Module II

ORGANIZATION OF FILES FOR PERFORMANCE AND INDEXING

DATA COMPRESSION

Data compression involves encoding the information in a file in such a way that it takes up less space. Many techniques are available for data compression.

Smaller files are created, so that

i) They use less storage space, resulting in cost savings.
ii) Can be transmitted faster, decreasing access time or alternatively, allowing the same access time but with a lower and cheaper bandwidth.
iii) Can be processed faster sequentially.

Data Compression Techniques –

- Using a different notation
- Suppressing repeating sequences

} Redundancy reduction techniques

- Assigning variable-length codes
- Irreversible compression techniques
- Compression in Unix

a) Using a Different Notation: (Redundancy reduction)

Fixed length fields are good example for compression of this type. For example if the student record contains name of their state, which would occupy 2 ASCII bytes, ie. 16 bits. There are 29 states in India, they can be represented using 5 bits only, instead of using 16 bits. Thus saving 11 bits or more than 50% of address space.

This type of compression technique decreases the number of bits required to store records, by finding a more compact notation.

Disadvatages :

i) File is unreadable - by using pure binary encoding, we have made the file unreadable by humans.
ii) Cost of Encoding / Decoding Time – Time is consumed for encoding and decoding (eg - state name given as KA must be converted into binary format).

iii) Increased Software Complexity - Encoding / Decoding Modules must be used wherever the state value is entered.

This technique is not good if file is very small and if used by many programs. This can be used in files containing several million records and is processed by 1 or 2 programs.

b) Suppressing Repeating Sequences (Run Length encoding) (Redundancy reduction)

If there is a repeating sequences of bits, then this type of compression technique called the Run Length encoding can be used.

Imagine a image of dark sky with bright stars. Here the pixel values of dark sky are 0's and bright stars are 1's. Here the image consists of only 0's & 1's and can be compressed using this technique.

- The conversion algorithm (or **Procedure**) is as follows -
  i) Read through the array in sequence except where the same value occurs more than once in succession.
  ii) When the same value occurs more than once, substitute the following three bytes in order.
    - Special run length code indicator.
    - Value that is repeated.
    - Number of time the value is repeated.

Example: Encode the following sequence of hexadecimal values, choose 'ff' as run length indicator.

22 23 24 24 24 24 24 24 24 25 26 26 26 26 26 26 25 24

Soln: Resulting sequence is:

22 23 ff 24 07 25 ff 26 06 25 24

- Disadvantage:
  i) No guarantee that space will be saved. [If there are no many repeated sequences, then space is not saved].

c) **Assigning variable – length codes**

The principle of working of this method is assign short codes to the most frequent occurring values and long codes to the least frequent ones.

This technique was earlier used in Morse code - for telegraphic communication. It used a standard look up table. But here the code does not change depending upon the occurance of the characters in the current input.

In Hauffman code, the information is coded depending on the frequency of occurrence of each character. In Huffman code the probabilities of each values occurring in the data set is determined and then creates a binary tree depending on probabilities. More frequently occurring values are given shorter search paths in the tree. This tree is turned into a table that can be used to encode and decode the data.

- In morse code **.** and **-** are used to represent characters. Each character have different length of symbols. A look-up table is maintained for each character(A-Z).
- More frequently occurring characters are given shorter search paths in a tree.

➨ Algorithm for creation of hauffman code -

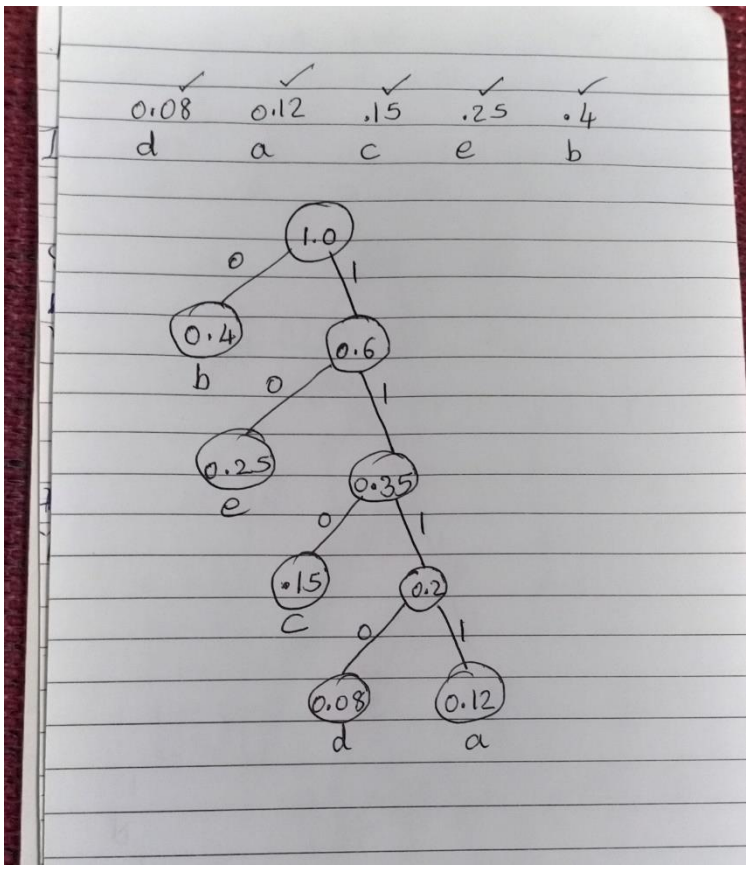
1. Arrange elements in ascender order of probability.
2. Take the 2 left most values (least probability) and create a tree.
3. The root of the created tree is the summation of its probability.
4. Repeat the steps 1 to 3 till all the characters are nodes of the tree
5. Assign the left branch with value '0' and right branch with value '1', for all the branches of the tree.
6. Find the hauffman code of all the characters.

Eg -

Probability of occurrence	0.12	0.4	0.15	0.08	0.25
Characters	**a**	**b**	**c**	**d**	**e**

1. Arrange elements in ascender order of probability.

Probability of occurrence	**0.08**	**0.12**	**0.15**	**0.25**	**0.4**
Characters	**d**	**a**	**c**	**e**	**b**



The hauffman code of all the characters are -

a -> 1111
b -> 0
c -> 110
d -> 1110
e -> 10

d) Irreversible compression technique

- It is based on the assumption that some information can be sacrificed.
- Ex: Shrinking the raster image from 400 by 400 pixels to 100 by 100 pixels. The new image contains 1 pixel for every 16 pixels in the original image. But there is no way to determine what the original pixels were from the one new pixel.
- In data files irreversible compression is seldom used, however they are used in image and speech processing.

e) **Compression in Unix**

- System V Unix has routines called pack and unpack, which uses Huffman codes on a byte by byte basis typically pack achieves 25 to 40% reduction on text files, but less on binary files that have a more uniform distribution of byte values.
  pack appends a .z to the end of file it has compressed.
- Berkeley Unix has routines called compress and uncompress, which uses effective dynamic method called Lempel-zip.
  Compress appends a .Z to the end of the file it has compressed.

RECLAIMING SPACE IN FILES

What is reclaiming of space ?
Suppose a record is deleted (or modified) from a file, the space that was used by the deleted record (called hole), can be reused. This is called reclaiming space in files.

In fixed length records,

- modification of record does not have any problem, as it is modified to same size only.
- deletion of records, the new record inserted can occupy this space as it is of same size.

In variable length records,

- when a record is modified, the length may vary and may not fit into the earlier space. In such case the record is shifted to new address and thus creating a hole in the earlier address.
- When a record is deleted, the new record may not fit exactly to that space, and thus wasting memory.

- Modifications can take any one of 3 forms
  - Record addition
  - Record updating
  - Record deleting

Here, we focus on record deletion, as space is reclaimed on deletion.

Record Deletion and storage compaction:

Storage compaction makes files smaller by looking for places in a file where there is no data at all and recovering this space (or)

Removing the unused space from the file is called storage compaction.

- How to indicate the records as deleted?
  Simple approach is to place a special mark in each deleted record.

Amar | 1VAIS09005 | 23 | 54 | 65
Bharath | 1VAIS09007 | 32 | 54 | 65
Harish | 1VAIS09012 | 36 | 54 | 65

Ex: fig (1): Before a record is marked as deleted.

Amar | 1VAIS09005 | 23 | 54 | 65
* | arath | 1VAIS09007 | 32 | 54 | 65
Harish | 1VAIS09012 | 36 | 54 | 65

Fig (2): After the second record is marked as deleted.

The * symbol as the first field of the record is used to indicate a deleted record.

- After marking the deleted records with *, they are left in file for a period of time. After many records are deleted, the reclamation of space from the deleted records happen all at once. A special program is used to reconstruct the file with all deleted records squeezed out as shown

Amar | 1VAIS09005 | 23 | 54 | 65
Harish | 1VAIS09012 | 36 | 54 | 65

- Storage compaction can be used with both fixed and variable – length records.
- But it does not return the unused space, instantly.
- Dynamic storage reclamation is required , the different techniques used are illustrated in next topics.

Reclaiming space in case of **fixed length** records

- It is necessary to reclaim space of deleted records, so that memory is not wasted.
- To make space reuse happen more quickly, two things are needed

- A way to know immediately if there are unused slots(deleted records) in the file.
- A way to jump directly to one of those slots if they exist.

- The deleted records are marked in some special way, so that it is easy to identify the unused spaces (space that used occupied by deleted records). Usually it is indicated by putting '*' as the first character.

Solution: use linked list in the form of a stack, RRN plays the role of a pointer.

Linked list as stack:

Both the above requirements can be solved by using a linked list for connecting all available(unused) record spaces.

Each node in the list consists of the RRN of deleted record and a link to the next node. A head reference to the first node in the list is used to move through the list by using the node's link field. The traversal of the list is stopped when an end-of-list(-1) is encountered.

A list containing the space available due to deletion of records is called Avail list.

The avail list is used to know the location of free space available to add new records. As it is fixed length records, all the deleted records are of same size

Stack is used to handle the list. A stack is a list in which all insertions and removals of nodes take place at one end of the list. It stores the RRN of the record.

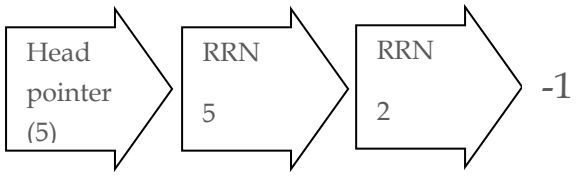


fig: when Record with RRN 5&2 are deleted.

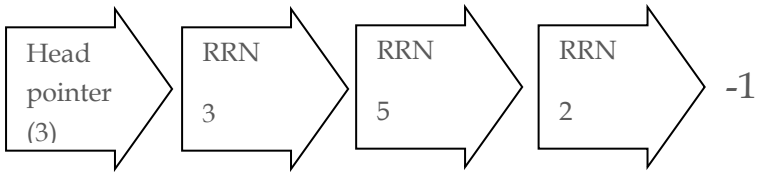


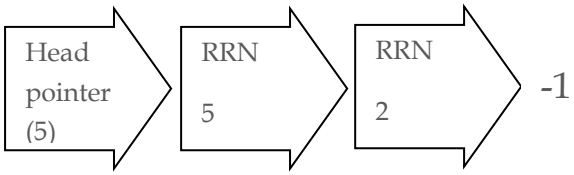
fig: when Record with RRN 3 is also deleted.

Fig: avail list (list of deleted records)

Placing the deleted records on a stack meets both the criteria –

- Finding if there are empty slots
- To jump directly to the slot

Now, if a new record has to be placed in file, the avail list shows that record with RRN 3 is deleted and that space can be reused. When that space is used by new record, the avail list becomes,



If the pointer to the top of the stack contains -1 value, then we know that there are no empty slots and the new records are appended to the end of the file. If the pointer to the stack contains a valid node reference, then we know that a reusable slot is available, and also know the exact position.

Linking and Stacking Deleted records: (Another technique for reclaiming space in fixed length records)

In placing the deleted records on a stack, a separate file has to be maintained to keep the stack. The **linking and stacking deleted records** are done by arranging the links to make one available record slot point to the next, by using the RRN as pointer.

Below figure shows sample file of linked lists of deleted records.

First field of deleted record link to next record on the avail list using RRN.

List head contains the first available record

a) Before deletion of record

List head -> -1

0	1	2	3	4	5	6
Chandan…..	Amar….	Ganesh…	Suraj…..	Master…	Syed…..	John…

b) After deletion of record with RRN 5

List head -> **5**

0	1	2	3	4	**5**	6
Chandan…..	Amar….	Ganesh…	Suraj…..	Master…	***-1**	John…

c) After deletion of record with **RRN 3and 5** in the list
List head **->3**

0	1	2	**3**	4	**5**	6
Chandan…..	Amar….	Ganesh…	***5**	Master…	***-1**	John…

d) After deletion of records **3,5&1** in the order list
List head **-> 1**

0	**1**	2	**3**	4	**5**	6
Chandan…..	***3**	Ganesh…	***5**	Master…	***-1**	John…

e) After **insertion** of a new record(Ramesh……)
List head -> **3**

0	**1**	2	**3**	4	**5**	6
Chandan…..	**Ramesh….**	Ganesh…	***5**	Master…	***-1**	John…

f) After insertion of **2nd** and **3rd** new recrods
List head -> **-1**

0	1	2	**3**	4	**5**	6
Chandan…	Ramesh….	Ganesh…	**2nd new record**	Master…	**3rd new record**	John …

<u>On deletion</u>

- Put * as the 1st field followed by previously used RRN in "List head".
- Assign the RRN of the new deleted record as the first available record in List head.

<u>On addition of new record</u>

- Read the first available slot (RRN) from the list head
- Go to the location, read the RRN of next free slot and assign to the list head.
- Place the new record in current location.

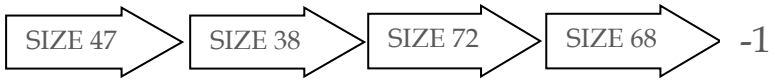
Reclaiming space in case of **variable length** records

The mechanism of reusing the space is complex for variable-length records.

- For record reuse, we need
  - A way to link the deleted records together in a list.
  - An algorithm for adding newly deleted records to the avail list
  - An algorithm for finding (enough space) and removing records from the avail list when we are ready to use them.
  - A way to know the byteoffset of the unused space.

An avail list of variable length records

- Same ideas as in fixed length record is used, but with few modifications.
- We place an asterisk in first field of deleted record followed by a binary link field of pointing to next deleted record on avail list. We cannot use RRN for links, here byte offset is used to point to the next records.



Thus there is a pointer from the first available byteoffset to the next available byteoffset with space available.

Example:

- Fig shows sample file illustrating variable length record deletion.

HEAD FIRST_AVAIL: -1

28 Ames|1VA09IS013|034|023|067| 60 James Watt George Willington|1VA10IS020|075|065|087| 31 Michael|1VA10IS010|050|045|067|

Fig(a): original sample file stored in variable length format with byte count. (Records of Ames,James and Michael)

HEAD FIRST_AVAIL: 35

33 Ames John|1VA09IS013|034|023|067| 60 *|-1……..…… … ……. … … ..| 31 Michael|1VA10IS010|050|045|067|

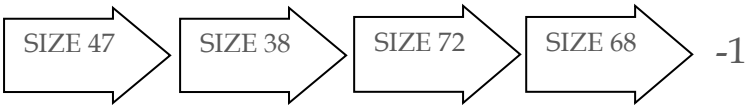
Fig(b): Sample file after deletion of a record of '**James**'

The free space is available at byte offset of 35 and a space of 60 bytes is available.

Adding and removing records:

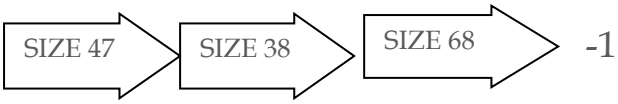
Here, we cannot access the avail list as a stack since the avail list differ in size. We search through the avail list for a record slot though is the right size ("big enough")

Fig shows removal of a record from avail list



a) Before removal

Suppose the new record to be added is of 55 bytes. Traverse the records whose sizes are 47,38,72 and 68 . If a slot big enough to hold the new record is found, remove it from the avail list.



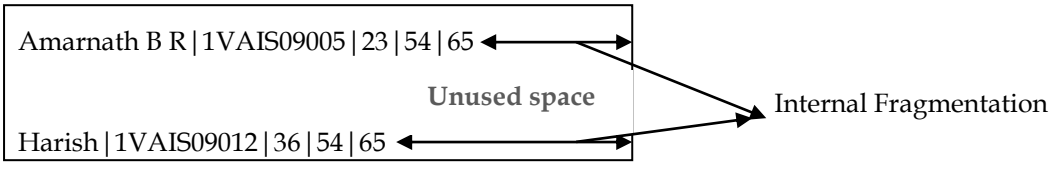
Removed record ->

b) After removal 

The record of size 55 bytes is inserted in the available size of 72.

Storage Fragmentation:

- Internal fragmentation: - wasted space within a record is called internal fragmentation.
- Fixed length record structures often result in internal fragmentation.
- Variable – length records do not suffer from internal fragmentation. However external fragmentation is not avoided.



In variable length records, there is no fragmentation when there is only insertion of records. But when a variable length record file is deleted and a **shorter record is placed in that space – fragmentation occurs.**

Example -

HEAD FIRST_AVAIL: -1

28 Ames|1VA09IS013|034|023|067| 60 James Watt George
Willington|1VA10IS020|075|065|087| 31 Michael|1VA10IS010|050|045|067|

Deleted record space

HEAD FIRST_AVAIL: 35

33 Ames John|1VA09IS013|034|023|067| **60** *| -1…….
…….| 31 Michael|1VA10IS010|050|045|067|

Fig(a): After deletion of first record.

After insertion of new record to deleted space, if there is space remaining, then it is **internal fragmentation**. (as the remaining space is not put back to the avail list)

HEAD FIRST_AVAIL: -1 Internal fragmentation

33 Ames John|1VA09IS013|034|023|067| **20 Ajay|IS01|012|043|065**|…… …. ….
……. …… … . ……. ……. 31 Michael|1VA10IS010|050|045|067|

Fig(b): After insertion of a record into deleted space.

This type of internal fragmentation in variable length record is **avoided** by putting the unused part of the **deleted slot back on to the avail list**.

While inserting the new record, the space available is broken into two parts –

a) Space back on avail list – to avoid internal fragmentation
b) Space for new record.

Example -

A new record of size 20 is added, in to the unused space.

HEAD FIRST_AVAIL: 35

33 Ames John|1VA09IS013|034|023|067| **40** *| -1…… ……. ……
… . . ……. **20 Ajay|IS01|012|043|065**| 31 Michael|1VA10IS010|050|045|067|

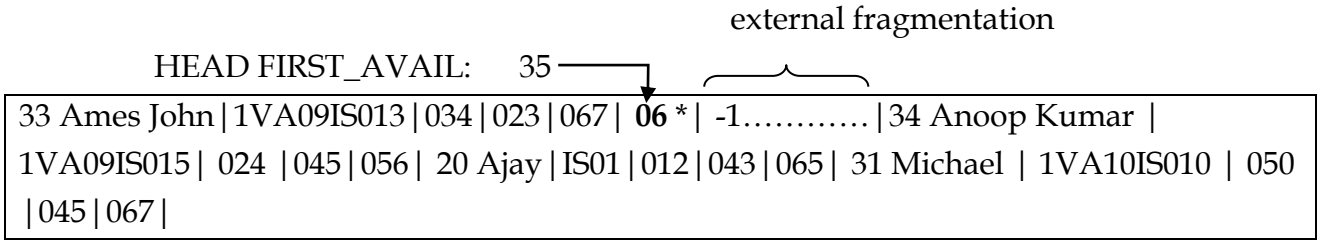
Fig(b): After insertion of a record into deleted space.

Thus the internal fragmentation is avoided.

External Fragmentation: - Form of fragmentation that occurs in a file when there is unused space outside or between individual records.

After the available space is used as shown in above example, the available space becomes so small that it is no more usable. Such fragment which **cannot be used any more** by records is called **external fragmentation**.

Suppose a new record of size 34 is added, in to the unused space. Then the remaining space available is of **size 6.** This small space which cannot be used to store any records is called external fragmentation.



Fig(c): After insertion of another record into deleted space.

- Three ways to deal with external fragmentation
  - Storage compaction
  - Coalescing the holes - if two record slots on the avail list are physically adjacent, combine them to make a single, large record slot.
  - Use a clever placement strategy – try to minimize fragmentation before it happens.

Placement Strategies:

A Placement strategy is a mechanism for selecting the space from the avail list for a new record.

There are three placement strategies used in variable length records –

- First fit placement Strategy: Accept the first available record slot that can accommodate (or large enough to hold) the new record.
- Best fit placement Strategy: Finds the available record slot that is closest in size to what is needed to hold the new record.

- Worst fit Placement strategy: Selects the largest available record slot, regardless of how small the new record.

- First fit strategy:
  The first available space, large enough to hold the new record is selected.
  In this method, the avail list is traced and the first space that is **large enough** to hold the record is selected.
  Advantages:
    - Least possible amount of work to search available space.

  Disadvantages:
    - The selected space may be 10 times bigger than required.

- Best fit strategy:
  The unused space which is minimum required to store the new record is selected. The avail list, containing the slot size is arranged in ascending order, so the min required space is retrieved easily from the avail list.
  Advantages:
    - Only the min required space is taken.

  Disadvantages:
    - Avail list should be arranged in ascending order.
    - We should search through at least part of the avail list to get the required space.
    - Results in external fragmentation

- Worst fit strategy:
  The largest available space is selected from the avail list to store the new record. The avail list is arranged in descending order, so the largest available space in the avail list will be the first to be retrieved. So it can be checked at first itself, whether there is enough space to store the new record.
  Advantages:
    - Need to check only for the first available space, to check the availability of space. No much searching required.
    - Decrease the likely hood of external fragmentation.

  Disadvantages :
    - Avail list should be arranged in descending order.

- Remarks about placement strategies:
  - Placement strategies only apply to variable length records.
  - If space is lost due to internal fragmentation, the choice is between first fit and best fit. A worst strategy truly makes internal fragmentation worse.
  - If the space is lost is due to external fragmentation, one should give careful consideration to a worst fit strategy.

Finding Things Quickly: An Introduction To Internal Sorting And Binary Searching:

- The cost of seeking and retrieving a particular record is very high.
- This cost has to be taken in to consideration when determining a strategy for searching.

(Note: often sorting is the first step to searching efficiently. We develop approaches for searching and sorting that minimizes number of disk access (or seeks)).

Finding things in simple field and record files:

- So far, in case of fixed length records, the only way we have to retrieve of find records quickly is by using their RRN.
- Without a RRN or in case of variable length records, only way so far to do it is by using a sequential search which is very inefficient method.
- We are interested in more efficient ways to retrieve records based on their key value.

Search by **Sequential search**:

- Suppose we are looking for a record by name '**Bharath**' in a file.
- We start by comparing '**Bharath**' with the first record's key in the file.
- If not matched, read the next record and compare the keys.
- This process is repeated until either Bharath's record is found or we have finished searching all the records.
- This kind of searching is called as Sequential Searching.

- Sequential Search Algorithm
  Suppose 'k1' is the element to be found.
  Step 1: Read the next record's key.
  Step 2: Compare k1 and key [k1==key]
  Step 3: If same, key is found
  Otherwise, repeat the Steps 1 and 2.

Search by **Binary search**:

- Suppose we are looking for a record by name '**Bharath**' in a file of 1000 fixed length records.
- Assume the file is sorted in ascending order based on the Key (name).
- We started by comparing '**Bharath**' with the middle record's key in the file, which is the key whose RRN is 500.
- Result of the comparison tells us which half of the file contains Bharath's record.
- Next we compare '**Bharath**' with the middle key among records in the selected half of the file to find out which quarter of the file contains Bharath's record.
- This process is repeated until either Bharath's record is found or we have narrowed the no. of potential records to zero.
- This kind of searching is called as Binary Searching.

- Binary algorithm:

Suppose 'key' is the search element to be found.
Step 1: Assign low=0 & high=num_rec -1.
Step 2: Check if low value is less than high [low<=high]
  Step 2a: Find middle value of low and high
    mid= (low+high)/2
  Step 2b: Check if middle value =key , key found
    [a[mid]=key]
  Step 2c: Check if middle value <key, then assign, low as midvalue+1
    low= mid+1
  Step 2d: Check if middle value >key, then assign, high as midvalue-1
    high= mid-1
Step 3: Repeat step 2 until key is found.

```
int Binary search (Fixed Record File & file, Record type &obj, Key type & key)
\\if Key found, obj contains corresponding record, 1 returned
\\if key not found, 0 returned.
{
int low=0, int high= num Recs()-1;
  while (low<=high)
  {
       int guess= (high-low)/2;
       file.ReadbyRRN (obj, guess);
        if (obj.key()==key) return 1;
        if ( obj.key()<key) high=guess-1;
        else   low= guess+1;
   }
return 0;
}
```

Binary Search	Sequential Search
• Only few records are accessed and checked with the key	• All the records are accessed and checked with the key
• Takes O($\mathbf{\log_2 n}$) comparisons n->no. of records.	• Takes O(n) comparisons
• When the file size is doubled, it adds only one more guess to our worst case	• When the no. of records (file size) is doubled, it doubles the no. of comparisons required.
• File must be sorted	• No need of sorting

Sorting a disk file in memory:

- The process of copying the entire file into memory and sorting it with respect to the key is called **internal sorting**.

- If the entire contents of the file can be held in memory, we can perform an **internal sort** (sorting in memory) which is very efficient.
- But, most often, the entire file cannot be held in memory. In such case keysorting can be done.

Limitations of binary search and internal sorting

Problem 1: Binary search requires more than one or two accesses

- On an average binary search requires approximately $\lfloor \log_2 n \rfloor + 1/2$ comparisons.
- If each comparison requires a disk access, a series of binary search on a list of one thousand items requires on the average 9.5 access per request.
- The cost of this searching is particularly noticeable and objectionable, if we are working on a large file.
- When we access records by RRN rather than by key, we are able to retrieve a record by single access.
- Ideally, are would like to approach RRN retrieval performance while still maintaining the advantages of access by key. (indexing)

Problem 2: Keeping a file sorted is very expensive.

- In order to perform binary search on the file, it must be sorted.
- Many problems are encountered to maintain the file in sorted order.
- To keep the file sorted the new record must be inserted at the correct position.
- To locate the correct position we need to read through half of the records on an average.
- Once this location is found, we have to shift records to open up the space for insertion.

Due to these problems keeping a file sorted is very expensive.

Problem 3: An internal sort works only on small files.

- An internal sort works only if we can read the entire content of a file into the primary memory.
- If the file is so large that we cannot store the entire file into memory, a different sort technique has to be done.

KEY SORTING

- It is a method of sorting a file that does not require holding the entire file in memory.
  Only the keys are copied to memory, then these keys are sorted in memory, and the sorted list of keys is used to construct a new file that has the records in sorted order.
- Advantages: requires less memory than a internal sort
- Disadavtages: process of constructing a new file requires a lot of seeking for records.

- Two differences in key sorting from internal sorting:
  - Rather than read an entire record into memory, we simply read each record into temporary buffer, extract the key, then discard it.
  - When we are writing the records in sorted order, we have to read them according to the sorted order of keys. Thus the records are read two times from original file, and then the new file is created in sorted order.
- Algorithm for key sort:

1. Read record into buffer
2. Store only the keys in memory.
3. Sort the keys
4. Create a new file to store the sorted records.
5. Read a record in sorted order of keys.
6. Copy the record to the new file
7. Continue the steps 5 and 6 till the end of sorted keys

```
int key sort (Fixed Record file & infile, char *out file)
{
      for(int i=0;, i<infile.numrecs();i++)
      {
              infile.readbyRRN(obj.i);                    //1
              keynodes[i]= keyRRN(obj.key(),i);          //2
        }

         sort(KEYNODES, infile.numrecs());        //3
         outfile.create (outfilename);                //4
        for(int j=0, j<infile.numrecs(); j++)
        {
```

```
            infile.readby RRN(obj, KEYNODES[j].RRN);     //5
            outfile.APPEND (obj);                        //6
        }
        return -1;
}
```

Limitations of the key sort method

- Writing the records in sorted order requires as many random seeks as there are records.
- Need to read the records a second time according to the sorted list, so has to move the read/write head, each time to get a different record out of sequence.
- After reading, the record has to be written to another file. So the read/write pointer has to move ones to read data and then to write data to another file.
- Even though writing is in sequence, the r/w head as to move to read the next record.

Another Solution – to maintain the sorted record and access easily

Maintain another separate file, containing the RRN and the key, called index. Sort them and use the RRN to find the required record. Whenever a particular record is required the sorted index file is searched (binary search) and the RRN of that file is used to retrieve required record.

Maintaining the index file is called **indexing**. In records of variable length, key and the address of each record is stored and sorted in index file.

Pinned Records

- A record is pinned when there are other records or file structures that refer to it by its physical location. (in same file or different)
- The avail list is a list of deleted records, 1 record pointing to the next deleted record. Such record having a reference to the next record is called as pinned record.
- Pinned records cannot be moved, because these reference no longer lead to the record; they become dangling pointer.
- Use of pinned records in a file makes sorting more difficult or sometimes impossible.

Updating and maintaining a file is always a problem. Major problem is deleting records and keeping track of the space vacated by deleted records. An avail list of

deleted records slots is created by linking all of the available slots together. This linking is done by writing a link field into each deleted record that points to the next deleted record. This link field gives information about the exact physical location of next available record. When a file contains such references to the physical locations of records, we say that these records are pinned. A pinned record is one that cannot be moved. There exists a connection from one record to another. So the use of pinned records in a file makes sorting more difficult and sometimes impossible.

Solution: use index file to keep the sorted order of the records while keeping the data file in its original order.

INDEXING

WHAT IS AN INDEX?

- An index is a table containing a list of keys and corresponding reference fields. The reference field (address) points to the record where the information referenced by the key is found.
  (Or)
  An index is a tool for finding records in a file it consists of
  - Key field on which the index is searched.
  - Reference field that tells particular address of the key.

- An index lets us to maintain the order of a file without rearranging the file;
- The records in the file are not sorted, but the index file is sorted.
- Indexing gives us keyed access to variable length record files.

Record address (byte offset)	Name	USN	Sem
17	Nikil	1VA09IS051	4
62	Gayatri	1VA09IS023	5
111	Amar	1VA09IS001	6
152	Bharath	1VA09IS010	6
241	Mary	1VA09IS045	4

Fig: contents the sample data file.

Index file

Key	**Address**
1VA09IS001	111
1VA09IS010	152
1VA09IS023	62
1VA09IS045	241
1VA09IS051	17

Index file is sorted according to the key, so that searching and thus accessing of records is easy.

- The index has to be sorted while the file is not this means that the data file may be entry sequenced i.e., the record occur in the order they are entered in the file.

Note on index:

- The index is easier to use than the data file because,
  - It uses fixed length records.
  - Likely to be much smaller than the data file
- By requiring fixed length records in the index file, we impose a limit on size of the primary key field this could cause problems
- The index could carry more information other than the key and reference fields.(e.g., : length of each data record)

Object Oriented Support For Indexed, Entry Sequenced Files Of Data Objects

Operations required to maintain an indexed file

Some operations used to find things by means of the index include the following.

- Create the original empty index and data files.
- Load index file into memory before using it
- Rewrite the index file from memory after using it
- Add data records to data file
- Delete records in data file
- Update records in data file

Creating the files

- Two files must be created: a data file to hold the data objects and an index file to hold the primary key index. Both the index file and the data file are initially created as empty files.

Loading the index into memory

- Only the index file is loaded into the memory.
- The loading into memory can be done sequentially reading a large number of index records.
- Using the index file, the required record is searched and the corresponding record in the file is read through the reference in the index file.

Rewriting the index file from memory

The index file that was loaded on to memory needs to be rewritten back to the index file on the close operation. The changed index file has to be rewritten back to the memory. This action is very important to guard against the power failure, the operator turning the machine off at the wrong time and other such disasters. When such disasters occur the system is suddenly shutdown, and the index file is not rewritten to memory. To prevent such disasters some safeguard measures can be implemented, such as –

- Use a mechanism that permits the program to know when the index is out of date. This can be done by setting a status flag as soon as the copy of the index in memory is changed.
- Have a procedure that reconstructs the index from the data file in case it is out of date.

Record addition

- Adding a new record to data file requires that we also add an entry to the index.
- In data file, record can be added anywhere., however the byte offset of new record should be saved.
- Since the index is kept in sorted order by key insertion of new index entry probably requires some rearrangement of the index.
  We have to shift all the records below by one, to open up space for inserting the new record.
  However, this operation is not costly (no file access) as it is performed in memory.(the entire index file is in memory).

Record deletion:

- The main advantage of indexed file organization is that the records in the file always in sorted order.

- To delete a record we just remove the corresponding entry from index and the space created can be filled up by shifting the below entries to close up the space.
- Since the record deletion takes place in memory, record shifting is not too costly.

Record updating

- Record updating falls in two categories:
  - The update changes the value of the key field
  - The update does not affect the key field.

a) The update changes the value of the key field

- Here, both index and data file may need to be updated, and the index file has to be sorted.
- Conceptually, the easiest way to think of this kind of change is as a deletion followed by an insertion.

b) The update does not affect the key field.

- Does not require rearrangement of the index file.
- If the record size is unchanged or decreased by the update, the record can be written directly into its old space.
- But, if the record size is increased by the update, a new slot for the record will have to be found.

Indexes That Are Too Large To Hold In Memory

- If the index is too large, then: Index access and maintenance must be done on secondary storage.

Disadvantages of storing index file in secondary memory –

- Binary Searching of index requires several seeks rather than being performed at memory speed.
- Index rearrangement (due to record addition/detection) requires shifting or sorting records on secondary storage, this is extremely time consuming.

Solution: if the simple index file is too large to be held in memory, the following techniques can be used –

- Hashed organization- If access speed is a top priority.

- Tree structured or multilevel index (like B-tree) – if you need the flexibility of both keyed access and ordered sequential access.

Advantages of storing simple indexes on secondary storage over the use of data file sorted by key are:

- A simple index allows use of binary search in a variable length record file.
- If the index entries are substantially smaller than the data file records, sorting and maintaining the index can be less expensive than the data file.
- If there are pinned records in the data file, the use of an index lets us rearrange the keys without moving the data records.
- Provides multiple views of a data file.

Indexing To Provide Access By Multiple Keys

- So far, our index allows only key access. i.e., you can retrieve record 1VA09IS023, but you cannot retrieve a recording of Gayatri.
- We could build catalog for our record collection consisting of entries for name, and the semester. These fields are secondary key fields.
- Fig shows index file that relates name to USN

Name index file

Secondary key (NAME)	Primary key (USN)
Amar	1VA09IS001
Bharath	1VA09IS010
Gayatri	1VA09IS023
Mary	1VA09IS045

Fig : **secondary key index** organized by NAME.

- There can be NAMES which are same, in different records, ie. Secondary key may be the same(grouped together, as the index file is sorted), but primary key is always unique. So to find the actual byte offset of a record, we relate the secondary key to a primary key which then will point to the actual byte offset.

Record addition:

- When a secondary index is used, adding a record involves updating the data file, the primary index, and the secondary index. Secondary index update is similar to primary update (i.e., in both records must be shifted)
- Similar to primary indexes, cost of doing this greatly decreases if the secondary indexes can be read into memory and changed there.
- Secondary keys (or key field in sec. index file) are stored in **canonical form** (all the names in capitals)
- Fig shows sec key index organized by name
  Title index

Secondary key	Primary key
Amar	1VA09IS001
Bharath	1VA09IS010
Chethan	1VA09IS015
Chethan	1VA09IS018
Chethan	1VA09IS019
Gayatri	1VA09IS023
Mary	1VA09IS045

One important difference between secondary index and primary index is that a secondary index can contain duplicate keys (grouped together) and the primary index will not have duplicate keys.

In the example shown above, there are three records with the name 'Chethan'. Within this group, they should be ordered according to the values of the reference field(primary keys).

Record Deletion

- Removing a record from data file means removing the corresponding entry in primary index and all the entries in secondary indexes that refer to this primary index entry.
- Like primary index, the secondary indexes are maintained in sorted order by key. Deleting an entry would involve rearranging the remaining entries to close up the space left open by deletion.
- Thus deleting a record consumes more time with respect to secondary index file.

- This can be avoided by deleting corresponding entry of only the primary index file. The secondary index file is not changed. Thus eliminating the modification and rearrangement in secondary index.
- When a record is searched using secondary key, the key is searched in secondary index and the corresponding primary key is found. The corresponding entry is not found in primary index file , and the search ends, informing the user that the record does not exists.

Disadvantage: Deleted records take up space in the secondary index files.
Solution: B-Tree (allows for deletion without having to rearrange a lot of records)

Record updating

- There are 3 possible situations
  i. Update changes the secondary key:
  We may have to rearrange the secondary key so it stays in sorted order. (Relatively expensive operation)
  ii. Update changes the primary key:
  Has large impact (or changes) on primary key index but often requires that we update only the affected primary key (reference field) in all secondary index also. If the secondary key of the affected primary key is the same, then sorting is done according to the primary key.
  iii. Update confirmed to other fields:
  No changes necessary to secondary indexes. But the primary index file will change if the address of the modified record changes.

Retrieval Using Combinations Of Secondary Keys

- One important application of secondary keys involves using two or more of them in combination to retrieve special subsets of records from the data file.
- With secondary keys, we can search for things like.
  - Records with name 'Chethan'
  - Records with semester 'I'
  - Records with name 'Chethan'and semester 'I'.

Suppose, there are two secondary indexes, one containing name as secondary key and the other containing semester as secondary key. To retrieve the record of **Chetan studying in I semester**.

Secondary indexes are –
Name as secondary key

Secondary key	Primary key
Amar	1VA09IS001
Bharath	1VA09IS010
Chethan	1VA09IS015
Chethan	1VA09IS018
Chethan	1VA09IS019
Gayatri	1VA09IS023
Mary	1VA09IS045

Selected students named 'Chethan'

Chethan	1VA09IS015
Chethan	1VA09IS018
Chethan	1VA09IS019

Semester as Secondary key

Secondary key	Primary key
I	1VA09IS001
I	1VA09IS010
I	1VA09IS015
II	1VA09IS018
II	1VA09IS019
III	1VA09IS023
III	1VA09IS045

Selected students of I semester

I	1VA09IS001
I	1VA09IS010
I	1VA09IS015

The student name and semester requested coincides with the record having primary key as 1VA09IS015. So from the primary index the byteoffset is retrieved for that primary key to retrieve the record.

Improving The Secondary Index Structure: Inverted Lists

Inverted list are indexes in which a key is associated with a list of reference fields.

- Secondary index structures results in two distinct difficulties:
  - Addition of new record with same secondary key - We have to rearrange the index file every time a new record is added to the file even if the new record is for an existing secondary key
  - Redundant secondary key value - If there are duplicate secondary keys, the secondary key field is repeated for each entry and space is wasted larger index files are less likely to fit in memory

Solution for these difficulties is created of files such as secondary indexes, in which a secondary key leads to a set of one or more primary keys, called inverted list.

- There are two solutions
  Solutions 1
  - Change the secondary index structure so it associates an array of references with each secondary key.
    E.g. Chethan 1VA09IS015 1VA09IS018 1VA09IS019
- Fig shows secondary key index containing space for multiple references for each secondary key.

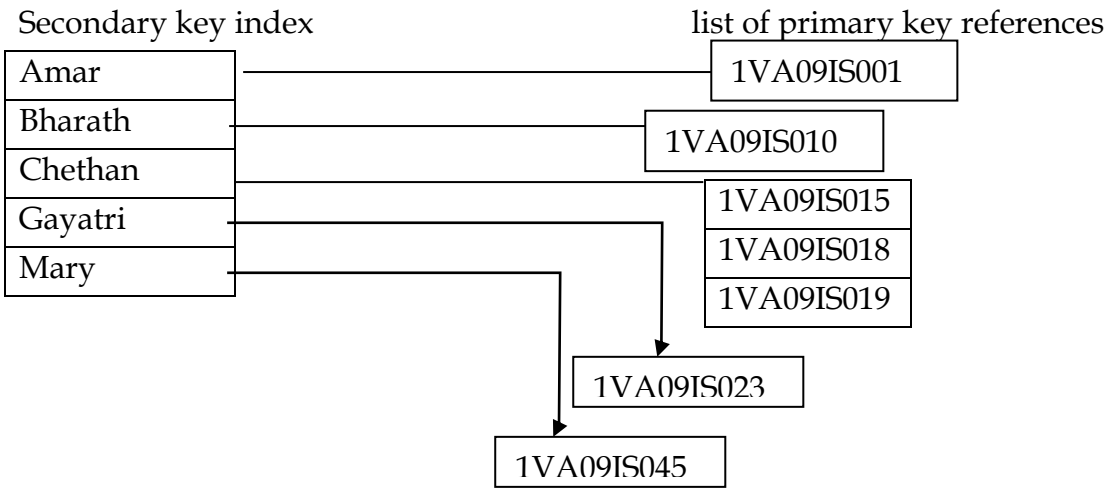
Revised Student index	
Secondary key	Set of primary key references
Amar	1VA09IS001
Bharath	1VA09IS010
Chethan	1VA09IS015 1VA09IS018 1VA09IS019
Gayatri	1VA09IS023
Mary	1VA09IS045

ADV	DISADV
Avoids the need to rearrange the secondary index file until a new secondary key is added.	• May restrict the number of references that can be associated with each secondary key. • Cause internal fragmentation

Solution 2: Linking the list of references (better solution)

- Method: Each secondary key point 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.

- Fig shows conceptual view of primary key reference fields as a series of lists.



Advantages

- No wastage of space due to internal fragmentation.
- When a new record is added, the new primary key in secondary index file are stored and arranged.

Disadvantage

- A large number of small files are required to store the list of primary keys.

An alternative approach to the above method

So, an alternative approach was used that is to redefine the secondary index as consisting of 2 fields – the secondary key and the RRN number of the primary key reference. The primary reference file are sequentially stored in a separate file called **primary key reference file**.

Secondary index file

Secondary key	RRN
Amar	0
Bharath	1
Chethan	2
Gayatri	3
Mary	4

Primary key reference file

RRN	Primary key	Link (RRN)
0	1VA09IS001	-1
1	1VA09IS010	-1
2	1VA09IS015	5
3	1VA09IS023	-1
4	1VA09IS045	-1
5	1VA09IS018	6
6	1VA09IS019	-1

Rearranging of secondary index file is required when a new secondary key is added to the data file or when there is no record of that name. When a record with an

existing name is added the primary key should be added to primary key reference file only.

Advantages

- Secondary index file needs to be rearranged only when new record, with a different student name is added (i.e., when new student's name is added or modify)
- Re arranging is faster since there are fewer records and each record is smaller.
- There is less need for sorting. Therefore we can keep secondary index file on disk. So that more space is available in primary memory.
- Primary key reference file is entry sequenced i.e. the USNs of new records are added at the end of the reference file, primary index never needs to be sorted.
- Space from deleted primary key reference file can easily be reused.

Disadvantage

- The reference ID associated with a given name are no longer guaranteed to be grouped together physically. i.e., locality (together) in the secondary index has been lost.
- Since the reference file is very long, they are usually stored in secondary memory, leading to large seek time.

SELECTIVE INDEXES

- A selective index contains keys for only a portion of the records in the data file. Such an index provides the user with a view of a specific subset of the file's records.

BINDING

The process of bounding the key to the physical address of its associated record is called binding.

- The binding of our primary keys takes place at construction time.
  Adv: faster access.
  Disadv: Re organization of data file must result in modifications to all bound index files.
- Binding of our secondary keys takes place at the time they are used.
  Adv: Safer.

- Trade off in binding decisions:

- Tight binding (construction time binding i.e. during preparation of data file) is preferable when data file is static or nearly static, requiring little or no adding, deleting, or updating.
  Rapid performance during actual retrieval is a high priority.

Note: In tight binding, indexes contain explicit references to the associated physical data record.
Postponing binding as long as possible is simpler and safer when the data file requires a lot of adding, deleting and updating.

Note: Here the connection between a key and a particular physical record is postponed until the record is retrieved in the course of program execution.

Important Questions:

1. Define data compression. Explain the different techniques to compress data. (10)
2. Explain irreversible compression technique. (4)
3. Explain how spaces can be reclaimed in files. (10)
4. What is an index? Explain a simple index for entry sequenced file. (8)
5. Explain key sorting algorithm, with an example. Explain its limitations.(6)
6. Explain the limitations of binary searching and internal sorting (4)
7. Explain the operations required to maintain an indexed file in detail.(6)
8. Discuss the advantages and disadvantages of indices that are too large to hold in memory. (4)
9. Define fragmentation and types of fragmentation. (4)
10. How is the secondary index structure improved? (8)
11. Problems on haufmann code. (6)
12. Explain the different placement strategies with ex. (10)

13. What is redundancy reduction? Explain how run-length-encoding helps i redundancy reduction with an example
14. Write an algorithm for searching a record from a file using i)binary search ii)sequential search
15.

16. Explain how spaces can be reclaimed in files.
    Or
17. How spaces can be reclaimed from deletion of records from fixed length record file and variable length record file?
18. What is an index? Explain a simple index for entry-sequenced file.