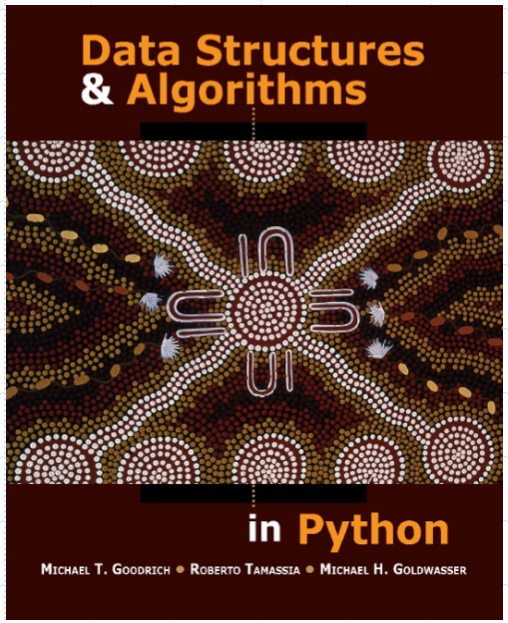
Welcome to DS2030 – Data Structures and Algorithms for Data Science

Narayanan (CK) C Krishnan
ckn@iitpkd.ac.in

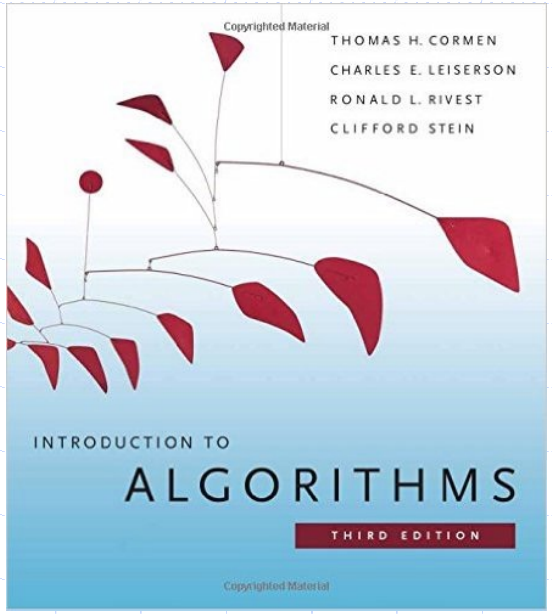
Administrative Trivia

- Course Structure
  - 3-0-3 (5 credits)
- Lecture Timings
  - Monday 9.00-9.50am
  - Wednesday 9.00-9.50am
  - Friday 9.00-9.50am
- Lab hours
  - Tuesday 2.00-5.00pm
- Teaching Assistant
  - Head TA – Hritik Suresh
  - Bhaskar Sharma
  - Yeswanth Sai
  - Vidhya Sudevan
  - Sivani E
- Office Hours
  - Instructor – M,W,F immediately after class or appointment over email
- Pre-registered students have already been added
  - other students – send email to the instructor

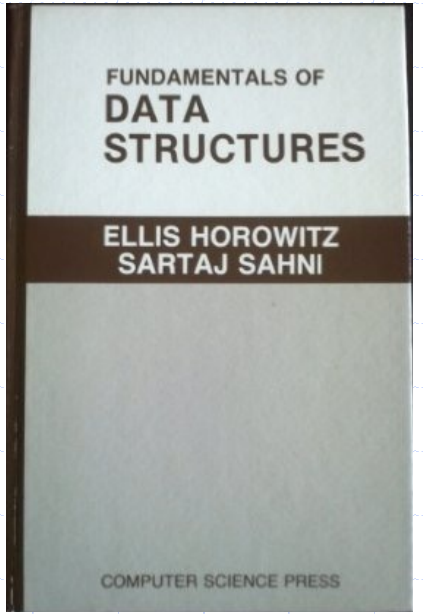
Reference Material



Data Structures and Algorithms in Python
Goodrich, Tamasia and Goldwasser
1st edition



Introduction to Algorithms
Cormen, Leiserson, Rivest, Stein
3rd edition



Fundamentals of Data Structures
Horowitz Sahni

Tentative Course Schedule

Week	Topic	Quiz	Lab
1	Introduction, OOP, Arrays, and Linked Lists		L0
2	Analysis Tools and Recursion		L0
3	Stacks and Queues		L1
4	Tree Structures and Heaps		L2
5	Sorting and Priority Queues		L3
6	Maps, Hash Tables	Q1 (Test 1)	Lab buffer
7	Search Tree Structures I		L4
8	Search Tree Structures II		Lab Exam 1
9	Search Tree Structures III		L5
10	Buffer Week		L6
11	String Processing	Q2 (Test 2)	
12	Dynamic Programming		Lab buffer
13	Graphs I		L7
14	Graphs II		L8
15	Graphs III		Lab buffer
16	Buffer Week		Lab Exam 2

Quizzes – 20%

- Institute mandated Test 1 and Test 2
- Cover material discussed from the previous quiz till the current week
- Duration – 50 minutes
- Makeup quizzes will be allowed Test 1 and Test 2 according to the institute mandated guidelines

Quiz	Week
Q1	6
Q2	11

Graded Labs – 30%

- Programming Assignments Consisting of two graded parts
  - Take Home: announced one week prior to the submission deadline
    - Treat it as a practice lab
  - In Person: announced at the start of the lab hour and should be submitted at the end of the lab hour
    - Internet access will not be available during the in-person lab hours
    - Discussions are allowed
    - Instructor and TA will be available for assistance
- Python – programming language
  - Warm up Lab (L0) will be posted soon.

Lab Number	Week
L1	3
L2	4
L3	5
L4	7
L5	9
L6	10
L7	13
L8	14

Exams – 50%

- Lab Exams – 20%
  - No internet, no discussions allowed
- Two exams
  - Lab exam 1 Week 8
  - Lab exam 2 Week 16
- No makeup lab exams will be allowed.
- End-semester Exam – 30%
  - As per the institute exam schedule
  - Covers the entire syllabus

Grading Scheme

- Tentative Breakup
  - Quizzes (2) - 20%
  - Labs (8) - 30%
  - Lab Exams (2) - 20%
  - End-Semester Exam - 30%

Attendance – Bonus 1%

- Encourage you to attend classes
- Attendance will be taken during every class
- 1% will count towards the final score
  - Borderline grades will have an edge

Honor Code

- Encourage to form study groups to review class lectures/textbook material
  - Group discussions are encouraged
- Unless explicitly stated all quizzes and labs
  - are to be done individually.
  - web trawled code or code copied from classmates is strictly forbidden.
- Any infraction will be dealt in severest terms
  - Direct Fail grade in the course.

I reserve the right to question you, if I suspect any misconduct in any submission.

Course Website

- All class related material will be accessible from the course module on LMS.

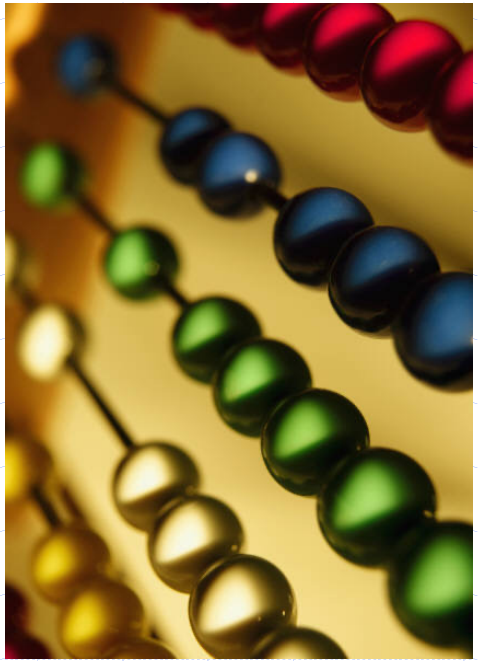
Data Structures

- What are data structures?

Data Structures

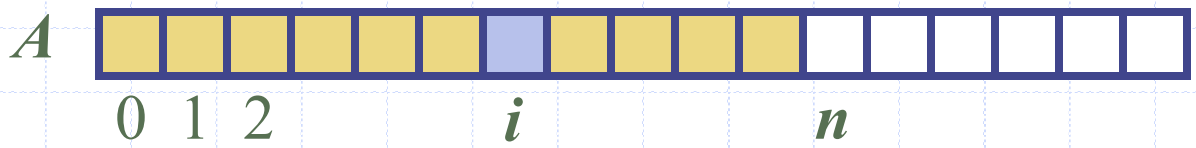
- What are data structures?
  - a way to organize data
  - easier access/manipulate to data
- Topics for the next couple of weeks
  - Concrete Data Structures
    - Arrays and Linked Lists
  - Algorithmic Analysis
  - Recursion

Arrays



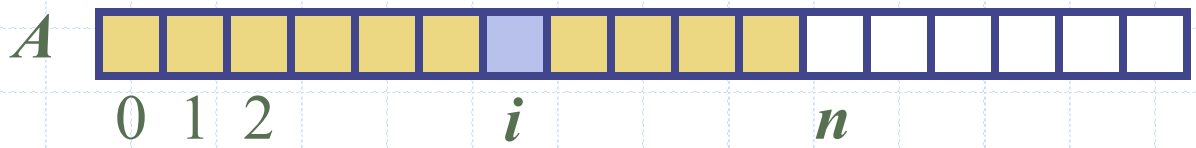
Python Sequence Classes

- Python has built-in types, **list**, **tuple**, and **str**.
- Each of these **sequence** types supports indexing to access an individual element of a sequence, using a syntax such as A[i]
- Each of these types uses an **array** to represent the sequence.
  - An array is a set of memory locations that can be addressed using consecutive indices, which, in Python, start with index 0.



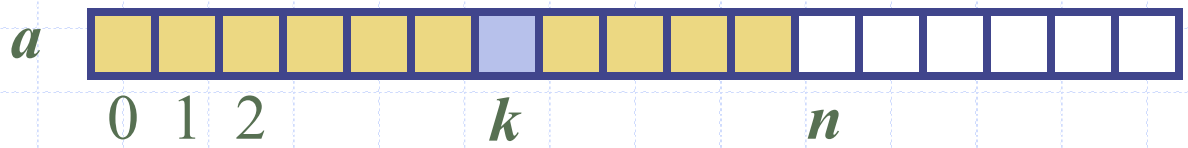
Array Definition

- An ***array*** is a sequenced collection of variables all of the same type. Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell. The cells of an array, A, are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an ***element*** of that array.



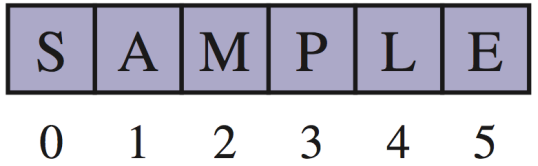
Array Length and Capacity

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its ***capacity***.
- In Python, the length of an array named *a* can be accessed using the syntax *len()*. Thus, the cells of an array, *a*, are numbered 0, 1, 2, and so on, up through *len(a)*–1, and the cell with index *k* can be accessed with syntax *a*[*k*].

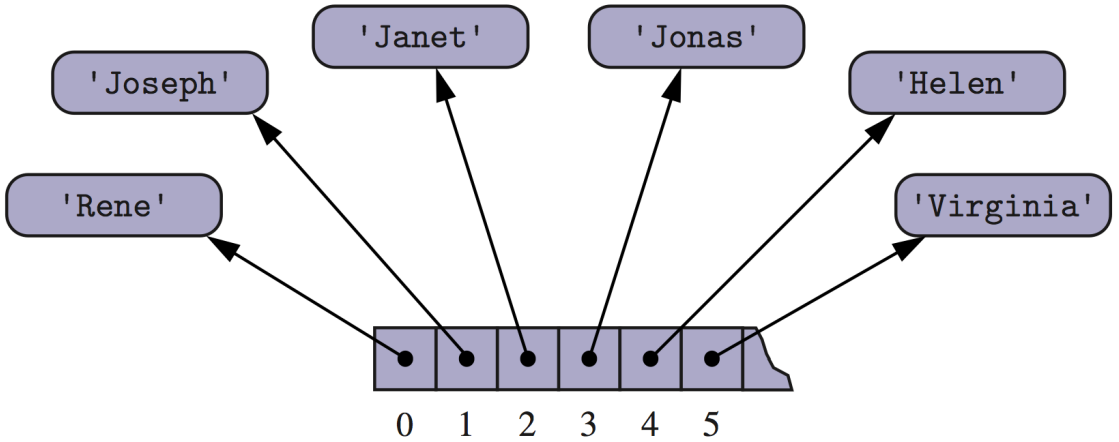


Arrays of Characters or Object References

- An array can store primitive elements, such as characters, giving us a **compact array.**



- An array can also store references to objects.



Compact Arrays

- Primary support for compact arrays is in a module named **array**.
  - That module defines a class, also named array, providing compact storage for arrays of primitive data types.
- The constructor for the array class requires a type code as a first parameter, which is a character that designates the type of data that will be stored in the array.

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

Type Codes in the array Class

- Python's array class has the following type codes:

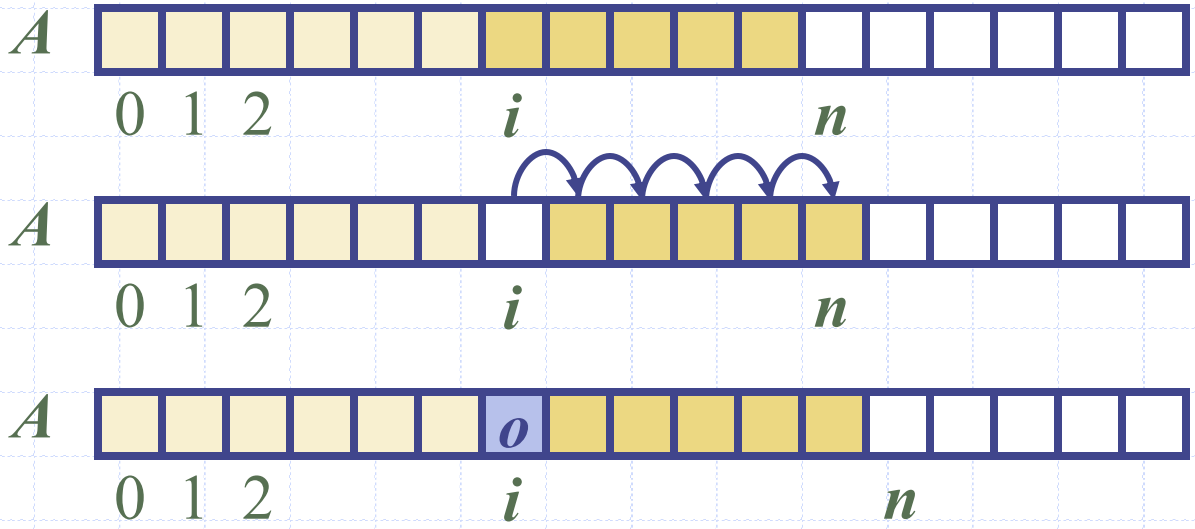
Code	C Data Type	Typical Number of Bytes
`'b'`	signed char	1
`'B'`	unsigned char	1
`'u'`	Unicode char	2 or 4
`'h'`	signed short int	2
`'H'`	unsigned short int	2
`'i'`	signed int	2 or 4
`'I'`	unsigned int	2 or 4
`'l'`	signed long int	4
`'L'`	unsigned long int	4
`'f'`	float	4
`'d'`	float	8

Insertion

- Array - A, Length - N
- Insert object ‘o’ at index k
  - A[0], A[1],…A[N]
- Solution - modifies the array
  - A[k] = o
- Solution
  - Create a new array B - size N+1
    - Copy all the elements in A to B until the index k
    - B[k] = o
    - Copy the remaining elements in A starting from k to N into B from k+1 through N+1

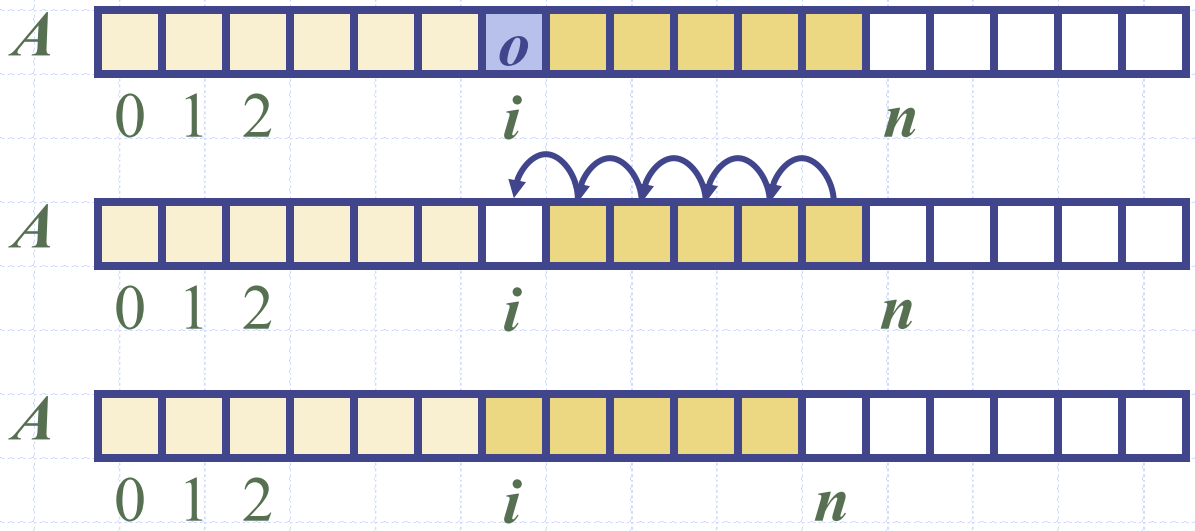
Insertion

- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

- In an operation *remove*(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i+1], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

- In an array-based implementation of a dynamic list:
  - The space used by the data structure is $O(n)$
  - Indexing the element at I takes $O(1)$ time
  - *add* and *remove* run in $O(n)$ time in worst case
- In an *add* operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one…

Growable Array-based Array List

- In an add(o) operation (without an index), we could always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

```
Algorithm add(o)
  if t = len(S) – 1 then
    A ← new array of
          size …
    for i ← 0 to n–1 do
      A[i] ← S[i]
    S ← A
  n ← n + 1
  S[n–1] ← o
```

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

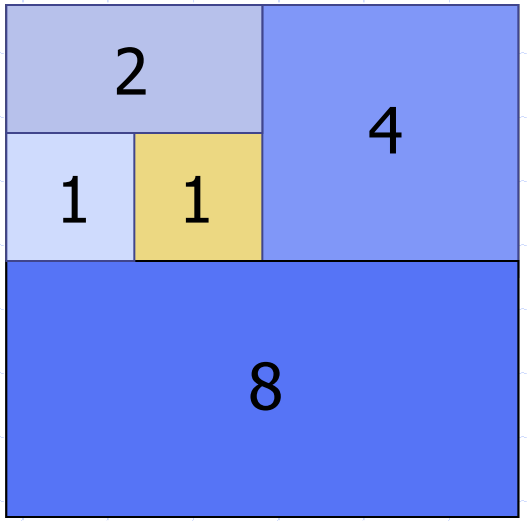
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series



Python Implementation

```
1  import ctypes                                    # provides low-level arrays
2
3  class DynamicArray:
4    """A dynamic array class akin to a simplified Python list."""
5
6    def __init__(self):
7      """Create an empty array."""
8      self._n = 0                                  # count actual elements
9      self._capacity = 1                           # default array capacity
10     self._A = self._make_array(self._capacity)   # low-level array
11
12   def __len__(self):
13     """Return number of elements stored in the array."""
14     return self._n
15
16   def __getitem__(self, k):
17     """Return element at index k."""
18     if not 0 <= k < self._n:
19       raise IndexError('invalid index')
20     return self._A[k]                            # retrieve from array
21
22   def append(self, obj):
23     """Add object to end of the array."""
24     if self._n == self._capacity:                # not enough room
25       self._resize(2 * self._capacity)           # so double capacity
26     self._A[self._n] = obj
27     self._n += 1
28
29   def _resize(self, c):                          # nonpublic utitity
30     """Resize internal array to capacity c."""
31     B = self._make_array(c)                      # new (bigger) array
32     for k in range(self._n):                     # for each existing value
33       B[k] = self._A[k]
34     self._A = B                                  # use the bigger array
35     self._capacity = c
36
37   def _make_array(self, c):                      # nonpublic utitity
38     """Return new array with capacity c."""
39     return (c * ctypes.py_object)( )             # see ctypes documentation
```

Insertion Sort - Example

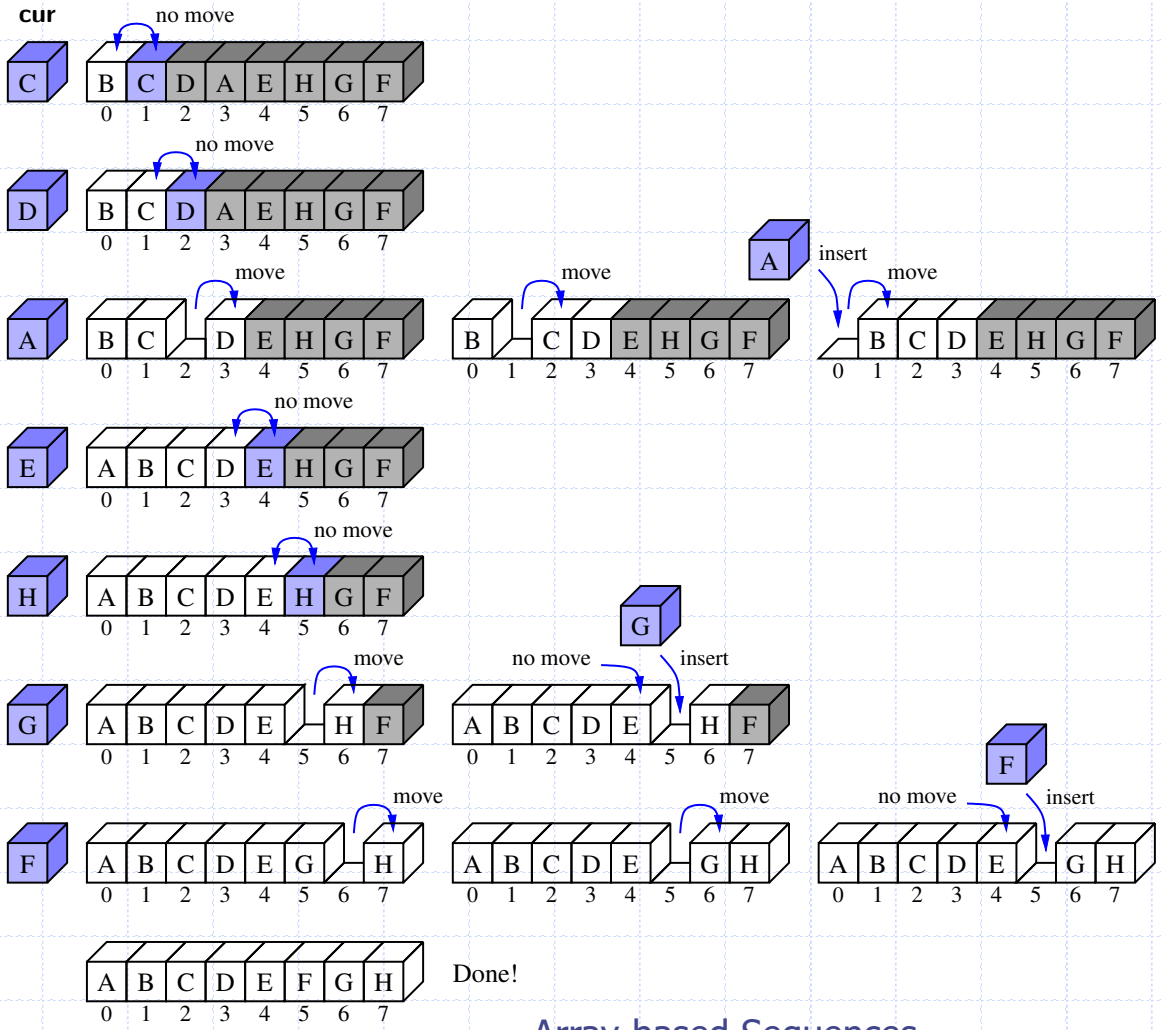


Figure 5.20

Sorting an Array – Insertion Sort

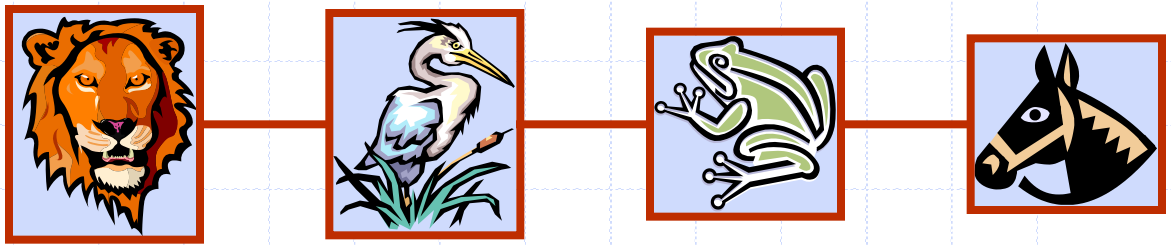
```
def insertion_sort(A):
  """Sort list of comparable elements into nondecreasing order."""
  for k in range(1, len(A)):                  # from 1 to n-1
    cur = A[k]                                # current element to be inserted
    j = k                                     # find correct index j for current
    while j > 0 and A[j-1] > cur:             # element A[j-1] must be after current
      A[j] = A[j-1]
      j -= 1
    A[j] = cur                                # cur is now in the right place
```

Code Fragment 5.10

Drawbacks of Arrays

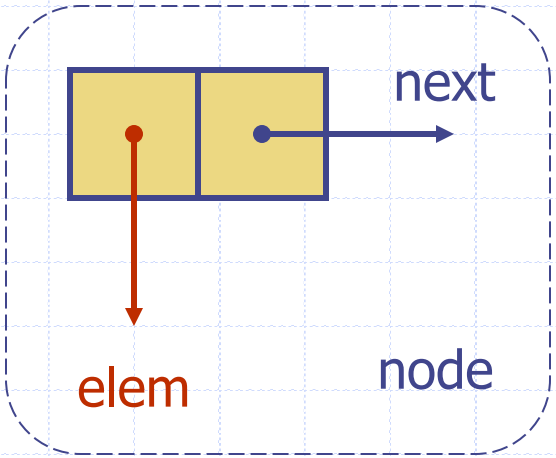
- Fixed capacity
- Insertions and Deletions involve many shifting operations

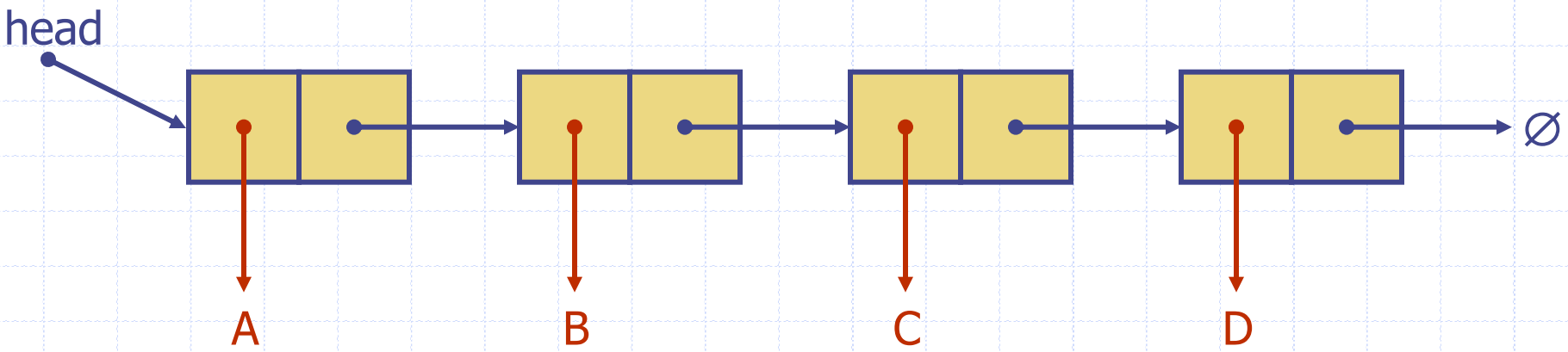
Singly Linked Lists



Singly Linked List

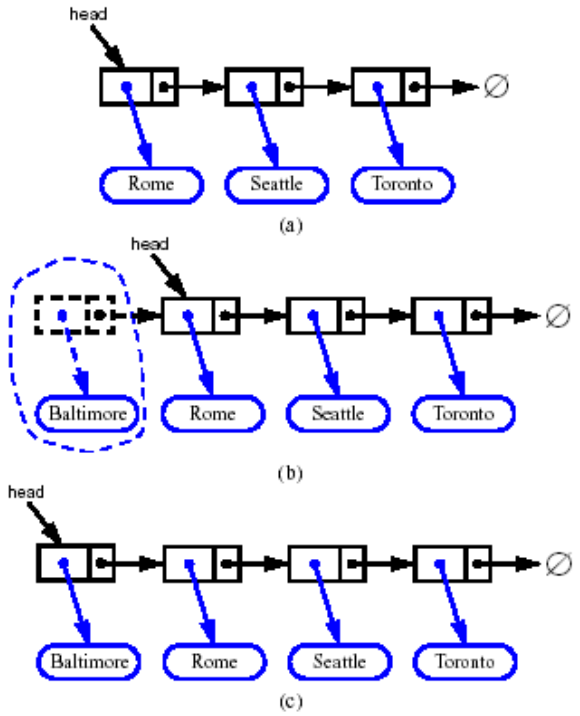
- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





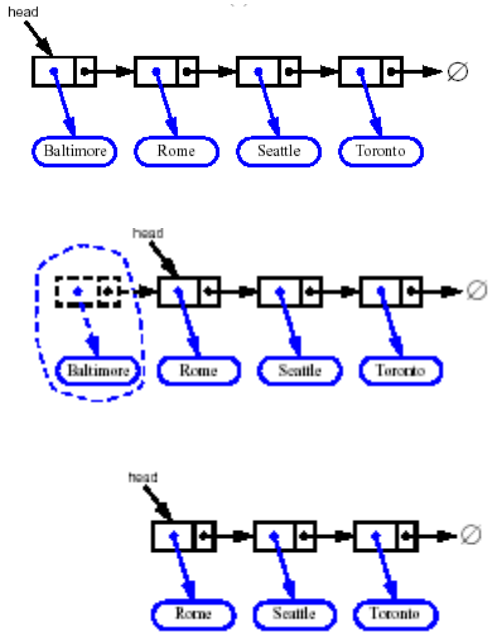
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



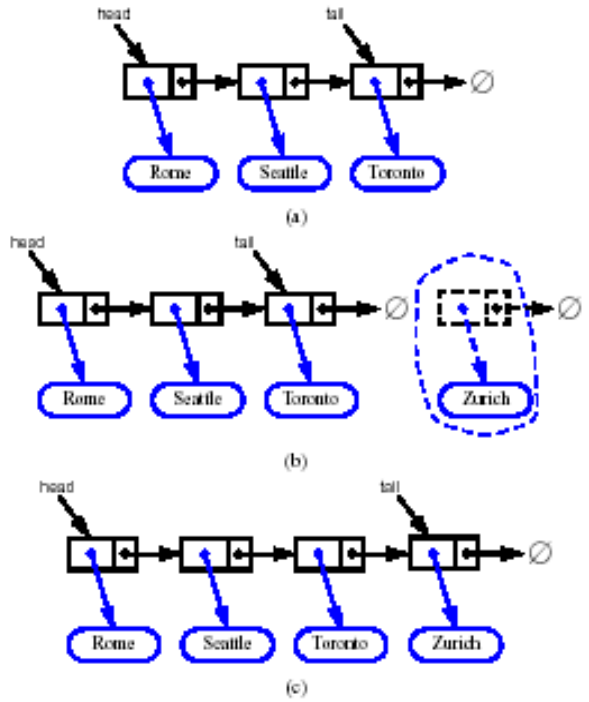
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



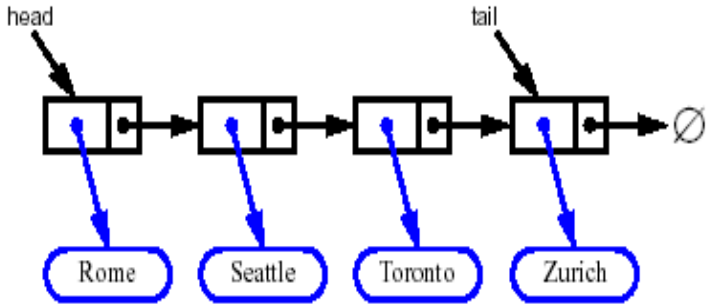
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

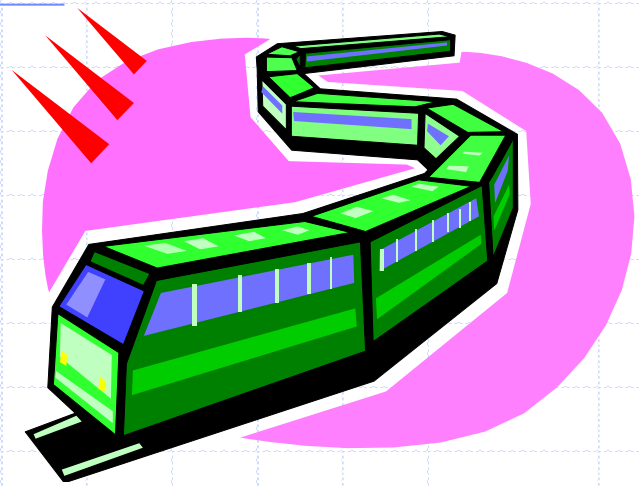


Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node

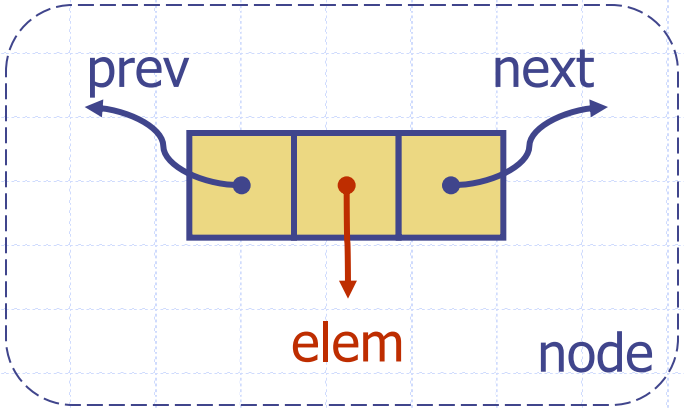


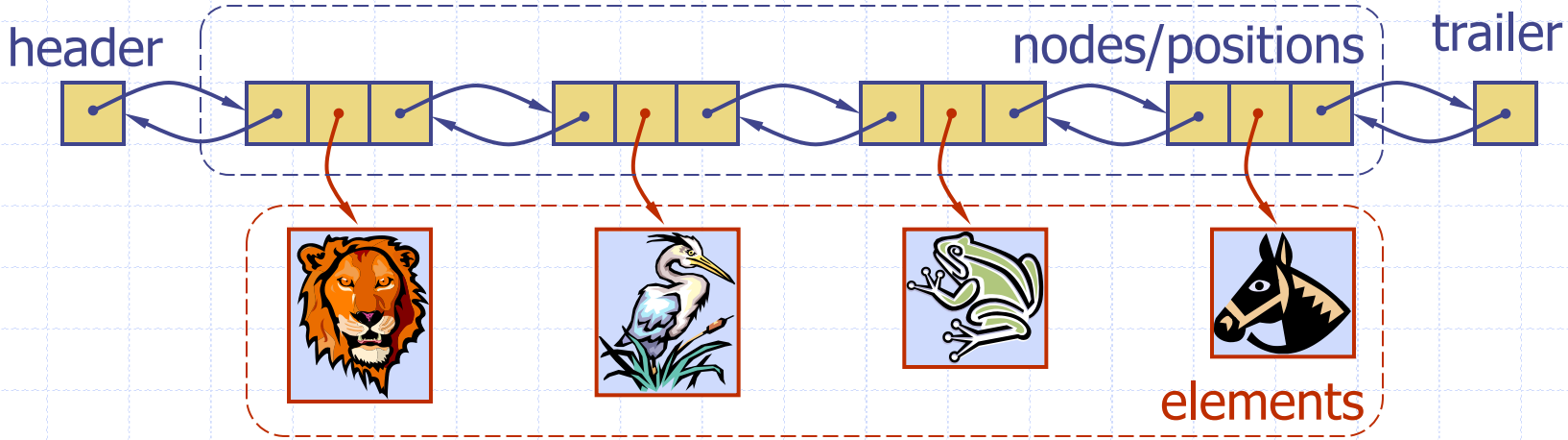
Doubly Linked Lists



Doubly Linked List

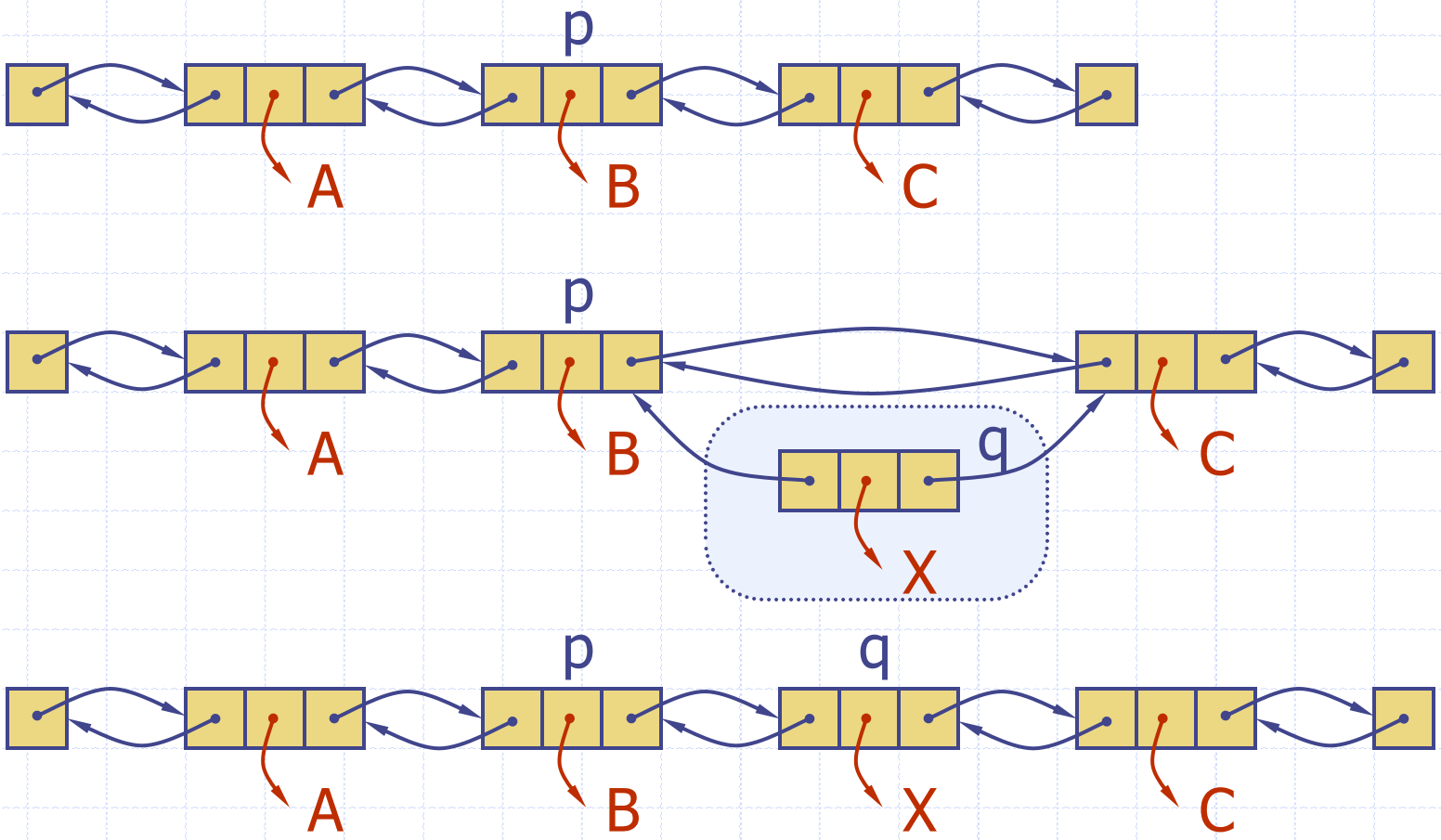
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





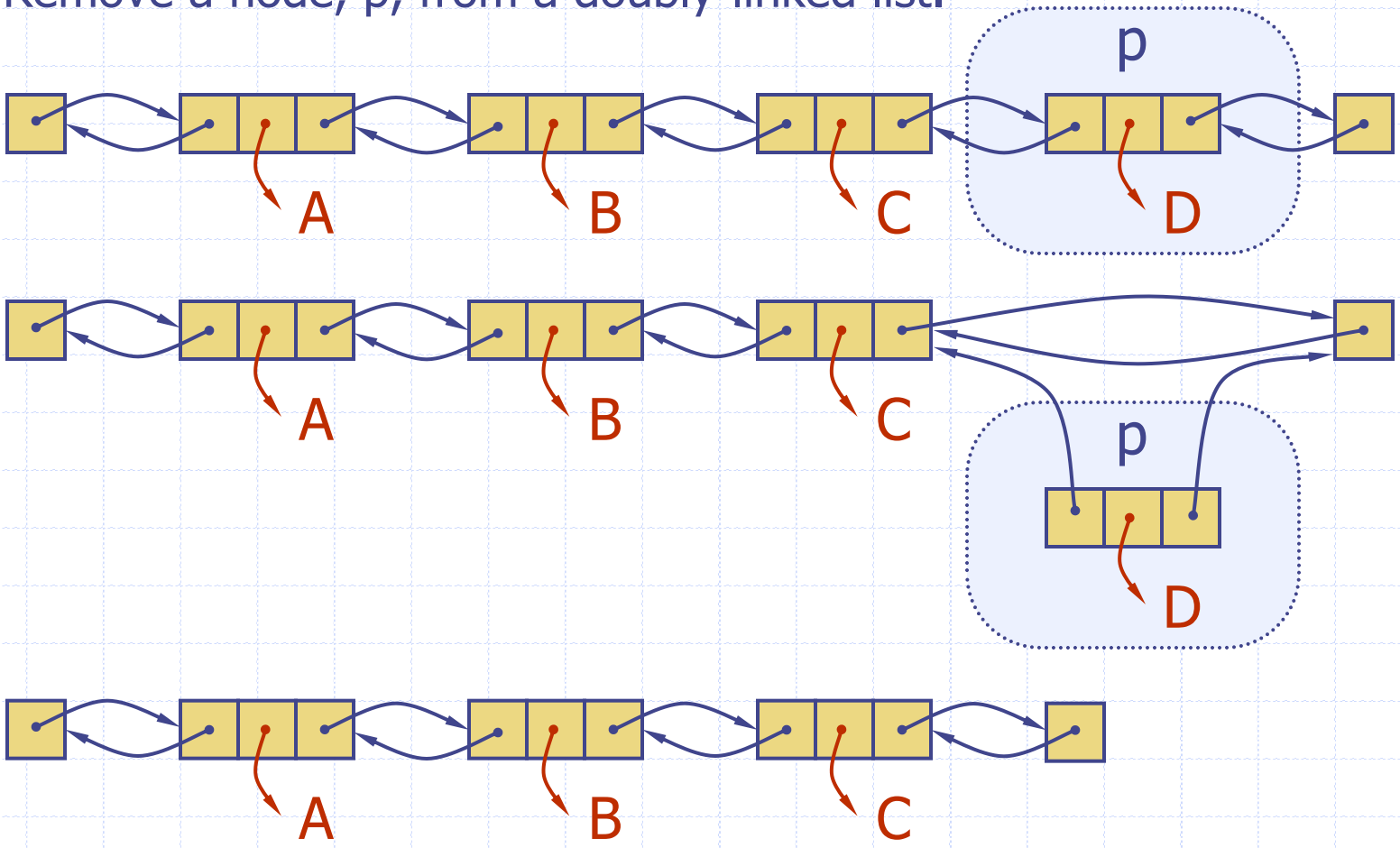
Insertion

- Insert a new node, q, between p and its successor.



Deletion

- Remove a node, p, from a doubly-linked list.



Doubly-Linked List in Python

```
1  class _DoublyLinkedBase:
2    """A base class providing a doubly linked list representation."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a doubly linked node."""
6      (omitted here; see previous code fragment)
7
8    def __init__(self):
9      """Create an empty list."""
10     self._header = self._Node(None, None, None)
11     self._trailer = self._Node(None, None, None)
12     self._header._next = self._trailer                # trailer is after header
13     self._trailer._prev = self._header                # header is before trailer
14     self._size = 0                                    # number of elements
15
16   def __len__(self):
17     """Return the number of elements in the list."""
18     return self._size
19
20   def is_empty(self):
21     """Return True if list is empty."""
22     return self._size == 0
23
```

```
24   def _insert_between(self, e, predecessor, successor):
25     """Add element e between two existing nodes and return new node."""
26     newest = self._Node(e, predecessor, successor)   # linked to neighbors
27     predecessor._next = newest
28     successor._prev = newest
29     self._size += 1
30     return newest
31
32   def _delete_node(self, node):
33     """Delete nonsentinel node from the list and return its element."""
34     predecessor = node._prev
35     successor = node._next
36     predecessor._next = successor
37     successor._prev = predecessor
38     self._size -= 1
39     element = node._element                          # record deleted element
40     node._prev = node._next = node._element = None   # deprecate node
41     return element                                   # return deleted element
```

Performance

- In a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the standard operations of a list run in $O(1)$ time