File Structures

Ch09. A. Multilevel Indexing and B-Trees

2020. Spring

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

Slides are based on the textbook.

datalab
data science laboratory

Outline

- 9.1 The invention of the B-tree
- 9.2 Statement of the Problem
- 9.3 Indexing with Binary Search Trees
- 9.4 Multilevel Index

The invention of the B-tree

- B-tree
  - de facto, the standard organization for indexes in a database system
  - to solve how to access and efficiently maintain an index that is too large to hold in memory
  - the notion of a paged index with retrieval time proportional to $log_k I$ (I : index size, k : page size)
  - why the name B-tree?
    - B : balanced, broad, bushy, Bayer, Boeing

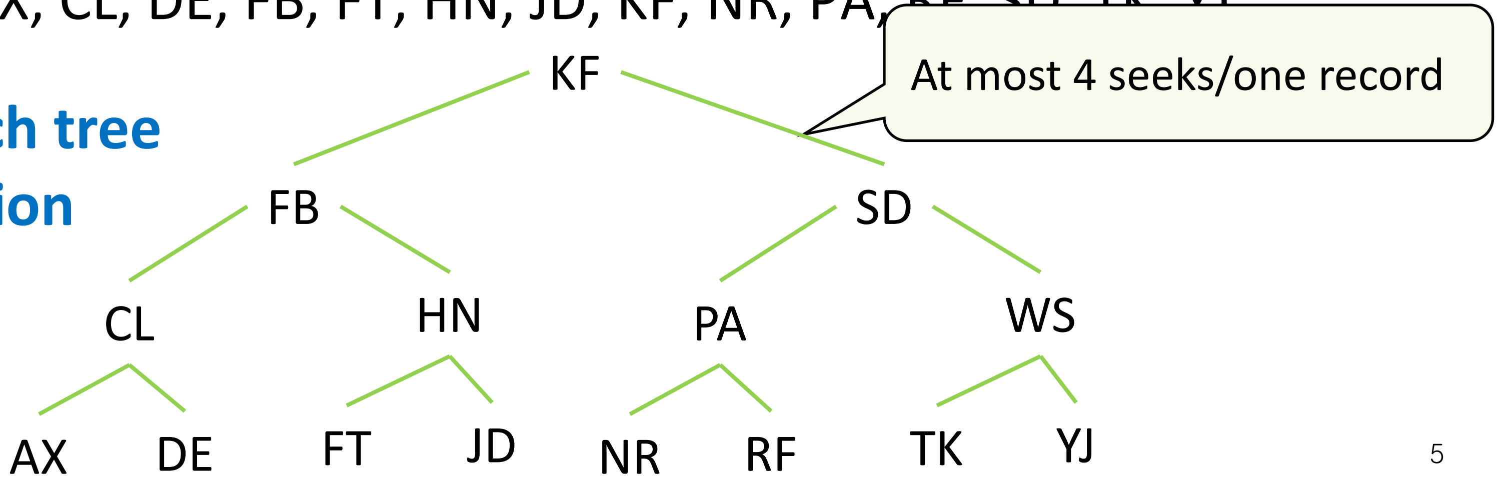
Statement of the Problem

- Two problems !!!
  - searching the index must be faster than binary searching
    - Binary searching requires to many seeks
    - searching for a key on a disk involves seeking to different disk tracks
  - insertion and deletion must be as fast as search
    - Inserting a key into an index involves moving a large # of other keys in the index
      - Why? Keep the index in sort order
      - → index maintenance is very impractical on disk
    - need to find a way to make insertions and deletions that have only local effects in the index

Indexing with Binary Search Trees (1/4)

- Binary search tree
  - BSTs keep their keys in sorted order
    - lookup and other operations can use the principle of binary search
  - Example: Sorted list of keys
    - AX, CL, DE, FB, FT, HN, JD, KF, NR, PA, RF, SD, TK, YJ

Binary search tree representation



Indexing with Binary Search Trees (2/4)

- Internal Representation of Binary Tree
  - With RRN(fixed length record) or pointer

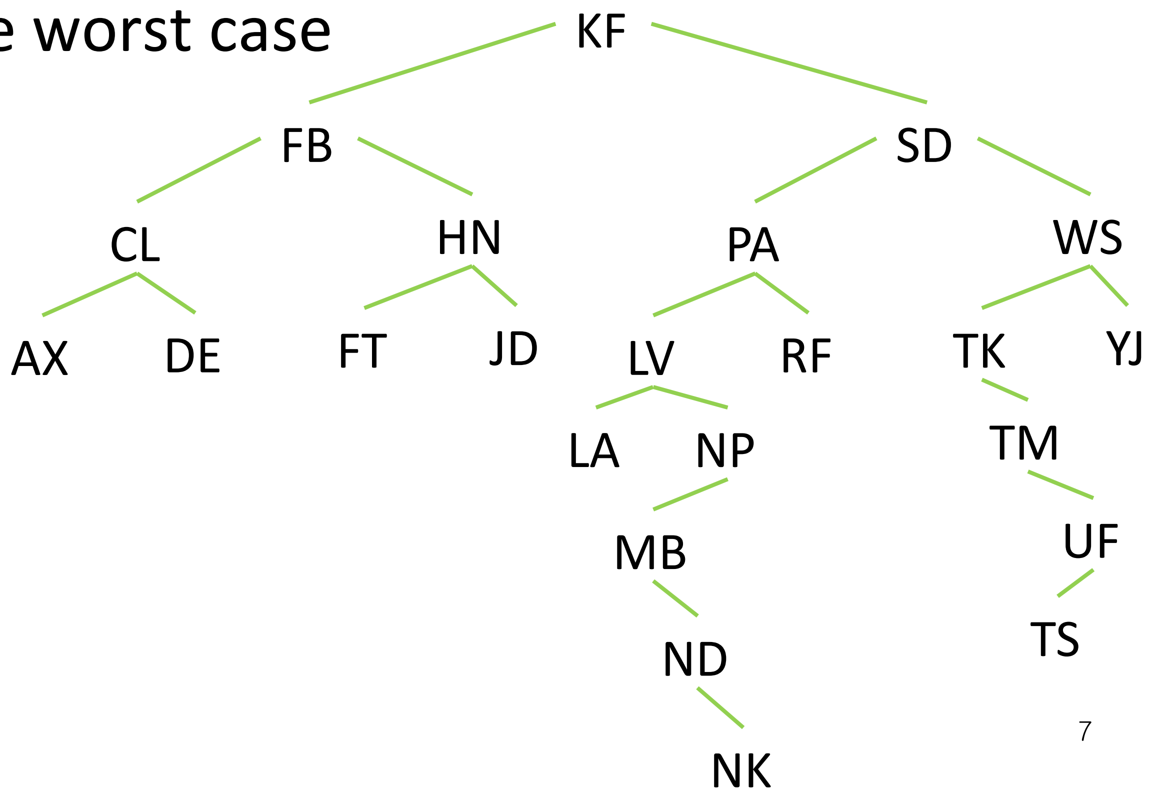
ROOT ⇨ 9

	key	*left*	*right*
0	FB	10	8
1	JD		
2	RF		
3	SD	6	13
4	AX		
5	YJ		
6	PA	11	2
7	FT		

	key	*left*	*right*
8	HN	7	1
9	KF	0	3
10	CL	4	12
11	NR		
12	DE		
13	WS	14	5
14	TK		

Indexing with Binary Search Trees (3/4)

- Unbalanced Binary Tree
  - require 7, 8, or 9 seeks for retrieval
  - Binary search on a sorted list of 24 keys
  - => 5 seeks in the worst case



Indexing with Binary Search Trees (4/4)

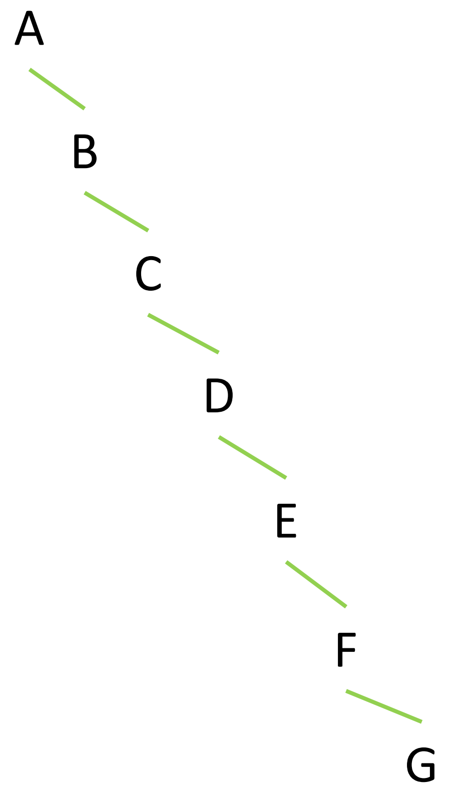
- Problems
  - A BST is not fast enough for disk resident indexing
  - lack of an effective strategy of balancing the tree
- Alternatives
  - AVL tree
  - Paged Binary Tree

AVL Trees (1/5)

- undesirable tree organizations
  - alphabetical order produces a degenerate tree

  → need to reorganize the nodes of the tree

A degenerate tree

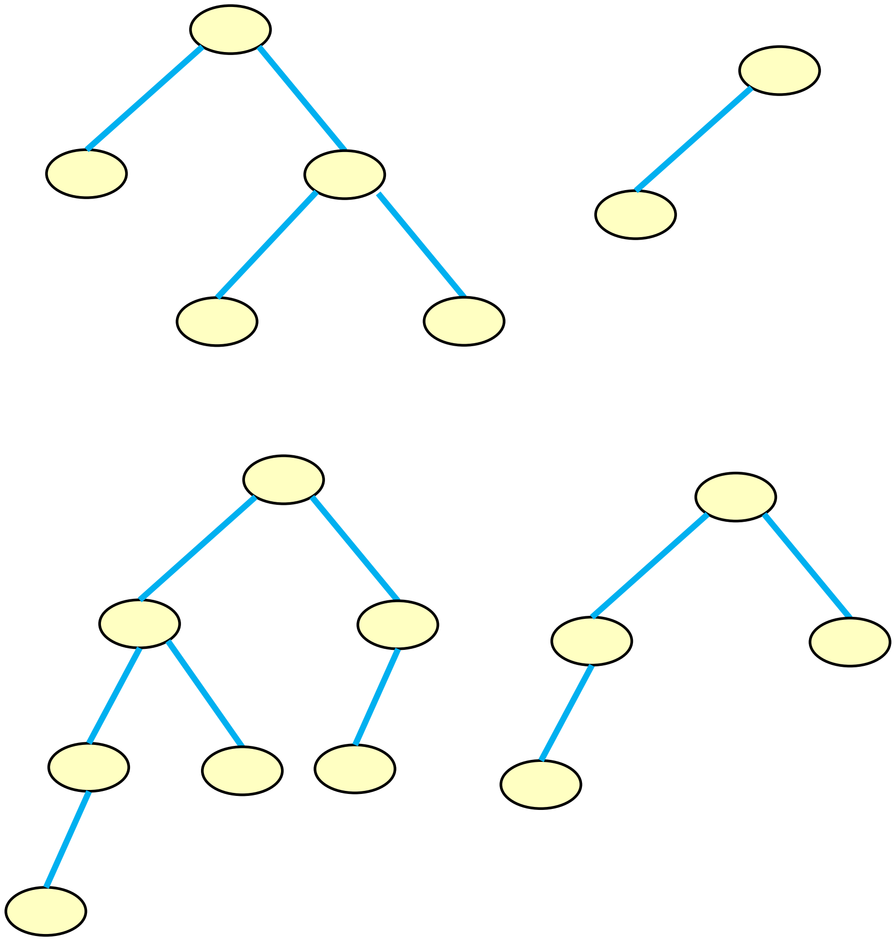


AVL Trees (2/5)

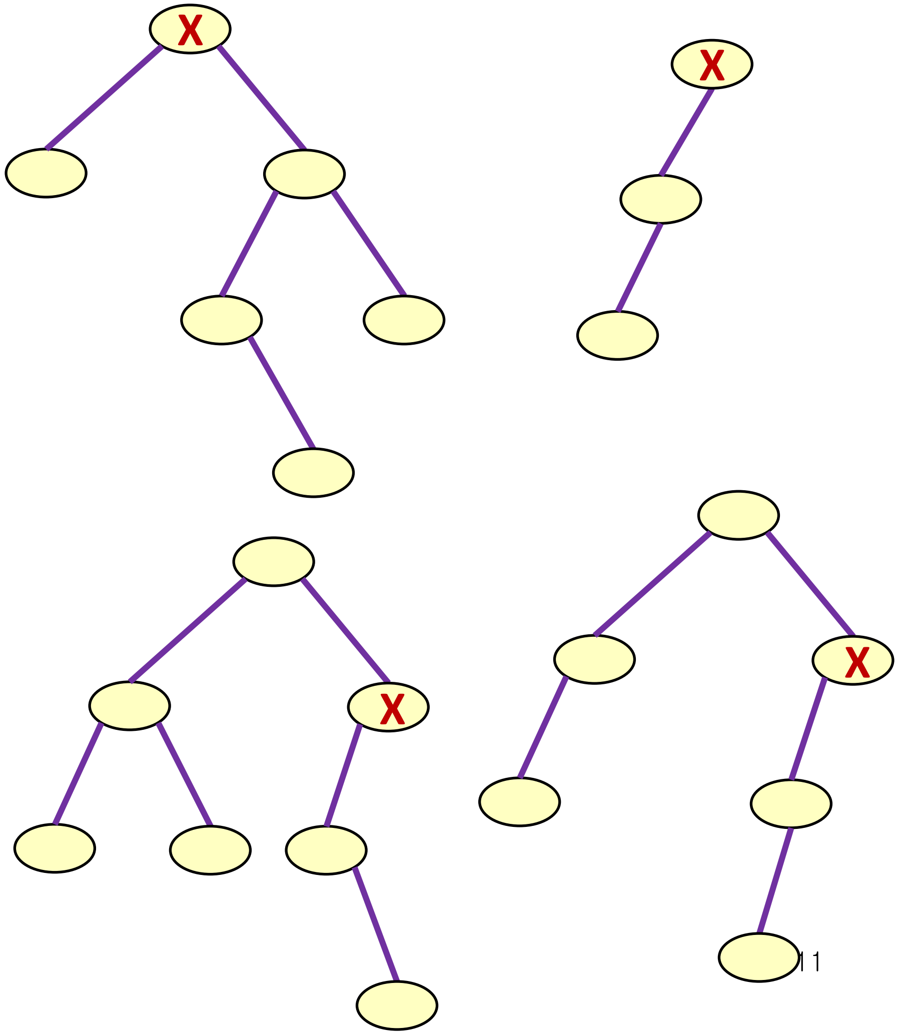
- AVL trees
  - defined by Russian mathematicians, G. M. Adel'son-Vel'skii and E.M. Landis
  - a near optimal tree structure
  - height-balanced 1-tree (HB(1) tree)
    - height-balanced trees with the maximum difference is 1
  - no two subtrees of any root differ by more than one level
  - not directly applicable to file structure problems
    - AVL trees have too many levels
  - guarantees that search performance approximates that of a completely balanced tree

AVL Trees (3/5)

- AVL Trees

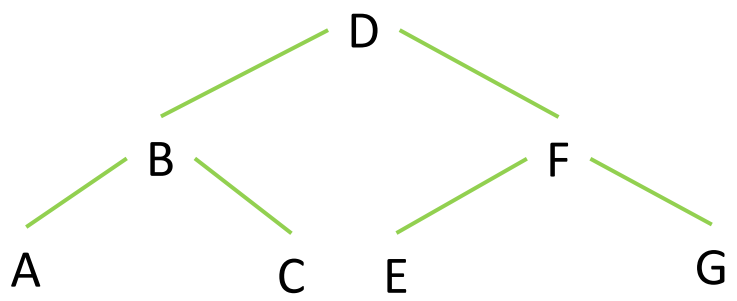


- Non AVL Trees

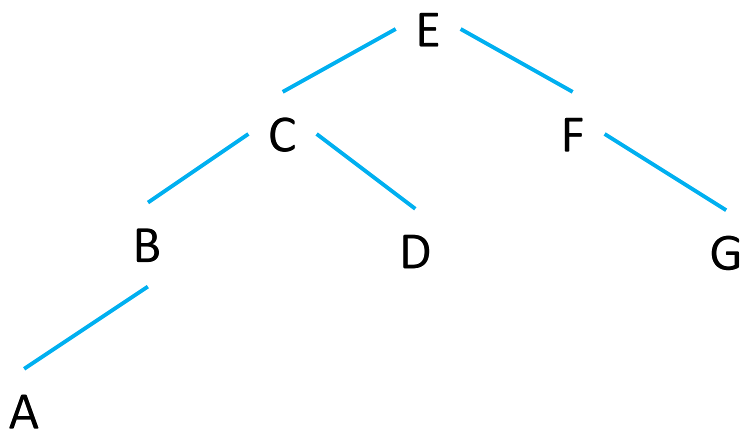


AVL Trees (4/5)

- Completed balanced BST from B,C,G,E,F,D,A



- AVL tree from B,C,G,E,F,D,A



AVL Trees (5/5)

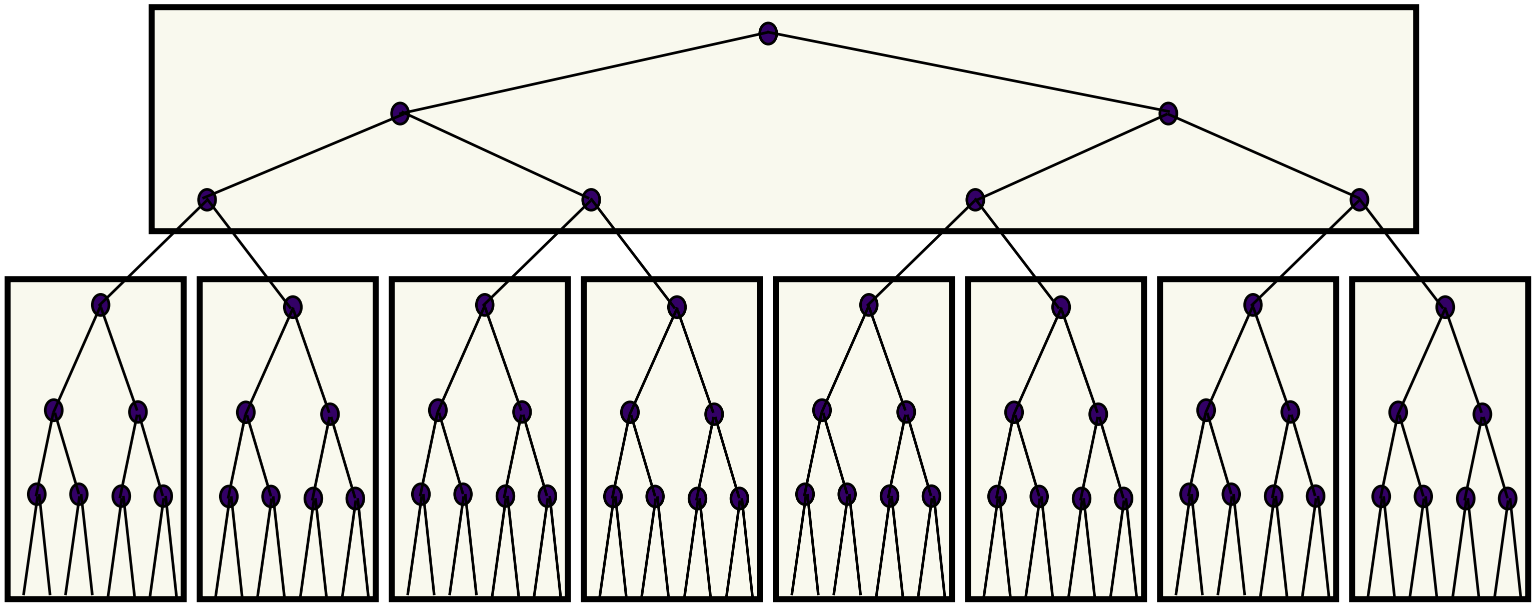
- Complexity comparison (e.g., 1,000,000 keys)
  - (1) completely balanced tree
    - the worst-case search to find a key: log2(N + 1)
    - require seeking 20 levels
  - (2) AVL trees
    - the worst-case search to find a key: 1.44 log2(N + 2)
    - require seeking 29 levels
- Again, two problems
  - Binary searching requires too many seeks
  - keeping an index in sorted order is expensive
    - binary trees provide an acceptable solution

Paged Binary Tree (1/3)

- Page
  - A unit of disk I/O for handling seek and transfer of disk data
  - Typically, 4k, 8k, 16k ...
- Paged Binary Tree
  - Divide a binary tree into pages and then store each page in a block of contiguous locations on disk.

Paged Binary Tree (2/3)

- An example of Paged binary tree
  - If every page holds 7 keys, 511 nodes(keys) in only three seeks



Paged Binary Tree (3/3)

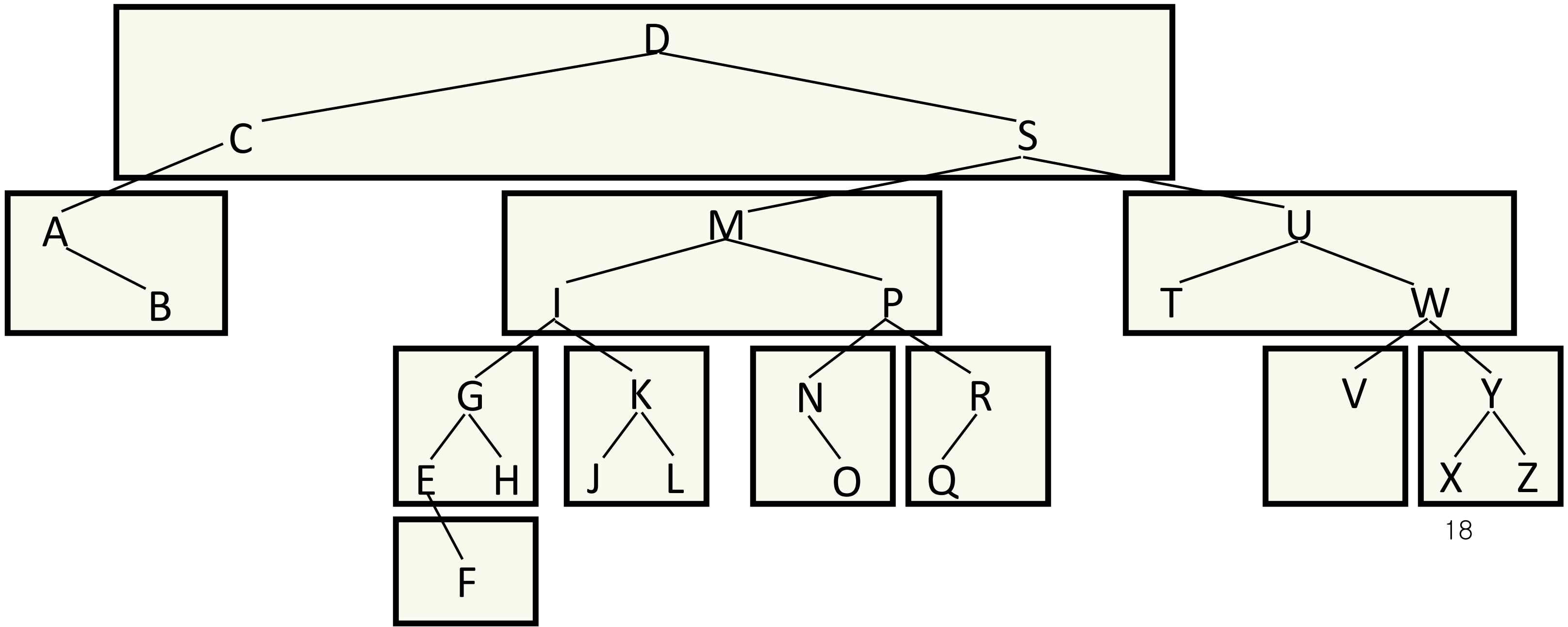
- A typical example
  - page size : 8 KB for k=511 key/reference field pairs
  - N (# of keys) = 134,217,727
  - ① Completely full, balanced binary tree
    - $log_2(N+1) = 27$ seeks
  - ② Paged version of a completely full, balanced binary tree
    - $log_{k+1}(N+1) =$3 seeks (k : # of keys held in a single page)
    - => can reduce disk seeks

Problems with Paged Trees (1/3)

- Only valid when we have the entire set of keys in hand before the tree is built
- Problems due to <u>out of balance</u>
  - How to select a good separator
  - How to group keys
  - How to guarantee the maximum loading
- B-tree provides a solution for above problems!

Problems with Paged Trees (2/3)

- Construction of the paged binary trees
  - from the following sequence of keys:
    - 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
  - contain a maximum of three keys per pages
  - rotate keys within a page to keep each page as balanced as possible



Problems with Paged Trees (3/3)

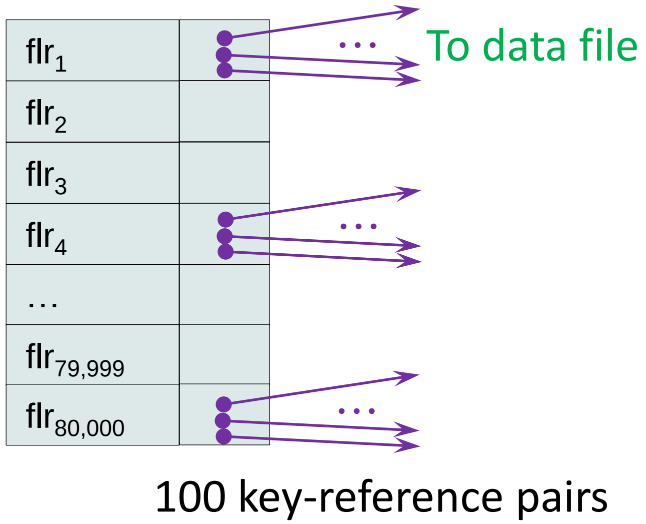
- Three unsolved questions
  - (1) the keys in the root turn out to be good separator keys ?
    - divide up the set of other keys more or less evenly
  - (2) how do we avoid grouping keys that should not share a page ?
    - C, D, S should not share the same page
  - (3) each of the pages contains at least some minimum number of keys?
    - maintain a lower bound
  - => B-trees : build trees upward from bottom instead of downward from the top

Multilevel indexing (1/5)

- Extension of the single record indexing
  - Approach as <u>simple index record</u>
    - limited on the number of keys allowed
  - Approach as <u>multirecord index</u>
    - consists of a sequence of simple index records
    - binary search is too expensive
  - Approach as <u>multilevel index</u>
    - reduced the number of records to be searched
    - speed up the search

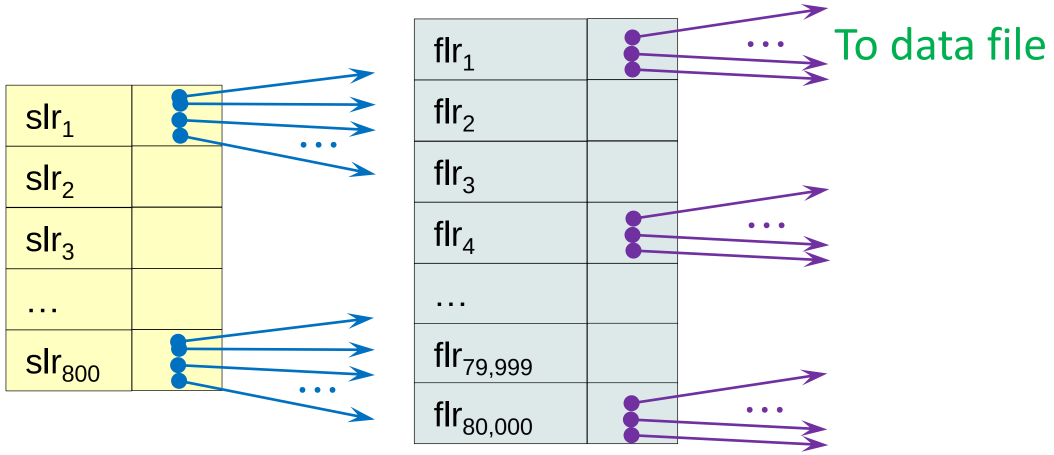
Multilevel indexing (2/5)

- 800 MB file of 8,000,000 records
  - 100 bytes each, with 10 byte keys
  - Index has 8,000,000 key-reference pairs
  - Each index record has 100 key-reference pairs
- (1) first level index
  - 80,000 (=8,000,000/100) records for 8,000,000 keys
  - Index to the data file



Multilevel indexing (3/5)

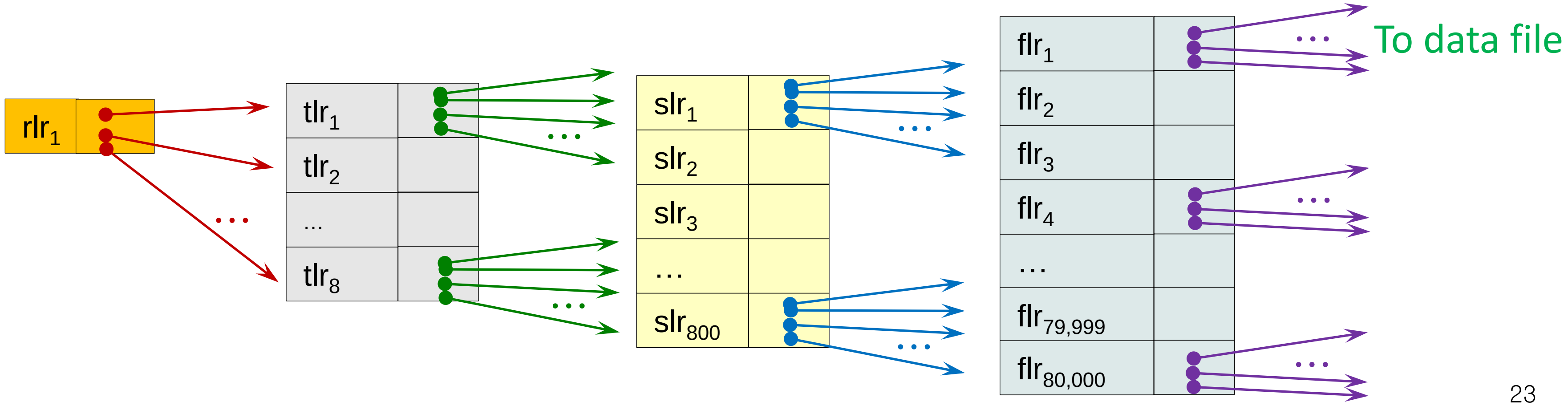
- (2) second level index
  - 800 index records for 80,000 keys
  - choose one of the keys (the largest) in each index record
  - use reference fields for index record addresses



100 key-reference pairs

Multilevel indexing (4/5)

- (3) third level index / (4) fourth level index
  - 8 index records for 800 keys /1 index records for 8 keys
- Total
  - ① total 80,809 index records
  - ② average, min., max. # of disk access = 4 (∵ 4 levels)



100 key-reference pairs

Multilevel indexing (5/5)

- How can we insert new keys into the multilevel index?
    - The index records in some level might be full
    - The several levels of indexes might be rebuilt
    - Overflow chain may be helpful, but still ugly

- Multi-level index structure is not strong in dynamic data processing applications

- B-tree will give you the right solution!

Outline

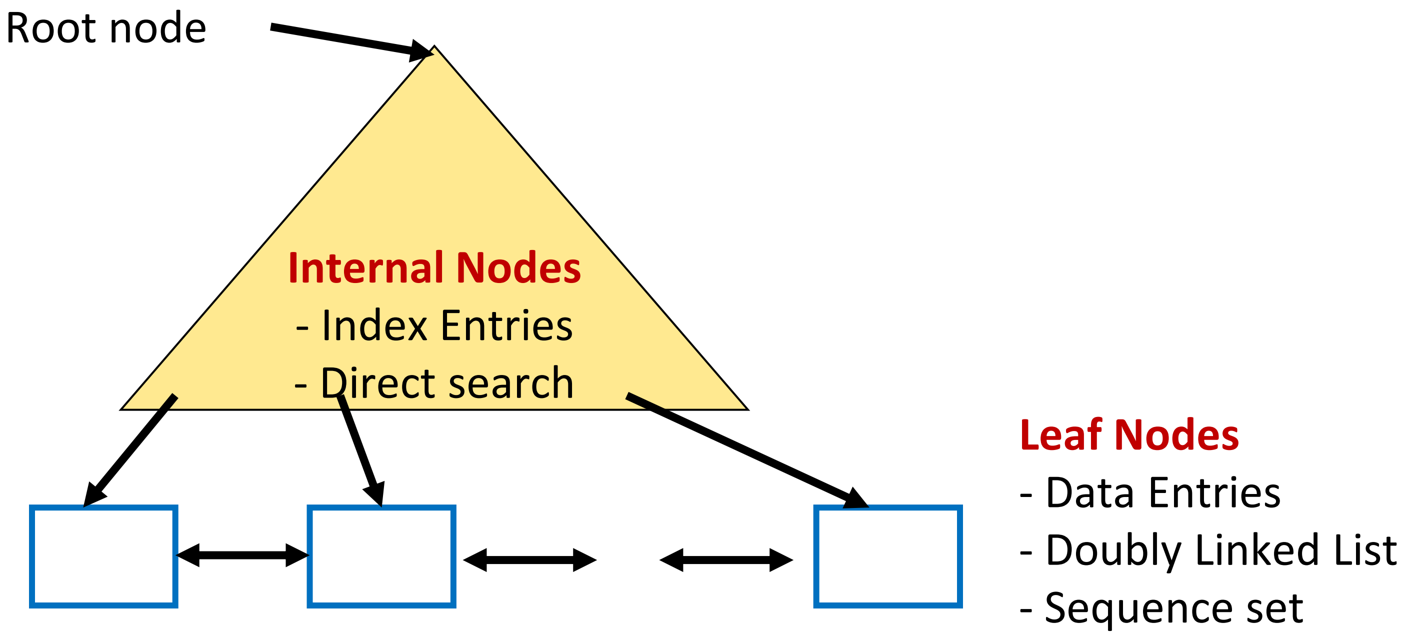
- 9.5 B-Trees
- 9.10 Formal Definition of B-Tree Properties
- 9.6 Example of Creating a B-Tree

B-Trees: Working up from the bottom

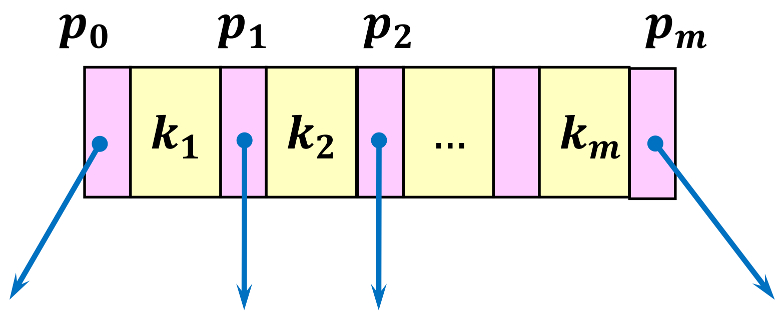
- B-trees
  - multilevel indexes defined by Bayer and McCreight
  - built upward from the bottom(i.e., leaves) instead of downward from the top
  - do not require that the index records be full
  - do not shift the overflow keys to the next record
    - split an overfull record into two records, each half full
  - deletion takes a similar strategy of merging two records into a single record when necessary

B-Trees

- Measures
  - **Order:** the (maximum) number of indexing field values at each node
  - Height: the number of indexing levels from root to any leaf

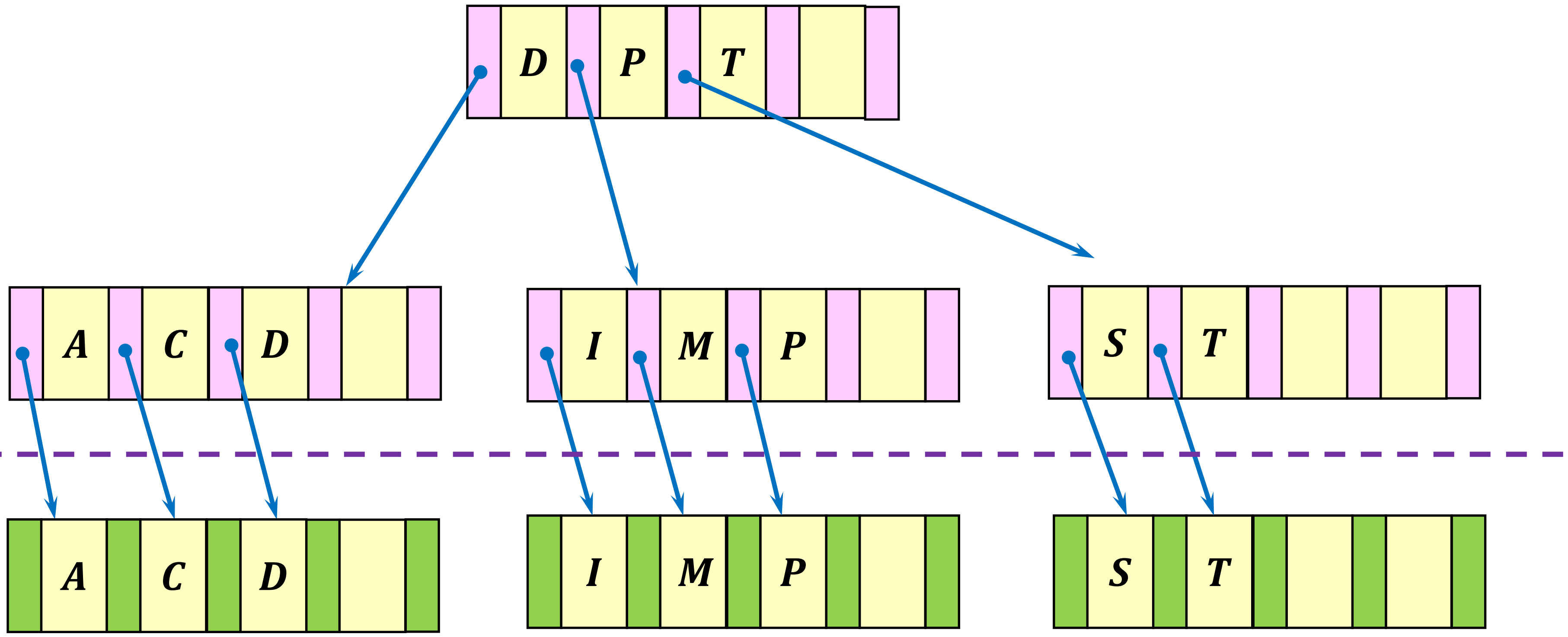


B-Tree Properties



- The properties of a B-tree of order n
  - 1. Every page has a maximum of n descendants
  - 2. Every page, except for the root and the leaves, has at least ceiling of (n/2) descendants
    - n/2 <= m <= n (half-full), n = order of tree (fanout)
  - 3. The root has at least two descendants (unless it is a leaf)
  - 4. All the leaves appear **on the same level**
  - 5. The leaf level forms a complete, ordered index of the associated data file

Sample B-Tree



Record Insertion(1/2)

- When a record is inserted in the data file, the B+-tree must be changed accordingly:
  - simple case
    - leaf not full: just insert (key, pointer-to-record)
  - leaf overflow
  - Internal node overflow
  - new root

Record Insertion(2/2)

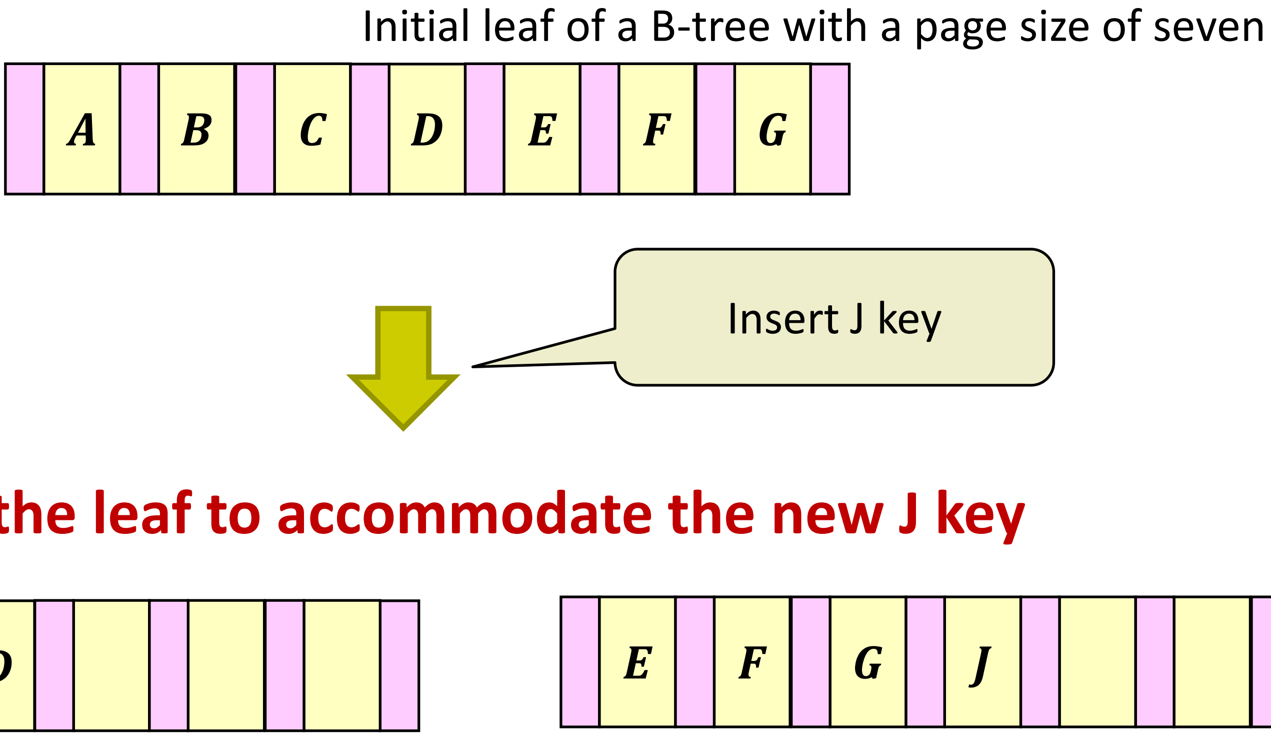
- Algorithm: (from DB book, slightly different)
  - Find correct leaf *L.*
  - Put data entry onto *L*.
    - If *L* has enough space, *done*!
    - Else, must *split* *L (into L and a new node L2)*
      - Redistribute entries evenly, **copy up** middle key.
  - parent node may overflow
    - but then: **push up** middle key. Splits "grow" tree; root split increases height.

Splitting & Promoting (1/3)

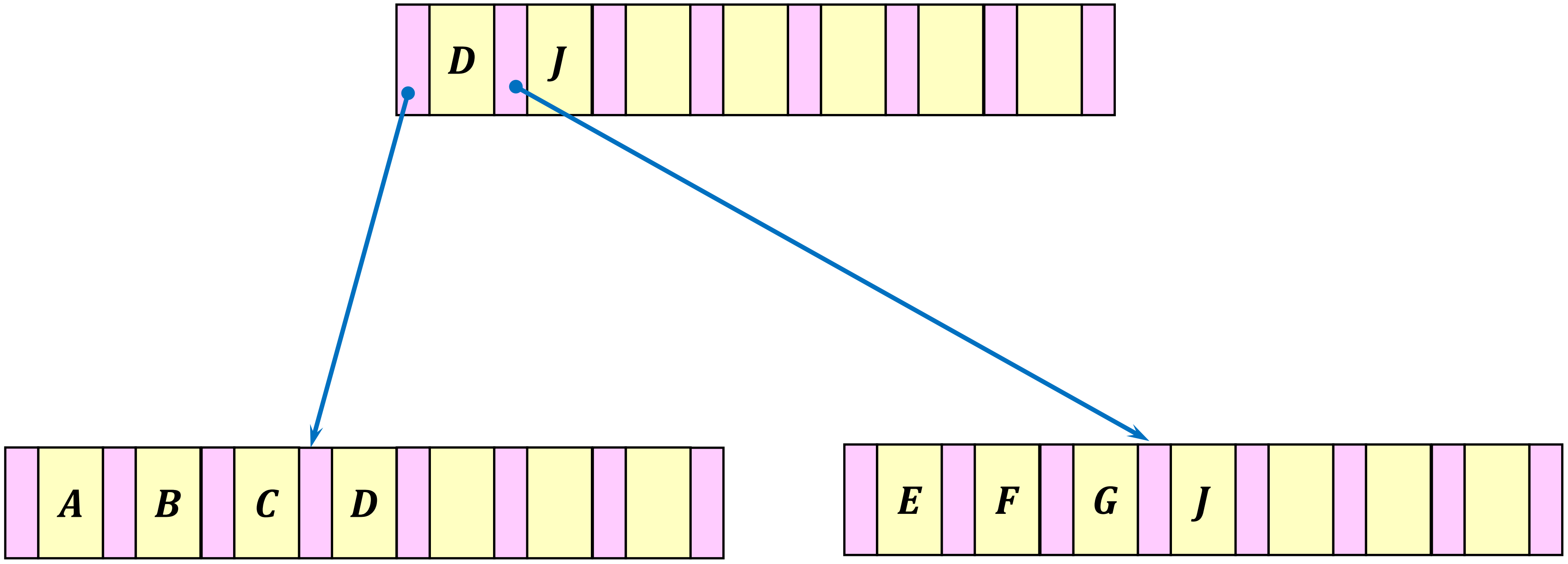
- Splitting
    - Creation of two nodes out of one because the original node becomes overfull
    - Result in the need to promote a key to a higher-level node to provide an index separating the two new nodes
- Promotion of a key
    - Movement of a largest key from one node into a higher-level node when split occurs

Splitting & Promoting (2/3)

- Assume: order = 7


Initial leaf of a B-tree with a page size of seven
A
B
C
D
E
F
G
Insert J key
Splitting the leaf to accommodate the new J key
A
B
C
D
E
F
G
J

Splitting & Promoting (3/3)

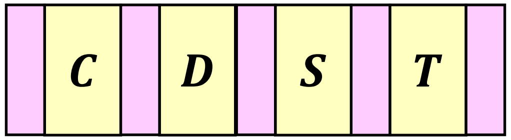


Promotion of the D, J keys into a root node (copy up)

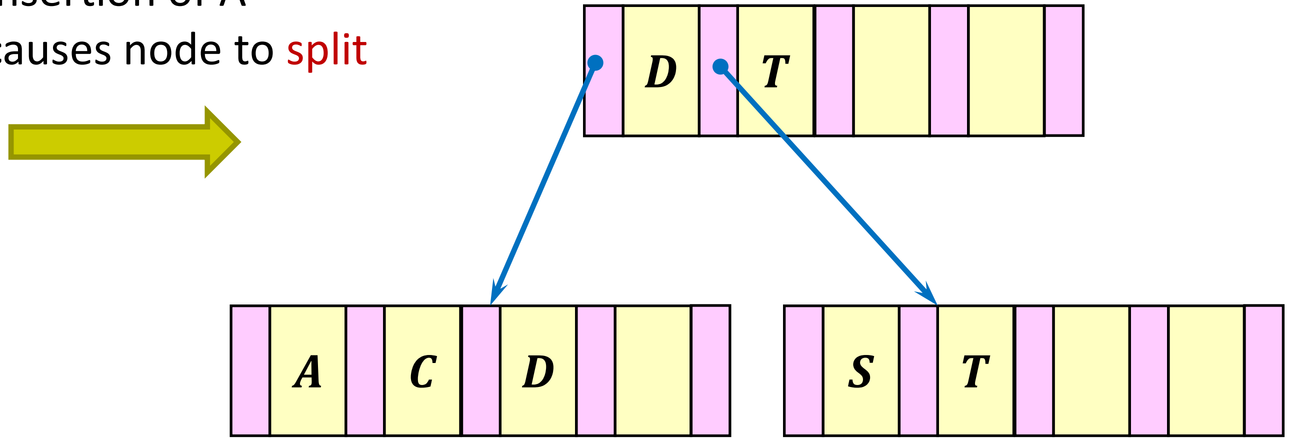
Creating a B-tree (1/10)

- Assume: Order of a B-tree = 4
- Input Sequence
  - 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

Insertion of C, S, D, T into the initial page



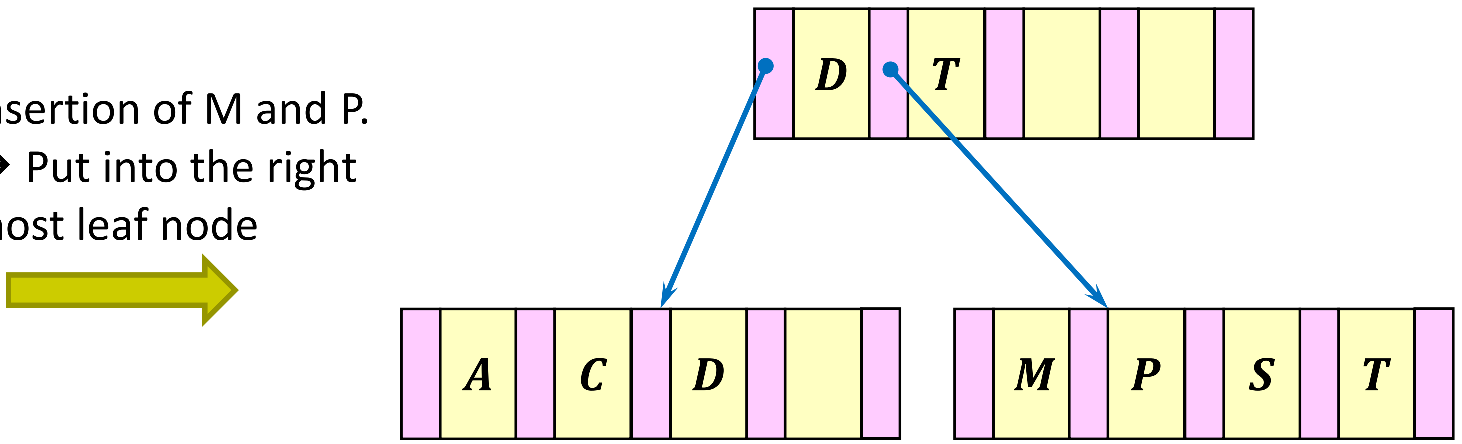
Insertion of A causes node to split



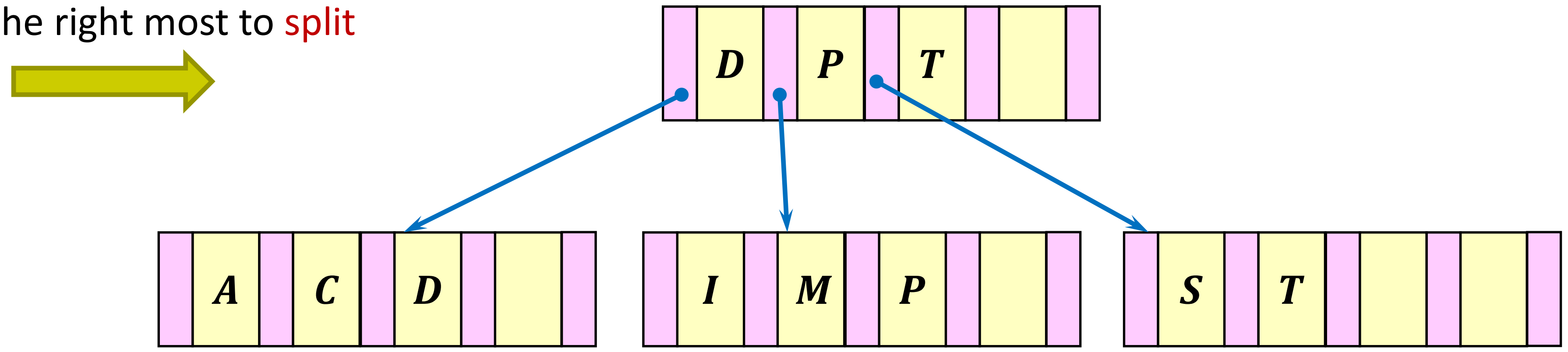
the largest key in each leaf node (D and T)to be placed in the root node

Creating a B-tree (2/10)

Insertion of M and P.
→ Put into the right most leaf node

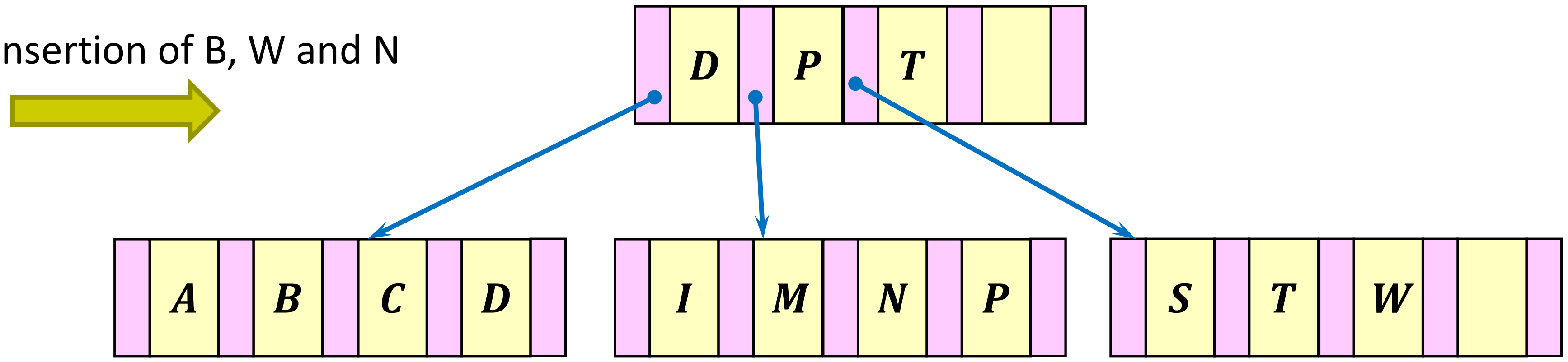


Insertion of I causes the right most to split

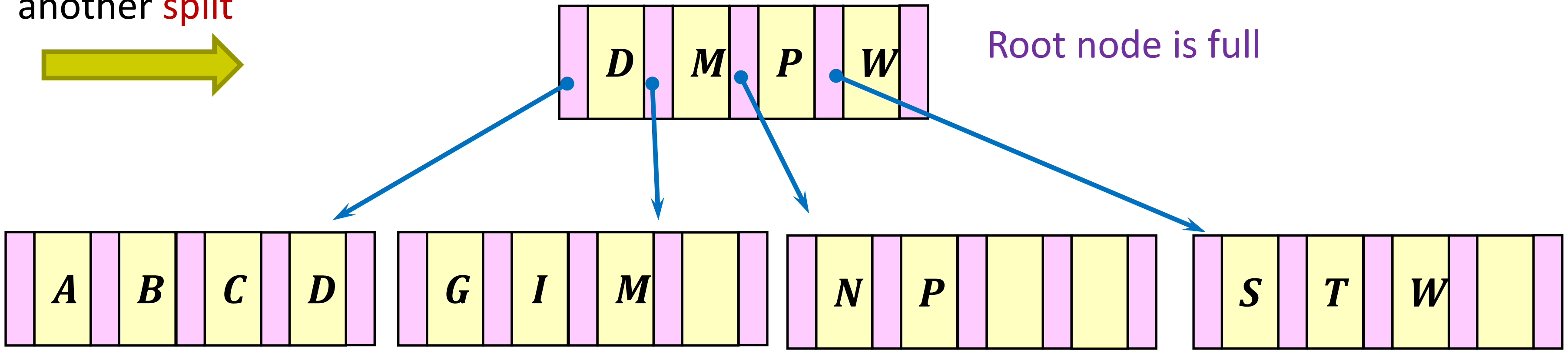


Creating a B-tree (3/10)

Insertion of B, W and N



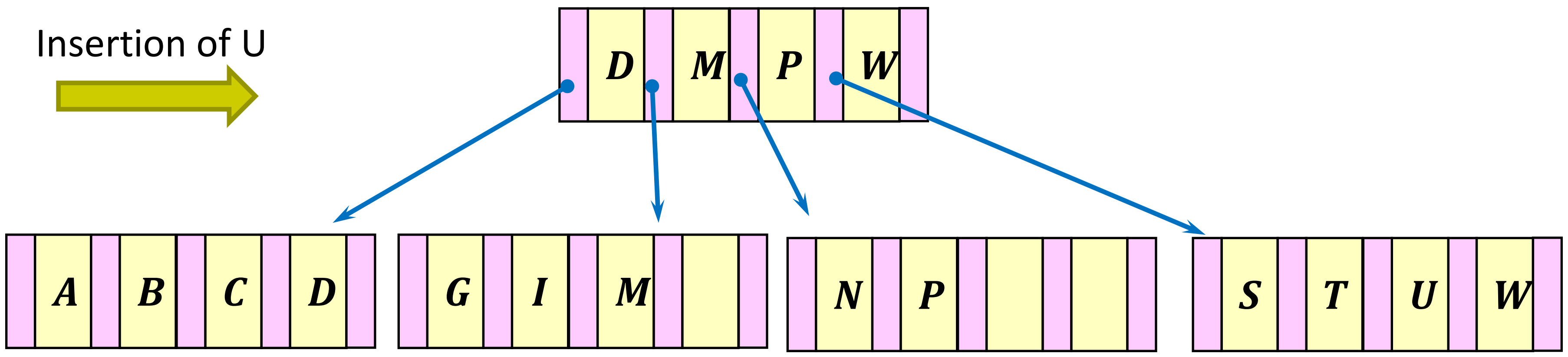
Insertion of G causes another split



Root node is full

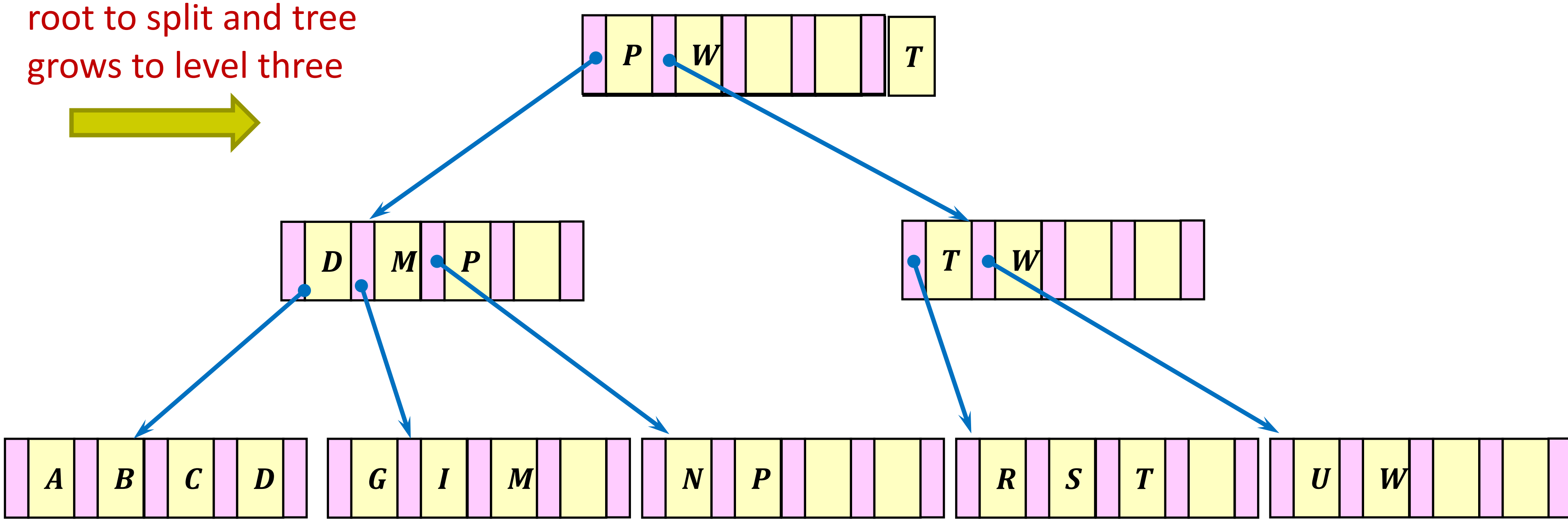
Creating a B-tree (4/10)

Insertion of U



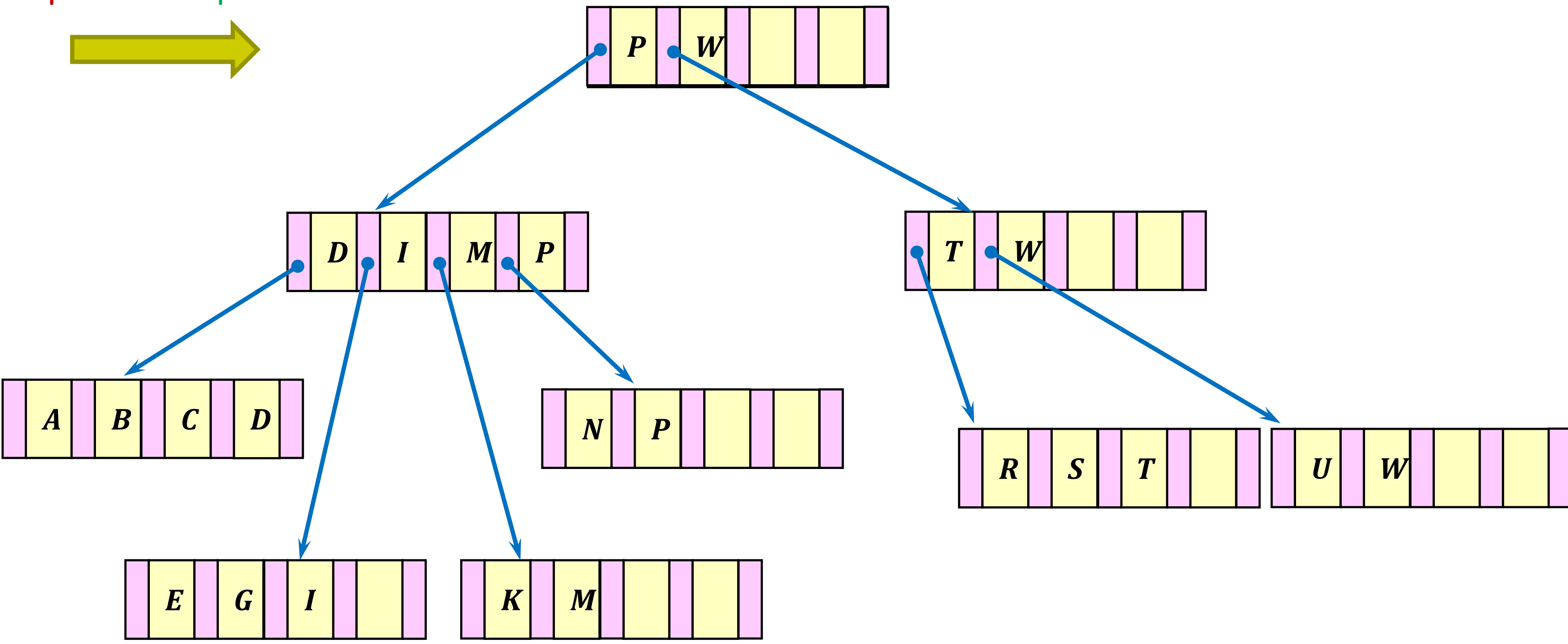
Creating a B-tree (5/10)

Insertion of R into the rightmost leaf to split, Insertion into the root to split and tree grows to level three



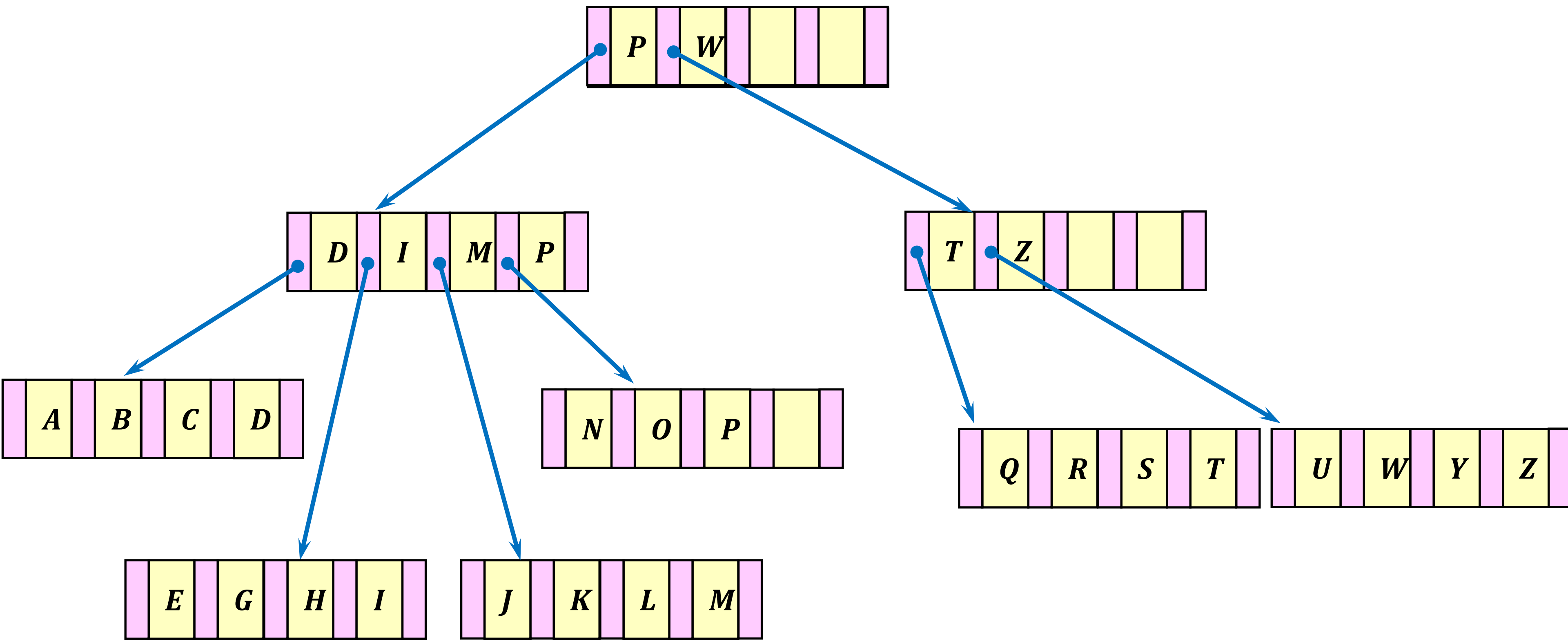
Creating a B-tree (6/10)

Insertion of K, E →
the second node to
split → I is promoted



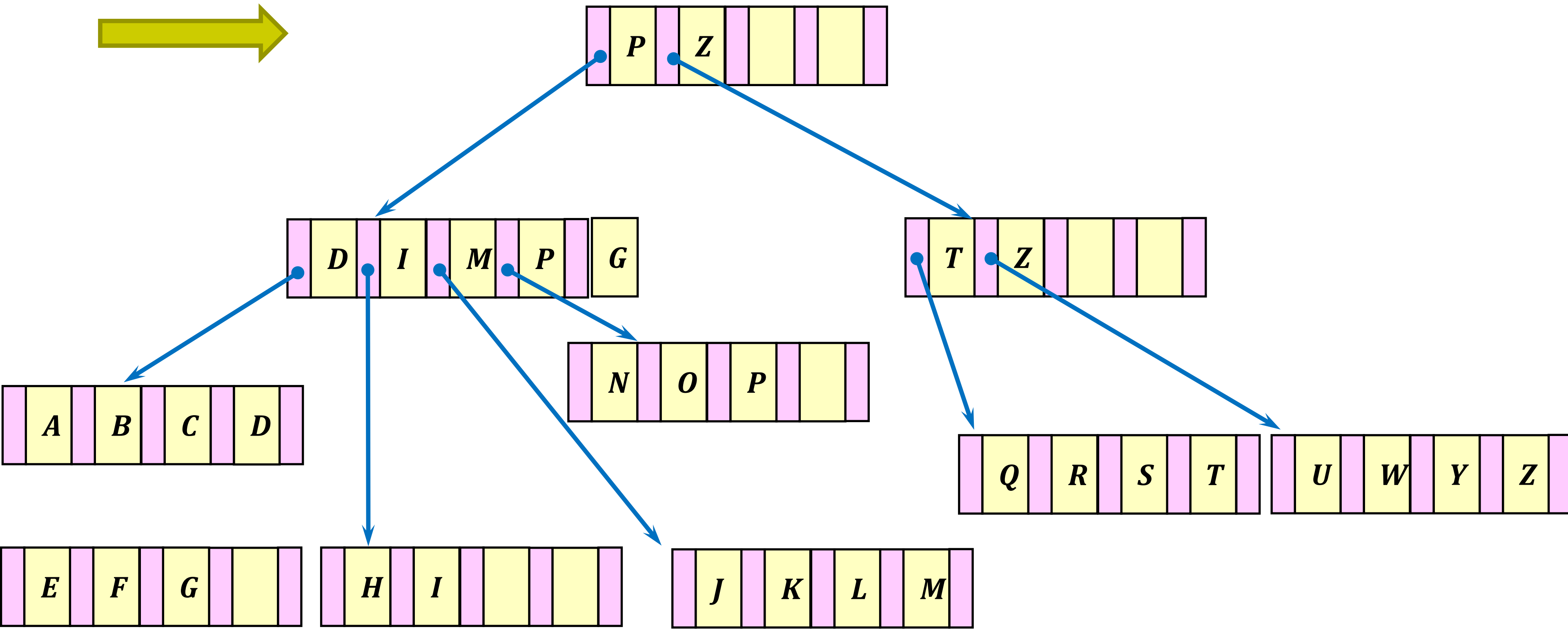
Creating a B-tree (7/10)

Insertion of H, O, L, J, Y, Q and Z



Creating a B-tree (8/10)

Insertion of F → split
Of the second leaf node
→ G is promoted

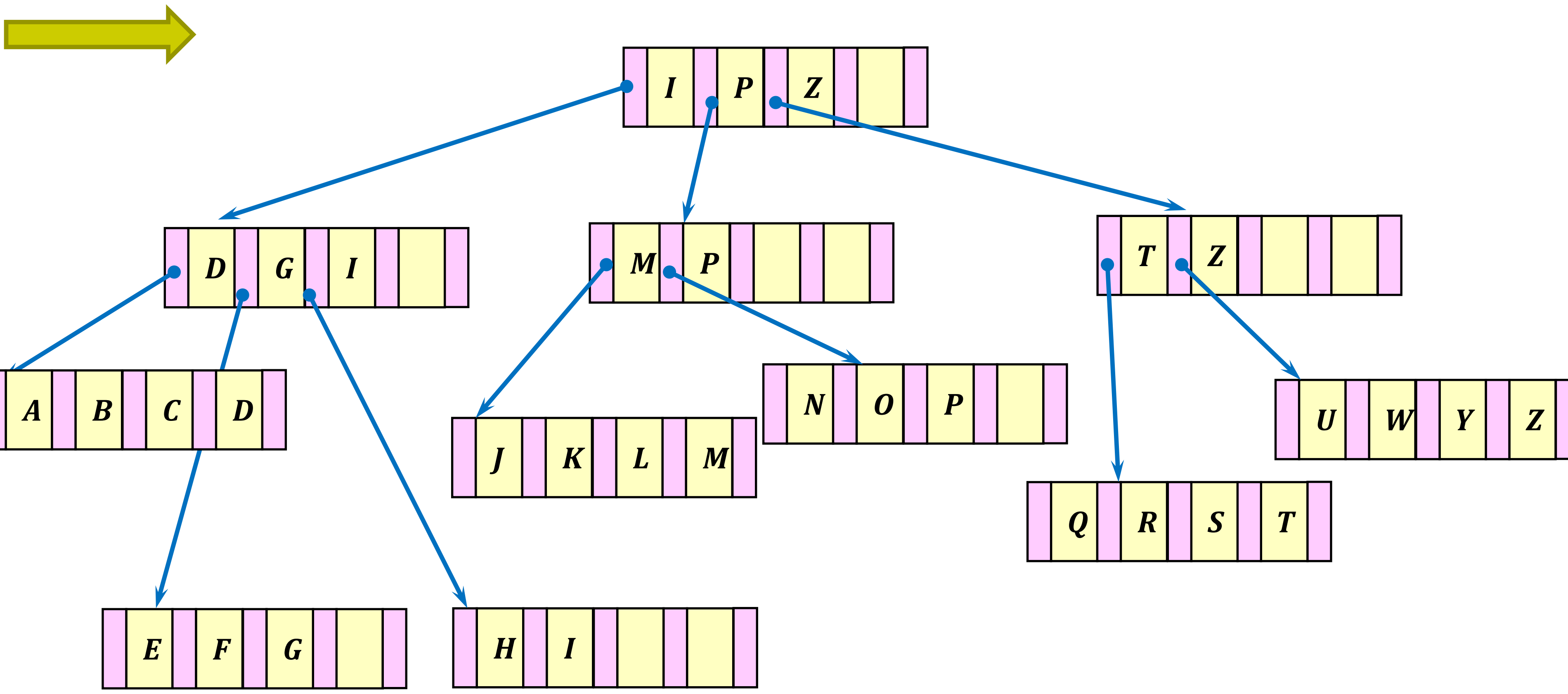


Creating a B-tree (9/10)

Insertion of F → split the second leaf node

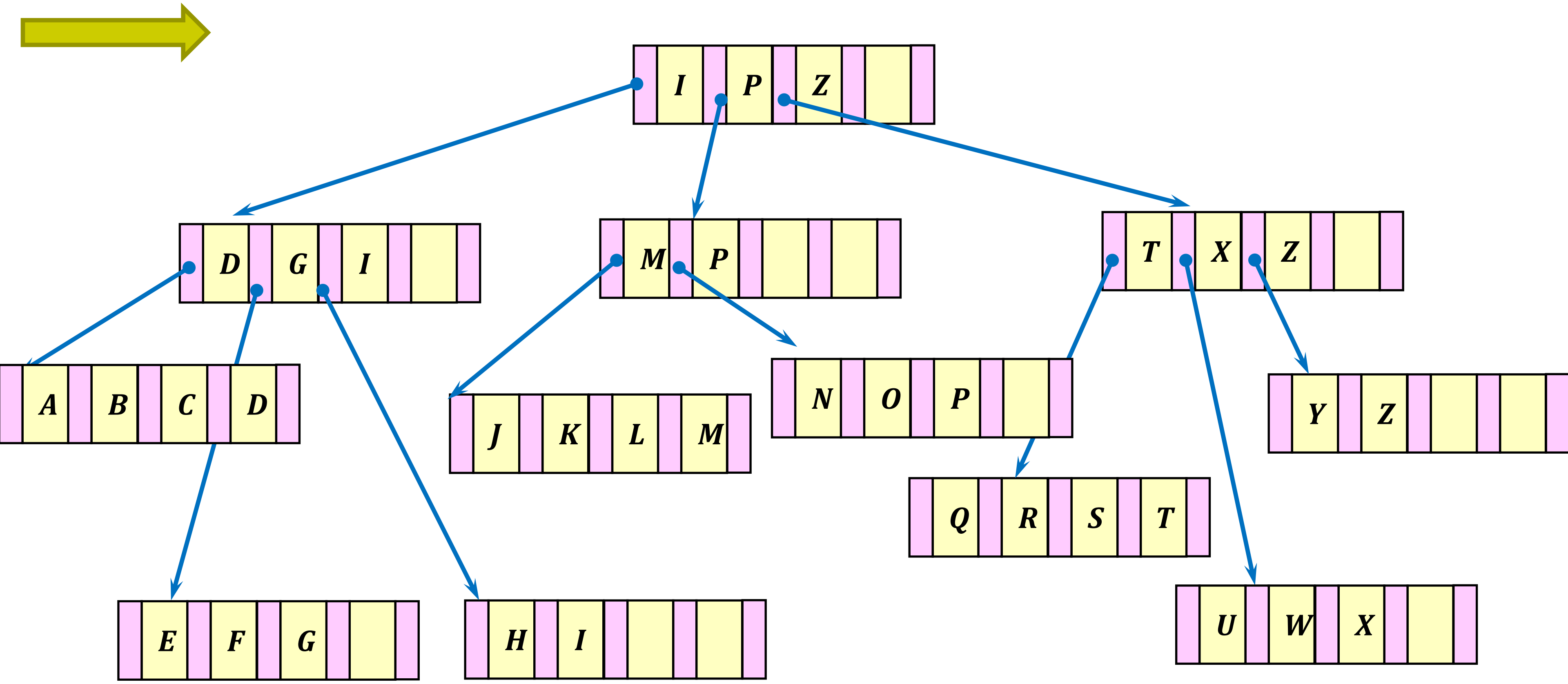
→ G is promoted

→ Cascade split and I is promoted



Creating a B-tree (10/10)

Insertion of X → split the rightmost node
→ X is promoted



Recap: Insertion in B-trees

- Major components of insertion
  - Split the node
  - Promote the middle key
  - Increase the height of the B-tree

- Insertion may touch no more than 2 nodes per level
- Insertion cost is strictly linear in the height of the tree

Q&A