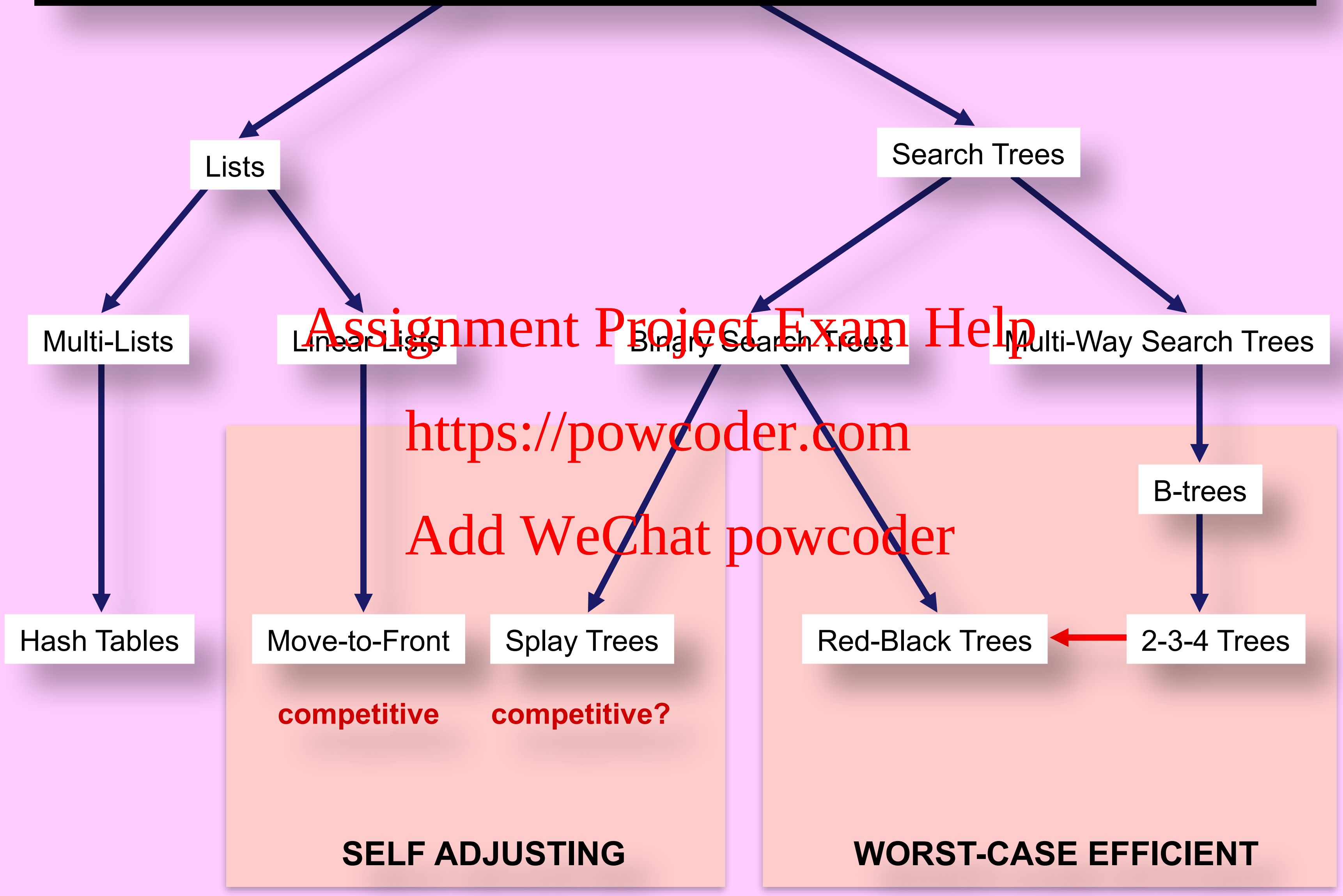
EECS 4101/5101

Prof. Andy Mirzaian

B-trees
2-3-4 trees

DICTIONARIES



References:

- [CLRS] chapter 18

B-trees

R. Bayer, E.M. McCreight,
"Organization and maintenance of large ordered indexes,"
Acta Informatica 1(3), 173-189, 1972.

Boeing Company

B…

Definition

- B-trees are a special class of multi-way search trees.

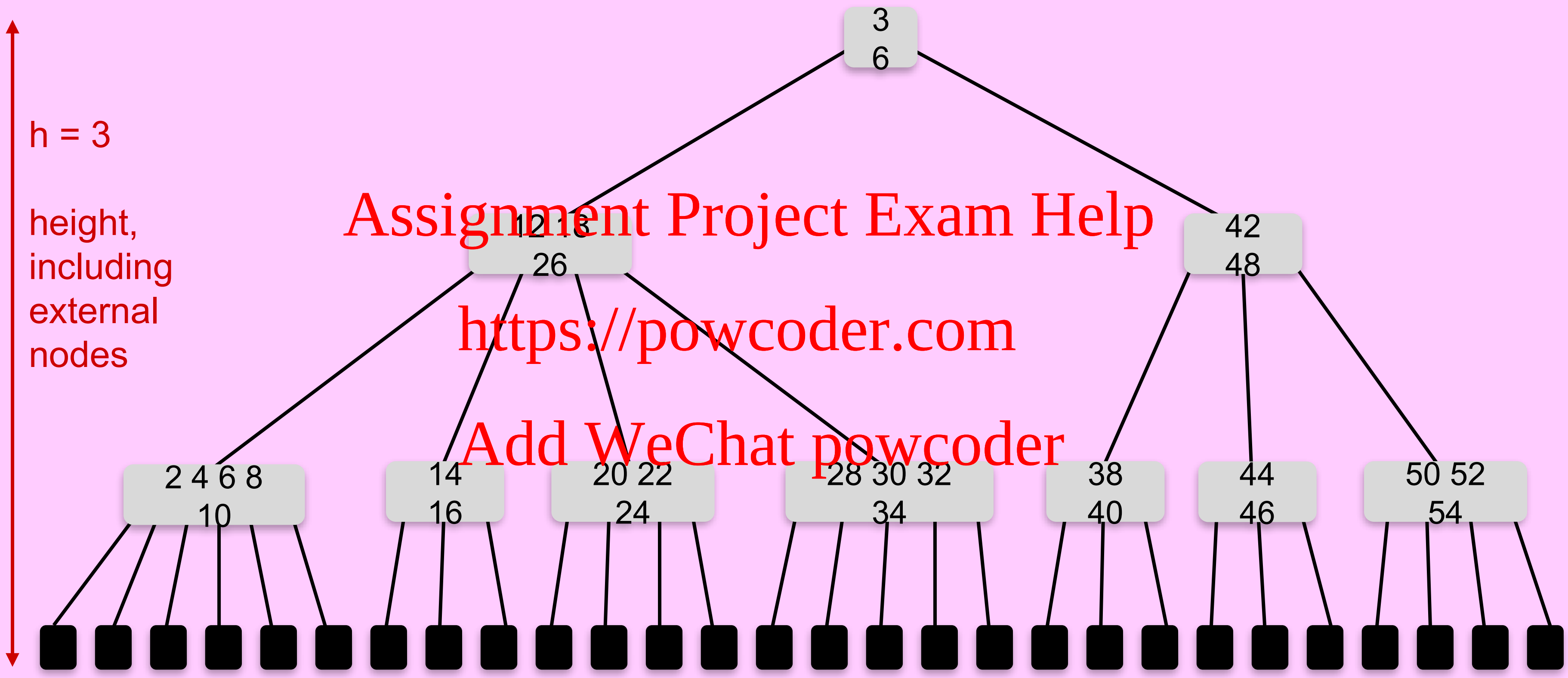
- Node size:
  - $d[x]$ = degree of node x, i.e., number of subtrees of x.
  - $d[x] - 1$ = number of keys stored in node x. (This is $n[x]$ in [CLRS].)

- **Definition:** Suppose d 2 is a fixed integer.
  T is a B-tree of order d if it satisfies the following properties:

1. **Search property:** T is a multi-way search tree.
2. **Perfect balance:** All external nodes of T have the same depth (h).
3. **Node size range:** $d \le d[x] \le 2d \qquad \forall$ nodes $x \ne root[T]$
   $2 \le d[x] \le 2d \qquad$ for $x = root[T]$.

Example

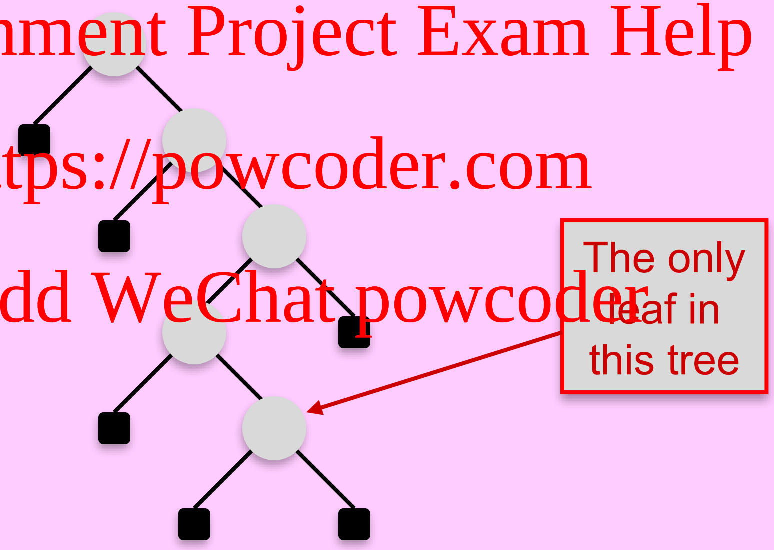
A B-tree of order d = 3 with n = 27 keys.



Warning

2. Perfect Balance: All external nodes of T have the same depth.

[CLRS] says: All leaves have the same depth.



Applications

External Memory Dictionary:

- Large dictionaries stored in external memory.
- Each node occupies a memory page.
- Each node access triggers a slow page I/O.
  Keep height low. Make d large.
- A memory page should hold the largest possible node (of degree 2d).
  Typical d is in the range 50 .. 2000 depending on page size.

Internal Memory Dictionary:

- 2-3-4 tree = B-tree of order 2.

B-tree Height

How small or large can a B-tree of order and height be?

- keys in the B-tree
- external nodes, all at depth
- This is lower/upper bounded by the min/max aggregate branching at depths :

- Take logarithm:

Height in B-trees & 2-3-4 trees

B-trees: $\log_{2d}(n+1) \le h \le 1 + \log_d \frac{n+1}{2}$.

2-3-4 trees: $\frac{1}{2}\log_2(n+1) \le h \le \log_2(n+1)$.

This includes the level of external nodes.

SEARCH

Algorithm:

Simply follow the search path for the given key.

Complexity:

At most O(h) nodes probed along the search path.
Each node has O(d) keys.

 # I/O operations:

- $O(h) = O(\log_d n) = O(\log n / \log d)$.
- So, for external memory use keep d high.
  Page size is the limiting factor.

Search time:

- binary search probing within node: O(h log d) = O(log n).
- sequential probing within node: O(hd) = O(d log n / log d).
- INSERT & DELETE (to be shown) also take O(hd) time.
- So, for internal memory use keep d low (e.g., 2-3-4 tree).

Local Restructuring

INSERT and DELETE need the following local operations:

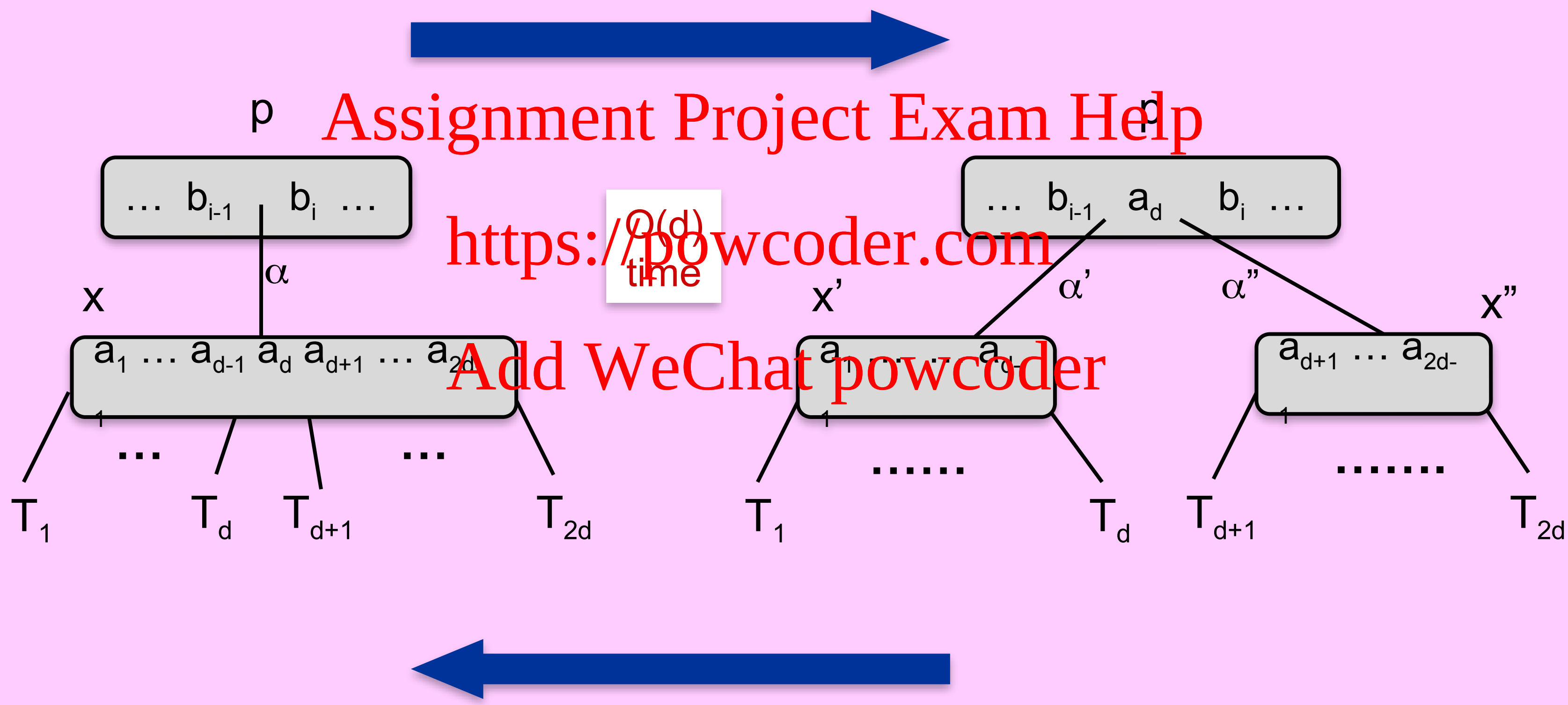
- **Node splitting**
- **Node fusing**
- **Key sharing (or key borrowing)**

The first one is used by INSERT.
The last 2 are used by DELETE.
Each of them takes O(d) time.

Node Splitting & Fusing

Split(x)

Parent p gains a new key. (If p was full, it needs repair too.)
If x was the root, p becomes the new root (of degree 2) & tree height grows by 1.



Fuse (x', a_d, x")

Parent p loses a key. (If non-root p was a d-node, it needs repair too.)
If p was the root & becomes empty, x becomes the new root & tree height shrinks by 1.

Key Sharing

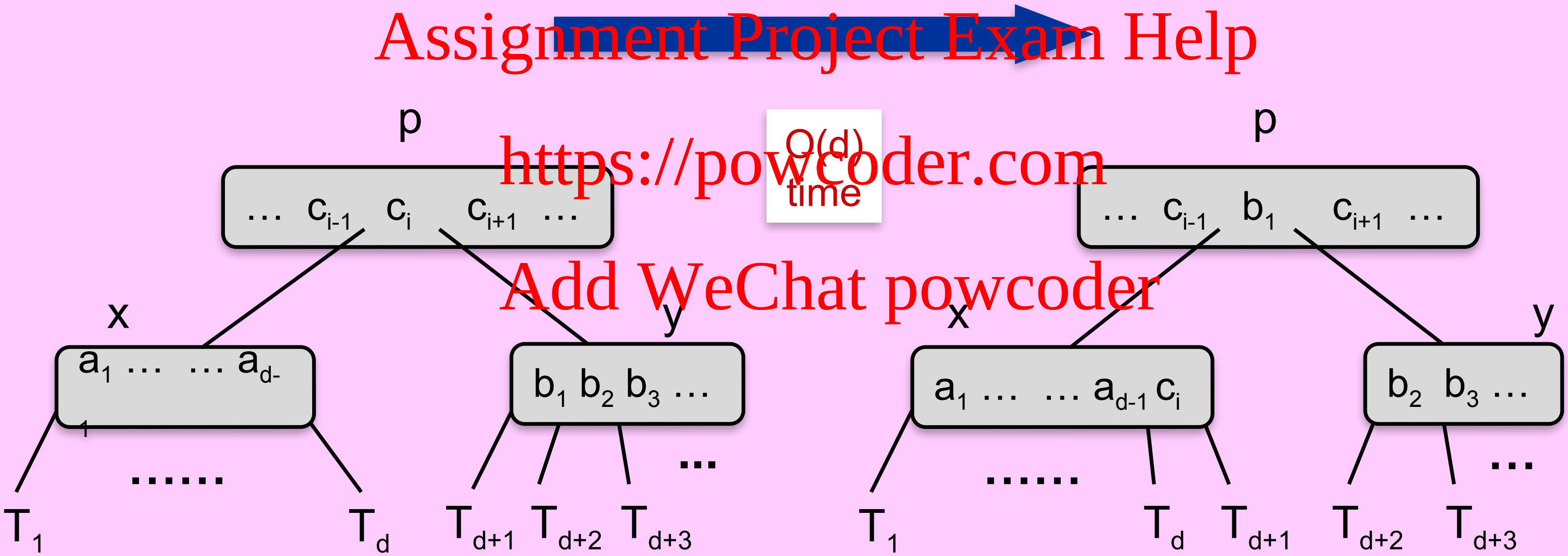
KeyShare(x,y)

Node y is the (left or right) immediate sibling of x.

d[x]=d, d[y]>d.

x “borrows” a key from y to make d[x] > d.

Note: the inorder sequence of keys is not disturbed.



Bottom-Up INSERT

Bottom-Up INSERT (K,T)

Follow the search path of K down B-tree T.
if K is found **then return**
Otherwise, we are at a leaf x where K needs to be inserted.
K' ← K (* the insertion key at the current node x *)
while d[x] = 2d **do** (* repeated node splitting up the search path *)

Split x into x' and x'' with middle key $key_d[x]$
if K' < $key_d[x]$ **then** insert K' into x' **else** insert K' into x''

K' ← $key_d[x]$
x ← p[x]

end-while
if x ≠ nil **then** insert K' into x
else do
create a new node r
root[T] ← r
insert K' into r
make x' and x'' the two children of r
end-else
end

O(d $\log_d$ n) time

Top-Down INSERT

Top-Down INSERT (K,T)

As you follow the search path of K down B-tree T, keep splitting every full node along the way and insert its middle key into its (now non-full) parent.

If the root was full and got split, a new degree 2 root would be created during this process.

if K is found **then return**

else insert K into the (now non-full) leaf you end up.

end

$O(d \log_d n)$ time

Pros and Cons:

- Top-Down Insert tends to over split. It could potentially split O(h) nodes, while the Bottom-Up Insert may not split at all!
- Top-Down makes only one pass down the search path and does not need parent pointers, while Bottom-Up may need to climb up to repair by repeated node splitting.

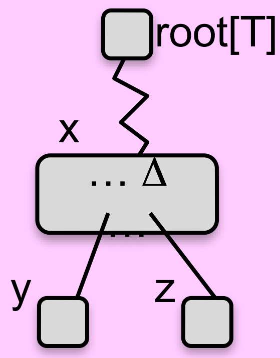
Top-Down DELETE

Top-Down DELETE (K,T)

The idea is to move the current node x down the search path of K in T and maintain: Loop Invariant: d[x] > d or x=root[T].
By LI, x can afford to lose a key.
At the start x = root[T] and LI is maintained.

Case 0: K∉x and x is a leaf. Return

Case 1: K∉x and we have to move to child y of x.
Case 1a: d[y] > d:
x ← y (*LI is maintained *)
Case 1b: d[y] = d, d[z] > d, (z=left/right imm. sib. of y)
KeyShare(y,z);
x ← y (*LI is maintained *)
Case 1c: d[y] = d & d[z]=d separated by key Δ at x:
x ← Fuse(y,Δ,z) (* LI is maintained *)

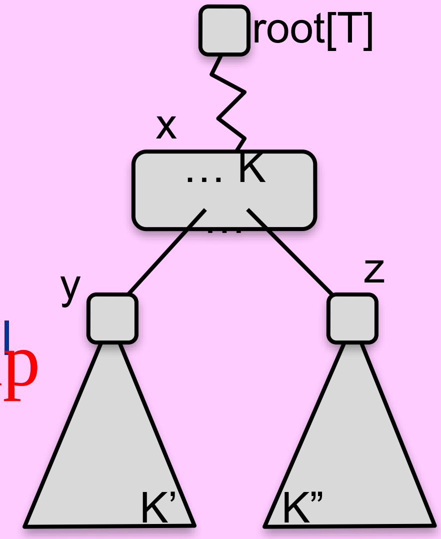


Case 2: K∈x and we have to remove K from x:
see next page

Continued

Top-Down DELETE

Top-Down DELETE (K,T)



Case 2: $K \in x$ and we have to remove K from x:

Case 2a: x is a leaf: (* LI *)
remove K from x
If x=root[T] & becomes empty, set root[T] ← nil

Case 2b: x is not a leaf:
Let y and z be children of x imm. left/right of K

Case 2b1: $d[y] > d$ (* LI *)
Recursively remove predecessor K' of K from subtree rooted at y and replace K in x by K'.

$O(d \log_d n)$ time

Case 2b2: $d[y] = d$, $d[z] > d$ (* LI *)
Recursively remove successor K" of K from subtree rooted at z and replace K in x by K".

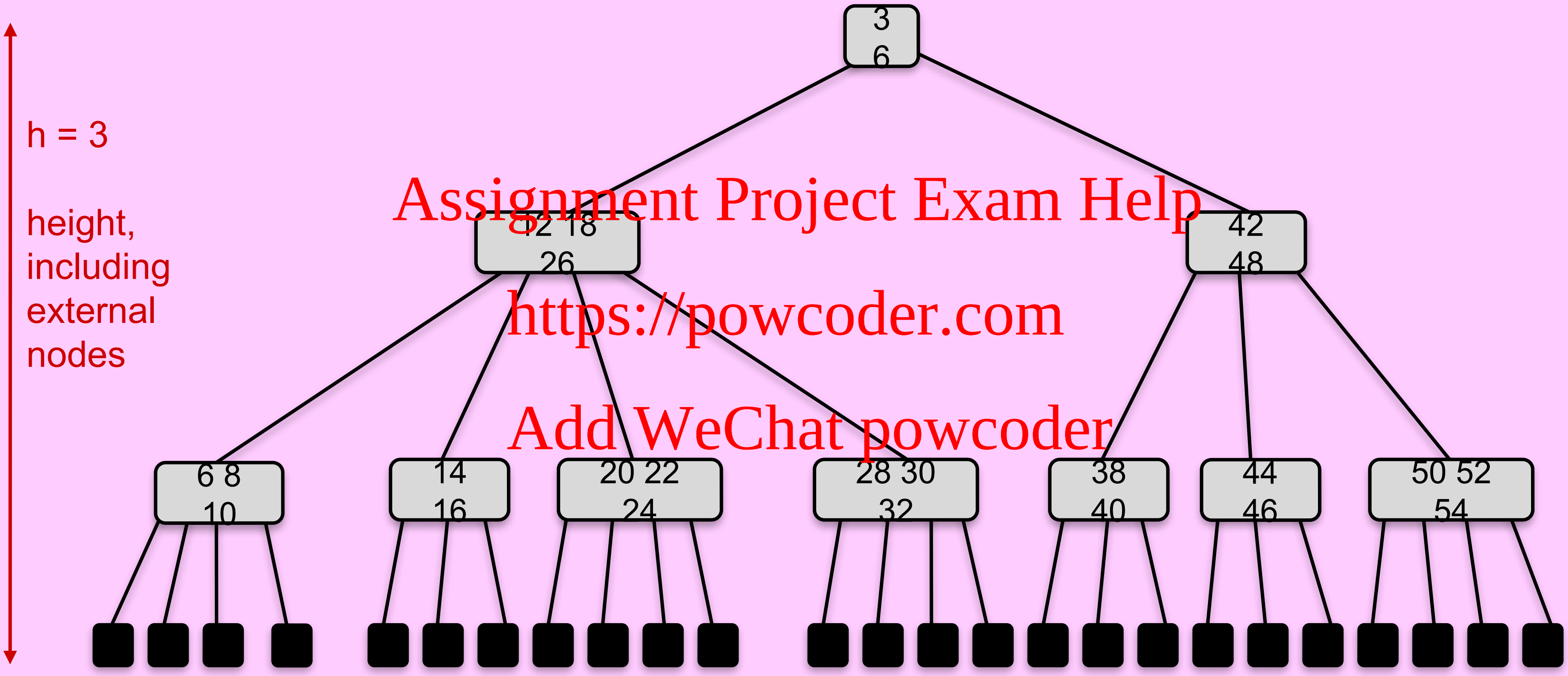
Bottom-Up Delete left as exercise

Case 2b3: $d[y] = d[z] = d$
x ← Fuze(y,K,z) (* LI *)
Repeat Case 2 (one level lower in T).

2-3-4 trees

2-3-4 tree

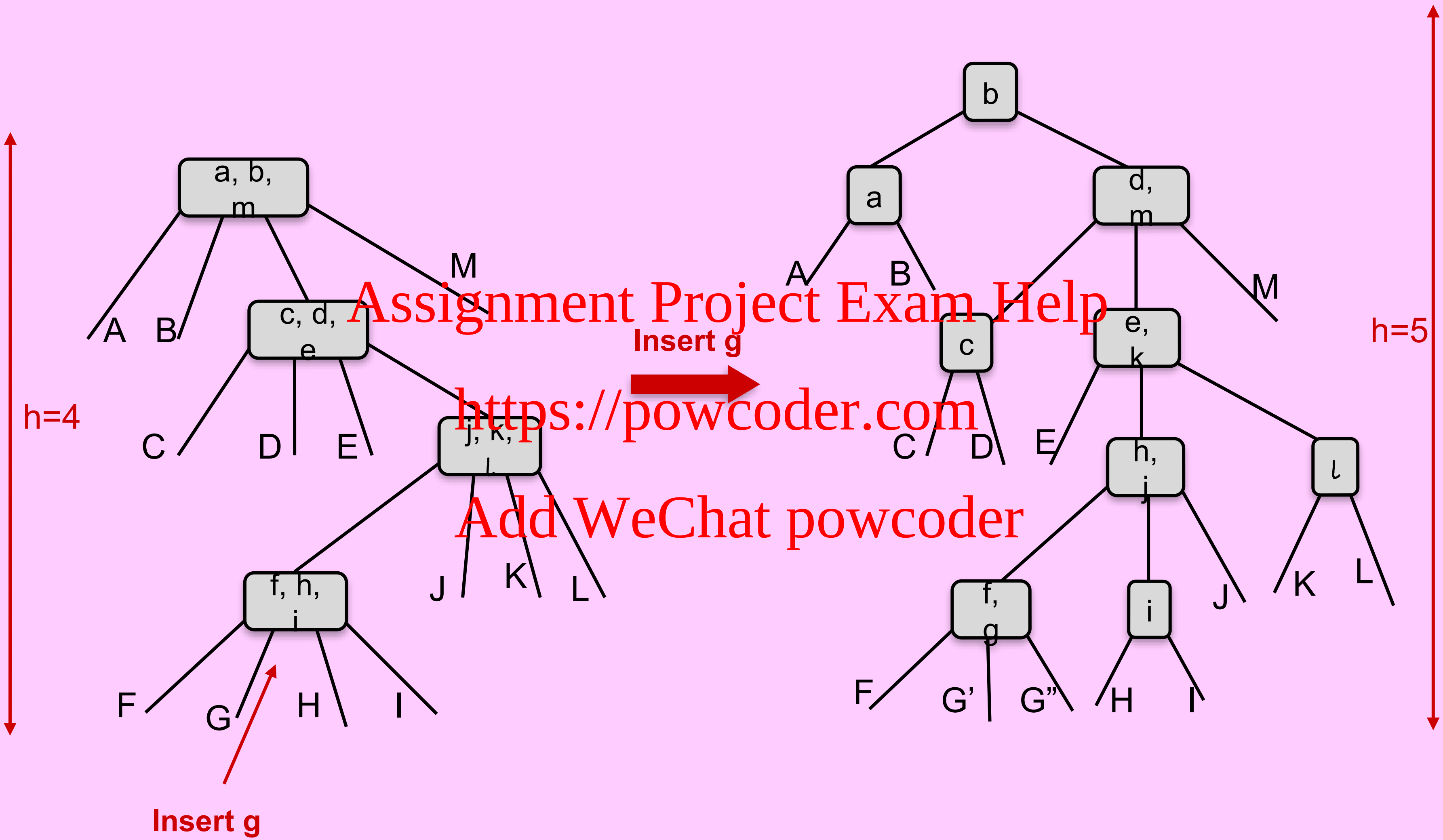
2-3-4 tree is a B-tree of order 2, i.e., a multi-way search tree with 2-nodes, 3-nodes, 4-nodes, and all external nodes at the same depth.



$$\tfrac{1}{2}\log_2(n+1) \le h \le \log_2(n+1).$$

This includes the level of external nodes.

Example: Bottom-Up Insert



Exercises

1. **Bottom-Up Delete:** Design and analyze an efficient implementation of bottom-up Delete on B-trees.

2. **Insert & Delete Examples:** Demonstrate the following processes, each time starting with the B-tree of order 3 example on page 6 of these slides.
   - (a) Insert 1.
   - (b) Delete 48 top-down.
   - (c) Delete 48 bottom-up.

3. **B-tree Insertion sequence:** Draw the B-tree of order d resulting from inserting the following keys, in the given order, using bottom-up insert, into an initially empty tree:

   $\langle 4, 40, 23, 50, 11, 34, 62, 78, 66, 22, 90, 59, 25, 72, 64, 77, 39, 12 \rangle$,
   - (a) with d = 2.
   - (b) with d = 4.

4. **2-3-4 tree Insertion sequence:** Insert integers 1..32 in that order into an initially empty 2-3-4 tree.
   - (a) Show some intermediate snapshots as well as the final 2-3-4 tree.
   - (b) Would you get a different resulting tree with top-down versus bottom-up Insert?
   - (c) Can you describe the general pattern for the insertion sequence 1..n?

   [Hint: any connection with the Binary Counter with the Increment operation?]

5. **Split and Join on 2-3-4 trees:** These are cut and paste operations on dictionaries. The Split operation takes as input a dictionary (a set of keys) A and a key value K (not necessarily in A), and splits A into two disjoint dictionaries $B = \{ x \in A \mid key[x] \le K \}$ and $C = \{ x \in A \mid key[x] > K \}$. (Dictionary A is destroyed as a result of this operation.) The Join operation is essentially the reverse; it takes two input dictionaries A and B such that every key in A < every key in B, and replaces them with their union dictionary $C = A \cup B$. (A and B are destroyed as a result of this operation.) Design and analyze efficient Split and Join on 2-3-4 trees. [Note: Definition of Split and Join here is the same we gave on BST's and slightly different than the one in [CLRS, Problem 18-2, pp: 503-504].]

6. **B*-trees:** A B*-tree T (of order d>0) is a variant of a B-tree. The only difference is the node size range. More specifically, for each non-root node x, $2d \le d[x] \le 3d$
(i.e., at least 2/3 full.)

(a) Specify appropriate lower and upper bounds on d[root[T]] of a B*-tree.

(b) Describe an insertion procedure for B*-trees. What is the running time?
[Hint: Before splitting a node see whether its sibling is full. Avoid splitting if possible.)

(c) What are the advantages of a B*-tree over a standard B-tree?

END