

# Teach A level Computing: Algorithms and Data Structures

Eliot Williams

@MrEliotWilliams

# Course Outline

1	Representations of data structures: Arrays, tuples, Stacks, Queues, Lists
2	Recursive Algorithms
3	Searching and Sorting - EW will be late!
4	Hashing and Dictionaries, Graphs and Trees
5	Depth and breadth first searching ; tree traversals

# Data Structures

# Data structures are complex data types

- Allow you to store multiple data items of one or more data types....

a mechanism for grouping and organizing data to make it easier to use.

Data structures can be made from multiple data types and other data structures...

# Abstract vs Built in

- Abstract Data type
  - A model of how a data is stored and what operations can be carried out on the data
- Built in Type
  - A predefined implementation of a data type in a programming language

# Simple/Primitive data types

- Boolean
  - Int
  - Float
  - Char (python doesn't really have chars)
- 
- Define how a fixed length binary code should be interpreted by a program

# Lists

- Lists are a complex and powerful data structure.
- Very easy to use in python
- What are the properties of a list?

# Lists

- Lists are a complex and powerful data structure.
- Very easy to use in python
- `mylist=["sausages", "tin of beans", "eggs", 10]`
- `mylist[0] = "6 sausages"`
- `mylist.append("mushrooms")`
- `print(mylist)`
- `mylist.remove("tin of beans")`
- `print(mylist)`



# Lists

- Lists are a complex and powerful data structure.
- Very easy to use in python
- List are mutable
- Lists have variable length
- Lists are ordered.
- Constituent parts can be of different sizes and data types.
- Hard to understand how they work though so we will look at some simpler data structures first....

# Strings

- Built in data structure made up of?
- Properties of a string
  - Mutable?
  - Ordered?
  - Fixed length?

# Strings

- Built in data structure made up of?
- Properties of a string
  - Immutable
  - Ordered
  - Fixed length
  - Component parts all the same size...
- So not a list....

# Arrays

- “Simple” data structure – built in many languages(not python)
- One variable that can contain multiple items of the same type that are held in consecutive memory locations

Grade	0	1	2	3	4	5	6	7	8
	20	80	50	60	70	55	92	60	85

- We can access parts of an array by indexing e.g. `Grade[0]`
- Accessing any element takes the same time..
- Arrays have fixed length
- Arrays are mutable – a component value can be changed
- e.g. `Grade[0]=90`

# Arrays vs list

Arrays	Lists
Fixed number of entries	Can be extended
All entries the same size	Can have different entries
Continuous in memory	... more complex
mutable	mutable

Arrays are a simple data structure which are

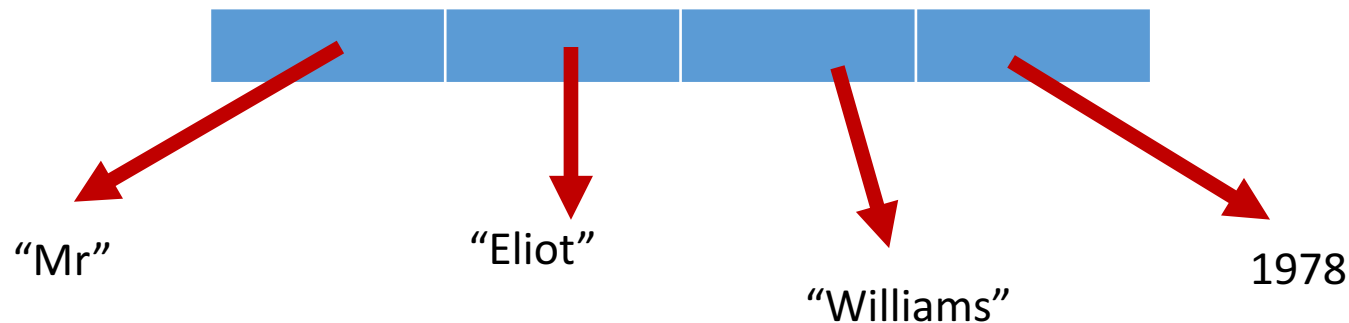
- Easy to understand how it works
- Very efficient. Accessing any element takes the same time....
- Allow us to create more complex data structures from them

# Arrays vs Strings

Arrays	Strings
Fixed number of entries	Fixed number of entries
All entries the same size	All entries the same size
mutable	immutable

# tuples

- A *tuple* is a finite ordered list of elements.
- Tuples are sometimes referred to as records
- Elements in a tuple can be of different data types.
- `teacher = ("Mr","Eliot","Williams",1978)`



- Tuples are ordered, fixed length & immutable

Print teacher

Will return the contents separated by commas

Print `teacher[2]` will return "Williams"

# Tuple uses

- Collecting together data about one “thing” – Record
- Returning multiple values from a procedure
- Write a function takes an input of a radius that returns a tuple of both the area and the circumference of the circle:



# unpacking tuples (python only)

- `teacher1 = ("Mr","Eliot","Williams",1978)`
- `(title,first,surname,year)=teacher1`
- `print (surname)`
- This also allows us to swap the contents of variables easily
- `(a,b)=(b,a)`

# Multidimensional arrays (NB using lists)

- Names =["Bob","Sally","James"]
- Mark1=[20,23,19]
- Marks2=[21,22,20]
- Marks3=[19,23,20]

Names	Bob	Sally	James
Mark1	20	23	19
Mark2	21	22	20
Mark3	19	23	20

- Could be created as
- Names =["Bob","Sally","James"]
- grades=[[20,23,19],[21,22,20],[19,23,20]]

Names	Bob	Sally	James
Mark1	20	23	19
Mark2	21	22	20
Mark3	19	23	20

- `grades=[[20,23,19],[21,22,20],[19,23,20]]`
- How could we access Bob's score for test1?
- How could we work out Bob's total score?
- How could we work out the average for test3?

Names	Bob	Sally	James
Mark1	20	23	19
Mark2	21	22	20
Mark3	19	23	20

- `grades=[[20,23,19],[21,22,20],[19,23,20]]`
- How could we access Bob's score for test2?

Names	Bob	Sally	James
Mark1	20	23	19
Mark2	21	22	20
Mark3	19	23	20

- `grades=[[20,23,19],[21,22,20],[19,23,20]]`
- How could we access Bob's score for test2?
- `grades=[a,b,c]`
- `grades[1] >>>>b>>>>[21,22,20] b[0]>>>>21`
- `grades[1][0]`

Names	Bob	Sally	James
Mark1	20	23	19
Mark2	21	22	20
Mark3	19	23	20

- `grades=[[20,23,19],[21,22,20],[19,23,20]]`
- How could we work out Bob's total score?
- `Score =grades[0][0]+grades[0][1]+grades[0][2]`  
but this might take a long time to type if there were lots of tests

Score=0

For i in range(0,3):

`score=score + grades[0][i]`

Names	Bob	Sally	James
Mark1	20	23	19
Mark2	21	22	20
Mark3	19	23	20

- `grades=[[20,23,19],[21,22,20],[19,23,20]]`
- How could we work out the average for test3?
- `Average=(grades[2][0]+grades[2][1]+grades[2][2])/3`

# Multi-Dimensional Array Tasks

- Sum Columns of Table (page 1)



# We can have n dimensional arrays

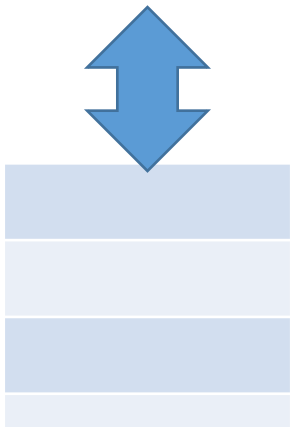
- Marks[test][question][person]
- [[[3,5,7],[...

# Stacks & Queues

- Abstract data types - ordered collections of data
- Queue – Last In First Out



- Stack – First In, Last Out



# Stacks



2 stack operations

Push – puts data at the top of the stack



Pop – removes data from the top of the stack and returns it



# Stack Exercises

- Theory p2
- Practical P3

# Python list commands....

- `list.append()`
- `list.pop()`

# Queue

- Data enters a queue from the back
- Data leaves a queue from the front



# Implementing queues




- Very similar to a stack
- We could use a fixed length array



## 2 operations

- Enqueue
  - Add an item into the queue
- Dequeue
  - Remove the first item in the queue
  - move all items up?

# Queue Exercise (page 4)

- Write two procedures
- enqueue
- #put the item at the back of the queue
- # change back of queue value
- dequeue
- #return first value
- #update rest of queue..
- 
- all errors should be appropriately handled
- 
- Queue and backofqueue may be used as global variables in this assignment
- 
- easy, medium and hard code skeletons are available



# Priority Queue

- Some things are more important than others!
- Items should be dequeued in order of priority then queue position
- Several ways of doing this:

# Priority Queue

- Some things are more important than others!
- Items should be dequeued in order of priority then queue position
- Several ways of doing this:
  - Change enqueue to put things in the right place!
    - Lots of data shuffling, inefficient in an array based structure
  - Change dequeue to find the high priority items first

# Priority Queue- Dequeue

```
queue=[("bob",1),("James",1),("David",2),("John",1),"","","","","",""]  
backofqueue=2
```

dequeue():

- loop through the queue to find the highest priority and its position
- store the value at that position
- move all the items from that point up down one
- update and clear back of queue
- return stored answer

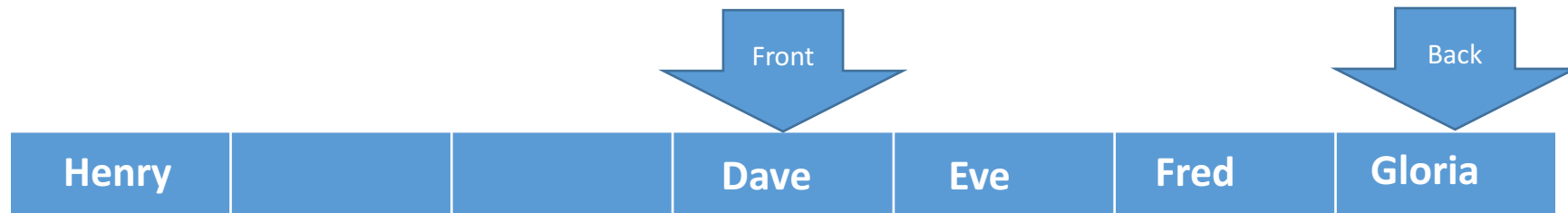
- All this shuffling data about is wasteful
- We could just move a pointer



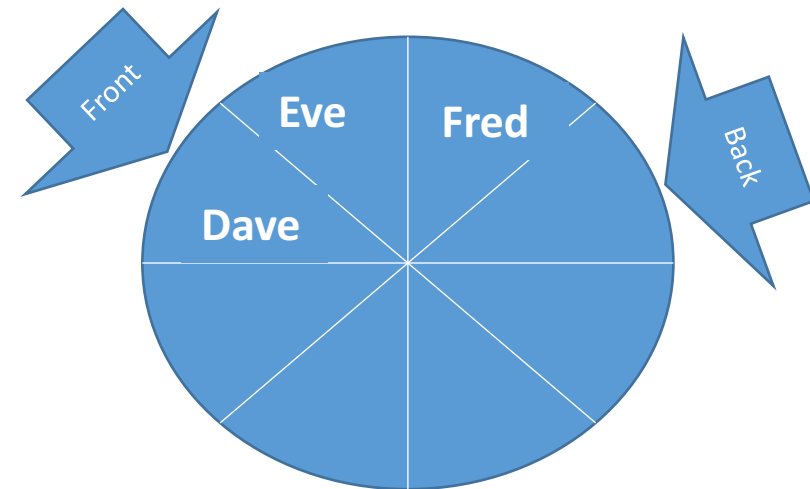
# Priority Queue Exercise (p5)

- Write the dequeue procedure
- dequeue
- #find the highest priority item in the queue and return it,
- #update queue
- all errors should be appropriately handled
- Queue and backofqueue may be used as global variables in this assignment
- easy, medium and hard code skeletons are available

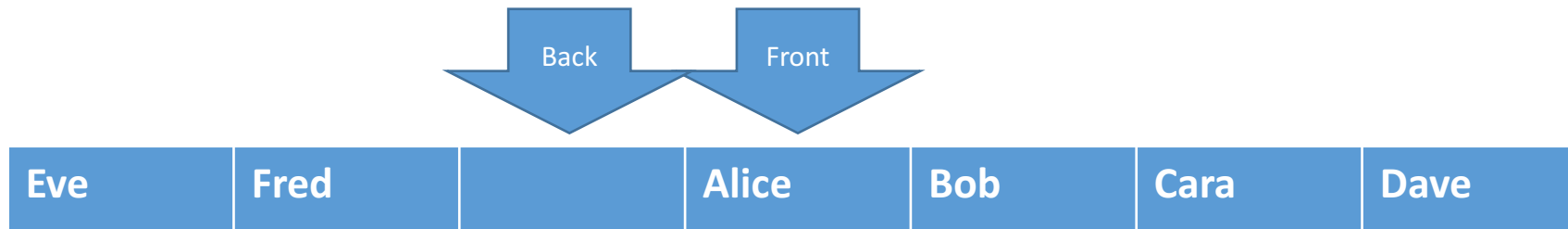
# Circular Queue



- Pointer movement for enqueue and dequeue needs to wrap round
- How do we tell if a queue is full?
- How do we tell if a queue is empty?



- If  $\text{head} = \text{tail}$  then the queue is defined to be empty
- if  $\text{head} = \text{tail} + 1$ , or  $\text{tail} = \text{max}$  and  $\text{head} = 0$ , it is defined to be full.



# Circular Queue Exercise (page 6)



# Circular Priority Queue!

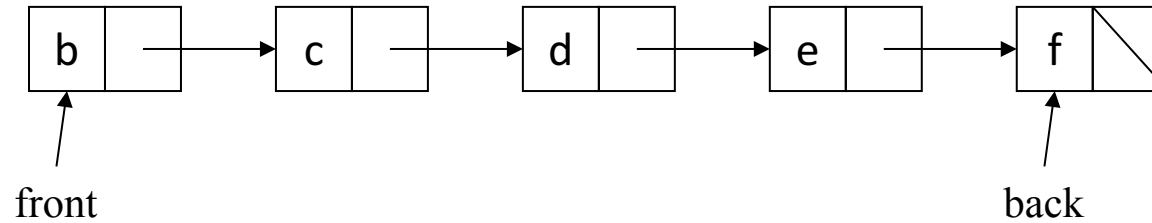
# Lists

- Variable size
- Mutable
- ordered

# Linked Lists

An implementation of the list abstractions

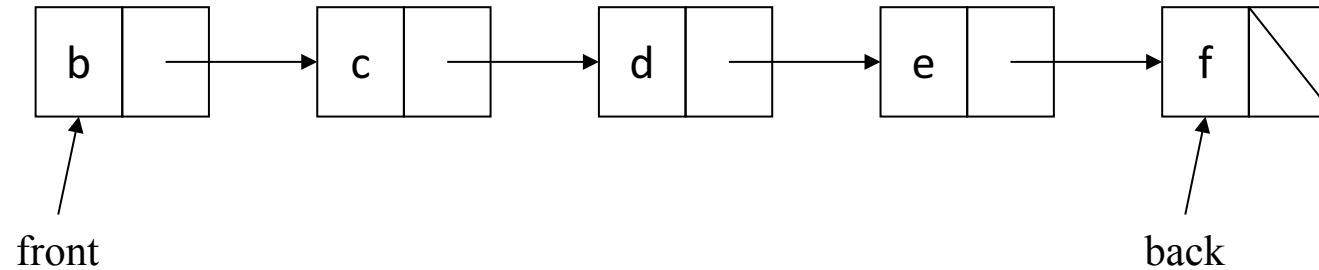
# Linked List Concept



- Each entry has
  - A value
  - A pointer to the next entry
- Keep a pointer to the front entry
- The pointer of the last entry is None

- We could represent this in python by
- `list=[['James',2],['Bob',0],['Sarah',-1]]`
- `startpos=1`
- `length=3`
  
- We would need procedures to
- find a values position
- Find an pointers position
- Add
- Delete (and update the previous pointer – using find a pointer)
- Print in order
- Insert a new value

# Exercise



- Redraw list after:
  - appending a new entry at the end
  - inserting before entry zero
  - inserting before entry 3

# Linked List exercises

# Linked List Index

```
myList.index(i)
```

- Count along the list to entry i
- Return the value

```
pointer = front  
count = 0  
while count < i:  
    pointer ← next of current entry  
    count = count + 1  
return the value of current entry
```



# Linked List Update

```
myList.update(idx, newValue)
```

- Count along the list to entry index
- Replace value with new value

```
pointer = front  
count = 0  
while count < idx:  
    pointer ← next of currentEntry  
    count = count + 1  
currentEntry.value = newValue
```

# Linked List Insert

```
myList.insert(idx, newValue)
```

- Count along the list to entry idx-1
- Insert a new entry
  - Next pointer of current entry points to new entry
  - Next pointer of new entry points to following entry

# Linked List Code

```
class Entry:
    def __init__(self, v):
        self.value = v
        self.next = None

    def setValue(self, v):
        self.value = v
    def getValue(self):
        return self.value

    def setNext(self, n):
        self.next = n
    def getNext(self):
        return self.next
```

```
class List:
    def __init__(self):
        self.length = 0
        self.first = None

    def append(self, value):
        entry = Entry(value)
        if self.first == None :
            self.first = entry
            return
        p = self.first
        q = p.getNext()
        while q != None:
            p = q
            q = p.getNext()
        p.setNext(entry)

    def index(self, i):
        count = 0
        p = self.first
        while count < i:
            p = p.getNext()
            count = count + 1
        return p.getValue()
```

# Complexity of Linked List

- Indexing is linear:  $O(n)$ 
  - c.f. array index is  $O(1)$
  - need to do better!
- Often need to visit every entry in the list
  - e.g. sum, **search**
  - This is  $O(n^2)$  if we use indexing
  - Easy to improve this by keeping track of place in list
- Search is  $O(n)$

# Some Additional resources

- Linked lists
- <http://openbookproject.net/thinkcs/python/english2e/ch18.html>