Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Advanced Data Structures

B-Trees

Dr. Amin Allam

[For more details, refer to “Data Structures and Algorithms in C++” by *Adam Drozdek*]

1 2-3 trees

Instead of trying to balance a *binary search tree* by incorporating the *red-black* properties, we can relax the binary requirement by allowing some *nodes* to have 3 children, which allows to keep all *leaves* in the same level, which is a more straightforward property that leads to balancing the *tree*.

A *2-3 tree* has the following properties:
- Each *internal node* has 2 or 3 *children*.
- Each *node* has 1 or 2 *keys*.
- All *leaves* are on the same level.
- The number of *keys* in each *internal node* $=$ the number of its *children* -1.
- The *keys* in each *node* are in ascending order. The *keys* in the first i *children* of an *internal node* are smaller than its i^{th} *key*, while the *keys* in remaining *children* are larger.

- To insert a *new key*: The *new key* is added to the suitable *leaf* in the correct place to keep all *keys* in this *leaf* sorted. Now if the *leaf* contains 3 *keys*, it is split into two *leaves*: one contains the smallest *key*, the other contains the largest *key*. The middle *key* is moved to *parent*. Now if *parent* contains 3 *keys*, this step is repeated. If there is no *parent*, a *new root* is created.

- To delete a *key* from a *leaf*: After deleting the *key*, if the *leaf* is empty and it has a *sibling leaf* having 2 *key*, one of these 2 *keys* is moved into *parent*, and the in-between *key* from *parent* is moved to the empty *leaf*. If it has a *sibling leaf* having 1 *key*, they are merged along with the in-between *key* from *parent* into one *leaf*. If *parent* becomes empty, this step is repeated unless it is the *root* it is just deleted.

- To delete a *key* from an *internal node*: the *key* to be deleted is replaced by its immediate predecessor or successor, which can only exist in a *leaf*, then the previous procedure is performed.

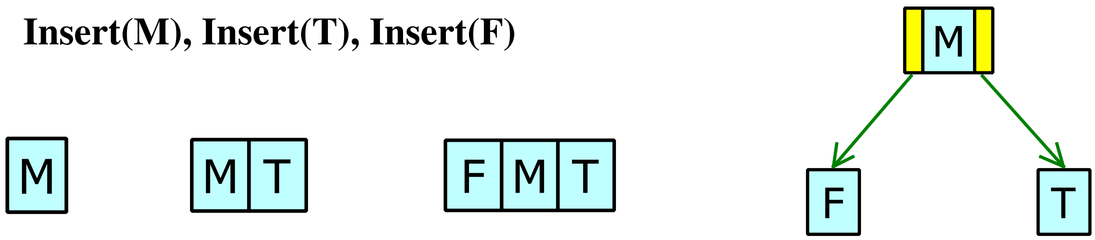
Since the number of levels changes only by creating a *new root* or deleting the existing *root*, all *leaves* always remain at the same level. Also, all other *2-3 tree* properties are maintained.

To analyse the complexity of searching a *2-3 tree*, we look into the maximum possible *number of levels* h of a *2-3 tree* in terms of the number of existing *keys* n. The maximum h is achieved when the number of *keys* per *node* is minimum (which is 1 *key* per *node*). The number of *keys* in the first level (*root*) is ≥ 1. Since the *root* must have at least 2 *children*, the number of *keys* in the second level is ≥ 2. Since each of these 2 *nodes* must have at least 2 *children*, the number of *keys* in the third level is ≥ 4. Similar reasoning leads to conclude that the number of *keys* $n \geq 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = 2^h - 1$. Hence, $h \leq log_2(n+1)$.

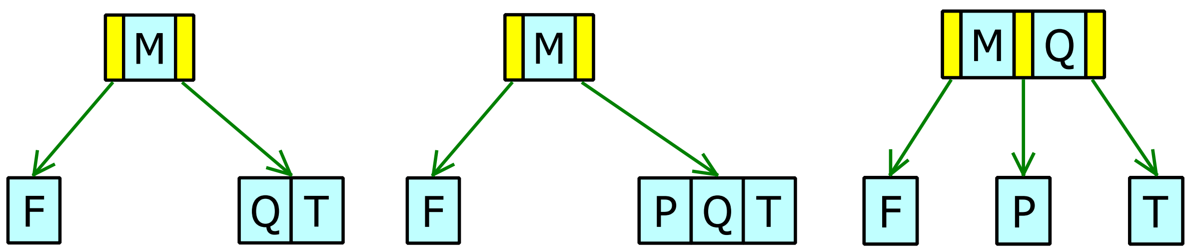
2 2-3 tree example:

Starting from an initial empty *2-3 tree*, draw all intermediate trees and draw the tree after each of the following operations:
Insert(M), Insert(T), Insert(F), Insert(Q), Insert(P), Delete(F), Delete(Q), Delete(T).

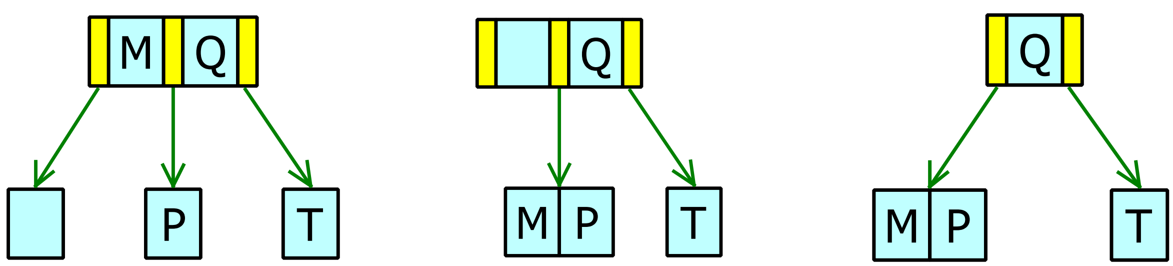
2.1 Insert(M), Insert(T), Insert(F)



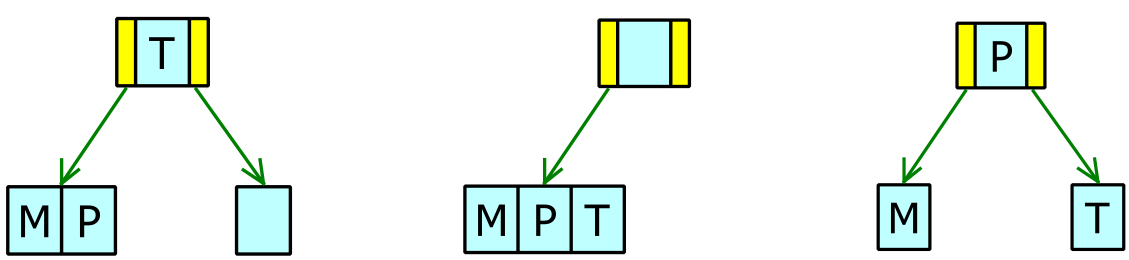
2.2 Insert(Q), Insert(P)



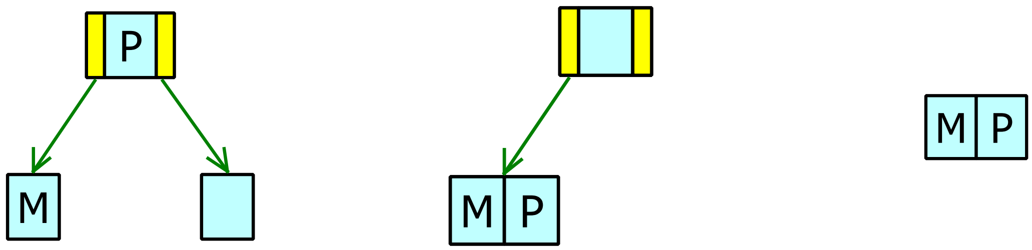
2.3 Delete(F)



2.4 Delete(Q)



2.5 Delete(T)



3 B-trees

B-trees can be viewed as a generalization of *2-3 trees*. That is, a *2-3 tree* is a *B-tree* of order 3.
A *B-tree* of order *m* has the following properties:

- The *root* has between 1 and $m-1$ keys.
- Each *internal node* (except *root*) has between $\lceil \frac{m}{2} \rceil$ and m *children*.
- Each *node* (except *root*) has between $\lceil \frac{m}{2} \rceil - 1$ and $m-1$ *keys*.
- All *leaves* are on the same level.
- The number of *keys* in each *internal node* $=$ the number of its *children* -1.
- The *keys* in each *node* are in ascending order. The *keys* in the first i *children* of an *internal node* are smaller than its i^{th} *key*, while the *keys* in remaining *children* are larger.

- To insert a *new key*: The *new key* is added to the suitable *leaf* in the correct place to keep all *keys* in this *leaf* sorted. Now if the *leaf* contains m *keys*: its middle *key* is moved to the *parent*, and the remaining keys are split into two *leaves*. Now if *parent* contains m *keys*, this step is repeated. If there is no *parent*, a *new root* is created.

- To delete a *key* from a *leaf*: After deleting the *key*, if the *leaf* contains $\lceil \frac{m}{2} \rceil - 2$ *keys* and it has a *sibling leaf* having $\geq \lceil \frac{m}{2} \rceil$ *keys*, they are merged along with the in-between *key* from *parent*, their middle *key* is moved to *parent*, and the remaining *keys* are distributed to the two *leaves*. If it has a *sibling leaf* having only $\lceil \frac{m}{2} \rceil - 1$ *keys*, they are merged along with the in-between *key* from *parent* into one *leaf*. Now if *parent* has $\lceil \frac{m}{2} \rceil - 2$ *keys*, this step is repeated unless it is the *root*.

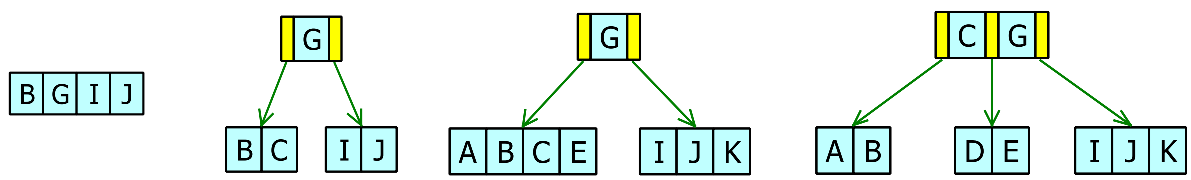
- To delete a *key* from an *internal node*: the *key* to be deleted is replaced by its immediate predecessor or successor, which can only exist in a *leaf*, then the previous procedure is performed.

A similar analysis to the *2-3 tree* leads to conclude that the *number of levels* h in *B-tree* of order m is approximately $log_{\lceil \frac{m}{2} \rceil}(n)$ where n is the number of *keys*. Therefore, h decreases with the increase of m. This makes a *B-tree* with large enough m to be very suitable to be stored on *secondary devices (hard drives)*, where the overhead of accessing a new *node* is much more than the overhead of accessing other *keys* in the same *node*. That is because *nodes* are generally not *contiguous* on the drive, hence a costly *seek* is required to move from one *node* to another. In contrast, *keys* in the same *node* are stored *contiguously* on the drive, hence little overhead is required to access all *keys* in the same *node*. Moreover, *B-trees* are often more efficient than *red-black trees* even if they are stored in *main memory* since they significantly reduce *cache misses*, but they require more storage than *red-black trees* since most *B-tree nodes* contain unused spaces.

4 B-tree example:

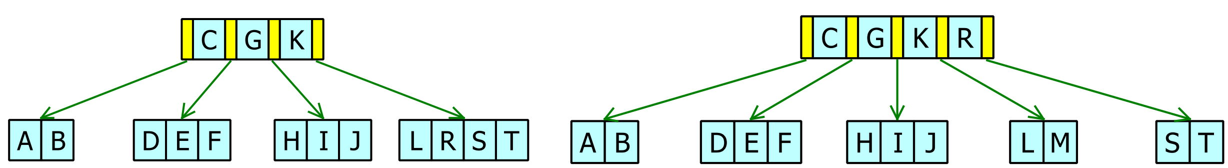
Starting from an initial empty *B-tree* of order 5, draw all intermediate trees and draw the tree after each of the following operations:
Insert(G), Insert(I), Insert(B), Insert(J), Insert(C), Insert(A), Insert(K), Insert(E), Insert(D), Insert(S), Insert(T), Insert(R), Insert(L), Insert(F), Insert(H), Insert(M), Insert(N), Insert(P), Insert(Q), Delete(E), Delete(F), Delete(G), Delete(K)

4.1 Insert(G), Insert(I), Insert(B), Insert(J) - Insert(C) - Insert(A), Insert(K), Insert(E) - Insert(D)

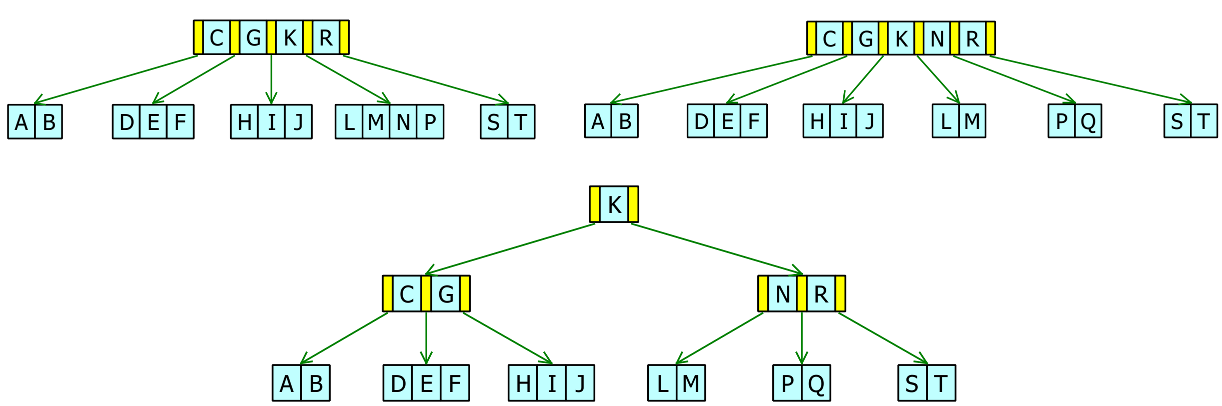


4.2 Insert(S) - Insert(T)

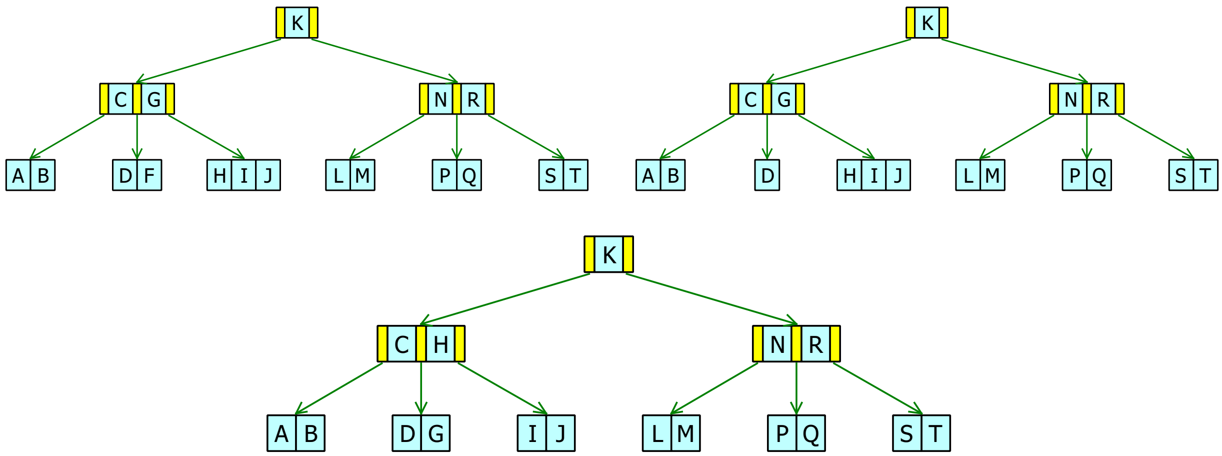
4.3 Insert(R), Insert(L), Insert(F), Insert(H) - Insert(M)



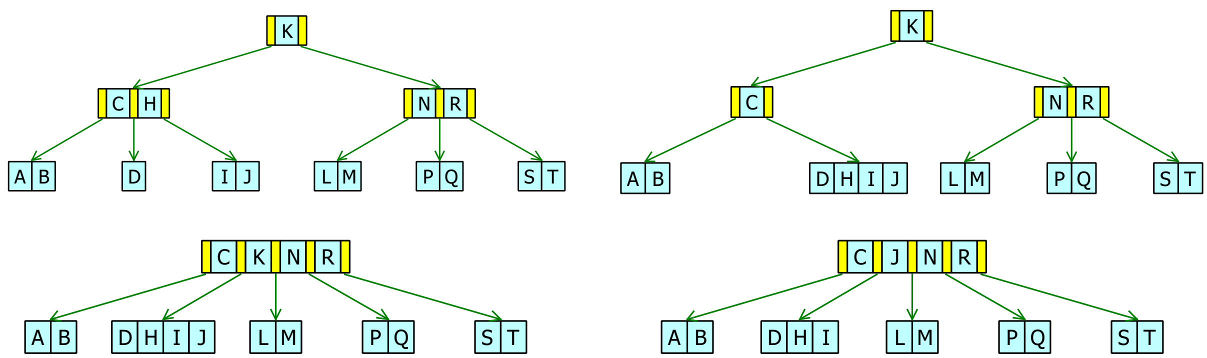
4.4 Insert(N), Insert(P) - Insert(Q)



4.5 Delete(E) - Delete(F)

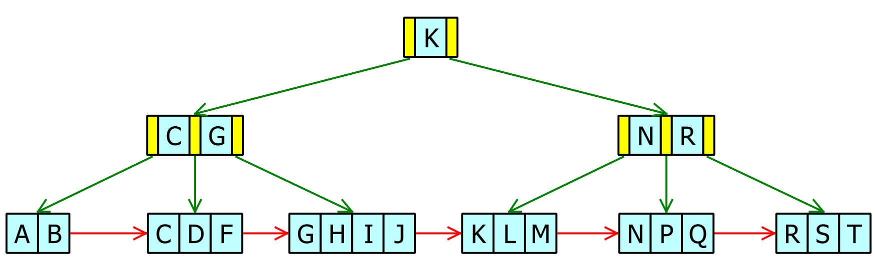


4.6 Delete(G) - Delete(K)



5 B$^+$-trees

A *B$^+$-tree* is a *B-tree* where all *keys* exist in *leaves*. *Keys* in intermediate *nodes* are used only as separators and for directing search queries. Additional links exist from each *leaf* to its right *sibling*. The purpose of such augmentations is to facilitate range queries, especially for *secondary storage*. Once a search query reaches a *leaf*, all subsequent records can be accessed without accessing *nodes* at higher levels.



A *static* version of *B$^+$-tree* is called *multi-level indexing*. If data is *static* and no updates are required, data can be initially sorted once to construct the deepest *B$^+$-tree* level, then higher levels are constructed statically. All links are substituted by formulas as functions of a fixed *node* size.