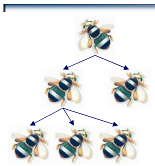
B trees, Sorting, indices



CS 2541: Database Systems
DBMS File Organization

Some slide content courtesy of Zack Ives & Raghu Ramakrishnan

File Organization: Recap

- How files are organized impacts performance of queries
- Concept of index records and indexing
  - Speeds up search
- Today: How to organize the index records & how to sort the file (on disk)

2

Data/File Organizations -- Speeding Operations over Data

- Three general data organization techniques:
  - Indexing
  - Sorting
  - Hashing
- There is also the notion of a "heap", but that is data *disorganization* (or storage) rather than organization…
  - But, it is easy to maintain in the face of insertions and deletionsdifficult to find things quickly.

3

Algorithms & 'Data' Structures for DBMS file organization

- B-trees: multi-level index
  - Most commonly used database index structure today
- Hash index
  - 'standard' hash table concept
- External sorting algorithms
  - Sorting data residing on disk
  - Time complexity measured in terms of disk read/write

4

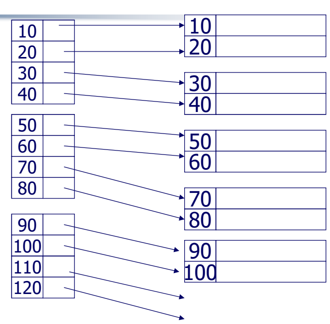
Recall: Indexing

- An *index* on a file speeds up selections on the *search key attributes* for the index (trade space for speed).
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.
  - Index record contains key k and a pointer (disk address) to the data record with that key value

Classes of Indices

- *Primary* vs. *secondary*:
- *Clustered* vs. *unclustered*: keu used to order records on the file and index key approximately same
- *Dense* vs. *Sparse*: dense has index entry per data value; sparse may "skip" some

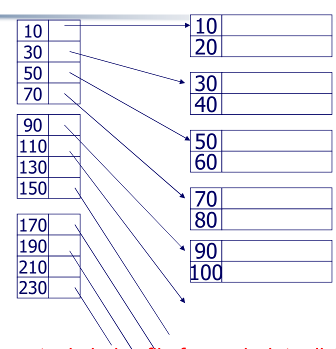
Dense Index Sequential File



Dense Index: one entry in index file for each data record

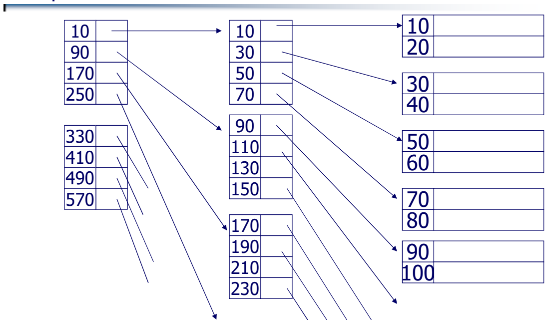
7

Sparse Index Sequential File



Sparse Index: one entry in index file for each data disk block when records on disk are ordered by search key

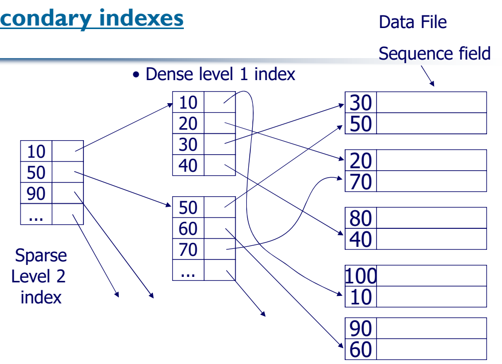
8



Two Level Index: 2nd level index points to 1st level index

9

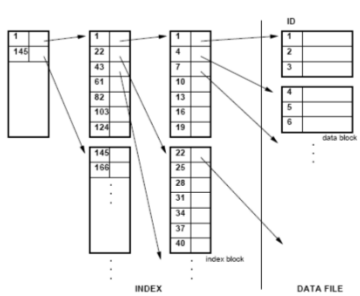
Secondary indexes



Two Level Index: 2nd level index points to 1st level index
2nd level is sparse index
1st level is dense index

10

Multilevel Index



11

Multilevel Indices and B-trees

12

B-Trees and Indexing

- multi-level indexes can improve search performance.
- One of the challenges in creating multi-level indexes is maintaining the index in the presence of inserts and deletes.
- We look at B-trees which are the most common form of index used in database systems today.

Review of B-tree concepts…

- Search in trees = O(h), h is height of tree
- Binary tree worst case height = O(n)
  - Tree may get unbalanced
- B-trees – are a class of **balanced trees**
- Forces all leaves to have same height
- Original motivation was search trees (not databases)

14

Balancing the tree…

- need to place some constraint that will force the tree to be balanced
  - This is accomplished by specifying the minimum and maximum number of entries at each node – the *order d* of tree
  - Alternately, can specify minimum and maximum number of children at each node – called the *degree m* of tree

15

B-trees

A ***B-tree*** is a search tree where each node has >= ***n*** data values and <= ***2n***, where we chose ***n*** for our particular tree.

- Each key in a node is stored in a sorted array.
  - key[0] is the first key, key[1] is second key,…,key[2*n*-1] is the 2*n*th key
  - key[0] < key[1] < key[2] < … < key[2*n*-1]
- There is also an array of pointers to children nodes:
  - child[0], child[1], child[2], …, child[2*n*]
  - **Recursive definition**: Each subtree pointed to by child[i] is also a B-tree.
- For any key[i]:
  - 1) key[i] > all entries in subtree pointed to by child[i]
  - 2) key[i] <= all entries in subtree pointed to by child[i+1]
- A node may not contain all key values.
  - # of children = # of keys +1

- A B-tree is **balanced** as every leaf has the same depth.

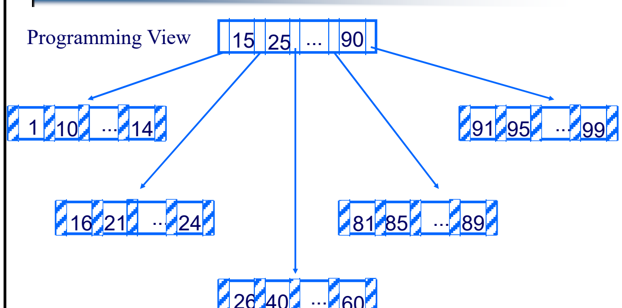
B-tree definition

- Every tree node of B-tree of order *d* (d >= 1) must have:
  - (except the root) must have at least *d* key values (entries) sorted inside the node
  - Cannot have more than *2d* key values
  - Has one more tree pointer than the number of key values; i.e., between *d+1* and *2d+1*
  - Is either a leaf or an internal node

B-tree Alternate definition

- Every tree node of B-tree of degree *m* (m >1) must have:
  - (except the root) must have at least *m-1* key values (entries) sorted inside the node
  - Cannot have more than *2m-1* key values
  - Has one more tree pointer than the number of key values; i.e., between *m* and *2m*
  - Is either a leaf or an internal node

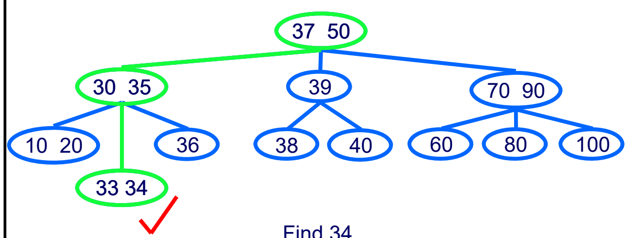
B-trees Example



Check your definitions

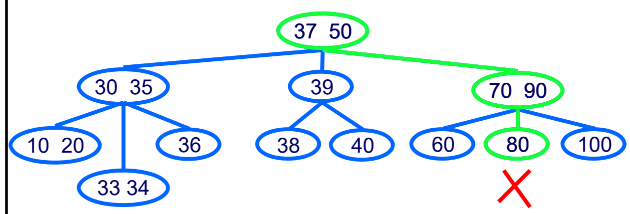
- If B tree has maximum of 10 keys per node, then what is the maximum number of children that a node can have ?
- If B tree has degree 4 then what is the maximum number of keys it can have at a node ?

Searching a B Tree: Example #1
B tree order 2 (1or 2 keys at node)



Find 34

Searching a BTree
Example #2



Find 82

Height of B-tree of n nodes ?

- Compute worst case height of B-tree of degree m (order m-1), with total *n* nodes and height *h*
  - Root at level 0 has only 1 node
  - Level 1 has 2 nodes, each has at least m children
  - Level 2 has at least 2m nodes, each has at least m children

Height of B-tree of n nodes ?

- Compute worst case height of B-tree of degree m (order m-1), with total *n* nodes and height *h*
  - Root at level 0 has only 1 node
  - Level 1 has 2 nodes, each has at least m children
  - Level 2 has at least 2m nodes, each has at least m children
  - Level 3 has at least $2m^2$ nodes, each has at least m children
  - ….
  - Level *h* has $2m^{(h-1)}$ nodes
- Therefore $n = 1 + 2 + 2m + 2m^2 + \ldots + 2m^{(h-1)}$
  - $n = 1 + 2(m^h - 1/m-1)$
- Therefore, $h = O(\log_m n)$

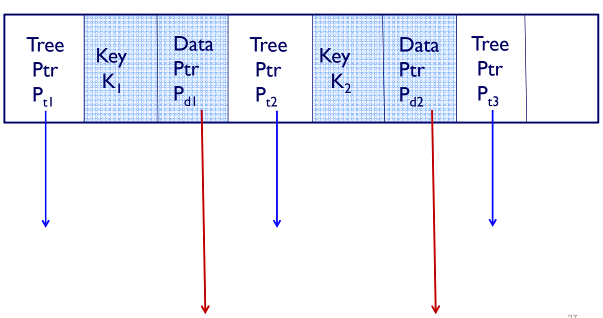
B-trees as External Data Structures

- Now that we have seen how a B-tree works as a data structure – how can it be used for an index.
- A regular B-tree can be used as an index by:
  - Each node in the B-tree stores not only keys, but also a record pointer for each key to the actual data being stored.
    - Could also potentially store the record in the B-tree node itself.
  - To find the data you want, search the B-tree using the key, and then use the pointer to retrieve the data.
    - No additional disk access is required if the record is stored in the node.

B-tree nodes & Index records

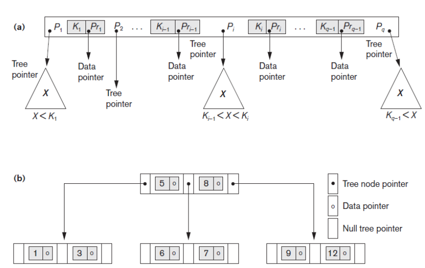
- B-tree is collection of blocks/nodes that contain
  - Search-key (index) values
  - Data pointers to data records
  - Tree pointers to next/children node
  - Some info local to block
- Each B-tree node resides on one disk block
  - When tree is traversed, relevant nodes/blocks are brought into main memory
- Given this description, how might we calculate the best B-tree ***order***.
  - Depends on disk block and record size.
  - We want a node to occupy an entire block.

B-tree node



B-tree structure

(a) Node in B-tree with q-1 keys; (b) B tree with max 2 at node



Size of B-tree node

- What is the maximum number of bytes needed by a B-tree node of order d ?
  - Tree pointers, data pointers, key fields, byte specific info
- What is the maximum number of bytes you can use for the node ?
  - Size of the disk block !
  - One node fits in one disk block
- What is maximum no of entries of B-tree if key requires 5 bytes, and tree and data pointer need 10 bytes and disk block size=100 bytes ?

29

Picking the order of a B-tree

- Best "packing": when entire node fits into one disk block with minimum space wasted
- Assume P_d bytes needed to specify address of data pointer
- Assume P_t bytes needed for address of tree pointer
- Assume key field requires K bytes
- Assume disk block size = B bytes
- Assume x bytes needed for block-specific information

30

Order of B-tree

- Order d tree has at most 2d+1 tree pointers, 2d data pointers and 2d key values
- $x + 2d*(K + P_d) + (2d+1)*P_t <= B$

31

Degree of B-tree

- Degree m tree has at most 2m tree pointers, 2m -1 data pointers and 2m-1 key values
- $x + (2m - 1)*(K + P_d) + (2m)*P_t <= B$

32

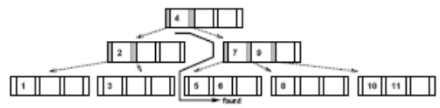
Example: Calculating order of B-tree

- Disk block size =512 bytes, Key is varchar(20), and 32 bit (4 byte) addresses
- B=512 bytes, x=2, K=20, $P_t = P_d = 4$

- 2+ 2d(20+4) + (2d+1)4 <= 512
  - 56d + 6 <= 512
  - d <= 9

Search in B-tree

Search for key=6:
find node with the key=6
follow record pointer to fetch the data record from disk



Search in B-tree

- In-order search

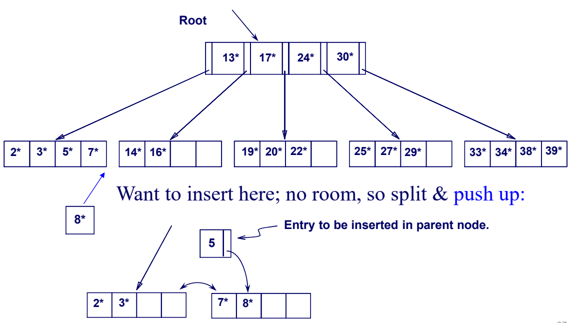
- Time complexity ?
  - Height of tree = log (N)

Inserting Data into a B Tree

- Find correct leaf L : search O(log N)
- 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, push up middle key.
  - Insert index entry pointing to L2 into parent of L.
- This can happen recursively

- Splits "grow" tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.

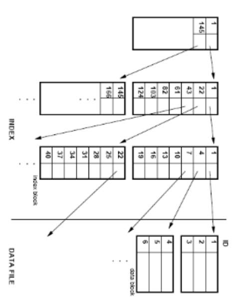
Inserting 8 Example: d=2



B-tree and Multilevel Index

- Same ? Or Different ?

Multilevel

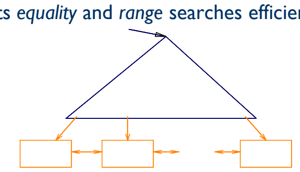


Key differences

- In multi-level index:
  - Data records at level 0 and index records at all other levels (contain key value but no data pointer)
  - The lowest level index records are ordered
    - Leaves form a linked list
- In B tree
  - Nodes contain both data pointers and tree (index) pointers
  - To get a list of sequential records, we have to search multiple levels of the tree

B+ Tree: The DB World's Favorite Index

- Insert/delete at log $_F$ N cost
  - (F = fanout, N = # leaf pages)
  - Keep tree *height-balanced*
- Minimum 50% occupancy (except for root).
- Each node contains **d** <= *m* <= 2**d** entries.
  d is called the *order* of the tree.
- Supports *equality* and *range* searches efficiently.



B+ Trees – Definition

- Similar structure to B-tree
- Distinguish between internal nodes and leaf nodes
  - Leaf nodes contain all search keys inserted into tree and connected by linked list in sorted order
  - Each leaf node:
    - Has no tree pointers
    - Has only search keys and data pointers
    - Has linked list pointer to next leaf node
  - Each internal node
    - Has search keys
    - Has tree pointers
    - Has no data pointers
  - Search keys in internal nodes used only for navigation

42

B+ trees

- Search keys in internal nodes are repeated in leaves
  - Similar to multilevel index
- Every search key does not occur as internal value

- To get the data record, we have to get to the leaf level
  - All searches will take O(h) where h is height of tree

43

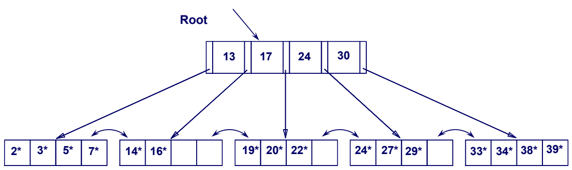
B+ trees: Definition

- B+-tree of order d
  - Each internal node or leaf node must contain at least d keys
  - Each internal or leaf node can contain at most 2d keys
  - Each internal node can contain at most 2d+1 pointers to next node in tree
  - Each leaf node must contain as many data pointers as there are keys in the node
  - Each leaf node has pointer to next leaf node in linked list

44

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24* ...



➢ *Based on the search for 15*, we know it is not in the tree!*

Picking the order of a B+-tree

- Same logic as for B-tree:
- Best "packing": when entire node fits into one disk block with minimum space wasted
- Assume P_t bytes needed to specify address of data/tree pointer
- Assume key field requires K bytes
- Assume disk block size = B bytes
- Assume x bytes needed for block-specific information

Size of B+-tree node

- What is the maximum number of bytes needed by a B+-tree node of order d ?
  - pointers, key fields, byte specific info
- What is the maximum number of bytes you can use for the node ?

Order of B-tree

- Order d tree has at most 2d key values and 2d+1 pointers (data pointers and linked list pointer for leaf node, and tree pointers for internal node)

- $x + 2d*(K) + (2d+1)*P_t <= B$

Example

- B=512 bytes, x=2, K=20, $P_t = 4$
- 2+ 2d(20) + (2d+1)4 <= 512
  - 48d + 6 <= 512
  - d <= 10.54
  - Higher than degree of B-tree!

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Inserting Data into a B+ Tree

- Find correct leaf L.
- Put data entry onto 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 index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.

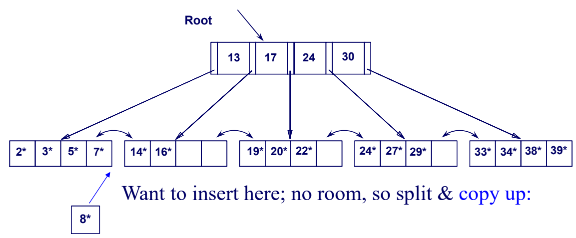
B+ tree insertion Algo: Outline

- Input (Key, Pointer to Data) (K,P), B+ tree of order N (max 2N values per node)
- Insert (T, K,P)
- Search Tree to find Leaf L_i to insert (K,P)
  - Insert into Leaf L_i
  - If number of entries in L_i =< 2N then done
  - Else Split(L_i)
    - Find key value of N+1 entry (i.e., median) Z_i in leaf
    - Split L_i into two leaves L_{i1} and L_{i2}
      - L_{i1} contains all entries less than Z_i
      - L_{i2} contains all entries greater than or equal to Z_i
      - Create pointer from L_{i1} to L_{i2}
      - Create pointer P_{i1} pointing to L_{i1} and P_{i2} pointing to L_{i2}
      - If L_i was root node, then create new root and insert Z_i into new root and stop.
      - Else Insert (T,Z_i)

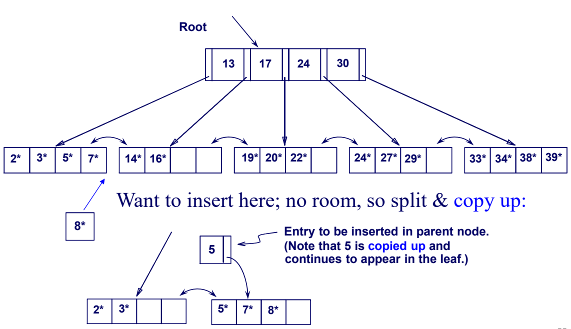
Inserting 8* into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Recall that all data items are in leaves, and partition values for keys are in intermediate nodes
  Note difference between *copy-up* and *push-up*.

Inserting 8* Example: Order 2 B+ tre

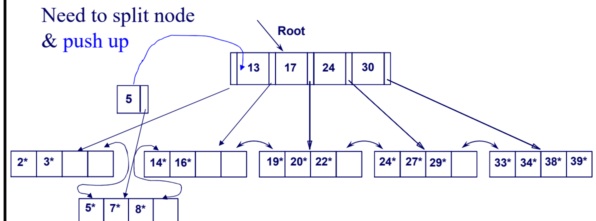


Inserting 8* Example: Copy up



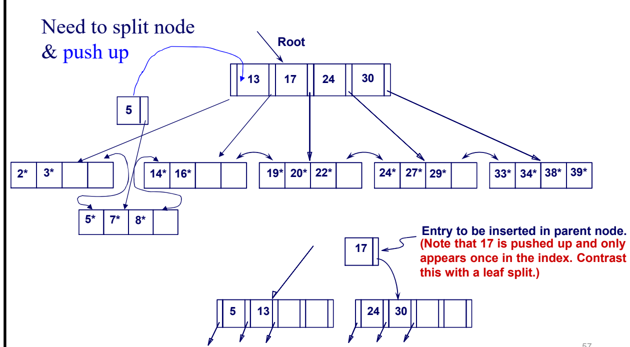
Inserting 8* Example: Push up

Need to split node
& push up



Problem: Root node
is also full…so split root!

Inserting 8* Example: Push up



57

Deleting Data from a B+ Tree

- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has only d-1 entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.
- …details in notes

B+ Tree Summary

B+ tree and other indices ideal for range searches, good for equality searches.

- Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
- High fanout (F) means depth rarely more than 3 or 4.
- Almost always better than maintaining a sorted file.
- Typically, 67% occupancy on average.
- Note: Order (d) concept replaced by physical space criterion in practice ("at least half-full").
  - Records may be variable sized
  - Index pages typically hold more entries than leaves

Sorting Algorithms

- Quicksort, Heapsort
- Complexity ?
- Use the same algorithm to sort our data files?

60

External Sorting

- Data to be sorted = data file stored on disk
- Sorted file must be stored back on disk

- External sorting problem:
  - Data file too large to fit in memory
  - Data must be sorted in pieces
  - Data usually heap size
  - Desired result: sorted file (sorted on some key field)

61

Why Sorting ?

- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing *gpa* order
- *Sort-merge* join algorithm involves sorting.
- *Internal sorting*
  - *Quicksort, heapsort, etc.*
- Sorting Problem considered here: sort data much larger than main memory
  - Example: sort 1Gb of data with 1Mb of RAM.
    - Cannot hold all data in main memory – cannot use internal sort

Mergesort: Recall

- Merge two sorted lists
  - (2,5,10,19)
  - (4,6,7,20)

- If we have two sorted files of length n/2 then binary merge gives sorted file of length n

63

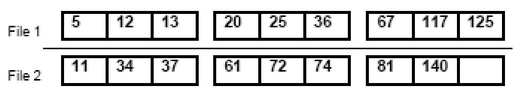
External Mergesort

- File is set of pages/blocks

- Read one block at a time from disk into main memory
  - Assume 2 pages to hold a page from each input file and 1 page for output sorted block

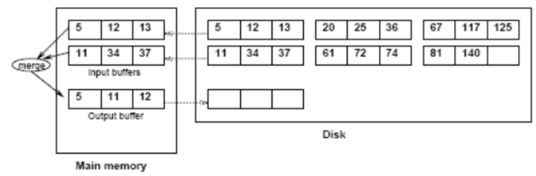
64

External Mergesort

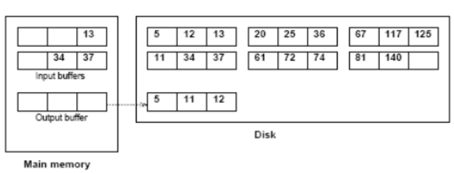


65

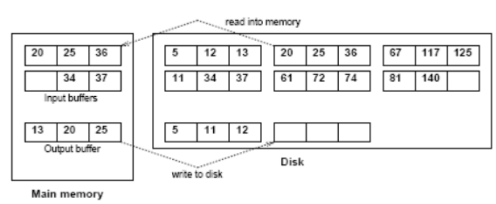
External Mergesort



66

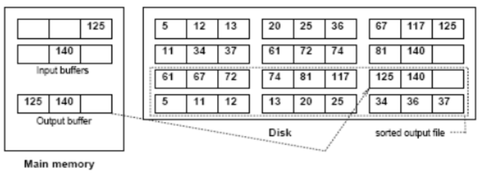


67



68

Finally…



Observations

- How many disk I/Os ? (how many read and write operations)
- Length of sorted file vs length of input file(s) ?

External Sorting: Key ideas

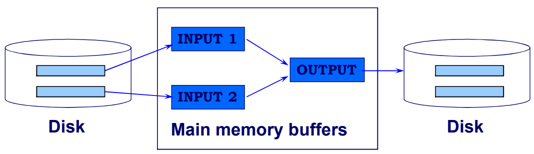
- Individual blocks/pages can be easily sorted
  - Read them into memory and use internal sort algorithm
- Create *runs*
  - A run is a group/set of sorted blocks (i.e., sorted piece of a file)
  - Runs can be created by merging data from several blocks
- Merge shorter runs into longer runs
  - Merge runs to get sorted file

External Mergesort

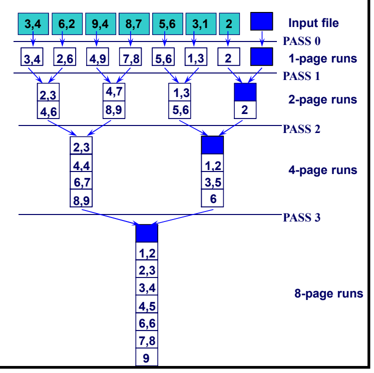
- What is unsorted input ?
  - Unsorted file of n blocks is n/2 groups of 2 blocks each
    - Each group has two blocks to be sorted
    - Sorting each group gives n/2 groups of runs, each run is 2 blocks
- What if we merge n/2 groups, each group of 2 sorted blocks?
  - n/4 groups, each of run 4 sorted blocks
  - How many disk IOs ?
- What is we merge n/4 groups, each group is run of size 4 (i.e., 4 sorted blocks
  - n/8 groups, each of size 8 blocks
  - How many disk IOs?
- …..
- What if we merge 2 groups, each group is run of size n/2
  - We get sorted file of n blocks
  - How many disk IOs ?
  - How many phases to get to this last step ?

External Sorting

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.



Two-Way External Merge Sort

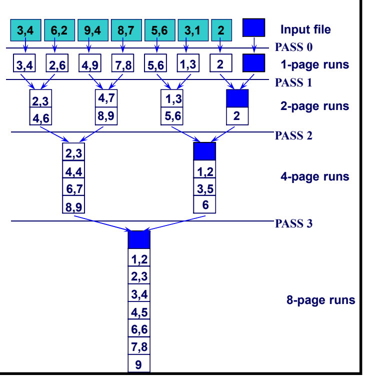


Two-Way External Merge Sort

- Each pass we read, write each page in file.
- N pages in the file => the number of passes

  $$= \lceil \log_2 N \rceil + 1$$
- Total cost is:

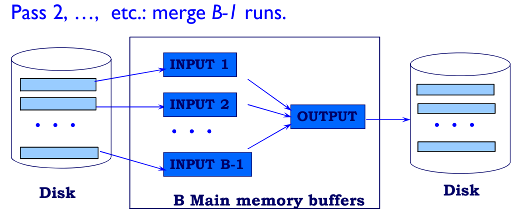
  $$2N\left(\lceil \log_2 N \rceil + 1\right)$$
- *Idea:* ***Divide and conquer:*** sort subfiles and merge



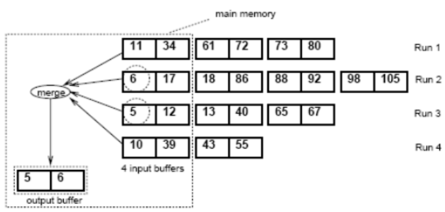
General External Merge Sort

➢ *How can we utilize more than 3 buffer pages?*

- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use *B* buffer pages. Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
  - Pass 2, …, etc.: merge *B-1* runs.



M-way External Mergesort



M-way Mergesort

- Divide file into groups of M blocks each
- Merge each group into a sorted run: N/M runs
- Create groups of N/M runs, merge each group into single run
- Example: N=400 block file, M=4 input buffers
  - Phase 1: create groups of 4 blocks
    - 100 groups of 4 blocks each = 100 runs of 4 blocks each
  - Phase 2: create groups of 4 runs; 100/4=25 groups
    - 25 groups of 16 blocks each = 25 runs of 16 blocks each
  - Phase 3: create groups of 32 runs; 25/4= 7 groups
    - 7 groups of upto 32 blocks each = 7 runs of 32 blocks
  - Phase 4: groups of 64 runs; 7/4= 2 groups
    - 2 groups, upto 64 blocks = 2 groups of 64 blocks
  - Phase 5: group of 128 runs; 2/4=1 group
    - 1 group, upto 128 blocks; 4-way merge gives 400 blocks

Cost of External Merge Sort

- Number of passes: $1+\lceil \log_M \lceil N \rceil \rceil$
- Cost = 2N * (# of passes)