B-Trees

Nov 28, 2022

- Another "Dictionary ADT" solution.
- one favoured in situations where...

dictionary is so large it cannot be read into memory in its entirety

- in this case, we seek to minimize the number of disk transfers - ie moving in a page of data from the disk.
- a page is usually pretty big - want to ensure the data is useful.

When we navigate a tree, we identify the next node to go to and read it in...

... a node should be page sized ie quite large ... how can we make nodes usefully large (useful for navigating the tree from root to leaf)?

Definition: A B-tree T is a rooted tree (root is $T \to root$) where

1. $\forall$ node x

   $x \to n$ = number of keys in node x

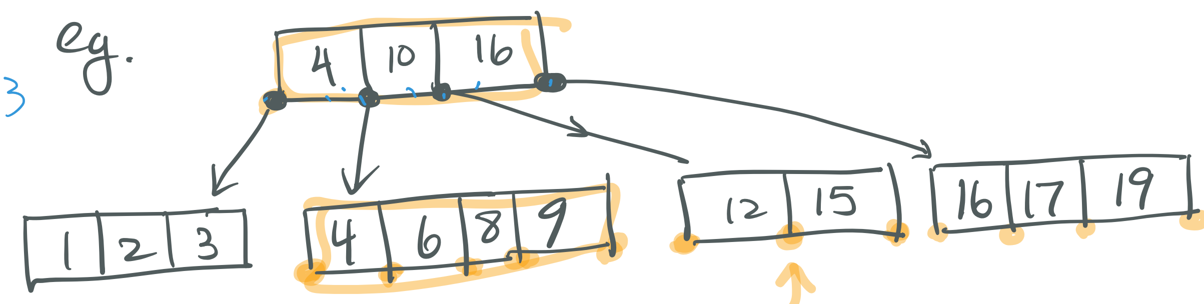
   $x \to key$ = an array $[1 .. x \to n]$ of keys

   $x \to leaf$ = true if x is a leaf (has no children

2. if $!x \to leaf$, then $x \to c[i]$ is a pointer to x's i^{th} child $(0 \le i \le x \to n)$

3. The keys of a node separate the ranges of the keys of the children

$t = 3$ eg.



4. All leaves have same depth, which is height h.

5. $\exists$ upper and lower bounds on number of keys that a node can contain

- lower bound $t-1$ keys (t children)
- lower bound does not apply to root
- root has 0 keys $\Leftrightarrow$ tree is empty.
- upper bound $= 2t-1$ keys, ($2t$ children)

  A node is <u>full</u> if it has exactly $2t-1$ keys.

Eg $t=2 \Rightarrow$ nodes have 2, 3, or 4 children

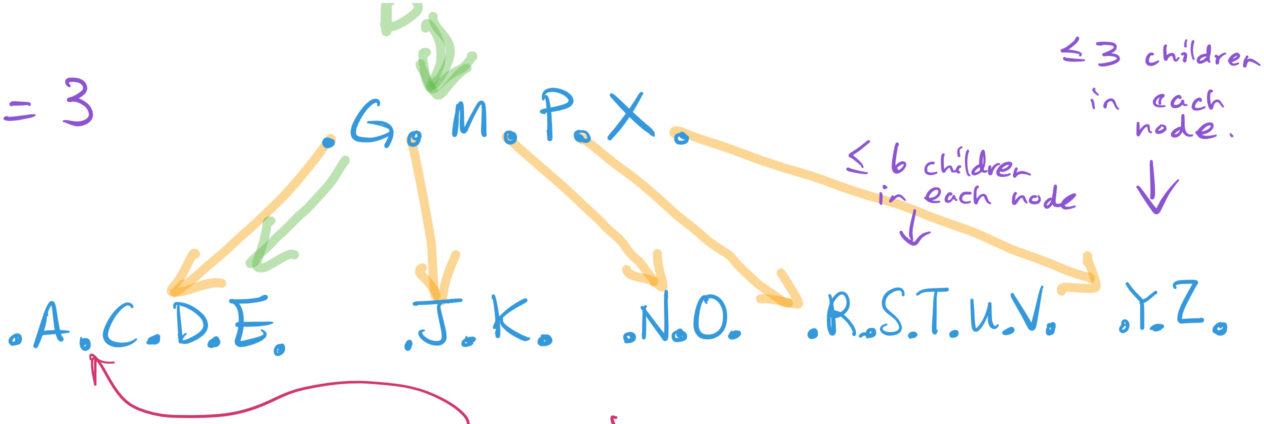
- also known as a 2-3-4 tree.

We made nodes as large as possible and still fit into one <u>page</u> of memory.

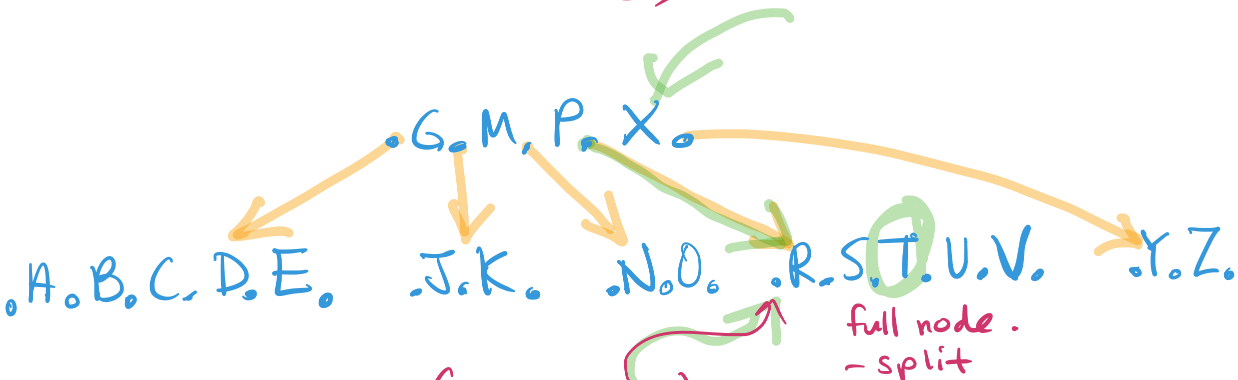
This reduces height of tree

Each time we step down a level in the tree, we do a disk swap. We always go down to a leaf (in B-trees, data is stored at leaf-level)

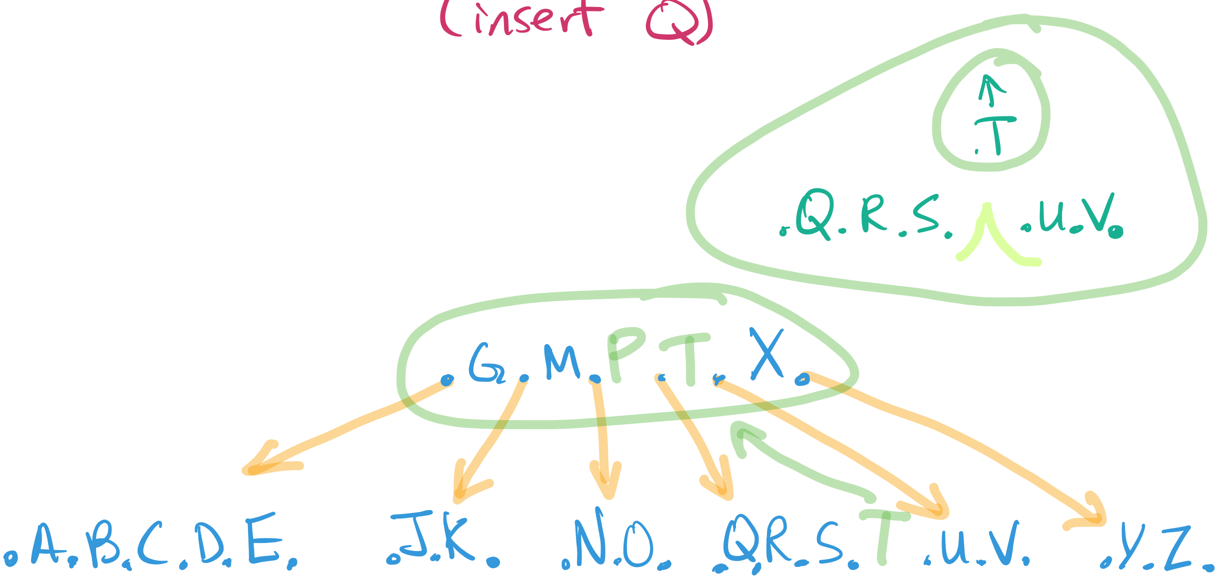
$t = 3$



(insert B)

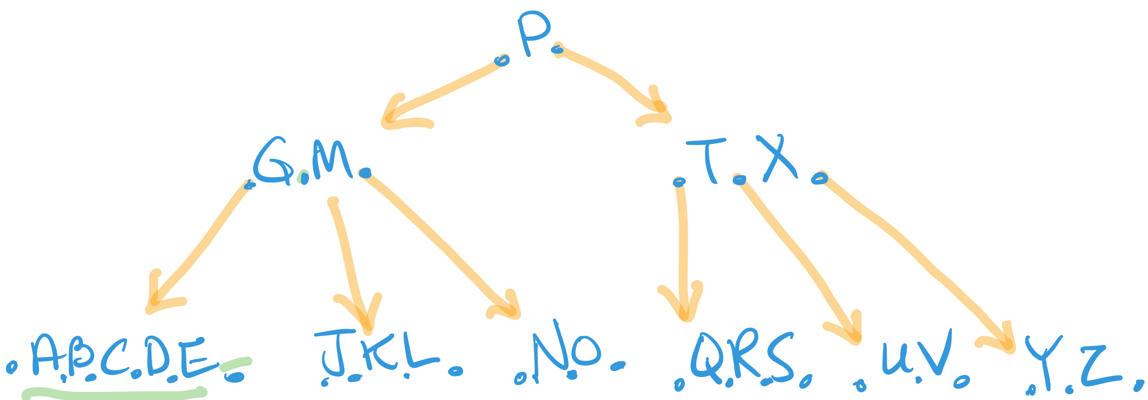


(insert Q)

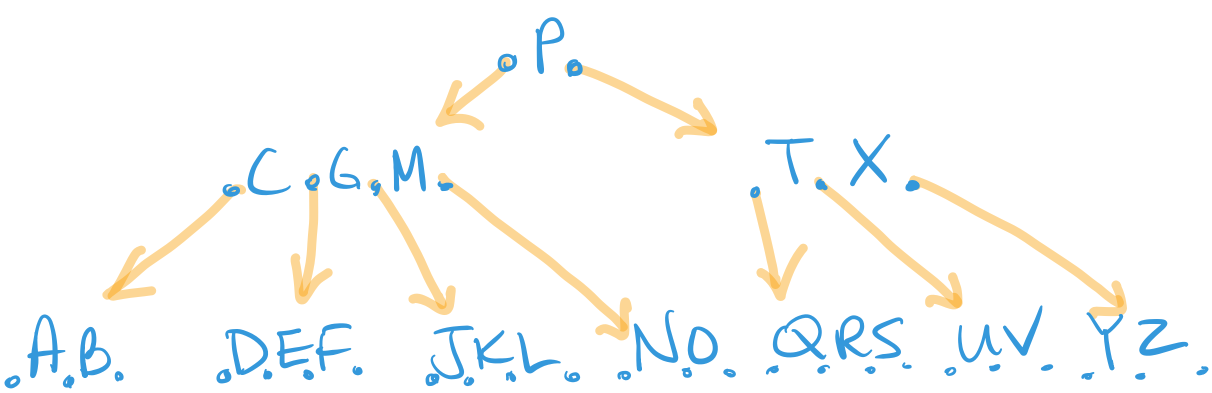


(insert L)

.G.M.T.P.X is full, so split.



(insert F)



Notes on insertion into B-trees:

- root can have fewer than t children. Other non-leaf nodes must have t to $2t$ children
- If root is <u>full</u> and an insertion is executed, the root node is <u>split</u>. Other nodes have a different behaviour.

- a non-leaf node x that is given a key to insert checks if the appropriate child is full - if so, the child is split, the centre key is brought into the current node x, and insertion proceeds to the child node.

Pseudo code for split, insert_Not_Full, and Insert are given below.

```
BTreeSplitChild ( x, i, y)
/* y is x's ith child. Result will be
/* that children i+1 ..x->n are shifted left
/* in x's list of children, and each of
/* x->c[i] and x->c[i+1] will have room
/* for an insertion

node* newc = new node
newc->leaf = y->leaf
newc->n = t-1
for j=1 to t-1
      newc->key[j] = y->key[j+t]
if !y->leaf
      for j=1 to t
            newc->c[j] = y->c[j+t]
y->n = t-1
for j = x->n downto i
      x->key[j+1] = x->key[j]
x->key[i] = y->key[t]
x->n++
DiskWrite(y); DiskWrite(newc); DiskWrite(x).
```

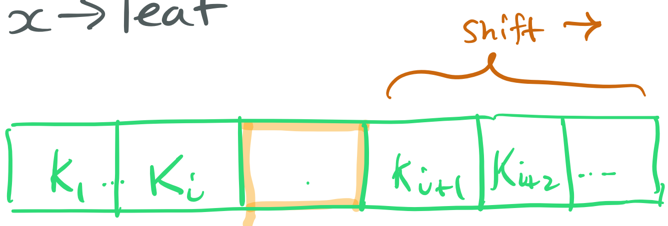
```
BTreeInsert(T, k)
/* Insert key k, starting at T->root */

r = T->root
if r->n = 2t-1
    node* s = new node
    T->root = s
    s->leaf = false
    s->n = 0
    s->c[1] = r
    BTreeSplitChild(s, 1, r)
    BTreeInsertNonFull(s, k)

else
    BTreeInsertNonFull(r, k)
```

BTreeInsertNonFull(x, k)

/* pseudocode with diagrams for insertion when

/* node is not full */

if x→leaf



1. Shift the back end of x's arrays of children to make room for <u>K</u> to be inserted into its spot in sorted order
2. write K into its spot
3. x→n++

else // x not a leaf... has children, too

1. Find the child i of x into which K should be inserted – DiskRead this child node
2. if x→c[i] is full

   BTreeSplitChild(x, i, x→c[i])

   if K > x→key[i]

   i++

   BTreeInsertNonFull(x→c[i], k)