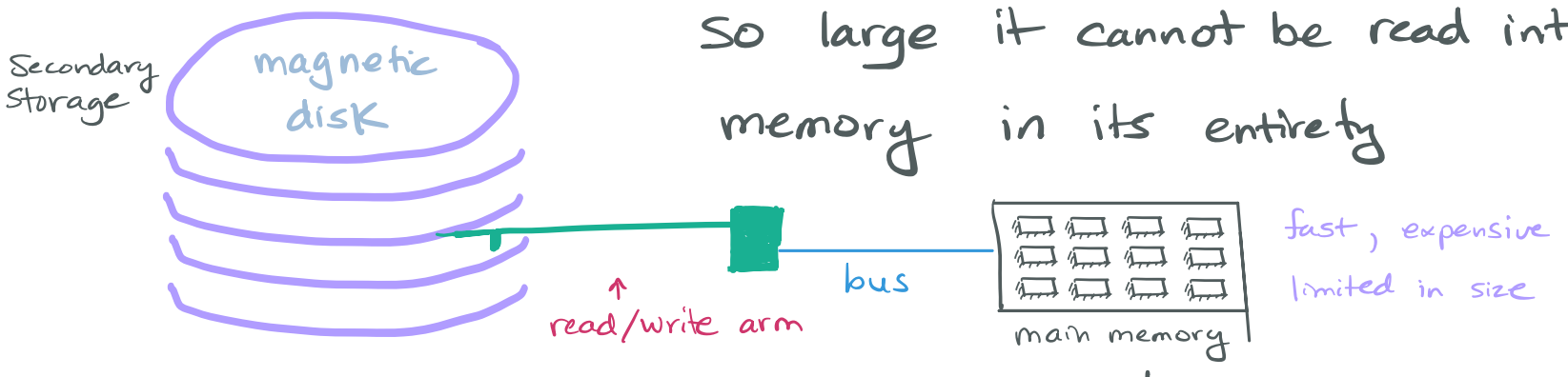
# B-Trees

- Another "Dictionary ADT" solution.
- one favoured in situations where the dictionary is so large it cannot be read into memory in its entirety



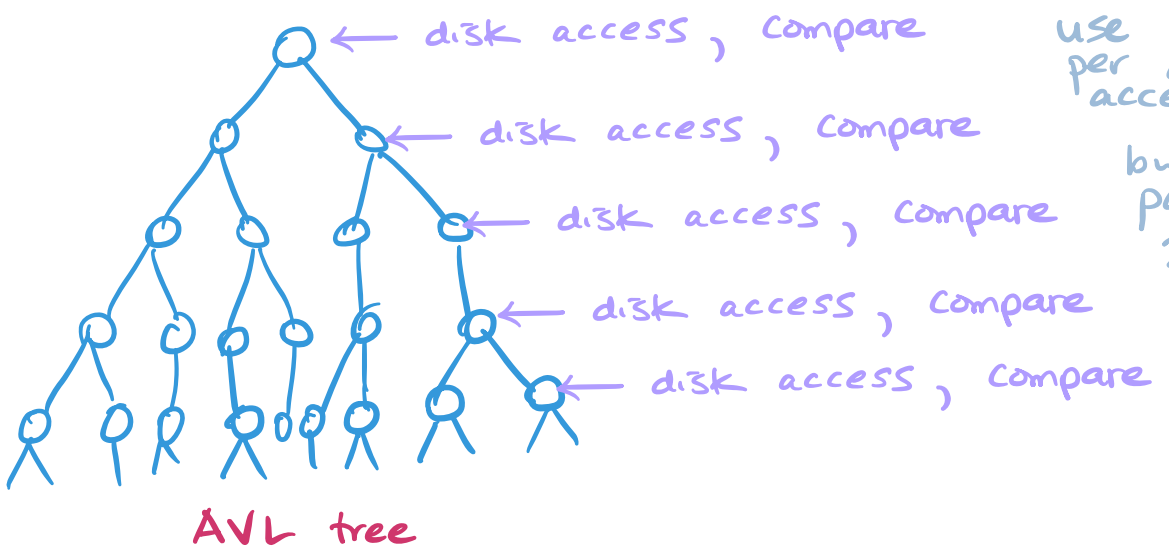

In this case, we seek to minimize the number of disk transfers - ie, swapping in a page of data from the disk.

A page is usually pretty big. We want to ensure the data is useful. Access is slow - maybe 20 ms

When we navigate a tree, we identify the next node to go to and read it in. A node should be page-sized (i.e. quite large).

How can we make nodes usefully large (useful for navigating the tree from root to leaf) ?

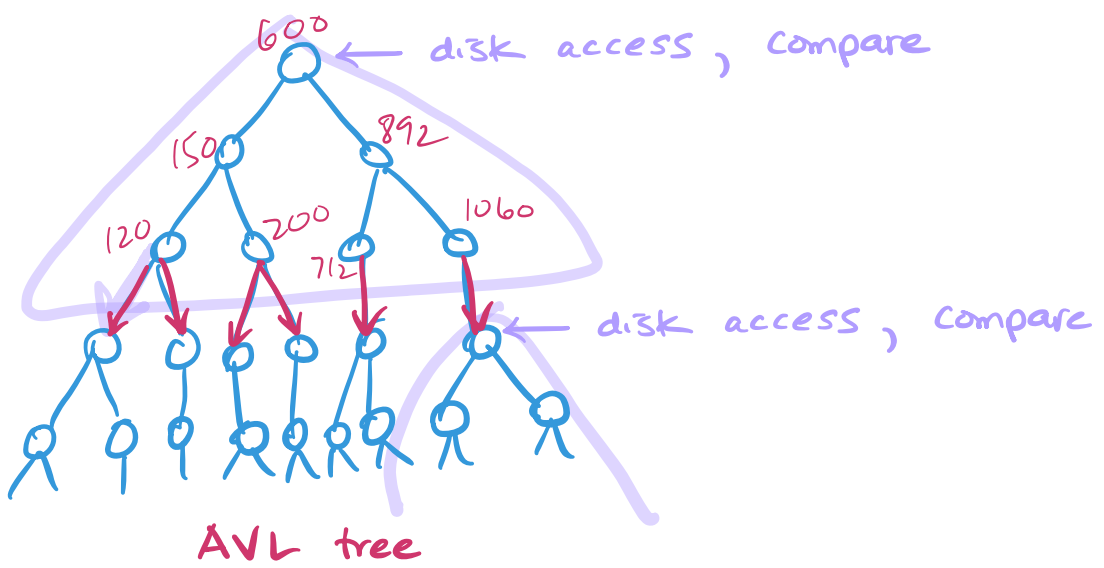
# Evolving the rationale for B-trees:



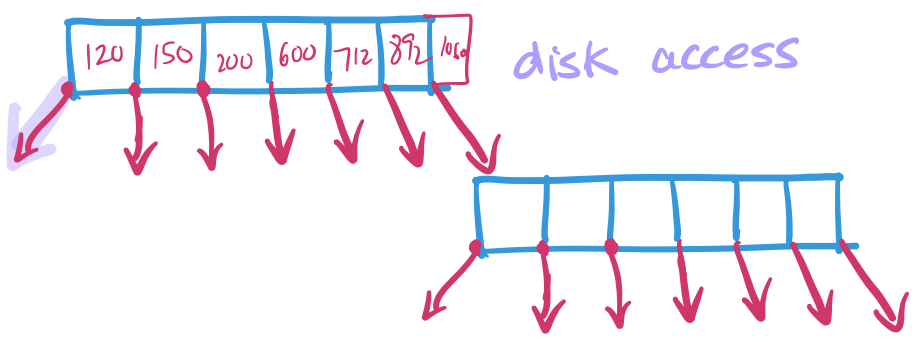

use 8 bytes per disk access

but a page has 2048 bytes

AVL tree




AVL tree



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 $x$

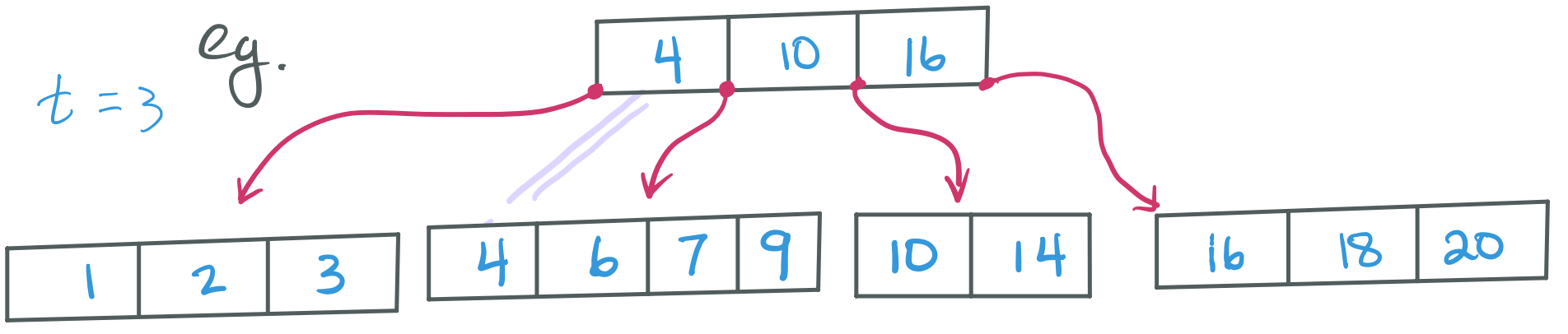
   $x \to key$ = an array $[1 .. x \to n]$ of keys

   $x \to leaf$ = 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. ∃ upper and lower bounds on number of keys a node can contain:

- lower bound $t-1$ keys in node ($t$ children)
  - lower bound does not apply to root
    root has 0 keys ↔ tree is empty
- upper bound = $2t-1$ keys ($2t$ children)
  - a node is full if it has exactly $2t-1$ keys

Eg $t=2$ → nodes have 2, 3, or 4 children
- also known as a 2-3-4 tree.

We choose node size to be as large as possible and still fit onto one page 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)

$t = 3$

.G.M.P.X.

≤ 3 children in each node.

≤ 6 children in each node

.A.C.D.E. .J.K. .N.O. .R.S.T.U.V. .Y.Z.

(insert B)

.G.M.P.X.

.A.B.C.D.E. .J.K. .N.O. .R.S.T.U.V. .Y.Z.

full node. - split

(insert Q)

↑T

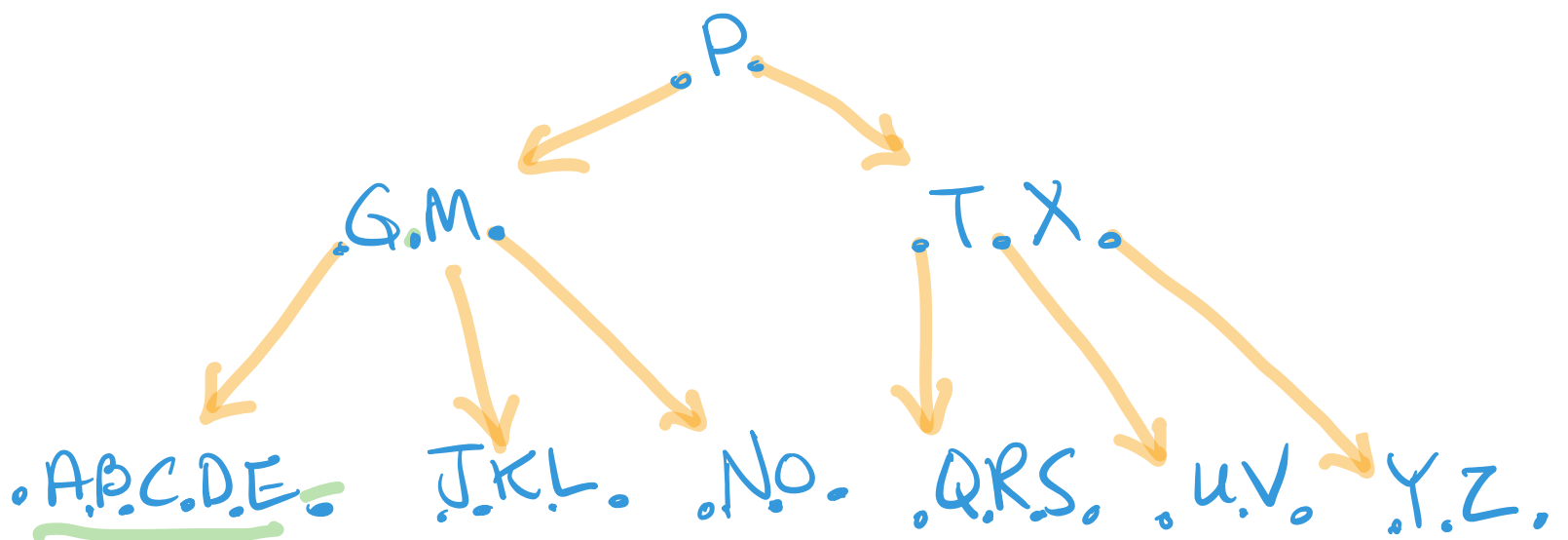
.Q.R.S. .U.V.

.G.M.P.T.X.

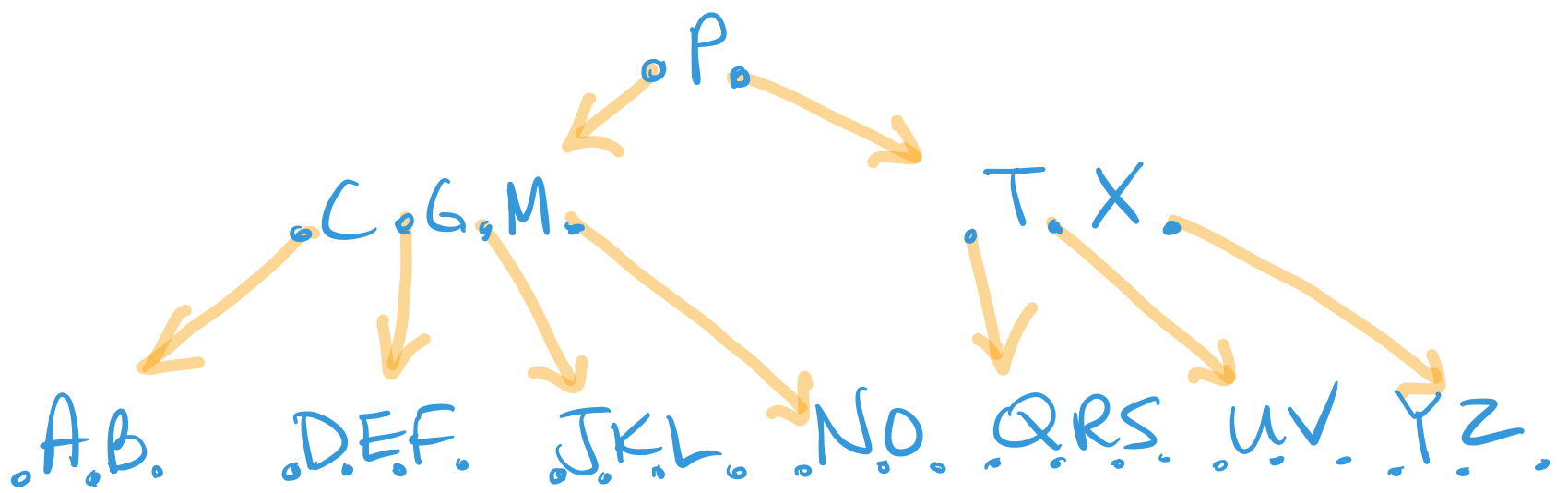
.A.B.C.D.E. .J.K. .N.O. .Q.R.S. .U.V. .Y.Z.

(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 full and an insertion is executed, the root node is split. 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->c[i+1] = newc
x->key[i] = y->key[t]
x->n++
DiskWrite(y); DiskWrite(newc); DiskWrite(x).
```

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

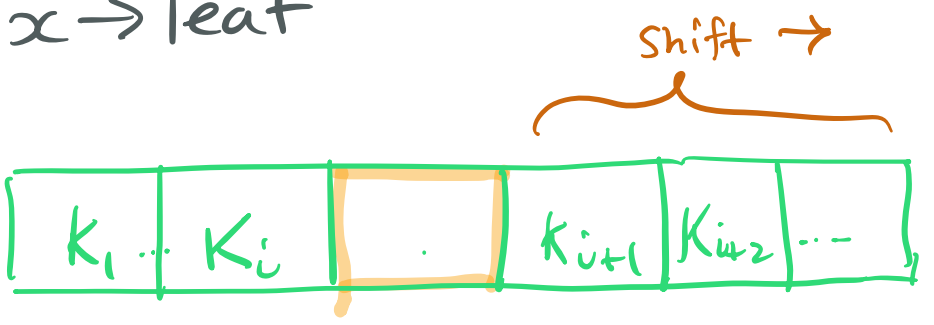
r = T->root
if r->n = 2t-1                          // pro-active node-
                                        // splitting - so
    node* s = new node                  // you don't have to
    T->root = s                         // recurse up the
    s->leaf = false                     // tree and are
    s->n = 0                            // guaranteed to
    s->c[1] = r                         // have room.
    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 k 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)