CS F211: DATA STRUCTURES & ALGORITHMS (2ND SEMESTER 2024-25)

Vectors, Lists, Amortization, & Sequences

Chittaranjan Hota, PhD
Senior Professor, Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

VECTOR ADT (STL IN C++)

```
#include <vector>
std::vector <int> v;
v.insert (0,5)
v.insert (0,3)
v.insert (1,4)
v.insert (3,6)
v.at(2)=8
v.erase(1)
at(3)
```

```
v.push_back(35);
v.pop_back();
```

SIMPLE ARRAY-BASED IMPLEMENTATION

Use an array ***A*** of size ***N***

A variable ***n*** keeps track of the size of the array list (number of elements stored)

How will you implement?

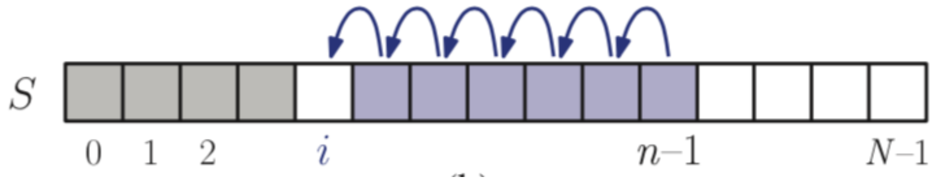
at (i)

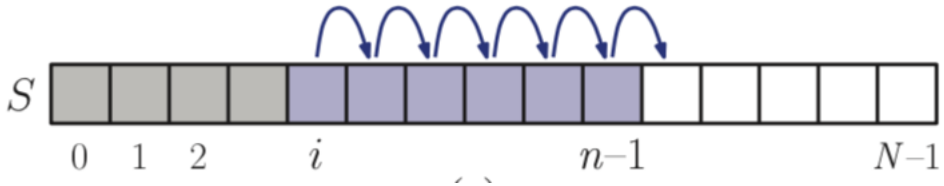
set (i, o)

insert (i, o

erase (i)

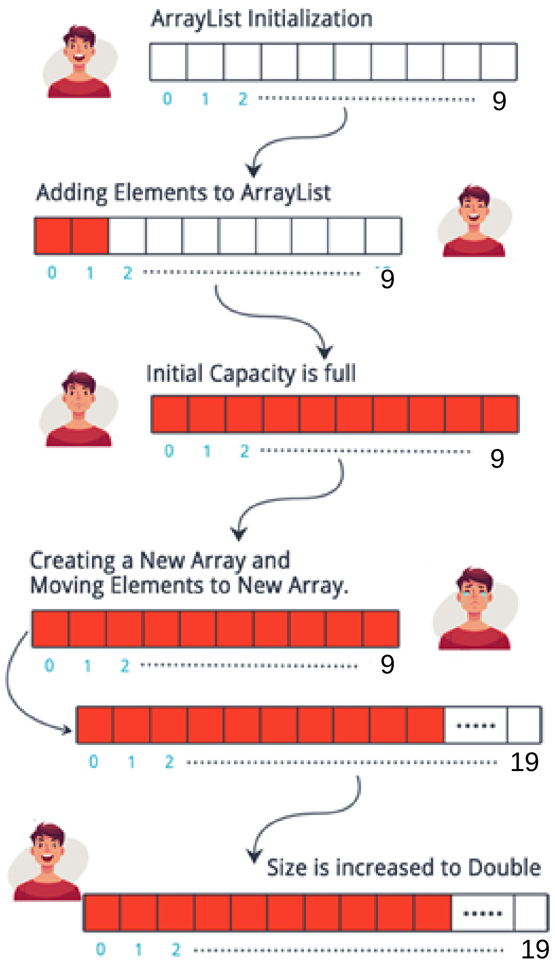
Algorithm insert(*i*,*o*): **Algorithm** erase(*i*):





What would be the performance of a vector realized by an array?

```
1 : Insert
2 : Erase
3 : Get element at index i
4 : Get size
5 : Check if vector is empty
6 : Exit
1
Enter index and element :
0 10
3
Enter index :
0
10
1
Enter index and element :
1 20
4
Getting size
2
5
Vector is not empty
2
Enter index :
1
4
Getting size
1
```



COMPARISON OF STRATEGIES: AMORTIZATION (A DESIGN PATTERN)

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n insert(o) operations.
- We assume that we start with an empty stack represented by an array of size 1
- We call **amortized time** of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$

INCREMENTAL

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc = n + c(1 + 2 + 3 + \ldots + k) = n + ck(k + 1)/2$$

Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$

The amortized time of an insert operation is $O(n)$

(Increase the size by a constant c)

Doubling Strategy

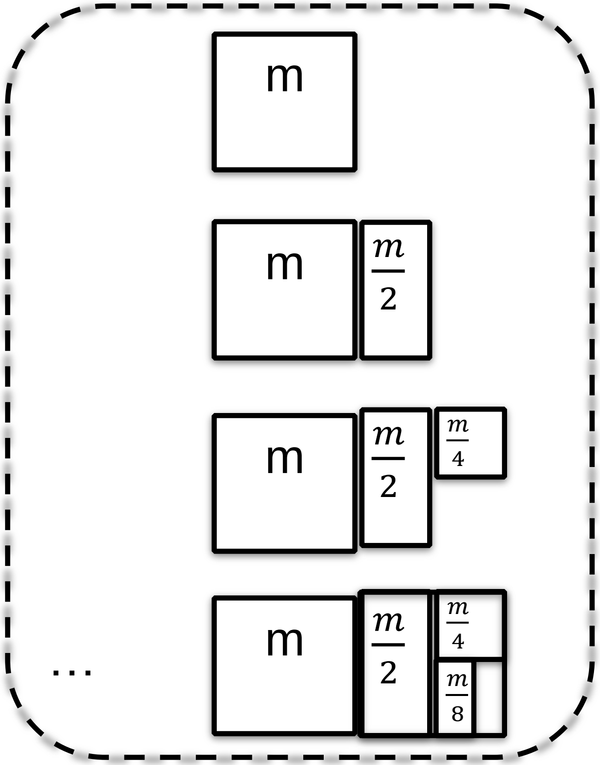
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k$$
$$= n + 2^{k+1} - 1 = 3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an insert operation is $O(1)$

ACCOUNTING METHOD

- "Amortize" is a fancy verb used in finance that refers to paying off the cost of something gradually. With dynamic arrays, every expensive append where we have to grow the array "buys" us many cheap appends in the future. Conceptually, we can spread the cost of the expensive append over all those cheap appends.
- The cost of doing m appends is *m* (since we're appending every item), *plus* the cost of any doubling when the dynamic array needs to grow. How much does the doubling cost?
- Say we start off with space for just one item. Then the first doubling costs 1. The second costs 2. The third costs 4. The fourth costs 8. so …on.
- $1+2+4+8+\ldots+ m/2 \ +m \ \rightarrow m + m/2 + m/4 + \ldots + 4 + 2 + 1$



We see that the whole right side ends up being another square of size *m*, making the sum m + m = 2*m*.

- So when we do *m* appends, the appends themselves cost *m*, and the doubling costs 2*m*. Put together, we've got a cost of 3*m*, which is O(*m*). So on an average, each individual append is O(1). *m* appends cost us O(*m*).

STL VECTORS WITH ALGORITHMS

```
#include <vector>
using std::vector;

vector<int> myVector(100);
```

vector(n):

size():

empty():

resize(n):

reserve(n):

operator[i]:

at(i):

front():

back():

push_back(e):

pop_back():

sort(p,q):

random_shuffle(p,q):

reverse(p,q):

find(p,q,e):

min_element(p,q):

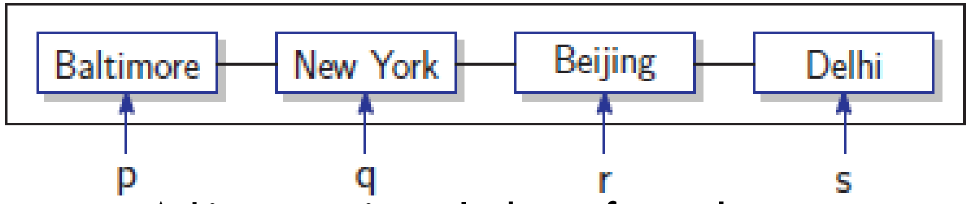
max_element(p,q):

for_each(p,q,f):

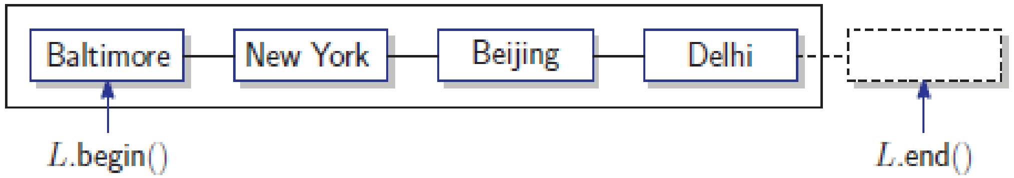
```
#include <cstdlib>                              // provides EXIT_SUCCESS
#include <iostream>                             // I/O definitions
#include <vector>                               // provides vector
#include <algorithm>                            // for sort, random_shuffle

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;
}
```

POSITION ADT & ITERATORS

- The Position ADT models the notion of place within a data structure where a single object is stored.
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Just one method:
  - object p.element(): returns the element at position
  - In C++ it is convenient to implement this as *p



A List container: Index of an element may change but not the position/ memory location.



Although a position is a useful object, it would be more useful still to be able to navigate through the container, for example, by advancing to the next position in the container. Such an object is called an **iterator**.

CONTAINERS AND ITERATORS

- What is a Container?
- Can you give some examples?
- Various notions of iterator:
  - (standard) iterator: allows read-write access to elements
  - const iterator: provides read-only access to elements
  - bidirectional iterator: supports both ++p and --p
  - random-access iterator: supports both p+i and p-i

Let C be a container and p be an iterator for C:

How will you iterate through the container?

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;
```

STL LISTS IN C++

```
#include <list>
using std::list;
list<int>myList;
```

```
List 1 (gqlist1) is :   0   2   4   6   8   10  12  14  16  18

List 2 (gqlist2) is :   27  24  21  18  15  12  9   6   3   0

gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() :   2   4   6   8   10  12  14  16  18

gqlist2.pop_back() :    27  24  21  18  15  12  9   6   3

gqlist1.reverse() :     18  16  14  12  10  8   6   4   2

gqlist2.sort():     3   6   9   12  15  18  21  24  27
```

program source: https://www.geeksforgeeks.org/

```cpp
#include <iostream>
#include <list>
#include <iterator>
using namespace std;
//function for printing the elements in a list
void showlist(list <int> g)
{
    list <int> :: iterator it;
    for(it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}
int main() {
    list <int> gqlist1, gqlist2;
    for (int i = 0; i < 10; ++i)
    {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);
    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);
    cout << "\ngqlist1.front() : " << gqlist1.front();
    cout << "\ngqlist1.back() : " << gqlist1.back();
    cout << "\ngqlist1.pop_front() : ";
    gqlist1.pop_front();
    showlist(gqlist1);
    cout << "\ngqlist2.pop_back() : ";
    gqlist2.pop_back();
    showlist(gqlist2);
    cout << "\ngqlist1.reverse() : ";
    gqlist1.reverse();
    showlist(gqlist1);
    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);
    return 0;
}
```

INDEX VS POSITION: MORE EXAMPLES

Using Indexing Operator

```cpp
#include <iostream>
#include <vector>
using namespace std;
int vectorSum1(const vector<int>& V) {
    int sum = 0;
    for (int i = 0; i < V.size(); i++)
        sum += V[i];
    return sum;
}
int main(){
    vector<int> v;
    int size;
    cout<<"Enter size of input vector : ";
    cin>>size;
    int aux;
    for(int i=0;i<size;i++){
        cin>>aux;
        v.push_back(aux);
    }
    cout<<"\nSum : "<<vectorSum1(v)<<endl;
    return 0;
}
```

```
Enter size of input vector : 4
23 56 2 5

Sum : 86
```

Using Iterators

```cpp
#include <iostream>
#include <vector>
using namespace std;
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;
}
int main(){
    vector<int> v;
    int size;
    cout<<"Enter size of input vector : ";
    cin>>size;
    int aux;
    for(int i=0;i<size;i++){
        cin>>aux;
        v.push_back(aux);
    }
    cout<<"\nSum : "<<vectorSum2(v)<<endl;
    return 0;
}
```

```
Enter size of input vector : 4
12 56 34 2

Sum : 104
```

SEQUENCE ADT

- The Sequence ADT generalizes the Vector and List ADTs.
- Elements are accessed by:
  - Index, or Position

```cpp
#include <iostream>
#include <list>
class NodeSequence {
public:
    using Iterator = std::list<int>::const_iterator;
    NodeSequence(const std::list<int>& data) : nodes(data) {}
    Iterator atIndex(int i) const {
        Iterator p = nodes.begin();
        for (int j = 0; j < i && p != nodes.end(); ++j) {
            ++p;
        }
        return p;
    }
    int indexOf(const Iterator& p) const {
        Iterator q = nodes.begin();
        int j = 0;
        while (q != p && q != nodes.end()) {
            ++q;
            ++j;
        }
        return (q == p) ? j : -1;
    }

private:
    std::list<int> nodes;
};
```

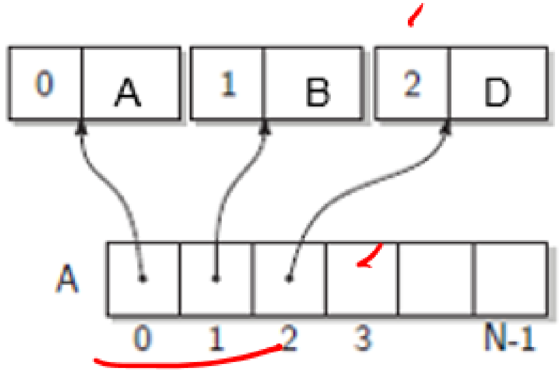
```cpp
int main() {
    std::list<int> data = {10, 20, 30, 40, 50};
    NodeSequence seq(data);
    auto it = seq.atIndex(2);
    if (it != data.end()) {
        std::cout << "Element at index 2: "
                  << *it << std::endl;
    }
    int index = seq.indexOf(it);
    std::cout << "Index of element 30: "
              << index << std::endl;
    return 0;
}
```

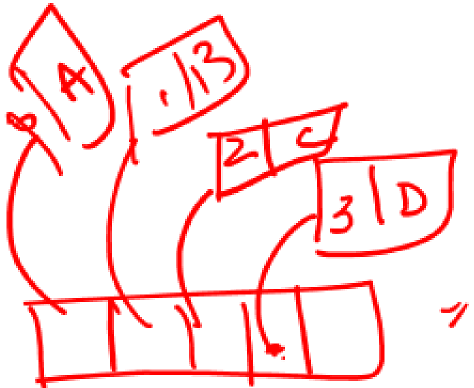
```
Element at index 2: 30
Index of element 30: 2
```

SEQUENCE ADT: ARRAY BASED

- We use a circular array storing positions.

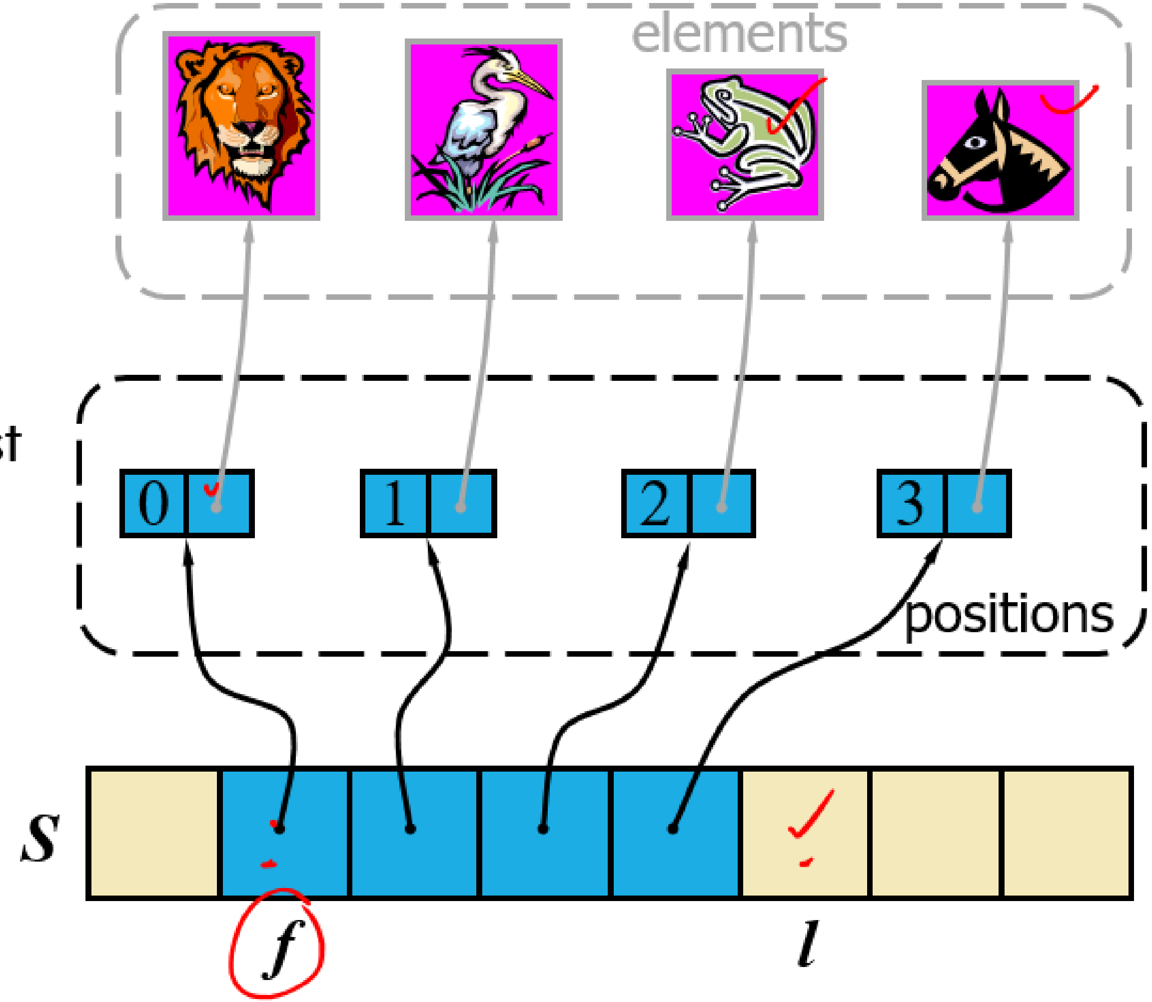
  A position object stores:
  - Element
  - Index
- Indices *f* and *l* keep track of first and last positions.





(Inserting C just before D?)

0 1 2 3



USAGE OF SEQUENCE ADT: BUBBLE SORT

- We will examine the usage of Sequence ADT and its implementation trade-offs using Bubble sort algorithm.

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)

For the sequence (5, 7, 2, 6, 9, 3) how many passes?

➔ i-th pass is limited to first n-i+1 elements.

ANALYSIS OF BUBBLE SORT

- Assuming that the sequence is implemented in such a way that access to elements and swaps take O(1) time each. ➔ Running time of i-th pass is ?.
- Total run-time = ?
- Hence, Bubble sort runs in ?

```
void bubbleSort1(NodeSequence& S) {
    int n = S.size();
    for (int i = 0; i < n; i++) {                // i-th pass
        for (int j = 1; j < n-i; j++) {
            NodeSequence::Iterator prec = S.atIndex(j-1);
            NodeSequence::Iterator succ = S.atIndex(j);
            if (*prec > *succ) {
                int tmp = *prec; *prec = *succ; *succ = tmp;
            }
        }
    }
}
```

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

Array based: atIndex takes O(1) ➔ O(n^2) for Bubble sort.

Node based: atIndex takes O(n) ➔ O(n^3) for Bubble sort.

Iterator increment takes O(1) in either array or node-based sequence implementations ➔ O(n^2) worst case for Bubble sort.

THANK YOU!

Next class: Trees, Priority queues, …