Dynamic Sets

- Sets in computer science change over time

⇒ Dynamic sets

- May contain a unique object attribute called "key"
- support common operations: insertion, deletion, membership/search

Search (S, k) : k: key , S : set

Insert (S, x) : x: object (or pointer)

Delete (S, x)

Minimum(S), Maximum(S) : returns object with min/max key.

successor(S, x), Predecessor(S, x) : returns next/prev object in an ordered set S

Arrays

- Contigous memory for story elements of a set.
  - fixed length
  - Contigous
  - Same type of objects

- Indexing an array
  - row major (a.k.a. Scan order)
    (Left → right) – (Top → bottom)
  - column major
    (top → bottom) – (Left → right)

  - Given starting address a, each element occupying b bytes in a s-based indexing (s=0, or s=1)

    i^{th} element can be accessed at

    $a + b(i-s)$ ← starting address

  - 2D arrays

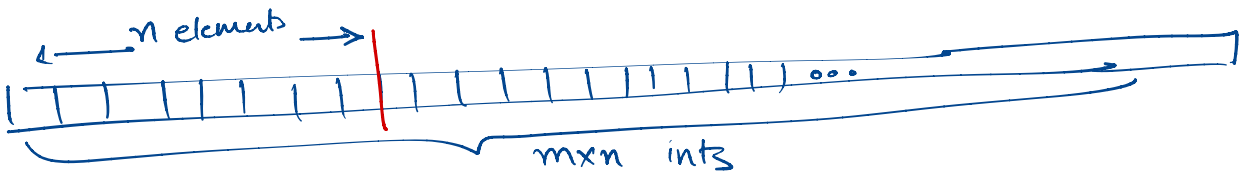
    allocate an mxn array (of ints) in C :

    int *mat = (int *) malloc (no. of elements * sizeof(elem))

    = (int *) malloc (m * n * sizeof(int))

    Access an element at (i, j) location

    mat[i * n + j] or * (mat + i * n + j)



Stacks

- A dynamic set implementation where:
  - → Implements a LIFO policy
    - ↳ Last-in, first-out
  - → Insert is called Push
  - → Delete is called Pop

- Array-based implementation
  - → a stack of atmost size n is implemented by an array $S[1..n]$
  - → Attributes: $S.top$: index of the most recently inserted element
  - → $S[1]$ is the bottom of stack
  - → when $S.top = 0$, stack is empty "underflow"
  - → when $S.top = S.size$, stack is full "overflow"

Stack-Empty (S)

```
if S.top == 0
    return true
else return false
```

O(1)

Pop (S)

```
if Stack-Empty (S)
    error "underflow"
else S.top = S.top - 1
     return S[S.top + 1]
```

O(1)

Push (S, x)

```
if S.top == S.size
    error "overflow"
else S.top = S.top + 1
     S[S.top] = x
```

O(1)

Queues

- In a queue, element are deleted at one end and inserted at the other end
  - ↳ Implements FIFO policy
    - ↳ First-in, first-out
  - → Insert op. is called Enqueue
  - → Delete op. is called Dequeue
  - → Two attributes Head (for dequeue)
    Tail (for enqueue)

→ Array based implementation
- ↳ A queue of at most $(n-1)$ elements is implemented by an array $Q[1..n]$
  - $Q.head$ → index of head
  - $Q.tail$ → index of next enqueue location
- → Elements reside in $Q.head, Q.head+1, \ldots Q.tail-1$
- → Implement a "Circular indexing"
  1 follows n
- → Queue is empty when $Q.head == Q.tail$
  (initially $Q.head = Q.tail = 1$)
- → Queue is full when $Q.head = Q.tail + 1$
  OR (both $Q.head = 1$ AND $Q.tail = Q.size$)

Dequeue happens at head
Enqueue happens at tail.

Assume $n = Q.size$

Enqueue (Q, x)

```
Q[Q.tail] = x
if Q.tail == Q.size
        Q.tail = 1
else    Q.tail = Q.tail + 1
```

→ Check for "overflow"

Dequeue (Q)

```
x = Q[Q.head]
if Q.head == Q.size
        Q.head = 1
else    Q.head = Q.head + 1
return x
```

→ Check for "underflow"