Übungen Betriebssysteme (BS)

U5 – Dateioperationen

Alwin Berger

TU Dortmund - AG Systemsoftware

Agenda

Hauptthemen

- Dateien und Dateisysteme
- Schnittstellen für Dateioperationen in C

Zusatzinhalte

- Festplatten
- RAID

Dateien und Dateisysteme

- Daten auf einer Festplatte werden meistens zu abgeschlossenen Datenmengen, zu sogenannten **Dateien** abgefasst
- Dateien können je nach Anwendung in unterschiedlichsten Dateiformaten vorliegen und besitzen neben einen **Dateinamen** weitere **Meta-Informationen** wie **Dateigröße**, **Besitzer**, etc…
- Das **Dateisystem** legt fest, wie und wo Dateien auf der Festplatte gehalten, geschrieben und gelöscht werden

Ordner

- Viele Dateisysteme erlauben das organisieren von Dateien in **Verzeichnisstrukturen**
  - Rekursive, baumartige Struktur (**Verzeichnisbaum**)
- Jede Datei besitzt einen eindeutigen **Dateipfad**

Dateiartige Objekte

- UNIX-Systeme bilden unterschiedliche Betriebsmittel auf das Dateisystem (als ‘**nichtreguläre** Dateien’) ab:
  - Symbolische Links
  - E/A-Geräte und Datenträger
  - Feststehende Programmverbindungen (Pipes)
  - Netzwerkkommunikationsendpunkte (Sockets)
- Eine wie zuvor definierte Datei wird als **regulär** bezeichnet

Dateioperationen in C

- Exemplarischer Ablauf
  1. Datei öffnen
  2. Lesen
  3. Schreiben
  4. ...
  5. Datei schließen
- Öffnen einer Datei liefert Handle
  - Wird bei jeder Operation gebraucht - inkl. Schließen
  - Vermerkt den Modus (Lesen, Schreiben, beides) der geöffneten Datei
  - Beinhaltet Schreib-Lese-Zeiger
- Eine Datei kann mehrfach geöffnet werden
- Dateien müssen **nach** ihrer Verwendung geschlossen werden
  → Vermeidung von Ressourcenlecks

Schnittstellen für Dateioperationen in C

- Unter Linux gibt es mehrere Möglichkeiten
  - Syscall-Wrapper der C-Bibliothek (“low-level”), z.B. `open(2)`
  - Arbeitet mit Dateideskriptoren.
  - Nur eingeschränkt auf andere Betriebssysteme anwendbar

```
int open(const char *path, int flags)
  syscall(__NR_open, path, flags)
```

  - Abstrakte Stream-Schnittstelle in C (“high-level”), z.B. `fopen(3)` aus **stdio.h**
  - Abstraktion von Dateideskriptoren zu (gepuffertem) Strom

```
FILE* fopen(const char *path, const char *mode)
```

“low-level”-Dateioperationen

- open(2): Datei öffnen
  - `int open(const char *path, int flags)`
  - Gibt Dateideskriptor zurück - wird später benötigt
- read(2): Aus Datei lesen
  - `size_t read(int fd, void *buf, size_t count)`
- lseek(2): Schreib-/Leseposition verändern
  - `off_t lseek(int fd, off_t offset, int whence)`
- close(2): Offene Datei schließen
  - `int close(int fd)`
- Standardisierte Schnittstelle, u.a. in POSIX.1-2001

C-Streams (“high-level”)

- Abstrakter, plattformübergreifender Kommunikationskanal (z.B. unter Windows)
- Teil des C-Standards!
- Interne Pufferung: Blockweises Lesen → höhere Verarbeitungsgeschwindigkeit
- Unabhängiges Lesen und Schreiben an verschiedenen Positionen
- Stream-Repräsentation durch `struct FILE`
- Definierte Standard-Streams: `stdin, stdout, stderr`

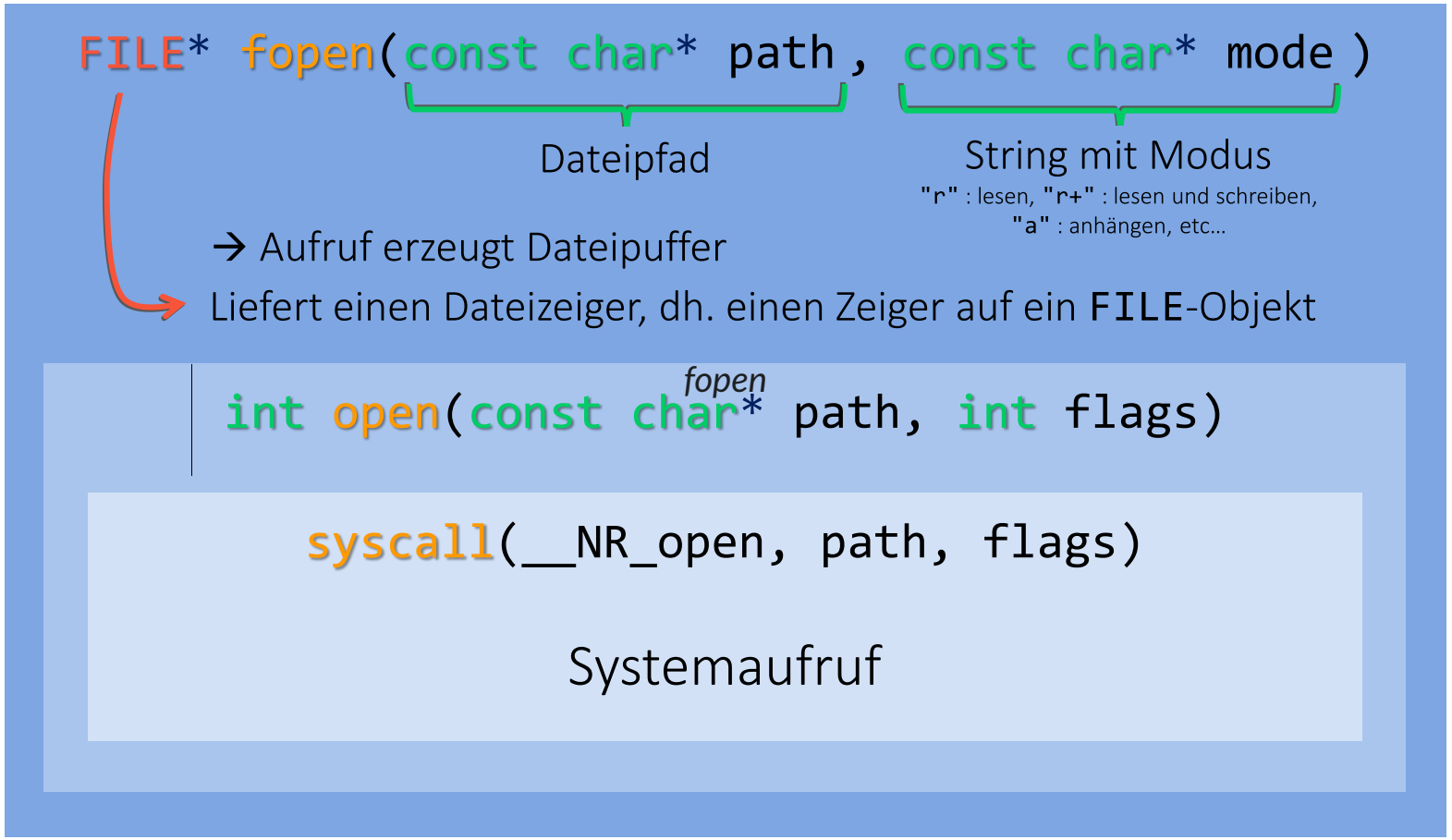
“high-level”-Dateioperationen (1)

- fopen(3): Datei öffnen
  - `FILE* fopen(const char *path, const char *mode)`
  - `mode`: z.B. “r” (Lesen), “r+” (Schreiben & Lesen), “a” (Anhängen)
    - Gibt Zeiger auf `FILE`-Datenstruktur zurück - wird später benötigt
- fread(3): Aus Datei lesen
  - `size_t fread(void *buf, size_t itemsize, size_t count, FILE *stream)`
- fseek(3): Schreib-/Leseposition verändern
  - `off_t fseek(FILE *stream, off_t offset, int whence)`
- fclose(3): Offene Datei schließen
  - `int fclose(FILE *stream)`

“high-level”-Dateioperationen (2)

- ftell(3): Gibt den Datei-Positionszeiger für den Stream `stream` aus
  - `long ftell(FILE *stream)`
- fscanf(3): Liest den Inhalt einer Datei gemäß Formatstring aus
  - `int fscanf(FILE *stream, const char *format, ...)`
  - Funktioniert wie `scanf(3)`
- fprintf(3): Schreibt einen formatierten String in eine Datei
  - `int fprintf(FILE *stream, const char *format, ...)`
  - Analog zu `printf(3)`
- Ebenfalls standardisierte Schnittstelle, u.a. in POSIX.1-2001

fopen(3) im Detail



Das FILE-Objekt

- Das FILE-Objekt enthält neben dem Dateideskriptor zum identifizieren einer geöffneten Datei auch den Modus sowie Zeiger auf den Puffer inklusive den Positionszeiger
- Achtung: Auf die Felder des FILE-Objekts darf nicht direkt zugegriffen werden, da dies laut C-Standard zu undefinierten Verhalten führt!
- Je nach System unterscheiden sich die Implementierungen von FILE
- DIR-Objekt als Analogon für Verzeichnisse

Dateiinformationen abfragen und auswerten

- Dateien haben verschiedene Eigenschaften
  - Typ, Eigentümer:in (Nutzer:in und Gruppe), Größe, Berechtigungen
- Abfrage mit `int stat(const char* pathname, struct stat* statbuf);`
  - Untersucht die Datei bzw. das dateiartige Objekt auf welches der Pfad path verweist und liefert bei Erfolg 0
  - Die Resultate dieser Untersuchung werden in finfo gespeichert
  - `S_ISREG(statbuf.st_mode)` liefert für eine reguläre Datei true
  - `S_ISDIR(statbuf.st_mode)` liefert für ein Verzeichnis true

Verzeichnisoperationen

- opendir(3): Öffnet ein Verzeichnis zum Lesen
  - `DIR* opendir(const char *path)`
  - Liefert einen Zeiger auf `DIR`-Datenstruktur - wird später benötigt
- readdir(3): Liest den nächsten Eintrag aus dem Verzeichnis
  - `struct dirent* opendir(DIR *dir)`
  - Liefer einen Zeiger auf `dirent`-Datenstruktur
  - Gibt Auskunft über den aktuellen Eintrag im Verzeichnis
  - Siehe `man 3 readdir`
- closedir(3): Schließt das aktuelle Verzeichnis
  - `int closedir(DIR *dir)`

Dateipfade in C erstellen

- Speicherplatz für zu konstruierenden Pfad sicherstellen
- Benötigter Speicher hängt von der Länge ab
  - “foo/bar.txt” vs. “irgendein/furchtbar/langer/Pfad.txt”
- Zur Erinnerung:
  - Länge einer Zeichenkette mit `strlen(3)` bestimmen
  - Speicher mit `malloc(3)` allozieren un später freigeben (`free(3)`)

```c
int bufsize = strlen("foo") + strlen("bar") + 2;

char *buf = malloc(bufsize);
if (buf == NULL) { /* Fehler */ }

snprintf (buf, bufsize, "%s/%s", "foo", "bar");
// buf enthält jetzt "foo/bar" (inkl. Nullterminator)
```

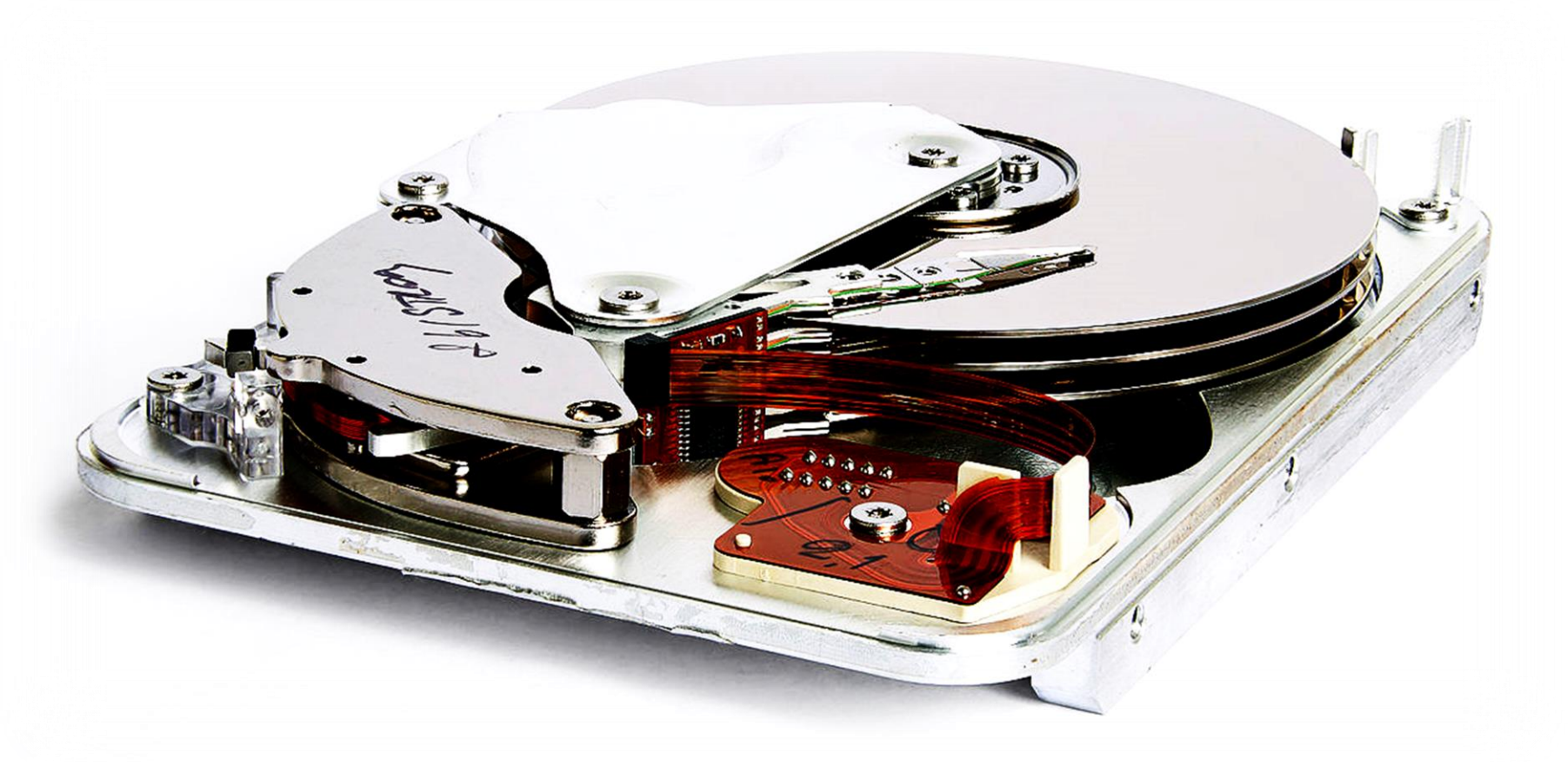
Zusatzinhalte

Festplatten und Bandlaufwerke

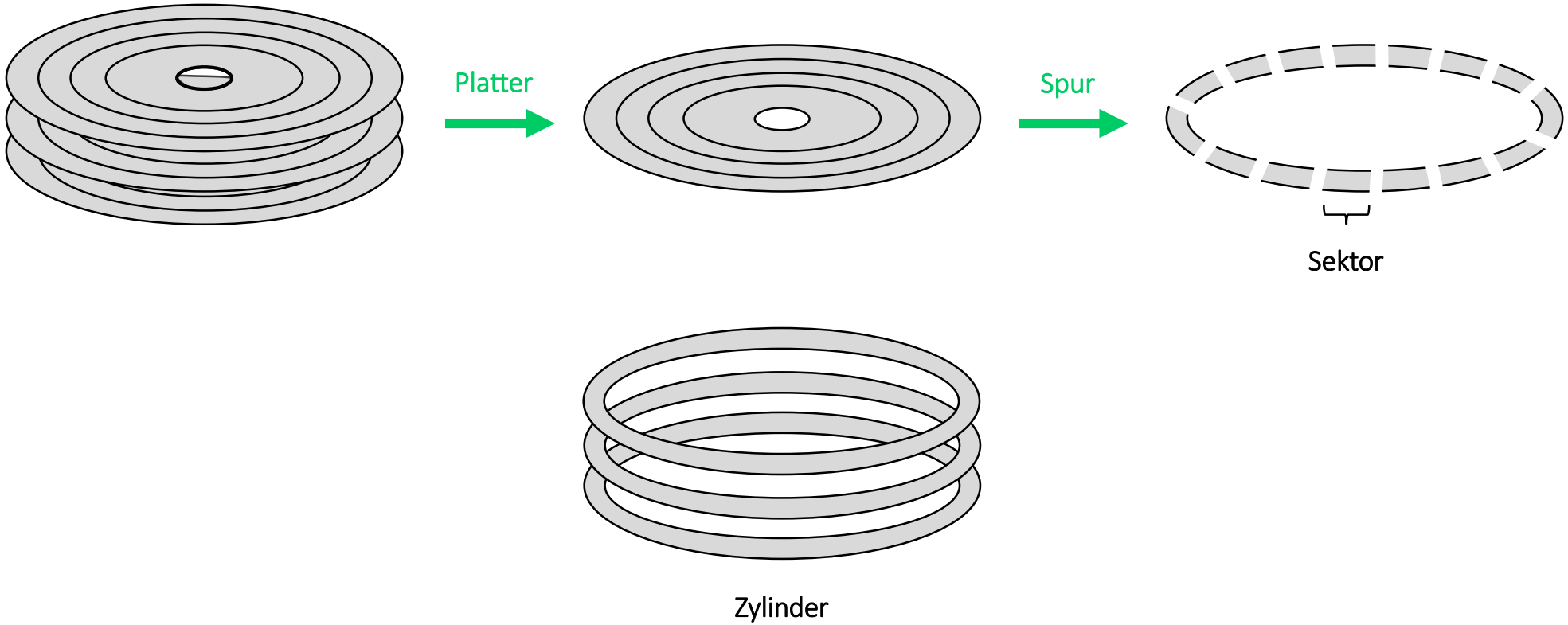
Festplatten (HDD)

- Bei **Festplatten** (engl. **Hard disk drive**, kurz **HDD**) handelt es sich um elektromechanische persistente Speichermedien
- Ein **Lesekopf** trägt auf sich schnell rotierende Scheiben (**Platter**) eine **Magnetisierung** auf: Speicherung mittels **Remanenz**

Aufbau von HDDs



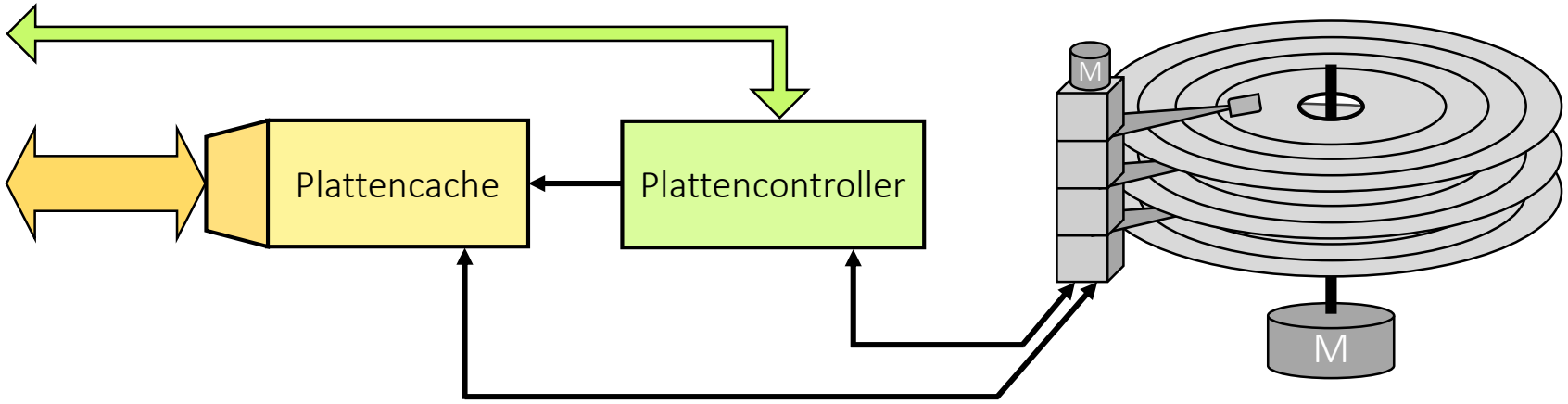
Aufbau von HDDs (Fortsetzung)



Lesen und Schreiben von HDDs

- Bei einem **Schreibvorgang** werden die zu schreibenden Blöcke in einen **Platten-Cache** geschrieben: **Plattencontroller** schreibt selbständig den Block in den entsprechend adressierten Sektor
- Bei einem **Lesevorgang** werden die gewünschten Sektoren beim Plattencontroller angefragt, dieser schreibt diesen dann in Schritt für Schritt den Platten-Cache

Festplatte als Blockschaltbild



E/A-Geräteklassen

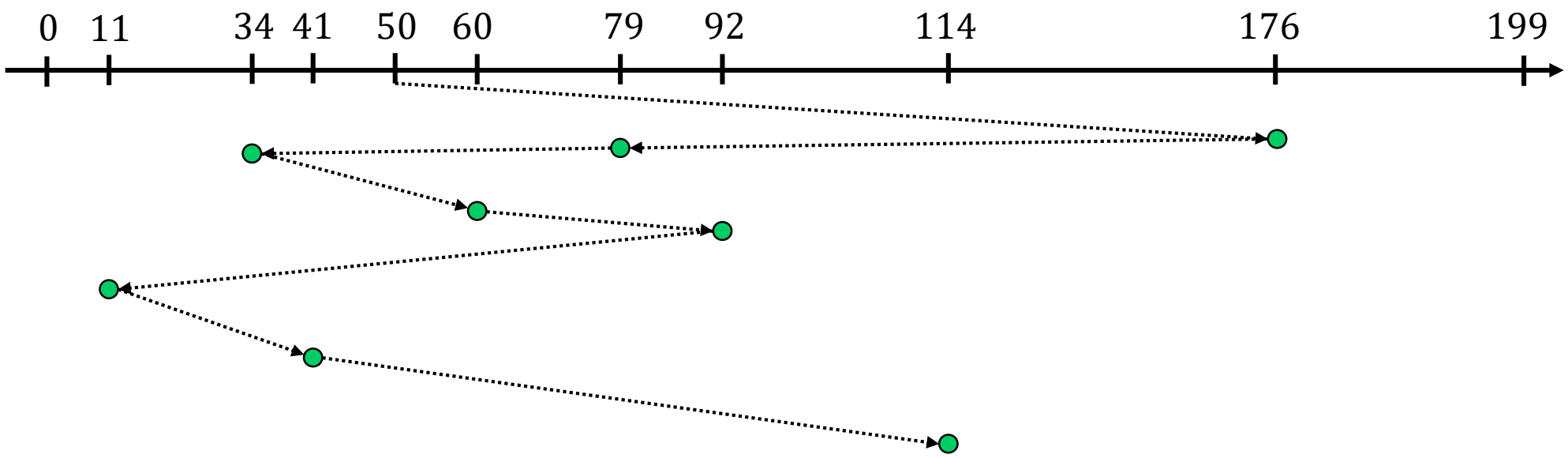
- **Zeichenorientierte Geräte**
  - Sequentieller dh. zeichenweise Zugriff auf Daten
  - Maus und Tastatur sind zeichenorientiert
- **Blockorientierte Geräte**
  - Wahlfreier Zugriff auf die Daten
  - Festplatten sind blockorientiert

Lesen und Schreiben von HDDs (Scheduling)

- Zum Bearbeiten solcher Anfragen ist wegen der deutlich langsameren Lese/Schreibgeschwindigkeit der HDDs gegenüber dem restlichen System eine **Scheduling-Strategie** nötig
- **Naiver Ansatz: FCFS** (Windhundprinzip)

HDD-Scheduling nach Windhundprinzip (FCFS)

$$L_0 = \{176, 79, 34, 60, 92, 11, 41, 114\}$$



Quelle: https://www.geeksforgeeks.org/scan-elevator-disk-scheduling-algorithms/, abgerufen am 22.06.2024

Lesen und Schreiben von HDDs (Fortsetzung)

- Lese- und Schreibvorgänge beanspruchen bei HDDs wesentlich mehr Zeit als bei Festkörperspeichern (SSD)
- Ständige **Spurwechsel** sollten daher unbedingt vermieden werden, um wertvolle Zeit zu sparen
- Analogie: Automatischer Aufzug über mehrere Etagen

SSTF

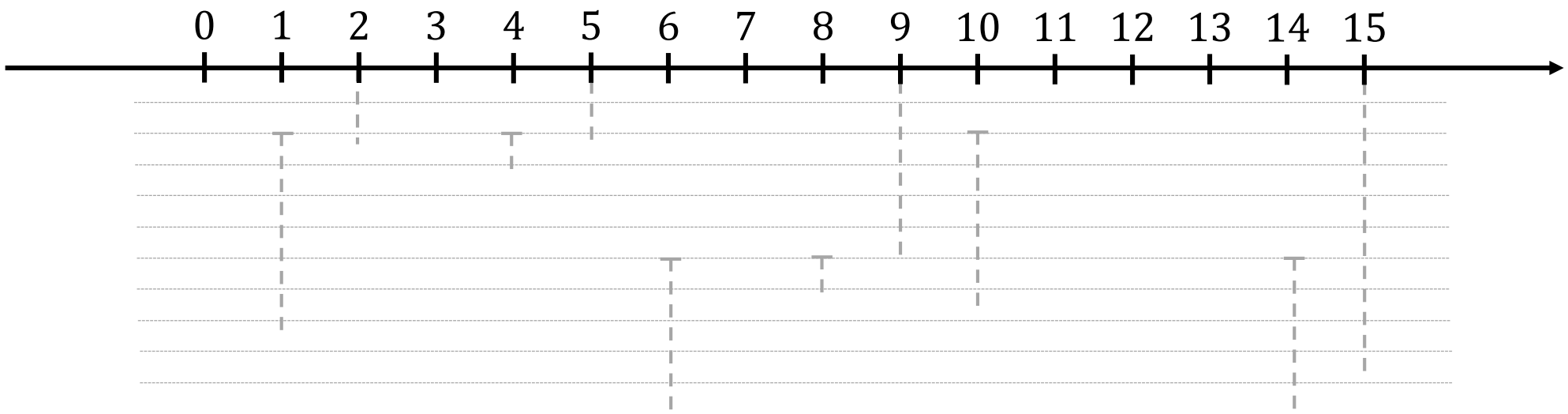
Bei dem Scheduling-Verfahren **SSTF** (engl. **Shortest seek time first**) wird die Anfrage mit der kürzesten Positionierungszeit vorgezogen

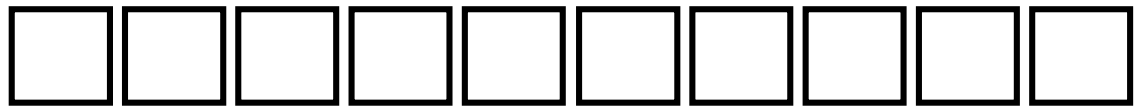
```
SSTF(pos ∈ [N], request):
    while request ≠ ∅:
      pos := find(next ∈ request, |next − pos| = min{|req − pos| | req ∈ request})
      request.Remove(pos)
      ⋮
```

SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

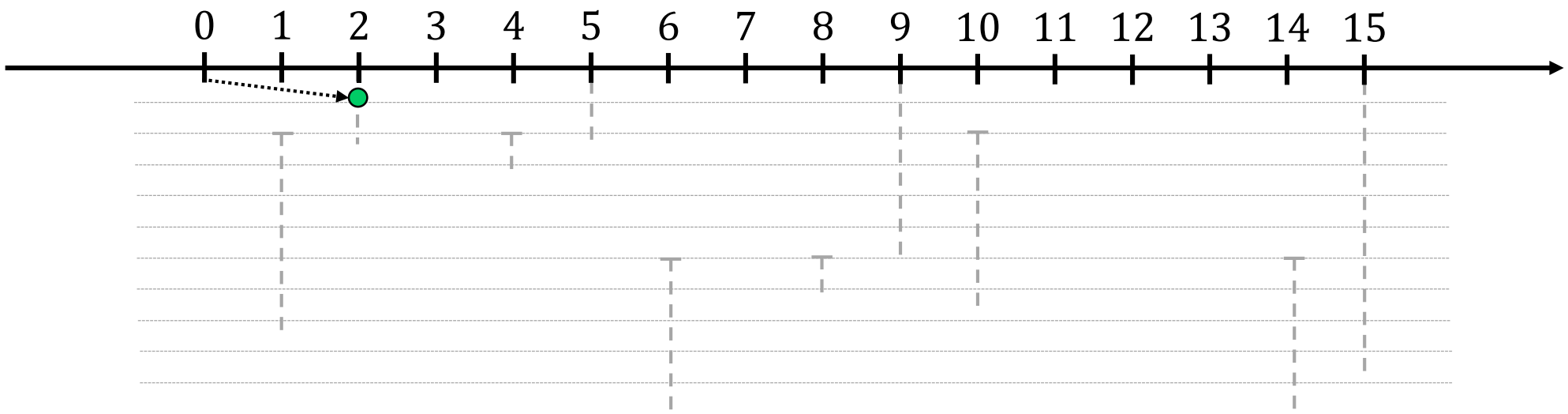


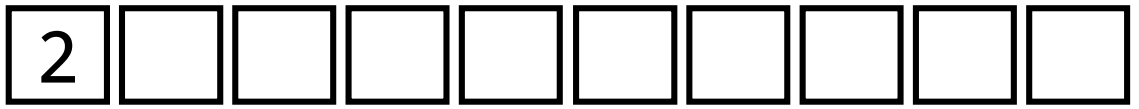


SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

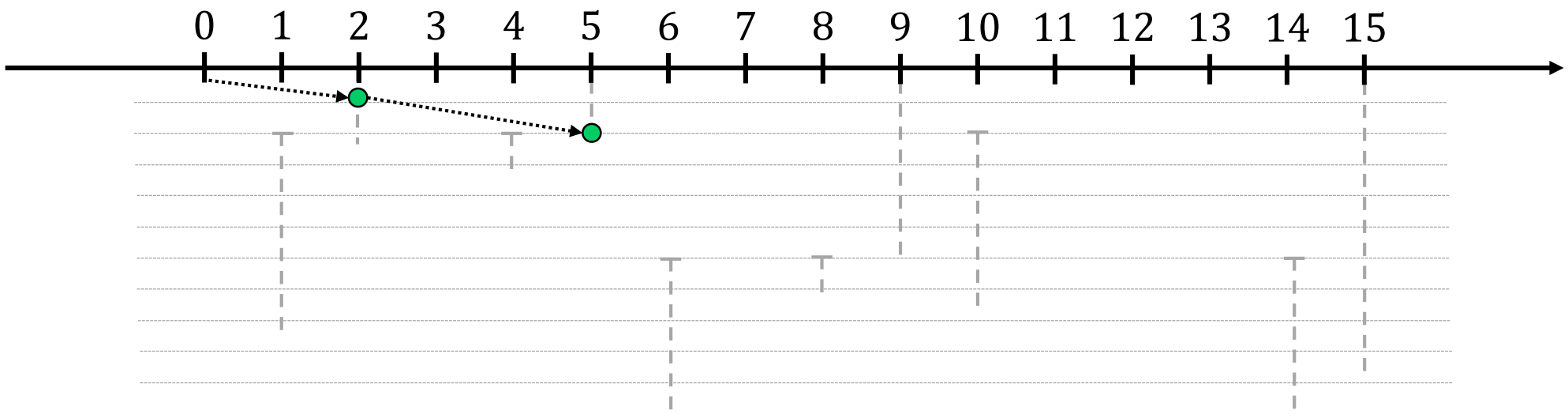


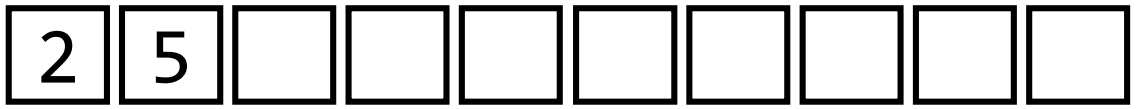


SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

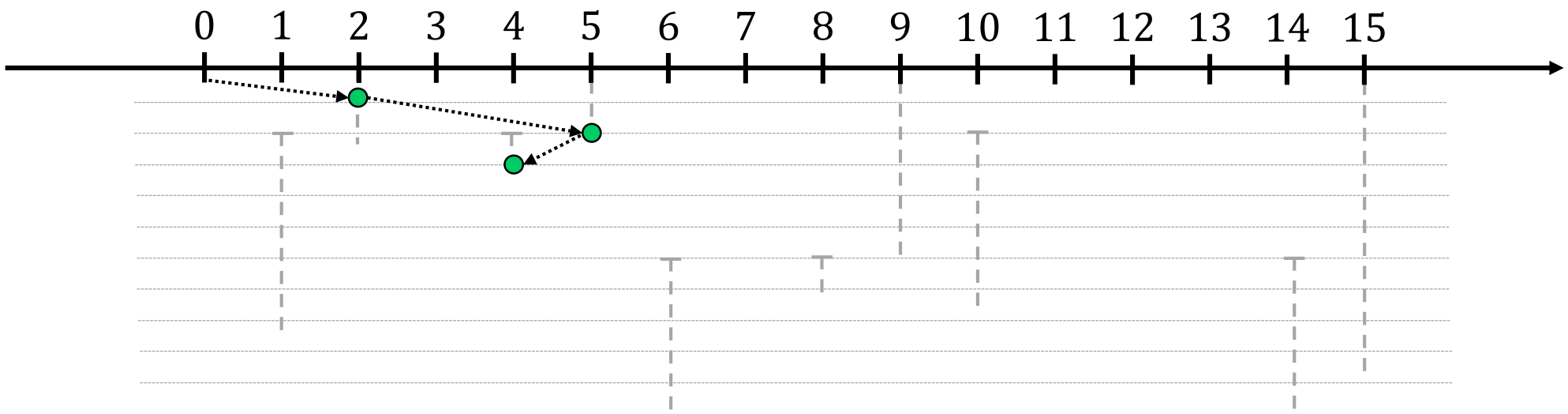




SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

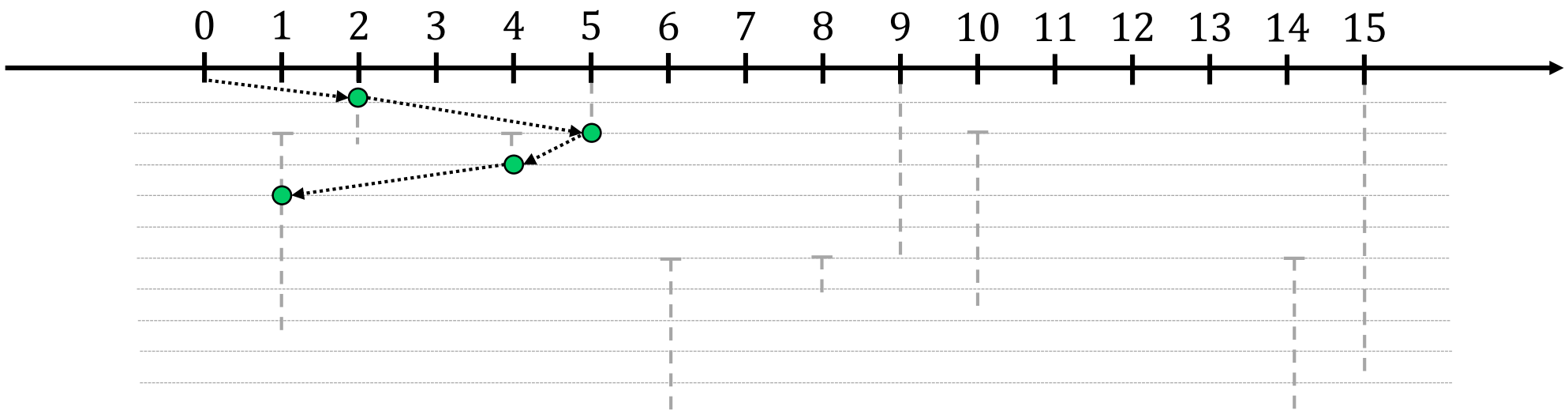


2	5	4							

SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

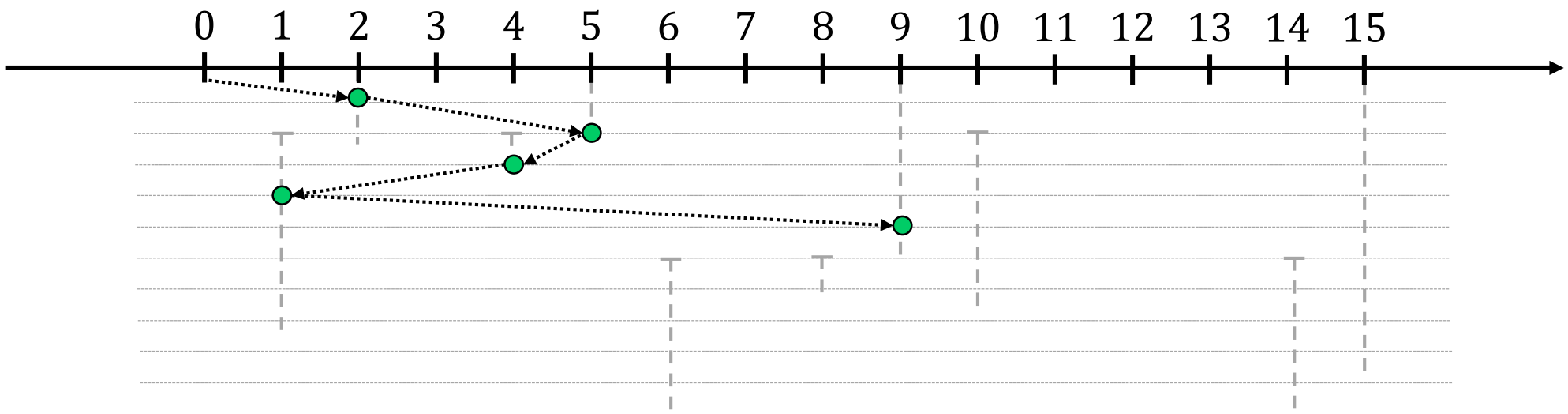


2	5	4	1						

SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

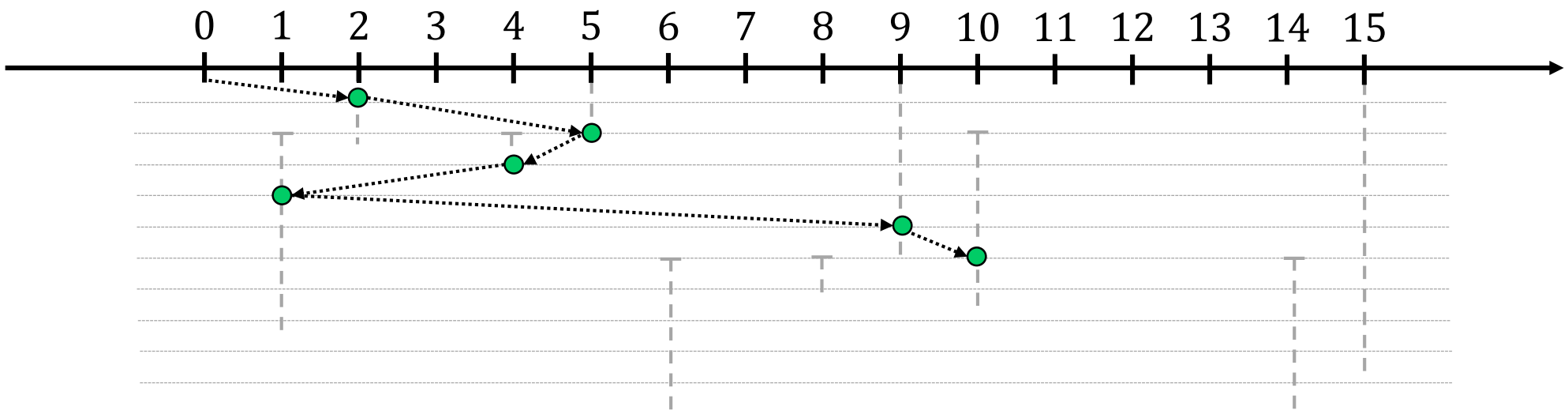


2	5	4	1	9					

SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

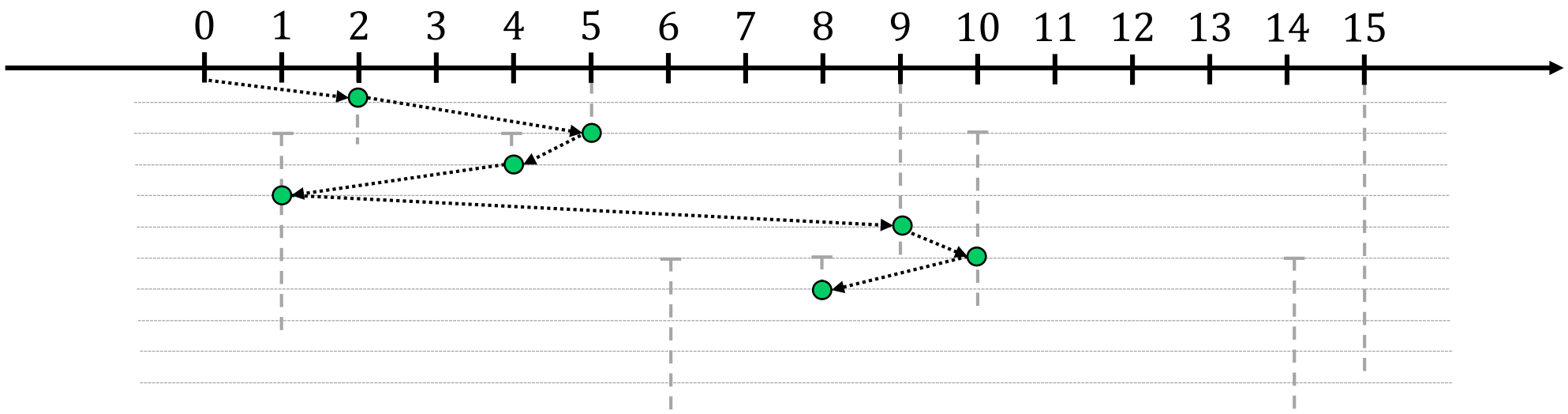


2	5	4	1	9	10				

SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

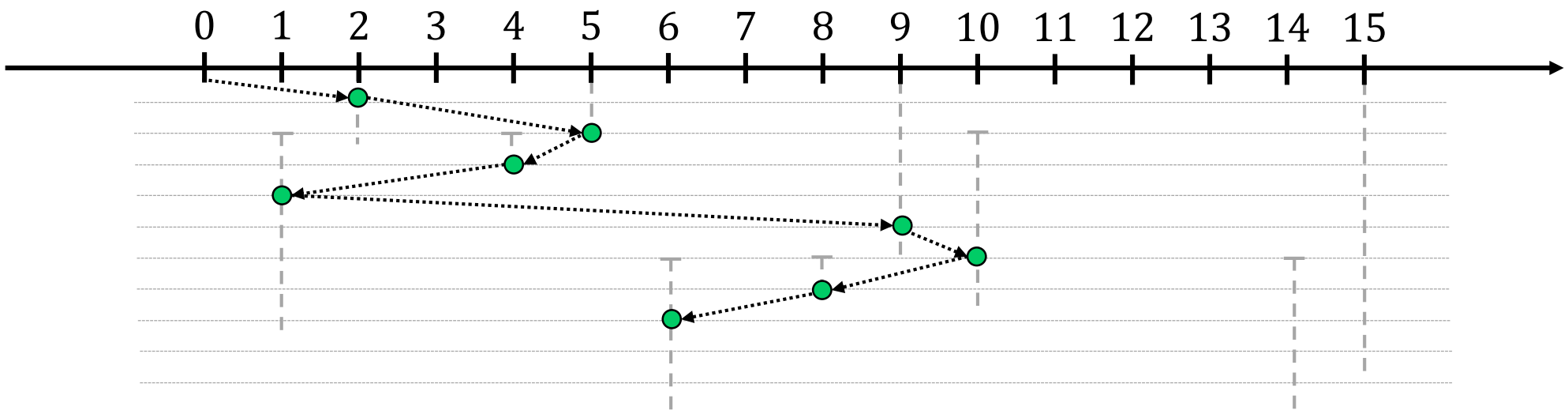


2	5	4	1	9	10	8			

SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

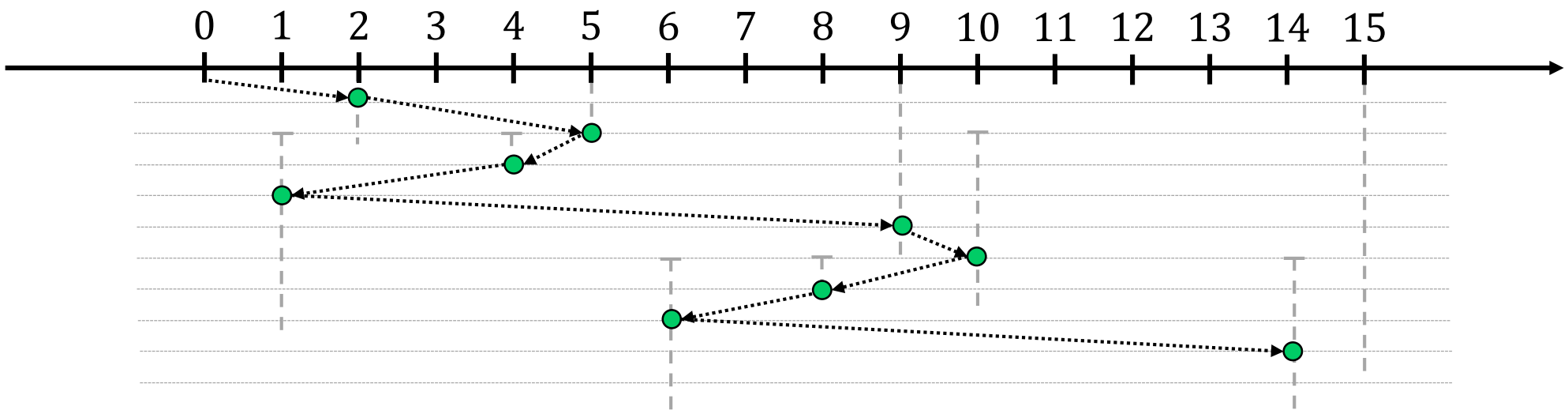


2	5	4	1	9	10	8	6		

SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

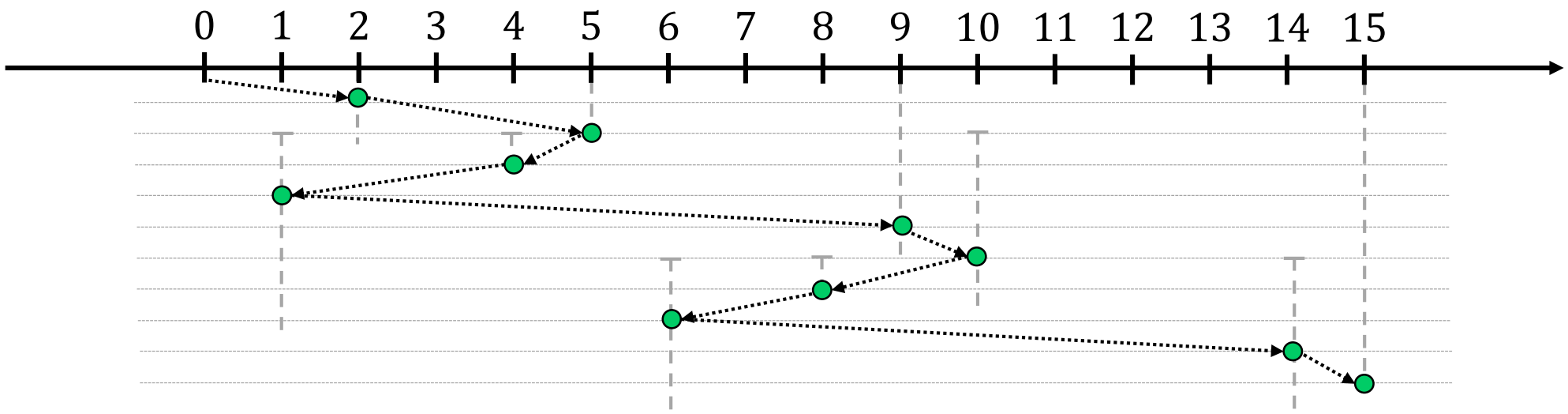


2	5	4	1	9	10	8	6	14	

SSTF in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$



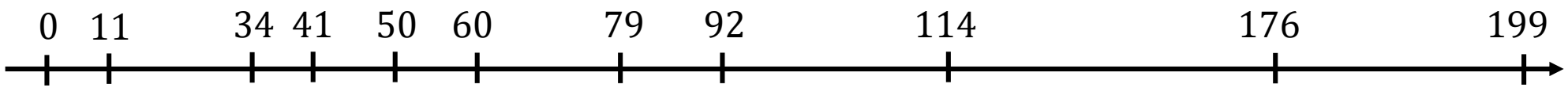
2	5	4	1	9	10	8	6	14	15

Aufzugalgorithmus für Anfragemengen

1) **Ausgangssituation**: Der Aufzug befindet sich in einer bestimmten Etage. Es gibt eine Menge L von Anfragen für verschiedene Etagen. Der Aufzug hat eine aktuelle Bewegungsrichtung (aufwärts $\uparrow = 1$ oder abwärts $\downarrow = -1$).
2) **Bewegungsrichtung festlegen**: Wenn der Aufzug still steht, wähle die Richtung zur nächsten Anfrage. Ansonsten behalte die aktuelle Richtung bei.
3) **Anfragen sortieren**: Sortiere die Anfragen in der aktuellen Richtung.
4) **Abarbeiten der Anfragen**: Bewege den Aufzug in der festgelegten Richtung. Halte an jeder Etage, für die eine Anfrage vorliegt. Nimm neue Anfragen in derselben Richtung auf.
5) **Richtungswechsel**: Wenn keine weiteren Anfragen in der aktuellen Richtung vorliegen, wechsle die Richtung. Beginne wieder mit Schritt 3.
6) **Leerlauf**: Wenn keine Anfragen mehr vorliegen, warte auf neue Anfragen. Bei einer neuen Anfrage, beginne wieder mit Schritt 2.

HDD-Scheduling nach Aufzugalgorithmus (SCAN)

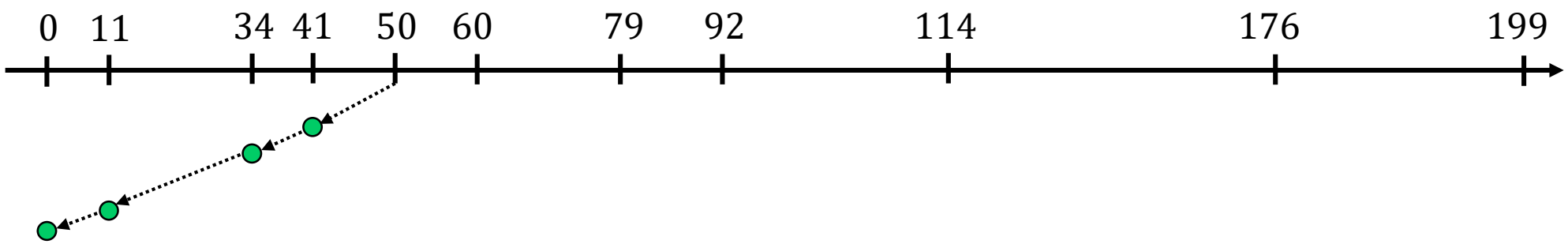
$$L_0 = \{176, 79, 34, 60, 92, 11, 41, 114, 0\}$$



Quelle: https://www.geeksforgeeks.org/scan-elevator-disk-scheduling-algorithms/, abgerufen am 22.06.2024

HDD-Scheduling nach Aufzugalgorithmus (SCAN)

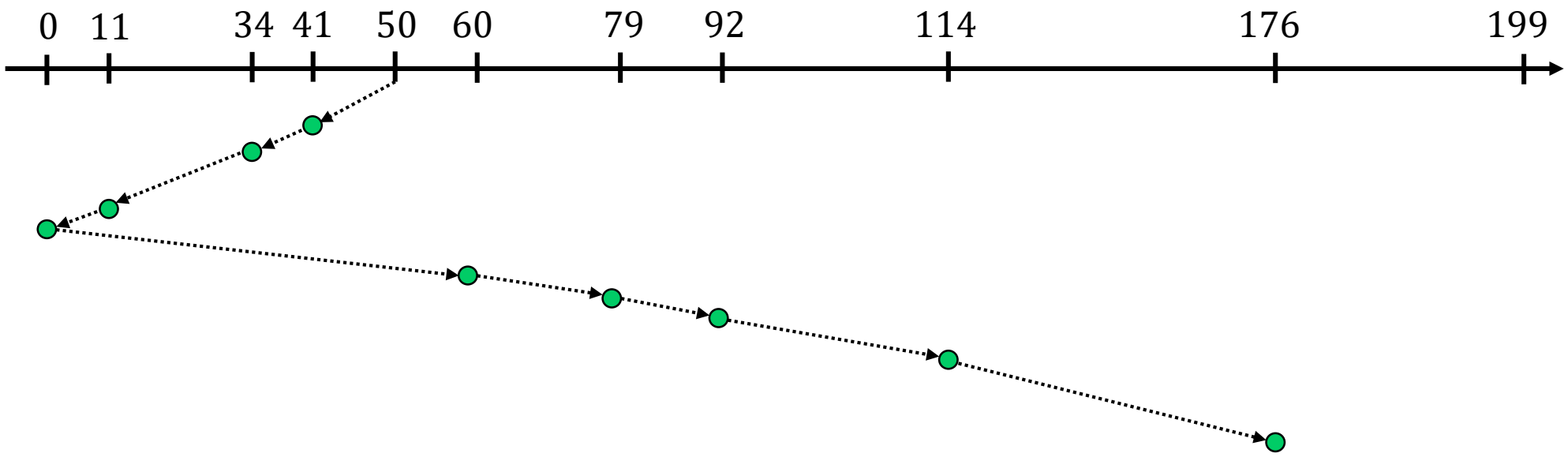
$$L_0 = \{176, 79, 34, 60, 92, 11, 41, 114, 0\}$$



Quelle: https://www.geeksforgeeks.org/scan-elevator-disk-scheduling-algorithms/, abgerufen am 22.06.2024

HDD-Scheduling nach Aufzugalgorithmus (SCAN)

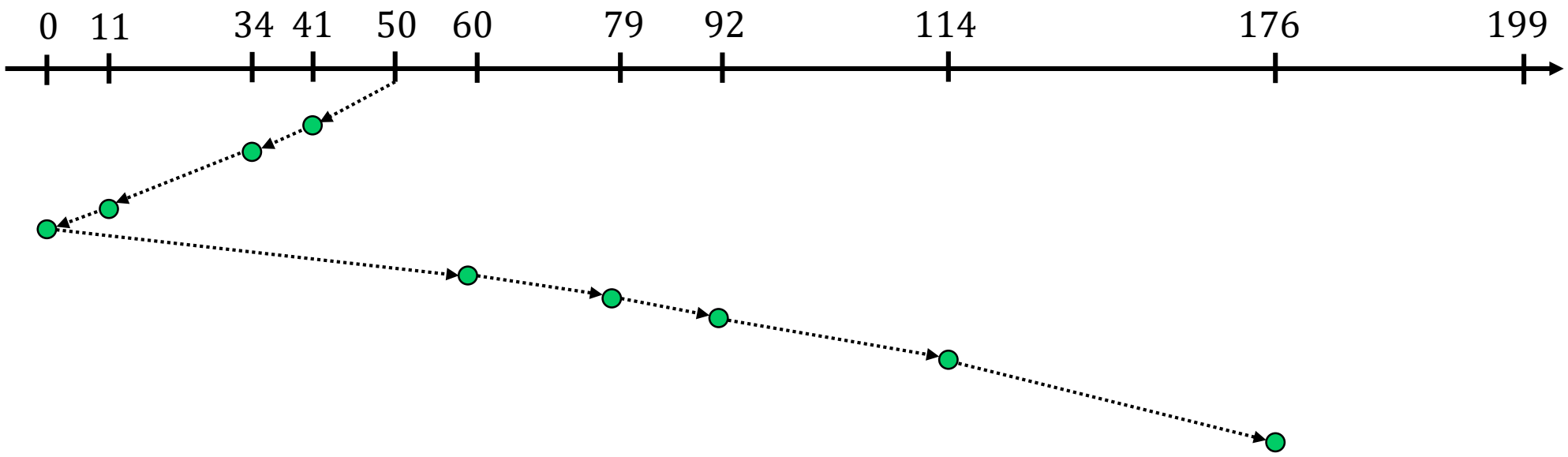
$$L_0 = \{176, 79, 34, 60, 92, 11, 41, 114, 0\}$$



Quelle: https://www.geeksforgeeks.org/scan-elevator-disk-scheduling-algorithms/, abgerufen am 22.06.2024

HDD-Scheduling nach Aufzugalgorithmus (SCAN)

$$L_0 = \{176, 79, 34, 60, 92, 11, 41, 114, 0\}$$



Quelle: https://www.geeksforgeeks.org/scan-elevator-disk-scheduling-algorithms/, abgerufen am 22.06.2024

Aufzugalgorithmus in Echtzeit

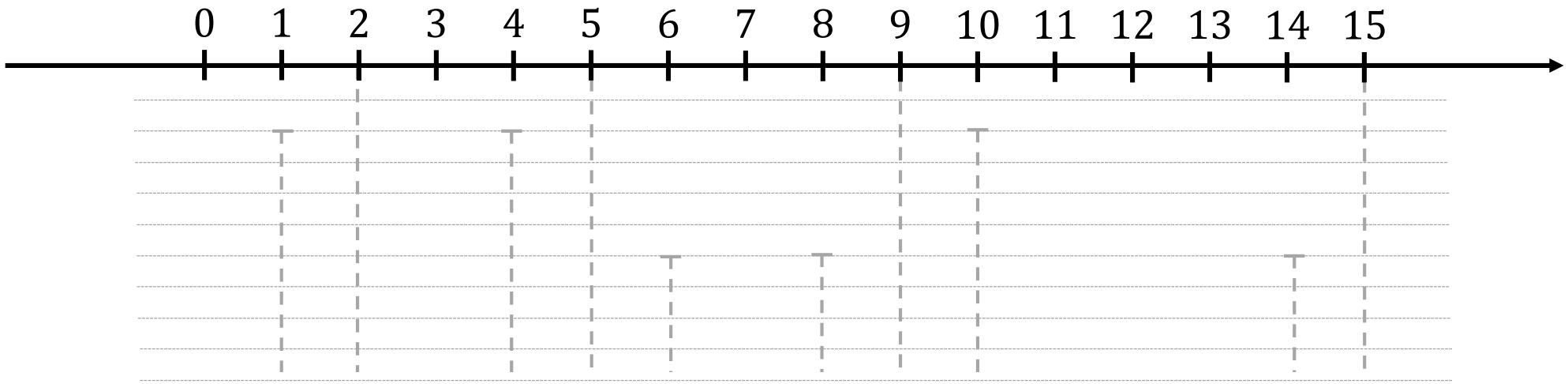
- Im realen System ist es möglich, dass im laufenden E/A-Vorgang die Warteschlange um weitere Anfragen ergänzt wird
- Mengen von Anfragen zum Zeitpunkt t werden mit L_t notiert

Aufzugalgorithmus in einem Gebäude (Beispiel)

- Gegeben sei ein Hochhaus, das über dem Erdgeschoss 15 weitere Etagen Besitzt: Alle 16 Etagen werden von einem Aufzug bedient
- Der Aufzug befindet sich zu beginn im Erdgeschoss (Etage 0)
- Im Erdgeschoss steigen vier Personen in den Aufzug
- Diese möchten jeweils in die Etagen 5, 15, 2 und 9 transportiert werden
- Auf der Etage 5 steigen Personen zu, die auf den Etagen 4, 10 und 1 aussteigen wollen und auf dem Weg nach Etage 1 steigen dann weitere Personen zu, die auf den Etagen 8, 6 und 14 austeigen wollen
- Aus Effizienz- und Brandschutzgründen soll der Aufzug nach Abarbeitung aller Haltewünsche zurück zum Erdgeschoss bewegen

Aufzugalgorithmus in Echtzeit (Demonstration)

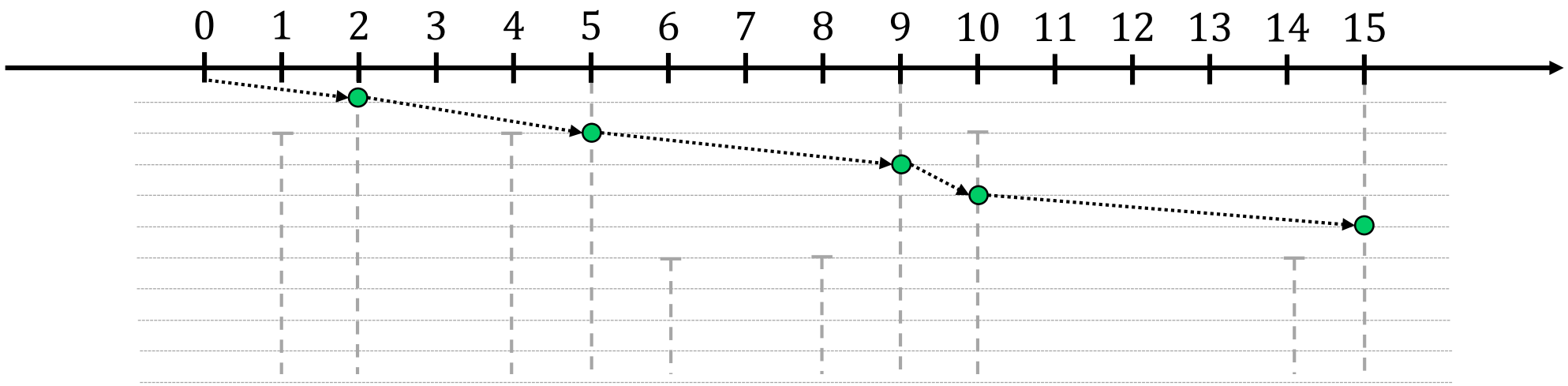
$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$
$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$



0 □□□□□□□□□□□□□□□□

Aufzugalgorithmus in Echtzeit (Demonstration)

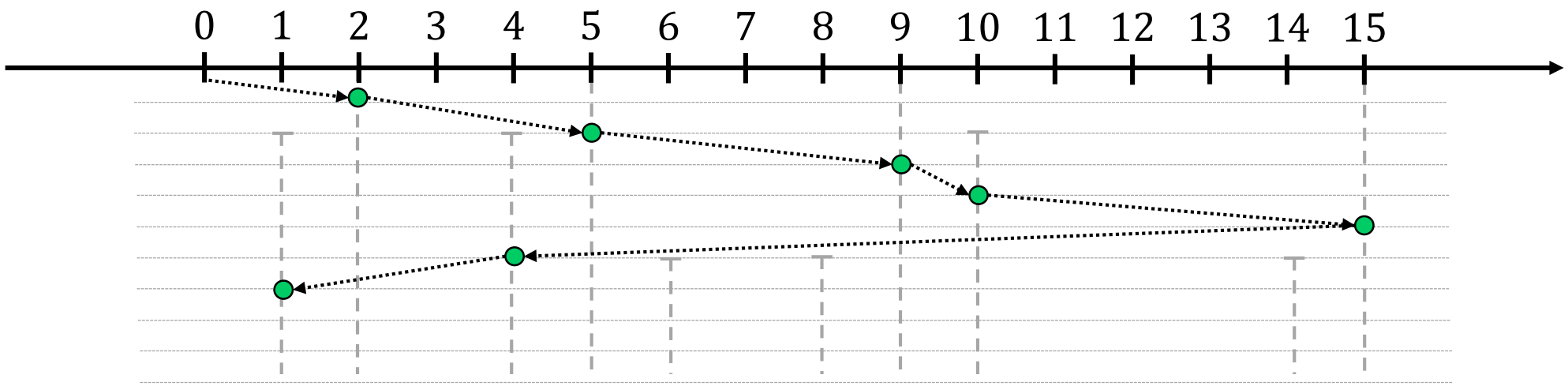
$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$
$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$



0 | 2 | 5 | 9 | 10 | 15 | | | | | |

Aufzugalgorithmus in Echtzeit (Demonstration)

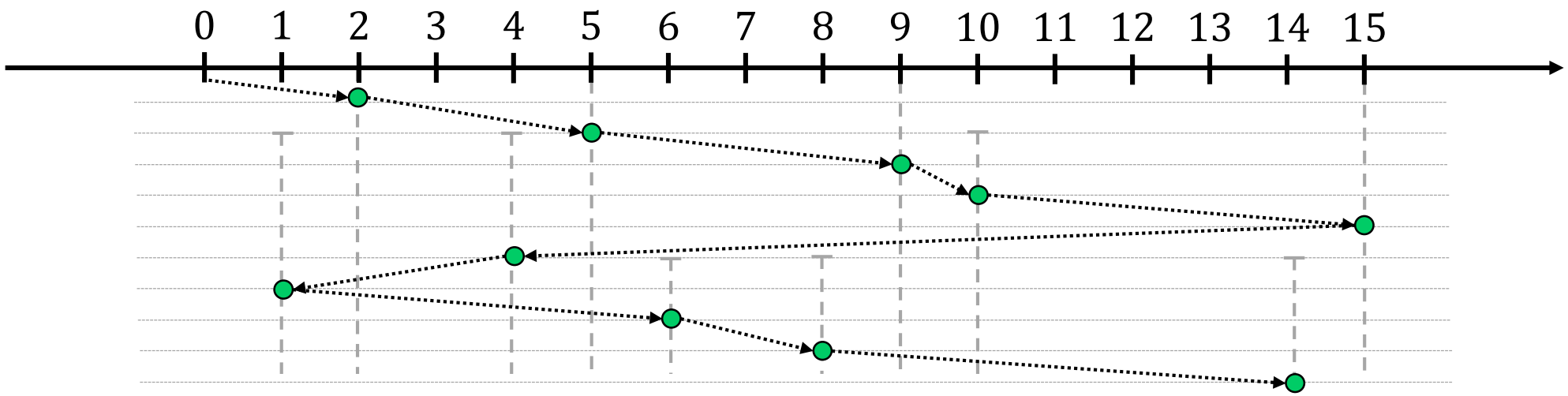
$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$
$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$



0 | 2 | 5 | 9 | 10 | 15 | 4 | 1 | | | |

Aufzugalgorithmus in Echtzeit (Demonstration)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$
$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

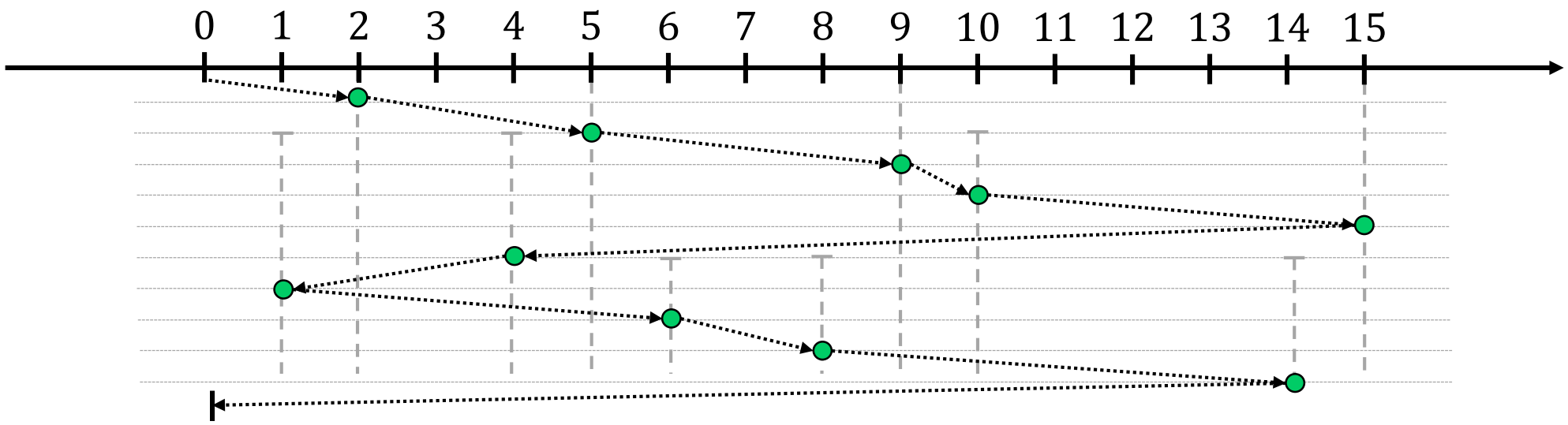


0 | 2 | 5 | 9 | 10 | 15 | 4 | 1 | 6 | 8 | 14

Aufzugalgorithmus in Echtzeit (Demonstration)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{5,15,2,9\},\ L_2 = \{4,10,1\},\ L_6 = \{8,6,14\}$$

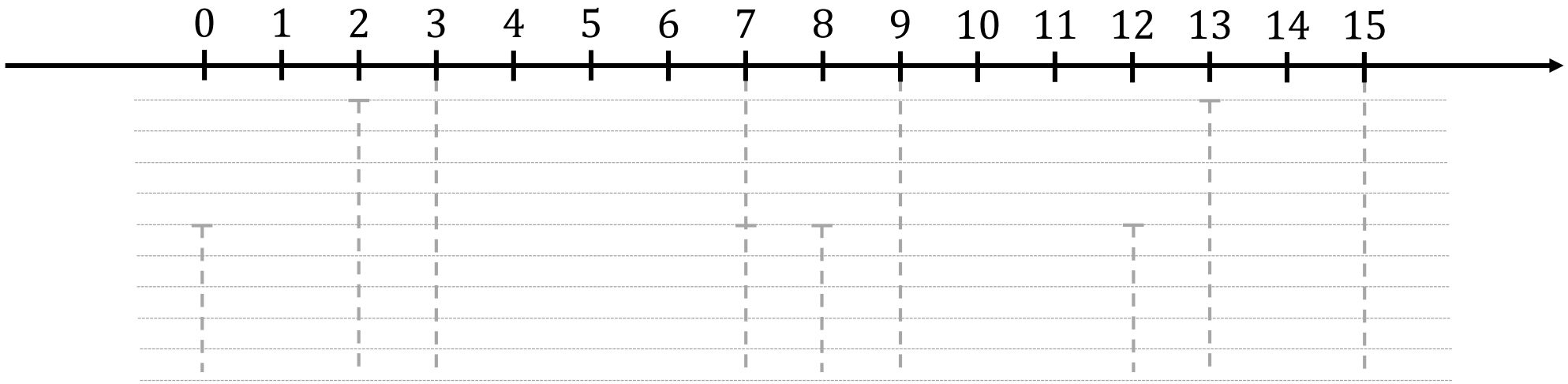


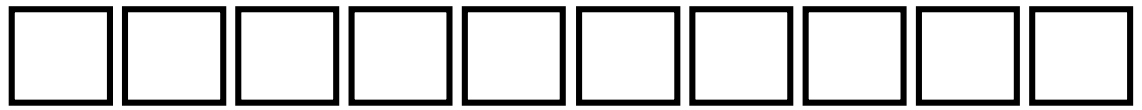
0 | 2 | 5 | 9 | 10 | 15 | 4 | 1 | 6 | 8 | 14 | 0

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$

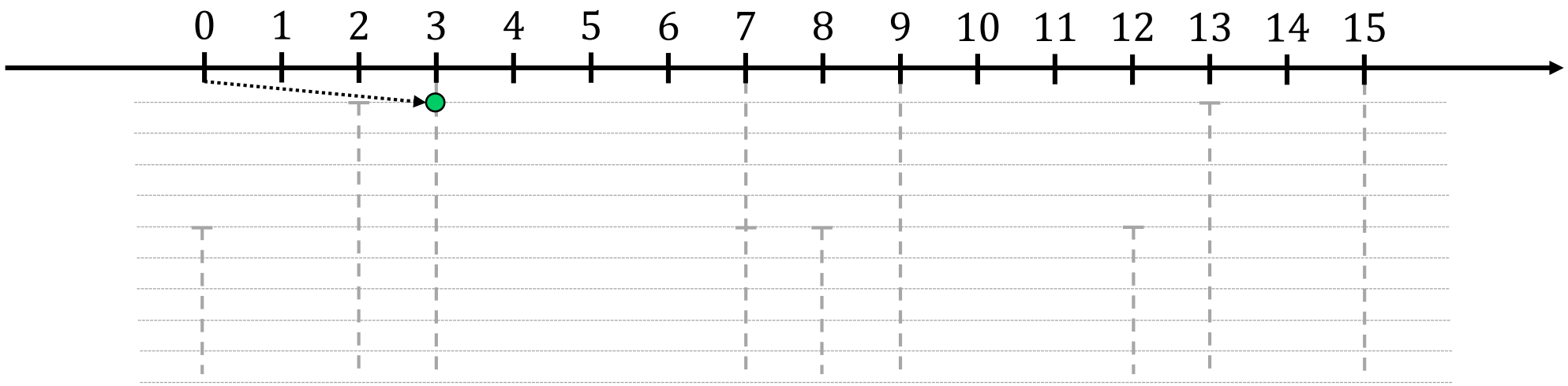


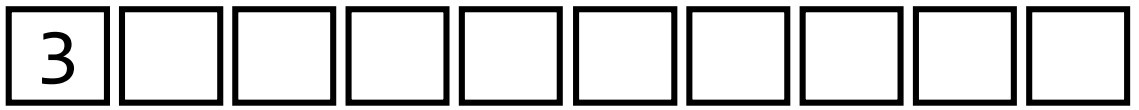


Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$

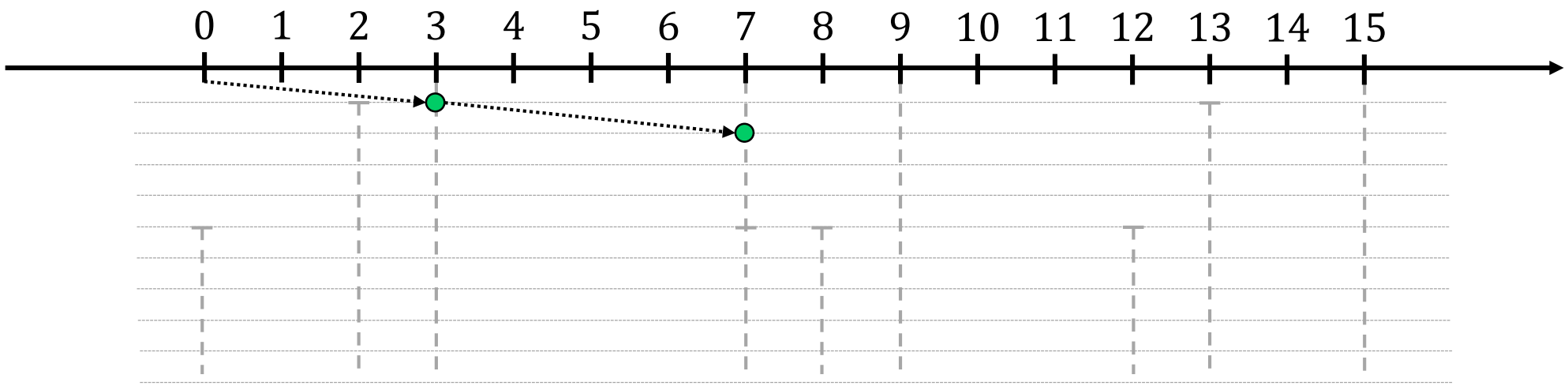




Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$

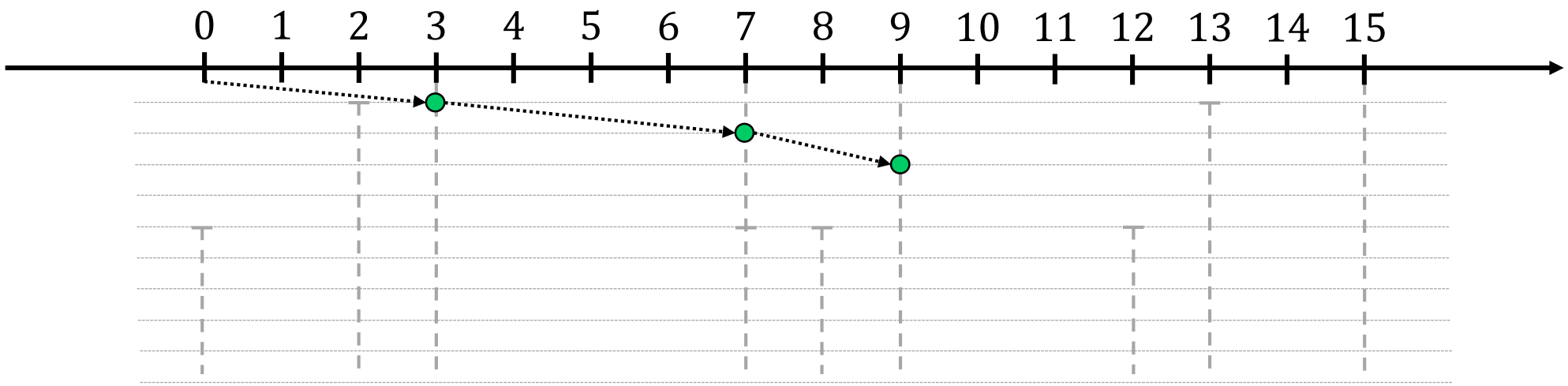


3	7								

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$

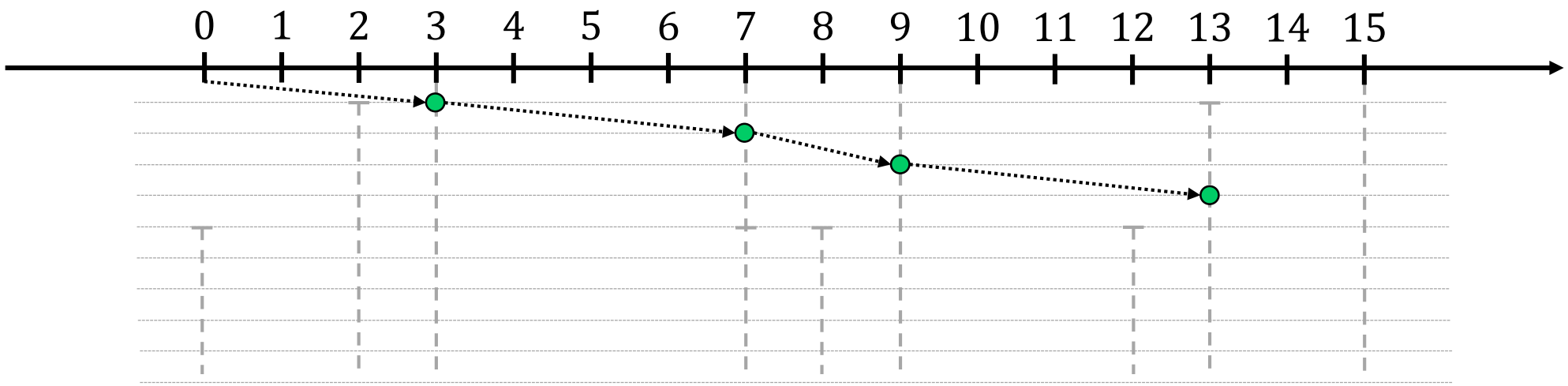


3	7	9							

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

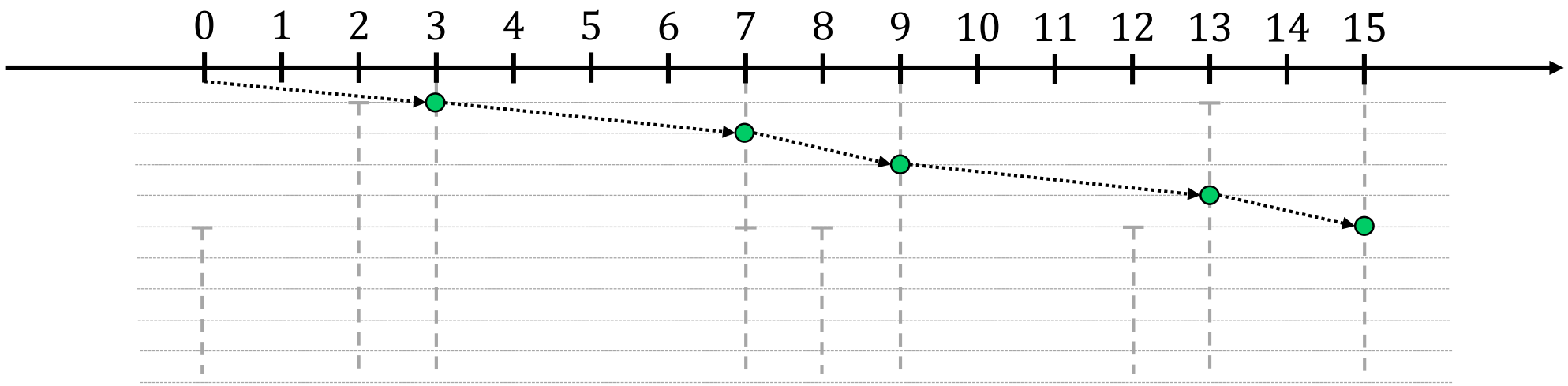
$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$



3	7	9	13						

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

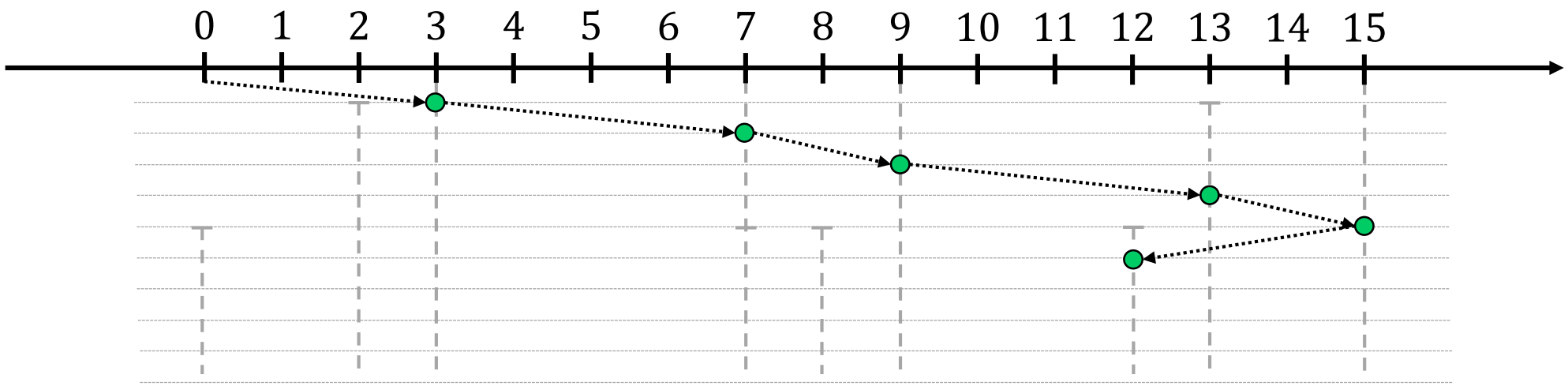
$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$
$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$



3	7	9	13	15					

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

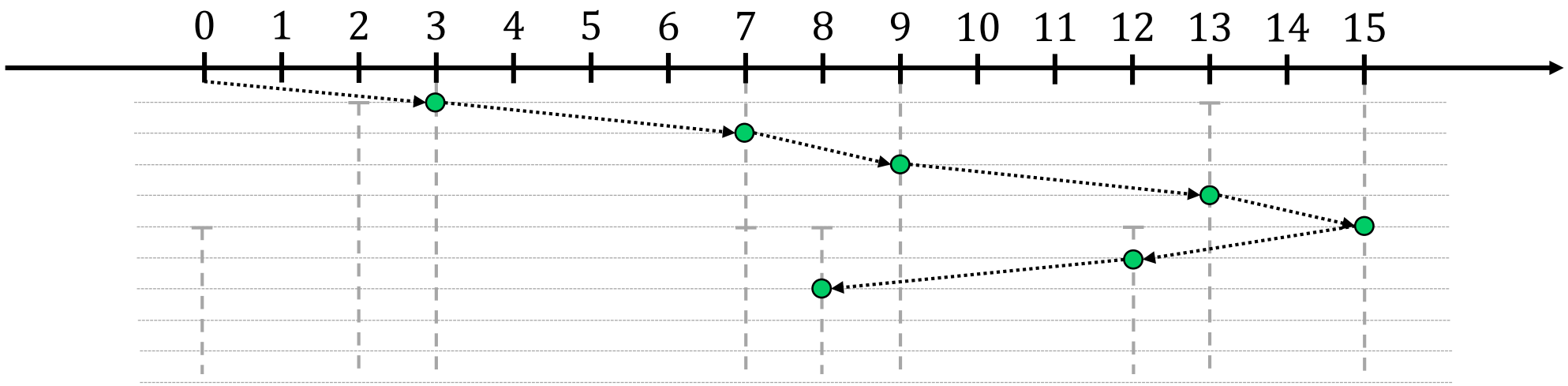
$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$
$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$



3	7	9	13	15	12				

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

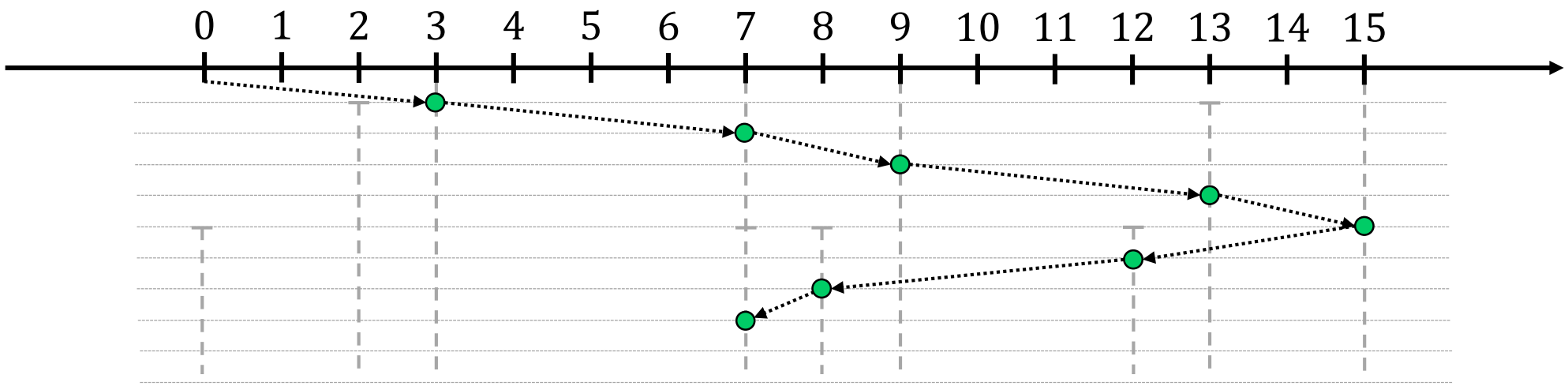
$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$
$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$



3	7	9	13	15	12	8			

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$
$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$

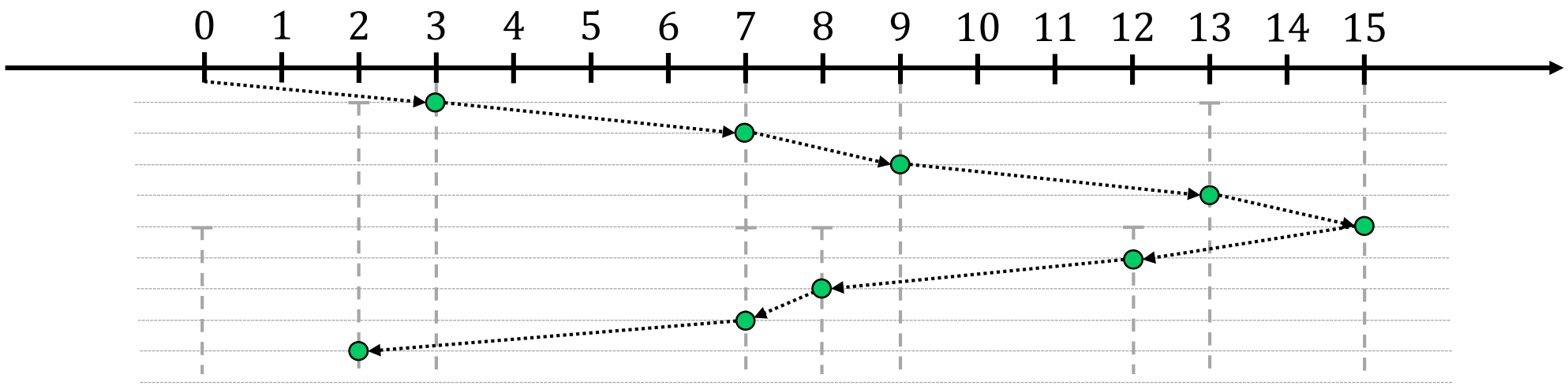


3	7	9	13	15	12	8	7		

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$

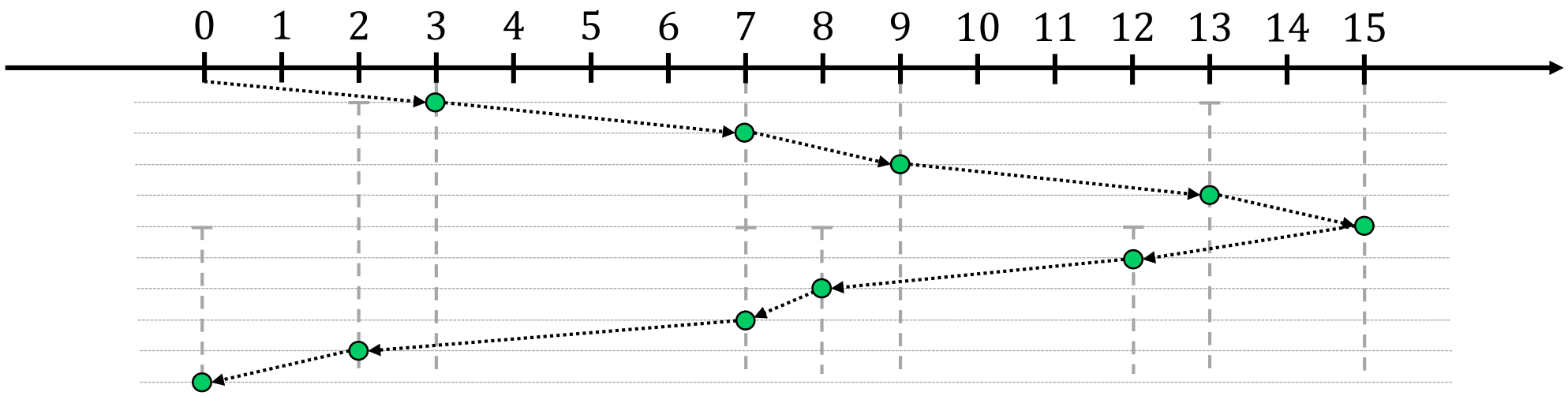


3	7	9	13	15	12	8	7	2	

Aufzugalgorithmus in Echtzeit (Übungsbeispiel)

$$N = 16, \text{-}pos = 0, \text{-}dir = \uparrow$$

$$L_0 = \{3,7,9,15\},\ L_1 = \{2,13\},\ L_5 = \{0,7,8,12\}$$

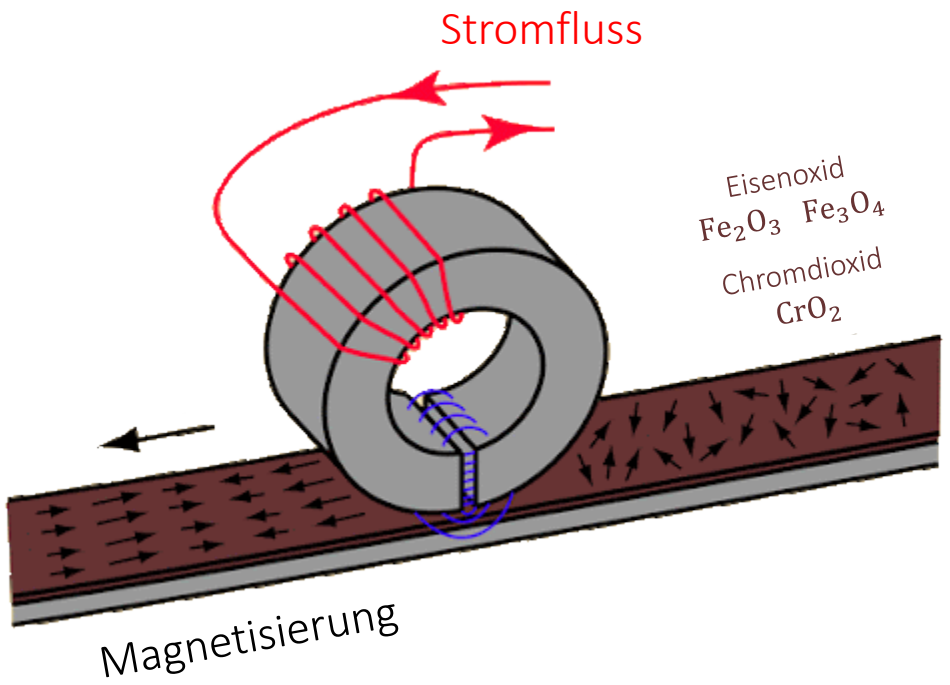


3	7	9	13	15	12	8	7	2	0

Datenbänder und Bandlaufwerke

- Ein **Bandlaufwerk** schreibt im Gegensatz zur HDD die Daten nicht auf eine sich rotierende Scheibe, sondern auf ein magnetisierbares **Datenband**, das auf zwei Rollen gewickelt ist
- Datenbänder sind günstiger als HDDs und werden vor allem im professionellen Bereich als sog. **Backup-Lösung** eingesetzt

Magnetbänder

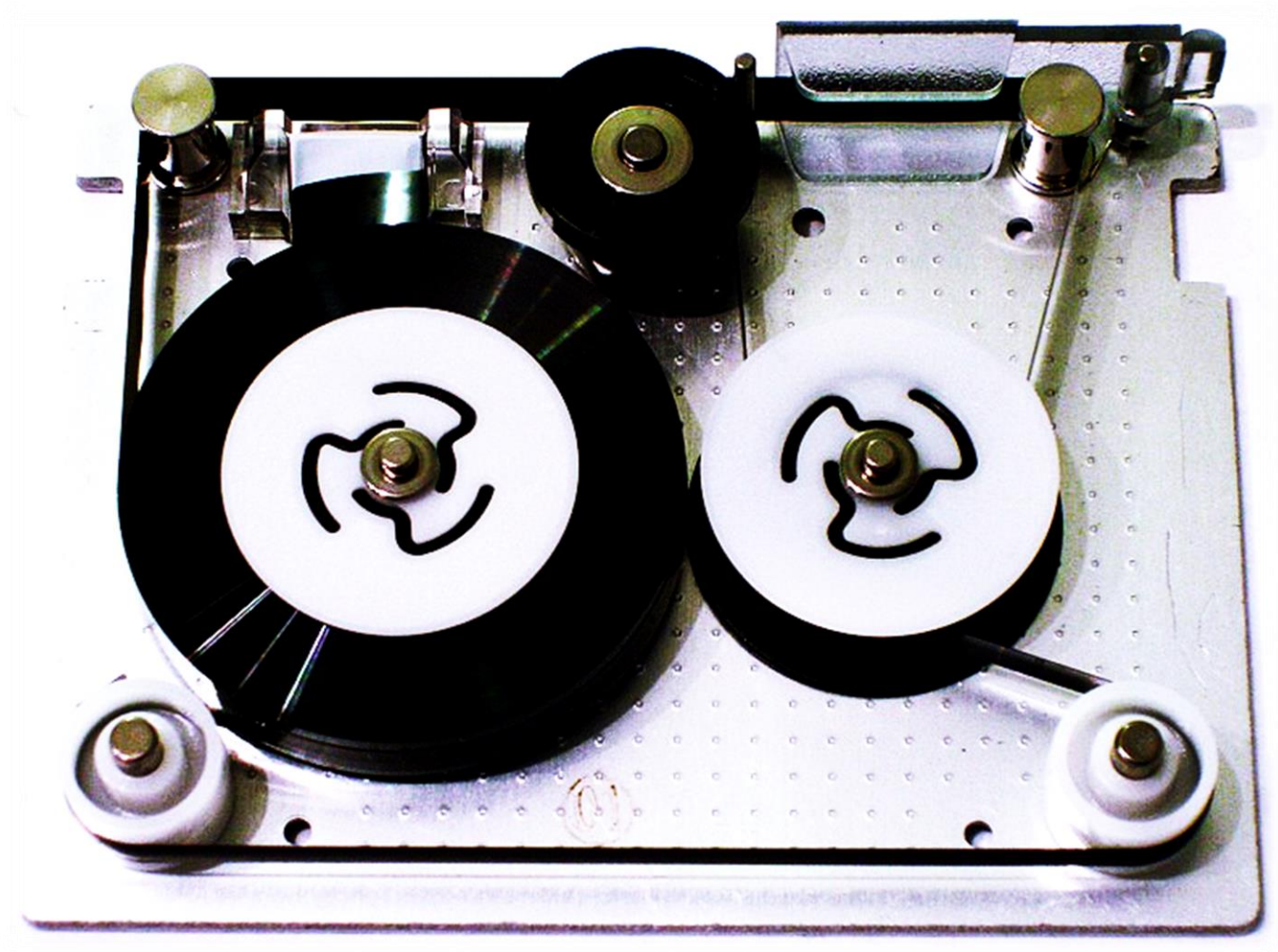


Bildquelle: http://hyperphysics.phy-astr.gsu.edu/hbase/Audio/tape2.html, abgerufen am 22.06.2024

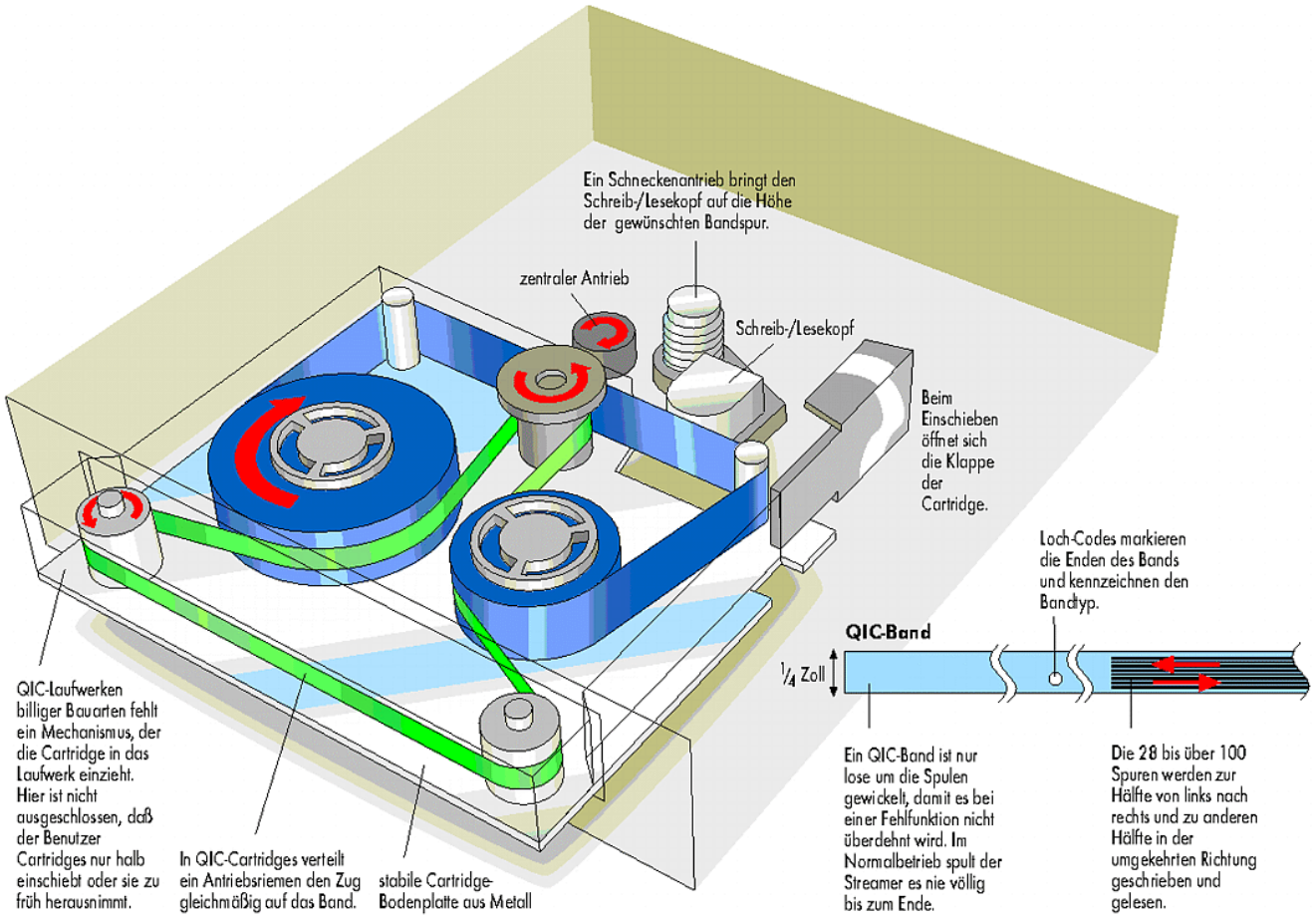
Bandlaufwerk-Technologien

- **Viertelzoll-Magnetband** (engl. **Quarter-Inch Cartridge**, kurz **QIC**): Spuren werden horizontal auf das Band aufgetragen, solche Bänder können bis zu $20\ \mathrm{GB}$ an Daten halten
- **Digitales Audioband** (engl. **Digital Audio Tape**, kurz **DAT**): Datenbänder die insbesondere für professionelle Audioanwendungen entwickelt wurden
- **Ultrium®-Band** (**Linear Tape-Open**, kurz **LTO**): Datenband auf den Datenmengen in der Größenordnung von TB abgespeichert werden können

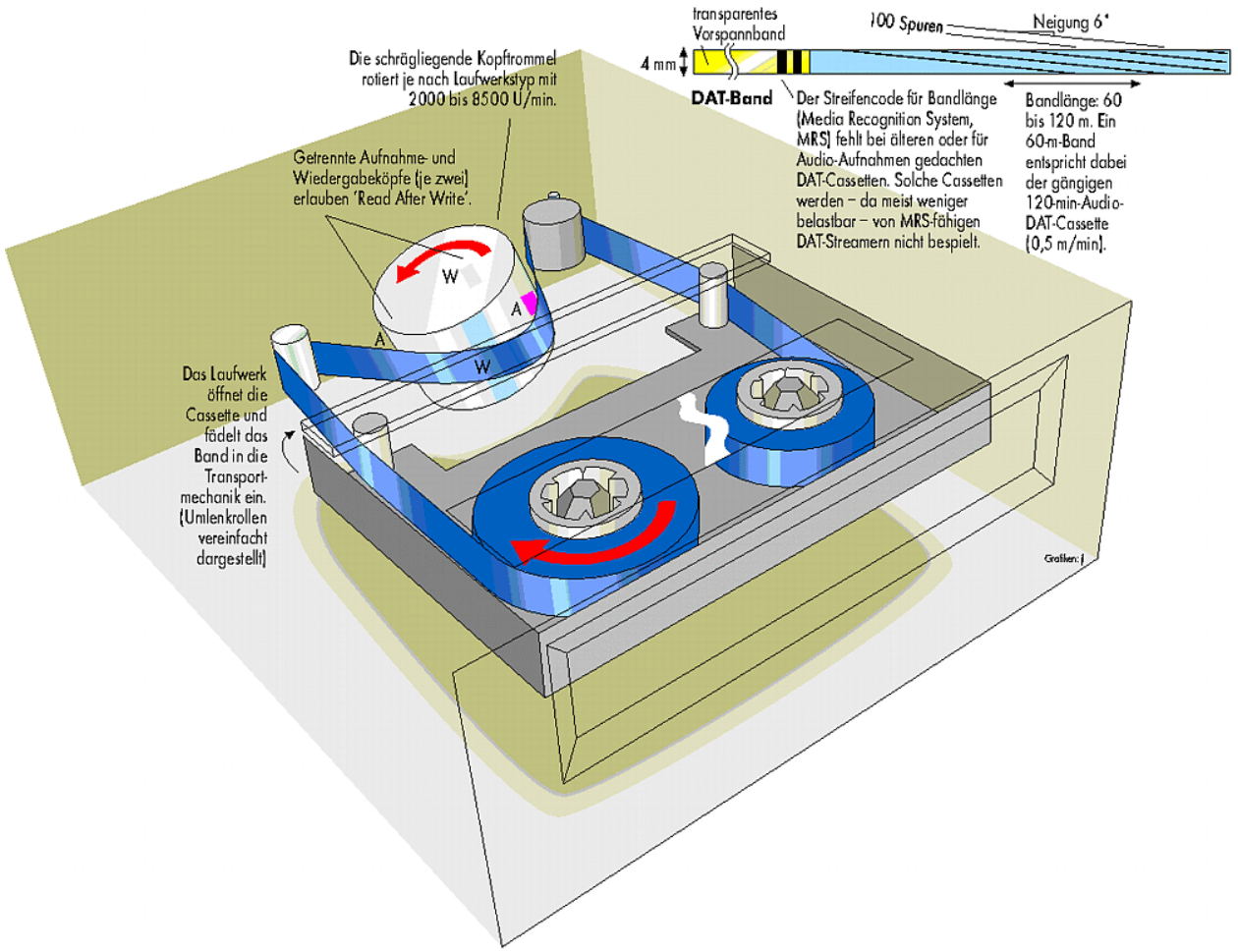
QIC-Kassette (Offen)



Aufbau eines QIC-Laufwerks



Aufbau eines DAT-Laufwerks



LTO-Kassette (Offen)

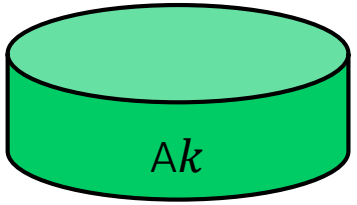
RAID

Plattenverbund und Datensicherheit

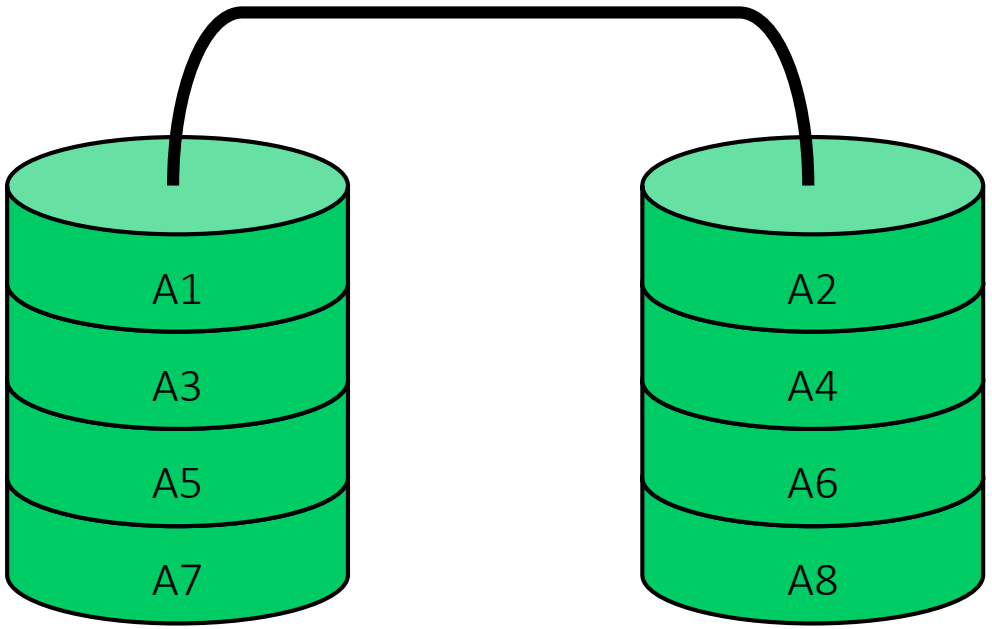
- Platten lassen sich zu **Plattenverbünden** (engl. **disc arrays**) zusammen schließen: Wird im System als 'ein großer' zusammenhängender Speicher gehandhabt
- Gespeicherte Daten werden zerlegt und auf verschiedene Platten **redundant** verteilt → Vorbeugung von Datenverlusten
- Dank der **Redundanzen** können im laufenden Betrieb gealterte Festplatten ohne Datenverlust ausgetauscht werden
- **S.M.A.R.T-Daten** enthalten Alterungsinformationen der Platte

RAID

- **Redundant array of inexpensive discs** (kurz **RAID**) basiert auf
  - **Spiegelung** (engl. **mirroring**)
  - Zerteilen und verteilen (engl. **striping**)
  - Paritätsinformation
- Die verschiedenen **RAID-Verfahren** sind durch $\mathrm{RAID}\ n$ bezeichnet
- RAID-Verfahren lassen sich miteinander kombinieren ($\mathrm{RAID}\ n\ m$)
- **Striping-Parameter**: Länge der Einheiten, in welche Daten Zerlegt werden

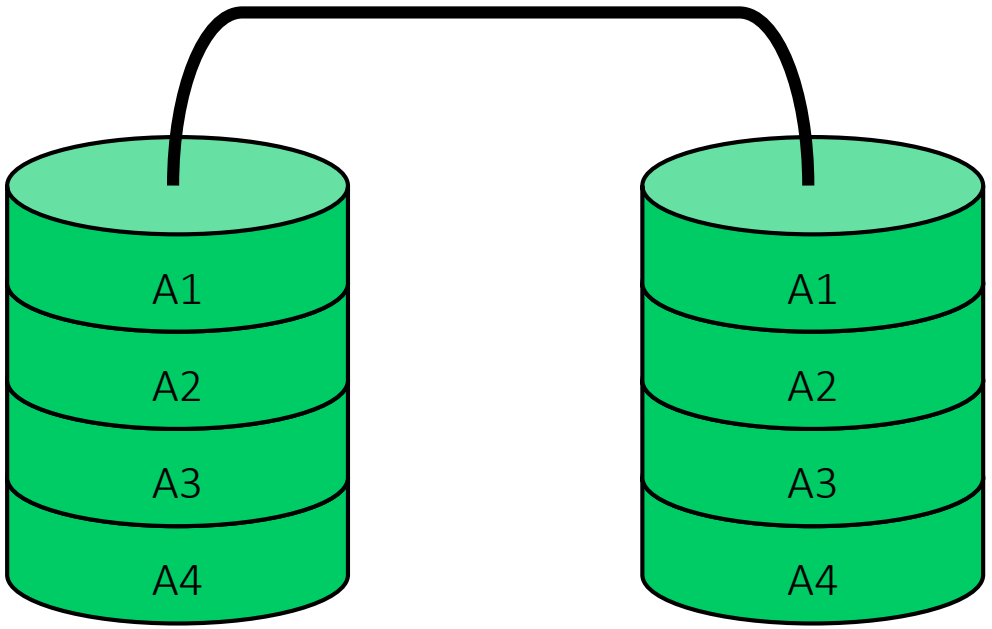


RAID 0



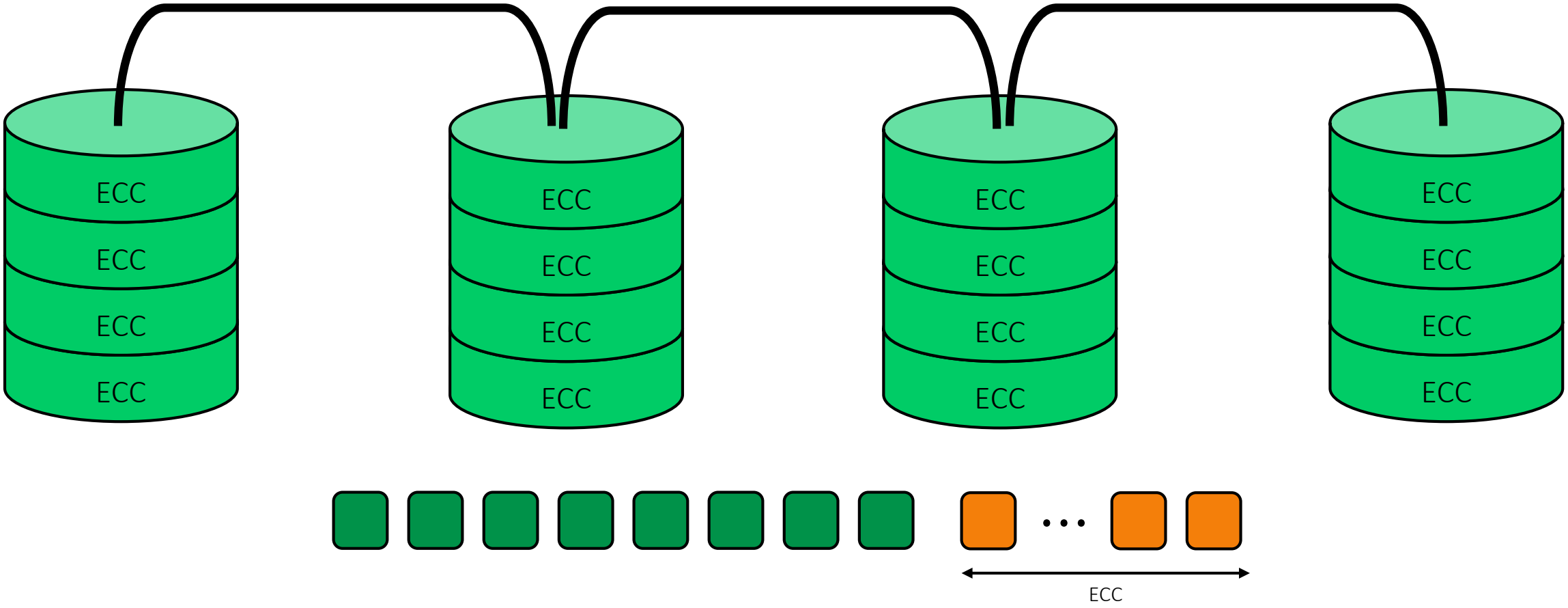
Daten werden zerlegt und auf (mindestens zwei) kleinere Festplatten verteilt. Diese werden zu einem logischem Laufwerk zusammengeschaltet. Es bestehen bei dieser Variante keine Redundanzen.

RAID 1



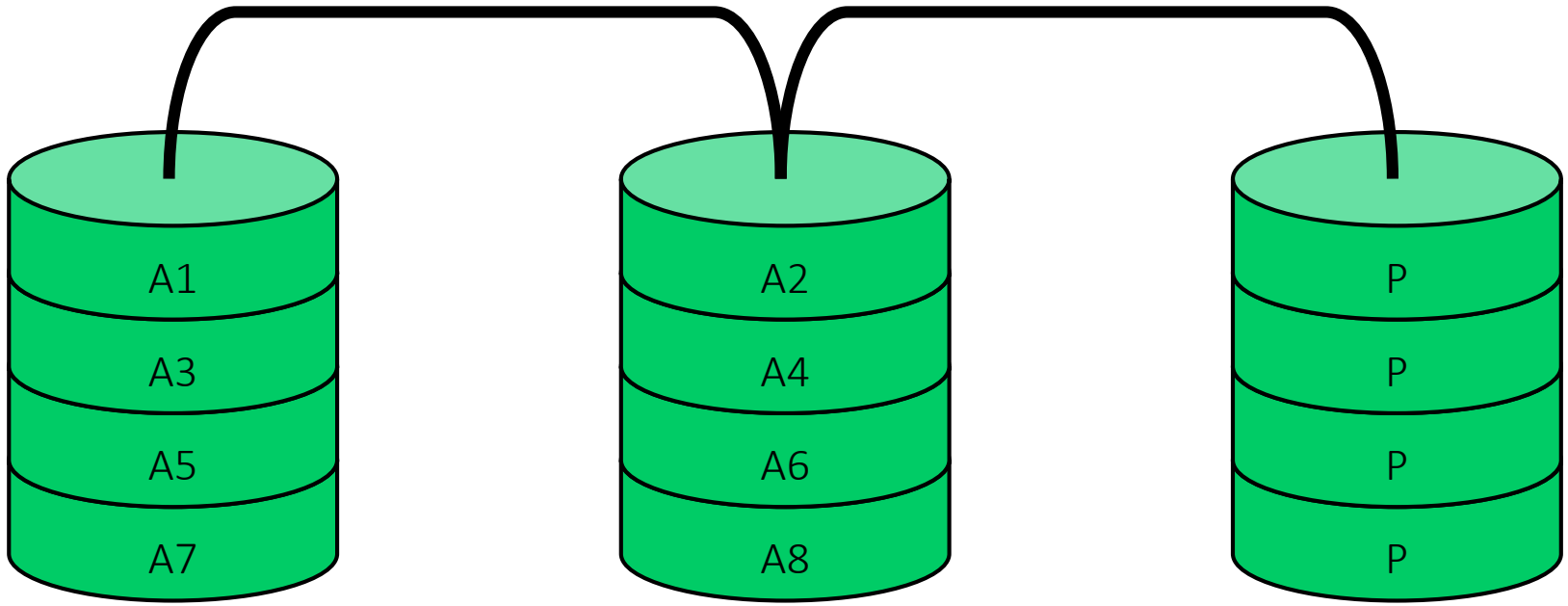
Eine identische Sicherheitskopie wird auf einer weiteren Festplatte gespeichert.

RAID 2



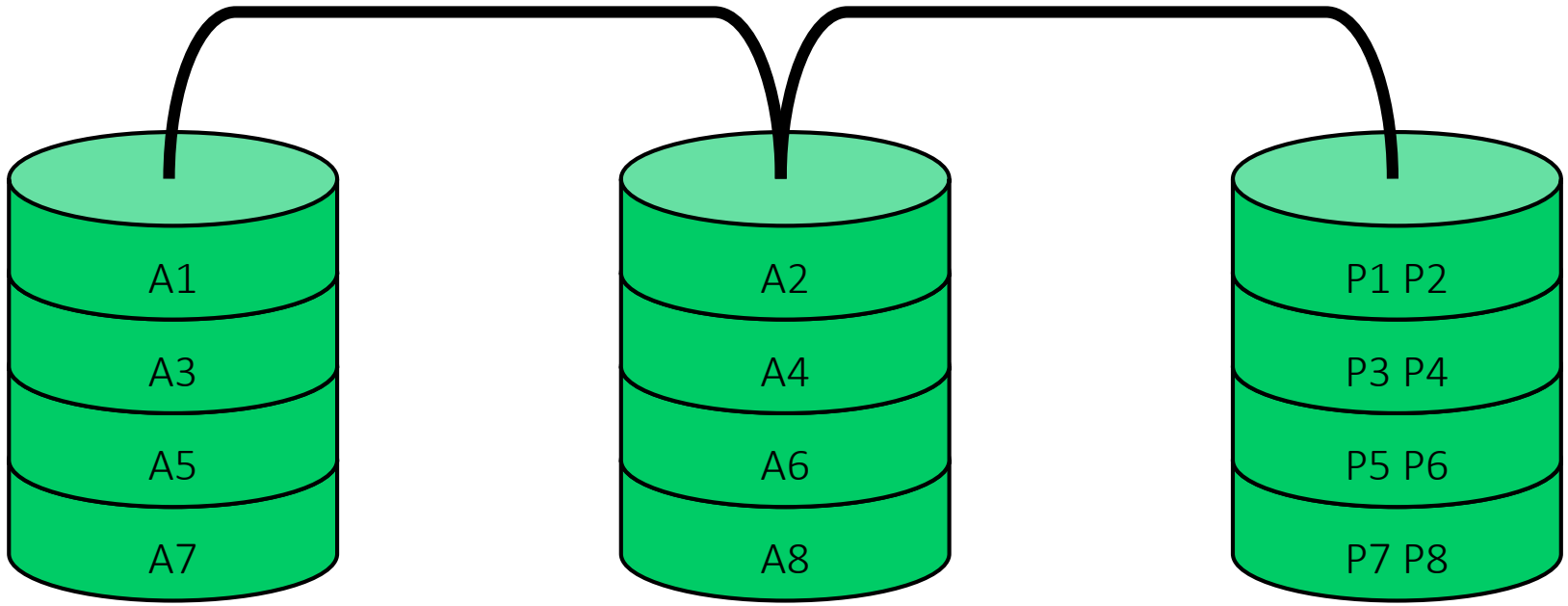
Jedes Byte wird im Sinne eines fehlerkorrigierendem Codes gespeichert. Zusätzlich Striping mit Striping-Parameter von einem Bit. Für dieses Verfahren ist sehr spezielle Hardware nötig.

RAID 3



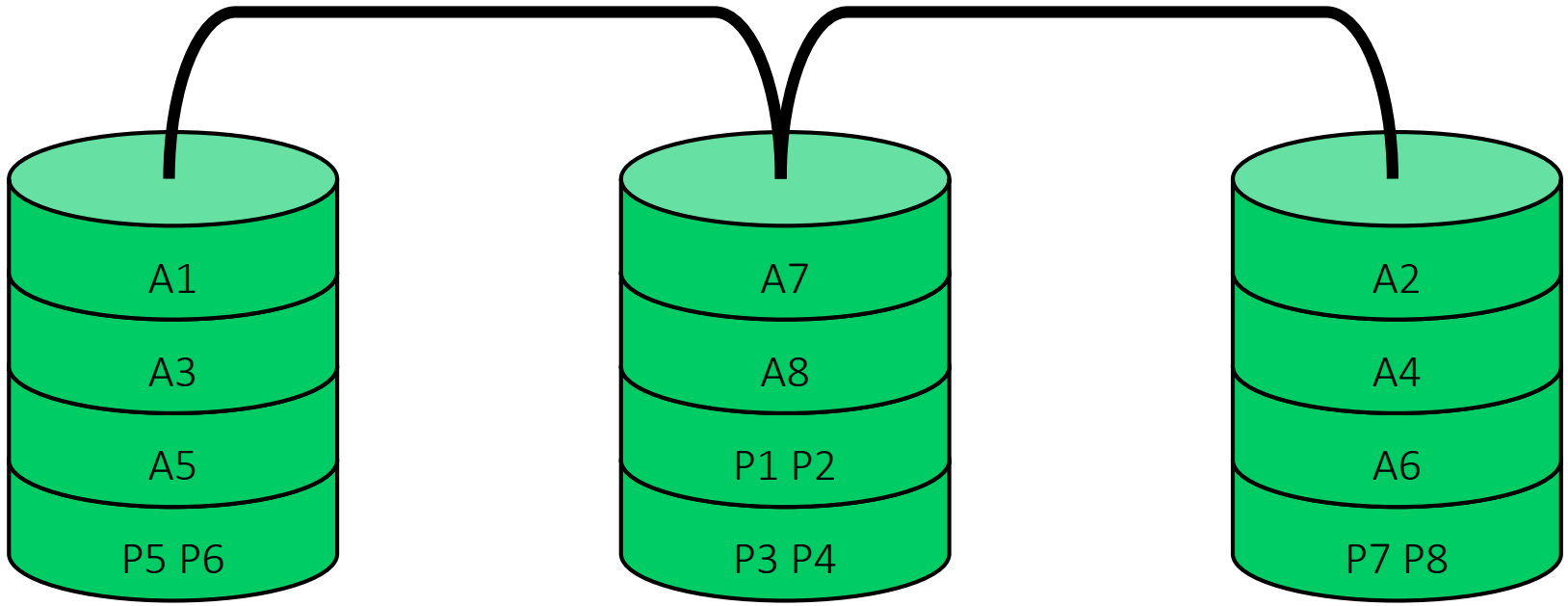
Striping mit einem Striping-Faktor von einigen wenigen Bytes. Zu jedem dieser Gruppen von Bytes wird auf einer **Paritätsplatte** das Paritätsbit gespeichert.

RAID 4



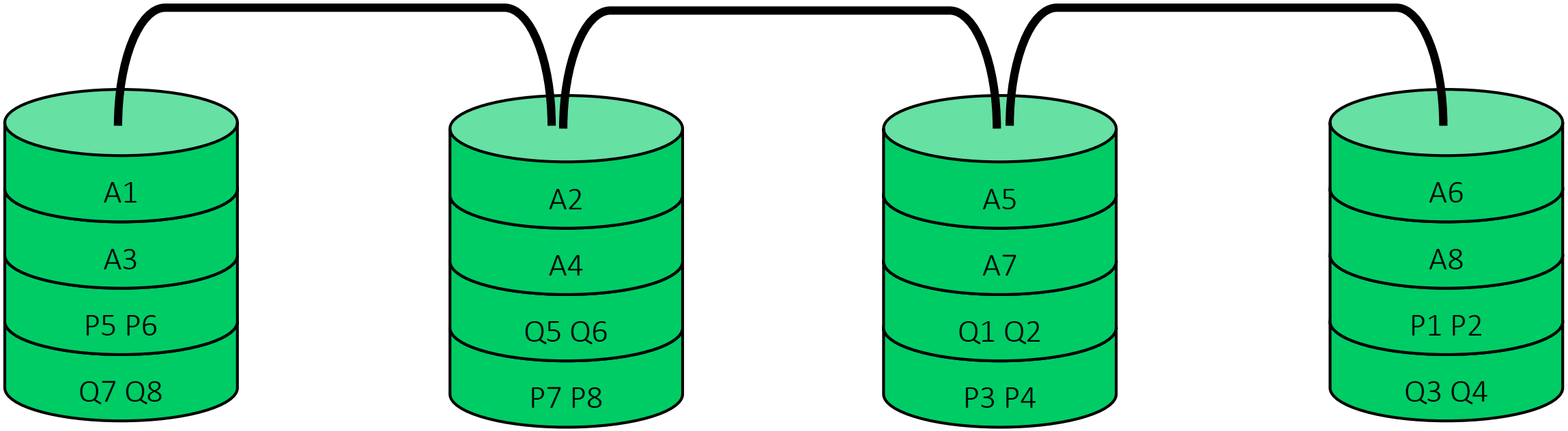
Wie RAID 3 aber mit einem Striping-Parameter in Blöcken.

RAID 5



Wie RAID 4 aber mit Striping der Parität und Verteilung auf alle Platten um die Paritätsplatte zu entlasten.

RAID 6



Wie RAID 6 aber mit zusätzlichen q-Paritätsbits.