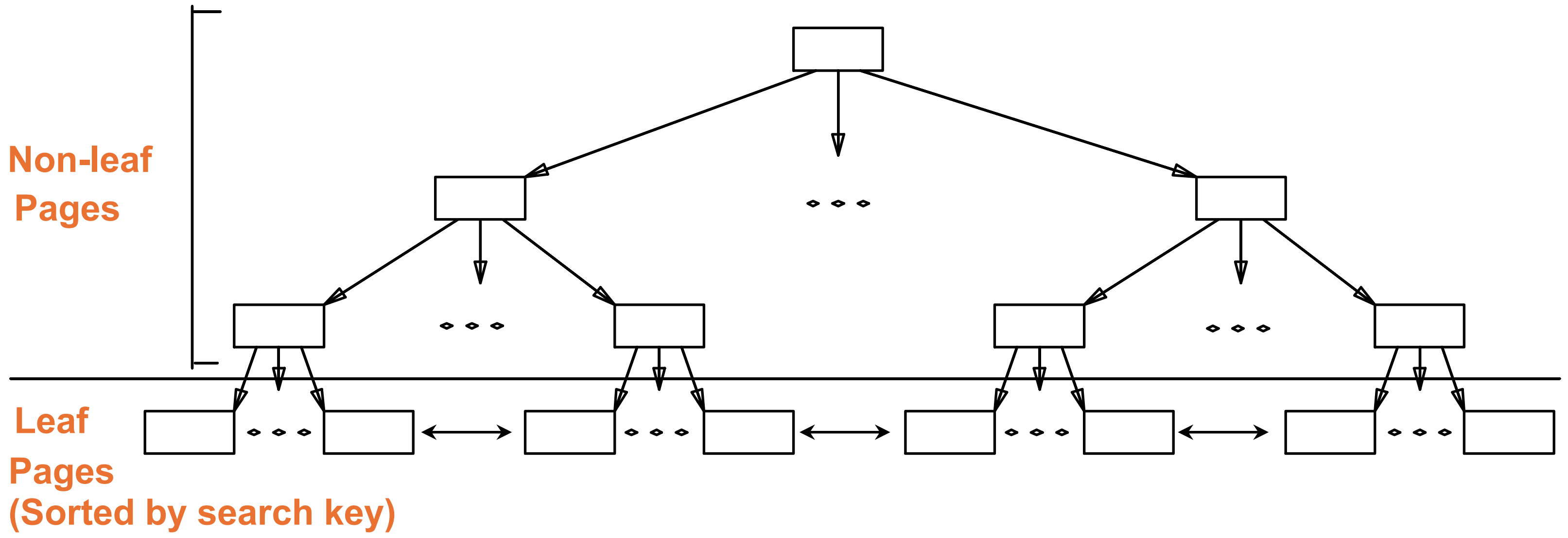
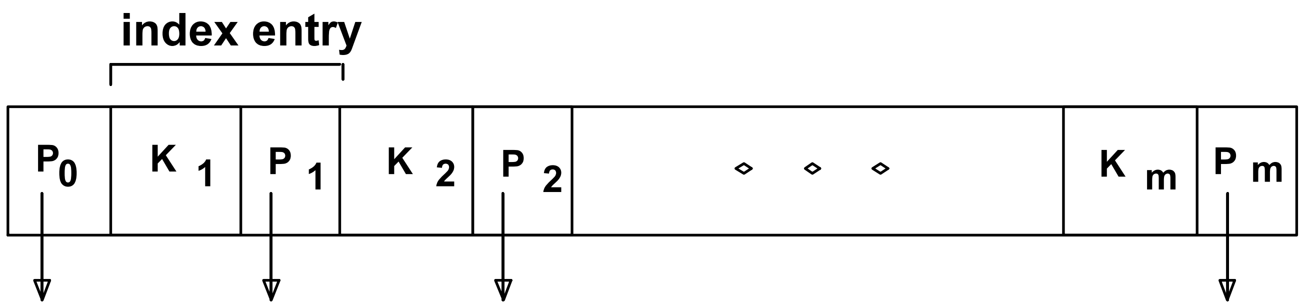
Indexes

B^+ Tree

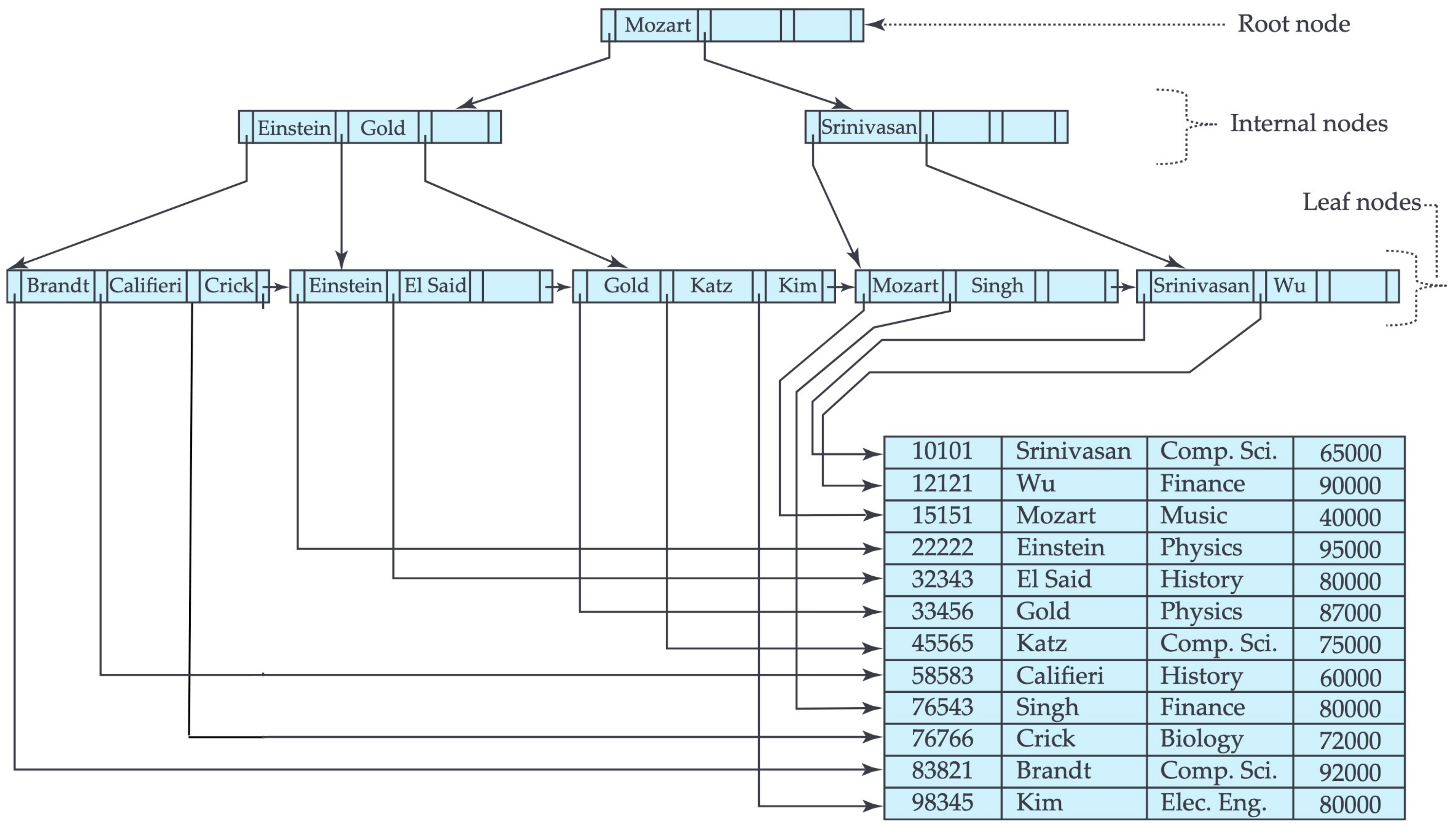
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries;* only used to direct searches:



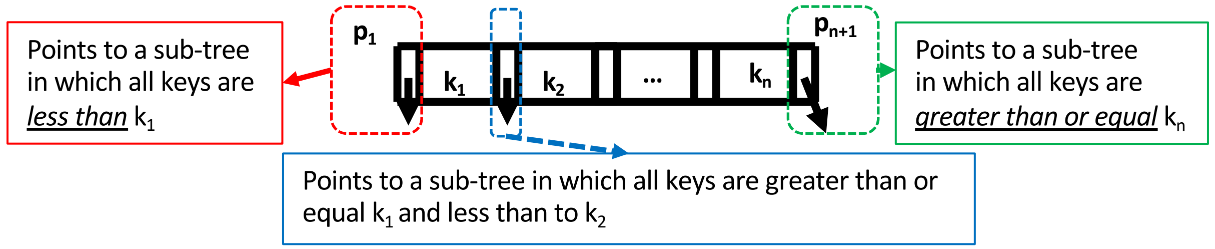
B+ Tree Indexes



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

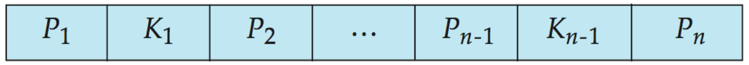
B $^+$ Tree

- Among the **most successful dynamic index** schemes is the B+ tree
- Each node in a B$^+$ tree of order ***d*** (this is a measure of the capacity of a tree):
  - Has at most ***2d*** keys
  - Has at least ***d*** keys (except the root, which may have just 1 key)
  - All leaves are on the same level
  - Has exactly ***n-1*** keys if the number of pointers is ***n***


Points to a sub-tree in which all keys are *less than* k_1
p_1
k_1
k_2
…
k_n
p_{n+1}
Points to a sub-tree in which all keys are *greater than or equal* k_n
Points to a sub-tree in which all keys are greater than or equal k_1 and less than to k_2

B + Tree

- **Typical node**



  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered

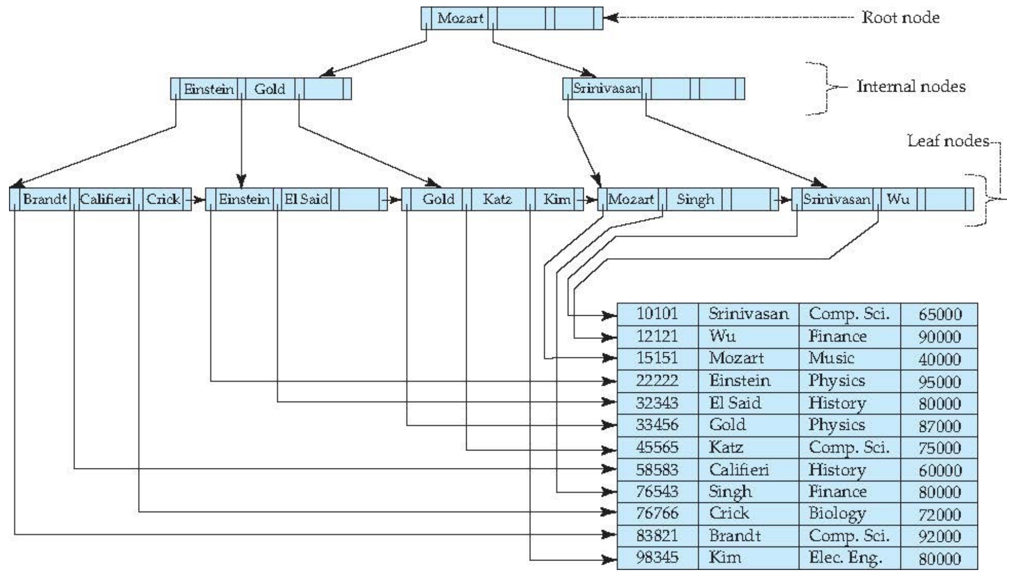
$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

B + Tree

B+ tree is a tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between(*n*/2) and *n* children (**pointers**).
  - **n is given and is the same for all nodes.**
  - n is called the order of the tree.
    - **Some books refer to (n/2) as the order of the tree.**
- A leaf node has between (*n*–1)/2 and *n*–1 **values**
- Special cases:
  - If the root is not a leaf, it has at least 2 **children**.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (*n*–1) **values**.

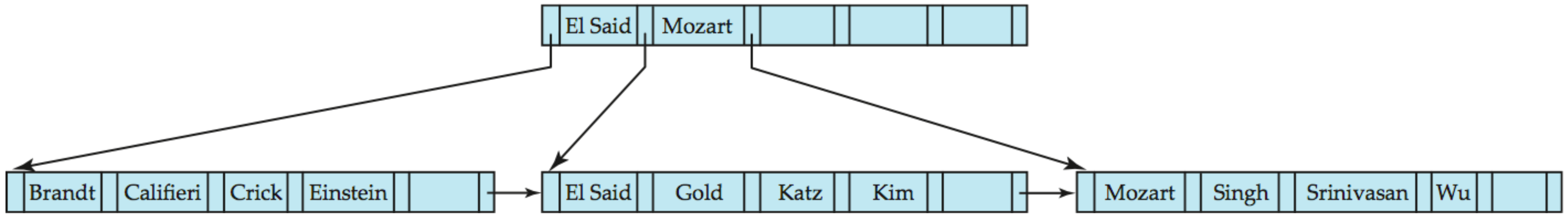


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B $^+$ Tree

Example:

B$^+$ tree for *person* file (n = 6 pointers → n – 1 = 5 keys (values))



- Leaf nodes must have between 3 and 5 values

  ((n–1)/2) and n –1, with n = 6.

- Non-leaf nodes other than root must have between 3 and 6 children

  ((n/2) and n with n =6.

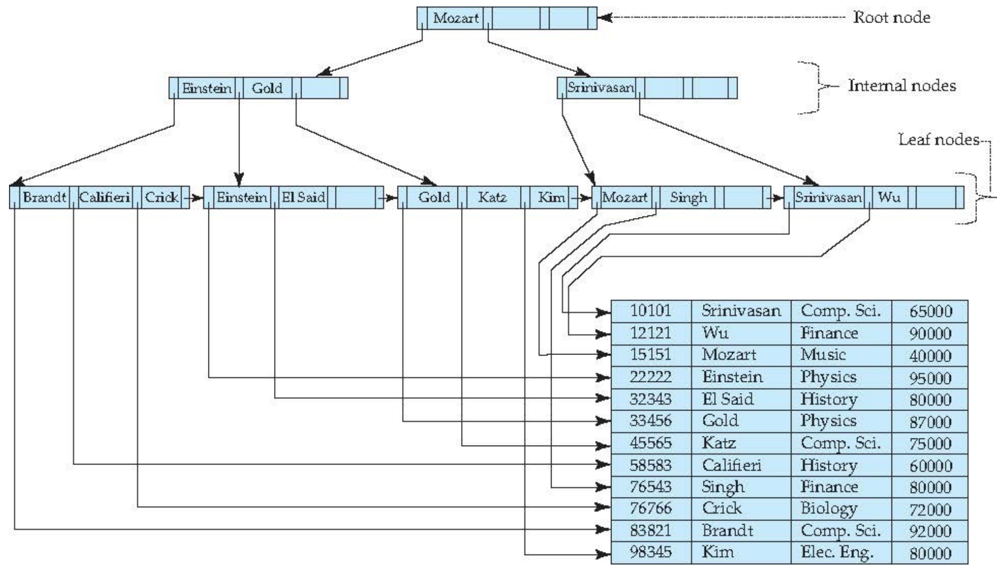
- Root must have at least 2 children.

B $^+$ Tree - Observations

- Since the inter-node connections are done by pointers,
- The non-leaf levels of the B$^+$ tree form a hierarchy of sparse indices.
- The B$^+$ tree contains a relatively small number of levels
    - Level below root has at least **2* (n/2) values**
    - Next level has at least **2* (n/2) * (n/2) values**
    - .. etc.
  - If there are *K* search-key values in the file, the tree height is no more than **($\log_{(n/2)}(K)$),** thus searches can be conducted efficiently.

B + Tree - Observations

- If the node is leaf
  - P_i points to a record with Ki value or P_i points to a record container with K_i
    - necessary if the search key is not primary key
  - P_n points to the next node (same level)
- If the node is not leaf
  - forms a sparse multi-level index for the leaf
  - all keys in the subtree of P_1 are less than K_1
  - in general, all keys in the P_i subtree are $\geq K_{i-1}$ and $<K_i$
  - the keys $\geq K_{n-1}$ are in the subtree pointed out by P_n



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- in indexed sequential files, performance degrades as the file grows, - we need to rearrange the entire file from time to time
- In B+ trees, only minor local changes are made, which are necessary when inserting and removing

B $^+$ Tree

INSERTION

B^+ Tree

INSERTIONS

```
IF it exists,
        THEN add record to file and insert pointer in container
        ELSE add record and create new container
            Insert the entry (value, pointer) on the leaf
              IF the leaf exceeds the maximum of n-1 values
              THEN **Separate the n values into 2 nodes**
```

B^+ Tree

INSERTIONS

- **Separation of n values into 2 nodes**
  - leave the first (n/2) values on the original node, the others move to a new node
  - insert the entry (k,p) on the **ascending node,** where
    - k is the smallest key of the new node
    - p is a pointer to the new node

```
IF the ascending node has exceeded the maximum size,
THEN    break it according to the same procedure
        IF  the division spreads to the root
        THEN  the root is broken into 2 nodes
              A new root is created as the ascendant of these 2 nodes
```

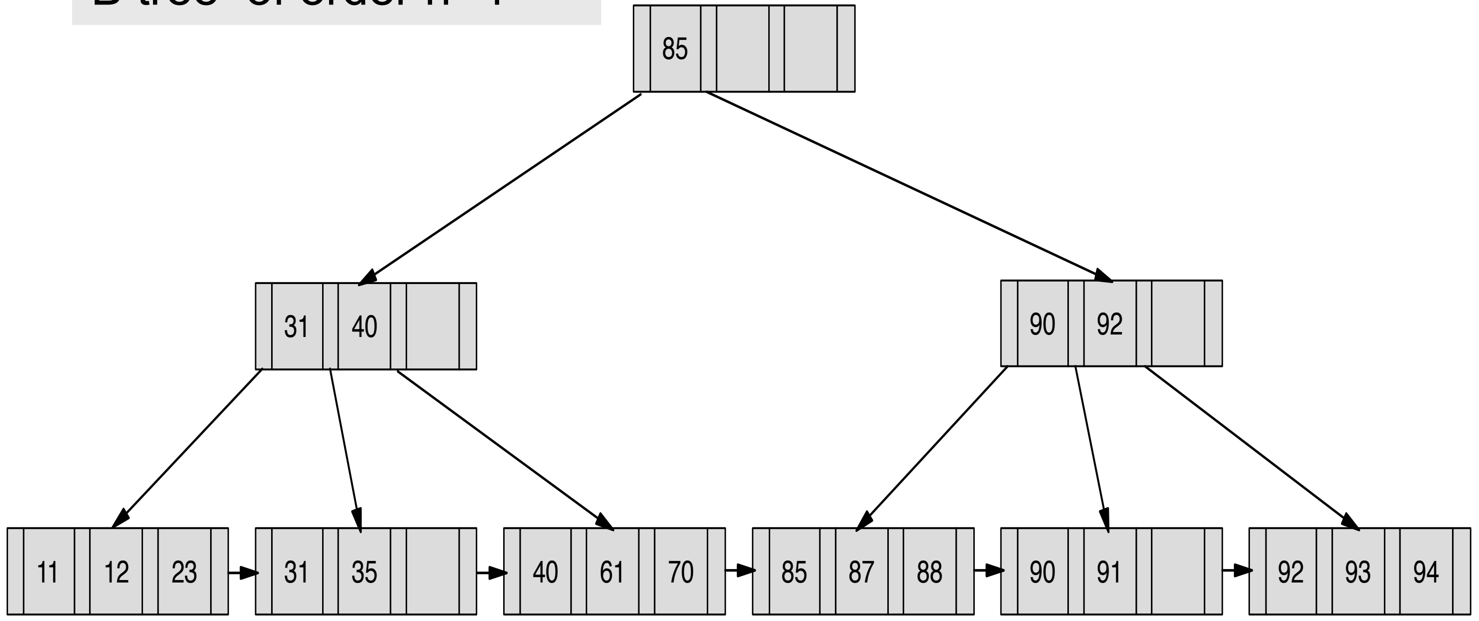
B + Tree

B $^{+}$ Tree

Insertion

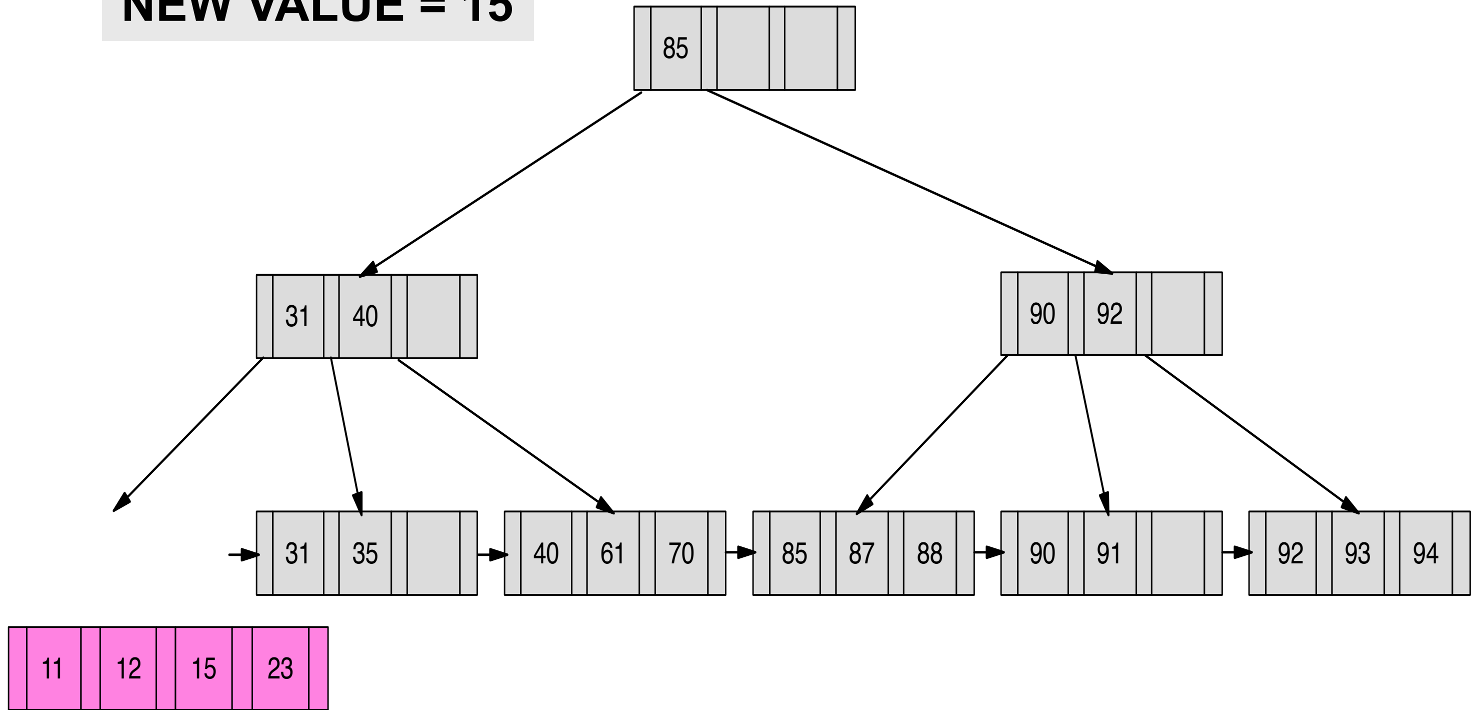
NEW VALUE = 15

B tree of order n=4



B + Tree

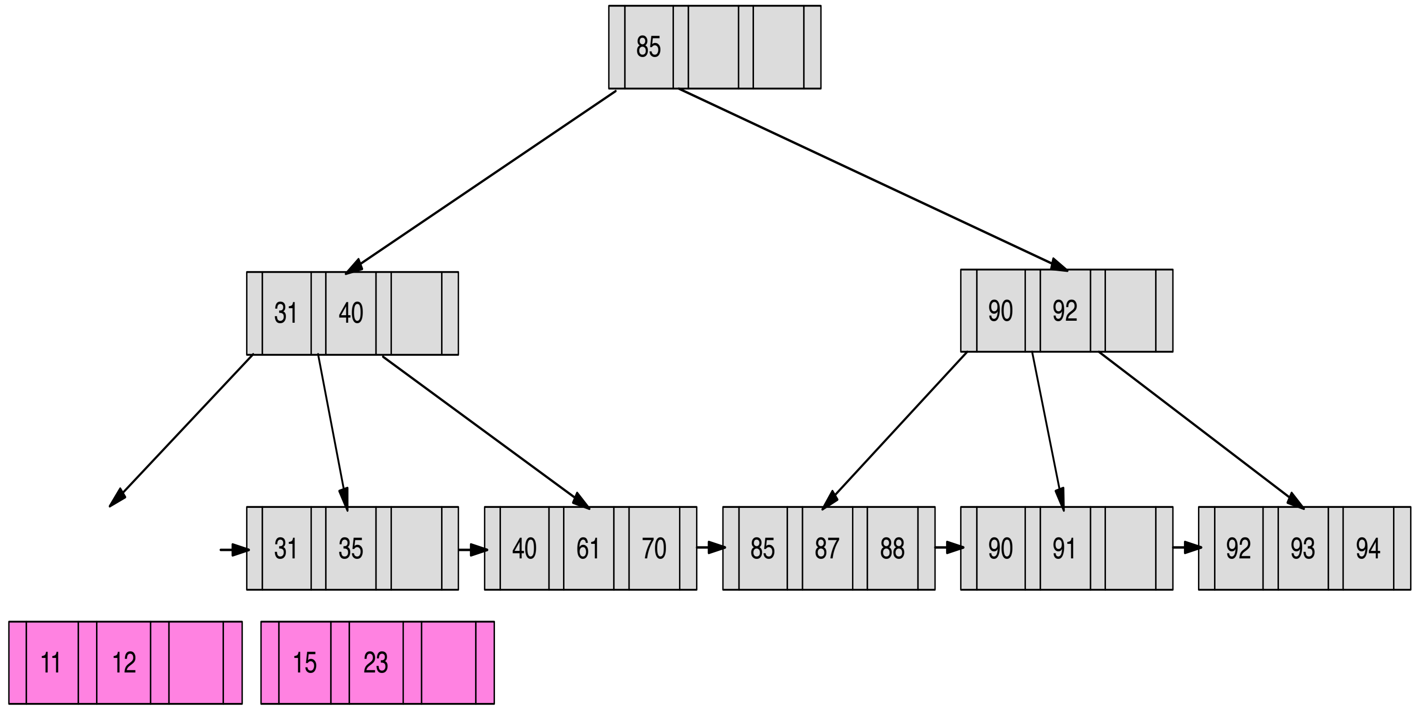
NEW VALUE = 15



The new value 15 is inserted in the correct leaf, which in this case will be too large

B + Tree

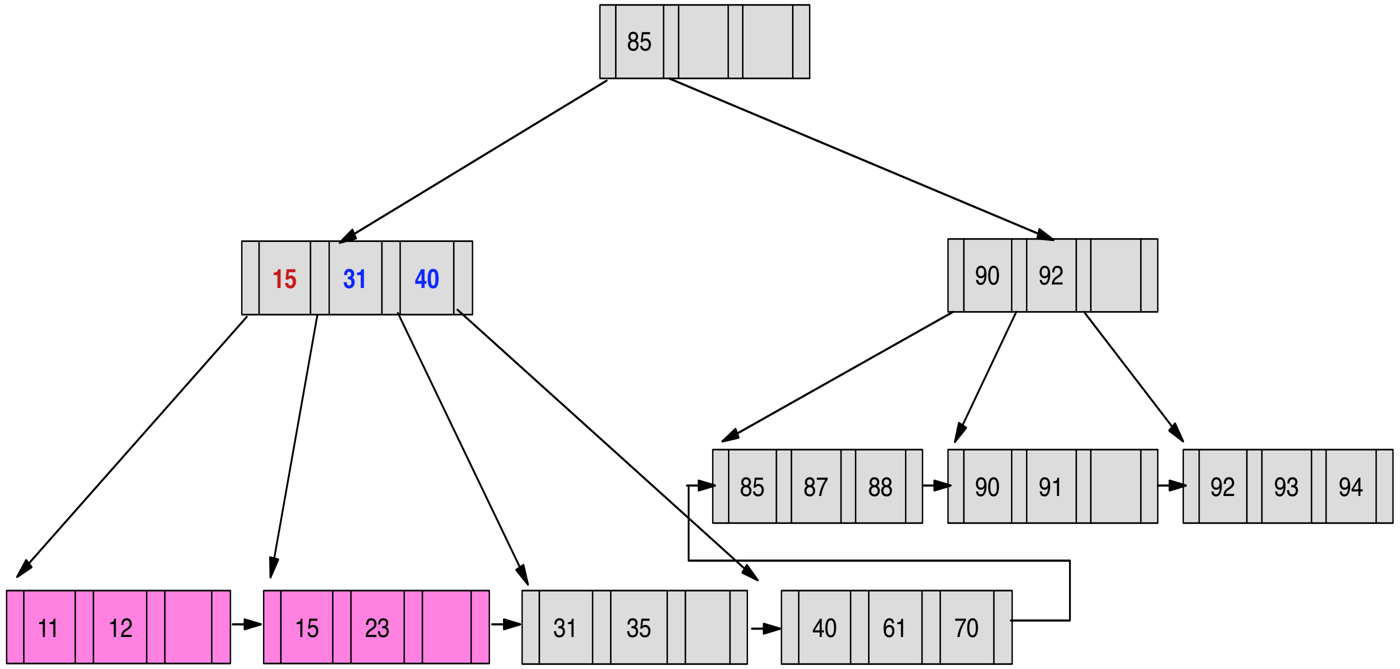
NEW VALUE = 15



the leaf is then broken in two

B + Tree

NEW VALUE = 15



The first value of the right leaf, which happens to be 15, is entered in the parent, who happens to have 1 new child

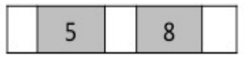
B + Tree

B + Tree

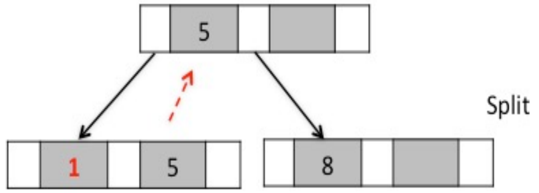
Example

5,8,1,7,3,12
Tree order = 3

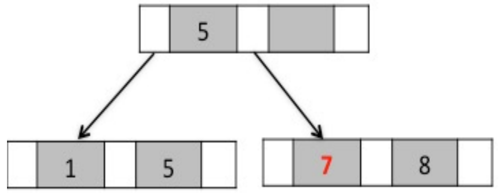
1 – We start by inserting 5 and 8



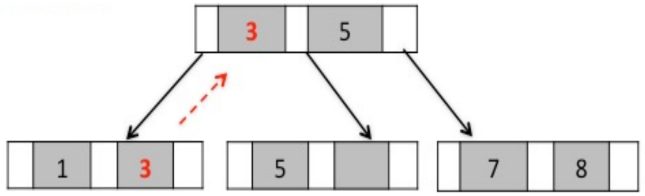
2 – Let's insert the 1 – there is no previous node. Because it's full - > Then we have to split the node and create a new level



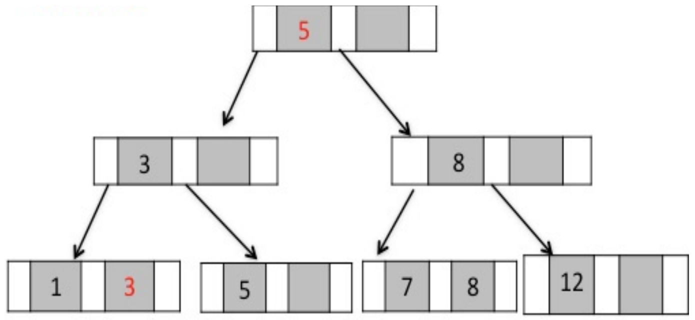
3 – Let's insert the 7 – we have space at the level of the 8



4. Let's insert the 3 – it should be between the 1 and the 5. - > Separation and propagation



5. Let's insert the 12. Overflow -> separation and propoagation and new level

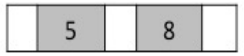


B + Tree

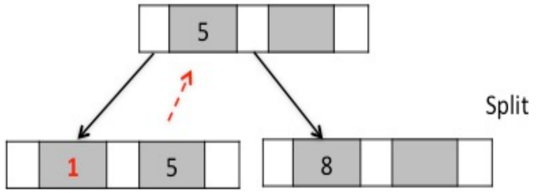
Example

5,8,1,7,3,12
Tree order = 3

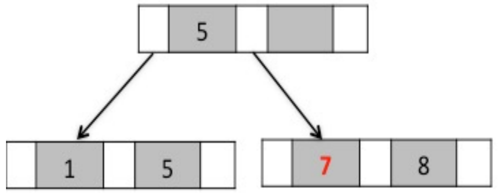
1 – We start by inserting 5 and 8



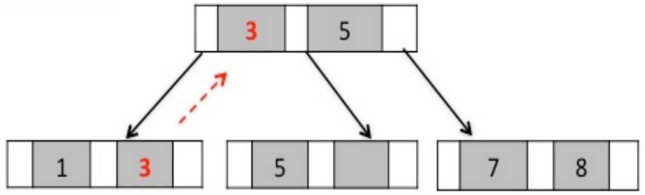
2 – Let's insert the 1 – there is no previous node. Because it's full - > Then we have to split the node and create a new level



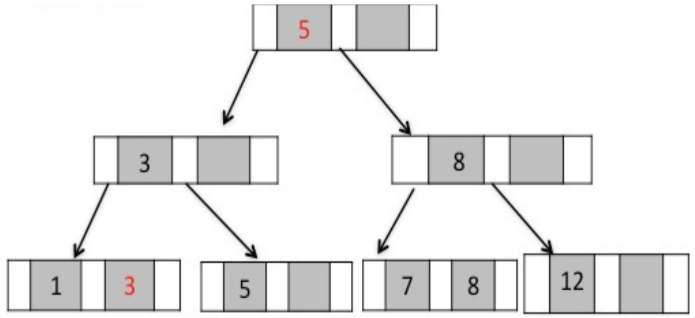
3 – Let's insert the 7 – we have space at the level of the 9



4. Let's insert the 3 – it should be between the 1 and the 5. - > Separation and propagation



5. Let's insert the 12. Overflow -> separation and propoagation and new level



B $^+$ Tree – Search/Update

Search

- ➢ Find all records with k search key

1. start at the root
   a) search in the root for the smallest value*Ki* such that *Ki* >= *k*

      IF it exists

      THEN follow Pi to the descending node

      ELSE follow Pn

2. Repeat the previous step until you reach a leaf

   As soon as we get to a leaf

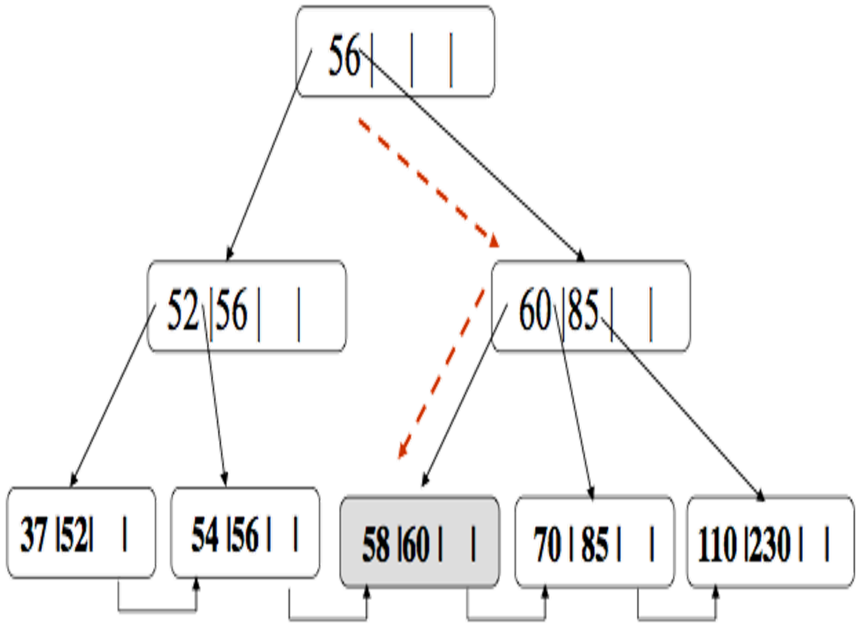
   a) search on the sheet for the value ki such that ki=K

      IF EXISTS

      THEN *is follow Pi to the register or container*

      ELSE It is concluded that the register does not exist

Search for the value 60



B $^+$ Tree

Update

➢ **Modifications to B+ trees follow these steps:**

1. Search for the desired value in the tree, from the root to the respective leaf (top-down)
2. Make the changes you want at the leaf level
3. Pass on the effects in a bottom-up way, that is, after changing nodes at a level, check whether the ascending node is coherent, or propagate the effects recursively (to the root, if necessary).

➢ **General properties that must be maintained:**

- to the left of a value is a subtree with ever-smaller values
- To the right of a value is a subtree with values greater than or equal
- Each node must always have at least (n-1)/2 values (or n/2 pointers) filled in, otherwise it is necessary to redistribute values (and pointers) to neighboring nodes

B $^+$ Tree - Delete

Search for the record and remove it from the file

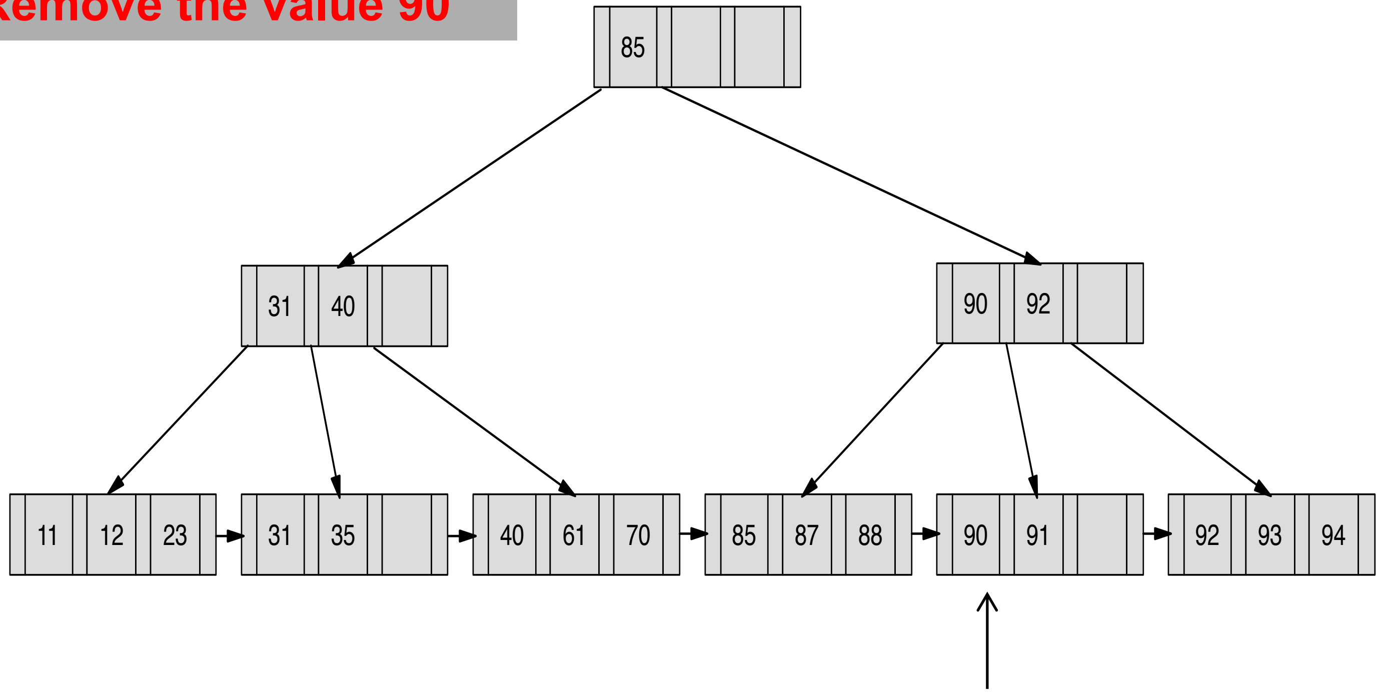
- **If the pointer or container is empty,** remove the entry (value, pointer) from the leaf
- **if the leaf fell below the minimum of [(n-1)/2] values**

  If possible, **pass these values to the left node, remove the leaf and remove the respective entry in the ascending node**
- **If when modifying the ascending node, it gets a few pointers below [n/2]** repeat the procedure recursively
- These effects "go up" up the tree until they reach a node that has at least [n/2] pointers
- **If the root is left with only one descendant after remove,** then the descendant becomes the new root removal, then the descendant becomes the new root

B $^+$ Tree - Delete

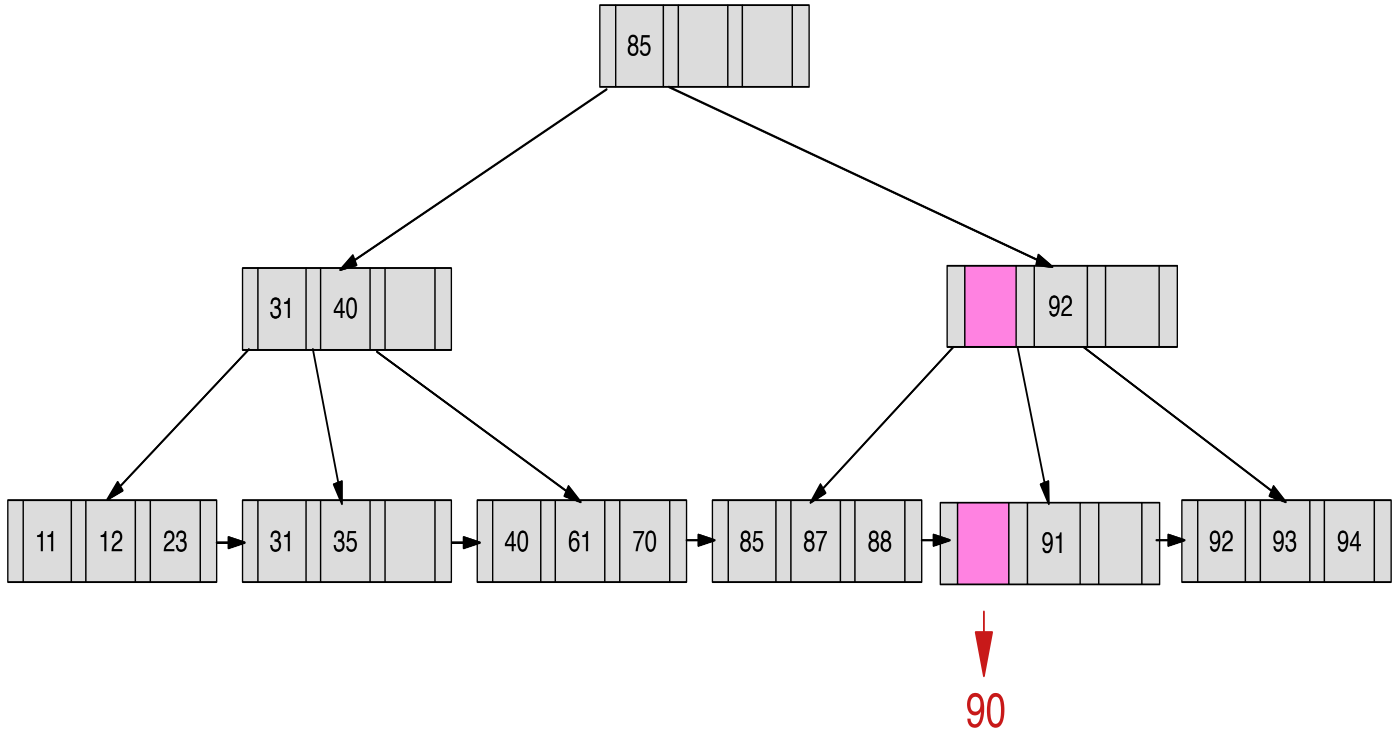
B $^+$ Tree - Delete

Remove the value 90



B $^+$ Tree - Delete

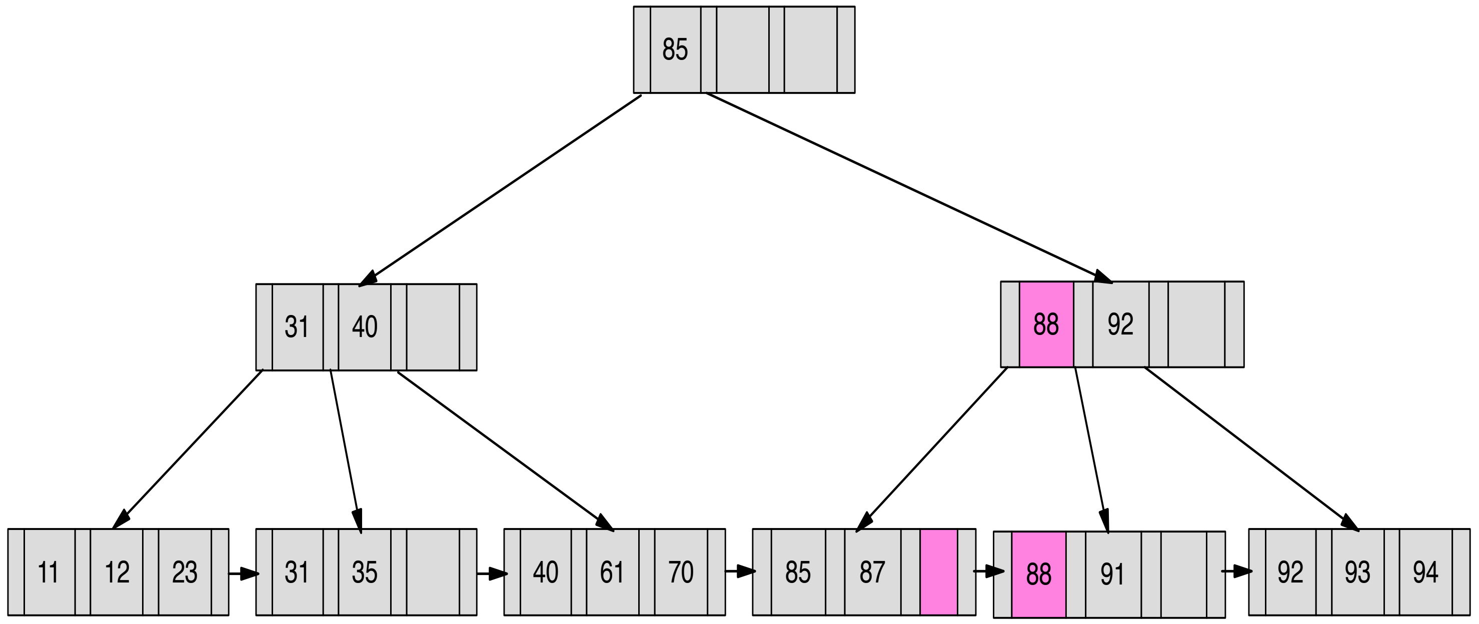
Remove the value 90



The value 90 is removed from the correct leaf, which in this case will be too small

B + Tree - Delete

Remove the value 90



The last value of the left leaf, which in this case is 88, is inserted on right leaf, which happens to be the correct size

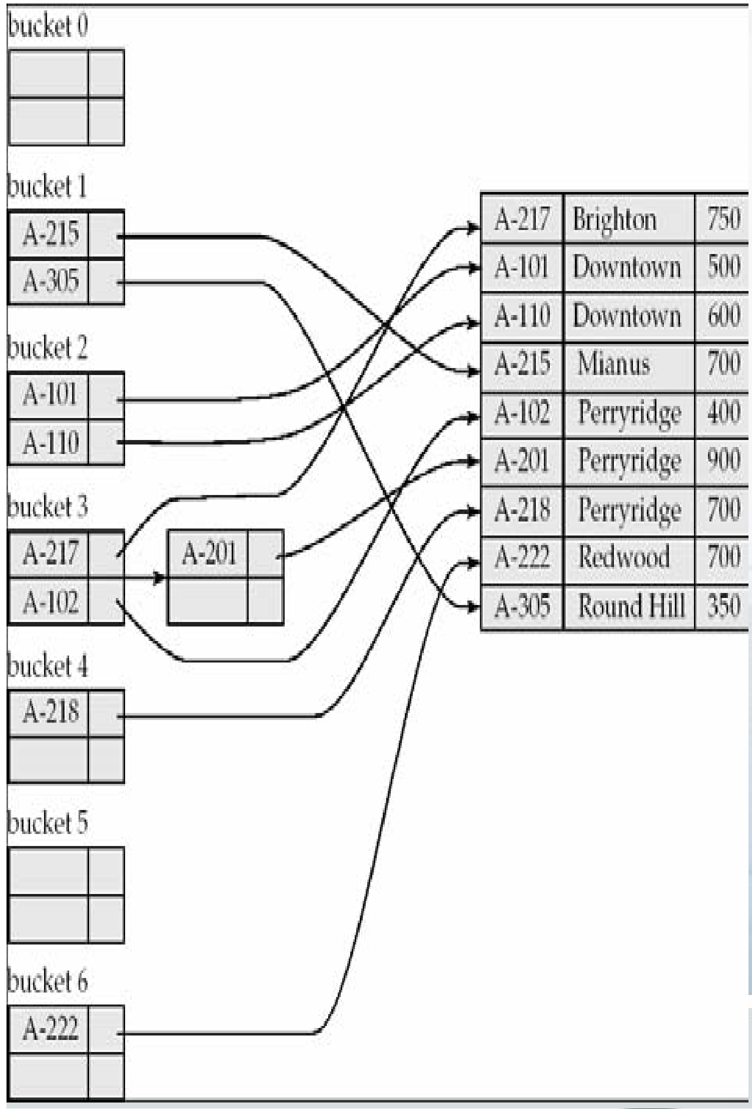
Hash Index

Definition

- secondary memory divided into B buckets (typically, B=2n)
  - hash function h maps the keys to integers in the range [0,B]
  - buckets can get "full" (bucket overflow):

  Example:

    - Use the first bucket with empty space (linear probing), or create a chained list of buckets
    - Hashing can be used for file organization and index creation
- hash-type indexes are typically used as secondary indexes



Hash Index

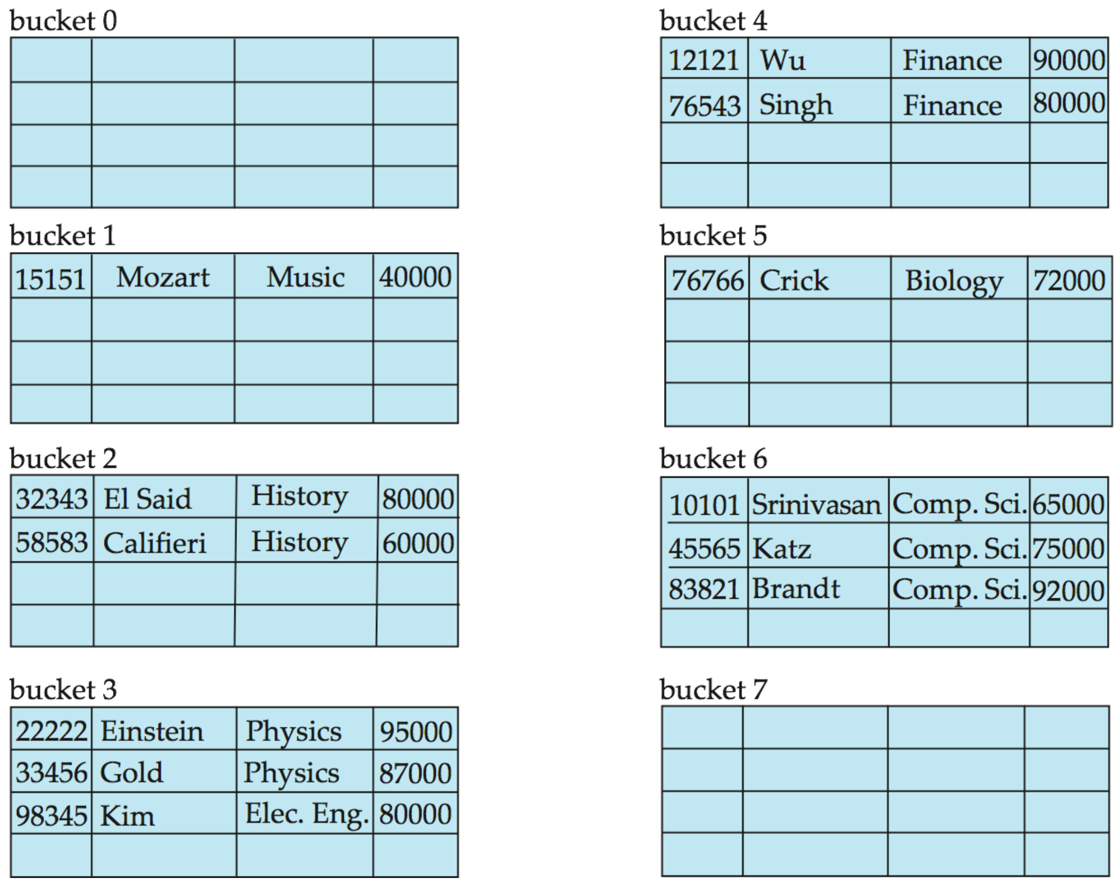
- The central idea of the Hash is to use a function (Hashing Function), applied over part of the information (key), to return the index where the information should or should be stored.

h(key) mod B

- The hash function is used to find records for access, insertion, as well as delete operations.
- Records with different search-key values may be mapped to the same bucket; thus, entire bucket must be searched sequentially to locate a record.

Example of Hash File Organization

- There are 10 buckets,
- The binary representation of the ***i***th character is assumed to be the integer ***i.***
- The hash function returns the sum of the binary representations of the characters **modulo 10**
  - E.g. h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3



bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key

Hash Index

- Hashing can be used **not only for file organization**, but also for **index-structure creation.**
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indexes **are always secondary indices**
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Hash Index – Static

- A static hash index organizes the search keys, with their associated record pointers, within the hash file structure

- **Problems:**
  - The hash function maps search key values to a fixed number of containers If the Database grows, performance suffers from the excess of extra containers Even if the size can be predicted, there will initially be a waste of space If the DB decreases, the space will be wasted, this implies reorganization. Periodic reorganization is expensive

Solution: vary the number of containers dynamically

Hash Index – Static

The HASH function indicates the bucket of the intended input

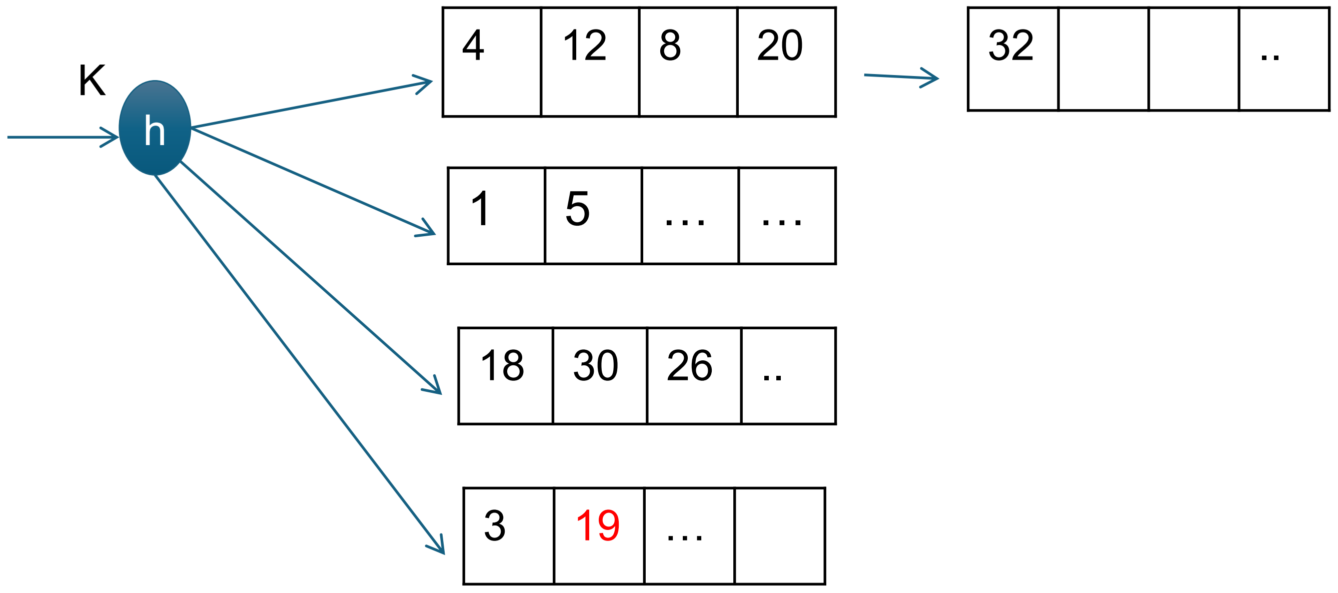
h(1) = 1 mod 4 = 1

H(32)= 32 mod 4= 0

Insertion

H(19)= 19 mod 4= 3

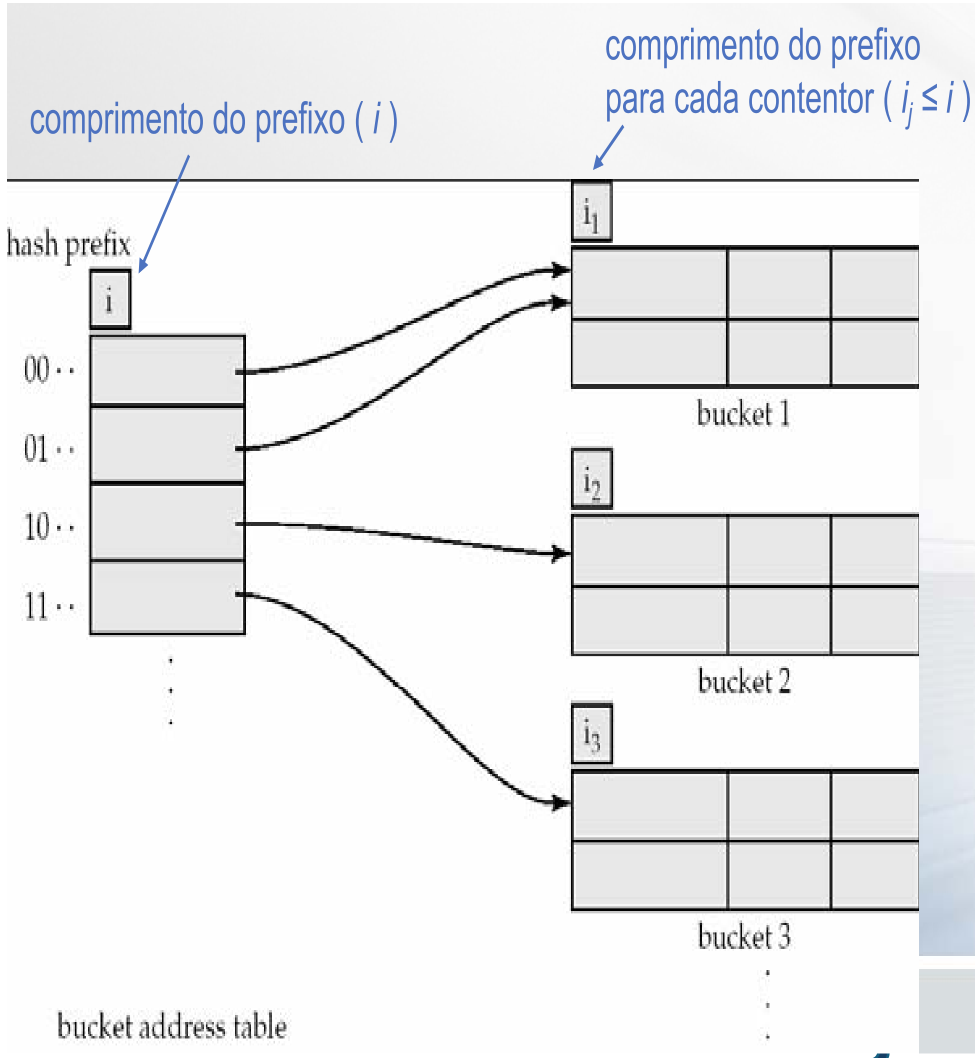
If it is full, a new page is created



Hash Index – Dynamic

- Allows you to modify the hash function dynamically

- Extensible hashing
  - The hash function generates values over a wide range – typically 32-bit integers
  - The idea is to use at each moment only a prefix to address a set of containers
    - Let i be the length of the prefix, with $0 \leq i \leq 32$
    - Maximum number of addressable containers : 2^{i}
    - the value of i varies depending on the size of the DB



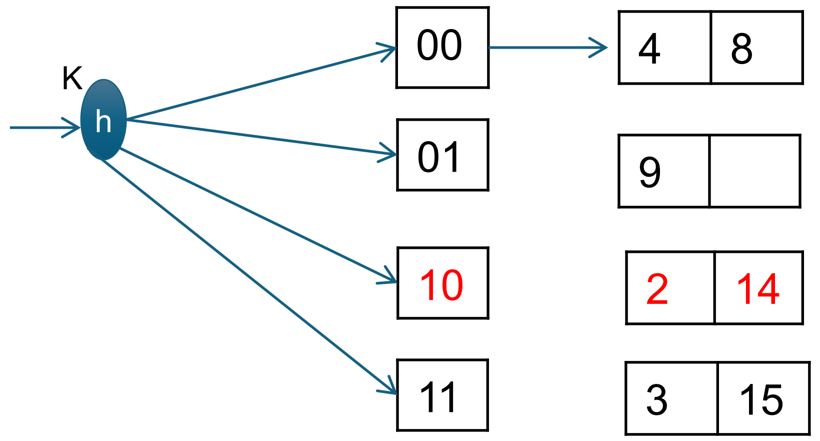
Hash Index – Dynamic

- It has a table of addresses;
- The HASH function indicates directory elements instead of buckets

  It only considers n bits of the result of n.

  h(14) = 14 mod 100 = 14

  1110 (considering 2-bit) is 10



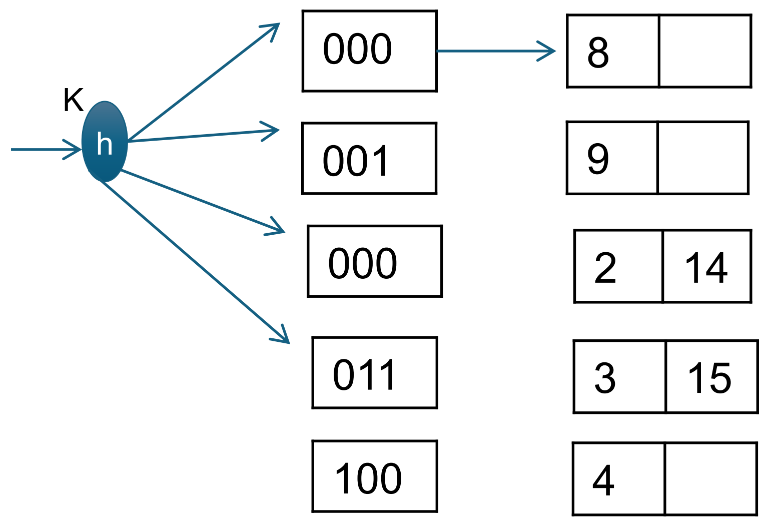
Hash Index – Dynamic

Insert

h(12) = 12 mod 100 = 12 -> 1100 , consider +1

It's full, then

Increase local depth from 2 to 3



Search

h(4)= 4 mod 100= 4 -> 0100

Indexes

Bitmap Index

Bitmap Index

- **Bitmap indices are a special type of index designed for efficient querying on multiple keys**
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Index

- In its simplest form a bitmap index on **an attribute has a bitmap for each value of the attribute**
  - Bitmap has as many bits as records
  - In a bitmap for value v,
    - the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3
	……	….	…..
	……	….	…..

Bitmaps for *gender*

m	10010
f	01101
….	………

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Index

- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)

- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap

E.g. 100110 AND 110011 = 100010

100110 OR 110011 = 110111

NOT 100110 = 011001

Males with income level L1: 10010 AND 10100 = 10000

- Can then retrieve required tuples.
- Counting number of matching tuples is even faster

How to Create Indexes

Create an index

create index <index-name> **on** <relation-name> (<attribute-list>)

E.g.: **create index** *b-index* **on** *branch(branch_name)*

The CREATE INDEX statement can include various parameters, e.g.,

- *space allocated for the index,*
- *computation of statistics for optimizer*
- *. . .*

CREATE INDEX sailor name idx ON sailor(name)

STORAGE(INITIAL 40K

NEXT 20K)

COMPUTE STATISTICS;

. . . for all this you need privileges

Drop an index

drop index <index-name>

Indexes

Oracle DB

Oracle Index Types

Examples:

- ***CREATE INDEX sailor name idx ON sailor(name);***
- ***DROP INDEX sailor name idx;***
- ***CREATE INDEX sailor name and age idx ON sailor(name, age);***

➢ Oracle create **a non-clustered B+-tree** index:

1. **Primary Key**
    a. If key consists of more than one attributes, the index is a composite or concatenated index
    b. A composite index is stored lexicographically according to order of the attributes .

2. Any **attribute** with **a Unique constraint**
    a. the index is needed to maintain this constraint

➢ **Principles:** An index has a name; is created for a sequence of attributes over a table and can be dropped

The standard index is the B$^+$ tree

Oracle Index Types

- Create a **cluster** (= shared storage for tables with common attributes)

```
CREATE CLUSTER sailor _id _cluster (id int);
```

A cluster is made for a fixed number of typed attributes

- Create a clustered index

```
CREATE INDEX sailor _id idx ON CLUSTER sailor _id_cluster;
```

There can only be one index for one cluster

- Create table for the cluster

```
CREATE TABLE sailor (id INT NOT NULL,
      name VARCHAR(20),
     ranking INT,
     age INT CLUSTER sailor id cluster (id)
    ) ;
```

Oracle Index Types

HASH is one of four index types

CREATE INDEX sailor id idx ON sailor USING HASH(id) ;

- In general, hash indexes need more space than B$^+$ trees and their construction takes much longer than the one of of a B+-tree

- Hash Clusters realize the concept of a hash file

. . . and a bit more than that

```
CREATE CLUSTER sailor id cluster (id int)
                    SIZE 2K
                    HASH IS (id)
                    HASHKEYS 10000;
```

Oracle Index Types

Function Based Indexes

- An index can be based on the values of a function over one or more attributes

  CREATE INDEX emp income idx ON emp(sal + com);

- The query optimizer can match this to a condition containing the function:

  SELECT e.ename

  FROM emp e

  WHERE **e.sal + e.com** BETWEEN 1000 AND 1500

Oracle Index Types

Bitmap indexes

- for each value of a domain, there is a bitmap identifying the rid's of satisfying tuples.

  E.g., bitmaps for all tuples with Approved =Yes and Approved = No

  *...**good for small domains tables that don't change too much***

- Bitmap indexes are **most effective** for **queries** that **contain multiple conditions** in the **WHERE clause**

- A bitmap join uses a bitmap for key values and a mapping function that converts each bit position to a rowid.

- ***Bitmaps can efficiently merge indexes that correspond to several*** **conditions in a WHERE clause**, using Boolean operations to resolve AND and OR conditions.

When to use Indexes

- **Function-based indexes**
  - An index range scan has a fast response time when the WHERE clause returns **fewer than 15% of the rows of a large table.**
  - **reduces computation for the database**.
    - If we have a query that consists of an expression and use this query many times, the database has to calculate the expression each time you execute the query.
  - helps you perform more flexible sorts.
- **Bitmap indexes**
  - They are usually easier to destroy and recreate than to maintain.
  - Maintaining a bitmap index takes a lot of resources, therefore, bitmap indexes **are only good for read-only tables or tables that have infrequently updates.**
  - We should **use** the bitmap index **for the columns that have low cardinality.**

Indexes Oracle

- **INDEX UNIQUE SCAN** - performs the tree traversal only. The Oracle database uses this operation if a **unique constraint ensures that the search** criteria **will match no more than one entry.**

- **INDEX RANGE SCAN-** performs the tree traversal and follows the leaf node chain to find all matching entries. This is the fallback operation if multiple entries could possibly match the search criteria.

  The important point - an INDEX RANGE SCAN can potentially read a large part of an index. If there is one more table access for each row, the query can become slow even when using an index.

- **TABLE ACCESS BY INDEX ROWID-** retrieves the row from the table. This operation is (often) performed for every matched record **from a preceding index scan operation**.

References

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

Oracle Database Documentation
https://docs.oracle.com/cd/B10500_01/appdev.920/a96590/adg06idx.htm