CH 6.
VECTORS, LISTS, AND SEQUENCES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO
VECTORS

• The Vector ADT (Ch. 6.1.1)
• Array-based implementation (Ch. 6.1.2)
APPLICATIONS OF VECTORS

• Direct applications
  • Sorted collection of objects (elementary database)

• Indirect applications
  • Auxiliary data structure for algorithms
  • Component of other data structures
VECTOR ADT

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

Main vector operations:
- at(i): returns the element at index i
- set(i, e): replace the element at index i with e
- insert(i, e): insert a new element e to have index i
- erase(i): removes the element at index i

Additional operations size() and empty()
ARRAY-BASED VECTOR STORAGE

• Use an array $V$ of size $N$
• A variable $n$ keeps track of the size of the vector (number of elements stored)
• $at(i)$ is implemented in $O(1)$ time by returning $V[i]$
ARRAY-BASED VECTOR
INSERTION

• In `insert(i, e)`, we need to make room for the new element by shifting forward the `n - i` elements `V[i], ..., V[n - 1]`

• In the worst case (`i = 0`), this takes `O(n)` time
**ARRAY-BASED VECTOR DELETION**

- In \( \text{erase}(i) \), we need to fill the hole left by the removed element by shifting backward the \( n - r - 1 \) elements \( V[r + 1], \ldots, V[n - 1] \).

- In the worst case \( (r = 0) \), this takes \( O(n) \) time.
PERFORMANCE

• In the array based implementation of a Vector
  • The space used by the data structure is $O(n)$
  • size(), empty(), at(i), and set(i, e) run in $O(1)$ time
  • insert(i, e), and erase(i) run in $O(n)$ time

• In an insert(i, e), when the array is full, instead of throwing an exception, we can replace the array with a larger one
EXERCISE:

• Implement the Deque ADT using Vector functions
  • Deque functions:
    • front(), back(), insertFront(e), insertBack(e), eraseFront(),
      eraseBack(), size(), empty()
  • Vector functions:
    • at(i), set(i, e), insert(i, e), erase(i), size(), empty()
EXERCISE SOLUTION:

Deque
- size() and empty()
- front()
- back()
- insertFront(e)
- insertBack(e)
- eraseFront()
- eraseBack()

Realization using Vector Functions
- size() and empty()
- at(0)
- at(size() - 1)
- insert(0, e)
- insert(size(), e)
- erase(0)
- erase(size() - 1)
## VECTOR SUMMARY

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>List Singly or Doubly Linked</th>
</tr>
</thead>
</table>
| **insert\((i, e)\) and erase\((i)\)** | \(O(1)\) Best Case \((i = n)\)  
\(O(n)\) Worst Case  
\(O(n)\) Average Case | ?                           |
| **at\((i)\) and set\((i, e)\)**      | \(O(1)\)             | ?                           |
| **size\() and empty\()**            | \(O(1)\)             | ?                           |
ITERATORS AND POSITIONS

• An iterator abstracts the process of scanning through a collection of elements

• Can be implemented on most data structures in this course, e.g., vector and list

• Methods of the Iterator ADT:
  • hasNext() – returns whether another element follows
  • next() – returns iterator for next element
  • elem() – return element at position, also known as dereference in C++ (* operator)

• Iterators handle many operations in a uniform way
  • Example – insert for list and vector take iterators so the functions are called the same way
  • Traversal of data structure from begin() to end()
LISTS AND SEQUENCES

- Iterators (Ch. 6.2.1)
- List ADT (Ch. 6.2.2)
- Doubly linked list (Ch. 6.2.3)
- Sequence ADT (Ch. 6.3.1)
- Implementations of the sequence ADT (Ch. 6.3.2-3)
LIST ADT

- The List ADT models a sequence of positions storing arbitrary objects.
  - Establishes a before/after relation between positions.
- It allows for insertion and removal in the “middle”.
- Generic methods:
  - size() and empty()
- Accessor methods:
  - begin() and end()
- Update methods:
  - insertFront(e), insertBack(e), insert(p, e) – Note insert will insert e before iterator p
  - eraseFront(), eraseBack(), erase(p)
\textbf{INSERT}(p, e)
ERASE($p$)
PERFORMANCE

- Assume doubly-linked list implementation of List ADT
  - The space used by a list with \( n \) elements is \( O(n) \)
  - The space used by each iterator of the list is \( O(1) \)
  - All the operations of the List ADT run in \( O(1) \) time
### List Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>List Singly-Linked</th>
<th>List Doubly-Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>begin()</code>, <code>end()</code>, <code>insertFront()</code>, <code>insertBack()</code>, <code>eraseFront()</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
| `insert(p, e)`, `eraseBack()`, `erase()` | $O(n)$ Worst and Average case  
$O(1)$ Best case         | $O(1)$             |
| `size()` and `empty()`         | $O(1)$             | $O(1)$             |
SEQUENCE ADT

- The **Sequence ADT** is a combination of the Vector and List ADTs
- Elements accessed by
  - Index or
  - Iterator (Position)
- All items in the List ADT plus the following “bridging” functions:
  - `atIndex(i)` – returns position of element at index `i`
  - `indexOf(p)` – returns index of element at position `p`
APPLICATIONS OF SEQUENCES

• The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements

• Direct applications:
  • Generic replacement for stack, queue, vector, or list
  • Small database (e.g., address book)

• Indirect applications:
  • Building block of more complex data structures
ARRAY-BASED IMPLEMENTATION

- We use a circular array storing positions.
- A position object stores:
  - Element
  - Index
- Indices $f$ and $l$ keep track of first and last positions.

![Diagram showing the array-based implementation with elements and positions labeled](image-url)
### Sequence Implementations

<table>
<thead>
<tr>
<th>Function</th>
<th>Circular Array</th>
<th>List Doubly-Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size()</code>, <code>empty()</code>, <code>begin()</code>, <code>end()</code>, <code>insertFront()</code>, <code>insertBack()</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>atIndex(i)</code> and <code>indexOf(p)</code></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><code>insert(p, e)</code> and <code>erase(p)</code></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
INTERVIEW QUESTION 1

• Write code to partition a list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.

INTERVIEW QUESTION 2

• Implement a function to check if a list is a palindrome.