# COMP2054-ADE

# ADE Lec06
# Linked Lists

Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

## Note

- This short set of slides is just a quick introduction to linked lists, and not everything you need to know.
- Hopefully, most of it is revision from 1st year (or before).



This is just a quick lecture on the topic of linked lists. It assumes that you have met linked lists before – and so most will be revision.

# Relevance

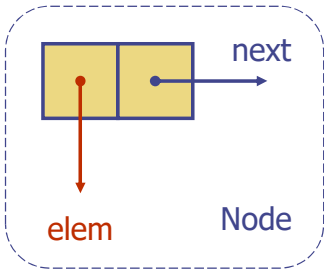
- It is generally assumed that you will have done singly and doubly linked lists
  - Please revise them, if not then these slides just cover the basic ideas
- Our goal in this module is to be more careful about their efficiency, that is, to observe the complexity, O(1) versus O(n), of the various standard operations
  - Where 'n' is the length of the list
- Also, they are relevant to "simple sorting" algorithms, and also to "stacks" and "queues"

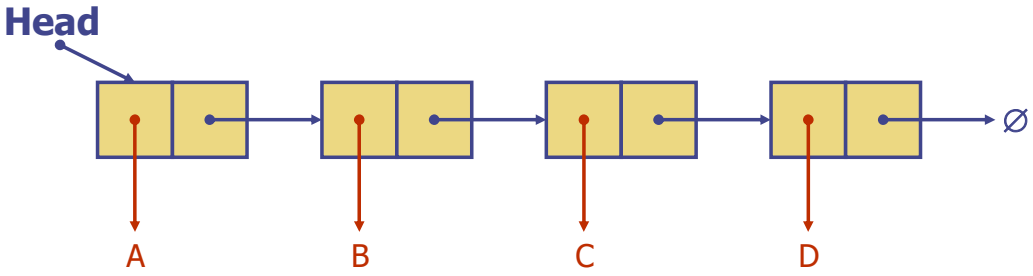
COMP2054-ADE Linked Lists 3

But this time the focus will be on the efficiency of operations in terms of the to the length of the list.

# Recap: Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element e.g.
    - Reference to an Object
    - A primitive date type (int,…)
  - "link": a reference (pointer) to the next node




COMP2054-ADE Linked Lists 4

Firstly, as you should know, a list is just a set of nodes.
Each node contains some data – or more typically a pointer to it,

Then there is a pointer to the next node.
We also have to keep a general pointer to the head of the list – otherwise we would lose it.

# A Node Class for List Nodes

The relevant code usually looks like:

```
public class Node  {
   // Instance variables:
   private  Object element;
   private  Node next; // reference ("pointer") to a 'Node'

   /** Creates a node with the given element and next
   node. */
   public  Node(Object e,  Node n)  {
        element  =  e;
        next  =  n;
   }
```



As an example, the code might look something like this.
Usually we store objects, and so 'e' is an object reference.

# The Node Class for List Nodes

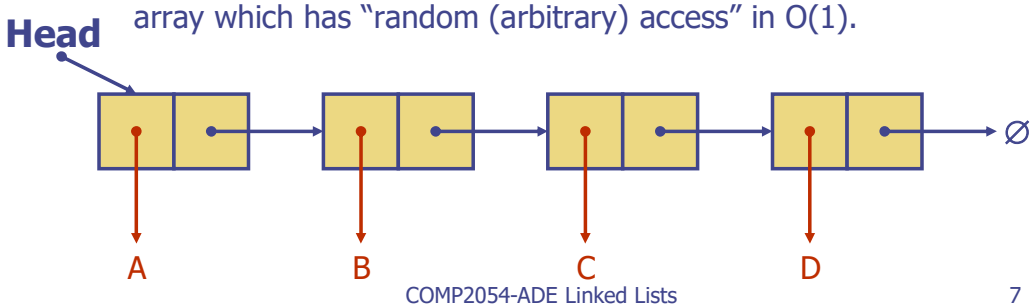
```
// Accessor methods:
 public Object getElement() {
   return element;
}
 public Node getNext() {
   return next;
}
 // Modifier methods:
 public void setElement(Object newElem) {
     element = newElem;
}
 public void setNext(Node newNext) {
     next = newNext;
}
} // end of class Node
```

COMP2054-ADE Linked Lists 6

And a bit more code, that should be ‘obvious’.

# Usage?

- In a simple linked list, all the data is accessible from the head by just "walking along the list"
  - Hence, it is clear (hopefully) that all "standard" operations (insert/delete etc) are implementable
- The key question is what operations are implementable **efficiently**!?
  - Need O(.) of operations in terms of the length n of the list
    - Note: Typically the overall length of a list (or other data structure) is stored in an auxiliary element to allow fast access as it is generally used a lot.
  - Note: There is no direct access to the middle of a list - unlike an array which has "random (arbitrary) access" in O(1).




COMP2054-ADE Linked Lists 7

How do we use them. Given the length of the list in n, then what is the big-Oh of various operations in terms of n ?

(For speed, an implementation will generally also directly store the value of n, and update it as needed.
Note that access is quite different from an array.

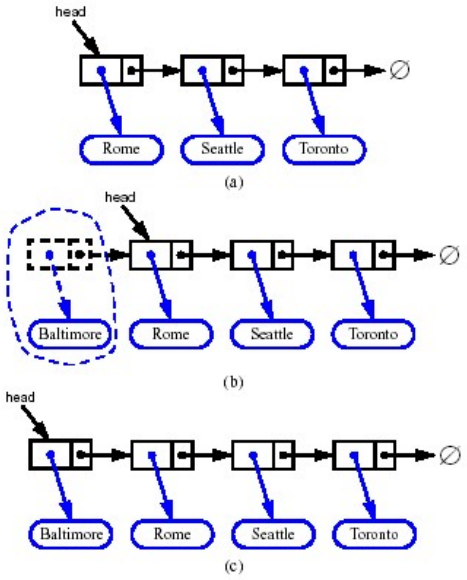
The same typically applies to other data structures as well, as algorithms often want to know n)
In an array we have "random access" in O(1) – getting A[p] has a constant cost.
Note that when saying "random" we really mean it in the sense of "arbitrary" – it is not anything to do with random numbers!
In a list we may need to walk along and so it may be different.
So lets do some basic essential operations.
We will look here at just add/delete from the head or tail of the list.

# Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node

What is the complexity (with n elements in list)?

- Answer: O(1)
- Very efficient!



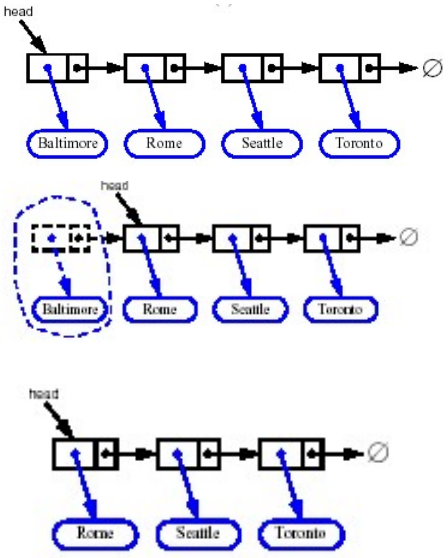

COMP2054-ADE Linked Lists 8

How do we add at the head.
We just create a new node.
Make it point to the old head, and then just make the new head point to the new node.
This is clearly O(1).
It cannot depend on n at all, because the operation never looks at the rest of the list.

# Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node
    1. Or do explicit free in C/C++

Again, the operation is O(1), and so efficient.




9

The removal at the head is also fast.
The head needs to be update to point to the next, but we can easily do this by just walking to the next node.
Again, the operation has no need to see the rest of the list and so the big-Oh cannot depend on n, hence is O(1).

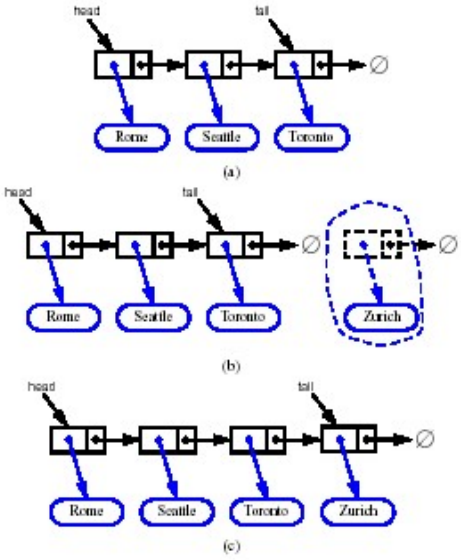
# Inserting at the Tail

For operations at the tail, we also assume keeping a pointer 'tail' to the last element. (See the figure).

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

Complexity: O(1)

Without the tail pointer this would be O(n)




COMP2054-ADE Linked Lists 10

To insert at the tail, we will assume that as well as the "head" we also have and maintain a pointer to the end of the list.

Without this, then accessing the tail is clearly O(n) as we have to walk the entire list.

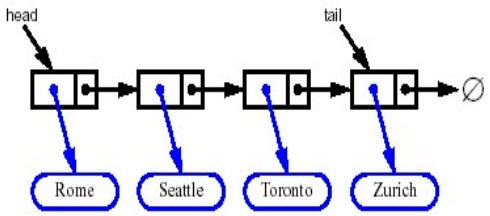
Then adding a new entry as the tail is easy.

In particular, the tail can be updated to point to the new end of the list.

This does not need any other access, and so is O(1).

# Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- To find new tail we must walk the list from the head
  - There is no constant-time way to update the tail to point to the previous node
  - Exercise: Why not keep a "pre-tail" pointing to one before the "tail"? ("Toronto" here)
  - Advanced optional: look up "skip lists"
- Complexity: O(n)




COMP2054-ADE Linked Lists 11

However, removing at the tail is more difficult, because we cannot efficiently access the "last but one entry" in order to make it the last entry.

The only way to get to "Toronto" is to walk from the head of the list, and this will hence take O(n).
A common thought is "Why not also keep a pointer to the last but one".
Suppose we call it pre-tail. Then we would be able to update tail to be the pre-tail and remove the last node.
But then we are again stuck because we have no efficient way to update the "pre-tail".
If we assume that at the start pre-tail satisfies a constraint "a contract" that is points to the last but one, then we must ensure the same contract/constraint is satisfied when we have finished.

ASIDE: There are data structures called "skip lists" that maintain more pointers and allow larger jumps.

# A SinglyLinkedList Class

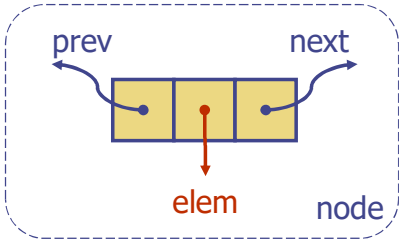
```
class SinglyLinkedList {
    private Node head;

    public SinglyLinkedList() {
    } // head automatically set to null by Java

    public void insertAtHead(Object newElem)  {
        Node newHead = new Node(newElem,head);
        head = newHead;
    }
// etc
}
```

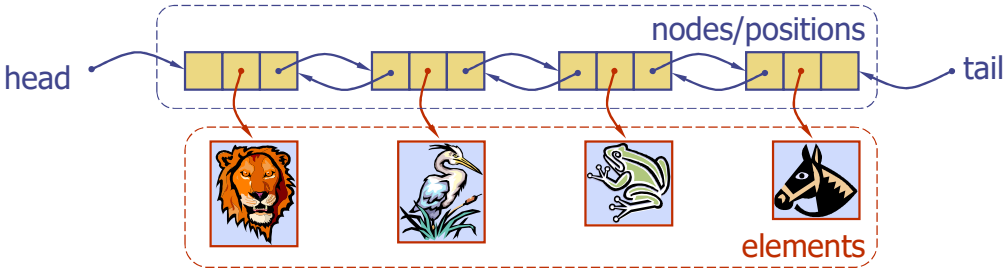


Some example of what code might look like.

# Doubly Linked List

- A doubly linked list provides a natural extension of a singly linked list
- Nodes store:
  - element
  - link to the next node
  - **link to the previous node**
- **Deletion at the tail is now O(1)**
- **But uses more memory**




COMP2054-ADE Linked Lists 13

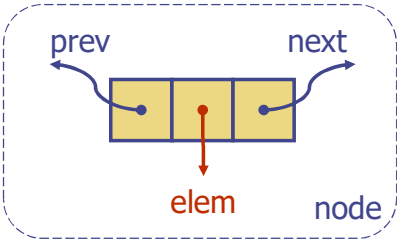
An easier fix to the problem of not being able to "walk backwards" is of course to add pointers to allow walking backwards.
This is easy to do.
It will clearly mean that removal at the tail also becomes O(1).
The doubly-linked means that the node now stores 2 pointers not 1, and so memory usage is increased.
Besides risking running out of memory, it might also need more energy.
Also the increase in memory might increase the rate of cache misses.

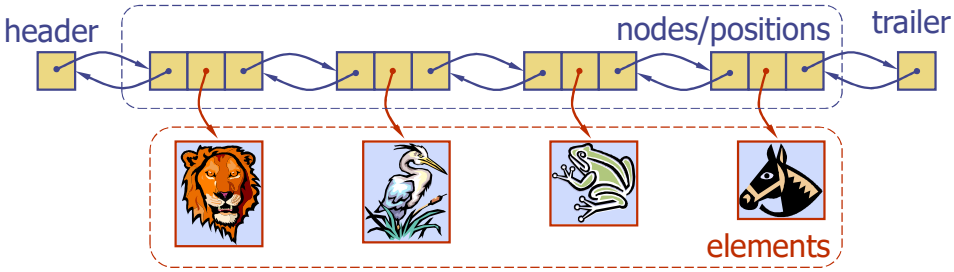
Also, there are more pointers to maintain, and so it can slow down.

This might be important.
So, in general, a design decision should be taken as to whether to use singly or doubly linked lists.

# Doubly Linked List (version 2)

- A doubly linked list provides a natural implementation of a List
- Nodes implement "Position" and store:
  - element
  - **link to the previous node**
  - link to the next node
- Special trailer and header nodes for convenience (an alternative)



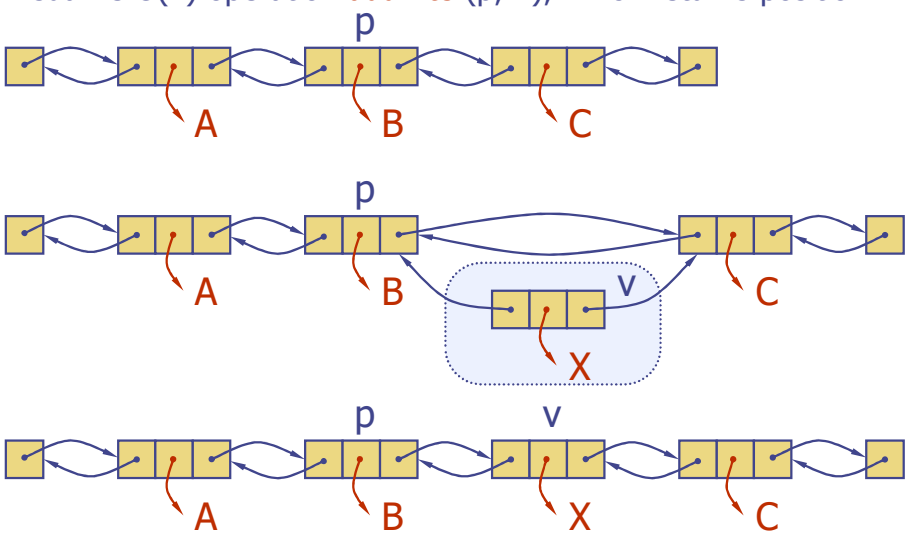




COMP2054-ADE Linked Lists 14

Sometimes, to make the code neater, one might have special nodes for the head and tail.
This can reduce the times that have to check the head/tail status. But it is just for making the code a little bit more tidy and does not affect efficiency.

# Insertion

- We visualize O(1) operation addAfter(p, X), which returns position v




COMP2054-ADE Linked Lists 15

As well as insert/delete at the head/tail we should consider operations in the middle of the list.
So suppose we are given a "position pointer" p, that points to some node in the middle of the list.
(It might have been arrived at during some other algorithm.)
Then we may want to insert a new entry after p.
This can easily be done in O(1) because all the required operations are just local to p, and we do not need to walk along all the list.

# Insertion Algorithm

**Algorithm** addAfter(*p*,*e*):
Create a new node *v*
*v*.setElement(*e*)
*v*.setPrev(*p*) {link *v* to its predecessor}
*v*.setNext(*p*.getNext()) {link *v* to its successor}
(*p*.getNext()).setPrev(*v*) {link *p*'s old successor to *v*}
*p*.setNext(*v*) {link *p* to its new successor, *v*}
**return** *v* {the position for the element *e*}



In terms of explicit code it looks like this.
Of course, the cutting and re-joining of links is somewhat bug-prone and would need to be done carefully and well-tested.

## Minimum Expectations

- It is generally assumed that you will be familiar with singly and doubly linked lists
- You should be aware of the complexity, O(1) versus O(n), of the various standard operations, and how to implement them.



Hence, overall, this should mostly have been revision. But it is important to be aware of which operations are O(1) and which are forced to be O(n).