**Exercise 5**
**Solution:**

You can implement two stacks using a single array by dividing the array into two sections and managing the stack operations within those sections. To ensure that you do not declare an overflow until every slot in the array is used, you can implement a fixed-size stack with a predefined array size and two stack pointers.

```
class TwoStacks {
private:
    std::vector<int> arr;  // The single array to hold both stacks
    int top1;          // Top of the first stack
    int top2;          // Top of the second stack
    int size;          // Total size of the array
public:
    // Constructor to initialize the class with the size of the array
    TwoStacks(int n) {
        size = n;
        arr.resize(n);     // Resize the vector to the specified size
        top1 = -1;         // Initialize top1 to -1, indicating an empty stack
        top2 = n;          // Initialize top2 to n, indicating an empty stack
    }
// Methods to push elements onto the two stacks
    void push1(int x) {
        // Check if there is space for the first stack
        if (top1 < top2 - 1) {
            arr[++top1] = x;  // Increment top1 and push the element onto the first stack
        } else {
            std::cout << "Stack 1 is full, cannot push " << x << std::endl;
        }
    }
    void push2(int x) {
        // Check if there is space for the second stack
        if (top1 < top2 - 1) {
            arr[--top2] = x;  // Decrement top2 and push the element onto the second stack
        } else {
            std::cout << "Stack 2 is full, cannot push " << x << std::endl;
        }
    }
    // Methods to pop elements from the two stacks
    int pop1() {
        if (top1 >= 0) {
            return arr[top1--];  // Pop an element from the first stack and decrement top1
        } else {
            std::cout << "Stack 1 is empty" << std::endl;
            return -1;  // Indicates an empty stack
        }
    }
    int pop2() {
        if (top2 < size) {
            return arr[top2++];  // Pop an element from the second stack and increment top2
        } else {
            std::cout << "Stack 2 is empty" << std::endl;
            return -1;  // Indicates an empty stack
        }
    }
};
```

**Exercise 6**

1)- The advantages are that it is simpler to code, and there is a possible saving if deleted keys are subsequently reinserted (in the same place).

- The disadvantage is that it uses more space, because each cell needs an extra bit (which is typically a byte), and unused cells are not freed.

2)- Implementation

```
class Node {
public:
   int data;
   Node* next;
   bool is_deleted;

   // Constructor to initialize a node with data, initially no next node, and not marked as deleted.
   Node(int value) : data(value), next(nullptr), is_deleted(false) {}
};

class LinkedList {
private:
   Node* head;       // Pointer to the first node in the linked list.
   int deleted_count;  // Count of deleted nodes.
   int size;         // Total count of nodes in the linked list.

public:
..........
}
       //routines to be completed
```

**Exercise 7**

**Solution:**

```
Class CircularQueue {
   maxSize = 20        // maximum size of the queue: Number
                       //of elements

   front = -1  // empty queue
   rear = -1
  element-type array [maxSize]        // new array of size maxSize

public

FUNCTION Front() : element_type
{   IF isEmpty()
      { PRINT ("Queue is empty") ;
       RETURN ;}
   ELSE
      RETURN array[front]
}

FUNCTION Rear() : element_type
{  IF isEmpty()
          { PRINT ("Queue is empty") ;
       RETURN ;}

   ELSE
```

```
      RETURN array[rear]
}

FUNCTION enQueue(value)
{
 IF isFull()
      { PRINT ("Queue is full") ;
        RETURN ;}
   ELSE
     { rear = (rear + 1) % maxSize
       array[rear] = value
           if (front = =  -1)  then { front = 0}
       }
}


FUNCTION deQueue()
   {IF isEmpty()
             { PRINT ("Queue is empty") ;
               RETURN ;}
   ELSE
         If (front == rear ) {// the queue contain one element
          front =- 1 ; rear=-1// the queue becomes empty }
else {
      array[front] = null  // or any placeholder value (ex:  -1)
      front = (front + 1) % maxSize}
}
FUNCTION isEmpty() :boolean
 {  RETURN ((front == -1)  && (rear == -1)}

FUNCTION isFull() : boolean
 {  RETURN ((rear + 1) % maxSize == front)}

FUNCTION search(value) : boolean
  {  int i= front
if ( empty () ) then write ("empty queue, value not found')
else



{ while (array[i]  <> value ) And  (i < > rear  ) do
          {   i =   ( i+1)  % maxsize }

If   (array[i]  == value) return true
Else return false
}

FUNCTION size() : integer
  {  if empty () return 0;
 else RETURN (maxSize - front + rear+1) % maxSize}

FUNCTION Get(index) : element
 {  IF index < 0 OR index >= size()
      RETURN "Invalid index"
   ELSE
      RETURN array[(front + index) % maxSize]}
```

**Time Complexity Analysis:**

| | |
|---|---|
| Front() and Rear() | O(1) |
| enQueue(value) and deQueue() | O(1) |
| isEmpty() , isFull() and size() | O(1) |
| search(value) | O(n), where 'n' is the number of elements in the queue |
| Get(index) | O(1) |