CSE221 Data Structures
Lecture 11: Vectors and Lists

Antoine Vigneron

antoine@unist.ac.kr

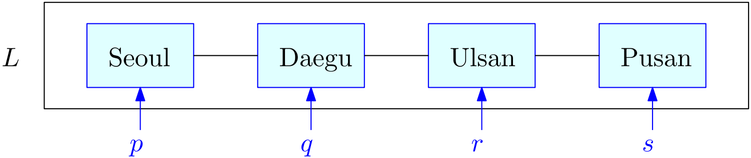
Ulsan National Institute of Science and Technology

October 25, 2021

Introduction

- We are now using zoom for lectures. Please use a zoom name that includes your student ID number so that I can check attendance automatically.
- I updated attendance records up to the midterm week.
- Assignment 2 is due on Thursday. It will be graded by Hyeyun Yang (`gm1225@unist.ac.kr`).
- Please follow our academic integrity rules. I use a plagiarism checker.
- I will grade the midterm this week.
- Reference for this lecture: Textbook Section 6-6.2.3

Vectors



- A *list* or *sequence* is a collection of elements arranged in linear order.
- The *index* i of an element e is the number of elements that are before e in S.
- Its *previous* element has index $i-1$, and its *next* element has index $i+1$.
- A sequence that supports access to its elements by their indices is called a *vector*.

The Vector ADT

Definition

A vector ADT supports the following operations:

- **at**(i): Return the element of V with index i.
- **set**(i, e): Replace the element at index i with e.
- **insert**(i, e): Insert a new element e into V to have index i.
- **erase**(i): Remove from V the element at index i.

In all of these operations, an error occurs if i is out of range.

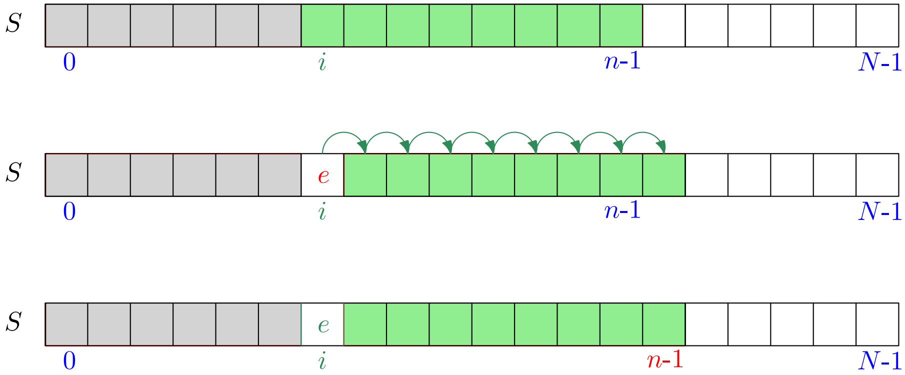
- An array is a natural implementation of a vector, but there are other possibilities.

The Vector ADT

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)

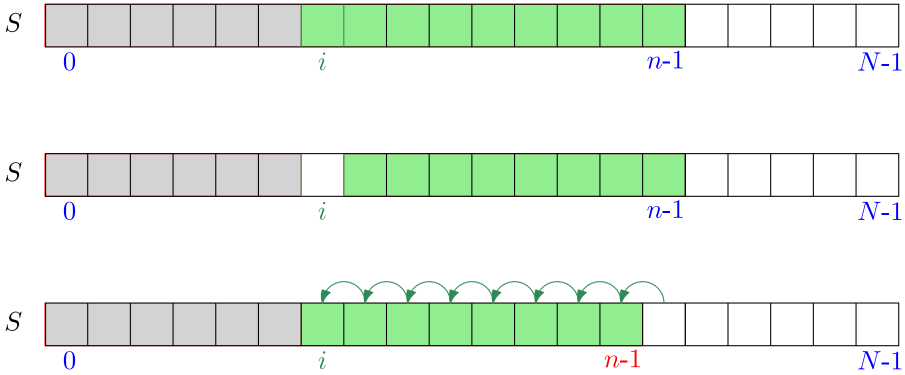
- Operations performed on an initially empty vector V.

Simple Array-Based Implementation



- performing the operation: insert(i,e)

Simple Array-Based Implementation



- performing the operation: erase(i)

Simple Array-Based Implementation

Pseudocode

```
procedure INSERT(i, e)
    for j ← n − 1, i do
        A[j + 1] ← A[j]
    A[i] ← e
    n ← n + 1
```

```
procedure ERASE(i)
    for j ← i + 1, n − 1 do
        A[j − 1] ← A[j]
    n ← n − 1
```

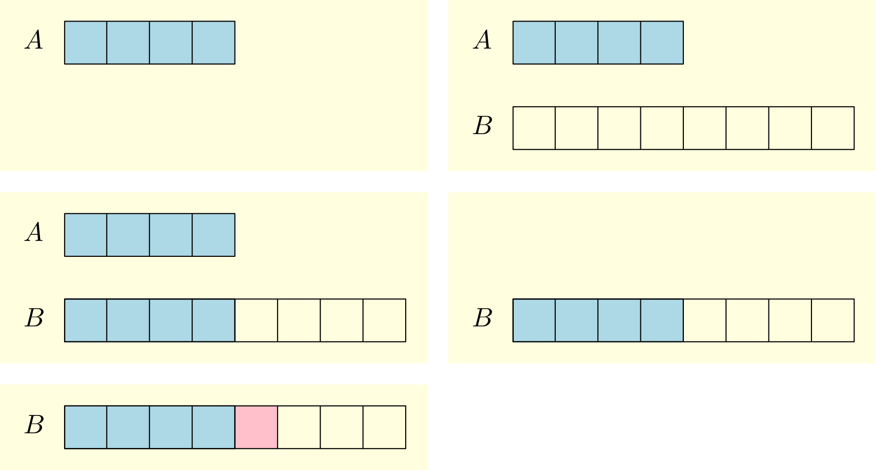
Performance

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)

- The space usage is $O(N)$.
- Inserting or deleting at the *end* of the array takes $O(1)$ time.

Extendable Array Implementation

- Suppose that $n = N$ and we want to insert a new element.
- Problem: the array is full. Solution:



Extendable Array Implementation

- When we call insert() and $n = N$, we have an *overflow*.
- As shown in previous slide, we do the following:

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.
4. Perform the insertion.

Extendable Array Implementation

```
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);                       // element at index (safe)
  void erase(int i);                     // remove element at index
  void insert(int i, const Elem& e);             // insert element
  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
};
```

Extendable Array Implementation

```
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; }
```

Extendable Array Implementation

```
Elem& ArrayVector::operator[ ](int i) // element at index
  { return A[i]; }
```

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

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

Extendable Array Implementation

```
                                    // reserve at least N spots
void ArrayVector::reserve(int N) {
  if (capacity >= N) return;              // already big enough
  Elem* B = new Elem[N];               // allocate bigger array
  for (int j = 0; j < n; j++)             // copy 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
}
```

Extendable Array Implementation

```
                                            // insert element at index
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
}
```

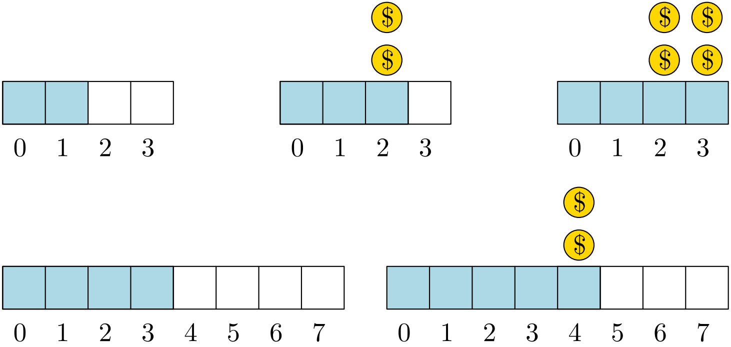
Analysis

- Inserting or erasing an element at an arbitrary index i takes $O(n)$ time, which is not very good.
- On the other hand, erasing the last element, i.e. erase(n-1) takes $O(1)$ time.
- A *push* operation is an insertion at the end of the vector, i.e. insert($n-1$,e) for some element e.
- A push operation takes $O(1)$ time if we don't need to resize, and $O(n)$ time if we do.
- Intuitively, there are not many resizing operations.
- How can we analyze it? We use *amortization*.
- We show that a sequence of push operations takes time proportional to the number of these operations.

Analysis

Proposition

Let V be a vector implemented by means of an extendable array A, as described above. The total time to perform a series of n push operations in V, starting from V being empty and A having size $N = 1$, is $O(n)$.



Analysis

- We use a *charging* argument.
- We charge 3$ for each push operation.
- \$1 pays for a push.
- $\$k$ pay for growing the array from size k to $2k$.
- Suppose an overflow occurs at $n = 2^i$.
- Then we saved $\$2^i$ between push operations number 2^{i-1} and $2^i - 1$.
- So we have enough coins to pay for growing the array to size 2^{i+1}.
- In the end, the $3n$ coins we collected are sufficient to pay for all the operations.
- So the overall running time is $O(n)$.

STL Vectors

- A *container* is a data structure that stores a collection of objects.
- The STL vector is one of the most basic containers.

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

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

- The *base type* is the type of the elements stored in the vector.
- So in the example above, they are integers.

- Elements can be accessed with the index operator [].
- For instance, myVector[15] is the element at index 15.
- We can also use the member function at(i), which checks whether i is in range.

STL Vectors

- As opposed to C++ arrays:
  - STL vectors can be dynamically resized.
  - New elements can be appended or removed from the end of the vector in $O(1)$ time.
  - When an STL vector of class objects is destroyed, it automatically invokes the destructor for each of its elements.

Member Functions

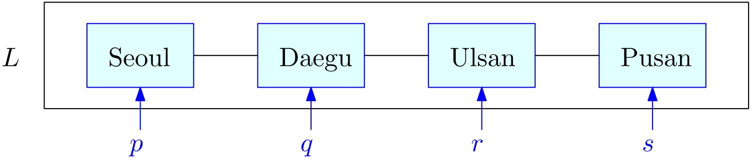
- **vector**(n): Construct a vector with space for n elements; if no argument is given, create an empty vector.
- **size**(): Return the number of elements in V.
- **empty**(): Return true if V is empty and false otherwise.
- **resize**(n): Resize V, so that it has space for n elements.
- **reserve**(n): Request that the allocated storage space be large enough to hold n elements.

STL Vectors

Member Functions

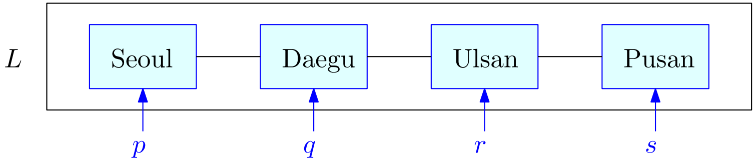
- **operator**[i]: Return a reference to the ith element of V.
- **at**(i): Same as V [i], but throw an out of range exception if i is out of bounds, that is, if $i < 0$ or $i \geqslant$ V.size().
- **front**(): Return a reference to the first element of V.
- **back**(): Return a reference to the last 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.

Lists



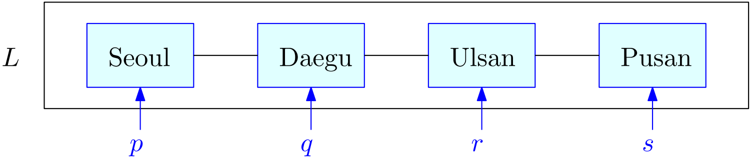
- We refer to an individual element contained in an array or a vector using its *index* i.
- A *list* is a container that uses *nodes* to refer to its elements.

Lists



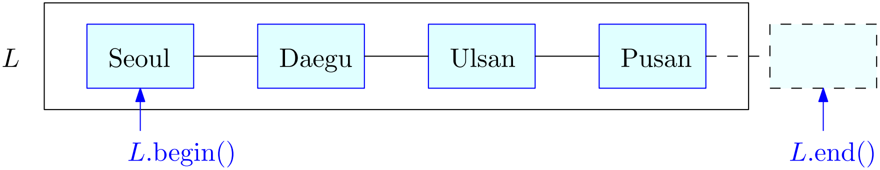
- A *position* is an abstract data type associated with a particular container, and supporting the member function **element**(), which returns a reference to the element stored at this position.
- For instance, in the figure above, **element**(r) returns a reference to "Ulsan".
- Instead of writing **element**(r), we can write $*r$ as the $*$ has been overloaded.

Lists



- We may want to be able to traverse the list.
- For instance, we would like the operation p++ to advance p to the next position, i.e. "Daegu".
- We can also move backwards, i.e. $r--$ moves to "Daegu".
- An *iterator* is an extension of a position that allows us to do it.

Lists



- We also have two special iterators, begin() and end().
- In order to enumerate all the elements of the container, we can start from $p = L.\text{begin}()$, and perform $p{+}{+}$ until p is equal to $L.\text{end}()$.
- We can access the current element using $*p$.

The List ADT

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.
- **insertBack**(e): Insert a new element e into L as the last element.
- **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.

The 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)

- Operations performed on an initially empty list L.

Doubly Linked List Implementation: Nodes

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

Doubly Linked List Implementation: Iterators

```
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
};
```

Doubly Linked List Implementation: Iterators

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

```
                                    // reference to the element
Elem& NodeList::Iterator::operator*()
  { 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; }
```

Doubly Linked List Implementation: Iterators

```
                                   // 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; }
```

- The increment and decrement operators not only update the position, but they also return a reference to the updated position.
- This makes it possible to use the result of the increment operation, as in "q = ++p."

Doubly Linked List Implementation

```
typedef int Elem;
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
                                                // insert e before p
  void insert(const Iterator& p, const Elem& e);
```

Doubly Linked List Implementation

```
  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
};
```

Doubly Linked List Implementation

```
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); }
```

Doubly Linked List Implementation

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

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

```cpp
void NodeList::insert(const NodeList::Iterator& p,
                      const Elem& e) {      // insert e before p
  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 v before w
  v->prev = u; u->next = v;          // link v after u
  n++;
}
```

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

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

Doubly Linked List Implementation

```
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()); }
```