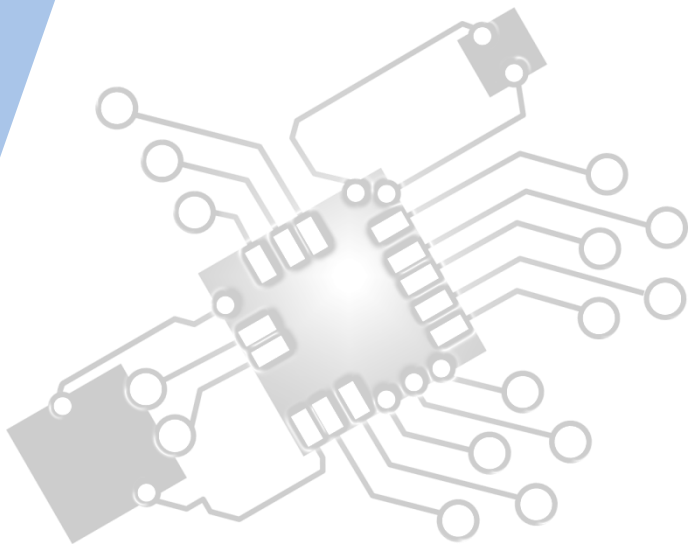
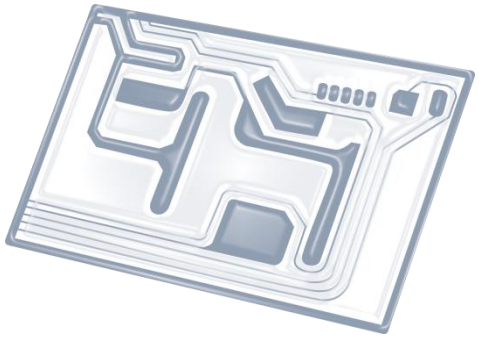
Abstract Data Structures

IB Computer Science



Content developed by
Dartford Grammar School
Computer Science Department

HL Topics 1-7, D1-4



1: System design

2: Computer Organisation

3: Networks

4: Computational thinking

5: Abstract data structures

6: Resource management

7: Control

D: OOP

HL only 5 Overview

Thinking recursively
5.1.1 Identify a situation that requires the use of recursive thinking
5.1.2 Identify recursive thinking in a specified problem solution
5.1.3 Trace a recursive algorithm to express a solution to a problem
Abstract data structures
5.1.4 Describe the characteristics of a two-dimensional array
5.1.5 Construct algorithms using two-dimensional arrays
5.1.6 Describe the characteristics and applications of a stack
5.1.7 Construct algorithms using the access methods of a stack
5.1.8 Describe the characteristics and applications of a queue
5.1.9 Construct algorithms using the access methods of a queue
5.1.10 Explain the use of arrays as static stacks and queues
Linked lists
5.1.11 Describe the features and characteristics of a dynamic data structure
5.1.12 Describe how linked lists operate logically
5.1.13 Sketch linked lists (single, double and circular)
Trees
5.1.14 Describe how trees operate logically (both binary and non-binary)
5.1.15 Define the terms: parent, left-child, right-child, subtree, root and leaf
5.1.16 State the result of inorder, postorder and preorder tree traversal
5.1.17 Sketch binary trees
Applications
5.1.18 Define the term dynamic data structure
5.1.19 Compare the use of static and dynamic data structures
5.1.20 Suggest a suitable structure for a given situation

1: System design

2: Computer Organisation

3: Networks

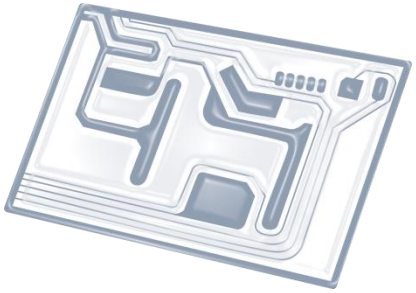
4: Computational thinking

5: Abstract data structures

6: Resource management

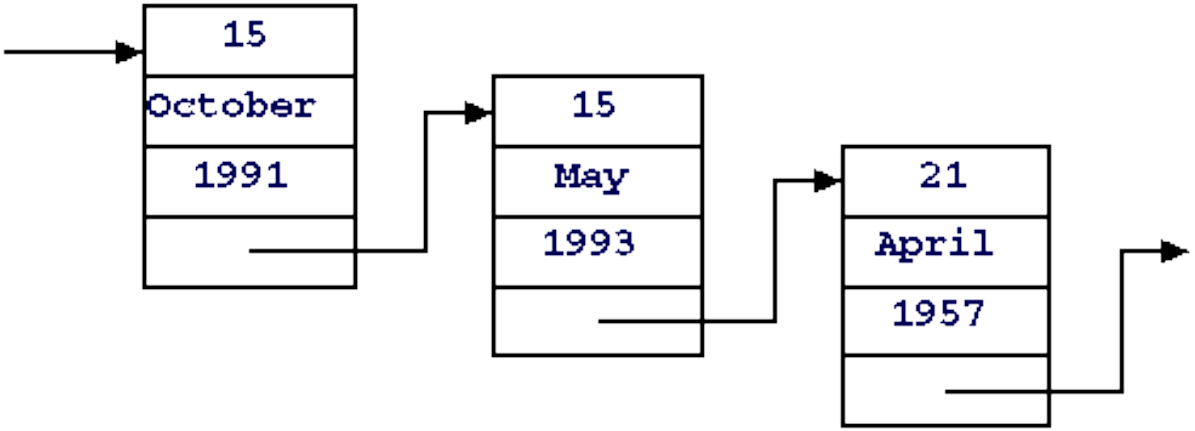
7: Control

D: OOP



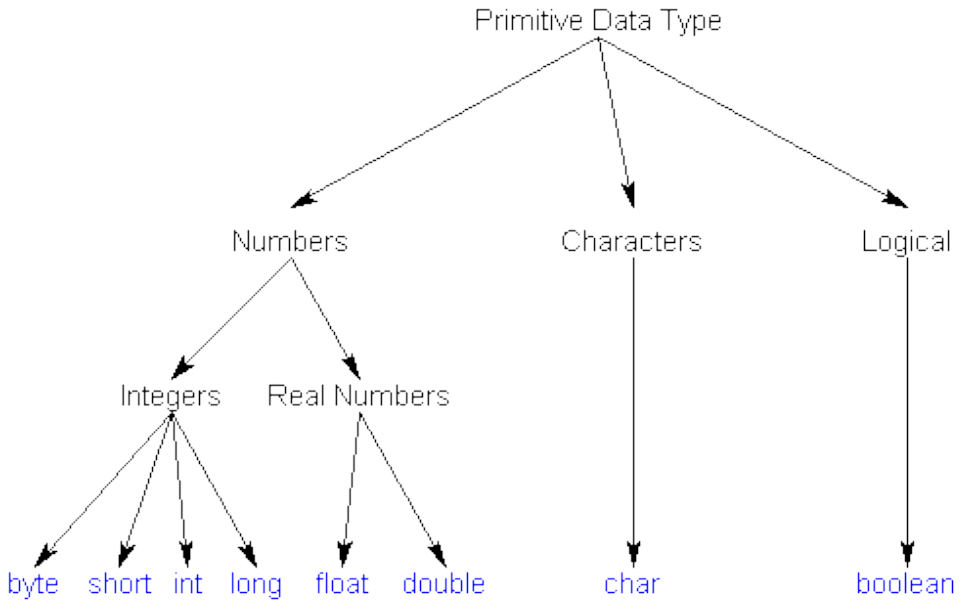
Topic 5.1.11

Describe how **linked lists** operate logically



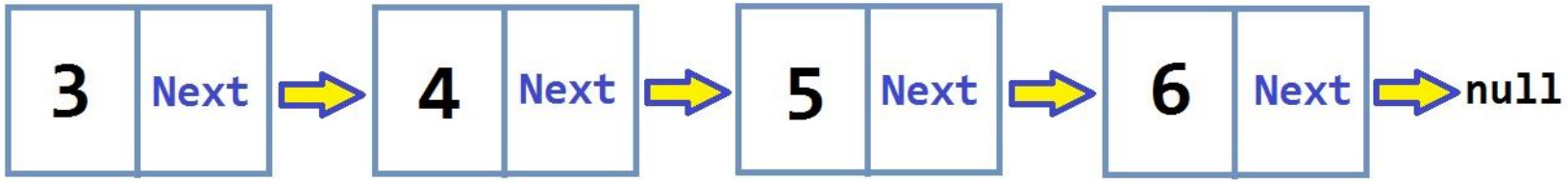
Abstract Data Structures (ADTs)

- 2D array
- Stack
- Queue
- **Linked List**
- (Binary) Tree
- Recursion

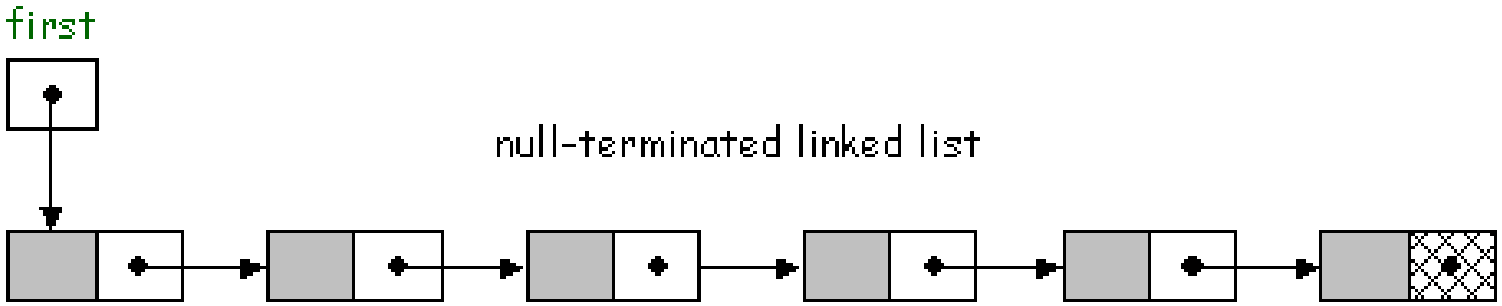


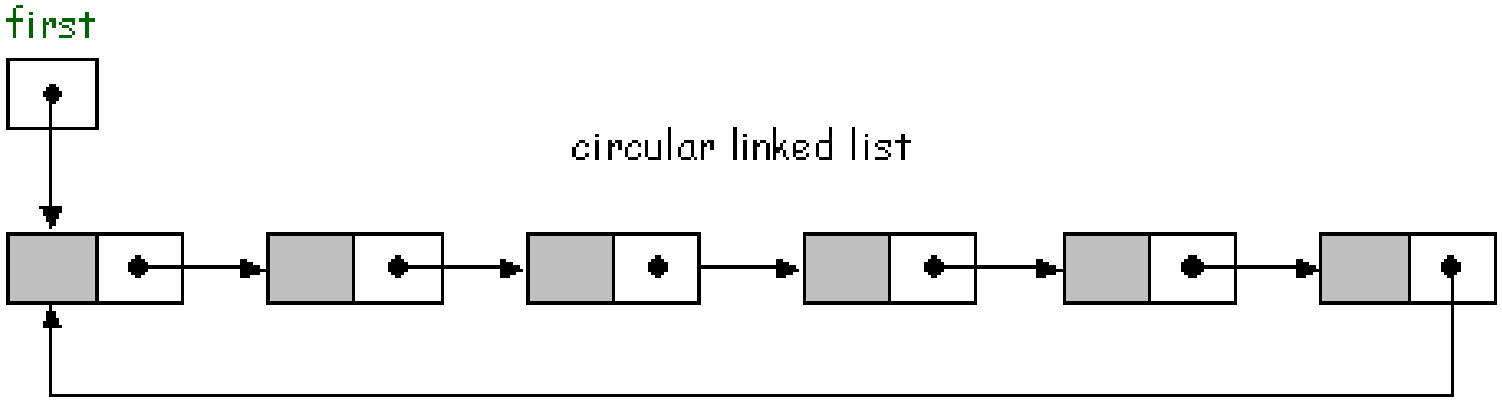
Linked list

- A **linked list** is a linear collection of self-referential structures, called **nodes**, connected by pointer **links**.
- A linked list is accessed by keeping a pointer to the first node of the list.
- This pointer to the first node of a list is typically named **head**.
- Subsequent nodes are accessed via a link pointer member that is stored in each node.



Linear & Circular Linked Lists

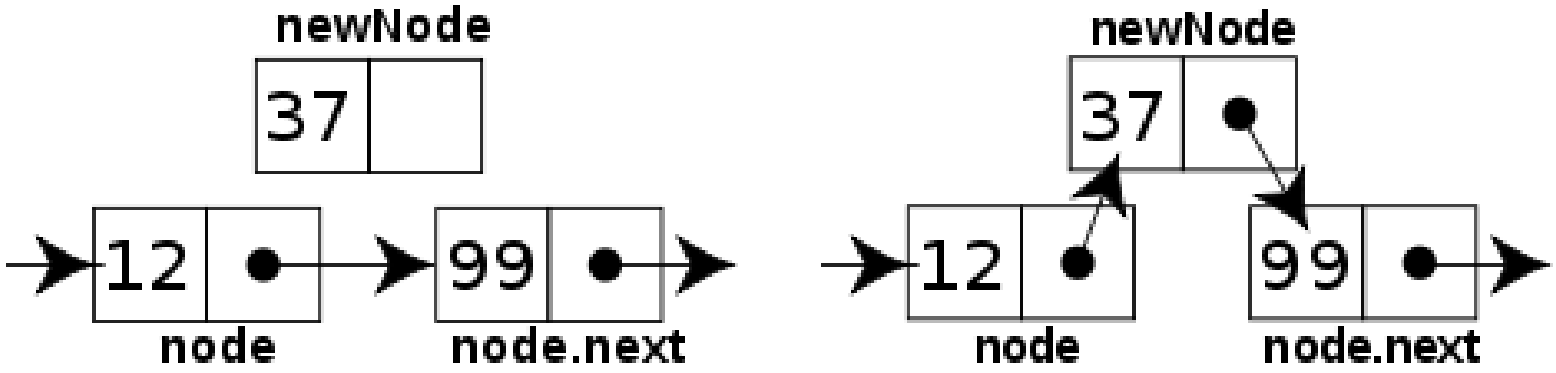




Inserting into a linked list

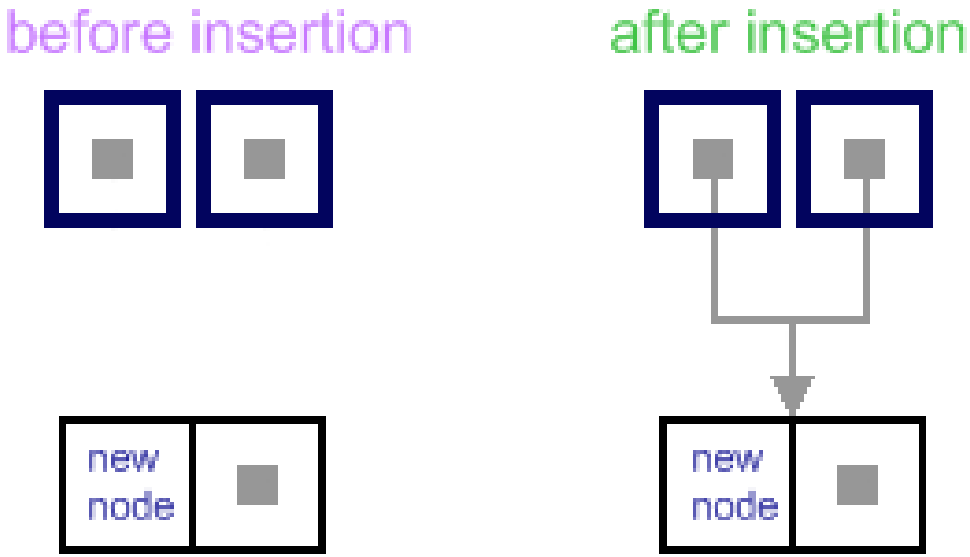
Insertion into a linked list has three special cases:

1. Inserting a new node to the **very beginning** of the list
2. Inserting a new node at the **very end** of the list
3. Inserting a new node in the **middle of the list** and so, has a predecessor and successor in the list



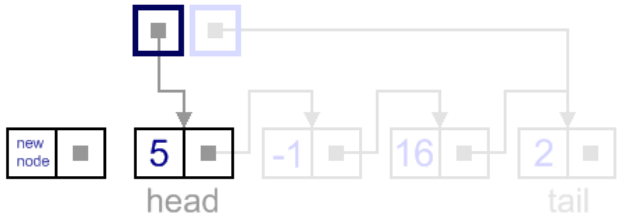
An empty list

- When list is empty, which is indicated by (**`head == NULL`**)condition, the insertion is quite simple.
- Algorithm sets both head and tail to point to the new node.

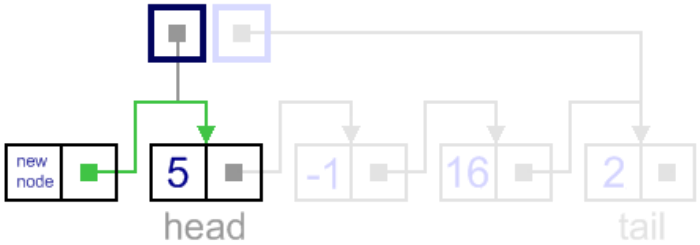


Inserting at the BEGINNING

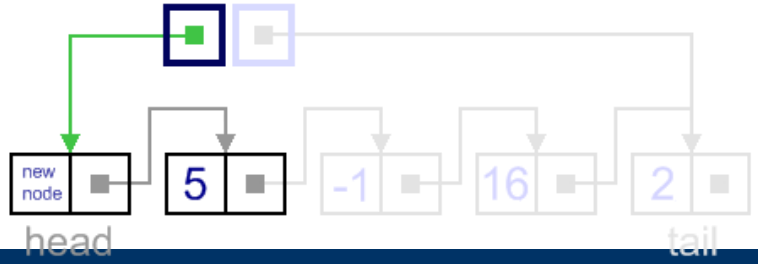
- In this case, new node is inserted right before the current head node.



- Update the next link of a new node, to point to the current head node.

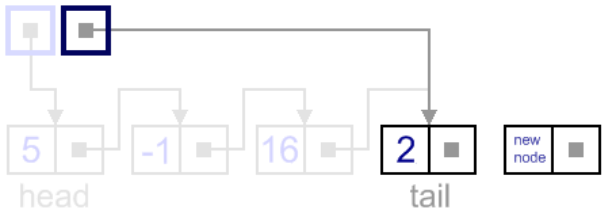


- Update head link to point to the new node.

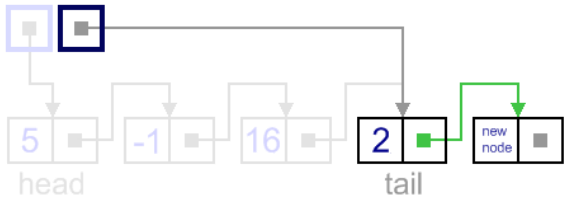


Inserting at the END

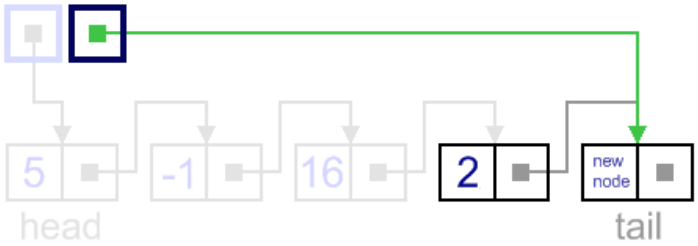
- In this case, new node is inserted right after the current tail node.



- Update the next link of the current tail node, to point to the new node.

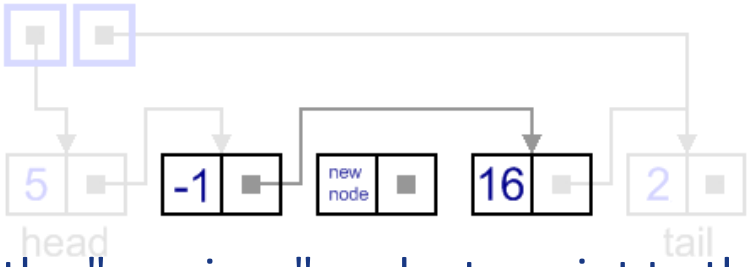


- Update tail link to point to the new node.

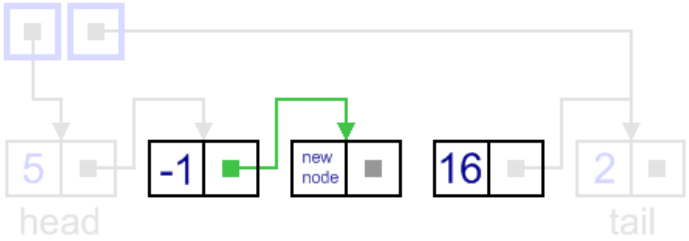


Inserting in the MIDDLE

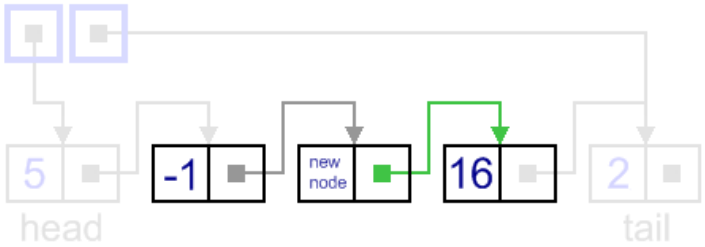
- In general case, new node is always inserted between two existing nodes. Head and tail links are not updated in this case.



- Update link of the "previous" node, to point to the new node.

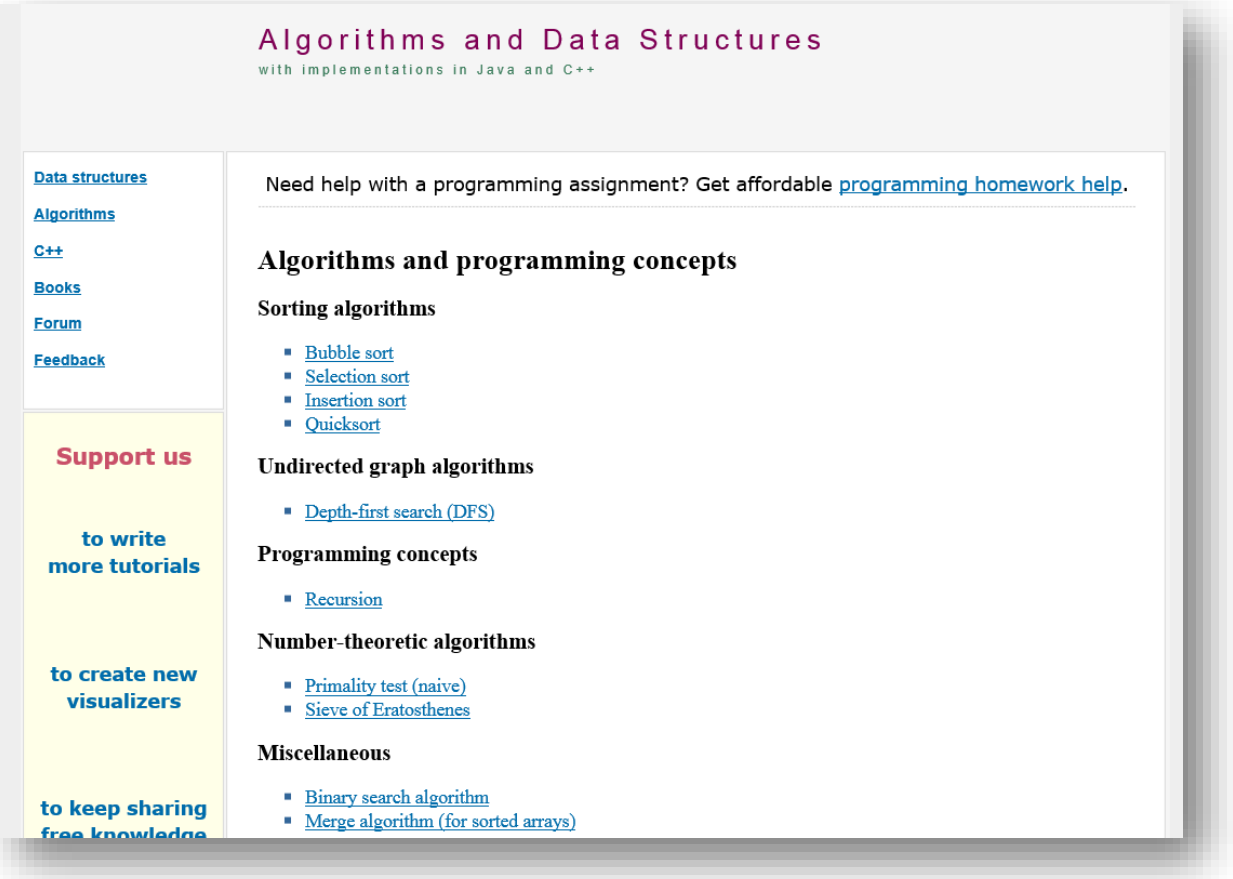


- Update link of the new node, to point to the "next" node.



Super useful site for algorithms

http://www.algolist.net/Data_structures/Singly-linked_list/Insertion



Exam question about ADTs

(Linked Lists, Queue, Stacks)

In a small airport, the details of all flights due to arrive on a particular day are held in a collection, `FLIGHTS`. Each object in the collection contains the following information:

`ID`: unique flight number
`PLACE`: where the plane is coming from
`DUE`: the time it is scheduled to arrive
`EXPECTED`: the time it is expected to arrive (only if it is early or if it is delayed)
`ARRIVED`: the time of actual arrival.

`EXPECTED` and `ARRIVED` are blank at the beginning of the day and the collection is sorted in order of `DUE`.

A screen in the airport can display information on 20 planes at a time, which are held in a linked list.

(a) Describe the features of a linked list of 20 planes that have the above information. [3]

All times are stored in the collection as the number of minutes since midnight. However they are displayed on the screen in 24-hour format (for example, 10:58 is stored in the collection as 658).

(b) Construct an algorithm to convert the times held in the collection into hours and minutes needed for the 24-hour format displayed on the screen. [3]

If a plane arrived more than 30 minutes ago it is removed from the linked list and the next one in the collection is added to the end of the list.

(c) With the aid of a diagram, explain how a plane which arrived more than 30 minutes ago could be removed from the linked list. [4]

(d) For the application described above, compare the use of a linked list with the use of a queue of objects. [5]

Solution for (a)

(a) *Award **[1 mark]** for data, **[1 mark]** for pointers, **[1 mark]** for order.*
Example:
Each node would hold the data for one plane (ID, place, time due, time expected, landed);
Head pointer points to the first in the list;
Each subsequent pointer points to the next in the list and last node has null pointer; **[3]**

Solution for (b)

(b) *Award **[1 mark]** for calculating hours.*
*Award **[1 mark]** for calculating minutes.*
*Award **[1 mark]** for input and output/return.*

Example 1:

```
input CTIME // time held in the collection in minutes
     HOURS = CTIME div 60
     MINUTES = CTIME mod 60
output HOURS, MINUTES // time to be displayed on the screen
```

Example 2:

```
input CTIME // time held in the collection in minutes
HOURS = 0
MINUTES = CTIME
WHILE MINUTES>59
     MINUTES=MINUTES-60
     HOURS=HOURS+1
ENDWHILE
output HOURS, MINUTES // time to be displayed on the screen
```

Example 3:

```
Format24 (CTIME)
// method accepts time held in the collection in minutes
     HOURS = CTIME div 60
     MINUTES = CTIME mod 60
     return  HOURS + ":" + MINUTES
     // returns time to be displayed on the screen
end Format24
```

[3]

Solution for (c) part 1

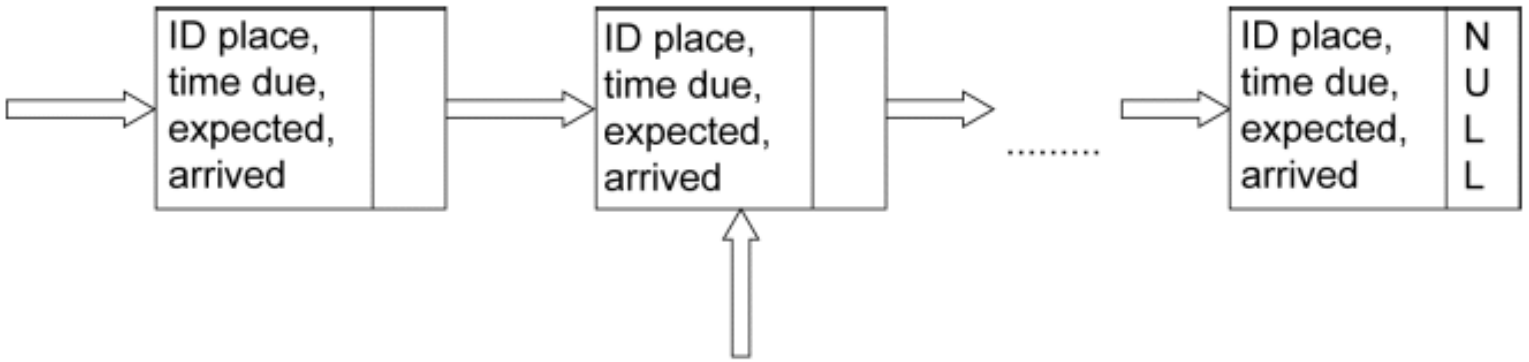
(c) *Award marks as follows, up to **[4 marks max]**.*
*Award **[1 mark]** for a diagram and explanation showing access to each plane via pointers;*
*Award **[1 mark]** for comparison of current time with time arrived;*
*Award **[1 mark]** for correct change of pointer from plane deleted;*
*Award **[1 mark]** for correct change of pointer to next plane;*

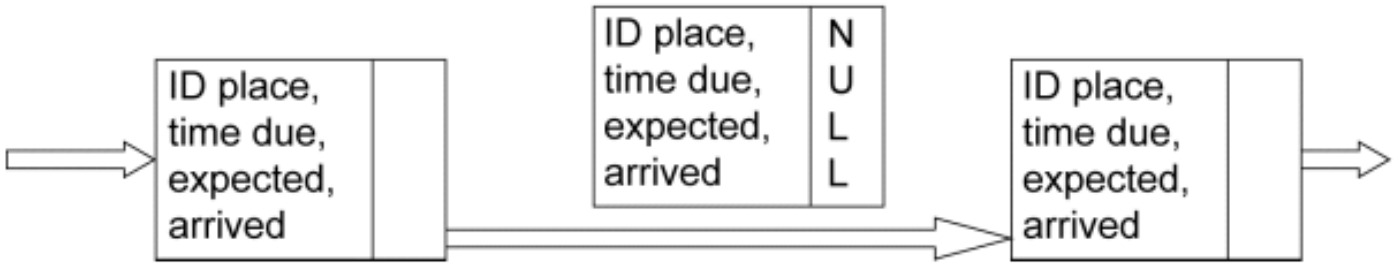
***Note:** The plane to be deleted could be at the beginning of the list **OR** at the end of the list **OR** in the middle of the list; award third and fourth mark (change of pointers) depending on the position of the node shown in the candidates' diagram/explanation.*

For example:

```
PLANES accessed sequentially via pointers;
PLANE.ARRIVED checked against current time;
    if > 30 minutes;
        if pointer is head pointer;
            move head pointer to point to next PLANE;
        else if plane is last in list previous pointer points to NULL;
        else previous pointer changed to subsequent plane;
        pointer of deleted plane null;
```

Solution for (c) part 2





[4]

Solution for (d)

(d) *Award up to **[5 marks max]**.*

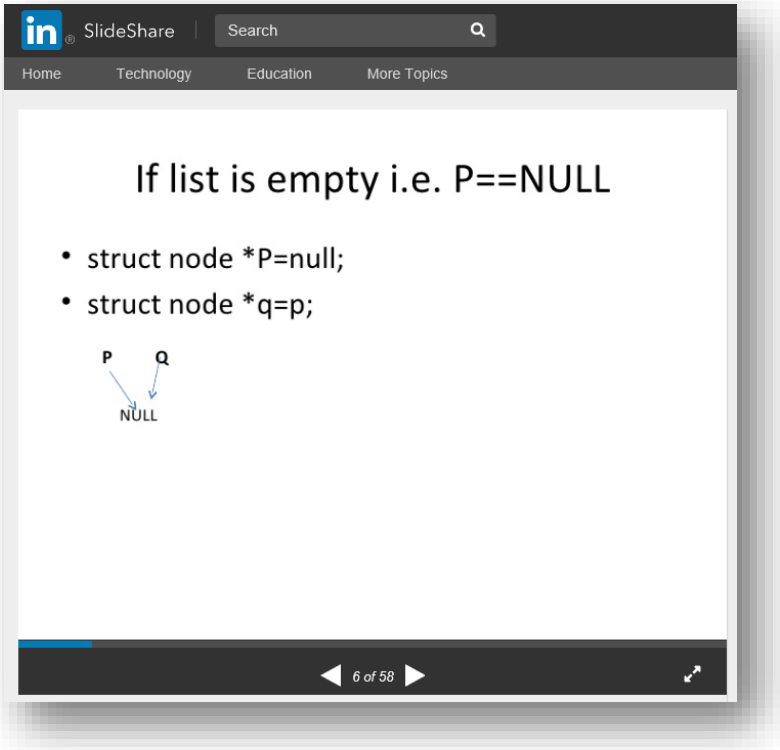
A queue would hold the elements in order of arrival;
And enqueue correctly to the end as required;

Dequeue would take planes from the top of the screen;
Which is not wanted as they arrive at different times;

Elements in a linked list could be removed from any position in the list;
Hence a linked list is better;

Searching for ID to amend will be equivalent; **[5]**

Great presentation about linked lists



http://www.slideshare.net/sshinchan/single-linked-list