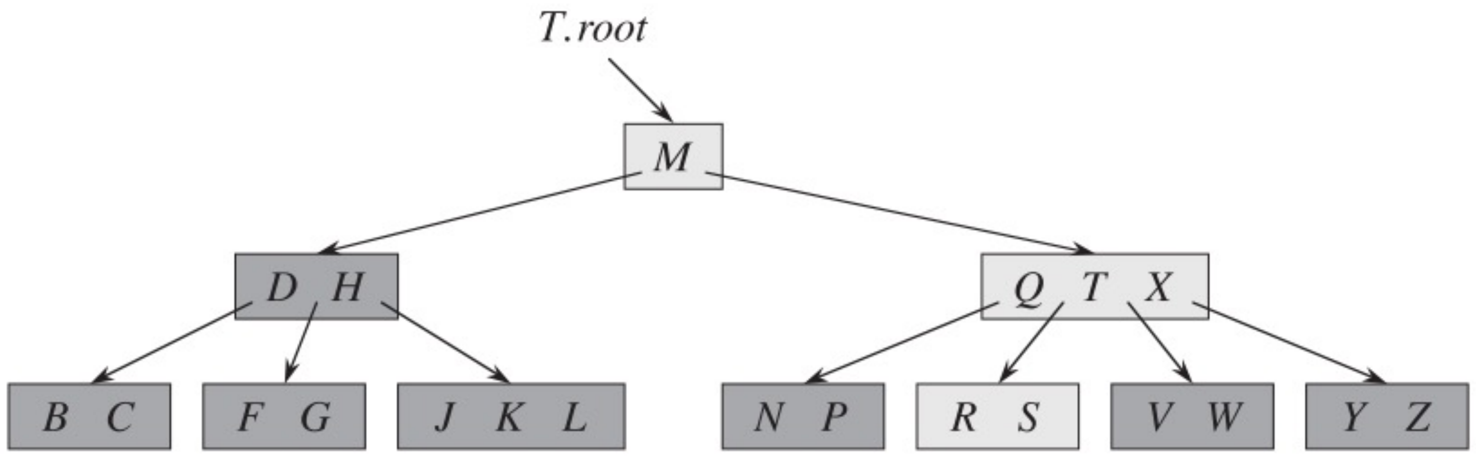
B-Trees

What is a B-Tree?

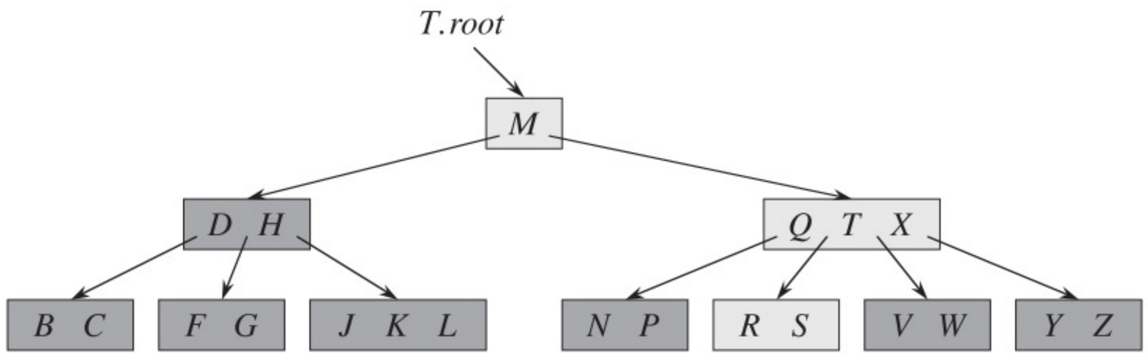
- Balanced search tree
- Each B-tree node can have many children
  - Branching factor: a few to thousands
- Height: O(lg n)



Properties of B-trees

- Keys are stored in non-decreasing order in a node
- All leaves have the same depth – tree height h
- For each non-leaf nodes with i elements contains i+1 pointers
- **t: minimum degree**, integer, t>=2
  - Every non-root node must have >= t-1 keys
    - Every non-root internal node have >= t children
  - Every node contains <= 2t-1 keys
    - Every internal nodes contain <= 2t children

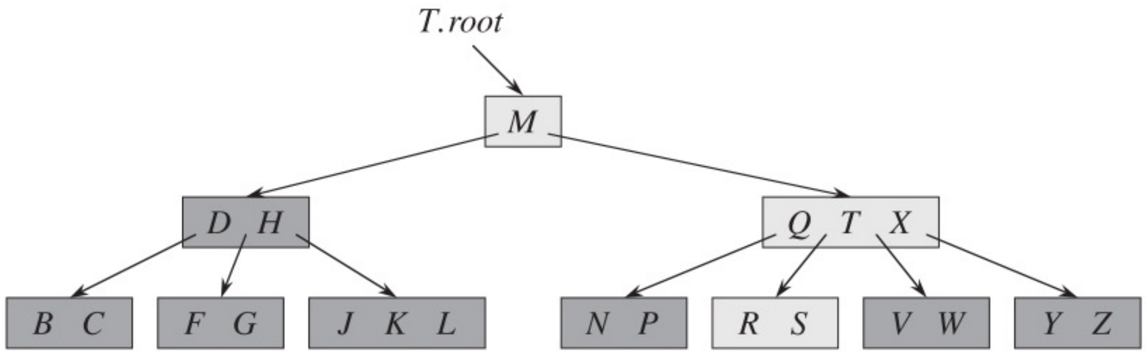
t = 2



Properties of B-trees

- **t: minimum degree**, integer, t>=2
  - Every non-root node must have >= t-1 keys
    - Every non-root internal node have >= t children
  - Every node contains <= 2t-1 keys
    - Every internal nodes contain <= 2t children
- Can be summarized in 3 main rules:
  - Every none root node must have $\lceil t/2 \rceil$ (Ceiling of t/2) children
  - Root must have min 2 children
  - All leaf nodes must be at the same level.

t = 2



B-tree search

Search tree rooted at node x for key k

```
B-TREE-SEARCH(x, k)
  i = 1
  while i ≤ x.n and k > x.key_i
      i = i + 1
  if i ≤ x.n and k == x.key_i
      return (x, i)
  elseif x.leaf
      return NIL
  else DISK-READ(x.c_i)
      return B-TREE-SEARCH(x.c_i, k)
```

pointer

find smallest index i in the tree where key[i] >= k

if k == key[i], found the key and return the results

if k != key[i] and x is a leaf, could not find k

if k != key[i] and x is not a leaf, recurse and search the subtree of x

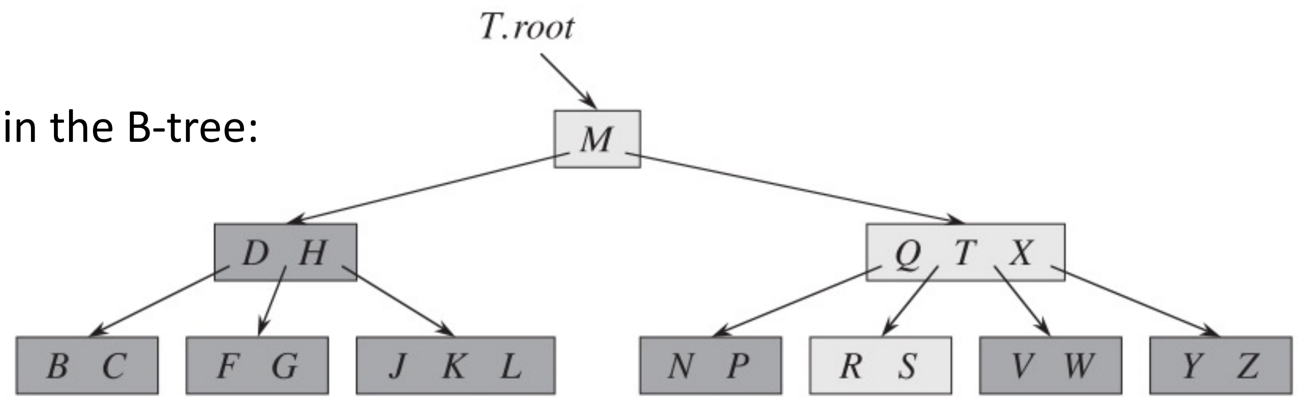
height

complexity: $O(t\ h) = O(t \log_t n)$

minimum degree

num of keys

Search for R in the B-tree:



Create an empty B-tree

```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```

Insert a key into B-trees – Splitting a node

Before inserting, if the node is full with 2t-1 keys, need to split the node first

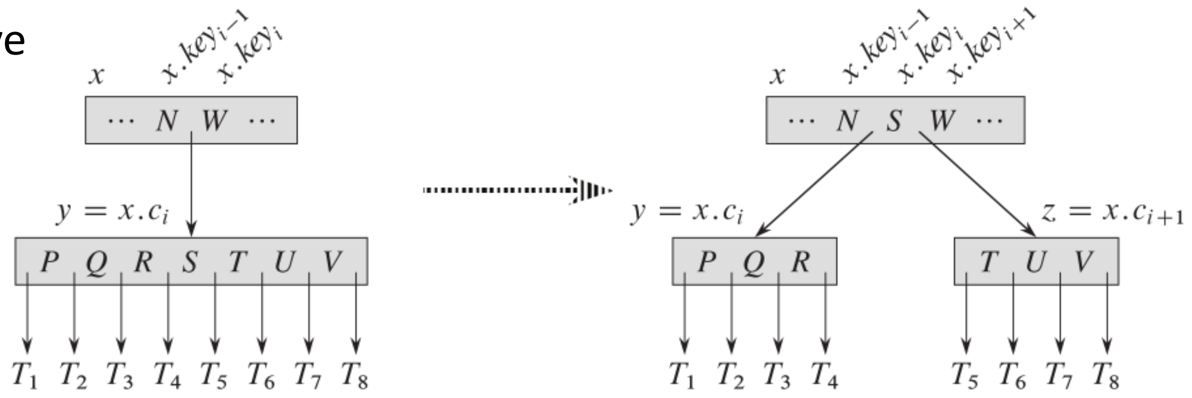
B-TREE-SPLIT-CHILD(x, i) split the full node x.ci, which is y, where x is is a nonfull internal node

```
z = ALLOCATE-NODE()
y = x.c_i
z.leaf = y.leaf
z.n = t - 1
for j = 1 to t - 1
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t
        z.c_j = y.c_{j+t}
y.n = t - 1
for j = x.n + 1 downto i + 1
    x.c_{j+1} = x.c_j
x.c_{i+1} = z
for j = x.n downto i
    x.key_{j+1} = x.key_j
x.key_i = y.key_t
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

y has 2t-1 elements.
This creates node z and allocate the largest t-1 elements in y into z, and adjust key count for y

complexity: Theta(t)

Assign z as a child of x, move the median key from y to x, and adjust key count of x



Insert a key into B-trees – nonfull node

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n
if x.leaf
    while i ≥ 1 and k < x.key_i
        x.key_{i+1} = x.key_i
        i = i − 1
    x.key_{i+1} = k
    x.n = x.n + 1
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key_i
        i = i − 1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t − 1
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

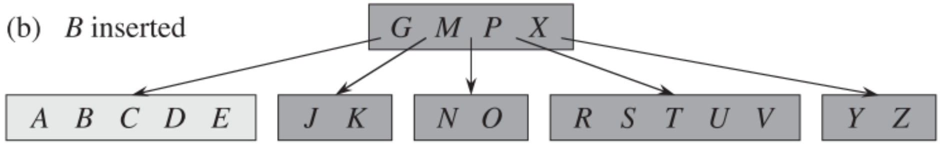
insert a key k into a non-full node x

if x is a leaf node, then insert k into x

if x is not a leaf node, and if x.ci is full, then split x.ci, and find which child node to insert k into

recursively insert k into that nonfull child node

(b) *B* inserted



Insert a key into B-trees – general case

```
B-TREE-INSERT(T, k)
  r = T.root
  if r.n == 2t − 1
      s = ALLOCATE-NODE()
      T.root = s
      s.leaf = FALSE
      s.n = 0
      s.c_1 = r
      B-TREE-SPLIT-CHILD(s, 1)
      B-TREE-INSERT-NONFULL(s, k)
  else B-TREE-INSERT-NONFULL(r, k)
```

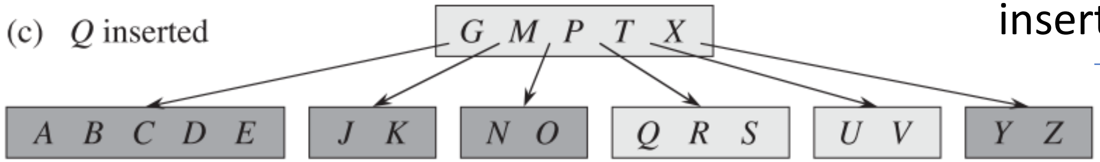
insert new key k into B-tree T

if the root node r is full, split the root

insert k into the new nonfull root by calling B-tree-Insert-Nonfull

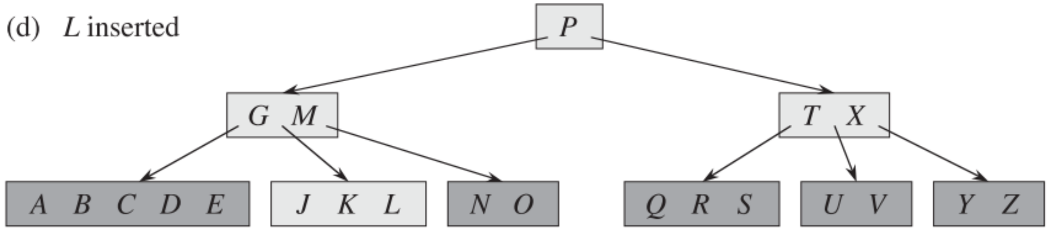
if root is nonfull, insert k into the root by calling B-tree-Insert-Nonfull

(c) *Q* inserted



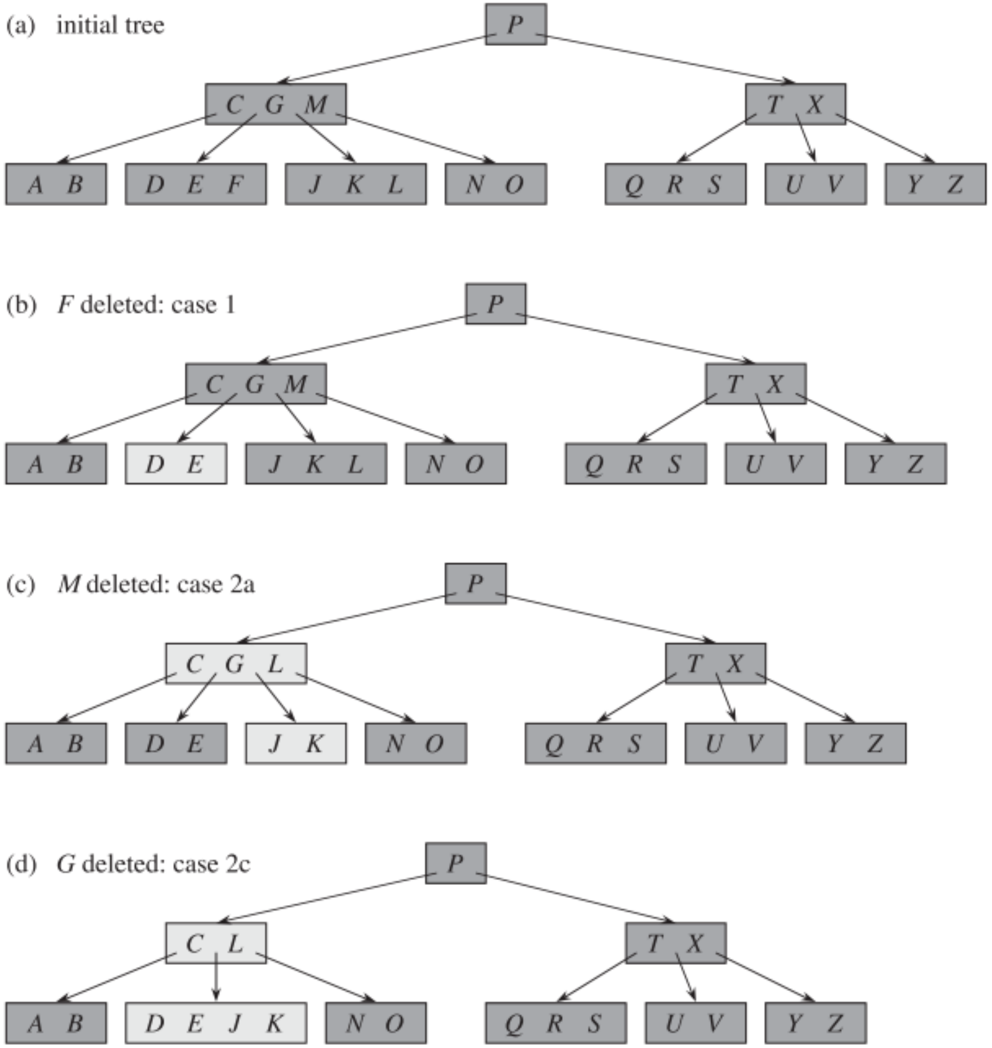
insert L into the B-tree

(d) *L* inserted



Delete from B-Tree

t = 3



initial case

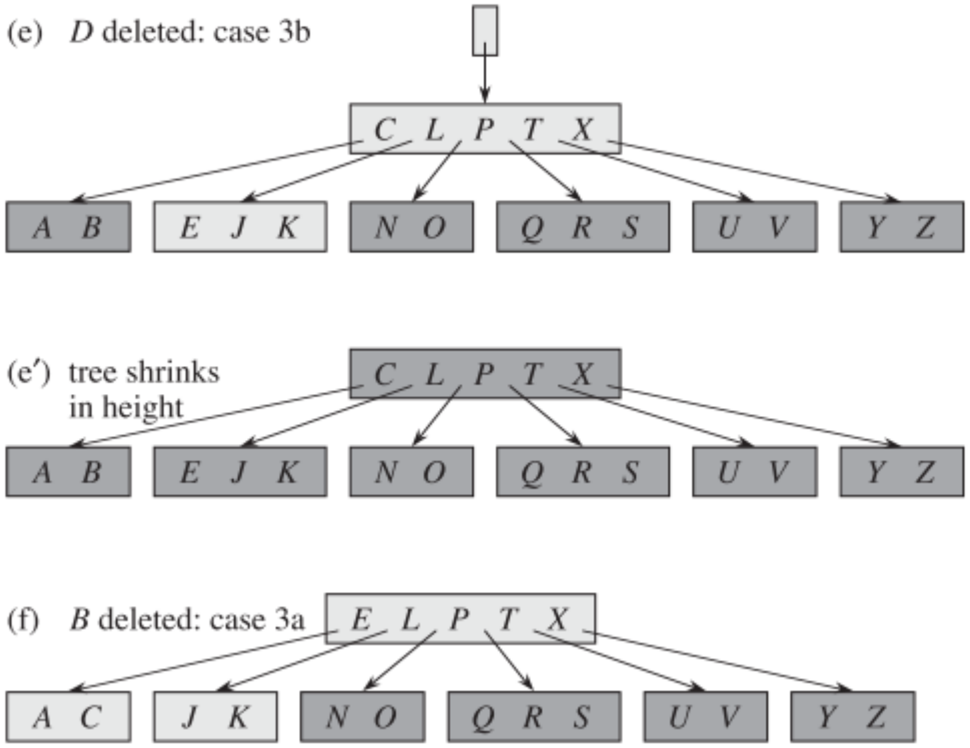
Case 1: simple deletion from leaf node

Case 2a: If child y preceding k has at least t keys, float predecessor k' to replace k, do it recursively

Case 2b: if y has less than t keys, find z that follows k, if z has at least t keys, float the successor of k' of k to replace k, do it recursively

Case 2c: otherwise, merge k and all of z into y and recursively delete k from y

Delete from B-tree continued



Case 3a: if x.ci has only t-1 keys but immediate sibling has at least t keys, move a key from x into ci and move a key from x.ci sibling up into x, and then delete k, recurse on child of x

Case 3b: if x.ci and x.ci's immediate siblings have t-1 keys, merge x.ci with one sibling and move a key from x down to the median of the new node, do it recursively

Reading

- CLRS Book.

Introduction to Algorithms, 3rd Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

B-Tree Visualization

https://www.cs.usfca.edu/~galles/visualization/BTree.html