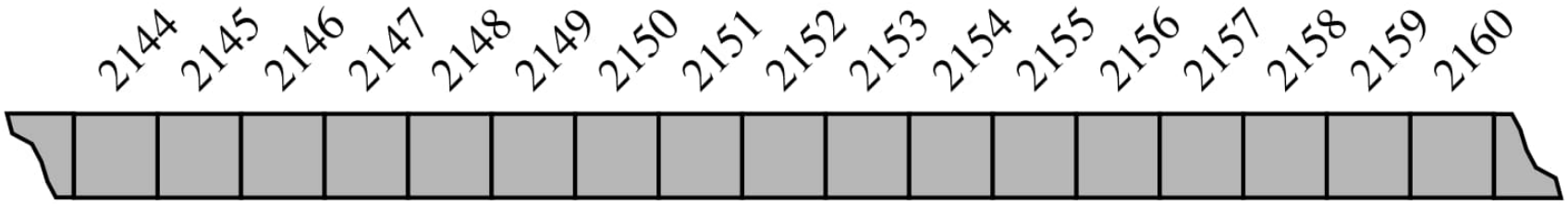
Algorithms and Complexity

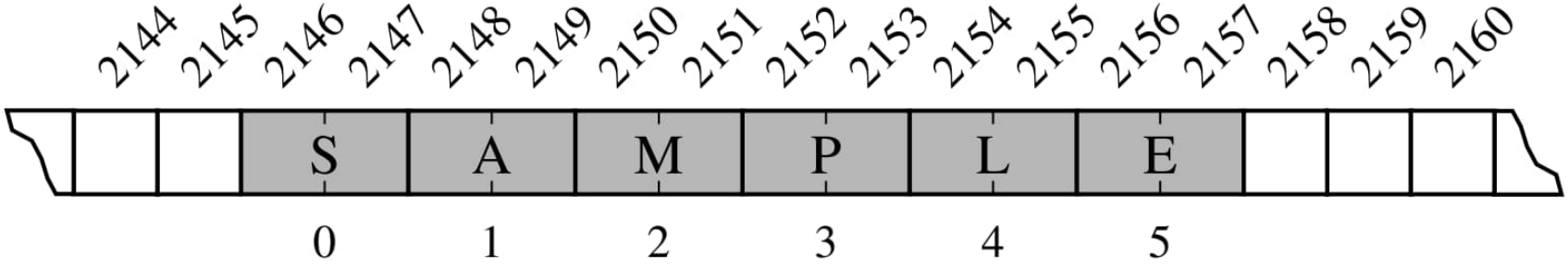
Module 2

Arrays

- To keep track of what information is stored in what memory byte, the computer uses an abstraction known as a memory address



- A group of related variables can be stored one after another in a contiguous portion of the computer's memory
- Such a representation is denoted as an array



Arrays

- The most commonly used data structure
- Straightforward to understand and match closely the underlying computer hardware
- Almost all CPUs make it very fast to access data at known offsets from a base address
- Almost every programming language supports arrays as part of the core data structures
- Many of the more complex data structures are built using arrays

Elementary Operations on Arrays

- Insertion of a new item into the array
  - Takes constant time - $O(1)$
- Searching for a particular item in the array
  - Time is proportional to the array size n – $O(n)$
- Deletion of a particular item from the array
  - Since holes are not allowed in the array, all items on the right of the deleted item must be moved one position to the left
  - Thus, deletion time is also proportional to the array size n – $O(n)$

Binary Search

- Searching cost can be significantly reduced to $O(\log n)$ if the array is sorted
- Let's denote as `low` and `high` the lowest and highest index in the currently searched part of the array, respectively
- Initially `low` = 0 and `high` = n -1
- Then, we compare the target value to the median candidate `data[mid]` with index:

```
mid = (low + high) // 2
```

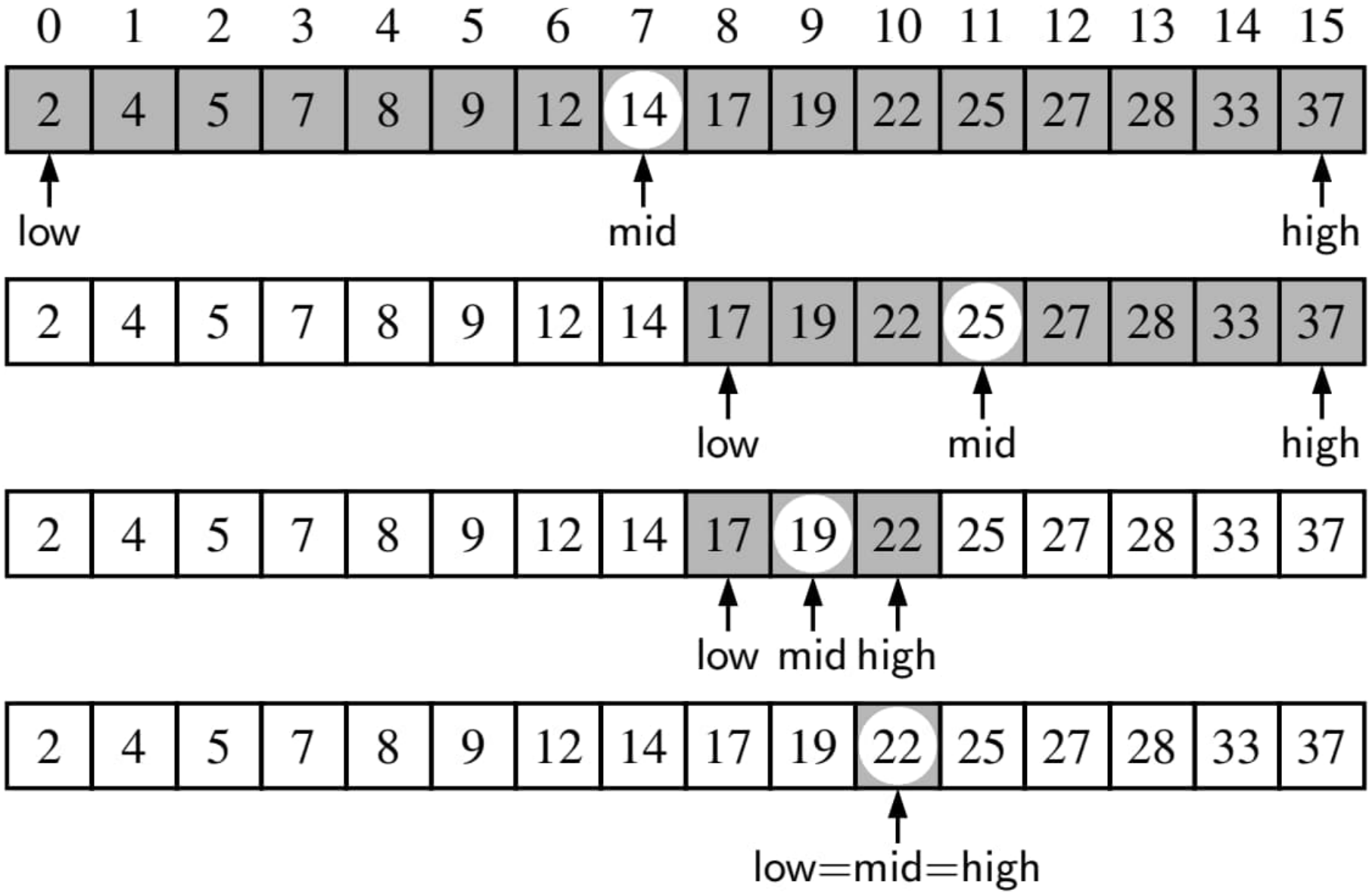
- With each unsuccessful comparison, the search range is halved

Binary Search

- Three cases are possible:
  - `target == data[mid]` and the search terminates successfully
  - `target < data[mid]`, then we search the left half of the sequence, that is, on the interval of indices from `low` to `mid-1`
  - `target > data[mid]`, then we search the right half of the sequence, that is, on the interval of indices from `mid+1` to `high`
- An unsuccessful search occurs if `low > high`, as the interval `[low, high]` is empty

Binary Search

- Example of a binary search for target value 22:



Binary Search

- Iterative implementation in Python:

```python
def binarySearch(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return True  # Target found
        elif arr[mid] < target:
            low = mid + 1  # Search in the right half
        else:
            high = mid - 1  # Search in the left half

    return False  # Target not found
```

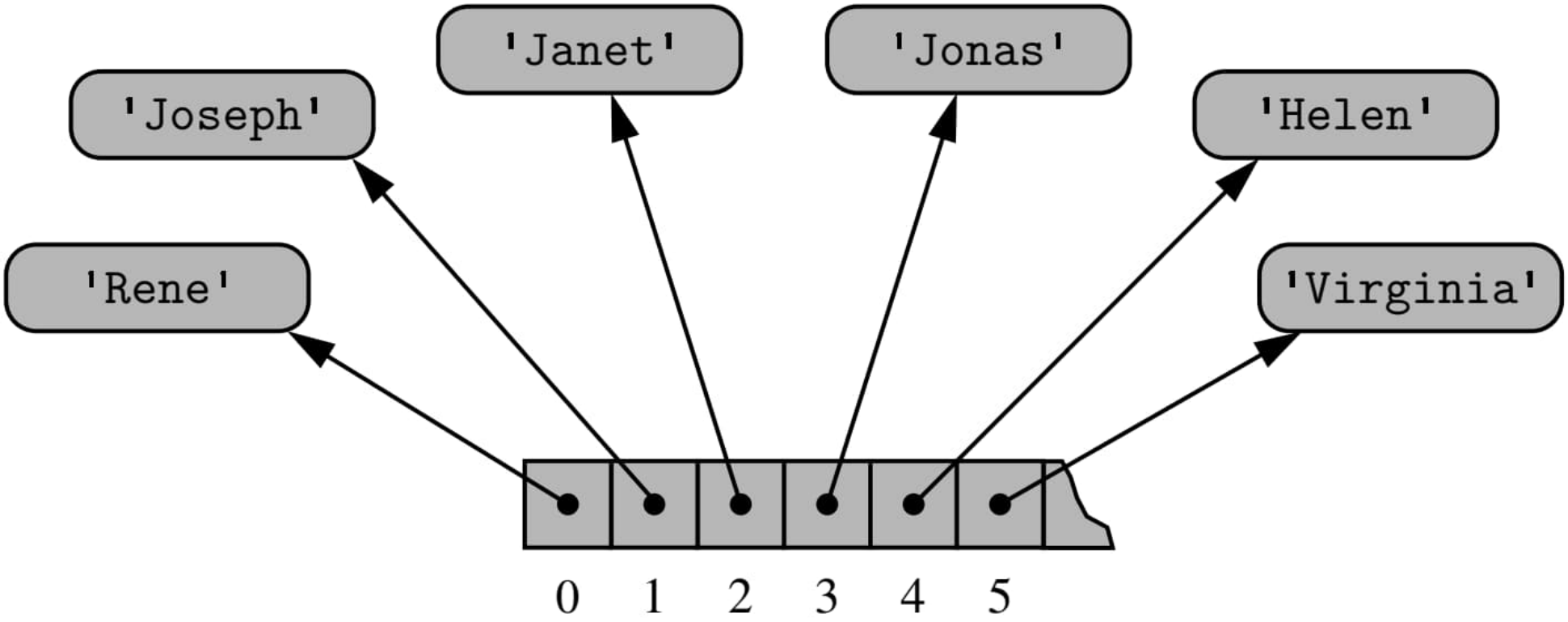
Binary Search

- Recursive implementation in Python:

```python
def binarySearchRec(arr, target, low, high):
    if low > high:
        return False  # Interval is empty; no match
    else:
        mid = (low + high) // 2
        if target == arr[mid]:  # Target found
            return True
        elif arr[mid] < target: # Search in the right half
            return binarySearchRec(arr, target, mid + 1, high)
        else:  # Search in the left half
            return binarySearchRec(arr, target, low, mid - 1)
```

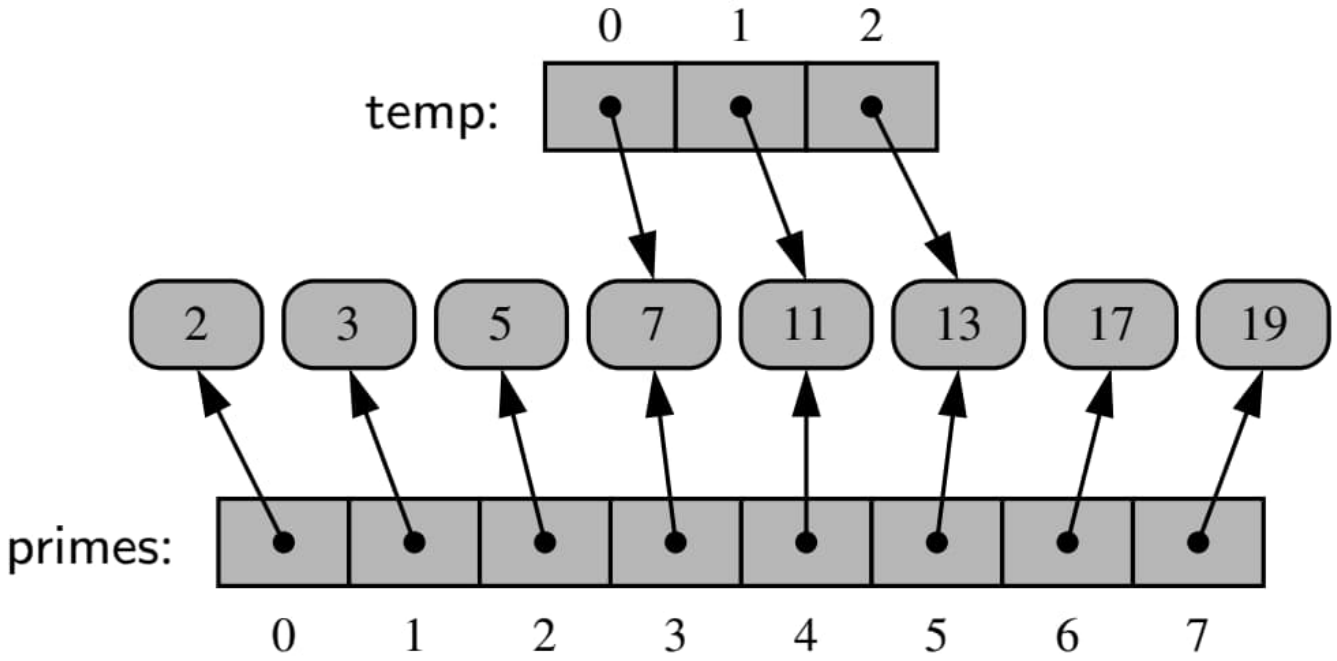
Referential Arrays

- Python's list can store objects of different types and sizes
- Thus, it's impossible to guarantee that each cell of the array use the same number of bytes
- Hence, Python represents a list as an array of object references, i.e., object addresses (pointers)



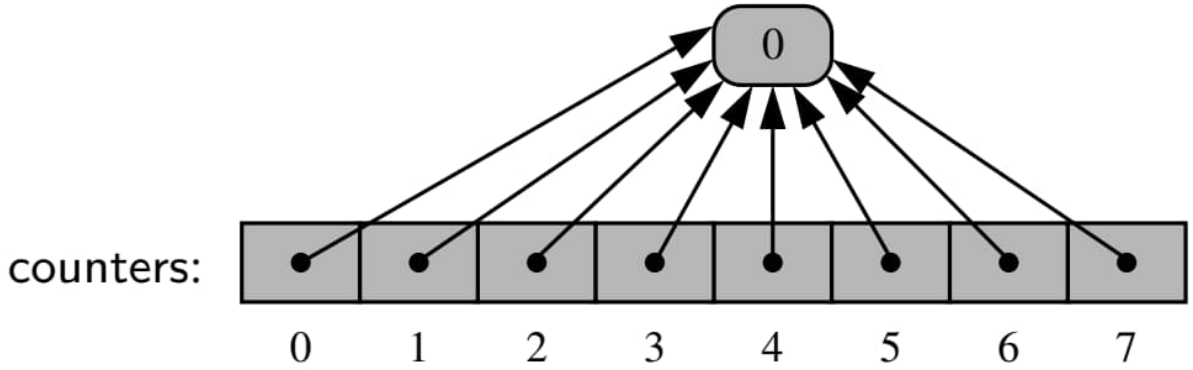
Referential Arrays

- A single list instance may include multiple references to the same object as elements of the list
- It is also possible for a single object to be an element of two or more lists
- For instance, a slice of a list is a new list instance, but that new list has references to the same elements that are in the original list, e.g., `temp = primes[3:6]`

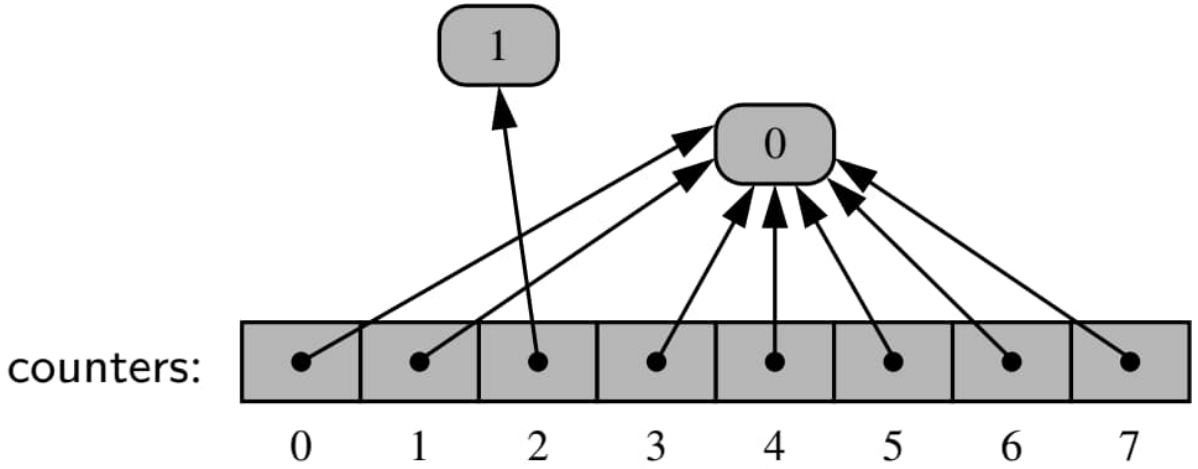


Referential Arrays

- The syntax `counters = [0] * 8` produces a list of length eight, with all eight elements being the value zero
- However, all eight cells of the list reference the same object



- Operation counters[2] += 1 computes a new integer, with value 0+1, and sets cell 2 to reference the newly computed value

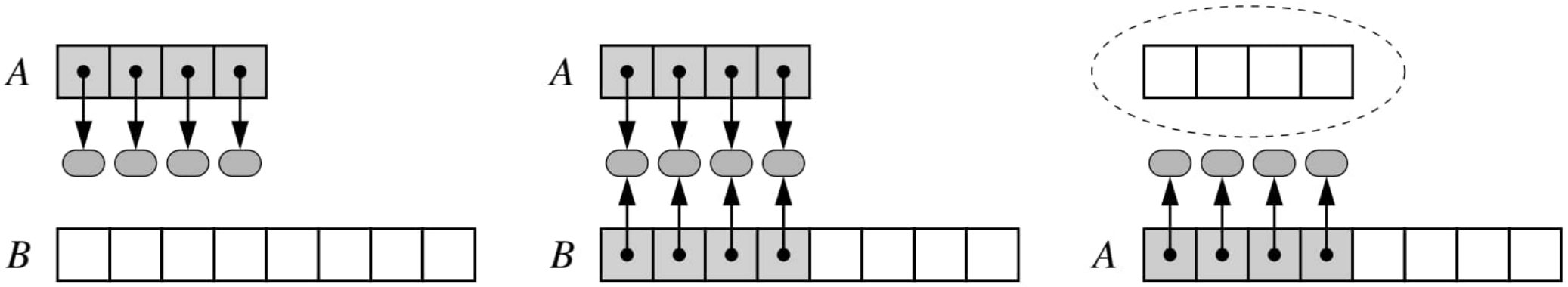


Dynamic Arrays

- The precise size of an array must be explicitly declared at the moment of creation
- On that basis, the system can properly allocate a continuous piece of memory for its storage
- Array neighboring memory locations can store other data, so the capacity of the array cannot be increased simply by expanding into subsequent cells
- Dynamic array can extend its size by requesting a new, larger array from the system, and initializing the new array so that its prefix matches that of the existing smaller array

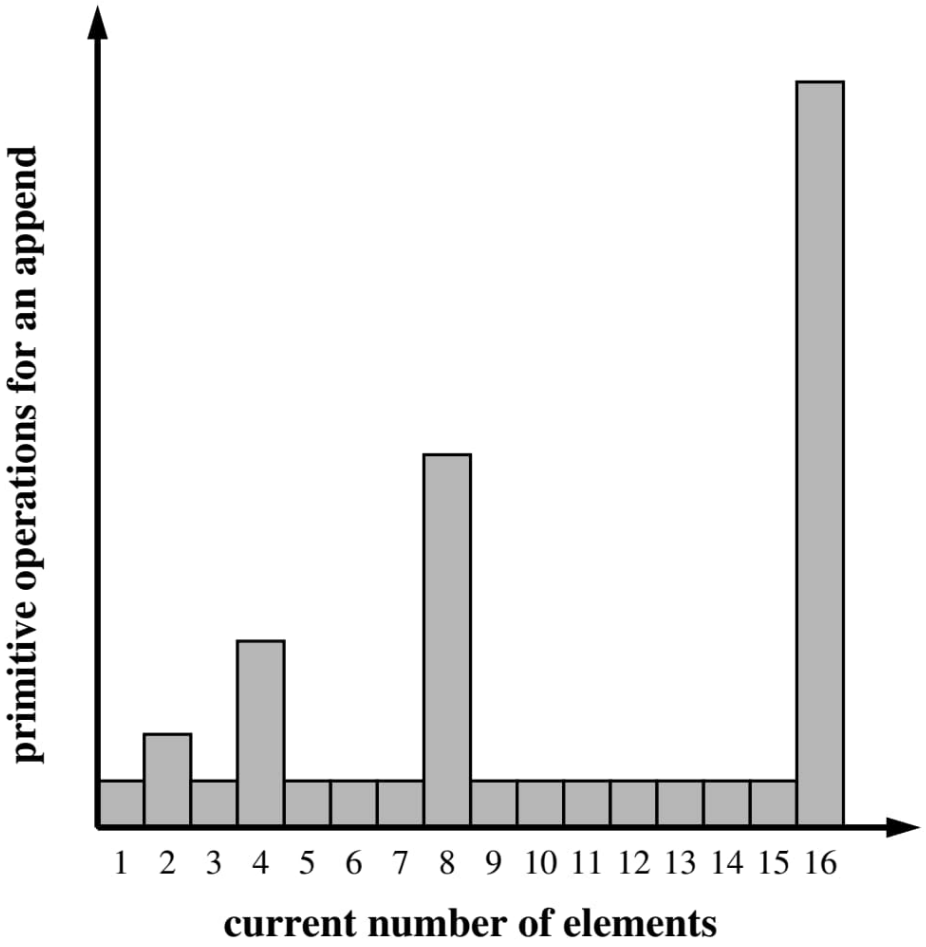
Dynamic Arrays

- Dynamic array A keeps track of its capacity C_A and the number of stored elements n_A
- When the capacity is exhausted ($n_A == C_A$):
  - New array B with $C_B = 2 \times C_A$ is allocated
  - $B[i] = A[i]$, for $i = 0, \ldots, n_A - 1$
  - A becomes B ($A = B$)
  - The new element is inserted to A



Dynamic Arrays

- Doubling the capacity during an array replacement costs *n* copying operations, but allows us to add *n* new elements before the array must be replaced again
- Thus, the amortized running time of each append operation is *O*(1)



Abstract Data Types (ADTs)

- An abstract data type (ADT) is a set of objects together with a set of operations
- Abstract data types are mathematical abstractions
- Nowhere in an ADT's definition is there any mention of how the set of operations is implemented
- Objects such as stacks, lists, and graphs, along with their operations, can be viewed as ADTs
- There is no rule telling which operations must be supported for each ADT; this is a design decision

Stacks

- A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle
- A user may insert (push) objects into a stack at any time
- However, only the most recently inserted object that remains at the top of the stack is accessible
- Two fundamental stack operations involve the pushing and popping of objects on the stack
- Both take constant, i.e., $O(1)$ time

Stacks

- Example 1:
  - Internet Web browsers store the addresses of recently visited sites in a stack
  - Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses
  - The browser then allows the user to "pop" back to previously visited sites using the "back" button
- Example 2:
  - Text editors usually provide an "undo" mechanism that cancels recent editing operations
  - This undo operation can be accomplished by keeping text changes in a stack

Stacks

- A stack is an abstract data type (ADT) such that an instance S supports the following two methods:
  - `S.push(e)`: Add element `e` to the top of stack `S`
  - S.pop(): Remove and return the top element from the stack `S`; return `None` if the stack is empty
- Additionally, `top()`, `is_empty()`, and `len()` methods can be defined for convenience:
  - S.top(): Return a reference to the top element of stack `S`, without removing it; return `None` if the stack is empty
  - S.is_empty(): Return `True` if stack `S` does not contain any elements
  - len(S): Return the number of elements in stack `S`

Stacks

- By convention, we assume that a newly created stack is empty, and that there is no a priori bound on the capacity of the stack
- Example of a series of stack operations and their effects on an initially empty stack S of integers:

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]

Stacks

- Stack can be easily implemented on top of an array (list) data structure
- The Python list class already supports adding an element to the end with the `append()` method, and removing the last element with the `pop()` method
- Realization of a stack S as an adaptation of a Python list L:

Stack Method	***Realization with Python list***
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[−1]
S.is_empty()	len(L) == 0
len(S)	len(L)

Stacks

- The (amortized) running times for the list-based stack methods:

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop( )	$O(1)^*$
S.top( )	$O(1)$
S.is_empty( )	$O(1)$
len(S)	$O(1)$

*amortized

- A typical call to `push( )` and `pop( )` methods uses constant time, but there is occasionally an $O(n)$-time worst case, when an operation causes the list to resize its internal array

Matching Parentheses

- Let's consider arithmetic expressions that may contain various pairs of grouping symbols:
  - Parentheses: "(" and ")"
  - Braces: "{" and "}"
  - Brackets: "[" and "]"
- Each opening symbol must match its corresponding closing symbol
- Examples:
  - Correct: `()(()){([()])}`
  - Correct: `((()(()){([()])}))`
  - Incorrect: `)(()){([()])}`
  - Incorrect: `({[])}`
  - Incorrect: `(`

Matching Parentheses

- The simple algorithm which uses a stack:
  - Make an empty stack and read characters from left to right.
  - If the character is an opening symbol, push it onto the stack.
  - If it is a closing symbol and the stack is empty, report an error. Otherwise, pop the stack.
  - If the symbol popped is not the corresponding opening symbol, then report an error.
  - If you reach the end of the expression and the stack is empty, then the original expression was properly matched. Otherwise, report an error.

Matching Parentheses

- Algorithm complexity:
  - If the length of the expression is n, the algorithm will make at most n calls to push and n calls to pop
  - Those calls run in a total of $O(n)$
  - Auxiliary tests such as if c is a left or right delimiter each run in $O(1)$
  - Combining these operations, the matching algorithm on a sequence of length n runs in $O(n)$ time

Evaluation of Postfix Expressions

- Everyday arithmetic expressions are written with an operator (+, –, ×, or /) placed between two operands, e.g., 3 + 4 × 5
- This is called infix notation
- In postfix notation (Reverse Polish Notation, or RPN), the operator follows the two operands, e.g., 3 4 5 × +
- RPN removes the need for order of operations and parentheses that are required by infix notation and can be evaluated linearly, left-to-right
- For example, the infix expression (3 + 4) × (5 + 6) becomes 3 4 + 5 6 + ×

Evaluation of Postfix Expressions

- Postfix evaluation algorithm with a stack:
  - Make an empty stack and read characters from left to right.
  - If the character is an operand, push it onto the stack.
  - If the character is an operator symbol, pop the top two operands from the stack and apply the operator to them. Push the result.
  - When you reach the end of the expression, pop the stack to obtain the answer.
- The time to evaluate a postfix expression is $O(n)$, because processing each element in the input consists of stack operations and therefore takes constant time

Evaluation of Postfix Expressions

- For example, the postfix expression 6 5 2 3 + 8 ∗ +3 + ∗ is evaluated as follows:
  - The first four symbols are placed on the stack. The resulting stack is [6, 5, 2, 3]
  - Next, a '+' is read, so 3 and 2 are popped from the stack, and their sum, 5, is pushed: [6, 5, 5]
  - Next, 8 is pushed: [6, 5, 5, 8]
  - Next, a '∗' is read, so 8 and 5 are popped, and 5 ∗ 8 = 40 is pushed: [6, 5, 40]
  - Next, a '+' is read, so 40 and 5 are popped, and 5 + 40 = 45 is pushed: [6, 45]
  - Next, 3 is pushed: [6, 45, 3]
  - Next, '+' pops 3 and 45 and pushes 45 + 3 = 48: [6, 48]
  - Finally, a '∗' is read and 48 and 6 are popped; the result, 6 ∗ 48 = 288, is pushed: [288]

Queues

- Queue is a collection of objects that are inserted and removed according to the first-in, first-out (FIFO) principle
- Elements enter a queue at the back and are removed from the front
- Queues are used extensively in computer systems: the jobs waiting to run, the messages to be passed over a network, the sequence of characters waiting to be printed on a terminal
- They’re used to model real-world situations such as people waiting in line for tickets or airplanes waiting to take off

Queues

- The queue abstract data type (ADT) supports the following two fundamental methods for a queue Q:
  - Q.enqueue(e): Add element e to the back of queue Q
  - Q.dequeue(): Remove and return the first element from queue Q; return None if the queue is empty
- The queue ADT also includes the following supporting methods:
  - Q.first(): Return a reference to the element at the front of queue Q, without removing it; return None if the queue is empty
  - Q.is_empty(): Return True if queue Q does not contain any elements
  - len(Q): Return the number of elements in queue Q

Queues

- By convention, we assume that a newly created queue is empty, and that there is no a priori bound on the capacity of the queue
- Example of a series of queue operations and their effects on an initially empty queue Q of integers:

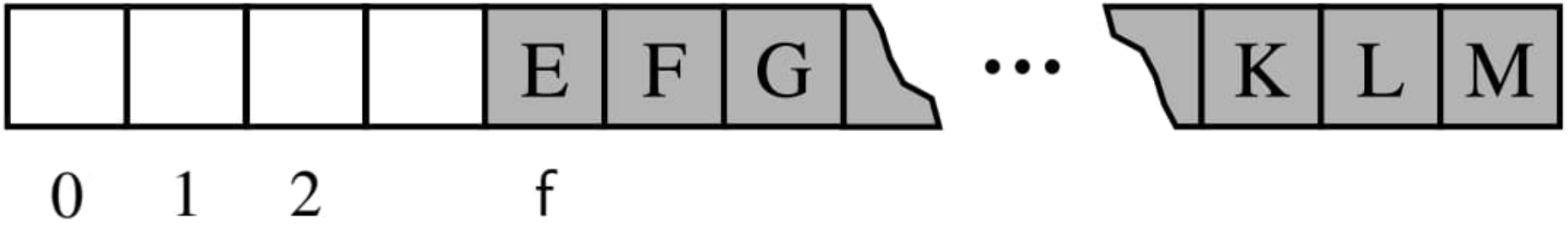
Operation	**Return Value**	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]

Queues

- To implement a queue on the top of a Python list, we could use `append(e)` to enqueue element `e` and `pop(0)` to remove the first element from the list when dequeuing
- However, when `pop` is called on a list with a non-default index, a loop is executed to shift all elements beyond the specified index to the left
- Therefore, a call to `pop(0)` always causes the worst-case behavior of $O(n)$ time

Queues

- To avoid the call to `pop(0)`, we can replace the dequeued entry in the array with a reference to `None`
- An explicit variable *f* can store the index of the element that is currently at the front of the queue
- Such an algorithm for dequeue would run in *O*(1) time
- However, over time, the size of the underlying list would grow to *O*(*m*), where *m* is the total number of enqueue operations since the creation of the queue, rather than the current number of elements in the queue



Queues

- In a more robust queue implementation, the front of the queue can drift rightward, while we allow the contents of the queue to “wrap around” the end of an underlying array
- The underlying array has fixed length *N* that is greater that the actual number of elements in the queue (*n*)
- When an element is enqueued, its location is computed as (*f* + *n*) % *N*
- When an element is dequeued, the new value of the front index *f* = (*f* + 1) % *N*

Queues

- The (amortized) running times for the list-based queue methods:

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

- The bounds for enqueue and dequeue are amortized due to the resizing of the array
- The space usage is $O(n)$, where n is the current number of elements in the queue

Double-Ended Queues (Deques)

- Deque (pronounced “deck”) supports insertion and deletion at both the front and the back of the queue
- The deque ADT is defined so that deque D supports the following methods:
  - D.add_first(e): Add element e to the front of deque D
  - D.add_last(e): Add element e to the back of deque D
  - D.delete_first(): Remove and return the first element from deque D; return None if the deque is empty
  - D.delete_last(): Remove and return the last element from deque D; return None if the deque is empty

Double-Ended Queues (Deques)

- The deque also includes the following supporting methods:
  - D.first(): Return (but do not remove) the first element of deque D; return None if the deque is empty
  - D.last(): Return (but do not remove) the last element of deque D; return None if the deque is empty
  - D.is_empty(): Return True if deque D does not contain any elements
  - len(D): Return the number of elements in deque D
- The deque can be implemented the same way as the queue using the circular array

Double-Ended Queues (Deques)

- Example of a series of operations and their effects on an initially empty deque D of integers:

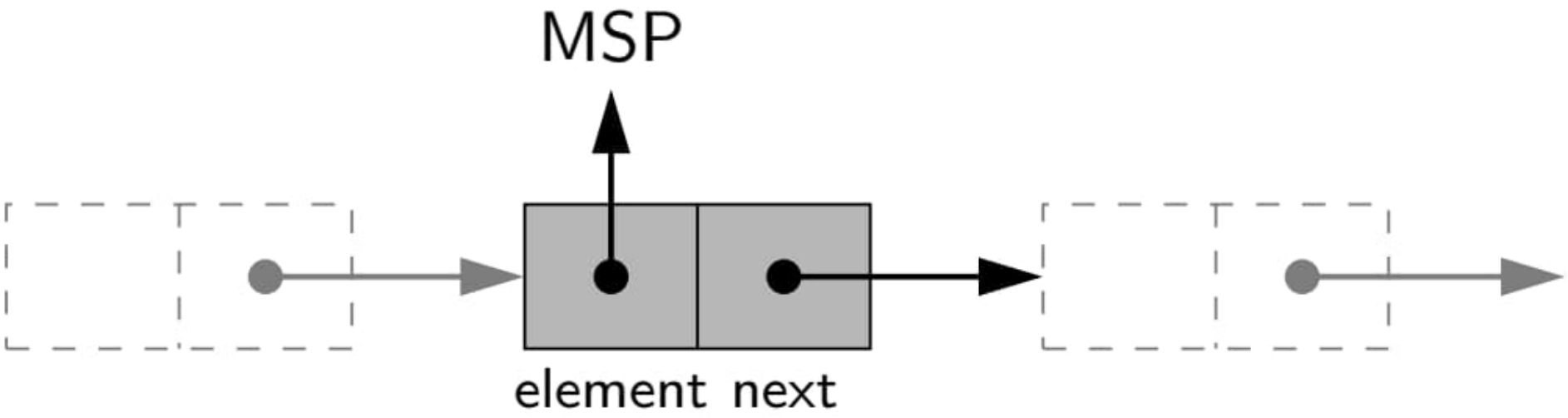
Operation	Return Value	Deque
D.add_last(5)	–	[5]
D.add_first(3)	–	[3, 5]
D.add_first(7)	–	[7, 3, 5]
D.first( )	7	[7, 3, 5]
D.delete_last( )	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last( )	3	[7]
D.delete_last( )	7	[ ]
D.add_first(6)	–	[6]
D.last( )	6	[6]
D.add_first(8)	–	[8, 6]
D.is_empty( )	False	[8, 6]
D.last( )	6	[8, 6]

Linked Lists

- The length of a dynamic array might be longer than the actual number of elements that it stores
- Amortized bounds for operations may be unacceptable in real-time systems
- Insertions and deletions at interior positions of an array are expensive
- Linked list relies on a more distributed representation in which a lightweight object, known as a node, is allocated for each element
- Each node maintains a reference to its element and one or more references to neighboring nodes to collectively represent the linear order of the sequence

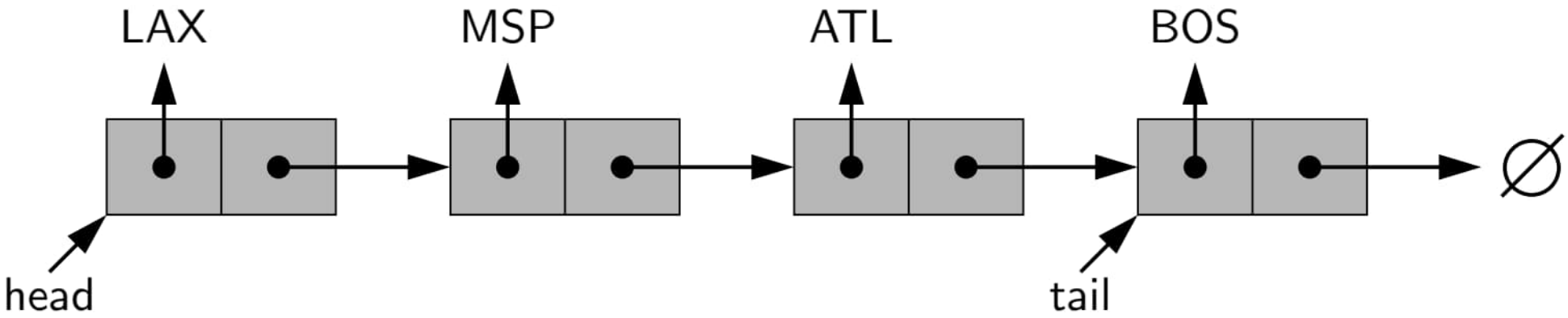
Singly Linked Lists

- A singly linked list is a collection of nodes that collectively form a linear sequence
- Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list



Singly Linked Lists

- The first and last node of a linked list are known as the head and tail of the list, respectively
- By starting at the head, and moving from one node to another by following each node's next reference, we can reach the tail of the list
- This process is known as traversing the linked list

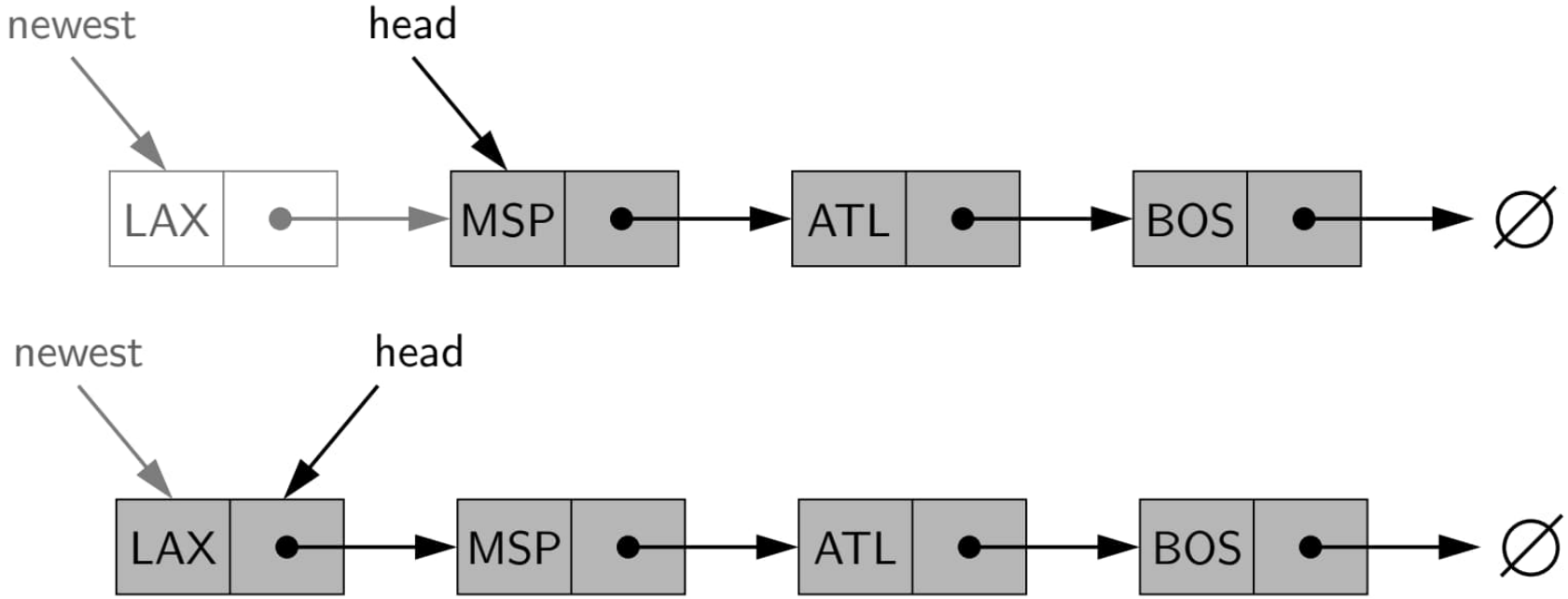


Singly Linked Lists

- We can identify the tail as the node having None as its next reference
- Minimally, the linked list instance must keep a reference to the head of the list
- There is not an absolute need to store a direct reference to the tail of the list, however, it is a common convenience to avoid the traversal
- It is also common for the linked list instance to keep a count of the total number of nodes that comprise the list, to avoid the need to traverse the list to count the nodes

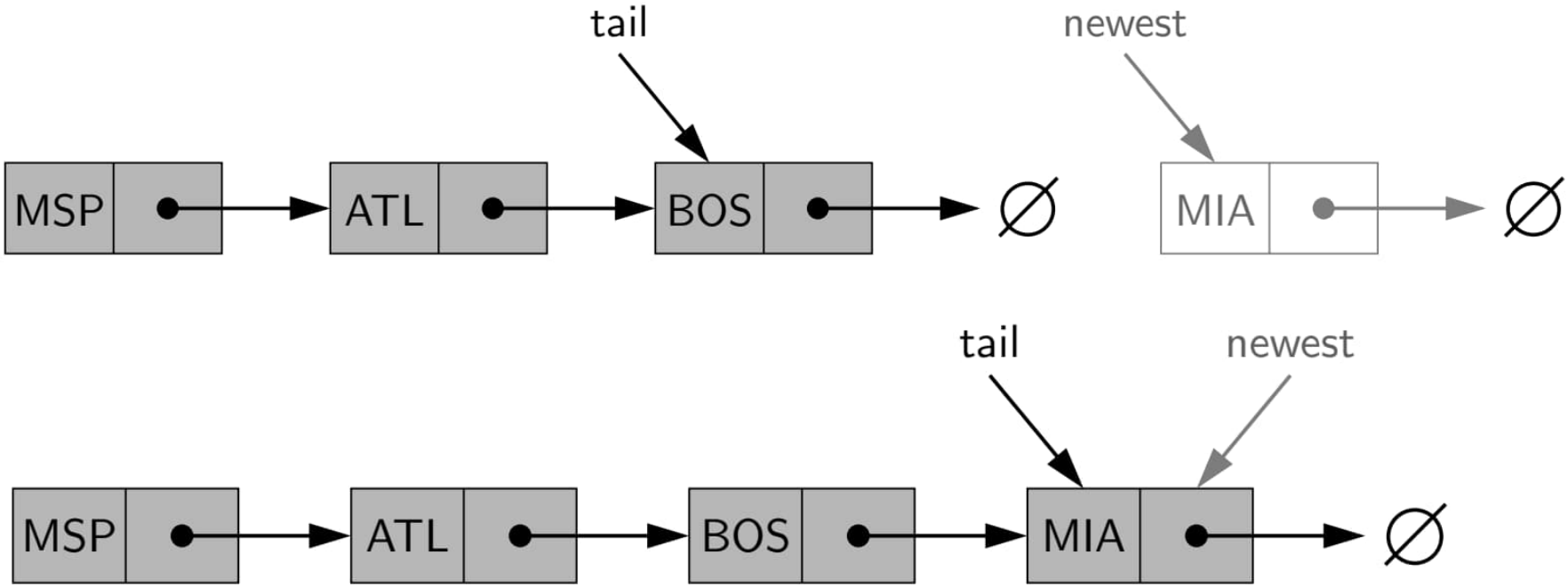
Singly Linked Lists

- Linked list does not have a predetermined fixed size; it uses space proportionally to its current number of elements
- A new element can be easily inserted at the head of the list
- We create a new node, set its element to the new element, set its next link to refer to the current head, and then set the list's head to point to the new node



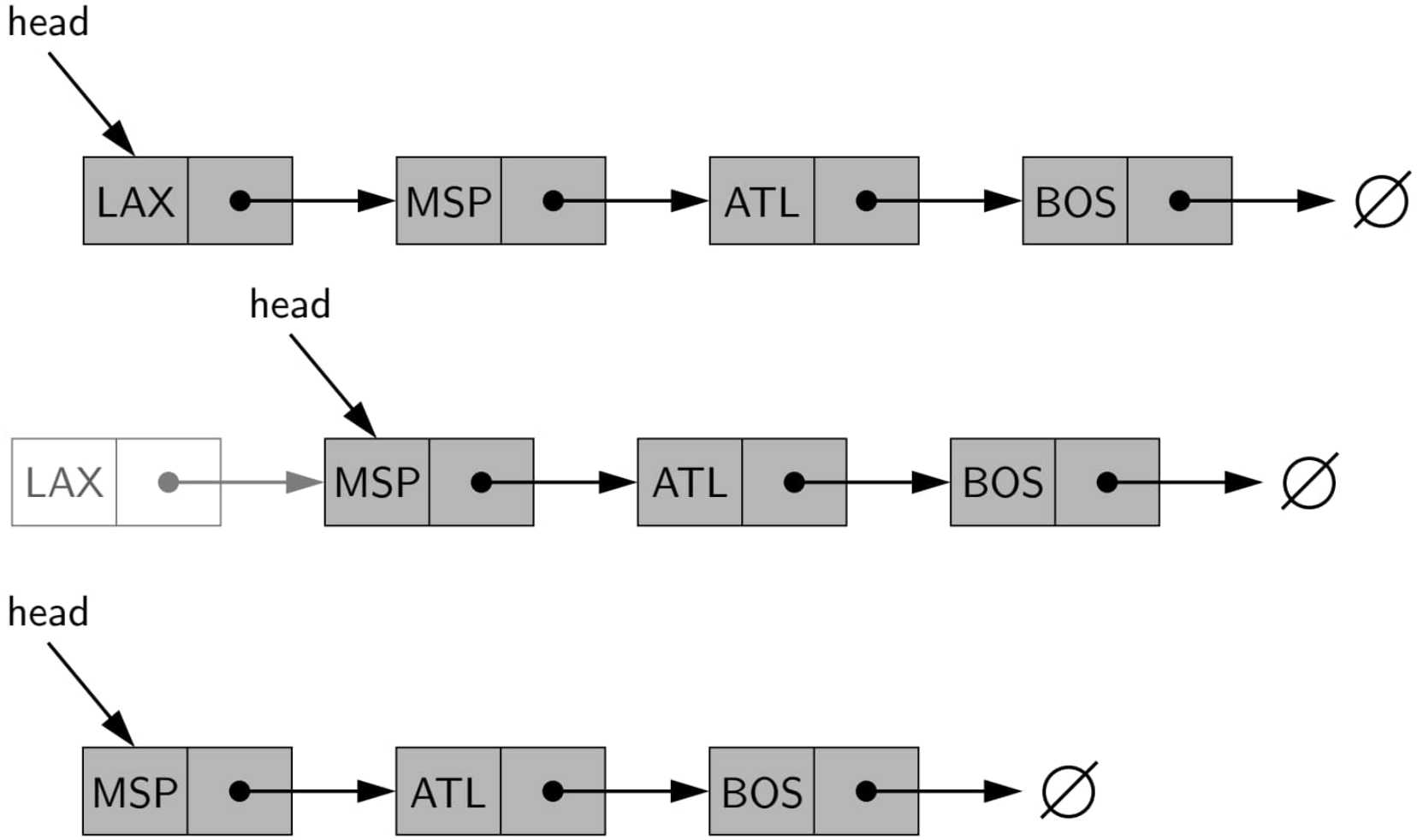
Singly Linked Lists

- A new element can also be easily inserted at the tail of the list, provided we keep a reference to the tail node
- We create a new node, assign its next reference to None, set the next reference of the tail to point to this new node, and then update the tail reference itself to this new node



Singly Linked Lists

- Removing an element from the head of a singly linked list is essentially the reverse operation of inserting a new element at the head



Singly Linked Lists

- We cannot easily delete the last node of a singly linked list since we cannot reach the node before the tail by following next links from the tail
- The only way to access this node is to start from the head of the list and search all the way through the list
- Such a sequence of link-hopping operations could take a long time
- To support such an operation efficiently, we need a doubly linked list

Singly Linked Lists

- Singly linked list can be used to model a stack
- Since all stack operations affect the top, we orient the top of the stack at the head of the list
- The implementation of push essentially mirrors insertion at the head of the list
- The implementation of pop essentially mirrors removal at the head of the list
- All methods of the list-based stack implementation complete in worst-case constant time:

Operation	Running Time
S.push(e)	$O(1)$
S.pop( )	$O(1)$
S.top( )	$O(1)$
len(S)	$O(1)$
S.is_empty( )	$O(1)$

Singly Linked Lists

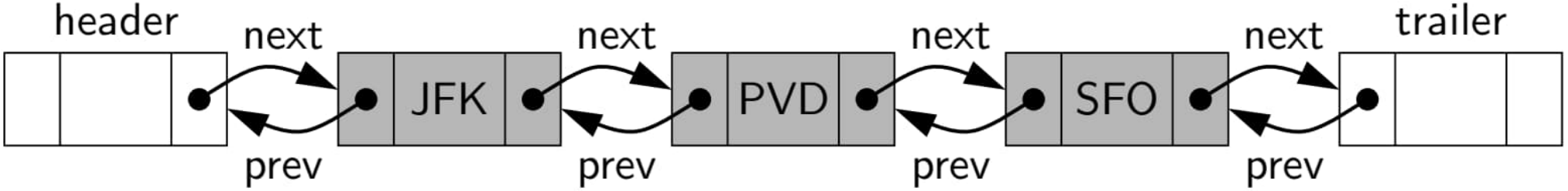
- Singly linked list can be also used to implement the queue ADT supporting worst-case $O(1)$-time for all operations
- Because we need to perform operations on both ends of the queue, we need to keep the reference both to the head and tail
- We must be able to enqueue elements at the back, and dequeue them from the front
- Thus, the natural orientation for a queue is to align the front of the queue with the head of the list, and the back of the queue with the tail of the list

Doubly Linked Lists

- Singly linked lists have limitations that stem from their asymmetry - each node maintains a reference to the node that is immediately after it
- We can efficiently insert a node at either end of a singly linked list, but efficiently delete a node at the head of the list only
- In general, we cannot efficiently delete an arbitrary node from an interior position of the list if only given a reference to that node
- In a doubly linked list, each node keeps an explicit reference to the node before it (`prev`) and a reference to the node after it (`next`)

Doubly Linked Lists

- These lists allow a greater variety of *O*(1)-time update operations, including insertions and deletions at arbitrary positions within the list
- In order to avoid some special cases when operating near the boundaries of a doubly linked list, two dummy nodes are added at both ends of the list : a header node at the beginning of the list, and a trailer node at the end of the list
- These nodes are known as sentinels (or guards), and they do not store elements of the primary sequence

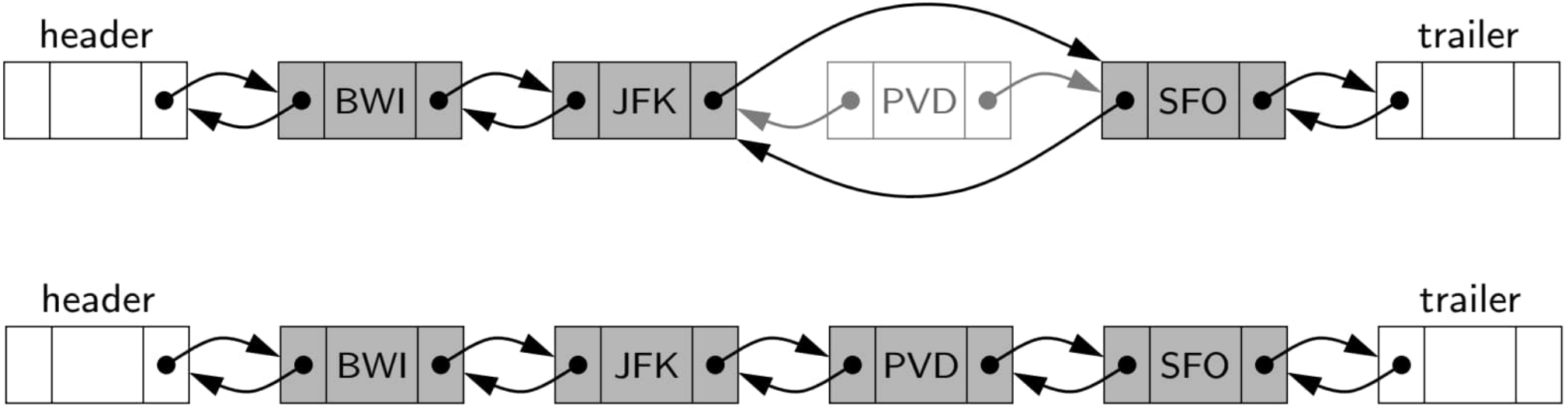


Doubly Linked Lists

- An empty list is initialized so that the `next` field of the header points to the trailer, and the `prev` field of the trailer points to the header
- The header and trailer nodes never change - only the nodes between them change
- Thus, all insertions can be treated in a unified manner, because a new node will always be placed between a pair of existing nodes
- Similarly, every element that is to be deleted is guaranteed to be stored in a node that has neighbors on each side

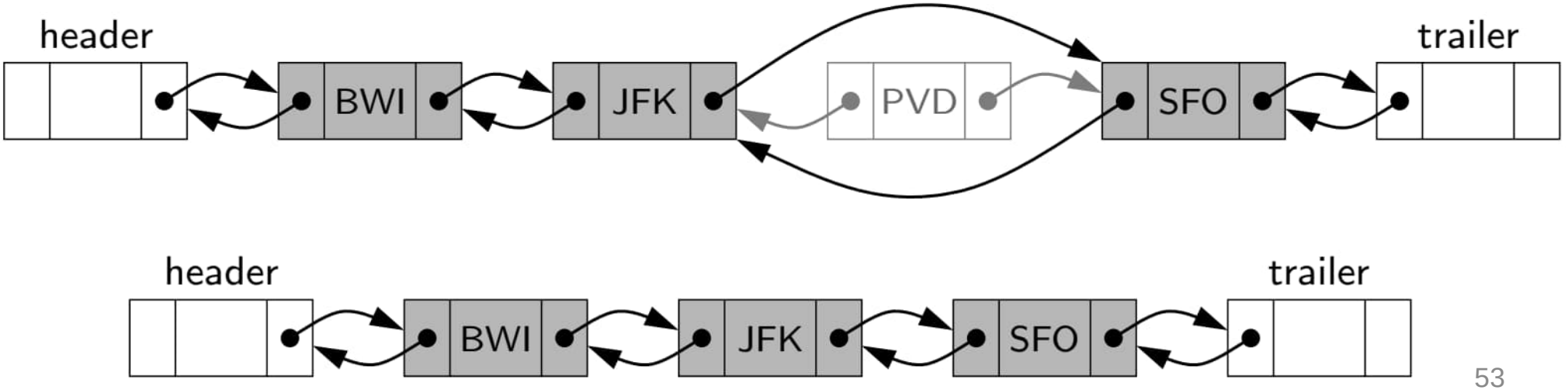
Doubly Linked Lists

- Every insertion into a doubly linked list takes place between a pair of existing nodes



Doubly Linked Lists

- The deletion of a node proceeds in the opposite fashion of an insertion
- The two neighbors of the node to be deleted are linked directly to each other, thereby bypassing the original node
- As a result, that node will no longer be considered part of the list and it can be reclaimed by the system



Advantages of Array-Based Sequences

- Arrays provide $O(1)$-time access to an element based on an integer index
- In contrast, locating the k^{th} element in a linked list requires $O(k)$ time to traverse the list from the beginning, or possibly $O(n - k)$ time, if traversing backward from the end of a doubly linked list
- Operations with equivalent asymptotic bounds typically run a constant factor more efficiently with an array-based structure versus a linked structure
- For instance, the enqueue operation for an array-based queue involves calculation of the new index and storing a reference to the element in the array
- The same operation for a link-based queue requires more expensive instantiation of a node and appropriate linking of nodes

Advantages of Array-Based Sequences

- Array-based representations typically use proportionally less memory than linked structures
- Both array-based lists and linked lists are referential structures, so the primary memory for storing the actual objects that are elements is the same for either structure
- For an array-based container of n elements, a typical worst case may be that a recently resized dynamic array has allocated memory for $2n$ object references
- With linked lists, memory must be devoted not only to store a reference to each contained object, but also explicit references that link the nodes
- So a singly linked list of length n already requires $2n$ references, and a doubly linked list - $3n$ references

Advantages of Link-Based Sequences

- Link-based structures provide worst-case time bounds for their operations in contrast to the amortized bounds associated with the expansion or contraction of a dynamic array
- When we only care about the total time of computation, an amortized bound is as good as a worst-case bound because it gives a guarantee on the sum of the time spent on the individual operations
- However, if data structure operations are used in a real-time system that is designed to provide more immediate responses, a long delay caused by a single (amortized) operation may have an adverse effect

Advantages of Link-Based Sequences

- Link-based structures support $O(1)$-time insertions and deletions at arbitrary positions
- Inserting or deleting an element from the end of an array-based list can be done in constant time
- However, more general insertions and deletions are expensive
- For example, with Python's array-based list class, a call to `insert` or `pop` with index k uses $O(n - k + 1)$ time because of the loop to shift all subsequent elements