CE204 Data Structures and Algorithms

David Richerby

University of Essex

Autumn term 2023–24

Unit 1: Linear data structures

Preamble: what is an array?

A collection of values of the same type, each identified by an integer index and stored in consecutive memory locations, so that the location of each element can be computed from its index.

index	0	1	2	$\cdots$	$n-1$
value	$\star$	$\star$	$\star$	$\cdots$	$\star$

What is an array, behaviourally?

As users of modern programming languages, we don't normally care about these precise implementation details.

We can:

- create a new array,
- set the value of the `ith` element,
- get the value of the `ith` element.

Java interfaces

Suggestive of Java interfaces:

```
interface ObjectArray {
    void create (int length);
    void set (int index, Object value);
    Object get (int index);
}
```

A valid implementation (?)

```
class UselessArray implements ObjectArray {
    void create (int length) { return; }

    void set (int index, Object value) {
        System.out.println ("Sure, I did that.");
    }

    Object get (int index) {
        System.out.println ("Oops, I forgot what you said.");
        return null;
    }
}
```

Abstract data types (ADTs)

To specify a data structure, we need more than just an interface.

Abstract datatypes combine:

- an interface – the available operations;
- a behavioural description of these operations;
- requirements on running time.

ADTs say nothing about implementation or language.

ADTs were proposed by Barbara Liskov (one of the inventors of OOP) and Stephen Zilles.

Arrays as an ADT

Operations. `create`, `set`, `get` as before.

Behaviour. For any integer value $i \in \{0, \ldots, \texttt{length} - 1\}$, `get(`$i$`)` returns the `Object` argument of the most recent call `set(`i`, something)`.

```
a.create (10);
a.set (4, "Hello");
// arbitrary code with NO calls to a.set (4, ...)
a.get (4); // returns "Hello"
```

Running time of `set` and `get` does not depend on the length of the array.

Abstract vs concrete

Concrete datatypes are actual datatypes in actual languages, e.g., `int`, `double[][]`, `class MyClass { ... }` in Java.

Concrete datatypes defined entirely by their implementation.

ADTs are implemented as concrete datatypes.

Implementing ADTs

In Java, ADTs are implemented as classes.

Application programmers should be able to switch to a different implementation of an ADT with

- minimal code changes;
- no change in behaviour.

ADT implementers should only allow users to access data through the defined operations (e.g., fields and helper methods should be private).

Abstract data types – summary

- Abstract data types (ADTs) specify a datatype by defining:
  - the operations on the data,
  - the behaviour of the operations and
  - their running times.
- We defined arrays as an ADT as an example.

Lists

- A list is what you think it is – a sequence of data items.
- e.g., as implemented by Java's `LinkedList` and `ArrayList`.
- The first item is called the *head*.
- The last item is the *tail*.

The list ADT

Operations and behaviour.

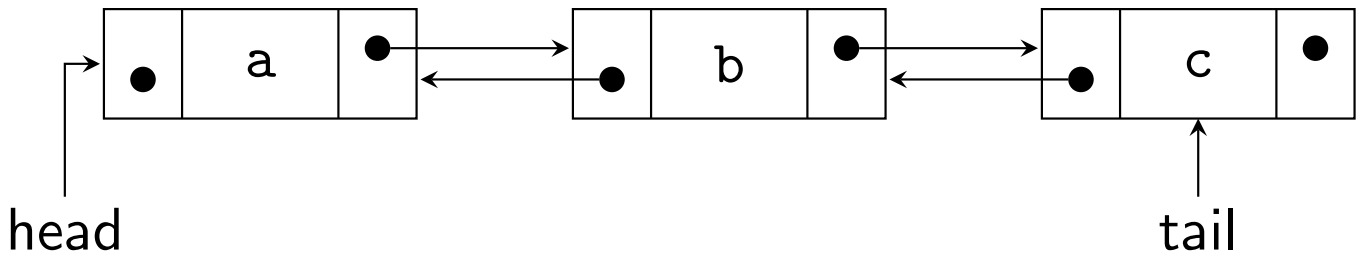
- `void create()`, `boolean isEmpty()`, `int length()`.
- `void insert(int index, String s)` insert `s` so it becomes the `index`th item in the list.
- `String get(int index)` returns the `index`th item.
- `void delete(int index)` deletes the `index`th item.

Running time of `insert()`, `get()`, `delete()` is proportional to size of list.

NB: different authors disagree on exactly what the operations should be.

Linked lists

- Doubly linked list: each item has a reference to the next and previous items.



- Singly linked list: each item only has a reference to the next one.

Lists vs arrays

Size:

- List: can grow and shrink as needed.
- Array: size fixed at creation time.

Insert/delete:

- List: naturally supported.
- Array: requires copying to new array.

Get `i`th item:

- List: expensive (time proportional to length).
- Array: cheap (constant time).

Java implementation (1)

```
public class DoublyLinkedList {
    private class Item {
        String value; /* This is a list of Strings. */
        Item next;
        Item prev; /* Omitted for singly linked list. */

        Item (String value, Item prev, Item next) {
            this.value = value;
            this.next = next;
            this.prev = prev;
        }
    }

    private Item head = null;
    private Item tail = null;
    private int length = 0;
    ...
```

Java implementation (2)

The following is a useful helper method – returns the `indexth Item` object in the list.

For internal use only! Caller responsible for ensuring `index` is valid.

```
private Item getItem (int index) {
    Item cur = head;
    for (int i = 0; i < index; i++)
        cur = cur.next;
    return cur;
}
```

NB: `getItem (k)` makes `k` steps along the list.

To find things in the list, we must walk along it, item-by-item.

Java implementation (3)

Retrieving data from the list is now easy:

```
public String get (int index) {
    if (index < 0 || index >= length)
        return null;
    else
        return getItem(index).value;
}
```

Inserting at the head

- Create a new item whose next item is the current head.
- Make the new item be the head.
- Set the old head's previous to be the new item.
- Fix the tail if necessary.
- Increase length.

```
public void insertAtHead(String value) {
    Item e = new Item(value, null, head);
    head = e;
    if (e.next == null) tail = e;
    else e.next.prev = e;
    length++;
}
```

Inserting at the tail is similar.

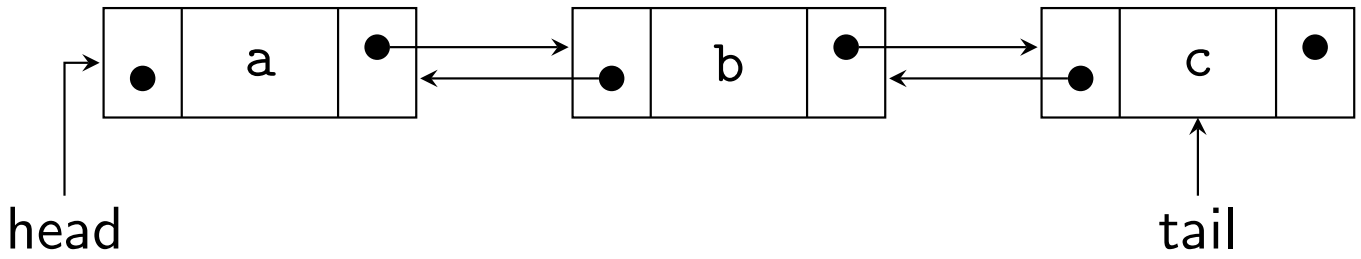
Deleting the head

- Undo the steps of insertion.
- The second item becomes the head.
- The new head now has no previous item.
- Fix the tail if needed; fix the length

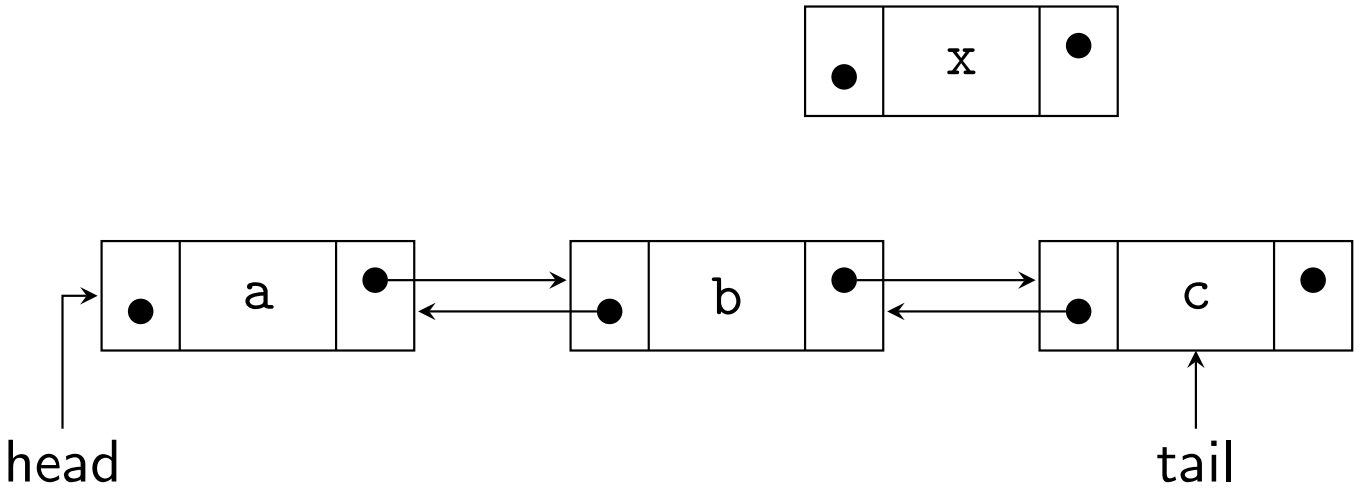
```
public void deleteHead() {
   if (length == 0) return;
   else if (length == 1)
       head = tail = null;
   else {
       head = head.next;
       head.prev = null;
   }
   length--;
}
```

Deleting the tail is similar.

Inserting and deleting in the middle

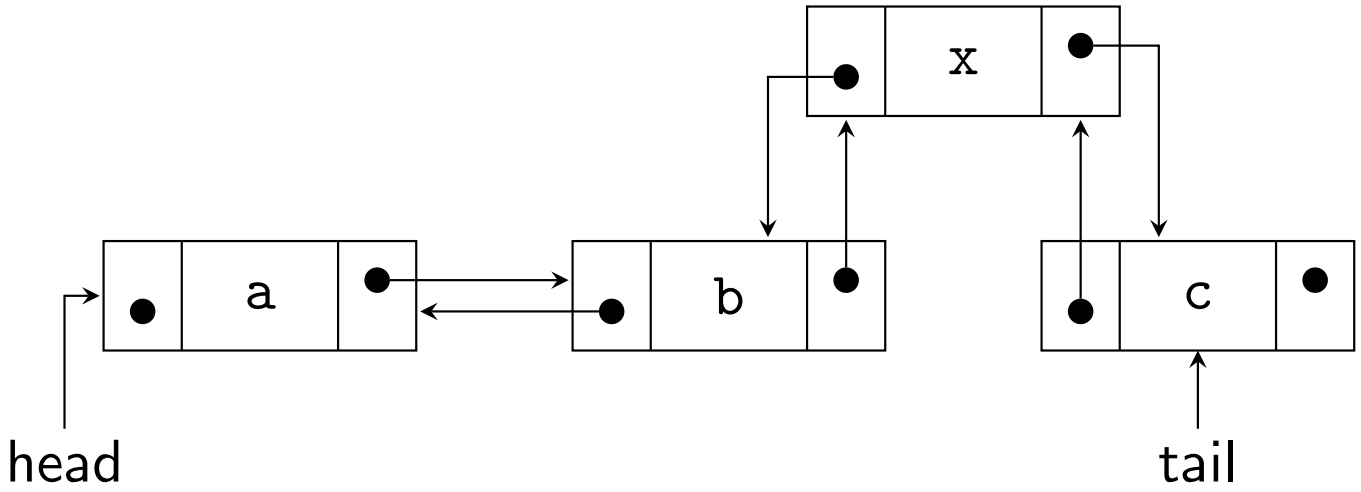


Inserting and deleting in the middle



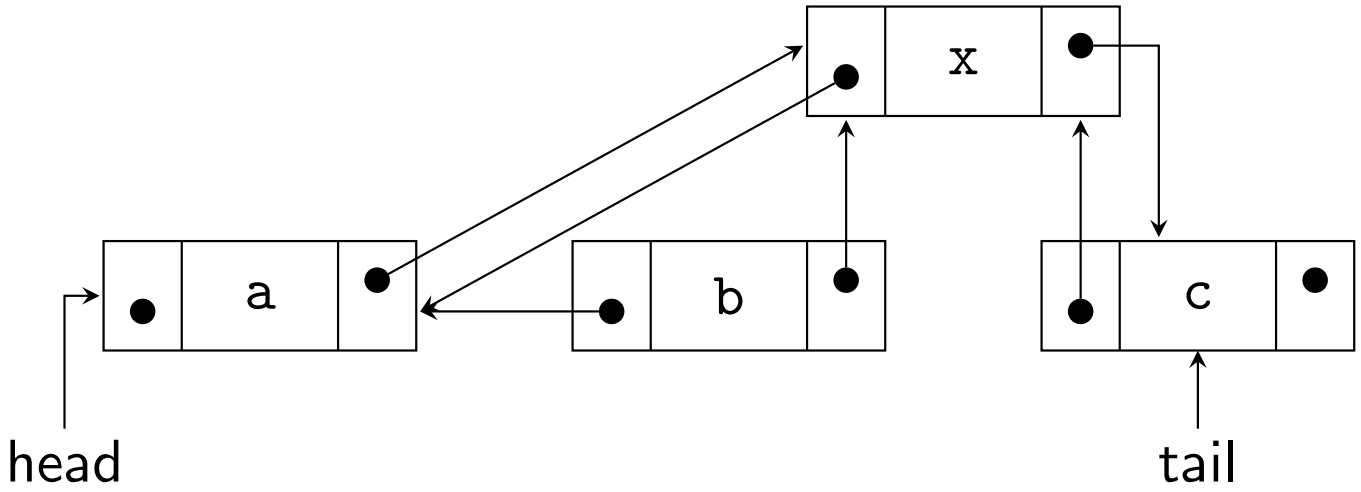

```
insert (2, "x");
```

Inserting and deleting in the middle



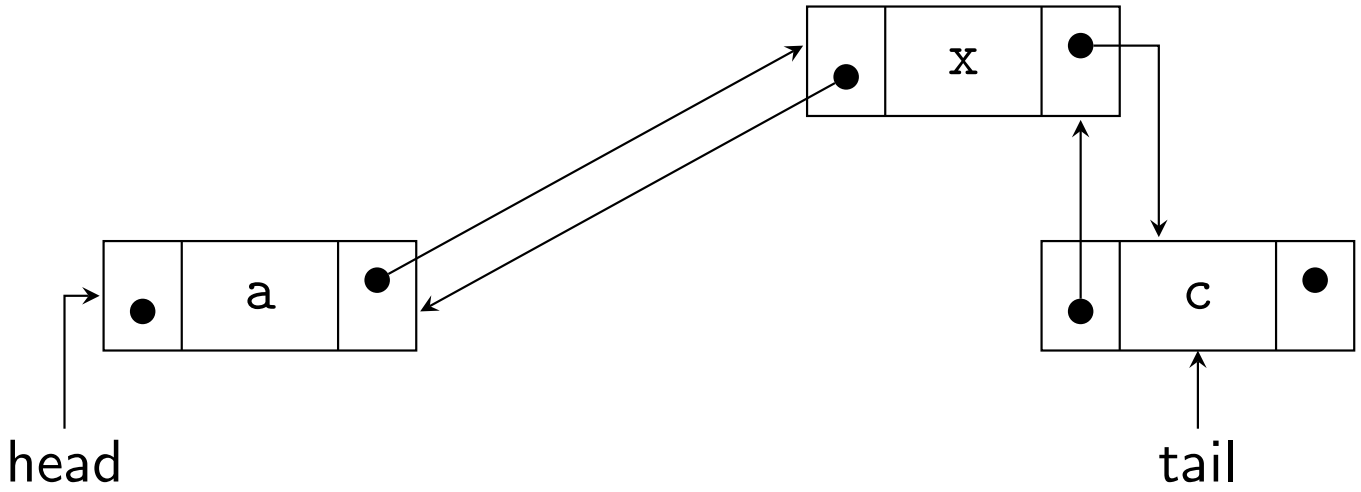

```
insert (2, "x");
```

Inserting and deleting in the middle




```
insert (2, "x"); delete (1);
```

Inserting and deleting in the middle




```
insert (2, "x"); delete (1);
```

insert()

```
public void insert(int index, String value) {
    if (index < 0 || index > length)
        return;

    if (index == 0)
        insertAtHead(value);
    else if (index == length)
        insertAtTail(value);
    else {
        Item cur = getItem(index-1);
        Item e = new Item(value, cur, cur.next);
        e.next.prev = e;
        e.prev.next = e;
        length++;
    }
}
```

delete()

```
public void delete(int index) {
    if (index < 0 || index >= length)
        return;

    if (index == 0)
        deleteHead(value);
    else if (index == length-1)
        deleteTail(value);
    else {
        Item cur = getItem(index);
        e.next.prev = e.prev;
        e.prev.next = e.next;
        length--;
    }
}
```

List iteration (1)

Code like the following is very often needed

```
for (int i = 0; i < list.length(); i++) {
    String s = list.get(i);
    // Do something with s.
}
```

For a list of length n, this makes n calls to `getItem()`.

This makes $0+1+\cdots+n-1 = \frac{1}{2}n(n-1) \approx \frac{1}{2}n^2$ total steps along the list.

Very expensive: nearly 5 000 steps for a 100-item list; nearly 500 000 steps for a 1 000-item list

List iteration (2)

Solution: add a "current" item of the list (a.k.a. "cursor").

```
private Item cur = null;

public void rewind () { cur = null; } /* reset to start */

public String getNext () {
    if (cur == null) cur = head;
    else cur = cur.next;
    return cur == null ? null : cur.value;
}

public boolean hasNext () {
    if (cur == null) return head != null;
    else return cur.next != null;
}
```

List Iteration (3)

Now we can efficiently iterate through our list:

```
list.rewind ();
while (list.hasNext ()) {
    String s = list.getNext ();
    // Do something with s.
}
```

This only takes $n - 1$ steps along the list.

Can similarly implement `hasPrev()` and `getPrev()` to go backwards (not in a singly linked list).

Can also implement methods to insert at cursor position, and delete the cursor position.

Java standard libraries

Two implementations of lists: `ArrayList` and `LinkedList`.

`ArrayList` stores entries in an array

- `get` is fast (constant time);
- insertion and deletion are slow (time $\propto$ length);
- append is fast (constant time) on average, like array stack.

`LinkedList` is a doubly linked list.

`ListIterator` interface and `List.listIterator()` method define iterators. Also allow insertion, deletion, etc. at current position.

Lists: summary

- Like growable arrays with insertion and deletion.
- Random access typically expensive.
- Efficient iteration possible.
- Different implementations suited to different purposes.