LIST STRUCTURES

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

WHAT IS A DATA STRUCTURE?

- Usually (FORTRAN programmers) use arrays
- A different column for each different class of information
- Ex: airline reservation system
  for each passenger on a specific flight:
  1. name
  2. address
  3. phone #
  4. seat #
  5. destination (on a multi-stop flight)
- Notes:

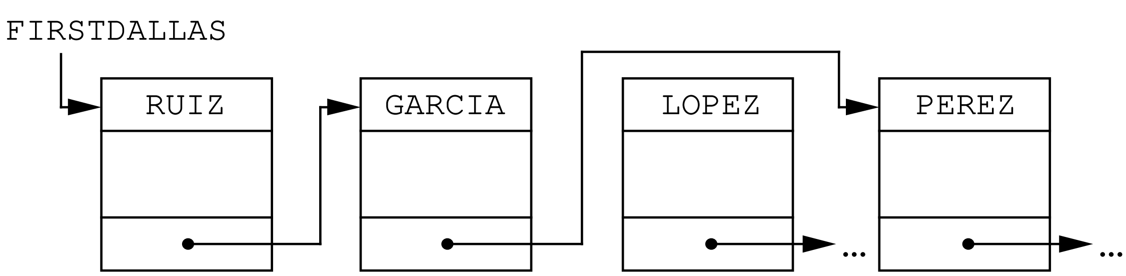
1. not all fields contain numeric information
2. fields need not correspond to whole computer words
   - sex is binary
   - several fields can be packed into one word
   - some fields can occupy more than one word

DIFFERENT REPRESENTATIONS FOR NUMBERS DEPENDING ON THEIR USE:

- Type
  1. BCD
     - social security number 123-45-6789
     - telephone number (123) 456-7890
     - can print character by character by shifting rather than modulo division
  2. ASCII
  3. Fieldata

- Manner of using the data may dictate the representation
  1. sometimes a dual representation – deck of cards
  2. string and numeric

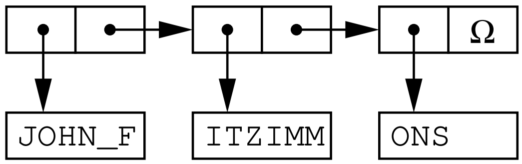
- Ex: airline reservation system
  - Los Angeles → Dallas → Baltimore
  - task: find all passengers with the same destination
  - field: SAMEDEST (LINK or pointer information)



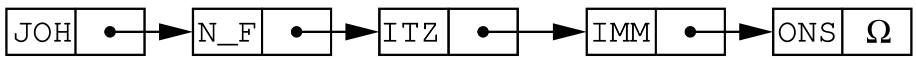
  - alternatively, scan through the passenger list each time the query is posed

CHARACTER DATA

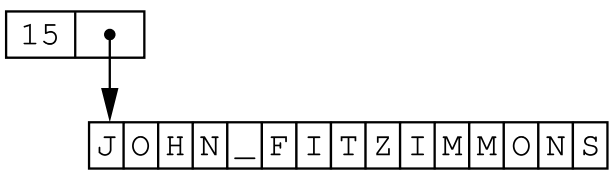
1.



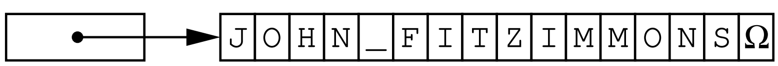
2.



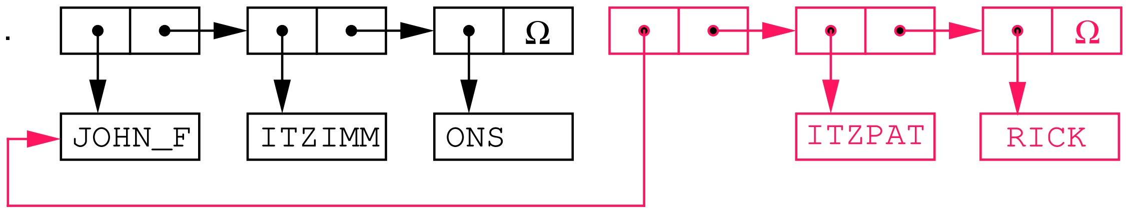
3.

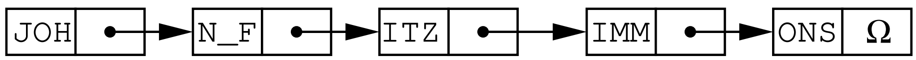


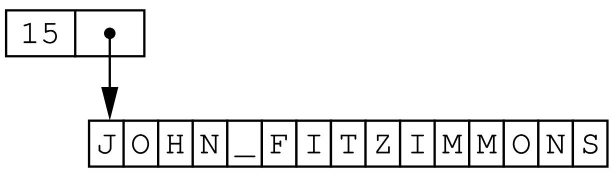
4.

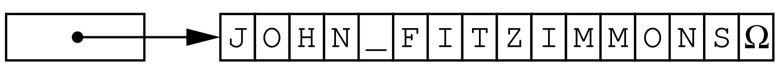


CHARACTER DATA

1. 

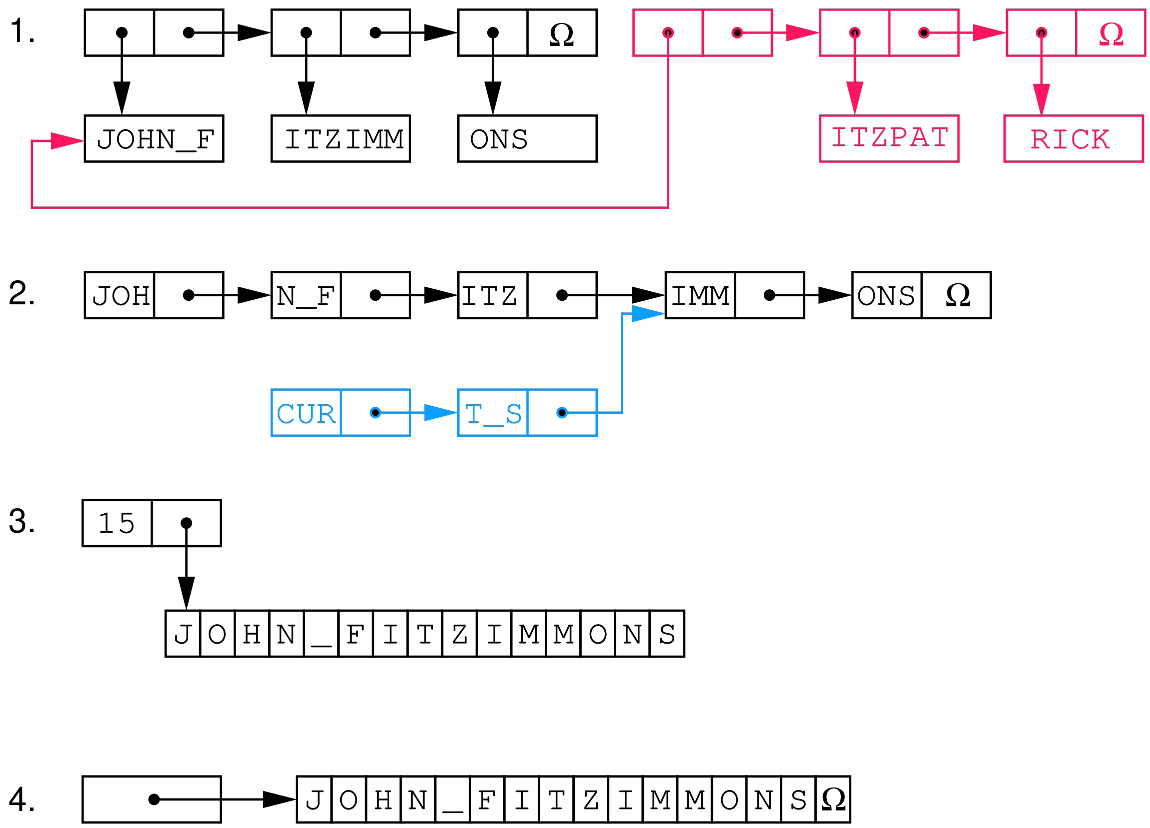
2. 

3. 

4. 

- 1 permits sharing arbitrary segments of strings (start, middle, end)

CHARACTER DATA

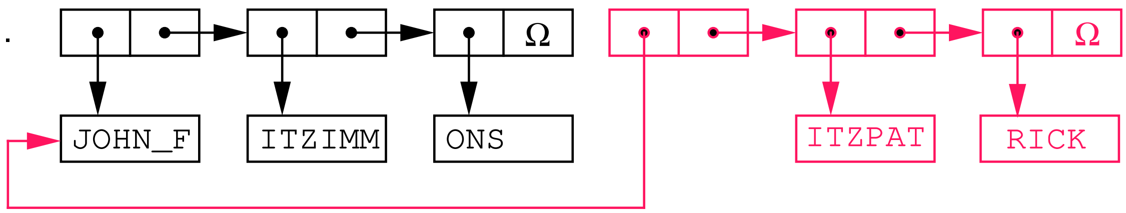


- 1 permits sharing arbitrary segments of strings (start, middle, end)

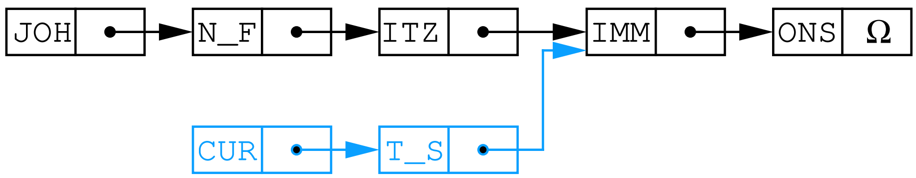
- 2 only permits sharing endings
  2 may occupy one less word than 1

CHARACTER DATA

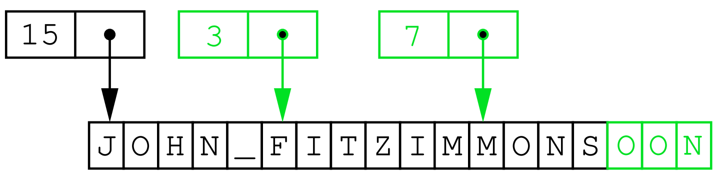
1.



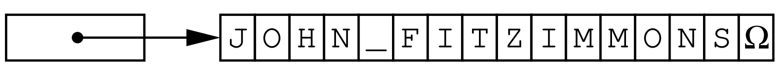
2.



3.

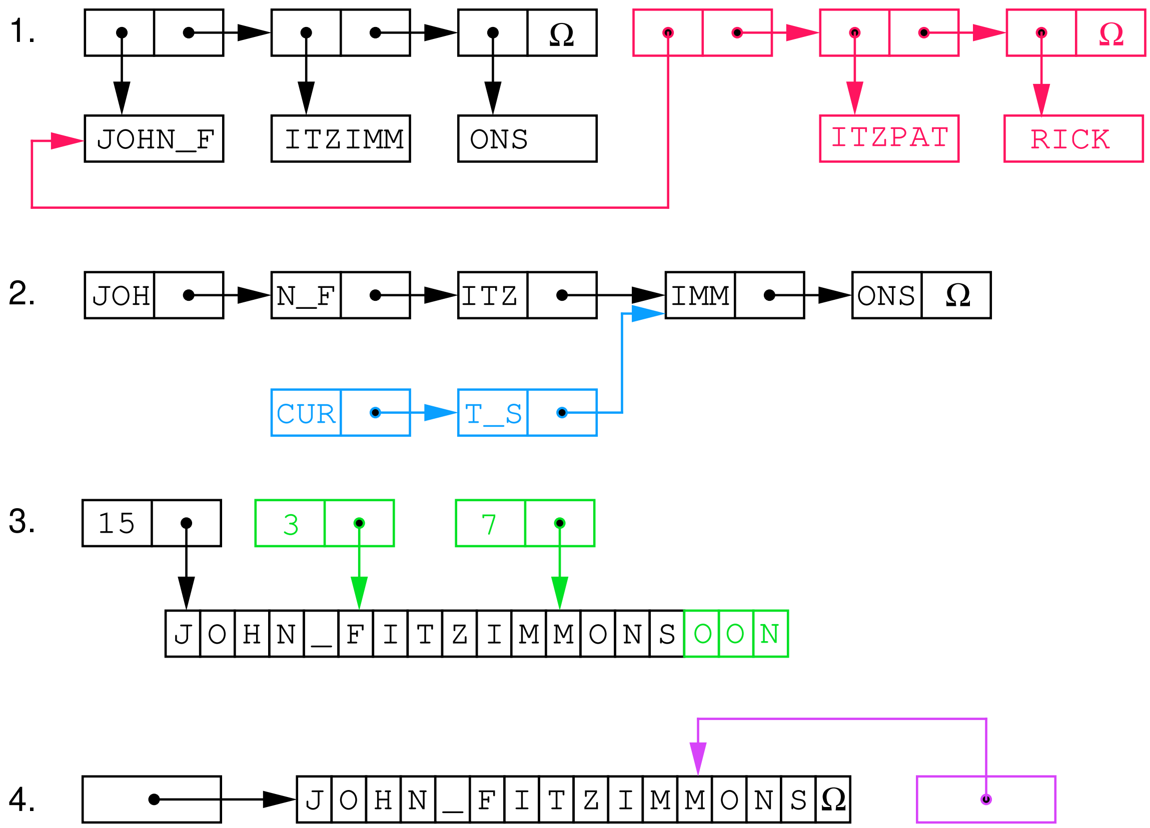


4.



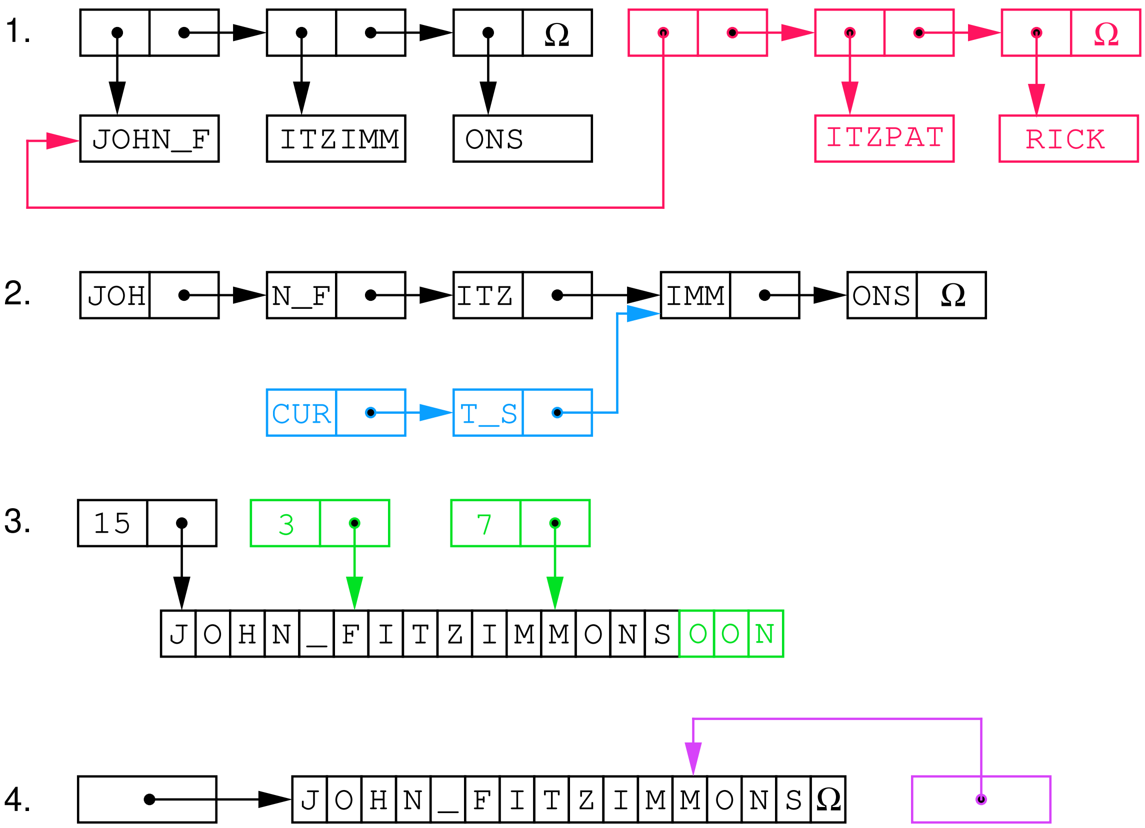
- 1 permits sharing arbitrary segments of strings (start, middle, end)
- 2 only permits sharing endings
  2 may occupy one less word than 1
- 3 only permits sharing when one string is a substring of another, or one string extends into the next string

CHARACTER DATA



- 1 permits sharing arbitrary segments of strings (start, middle, end)
- 2 only permits sharing endings
  2 may occupy one less word than 1
- 3 only permits sharing when one string is a substring of another, or one string extends into the next string
- 4 only permits sharing a terminating substring

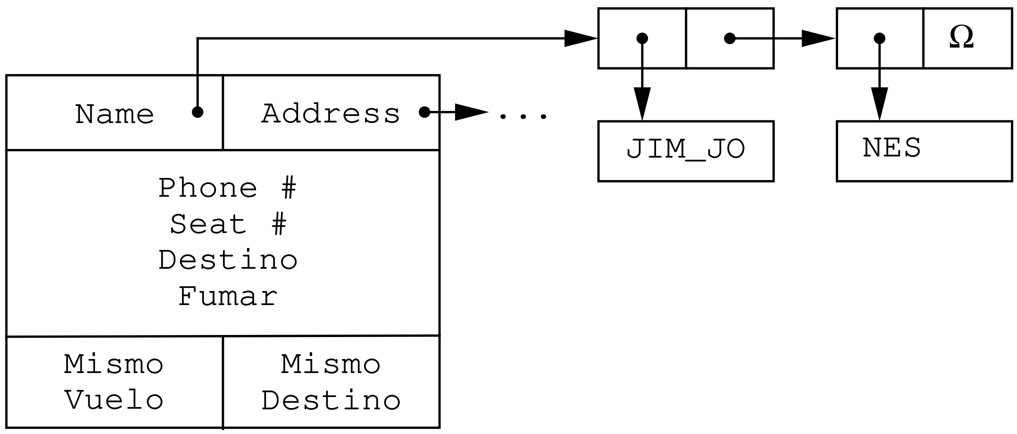
CHARACTER DATA



- 1 permits sharing arbitrary segments of strings (start, middle, end)
- 2 only permits sharing endings
  2 may occupy one less word than 1
- 3 only permits sharing when one string is a substring of another, or one string extends into the next string
- 4 only permits sharing a terminating substring
- 1 is superior to 2 because data and links are separate
- 3 is superior to 4

PASSENGER DATA STRUCTURE

```
JIM JONES
40 ELM ST.  ANYTOWN, ANYSTATE 01234
(123) 456-7890
45
DALLAS
NO SMOKING
```




```
Passenger = RECORD
    Name:     ^CharString;
    Addr:     ^CharString;
    Phone:    Integer;
    Seat:     Integer;
    Destino:  ^CharString;
    Fumar:    Boolean;
    MVuelo:   ^Passenger;
    MDestino: ^Passenger;
END;
```

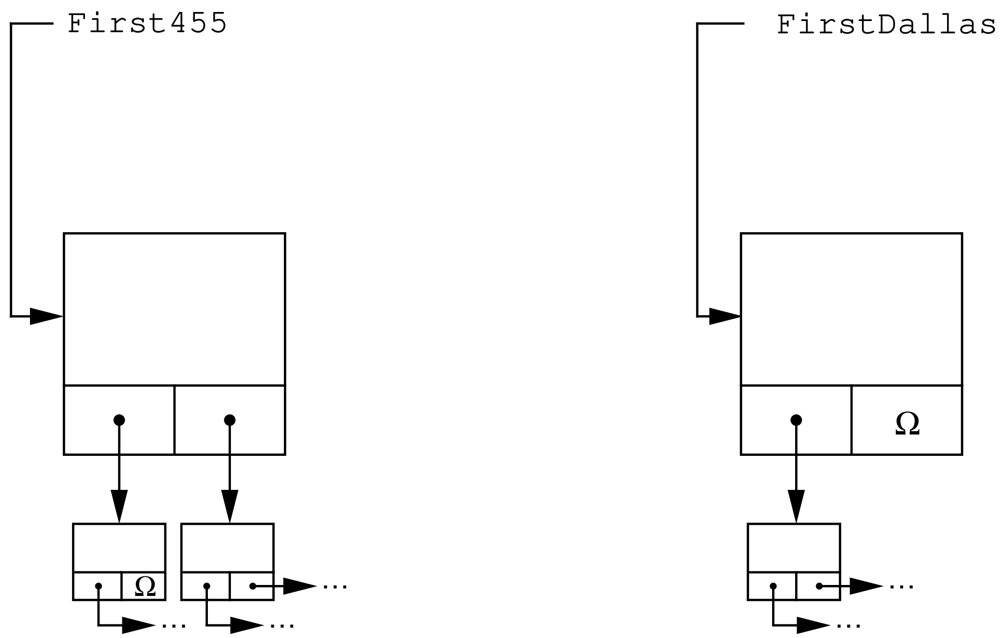
PROBLEM: Add a passenger to flight 455 who gets off at Dallas.

`First455` ≡ pointer to the first passenger on flight 455

`FirstDallas` ≡ pointer to the first passenger to Dallas

`NewPass` ≡ pointer to the new passenger.

	PASCAL
1. MVuelo(NewPass)←First455	NewPass↑.MVuelo←First455;
2. First455←NewPass;	First455←NewPass;
3. MDestino(NewPass)← FirstDallas;	NewPass↑.MDestino← FirstDallas;
4. FirstDallas←NewPass;	FirstDallas←NewPass;



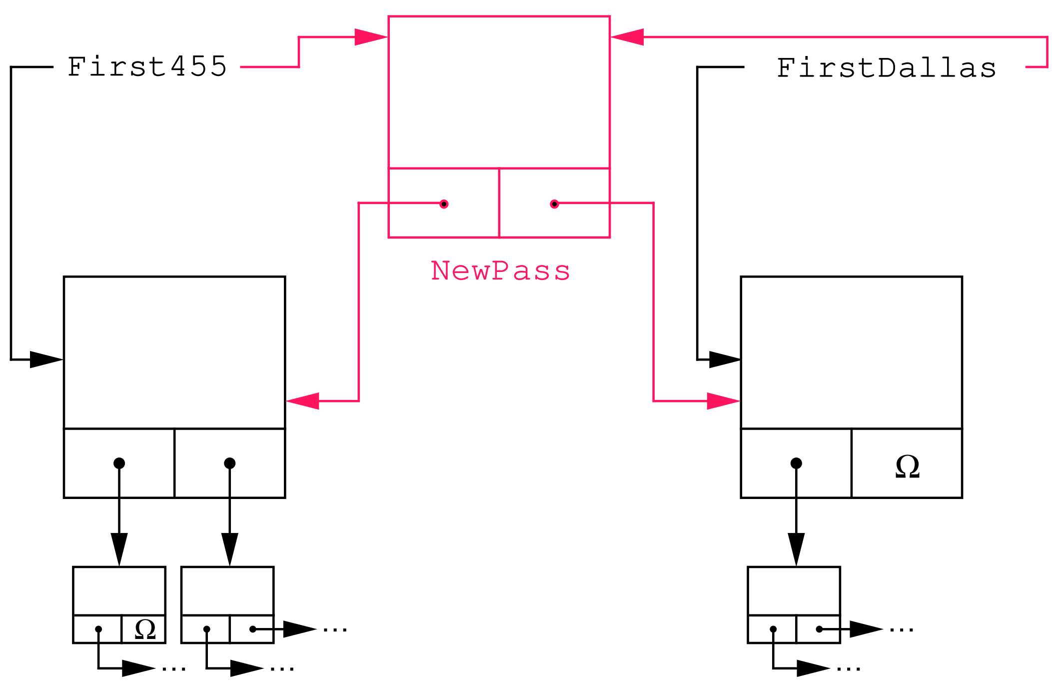
PROBLEM: Add a passenger to flight 455 who gets off at Dallas.

`First455` ≡ pointer to the first passenger on flight 455

`FirstDallas` ≡ pointer to the first passenger to Dallas

`NewPass` ≡ pointer to the new passenger.

		PASCAL
1.	MVuelo(NewPass)←First455	NewPass↑.MVuelo←First455;
2.	First455←NewPass;	First455←NewPass;
3.	MDestino(NewPass)← FirstDallas;	NewPass↑.MDestino← FirstDallas;
4.	FirstDallas←NewPass;	FirstDallas←NewPass;



PROBLEM: How many passengers get off at Dallas?

```
1.  n←0;
2.  x←FirstDallas;
3.  if x=Ω then HALT;
4.  n←n+1;
5.  x←MDestino(x);
6.  goto 3;
```

PASCAL:

```
n←0;
x←FirstDallas;
while x≠Ω do
    begin
        n←n+1;
        x←x↑.MDestino;
    end;
```

Field names:	`MVuelo, MDestino`
Variable names:	`n, x, First455, FirstDallas, NewPass`
Integer variable:	`n`
Link variables:	`x, First455, FirstDallas, NewPass` contain addresses!

DATA STRUCTURE SELECTION

1. Will the information be used?
   - playing cards – is the card face up or face down?

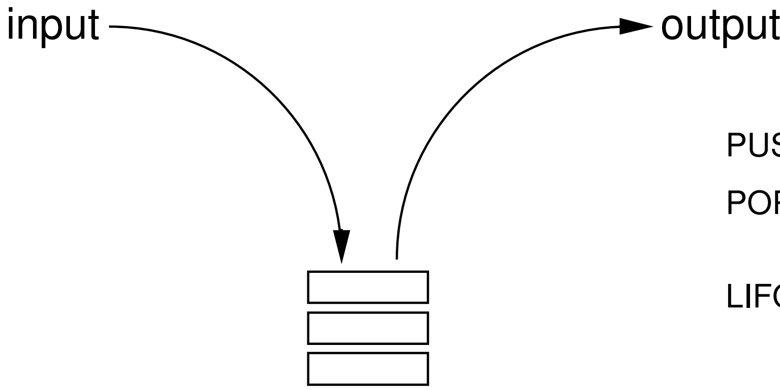
2. How accessible should the information be?
   - Ex: game of Hearts
     a. how many hearts in the hand
     b. explicit ⇒ must constantly update
     c. implicit ⇒ must look at all cards

- the choice of representation is dominated by the class of operations to be performed on the data

LINEAR LIST

- Set of nodes x[1], x[2], … x[n]   ($n \geq 1$)
- Principal property is that x[k] is followed by x[k+1]
- Possible Operations:
  1. gain access to the k^{th} node
  2. insert before the k^{th} node
  3. delete the k^{th} node
  4. combine 2 or more lists
  5. split a list into 2 or more lists
  6. make a copy of a list
  7. determine the number of nodes in a list
  8. sort the elements of the list
  9. search the list for a node with a particular value
- For operations 1, 2, and 3 k=1 or k=n are interesting
  1. stack: insert and delete at the same end
  2. queue: insert at one end
     delete at the other end
  3. deque: insert and delete at both ends

STACKS



PUSH ≡ insert
POP ≡ remove

LIFO

- Useful for processing goals and subgoals
- Subroutines and parameter transmittal
- Some computers have stack-like instructions

Ex: Translate arithmetic expression from infix to postfix

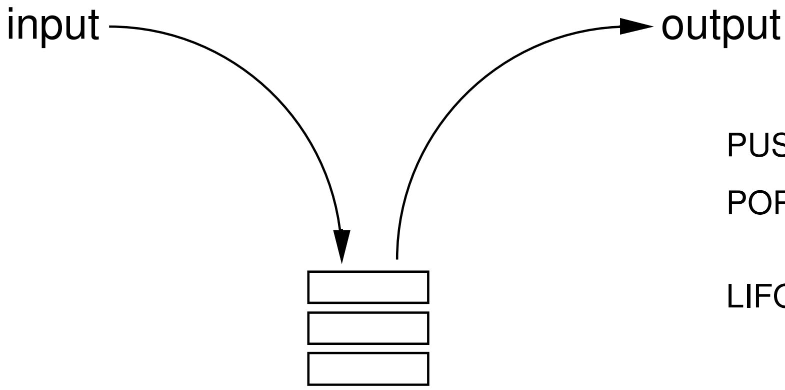
Infix:	operand	operator	operand	`A+B`
Prefix:	operator	operand	operand	`+AB`
Postfix:	operand	operand	operator	`AB+`

Postfix ≡ ‘Polish notation’

`A+B*C` ⇒ `ABC*+`

	Stack
Enter `A`	`C`
Enter `B`	`B`
Enter `C`	`A`

STACKS



PUSH ≡ insert
POP ≡ remove

LIFO

- Useful for processing goals and subgoals
- Subroutines and parameter transmittal
- Some computers have stack-like instructions

Ex: Translate arithmetic expression from infix to postfix

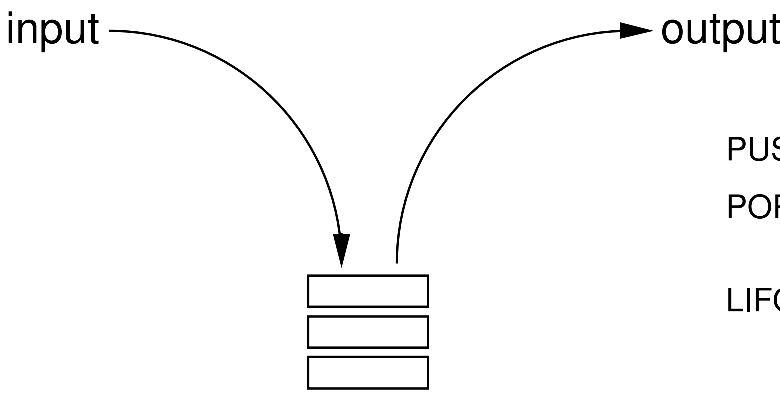
Infix:	operand	operator	operand	`A+B`
Prefix:	operator	operand	operand	`+AB`
Postfix:	operand	operand	operator	`AB+`

Postfix ≡ 'Polish notation'

`A+B*C` ⇒ `ABC*+`

	Stack	
Enter `A`	`C`	
Enter `B`	`B`	`B*C`
Enter `C`	`A`	`A`
`*`		

STACKS



PUSH ≡ insert

POP ≡ remove

LIFO

- Useful for processing goals and subgoals
- Subroutines and parameter transmittal
- Some computers have stack-like instructions

Ex: Translate arithmetic expression from infix to postfix

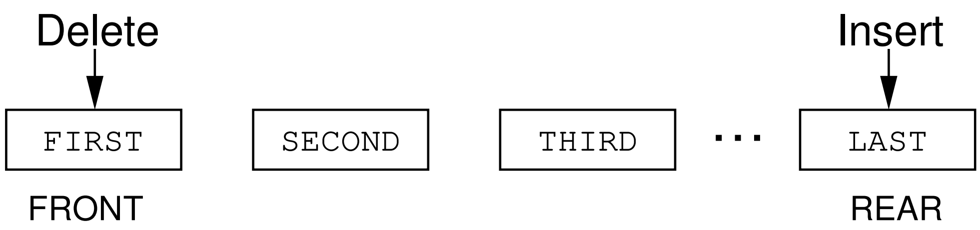
Infix:	operand	operator	operand	A+B
Prefix:	operator	operand	operand	+AB
Postfix:	operand	operand	operator	AB+

Postfix ≡ ‘Polish notation’

A+B*C ⇒ ABC*+

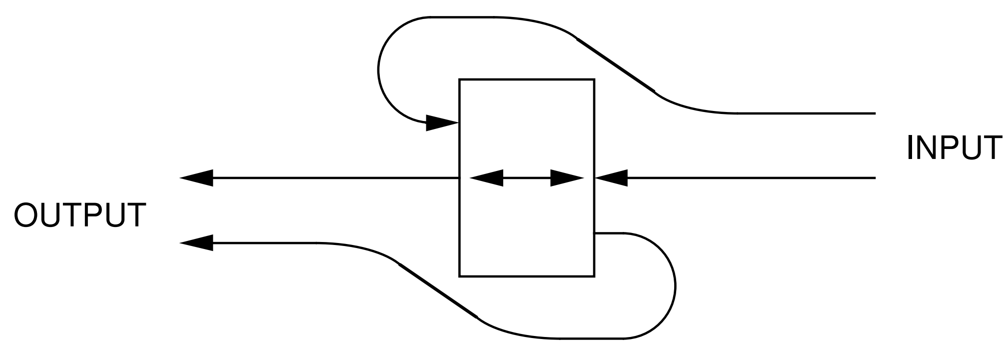
	Stack		
Enter A	C		
Enter B	B	B*C	
Enter C	A	A	A+B*C
*			
+			

QUEUE:

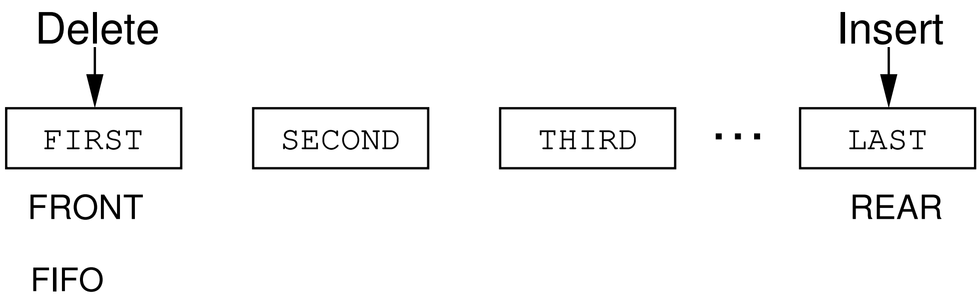


FIFO

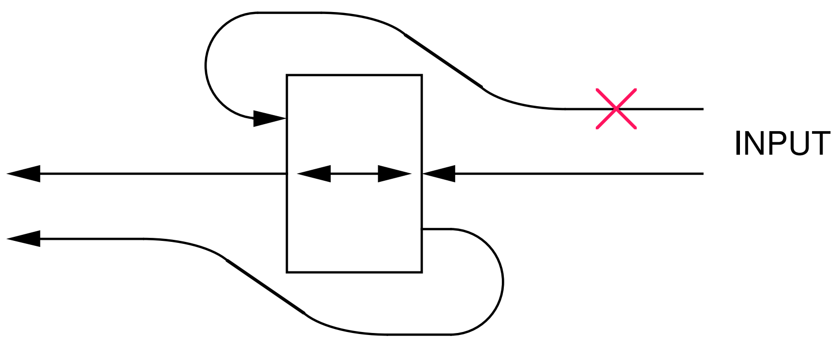
DEQUE:



QUEUE:

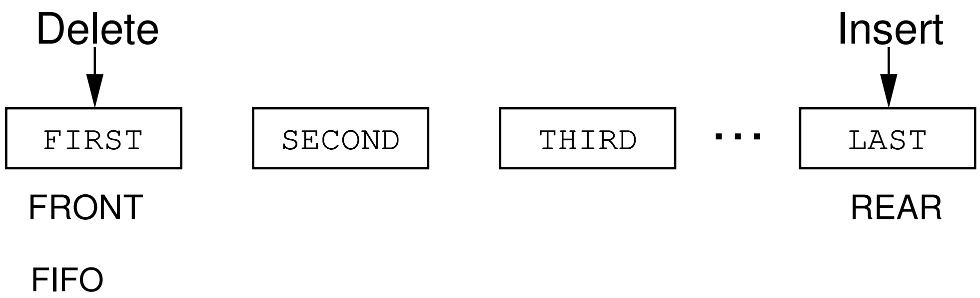


DEQUE:

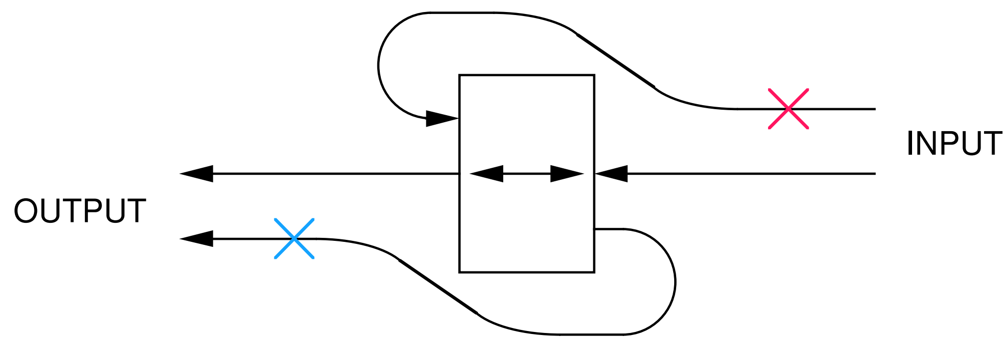


Input restricted deque

QUEUE:

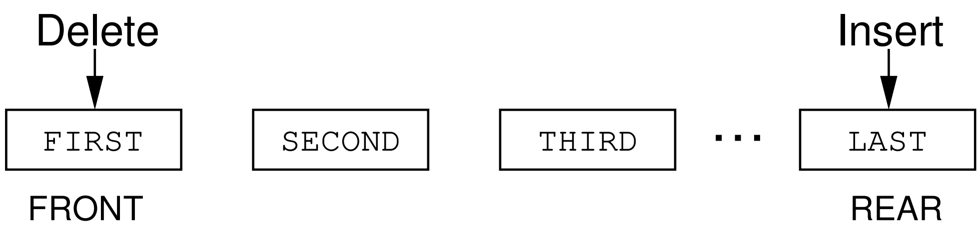


DEQUE:



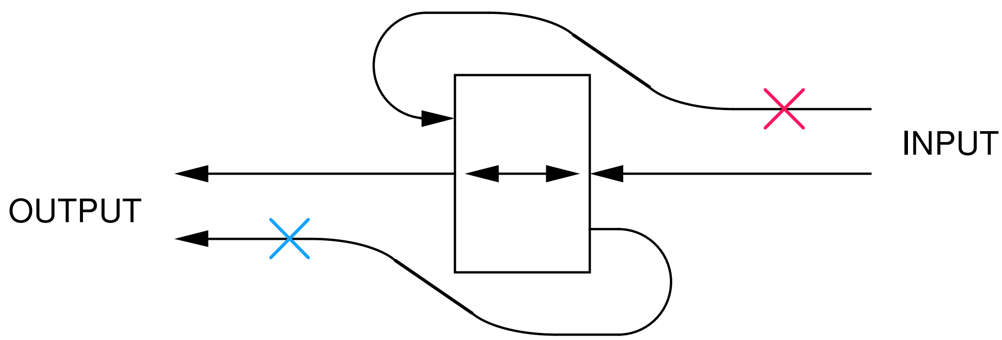
Input restricted deque
Output restricted deque

QUEUE:



FIFO

DEQUE:



Input restricted deque
Output restricted deque

Question: how would you construct a stack from a deque?

SEQUENTIAL ALLOCATION

- Easiest way to store a list in a computer is sequentially

  $\mathrm{LOC}(x[j+1]) = \mathrm{LOC}(x[j])+C$

  node size = C

  $\mathrm{LOC}(x[j]) = L_0+C\bullet j$ where $L_0 = \mathrm{LOC}(x[0])$

- STACK:

  1. sequential block of storage
  2. variable T($\equiv$ stack pointer) indicates the top of the stack
  3. T=0 $\Rightarrow$ stack is empty

- To enter a new value Y on the stack:

```
T←T+1;
x[T]←Y;
```

- To remove an entry from the stack we reverse entry sequence:

```
Y←x[T];
T←T-1;
```

QUEUE

- Two pointers:

  1. R to rear
  2. F to front
  3. R = F = 0 when the queue is empty

- Insertion at the rear of the queue:

```
R←R+1;
x[R]←Y;
```

- Removal of an entry from the front of the queue:

```
F←F+1;
Y←x[F];

if F=R then F←R←0;
```

- Note that the sequence of operations for removal is not the reverse of the sequence for insertion (i.e., we don't remove front and update pointer)

QUEUE

- Two pointers:

  1. R to rear
  2. F to front
  3. R = F = 0 when the queue is empty

- Insertion at the rear of the queue:

```
R←R+1;
x[R]←Y;
```

- Removal of an entry from the front of the queue:

```
F←F+1;
Y←x[F];

if F=R then F←R←0;
```

- Note that the sequence of operations for removal is not the reverse of the sequence for insertion (i.e., we don’t remove front and update pointer)

- Problem: suppose R is always > F ?

QUEUE

- Two pointers:

  1. R to rear
  2. F to front
  3. R = F = 0 when the queue is empty

- Insertion at the rear of the queue:

```
if R=M then R←1
else    R←R+1;
        x[R]←Y;
```

- Removal of an entry from the front of the queue:

```
if F=M then F←1
else    F←F+1;
        Y←x[F];
```

```
        if F=R then F←R←0;
```

- Note that the sequence of operations for removal is not the reverse of the sequence for insertion (i.e., we don't remove front and update pointer)

- Problem: suppose R is always > F ?

- Solution: make the queue implicitly circular
  x[1] x[2] … x[M] x[1]
  R = F = M when the queue is empty (initially)

QUEUE

- Two pointers:

  1. R to rear
  2. F to front
  3. R = F = 0 when the queue is empty

- Insertion at the rear of the queue:

```
if R=M then R←1
else   R←R+1;
       x[R]←Y;
```

- Removal of an entry from the front of the queue:

```
if F=M then F←1
else   F←F+1;
       Y←x[F];
```

```
       if F=R then F←R←0;
```

- Note that the sequence of operations for removal is not the reverse of the sequence for insertion (i.e., we don't remove front and update pointer)

- Problem: suppose R is always > F ?

- Solution: make the queue implicitly circular
  x[1] x[2] … x[M] x[1]
  R = F = M when the queue is empty (initially)

- Question: Why not a problem in a bank line?

QUEUE

- Two pointers:

  1. R to rear
  2. F to front
  3. R = F = 0 when the queue is empty

- Insertion at the rear of the queue:

```
if R=M then R←1
else   R←R+1;
       x[R]←Y;
```

- Removal of an entry from the front of the queue:

```
if F=M then F←1
else   F←F+1;
       Y←x[F];
```

```
       if F=R then F←R←0;
```

- Note that the sequence of operations for removal is not the reverse of the sequence for insertion (i.e., we don’t remove front and update pointer)

- Problem: suppose R is always > F ?

- Solution: make the queue implicitly circular
  x[1] x[2] … x[M] x[1]
  R = F = M when the queue is empty (initially)

- Question: Why not a problem in a bank line?

- Answer: Because the people move from position to position in the line

OVERFLOW

- Suppose we run out of memory?
- Assume only M locations are available

1. Stack insertion

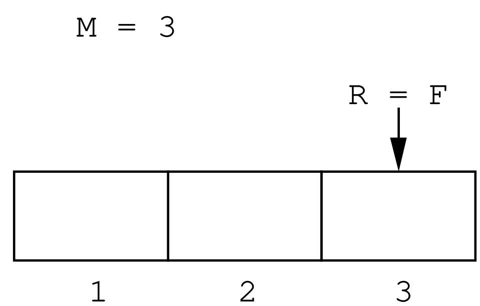
```
T←T+1;
if T>M then OVERFLOW;
x[T]←Y;
```

2. Stack deletion:

```
if T=0 then UNDERFLOW;
Y←x[T];
T←T-1;
```

3. Queue insertion:

```
if R=M then R←1;
else R←R+1;
if R=F then OVERFLOW
else x[R]←Y;
```



4. Queue deletion:

```
if R=F then UNDERFLOW
else
  begin
    if F=M then F←1
    else F←F+1;
    Y←x[F];
  end;
```

- We start with F = R = M
- UNDERFLOW is not a real problem

OVERFLOW

- Suppose we run out of memory?
- Assume only M locations are available

1. Stack insertion

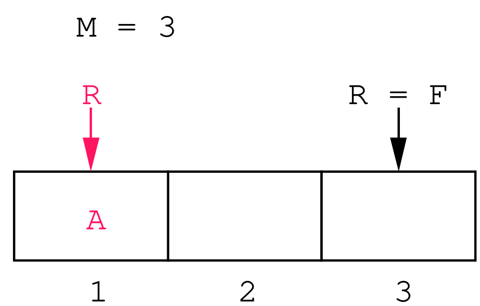
```
T←T+1;
if T>M then OVERFLOW;
x[T]←Y;
```

2. Stack deletion:

```
if T=0 then UNDERFLOW;
Y←x[T];
T←T-1;
```

3. Queue insertion:

```
if R=M then R←1;
else R←R+1;
if R=F then OVERFLOW
else x[R]←Y;
```



Insert A

4. Queue deletion:

```
if R=F then UNDERFLOW
else
  begin
    if F=M then F←1
    else F←F+1;
    Y←x[F];
  end;
```

- We start with F = R = M
- UNDERFLOW is not a real problem

OVERFLOW

- Suppose we run out of memory?
- Assume only M locations are available

1. Stack insertion

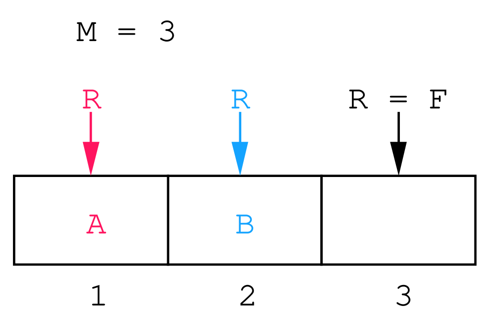
```
T←T+1;
if T>M then OVERFLOW;
x[T]←Y;
```

2. Stack deletion:

```
if T=0 then UNDERFLOW;
Y←x[T];
T←T-1;
```

3. Queue insertion:

```
if R=M then R←1;
else R←R+1;
if R=F then OVERFLOW
else x[R]←Y;
```



Insert A
Insert B

4. Queue deletion:

```
if R=F then UNDERFLOW
else
  begin
    if F=M then F←1
    else F←F+1;
    Y←x[F];
  end;
```

- We start with F = R = M
- UNDERFLOW is not a real problem

OVERFLOW

- Suppose we run out of memory?
- Assume only M locations are available

1. Stack insertion

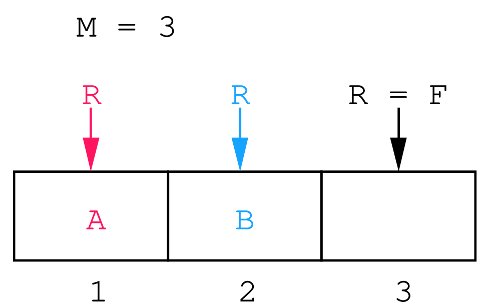
```
T←T+1;
if T>M then OVERFLOW;
x[T]←Y;
```

2. Stack deletion:

```
if T=0 then UNDERFLOW;
Y←x[T];
T←T-1;
```

3. Queue insertion:

```
if R=M then R←1;
else R←R+1;
if R=F then OVERFLOW
else x[R]←Y;
```



Insert A
Insert B
Insert C ⟹ OVERFLOW!

4. Queue deletion:

```
if R=F then UNDERFLOW
else
  begin
    if F=M then F←1
    else F←F+1;
    Y←x[F];
  end;
```

- We start with F = R = M
- UNDERFLOW is not a real problem

MULTIPLE STACKS

- Two stacks can grow towards each other

stack1 → ← stack2

- More than 2 stacks requires variable locations for base of stack

`BASE[i]` ≡ starting address of stack i
`TOP[i]` ≡ top of stack i

Insertion into stack i:

```
TOP[i]←TOP[i]+1;
if TOP[i]>BASE[i+1] then OVERFLOW;
else CONTENTS(TOP[i])← Y
```

Deletion from stack i:

```
if TOP[i]=BASE[i] then UNDERFLOW;
Y←CONTENTS(TOP[i]);
TOP[i]←TOP[i]−1;
```

When stack i overflows:

```
1.  find smallest k ∋ i<k≤n and TOP[k]<BASE[k+1]
    for TOP[k] ≥ m > BASE[i+1]
        CONTENTS(m+1) ← CONTENTS(m)
    for i < j ≤ k
        BASE[j]←BASE[j]+1; TOP[j]←TOP[j]+1;

2.  find largest k ∋ 1≤k<i and TOP[k]<BASE[k+1]
    for BASE[k+1] < m < TOP[i]
        CONTENTS(m−1)←CONTENTS(m)
    for k < j ≤ i
        BASE[j]←BASE[j]−1; TOP[j]←TOP[j]−1;

3.  if TOP[k]=BASE[k+1] ∀ k≠i then REAL OVERFLOW
```

LINKED ALLOCATION

- Next node need not be physically adjacent
- Use an extra field to indicate address of next node

Sequential

Item 1
Item 2
Item 3
⋮
Item n

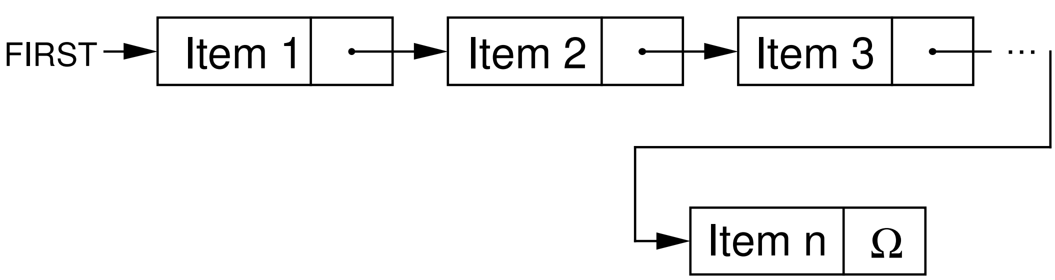
Linked

	Item 1	B
B	Item 2	C
C	Item 3	D
	⋮	
	Item n	Ω

- Each node has two fields

Info	Link

- Need a pointer to FIRST element



Ω denotes the end of the list

COMPARISON OF LINKED(L) VS SEQUENTIAL(S)

1. L requires extra space for links
   - but if a node has many fields, then overhead is small
   - can share storage with L
   - repacking is inefficient with S when memory is densely packed

2. Easy to insert and delete with L
   - no need to move data as with S

3. S is superior for random access into a list (i.e., Kth element)
   - S: add an offset (K) to base address
   - L: traverse K links

4. L facilitates joining and breaking lists

5. L allows more complex data structures

6. S is superior for marching sequentially through a list
   - S makes use of indexing
   - L makes use of indirect addressing ($\Rightarrow$ memory access)

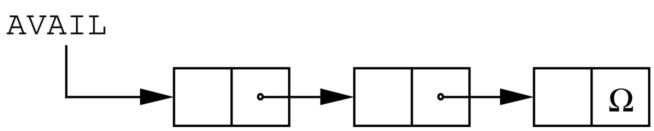
7. S takes advantage of locality

STORAGE MANAGEMENT

- Linked list of available storage
- `AVAIL` points to the first element
- Use `LINK` field

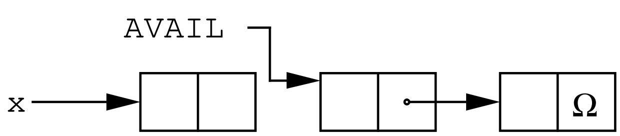
`x⇐AVAIL` is short hand notation for allocating a new node as follows:

```
if AVAIL=Ω then OVERFLOW
else
  begin
    x←AVAIL;
    AVAIL←LINK(AVAIL);
    LINK(x)←Ω;
  end;
```



`AVAIL⇐x` is short hand notation for returning a node as follows:

```
LINK(x)←AVAIL;
AVAIL←x;
```

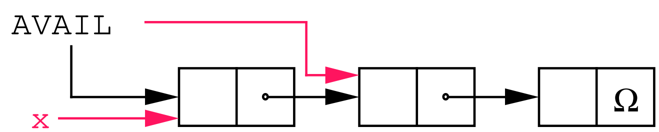


STORAGE MANAGEMENT

- Linked list of available storage
- `AVAIL` points to the first element
- Use `LINK` field

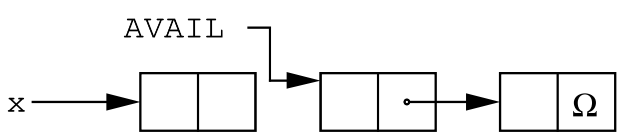
`x⇐AVAIL` is short hand notation for allocating a new node as follows:

```
if AVAIL=Ω then OVERFLOW
else
  begin
    x←AVAIL;
    AVAIL←LINK(AVAIL);
    LINK(x)←Ω;
  end;
```



`AVAIL⇐x` is short hand notation for returning a node as follows:

```
LINK(x)←AVAIL;
AVAIL←x;
```

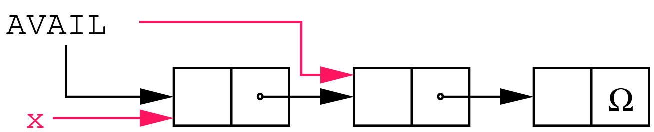


STORAGE MANAGEMENT

- Linked list of available storage
- `AVAIL` points to the first element
- Use `LINK` field

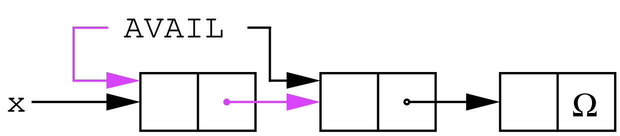
`x⇐AVAIL` is short hand notation for allocating a new node as follows:

```
if AVAIL=Ω then OVERFLOW
else
  begin
    x←AVAIL;
    AVAIL←LINK(AVAIL);
    LINK(x)←Ω;
  end;
```

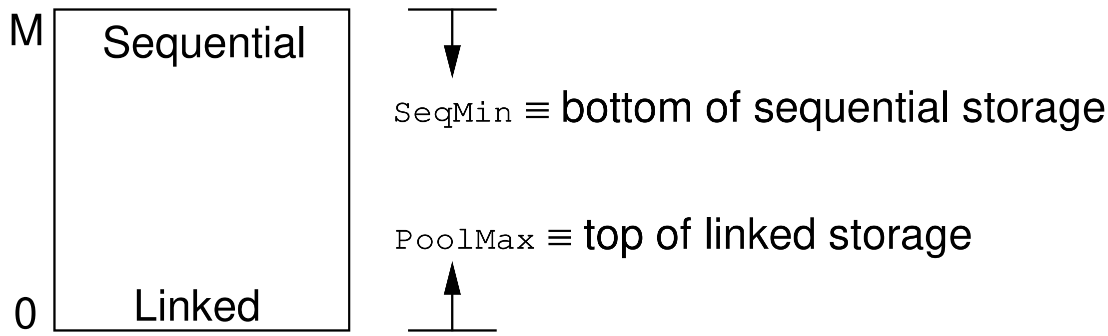


`AVAIL⇐x` is short hand notation for returning a node as follows:

```
LINK(x)←AVAIL;
AVAIL←x;
```



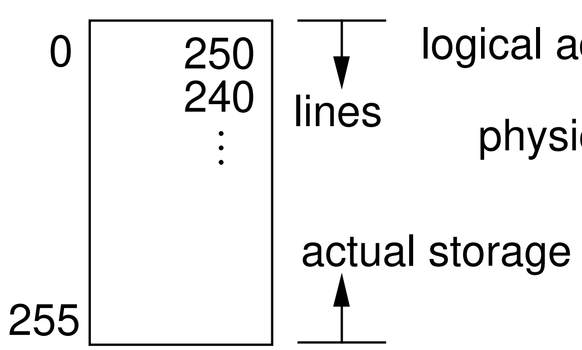
COMBINING SEQUENTIAL AND LINKED STORAGE



Allocation of a node of linked storage (x):

```
if AVAIL=Ω then
   if PoolMax>SeqMin then OVERFLOW
   else
     begin
      PoolMax←PoolMax+1;
      x⇐PoolMax;
     end;
else x⇐AVAIL;
```

- No need to initially link up AVAIL
- A similar scheme is used in DBMS-10 for storing records on disk pages



logical address = la = | page # | line # |

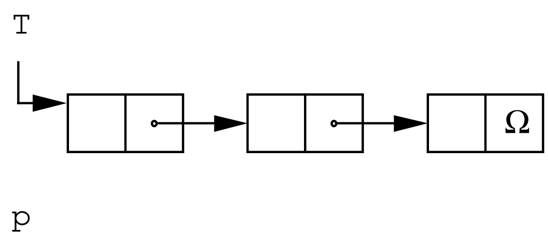
physical address = | page #(la) | 0 | + CONTENTS[line #(la)]

LINKED STACKS

Insert Y into a linked stack:

T = top of stack pointer

```
p⇐AVAIL;
INFO(p)←Y;
LINK(p)←T;
T←p;
```



Delete Y from a linked stack:

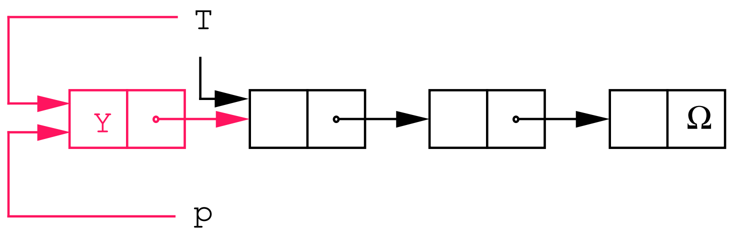
```
if T=Ω then UNDERFLOW;
p←T;
T←LINK(p);
Y←INFO(p);
AVAIL⇐p;
```

LINKED STACKS

Insert Y into a linked stack:

T = top of stack pointer

```
p⇐AVAIL;
INFO(p)←Y;
LINK(p)←T;
T←p;
```



Delete Y from a linked stack:

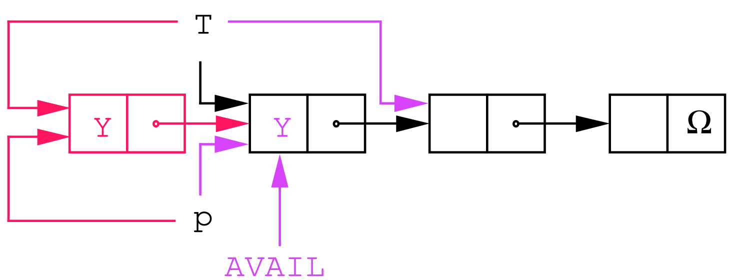
```
if T=Ω then UNDERFLOW;
p←T;
T←LINK(p);
Y←INFO(p);
AVAIL⇐p;
```

LINKED STACKS

Insert Y into a linked stack:

T = top of stack pointer

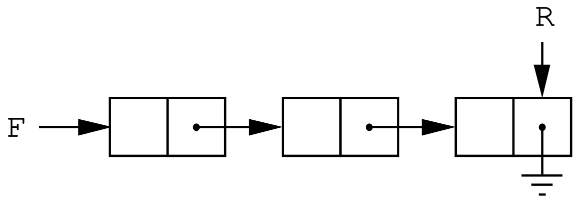
```
p⇐AVAIL;
INFO(p)←Y;
LINK(p)←T;
T←p;
```



Delete Y from a linked stack:

```
if T=Ω then UNDERFLOW;
p←T;
T←LINK(p);
Y←INFO(p);
AVAIL⇐p;
```

LINKED QUEUES



F=Ω signifies an empty queue

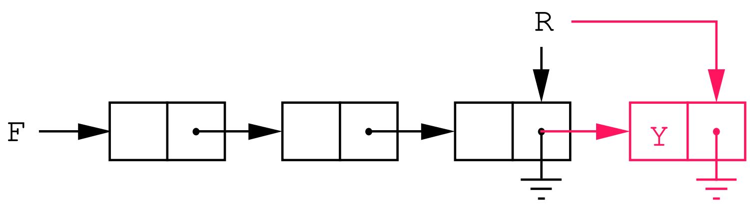
Insert Y at the rear of a queue:

```
P⇐AVAIL;
INFO(P)←Y;
LINK(P)←Ω;
if F=Ω then F←P;
else LINK(R)←P;
R←P;
```

Delete Y from the front of a queue:

```
if F=Ω then UNDERFLOW;
P←F;
F←LINK(P);
Y←INFO(P);
AVAIL⇐P;
```

LINKED QUEUES



F=Ω signifies an empty queue

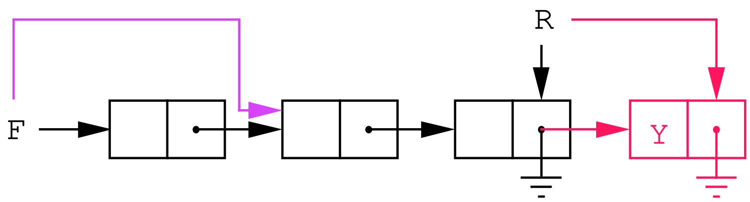
Insert Y at the rear of a queue:

```
P⇐AVAIL;
INFO(P)←Y;
LINK(P)←Ω;
if F=Ω then F←P;
else LINK(R)←P;
R←P;
```

Delete Y from the front of a queue:

```
if F=Ω then UNDERFLOW;
P←F;
F←LINK(P);
Y←INFO(P);
AVAIL⇐P;
```

LINKED QUEUES



$F=\Omega$ signifies an empty queue

Insert Y at the rear of a queue:

```
P⇐AVAIL;
INFO(P)←Y;
LINK(P)←Ω;
if F=Ω then F←P;
else LINK(R)←P;
R←P;
```

Delete Y from the front of a queue:

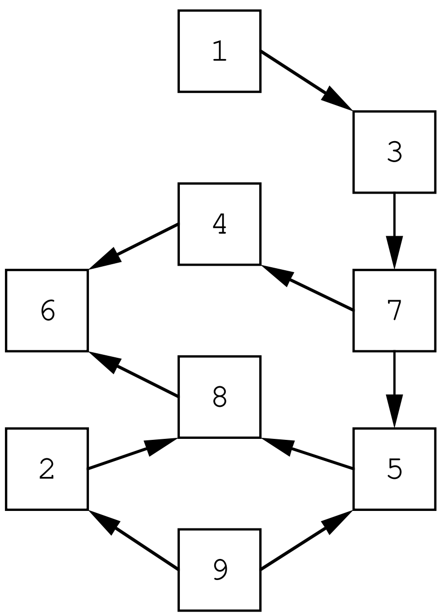
```
if F=Ω then UNDERFLOW;
P←F;
F←LINK(P);
Y←INFO(P);
AVAIL⇐P;
```

TOPOLOGICAL SORT

- Given: relations as to what precedes what ($a<b$)
- Desired: a partial ordering

- Formal definition of a partial ordering
  1. If $X<Y$ and $Y<Z$ then $X<Z$ (transitivity)
  2. If $X<Y$ then $Y \not< X$ (asymmetry)
  3. $X \not< X$ (irreflexivity)

  2 implies the absence of loops



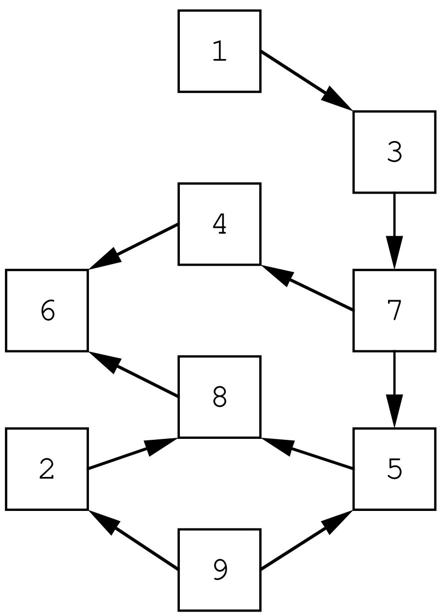
- Applications:
  1. job scheduling — PERT networks, CPM
  2. system tapes
  3. subroutine order so no routine is invoked before it is declared
     - But see PASCAL FORWARD declarations

TOPOLOGICAL SORT

- Given: relations as to what precedes what ($a<b$)
- Desired: a partial ordering

- Formal definition of a partial ordering
  1. If $X<Y$ and $Y<Z$ then $X<Z$ (transitivity)
  2. If $X<Y$ then $Y \not< X$ (asymmetry)
  3. $X \not< X$ (irreflexivity)

  2 implies the absence of loops



1 3 7 4 9 2 5 8 6

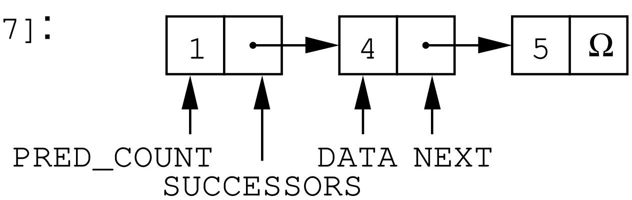
- Applications:
  1. job scheduling — PERT networks, CPM
  2. system tapes
  3. subroutine order so no routine is invoked before it is declared
     - But see PASCAL FORWARD declarations

ALGORITHM

- Performs topological sort
- Proves by construction the existence of the ordering
- Recursive algorithm
  1. find an item, *i*, not preceded by any other item
  2. remove *i* and perform the sort on the remaining items
- Brute force solution takes $O(n \cdot m)$ time for n items and m successor-predecessor relation pairs by executing the following for each of the n items
  1. make a pass over successor-predecessor list S and find items that do not appear as a successor (m operations)
  2. remove all relations from S where an item found in 1 appears as a predecessor (m operations)
- Data Structure for better solution:

  `t[K]` corresponds to item K with 2 fields:
  - `PRED_COUNT[t[K]]` $\equiv$ # of direct predecessors of `K` (i. e., `L` < `K`)
  - `SUCCESSORS[t[K]]` $\equiv$ pointer to a linked list containing the direct successors of item `K`

Ex: `t[7]`:



- Maintain a queue of all items having 0 predecessors
- Each time item `K` is output:
  1. remove `t[K]` from the queue
  2. decrement `PRED_COUNT` field of all successors of `K`
  3. add to the queue any node whose `PRED_COUNT` field has gone to 0
- $O(m+n)$ time and space

OBSERVATIONS

- Can use a stack instead of a queue

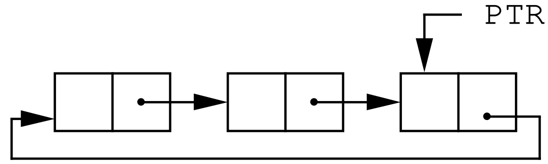
- The queue can be kept in the `PRED_COUNT` field of `t[K]` since once this field has gone to zero it will not be referenced again – i.e., it can no longer be decremented

- Sequential allocation for `t[K]` whose size is fixed
- Linked allocation for the successor relations

- Queue is linked by index (à la FORTRAN)
- Successor list is linked by address

CIRCULAR LISTS

- Last node points back to first node
- No need to think of any node as a 'last' or 'first' node



1. Insert Y at the left:

```
P⇐AVAIL;  INFO(P)←Y;
if PTR=Ω then PTR←LINK(P)←P
else
 begin
    LINK(P)←LINK(PTR);  LINK(PTR)←P;
 end;
```

2. Insert Y at the right:
   Insert Y at the left;

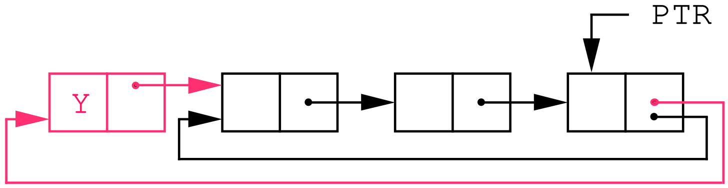
```
PTR←P;
```

3. Set Y to the left node and delete:

```
if PTR=Ω  then UNDERFLOW;
P←LINK(PTR);  Y←INFO(P);
LINK(PTR)←LINK(P);  AVAIL⇐P;
if PTR=P then PTR←Ω;
/* Check for a list of one element */
/* before deleting                 */
```

CIRCULAR LISTS

- Last node points back to first node
- No need to think of any node as a ‘last’ or ‘first’ node



1. Insert Y at the left:

```
P⇐AVAIL;   INFO(P)←Y;
if PTR=Ω then PTR←LINK(P)←P
else
 begin
    LINK(P)←LINK(PTR);   LINK(PTR)←P;
 end;
```

2. Insert Y at the right:
   Insert Y at the left;

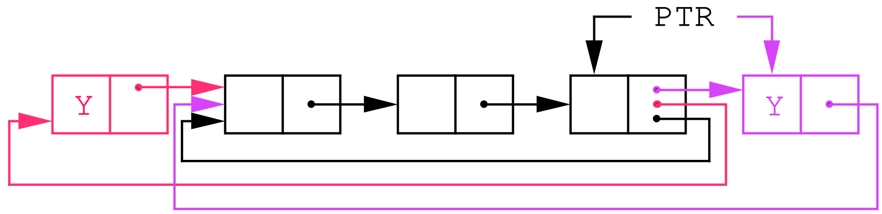
```
PTR←P;
```

3. Set Y to the left node and delete:

```
if PTR=Ω  then UNDERFLOW;
P←LINK(PTR);   Y←INFO(P);
LINK(PTR)←LINK(P);   AVAIL⇐P;
if PTR=P then PTR←Ω;
/* Check for a list of one element */
/* before deleting                 */
```

CIRCULAR LISTS

- Last node points back to first node
- No need to think of any node as a ‘last’ or ‘first’ node



1. Insert Y at the left:

```
P⇐AVAIL;  INFO(P)←Y;
if PTR=Ω then PTR←LINK(P)←P
else
 begin
    LINK(P)←LINK(PTR);  LINK(PTR)←P;
 end;
```

2. Insert Y at the right:

   Insert Y at the left;

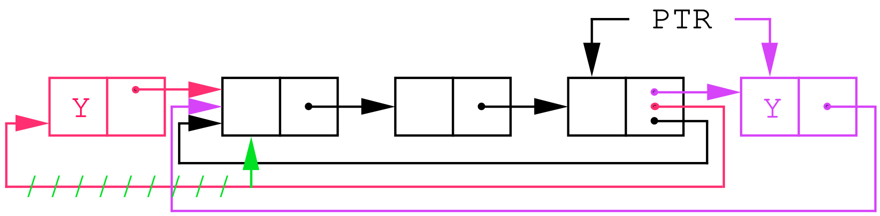
```
PTR←P;
```

3. Set Y to the left node and delete:

```
if PTR=Ω  then UNDERFLOW;
P←LINK(PTR);  Y←INFO(P);
LINK(PTR)←LINK(P);  AVAIL⇐P;
if PTR=P then PTR←Ω;
/* Check for a list of one element */
/* before deleting                 */
```

CIRCULAR LISTS

- Last node points back to first node
- No need to think of any node as a ‘last’ or ‘first’ node



1. Insert Y at the left:

```
P⇐AVAIL;  INFO(P)←Y;
if PTR=Ω then PTR←LINK(P)←P
else
 begin
    LINK(P)←LINK(PTR);  LINK(PTR)←P;
 end;
```

2. Insert Y at the right:
   Insert Y at the left;

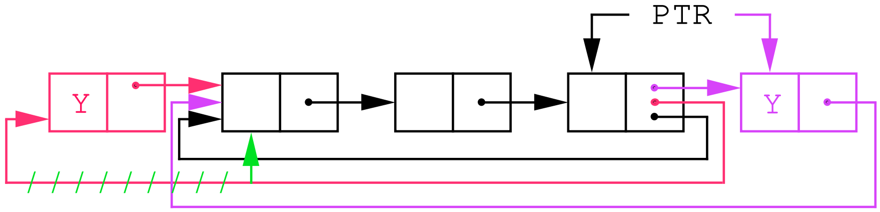
```
PTR←P;
```

3. Set Y to the left node and delete:

```
if PTR=Ω  then UNDERFLOW;
P←LINK(PTR);  Y←INFO(P);
LINK(PTR)←LINK(P);  AVAIL⇐P;
if PTR=P then PTR←Ω;
/* Check for a list of one element */
/* before deleting                 */
```

CIRCULAR LISTS

- Last node points back to first node
- No need to think of any node as a ‘last’ or ‘first’ node



1. Insert Y at the left:

```
P⇐AVAIL;  INFO(P)←Y;
if PTR=Ω then PTR←LINK(P)←P
else
 begin
    LINK(P)←LINK(PTR);  LINK(PTR)←P;
 end;
```

2. Insert Y at the right:
   Insert Y at the left;

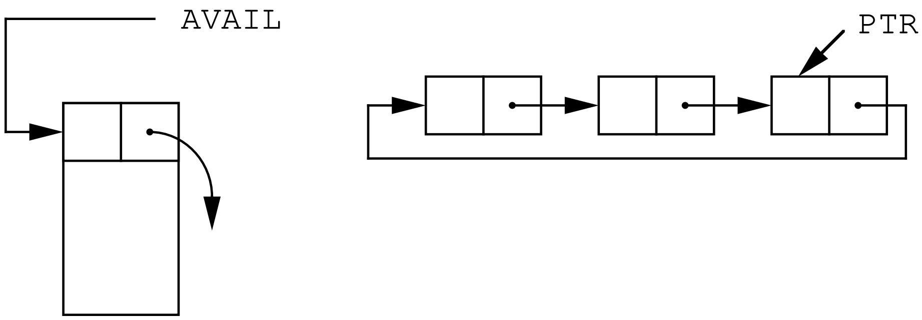
```
PTR←P;
```

3. Set Y to the left node and delete:

```
if PTR=Ω  then UNDERFLOW;
P←LINK(PTR);  Y←INFO(P);
LINK(PTR)←LINK(P);  AVAIL⇐P;
if PTR=P then PTR←Ω;
/* Check for a list of one element */
/* before deleting                 */
```

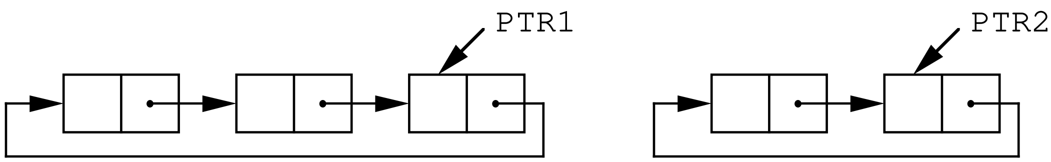
1 and 3 imply stack
2 and 3 imply queue
1, 2, and 3 imply output restricted deque

ERASING A CIRCULAR LIST



Note: PTR is meaningless after erasing a list

Inserting Circular List L2 at the Right of Circular List L1:

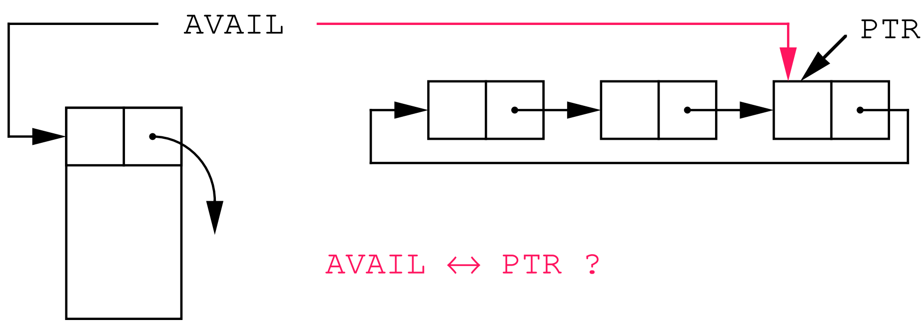


Assume PTR1 points to L1 and PTR2 points to L2.

```
if PTR2≠Ω then
  begin
    if PTR1≠Ω then LINK(PTR1)↔LINK(PTR2);
    PTR1←PTR2;
    PTR2←Ω;
  end
```

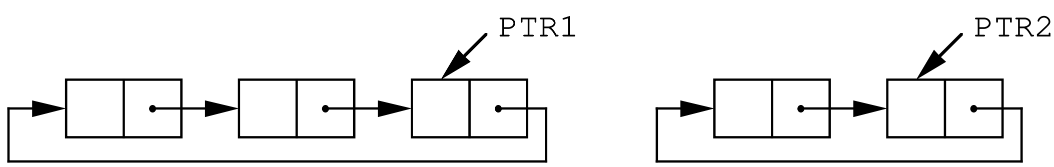
- A circular list can also be split into two lists
- Analogous to concatenation and deconcatenation of strings.

ERASING A CIRCULAR LIST



Note: PTR is meaningless after erasing a list

Inserting Circular List L2 at the Right of Circular List L1:

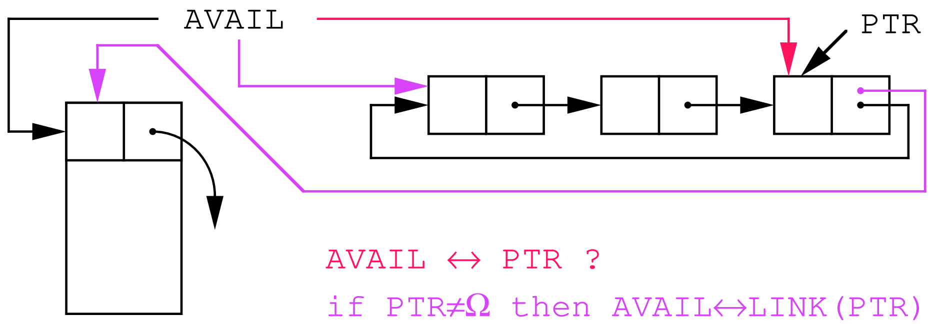


Assume PTR1 points to L1 and PTR2 points to L2.

```
if PTR2≠Ω then
  begin
    if PTR1≠Ω then LINK(PTR1)↔LINK(PTR2);
    PTR1←PTR2;
    PTR2←Ω;
  end
```

- A circular list can also be split into two lists
- Analogous to concatenation and deconcatenation of strings.

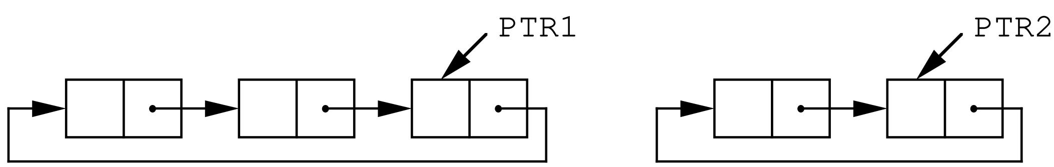
ERASING A CIRCULAR LIST




```
AVAIL ↔ PTR ?
if PTR≠Ω then AVAIL↔LINK(PTR)
```

Note: `PTR` is meaningless after erasing a list

Inserting Circular List L2 at the Right of Circular List L1:

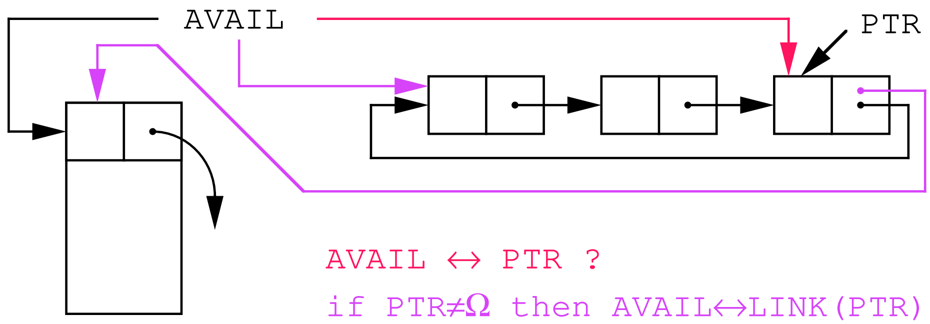


Assume `PTR1` points to L1 and `PTR2` points to L2.

```
if PTR2≠Ω then
  begin
    if PTR1≠Ω then LINK(PTR1)↔LINK(PTR2);
    PTR1←PTR2;
    PTR2←Ω;
  end
```

- A circular list can also be split into two lists
- Analogous to concatenation and deconcatenation of strings.

ERASING A CIRCULAR LIST

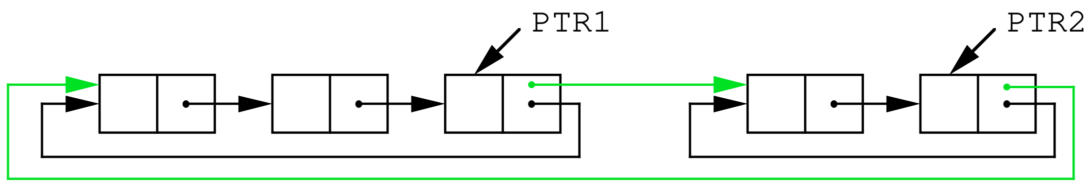


AVAIL ↔ PTR ?

if PTR≠Ω then AVAIL↔LINK(PTR)

Note: PTR is meaningless after erasing a list

Inserting Circular List L2 at the Right of Circular List L1:

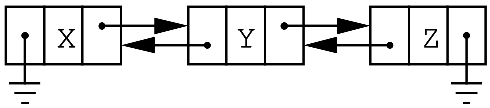


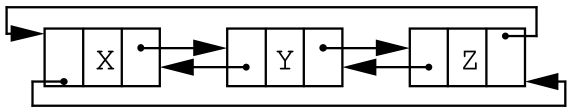
Assume PTR1 points to L1 and PTR2 points to L2.

```
if PTR2≠Ω then
  begin
    if PTR1≠Ω then LINK(PTR1)↔LINK(PTR2);
    PTR1←PTR2;
    PTR2←Ω;
  end
```

- A circular list can also be split into two lists
- Analogous to concatenation and deconcatenation of strings.

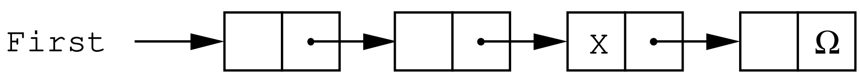
DOUBLY-LINKED LISTS





`RLINK(LLINK(Y)) = LLINK(RLINK(Y)) = Y`

- Disadvantage: More space for links
- Advantage: Given X, it can be deleted without having to locate its predecessor as is necessary with singly-linked lists



Easy to insert a node to the left or right of another node:

Insert to the right of `Z`:

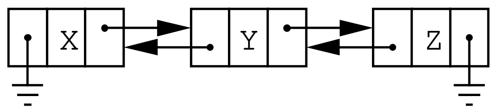
```
P⇐AVAIL;
LLINK(P)←Z; RLINK(P)←RLINK(Z);
LLINK(RLINK(Z))←P; RLINK(Z)←P;
```

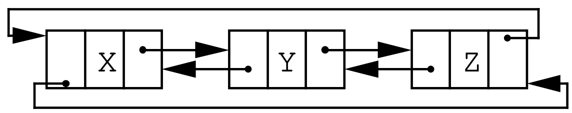
Insert to the left of `X`:

Interchange LEFT and RIGHT in ‘Insertion to the right’.

- 4 links are changed (only 2 changed with singly-linked list)

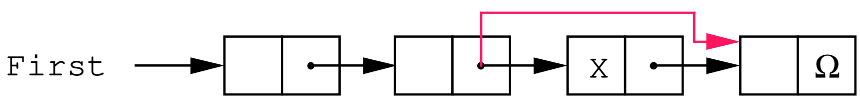
DOUBLY-LINKED LISTS





RLINK(LLINK(Y)) = LLINK(RLINK(Y)) = Y

- Disadvantage: More space for links
- Advantage: Given X, it can be deleted without having to locate its predecessor as is necessary with singly-linked lists



Easy to insert a node to the left or right of another node:

Insert to the right of Z:

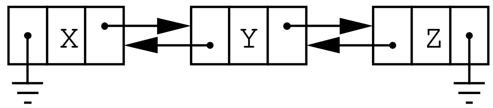
```
P⇐AVAIL;
LLINK(P)←Z; RLINK(P)←RLINK(Z);
LLINK(RLINK(Z))←P; RLINK(Z)←P;
```

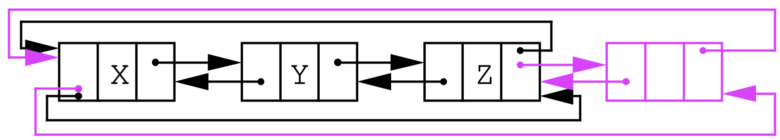
Insert to the left of X:

Interchange LEFT and RIGHT in ‘Insertion to the right’.

- 4 links are changed (only 2 changed with singly-linked list)

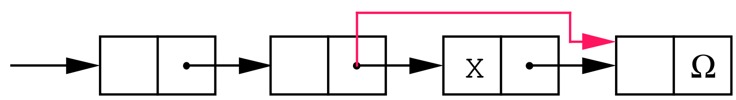
DOUBLY-LINKED LISTS






```
RLINK(LLINK(Y)) = LLINK(RLINK(Y)) = Y
```

- Disadvantage: More space for links
- Advantage: Given X, it can be deleted without having to locate its predecessor as is necessary with singly-linked lists



Easy to insert a node to the left or right of another node:

Insert to the right of `Z`:

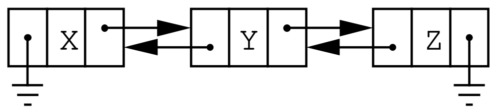
```
P⇐AVAIL;
LLINK(P)←Z; RLINK(P)←RLINK(Z);
LLINK(RLINK(Z))←P; RLINK(Z)←P;
```

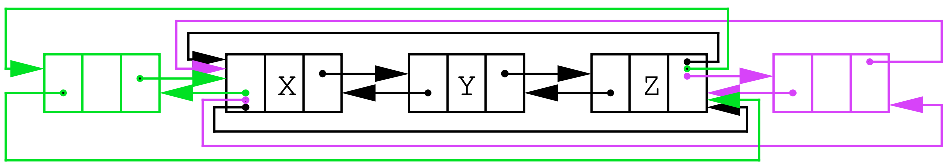
Insert to the left of `X`:

Interchange LEFT and RIGHT in ‘Insertion to the right’.

- 4 links are changed (only 2 changed with singly-linked list)

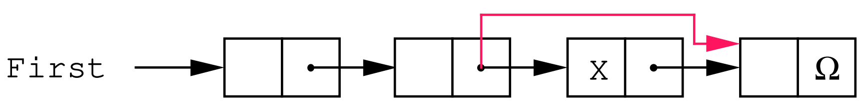
DOUBLY-LINKED LISTS





`RLINK(LLINK(Y)) = LLINK(RLINK(Y)) = Y`

- Disadvantage: More space for links
- Advantage: Given X, it can be deleted without having to locate its predecessor as is necessary with singly-linked lists



Easy to insert a node to the left or right of another node:

Insert to the right of `Z`:

```
P⇐AVAIL;
LLINK(P)←Z; RLINK(P)←RLINK(Z);
LLINK(RLINK(Z))←P; RLINK(Z)←P;
```

Insert to the left of `X`:

Interchange LEFT and RIGHT in ‘Insertion to the right’.

- 4 links are changed (only 2 changed with singly-linked list)

TWO LINKS FOR THE PRICE OF ONE

Exclusive Or:

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

$A \oplus A = 0$

$A \oplus 0 = A \qquad A \oplus 1 = \overline{A}$

$A \oplus B = B \oplus A$

$(A \oplus B) \oplus C = A \oplus (B \oplus C)$

$A \oplus A \oplus B = B$

Let $\mathrm{LINK}(X_i) = \mathrm{LOC}(X_{i+1}) \oplus \mathrm{LOC}(X_{i-1})$

$X_{i-1} \qquad X_i \qquad X_{i+1}$

Knowing 2 successive locations (L_i, L_{i+1}) allows going left and right.

$L_0: \qquad L_1: \qquad L_2: \qquad L_3:$

$\mathrm{RIGHT}(L_2) = \mathrm{LINK}(L_2) \oplus L_1 = L_3 \oplus L_1 \oplus L_1 = L_3$

$\mathrm{LEFT}(L_1) = \mathrm{LINK}(L_1) \oplus L_2 = L_0 \oplus L_2 \oplus L_2 = L_0$

Ex: Exchange the contents of two locations without using temporaries

$B \leftarrow A \oplus B \qquad A \oplus B$

$A \leftarrow A \oplus B \qquad A \oplus (A \oplus B) = B$

$B \leftarrow A \oplus B \qquad B \oplus (A \oplus B) = A$

ARRAYS

- Generalization of a linear list
- Allocate storage sequentially
- LOC(A[m,n]) $\equiv A_0 + A_1 \cdot m + A_2 \cdot n$
  A_0, A_1, A_2 are constants
- Ex: Q[0:3,0:2,0:1]

Q[0,0,0]	Q[0,0,0]
Q[0,0,1]	Q[1,0,0]
Q[0,1,0]	Q[2,0,0]
Q[0,1,1]	Q[3,0,0]
Q[0,2,0]	Q[0,1,0]
Q[0,2,1]	Q[1,1,0]
Q[1,0,0]	Q[2,1,0]
⋮	⋮
Q[3,2,0]	Q[2,2,1]
Q[3,2,1]	Q[3,2,1]
Row-major order ALGOL	Column-major order FORTRAN

- Row-major is preferable = lexicographic order of indices
- LOC(Q[i,j,k]) = LOC(Q[0,0,0]) $+ 6 \cdot i + 2 \cdot j + k$

K-DIMENSIONAL ARRAYS

- $A[l_1{:}u_1,\ l_2{:}u_2,\ \ldots,\ l_k{:}u_k]$

- $$\begin{aligned}\textsc{loc}(A[i_1, i_2, \ldots, i_k]) &= \textsc{loc}(A[l_1,l_2,l_3,\ldots,l_k]) + \\ &\quad (u_2-l_2+1) \ldots (u_k-l_k+1)\cdot(i_1-l_1) + \ldots \\ &\quad (u_k-l_k+1)\cdot(i_{k-1}-l_{k-1}) + i_k-l_k \\ &= \textsc{loc}(A[l_1,l_2,l_3,\ldots,l_k]) + \sum_{r=1}^{k} A_r\cdot(i_r-l_r) \\ &= \{\textsc{loc}(A[l_1,l_2,l_3,\ldots,l_k]) - \sum_{r=1}^{k} A_r\cdot l_r\} + \sum_{r=1}^{k} A_r\cdot i_r\end{aligned}$$

$$A_r = \prod_{r<s\le k} (u_s-l_s+1)$$

$$A_k = 1$$

- Semantics of A_r:
  1. let $i_1, i_2, \ldots, i_r$ be constant
  2. let $j_{r+1}, j_{r+2}, \ldots, j_k$ vary through $l_i \le j_i \le u_i$
  3. consider $A[i_1, i_2, \ldots, i_r, j_{r+1}, j_{r+2}, \ldots, j_k]$
     - when i_r changes by 1 $\textsc{loc}(A[i_1, i_2, \ldots, i_k])$ changes by A_r

ARRAY DESCRIPTOR

- 'Dope vector'
- Ex: Q[0:3,0:2,0:1]

Q_0	Address of first element
Real	Type (string, real, complex, ?)
3	# of dimensions
0	l_1
3	u_1
6	A_1
0	l_2
2	u_2
2	A_2
⋮	
0	l_n
1	u_n
1	A_n

- Why store the bounds?
- Not needed in the access function!

TRIANGULAR MATRIX

- LOC(A[j,k]) = $A_0 + F_1(j) + F_2(k)$

$$\begin{bmatrix} A[0,0] & & & \\ A[1,0] & A[1,1] & & \\ \vdots & & & \\ A[n,0] & A[n,1] & \ldots & A[n,n] \end{bmatrix}$$

- Two triangular matrices:

$$\begin{bmatrix} A[0,0] & B[0,0] & B[1,0] & \ldots & B[n,0] \\ A[1,0] & A[1,1] & B[1,1] & \ldots & B[n,1] \\ \vdots & & & & \\ A[n,0] & A[n,1] & \ldots & A[n,n] & B[n,n] \end{bmatrix} = C$$

A[j,k] =

B[j,k] =

TRIANGULAR MATRIX

- $\text{LOC}(A[j,k]) = A_0 + F_1(j) + F_2(k)$

$$\begin{bmatrix} A[0,0] & & & \\ A[1,0] & A[1,1] & & \\ \vdots & & & \\ A[n,0] & A[n,1] & \ldots & A[n,n] \end{bmatrix}$$

$$\text{LOC}(A[j,k]) = \text{LOC}(A[0,0]) + \left(\sum_{i=0}^{j-1} i+1\right) + k$$

$$= \text{LOC}(A[0,0]) + \frac{j\cdot(j+1)}{2} + k$$

- quadratic access function (not linear)

- Two triangular matrices:

$$\begin{bmatrix} A[0,0] & B[0,0] & B[1,0] & \ldots & B[n,0] \\ A[1,0] & A[1,1] & B[1,1] & \ldots & B[n,1] \\ \vdots & & & & \\ A[n,0] & A[n,1] & \ldots & A[n,n] & B[n,n] \end{bmatrix} = C$$

$A[j,k] =$

$B[j,k] =$

TRIANGULAR MATRIX

- $\text{LOC}(A[j,k]) = A_0 + F_1(j) + F_2(k)$

$$\begin{bmatrix} A[0,0] & & & \\ A[1,0] & A[1,1] & & \\ \vdots & & & \\ A[n,0] & A[n,1] & \dots & A[n,n] \end{bmatrix}$$

$$\begin{aligned} \text{LOC}(A[j,k]) &= \text{LOC}(A[0,0]) + \left(\sum_{i=0}^{j-1} i+1\right) + k \\ &= \text{LOC}(A[0,0]) + \frac{j\cdot(j+1)}{2} + k \end{aligned}$$

- quadratic access function (not linear)

- Two triangular matrices:

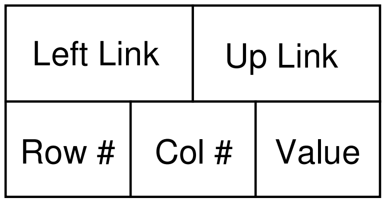
$$\begin{bmatrix} A[0,0] & B[0,0] & B[1,0] & \dots & B[n,0] \\ A[1,0] & A[1,1] & B[1,1] & \dots & B[n,1] \\ \vdots & & & & \\ A[n,0] & A[n,1] & \dots & A[n,n] & B[n,n] \end{bmatrix} = C$$

$A[j,k] = C[j,k]$

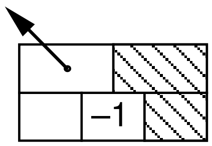
$B[j,k] = C[k,j+1]$

SPARSE MATRICES

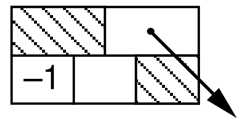
• For each item:



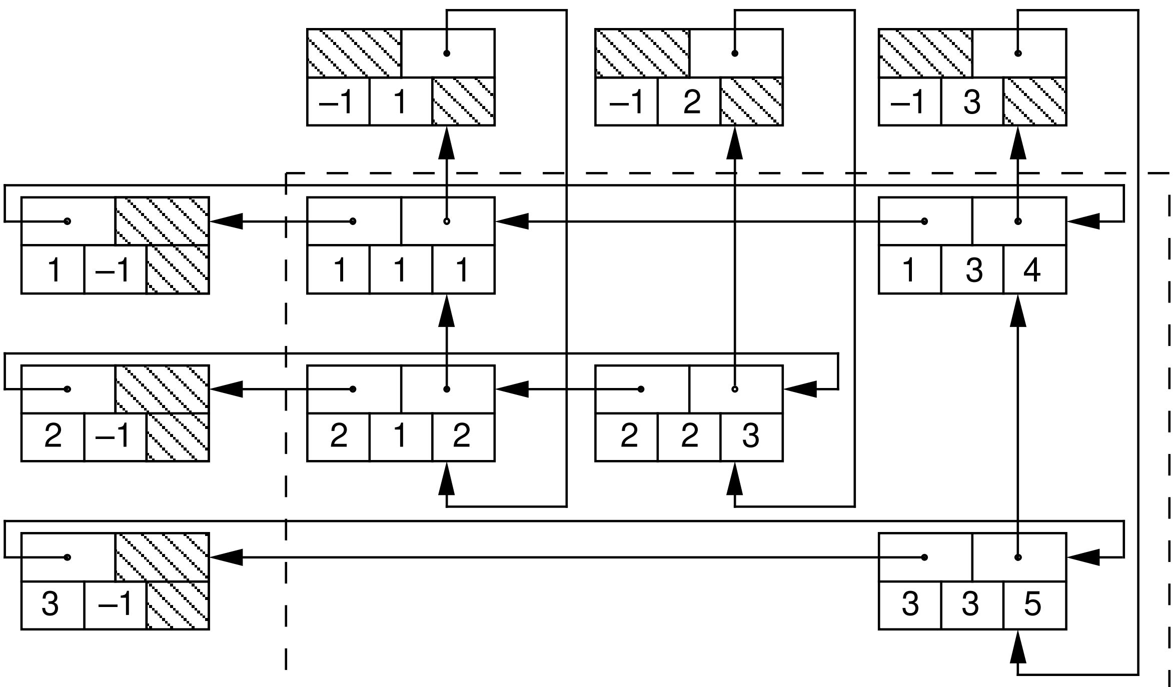
• For each row:



For each column:



• Ex: $\begin{pmatrix} 1 & & 4 \\ 2 & 3 & \\ & & 5 \end{pmatrix}$



• Circular list is useful for insertion and deletion of elements

• Ex: compute $C = C + A \cdot B$

$$C_{ik} = C_{ik} + \sum_j A_{ij} \cdot B_{jk}$$