MODULE 3
SQL DML, Physical Data Organization

SYLLABUS

- **SQL DML (Data Manipulation Language)**
  - SQL queries on single and multiple tables, Nested queries (correlated and non-correlated), Aggregation and grouping, Views, assertions, Triggers, SQL data types.
- **Physical Data Organization**
  - Review of terms: physical and logical records, blocking factor, pinned and unpinned organization. Heap files, Indexing, Singe level indices, numerical examples, Multi-level-indices, numerical examples, B-Trees & B+-Trees (structure only, algorithms not required), Extendible Hashing, Indexing on multiple keys – grid files

Fixed and Variable length records

- A file is a sequence of records.
- In many cases, all records in a file are of the same record type.
- If every record in the file has exactly the same size (in bytes), the file is said to be made up of fixed-length records.
- If different records in the file have different sizes, the file is said to be made up of variable-length records.

Physical and Logical records

Physical files contain the actual data that is stored on the system, and a description of how data is to be presented to or received from a program.A **physical record** often is unstructured and has a fixed size related to the kind of physical media that stores it, and possibly to the location of the record on the media.

Logical files do not contain data. They contain a description of records found in one or more physical files. A logical file is a view or representation of one or more physical files. A **logical record** often is structured (has various program-specific fields) and might be stored in some number of full or partial physical records.

Pinned and Un pinned Records

A record is said to be **pinned record,** if there exists a pointer to it somewhere in the database. For example, when a table look up approach is used to locate a record, the table contains a pointer to the record and the record becomes pinned down.

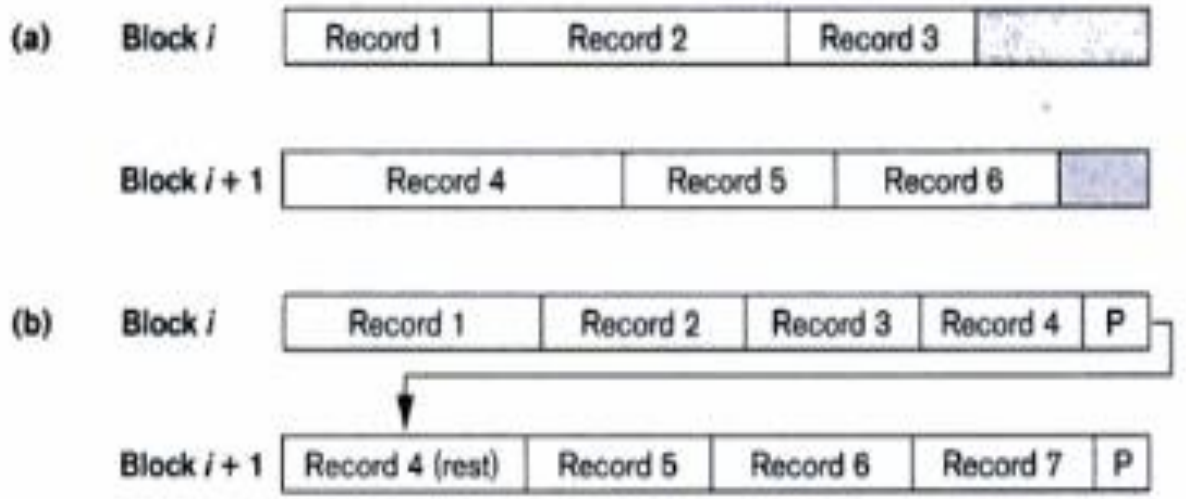
A record is said to be **unpinned record**, if there does not exist any pointer pointing to it in the database. In fact, it is the independent record.

Spanned and Unspanned Organization

- Records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory.

- Part of the record can be stored on one block and the rest on another.
- A pointer at the end of the first block points to the block containing the remainder of the record.
- This organization is called **spanned** because records can span more than one block.
- Whenever a record is larger than a block, we must use a spanned organization.
- If records are not allowed to cross block boundaries, the organization is called **unspanned.**

Figure 13.6
Types of record organization. (a) Unspanned. (b) Spanned.



6. Other schemes are also possible for representing variable-length records.

Heap Files

It is the simplest and most basic type of organization. It works with data blocks. Heap file (or unordered file) places the records on disk in no particular order by appending new records at the end of the file.When the records are inserted, it doesn't require the sorting and ordering of records. It is the DBMS responsibility to store and manage the new records.

Advantages of Heap file organization

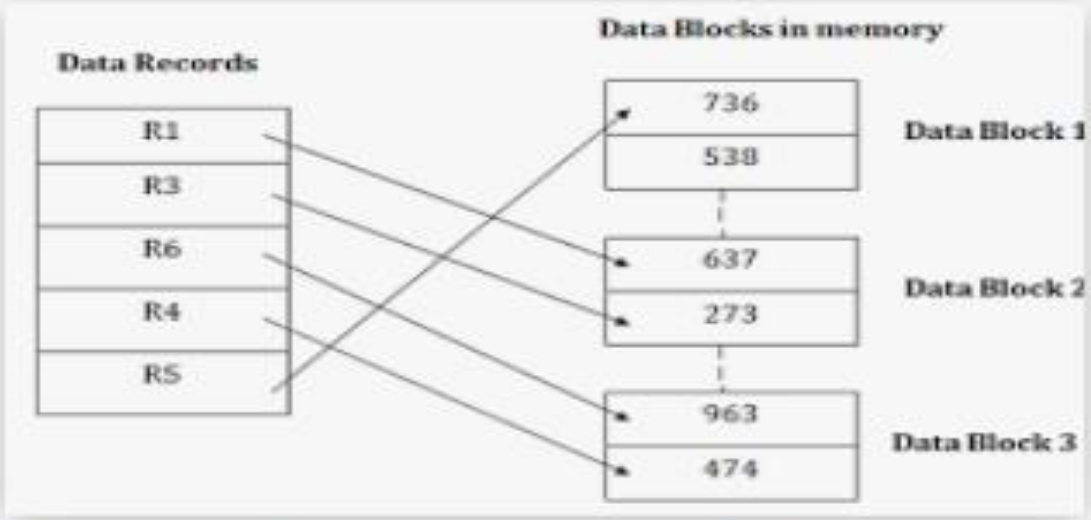
It is a very good method of file organization for bulk insertion. If there is a large number of data which needs to load into the database at a time, then this method is best suited.

In case of a small database, fetching and retrieving of records is faster than the sequential record.

Disadvanatages of Heap file organization

This method is inefficient for the large database because it takes time to search or modify the record.

This method is inefficient for large databases.



DBMS Heap File Organization

Blocking factor for the file

- The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory.
- When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block.
- Suppose that the block size is B bytes.
- For a file of fixed-length records of size R bytes, with $B \geq R$, we can fit $bfr = \lfloor B / R \rfloor$ records per block.
- The **value bfr** is called the **blocking factor** for the file.

Index Structures

- Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.
- An index on a database table provides a convenient mechanism for locating a row (data record) without scanning the entire table and thus greatly reduces the time it takes to process a query.
- The index is usually specified on one field of the file.
- One form of an index is a file of entries <**field value**, **pointer to record**>, which is ordered by field value.
- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller.

Types of index

- Indexes can be characterized as
  1. Dense index
  2. Sparse index
- A dense index has an index entry for every search key value (and hence every record) in the data file.
- A sparse (or nondense) index, on the other hand, has index entries for only some of the search values.

- Advantages:
  - Stores and organizes data into computer files.
  - Makes it easier to find and access data at any given time.
  - It is a data structure that is added to a file to provide faster access to the data.
  - It reduces the number of blocks that the DBMS has to check.
- Disadvantages
  - Index needs to be updated periodically for insertion or deletion of records in the main table.

Structure of index

- An index is a small table having only two columns.
- The first column contains a copy of the primary or candidate key of a table
- The second column contains a set of pointers holding the address of the disk block where that particular key value can be found.
- The two field values of index entry i are referred as <K(i), P(i)> where

  K(i) = the value of the key field

  P(i) = a pointer to that block
- If the indexes are sorted, then it is called as ordered indices.

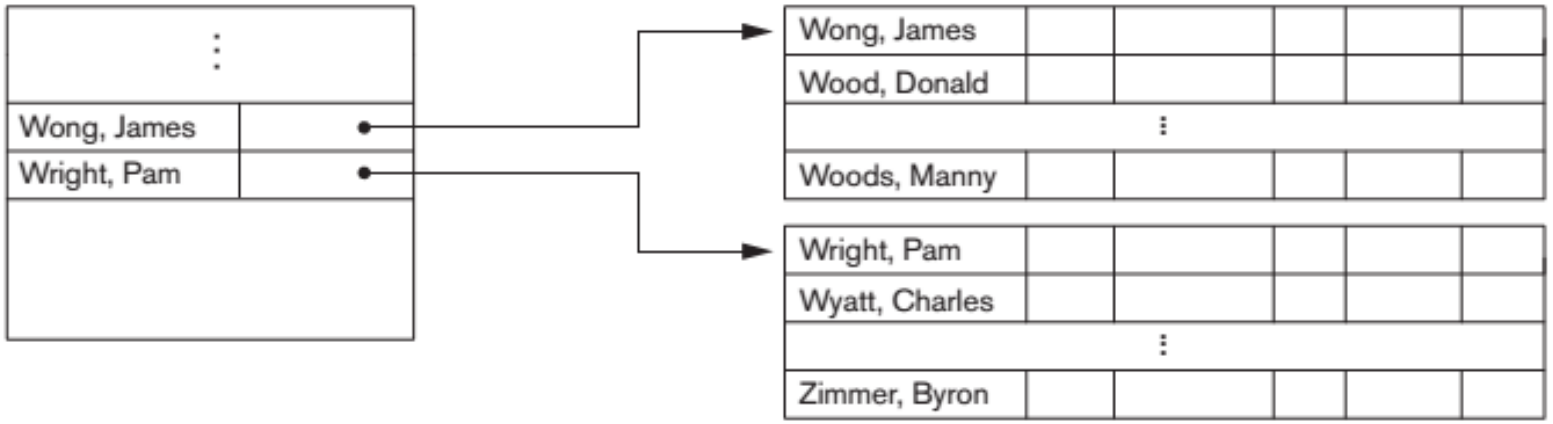
Example

- Suppose we have an ordered file with 30,000 records and stored on a disk of block size 1024 bytes and records are of fixed size, unspanned organisation. Record length = 100 bytes. How many block access needed to search a record?

No. of records,	r	= 30,000	
Block size,	B	= 1024 bytes	
Record size,	R	= 100 bytes	
Blocking factor,	bfr	= $\lfloor B / R \rfloor$	= $\lfloor 1024 / 100 \rfloor$ = 10 records/block
No. blocks,	b	= $\lceil r/bfr \rceil$	= $\lceil 30,000/ 10 \rceil$ = 3000 blocks
If linear search		= 3000 / 2	= 1500 block access
If binary search		= $\log_2(3000)$	= 12 block access

Primary Index

- Primary index is defined on an ordered data file. The data file is ordered on a key field.
- The key field is generally the primary key of the relation.
- A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.
- The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor** (shown below).



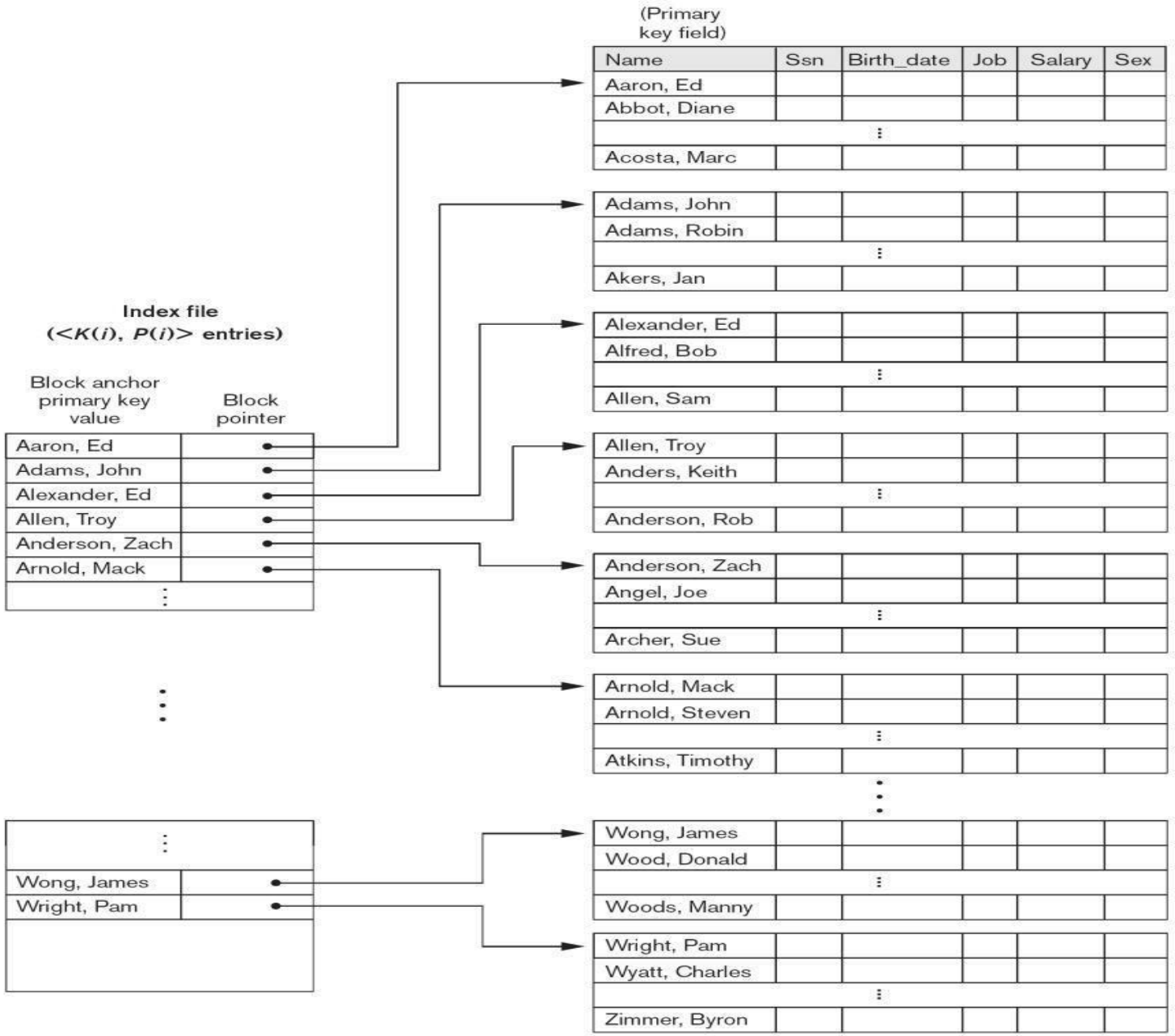


Figure 5.1: Primary Index on the Ordering Key Field of the File

Example 1

- Suppose we have an ordered file with 30,000 records and stored on a disk of block size 1024 bytes and records are of fixed size, unspanned organisation.
- Record length = 100 bytes. How many block access if using a primary index file, with an ordering key field of the file 9 bytes and block pointer size 6 bytes.

Blocking factor,	bfr	= ⌊B / R⌋	= ⌊1024 / (9 + 6)⌋	= 68 entries/block
Total no. of index entries		= No. of blocks in data file		= 3000
No. of blocks needed		= ⌈r/bfr⌉	= ⌈3000/68⌉	= 45 blocks
If binary search		= $\log_2(45)$	= 6 block access	
Total no. of search		= 6 block access + 1 block access (data file)		
		= 7 block access		

Clustering Index

- Defined on an ordered data file.
- The data file is ordered on a non-key field unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
- Includes one index entry for each distinct value of the field;
  - the index entry points to the first data block that contains records with that field value.
- It is another example of nondense index where Insertion and Deletion is relatively straightforward with a clustering index.

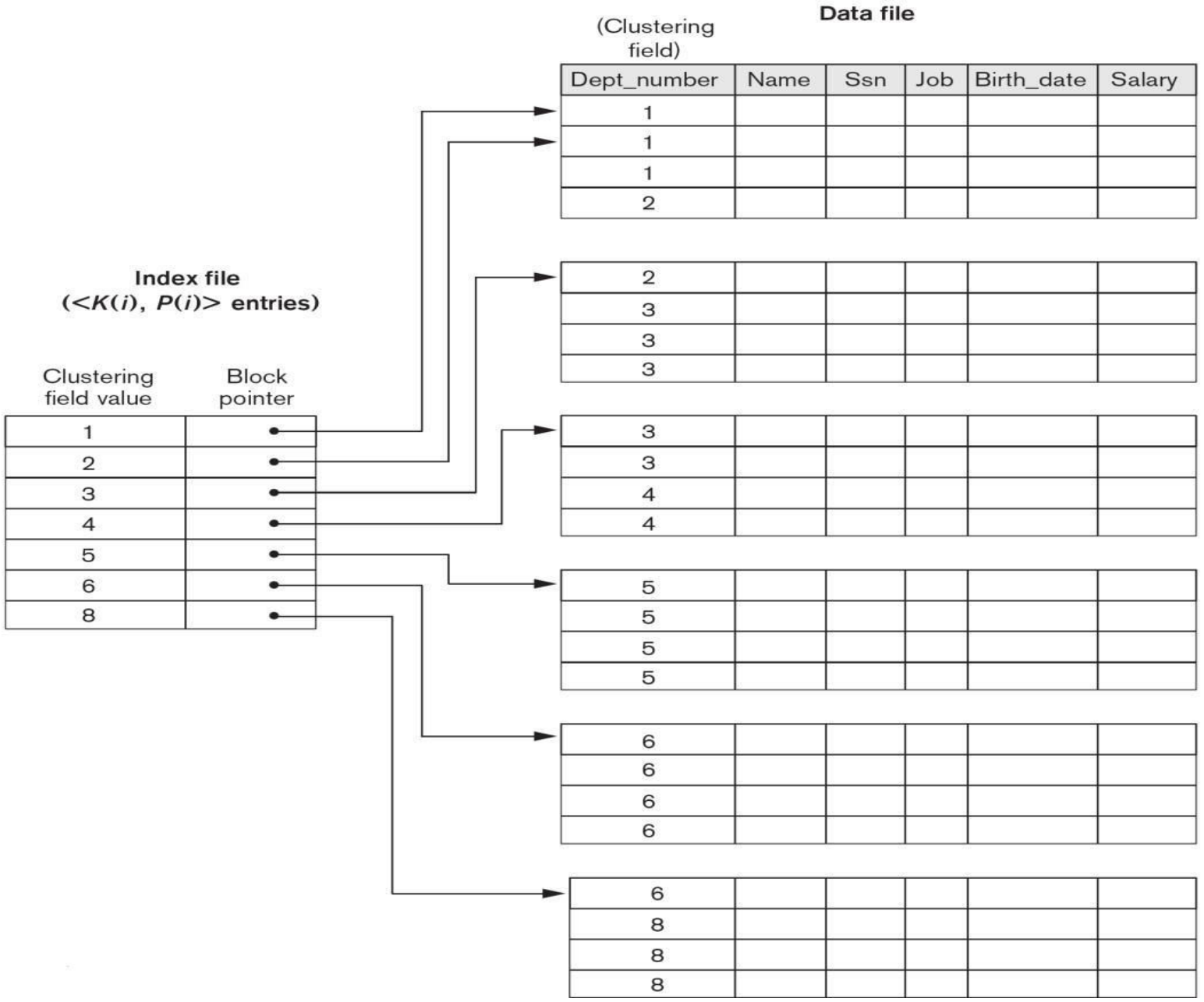


Figure 5.2: A Clustering Index on Dept_number Ordering Nonkey Field of a File

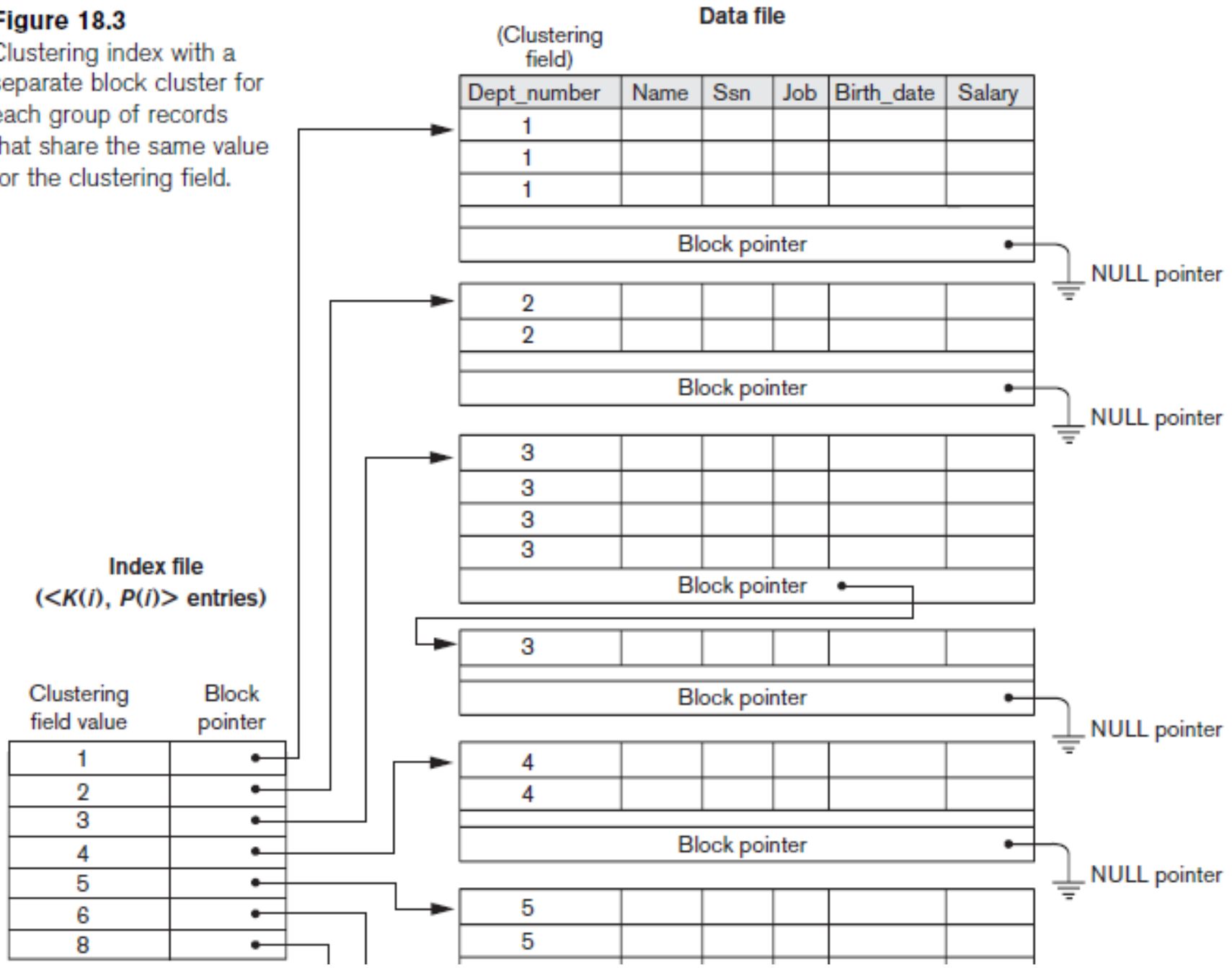
Clustering Index(contd..)

It is common to reserve a whole block (or a cluster of contiguous blocks) for each value of the clustering field; all records with that value are placed in the block (or block cluster).

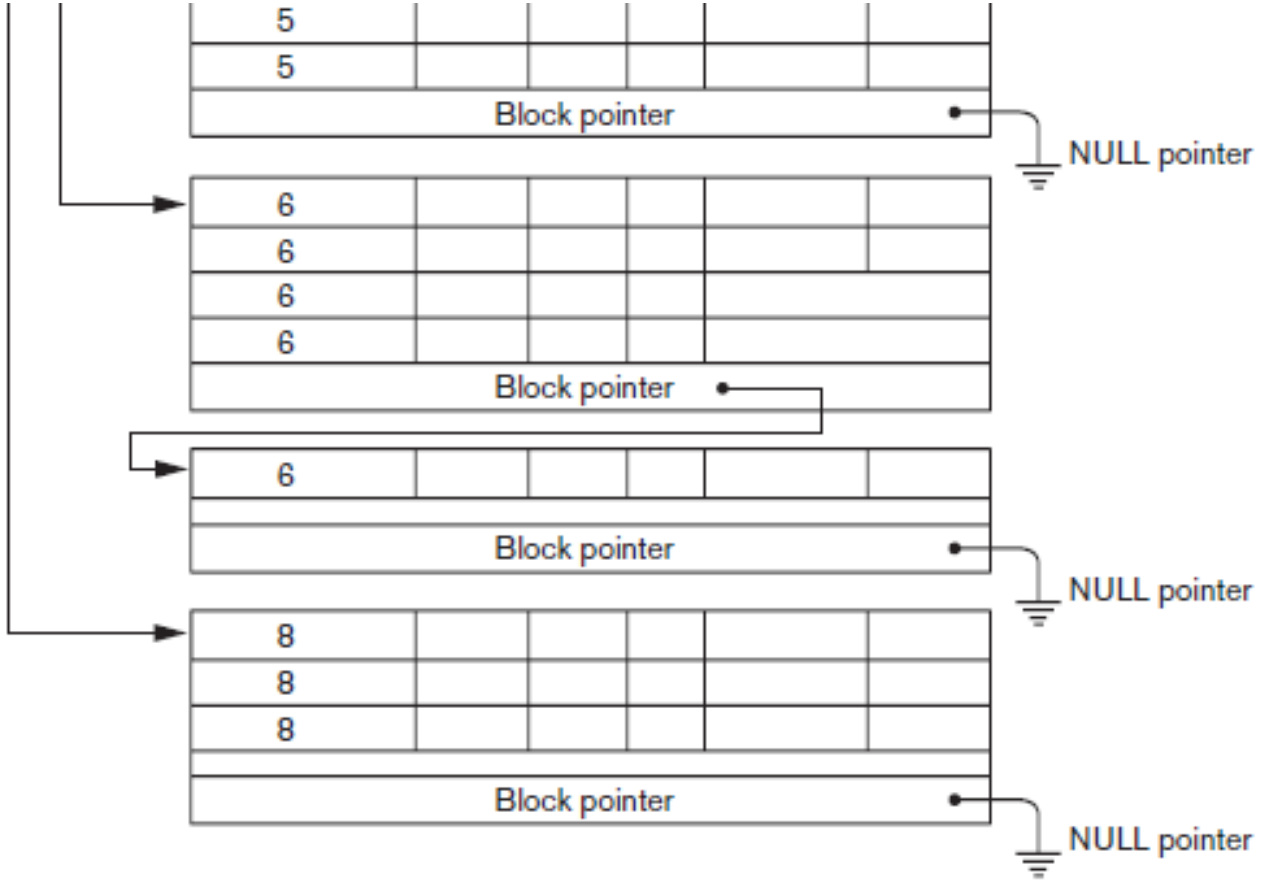
This makes insertion and deletion relatively straightforward.

Figure 18.3 shows this scheme.

Figure 18.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



Contd..


5
5
Block pointer
NULL pointer
6
6
6
6
Block pointer
6
Block pointer
NULL pointer
8
8
8
Block pointer
NULL pointer

Secondary Index

- A secondary index provides a secondary means of accessing a file for which some primary access already exists.
- The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- The index is an ordered file with two fields.
- The first field is of the same data type as some non-ordering field of the data file that is an indexing field.
- The second field is either a block pointer or a record pointer.
- There can be many secondary indexes (and hence, indexing fields) for the same file.
- Includes one entry for each record in the data file; hence, it is a dense index.

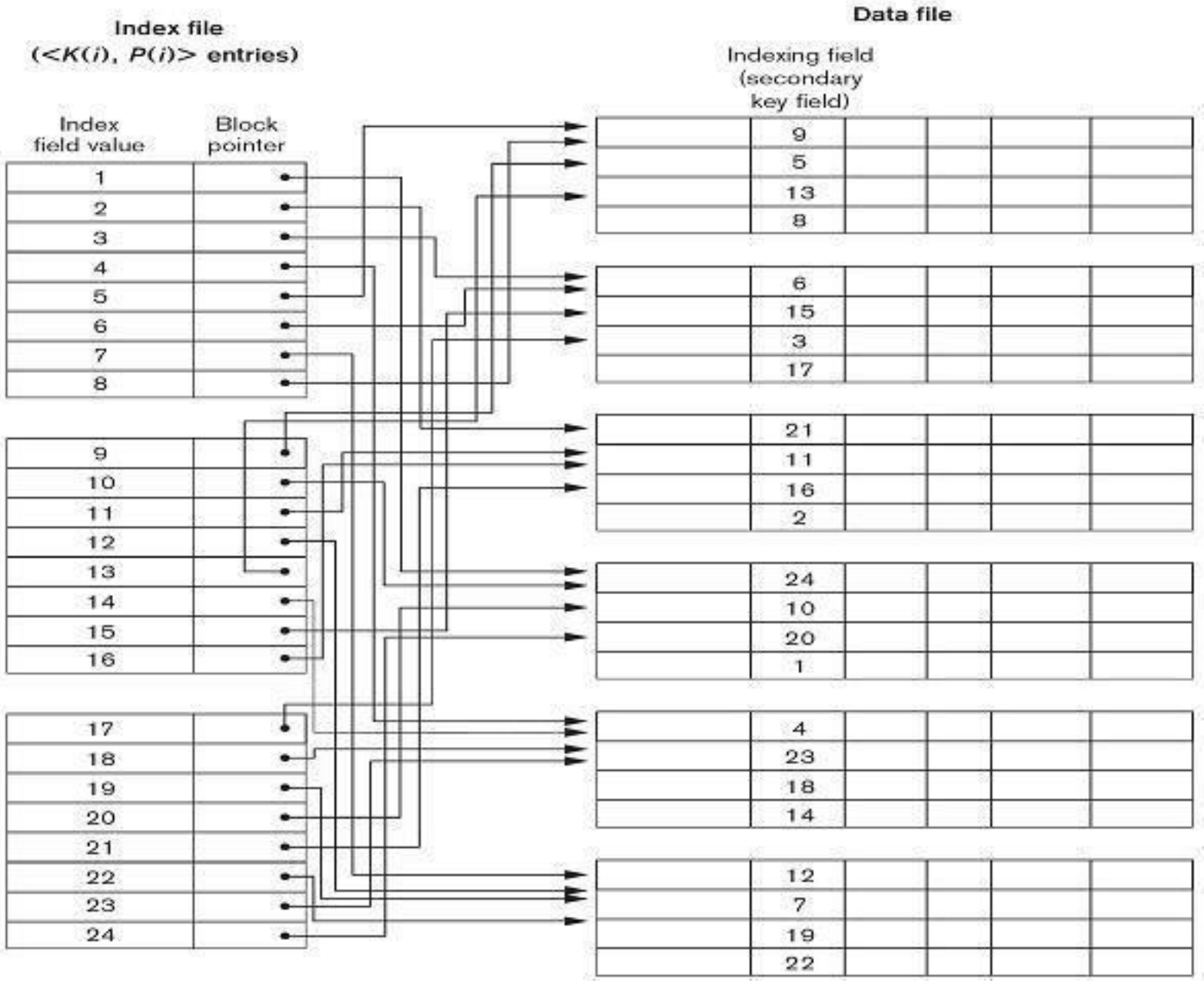


Figure 5.3: Dense Secondary Index (with Block Pointer) on a Non Ordering Key Field of the File

Example 2

- Suppose we have an ordered file with 30,000 records and stored on a disk of block size 1024 bytes and records are of fixed size, unspanned organisation. Record length = 100 bytes. How many block access if using a secondary index file.

Total no. of index entries	= No. of records in data file		= 30000
No. of blocks needed	= ⌈r/bfr⌉	= ⌈30000/68⌉	= 442 blocks
If binary search	= $\log_2(442)$	= 9 block access	
Total no. of search	= 9 block access + 1 block access (data file)		
	= 10 block access		

If some value K(i) occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used.

This technique is illustrated in Figure 5.4

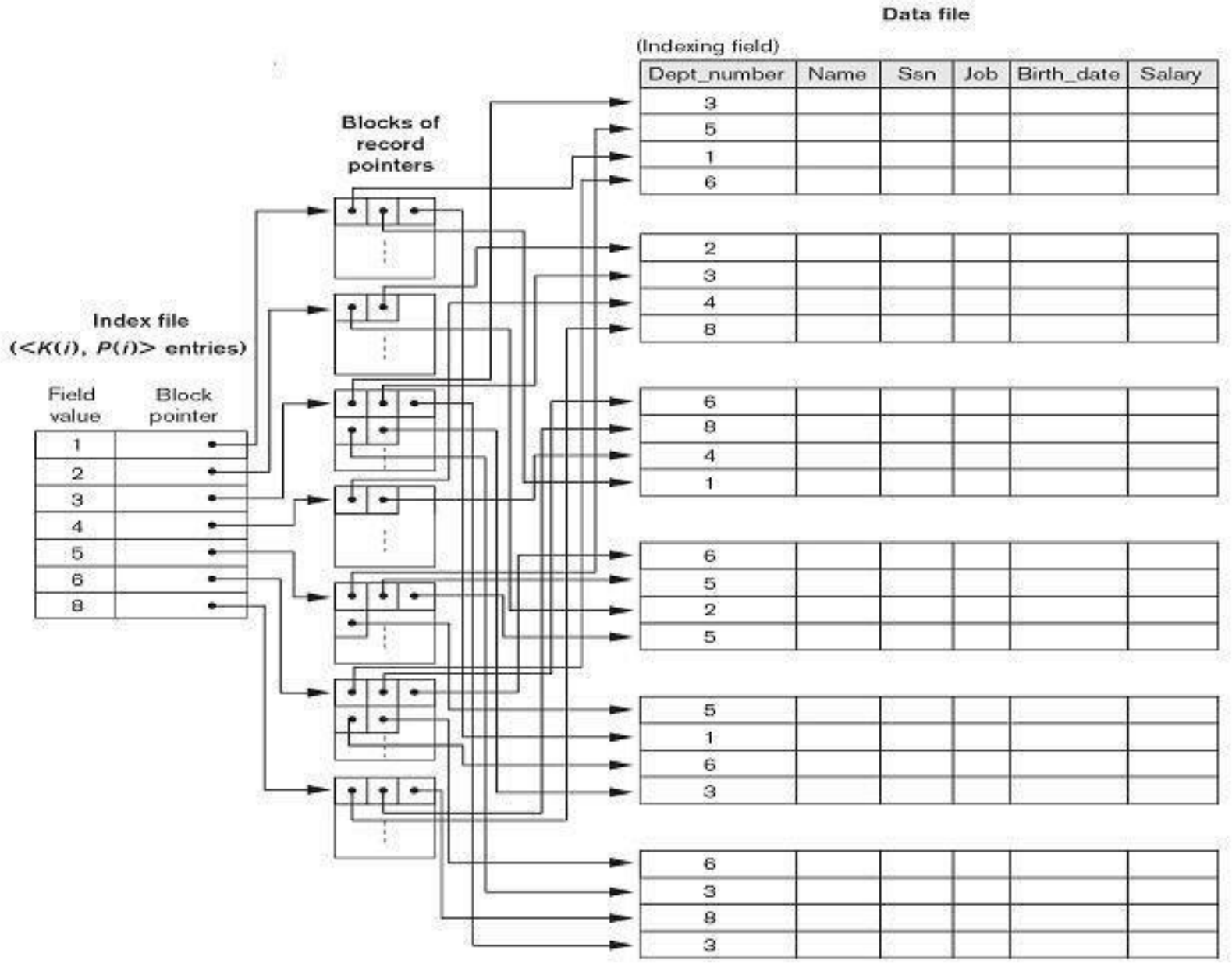


Figure 5.4: Secondary Index (with Record Pointer) on a Non Key Field implemented using one level of indirection so that Index entries are of Fixed Length and have unique field values

Table 5.1: Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

Single level and Multi-level indexing

- Because a single-level index is an ordered file, we can create a primary index to the index itself;
- In this case, the original index file is called the first-level index and the index to the index is called the second-level index.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the top level fit in one disk block.
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of more than one disk block

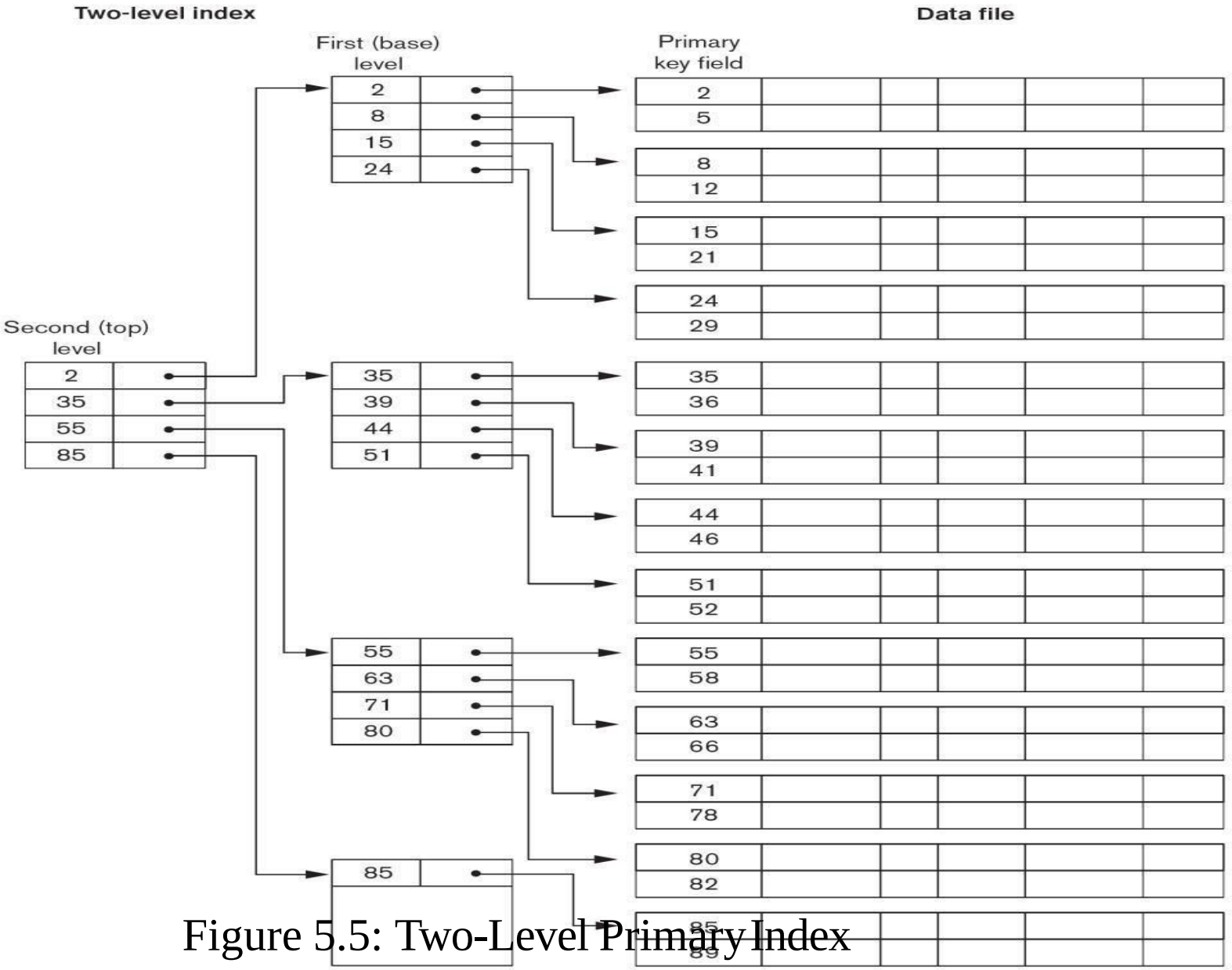


Figure 5.5: Two-Level Primary Index

Example 3

- Suppose that the dense secondary index of Example 2 is converted into a multilevel index.
- We calculated the index blocking factor bfr_i (or fo) = 68 index entries per block, which is also the **fan-out (fo)** for the multilevel index; the number of first level blocks b_1 = 442 blocks was also calculated.
- The number of second-level blocks will be

$$b2 = \lceil (b1/fo) \rceil$$
$$= \lceil (442/68) \rceil$$
$$= 7 \text{ blocks, and}$$

Contd...

- The number of third-level blocks will be

$$b3 = \lceil (b2/fo) \rceil$$
$$= \lceil (7/68) \rceil$$
$$= 1 \text{ block.}$$

- Hence, the third level is the top level of the index, and t = 3.

Contd...

- To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need

$$t + 1 = 3 + 1$$

$$= 4 \text{ block accesses.}$$

- Compare this to Example 2, where 10 block accesses were needed when a single-level index and binary search were used.

Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- B-trees and B+-trees are special cases of the well-known search data structure known as a tree.
- A tree is formed of nodes. Each node in the tree, except for a special node called the **root**, has one parent node and zero or more child nodes.
- The root node has no parent. A node that does not have any child nodes is called a **leaf node**; a nonleaf node is called an **internal node.**
- The level of a node is always one more than the level of its parent, with the level of the root node being zero.
- A subtree of a node consists of that node and all its descendant nodes its child nodes, the child nodes of its child nodes, and so on

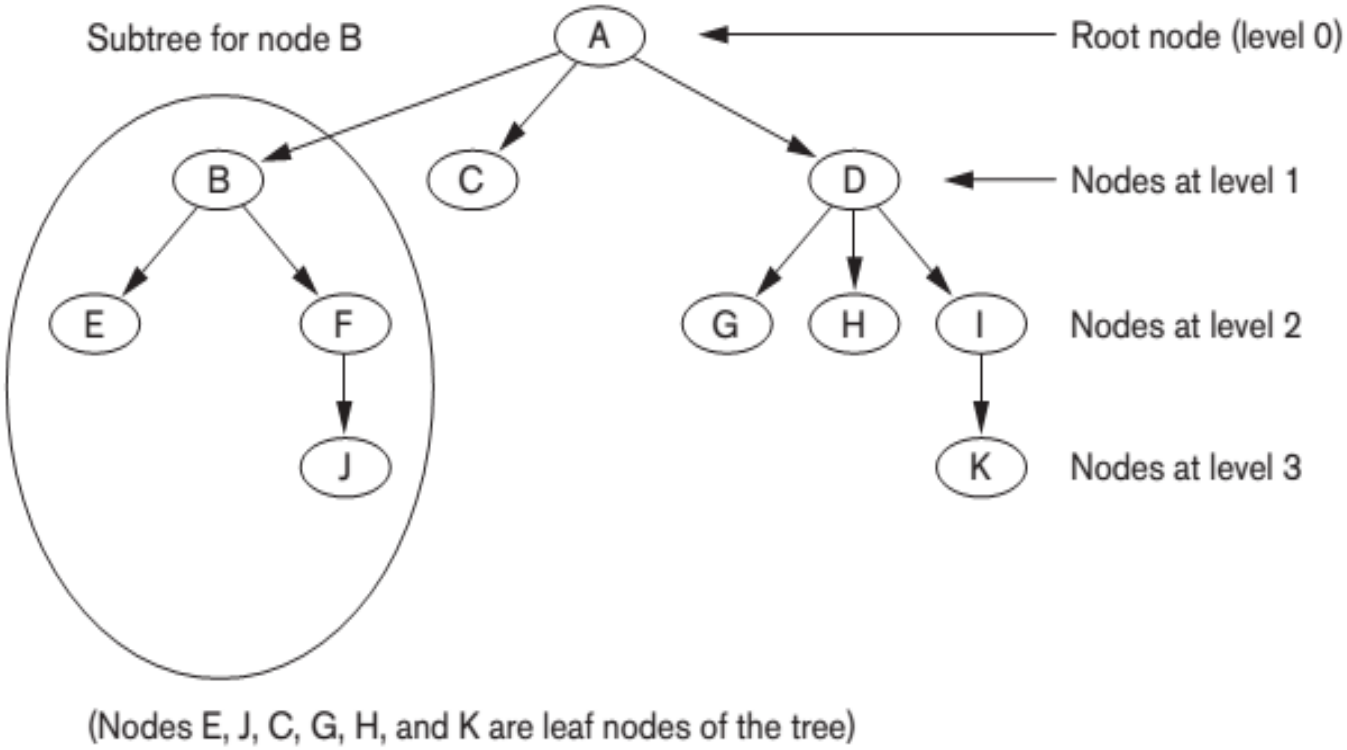


Figure 18.7
A tree data structure that shows an unbalanced tree.

- We can use a **search tree** as a mechanism to search for records stored in a disk file.
- The values in the tree can be the values of one of the fields of the file, called the search field (which is the same as the index field if a multilevel index guides the search).
- Each key value in the tree is associated with a pointer to the record in the data file having that value.
- Alternatively, the pointer could be to the disk block containing that record.
- The search tree itself can be stored on disk by assigning each tree node to a disk block.
- When a new record is inserted in the file, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

Search Trees

- A search tree is slightly different from a multilevel index.
- A search tree of **order p** is a tree such that each node contains **at most p − 1 search values** and **p pointers** in the order <P1, K1, P2, K2, ..., Pq−1, Kq−1, Pq>, where $q \leq p$.
- Each Pi is a pointer to a child node (or a NULL pointer), and each Ki is a search value from some ordered set of values.
- Two constraints must hold at all times on the search tree:
  1. Within each node, K1 < K2 < ... < Kq−1.
  2. For all values X in the subtree pointed at by Pi, we have Ki−1<X < Ki for 1<i < q; X < Ki for i=1; and Ki−1 < X for i = q

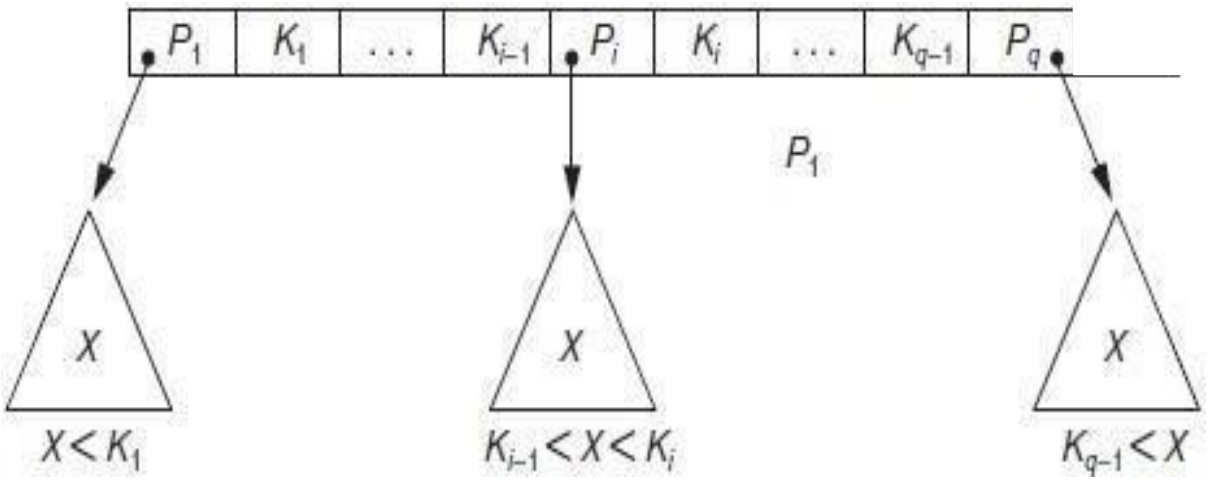


Figure 5.6: A node in a search tree with pointers to subtrees below it

B-Trees

- The B-tree has additional constraints that ensure that the tree is always balanced.
- A B-tree of order p, when used as an access structure on a key field to search for records in a data file, can be defined as follows:
  1. Each internal node in the B-tree is of the form

     <P1, <K1, Pr1>, P2, <K2, Pr2>, …, <Kq–1, Prq–1>, Pq>

     where q ≤ p. Each **Pi** is a **tree pointer**—a pointer to another node in the Btree. Each **Pri** is a **data pointer**—a pointer to the record whose search key field value is equal to Ki.

2. Within each node, K1 < K2 < ... < Kq−1.
3. For all search key field values X in the subtree pointed at by Pi, we have: Ki–1 < X < Ki for 1 < i < q; X < Ki for i = 1; and Ki–1 < X for i = q.
4. Each node has at most p tree pointers.
5. Each node, except the root and leaf nodes, has at least ⌈(p/2)⌉ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
6. A node with q tree pointers, q ≤ p, has q – 1 search key field values (and hence has q – 1 data pointers).
7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers Pi are NULL

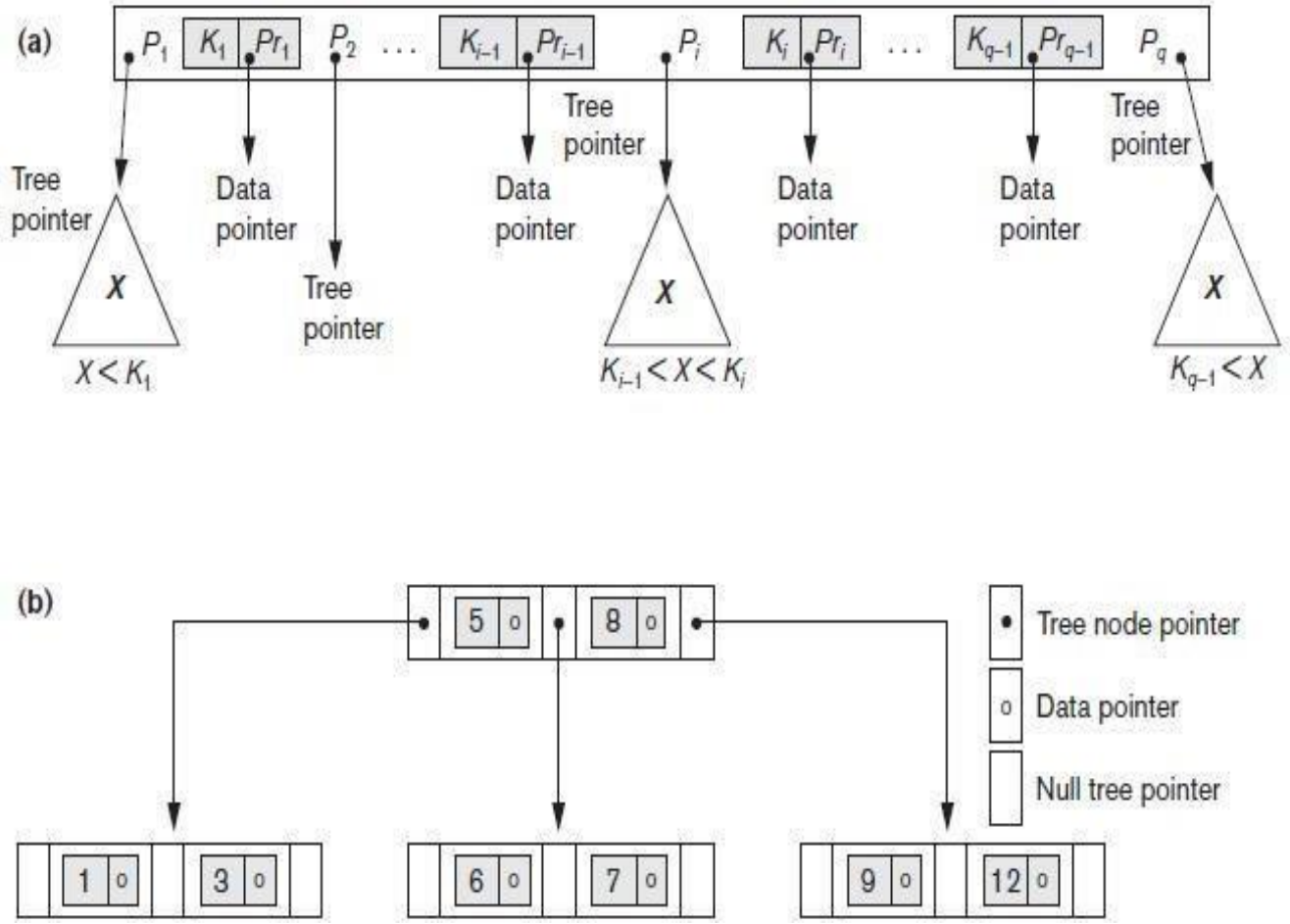


Figure 5.7: B-tree structures. (a) A node in a B-tree with $q - 1$ search values.

(b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

- A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero).
- Once the root node is full with p – 1 search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1.
- Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes.
- When a nonroot node is full and a new entry is inserted into it,
  - that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes.
- If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split.

- If deletion of a value causes a node to be less than half full,
  - it is combined with its neighboring nodes, and this can also propagate all the way to the root.
  - Hence, deletion can reduce the number of tree levels.

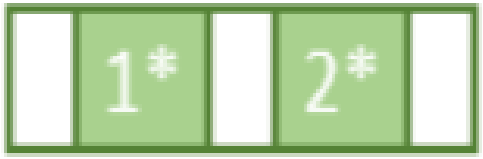
Properties of a B-tree

- For a tree to be classified as a B-tree, it must fulfill the following conditions:
  - the nodes in a B-tree of **order m** can have a **maximum of m children**
  - each internal node (non-leaf and non-root) can have **at least (m/2)** children (rounded up)
  - the root should have at least two children – unless it's a leaf
  - a non-leaf node with k children should have k-1 keys
  - all leaves must appear on the same level

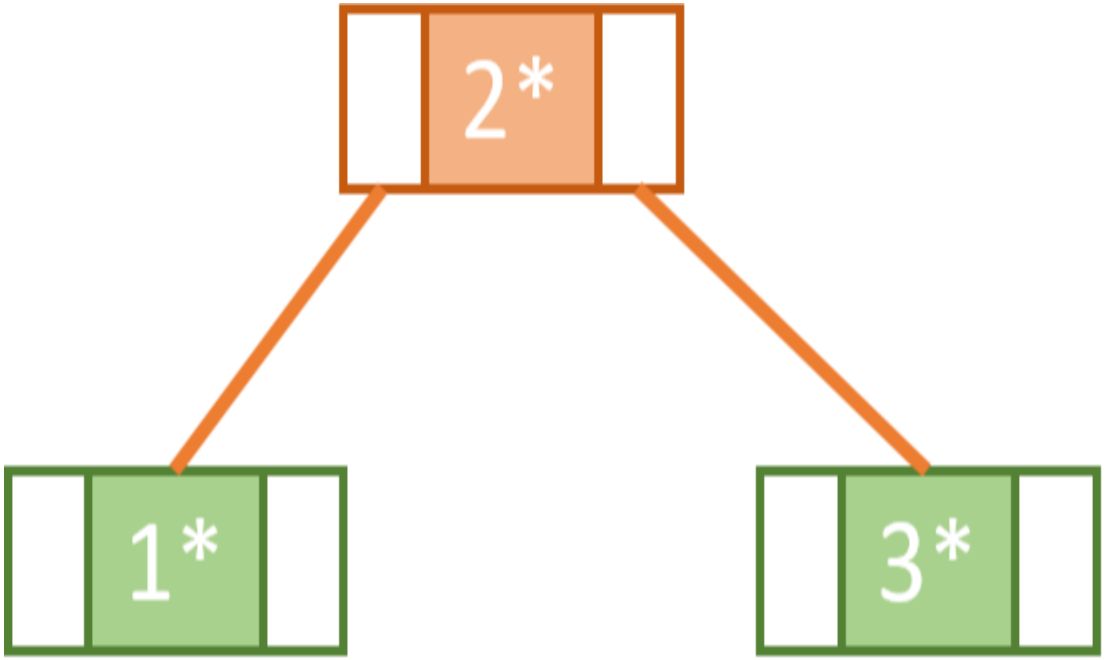
Building a B-tree

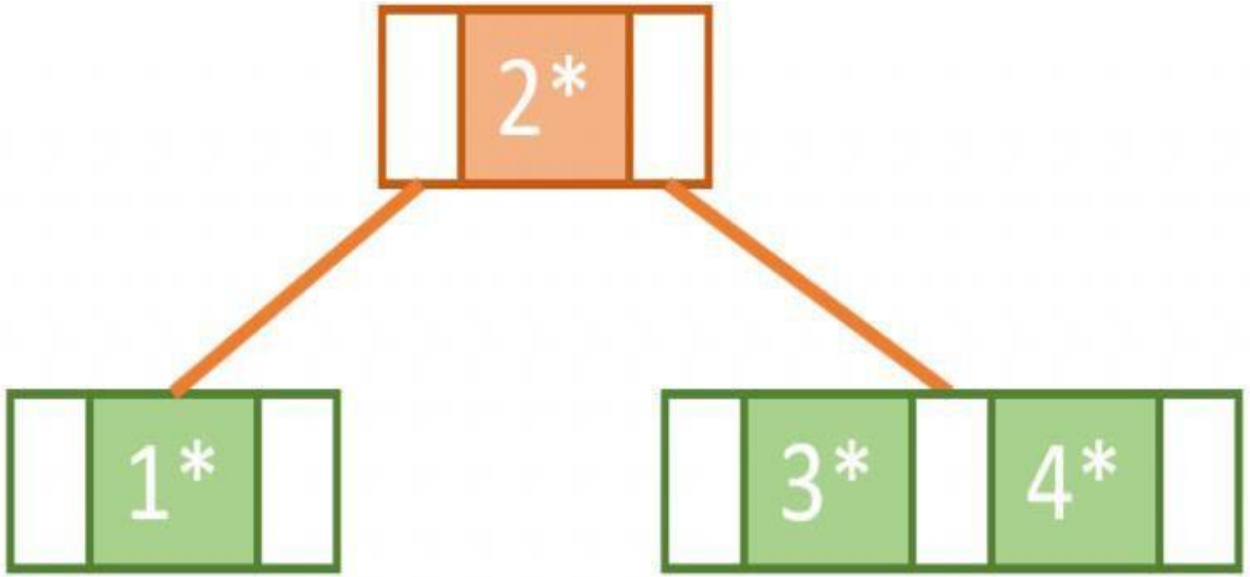
- Since we're starting with an empty tree, the first item we insert
  will become the root node of our tree.
- At this point, the root node has the key/value pair.
- The key is 1, but the value is depicted as a star to make it easier to represent, and to indicate it is a reference to a record.
- The root node also has pointers to its left and right children shown as small rectangles to the left and right of the key.
- Since the node has no children, those pointers will be empty for now:

We know that this tree has **order of 3**, so it can have **only up to 2 keys** in it. So we can add the payload with key 2 to the root node in ascending order:

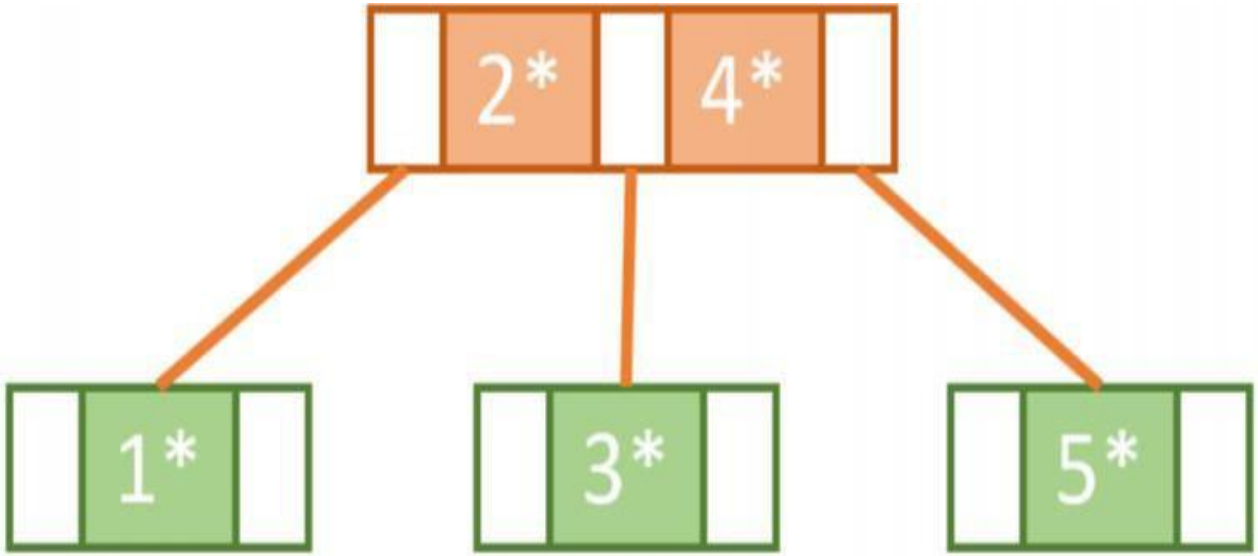


Next, if we wanted to insert 3, for us to keep the tree balanced and fulfilling the conditions of a B-tree we need to perform what is called a split operation.
We can determine how to split the node by picking the middle key.

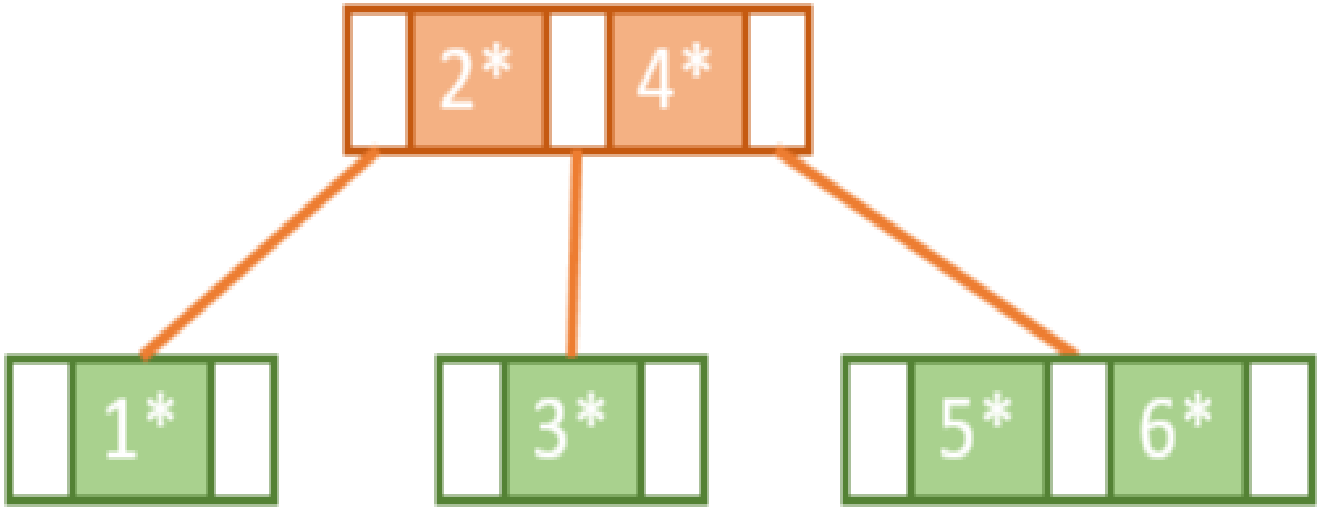

2*
1*
3*



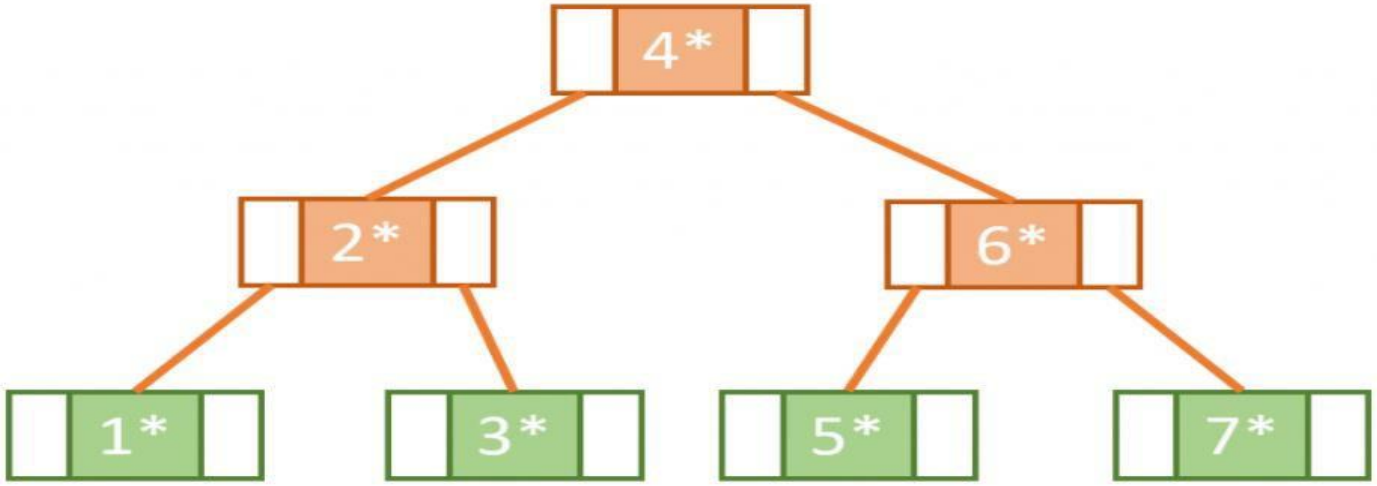
Now, let's insert 4. To determine where this needs to be placed we must remember that B-trees are organized such that sub-trees on the right have greater keys than sub-trees on the left. Consequently, Key 4 belongs in the right sub-tree. And since the right sub-tree still has the capacity, we can simply add 4 to it alongside 3 in ascending order:



Our right sub-tree is now at full capacity, so to insert 5 we need to use the same splitting logic explained above. We split the node into two so that Key 3 goes to a left sub-tree and 5 goes to a right sub-tree leaving 4 to be promoted to the root node alongside 2.



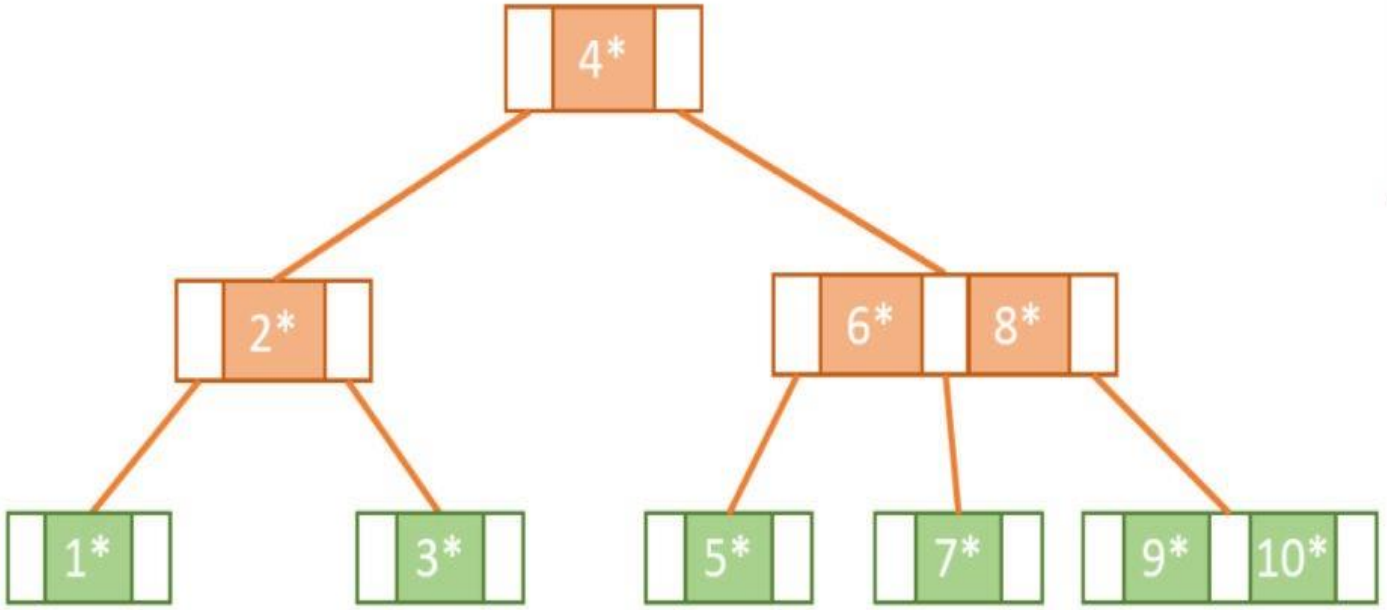
This rebalancing gives us space in the rightmost sub-tree to insert 6:



Next, we try to insert 7. However, since the rightmost tree is now at full capacity we know that we need to do another split operation and promote one of the keys. But wait! The root node is also at full capacity, which means that it also needs to be split.

So, we end up doing this in two steps. First, we need to split the right nodes 5 and 6 so that 7 will be on the right, 5 will be on the left, and 6 will be promoted.

Then, to promote 6, we need to split the root node such that 4 will become a part of new root and 6 and 2 become the parents of the right and left subtree.



Continuing in this way, we fill the tree by adding Keys 8,9 and 10 until we get the final tree:

B+-Trees

- In a B+-tree, data pointers are stored only at the leaf nodes of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes.
- The leaf nodes have an entry for every value of the search field, along with a data pointer to the record.

- The structure of the **internal nodes** of a B+ tree of **order p** is as follows:
  1. Each internal node is of the form <P1, K1, P2, K2, ..., Pq– 1, Kq –1, Pq> where $q \leq p$ and each Pi is a tree pointer.
  2. Within each internal node, $K1 < K2 < ... < Kq-1$.
  3. For all search field values X in the subtree pointed at by Pi, we have $Ki-1 < X \leq Ki$ for $1 < i < q$; $X \leq Ki$ for $i = 1$; and $Ki-1 < X$ for $i = q$.
  2. Each internal node has at most p tree pointers.
  3. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at leasttwo tree pointers if it is an internal node.
  6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

- The structure of the **leaf nodes** of a B+-tree of **order p** is as follows:
  1. Each leaf node is of the form <<K1, Pr1>, <K2, Pr2>, ..., <Kq–1, Prq–1>, Pnext> where q ≤ p, each Pri is a data pointer, and Pnext points to the next leaf node of the B+-tree.
  2. Within each leaf node, K1 ≤ K2 ... , Kq–1, q ≤ p.
  3. Each Pri is a data pointer that points to the record whose search field value is Ki or to a file block containing the record
  4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
  5. All leaf nodes are at the same level.

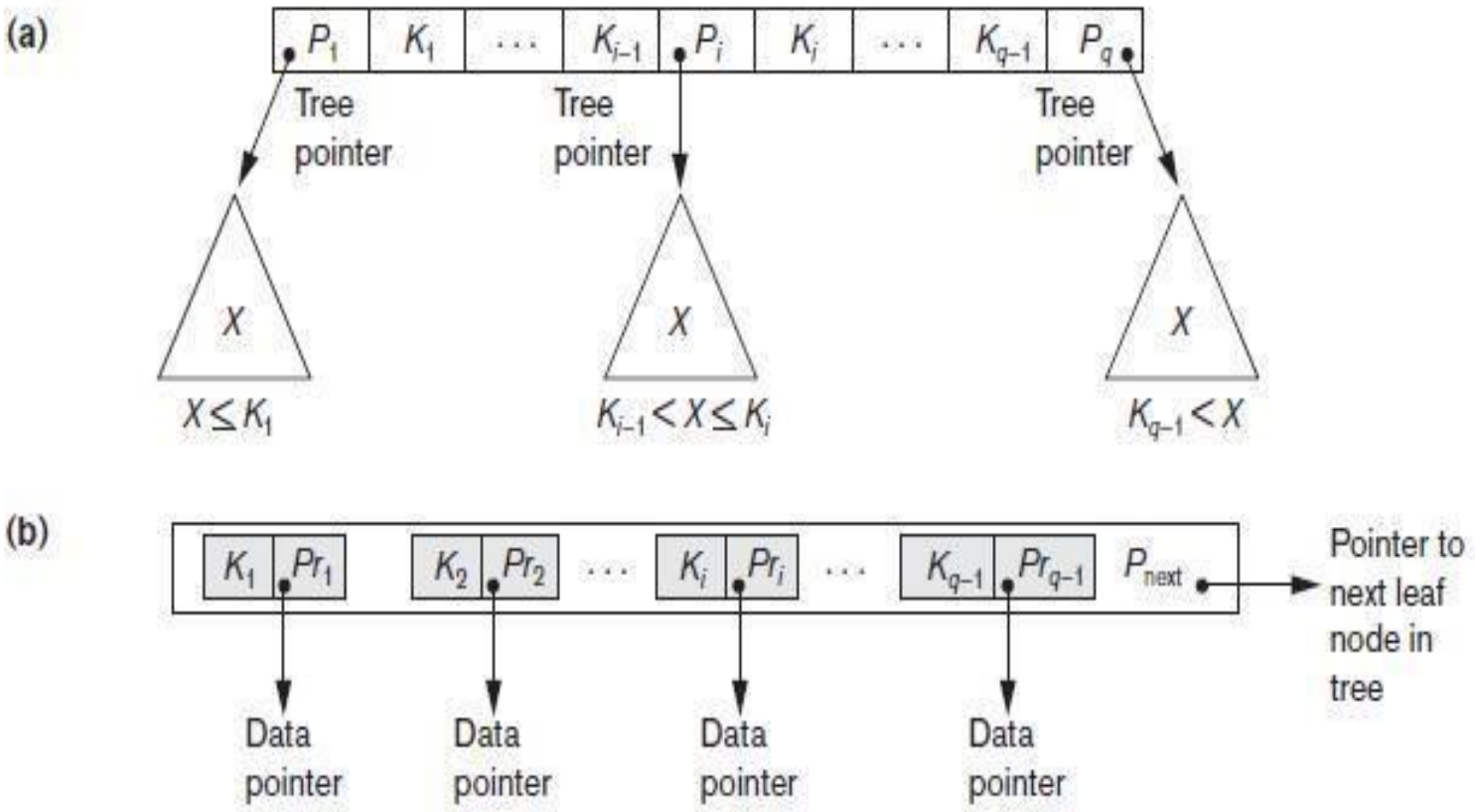
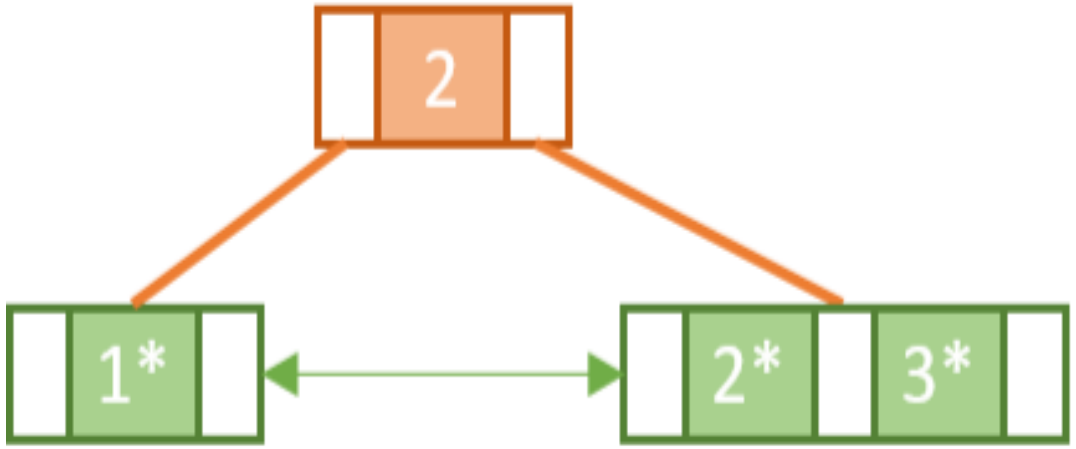


Figure 5.8: The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values.
(b) Leaf node of a B+-tree with $q - 1$ search $q - 1$ data pointers.

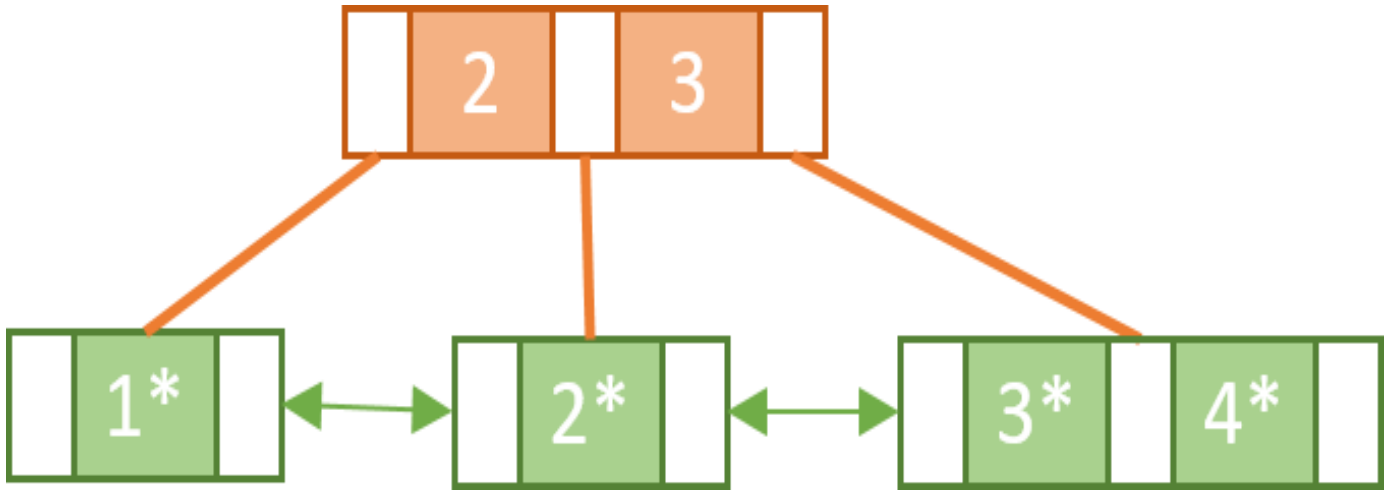
- When a leaf node is full and a new entry is inserted there, the node overflows and must be split.
- The first j = ⌐((pleaf + 1)/2)¬ entries in the original node are kept there, and the remaining entries are moved to a new leaf node.
- The jth search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent.
- These must be inserted in the parent node in their correct sequence.
- If the parent internal node is full, the new value will cause it to overflow also, so it must be split.
- The entries in the internal node up to Pj—the jth tree pointer after inserting the new value and pointer, where j = ⌊((p + 1)/2)⌋—are kept, while the jth search value is moved to the parent, not replicated.
- A new internal node will hold the entries from Pj+1 to the end of the entries in the node
- This splitting can propagate all the way up to create a new root

Building a B+tree

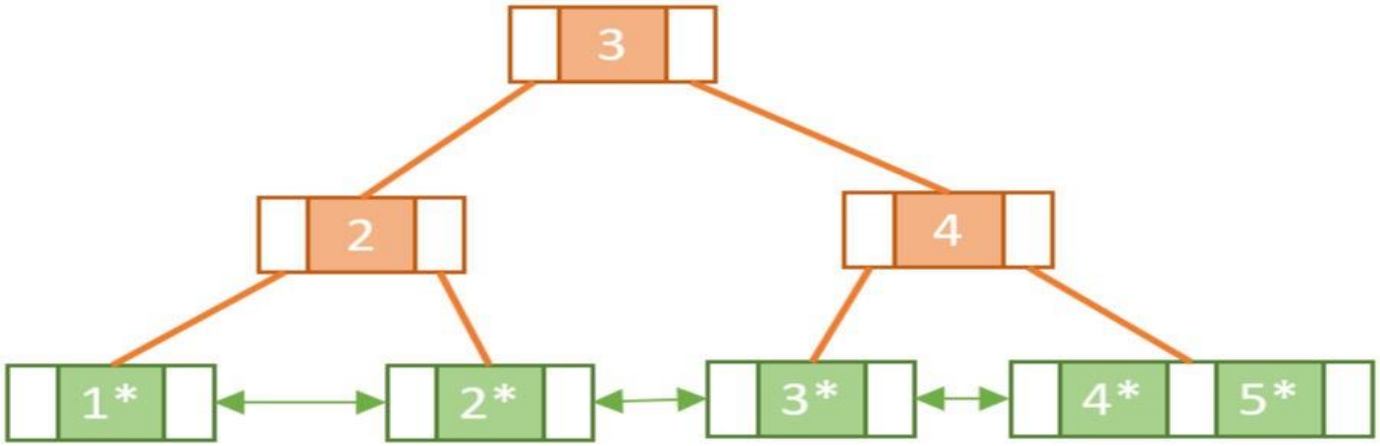
To start with, we'll insert Keys 1 and 2 into the root node in ascending order:



When we come to insert Key 3, we find that in doing so we will exceed the capacity of the root node.Similar to a normal B-tree this means we need to perform a split operation. However, unlike with the B-tree, we must copy-up the first key in the new rightmost leaf node. As mentioned, this is so we can make sure we have a key/value pair for Key 2 in the leaf nodes:

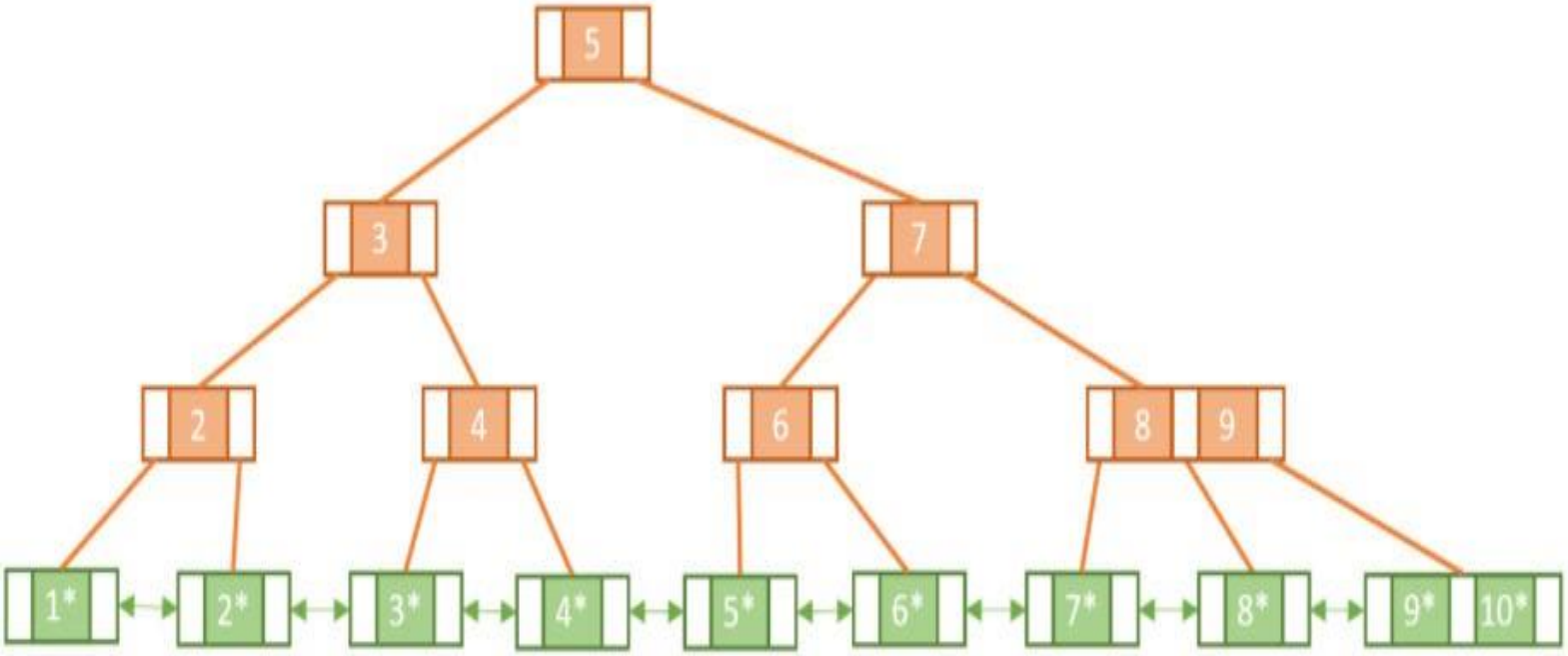


Next, we add Key 4 to the rightmost leaf node. Since it's full, we need to perform another a split operation and copy-up Key 3 to the root node:



Now, let's add 5 to the rightmost leaf node. Once again to keep the order, we'll split the leaf node and copy-up 4. Since that will overflow the root node, we'll have to perform another split operation splitting the root node into two nodes and promoting 3 into a new root node

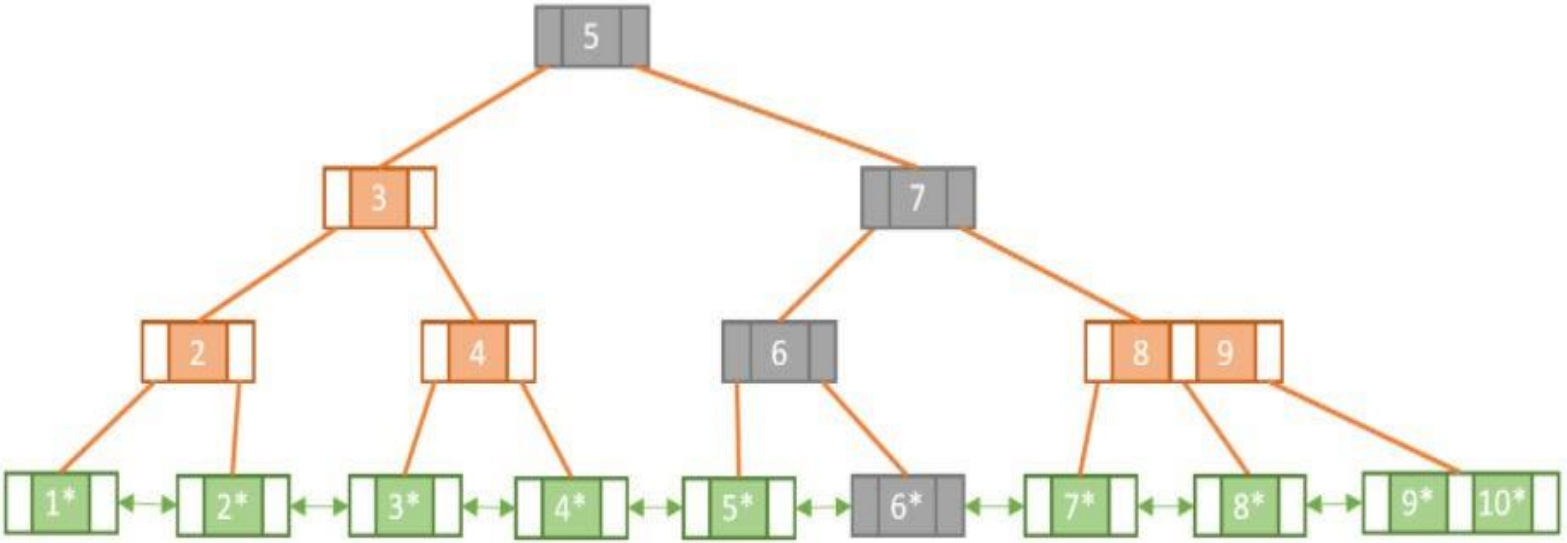
Notice the difference between splitting a leaf node and splitting an internal node. When we split the internal node in the second split operation we didn't copy-up Key 3.



In the same way, we keep adding the keys from 6 to 10, each time splitting and copying-up when necessary until we have reached our final tree:

Searching a B+tree

- Searching for a specific key within a B+tree is very similar to searching for a key in a normal B-tree.
- Let's see what it would be like to search for Key 6 again but on the
  B+tree:



- The shaded nodes show us the path we have taken in order to find our match. Deduction tells us that searching within a B+tree means that we must go all the way down to a leaf node to get the satellite data. As opposed to B-trees where we could find the data at any level.
- In addition to exact key match queries, B+trees support range queries. This is enabled by the fact that the B+tree leaf nodes are all linked together. To perform a range query all we need to do is:
  - find an exact match search for the lowest key
  - and from there, follow the linked list until we reach the leaf node with the maximum key

Example of a Deletion in a B +-tree

- When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there.
- In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree.
- Deletion may cause underflow by reducing the number of entries in the leaf node to below the minimum required.
- In this case, we try to find a sibling leaf node
  - a leaf node directly to the left or to the right of the node with underflow and
  - redistribute the entries among the node and its sibling so that both are at least half full;
  - otherwise, the node is merged with its siblings and the number of leaf

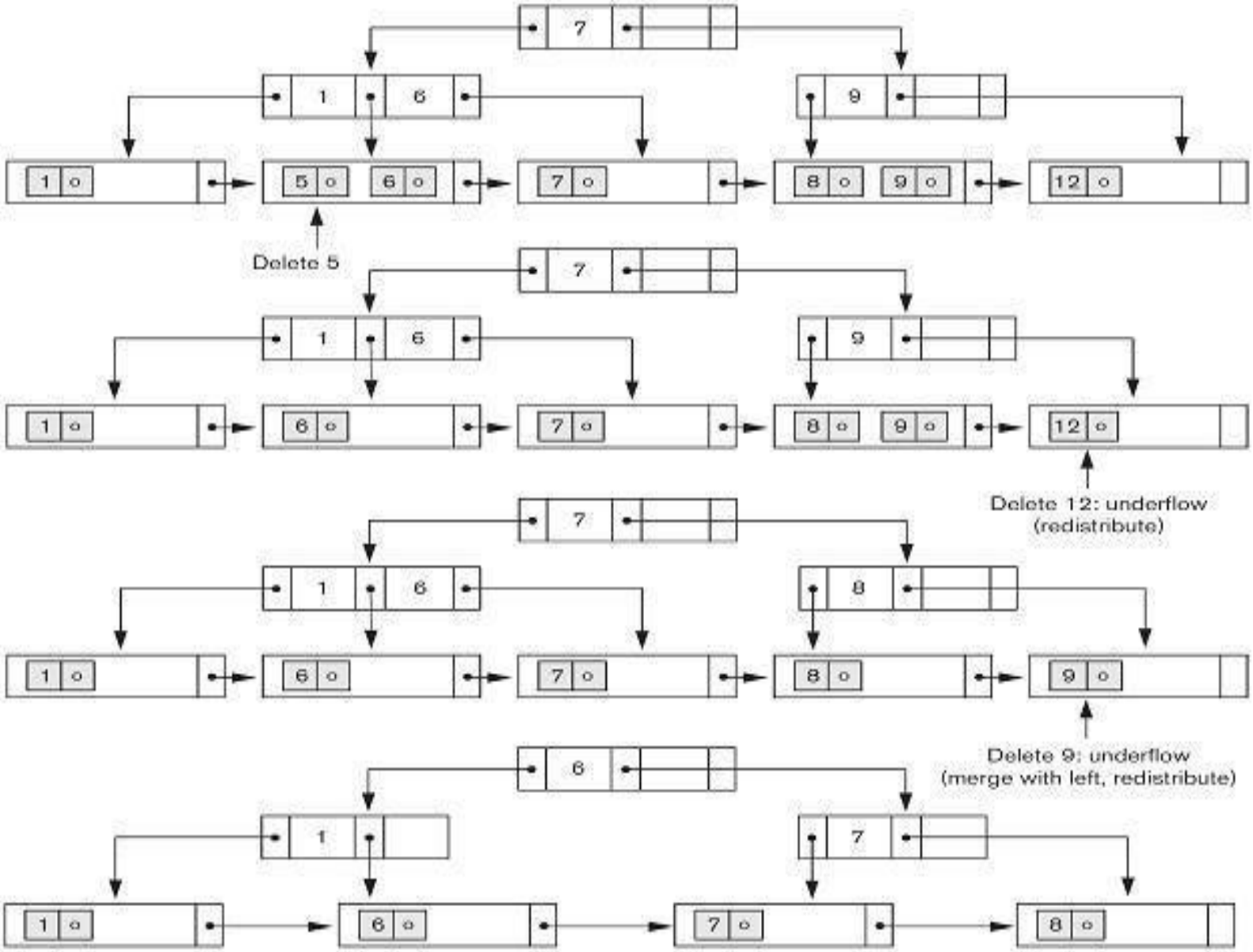


Figure 5.10:An example of deletion from a B+-tree with Deletion
sequence: 5, 12, 9

- Most **multi-level indexes** use **B-tree or B+-tree** data structures because of the insertion and deletion problem.
- This leaves space in each tree node (disk block) to allow for new index entries.
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, **each node corresponds to a disk block.**
- Each node is kept between half-full and completely full.
- An insertion into a node that is not full is quite efficient.
- If a node is full the insertion causes a split into two nodes.
- Splitting may propagate to other tree levels.
- A deletion is quite efficient if a node does not become less than half full.
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

Difference between B-tree and B+-tree

- In B+trees, search keys can be repeated but this is not the case for B-trees
- B+trees allow data to be stored in leaf nodes only, whereas B-trees store data in both leaf and internal nodes
- In B+trees, data stored on the leaf node makes the search more efficient since we can store more keys in internal nodes
  - this means we need to access fewer nodes
- Deleting data from a B+tree is easier and less time consuming because we only need to remove data from leaf nodes
- Leaf nodes in a B+tree are linked together making range search operations efficient and quick

- Finally, although B-trees are useful, B+trees are more popular. In fact, 99% of database management systems use B+trees for indexing.
- This is because the B+tree holds no data in the internal nodes.
- This maximizes the number of keys stored in a node thereby minimizing the number of levels needed in a tree.
- Smaller tree depth invariably means faster search.

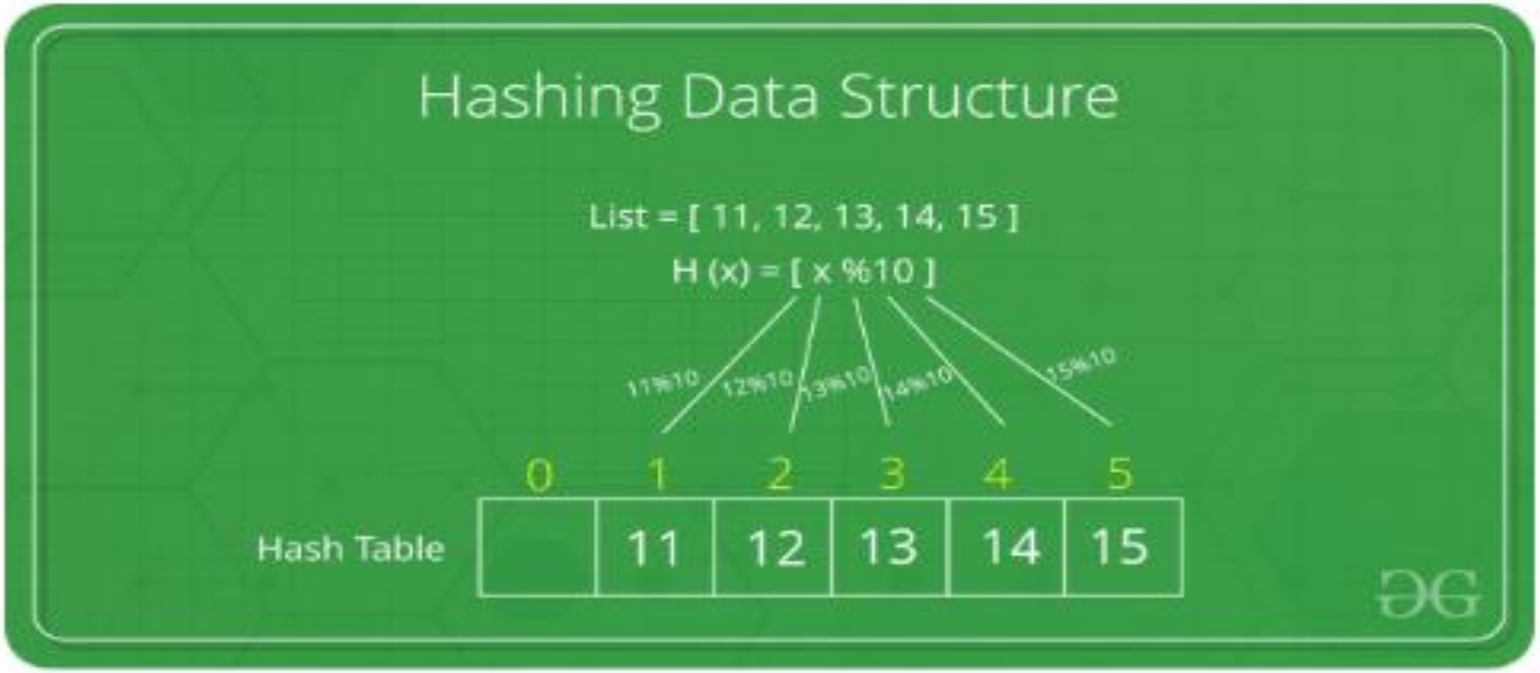
Hashing

Hashing is a technique or process of mapping keys, values into the hash table by using a **hash function**. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a **hash function H(x)** maps the value at the index in an Array.

H(x) = x%10

For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values(ie. 0,1,2,3,4). The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.(Example in slide No 73)

Operations

Insertion – When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.

Bucket address = h(K)

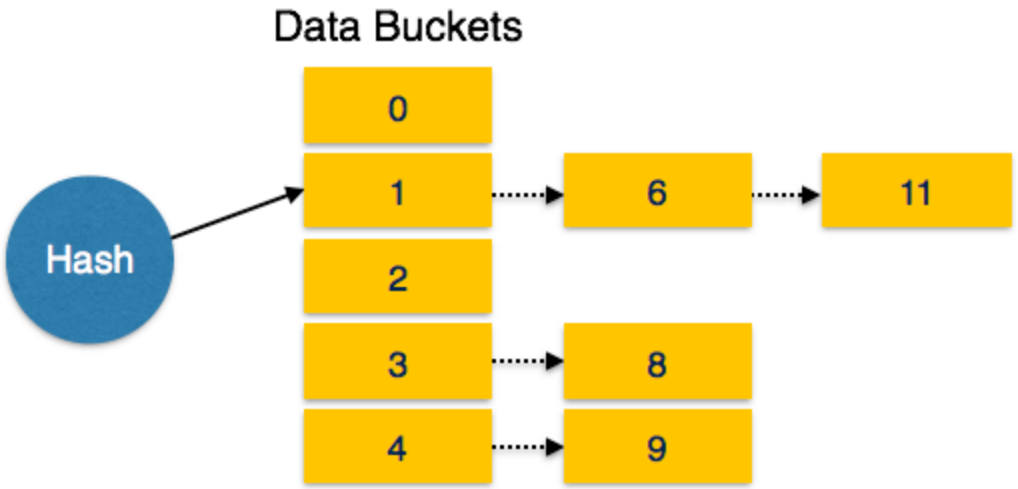
Search – When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.

Delete – This is simply a search followed by a deletion operation.

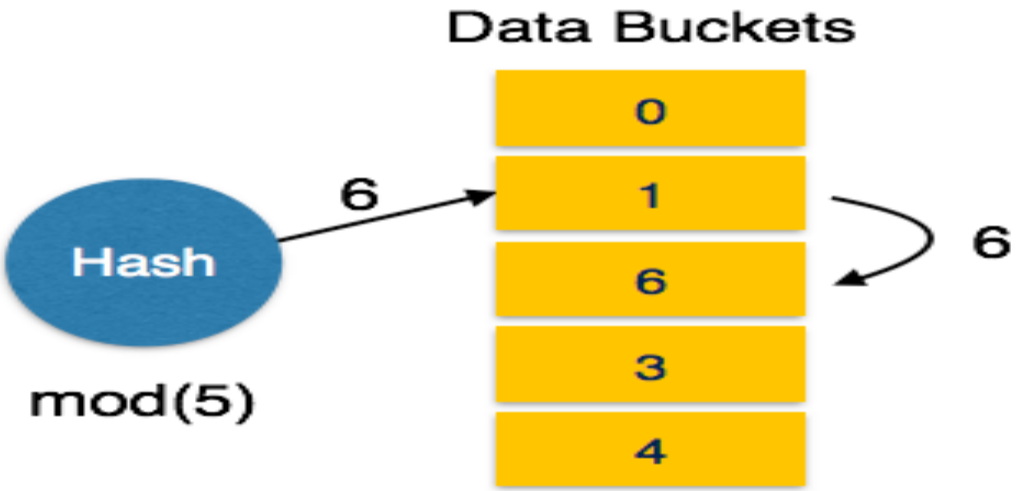
Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

Overflow Chaining – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



Linear Probing – When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



Dynamic Hashing or Extendible Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. **Dynamic hashing** is also known as **Extendible Hashing**.

Main features of Extendible Hashing: The main features in this hashing technique are:

Directories: The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.

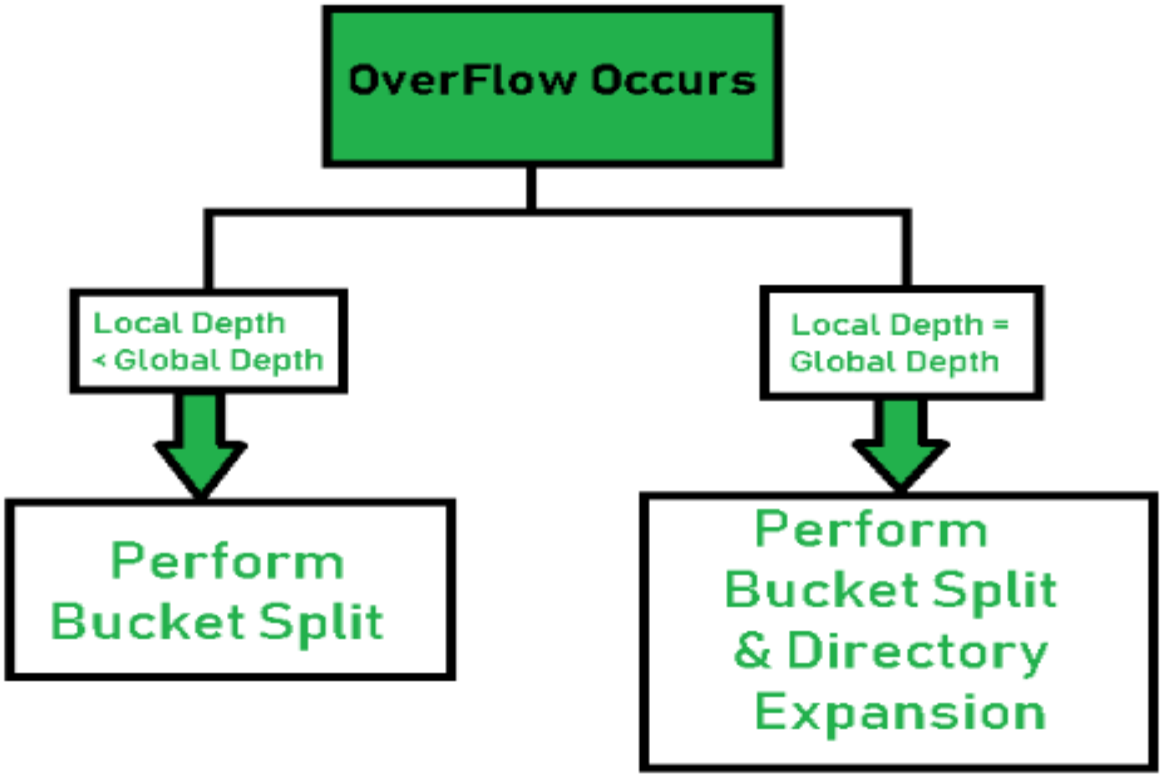
Buckets: They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.

Global Depth: It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.

Local Depth: It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.

Bucket Splitting: When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.

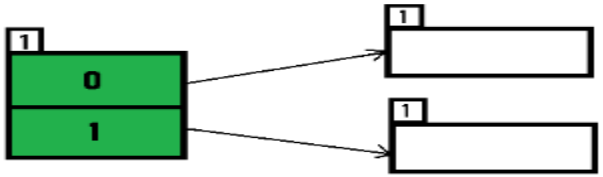
Directory Expansion: Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket **is equal to** the global depth.


OverFlow Occurs
Local Depth
< Global Depth
Local Depth =
Global Depth
Perform
Bucket Split
Perform
Bucket Split
& Directory
Expansion

Example based on Extendible Hashing: Now, let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**
Bucket Size: 3 (Assume)
Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

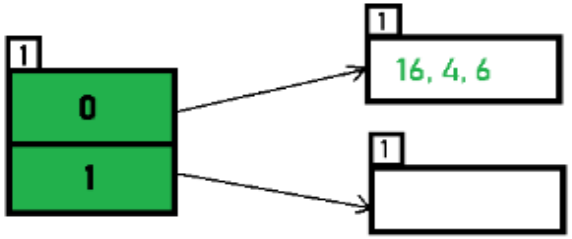
Solution: First, calculate the binary forms of each of the given numbers.
16- 10000
4- 00100
6- 00110
22- 10110
24- 11000
10- 01010
31- 11111
7- 00111
9- 01001
20- 10100
26- 11010

Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:

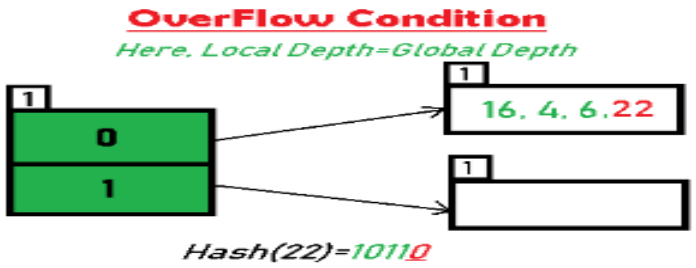


Inserting 16,4,6:

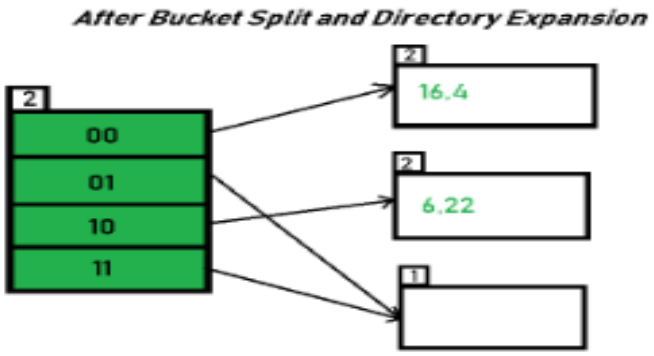
The hash function returns 1 LSB of 16(1000**0),** 4(10**0**) and 6(11**0**)have 0 in their LSB . So it is mapped to the directory with id=0.



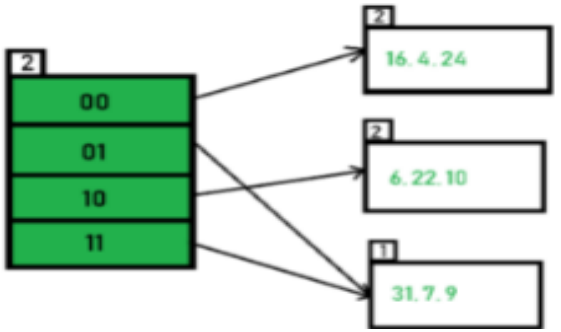
Inserting 22: The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs since bucket capacity is only 3.



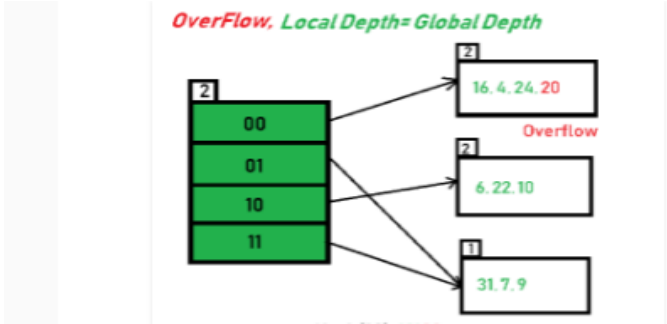
Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(1011**0**)]



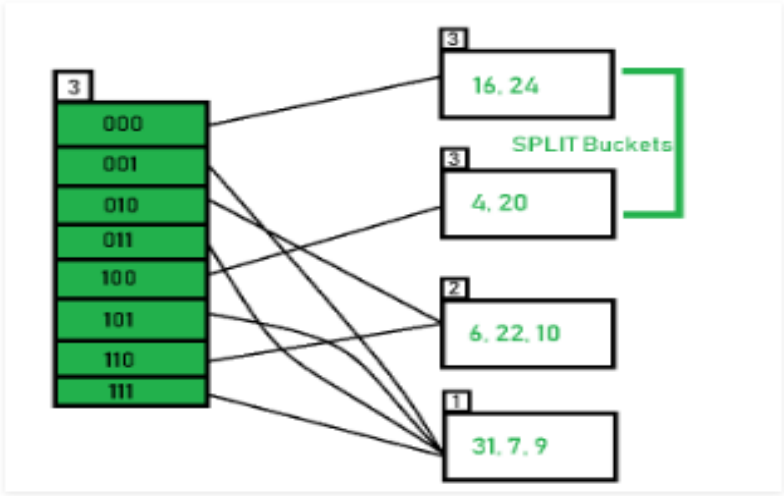
Inserting 24,10 31,7,9: 24(11**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition. Similarly the elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



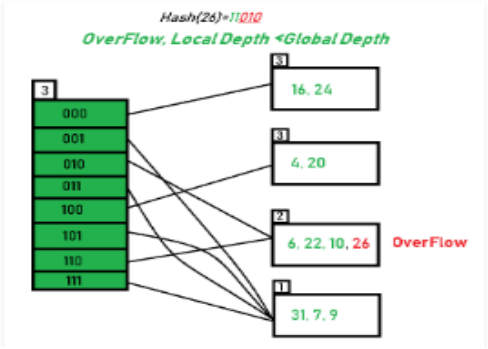
Inserting 20: Insertion of data element 20 (101**00**) will again cause the overflow problem.



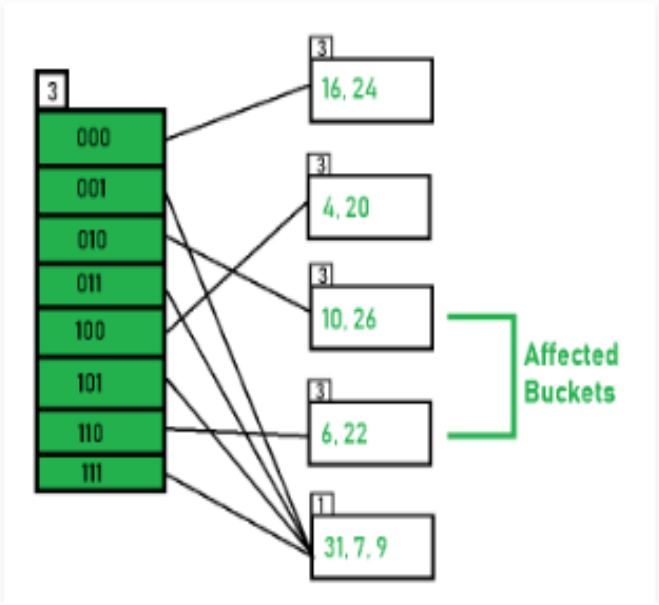
20 is inserted in bucket pointed out by 00. Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



Inserting 26: Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



The bucket overflows, and, since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed. Finally, the output of hashing the given list of numbers is obtained.



Indexes on Multiple Keys

- Till now we have assumed that the primary or secondary keys on which files were accessed were single attributes (fields).
- In many retrieval and update requests, multiple attributes are involved.
- Keys containing multiple attributes as composite keys

For example, consider an EMPLOYEE file containing attributes Dno (department number), Age, Street, City, Zip_code, Salary and Skill_code, with the key of Ssn (Social Security number). Consider the query: List the employees in department number 4 whose age is 59.

Note that both Dno and Age are nonkey attributes, which means that a search value for either of these will point to multiple records. The following alternative search strategies may be considered:

1. Assuming Dno has an index, but Age does not, access the records having Dno = 4 using the index, and then select from among them those records that satisfy Age = 59.
2. Alternately, if Age is indexed but Dno is not, access the records having Age = 59 using the index, and then select from among them those records that satisfy Dno = 4.
3. If indexes have been created on both Dno and Age, both indexes may be used; each gives a set of records or a set of pointers (to blocks or records). An intersection of these sets of records or pointers yields those records or pointers that satisfy both conditions.

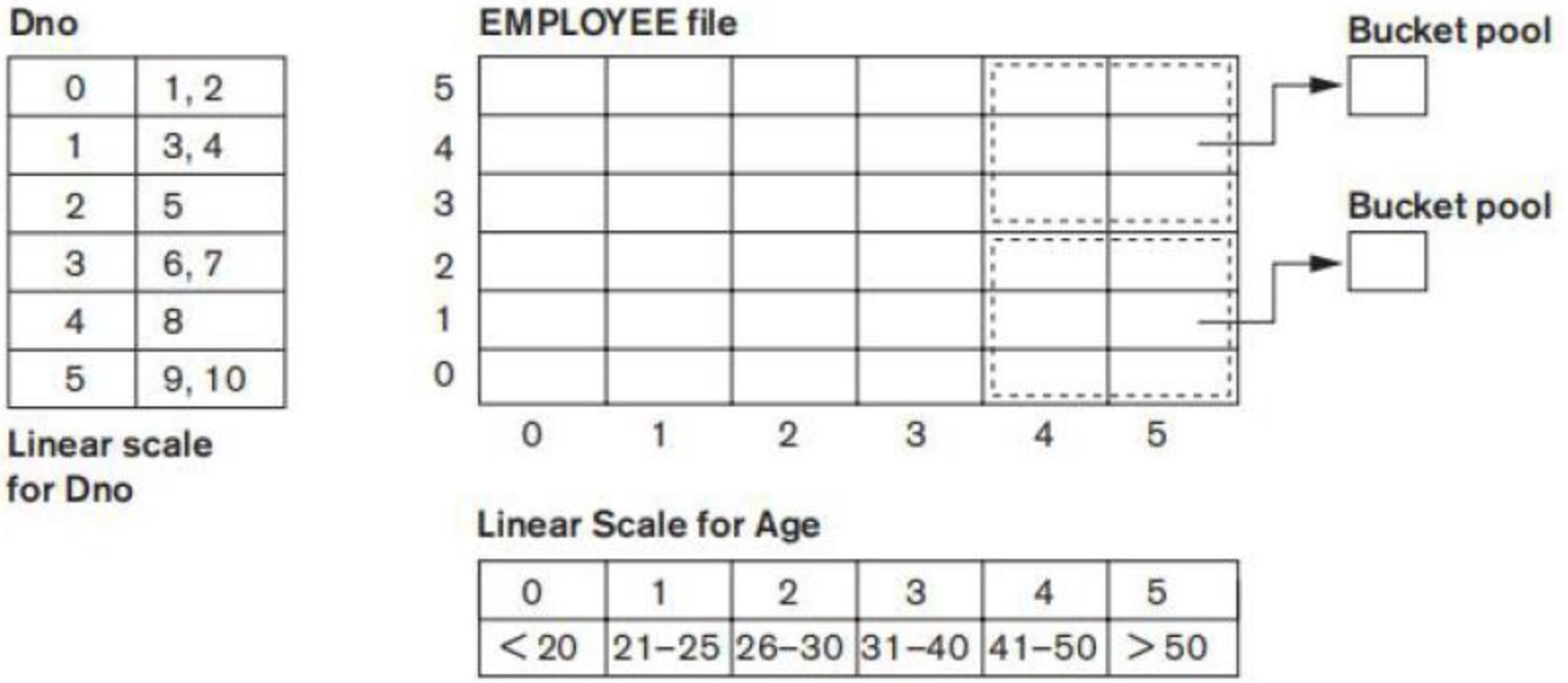
All of these alternatives eventually give the correct result. However, if the set of records that meet each condition (Dno = 4 or Age = 59) individually are large, yet only a few records satisfy the combined condition, then none of the above is an efficient technique for the given search request.

Grid Files

- For n search keys, the grid array would have n dimensions.
- The grid array thus allows a partitioning of the file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions.
- Grid files perform well in terms of reduction in time for multiple key access.
- However, they represent a space overhead in terms of the grid array structure.

If we want to access a file on two keys, say Dno and Age as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes. Figure below shows a grid array for the EMPLOYEE file with one linear scale for Dno and another for the Age attribute. The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for Dno has Dno = 1, 2 combined as one value 0 on the scale, while Dno = 5 corresponds to the value 2 on that scale. Similarly, Age is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age. The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored.

Example of a grid array on Dno and Age attributes.



Thus our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket.