Prepared by: [Arun Konagurthu]

FIT3155 S1/2020: Algorithms and Data Structures

Week 7: **B-Trees: A generalized balanced Search Tree**

Faculty of Information Technology, Monash University

What is covered in this lecture?

- A **generalization** of the balanced search trees
- **Standard operations** on B-trees (search, insert, delete)
- Space and time **complexity** issues

Source material and recommended reading

- Cormen et al. "Introduction to Algorithms" (Chapter 18) [link]

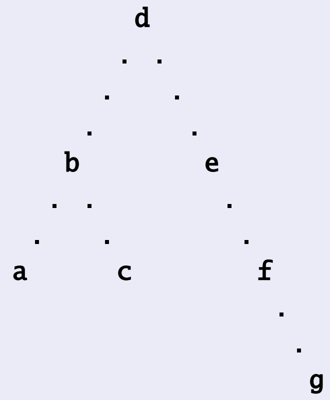
Original papers

- R. Bayer and E. McCreight
  - *Organization and Maintenance of Large Ordered Indices*, **Mathematical and Information Sciences** Report No. 20, Boeing Scientific Research Laboratories. 1970
- R. Bayer,
  - *Binary B-Trees for Virtual Memory*, **Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control**, San Diego, California. November 1112, 1971.

Revision from FIT2004: **Balanced** **Binary** Search Trees

- The empty tree is a balanced BST
- If the tree is not empty,
  1. the elements in the **left subtree** are $\leq$ the element in the root
  2. the elements in the **right subtree** are $>$ the element in the root
  3. the **left subtree is a balanced BST**
  4. the **right subtree is a balanced BST**

BST example



Detour: Data structures on secondary storage

- Computers, as you know, has different kinds of storage/memory.

  primary storage:

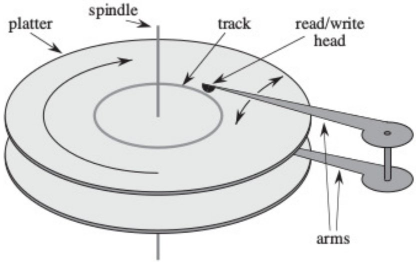
  - The **main memory** consists of silicon chip.
  - Orders of magnitude more expensive per bit stored (compared to magnetic storages).

  secondary storage:

  - Mostly, based on magnetic storage technology. (Solid state technology is surging.)
  - Orders of magnitude more space (compared to primary storages).

Detour: A typical disk drive

- A typical secondary (external) drive consists of:
  - one or more **`platters`** covered with magnetizable material that rotate around a **`spindle`**.
  - Read and write to each platter is via a **`head`** attached to an **`arm`**.
  - The arms can move their heads towards or away from the spindle.
  - When the arm is stationary, the surface below the head is called a **`track`**.



Detour: Read/Write Access times

- Secondary storage is much slower because they have moving parts
  - Platter rotations (5,400-15,000 RPM).
  - Arm head forward/backward movement.
- At 7,200 RPM, one rotation takes 8.33 milliseconds, which is...
- ... 5 orders of magnitude $>$ than 50 nanoseconds access time to main memory.
- That is, in time to wait for one read/write on a secondary storage, we can access primary memory 100,000 times during that span.
- Average access times for commodity disks are in the range 8 to 11 milliseconds.

Detour: Disk access a **page** worth of information at a time

- To amortize the time wasted during mechanical movements, each disk access reads
  - not just one item but several items at a time.
- Information is divided (logically) into equal-sized **pages** of bits.
- Each access reads/writes one or more **pages** worth of information at a time.
- A page contains from 4KB to 64KB (in some cases) worth of information.
- The number of disk accesses is measured in terms of the number of pages of information.

Disk based search tree structures

- Consider the cases when we have large (dynamic) database/dictionary, too big to fit in main memory.
- Accessing its information, we have the following cost model to optimize:
  - Minimize expected number of disk accesses during various operations (insert/delete/search).
  - Keep space requirement to $O(n)$.
- Binary trees such as AVL trees are not optimal for disk-based representations.
- Generalization to multi-way search trees greatly reduce the accesses.
- **`B-tree`** (or its variants) is a very practical, widely-used data structure for this purpose.

B-trees: Introduction

- B-trees **generalize** **balanced** search trees

B-trees: Introduction

- B-trees **generalize** **balanced** search trees
- By generalization, the tree is **no longer** **`binary`** but has **`many branches per node`**.

B-trees: Introduction

- B-trees **generalize** **balanced** search trees
- By generalization, the tree is **no longer** **binary** but has **many branches per node**.
- They are really powerful and are used on many mission critical systems that rely on a large amount of data stored on a **secondary storage** device (disk)

B-trees: Introduction

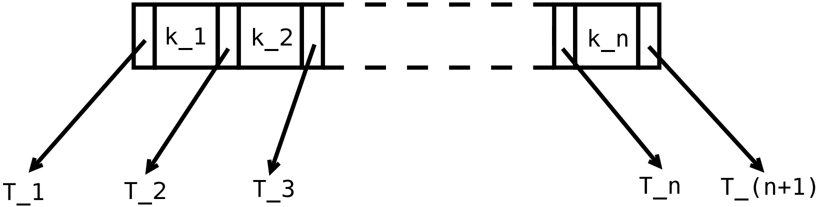
- B-trees **generalize** **balanced** search trees
- By generalization, the tree is **no longer** **`binary`** but has **`many branches per node`**.
- They are really powerful and are used on many mission critical systems that rely on a large amount of data stored on a **`secondary storage`** device (disk)
- Common examples are **very large** **databases** and **filesystems**

B-tree properties

1. A **B-tree** is a **rooted tree**.

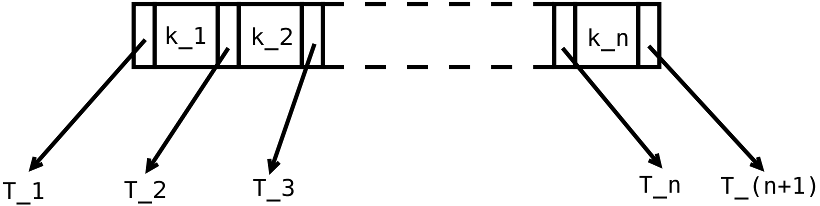
B-tree properties

1. A **B-tree** is a **rooted tree**.
2. An arbitrary node looks like this:



B-tree properties

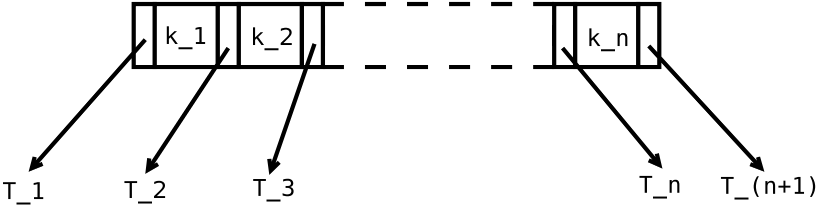
1. A **B-tree** is a **rooted tree**.
2. An arbitrary node looks like this:



3. It has n elements: $k_1 \leq k_2 \leq \cdots \leq k_n$ that are stored in a **non-decreasing** (sorted) order.

B-tree properties

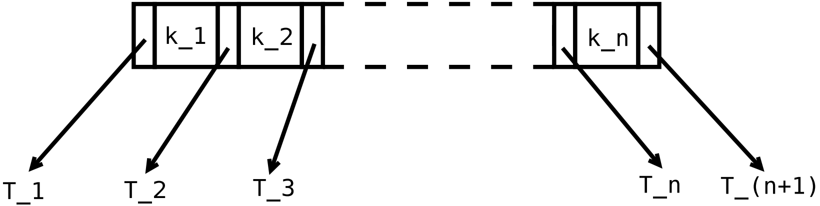
1. A **B-tree** is a **rooted tree**.
2. An arbitrary node looks like this:



3. It has n elements: $k_1 \leq k_2 \leq \cdots \leq k_n$ that are stored in a **non-decreasing** (sorted) order.
4. It also stores $n+1$ **pointers/links to subtrees**: $T_1, T_2, \cdots, T_n, T_{n+1}$ that are **distributed regularly** between the elements.

B-tree properties

1. A **B-tree** is a **rooted tree**.
2. An arbitrary node looks like this:



3. It has n elements: $k_1 \leq k_2 \leq \cdots \leq k_n$ that are stored in a **non-decreasing** (sorted) order.
4. It also stores $n+1$ **pointers/links to subtrees**: $T_1, T_2, \cdots, T_n, T_{n+1}$ that are **distributed regularly** between the elements.
5. Each successive (k_i, T_i, T_{i+1}) resembles a **fork**/node of a **binary** tree and has the BST **structural property**: $\{T_i\} \leq k_i \leq \{T_{i+1}\}$. Or, generalizing

$$\{T_1\} \leq k_1 \leq \{T_2\} \leq k_2 \leq \{T_3\} \leq \cdots \leq \{T_n\} \leq k_n \leq \{T_{n+1}\}$$

B-tree properties...continued

6. All leaves have the **same depth** (from the root).

B-tree properties...continued

6. All leaves have the **same depth** (from the root).
7. **IMPORTANT!** The **number of elements per node** has a **lower bound** and an **upper bound**. These bounds are expressed using the parameter $t \geq 2$, also called the **minimum degree** of the B-tree:

B-tree properties...continued

6. All leaves have the **same depth** (from the root).
7. **IMPORTANT!** The **number of elements per node** has a **lower bound** and an **upper bound**. These bounds are expressed using the parameter $t \geq 2$, also called the **minimum degree** of the B-tree:

Lower bound: Each node must have **AT LEAST** t **subtrees** as children or equivalently $t-1$ elements.

B-tree properties...continued

6. All leaves have the **same depth** (from the root).
7. **IMPORTANT!** The **number of elements per node** has a **lower bound** and an **upper bound**. These bounds are expressed using the parameter $t \geq 2$, also called the **minimum degree** of the B-tree:

Lower bound: Each node must have **AT LEAST** t **subtrees** as children or equivalently $t-1$ elements.

- The only exception to this bound is for the root node.

B-tree properties...continued

6. All leaves have the **same depth** (from the root).
7. **IMPORTANT!** The **number of elements per node** has a **lower bound** and an **upper bound**. These bounds are expressed using the parameter $t \geq 2$, also called the **minimum degree** of the B-tree:

Lower bound: Each node must have **AT LEAST** t **subtrees** as children or equivalently $t-1$ elements.

- The only exception to this bound is for the root node.
- A root node with **0** elements is an **empty** B-tree.

B-tree properties...continued

6. All leaves have the **same depth** (from the root).
7. **IMPORTANT!** The **number of elements per node** has a **lower bound** and an **upper bound**. These bounds are expressed using the parameter $t \geq 2$, also called the **minimum degree** of the B-tree:

   Lower bound: Each node must have **AT LEAST** t **subtrees** as children or equivalently $t - 1$ elements.
   - The only exception to this bound is for the root node.
   - A root node with **0** elements is an **empty** B-tree.
   - root node with $\geq$ **1** elements is a **non-empty** B-tree.

B-tree properties...continued

6. All leaves have the **same depth** (from the root).
7. **IMPORTANT!** The **number of elements per node** has a **lower bound** and an **upper bound**. These bounds are expressed using the parameter $t \geq 2$, also called the **minimum degree** of the B-tree:

Lower bound: Each node must have **AT LEAST** t **subtrees** as children or equivalently $t-1$ elements.

- The only exception to this bound is for the root node.
- A root node with **0** elements is an **empty** B-tree.
- root node with $\geq$ **1** elements is a **non-empty** B-tree.

upper bound: Every node must have **AT MOST** $2t$ subtrees (or equivalently $2t-1$ elements). A node of a B-tree is **full** if it reaches this bound.

B-tree properties...continued

6. All leaves have the **same depth** (from the root).
7. **IMPORTANT!** The **number of elements per node** has a **lower bound** and an **upper bound**. These bounds are expressed using the parameter $t \geq 2$, also called the **minimum degree** of the B-tree:

Lower bound: Each node must have **AT LEAST** t **subtrees** as children or equivalently $t-1$ elements.

- The only exception to this bound is for the root node.
- A root node with **0** elements is an **empty** B-tree.
- root node with $\geq$ **1** elements is a **non-empty** B-tree.

upper bound: Every node must have **AT MOST** $2t$ subtrees (or equivalently $2t-1$ elements). A node of a B-tree is **full** if it reaches this bound.

B-tree properties...continued

6. All leaves have the **same depth** (from the root).
7. **IMPORTANT!** The **number of elements per node** has a **lower bound** and an **upper bound**. These bounds are expressed using the parameter $t \geq 2$, also called the **minimum degree** of the B-tree:

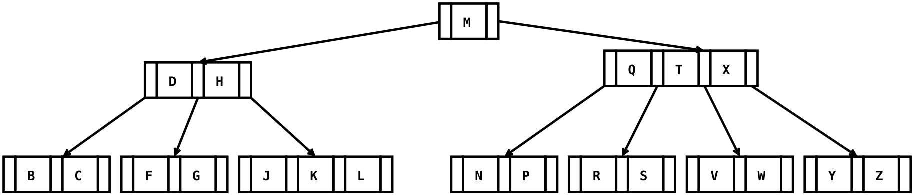
Lower bound: Each node must have **AT LEAST** t **subtrees** as children or equivalently $t - 1$ elements.

- The only exception to this bound is for the root node.
- A root node with **0** elements is an **empty** B-tree.
- root node with $\geq$ **1** elements is a **non-empty** B-tree.

upper bound: Every node must have **AT MOST** $2t$ subtrees (or equivalently $2t - 1$ elements). A node of a B-tree is **full** if it reaches this bound.

The simplest B-tree is when $t = 2$. Every internal node has either **2, 3, or 4 subtrees** connecting its **1, 2 or 3 elements**. This is specifically called a [2-3-4] tree.

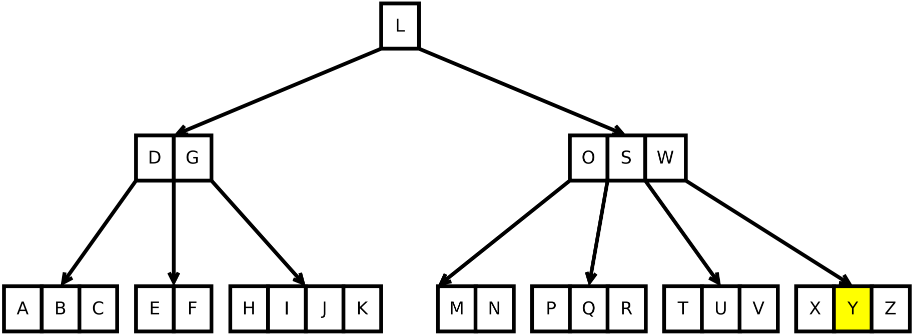
Example of a B-tree containing consonants of the English Alphabet



Searching for element **x** in a B-tree

- Searching in B-tree is no different from searching in any **binary** search tree..
- However, since each node has $t \geq 2$ children, the decision is no longer **two-way** or **binary**.
- A **multi-way** branching decision has to be made according to its number of children/subtrees.

Search for '**Y**' in a B-tree made from all letters from the English alphabet.



Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **`binary`** search tree (BST).

Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **`binary`** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.

Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **binary** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.
- However, with B-trees we simply cannot create a new leaf node and insert it, as it can flout the B-tree property.

Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **binary** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.
- However, with B-trees we simply cannot create a new leaf node and insert it, as it can flout the B-tree property.
- In B-trees, a new key is inserted into an **existing leaf node**.

Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **binary** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.
- However, with B-trees we simply cannot create a new leaf node and insert it, as it can flout the B-tree property.
- In B-trees, a new key is inserted into an **existing leaf node**.
  - However, this is possible only if the leaf node is **NOT** full.

Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **binary** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.
- However, with B-trees we simply cannot create a new leaf node and insert it, as it can flout the B-tree property.
- In B-trees, a new key is inserted into an **existing leaf node**.
  - However, this is possible only if the leaf node is **NOT** full.
  - If the leaf node is **full**

Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **binary** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.
- However, with B-trees we simply cannot create a new leaf node and insert it, as it can flout the B-tree property.
- In B-trees, a new key is inserted into an **existing leaf node**.
  - However, this is possible only if the leaf node is **NOT** full.
  - If the leaf node is **full**
    - the leaf node is **split** around its **median** element to form 2 nodes.

Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **binary** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.
- However, with B-trees we simply cannot create a new leaf node and insert it, as it can flout the B-tree property.
- In B-trees, a new key is inserted into an **existing leaf node**.
  - However, this is possible only if the leaf node is **NOT** full.
  - If the leaf node is **full**
    - the leaf node is **split** around its **median** element to form 2 nodes.
    - The median element **moves up** to the **parent node** of the original leaf node, to identify the dividing point of the two split nodes.

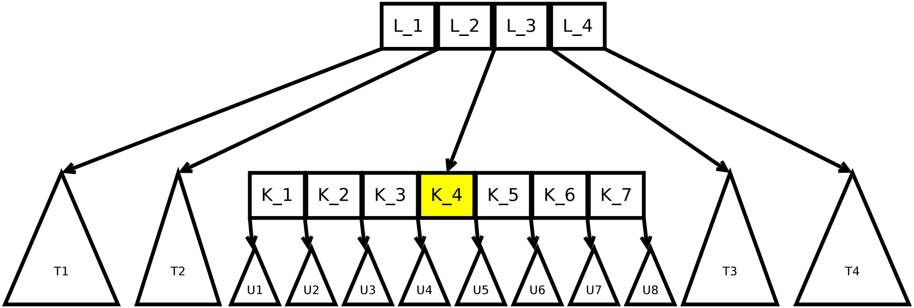
Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **binary** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.
- However, with B-trees we simply cannot create a new leaf node and insert it, as it can flout the B-tree property.
- In B-trees, a new key is inserted into an **existing leaf node**.
  - However, this is possible only if the leaf node is **NOT** full.
  - If the leaf node is **full**
    - ★ the leaf node is **split** around its **median** element to form 2 nodes.
    - ★ The median element **moves up** to the **parent node** of the original leaf node, to identify the dividing point of the two split nodes.
    - ★ However, if the parent node is **full**, **split** the node to make room for the median to be pushed up (before insertion) into parent.

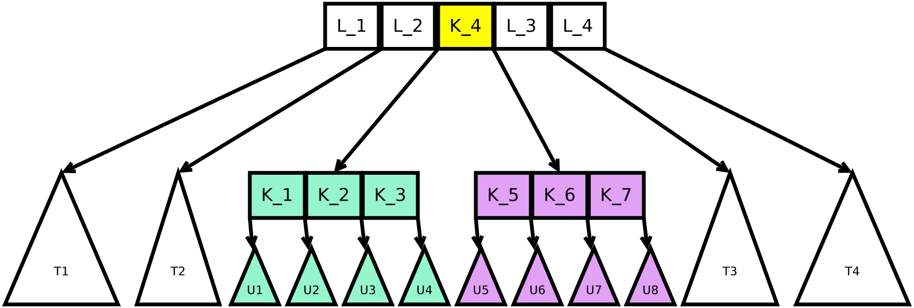
Inserting into a B-tree

- Inserting into a B-tree is **slightly more** involved than inserting into a **binary** search tree (BST).
- As with a BST, we first search for a leaf position at which to insert the new key.
- However, with B-trees we simply cannot create a new leaf node and insert it, as it can flout the B-tree property.
- In B-trees, a new key is inserted into an **existing leaf node**.
  - However, this is possible only if the leaf node is **NOT** full.
  - If the leaf node is **full**
    - the leaf node is **split** around its **median** element to form 2 nodes.
    - The median element **moves up** to the **parent node** of the original leaf node, to identify the dividing point of the two split nodes.
    - However, if the parent node is **full**, **split** the node to make room for the median to be pushed up (before insertion) into parent.
    - This implies, when traversing from root to leaf in a B-tree, **split** any **full parent node** of a (sub)tree along the insertion path.

Splitting illustration (general case)



Splitting illustration (general case)

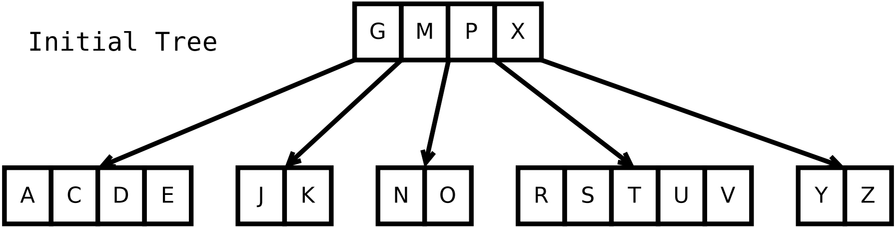


Inserting into a B-tree of degree $t = 3$

Note:

Lower bound on the number of **elements**: $t - 1 = 2$ (except root node)

Upper bound on the number of **elements**: $2t - 1 = 5$

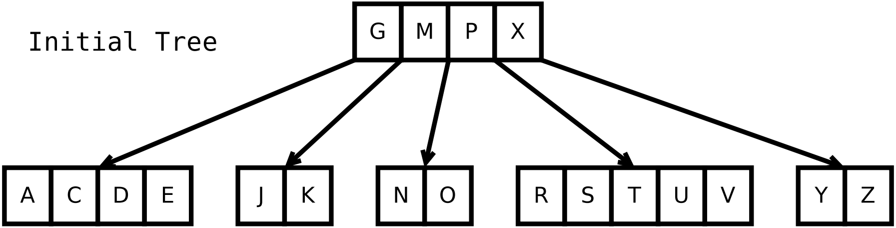


Inserting into a B-tree of degree $t = 3$

Note:

Lower bound on the number of **elements**: $t - 1 = 2$ (except root node)
Upper bound on the number of **elements**: $2t - 1 = 5$

Initial Tree

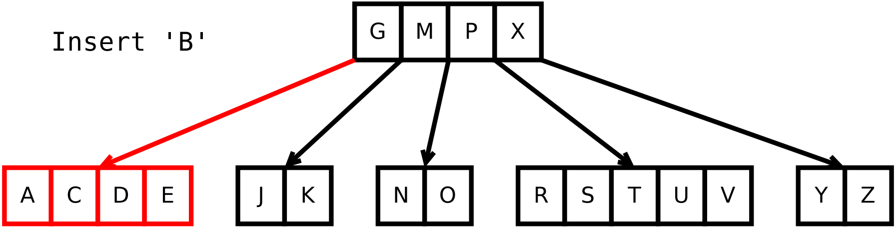


Next

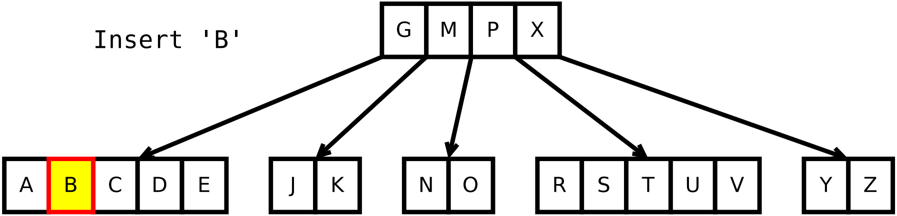
insert('B') into this tree.

Inserting into a B-tree of degree $t = 3$

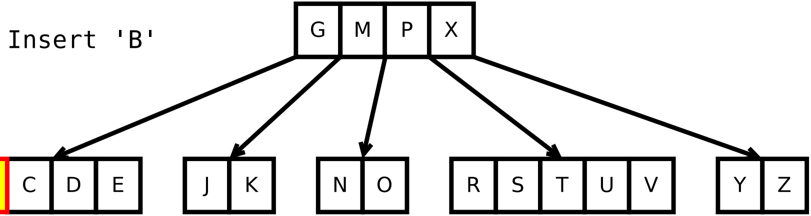
To insert element **B**, traverse along an 'appropriate' subtree



Inserting into a B-tree of degree $t = 3$



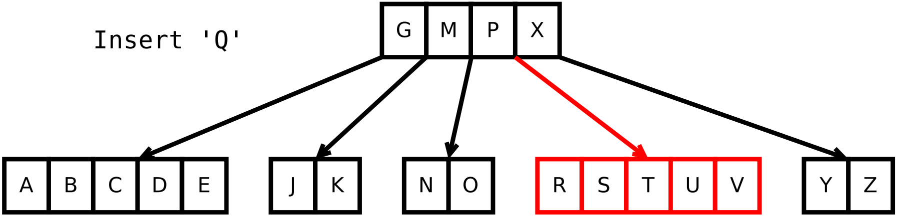
Inserting into a B-tree of degree $t = 3$



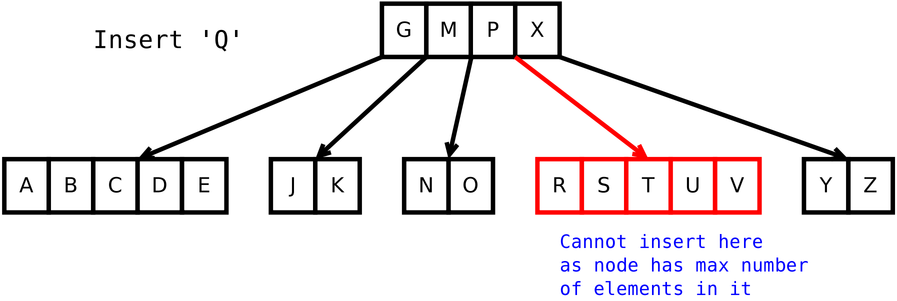
Next

insert('Q') into this tree.

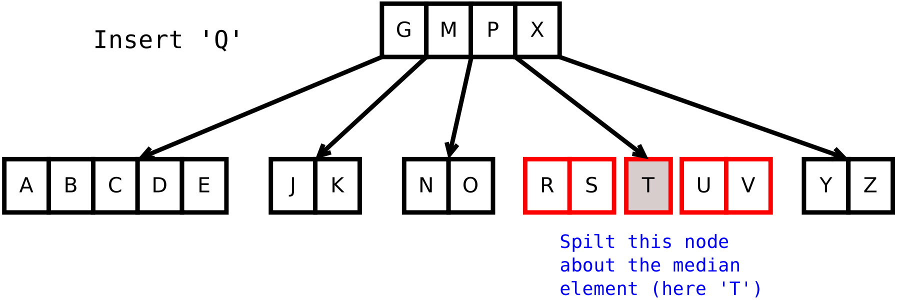
Inserting into a B-tree of degree $t = 3$



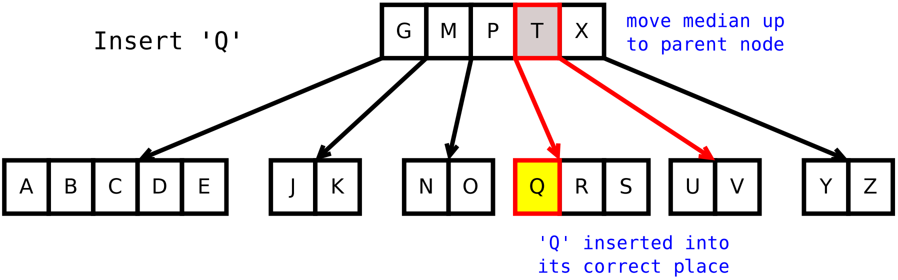
Inserting into a B-tree of degree $t = 3$



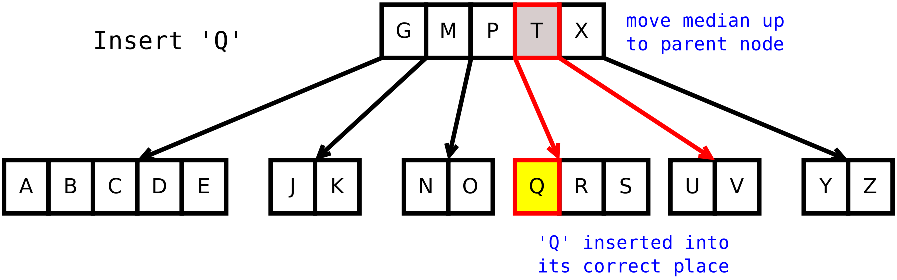
Inserting into a B-tree of degree $t = 3$



Inserting into a B-tree of degree $t = 3$



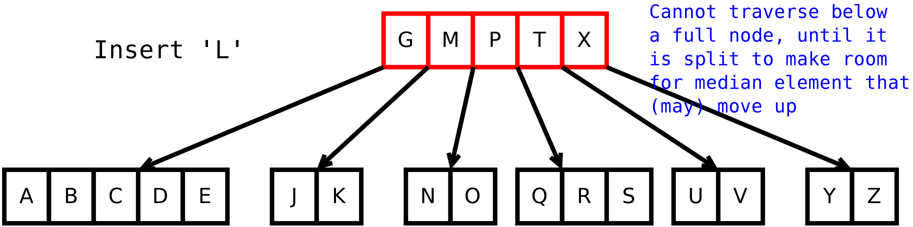
Inserting into a B-tree of degree $t = 3$



Next

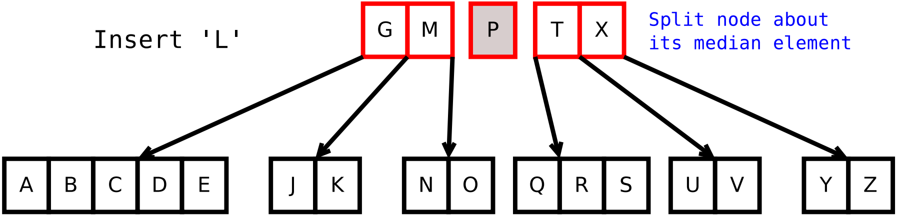
insert('L') into this tree.

Inserting into a B-tree of degree $t = 3$

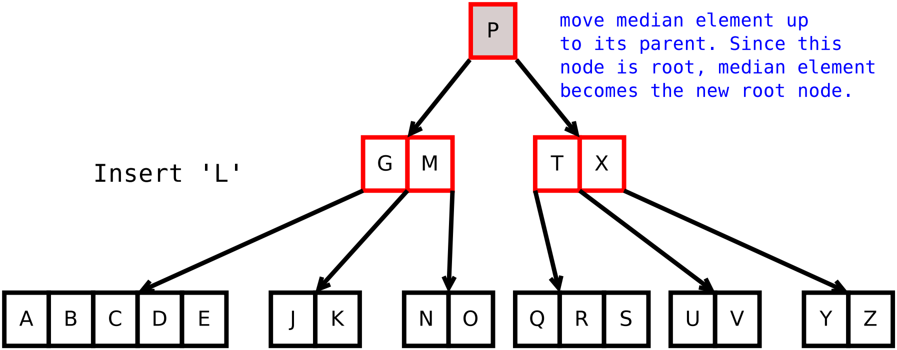


Inserting into a B-tree of degree $t = 3$

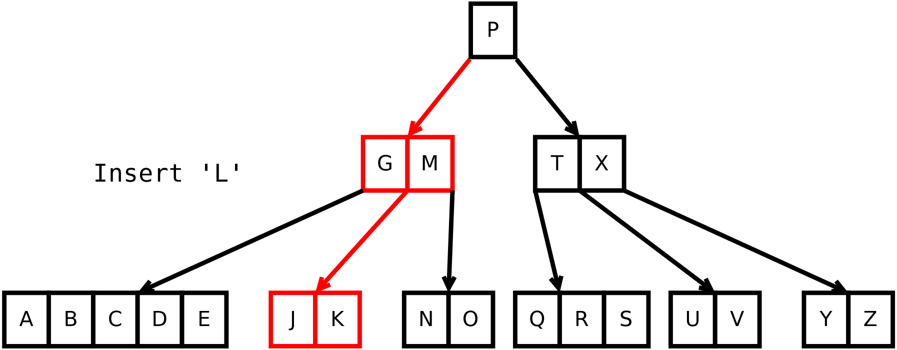
Insert 'L'



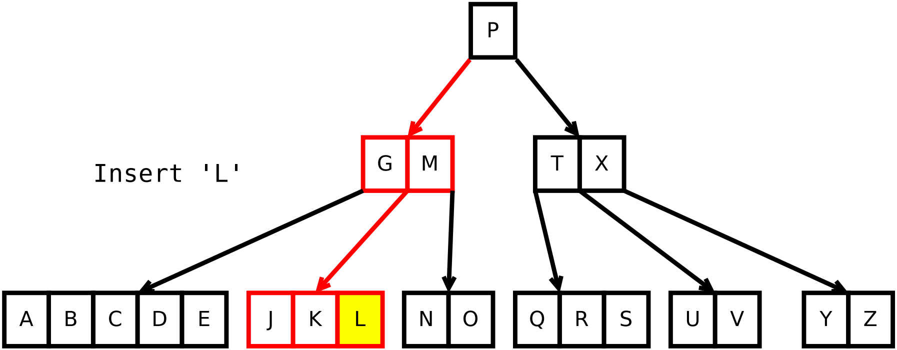
Inserting into a B-tree of degree $t = 3$



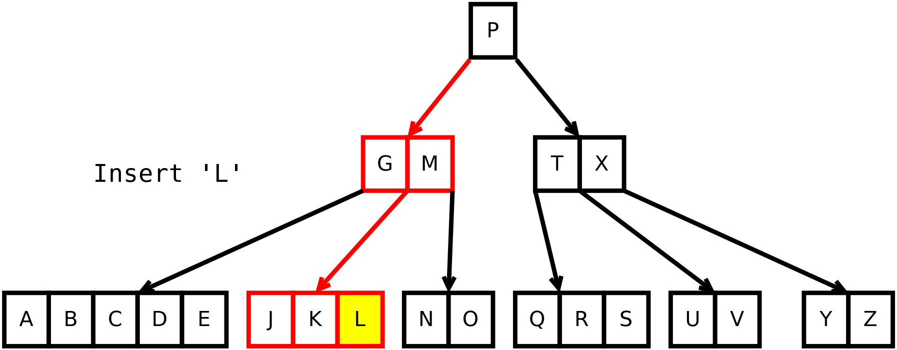
Inserting into a B-tree of degree $t = 3$



Inserting into a B-tree of degree $t = 3$



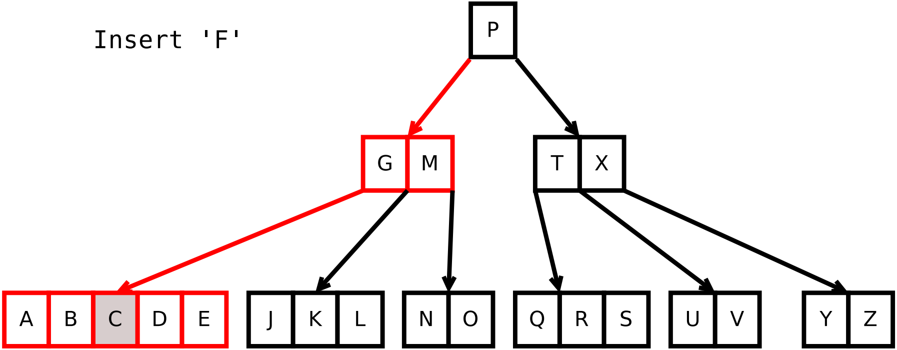
Inserting into a B-tree of degree $t = 3$



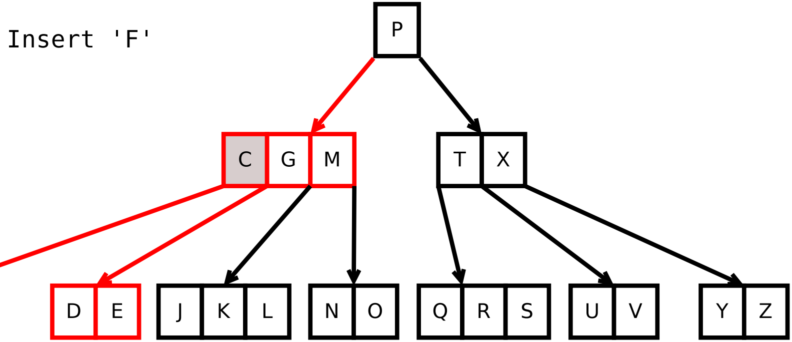
Next

insert('F') into this tree.

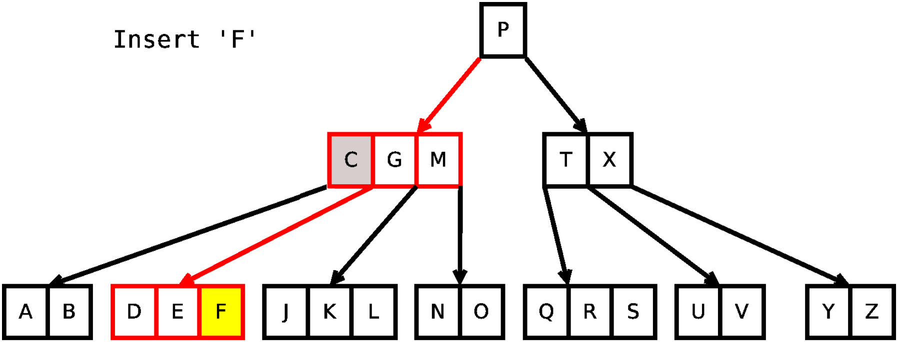
Inserting into a B-tree of degree $t = 3$



Inserting into a B-tree of degree $t = 3$



Inserting into a B-tree of degree $t = 3$



Deleting **x** from a B-tree – Case 1

- **Case** 1: If **x** belongs to a **leaf node** with $\geq t$ elements. Delete **x** – **trivial**!

Deleting **x** from a B-tree – Case 2

- **Case** 2:If **x** belongs to an **internal node**:
  a. If the left child node of **x** has at least t elements, then find the **in-order predecessor** (say, w) in the left subtree, replace **x** with w, and then recursively delete w in the left subtree.
  b. Else if the right child node of **x** has at least t elements, then find the **in-order successor** (say, y) in the right subtree, replace **x** with y, and then recursively delete y in the right subtree.
  c. Else, both **left** and **right** child nodes of **x** have **exactly** $t-1$ elements, then merge **x** and child nodes, and recursively delete **x**.

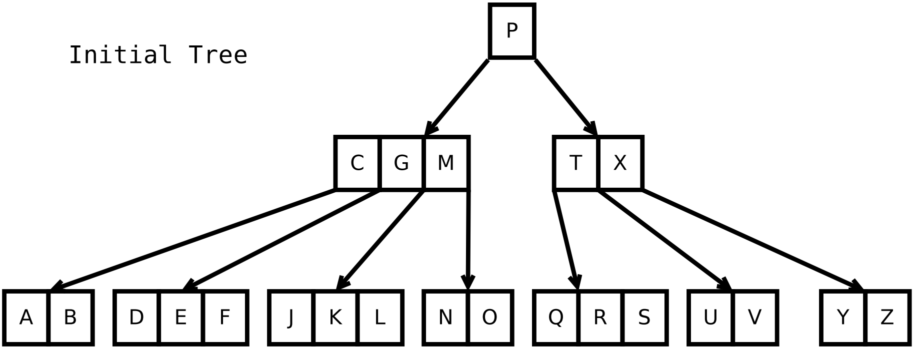
Deleting **x** from a B-tree – Case 3

- **Case 3**: If the **traversal** is stopped because the 'appropriate' subtree containing **x** has a node with **exactly** $t-1$ elements, then :
  a. if this subtree's '**immediate sibling**' has **at least** t elements, give an extra element to the appropriate subtree by **rotating** the predecessor or successor element within the sibling node to parent, followed by moving the original parent element to the appropriate subtree.
  b. else, if its **immediate sibling** also has **exactly** $t-1$ elements, merge the parent element with both sibling elements to form a single node. This **may** reduce the height of the tree.

Deleting from a B-Tree (degree $t = 3$): Running examples

Note:

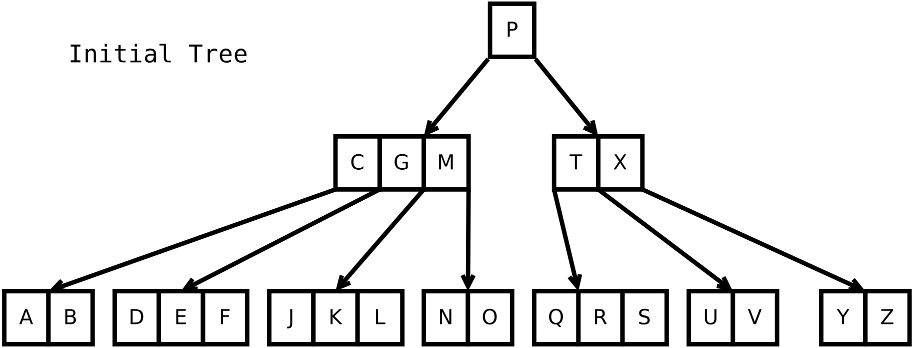
Number of elements **lowerbound** $t - 1 = 2$ (except the root node)
Number of elements **upperbound** $2t - 1 = 5$



Deleting from a B-Tree (degree $t = 3$): Running examples

Note:

Number of elements **lowerbound** $t - 1 = 2$ (except the root node)
Number of elements **upperbound** $2t - 1 = 5$

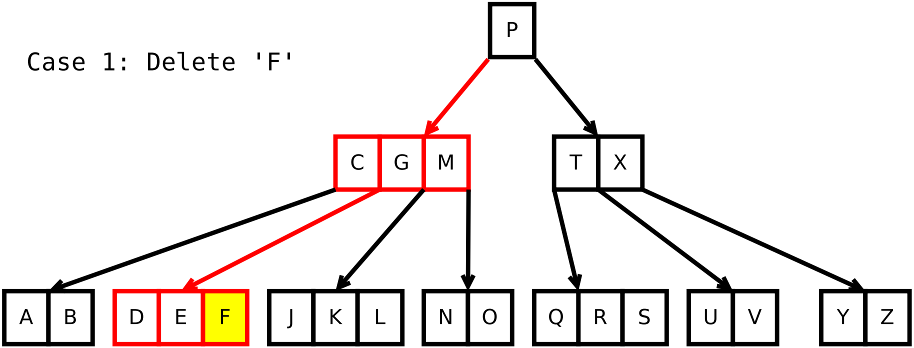


Next

delete('F') from this tree.

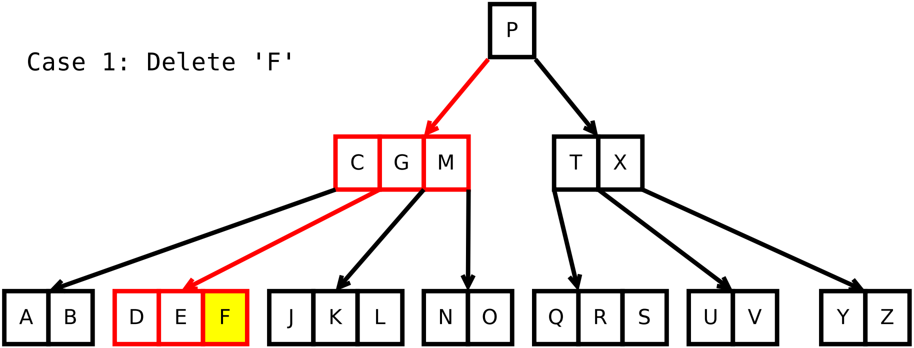
Delete 'F' – Case 1 example

1. $\mathbf{x}$='F' belongs to a **leaf node** with $\geq t(=3)$ elements.



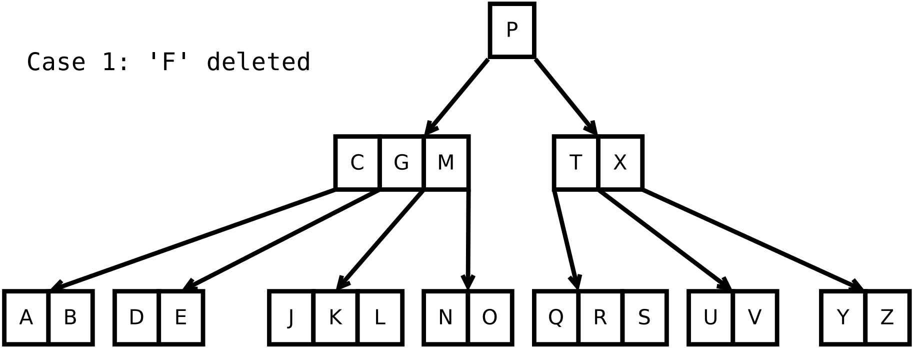
Delete 'F' – Case 1 example

1. **x**='F' belongs to a **leaf node** with $\geq t(=3)$ elements.

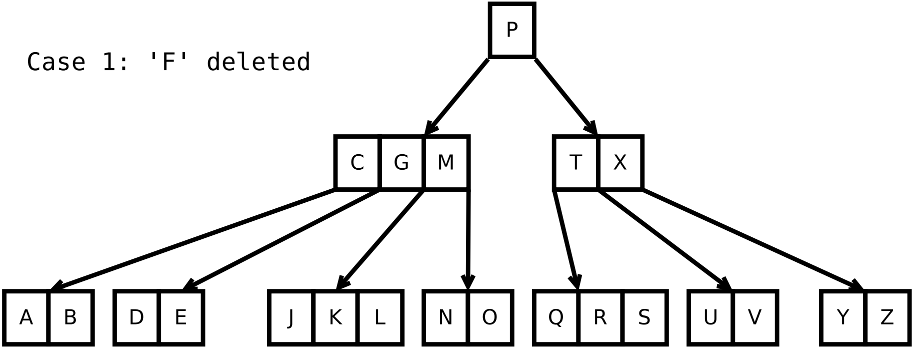


Since 'F' is a leaf node with $\geq t(=3)$ elements, deleting 'F' involves just removing that element – **trivial**!

Delete 'F' – Case 1 example



Delete 'F' – Case 1 example

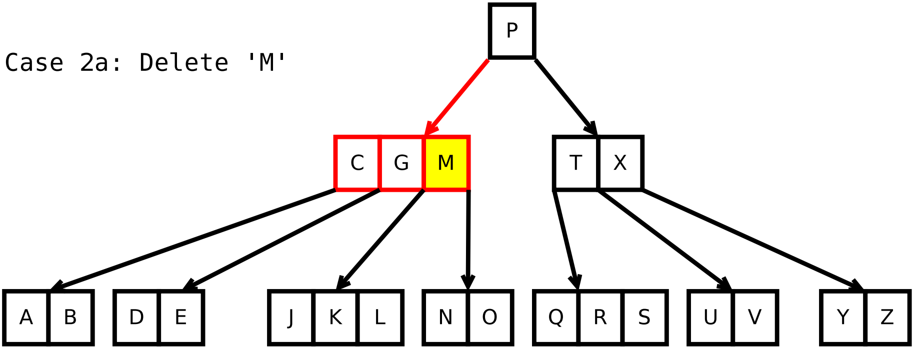


Next

`delete`('M') from this tree.

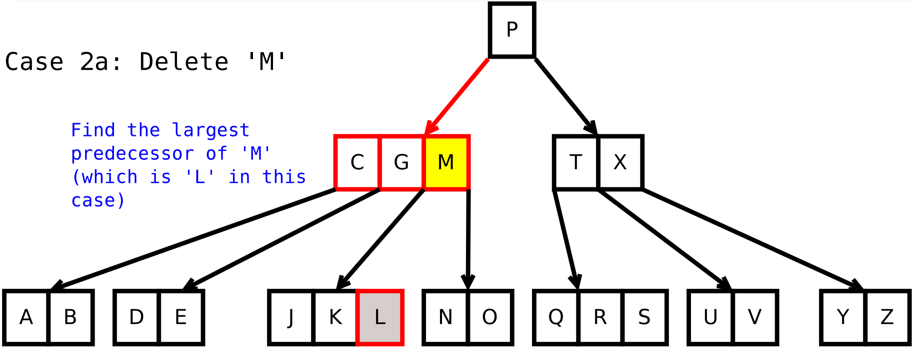
Delete ‘M’ – Case 2a example

2. $\mathbf{x}$ = ‘M’ belongs to an **internal node**:
   a. If the left child node of $\mathbf{x}$ has at least t elements, then find the **in-order predecessor** (say, $\mathtt{w}$ = ‘L’ in this example) in the left subtree, replace $\mathbf{x}$ with $\mathtt{w}$, and then recursively delete $\mathtt{w}$ in the left subtree.



Delete 'M' – Case 2a example (continued)

In-order predecessor of any element **x** in the B-tree is the largest element that is $\leq$ **x**. This can be derived by traversing the left subtree of **x** and accessing the **rightmost** element in that subtree.

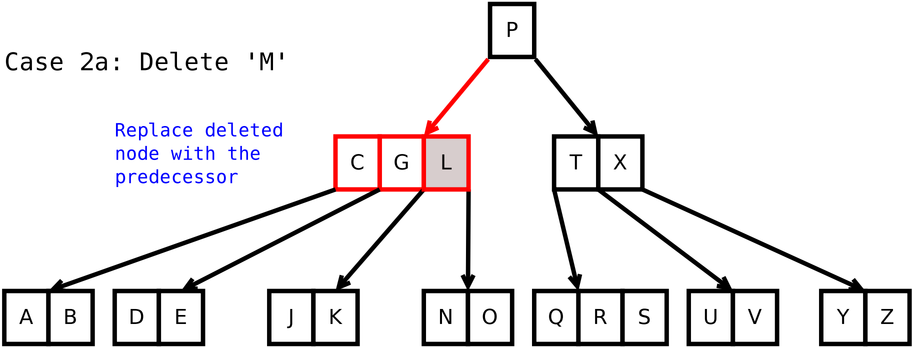


In this example, the **in-order predecessor** of 'M' is 'L'.
To delete 'M', replace 'M' with its **predecessor** 'L', and (recursively) delete 'L' in the left subtree (next slide)

Delete ‘M’ – Case 2a example (continued)

Case 2a: Delete 'M'

Replace deleted node with the predecessor



Case 2b mirrors Case 2a

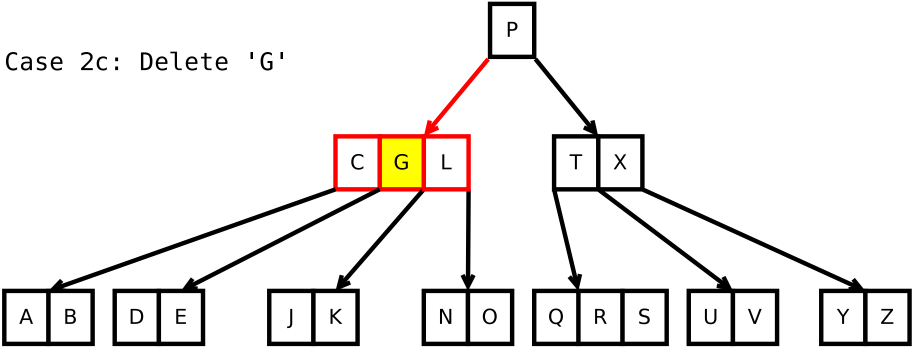
2. If **x** belongs to an **internal node**:
   - (b) ... if the right child node of **x** has at least t elements, then find the **in-order successor** (say, y) in the right subtree, replace **x** with y, and then recursively delete y in the right subtree.

In-order successor of any element **x** in the B-tree is the smallest element that is $>$ **x**. This can be derived by traversing the right subtree of **x** and accessing the **leftmost** element in that subtree.

No explicit example needed to demonstrate Case 2b, as it is symmetric (or mirror operation) to Case 2a, whose example we have seen above.

Delete 'G' – Case 2c example

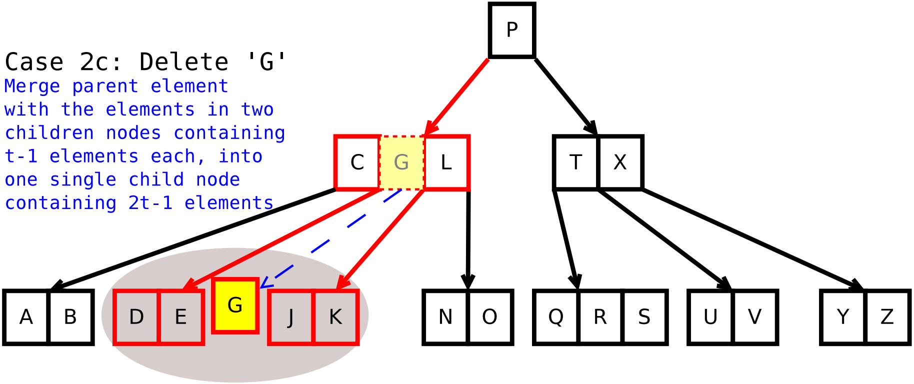
2. If **x**='G' belongs to an **internal node**:
   c. and both **left** and **right** child nodes of **x** have **exactly** $t-1=2$ elements, then merge **x** and child nodes, and recursively delete **x**.



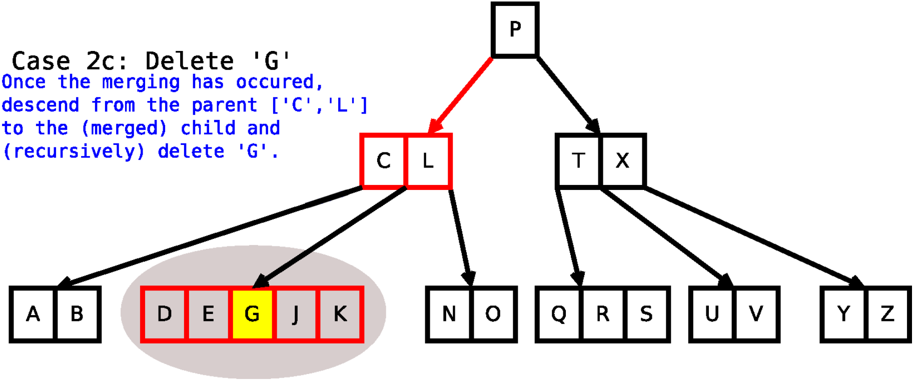
Delete 'G' – Case 2c example (continued)

```
Case 2c: Delete 'G'
```

```
Merge parent element
with the elements in two
children nodes containing
t-1 elements each, into
one single child node
containing 2t-1 elements
```

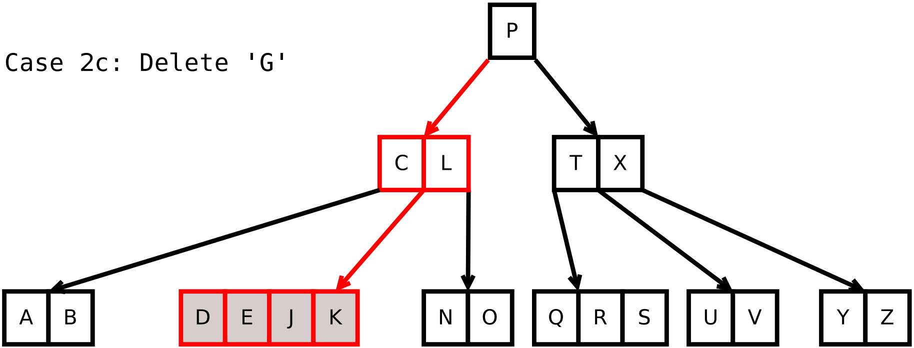


Delete ‘G’ – Case 2c example (continued)



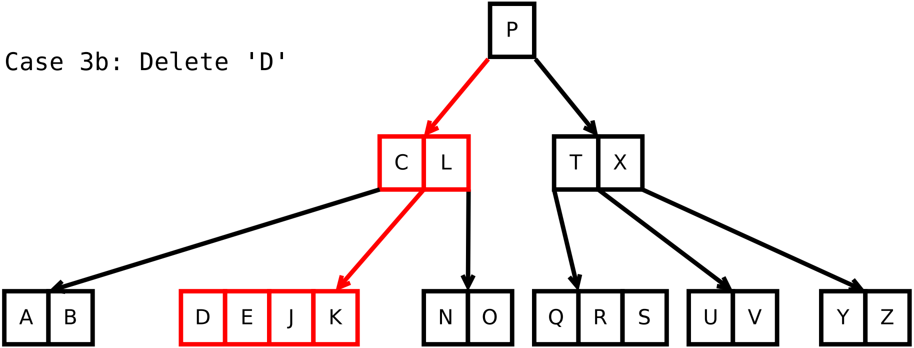
Delete 'G' – Case 2c example (continued)

`Case 2c: Delete 'G'`



Delete 'D' – Case 3b example*

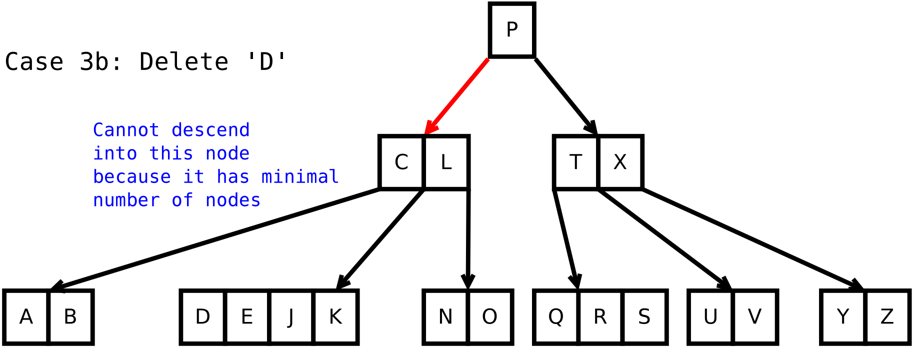
Case 3b: Delete 'D'



*we will first see case 3b before case 3a, stay tuned!

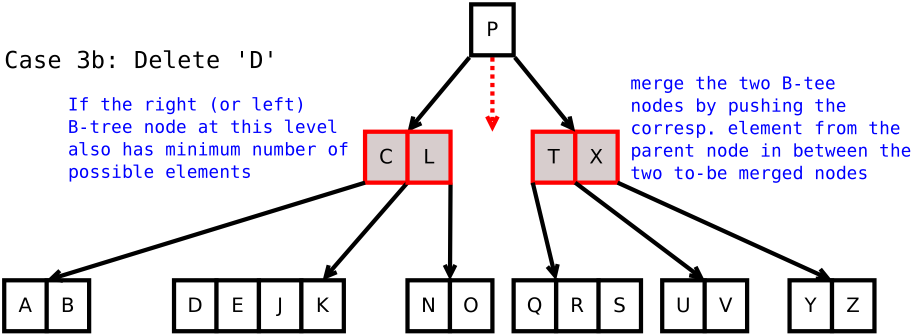
Delete 'D' – Case 3b example* (continued)

3. If the **traversal** is stopped because the 'appropriate' subtree containing **x** has a node with **exactly** $t-1$ elements, then:
   - (b) ..., if its **immediate sibling** also has **exactly** $t-1$ elements, merge the parent element with both sibling elements to form a single node. This **may** reduce the height of the tree.



*we will first see case 3b before case 3a, stay tuned!

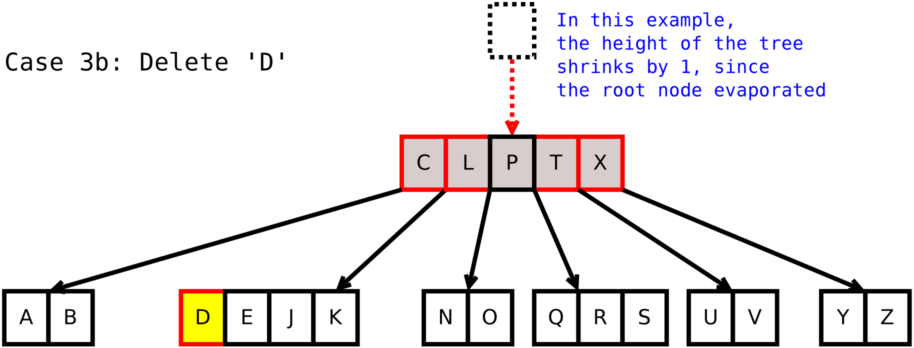
Delete 'D' – Case 3b example* (continued)



*we will first see case 3b before case 3a, stay tuned!

Delete ‘D’ – Case 3b example* (continued)

Merge parent element with siblings.

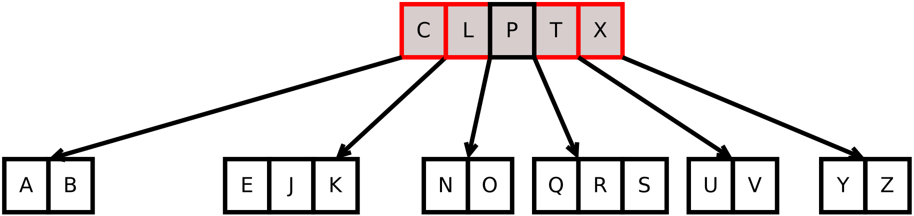


*we will first see case 3b before case 3a, stay tuned!

Delete 'D' – Case 3b example* (continued)

Recursively delete 'D'

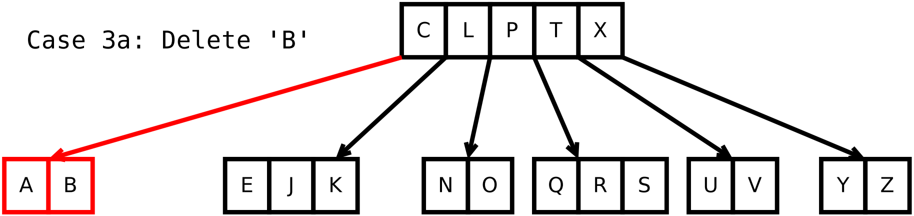
```
Case 3b: Delete 'D'
```



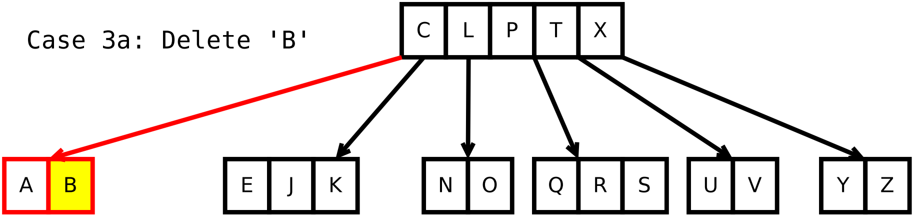
*we will first see case 3b before case 3a, stay tuned!

Delete ‘B’ – Case 3a example

3. If the **traversal** is stopped because the ‘appropriate’ subtree containing **x** has a node with **exactly** $t - 1$ elements, then:
   a. if this subtree’s ‘**immediate sibling**’ has **at least** t elements, give an extra element to the appropriate subtree by **rotating** the predecessor or successor within the sibling node to its parent, followed by moving the original parent element to the appropriate subtree.

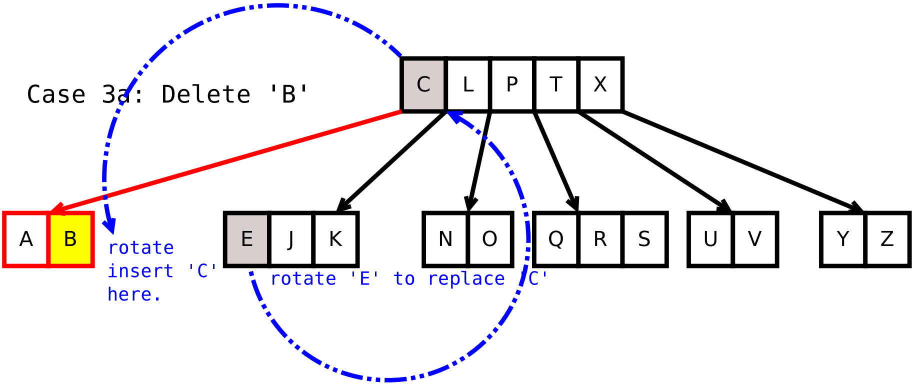


Delete ‘B’ – Case 3a example (continued)



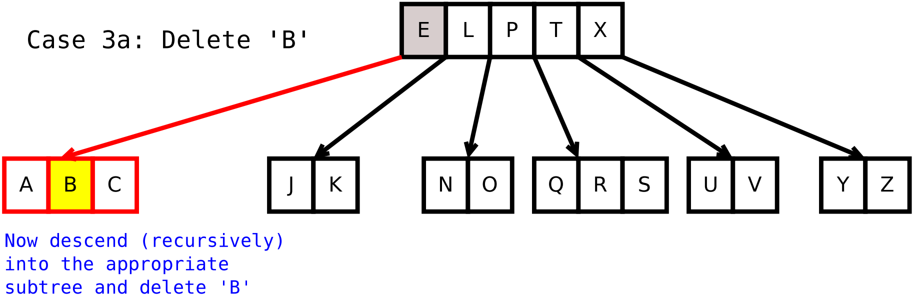

```
Cannot descend into a minimal node with exactly t-1 elements
in it. In this case, 'Immediate sibling' node (to the right
['E','J', 'K']) has at least t elements in it. Rotate the
immediate successor ('E') of the parent element 'C' into the
parent node, followed by pushing the parent element 'C' into
the appropriate ['A','B'] subtree, so that it now has at least
t elements. By doing this, the algorithm can descend into
this node/subtree.
```

Delete 'B' – Case 3a example (continued)

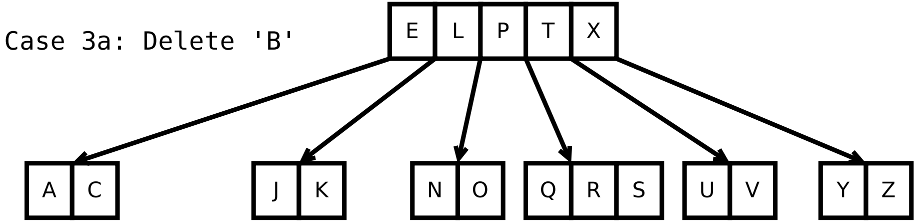


*Self-study question: How you would handle the case when [E, J, K] (and other currently leaf-level nodes) were all internal nodes of a B-tree and had their own children? Specifically, what would happen to the left-tree (before rotation) of 'E' in such a scenario? Hint: Take inspiration from the rotations you learnt for AVL trees in FIT2004!

Delete 'B' – Case 3a example (continued)



Delete ‘B’ – Case 3a example (continued)



B- tree – Space and Time complexities

	Average case	Worst case
Time Complexity of operations		
Search	$O(\log(n))$	$O(\log(n))$
Insert	$O(\log(n))$	$O(\log(n))$
Delete	$O(\log(n))$	$O(\log(n))$
Space Complexity		
	$O(n)$	$O(n)$

B-Tree Summary

- B-trees are generalizations of balanced search trees
- Designed to optimize access to secondary storage (eg. hard disks).
- All leaf nodes are at the same height, so insertion and deletion $O(\log(n))$-time, all cases

coming up...

Week 8: (Semi-)numerical algorithms – Primality testing etc.
Week 9: Lossless compression algorithms – Lempel-Ziv etc.

-=o0o=-
END
-=o0o=-