OCR Computer Science A Level

2.3.1 Algorithms for the Main Data Structures

Concise Notes

Specification:

- Stacks
- Queues
- Linked lists
- Trees
- Traversal of trees
    - Depth-first (post-order)
    - Breadth-first

Algorithms for the Main Data Structures

- Each data structure has its own algorithms associated with it
- These allow data to be manipulated in useful ways
- All data structures mentioned are covered in greater detail in 1.4.2 Data Structures

Stacks

- Example of a first in, last out (FILO) data structure
- Often implemented as an array
- Use a single pointer which keeps track of the top of the stack (called the top pointer)
  - Points to the element which is currently at the top of the stack
  - Is initialised at -1, as the first element in the stack is in position 0
- Algorithms for stacks include adding to the stack, removing from the stack and checking whether the stack is empty/full
- All of the operations have their own special names, as shown in the table below

Operation	Name
Check size	`size()`
Check if empty	`isEmpty()`
Return top element (but don't remove)	`peek()`
Add to the stack	`push(element)`
Remove top element from the stack and return removed element	`pop()`

size()

- Returns the number of elements on the stack
- Returns the value of the top pointer plus one

```
size()
    return top + 1
```

isEmpty()

- Returns True if the stack is empty, otherwise returns False
- Works by checking whether the top pointer is less than 0

```
isEmpty()
     if top < 0:
          return True
     else:
          return False
     endif
```

peek()

- Returns the item at the top of the stack, without removing it
- Returns the item at the position indicated by the top pointer
- Important to check that the stack has data in it before attempting to return anything

```
peek()
     if isEmpty():
          return error
     else:
          return A[top]
     endif
```

push(element)

- Adds an item to a stack
- The new item must be passed as a parameter
- Firstly, the top pointer is updated accordingly
- Then the new element can be inserted at the position of the top pointer

```
push(element)
     top += 1
     A[top] = element
```

pop()

- Removes an item from a stack
- Element at the position of the top pointer is recorded before being removed
- Top pointer decremented by one
- The removed item is returned
- As with `peek()`, it's important to first check that the stack isn't empty

```
pop()
     if isEmpty():
          return error
     else:
          toRemove = A[top]
          A[top] = “”
          top -= 1
          return toRemove
     endif
```

Queues

- A type of first in, first out (FIFO) data structure
- Just like stacks, queues are often represented as arrays
- Unlike stacks, queues make use of two pointers:
  - Front holds the position of the first element
  - Back stores the next available space
- Operations which can be carried out on queues are similar to those of stacks

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
John	Sarah	Mel	Lucy	Stephen			
↑ Front					↑ Back		

Operation	Name
Check size	`size()`
Check if empty	`isEmpty()`
Return top element (but don’t remove)	`peek()`
Add to the queue	`enqueue(element)`
Remove element at the front of the queue and return removed element	`dequeue()`

`size()`
- Returns the number of elements in a queue
- Simply subtracts the value of front from back

```
size()
     return back - front
```

`isEmpty()`
- Returns True if a queue is empty, and False otherwise
- When a queue is empty, front and back point to the same position

```
isEmpty()
     if front == back:
          return True
     else:
          return False
     endif
```

`peek()`
- Returns the element at the front of the queue without removing it

```
peek()
     return A[front]
```

`enqueue(element)`
- Adds an element to the back of a queue
- The new element is placed in the position of back
- Back is incremented by one

```
enqueue(element)
     A[back] = element
     back += 1
```

dequeue()

- Removes the item at the front of the queue
- Items are removed from a queue from the position of the front pointer
- Just as with stacks, it's important to check that the queue isn't empty
- After the element has been removed, the front pointer must be incremented

```
dequeue()
     if isEmpty():
          return error
     else:
          toDequeue = A[front]
          A[front] = “”
          front += 1
          return toDequeue
     endif
```

Linked Lists

- Composed of nodes, each of which has a pointer to the next item in the list
- If a node is referred to as N, the next node can be accessed using N.next
- The first item in a list is referred to as the head and the last as the tail
- Searching a list is performed using a linear search
  - Carried out by sequential next operations until the desired element is found

Trees

- Formed from nodes and edges
- Cannot contain cycles
- Edges are not directed
- Useful as a data structure because trees can be traversed
- There are two types of traversal to cover: depth first (post-order) and breadth first
- Both can be implemented recursively

Depth first (post-order) traversal

- Goes as far into the tree as possible before backtracking
- Uses a stack and goes to the left child node of the current node when it can
- If there is no left child then the algorithm goes to the right child
- If there are no child nodes, the algorithm visits the current node, outputting the value of this node
- It then backtracks to the next node on the stack and moves right
- See the example in the full notes

Breadth first

- Starting from the left, breadth-first visits all the children of the start node
- The algorithm then visits all nodes directly connected to each of those nodes in turn, continuing until every node has been visited
- Unlike depth first traversal (which uses a stack), breadth first uses a queue