MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

21 Containers

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 10, 2022

List and Sequence Containers

- Vector (also called Array List)
  - Access each element using a notion of index in [0,n-1]
  - Index of element e: the number of elements that are before e
  - Typically we use the “index” (e.g., [ ])
  - A more general ADT than “array”
- List
  - Not using an index to access, but use a node to access
  - Insert a new element e before some “position” p
  - A more general ADT than “linked list”
- Sequence
  - Can access an element as vector and list (using both index and position)

The Vector ADT

- The Vector or Array List ADT extends the notion of array by storing a sequence of objects
- An element can be accessed, inserted or removed by specifying its index (number of elements preceding it)
- An exception is thrown if an incorrect index is given (e.g., a negative index)
- Main methods:

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.

Array-based Implementation of Vector

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation **at(i)** is implemented in O(1) time by returning A[i]
- Operation set(i,o) is implemented in O(1) time by performing A[i] = o

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

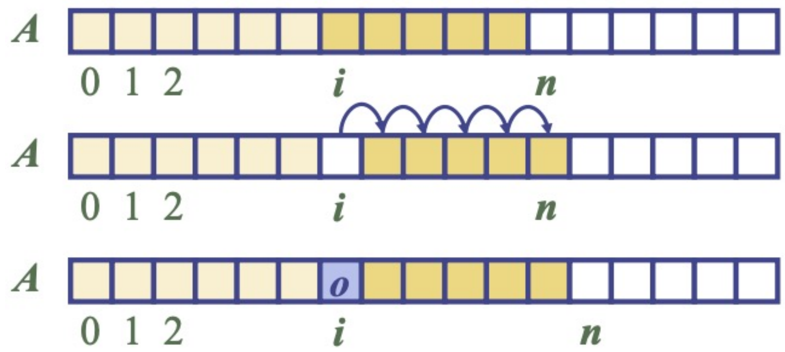
set(i,e): Replace the element at index i with e; an error condition occurs if i is out of range.

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.

Array-based Implementation of Vector - Insertion

- In operation insert(i, o), we need to make room for the new element by shifting forward the n - i elements A[i], …, A[n - 1]
  - In the worst case (i = 0), this takes O(n) time



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

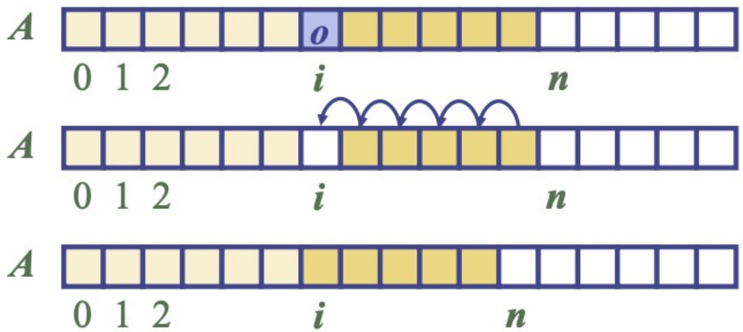
set(i,e): Replace the element at index i with e; an error condition occurs if i is out of range.

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.

Array-based Implementation of Vector - Removal

- In operation erase(i), we need to fill the hole left by the removed element by shifting backward the n - i - 1 elements A[i + 1], …, A[n - 1]
    - In the worst case (i = 0), this takes O(n) time



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.

Array-based Implementation of Vector in C++

```
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
};
```

Array-based Implementation of Vector in C++

```
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];
}

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

Array-based Implementation of Vector in C++

- The reserve function first checks whether the capacity already exceeds ***n***, in which case nothing needs to be done.
- The insert function first checks whether there is sufficient capacity for one more element. If not, it sets the capacity to the maximum of 1 and twice the current capacity.

```
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
}
```

Array-based Implementation of Vector - Performance

- In the array-based implementation of an array list:
  - The space used by the data structure is O(n)
  - size, empty, at and set run in O(1) time
  - insert and erase run in O(n) time in worst case
- If we use the array in a circular fashion, operations insert(0, x) and erase(0, x) run in O(1) time
- In an insert operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

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

Comparison of the Strategies to Resize Array

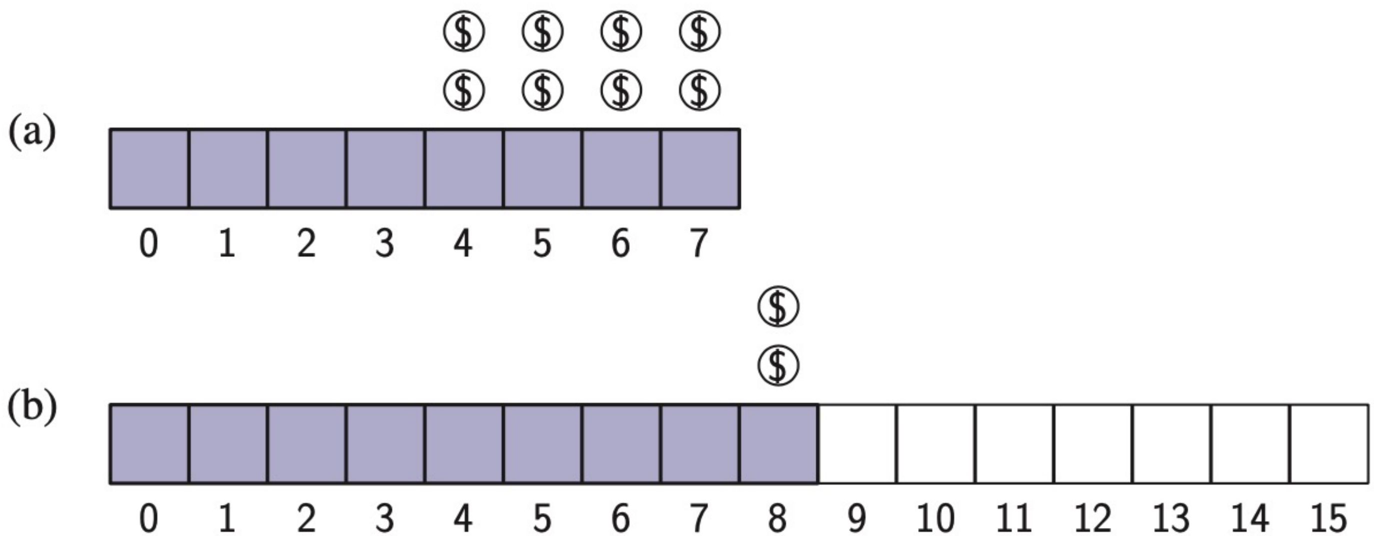
- Suppose we are doing only push operations
- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n insert(o) operations
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

Algorithm *insert*(o)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

- We assume that we start with an empty array of size 2
- We call **amortized time** of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$

Comparison of the Strategies to Resize Array

- Amortizations:
  - Certain operations may be extremely costly
  - But they cannot occur frequently enough to slow down the entire program
    - The less costly operations far outnumber the costly one
    - Thus, over the long term they are "paying back"



Comparison of the Strategies to Resize Array

- Amortizations:
  - Certain operations may be extremely costly
  - But they cannot occur frequently enough to slow down the entire program
    - The less costly operations far outnumber the costly one
    - Thus, over the long term they are "paying back"
  - The idea:
    - The worst-case operation can alter the state in such a way that "the worst case cannot occur again for a long time.
      - Thus, amortizing its cost

Comparison of the Strategies to Resize Array

- Suppose we are doing only push operations
- total time $T(n)$: to perform a series of n insert(o) operations
- Remember that every push (storing an element) takes 1 unit of time. After all we will have n pushes.
- Each array resize needs a time proportional to the size of the old array
- What is the time for n push operation
- For simplicity, we start with an array of capacity 2 and size zero and grow it dynamically
- We have to identify how many times the array resizes
- 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 Strategy Analysis

- We replace the array $k = n/c$ times
  - Suppose if you want to increment its size by 2, adding two more spaces.
- We suppose c = 2. There will be k=n/2 array resizes.
- For the incremental approach, each time we go past our capacity (k=(n/c)=(n/2)) times, we will increase capacity by **c=2** and we will have to copy the stuff already in the array into the new array.
- Assuming each item we copy requires 1 time unit.
- For 2 items: 2 units of time

  For 4 items: 4 units of time

  For 6 items: 6 units of time
- We the have need 2 + 4 + 6 + 8 + … + 2*k units of time
- Total time = n + 2 + 4 + 6 + 8 + … + 2*k = n + c(1+2+... + k).
- 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 $O(n^2)$ /n which is $O(n)$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
  - Suppose if you want to increment its size by 2, adding two more spaces. There will be k=n/2 array resizes.

- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

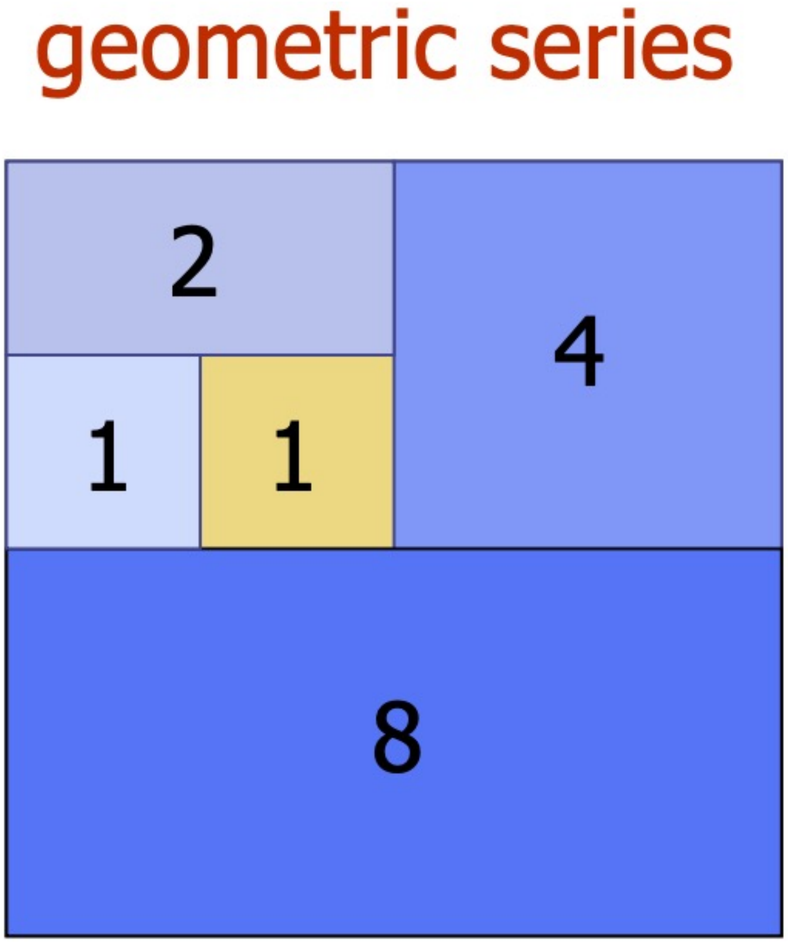
$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- 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 $O(n)$

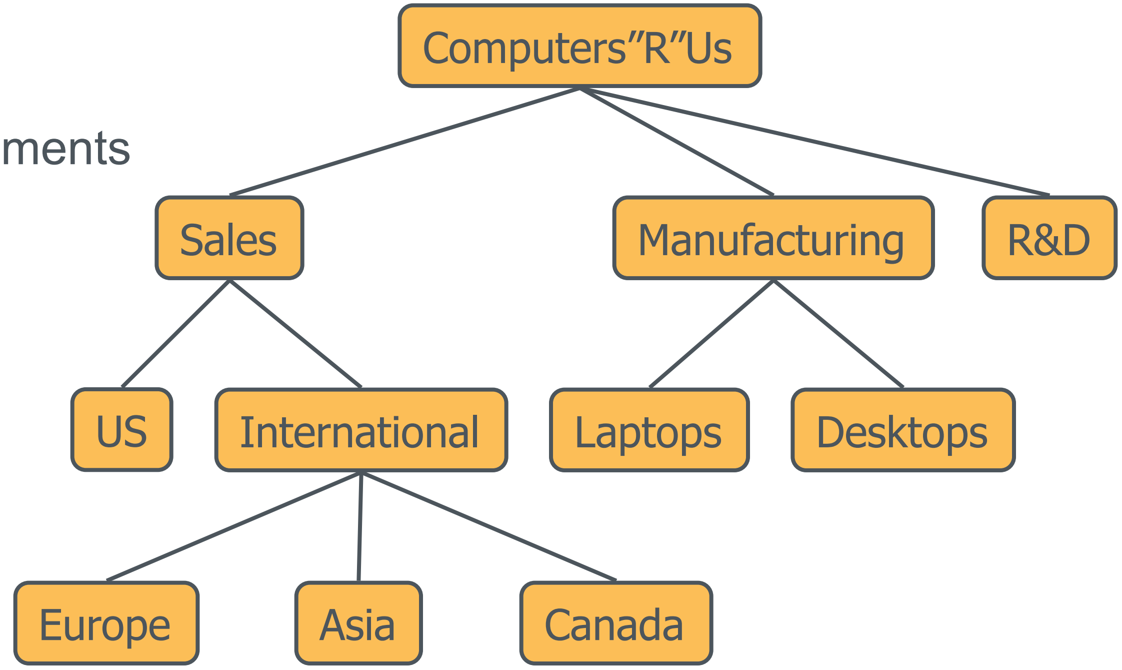
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n insert operations is proportional to
- $n + 2 + 4 + 8 + \ldots + 2^k =$
  $n + 2^{k+1} - 1 =$
- $3n - 1$
- $T(n)$ is $O(n)$
- The amortized time of an insert operation is $O(1)$



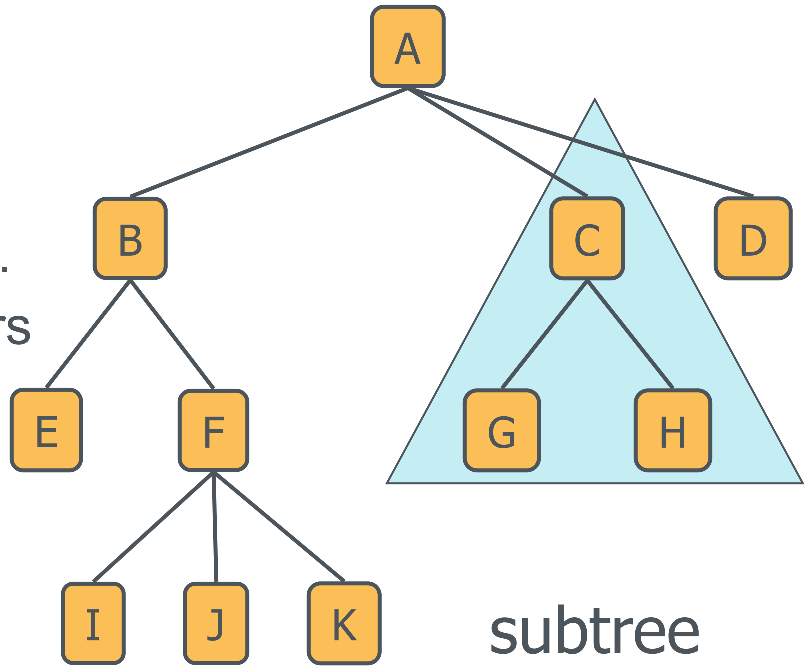
Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Trees

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.



- Subtree: tree consisting of a node and its descendants

Questions?