P1 **Write a program to read and write student objects with fixed-length records and the fields delimited by "|". Implement pack ( ), unpack ( ), modify ( ) and search ( ) methods.**

Fixed length record

A record which is predetermined to be the same length as the other records in the file.

Record 1	Record 2	Record 3	Record 4	Record 5

- The file is divided into records of equal size.
- All records within a file have the same size.
- Different files can have different length records.
- Programs which access the file must know the record length.
- Offset, or position, of the nth record of a file can be calculated.
- There is no external overhead for record separation.
- There may be internal fragmentation (unused space within records.)
- There will be no external fragmentation (unused space outside of records) except for deleted records.
- Individual records can always be updated in place

Delimited Variable Length Fields

Record 1	|	Record 2	|	Record 3	|	Record 4	|	Record 5

- The fields within a record are followed by a delimiting byte or series of bytes.
- Fields within a record can have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the delimiter.
- The delimiter cannot occur within the data.
- If used with delimited records, the field delimiter must be different from the record delimiter.
- There is external overhead for field separation equal to the size of the delimiter per field.
- There should be no internal fragmentation (unused space within fields.)

Pack():

This method is used to group all the related field values of particular record taken by the application in buffer.

Unpack():

This method is used to ungroup all the related field values of percular record taken from the file in buffer.

P2 **Write a C++ program to read and write and student objects with variable-length records using any suitable record structure. Implement pack(),unpack(),modify() and search() methods.**

Variable length record

A record which can differ in length from the other records of the file.

- **delimited record**

A variable length record which is terminated by a special character or sequence of characters.

- **delimiter**

  A special character or group of characters stored after a field or record, which indicates the end of the preceding unit.

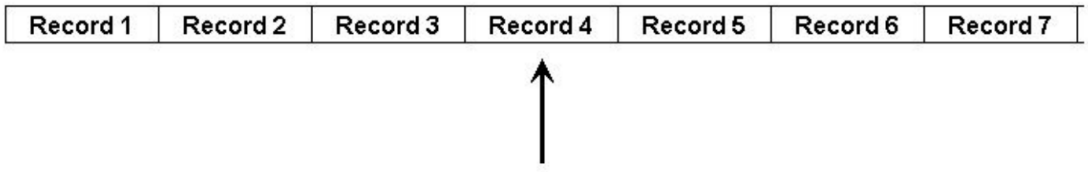
  - The records within a file are followed by a delimiting byte or series of bytes.
  - The delimiter cannot occur within the records.
  - Records within a file can have different sizes.
  - Different files can have different length records.
  - Programs which access the file must know the delimiter.
  - Offset, or position, of the nth record of a file cannot be calculated.
  - There is external overhead for record separation equal to the size of the delimiter per record.
  - There should be no internal fragmentation (unused space within records.)
  - There may be no external fragmentation (unused space outside of records) after file updating.
  - Individual records cannot always be updated in place.

P3 **Write a c++ program to write student objects with variable-length records using any suitable record structure and to read from this file a student record using RRN.**

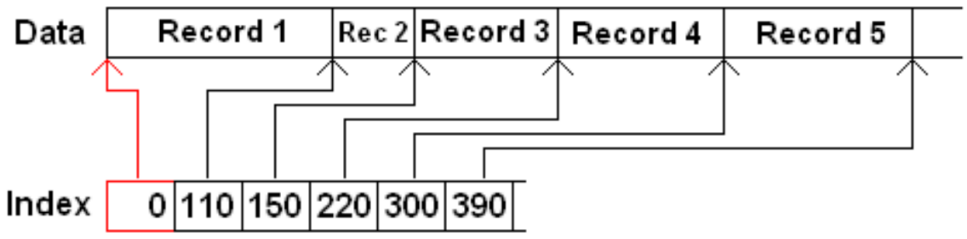
RRN(relative record number)

- RRN is an ordinary number that gives the distance of current record from first record. Using RRN, Direct access allows individual records to be read from different locations in the file without reading intervening records.
- When we are using fixed length record, we can calculate the byte offset of each record using the fallowing formula
- ByteOffset = (RRN - 1) × RecLen
  - RRN: relative record number(starts fron 0)
  - RecLen: size of fixed lenth record

Direct Access



P4 **Write a C++ program to implement simple index on primary key for a file of student objects. Implement add(),search(),delete() using the index.**

	Index 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. **Key field** The part of an index which contains keys. **Reference field** The part of an index which contains information to locate records. • An index imposes order on a file without rearranging the file. • Indexing works by indirection. **Simple Index for Entry-Sequenced Files** **Simple index** • An index in which the entries are a key ordered linear list. Simple indexing can be useful when the entire index can be held in memory. Changes (additions and deletions) require both the index and the data file to be changed. • Updates affect the index if the key field is changed, or if the record is moved. An update which moves a record can be handled as a deletion followed by an addition. 
P6	**Write a C++ program to read two lists of names and then match the names in the two lists using Consequential Match based on a single loop. Output the names common to both the lists.**

Cosequential operations

Operations which involve accessing two or more input files sequentially and in parallel, resulting in one or more output files produced by the combination of the input data.

Considerations for Cosequential Algorithms

- Initialization - What has to be set up for the main loop to work correctly?
- Getting the next item on each list - This should be simple and easy, from the main algorithm.
- Synchronization - Progress of access in the lists should be coordinated.
- Handling End-Of-File conditions - For a match, processing can stop when the end of any list is reached.
- Recognizing Errors - Items out of sequence can "break" the synchronization.

Matching Names in Two Lists

Match

The process of forming a list containing all items common to two or more lists.

Cosequential Match Algorithm

- Initialize (open the input and output files.)
- Get the first item from each list.
- While there is more to do:

    Compare the current items from each list.

        If the items are equal,

            Process the item.

            Get the next item from each list.

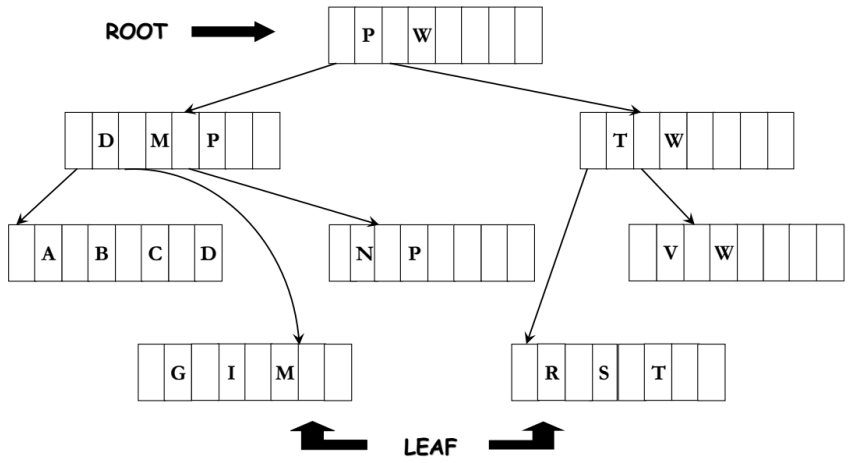
            Set *more* to true iff none of this lists is at end of file.

        If the item from list *A* is less than the item from list *B*,

            Get the next item from list *A*.

	Set *more* to true iff list *A* is not at end-of-file. If the item from list *A* is more than the item from list *B*, Get the next item from list *B*. Set *more* to true iff list *B* is not at end-of-file. • Finalize (close the files.)
P7	**Write a C++ program to read k Lists of names and merge them using kway merge algorithm with k = 8.** **Merge** The process of forming a list containing all items in any of two or more lists. **K-way merge** A merge of order k. **Order of a merge** The number of input lists being merged. • If the distribution phase creates *k* runs, a single k-way merge can be used to produce the final sorted file. • A significant amount of seeking is used by a k-way merge, assuming the input runs are on the same disk.
M1	**Write a C++ program to implement B-Tree for a given set of integers and its operations insert ( ) and search ( ). Display the tree**

B Tree of Order 4



Operations on B Trees

- Insertion
- Deletion
- Searching

Insertion

- Begins with a search that proceeds all the way down to the leaf level.
- After finding the insertion location at the leaf level, the work of insertion, overflow detection & splitting proceeds upwards from the bottom.
- Create new root node, if current root was split.

B TREES : An Overview

- ✦ Multilevel indexes solving the problem of linear cost of insertion & deletion.
- ✦ Built upwards from the bottom rather than downward from the top.
- ✦ Order is the maximum number of immediate descendants that a node can have.
- ✦ Page is an index that holds many keys.
- ✦ Leaf is a page at the lowest level of the B Tree.
- ✦ Height is the longest simple path from the root to a leaf.
- ✦ Space Utilization is the ratio of amount of space used to hold the keys to the amount of space required to hold the B Tree.

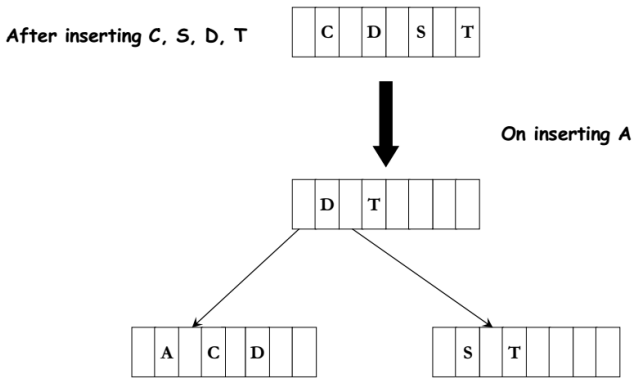
Properties

For a B Tree of Order *m*

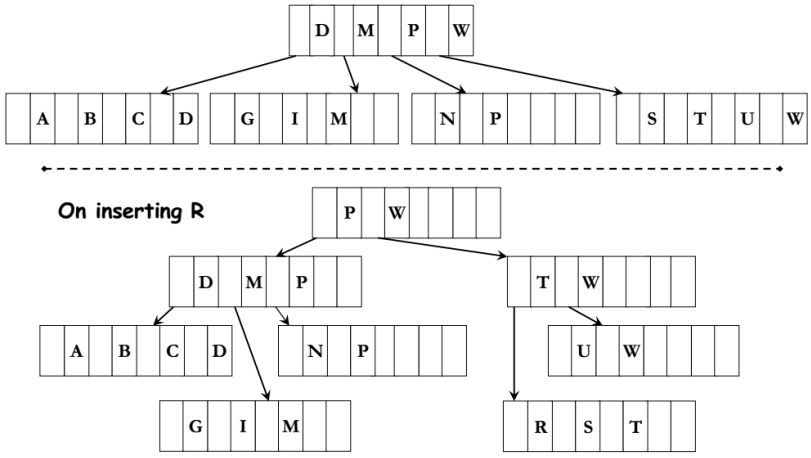
- ◆ Every node has a maximum of *m* descendants
- ◆ Every node, except for root & leaves, has at least *m/2* descendants
- ◆ Root has at least 2 descendants

  (unless it is a leaf)
- ◆ All leaves are at the same level

No Split & Contained Split



Recursive Split



Searching

Recursive

Loads the page into memory & then searches through it.

Looks for key at successive lower levels of tree until it is found

M2	**Write a C++ program to implement B+ tree for a given set of integers and its operations insert ( ), and search ( ). Display the tree.**

B+ tree structure is designed mainly in order to overcome the disadvantages of the B tree structure. The functionality of a B+ tree structure is although is similar to the behavior B tree structure, the flexibility and efficiency wise B+ tree structure is far better than its counterpart.

Basically B+ tree is a combination of B tree structure and the sequence structure. The B tree structural nodes are known as the index set and the sequence structure are known as the sequence set.

In all B-tree type structures, key search proceeds from the root downwards, following pointers to the nodes which contain the appropriate range of keys, as indicated by the reference values. Likewise, all B-trees grow from the leaves up. After obtaining the appropriate location for the new entry, it is inserted. If the node becomes overfull it splits in half and a pointer to the new half is returned for insertion in the parent node, which if full will in turn split, and so on.

B+trees distinguish **internal and leaf nodes**, keeping data only at the **leaves,** whereas ordinary B-trees would also **store keys in the interior**. B+tree insertion, therefore, requires managing the interior node reference values in addition to simply finding a spot for the data, as in the simpler B-tree algorithm.

B+tree algorithms incorporate an **insertion overflow** mechanism to enforce higher node utilization levels. B+tree insertion at full nodes may avoid splitting by first checking neighboring nodes. Keys from the full node are redistributed to a less full neighbor. If both neighbors are full, however, the split must take place.

Sequence set:

A sequence set is a **linked structure** of many number of sequence nodes. Each node of a sequence set contains **key-reference pairs**, which are sequenced based on key and a pointer, which points to its next successor. All the nodes are linked such that, when we start off at the first sequence node and if traverse through all the nodes, we can access all the records of the data file in ascending or descending order of key. The sequence set is considered as the **leaf level nodes** of the B+ tree structure.

Index set:

An index set structure of B+ tree is quite similar to the B tree structure. In B tree nodes, for every **key there is a corresponding reference value**. But in B+ tree index node, if the number of keys or separators is n, then number of reference values is always (n+1).

Here the separator **is the key value** which helps in distinguishing either index nodes or the sequence nodes and also the keys in combination with the reference values provide a path in order to search for a particular records present at the leaf level of B+ tree structure.

M3 **Write a C++ program to store and retrieve student data from file using hashing. Use any collision resolution technique**

WHAT IS HASHING?

A *hash* function is like a black box which produces an address every time you drop in a key.

It is a function h(K) that transforms a key K into an address.

The addresses generated appear to be random hence hashing is also referred as *randomizing.*

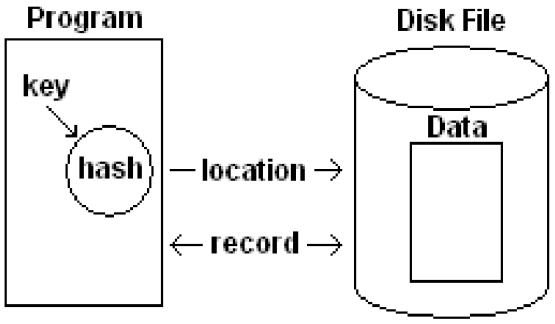
The resulting address is used as the basis for storing and retrieving records.The key in our project is the student University Seat Number (USN)

Hashing

- How else can a single record in a large file be accessed?

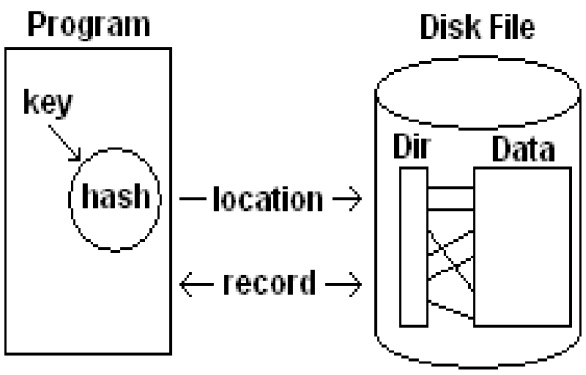
hashing

The transformation of a search key into a number by means of mathematical calculations.



Extendible Hashing

- How can hashing be applied go large, growing, files?
- **extendible hashing**
  An application of hashing that works well with files that over time undergo substantial changes in size.



HASHING ALGORITHM

REPRESENT THE KEY IN NUMERICAL FORM

If the key is a number, then this step is already accomplished.

For a string of characters, we take the ASCII code of each of the characters and use it to form a number.

For e.g.. LOWELL

76 79 87 69 76 76 32 32 32 32 32 32

L O W E L L | |

Blanks

In this algorithm we use the entire key rather than a part of it.

By using more parts of the key the probability of difference in address due to different keys is high.

FOLD AND ADD

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

- Integer values exceeding 32767 might cause an overflow
- Assuming the keys consist of only uppercase and blank characters, the largest addend is 9090
- These number pairs are considered as integer variables

- Hence the largest allowable intermediate result will be
  32767-9090=19937
- To ensure this ,we apply mod 19937 to the result

DIVIDE BY SIZE OF THE ADDRESS SPACE

In this step we divide the number produced in step-2 by the address size of the file.the remainder will be the home address of the file.

CREATING ADDRESSES

Because extendible hashing uses more bits of the hashed address as they are needed to distinguish between buckets, we need a function *makeaddress* that extracts a part of the full address.

It is also used to *reverse* the order of the bits in a hash address.

This is done because the LSBs of these integer values tend to have more *variation* than the MSBs

For e.g.. In a 4-bit address space we want to avoid the address of Bill ,Lee,Pauline to be 0000

Bill	0000 0011 0110 1100
Lee	0000 0100 0010 1000
Pauline	0000 1111 0110 0101

HASHING SCHEME

A function Hash ( ) generates the hash value for a key.

Make address ( ) reverses the order of bits to achieve more randomization.