Introduction to Searching

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.

However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.

There are two popular methods for searching the array elements:

- **Linear Search**
- **Binary Search**

The algorithm that should be used depends entirely on how the values are organized in the array.

For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.

Linear Search

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array A[10] is declared and initialized as,

int A[10] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5}; Val = 7 then Pos = 3

Linear Search Algorithm

LINEAR_SEARCH(A, N, VAL)

Step 1: [INITIALIZE] SET POS = -1

Step 2: [INITIALIZE] SET I = 1

Step 3: Repeat Step 4 while I<=N

Step 4: IF A[I] = VAL

SET POS = I

PRINT POS

Go to Step 6

[END OF IF]

SET I = I + 1

[END OF LOOP]

Step 5: IF POS = –1

PRINT VALUE IS NOT PRESENT IN THE ARRAY

[END OF IF]

Step 6: EXIT

Complexity of Linear Search Algorithm

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.

Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made.

Example

Consider the following list of element and search element..

Step 1:

search element (12) is compared with first element (65)

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Binary Search

Binary search is a searching algorithm that works efficiently with a sorted list. The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

The same mechanism is applied in the binary search.

Now, let us consider how this mechanism is applied to search for a value in a sorted array.

Consider an array A[11] that is declared and initialized as, int A[11] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; and value to be searched is VAL = 9

The algorithm will proceed in the following manner.

BEG = 0, END = 10, MID = (0 + 10)/2 = 5

Now, VAL = 9 and A[MID] = A[5] = 5

A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So, we change the values of BEG and MID.

Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8

VAL = 9 and A[MID] = A[8] = 8

A[8] is less than VAL, therefore, we now search for the value in the second half of the segment. So, again we change the values of BEG and MID.

Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9

Now, VAL = 9 and A[MID] = 9.

The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.

However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If VAL <A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.

(b) If VAL >A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.

Binary Search Algorithm

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET BEG = lower_bound

END = upper_bound, POS = - 1

Step 2: Repeat Steps 3 and 4 while BEG <= END Step 3: SET MID = (BEG + END)/2 Step 4: IF A[MID] = VAL SET POS = MID PRINT POS Go to Step 6 ELSE IF A[MID] > VAL

SET END = MID - 1

ELSE SET BEG = MID + 1

[END OF IF]

[END OF LOOP]

Step 5: IF POS = -1

PRINT “VALUE IS NOT PRESENT IN THE ARRAY”

[END OF IF]

Step 6: EXIT

Time complexity

As we dispose off one part of the search case during every step of binary search, and perform the search operation on the other half, this results in a worst case time complexity of O(log2N).

Example

Consider the following list of element and search element..

10 12 20 32 50 55 65 80 99

search element 12

Step 1:

search element (12) is compared with middle element (50)

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

search element 80

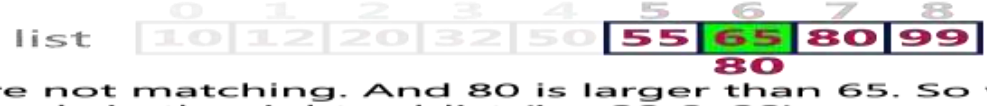
Step 1:

search element (80) is compared with middle element (50)

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)



Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

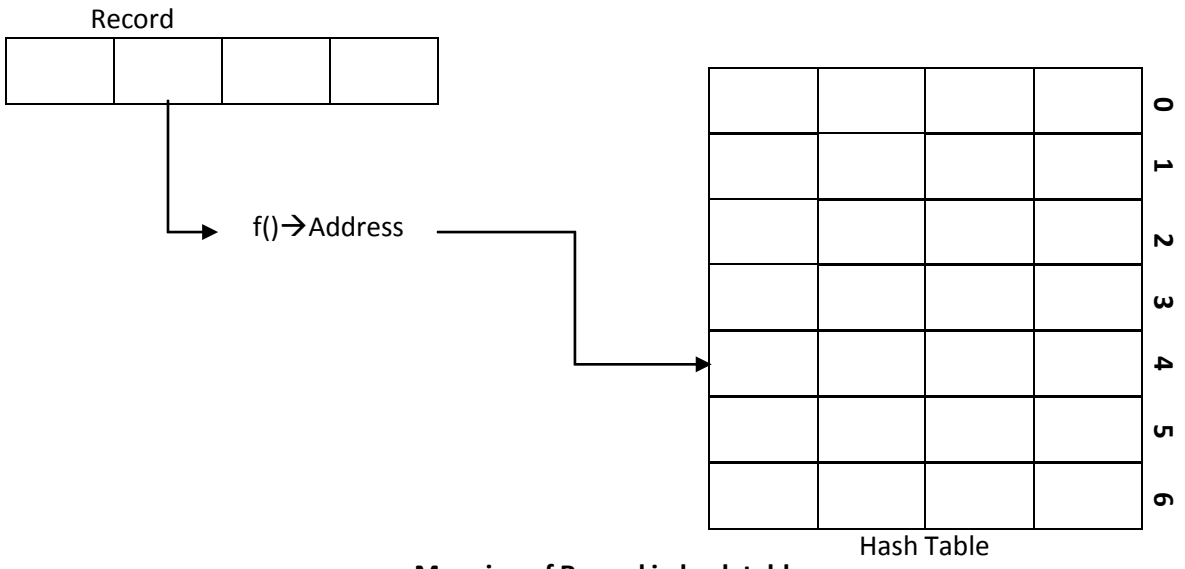
search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

Hashing

What is Hashing?

- Sequential search requires, on the average O(n) comparisons to locate an element. So many comparisons are not desirable for a large database of elements.
- Binary search requires much fewer comparisons on the average O (log n) but there is an additional requirement that the data should be sorted. Even with best sorting algorithm, sorting of elements require 0(n log n) comparisons.
- There is another widely used technique for storing of data called hashing. It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order (0(1)). In its worst case, hashing algorithm starts behaving like linear search.
- Best case timing behavior of searching using hashing = O(1)
- Worst case timing Behavior of searching using hashing = O(n)
- In hashing, the record for a key value "key", is directly referred by calculating the address from the key value. Address or location of an element or record, x, is obtained by computing some arithmetic function f. f(key) gives the address of x in the table.



Mapping of Record in hash table

Hash Table Data Structure:

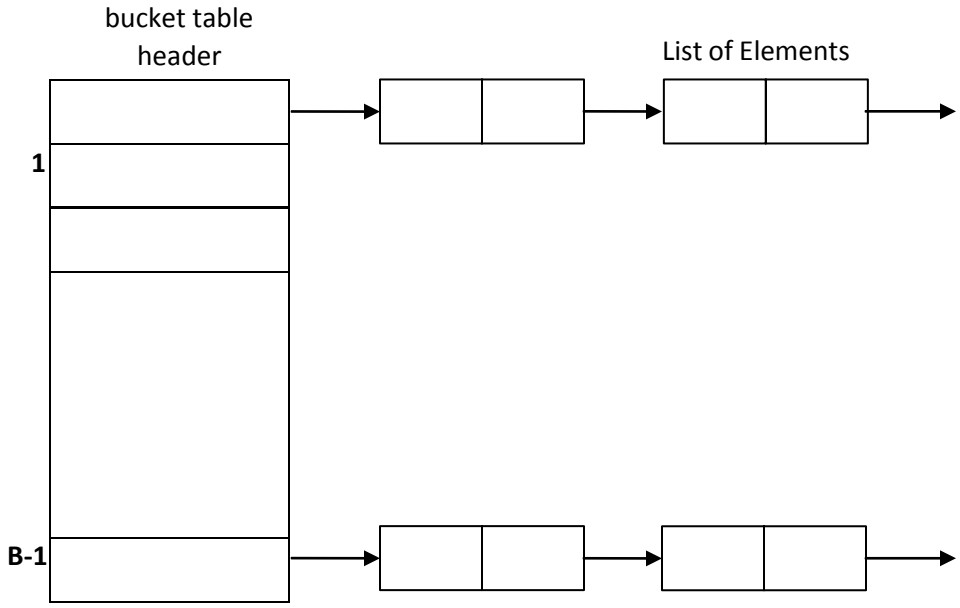
There are two different forms of hashing.

1. **Open hashing or external hashing**
   Open or external hashing, allows records to be stored in unlimited space (could be a hard disk). It places no limitation on the size of the tables.

2. Close hashing or internal hashing

Closed or internal hashing, uses a fixed space for storage and thus limits the size of hash table.

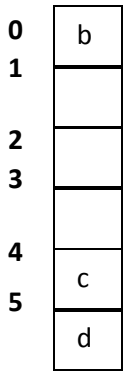
1. Open Hashing Data Structure



The open hashing data organization

- The basic idea is that the records [elements] are partitioned into B classes, numbered 0,1,2 … B-l
- A Hashing function f(x) maps a record with key n to an integer value between 0 and B-l.
- Each bucket in the bucket table is the head of the linked list of records mapped to that bucket.

2. Close Hashing Data Structure



- A closed hash table keeps the elements in the bucket itself.
- Only one element can be put in the bucket
- If we try to place an element in the bucket f(n) and find it already holds an element, then we say that a collision has occurred.
- In case of collision, the element should be rehashed to alternate empty location f1(x), f2(x), ... within the bucket table
- In closed hashing, collision handling is a very important issue.

Hashing Functions

Characteristics of a Good Hash Function

- A good hash function avoids collisions.
- A good hash function tends to spread keys evenly in the array.
- A good hash function is easy to compute.

Different hashing functions

1. Division-Method
2. Midsquare Methods
3. Folding Method
4. Digit Analysis
5. Length Dependent Method
6. Algebraic Coding
7. Multiplicative Hashing

1. Division-Method

- In this method we use modular arithmetic system to divide the key value by some integer divisor m (may be table size).
- It gives us the location value, where the element can be placed.
- We can write,
  L = (K mod m) + 1
  where L => location in table/file
  K => key value
  m => table size/number of slots in file
- Suppose, k = 23, m = 10 then
  L = (23 mod 10) + 1= 3 + 1=4, The key whose value is 23 is placed in 4th location.

2. Midsquare Methods

- In this case, we square the value of a key and take the number of digits required to form an address, from the middle position of squared value.
- Suppose a key value is 16, then its square is 256. Now if we want address of two digits, then you select the address as 56 (i.e. two digits starting from middle of 256).

3. Folding Method

- Most machines have a small number of primitive data types for which there are arithmetic instructions.
- Frequently key to be used will not fit easily in to one of these data types
- It is not possible to discard the portion of the key that does not fit into such an arithmetic data type

- The solution is to combine the various parts of the key in such a way that all parts of the key affect for final result such an operation is termed folding of the key.
- That is the key is actually partitioned into number of parts, each part having the same length as that of the required address.
- Add the value of each parts, ignoring the final carry to get the required address.
- This is done in two ways :
    - **Fold-shifting:** Here actual values of each parts of key are added.
        - Suppose, the key is : 12345678, and the required address is of two digits,
        - Then break the key into: 12, 34, 56, 78.
        - Add these, we get 12 + 34 + 56 + 78 : 180, ignore first 1 we get 80 as location
    - **Fold-boundary:** Here the reversed values of outer parts of key are added.
        - Suppose, the key is : 12345678, and the required address is of two digits,
        - Then break the key into: 21, 34, 56, 87.
        - Add these, we get 21 + 34 + 56 + 87 : 198, ignore first 1 we get 98 as location

4. Digit Analysis

- This hashing function is a distribution-dependent.
- Here we make a statistical analysis of digits of the key, and select those digits (of fixed position) which occur quite frequently.
- Then reverse or shifts the digits to get the address.
- For example, if the key is : 9861234. If the statistical analysis has revealed the fact that the third and fifth position digits occur quite frequently, then we choose the digits in these positions from the key. So we get, 62. Reversing it we get 26 as the address.

5. Length Dependent Method

- In this type of hashing function we use the length of the key along with some portion of the key j to produce the address, directly.
- In the indirect method, the length of the key along with some portion of the key is used to obtain intermediate value.

6. Algebraic Coding

- Here a n bit key value is represented as a polynomial.
- The divisor polynomial is then constructed based on the address range required.
- The modular division of key-polynomial by divisor polynomial, to get the address-polynomial.
- Let f(x) = polynomial of n bit key = $a_1 + a_2x + ... + a_nx^{n-1}$
- d(x) = divisor polynomial = $x^1 + d_1 + d_2x + ... + d_1x^{1-1}$
- then the required address polynomial will be f(x) mod d(x)

7. Multiplicative Hashing

- This method is based on obtaining an address of a key, based on the multiplication value.

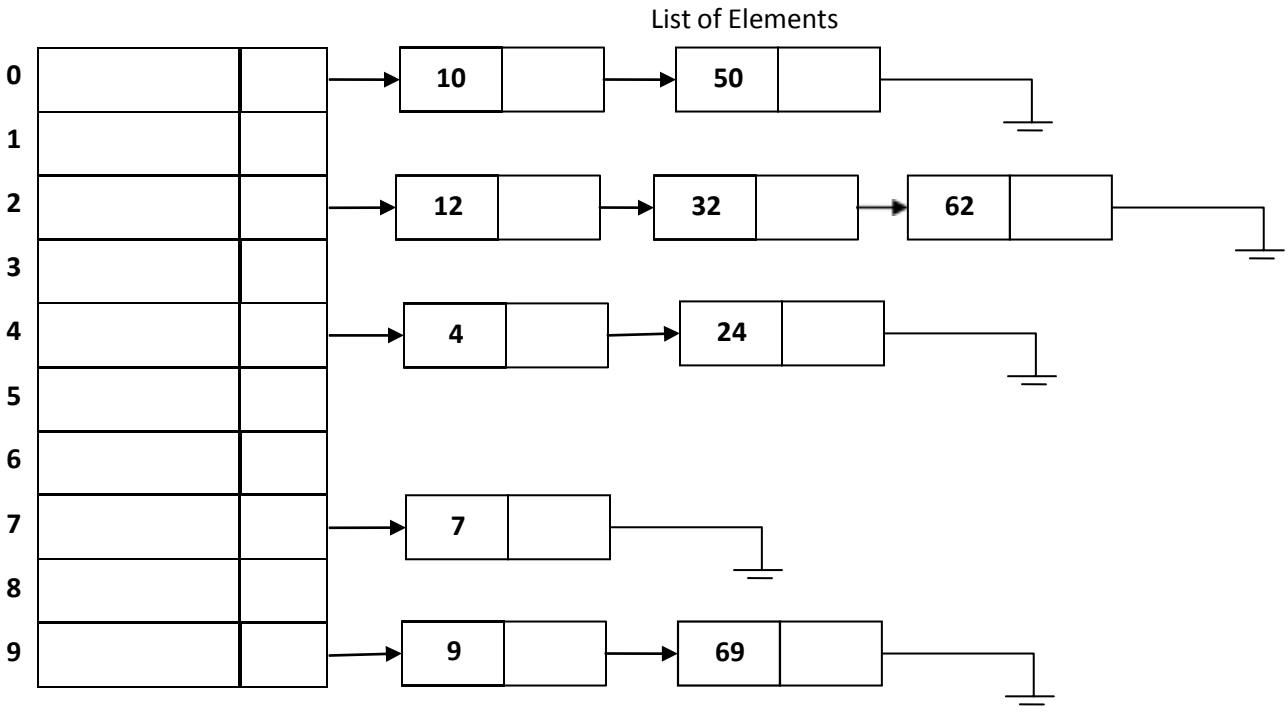
- If k is the non-negative key, and a constant c, (0 < c < 1), compute kc mod 1, which is a fractional part of kc.
- Multiply this fractional part by m and take a floor value to get the address
- $m\ (kc\ mod\ 1)$ J
- 0 < h (k) < m

Collision Resolution Strategies (Synonym Resolution)

- Collision resolution is the main problem in hashing.
- If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved.
- There are several strategies for collision resolution. The most commonly used are :
    1. **Separate chaining** - used with open hashing
    2. **Open addressing** - used with closed hashing

1. Separate chaining

- In this strategy, a separate list of all elements mapped to the same value is maintained.
- Separate chaining is based on collision avoidance.
- If memory space is tight, separate chaining should be avoided.
- Additional memory space for links is wasted in storing address of linked elements.
- Hashing function should ensure even distribution of elements among buckets; otherwise the timing behavior of most operations on hash table will deteriorate.

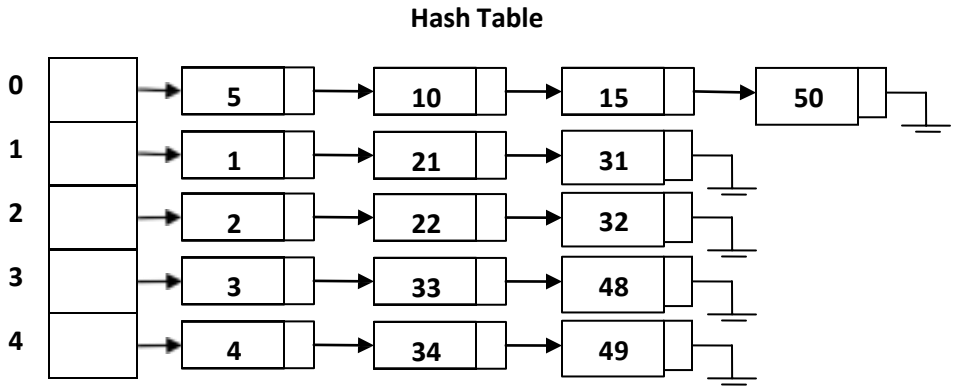


A Separate Chaining Hash Table

Example : The integers given below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function. 1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50

An element can be mapped to a location in the hash table using the mapping function **key % 10**.

Hash Table Location	Mapped element
0	5, 10, 15, 50
1	1, 21, 31
2	2, 22, 32
3	3, 33, 48
4	4, 34, 49



---

2. Open Addressing

- Separate chaining requires additional memory space for pointers. Open addressing hashing is an alternate method of handling collision.
- In open addressing, if a collision occurs, alternate cells are tried until an empty cell is found.
    a. Linear probing
    b. Quadratic probing
    c. Double hashing.

a) Linear Probing

- In linear probing, whenever there is a collision, cells are searched sequentially (with wraparound) for an empty cell.
- Fig. shows the result of inserting keys {5,18,55,78,35,15} using the hash function **(f(key)= key%10)** and linear probing strategy.

	Empty Table	After 5	After 18	After 55	After 78	After 35	After 15
0							15
1							
2							
3							
4							
5		5	5	5	5	5	5
6				55	55	55	55
7						35	35
8			18	18	18	18	18
9					78	78	78

- Linear probing is easy to implement but it suffers from "**primary clustering**"

- When many keys are mapped to the same location (clustering), linear probing will not distribute these keys evenly in the hash table. These keys will be stored in neighborhood of the location where they are mapped. This will lead to clustering of keys around the point of collision

b) Quadratic probing

- One way of reducing "primary clustering" is to use quadratic probing to resolve collision.
- Suppose the "key" is mapped to the location j and the cell j is already occupied. In quadratic probing, the location j, (j+1), (j+4), (j+9), ... are examined to find the first empty cell where the key is to be inserted.
- This table reduces primary clustering.
- It does not ensure that all cells in the table will be examined to find an empty cell. Thus, it may be possible that key will not be inserted even if there is an empty cell in the table.

c) Double Hashing

- This method requires two hashing functions f1 (key) and f2 (key).
- Problem of clustering can easily be handled through double hashing.
- Function f1 (key) is known as primary hash function.
- In case the address obtained by f1 (key) is already occupied by a key, the function f2 (key) is evaluated.
- The second function f2 (key) is used to compute the increment to be added to the address obtained by the first hash function f1 (key) in case of collision.
- The search for an empty location is made successively at the addresses f1 (key) + f2(key), f1 (key) + 2f2 (key), f1 (key) + 3f2(key),...

What is File?

- A file is a collection of records where a record consists of one or more fields. Each contains the same sequence of fields.
- Each field is normally of fixed length.
- A sample file with four records is shown below:

Name	Roll No.	Year	Marks
AMIT	1000	1	82
KALPESH	1005	2	54
JITENDRA	1009	1	75
RAVI	1010	1	79

- There are four records
- There are four fields (Name, Roll No., Year, Marks)
- Records can be uniquely identified on the field 'Roll No.' Therefore, Roll No. is the key field.
- A database is a collection of files.
- Commonly, used file organizations are :
  1. Sequential files
  2. Relative files
  3. Direct files
  4. Indexed Sequential files
  5. Index files
- Primitive Operations on a File :
  1. Creation
  2. Reading
  3. Insertion
  4. Deletion
  5. Updating
  6. Searching

Sequential Files

It is the most common type of file. In this type of file:

- A fixed format is used for record.
- All records are of the same length.
- Position of each field in record and length of field is fixed.
- Records are physically ordered on the value of one of the fields - called the ordering field.

Block 1

Name	Roll No.	Year	Marks
AMIT	1000	1	82
KALPESH	1005	2	54
JITENDRA	1009	1	75
RAVI	1010	1	79

Block 2

NILESH	1011	2	89

Some blocks of an ordered (sequential) file of students records with Roll no. as the ordering field

Advantages of sequential file over unordered files :

- Reading of records in order of the ordering key is extremely efficient.
- Finding the next record in order of the ordering key usually, does not require additional block access. Next record may be found in the same block.
- Searching operation on ordering key is must faster. Binary search can be utilized. A binary search will require $\mathbf{\log_2 b}$ block accesses where b is the total number of blocks in the file.

Disadvantages of sequential file :

- Sequential file does not give any advantage when the search operation is to be carried out on non-ordering field.
- Inserting a record is an expensive operation. Insertion of a new record requires finding of place of insertion and then all records ahead of it must be moved to create space for the record to be inserted. This could be very expensive for large files.
- Deleting a record is an expensive operation. Deletion too requires movement of records.
- Modification of field value of ordering key could be time consuming. Modifying the ordering field means the record can change its position. This requires deletion of the old record followed by insertion of the modified record.

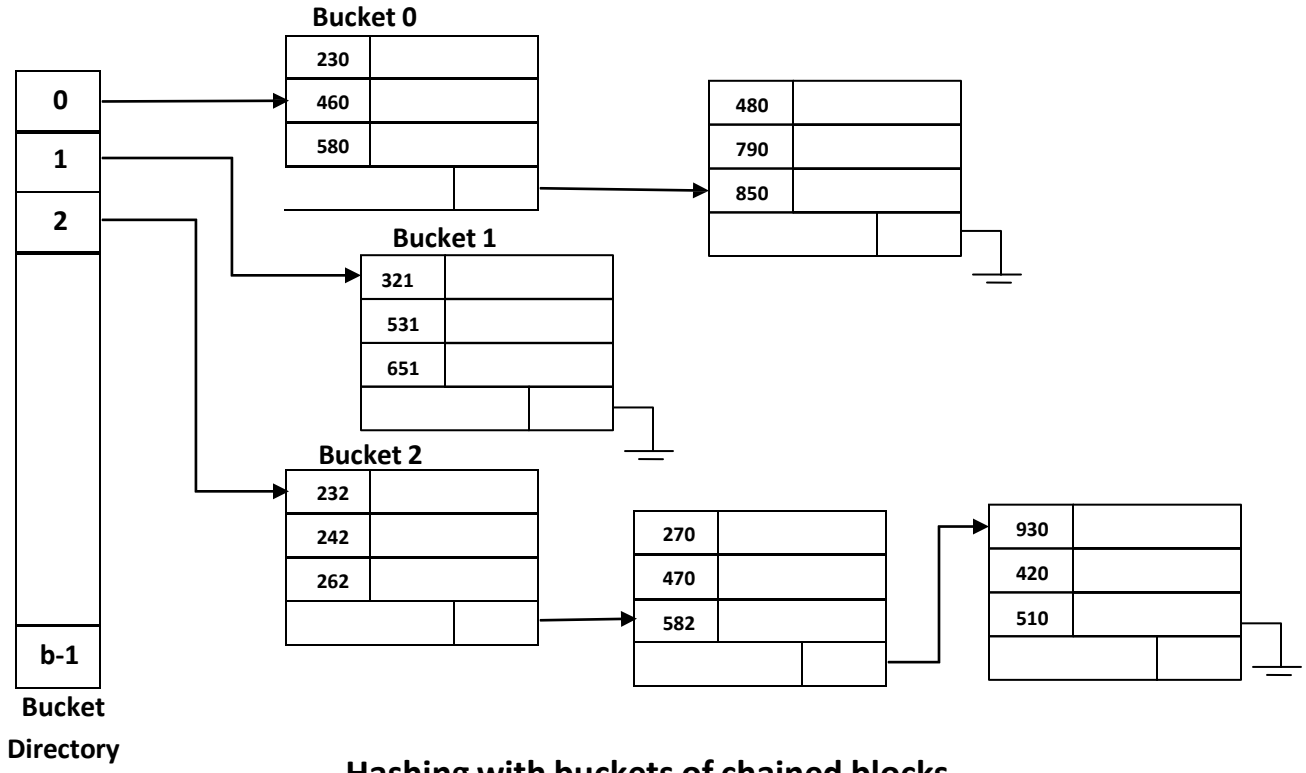
Hashing (Direct file organization):

- It is a common technique used for fast accessing of records on secondary storage.
- Records of a file are divided among buckets.
- A bucket is either one disk block or cluster of contiguous blocks.
- A hashing function maps a key into a bucket number. The buckets are numbered 0, 1,2...b-1.
- A hash function f maps each key value into one of the integers 0 through b - 1.
- If x is a key, f(x) is the number of bucket that contains the record with key x.

- The blocks making up each bucket could either be contiguous blocks or they can be chained together in a linked list.
- Translation of bucket number to disk block address is done with the help of bucket directory. It gives the address of the first block of the chained blocks in a linked list.
- Hashing is quite efficient in retrieving a record on hashed key. The average number of block accesses for retrieving a record.

$$= \mathbf{1\ (bucket\ directory)} + \frac{No\ of\ records}{No\ of\ buckets \times No\ of\ records\ per\ block}$$

- Thus the operation is b times faster (b = number of buckets) than unordered file.
- To insert a record with key value x, the new record can added to the last block in the chain for bucket f(x). If the record does not fit into the existing block, record is stored in a new block and this new block is added at the end of the chain for bucket f(x).
- A well designed hashed structure requires two block accesses for most operations

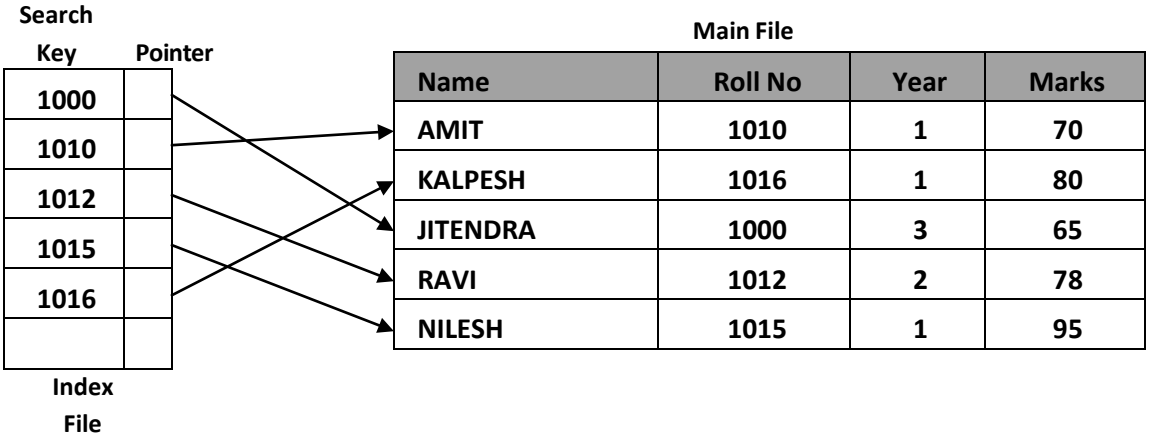


Hashing with buckets of chained blocks

Indexing

- Indexing is used to speed up retrieval of records.

- It is done with the help of a separate sequential file. Each record of in the index file consists of two fields, a key field and a pointer into the main file.
- To find a specific record for the given key value, index is searched for the given key value.
- Binary search can used to search in index file. After getting the address of record from index file, the record in main file can easily be retrieved.



- Number of blocks required to store the file = $\frac{1024\ x\ 128}{2048} = 64$
- Number of block accesses for searching a record = $\log_2 64 = 6$
- Suppose, we want to construct an index on a key field that is V = 4 bytes long and the block pointer is P = 4 bytes long.
- A record of an index file is of the form <V;, Pj> and it will need 8 bytes per entry.
- Total Number of index entries = 1024
- Number of blocks b' required to store the file = $\frac{1024\ x\ 8}{2048} = 4$
- Number of block accesses for searching a record = $\log_2 4 = 2$

- With indexing, new records can be added at the end of the main file. It will not require movement of records as in the case of sequential file. Updation of index file requires fewer block accesses compare to sequential file

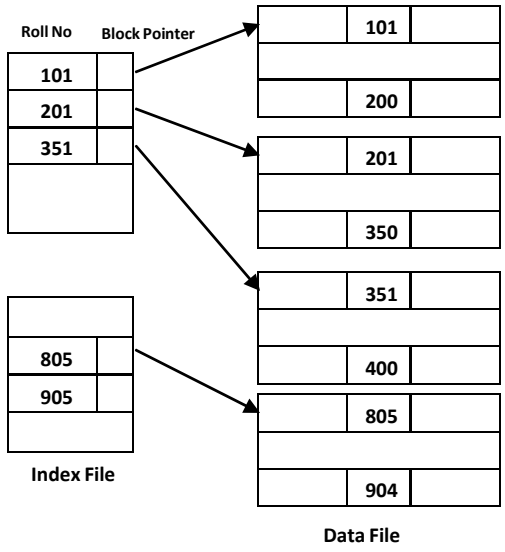
Types of Indexes:

1. Primary indexes
2. Clustering indexes
3. Secondary indexes

Primary Indexes (Indexed Sequential File):

- An indexed sequential file is characterized by
    - Sequential organization (ordered on primary key)
    - Indexed on primary key
- An indexed sequential file is both ordered and indexed.
- Records are organized in sequence based on a key field, known as primary key.
- An index to the file is added to support random access. Each record in the index file consists of two fields: a key field, which is the same as the key field in the main file.
- Number of records in the index file is equal to the number of blocks in the main file (data file) and not equal to the number of records in the main file (data file).
- To create a primary index on the ordered file shown in the Fig. we use the rollno field as primary key. Each entry in the index file has rollno value and a block pointer. The first three index entries are as follows.

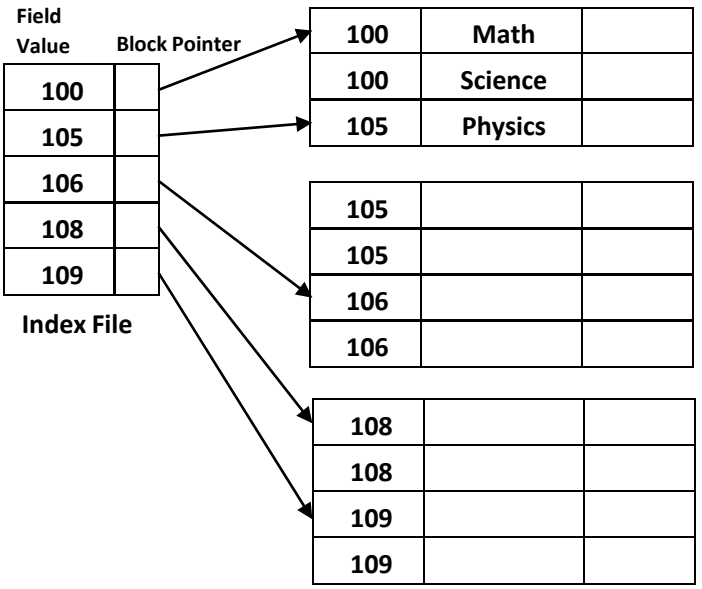
  - <101, address of block 1>
  - <201, address of block 2>
  - <351, address of block 3>
- Total number of entries in index is same as the number of disk blocks in the ordered data file.
- A binary search on the index file requires very few block accesses



Primary Index on ordering key field roll number

Clustering Indexes

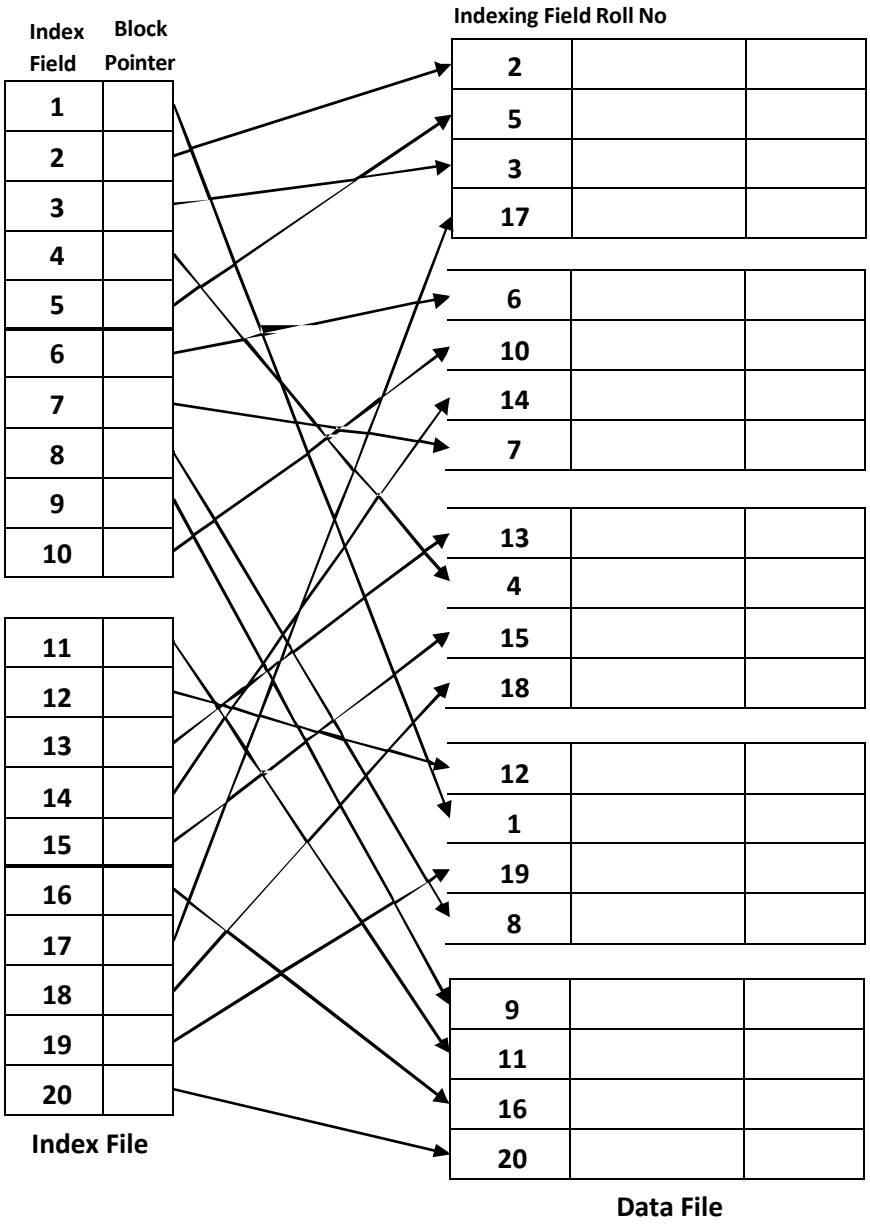
- If records of a file are ordered on a non-key field, we can create a different type of index known as clustering index.
- A non-key field does not have distinct value for each record.
- A Clustering index is also an ordered file with two fields.



Example of clustering index on roll no

Secondary indexes (Simple Index File)

- While the hashed, sequential and indexed sequential files are suitable for operations based on ordering key or the hashed key. Above file organizations are not suitable for operations involving a search on a field other than ordering or hashed key.
- If searching is required on various keys, secondary indexes on these fields must be maintained. A secondary index is an ordered file with two fields.
    - Some non-ordering field of the data file.
    - A block pointer
- There could be several secondary indexes for the same file.
- One could use binary search on index file as entries of the index file are ordered on secondary key field. Records of the data files are not ordered on secondary key field.
- A secondary index requires more storage space and longer search time than does a primary index.
- A secondary index file has an entry for every record whereas primary index file has an entry for every block in data file.
- There is a single primary index file but the number of secondary indexes could be quite a few.



A secondary index on a non-ordering key field

B - Tree Data-structure

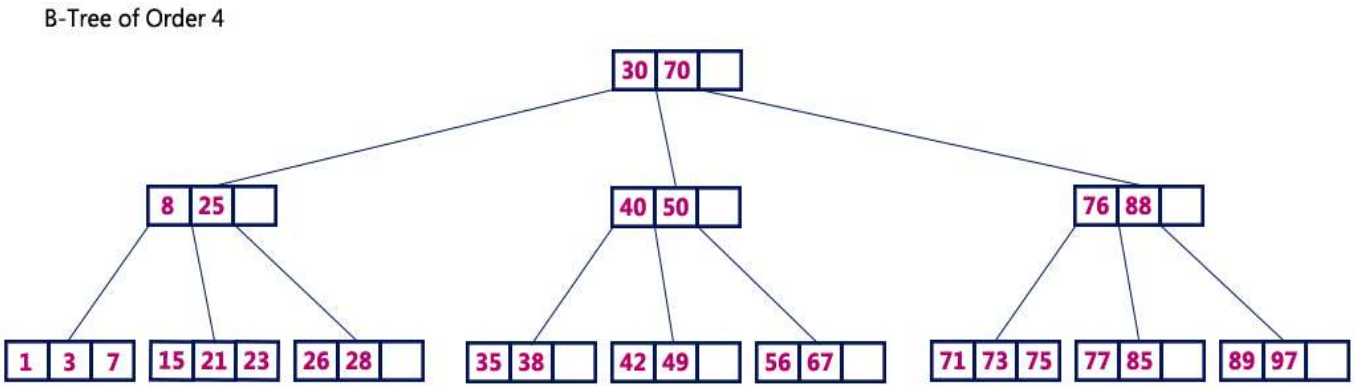
In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children. But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children. B-Tree was developed in the year 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree.

B-Tree can be defined as follows...
B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children. Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

B-Tree of Order m has the following properties...

- Property #1 - All leaf nodes must be at same level.
- Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
- Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
- Property #4 - If the root node is a non leaf node, then it must have at least 2 children.
- Property #5 - A non leaf node with n-1 keys must have n number of children.
- Property #6 - All the key values in a node must be in Ascending Order.

For example,
B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.



Operations on a B-Tree
The following operations are performed on a B-Tree...

1. **Search** **2.Insertion** **3.Deletion**

The search operation in B-Tree is similar to the search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left sub-tree or right sub-tree). In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has. In a B-Tree, the search operation is performed with **O(log n)** time complexity.

The search operation is performed as follows...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with first key value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that key value.
- **Step 5 -** If search element is smaller, then continue the search process in left sub-tree.
- **Step 6 -** If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
- **Step 7 -** If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new key Value is always attached to the leaf node only.

The insertion operation is performed as follows...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.
- **Step 3 -** If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
- **Step 4 -** If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
- **Step 5 -** If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
- **Step 6 -** If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example: Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

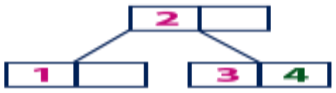
Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.

insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

insert(6)

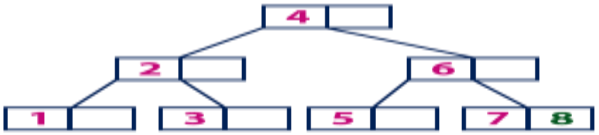
Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.

insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.

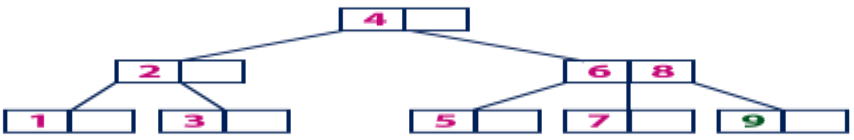
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



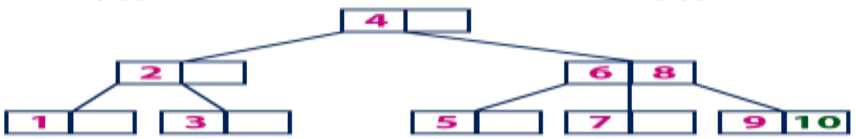
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



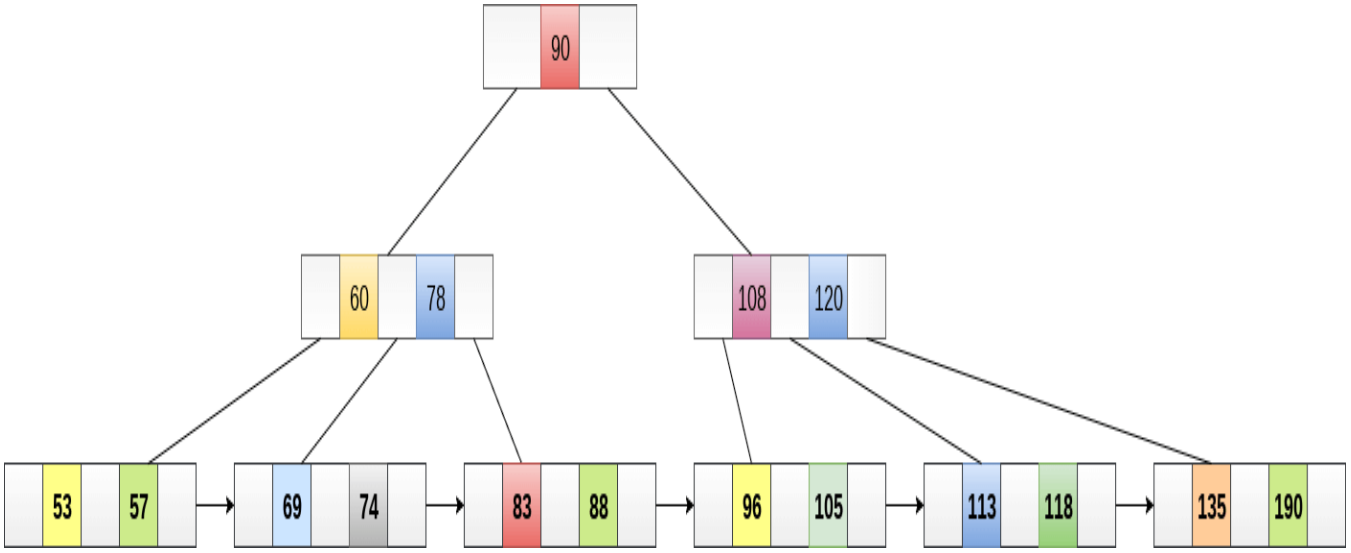
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes whileinternal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

Insertion in B+ Tree

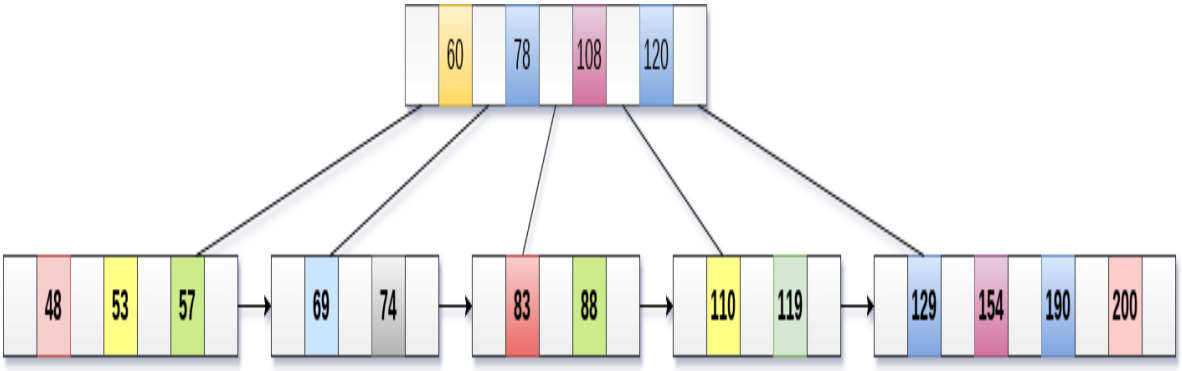
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

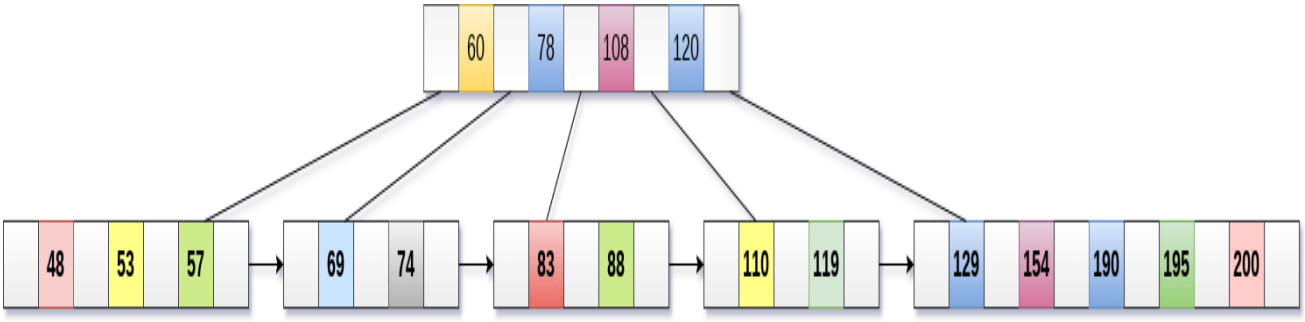
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

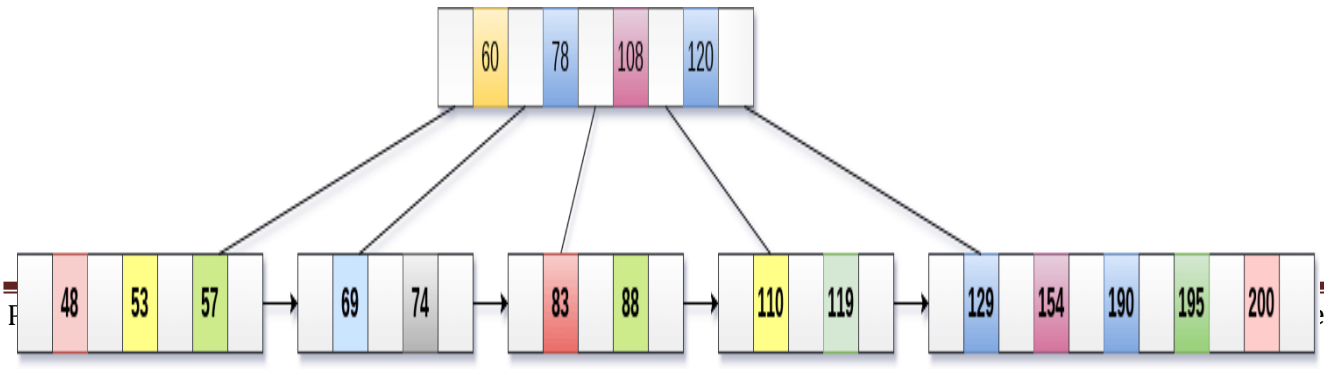
Insert the value 195 into the B+ tree of order 5 shown in the following figure.



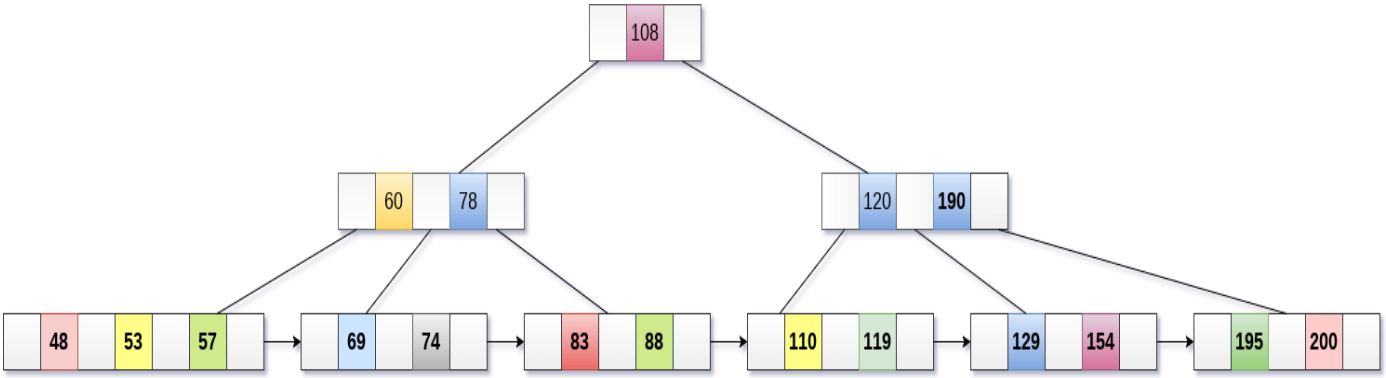
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

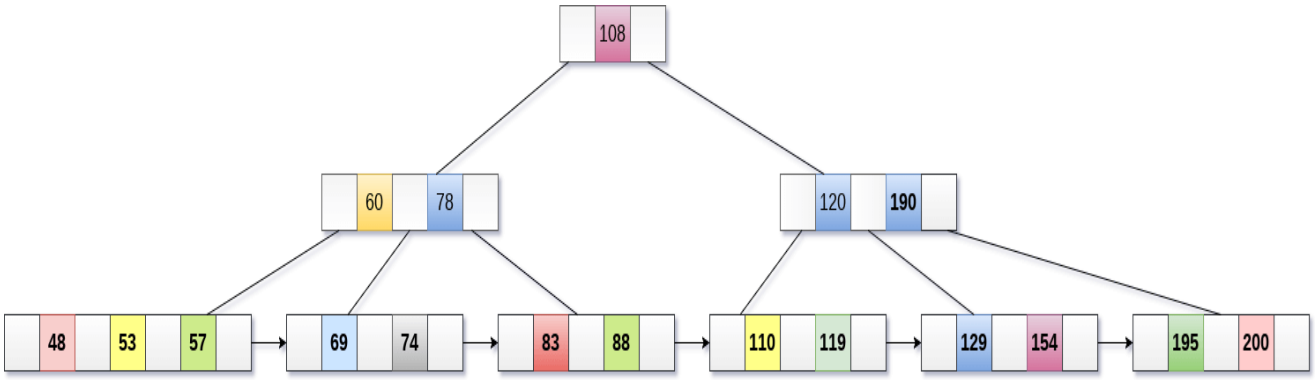
Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

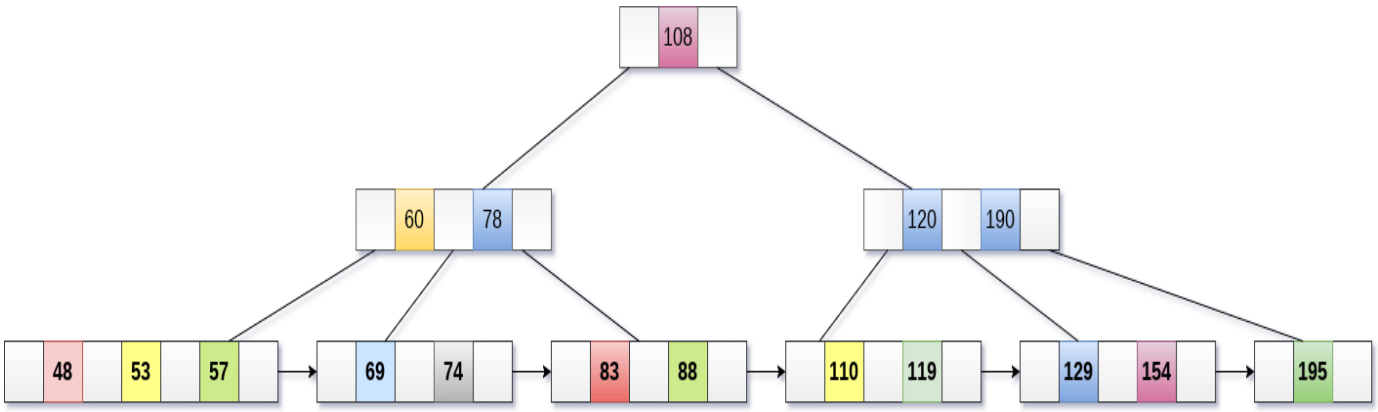
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

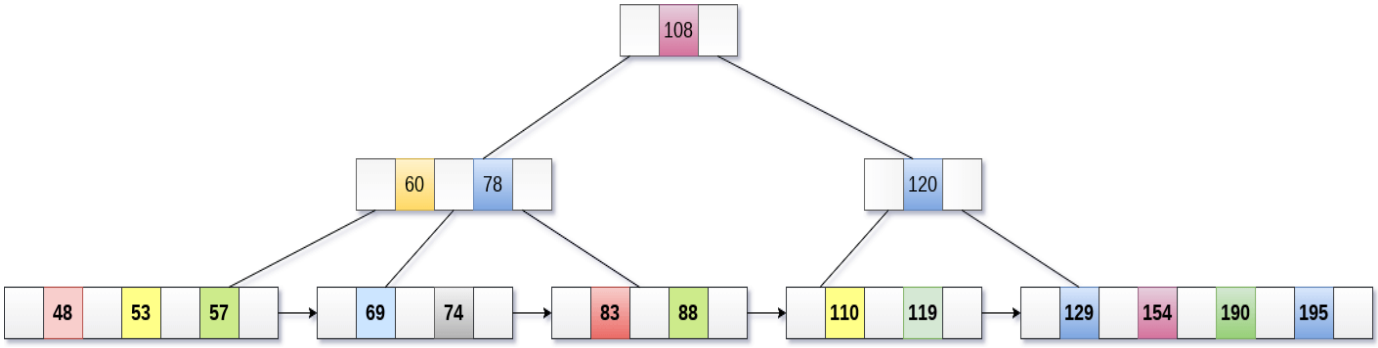
Delete the key 200 from the B+ Tree shown in the following figure.



200 is present in the right sub-tree of 190, after 195. delete it.

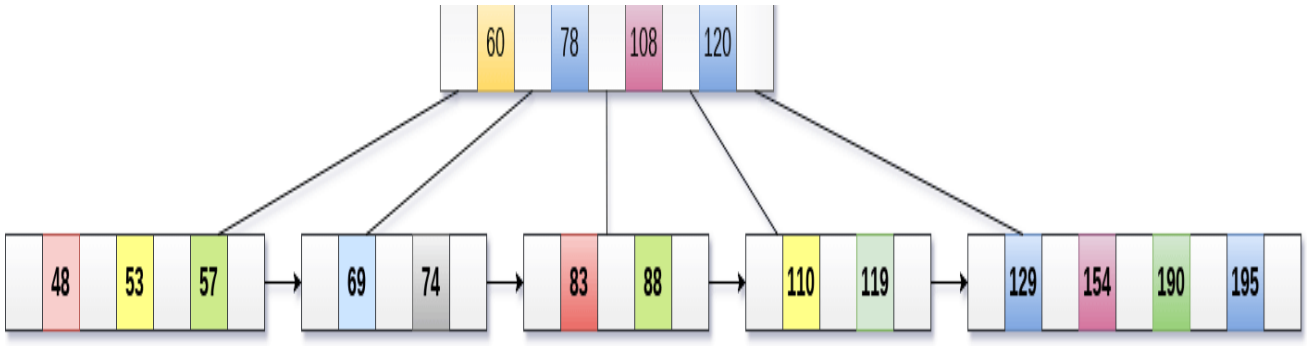


Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

Now, the height of B+ tree will be decreased by 1.



Why B+ trees?

- B+ trees store the records which later can be fetched in an equal number of disk accesses.
- The height of the B+ tree **remains balanced** and very less as when compared to B trees even if the number of records store in them is the same.
- Having less number of levels makes the accessing of records very easy.
- As the leaf nodes are connected with each other **like a linked list**, we can easily search elements in sequential manners.

B Tree VS B+ Tree

S.NO	B Tree	B+ Tree
1	Search keys cannot be repeatedly stored.	Redundant search keys can be present.
2	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes
3	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
4	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

CYLLENDRICAL SURFACE INDEXING

Index Techniques

- A directory is a collection of indexes. It may contain one index for every key or an index for only some of the keys. Some of the indexes may be **dense** (i.e. contain an entry for every record) while others may be **non-dense** (i.e. contain an entry for only some of the records).
- In some cases, all indexes may be integrated into one large index.
- **Index is a collection of the pairs of the form (key value, address).**
- E.g. Employee record can have indexes (800, a1) a1 being the disk address, assuming all the key values being unique.
- Functions to be performed on index are: searching for a key, insert a new pair, delete a pair, and modify existing entry.

1. Cylinder-surface indexing

- Simplest type of index organization. It is useful only for the primary key index of a sequentially ordered file. It assumes that records are stored sequentially in increasing order of primary key. The index consists of a cylinder index and several surface indexes. If the file requires c cylinders for storage then the cylinder index contains c entries.
- Associated with each of the c cylinders is a surface index. If the disk has s usable surfaces then each surface index has s entries. The ith entry in the surface index for cylinder j, is the value of the largest key on the jth track on the ith surface. The total no. of surface index entries is therefore, c·s.
- A search for a record with a particular key value X is carried out by first reading into memory the cylinder index. The cylinder index is searched (use binary search coz no. high ~ 100) to determine which cylinder possibly contains desired record. Then surface index corresponding to that cylinder is retrieved (sequential search coz no. small ~10) from the disk. Then track is read in and searched for record with key X.
- Total no. of disk accesses needed for retrieval is three (one to access the cylinder index, one for surface index and one to get the track of records.)

- When track sizes are very large then it is not feasible to read entire track, in such cases an extra level of indexing: sector index be needed. When the file extends to several disks a disk index is also maintained.
- The method of maintaining a file and index is called ISAM (indexed sequential access method). It is most popular and simplest file organization in use for single key files.

2. Hashed indexes

- Hash functions and overflow handling techniques are used for hashing. Since the index is to be maintained on disk and disk access times are generally several orders of magnitude larger than internal memory access times, much consideration is given to hash table design and the choice of an overflow handling technique.
- Overflow handling techniques:

1. rehashing
2. open addressing (*random*[f(i)=random()], *quadratic, linear*)
3. chaining

- Expected no. of bucket accesses when s=1 is roughly same for method rehashing, random and quadratic probing. Since the hash table is on a disk and these overflow techniques tend to randomize the use of hash table, we can expect each bucket access to take almost the maximum seek time. In case of linear probing however overflow buckets are adjacent to home bucket so their retrieval will require minimum seek time. While using chaining we can minimize the tendency to randomize use of overflow area by designating certain tracks as overflow tracks for particular buckets.

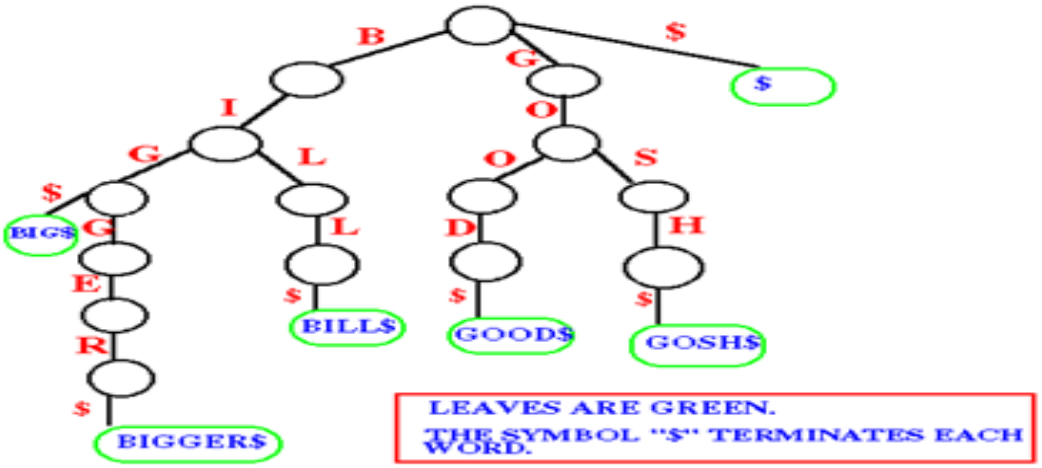
3. Tree indexing-B Trees

- Using m-way search tree can minimize the search time, rather than using AVL trees.
- B trees are used for indexing

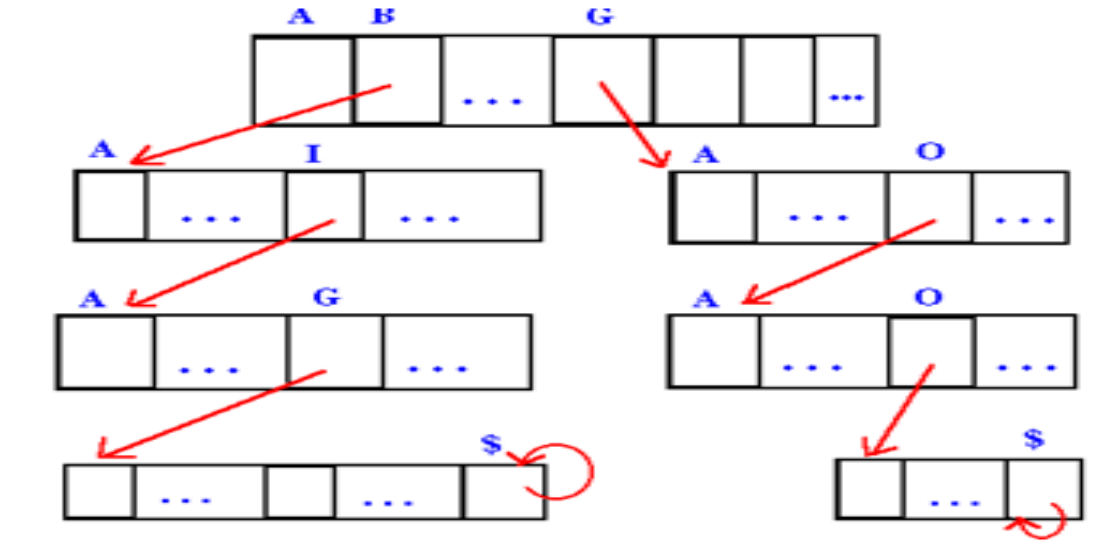
4. Trie indexing

- An index structure that is particularly useful when key values are of varying size is trie.
- A **trie** is a tree of degree m>=2 in which the branching at any level is determined not by the entire key value but by only a portion of it.

- The trie contains two types of nodes. First type called the **branch node** and **second information node.**
- Searching a trie for a key value X requires breaking up X into its consequent characters and following the branching patterns determined by these characters.
- The trie is a data structure that can be used to do a fast search in a large text. For example, we can think of the Oxford English dictionary which contains several gigabytes of text. The Oxford dictionary is a static structure because we do not want to add or delete any items. However, searching for an item in the dictionary is very important. Also, searching for a string should be efficient because overlapping of strings can occur.



In this figure, the strings either start with B or G. Therefore, the root of the trie is connected to 3 edges called B, G and $.



Trie representation for words BIG and GOO using array of pointers