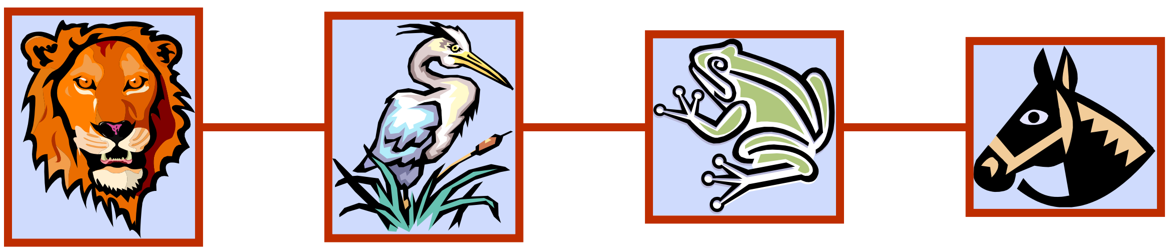
Stacks and Queues by Using Linked Lists



Andrew Parkes

http://www.cs.nott.ac.uk/~pszajp/

G52ACE 2017-18

Objectives

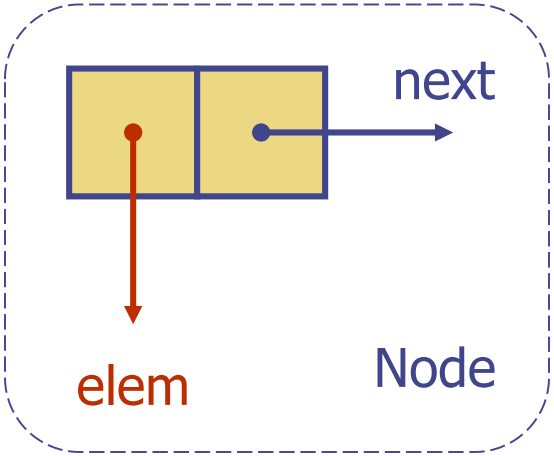
ADT: Abstract Data Type (“interface”)

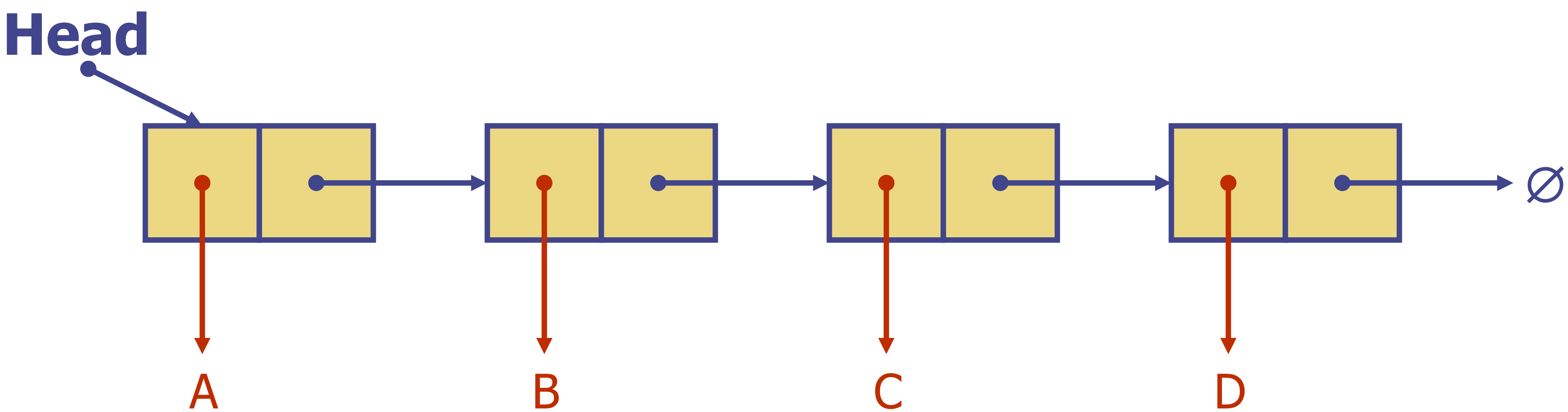
CDT: Concrete Data Type (“implementation”)

- Queues and Stacks are ADTs and so need some choice of CDT.
- Previous lecture showed how to use an array for their CDT
- Here we look at using linked lists as the CDT for Stacks and Queues

Recall: Singly Linked List

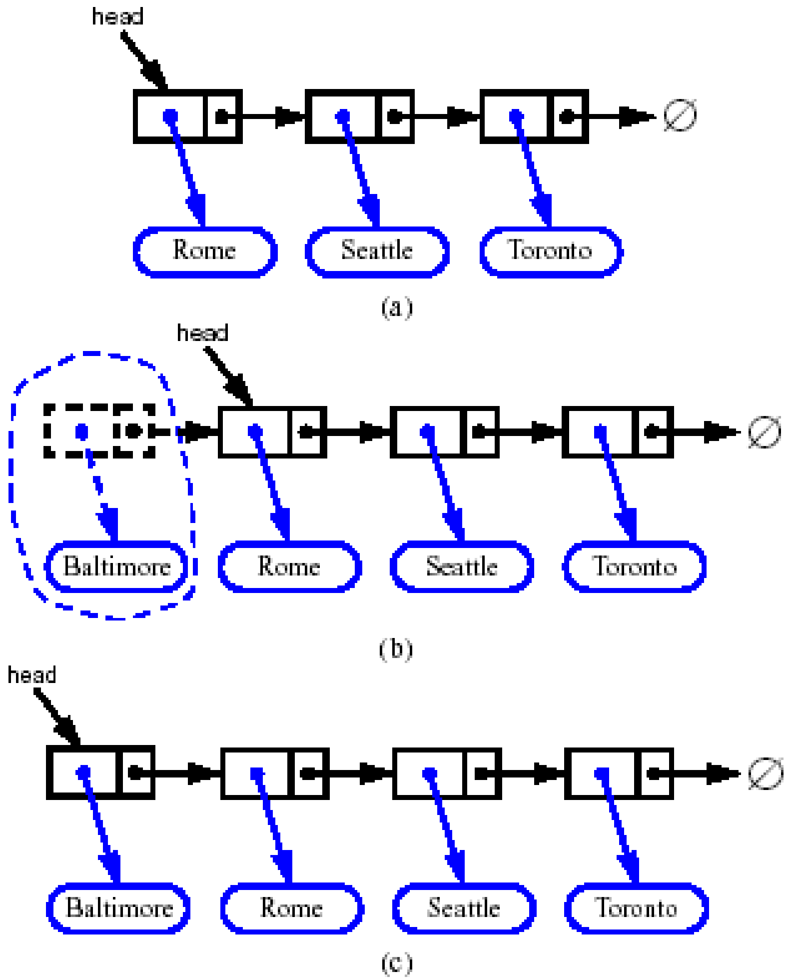
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element (e.g. reference to an Object)
  - link: reference to the next node





Recall: 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



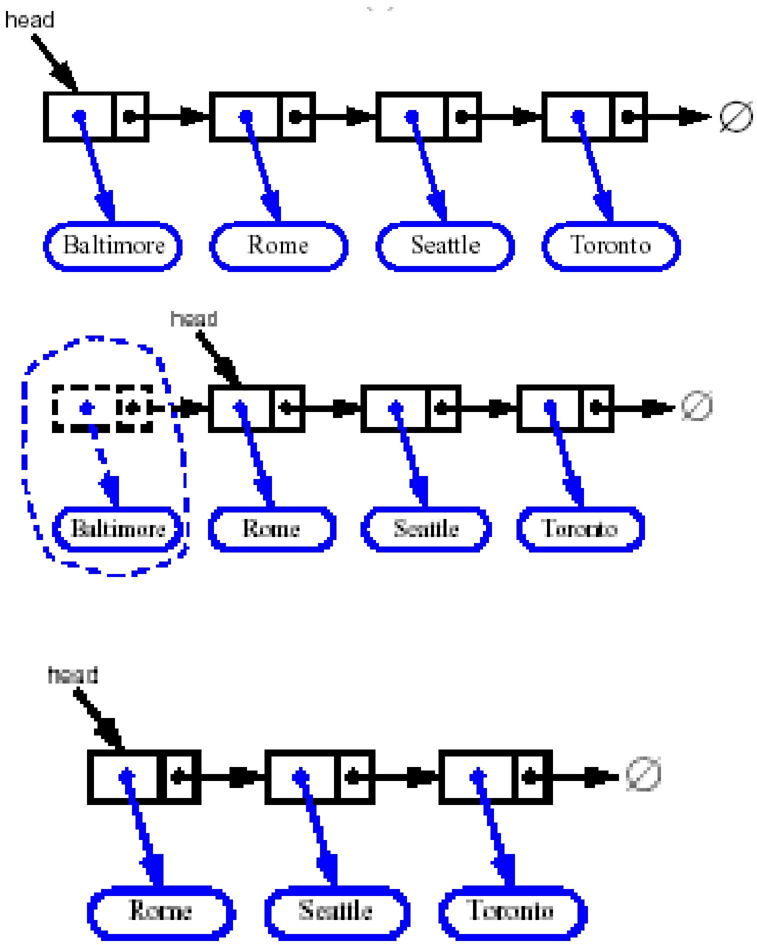
What is the complexity (with n elements in list)?

- Answer: O(1)
- Very efficient!

Recall: 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

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

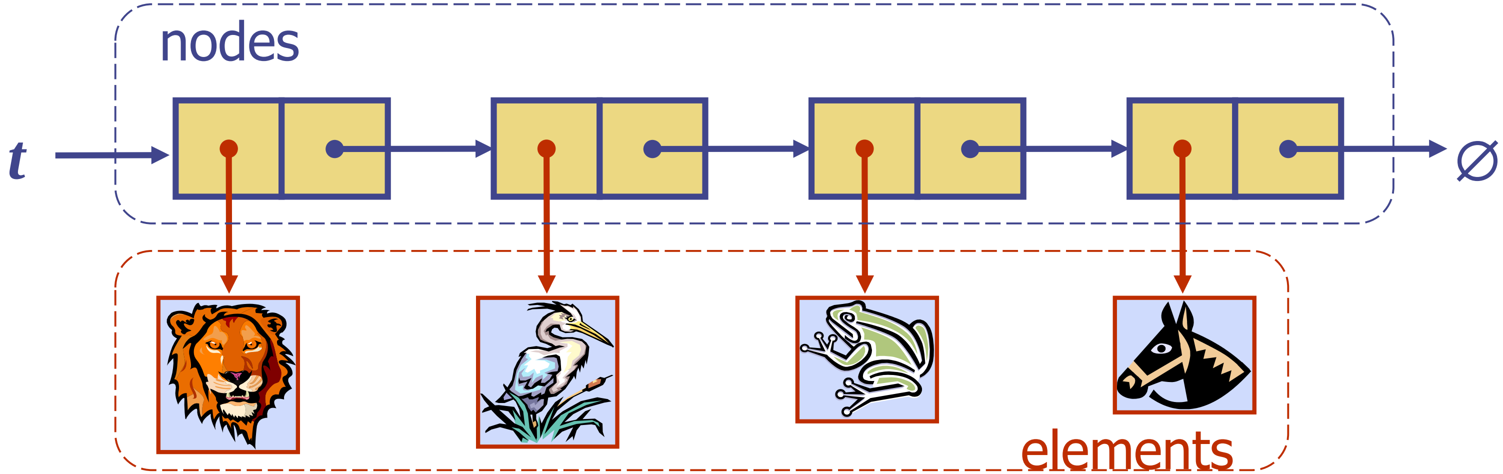


Usage of Simplest Linked List

- In principle, could use them for any ADT
- Observe that insertion and removal at head are very efficient, O(1).
- If insert two elements then remove two elements, we get “Last In First Out” behaviour
- Hence, most natural usage is as a Stack

Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time
- Exercise (online): Spot the deliberate mistake



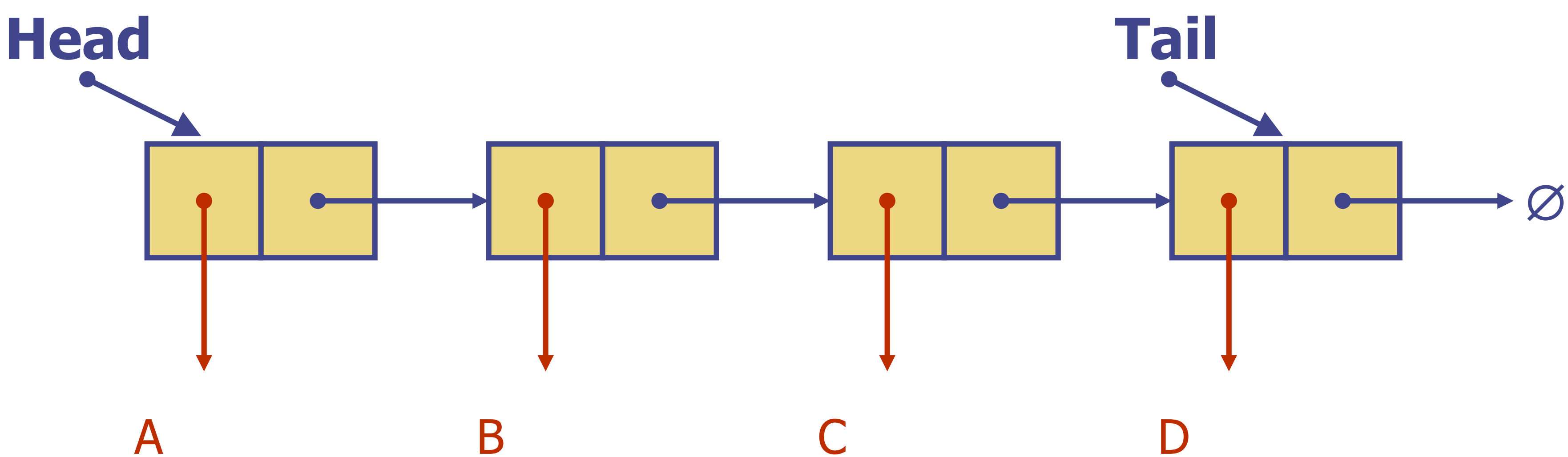
Stack size() operation?

- If given only the “head”, we need to walk the list to find the size()
- Fixes?

1. Take size() out of the Stack ADT
   - most of the time we only use “size() > 0” and so could use !isEmpty() instead
     (i.e. just test if the head is null)
2. In the list implementation, add a size counter
   `private int nbElements;`
   - maintenance and access is O(1) as desired
     (Exercise: verify this)

Linked List usable as a Queue?

- Queue needs access to “both ends”
- To do this efficiently, as well as the ‘head’ we also need to store and maintain a ‘tail’

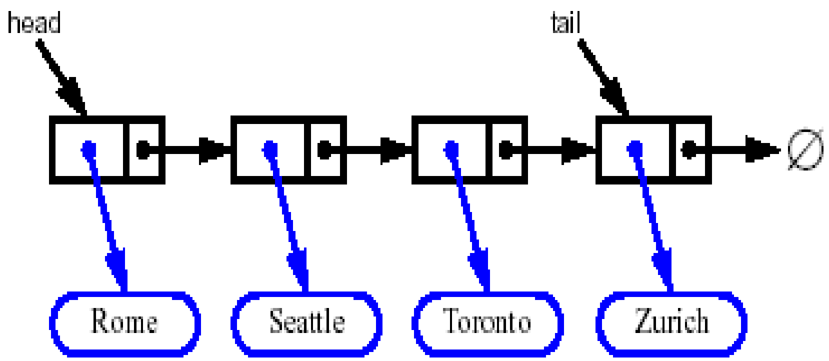


Queues from singly-linked lists

- The arrows (pointers) flow from head to the tail, so
- the new arrivals are placed at the head
- deletions are made at the tail.
- (So that the flow follows the arrows).
- What are the complexities of the operations?
  - already seen that insertion at the head is O(1)

Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- To find new tail have to walk list from head
  - There is no constant-time way to update the tail to point to the previous node
- Complexity: O(n)



Linked Lists:

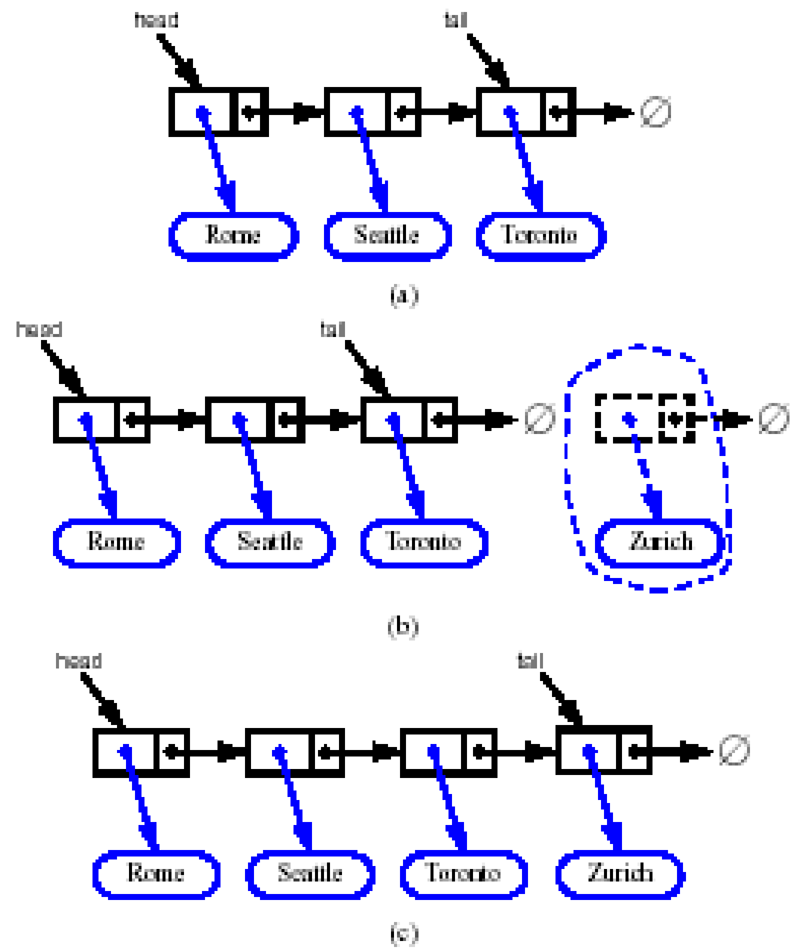
- So far:
  - Insertion at head: $O(1)$
  - Removal at tail: $O(n)$

- Are we done yet? Can we do better?

Inserting at the Tail

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)



Spot the Error!

- “The arrows (pointers) flow from head to the tail, so
- the new arrivals are placed at the head
- deletions are made at the tail.”
- Exercise: where did I ‘lie’ to you!? ☺
- **The flow of items does not necessarily follow the flow of the pointers.**
- (Mnemonic: Sometimes, the locations of queues on motorways move backwards, even though the vehicles move forward.)

Backwards is Better!

- Insert at the tail: O(1)
- Remove at the head: O(1)

- Much better!

Tailed Linked List usable as a Queue?

  - Insertion at head is O(1)
  - Removal at head is O(1)
  - Insertion at tail is O(1)
  - Removal at tail is O(n)
- Could implement Queue by insert at head and remove at tail, but this should get you sacked
- **Notice how the big-Oh analysis helps us make sensible design decisions!**
  - **Notice that did not need all the constant factors; the big-Oh gives a useful level of abstraction**

Linked List vs. Array based CDTs

- Array
  - Con: Fixed size – might need a lot of unused space
  - Pro: Contiguous in memory
    - Localised memory access
    - Gives better use of the (CPU) cache – e.g. machine level "pre-fetch" will tend to do the right thing
- Linked List
  - Pro: size grows and shrinks automatically
  - Con: can be scattered around memory, and give poor usage of the cache
  - Con: storage of the "next" references doubles the space usage

Remarks

- Data structure design is an incremental process
- Can include judicious use of
  - Addition of new fields or Objects to store data that is otherwise expensive to re-compute
  - Removal of fields or Objects when the are not needed
  - Use of big-Oh analysis of operations
- Expectation is that you will know **why** each part of an ADT/CDT is there, and not just that you have memorised them!

Remarks

- Linked lists are a very common structure and all programmers are expected to be totally comfortable
  - using them directly, and
  - using the ideas to build more complex data structures not provided in standard libraries
- Exercise (offline) Implement all these in Java