CMR
INSTITUTE OF
TECHNOLOGY

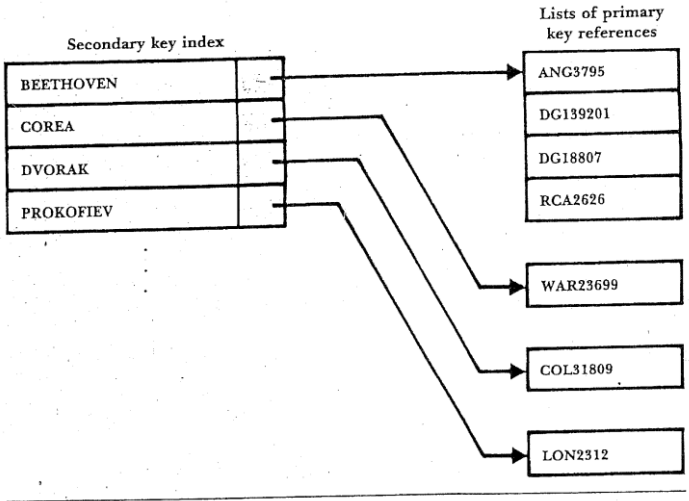
USN | | | | | | | | | | |

Second Internal Test

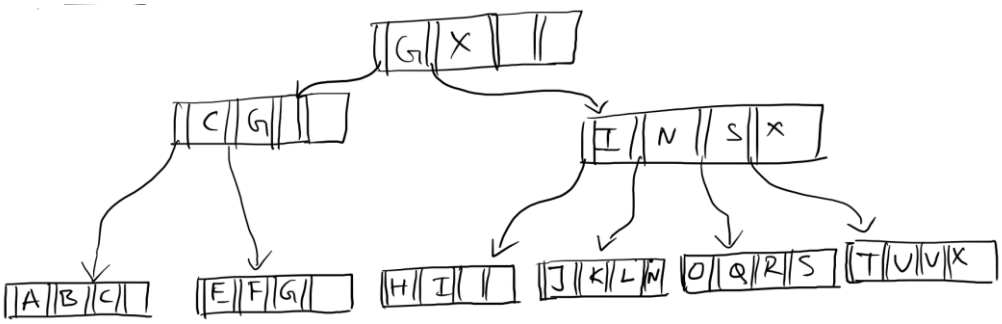
Sub:	File Structures							Code :	18IS61/ 17IS62
Date:	22/06/ 2021	Duration:	90 mins	Max Marks:	50	Sem:	VI A,B&C	Branch:	ISE

Answer Any **FIVE FULL** Questions

		Marks	OBE	
			CO	RBT
1	What is Indexing? List and explain the different operations required maintain an indexed file.	10	CO3	L2
	: Indexing is a structure containing a set of entries, each consisting of a key field and a reference field, which is used to locate records in a data file. It helps in faster access of records in a file if the size of the index file is small. Since index files are sorted on key field, binary search can be applied to find the presence of the key and use the reference field for performing a direct access to locate the record in single seek. Operations required to maintain an indexed file: 1) Creating the data and index files. 2) Loading the index file to memory 3) Rewriting the index file from memory 4) Record addition 5) Record deletion 6) Record updating			
2	What is secondary indexing? What are the limitations of secondary indexing? Explain the solution by using 'linking the reference' techniques.	10	CO3	L3
	Secondary Indexing: In secondary indexing the secondary key is related to a primary key which then will point to the actual byte offset. • When a secondary index is used, adding a record involves updating the data file, the primary index and the secondary index. • The secondary index update is similar to the primary index update. Secondary keys are entered in canonical form (all capitals). • The upper- and lower- case form must be obtained from the data file. As well, because of the length restriction on keys, secondary keys may sometimes be truncated. • The secondary index may contain duplicate (the primary index couldn't). Ex : Secondary index file of all students with name as secondary key • Secondary indexes lead to two difficulties: – The index file has to be rearranged every time a new record is added to the file.			

	– If there are duplicate secondary keys, the secondary key field is repeated for each entry ==> Space is wasted. Linking the reference: each secondary key points to a different list of primary key references. Each of these lists could grow to be as long as it needs to be and no space would be lost to internal fragmentation.  Figure 7.12 Conceptual view of the primary key reference fields as a series of lists. Improved revision of the composer index *Secondary Index file* 0 BEETHOVEN 3 1 COREA 2 2 DVORAK 7 3 PROKOFIEV 10 4 RIMSKY-KORSAKOV 6 5 SPRINGSTEEN 4 6 SWEET HONEY IN THE R 9 *Label ID List file* 0 LON2312 −1 1 RCA2626 −1 2 WAR23699 −1 3 ANG3795 8 4 COL38358 −1 5 DG18807 1 6 MER75016 −1 7 COL31809 −1 8 DG139201 5 9 FF245 −1 10 ANG36193 0 Figure 7.13 Secondary key index referencing linked lists of primary key references.			
3	Explain the object oriented modeling for implementing consequential processing. Write and explain an algorithm for consequential match and illustrate the same its functioning for the following lists of names: List 1: Ajay, Amith, Arun, Bhavesh, Bhuvan, Dhruv, John, List 2: Ajay, Asha, Bhavesh, John, Kiran, Lakshmi, Siri	10	CO2	L3
	Co-sequential Processing: Cosequential operations involve the coordinated processing of two or more sequential lists to produce a single output list. **Components of model:** • **Initializing:** we need to arrange things so that the procedure gets going properly. • **Getting and accessing the next list item:** we need simple methods to do so.			

	• **Synchronizing:** we have to make sure that the current item from one list is never so far ahead of the current item on the other that a match will be missed.
	• **Handling end-of-file conditions:** Halt the program on reaching end of list1 or list2
	• **Recognizing Errors:** Duplicate items or items out of sequence.
	
	`int Match (char * List1Name, char * List2Name, char * OutputListName)`
	`{`
	`int MoreItems;// true if items remain in both of the lists`
	`// initialize input and output lists`
	`InitializeList (1, List1Name);// initialize List 1`
	`InitializeList (2, List2Name);// initialize List 2`
	`InitializeOutput(OutputListName);`
	`// get first item from both lists`
	`MoreItems = NextItemInList(1) && NextItemInList(2);`
	`while (MoreItems){// loop until no items in one of the lists`
	`if (Item(1) < Item(2))`
	`MoreItems = NextItemInList(1);`
	`else if (Item(1) == Item(2)) // Item1 == Item2`
	`{`
	`ProcessItem (1); // match found`
	`MoreItems = NextItemInList(1) && NextItemInList(2);`
	`}`
	`else // Item(1) > Item(2)`
	`MoreItems = NextItemInList(2);`
	`}`
	`FinishUp();`
	`return 1;`
	`}`
	Figure 8.2 Cosequential match function based on a single loop.
	
	Illustrate the algorithm for the given example:			

4	Construct a B Tree of order 4 for the following sequence of keys: C G J X N S U O A E B H I F K L Q R T V	10	CO3	L3
	Refer class notes for steps : **Final solution**  			
5	Explain K-way merge technique for merging large number of lists. Illustrate concept of selection tree using 8 lists of sorted numbers	10	CO2	L3
	• Merge *k* sequential lists • An array of *k* lists and			

	• An array of k index values corresponding to the current element in each of the k lists, respectively. • Main loop of the K-Way Merge algorithm: Step 1: Find the index of the minimum current item, *minItem* Step 2: Process *minItem*(output it to the output list) Step 3: For *i*=0 until *i*=*k*-1 (in increments of 1) If the current item of list *i* is equal to *minItem* then advance list *i* (read the next item in list *i*). Step 4: Go back to step 1 • This algorithm works well if k < 8. Otherwise, the number of comparisons needed to find the minimum value each step of the way is very large. • Instead, it is easier to use a selection tree which allows us to determine a minimum key value more quickly. Merging k lists using this method is related to $\log_2 k$ (the depth of the selection tree) rather than to k.			

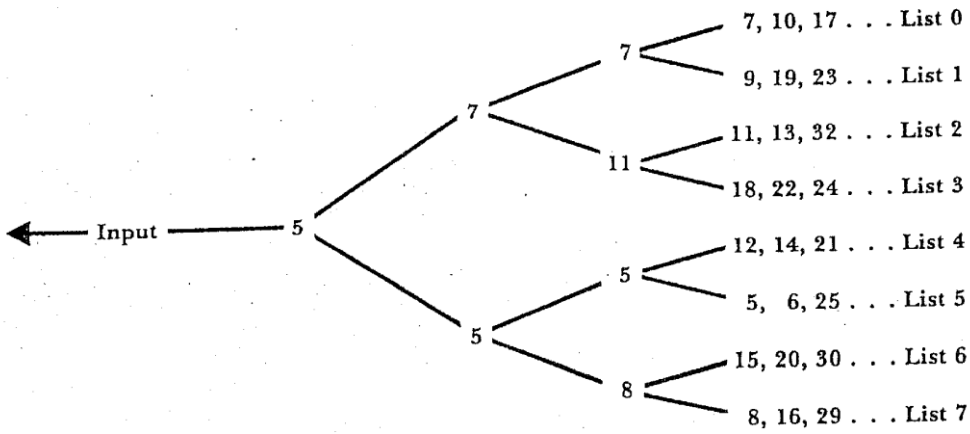


Figure 8.15 Use of a selection tree to assist in the selection of a key with minimum value in a *K*-way merge.

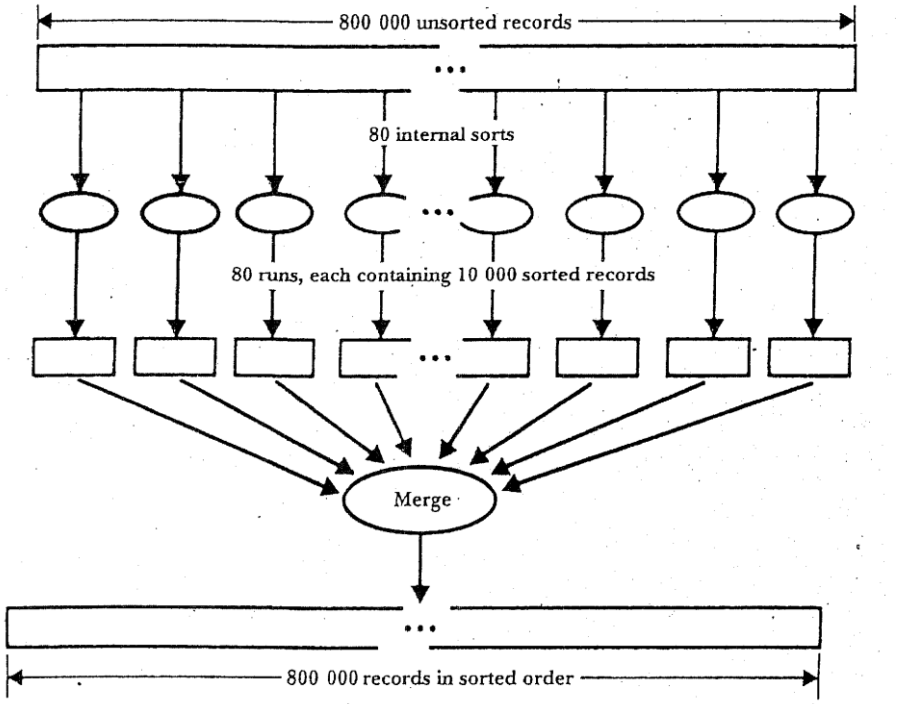
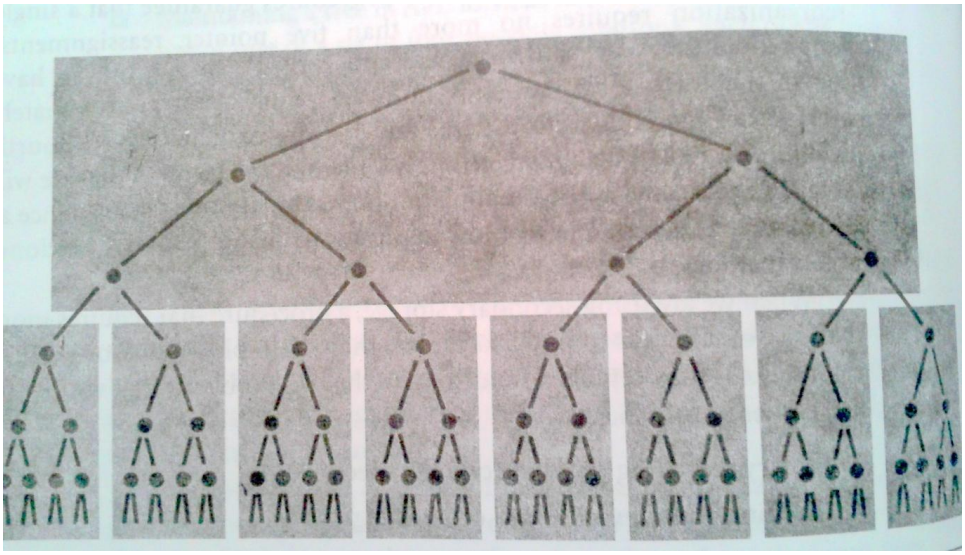


Figure 8.21 Sorting through the creation of runs (sorted subfiles) and subsequent merging of runs.

6	Explain the following Terms: 1. Paged Binary Trees 2. Heap Sort with replacement selection	10	CO2, CO3	L2
	1. Paged Binary Tree Disk utilization of a binary search tree is extremely inefficient. That is when a node of binary search tree is read, there are only three useful information. They are the key value, the address of the left and right sub trees. Each disk read produces a minimum of single page. The paged binary tree attempts to address this problem by locating multiple binary nodes on the same disk page.			

Paging divides a binary tree in to pages and then storing each page in a block of contiguous locations on disk, so that reduces the number of seeks associated with search.

- Worst case search with a balanced paged binary tree with page size M is logM + 1 (N + 1) compares.
- Balancing a paged binary tree can involve rotations across pages, involving physical movement of nodes.



The Problem with Paged Binary Trees

➢ Only valid when we have the entire set of keys in hand before the tree is built.

➢ Problems due to out of balance. Which requires accessing nodes on other pages. Rotation to balance the tree becomes costly.

2. HEAP sort with replacement selection

Heap sort with replacement selection is used for creating longer runs during sorting of large files using merging.

Replacement Selection Procedure:

1. Read a collection of records and sort them using heapsort. The resulting heap is called the **primary heap.**
2. Instead of writing the entire primary heap in sorted order, write only the record whose key has the lowest value.
3. Bring in a new record and compare the values of its key with that of the key that has just been output.
    a. If the new key value is higher, insert the new record into its proper place in the primary heap along with the other records that are being selected for output.
    b. If the new record's key value is lower, **place** the record in a secondary heap of records with key values smaller than those already written.
4. Repeat Step 3 as long as there are records left in the primary heap and there are records to be read. When the primary heap is empty, make the secondary heap into the primary heap and repeat steps 2 and 3.