

DBMS

Unit- V

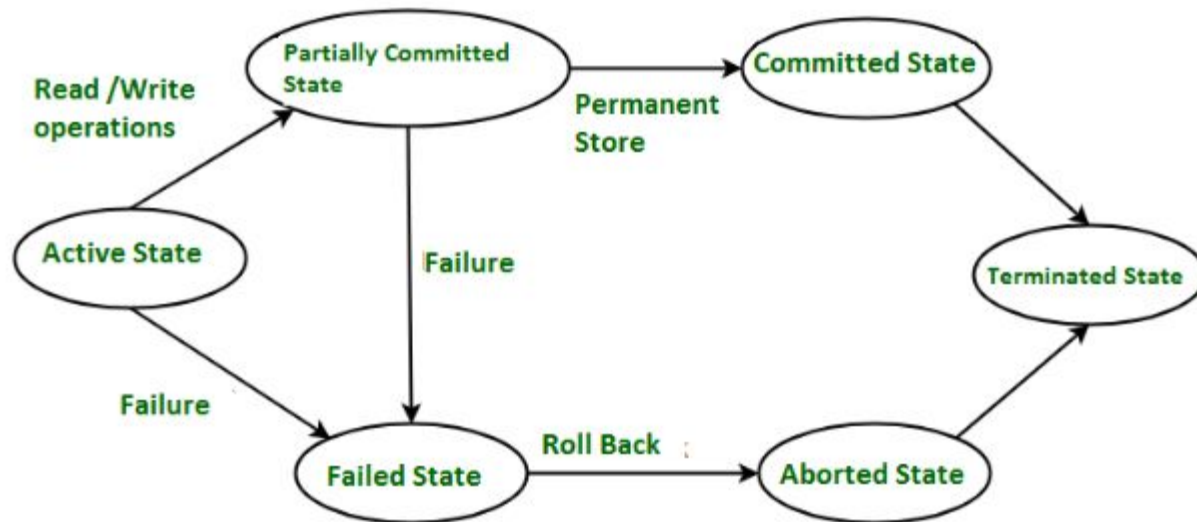
UNIT - V

- Transaction Concept: Transaction State, ACID properties, Concurrent Execution of transactions, Schedules, Serializability, Recoverability, Testing for Serializability, Lock based and timestamp-based concurrency protocols, Implementation of Isolation, Failure Classification, ARIES Recovery algorithm, Introduction to Indexing Techniques, B+ Trees, operations on B+ Trees, Hash Based Indexing.

Transaction in DBMS

In DBMS, a transaction is a set of logical operations performed to access and modify the contents of the database as per the user's request.

Transaction States in DBMS



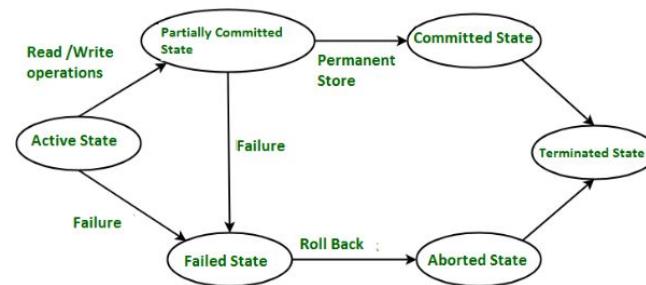
Transaction States in DBMS

Transaction States in DBMS

- **Active State –**

When the instructions of the transaction are running then the transaction is in active state.

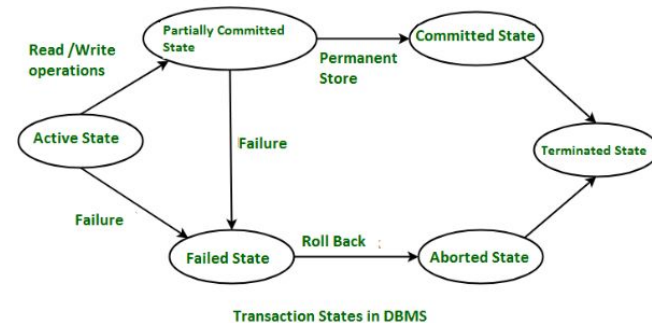
If all the ‘read and write’ operations are performed without any error then it goes to the “partially committed state”; if any instruction fails, it goes to the “failed state”.



Transaction States in DBMS

Transaction States in DBMS

- **Partially Committed –**



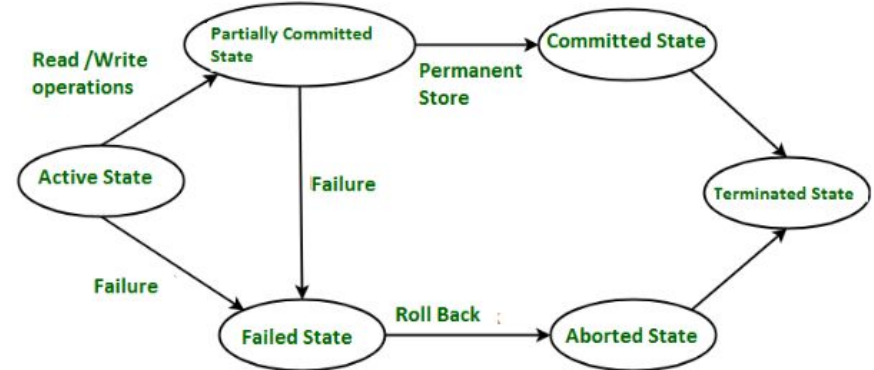
After completion of all the read and write operation the changes are made in main memory or local buffer.

If the changes are made permanent on the Database then the state will change to “committed state” and in case of failure it will go to the “failed state”.

Transaction States in DBMS

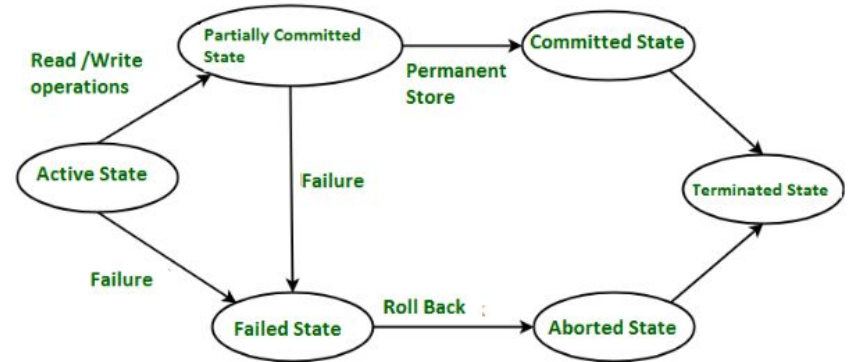
- **Failed State**

When any instruction of the transaction fails, it goes to the “failed state” or if failure occurs in making a permanent change of data on Data Base.



Transaction States in DBMS

Transaction States in DBMS



Transaction States in DBMS

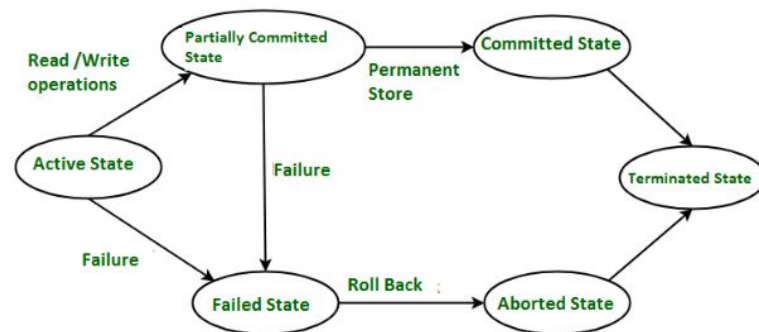
- **Aborted State**

After having any type of failure the transaction goes from “failed state” to “aborted state” and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.

Transaction States in DBMS

- **Committed State**

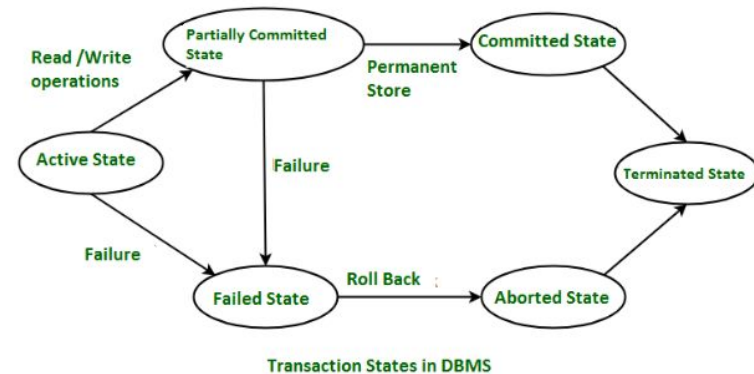
It is the state when the changes are made permanent on the Data Base and the transaction is complete and therefore terminated in the “terminated state”.



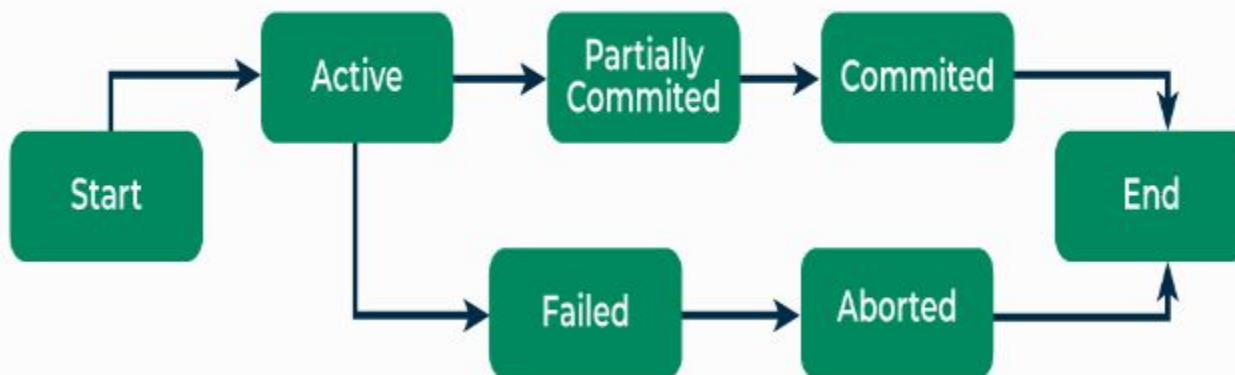
Transaction States in DBMS

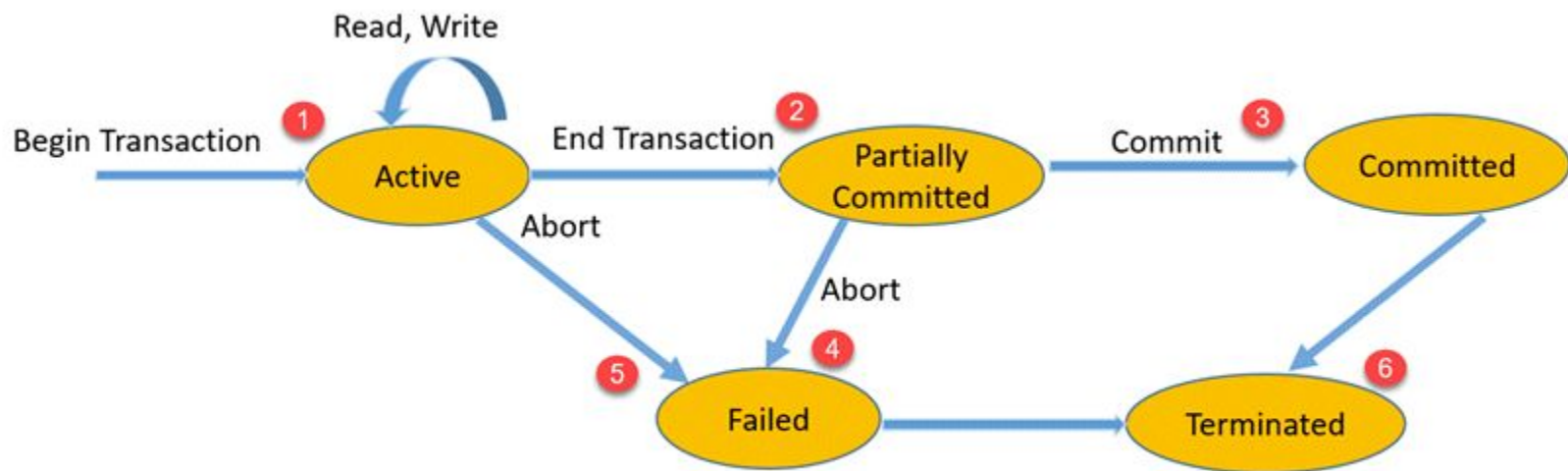
Transaction States in DBMS

- **Terminated State**



If there isn't any roll-back or the transaction comes from the "committed state", then the system is consistent and ready for new transaction and the old transaction is terminated.





State Transition Diagram for a Database Transaction

States of Transactions

State	Transaction types
Active State	A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed.
Partially Committed	A transaction goes into the partially committed state after the end of a transaction.
Committed State	When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently.
Failed State	A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state.
Terminated State	State of transaction reaches terminated state when certain transactions which are leaving the system can't be restarted.

Transaction Control in DBMS

- The transaction is a single logical unit that accesses and modifies the contents of the database.
- Transactions access using read and write operations.
- Transaction is a single operation of processing that can have many operations.
- Transaction is needed when more than one user wants to access same database. Transaction has ACID properties.

ACID Properties in Transaction

- *ACID is an acronym used for the properties of transaction in DBMS. It is used to denote the properties of transactions,*

*i.e. **Atomicity**, **Consistency**, **Isolation** and **Durability**.*

ACID Properties of Transaction:

ACID stands for

- Atomicity
- Consistency
- Isolation
- Durability

ACID Properties of Transaction:

- **Atomicity:**

All the operations in a transaction are considered to be atomic and as one unit.

If system fails or any read/write conflicts occur during transaction the system needs to revert back to its previous state.

Atomicity is maintained by the Transaction Management Component.

ACID Properties of Transaction:

- **Consistency:**

Every transaction should lead to database connection from one valid state to other valid state.

If system fails because of invalid data while doing an operation revert back the system to its previous state.

Consistency is maintained by the Application manager.

ACID Properties of Transaction:

Isolation:

If multiple transactions are executing on single database, each transaction should be isolated from other transaction.

If multiple transactions are performed on single database, operation from any transaction should not interfere with operation in other transaction.

Isolation is maintained by the concurrency control manager.

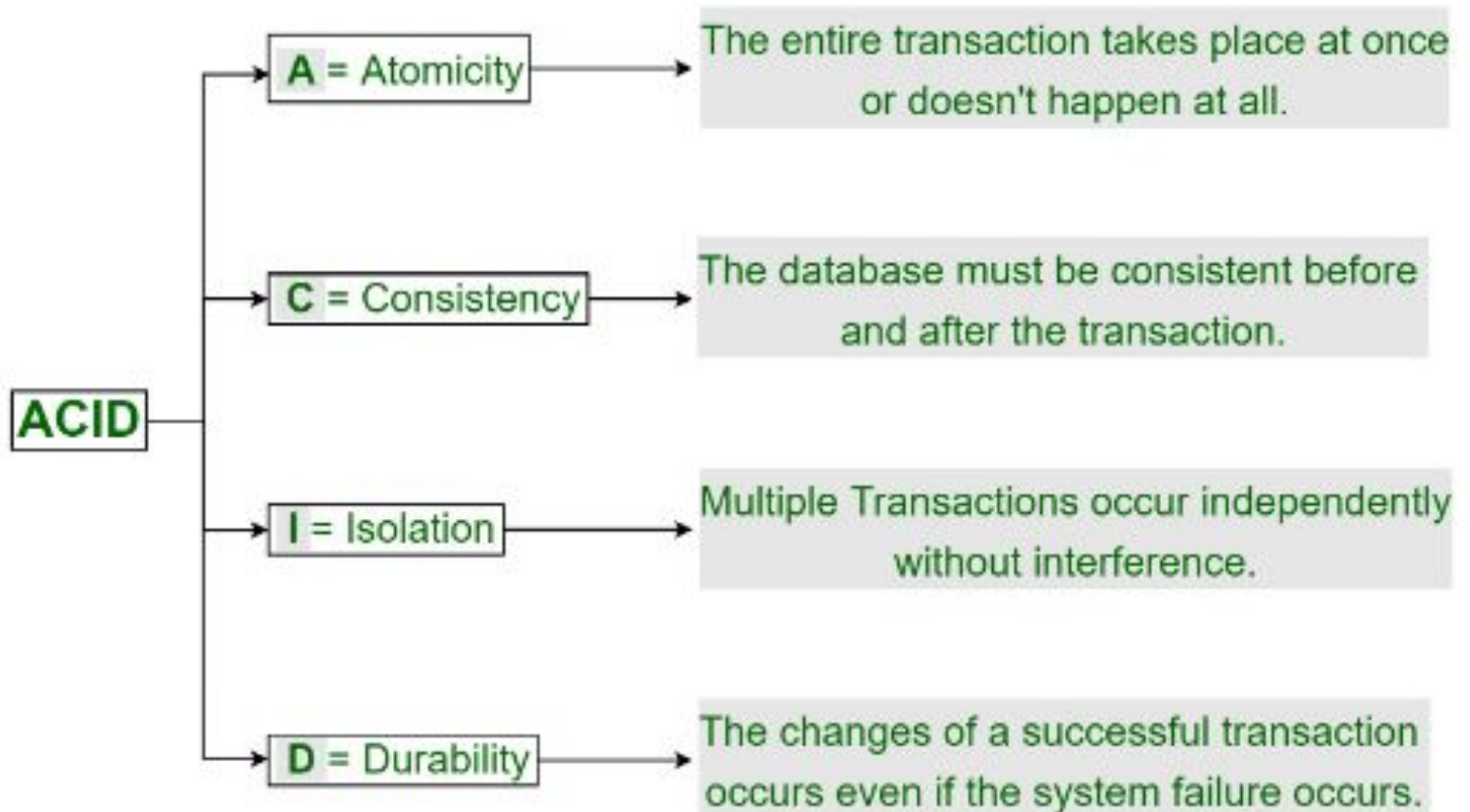
ACID Properties of Transaction:

- **Durability:**

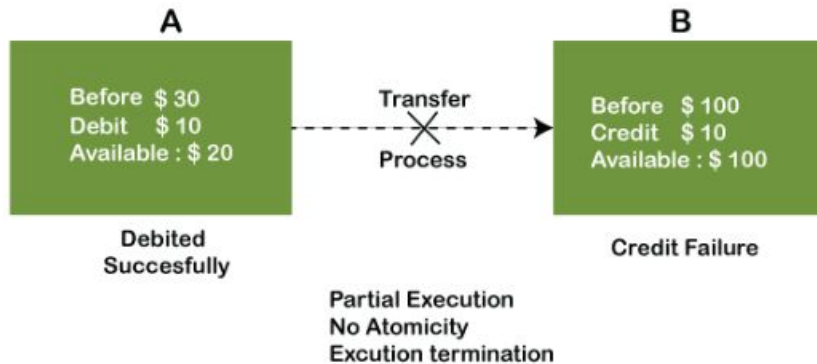
Durability means the changes made during the transactions should exist after completion of transaction.

Changes must be permanent and must not be lost due to any database failure. It is maintained by the recovery manager.

ACID Properties of Transaction



Atomicity



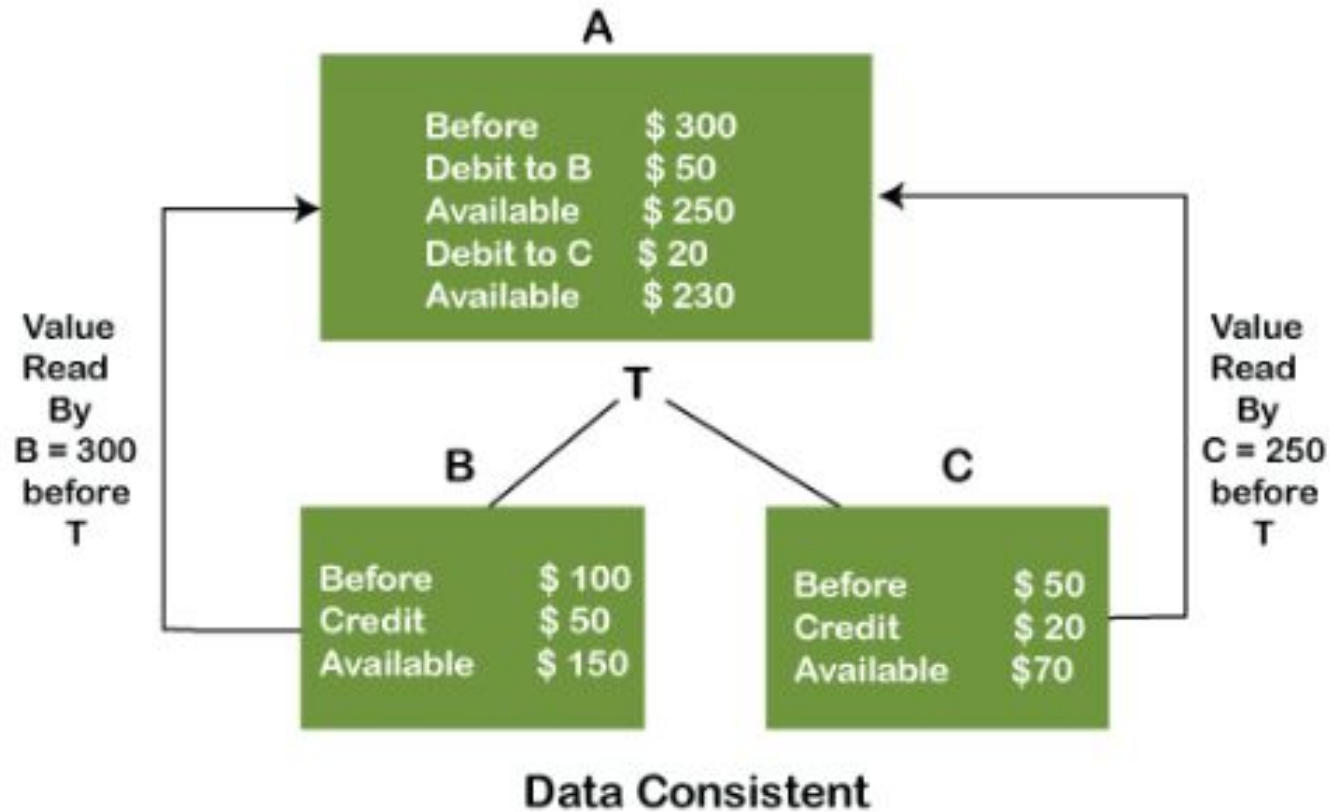
In the above diagram, it can be seen that after crediting \$10, the amount is still \$100 in account B. So, it is not an atomic transaction.

The below image shows that both debit and credit operations are done successfully. Thus the transaction is atomic.



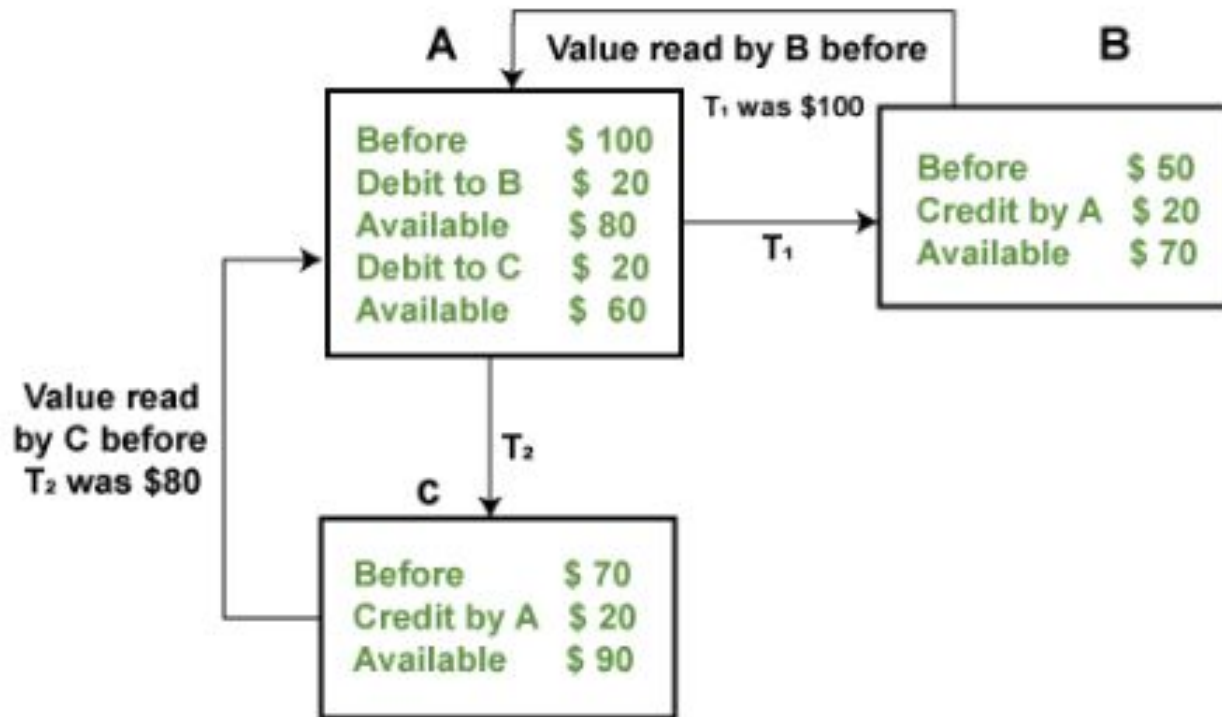
Consistency

The database must be consistent before and after the transaction



Isolation

Multiple transactions occurs independently without any interference

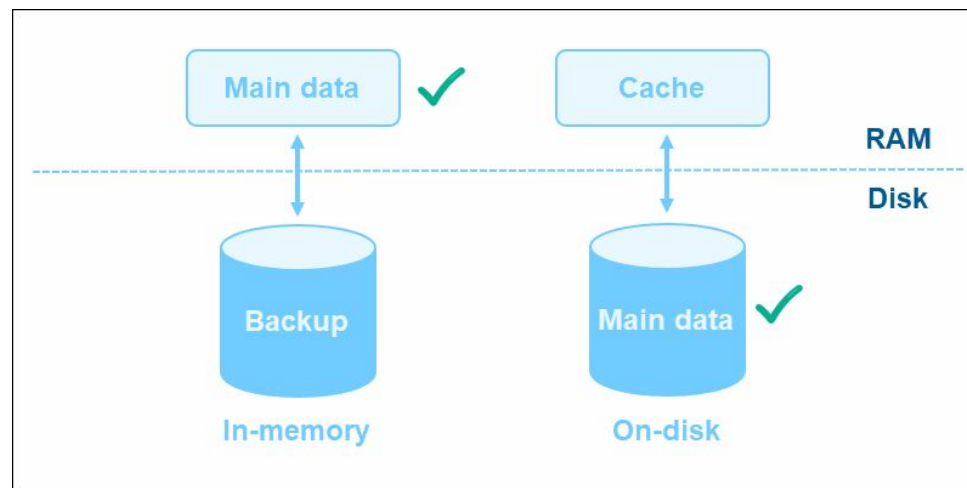


Isolation - Independent execution of T₁ & T₂ by A

Durability

The Changes of a successful transaction occurs even if the system failure occurs

- Durability ensures the permanent.
- Durability ensures that the data after the successful execution of the operation becomes permanent in the database.
- The durability of the data should be perfect that even if the system fails or leads to a crash, the database still survives.



Transaction

- **Example:** Consider the following example of transaction operations to be performed to withdraw cash from an ATM.

Steps for ATM Transaction

1. Transaction Start.
2. Insert your ATM card.
3. Select a language for your transaction.
4. Select the Savings Account option.
5. Enter the amount you want to withdraw.
6. Enter your secret pin.
7. Wait for some time for processing.
8. Collect your Cash.
9. Transaction Completed.

Transaction

- **A transaction can include the following basic database access operation.**
- **Read/Access data (R):** Accessing the database item from disk (where the database stored data) to memory variable.
- **Write/Change data (W): Write** the data item from the memory variable to the disk.
- **Commit:** Commit is a transaction control language that is used to permanently save the changes done in a transaction

Transaction

- **Example:** Transfer of 50₹ from Account A to Account B. Initially A= 500₹, B= 800₹. This data is brought to RAM from Hard Disk.

R(A) – 500 // Accessed from RAM.

A = A-50 // Deducting 50₹ from A.

W(A)--450 // Updated in RAM.

R(B) -- 800 // Accessed from RAM.

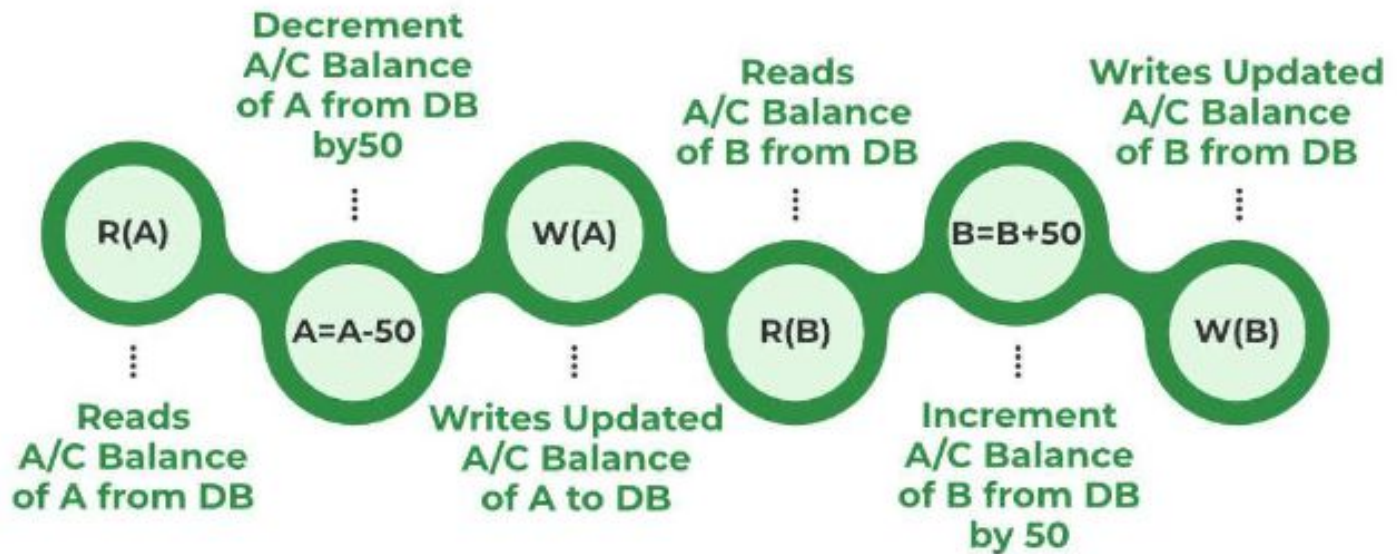
B=B+50 // 50₹ is added to B's Account.

W(B) --850 // Updated in RAM.

commit // The data in RAM is taken back to Hard Disk.

The updated value of Account A = 450₹ and Account B = 850₹.

Stages in Transaction



The updated value of Account A = 450₹ and Account B = 850₹.

Schedule of Transactions in DBMS

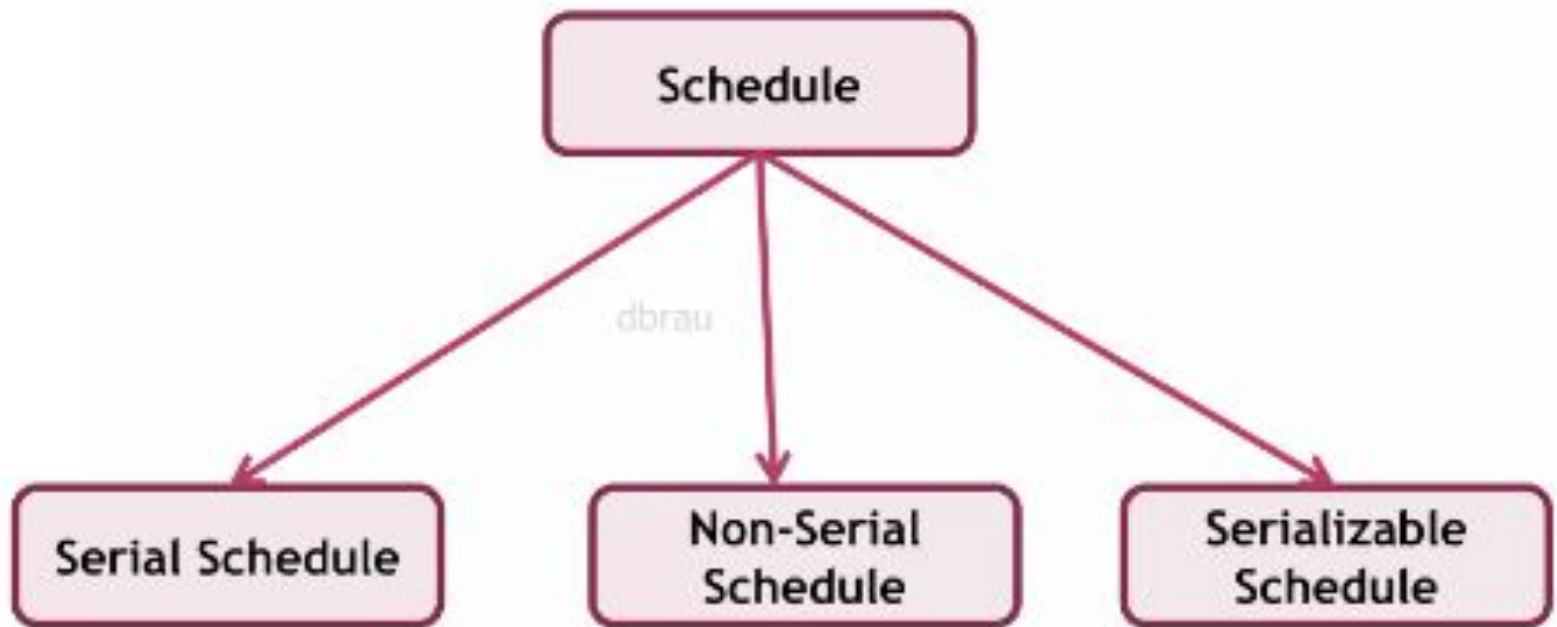
Multiple transaction requests are made at the same time, needed to decide their order of execution.

A transaction schedule can be defined as a chronological order of execution of multiple transactions.

There are broadly two types of transaction schedules

i) Serial Schedule

ii) Non-Serial Schedule



Serial Schedule

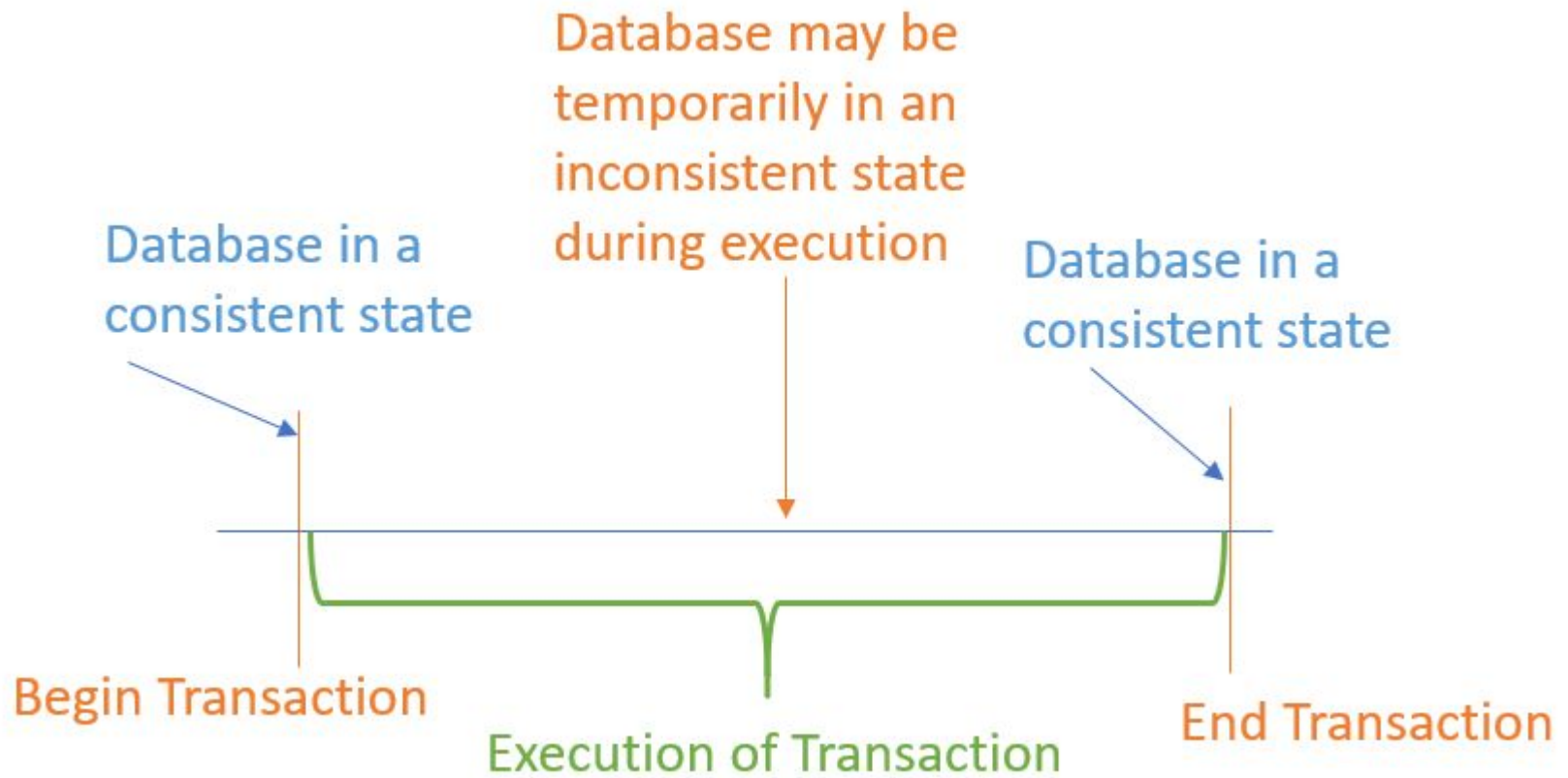
- Multiple transactions are to be executed, they are executed serially, i.e. at one time only one transaction is executed while others wait for the execution of the current transaction to be completed.
- This ensures consistency in the database as transactions do not execute simultaneously.
- It increases the waiting time of the transactions in the queue, which in turn lowers the throughput of the system, i.e. number of transactions executed per time.
- To improve the throughput of the system, another kind of schedule are used which has some more strict rules which help the database to remain consistent even when transactions execute simultaneously.

Non-Serial Schedule

- To reduce the waiting time of transactions in the waiting queue and improve the system efficiency, use non serial schedules which allow multiple transactions to start before a transaction is completely executed.
- This may sometimes result in inconsistency and errors in database operation.
- So, these errors are handled with specific algorithms to maintain the consistency of the database and improve CPU throughput as well.
- Non-Serial Schedules are also sometimes referred to as parallel schedules as transactions execute in parallel in this kind of schedules.

Serializable

- Serializability in DBMS is the property of a nonserial schedule that determines whether it would maintain the database consistency or not.
- The non serial schedule which ensures that the database would be consistent after the transactions are executed in the order determined by that schedule is said to be Serializable Schedules.
- The serial schedules always maintain database consistency as a transaction starts only when the execution of the other transaction has been completed under it. Thus, serial schedules are always serializable.
- A transaction is a series of operations, so various states occur in its completion journey.



Facts about Database Transactions

- A transaction is a program unit whose execution may or may not change the contents of a database.
- The transaction concept in DBMS is executed as a single unit.
- If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
- A successful transaction can change the database from one CONSISTENT STATE to another
- DBMS transactions must be atomic, consistent, isolated and durable
- If the database were in an inconsistent state before a transaction, it would remain in the inconsistent state after the transaction.

ACID Properties

- **ACID Properties** are used for maintaining the integrity of database during transaction processing. ACID in DBMS stands for **A**tomicity, **C**onsistency, **I**solation, and **D**urability.
- **Atomicity:** A transaction is a single unit of operation. You either execute it entirely or do not execute it at all. There cannot be partial execution.
- **Consistency:** Once the transaction is executed, it should move from one consistent state to another.
- **Isolation:** Transaction should be executed in isolation from other transactions (no Locks). During concurrent transaction execution, intermediate transaction results from simultaneously executed transactions should not be made available to each other. (Level 0,1,2,3)
- **Durability:** After successful completion of a transaction, the changes in the database should persist. Even in the case of system failures.

ACID Property in DBMS with example

Transaction 1: Begin $X=X-50$, $Y = Y+50$ END

Transaction 2: Begin $X= X+0.1*X$, $Y= Y+0.1*Y$ END

- Transaction 1 is transferring \$50 from account X to account Y.
- Transaction 2 is crediting each account with a 10% interest payment.
- If both transactions are submitted together, there is no guarantee that the Transaction 1 will execute before Transaction 2 or vice versa. Irrespective of the order, the result must be as if the transactions take place serially one after the other.

- ⊙ Model used for representing database modifications of a transaction:
- ⊙ **read(A,x):** assign value of database object A to variable x;
- ⊙ **write(x,A):** write value of variable x to database object A.

⊙ Example of a Transaction T

read(A,x)

x := x - 200

write(x,A)

read(B,y)

y := y + 100

write(y,B)

Transaction Schedule reflect
chronological order of operations

Concurrent Executions

- Multiple transactions are allowed to **run concurrently** in the system.
- Advantages are:
 - **Increased processor and disk utilization**, leading to better transaction *throughput*
 - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
 - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms **to achieve isolation**
 - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
 - Will study it after studying the notion of **correctness of concurrent executions**.

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Schedules

- **Schedule** – a sequences of instructions that specify the **chronological order** in which instructions of concurrent transactions are executed
 - A schedule for a set of transactions must **consist of all instructions of those transactions**
 - Must **preserve the order** in which the instructions appear **in each individual transaction**.
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
 - **By default** transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

- ⊙ **Goal:** Synchronization" of transactions; allowing concurrency (instead of insisting on a strict serial transaction execution, i.e., process complete T1, then T2, then T3 etc.) ; increase the throughput of the system, ; minimize response time for each transaction.

Time	Transaction T1	Transaction T2
1	read(A,x)	
2	x:=x+200	
3		read(A,y)
4		y:=y+100
5	write(x,A)	
6		write(y,A)
7		commit
8	commit	

- ⊙ The update performed by T1 gets lost; possible solution: T1
- ⊙ locks/unlocks database object A
- ⊙ ⇒ T2 cannot read A while A is modified by T1

SERIALIZABILITY :

- ⊙ DBMS must control concurrent execution of transactions to ensure read consistency, i.e., to avoid dirty reads etc.
- ⊙ A (possibly concurrent) schedule S is serializable if it is equivalent to a serial schedule S' , i.e., S has the same result database state as S' .

- Conflicts between operations of two transactions:

Ti	Tj	Ti	Tj
read(A,x)		read(A,x)	
	read(A,y)		write(y,A)
(order does not matter)		(order matters)	
Ti	Tj	Ti	Tj
write(x,A)		write(x,A)	
	read(A,y)		write(y,A)
(order matters)		(order matters)	

- A schedule S is serializable with regard to the above conflicts if
- S can be transformed into a serial schedule S' by a series of swaps
- of non-conflicting operations.

- ⊙ Checks for serializability are based on precedence graph that describes dependencies among concurrent transactions; if the graph has no cycle, then the transactions are serializable.
- ⊙ they can be executed concurrently without affecting each others transaction result.

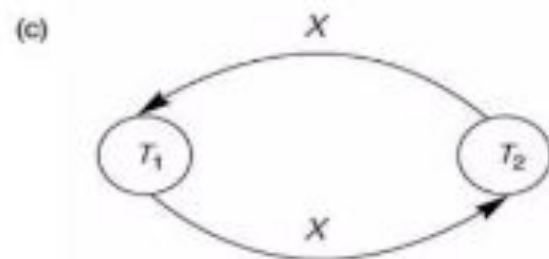
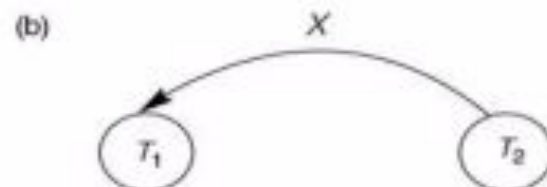
CHARACTERIZING SCHEDULES BASED ON SERIALIZABILITY

Testing for conflict serializability: Algorithm

- Looks at only read_Item (X) and write_Item (X) operations.
- Constructs a precedence graph (serialization graph) - a graph with directed edges.
- An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j .
- The schedule is serializable if and only if the precedence graph has no cycles.

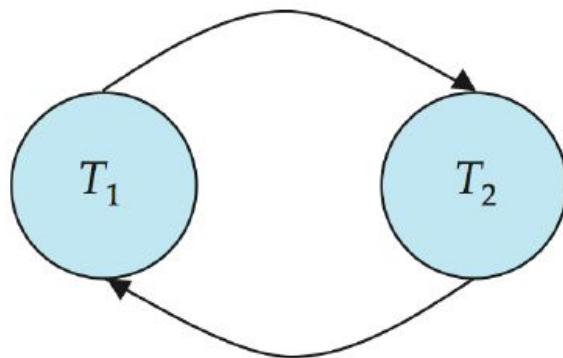
⊗ **FIGURE 17.7 Constructing the precedence graphs for schedules A and D from Figure 17.5 to test for conflict serializability.**

- (a) Precedence graph for serial schedule A.
- (b) Precedence graph for serial schedule B.
- (c) Precedence graph for schedule C (not serializable).
- (d) Precedence graph for schedule D (serializable, equivalent to schedule A).



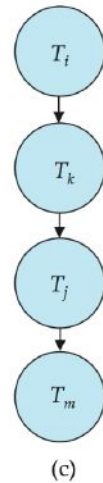
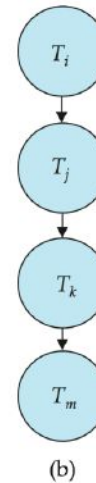
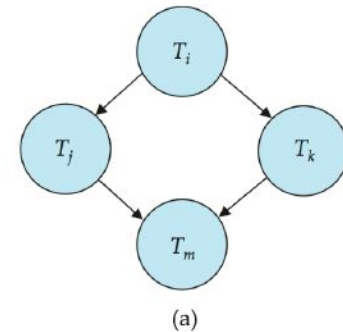
Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, \dots, T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example**



Testing for Conflict Serializability

- A schedule is conflict serializable **if and only if** its precedence graph is acyclic.
- **Cycle-detection algorithms** exist which take order n^2 time, where n is the number of vertices in the graph.
 - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a **topological sorting** of the graph.
 - That is, a linear order consistent with the partial order of the graph.
 - For example, a serializability order for the schedule (a) would be one of either (b) or (c)



1. SERIAL SCHEDULE:

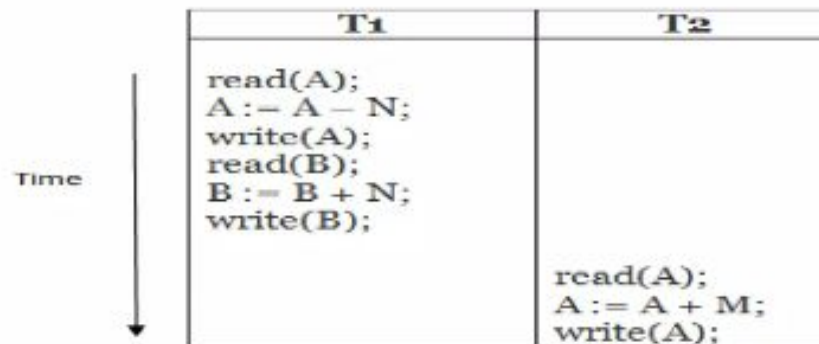
- The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction.
- A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

EXAMPLE:

- If there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible results:

SERIAL SCHEDULE EXAMPLE 1:

- Execute all the operations of T1 which was followed by all the operations of T2.

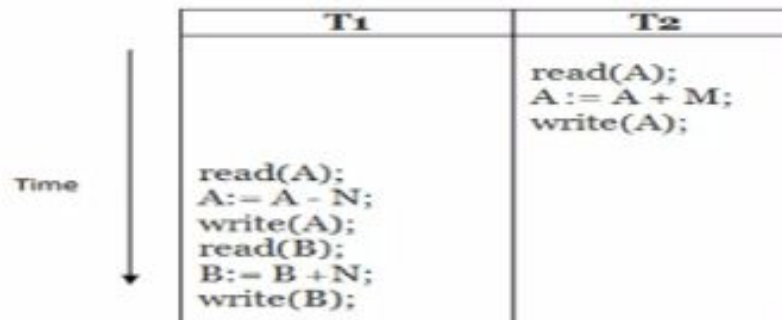


Schedule A

- Here, Schedule A shows the serial schedule where T1 followed by T2.

SERIAL SCHEDULE EXAMPLE 2:

- Execute all the operations of T2 which was followed by all the operations of T1.



Schedule B

- Here, Schedule A shows the serial schedule where T2 followed by T1.

2. NON-SERIAL SCHEDULE:

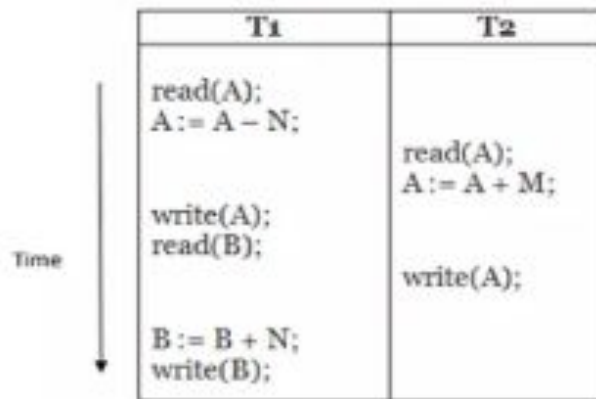
- ◉ If interleaving of operations is allowed, then there will be a non-serial schedule.
- ◉ There are many possible orders in which the system can execute the individual operations of the transactions.

EXAMPLE:

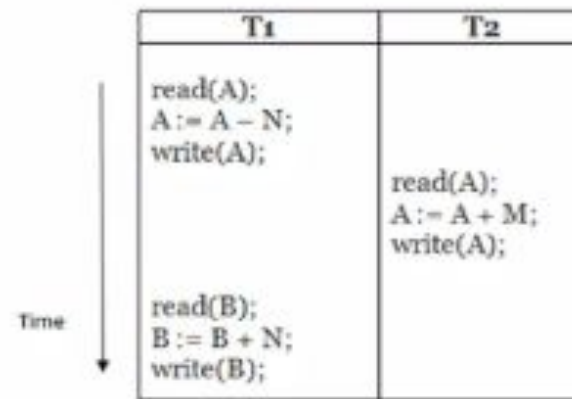
- ◉ If there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible results:

NON-SERIAL SCHEDULE EXAMPLE:

- Schedule C and Schedule D are the non-serial schedules. Because It has interleaving of operations.



Schedule C



Schedule D

3. SERIALIZABLE SCHEDULE:

- ⦿ A serializable schedule always leaves the database in consistent state. A serial schedule is always a serializable schedule.
- ⦿ However, a non-serial schedule needs to be checked for Serializability.
- ⦿ A transaction schedule is serializable if its outcome is equal to the outcome of its transaction executing serially.

CONFLICTING INSTRUCTIONS:

- Let us consider I1 and I2 are two consecutive instructions of T1 and T2 respectively. Both the instructions are operating on same data item Q, these instructions are said to be conflicting instructions if any one of the following instructions occurs simultaneously.
- I1 = read(Q) & I2 = write(Q)
- I1 = write(Q) & I2 = read(Q)
- I1 = write(Q) & I2 = write(Q)

It means in both the instructions one operation is write operation on the same data item leads to the conflicts.

- Conflicting instructions cannot be swapped (interchange) otherwise the operation of the schedule will be changed.

Conflict Serializability

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
write (Q)	write (Q)

- We are unable to swap instructions in the above schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$, or the serial schedule $\langle T_4, T_3 \rangle$.

Recoverable Schedules

- **Recoverable schedule** — if a transaction T_k reads a data item previously written by a transaction T_i , then the **commit operation of T_i** must appear before the **commit operation of T_k** .

- Example:

this schedule is **not recoverable** if T_9 commits immediately after the read(A) operation.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Recoverable Schedule

T1	T2
• Read(x)	
• Write(x)	
	• Read(y)
• Commit	
	• Read(x)

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state.
- Hence, **database must ensure that schedules are recoverable.**

Cascading Rollbacks


- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A) read (B) write (A)	read (A) write (A)	
abort		read (A)

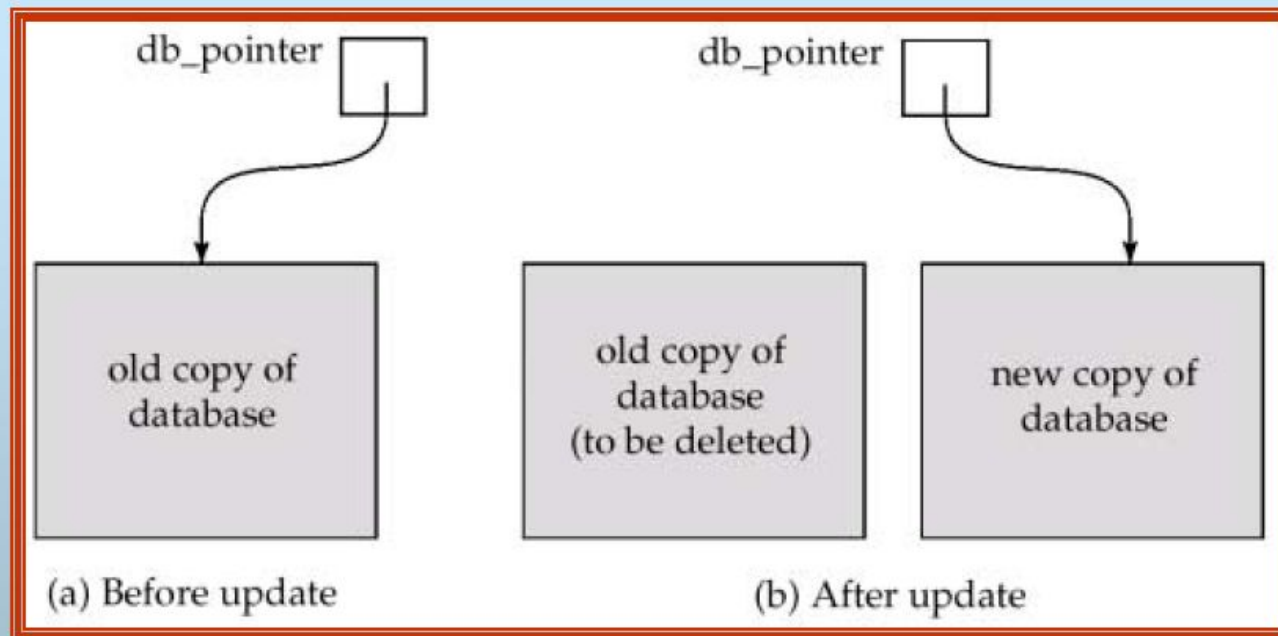
If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.
 - The *shadow-database* scheme:
 - ★ assume that only one transaction is active at a time.
 - ★ a pointer called `db_pointer` always points to the current consistent copy of the database.
 - ★ all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
 - ★ in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.
- 

The shadow-database scheme:



- Assumes disks to not fail
- Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying



Implementation of Isolation Levels

- Locking
 - Lock on whole database vs lock on items
 - How long to hold lock?
 - Shared vs exclusive locks
- Timestamps
 - Transaction timestamp assigned e.g. when a transaction begins
 - Data items store two timestamps
 - Read timestamp
 - Write timestamp
 - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
 - Allow transactions to read from a “snapshot” of the database

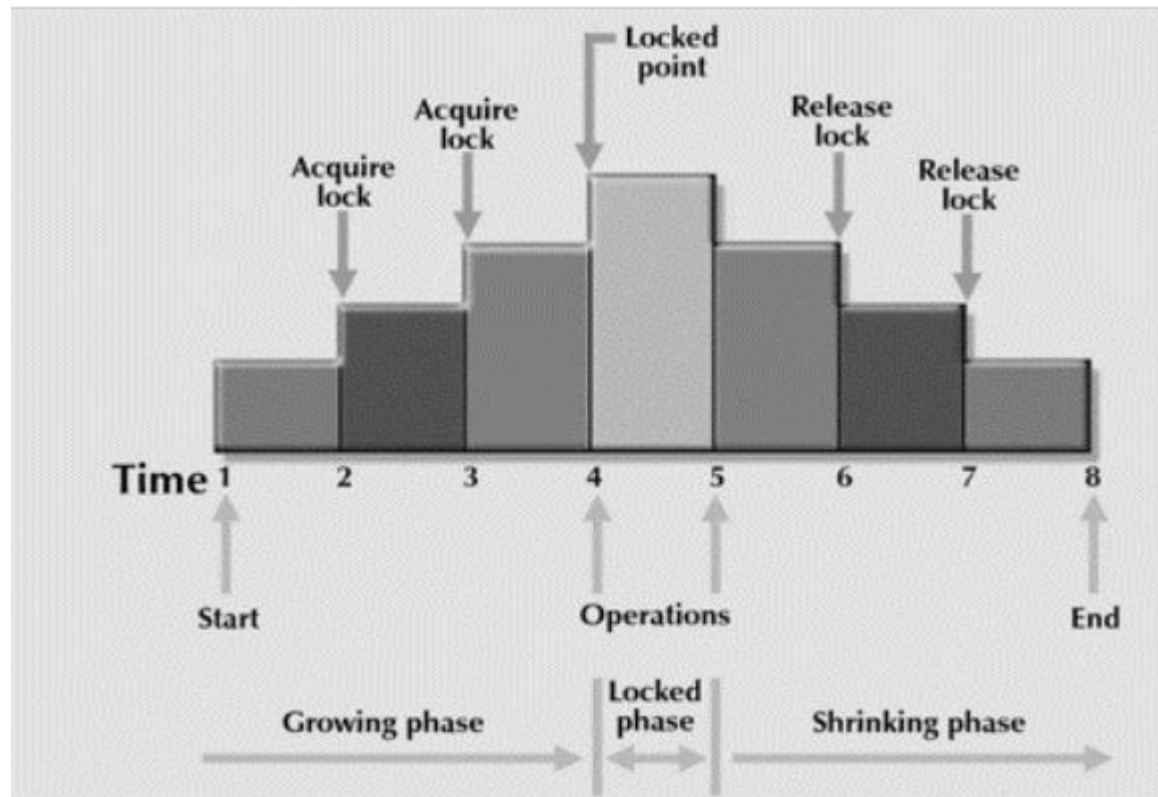
Two phase locking (2PL) protocol(DBMS)

- The protocol uses locks, applied by a transaction to data, which may block (interpreted as signals to stop) other transactions from accessing the same data during the transaction's life.

2PL locking protocol

- Every [transaction](#) will lock and unlock the data item in two different phases.
- **Growing Phase** – All the locks are issued in this phase. No locks are released, after all changes to data-items are committed and then the second phase (shrinking phase) starts.
- **Shrinking phase** – No locks are issued in this phase, all the changes to data-items are noted (stored) and then locks are released.

2PL locking protocol



Two Phase Locking

- A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases.
- **Growing Phase:** New locks on data items may be acquired but none can be released.
- **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

If lock conversion is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in the shrinking phase.

Transaction implementing 2-PL

	T1	T2
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10

2-Phase Locking Protocol is a concurrency protocol that helps in serializability.

Lock Point

The Point at which the growing phase ends, i.e., when a transaction takes the final lock it needs to carry on its work.

- **Transaction T_1**

The growing Phase is from steps 1-3

The shrinking Phase is from steps 5-7

Lock Point at 3

- **Transaction T_2**

The growing Phase is from steps 2-6

The shrinking Phase is from steps 8-9

Lock Point at 6

	T1	T2
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10

Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it.

There are two types of lock:

1. **Shared lock:**

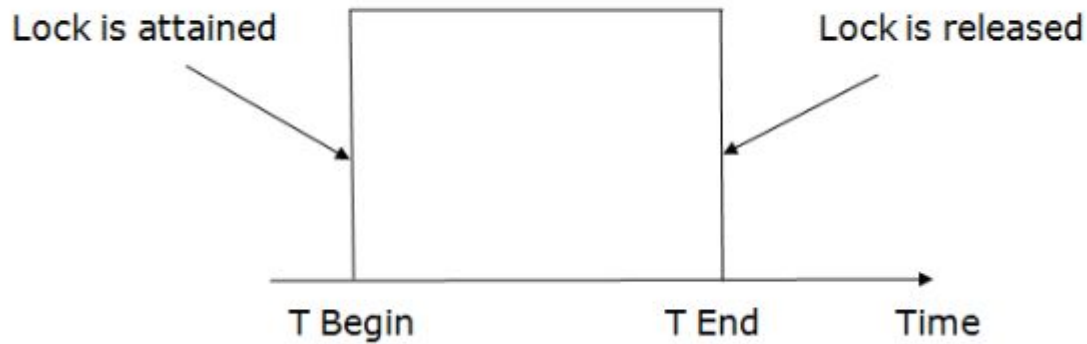
It is also known as a Read-only lock. In a shared lock, the data item can only be read by the transaction.

It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. **Exclusive lock:**

In the exclusive lock, the data item can be both read as well as written by the transaction.

This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.



Two-phase locking (2PL)

The two-phase locking protocol divides the execution phase of the transaction into three parts.

1. In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
2. In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
3. In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.

Types of 2-Phase Locking

- Strict 2-Phase Locking
- Rigorous Two Phase Locking (Rigorous 2PL)
- Conservative Two Phase Locking (Conservative 2PL)

Strict 2-Phase Locking

```
# Strict 2PL example
def strict_2pl(transaction_a, transaction_b):

    # Begin transaction A
    transaction_a.begin()

    # Transaction A acquires lock on resource X
    transaction_a.lock(resource_x)

    # Transaction A performs operation on resource X
    transaction_a.operate(resource_x)

    # Commit transaction A
    transaction_a.commit()

    # Begin transaction B
    transaction_b.begin()

    # Transaction B acquires lock on resource X
    transaction_b.lock(resource_x)

    # Transaction B performs operation on resource X
    transaction_b.operate(resource_x)

    # Commit transaction B
    transaction_b.commit()
```

In strict 2PL, a transaction is not allowed to release any locks until it has reached the commit point.

This means that a transaction will hold all of its locks until it has completed its execution and is ready to be committed.

Rigorous Two Phase Locking (Rigorous 2PL)

- Rigorous 2PL is similar to strict 2PL, but with a slight relaxation of the locking protocol.
- In rigorous 2PL, a transaction is allowed to release a lock if it is certain that it will not need the lock again. For example, if a transaction is reading a resource and it knows that it will not need to write to the resource, it can release the lock after reading.
- The advantage of rigorous 2PL is that it allows for increased concurrency, as transactions are able to release locks that they no longer need. This can lead to less contention for resources and improved performance.

Rigorous Two Phase Locking (Rigorous 2PL)

```
# Rigorous 2PL example
def rigorous_2pl(transaction_a, transaction_b):

    # Begin transaction A
    transaction_a.begin()

    # Transaction A acquires lock on resource X for reading
    transaction_a.lock_shared(resource_x)

    # Transaction A reads resource X
    data = transaction_a.read(resource_x)

    # Transaction A releases lock on resource X
    transaction_a.unlock(resource_x)

    # Begin transaction B
    transaction_b.begin()

    # Transaction B acquires lock on resource X for writing
    transaction_b.lock(resource_x)

    # Transaction B updates resource X with new data
    transaction_b.write(resource_x, data)

    # Commit transaction B
    transaction_b.commit()
```

Conservative Two Phase Locking (Conservative 2PL)

- Conservative 2PL is a less restrictive form of 2PL than strict 2PL and rigorous 2PL. In conservative 2PL, a transaction is allowed to release any lock at any time, regardless of whether it will need the lock again.
- The advantage of conservative 2PL is that it allows for maximum concurrency, as transactions are able to release locks at any time. This can lead to the best performance in terms of throughput and response time.
- The disadvantage of conservative 2PL is that it does not guarantee serializability and can lead to inconsistent results if not implemented carefully. Additionally, it does not prevent deadlocks which could cause transaction to hang.

Conservative Two Phase Locking (Conservative 2PL)

```
# Conservative 2PL example
def conservative_2pl(transaction_a, transaction_b):

    # Begin transaction A
    transaction_a.begin()

    # Transaction A acquires lock on resource X
    transaction_a.lock(resource_x)

    # Transaction A performs operation on resource X
    transaction_a.operate(resource_x)

    # Transaction A releases lock on resource X
    transaction_a.unlock(resource_x)

    # Begin transaction B
    transaction_b.begin()

    # Transaction B acquires lock on resource X
    transaction_b.lock(resource_x)

    # Transaction B performs operation on resource X
    transaction_b.operate(resource_x)

    # Commit transaction B
    transaction_b.commit()
```

Introduction to Indexing Techniques

Database indexing plays a crucial role in improving the performance and efficiency of database systems.

By utilizing indexing techniques, speed up data retrieval operations and enhance overall system responsiveness.

Database indexing techniques, including B-tree, Hash Indexing,.

Indexing

- Indexing is a technique in **DBMS** that is used to optimize the performance of a database by reducing the number of disk access required.
- An index is a type of data structure. With the help of an index, we can locate and access data in database tables faster.
- The dense index and Sparse index are two different approaches to organizing and accessing data in the data structure. These are commonly used in databases and information retrieval systems.

Structure of index

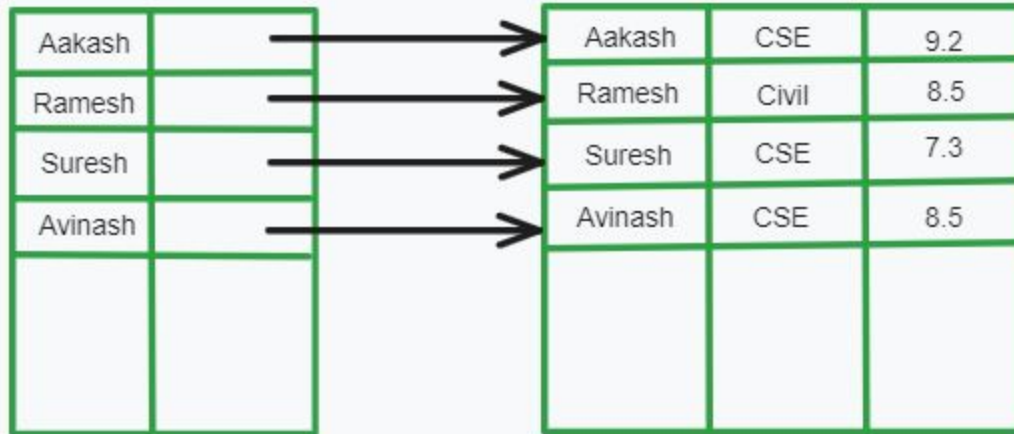
Structure of index

Search key	Data reference
------------	----------------

Dense Index

- It contains an index record for every search key value in the file. This will result in making searching faster.
- The total number of records in the index table and main table are the same.
- It will result in the requirement for more space to store the index of records itself.

Dense Index



Advantages

- Gives quick access to records, particularly for Small datasets.
- Effective for range searches since each key value has an entry.

Disadvantages

- Can be memory-intensive and may require a significant amount of storage space.
- Insertions and deletions result in a higher maintenance overhead because the
- index must be updated more frequently.

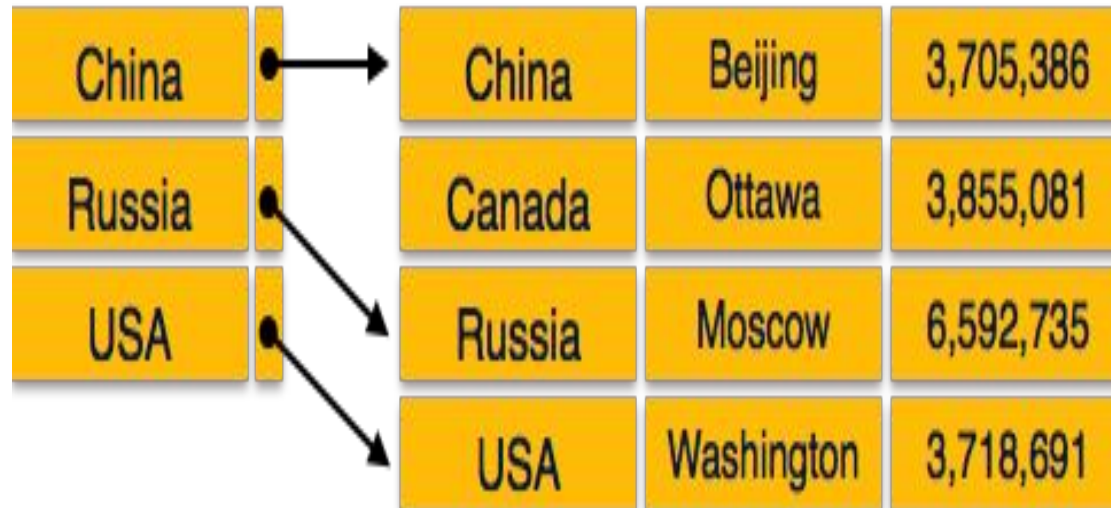
Sparse Index

Sparse index contains an index entry only for some records. In the place of pointing to all the records in the main table index points records in a specific gap. This indexing helps you to overcome the issues of dense indexing in DBMS.

-
- **Advantages**
- Uses less storage space than thick indexes, particularly for large datasets.
- Lessens the effect that insertions and deletions have from index maintenance operations.
- **Disadvantages**
- Since there may not be an index entry for every key value, access may involve additional steps.

.

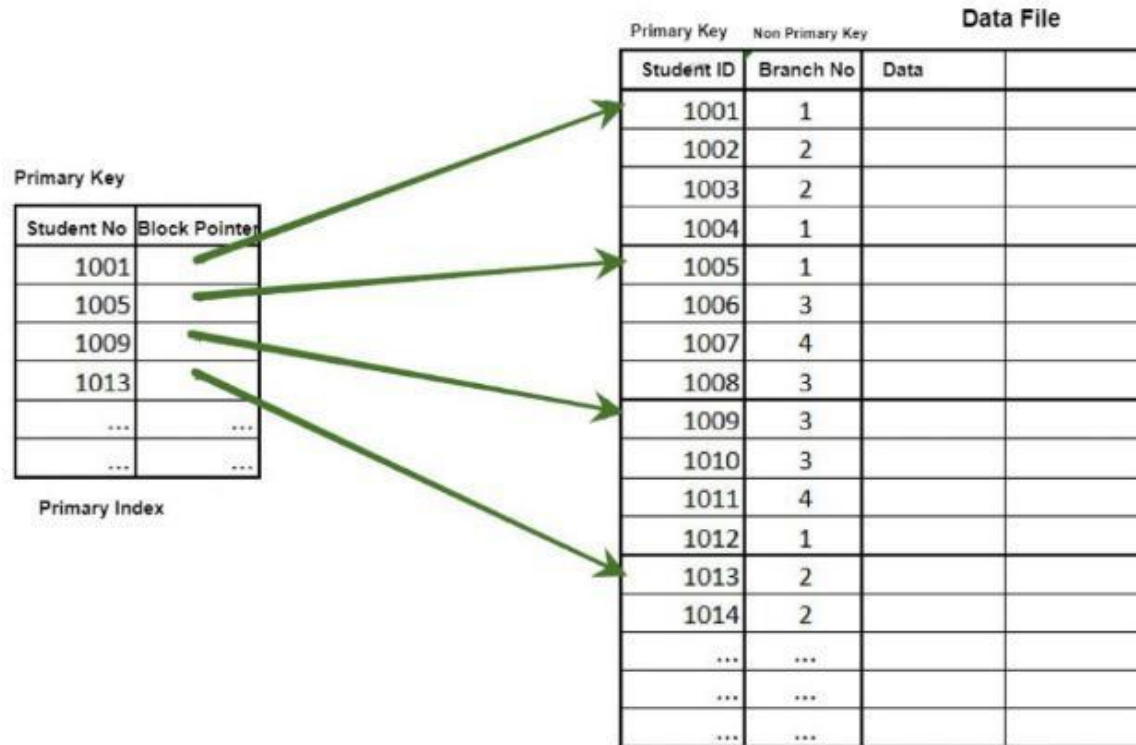
Sparse Index



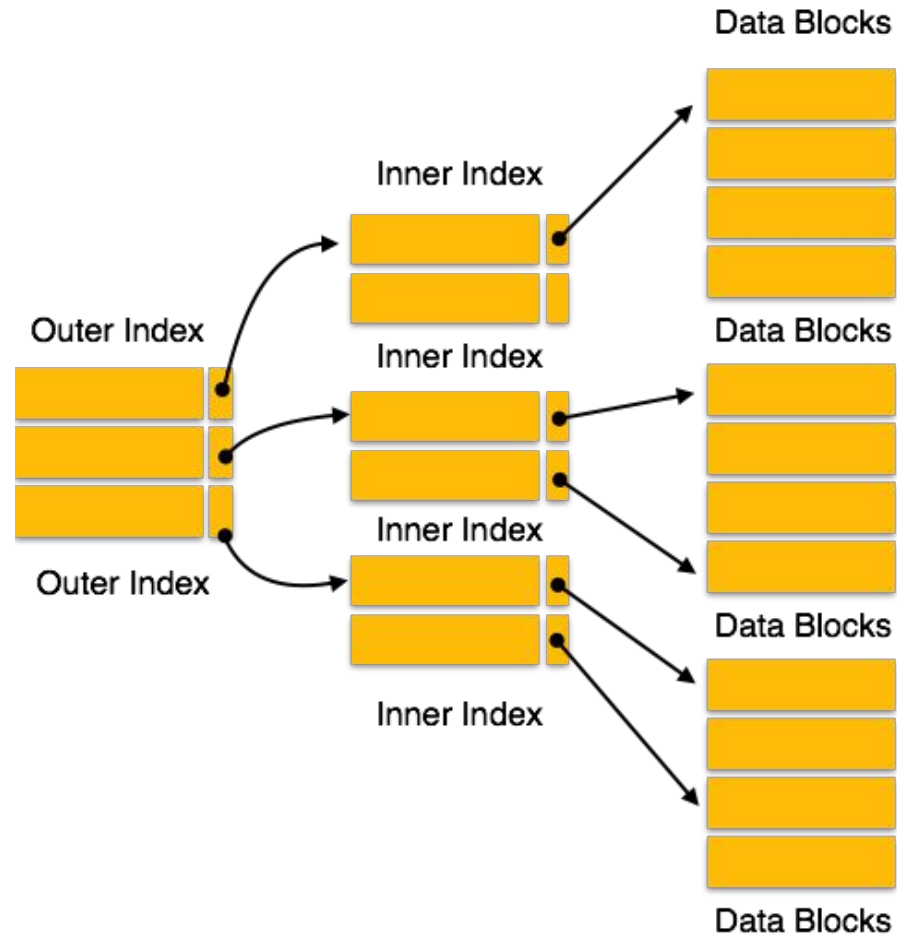
Multilevel Index

- Index records comprise search-key values and data pointers. Multilevel index is stored on the disk along with the actual database files.
- As the size of the database grows, so does the size of the indices.
- There is an immense need to keep the index records in the main memory so as to speed up the search operations.
- If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.

Primary Index

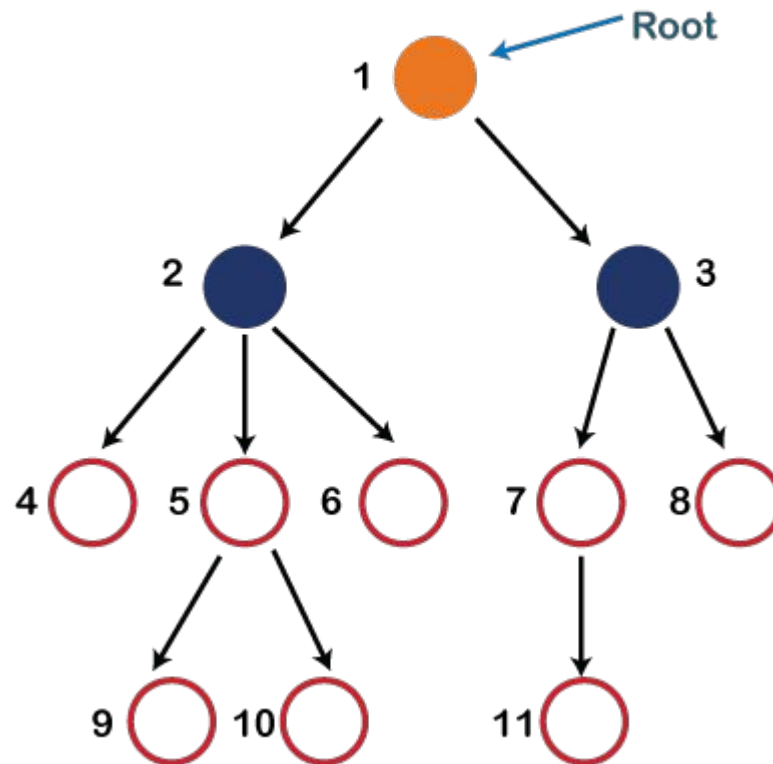


Multilevel Index

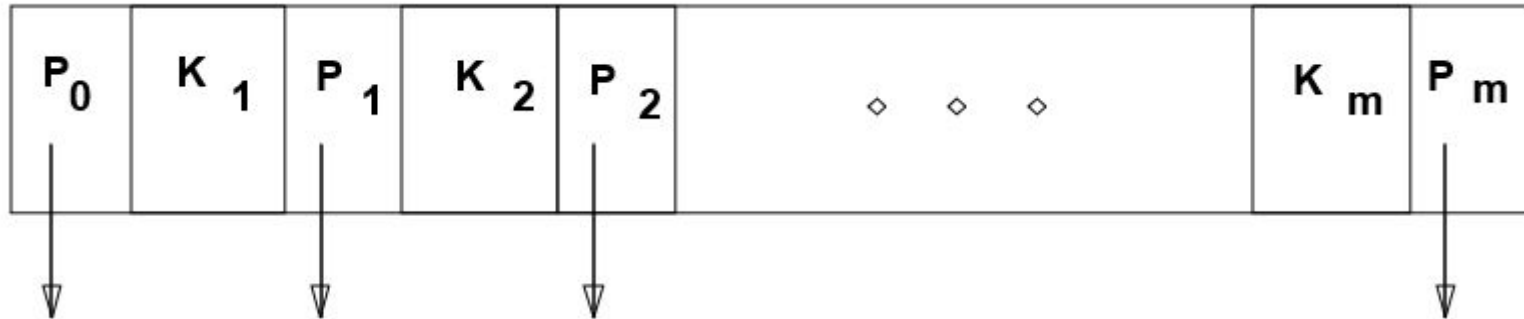


Trees

Introduction to Trees



M-Way Search Tree



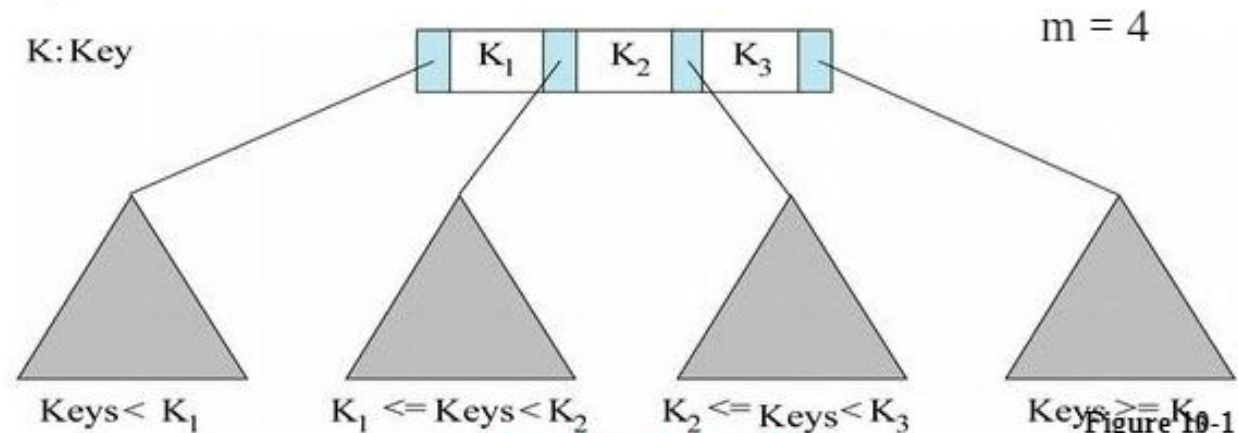
M-Way Search Tree

Each node has 0 to m subtrees.

Given a node contains k subtree pointers, some of which may be null, and k-1 data entries.

The key values in the first subtree are all less than the key in the first entry, the key values in the other subtrees are all greater than or equal to the key in their parent entry.

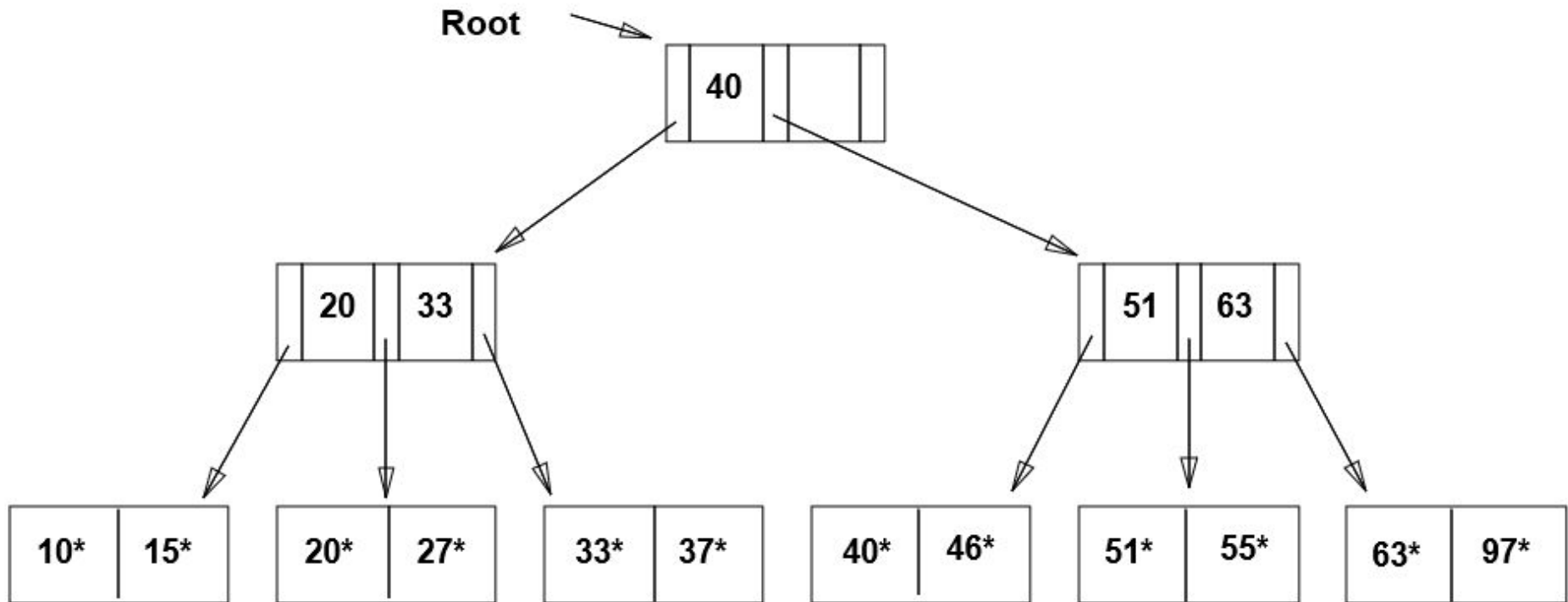
The keys of the data entries are ordered.

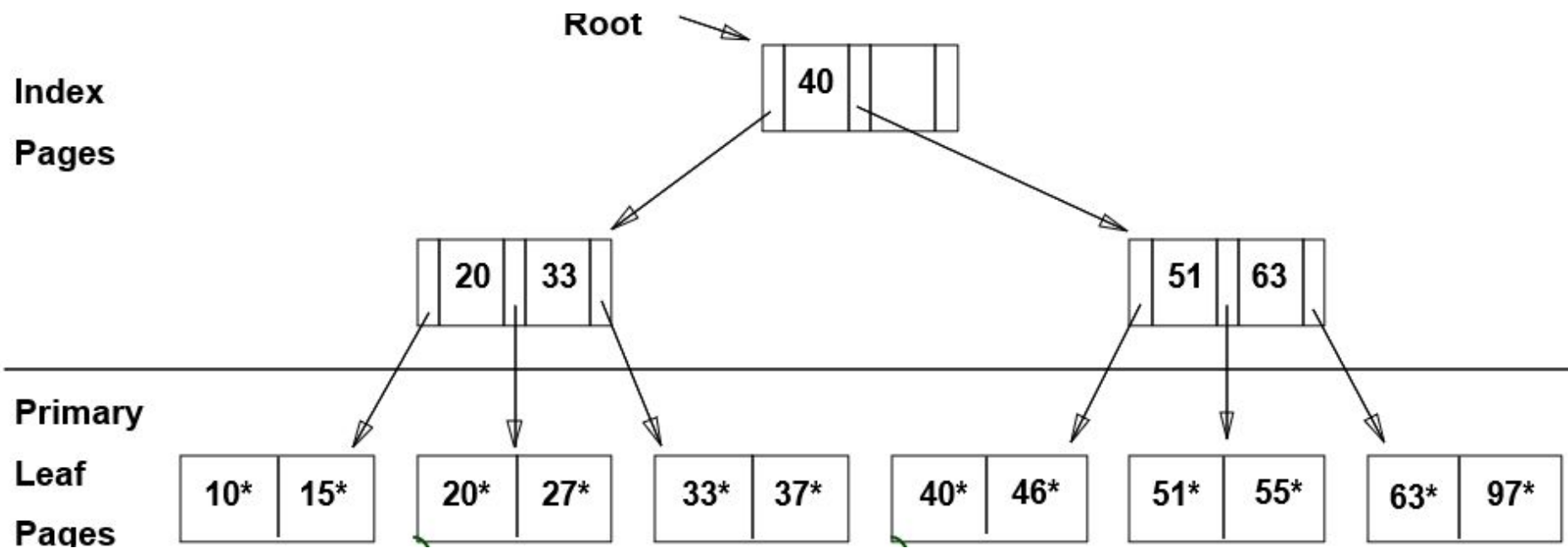


It is not balanced!

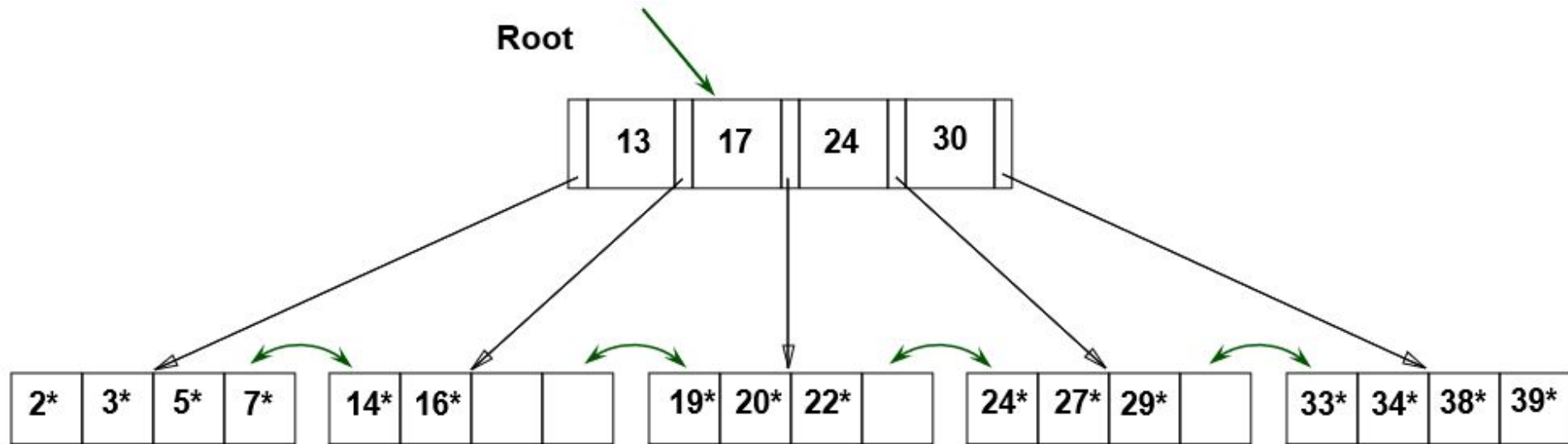
Figure 10-1

3-Way Search Tree

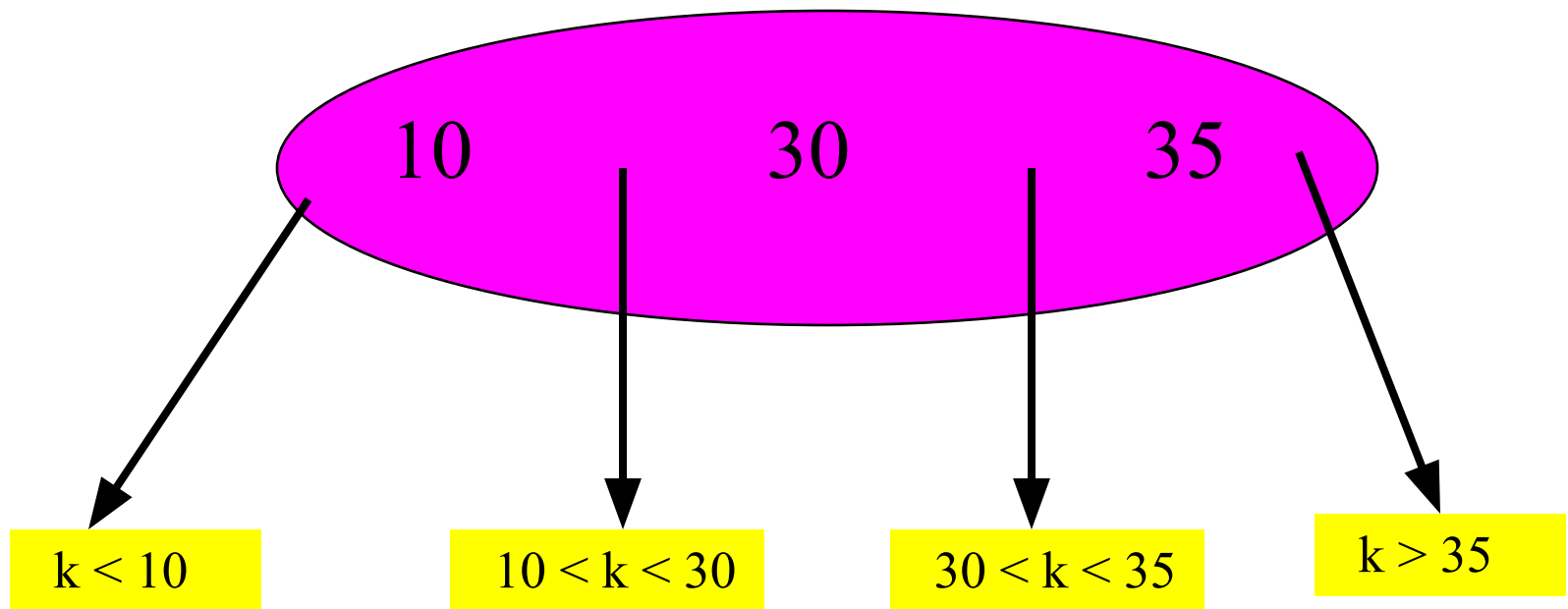




5-Way Search Tree



4-Way Search Tree



What is B-tree Indexing?

B-tree is one of the popular techniques in relational database management systems (RDBMS) is b-tree indexing.

It arranges information in a balanced tree structure to facilitate effective insertion, deletion, and searching.

Large dataset databases that need frequent range queries and dynamic updates are best suited for B-trees.

B-Tree

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

Properties of B-Tree

- Every node has at most m children where m is the order of the B-Tree.
- A node with κ children contains $\kappa-1$ keys.
- Every non-leaf node except the root node must have at least $\lceil m/2 \rceil$ child nodes.
- The root must have at least 2 children if it is not the leaf node too.
- All leaves of a B-Tree stays at the same level.
- Unlike other trees, its height increases upwards towards the root, and insertion happens at the leaf node.

B-Tree Properties

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

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

B+ Tree

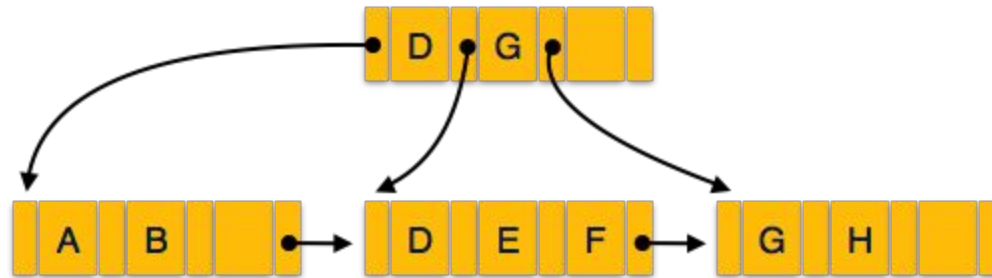
Definition:-

- ▶ The most commonly implemented form of the B-Tree is the B⁺-Tree.
- ▶ The order of the B+ tree determines the number of subtrees, which exactly follows the B-Tree definition.
- ▶ A B+ tree is a data structure often used in the implementation of database indexes.
- ▶ Only Leaf nodes store references to data.
- ▶ A leaf node may store more or less data records than an internal node stores keys.
- ▶ It contains index pages and data pages.
- ▶ The data pages always appear as leaf nodes in the tree.
- ▶ The root node and intermediate nodes are always index pages.

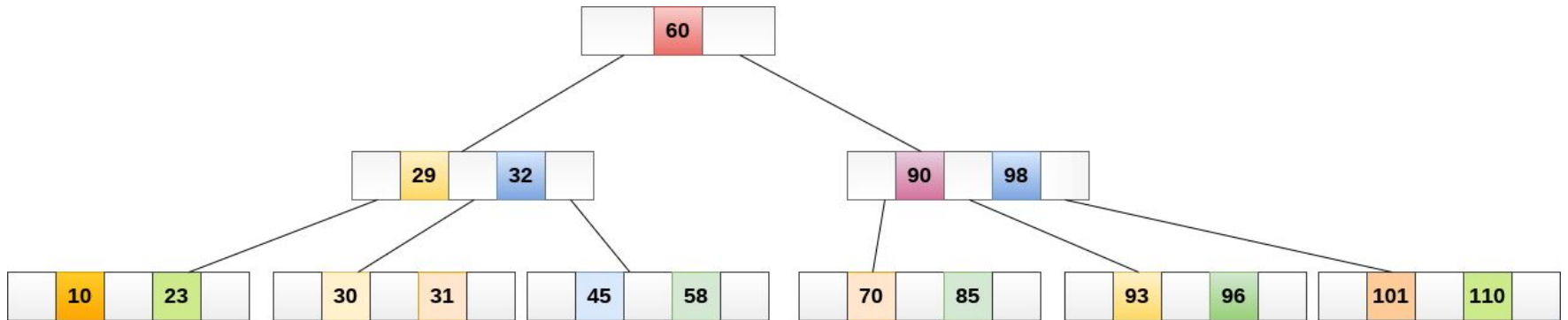
Differences between B-Tree and B+ Tree

B-Tree	B+ Tree
All internal nodes and leaf nodes contain data pointers along with keys.	Only leaf nodes contain data pointers along with keys, internal nodes contain keys only.
There are no duplicate keys.	Duplicate keys are present in this, all internal nodes are also present at leaves.
Leaf nodes are not linked to each other.	Leaf nodes are linked to each other.
Sequential access of nodes is not possible.	All nodes are present at leaves, so sequential access is possible just like a linked list.
Searching for a key is slower.	Searching is faster.

B-Tree Example



A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

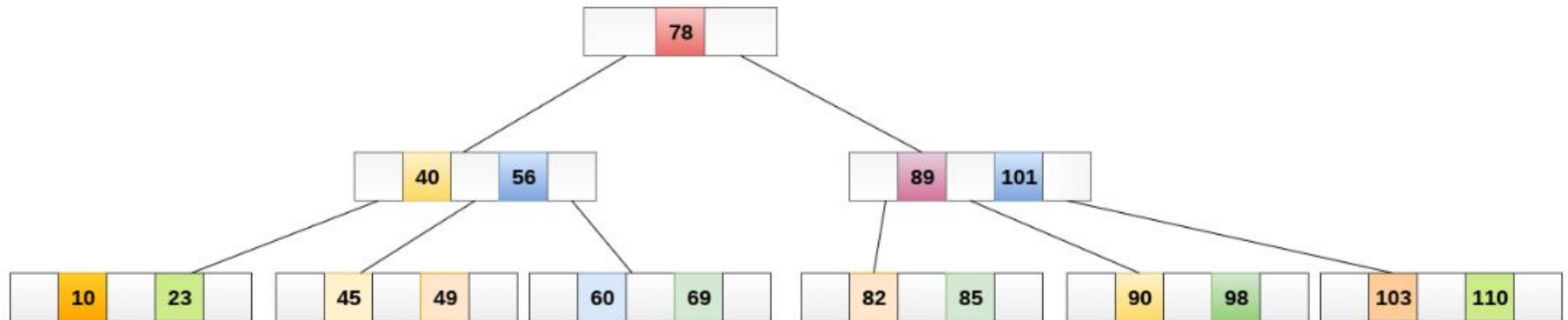
Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since $49 < 78$ hence, move to its left sub-tree.
2. Since, $40 < 49 < 56$, traverse right sub-tree of 40.
3. $49 > 45$, move to right. Compare 49.
4. match found, return.

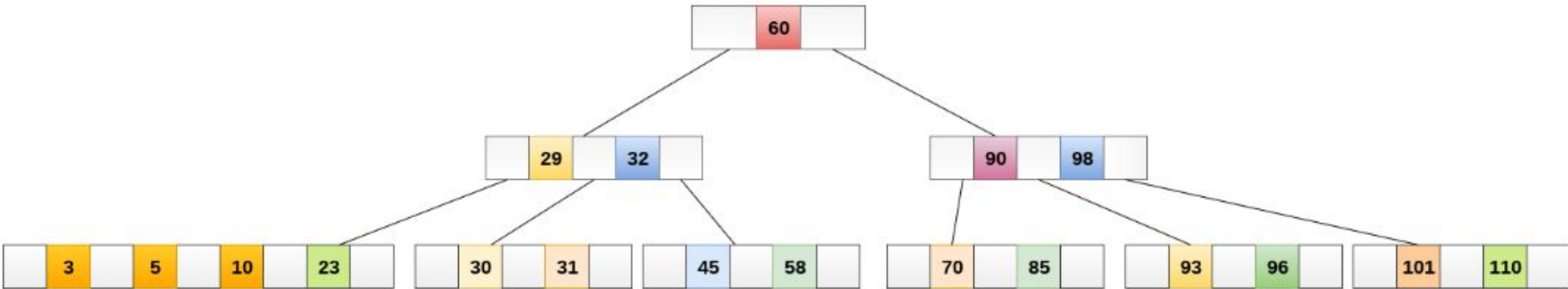
Searching in a B tree depends upon the height of the tree. The search algorithm takes $O(\log n)$ time to search any element in a B tree.



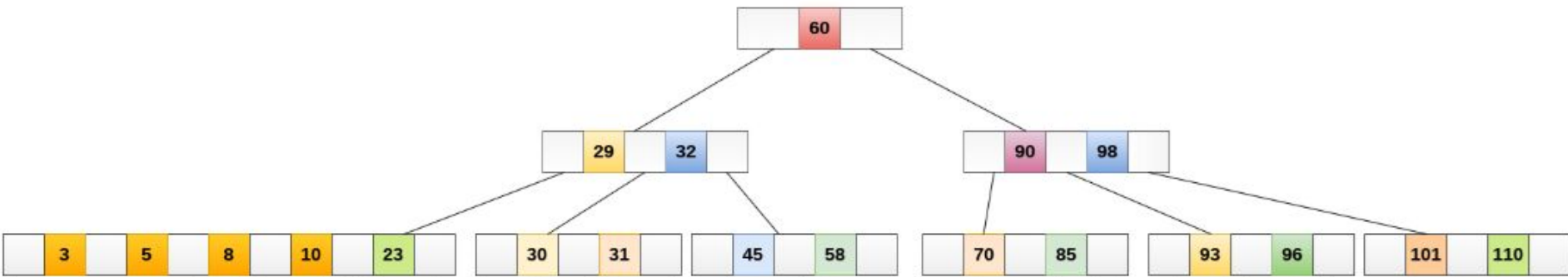
Insert in to B-Tree

- Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.
- Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
- If the leaf node contain less than $m-1$ keys then insert the element in the increasing order.
- Else, if the leaf node contains $m-1$ keys, then follow the following steps.
 - Insert the new element in the increasing order of elements.
 - Split the node into the two nodes at the median.
 - Push the median element upto its parent node.
 - If the parent node also contain $m-1$ number of keys, then split it too by following the same steps.
- Insert the node 8 into the B Tree of order 5

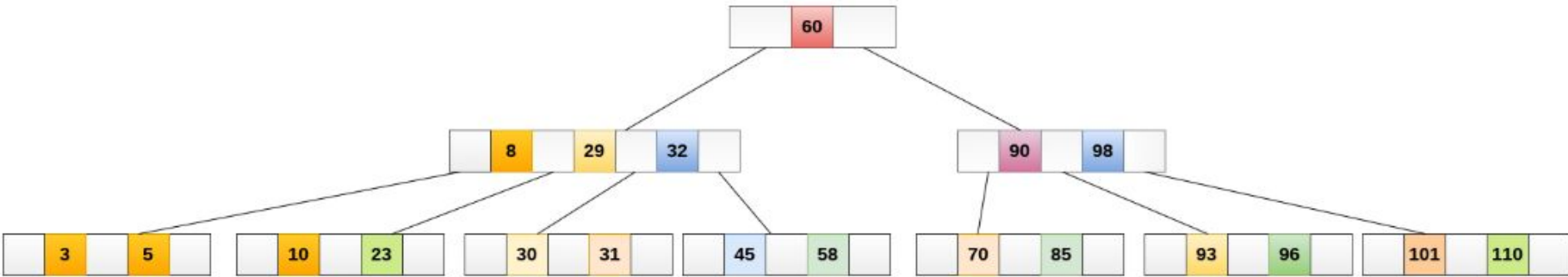
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



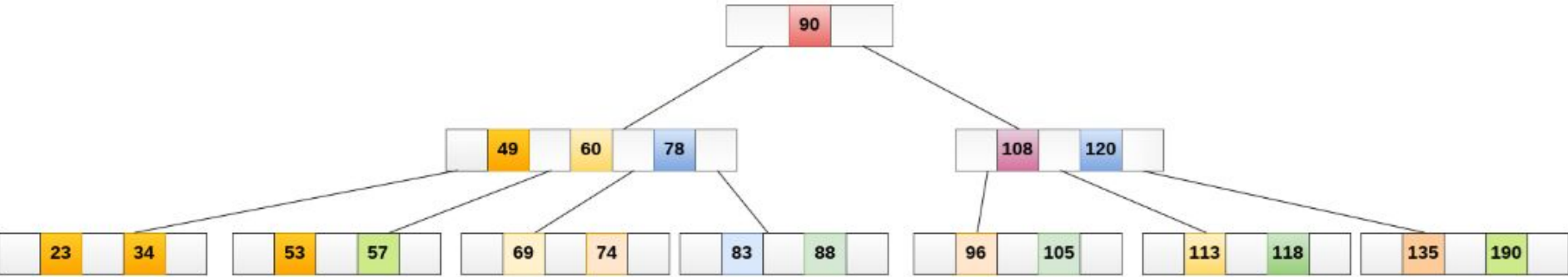
The node, now contain 5 keys which is greater than $(5 - 1 = 4)$ keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



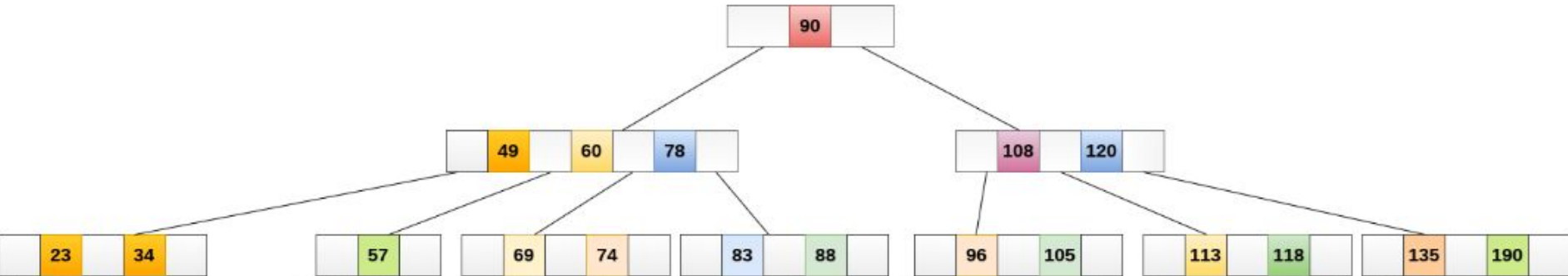
Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
 2. If there are more than $m/2$ keys in the leaf node then delete the desired key from the node.
 3. If the leaf node doesn't contain $m/2$ keys then complete the keys by taking the element from right or left sibling.
 - If the left sibling contains more than $m/2$ elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
 - If the right sibling contains more than $m/2$ elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
-
1. If neither of the sibling contain more than $m/2$ elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
 2. If parent is left with less than $m/2$ nodes then, apply the above process on the parent too.
- If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

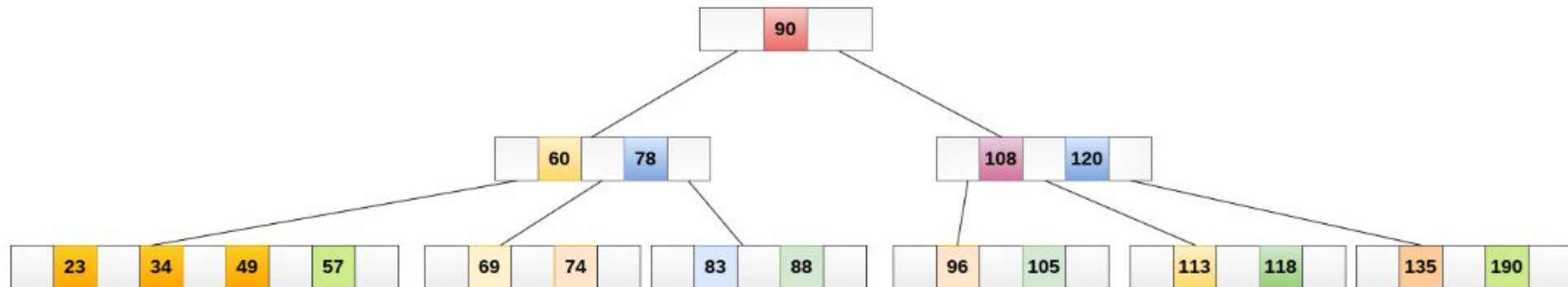


53 is present in the right child of element 49. Delete it.

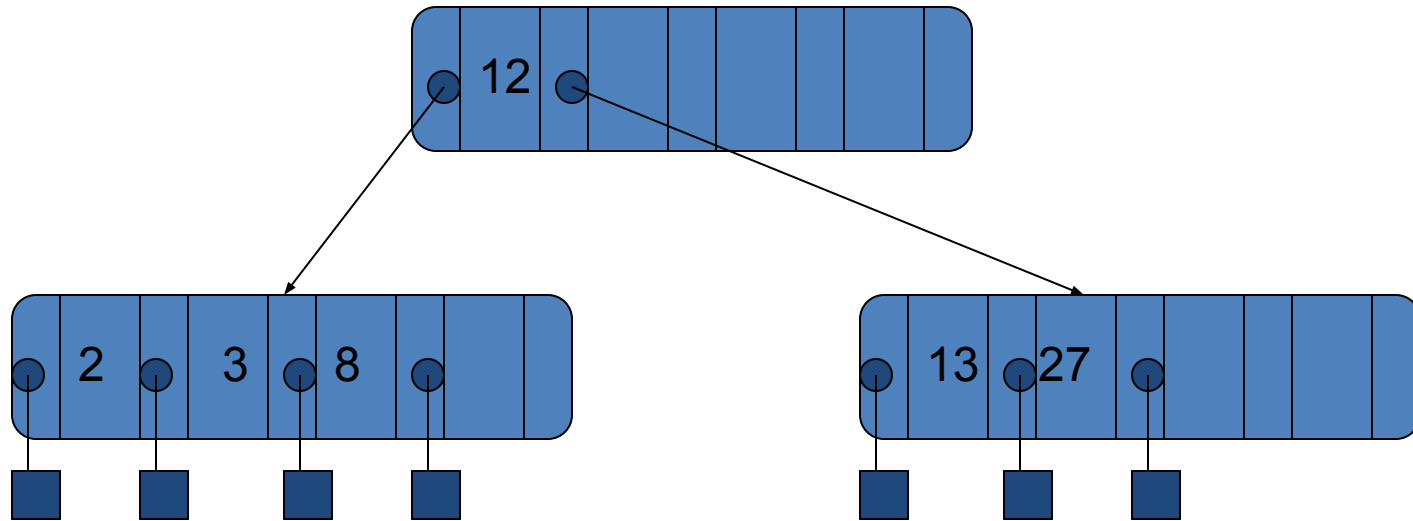


Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows after Deletion



B-Tree Example with $m = 5$

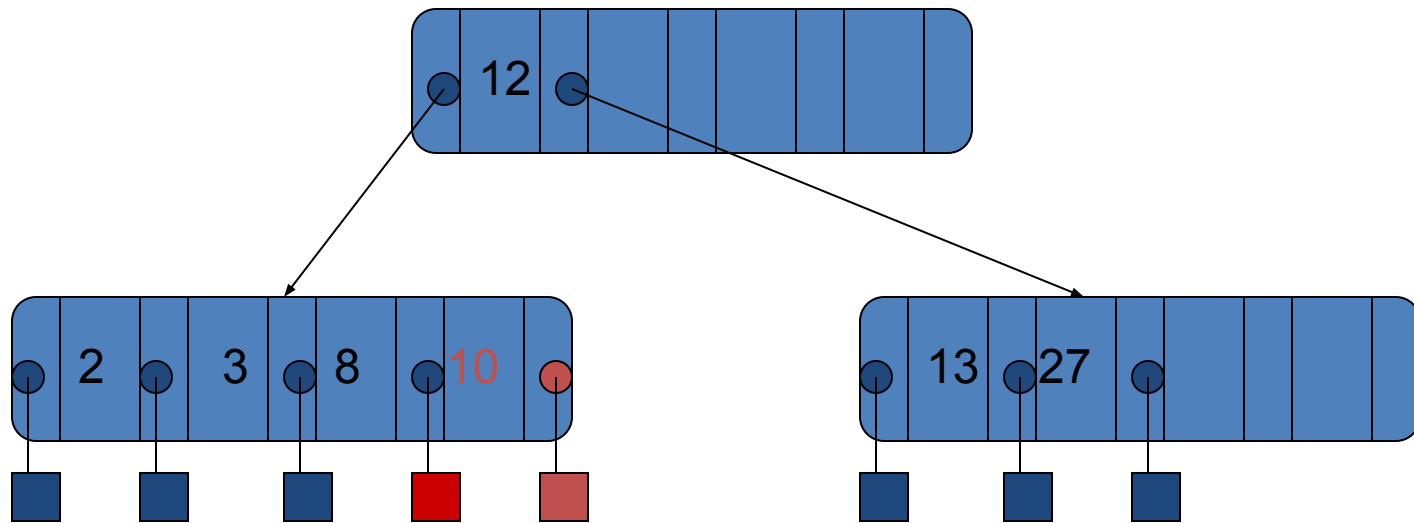


The root has between 2 and m children.

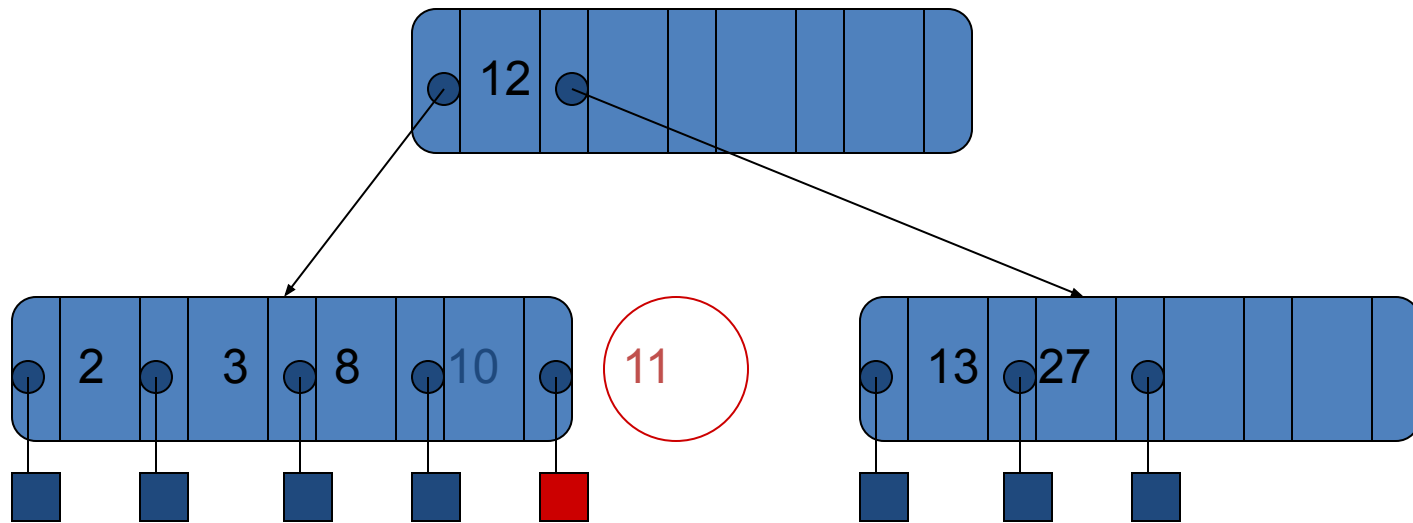
Each non-root internal node has between $\lceil m/2 \rceil$ and m children.

All external nodes are at the same level. (External nodes are actually represented by null pointers in implementations.)

Insert 10



Insert 11

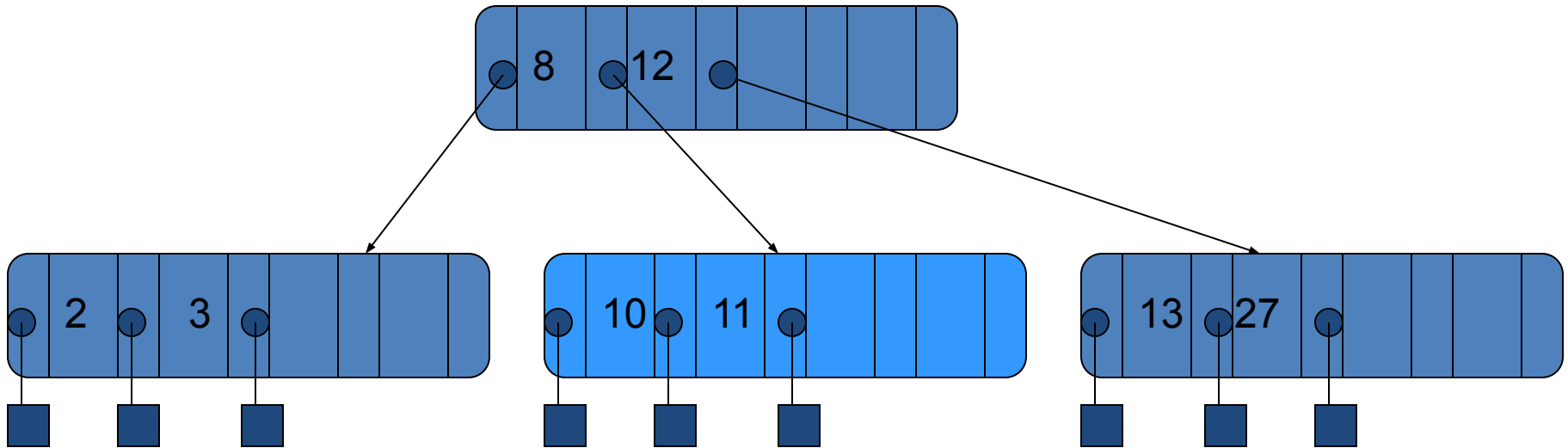


We fall out of the tree at the child to the right of key 10.

But there is no more room in the left child of the root to hold 11.

Therefore, we must *split* this node...

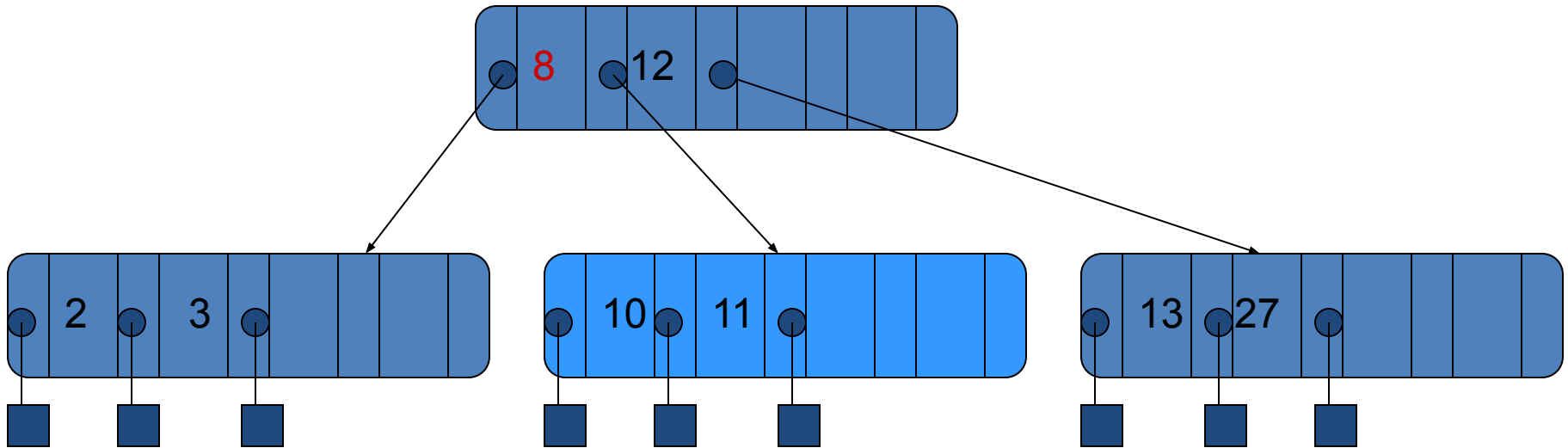
Insert 11 (Continued)



The $m + 1$ children are divided evenly between the old and new nodes.

The parent gets one new child. (If the parent become overfull, then it, too, will have to be split).

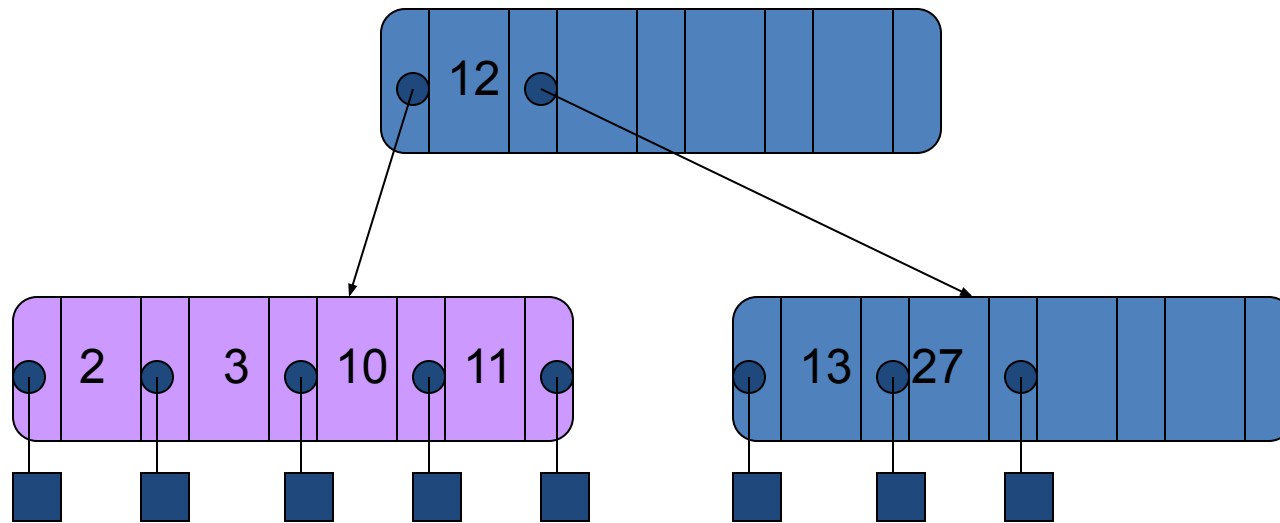
Remove 8



Removing 8 might force us to move another key up from one of the children. It could either be the 3 from the 1st child or the 10 from the second child.

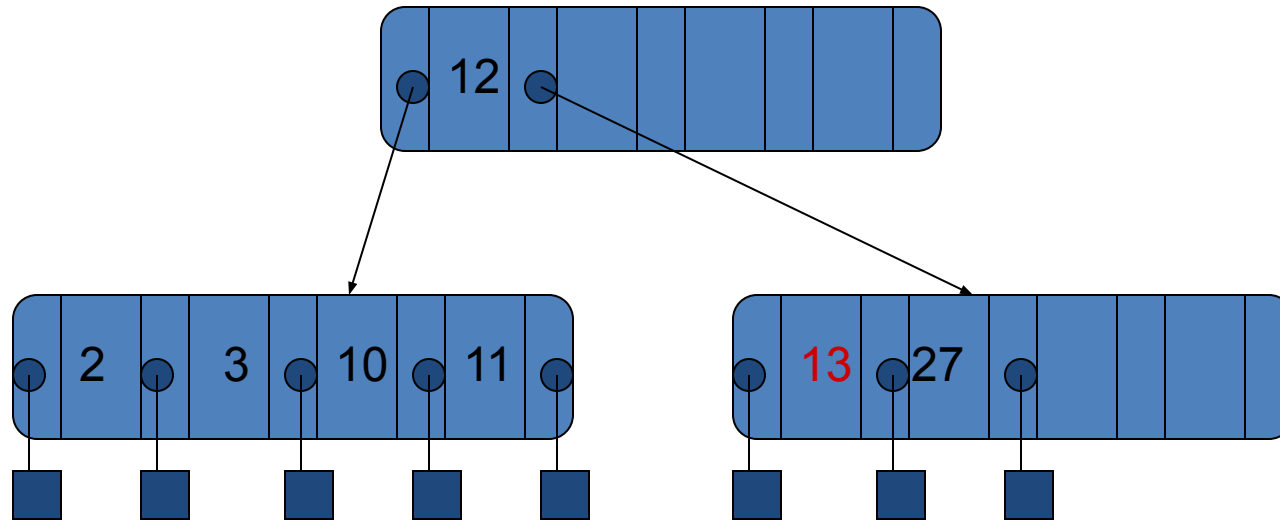
However, neither child has more than the minimum number of children (3), so the two nodes will have to be merged. Nothing moves up.

Remove 8 (Continued)



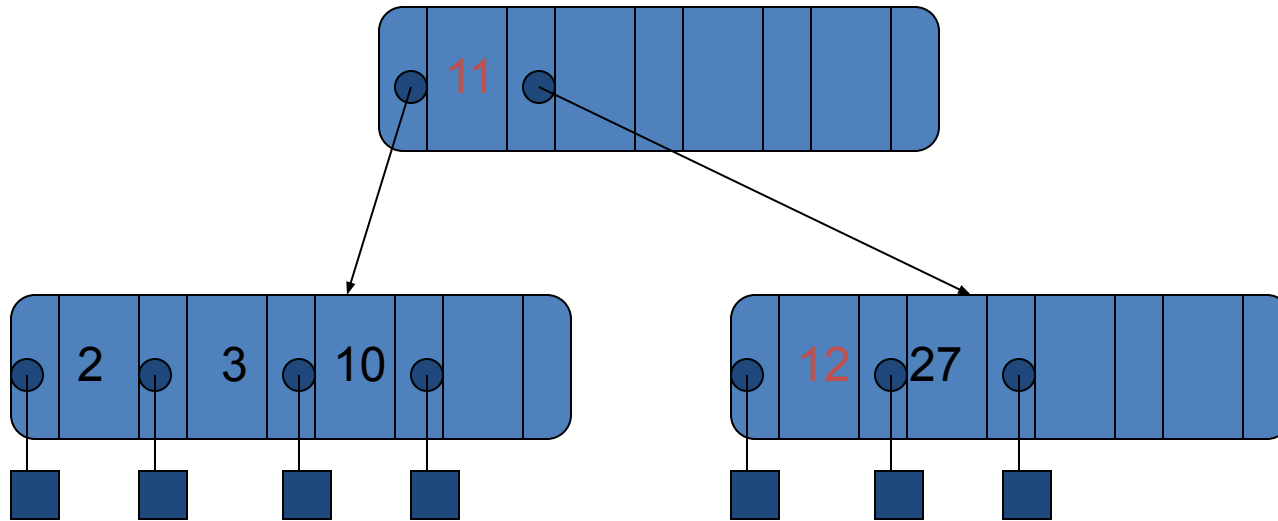
The root contains one fewer key, and has one fewer child.

Remove 13



Removing 13 would cause the node containing it to become underfull.
To fix this, we try to reassign one key from a sibling that has spares.

Remove 13 (Cont)

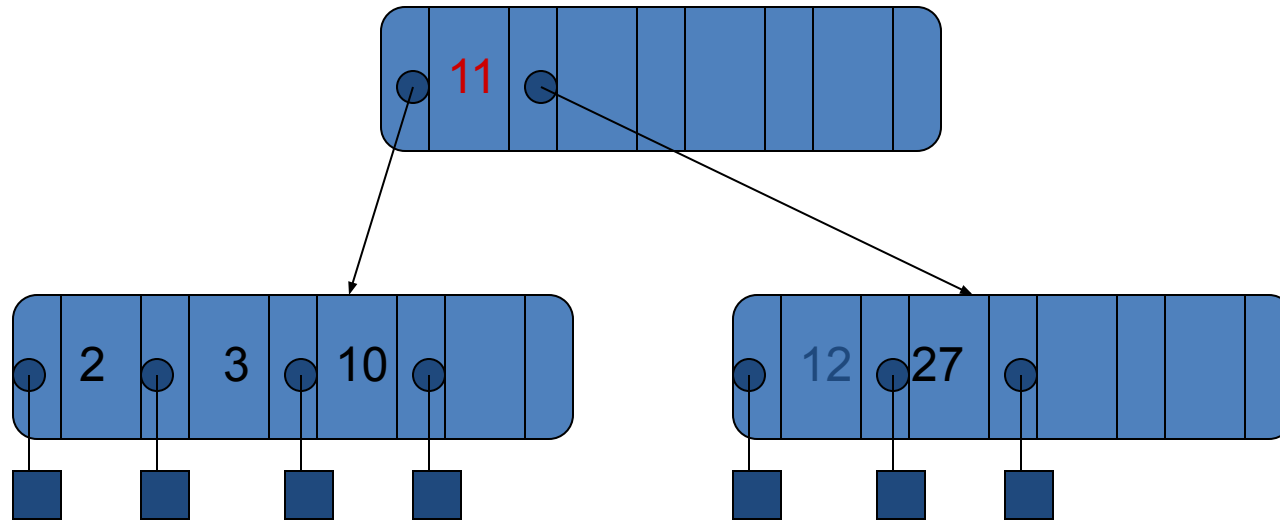


The 13 is replaced by the parent's key 12.

The parent's key 12 is replaced by the spare key 11 from the left sibling.

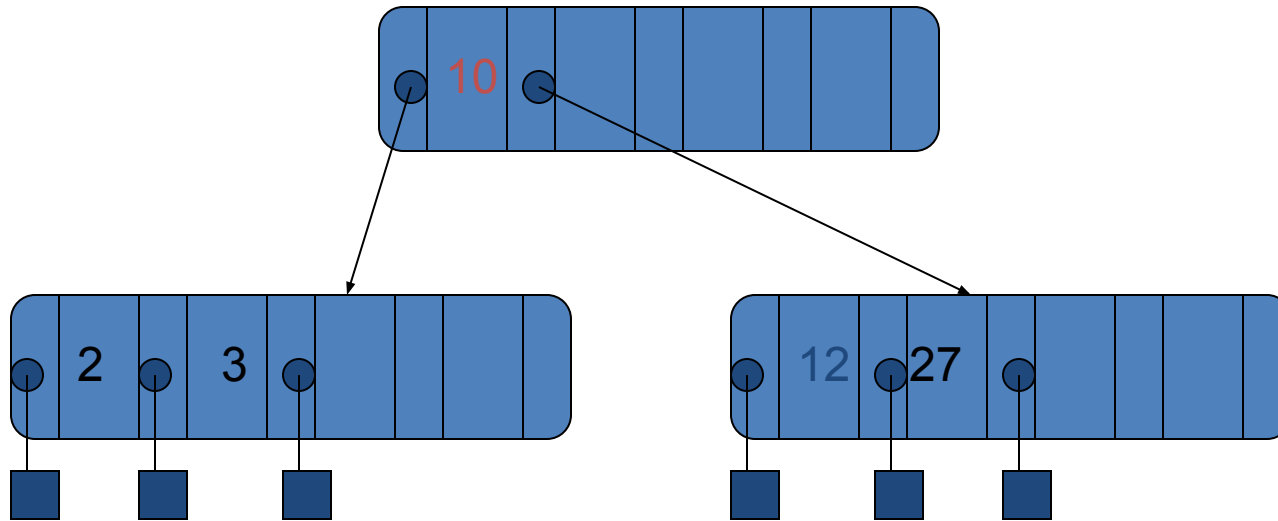
The sibling has one fewer element.

Remove 11

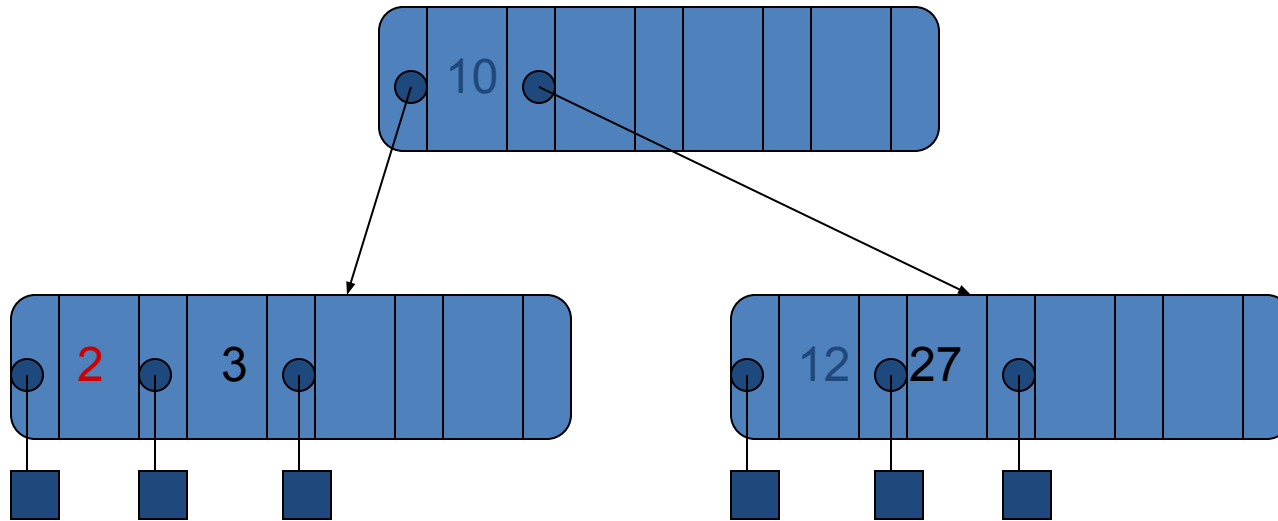


11 is in a non-leaf, so replace it by the value immediately preceding: 10.
10 is at leaf, and this node has spares, so just delete it there.

Remove 11 (Cont)



Remove 2

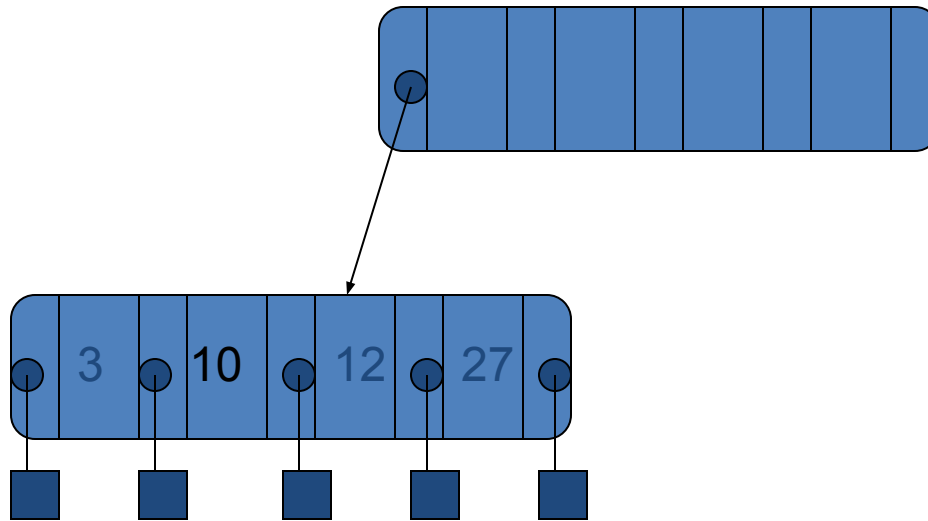


Although 2 is at leaf level, removing it leads to an underfull node.

The node has no left sibling. It does have a right sibling, but that node is at its minimum occupancy already.

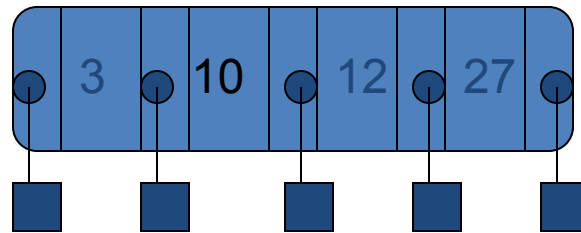
Therefore, the node must be merged with its right sibling.

Remove 2 (Cont)



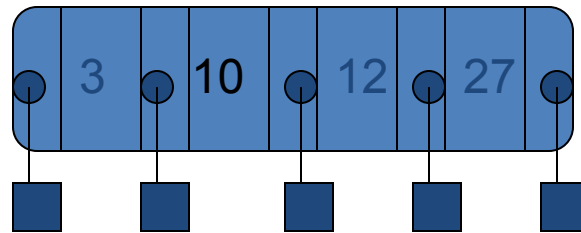
The result is illegal, because the root does not have at least 2 children.
Therefore, we must remove the root, making its child the new root.

Remove 2 (Cont)



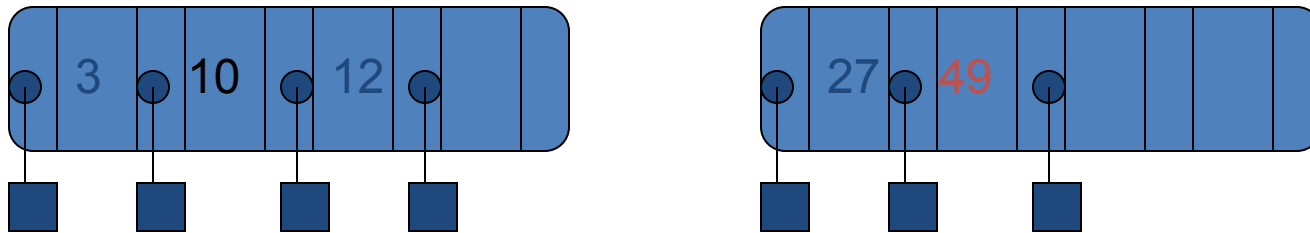
The new B-tree has only one node, the root.

Insert 49



Let's put an element into this B-tree.

Insert 49 (Cont)

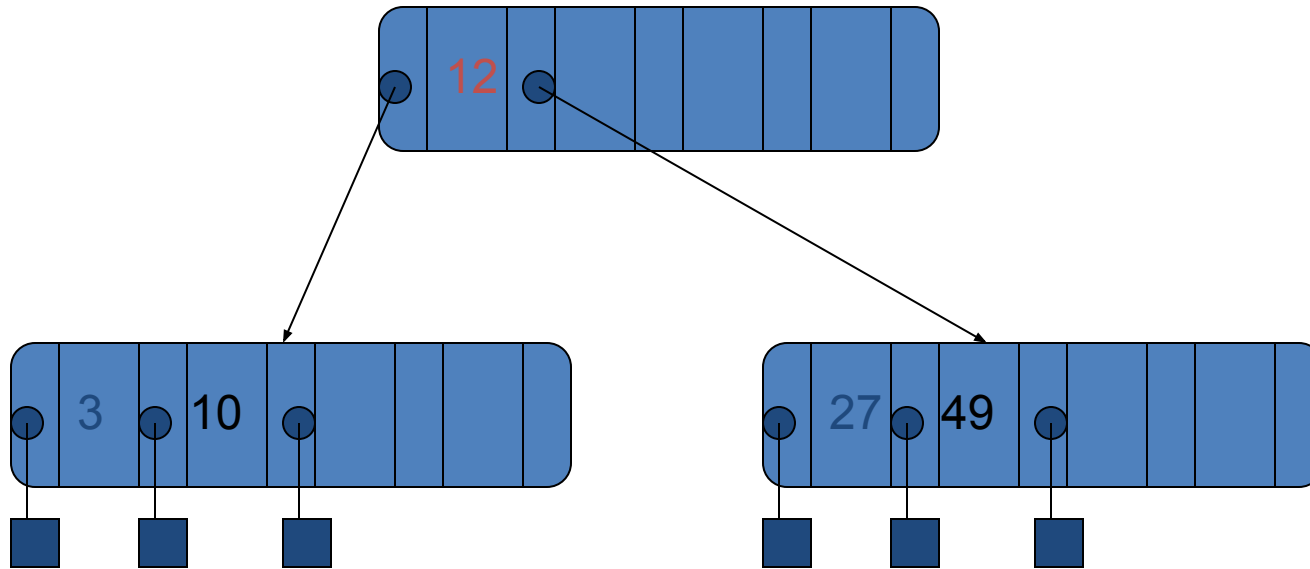


Adding this key make the node overfull, so it must be split into two.

But this node was the root.

So we must construct a new root, and make these its children.

Insert 49 (Cont)



The middle key (12) is moved up into the root.

The result is a B-tree with one more level.

B+ tree

B⁺ Tree

A B⁺ tree is a balanced binary search tree that follows a multi-level index format. The leaf nodes of a B⁺ tree denote actual data pointers. B⁺ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, the leaf nodes are linked using a link list; therefore, a B⁺ tree can support random access as well as sequential access.

B+ Tree

Definition:-

- ▶ The most commonly implemented form of the B-Tree is the B⁺-Tree.
- ▶ The order of the B+ tree determines the number of subtrees, which exactly follows the B-Tree definition.
- ▶ A B+ tree is a data structure often used in the implementation of database indexes.
- ▶ Only Leaf nodes store references to data.
- ▶ A leaf node may store more or less data records than an internal node stores keys.
- ▶ It contains index pages and data pages.
- ▶ The data pages always appear as leaf nodes in the tree.
- ▶ The root node and intermediate nodes are always index pages.

B+ Tree

Structure of B⁺ Tree

Every leaf node is at equal distance from the root node. A B⁺ tree is of the order **n** where **n** is fixed for every B⁺ tree.

Internal nodes –

- Internal (non-leaf) nodes contain at least $\lceil n/2 \rceil$ pointers, except the root node.
- At most, an internal node can contain **n** pointers.

Leaf nodes –

- Leaf nodes contain at least $\lceil n/2 \rceil$ record pointers and $\lceil n/2 \rceil$ key values.
- At most, a leaf node can contain **n** record pointers and **n** key values.
- Every leaf node contains one block pointer **P** to point to next leaf node and forms a linked list.

B⁺ Tree Insertion

- B⁺ Tree Insertion
- B⁺ trees are filled from bottom and each entry is done at the leaf node.
- If a leaf node overflows –
 - Split node into two parts.
 - Partition at $i = \lfloor (m+1)/2 \rfloor$.
 - First i entries are stored in one node.
 - Rest of the entries ($i+1$ onwards) are moved to a new node.
 - i^{th} key is duplicated at the parent of the leaf.
- If a non-leaf node overflows –
 - Split node into two parts.
 - Partition the node at $i = \lceil (m+1)/2 \rceil$.
 - Entries up to i are kept in one node.
 - Rest of the entries are moved to a new node.

B⁺ Tree Deletion

- B⁺ tree entries are deleted at the leaf nodes.
- The target entry is searched and deleted.
 - If it is an internal node, delete and replace with the entry from the left position.
- After deletion, underflow is tested,
 - If underflow occurs, distribute the entries from the nodes left to it.
- If distribution is not possible from left, then
 - Distribute from the nodes right to it.
- If distribution is not possible from left or from right, then
 - Merge the node with left and right to it.

Properties of B+ Tree:-

For a B-tree of order m :

- ▶ All data are on the same level.
- ▶ If a B+ tree consists of a single node, then the node is both a root and a leaf.
- ▶ Each internal node other than the root contains between $m/2$ search keys.
- ▶ The capacity of a leaf has to be 50% or more.

Operation on B+ Tree:-

- ▶ Searching.
- ▶ Insertion.
- ▶ Deletion.

Searching:-

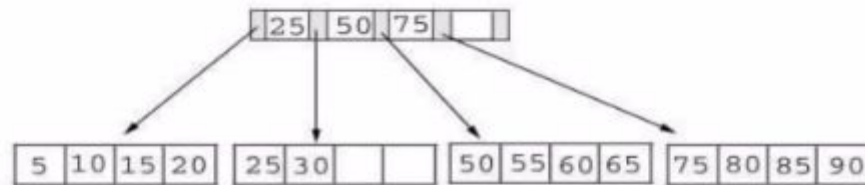
1. Searching just like in a binary search tree.
2. Starts at the root, works down to the leaf level.
3. Does a comparison of the search value and the current “separation value”, goes left or right.
4. Since no structure change in a B+ tree during a searching process.
5. So just compare the key value with the data in the tree, then give the result back.

Insertion:-

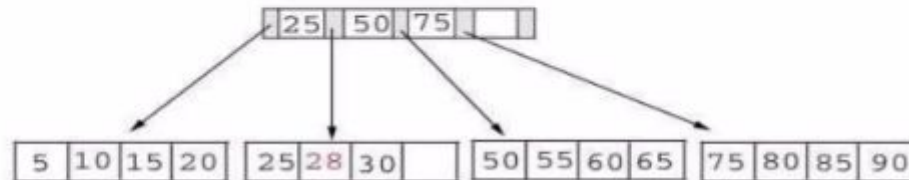
1. A search is first performed, using the value to be added.
2. After the search is completed, the location for the new value is known.
3. If the tree is empty, add to the root.
4. Once the root is full, split the data into 2 leaves, using the root to hold keys and pointers.
5. If adding an element will overload a leaf, take the median and split it.

Insertion Example :-

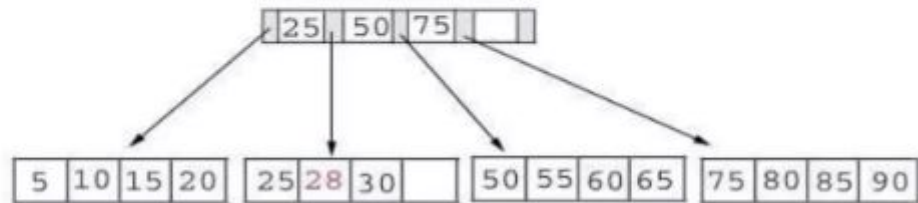
- Example #1: insert 28 into the below tree.



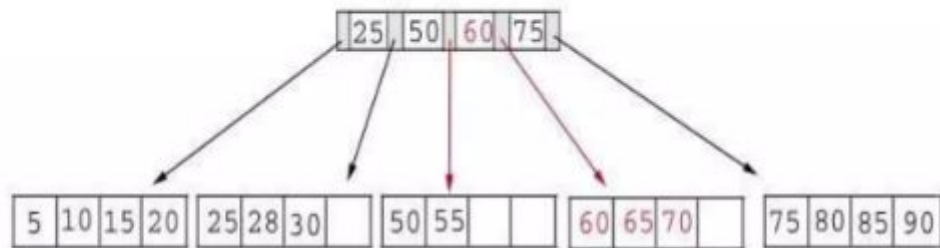
Result:-



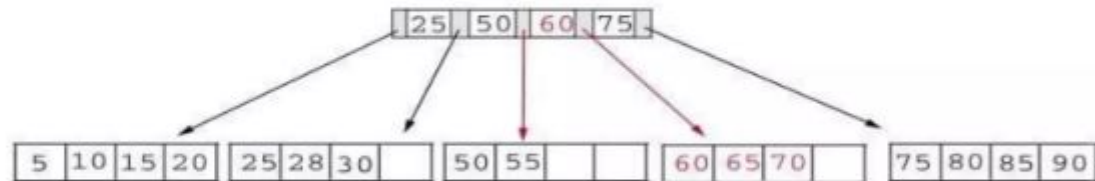
- Example #2: insert 70 into below tree



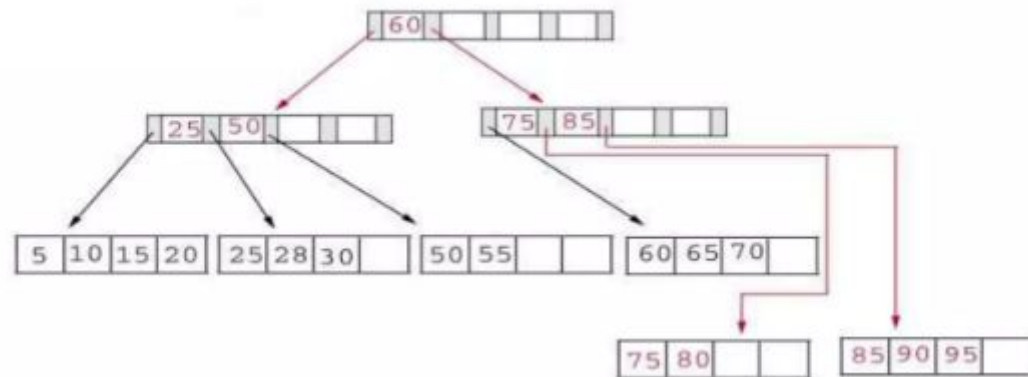
Result:-



- Example #3: insert 95 into below tree



Result:-



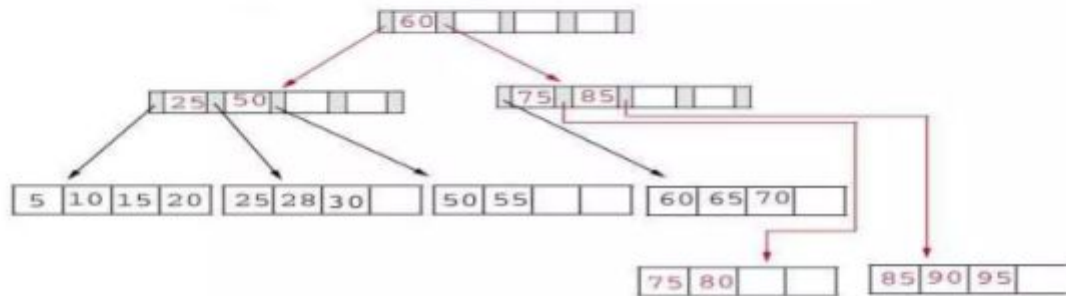
Notice that root was split, leading to increase in height.

Deletion:-

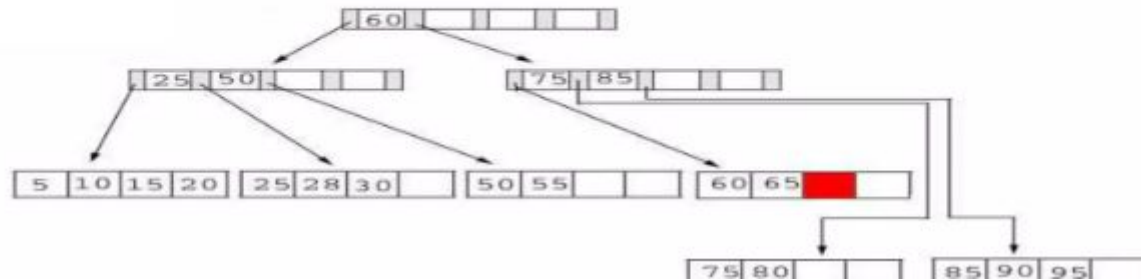
1. Deletion, like insertion, begins with a search.
2. When the item to be deleted is located and removed, the tree must be checked to make sure no rules are violated.
3. The rule to focus on is to ensure that each node has at least $\lceil n/2 \rceil$ pointers.

Deletion Example :-

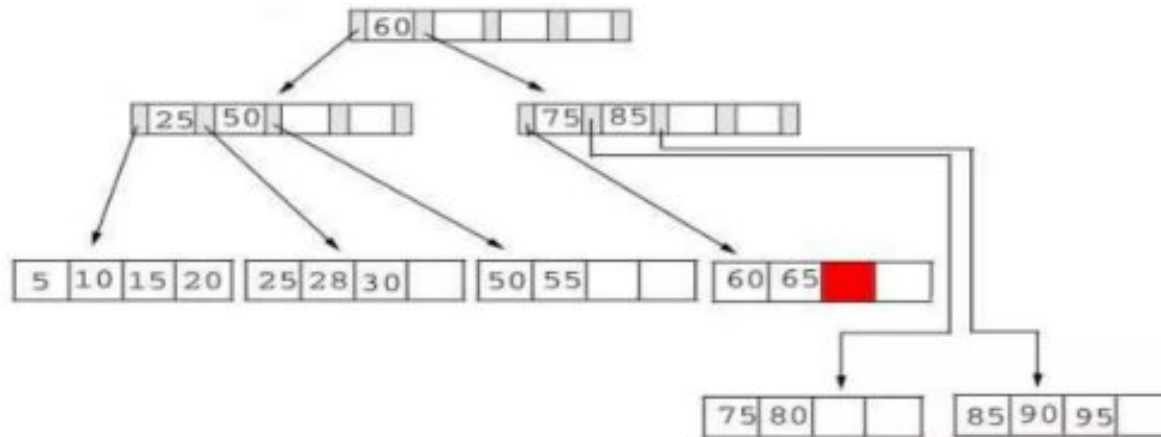
- Example #1: delete 70 from the tree



Result:-

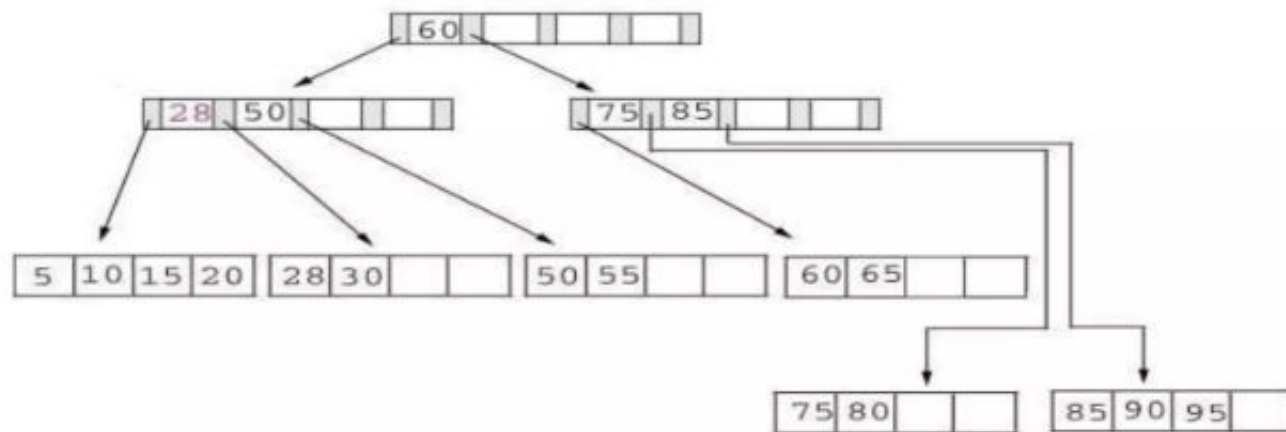


- Example #2: delete 25 from below tree

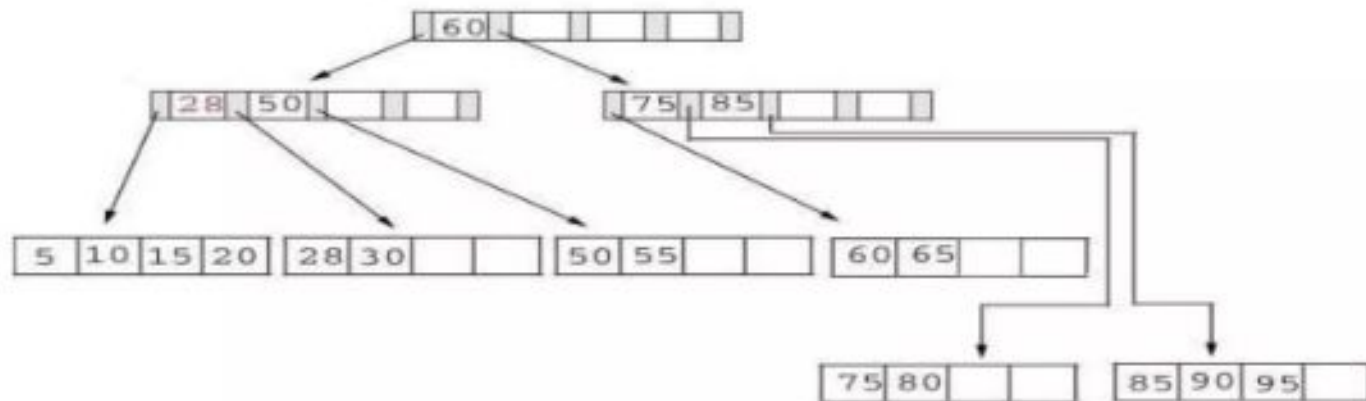


Result:-

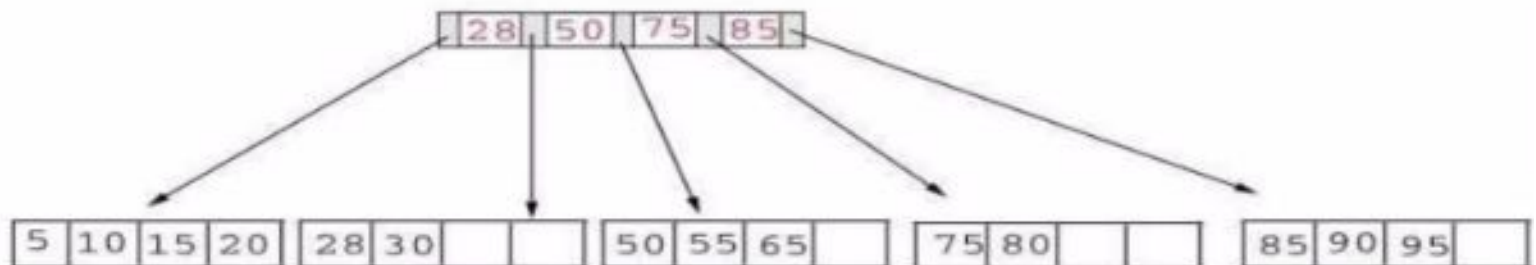
Replace 25 with 28 in the index page.



- Example #3: delete 60 from the below tree



Result:-



Differences between B-Tree and B+ Tree

B-Tree	B+ Tree
All internal nodes and leaf nodes contain data pointers along with keys.	Only leaf nodes contain data pointers along with keys, internal nodes contain keys only.
There are no duplicate keys.	Duplicate keys are present in this, all internal nodes are also present at leaves.
Leaf nodes are not linked to each other.	Leaf nodes are linked to each other.
Sequential access of nodes is not possible.	All nodes are present at leaves, so sequential access is possible just like a linked list.
Searching for a key is slower.	Searching is faster.

Hash Based Indexing

Hashing in DBMS

In a huge database structure, it is very inefficient to search all the index values and reach the desired data.

Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

In this technique, data is stored at the data blocks whose address is generated by using the hashing function.

The memory location where these records are stored is known as data bucket or data blocks.

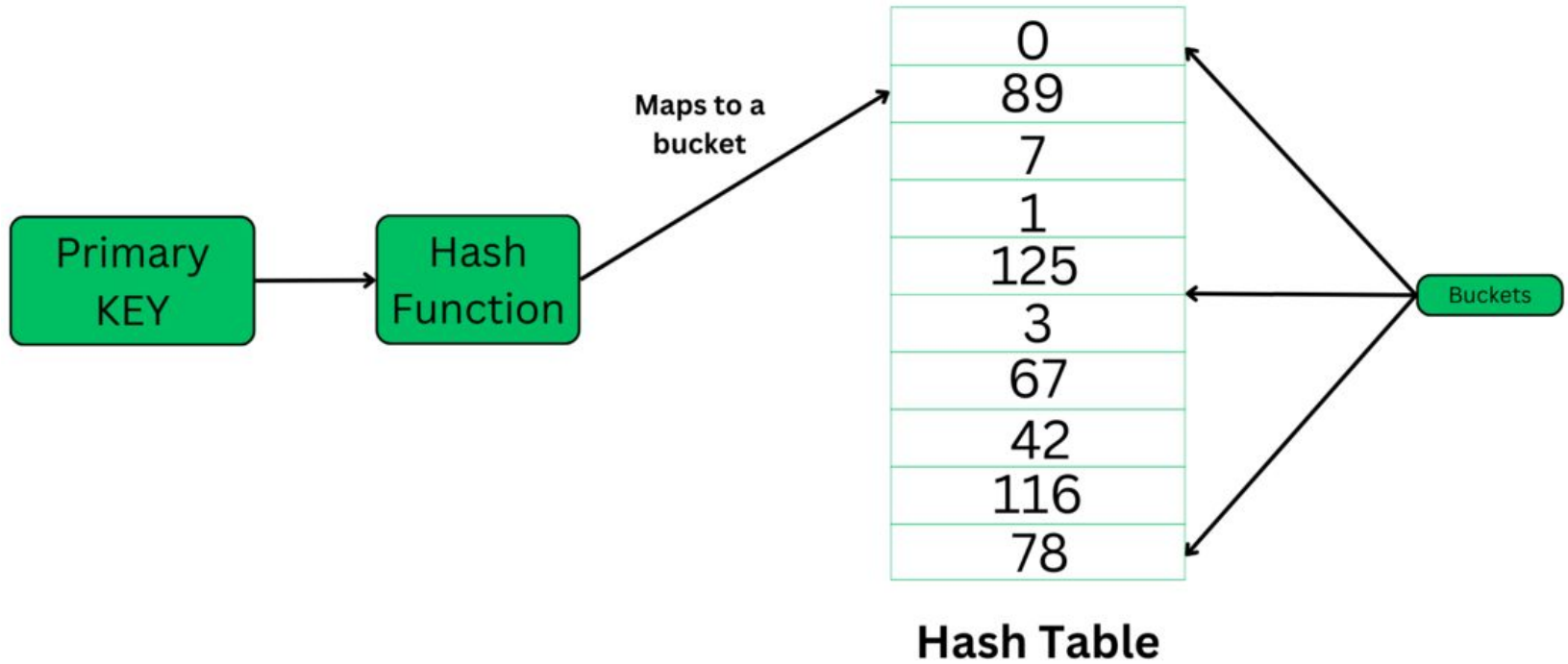
A hash function can choose any of the column value to generate the address.

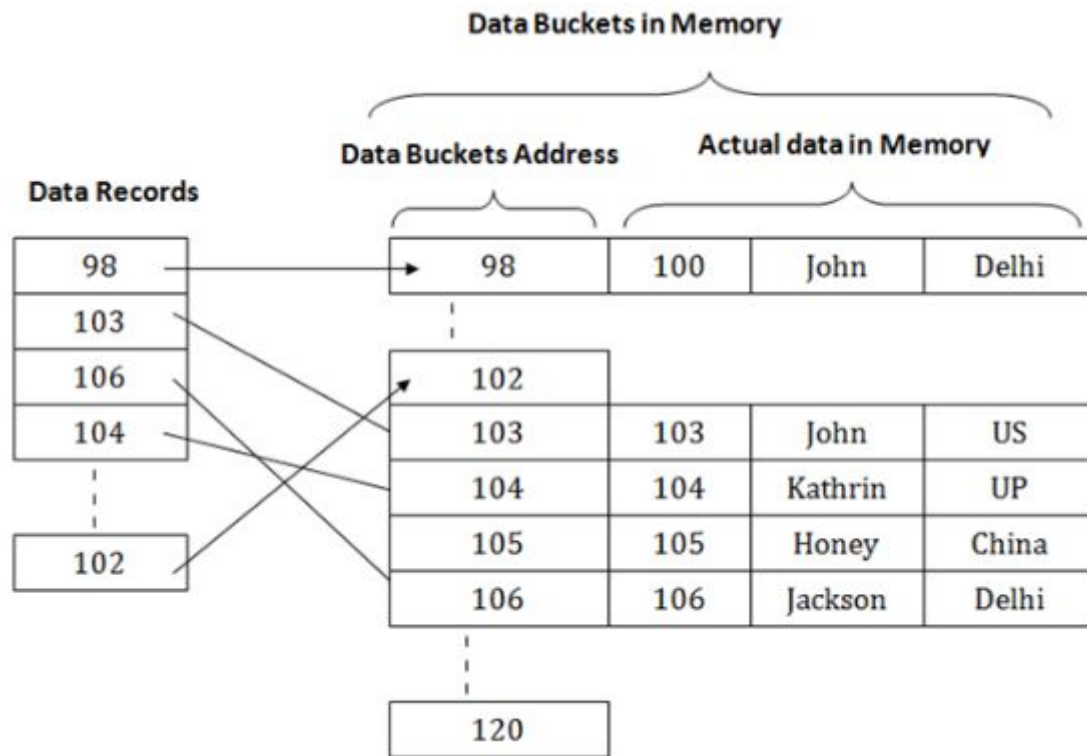
The hash function uses the primary key to generate the address of the data block. A hash function is a simple mathematical function to any complex mathematical function.

The primary key itself as the address of the data block.

That means each row whose address will be the same as a primary key stored in the data block.

Hashing

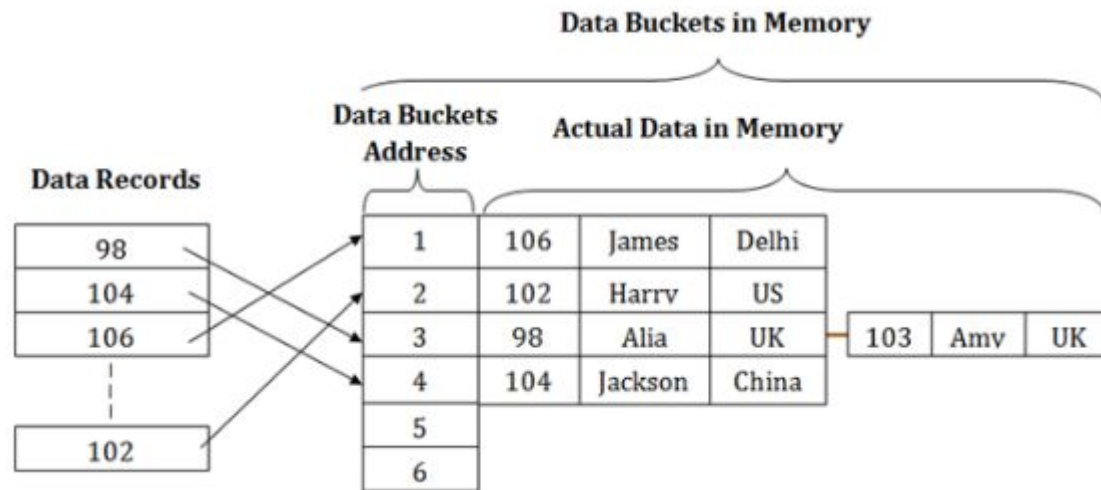




This hash function can also be a simple mathematical function like exponential, mod, cos, sin, etc.

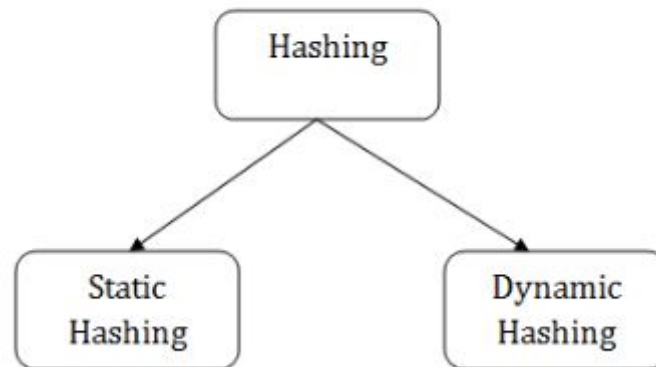
Suppose we have mod (5) hash function to determine the address of the data block.

In this case, it applies mod (5) hash function on the primary keys and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses.



Types of Hashing

Types of Hashing:



Static Hashing

In static hashing, the resultant data bucket address will always be the same.

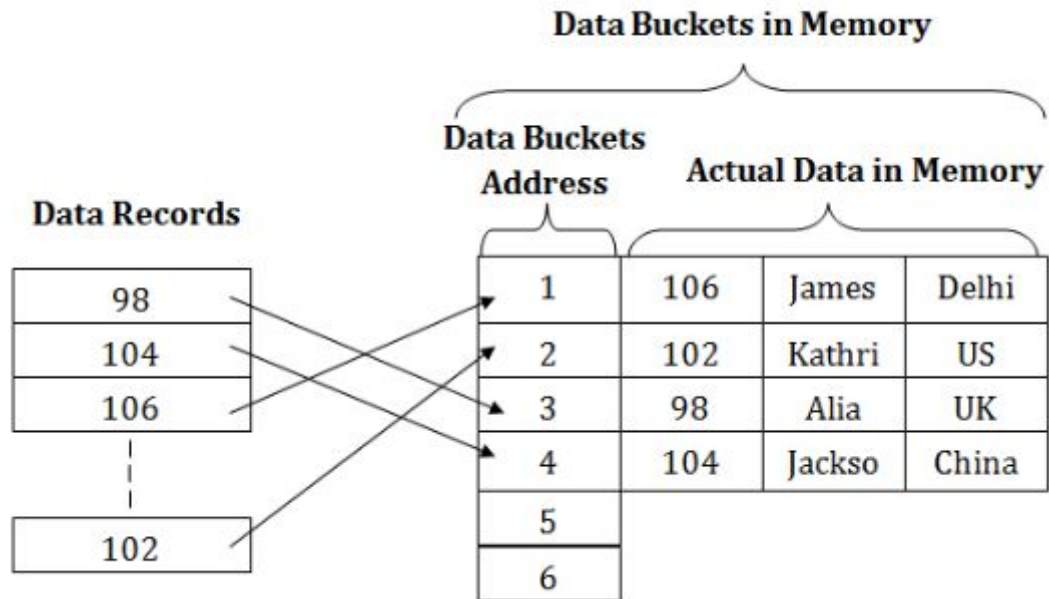
That means if we generate an address for EMP_ID =103 using the hash function $\text{mod } (5)$ then it will always result in same bucket address 3.

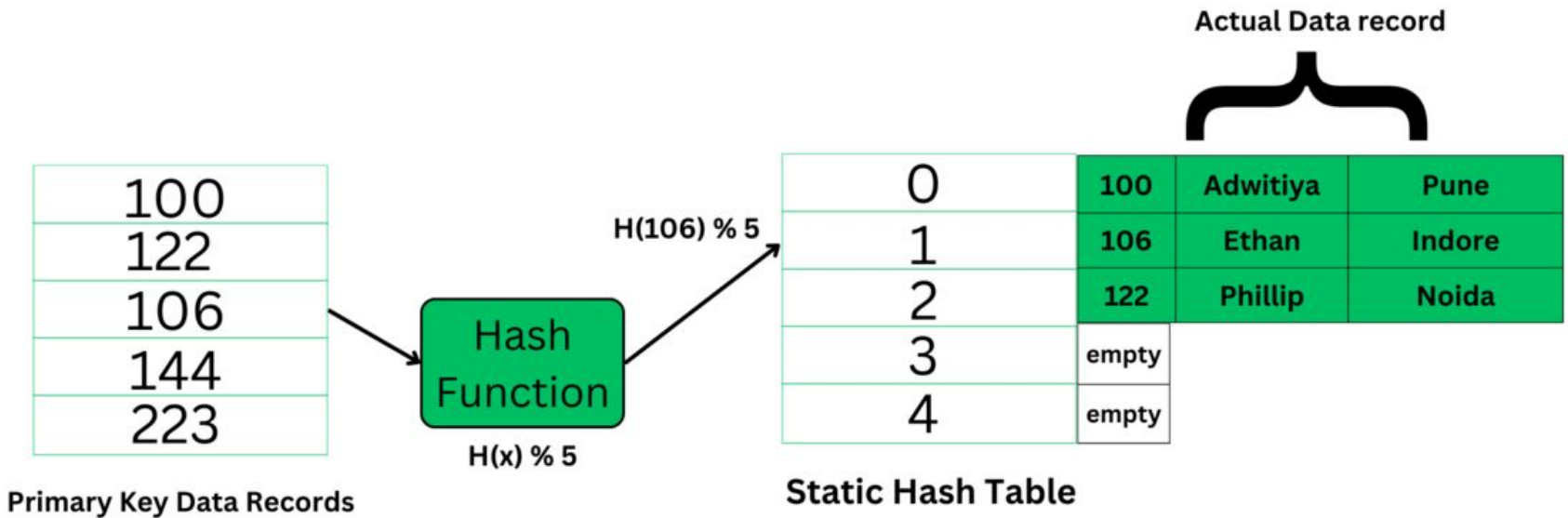
Here, there will be no change in the bucket address.

Hence in this static hashing, the number of data buckets in memory remains constant throughout.

In this example, we will have five data buckets in the memory used to store the data.

Static Hashing





Operations of Static Hashing

Operations of Static Hashing

Searching a record

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

Insert a Record

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

Delete a Record

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

Update a Record

To update a record, we will first search it using a hash function, and then the data record is updated.

Operations of Static Hashing

Operations of Static Hashing

If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address.

This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.

To overcome this situation, there are various methods. Some commonly used methods are as follows:

Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.

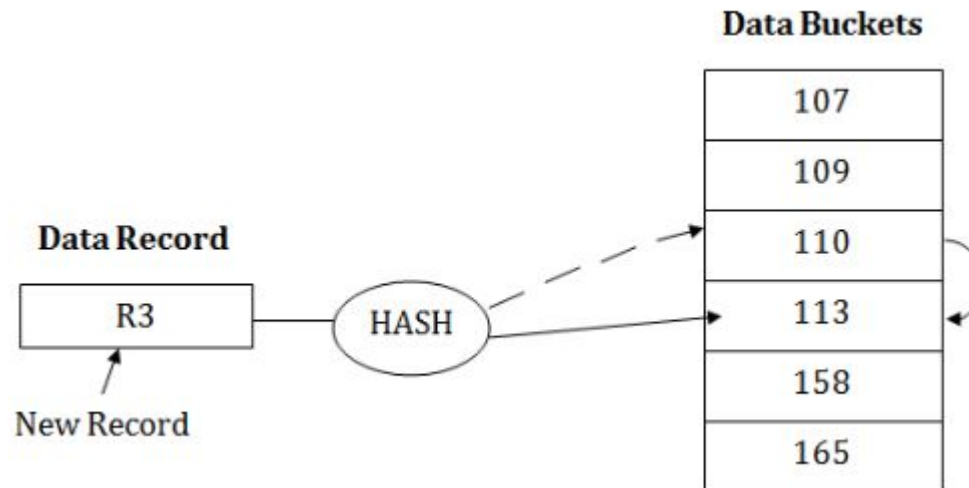
For example: suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.

Operations of Static Hashing

Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.

For example: suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.

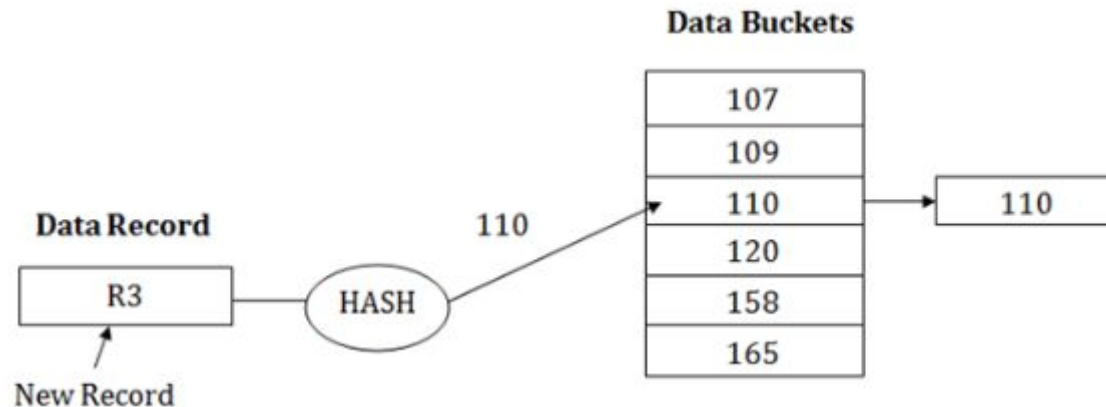


Operations of Static Hashing

Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.

For example: Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



Dynamic hashing

Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i .
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

Dynamic hashing

Dynamic Hashing

How to insert a new record

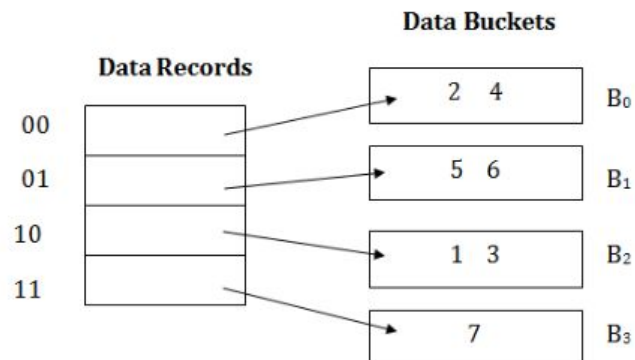
- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

The last two bits of 2 and 4 are 00. So it will go into bucket B₀. The last two bits of 5 and 6 are 01, so it will go into bucket B₁. The last two bits of 1 and 3 are 10, so it will go into bucket B₂. The last two bits of 7 are 11, so it will go into B₃.



Data Records

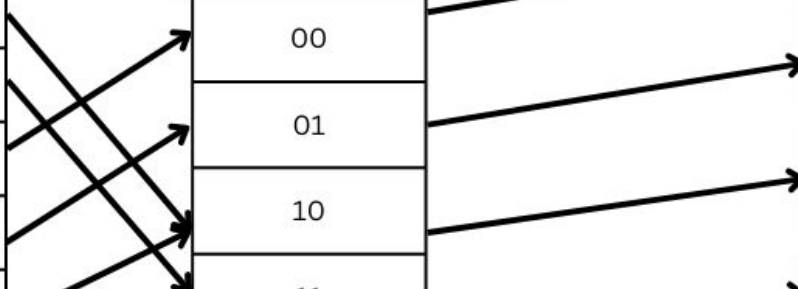
Key	Hash Index
102	11001 10
107	11010 11
64	10000 00
109	11011 01
110	11011 10

Directories

Directory IDs
00
01
10
11

Buckets

64	B ₀
109	B ₁
102, 110	B ₂
107	B ₃



Insert

Insert key 9 with hash address 10001 into the above structure:

- Since key 9 has hash address 10001, it must go into the first bucket. But bucket B₁ is full, so it will get split.
- The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B₁, and the last three bits of 6 are 101, so it will go into bucket B₅.
- Keys 2 and 4 are still in B₀. The record in B₀ pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- Keys 1 and 3 are still in B₂. The record in B₂ pointed by the 010 and 110 entry because last two bits of both the entry are 10.
- Key 7 are still in B₃. The record in B₃ pointed by the 101 and 011 entry because last two bits of both the entry are 11.

