TCS-503: Design and Analysis of Algorithms

Unit II

Advanced Data Structures:

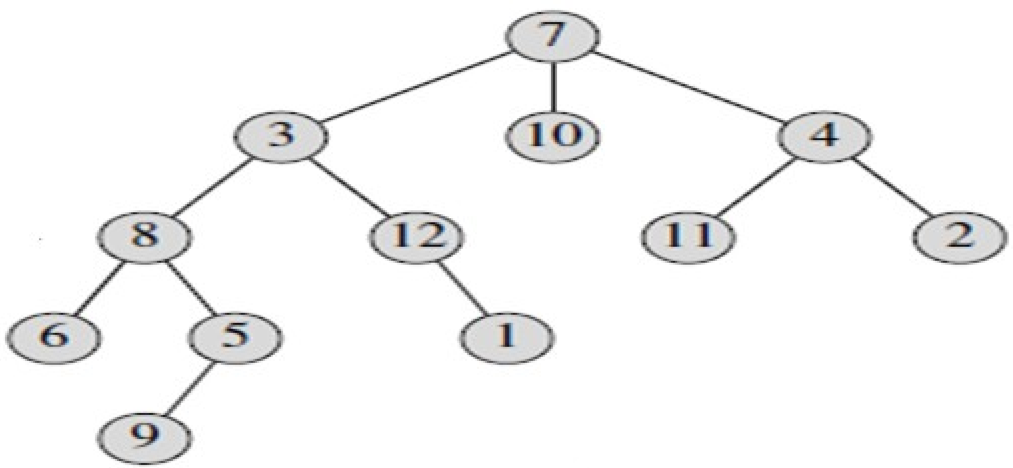
B-Tree

Unit II

- Advanced Data Structures:
  - Red-Black Trees
  - Augmenting Data Structure
  - B-Trees
  - Binomial Heaps
  - Fibonacci Heaps
  - Data Structure for Disjoint Sets

What is rooted tree?

A rooted tree is a free tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the root of the tree.



B-Tree

A B-Tree T is a rooted tree having the following properties:

1. Every node x has the following fields:

   a. n[x], the number of keys currently stored in node x.

   b. the n[x] keys themselves, stored in increasing order, so that

   $$key_1[x] <= key_2[x] <= \ldots\ldots <= key_{n[x]}[x].$$

   c. leaf[x], a Boolean value that is TRUE: if x is a leaf and FALSE if x is an internal node.

n[x]	$key_1[x]$	$key_2[x]$	$key_3[x]$	$key_4[x]$	$key_5[x]$	$key_6[x]$	$key_7[x]$	$key_8[x]$	leaf[x]
$c_1[x]$	$c_2[x]$	$c_3[x]$	$c_4[x]$	$c_5[x]$	$c_6[x]$	$c_7[x]$	$c_8[x]$	$c_9[x]$	

B-Tree

2. Each internal node x also contains $n[x] + 1$ pointers $c_1[x], c_2[x], \ldots\ldots\ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

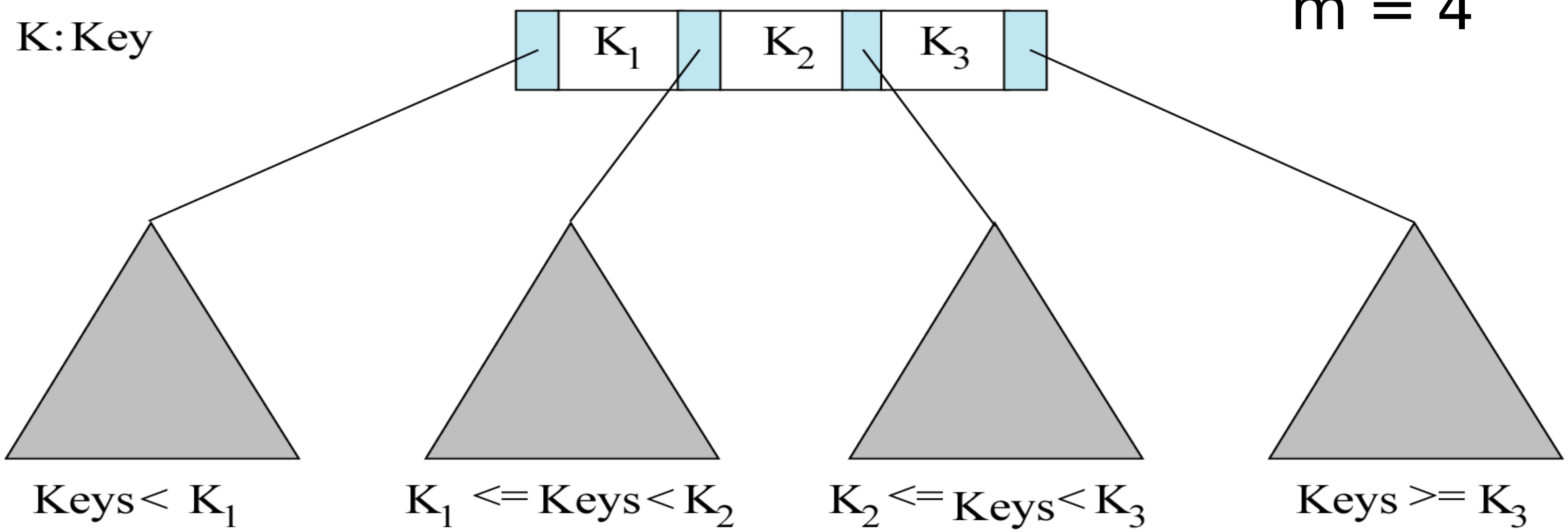
3. The keys $key_i[x]$ separates the ranges of keys stored in each sub-tree: if k_i is any key stored in the sub- tree with root $c_i[x]$, then

$$k_1 <= key_1[x] <= k_2 <= key_2[x] <= \ldots\ldots <= key_{n[x]}[x] <= k_{n[x]+1}$$

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

m-way Search Tree

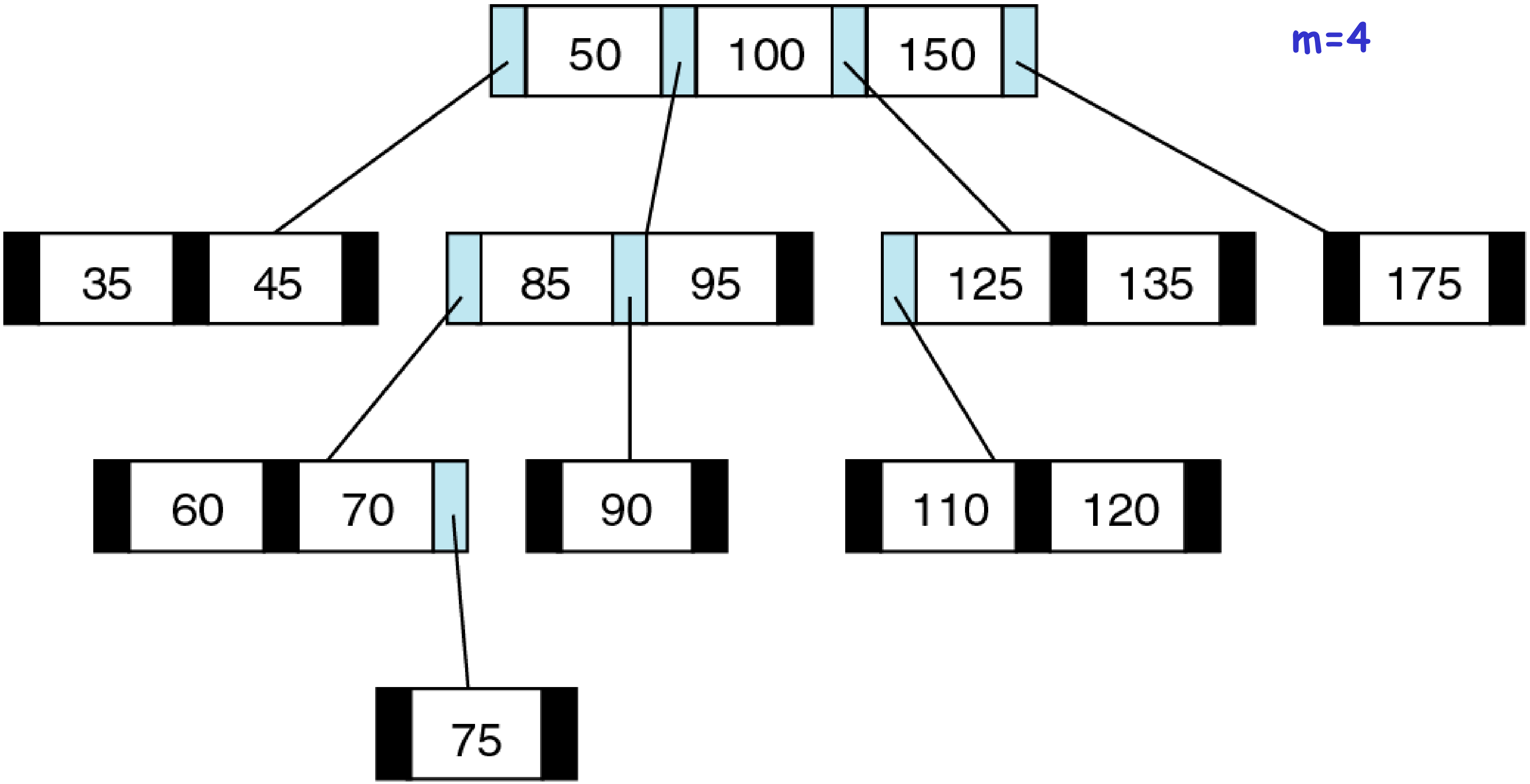
m = 4



It is not balanced!

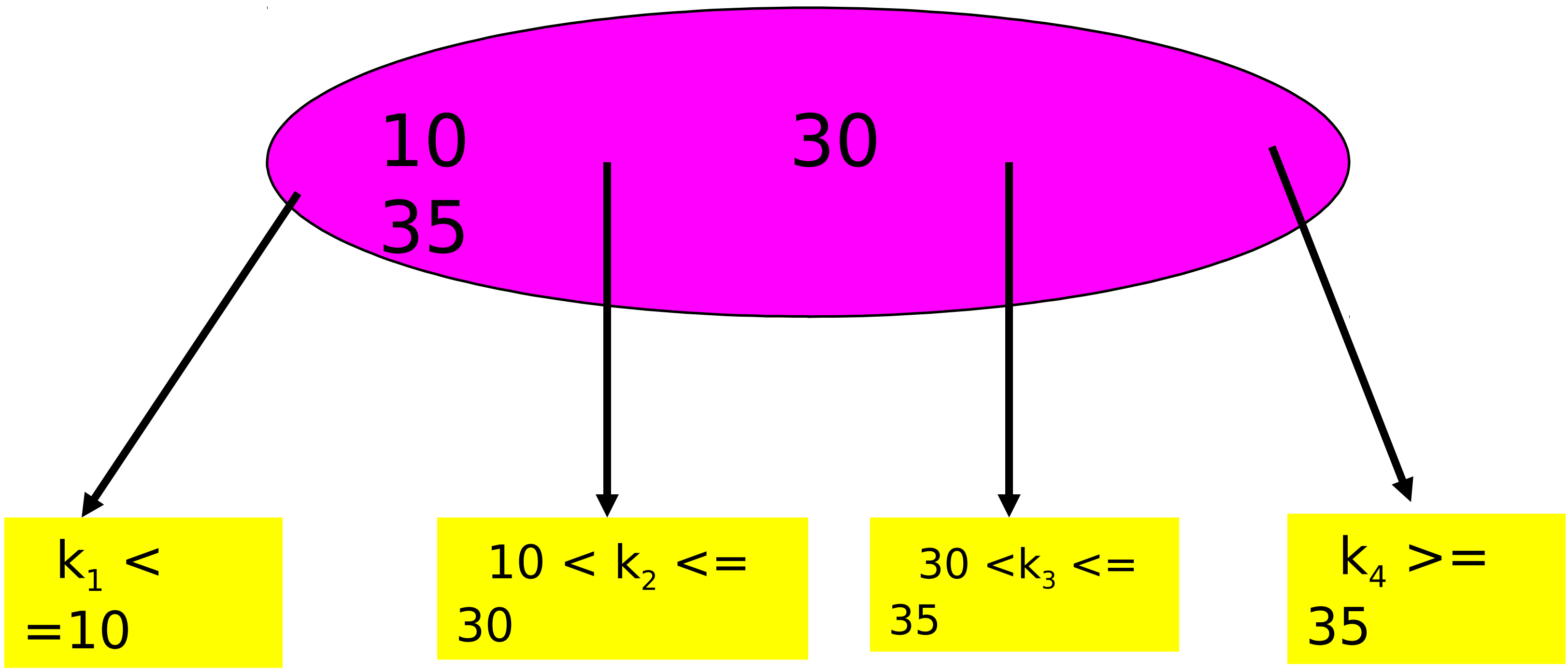
Learn DAA: From B K Sharma

m-way Search Tree



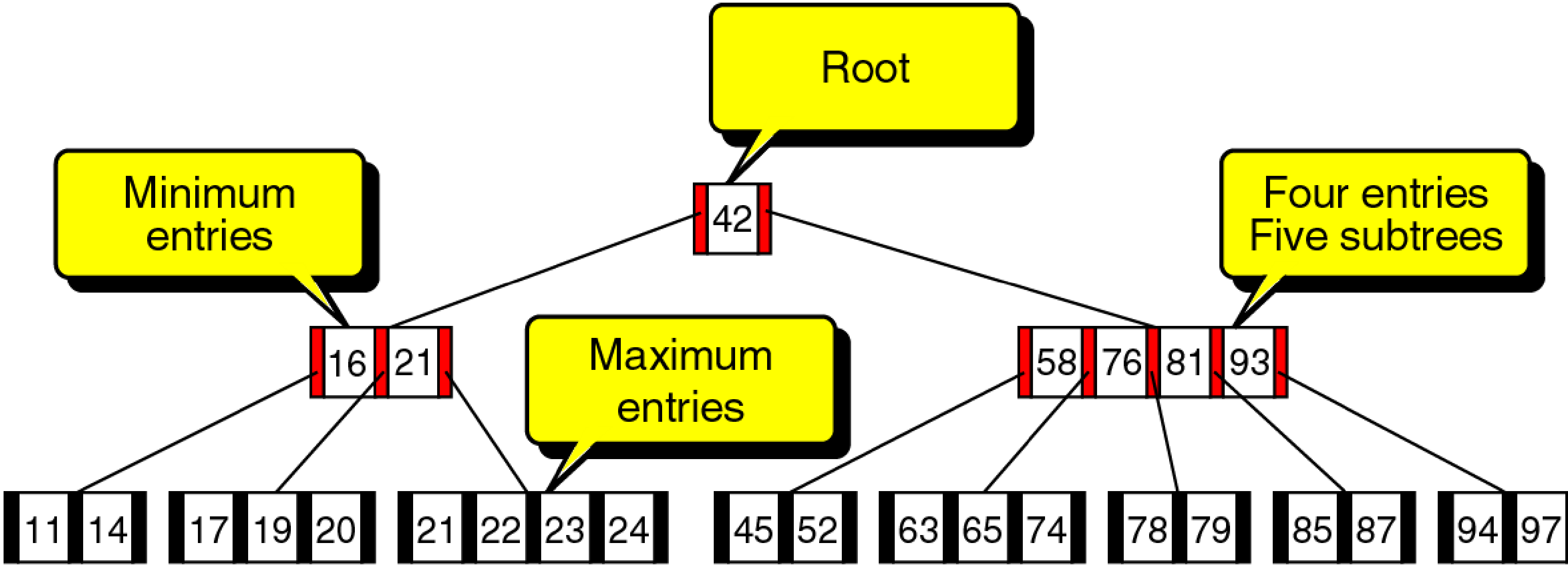
Learn DAA: From B K Sharma

B-Tree: Perfectly balanced (m=) 4-way search tree



Learn DAA: From B K Sharma

B-Tree: Perfectly Balanced m- way search Tree



A B-tree of order; m=5.

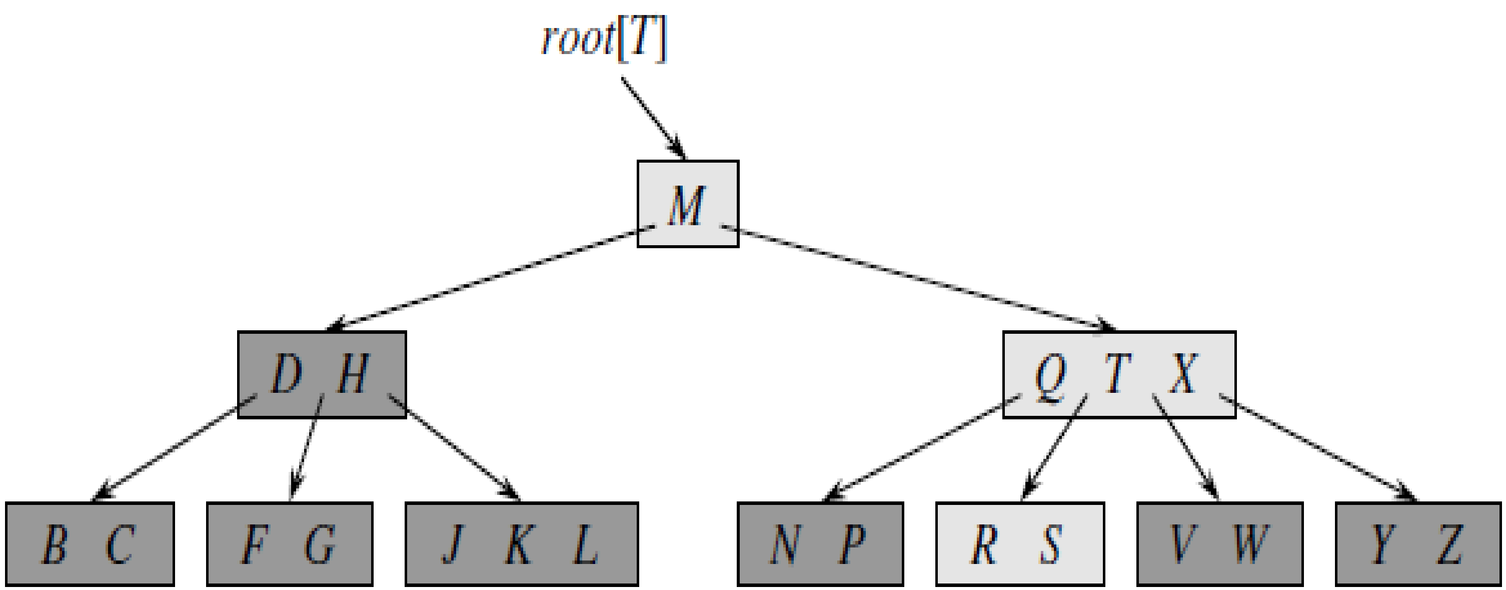
Min entry : Ceiling|5/2| - 1 = 2 entries.

Max entry: 5 – 1 = 4 entries.

Min subtrees : Ceiling|5/2|=3

Max subtrees: 5

B-Tree



An internal node x containing n[x] keys has n[x] + 1 children.

All leaves are at the same depth in the tree.

B-Tree

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:

a. 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 .

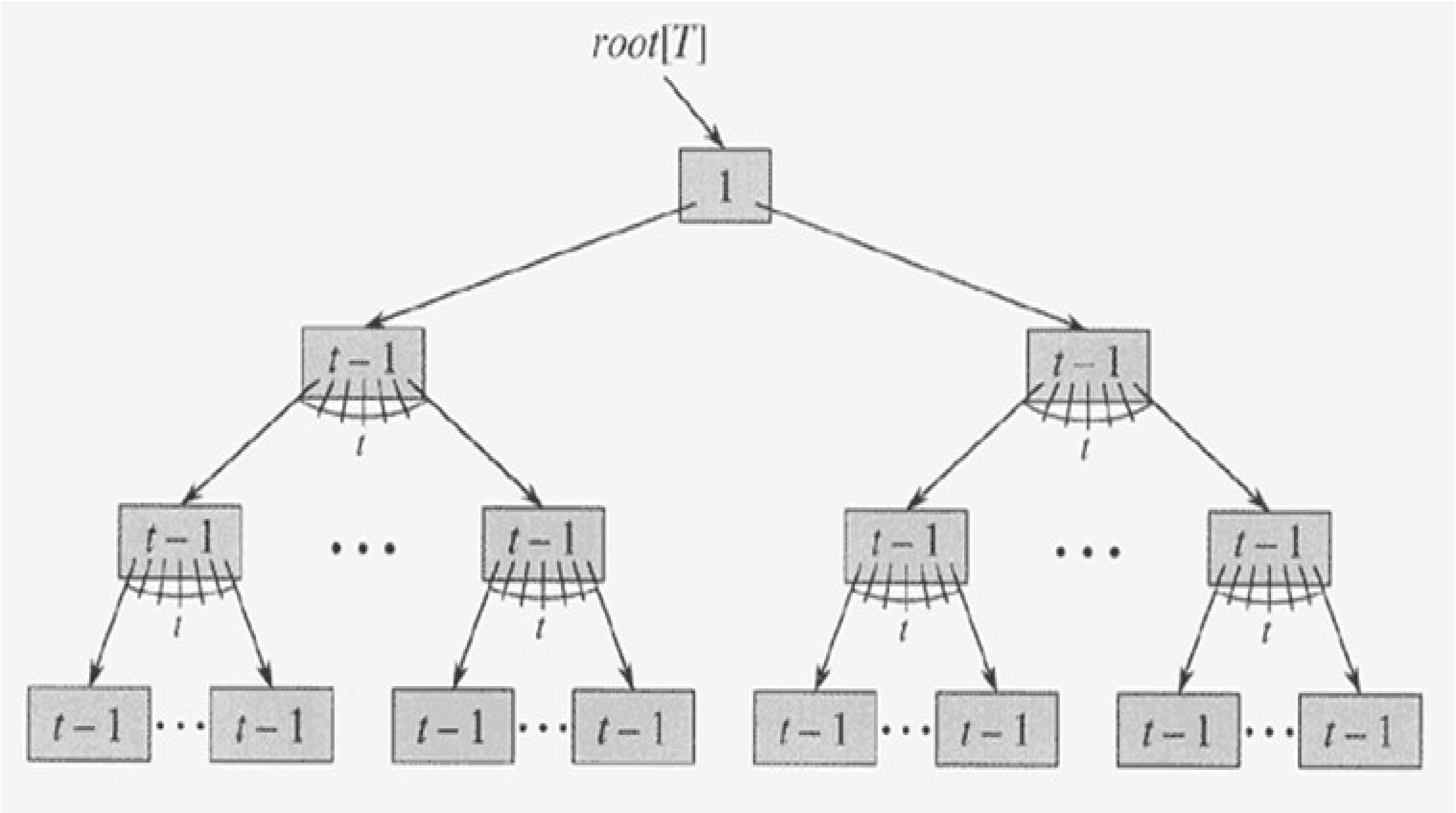
If the tree is non-empty, the root must have at least one key.

b. 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.

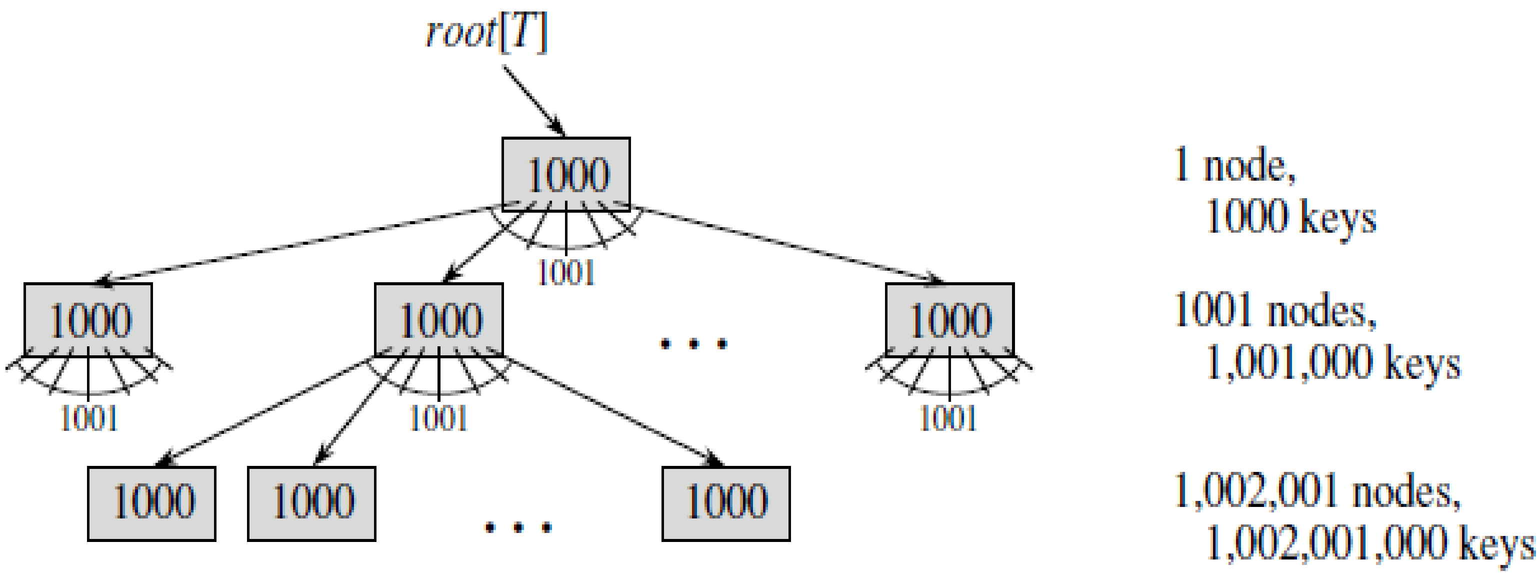
B-Tree



B-Tree of height h=3 containing a minimum possible number of keys.

Shown inside each node *x is n[x].*

B-Tree



A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2.
Shown inside each node x is n[x], the number of keys in x.

B-Tree

The simplest B-Tree occurs when t=2 (Binary Tree).

Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree.***

B-Tree

$x \leftarrow$ a pointer to some object

Basic Operation on B-Tree

x ← a pointer to some object

B-TREE-SEARCH(x, k)

B-TREE-CREATE(T)

B-TREE-INSERT(T, k)

B-TREE-DELETE(x , k)

Search Operation on B-Tree

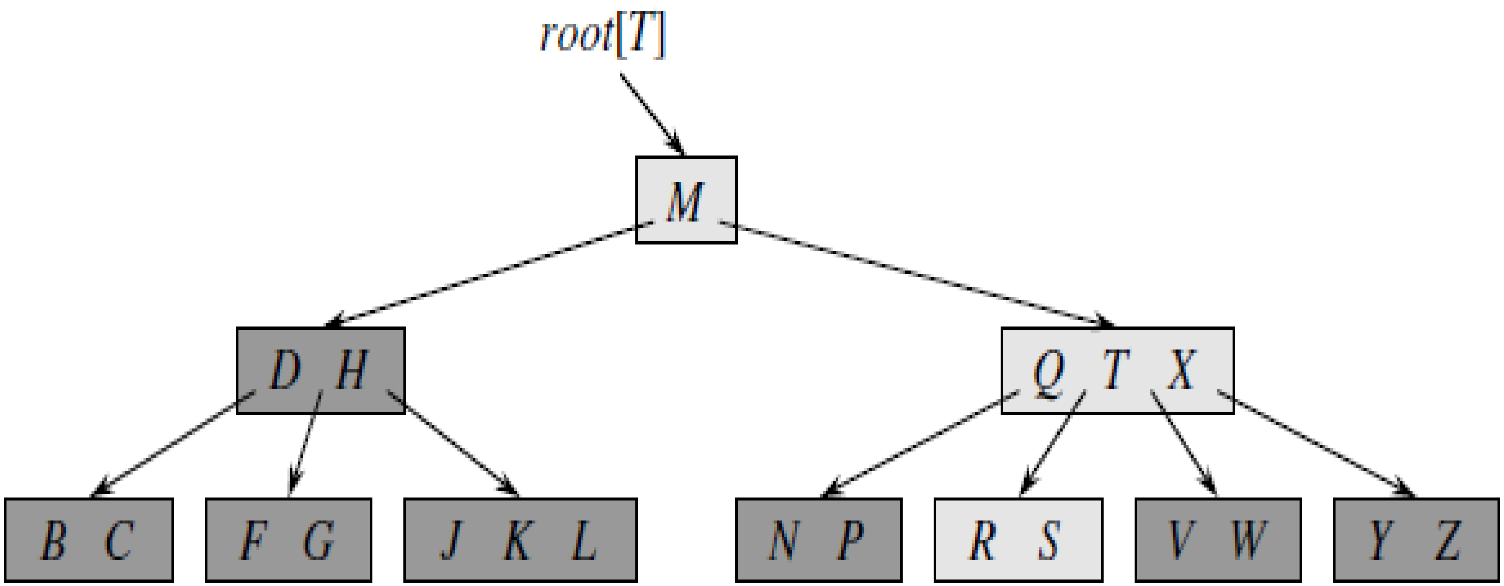
B-TREE-SEARCH(root[T], R)

Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way," branching decision at each node, we make a multi-way branching decision according to the number of the node's children.

More precisely, at each internal node *x, we make an (n[x] + 1)-way branching decision.*

Search Operation on B-Tree

B-TREE-SEARCH(root[T], R)



Insertion Operation on B-Tree

We insert the new key into an existing leaf node.

Since we cannot insert a key into a leaf node that is full, we introduce an operation that *splits a full node y* (having 2*t* – 1 *keys) around its median key* $key_t[y]$ *into two nodes having t – 1* keys each.

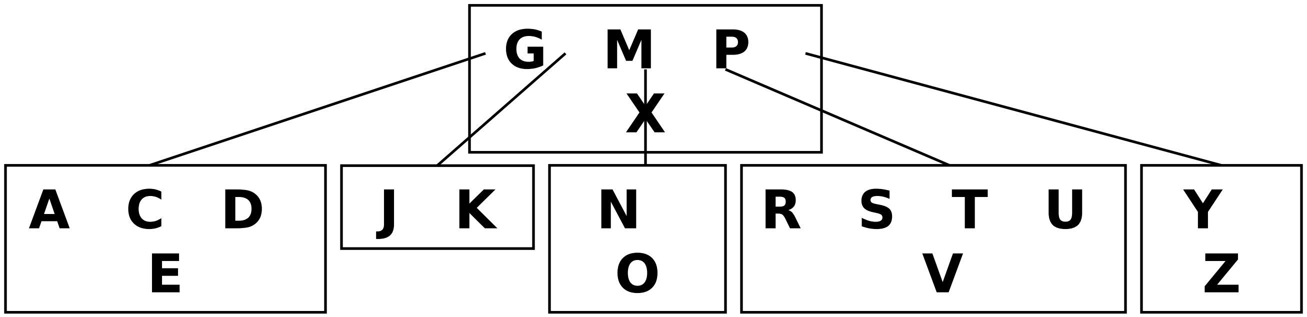
The median key moves up into *y's parent to identify the dividing point* between the two new trees.

But if *y's parent is also full, it must be split before the* new key can be inserted, and thus this need to split full nodes can propagate all the way up the tree.

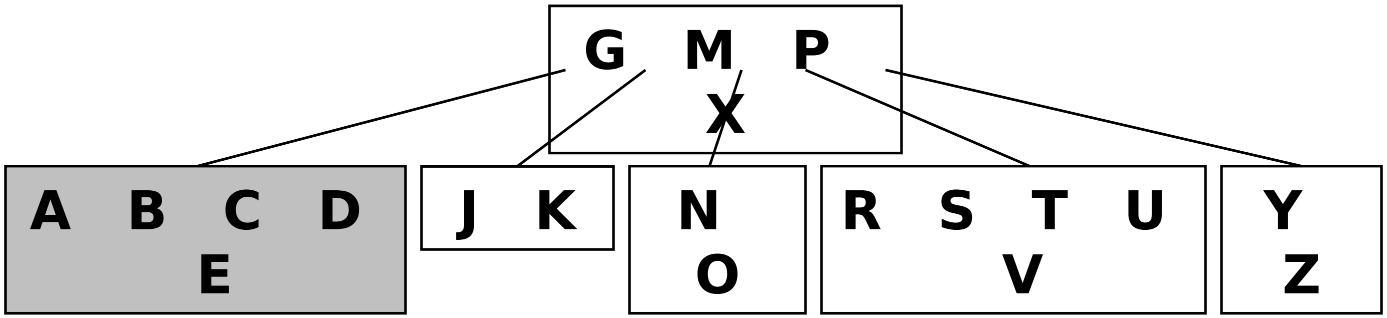
Insertion Operation on B-Tree

The minimum degree *t for this B-tree is 3 (t=3), so a node can* hold at most 5 keys.

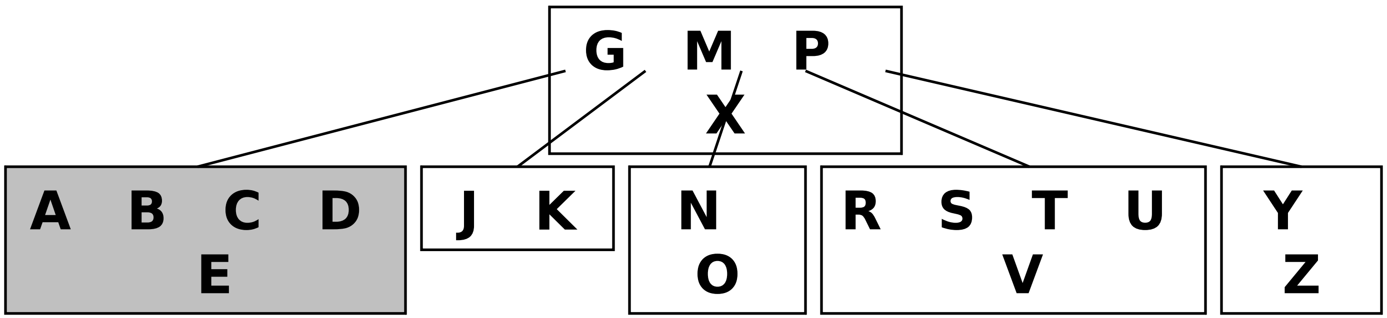
(a) Initial Tree



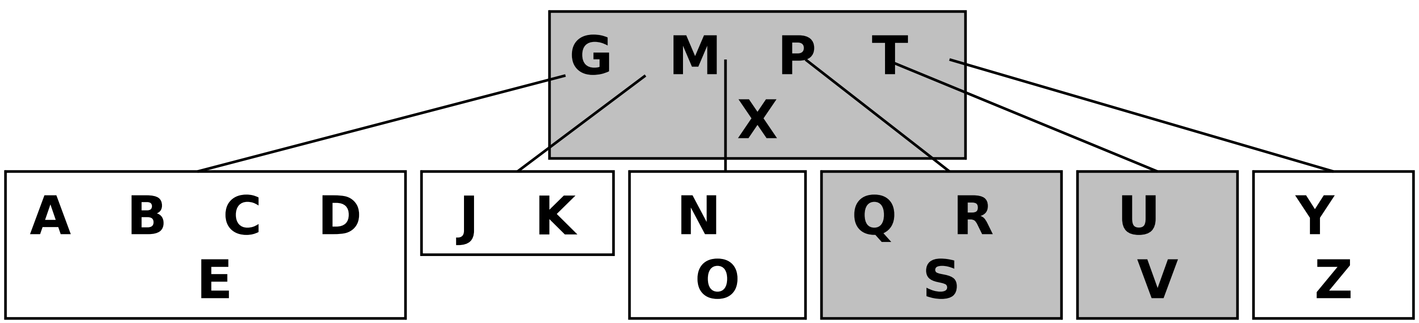
(b) Insert B



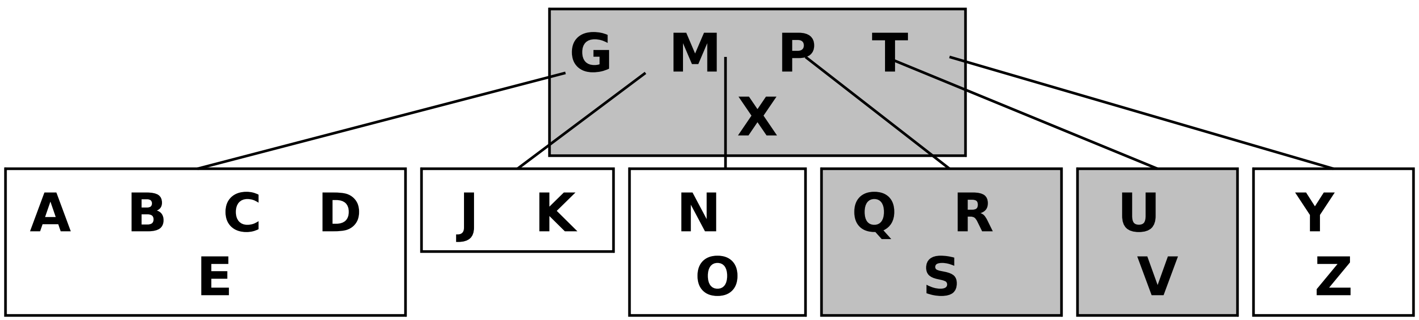
Insertion Operation on B-Tree



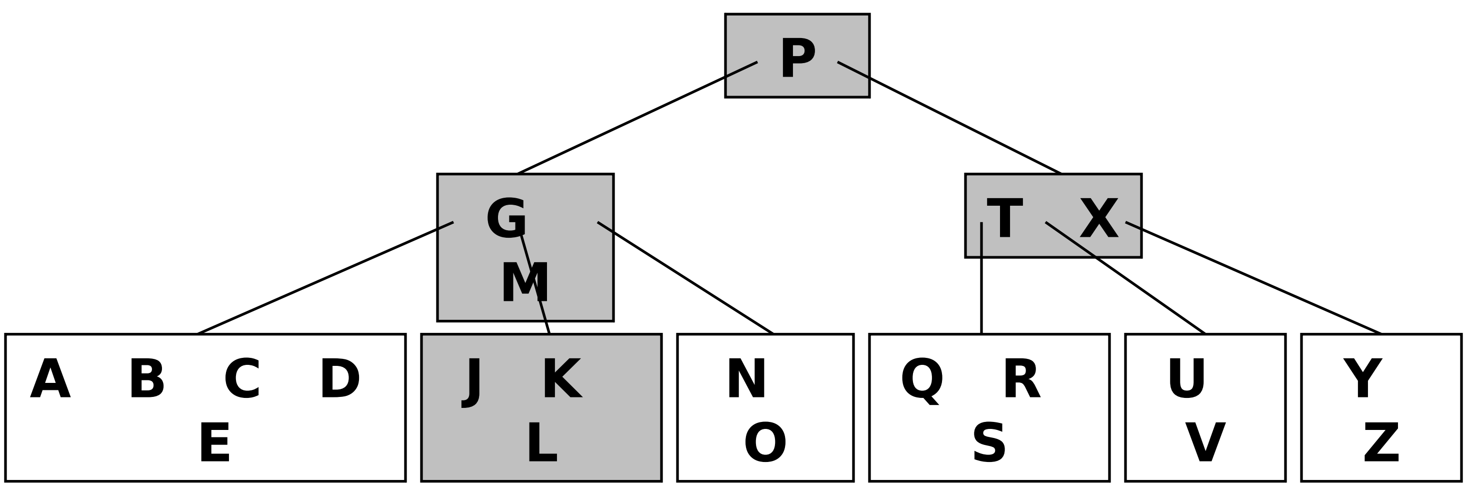
(c) Insert Q



Insertion Operation on B-Tree

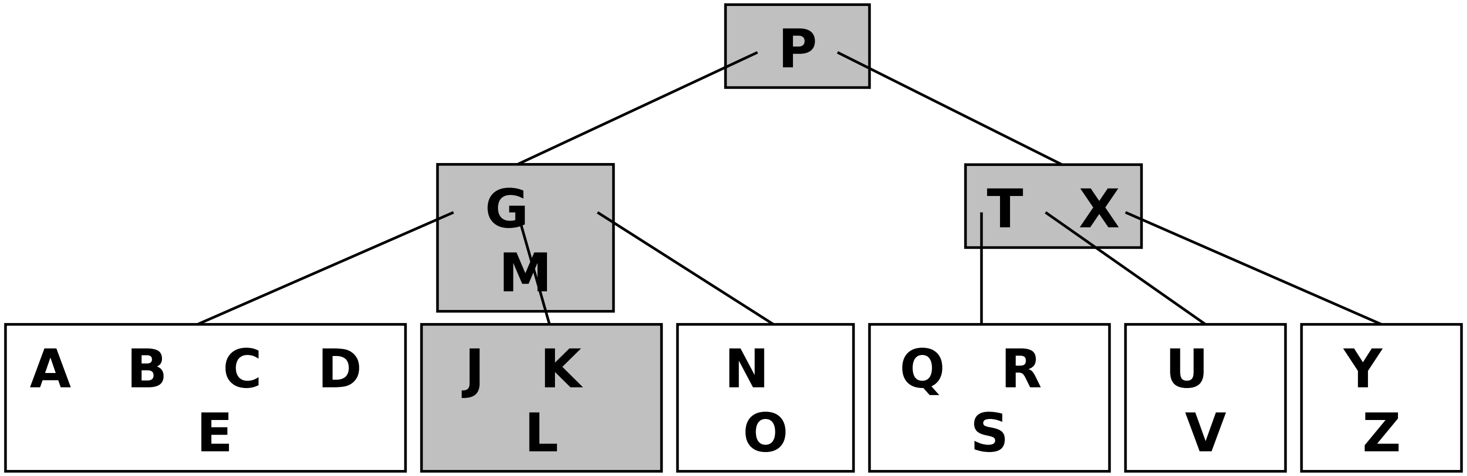


(d) Insert L

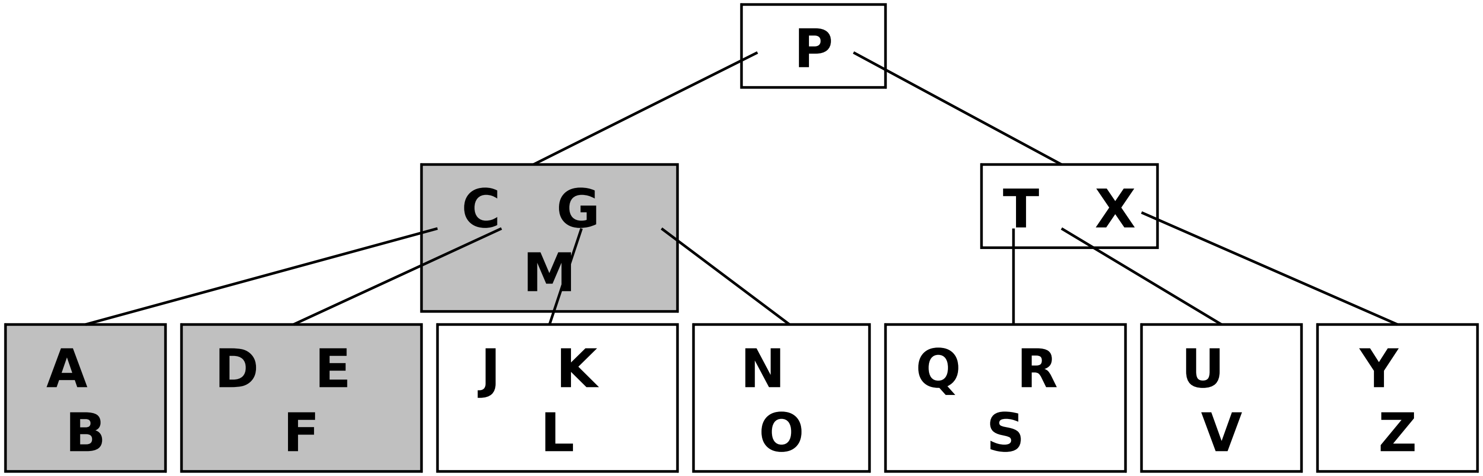


Learn DAA: From B K Sharma

Insertion Operation on B-Tree



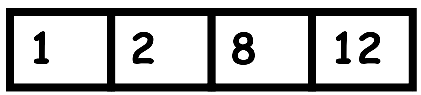
(e) Insert F



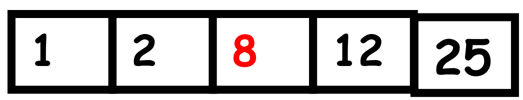
Insertion Operation on B-Tree

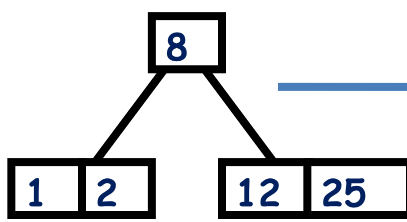
Insert the following keys into an empty B-Tree of order t=5 in order:1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

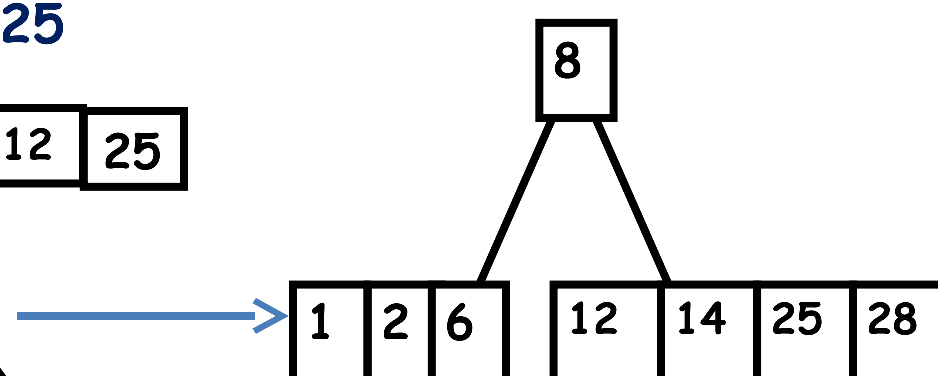
The first four items go into the root:



Insert 25



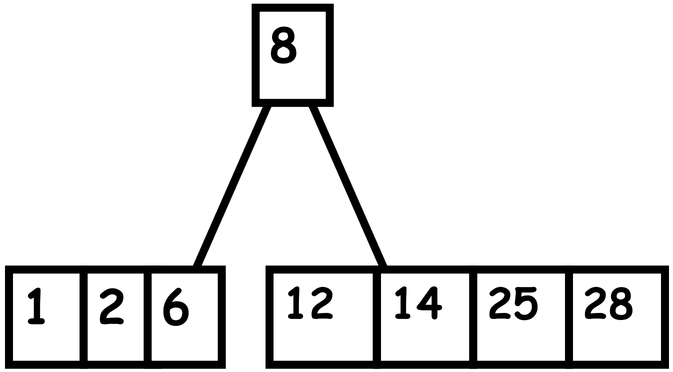




6, 14, 28 get added to the leaf nodes:

Insertion Operation on B-Tree

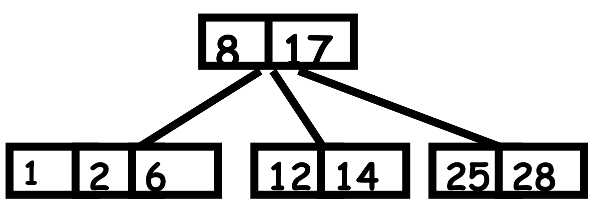
1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Insert 17

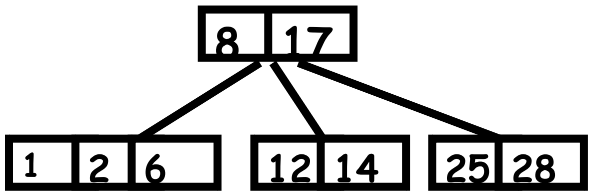
Inserting 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

12,14,17,25,28

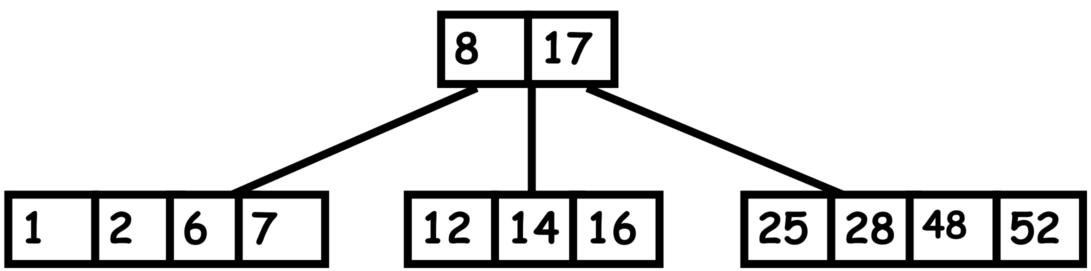


Insertion Operation on B-Tree

1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



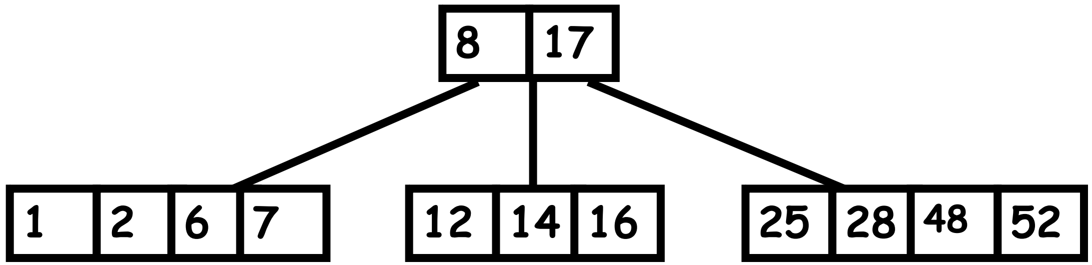
7, 52, 16, 48 get added to the leaf nodes



Learn DAA: From B K Sharma

Insertion Operation on B-Tree

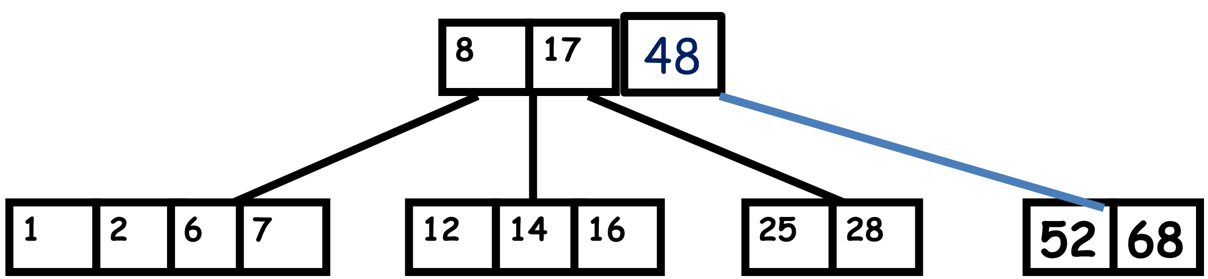
1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Insert 68

Inserting 68 causes us to split the right most leaf, promoting 48 to the root:

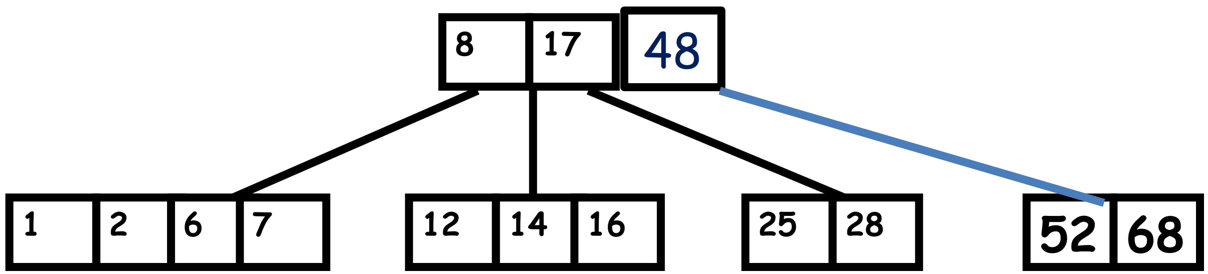
25, 28, 48, 52,68



Learn DAA: From B K Sharma

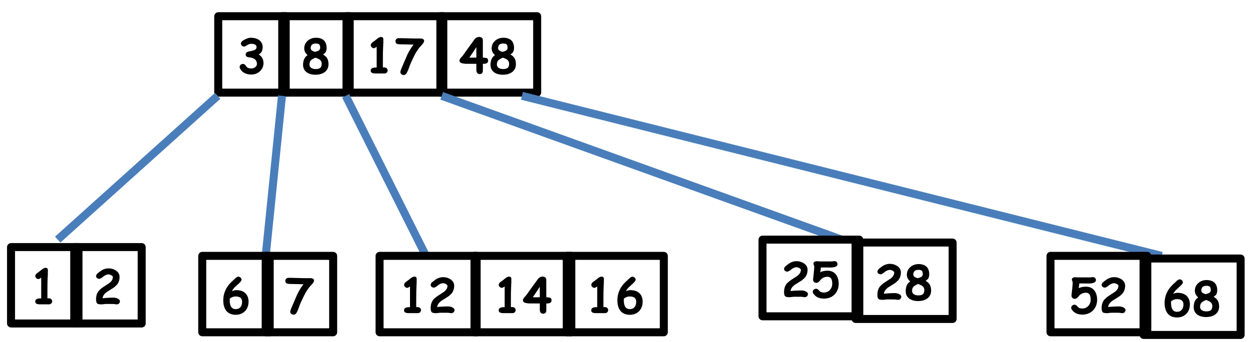
Insertion Operation on B-Tree

1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



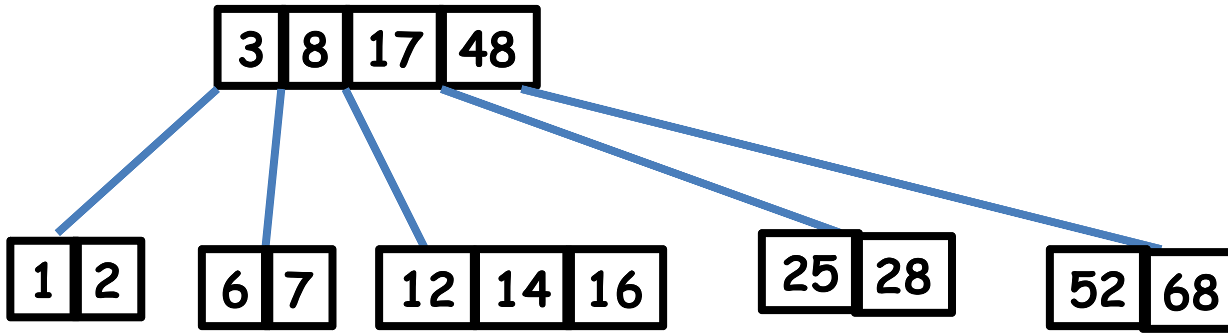
Insert 3

Insertig 3 causes us to split the left most leaf, promoting 3 to the root: 1,2,3,6,7

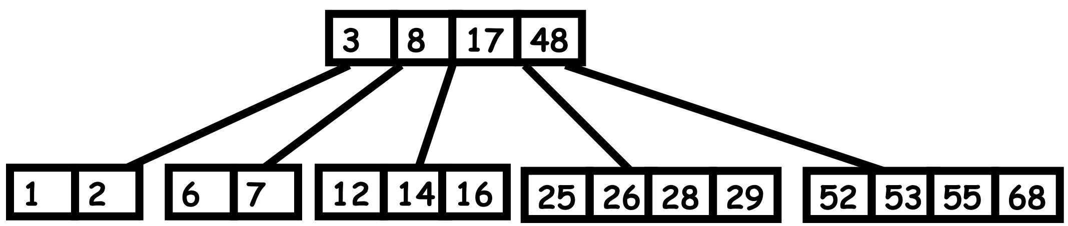


Insertion Operation on B-Tree

1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

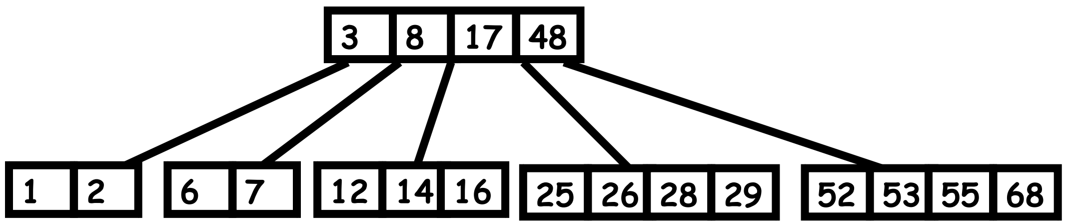


26, 29, 53, 55 then go into the leaves:



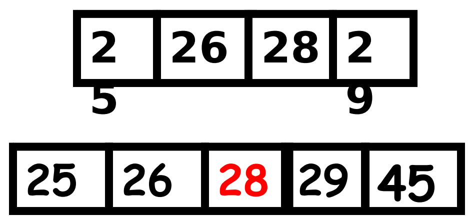
Insertion Operation on B-Tree

1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

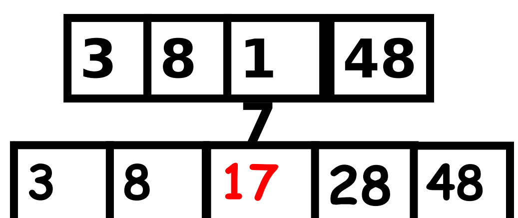


Insert 45

Inserting 45 causes a split of



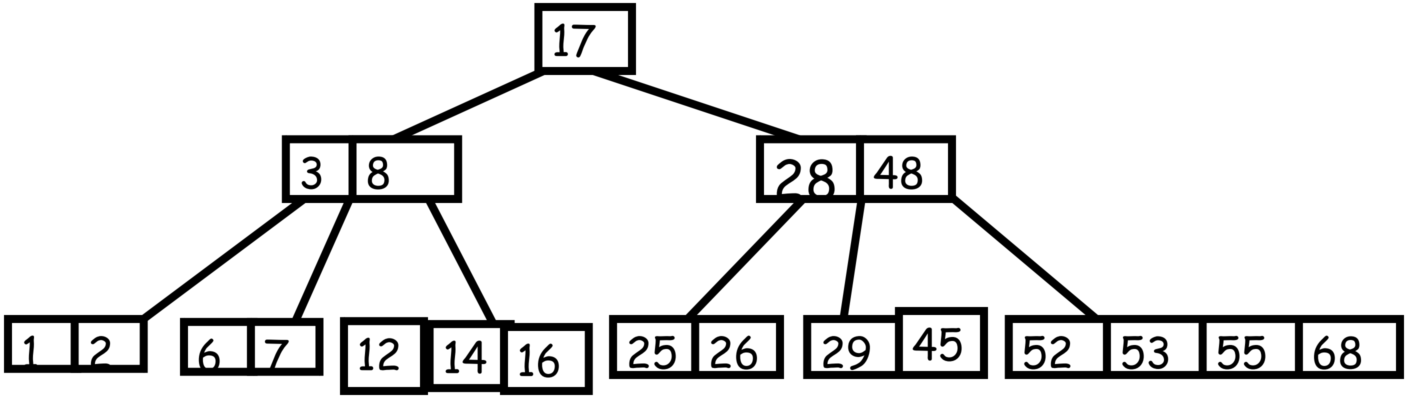
and promoting 28 to the root



Learn DAA: From B K Sharma

Insertion Operation on B-Tree

1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45



Exercise in Inserting a B-Tree

Insert the following keys to a 5-way B-tree:

3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Show the results of inserting the keys

F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E

in order into an empty B-tree with minimum degree 2.

Only draw the configurations of the tree just before some node must split, and also draw the final configuration.

Deletion in a B-Tree

B-tree Delete distinguishes three different stages/scenarios for deletion:

Case 1: key *k* found in leaf node

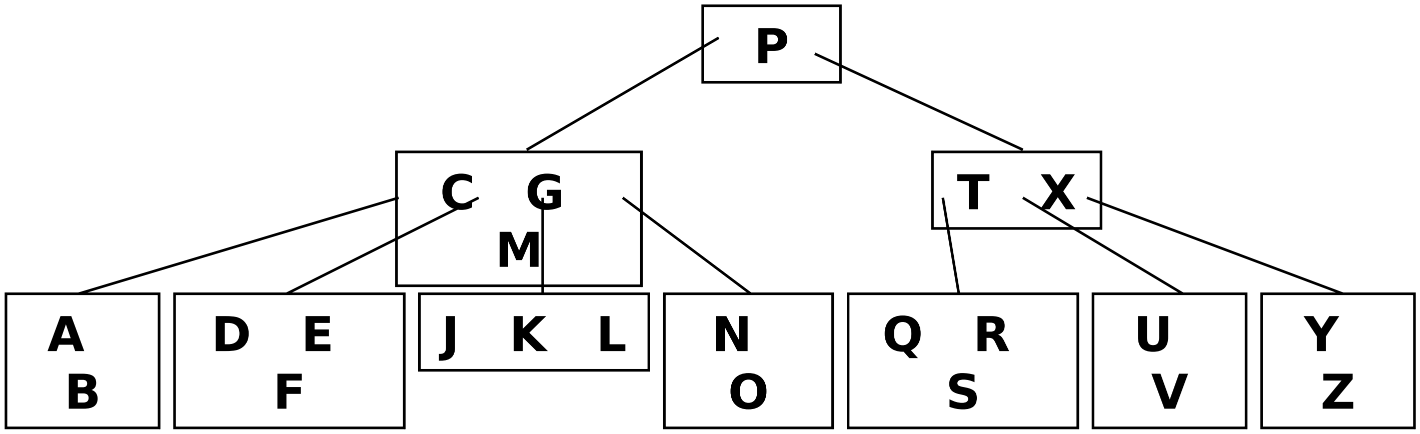
Case 2: key *k* found in internal node

Case 3: key *k* suspected in lower level node

Deletion in a B-Tree

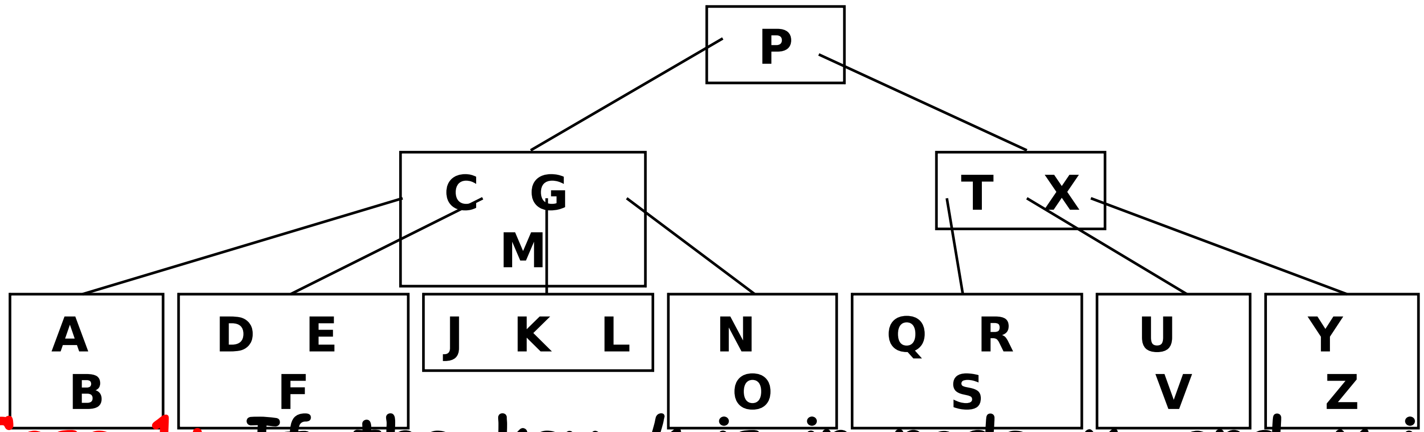
The minimum degree for this B-tree is $t = 3$, *so a node* (other than the root) cannot have fewer than 2 keys.

Initial Tree



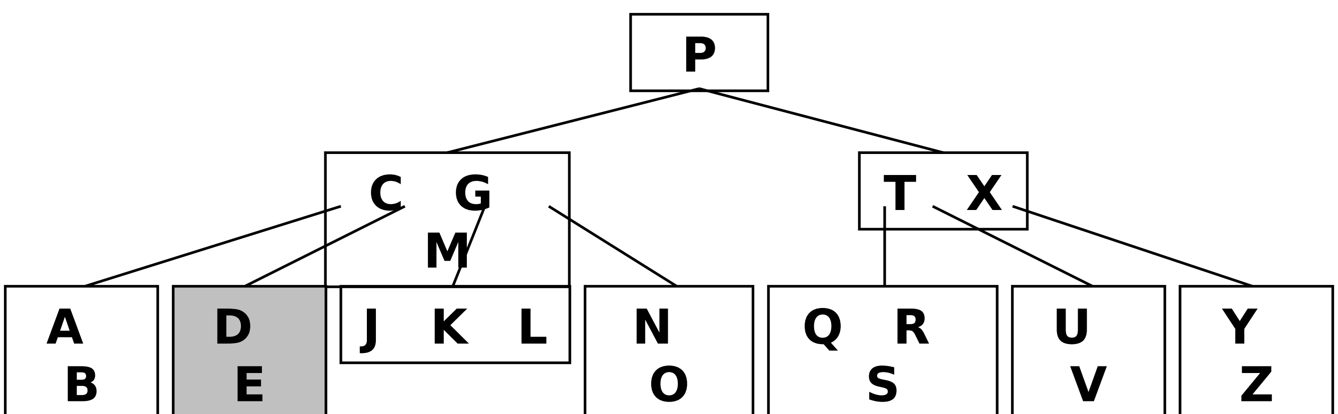
Deleting Keys

Initial Tree



Case 1: If the key *k* is in node *x*, and *x* is a leaf, delete *k* from *x*

Delete F



Deleting Keys

Case 2: If the key *k* is in node *x*, and *x* is not a leaf, delete *k* from *x*

a) If the child *y* that precedes *k* in node *x* has at least *t* keys, then find the predecessor *k*' of *k* in the sub-tree rooted at *y*. Recursively delete *k*', and replace *k* with *k*' in *x*. (Finding *k and deleting it can be performed in a single* downward pass.)

b) Symmetrically, if the child *z that follows k in node x has at least t keys, then* find the successor *k of k in the subtree rooted at z. Recursively delete k,* and replace *k by k in x. (Finding k and deleting it can be performed in a* single downward pass.)

Deleting Keys

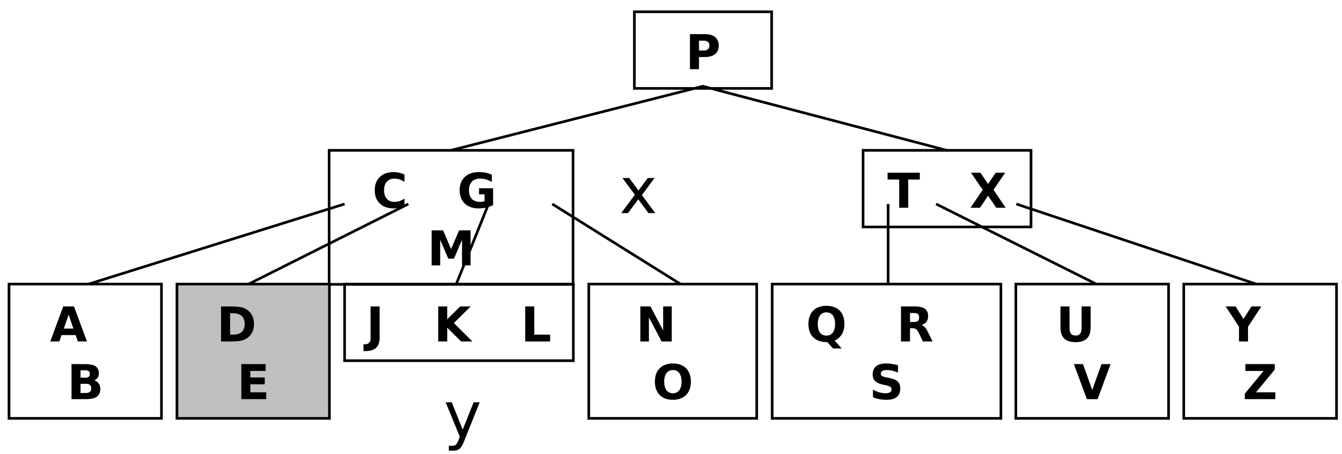
c) Otherwise, if both *y and z have only t − 1 keys, merge k and all of z into y,* so that *x loses both k and the pointer to z, and y now contains 2t − 1 keys.* Then, *free z and recursively delete k from y.*

Deleting Keys

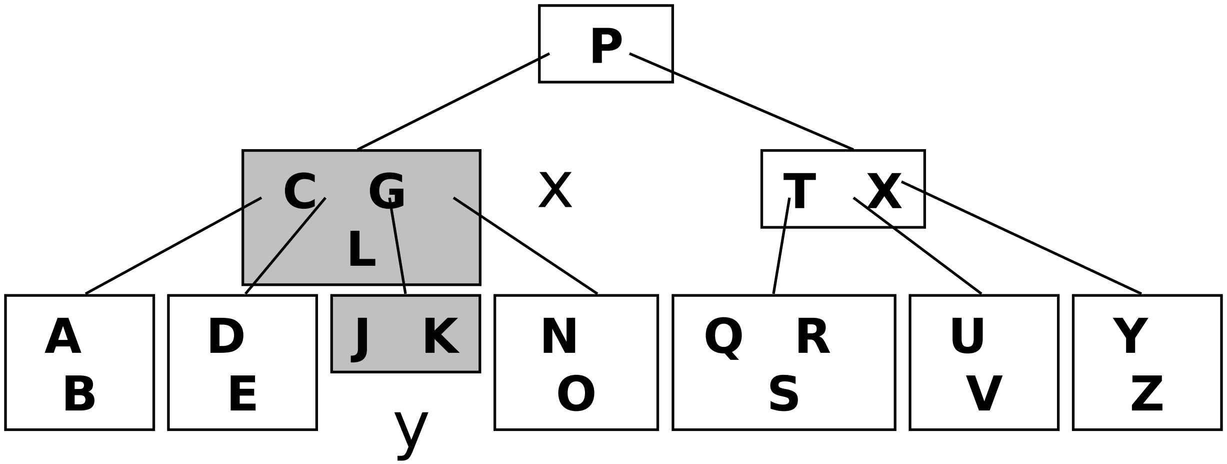
Case 2a: If the child *y* that precedes *k* in node *x* has at least *t* keys, then find the predecessor *k'* of *k* in the sub-tree rooted at *y*. Recursively delete *k'*, and replace *k* with *k'* in *x*.

Delete M

This is case 2a: the predecessor *L of M is moved up to take M's position.*

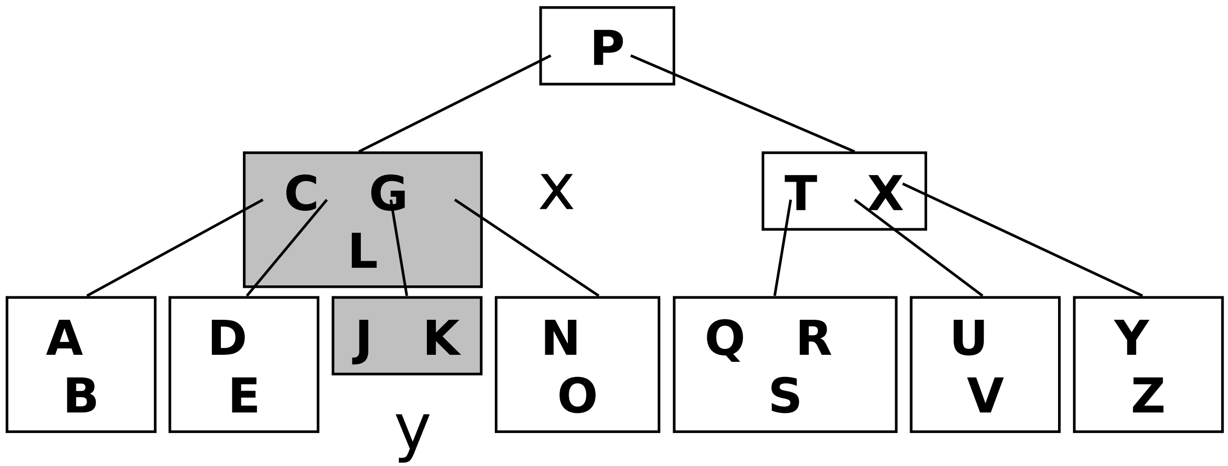


Deleting Keys



Deleting Keys

Delete *G*.

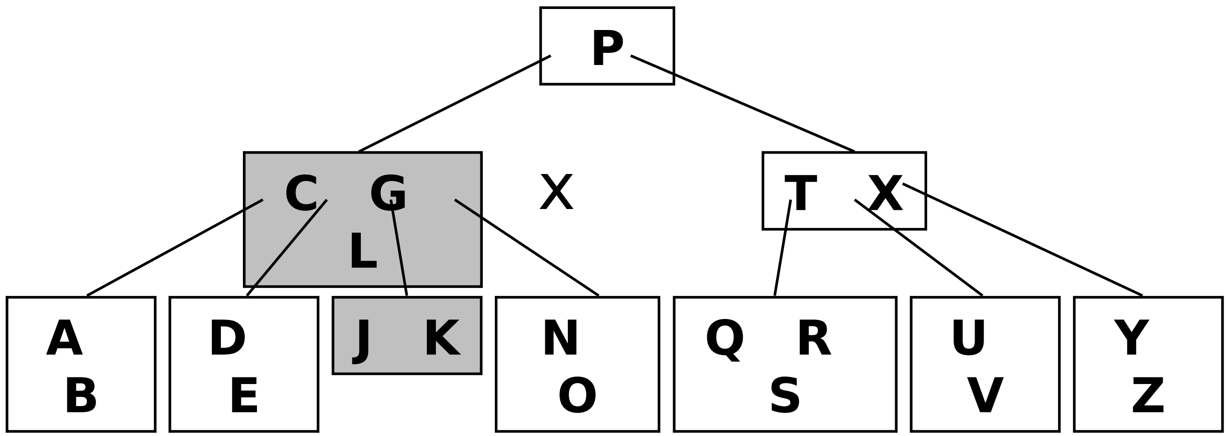


This is case 2c: if both *y and z have only t – 1 keys, merge k and all of z into y*, so that *x loses both k and the pointer to z, and y now contains 2t – 1 keys.* Then, *free z and recursively delete k from y.*

Deleting Keys

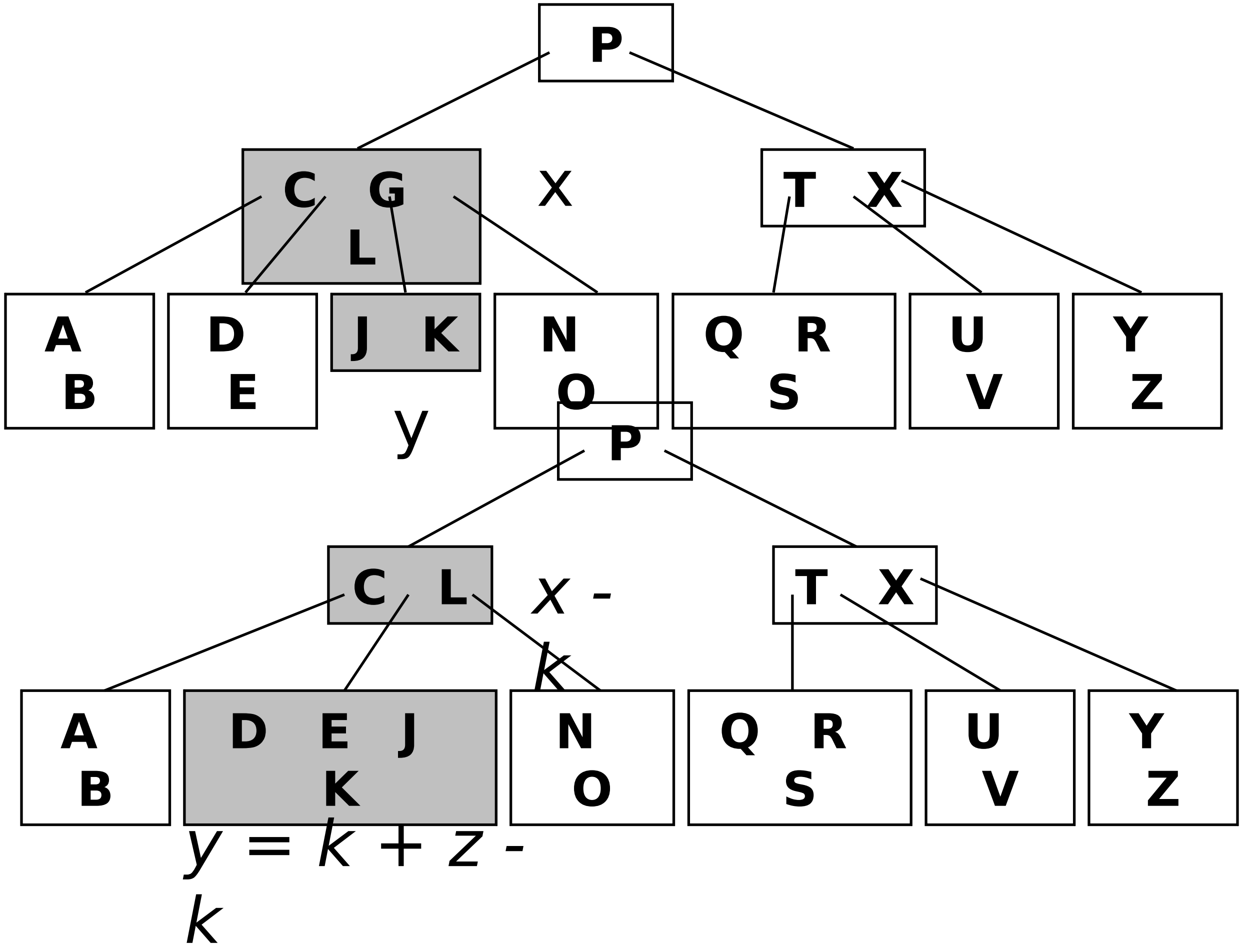
Delete *G*.

G is pushed down to make node DEGJK, and then G is deleted from this leaf (case 1).



Deleting Keys

Delete G.



Deleting Keys

Case 3. If the key *k is not present in internal node x, determine the root* $c_i[x]$ *of the* appropriate subtree that must contain *k, if k is in the tree at all. If* c_i *[x] has only* $t - 1$ *keys, execute step 3a or 3b as necessary to guarantee that we descend to a* node containing at least *t keys. Then, finish by recursing on the appropriate* child of *x.*

Deleting Keys

a. If $c_i[x]$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.

b. If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t - 1$ keys, merge $c_i[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Deleting Keys: Distribution

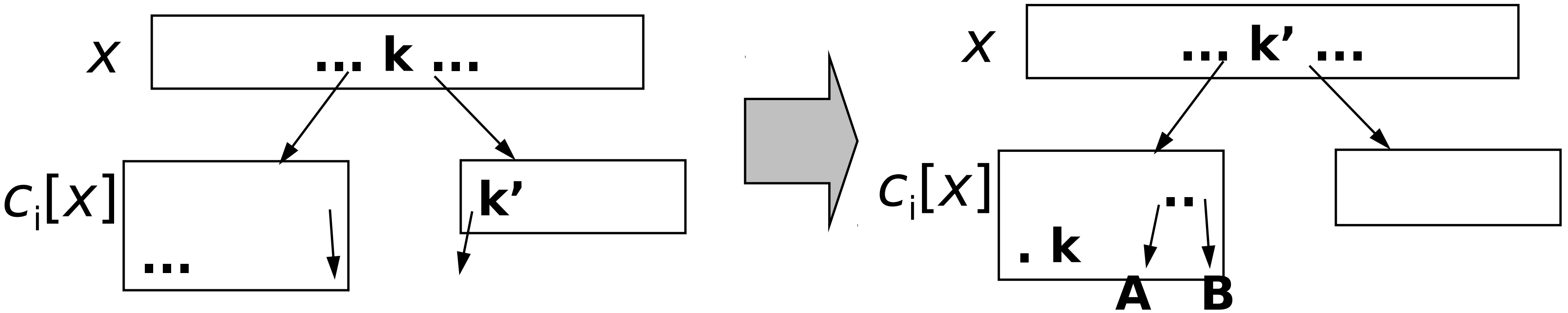
Descending down the tree: if k not found in current node x, find the sub-tree $c_i[x]$ that has to contain k.

If $c_i[x]$ has only $t - 1$ keys take action to ensure that we descent to a node of size at least t.

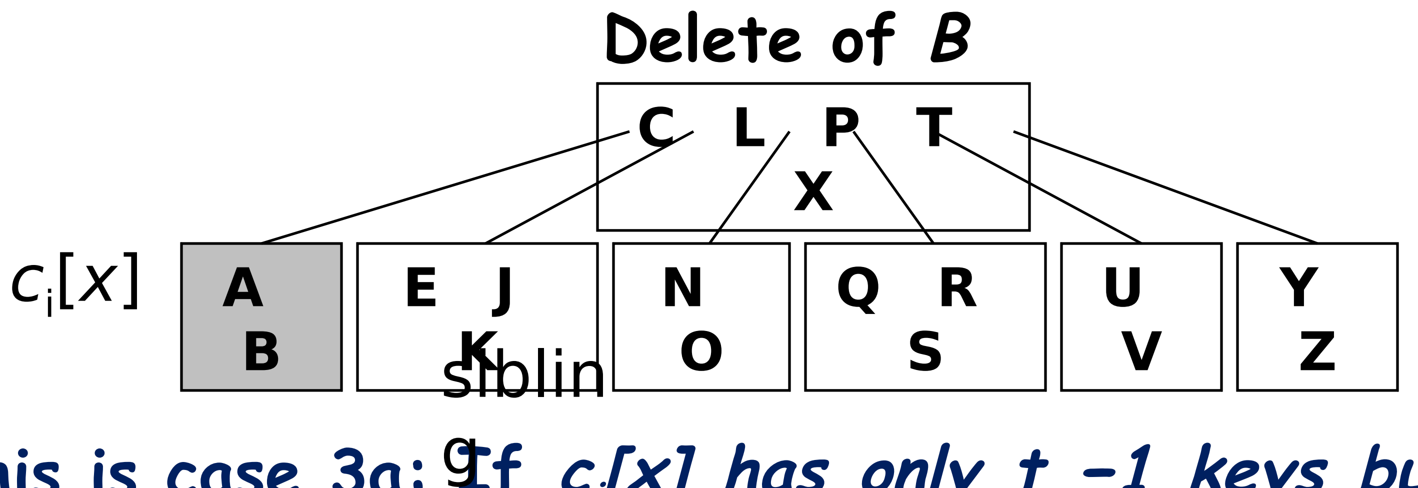
We can encounter two cases.

If $c_i[x]$ has only t-1 keys, but a sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x to $c_i[x]$, moving a key from $c_i[x]$'s immediate left and right sibling up into x, and moving the appropriate child from the sibling into $c_i[x]$ - *distribution*

Deleting Keys: Distribution



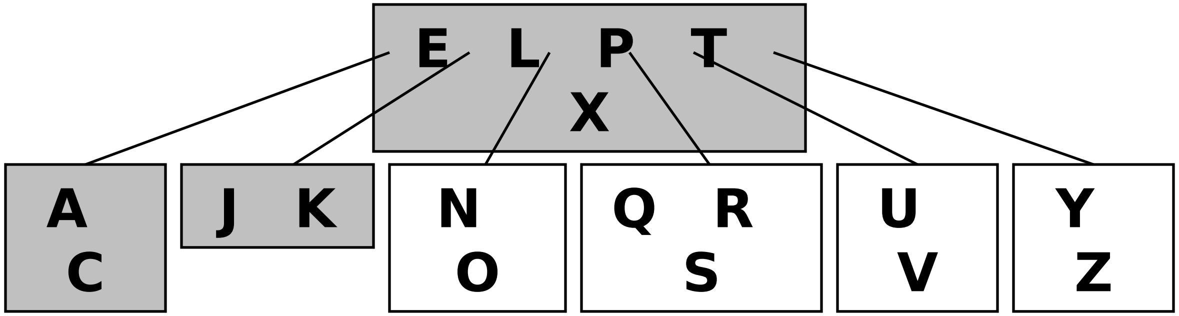
Deleting Keys



This is case 3a: *If $c_i[x]$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.*

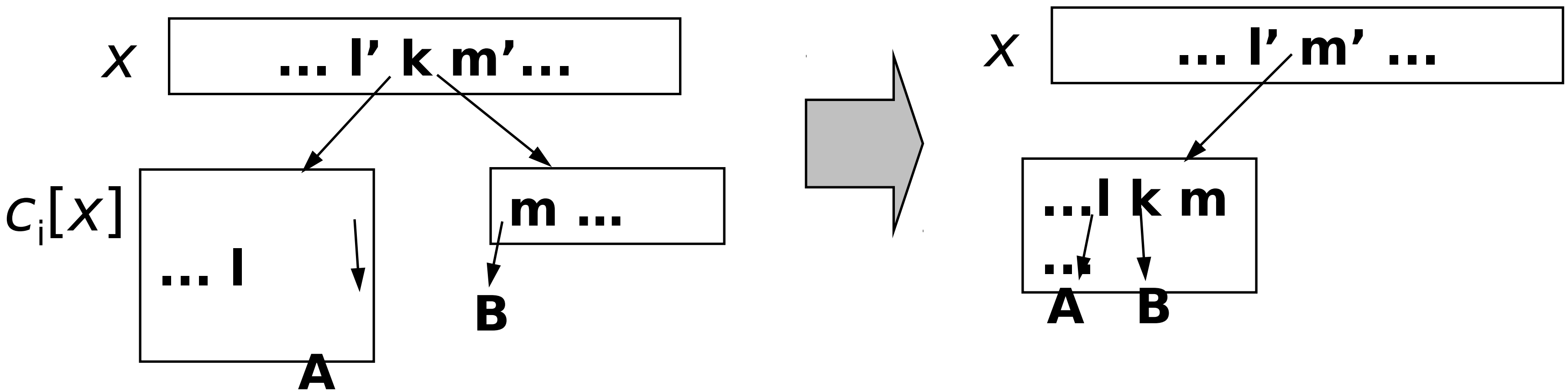
Deleting Keys

C is moved to fill B's position and E is moved to fill C's **position.**



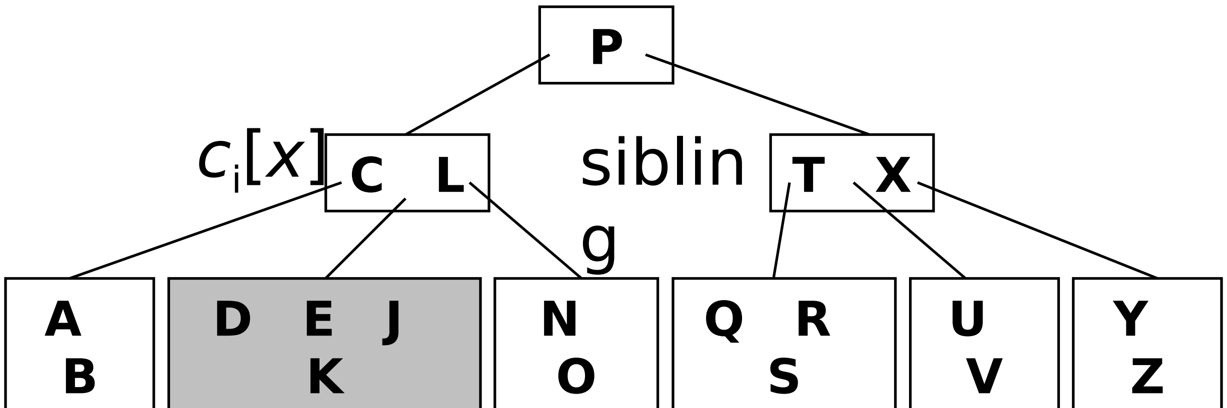
Deleting Keys- Merging

If $c_i[x]$ and both of $c_i[x]$'s siblings have $t - 1$ keys, *merge* c_i with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node



Deleting Keys

Delete D

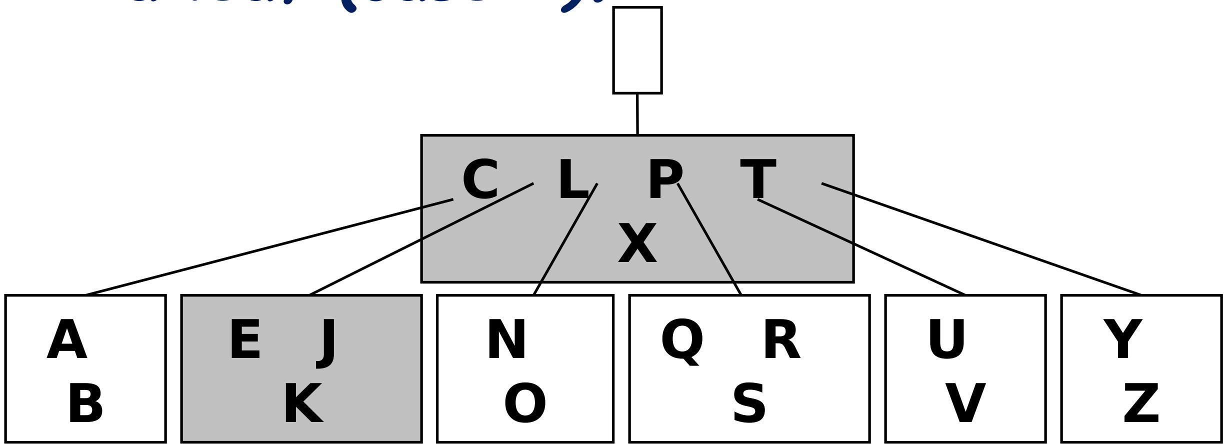


This is case 3b: If *$c_i[x]$ and both of $c_i[x]$'s immediate siblings have t – 1 keys, merge $c_i[x]$* with one sibling, which involves moving a key from *x down into the new* merged node to become the median key for that node.

Deleting Keys

Delete D

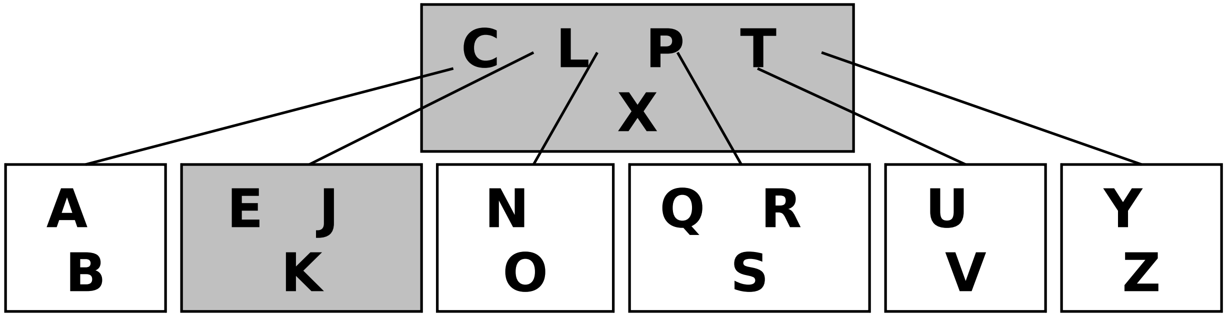
the recursion can't descend to node *CL* because it has only 2 keys, so *P* is *pushed down and merged with CL and T X to form CLPTX;* then, *D is deleted from a leaf (case 1).*



Deleting Keys

Delete D

(e') After (d), the root is deleted and the tree shrinks in height by one.



Deleting Keys

Exercise:

Show the results of deleting *C, P, and V, in order, from the tree:*