Three Reasons for using B-trees

(1) storage-friendly:

B-trees work well on any layer of the storage hierarchy

(2) good trade-off random/sequential:

seeks/cache misses once for **each node** rather than once for each pivot step

(3) universal applicability:

You have a fancy indexing problem?

Did you try a B-tree (or a variant of it)?

Did not work? Did you try **really** hard?

Intuition for B-Tree Indexes

k	67	72	84	105
v	T	K	V	D

A Second Page...

67	72	84	105
T	K	V	D

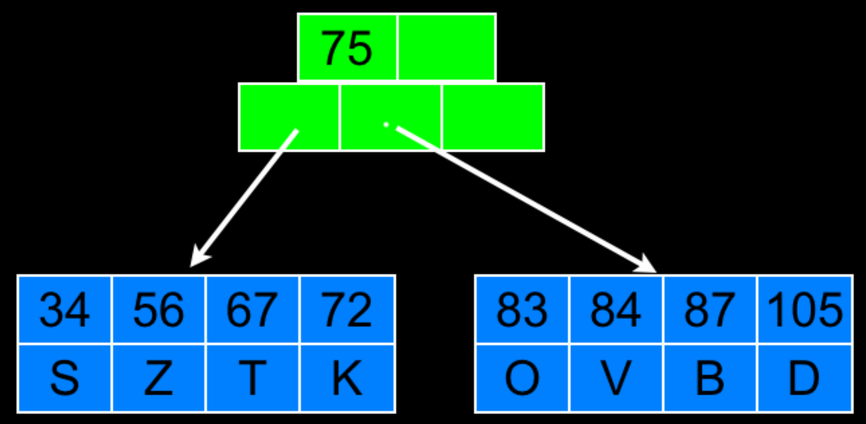
34	56	83	87
S	Z	O	B

Redistribute...

34	56	67	72
S	Z	T	K

83	84	87	105
O	V	B	D

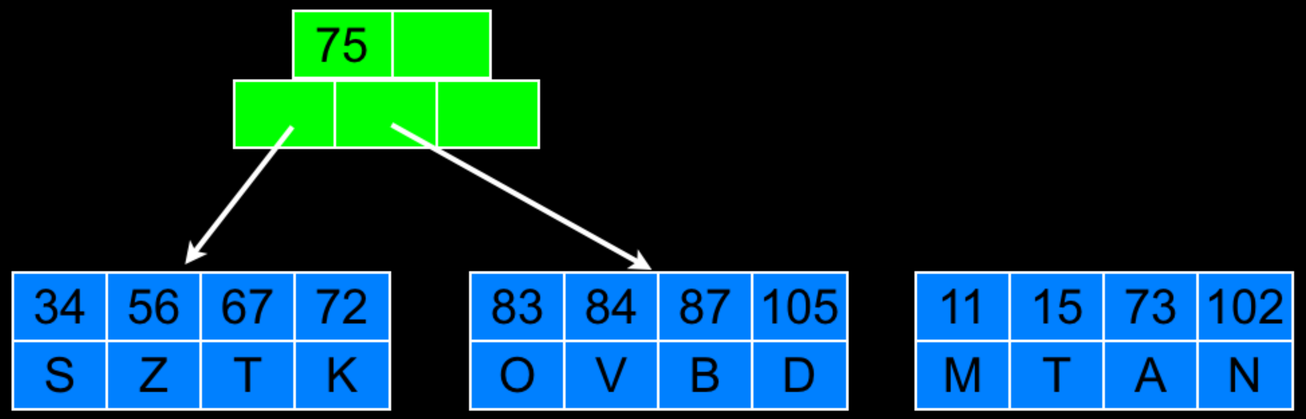
Create Descriptor...



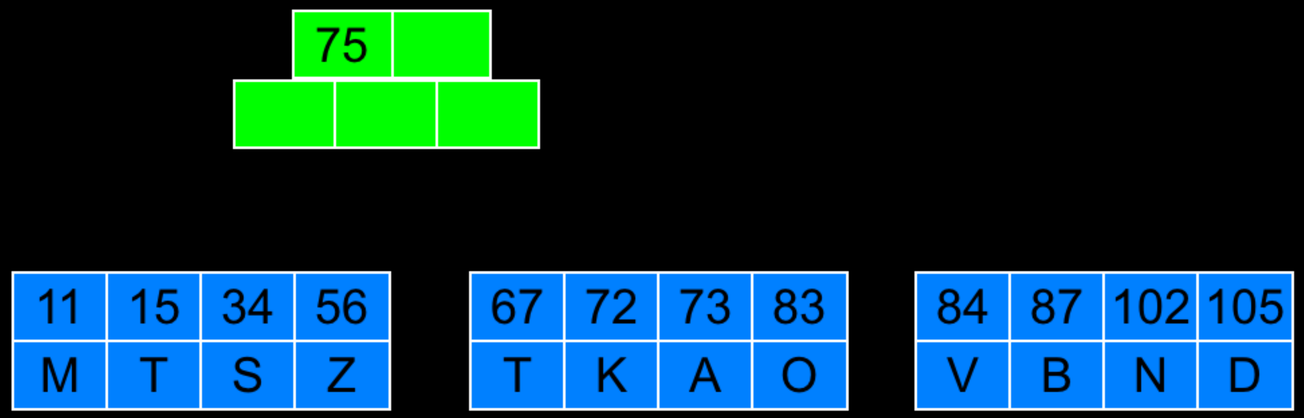
Node

Leaf

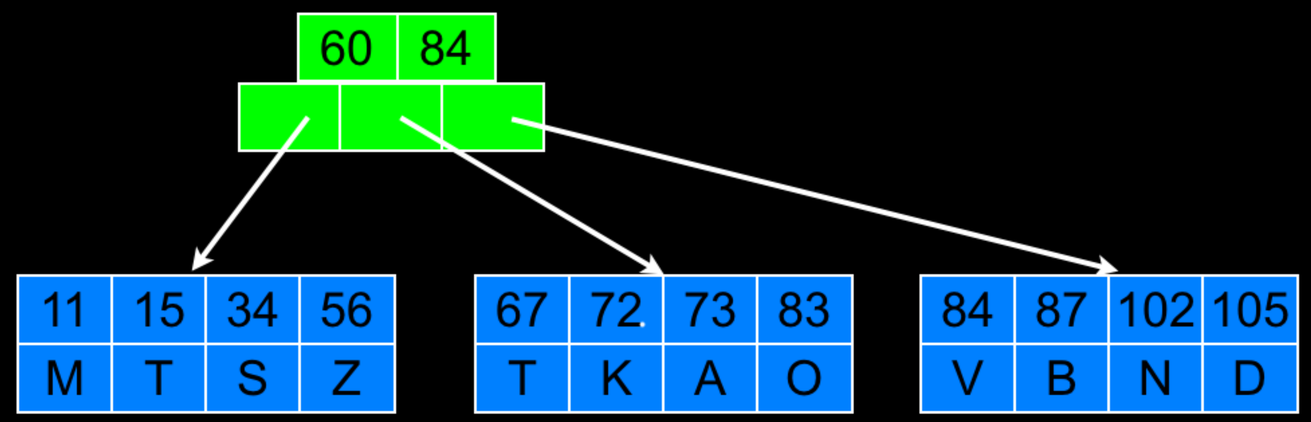
A Third Page...



Redistribute...

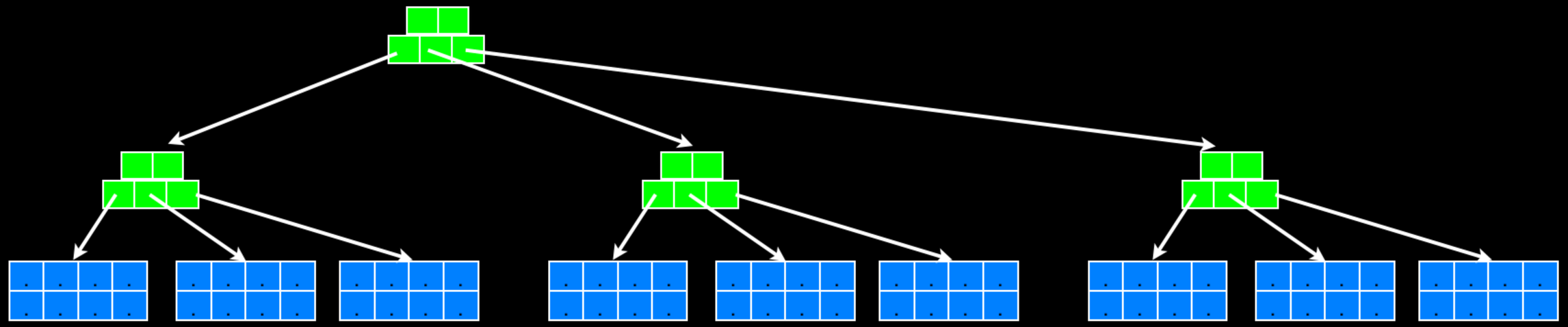


Fix Descriptors...

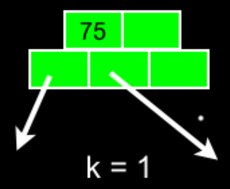
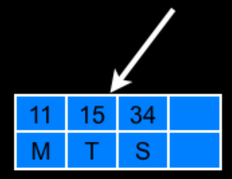


[60; 84[

Recurse...



B-Tree Node and Leaf Sizes

	if not root:	if root:
Nodes  4k13	$n \in [k;2k]$ keys => n+1 children	$n \in [1;2k]$ keys => n+1 children K=4
Leaves  [2;4]	$[k^*;2k^*]$ key/value-pairs	$[1;2k^*]$ key/value-pairs

B-Tree Properties

path from root to any leaf: same length

k and k* implicitly defined by data types and node size

no values in index nodes

all data sits in leaf nodes

B-tree

B*-tree

$\boxed{\text{B}^+\text{-tree}}$

B-Tree Properties

path from root to any leaf: same length

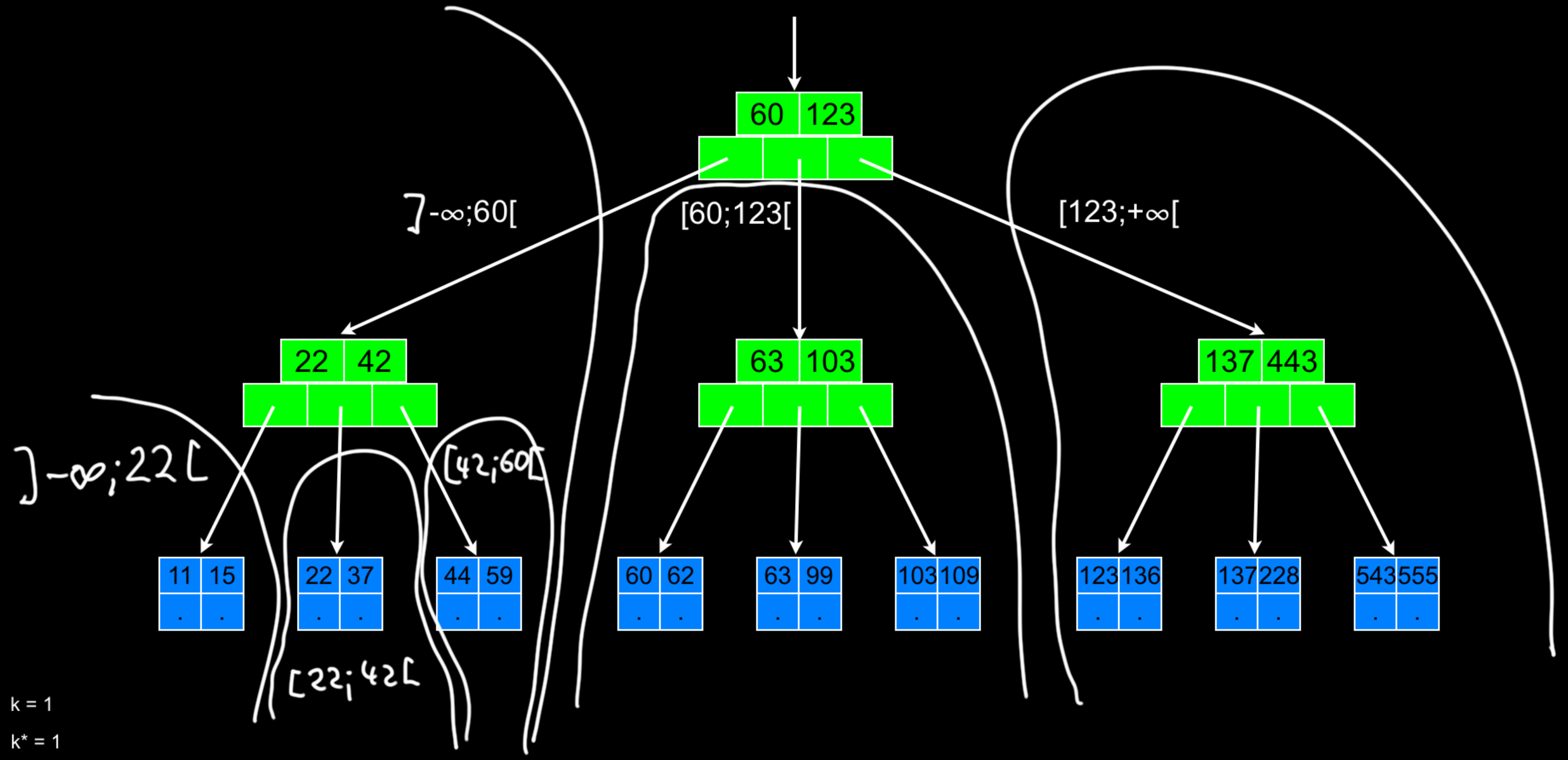
k and k* implicitly defined by data types and node size

no values in index nodes

all data sits in leaf nodes

(data inside nodes and leaves sorted by key)

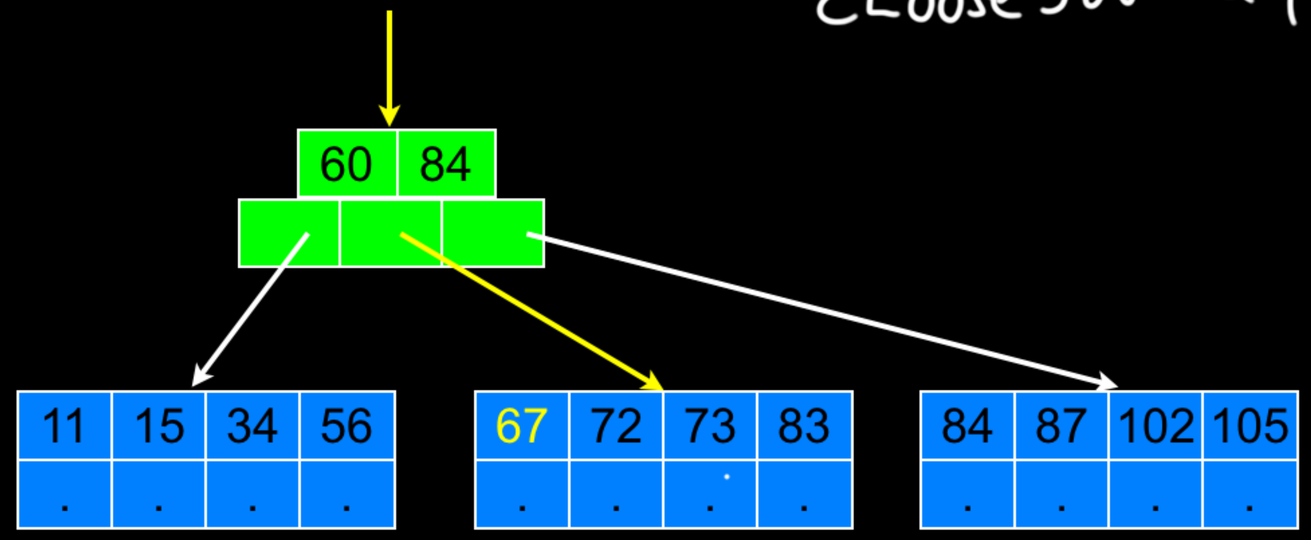
B-Trees and Interval Partitioning



k = 1

k* = 1

find_key(67)

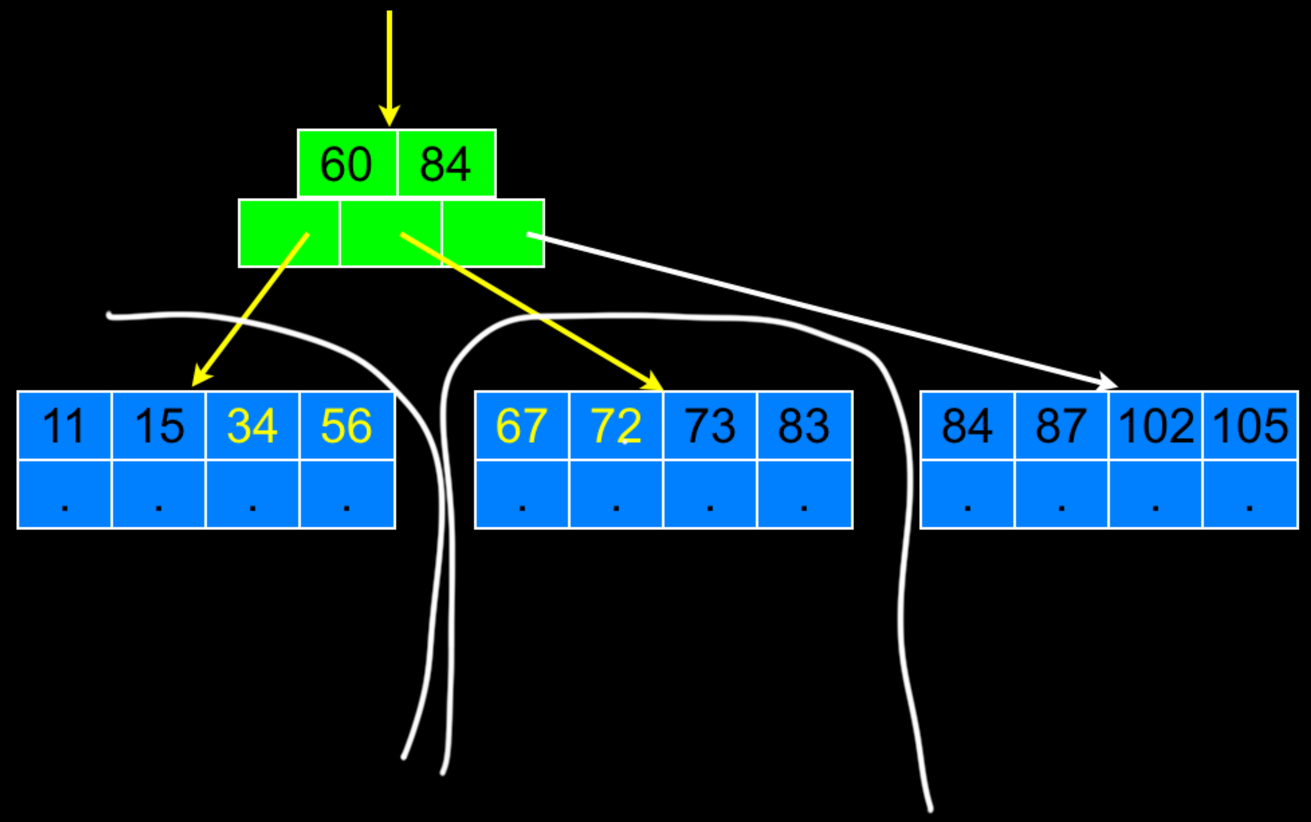


chooseSubtree(key k)

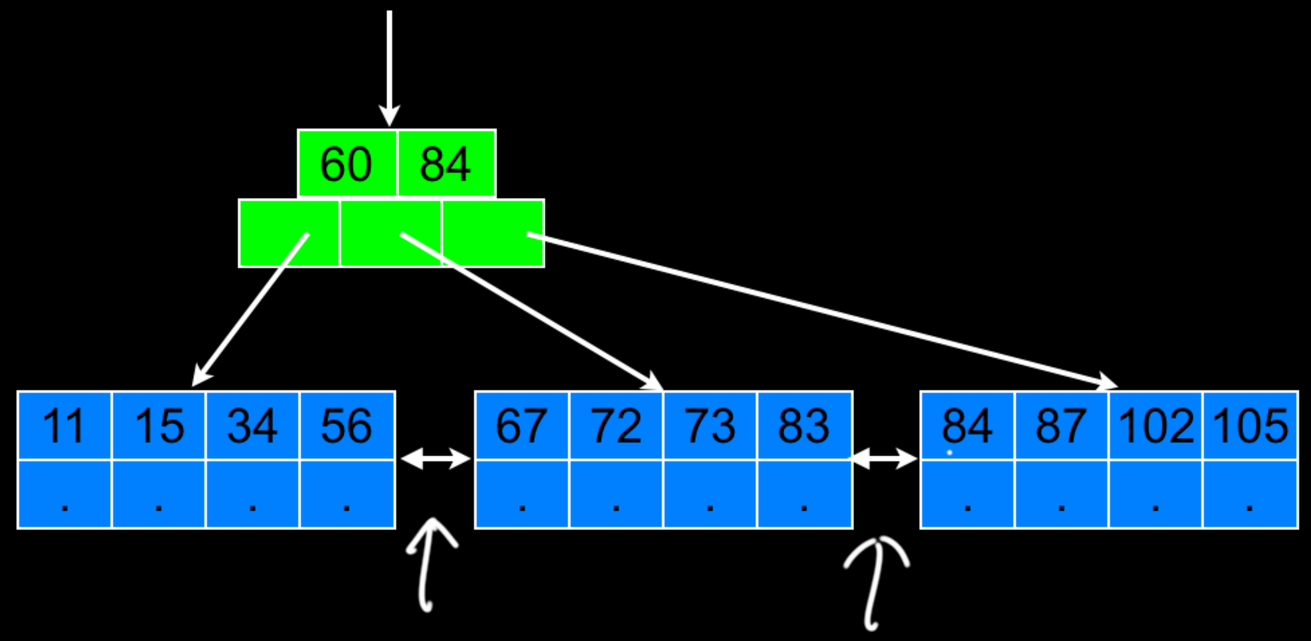
↑

67

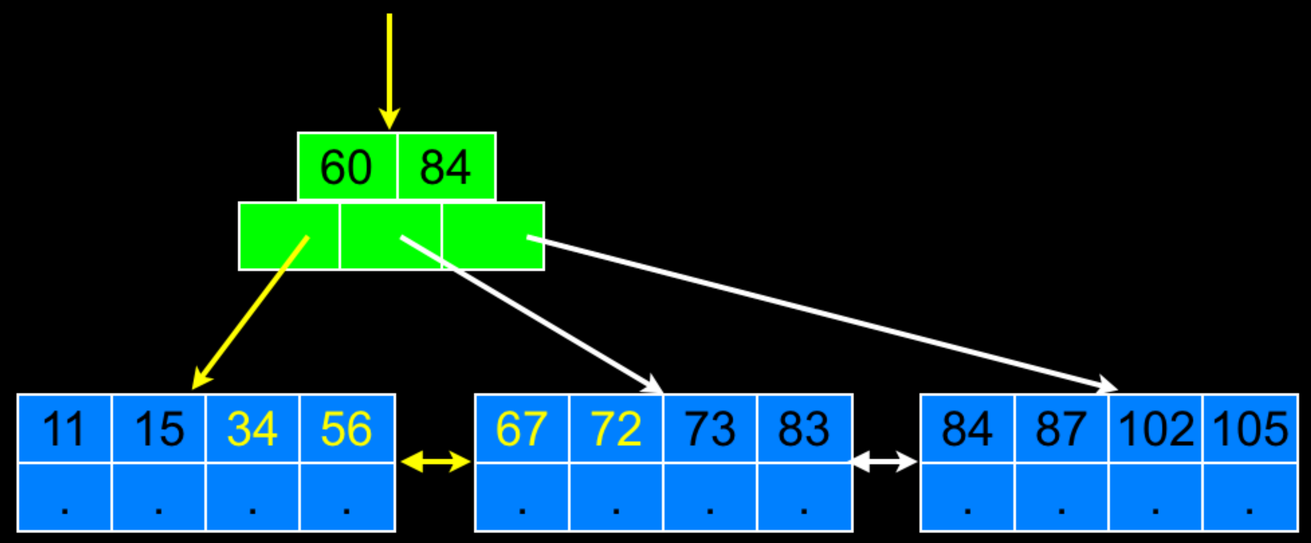
find_range([34;72])



Index Sequential Access Method (ISAM)



find_range([34;72])



1. point query
2. scan along leaves