ADSII3ILV

Algorithms and Data Structures II

Recap

Dr. Alessio Gambi

Algorithms

Definitions

- Sets of **unambiguous** instructions for solving problems or subproblems in a **finite** amount of **time** (must **terminate**) using a **finite** amount of **data**
- **Well-defined computational procedures** that take some **input** and produce some **output**
- Sequences of **computational steps** that transform the input into the **intended** output
- **Correct** algorithms **halt** with the **correct output** for every input (satisfying the appropriate constraints)

Open Discussion

- Why must instructions be **unambiguous**?
- Why must time and data be **finite**?
- Why do we talk about **algorithms** and not **programs**?

Programs

- Programs can be written in **many programming languages**:
  - Imperative or procedural
  - Functional
  - Declarative
  - Object-oriented
- Each programming language, despite sharing a common set of concepts (variables, statements, branching, looping) has its own peculiarity.
- Do we need all those **details** to solve a problem?

Algorithms abstract from the implementation

- Suppose I want to describe a program for you to write, but I don't know which language you will use
- We need a way to describe a program which is independent of a specific language
- Algorithms can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a **precise description** of the computational procedure to be followed

Pseudocode

- A way of expressing algorithms that uses a mixture of English phrases and indentation to make the steps in the algorithm explicit
- Pseudocode is not case sensitive, and there are no grammar rules

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

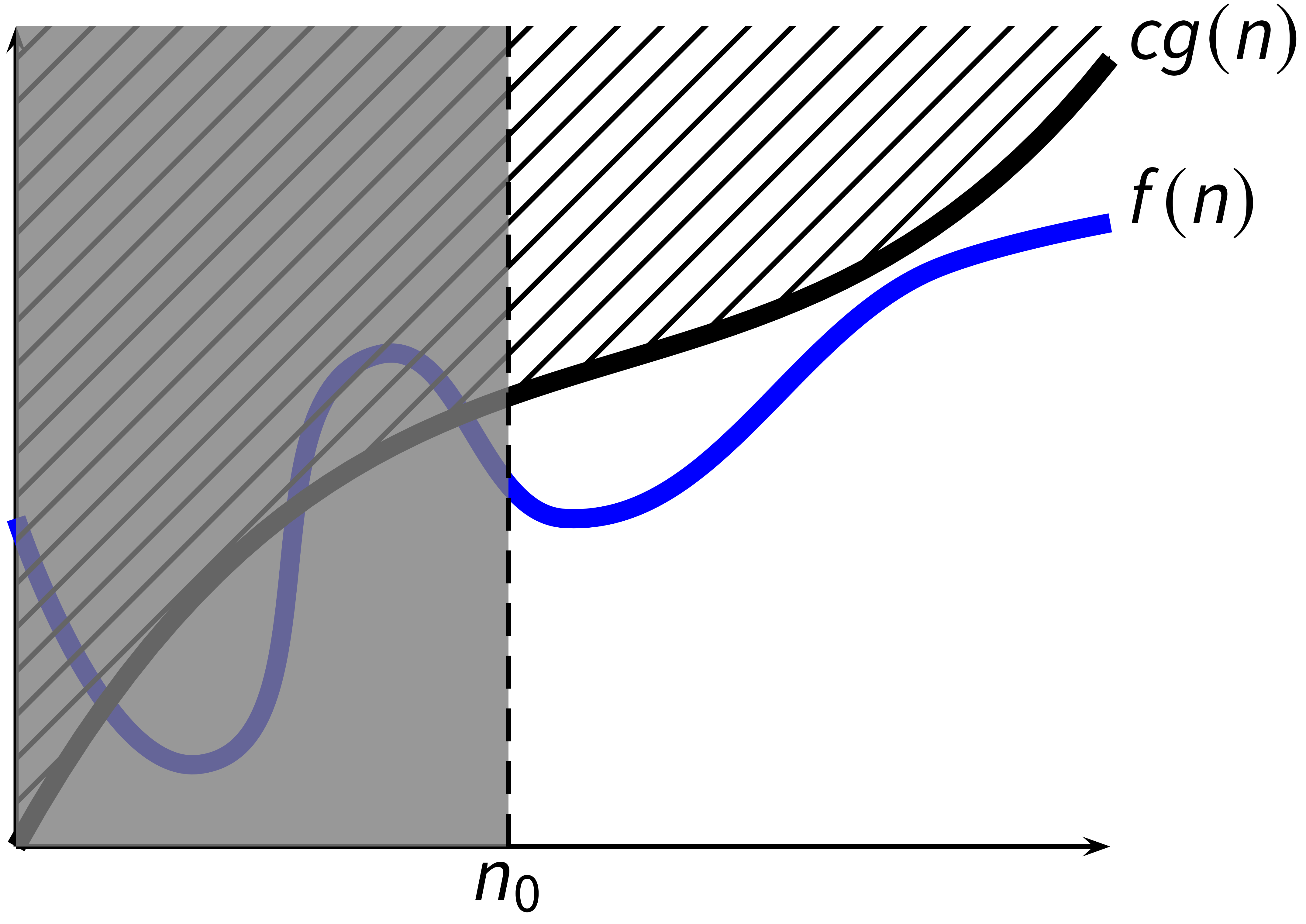
```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

Common Conventions

- **Indentation** indicates block structure (like Python) and applies to loops, branches, etc. Reduced clutter compared to BEGIN...END blocks (like shell)
- **Looping** constructs (e.g, while, for) and **conditional** constructs have interpretations similar to those in common programming languages
- **Comments** are introduced using the symbol **"//"**
- **Assignments** are defined using the left arrow symbol "⟵"
- **Variables** are local unless declared global
- **Arrays** use the standard notation with square brackets
- **Compound** data are treated as objects; we use the dot notation to access their attributes
- **Return** statements can return multiple values at once (like Python)

O-Notation

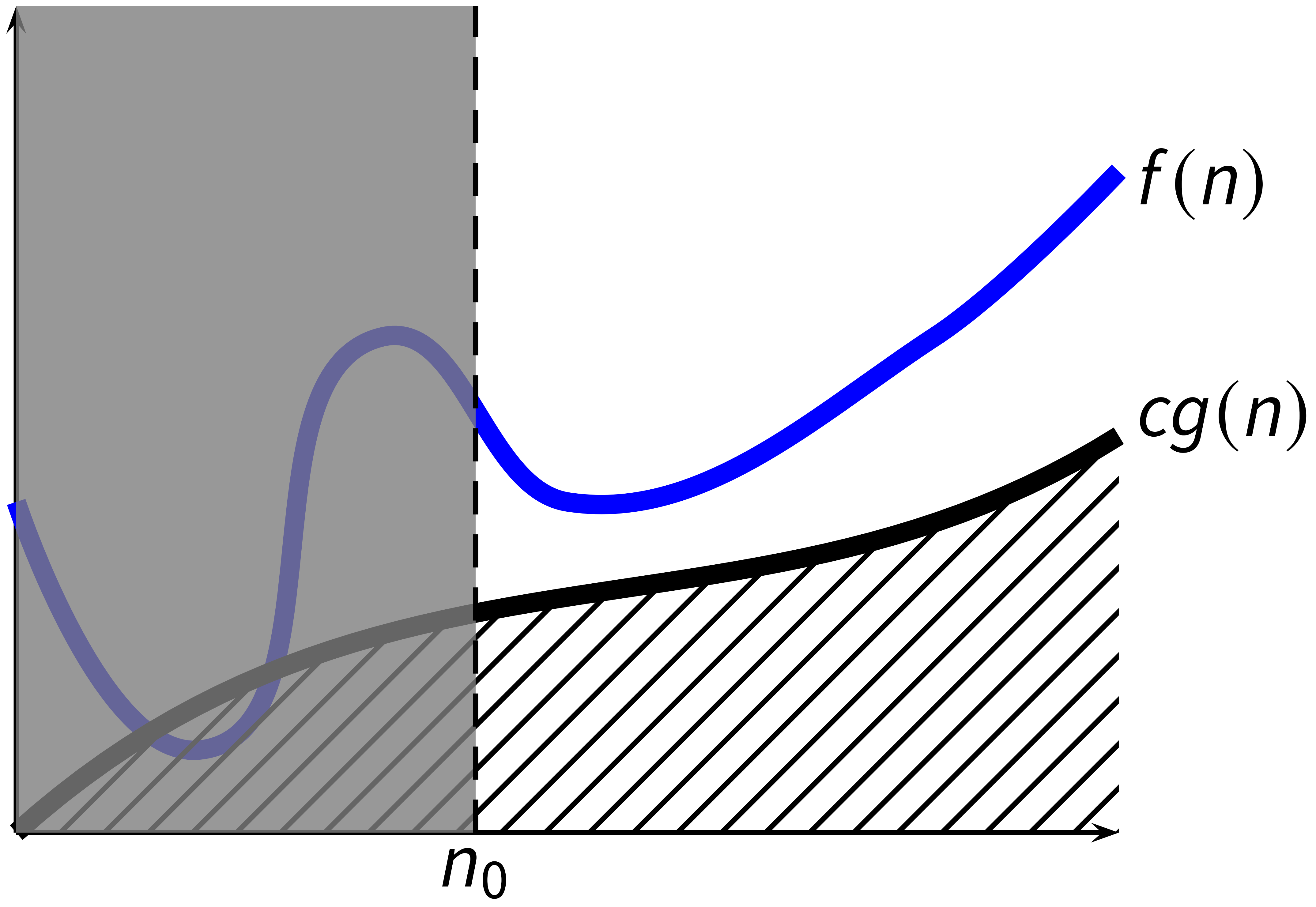
- Given a function g(n), we define the **family of functions** O(g(n))



"f(n) is big-oh of g(n)"

Ω-Notation

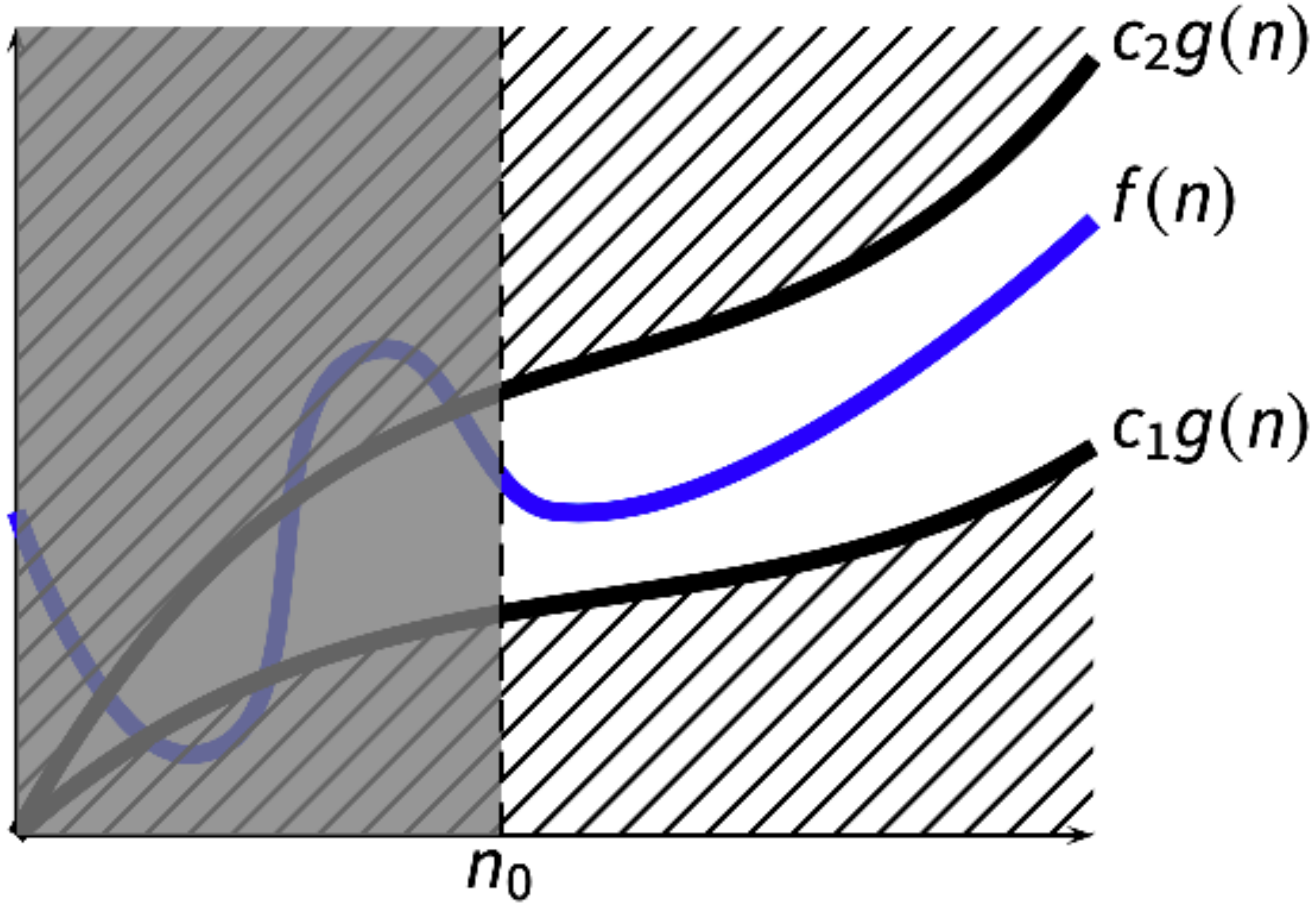
- Given a function g(n), we define the **family of functions** Ω(g(n))



$$\Omega(g(n))=\{f(n) : \exists\, c>0, \exists\, \exists\, n_0>0 : 0 \leq cg(n) \leq f(n) \text{ for } n \geq n_0\}$$

Θ-Notation

- Given a function g(n), we define the **family of functions** Θ(g(n))



$$\Theta(g(n)) = \{f(n) :\ \exists\, c_1 > 0, \exists\, c_2 > 0, \exists\, n_0 > 0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for } n \geq n_0\}$$

Data Structures

- Store and organize data in order to **facilitate** access and modifications
- No Silver Bullet: No single data structure works well for all purposes! Thus, it is important to know the **strengths** and **limitations** of several of them
- Algorithms may require several different types of operations to be performed on data (e.g, insert elements into a set, test membership)
- **Operations** can be grouped into **two main categories**:
  - Queries/observers return information about the data without modifying it
  - Modifying operations change the state of the data

Abstract Data Types

- Data type: a **set of values** and a **set of operations** on those values
  - We refer to the set of operations as the API
- Abstract data type: a data type whose data and operations are specified **independently** of any particular **implementation**
- Most programming languages are able to create some implementation of an abstract data type (for instance, based on the concept of Class)

Abstract Data Types

- Data type: a **set of values** and a **set of operations** on those values
  - We refer to the set of operations as the API
- Abstract data type: a data type whose data and operations are specified **independently** of any particular **implementation**
- Most programming languages are able to create some implementation of an abstract data type (for instance, based on the concept of Class)

- Can you name some abstract data type?

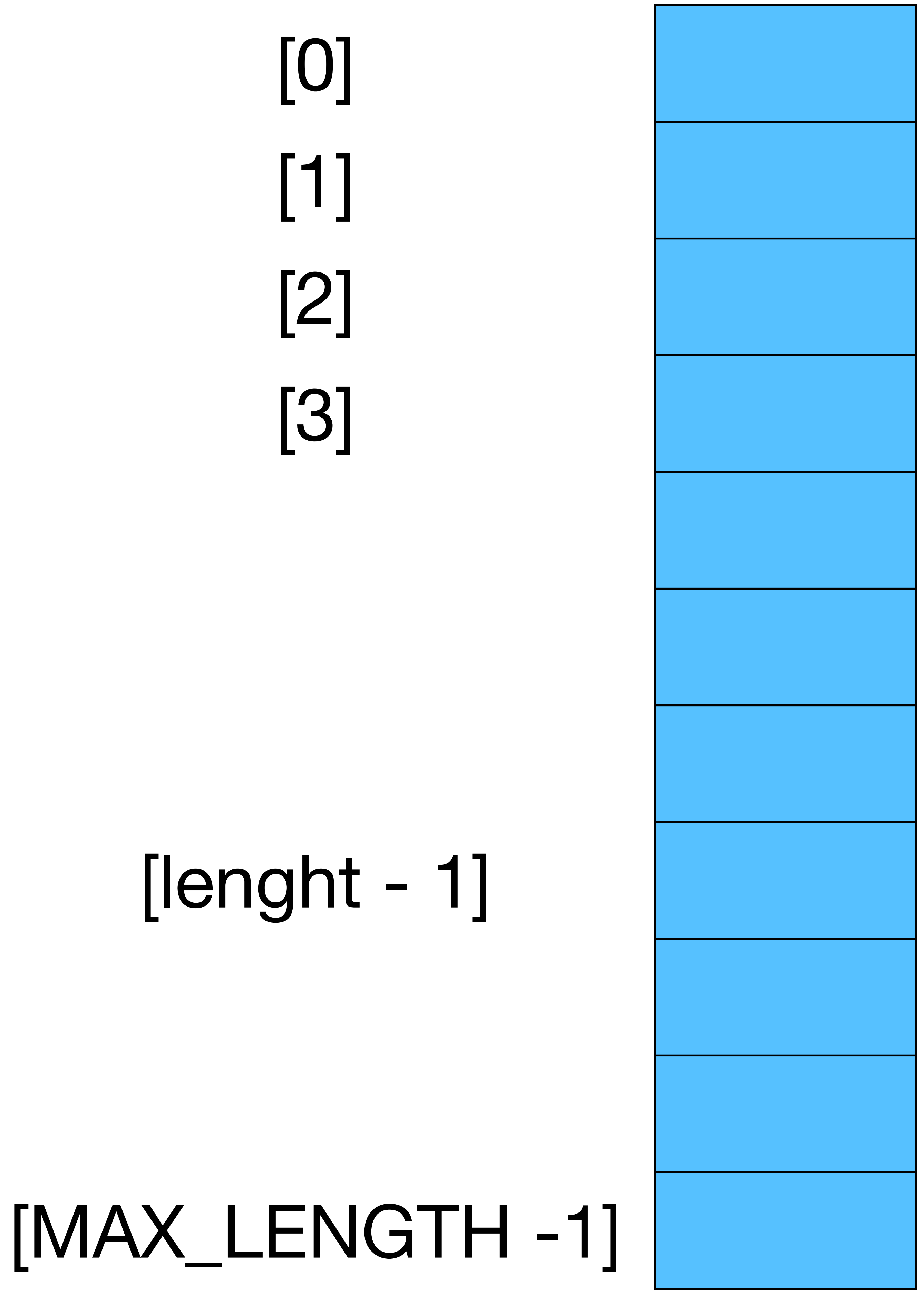
Collections/Bags

- Collection of data that clients want to **store together**
- **Generic container** of items
- Provide **no predefined** way to **access** the stored data
- Operations:
  - add an element
  - check if empty
  - return the size/count of the elements

Sets

- **Generic container** of items
- Ensure that **no duplicate** elements are contained in the same set
- Provide **no predefined** way to **access** the stored data
  - no specific ordering of the elements
- Operations:
  - add an element
  - check if empty
  - return the size/count of the elements

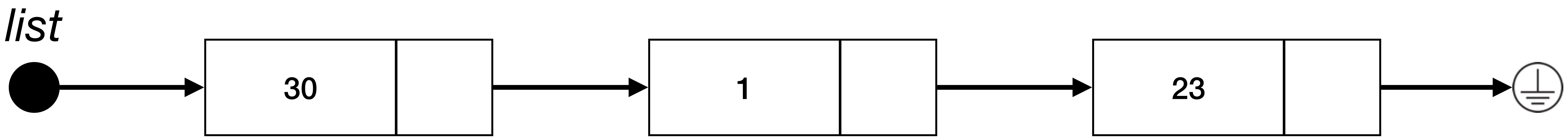
Arrays/Direct Access Tables



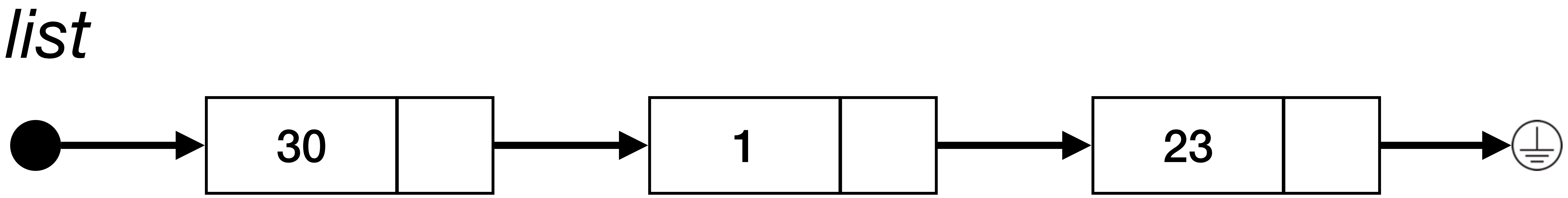
- **Random/Direct** access:
  - Access element at position **k** directly *arrayname*[k]
  - Positions start at **0**
- **Limited** size (Maximum allowed size is MAX_LENGTH)
- Insert/Remove operation:
  - **without** shifting (lookup table)
  - **with** shifting (list)

Linked Lists

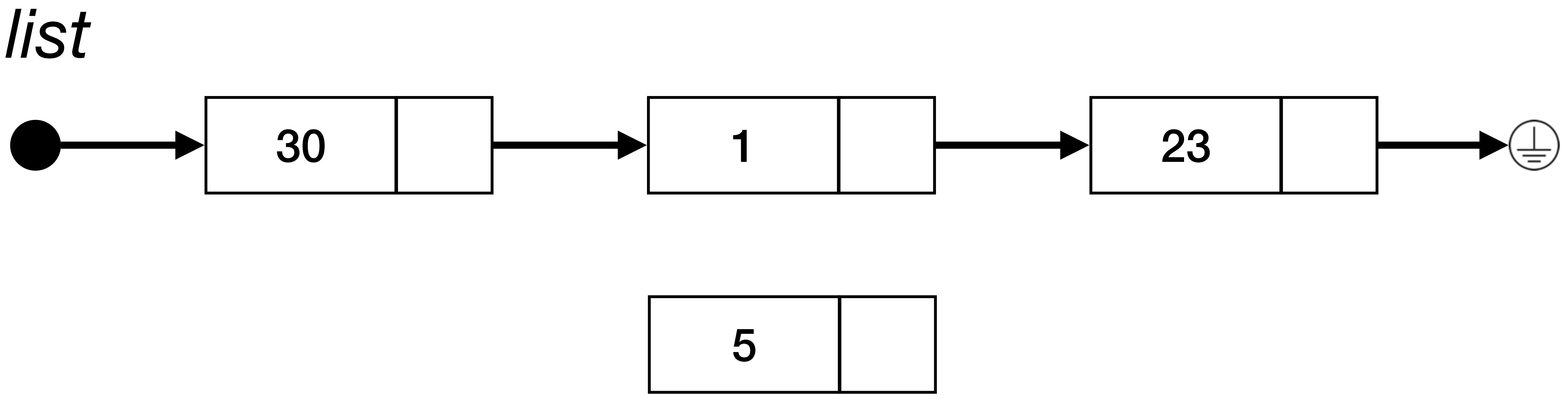
- Based on the concept of a **node** that holds two pieces of information:
  - the **item** that must be stored (sometimes called Key)
  - the **pointer** to the **next node** in the list
- **Sequential access**: Access element at position k requires navigating all the elements before it. Useful to have an Iterator (*current, getNext, hasNext*)
- *Virtually* unbounded
- Insert/Delete have always constant cost



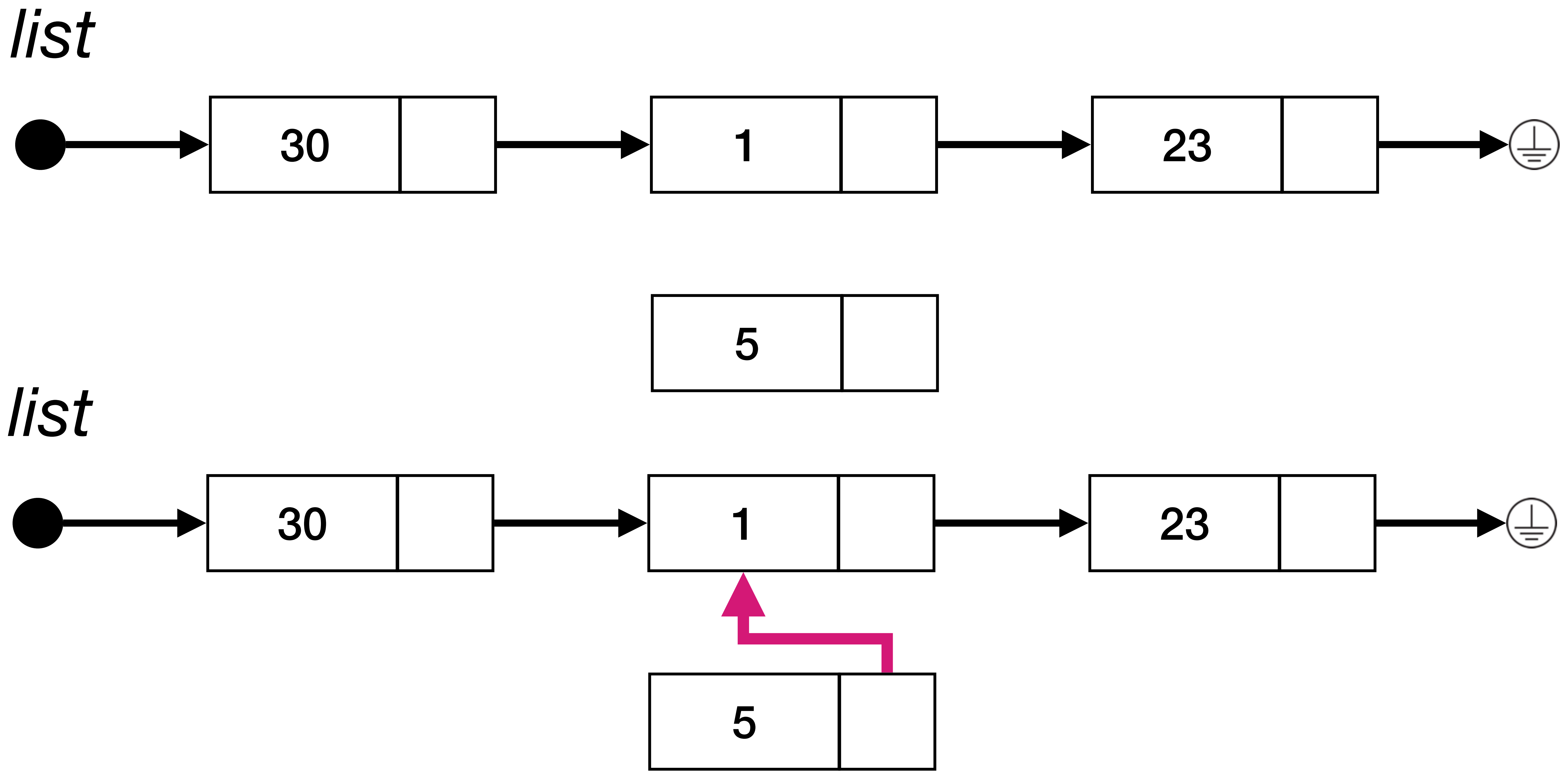
Insert element: add(value=5, position=1)



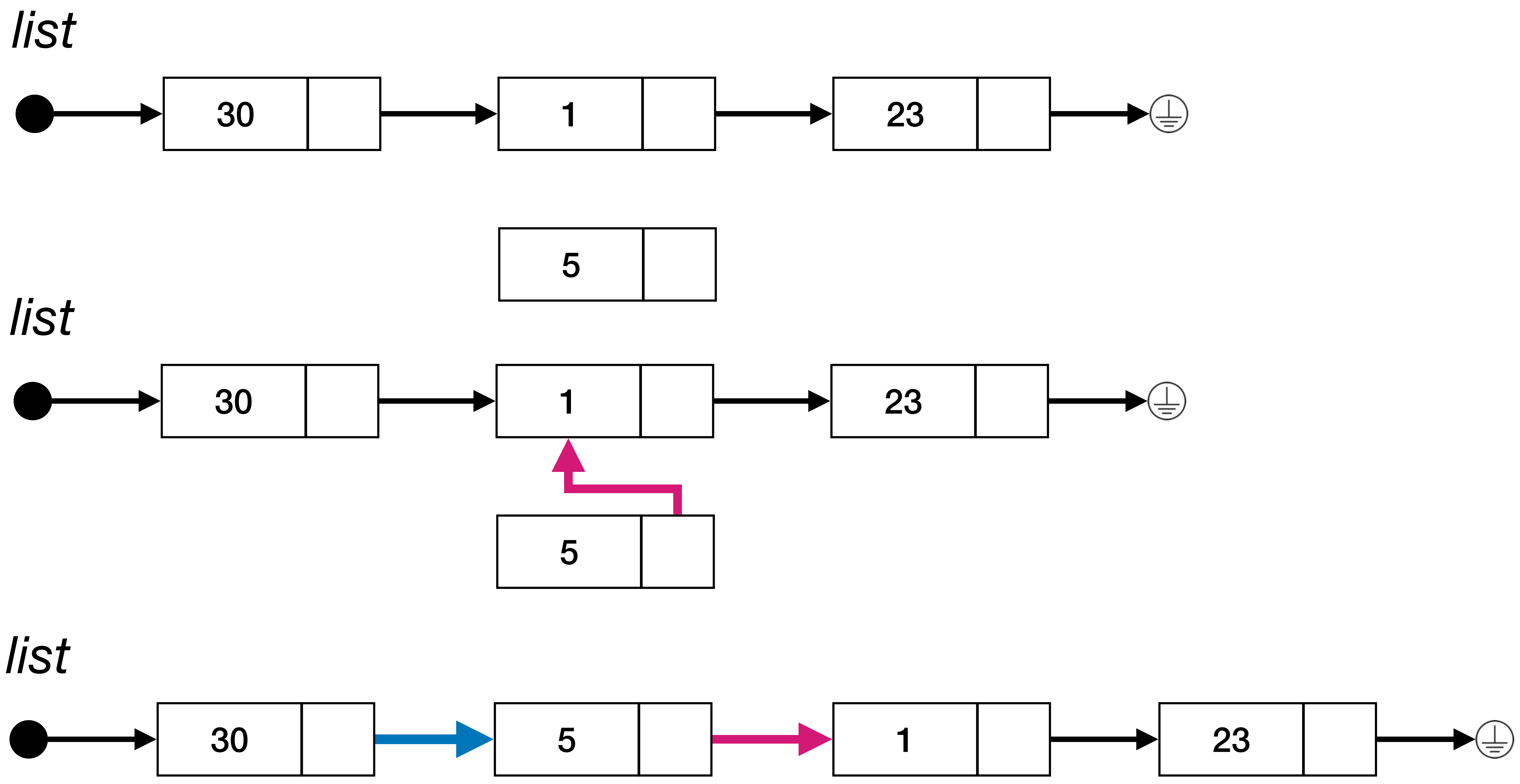
Insert element: add(value=5, position=1)



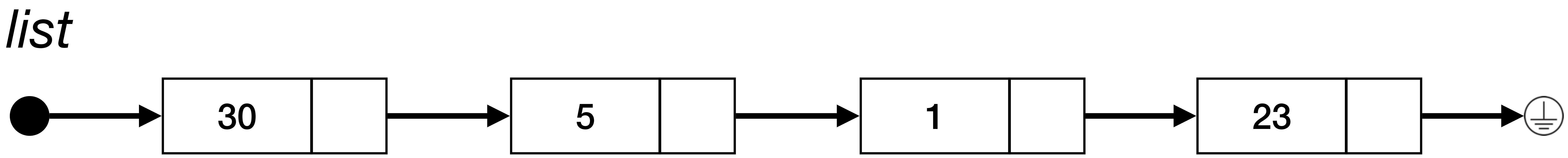
Insert element: add(value=5, position=1)



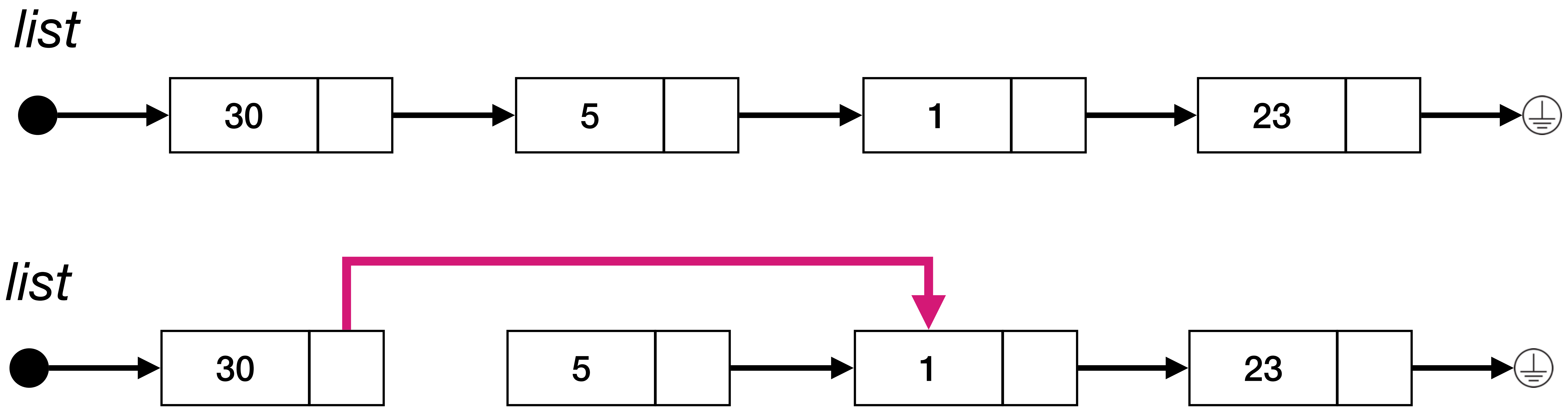
Insert element: add(value=5, position=1)



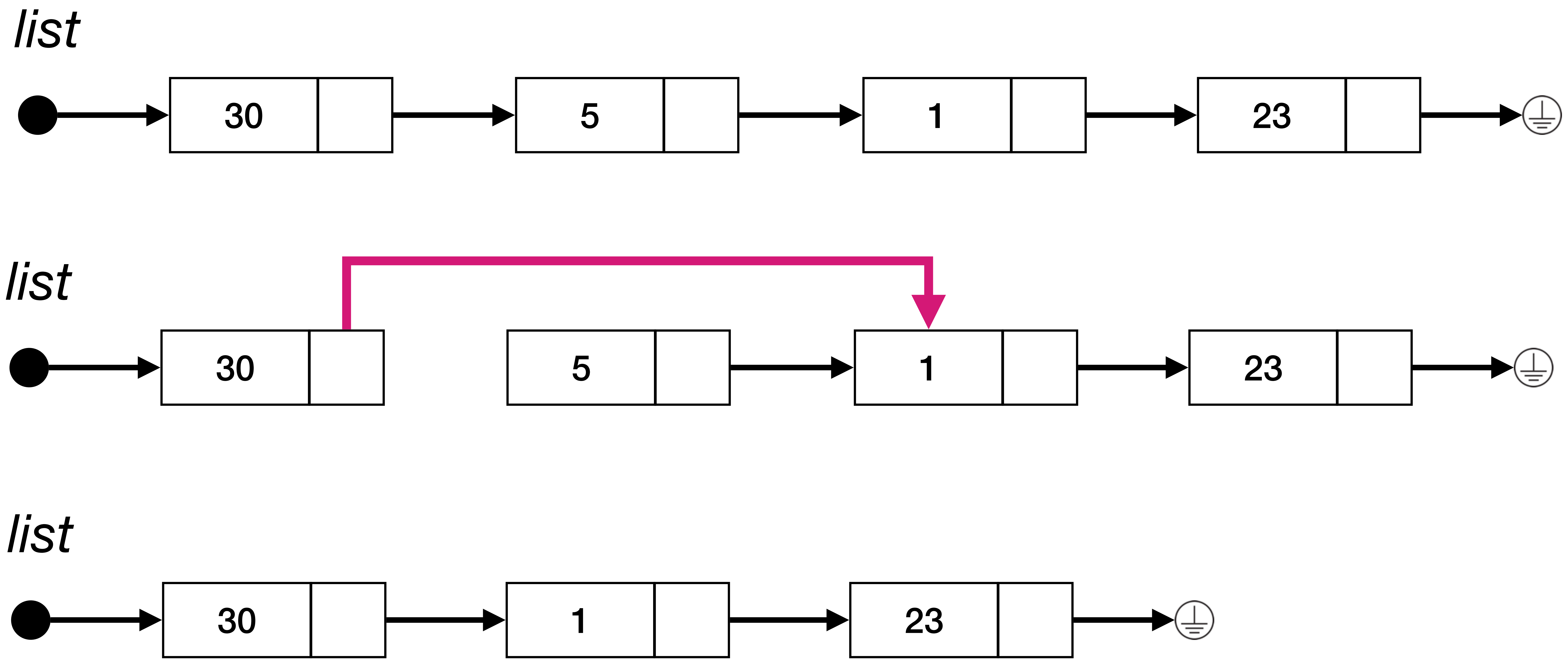
Remove element: remove(position=1)



Remove element: remove(position=1)



Remove element: remove(position=1)



More Linked Lists

- Singly linked lists
- Doubly linked lists
- Circular linked lists
- Circular doubly linked lists

Open Discussion

- What **typical operations** can one commonly do with collections/lists/arrays?

Open Discussion

- What **typical operations** can one commonly do with collections/lists/arrays?

- Adding and removing elements
- Checking for emptiness

Open Discussion

- What **typical operations** can one commonly do with collections/lists/arrays?

- Adding and removing elements
- Checking for emptiness

Sorting

Searching

Sorting

- Process of rearranging a sequence of objects so that there is a **logical ordering** on one (or more) of the fields in the items
- Sort **key**: The field (or fields) on which the logical ordering is based
- Sorting **algorithm**: The algorithm that orders the items based on the sort key

Input: A **sequence** of **n** items $\mathbf{a_1, a_2, a_3, \ldots, a_n}$

Output: A **permutation** of the input sequence $\mathbf{a'_1, a'_2, a'_3, \ldots, a'_n}$, such that $\mathbf{a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n}$

Open Discussion

- Why is sorting important?

- Although it is not the end-game/final goal, sorting plays a major role in commercial data processing and in modern scientific computing as enabler for efficient algorithms

- Examples:
    - **Searching**: some search algorithms require the data to be sorted (i.e., binary search)
    - **Graphics**: rendering need objects organized in layers sorted by distance (i.e., z-buffer)

(Some) Sorting Algorithms

Based on direct comparison of the item/keys

- **Selection sort**
- **Insertion sort**
- Shell sort
- **Bubble sort**
- **Quick sort**
- **Merge sort**
- **Heap sort**

Not based only on comparison

- **Counting sort**
- **Radix sort**
- Bucket/Bin sort

Which algorithm is faster?

- Since **actual time** varies depending on memory, processor speed, etc., the number of **comparisons** or **swaps** is a good measure, i.e., a proxy, for comparing algorithms in a way that is independent of the platform that runs them
- Example:
    - How many comparisons did selection sort do in the worst case?
    - What about bubble sort?

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort**
- **Insertion sort**
- Shell sort
- **Bubble sort**
- **Quick sort**
- **Merge sort**
- **Heap sort**

Not based only on comparison

- **Counting sort**
- **Radix sort**
- Bucket/Bin sort

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort** $O(n^2)$
- **Insertion sort**
- Shell sort
- **Bubble sort**
- **Quick sort**
- **Merge sort**
- **Heap sort**

Not based only on comparison

- **Counting sort**
- **Radix sort**
- Bucket/Bin sort

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort** $O(n^2)$
- **Insertion sort** $O(n^2)$
- Shell sort
- **Bubble sort**
- **Quick sort**
- **Merge sort**
- **Heap sort**

Not based only on comparison

- **Counting sort**
- **Radix sort**
- Bucket/Bin sort

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort** $O(n^2)$
- **Insertion sort** $O(n^2)$
- Shell sort
- **Bubble sort** $O(n^2)$
- **Quick sort**
- **Merge sort**
- **Heap sort**

Not based only on comparison

- **Counting sort**
- **Radix sort**
- Bucket/Bin sort

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort** $O(n^2)$
- **Insertion sort** $O(n^2)$
- Shell sort
- **Bubble sort** $O(n^2)$
- **Quick sort** $O(n^2)$
- **Merge sort**
- **Heap sort**

Not based only on comparison

- **Counting sort**
- **Radix sort**
- Bucket/Bin sort

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort** $O(n^2)$
- **Insertion sort** $O(n^2)$
- Shell sort
- **Bubble sort** $O(n^2)$
- **Quick sort** $O(n^2)$
- **Merge sort** $O(n \log(n))$
- **Heap sort**

Not based only on comparison

- **Counting sort**
- **Radix sort**
- Bucket/Bin sort

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort** $O(n^2)$
- **Insertion sort** $O(n^2)$
- Shell sort
- **Bubble sort** $O(n^2)$
- **Quick sort** $O(n^2)$
- **Merge sort** $O(n \log(n))$
- **Heap sort** $O(n \log(n))$

Not based only on comparison

- **Counting sort**
- **Radix sort**
- Bucket/Bin sort

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort** $O(n^2)$
- **Insertion sort** $O(n^2)$
- Shell sort
- **Bubble sort** $O(n^2)$
- **Quick sort** $O(n^2)$
- **Merge sort** $O(n \log(n))$
- **Heap sort** $O(n \log(n))$

Not based only on comparison

- **Counting sort** $O(k+n)$
- **Radix sort**
- Bucket/Bin sort

(Some) Sorting Algorithms

Worst case complexity

Based on direct comparison of the item/keys

- **Selection sort** $O(n^2)$
- **Insertion sort** $O(n^2)$
- Shell sort
- **Bubble sort** $O(n^2)$
- **Quick sort** $O(n^2)$
- **Merge sort** $O(n \log(n))$
- **Heap sort** $O(n \log(n))$

Not based only on comparison

- **Counting sort** $O(k+n)$
- **Radix sort** $O(kn)$
- Bucket/Bin sort

Searching

- Linear search
  - Brute force
  - Starts at the beginning and scan the entire data structure until either it finds the element or there are no more elements
  - Complexity O(n)
- Binary search
  - Divide and conquer
  - Smarter than linear search, but requires the elements to be sorted
  - Starts in the middle, does a comparison and repeats in the first or second half of the structure
  - Complexity $O(\log_2(n))$

Other Linear Data Structures

- Linear structures can be represented as lines
- Can be implemented on top of arrays (max size) or linked lists
- Queues:
  - Ensure FIFO
- Stacks
  - Ensure LIFO
- Stequeues:
  - Combines the two above

Other Data Structures

- Hash maps:
  - Direct access tables with an hash functions. Can deal with collisions by using a linked list
- Make-Set/Disjoint Sets
  - A set of sets that can be merged
- (Min/Max) Heap:
  - An array that pretends to be a tree.

Exercise and Homework

- Each session ends with an exercise that becomes your homework for the week
- The exercises are (will be made) available in the public repo:
  - https://github.com/IMC-UAS-Krems/ADSII3ILV-WS23-24-Homework
- However, for the homework, we follow the same approach of assignments
  - Checkout the assignment from Github Classroom
  - Commit the solution and the tests on your Github repo
  - The lecturer checks your progress directly on Github (so commit and push frequently)

ADSII3ILV WS23/24 Homework

https://classroom.github.com/a/e2gGNSqD