Week 7 - Recap

Pamela Delgado

April 03, 2018

Demand Paging

- Demand paging
- Page fault handling
- Page replacement
- Frame allocation
- Global vs local replacement
- Optimizations

Demand Paging

- Demand paging
  - Only part of a program is in memory
- Page fault handling
- Page replacement
- Frame allocation
- Global vs local replacement
- Optimizations

Demand Paging

- Demand paging
- Page fault handling
  - Bringing page from disk to memory
- Page replacement
- Frame allocation
- Global vs local replacement
- Optimizations

Demand Paging

- Demand paging
- Page fault handling
- Page replacement
  - Selecting page to replace if out of frames
- Frame allocation
- Global vs local replacement
- Optimizations

Demand Paging

- Demand paging
- Page fault handling
- Page replacement
- Frame allocation
  - How many frames to allocate to a process
  - Working set
- Global vs local replacement
- Optimizations

Frame allocation

- How many frames to allocate to a process?
- Minimum number of frames
  1. Reason: performance
  2. Architecture dependent
- Maximum number of frames
  – Physical memory size
- Degree of multiprocessing vs page faults tradeoff

Frame allocation

- How many frames to allocate to a process?
- Working set
  - Set of pages needed for execution over the next execution interval
  - Intuition: program's locality → less page faults
  - Choose right Δ

Frame allocation

- How many frames to allocate to a process?
- Working set
  – Set of pages needed for execution over the next execution interval
  – Intuition: program’s locality → less page faults
  – Choose right Δ
- Working set implementation
  – Set of pages in most recent Δ page references
  – Size (Δ): periodically count/update a reference bit

Demand Paging

- Demand paging
- Page fault handling
- Page replacement
- Frame allocation
- Global vs local replacement
- Optimizations

Global vs local replacement

- Local = replace frame of own set
- Global = replace any frame
  - Priorities among processes
  - Problem: cant control own page-fault rate
    - Variable execution time
  - Greater good, generally used
- Thrashing
  - Process doing more paging than executing

Week 8: File Systems - Introduction

Pamela Delgado

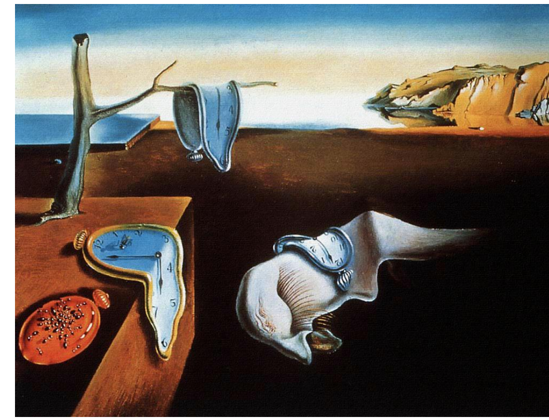
April 10, 2019

based on:
- W. Zwaenepoel slides
- Arpaci-Dusseau book
- Silbershatz book

Key Points

- Persistence: notion of “permanent” storage
- File system interface
- Disk management

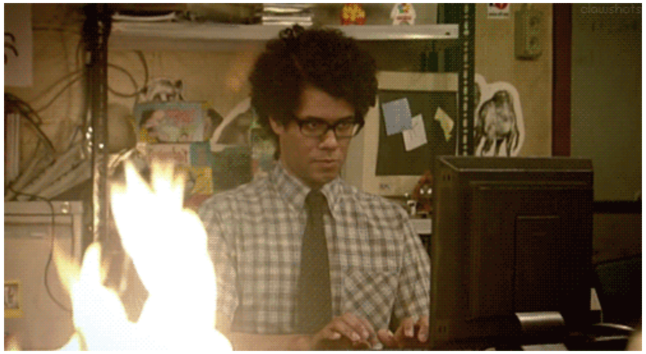
“Permanent” Storage



- How permanent is permanent?
- Across program invocations
- Across login
- Across machine failures/restarts
- Across disk failures
- Across multiple disk (data center) failures

“Permanent” Storage

- For this course
- Across program invocations
- Across login
- Across machine failures/restarts
- Across disk failures
- Across data center failures



Permanent Storage Media

- Main memory – not suitable

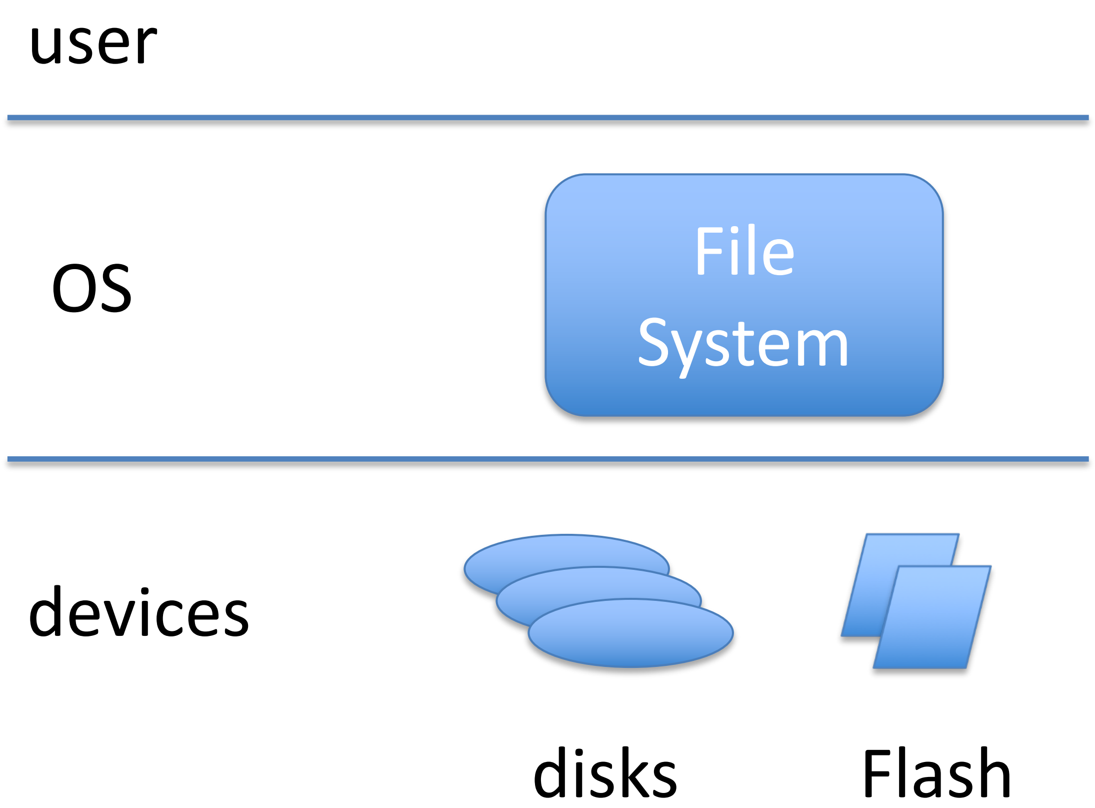
- Battery-backed memory
- Nonvolatile memory
  - Flash, but also other technologies coming
- Disks
- Tapes

Permanent Storage Media

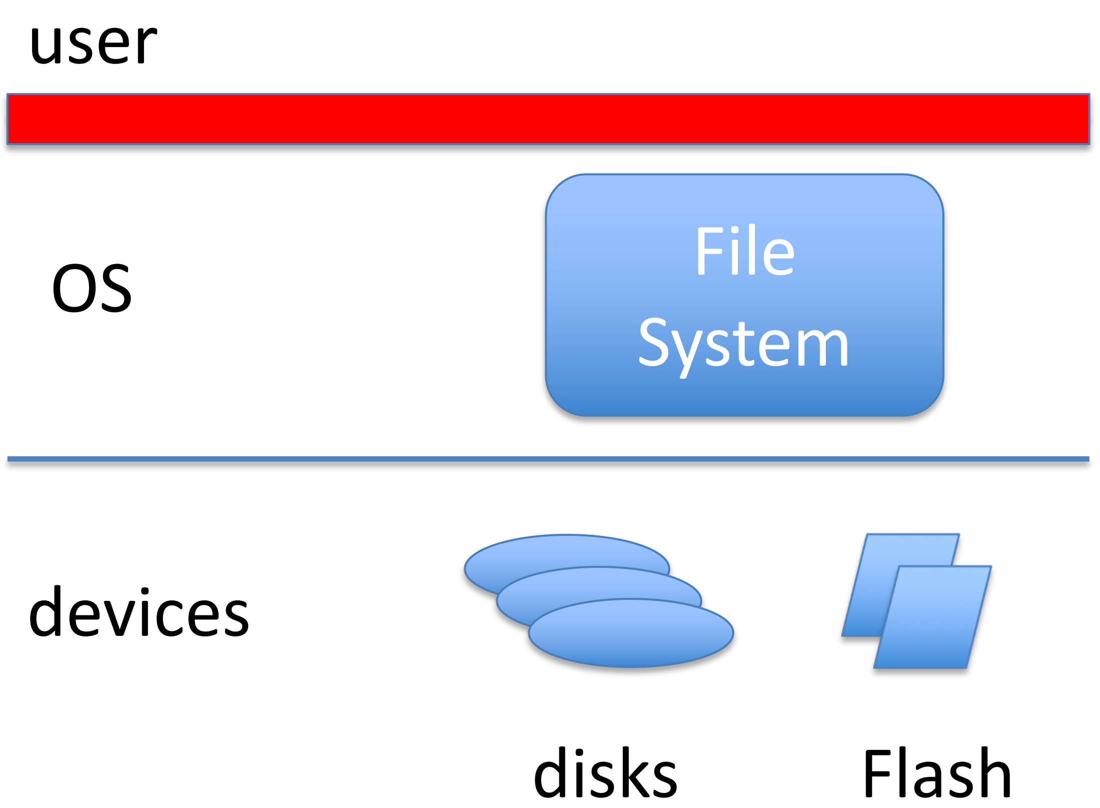
- For this course

- Battery-backed up memory
- Nonvolatile memory
  - Flash, but also other technologies coming
- Disks
- Tapes

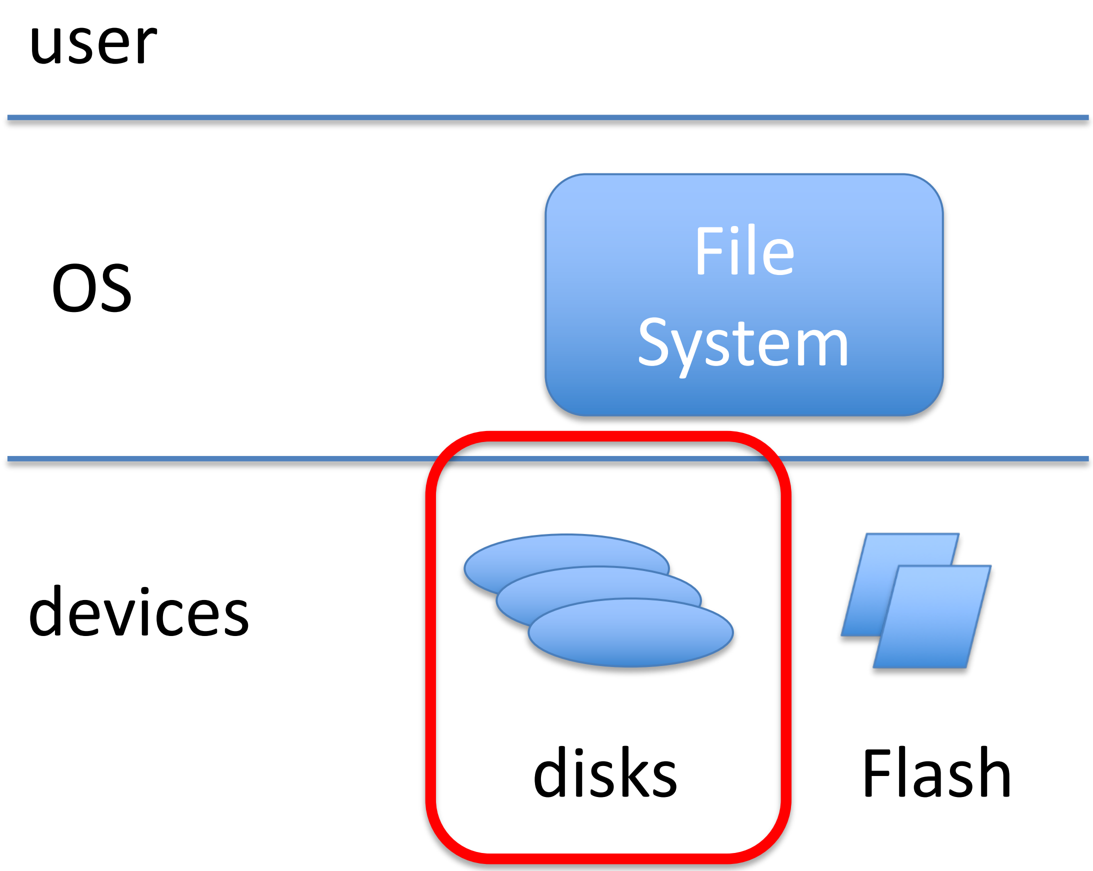
Overall Picture



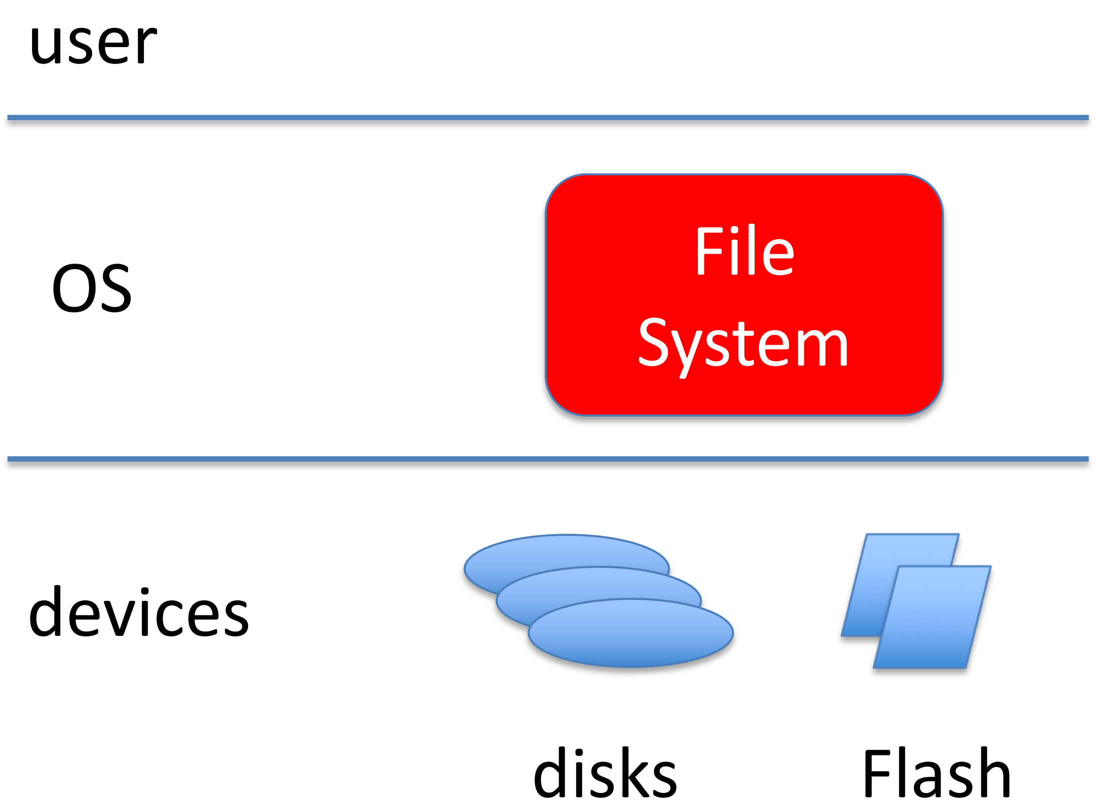
Today's Lecture – First Part



Today's Lecture – Second Half



Next Lecture



What is a File?

- Un-interpreted collection of objects
  - Represent related information

- Un-interpreted ~=
  - File system does not know what data means
  - Only application knows

What is a File?

- Un-interpreted collection of objects
  - Represent related information
- Objects:
  - Bytes
  - Records
  - …
- We will look at bytes (as in Linux)

Typed or Untyped? 

- Typed = FS knows what the object means

- Advantages
  - Invoke certain programs by default
  - Prevent errors
  - More efficient storage

Typed or Untyped?

- Typed = FS knows what the object means

- Disadvantages
  - Can be inflexible (typecast)
  - Can become a lot of code (many types)

- We will look at untyped files

An Aside: File Name Extensions = Types ?

- Pure convention, hint (Linux)
  - User knows, system does not do anything with it

- Known to the system (Windows/Mac OS X)
  - User knows, systems knows (and enforces)
  - In Mac OS X also creator information

File System Primitives

- Access
- Concurrency
- Naming
- Protection

File System Primitives

- Access
- Concurrency
- Naming
- Protection

Main Access Primitives

- Create()
- Delete()

- Read()
- Write()

Create() and Delete()

- uid = Create([optional arguments])
  - *uid* unique identifier, not human-readable string
  - Creates an empty file

- Delete(uid)
  - Deletes file with identifier *uid*
  - Usually also deletes all of its contents

Read()

- Read(uid, buffer, from, to)
  - Reads from file with identifier *uid*
  - From byte *from* to byte *to*
    - Can cause EOF (End-of-file) condition
  - Into a memory buffer *buffer*
    - previously allocated
    - must be of sufficient size

Write()

- Write(uid, buffer, from, to)
  - Write to file with identifier *uid*
  - Into byte *from* to byte *to*
  - From a memory buffer *buffer*

Sequential vs Random Access

- Read() and Write() in previous slide:
  - *Random-access* primitives
  - No connection between two successive accesses
- Sequential access is very common:
  - Read from where you stopped reading
  - Write to where you stopped writing
  - In particular, whole file access is common
- For this reason, sequential access methods

Sequential Read()

- File system keeps file pointer *fp* (initially 0)
- Read(uid, buffer, bytes)
  - Read from file with unique identifier *uid*
  - Starting from byte *fp*
  - *Bytes* bytes
  - Into memory buffer *buffer*
  - *fp += bytes*

Sequential can be built on Random

- Maintain *fp*-equivalent in user code

- myfp = 0
- Read(uid, buffer, myfp, myfp+bytes-1)
- myfp += bytes
- Read(uid, buffer, myfp, myfp+bytes-1)
- …

Can Random be built on Sequential?

- Not without an additional primitives

- Seek(uid, to)
  - fp = to

Using Seek to Implement Random

- Read(uid, buffer, from, to)

- Seek(uid, from)
- Read(uid, buffer, to-from+1)

Random vs. Sequential

- Sequential access is very common
- All systems provide sequential access
- Some systems provide
  - Only sequential access
  - Plus Seek()

File System Primitives

- Access
- Concurrency
- Naming
- Protection

Concurrent (Sequential) Access

- Two processes access the same file
- What about *fp*?

The Notion of an “Open” File

- Open()
- Close()

Open()

- tid = Open(uid, [optional args])
  - Creates an instance of file with *uid*
  - Accessible by this process only
  - With the temporary process-unique id *tid*
  - *fp* is associated with *tid*, not with *uid*

- Close(tid)
  - Destroys the instance

Putting Open() together with Read()

- tid = Open()
- Read(tid, buffer, bytes)
- Other Read()s or Write()s
- …
- Close(tid)

Semantics of Concurrent Open()s

- Separate instances altogether
  - Write()s by one not visible to others
- Separate instances until Close()
  - Write()s visible after Close()
- One single instance of the file
  - Write()s visible immediately to others

- *fp* is private!

File System Primitives

- Access
- Concurrency
- Naming
- Protection

Naming Primitives

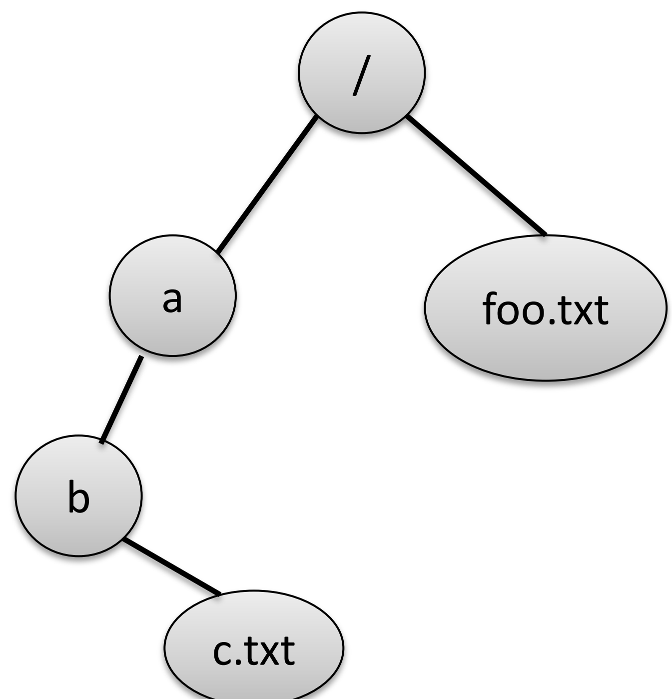
- Naming = mapping

  human-readable string ➔ uid

- Directory = collection of such mappings

Directory Structure

- Flat
- Two-level: [user] filename
- Hierarchical: /a/b/c …
  - Root directory
  - Working directory



Naming Primitives

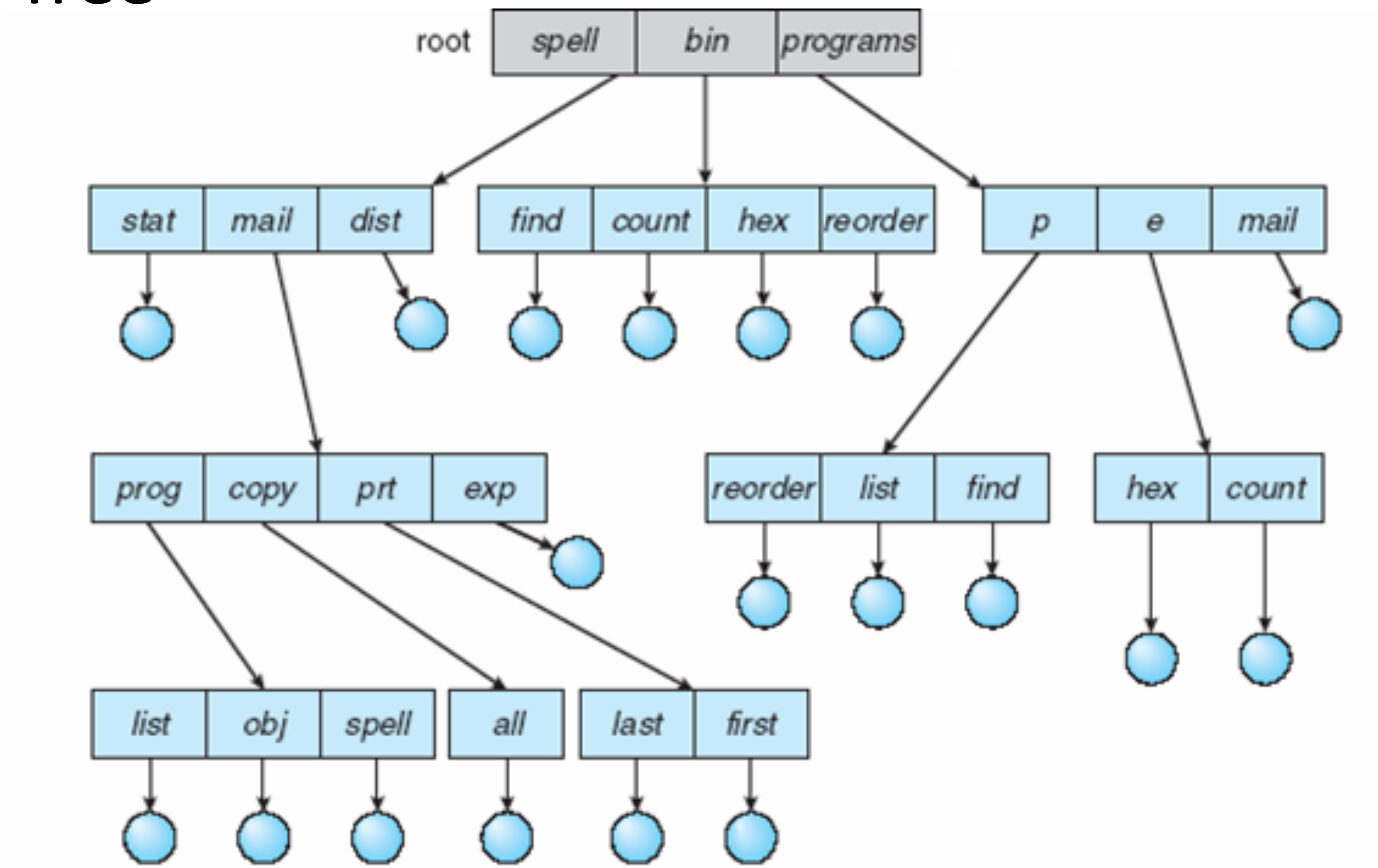
- Insert(string, uid)
- uid = Lookup(string)
- Remove(string, uid)

Directory Primitives

- CreateDirectory(string)
- DeleteDirectory(string)
- SetWorkingDirectory(string)
- string = ListWorkingDirectory()
- List(directory)

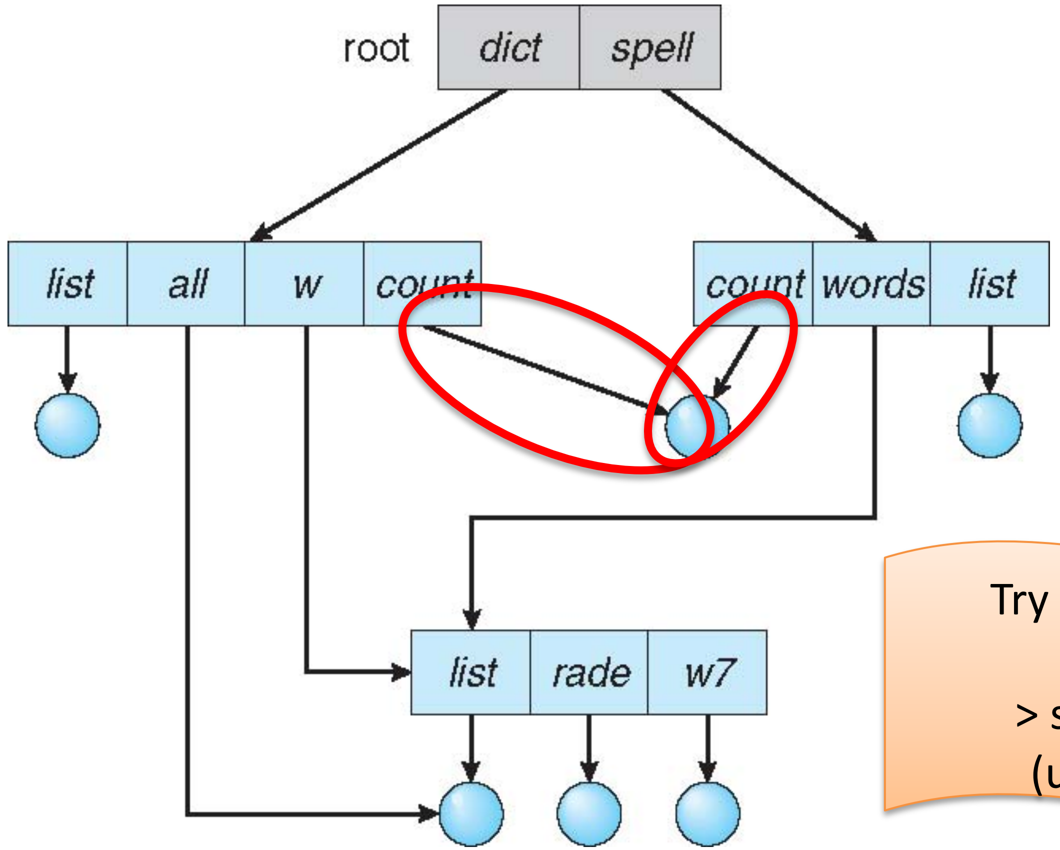
Hierarchical Directory Structures

- Tree



Hierarchical Directory Structures

- (Acyclic) Graph
  - Allows sharing of *uids* under different names



Try (assuming foo file exists)
> strace rm foo
(unlinkat call)

Hard Link

- Assume mapping (string1, uid) already exists
- HardLink(string2, uid)

- Insert(string2, uid)

- After HardLink, two mappings are equivalent

Try (assuming foo file exists)

> ln foo bar

> cat bar

> ls –i foo bar

Soft Link

- Assume mapping (string1, uid) exists
- SoftLink(string2, string1)

- Insert(string2, string1)

- After SoftLink, two mappings are different

Try (assuming foo file exists)

> ln -s foo bar

> cat bar

> stat foo bar; ls -al

Hard/Soft Link Differences

- HardLink(string2, uid)
- Remove(string1, uid)

- Mapping (string2, uid) remains

Hard/Soft Link Differences

- SoftLink(string2, string1)
- Remove(string1, uid)

- Mapping (string2, string1) dangling reference

Why Keep Graph Acyclic?

- (Later) disk storage reclamation by refcounts
- Cycles cause wasted disk space

How to Keep Graph Acyclic

- Soft links cannot make cycles
- Hard links can make cycles
- Do not allow hard links to directories, only to leafs in the graph

Linux Primitives

- Collapses in a single interface
  - Access
  - Concurrency
  - Naming

Linux

- Creat(string)
  - uid = Create()
  - Insert(string, uid)
- fd = Open(string, [optional args])
  - uid = Lookup(string)
  - fd = (tid =) Open(uid, [optional args])
- …
- *uid* is never visible at the user level

Careful with Links

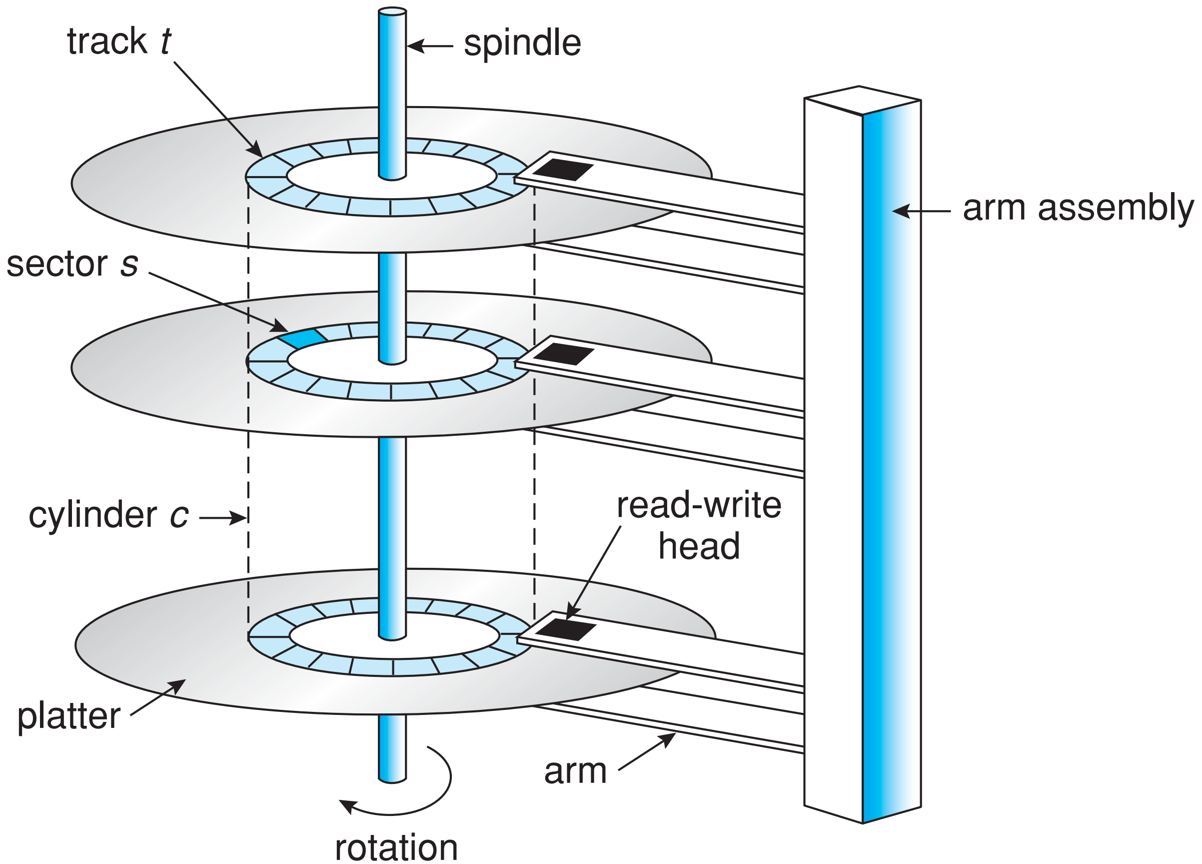
- HardLink(string2, string1)
- SoftLink(string2, string1)
- Look very similar
- Are very different
  - HardLink is a mapping to a file
  - SoftLink is a mapping to a string

Summary

- Permanent storage
- File
- File system primitives
  - Access, concurrency, naming (, protection)
- Linux file system primitives

Disk

Disk


track t
spindle
arm assembly
sector s
cylinder c
read-write
head
platter
arm
rotation

Disk Terminology - Mechanical

- Arm assembly
- Arm
- Read/write head
- Platter

Disk Terminology - Information

- Sector
- Track
- Cylinder

Disk Characteristics

- Size – typically from 1.8” to 3.5”
- From tens of GB to a few TB

Disk Interface

- Accessible by sector only

- ReadSector (logical_sector_number, buffer)
- WriteSector(logical_sector_number, buffer)

- Logical_sector_number =
  - Platter
  - Cylinder or track
  - Sector

A Look Ahead at File System Implementation

- The main task of the file system is to translate

- From user interface methods
- Read(uid, buffer, bytes)

- To disk interface methods
- ReadSector(logical_sector_number, buffer)

Two Small Simplifications - 1

- User Read() allows arbitrary number of bytes
- Simplify to only allowing Read() of a block
  - Read(uid, block_number)
- A block is fixed-size

Two Small Simplifications - 2

- Typically
  - Block size = 2^n * sector size
- For instance
  - Block size = 4,096 bytes
  - Sector size = 512 bytes
- For simplicity of presentation in class
  - Block size = sector size

Two Small Simplifications

- Both of these easily implemented in libraries

Back to Disk Interface

- Accessible by sector only

- ReadSector (logical_sector_number, buffer)
- WriteSector(logical_sector_number, buffer)

- Logical_sector_number =
  – Platter
  – Cylinder or track
  – Sector

Disk Access

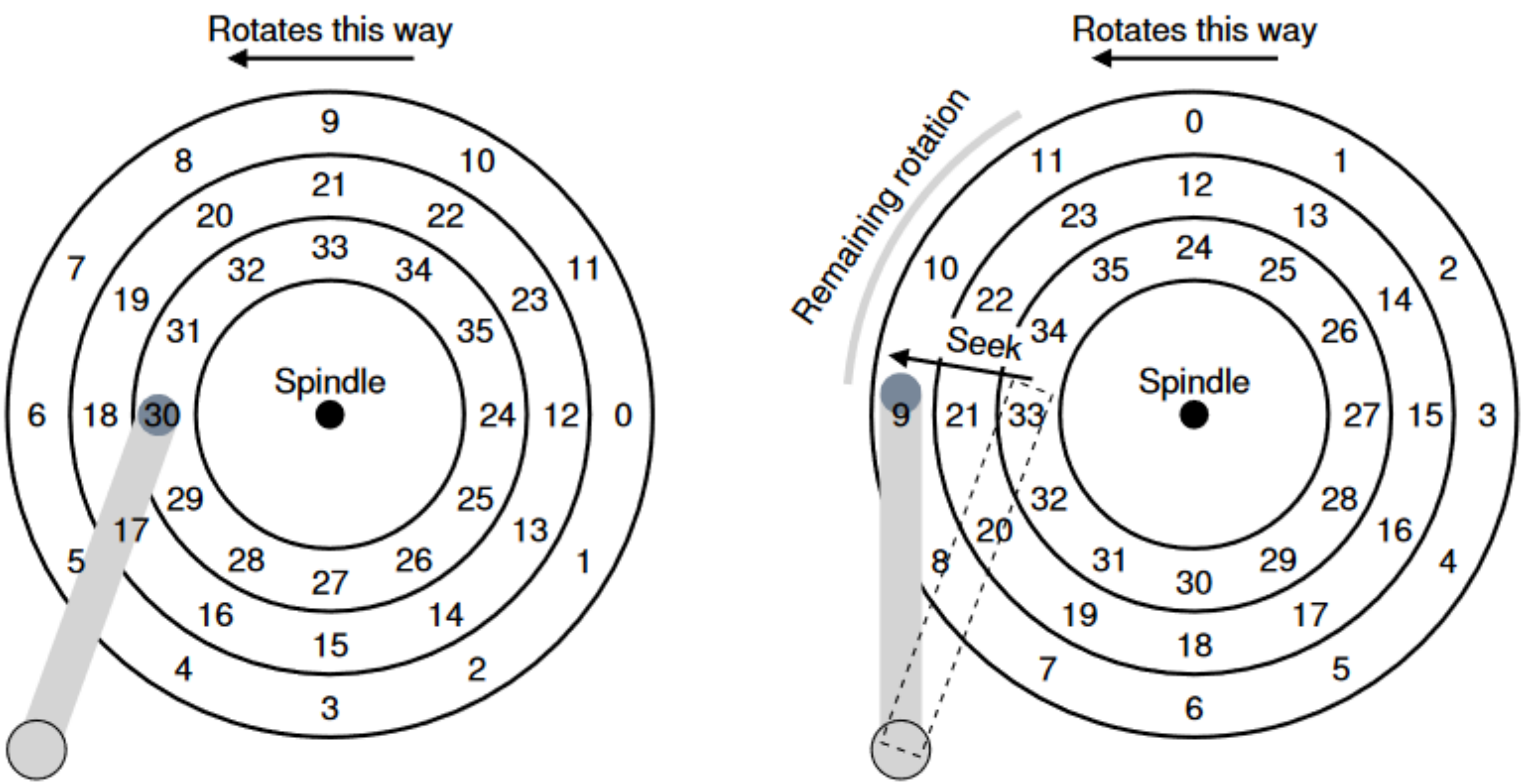
- Head selection – select platter
- Seek – move arm over cylinder
- Rotational latency – move head over sector
- Transfer time – read from sector

Disk Access Time – Head Selection

- Electronic switch
- ~ nanoseconds

Disk Access Times – Seek Time

- Approx. linear in the number of cylinders
- 3 to 12 milliseconds



Disk Access Time – Rotational Latency

- Linear in the number of sectors
- Rotational speed: 4,500 -15,000 RPM
- One revolution = 1 / (RPM/60) seconds
- Average rotational latency = ½ revolution
- From 2 to 7.1 milliseconds

Disk Access Time - Transfer

- Effective transfer rate ~ 1 GB per second
- Sector = 512 B
- Transfer time ~ 0.5 microseconds

Disk Access Time

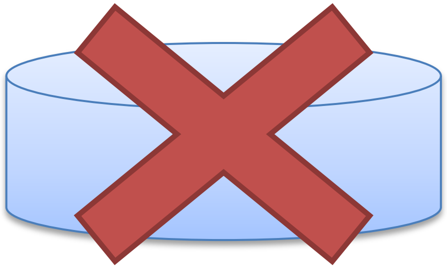
Component	Time
Head Selection	nanoseconds
Seek Time	3-12 milliseconds
Rotational Latency	2-7 milliseconds
Transfer Time	microseconds
Controller Overhead	< 1 millisecond

Disk Access Time - Observations

- Disk access time >> memory access time
- Seek time dominates
- Followed by rotational latency

Optimize Disk Access

- Rule 1: Do not access disk
- Use a cache



File System Cache (Buffer Cache)

- What?
  - Keep recently accessed blocks in memory
- Why?
  - Reduce latency
  - Reduce disk load
- How?
  - Reserve kernel memory for cache
  - Cache entries: file blocks (of block size)

Read with a Cache

- If in cache
  - Return data from cache
- If not
  - Find free cache slot
  - Initiate disk read
  - When disk read completes, return data

Write with a Cache

- Always write in cache (8/16MB)
- How does it get to disk?
  - Write-through
  - Write-behind

Write-Through

- Write to cache
- Write to disk
- Return to user

Write-Behind

- Write to cache
- Return to user
- Later: write to disk

Write-Through vs. Write-Behind

- Response time:
  - Write-behind is (much) better
- Disk load:
  - Write-behind is (much) better
  - Much data overwritten before it gets to disk
- Crash:
  - Write-through is much better
  - No “window of vulnerability”

In Practice

- Write-behind
- Periodic cache flush
- User primitive to flush data

Optimize Disk Access - 2

- Rule 2: Don’t wait for disk
- Read-ahead (or prefetching)
- Only for sequential access

Read-Ahead

- What?
  - User request for block i of a file
  - Also read block i+1 from disk
- How?
  - Put block i+1 in the buffer cache
- Why?
  - No disk I/O on (expected) user access to block i+1

Read-Ahead

- Works for sequential access

- Most access is sequential
- In Linux it is the default

Caveat about Read-Ahead

- Does not reduce number of disk I/Os
- In fact, could increase them (if not sequential)

- In practice, very often a win
- Linux always reads one block ahead

Optimize Disk Access - 3

- Rule 3: Minimize seeks
- Two approaches:
  - Clever disk allocation
    - Locate related data (same file) on same cylinder
  - Clever scheduling
    - Reorder requests to seek as little as possible

Clever Disk Allocation

- Allocate “related” blocks “together”

- “together”
  - On the same cylinder
  - On a nearby cylinder
- “related”
  - Consecutive blocks in the same file
  - Sequential access

Disk Scheduling

- FCFS – First-Come-First-Served
- SSTF – Shortest-Seek-Time-First
- SCAN
- C-SCAN
- LOOK
- C-LOOK

Disk Scheduling Illustration

- Initial position of the head = cylinder 53
- Queue of requests:

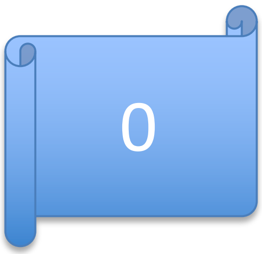
    98, 193, 37, 122, 14, 124, 65, 67

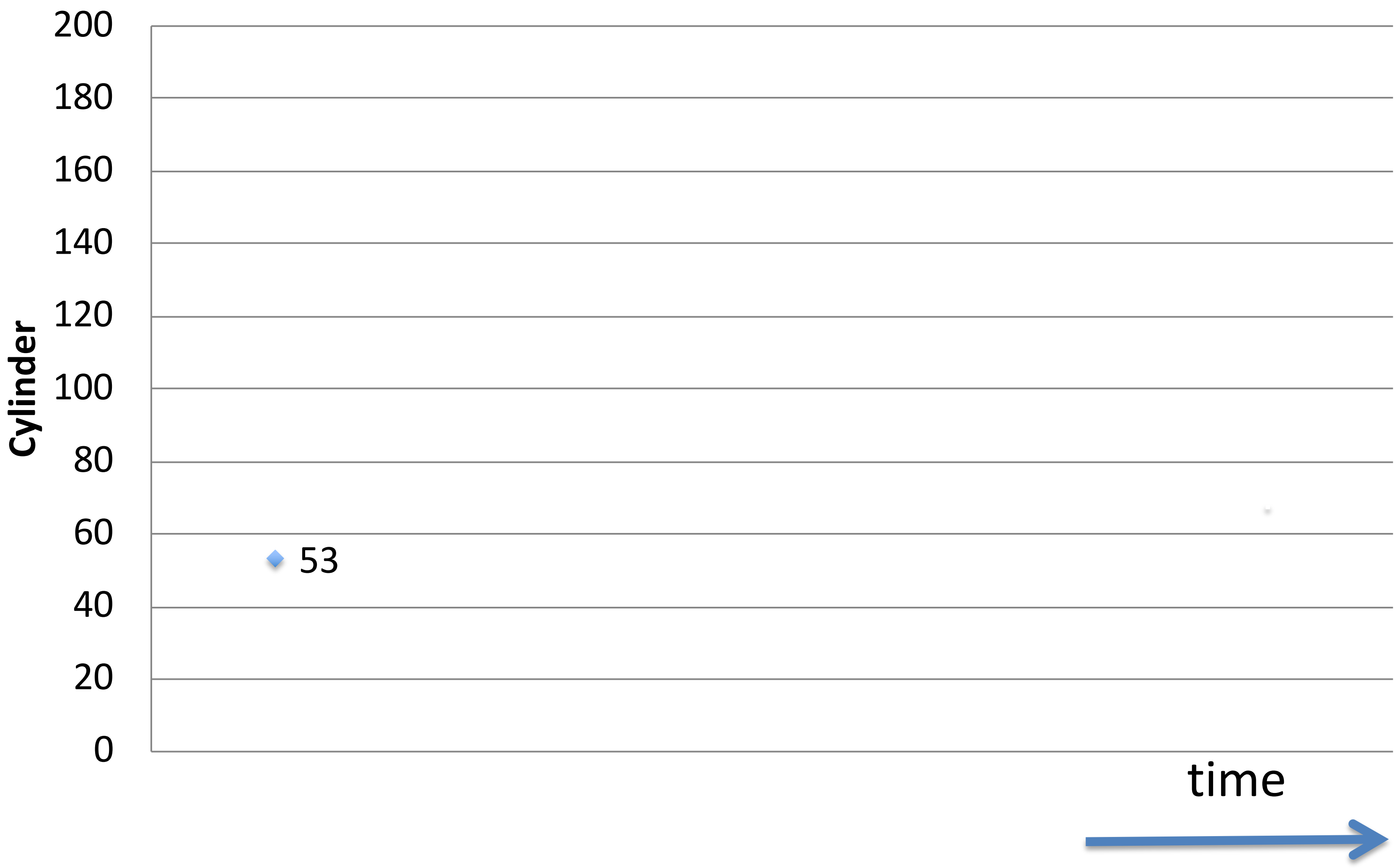
FCFS

- Next request in the queue

Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

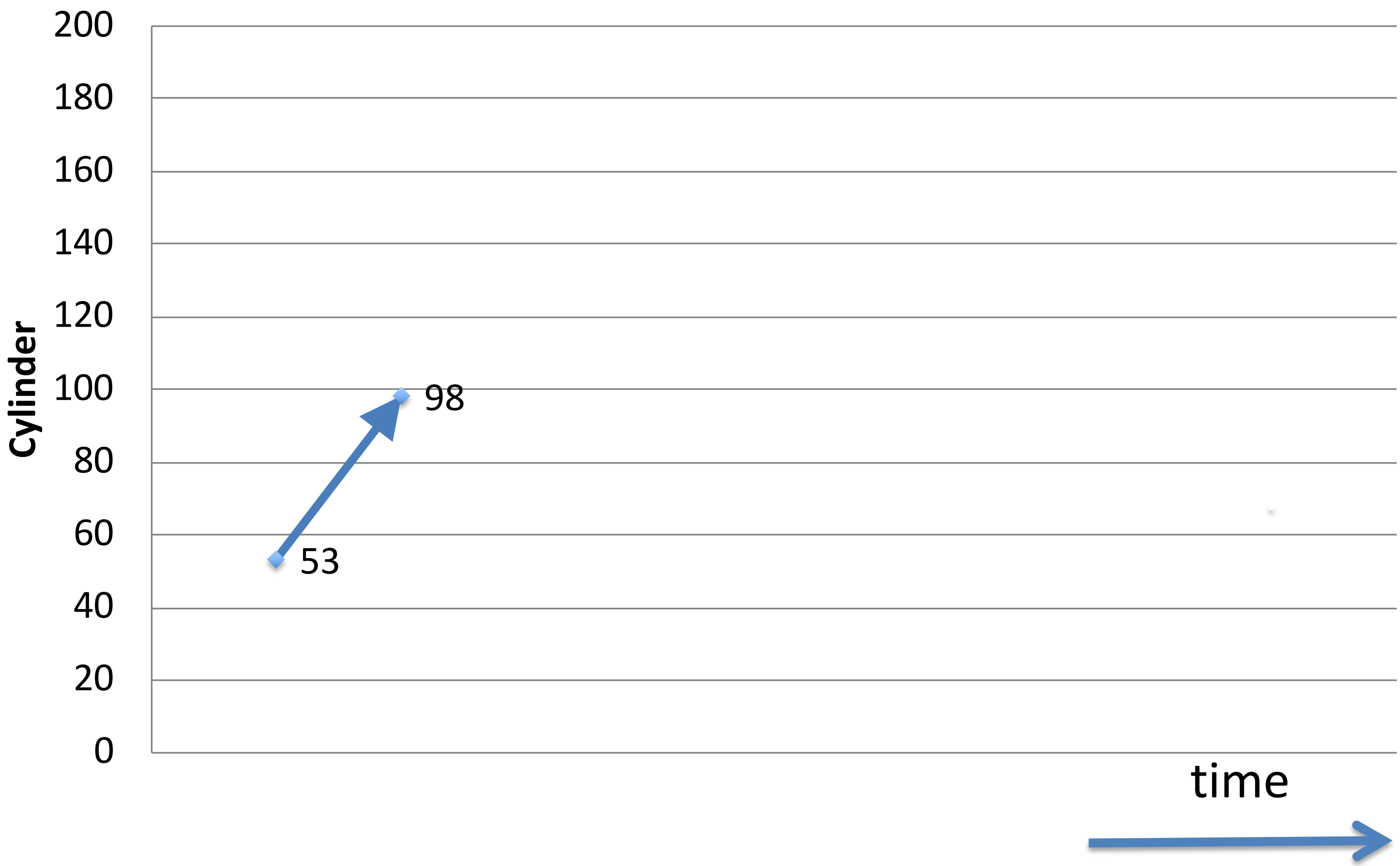




Head = 53

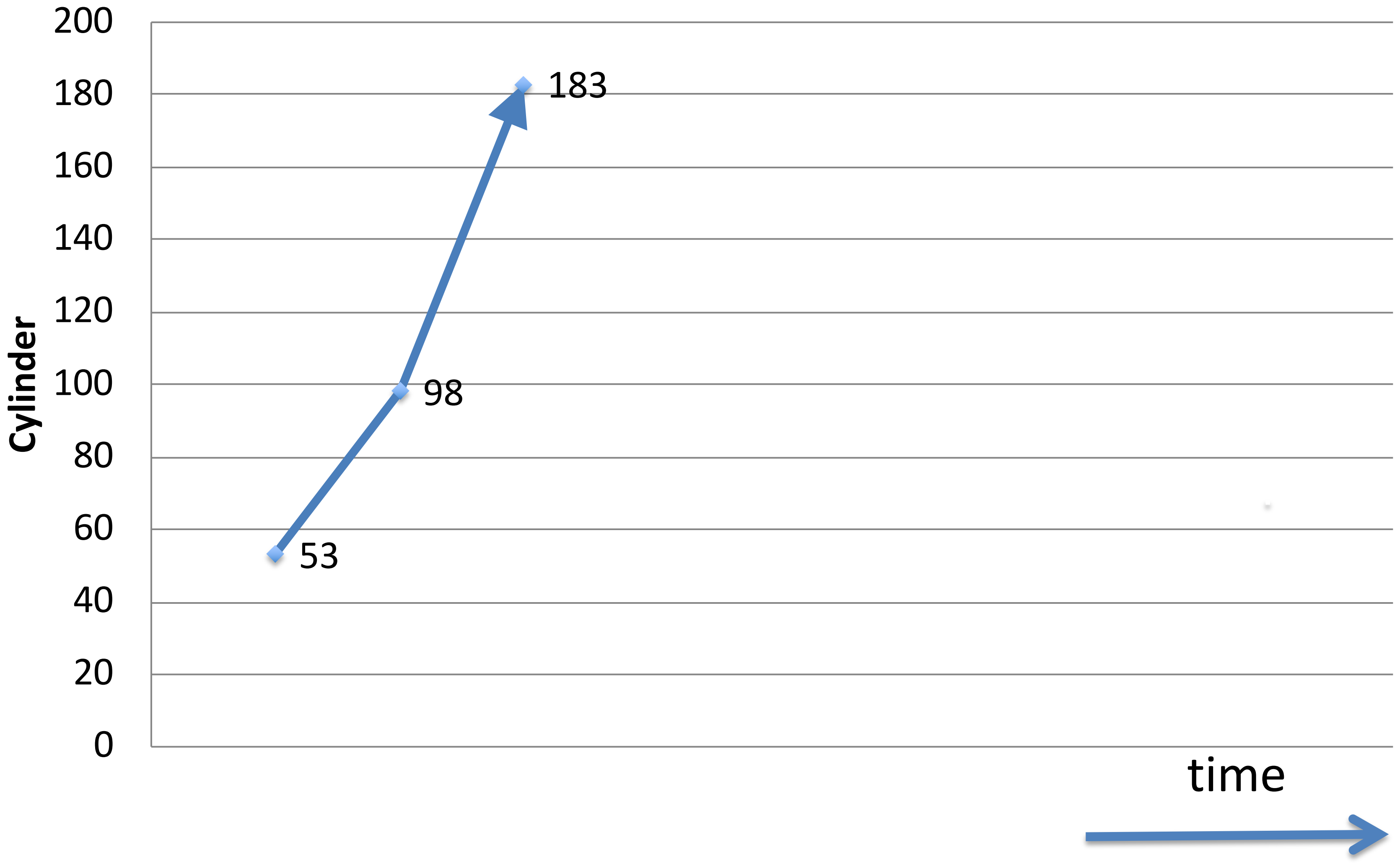
Queue = 98, 183, 37, 122, 14, 124, 65, 67

45



Head = 53

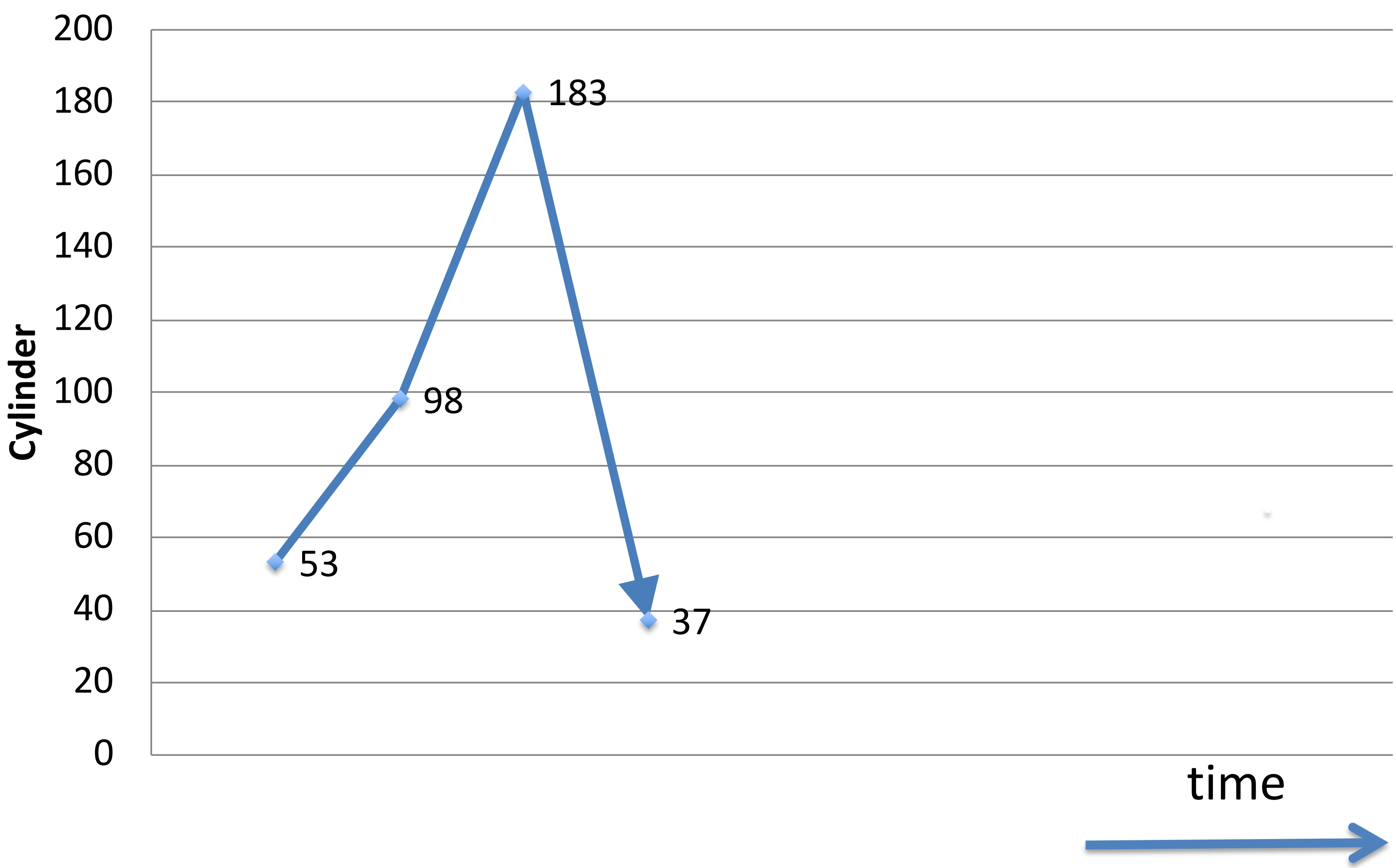
Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

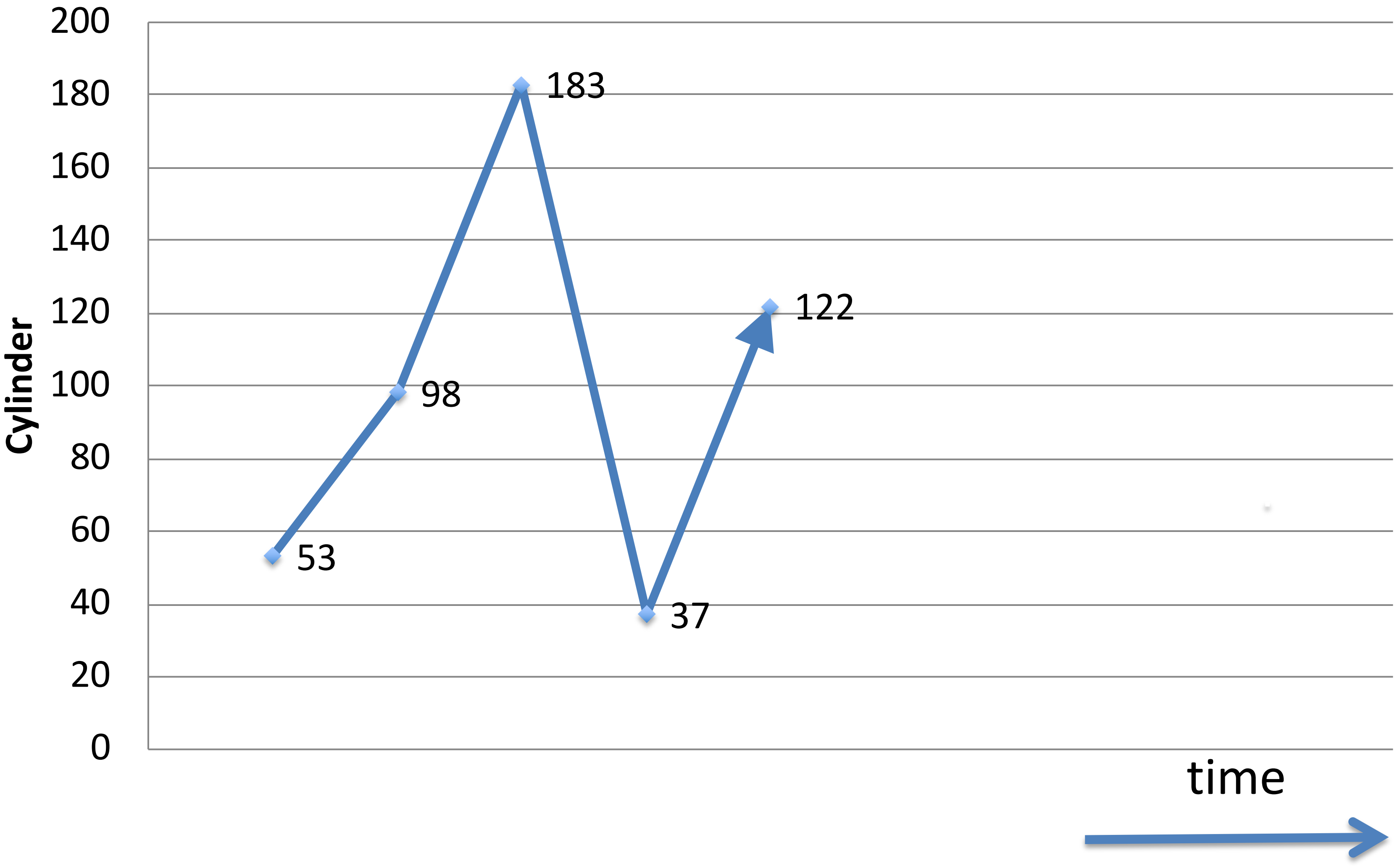
Queue = 98, 183, 37, 122, 14, 124, 65, 67

276



Head = 53

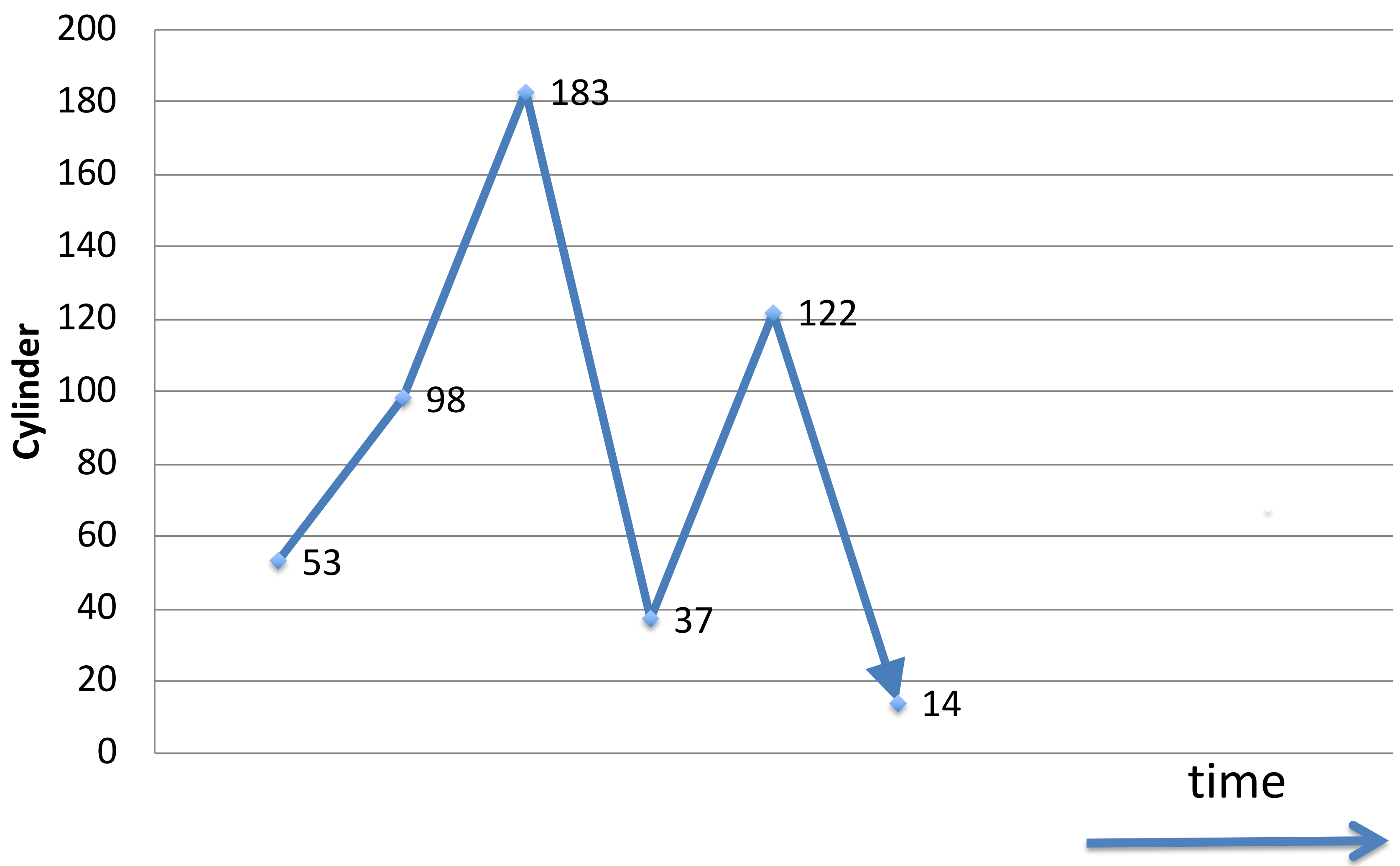
Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

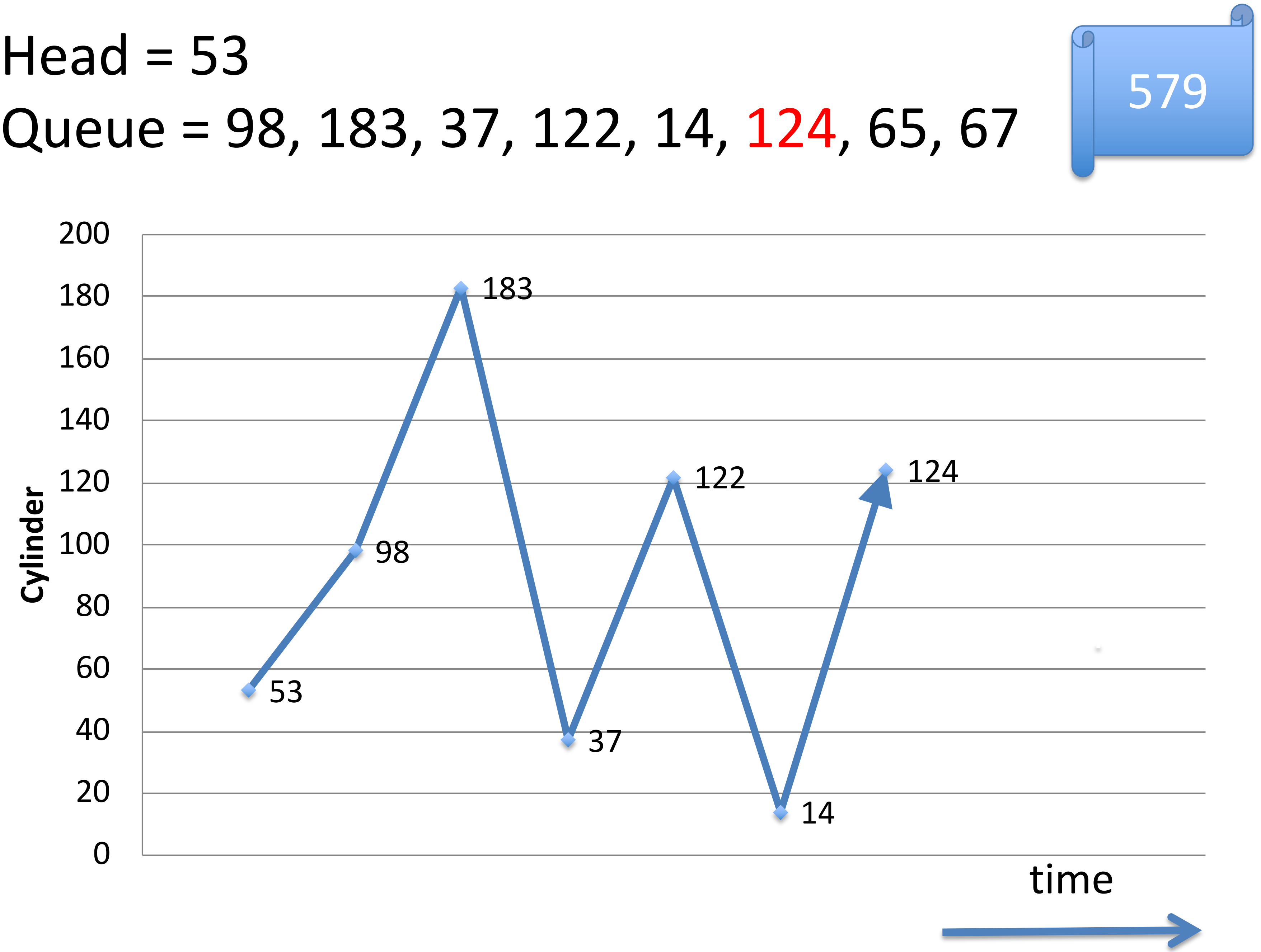
Queue = 98, 183, 37, 122, 14, 124, 65, 67

469



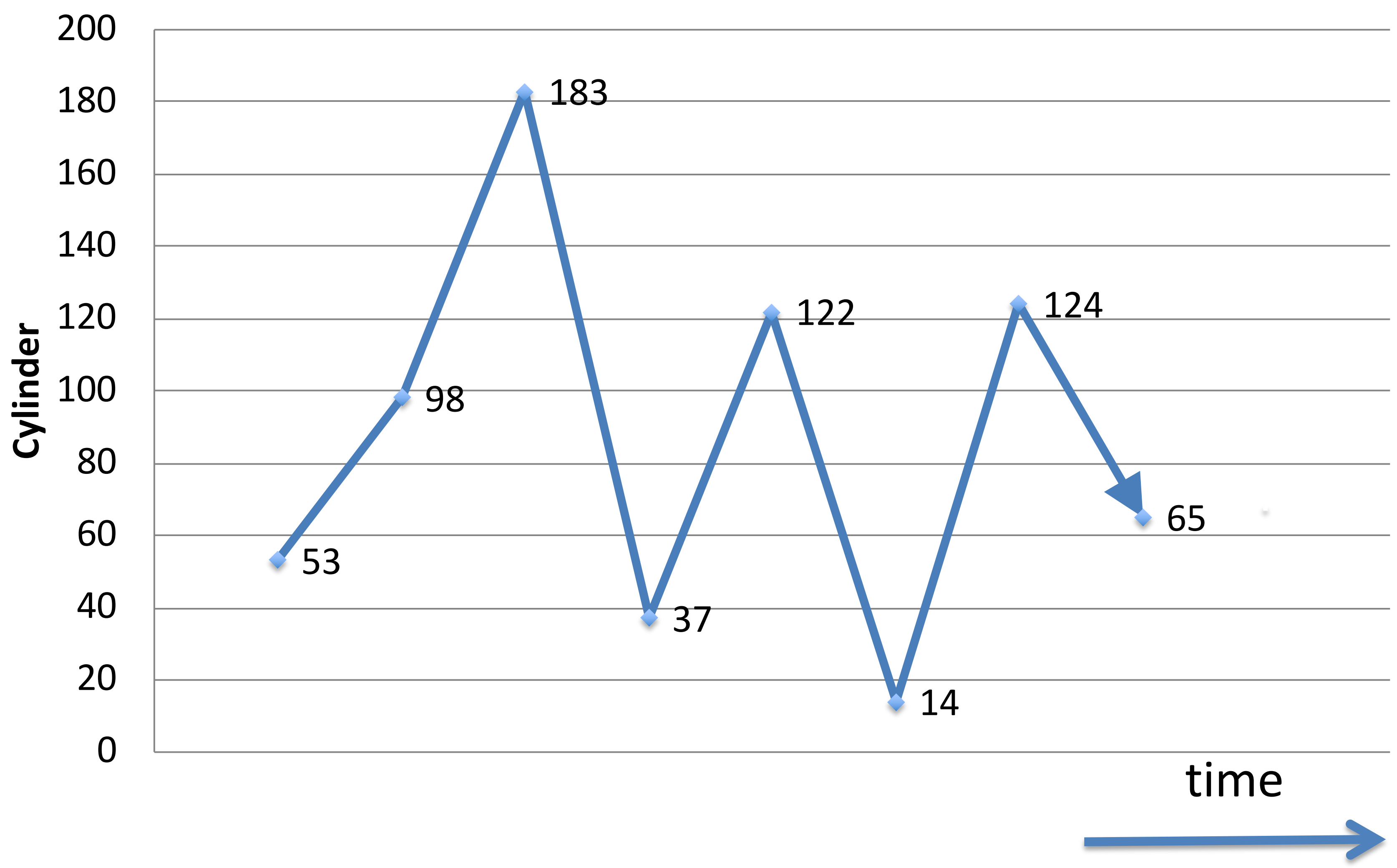
Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67



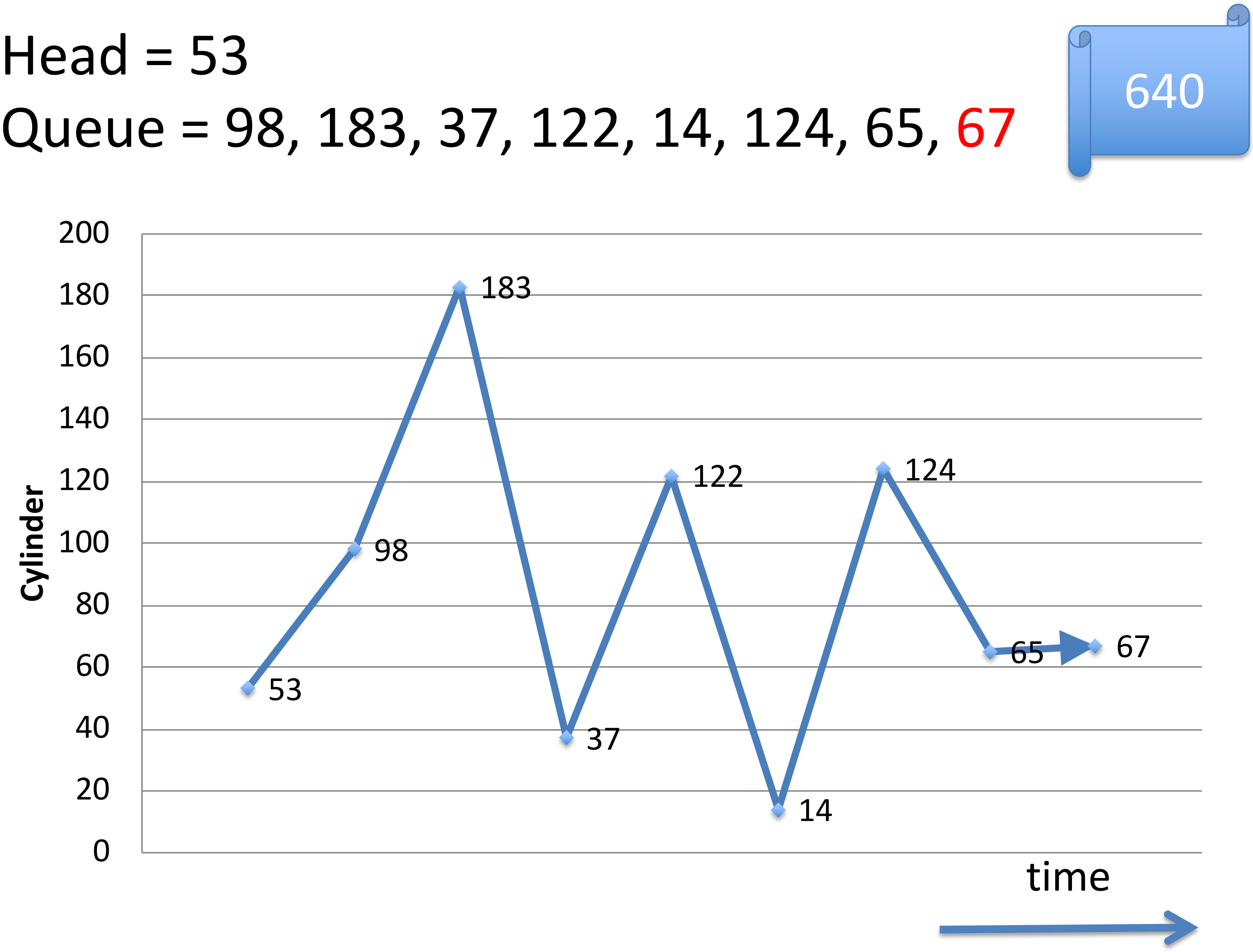
Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

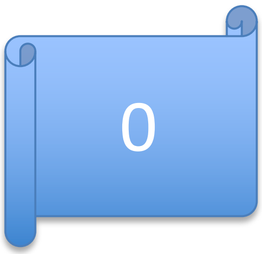


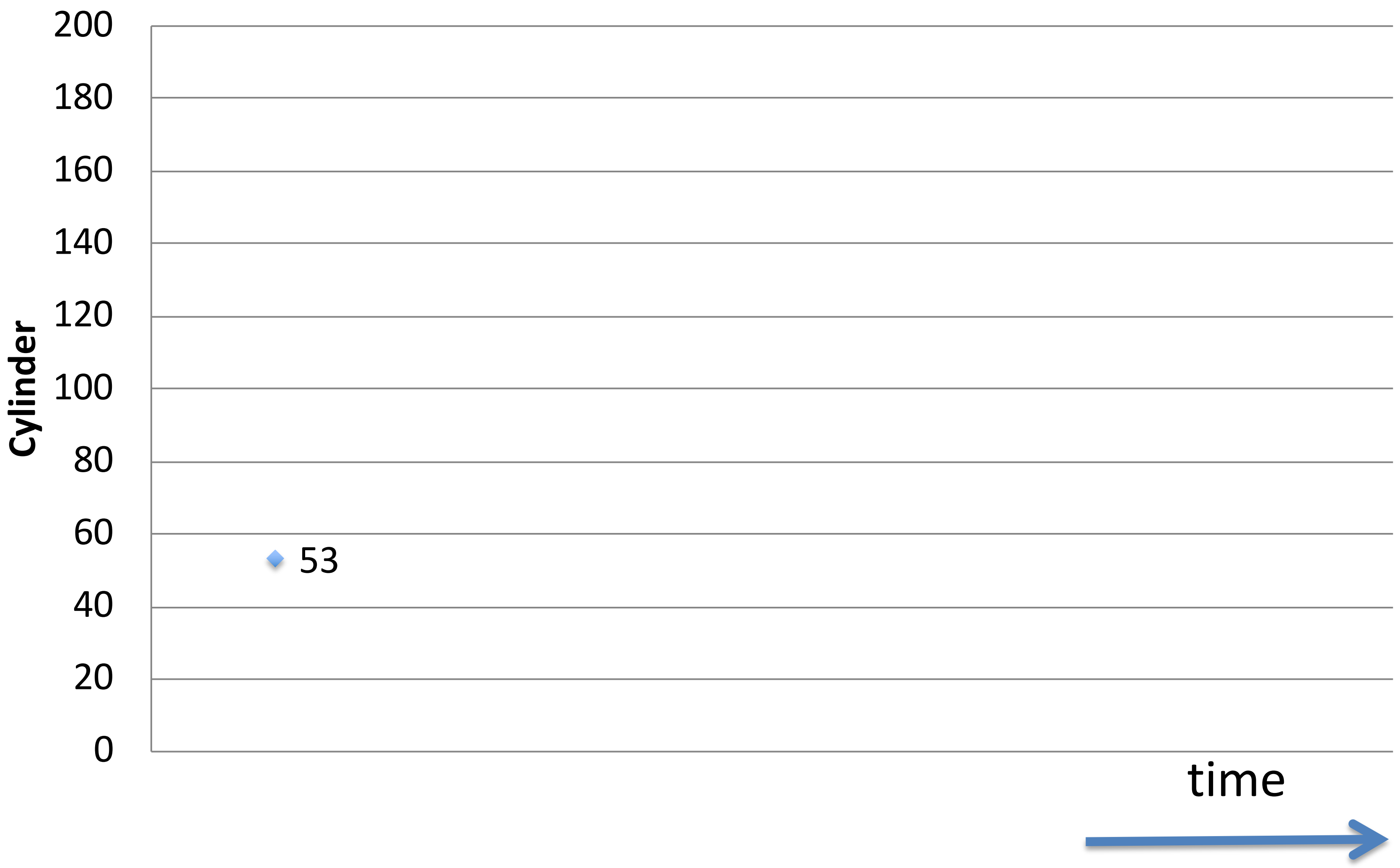
SSTF

- Shortest Seek Time First
- Pick “nearest” request in queue

Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

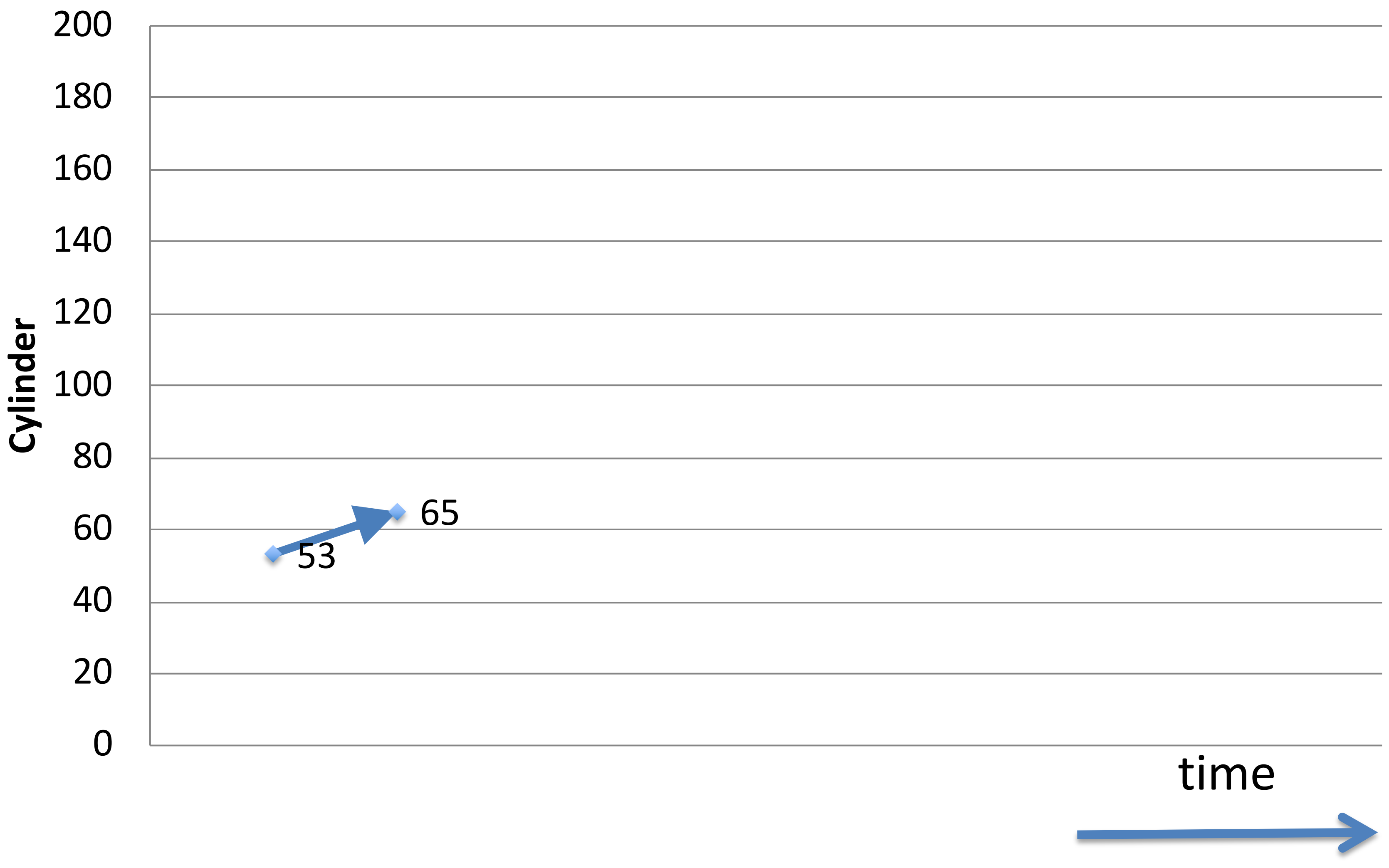




Head = 53

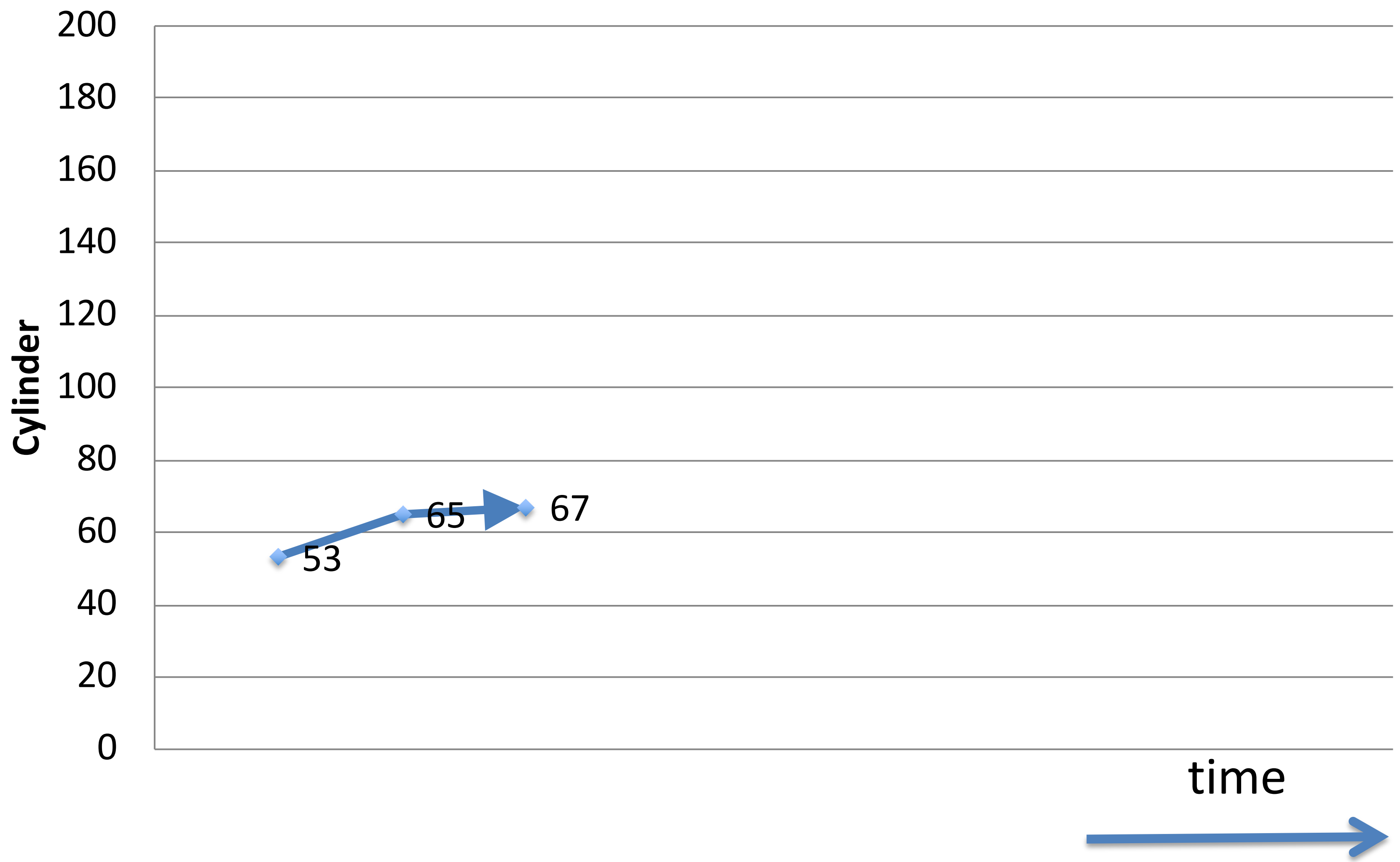
Queue = 98, 183, 37, 122, 14, 124, 65, 67

12



Head = 53

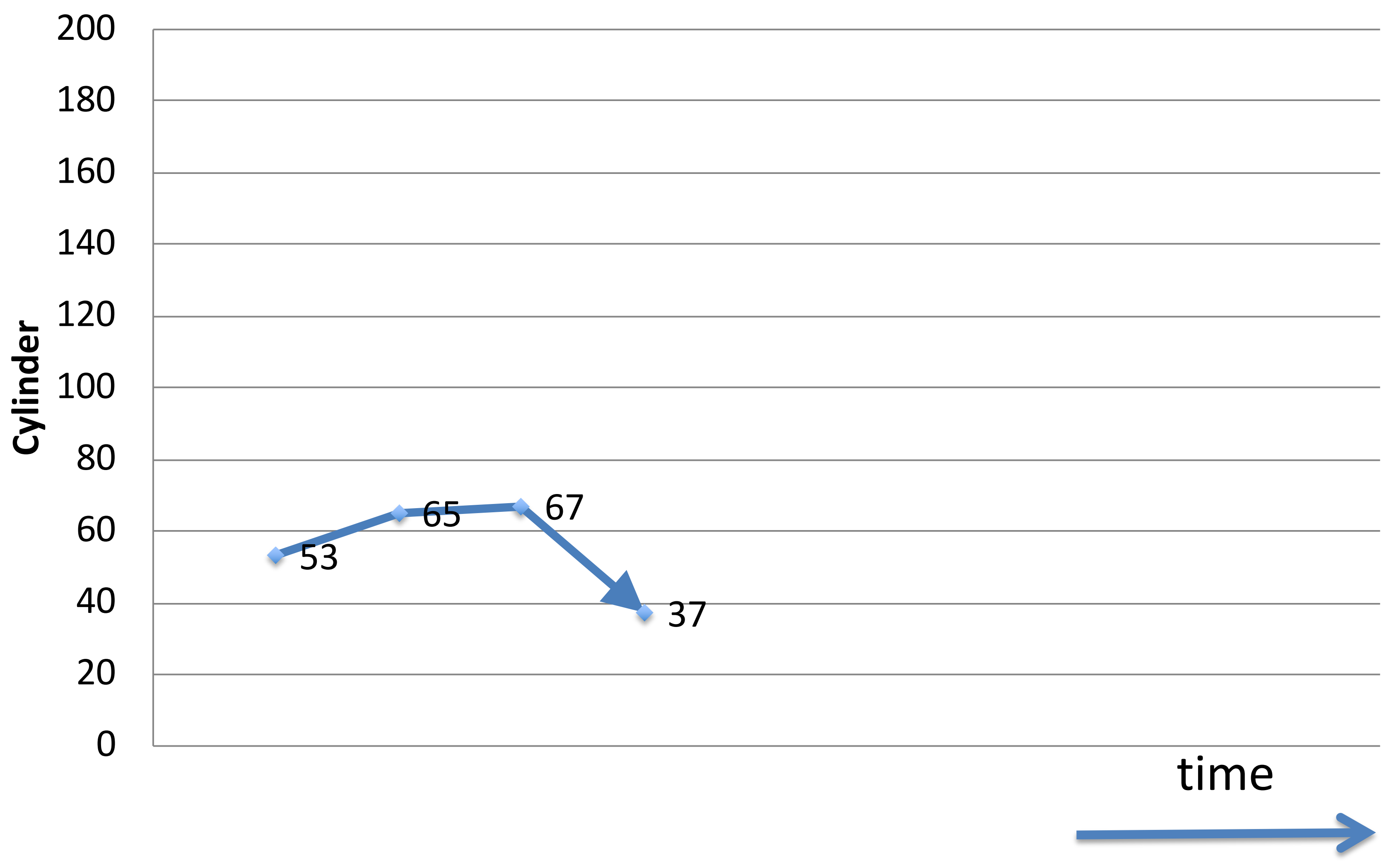
Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

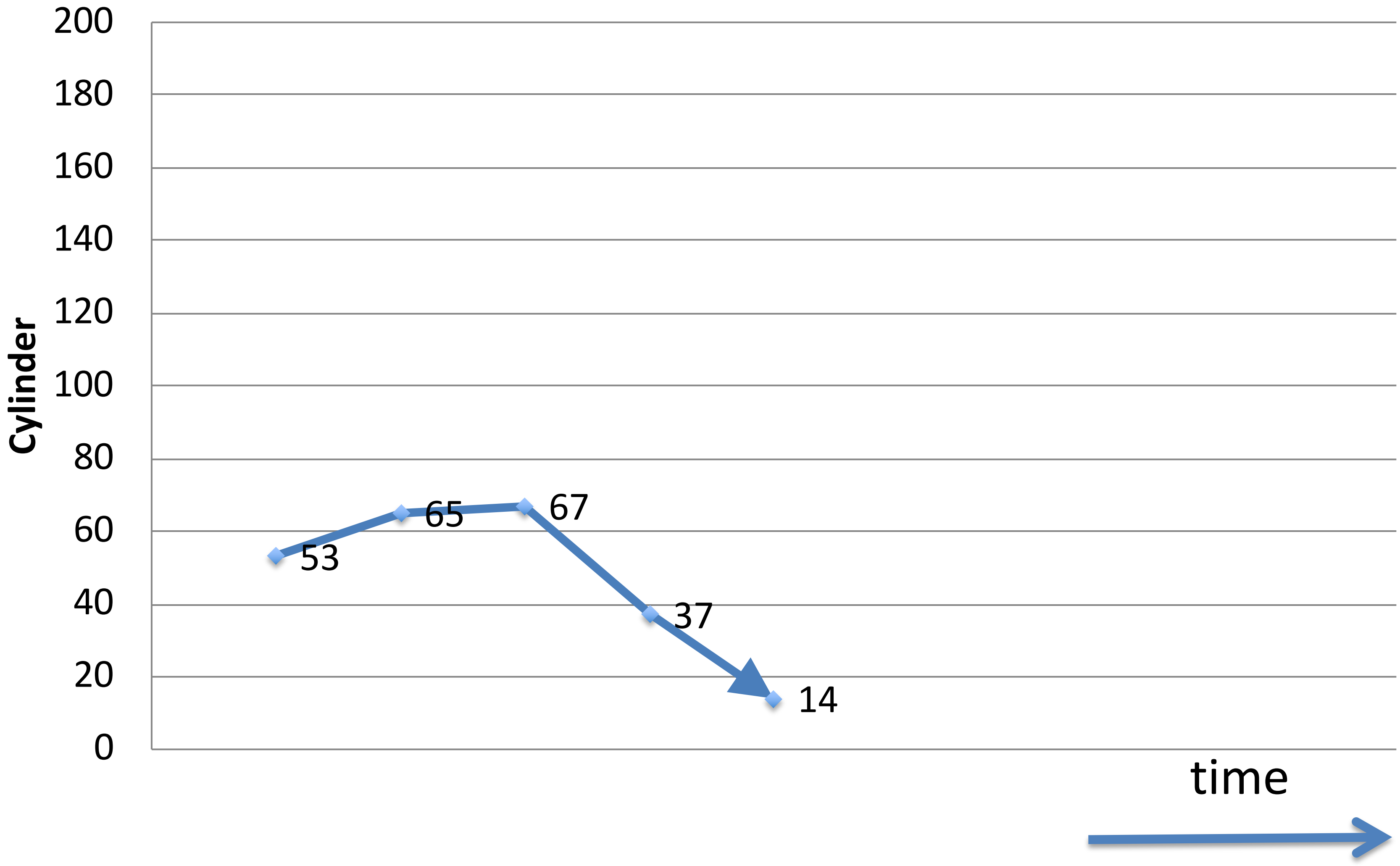
Queue = 98, 183, 37, 122, 14, 124, 65, 67

44



Head = 53

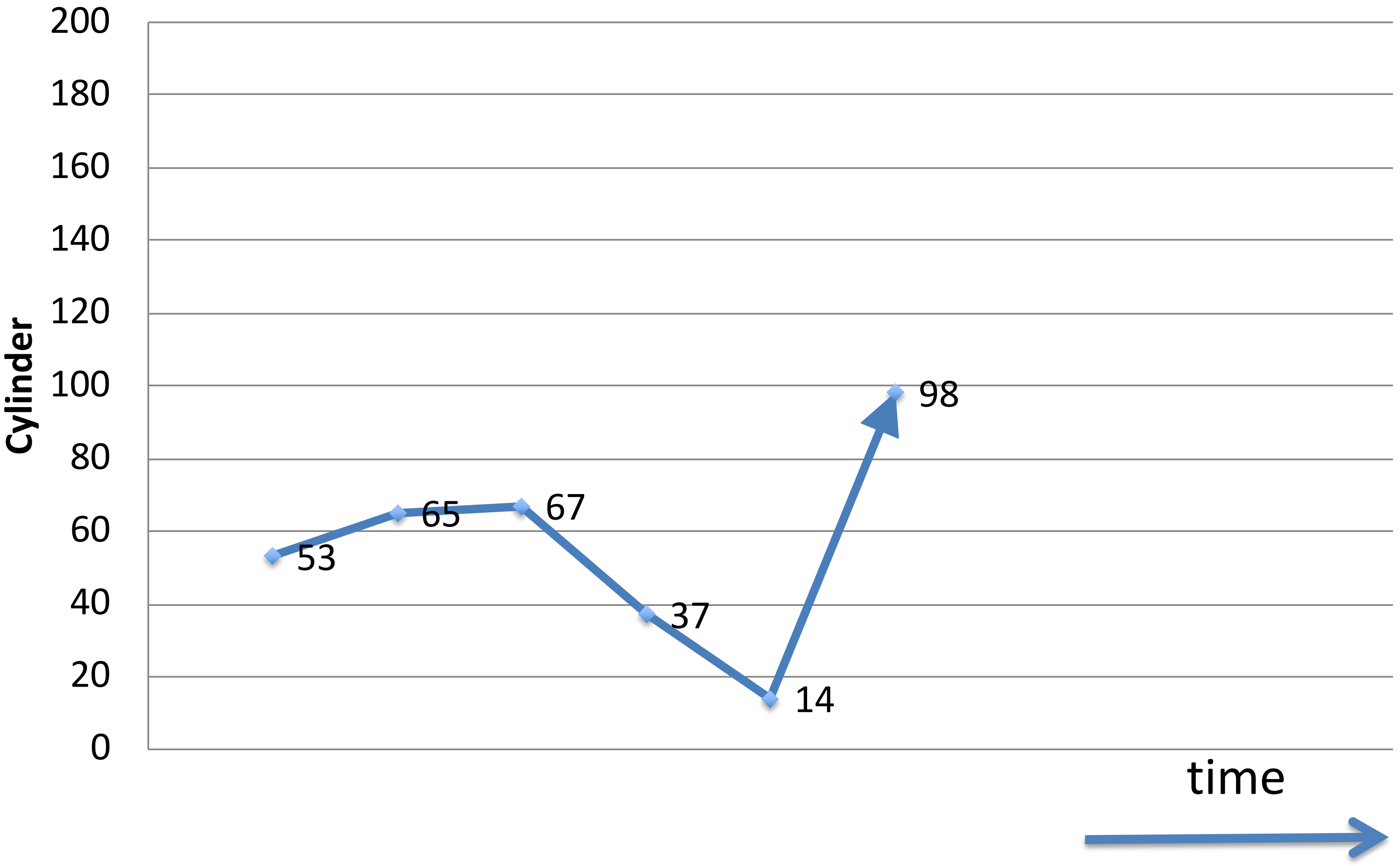
Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

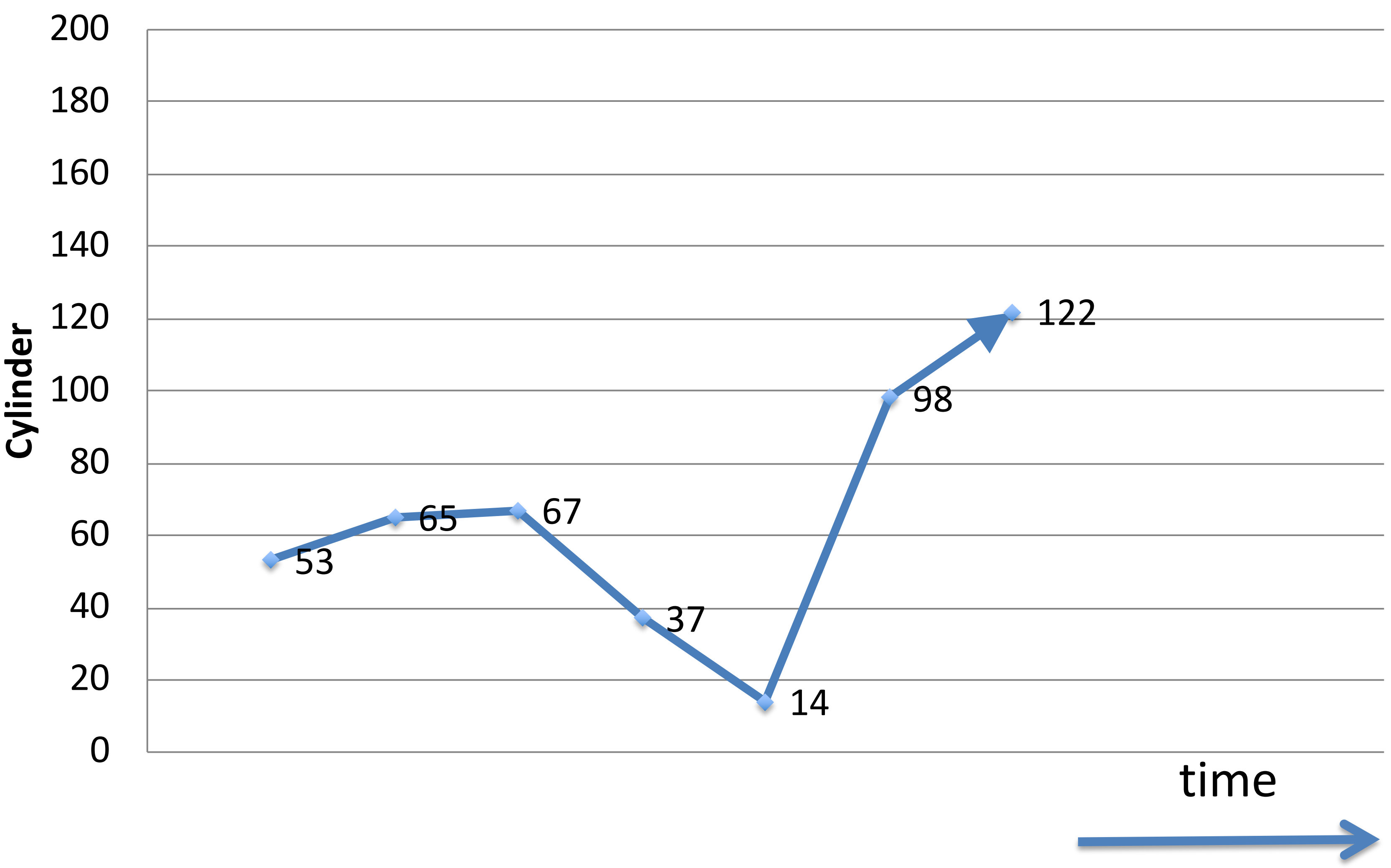
Queue = 98, 183, 37, 122, 14, 124, 65, 67

151



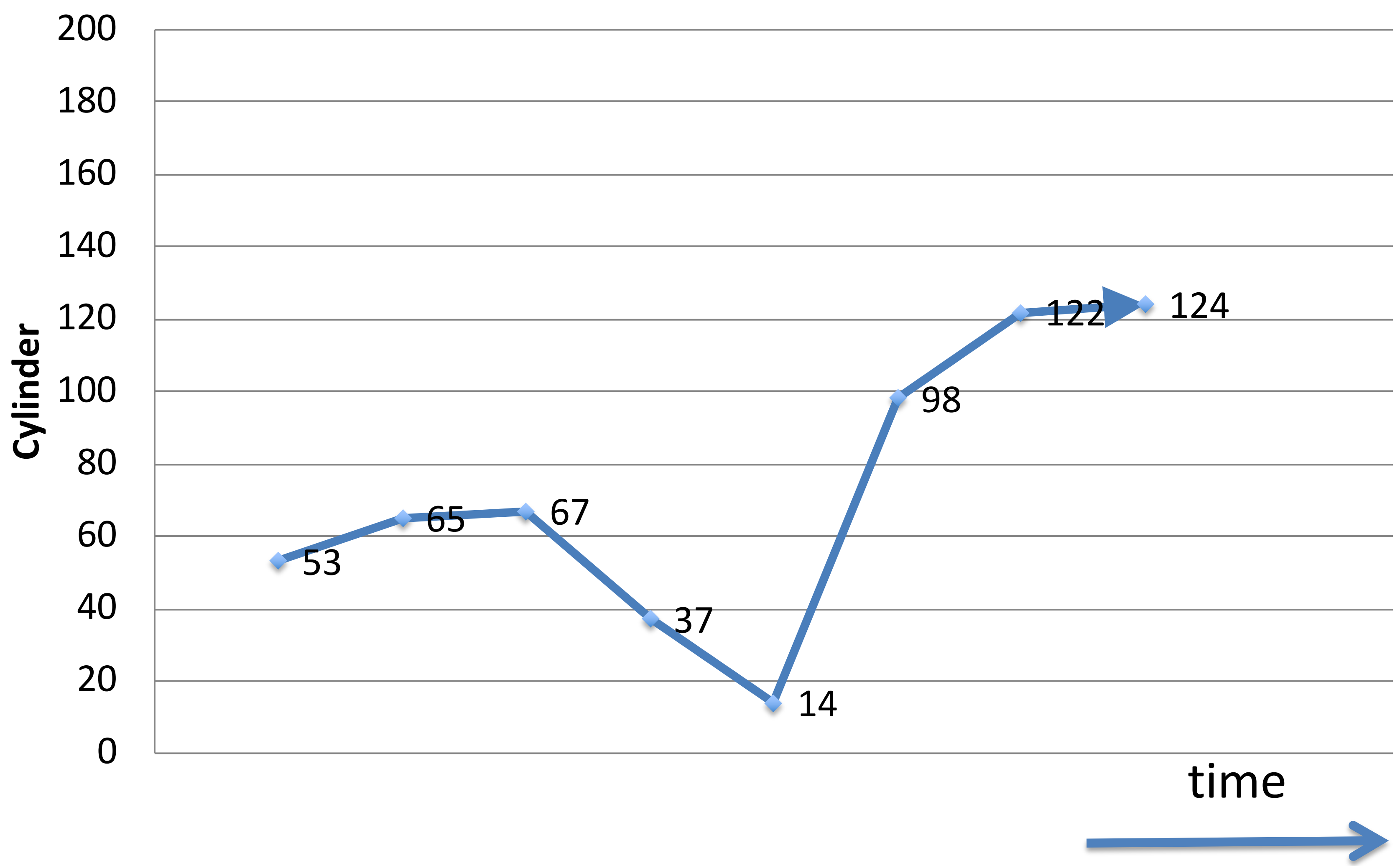
Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67



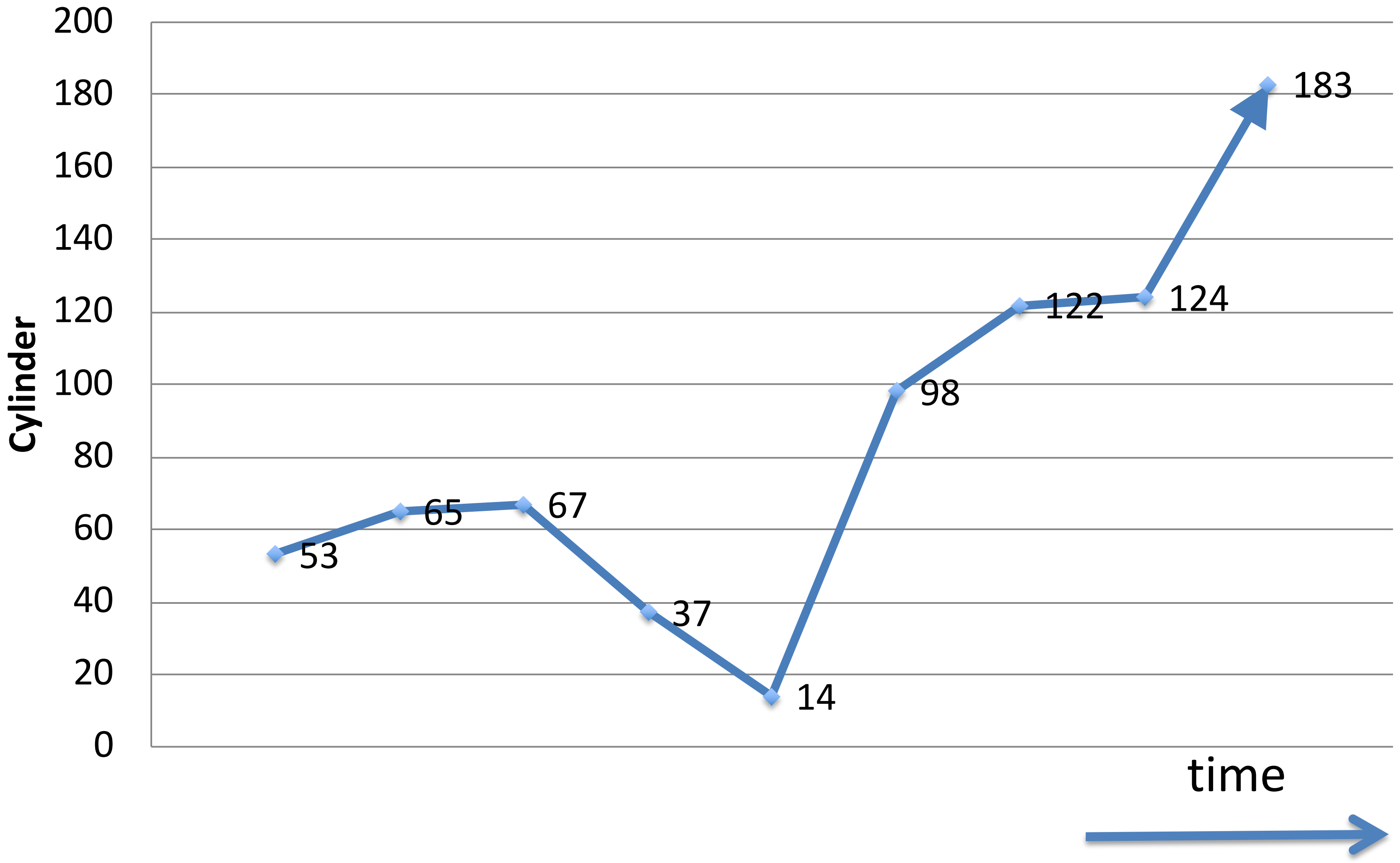
Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67



SSTF

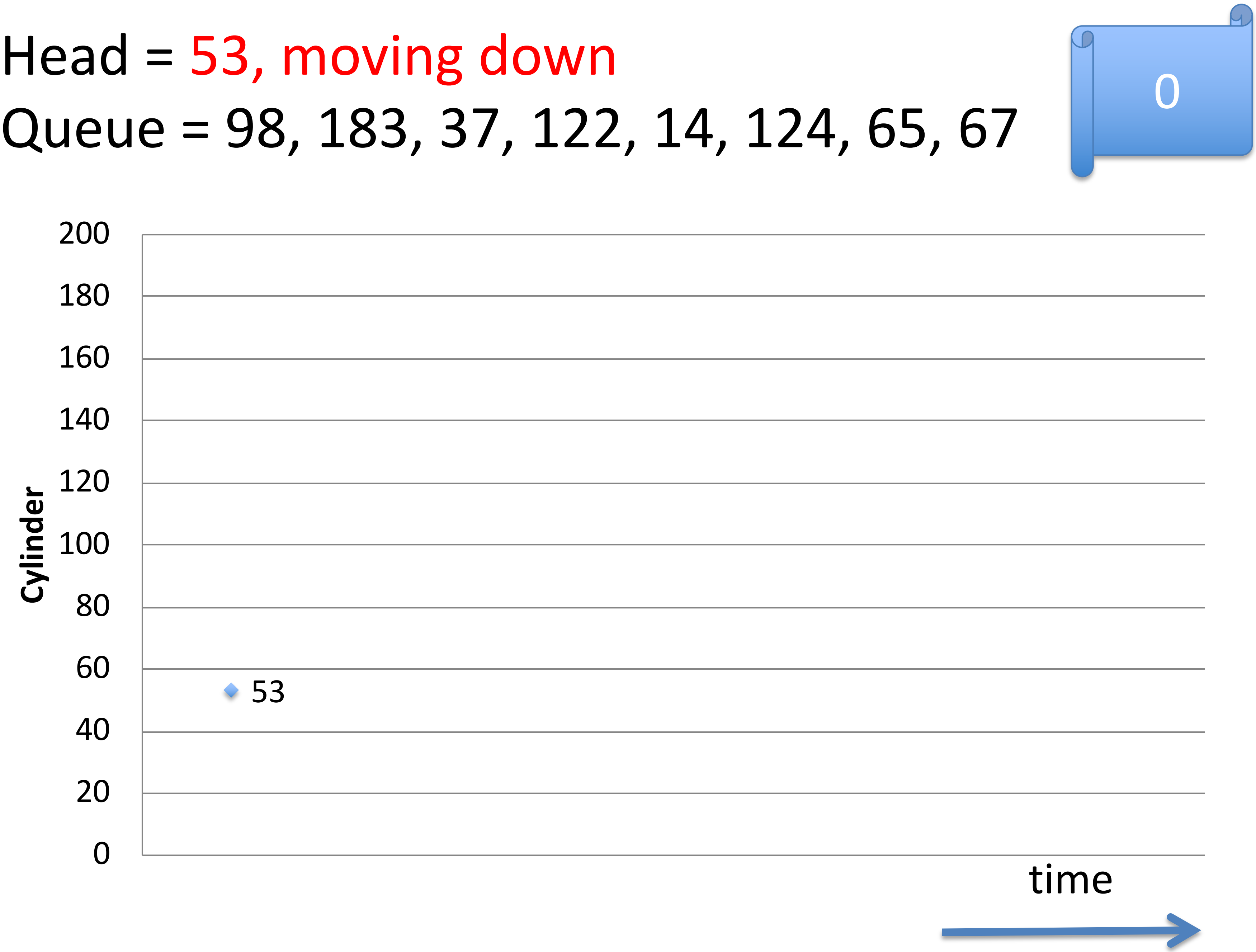
- Very good seek times
- Subject to starvation
  - Request on inside or outside can get starved

SCAN

- Continue moving head in one direction
  - From O to MAX_CYL
  - Then, from MAX_CYL to 0
- Pick up requests as you move head

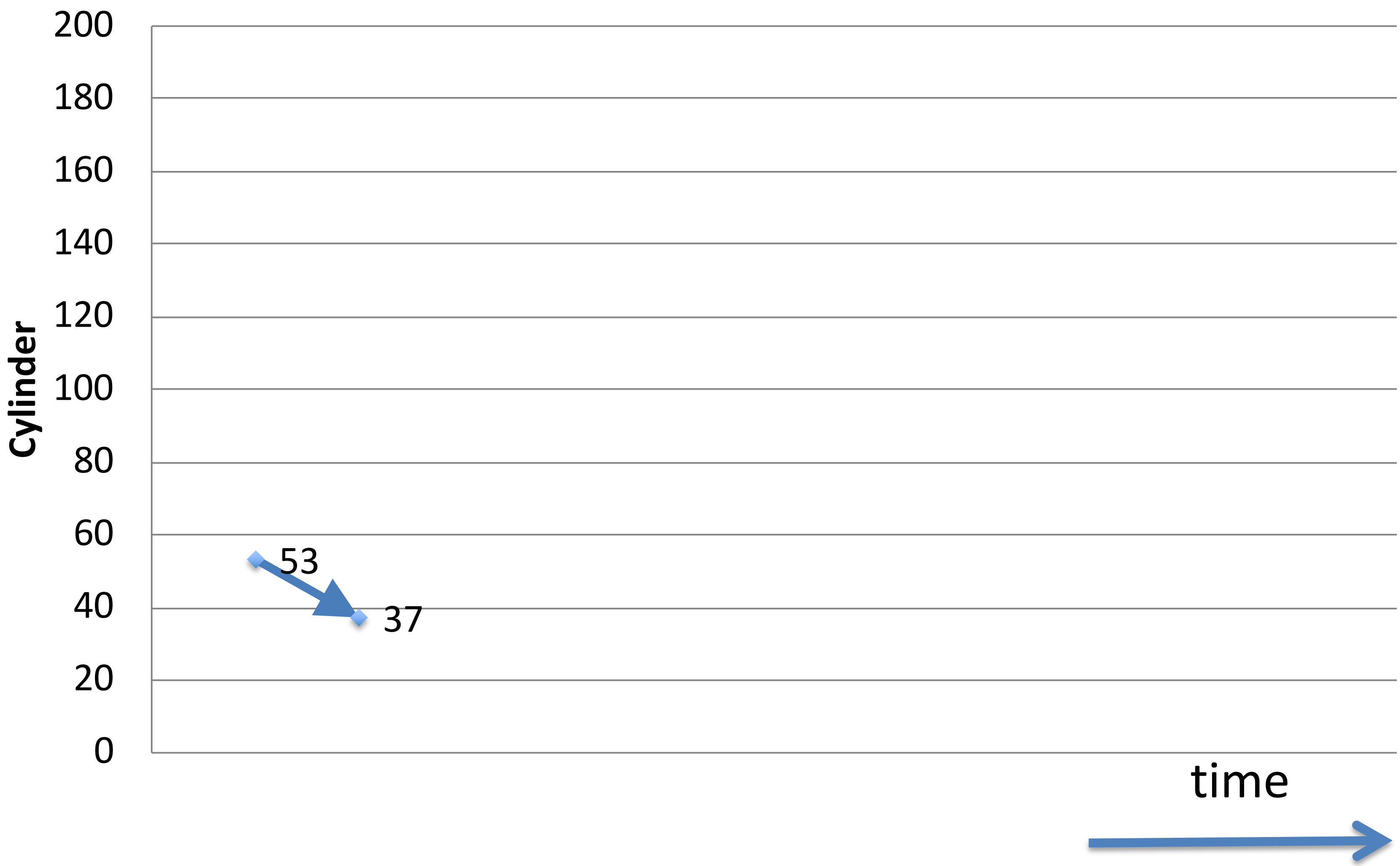
Head = 53, moving down

Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

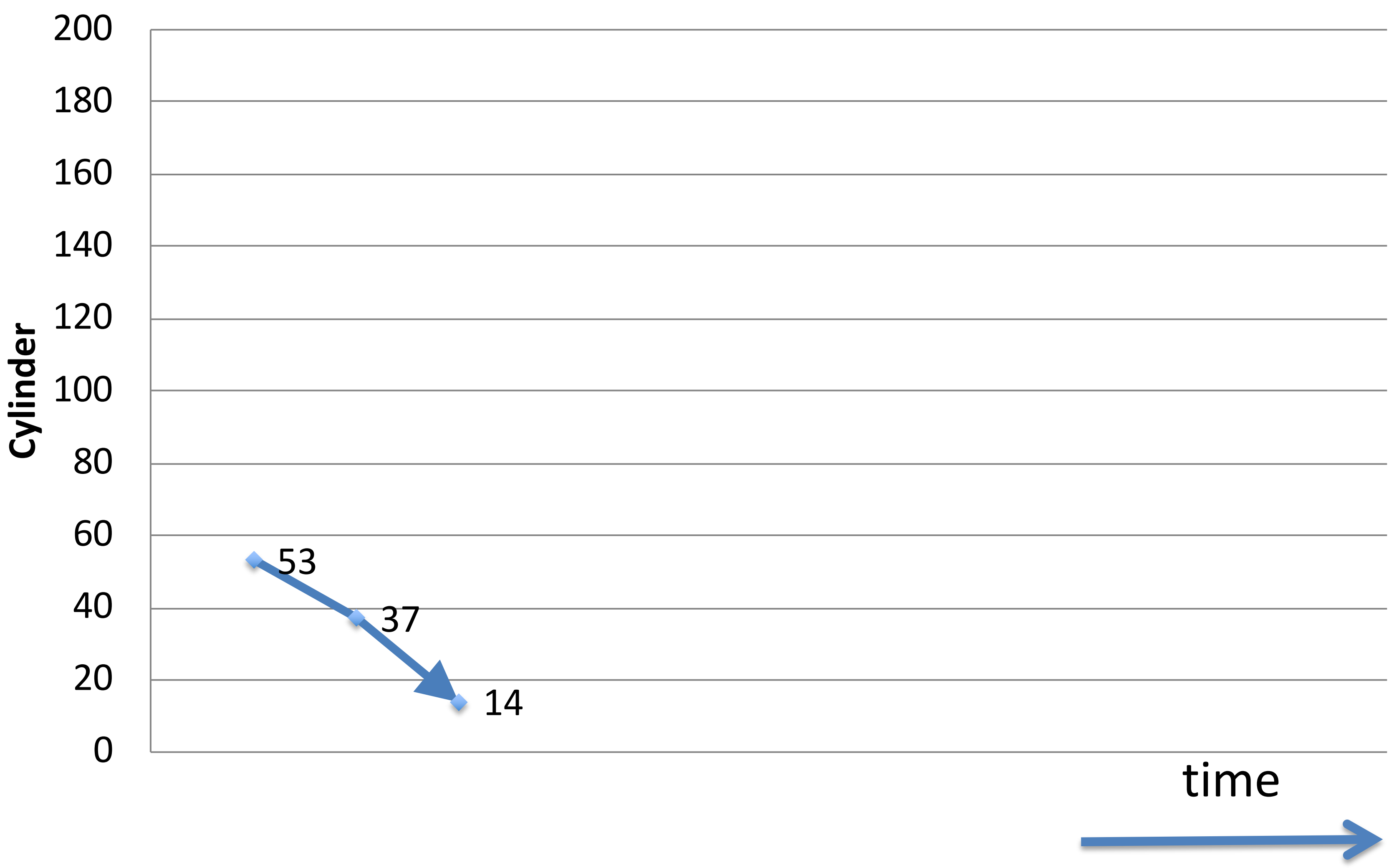
Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

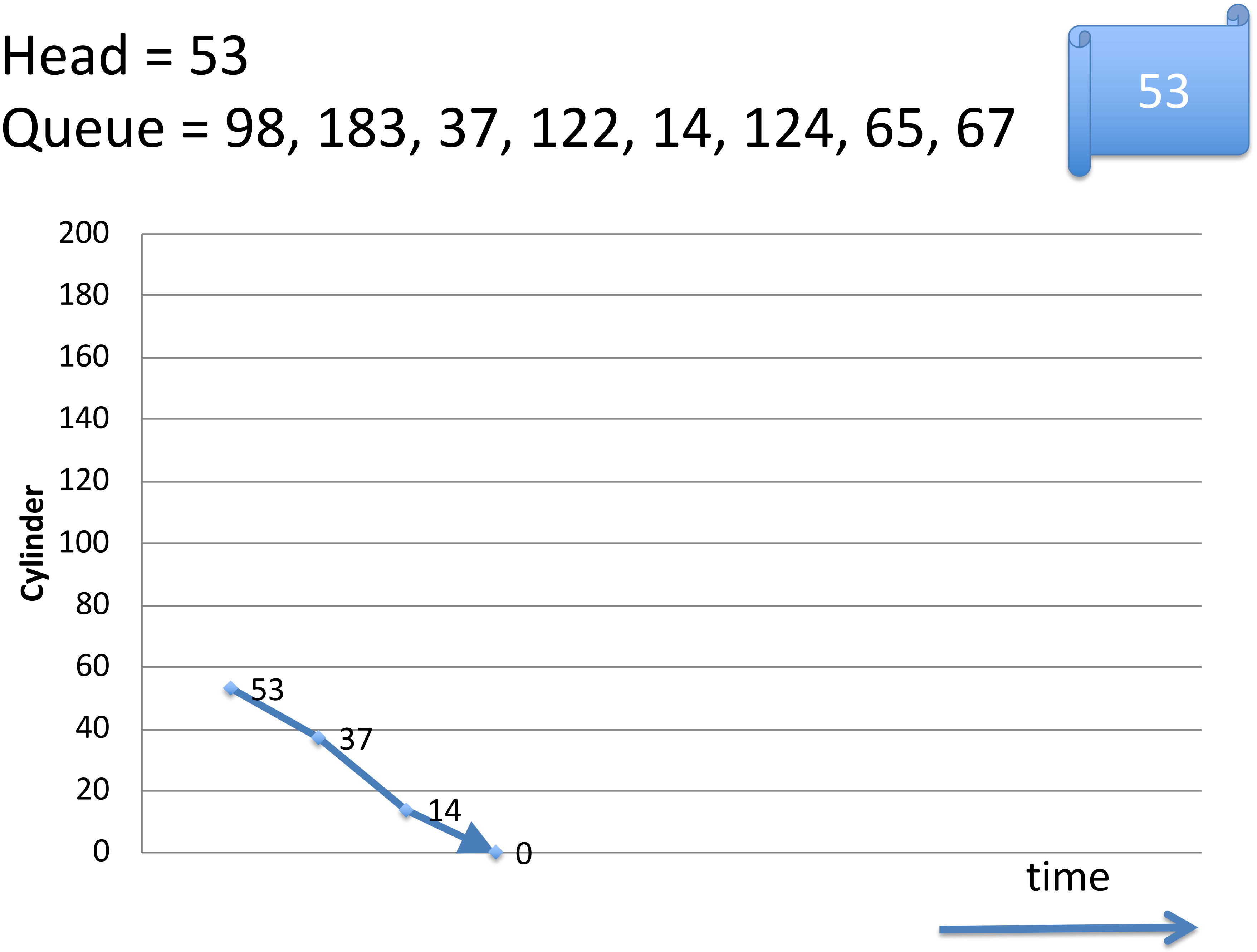
Queue = 98, 183, 37, 122, 14, 124, 65, 67

39



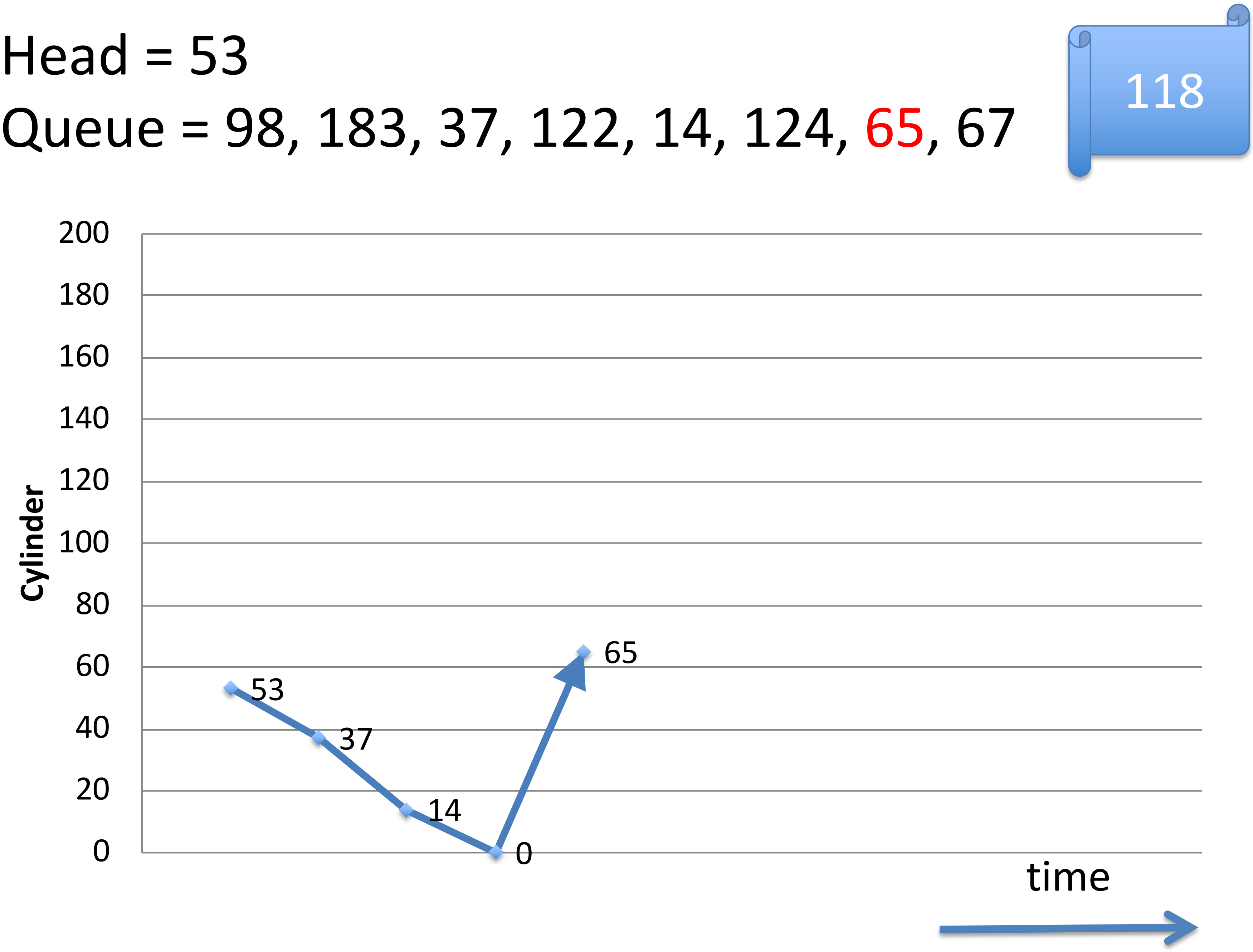
Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67



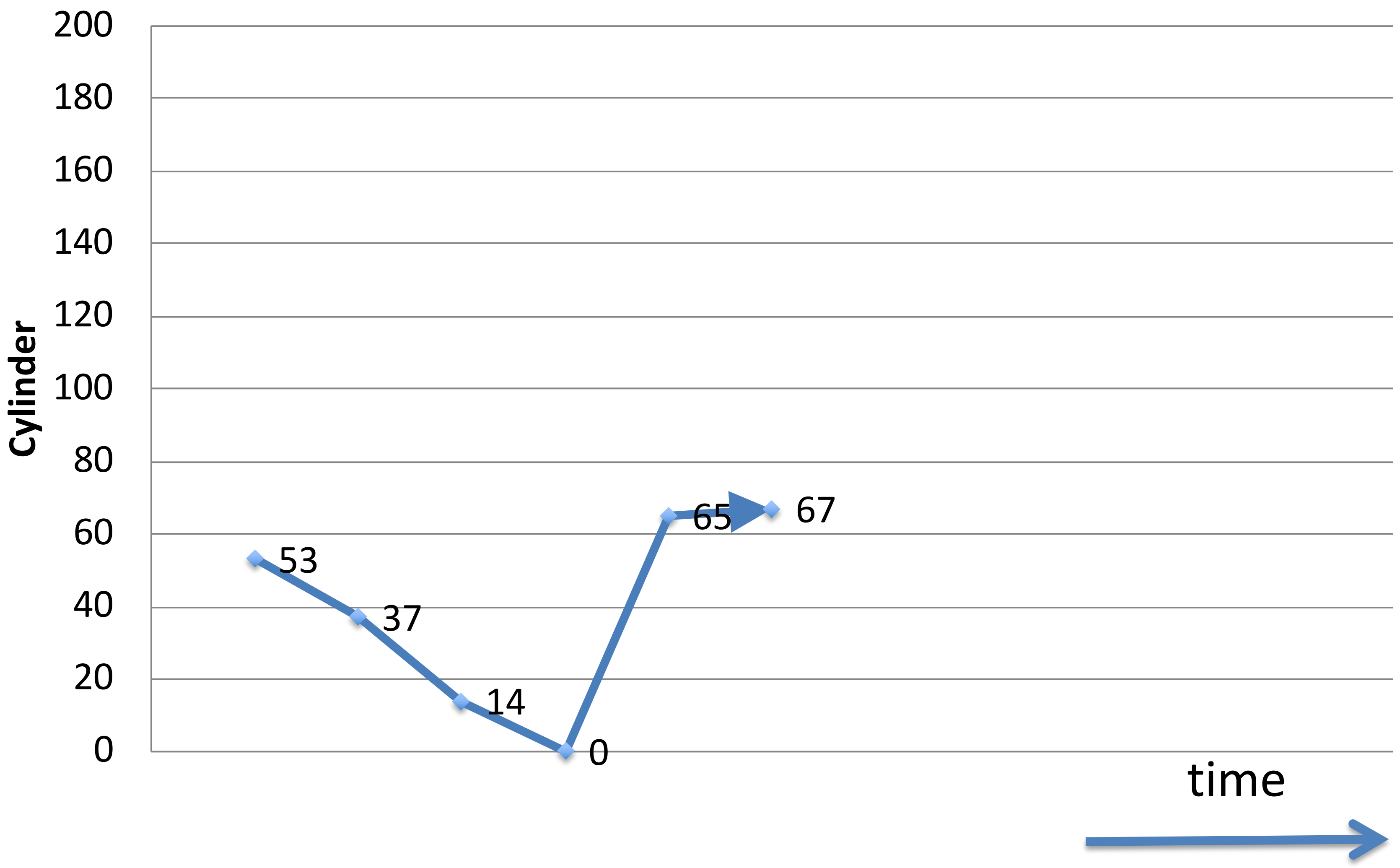
Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67



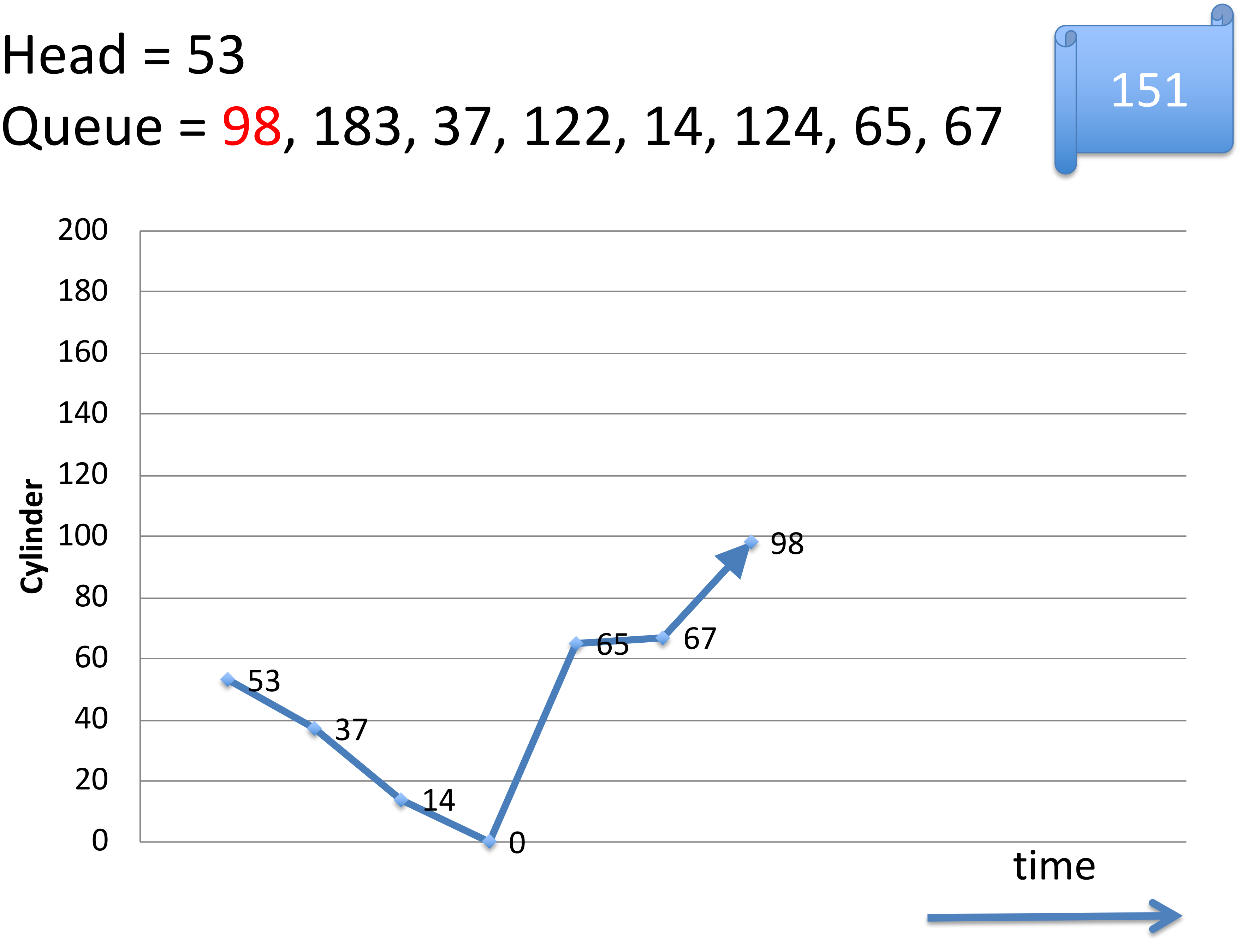
Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

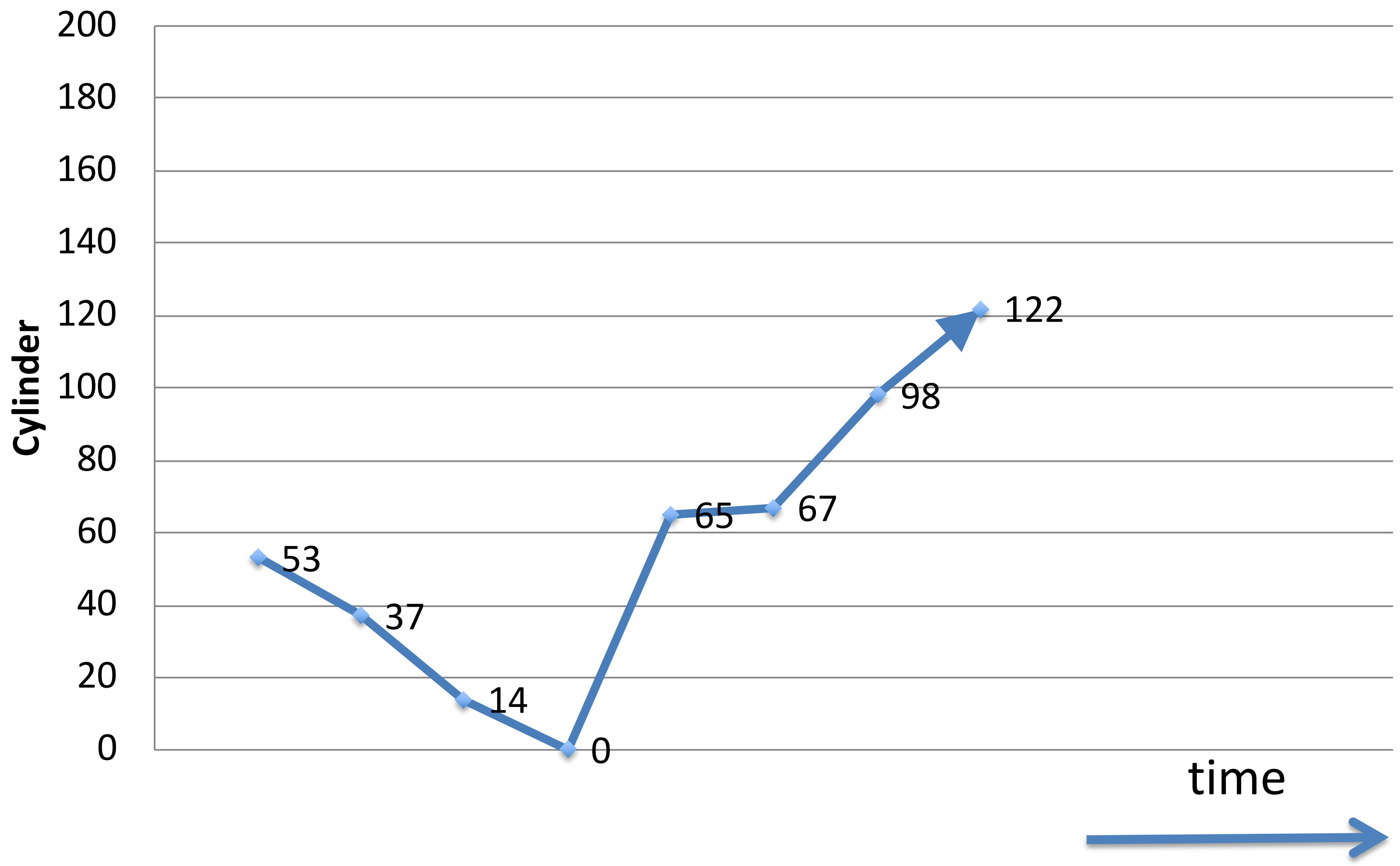
Queue = 98, 183, 37, 122, 14, 124, 65, 67



Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

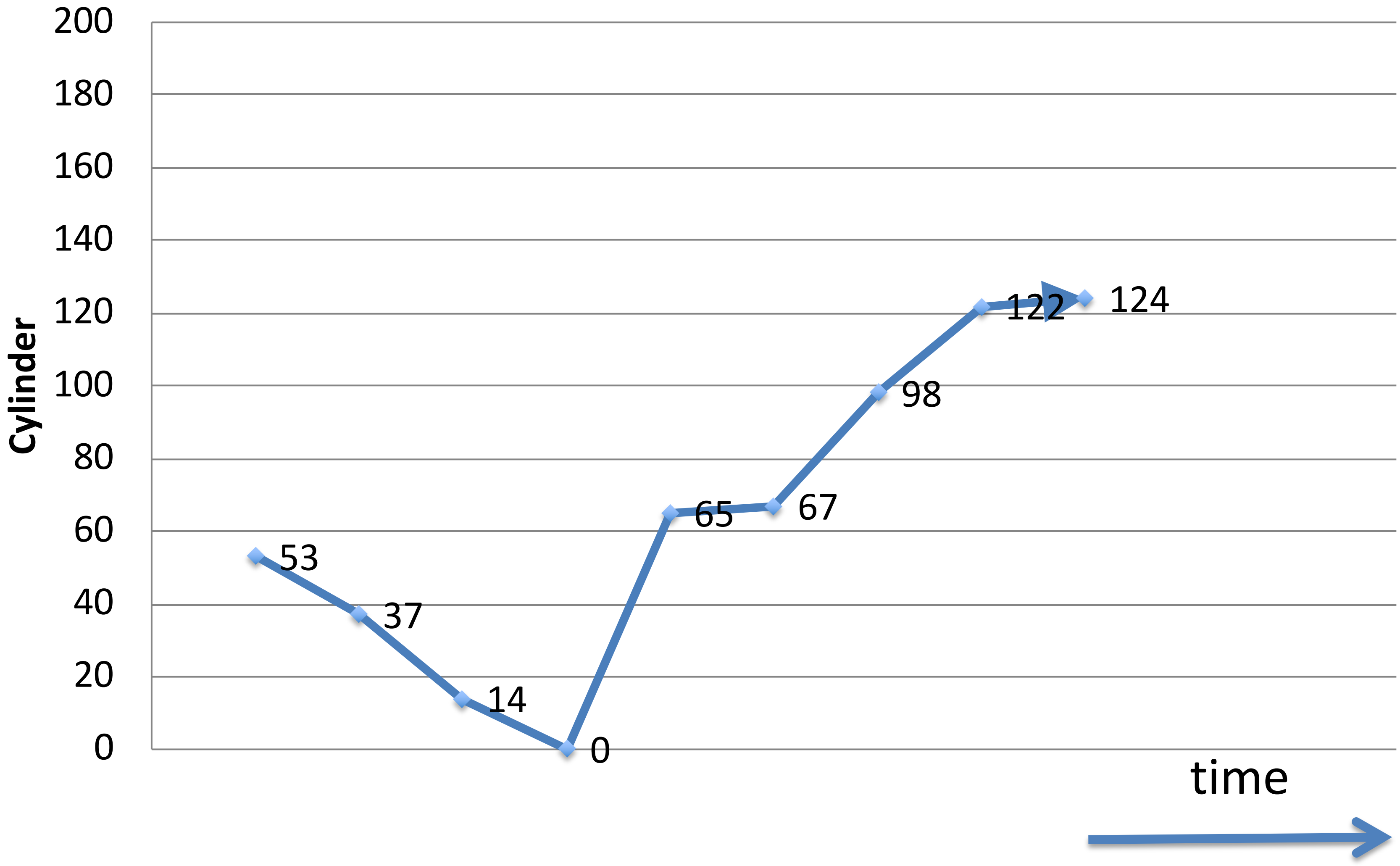
175



Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

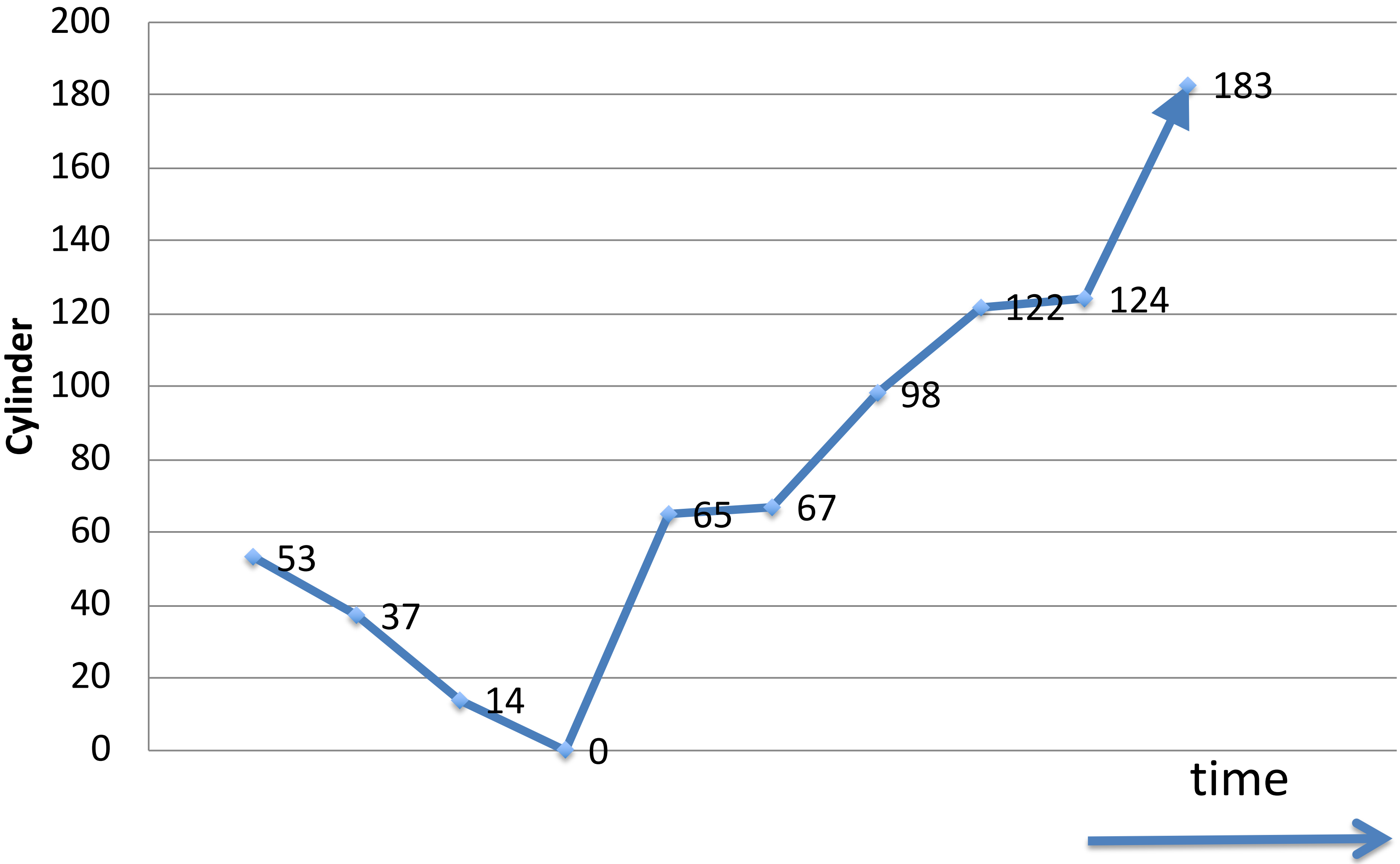
177



Head = 53

Queue = 98, 183, 37, 122, 14, 124, 65, 67

236



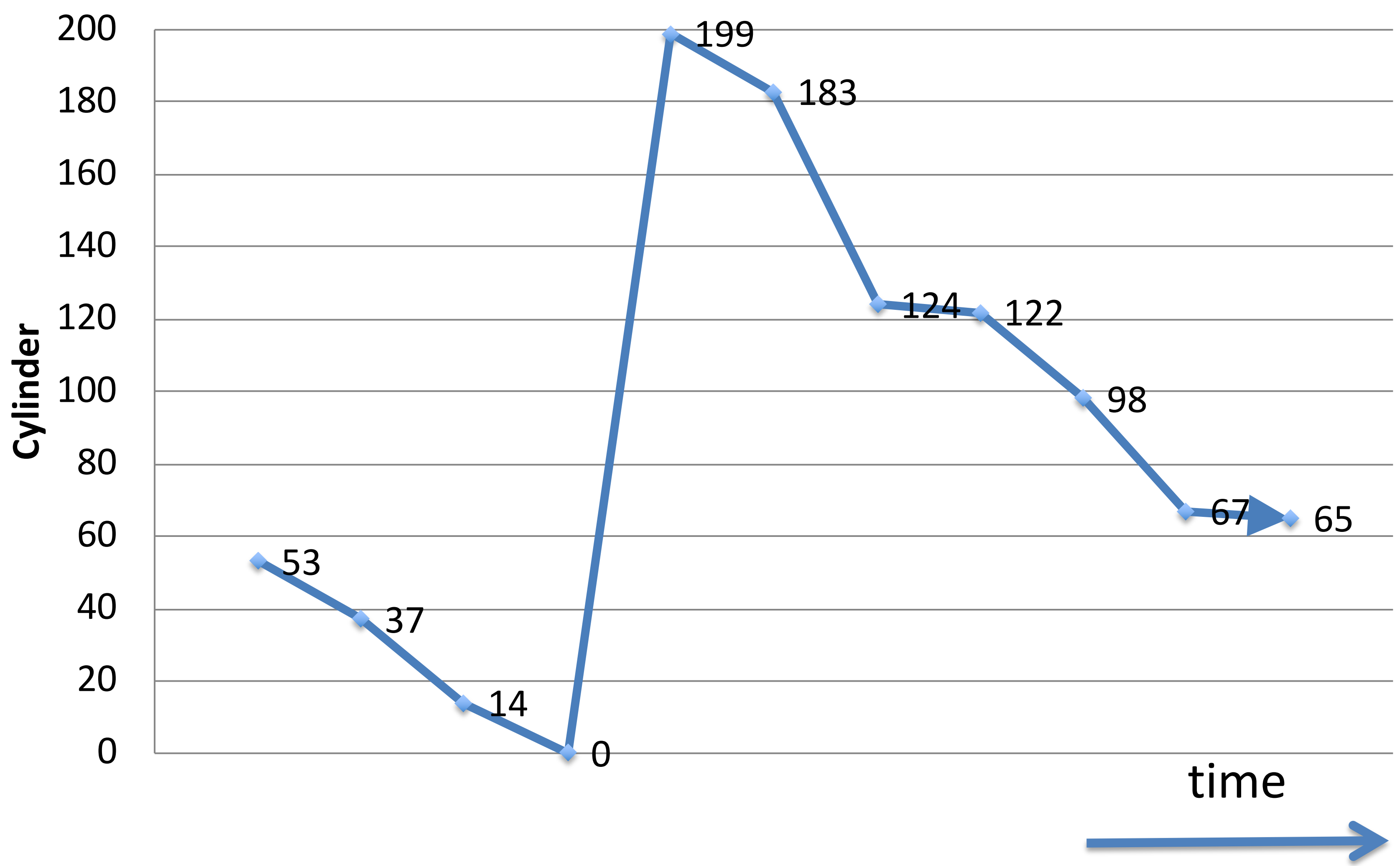
C-SCAN

- Similar to SCAN
- Move head in a circular way
  - From 0 to MAX_CYL; pick up requests as head moves
  - From MAX_CYL to 0; no requests served
- More uniform wait time

Head = 53, moving down

Queue = 98, 183, 37, 122, 14, 124, 65, 67

187



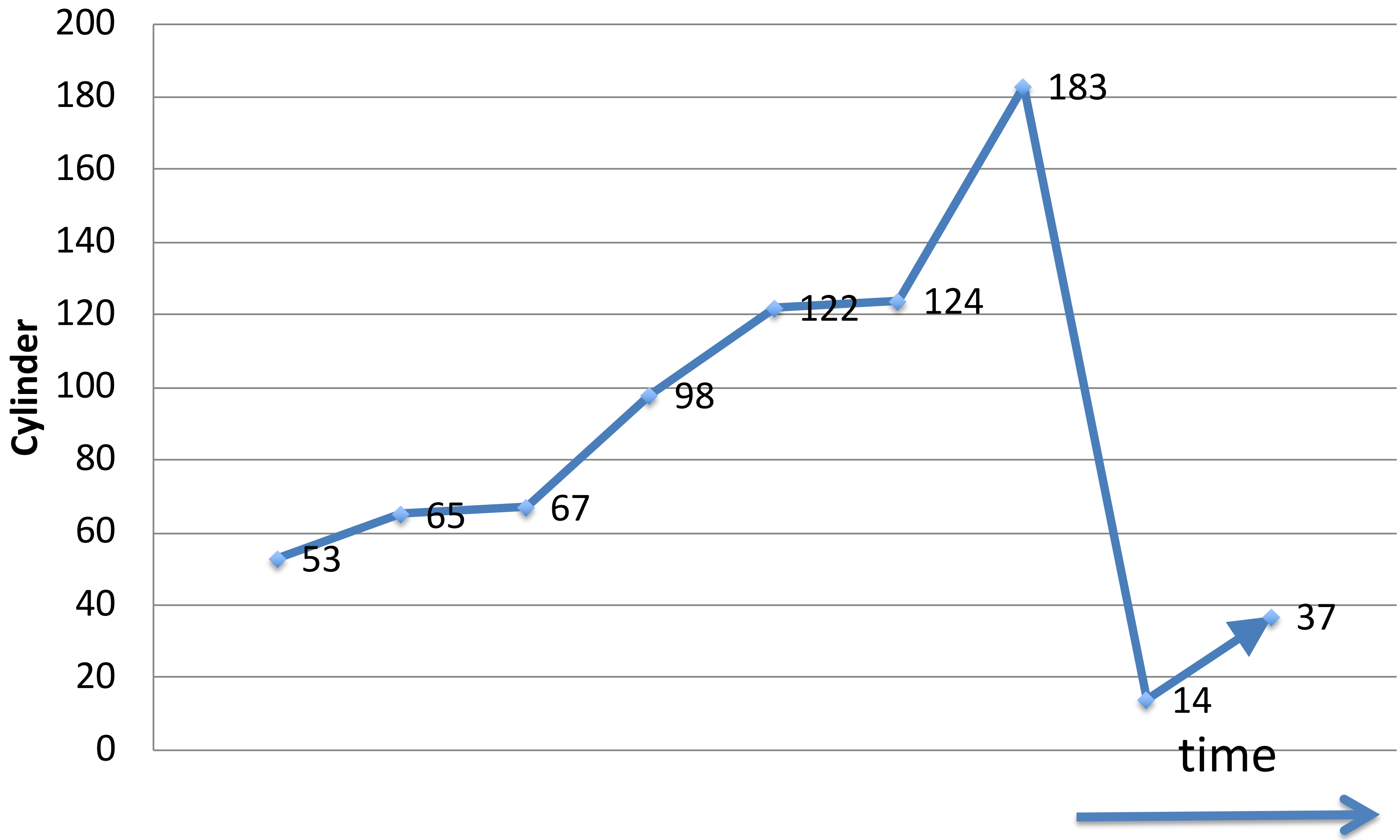
C-LOOK

- Similar to C-SCAN
- Always move head
  - From min_cyl to max_cyl; serve requests as head moves
  - From max_cyl to min_cyl; no requests served

Head = 53, moving down

Queue = 98, 183, 37, 122, 14, 124, 65, 67

153



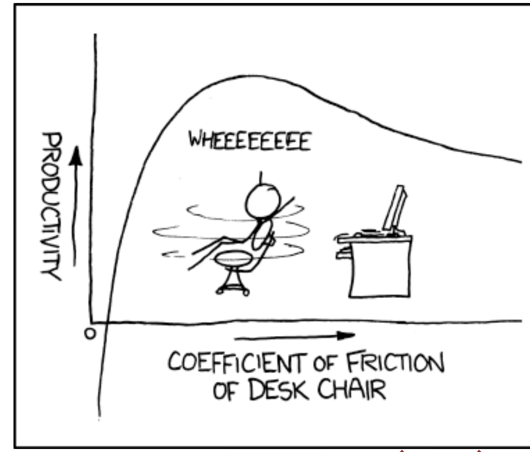
In Practice

- Some variation of C-LOOK (circular look)

Optimize Disk Access Time - 4

- Rule 4: Avoid rotational latency

- Clever disk allocation
- Locate consecutive blocks of file on consecutive sectors in a cylinder



When does what work well?

- Low load: clever allocation
- High load: disk scheduling

Why? – Under High Load

- Many scheduling opportunities
  - Many requests in the queue
- Allocation gets defeated
  - By interleaved requests for different files

Why? – Under Low Load

- Not much scheduling opportunity
  - Not many requests in the queue
- Sequential user access -> sequential disk access

- Cache tends to reduce load

Summary

- Disk characteristics
  - Access disk >> access memory
  - Seek > Rotational Latency > Transfer
- Optimizations
  - Cache
  - Read-ahead
  - Disk allocation
  - Disk scheduling


Happy Easter