STL CONTAINER METHODS

- The following methods are defined for STL vectors, STL lists, and STL deques:

  vector(p,q) or list(p,q) or deque(p,q): **Construct** the container **by iterating** the range [p,q) and **copying** each of these elements into the new container.

Example:

```
vector<int> x;
. . .
vector<int> y( x.begin(), x.end() );
```

This way, y is assigned a new vector containing copies of the elements in x.
Of course, x must have the same base type as y.

> Note that the method operates on the range [p,q), i.e., from p to q-1. However, since x.end() returns an imaginary element that lies just after the end of the container, then y(x.begin(), x.end()) would copy **all** elements of x.

STL CONTAINER METHODS

- The following methods are defined for STL vectors, STL lists, and STL deques:

  vector(p,q) or list(p,q) or deque(p,q): **Construct** the container **by iterating** the range [p,q) and **copying** each of these elements into the new container.

  assign(p,q): **Delete** the contents, and **assigns its new contents by iterating** the range [p,q) and **copying** each of these elements into the container.

Example:

```
vector<int> x;
. . .
y.assign( x.begin(), x.end() );
```

This way, the content of y will be deleted, and then y will be assigned a new vector containing copies of the elements in x. Of course, x must have the same base type as y.

STL CONTAINER METHODS

- The following methods are defined for STL vectors, STL lists, and STL deques:

  vector(p,q) or list(p,q) or deque(p,q): **Construct** the container **by iterating** the range [p,q) and **copying** each of these elements into the new container.

  assign(p,q): **Delete** the contents, and **assigns its new contents by iterating** the range [p,q) and **copying** each of these elements into the container.

  insert(p,e): **Insert a copy** of e just prior to the position given by iterator p and **shift** the subsequent elements one position to the right. (except for STL lists, where no shifting occurs, since it is **implemented as a doubly-linked list**)

  erase(p): **Remove and destroy** the element at the position given by p and **shift** the subsequent elements one position to the left. (except for STL lists)

  erase(p,q): Iterate the range [p,q), **removing and destroying** all these elements and **shifting** subsequent elements to the left to fill the gap. (except for STL lists)

  clear(): **Delete** all elements of the container.

STL CONTAINER METHODS

- The following methods are defined for STL vectors, STL lists, and STL deques:

  sort(p,q): **Sort** the elements in the range [p,q) in **ascending order**. It is assumed that ("<") is defined for the base type. (sort is not supported by STL lists)

<u>Example:</u>

```
vector<int> v;
. . .
v.sort( v.begin(), v.end() ); // this sorts the vector
v.sort( v.begin(), v.begin()+10 ); // this sorts the first 10
                                   // elements of the vector
```

STL CONTAINER METHODS

- The following methods are defined for STL vectors, STL lists, and STL deques:

  sort(p,q): **Sort** the elements in the range [*p,q*) in **ascending order**. It is assumed that ("<") is defined for the base type. (sort is not supported by STL lists)

  random_shuffle(p,q): **Rearrange** the elements in the range [*p,q*) **in random order**. (random_shuffle is not supported by STL lists)

  reverse(p,q): **Reverse** the elements in the range [*p,q*).

  find(p,q,e): Return an iterator to **the first element** in the range [*p,q*) **that is equal to *e***; if *e* is not found, *q* is returned.

  min_element(p,q): Return an iterator to **the minimum element** in the range [*p,q*).

  max_element(p,q): Return an iterator to **the maximum element** in the range [*p,q*).

  for_each(p,q,f): **Apply the function** *f* to the elements in the range [*p,q*).

Write the output of each line here

EXAMPLE

```
int a[ ] = {17, 12, 33, 15, 62, 45};
vector<int> v(a, a + 6);                          // v: 17 12 33 15 62 45
cout << v.size();                                 // outputs: 6
v.pop_back();                                     // v: 17 12 33 15 62
cout << v.size();                                 // outputs: 5
v.push_back(19);                                  // v: 17 12 33 15 62 19
cout << v.front() << " " << v.back();             // outputs: 17 19
sort(v.begin(), v.begin() + 4);                   // v: (12 15 17 33) 62 19
v.erase(v.end() − 4, v.end() − 2);                // v: 12 15 62 19
cout << v.size();                                 // outputs: 4

char x[ ] = {'b', 'r', 'a', 'v', 'o'};
vector<char> w(x, x + 5);                         // w: b r a v o
// #include <algorithm> to use random_shuffle
random_shuffle(w.begin(), w.end());               // w: o v r a b
w.insert(w.begin(), 's');                         // w: s o v r a b
for (vector<char>::iterator p = w.begin(); p != w.end(); ++p)
    cout << *p << " ";                            // outputs: s o v r a b

return EXIT SUCCESS;
}
```

63

SEQUENCES

SEQUENCES

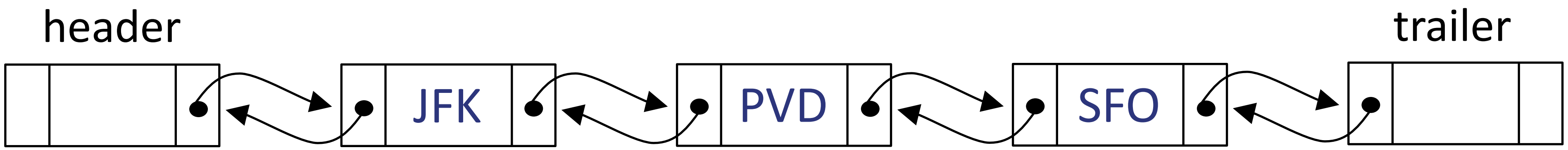
- A sequence is an **ADT** that:
  - Supports all the functions of the **List ADT**
  - Provides functions for accessing elements by their **index** (as in the **Vector ADT**)

More specifically, the Sequence ADT consists of the operations of the List ADT, plus the following two functions:

- atIndex(i): Return the **position/iterator** of the element at **index** i
- indexOf(p): Return the **index** of the element at **position/iterator** p

- A sequence can be implemented using either a **linked list**, or an **array**.

SEQUENCES – LIST IMPLEMENTATION




```
// get index from position
int indexOf(const Iterator& p) const {
   Iterator q = begin();
   int j = 0;
   while (q != p) // until finding p
      {++q; ++j;} // advance and count hops
   return j;
}
```

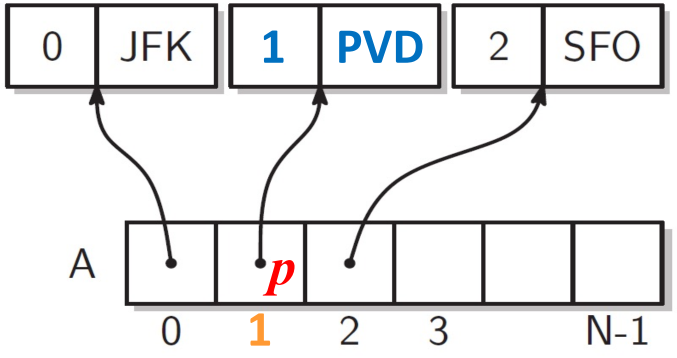
```
// get position from index
Iterator atIndex(int i) const {
   Iterator p = begin();
   for (int j = 0; j < i; j++)
      ++p; // advance
   return p;

}
```

- **Disadvantage:** atIndex(i) and indexOf(p) run in O(n); they <u>may perform n iterations!</u>
- **Advantage:** insert(p, e) and erase(p) run in O(1); <u>no need to shift elements!</u>

SEQUENCES – ARRAY IMPLEMENTATION

- Recall that a sequence is an ADT that supports "indexOf(p)" and "atIndex(i)"
- **If implemented as an array**, we must have an array of positions, each of which points to a pair consisting of an element, and the index of that element.



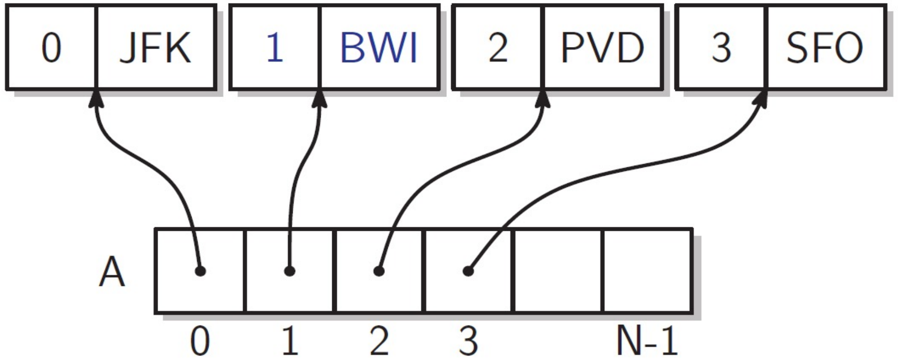
A[1] is a position, ***p***, pointing to a pair consisting of:

- An **element**, which is **PVD**
- The **index** of that element, which is **1**

To insert an element at index 1, we must:

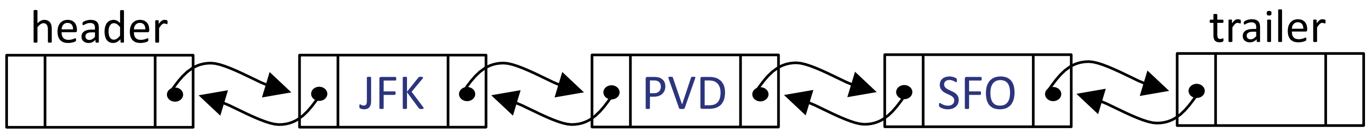
- **Shift** all the positions whose index is ≥ 1
- **Update the indices** of all the positions whose index is ≥ 1

What is the time complexity? O(n)



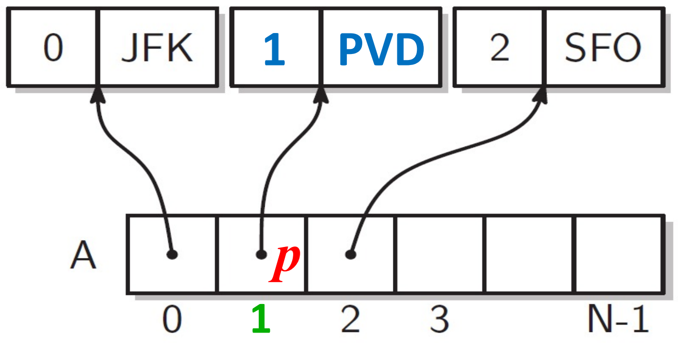
LIST VS. ARRAY IMPLEMENTATION

List:



- **Disadvantage:** atIndex(i) and indexOf(p) run in O(n); they may perform n iterations!
- **Advantage:** insert(p, e) and erase(p) run in O(1); no need to shift elements!

Array:



- **Disadvantage:** insert(p, e) and erase(p) run in O(n); we must shift elements!
- **Advantage:** atIndex(i) and indexOf(p) run in O(1), because:
  - Given an index i, the position at i is stored in A[i].
  - Given a position p, the index of p is stored in the object that p points to.

BUBBLE SORT

BUBBLE SORT

- “Bubble Sort” is a **sorting algorithm** that performs a series of passes over a sequence.
- In each pass, it compares each element to the one after it, and swaps them if needed!

1st pass:

initial array: 5, 7, 2, 6, 9, 3

Swap? No ! 5, 7, 2, 6, 9, 3

Swap? Yes 5, 7, 2, 6, 9, 3

Swap? Yes 5, 2, 7, 6, 9, 3

Swap? No ! 5, 2, 6, 7, 9, 3

Swap? Yes 5, 2, 6, 7, 9, 3

After 1st pass: 5, 2, 6, 7, 3, 9

2nd pass:

initial array: 5, 2, 6, 7, 3, 9

Swap? Yes 5, 2, 6, 7, 3, 9

Swap? No ! 2, 5, 6, 7, 3, 9

Swap? No ! 2, 5, 6, 7, 3, 9

Swap? Yes 2, 5, 6, 7, 3, 9

Swap? No ! 2, 5, 6, 3, 7, 9

After 2nd pass: 2, 5, 6, 3, 7, 9

3rd pass:

initial array: 2, 5, 6, 3, 7, 9

Swap? No ! 2, 5, 6, 3, 7, 9

Swap? No ! 2, 5, 6, 3, 7, 9

Swap? Yes 2, 5, 6, 3, 7, 9

Swap? No ! 2, 5, 3, 6, 7, 9

Swap? No ! 2, 5, 3, 6, 7, 9

After 3rd pass: 2, 5, 3, 6, 7, 9

4th pass:

initial array: 5, 7, 2, 6, 9, 3

Swap? No ! 2, 5, 3, 6, 7, 9

Swap? Yes 2, 5, 3, 6, 7, 9

Swap? No ! 2, 3, 5, 6, 7, 9

Swap? No ! 2, 3, 5, 6, 7, 9

Swap? No ! 2, 3, 5, 6, 7, 9

After 4th pass: 2, 3, 5, 6, 7, 9

BUBBLE SORT

- "Bubble Sort" has the following properties:
  - In 1st pass, once the **largest element** is reached, it keeps on being swapped until it gets to the **last position**
  - In 2nd pass, once the **second largest element** is reached, it keeps on being swapped until it gets to the **second-to-last position**
  - At the end of the i th pass, the **last i elements** (i.e., those at indices n-i to $n-1$) are **sorted**.
- The last property implies that
  - It suffices to **stop after n passes**.
  - The i th pass can be limited to **the first $n-i+1$ elements**.

1st pass:

initial array: 5, 7, 9, 6, 2, 3

Swap? No ! $\boxed{5, 7}$, 9, 6, 2, 3

Swap? No ! 5, $\boxed{7, 9}$, 6, 2, 3

Swap? Yes 5, 7, $\boxed{9, 6}$, 2, 3

Swap? Yes 5, 7, 6, $\boxed{9, 2}$, 3

Swap? Yes 5, 7, 6, 2, $\boxed{9, 3}$

After 1st pass: 5, 7, 6, 7, 3, 9

BUBBLE SORT

- In the example we saw earlier, we can see how:
  - At the end of the i^{th} pass, the **last *i* elements are sorted**, implying that the i^{th} pass can be limited to **the first *n*−*i*+1 elements**.

1^{st} pass:

initial array:		5, 7, 2, 6, 9, 3
Swap?	No !	5, 7, 2, 6, 9, 3
Swap?	**Yes**	5, 7, 2, 6, 9, 3
Swap?	**Yes**	5, 2, 7, 6, 9, 3
Swap?	No !	5, 2, 6, 7, 9, 3
Swap?	**Yes**	5, 2, 6, 7, 9, 3
After 1^{st} pass:		5, 2, 6, 7, 3, **9**

2^{nd} pass:

initial array:		5, 2, 6, 7, 3, 9
Swap?	**Yes**	5, 2, 6, 7, 3, 9
Swap?	No !	2, 5, 6, 7, 3, 9
Swap?	No !	2, 5, 6, 7, 3, 9
Swap?	**Yes**	2, 5, 6, 7, 3, 9
Swap?	No !	2, 5, 6, 3, **7, 9**
After 2^{nd} pass:		2, 5, 6, 3, **7, 9**

3^{rd} pass:

initial array:		2, 5, 6, 3, 7, 9
Swap?	No !	2, 5, 6, 3, 7, 9
Swap?	No !	2, 5, 6, 3, 7, 9
Swap?	**Yes**	2, 5, 6, 3, 7, 9
Swap?	No !	2, 5, 3, **6, 7, 9**
Swap?	No !	2, 5, 3, **6, 7, 9**
After 3^{rd} pass:		2, 5, 3, **6, 7, 9**

4^{th} pass:

initial array:		5, 7, 2, 6, 9, 3
Swap?	No !	2, 5, 3, 6, 7, 9
Swap?	**Yes**	2, 5, 3, 6, 7, 9
Swap?	No !	2, 3, **5, 6, 7, 9**
Swap?	No !	2, 3, **5, 6, 7, 9**
Swap?	No !	2, 3, **5, 6, 7, 9**
After 4^{th} pass:		2, 3, **5, 6, 7, 9**

BUBBLE SORT

- What is the complexity of Bubble Sort?

The number of comparisons is $\approx$ n + (n-1) + (n-2) + … + 2 + 1 = $\frac{n(n+1)}{2}$ which is O(n^2)

1^{st} pass:

initial array:		5, 7, 2, 6, 9, 3
Swap?	No!	5, 7, 2, 6, 9, 3
Swap?	**Yes**	5, 7, 2, 6, 9, 3
Swap?	**Yes**	5, 2, 7, 6, 9, 3
Swap?	No!	5, 2, 6, 7, 9, 3
Swap?	**Yes**	5, 2, 6, 7, 9, 3
After 1^{st} pass:		5, 2, 6, 7, 3, **9**

2^{nd} pass:

initial array:		5, 2, 6, 7, 3, 9
Swap?	**Yes**	5, 2, 6, 7, 3, 9
Swap?	No!	2, 5, 6, 7, 3, 9
Swap?	No!	2, 5, 6, 7, 3, 9
Swap?	**Yes**	2, 5, 6, 7, 3, 9
Swap?	No!	2, 5, 6, 3, **7, 9**
After 2^{nd} pass:		2, 5, 6, 3, **7, 9**

3^{rd} pass:

initial array:		2, 5, 6, 3, 7, 9
Swap?	No!	2, 5, 6, 3, 7, 9
Swap?	No!	2, 5, 6, 3, 7, 9
Swap?	**Yes**	2, 5, 6, 3, 7, 9
Swap?	No!	2, 5, 3, **6, 7, 9**
Swap?	No!	2, 5, 3, **6, 7, 9**
After 3^{rd} pass:		2, 5, 3, **6, 7, 9**

4^{th} pass:

initial array:		5, 7, 2, 6, 9, 3
Swap?	No!	2, 5, 3, 6, 7, 9
Swap?	**Yes**	2, 5, 3, 6, 7, 9
Swap?	No!	2, 3, **5, 6, 7, 9**
Swap?	No!	2, 3, **5, 6, 7, 9**
Swap?	No!	2, 3, **5, 6, 7, 9**
After 4^{th} pass:		2, 3, **5, 6, 7, 9**

BUBBLE SORT

- What is the complexity of Bubble Sort?

  The number of comparisons is $\approx$ n + (n-1) + (n-2) + … + 2 + 1 $= \frac{n(n+1)}{2}$ which is $O(n^2)$

- Clearly, Bubble Sort requires accessing elements at different indices. The above analysis assumes that an **element at index *i* can be accessed in O(1) time**.

- Now, suppose we are sorting a sequence. Then:

  - If we use an **array-based implementation**, where `atIndex(i)` runs in O(1), the total runtime of BubbleSort is indeed $O(n^2)$

  - On the other hand, if we use a **list-based implementation**, where `atIndex(i)` runs in $O(n)$ time, the total runtime becomes $O(n^3)$

- Thus, choosing the appropriate implementation of the ADT matters!