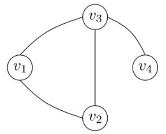
① Network Layer

1. Graphs

→ **Graph:** Pair (V,E), where V is a set of nodes and $E \subseteq V \times V$ is a set of edges between nodes



	v_1	v_2	v_3	v_4
v_1	0	1	1	0
v_2	1	0	1	0
v_3	1	1	0	1
v_4	0	0	1	0

→ adjacency matrix

Figure 2.2: A graph $G = (V, E)$ with node set $V = \{v_1, v_2, v_3, v_4\}$ and edge set $E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}\}$, and the adjacency matrix of G.

edge $e = \{v, u\} \in E$:
⇒ e and v,u are incident
⇒ v and u are adjacent

→ **weighted graph:** Assign a weight $\omega(e): E \to \mathbb{R}$ to each edge. Weight of a graph is $\sum_{\forall e \in E} \omega(e)$

→ **directed graph:** Distinguisch between (u,v) and (v,u)

→ **Path:** Path between nodes v_1 and v_k is a sequence of nodes $(v_1, v_2, \dots, v_k)$

→ **Connected graph:** There exists a path between any two nodes u,v (no „isolated" parts)

→ **Cycle:** Loop → Sequence $(v_1, v_2, \dots, v_k, v_1)$ such that $\{v_k, v_1\}, \{v_i, v_{i+1}\} \in E$ for all $1 \leq i \leq k$ and no node appears twice

→ **Tree:** Connected graph with no cycles

→ **Subgraph:** Graph (V', E') such that $V' \subseteq V$ and $E' \subseteq E$

→ **Spanning tree:** Given graph (V,E), spanning tree is a subgraph $T(V,E')$ that is a tree.

→ **MST:** find spanning tree that minimizes total weight

Algorithm 2.11 MST Algorithm

```
Given a weighted graph G = (V, E, ω)
Let S = {u} be a set of visited nodes, initialized with any node u ∈ V
Let T be a tree just consisting of the single node u ∈ S, no edges
while S ≠ V do
    Find minimum weight edge e = {v, w} with v ∈ S and w ∈ V \ S
    Add node w to S
    Add edge e to T
end while
```

$\mathcal{O}(m \log(n))$

$m = \#$ edges
$n = \#$ nodes

→ **Shortest Path:** Path between u and v with minimum total weight

→ **Distance:** total weight of shortest path → $d(u,v)$

→ **SPT:**

Algorithm 2.16 SPT Algorithm

```
Given a weighted graph G = (V, E, ω) and a node r ∈ V
Set a parent node p_v = null for every node v ∈ V
Set d_r = 0 and d_v = ∞ for every node r ≠ v ∈ V
Let S = {r} be the set of visited nodes
while S ≠ V do
    Find edge e = {v, w} with v ∈ S and w ∈ V \ S with minimum d_v + ω(e)
    Set p_w = v
    Set d_w = d_v + ω(e)
    S = S ∪ {w}
end while
```

$\mathcal{O}(m \log(n))$

$m = \#$ edges
$n = \#$ nodes

2. Addressing

→ **IPv4:** 32 bit address. 4 chunks of 8 bits (decimal)

Example: 173. 55. 17. 69 (173 = 8 bits)

• Prefix: first k bits of the address

• Block: Set of addresses that share a common prefix

172.16.0.0 to 172.31.255.255 ~~form~~ form a block of 172.16.0.0/12 (12 = prefix)

• Problem: not enough different IPv4 addresses (only 2^{32})
↳ Solution IPv6

→ **IPv6:** 128 bit addresses. 8 chunks of 16 bits (separated by „:") (hexadecimal)

Example: 6666 : db8 :: ff00 : 0 : 42 (6666 = 16 bits; :: ↳ 3 chunks with only zeros)

ab.cd.ef.gh $\xrightarrow{\text{IPv4 to IPv6}}$ :: ffff : abcd : efgh

→ The : in the IPv6 address can only appear once

- ::6666:3f1a:0 is a valid IPv6 address ✗
- ::6666::3f1a:0 is <u>not</u> a valid IPv6 address ✓

→ Some IP-addresses have special meaning

Example: 127.0.0.1 = localhost → IP-address that points to the device itself

3. Packets

Header	Payload

- Header
  - Source and destination
  - Version (IPv4 or IPv6)
  - Size of header
  - Size of payload
  - TTL = Time-to-live → decreases by one every time it goes through a node. When it reaches zero the router just „drops" the packet
- Payload
  - actual data

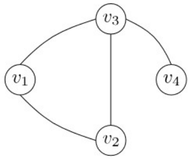
4. Routing

→ The task of a routing protocol is to decide along which path(s) a packet travels from its source to its destination.

→ **Routing table:** Maps every destination address to a neighbor of v.

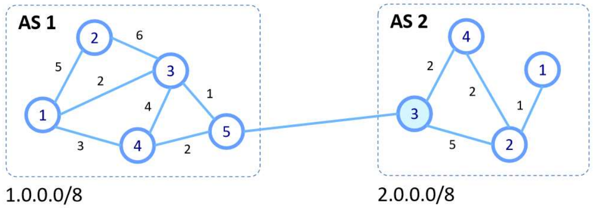
→ **Forwarding:** Process of an intermediate node receiving a packet and sending it to the next node

Routing table of v_1	
Destination	Next node
v_1	deliver
v_2	v_2
v_3	v_3
v_4	v_3



→ Problem: It is impossible to save all the possible paths from the whole internet

→ **Autonomous System (AS):** Collection of nodes (owned by one company) where every node has knowledge of the whole topology of the graph.



→ **Link-State (LS) Routing Algorithm:** Algorithm to find the routing table assuming the node has knowledge of the whole topology

- only for small networks (know the whole topology!)
- For AS, intra-domain

Algorithm 1.26 Link-State (LS) Routing Algorithm.

1: Given a weighted graph $G = (V, E, \omega)$
2: Learn $\omega(e)$ for every edge $e \in E$
3: Compute shortest paths to between all nodes, e.g., by using Algorithm 1.16

→ **Distance-Vector (DV) Routing Algorithm:** Algorithm to find the routing table (a simplified one) in case we don't know the whole topology

- DV routing is distributed ⟹ nodes do not need to acquire the knowledge of the whole network topology to perform routing
- For large networks outside/between ASs, inter-domain

Algorithm 2.27 Distance-Vector (DV) Routing Algorithm.

1: Given a weighted graph $G = (V, E, \omega)$ and a node $u \in V$
2: Initialize a distance estimate $D(u \to v) = \omega(\{u, v\})$ for all neighbors $N(u)$ and $D(u \to w) = \infty$ for all other nodes
3: Send distance vector $\mathcal{D}(u) = \{D(u \to v) \mid v \in N(u)\}$ to all neighbors $N(u)$
4: **while** true **do**
5: Upon receiving a distance vector $\mathcal{D}(v)$ from a neighbor v, update the distance estimate to all destinations accordingly
6: **if** $D(u \to w)$ changed for any w **then**
7: Send the updated distance vector $\mathcal{D}(u)$ to all neighbors
8: **end if**
9: **end while**

<u>Border Gateway Protocol (BGP)</u>
↓
DV routing protocol (for inter-domain)

→ **Intra-Domain:** Routing inside an AS → LS

→ **Inter-Domain:** Routing between one or more ASs → DV (BGP)

② Transport Layer

1. Flows

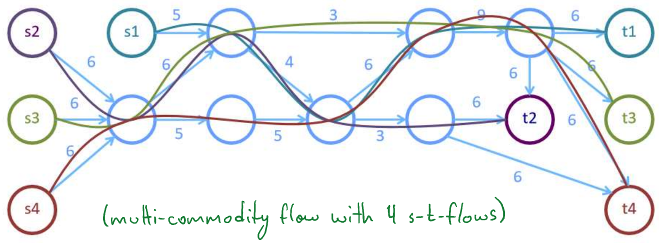
→ **Flow, Rate:** Let s, t be two nodes. A flow from source (s) to destination (t) [s-t-flow] is a function $F: E \to \mathbb{R}_{\geq 0}$ such that

i. $F(e) \leq c(e) \quad \forall e \in E$
↓ don't exceed capacity

ii. $\sum_{e \in in(v)} F(e) = \sum_{e \in out(v)} F(e) \quad \forall v \in V \{s,t\}$
↓ what goes in goes out (not true for s,t)

→ Weights of the edges are the respective capacities

→ Multi-Commodity Flow: $\mathcal{F} = (F_1, F_2, \dots, F_k)$ is a collection of $s_i - t_i$-flows F_i such that for each edge $e \in E$ the sum of the flows' rates on e does not exceed the capacity of e



(multi-commodity flow with 4 s-t-flows)

2. Linear-Programming (LP)

→ Linear-Programming: Tool for optimization problems

→ Consists of m inequalities and a linear function. We are looking for the $x = (x_1, x_2, \dots, x_n)^T$ that maximizes f respecting the restrictions

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &\leq b_2 \\ \vdots \quad\quad \vdots \quad\quad\quad \vdots \quad & \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &\leq b_m \end{aligned}$$

m inequalities

$$f(\mathbf{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n .$$

linear function

- $ax \geq b \longrightarrow -ax \leq -b$
- $ax = b \longrightarrow \begin{cases} ax \leq b \\ -ax \leq -b \end{cases}$

Algorithm 3.6 Simplex Algorithm

```
choose a vertex x of the polytope
while there is a neighboring vertex y such that f(y) > f(x) do
    x := y
end while
return x
```

→ good for discrete values

→ LP for s-t-flows: Maximize $f(x) = \sum_{e \in out(s)} x_e$

1. $x_e \geq 0 \quad \forall e \in E$
2. $x_e \leq c(e) \quad \forall e \in E$
3. $\sum_{e \in in(s)} x_e = 0$
4. $\sum_{e \in in(v)} x_e = \sum_{e \in out(v)} x_e \quad \forall v \in V \setminus \{s,t\}$

→ **Unsplittable flow:** flow forms a sing s-t-path. If we do not impose this path restriction on a flow, we call it splittable

3. Fairness

→ **Demand:** The demand $d_i \in \mathbb{R}_{\geq 0}$ of a flow F_i is the rate at which F_i wants to transmit

- Since in a multi-commodity flow we have multiple s-t-flows flowing through the same edge, the demand is not always reached

→ **Max-Min-Fairness:** A bandwidth allocation is called max-min-fair if increasing the allocation of a flow would necessarily decrease the allocation of a smaller or equal-sized flow.
" = we basically reached the maximum flow for all s-t-flows"

Algorithm 3.12 Max-Min-Fair Allocation

```
Given a graph G, a set 𝓕 = {F_1, ..., F_k} of flows with initial rate 0 on all
  edges, paths p_1, ..., p_k along which the respective flows are to be routed and
  demands d_1, ..., d_k
while 𝓕 ≠ ∅ do
  repeat
    increase rate of all flows in 𝓕 evenly, but at most up to the respective
    demands
  until there is an edge e ∈ E such that Σ_{i:e∈p_i} F_i = c(e)
  for all such edges e do
    for all i such that e ∈ p_i do
      𝓕 := 𝓕 \ {F_i}
    end for
    E := E \ {e}
  end for
end while
```

1. Increase bandwidth of all s-t-flows by one
2. Increase until some edge reaches its maximum rate
3. Stop increasing (lock) bandwidth of s-t-flows going through that edge.
4. Continue increasing bandwidth of other s-t-flows until all of them are locked

assuming the s-t-flows' path does not change!

→ Congestion control: Trying to find Max-Min-Fairness in large networks is very hard. An alternative is AIMD (additive increase multiplicative decrease)

→ AIMD (additive increase / multiplicative decrease): Type of congestion control

- no congestion ⇒ additive increase on flow rate
- congestion ⇒ multiplicative on flow rate (like $\times \frac{1}{2}$)

- Sawtooth behavior
- Congestion happens on the nodes (routers) and not edges
- Congestion ⇒ routers drop packets so receivers know that they should decrease rate (multiplicative decrease)
- For omnipresent distributed transport protocol like TCP

4. UDP - User Datagram Protocol

→ Allows the transport of packets from client to server

- UDP does not include any protection against packet loss.
- UDP does not guarantee any order on the delivery of packets

⇒ no type of „connection" between client and server

→ Used for applications where a fast information exchange is needed (like Skype etc) → real-time applications

→ Commonly used for DNS servers

5. TCP - Transmission Control Protocol

→ Connection oriented (connection ⇒ bidirectional)

- Guarantees right order of packets
- Handles packet loss

→ Segments: TCP packets

→ Acknowledgement (ACK): Confirmation of the arrival of the packet.

- ACK also sends the number of the next expected segment

→ Round-Trip Time (RTT): Time between sending the packet and receiving the ACK.

→ Congestion control

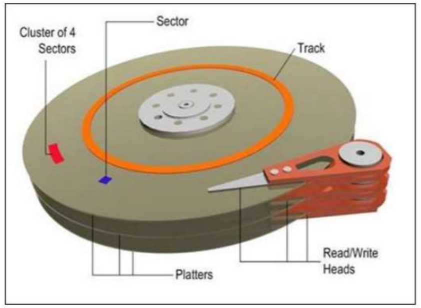
i. Slow start: Flow rate increases exponentially (like $\times 2$) on each successfull transmission until threshold is reached

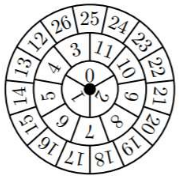
ii. AIMD

6 Storage and File Systems

→ A storage device consists of n pages (also known as sectors or blocks) of fixed size, e.g. 512 bytes per page. The address space of the device is 0 to n-1. To write to or read from a storage device, the OS specifies the address(es) of the page(s) it wants to access.

1. HDD - Hard Disk Drive





→ Inner tracks have obviously less pages

→ I/O time: looking for page S

$$T_{I/O} = \underbrace{T_{seek} + T_{rotation}}_{T_{positioning}} + T_{transfer}$$

T_{seek} = time it takes to move the read/write head to the track on which S lies

$T_{rotation}$ = time it takes the platter to rotate to S

$T_{transfer}$ = time it takes to transfer the data to/from S.

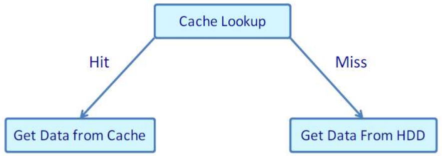
$T_{positioning} = T_{seek} + T_{rotation}$ = time it takes to move the read/write head to the beginning of S

→ Rate of I/O: Rate at which data is transfered

$$R_{I/O} = \frac{\text{size of } A}{T_{I/O}}$$

→ A is the requested size (how much data to transfer)

→ If there are caches, HDD is only accessed if there is a cache miss



2. Disk Scheduling

Algorithm	Description
First Come First Serve (FCFS)	Process requests in the order they arrived.
Shortest Seek Time First (SSTF)	Pick request on nearest track. → nearest Track
Shortest Positioning Time First (SPTF)	Pick request with shortest positioning time.
Elevator (SCAN)	Move the head like an elevator, inside to outside and back again, and service all pending requests on the way.
C-SCAN	Similar to SCAN; starting from the current head position, service requests in ascending order towards the outermost track, then move head without servicing any requests to the now-innermost request.
F-SCAN	Like SCAN, but service requests in batches; wait with sending a new batch of requests to disk until the last one was fully serviced.

→ FCFS, SSTF, SPTF → starvation if there are always new requests on the same track.

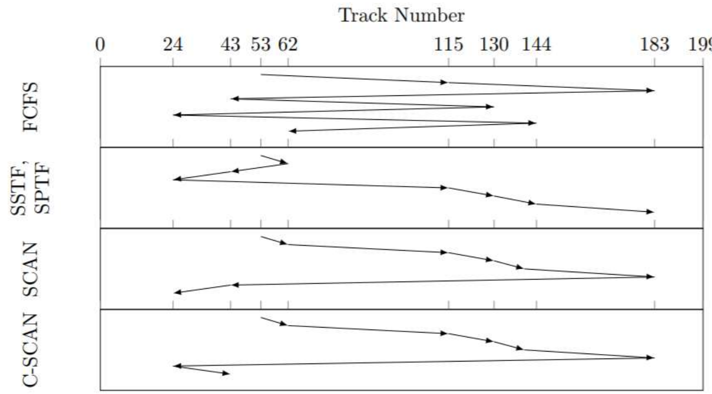
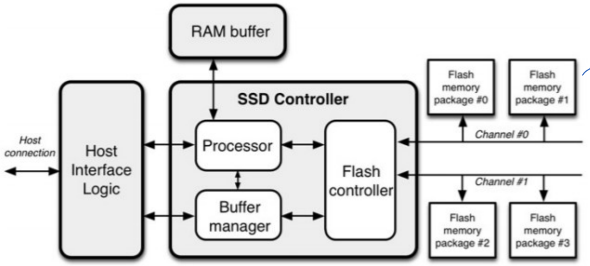


Figure 6.9: Head movements for different scheduling algorithms. The head starts at track 53 in each example run, and the sequence of requests sent to the disk is for pages on tracks 115, 183, 43, 130, 24, 140, 62.

3. SSD - Solid State Drive

→ Flash memory: Electronic storage medium without moving parts



→ transistors/capacitors

→ Organized in blocks and pages

→ Three data states
i) Invalid
ii) Erased
iii) Valid

→ only erased state pages can have new data written to it

→ If a page wants to be deleted, the whole block is deleted

the SSD has to copy valid data to a new block (or the now empty block) ⇒ Internal write request (OS ⇒ external write request)

→ If we want to update a value, instead of overwriting the page p, we write it to a new page p' and set p to invalid.

Block	0				1			
Page	0	1	2	3	4	5	6	7
Content	a'			a				
State	v	e	e	i	e	e	e	e

4. FTL- Flash Transition Layer

→ The addresses in access requests sent by the OS to the SSD are called *logical addresses*. When a write request for a logical address arrives at the SSD, the SSD chooses a *physical address* where the data will be written. The SSD stores the mapping from logical to physical addresses, and this mapping is called the *flash transition layer* (FTL)

→ No FTL ⇒ logical address = physical address

⇒ Direct mapping

Example 6.15 (Page Level FTL). *We give an example of a page level FTL. Initially, the mapping is empty, and all pages are erased. For simplicity, every write goes to the next available erased page, and we erase round-robin. The rows "Table" and "State" describe the mapping and validity information stored in the FTL, respectively.*

Table												
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
Content	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
State	e	e	e	e	e	e	e	e	e	e	e	e

First we write logical pages 0 to 4.

Table	$0 \to 0, \ldots, 4 \to 4$											
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
Content	0	1	2	3	4	⊥	⊥	⊥	⊥	⊥	⊥	⊥
State	v	v	v	v	v	e	e	e	e	e	e	e

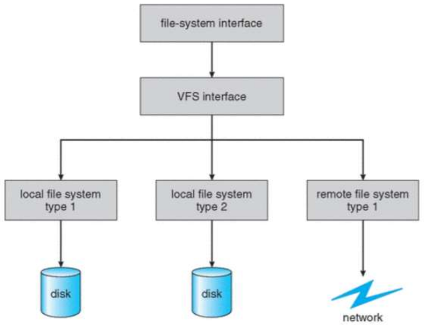
Next we write (update) logical pages 2 to 8.

Table	$0 \to 0, 1 \to 1, 2 \to 5, \ldots, 8 \to 11$											
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
Content	0	1	2	3	4	2	3	4	5	6	7	8
State	v	v	i	i	i	v	v	v	v	v	v	v

Now we write logical page 6 (erase block 0, write back 0 and 1).

Table	$0 \to 1, 1 \to 2, 2 \to 5, 3 \to 6, 4 \to 7, 5 \to 8, 6 \to 0, 7 \to 10, 8 \to 11$											
Block	0				1				2			
Page	0	1	2	3	4	5	6	7	8	9	10	11
Content	6	0	1	⊥	4	2	3	4	5	6	7	8
State	v	v	v	e	i	v	v	v	v	i	v	v

5. Logical File System



→ The arrangement of the data on a storage device is the *physical file system*. The data structures maintained in memory by the OS to manage data is the *logical file system*

→ The software connecting those two is called the *virtual file system* (VFS)

installable file system (Win) (Linux)

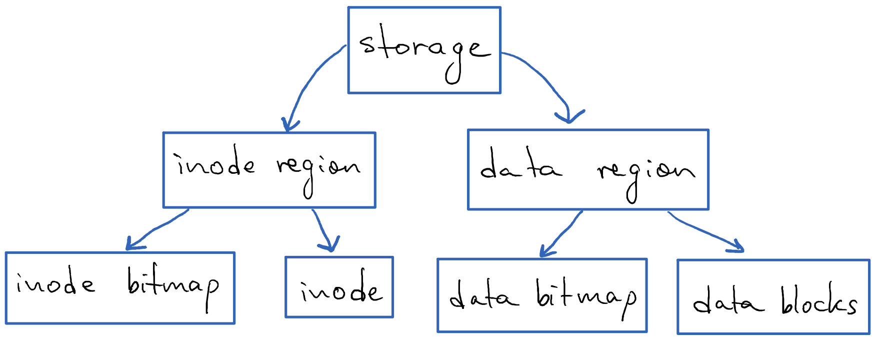
addresses, and this mapping is called the flash transition layer (FTL)

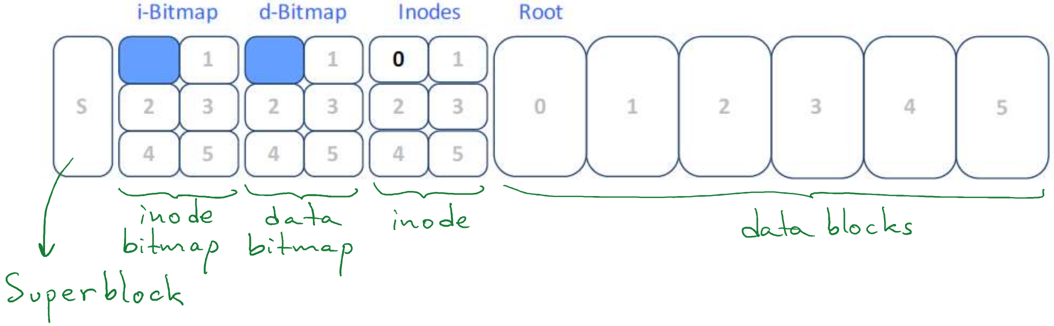
installable file system (Win) (Linux)

→ **Blocks, Inodes, Pointers**: The physical file system groups multiple storage device pages into a <u>block</u> (sometimes <u>clusters</u> or <u>allocation units</u>).
Every file in the physical file system is represented uniquely by an <u>inode</u>.
The data of the file is referenced via direct, single indirect, double indirect, or triple indirect <u>pointers</u> that encode on which blocks the data is.

→ An inode contains metadata about the file, such as type, owner, permissions etc.

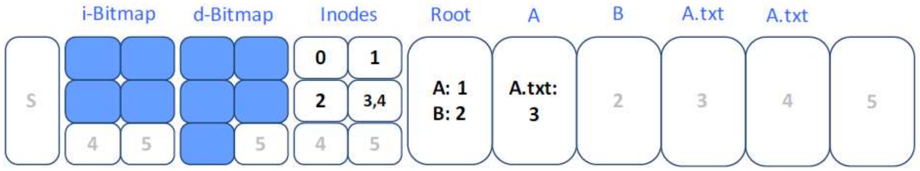
→ small files can be referenced via direct pointers. Large files need sometimes indirect pointers (up to triple undirect pointers). Very small files can be saved directly in the inode





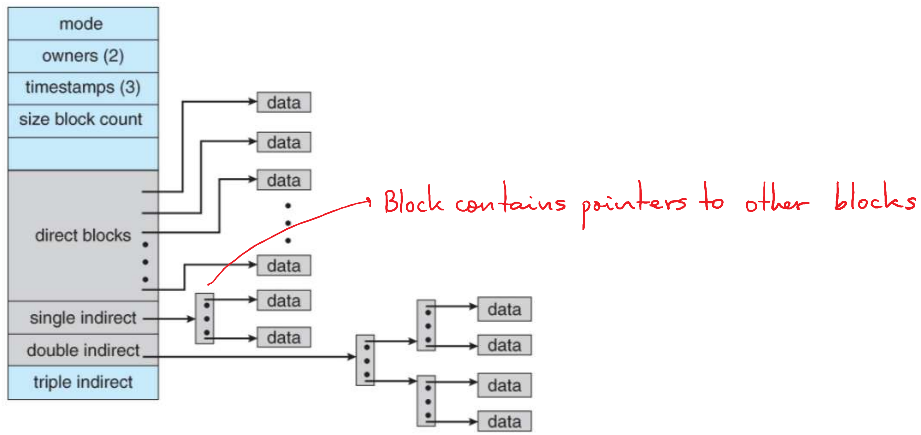
• Superblock: gives information about the file system (FAT, NTFS, ext4 etc), where to find inode etc.

• Bitmaps: Are either set to 0 or 1 depending whether the corresponding inode/data block has valid information/data.



↳ data blocks 0,1,2,3,4 have valid information
↳ inodes 0,1,2,3 are active (have valid information)

→ Structure of an inode:



→ Types of links:

i) **Hard Link**: Entry in a directory that refers to an inode, i.e. a key-value pair (name: inode number) in the directory data. Thus, the same inode (the same file!) can be accessed via multiple hard links

⇒ different directories have files that point to the same inode

ii) **Soft Link**: Also called <u>symbolic link</u> or <u>symlink</u> is a file with its own inode whose only content is the absolute path of the file it points to, the <u>target</u>

→ One can not tell which hard link is the „real" one

→ None of these two links make copies of the file. They are just two different methods to <u>point</u> to a file

→ We can only delete a file if there exists no additional hard link that points to it.

8 Dictionaries and Hashing

1. Binary Search Tree

→ Left child always < right child

→ Normal case: $\mathcal{O}(\log(n))$
Worst case: $\mathcal{O}(n)$

2. Hashing

→ Keys: Identification for an object

→ Universe (U): Set of all possible keys

→ Key Set (N): Set of relevant keys ($N \subset U$) for the problem

→ Hash Table (M): Array where keys are saved

→ Bucket M[i]: One single entry of the array M

→ Hash Function h(k): Gives the location on the array for a given key

Example: $h(k) = k \bmod (m)$

→ Collision: When for $\ell \neq k$ we get $h(\ell) = h(k)$

→ Load factor α: $\alpha := \frac{n}{m}$

- In java the array is doubled if $\alpha \geq 0.75$
- Problem:
  - → large m ⇒ too much storage needed
  - → small m ⇒ more collisions

$P[\text{no collision}] \leq e^{\frac{-n(m-1)}{m}}$ → for $n = \sqrt{m} \Rightarrow P \to 1$

→ Universal Family: When for a collection of Hash Functions

$Pr[h(k) = h(\ell)] \leq \frac{1}{m}$ for distinct keys k and ℓ

↓ probability of collision

- We expect a good distribution of the keys when choosing a hash function

---

Algorithm 8.12 Perfect Static Hashing

Input : fixed set of keys N
Output : Primary hash table M and secondary hash tables M_i
Function: $N_i := \{k \in N : h(k) = i\}$ → Set of keys with same location (bucket)
Function: $n_i := |N_i|$ → Size of Set N_i (how many keys)

```
M := hash table with n buckets
repeat
    h := hash function N → M (sampled from universal family)
until C(h, N) < n
for i ∈ M do
    M_i := hash table with 2(n_i choose 2) = n_i(n_i − 1) buckets
    repeat
        h_i := hash function N_i → M_i (sampled from universal family)
    until C(h_i, N_i) < 1
end for
return (M, h, (M_i)_{i∈[m]}, (h_i)_{i∈[m]})
```

} Find h with no more than n collisions (lines: repeat … until $C(h, N) < n$)

↳ Size of M together with $\forall M_i$ is $< 3n$

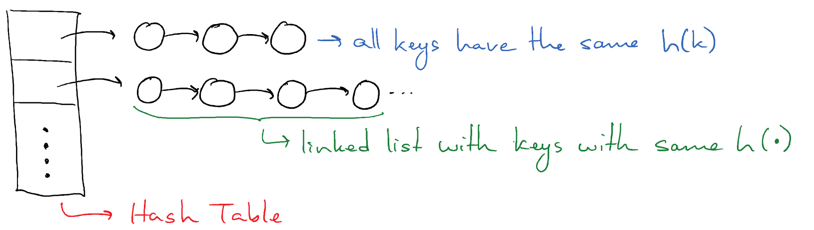
→ Hashing with probing

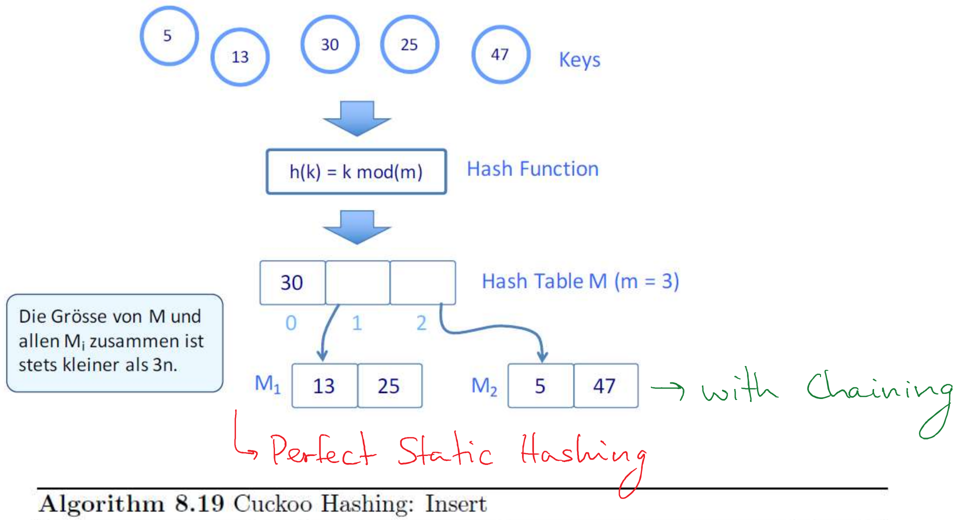
- Collision ⇒ find another place

Type	$h_i(k)$	≈ cost successful	≈ cost unsuccessful
Linear probing	$h(k) + i$	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$
Quadratic probing	$h(k) + i^2$	$\frac{1}{1-\alpha} + \ln\frac{1}{1-\alpha} - \alpha$	$1 + \ln\frac{1}{1-\alpha} - \frac{\alpha}{2}$
Double hashing	$h_1(k) + i \cdot h_2(k)$	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$

- If $h_i(k)$ is greater than m, then just start counting from zero again

→ Hashing with Chaining: Hashing Table stores pointers for data structures (linked lists, for example) that save keys with the same bucket





Algorithm 8.19 Cuckoo Hashing: Insert

Input: key $k \in U$ we want to insert; counter `limit` specifying the maximum number of tries

Data Structures: arrays M_1, M_2 of equal size

Functions: hash functions $h_1 : U \to M_1, h_2 : U \to M_2$; chosen independently and uniformly at random from universal families

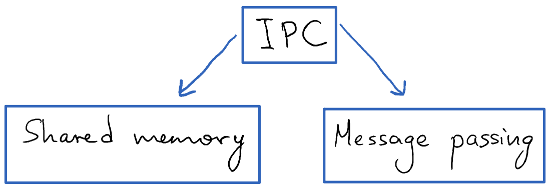
if $M_1[h_1(k)] = k$ or $M_2[h_2(k)] = k$ **then**
    return
end if
$t := 1$
while $t \leq$ `limit` **do**
    swap k with $M_1[h_1(k)]$
    if $k = \perp$ **then**
        return
    end if
    swap k with $M_2[h_2(k)]$
    if $k = \perp$ **then**
        return
    end if
    $t := t + 1$
end while
`rehash`()
`CuckooHashingInsert`(k, `limit`)

9 Processes and Concurrency

→ Independent process: A process that cannot affect or be affected by other processes

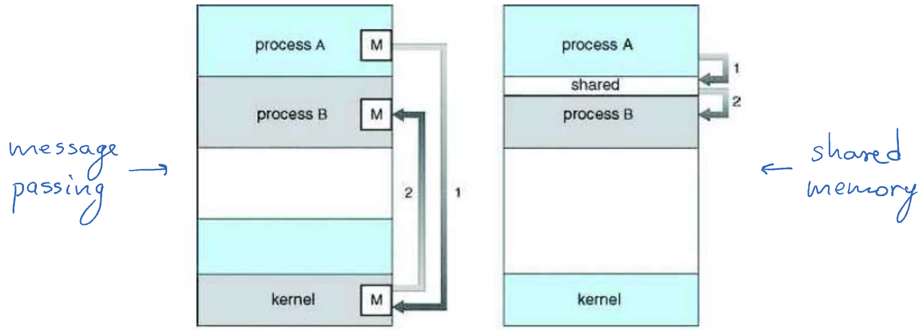
→ Cooperating process: Opposite of independent process

→ Cooperating process requires an interprocess communication (IPC)



• Shared memory: Part of the memory is used by both processes. They can communicate by reading/writing data from/to the memory.

• Message passing: „direct" communication channel between the processes



→ Race condition: Output depends on the order of the commands (like counter++ vs. counter--)

```
x = 3;
y = 4;
z = x * y;
```

```
x = 2;
y = 3;
z = x + y;
```

x = 3;	x = 2;	x = 3;
y = 4;	y = 3;	y = 4;
z = 12;	z = 5;	x = 2;
x = 2;	x = 3;	z = 6;
y = 3;	y = 4;	y = 3;
z = 5;	**z = 12;**	**z = 6;**

→ Critical section: Part of the code with shared memory

```
x = 3;
y = 4;
x = x + 2;
z = x * y;
```

```
a = 1;
x = 3;
y = 4;
z = x * y;
b = 2;
a = a * b;
```

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

→ problem divided into
- entry section
- critical section
- exit section
- remainder section

→ Solution must satisfy 3 conditions

i) Mutual exclusion: Only one process in critical section

ii) Progress: Processes that are not willing to enter the critical section have no role on deciding who is going to enter the critical section next.

iii) Bounded waiting: There is a limited number of times other processes can enter the critical section while the process is waiting → processes should not wait forever.

→ Preemptive and nonpreemptive kernels: Preemptive means it can be interrupted in the middle of a task.

Nonpreemptive kernels, since they cannot be interrupted in the middle of a task, do not carry out race conditions

1. Peterson's algorithms

→ Restricted to two processes, P_0 and P_1

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

→ shared information between P_0 and P_1 is
- int turn
- boolean flag[2]

→ First give the other process the chance to run
turn = j;

→ Wait until it exits the critical section by setting flag[j] = false;

while (flag[j] && turn == j);

→ Now run critical section and at the set flag[i] to false

```
y = 4;
x = x + 2;
z = x * y;
```

```
x = 3;
y = 4;
z = x * y;
b = 2;
a = a * b;
```

while (flag[j] && turn == j);

→ Now run critical section and at the set flag[i] to false

2. Semaphores

→ Use of an integer S

→ 2 atomic operations: wait (S) and signal (S)

```
Wait (S) {
        while S <= 0;
        S--;
}
```

```
Signal (S) {
        S++;
}
```

→

```
do {
        wait (S);

        critical section

        signal(S);

        remainder section
} while (TRUE);
```

while S <= 0; → Spin until „lock" is free (which is when S > 0).

S++; → makes S go from 0 to 1 ⇒ free the lock

→ **Busy waiting:** Other processes spin until they get the chance to enter the critical section

⇒ Semaphores are Spinlocks

It is not ideal, since CPU time is wasted (it could be productively used by some process that does not want to enter the critical section)

→ Deadlock

```
P0              P1

wait(S);        wait(Q);
wait(Q);        wait(S);
  .               .
  .               .
  .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Suppose that P_0 executes wait(S) and then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S). Since these signal() operations cannot be executed, P_0 and P_1 are deadlocked.

→ **Starvation:** Processes wait undefinitely within the semaphore

↳ Can happen when processes in the list associated with semaphores are removed in LIFO (last-in, first-out) order.

(10) Locks and Contention

→ Problem: Sometimes write data is first stored in so called write-buffers and only really saved on the memory when this data is needed again (143)

```
do {
        aquire lock;
        critical section
        release lock;
        remainder section
} while (TRUE);
```

→ In Java → Set lock before the try command (because setting the lock can cause an exception)

→ Test and Set (TAS)

➢ RMW Instruktion

```
synchronized int testAndSet () {
        int prior = value;
        value = 1;
        return prior; }
```

- value = 1; → lock it
- return prior; → return previous value: false ⇒ free; true ⇒ locked

➢ Locking

```
Public void lock() {
        while (state.testAndSet()) {}  }
```

→ Wait until it returns false ⇒ becomes free

➢ Unlock

```
Public void unlock() {
        state.set(0);  }
```

→ Set free

→ Test and Test and Set (TTAS)

• Wait until it supposingly becomes free and then apply the TAS (Test and Set)

➢ Locking

```
Public void lock() {
        while (true)  {
                while (state.get())  {}
                if (!state.getAndSet())
                        return;
        }
}
```

→ Only locking part is different

- while (state.get()) {} → until false ⇒ supposingly false
- if (!state.getAndSet()) → TAS

→ TTAS is more efficient because it reads the state.get() from cache (not using the communication bus)

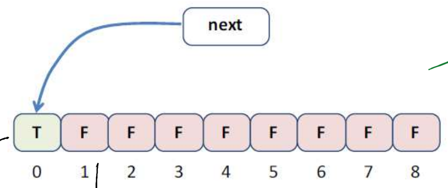
In TAS it must always write and read on the memory, using the bus

If there is only one bus for all processes ⇒ slow!

→ Queue Locks

• Array based locks → Processes / Threads follow a queue. They just have to see if the predecessor has finished (in the array).

→ A-lock



Process 0 executing

Process 1 Spins until Process 0 sets field 1 to true and field 0 to false ⇒ now Process 1 has the lock

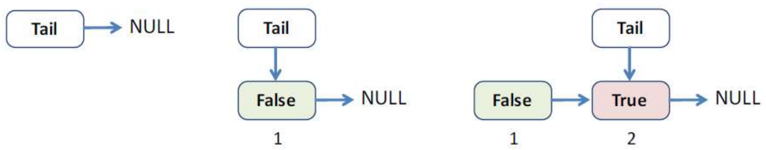
```
public class ALock implements Lock {
  ThreadLocal<Integer> mySlotIndex = new ThreadLocal<Integer> (){
    protected Integer initialValue() {
      return 0;
    }
  };
  AtomicInteger tail;
  boolean[] flag;
  int size;
  public ALock(int capacity) {
    size = capacity;
    tail = new AtomicInteger(0);
    flag = new boolean[capacity];
    flag[0] = true;
  }
  public void lock() {
    int slot = tail.getAndIncrement() % size;
    mySlotIndex.set(slot);
    while (! flag[slot]) {};
  }
  public void unlock() {
    int slot = mySlotIndex.get();
    flag[slot] = false;
    flag[(slot + 1) % size] = true;
  }
}
```

- CLH Queue lock:

i) Linked list instead of an array

ii) One node for each process/thread

iii) Shared resource is tail of the list



To acquire a slot the process/thread appends a node to the tail.

When finished, the process sets to false the value of the next node and to true his own value.

The process then spins until its value is set to false.

- Composite locks:

i) Limited number of nodes (they already exist)

ii) A process wants a lock → selects a random node and sees if it is free

- yes → take it
- no → keep looking for another node

iii) If it finished → set free the node

(11) Concurrent Data Structures

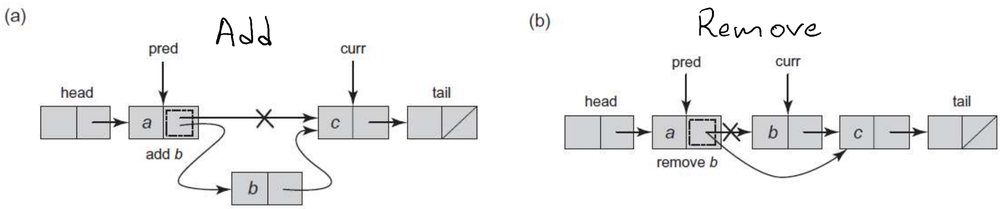
→ Synchronization for linked lists

→ Methods:

- **Wait-free**: Methods that guarantee that every call finishes in a finite number of steps
- **Lock-free**: Methods that guarantee that <u>some</u> call finishes in a finite number of steps

```
private class Node {
  T item;
  int key;
  Node next;
}
```

→ we will be considering this data structures



1. Coarse-Grained Synchronization

→ The lock is set for the whole linked list once

⇒ Only one process can work on the list at a time

Problem: Multiple threads working on the list. Although they work on different nodes, the whole list is locked for just one process.

```
public boolean add(T item) {
  Node pred, curr;
  int key = item.hashCode();
  lock.lock();
  try {
    pred = head;
    curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    if (key == curr.key) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
  } finally {
    lock.unlock();
  }
}
```

2. Fine-Grained Synchronization

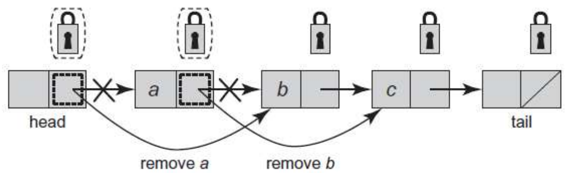
→ Improve concurrency by locking individual nodes (instead of whole list)

→ As a thread traverses a list, it locks each node when it first visits, and some time later releases it.

```
public boolean add(T item) {
  int key = item.hashCode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
      return true;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

```
public boolean remove(T item) {
  Node pred = null, curr = null;
  int key = item.hashCode();
  head.lock();
  try {
    pred = head;
    curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        pred.next = curr.next;
        return true;
      }
      return false;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

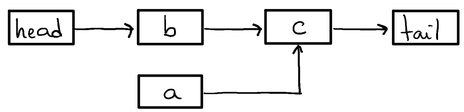
→ Why always lock two nodes? And why lock next node before unlocking the predecessor?



Process A → remove a
Process B → remove b

1. A locks head
2. B locks a
3. A sets head.next to b
4. B sets a.next to c

⇒



→ Deadlock

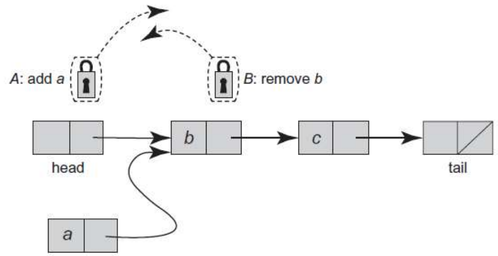


Figure 9.10 The FineList class: a deadlock can occur if, for example, remove() and add() calls acquire locks in opposite order. Thread A is about to insert a by locking first b and then head, and thread B is about to remove node b by locking first head and then b. Each thread holds the lock the other is waiting to acquire, so neither makes progress.

Problem: If a thread locks a node, no other thread can reach nodes that come after.

3. Optimistic - Synchronization

→ Optimistic point of view: Search without acquiring locks, lock the nodes found, and then confirm that the locked nodes are correct.

↳ If a synchronization conflict causes the wrong nodes to be locked ⇒ unlock nodes and start over.

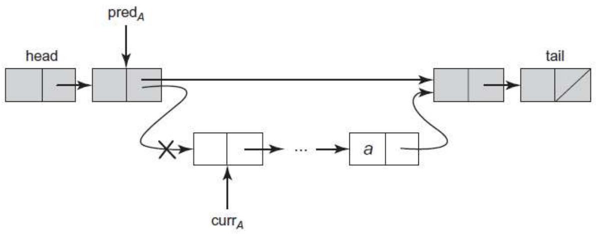
```java
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key <= key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) {
          return false;
        } else {
          Node node = new Node(item);
          node.next = curr;
          pred.next = node;
          return true;
        }
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

no locks

found ⇒ lock

```java
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) {
          pred.next = curr.next;
          return true;
        } else {
          return false;
        }
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

validate = check if $pred_A$ is reachable and $pred_A$.next is still $curr_A$




```java
private boolean validate(Node pred, Node curr) {
  Node node = head;
  while (node.key <= pred.key) {
    if (node == pred)
      return pred.next == curr;
    node = node.next;
  }
  return false;
}
```

Figure 9.15 The OptimisticList class: why validation is needed. Thread A is attempting to remove a node a. While traversing the list, $curr_A$ and all nodes between $curr_A$ and a (including a) might be removed (denoted by a lighter node color). In such a case, thread A would proceed to the point where $curr_A$ points to a, and, without validation, would successfully remove a, even though it is no longer in the list. Validation is required to determine that a is no longer reachable from head.

- Validation is required to determine that a is no longer reachable from head

- This method only makes sense when it is cheaper to traverse the list twice rather than once with locking.

4. Lazy - Synchronization

→ Boolean *marked* indicating whether the node is in the set

→ All unmarked nodes are considered to be reachable ⇒ no need to traverse twice the set (validation) for the contains-method

→ **Contains**: Traverse list (unmarked nodes) without validation and locks.

→ **Add**: Traverse list once, lock predecessor and current nodes, add node without validation

→ **Remove**: two steps

i) Logical removal → mark the node
ii) Physical removal → change pointer of predecessor } no "validation"

There is validation, but it does not traverse the entire list!

Validation: Check whether predecessor and current nodes are still unmarked

```java
private boolean validate(Node pred, Node curr) {
  return !pred.marked && !curr.marked && pred.next == curr;
}
```

```java
public boolean add(T item) {
 int key = item.hashCode();
 while (true) {
    Node pred = head;
    Node curr = head.next;
    while (curr.key < key) {
     pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) {
            return false;
          } else {
            Node node = new Node(item);
            node.next = curr;
            pred.next = node;
            return true;
          }
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
}
```

```java
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
   Node pred = head;
   Node curr = head.next;
   while (curr.key < key) {
    pred = curr; curr = curr.next;
   }
   pred.lock();
   try {
     curr.lock();
     try {
       if (validate(pred, curr)) {
         if (curr.key != key) {
           return false;
         } else {
           curr.marked = true;
           pred.next = curr.next;
           return true;
         }
       }
     } finally {
       curr.unlock();
     }
   } finally {
     pred.unlock();
   }
  }
}
```

```java
public boolean contains(T item) {
  int key = item.hashCode();
  Node curr = head;
  while (curr.key < key)
    curr = curr.next;
  return curr.key == key && !curr.marked;
}
```

5. Lock-free data structures

→ For example with CompareAndSet (CAS) → Logical removal and pointer change (physical removal) in just one step

→ Atomic markable reference

```java
class Window {
  public Node pred, curr;
  Window(Node myPred, Node myCurr) {
    pred = myPred; curr = myCurr;
  }
}
public Window find(Node head, int key) {
  Node pred = null, curr = null, succ = null;
  boolean[] marked = {false};
  boolean snip;
  retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ, false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
      }
      if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
  }
}
```

```java
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
     return false;
   } else {
     Node node = new Node(item);
     node.next = new AtomicMarkableReference(curr, false);
     if (pred.next.compareAndSet(curr, node, false, false)) {
       return true;
     }
   }
  }
}
```

```java
public boolean remove(T item) {
  int key = item.hashCode();
  boolean snip;
  while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key != key) {
     return false;
   } else {
     Node succ = curr.next.getReference();
     snip = curr.next.attemptMark(succ, true);
     if (!snip)
       continue;
     pred.next.compareAndSet(curr, succ, false, false);
     return true;
   }
  }
}
```

```java
public boolean contains(T item) {
  boolean[] marked = false{};
  int key = item.hashCode();
  Node curr = head;
  while (curr.key < key) {
   curr = curr.next;
   Node succ = curr.next.get(marked);
  }
  return (curr.key == key && !marked[0])
}
```

→ Problems:

- The need to support atomic modification of a reference and a Boolean mark has an added performance cost.[5]
- As add() and remove() traverse the list, they must engage in concurrent cleanup of removed nodes, introducing the possibility of contention among threads, sometimes forcing threads to restart traversals, even if there was no change near the node each was trying to modify.

Locks for Hashing

→ See PVK stuff

(12) Markov Chains and Page Rank

1. Markov Chains

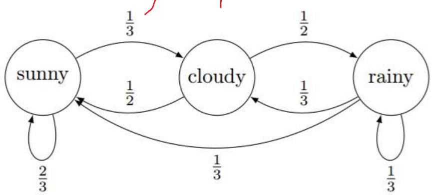
→ **Definition**: Let S be a finite or countably infinite set of states. A Markov Chain is a sequence of random variables $X_0, X_1, \ldots \in S$ that satisfies the Markov Property

→ **Markov Property**: A sequence (X_t) of random variables has the Markov Property if for all t, the probability ditribution for X_{t+1} depends only on X_t, but not on $X_{t-1}, X_{t-2}, \ldots, X_0$.

$$Pr[X_{t+1} = s_{t+1} \mid X_0 = s_0, X_1 = s_1, \ldots, X_t = s_t] = Pr[X_{t+1} = s_{t+1} \mid X_t = s_t]$$

→ **Time Homogeneous Markov Chain**: $Pr[X_{t+1} = s_{t+1} \mid X_t = s_t]$ is independent of t $\Rightarrow$ $p_{i,j} = Pr[X_{t+1} = i \mid X_t = j]$

Example: ↗ independent of time



Matrix →

from \ to	sunny	cloudy	rainy
sunny	2/3	1/3	0
cloudy	1/2	0	1/2
rainy	1/3	1/3	1/3

- $q_{t+1,j} = \sum_{i \in S} Pr[X_t = i] \, Pr[X_{t+1} = j \mid X_t = i] = \sum_{i \in S} q_{t,i} \cdot p_{i,j}$

$\Rightarrow q_{t+1} = q_t \cdot P \Rightarrow q_t = q_0 P^t$ ⟶ initial state → $q_0 (1,0,0)$

↳ Matrix of Markov Chain

- q_0 = initial state → $q_0 = (1,0,0) \Rightarrow$ start at node sunny with probability 1. $q_0 = (0,1,0) \Rightarrow$ start at node cloudy

- entry at (i,j) from $P^t \Rightarrow$ „Probability of reaching j from i in t steps"

→ **Random Walk**:

Definition 12.5 (Random Walk). *Let $G = (V, E)$ be a directed graph, and let $\omega : E \to [0,1]$ be a weight function so that $\sum_{v:(u,v)\in E} \omega(u,v) = 1$ for all nodes u. Let $u \in V$ be the* **starting node**. *A* **weighted random walk** *on G* **starting at** *u is the following discrete Markov chain in discrete time. Beginning with $X_0 = u$, in every step t, the node X_{t+1} is chosen according to the weights $\omega(X_t, v)$, where v are the neighbors of X_t. If G is undirected and unweighted, then X_{t+1} is chosen uniformly at random among X_t's neighbors and the random walk is called* **simple**.

2. Hitting Time and Arrival Probability

→ **Sojourn time T_i**: T_i of state i is the time the process stays at i.

$$Pr[T_i = k] = p_{i,i}^{k-1} (1 - p_{i,i})$$ → geometrical distribution

→ **Arrival Probability**: Probability that we end on a specific node

$$f_{i,j} = Pr[T_{i,j} < \infty] = p_{i,j} + \sum_{k \neq j} p_{i,k} \cdot f_{k,i}$$

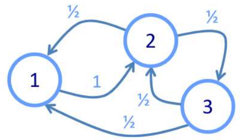
→ **Hitting Time $T_{i,j}$**: Random variable counting the number of steps until visiting j the first time when starting from state i

→ **Commute Time $c_{i,j}$**: $h_{i,j} + h_{j,i}$

$$h_{i,j} = E[T_{i,j}] = 1 + \sum_{k \neq j} p_{i,k} \cdot h_{k,j}$$

$h_{1,1} = 1 + p_{1,2} h_{2,1}$
$h_{1,2} = 1$
$h_{1,3} = 1 + p_{1,2} h_{2,3}$
…

$$c_{i,j} = h_{i,j} + h_{j,i}$$



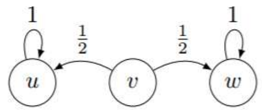
→ **Stationary Distribution**: If we look at q_t for $t \to \infty$ and it converges to some vector π, then it holds that $\pi = \pi \cdot P$

$$(\pi_1, \pi_2, \pi_3)^T \begin{bmatrix} p_{11} & \cdots \\ \vdots & \ddots \end{bmatrix} = (\pi_1, \pi_2, \pi_3)^T$$ Linear System of equations → $\pi_1 = \frac{1}{2}\pi_2 + \frac{1}{2}\pi_3$, $\pi_2 = \pi_1 + \frac{1}{2}\pi_3$, ⋮

- It must hold $\pi_i \geq 0$ and $\sum_i \pi_i = 1$

→ **Irreducible Markov Chain**: All states are reachable from all other states. ⟶ $f_{i,j} = 1 \; \forall i,j$ (direct connection!)

→ **Absorbing State**: States where „once you get in you can't get out"



- u, w are absorbing states
- if there exist absorbing states ⇒ Markov chain is not irreducible

- Every finite irreducible Markov Chain has a unique stationary distribution π given by

$$\pi_j = \frac{1}{h_{j,j}}$$

→ Aperiodic Markov Chain: When all states have Period = 1

- Find period of a state:
  i. Count number of „hops" for all Roundtrips starting from the node of interest.
  ii. Period is the greatest common divisor of all the hopcounts (ggT) → maior valor inteiro que consegue dividir os valores dados

→ Ergodic Markov Chain: When Markov Chain is irreducible and aperiodic

- Ergodic Markov Chain ⇒ $\lim_{t \to \infty} q_t = \pi$ (and it is unique)

→ If it holds $(i,i) \neq 0$ $\forall i$ in P ⇒ Markov Chain is aperiodic

$(i,i) \neq 0$ ⇒ connection to itself → (i) → $ggT(1, \ldots) = 1$ ⇒ aperiodic

↳ since it holds for all states ⇒ Markov Chain aperiodic

→ Markov Chain irreducible and at least one state with period 1 ⇒ ergodic

→ Irreducible ⇒ All states have the same period (different periods ⇒ not irreducible ⇒ not ergodic)

3. Page Rank Algorithm

→ Idea: directed graph in which the nodes are websites and an edge (v,u) indicates that website v contains a hyperlink to website u.

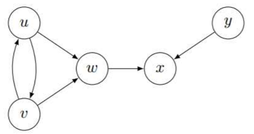


Figure 12.21: An example of a web graph with 5 websites. Website x does not link to any other website, i.e., x is a *sink*.

→ Find „importance" of websites:

i. Naïve approach: Rank sites by number of incoming hyperlinks

ii. Google's approach: Random walker that follows hyperlinks and counts how many times a website was visited.

Dead node (sink) ⇒ start again from a random website

→ Lets denote the random walker matrix as W. Calculate the stationary distribution is not feasable (too many websites and $q_0 W$, $q_1 W$ ... will not necessarily converge)

Solution: Make the Markov Chain ergodic by adding an edge between each node

→ Google Matrix:

$$M = \alpha W + (1-\alpha) R$$

M = Google Matrix
α = constant, $\alpha \in (0,1)$
R = Matrix with all entries $1/n$

- with probability $1-\alpha$ the random surfer gets „bored" and goes to a random website

- R changes the stationary distribution → Solution is to make the probability described above small ($\alpha \approx 0.85$)

→ Fooling Page Ranks:

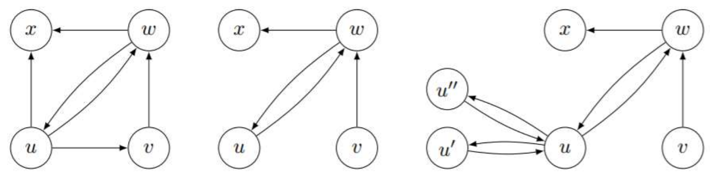


Figure 12.23: Website u wants to improve its PageRank, which is ≈ 0.23 in the initial setting on the left. First, all outgoing links to websites that do not link back are removed. The PageRank improves to ≈ 0.27. In a *Sybil attack* (right) the owner of u creates fake websites u' and u'' whose purpose is to exchange links with u. Moreover, the new websites increase the probability to visit u after a sink. Now, website u is the highest ranked site in the network with a rank of ≈ 0.41.

- Sybil attack: Single party pretends to be more than one individual, creating fake websites that point to the original website (increasing its ranking)

→ Simple Random Walks (simple ⇒ undirected Graph)

Let G be a graph with m edges. The stationary distribution of any simple random walk on G is

$$\pi_u = \frac{\delta(u)}{2m}$$

(13) Security and Cryptography

1. Secret sharing and Bulk encryption

→ **Perfect Secrecy**: encrypted message reveals no information to an attacker, except for the possible maximum length of the message

→ **Threshold Secret Sharing**: Let $t, n \in \mathbb{N}$ with $1 \leq t \leq n$. An algorithm that distributes a secret among n participants such that t participants need to collaborate to recover the secret is called a (t,n)-threshold secret sharing sheme.

Algorithm 13.4 (n, n)-Threshold Secret Sharing

Input: A secret k, encoded in binary representation of length $l(k)$.

Secret distribution

1: Generate $n-1$ random binary numbers k_i of length $l(k)$ and distribute them among $n-1$ participants
2: Give participant n the value k_n as the result of *XOR* of k and $k_1, \dots, k_{n-1}$, i.e., $k_n = k \oplus k_1 \oplus k_2 \oplus \dots \oplus k_{n-1}$

Secret recovery

1: Collect all n values $k_1, \dots, k_n$ and obtain $k = k_1 \oplus k_2 \oplus \dots \oplus k_{n-1} \oplus k_n$

→ Perfect Secrecy even with n-1 collaborators

→ $a \oplus a = 0$ and $b \oplus 0 = b$

Algorithm 13.6 (t, n)-Threshold Secret Sharing

Input: A secret k, represented as a real number.

Secret distribution

1: Generate $t-1$ random $a_1, \dots, a_{t-1} \in \mathbb{R}$
2: Obtain a polynomial f of degree $t-1$ with $f(x) = k + a_1x + \dots + a_{t-1}x^{t-1}$
3: Generate n distinct $x_1, \dots, x_n \in \mathbb{R} \setminus \{0\}$
4: Distribute $(x_1, f(x_1))$ to participant $P_1, \dots, (x_n, f(x_n))$ to P_n

Secret recovery

1: Collect t pairs $(x_i, f(x_i))$ from at least t participants
2: Use Lagrange's interpolation formula to obtain $f(0) = k$

→ $L_k^n = \prod_{i=0}^{n} \frac{x - x_i}{x_k - x_i}$

Algorithm 13.7 One-Time Pad

Input: A message m known to Alice, and a symmetric key k (as a random bitstring) of length $l(k)$, known by both Alice and Bob.

Encryption

1: Alice sends $c = m \oplus k$ to Bob

↗ Also has perfect secrecy

Decryption

1: Bob obtains m by $m = c \oplus k$

→ **Bulk Encryption Algorithm**: A bulk encryption algorithm can securely encrypt a message of any size.

→ **Electronic Code Book - ECB**: Given a method to encrypt a block of x bits, ECB encrypts a message of length rx by splitting the message into r blocks of length x, encrypting each block separately.

Problem: If we encrypt m_1 and m_2 with k, becoming $c_1 = m_1 \oplus k$ and $c_2 = m_2 \oplus k$, we can find $m_1 \oplus m_2$

$c_1 \oplus c_2 = m_1 \oplus m_2 \implies$ too much information for Eve (besides just length of the messages)

→ **Cipher Block Chaining - CBC**: Splitting of the message into r blocks of length x, but not using always k to encrypt

$$c_i = m_i \oplus c_{i-1}, \quad c_0 \text{ is initialized randomly}$$

Problem: Just like before, Eve can still find $m_4 \oplus m_5$, for example.

Problem: Bob and Alice have to agree on a large number in secret!

2. Key Exchange

→ **Problem**: try to agree on a number (secret) without actually meeting → <u>public key</u>

→ **Primitive root**: Let $p \in \mathbb{N}$ be a prime number. $g \in \mathbb{N}$ is a primitive root of p if

$$\forall h \in \mathbb{N}, \ 1 \leq h < p \ \exists k \in \mathbb{N} \text{ s.t. } g^k = h \bmod p$$

p = 5
g = 2 → Primitive root of p

$2^1 = 2 \bmod 5$
$2^2 = 4 \bmod 5$
$2^3 = 3 \bmod 5$
$2^4 = 1 \bmod 5$

→ there is a k for all $h < p$ [1, 2, 3, 4]

Algorithm 12.3 Diffie-Hellman Key Exchange

Input: Publicly known prime p and a primitive root g of p.
Result: Alice and Bob agree on a common secret key.

1: Alice picks k_A, with $1 \leq k_A \leq p-2$ and sends $g^{k_A} \bmod p$ to Bob
2: Bob picks k_B, with $1 \leq k_B \leq p-2$ and sends $g^{k_B} \bmod p$ to Alice
3: Alice calculates $(g^{k_B})^{k_A} \bmod p = g^{k_B k_A} \bmod p$
4: Bob calculates $(g^{k_A})^{k_B} \bmod p = g^{k_A k_B} \bmod p$
5: Alice & Bob have a common secret key $g^{k_A k_B} \bmod p = g^{k_B k_A} \bmod p$

Alice receives $g^{k_B} \bmod p$

what we send ↓

→ $g^{k_B k_A} = ? \bmod p$ ↓

Schreibweise $g^{k_B k_A} \bmod p$

➤ **Alice**
- Wählt $k_A = 2$
- Sendet $2^2 \bmod 5$ (also 4) an Bob
- Rechnet $3^2 \bmod 5 =$ **4**

➤ **Bob**
- Wählt $k_B = 3$
- Sendet $2^3 \bmod 5$ (also 3) an Bob
- Rechnet $4^3 \bmod 5 =$ **4**

1. Alice sends $g^{k_A} \bmod p =: C_{Alice}$
   Bob sends $g^{k_B} \bmod p =: C_{Bob}$

2. Alice receives C_{Bob} and calculates $(C_{Bob})^{k_A} \bmod p$
   Bob receives C_{Alice} and calculates $(C_{Alice})^{k_B} \bmod p$ } Same!

They agreed on a secret number

$$K = B^{k_A} = (g^{k_B})^{k_A} = g^{k_B k_A} = (g^{k_A})^{k_B} = A^{k_B} = K \bmod p$$

→ **Discrete Logarithm Problem**: Let $p \in \mathbb{N}$ be a prime, and let $g, a \in \mathbb{N}$ with $1 \leq g, a < p$. The discrete logarithm problem is defined as finding an $x \in \mathbb{N}$ with $g^x = a \bmod p$

for key exchange it could be used to find $k_A, k_B, k_A \cdot k_B$

→ Finding large prime numbers:

Algorithm 13.18 Probabilistic Primality Testing

Input: An odd number $p \in \mathbb{N}$.
Result: Is p a prime?

```
Let j, r ∈ ℕ and j odd with p − 1 = 2^r j
Select x ∈ ℕ uniformly at random, 1 ≤ x < p
Set x_0 = x^j  mod p
if x_0 = 1 or x_0 = p − 1 then
    Output "p is probably prime" and stop
end if
for i = 1, . . . r − 1 do
    Set x_i = x_{i−1}^2  mod p
    if x_i = p − 1 then
        Output "p is probably prime" and stop
    end if
end for
Output "p is not prime"
```

Lemma 13.19. *Algorithm 13.18 is correct with probability 75% if it outputs "p is probably prime", and 100% correct if it outputs "p is not prime".*

Corollary 13.20. *The runtime of Algorithm 13.18 is $O(r) \in O(\log p)$*

→ **Man in the Middle Attack**: Attacker Eve deciphering or changing the message between Alice and Bob (without them noticing)

→ Diffie-Hellman Key Exchange is vulnerable to a man in the middle attack

↳ Eve can start a communication with Alice pretending to be Bob and with Bob pretending to be Alice (Bob and Alice will think they are communicating between them)

↳ Solution would be meeting in private and agreeing on a private key $k_{A,B}$.

→ **Forward secrecy**: A sequence of secured communications has the property of forward secrecy, if finding the secret key in one of the rounds does not reveal the messages of the previous communications ⇒ Not use the same private key multiple times

Algorithm 13.24 Diffie-Hellman Key Exchange with Forward Secrecy

Input: Alice's and Bob's common secret key $k_{A,B}$, and furthermore a prime p with a primitive root g for p.
Result: A Diffie-Hellman key exchange not vulnerable to a man in the middle attack, and with forward secrecy.

1: Bob picks a random number $k_B \in \{1, 2, \ldots, p-1\}$ and sends Alice $(g^{k_B} \bmod p)$ encrypted with $k_{A,B}$ as c_B as a challenge
2: Alice picks a random number $k_A \in \{1, 2, \ldots, p-1\}$ and sends $(g^{k_A} \bmod p)$ encrypted with $k_{A,B}$ as c_A to Bob as a challenge
3: Alice and Bob decrypt the respective messages, and Alice sends $g^{k_B} + 1$ encrypted with $k_{A,B}$ to Bob as a response (and Bob as well with $g^{k_A} + 1$)
4: If decryption yields $g^{k_A} + 1$ for Alice, and $g^{k_B} + 1$ for Bob, respectively, they accept the round key $g^{k_A k_B} \bmod p$

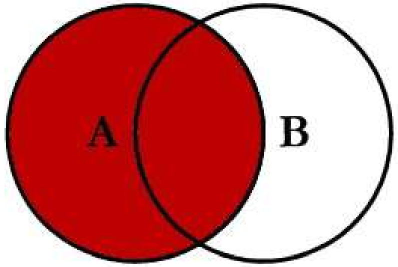
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

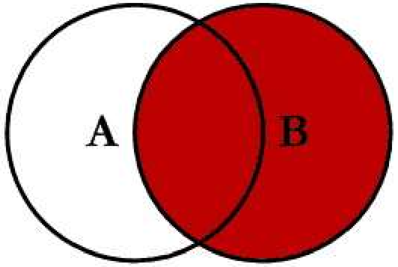
d	r	w	x	r	–	x	r	–	–
	read	write	exec	read	write	exec	read	write	exec
File type (directory)	Owner permissions			Group permissions			User permissions		
	4	2	1	4	2	1	4	2	1
	7			5			4		

„–" for files

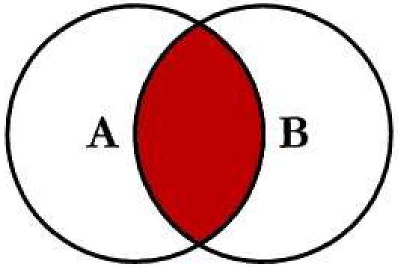
SQL JOINS



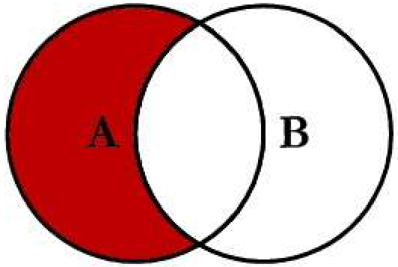

```sql
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



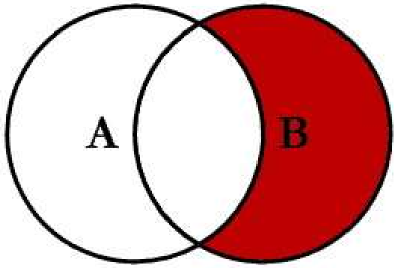

```sql
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



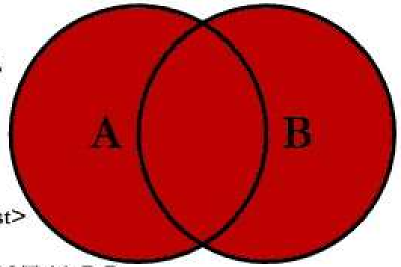

```sql
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



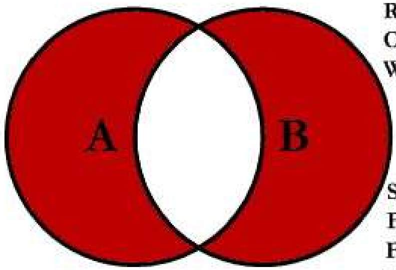

```sql
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```sql
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```sql
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```sql
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

SQL cheat sheet

Basic Queries

-- filter your columns
SELECT col1, col2, col3, ... **FROM** table1
-- filter the rows
WHERE col4 = 1 **AND** col5 = 2
-- aggregate the data
GROUP by ...
-- limit aggregated data
HAVING count(*) > 1
-- order of the results
ORDER BY col2

Useful keywords for **SELECTS**:

DISTINCT - return unique results
BETWEEN a **AND** b - limit the range, the values can be numbers, text, or dates
LIKE - pattern search within the column text
IN (a, b, c) - check if the value is contained among given.

Data Modification

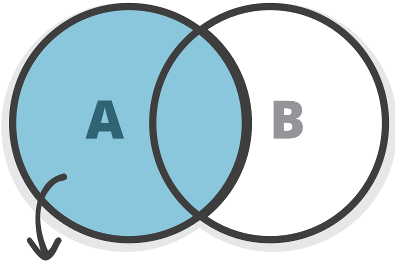
-- update specific data with the **WHERE** clause
UPDATE table1 **SET** col1 = 1 **WHERE** col2 = 2
-- insert values manually
INSERT INTO table1 **(ID, FIRST_NAME, LAST_NAME)**
VALUES (1, 'Rebel', 'Labs');
-- or by using the results of a query
INSERT INTO table1 **(ID, FIRST_NAME, LAST_NAME)**
SELECT id, last_name, first_name **FROM** table2

Views

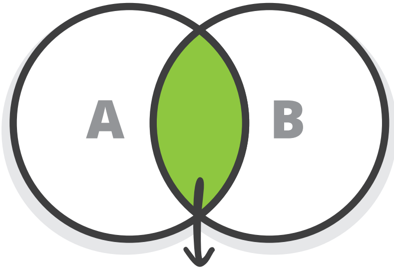
A **VIEW** is a virtual table, which is a result of a query.
They can be used to create virtual tables of complex queries.

CREATE VIEW view1 **AS**
SELECT col1, col2
FROM table1
WHERE ...

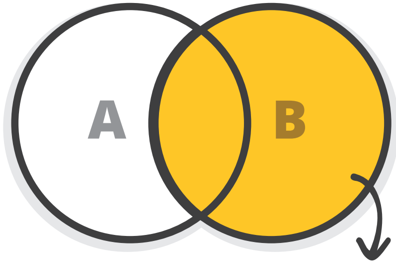
The Joy of JOINs



***LEFT OUTER JOIN** - all rows from table A, even if they do not exist in table B*



***INNER JOIN** - fetch the results that exist in both tables*



***RIGHT OUTER JOIN** - all rows from table B, even if they do not exist in table A*

Updates on JOINed Queries

You can use **JOIN**s in your **UPDATE**s
UPDATE t1 **SET** a = 1
FROM table1 t1 **JOIN** table2 t2 **ON** t1.id = t2.t1_id
WHERE t1.col1 = 0 **AND** t2.col2 **IS NULL**;

NB! Use database specific syntax, it might be faster!

Semi JOINs

You can use subqueries instead of **JOIN**s:

SELECT col1, col2 **FROM** table1 **WHERE** id **IN**
(**SELECT** t1_id **FROM** table2 **WHERE** date >
CURRENT_TIMESTAMP)

Indexes

If you query by a column, index it!
CREATE INDEX index1 **ON** table1 (col1)

Don't forget:

Avoid overlapping indexes
Avoid indexing on too many columns
Indexes can speed up **DELETE** and **UPDATE** operations

Useful Utility Functions

-- convert strings to dates:
TO_DATE (Oracle, PostgreSQL), **STR_TO_DATE** (MySQL)
-- return the first non-NULL argument:
COALESCE (col1, col2, "default value")
-- return current time:
CURRENT_TIMESTAMP
-- compute set operations on two result sets
SELECT col1, col2 **FROM** table1
UNION / EXCEPT / INTERSECT
SELECT col3, col4 FROM table2;

Union - returns data from both queries
Except - rows from the first query that are not present in the second query
Intersect - rows that are returned from both queries

Reporting

Use aggregation functions

COUNT - return the number of rows
SUM - cumulate the values
AVG - return the average for the group
MIN / MAX - smallest / largest value

OSI Model

OSI model

Layer	Name	Example protocols
7	Application Layer	HTTP, FTP, DNS, SNMP, Telnet
6	Presentation Layer	SSL, TLS
5	Session Layer	NetBIOS, PPTP
4	Transport Layer	TCP, UDP
3	Network Layer	IP, ARP, ICMP, IPSec
2	Data Link Layer	PPP, ATM, Ethernet
1	Physical Layer	Ethernet, USB, Bluetooth, IEEE802.11

OSI (Open Source Interconnection) 7 Layer Model

Layer	Application/Example		Central Device/ Protocols			DOD4 Model
Application (7) Serves as the window for users and application processes to access the network services.	**End User layer** Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management		**User Applications** SMTP		GATEWAY Can be used on all layers	Process
Presentation (6) Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	**Syntax layer** encrypt & decrypt (if needed) Character code translation • Data conversion • Data compression • Data encryption • **Character Set Translation**		JPEG/ASCII EBDIC/TIFF/GIF PICT			
Session (5) Allows session establishment between processes running on different stations.	**Synch & send to ports** (logical ports) Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.		**Logical Ports** RPC/SQL/NFS NetBIOS names			
Transport (4) Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	**TCP** Host to Host, Flow Control Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing	PACKET FILTERING	TCP/SPX/UDP			Host to Host
Network (3) Controls the operations of the subnet, deciding which physical path the data takes.	**Packets** ("letter", contains IP address) Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting		**Routers** IP/IPX/ICMP			Internet
Data Link (2) Provides error-free transfer of data frames from one node to another over the Physical layer.	**Frames** ("envelopes", contains MAC address) [NIC card — Switch — NIC card] (end to end) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control		**Switch Bridge WAP** PPP/SLIP	Land Based Layers		Network
Physical (1) Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	**Physical structure** Cables, hubs, etc. Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts		**Hub**			