CMR
INSTITUTE OF
TECHNOLOGY

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

Second Internal Test

Sub:	File Structures							Code:	15IS62
Date:	16/ 04 / 2018	Duration:	90 mins	Max Marks:	50	Sem:	VI	Branch:	ISE

Answer Any **FIVE FULL** Questions

Scheme and solution	Marks	OBE CO	OBE RBT
1 (a) Define Indexing and its importance in File Structures. List and explain the different operations required maintain an indexed file	[10]	CO2	L2
Soln 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 Explain each operation in detail.			
2 (a) Write and explain the algorithm to match two lists consisting of names using co-sequential processing, with a suitable example.	[10]	CO2	L2
Soln Cosequential operations involve the coordinated processing of two or more sequential lists to produce a single output list. **Matching names in two lists: Matters to Consider:** • **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.

Example showing matching of two sorted lists.

3 (a)	What are Inverted Lists? Explain how it improves the secondary index structure. Illustrate with an example.	[10]	CO2, CO5	L2

Soln

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.

- Solve the problems associated with the variability in the number of references a secondary key can have
- Greatly reduces the need to reorganize / sort the secondary index
- Store primary keys in the order they are entered, do not need to be sorted
- The downside is that references for one secondary key are spread across the inverted list

4 (a) With example, explain K-way merge and selection tree for merging large number of lists. [10] | CO2 | L2

Soln:

- 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:
  - Find the index of the minimum current item, *minItem*
  - Process *minItem*(output it to the output list)
  - 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*).

- Go back to the first step

- 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.

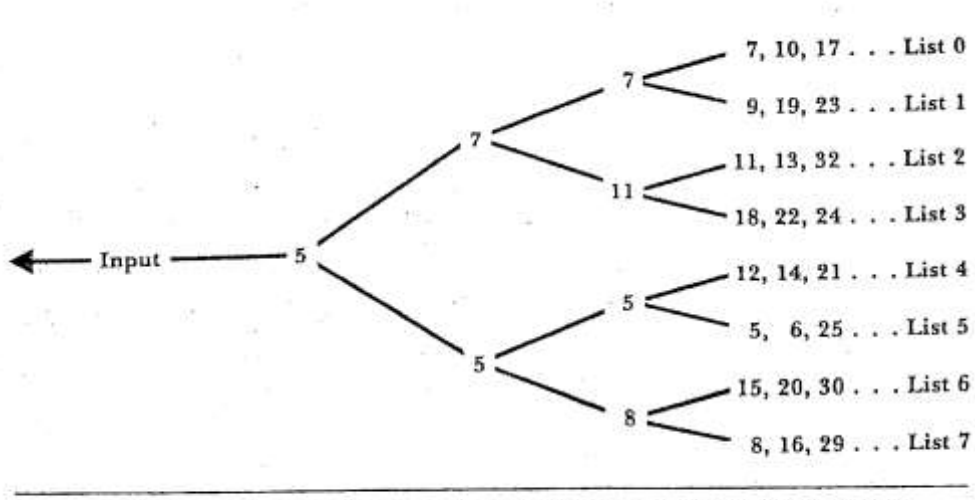
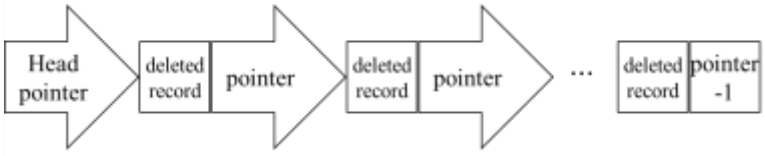


Figure 8.15 Use of a selection tree to assist in the selection of a key with minimum value in a *K*-way merge.

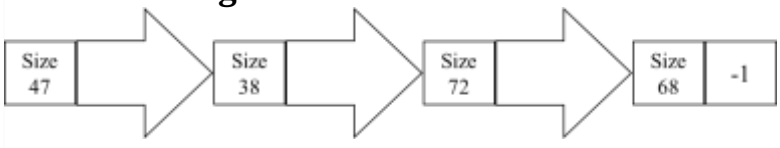
5 (a) What is an Avail List? Explain how avail lists are used in recovering free spaces in fixed length and variable length record files. [10] | CO1, CO2 | L2

Issues on reclaiming space quickly:
– How to know immediately if there are empty slots in the file?
– How to jump to one of those slots, if they exist?
It is done by creating by linking all deleted records together using a **linked list** (Avail List).

Fixed-Length Records



Variable-Length Records



Explain the mechanism of making use of avail list for reclaiming the space created by a deleted record.

6 (a)	Write and explain Keysorting algorithm with an example. What are its limitations?	[10]	CO2, CO5	L2

Keysorting: A sort performed by first sorting keys and then moving records.

- Read each record sequentially into memory, one by one.
- Save the key of the record and the location of the record, in an array (KEYNODES)
- After all records have been read, internally sort the KEYNODES array of record keys and locations
- Using the KEYNODES array, read each record back into memory a second time using direct access.
- Write each record sequentially into a sorted file.

```
int KeySort (FixedRecordFile & inFile, char * outFileName)
{
  RecType obj;
  KeyRRN * KEYNODES = new KeyRRN [inFile . NumRecs()];
  // read file and load Keys
  for (int i = 0; i < inFile . NumRecs(); i++)
  {
    inFile . ReadByRRN (obj, i);// read record i
    KEYNODES[i] = KeyRRN(obj.Key(),i);//put key and RRN into Keys
  }
  Sort (KEYNODES, inFile . NumRecs());// sort Keys
  FixedRecordFile outFile;// file to hold records in key order
  outFile . Create (outFileName);// create a new file
  // write new file in key order
  for (int j = 0; j < inFile . NumRecs(); j++)
  {
    inFile . ReadByRRN (obj, KEYNODES[j].RRN);//read in key order
    outFile . Append (obj);// write in key order
  }
  return 1;
}
```

Limitations:

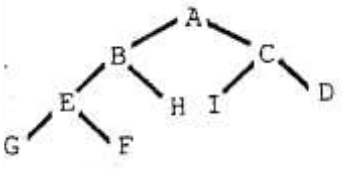
- Only possible when the KEYNODES array is small enough to be held in memory.
- Each record must be read twice: Once sequentially and once directly.
- Each direct access requires a seek.
- Key sorting is a way to sort medium sized files.

7 (a)	Write an algorithm for Heap sort method for insertion. Show the construction of Heap for the following sequence F D C G H I B E A.	[10]	CO2	L3

Insert function for adding a new key to the heap:

```
Insert(NewKey) {
if  (NumElements==MaxElements) return false;
NumElement++;
HeapArray[NumElements]= NewKey;
```

```
int k=NumElements;
int parent;
while (k>1){        // k has a parent
   parent=k/2;
                if (Compare(k, parent) >= 0)//already in order
                        break;
                else
                        Exchange(k, parent);
     k=parent;
                }
return true;
}
```



8 (a)	Write a short notes on: 1. Pinned Records 3. Selective Index 2. Internal Sorting 4. Replacement Selection.	CO2	L2

Soln: **Pinned Records:** A record is pinned when there are other files or file structures that refer to it by its physical location. It is pinned in the sense that we are not free to alter the physical location of the record: doing so destroys the validity of the physical references to the record. Moving such pinned records results in 'dangling pointer', pointers leading to incorrect, meaningless locations in the file.

Internal Sorting: If the entire contents of the file can be held in memory, then the entire file from the disk can be read into memory and then do the sorting there

Selective Index: Index on a subset of records, Provides a selective view of the data. Will contain only some part of the entire index data.

Replacement Selection: It increases the run length during merge sort by making use of multiple heaps. One is called the primary heap and the other as secondary heap.