B-TREES

Alejandro Martín
Juan Mendivelso

Contents

1. INTRODUCTION

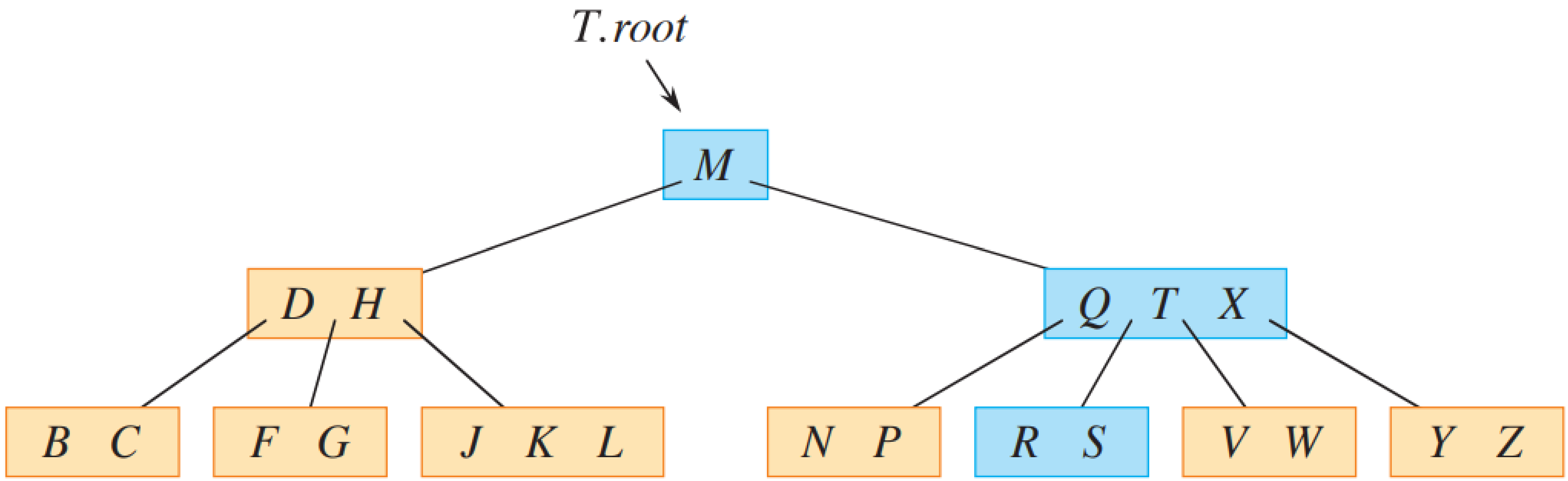
B-Trees

- B-Trees are balanced search trees designed to work well on disk drives or other direct-access secondary storage devices.
- Similar to Red-Black trees, but they are better at minimizing the number of operations that access disks.
- B-Tree nodes may have many children from few to thousands, contrary to Red-Black trees.
- Every B-Tree has height $O(\lg n)$. But a B-Tree has a larger branching factor than a Red-Black tree.

B-Trees

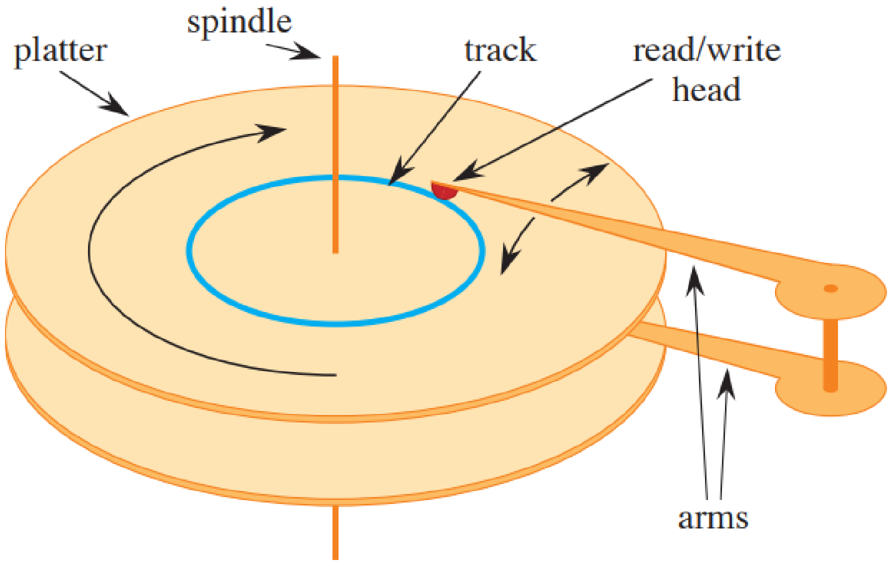
- B-Trees generalize binary search trees in a natural manner.
- If an internal B-tree node x contains $x.n$ keys, then x has $x.n + 1$ children.
- The keys in node x serve as dividing points separating the range of keys handled by x intro $x.n + 1$ subranges, each handled by one child of x.
- A search for a key in a B-tree makes an $(x.n + 1)$-way decision based on comparisons with the $x.n$ keys stored at node x.

B-Trees



Data structures on secondary storage

- Computer systems take advantage of various technologies that provide memory capacity:
  - Main memory
  - Secondary storage (Magnetic disk drives, SSD)



Data structures on secondary storage

- In order to amortize the time spent waiting for mechanical movements (latency), disk drives access more than one item at a time.

- Information is divided into a number of equal-sized blocks of bits that appear consecutively within tracks, and each disk read or write is one or more entire blocks.

Data structures on secondary storage

- Often, accessing a block of information and reading it from a disk drive takes longer than processing all the information read.

- We will look separately at the two principal components of the running time:
  - The number of disk accesses
  - The CPU (computing) time.

Data structures on secondary storage

- We measure the number of disk accesses in terms of the number of blocks of information that need to be read from or written to the disk drive.
- In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once.
- The B-tree algorithms copy selected blocks form disk into main memory as needed and write back onto disk the blocks that have changed.
- B-Tree algorithms keep only a constant number of blocks in main memory at any time, and thus the size of main memory does not limit the size of B-trees that can be handled.

Data structures on secondary storage

- B-tree procedures need to be able to read information from disk into main memory and write information from main memory to disk.
- Consider some object x. If x is currently in the computer's main memory, then the code can refer to the attributes of x as usual.
- If x resides on disk, however, then the procedure must perform the operation *DISK-READ(x)* to read the block containing x into main memory before it can refer to x's attributes.
- Similarly, procedures call *DISK-WRITE(x)* to save any changes that have been made to the attributes of object x by writing to disk the block containing x.

Data structures on secondary storage

x = a pointer to some object
DISK-READ(x)
operations that access and/or modify the attributes of x
DISK-WRITE(x) // omitted if no attributes of x were changed
other operations that access but do not modify attributes of x

2. DEFINITION OF B-TREES

Definition of B-Trees

- A ***B-tree*** T is a rooted tree with root *T.root* having the following properties:
  1. Every node x has the following attributes:
     a) $x.n$, the number of keys currently stored in node x,
     b) The $x.n$ keys themselves, stored in monotonically increased order.
     c) $x.leaf$, a Boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
  2. Each internal node x also contains $x.n+1$ pointers $x.c_1, x.c_2, \dots, x.c_{x.n+1}$ to its children. Leaf nodes have their c_i attributes undefined.
  3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1}$$

Definition of B-Trees

- A ***B-tree*** T is a rooted tree with root *T.root* having the following properties:
    4. All leaves have the same depth, which is the tree's height h.
    5. Nodes have lower and upper bounds on the number of keys they can contain, 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 nonempty the root must have at least one key.
        b) Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.

- The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3 or 4 children, and it is a ***2-3-4 tree*.**

The height of a B-Tree

- The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. The following theorem bounds the worst-case height of a B-Tree.
- **Theorem:** if $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq log_t \frac{n+1}{2}$$

- **Proof:**
  - The root of a nonempty B-tree T contains at least one key, and all other nodes contain at least $t-1$ keys.
  - Then T contains at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3 and so on, until at depth h it has at least $2t^{h-1}$ nodes.

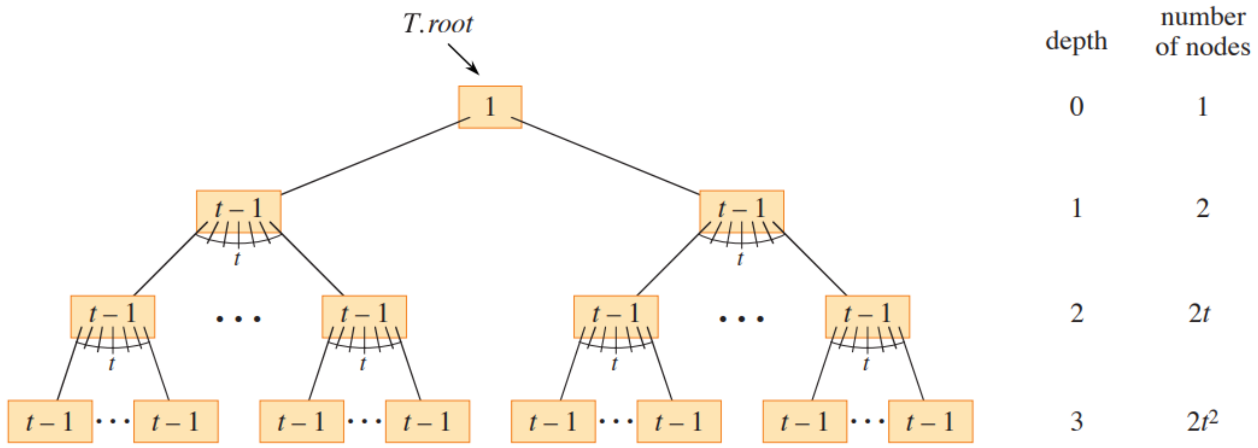
The height of a B-Tree

- **Proof:**
  - The number n of keys therefore satisfies the inequality:

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right)$$

$$= 2t^h - 1$$



  - So that $t^h \leq (n+1)/2$. Then we take base-t logarithm of both sides and we finish

3. BASIC OPERATION ON B-TREES

Basic operations on B-Trees

- We present the operations B-TREE-SEARCH, B-TREE-CREATE and B-TREE-INSERT. These procedures observe two conventions:
    - The root of the B-Tree is always in main memory, so that no procedure ever needs to perform a DISK-READ on the root. However, if any changes occur in the root node, then DISK-WRITE must be called there.
    - Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

Searching a B-Tree

- Similar like searching a BST, except that at each internal node x, the search makes an $(x.n + 1)$-way branching decision (instead of the two-way).
- The procedure B-TREE-SEARCH generalizes the procedure defined for BST.

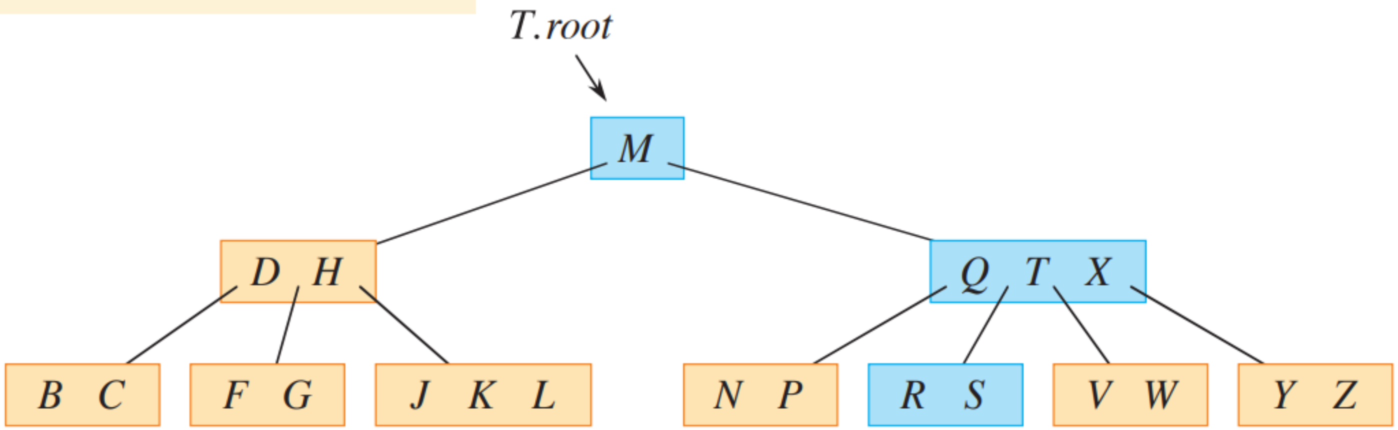
```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

$O(h)$ disk accesses and $O(th)$ CPU time

Searching a B-Tree

```
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)
```

B-TREE-SEARCH(R,T)



Creating an empty B-Tree

- We use the B-TREE-CREATE procedure to create an empty root node and then call the B-TREE-INSERT procedure to add new keys
- Both procedures use an auxiliary procedure (ALLOCATE-NODE) that allocates one disk block to be used as a new one in $O(1)$ time.

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

It takes $O(1)$ time

Inserting a key into a B-Tree

- With a B-Tree, you cannot simply create a new leaf node and insert it. Instead, we insert the new key into an existing leaf node.
- Since we cannot insert a key into a leaf node that is full, we need an operation that splits a full node y around its ***median key*** into two nodes having only $t - 1$ 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, then we must split it before we can insert the new key.
- To avoid having to go back up the tree, we just split every full node we encounter as we go down the tree

Splitting a node in a B-Tree

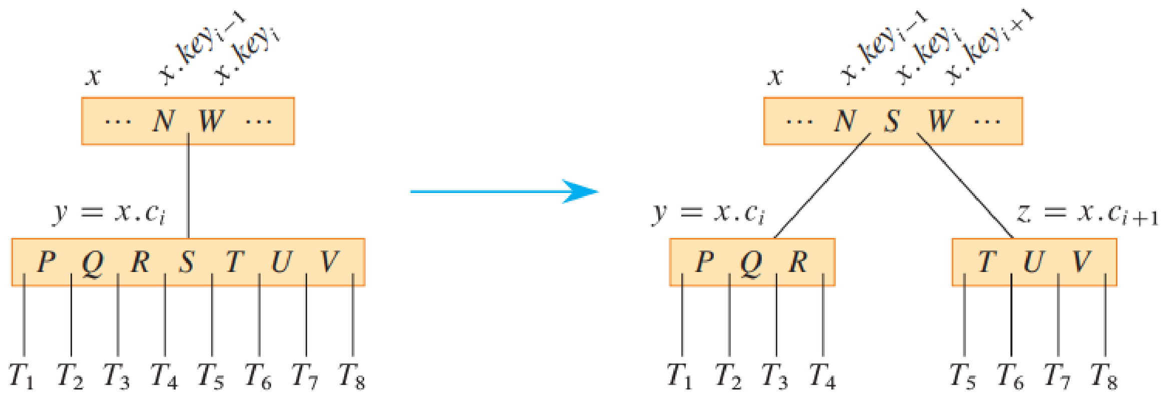
- The procedure B-TREE-SPLIT-CHILD takes as input a nonfull internal node x and an index i such that $x.c_i$ is a full child of x

```
B-TREE-SPLIT-CHILD(x, i)
  y = x.c_i                          // full node to split
  z = ALLOCATE-NODE()                // z will take half of y
  z.leaf = y.leaf
  z.n = t − 1
  for j = 1 to t − 1                 // z gets y's greatest keys ...
      z.key_j = y.key_{j+t}
  if not y.leaf
      for j = 1 to t                 // ... and its corresponding children
          z.c_j = y.c_{j+t}
  y.n = t − 1                        // y keeps t − 1 keys
  for j = x.n + 1 downto i + 1       // shift x's children to the right ...
      x.c_{j+1} = x.c_j
  x.c_{i+1} = z                      // ... to make room for z as a child
  for j = x.n downto i               // shift the corresponding keys in x
      x.key_{j+1} = x.key_j
  x.key_i = y.key_t                  // insert y's median key
  x.n = x.n + 1                      // x has gained a child
  DISK-WRITE(y)
  DISK-WRITE(z)
  DISK-WRITE(x)
```

It takes $\Theta(t)$ CPU time and $O(1)$ disk operations

Splitting a node in a B-Tree

```
B-TREE-SPLIT-CHILD(x, i)
y = x.c_i                          // full node to split
z = ALLOCATE-NODE()                // z will take half of y
z.leaf = y.leaf
z.n = t − 1
for j = 1 to t − 1                 // z gets y's greatest keys ...
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t                 // ... and its corresponding children
        z.c_j = y.c_{j+t}
y.n = t − 1                        // y keeps t − 1 keys
for j = x.n + 1 downto i + 1       // shift x's children to the right ...
    x.c_{j+1} = x.c_j
x.c_{i+1} = z                      // ... to make room for z as a child
for j = x.n downto i               // shift the corresponding keys in x
    x.key_{j+1} = x.key_j
x.key_i = y.key_t                  // insert y's median key
x.n = x.n + 1                      // x has gained a child
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```



Splitting a node with $t = 4$

Inserting a key into a B-Tree in a single pass down the tree

- We use procedure B-TREE-INSERT to insert a key k into a B-Tree T of height h

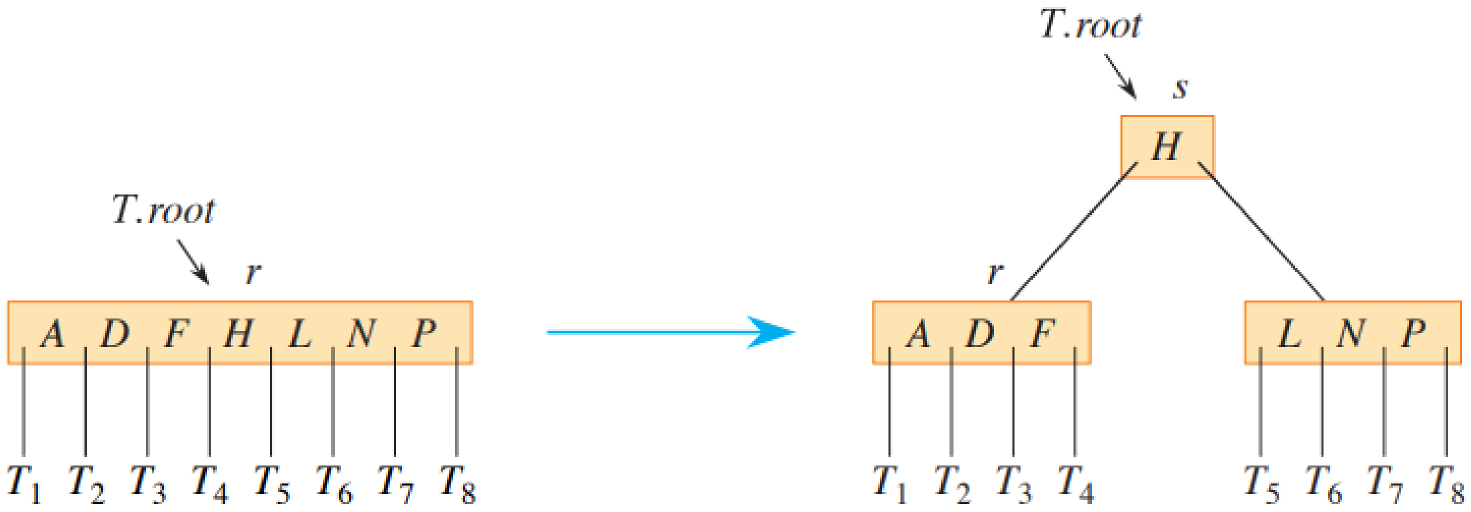
```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t - 1
    s = B-TREE-SPLIT-ROOT(T)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

It takes $\Theta(th)$ CPU time and $O(h)$ disk accesses

Inserting a key into a B-Tree in a single pass down the tree

- If the root is full, we use the procedure B-TREE-SPLIT-ROOT to split it (this is the only way to increase the height of the B-Tree).

```
B-TREE-SPLIT-ROOT(T)
  s = ALLOCATE-NODE()
  s.leaf = FALSE
  s.n = 0
  s.c_1 = T.root
  T.root = s
  B-TREE-SPLIT-CHILD(s, 1)
  return s
```



Splitting the root with $t = 4$

Inserting a key into a B-Tree in a single pass down the tree

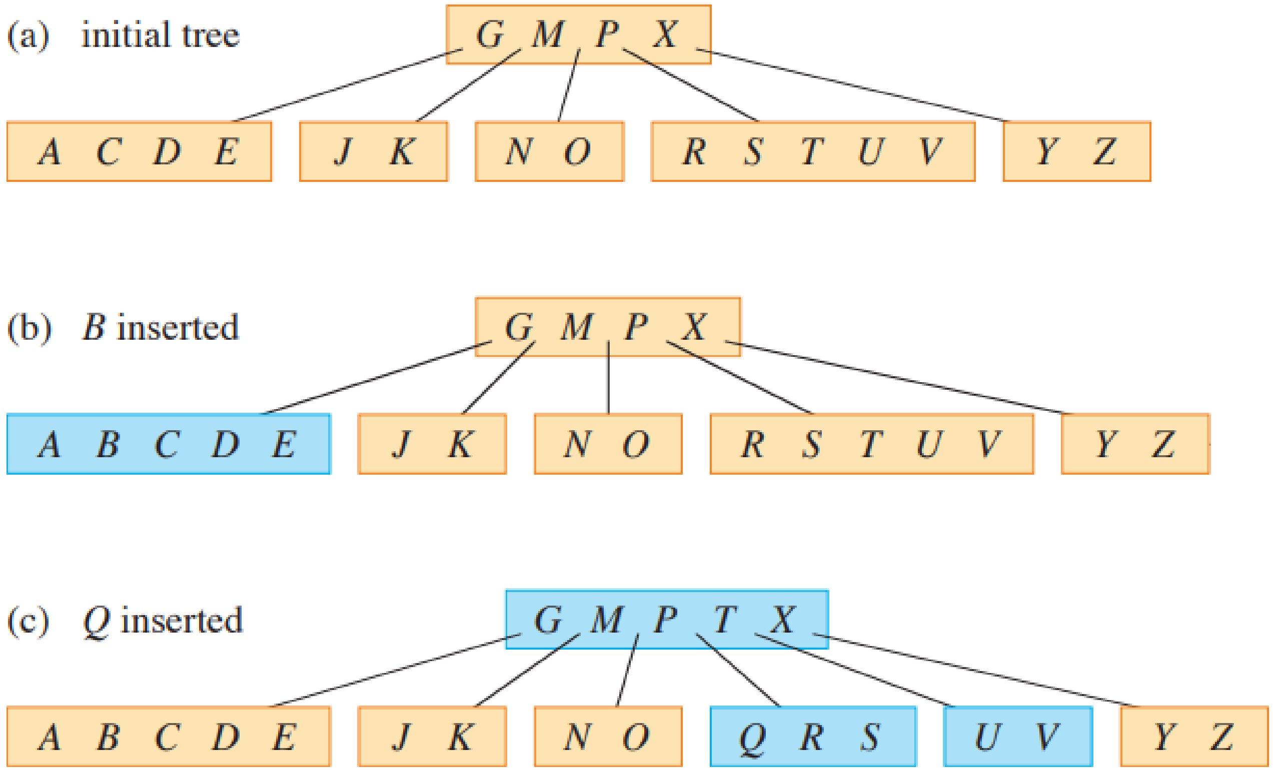
- The auxiliary procedure B-TREE-INSERT-NONFULL inserts key k into node x, which is assumed to be nonfull when the procedure is called.
- It recurses as necessary down the tree, at all time guaranteeing that the node to which it recurses is not full by calling B-TREE-SPLIT-CHILD if necessary.

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n
if x.leaf                              // inserting into a leaf?
    while i ≥ 1 and k < x.key_i        // shift keys in x to make room for k
        x.key_{i+1} = x.key_i
        i = i − 1
    x.key_{i+1} = k                    // insert key k in x
    x.n = x.n + 1                      // now x has 1 more key
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key_i       // find the child where k belongs
        i = i − 1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t − 1               // split the child if it's full
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i                 // does k go into x.c_i or x.c_{i+1}?
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

Inserting a key into a B-Tree in a single pass down the tree

```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t − 1
    s = B-TREE-SPLIT-ROOT(T)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-INSERT-NONFULL(x, k)
i = x.n
if x.leaf                                  // inserting into a leaf?
    while i ≥ 1 and k < x.key_i            // shift keys in x to make room for k
        x.key_{i+1} = x.key_i
        i = i − 1
    x.key_{i+1} = k                        // insert key k in x
    x.n = x.n + 1                          // now x has 1 more key
    DISK-WRITE(x)
else while i ≥ 1 and k < x.key_i           // find the child where k belongs
        i = i − 1
    i = i + 1
    DISK-READ(x.c_i)
    if x.c_i.n == 2t − 1                   // split the child if it's full
        B-TREE-SPLIT-CHILD(x, i)
        if k > x.key_i                     // does k go into x.c_i or x.c_{i+1}?
            i = i + 1
    B-TREE-INSERT-NONFULL(x.c_i, k)
```

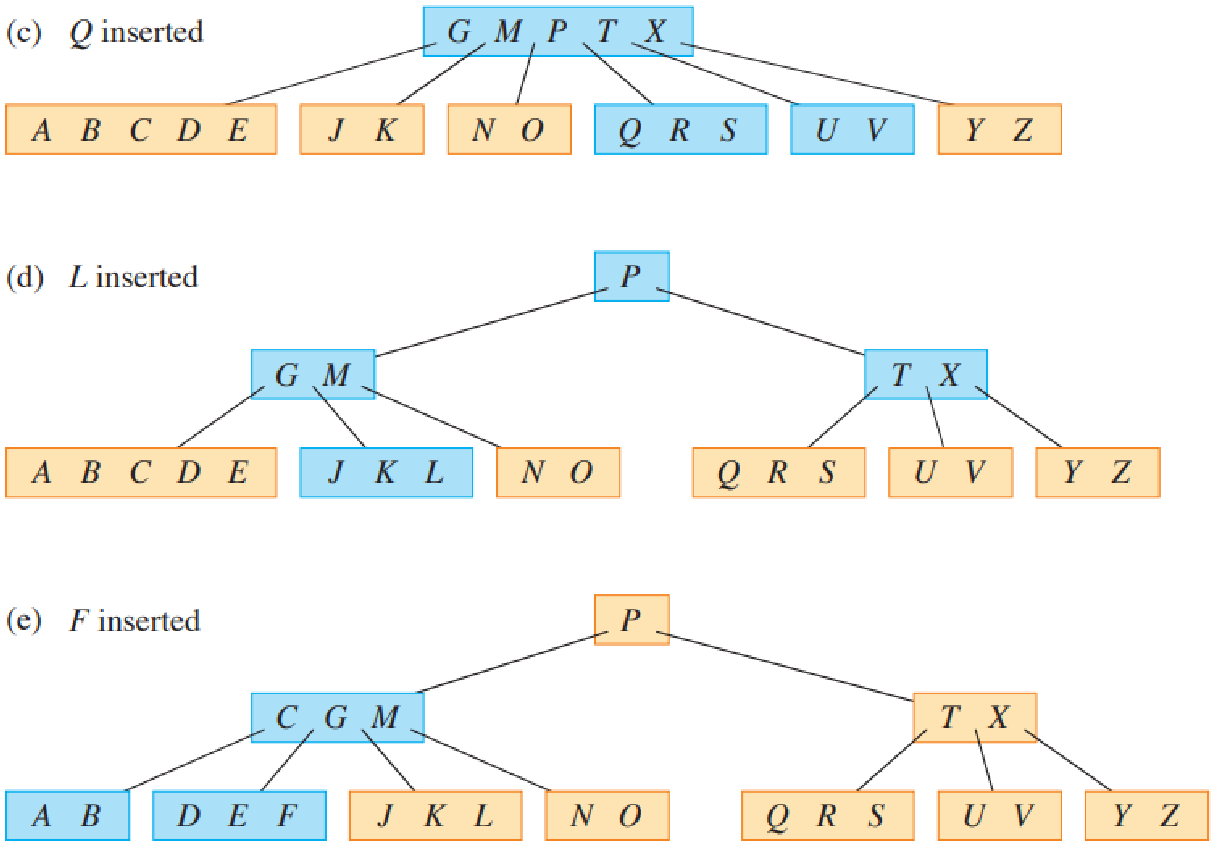


The minimum degree for this B-Tree is $t = 3$

Inserting a key into a B-Tree in a single pass down the tree

```
B-TREE-INSERT(T, k)
  r = T.root
  if r.n == 2t − 1
      s = B-TREE-SPLIT-ROOT(T)
      B-TREE-INSERT-NONFULL(s, k)
  else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-INSERT-NONFULL(x, k)
  i = x.n
  if x.leaf                              // inserting into a leaf?
      while i ≥ 1 and k < x.key_i        // shift keys in x to make room for k
          x.key_{i+1} = x.key_i
          i = i − 1
      x.key_{i+1} = k                    // insert key k in x
      x.n = x.n + 1                      // now x has 1 more key
      DISK-WRITE(x)
  else while i ≥ 1 and k < x.key_i       // find the child where k belongs
          i = i − 1
      i = i + 1
      DISK-READ(x.c_i)
      if x.c_i.n == 2t − 1               // split the child if it's full
          B-TREE-SPLIT-CHILD(x, i)
          if k > x.key_i                 // does k go into x.c_i or x.c_{i+1}?
              i = i + 1
      B-TREE-INSERT-NONFULL(x.c_i, k)
```



The minimum degree for this B-Tree is $t = 3$

Presentation made by Martín & Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Fourth Edition. Cormen, Leiserson, Rivests & Stein. The MIT Press. 2022.

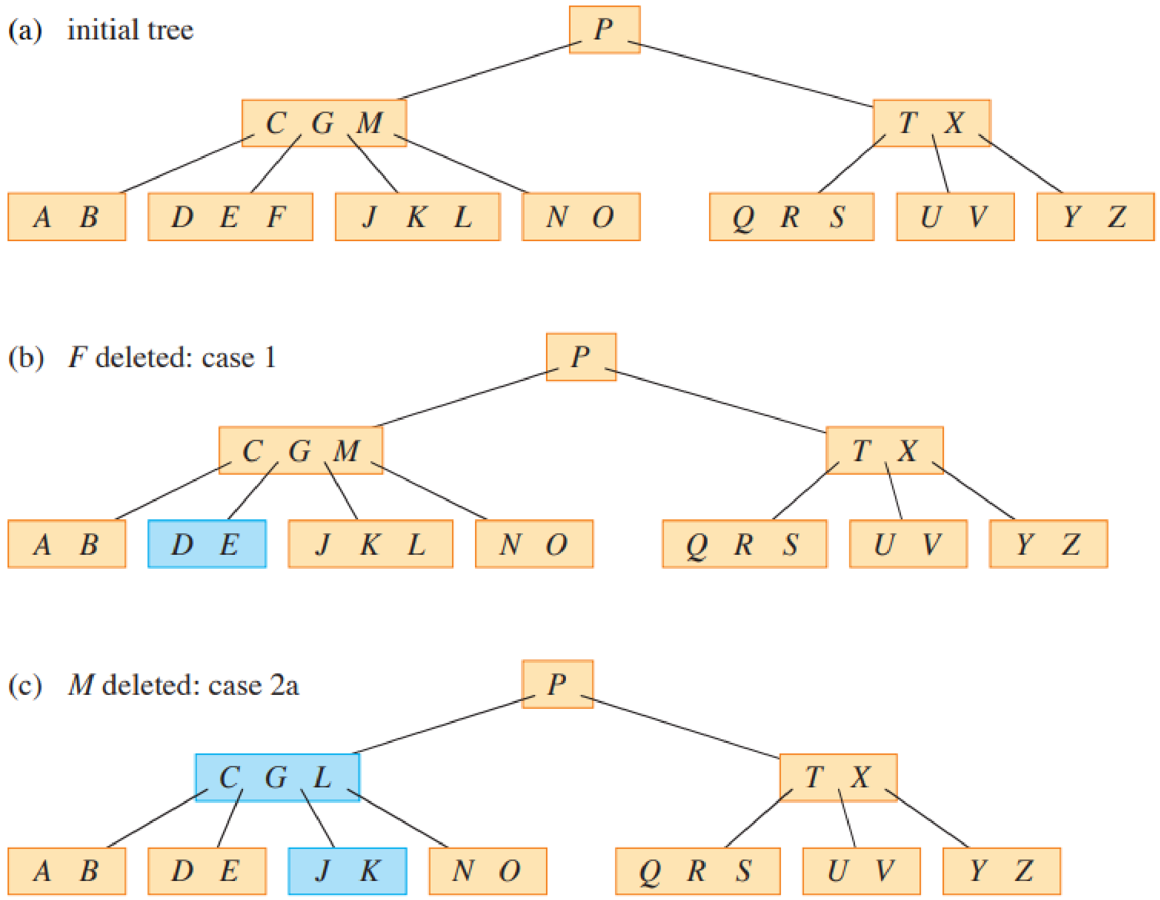
4. DELETING A KEY FROM A B-TREE

Deleting a key from a B-Tree

- It is analogous to insertion but more complicated. Because we can delete a key from any node, so that we must rearrange the node's children if needed.
- Just as a node should not get too big due to insertion, a node must not get too small during deletion.
- The procedure B-TREE-DELETE deletes the key k from the subtree rooted at x.
- B-TREE-DELETE prevents any node from becoming underfull (i.e having fewer than $t - 1$ keys) while also making a single pass down the tree, searching for and deleting the key.

Deleting a key from a B-Tree

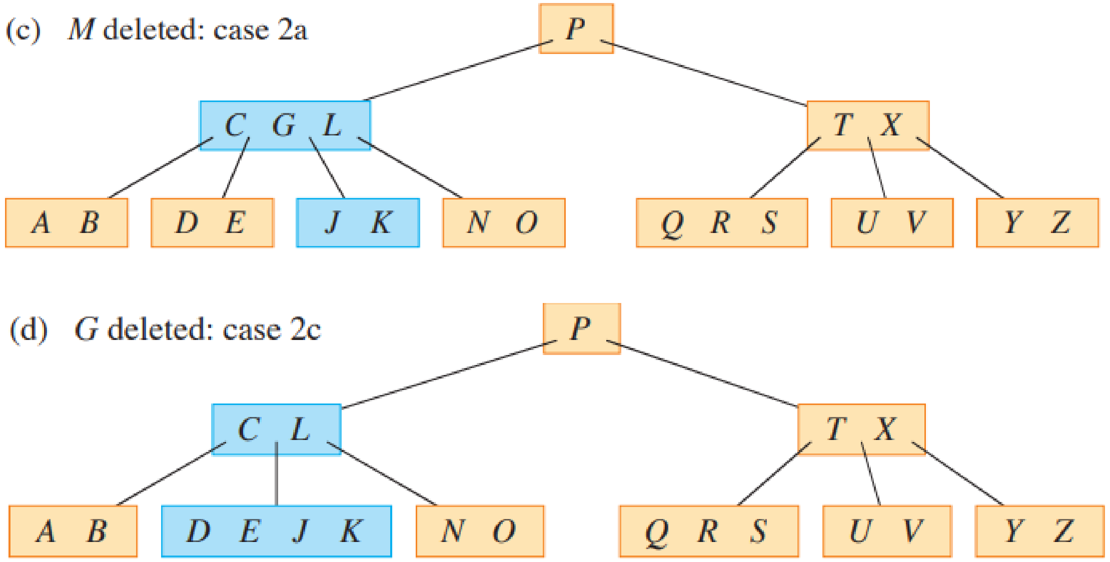
- We explain how the procedure works:
  - **Case 1**: The search arrives at a leaf node. If x contains key k, then delete k from x. If x does not contain key k, then k was not in the B-Tree, then do nothing.
  - **Case 2**: The search arrives at an internal node x that contains key k. Let $k = x.key_i$, $x.c_i$ the child that precedes k and $x.c_{i+1}$ the child that follows k. Then 3 subcases arise:
    - **Case 2a**: $x.c_i$ has at least t keys. Find the predecessor k' of k in the subtree rooted at $x.c_i$. Recursively delete k' from $x.c_i$, and replace k by k' in x.



The minimum degree for this B-Tree is $t = 3$

Deleting a key from a B-Tree

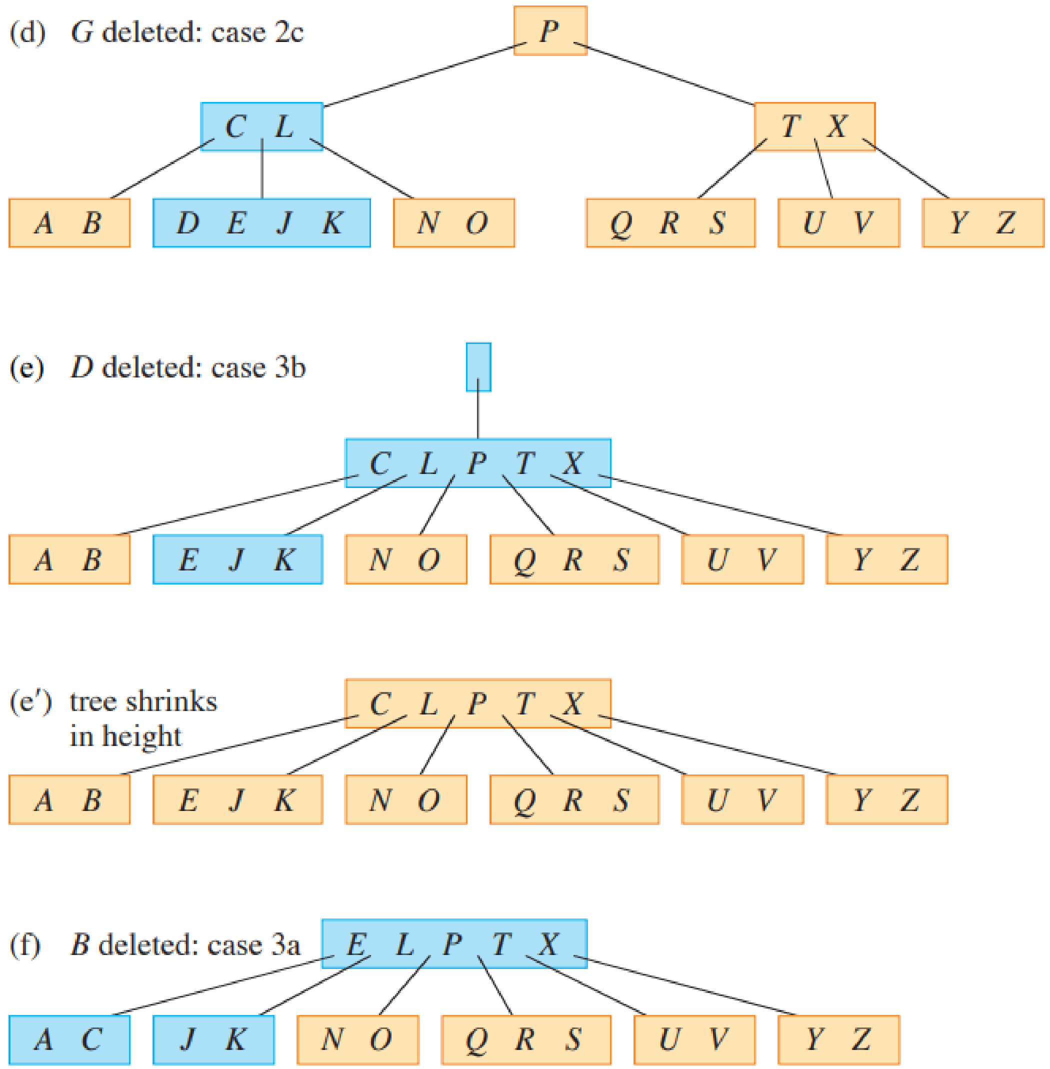
- We explain how the procedure works:
  - **Case 2**: x is internal node and contains k. Let $k = x.key_i$, $x.c_i$ child that precedes k and $x.c_{i+1}$ child that follows k:
    - **Case 2b**: $x.c_i$ has t-1 keys and $x.c_{i+1}$ has at least t keys. Find the successor k' of k in the subtree rooted at $x.c_{i+1}$. Recursively delete k' from $x.c_{i+1}$ and replace k by k' in x.
    - **Case 2c**: Both $x.c_i$ and $x.c_{i+1}$ have $t-1$ keys. Merge k and all $x.c_{i+1}$ into $x.c_i$, so that x loses both k and the pointer to $x.c_{i+1}$, and $x.c_i$ now contains $2t-1$ keys. Then free $x.c_{i+1}$ and recursively delete k from $x.c_i$



The minimum degree for this B-Tree is $t = 3$

Deleting a key from a B-Tree

- We explain how the procedure works:
  - **Case 3:** The search arrives at an internal node x that does not contain key k. Ensure that each node visited has at least t keys. To do so, determine the root $x.c_i$ of the appropriate subtree that must contain k. If $x.c_i$ only has $t-1$ keys 2 cases arise:
    - **Case 3a:** $x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys. Give $x.c_i$ an extra key by moving a key from x into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.
    - **Case 3b**: $x.c_i$ and each of $x.c_i$'s immediate siblings have $t-1$ keys. Merge $x.c_i$ with one sibling which involves moving a key from x down into the new merged node to become the median key for that node.



The minimum degree for this B-Tree is $t=3$

Deleting a key from a B-Tree

- In cases 2c and 3b, if node x is the root, it could end up having no keys. When this occurs, x is deleted and x's only child $x.c_i$ becomes the new root of the tree.

- This procedure involves only $O(h)$ disk operations for a B-Tree of height h. The CPU time required is $O(th)$.