WEEK 3

USING MULTILEVEL INDICES FOR EVEN MORE SPEED!

STUDENT OBJECTIVES

- Upon completion of this video, you should be able to:
  - Differentiate between a single level index and a multi-level index.
  - Recognize what data structure a multilevel index resembles
  - Distinguish between a B – Tree and B+ Tree
  - Given an order for a tree, draw the node
  - Given a node, determine the order of the tree
  - Given a tree, figure out the depth of the tree (number of levels in the multileveled index)
  - Given a B+ Tree and a key value, show how to traverse the tree to find that value.

OH NO…EVEN OUR INDEX FILE IS HUGE!

- **QUESTION: What could we do if the index file that points to our records is also too big to fit into memory? How could we speed up the search on the index file?**
- ANSWER: what if we made another index ON our index file!
  - Just one index file is a single level index
  - If we have an index on our index (and so on), we have a multileveled index.

MULTI LEVEL INDEX

- The index itself is ordered, so maybe we could create an index on the index!
  - Original file is called the first level index
  - Index to index is called the second level index
- We could repeat, create a third, fourth, … until the top level index fits on ONE block
- Can do this for a any type of index (primary, clustering, secondary) as long as the first level index is more than 1 block

LOOKS KINDA LIKE THIS:

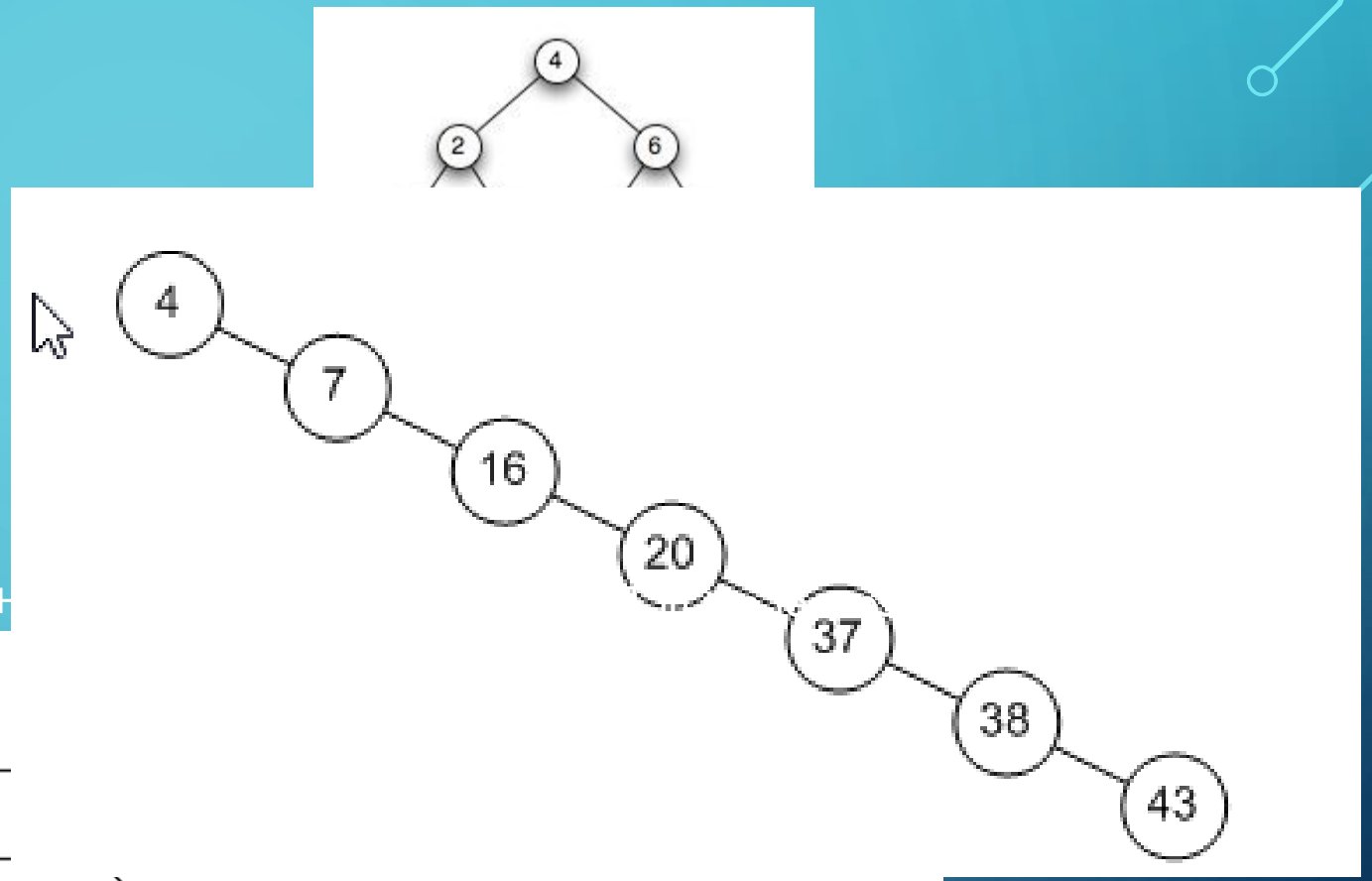
- **QUESTION: All you CS people, what does the structure look like (something you have already learned about in CS2210)?**

TREE

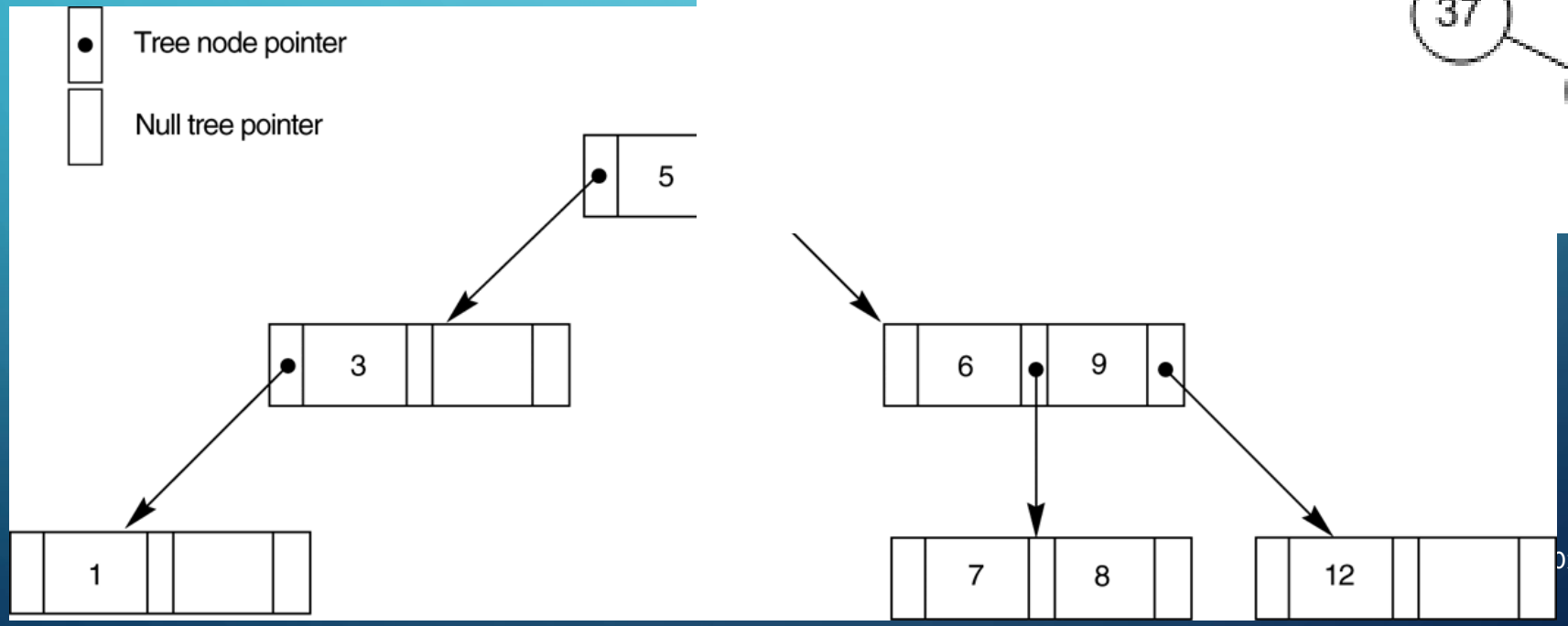
- Each level of our tree will be an index to the next level (the next index)
- A tree is really an index to an index to an index, where the high level index is the root and the lowest level, the leaves, is the final index pointing to the data on the disk.
- Problem: as new records are added to the data file and old records are deleted, we need to keep our index file correct and still pointing to the correct records. How do we insert and delete in our index levels when the file/records are updated?

PROBLEM

- Can become unbalanced!



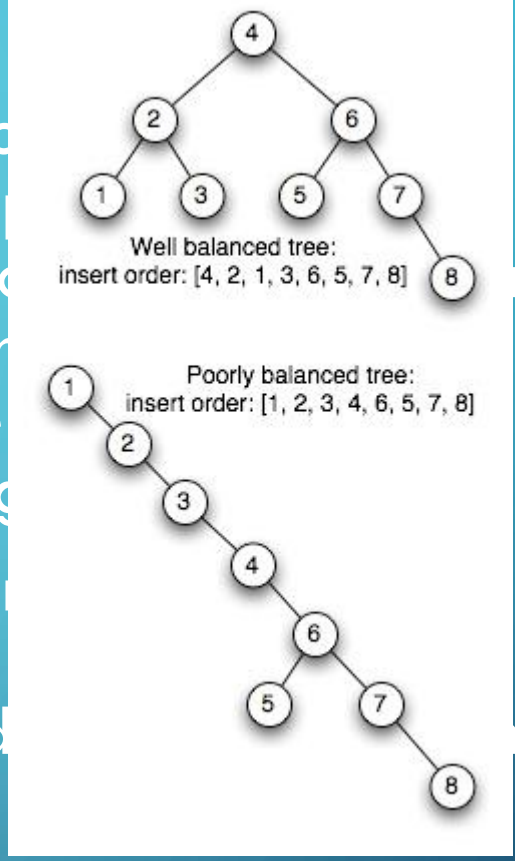
NOTE: This balanced tree is NOT a B+



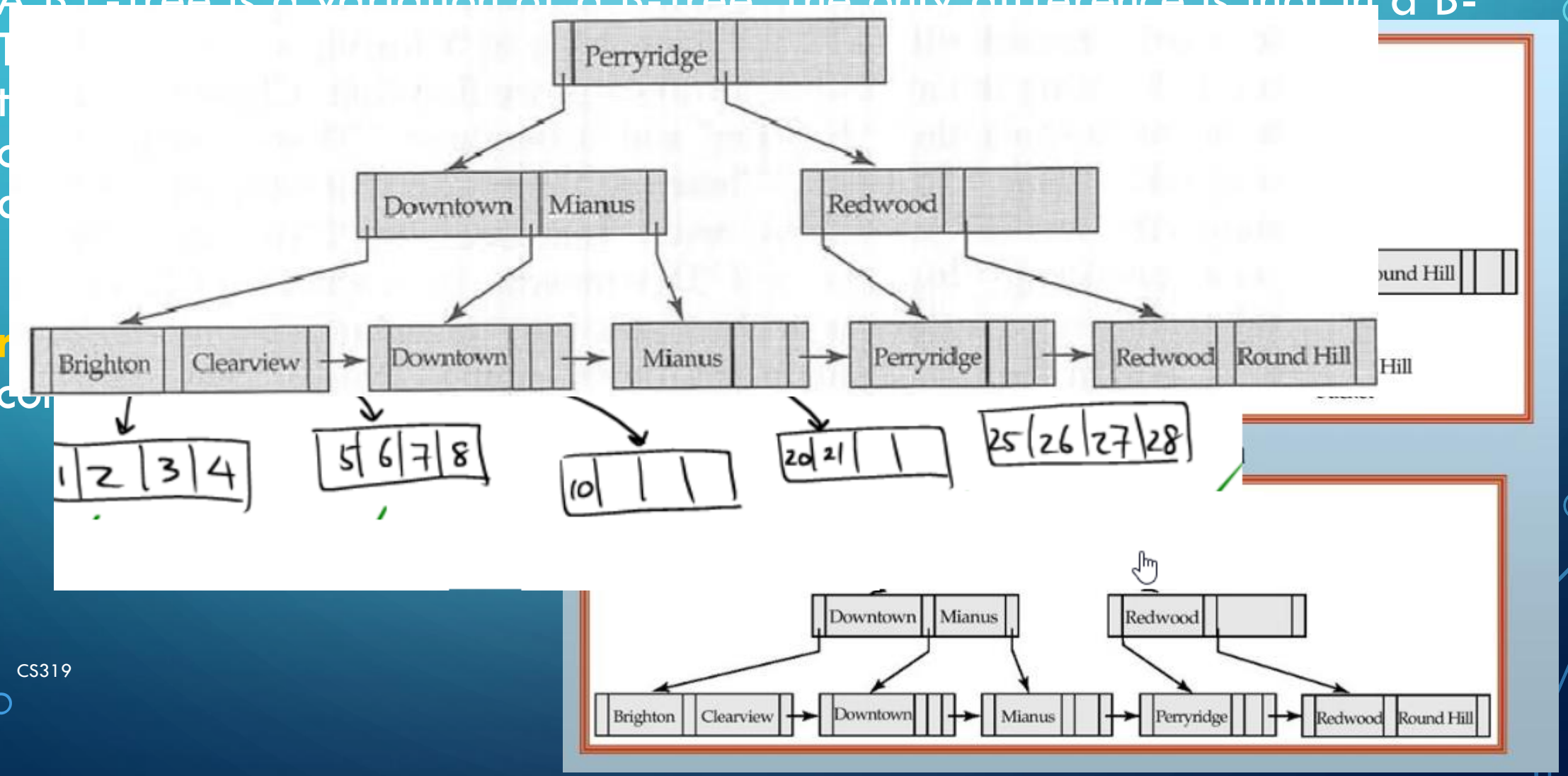
BALANCED TREES

- More versatile than hashing: allows for retrieval based on exact key match, pattern matching, range of values and part key specification
- More versatile than ISAM because the B+-Tree grows as the relation grows
- Retrieval more efficient too
- B+- Trees are a type of Multilevel Index!

- Searching on a sorted file using a binary searc
index) typically requires approximately $\log_2 P$ r
an index with p pages. A multilevel index overc m
by reducing the search range by treating the in
data file and builds an index for the index file
another index file for that index file by splitting
- DBMS uses a *tree data structure* to hold data a
tree consists of a *root node* that can have child
these child nodes can have their own child nod o
children is called a *leaf node*
- The *depth of the tree* is the maximum number of levels between the root node and a leaf node. If the depth is the same from the parent node to each leaf node then the tree is called a *balance tree or B- Tree.*



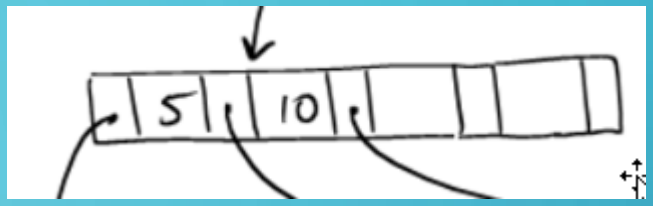
- The *degree (order)* of the tree is the maximum number of children allowed per parent. Larger degrees create broader, shallower trees
- A B+-Tree is a variation of a B-Tree. The only difference is that in a B-



RULES FOR B+-TREES

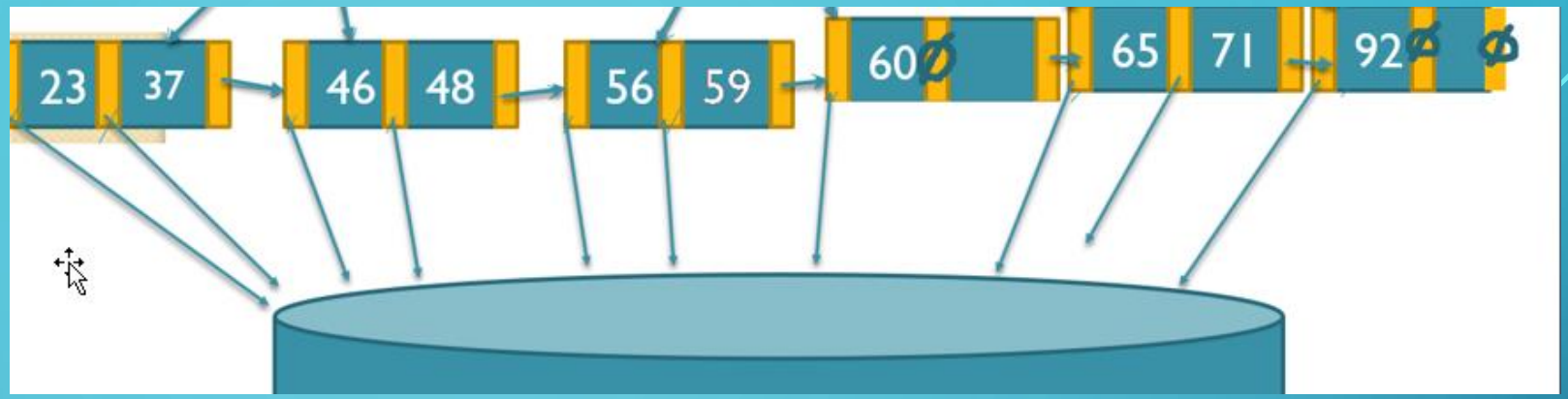
- If the root is not a leaf node, it must have at least 2 children
- For a tree of order n, each node (except root and leaf) must have between n/2 and n pointers and children. If n/2 is not an integer, round the result set up
- For a tree of order n, the number of key values in a leaf node must be between (n-1)/2 and (n-1) pointers and children. If (n-1)/2 is not an integer, round up the result
- The number of key values contained in a non-leaf node is 1 less than the number of pointers
- The tree must always be balanced: i.e. every path from the root node to a leaf must have the same length
- Leaf nodes are linked in order of key values

INTERNAL NODES



- Order p
- Each internal node is of the form <P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>, where q<=p and each P_i is a tree pointer.
- Within each internal node K_1<K_2<...<K_{q-1}
- For all search field values X in the subtree pointed at by P_i, we have K_{i-1} < X <= K_i for 1 <i<q, X<=K_i for i =1, and K_{i-1} < X for i=q
- Each internal node has at most p tree pointers
- Each internal node, except the root has at least [(p/2)] tree pointers. The root node at least 2 tree pointer if it is internal node
- An internal node with q pointers, q <= p, has q-1 search field values.

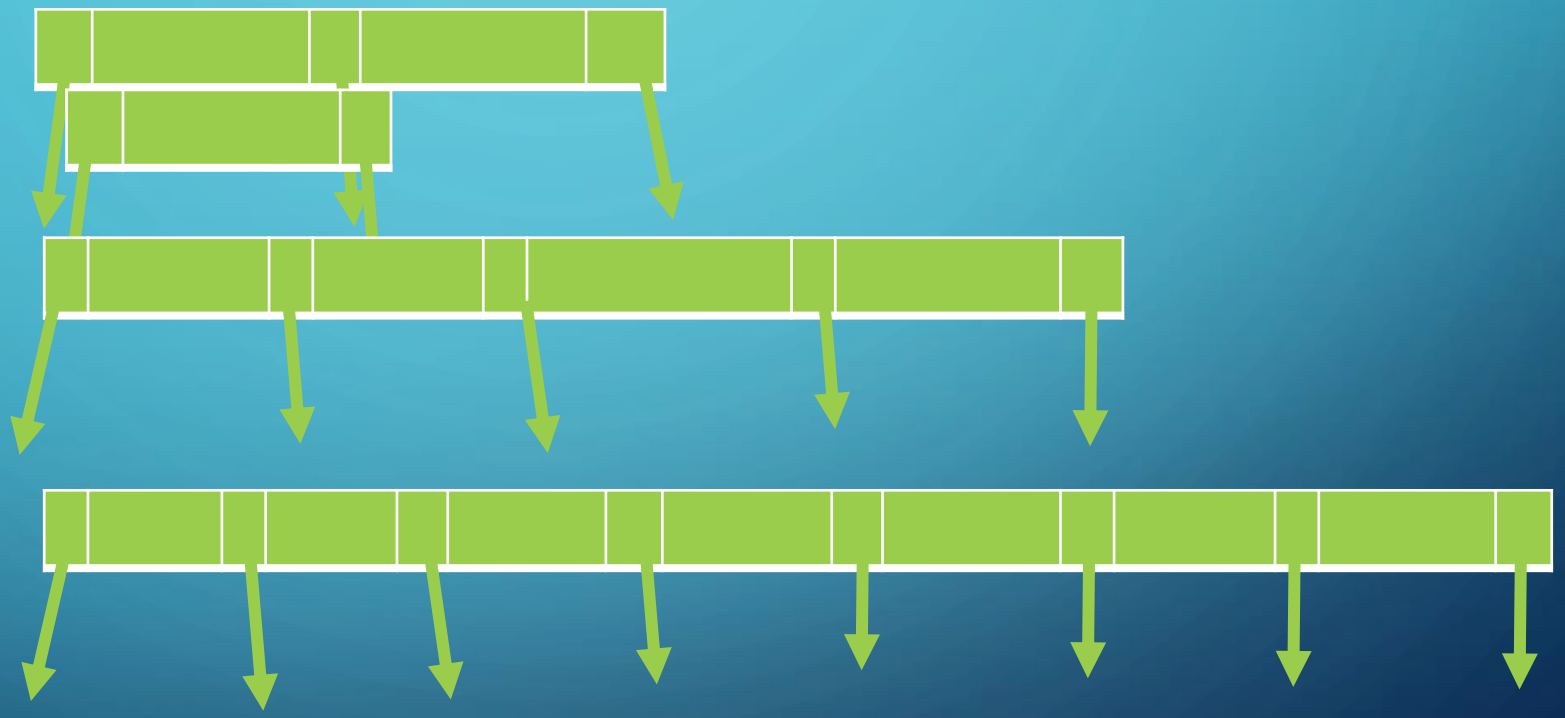
LEAF NODES



- Each leaf node is of the form: <<K1, Pr1>, <K2,Pr2>,...,<Kq-1,Prq-1>, Pnext>
- Pnext points to the next leaf node of the B+ tree
- Within each leaf node, K1<K2<....<Kq-1, q<=p
- Each Pri is a data pointer (points to the disk) that points to the record whose search field value is Ki or to a file block containing the record
- Each leaf node has at least [(p/2)] values.
- All leaf nodes are at the same level.

ALWAYS FIGURE OUT WHAT YOUR NODES LOOK LIKE FIRST BASED ON YOUR ORDER

- Order 3
- Order 2
- Order 5
- Order 8
- We will MAINLY DO ORDER 3!



TRAVERSING (LOOKING FOR SOMETHING IN) A B+ TREE

- Let's look for the item with a key of 65
- Now let's look for an item with a key of 48
- Now let's find all the items between 46 and 65