17 Sorting and Insertion Sort

Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

April 1, 2019

Outline

1. **Sorting**
2. **Insertion Sort**
   - Array-Based Lists
   - Linked Lists
   - Analysis

Sorting

- Given a list of elements, sorting returns a list of the elements in ascending order or descending order.

$$3, 2, 8, 5, 1, 7, 6, 4, 9 \text{ becomes } 1, 2, 3, 4, 5, 6, 7, 8, 9.$$

- Our sorting is based on *comparisons* of two elements, so we assume the input elements can be compared with each other.
- A list of elements can be stored in an array-based list or a linked list.
- Since sorting is a re-arrangement of elements, for an array-based list, *swapping* two elements, and sometimes *rotating* several elements are essential.
- For a linked list, we change link references to rearrange nodes.

Stability

If duplicated keys are allowed in a list, and the elements each have other components besides keys, for example

$$\langle number, name \rangle \,,$$

then the result of sorting is not unique. If two elements of the same key are in their *original* input order after the sorting, we call the sorting *stable*. Otherwise, the sorting is *unstable*.

- Stable:

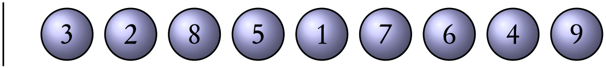
$$\langle 3,a\rangle, \langle 1,b\rangle, \langle 4,c\rangle, \langle 3,d\rangle, \langle 1,e\rangle \longrightarrow \langle 1,b\rangle, \langle 1,e\rangle, \langle 3,a\rangle, \langle 3,d\rangle, \langle 4,c\rangle$$

- Unstable:

$$\langle 3,a\rangle, \langle 1,b\rangle, \langle 4,c\rangle, \langle 3,d\rangle, \langle 1,e\rangle \longrightarrow \langle 1,e\rangle, \langle 1,b\rangle, \langle 3,a\rangle, \langle 3,d\rangle, \langle 4,c\rangle$$

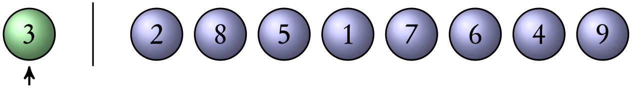
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



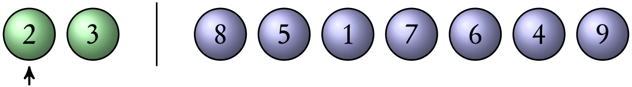
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



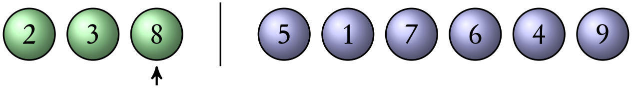
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



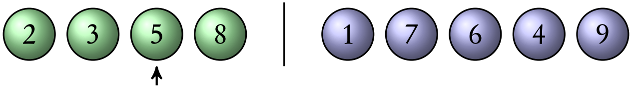
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



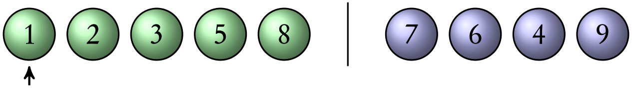
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



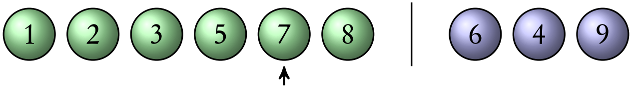
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



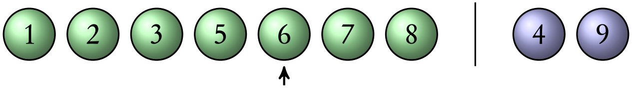
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



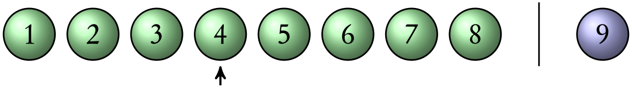
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



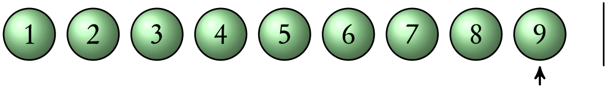
Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.



Insertion Sort

- Insertion sort is one of the simplest and most straightforward sorting algorithms. We just keep inserting elements to an ordered list, while maintaining the order.
- To do this in-place for arrays, we break the input list into two parts, one is an ordered list (already sorted, initially empty), the other is the list of unsorted elements.
  We extend the ordered part while shrinking the unsorted part, by moving (*rotating*) elements.
- For linked lists, it is simpler. We can insert/delete a node to/from a list in constant time. We need only to find the right places in the list.

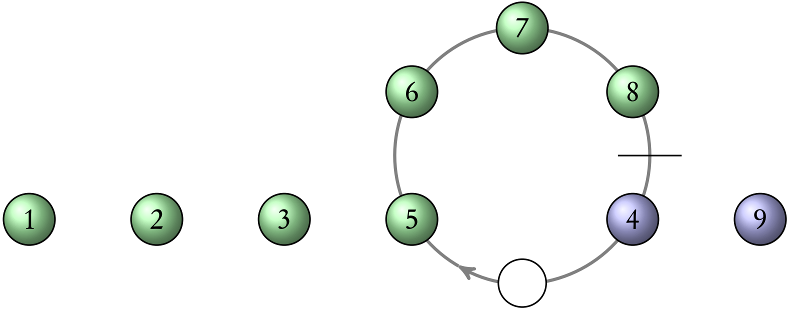


Rotating Array Elements

To remove a rear element in an array and insert it into some front position, for example

$$a_0, \ldots, a_i, \ldots, a_j, a_{j+1}, \ldots, a_{n-1} \longrightarrow a_0, \ldots, a_{j+1}, a_i, \ldots, a_j, \ldots, a_{n-1}$$

is to rotate the elements in between.

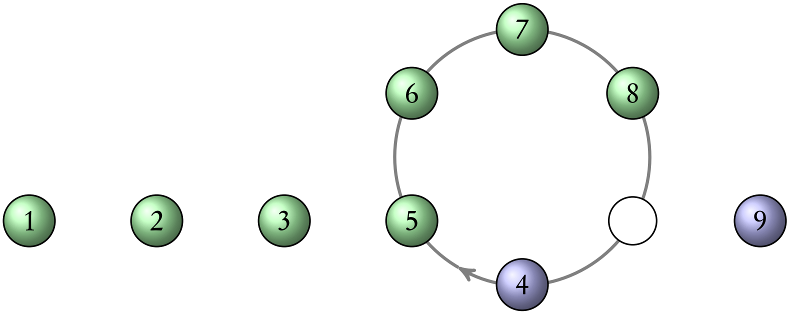


Rotating Array Elements

To remove a rear element in an array and insert it into some front position, for example

$$a_0, \ldots, a_i, \ldots, a_j, a_{j+1}, \ldots, a_{n-1} \longrightarrow a_0, \ldots, a_{j+1}, a_i, \ldots, a_j, \ldots, a_{n-1}$$

is to rotate the elements in between.

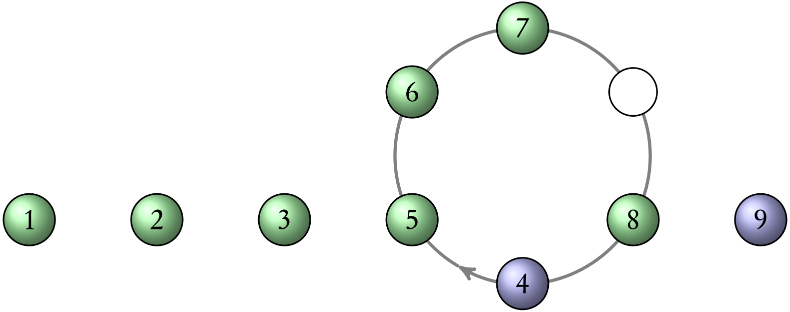


Rotating Array Elements

To remove a rear element in an array and insert it into some front position, for example

$$a_0, \ldots, {\color{blue}a_i, \ldots, a_j, a_{j+1}}, \ldots, a_{n-1} \longrightarrow a_0, \ldots, {\color{blue}a_{j+1}, a_i, \ldots, a_j}, \ldots, a_{n-1}$$

is to rotate the elements in between.

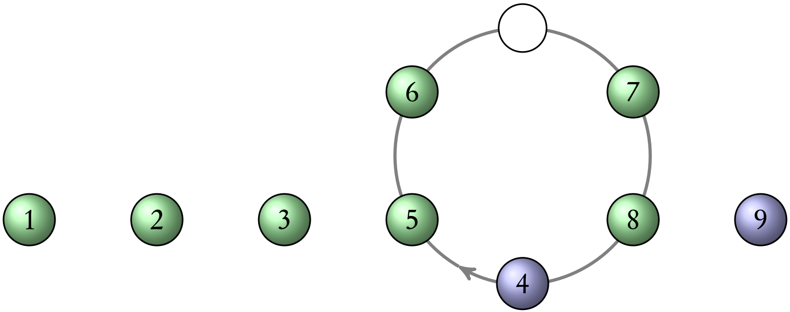


Rotating Array Elements

To remove a rear element in an array and insert it into some front position, for example

$$a_0, \dots, a_i, \dots, a_j, a_{j+1}, \dots, a_{n-1} \longrightarrow a_0, \dots, a_{j+1}, a_i, \dots, a_j, \dots, a_{n-1}$$

is to rotate the elements in between.

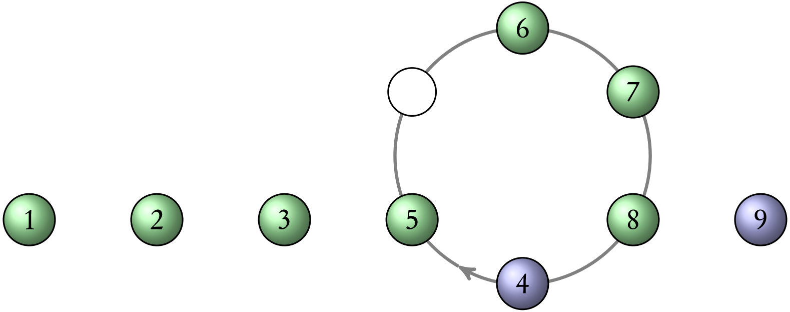


Rotating Array Elements

To remove a rear element in an array and insert it into some front position, for example

$$a_0, \ldots, a_i, \ldots, a_j, a_{j+1}, \ldots, a_{n-1} \longrightarrow a_0, \ldots, a_{j+1}, a_i, \ldots, a_j, \ldots, a_{n-1}$$

is to rotate the elements in between.

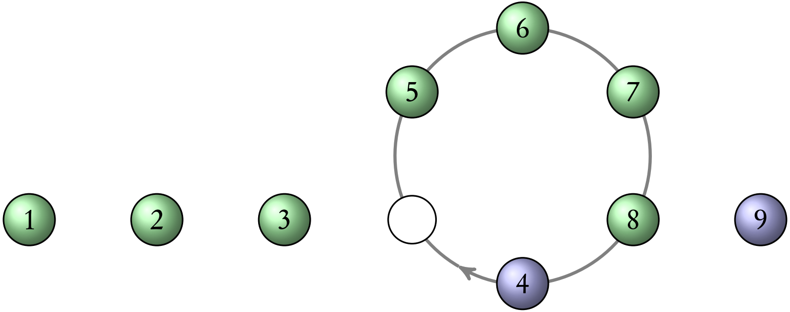


Rotating Array Elements

To remove a rear element in an array and insert it into some front position, for example

$$a_0, \ldots, a_i, \ldots, a_j, a_{j+1}, \ldots, a_{n-1} \longrightarrow a_0, \ldots, a_{j+1}, a_i, \ldots, a_j, \ldots, a_{n-1}$$

is to rotate the elements in between.

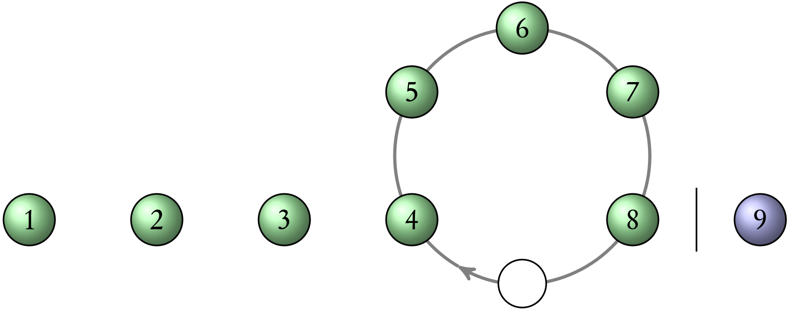


Rotating Array Elements

To remove a rear element in an array and insert it into some front position, for example

$$a_0, \ldots, a_i, \ldots, a_j, a_{j+1}, \ldots, a_{n-1} \longrightarrow a_0, \ldots, a_{j+1}, a_i, \ldots, a_j, \ldots, a_{n-1}$$

is to rotate the elements in between.



Insertion Sort — Insertion of the i^{th} Element

Rotate $a_{i-4}, \ldots, a_i$ (conditionally)

```
t      = a[i]
a[i]   = a[i-1]   # a_{i-1} > t
a[i-1] = a[i-2]   # a_{i-2} > t
a[i-2] = a[i-3]   # a_{i-3} > t
a[i-3] = a[i-4]   # a_{i-4} > t
a[i-4] = t        # a_{i-5} ≯ t
```

Rotate $a_j, \ldots, a_i$ where $a_{j-1} \leqslant a_i < a_j$

```
t = a[i]
j = i
while j > 0 and not a[j-1] <= t:
    a[j] = a[j-1]
    j -= 1
a[j] = t
```

Insertion Sort on Array-Based Lists

We keep inserting elements, from front to rear, into the ordered list occupying the front part of the array, starting from containing only the first element, until all the elements are inserted. The function *insertion_sort_a* sorts the elements of an array-based list *a*.

```
def insertion_sort_a(a):
    for i in range(1, len(a)):
        t = a[i]
        j = i
        while j > 0 and not a[j-1] <= t:
            a[j] = a[j-1]
            j -= 1
        a[j] = t
```

Insertion Sort on Doubly Linked Lists

For doubly linked lists, we don't need to move elements while comparing, we just scan for the proper position of each node, and perform a removal (from the old position) then an insertion (to the new position). The method *insertion_sort_c* sorts a circular doubly linked list, given its *dummy* node.

```
def insertion_sort_c(dummy):
    p = dummy.nxt.nxt
    while p is not dummy:
        r, q = p.prv, p.nxt
        while r is not dummy and not r.elm <= p.elm:
            r = r.prv
        if r.nxt is not p:
            delete_node(p)
            insert_node(p, r.nxt)
        p = q
```

Insertion Sort on Singly Linked Lists

For a singly linked list pointed to by *h*, we initialize an empty list pointed to by *s*, then repeatedly "pop" nodes from *h* and insert them to proper locations in *s*. The function *insertion_sort_l* sorts a singly linked list *h* and returns the new head node reference.

```
def insertion_sort_l(h):
    s = None
    while h is not None:
        p, h = h, h.nxt
        if s is None or not s.elm <= p.elm:
            s, p.nxt = p, s
        else:
            r, q = s, s.nxt
            while q is not None and q.elm <= p.elm:
                r, q = q, q.nxt
            r.nxt, p.nxt = p, q
    return s
```

Analysis of Insertion Sort

For a list of n elements, we only need a fix amount of auxiliary space for insertion sort.

- $\mathcal{O}(1)$ auxiliary space.

We count the number of element comparisons.

- *Best case* : when the input list is sorted, each element is compared once with its predecessor, thus there are $n-1$ comparisons, i.e. $\mathcal{O}(n)$.
- *Worst case* : when the input list is in reverse order, each element must go through all the way to the very front, thus there are $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ comparisons, i.e. $\mathcal{O}(n^2)$.
- *Average case* : each element is expected to go halfway to the front, thus there are expected $\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$ comparisons, i.e. $\mathcal{O}(n^2)$.

Insertion sort is stable, since we step an element towards the front only when its predecessor is *greater*, but *not* equal.