Lists

**Definition:** A *list* is an ordered sequence of *vertices* where associated with each vertex is a *data* item and a *next* vertex.

**Special Types of Lists:**

**Definition:** A *queue* is a special type of list where the only operations are inserting at one end and deleting from the other end.

**Definition:** A *stack* is a special type of list where the only operations are inserting and deleting items from the same end.
Array Implementation of Stacks

• An array $S[0] \ldots S[n-1]$.

• $top$ is the first unused position (initially 0).

• PUSH places a data item $d$ on top of the stack $S$.

• POP removes (and returns) the highest data item.
(array implementation of stacks continued)

**procedure** PUSH\((d,S)\)

if \(top \geq n\) then \{stack overflow\}
else begin
    \(S[top] := d\)
    \(top := top + 1\)
end
end

**function** POP\((S)\)

if \(top \leq 0\) then \{stack underflow\}
else begin
    \(top := top - 1\)
    return \(S[top]\)
end
end

(array implementation of stacks continued)
Simplified Stack Notation:

We sometimes use a simplier version of POP, called TOP, that simply returns data item on top of the stack $S$:

```plaintext
function TOP(S)
    if top ≤ 0 then {error}
    else return S[top–1]
end
```

When $S$ is understood, we may omit it as an argument (or in an object oriented programming language, the syntax might be something like S.PUSH, S.POP, etc.).

We sometimes use POP as a procedure (so it decrements $top$ and returns nothing).
Example: Using a Stack to Reverse an Array

Initialize an empty stack capable of holding \( n \) elements.

\[
\text{for } i=1 \text{ to } n \text{ do PUSH}(A[i])
\]

\[
\text{for } i=1 \text{ to } n \text{ do } A[i]=\text{POP}
\]

** Note that it is possible to reverse \( A \) in place.
Array Implementation of Queues ("Circular Queues")

- An array $Q[0] \ldots Q[n-1]$.
- Variables $front$, $rear$, and $size$ (all initially 0).
- $ENQUEUE$ places a data item at the rear of $Q$.
- $DEQUEUE$ removes from the front of $Q$.

Circular Implementation:

- As items are inserted (and $rear$ is incremented) and items are deleted (and $front$ is incremented), data migrates to the right.
- To avoid a jam at the right, allow the data to wrap back to position 0.
(circular queues continued)

procedure ENQUEUE(d,Q)
    if size ≥ n then \{queue overflow\}
    else begin
        Q[rear] := d
        rear := rear+1; if rear=n then rear := 0
        size := size+1
    end
end

function DEQUEUE(Q)
    if size=0 then \{queue underflow\}
    else begin
        temp := Q[front]
        front := front+1; if front=n then front := 0
        size := size–1
        return temp
    end
end
FRONT and REAR return the front and rear data items:

function FRONT(Q)
    if size=0 then {error}
    else return Q[front]
end

function REAR(Q)
    if size=0 then {error}
    else if rear=0 then return Q[n–1]
    else return Q[rear–1]
end
Simplified Queue Notation:

• When $Q$ is understood, we may omit it as an argument (or in an object oriented programming language, the syntax might be something like $Q$.ENQUEUE, $Q$.DEQUEUE, etc).

• Also, we sometimes use DEQUEUE as a procedure (so it deletes and returns nothing).
Example: Using a Queue to Separate an Array
(Rearrange $A[1]...A[n]$ so that the odd numbered elements are in the first half and the evens in the second half).

Initialize empty queues, $Q1$ and $Q2$.

for $i=1$ to $n$ do
  if $i$ is odd then ENQUEUE($A[i],Q1$)
  else ENQUEUE($A[i],Q2$)
end

for $i=1$ to $\lceil n/2 \rceil$ do $A[i]=DEQUEUE(Q1)$
for $i=\lceil n/2 \rceil +1$ to $n$ do $A[i]=DEQUEUE(Q2)$

** It is possible to use less space.
Testing If a Stack or Queue is Empty

*** Rather than having global variables $top$ and $size$ that we can read, we can complete our definition of the abstract data types STACK and QUEUE by defining the functions QSIZE and SSIZE that return the current values of these variables, and also for convenience, the functions SEMPTY and QEMPTY that return true if and only if the structure is currently empty.