Data Structure

- Formally define Data structure as:

- DS describes not only set of objects but the ways
  - they are related,
  - the set of operations which may be applied to the elements of data object.

- Example: INTEGER data structure consists of
  - set of values
  - operations like +, -, *, div, mod etc.
The study of data structure involves
- Understanding of the structural relationships present within data
- Technique for representing and manipulating such structures within a computer.

Alternatively, one can think DS as a combination of abstract (logical) structure and storage structure

\[ DS = (\text{Abstract} + \text{Storage}) \text{ structures} \]
Cont...

- **Abstract Structure**: deals with mathematical aspect of data with all possible operations to be applied.

- **Storage Structure**: deals with representation of data within computer storage.

- Example: DS for WHOLE NUMBER (WN)
  - **Abstract**: WN = \{0,1,2,…\}
    - Operations: test for zero, addition, test for equality, successor
  - **Storage for WN** in the form of binary digits
Cont…

- Data structures and algorithms are central to the development of good quality computer programs.
- Their role is brought out clearly in the following diagram.

<table>
<thead>
<tr>
<th>Mathematical Model</th>
<th>→ Abstract Dat Type</th>
<th>→ Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal Algorithm</td>
<td>→ Pseudo Language Program</td>
<td>→ Program in C or Java or ......</td>
</tr>
</tbody>
</table>
Abstract data Type - ADT

- ADT is a set of elements with a collection of well defined operations.
- The operations can take as operands not only instances of the ADT but other types of operands or instances of other ADTs.
- Similarly results need not be instances of the ADT.
- At least one operand or the result is of the ADT type in question.
Data Structures: An implementation of an ADT is:
- a translation into statements of a programming language,
- the declarations that define a variable to be of that ADT type and
- the operations defined on the ADT (using procedures of the programming language)

Object-oriented languages such as C++ and Java provide explicit support for expressing ADTs by means of classes.

Examples of ADTs include list, stack, queue, set, tree, graph, etc.
Data Type

- **Data Type.** Data type of a variable is the set of values that the variable may take.
- Basic data types in C / C++:
  - int char float double
- Data object is a term referring to a set of elements such as:
  - integer = { .., -2, -1, 0, 1, 2, ..}
Each programming language has its own basic data types.

Data structures are built up from the basic data types of the underlying programming language using the available data structuring facilities, such as

- arrays,
- records (structures in C),
- pointers,
- files,
- sets, etc.
STACK

Example: A "STACK" is an ADT which can be defined as:
- a sequence of elements and
- operations: null(S), push(S, x), pop(S), top(S)

This can be implemented using data structures such as
- array
- singly linked list
- doubly linked list
- circular array
Overall Data Structures

Data Structure

Primitive
- Integer
- Real
- Boolean
- Char
- String
- Pointer

Non Primitive
- Linear
  - array
  - list
  - stack
  - queue
- Non linear
  - tree
  - graph
  - sets
Primitive data structure

- Integer, real, char, boolean are data types directly handled by system or operated upon by machine level instructions.
- These are also called Data Types.
- String, pointers are data types handled by a programming languages.
Non Primitive DS

- It can be considered to consist of a structured set of primitive DS.

- **Linear DS**: It basically satisfies the property of linearity i.e., storage structure is sequential or contiguous.

- **Non linear DS**: Does not satisfy the property of sequentiality.
  - Ex: tree, graphs, sets etc.
Arrays

- It is an ordered set of items with fixed number of elements.
- In computer programming, an array is also known as a vector (for one-dimensional arrays) or a matrix (for two-dimensional arrays).
- Arrays elements are usually of the same size and data type.
  - Individual elements are accessed by their position in the array.
  - The position is given by an index, which is also called a subscript.
The index usually uses a consecutive range of integers.

Most programming languages have arrays as a built-in data type.

Any array component can be efficiently accessed (inspected or updated) using its index, in O(1) time.

Each array component has a fixed and unique index.
The indices range from a **lower bound** to an **upper bound**.

An array \( A \) with lower bound as -5 and upper bound as 10 can be declared as \( A[-5..10] \). (Pascal)

In most programming languages, the first array index is fixed as 0 or 1.

An array \( A \) with \( n \) items declared as \( A[10] \) is accessed as

- \( A(1), A(2), \ldots, A(n) \) or
- \( A(0), A(1), \ldots, A(n-1) \) depending of start index.
Cont…

- Array size can not be
  - Decreased (No addition)
  - Increased (No deletion)
- Possible operations are
  - Retrieve the value
  - Update the value
- Mathematically, array is defined as set of
  - pairs <index, value>.
  - For each index, there is a associated value.
  - Array is given a name.
Representation of One-dim Array

- Mapping of one-dimensional array into memory is obvious.
- Memory is logically viewed as one-dimensional array (very large).
  - Address of A(i) = $\alpha + (i-1)$.
  - No control over values stored at $(\alpha-1)$ or $(\alpha-n)$

<table>
<thead>
<tr>
<th>Memory</th>
<th>A(1)</th>
<th>A(2)</th>
<th>A(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\alpha$</td>
<td>$\alpha + 1$</td>
<td>$\alpha + (n-1)$</td>
</tr>
</tbody>
</table>
Two-dimensional Array

- Two-dimensional array is represented by
  - pair of indices and value \<(i,j), \text{value}\>.

- A few common representations include:
  - Row-major & Column-major representations
Row-major

- The elements of each row are stored in order.
- Example:
  - Consider two dimensional array declared as A[3,2] (3 rows and 2 columns) whose start index starts with 1.
  - The elements are referred as A(1,1), A(1,2), A(2,1), A(2,2), A(3,1), A(3,2)
  - Or we can write as A(i, j), i= 1,3; j=1,2
  - Here i refers to row number and j as column number.
  - Address of A(i, j) = α + (i-1) * u₁ + (j-1).
    - Here α is start address and u₁ is upper bound of first dimension.
Column-major

- The elements of each column are stored in order.
- Elements of an array $A[3,2]$ are stored as:
  - $A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2)$
- Or we can write as $A(i, j), j=1,3; i=1,2$.
- Address of $A(i, j) = \alpha + (j-1)u_2 + (i-1)$.
  - Here $u_2$ is upper bound of second dimension.
Multidimensional Array

- Multidimensional array is array of **arrays**.
- Multi-dimensional arrays are typically represented by one-dimensional arrays of references to other one-dimensional arrays.
- Array of arrays are also of value in programming languages that only supply one-dimensional arrays as primitives.
The subarrays can be either the rows or columns.
Address of cell in Multi-dim Array

- Let n-dim array is declared as
  \( A[u_1,u_2, \ldots ,u_n] \)
- Assume \( \alpha \) to be start address.
- Address of \( A(i_1,i_2,\ldots i_n) \) is:
  \[
  = \alpha + \sum (i_j - 1) a_j , \quad j = 1,\ldots n, \text{ where}
  \]
  \[
  a_j = \begin{cases} 
  \prod u_k , & \text{for } k = j+1,\ldots,n-1 \\
  1 , & j = n 
  \end{cases}
  \]
Ordered List

- It is an ordered set consisting of a variable number of elements. Size of the list can grow or shrink.
  - List is either empty or can be written as
    \[(a_1, a_2, \ldots, a_n), a_i \in S \quad (\text{some set})\]
  - The list is a flexible abstract data type.
  - It is useful when the number of elements are not known before running a program or can change during running.
  - It is well suited to sequential processing.
  - The next element of a list is readily accessible from the current element.
  - It is less suitable for random access to elements.
List Operations

- Possible operations are: (for $i = 1,..,n$), where, $n= size$
  - Retrieve $i^{th}$ element,
  - Store a new value at $i^{th}$ position,
  - Shifting of data
    - Insert a new value at $i^{th}$ position $\rightarrow$ size = $n+1$
    - Delete element at $i^{th}$ position $\rightarrow$ size = $n-1$
The List Abstract Data Type

- **Type**: list $L = \text{nil} \mid \text{cons } e \ast \text{list } L$
  - Here $L$ is type of list elements, $e$ is an element to be inserted.

- **Operations**:
  - `null : list L -> boolean`
  - `head : list L -> L`
  - `tail : list L -> list L`
Application

- Most common way to implement list is an array, where list element $a_i$ is associated with array index $i$.

Example: Representation of polynomial using list.

- General form of polynomial $P(x)$ of degree $n$ is:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0,$$

where $a_n \neq 0$.
Cont…

- $P(x)$ can be represented by an ordered list as:

$$P = (n, a_n, a_{n-1}, \ldots, a_0)$$

- Coefficients are in decreasing order of exponent.
- This representation is suitable for non-sparse polynomial.
Sparse Polynomial

- Sparse polynomial are those whose most of coefficients are zero.

- Example: Consider the following polynomial

  \[ P(x) = 4x^{100} + 56x^2 + 10 \]

  - One can represent it as follows:
    \[ (100, 100, 4, 2, 56, 0, 10) \]
  
  - Alternatively, store pair of exponent and coefficient for nonzero elements of a polynomial.
Example: If $P(x) = 4x^{100} + 56x^2 + 10$

$P(x) = ((100,4), (2,56), (0,10))$

Use array of structure {exp, coef}

<table>
<thead>
<tr>
<th>exp</th>
<th>coef</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>56</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
Single array for multiple polynomials

- Since polynomials are of unpredictable sizes, one can use global array for representing number of polynomials.
  - Let P, Q and R are polynomials
  - Multiplication of two polynomials
    \[ \text{ps} \quad \text{pe} \quad \text{qs} \quad \text{qe} \quad \text{rs} \quad \text{re} \quad \text{free} \]
    - ‘free’ is an index to location in an array available free for further use.
    - \{ps, qs, rs\} are start and \{pe, qe and re\} are end indices of corresponding polynomials.
Sparse Matrix

- Matrix is said to be sparse if the number of non-zero elements is quite few as compared to the total number of elements.

- Example: Let $M[1000, 1000]$ be a matrix with 0.01% elements nonzero i.e., 100 elements are non-zero.
  - Obviously, 2-dim array representation will be quite inefficient w.r.t. memory requirement.
  - One can use an array of triples (row, col, val).
Let $M[100,200]$ with 4 elements to be non zero. 

$M = ((2,14,204), (2,8,56), (5,2,789), (5,9,123))$

One can also use 2-dim array $A[t+1, t+1]$, where $t =$ number of non zero elements and val is of integer type.

<table>
<thead>
<tr>
<th>Row (=n)</th>
<th>Col (=m)</th>
<th>Val (=t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>204</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>789</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>123</td>
</tr>
</tbody>
</table>
Algorithm 1

for each row in a matrix A

{ take row \((i, j, v)\) from A and store as \((j, i, v)\) in B }

– Here B is transpose matrix of A.
– The rows in B are not in increasing order of 1\(^{st}\) column
– So sort it in increasing order on 1\(^{st}\) column.
– Complexity: \(O(t \log_2 t)\), where \(t\) is number of non-zero elements.
# Representation of sparse matrix

## Matrix A

<table>
<thead>
<tr>
<th>Rows</th>
<th>Cols</th>
<th>Non-zero terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>204</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>789</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>123</td>
</tr>
</tbody>
</table>

## B - Transpose of A

<table>
<thead>
<tr>
<th>Rows</th>
<th>Cols</th>
<th>Non-zero terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>204</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>789</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>123</td>
</tr>
</tbody>
</table>
Representation of Transpose matrix

- **Matrix B**

<table>
<thead>
<tr>
<th>Rows</th>
<th>Cols</th>
<th>Non-zero terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>204</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>789</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>123</td>
</tr>
</tbody>
</table>

- **B sorted on rows**

<table>
<thead>
<tr>
<th>Rows</th>
<th>Cols</th>
<th>Non-zero terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>789</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>123</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>204</td>
</tr>
</tbody>
</table>
Algorithm 2

for $j = 1$ to $m$ (column of $A$)

{ take row $(i, j, v)$ from $A$ and store as $(j, i, v)$ in $B$ }

- Here $B$ is transpose matrix of $A$.
- Complexity: $O(t \times m)$
- In the worst situations, when all elements are nonzero, then complexity is $O(m^2n)$ as $t = n \times m$.
- This is much higher than $O(n \times m)$ in case if matrix is stored using ordinary 2-dim representation with $n$ rows & $m$ columns.
Representation of transpose using Algorithm 2

- **Matrix A**

<table>
<thead>
<tr>
<th>Rows</th>
<th>Cols</th>
<th>Non-zero terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>204</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>789</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>123</td>
</tr>
</tbody>
</table>

- **Transpose of A as B**

<table>
<thead>
<tr>
<th>Rows</th>
<th>Cols</th>
<th>Non-zero terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>789</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>123</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>204</td>
</tr>
</tbody>
</table>
Fast Transpose

Can we design algorithm for transpose using new representation having worst complexity $O(n*m)$? **Answer is : Yes**

- Compute $s[j]$ - Number of rows ‘ $j$ ’ in transpose matrix $B$ of $A$.
- Compute $\text{start}[j]$ - Start index of ‘ $j$ ’ in transpose matrix $B$

\[
\text{start}[j] = \text{start}[j-1] + s[j-1]
\]

- Pick up row from $A$ as $(i, j, v)$ and store it in $B$ as $(j, i, v)$ at $\text{start}[j]$ index
Pseudo-Algorithm for fast Transpose

\[ n = A[1,1]; \ m = A[1,2]; \ t=A[1,3]; \]
\[ B[1,1] = m; \ B[1,2] = n; \ B[1,3]=t; \]
for \( i = 1 \) to \( m \) \{ \( s[j] = 0 \); \}
for \( j = 1 \) to \( t \) \{ \( s[A[j, 2]] = s[A[j, 2]] + 1 \); \}
\[ \text{start}[1] = 2; \]
for \( i = 2 \) to \( m \) \{ \( \text{start}[i] = \text{start}[i-1]+s[i-1] \) \}
for \( i = 2 \) to \( t+1 \)
\[ \{ \]
\( j = \text{start}[A[i,2]] ; \)
\( \text{start}[A[i,2]] = j+1 \)
\[ \}

- Complexity = \( O(t+m) \cong O(nm) \), for \( t = n*m \)