MODULE 9

QUEUE

DATA STRUCTURE AND ALGORITHMS

FACULTY OF COMPUTING
UNIVERSITI TEKNOLOGI MALAYSIA

OBJECTIVES FOR STUDENTS

1. Understand the queue concept and the purpose of queuing operation.
2. Understand the implementation of basic queuing algorithm.
3. Able to implement queuing technique in problem solving using array and linked list.

KEY CONCEPT

1.0 INTRODUCTION TO QUEUE

1.1. **Introduction to queue**

- New items enter at the back, or rear, of the queue
- Items leave from the front of the queue
- First-in, first-out (FIFO) property
- The first item inserted into a queue is the first item to leave
- Middle elements are logically inaccessible
- Important in simulation & analyzing the behavior of complex systems

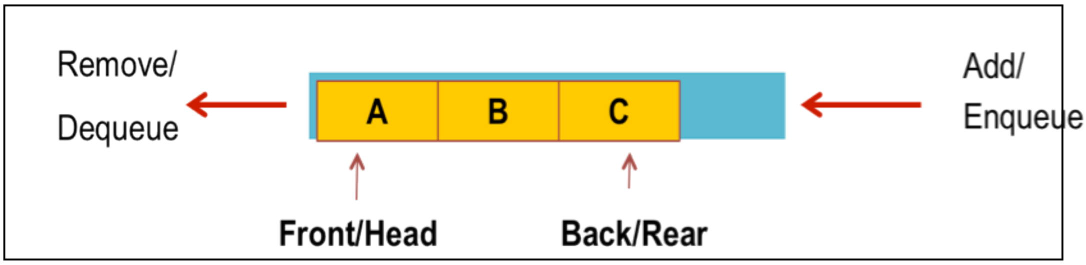
1.2. **Queue applications** :

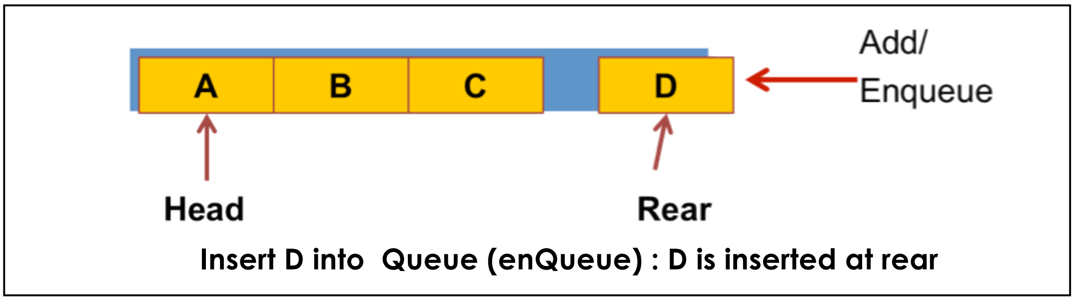
- Real-World Applications

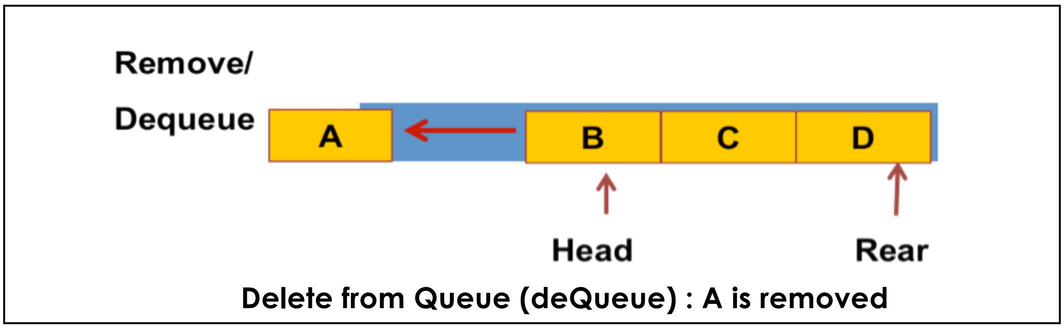
  - Cashier lines in any store
  - Check out at a bookstore
  - Bank / ATM
  - Call an airline
- Computer Science Applications
  - Print lines of a document
  - Printer sharing between computers
  - Recognizing palindromes
  - Shared resource usage (CPU, memory access, …)
- Simulation - A study to see how to reduce the wait involved in an application

1.3. **Queue implementation** :

- Basic Structure of a Queue:
    - Data structure that hold the queue
    - head
    - rear







1.4. **Abstract Data Type Queue**:

- ADT queue operations
    - Create an empty queue
    - Destroy a queue
    - Determine whether a queue is empty
    - Add a new item to the queue
    - Remove the item that was added earliest
    - Retrieve at Front
    - Retrieve at Back

- Implementation:
    - **Array-based (Linear or Circular)**
    - **Pointer-based : Link list (Linear or Circular)**

2.0 QUEUE : LINEAR ARRAY IMPLEMENTATION

2.1 Queue : Linear Array Implementation

Queue
front rear items

createQueue()
destroyQueue()
isEmpty();
isFull();
enQueue();
deQueue();
getFront();
getRear();

- Number of elements in Queue are fixed during declaration.
- Need **isFull()** operation to determine whether a queue is full or not.
- Queue structure need at least 3 elements:
  - Element to store items in **Queue**
  - Element to store index at **head**
  - Element to store index at **rear**

2.2 Queue declaration

```
// Program 9.1
class Queue
{ private:
     int front; // index at front
     int back; // index at rear queue
     char items[size]; //store item in Q
  public:
    Queue(); // Constructor - create Q
   ~Queue(); // Destructor - destroy Q
    bool isEmpty(); // check Q empty
    bool isFull(); // check Q full
    void enQueue(char); // insert into Q
    void deQueue(); // remove item from Q
    char getFront(); // get item at Front
    char getRear(); // get item at back Q
};
```

2.3 createQueue() operation

- Linear Array implementation
  - *front* & *back* are indexes in the array
  - Initial condition: *front =0* & *back = -1*

```
// Program 9.2
Queue::Queue()
{ front = 0;
  back = -1;
}
```

0	**-1**						
front	**back**	**0**	**1**	**2**	**3**	**....**	**maxQ-1**

Initial state for a queue linear array

2.4 Queue operations

- Destructor – To destroy Queue
  - All elements in the queue will be disposed.

```
// Program 9.3
queue::~queue()
{ delete [ ] items; }
```

- **isEmpty()-** Check whether a queue is empty
  - Queue is empty when: **back < front**

```
// Program 9.4
bool queue::isEmpty()
{ return bool(back < front); }
```

- **isFull()-** Check whether a queue is full

  - Queue is full when : **back = size -1**
  - No more item can be insert into a queue, when a queue is full

```
// Program 9.5
bool queue::isFull()
{ return bool(back == size – 1); }
```

- **enQueue()** - Insert into a queue
  - First, increment **back**
  - Then, insert item in **item [back]**

```
// Program 9.6
void queue::enQueue(char insertItem)
{ if (isFull())
     cout<< "\nCannot Insert. Queue is full!";
  else
  { //insert at back
    back++;
    items[back] = insertItem;
  } // end else if
}
```

- **enQueue()**example

`Queue myQueue;`

0	-1						
front	back	0	1	2	3	4	5

`myQueue.enQueue('A');`

0	0	A					
front	back	0	1	2	3	4	5

`myQueue.enQueue('B');`

0	1	A	B				
front	back	0	1	2	3	4	5

`myQueue.enQueue('C');`

0	2	A	B	C			
front	back	0	1	2	3	4	5

- **getFront()** – Retrieve item at front queue.

```
// Program 9.7
char queue:: getFront() // get item at front
{ return items[front] ; }
```

- **getRear()** – Retrieve item at back of the queue.

```
// Program 9.8
char queue::getRear() // get item at back
{ return items[back] ; }
```

  - Statement to access item at front or back of the queue are as follows:

cout << “Item at front queue: “ << myQueue.getFront();
cout << “Item at the back queue:” << myQueue.getRear();

- **deQueue()-** delete from a queue
  - Retrieve item at front queue (if necessary)
  - Increment **front**

```
// Program 9.9
void queue::deQueue()
{ if (isEmpty())
    cout<<"\nCannot remove item.Empty Queue!";
  else
  { //retrieve item at front
    deletedItem = items[front];
    front++;
  } // end else if
}
```

  - **deQueue()**example

```
myQueue.deQueue();
```

deletedItem
A

1	2
front	back

	B	C			
0	1	2	3	4	5

```
myQueue.deQueue();
```

deletedItem
B

2	2
front	back

		C			
0	1	2	3	4	5

```
myQueue.deQueue();
```

deletedItem
C

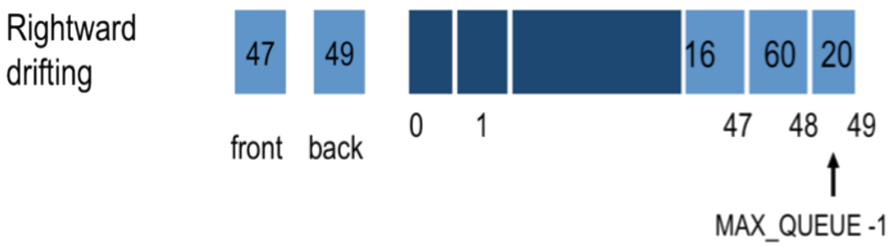
3	2
front	back

		C			
0	1	2	3	4	5

```
myQueue.deQueue();      Cannot remove item. Queue Empty
```

2.5 Linear Queue Array Problem: Rightward-Drifting

- After a sequence of additions and removals, items in the queue will drift towards the end of the array. **enQueue** operation cannot be performed on the queue as shown below, since **back = max_queue – 1**.

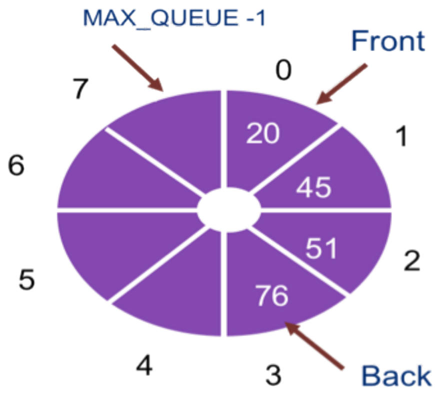


2.6 Rightward drift solutions

- Two solutions can be performed to solve rightward drift
  1. Shift array elements after each deletion to occupy the space being deleted at front
     - However, shifting dominates the cost of the implementation
  2. Use a circular array: When **front** or **back** reach the end of the array, wrap them around to the beginning of the array.
     - However there is a problem
     - **front** and **back** cannot be used as condition to distinguish between queue-full and queue-empty
     - Therefore, use counter as a solution:
       - **count == 0** means empty queue
       - **count == MAX_QUEUE** means full queue

3.0 QUEUE : CIRCULAR ARRAY IMPLEMENTATION

3.1 Circular Queue Array Implementation



3.2 Modifications need to be performed in circular queue

- Queue declaration, add variable **count**

```
//Program 9.10
const int MAX_QUEUE = maximum-size-of-queue;
QueueItemType items [ MAX_QUEUE ];
int      front;
int      back;
int      count
```

- Initial condition - **createQueue()**

```
//Program 9.13
count = 0;
front = 0;
back = MAX_QUEUE – 1;
```

- The Wrap-around effect is obtained by using modulo arithmetic (%-operator)
- Insertion operation - **enQueue()**
  - Increment **back**, using modulo arithmetic
  - Insert item
  - Increment **count**

```
//Program 9.11
back = ( back + 1 ) % MAX_QUEUE;
items[back] = newItem;
++count;
```

- Deletion operation - **deQueue()**
  - Increment **front** using modulo arithmetic
  - Decrement **count**

```
//Program 9.12
front = ( front + 1 ) % MAX_QUEUE;
--count;
```

- Disadvantage
  - The overhead of maintaining a counter or flag
  - Queue Operation examples:
    - **front** passes **back** when the queue becomes empty;
    - **back** catches up to **frontt** when the queue becomes full
    - The figure below shows the **enQueue()** and **deQueue process** implemented in circular array

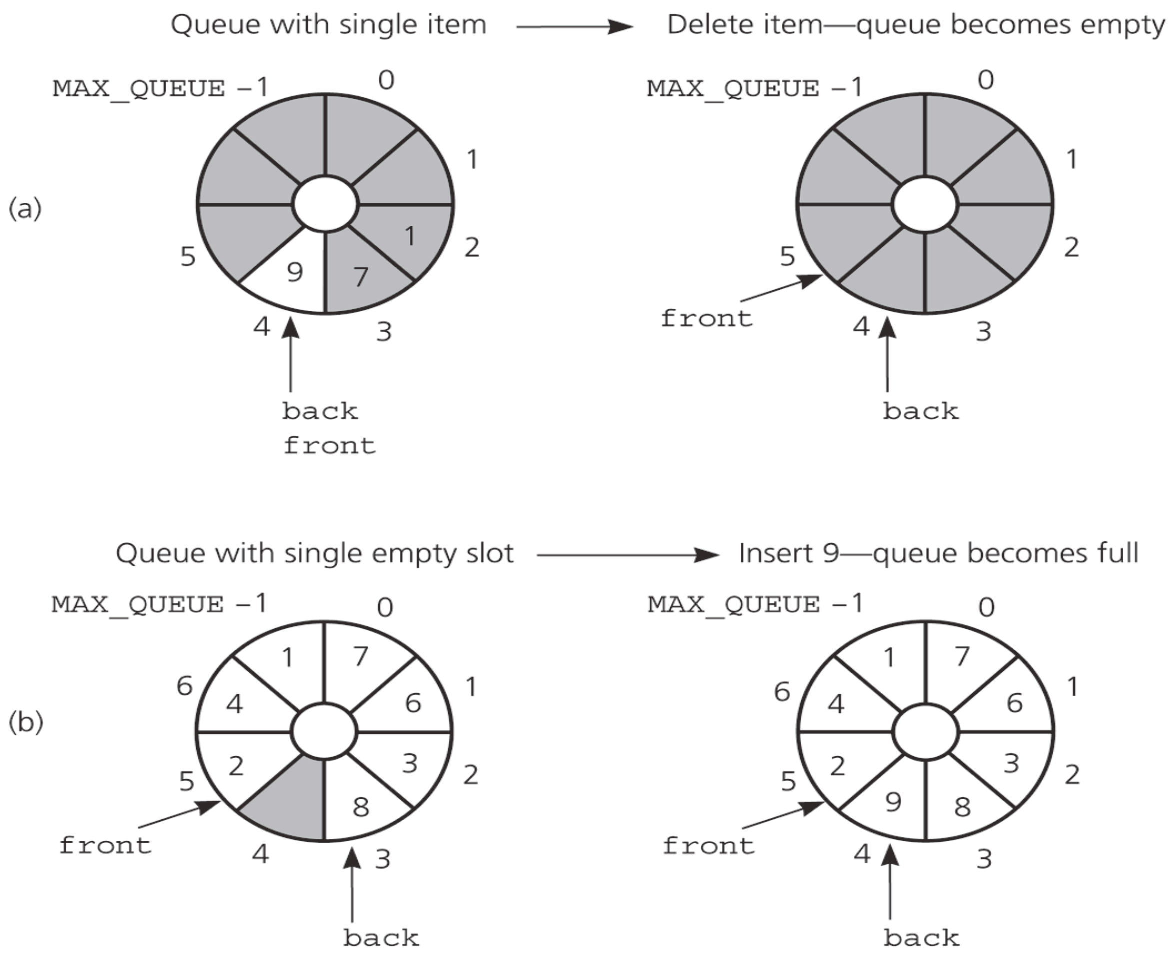
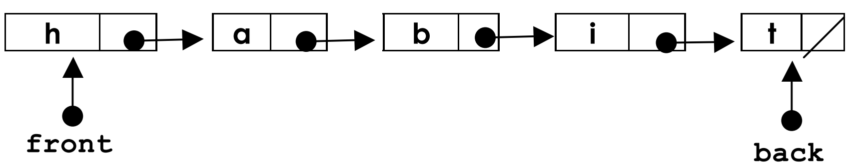


Figure extracted from Carrano, F (2007)

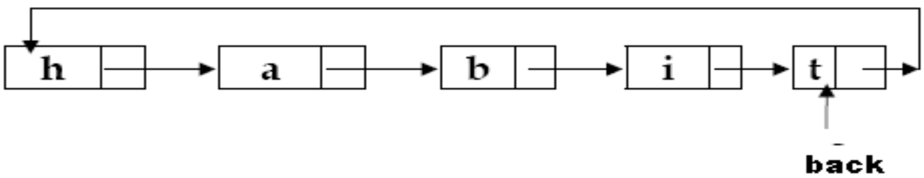
4.0 QUEUE IMPLEMENTATION LINKED LIST

4.1 Pointer-Based Implementation

- More straightforward than array-based
- There are 2 possible options, which are:
  - Linear linked list
    - Have two external pointers **front** and **back** that are used to point to the first node in the queue and point to the last node in the queue



  - Circular linked list
    - Have one external pointer, **back** that point to the last node of the queue.



4.2 Queue Linear Linked List

- Need 2 structures, which are declaration of class node and class queue
  - Declaration of the node with one data item

```
//Program 9.14
struct nodeQ {
                char item;
                nodeQ * next;
}
```

  - Declaration of the queue which has pointers, **backPtr** and **frontPtr**

```
//Program 9.15
class queue
{public:
    nodeQ *backPtr, * frontPtr;
```

```
                    // operations for queue
};
```

- **createQueue()** operation. Initialize **backPtr** and **frontPtr** to **NULL**.

```
//Program 9.16
backPtr = Null; frontPtr = NULL;
```

- **destroyQueue()** operation
  - destroy the whole nodes in the queue

```
//Program 9.17
nodeQ *temp = frontPtr;
while (temp)
{      frontPtr = temp->next;
       delete temp;
       temp=frontPtr;
}
```

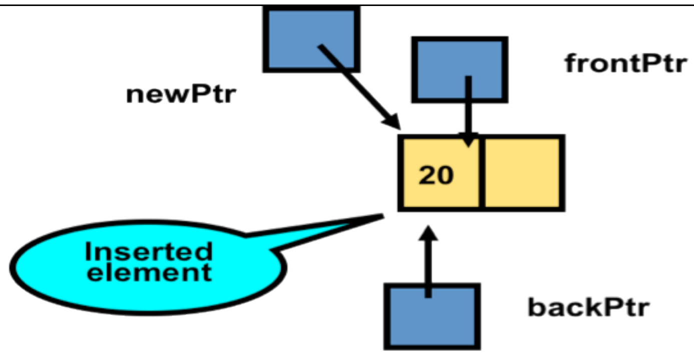
- **isEmpty()** operation

```
//Program 9.18
backPtr == Null && frontPtr == NULL
```

- Insert to a linear queue
  - Inserting a new node at the back queue needs 3 pointer changes
    - Change **next** pointer in the new node
    - Change the **next** pointer in the back node
    - Change the external pointer, **backPtr**
  - Special case:
    - If the queue is empty, both **backPtr** and **frontPtr** will point to the new node.

5.0 QUEUE IMPLEMENTATION: LINEAR LINKED LIST AND CIRCULAR LINKED LIST

5.1 Linear linked list with 2 external pointers



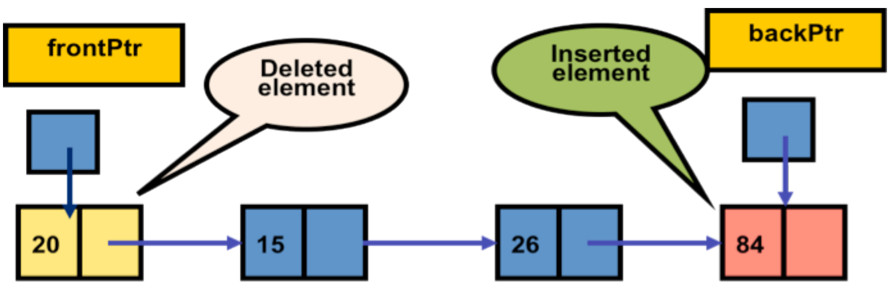
- Insertion into a queue implementation linear linked list
  - Create a new node, **newPtr**
  - Insertion to an empty queue is as follows

```
//Program 9.19
frontPtr = backPtr  = newPtr ;
```

  - Insertion to a non-empty queue is as follows

```
//Program 9.20
newPtr -> next = NULL;
backPtr -> next = newPtr ;
backPtr = newPtr ;
```

- Deletion
  - Delete from the front of the queue
  - Only one pointer change is needed, **frontPtr**
  - Special case:
    - If the queue contains one item only, both **backPtr** and **frontPtr** will be set to **NULL**



  - Deletion code for a queue with more than one node.

```
//Program 9.21
tempPtr = frontPtr
frontPtr = frontPtr -> next
```

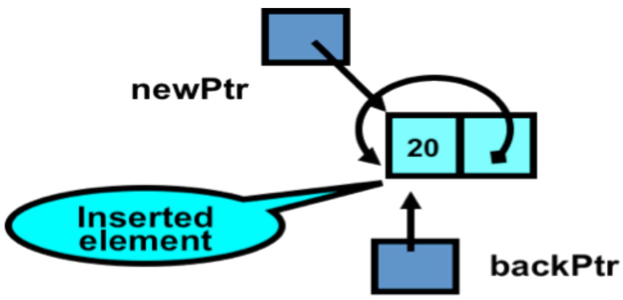
```
tempPtr -> next = NULL
delete tempPtr ;
```

- If the queue contains one item only, need to add this statement

```
//Program 9.22
if (!frontPtr)
    backPtr = NULL;
```

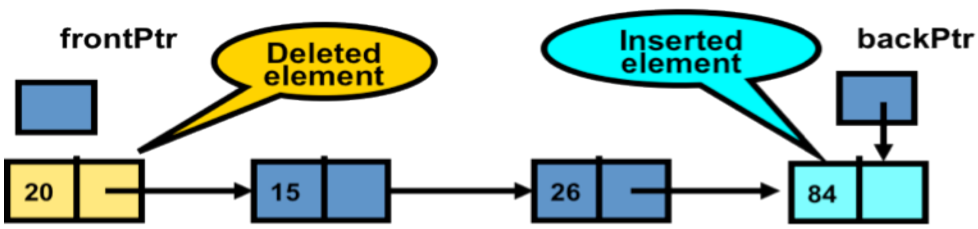
5.2 **Circular linear linked list with one external pointer**

- Insertion operation
  - Insert into an empty queue



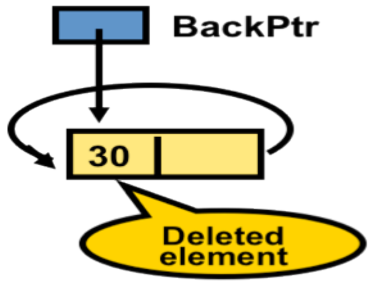

```
//Program 9.23
// the node point to itself
NewPtr -> Next = NewPtr
Back pointer point to the newNode
backPtr = NewPtr
```

  - Insert into a non-empty queue



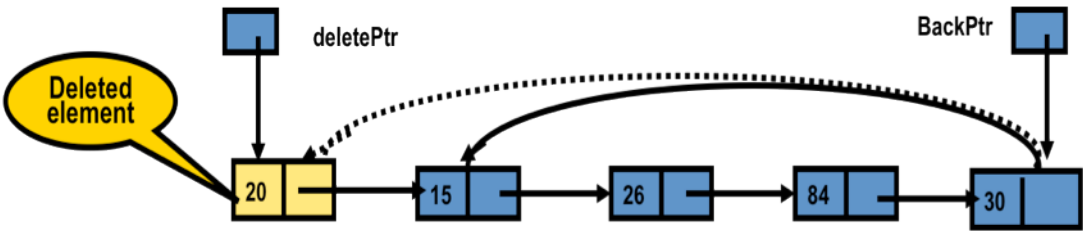

```
//Program 9.24
NewPtr -> Next = backPtr-> Next
backPtr -> Next = NewPtr
backPtr = NewPtr
```

- Delete operation
    - Delete from a queue with one-node in the queue




```
//Program 9.25
deletePtr  = backPtr -> Next
If (deletePtr = backPtr)
   BackPtr = NULL
delete deletePtr
```

- Delete from a non-empty queue, whereby the queue has more than one item




```
//Program 9.26
deletePtr  = backPtr -> Next
backPtr -> Next =  deletePtr -> Next
delete deletePtr
```

5.3 Comparison between array and pointer based implementations

- Fixed size (array) versus dynamic size (pointer based)
    - A statically allocated array
        - Prevents the **enqueue** operation from adding an item to the queue if the array is full
    - A resizable array or a reference-based implementation
        - Does not impose this restriction on the **enqueue** operation

- Pointer-based implementation
    - A linked list implementation

- More efficient in insert and delete operations
- More complicated than Abstract Data Type (ADT) list
- Most flexible, since no size restrictions

- Array-Based
  - No overhead of pointer manipulation
  - Prevents adding elements if the array is full

5.4 Summary of Position Oriented ADTs

- **Stacks :**
  - Operations are defined in terms of position of data items
  - Position is restricted to the Top of the stack. Only one end position can be accessed
  - Operations:
    - **Create():** Creates an empty ADT of the Stack type
    - **isEmpty():** Determines whether an item exists in the ADT
    - **push():** Inserts a new item into the top position
    - **pop():** Deletes an item from the top position
    - **satckTop():** Retrieves the item from the top position

- **Queues :**
  - Operations are defined in terms of position of data items
  - Position is restricted to the front and the back of the queue. Only the end positions can be accessed.
  - Operations:
    - **create ():** Creates an empty ADT of the Queue type
    - **isEmpty():** Determines whether an item exists in the ADT
    - **dequeue():** Deletes an item from the front position
    - **enqueue():** Inserts a new item in the back position
    - **peek():** Retrieves the item from the front position

- **Stacks and queues are very similar in terms of :**
  - Operations of stacks and queues can be paired off as
    - **createStack()** and **createQueue()**
    - Stack **isEmpty()** and queue **isEmpty()**
    - **Push()** and **enqueue()**
    - **Pop()** and **dequeue()**
    - **getTop()** and queue **getFront()**