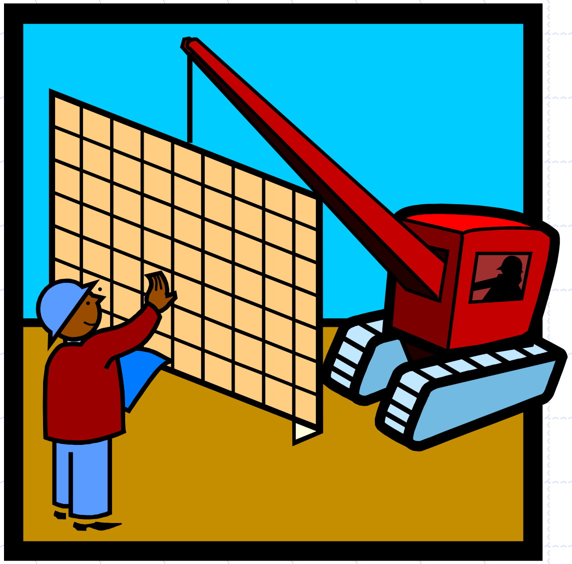
Array Lists

Section 6.1



Lists (Sequences)

- A list or sequence is a collection of elements stored in a linear order, so we can refer to the elements as first, second, third, etc.
- The index of an element in a list is the number of elements before it. The rank of an element is one more than its index.
- Computer languages, because it is more natural for the compiler, mostly use index rather than rank for accessing list or array elements.
- A sequence that allows access to its elements by their indices is called a vector or array list.

The Array List 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
- An exception is thrown if an incorrect index is given (e.g., a negative index)

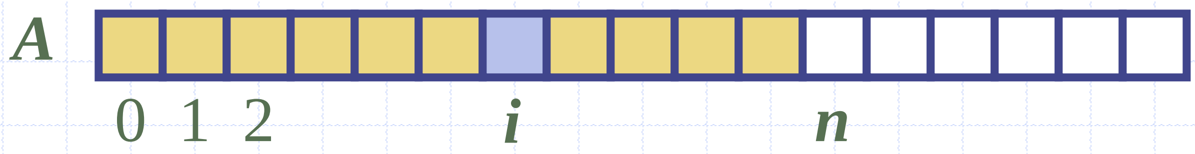
- Main methods:
  - at(integer i): returns the element at index i without removing it
  - set(integer i, object o): replace the element at index i with o
  - insert(integer i, object o): insert a new element o to have index i
  - erase(integer i): removes element at index i
- Additional methods:
  - size()
  - empty()

Applications of Array Lists

- Direct applications
  - Sorted collection of objects (elementary database or dictionary)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

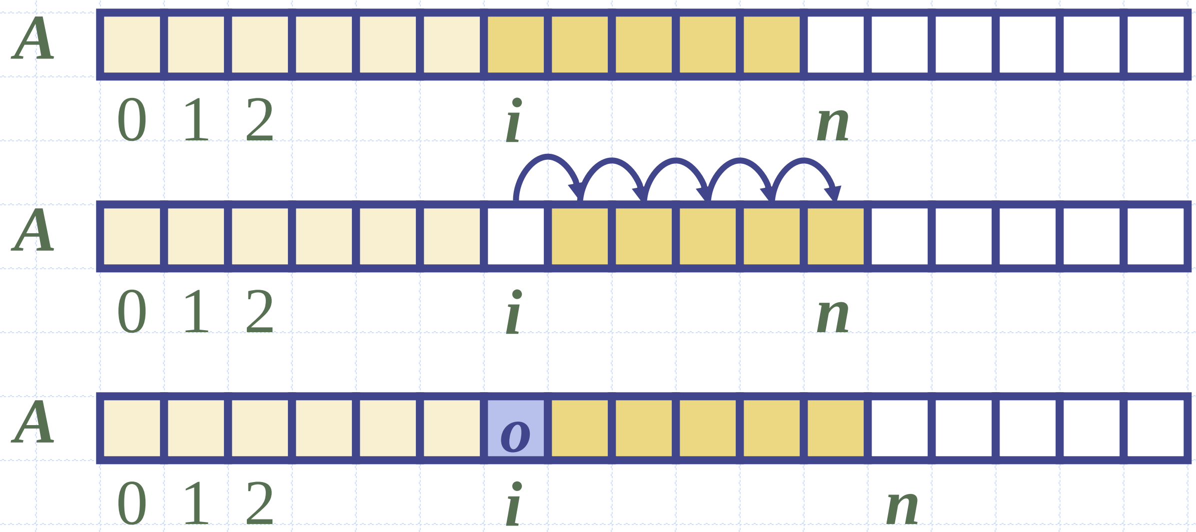
Array-based Implementation

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- 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 $A[i] = o$



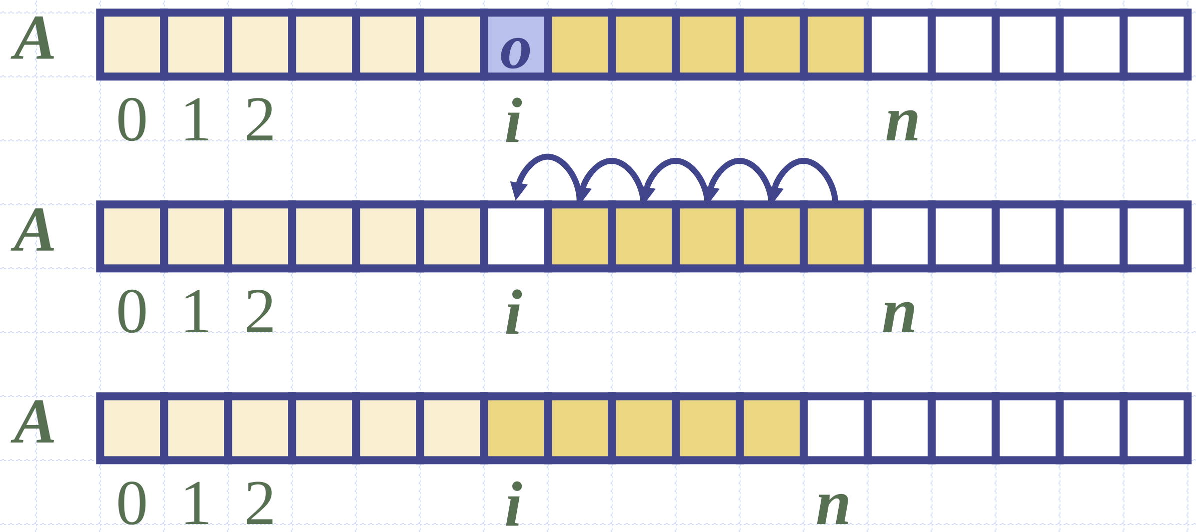
Insertion

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



Element Removal

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



Algorithms Insert and Erase

- *n* is a member variable that stores the number of elements in the ArrayList.
- An implementation of *insert()* should start by throwing an exception if the array A is full. This has been considered an implementation detail and left out of the algorithm.

Algorithm ***insert*(*i, e*)**
  for $j \leftarrow n - 1$ **downto** i **do**
    $A[j+1] \leftarrow A[j]$
  $A[i] \leftarrow e$
  $n \leftarrow n + 1$

Algorithm ***erase*(*i*)**
  for $j \leftarrow i+1$ **to** $n - 1$ **do**
    $A[j - 1] \leftarrow A[j]$
  $n \leftarrow n - 1$

Performance

- In the array-based implementation of an array list:
  - The space used by the data structure is $O(1)$
  - ***size***, ***empty***, ***at*** and ***set*** run in $O(1)$ time
  - ***insert*** and ***erase*** run in $O(n)$ time in the worst case
  - constrained by initial capacity of array
- In a linked-list-based implementation of an array list:
  - The space used by the data structure is $O(n)$
  - ***size*** and ***empty*** run in $O(1)$ time
  - ***at***, ***set***, ***insert***, and ***erase*** run in $O(n)$ time
  - no size constraint

Performance

- If we use the array in a circular fashion, operations ***insert***$(0, x)$ and ***erase***$(0, x)$ run in $O(1)$ time.
- We'd like to have the faster ***at*** and ***set*** of the array-based implementation, without the size constraint.
- This is possible using an extendable array, which brings our space cost to O(n).

Extendable Array

- An operation that accesses an index past the size of the array is called an overflow.
- When we get an overflow, we replace the array with a larger one.
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

```
Algorithm handleOverflow
    S ← new array of
          size …
    for i ← 0 to n−1 do
        S[i] ← A[i]
    A ← S
```

```
Algorithm insert(i, e)
  if n = A.length − 1 then
     handleOverflow( )

  for j ← n−1 downto i do
      A[j + 1] ← A[j]
  A[i] ← e
  n ← n + 1
```

Comparison of the Strategies

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

Incremental Strategy Analysis

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

$$\boldsymbol{k} + \boldsymbol{c} + 2\boldsymbol{c} + 3\boldsymbol{c} + 4\boldsymbol{c} + \ldots + \boldsymbol{mc} =$$

$$\boldsymbol{k} + \boldsymbol{c}(1 + 2 + 3 + \ldots + \boldsymbol{m}) =$$

$$\boldsymbol{k} + \boldsymbol{cm}(\boldsymbol{m} + 1)/2$$

- Since $\boldsymbol{c}$ is a constant, $\boldsymbol{T}(\boldsymbol{k})$ is $\boldsymbol{O}(\boldsymbol{k} + \boldsymbol{m}^2)$, i.e., $\boldsymbol{O}(\boldsymbol{k}^2)$
- The amortized time of an insert operation is $\boldsymbol{O}(\boldsymbol{k})$

Doubling Strategy Analysis

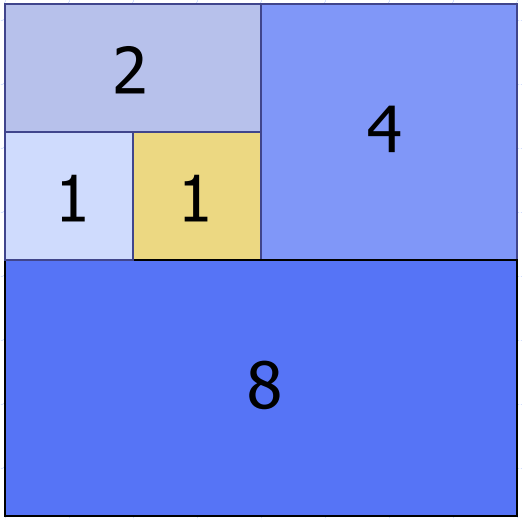
- We replace the array $m = \log_2 k$ times
- The total time $T(k)$ of a series of k insert operations is proportional to

  $k + 1 + 2 + 4 + 8 + \ldots + 2^m =$

  $k + 2^{m+1} - 1 =$

  $3k - 1$
- $T(k)$ is $O(k)$
- The amortized time of an insert operation is $O(1)$

geometric series



Implementing Vector with an Extendable Array

```
typedef int Elem;
class ArrayVector {
public:
    ArrayVector();
    int size() const;
    bool empty() const;
    Elem& operator[ ](int i);
    Elem& at(int i)
        throw(IndexOutOfBounds);
    void erase(int i);
    void insert(int I, const Elem& e);
    void reserve(int N);
    // …
private:
    int capacity;
    int n;
    Elem* A;
}

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

int ArrayVector::size() const
    { return n; }

bool ArrayVector::empty() const
    { return size() == 0; }
```

Implementing Vector with an Extendable Array

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

Elem& ArrayVector:at(int i)
    throw(IndexOutOfBounds) {
    if (i < 0 || i >= n)
        throw IndexOutOfBounds("…");
    return A[i];
}

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

```
void ArrayVector::reserve(int N) {
    if(capacity >= N) return;
    Elem* B = new Elem[N];
    for (int j=0; j<n; j++)  {
        B[j] = A[j];
    }
    if (A != NULL)
        delete [] A
    A = B;
    capacity = N;
}
```

Implementing Vector with an Extendable Array

```
void ArrayVector::insert(int i, const Elem& e) {
    if (n >= capacity)
        reserve(max(1, 2 * capacity));
    for (int j = n-1; j >= i; j--) {
        A[j+1] = A[j];
    }
    A[i] = e;
    n++;
}
```

STL vectors

- STL has a class vector.
- STL vectors are a type of container, which is a data structure that is used to hold a collection of objects.
- STL vectors provide all of the operations ArrayVector provided, along with push_back(e) and pop_back().
- STL vectors also provide many other auxiliary functions.
- When destroyed, an STL vector will destroy all objects it contains.