B-Trees

Version of October 2, 2014

Motivation

- An AVL tree can be an excellent data structure for implementing dictionary search, insertion and deletion
  - Each operation on an n-node AVL tree takes $O(\log n)$ time

Motivation

- An AVL tree can be an excellent data structure for implementing dictionary search, insertion and deletion
  - Each operation on an n-node AVL tree takes $O(\log n)$ time
- This only works, though, long as the entire data structure fits into main memory

Motivation

- An AVL tree can be an excellent data structure for implementing dictionary search, insertion and deletion
  - Each operation on an n-node AVL tree takes $O(\log n)$ time
- This only works, though, long as the entire data structure fits into main memory
- When the data size is too large and data must reside on disk, AVL performance may deteriorate rapidly.

A Practical Example

- For a typical machine
  - Main memory: 100 nanoseconds per access
    (a nanosecond is 10^{-9} second)

A Practical Example

- For a typical machine
  - Main memory: 100 nanoseconds per access
    (a nanosecond is 10^{-9} second)
  - Hard disk: 0.01 seconds per access
    (seek time $+$ rotational latency)

A Practical Example

- For a typical machine
  - Main memory: 100 nanoseconds per access
    (a nanosecond is 10^{-9} second)
  - Hard disk: 0.01 seconds per access
    (seek time + rotational latency)
  - HD is 5 orders of magnitude slower than main memory

A Practical Example

- For a typical machine
  - Main memory: 100 nanoseconds per access
    (a nanosecond is 10^{-9} second)
  - Hard disk: 0.01 seconds per access
    (seek time + rotational latency)
  - HD is 5 orders of magnitude slower than main memory
  - HD access reads a large block of data at one time
    Reading one byte and full block of data take $\sim$ the same time.

A Practical Example

- For a typical machine
  - Main memory: 100 nanoseconds per access
    (a nanosecond is 10^{-9} second)
  - Hard disk: 0.01 seconds per access
    (seek time + rotational latency)
  - HD is 5 orders of magnitude slower than main memory
  - HD access reads a large block of data at one time
    Reading one byte and full block of data take $\sim$ the same time.

- Consider a database with 10^9 items (stored on disk)
  - Tree would have height $\sim \log_2 10^9 = 30$
  - Operations on these BSTS would need 30 disk accesses
  - a *very* slow 0.3 second!

A Practical Example

- For a typical machine
  - Main memory: 100 nanoseconds per access
    (a nanosecond is 10^{-9} second)
  - Hard disk: 0.01 seconds per access
    (seek time + rotational latency)
  - HD is 5 orders of magnitude slower than main memory
  - HD access reads a large block of data at one time
    Reading one byte and full block of data take $\sim$ the same time.
- Consider a database with 10^9 items (stored on disk)
  - Tree would have height $\sim \log_2 10^9 = 30$
  - Operations on these BSTS would need 30 disk accesses
  - a *very* slow 0.3 second!
- Want a way to substantially reduce number of disk accesses.

From Binary to M-ary

- Idea: allow the nodes to have many children

From Binary to M-ary

- Idea: allow the nodes to have many children
  - More branching $\Rightarrow$ Shallower Tree $\Rightarrow$ Fewer disk accesses

From Binary to M-ary

- Idea: allow the nodes to have many children
  - More branching $\Rightarrow$ Shallower Tree $\Rightarrow$ Fewer disk accesses
- As branching increases, tree height decreases (shallower)

From Binary to M-ary

- Idea: allow the nodes to have many children
  - More branching $\Rightarrow$ Shallower Tree $\Rightarrow$ Fewer disk accesses
- As branching increases, tree height decreases (shallower)
- An m-ary tree allows m-way branching
  - Each internal node has at most $m - 1$ keys

From Binary to M-ary

- Idea: allow the nodes to have many children
  - More branching $\Rightarrow$ Shallower Tree $\Rightarrow$ Fewer disk accesses
- As branching increases, tree height decreases (shallower)
- An m-ary tree allows m-way branching
  - Each internal node has at most $m - 1$ keys
- Complete m-ary tree has height $\sim \log_m n$ instead of $\sim \log_2 n$

From Binary to *M*-ary

- Idea: allow the nodes to have many children
  - More branching $\Rightarrow$ Shallower Tree $\Rightarrow$ Fewer disk accesses
- As branching increases, tree height decreases (shallower)
- An m-ary tree allows m-way branching
  - Each internal node has at most $m - 1$ keys
- Complete m-ary tree has height $\sim \log_m n$ instead of $\sim \log_2 n$
  - Example: if $m = 100$, then $\log_{100} 10^9 < 5$

From Binary to M-ary

- Idea: allow the nodes to have many children
  - More branching $\Rightarrow$ Shallower Tree $\Rightarrow$ Fewer disk accesses
- As branching increases, tree height decreases (shallower)
- An m-ary tree allows m-way branching
  - Each internal node has at most $m - 1$ keys
- Complete m-ary tree has height $\sim \log_m n$ instead of $\sim \log_2 n$
  - Example: if $m = 100$, then $\log_{100} 10^9 < 5$
  - This reduces disk accesses and speeds up search significantly

A **B-tree** of **(minimum) degree** $t \geq 2$ has following properties:

B-Trees

A **B-tree** of **(minimum) degree** $t \geq 2$ has following properties:

1. Every node x (except root) has between t and $2t$ children
   - Node with $n[x]$ keys has $n[x] + 1$ children.
     $\Rightarrow$ between $t - 1$ and $2t - 1$ keys
   - Root has at most $2t$ children
     $\Rightarrow$ at most $2t - 1$ keys
2. All leaves appear on *the same level*

B-Trees

A **B-tree** of **(minimum) degree** $t \geq 2$ has following properties:

1. Every node x (except root) has between t and $2t$ children
   - Node with $n[x]$ keys has $n[x] + 1$ children.
     $\Rightarrow$ between $t - 1$ and $2t - 1$ keys
   - Root has at most $2t$ children
     $\Rightarrow$ at most $2t - 1$ keys
2. All leaves appear on *the same level*
3. Every node x has the following fields:
   a. $n[x]$, the number of keys currently stored in node x

A **B-tree** of **(minimum) degree** $t \geq 2$ has following properties:

1. Every node x (except root) has between t and $2t$ children
   - Node with $n[x]$ keys has $n[x] + 1$ children.
     $\Rightarrow$ between $t - 1$ and $2t - 1$ keys
   - Root has at most $2t$ children
     $\Rightarrow$ at most $2t - 1$ keys
2. All leaves appear on *the same level*
3. 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 nondecreasing order

B-Trees

A **B-tree** of **(minimum) degree** $t \geq 2$ has following properties:

1. Every node x (except root) has between t and $2t$ children
   - Node with $n[x]$ keys has $n[x] + 1$ children.
     $\Rightarrow$ between $t - 1$ and $2t - 1$ keys
   - Root has at most $2t$ children
     $\Rightarrow$ at most $2t - 1$ keys
2. All leaves appear on *the same level*
3. 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 nondecreasing order
   c. $n[x] + 1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children
      (Leaf nodes have no children, so their c_i fields are undefined)

A **B-tree** of **(minimum) degree** $t \geq 2$ has following properties:

1. Every node x (except root) has between t and $2t$ children
   - Node with $n[x]$ keys has $n[x] + 1$ children.
     $\Rightarrow$ between $t - 1$ and $2t - 1$ keys
   - Root has at most $2t$ children
     $\Rightarrow$ at most $2t - 1$ keys
2. All leaves appear on *the same level*
3. 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 nondecreasing order
   c. $n[x] + 1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children
      (Leaf nodes have no children, so their c_i fields are undefined)
4. Keys $\mathrm{key}_i[x]$ separate ranges of keys in subtrees:
   if k_i is a key stored in the subtree with root $c_i[x]$, then

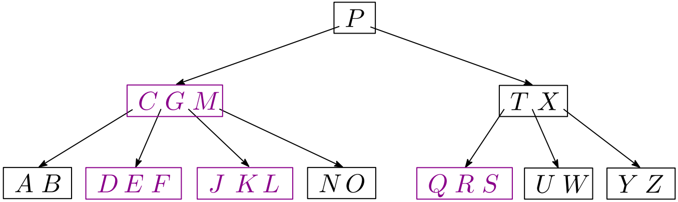
$$k_1 \leq \mathrm{key}_1[x] \leq k_2 \leq \mathrm{key}_2[x] \leq \cdots \leq \mathrm{key}_{n[x]}[x] \leq k_{n[x]+1}$$

A **B-tree** of **(minimum) degree** $t \geq 2$ has following properties:

1. Every node x (except root) has between t and $2t$ children
   - Node with $n[x]$ keys has $n[x]+1$ children.
     $\Rightarrow$ between $t-1$ and $2t-1$ keys
   - Root has at most $2t$ children
     $\Rightarrow$ at most $2t-1$ keys
2. All leaves appear on *the same level*
3. 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 nondecreasing order
   c. $n[x]+1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children
      (Leaf nodes have no children, so their c_i fields are undefined)
4. Keys $\mathrm{key}_i[x]$ separate ranges of keys in subtrees:
   if k_i is a key stored in the subtree with root $c_i[x]$, then

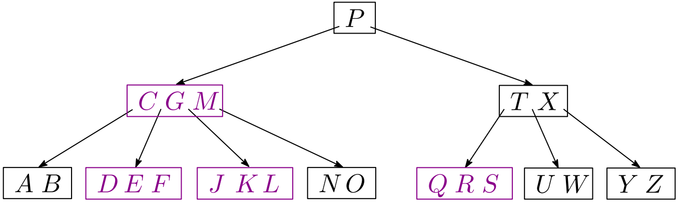
$$k_1 \leq \mathrm{key}_1[x] \leq k_2 \leq \mathrm{key}_2[x] \leq \cdots \leq \mathrm{key}_{n[x]}[x] \leq k_{n[x]+1}$$

B-Tree Example



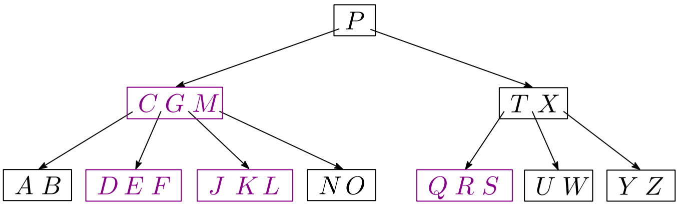
- $t = 2$: the simplest B-tree

B-Tree Example

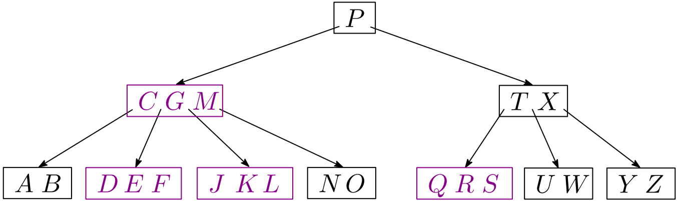


- $t = 2$: the simplest B-tree
  - Every node has at least one key.
    Every internal node has at least 2 children

B-Tree Example



- $t = 2$: the simplest B-tree
  - Every node has at least one key.
    Every internal node has at least 2 children
  - Every node has at most 3 keys.
    Every internal node has at most 4 children



- $t = 2$: the simplest B-tree
  - Every node has at least one key.
    Every internal node has at least 2 children
  - Every node has at most 3 keys.
    Every internal node has at most 4 children

- A node is **full** if it contains exactly $2t - 1$ keys (e.g., nodes colored in the above example)
- We choose t such that an internal node fits in one disk block.

Height of B-Tree

Consider the worst case

- the root contains one key

Height of B-Tree

Consider the worst case

- the root contains one key
- all other nodes contain $t-1$ keys

Height of B-Tree

Consider the worst case

- the root contains one key
- all other nodes contain $t - 1$ keys

which implies,

- 1 node at depth 0; 2 nodes at depth 1; $2t$ nodes at depth 2; $2t^2$ nodes at depth 3; ...; $2t^{h-1}$ nodes at depth h

Height of B-Tree

Consider the worst case

- the root contains one key
- all other nodes contain $t-1$ keys

which implies,

- 1 node at depth 0; 2 nodes at depth 1; $2t$ nodes at depth 2; $2t^2$ nodes at depth 3; …; $2t^{h-1}$ nodes at depth h

Thus, for any n-key B-tree of minimum degree $t \geq 2$ and height h

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1. \end{aligned}$$

Height of B-Tree

Consider the worst case

- the root contains one key
- all other nodes contain $t-1$ keys

which implies,

- 1 node at depth 0; 2 nodes at depth 1; $2t$ nodes at depth 2; $2t^2$ nodes at depth 3; $\ldots$; $2t^{h-1}$ nodes at depth h

Thus, for any n-key B-tree of minimum degree $t \geq 2$ and height h

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1. \end{aligned}$$

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

Height of B-Tree

Consider the worst case

- the root contains one key
- all other nodes contain $t-1$ keys

which implies,

- 1 node at depth 0; 2 nodes at depth 1; $2t$ nodes at depth 2; $2t^2$ nodes at depth 3; ...; $2t^{h-1}$ nodes at depth h

Thus, for any n-key B-tree of minimum degree $t \geq 2$ and height h

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1. \end{aligned}$$

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

- Compared with AVL trees, a factor of about $\log_2 t$ is saved in the number of nodes examined for most tree operations.

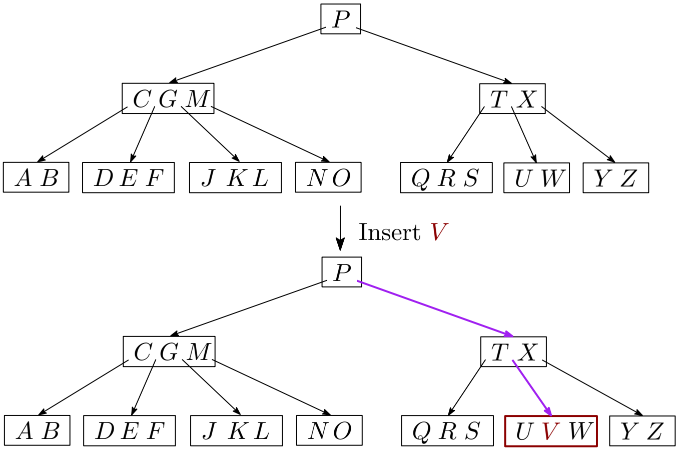
Insertion

- Basically follows insertion strategy of binary search tree

Insertion

- Basically follows insertion strategy of binary search tree
  - Insert the new key into an existing leaf node

- Basically follows insertion strategy of binary search tree
  - Insert the new key into an existing leaf node

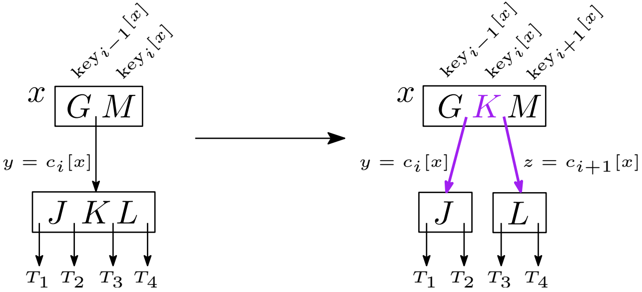


Insertion: How to insert into a full node?

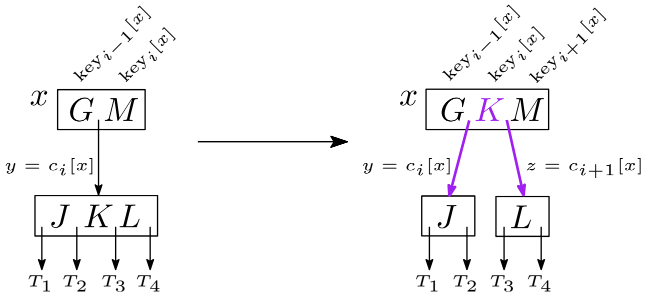
- Don't. Split full nodes BEFORE inserting into them!

Insertion: How to insert into a full node?

- Don't. Split full nodes BEFORE inserting into them!
- Given a nonfull internal node x, an index i, and a node y such that $y = c_i[x]$ is a full child of x.



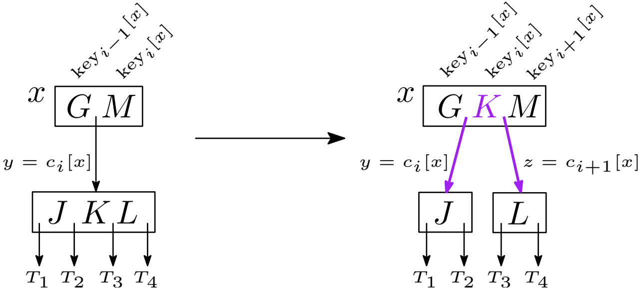
- Don't. Split full nodes BEFORE inserting into them!
- Given a nonfull internal node x, an index i, and a node y such that $y = c_i[x]$ is a full child of x.



1. split the full node y (with $2t-1$ keys) around its median key $\text{key}_t[y]$ into two nodes with $t-1$ keys each

Insertion: How to insert into a full node?

- Don't. Split full nodes BEFORE inserting into them!
- Given a nonfull internal node x, an index i, and a node y such that $y = c_i[x]$ is a full child of x.



1. split the full node y (with $2t - 1$ keys) around its median key $\text{key}_t[y]$ into two nodes with $t - 1$ keys each
2. move $\text{key}_t[y]$ up into y's parent x to separate the two nodes

Question

How can we insure that the parent of a full node is not full?

Insertion Strategy

Question

How can we insure that the parent of a full node is not full?

Answer

While moving down the tree, split every full node along the path from the root to the leaf where the new key will be inserted

Insertion Strategy

Question

How can we insure that the parent of a full node is not full?

Answer

While moving down the tree, split every full node along the path from the root to the leaf where the new key will be inserted

- A key can be inserted into a B-tree in a single pass down the tree from the root to a leaf

Insertion Strategy

Question

How can we insure that the parent of a full node is not full?

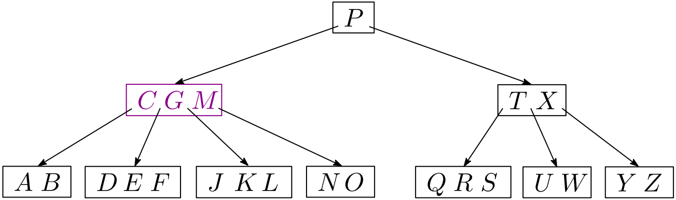
Answer

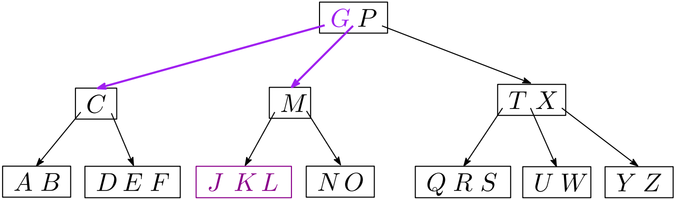
While moving down the tree, split every full node along the path from the root to the leaf where the new key will be inserted

- A key can be inserted into a B-tree in a single pass down the tree from the root to a leaf
- Splitting the root is the only way to increase the height of a B-tree

Insertion: Example

(a) initial tree

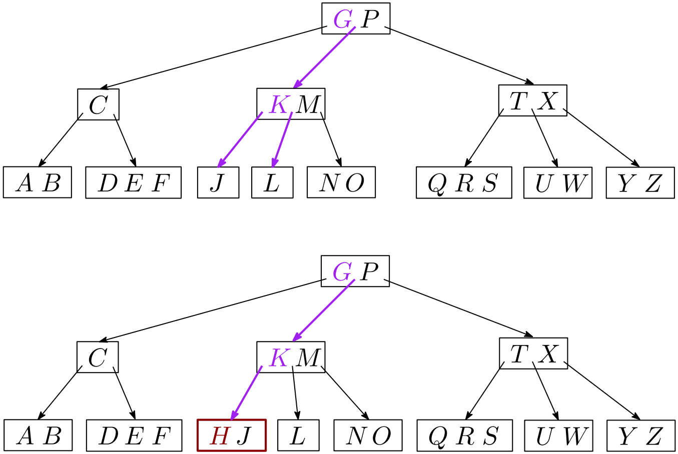




(b) insert H: split the encountered full node

Insertion: Example

(c) insert H: split the encountered full node



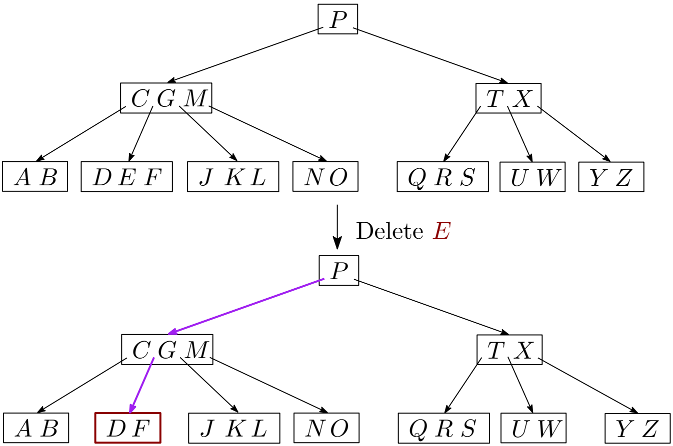
(d) insert H: insert into an existing nonfull leaf node

Deletion

- Trivial case: the leaf that contains the deleted key is not small (i.e., before deletion, it contains at least t keys)

Deletion

- Trivial case: the leaf that contains the deleted key is not small (i.e., before deletion, it contains at least t keys)



Deletion Strategy

Question

How to delete key k moving down from root without "backing up"?

Deletion Strategy

Question

How to delete key k moving down from root without "backing up"?

Answer

Remove(x, k) will remove k from subtree rooted at x.. Algorithm walks down tree towards k. Will (for non-root x) first ensure that x contains at least t keys. Then will either remove k or recursively call *Remove*(x', k'), where k' is some key (possibly not k) and x' is the root of subtree of x containing k'.

Deletion Strategy

Question

How to delete key k moving down from root without "backing up"?

Answer

Remove(x, k) will remove k from subtree rooted at x.. Algorithm walks down tree towards k. Will (for non-root x) first ensure that x contains at least t keys. Then will either remove k or recursively call *Remove*(x', k'), where k' is some key (possibly not k) and x' is the root of subtree of x containing k'.

- If k is in leaf x, condition ensures deletion is trivial

Deletion Strategy

Question

How to delete key k moving down from root without "backing up"?

Answer

$Remove(x, k)$ will remove k from subtree rooted at x.. Algorithm walks down tree towards k. Will (for non-root x) first ensure that x contains at least t keys. Then will either remove k or recursively call $Remove(x', k')$, where k' is some key (possibly not k) and x' is the root of subtree of x containing k'.

- If k is in leaf x, condition ensures deletion is trivial
- Two other, more complicated cases, to consider

Case 1 k is in the internal node x

Deletion Strategy

Question

How to delete key k moving down from root without "backing up"?

Answer

Remove(x, k) will remove k from subtree rooted at x.. Algorithm walks down tree towards k. Will (for non-root x) first ensure that x contains at least t keys. Then will either remove k or recursively call *Remove*(x', k'), where k' is some key (possibly not k) and x' is the root of subtree of x containing k'.

- If k is in leaf x, condition ensures deletion is trivial
- Two other, more complicated cases, to consider

Case 1 k is in the internal node x
Case 2 k is not in the internal node x

Deletion: Comments on root deletion

When viewing the following, note that height of tree remains unchanged *except* in special cases of 1(c) and 2(c) when

- root x contains exactly one key
- root x's two children $c_1[x]$ and $c_2[x]$ each contain exactly $t-1$ keys.

Both 1(c) and 2(c) will then merge $c_1[x]$ $key[x]$ and $c_2[x]$ into one new root node and then proceed to delete k from the new tree rooted at this new node.

We will not explicitly illustrate these cases in the following slides.

Deletion: Case 1a

Case 1: key k is in the internal node x

a. If the child y that precedes k in node x has at least t keys

Case 1: key k is in the internal node x

a. If the child y that precedes k in node x has at least t keys

1. find the predecessor k' of k in the subtree rooted at y

Deletion: Case 1a

Case 1: key k is in the internal node x

a. If the child y that precedes k in node x has at least t keys

1. find the predecessor k' of k in the subtree rooted at y
2. replace k by k' in x

Deletion: Case 1a

Case 1: key k is in the internal node x

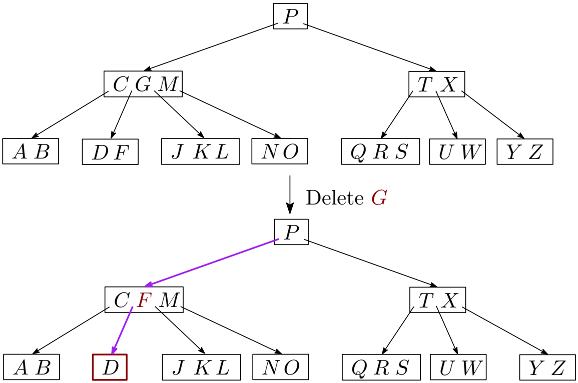
a. If the child y that precedes k in node x has at least t keys

1. find the predecessor k' of k in the subtree rooted at y
2. replace k by k' in x
3. recursively delete k'

Case 1: key k is in the internal node x

a. If the child y that precedes k in node x has at least t keys

1. find the predecessor k' of k in the subtree rooted at y
2. replace k by k' in x
3. recursively delete k'



- the predecessor F of G is moved up to take G's position

Deletion: Case 1b

Case 1: key k is in the internal node x

b. If the child z that follows k in node x has at least t keys

Deletion: Case 1b

Case 1: key k is in the internal node x

b. If the child z that follows k in node x has at least t keys

1. find the successor k' of k in the subtree rooted at z

Deletion: Case 1b

Case 1: key k is in the internal node x

b. If the child z that follows k in node x has at least t keys
   1. find the successor k' of k in the subtree rooted at z
   2. replace k by k' in x

Deletion: Case 1b

Case 1: key k is in the internal node x

b. If the child z that follows k in node x has at least t keys

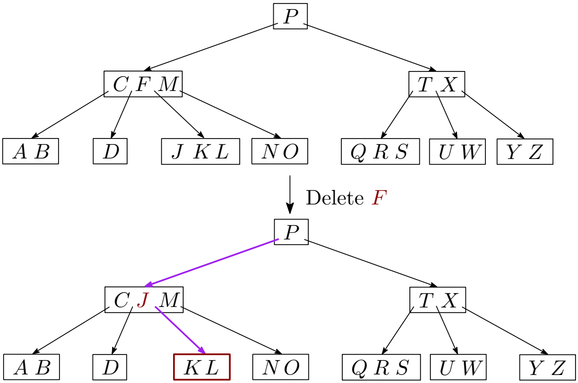
1. find the successor k' of k in the subtree rooted at z
2. replace k by k' in x
3. recursively delete k'

Deletion: Case 1b

Case 1: key k is in the internal node x

b. If the child z that follows k in node x has at least t keys

1. find the successor k' of k in the subtree rooted at z
2. replace k by k' in x
3. recursively delete k'



- the successor J of F is moved up to take F's position

Deletion: Case 1c

Case 1: key k is in the internal node x

c. If both y and z have only $t-1$ keys

Deletion: Case 1c

Case 1: key k is in the internal node x

c. If both y and z have only $t - 1$ keys

1. merge k and z into y (y now contains $2t - 1$ keys)

Deletion: Case 1c

Case 1: key k is in the internal node x

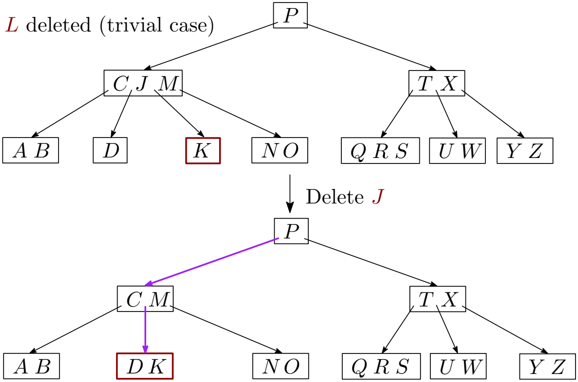
c. If both y and z have only $t-1$ keys

1. merge k and z into y (y now contains $2t-1$ keys)
2. recursively delete k from y

Case 1: key k is in the internal node x

c. If both y and z have only $t-1$ keys

1. merge k and z into y (y now contains $2t-1$ keys)
2. recursively delete k from y



- J is pushed down to make node DJK, from where J is deleted

Deletion: Case 2a

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has $> t-1$ keys then

- Recursively delete k from $c_i[x]$

Deletion: Case 2b

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

Deletion: Case 2b

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

b. If $c_i[x]$ has an immediate sibling with at least t keys

Deletion: Case 2b

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t - 1$ keys

b. If $c_i[x]$ has an immediate sibling with at least t keys
   1. give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$

Deletion: Case 2b

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

b. If $c_i[x]$ has an immediate sibling with at least t keys
   1. give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$
   2. move a key from $c_i[x]$'s immediate left or right sibling up into x

Deletion: Case 2b

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

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

Deletion: Case 2b

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

b. If $c_i[x]$ has an immediate sibling with at least t keys

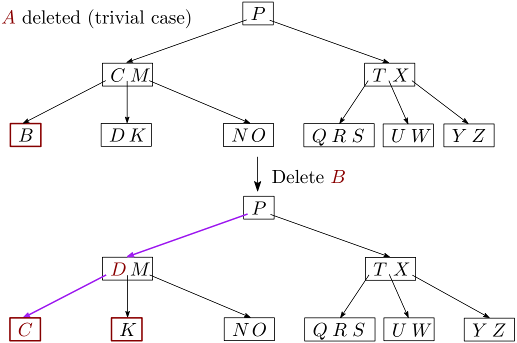
1. give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$
2. move a key from $c_i[x]$'s immediate left or right sibling up into x
3. move the appropriate child pointer from the sibling into $c_i[x]$
4. recursively delete k from the appropriate child of x

Deletion: Case 2b

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

b. If $c_i[x]$ has an immediate sibling with at least t keys

1. give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$
2. move a key from $c_i[x]$'s immediate left or right sibling up into x
3. move the appropriate child pointer from the sibling into $c_i[x]$
4. recursively delete k from the appropriate child of x



- C is moved to fill B's position, and D is moved to fill C's

Deletion: Case 2c

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

c. and both of its immediate siblings have $t-1$ keys

Deletion: Case 2c

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t - 1$ keys

c. and both of its immediate siblings have $t - 1$ keys
   1. merge $c_i[x]$ with one sibling

Deletion: Case 2c

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

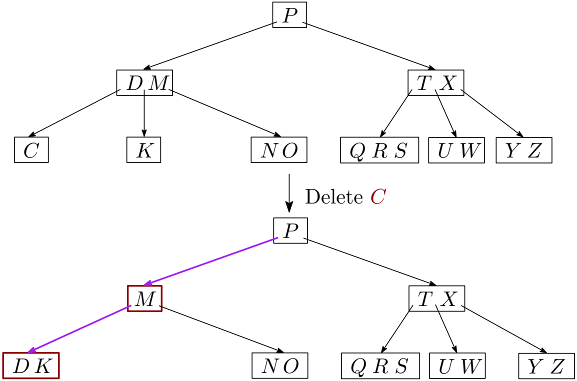
c. and both of its immediate siblings have $t-1$ keys

1. merge $c_i[x]$ with one sibling
2. recursively delete k from the appropriate child of x

Case 2: the key k is not in the internal node x, then determine the root $c_i[x]$ whose subtree contains k. If $c_i[x]$ has only $t-1$ keys

c. and both of its immediate siblings have $t-1$ keys

1. merge $c_i[x]$ with one sibling
2. recursively delete k from the appropriate child of x



- D is pushed down to get node CDK, from where C is deleted

B-Trees: More

- Saw how to maintain a B-tree using $\log_t n$ "operations"
  - each operation requires constant number of disk reads.
  - could also require many internal memory operations
- For "large" t; useful for storing large databases on disk
  - with each node a disk page

- B-Trees created by Bayer and McCreight at Boeing in 1972
- B^+ tree variant keeps data keys in leaves

- Simplest B-Tree is $(2-3-4)$-tree
  - Balanced tree good for internal memory storage

- Another variation is (a, b)-trees: all non-root nodes have between a and b children
  - Is a B-tree if $b = 2a$.
  - Smallest (non B-Tree) version is $(2, 3)$-trees