CS213/CS213M DS&A
Sharat Chandran

www.cse.iitb.ac.in/~sharat/current/213m

(Many slides obtained from colleagues, and the Internet and gratefully acknowledged using the fair use copyright law)

Agenda

- Motivation
  - The Standard Template Library in C++ is a very powerful way of coding algorithms
  - In python, the libraries and built-in functions are essentially equivalent in many case
- Containers, Iterators, and Position are fundamental concepts in using STL

Iterator (Python)

- Example: **for** element in **iterable**
  - Clear, concise and convenient
  - Example objects in Python that are **iterable** are (container objects) list, tuple, set; strings; dictionary (keys); file (lines)
- An **iterable** is an object *obj* that produces an iterator via the syntax *iter(obj)*
- An **iterator** is an object that manages an iteration through a series of values. If **i** is an iterator object next(i) produces a subsequent element
  - The for loop syntax automates to facilitate looping

Aside: Lazy being virtue

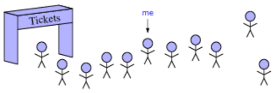
- Example: **for** index in **range(1000000)**
  - The range object is iterable
  - The syntax does not reserve memory, but produces the next element lazily as and when it is needed
- Lazy evaluation is used in many Python's libraries and produces a view of the objects underneath but does not necessarily reserves memory

Position

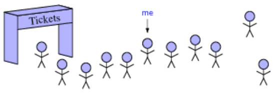
- An array index is used to get at a value
- Example: (with an STL vector)

```
Vector <int> V;
int sum = 0;
for (int i=0; i< V.size(); i++)
   sum += V[i];
return sum;
```

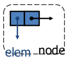
- What about linked lists?



Position



- Numeric indices do not work well with Linked Lists
  - "return an element at the sixth position": What if we insert an item at position 3. Do we now change the way we access the item?
  - Do not want to explicitly use a Node reference
- A **position** ADT describes a location
  - Example: Notion of a cursor



Positional List

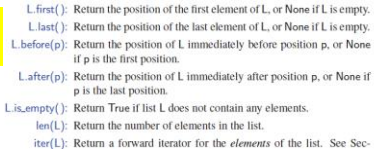
- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a positional list ADT.
  - A position acts as a marker or token within the broader positional list.
  - A position **p** is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - p.element( ): Return the element stored at position p.

Positional Accessor Operations

- Only method for a position p: p.element()
- A linked list can now support the following additional

```
cursor = data.first() # data is a linked list
while cursor is not None:
  print (cursor.element())
  cursor = data.after(cursor)
```

```
for e in data()
  print(e)
```



L.first( ): Return the position of the first element of L, or None if L is empty.

L.last( ): Return the position of the last element of L, or None if L is empty.

L.before(p): Return the position of L immediately before position p, or None if p is the first position.

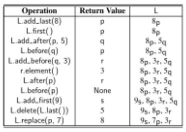
L.after(p): Return the position of L immediately after position p, or None if p is the last position.

L.is_empty( ): Return True if list L does not contain any elements.

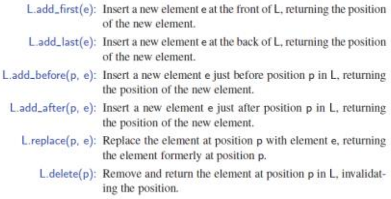
len(L): Return the number of elements in the list.

iter(L): Return a forward iterator for the *elements* of the list. See Sec-

Positional Update Operations



Operation	Return Value	L
L.add_last(8)	p	8_p
L.first( )	p	8_p
L.add_after(p, 5)	q	8_p, 5_q
L.before(q)	p	8_p, 5_q
L.add_before(q, 3)	r	8_p, 3_r, 5_q
r.element( )	3	8_p, 3_r, 5_q
L.after(p)	r	8_p, 3_r, 5_q
L.before(p)	None	8_p, 3_r, 5_q
L.add_first(9)	s	9_s, 8_p, 3_r, 5_q
L.delete(L.last( ))	5	9_s, 8_p, 3_r
L.replace(p, 7)	8	9_s, 7_p, 3_r



L.add_first(e): Insert a new element e at the front of L, returning the position of the new element.

L.add_last(e): Insert a new element e at the back of L, returning the position of the new element.

L.add_before(p, e): Insert a new element e just before position p in L, returning the position of the new element.

L.add_after(p, e): Insert a new element e just after position p in L, returning the position of the new element.

L.replace(p, e): Replace the element at position p with element e, returning the element formerly at position p.

L.delete(p): Remove and return the element at position p in L, invalidating the position.

Iterating through a Container

- Let C be a container and p be an iterator for C. In C++

```
for (p = C.begin(); p != C.end(); ++p)
    loop_body
```

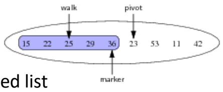
- Example: (with an STL vector)

```
typedef vector<int>::iterator Iterator;
int sum = 0;
for (Iterator p = V.begin(); p != V.end();
   ++p)
sum += *p;
return sum;
```

- Example: (with an STL vector)

```
Vector <int> V;
int sum = 0;
for (int i=0; i< V.size(); i++)
   sum += V[i];
return sum;
```

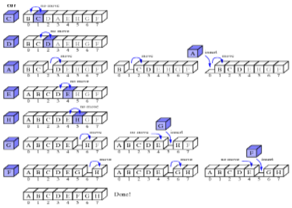
Insertion Sort



- marker points to end of current sorted list

```
def insertion_sort(L):
  """Sort PositionalList of comparable elements into nondecreasing order."""
  if len(L) > 1:                    # otherwise, no need to sort it
    marker = L.first()
    while marker != L.last():
      pivot = L.after(marker)       # next item to place
      value = pivot.element()
      if value > marker.element():  # pivot is already sorted
        marker = pivot              # pivot becomes new marker
      else:                         # must relocate pivot
        walk = marker               # find leftmost item greater than value
        while walk != L.first() and L.before(walk).element() > value:
          walk = L.before(walk)
        L.delete(pivot)
        L.add_before(walk, value)   # reinsert value before walk
```

Insertion Sort Array



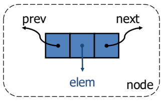
```
def insertion_sort(A):
  """Sort list of comparable elements into nondecreasing order."""
  for k in range(1, len(A)):         # from 1 to n-1
    cur = A[k]                       # current element to be inserted
    j = k                            # find correct index j for current
    while j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

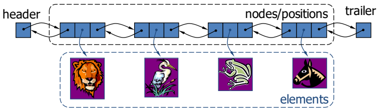
Summary: Containers and Iterators

- An iterator abstracts the process of scanning through a collection
- A container is an abstract data structure that supports element access through iterators
  - begin(): returns an iterator to the first element
  - end(): return an iterator to an imaginary position just after the last element
- An iterator (C++) behaves like a pointer to an element
  - *p: returns the element referenced by this iterator
  - ++p: advances to the next element
- Extends the concept of position by adding a traversal capability

Doubly Linked List

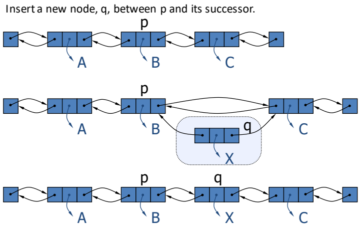
- A doubly linked list has two pointers
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





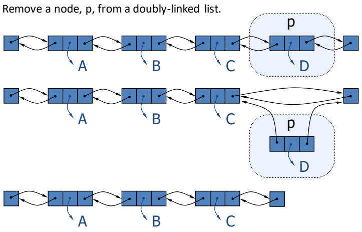
Insertion

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



Deletion

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



Performance

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