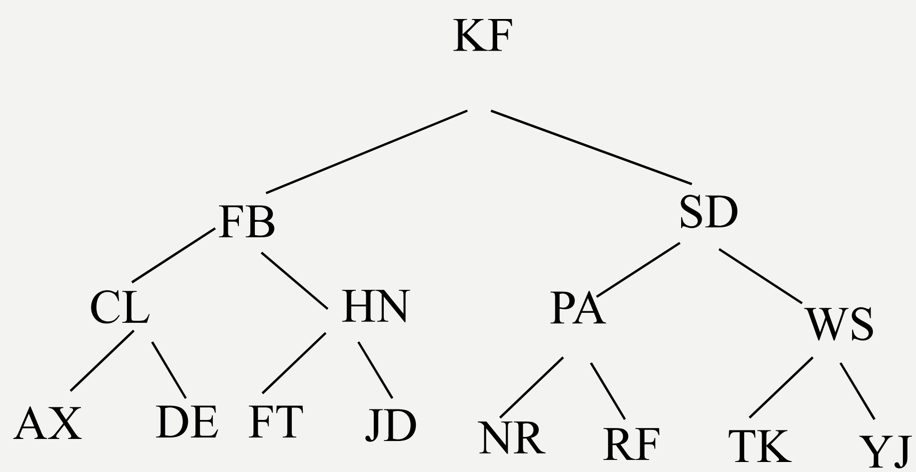
BINARY SEARCH TREE

- An example



- Node structure

Key	Left Child	Right Child

IMPLEMENTING A BINARY SEARCH TREE USING AN ARRAY:

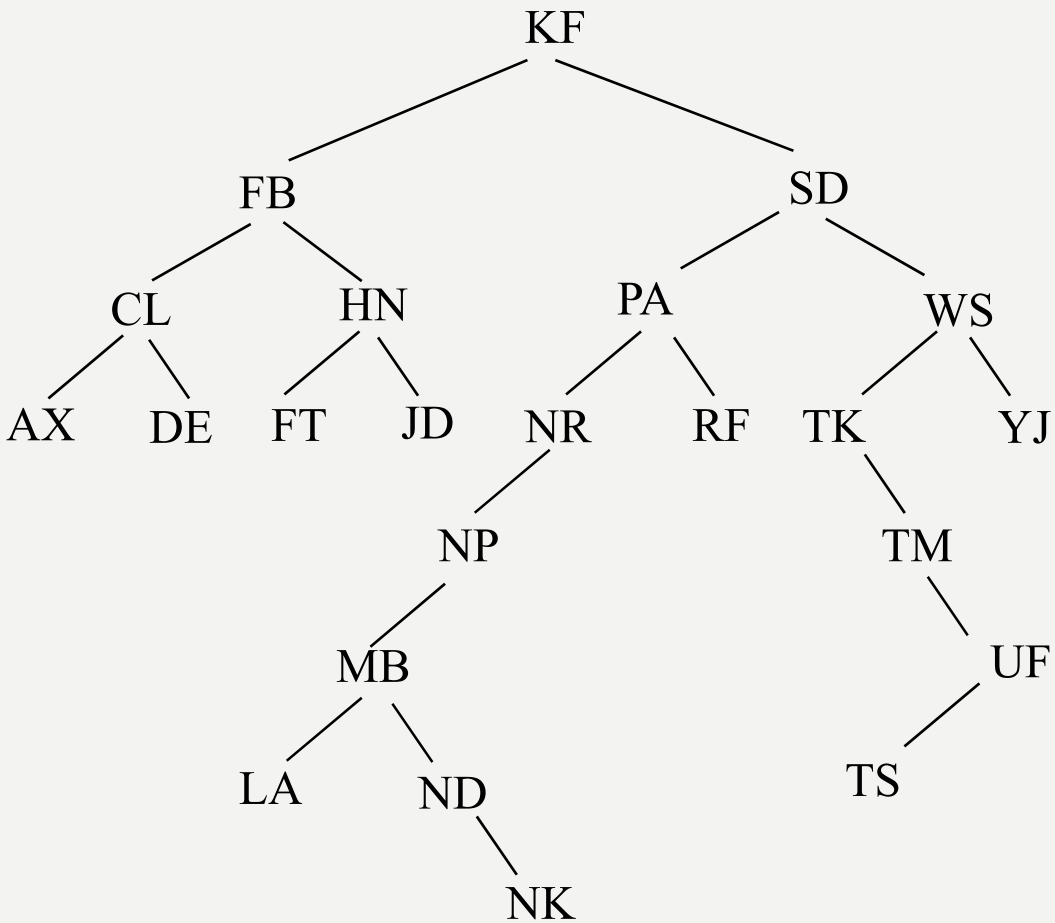
ROOT → 0

	Key	Left	Right
0	KF	2	1
1	SD	4	3
2	FB	5	6
3	WS	14	10
4	PA	11	9
5	CL	8	12
6	HN	7	13
7	FT	Λ	Λ

	Key	Left	Right
8	AX	Λ	Λ
9	RF	Λ	Λ
10	YJ	Λ	Λ
11	NR	Λ	Λ
12	DE	Λ	Λ
13	JD	Λ	Λ
14	TK	Λ	Λ

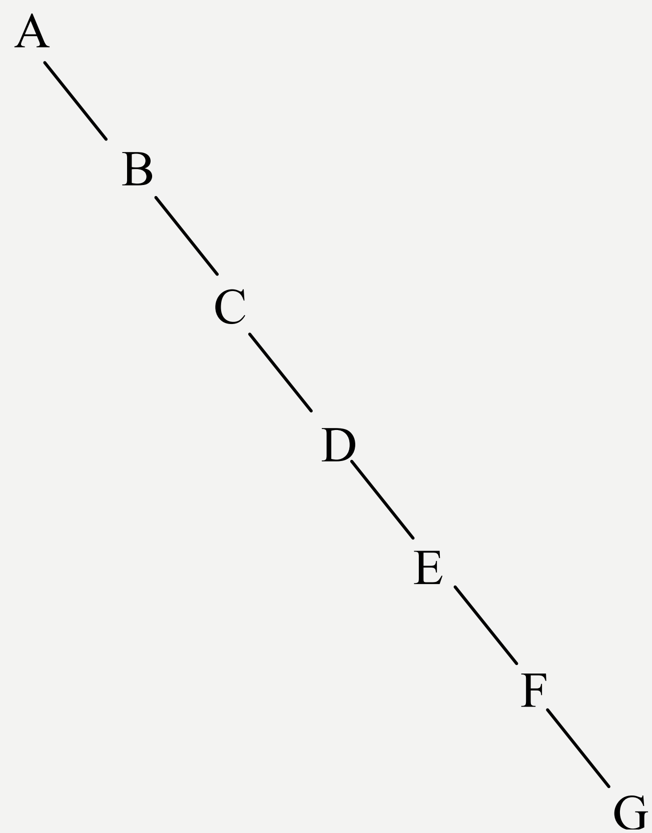
PROBLEM: UNBALANCED

- After inserting NP, MB, TM, LA, UF, ND, TS, and NK:



WORST CASE SCENARIO

- a binary search built by inserting A, B, C, D, E, F, G, in that order:



◆ A search based on this tree is essentially a sequential search!

B+-TREES

- Node structure

P_1	K_1	P_2	K_2	… …	P_q	K_q

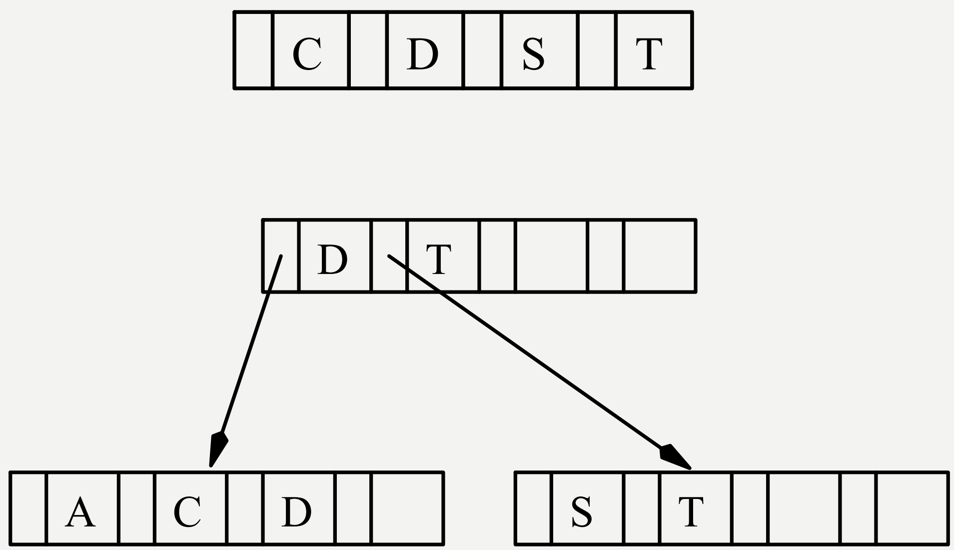
- P_i are pointers to the descendants and K_i are keys.
- $\lceil m/2 \rceil \leq q \leq m$.

Properties:

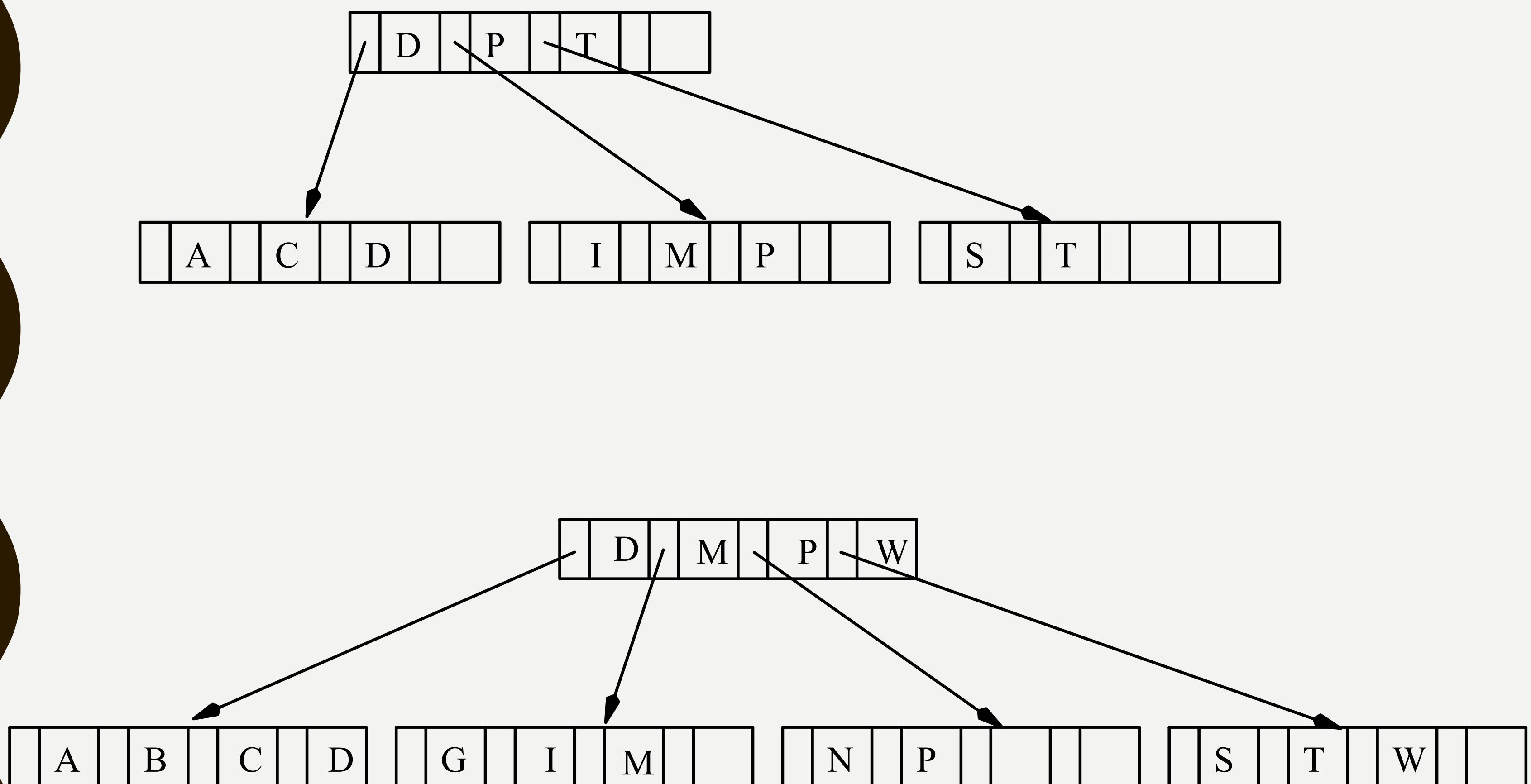
- Every page has a maximum of m descendants
- Every page, except for the root and the leaves, has at least $\lceil m/2 \rceil$ descendants
- The root has at least two descendants (unless it is a leaf)
- All the leaves appear on the same level
- The leaf level forms a complete, ordered index of the associated record file

B+-TREES

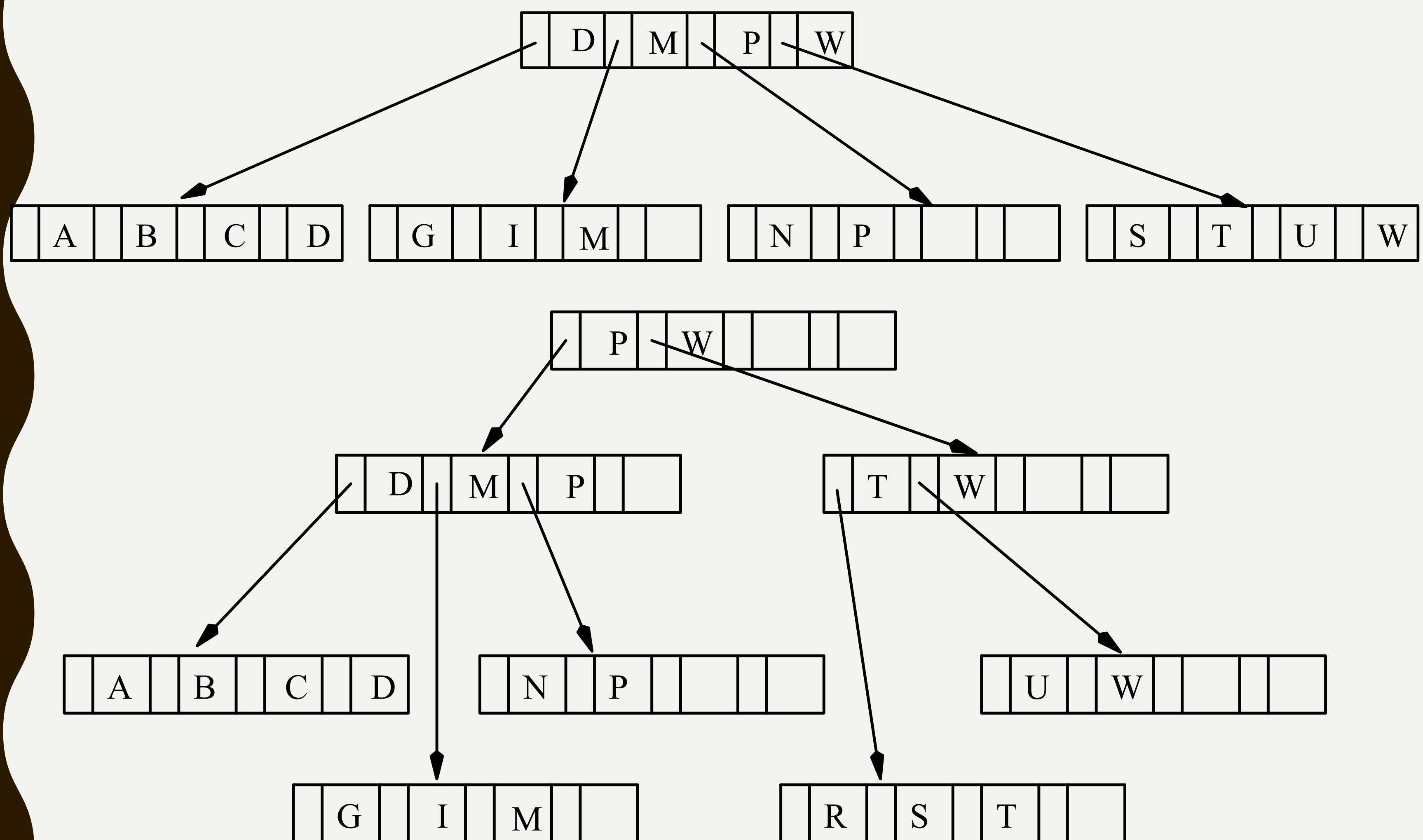
- Example: A B-tree of order 4 (m=4) built by inserting C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J, Y, Q, Z, F, X, V in that order.



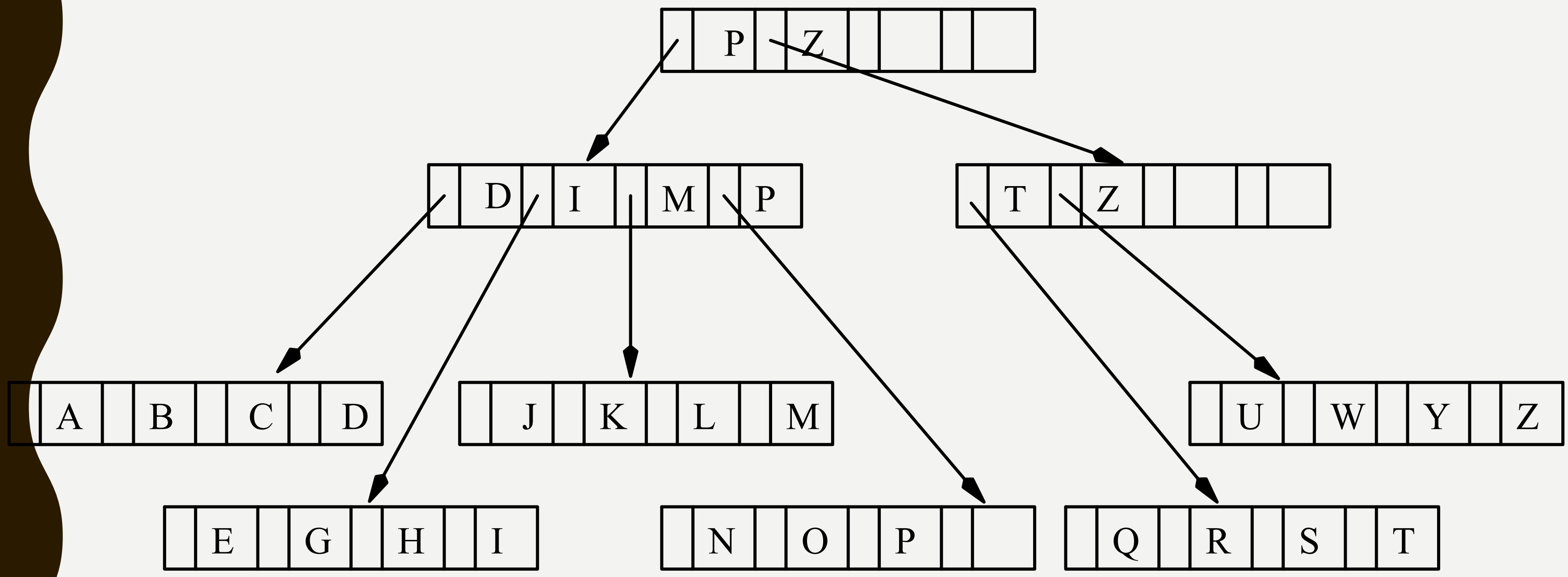
B+-TREES



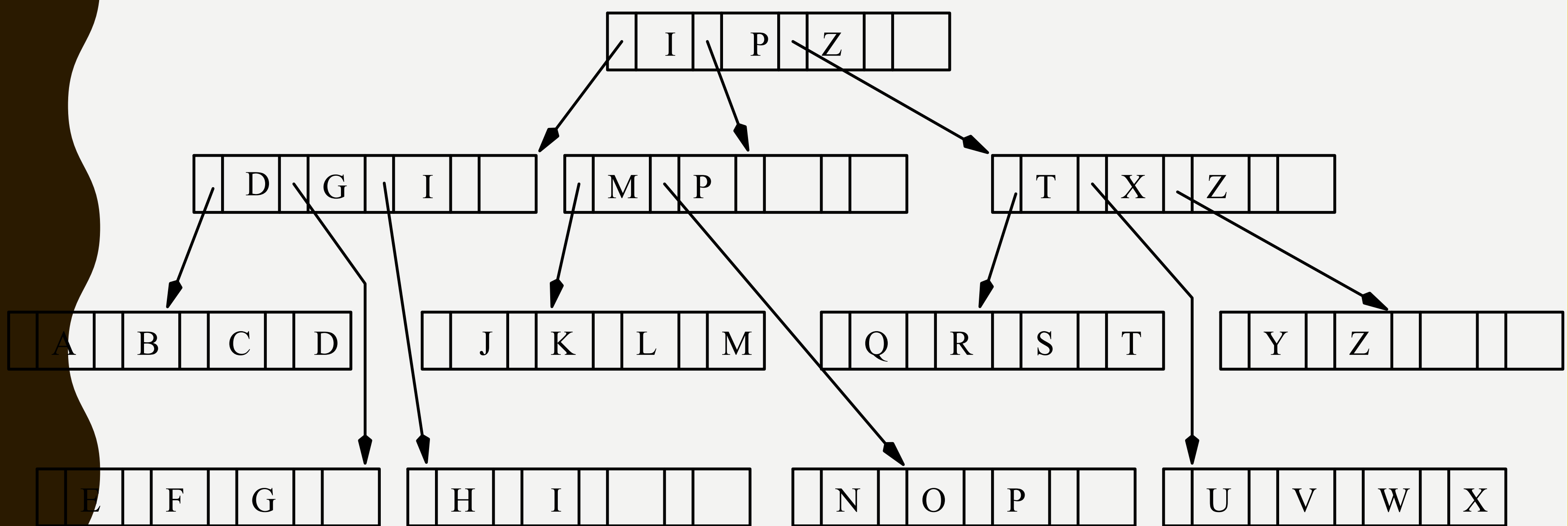
B+-TREES



B+-TREES



B+-TREES



B+-TREES PERFORMANCE

- The worst-case search depth:

level	minimum # of descendants
1	2
2	$2 * \lceil m/2 \rceil$
3	$2 * \lceil m/2 \rceil^2$
4	$2 * \lceil m/2 \rceil^3$
…	…
d	$2 * \lceil m/2 \rceil^{d-1}$

- For a B-tree of N keys, $N \geq 2 * \lceil m/2 \rceil^{d-1}$ or $d \leq 1 + \log_{\lceil m/2 \rceil}(N/2)$.
- With N = 1, 000, 000 and m = 512, $d \leq 3.37$.
- Notice that this formula is similar to that of Paged Binary Search trees.

SUMMARY OF OPERATIONS ON B+-TREES

- Search for key *K*

  Start from the root repeat the following

  Let *N* be the current node;

  if *N* is a leaf then

  if *K* is in *N*, return the data pointer;

  else report *K* not found and return;

  else if (K <= K1)

  look for *K* in descendant P_1 ;

  else if ($K > K_i$ && $K <= K_{i+1}$)

  look for *K* in descendant P_{i+1} ;

SUMMARY OF OPERATIONS ON B+-TREES

- Insert key *K*

Search for key *K*;

if found report key *K* exists and return;

else let *L* be the leaf node at the end of the search;

if *L* is not full, insert *K* to *L*;

else split the node to two and inert them to the parent node, if the parent node also overflows, split this node

too; propagate the split up until overflows do not occur or a new root is created;

if the insertion causes the largest key in a leaf to change,

modify the upper level(s) to reflect the change.

SUMMARY OF OPERATIONS ON B+-TREES

- Delete key *K*

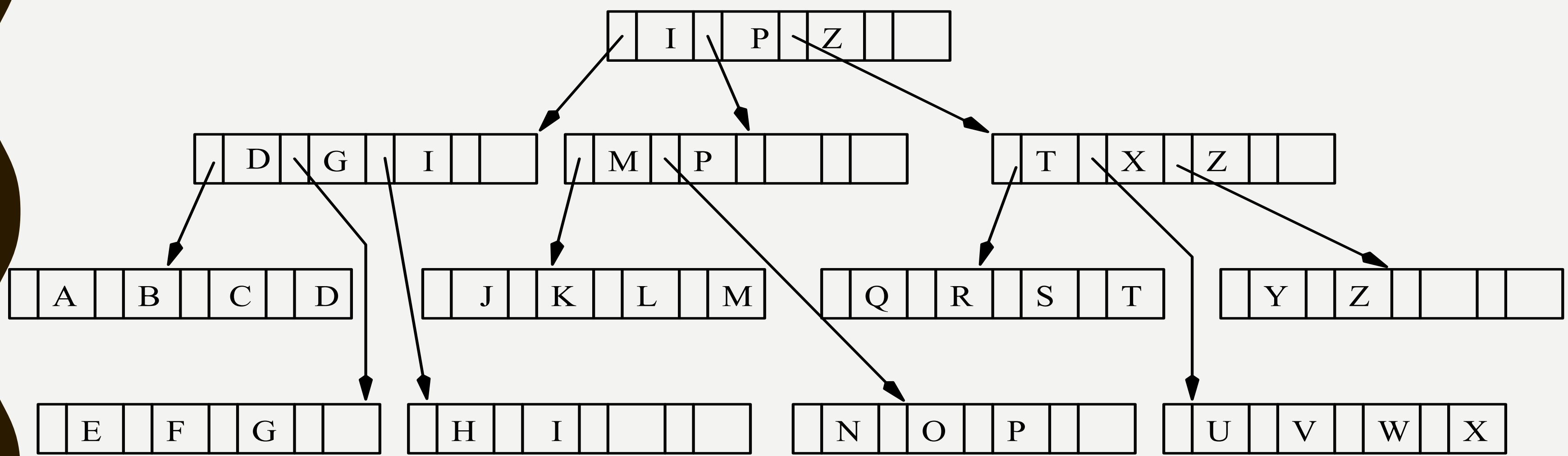
Search for key *K*;

If no-found return;

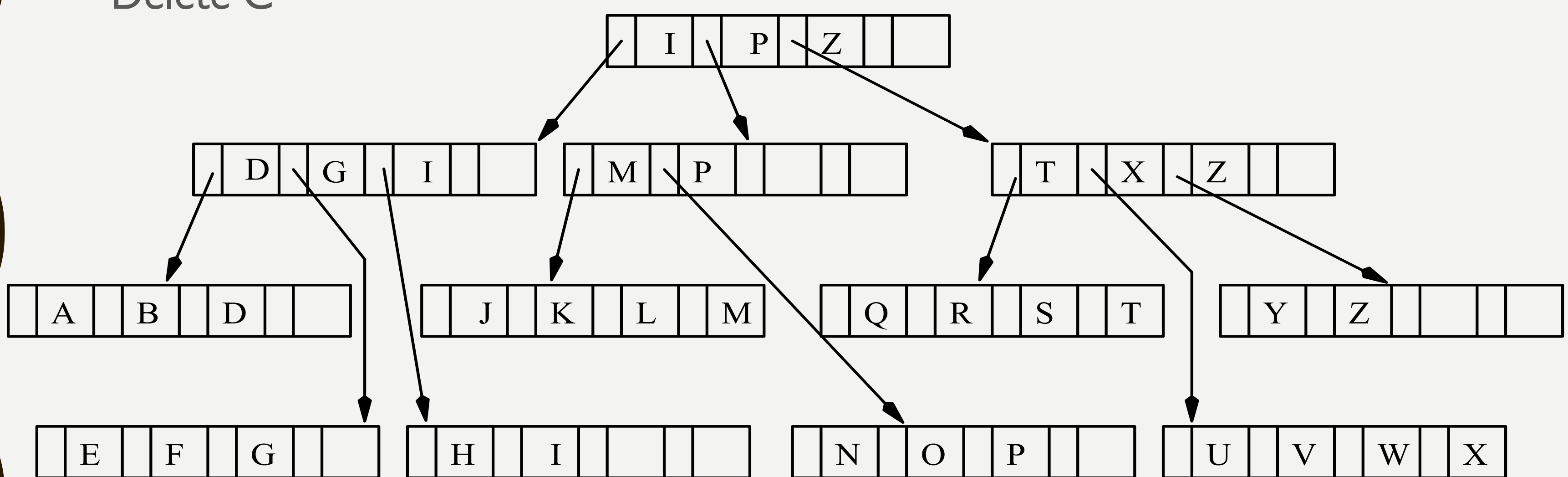
else let *L* be the leaf node at the end of the search;

1) if *L* has at least $\lceil m/2 \rceil + 1$ keys and *K* is not the largest, simply delete *K*;
2) if *L* has at least $\lceil m/2 \rceil + 1$ keys and *K* is the largest, delete *K* and modify the upper level indexes to reflect the new largest key in *L*;
3) if *L* has exactly $\lceil m/2 \rceil$ keys, and one of the siblings of *L* has less than $\lceil m/2 \rceil + 1$ keys, merge *L* with this sibling and delete one key from the parent node;
4) if *L* has exactly $\lceil m/2 \rceil$ keys, and its siblings all have at least $\lceil m/2 \rceil + 1$ keys, redistribute some keys from one of its sibling to *L*, and modify the upper level nodes if necessary.

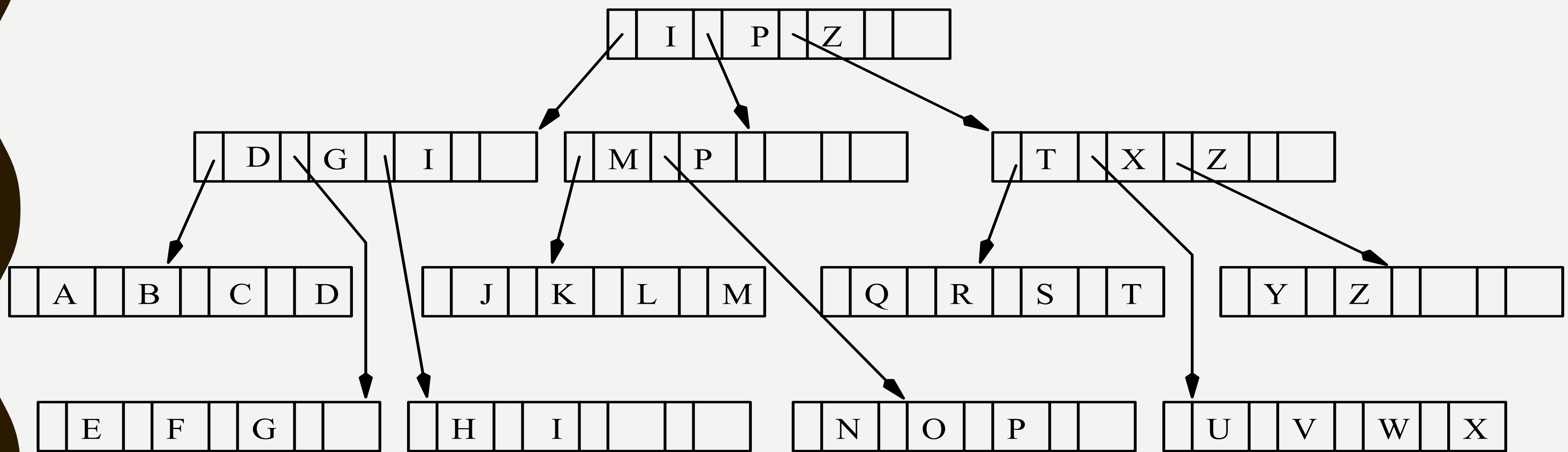
DELETIONS ON B+-TREES



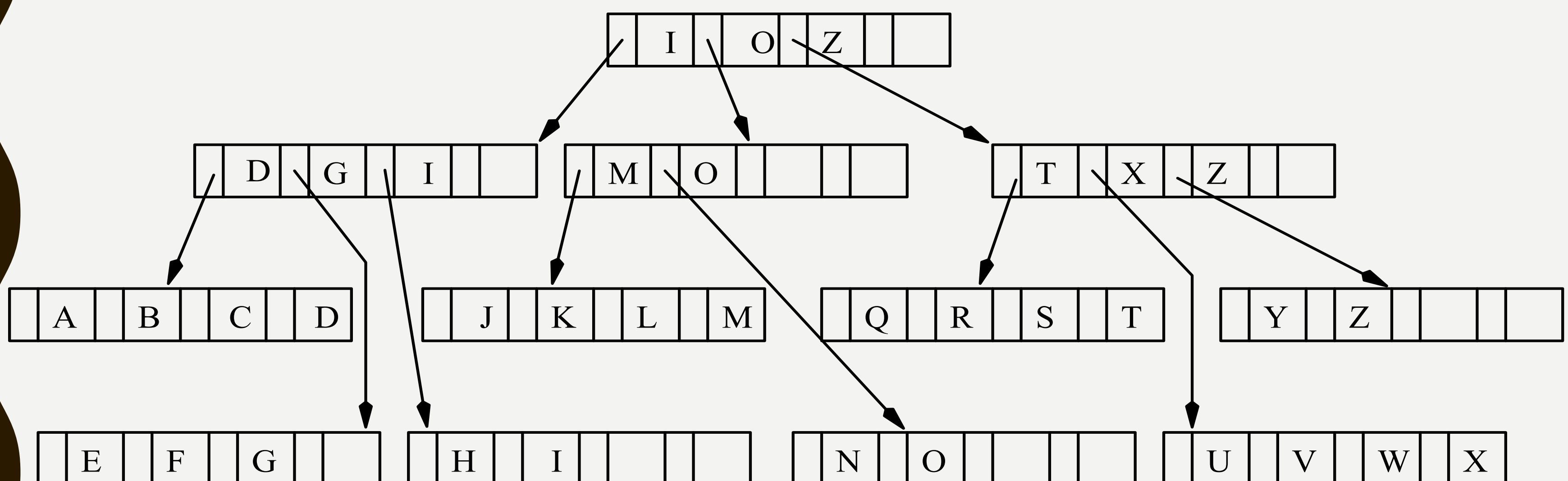
- Delete C



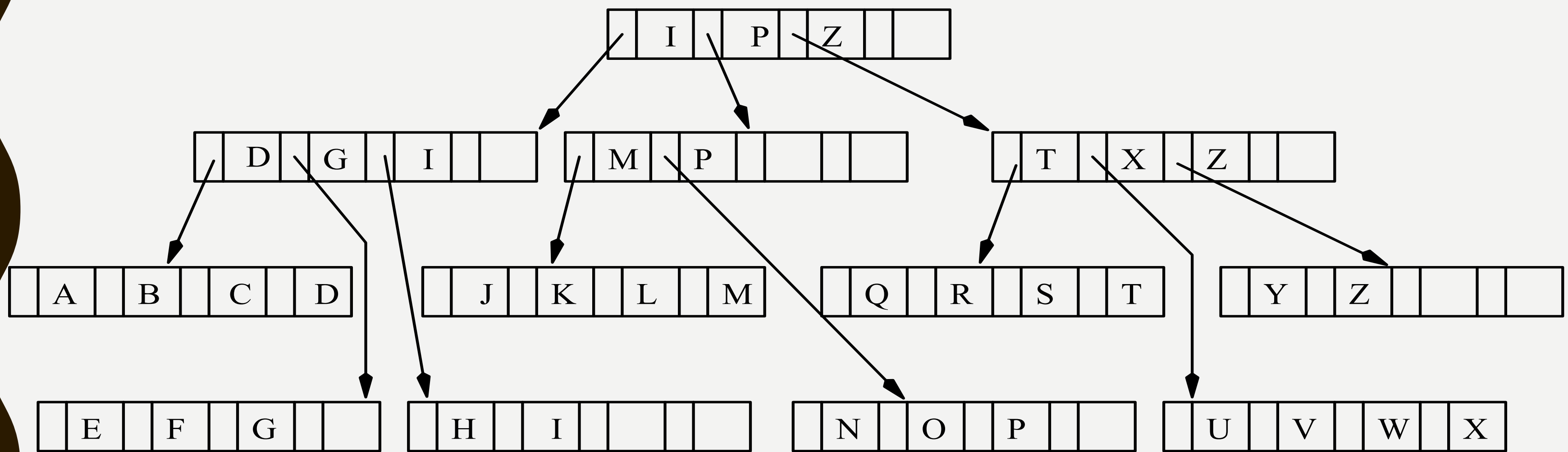
DELETIONS ON B+-TREES



- Delete P



DELETIONS ON B+-TREES



- Delete H