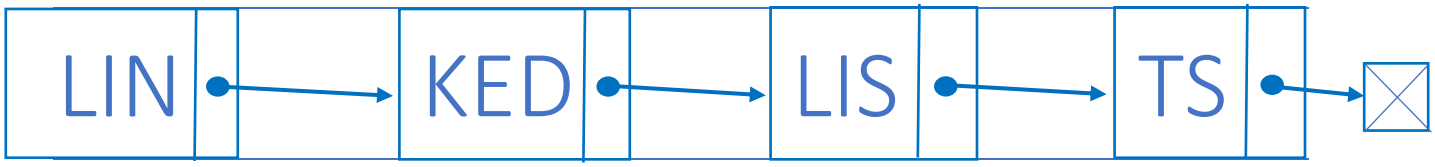
DATA STRUCTURES

AND ALGORITHMS



LECTURE 9

FEBRUARY 19, 2019

COMP20230

Scribed by Marian Barry and Jessica Butler

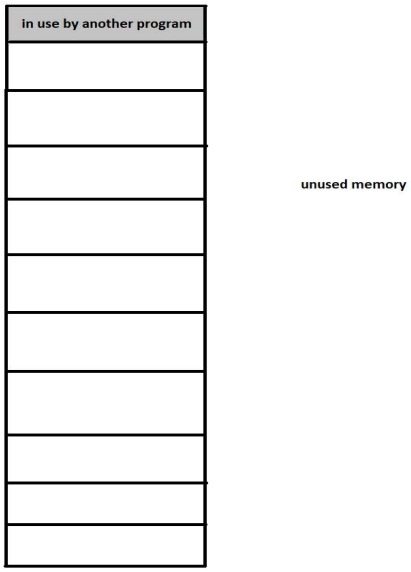
Contents:

Table of Definitions Used

ABBREVIATION	DESCRIPTION
DATA TYPE	Primitive data type, reference data type, string, int, double, float, long, short, char, boolean
ABSTRACT **D**ATA **T**YPE (ADT)	An abstract data type looks at how the different pieces of the problem fit together and what actions you can perform, without going in to the detail of how this is implemented e.g. a sequence or set implemented with an array
ALGORITHM	
DATA STRUCTURE	Different options for implementing your algorithm
ARRAY	Collection of elements of the same data type. It is an option for implementing ADT.
LIST	A collection of elements. Dynamic
SET	A type of list with the rules: no duplicates, is unordered
SEQUENCE	An ordered list
QUEUE	A type of list with the rules: only join at the end, only remove from the start (first in, first out (FIFO))
SINGLY LINKED LIST (SLL)	A list in which all the elements are joined with pointers (like a clue in a treasure hunt to tell you where to go next). Singly linked lists can only move forward through the list, never backwards.
DOUBLY LINKED LIST (DLL)	A list in which all the elements are joined with pointers (like a clue in a treasure hunt to tell you where to go next). Doubly linked lists can move forward and backwards through the list.
FRAGMENTATION	Broken up into smaller bits. Not good for memory as if all the space is split up, can't fit in a large piece of data even though there is space (external fragmentation of memory)
MEMORY	Space for storing things in a computer
OPERATION COMPLEXITY	A calculation of how long the run time will take by looking at how many operations could potentially be carried out in the worst case scenario
TRAVERSING A LIST	the act of moving through a linked list. The next reference of a node acts as a pointer to the next node. Also known as link hopping or pointer hopping
NODE	Also known as an element, or less commonly a data member, it is the item which contains the data, and the pointer(s) at that address in memory – bit like a box
POINTER	The address or arrow that shows you where to go to find the next piece of information in the linked list
ADDRESS	Location in memory

Previous Lecture Recap

In lecture 8, we looked at the first of the Abstract Data Types (ADT), an array. However, certain options such as editing required a lot of operations to make one simple change.



(DOUBLE CLICK PICTURE TO PLAY)

Why Linked Lists?

Arrays can be used to store linear data of similar types, but arrays have following limitations:

- The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

This is not ideal as the number of operations for one action or change will scale greatly as a program grows, and we don't want to waste effort or memory.

ARRAY OPERATIONS	ARRAY COMPLEXITY
size, is empty	O(1)
get elem at rank	O(1)
set elem at rank	O(1)
insert element at rank	**O(n)**
remove element at rank	**O(n)**
insert first, insert last	O(1)
insert after, insert before	**O(n)**

Linked Lists

This lecture looked at another abstract data type, the linked list, and how it differs / compares to an array. A linked list is linear sequential data structure where each element is a separate object in memory. The pointer at the end of each element tells the list where its next item is. The size of the list refers to the total amount of nodes. This avoids the need to traverse the list to count the nodes, but also means individual nodes do not hold an index. The size of a linked list is dynamic, meaning we can add or remove nodes easily.

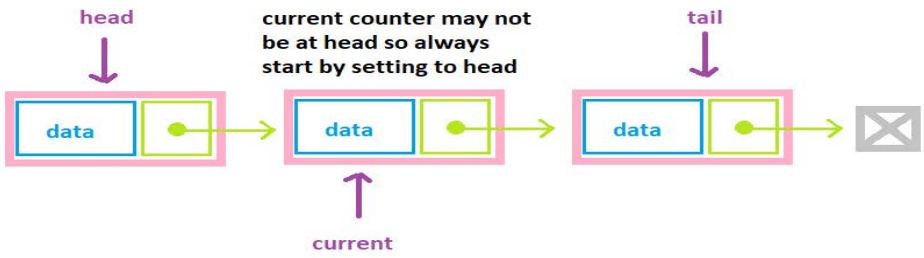
(DOUBLE CLICK PICTURE TO PLAY)

Each element (or node) of a list has two objects: the data (contents) and a reference/address to the next node (pointer). The element itself acts as an object or as a primitive data type (String, integer, instance of a class etc…). The pointer or reference acts as a reference to the next node. These elements are threaded together to form a linked list. The linked list must keep reference to the head of the list. It is common to keep a reference to the tail of the list for convenience. The null terminator denotes the end of the list.

Operations on Linked Lists

Traversing a List

Traversing a list is the act of iterating or moving through a linked list. The next reference of a node acts as a pointer to the next node (known as link hopping or pointer hopping). Linked lists must me traversed starting from the head node until the end of the list. This is always sequential. We always start at the head of the list. The content in the head is then accessed if it is not null. We can then access the next node in the list and access the node information. This continues until we reach the very last node.



(DOUBLE CLICK PICTURE TO PLAY)

Algorithm O(1)

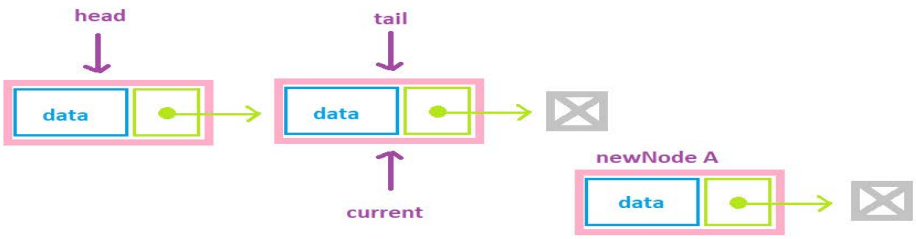
```
List.current = list.head;
while list.current != None/null:
    list.current = list.next()
```

Adding a node:

For linked lists, the list is built “organically” node by node, not created all at once like an array.

Next in the list:

As simple as concatenation, the pointer aiming at null is now redirected to the newly added node, and that new node’s pointer now points to null. Each addition is just modifying the “next” pointer target address.



(DOUBLE CLICK PICTURE TO PLAY)

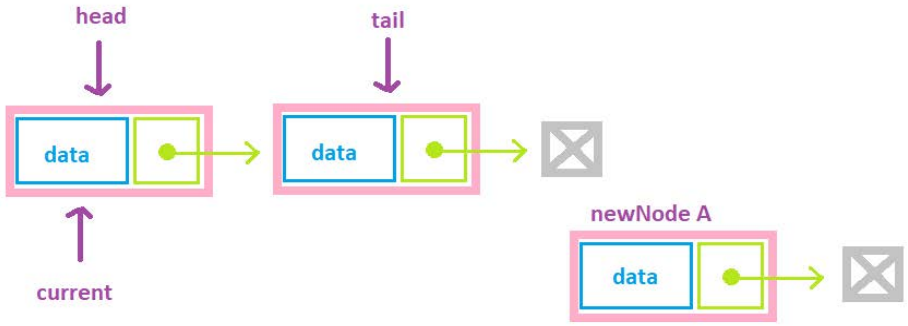
Algorithm O(1)

```
newNode A;

list.next() = A
```

At the end of the list:

When inserting a new node at the end of the list we must first traverse the list and find the last node. The new node is then inserted. Note, we must consider special cases such as the list being empty in this case.



(DOUBLE CLICK PICTURE TO PLAY)

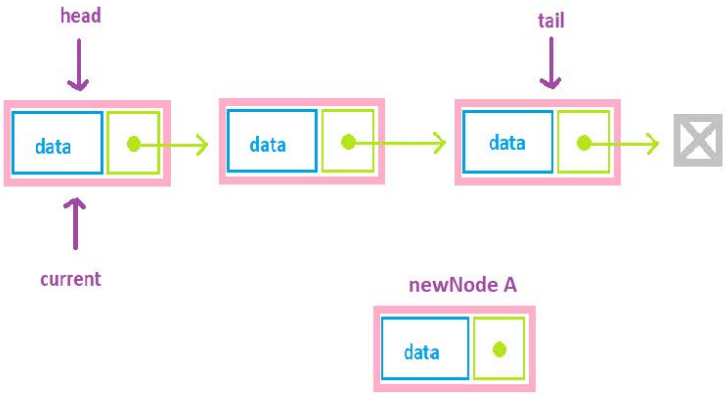
Algorithm O(1)*

```
newNode A;
if list.size() > 0:
    if list.next() == null:
        List.next() = A
    Else:
        List.next()
```

depending on how long the list is, searching for the end/tail has the potential to be O(1) up to O(n), so it is written as O(1)

Somewhere between the head and the tail:

When inserting a new node in the middle of a list we must traverse the list first to find the specific location to insert the node as with the previous operation. We have to maintain a reference to the previous and the next node and then insert the new node between them.



(DOUBLE CLICK PICTURE TO PLAY)

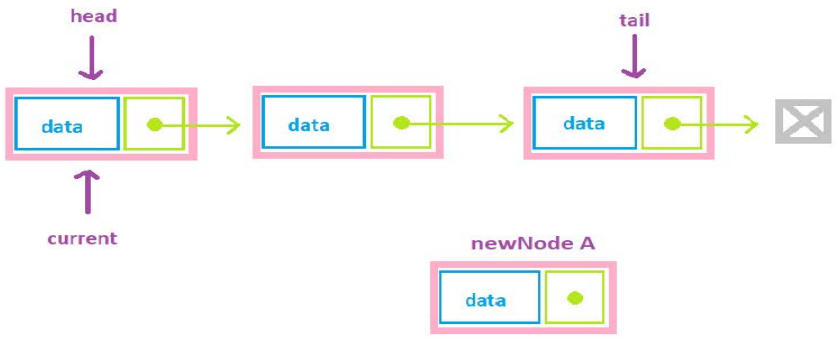
Algorithm O(1)*

```
newNode A, insertAddress;
list.current = head;
while list.next != null && list.next != insert.address:
        previous = current
        current = list.next ()
if list.next == null:
        print('Index not found')
else:
        list.next = A
```

depending on how long the list is, and how many nodes are before the position you want, searching for the end/tail has the potential to be O(1) up to O(n), so it is written as O(1)

At the beginning of the list:

When inserting a new node at the beginning of the list we do not need to traverse the list. If the list is empty the newly inserted node becomes the beginning of a list. Otherwise the node is connected to the original head, and the newly inserted node becomes the head of the list.



(DOUBLE CLICK PICTURE TO PLAY)

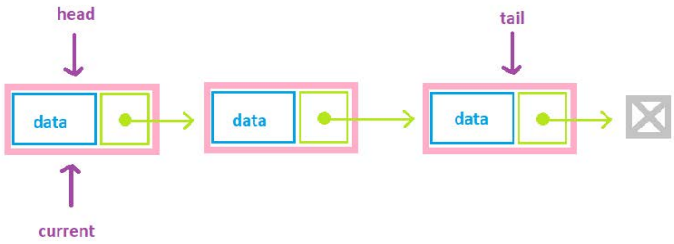
Algorithm O(1)

```
If list.size() > 0:
    List.head = newNode A;
    List.current = list.head;
```

Removing a Node:

Remove the last node

Before removing the last node of a linked list we must consider the following possibilities. The linked list we are working with may just consist of one node, when this is removed we will be left with an empty list. The linked list may also be empty to begin with, so removing nodes would not be successful. To remove a node from the end of a list, the last node must be unchained pointing the second last node to None/null and making it new last node. You cannot go backwards in a singly linked list (only forwards, i.e. next), so you need to work through the list and keep track of where you are with a pointer.



(DOUBLE CLICK PICTURE TO PLAY)

Algorithm O(1)*

```
If head = NULL then
    Write “Linked List is Empty”
 If head->LINK = NULL then
    Return head->INFO
    head=NULL
 Else
    SAVE=head
```

Repeat while SAVE->LINK ≠ NULL

PRED=SAVE

SAVE=SAVE->LINK

Return SAVE->INFO

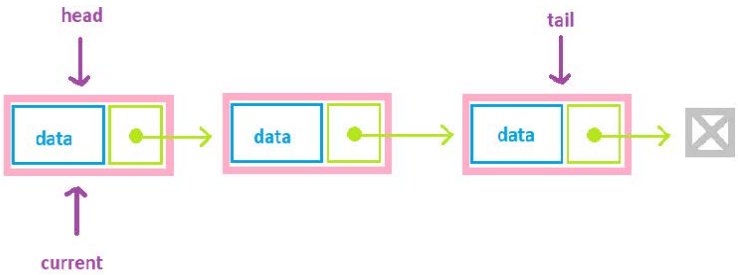
PRED->LINK=NULL

Exit

Remove a node from somewhere in the middle

Before removing a node from the middle of a linked list we must consider the following possibilities. The linked list we are working with may just consist of one node, when this is removed we will be left with an empty list. The linked list may also be empty to begin with, so removing nodes would not be successful.

Firstly, we must establish where this node is located. This will be done by traversing the list in order to find the specific place of the node. When the node is located, the node and the pointer will be removed. Then the node located before the node specified for deletion will have its pointer updated.



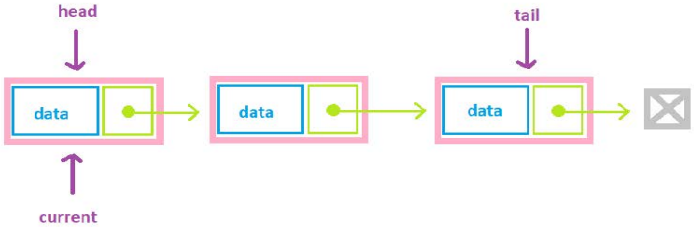
(DOUBLE CLICK PICTURE TO PLAY)

Algorithm O(1)*

```
head = head
curPos = 1
if curPos == P || head == NULL
        break
prevNode = head
head = head->Next
curPos++
if head != NULL:
        prevNode->Next = head->Next
        free head
```

Remove a node at the beginning

There are a number of possibilities to consider when removing the first node of a linked list. Firstly, we must consider the possibility that there is no head node, in this case we cannot remove anything as the linked list is empty. If the linked list consists of just the head then removing the node will result in an empty list. When the head is established as being in linked list with multiple nodes we can begin the process of removing the current head. First we establish where the head of the list is located, then we remove the node and the pointer.



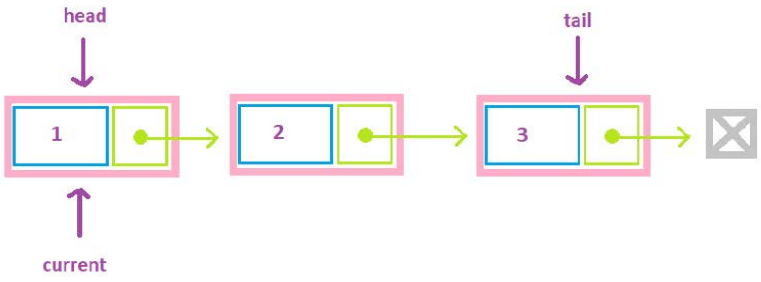
Algorithm O(1)

```
if List.head == None/null:
        print('Error : List is empty')
else:
        List.head = List.next;
```

Get a particular node:

Search using 'current' pointer Get = get element at, or get element next

As linked lists do not allow us to directly access a specified node, in order to search a linked list we must first traverse the list. We start at the head node and will compare the current position to the specified node we are searching for. When the correct node has been located, we will set the count in the algorithm below to 1 to indicate success.



(DOUBLE CLICK PICTURE TO PLAY)

```
Algorithm O(n)
count = 0
SAVE=head
while SAVE ≠ NULL:
    If SAVE->INFO = X then
        count = 1
        SAVE=SAVE->LINK
    Else
        SAVE=SAVE->LINK
        If count = 1 then
            Write “Search is Successful”
        Else
            Write “Search is not successful”
        Exit
```

Operations on Arrays versus Linked Lists

Operation	Array	Linked List
size, is empty	O(1)	O(1)
get elem at rank	O(1)	O(n)
set elem at rank	O(1)	O(n)
insert element at rank	O(n)	O(1)*
remove element at rank	O(n)	O(1)*
insert first, insert last	O(1)	O(1)
insert after, insert before	O(n)	O(1)

*** “potential search + operation”

Advantages and Disadvantages of Linked Lists

Advantages	Disadvantages
No predetermined size	More memory is required
Insertion or deletion is easy	We can not randomly access any elements
Memory is only allocated when required - there is no memory wastage.	As there are no back pointers in singly linked lists, accessing passed nodes is difficult.

Applications for linked lists

- Like arrays, linked lists can be used to implement stacks or queues, however linked lists also offer the implementation of hash tables.
- The CPU uses a circular linked list to manage the queue of programs moving from the running state to the ready or waiting states, and vice versa.
- Undo and redo functionalities are also commonly implemented using linked lists.

Other types of linked lists

Ordered vs unordered Singly Linked Lists (SLL)

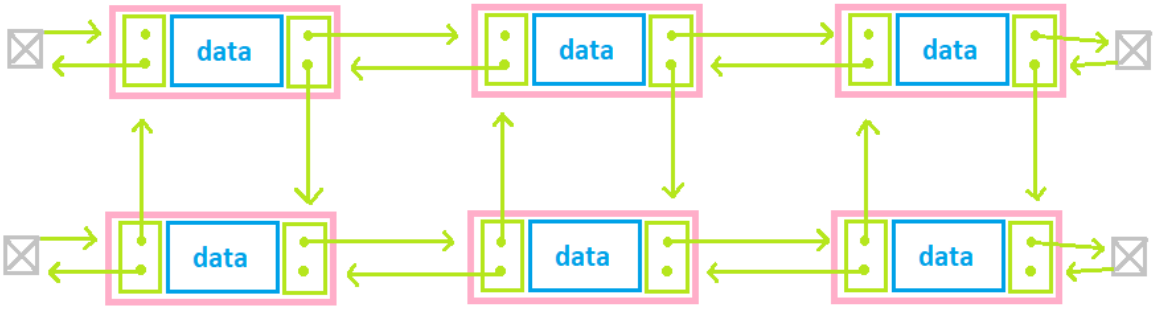
In a sorted list you can stop searching earlier as soon as the value to be found is less/greater than the current node's value. Because the sorting order tells you the value to be found cannot be present beyond that point. In case of order linked list time complexity can be reduced while performing search operation. You can implement skip list for this purpose. But if your linked list need to be a singly linked list strictly then there is no difference other than already stated.

Doubly Linked Lists (DLL)

A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list, which removes some unnecessary operations having to re-traverse a list to access a previous node, or saving the previous node location to a variable.

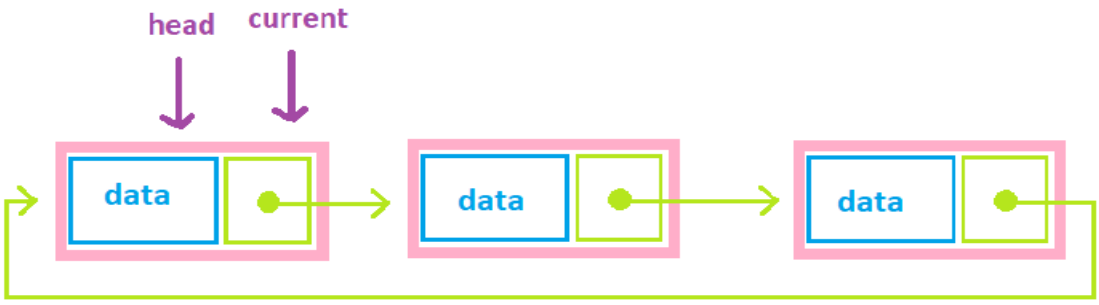
Triple Linked Lists

If you consider singly and doubly linked lists as moving left and right across the x axis, then a triple linked list is similar but also has the capability to move up and down levels on the y axis.



Circular linked lists

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Skip Lists

A skip list is a data structure that allows O(1) search complexity as well as O(1) insertion complexity within an ordered sequence of n elements. Thus it can get the best of array functionality while maintaining a linked list-like structure that allows 0(1) insertion (which is not possible in a normal array). Fast search is made possible by maintaining a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one.

More Materials on Linked Lists

Playlist on linked lists:

https://www.youtube.com/playlist?list=PLqM7alHXFySH41ZxzrPNj2pAYPOI8ITe7

Sample Exam Questions

2015/2016 question 4, C

(i) Provide a definition of Abstract Data Type (ADT) in your own words.

ADT's or abstract data types describe the operations on a data type. They are independent of programming languages and exist as tools to help us work with different types of data.

(ii) What are some of the differences between arrays and linked lists?

Arrays are an indexed based data structure. Each element of an array is associated with an index and has a fixed size. The data is ordered consecutively and can be directly or randomly accessed. The data stored in an array must one data type only i.e. all integers, floats or characters.

Linked lists are a data structure that relies on references. Each node consists of data and references the next node. Unlike the array, there is no need to specify the size of a linked list. The linked list can also be updated by inserting new nodes, deleting nodes or updating the head or tail nodes. This allows the list to grow or shrink. To access the data in a linked list we must traverse the list starting from the first node, going through the list until we reach the required information.

(iii) Provide examples of which show their different behaviours when representing ADTs.

A linked list ADT allows us to traverse the linked list, update nodes or insert or delete nodes. It is flexible and can expand or shrink according to user needs, users can also specify where new nodes are inserted. In comparison to an array, accessing a specific element in the linked list takes more time since we must traverse the list to access the data.

Array ADT's allow us to access specific elements that are associated with an index, this is process is notably faster than linked lists. Arrays are limited in size according to the size defined in its set up.

2016/2017 question 4, C

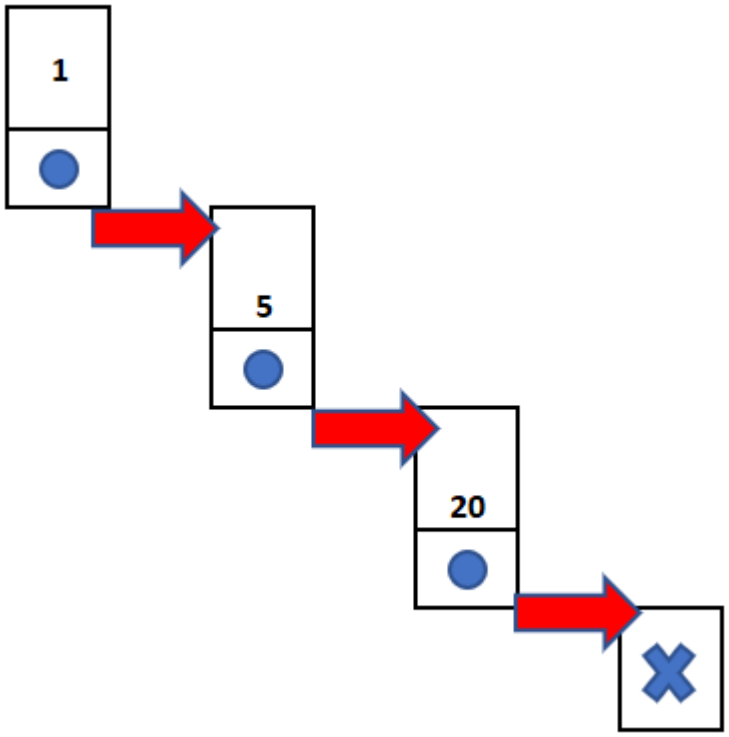
(v) Describe in your own words how a Stack ADT can be represented using a linked list. You can illustrate your description with figures.

A Stack ADT can be implemented using a linked list, this is a data structure that links together individual data objects as if they were links in a chain of data.

An empty stack consists of the head and the null terminator.

Stack operations are based on the LIFO (last in first out) principle.

Each time an object is pushed a new element is constructed and then linked to the head of the chain of elements.



(vi) Give the pseudo-code implementation of the operations listed above assuming a linked list as input.

L is an empty linked list.

Before every push to the linked list we will check if the list is empty. This will enable us to determine the head of the list.

The new node (N) will be pushed to the list.

N will then point to the old head of the list and is now the new head.

As stacks are LIFO (last in, first out) we will always remove the head of the list. Before popping the head node we will check to see if the list is empty.

The position of the head will be determined and then popped if necessary.

Quiz

1) Why is a linked list different from an array?

a) A linked list can handle more types of information than an array
b) An array cannot be sorted but a linked list can be
c) An array is fixed in size, but a linked list is dynamically sizable

2) What is the purpose of using an ADT for a linked list?

a) To unnecessarily complicate the design
b) Using an abstract data type is the only method for building linked lists
c) To force a specific set of rules onto the behaviour of the data stored

3) In addition to the info and link components, the linked list must also contain what other components?

a) Sorting information about the list
b) Head and tail pointers to the first and last nodes
c) The current node that was last accessed

4) What does the following fragment of code do with a linked list?

```
current = head;
while (current != null) {
    current = current.link;
}
```

a) It initializes the list
b) It counts the number of items in the list
c) It traverses the list

5) A singly linked list can be traversed in both directions

a) True
b) False

6) Lists allow random access

a) True
b) False

7) What is the difference between building a linked list forward and backwards?

a) The forward list uses the unordered linked list and backwards uses the ordered linked list
b) Nothing, only the insertion of the information at the head or tail of the linked list
c) The head and tail are reversed definitions in the backwards list

8) Which data structure is more efficient at retrieving information?

a) A linked list
b) An array
c) Neither is more efficient over the other

9) The 'insert' operation on a linked list is linear

a) True
b) False

10) Why is the removing operation on a linked list O(1)* and not O(1)

a) It depends on how far it needs to traverse the list first before it can perform the removal
b) The removal operation is more complex than the others
c) The removal operation may leave behind 'garbage' as an unaddressed node

Answers:

1=c, 2=c, 3=b, 4=c, 5=b, 6=b, 7=b, 8=b, 9=b, 10=a

References / Bibliography

As the facilitative approach of these notes was designed to support different learning styles in a simple manner, few references were used outside of the online lecture notes and notes taken in class. The hope is through this approach, it will be easier to approach further reading on the subject, and support anyone who may have missed this lecture with the fundamentals.

Ordered vs Unordered Lists: https://stackoverflow.com/questions/36905569/searching-ordered-and-unordered-linked-lists

Doubly Linked Lists: https://www.geeksforgeeks.org/doubly-linked-list/

Circular linked lists: https://www.geeksforgeeks.org/circular-linked-list/

Skip lists: https://en.wikipedia.org/wiki/Skip_list

Quizzes altered from:

- https://www.proprofs.com/quiz-school/quizshow.php?title=linked-list&q=20
- https://quiz.virtuq.com/quizzes/generate/practical-data-structures/linked-lists-and-dynamic-arrays