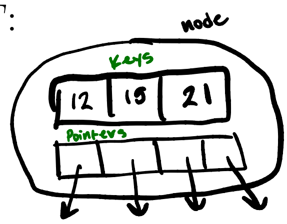
k-ary Search Trees

A k-ary search tree T is defined such that for each node x of T:



- x has at most k children (i.e., T is a k-ary tree)
- x stores an ordered list of pointers to its children and an ordered list of keys
- For every internal node: # of keys = # of children−1
- x fulfills the fulfills the **search tree property**:
  keys in subtree rooted at i^{th} child $\leq i^{\text{th}}$ key $<$ keys in subtree rooted at $(i+1)^{\text{th}}$ child

Example of a 4-ary search tree:

=> These notes are in data structures journal

B-Trees

A B-tree T with minimum degree $k \geq 2$ is defined as follows:

↳ Nodes shouldn't be empty

- T is a $2k$-ary search tree
- Every node, except the root, stores at least $k - 1$ keys
  (Thus, every internal non-root node has at least k children)
- The root must store at least one key
- All leaves are on the same level of the tree

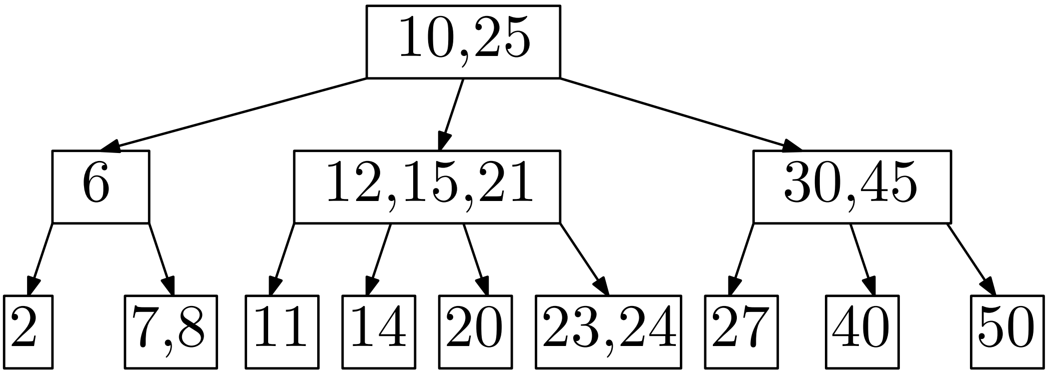
min //except root max

$k \leq$ # children $\leq 2k$

$k-1 \leq$ # keys $\leq 2k-1$

$1 \leq$ # root keys $\leq 2k-1$

Example of a B-tree with $k = 2$:

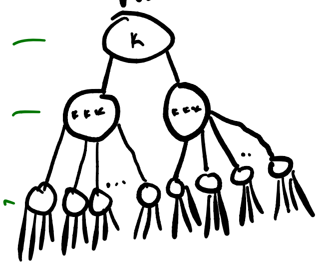


Theorem.
Given a B-tree with minimum degree $k \geq 2$ which stores n keys and has height h it is the case that $h \leq \log_k((n+1)/2)$.

Proof: what is max height?

What does a B-tree with the minimum number of keys per level look like?

root



level	# nodes	(min) # keys
0	1	1
1	2	$2\cdot(k-1)$
2	$2\cdot k$	$2k\cdot(k-1)$
3	$2k^2$	$2k^2\cdot(k-1)$
⋮		
L	$2k^{L-1}$	$2k^{L-1}(k-1)$

// given any $L > 0$, we can find # nodes via..

$$\Rightarrow 2k^{L-1}$$

If the above tree had height h, how many keys would it contain? //using info above

keys in above tree:

//change index

$$\Rightarrow 1 + \sum_{L=1}^{h} 2k^{L-1}(k-1) = 1 + \sum_{i=0}^{h-1} 2\cdot k^i (k-1)$$

level 0 key

$$= 1 + 2(k-1)\sum_{i=0}^{h-1} k^i$$

= # of total nodes in above B-Tree

Since we put our n keys in a B-tree of height h we know:

// lower bound

- geometric series

$$n \geq 1 + 2(k-1)\sum_{i=0}^{h-1} k^i = 1 + 2\cancel{(k-1)}\cdot\left(\frac{k^h-1}{\cancel{k-1}}\right)$$

$$= 1 + 2k^h - 2$$

$$= 2k^h - 1$$

$$\Rightarrow n \geq 2k^h - 1$$ //solve for h

$$\Rightarrow \frac{n+1}{2} \geq k^h$$ //$\log_k$ here ↳ $\log_k k^h = h$

$$\Rightarrow \log_k\left(\frac{n+1}{2}\right) \geq h$$

Therefore $h \in O(\log_k n)$

Why Use B-Trees? Make Tree nodes bigger

Problem: Given a large amount of data that does not fit into main memory (i.e., in RAM), process it into a dictionary data structure.

- (Thus we have to store our data on disk which is very expensive to access)

- Need to minimize number of disk accesses
- With each disk read, read a whole block of data } only some of data is useful
- Construct a balanced search tree that uses one disk block per tree node } Tree node should fill entire disk block, then all is our data
- Thus we want each node to contain more than one key (B-trees!)

Find the a key k in our B-tree using the following algorithm.

Algorithm 1 node BTreeSearch(node x, int k)

```
i = 1;
while i ≤# of keys in x and k > i^th key in x do   } search all keys in node until data location found
    i = i + 1;
end while
if i ≤# of keys in x and k = i^th key in x then
    return (x, i);   → // data found return it
end if
if x is a leaf then
    return NIL;   → // data not found
else
    b =DISK-READ(i^th child in x);
    return BTreeSearch(b, k);
end if
```

Runtime:

Worst case: $O(k \cdot \log_k n)$ // data not found

from while loop

B.S.T: $O(\log_2 n)$

# of disk accesses:

$O(\log_k n)$ | B.S.T $O(\log_2 n)$

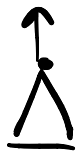
expensive want to minimize

B-Tree insert

46:58

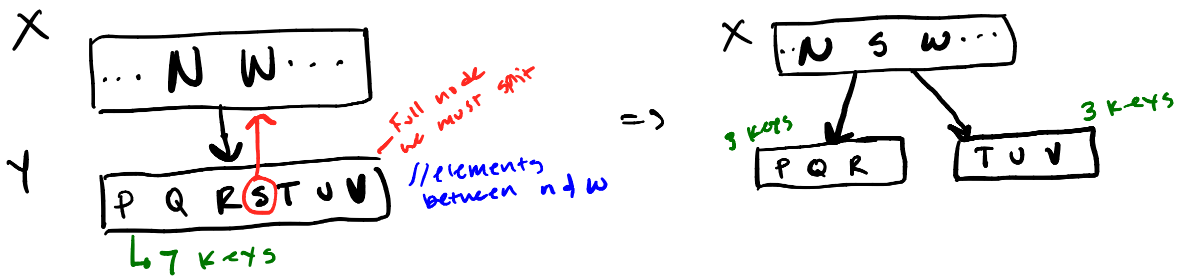
Make one pass down the tree:

- The goal is to insert the new key into a leaf
- Search where key should be inserted
- Only descend into non-full nodes: // early split
  - If a node is full, split it. Then continue descending.
  - Splitting of the root node is the only way a B-tree grows in height. ↳ grows from root



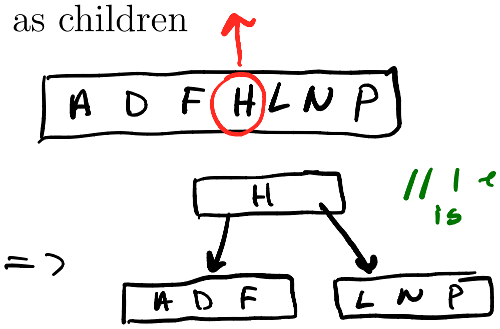
Overview of Insert Helper Functions

- BTreeSplitChild(x, i, y)
  - Split full node y into two nodes y and z of $k-1$ keys
  - Median key of y is moved to x (which is the parent y)
  - It becomes the new i^{th} key in x



=> $4 \le$ children ≤ 8] From properties of B-Tree
$3 \le$ keys ≤ 7

- BTreeSplitChild$(s, 1, r)$
  - The full root node r is split in two
  - A new root node s is created
  - s contains the median key of r and has the two halves of r as children



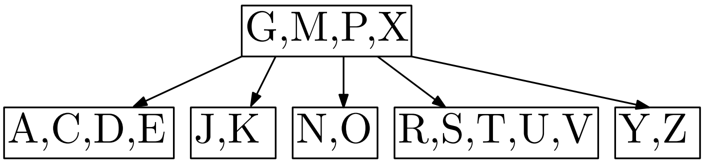
// 1 element @ root is O.K.

$K = 3$

$2 \le \text{keys} \le 5$

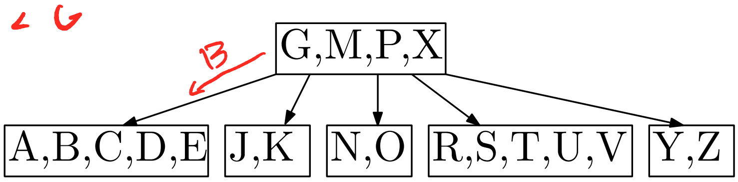
$3 \le \text{children} \le 6$

Example Insertions:



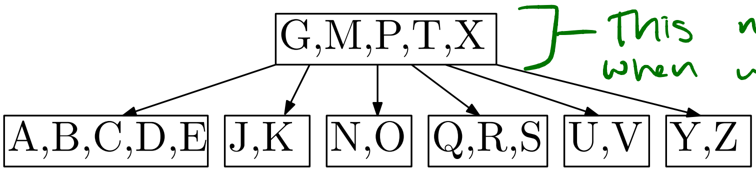
INSERT B (insert at leafs)

B < G



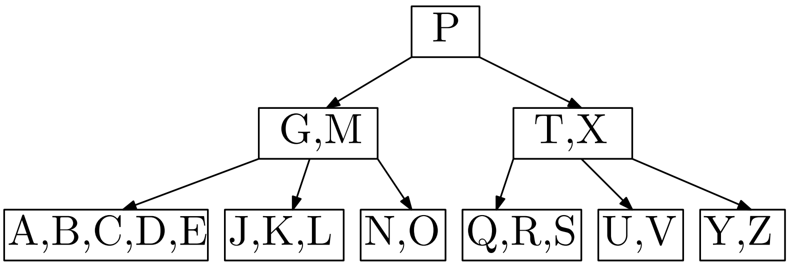
INSERT Q

// $\boxed{\text{Q R S \{T\} U V}}$

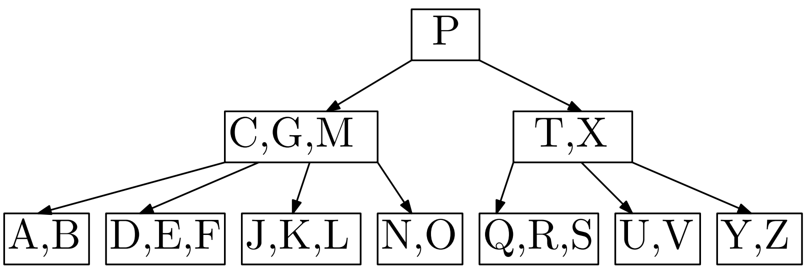


This node splits when we insert L because computer sees this node is full before inserting which leads to the split at root

INSERT L



INSERT F



Algorithm 2 void BTreeInsert(tree T, int key)

```
r = root[T];
if # of keys in r = 2k − 1 then //root r is full
    //create and insert new root node s
    s = AllocateNode();
    //split r to be two children of new root create and insert new root node s
    BTreeSplitChild(s, 1, r);
    BTreeInsertNonfull(s, key);
else
    BTreeInsertNonfull(r, key);
end if
```

Algorithm 3 void BTreeInsertNonfull(node x, int key)

```
if x is a leaf then
    insert key at correct (i.e., sorted) position in x
    DISK-WRITE(x);
else
    find child c of x which by the search tree property should contain should contain key
    DISK-READ(c);
    if c is full then //c contains 2k − 1
        //Let the i^th key in x be the largest key smaller than the keys in c
        BTreeSplitChild(s, i, r);
        c =child of x which should contain key
    end if
    BTreeInsertNonfull(c, key);
end if
```