B+ Trees

Some interesting, practical trees...

Learning Goals

After this unit, you should be able to...

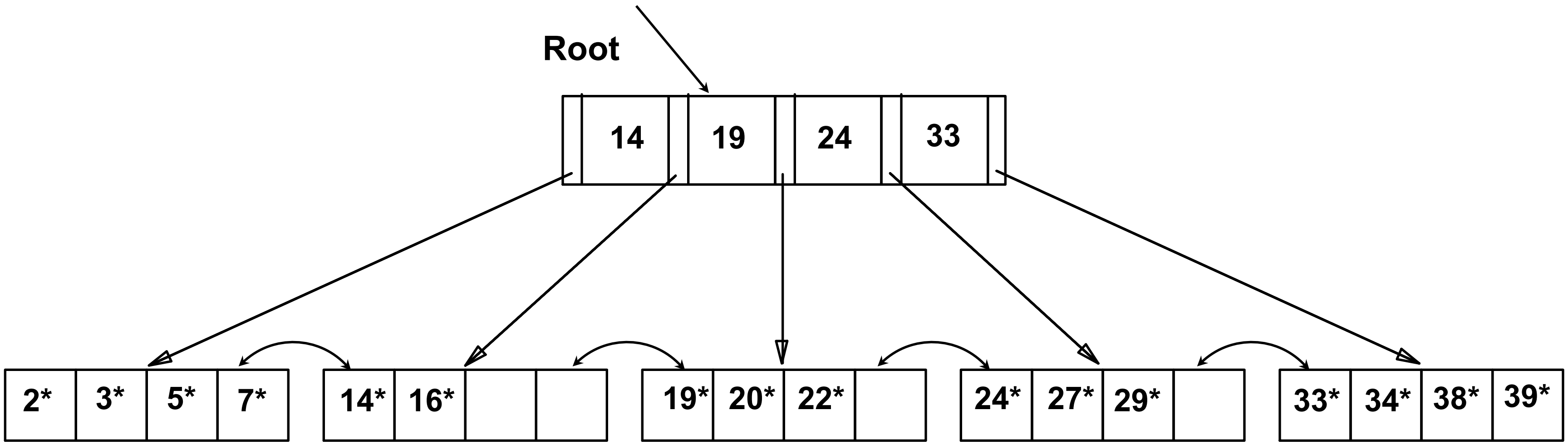
- Describe the structure, navigation and complexity of an order m B+ tree.
- Insert and delete elements from a B+ tree.
- Explain the relationship among the order of a B+ tree, the number of nodes, and the min and max capacities of internal and external nodes.
- Give examples of the types of problems that B+ trees can solve efficiently .
- Compare and contrast B+ trees and hash data structures. Explain and justify the relationship between nodes in a B+ tree and blocks/pages on disk.
- Justify why the number of I/Os becomes a more appropriate complexity measure (than the number of operations/steps) when dealing with larger datasets and their indexing structures (e.g., B+ trees).

B^+-Trees

(Note: This material is not in our two textbooks.)

A B^+-tree is a very efficient, dynamic, balanced, search tree that can be used even when the data structure is too big to fit into main memory. It is a generalization of a binary search tree, with many keys allowed per internal and external node.

Here is an example of a B^+-tree containing 16 data entries in the leaves. Can you see any similarities to a binary search tree?



General Comments

- Like other search structures, a B^+-tree is an *index*.
- The keys in the tree are ordered.
- Internal nodes simply “direct traffic”. They contain some key values, along with pointers to their children.
- External nodes (leaves) contain all of the keys. In the leaf pages, each key also has a “value” part. So far in this course, we have often considered <key, value> pairs. With B^+-trees, we can do this, too; however, sometimes the “value” is a pointer (e.g., 10 bytes long) that contains the disk address of the object to which the key applies (e.g., employee record/structure, video, file). This is a great idea, especially when the data values would take up too many bytes of memory/storage.

General Comments

- A typical size for a node in a B^+-tree is _________, which is a common page size for file systems.
- An *I/O* or *an I/O operation* (input-output operation) is defined to be a transfer of a "block" or page of data between ________________ and _________________.
- Disk access (I/O) times exceed memory-access times by several orders of magnitude. Therefore, the number of I/Os will provide us with a *very* useful complexity measure for many types of applications.
- B^+-trees belong to a family of trees called B-trees.
- B+ trees are very heavily used in relational database systems.

Order of a B$^+$-Tree

- Let us define the order *m* of a B+ tree as the maximum number of *data entries* (e.g., <key,pointer> pairs) that can fit in a leaf page (node).
  - Usually, longer keys (e.g., strings vs. integers) mean that fewer data entries can fit in a leaf page.
- Note: Different authors may have different definitions of order. For example, some authors say that the order is:
  - the *minimum* number *d* of search keys permitted by a non-root node. [Ramakrishnan & Gehrke]. The maximum number of search keys that will fit in a node is therefore 2*d*, which is what we call *m*.
  - the *maximum* number *d* of <u>children</u> permitted in an internal node [Silberschatz, Korth, & Sudarshan]

Example: Two B$^+$-Trees of Order 3

- This example shows two different order 3 B+ trees and the (same) data records that they point to.

Download the PDF slide (full page), separately, on WebCT.

Properties of a B^+-Tree of Order *m*

- All leaves are on the same level.
- If a B+ tree consists of a single node, then the node is both a root and a leaf. (It's an external node in this case, not an internal node.)
- "Half-full" rule, part 1: Each *leaf node* (unless it's a root) must contain between $\lceil m/2 \rceil$ and m <key,pointer> pairs.
- "Half-full" rule, part 2: Each *internal node* other than the root has between $\lceil (m+1)/2 \rceil$ and $m+1$ *children*, where $m \geq 2$.
  - How does the number of keys in an internal node relate to the number of children (child pointers) that it has?

Properties of a B$^+$-Tree of Order m (cont.)

- Equivalently, each internal node other than the root contains between $\lfloor m/2 \rfloor$ and m *search keys:* $x_1 < x_2 < \ldots < x_k$ where $\lfloor m/2 \rfloor \le k \le m$
  - The internal node's i^{th} child has keys in the range $[x_{i-1}, x_i)$ for $i = 1, 2, \ldots, k+1$ (where x_0 and x_{k+1} are arbitrarily small and large values, respectively).
- The *root* node contains between 1 and m search keys (or between 1 and m <key,pointer> pairs if the root is a leaf).
- Each leaf node also contains pointers to its adjacent siblings—forming a doubly-linked list. Why might this be useful?

Sample Calculations

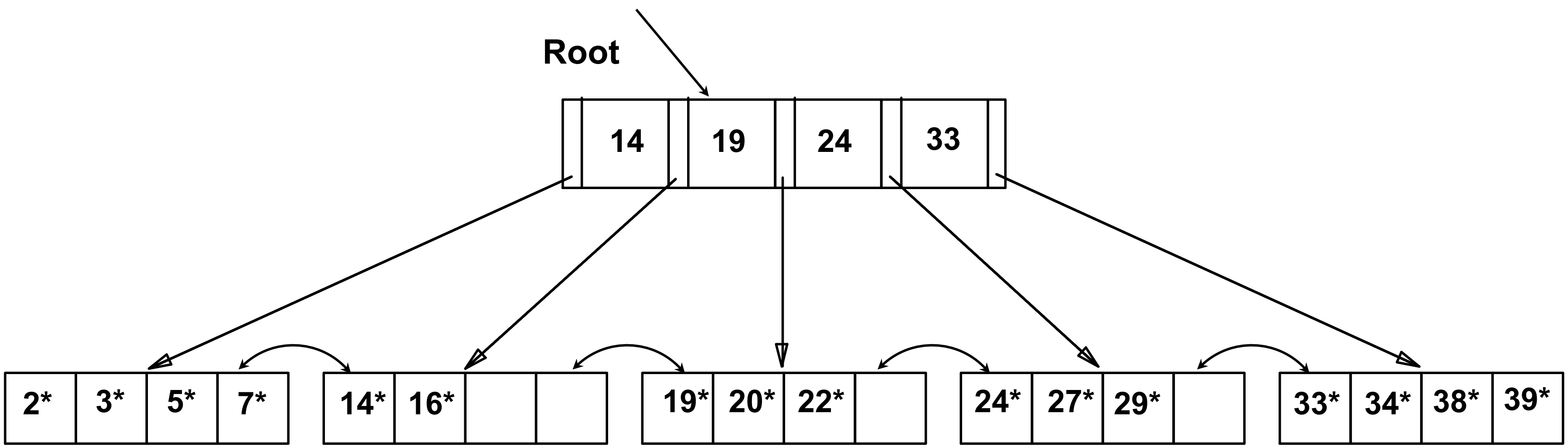
Let's compute the number of *data entries* per leaf page, given two scenarios:

- <key, pointer>: 4K page, 4-byte integer key, 10-byte disk address:

- <key, record>: 4K page, 4-byte integer key, 800-byte data record having many fields:

Searching a B$^+$-Tree of Order 4

- Searching begins at the <u>root</u>, and key comparisons direct us to a leaf
- Example: Search for 5*, 15*, all data entries ≥ 22* …

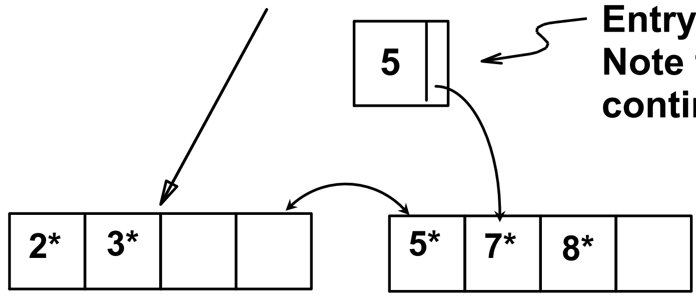


Inserting into a B^+-Tree

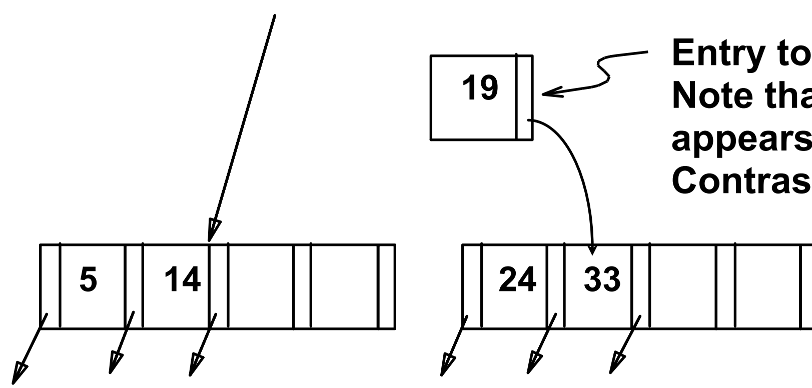
- Find correct leaf *L*
- Try to put (key,pointer) pair into *L*
  - If *L* has enough space, then put it here.
  - Else, *split* *L* (into *L* and a new node *L2*)
    - Redistribute *L*'s entries evenly between *L* and *L2*
    - **Copy up** the middle key, i.e., recursively insert middle key into parent of *L* and add a pointer from *L*'s parent to *L2*
- When inserting into an internal node *V*:
  - If *V* has enough space, then put it here.
  - Else, split *V* (into *V* and a new node *V2*)
    - Redistribute *V*'s entries evenly between *V* and *V2*
    - **Move up** the middle key. (Contrast this with leaf splits.)
- Splits "grow" the tree by making it wider. If the root splits, the tree increases in height by one.

Inserting 8* into the Sample B+-Tree from Previous Pages

- Observe how minimum occupancy is guaranteed in leaf page splits
- Note the difference between *copy up* and *move up*.

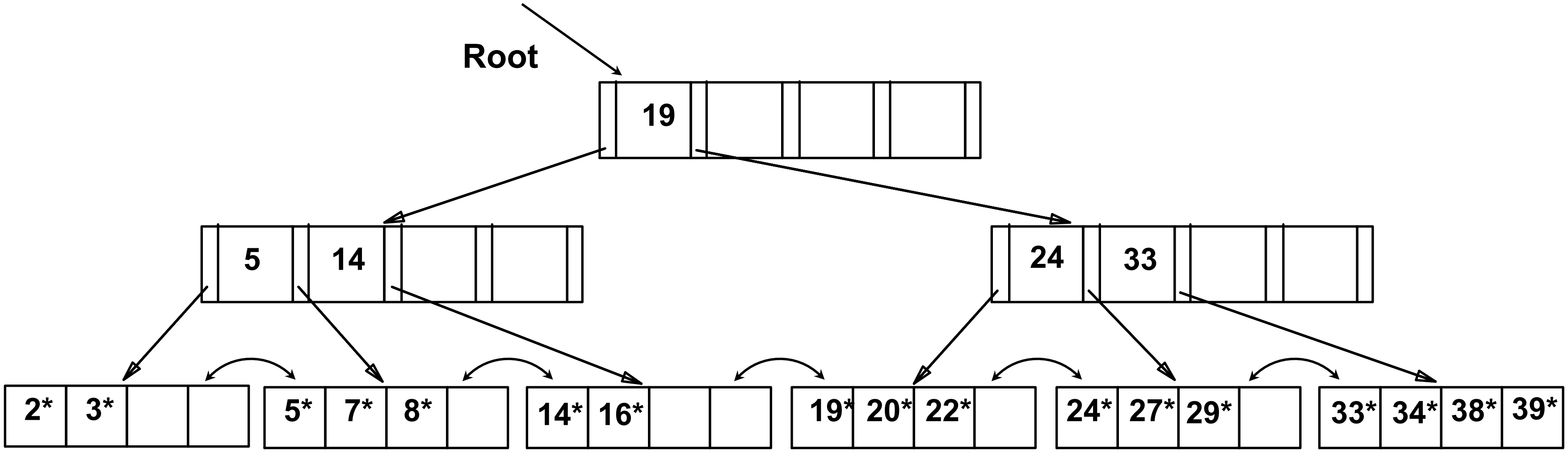


Entry to be inserted in parent node. Note that 5 is copied up and continues to appear in the leaf.



Entry to be inserted in parent node. Note that 19 is moved up and only appears once in the internal nodes. Contrast this with a leaf split.

B^+-Tree After Inserting 8*



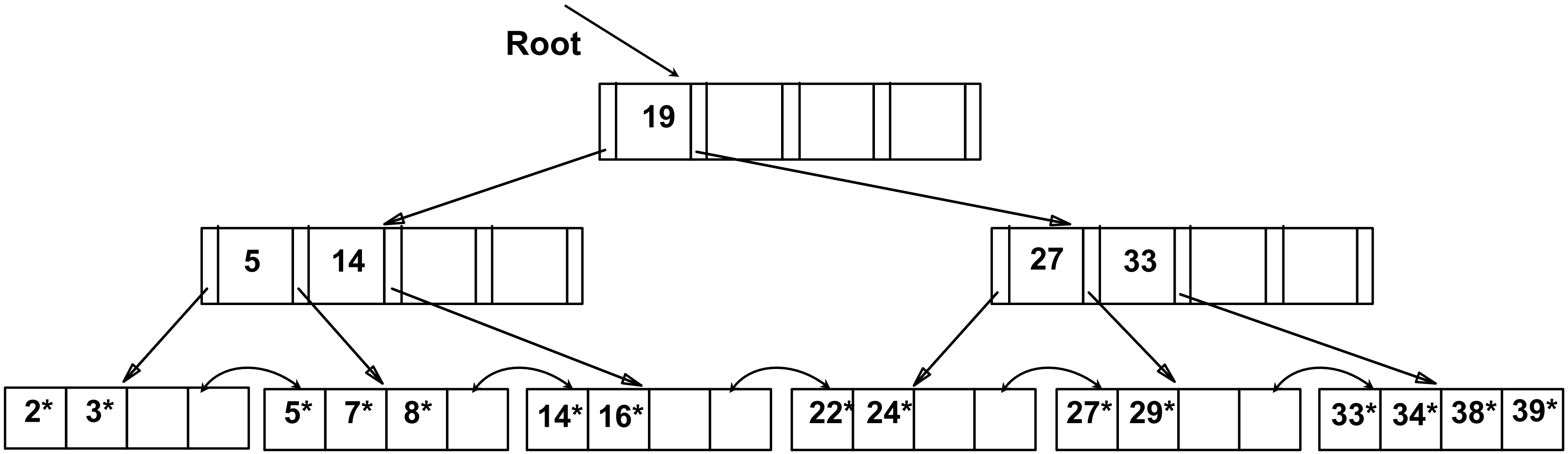
❖ Note that root was split, leading to increase in height

❖ In this example, we can avoid splitting by re-distributing entries; however, this is usually not done in practice.

Deleting from a B^+-Tree

- Find leaf *L* containing (key,pointer) entry to delete
- Remove entry from *L*
  - If *L* meets the "half full" criteria, then we're done.
  - Otherwise, *L* has too few data entries.
    - If *L*'s right sibling can spare an entry, then move smallest entry in right sibling to *L*
    - Else, if *L*'s left sibling can spare an entry then move largest entry in left sibling to *L*
    - Else, <u>*merge*</u> *L* and a sibling
- If merging, then recursively delete the entry (pointing to *L* or sibling) from the parent.
- Merge could propagate to root, decreasing height

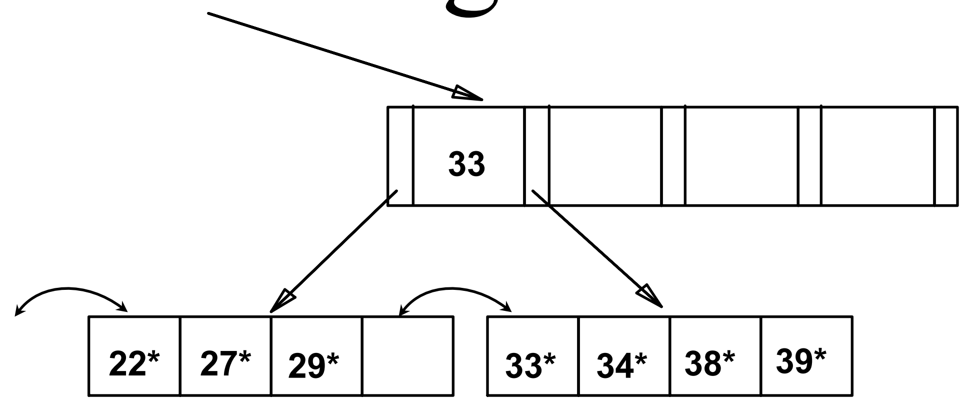
Tree after (Inserting 8* and then) Deleting 19* and 20* …

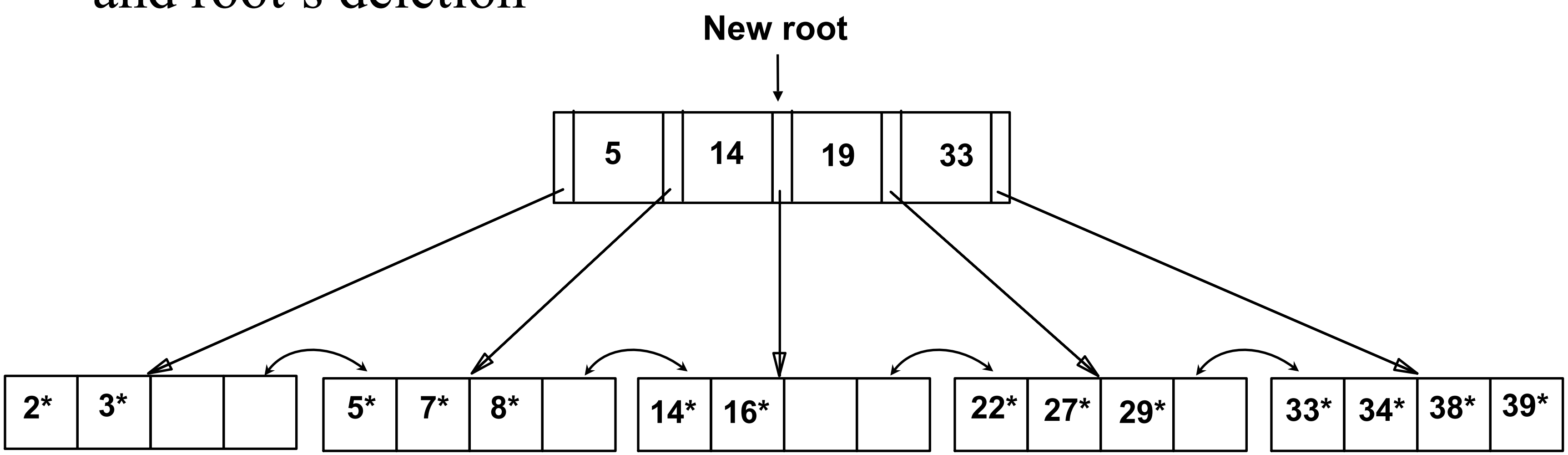


- Deleting 19* is easy (root can continue to hold 19 since it still directs searches properly)
- Deleting 20* is done with re-distribution; notice how key 27 is *copied up* to replace 24.

... and then Deleting 24*

- Merging two leaves causes recursive delete of search key 27 from parent
- Merging root’s children causes “pull down” of search key 19 from root and root’s deletion





Some B^+-Tree Statistics, in Practice

- Typical order = 200
- Typical fill-factor = 66% (about 132 pairs per leaf)
- Average fanout (# of children) = 133
- Typical capacities (approx. # of records pointed to):
  - Height 3: 133^3 * 132 = 310,548,084 records
  - Height 2: 133^2 * 132 = 2,334,948 records
- Can often retain (cache/hold) the top 2 levels in the buffer pool (i.e, RAM) for an actively used B+ tree:
  - Level 0 = 1 page = 4 KB
  - Level 1 = 133 pages = 0.5 MB (approx.)
  - Level 2 = 17,689 pages = 66.5MB (approx.)
  - Level 3 = 2.35M pages = 9.4 GB (approx.)

Equality and Range Searches

- B^+-trees are great for performing equality or range searches. Examples:
  - Find all information about the student whose ID is 78358990.
  - Find all employees who make more than $100,000 per year.
  - Find all employees who make less than $17,000 per year.
  - Find all employees who make between $46,500 and $46,999 per year.

- Indexes can be created on unique or non-unique search keys, but in order for us to efficiently look up an employee that makes *x* dollars per year, we need to build an index for the salary field (else we're forced to do a linear (exhaustive) search).

- Hash indexes are great for equality searches, but they're not useful for range searches. Why not?

Complexity Questions about B^+-Trees

Let us assume that an order *m* B+ tree contains *n* unique keys (e.g., customer numbers for *n* customers). Suppose further that there are N nodes in this tree.

- In the worst case, how many *nodes* need to be visited to find out if a particular customer number exists?

- How many *nodes* need to be visited to print all the keys in order?

- Why should we measure complexity in terms of I/Os rather than, say, CPU time or the # of instructions executed?

Learning Goals

After this unit, you should be able to...

- Describe the structure, navigation and complexity of an order m B+ tree.
- Insert and delete elements from a B+ tree.
- Explain the relationship among the order of a B+ tree, the number of nodes, and the min and max capacities of internal and external nodes.
- Give examples of the types of problems that B+ trees can solve efficiently .
- Compare and contrast B+ trees and hash data structures. Explain and justify the relationship between nodes in a B+ tree and blocks/pages on disk.
- Justify why the number of I/Os becomes a more appropriate complexity measure (than the number of operations/steps) when dealing with larger datasets and their indexing structures (e.g., B+ trees).