Assignment 3

Oct 4th
Kyle Klassy

Announcements

- Assignment 2 will be graded by October 8th (Tuesday)
    - Regrade Google form will be set up again
        - Please don't send an email to the TAs or the professor. If you have a grade issue, submit the form explaining the problem and we'll look at it and decide what to do.
- Assignment 3 out right now
    - Link on the website, also available here.
    - Starting tar file available here.

B+ Tree Review

B+ Trees Vocab

- **Relation:** A table
- **Record:** A row in a table
- **Page:** Units of transfer between disk and buffer pool in main memory
    - At least ~8,192 Bytes (depending on machine and program)
- **File:** Literal file in the filesystem that contains the database pages
- **Index Key:** The indexed attribute, what going to get searched for in our tree

Index Files

- Could have lots of pages in our DB file...what if they don’t all fit in main memory at the same time?
  - Then we can’t search through them efficiently-- would take too long.
  - Want to be able to store a structure in main memory to give us some sort of direction.
- Index file
  - Second file that will hold key values of a record and their locations in the DB file
  - One record per page in the original file
    - <1st key value on page, pointer to page holding that key>

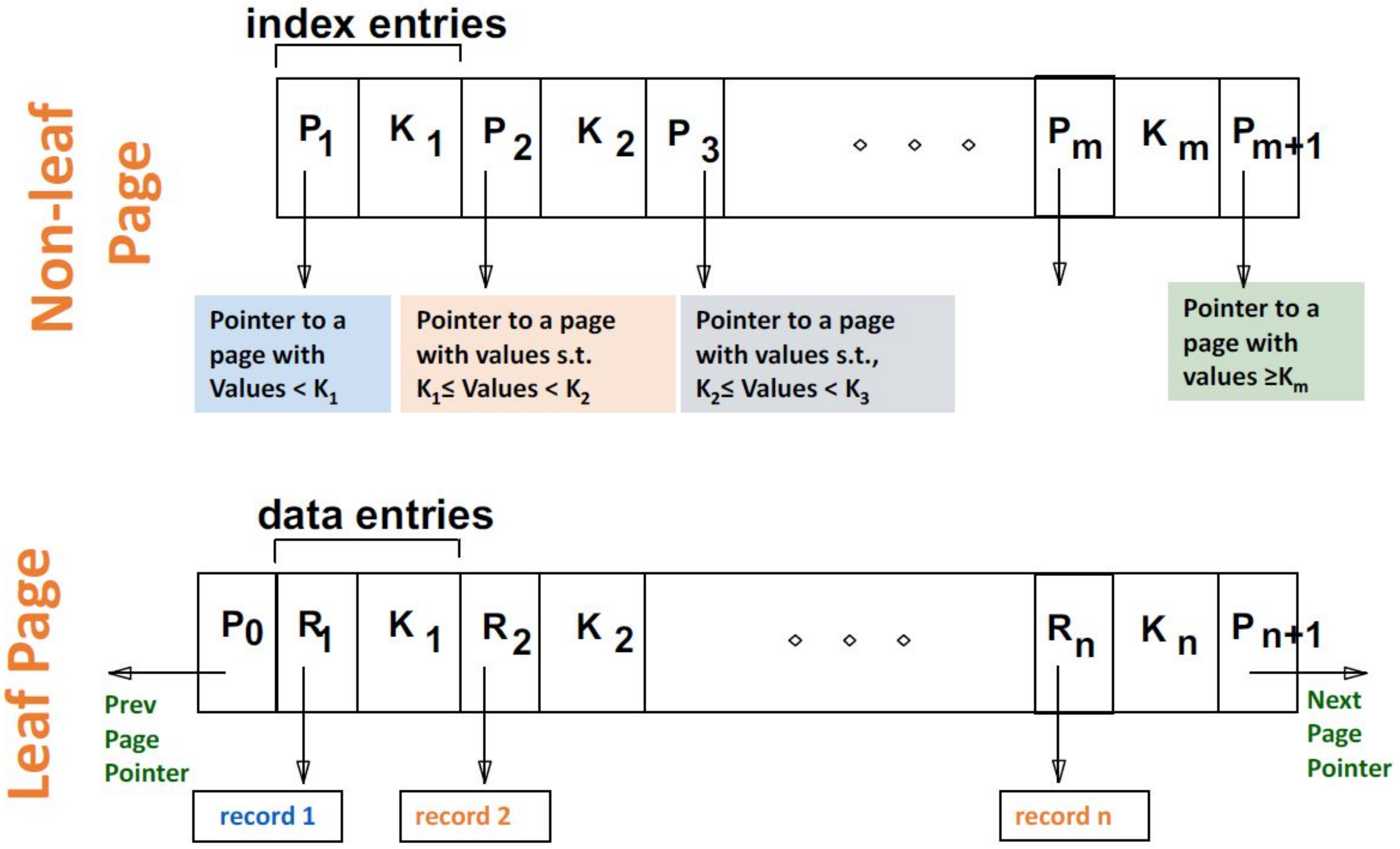
Index Files

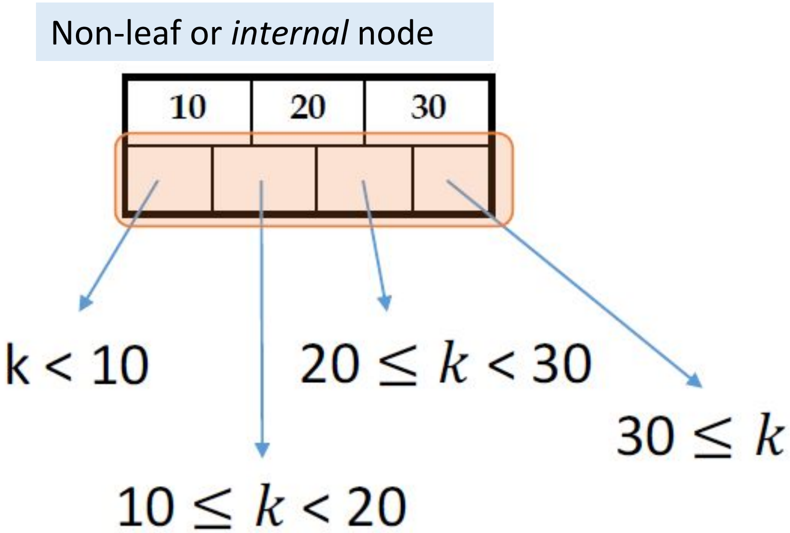
- Essentially created a new table
    - Columns are key on page and location of page.
- Can extend this idea
    - What if the index file doesn’t fit on a single page?
    - What if we repeated the “new table” idea until we got to a level where the “index file” fit on a single page?
        - We’re indexing the index file
- Creates a tree structure!
    - Index file points to places in the index file, until eventually we get to a record in the DB file.
    - The root node fits in a single page

B+ Trees

- Non-leaf nodes are index pages
    - Size of a system page and have “index entries”
        - Key values and pointers
- Leaf pages hold the database records and their index key values
    - Also have pointers to next and previous leaf nodes (database pages)
    - So, all pages are sorted by the index key value.
    - Since all pages are sorted, can easily traverse sequential entries.

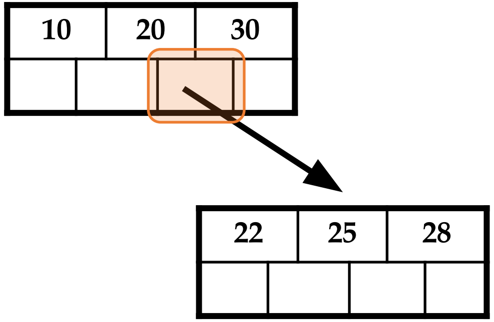
B+ Tree Page Format



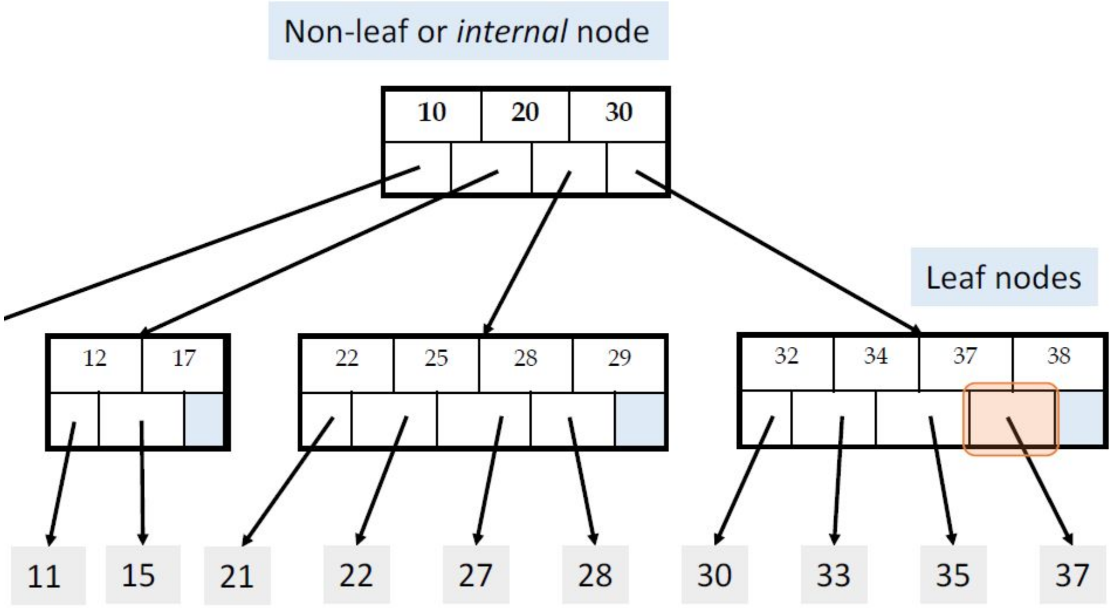


The *n* entries in a node define *n+1* ranges

Non-leaf or *internal* node

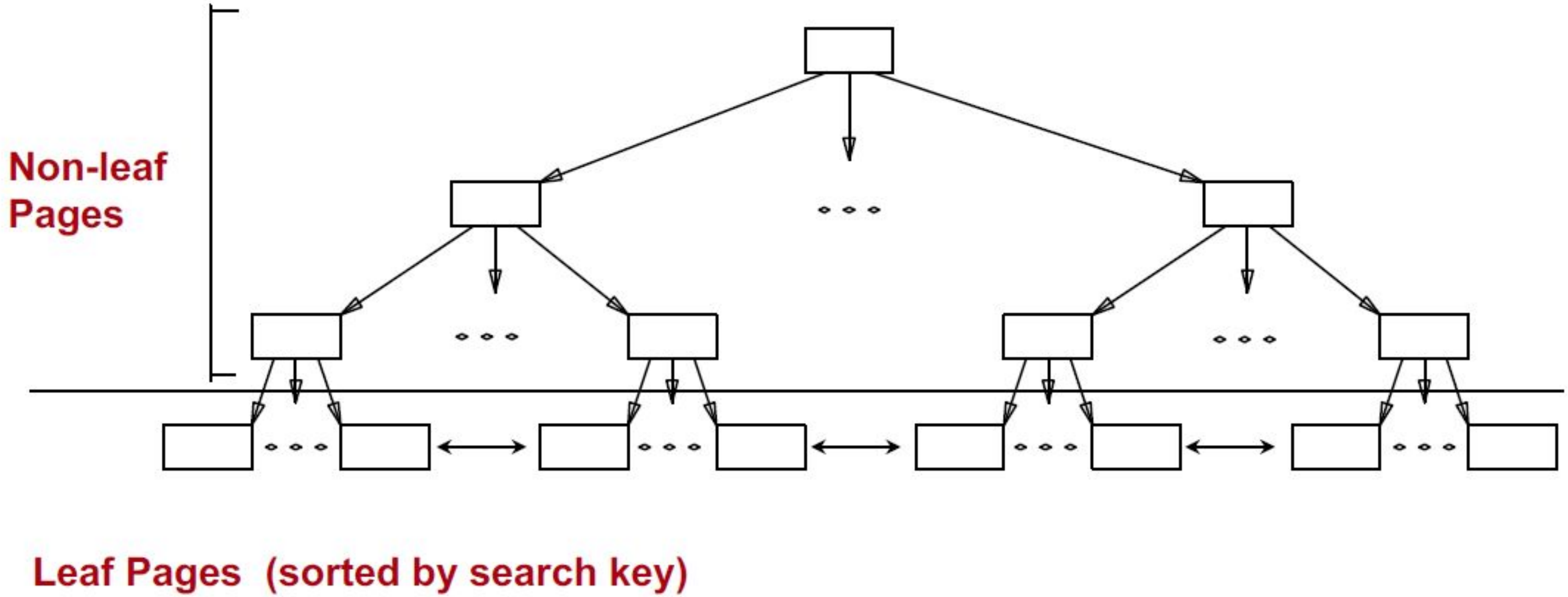


For each range, in a *non-leaf* node, there is a **pointer** to another node with entries in that range



Leaf nodes also have between *d* and *2d* entries, and are different in that:

Their entry slots contain pointers to data records


Non-leaf
Pages
Leaf Pages (sorted by search key)

Structure Requirements

- **Index entries** (either pointers to data or pointers to more index pages) are **non-leaf** nodes
- **Data entries** are **leaf nodes**
  - Doubly linked between each leaf page
- Parameter *d* for each non-leaf node
  - “Order” of the node-- minimum number of key entries per non-leaf node
    - Determined by page size
  - Each non-leaf node (except the root) has to be filled to at least half capacity
    - So the number of entries n is such that $d \leq n \leq 2d$
- The distance from the root to **any** leaf node is the same.

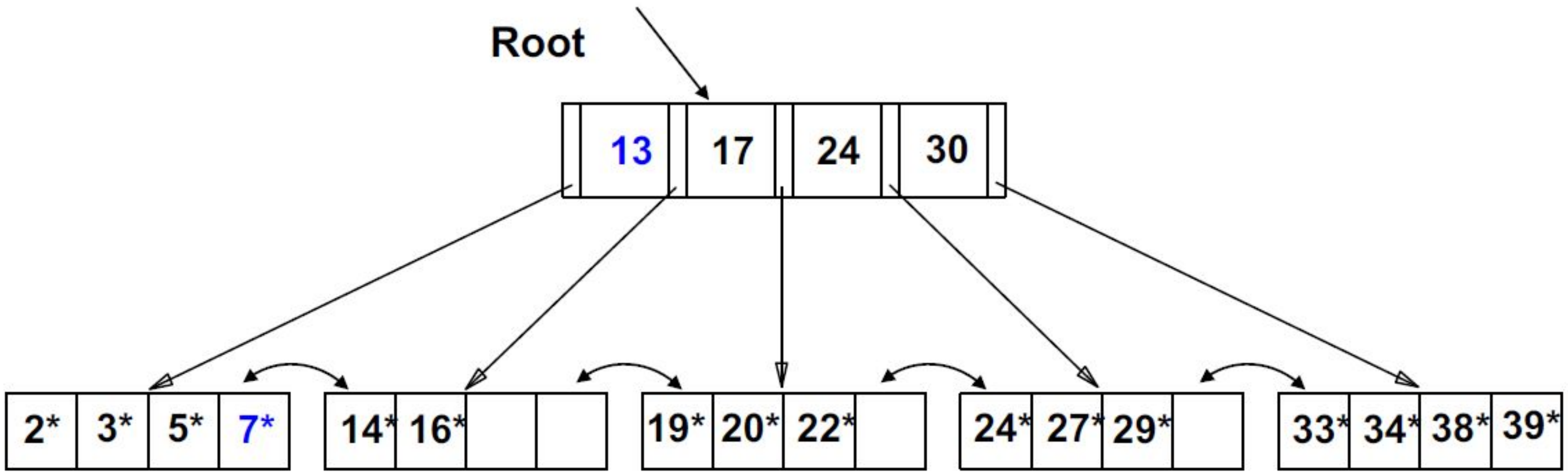
Tree Traversal

- Locating a record
  - Traversal from root to leaf with one I/O per level
  - Trees rarely have more than 3-4 levels
    - Fanout is very large because a <key, pointer> row represents a whole page in memory
  - Navigate down the tree in logarithmic time complexity with linear search of every non-leaf node read into memory
    - However, disk operations dominate time complexity
      - Why we want such a high fanout at each node.
- Range search
  - Locate the record at the start of the range and traverse the leaves of the B+ Tree

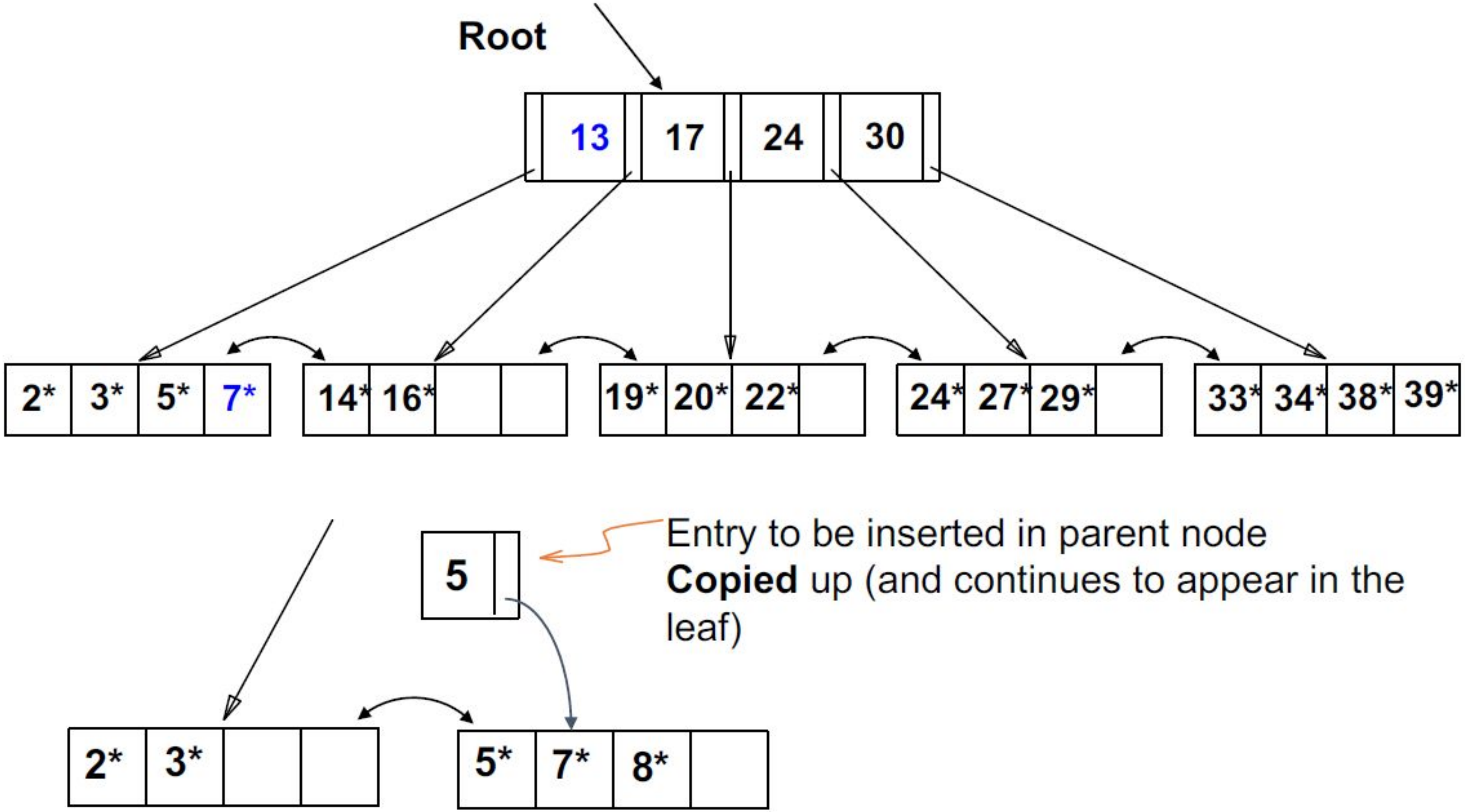
B+ Tree: Insert Algorithm

- Find correct leaf *L.*
- Put data entry into *L.*
  - If L has enough space, done!
  - Else, must **split** *L* (into *L* and a new node *L2*)
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split non-leaf node, redistribute entries evenly by **pushing up** the middle key.
- Splits "grow" tree; root split increases height.
  - Tree growth: get wider or one level taller at top.

Inserting 8* into B+ Tree

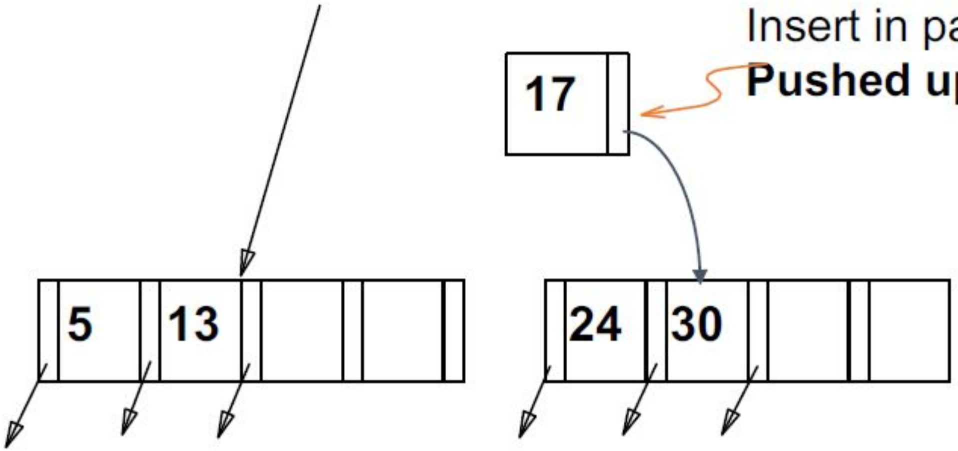


Inserting 8* into B+ Tree



Entry to be inserted in parent node
Copied up (and continues to appear in the leaf)

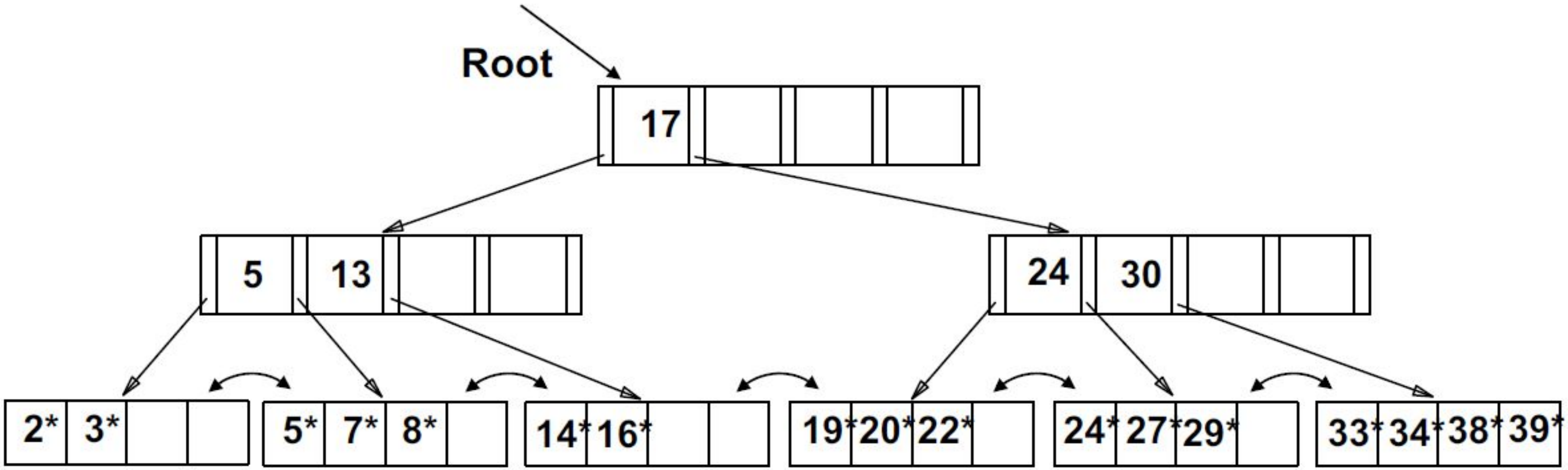
Inserting 8* into B+ Tree



Insert in parent node.
Pushed up (and only appears once in the index)

Minimum occupancy is guaranteed in both leaf and index page splits

Inserting 8* into B+ Tree



- Root was split: height increases by 1

B+ Tree Index Manager

BTreeIndex Functions to Implement

- Implement `BTreeIndex` constructor
  - Creates a new index file
- `insertEntry()` function given <key, rid> pair
  - Going to take up most of your time, more complicated than you think
  - Split nodes, copy up values, and push up index keys
- `startScan()` function given range to search
  - Traverse down the tree to get to a leaf page to put into the buffer
- `scanNext()` function
  - Get next record from current page being scanned. If you've scanned the whole page, move to its sibling.
- `BTreeIndex` destructor, `endScan()` as well

Other B+ Tree Classes

- `PageFile`: Will store all relations (leaf nodes)
  - Linked by prevPage/nextPage links
- `BlobFile`: Will store your B+ index file
  - Each page in the file is a node in the B+ tree.
  - Only have to worry about creating these types of files (just the index file).
- `FileScan`: Scans records in the “base” relation file
  - Can use it to start your B+ index file.
  - Get a new file, scan it to get all records, insert them one by one into your tree.

Assumptions

1. All records in a file have the same length
    a. i.e. for a given attribute, its offset in the record is always the same
2. Only need to support single-attribute indexing, not a composite attribute where the key has more than one value
3. The indexed attribute may only be of type integer.
4. We will never insert two data entries into the index with the same key value
    a. Simplifies implementation, think about why.
    b. Duplicate handling: Cow Book chapter 10.7

Getting Started

- Get the files from here and untar using `tar xvzf p3_Btree.tar.gz`
- Go through Doxygen pages (in doc folder in tar file)
    - Very good documentation of all classes in B+ Tree
- Read btree.h carefully
    - Lots of structs, variables, and functions defined here that will be very useful.
- Once you've understood the structure of the tree, look at the tests already implemented in main.cpp, then implement some of your own.
- Move on to implementing the `BTreeIndex` functions.

Additional Notes

- Must add your own tests somewhere
  - Main.cpp is a good place to add them
  - I suggest writing these first!
- Outline.txt file explaining test location and what they're testing
  - 15% of your grade is the tests!
  - No minimum number you must write, but they should try to be comprehensive.
- Make sure your code compiles on the CS lab machines before submitting
- This will take you a while! Get started early.
  - Due October 20th @ 11:59pm