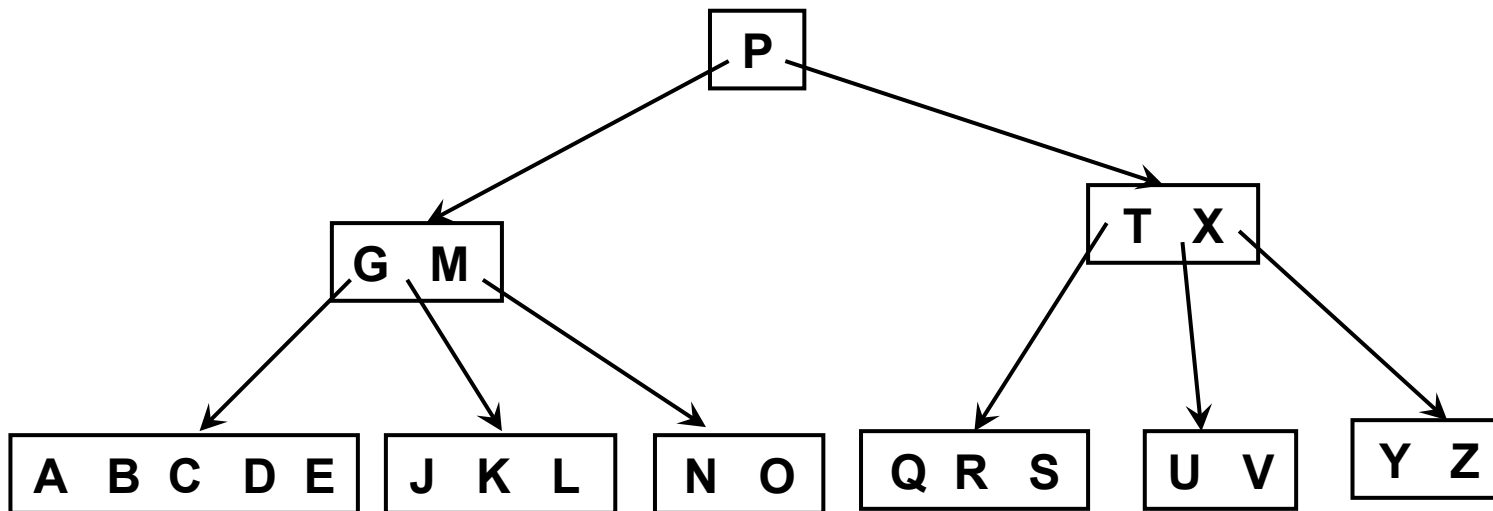
Inserting L



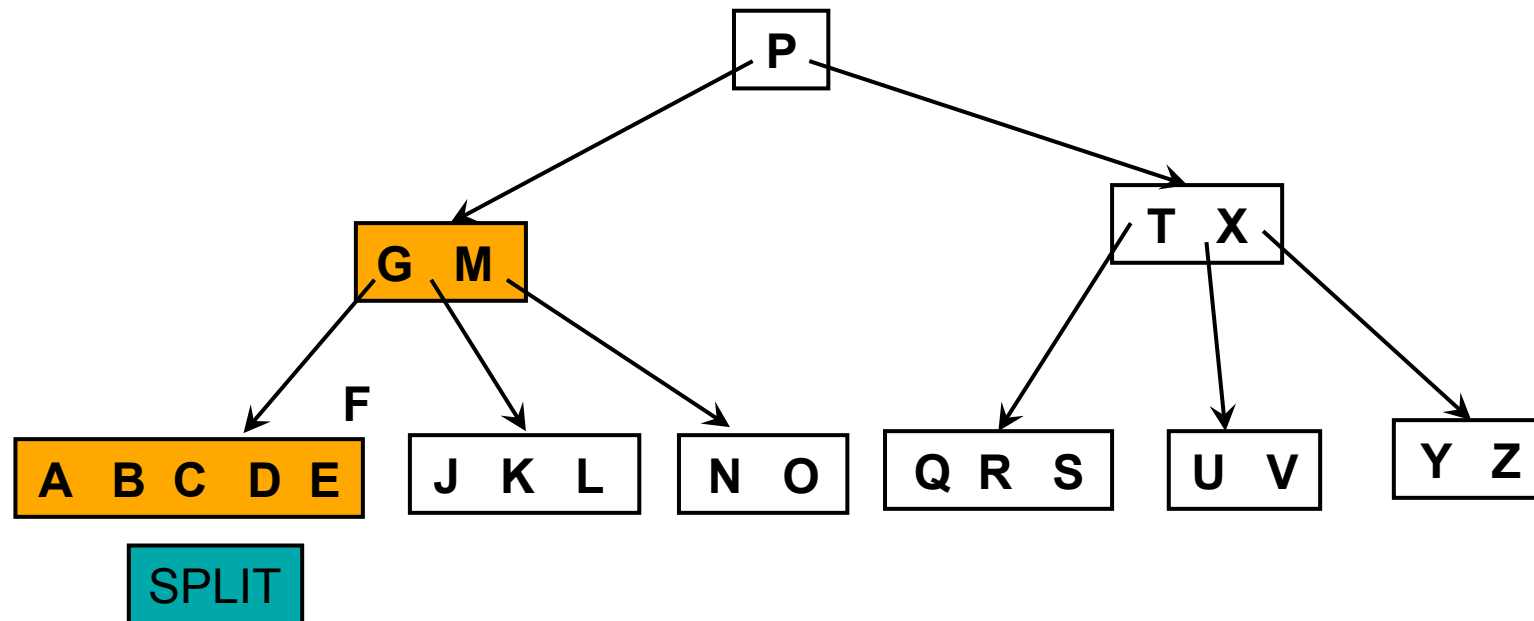
Inserting L



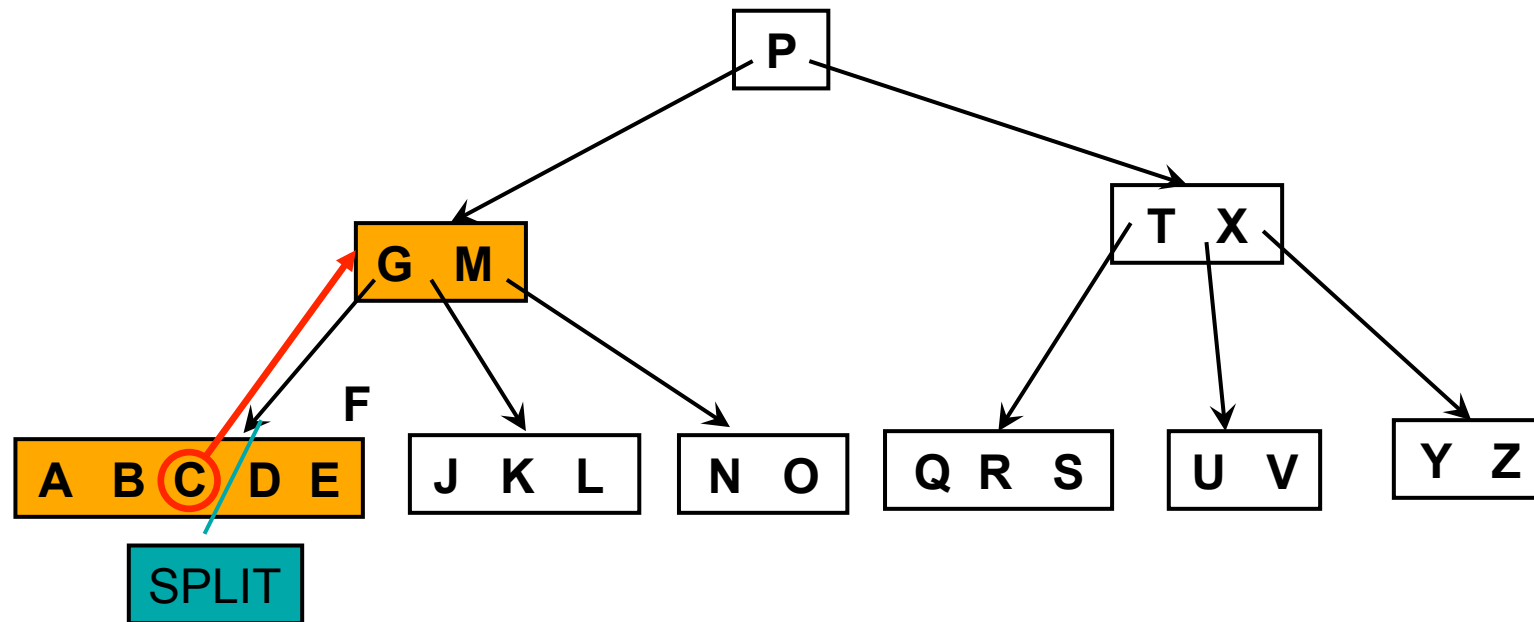
Inserting F



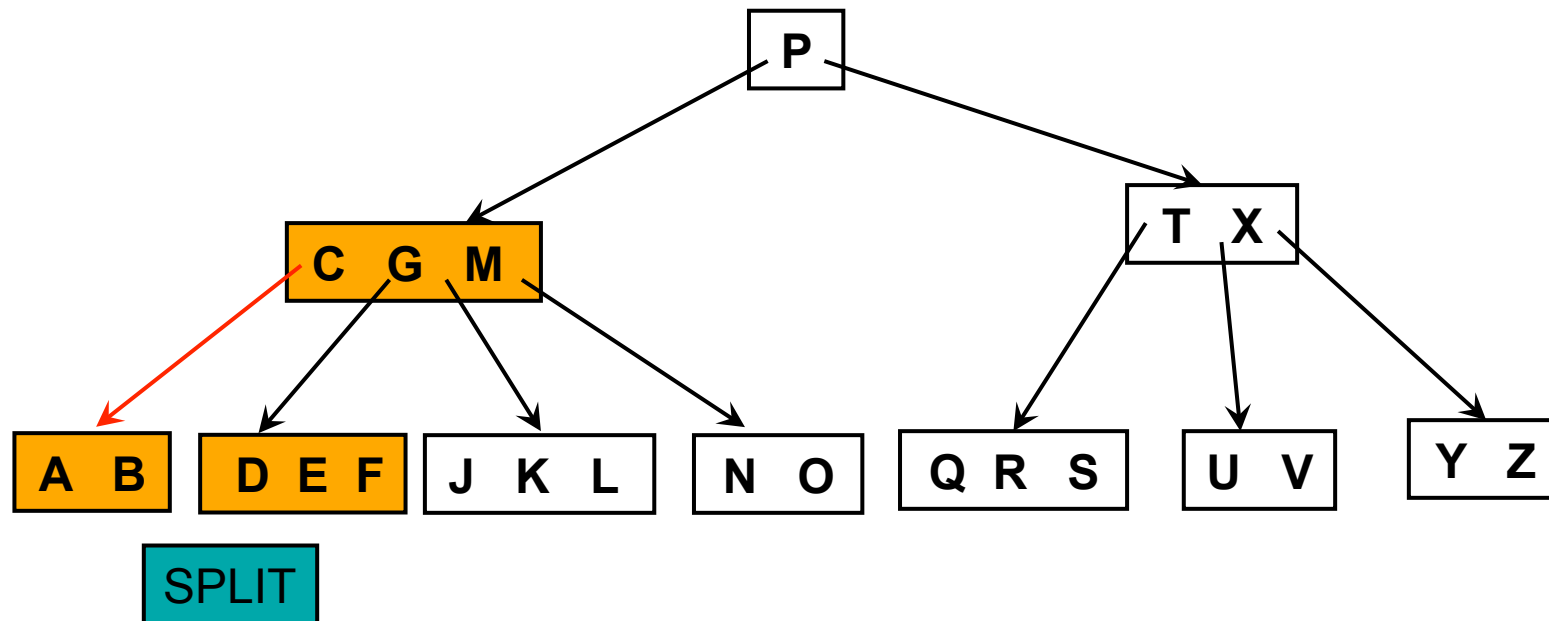
Inserting F



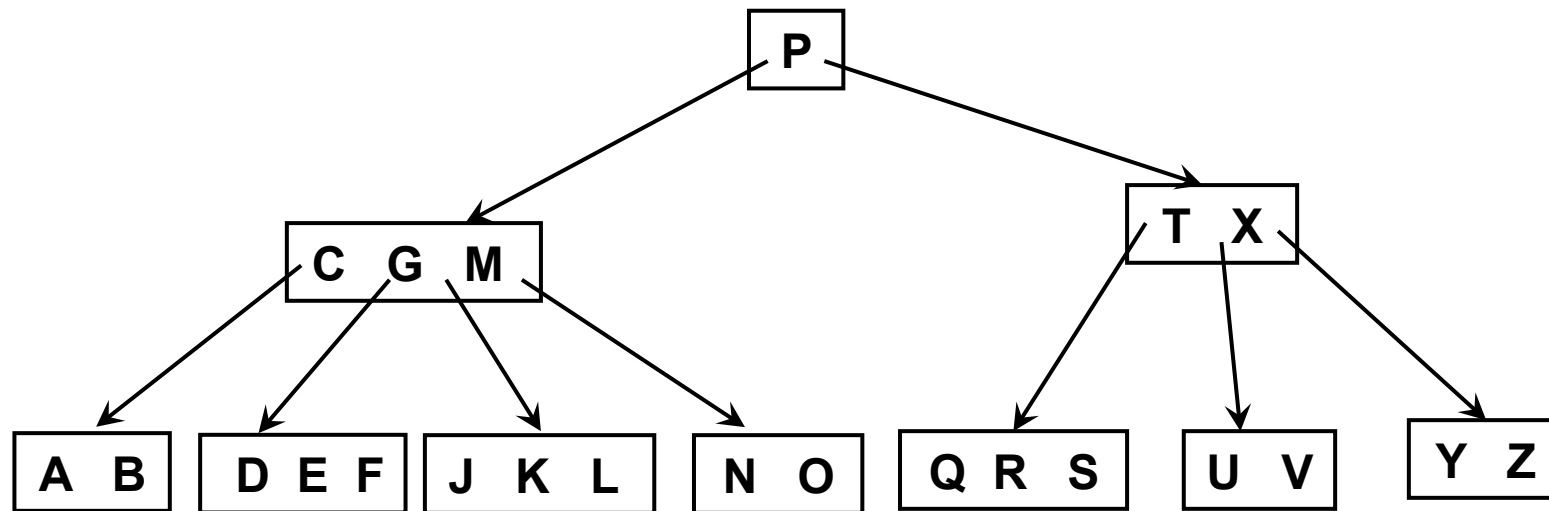
Inserting F



Inserting F



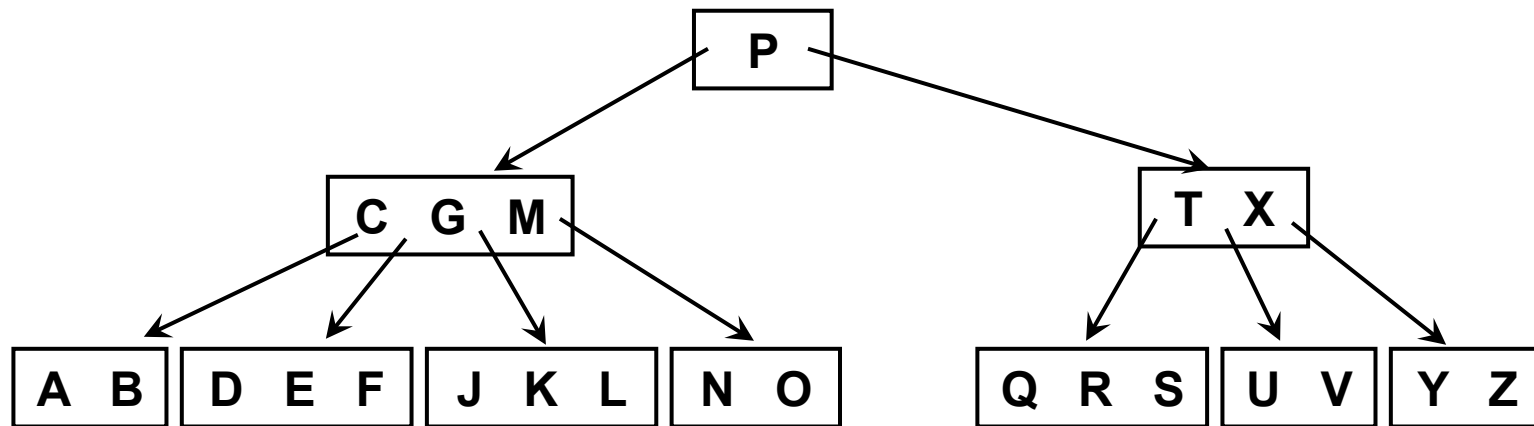
Inserting F



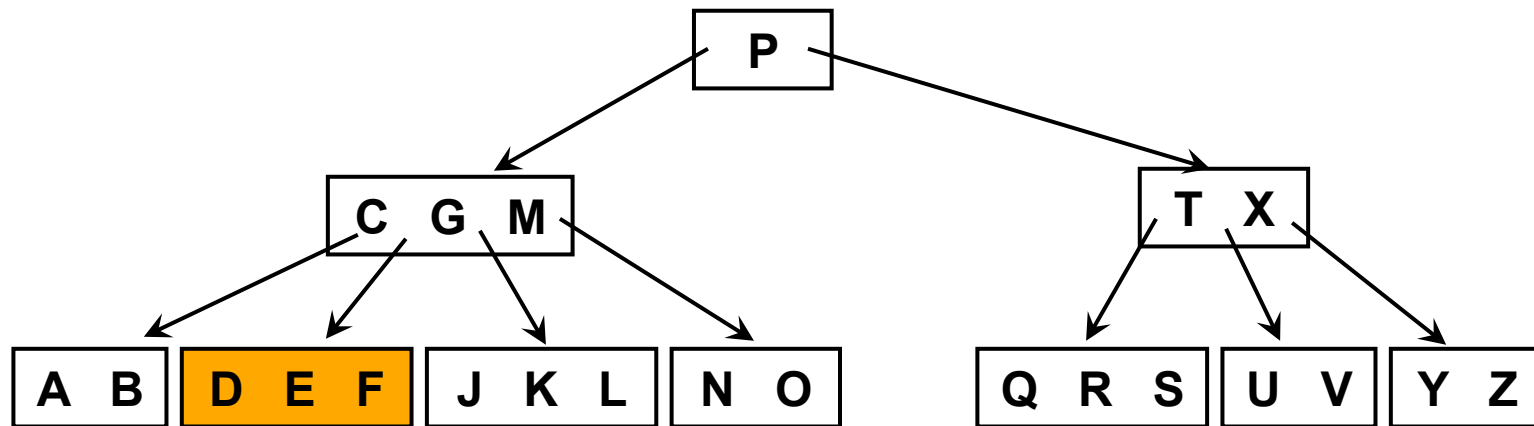
Deleting

- Case 1: key k is in a leaf node
- Case 2: key k is in an internal node
 - Subcase a: having a child with at least t keys preceding k
 - Subcase b: having a child with at least t keys following k
 - Subcase c: both have $t-1$ keys
- Case 3: key k is not in an internal node and root of an appropriate subtree has only $t-1$ keys
 - Subcase a: subtree has only $t-1$ keys having a sibling with at least t keys
 - Subcase b: both subtree and immediate siblings have $t-1$ keys

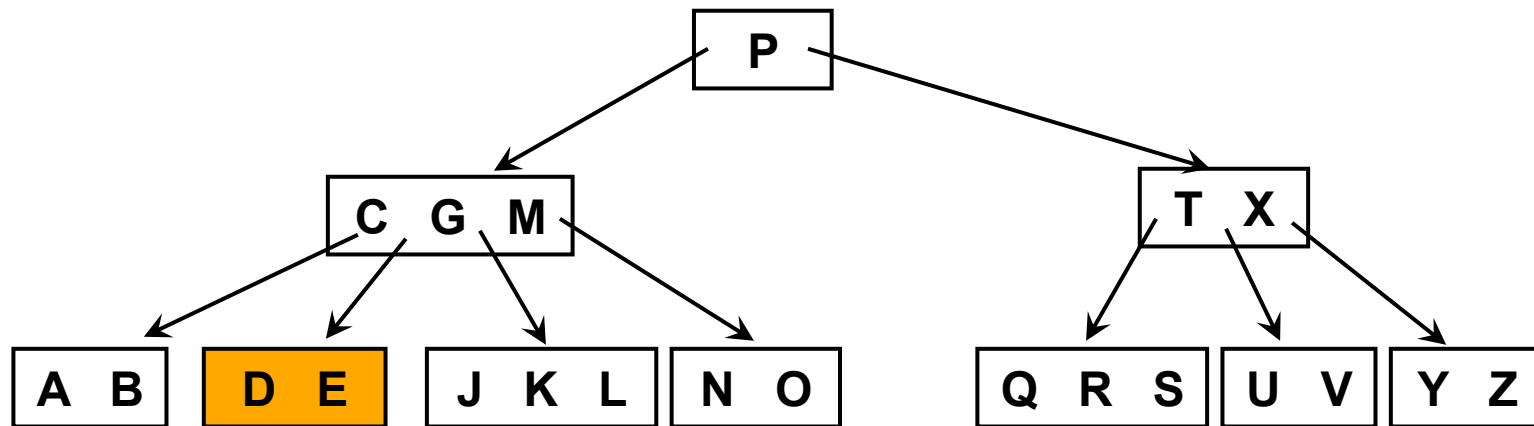
Deleting F (Case 1)



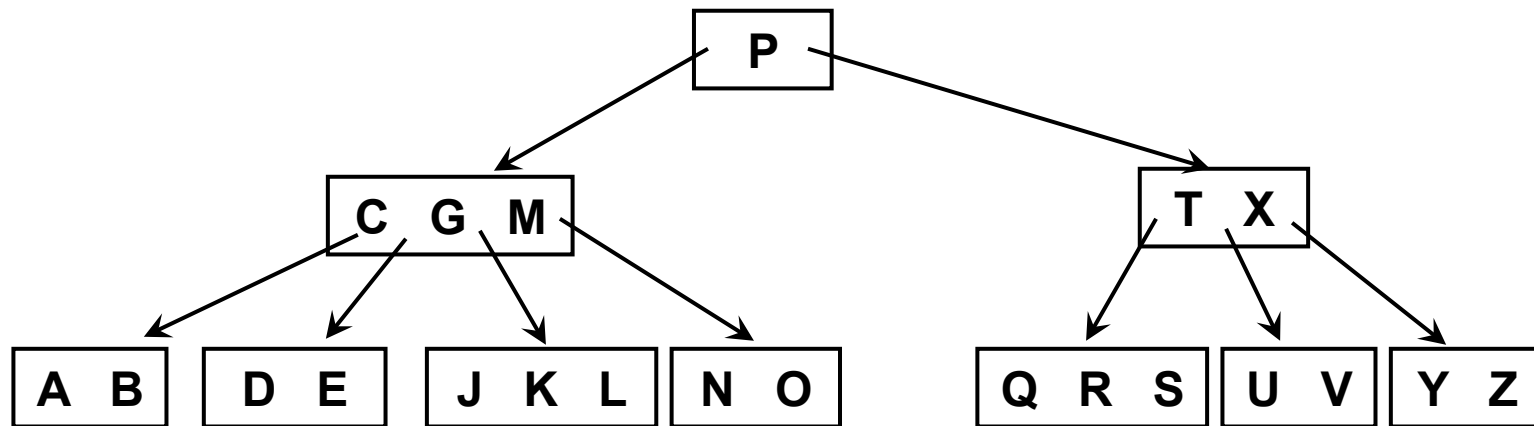
Deleting F



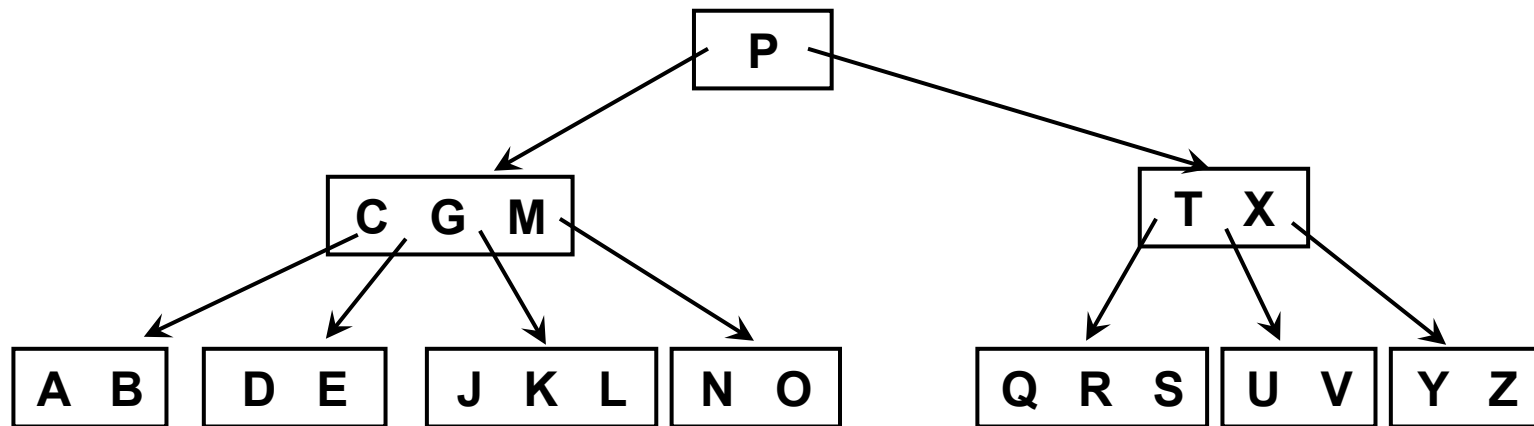
Deleting F



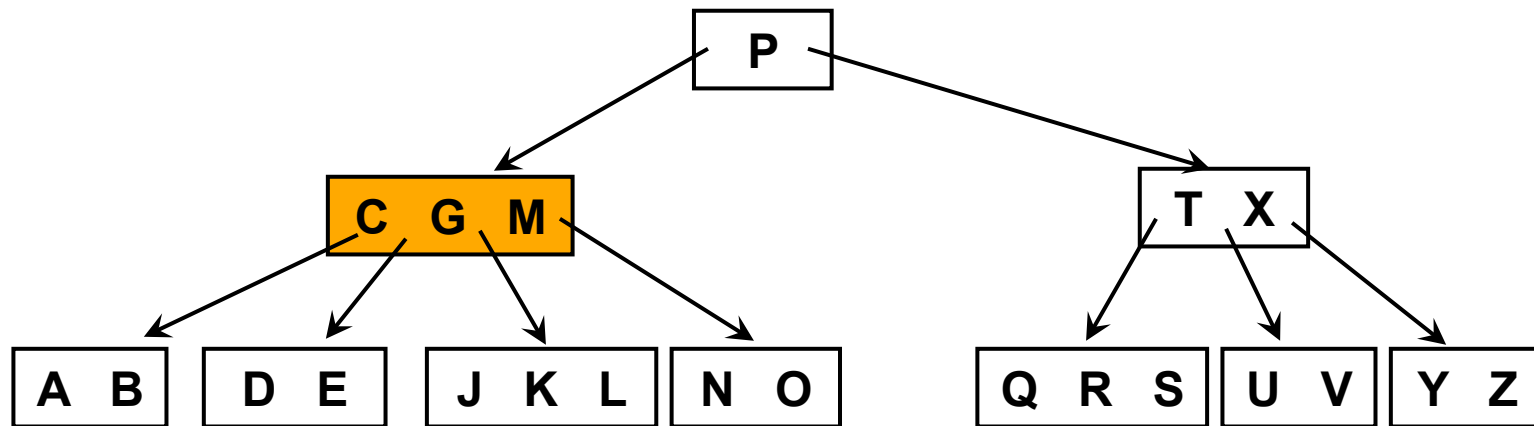
Deleting F



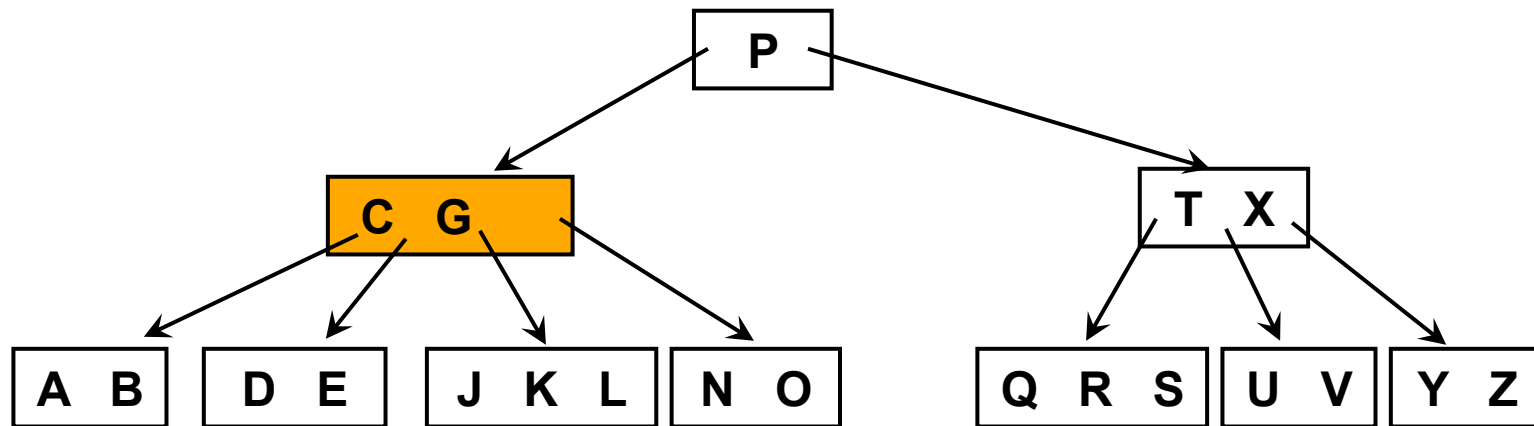
Deleting M (Case 2a)



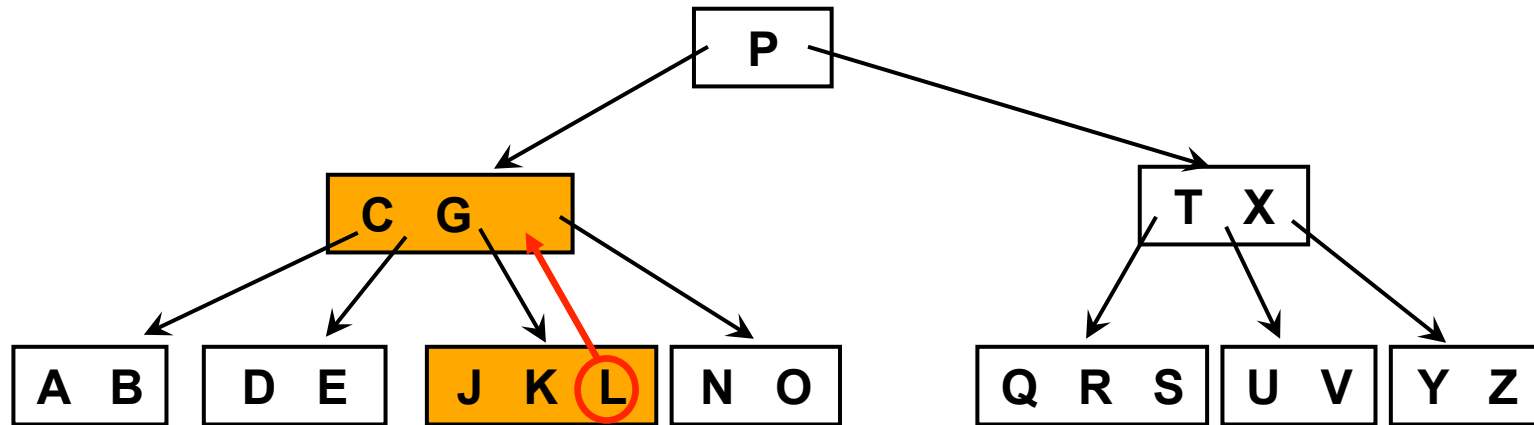
Deleting M



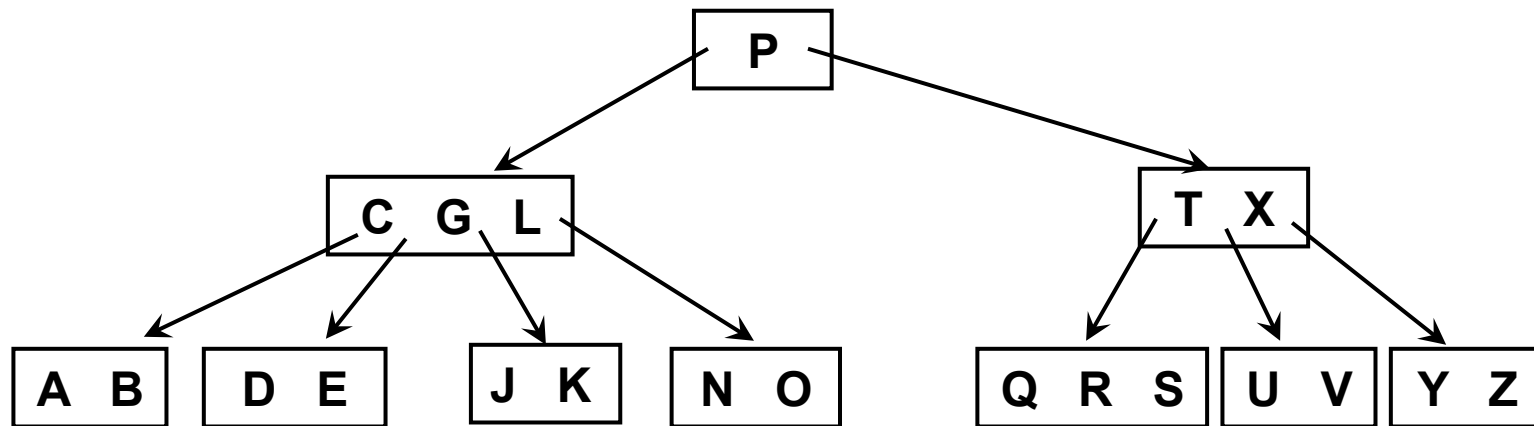
Deleting M



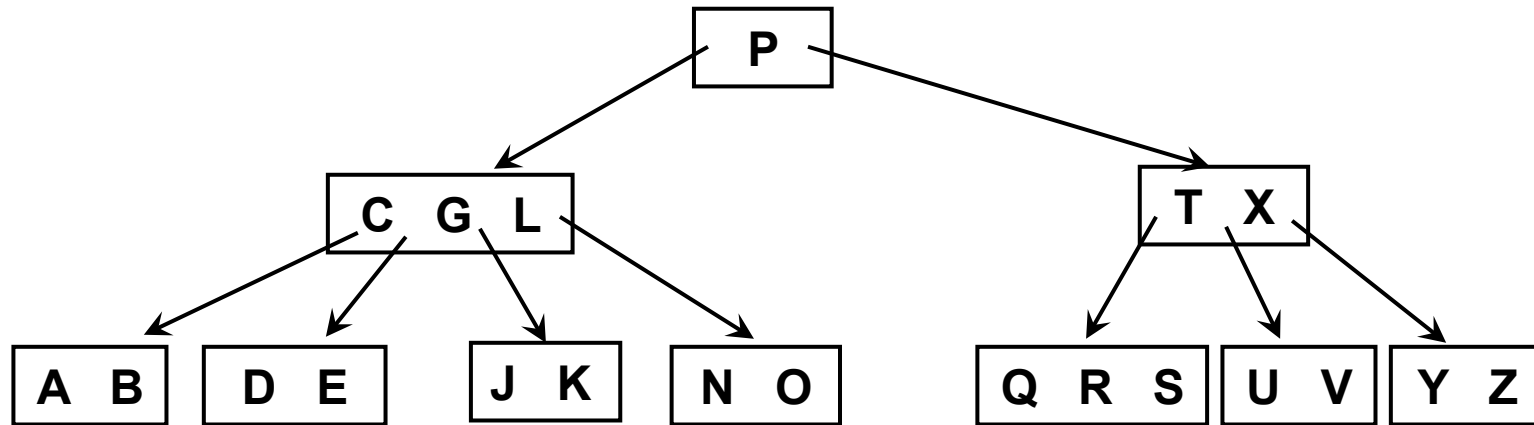
Deleting M



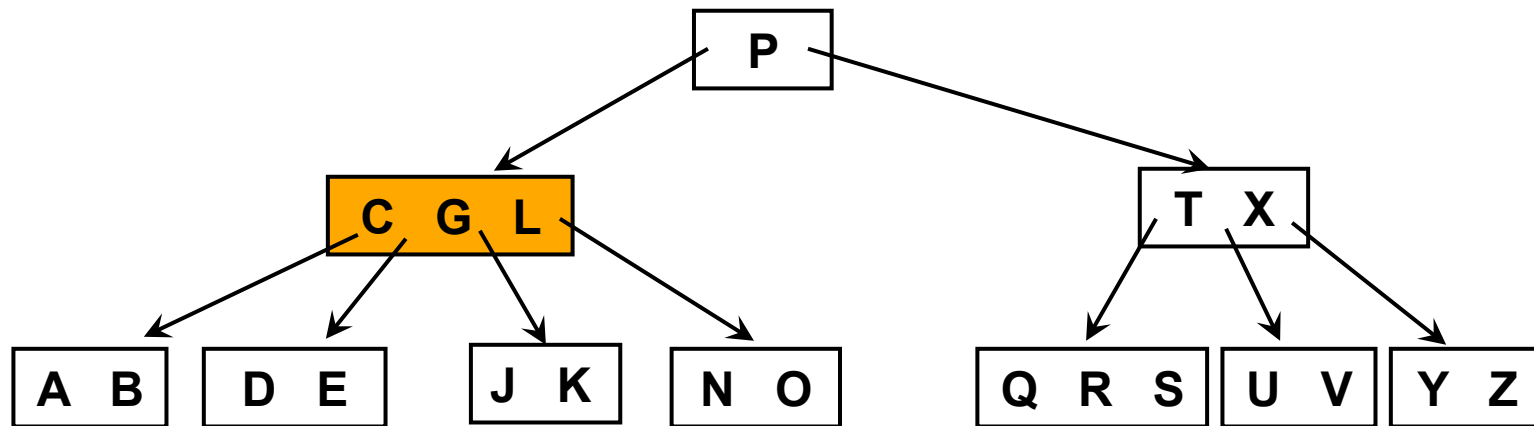
Deleting M



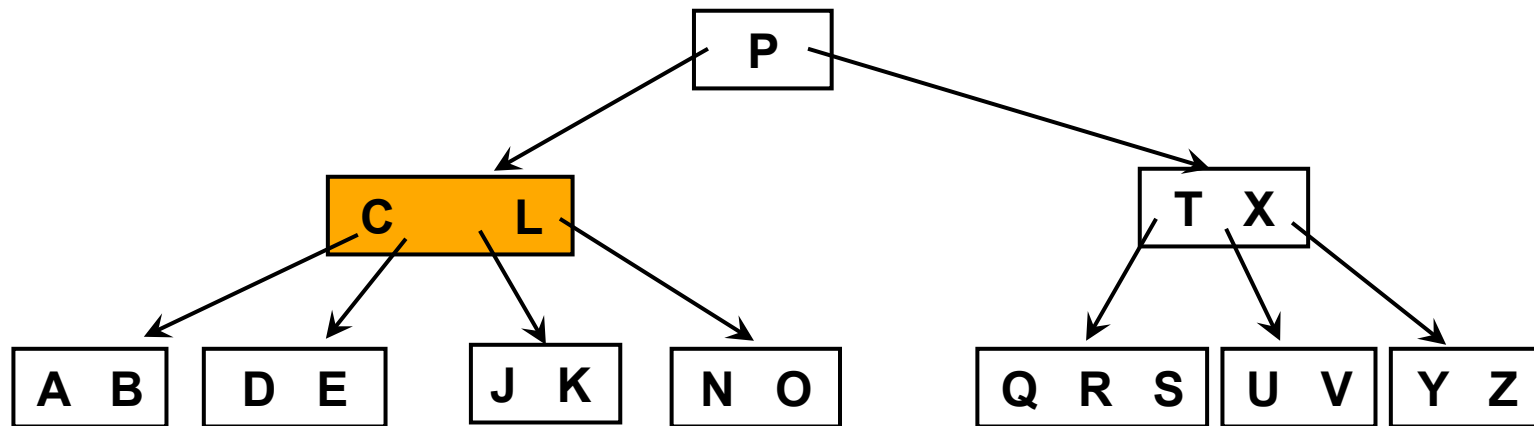
Deleting G (Case 2c)



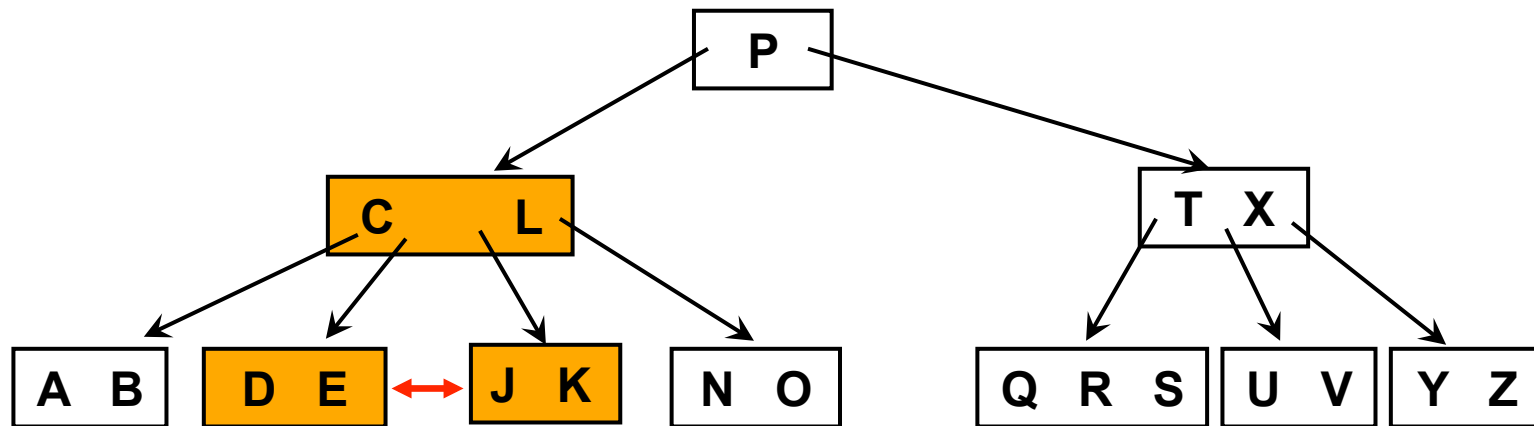
Deleting G



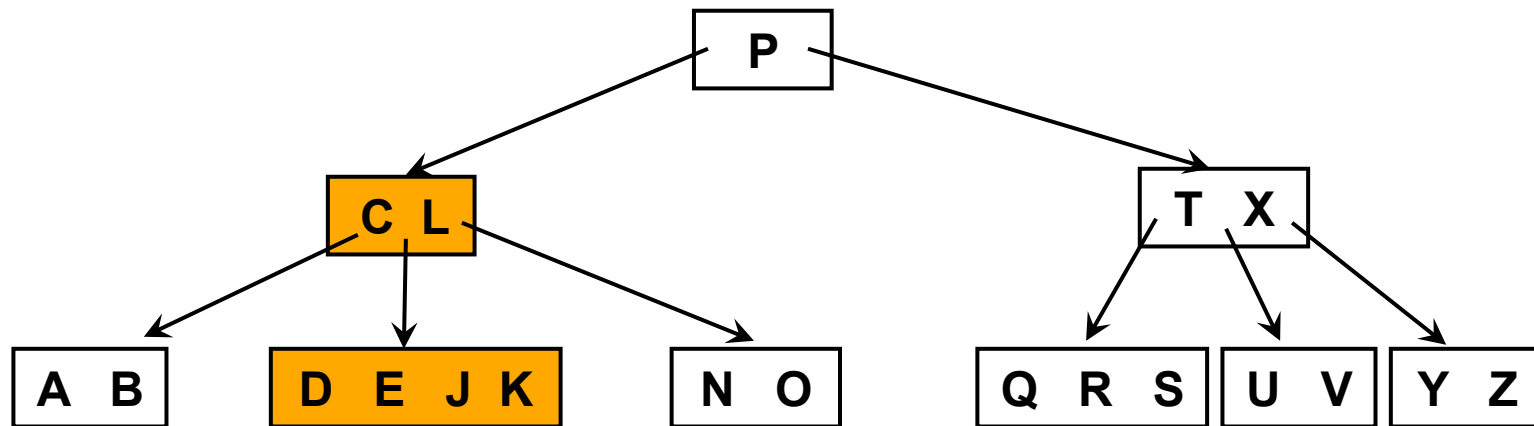
Deleting G



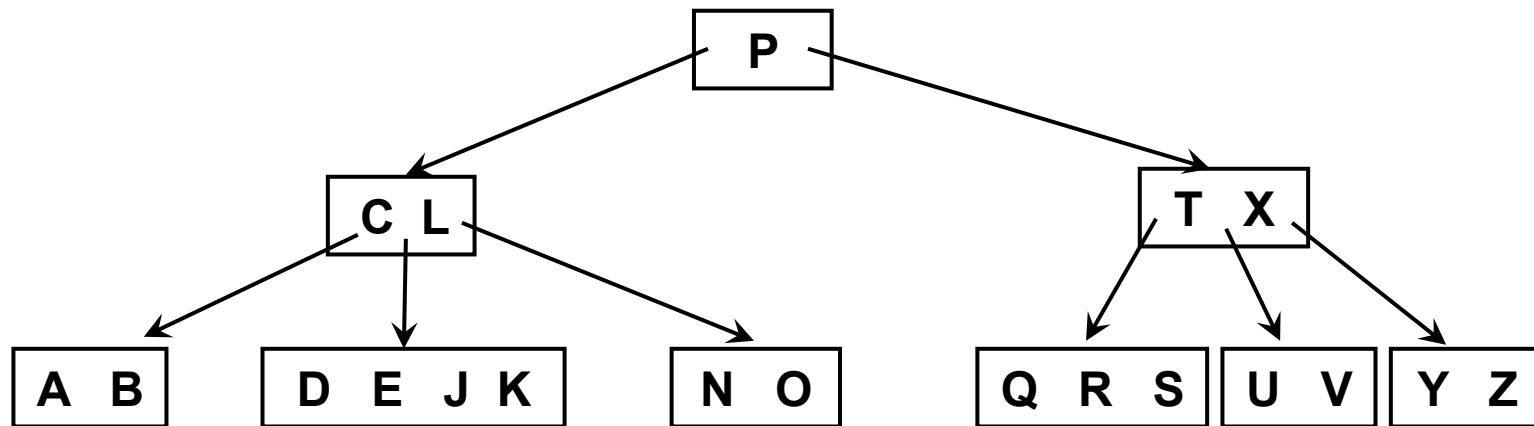
Deleting G



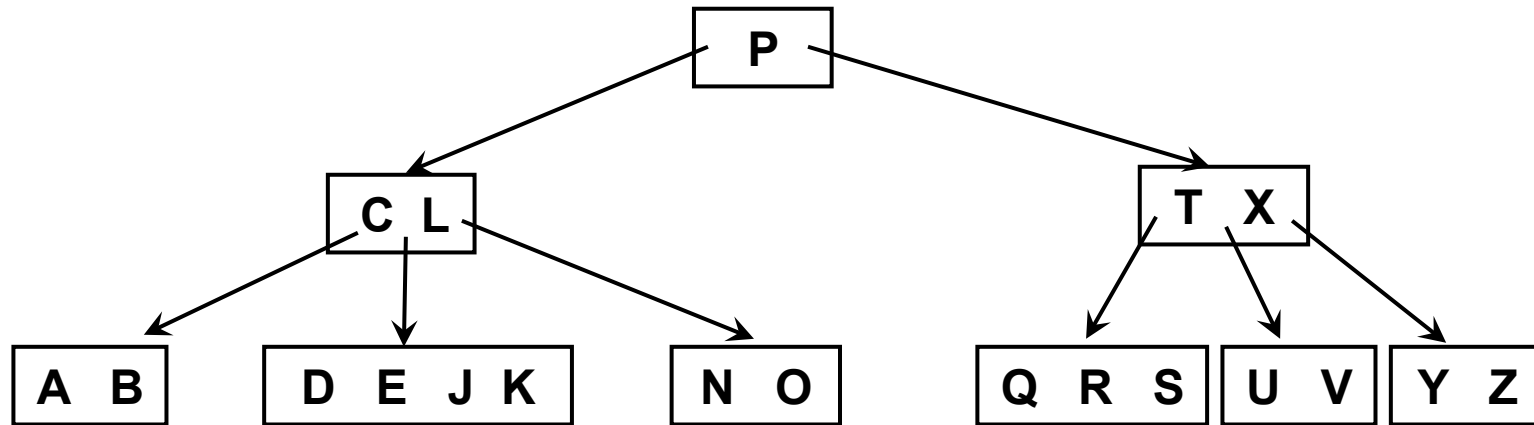
Deleting G



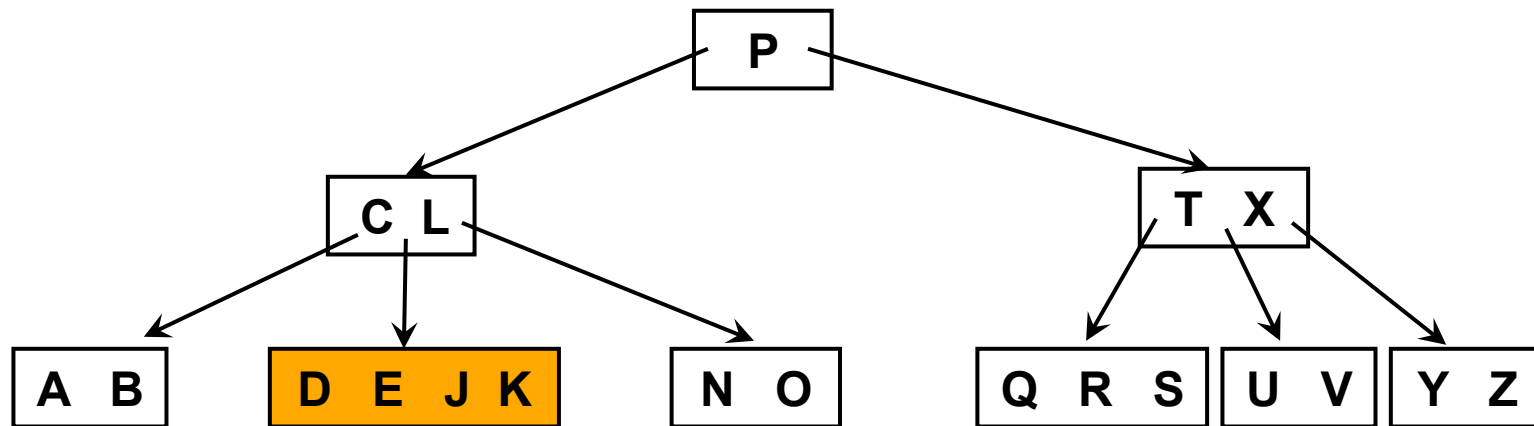
Deleting G



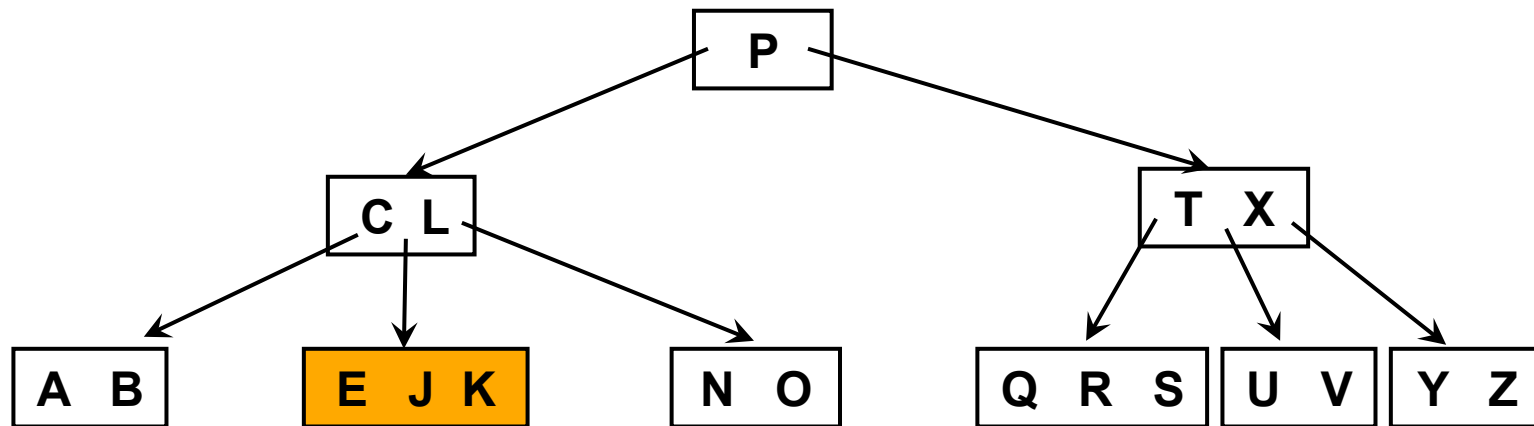
Deleting D (Case 3b)



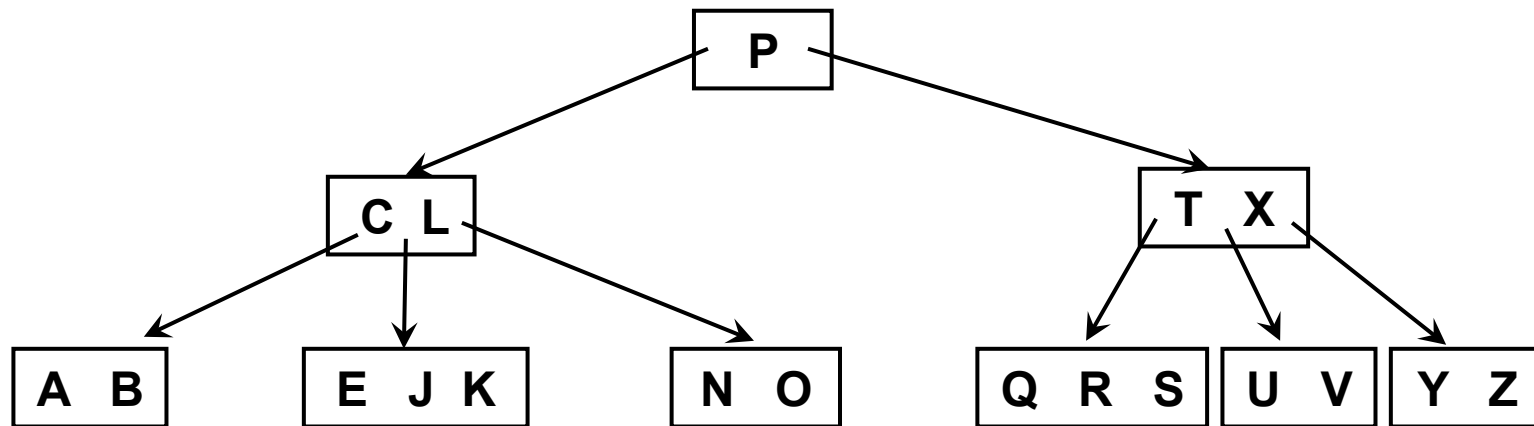
Deleting D



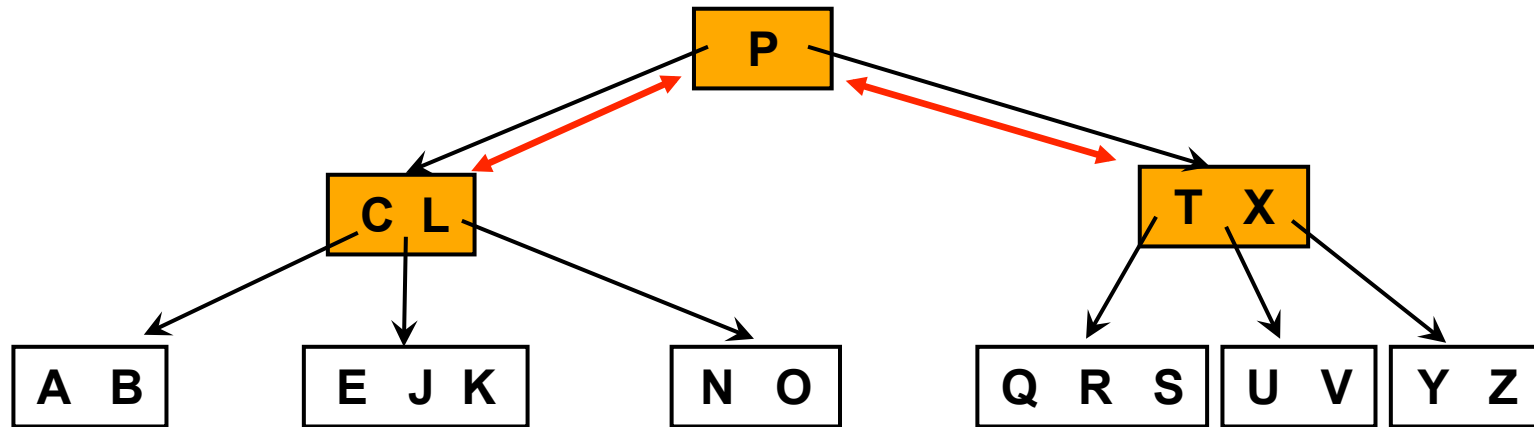
Deleting D



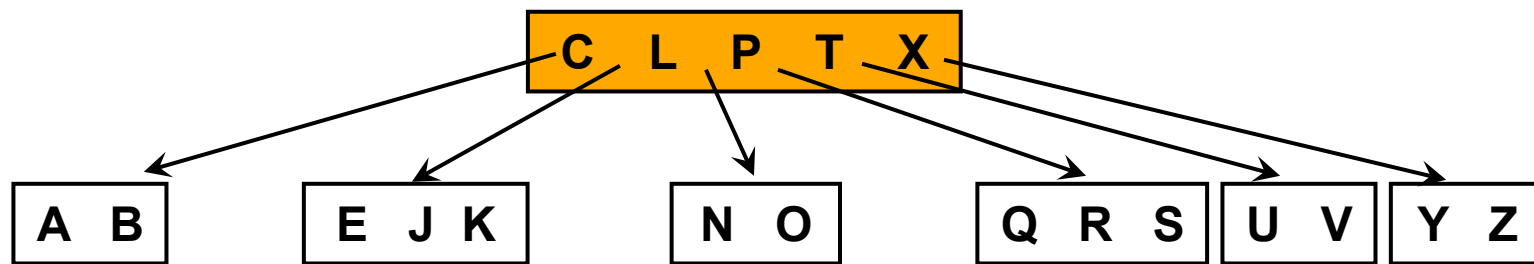
Deleting D



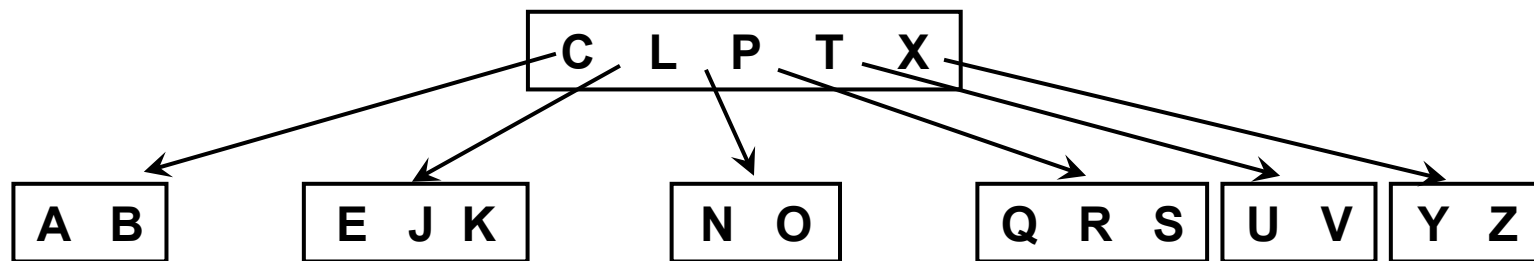
Deleting D



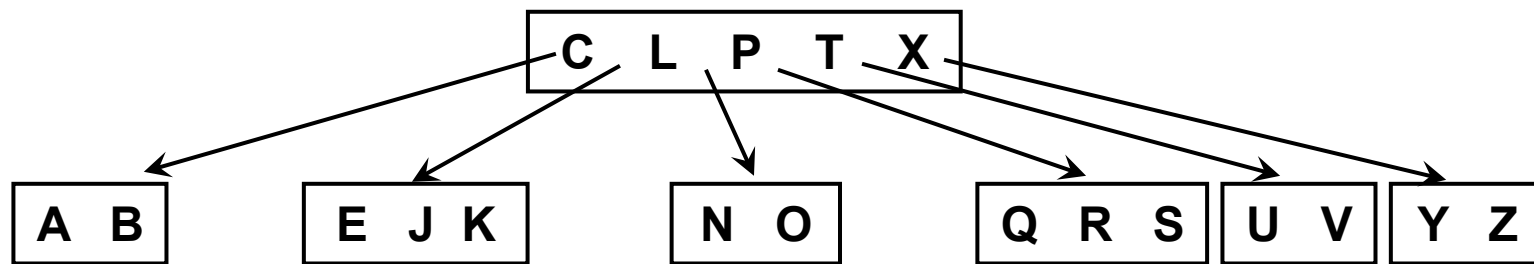
Deleting D



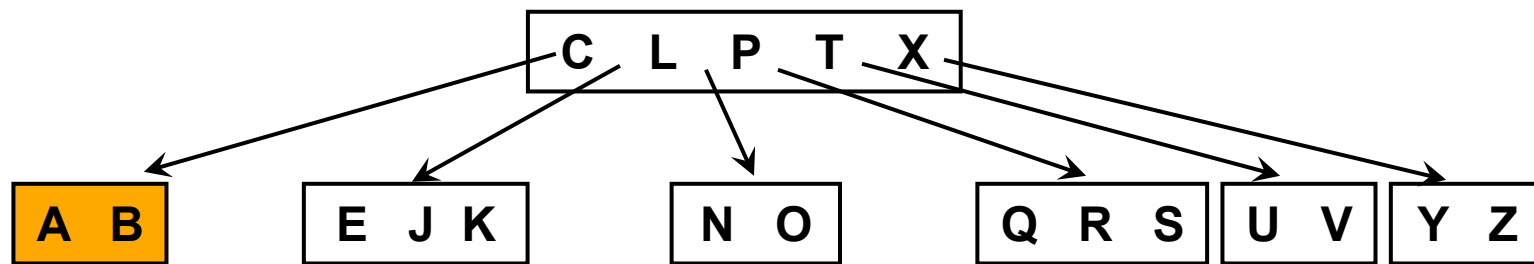
Deleting D



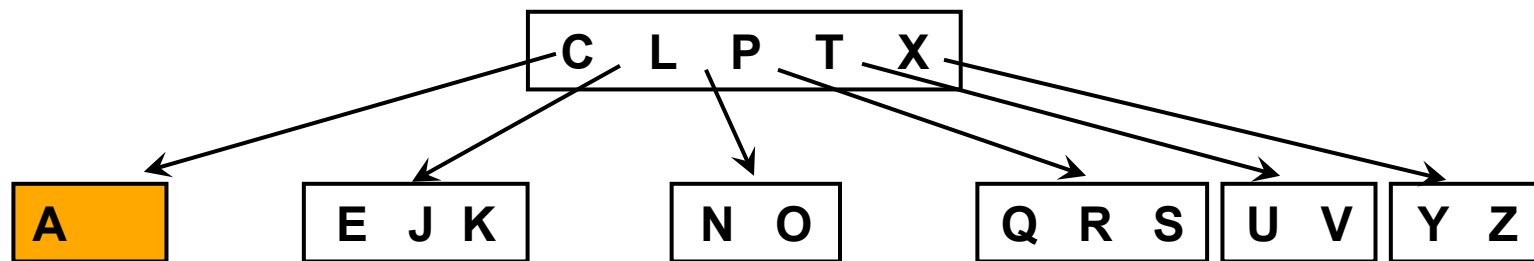
Deleting B (Case 3a)



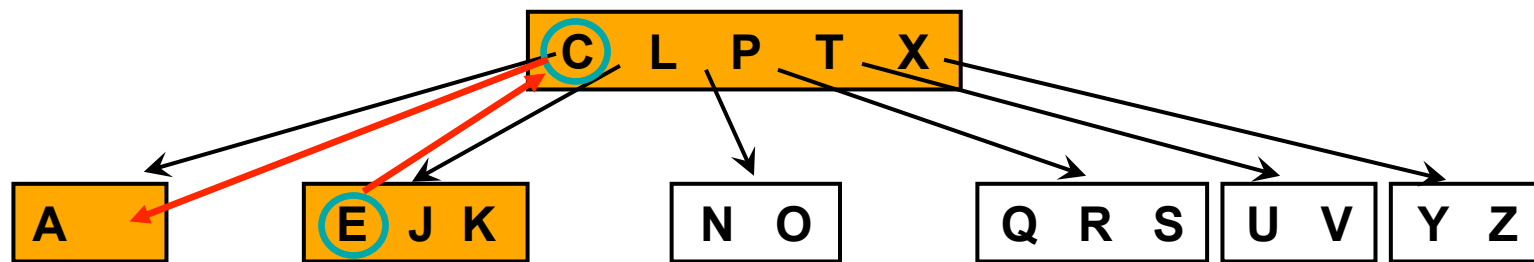
Deleting B



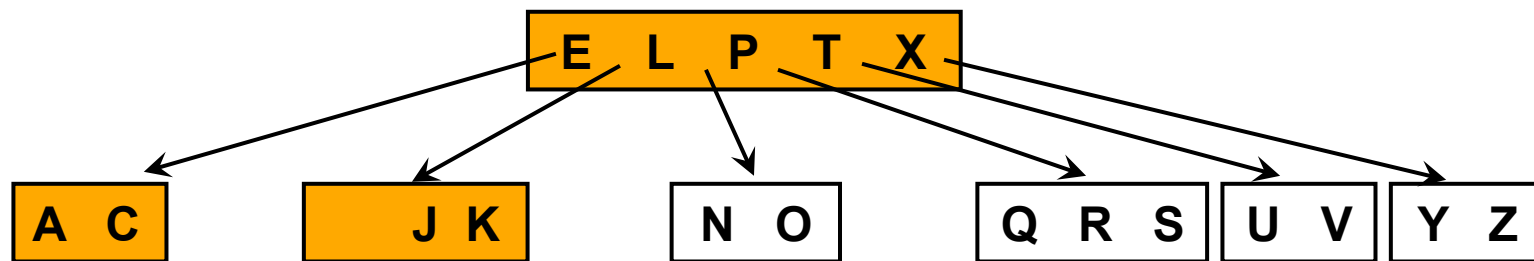
Deleting B



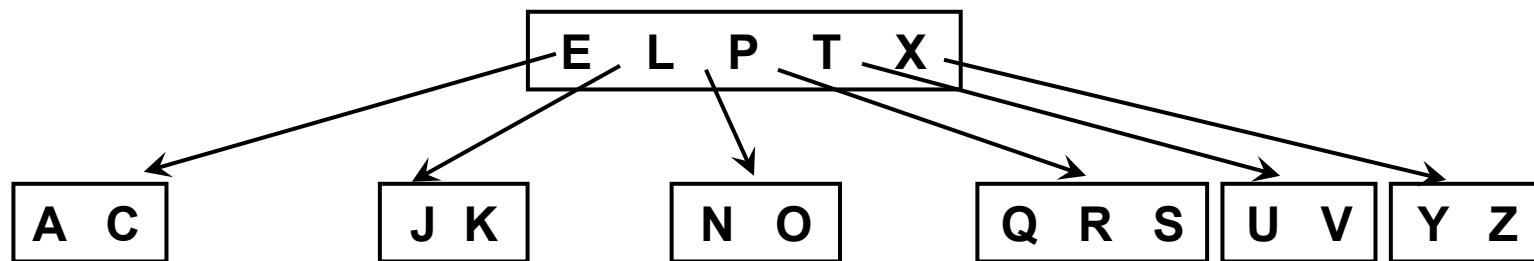
Deleting B



Deleting B



Deleting B



Another type of B-Tree: m-way

- m-way: m pointers and m keys
- For most of the database applications satellite data is kept only in the leaves
- Requires changes in search, insert delete algorithms!
- Input order

C S D T A M P I B W N G U
R K E H O L J Y Q Z F X V

- Insert into a 4-way B-Tree

Insert C, S, D, T

•	C	•	S	•		•	
---	----------	---	----------	---	--	---	--

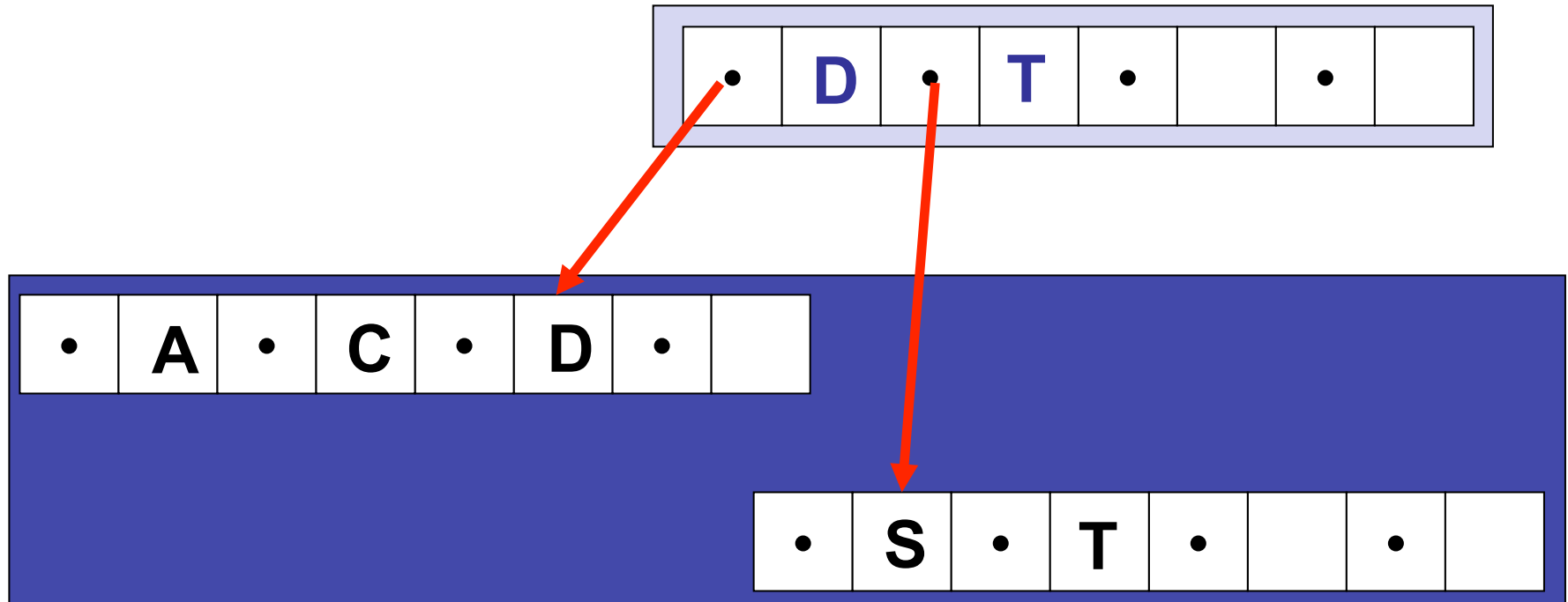
Insert C, S, D, T

•	C	•	D	•	S	•	
---	----------	---	----------	---	----------	---	--

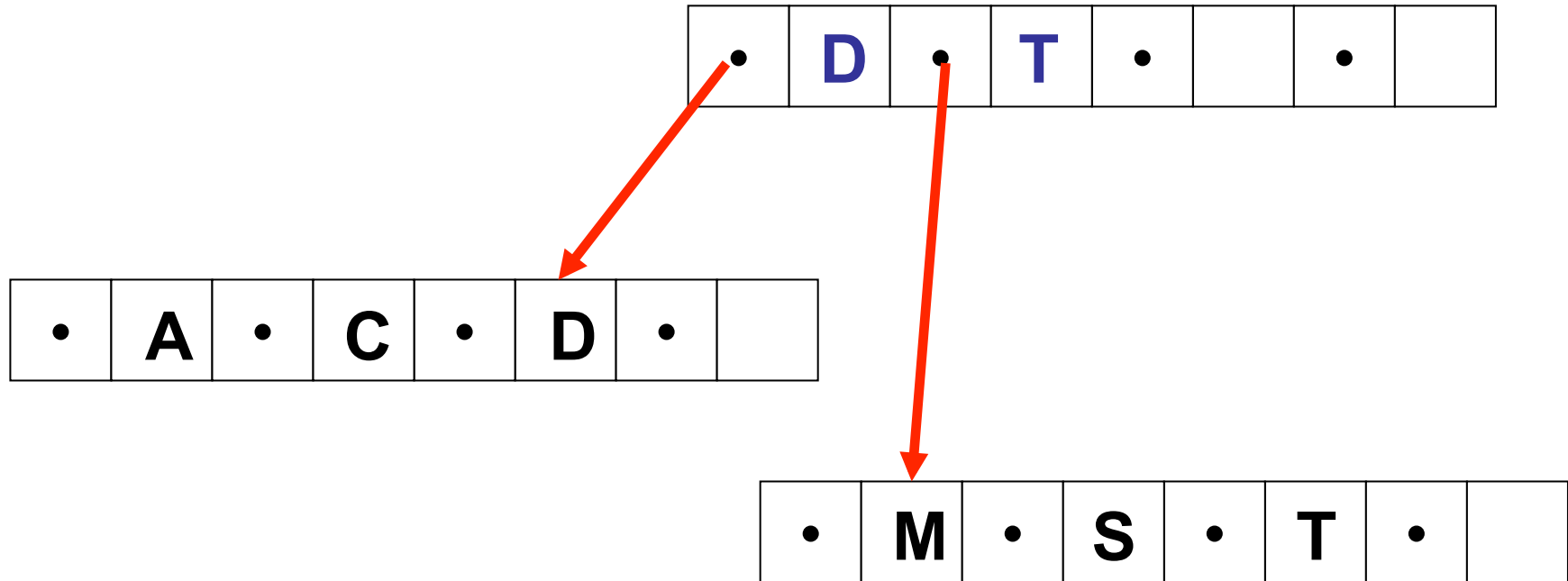
Insert C, S, D, T, A

•	C	•	D	•	S	•	T
---	---	---	---	---	---	---	---

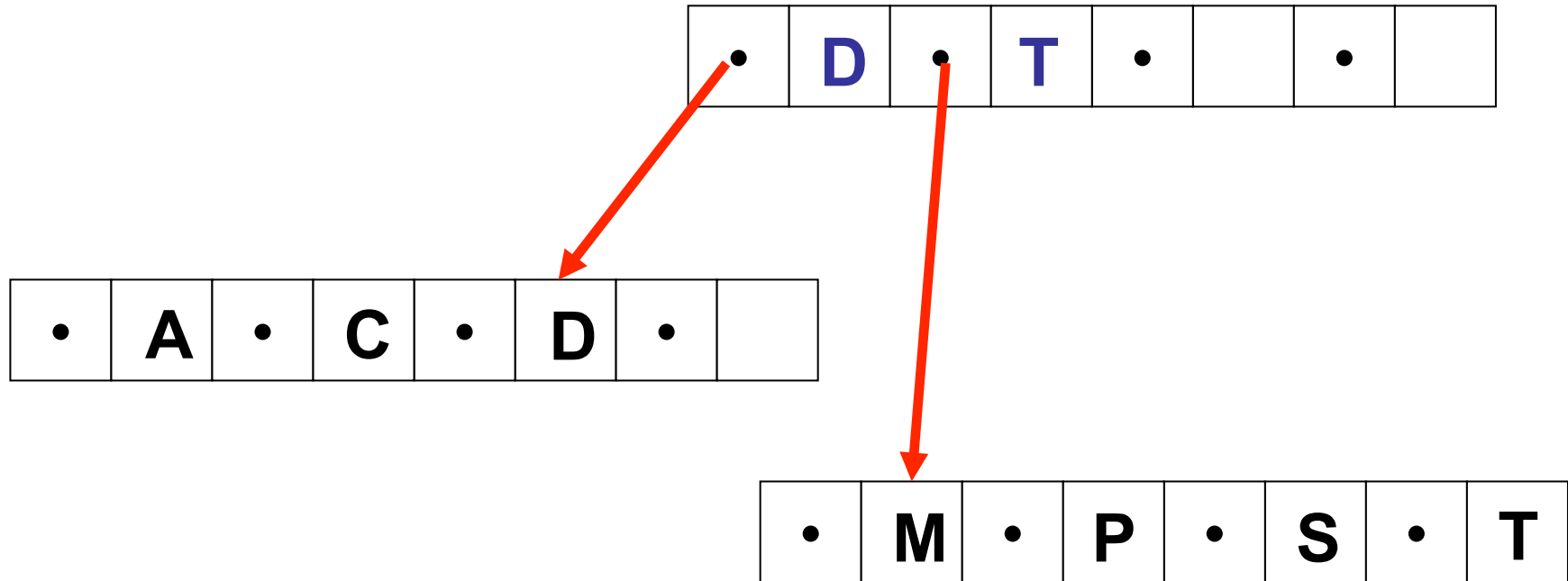
Insert A, M



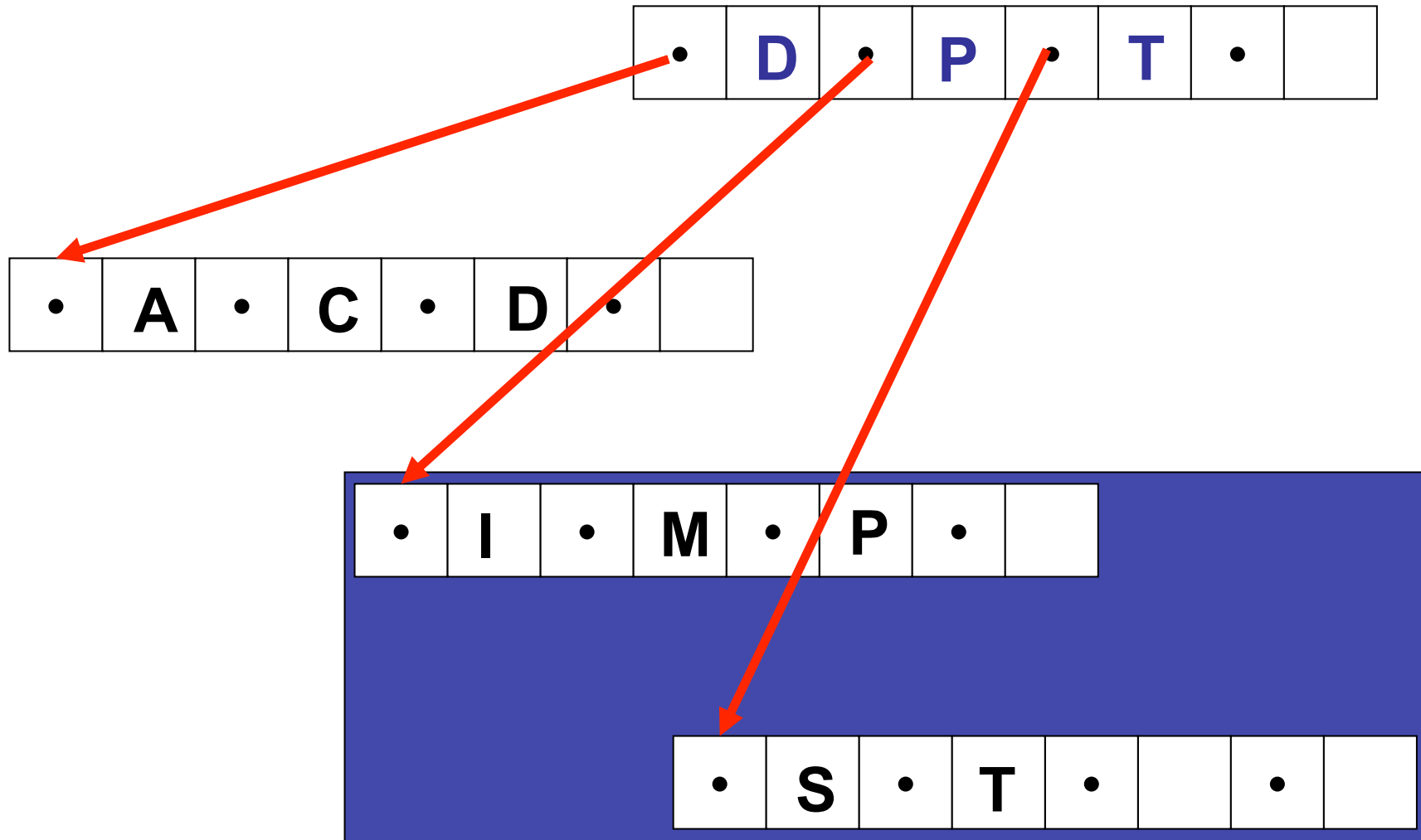
Insert M, P



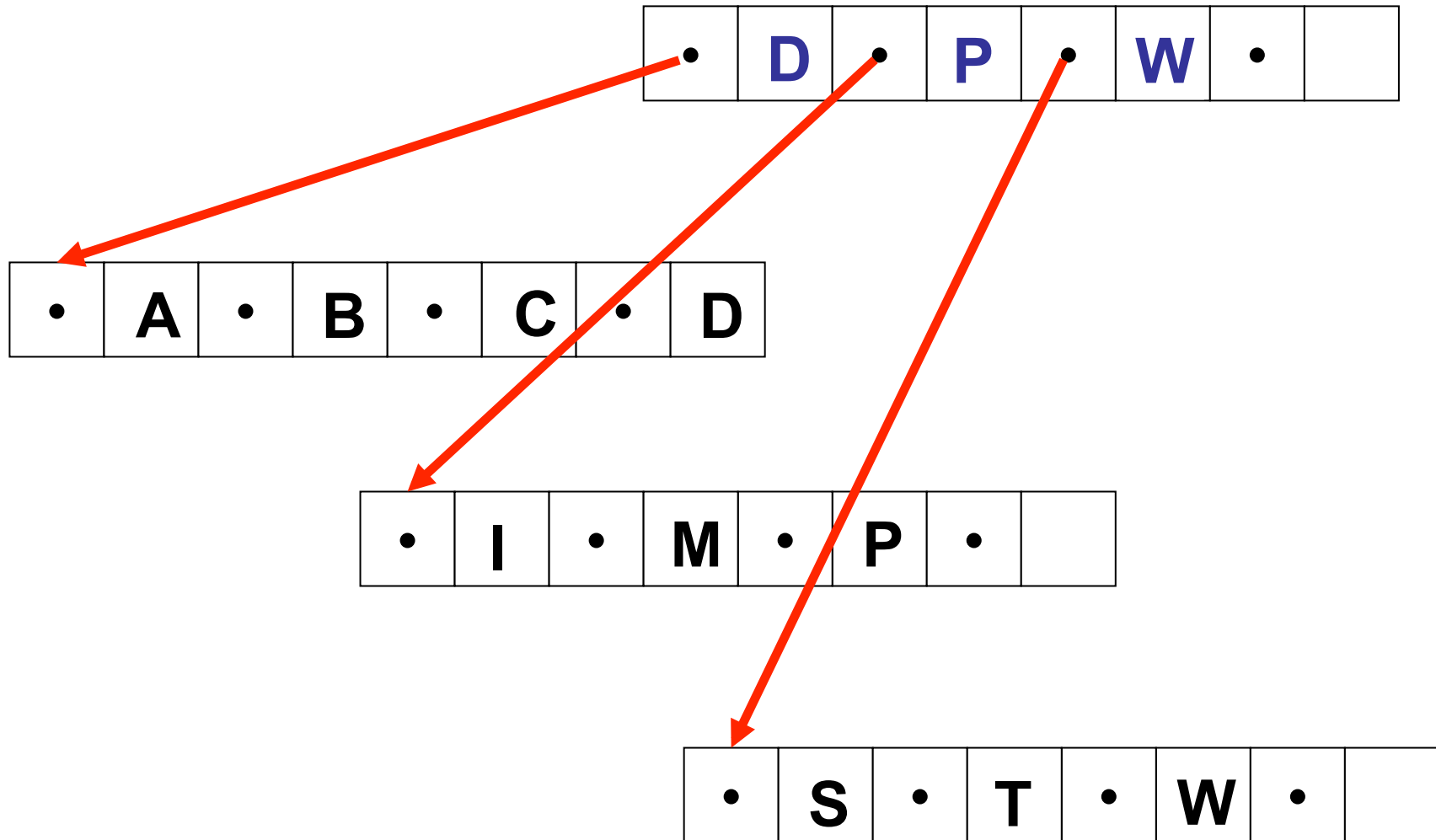
Insert P, I



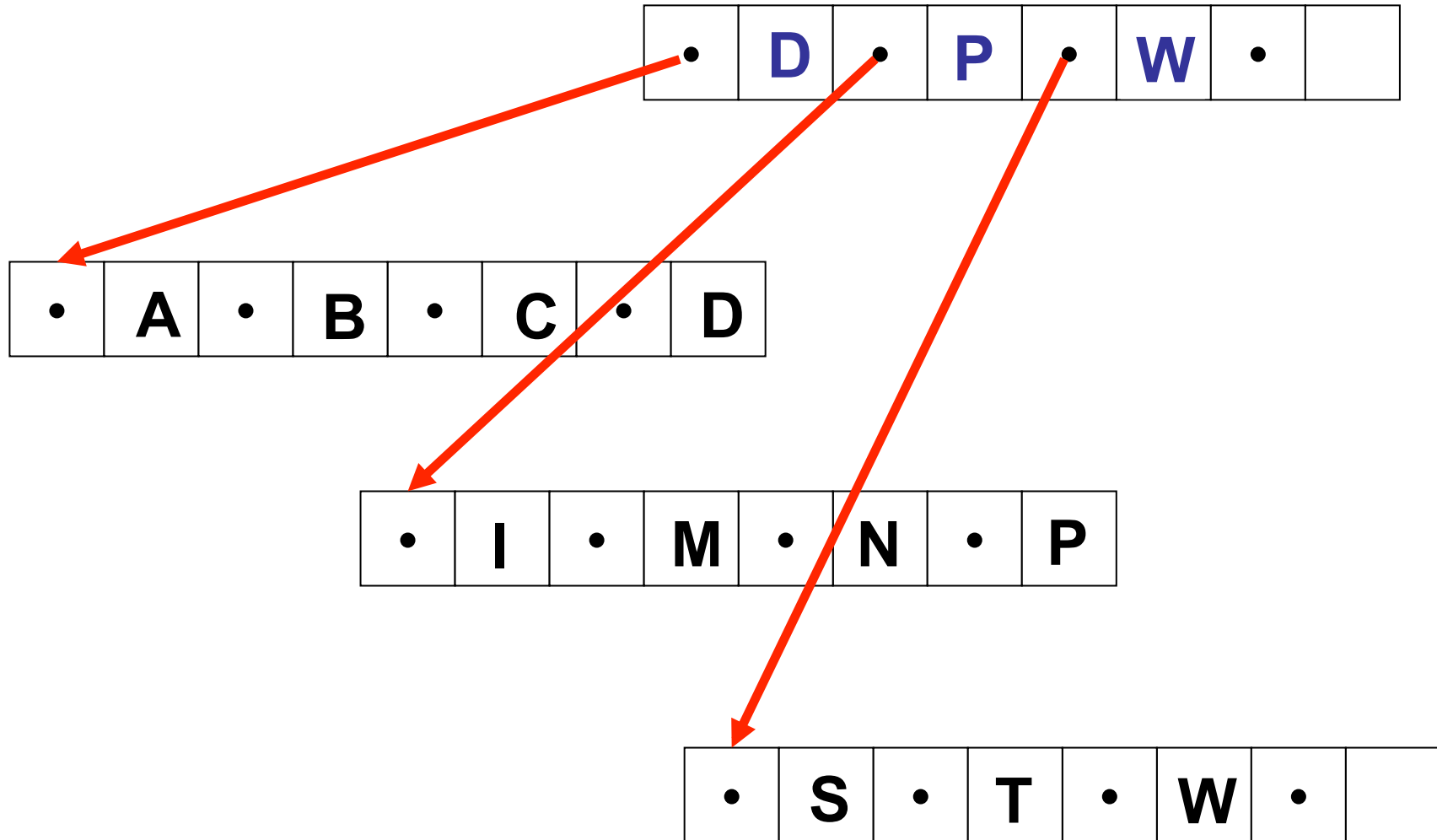
Insert I, B



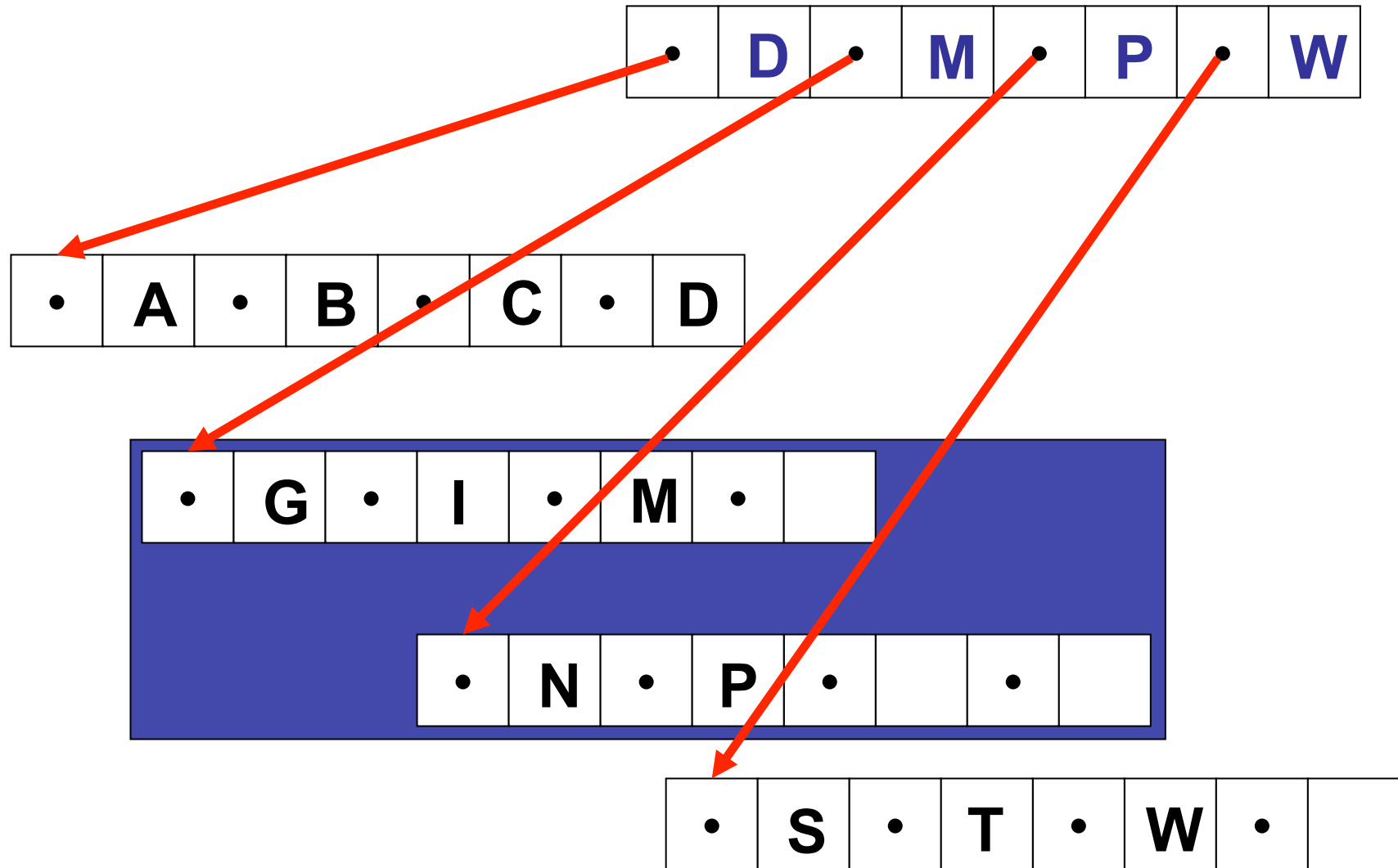
Insert B, W, N



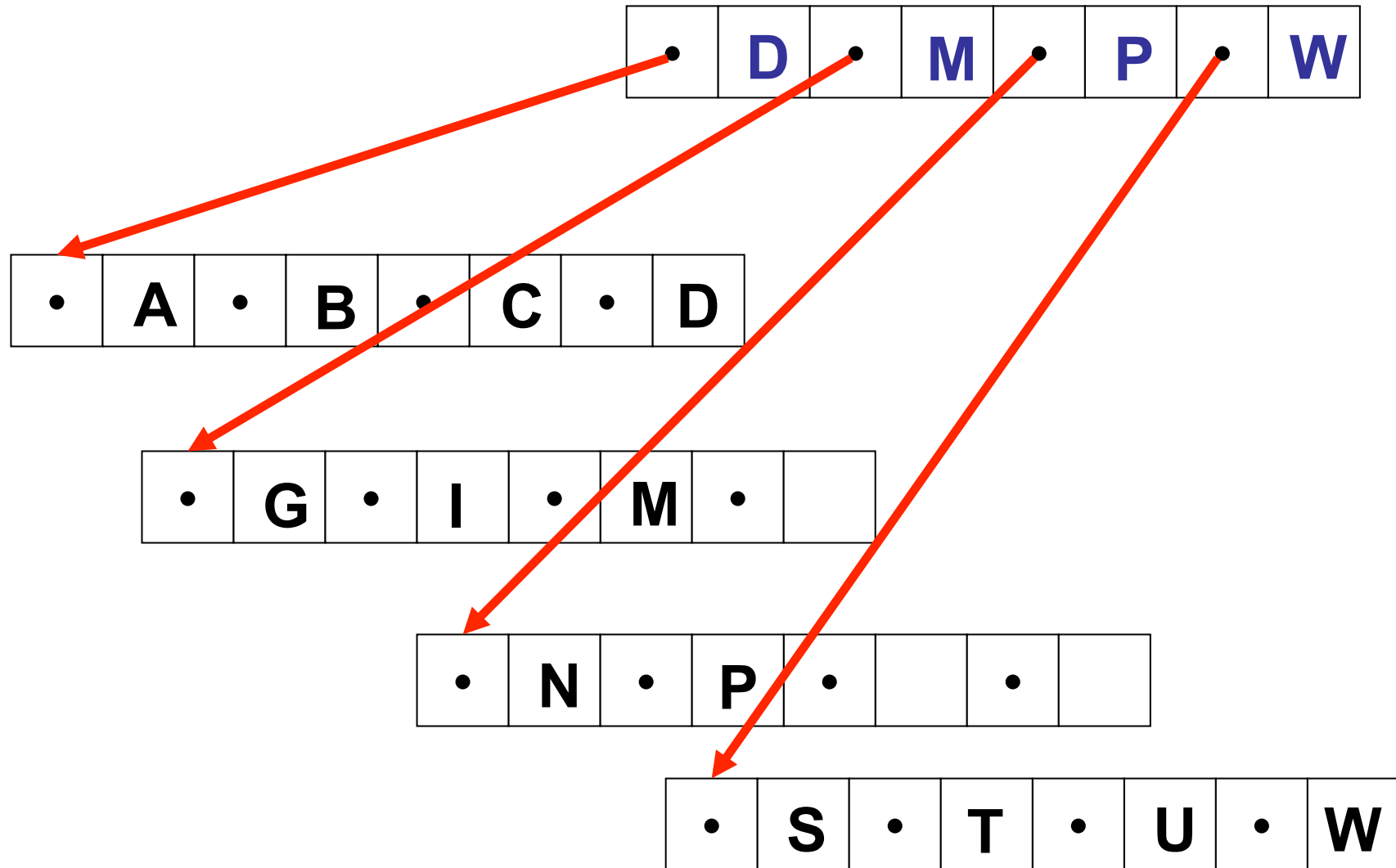
Insert N, G



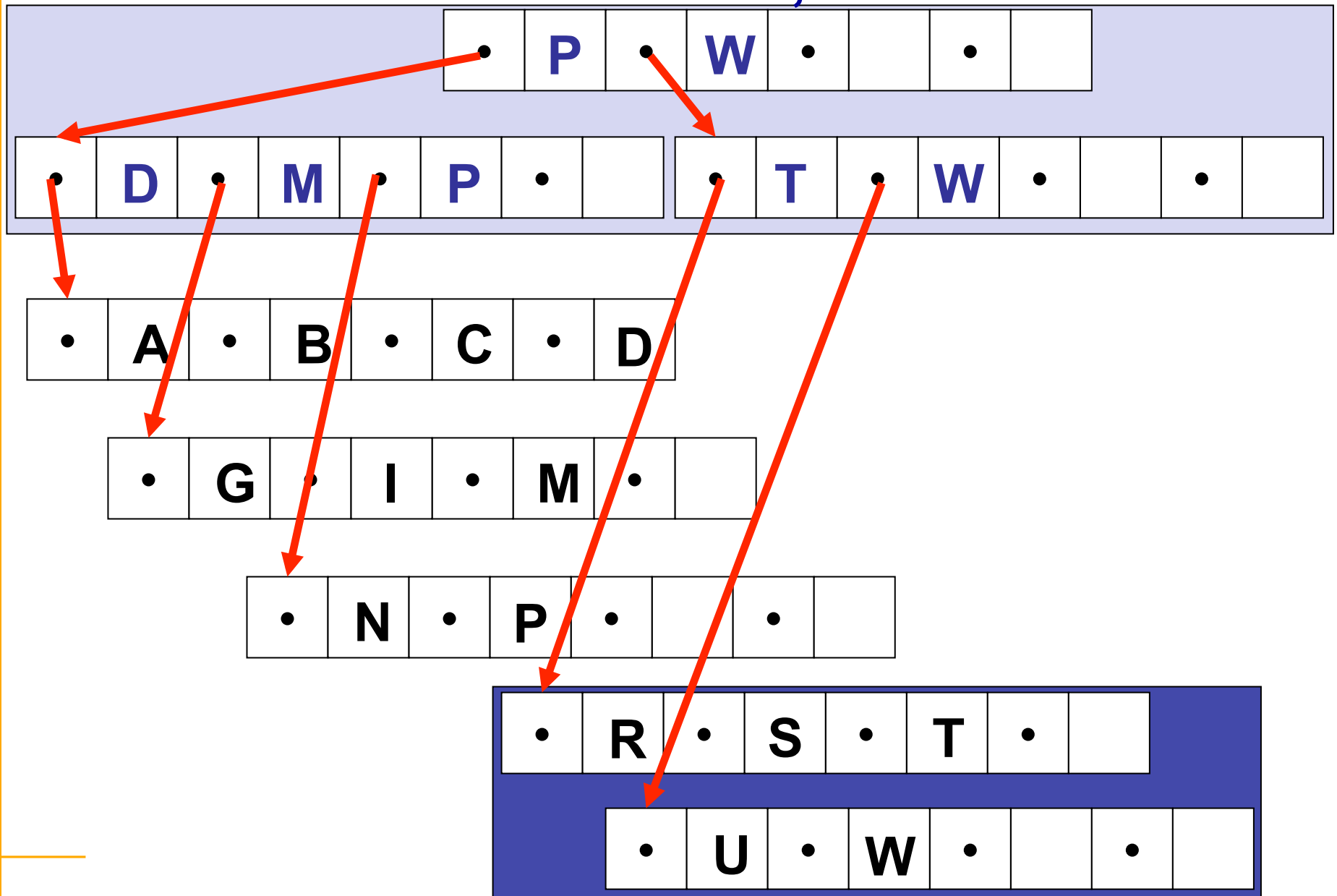
Insert G, U



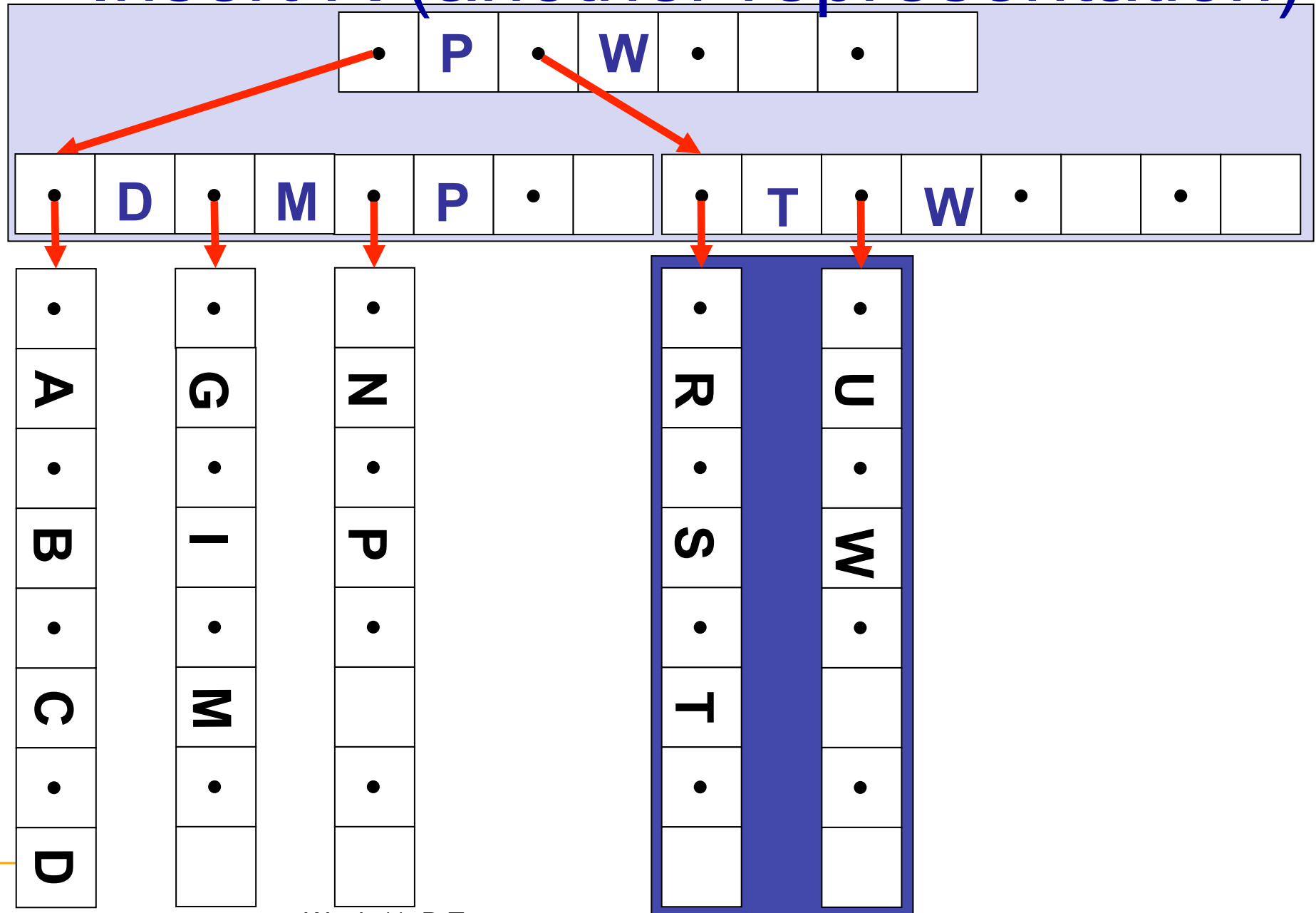
Insert U, R



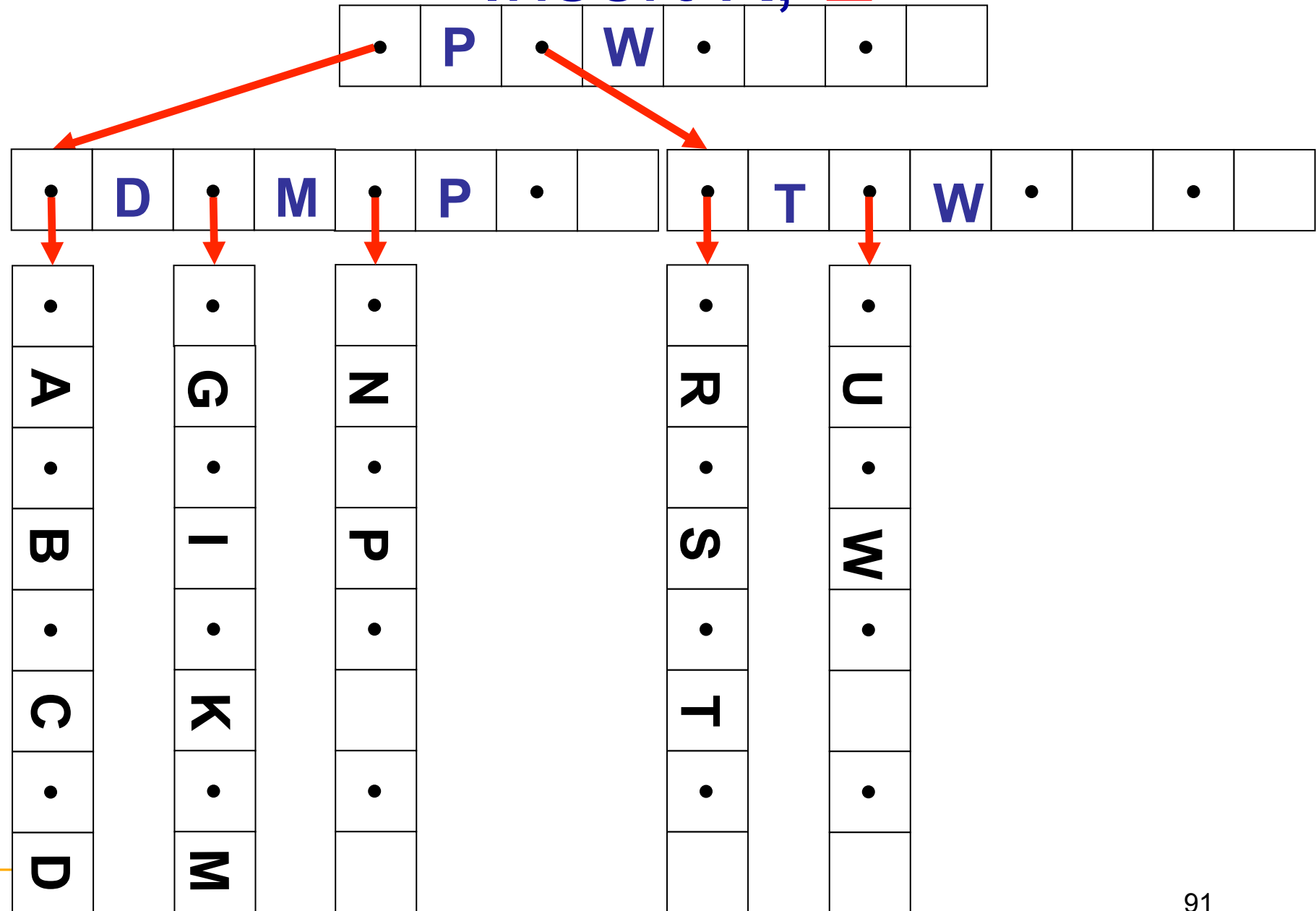
Insert R, K



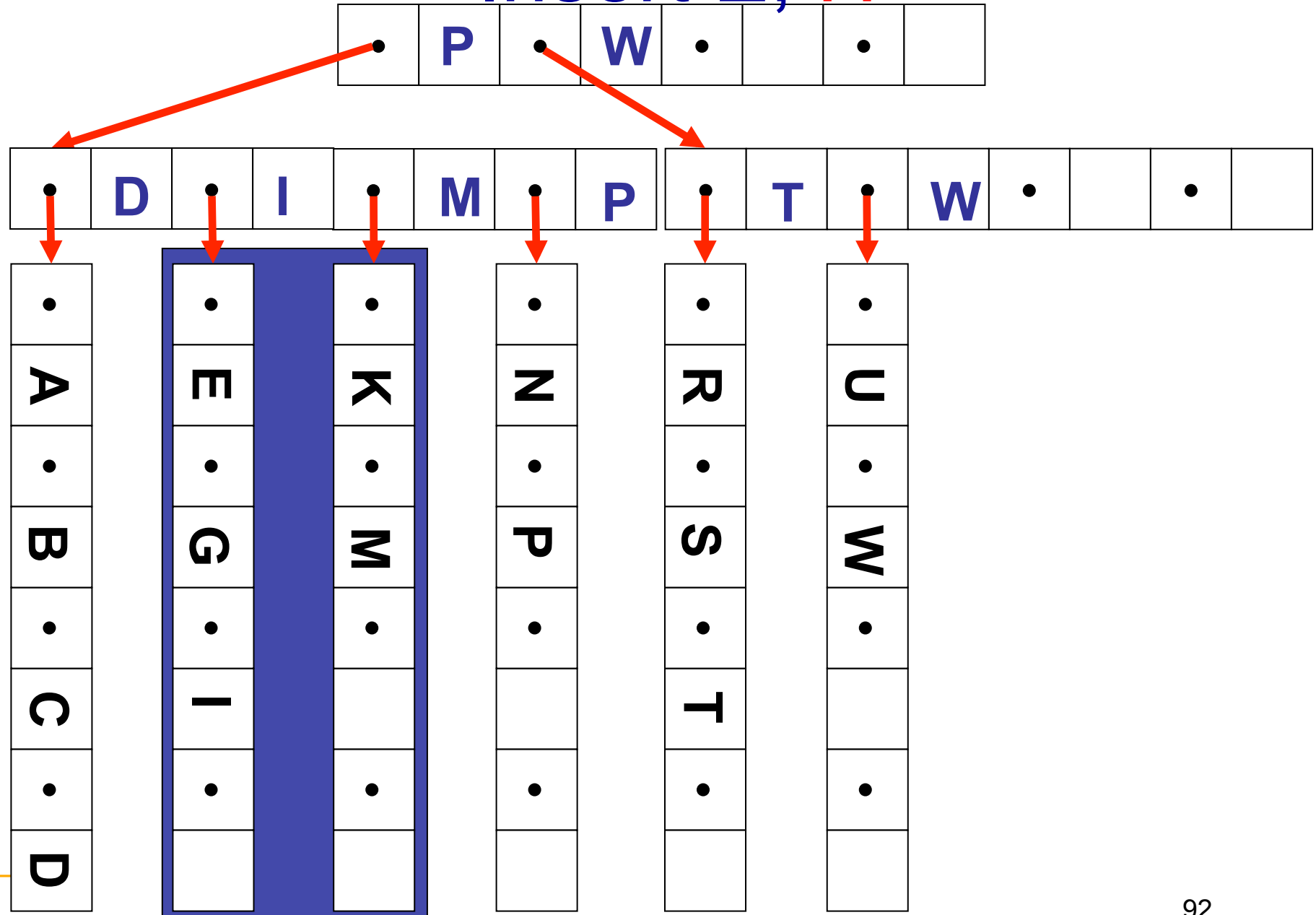
Insert R (another representation)



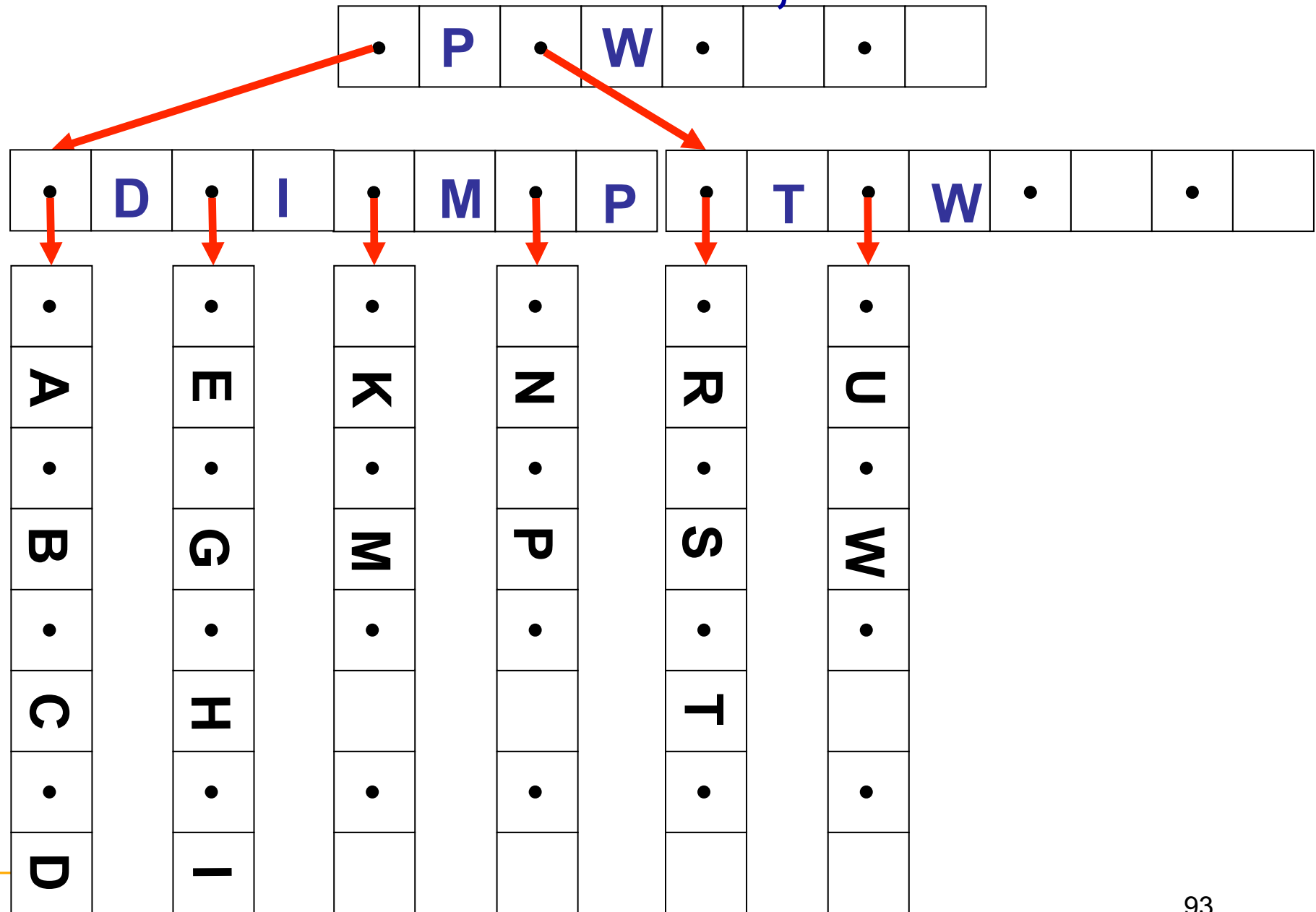
Insert K, E



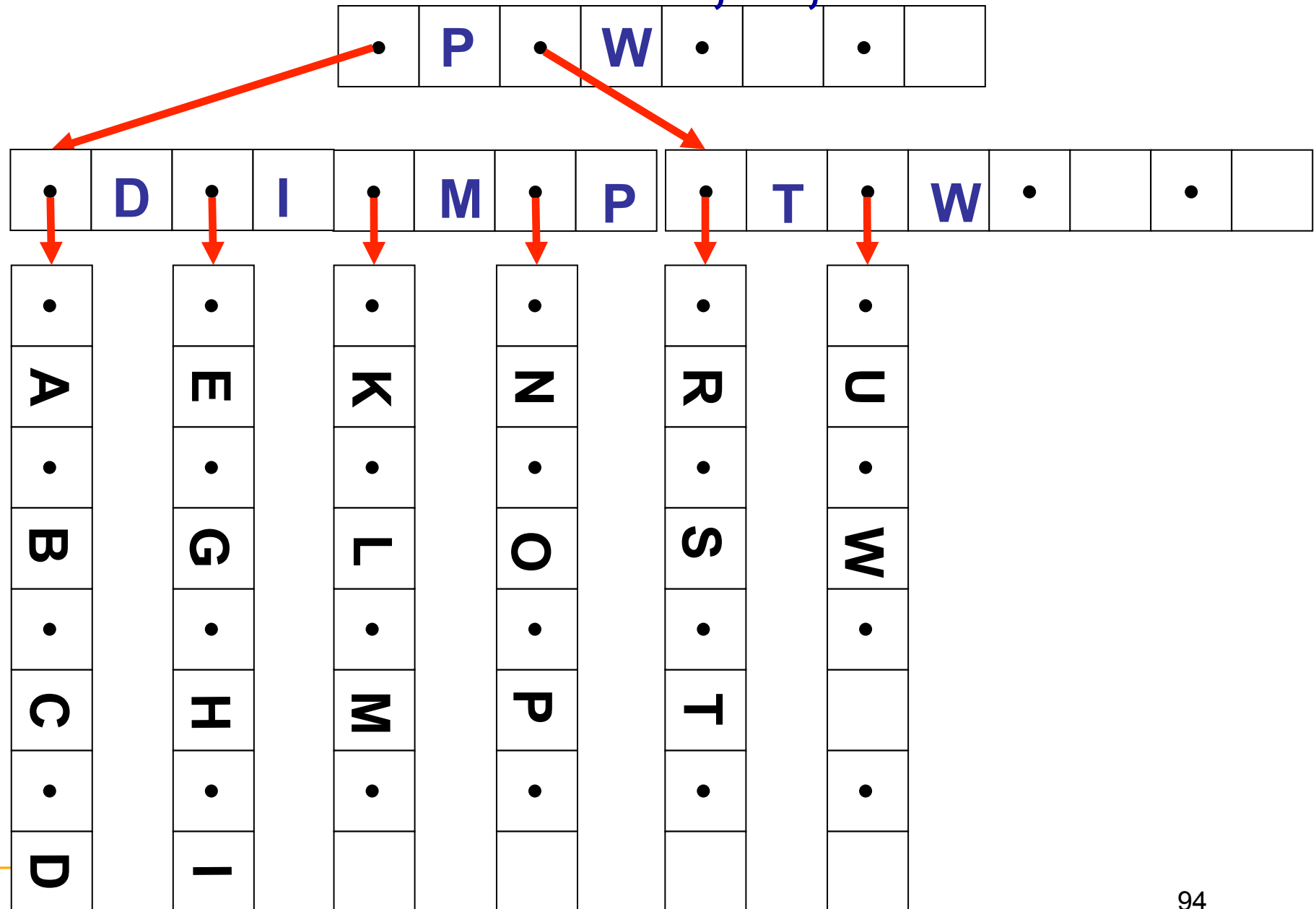
Insert E, H



Insert H, O



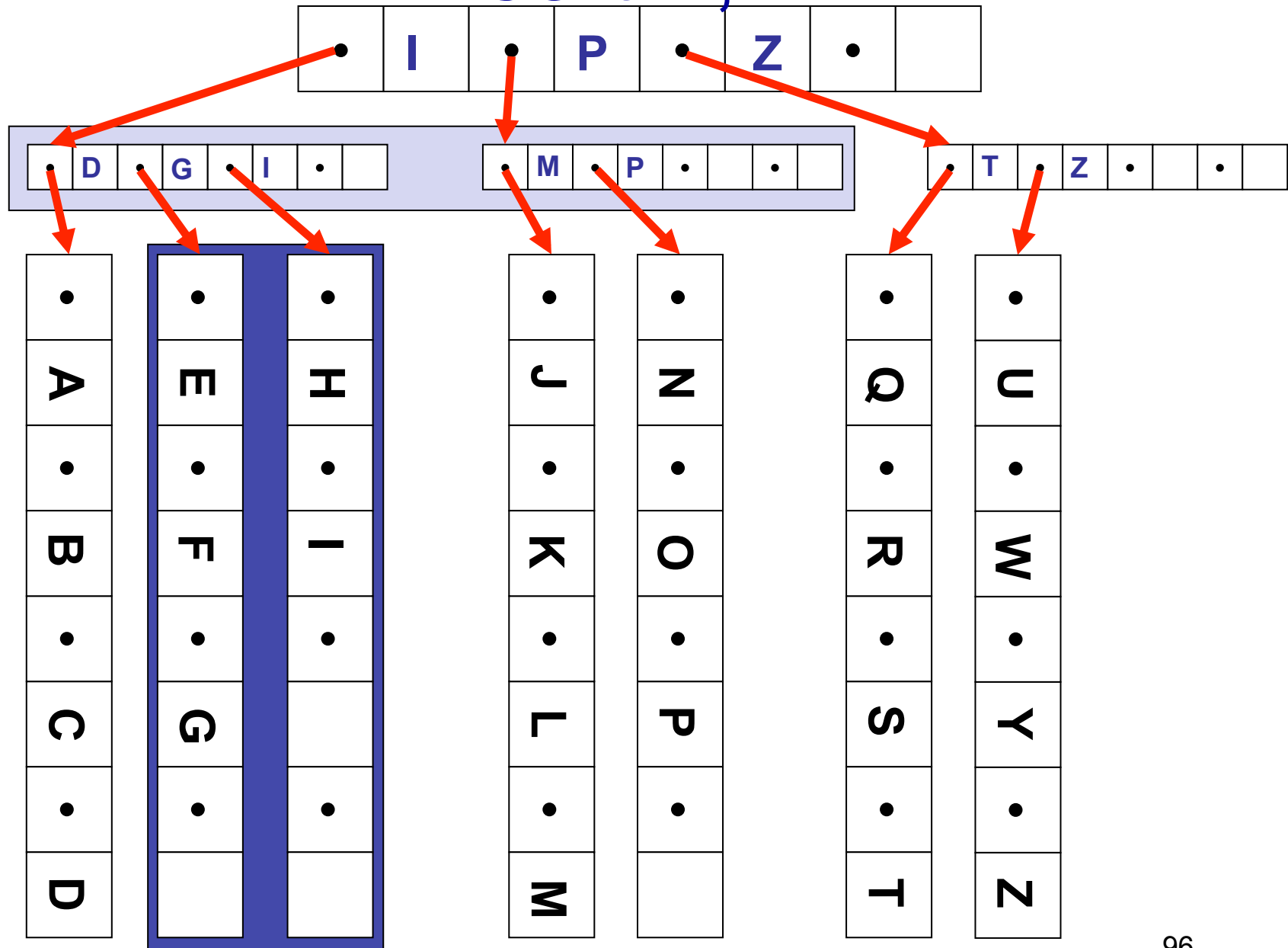
Insert O, L, J



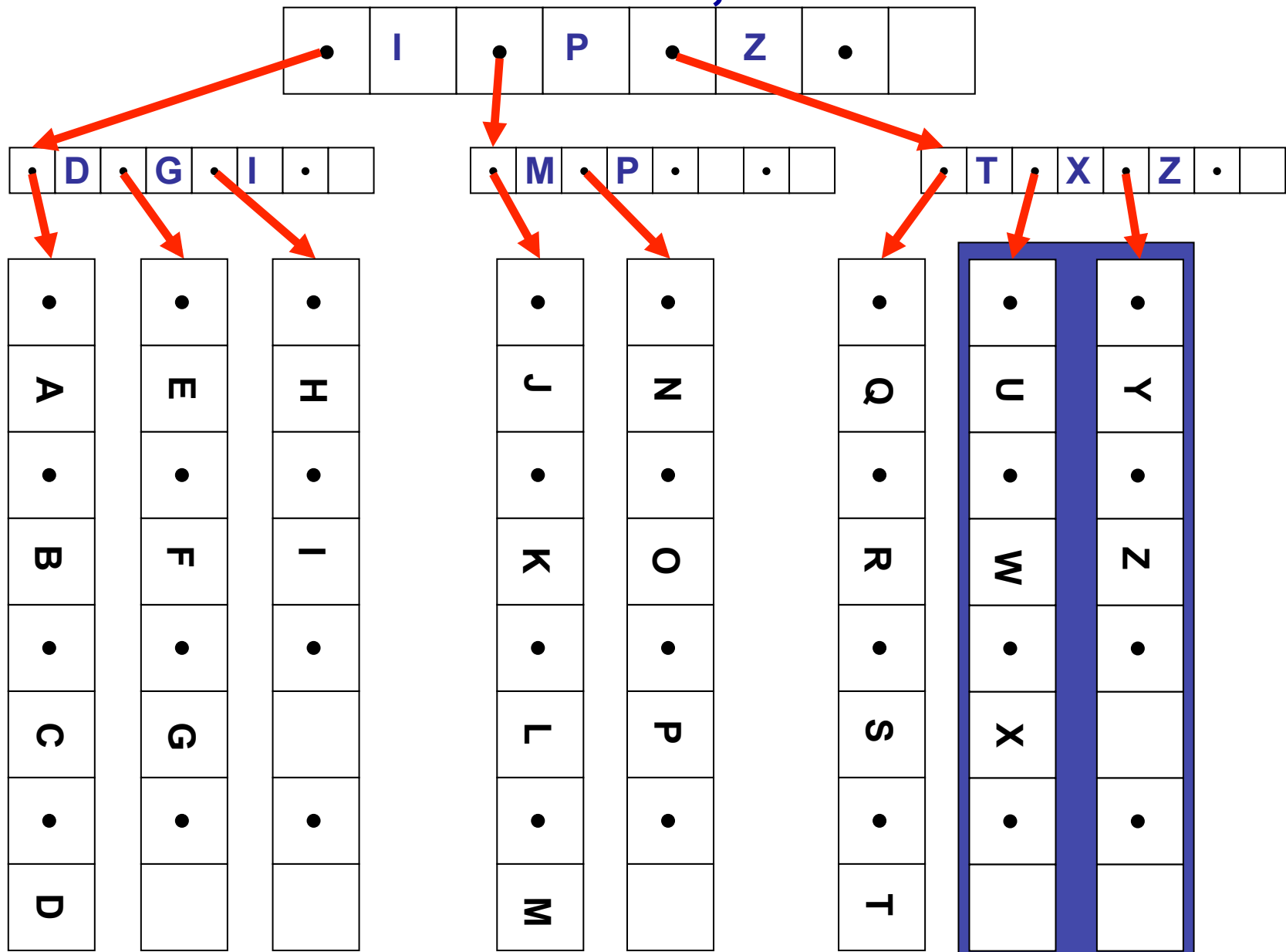
Insert J, Y, Q, Z, F



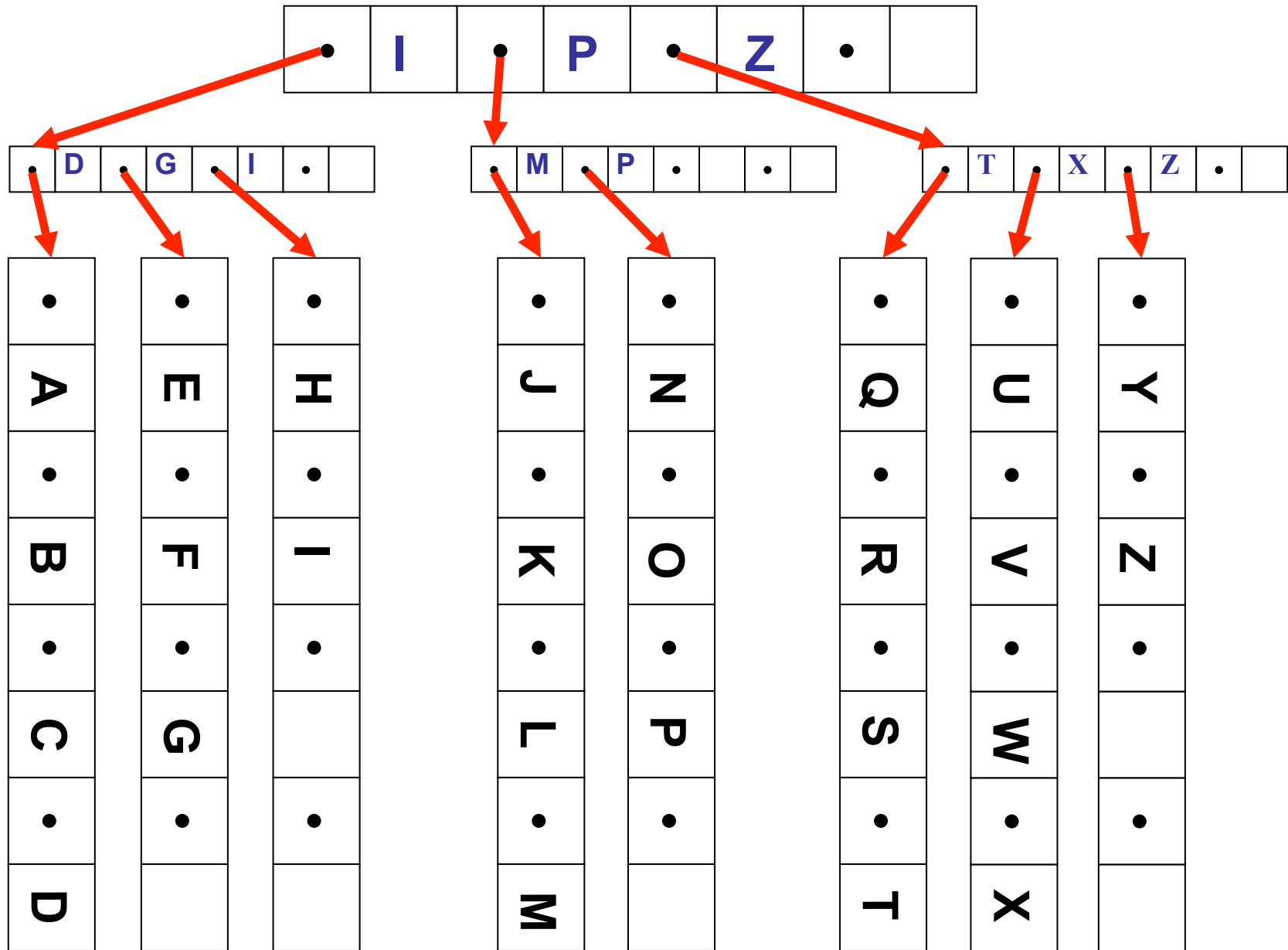
Insert F, X



Insert X, V



Insert V



B-Tree Revisited

- Linear cost of insertion and deletion
- Index records not to be fully occupied
- Does not shift record to another node but splits
- Some variations on B Trees:
 - B* Trees
 - B+ Trees

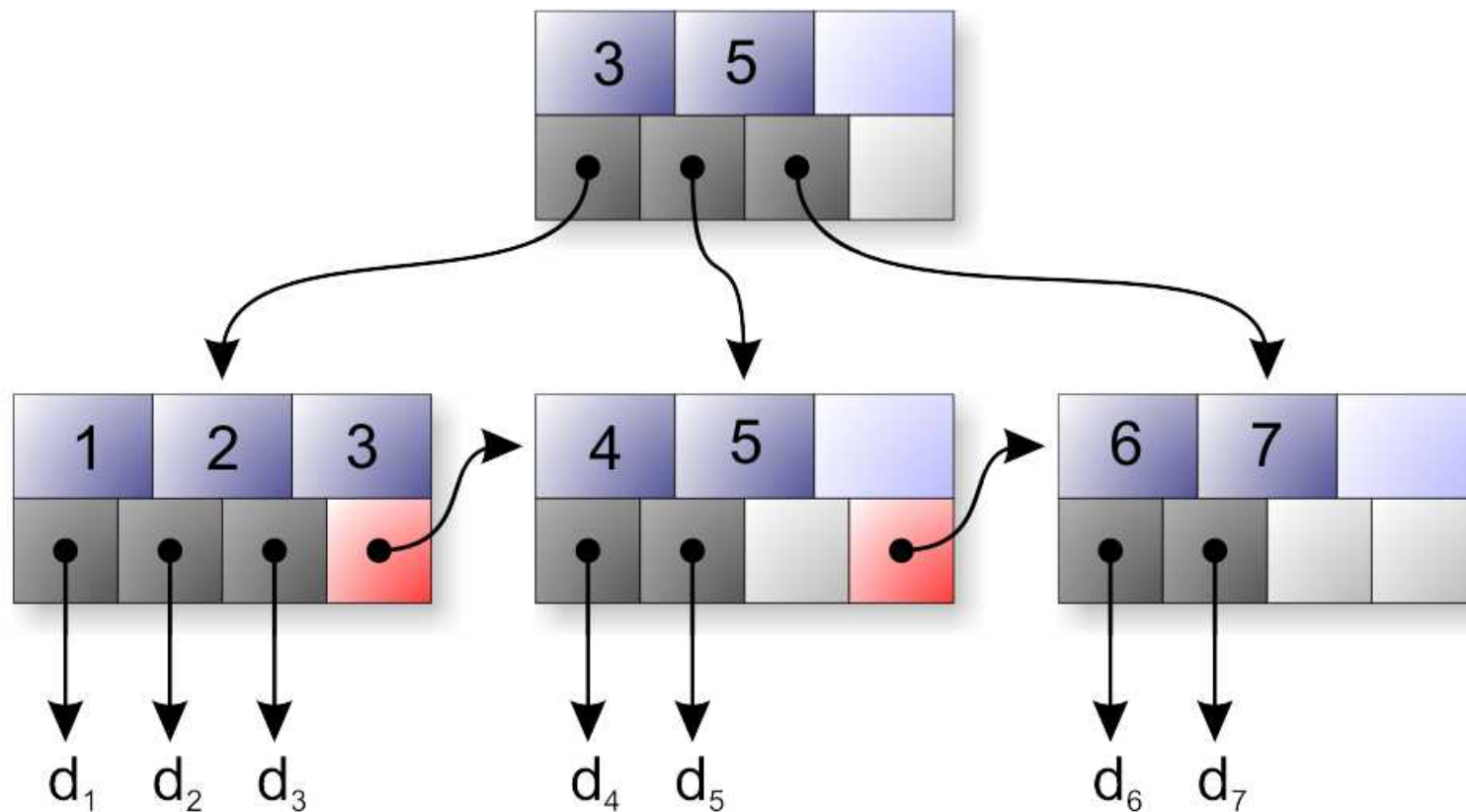
B*-Trees

- A variant of B-Tree
- Essential approach is to delay a split as long as possible
- Two thirds of a node (as opposed to $1/2$) has to be full to split (except the root)
 - If sibling is not full then redistribute
 - If sibling is full then split
 - Split into three, not two

B⁺-Trees

- B+ tree of order m consists of a root, internal nodes and leaves
- The root may be either leaf or node with two or more children
- Internal nodes contain between m and $2m$ keys, and a node with k keys has $k + 1$ children
- Leaves are always on the same level
- If a leaf is a primary index, it consists of a bucket of records, sorted by search key
- If it is a secondary index, it will have many short records consisting of a key and a pointer to the actual record

B+ Tree Example



Linked list (red) allows rapid inorder traversal

Source: encyclopedia.thefreedictionary.com

Theoretical Results

- Robert Tarjan proved
 - amortized number of splits/merges for a B Tree is 2

Summary

- B-Tree
- B-Tree Search, Insert, Delete
- B* Tree, B+ Tree

References on B-Trees

- Cormen, Leiserson, Rivest, Stein, Ch. 18
- Folk, Zoellick, Riccardi, Ch.9