

DATABASE STORAGE AND INDEXING*

Nguyen Kim Anh

This work is produced by OpenStax-CNX and licensed under the Creative Commons Attribution License 3.0[†]

In previous lectures, we have emphasized the higher-level models of database. The goal of database system is to simplify and facilitate access to data. Users of the system should not be burdened with the physical details of the implementation of the systems. However, the way data is organized in the database storage media has important effects on the performance of the whole system.

Therefore, in this lecture, we will give a brief introduction of the organization of database in storage and various techniques for accessing data efficiently.

1 1. Introduction of Database Storage

Databases are stored physically as files of records on some storage medium. This section will deal with the overview of available storage media then briefly describes the magnetic storage devices.

1.1 1.1 Physical Storage Media

The collection of data in a database system must be stored physically on some storage medium. These storage media are classified by the speed with which data can be accessed, by the cost per unit of data to buy the medium and by the medium's reliability. There are several typical storage media available:

- **Cache memory:** Cache memory is a primary storage media like the main memory. Data on these devices can be directly processed by the Central Processing Unit (CPU). Cache memory is the fastest but is also the most expensive form of storage.
- **Main memory:** Data that are available to be operated are stored in main memory. The machine's instructions operate on main memory. Main memory is lower cost and also lower speed in comparison with cache memory. However, main memory is generally too small to store the entire database. Main memory is volatile that means contents of main memory are lost in case of power outage.
- **Flash memory :** This memory is non-volatile and has fast access speed. However, the drawback of this is the complication when writing data to flash memory. Data in flash memory cannot be overwritten directly. To overwrite memory that has been written already, we have to erase an entire block of memory at once, it is then ready to be written again.
- **Magnetic-disk storage:** This is the primary medium for long-term storage of data. This is a type of secondary storage which usually has large capacity, is low cost and is volatile. Data in secondary storage such as magnetic disk cannot be accessed directly by CPU, first it must be copied into primary storage.
- **Optical storage:** The most popular optical storage is CD-ROM. In this device data are stored optically and are read by laser. CD-ROMs contain prerecorded data that cannot be overwritten. Optical storage has gigabytes in capacity and lasts much longer than magnetic disk. Optical jukebox memories use an array of CD-ROM platters which are loaded onto drives on demand.

*Version 1.1: Jul 8, 2009 1:19 am -0500

[†]<http://creativecommons.org/licenses/by/3.0/>

- **Tape storage:** This storage is used for backup and archival data. Although magnetic tape is much cheaper than disks, access to data is much slower because tape must be accessed sequentially from the beginning. Tape jukeboxes are used to hold large collections of data and is becoming a popular tertiary storage.

In the next section, we will give brief revision of the physical characteristics of magnetic disk the most popular online storage for database system.

1.2 1.2 Magnetic Disk Devices

Magnetic disks are used for storing large amount of data. The capacity of disk is the number of bytes it can store.

Disk platter has a flat circular shape. Its two surface are covered with magnetic material and data is recorded on the surface. The disk surface is divided into tracks, each track is a circle of distinct diameter. Track is subdivided into blocks (sectors). Depending on the disk type, block size varies from 32 bytes to 4096 bytes. There may be hundreds of concentric tracks on a disk surface containing thousands of sectors. In disk packs, tracks with the same diameter on the various surfaces form a cylinder.

A disk typically contains many platters.

A disk is a random access addressable device. Transfer of data between main memory and disk takes place in units of disk block.

The hardware mechanism that reads or writes a block is the disk read/write head (disk drive). A disk or disk pack is mounted into the disk drive, which includes a motor to rotate the disks. A read/write head includes the electronic component attached to a mechanical arm. The arm moves the read/write heads, positions them precisely over the cylinder or tracks specified in a block address.

2 2. Placing File records on Disks

A file is organized logically as a sequence of records. Each record consists of a collection of related data values or items which corresponds to a particular field of the record. In database system, a record usually represents an entity. For example, an EMPLOYEE record represents an employee entity and each item in this record specifies the value of an attribute of that employee, such as Name, Address, Birthdate etc.

In most cases, all records in the file have the same type. That means every record has the same fields, each field has fixed length data type. If all records have the same size (in bytes) then the file is file of fixed-length records. If records in a file have different sizes, the file is made up of variable length-records. In this lecture, we focus on only fixed-length record file.

The records of a file must be allocated to disk blocks in some ways. When a record size is much smaller than block size a block can contain several records. However, unless the block size happens to be a multiple of record size, some records might cross block boundaries. In this situation, a part of a record is stored in one block and the other part is in another block. It would thus require two block accesses to read or write such a record. This organization is called spanned.

If records are not allowed to cross block boundaries, we have the unspanned organization. In this lecture, from now on, we assume that records in a file are allocated in the unspanned manner.

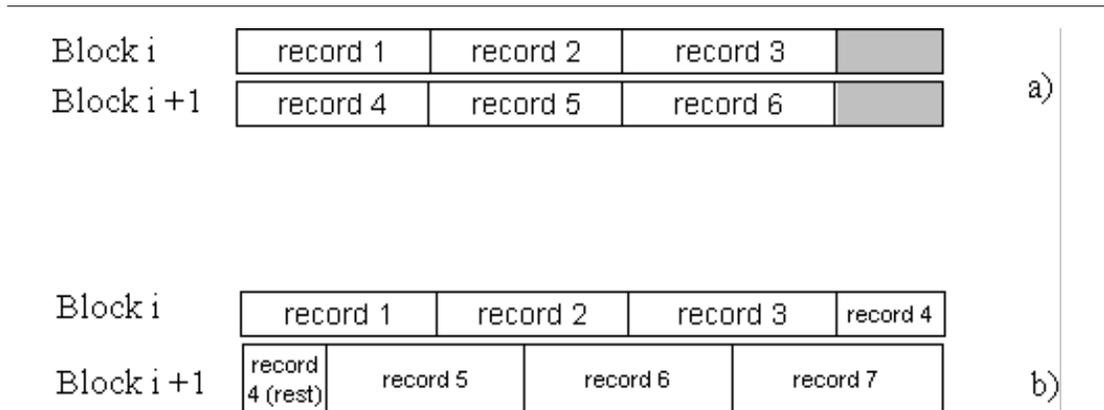


Figure 1: Records organization. a) Unspanned. b) Spanned

To allocate the blocks of a file on disk, we consider several techniques. In contiguous allocation, the file blocks are allocated to consecutive disk blocks. In linked allocation, each file block contains a pointer to the next file block. Another possibility is index allocation in which one or more index blocks contains pointers to the actual file blocks.

3.3. Basic Organizations of Records in Files

In this section, we will examine several ways of organizing a collection of records in a file on the disk and discuss the access methods that can be applied to each method.

3.1.3.1 Heap Files Organization.

In this organization, any record can simply be placed in the file in the order in which they are inserted. That means there is no ordering of records, a new record is always inserted at the end of the file. Therefore, this is sometimes called the Unordered File organization to differentiate from the Ordered File organization which will be presented in the next section.

In the below figure, we can see a sample of heap file organization for EMPLOYEE relation which consists of 8 records stored in 3 contiguous blocks, each block can contain at most 3 records.

	Name	EID	Address	Birthdate	Salary
block 1	Adams John
	Melinda, Perkin				
	Raymond, Wong				
block 2	Alan, Delon				
	Bill, Clinton				
	Nancy, Davies				
block 3	Jay, Shana				
	Son, Nguyen				

Figure 2: Sample Heap File Organization for EMPLOYEE relation

3.1.1 Operations on Heap Files

- Search for a record

Given a value to used as the condition to find a record. In order to find such record with that value, we need to scan the whole file (do linear search) or search half of the file on average. This operation is not efficient if the file is large, data on that file are stored in a large number of disk block.

- Insert a new record

Insertion into heap file is very simple. The new record is placed right after the last record of the file. We assume that

- Delete an existing record

To delete a record, the system must first search for the records and delete it from the block. Deleting records in this way may lead to waste storage space because this leave unused space in the disk blocks. Another technique used for record deletion is specifying a deletion marker for each record. Instead of remove the record physically from the block, the system marks the record to be deleted by setting deletion marker to a certain value. The marked records will not be considered in the next search. The system need to have a periodic reorganization of the file to reclaim the space of deleted records.

- Update an existing record

To update a record, at first we need to do a search to allocate the blocks, copy the blocks to buffer, make changes in the record then rewrite the blocks to disk.

Example: For the EMPLOYEE heap file organization, the file after inserting a record with employee's name Mary Ann Smith is

	Name	EID	Address	Birthdate	Salary
block1	Adams John				
	Melinda, Perkin				
	Raymond, Wong				
block 2	Alan, Delon				
	Bill, Clinton				
	Nancy, Davies				
block 3	Jay, Shana				
	Son, Nguyen				
	Mary Ann Smith				

Figure 3: EMPLOYEE heap file after insertion

The file after deleting records of Raymond Wong and reorganizing the file is

	Name	EID	Address	Birthdate	Salary
block 1	Adams John				
	Melinda, Perkin				
	Alan, Delon				
block 2	Bill, Clinton				
	Nancy, Davies				
	Jay, Shana				
block 3	Son, Nguyen				
	Mary Ann Smith				

Figure 4: EMPLOYEE heap file after deletion

3.2 3.2 Ordered Files Organization

This organization is sometimes called the sequential file. In this organization, the records of a file are ordered on disk based on the values of one field which is called ordering field. The ordering field might also be the

key field of the file, if so we got the ordering key for the file.

ordering field		Name	EID	Address	Birthdate	Salary
block 1		Adams, John				
		Alan, Delon				
		Bill, Clinton				
block 2		Jay, Shana				
		Mary Ann, Smith				
		Melinda, Perkin				
block 3		Nancy, Davies				
		Son, Nguyen				

Figure 5: EMPLOYEE relation in sorted file with Name is the ordering field

The ordered file organization allows records to be read in sorted order of ordering field that can be useful for display purpose. In addition, for a search that involves the ordering field in search condition, a faster search technique can be used rather than using linear search as in unordered file. It is difficult, however to maintain physical sequential order as records are inserted and deleted since it is costly to move many records as a results of a single insertion or deletion.

3.2.1 Operations on Ordered Files

- Search for a record given the value K of ordering key field of the record: Firstly, we need to search for the block that might contains the record by using binary search on block algorithm as below:
 - Assume the file is stored in n contiguous blocks numbering from 1 to n . Records in a file are ordered by ascending value of their ordering key field. This binary search can be describe briefly as comparing K with the value of ordering key field of the records in the middle block b .
 - if $K < K'$ which is the value of ordering key field of first record in b then search in blocks $[1, b-1]$
 - if $K > K''$ which is the value of ordering key field of the last record in b then search in blocks $[b+1, n]$.
 - else, record with value K is in the block b , go and find the record.
- Insert a new record with value of ordering key field K , we make the following steps:
 - Find the block on that the new record will be stored (depends on the value K)
 - Make space in the block to insert the record in the correct position. This step is time consuming because, on average, we need to read and rewritten half of the file blocks since the records are moved among them.

To make insertion more efficient, we can used the overflow blocks. After locating the certain block to hold the new record, if there is free space in the block, insert new record there. Otherwise, insert the new

record in an overflow file which is a temporary unordered file. Periodically, the overflow file is sorted and merged with the data file during file reorganization. Insertion becomes more efficient but the cost for search is increased.

- Delete a record : To delete a record, at first, we need to locate the record in the file (depends on the search condition, this step can be done by linear search or binary search) then delete it physically or logically by using deletion marker.
- Updating a record: Similar to deletion, we need to locate the record. If the fields to be modified are not ordering field then we can change the record and rewrite the block in which the record is placed on disk. If modifying the order field, it means the record can change position after updating. In this case, updating is equivalent to first deleting the old record followed by inserting the modified record.

3.3 3.3 Hashing Techniques

This organization is based on the technique of hashing which provide a direct access to records under certain search condition. The search condition must be equality condition on a single field in the record called hash field. The idea behind hashing is the hash function which map a value of a hash field of a record in to an address of disk block in which the record is stored. For most records, we need only a single block access to retrieve the record. Hashing technique can be used for internal files or external files on disk. In this section, we will describe some properties of hashing for files on disk.

The basic idea of external hash file organization is partition the target address space into buckets, each bucket is one or a cluster of contiguous disk blocks. Each block contains a number of records. There is a structure called bucket directory will be used to match the bucket number into the corresponding disk block address (as shown in figure below). In many case, the size of this structure is small enough to store in the memory.

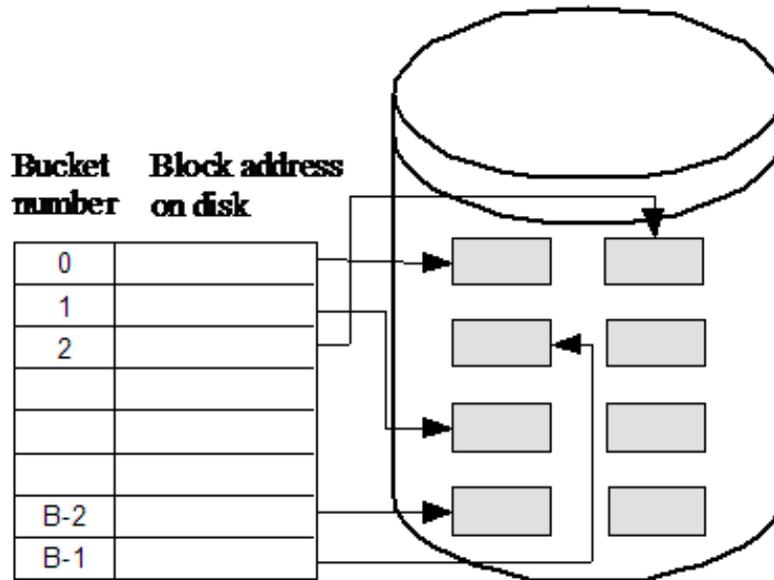


Figure 6: Bucket directory

The hashing function maps a search key K (value of hash field) to a bucket number. If there are B buckets which is range from 0 to $B-1$ then the hash function will returns a value in that range. One common hash function is the $h(K) = K \bmod B$. This function can be used for field of various type. Noninteger search-key values can be transformed into integers before applying the mod function.

One issue of hashing technique is choosing the right hash function. The goal of a good hash function is to distribute the records uniformly over the buckets such that the number of records in the buckets are similar. The worst possible hash function maps all search-key values to the same bucket.

3.3.1 Operations on Hashed Files

- Search for a record which have the hash field value is X : Firstly, applying the hash function $h(X)$. Assume we got the result i , from the bucket directory, we can get the address of the first block in the bucket number i . In a bucket, we can search for the record using linear search.
- Insert a record with hash field value X : To insert a new record with search-key value of X , we compute $h(X)$ which gives the address of the bucket for that record. Assume for now that there is a free space in the bucket to store the record and the new record is not already stored in the bucket, the record will be placed in the bucket.
- Delete a record with hash field value X : Deleting a record with search key X is straightforward, we firstly need to search for the corresponding bucket, then remove the record from its bucket.
- Update a record with hash field value X : We need to consider different situations in modifying the record. If the field to be updated is nonhash field, we can simply changing the record and rewrite it in the same bucket. Modifying the hash field means that the old record must be deleted then insert the modified record into the file.

When using external hashing, searching, deleting or updating a record given a value of some field other than the hash field is as expensive as in the case of unordered file.

Example: Assume we have a file S initially have 16 records, the file is partitioned into 5 different buckets which contains 3 disk blocks; each disk block in a bucket can hold at most 3 records, blocks in the buckets are linked together using pointer; the hash field of the record is an integer field. The hash function used in this example is $h(x) = x \bmod 5$. The hashed file organization is shown in figure 7. In figure 8 and 9, we will show the hashed file organization after performing several insertions and deletions.

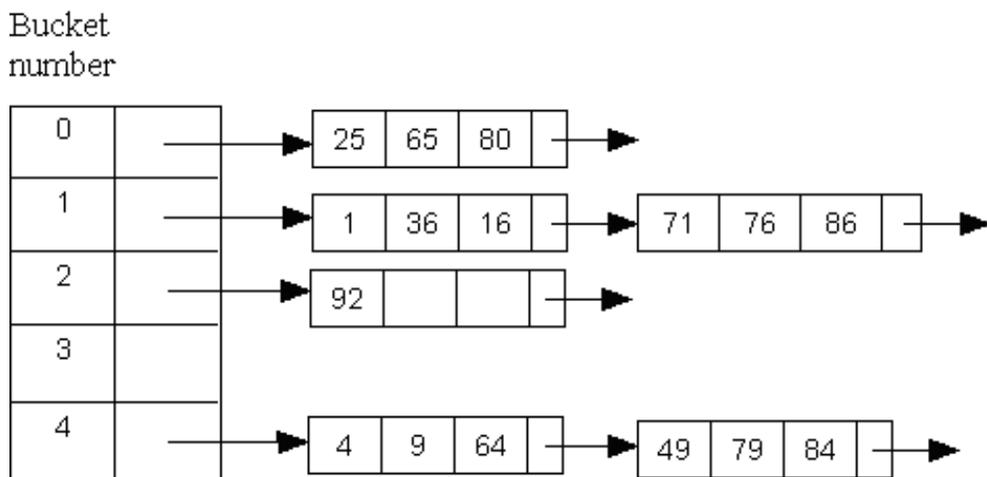


Figure 7: Example of hashed file organization

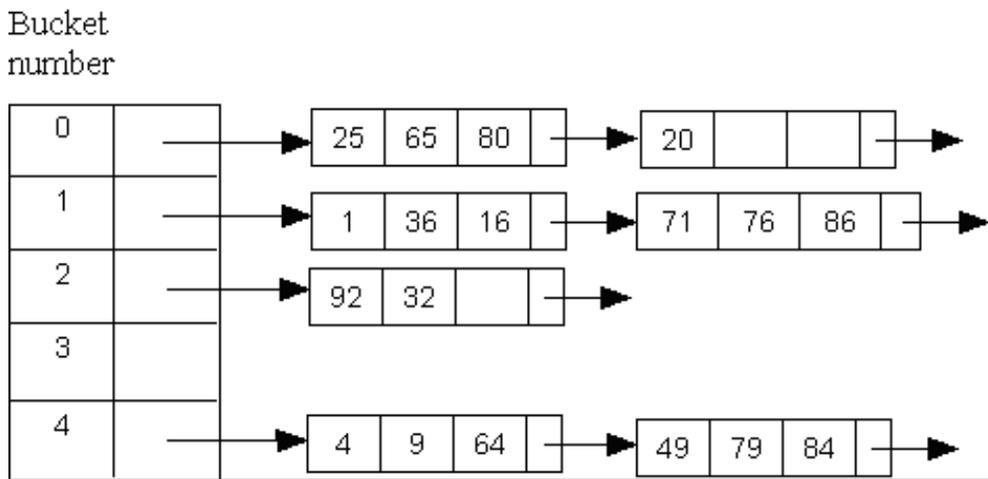


Figure 8: Hashed file after inserting records with hash value 32 and 20

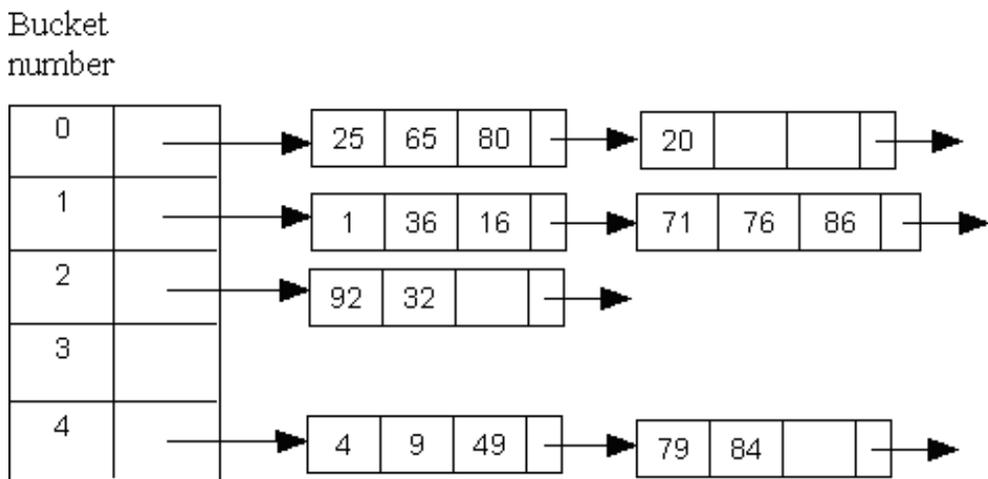


Figure 9: Hashed file after deleting records with hash value 64

So far, we have assumed that when a new record is inserted into the bucket, the bucket has space to store the record. If the bucket is currently full, we have the collision problem. We can handle this problem by using overflow chaining of buckets. If a record must be inserted in a bucket B which is already full, an overflow bucket is provided for B and the record is inserted to the overflow bucket. The overflow bucket is linked to the main bucket via pointer. The following figure shown the overflow chaining in hash structure.

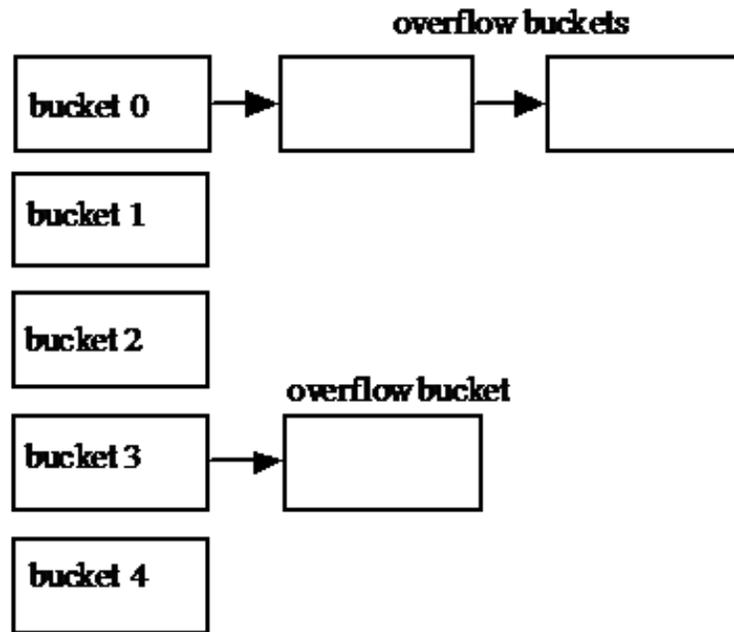


Figure 10: Overflow chaining in hash structure

4 4. Indexing Structures for Files

Index is an additional structure which are used to speed up the retrieval of records of a data file in response to certain search condition. The index provides secondary access path to records without affecting the physical placement of records on disk. In one data file, we can have several indexes which is defined on different fields. In this section, we will describe of single level index structure and dynamic multilevel index using B-trees.

4.1 4.1 Single-Level Ordered Index

An single-level orderd index based on an ordered data file. It works in much the same way as an index in the book. In the index of the book, we see a list of important terms is specified: terms in the list are placed in alphabetical order, along with each terms, there is a list of page numbers where the term appears in the book. When we want to search for a specific terms, use the index to locate the pages that contains the terms and then search those certain pages only.

An index access struture is usually defined on a single field of a data file, called an indexing field (or search field). Typically, each record consists of a value of index field and a pointer to a disk block that contains records with that value. Records in the index file may be stored in some sorted order based on the values of the index field. Thus we can do the binary search on the index. The index file is much smaller than the data file then using binary search on index structure is more efficient.

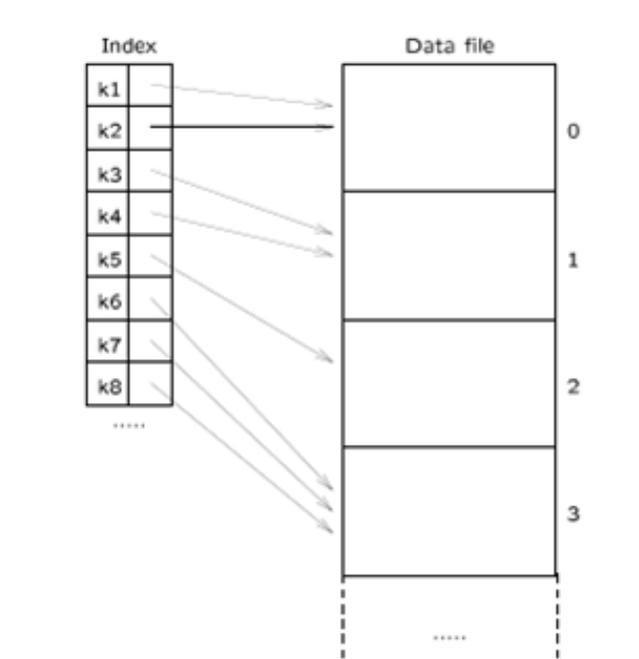


Figure 11: Index structure

There are several types of single-level ordered indexes:

- Primary index: this index is specified on the ordering key field of an ordered file. Ordering key field is the field that has the unique value for each record and the data file is ordered based on its value.
- Clustering index: this index is specified on the nonkey ordering field of ordered file
- Secondary index: this index is specified on a field which is not the ordering field of the data file. A file can have several secondary index.

Indexes can also be characterized as dense or sparse index:

- Dense index: there is an index entry for every search key value in the data file.
- Sparse index: An index entry is created for only some of the search values.

4.1.1 Primary Indexes

The index file includes a set of record. Each record (entry) in the primary index file has two field (k,p) : k is a key field which have the same data type as the ordering key field of data file, p is a pointer to a disk block . The entries in the index file are sorted based on the values of key fields.

Primary index can be dense or nondense.

Example: The EMPLOYEE data file is ordered by EID, dense index file using EID as the key value is shown in figure 11. Since the index file is sorted, we can do binary search on index file and followed the pointer in index entry (if found) to the record.

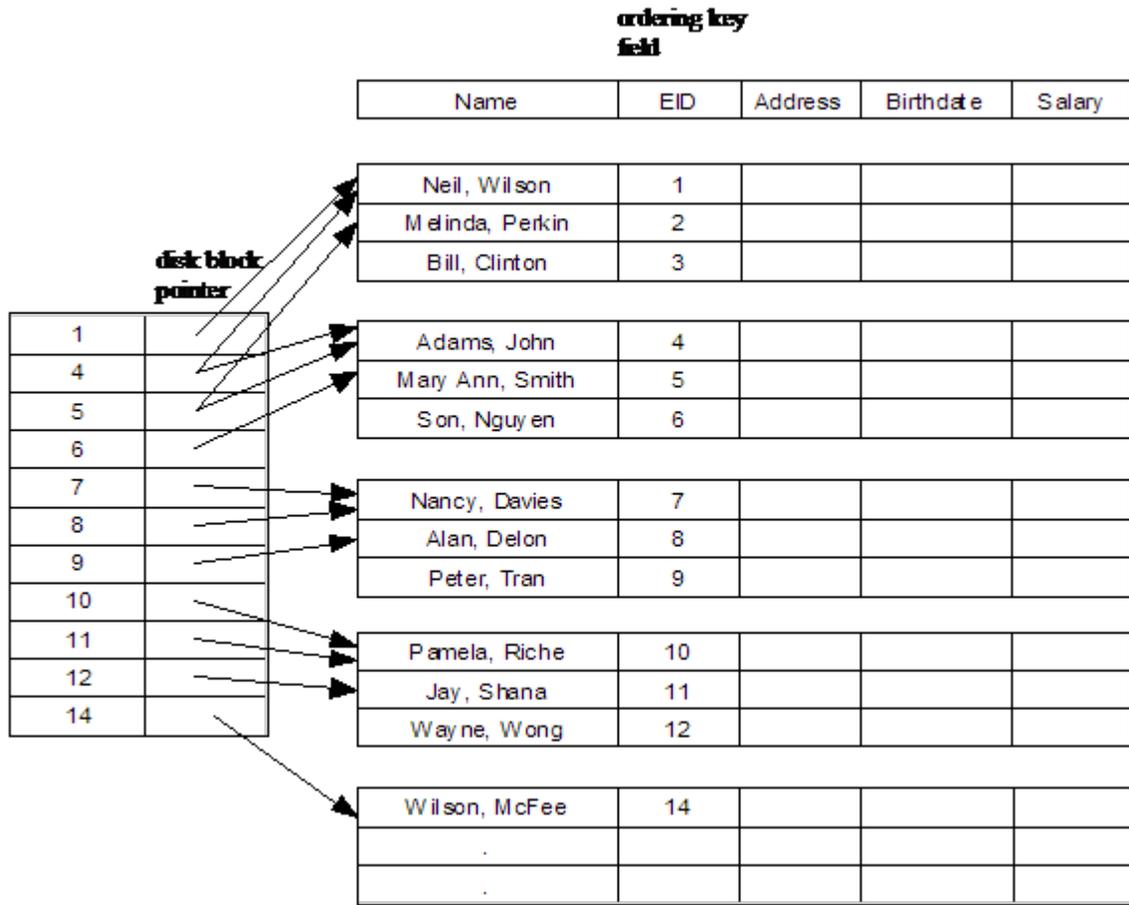


Figure 12: Example of Dense Primary Index in EMPLOYEE file

Example: Assume that the EMPLOYEE file is ordered by Name and each value of Name is unique so we have a primary index as shown in figure 13. This is a sparse index, each key value in an index entry is the value of Name of the first record in a disk block in the data file.

If we want to find the records of employee number 3, we cannot find the index entry with this value. Instead, we looking for the last entry before 3 which is 1 (for this, we can do binary search on index file) and follows that pointer to the block that might contains the expected record.

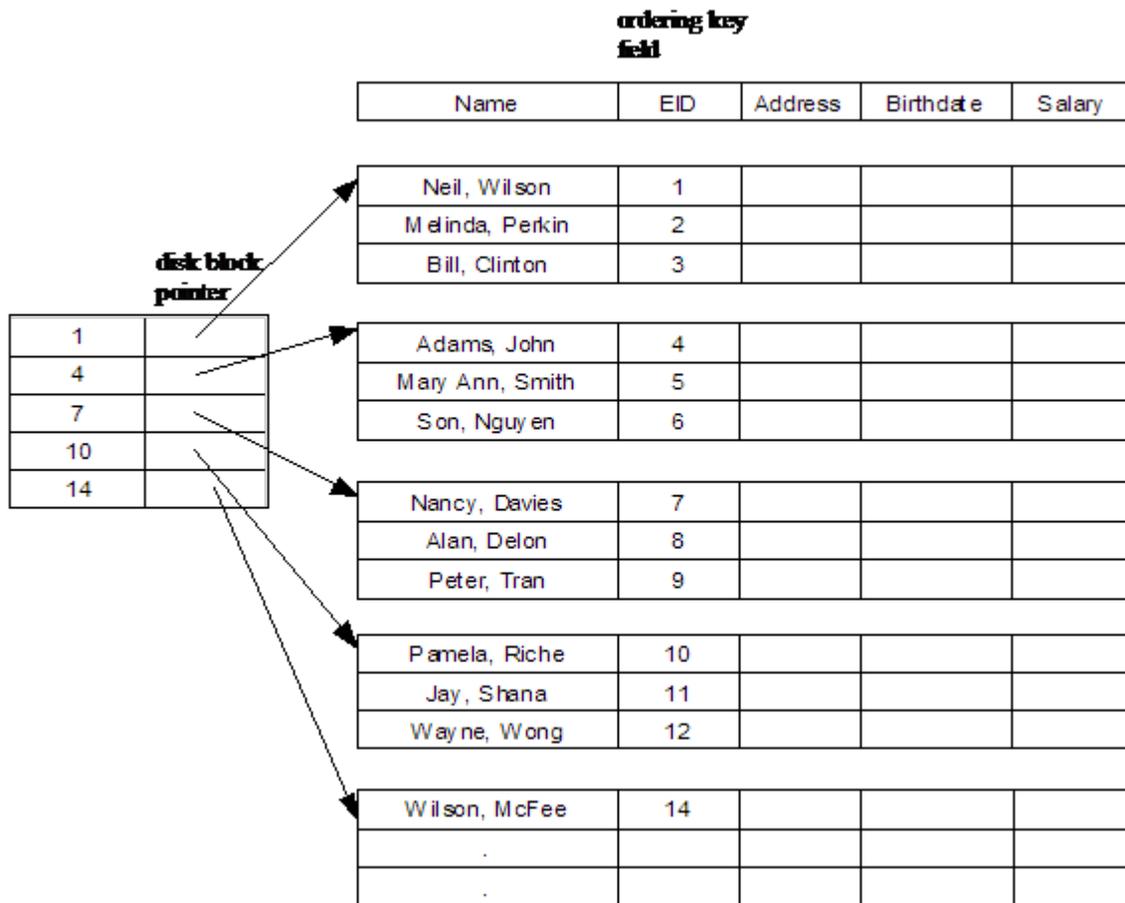


Figure 13: Example of Sparse Primary Index in EMPLOYEE file

4.1.2 Clustering Indexes

If the data file is ordered on a nonkey field (clustering field) which does not have unique value for each record, we can create clustering index.

An index entry in clustering index file has two fields, the first one is the same as the clustering field of the data file, the second one is the block pointer which points to the block that contains the first record with the value of the clustering field in the entry.

Example: Assume the EMPLOYEE file is ordered by DeptId as in figure 14, we are looking for the records of employees of D3. There is a index entry with value D3, follow the pointer in that index, we locate the first data record with value D3, continue processing records until we encounter a record for a department other than D3.

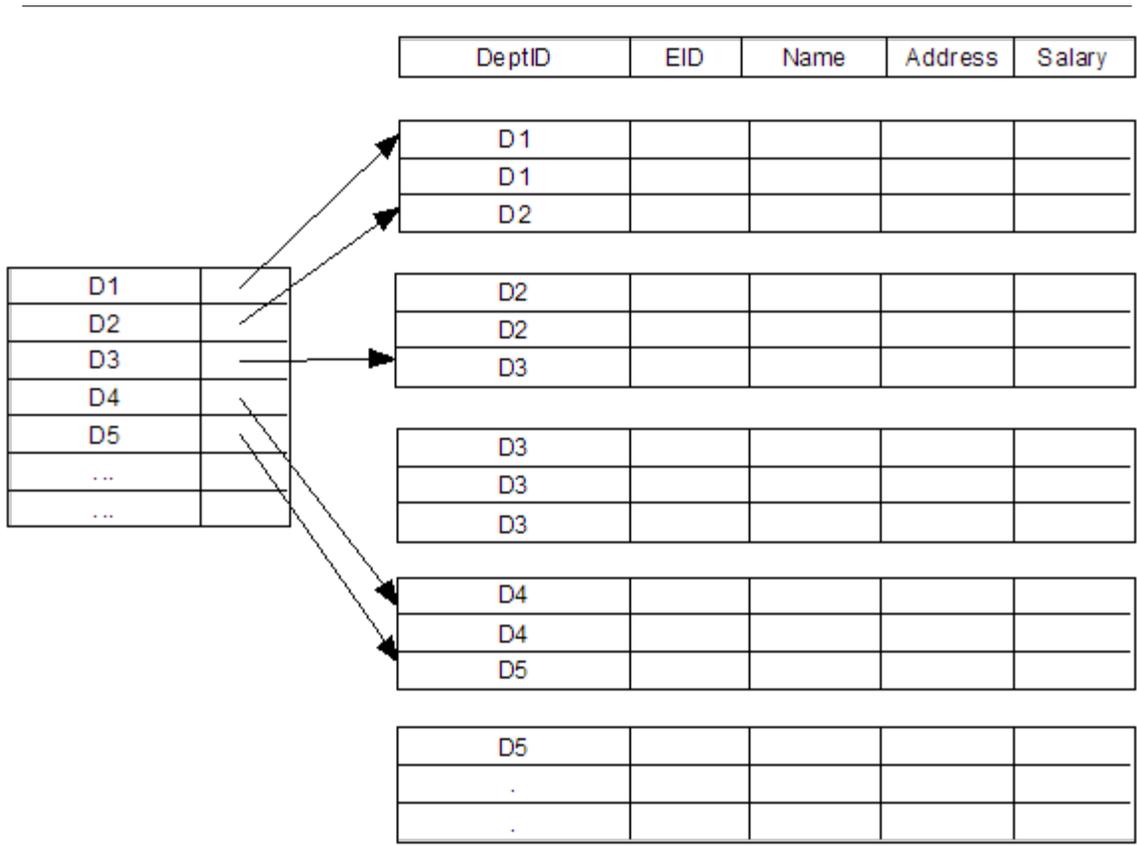


Figure 14: Clustering Index

4.1.3 Secondary Indexes

As mentioned above, secondary index is created on the field which is not an ordering field of the data file. This field might have unique value for every records or have duplicates values. Secondary index must be dense. Figure 15,16 illustrates a secondary index on a nonordering key field of a file and a secondary index on a nonordering , nonkey field respectively.

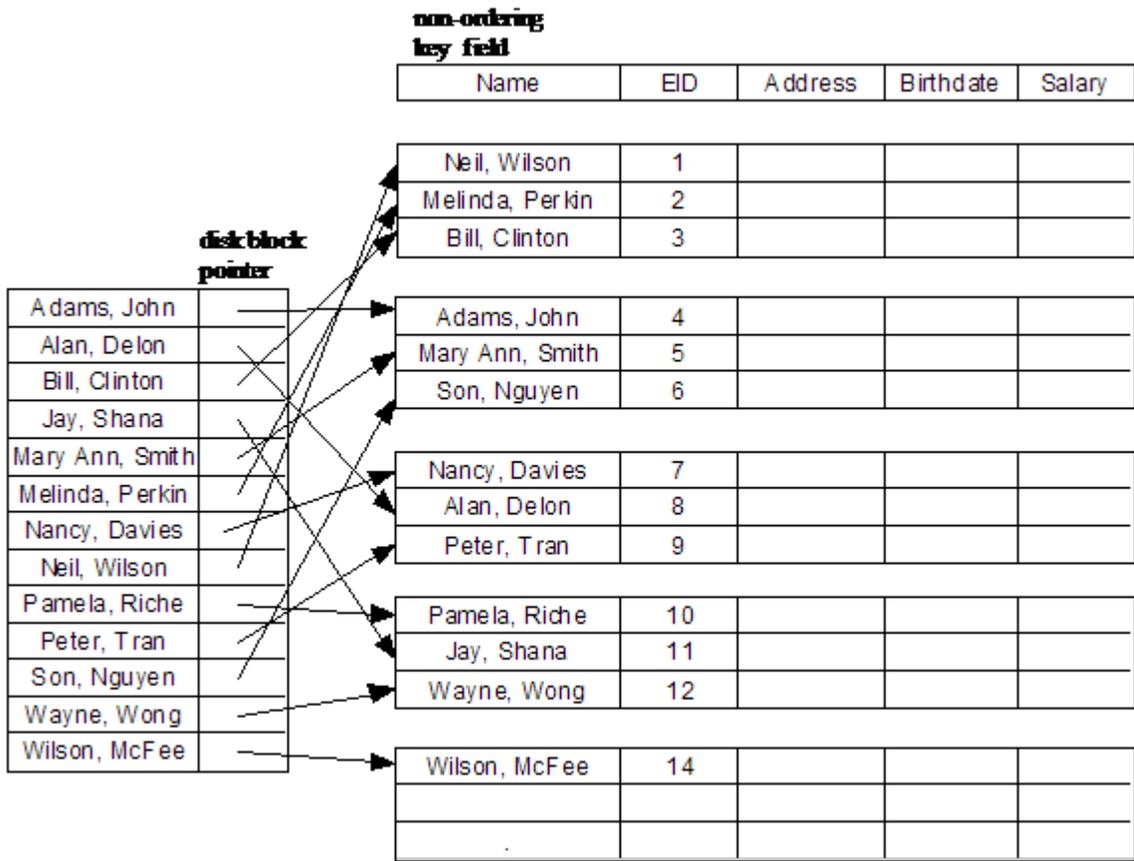


Figure 15: Secondary index on nonordering key field of a data file

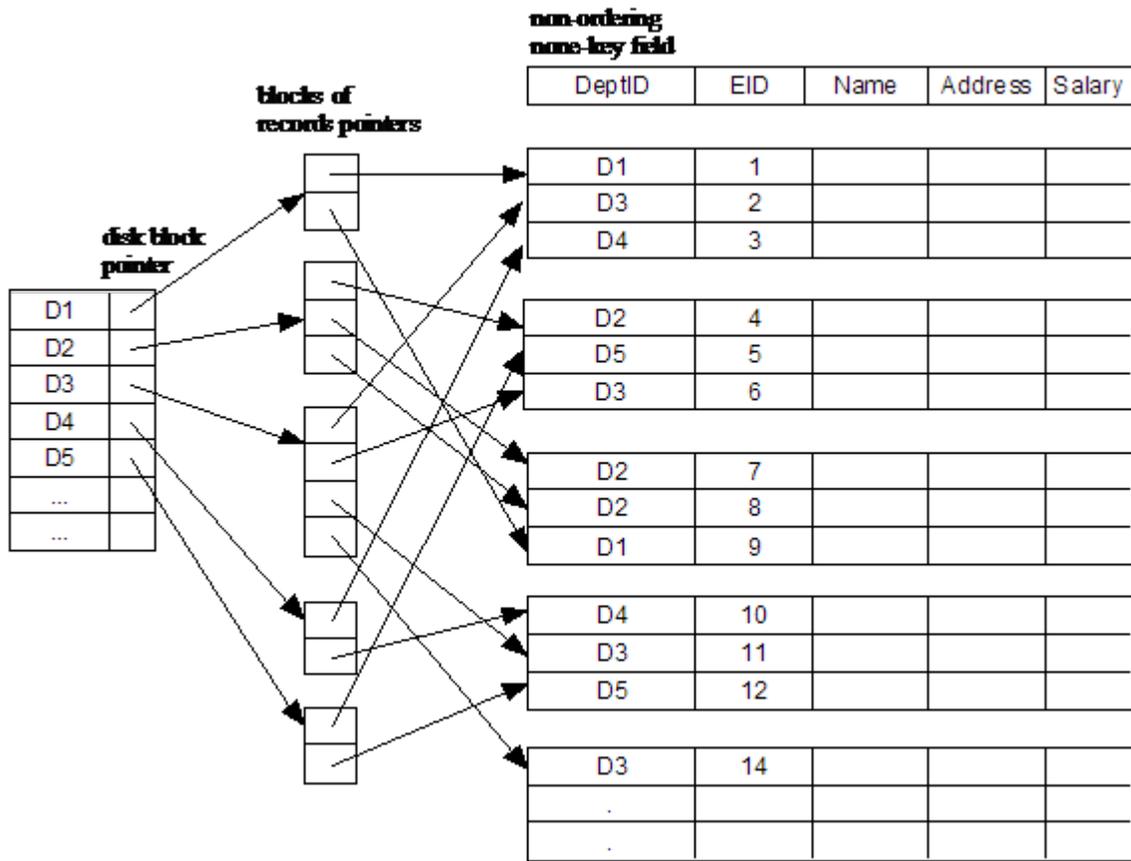


Figure 16: Secondary index on nonordering nonkey field of a data file using one level of indirection

Secondary index usually need more storage space and longer search time than a primary index because it has larger number of entries. However, it improves the performance of queries that use keys other than the search key of the primary index. If there is no secondary indices on such key, we would have to do linear search.

4.1.4 Operations in Sparse Primary Index

The insert, delete and modify operations might be different for various types of index. In this section, we will discuss those operations for the sparse primary index structure.

- Looking up for a record with search key value K: Firstly, find the index entry of which the key value is smaller or equal K. This searching in the index file can be done using linear or binary search. Once we have located the proper entry, following the pointer to the block that might contain the required record.
- Insertion: To insert a new record with search key value K, we firstly need to locate the data block by look up in the index file. Then we store the new record in the block. No change needs to be made to the index unless a new block is created or a new record is set to be the first record in the block. In

those case, we need to add a new entry in the index file or modify the key value in a existing index entry.

- Deletion: Similar to insertion, we find the data block that contains the deleted record and delete it from the block. The index file might be changed if the deleted record is the only record in the block (an index entry will be deleted either) or the deleted record is the first record in the block (we need to update the key value in the correspond index entry)
- Modification: First, locating the records to be updated. If the field to be changed is not index field, then change the record. Otherwise, delete the old record and then insert the modified one.

4.2 4.2 Dynamic Multilevel Indexes Using B + Trees

The main disadvantages of index –sequential file organization is that performance degrades as the file grows. B+ tree is a widely used index structure in the database system because of their efficiency despite insertion and deletion of data.

4.3 Structure of B+ tree

A B+ tree of order m has the following properties:

- The root node of the tree is either a leaf node or has at least two used pointers. Pointers point to B+tree node at the next level.
- The leaf node in B+ tree have an entry for every values of the search field along with a data pointer to the record (or to a block that contains this record). The last pointer points to the next leaf to the right. A leaf node contains at least $\lceil m/2 \rceil$ and at most m-1 values. Leaf node is of the form $\langle k_1, p_1 \rangle, \langle k_2, p_2 \rangle, \dots, \langle k_{m-1}, p_{m-1} \rangle, \langle k_m, p_m \rangle$ where p_m is the pointer to the next node.
- Each internal node in B+ tree is of the form $(p_1, k_1, p_2, k_2, \dots, p_{m-1}, k_{m-1}, p_m)$. It contains up to m-1 search key values $k_1, k_2, k_3, \dots, k_{m-1}$ and m pointers $p_1, p_2, p_3, \dots, p_m$. The search key values within a node is sorted $k_1 < k_2 < k_3 < \dots < k_{m-1}$. Actually, the internal nodes in B+ tree forms multilevel sparse index on the leaf nodes. At least $\lceil m/2 \rceil$ pointers in the internal node are used.
- All paths from root node to leaf nodes have equal length.

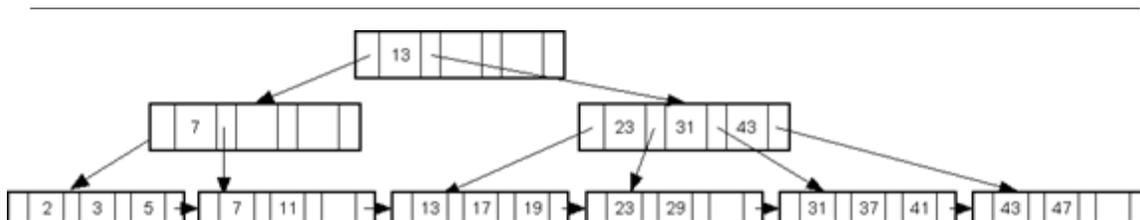


Figure 17: Example of B+ tree with order 4

4.3.1 Searching for a record with search key value k with B+ tree

Searching for the record with search key value k in a tree B means we need to find the path from root node to a leaf node that might contains values k.

- If the root of B is leaf node, look among the search key values there. If the values is found in position i then pointer i is the pointer to the desire record.
- If we are at a internal node with key values $k_1, k_2, k_3, \dots, k_{m-1}$, we examine the node, looking for the smallest search key value greater than k . Assume that this search key value is k_i , we follow the pointer p_i the the node in the next level. If $k < k_1$ then follows p_1 , if we have m pointers in the node and $k \geq k_{m-1}$ then we follows p_m the the node in next level. Recursively apply the search procedure at the node in next level

4.3.2 Inserting a record with search key value k in a B+ tree of order m

- Using searching to find the leaf node L to store new pair $\langle \text{key}, \text{pointer} \rangle$ to the new record.
- If there is enough space for the new pair in L , put the pair there.
- If there is no room in L , we split L into two leaf nodes and divide the keys between two leaf node so each is at least half full.
- Splitting at one level might lead to splitting in the higher level if a new key –pointer pair needs to be inserted into a full internal node in the higher level.

The following procedure describe the important steps in inserting a record in B+ tree

```

procedure insert-entry (node L, value V, pointer P) {
  if (L has space for (V,P) then insert (V,P) in L
  else begin /* split L */
    create node L' is the right sibling of L
    if (L is a leaf node) then begin
      ▪ move all entries of L to temp
      ▪ insert entry (V,P) in temp in correct position
      ▪  $j = \lceil m/2 \rceil$ 
      ▪ move first j entries in temp back to L
      ▪ move the remaining entries in temp to L'
      ▪ V' is the first key value in L'

    end
    else begin
      /* L is an internal node with m-1 keys and m pointers, we are trying to
      insert a new keys and new pointers to L */
      • leave at L the first  $\lceil (m+1)/2 \rceil$  pointers and move to L' the
      remaining pointers
      • Leave at L the first  $\lceil (m-1)/2 \rceil$  keys, move to L'  $\lfloor (m-1)/2 \rfloor$  keys
      to L'. V' is The key in the middle which is left over.

    end.
    if (L is not root of the tree) then begin
      P' is the pointer to L'
      M is the parent of L
      insert-entry(M, V', P')
    end
    else begin
      create node R with child nodes L and L' with single value V'
      make R is the new root of tree.
    end
  end
}

```

Figure 18: Algorithm of inserting a new entry into a node of B+ tree

Example of inserting new record with key value 40 into tree in figure 10.16: Key values 40 will be inserted into leaf node which is already full (with 3 keys 31, 37, 41). The node is split into two, first new node contains keys 31 and 37, second node contains keys 40 and 41. Then the pair <40,pointer> will be copied up to the node in higher level.

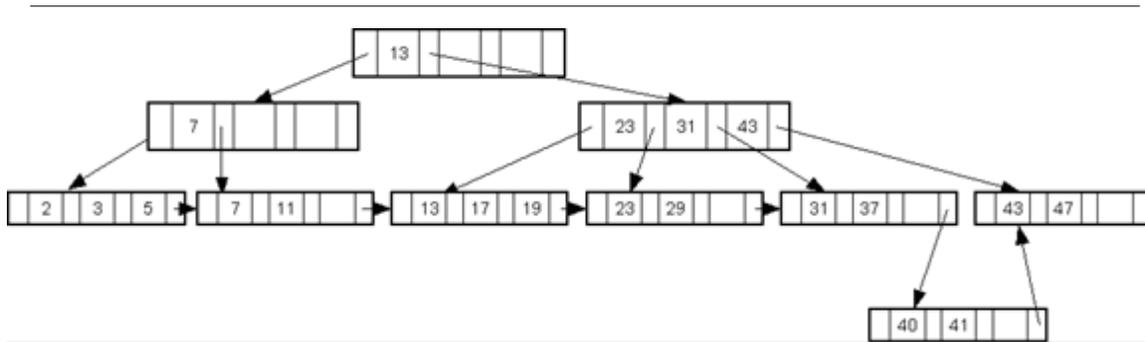


Figure 19: Beginning the insertion of key 40, split the leaf node

The internal node in which pair $\langle 40, \text{pointer} \rangle$ is inserted is also full (with keys 23, 31, 43 and 4 pointers), we have internal node splitting situation. Consider 4 keys 23, 31, 40, 43 and 5 pointers. According to the above algorithm, first 3 pointers and first 2 keys (23, 31) stay in the node, the last 2 pointers and last key (43) moved to the new right sibling of internal node. Key 40 is left over and push up to the node in the higher level.

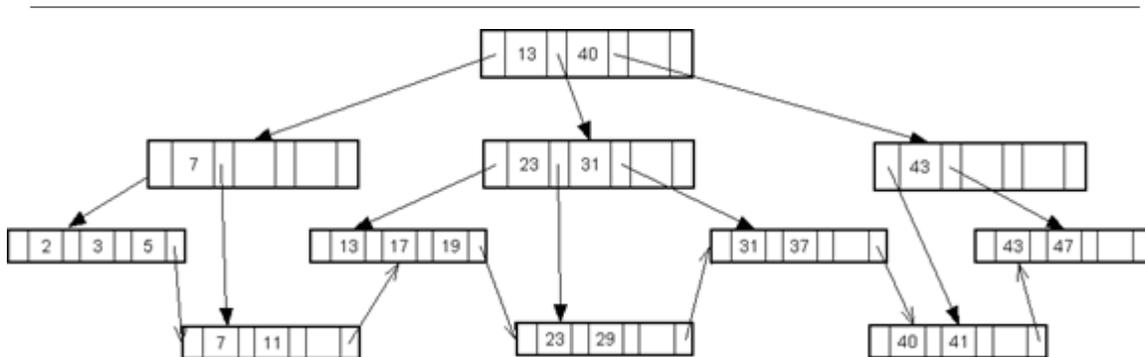


Figure 20: After inserting key 40

4.3.3 Deleting a record with search key value k in a B+ tree of order m

- Deleting begins with looking up the leaf node L that contains the records, delete the data record, then delete the key-pointer pair to that record in L
- If after deleting L still has at least the minimum number of keys and pointers, nothing more can be done.
- Otherwise, we need to do one of two things for L
 - If one of the adjacent siblings of L has more than minimum number of keys and pointers, then borrow one key-pointer pairs of the sibling, keeping the order of keys intact. Possibly, the keys at

the parent of L must be adjusted.

- If cannot borrowing from siblings but entries from L and one of the siblings, says L' can fit in a single node, we merge these two nodes together. We need to adjust the keys at the parent and then delete a key-pointer pair at the parent. If the parent still has enough number of keys and pointers, we are done. If not, then we recursively apply the deletion at the parent.

```

procedure delete-entry(node L, value V, pointer P) {
  delete (V,P) from L
  if (L is the root and L has only one child remaining) then make the child of L be
  the new root and delete L
  else if (L has too few keys/pointers) then begin
    let L' be one of the sibling of L ; V' is the keys between pointers L, L'
    if (L and L' can merge) then begin
      /* assume L' is the right sibling of L */
      if (L is not a leaf) then append V' and all values, pointers of L to
      L'
      else append all values, pointers of L to L'
      M is parent of L, P' is pointer to L
      delete-entry (M, V', P')
    end
    else begin /* Borrow from siblings*/
      if (borrow from left sibling L') then begin
        if (L is non leaf node) then begin
          let pn be the last pointer of L' then remove
          <kn-1, pn> from L'
          insert <pn, V'> as first pointer and keys in L
          replace V' in parent of L by kn-1
        end
        else begin
          let <kn, pn> be the last key-pointer pair of L'
          delete <kn, pn> from L'
          insert <kn, pn> into L (as the first entry)
          replace V' in parent L by kn
        end
      end
      else begin /* borrow from right sibling , do it similarly */
        .....
      end
    end
  end
end
}

```

Figure 21: Algorithm of deleting an entry in a node of B+ tree

Example:

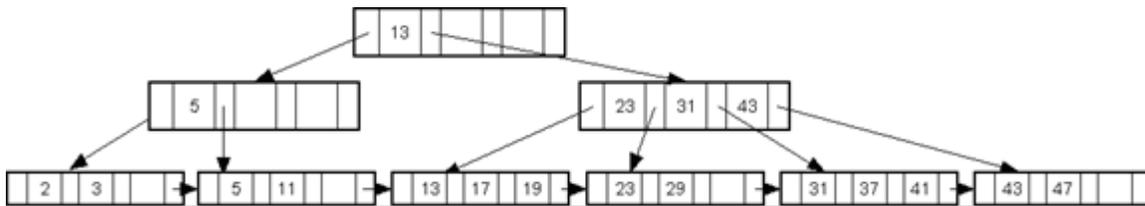


Figure 22: Delete record with key 7 from the tree in figure 17 Borrow from the sibling

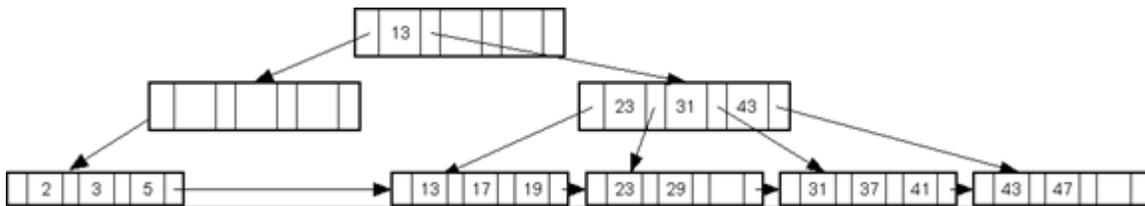


Figure 23: Beginning of deletion of record with key 11 from the tree in figure 22. This is the case of merge two leaf nodes

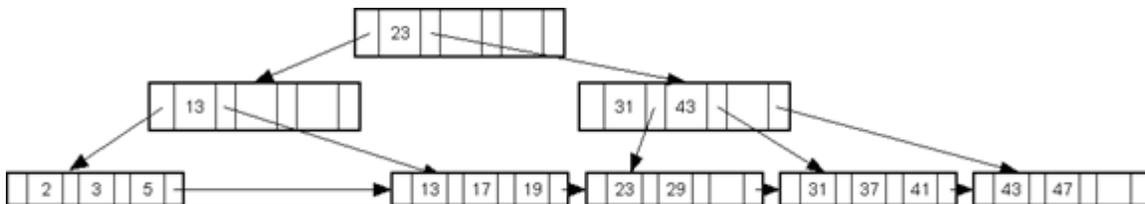


Figure 24: After deletion of key 11