Implementation of ADTs in Java

Recall ADT lists in lecture 2.

- A classic dynamic data structure of Computer Science
- an ADT = data (data structure) + operations on the data
- Note the difference between a *list* and a *set*
- Operations of a ADT list:

1. CreateList()
2. isEmpty()
3. size()
4. add(newPosition, newDataItem)
5. remove(Position)
6. initilise() (removeAll())
7. get(Position, dataItem)
8. etc.

Example 1

Given a list of integer

`L=[1, 223, 4, 3, 4, -8, 0]`

- The type of dataItems? Integers
- The result (postcondition) of an operation?

1. CreateList(L): []
2. isEmpty(L): false
3. size(L): 7
4. add(newPosition, newDataItem):
   precondition: let newPosition be 3, and newDataItem be 5;
   postcondition: [1, 223, 5, 4, 3, 4, -8, 0]
5. remove(Position):
   precondition: ?; postcondition: ?
6. initilise(), i.e. removeAll()
   precondition: ? ; postcondition: ?
7. get(Position, dataItem)
   preconditions: ? ; postcondition: ?

Example 2

Recall the q3 in Lab sheet 2. Given an appointment book: [(24-10-2002, 17:00, "about Java"), (24-12-2002, 19:00, "Cristmas Party")]

- The type of dataItems: ?
  An appointment: (24-10-2002, 17:00,"about Java")
- The result (postcondition) of an operations:
  1. CreateAppointmentBook(): []
  2. isAppointment(date, time): true
  3. makeAppointment(date, time, purpose): precondition?; postcondition?
  4. cancelAppointment(date, time): precondition?; postcondition?
  5. checkAppointment(date, time): precondition?; postcondition?
  6. checkAppointment(date): precondition?; postcondition?
  7. Size(appointmentBook): 2
  8. etc.

Oberservation

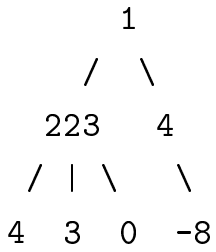
- Operations are independent of the type of items
- The type of the list elements in Java: ? class *Object*
- Every class in Java is ultimately derived from the class *Object*.
- Any class created in Java could be used as an item in a properly defined list class
- In fact, operations are also independent of the data structures

Trees

For example, let us have a look of a new data structure that is called tree.

- A tree is a data structure that represents a hierarchical relationships among data items.
- In a list, e.g, [1, 223, 4, 3, 4, -8, 0], each datum has only a one-to-one relationship with the one before it and the one after.
- In a tree structure, each datum has a one-to-one relationship to the one above it but a one-to-many relationship to the ones below it.

- For example, the same set of data could related in a *tree*:



Terms

root The top most node is called *root*.

leaves The nodes that have no children are called *leaves*.

parent The predecessor node that every node (except the root) has.

children The successor nodes that all nodes (except leaves) have.

siblings Successor nodes which share a common parent.

subtrees A subtree is a substructure of a tree. Each node in a tree may be thought of as the root of the *subtree*.

degree of a node The number of subtrees in a node.

degree of the tree The maximum of the degrees of the nodes of the tree.

ancestors of a node All the nodes along the path from the root to that node.

path from node n_1 to n_k A sequence of nodes from n_1 to n_k. The *length* of the path is the number of edges on the path.

depth of a node The length of the unique path from the root to the node.

height of a node The longest path from the node to a leaf.

forest A collection of trees is called a *forest*.

binary trees The trees in which every node has two subtrees, although either subtree could be empty.

Some advantages of a tree structure

- Recursive nature: [a]
- Certain structures, e.g. binary tree is very useful
- If we stored some data in a good way, we may have more efficient algorithms (solutions)

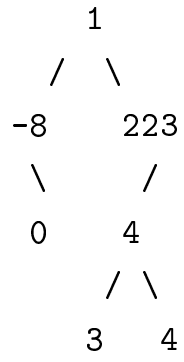
[a] A general tree is a set of nodes that is either empty or has a root from which zero or more subtrees descend hierarchically. Each subtree itself satisfies the definition of a tree.

Example

A binary search tree is a binary tree where each node has the following order properties in value:

- all its left children are smaller than itself
- all its right children are bigger than or equal to itself

This is a binary search tree for the list above.



Now suppose that we want to search for a number 22 in the set of data. In the list structure, we need to do 7 comparisons before we know that number 22 is not there, but only 4 comparisons for the tree structure.

- In general, the number of comparisons needed to search among n data in a *balanced* binary search tree structure is O(log n) in the worst case,
- while the number of comparisons needed in a list is O(n) in the worst case.

Let y be the number of comparisons needed by a searching Algorithm.
$y = log(n)$ for the binary search tree structure, and $y = n$ for the list structure.

If we plot the two functions, we see clearly, when n increases, y does not increase in the tree structure as much as in the list structure. This means the algorithm on a binary tree structure is more *efficient* then the algorithm on a list structure.

Oberservation

- How to implement what we have done so far?
  use Java *interfaces*

Interfaces

- Java provides a special kind of class called *interface* that defines specifications of a set of methods.
- In programming design, especially a large programming design, we often come across the situation at a stage where we may
  - not be sure about some details
  - not know some details
  - choose to decide some details later.
- but we know this part is needed in the program, and we know the relationship between this part to others.

Interfaces

Interfaces allows the designer to concentrate on the essentials of a class, in other words:

- what a class or method does
- the behaviour and interface of one kind to the world
- all the details to be filled later.

- Interface declaration

```
==============================
interface INTERFACENAME {
  ... // method specifications
==============================
}
```

- Interface implementation

```
========================================
class CLASSNAME implements INTERFACENAME {
  ... // bodies for the interface methods
  ... // own data and methods
========================================
}
```


Example

We could define typical operations used on simple movable machines as an interface below:

```
interface Movable {
  boolean start ();
  void stop ();
  boolean turn (int degrees);
  double fuelRemaining ();
  boolean changeSpeed (double kmPerHour);
}
```


Later on, we can implement the interface for a plane as a movable machine as follows:

```
class Planes implements Movable {
  boolean start () {
    ... // starting actions
  }

  void stop () {
    ... // stop actions
  }

  boolean turn (int degrees) {
    ... // turning actions
  }

  double fuelRemaining () {
    ... // return the amount of plan fuel remaining

  boolean changeSpeed (double kmPerHour) {
    ... // accelerate or decelerate if kmPerHour<0
  }
}
```

Example

We could define typical operations used for our appointment book as an interface below:

```
interface AppointmentBook{
  void CreateAppointmentBook();
  boolean isAppointment(Date date, Time time);
  List makeAppointment(Date date, Time time,
                               String purpose);
  List cancelAppointment(Date date, Time time);
  List checkAppointment(Date date, Time time);
  Appointment checkAppointment(Date date);
  int Size(List appointmentBook);
}
```


Later on, we can implement the interface for a YearPlan as an AppointmentBook as follows:

```
class YearPlan implements AppointmentBook {

  void CreateAppointmentBook() {
     ... //
  }

  boolean isAppointment(Date date, Time time) {
     ... //
  }

  List makeAppointment(Date date, Time time,
                               String purpose) {
    ... //
  }

  List cancelAppointment(Date date, Time time) {
    ... //
  } //
```



```
  List checkAppointment(Date date, Time time) {
    ... //
  } //

  Appointment checkAppointment(Date date) {
    ... //
  } //

  int Size(List appointmentBook) {
    ... //
  } //
} // end of YearPlan
```