Abstract Data Types

Previous lectures: algorithms and their efficiency analysis.

Coming lectures: data structures

In this lecture:

- Abstract data types
- ADTs as a design tool
- Examples: integer ADT, List ADT

Data Types

- All programming languages provide basic data types.
- These have core operations (or methods):
- For example, `int` has arithmetic operations `+, -, /, *` and comparisons `<,>,==`, etc.
- Also provide composite (structured) data types (arrays in Java, lists in Haskell).

Abstract Data Types

- Data type = data + methods
- Abstract data type:
  - Logical domain for data
  - Collection of core operations / methods
- ADTs abstract from implementation details, e.g.:
  - How data domain is built from more basic data types
  - How the core methods actually work
  - Efficiency (complexity) of algorithms for core methods
  - Correctness of algorithms

ADTs as Design Tool

- Useful aid in software design process.
- Consider what kind of data you need to solve the problem.
- Consider which operations do you need to perform with the data (which arguments the operations take, what is their return type).
- Essentially, designing a class interface (which methods the class should have).
- (The idea of ADTs predates object oriented programming and is more general.)

Principle of Abstraction

When solving a problem, separate

- what is to be done, and
- how it is to be done

Integer ADT

Logical domain: integers (whole numbers greater or equal to 0).

Methods:

- `Integer add(Integer x, Integer y)`
  Postcondition: returns the sum of `x` and `y`
- `Integer multiply(Integer x,Integer y)`
  Postcondition: returns `x` times `y`
- `boolean equals(Integer x, Integer y)`
  Postcondition: returns true of `x` equal to `y`

and so on.We don't know how this is going to be implemented - as a 16-bit or a 32-bit number etc.

List ADT (Informal)

Data (what things are lists?): linear collections of items.

Methods (what can one do with a list?):

- insert an element
- delete an element
- access the head of the list
- move to the next item from where you are

There is no single "true" list ADT; operations depend on what we want to do with lists.

List ADT contd.

- Need ADT for items in the list with methods for assigning items and comparing them. Let's call that ADT **`ItemType`**
- In Java implementation, **`ItemType`** will be assumed to be **`Object`**.
- The example is based on Shaffer's book Chapter 4. However Shaffer identifies an ADT with an interface in Java; this may be confusing because ADT can be described independently from any programming language.

List ADT methods

- a method to initialise a list (**`List()`** in Java syntax)
  Postcondition: creates an empty list
- **`void insert(ItemType item)`**
  Postcondition: **`item`** inserted into list (at the current position) .
- **`ItemType remove()`**
  Postcondition: item at current position deleted from list (and returned).
- **`ItemType currValue()`**
  Postcondition: The item at current position is returned.

List methods contd.

- **`boolean isEmpty()`**
  Postcondition: Returns true if list is empty, false otherwise.
- **`void setFirst()`**
  Postcondition: set current position at the first position in the list.
- **`void next()`**
  Postcondition: current position moves one to the right
- **`void prev()`**
  Postcondition: current position moves one to the left
- etc. (see Shaffer's book).

Possible modifications of List ADT

- insertion and deletion at a specified position
- no way to move back (to the previous item)
- ordered list: insertion in order
- (and different names for methods, obviously...)

Implementations of List ADT

Different *data structures* (concrete ways to organise data in computer memory) can be used to implement the List ADT:

- Various linked lists
- Recursive lists (consisting of head and tail)
- An array or vector

Array implementation of List ADT

Class for an array implementation of a List (`AList`, see Shaffer).

Fields: an array of Objects `listArray` to store items; `int numInList` to store the actual number of items in the list; `int curr` to store the index of the current position; etc.

Insertion at the current position

curr=0

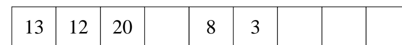
insert 23 at current position:

Insertion at the current position

curr=0

insert 23 at current position

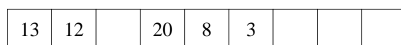
Insertion at the current position



curr=0

insert 23 at current position

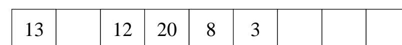
Insertion at the current position



curr=0

insert 23 at current position

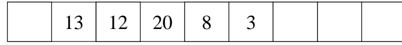
Insertion at the current position



curr=0

insert 23 at current position

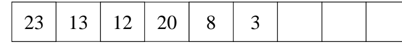
Insertion at the current position



`curr`=0

insert 23 at current position

Insertion at the current position



`curr`=0

insert 23 at current position

Array implementation of List ADT

```
class AList {
     …
   public void insert(Object it){
     … (check that there is space and curr is
        a valid index)
      for(int i=numInList; i>curr;i--){
         listArray[i]=listArray[i-1];
      }
      listArray[curr]=it;
      numInList++;
   }
```

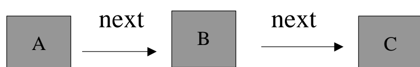
Array implementation of List ADT

`remove()` method is similar: also involves moving items to close the gap in the array.

Note that both `insert()` and `remove()` methods in this implementation have O(N) worst case complexity.

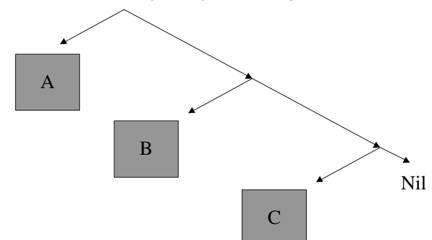
Linked List

A linked list consists of linked nodes:

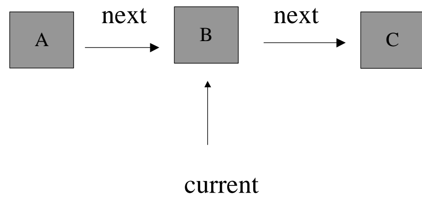


Other Lists

A linked list is not the only way to imagine lists*:*

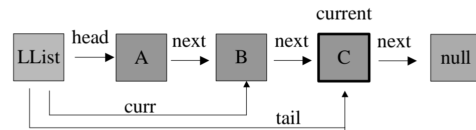


Linked List with Current



Linked List Implementation

Each node has a **`next`** field which says what the next item in the list is. The list has a **`head`** and **`tail`** fields which refer to the head and the last element of the list. There is also an indicator for where we are in the list: **`curr`** points to the node ***preceding*** the current element (technicality in Shaffer's implementation - does not have to be this way!)



Class for list elements (nodes)

```
class Link {
    private Object element;
    private Link next;
    Link(Object it, Link nextval) {
          element = it; next = nextval;}
    Object element() {return element;}
     Link next() {return next;}
     Link setNext(Link nextval) {
          return next = nextval;}
      Object setElement(Object it) {
          return element = it;}
}
```

Class for a linked list

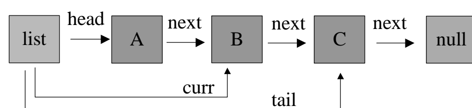
```
class LList {
  private Link head;
  private Link tail;
  protected Link curr;

  … (constructor etc.)
  public void insert(Object it){
    if (curr!=null) {
       curr.setNext(new Link(it, curr.next());
       if(tail == curr) tail = curr.next();
    }
    else ...
```

Linked List

`curr.setNext(new Link(it, curr.next());`

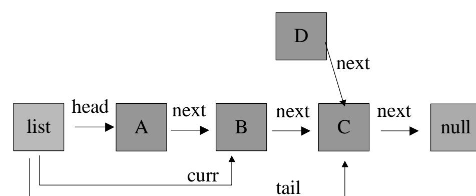
insert a new object D:



Linked List

`curr.setNext(new Link(it, curr.next());`

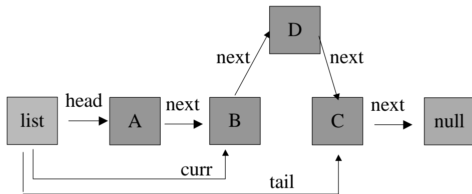
create a link with D:



Linked List

`curr.setNext(new Link(it, curr.next());`

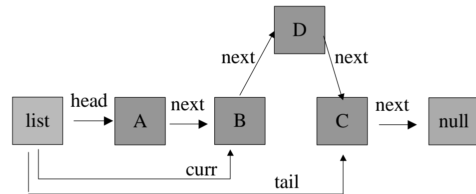
set this to be the next link from curr:



Linked List

`curr.setNext(curr.next().next());`

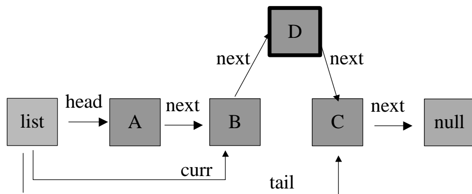
Remove the current element:



Linked List

`curr.setNext(curr.next().next());`

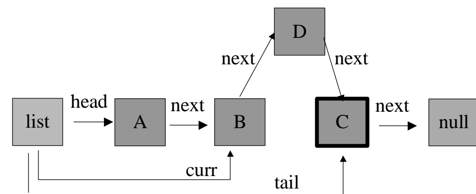
Find `curr.next()`:



Linked List

`curr.setNext(curr.next().next());`

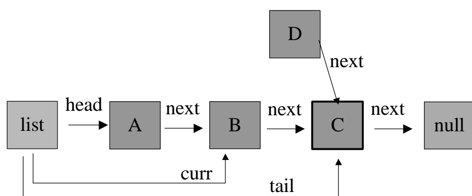
Find `curr.next().next`:



Linked List

`curr.setNext(curr.next().next());`

Set the link from `curr` to be to `curr.next().next`:



Comparison

- Complexity of insertion and deletion in linked lists: O(1) (faster than in the array implementation)
- Search for a given element: O(N) in both (given that the list is unordered).
- Static vs dynamic: memory for the AList is allocated in advance, for the LList it is allocated as new elements are added. For the former implementation, better to know the size in advance.
- AList is a bit simpler to implement...

Summary

Different stages in solving a problem:

- Designing ADTs (e.g. List ADT)
- Choosing data structures to implement them (e.g. as linked list,or as an array)
- Actual implementation: only here can you talk about efficiency of methods, but since there are standard ways to implement a linked list people refer to “complexity of insertion in a linked list with current”.

Reading

- Shaffer, Section 1,2 (Abstract data Types and data Structures), Section 4.1 (Lists)
- Other list implementations in Java - any textbook on data structures using Java