≡ B-Tree ≡ Basics, Insert

Think of "B" as "branching", not binary (unofficial)
Each node has <u>k</u> children ("k-ary tree")
Goal: Each node completely occupies one sector/unit/page on a hard drive: MINIMIZE # OF DISK ACESSES
↳ eval running time based on 2 things: compute time
#disk accesses

<u>Def: B-Tree</u> (see picture on page #4)
∃ a root node - all nodes are attached (directly or indirectly) to it
Each node has: (#) •
• int cKeys - count of keys in this node → AND data.
• int [] rgKeys - array of keys, sorted into (ascending) order ← length = 2t-1
• boolean fLeaf - true if this is a leaf node, false otherwise
Nonleaf nodes have (cKeys+1) references to child nodes - BTreeNode[] rgChildren
Leaf nodes have no children - rgChildren isn't defined
EVERY LEAF HAS THE SAME DEPTH (= height of tree, "h")
Each node has a minimum degree t (all nodes have same degree)
• Every node (EXCEPT the root) must have at least t-1 keys
• " " can have NO MORE than 2t-1 keys
(node w/ 2t-1 keys is FULL & has 2t children)
Note: height of B-Tree is $\log_t(N)$

(# side-discussion - struct vs. references (all in one node))

DiskPages read = $O(h) = O(\log_t N)$
ComputeTime = $O(t \cdot \log_t(N))$

```
≡BTreeSearch ( BTreeNode x, int k)   // target
{ for(int i = 0; i < x.cKeys && k > x.rgKeys[i]; i++)
        ;//empty on purpose - all work done in ↑
    if( i < x.cKeys && k == x.rgKeys[i])
        return x.rgKeys[i].data; //awk. syntax :)
    if( x.fLeaf) return null;
    else return BTreeSearch( DISK-READ( x.rgChildren[i]), k)
}
```

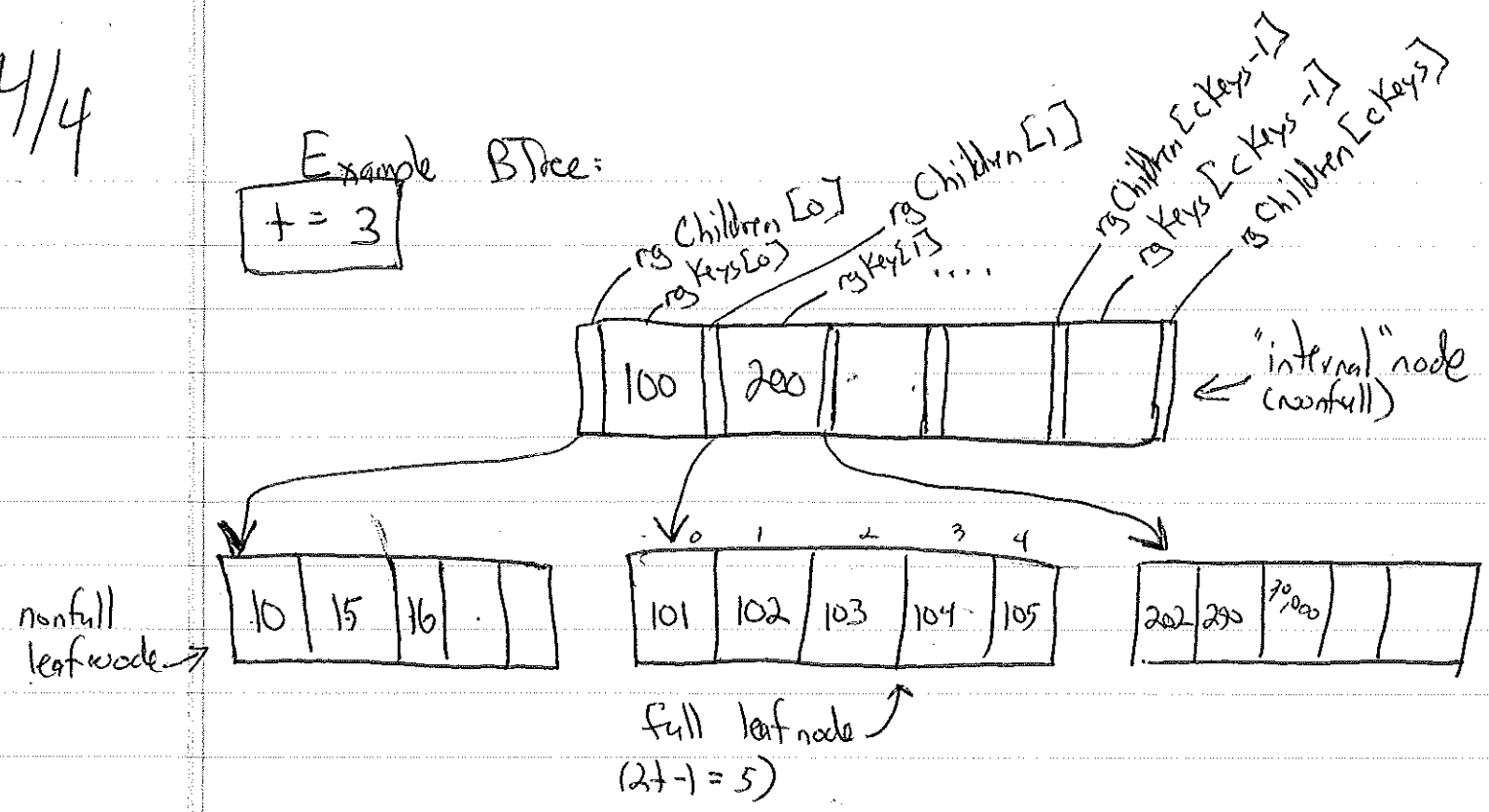
O(t)

```
BTreeCreate()
{   X = new BTreeNode();
    // X.fLeaf = true;        in constructor
    // X.cKeys = 0;           // also "alloc" arrays
    DISK-WRITE(x);
    root = x;
}


// int t; global variable t contains degree
BTreeSplitChild(BTreeNode parent, int iChild, BTreeNode child)
// PRE: parent is a NONFULL (internal) node
//      child is a FULL child of parent (may or may not be a leaf)
//      iChild index (in parent) of iChild
// POST: child has been split into 2 nodes, median value stored in parent.
  BTreeNode next = new BTreeNode()
    next.fLeaf = child.fLeaf; next.cKeys = t-1;
Copy top 1/2 into new node
    for(int i=0; i<t-1; i++) next.rgKeys[i] = child.rgKeys[i+t];   (t-1 not addition)
    if(!child.fLeaf) // internal, non leaf node - copy child ref.s
      for(int i=0; i<(t-1); i++) next.rgChildren[i] = child.rgChildren[i+(t-1)];
    child.cKeys = t-1;
    for(int i=parent.cKeys; i > iChild; i--) // move everything up one...
    {   parent.rgKeys[i] = parent.rgKeys[i-1];
        parent.rgChildren[i+1] = parent.rgChildren[i];
    }

    parent.rgChildren[iChild+1] = next; // add next to tree
    parent.rgKeys[iChild] = child.rgKeys[t-1];
    parent.cKeys++;
    DISK-WRITE(parent); DISK-WRITE(child); DISK-WRITE(next);
```

```
BTree Insert(BTree tree, int key)
{ //PRE: t is a valid B-Tree ; key is the value to insert
                                (may be an object w/ data)
  //POST: t is still a valid BTree; key has been inserted
  //This is a 'one pass' implementation - if we notice a child is
  //full, we will preemptive split it, BEFORE traversing it
    BTreeNode root = tree.root;
    if( root.cKeys == 2t-1) // need to split root
    {  BTreeNode newRoot = new BTreeNode();
       newRoot.fLeaf = false;
       newRoot.cKeys = 0;
       newRoot.rgChildren[0] = root;
       BTreeSplitChild (newRoot, 0, root);
       tree.root = newRoot;
                                  root = newRoot;
    }
    BTreeInsertNonfull(root, k);
}
BTreeInsertNonfull(BTreeNode node, int key)
{   if(node.fLeaf == true)
    {   for(int i = node.cKeys; i>=1 && key < node.rgKeys[i-1]; i--)
            node.rgKeys[i] = node.rgKeys[i-1];
        node.rgKeys[i] = key;
        node.cKeys++;
        DISK-WRITE(node);
    }
    else // internal node - figure out which child to add the key to
        for (int i = node.cKeys-1; i>=0 && key < node.rgKeys[i]; i--)
            ; /*for loop empty */   i++; //move i up 1 after loop
BTreeNode child = DISK-READ(node.rgChildren[i]);
        if ( child.cKeys == 2*t-1)
            BTreeSplitChild(node, i, child)
                if( key > node.rgKeys[i]) i++;
BTreeInsertNonfull(child, key);
```

4/4

Example BTree:



full leafnode
$(2t-1 = 5)$

1

B-Trees - Find, Insert

B- Trees - designed to minimize # times you have to access the disk → d.b.s
internal nodes & ~~external~~ leaf nodes
internal - if it has N keys, it'll have N+1 children
↳ relate this to binary trees

2 Primary vs. Secondary
↳ cheaper, much slower; persistent → physically moving parts
→ fast, expensive; volatile
relative to each other

1 HD diagram -
platters on the same spindle
↳ covered w/ magnetic material
Arm extends over spindle, head reads a track
↳ all arms are linked
3 Cylinder - set of tracks - each track divided into a bunch of pages, each containing a bunch of info
Disk-Access time isn't constant -

Data is too large to put it all into memory at once - copy bits & pieces as needed

Alg. mostly bounded by # disk accesses
Min. # disk accesses - make a node as large as a page
B^+-Tree - store all info (except keys) in leaf nodes - maximizes branching, to minimize # disk accesses

2

Properties - Every node X has - fields -

- N - # keys stored @ X
- keys[] - stored in nondecreasing (0...N) order
- leaf - true if it's a leaf

Internal Nodes -

X.N +1 pointers X.C[1] ... X.C[X.N+1] to children

The keys separate the ranges of keys stored in each subtree

$key_{i-1} \le k_i \le key_i$

- All leaves have exactly the same depth
  ↳ This is h, the height of the tree

- Upper & lower bounds on # keys a node can contain
  bound is $t \ge 2$ min: $\ge t-1$ max: $2t-1$ fullness $2t-1$ nodes

2-3-4 trees?

- Every node (except root) must have at least $t-1$
  $\Longrightarrow$ every node (except root) has at least t children
- Every node can contain at most $2t-1$ keys
  A node w/ $2t-1$ keys is full

Assume the root stays in-mem - will only need to be written

<u>Find</u> - much like a B.S.T., except that you gotta go through the array of keys till you find the one that's $\le$, and either that's it, or else go to child to it's immediate left.

⟨Demo this briefly⟩

$\Theta(h) = \Theta(\log_t(n))$

point at spanning factor

<u>Create a B-Tree</u> - get the disk page
Mark it as a leaf, w/ 0 entries
Write it to disk
Point root to it.

3

B-Tree Insert:

Basic idea: Find a leaf node to insert the value into.

Point out that you can store data @ nodes, as well.

B^+ tree use this to maximize fan-out.

If that node is full (already has $2t-1$ elts), split that node into 2 separate leaves of $t-1$ each, with the middle-most being moved up into the parent.

Likewise with the parent, etc

Once that's done, you can put the new elt. into one node or the other & be sure there's enough space.

Single-Pass Optimization: On your way down the tree, if it's full, split it pre-emptively.

index at which to insert new median

B-Tree-Split-Child(node, i, child)

```
z = AllocateNode(),  // this will be the right 1/2
copy child's leaf attr to z
z.numElts = t-1
Copy keys t through 2t-1 into z
(if it's not a leaf, copy the child ptrs, too)
child.numElts = t-1
Shuffle the key/ptr. pairs in node up by one, to make room @ i
Copy child.key[t] into node.key[i]
node.numElts ++
Write node, child, z to disk
```

This needs a special case for a full root.

4

```
B-Tree-Insert ( Tree, key/data)
    if root is full
    {
        S = Create a new disk page
        S. leaf = false;
        Tree. root = S ; // save old root in a temp
        S. numElts = 0
        S.Child[1] = r; // BTSC will copy the ptr to the right into S.Child[2]
        B-Tree SplitChild ( S, 1, oldRoot)
    }
    B-Tree-Insert-Nonfull ( Tree, root, key/data)
```

```
B-Tree-Insert-Nonfull (node, key/data)
    if (node. leaf == true)
    {       shuffle all keys up one, put new key into place),
            Write node to disk}
    }
    else // must be @ internal node - figure out where to add
    {       search highest to lowest to find it's spot
            Read the child off the disk.
            If ( child.numElts == 2t -1 )
            {       B-Tree-Split-Child(node, i, node.child[i])
                    if ( key > x. key[i]
                            i ++;
            }
            B-Tree-Insert-Nonfull (node.child[i], key/data)
```