Array-Based Implementations: -

13. Implement Stack using an Array

```
class stack{
    int *arr;
    int top,capacity;
public:
stack(int size){
    capacity=size;
    arr = new int[capacity];
    top =-1;
}
void push(int x){
    if(isFull())return;
    arr[++top]=x;
}
void pop(){
    if(isEmpty())return;
    top--;
}
int peek(){
    if(isEmpty())return -1;
    return arr[top];
}
bool isFull(){return top == capacity-1; }
bool isEmpty(){return top ==-1;}
```

14. Implement Queue using an Array

```
class Queue {
    int front, rear, size;
    int *arr;
public:
    Queue(int capacity) {
        size = capacity;
        arr = new int[size];
        front = -1;
        rear = -1;}
    bool isEmpty() { return front == -1; }
    bool isFull() { return rear == size - 1; }
    void enqueue(int value) {
        if (isFull()) {
            cout << "Queue Overflow!" << endl;
            return;}
        if (front == -1) front = 0;
        arr[++rear] = value;
        cout << value << " enqueued." << endl; }
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow!" << endl;
            return;}
```

```
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow!" << endl;
            return;}
        cout << arr[front] << " dequeued." << endl;
        if (front == rear) {
            front = rear = -1;
        } else {
            front++;}}

    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return -1;}
        return arr[front];}
```

15. Implement Circular Queue using an Array

```
class CircularQueue {
    int front, rear, size;
    int *arr;
public:
    CircularQueue(int capacity) {
        size = capacity;
        arr = new int[size];
        front = -1;
        rear = -1; }
    bool isEmpty() {return front == -1;}
    bool isFull() {return (rear + 1) % size == front; }
    void enqueue(int value) {
        if (isFull()) {
            cout << "Queue Overflow!" << endl;
            return;}
        if (front == -1) front = 0;
        rear = (rear + 1) % size;
        arr[rear] = value;
        cout << value << " enqueued." << endl;}
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow!" << endl;
            return; }
```

```
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow!" << endl;
            return; }
        cout << arr[front] << " dequeued." << endl;
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % size;
        }
    }

    int peek() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return -1;
        }
        return arr[front];
    }
```

16. Implement Deque using an Array

```
class Deque {
    int front, rear, size;
    int *arr;
public:
    Deque(int capacity) {
        size = capacity;
        arr = new int[size];
        front = -1;
        rear = -1;}
    bool isEmpty() {return front == -1;}
    bool isFull() {return (rear + 1) % size == front;}
    void insertFront(int value) {
        if (isFull()) {
            cout << "Deque Overflow!" << endl;
            return;}
        if (front == -1) {
            front = rear = 0;
        } else {
            front = (front - 1 + size) % size;
        }
        arr[front] = value;
        cout << value << " inserted at front." << endl;
    }
```

```cpp
    void insertRear(int value) {
        if (isFull()) {
            cout << "Deque Overflow!" << endl;
            return;}
        if (rear == -1) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % size;}
        arr[rear] = value;
        cout << value << " inserted at rear." << endl;}
    void deleteFront() {
        if (isEmpty()) {
            cout << "Deque Underflow!" << endl;
            return;
        }
        cout << arr[front] << " deleted from front." << endl;
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % size;
        }
    }
```

```cpp
    void deleteRear() {
        if (isEmpty()) {
            cout << "Deque Underflow!" << endl;
            return; }
        cout << arr[rear] << " deleted from rear." << endl;
        if (front == rear) {
            front = rear = -1;
        } else {
            rear = (rear - 1 + size) % size; }}
    int getFront() {
        if (isEmpty()) {
            cout << "Deque is empty!" << endl;
            return -1; }
        return arr[front]; }

    int getRear() {
        if (isEmpty()) {
            cout << "Deque is empty!" << endl;
            return -1; }
        return arr[rear];}
```

17. Implement Two Stacks in One Array

```cpp
class TwoStacks {
    int *arr, top1, top2, size;
public:
    TwoStacks(int capacity) {
        size = capacity;
        arr = new int[size];
        top1 = -1;
        top2 = size;}
    void push1(int value) {
        if (top1 < top2 - 1) {
            arr[++top1] = value;
            cout << value << " pushed to Stack 1." << endl;
        } else {
            cout << "Stack 1 Overflow!" << endl;    } }

    void push2(int value) {
        if (top1 < top2 - 1) {
            arr[--top2] = value;
            cout << value << " pushed to Stack 2." << endl;
        } else {
            cout << "Stack 2 Overflow!" << endl;
        }
    }
```

```
    void pop1() {
        if (top1 >= 0) {
            cout << arr[top1--] << " popped from Stack 1." << endl;
        } else {
            cout << "Stack 1 Underflow!" << endl;}}
    void pop2() {
        if (top2 < size) {
            cout << arr[top2++] << " popped from Stack 2." << endl;
        } else {
            cout << "Stack 2 Underflow!" << endl;}}
    int peek1() {
        return (top1 >= 0) ? arr[top1] : -1;}
    int peek2() {
        return (top2 < size) ? arr[top2] : -1;}
```

18. Implement k Stacks in a Single Array

```
class KStacks {
    int *arr, *top, *next;
    int freeSpot, size, k;
public:
    KStacks(int numStacks, int capacity) {
        k = numStacks;
        size = capacity;
        arr = new int[size];
        top = new int[k];
        next = new int[size];
        for (int i = 0; i < k; i++) top[i] = -1;
        for (int i = 0; i < size - 1; i++) next[i] = i + 1;
        next[size - 1] = -1;
        freeSpot = 0; }
    bool isFull() {return freeSpot == -1;}
    bool isEmpty(int stackNum) {return top[stackNum] == -1;}
    void push(int stackNum, int value) {
        if (isFull()) {
            cout << "Stack Overflow!" << endl;
            return;}
        int index = freeSpot;
        freeSpot = next[index];
        arr[index] = value;
```

```
        arr[index] = value;
        next[index] = top[stackNum];
        top[stackNum] = index;
        cout << value << " pushed to Stack " << stackNum << "." << endl;}

    void pop(int stackNum) {
        if (isEmpty(stackNum)) {
            cout << "Stack " << stackNum << " Underflow!" << endl;
            return;}
        int index = top[stackNum];
        top[stackNum] = next[index];
        next[index] = freeSpot;
        freeSpot = index;
        cout << arr[index] << " popped from Stack " << stackNum << "." << endl;}

    int peek(int stackNum) {
        if (isEmpty(stackNum)) {
            cout << "Stack " << stackNum << " is empty!" << endl;
            return -1;}
        return arr[top[stackNum]];}
```

19. Implement k Queues in a Single Array

```
class KQueues {
    int *arr, *front, *rear, *next;
    int freeSpot, size, k;
public:
    KQueues(int numQueues, int capacity) {
        k = numQueues;
        size = capacity;
        arr = new int[size];
        front = new int[k];
        rear = new int[k];
        next = new int[size];
        for (int i = 0; i < k; i++) front[i] = rear[i] = -1;
        for (int i = 0; i < size - 1; i++) next[i] = i + 1;
        next[size - 1] = -1;
        freeSpot = 0; }
    bool isFull() { return freeSpot == -1; }
    bool isEmpty(int queueNum) {  return front[queueNum] == -1; }
    void enqueue(int queueNum, int value) {
        if (isFull()) {
            cout << "Queue Overflow!" << endl;
            return; }
        int index = freeSpot;
        freeSpot = next[index];
```

```
        } else {
            next[rear[queueNum]] = index;
            rear[queueNum] = index;}
        next[index] = -1;
        cout << value << " enqueued to Queue " << queueNum << "." << endl;}
    void dequeue(int queueNum) {
        if (isEmpty(queueNum)) {
            cout << "Queue " << queueNum << " Underflow!" << endl;
            return; }
        int index = front[queueNum];
        front[queueNum] = next[index];
        if (front[queueNum] == -1) rear[queueNum] = -1;
        next[index] = freeSpot;
        freeSpot = index;
        cout << arr[index] << " dequeued from Queue " << queueNum << "." << endl;   }
    int peek(int queueNum) {
        if (isEmpty(queueNum)) {
            cout << "Queue " << queueNum << " is empty!" << endl;
            return -1;
        }return arr[front[queueNum]];}
```

20. Implement Min Heap using an Array

```
class MinHeap {
private:
    int *heap;
    int capacity, size;
    int parent(int i) { return (i - 1) / 2; }
    int leftChild(int i) { return 2 * i + 1; }
    int rightChild(int i) { return 2 * i + 2; }
    void heapifyUp(int i) {
        while (i > 0 && heap[parent(i)] > heap[i]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);}}
    void heapifyDown(int i) {
        int smallest = i;
        int left = leftChild(i);
        int right = rightChild(i);

        if (left < size && heap[left] < heap[smallest]) smallest = left;
        if (right < size && heap[right] < heap[smallest]) smallest = right;

        if (smallest != i) {
            swap(heap[i], heap[smallest]);
            heapifyDown(smallest);
        }
```

```
public:
    MinHeap(int cap) {
        capacity = cap;
        size = 0;
        heap = new int[capacity]; }
    void insert(int val) {
        if (size == capacity) {
            cout << "Heap Overflow!" << endl;
            return;}
        heap[size] = val;
        heapifyUp(size);
        size++;}
    void deleteMin() {
        if (size == 0) {
            cout << "Heap Underflow!" << endl;
            return; }
        heap[0] = heap[size - 1];
        size--;
        heapifyDown(0); }
```

21. Implement Max Heap using an Array

```
class MaxHeap {
    int *heap;
    int capacity, size;
    int parent(int i) { return (i - 1) / 2; }
    int leftChild(int i) { return 2 * i + 1; }
    int rightChild(int i) { return 2 * i + 2; }
    void heapifyUp(int i) {
        while (i > 0 && heap[i] > heap[parent(i)]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);}}
    void heapifyDown(int i) {
        int largest = i;
        int left = leftChild(i);
        int right = rightChild(i);
        if (left < size && heap[left] > heap[largest])
            largest = left;
        if (right < size && heap[right] > heap[largest])
            largest = right;
        if (largest != i) {
            swap(heap[i], heap[largest]);
            heapifyDown(largest);
        }}
```

```
public:
    MaxHeap(int cap) {
        capacity = cap;
        size = 0;
        heap = new int[capacity];}
    void insert(int value) {
        if (size == capacity) {
            cout << "Heap Overflow!" << endl;
            return;}
        heap[size] = value;
        heapifyUp(size);
        size++;
        cout << value << " inserted." << endl;}
    int extractMax() {
        if (size == 0) {
            cout << "Heap Underflow!" << endl;
            return -1;}
        int maxVal = heap[0];
        heap[0] = heap[size - 1];
        size--;
        heapifyDown(0);
        return maxVal; }
```

22. Implement Hash Table using an Array (Linear Probing & Chaining)

Linear Probing:

```
class HashTableLinear {
    int *table, *status;
    int capacity;
    int hashFunction(int key) {
        return key % capacity;}
public:
    HashTableLinear(int size) {
        capacity = size;
        table = new int[capacity];
        status = new int[capacity];
        for (int i = 0; i < capacity; i++) {
            table[i] = -1;
            status[i] = 0;}}
    void insert(int key) {
        int index = hashFunction(key);
        int start = index;
        while (status[index] == 1) {
            index = (index + 1) % capacity;
            if (index == start) {
                cout << "Hash table is full!" << endl;
                return;
            }
```

```
bool search(int key) {
    int index = hashFunction(key);
    int start = index;
    while (status[index] != 0) {
        if (status[index] == 1 && table[index] == key)
            return true;
        index = (index + 1) % capacity;
        if (index == start) break;}
    return false;}
void remove(int key) {
    int index = hashFunction(key);
    int start = index;
    while (status[index] != 0) {
        if (status[index] == 1 && table[index] == key) {
            status[index] = 2;
            cout << key << " deleted." << endl;
            return; }
        index = (index + 1) % capacity;
        if (index == start) break;}
    cout << key << " not found!" << endl;}
```

23. Implement Trie using an Array

```
class TrieNode {
public:
    TrieNode *children[26];
    bool isEndOfWord;
    TrieNode() {
        isEndOfWord = false;
        for (int i = 0; i < 26; i++)
            children[i] = nullptr;}};
class Trie {
    TrieNode *root;
public:
    Trie() { root = new TrieNode();}
    void insert(string word) {
        TrieNode *node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index])
                node->children[index] = new TrieNode();
            node = node->children[index];}
        node->isEndOfWord = true;
    }
```

```cpp
    bool search(string word) {
        TrieNode *node = root;
        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index])
                return false;
            node = node->children[index]; }
        return node->isEndOfWord;}
    bool startsWith(string prefix) {
        TrieNode *node = root;
        for (char c : prefix) {
            int index = c - 'a';
            if (!node->children[index])
                return false;
            node = node->children[index];}
        return true; }};
```

24. Implement Graph using Adjacency Matrix (2D Array)

```cpp
class Graph {
    int **adjMatrix;
    int vertices;
public:
    Graph(int v) {
        vertices = v;
        adjMatrix = new int *[vertices];
        for (int i = 0; i < vertices; i++) {
            adjMatrix[i] = new int[vertices];
            for (int j = 0; j < vertices; j++)
                adjMatrix[i][j] = 0; } }
    void addEdge(int u, int v, int weight = 1) {
        adjMatrix[u][v] = weight;
        adjMatrix[v][u] = weight;}
    void removeEdge(int u, int v) {
        adjMatrix[u][v] = 0;
        adjMatrix[v][u] = 0;}
    void display() {
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                cout << adjMatrix[i][j] << " ";}
            cout << endl; }}
```