# Chapter 4: Data structures

## Data Structure

- A particular organization for computer data (e.g., a list).
- And the allowed operations on the structure (e.g., can only add or remove items from one end of the list).

## Elementary Data Types

- **Bit** (Binary Digit) - Lowest level of every data structure.

- **Single Variable**
  E.g., Fixed Point (Integer)

  Floating Point (Real)

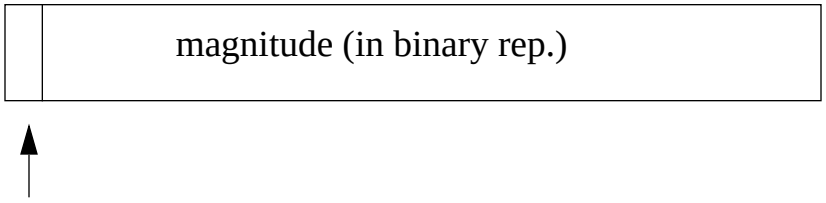
  Character String

  Boolean Variable

  User-Defined Type

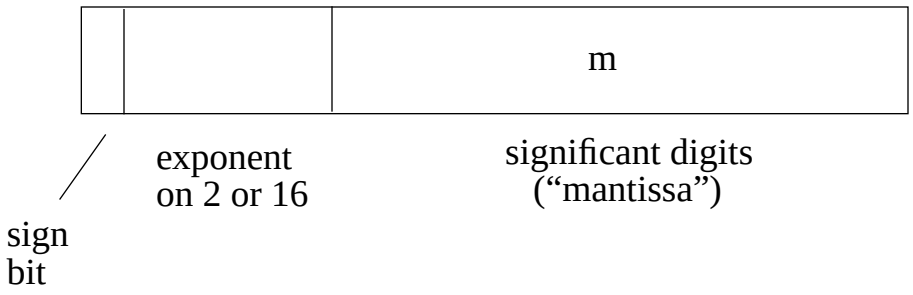
## Basic Data Formats

- **Integer**
  One way to store an integer:




sign bit (0 = +, 1 = -)

- **Real (Floating-Point)**




stored value = $\pm 0.mx2^{\theta}$

- with m normalized so first (leftmost) digit is not zero.

- **Character String**

With bit encoding scheme that satisfies collating sequence:

blank < a < b < . . . < z

etc.

- **Boolean**

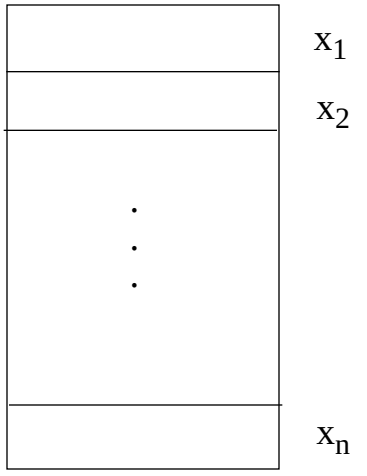
Stored as 1, 0 (True, False)

. . . and similarly for other types

## Higher-Order Data Structures

- **Array**

  Set of data values of one type (such as, integer, real, complex, string, etc.) stored in contiguous storage locations and referenced with a subscript (index).




Use single variable name and subscript to reference individual data items.

- **Record**

  Set of related data of various types stored in contiguous storage locations.

  Examples

  Student Record (name, address, age, sex, major, grade-point average, etc.)

  Equipment Record (part ID, description, manufacturer, price, etc.)

- **File -** Collection of records.

  E.g., Student File, inventory File

## List Structures

- **Linear Lists**

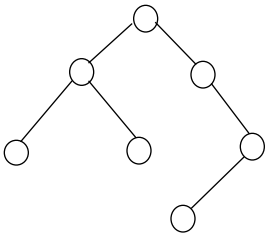
  Set of data with a linear ordering; i.e., each item in list has a single successor.

  E.g., List of names in alphabetical order (Could be stored in array or other available data structure in a particular language.)

- **Nonlinear Lists**

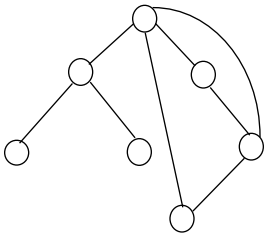
  Set of data items ordered so that any item may have multiple successors.

  E.g., Tree Structures



(such as organizatgional chart or family tree)

Graphs



(such as network representations -- can contain loops and closed paths)

## Basic Operations on Lists

- **Inserting a Data Item**
  (at beginning, at end, or elsewhere)

- **Deleting a Data Item**
  (at various list positions)

- **Sorting**
  (alphabetically, numerically; in ascending or descending order)

- **Searching**
  (for specified data item or set of items, or for those satisfying certain conditions)

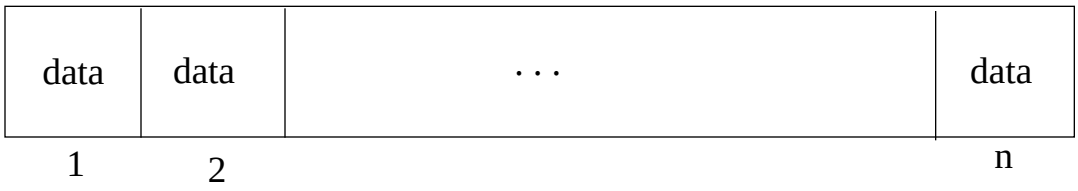
- **Copying**
  (parts of a list to another list)

- **Combining**
  (two or more lists)

- **Separating**
  (a list into sublists)

Can implement lists with:

- **Sequential Storage (Arrays) start**




Then reference data positions with array name and the appropriate subscript number: 1, 2, . . . n.

- **Linked Locations (Pointers)**


Each block of storage, containing a data part and a link part, is called a node. The link fields of each node contain the memory address of the next node in the list, and *start* contains the memory address of the first node in the list.

In this schema, the link fields and start are special types of variables called pointer variables. They can only store a memory address.

---

(Have pointers in C and Pascal, for example, but not in FORTRAN. All three languages have array structures.)

**Sequential Storage is Preferred When**:

- No data insertions or deletions are to be made to the list.
- No splitting or combining of lists is to be made.
- Direct access is required. (For example the list is to be frequently sorted on different fields, such as name field or ID number. Also fast search procedures can be implemented using direct access to the various data items.)

**Linked storage Locations are Preferred When**:

- Data insertions and deletions are required.
- Frequent combining of lists is necessary.
- The list is to be frequently divided into sublists.
- We need to store and manipulate “sparse” lists.

  (An example of a sparse list is a table with most of the entries having a common value, say 0. We can save space by storing only the nonzero values and their locations.)

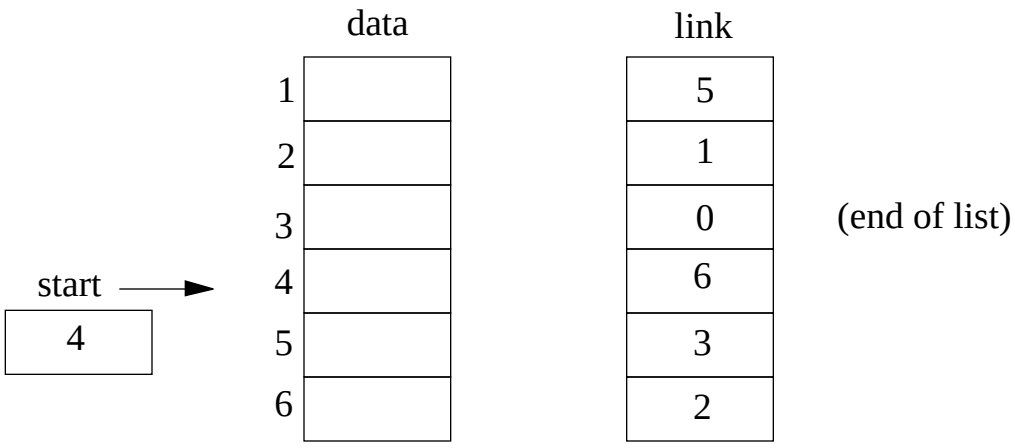
**Linked Storage:**

Requires extra storage for the pointers.

Does not provide direct access to data items.

## Array Implementation of Linked Lists

For programming languages that do not provide pointer variables, we can implement linked-list processing using two or more arrays. For example, using a data array, a link array, and a variable called start, we can store a set of data items and associated “links” as:




To insert or delete data items, we now only need to reset pointer values. Otherwise, with single array storage, we would need to shift data items around to insert or delete values.

For instance, if we want to delete the first node in the list (at position 4), we change the value of start to the value stored in link position 4 -- which is 6.

To handle these operations, we need to set up a program with subparts (modules) to process insertions and deletions, to keep track of available positions in the list, etc.

## Arrays

A common data structure provided in high-level programming languages.

- **One-Dimensional Arrays** (Vectors)

  - array elements referenced with a single subscript, e.g.,

  $$V_k \qquad k = 1, 2, \ldots$$

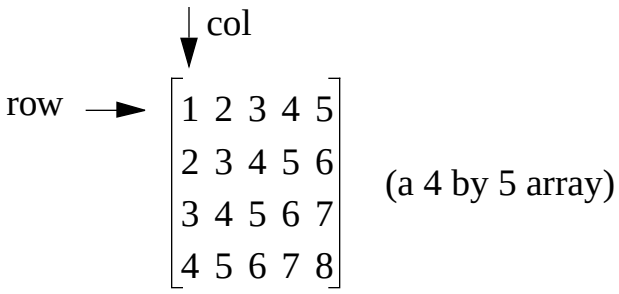
  Often, programming languages allow starting and stopping values to be specified as any integer values.

- **Two-Dimensional Arrays** (Matrices)

  - also sometimes referred to as Tab les.

  - array elements referenced with two subscripts; e.g.,

  $M_{ij}$ where $i, j$ reference row and column positions, respectively




(a 4 by 5 array)

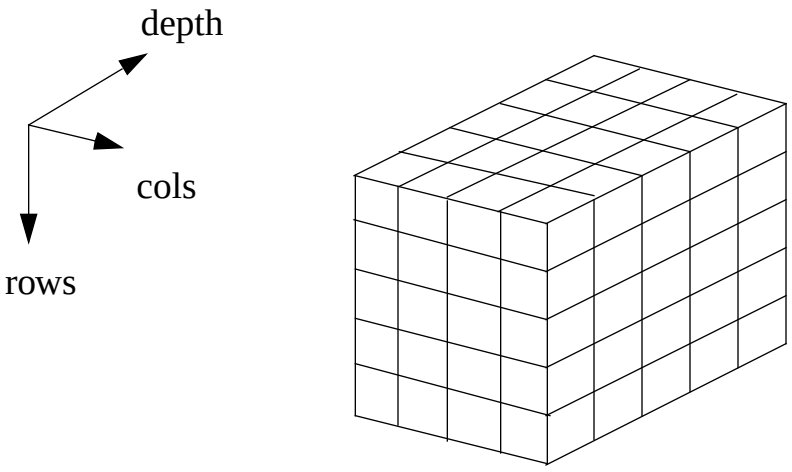
- **Higher-Dimensional Arrays**

  - In general, called Tensors (although, mathematically, the term "tensor" refers to arrays with certain transformation properties).

  - array elements referenced with multiple subscripts; e.g., a 3D array:

$$T_{ijk}$$

  Can represent a 3D array as data items stored within a 3D block:




Visually representing arrays with 4 or more subscripts more difficult.

Also, access time increases as the dimension of an array increases, since more calculation is required to locate element positions.

## List Structures in Mathematica

As in high-level programming languages, Mathematica provides mechanisms for creating and manipulating list structures.

### Creating Lists

One way to create a simple linear list (vector) is to explicitly specify the list elements; e.g.,

```
list1 = {10, 20, 30}
{10, 20, 30}

list2={“Curly”, “Moe”, “Larry”)
{Curly, Moe, Larry}

list3 = {a+Sin[x], a+2Sin[2x]
{a + Sin[x], a + 2 Sin[2 x]
```

Many Mathematica functions produce lists. E.g.,

```
Solve[x^2 == 9, x]
{{x -> 3}, {x -> -3}}
```

Each element of this list is a one-element list. For simultaneous equations, elements will be lists of length 2 or more, depending on number of variables.

Linear lists (vectors) can also be generated in Mathematica with the Range and Table functions.

## Range Function

Generates a list of equality spaced numbers. The general form is:

| Range [*nstart, nstop, dn*] |
|---|

- creates a list of numbers with spacing *dn*, starting with the number *nstart* and ending with a value less than or equal to *nstop*, Parameters *nstart* and *dn* are optional, with default values of 1.

Integer Examples:

```
Range [5]
{1, 2, 3, 4, 5}

Range [-2, 5]
{-2, -1, 0, 1, 2, 3, 4, 5}

Range [-2, 5, 3]
{-2. 1, 4}
```

We can also generate lists of floating-point numbers:

```
Range [2.5, 6.7, 1.2]
{2.5, 3.7, 4.9, 6.1}
```

And we can evaluate the numbers in the list with expressions:

```
x=2; Range[x^2, x^3, x]
{4, 6, 8}
```

## Table Function

A general list-generating function that can be used to produce arbitrary list elements (any data type or expression). General form is:

Table [*expr*,{*k, kstart, kstop, dk*}]

- generates a list of elements by evaluating *expr* with variable *k* varying from *kstart* to *kstop* in steps of *dk*. Default values for *kstart* and *dk* are 1. If *k, kstart*, and *dk* are all omitted, then *expr* is simply listed *kstop* times.

**Table Function Examples**:

```
Table [a^2, {4}]
{a^2, a^2, a^2, a^2}

Table [a^2, {a,4}]
{1, 4, 9, 16}

Table [x^k+k, {k, 0, 3}]
{1, 1 + x, 2 + x2, 3 + x2}

Table [x^(1/2), {x, 0, 1, 0.25}]
{0, 0.629961, 0.793701, 0.90856, 1.}

Table[Sin[n Pi/3], {n, 0, 6, 2}]
```

$$\left\{0, \frac{Sqrt[3]}{2}, \frac{-Sqrt[3]}{2}, 0\right\}$$

```
N[%]
{0, 0.866025, -0.866025, 0}
```

## List Size

We can determine the number of elements in a list with the `Length` function.

For example:

```
aList = {e, i, e, i, o};
aListSize = Length[aList]
5
```

## Referencing List Elements

Given: **`alphaList={a,b,c,d,e,f}`**

we can reference particular elements using subscript notation, as in

```
element3 = alphaList[[3]]
c
```

or we can use the `Part` function, as in

```
element3 = Part[alphabet, 3]
c
```

We can reference list positions relative to the end of the list using negative subscript values.

For example,

```
element3 = alphaList[[-4]]
c
```

Although we can reference the first and last elements of a list with subscripts, Mathematica provides the functions:

```
element1 = First[alphaList]
a

lastElement = Last[alphabet]
f
```

**List programming Example**

The following code uses the Table function and subscript notation to define a nonrecursive function for calculating the nth Fibonacci number, given an input value n.

Variable **fib** is initialized t the list {1,1, . . . , 1} of length n. Then a Do loop is used to calculate Fibonacci numbers, starting at list position 3.

```
Clear[fibonacci];
fibonacci[] :=                    {This defines a fcn with no args
  Module[{k,n,fib},
          n=1+Input[“Input an integer >0.”];
          fib=Table[1, {n}];
          Do[fib[[k]]=fib[[k-1]]+fib[[k-2]],
             k,3,n}
            ];
          fib[[n]]          (Since Mathematica subscripts are numbered
     ]                       from 1, we must shift the  Fibonacci
                             numbering by . Also, we could include
                             checks on input.)
```

Then, within some program block, we could reference the function as

```
fibNum=fibonacci[]          {returns fib[[n]], for an input value of n
89                             (For the input value: n = 10.)
```

## Set Operations

In Mathematica, a linear list can be treated as a set, so that the order of the elements is ignored. E.g., given `s1 = {b,a,c}; s2 = {f,c,b};`

```
unions1s2 = Union[s1,s2]
{a, b, c, f}        (listed in alphabetical order)

commons1s2 = Intersection[s1,s2]
{b, c}

subtracts1s2 = Complement[s1,s2]
{a}        (overlapping elements of s2 subtracted from s1)
```

## List Operations in Mathematica

- **Inserting List Elements**

```
list1 = {10, 20, 30}
(10, 20, 30)

list1 = Prepend[list1, 25]
{25, 10, 20, 30}

list1 = Append[list1, 5]
{25, 10, 20, 30, 5}

list1=Insert[list1, 0, 3]        (insert value 0
{25, 10, 0, 20, 30, 5}            at position 3)

list1=Insert[list1, 1            (insert value 15
{25, 10, 20, 30, 15, 5}           at position 2
                                  from end of list)
list1=Insert[list1, -1,{{2},{4},{6}}]
{25, -1, 10, 20, -1, 30, 15, -1, 5}
```

(Insertion positions 2,4, and 6 are relative to original list positions.)

- **Deleting-Replacing List Elements**

```
list1 = {25, -1, 10, 20, -1, 30, 15, -1, 5};

list1 = Delete[list1, 6]
{25, -1, 10, 20, -1, 15, -1, 5}

list1 = Delete[list1, {{4}, {8}}]
{25, -1, 10, -1, 15, -1}

list1 = ReplacePart[list1, 0, 3]
{25, -1, 0, -1, 15, -1}
```

(3rd element replaced with 0

```
list1 = ReplacePart[list1,5,-3]
{25, -1, 0, 5, 15, -1}
```

(3rd element from end replaced with 5)

```
list1=ReplacPart[list1,-2,{{4},{5}}]
{25, -1, 0, -2, -2, -1}
```

- **Sorting List Elements**

```
sortedlist1 = Sort[list1]
{-2, -2, -1, -1, 0, 25}
```

To sort and eliminate duplicate elements, we apply the `Union` function. This generates a "set" of data values. E.g.,

```
set1=Union[list1]
{-2, -1, 0, 25}
```

We can also sort lists of character strings or variable names with the Sort function. E.g.,

```
charList = {"d","a","m","f"};
sortedcharList=Sort[charList]
{a, d, f, m}
```

To obtain a list arranged in descending order, we can use the `Greater` ordering function:

```
Sort[charList, Greater]
{m, f, d, a}
```

Another way to sort a list in descending order is to use a combination of `Sort` and `Reverse`:

```
Reverse[Sort[charList] ]
{m, f, d, a}
```

(Other functions are available for rearranging lists.)

## Searching-Testing Lists

We determine whether an element is in a list with the MemberQ function.

E.g.,

```
aList = {e, i, e, i, 0};
elements = MemberQ[aList, e]
True
```

Similarly,

```
elementa = MemberQ[aList,a]
False
```

We determine the number of times “e” appears in the list with

```
counte = Count[aList, e]
2
```

And we determine where an element occurs with the Position function.

E.g.,

```
positione = Position[aList, e]
{{1}, {3}}
```

When an specified value is not in a list, the Position function returns the empty list:

```
positionf = Position[aList, f]
{}
```

- **Extracting Sublists**

Given: **`alphaList={a,b,c,d,e,f}`**

We can form sublists using the Take, Rest and Drop functions:

```
firstHalf = Take[alphaList, 3]
{a, b, c}

lastHalf = Take[alphaList, -3]
{d, e, f}

middle4 = Take[alphaList, {2, 5}]
{b, c, d, e, f}

allButFIrstElem = Rest[alphaList]
{b, c, d, e, f}

last4elements = Drop[alphaList, 2]
{c, d, e, f}

first4elements = Drop[alphaList, -2]
{a, b, c, d}

de1Mid4 = Drop[alphaList, {2, 5}]
{a, f}
```

Using combinations of Take, Drop, and Join operations, we can generate sublists with a variety of arrangements.

- **Partitioning Lists into Sublists**

Given `aList={a,b,c,d,e}`, we can form:

```
newList = Partition[aList,2]
{{a, b}, {c, d}}
```

which gives a new list whose elements are lists of length 2 (a 2 by 2 matrix), and drops the last element “e”.

Thus, the first element of newList is

```
newList[[1]]
{a, b}
```

We can also partition using an offset. For example:

```
offsetNewList=Partition[aList,3,1]
{{a, b, c}, {b, c, d}, {c, d, e}}
```

- **Concatenating Lists**

Any number of lists can be combined with the Join function:

```
catList = Join[{a,b,c},{d,e,f},{g}]
{a, b, c, d, e, f, g}
```

## String Manipulations

-similar to those for lists. Following are some of the string functions available in Mathematica

| | | |
|---|---|---|
| **StringLength** | - | give number of characters |
| **StringJoin** | - | concatenate two or more strings |
| **StringReverse** | - | reverse characters |
| **StringTake** | - | delete specified characters |
| **StringReplace** | - | replace specified characters |
| **StringInsert** | - | insert a substring |
| **StringPosition** | - | find position of a specified substring |
| **Characters** | - | list characters in the string |
| **UpperCaseQ** | - | test for all upper-case characters |
| **Sort** | - | sort a list of strings |

Examples:

```
StringLength[“A string.”]
9

StringReverse[“A string.”]
.gnirts A

StringPosition[“A string.”, “ “]
{{2, 2}}   (gives a list of starting and ending
            substring positions)

StringDrop[“A string.”, {3, 4}]
A ring.   (elements 3 through 4 deleted)

Characters[“A string.”]
{A,  , s, t, r, i, n, g, .}
```

## Numerical LIst Operations in Mathematica

For lists of numerical values, we can perform various arithmetic operations on the list elements. For example, given

```
numList = {10, -20, 30, -40}
```

we can form the following new lists:

```
add5 = numList + 5
{15, -15, 35, -35}

multiply5 = 5 numList
{50, -100, 150, -200}

squarenumlist = numList^2
{100, 400, 900, 1600}

sumE1ements = Apply[Plus,numList]
-20
```

(Similarly, have Apply[Times, . . .] to multiply all elements.)

```
addLists = add5 + multiply5
{65, -125, 185, -24
```

(must have equal length lists

```
multiplyLists = numList  {1,2,3,4}
{10, -40, 90, -160}
```

(must have equal length lists

```
smallestVal = Min[numList]
-40

biggestVal = Max[numList]
30

Positive[numList]
{True, False, True, False}
```

Lists in Mathematica are also used to represent vectors in chosen coordinate reference frames. Thus we can apply vector operations to lists.

Examples:

Given two vectors in 3D Courtesan space

```
v1={x1,y1,z1}, v2={x2,y2,z2}
```

we can perform the following vector operations:

1. Dot Product (Scalar Product)

```
v1otv2 = v1.v2
x1 x2 + y1 y2 + z1 z2
```

Similarly, **v1.v1** gives the magnitude squared of vector **v1** (Pythagorean Theorem).

2. Cross Product (Vector Product)

First need to load Vector Analysis package and select a coordinate representation:

```
<<Calculus‘VectorAnalysis‘
SetCoordinates[Cartesian[x,y,z]]

v1Xv2 = CrossProduct{v1, v2]
{-(y2 z1) + y1 z2, x2 z1 - x1 z2,
    -(x2 y1) + x1 y2)
```

3. Other Vector Functions

Includes Div and Curl, as well as vector operators such as Grad and Laplacian, that operate on scalar functions to produce vectors.

## Column Display of A Linear List

Given: **`abcList = {a, b, c}`**

We can display the list in column form:

```
a
b
c
```

using any of the following statements:

```
Column[abcList]

abcList //ColumnForm

TableForm[abcList]

abcList //TableForm
```

(however, TableForm skips a line between each element of the list)

## Array Function

Allows us to set up "symbolic" array names that can be referenced with a simplified subscript notation -- and allows subscripts to start any specified value. For example:

```
vect = Array[v, 3, 0]
{v[0], v[1], v2]}
```

(defines a 3-element vector, with a starting subscript value of 0)

Then, can assign values to elements as in

```
Do[v[k]=k+1,{k,0,2}];
vect
{1, 2, 3}
```

Note: Cannot reference entire array with symbolic name **`v`**, must use assigned name vect.

## Higher-Dimensional Arrays

- are specified in Mathematica as a list of lists, a list of lists of lists, etc.

Example: Creating a matrix with Table function.

```
matrx = Table[i+j, {i,0,2), {j,0,3}]
{{0, 1, 2}, {1, 2, 3}, {2, 3, 4}}
```

We can display a matrix in tabular form using either of the functions: MatrixForm or Table-Form. For example,

```
matrx //MatrixForm
0   1   2
1   2   3
2   2   4
```

MatrixForm displays each element with the same number of character positions. TableForm displays columns of varying width, in general.

```
Table[n^k, {n,4}, {k,3}] //TableForm
1   1   1
2   4   8
3   9   27
4   16  64
```

(Outer loop varies n from 1 to 4; inner loop varies k from 1 to 3.)

The above table could also be generated by eliminating the inner loop and replacing **n^k** with the list: **{n, n^2, n^3**.

Using **Array** function with higher-dimensional arrays, we can use symbolic array name, simplified subscript notation, and arbitrary starting subscript values. E.g.,

```
Clear[mat, m]; mat = Array[m, {3,3}, 0];
Do[m[j,k]=j+k, {j,0,2}, {k,0,2}]; mat
{{0, 1, 2}, (1, 2, 3), {2, 3, 4}}
```

**Matrix Operations**

```
matRow1 = mat[[1]]
{0, 1, 2}
```
(each row of a matrix is a linear list)

```
matRow1 = Part[mat, 1]
{0, 1, 2}

matElement00 = m[0,0] (or mat[[1,1]])
0

matRow2Diag = DiagonalMatrix[mat[[2]] ]
{{1, 0, 0}, {0, 2, 0}, {0, 0, 3}}
```
(i.e., creates a diagonal matrix using second row of mat)

```
MatrixForm[%]
1   0   0
0   2   0
0   0   3

Dimensions[matRow2DDiag]
{3,3}

Ident=IdentityMatrix[3] //MatrixForm
1   0   0
0   1   0
0   0   1
```

Given: **m1={{a,b},{c,d}} m2={{e,f},{g,h}**
**v={x,y}**

**detm1 = Det[m1]**
-(b c) + a d

**transpm1 = Transpose[m1]**

{{a, c}, {b, d}} (Interchange: rows <-> cofs)

**twoTimesm1 = 2m1**

{{2 a, 2 b}, {2 c, 2 d}}

As with vector dot products, matrix products are formed with the “Dot” operator (.):

**m12 = m1.m2**

{{a e + b g, a f + b h}, {c e + d g, c f
+ d h} (Matrix multiplication not cumulative, in general.)

**postTransfMatrixMultiply = v.m1**

{a x + c y, b x + d y}

**preTransfmatrixMultiply = m1.v**

{a x + b y, c x + d y} (Note: v treated as column vector when premultiplied by a “transformation” matrix)

(Also have other functions:
**Inverse, MatrixPower**, etc.)

Two special lists that are useful for processing items in a prescribed order are:

- **Stack**
  Items are inserted and deleted at only one end of the list.

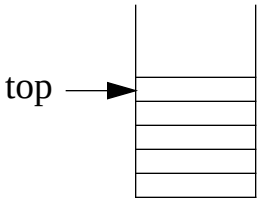
- **Queue**
  Items are inserted at one end of the list, and deleted at the other end.

## Stack Processing

A stack, also called a Last-In-First-Out (LIFO) list, has two operations:

Push (or Stack) - places an element at the top of the list.

Pop (or Unstack) - removes the top element from the list.




Examples:

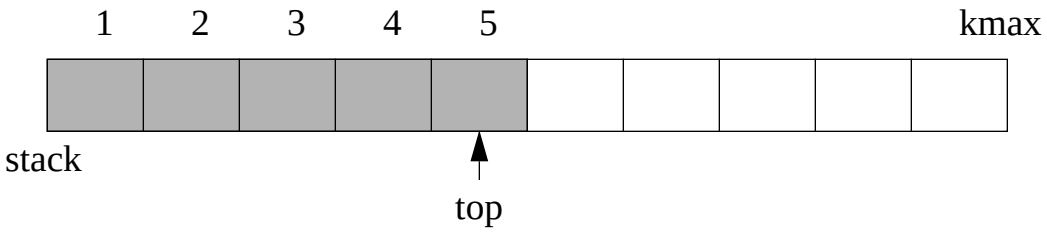
- Stack of boxes, or books
- Stack of trays in a cafeteria
- Spring-loaded coin changer
- Railroad shunting yard

Overflow Condition - triggered by Push operation on a full stack (finite storage).

Underflow Condition - triggered by Pop operation on an empty stack.

In high-level languages, can implement a stack as an array or as a linked list (depending on capabilities of language).

For example, consider an array implementation:




Have empty stack, when top=0.

Have full stack, when top=kmax.

Push Operation

If top<kmax, then

set: top=top+1,

assign input value to position top

else

stack full.

Pop Operation

If top>0, then

assign data at top of stack to output,

set top=top-1

else

stack empty.

In Mathematica, we can implement a stack as a list. And we have enough list operators so we don't need pointer "top".

- First, initialize the stack data structure to the empty list:

```
stack={}
{}
```

- To load items onto the top of the stack, we can use the Append function:

```
stack=Append[stack,item]
```

- To remove (and save) the top item from stack, we can use the Last and Delete functions:

```
If[stack!={},                {check for empty stack
    output=Last[stack];
    stack=Delete[stack,-1],
    emptyStackRoutine        {go here if stack empty
  ]
```

Note that the stack is not implemented as a fixed size array in Mathematica, but as a variable sized (linked) list.

Have overflow, when memory is used up.

## An Example Stack-Processing Program

Input two lists: a data list and an instruction-code list. Items in the data list are to be processed through a stack according to the instruction codes: “s” means stack the corresponding data item; “o” means send the item directly to an output list. After all data items are processed, unload any items in the stack to the output list.

```
Clear[stackProc];
stackProc[data_List,instr_List] :=
  Module[{k,dataLength,stack={},outList={}},
        dataLength=Length[data];
        Do[If[instr[[k]]==”s”,
           stack=Append[stack,data[[k]] ],
           outList=Append[outList,
                          data[[k]] ]
           ], {k,dataLength}
          ];(* Process Codes *)
        stackSize = Length[stack];
        Do[outList=
           Append[outList,stack[[k]] ],
           {k, stackSize, 1, -1}
          ];(* Unload Stack to Outlist *)
         outList
           ]
```

## Example Data Set

```
dataList={a,b,c,d,e};
instrList={“s”,”a”,”o”,”a”,”o”);
outputList=stackProc[dataList,instrCode]
{c, e, d, b, a}
```

---

(In general, stack processing would be accomplished with 3 separate routines; initialize, push, pop.}

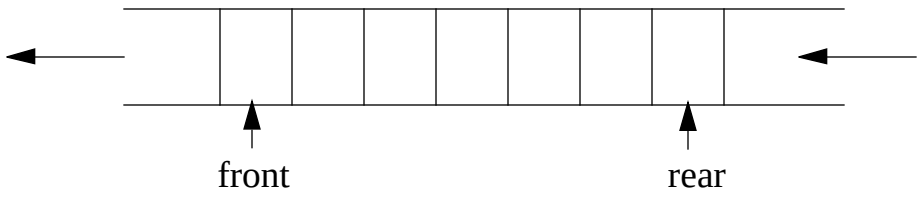
## Queue Processing

A queue, also called a First-In-First-Out (FIFO) list, has two operations:

Load - an item onto the rear of the list.

Unload - an item from the front of the list.

Need two pointers to mark front and rear:




Examples:

A waiting line with no priority cut-ins and no exits from middle of line.

Items moving along an assembly line or through a manufacturing process.

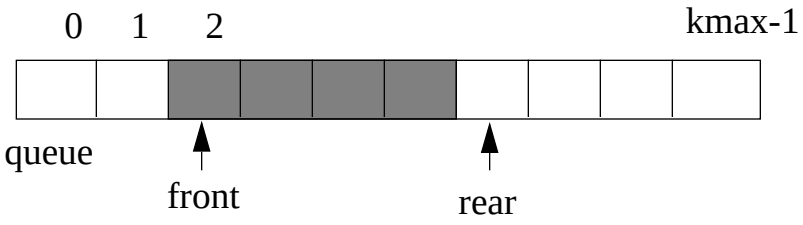
Traffic along a one-way, one-lane street, with no side exits.

Again:

Have overflow condition when attempting to load an item onto a full queue.

Have underflow condition when attempting to remove an item from an empty queue.

Array implementations of a queue are conveniently set up with array positions labeled from 0 to kmax-1, and with pointers referencing the positions shown below:

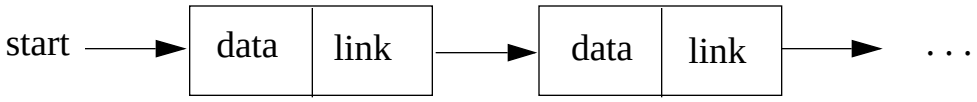



Items are loaded onto the queue at position “rear”, then the rear pointer is incremented by 1.

Items are unloaded at position “front”, then the front pointer is incremented by 1.

Since rear and front pointers may reach position kmax-1, the queue is implemented as a circular list. That is, if rear=kmax-1, the next position for the rear pointer is 0.

Linked list implementations are not restricted by finite list size (only by size of available memory).




Items are loaded into queue by affixing a new node at the end of the list. Items are deleted at front of queue by changing the address of the start pointer.

## Queue Implementation in Mathemataica

As with stack processing, we can handle queue processing without the need for list pointers.

Initialize:

```
queue = {}
{}
```

Load items onto rear of queue:

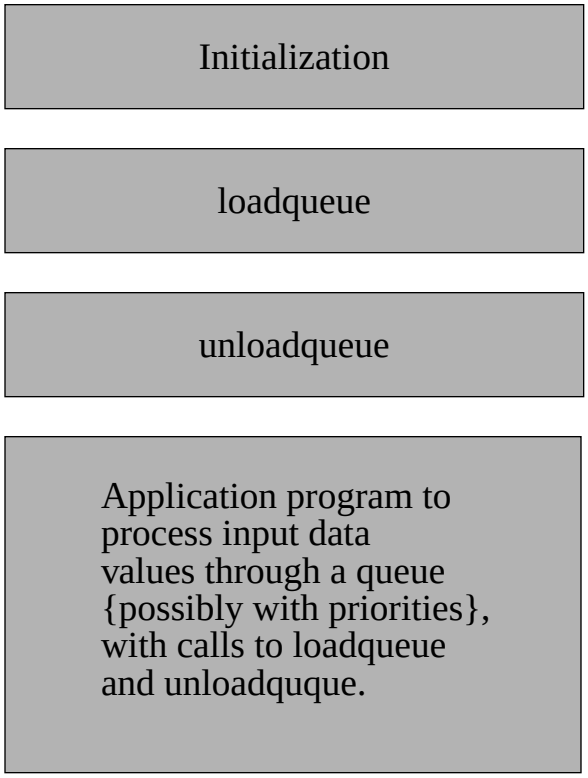
```
queue=Append[queue,item1]
{item1}
queue=Append[queue.item2]
{item1, item2}
```

Unload an item from front of queue:

```
If[queue!={},
    output=First[queue];
          queue=Rest[queue],
    queueEmptyRoutine
   ]
{item}
```

For this example, the queue now contains only item2 and the output is item1.

As with stack processing, queue operations can be modularized with 3 routines: Initialization, loadqueue and unloadqueue:



## Computer Programming Concepts

There are several methods for setting up programs in a high-level language, depending on the structure of the language.

- Procedural Programming

  - arrange program as a series of independent “blocks” of code.

- Functional Programming

  - statements are functions or nested sequence of functions.

- Rule-Based Programming

  - statements set up as pattern-matching constructs.

## Procedural Programming

- writing code sequences that contain a series of operations, such as assignment statements, loops, and conditionals.

We have already done a good bit of procedural programming examples in Mathematica:

Do, For, While

If

Modules

etc.

## Educational Programming

Statements are functions or nested functions. No assignment statements, expressions, etc.

Examples:

(1) Assignment as a function.

**`(set x a)`** - assign to x the value of a

**`(set lst '(a b c))`** - assign list (a,b,c) to lst

(2) Arithmetic operations.

```
(add a b)
(sub a b)
```

etc.

(3) List operations.

```
(append lst '(d e))
```

(4) Loops.

**`(Loop n f)`** - n iterations of f

(5) Nested functions (allow modules to be written as "one-liners").

```
(loop 3 (set a (add 1 (sub a b))))
```

(Above examples are based on Lisp language - which is the model for many Mathematica functions.)

## Functional Programming in Mathematica

- **Assignment Function**

```
set[a,3]
3
```
(assigns value 3 to variable a; must have variables as first arg.)

- **Arithmetic Functions**

```
- Plus, Subtract, Divide, Times, Power
```

Examples:

```
Plus[3,4]
7
```
(Plus and Times can have 3 or more args.)

```
Times[3,4]
12
```

```
Power[3,4]
81
```

- **Arithmetic List Operations**

- can be performed with above functions. E.g.,

```
Plus[{2,3},2]
{4,5}
```

```
Subtract[{2,3},2]
{0, 1}
```

```
Power[{2,3},2]
{4, 9}
```

**Other Mathematica functions include:**

Map[*fcn, list]* - maps *fcn* onto each separate element of *list*

Apply[*fcn, list*] - apply elements of *list* as arguments of *fcn*

Function[*arg, body*] - a 'pure' function (one with no name)

Function[*arg1, arg2, . . . , body*] - multiple arg. pure *fcn*

Examples:

```
Map[Sqrt,{2,3}]
{Sqrt[2], Sqrt[3]}
```

{obtain numerical values with Map[N, Map[Sqrt, [2,3]]], or use //N or N[%]

```
Apply[f,{a,b}]
f[a, b]

Apply[Plus,{2,3}]
5

Map[Function[x, x^3],{2,3}]
{8, 27}

Map[Function[x, x^3],{a,b}]
{{a - b}^3}
```

---

For some applications, functional programming can make code more efficient and easier to read.

Functional programming is also more of a mathematical approach to coding algorithms.