

Basic Index Structures

CSC443, Winter 2019

Sayyed Nezhadi

Chapter 10

Motivation for Indexing

- Heap file supports sequential scan of records. Theoretically, this is sufficient to implement all query operators in SQL. However, the resulting efficiency would be impractically poor (requires full scan).
- Indexing is to create secondary data storage to minimize disk I/O for record scan and search.

Sequential File

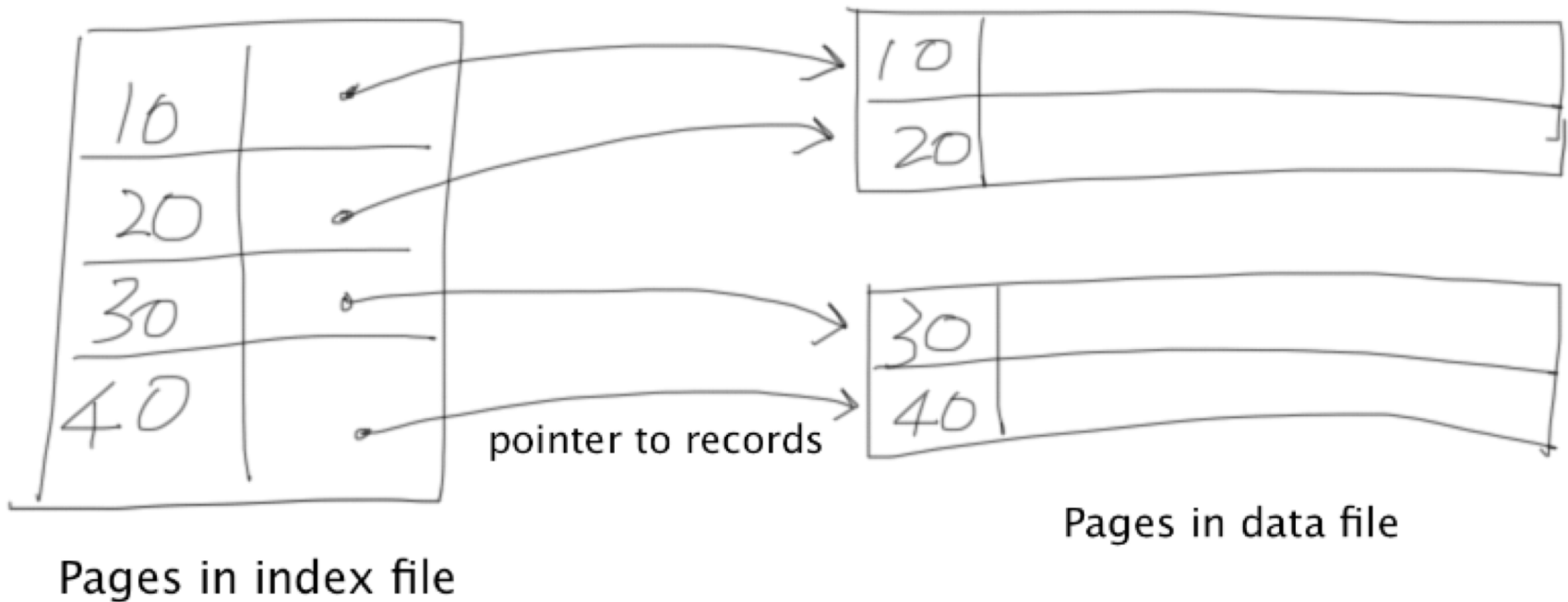
- A heap file that all records are sorted by some key attributes on each page and across multiple pages
- Still requires record scan for searching a record but requires less disk reads
- Search Algorithm:
 - Load page pointers from the heap page directory
 - Perform binary search by fetching the pages
- The total number of disk I/O required (worst case):

$$\log_2(N_{\text{data pages}}) + (N_{\text{directory pages}})$$

Dense Indexes

- Secondary storage which stores redundant data in order to improve the efficiency of query processing
- A *dense index* is a sequential file whose records only contain:
 - The values over some key attach index page contains much more recordsributes
 - A pointer to the address of the data record in the data
- Motivation:
 - There may be a significant reduction in the record size in the index file comparing to the records in the data file
 - Each index page contains much more records, resulting in far less disk I/O to scan sequentially (fewer pages to retrieve)

Dense Indexes

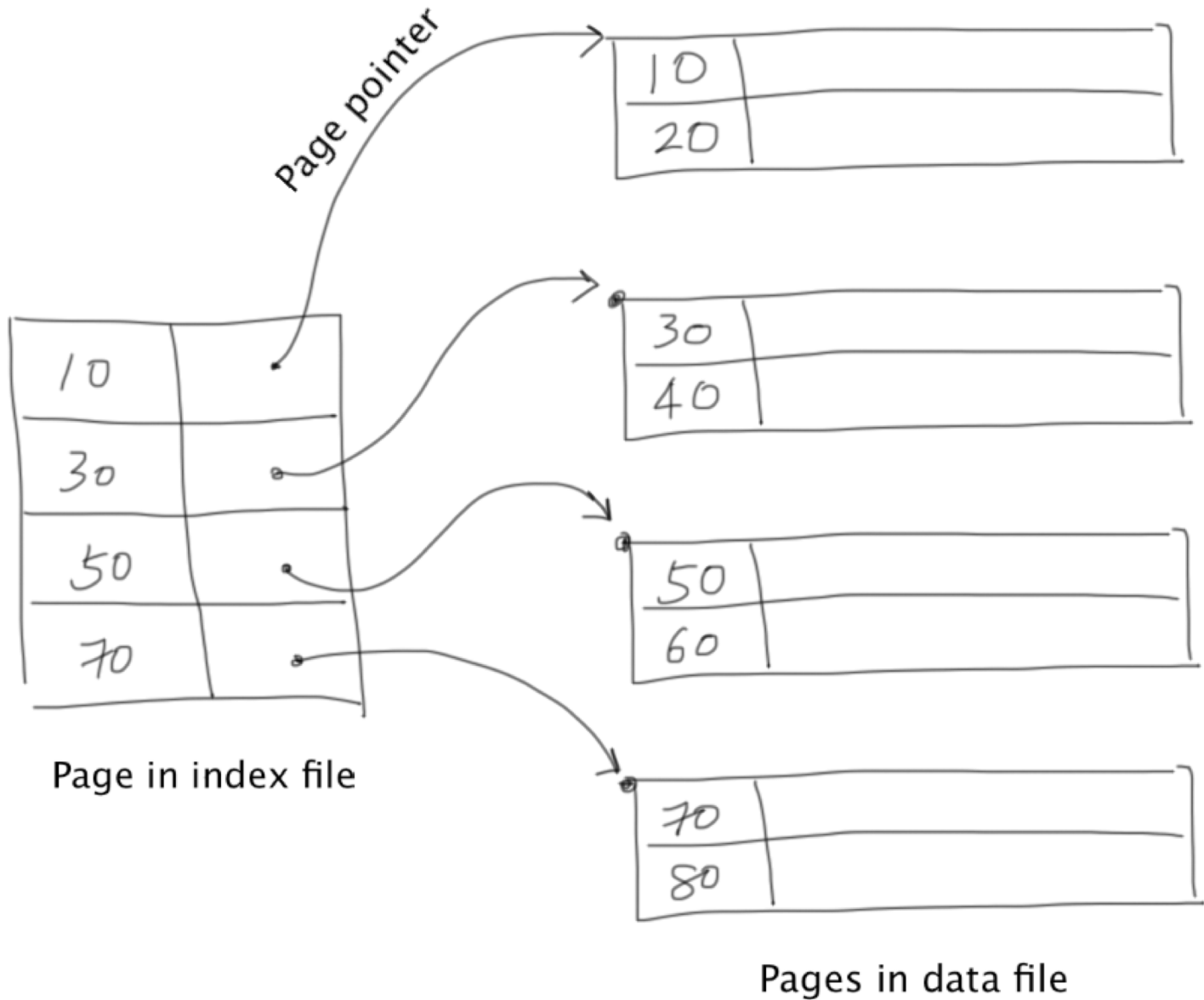


Sparse Indexes

- Combines the strengths of sequential files and dense indexes, for fast record lookup using binary search
- Data file is a sequential file (sorted by a key)
- The sparse index only stores the key values and address for the *first* record of each data page in the data file
- Motivation: To reduce the number of index pages resulting in less disk I/O

➡ **Assumption**: for simplicity, we assume that we only need to retrieve the first occurrence of a record.

Sparse Indexes



Sparse Indexes

- The total number of disk I/O required (to look for a record):

$$\log_2(N_{\text{index pages}}) + 1$$

- “log” term is the number of page reads needed to locate the index page
- “1” page read to load the data page

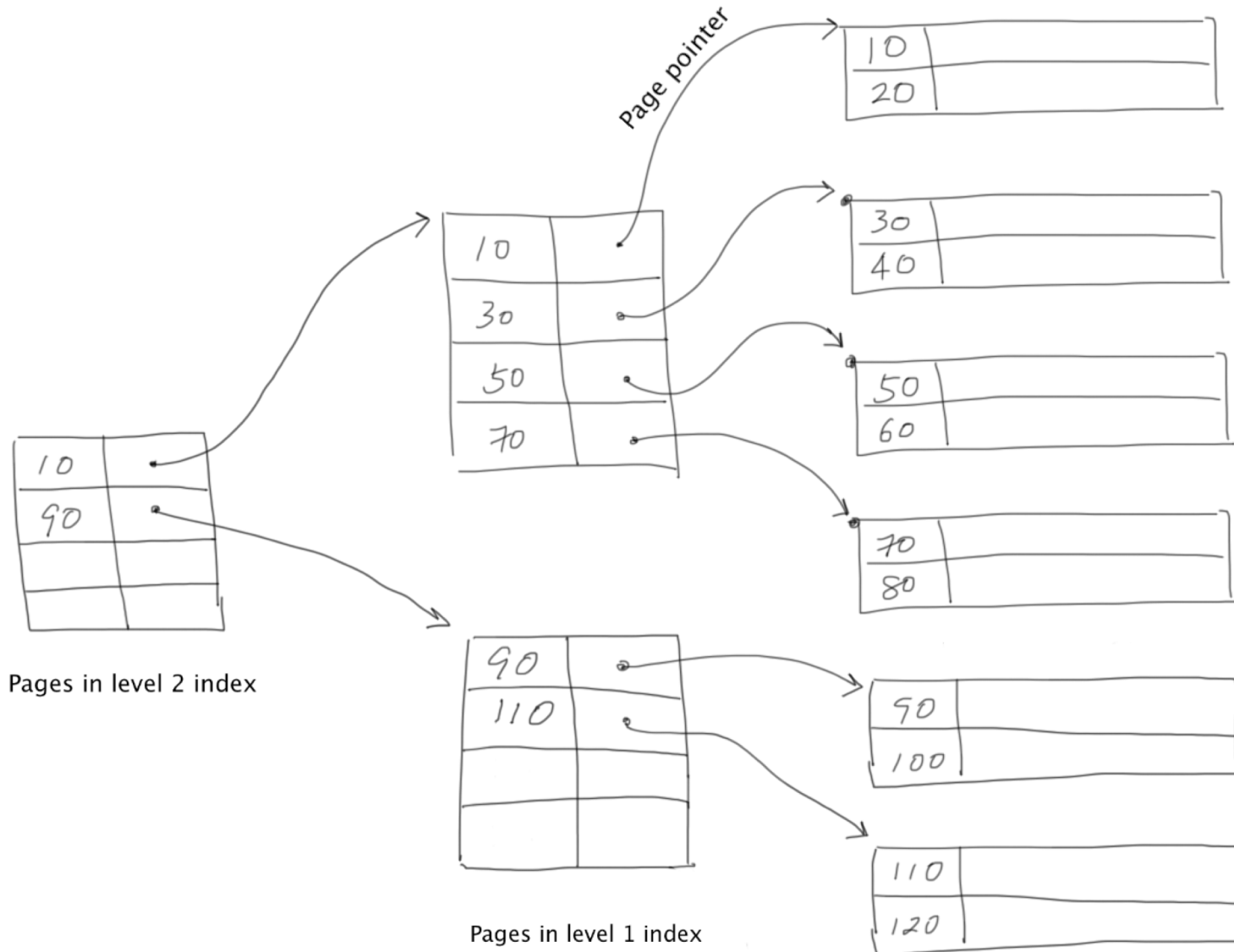
Multilevel Sparse Indexes (Tree Indexes)

- We can improve search using sparse index even more by indexing the index by yet another sparse index.
- The total number of disk I/O required (to look for a record):

$$\log_2(N_{I_2}) + 2$$

- " N_{I_2} " is the number of pages for second level index
- "log" term is the number of page reads needed to locate the index page in " I_2 "
- "1" page read to load the index page in " I_1 "
- "1" page read to load the data page

Multilevel Sparse Indexes (Tree Indexes)



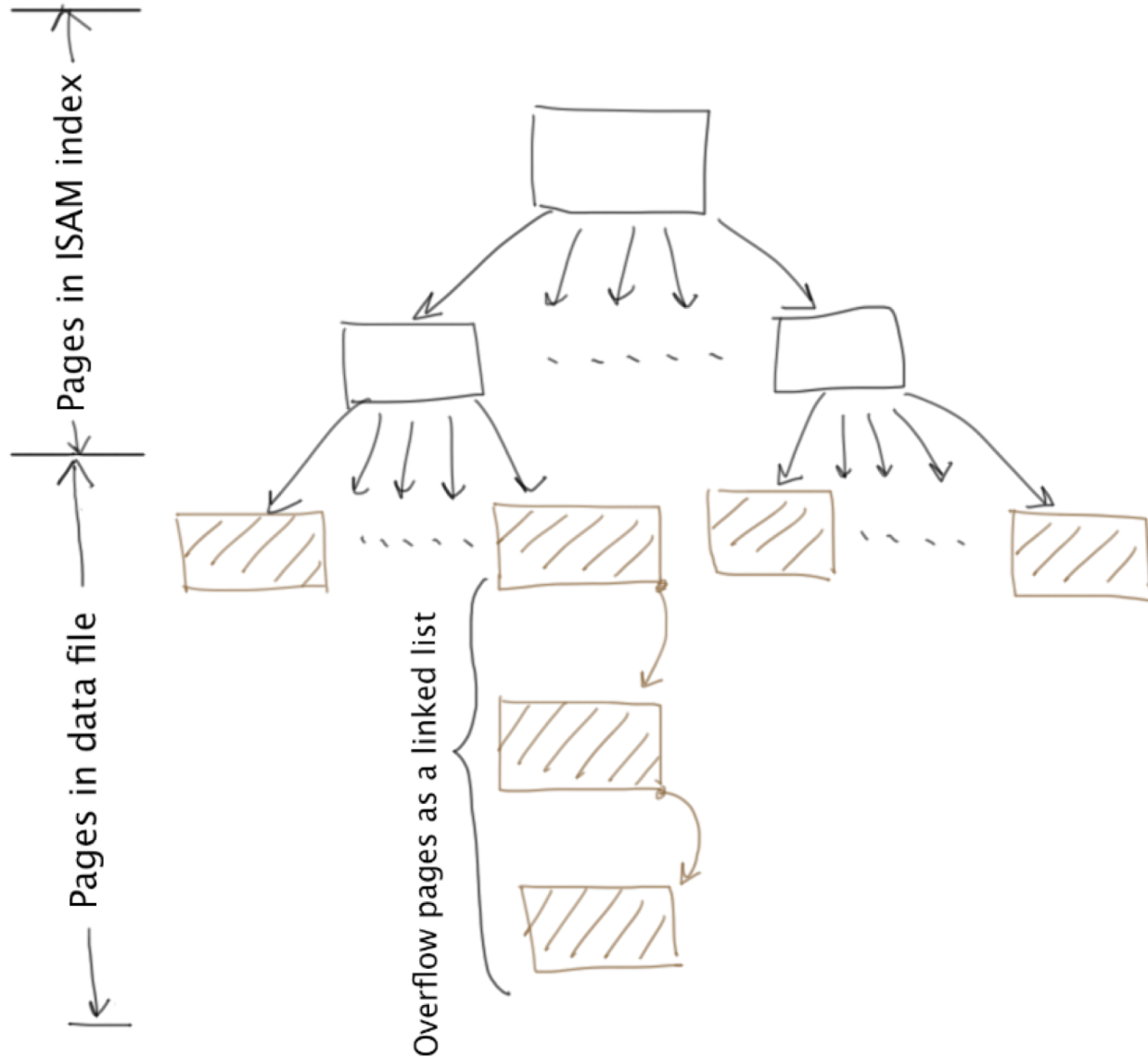
Limitation of Multilevel Sparse Indexes

- Multilevel sparse index can be applied successively. The number of index pages is reduced at each level, until at the top level, there should be only one index page.
- While the multilevel sparse index structure is efficient for record access, unfortunately, it cannot easily support record inserts and updates.
- Sparse indexes can only index sorted sequential file. So any record insert and update (on the sorting attributes) must maintain the ordering. Heap file only supports efficient record appends without any concerns on the ordering of the records.

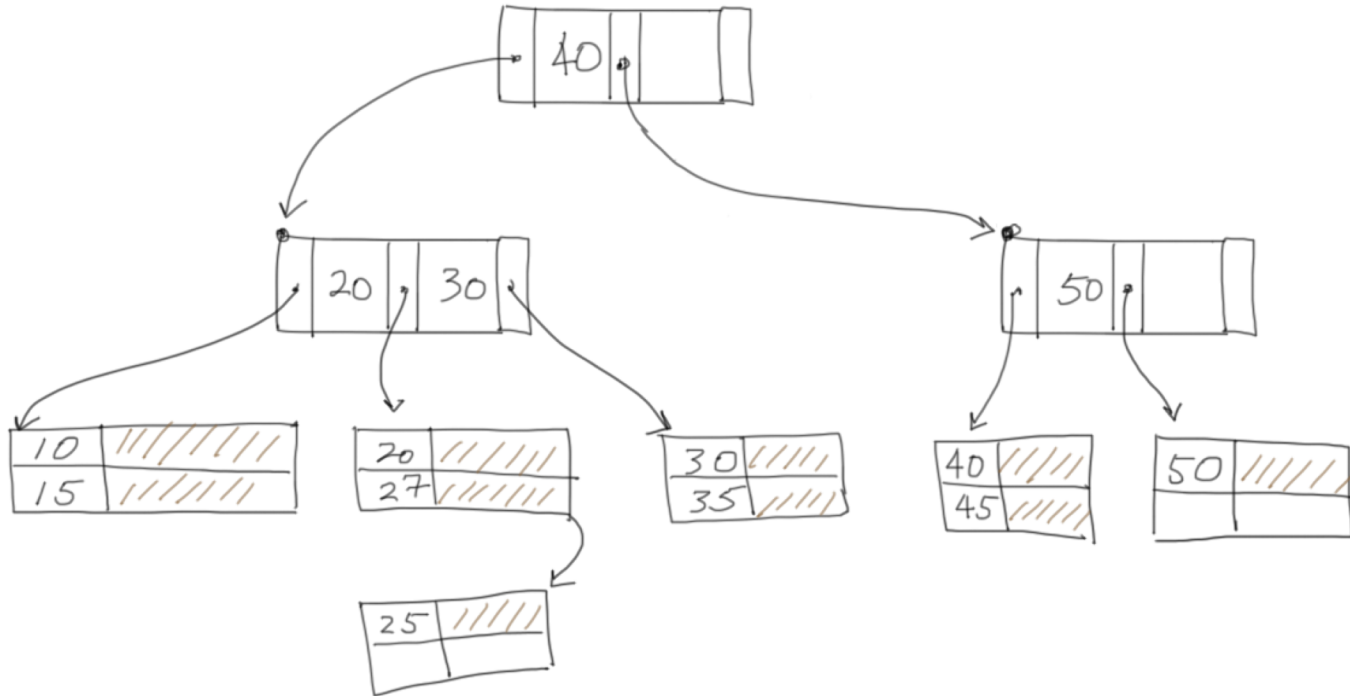
Indexed Sequential Access Method (ISAM)

- Overcomes the limitation of multilevel sparse index, and supports fast record insert and update.
- Each data page in the data file can have any number of overflow pages.
- Inserting a Record:
 - Locate the data page using the index
 - If the page or any of the overflow pages have free space, insert the new record in the that page
 - if all the overflow pages are also full, create a new overflow page to store the new record

Indexed Sequential Access Method (ISAM)



ISAM: Insert Example



The record (25, ...) is inserted *after* the index is built. Since there is *no* space available in the page(s) between the key 20 and 30, ISAM must create a new page which is appended into the appropriate linked list. The insertion of (25, ...) creates an overflow page.

Limitation of ISAM

- ISAM relaxes on the sortedness of records.
Consequently, the worst case performance for record retrieval becomes:

$$\log_2(N_{\text{index page}}) + \max(N_{\text{overflow page}})$$

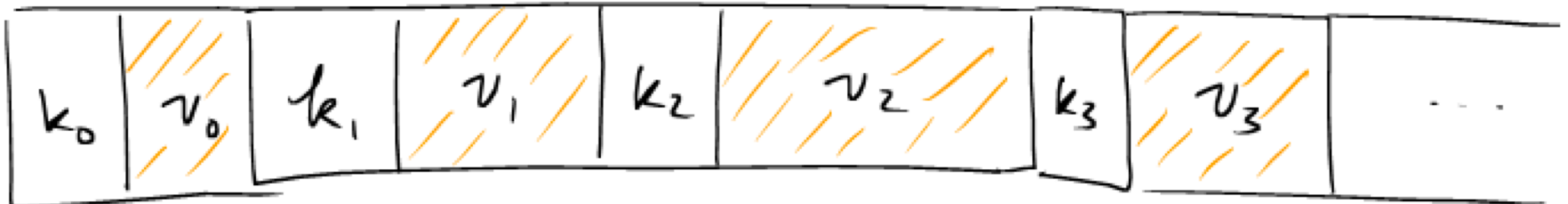
- Cause: the multilevel sparse indexes are static in the presence of data updates.
- Solution: B⁺-tree

B+tree

- A highly robust and popular data structure, which is an extension of ISAM.
- B+ tree offers:
 - fast record search
 - fast record traversal
 - maintaining sorted tree structure without overflow pages
- The key idea behind B+ tree is that it utilizes balanced sorted tree of page pointers, as opposed to just sorted tree in the case of ISAM.

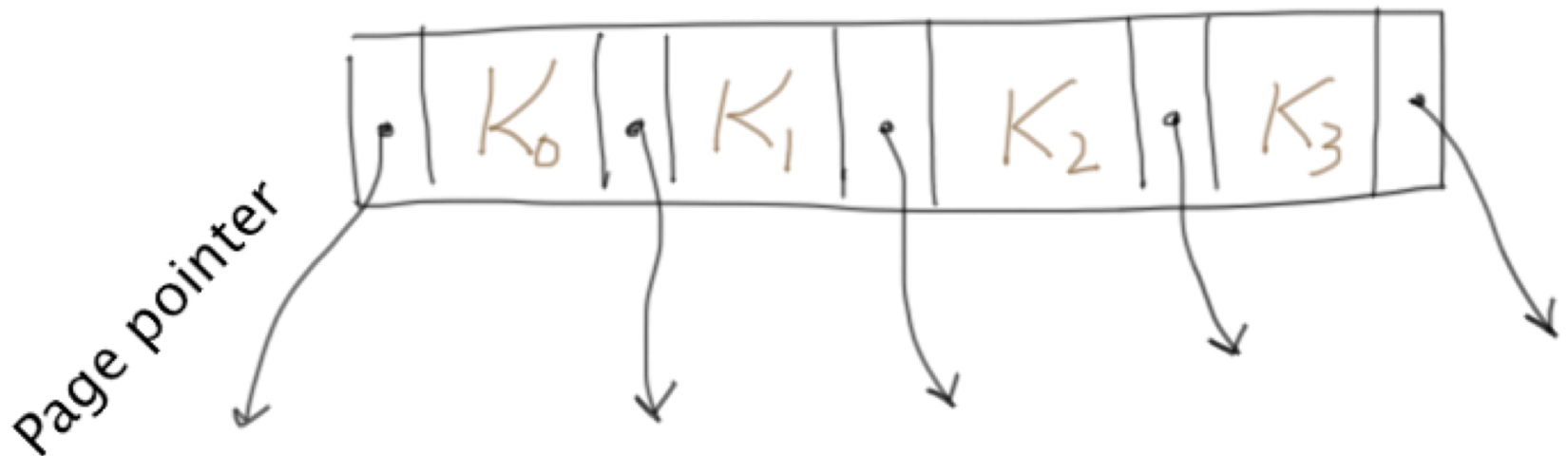
Definition of B⁺-tree

- A B+ tree is a tree whose nodes are pages on disk. We will distinguish the *leaf nodes* and the *interior nodes* of a B+ tree.
- Leaf Nodes: store the data entries in the form of (key, value). All leaf nodes are organized into a linked list of pages.



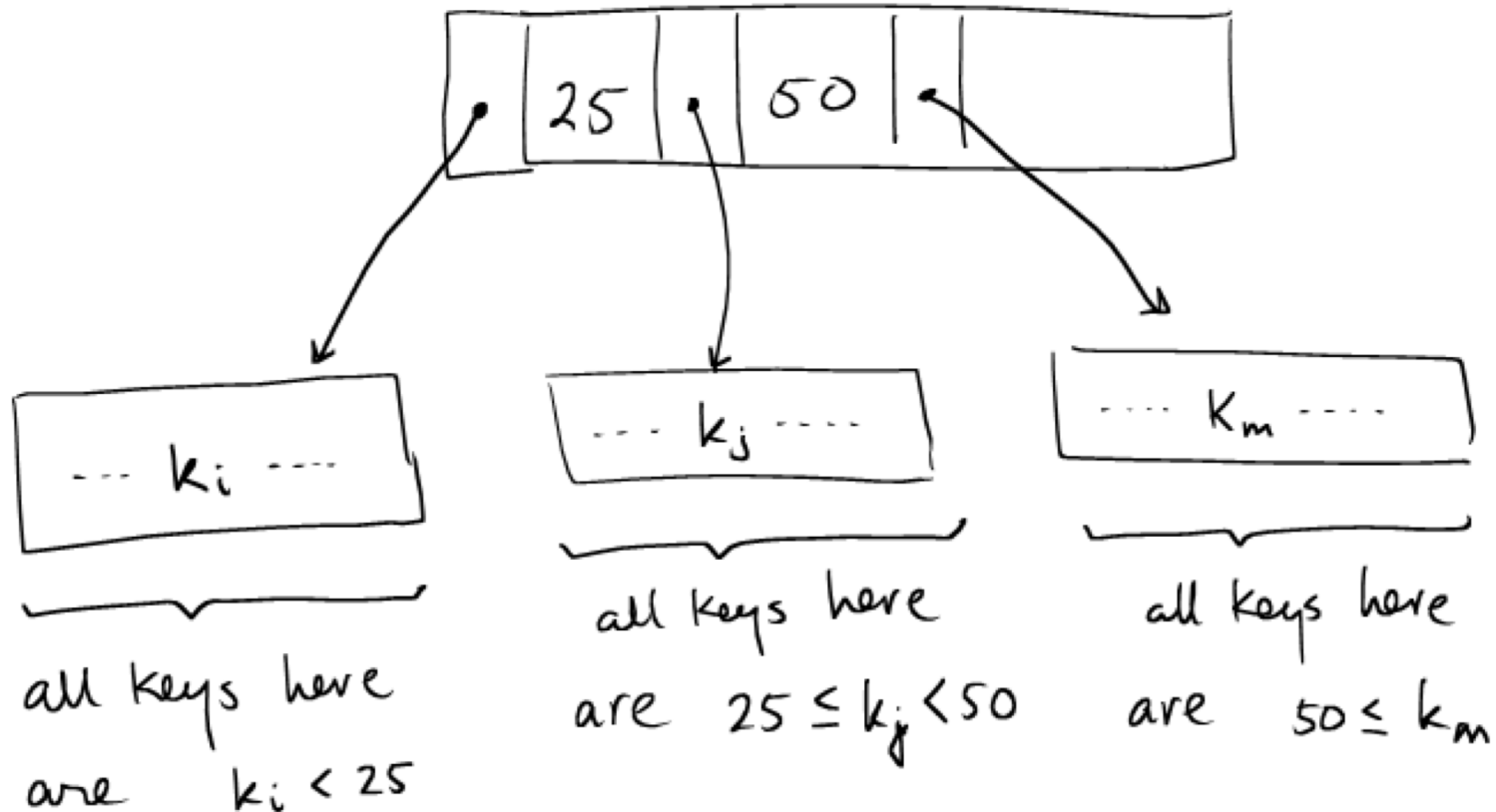
Definition of B⁺-tree

- Interior Nodes: form a tree structure, starting from a root node, to speed up lookup of *the* leaf node that contains a key of interest.
- The structure of an interior node can be visualized as a sequence of alternating page pointers and keys.



Constraints and Properties of B⁺-tree

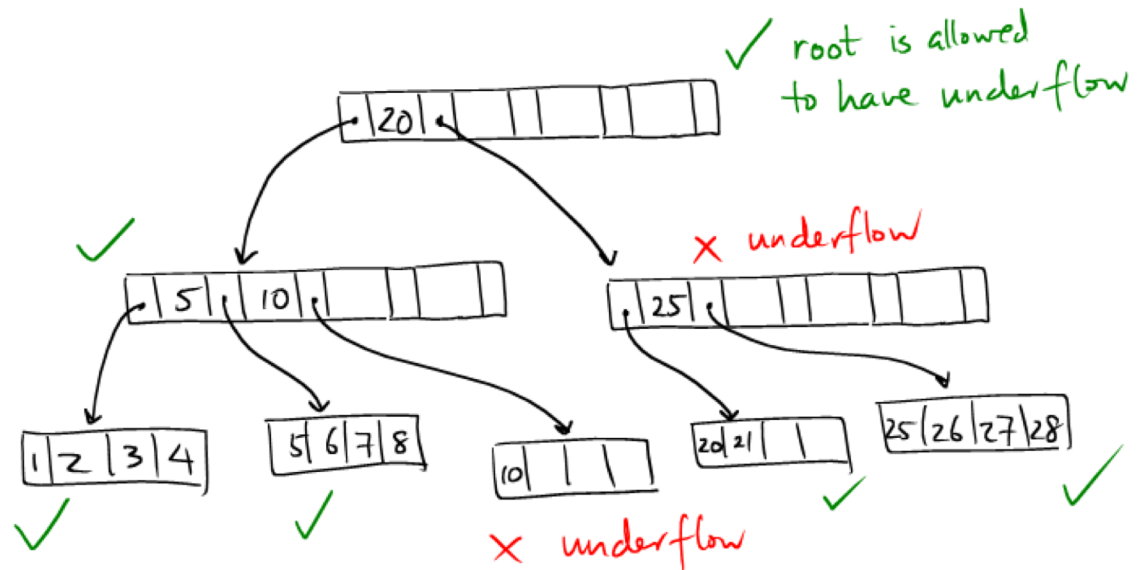
- Keys are sorted in the nodes
- Nodes are sorted by their keys (a sorted tree)



Constraints and Properties of B⁺-tree

- The tree is balanced: all paths from the root to the leaf nodes must be of equal length.
- The nodes are sufficiently filled: B+ tree specifies a percentage, known as the fill factor of the tree, that controls the *minimal* occupancy of all the non-root nodes.

- Example of a bad tree



B⁺-tree Search

The B+ tree search algorithm is a straight-forward tree traversal algorithm:

```
LeafNode search(Node p, Key key) {  
    if(p is LeafNode)  
        return root;  
    else {  
        if(key < p.keys[0])  
            return search(before(p.keys[0]), key);  
        else if(key > p.keys[-1])  
            return search(after(p.keys[-1]), key);  
        else {  
            let i be p.keys[i] <= key < p.keys[i+1]  
            return search(after(p.keys[i]), key)  
        }  
    }  
}
```

B⁺-tree Insertion

- Unlike the algorithm for insertion into a balanced sorted binary tree, the B+ tree insertion needs to deal with node overflows and underflows.
- The insertion algorithm starts with:
 - look for the right leaf for insertion
 - try to insert into the leaf node

```
InteriorNode insert_into_tree(InteriorNode root, Key newkey, Value val) {  
    LeafNode leaf = search(root, newkey);  
    return insert_into_node(leaf, newkey, val);  
}
```

B⁺-tree Insertion

- The insertion algorithm, *insert_into_node*, looks something like this:

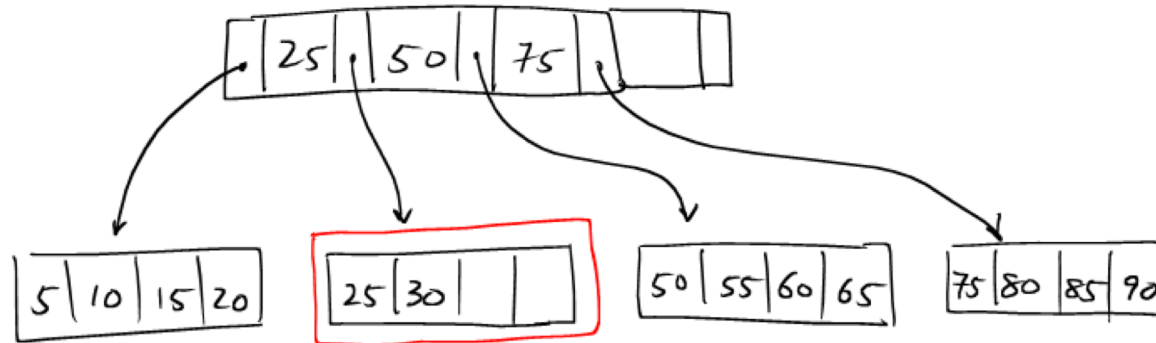
```
/**
 * Tries to inserts the (newkey/val) pair into
 * the node.
 *
 * If `target` is an interior node, then `val` must be a page pointer.
 */
InteriorNode insert_into_node(Node target, newkey, val)
{
    if( ... CASE 1 ... ) {
        /* handle CASE 1 */
    } else if( ... CASE 2 ... ) {
        /* handle CASE 2 */
    } else if( ... CASE 3 ... ) {
        /* handle CASE 2 */
    }
}
```

B⁺-tree Insertion – Case 1

The target node has available space for one more key:

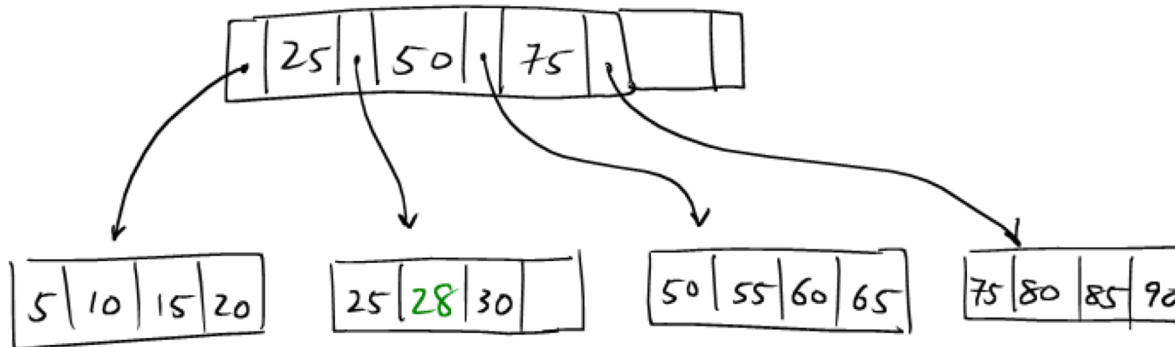
- The root of the B+ tree does not change.
- We have not discussed the disk I/O operations involved. Since all the node operations operate on one page at a time, the **buffer manager** can be used to provide the transparent virtual memory, as if all the nodes are stored in memory.

B⁺-tree Insertion – Case 1



Search (root, 28)

Leaf has vacancy $\Rightarrow$ CASE 1.



Insert (28, val)
into leaf node

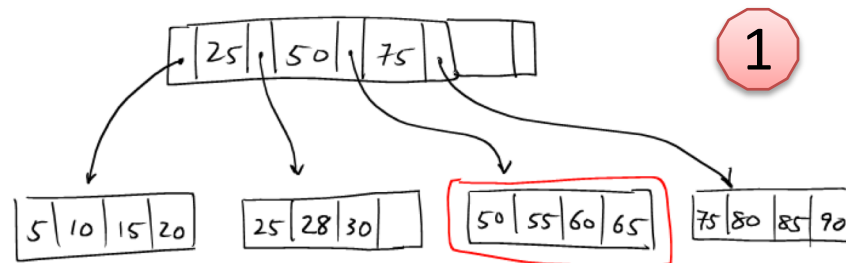
B⁺-tree Insertion – Case 2

The target node is full, but its parent has space for one more key:

- Create a new sibling node to target, call it `new_target`, and insert it *after* `target`.
- Distribute the data entries between `target` and `new_target`. From the assumption that `target` is full, we can safely assert that after distribution, there will be no underflow.
- `new_target` must be pointed to by $(k, p) = (\text{leaf2.keys}[0], \text{ADDRESS}[\text{leaf2}])$ in `PARENT[target]`. So (k, p) are to be inserted into `PARENT[leaf]`. By assumption, we know that `PARENT[leaf]` will not overflow.

B⁺-tree Insertion – Case 2

Insert an entry with key=70 in the previous example:

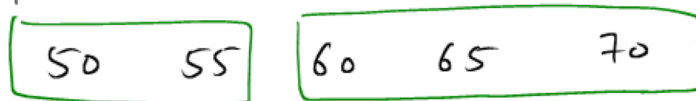


1

Search(root, 70)
node full, but
parent has space
⇒ CASE 2

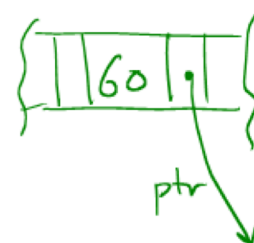
2

Keys are distributed

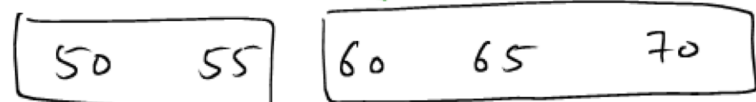


new node

3



Now we need to
insert (60, ptr)
into the parent.



new node

B⁺-tree Insertion – Case 3

The target node and its parent are both full:

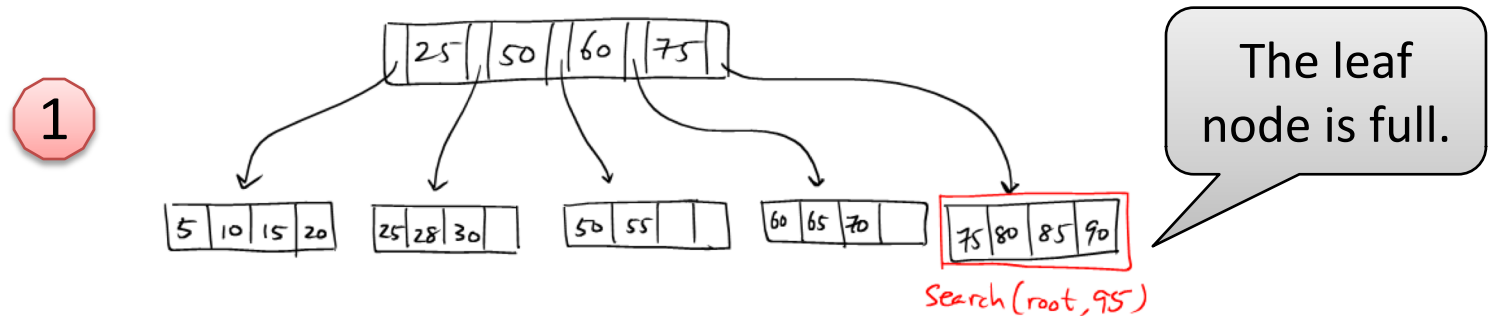
1. Create `new_target` node, and insert it after `target`.
2. Distribute data entries among `target` and `new_target`.

Now, let `k = leaf2.keys[0]` and `p = ADDRESS[leaf2]`. We need to insert `(k,p)` into `PARENT[leaf]`, which is full.

1. Let `target_parent = PARENT[target]`
2. Let `all_keys = sorted(target_parent.keys ∪ {k})`
3. Allocate new node `new_interior`
4. Let `i = floor(all_keys.size() / 2)`.
Let `middle_key = all_keys[i]`
5. Distribute `all_keys[0 .. i-1]` to `target_parent`, and
Distribute `all_keys[i+1 .. n]` to `new_interior`.
6. If `target_parent` is the root, then let `grandparent` be a newly allocated node.
Otherwise, `grandparent = PARENT[target_parent]`.
7. Recursively call:
`insert_into_node(grandparent, middle_key, ADDRESS[new_interior])`

B⁺-tree Insertion – Case 3

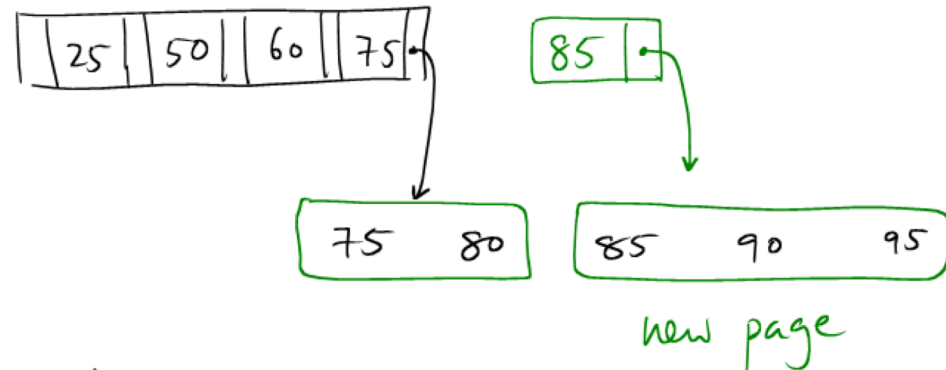
Insert an entry with key=90 in the previous example:



2

The parent interior node is also full.

Keys at the leaf distributed



This is case #3.

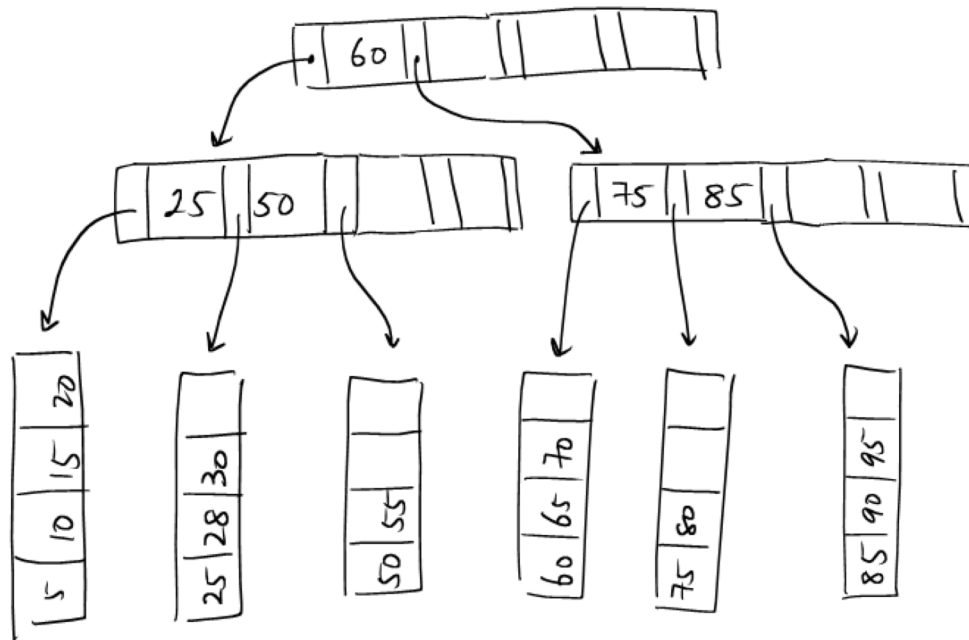
B⁺-tree Insertion – Case 3

all_keys:

25 50 60 75 85

└────────┘ ──┬── └────────┘

distribute middle new node,
to left key distribute to right



Other Things about B⁺-tree

- B+ tree also offers efficient deletion. The deletion algorithm is a mirror reversal of the insertion algorithm. During deletion, the algorithm tries to *merge* nodes to avoid underflow. If a merge took place, deletion recursively deletes the (Key, PagePointer) pair from the parent node.
- If the data entries are stored in a sequential file, sorted by a key, then one can very efficiently bulk load the sequential file into a B+ tree.
- B+ tree can be used as a sorting algorithm for disk based sorting.