ADT Queue

A queue is a well understood concept in daily life and is also widely used in computing. A queue follows the simple principle of FIFO (first in first out) as opposed to a stack which uses LIFO (last in first out).

The abstract interface to Queue can be specified in C++ by:

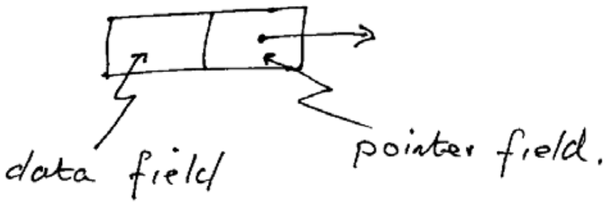
```
class Queue {
public:
    void enQueue(int x);
    int deQueue();
    bool isEmpty();
};
```

We will consider 2 implementation of the ADT Queue:
- linked list
- circular buffer

Remark on Linked Lists

We will construct our linked lists from node structures linked together with pointers. Each node in the simplest case has two fields, a data filed and a pointer to the next node.

Node Structure



```
struct Node {
    int data;
    Node * next;
};
```

Linked List Implementation

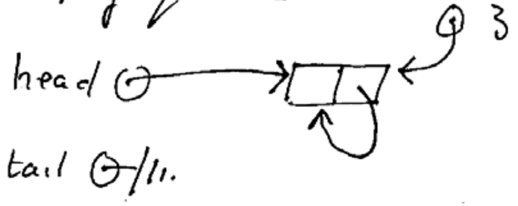
Since new linked list nodes are added to the back of the queue and nodes are removed from the front, it makes sense to use two pointers, *head* and *tail* pointing to the first and last node respectively.

Let us start with adding a value, enQueue(x), to an initially empty queue. For an empty queue, *head* points to a dummy node and *tail* will have a null value.

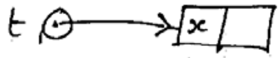
Operation enQueue (int x), insert x into a queue.

Two cases to consider.

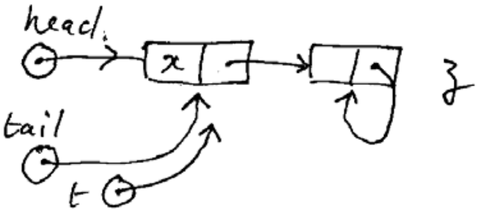
(i) Empty queue



Create a new node, place x in it



Rearrange pointers to insert it into linked list

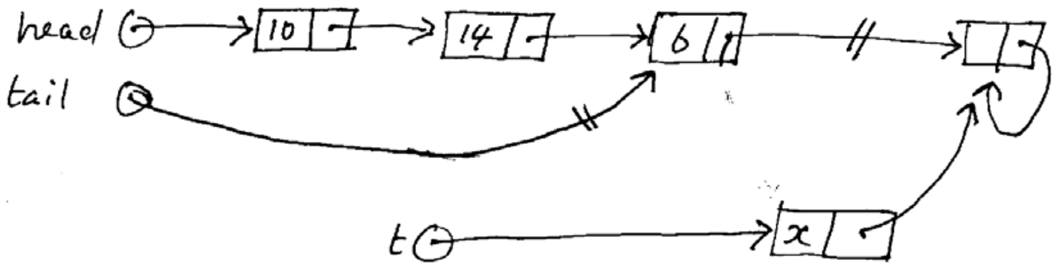


In code :

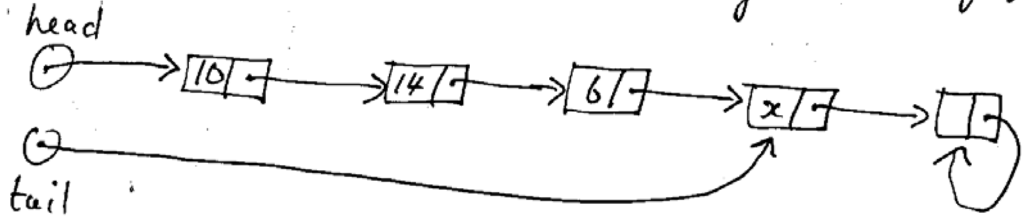
```
t = new Node ;
t -> data = x ;
t -> next = z ;
if ( head == head -> next )
    head = t ;
tail = t ;
```

(ii) Case of non-empty queue, i.e. head != head->next

e.g.



Create and initialise new node as before.
Change pointers as shown in diagram to get:



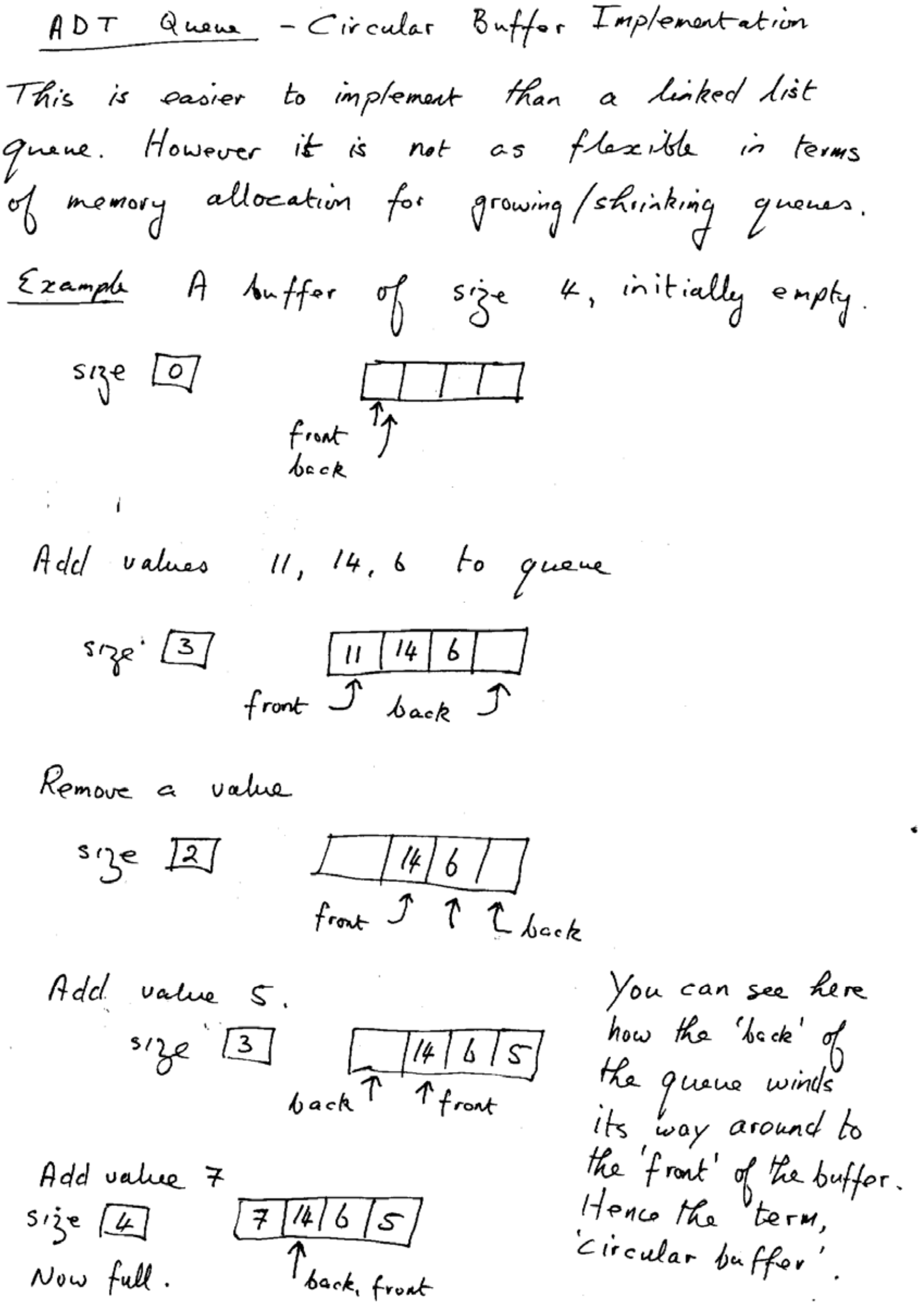
In code, both cases:

```
t = new Node ;
t -> data = x ;
t -> next = z ;
if ( head == head-> next)
    head = t;
else
    tail -> next = t ;
tail = t;
```

Some of our class code now could look like:

```
class Queue {
private:

    struct Node {
      int data;
      Node * next;
    };

    Node * z;
    Node * head;
    Node * tail;

public:
    Queue() {
        z = new Node; z->next = z;
        head = z;  tail = NULL;
    }

    void enQueue(int x);
    int deQueue();
    bool isEmpty();
};

void Queue::enQueue( int x) {
    Node * temp;

    temp = new Node;
    temp->data = x;
    temp->next = z;


    if(tail == NULL)  // case of empty list
        head = temp;
    else              // case of list not empty
        tail->next = temp;

    tail = temp;      // new node is now at the tail
}
```

Circular Buffer Implementation



ADT Queue – Circular Buffer Implementation

This is easier to implement than a linked list queue. However it is not as flexible in terms of memory allocation for growing/shrinking queues.

Example A buffer of size 4, initially empty.

size 0

front back

Add values 11, 14, 6 to queue

size 3

11	14	6	

front back

Remove a value

size 2

	14	6	

front back

Add value 5.

size 3

	14	6	5

back front

You can see here how the 'back' of the queue winds its way around to the 'front' of the buffer. Hence the term, 'circular buffer'.

Add value 7

size 4

7	14	6	5

Now full.

back, front

As you can see, when back is incremented after each enQueue(x), until it reaches the end of the buffer. Then if the beginning of the buffer is free, back will go from the end of the buffer to the start. This is easily implemented using modulo arithmetic.

This can see it in code fragment for the circular buffer Queue implementation:

```
class Queue {

  private:
    int *q, back, front;
    int qmax, size;

  public:
    Queue();
    void enQueue(int x);
    int deQueue();
    bool isEmpty();
    };



Queue::Queue( int _qmax) {
    qmax = _qmax;
    size = front = back = 0;
    q = new int[qmax];
}


void Queue::enQueue( int x)
{
    if( qmax == size) return;

    q[back] = x;
    back = (back + 1) % qmax;
    ++size;
}
```