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. Fielddata

• 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. 

• 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: How many passengers get off at Dallas?

1. \( n \leftarrow 0; \)
2. \( x \leftarrow \text{FirstDallas}; \)
3. if \( x = \Omega \) then HALT;
4. \( n \leftarrow n + 1; \)
5. \( x \leftarrow \text{MDestino}(x); \)
6. goto 3;

PASCAL:

\[ n \leftarrow 0; \]
\[ x \leftarrow \text{FirstDallas}; \]
while \( x \neq \Omega \) do
begin
  \[ n \leftarrow n + 1; \]
  \[ x \leftarrow x \uparrow \text{MDestino}; \]
end;

Field names: \( \text{MVuelo}, \text{MDestino} \)
Variable names: \( n, x, \text{First455}, \text{FirstDallas}, \text{NewPass} \)
Integer variable: \( n \)
Link variables: \( x, \text{First455}, \text{FirstDallas}, \text{NewPass} \)
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 $x[1], x[2], \ldots, x[n]$ \hspace{1cm} (n≥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 \hspace{1cm} $k=1$ or $k=n$ \hspace{1cm} are interesting
  1. stack: \hspace{1cm} insert and delete at the same end
  2. queue: \hspace{1cm} insert at one end
     \hspace{1cm} delete at the other end
  3. deque: \hspace{1cm} insert and delete at both ends
STACKS

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

Delete

FIRST
FRONT
FIFO

Insert

SECOND
THIRD
LAST
REAR

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

\[
\text{LOC}(x[j+1]) = \text{LOC}(x[j]) + c
\]

node size = \( c \)

\[
\text{LOC}(x[j]) = L_0 + c \cdot j \quad \text{where} \quad L_0 = \text{LOC}(x[0])
\]

• 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:

\[
T \leftarrow T + 1;
\]

\[
x[T] \leftarrow y;
\]

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

\[
y \leftarrow x[T];
\]

\[
T \leftarrow T - 1;
\]
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:

\[
\begin{align*}
\text{if } R = M & \text{ then } R \leftarrow 1 \\
\text{else} & \quad R \leftarrow R + 1; \\
& \quad x[R] \leftarrow Y;
\end{align*}
\]

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

\[
\begin{align*}
\text{if } F = M & \text{ then } F \leftarrow 1 \\
\text{else} & \quad F \leftarrow F + 1; \\
& \quad Y \leftarrow x[F];
\end{align*}
\]

\[
\text{if } F = R \text{ then } F \leftarrow R \leftarrow 0;
\]

• 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

\[
x[1] \ x[2] \ \ldots \ x[M] \ x[1]
\]

\( R = F = M \) 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
   \[ T \leftarrow T + 1; \]
   if \( T > M \) then OVERFLOW;
   \[ x[T] \leftarrow Y; \]

2. Stack deletion:
   if \( T = 0 \) then UNDERFLOW;
   \[ Y \leftarrow x[T]; \]
   \[ T \leftarrow T - 1; \]

3. Queue insertion:
   if \( R = M \) then \( R \leftarrow 1; \)
   else \( R \leftarrow R + 1; \)
   if \( R = F \) then OVERFLOW
   else \( x[R] \leftarrow Y; \)

4. Queue deletion:
   if \( R = F \) then UNDERFLOW
   else
   \[ \text{begin} \]
   if \( F = M \) then \( F \leftarrow 1 \)
   else \( F \leftarrow F + 1; \)
   \[ Y \leftarrow x[F]; \]
   \[ \text{end}; \]

- We start with \( F = R = M \)
- UNDERFLOW is not a real problem
MULTIPLE STACKS

- Two stacks can grow towards each other


- 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:

\[
\text{TOP}[i] \leftarrow \text{TOP}[i] + 1; \\
\text{if TOP}[i] > \text{BASE}[i+1] \text{ then OVERFLOW}; \\
\text{else CONTENTS(TOP}[i]) \leftarrow Y
\]

Deletion from stack i:

\[
\text{if TOP}[i] = \text{BASE}[i] \text{ then UNDERFLOW}; \\
Y \leftarrow \text{CONTENTS(TOP}[i]); \\
\text{TOP}[i] \leftarrow \text{TOP}[i] - 1;
\]

When stack i overflows:

1. find smallest \( k \ni i < k \leq n \) and \( \text{TOP}[k] < \text{BASE}[k+1] \)
   for \( \text{TOP}[k] \geq m > \text{BASE}[i+1] \)
   \( \text{CONTENTS(m+1)} \leftarrow \text{CONTENTS(m)} \)
   for \( i < j \leq k \)
   \( \text{BASE}[j] \leftarrow \text{BASE}[j] + 1; \text{TOP}[j] \leftarrow \text{TOP}[j] + 1; \)

2. find largest \( k \ni 1 \leq k < i \) and \( \text{TOP}[k] < \text{BASE}[k+1] \)
   for \( \text{BASE}[k+1] < m < \text{TOP}[i] \)
   \( \text{CONTENTS(m-1)} \leftarrow \text{CONTENTS(m)} \)
   for \( k < j \leq i \)
   \( \text{BASE}[j] \leftarrow \text{BASE}[j] - 1; \text{TOP}[j] \leftarrow \text{TOP}[j] - 1; \)

3. if \( \text{TOP}[k] = \text{BASE}[k+1] \forall k \neq i \) 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 |
| Item 2 | C |
| Item 3 | D |
| ... |   |
| Item n | Ω |

- Each node has two fields

Info Link

- Need a pointer to FIRST element

FIRST

Item 1

Item 2

Item 3

... 

Item n Ω

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 (⇒ memory access)

7. S takes advantage of locality
STORAGE MANAGEMENT

- Linked list of available storage
- \texttt{AVAIL} points to the first element
- Use \texttt{LINK} field

\texttt{x} \leftarrow \texttt{AVAIL} is short hand notation for allocating a new node as follows:

\begin{verbatim}
if AVAIL=\Omega then OVERFLOW
else
begin
  x \leftarrow \texttt{AVAIL};
  AVAIL \leftarrow \texttt{LINK(AVAIL)};
  \texttt{LINK(x)\leftarrow\Omega};
end;
\end{verbatim}

\texttt{AVAIL\leftarrow x} is short hand notation for returning a node as follows:

\begin{verbatim}
\texttt{LINK(x)\leftarrow AVAIL;}
\texttt{AVAIL\leftarrow x;}
\end{verbatim}
COMBINING SEQUENTIAL AND LINKED STORAGE

Allocation of a node of linked storage ($x$):

if $\text{AVAIL}=\Omega$ then
  if $\text{PoolMax}>\text{SeqMin}$ then OVERFLOW
else
  begin
    $\text{PoolMax} \leftarrow \text{PoolMax}+1$;
    $x \leftarrow \text{PoolMax}$;
  end;
else $x \leftarrow \text{AVAIL}$;

- No need to initially link up $\text{AVAIL}$

- A similar scheme is used in DBMS-10 for storing records on disk pages

logical address = $la = \begin{bmatrix} \text{page} & \text{line} \end{bmatrix}$

physical address = $\begin{bmatrix} \text{page} \end{bmatrix}(la) 0$ + CONTENTS[line #(la)]
LINKED STACKS

Insert $Y$ into a linked stack:

$T = \text{top of stack pointer}$

\begin{align*}
    p & \leftarrow \text{AVAIL}; \\
    \text{INFO}(p) & \leftarrow Y; \\
    \text{LINK}(p) & \leftarrow T; \\
    T & \leftarrow p;
\end{align*}

Delete $Y$ from a linked stack:

\begin{align*}
    & \text{if } T = \Omega \text{ then UNDERFLOW;} \\
    & p \leftarrow T; \\
    & T \leftarrow \text{LINK}(p); \\
    & Y \leftarrow \text{INFO}(p); \\
    & \text{AVAIL} \leftarrow p;
\end{align*}
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;
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≠X (asymmetry)
  3. X≠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
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:
  - \( \text{PRED\_COUNT}[t[K]] \equiv \# \text{ of direct predecessors of } K \) (i.e., \( L < K \))
  - \( \text{SUCCESSORS}[t[K]] \equiv \) pointer to a linked list containing the direct successors of item \( K \)
  
  Ex: \( t[7] \):
  
  ```
  1       4       5  \Omega
  \text{PRED\_COUNT} \downarrow \text{SUCCESSORS} \downarrow \text{DATA} \downarrow \text{NEXT}
  ```

- Maintain a queue of all items having 0 predecessors
- Each time item \( K \) is output:
  1. remove \( t[K] \) from the queue
  2. decrement \( \text{PRED\_COUNT} \) field of all successors of \( K \)
  3. add to the queue any node whose \( \text{PRED\_COUNT} \) 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 \texttt{PRED\_COUNT} field of \texttt{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 \texttt{t[K]} whose size is fixed
• Linked allocation for the successor relations

• Queue is linked by index (à la \texttt{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:
   
   $P \leftarrow \text{AVAIL}; \quad \text{INFO}(P) \leftarrow y$
   
   if $\text{PTR} = \Omega$ then $\text{PTR} \leftarrow \text{LINK}(P) \leftarrow P$
   
   else
   
   begin
   
   $\text{LINK}(P) \leftarrow \text{LINK}(\text{PTR}); \quad \text{LINK}(\text{PTR}) \leftarrow P$
   
   end;

2. Insert $y$ at the right:
   
   Insert $y$ at the left;

   $\text{PTR} \leftarrow P$;

3. Set $y$ to the left node and delete:
   
   if $\text{PTR} = \Omega$ then UNDERFLOW;

   $P \leftarrow \text{LINK}(\text{PTR}); \quad y \leftarrow \text{INFO}(P)$;

   $\text{LINK}(\text{PTR}) \leftarrow \text{LINK}(P); \quad \text{AVAIL} \leftarrow P$

   if $\text{PTR} = P$ then $\text{PTR} \leftarrow \Omega$

   
   /* 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.
DOUBLY-LINKED LISTS

\[ \text{RLINK(LLINK(Y)) = LLINK(RLINK(Y)) = Y} \]

- 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:**

\[
P \leftarrow \text{AVAIL};
\]

\[
\text{LLINK}(P) \leftarrow Z; \text{RLINK}(P) \leftarrow \text{RLINK}(Z);
\]

\[
\text{LLINK(RLINK}(Z)) \leftarrow P; \text{RLINK}(Z) \leftarrow P;
\]

**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:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A⊕B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A⊕A = 0  
A⊕0 = A  
A⊕1 = A

A⊕B = B⊕A

(A⊕B)⊕C = A⊕(B⊕C)

A⊕A⊕B = B

Let  LINK(X_i) = LOC(X_{i+1}) ⊕ LOC(X_{i-1})

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

Ex: Exchange the contents of two locations without using temporaries

B ← A⊕B  
A ← A⊕B  
B ← B⊕(A⊕B) = A
ARRAYS

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

<table>
<thead>
<tr>
<th>Row-major order</th>
<th>Column-major order</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q[0,0,0]$</td>
<td>$Q[0,0,0]$</td>
</tr>
<tr>
<td>$Q[0,0,1]$</td>
<td>$Q[1,0,0]$</td>
</tr>
<tr>
<td>$Q[0,1,0]$</td>
<td>$Q[2,0,0]$</td>
</tr>
<tr>
<td>$Q[0,1,1]$</td>
<td>$Q[3,0,0]$</td>
</tr>
<tr>
<td>$Q[0,2,0]$</td>
<td>$Q[0,1,0]$</td>
</tr>
<tr>
<td>$Q[0,2,1]$</td>
<td>$Q[1,1,0]$</td>
</tr>
<tr>
<td>$Q[1,0,0]$</td>
<td>$Q[2,1,0]$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$Q[3,2,0]$</td>
<td>$Q[2,2,1]$</td>
</tr>
<tr>
<td>$Q[3,2,1]$</td>
<td>$Q[3,2,1]$</td>
</tr>
</tbody>
</table>

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

- \( A[l_1:u_1, l_2:u_2, ..., l_k:u_k] \)

- \( \text{LOC}(A[i_1, i_2, ..., i_k]) = \text{LOC}(A[l_1,l_2,l_3,..,l_k]) + \\
  (u_2-l_2+1) \cdots (u_k-l_k+1)(i_1-l_1) + \cdots \\
  (u_k-l_k+1)(i_k-l_k) + i_k-l_k \\
  = \text{LOC}(A[l_1,l_2,l_3,..,l_k]) + \sum_{r=1}^{k} A_r \cdot (i_r-l_r) \\
  = \{ \text{LOC}(A[l_1,l_2,l_3,..,l_k]) - \sum_{r=1}^{k} A_r \cdot l_r \} + \sum_{r=1}^{k} A_r \cdot i_r \)

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

\( A_k = 1 \)

- 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 \( l_i \leq j_i \leq u_i \)
  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 \( \text{LOC}(A[i_1, i_2, \ldots, i_k]) \) changes by \( A_r \)
ARRAY DESCRIPTOR
• ‘Dope vector’
• Ex: Q[0:3,0:2,0:1]

<table>
<thead>
<tr>
<th>Q₀</th>
<th>Address of first element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>Type (string, real, complex, ?)</td>
</tr>
<tr>
<td>3</td>
<td># of dimensions</td>
</tr>
<tr>
<td>0</td>
<td>u₁</td>
</tr>
<tr>
<td>3</td>
<td>A₁</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>l₂</td>
</tr>
<tr>
<td>2</td>
<td>u₂</td>
</tr>
<tr>
<td>2</td>
<td>A₂</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>0</td>
<td>lₙ</td>
</tr>
<tr>
<td>1</td>
<td>uₙ</td>
</tr>
<tr>
<td>1</td>
<td>Aₙ</td>
</tr>
</tbody>
</table>

• Why store the bounds?
• Not needed in the access function!
TRIANGULAR MATRIX

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

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

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

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

• quadratic access function (not linear)

• Two triangular matrices:

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

\( A[j,k] = C[j,k] \)

\( B[j,k] = C[k,j+1] \)
SPARSE MATRICES

- For each item:

<table>
<thead>
<tr>
<th>Left Link</th>
<th>Up Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row #</td>
<td>Col #</td>
</tr>
</tbody>
</table>

- For each row:

- For each column:

- Ex: \[
\begin{pmatrix}
1 & 2 & 3 & 4 \\
2 & 3 & 4 & 5
\end{pmatrix}
\]

- Circular list is useful for insertion and deletion of elements

- Ex: compute \( C = C + A \cdot B \)

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