If the file is full, search comes back to where it began. Only then is it clear, that the record is not in the file.

Greatest strength of progressive overflow is its simplicity. In many cases it is a perfectly adequate method.

) What are the effects of buckets in resolving collision? What are buckets?

Bucket is sometimes used to describe block of records that is retrieved in one disk access, especially when those records are seen as sharing the same address. When record is to be stored or retrieved its home bucket address is determined by hashing. The entire bucket is loaded into primary memory. An in-memory search through successive records in the bucket can then be used to find the desired record.

```
int Hash (char key[12], int maxAddress)
{ int sum = 0;
  for (int j = 0; j < 12; j+=2)
      sum = sum + 100 * key[j] + key[j+1] % 100;
  return sum % maxAddress;
}
```

② What is collision? Explain collision resolution technique with progressive overflow.

Ans In hashing, there's a chance that more than one key map to same address. This is collision & there must be a way to over come this.

Suppose a record whose key is York has to be stored whose address map to same as for the key Rosen which already stored there. Since York can't fit into this address, it's an overflow record. If progressive overflow is used, the next several addresses are searched in sequence until an empty one is found.

An interesting problem occurs when there's a search for or for a record at the end of file. Assume a file can hold 100 records in address 0-99. Blue is hashed to record no. 99, which is already occupied by gello. This is handled by wrap around the address space of the file by choosing address as 0 as the next address ∴ address 0 is not occupied, in this case, Blue gets stored in address 0.

If open is encountered, the searching routine might assume that the record is not in the file.

76 79| 87 69| 76 76| 32 32| 32 32| ~~32 32~~

Before adding them, the fact that the sizes of no. that we add are limited, overflow might occur.

7679 + 8769 + 7676 + 3232 + 3232 = 30588

Assume that keys consist of blanks & upper cases, so largest addend would be 9090 (ZZ). Suppose we choose 19937 as allowable intermediate result. this differs much more than 9090, so we can be confident that no new addition will cause overflow. If it occurs, we use mode operator.

7679 + 8769 → 16448 mod 19937 → 16448

~~16448 + 7679~~

16448 + 7676 → 24124 mod 19937 → 4187

4187 + 3232 → 7419 mod 19937 → 7419

7419 + 3232 → 10651 mod 19937 → 10651

10651 + 3232 → 13883 mod 19937 → 13883.

Divide by size of address space.

a = 13883 mod 100

= 83

a = (s) mod (n)

n = address of the file.

s = sum produced in previous step.

A prime no. is usually used as divisor because prime tends to distribute remainders much more evenly than non primes.

∴ remainder is going to be address of a record, we choose a no. as close as possible to the desired size of addressed space.

Since there are only fewer keys, the index is smaller & hence shallower.

③ How is hashing different from indexing?

Ans Hash function is like a black box which generates an address every time a key is dropped in. It is like indexing involving association of key with a relative record address.

It differs from indexing in 2 important ways:

i) With hashing, addres appear to be in random - there is no obvious connection between the key & the location of corresponding record, even though the key is used to determine the location of the record. Hashing is randomizing.

ii) With hashing, two different keys may be transform -ed to the same address so two records may be sent to the same place in the file. This is collision & in some means, must be found a way to deal with it.

① Explain simple hashing algorithm.

Ans One simple example of hashing algorithm is taking first two letters of the key, convert them to ASCII code multiply them & the last 3 digits indicate address of that key. This is as simple as that.

eg:- i) Consider key = LOWELL

ASCII = 76 79 87 69 76 76 32 32 32 32 32

Thus letters are represented in numerical form — Blanks —

ii) fold & add

2) Write C++ function to find shortest separator between sequence set blocks.

Ans
```
void FindSeparator(char *key1, char *key2, char *sep)
{ while(1)
  { *sep = *key2;
    sep++;
    if (*key2 != *key1)
        break;
    if (*key2 == 0)
        break;
    key1++;
    key2++;
  }
  *sep = 0;
}
```

3) What is simple prefix B+ trees? Explain the maintainence of simple prefix B+ trees

Ans Let's suppose that we want to delete records for EMBRY & FOLKS & that neither of these deletions results in any merging or redistribution within the sequence sets.

The no. of sequence set blocks is unchanged & since no records are moved between blocks, the index set can also remain unchanged. This easy to see in case of EMBRY deletion; it is still a perfectly good separator for sequence set blocks 3 & 4, so there's no change it in the index set. The case of the FOLKS deletion is a little more confusing because the string FOLKS appears both as key in the deleted record & as separator within index set.

The effect of inserting into sequence set new records that do not cause block splitting is much the same as the effect of deletions that don't result in merging: the index set remains unchanged.

Changes involving multiple blocks in the sequence set. If we have more blocks, we need additional separators in the index set, & if we have fewer blocks, we need fewer separators. Changing the no. of separators certainly has an effect on the index set, where the separators are stored. Since index set for a simple B+ tree is just a B-tree, changes to the index set are handled according to the familiar rules for B-tree insertion & deletion.

① Explain common characteristics of B & B+ trees

Any i) All are paged index structures. The shape of these trees tends to be broad & shallow

ii) They are height balanced tree. There are no uneven growth

iii) Trees grows up from bottom up Balance is maintained thro upp block splitting, merging & redistribution.

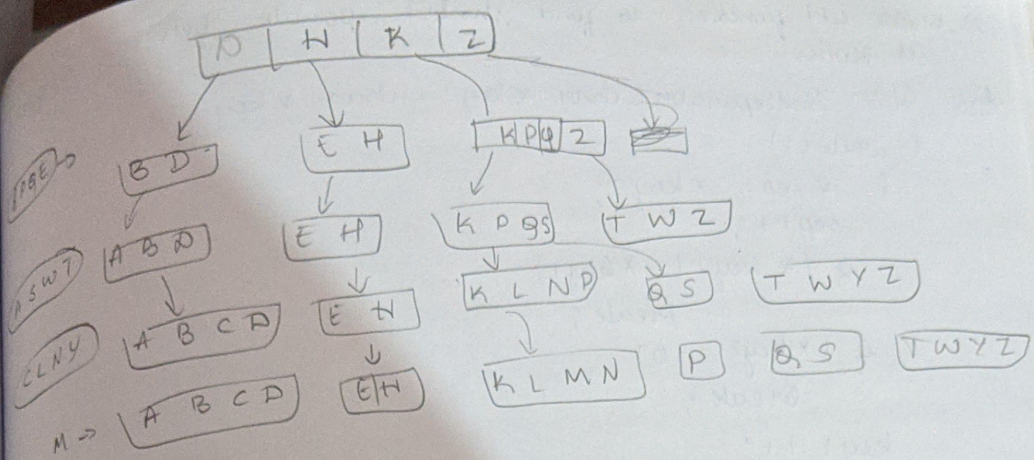
iv) can be adapted for the use of variable length record structure.

② What are the significant advantages of B+ tree over B tree

i) Sequence set can be processed in a truly linear, sequent ial way, providing access to record in order by key.

ii) The index is built with single key or separator per block of data records instead of one key per data record

iii) The size of lowest level index is reduced by the blocking factor of data file



1) What are the considerations for choice of block size for sequence set? Explain.

Ans Consideration 1: The block size should be such that we can hold several data in memory at once. For eg., in performing a block split or merging, we want to be able to hold at least two blocks in memory at a time. If we are to implement 2 to 3 splitting to conserve disk space, we need to hold atleast 3 blocks in memory at a time.

Consideration 2: Reading in or writing out a block should not take very long. Even if we had an unlimited amount of memory, we would want to place an upper limit on the block size so we wouldn't end up reading in the entire file just to get at a single record.

This consideration is redefined as - The block size should be such that we can access a block without having to bear the cost of a disk seek within the block read or block write operation.

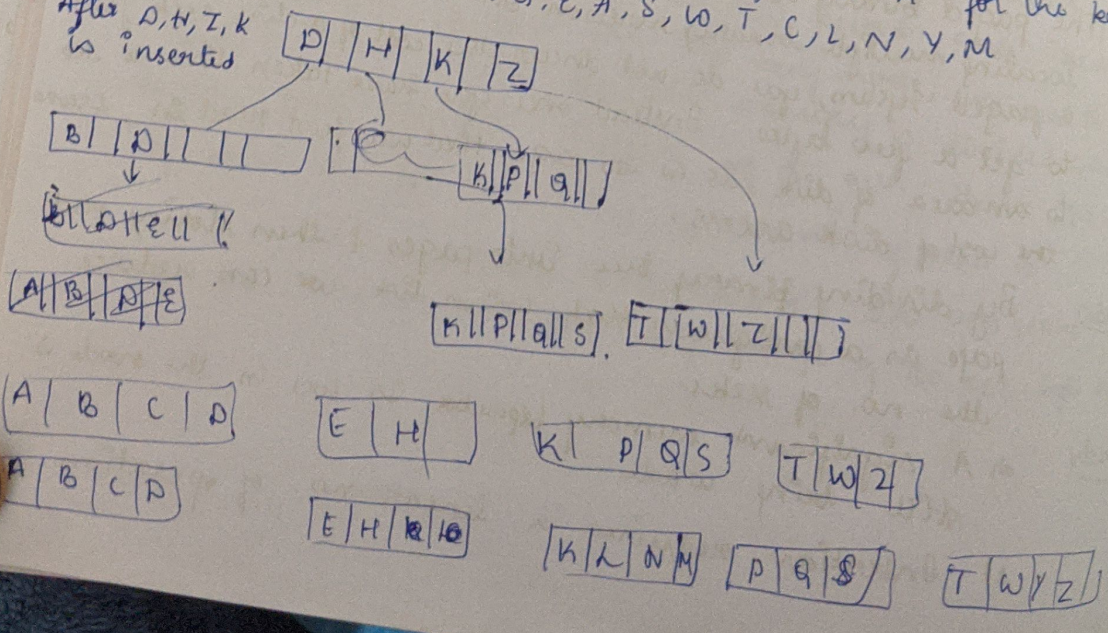
c) Building paged tree is complicated if we are receiving keys in random order & inserting them as soon as we receive them.

i) What is multi-level indexing?

Ans Single record index puts a limit on the no. of keys allowed & that large files need multirecord indexes. A multirecord index consists of a sequence of simple index records. The keys in one record in the list are all smaller that the keys of the next record. Binary search is possible on a file containing an ordered sequence of index records, but its expensive.

The second-level keys can be used to build a second level index with 800 index records. & continue to a 3rd level where 8 index records to index the largest keys in the 800 second level records. Finally 4th level consists of a single record with 8 keys. These 4 levels together form a index tree with a fan out of 100 & can be stored in a single index file.

ii) Construct a binary tree of order 4 for the keys D, H, Z, K, B, P, Q, E, A, S, W, T, C, L, N, Y, M



① what is AVL trees' organization of index file? What are the advantages?

Eg: Suppose, that keys consist of the letters A-G & that we recieve these keys in alphabetical order hitting, the nodes as we recieve them produces a degenerate tree that is, in fact, nothing more than a linked list. One elegant method for handling such reorganisation is using AVL trees. AVL tree is a height balanced tree. Two features of it are:

→ By setting a maximum allowable difference in the height of any subtree, AVL trees guarantee a minimum level of performance in searching.

→ Maintaining a tree in AVL form, as new nodes are inserted into tree use of one of a set of four possible rotations. Each of the rotations is confined to a single, local area of the tree. The most complex of the rotations require only 5 pointer assignment. Advantage is that the prbm of keeping an index in sorted order is solved & ∴ Binary search req/ too many seeks is also solved.

② what are paged binary tree? What are its advantages & disadvantages.

The paged binary tree attempts to address the problem by locating multiple binary nodes on the same disk page. In a paged system, you do not incur the cost of a disk seek just to get a few bytes. Instead once you have taken time to see to an area of disk is in the page that was just read in, & save the cost of disk access.

Adv: By dividing binary tree into pages & then storing each page in a contiguous block loc on disk, we can reduce the no. of seeks.

Disadv: a) A significant memory space is lost in the node still being waste.

b) Insertion requires a linear no. of operation

2) Explain the process of merge sort of large files & explain its advantages.

Ans Multiway Merge sort algorithm provides beginning of an effective solution to the problem of sorting large files such as this one. Since memory sorting alg such as heap sort can work in place, using only a small amount of overhead for maintaining pointers & some temporary variables, we can create a sorted key subset of is almost full, sorting the records in this work area, then writing the sorted records back to disk as sorted subfile.

$$\frac{10\,000\,000 \text{ bytes of memory}}{100 \text{ bytes per records}} = 100\,000 \text{ records.}$$

Once we create the first run, we then read a new set of rec once again filling memory, & create another run of 100,000 re

3) Explain replacement selection procedure with example.

Consider, sort of 80 000 000 records in which each record was 100 bytes. Out initial runs were limited to approx, 100000 records because memory work area was limited to 10 Mb. Suppose we are somehow able to create runs of twice this length, cont. 200 000 records each. Then rather than needing to perform an 800 way merge, we need to do only a 400 merge. Available memory is divided into 400 buffers, each holding 1/800th of a run. Hence, no. of seeks req per run is 800, & total no. of seeks is

800 seeks/run X 400 runs = 320 000 seeks.

Half no required for 800 way merge of 100 000 byte runs

```
class Heap
{ public:
    Heap (int maxElements);
    int Insert (char * newKey);
    char * Remove ();
  protected:
    int MaxElements; int NumElements;
    char ** HeapArray;
    void Exchange (int i, int j);
    int Compare (int i, int j);
        { return strcmp (HeapArray[i], HeapArray
                                        [j]);}
};
int Heap :: Insert (char *newKey)
{ if (NumElements == MaxElements) return False;
    NumElements ++;
    HeapArray [NumElements] = newKey;
    int k = NumElements; int parent;
    while (k > 1) {
            parent = k/2; if Compare (k, parent) >= 0)
    Exchange (k, parent);     break;
    k = parent;
} return True; }
```

~~# Explain process of merge sort of large~~

with multiple buffering, as we process keys in one block from the file, we can simultaneously read later block from file. If we use it, we put each new block at the end of array. Each add a new block, the array gets bigger by the size of block, in effect creating a new i/p buffer for each block in file. So no. of buffer is no. of blocks in file & they are located in sequence in the array.

```
    getch();
    exit(1);
  }
  cq.load();
  cout << "Opening output file \n";
  of.open("file3.txt", ios::out);
  cq.seq.sort(); cq.match();
  f1.close(); f2.close(); of.close();
  getch();
}
```

---

1. Explain process of improving internal sort processing using heap sort & multiple buffering

Ans Heap sort solves the problem by keeping all of key in heap. A heap is a binary tree with certain properties. Heap Sort has 2 parts: first we build heap, then output the keys in sorted order. The first step can occur at virtually the same time that we read the data, so in terms of elapsed it comes essentially for

We read a block of records at a time into a input buffer & then operate on all of the records at a time into an input buffer & ~~that~~ before going to next block. The input buffer for each new block of keys can be a part of memory area that is set up for the heap. The first new record is then at the end of heap array. as required by Insert function. Once that record is absorbed into heap, new one is at the end of heap array, ready to be absorbed into the heap, so forth.

```
for (int k = 0; k<= n2; k++)
{ for (int l= k+1; l <= n2 ; l++)
  { if (strcmp (list2[k], list2 [l]) >0)
    { strcpy_s (temp , list2[k] );
      strcpy_s ( list 2[k] , list2 [j] );
      strcpy_s ( list 2[l] , temp );
    }
  }
}
void coseq :: match ()
{ int i=0, j=0;
  cout << "The matched output is as follows \n";
  while ((i<=n1) && (j<=n2))
  { if (strcmp (list1 [i], list2[j]) == 0)
    { cout << list1 [i] << endl;
      of << list1 [i] << endl;
      i++; j++; continue;
    }
    else if (strcmp (list1[i], list2 [j] ) >0 )
          j++;
    else  i++;
  }
}
void main()
{ coseq cq; cout << "opening first file : \n";
  cout << "Opening first file : \n";
  f1.open ("file1.txt", ios :: in);
  cout << ("Opening second file \n");
  f2.open ("file2.txt", ios :: in);
  if (!f1 || !f2)
  { cout << "ERROR \n";
```

Lab pgm 7

```
#include <iostream>
#include <stdio.h>
#include <fstream>
        <stdlib.h>
        <string>
        <process.h>
        <conio.h>
using namespace std;
class coseq
{
    char list1[100][20], list2[100][20];
    int n1, n2;
  public: void load();
          void seq_sort();
          void match();
};
void coseq :: load()
{  n1=n2=-1;
   while(!f1.eof())
        f1.getline(list1[++n1], 20, '\n');
   while(!f2.eof())
        f2.getline(list2[++n2], 20, '\n');
}
void coseq :: seq_sort()
{
  char temp[30];
  for(int i=0; i<=n1; i++)
  { for(int j=i+1; j<=n1; j++)
    { if(strcmp(list1[i], list1[j])>0)
      {  strcpy_s(temp, list1[i]);
         strcpy_s(list1[i], list1[j]);
         strcpy_s(list1[j], temp);
      }
  } }
```

```
strcpy_s(min, "");
for (i=0; i<k; i++)
    if (!list[i].eof())
    {   strcpy(min, items[i]);
        break;
    }
count = 0;
for (i=0; i<k; i++)
{   if (list[i].eof())
        count++;
    else if (strcmp(items[i], min)<0)
        strcpy(min, items[i]);

}
if (count==8)
        break;
outfile<<min<<"\n";
cout<<min<<"\n";
for (i=0; i<k; i++)
    if (strcmp(items[i], min)==0)
{       list[i].getline(items[i], 20, '\n');

}for (i=0; i<8; i++)
        list[i].close();

-getch();

}
```

lab pgm 8.

```
#include <stdio.h>
#include <conio.h>
#include <iostream>
#include <fstream>
#define k 8
using namespace std;
void opener(fstream &file, char *fn, int mode)
{
    file.open(fn, mode);
    if (!file)
    {
        cout << "unable to open the file";
        getch();
        exit(1);
    }
}
void main()
{
    fstream list[8], outfile;
    char name[8][20] = {"name0.txt", "name1.txt", "name2.txt",
    "name3.txt", "name4.txt", "name5.txt", "name6.txt",
    "name7.txt"};
    char item[8][20], min[20];
    int i, count = 0;
    for (i=0; i<k; i++)
        opener(list[i], name[i], ios::in);
    opener(outfile, "merge8.txt", ios::out);
    for (i=0; i<k; i++)
    {
        list[i].getline(item[i], 20, '\n');
        if (list[i].eof())
            count++;
    }
    cout << "after k-way merging";
    while (count < k)
{
```

we reduce time required to find key with lowest value by using a data structure to save info. about relative key values across cycles of the procedure's main loop. The concept underlying a selection tree can readily communicated through a diagram such as that for

The selection tree is kind of tournament tree in which each higher-level represent the "winner" of comparison b/w the 2 descendant keys. The minimum value is always at the root node of tree. If each key has an associated pointer to the list from which it came, it is simple matter to take the key at the root, read the next element from the associated list, then run the tournament again.

$\lceil \log_2 k \rceil$ is depth for a merge of k-lists.

We can create our output list by working through the ledger & the sorted journal consecutively. We know multiple entries are possible in the journal must be posted to the ledger. Posting involves associating each transaction with its account in the ledger. For eg. the printed output produced for accounts 101, 102, 505, & 510 during the posting operation, given the journal entries). We move ahead to read entry in the journal. The account no. still match. We continue doing this until account no. no longer match.

```
int LedgerProcess::ProcessNewMaster()
{   ledger.PrintHeader(OutputList);
    ledger.Balances[MonthNumber] = ledger.Balances[Month
    Number-1];
}
int LedgerProcess::ProcessCurrentMaster()
{ ledger.Balances[MonthNumber]+ = journal.Amount;
}
int LedgerProcess::ProcessEndMaster()
{ printBalances(OutputList, ledger.Balances[MonthNumber-1],
      ledger.Balances[MonthNumber]);
}
```

(11) What is K-way merging? Explain how selection trees help in K-way merging.

Ans

K way merging operation can be viewed as a process of deciding which of two input items has the minimum value, outputting that item, then moving ahead in the list from which that is taken. In the event of duplicate input items, we move ahead in each list.

Use of selection tree is an classic example of classic time versus space trade off we so often encounter

An important difference between matching & merging is that with merging we must read completely through each of the lists. We need to keep More Names flag set to TRUE as long as there's items in the list (until the end of list). We also have to take care that if one list is exhausted, we must avoid reading from it again. This is achieved by,

a) set the stored 'Item' value for completed list to some value that
   → cannot possibly occur as a legal input value &
   → Has a higher collating sequence value than any possible legal input value. In other words, the special value would come after all legal input value in the files ordered sequence.

③ Explain the application of sequential processing to a general ledger program

Ans Suppose we are given the problem of designing a general ledger posting program as a part of an accounting s/m. The system includes a journal file & ledger file. The ledger contains month by month summaries of the values associated with each of the book keeping accounts.

The journal file contains the monthly transactions that are ultimately to be posted to the ledger file. Entries in journal files are paired. Once the journal file is complete for a given month, meaning it all contains transactions for that month, the journal must be posted to the ledger. Posting involves associating each transaction with its account in the ledger.

(2) Explain merging procedure of co-sequential processing with algorithm

Ans

```
template <class ItemType>
int CosequentialProcess <ItemType> :: Merge2Lists
    (char * List1Name, char * List2Name, char * OutputListName)
{ int MoreItems1, MoreItems2;
  InitializingList (1, List1Name);
  InitializingList (2, List2Name);
  InitializingOutput (OutputListName);
  MoreItems1 = NextItemInList (1);
  MoreItems2 = NextItemInList (2);
  while ( MoreItems1 || MoreItems2) {
     if (Item(1) < Item(2))
     {
        ProcessItem (1);
        MoreItems1 = NextItemInList (1);
     }
     else if ( Item(1) == Item(2))
     {
        ProcessItem (1);
        MoreItems1 = NextItemInList (1);
        MoreItems2 = NextItemInList (2);
     }
     else
     {
        ProcessItem (2);
        MoreItems2 = NextItemInList (2);
     }
  }
  FinishUp();
  return 1;
}
```

(3)

Ans