CSCU9A3
Data Structures

Data Structures

- We often want to organise some data together so that it can be considered as a single unit.
- The organisation determines how the information is accessed.
- We call these generic organisations of data, a ***data structure***.

Data Structures

- A data structure has two parts:
  - a representation,
  - a set of operations to manipulate the data structure.
- When dealing with data structures, it is best to think in terms of ***Abstract Data Types*** (ADTs).

ADTs

- ADTs : think of a data structure in abstract terms, irrespective of the language or implementation
- There may be many alternative ways in which an ADT may be represented, but they will all have the same set of visible operations.
- Classes allow us to hide representation details while offering a set of visible methods, they are therefore ideal for representing ADTs.
- ***Classes can be used to implement abstract data types.***

ADTs

- Typically, the person who designs and implements an ADT is different from the person who uses the ADT.
- The *user* of an ADT is primarily concerned with ***what*** the ADT does, not with ***how*** it is implemented.
- The *implementer* of an ADT concentrates on the internal details and should ignore the situations in which it is to be used.
- That way the ADT can be used in situations for which it was not planned - giving us a ***re-usable component***.
- The class representing an ADT is used as a component in the internal computations in a program and will typically not have a graphical representation defined within it.

5

Classes

- Typically, the person who designs and implements a class is different from the person who uses the class.
- The *user* of an class is primarily concerned with ***what*** the class does, not with ***how*** it does it.
- The *implementer* of a class concentrates on the internal details and should ignore the situations in which it is to be used.
- That way the class can be used in situations for which it was not planned - giving us a ***re-usable component***.
- The class is used as a component in the internal computations in a program and will typically not have a graphical representation defined within it.

6

Library classes

- Note that the last two slides are almost identical to show the connection between classes and ADTs.
- An important feature of a language like Java are the number of **library classes** that are built into the language.
- You have made use of the library classes such as JButton and JFrame, but there are library classes for a huge number of different situations.

7

Array List ADT

- Array List
  - An Array List extends the notion of an array by storing a variable length sequence of arbitrary 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)

Array Lists 8

Array List ADT

- Main methods:
  - get(integer i): returns the element at index i without removing it
  - set(integer i, object o): replace the element at index i with o and return the old element
  - add(integer i, object o): insert a new element o to have index i
  - remove(integer i): removes and returns the element at index i
- Additional methods:
  - size()
  - isEmpty()

Applications of Array Lists

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

Example in Java

```
public void arrayLists()
{
        ArrayList<String> players = new ArrayList<String>();

        players.add("Savi");
        players.add("David");
        players.add("John");
        players.add("Simon");

        System.out.println("The players are:");
        for (int p=0; p<players.size(); p++)
        {
                System.out.println("    " + players.get(p));
        }
}
```

Array Lists 11

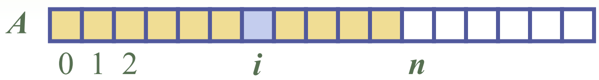
Example in Java : Alternative

```
public void arrayListsIn()
{
        ArrayList<String> players = new ArrayList<String>();

        players.add("Savi");
        players.add("David");
        players.add("John");
        players.add("Simon");

        System.out.println("The players are:");
        for (String s:players)
        {
                System.out.println("    " + s);
        }
}
```

Array Lists 12

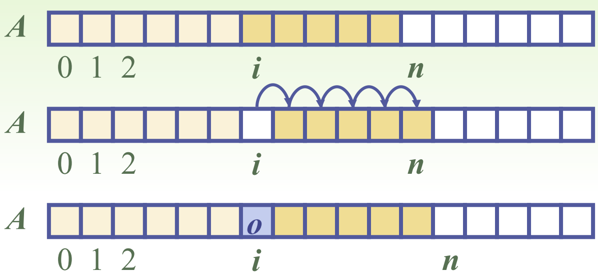
Array-Based Implementation

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



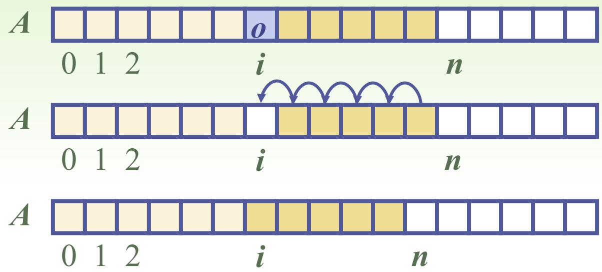
Insertion

- In operation ***add***(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 ***remove***(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



Performance

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

Growable Array-based Array List

- In an add(o) operation (without an index), we always add at the end
- When the array is full, 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 *add*(*o*)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

Comparison of the Strategies

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

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add 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., $O(n^2)$, since k is proportional to n.
- The amortized time of an add 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 add 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 add operation is $O(1)$

geometric series