CSE221 Data Structures
Lecture 12: Lists and Sequences

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

October 27, 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. They can be found in the portal under E-attendance.
- Assignment 2 is due tomorrow. 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 6.2.4-6.4

STL Lists

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

- When the base type of an STL vector is class object, all copying of elements (for example, in push_back) is performed by invoking the base class's copy constructor.
- Whenever elements are destroyed (for example, by invoking the destroyer or the pop_back member function) the class's destructor is invoked on each deleted element

STL Lists

- **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.
- **push_back**(e): Insert a copy of e at the end of L.
- **pop_front**(): Remove the fist element of L.
- **pop_back**(): Remove the last element of L.

STL Containers

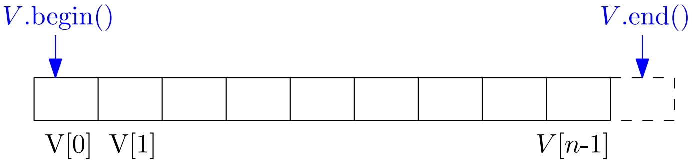
STL Container	Description
`vector`	Vector
`deque`	Double ended queue
`list`	List
`stack`	Last-in, first-out stack
`queue`	First-in, first-out queue
`priority_queue`	Priority queue
`set` (and `multiset`)	Set (and multiset)
`map` (and `multimap`)	Map (and multi-key map)

STL Iterators

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

- This approach works because we can directly access elements of a vector using the [] operator.
- What can we do with other types of containers?

STL Iterators




```
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 applies to *any* STL container.

STL Iterators

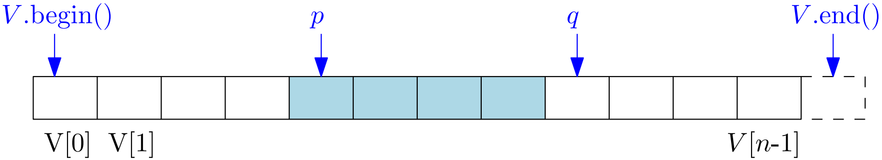
- In `vectorSum2`, we passed the vector V *by value*.
- Problem: This is very inefficient as it makes a copy of the whole vector V.
- We did this because the Iterator allows us to modify V, so it would generate an error message if we passed V by reference.

STL Iterators

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

- Solution: We use a *const iterator*.
- Such an iterator allows us to read a value, but not to modify the container.

STL Iterator-Based Container Functions



- Let p and q be iterators, V be a vector and e an element.
- **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.

STL Iterator-Based Container Functions

- **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 the elements of V.
- These functions are also defined for STL lists and deques.
- As they are implemented as doubly linked list, there is no need to shift when performing `insert()` or `erase()`.
- Vectors, lists and deques are called *sequence containers*.

STL Iterator-Based Container Functions

```
list<int> L;                          // an STL list of integers
// ...
vector<int> V(L.begin(), L.end());
                               // initialize V to be a copy of L
```

- The iterators p and q do not need to be drawn from the same type of container as V.
- It suffices that they have the same base type (here, `int`).
- *Pointer arithmetics:* $A + i$ points to $A[i]$. So we can do the following:

```
int A[] = {2, 5, -3, 8, 6};  // a C++ array of 5 integers
vector<int> V(A, A+5);                // V = (2, 5, -3, 8, 6)
```

STL Vectors and Algorithms

- **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 to the elements in the range from p to q.

Example

```
#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
```

Example

```
  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;
}
```

Implementing a Sequence with a Doubly Linked List

- The *sequence* ADT supports all the functions of the list ADT, and in addition:
- **atIndex**(i): Return the position of the element at index i.
- **indexOf**(p): Return the index of the element at position p.
- We define the `NodeSequence` class by extending `NodeList`:

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

Implementing a Sequence with a Doubly Linked List

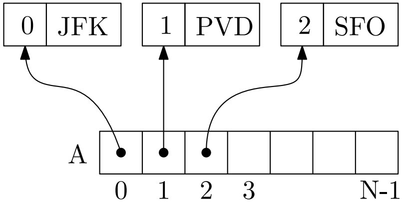
```
                                             // get position from index
NodeSequence::Iterator NodeSequence::atIndex(int i) const {
  Iterator p = begin();
  for (int j = 0; j < i; j++) ++p;
  return p;
}
```

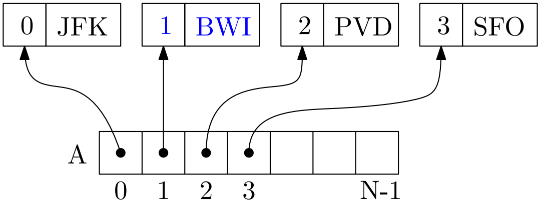
```
                                             // get index from position
int NodeSequence::indexOf(const Iterator& p) const {
  Iterator q = begin();
  int j = 0;
  while (q != p) {                                    // until finding p
    ++q; ++j;                                   // advance and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

Implementing a Sequence with a Doubly Linked List

- The list member functions still take $O(1)$ time.
- But the two new functions take $O(n)$ time.

Implementing a Sequence with an Array





Analysis

Operation	Circular array	List
`size`, `empty`	$O(1)$	$O(1)$
`atIndex`, `indexOf`	$O(1)$	$O(n)$
`begin`, `end`	$O(1)$	$O(1)$
`*p, ++p, --p`	$O(1)$	$O(1)$
`inserFront`, `insertBack`	$O(1)$	$O(1)$
`insert`, `erase`	$O(n)$	$O(1)$

- Space usage is $O(n)$ with a list, and $O(N)$ with an array, where N is the size of the array.

Bubble-Sort

pass	swaps	sequence
		(5, 7, 2, 6, 9, 3)
1st	$7 \leftrightarrow 2$ $7 \leftrightarrow 6$ $9 \leftrightarrow 3$	(5, 2, 6, 7, 3, 9)
2nd	$5 \leftrightarrow 2$ $7 \leftrightarrow 3$	(2, 5, 6, 3, 7, 9)
3rd	$6 \leftrightarrow 3$	(2, 5, 3, 6, 7, 9)
4th	$5 \leftrightarrow 3$	(2, 3, 5, 6, 7, 9)

- The *bubble-sort* algorithm sorts a sequence by performing a series of *passes*.
- Each pass proceeds from left to right, and any two consecutive elements that are in the wrong relative order are swapped.

Bubble-Sort

- At the end of the ith pass, the right-most i elements of the sequence (that is, those at indices from $n-1$ down to $n-i$ are in final position.
- It shows that we only need one pass, and that each pass i is limited to the first $n-i+1$ elements of the sequence.
- Analysis: the running time is proportional to

$$\begin{aligned}\sum_{i=1}^{n} n-i+1 &= n+(n-1)+\cdots+2+1 \\ &= \frac{n(n+1)}{2}.\end{aligned}$$

This is a polynomial of degree 2, so the running time is $O(n^2)$.

Bubble-Sort Implementation Based on Indices

```
void bubbleSort1(Sequence& S) {    // bubble-sort by indices
  int n = S.size();
  for (int i = 0; i < n; i++) {                    // i-th pass
    for (int j = 1; j < n-i; j++) {
      Sequence::Iterator prec = S.atIndex(j-1);
                                                 // predecessor
      Sequence::Iterator succ = S.atIndex(j);  // successor
      if (*prec > *succ) {          // swap if out of order
        int tmp = *prec; *prec = *succ; *succ = tmp;
      }
    }
  }
}
```

Node-Based Implementation of Bubble-Sort

```
                                // bubble-sort by positions
void bubbleSort2(Sequence& S) {
  int n = S.size();
  for (int i = 0; i < n; i++) {                 // i-th pass
    Sequence::Iterator prec = S.begin();    // predecessor
    for (int j = 1; j < n-i; j++) {
      Sequence::Iterator succ = prec;
      ++succ;                                   // successor
      if (*prec > *succ) { // swap if out of order
        int tmp = *prec; *prec = *succ; *succ = tmp;
      }
      ++prec;                          // advance predecessor
    }
  }
}
```

Analysis

- The node-based implementation `bubbleSort1` runs in $O(n^2)$ time for an array-based sequence, and $O(n^3)$ for a node-based sequence.
- On the other hand, the second implementation `bubbleSort2` runs in $O(n^2)$ time regardless of the sequence implementation, as the $++$ operator always runs in $O(1)$ time.