Multi-dimensional arrays and Decks

Single-dimensional or 1D arrays

type A[N]; // elements A[0], A[1], …, A[N–1]

0	1	2	3	…	k	…	N–1
A_0	A_1	A_2	A_3		A_k		A_{N-1}

Address of element A[k] = (base address) + k * (element size)

Multi-dimensional arrays

2D arrays

type A[M}[N]; // elements A[0][0] … A[M–1][N–1]

	0	1	2	3	…	c	…	N–1
0	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$		$A_{0,c}$		$A_{0,N-1}$
1	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$		$A_{1,c}$		$A_{1,N-1}$
2	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$		$A_{2,c}$		$A_{2,N-1}$
3	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$		$A_{3,c}$		$A_{3,N-1}$
r	$A_{r,0}$	$A_{r,1}$	$A_{r,2}$	$A_{r,3}$		$A_{r,c}$		$A_{r,N-1}$
M–1	$A_{M-1,0}$	$A_{M-1,1}$	$A_{M-1,2}$	$A_{M-1,3}$		$A_{M-1,c}$		$A_{M-1,N-1}$

In C and C++, 2D arrays are stored in row-major order:

Address of element A[r][c] = (base address) + r * N * (element size)

+ c * (element size)

In some languages, 2D arrays are stored in column-major order:

Address of element A[r][c] = (base address) + c * M * (element size)

+ r * (element size)

3D arrays

type A[M}[N][P];   // elements A[0][0][0] … A[M–1][N–1][P–1]

A[0][…][…]	0	…	z	…	P–1
0	$A_{0,0,0}$		$A_{0,0,z}$		$A_{0,0,P-1}$
y	$A_{0,y,0}$		$A_{0,y,z}$		$A_{0,y,P-1}$
N–1	$A_{0,N-1,0}$		$A_{0,N-1,z}$		$A_{0,N-1,P-1}$

A[x][…][…]	0	…	z	…	P–1
0	$A_{x,0,0}$		$A_{x,0,z}$		$A_{x,0,P-1}$
y	$A_{x,y,0}$		$A_{x,y,z}$		$A_{x,y,P-1}$
N–1	$A_{x,N-1,0}$		$A_{x,N-1,z}$		$A_{x,N-1,P-1}$

A[M–1][…][…]	0	…	z	…	P–1
0	$A_{M-1,0,0}$		$A_{M-1,0,z}$		$A_{M-1,0,P-1}$
y	$A_{M-1,y,0}$		$A_{M-1,y,z}$		$A_{M-1,y,P-1}$
N–1	$A_{M-1,N-1,0}$		$A_{M-1,N-1,z}$		$A_{M-1,N-1,P-1}$

In C and C++, 3D arrays are stored in row-major order:

Address of element A[x][y][z] = (base address) + x * N * P * (element size)

+ y * P * (element size) + z * (element size)

In some languages, 3D arrays are stored in column-major order:

Address of element A[x][y][z] = (base address) + z * M * N * (element size)

+ y * M * (element size) + x * (element size)

Higher-dimensional arrays

k-D arrays

type A[D_1}[D_2]…[D_k]; // elements A[0][0]…[0] to A[D_1–1][D_2–1]…[D_k–1]

Higher dimensions can get difficult to visualize graphically

In C and C++, k-D arrays are stored in row-major order:

A[0][0]…[0], A[0][0]…[1], A[0][0]…[2], …, A[0][0]…[D_k–1], …

Address of element A[j_1][j_2]…[j_k] =

(base address) + $\Sigma_{1\le a\le k}$ (j_a * $\Pi_{a<b\le k}$ D_b) * (element size)

In some languages, k-D arrays are stored in column-major order:

A[0][0]…[0], A[1][0]…[0], A[2][0]…[0], …, A[D_1–1][0]…[0], …

Address of element A[j_1][j_2]…[j_k] =

(base address) + $\Sigma_{1\le a\le k}$ (j_a * $\Pi_{1\le b<a}$ D_b) * (element size)

Decks (or double-ended queues or DEQs or deques)

Abstract data type that supports these operations:

- insertFirst(x): insert item x at beginning of the deck
- insertLast(x): insert item x at end of the deck
- removeFirst( ): remove and return item at beginning of the deck
- removeLast( ): remove and return item at end of the deck

Usually also supports these operations:

- first( ): return item at beginning of the deck without removing it
- last( ): return item at end of the deck without removing it
- size( ): return the number of items currently in the deck
- isEmpty( ): return true if the size is 0, otherwise return false

Stacks can be implemented two ways using a Deck:

push(x) = insertFirst(x) pop( ) = removeFirst( ) top( ) = first( )	push(x) = insertLast(x) pop( ) = removeLast( ) top( ) = last( )

Queues can also be implemented two ways using a Deck:

enqueue(x) = insertFirst(x) dequeue( ) = removeLast( ) front( ) = last( )	enqueue(x) = insertLast(x) dequeue( ) = removeFirst( ) front( ) = first( )

Every operation should run in O(1) time

Deck implemented as a Circular Array

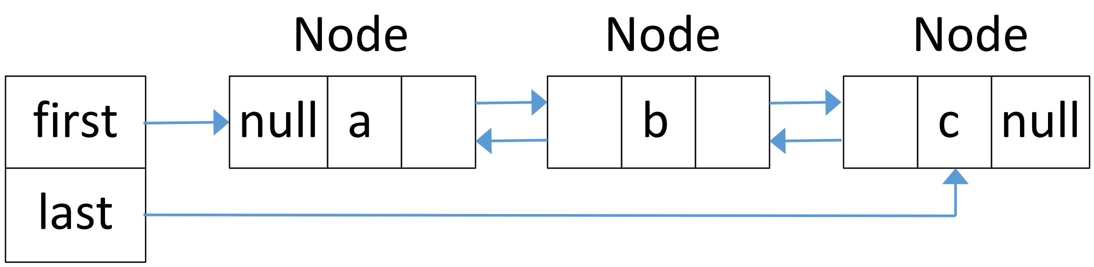
	0	1	2	3	4	5	6	7	…	MAX–1
array			a	b	c	d	e			
			first				last			

	0	1	2	3	…	MAX–3	MAX–2	MAX–1
array	d	e				a	b	c
		last				first		

```
class Deck {
        Type  array[MAX];
        int first, last, n;
        Deck( ) {
                first = 0;  last = MAX–1;  n = 0;
        }
        void insertLast (Type  x) {
                if (isFull( )) throw exception;
                last = (last + 1) % MAX;
                array[last] = x;
                n += 1;
        }
        void insertFirst (Type  x) {
                if (isFull( )) throw exception;
                first = (first + MAX – 1) % MAX;
                array[first] = x;
                n += 1;
        }
```

```
    Type  removeFirst( ) {
        if (isEmpty( )) throw exception;
        Type x = array[first];
        first = (first + 1) % MAX;
        n –= 1;
        return x;
    }
    Type  removeLast( ) {
        if (isEmpty( )) throw exception;
        Type x = array[last];
        last = (last + MAX – 1) % MAX;
        n –= 1;
        return x;
    }
    Type  first( ) {
        if (isEmpty( )) throw exception;
        return array[first];
    }
    Type  last( ) {
        if (isEmpty( )) throw exception;
        return array[last];
    }
    boolean isEmpty( ) { return n == 0; }
    boolean isFull( ) { return n == MAX; }
    int size( ) { return n; }
}
```

Deck implemented as a Doubly-Linked List




```
class Node {
        Type  data;
        Node prev, next;
        Node (Type x, Node p, Node q) {
                data=x;  prev=p; next=q;
        }
}
class Deck {
        Node first, last;
        int n;
        Deck( ) {
                first=null;  last=null;  n=0;
        }
        void insertFirst (Type  x) {
                Node t = new Node (x, null, first);
                if (isEmpty( ))  last = t;  else  first.prev = t;
                first = t;
                n += 1;
        }
        void insertLast (Type  x) {
                Node t = new Node (x, last, null);
                if (isEmpty( ))  first = t;  else  last.next = t;
```

```
            last = t;
            n += 1;
        }
        Type  removeFirst( ) {
            if (isEmpty( )) throw exception;
            Type x = first.data;
            first = first.next;
            if (first == null)  last = null;  else first.prev = null;
            n –= 1;
            return x;
        }
        Type  removeLast( ) {
            if (isEmpty( )) throw exception;
            Type x = last.data;
            last = last.prev;
            if (last == null)  first = null;  else last.next = null;
            n –= 1;
            return x;
        }
        Type  first( ) {
            if (isEmpty( )) throw exception;
            return first.data;
        }
        Type  last( ) {
            if (isEmpty( )) throw exception;
            return last.data;
        }
        boolean isEmpty( ) { return n==0; }
        int size( ) { return n; }
}
```