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

Copyright © 1997 Hanan Samet

These notes may not be reproduced by any means (mechanical or electronic or any other) without the express written permission of Hanan Samet
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.  
   \[
   \begin{array}{c}
   \text{JOHN_F} \quad \text{ITZIMM} \quad \text{ONS} \\
   \end{array}
   \]

2.  
   \[
   \begin{array}{c}
   \text{JOHN_FITZIMMONS} \\
   \end{array}
   \]

3.  
   \[
   \begin{array}{c}
   \text{JOHN_FITZIMMONS} \\
   \end{array}
   \]

4.  
   \[
   \begin{array}{c}
   \text{JOHN_FITZIMMONS} \Omega \\
   \end{array}
   \]

Copyright © 1998 by Hanan Samet
CHARACTER DATA

1. 
\[ \text{JOHN}_F \rightarrow \text{ITZIMM} \rightarrow \Omega \]
\[ \text{ITZPAT} \rightarrow \Omega \]

2. 
\[ \text{JOH} \rightarrow \text{N}_F \rightarrow \text{ITZ} \rightarrow \text{IMM} \rightarrow \text{ONS} \rightarrow \Omega \]

3. 
\[ 15 \]
\[ \text{JOHN}_F \text{FITZIMMONS} \]

4. 
\[ \text{JOHN}_F \text{FITZIMMONS} \rightarrow \Omega \]

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

1. \[ \text{JOHN}_F \rightarrow \text{ITZIMM} \rightarrow \Omega \rightarrow \text{ONS} \rightarrow \text{ITZPAT} \rightarrow \text{RICK} \]

2. \[ \text{JOHN}_F \rightarrow \text{ITZIMM} \rightarrow \Omega \rightarrow \text{ONS} \rightarrow \text{CUR} \rightarrow \text{T_S} \]

3. \[ \text{JOHN}_F \text{FITZIMMONS} \]

4. \[ \text{JOHN}_F \text{FITZIMMONS} \Omega \]

- 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. JOHN_F ITZIMM ONS
   ITZPAT RICK

2. JOH N_F ITZ IMM ONS
   CUR T_S

3. 15 3 7
   JOHN_F ITZIMM ONS
   ________________

4. __________
   JOHN_F ITZIMM ONS

• 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
• 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. 

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 \equiv \text{pointer to the first passenger on flight 455} \\
FirstDallas \equiv \text{pointer to the first passenger to Dallas} \\
NewPass \equiv \text{pointer to the new passenger.}

PASCAL

1. \text{MVuelo}(NewPass) \leftarrow \text{First455} \\
2. \text{First455} \leftarrow \text{NewPass} ; \\
3. \text{MDestino}(NewPass) \leftarrow \text{FirstDallas} ; \\
4. \text{FirstDallas} \leftarrow \text{NewPass} ;

Copyright © 1998 by Hanan Samet
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;
3. MDestino(NewPass)← FirstDallas;
4. 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:

\[
\begin{align*}
\text{n} & \leftarrow 0; \\
\text{x} & \leftarrow \text{FirstDallas}; \\
\text{while} \ x \neq \Omega \ 	ext{do} \\
\hspace{1cm} & \begin{align*}
\hspace{1cm} & \text{n} \leftarrow n+1; \\
\hspace{1cm} & \text{x} \leftarrow \text{x^{\uparrow}.MDestino}; \\
\hspace{1cm} & \text{end;}
\end{align*}
\end{align*}
\]

Field names: MVuelo, 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 $x[1], x[2], \ldots x[n]$  
  \( (n \geq 1) \)

- 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

Copyright © 1998 by Hanan Samet
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 A+B
Prefix: operator operand operand +AB
Postfix: operand operand operator AB+

Postfix = ‘Polish notation’

A+B*C ⇒ ABC*+

Stack

Enter A
Enter B
Enter C
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 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
Enter B
Enter C

Enter *
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   A+B
Prefix: operator operand operand  +AB
Postfix: operand operand operator AB+

Postfix \(\equiv\) ‘Polish notation’

\[ A+B\times C \Rightarrow ABC\times+ \]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Enter</th>
<th>Enter</th>
<th>Enter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>B</td>
<td>B*C</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>A</td>
<td>A+B*C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>+</td>
</tr>
</tbody>
</table>
QUEUE:

Delete
FIRST
FRONT
FIFO

Insert
SECOND
THIRD
LAST
REAR

DEQUE:

INPUT
OUTPUT
**QUEUE:**

- Delete
- Insert

**FIFO**

- FRONT
- FIRST
- SECOND
- THIRD
- ...
- LAST
- REAR

**DEQUE:**

- Input restricted deque
- Output restricted deque

Copyright © 1998 by Hanan Samet
Question: how would you construct a stack from a deque?
SEQUENTIAL ALLOCATION

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

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

\[ \text{node size} = c \]

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

• 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:

\[
  R \leftarrow R + 1;
  x[R] \leftarrow Y;
\]

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

\[
  F \leftarrow F + 1;
  Y \leftarrow x[F];
  \text{if } F = R \text{ then } F \leftarrow R \leftarrow 0;
\]

• 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)
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

```
```

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:

  \[
  \text{if } R = M \text{ then } R \leftarrow 1 \\
  \text{else } R \leftarrow R + 1; \\
  x[R] \leftarrow Y; \\
  \]

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

  \[
  \text{if } F = M \text{ then } F \leftarrow 1 \\
  \text{else } F \leftarrow F + 1; \\
  Y \leftarrow x[F]; \\
  \text{if } F = R \text{ then } F \leftarrow R \leftarrow 0; \\
  \]

• 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

  \[
  x[1] \ x[2] \ldots \ x[M] \ x[1] \\
  R = F = M \text{ when the queue is empty (initially)} \\
  \]

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

• Removal of an entry from the front of the queue:
  \[
  \begin{align*}
  &\text{if } F = M \text{ then } F \leftarrow 1 \\
  &\text{else} \quad F \leftarrow F + 1; \\
  &\quad Y \leftarrow x[F]; \\
  &\text{if } F = R \text{ then } F \leftarrow R \leftarrow 0;
  \end{align*}
  \]

• 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
   \[ T \leftarrow T + 1; \]
   \[ \text{if } T > M \text{ then OVERFLOW; } \]
   \[ x[T] \leftarrow Y; \]

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

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

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

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

Copyright © 1998 by Hanan Samet
OVERFLOW

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

1. Stack insertion
   \[
   T \leftarrow T+1;
   \text{if } T>M \text{ then OVERFLOW;}
   x[T] \leftarrow Y;
   \]

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

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

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

• 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
   \[ T \leftarrow T + 1; \]
   \[ \text{if } T > M \text{ then OVERFLOW; } \]
   \[ x[T] \leftarrow Y; \]

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

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

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

- 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
   \[ T \leftarrow T + 1; \]
   \[ \text{if } T > M \text{ then OVERFLOW;} \]
   \[ x[T] \leftarrow Y; \]

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

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

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

- 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

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

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 1</td>
<td>Item 1</td>
</tr>
<tr>
<td>Item 2</td>
<td>Item 2</td>
</tr>
<tr>
<td>Item 3</td>
<td>Item 3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Item n</td>
<td>Item n</td>
</tr>
</tbody>
</table>

- Each node has two fields
  - Info
  - Link

- Need a pointer to FIRST element

Ω denotes the end of the list

Copyright © 1998 by Hanan Samet
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 AVAIL} is short hand notation for allocating a new node as follows:

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

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

\begin{verbatim}
\text{LINK(x)} \leftarrow AVAIL;
AVAIL \leftarrow x;
\end{verbatim}
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 = W then OVERFLOW
else
  begin
    x ← AVAIL;
    AVAIL ← LINK(AVAIL);
    LINK(x) ← W;
  end;
```

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

LINK(x) ← AVAIL;
AVAIL ← x;
```

Copyright © 1998 by Hanan Samet
STORAGE MANAGEMENT

• Linked list of available storage

• AVAIL points to the first element

• Use LINK field

\[
x \leftarrow \text{AVAIL} \quad \text{is short hand notation for allocating a new node as follows:}
\]

\[
\text{if } \text{AVAIL} = \Omega \text{ then OVERFLOW}
\]

\[
\text{else begin}
\]

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

\[
\text{AVAIL} \leftarrow \text{LINK(AVAIL)};
\]

\[
\text{LINK(x)} \leftarrow \Omega;
\]

\[
\text{end;}
\]

\[
\text{AVAIL} \leftarrow x \quad \text{is short hand notation for returning a node as follows:}
\]

\[
\text{LINK(x)} \leftarrow \text{AVAIL};
\]

\[
\text{AVAIL} \leftarrow x;
\]
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)]

Copyright © 1998 by Hanan Samet
LINKED STACKS

Insert $y$ into a linked stack:

$T =$ top of stack pointer

$p \leftarrow$ AVAIL;
INFO($p$) $\leftarrow$ $y$;
LINK($p$) $\leftarrow$ $T$;
$T \leftarrow p$;

Delete $y$ from a linked stack:

if $T = \Omega$ then UNDERFLOW;
p $\leftarrow$ $T$;
$T \leftarrow$ LINK($p$);
$y \leftarrow$ INFO($p$);
AVAIL $\leftarrow p$;
LINKED STACKS

Insert $y$ into a linked stack:

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

$p \leftarrow \text{AVAIL};$
INFO($p$) $\leftarrow y;$
LINK($p$) $\leftarrow T;$
$T \leftarrow p;$

Delete $y$ from a linked stack:

if $T = \Omega$ then UNDERFLOW;
p $\leftarrow T;$
$T \leftarrow \text{LINK}(p);$
y $\leftarrow \text{INFO}(p);$
AVAIL $\leftarrow p;$
LINKED STACKS

Insert $y$ into a linked stack:

$T =$ 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;
LINKED QUEUES

\[ F = \omega \] signifies an empty queue

**Insert** \( y \) at the rear of a queue:

1. \( P \leftarrow \text{AVAIL}; \)
2. \( \text{INFO}(P) \leftarrow y; \)
3. \( \text{LINK}(P) \leftarrow \omega; \)
4. if \( F = \omega \) then \( F \leftarrow P; \)
5. else \( \text{LINK}(R) \leftarrow P; \)
6. \( R \leftarrow P; \)

**Delete** \( y \) from the front of a queue:

1. if \( F = \omega \) then \text{UNDERFLOW};
2. \( P \leftarrow F; \)
3. \( F \leftarrow \text{LINK}(P); \)
4. \( Y \leftarrow \text{INFO}(P); \)
5. \( \text{AVAIL} \leftarrow P; \)
LINKED QUEUES

\[ F = \emptyset \] signifies an empty queue

**Insert** \( y \) at the rear of a queue:

\[
\begin{align*}
P &\leftarrow \text{AVAIL}; \\
\text{INFO}(P) &\leftarrow y; \\
\text{LINK}(P) &\leftarrow \Omega; \\
\text{if } F = \Omega &\text{ then } F \leftarrow P; \\
\text{else } &\text{ LINK}(R) \leftarrow P; \\
R &\leftarrow P;
\end{align*}
\]

**Delete** \( y \) from the front of a queue:

\[
\begin{align*}
\text{if } F = \Omega &\text{ then UNDERFLOW}; \\
P &\leftarrow F; \\
F &\leftarrow \text{LINK}(P); \\
Y &\leftarrow \text{INFO}(P); \\
\text{AVAIL} &\leftarrow P;
\end{align*}
\]
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≠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., $L < K$)
    - $\text{SUCCESSORS}[t[K]] \equiv \text{pointer to a linked list containing the direct successors of item } K$
  - Ex: $t[7]$: [Diagram]
    - 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 `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\leftarrow\text{AVAIL}; \quad \text{INFO}(P)\leftarrow y;
   
   \text{if } \text{PTR} = \Omega \text{ then } \text{PTR} \leftarrow \text{LINK}(P) \leftarrow P
   \text{else}
   \text{begin}
     \text{LINK}(P) \leftarrow \text{LINK}(\text{PTR}); \quad \text{LINK}(\text{PTR}) \leftarrow P;
   \text{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:
   
   ```
   \text{if } \text{PTR} = \Omega \text{ 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;
   \text{if } \text{PTR} = P \text{ then } \text{PTR} \leftarrow \Omega;
   /* 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:
\[
P \leftarrow \text{AVAIL}; \quad \text{INFO}(P) \leftarrow y;
\]
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:
\[
\text{if PTR}=\Omega \quad \text{then UNDERFLOW;}
\]
\[
P \leftarrow \text{LINK}(\text{PTR}); \quad y \leftarrow \text{INFO}(P);
\]
\[
\text{LINK}(\text{PTR}) \leftarrow \text{LINK}(P); \quad \text{AVAIL} \leftarrow P;
\]
\[
\text{if PTR}=P \quad \text{then PTR} \leftarrow \Omega;
\]
/* 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 \leftarrow \text{AVAIL}; \quad \text{INFO}(P) \leftarrow Y;
   
   \text{if PTR} = \Omega \quad \text{then PTR} \leftarrow \text{LINK}(P) \leftarrow P
   
   \text{else}
   
   \quad \text{begin}
   
   \quad \text{LINK}(P) \leftarrow \text{LINK}(\text{PTR}); \quad \text{LINK}(\text{PTR}) \leftarrow P;
   
   \quad \text{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:**
   
   ```
   \text{if PTR} = \Omega \quad \text{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;
   
   \text{if PTR} = P \quad \text{then PTR} \leftarrow \Omega;
   
   /* 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:**
   
   \[
   P \leftarrow \text{AVAIL}; \quad \text{INFO}(P) \leftarrow y;
   \]
   
   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 \( \text{UNDERFLOW}; \)
   
   \( P \leftarrow \text{LINK}(\text{PTR}); \quad y \leftarrow \text{INFO}(P); \)
   
   \( \text{LINK}(\text{PTR}) \leftarrow \text{LINK}(P); \quad \text{AVAIL} \leftarrow P; \)
   
   if \( \text{PTR}=P \) then \( \text{PTR} \leftarrow \Omega; \)
   
   /* Check for a list of one element */
   /* before deleting */
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;
   
   \text{if } \text{PTR} = \Omega \text{ then } \text{PTR} \leftarrow \text{LINK}(P) \leftarrow P
   
   \text{else begin}
   
   \quad \text{LINK}(P) \leftarrow \text{LINK}(\text{PTR}); \quad \text{LINK}(\text{PTR}) \leftarrow P;
   
   \text{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:**
   
   ```
   \text{if } \text{PTR} = \Omega \text{ 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;
   
   \text{if } \text{PTR} = P \text{ 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: $\text{PTR}$ is meaningless after erasing a list

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

Assume $\text{PTR1}$ points to L1 and $\text{PTR2}$ points to L2.

if $\text{PTR2} \neq \Omega$ then
begin
  if $\text{PTR1} \neq \Omega$ then $\text{LINK(PTR1)} \leftrightarrow \text{LINK(PTR2)}$;
  $\text{PTR1} \leftarrow \text{PTR2}$;
  $\text{PTR2} \leftarrow \Omega$;
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.

Copyright © 1998 by Hanan Samet
ERASING A CIRCULAR LIST

Note: \textit{PTR} is meaningless after erasing a list

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

Assume \textit{PTR1} points to L1 and \textit{PTR2} points to L2.

\begin{verbatim}
if \textit{PTR2} \neq \Omega then
  begin
    if \textit{PTR1} \neq \Omega then \text{LINK(PTR1)} \leftrightarrow \text{LINK(PTR2)};
    \text{PTR1} \leftarrow \text{PTR2};
    \text{PTR2} \leftarrow \Omega;
  end
\end{verbatim}

- 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.
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:

\[
\begin{align*}
P & \leftarrow \text{AVAIL}; \\
\text{LLINK}(P) & \leftarrow Z; \ \text{RLINK}(P) & \leftarrow \text{RLINK}(Z); \\
\text{LLINK}(\text{RLINK}(Z)) & \leftarrow P; \ \text{RLINK}(Z) & \leftarrow P;
\end{align*}
\]

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

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

- 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:

\[
\begin{align*}
P & \leftarrow \text{AVAIL}; \\
\text{LLINK}(P) & \leftarrow Z; \ \text{RLINK}(P) \leftarrow \text{RLINK}(Z); \\
\text{LLINK}(\text{RLINK}(Z)) & \leftarrow P; \ \text{RLINK}(Z) \leftarrow P;
\end{align*}
\]

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

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

- **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:**

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

**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)
DOUBLY-LINKED LISTS

- Advantage: Given X, it can be deleted without having to locate its predecessor as is necessary with singly-linked lists.

Insert to the right of Z:

\[
\begin{align*}
P & \leftarrow \text{AVAIL}; \\
\text{LLINK}(P) & \leftarrow Z; \text{RLINK}(P) \leftarrow \text{RLINK}(Z); \\
\text{LLINK}(\text{RLINK}(Z)) & \leftarrow P; \text{RLINK}(Z) \leftarrow P;
\end{align*}
\]

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></th>
<th></th>
<th>A⊕B</th>
<th>A⊕A = 0</th>
<th>A⊕1 = ~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A⊕1 = ~A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(A⊕B)⊕C = A⊕(B⊕C)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A⊕A⊕B = B</td>
<td></td>
</tr>
</tbody>
</table>

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

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

\[
\begin{array}{c}
X_{i-1} \quad X_i \quad X_{i+1}
\end{array}
\]

\( \text{RIGHT}(L_2) = \text{LINK}(L_2) ⊕ L_1 = L_3 ⊕ L_1 ⊕ L_1 = L_3 \)
\( \text{LEFT}(L_1) = \text{LINK}(L_1) ⊕ L_2 = L_0 ⊕ L_2 ⊕ L_2 = L_0 \)

Ex: Exchange the contents of two locations
without using temporaries

\[
\begin{align*}
B & \leftarrow A ⊕ B \\
A & \leftarrow A ⊕ B \\
B & \leftarrow A ⊕ B
\end{align*}
\]

\[
\begin{align*}
A & = A ⊕ (A ⊕ B) \\
B & = B ⊕ (A ⊕ B)
\end{align*}
\]
ARRAYS

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

\[
\begin{array}{c|c}
\hline
Q[0,0,0] & Q[0,0,0] \\
Q[0,0,1] & Q[1,0,0] \\
Q[0,1,0] & Q[2,0,0] \\
Q[0,1,1] & Q[3,0,0] \\
Q[0,2,0] & Q[0,1,0] \\
Q[0,2,1] & Q[1,1,0] \\
Q[1,0,0] & Q[2,1,0] \\
\vdots & \vdots \\
Q[3,2,0] & Q[2,2,1] \\
Q[3,2,1] & Q[3,2,1] \\
\hline
\end{array}
\]

Row-major order  Column-major order
ALGOL          FORTRAN

• 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[l₁:u₁, l₂:u₂, …, lₖ:uₖ]

  • LOC(A[i₁, i₂, …, iₖ]) = LOC(A[l₁,l₂,…,lₖ]) +
    (u₂−l₂+1) · … · (uₖ−lₖ+1) · (i₁−l₁) + …
    (uₖ−lₖ+1) · (iₖ−lₖ−1) + iₖ−lₖ

  = LOC(A[l₁,l₂,…,lₖ]) + \sum_{r=1}^{k} A_r · (i_r−l_r)

  = {LOC(A[l₁,l₂,…,lₖ])−\sum_{r=1}^{k} A_r · l_r } + \sum_{r=1}^{k} A_r · i_r

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

  A_k = 1

• Semantics of A_r:
  1. let i₁, i₂, …, i_r be constant
  2. let j_{r+1}, j_{r+2}, …, j_k vary through l_i \leq j_i \leq u_i
  3. consider A[i₁, i₂, …, i_r, j_{r+1}, j_{r+2}, …, j_k]

    • when i_r changes by 1 LOC(A[i₁, i₂, …, i_k]) changes by A_r
### ARRAY DESCRIPTOR

- ‘Dope vector’
- Ex: $Q[0:3,0:2,0:1]$

<table>
<thead>
<tr>
<th>$Q_0$</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>$I_1$</td>
</tr>
<tr>
<td>3</td>
<td>$u_1$</td>
</tr>
<tr>
<td>6</td>
<td>$A_1$</td>
</tr>
<tr>
<td>0</td>
<td>$I_2$</td>
</tr>
<tr>
<td>2</td>
<td>$u_2$</td>
</tr>
<tr>
<td>2</td>
<td>$A_2$</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>$I_n$</td>
</tr>
<tr>
<td>1</td>
<td>$u_n$</td>
</tr>
<tr>
<td>1</td>
<td>$A_n$</td>
</tr>
</tbody>
</table>

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

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

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

- Two triangular matrices:

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

A[j,k] = 

B[j,k] =

Copyright © 1998 by Hanan Samet
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:

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

\[
A[j,k] =
\]

\[
B[j,k] =
\]
TRIANGULAR MATRIX

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

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

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

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

• quadratic access function (not linear)

• Two triangular matrices:

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

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

$B[j,k] = C[k,j+1]$
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: \[
\begin{pmatrix}
1 & 2 & 3 & 4 \\
3 & 2 & 1 & 2 \\
3 & 3 & 5 & \end{pmatrix}
\]

- Circular list is useful for insertion and deletion of elements

- Ex: compute \[ C = C + A \cdot B \]

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