Data Structures (in C++)

- List and Iterator ADTs -

부산대학교 정보·의생명공학대학
정보컴퓨터공학부

List and Iterator ADTs

Vector

- **List or Sequence**
  - A collection *S* of *n* elements stored in a certain linear order
  - Each element *e* in *S* can be uniquely referred using an integer in the range [0,n−1]
  - **index**
    - the number of elements that are before *e* in *S*
    - A simple yet powerful notion, since it can be used to specify where to insert a new element into a list or where to remove an old element

index

- **Vector**
  - A sequence that supports access to its elements by their indices

Vector ADT

- The vectors(also called an array list) ADT supports the following fundamental functions
  - the index parameter *i* is assumed to be in the range $0 \le i \le size()-1$

at(i): Return the element of V with index i; an error condition occurs if i is out of range.

set(i,e): Replace the element at index i with e; an error condition occurs if i is out of range.

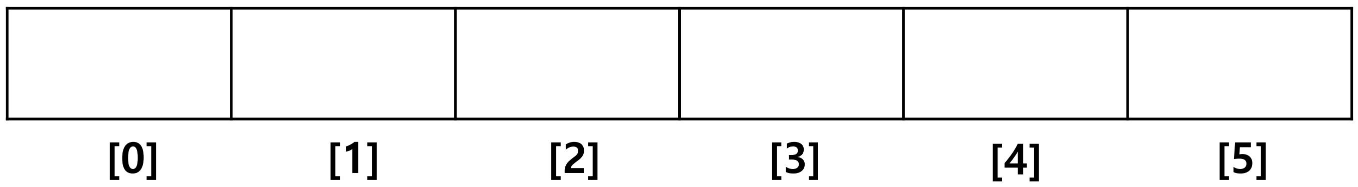
insert(i,e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

- The **index** definition offers us a way to refer to the "place" where an element is stored in a sequence
- However, the index of an element may change when the sequence is updated

Vector Example

Operation	*Output*	*V*
insert(0,7)	–	(7)
insert(0,4)	–	(4,7)
at(1)	7	(4,7)
insert(2,2)	–	(4,7,2)
at(3)	"error"	(4,7,2)
erase(1)	–	(4,2)
insert(1,5)	–	(4,5,2)
insert(1,3)	–	(4,3,5,2)
insert(4,9)	–	(4,3,5,2,9)
at(2)	5	(4,3,5,2,9)
set(3,8)	–	(4,3,5,8,9)



Simple Array-Based Implementation

- Use a fixed size array A, where $A[i]$ stores the element at index i
  - The sufficiently large size N of array A
  - the number $n < N$ of elements in the vector in a member variable.
  - *at(i)* operation just return A[i].

Operation	*Time*
size()	$O(1)$
empty()	$O(1)$
at(i)	$O(1)$
set(i,e)	$O(1)$
insert(i,e)	$O(n)$
erase(i)	$O(n)$

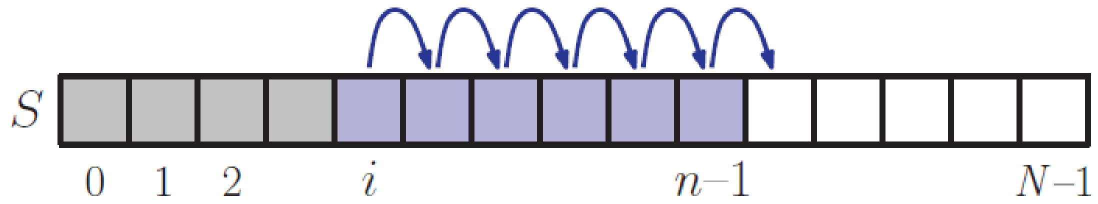
Algorithm insert(i,e):

  for $j = n-1, n-2, \ldots, i$ **do**

    $A[j+1] \leftarrow A[j]$ {make room for the new element}

  $A[i] \leftarrow e$

  $n \leftarrow n+1$



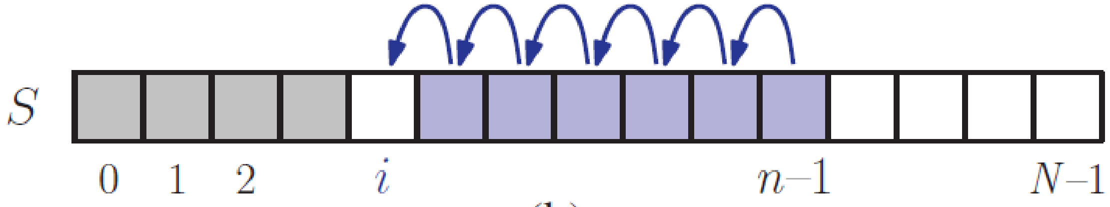
(a)

Algorithm erase(i):

  for $j = i+1, i+2, \ldots, n-1$ **do**

    $A[j-1] \leftarrow A[j]$ {fill in for the removed element}

  $n \leftarrow n-1$

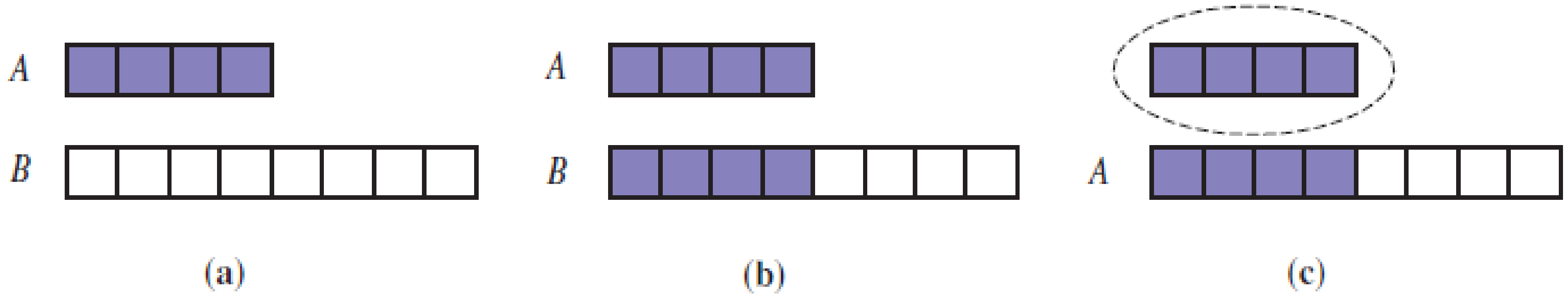


(b)

Extendable Array Implementation

- A major weakness of the simple array implementation for the vector ADT
  - It requires advance specification of a fixed capacity N

- Array replacement strategy (**Extendable Array**)
  - when an overflow occurs, that is, when n=N and function insert is called
  - 1. Allocate a new array *B* of capacity *2N*
  - 2. Copy *A[i]* to *B[i]*, for $i = 0, \ldots, N-1$
  - 3. Deallocate *A* and reassign *A* to point to the new array *B*



Extendable Array Implementation

- Vector implementation using an extendable array
  - Two means for accessing individual elements of the vector
    - ***at()*** function performs a range test
  - There is no need to explicitly define a ***set*** function
    - *v.set(i,5)* could be implemented either as *v[i] = 5* or, more safely, as *v.at(i) = 5*.

```
typedef int Elem;                               // base element type
class ArrayVector {
public:
  ArrayVector();                                // constructor
  int size() const;                             // number of elements
  bool empty() const;                           // is vector empty?
  Elem& operator[](int i);                      // element at index
  Elem& at(int i) throw(IndexOutOfBounds);      // element at index
  void erase(int i);                            // remove element at index
  void insert(int i, const Elem& e);            // insert element at index
  void reserve(int N);                          // reserve at least N spots
  // ... (housekeeping functions omitted)
private:
  int capacity;                                 // current array size
  int n;                                        // number of elements in vector
  Elem* A;                                      // array storing the elements
};
```

```
ArrayVector::ArrayVector()                      // constructor
  : capacity(0), n(0), A(NULL) { }

int ArrayVector::size() const                   // number of elements
  { return n; }

bool ArrayVector::empty() const                 // is vector empty?
  { return size() == 0; }

Elem& ArrayVector::operator[](int i)            // element at index
  { return A[i]; }
                                                // element at index (safe)
Elem& ArrayVector::at(int i) throw(IndexOutOfBounds) {
  if (i < 0 || i >= n)
    throw IndexOutOfBounds("illegal index in function at()");
  return A[i];
}
```

Extendable Array Implementation

- Vector implementation using an extendable array

```
void ArrayVector::erase(int i) {        // remove element at index
  for (int j = i+1; j < n; j++)
    A[j - 1] = A[j];
  n--;
}
```

```
void ArrayVector::reserve(int N) {              // reserve at least N spots
  if (capacity >= N) return;                    // already big enough
  Elem* B = new Elem[N];                        // allocate bigger array
  for (int j = 0; j < n; j++)                   // copy contents to new array
    B[j] = A[j];
  if (A != NULL) delete [] A;                   // discard old array
  A = B;                                        // make B the new array
  capacity = N;                                 // set new capacity
}
void ArrayVector::insert(int i, const Elem& e) {
  if (n >= capacity)                            // overflow?
    reserve(max(1, 2 * capacity));              // double array size
  for (int j = n - 1; j >= i; j--)              // shift elements up
    A[j+1] = A[j];
  A[i] = e;                                     // put in empty slot
  n++;                                          // one more element
}
```

STL Vectors

- The class vector is perhaps the most basic example of an STL container class
- Like standard C++ arrays, but they provide many additional features.
  - Elements can also be accessed by a member function called *at*
  - STL vectors can be dynamically resized
  - When an STL vector of class objects is destroyed, it automatically invokes the destructor for each of its elements
  - A number of useful functions

```
#include <vector>                  // provides definition of vector
using std::vector;                 // make vector accessible

vector<int> myVector(100);         // a vector with 100 integers
```

base type

STL Vectors

- Member functions of STL vector

vector(n):	Construct a vector with space for n elements; if no argument is given, create an empty vector.
size():	Return the number of elements in V.
empty():	Return true if V is empty and false otherwise.
resize(n):	Resize V, so that it has space for n elements.
reserve(n):	Request that the allocated storage space be large enough to hold n elements.
operator$[i]$:	Return a reference to the ith element of V.
at(i):	Same as $V[i]$, but throw an out_of_range exception if i is out of bounds, that is, if $i < 0$ or $i \geq V$.size().
front():	Return a reference to the first element of V.
back():	Return a reference to the last element of V.
push_back(e):	Append a copy of the element e to the end of V, thus increasing its size by one.
pop_back():	Remove the last element of V, thus reducing its size by one.

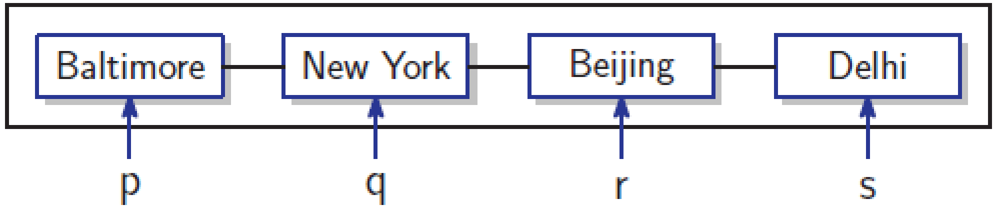
Node-Based Operations and Iterators

- If we have a list *L* implemented with a (singly or doubly) linked list
  - more natural and efficient to use a node instead of an index
- Node-based operations
  - Speedups over index-based functions
  - Finding the index of an element in a linked list requires searching through the list incrementally

- **Position**
  - data type that abstracts the notion of the relative **position** or place of an element within a list

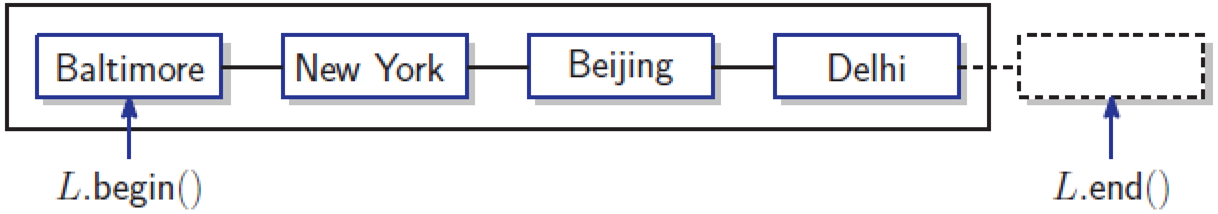
element(): Return a reference to the element stored at this position.

  - given a position variable p, the associated element can be accessed by *p
  - A position q, which is associated with some element e in a container, does not change, even if the index of e changes in the container



Iterators

- An iterator is an extension of a position
  - it also provides the ability to navigate forwards (and possibly backwards) through the container
- There are a number of ways in which to define an ADT for an iterator object
  - p.next()
  - ++p
- Two special iterator values, begin and end



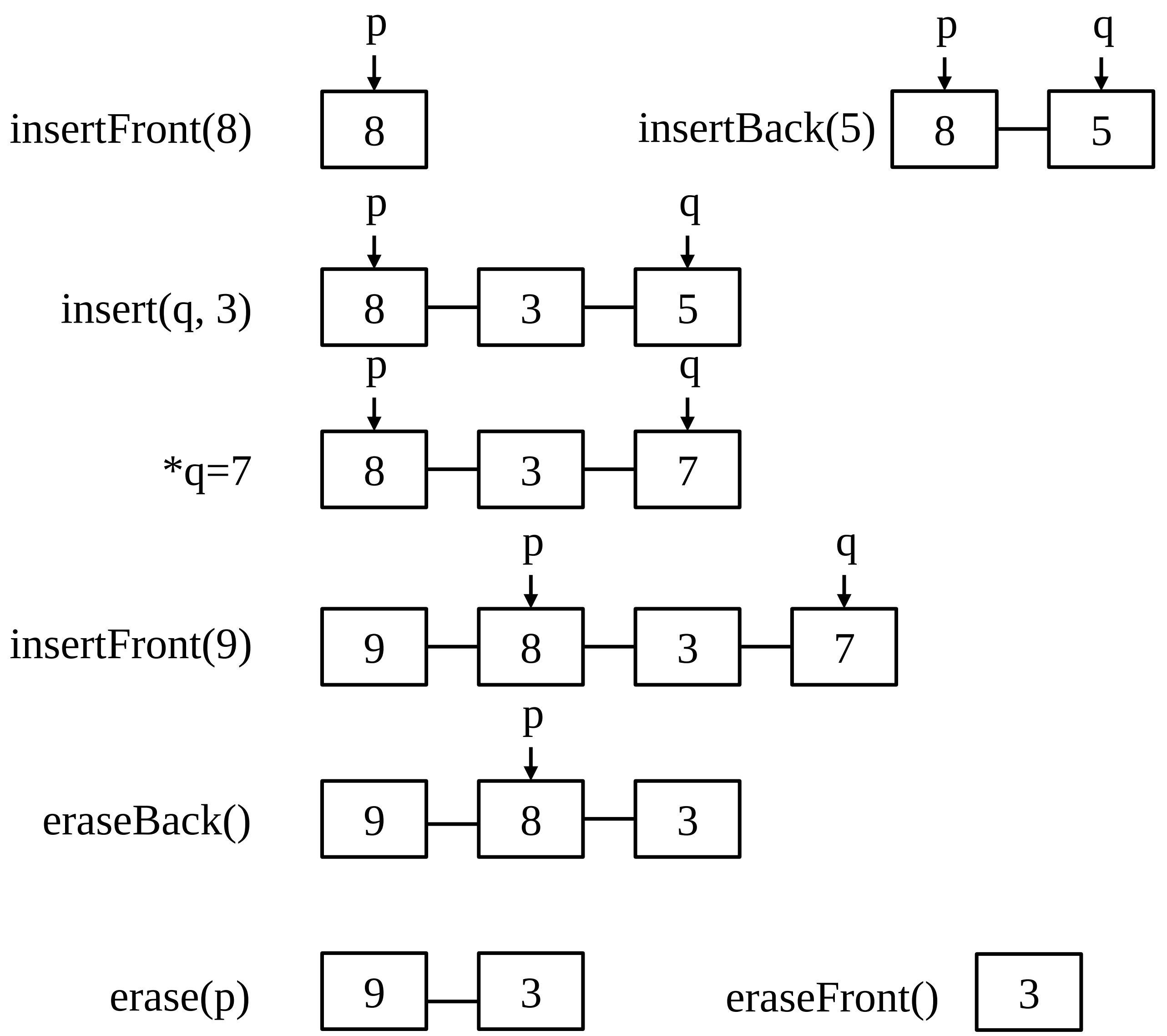
List ADT

- List ADT supports the following functions

begin():	Return an iterator referring to the first element of L; same as end() if L is empty.
end():	Return an iterator referring to an imaginary element just after the last element of L.
insertFront(e):	Insert a new element e into L as the first element. ← **insert(L.begin(),e)**
insertBack(e):	Insert a new element e into L as the last element. ← **insert(L.end(),e)**
insert(p,e):	Insert a new element e into L before position p in L.
eraseFront():	Remove the first element of L.
eraseBack():	Remove the last element of L.
erase(p):	Remove from L the element at position p; invalidates p as a position.

List ADT

Operation	Output	L
insertFront(8)	–	(8)
$p =$ begin()	$p : (8)$	(8)
insertBack(5)	–	(8,5)
$q = p;\ \ ++q$	$q : (5)$	(8,5)
$p ==$ begin()	*true*	(8,5)
insert(q,3)	–	(8,3,5)
$*q = 7$	–	(8,3,7)
insertFront(9)	–	(9,8,3,7)
eraseBack()	–	(9,8,3)
erase(p)	–	(9,3)
eraseFront()	–	(3)



List Implementation: Doubly Linked List

```
struct Node {                          // a node of the list
  Elem elem;                           // element value
  Node* prev;                          // previous in list
  Node* next;                          // next in list
};

class Iterator {                       // an iterator for the list
public:
  Elem& operator*();                   // reference to the element
  bool operator==(const Iterator& p) const; // compare positions
  bool operator!=(const Iterator& p) const;
  Iterator& operator++();              // move to next position
  Iterator& operator--();              // move to previous position
  friend class NodeList;               // give NodeList access
private:
  Node* v;                             // pointer to the node
  Iterator(Node* u);                   // create from node
};
```

nested class Iterator

```
NodeList::Iterator::Iterator(Node* u)          // constructor from Node*
  { v = u; }

Elem& NodeList::Iterator::operator*()          // reference to the element
  { return v->elem; }
                                               // compare positions
bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }
                                               // move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v->next; return *this; }
                                               // move to previous position
NodeList::Iterator& NodeList::Iterator::operator--()
  { v = v->prev; return *this; }
```

This makes it possible to use the result of the increment operation, as in "q = ++p

List Implementation: Doubly Linked List

```
typedef int Elem;                          // list base element type
class NodeList {                           // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                              // default constructor
  int size() const;                        // list size
  bool empty() const;                      // is the list empty?
  Iterator begin() const;                  // beginning position
  Iterator end() const;                    // (just beyond) last position
  void insertFront(const Elem& e);         // insert at front
  void insertBack(const Elem& e);          // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();                       // remove first
  void eraseBack();                        // remove last
  void erase(const Iterator& p);           // remove p
  // housekeeping functions omitted...
private:                                   // data members
  int     n;                               // number of items
  Node*   header;                          // head-of-list sentinel
  Node*   trailer;                         // tail-of-list sentinel
};
```

List Implementation: Doubly Linked List

```
NodeList::NodeList() {                         // constructor
  n = 0;                                       // initially empty
  header = new Node;                           // create sentinels
  trailer = new Node;
  header->next = trailer;                      // have them point to each other
  trailer->prev = header;
}

int NodeList::size() const                     // list size
  { return n; }

bool NodeList::empty() const                   // is the list empty?
  { return (n == 0); }

NodeList::Iterator NodeList::begin() const     // begin position is first item
  { return Iterator(header->next); }

NodeList::Iterator NodeList::end() const       // end position is just beyond last
  { return Iterator(trailer); }
```

List Implementation: Doubly Linked List

```
                                                    // insert e before p
void NodeList::insert(const NodeList::Iterator& p, const Elem& e) {
  Node* w = p.v;                                    // p's node
  Node* u = w->prev;                                // p's predecessor
  Node* v = new Node;                               // new node to insert
  v->elem = e;
  v->next = w; w->prev = v;                         // link in v before w
  v->prev = u; u->next = v;                         // link in v after u
  n++;
}

void NodeList::insertFront(const Elem& e)           // insert at front
  { insert(begin(), e); }

void NodeList::insertBack(const Elem& e)            // insert at rear
  { insert(end(), e); }
```

```
void NodeList::erase(const Iterator& p) {           // remove p
  Node* v = p.v;                                    // node to remove
  Node* w = v->next;                                // successor
  Node* u = v->prev;                                // predecessor
  u->next = w; w->prev = u;                         // unlink p
  delete v;                                         // delete this node
  n--;                                              // one fewer element
}

void NodeList::eraseFront()                         // remove first
  { erase(begin()); }

void NodeList::eraseBack()                          // remove last
  { erase(--end()); }
```

why we chose to define the iterator function end to return an imaginary position that lies just beyond the end of the list?

STL List

- The STL list is implemented as a doubly linked list.

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

list(n): Construct a list with n elements; if no argument list is given, an empty list is created.

size(): Return the number of elements in L.

empty(): Return true if L is empty and false otherwise.

front(): Return a reference to the first element of L.

back(): Return a reference to the last element of L.

push_front(e): Insert a copy of e at the beginning of L. ← **insertFront**

push_back(e): Insert a copy of e at the end of L. ← **insertBack**

pop_front(): Remove the fist element of L. ← **eraseFront**

pop_back(): Remove the last element of L. ← **eraseBack**

STL Containers and Iterators

- The STL provides a variety of different container classes
  - A container is a data structure that stores a collection of elements
- STL iterators provide a relatively uniform method for accessing and enumerating the elements stored in containers.

STL Container	*Description*
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

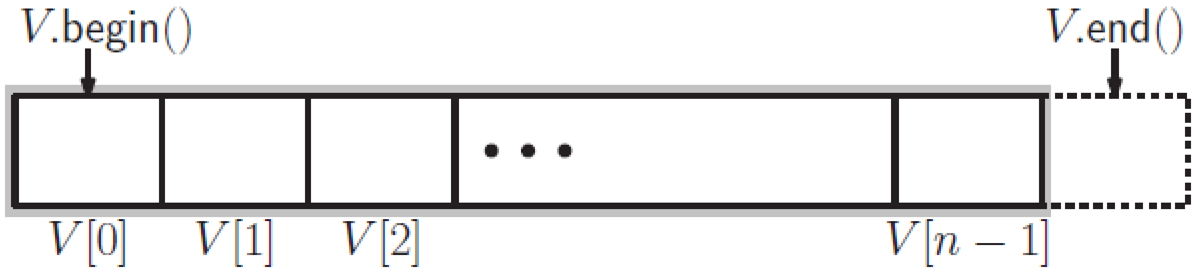
```
int vectorSum1(const vector<int>& V) {
  int sum = 0;
  for (int i = 0; i < V.size(); i++)
    sum += V[i];
  return sum;
}
```

Unfortunately, this method would not be applicable to other types of containers

because it relies on the fact that the elements of a vector can be accessed efficiently through indexing.

STL Iterators

- Every STL container class defines a special associated class called an iterator
- Iterator is an object that specifies a position within a container
- If p is an iterator
  - *p yields a reference to the associated element.
  - either ++p or p++ advances p to point to the next element of the container




```
int vectorSum2(vector<int> V) {
  typedef vector<int>::iterator Iterator;        // iterator type
  int sum = 0;
  for (Iterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

This approach can be applied to any STL container class, not just vectors

STL Iterators

- Const Iterators
  - it is possible to read the values of the container by dereferencing the iterator
  - it is not possible to modify the container's values

```
int vectorSum3(const vector<int>& V) {
  typedef vector<int>::const_iterator ConstIterator;  // iterator type
  int sum = 0;
  for (ConstIterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

STL Iterators

- STL Iterator-Based Container Functions
  - The member functions of the STL vector class that use iterators as arguments.
  - The above functions are also defined for the STL list and the STL deque
    - These three STL containers (vector, list, and deque) are called sequence containers

vector(p,q): Construct a vector by iterating between p and q, copying each of these elements into the new vector.

assign(p,q): Delete the contents of V, and assigns its new contents by iterating between p and q and copying each of these elements into V.

insert(p,e): Insert a copy of e just prior to the position given by iterator p and shifts the subsequent elements one position to the right.

erase(p): Remove and destroy the element of V at the position given by p and shifts the subsequent elements one position to the left.

erase(p,q): Iterate between p and q, removing and destroying all these elements and shifting subsequent elements to the left to fill the gap.

clear(): Delete all these elements of V.

the iterator range is understood to start with p and end just prior to q, [p,q)

Note that the vector member functions insert and erase move elements around in the vector. They can be quite slow.

STL Iterators

- STL Vectors and Algorithms
  - the STL also provides a number of algorithms that operate on containers
  - #include <algorithm>

sort(p,q): Sort the elements in the range from p to q in ascending order. It is assumed that less-than operator ("<") is defined for the base type.

random_shuffle(p,q): Rearrange the elements in the range from p to q in random order.

reverse(p,q): Reverse the elements in the range from p to q.

find(p,q,e): Return an iterator to the first element in the range from p to 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 from p to q.

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

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

For example, to sort an entire vector V, we would use sort(V.begin(),V.end()).

STL Iterators

- STL Vectors and Algorithms

```
#include <cstdlib>                          // provides EXIT_SUCCESS
#include <iostream>                         // I/O definitions
#include <vector>                           // provides vector
#include <algorithm>                        // for sort, random_shuffle

using namespace std;                        // make std:: accessible

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

  char b[] = {'b', 'r', 'a', 'v', 'o'};
  vector<char> w(b, b + 5);                 // w: b r a v o
  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
  cout << endl;
  return EXIT_SUCCESS;
}
```

random shuffle to permute the elements of the vector randomly

Sequence

- Sequence
  - An abstract data type that generalizes the vector and list ADTs
  - A sequence is an ADT that supports all the functions of the list ADT
  - It also provides functions for accessing elements by their index, as we did in the vector ADT
  - Provides the following two “bridging” functions

atIndex(i): Return the position of the element at index i.

indexOf(p): Return the index of the element at position p.

Sequence Implementation: Doubly Linked List

- Sequence

```
class NodeSequence : public NodeList {
public:
  Iterator atIndex(int i) const;                // get position from index
  int indexOf(const Iterator& p) const;         // get index from position
};

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

Sequence Implementation: Doubly Linked List vs. Array

Operations	***Circular Array***	***List***
size, empty	$O(1)$	$O(1)$
atIndex, indexOf	$O(1)$	$O(n)$
begin, end	$O(1)$	$O(1)$
$*p$, $++p$, $--p$	$O(1)$	$O(1)$
insertFront, insertBack	$O(1)$	$O(1)$
insert, erase	$O(n)$	$O(1)$