Accredited

AS and A LEVEL

Topic Exploration Pack

H046/H446

COMPUTER SCIENCE

Theme: Data Structures

September 2015

OCR
Oxford Cambridge and RSA

We will inform centres about any changes to the specification. We will also publish changes on our website. The latest version of our specification will always be the one on our website (www.ocr.org.uk) and this may differ from printed versions.

Oxford Cambridge and RSA Examinations is a Company Limited by Guarantee.
Registered in England. Registered company number 3484466.

Registered office: 1 Hills Road
Cambridge
CB1 2EU

OCR is an exempt charity.

Contents

This Topic Exploration Pack should accompany the OCR resource ‘Data Structures’ learner activities, which you can download from the OCR website.

Arrays

1.4.2 a) Arrays (of up to 3 dimensions), records, lists, tuples

Arrays, records, lists and tuples are the main data structures that learners will encounter when coding solutions to problems no matter what programming language they use. They should have prior knowledge of data types before moving onto structures to store these. A rundown of basic data types can be found at:

http://www.tutorialspoint.com/python/python_variable_types.htm

Some experience using the above data structures in a meaningful context would benefit learners, such as asking the user for a name and what score they got in a computer game, then at the end the program iterates through the dictionary (for example) and outputs a whole list of their names and scores. A worksheet for this is provided.

Activity 1 High Scores List using Dictionaries

Learner Task: produce a program that holds a table of high scores for a computer game.

Task 1

Create a dictionary of names and high scores. A breakdown of what dictionaries are and their functions can be found at http://www.tutorialspoint.com/python/python_dictionary.htm.

Task 2

Enable the user to look up someone's high score from the dictionary.

Task 3

Enable the user to update the high scores table for a person.

Task 4

Enable the user to print out all the high scores in order (descending).

Full code for this activity:

```
from operator import itemgetter
high_scores = {"mike":100,"joe":300,"emily":200}


def main():
    choice = None
    while choice != "0":
        print(
        """
        Hi scores system
        0-Quit
        1-Look up someone's high score
        2-Add a score
        3-Update a score
        4-Display all high-scores
        """
        )
        choice = input("Choice: ")
        #exit
        if choice == "0":
            print("Goodbye")
        #look up a score
        elif choice == "1":
            player = input("Whose score would you like to look at?")
            if player in high_scores:
                score = high_scores[player]
```

```
                print("\n", player, "'s high score is ", score)
            else:
                print("Sorry, player ", player, "not in system")
        elif choice == "2":
            player = input("Whose score would you like to add?")
            if player not in high_scores:
                score = input("What is their high-score")
                high_scores[player] = score
                print("\n", player, "'s score has been added to the system")
            else:
                print("That player already exists. If you want to edit their
score do so from the menu")
        #edit a high score
        elif choice == "3":
            player = input("Whose score would you like to edit?")
            if player in high_scores:
                score = input("What would you like their new score to be?")
                high_scores[player] = score
                print("\n", player, "'s high score is now", score)
            else:
                print("Sorry, player ", player, "not in system")
        #display all high-scores
        elif choice == "4":
            for key, value in sorted(high_scores.items(), key=itemgetter(1),
reverse = True):
                print(key, value)
```

Storing Data

1.4.2 b) The following structures to store data: linked-list, graph (directed and undirected), stack, queue, tree, binary search tree, hash table.

Stacks are best described as a stack of books. We can either add books to the stack (push) or remove them from the top of the stack (pop). To access the item at the bottom of the stack we have to pop the items off until we reach it. This data structure is known as FILO (First in last out).

Queues are best described as a queue for the bank. It is a FIFO (first in first out) structure. There are methods to append to the end of the queue and a delete method to remove items.

Implementations for Stacks and Queues are already built into the Python programming language as part of the ‘list’ documentation.

Linked lists are data structures that are advantageous over using arrays since they can grow or shrink in size. They are made up of single instances of a node class that hold pointers to the next node. Therefore if a linked list has no items in, it won’t take up a huge amount of room in memory, but an empty array will. Linked lists are not so often used in Python, as use of the ‘list’ class is more convenient a lot of the time. The list class in Python has an ‘append’ method, which usual ‘arrays’ found in other programming languages do not have and are fixed in size.

Activity 2 Linked lists in Python 3

Learners can implement the linked-list implementation in python using the worksheet. Diagrams are used to illustrate what is going on.

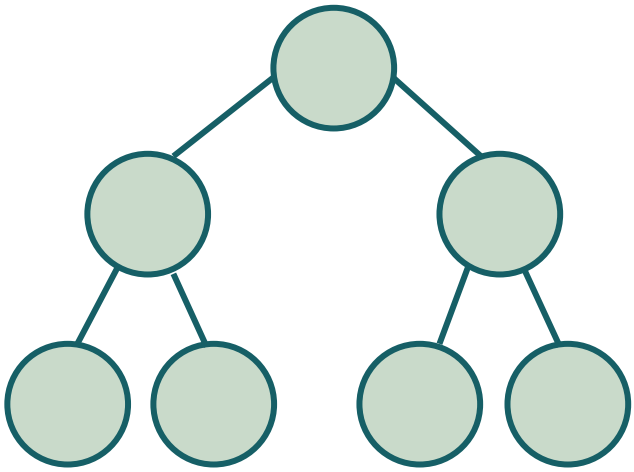
Once learners have worked through the activities, this is a fully working implementation of a linked-list. However, a list of extension activities given in the worksheet allow learners to increase the functionality of the implementation.

Full code for this activity:

```
def main():
    Node1 = Node("Apple") #head
    Node2 = Node("Orange")
    Node3 = Node("Pear")
    Node1.next = Node2
    Node2.next = Node3
    choice = None
    while choice != "0":
        choice = input("""0-Exit   1-Print the linked list   2-Insert a value at
a certain point   3-Delete a linked-list value""")
        if choice == "1":
            printList(Node1)
        elif choice == "2":
            dataToInsert = input("What data would you like to insert?")
            valueToInsertAfter = input("Where would you like that data to go
after?")
            InsertData(Node1,dataToInsert,valueToInsertAfter)
        elif choice == "3":
            dataToDelete = input("What data would you like to delete?")
            Node1 = DeleteData(Node1,dataToDelete)

def DeleteData(head,data):
    current = head
    previous = None
    found = False
    while not found:
        if current.data == data:
            found = True
        else:
            previous = current
            current = current.next
    if previous == None:
        head = current.next
    else:
        previous.next = current.next
    return head

def InsertData(head,data,location):
    node = head   # give the head node
    while node != None:
```

```
        if node.data == location:
            NewNode = Node(data)
            NewNode.next = node.next
            node.next = NewNode
        node = node.next

def printList(head):
    node = head   # give the head node
    print() #new line
    while node != None:
        print(node.data)
        node = node.next

class Node:
    def __init__(self, data):
        self.data = data  # instance variable to store the data (a name)
        self.next = None  # instance variable with address of next node
```

Binary trees are like an upside down tree where the root node is at the top. Each root node may have two child nodes, named the left child and the right child.



Binary trees are most commonly used in Binary search trees, where elements deemed smaller than the root node are given on the left child node and elements deemed larger are given on the right child node. This gives order to the data structure and therefore can provide quicker search times than other data structures.

Graphs are networks consisting of nodes that may be linked to one or more other nodes. Graphs may be directed or undirected, meaning that in a directed graph you may only be able to traverse to another node one way and not the other. Planning flight paths and train routes are good examples to give for uses for graphs. A problem that illustrates graphs well is the travelling salesman or this activity about trying to connect cities using as few paths as possible:

http://csunplugged.org/minimal-spanning-trees

Hash tables are data structures that process the data you want to add into the table by firstly performing a hash function to generate a value. This value is a ‘key’ which can later be used to retrieve the data. A simple example of this would be if a hash function generated a key by adding the two digits of a number together eg 16 would produce 7 as a key, and so would 25 and so 16 and 25 would be associated to this value. Hash tables are often used for database indexing as they can be much more efficient than other table lookup structures.

Learners may not necessarily use all of these data structures in their projects but they are still expected to understand how they work and why they are used. Learners should have covered the basics of arrays, records and lists before moving onto this section.

Activity 3 Binary Tree in Python 3

Learners can implement the binary tree implementation in python using the worksheet. Diagrams are used to illustrate what is going on.

The code in the worksheet is partially complete. For instance, the left hand tree is very much the same code as the right hand side, which will encourage learners to look at what the code is actually doing rather than blindly copying.

The worksheet finishes off with proof of how we know the code is working using a trick for traversing binary trees.

Full code for this activity:

```
class BinaryTree:
    def __init__(self,rootID):
        self.rootID = rootID
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            self.leftChild = t
            t.leftChild = self.leftChild

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t

#pre order
def printTreePre(tree):
        if tree != None:
            print(tree.rootID)
            printTreePre(tree.leftChild)
            printTreePre(tree.rightChild)

#in order
def printTreeIn(tree):
        if tree != None:
            printTreeIn(tree.leftChild)
```

```
            print(tree.rootID)
            printTreeIn(tree.rightChild)

#post order
def printTreePost(tree):
        if tree != None:
            printTreePost(tree.leftChild)
            printTreePost(tree.rightChild)
            print(tree.rootID)

r = BinaryTree('a')
r.insertLeft('b')
r.insertRight('c')
r.insertRight('d')
printTreePre(r)
print()
printTreeIn(r)
print()
printTreePost(r)
```

Creating, Traversing, Adding and Removing Data from Data Structures

1.4.2 c) How to create, traverse, add data to and remove data from the data structures mentioned above. (NB this can be either using arrays and procedural programming or an object-oriented approach).

Most learners will find the initial data structures easy to construct. In fact, in some cases these are built directly into the language such as lists, tuples, dictionaries, stacks and queues that are built into python. See this link for more information: https://docs.python.org/3.1/tutorial/datastructures.html

Learners may find linked-lists and binary trees difficult, but a breakdown of these are given in Activities 2 and 3. A simple implementation example of a graph can be found at https://www.python.org/doc/essays/graphs/. Hash tables can also be constructed in the same way, although the only difference would be that the key has been generated from a hash function of some sort.

Learners will usually be able to understand the point of having a specific data structure, but may have trouble understanding the relationship between nodes in a data structure using Object Oriented techniques like Linked-Lists or Binary Trees (see Activities 2 and 3).

OCR customer contact centre
General qualifications
Telephone 01223 553998
Facsimile 01223 552627
Email general.qualifications@ocr.org.uk