DATA STRUCTURES.

1. LINEAR LISTS

Each node of a linear list (excepting the first and last nodes) has a predecessor and successor. If all insertions, deletions and access are at the first node, we have 3 special types of l.l.:

STACK (LIFO): all at one end of the list

QUEUE (FIFO): all insertions at one end and access and deletion at other end.

DEQUE: at both ends of list.

1.1. STACKS

The stack pointer points to the top node and the pointer is moved as the stack changes. Checks must be made for underflow and overflow.

1.2. DEQUES AND QUEUES.

Two pointers are needed, for the beginning and end of the list. In an empty list, the pointers are equal. Usually a circular list will be used to prevent wastage of storage space.

1.3 PASCAL IMPLEMENTATION.

1.3.1. POINTERS

Pointers are declared thus:

```
TYPE
    pointer = ↑object
```

In this case, we are saying that (pointer) is a pointer to an (object). If there is no variable of type object associated to the pointer, we say:

```
pointer := NIL
```

We also need to declare the structure of (object). This will be a record consisting of data and other pointers, eg:

```
TYPE
    link = ↑ object;
    object = RECORD
                nextobject : link;
                data : datatype;
             END;
```

1.3.2. STACK IMPLEMENTATION

Once we have defined one type, as in the previous section, we can form a stack. We need two pointers: TOP (top of stack) and PTR (current pointer), declared thus:

```
VAR
    TOP, PTR : link;
```

We can then load the stack, using the ~~word~~ word new, which allocate storage for a node.

Eg
```
BEGIN
    TOP := NIL;
    WHILE NOT eof DO
        BEGIN
            new (PTR);
            read (PTR↑.data)
            PTR↑.nextobject := TOP;
            TOP := PTR;
        END; (* while *)
```

Similarly, we can unstack the list :

```
~~IF TOP = NIL THEN GOTO EMPTYSTACK~~
WHILE (TOP <> NIL) DO BEGIN
        write (TOP↑.data);
        TOP := TOP↑.nextobject;
    END; (* while *)
```

1.3.3. QUEUE IMPLEMENTATION.

DATA STRUCTURES (DETAIL)

D.S. consist of sets of nodes. The address (usually numerical) of a node is called the pointer or link to that node.

1. ARRAYS

When an array is declared, the information about it is stored in a DOPE VECTOR

1.1 MAPPING FUNCTIONS (contiguous storage of arrays)

As an example consider a three dimensional array:

TYPE

EXAMPLE = ARRAY [L1..U1, L2..U2, L3..U3] OF arraytype

The mapping function for the address of EXAMPLE [i, j, k] is

$$m(i,j,k) = BA^* + (U3-L3+1)(U2-L2+1)(i-L1) + (U3-L3+1)(j-L2) + (k-L3)$$

If $L1, L2, L3 = 1$ this becomes:

$$m(i,j,k) = BA + U3 \cdot U2 \cdot (i-1) + U3 \cdot (j-1) + (k-1)$$

$$= BA + 1 \cdot [(k-1) + U_3[(j-1) + U_2[(i-1)]]]$$

For an n-dimensional array $[1..U_1, 1..U_2, \ldots, 1..U_n]$ we can simplify by letting:

* BA, the base address, is the address of element [1, 1, 1]

$d_1 = 1$

$d_2 = 1 . u_2 = d_1 . u_2$

$d_3 = 1 . u_2 . u_3 = d_2 . u_3$

$\vdots$

$d_n = 1 . u_2 . u_3 \ldots u_n = d_{n-1} . u_n$ etc

Then: $m[i_1, i_2, i_3, \ldots i_n]$

$= BA + d_n . (i_1 - 1) + d_{n-1} . (i_2 - 1) - \ldots + d_1 (i_n - 1)$

TRIANGULAR ARRAYS

Symmetric and skew symmetric matrices can be represented by lower triangular matrices to save storage space. The mapping function is the independent of the size of the array. Consider the following array:

A[1, 1]

A[2, 1] A[2, 2]

A[3, 1] A[3, 2] A[3, 3]

A[4, 1] A[4, 2] A[4, 3] A[4, 4]

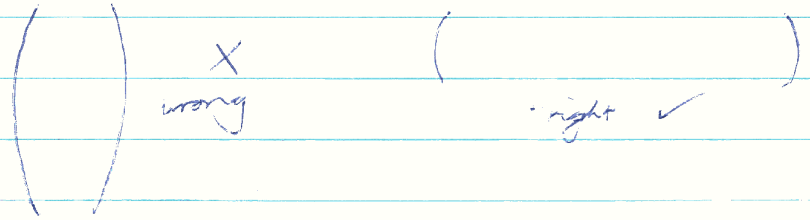
A[5, 1] etc

Then: $m(i, j) = BA + \frac{i(i-1)}{2} + (j - 1)$

1.2) ACCESS TABLES (Providing base address for each row)

Access tables eliminate the need for multiplication, are far more flexible than mapping functions, and do not require contiguous storage of array elements (but do of row

elements). The storage space required is larger than for mapping functions, but can be reduced by storing the arrays as follows: (for 'thin' matrices)



For a two dimensional array $m \times n$ the total storage required is $= m(n+1)$

APPLICATIONS

- Row interchanges : row interchanges are easy using access tables as only the access table pointers need be exchanged - no array elements need be moved.
- Jagged (irregular) arrays : these are easy to implement with access tables

1.3) ADVANTAGES & DISADVANTAGES OF TWO METHODS

MAPPING FUNCTION	ACCESS TABLE
SLOWER ACCESS	FASTER ACCESS
CONTIGUOUS STORAGE NECESSARY FOR WHOLE ARRAY	CONTIGUOUS STORAGE NECESSARY FOR INDIVIDUAL ROWS ONLY
CONSTANT SIZE ONLY	JAGGED ARRAYS POSSIBLE
FAVOURS OVERALL BOUNDS CHECK	FAVOURS INDIVIDUAL BOUNDS CHECK
	ROW INTERCHANGES EASY & FAST.

1.4) SPARSE ARRAYS AND LINKED LISTS.

If we have a large array which is mostly empty (i.e. sparse), arrays are usually wasteful on storage. One way to implement these would be to store the subscripts as well as data within linked lists. This means sequential access (disadv.) but reduced storage requirements (adv.)

2. LINKED LISTS.

A linked (linear) list is an ordered list of items where each item (except possibly the first and last) has a predecessor and successor (eg 1 dimensional array is a special case of linear list) The main types are
- stacks (LIFO)
- queues (FIFO)
- degue (both ends may be accessed)

2.1) STACKS

Instead of moving the items in a computer stack, we use a stack pointer to point to the top node, and we move this pointer. We thus have to deal with overflow and underflow considerations

2.1.1) IMPLEMENTATION IN PASCAL.

Consider a stack with pointer TOP. This can be defined by:

```
TYPE
    PTR = ↑ NODE;
    NODE = RECORD
              DATA : datatype;
              NEXT : PTR
           END;
VAR
    TOP, P : PTR;
```

To load the stack (no overflow check) we use:

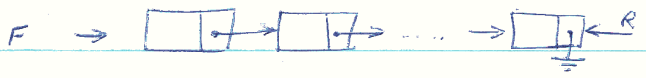
```
NEW (P);
P↑.DATA := whatever;
P↑.NEXT := TOP;
TOP := P;
```

To unload the stack we use:

```
IF TOP = NIL THEN GOTO EMPTYSTACK
   ELSE BEGIN
           whatever := TOP↑.DATA;
           TOP := TOP↑.NEXT;
        END;
```

2.2) QUEUES.

Consider a queue structure as follows:



We need to remove from the F side and insert from the R side. If F=R the queue is empty.

2.2.1) IMPLEMENTATION

This time we declare

```
VAR
    F, T, R : PTR;
```

To insert at rear, we do:

```
NEW (T)
T↑.DATA := whatever;
T↑.NEXT := NIL;
IF F = NIL THEN F := T          (create from an empty q)
           ELSE R↑.NEXT := T;   (q already exists)
R := T;
```

To remove from front:

```
IF F = NIL THEN GOTO EMPTYQUEUE
           ELSE BEGIN
                whatever := F↑.DATA;
                F := F↑.NEXT;
                IF F = NIL THEN R := NIL;   (empty queue)
                END; (* else *)
```

2.3) DEQUES.

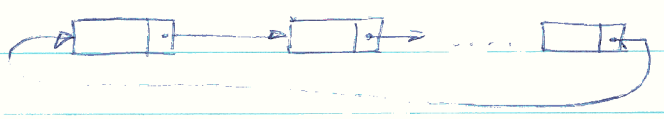
Those are similar to queues, but we need to add two operations:

- inserting at front
- removing from rear

2.3.1) IMPLEMENTATION

2.4) CIRCULARLY LINKED LISTS.

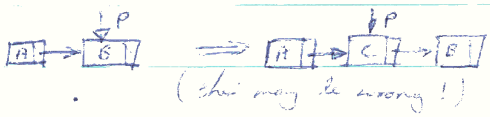
We have the following situation :



In this situation, only one pointer P is needed. If P is NIL, the list is empty.

2.4.1) IMPLEMENTATION

We declare : VAR T,P : PTR;

Ok, to insert in the list ( (this may be wrong !))

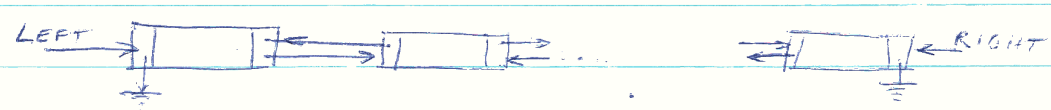
```
NEW (T);
T↑.DATA := whatever;
IF P = NIL THEN BEGIN                (create the list)
                P := T;
                T↑.NEXT := T
              END
         ELSE BEGIN                  (list already exists)
                T↑.NEXT := P↑.NEXT;
                P↑.NEXT := T
              END;
```

To remove from the list: ()

```
IF P <> NIL THEN BEGIN
                T := P↑.NEXT;
                whatever := T↑.DATA;
                P↑.NEXT := T↑.NEXT;
                IF P = T THEN P := NIL;      (list now empty)
              END
         ELSE GOTO EMPTY;
```

2.5) DOUBLY LINKED LISTS

Here we have:



Each node now must have two pointers, LLINK and RLINK.

2.5.1) IMPLEMENTATION

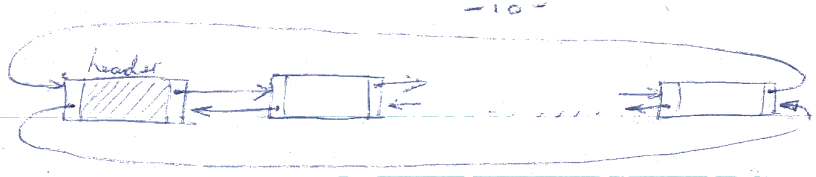
```
VAR  T, LEFT, RIGHT : PTR;
```

To insert on left (to insert on right exchange every occurrence of RIGHT with LEFT and RLINK with LLINK) we have:

```
NEW (T);
T↑.DATA := whatever;
T↑.LLINK := NIL;
T↑.RLINK := LEFT;
IF  LEFT = NIL  THEN  RIGHT := T
                ELSE
                      LEFT↑.LLINK := T
LEFT := T;
```

Etc.

Usually, it is more convenient to have a special node, the header record, and make the list circular, as follows:



The empty list is now represent by the list header's LLINK and RLINK both pointing to the header itself, i.e., LLINK = RLINK.

3. TREES

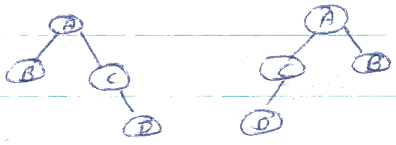
3.1 THEORY

A linear graph is a collection of nodes together with a collection of relationships between the nodes (called lines), which describe the connections between the nodes.

A tree is a linear graph with no closed circuits or loops. Thus any two nodes in a tree are connected by a unique path, and a tree with n nodes has n-1 lines.

A rooted tree is a tree in which one node is given special significance (called the root).

An ordered tree is one in which the order of the subtrees is important. If a tree is not ordered, it is an oriented tree. For example, the two trees shown below are equivalent oriented trees but different ordered trees:



Other term used in describing trees are as follows:

FOREST - a set of disjoint trees

TERMINAL NODE (OR LEAF) - a node with no subtrees

BRANCH NODE - a non-terminal node

DEGREE OF NODE - the number of subtrees of a node.

LEVEL OF NODE - 1 if the node is the root, else
n where n is the number of ~~branches~~ nodes
traversed from the root to the node.

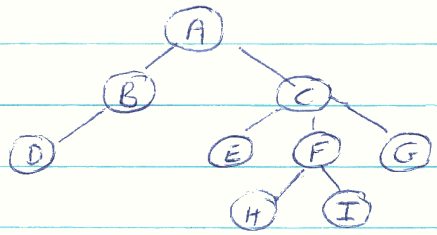
LENGTH BETWEEN NODES - the absolute difference between the
levels of the two nodes.

(STRICTLY) BINARY TREE - a tree in which every node is of degree
0 or 2.

KNUTH BINARY TREE - an binary but for 0, 1, or 2.

3.1.1) TREE TRAVERSAL OF GENERAL TREES.

Consider the tree



Level by level traversals:

- TOP DOWN : A BC DEFG HI
- BOTTOM UP HI DEFG BC A

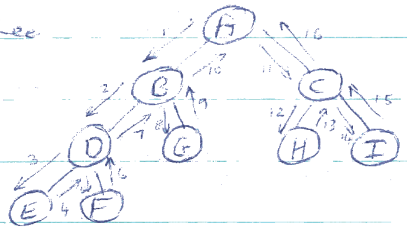
Family order traversals: (see Page & Wilson pg 116)

A B C E F G H I D

FAMILY ORDER - RECURSIVE

- Visit root of first tree
- traverse remaining trees
- traverse subtrees of first tree

3.1.2) BINARY TREE TRAVERSAL (ALSO APPLICABLE TO GENERAL TREES)

Consider the tree 

A walk around the tree is shown by the arrows, the numbering indicating the path.

- PRE-ORDER TRAVERSAL (PREFIX WALK). Start at root. From each branch node go to left subnode. From each leaf go to ~~root~~ right subnode of same subtree - if this does not exist, move up tree until it does. This is the same as walking around the tree and recording the node value on the first visit
  eg. (A) → B → D → E ... F ... G ... C → H ... I

- END ORDER TRAVERSAL (SUFFIX WALK) (POST ORDER) - walk around the tree, and record the node value only on the last visit to that node
  eg E F D G B H I C A

- IN ORDER TRAVERSAL - walk around the tree, and record the node value only when you pass under that node or the node is a terminal node.
  eg E D F B G A H C I

Prefix walks give Polish notation while suffix walks give Reverse Polish notation if the binary tree represents an algebraic expression

3.1.3) RECURSIVE DEFINITIONS OF TRAVERSALS

PRE-ORDER : visit the root and abstract its value.
(Prefix walk) traverse the left subtree, if it exists, in preorder
Polish traverse the right subtree, if it exists, in preorder.

IN-ORDER : traverse the left subtree, if it exists, in inorder
visit the root and abstract its value
traverse the right subtree, if it exists, in inorder.

POST ORDER : traverse the left subtree, if it exists, in post order
(suffix walk) traverse the right subtree, if it exists, in post order.
Reverse Polish visit the root and abstract its value.

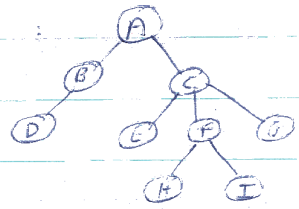
As can be seen, these definitions are basically equivalent except for when the root is visited - pre-order (first), post order (last), in order (middle).

3.1.4) TRANSFORMATION OF A GENERAL TREE INTO A STRICTLY BINARY TREE

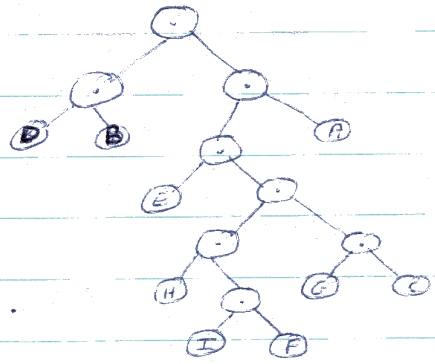
In this transformation, the nodes of the original tree are transformed into the terminal node of the resulting strictly binary tree. If the original tree has n nodes, the resulting tree has (2n-1) nodes. The method is as follows:

(1) If the tree is a single node, the binary tree is just the root
(2) If the tree is not a single node, cut the branch between the root and the leftmost subnode (eldest son). Make the subtree with the eldest son the left subtree, and the one with the root the right subtree.
(3) Recursively repeat for the subtrees.

For example :



becomes



3.2) IMPLEMENTATION IN PASCAL

The most common method of storing binary trees is as a doubly linked list. Each record contains the data for the node plus a pointer to the left and right subtrees. Thus we can declare :

```
TYPE
    PTR = ↑ NODE ;
    NODE = RECORD
            DATA : datatype ;
            LLINK, RLINK : PTR ;
           END;
```

The variables we require are then general pointers, as well as a pointer ROOT pointing to the root. To set up a tree, a possible method is that given by Jensen & Wirth. This is a simple recursive procedure which converts the input string to a tree (assuming that the nodes contain alphabetic data, and '.' represents a nonexistent node'). The tree should then be entered in pre-order. A leaf should be

followed by . . to indicate it has no subtrees. OK.
procedure is :

```
PROCEDURE enter (VAR p : PTR);
  BEGIN
      read (ch);
      IF ch <> '.' THEN BEGIN
            new (p);
            p↑.DATA := ch;
            enter (p↑.LLINK);
            enter (p↑.RLINK);
          END
      ELSE p := NIL;
  END.
```

The procedure should be called by :

```
READLN;
enter (ROOT);
```

A preorder traversal routine is :

```
PROCEDURE preorder (VAR p : PTR);
  BEGIN
      IF p <> NIL then begin
            write (p↑.DATA);
            preorder (p↑.LLINK);
            preorder (p↑.RLINK);
          END;
  END.
```

This should be called by : preorder (ROOT);
Routines for inorder and postorder are similar except

for the location of the write statement (see § 3.1.3).

4) GENERAL

4.1) SEQUENTIAL VS. LINKED ALLOCATION OF STORAGE

Linked lists have the advantages of easy insertion and deletion (only pointers need be changed - no data needs to be shuffled) and easy combining and separating of structures. The disadvantages are that access is sequential and not random, and that more storage space is required.