Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

DATA STRUCTURES

Abstract

Introduction to Data Structures and its Applications

Vandana M Ladwani

vandanamd@pes.edu

Contents

Introduction 2
Data Structure Operations 2
Classification of Data Structures 2
Simple Data Structure 2
Compound Data Structure 3
Linear Data Structures 3
Non-Linear Data Structures 3
Applications of Data Structures 4
Array 4
Linked Lists 4
Stacks 4
Queues 4
Trees 4
Heaps 5
Graphs 5

Introduction

Clever" ways to organize information in order to enable efficient computation

Data Structure is a way organizing data in such a way that we can perform operations on these data in an effective way for various applications

DS = Organized Data + Allowed operations

Data Structure Operations

- **Inserting**: Adding a new record to the structure.
- **Deleting**: Removing a record from the structure.
- **Traversing**: Accessing each record exactly once so that certain terms in the record may be proceeded.
- **Searching**: Finding the location of the record with a given key value.
- **Sorting**: Arranging the records either in ascending or descending order according to some key.
- **Merging**: Combining the records in two different sorted files into a single sorted file.
- **Copying**: Records in one file are copied to another file.

Classification of Data Structures

- Simple Data Structure
- Compound Data Structure
    - Linear Data Structure
    - Nonlinear Data Structure

➢ Simple Data Structure

It can be constructed with the help of Primitive Data Structure.
A **primitive data structure** is used to represent the standard data types of a particular programming language. Built in data types, arrays, pointers, structures, unions, etc. are examples of simple data structures.

➢ Compound Data Structure

Compound data structure can be constructed with the help of any one of the primitive data structure. Various operations that can be performed are decided by the designer to cater to the specific requirements of the applications.

It can be classified as

1) Linear Data Structure
2) Non-linear Data Structure

1. Linear Data Structures

In a linear data structure elements are accessed in a sequence.
Some of the operations that can be applied on linear data structure include

- Add an element
- Delete an element
- Traverse
- Sort the list of elements
- Search for a data element

Example: Stack, Queue, Tables, List, and Linked Lists.

2. Non-Linear Data Structures

Non-linear data structure can be constructed as a collection of randomly distributed set of data item. In non-linear Data structure the relationship of adjacency is not maintained between the Data items.
Some of the operations that can be applied on non-linear data structures:

- Add an element
- Delete an element
- Traverse
- Search for a data element

Example: Tree, Decision tree, Graph and Forest

Applications of Data Structures

Array

- Represent/implement other data structures in memory
- Store files in memory

Linked Lists

- Represent/implement other data structures in memory
- Dynamically allocate space in the main memory
- Allocate blocks in hard disk
- Manipulate large numbers

Stacks

- Recursion
- Call stack is used to keep track of program with multiple functions
- Infix to postfix expression conversion
- Evaluate a postfix expression
- Rearranging railroad cars
- Implement undo and redo operation for various applications

Queues

- Job scheduling
- Process scheduling in Operating system
- Handling events in event controlled applications
- Handling of interrupts by the operating system
- Store the browsing history

Trees

- Auto complete features (Trie)
- For easier substring matching
- For metadata indexing in file systems (B+ tree)
- To maintain table indices in relational database systems (B+ tree)
- Store dictionary in a mobile (Trie)

- To check spellings (Trie)
- To construct associative array (Red black trees)
- To ensure direct access of data blocks in file systems (B tree)
- Used by compilers to check the syntax of a statement in a program (Parse Trees)
- Used by operating systems to maintain the structure of a file system

➢ Heaps

- Priority queue Implementation
- Heapsort

➢ Graphs

- Social network analysis
- Shortest path problems
- Disease modelling
- Citation of journals
- Computer Networks

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

MEMORY ALLOCATION

Abstract

Static and Dynamic Memory allocation

Vandana M Ladwani

vandanamd@pes.edu

Contents

Static and Dynamic Memory Allocation 2
Static Memory Allocation 2
Advantages of Static Memory Allocation 2
Disadvantages of Static Memory Allocation 2
Dynamic Memory Allocation 2
Advantages of Dynamic Memory Allocation 3
Disadvantages of Dynamic Memory Allocation 3
Difference between static and Dynamic Memory Allocation 3
SMA(Static Memory Allocation) ..... **Error! Bookmark not defined.**
DMA(Dynamic Memory Allocation) ..... **Error! Bookmark not defined.**
Functions 4
malloc() 4
calloc() 4
realloc() 4
free() 4
Differences between malloc and calloc 5

Static and Dynamic Memory Allocation

Memory can be allocated for variables using two different techniques like static and dynamic memory allocation

Static Memory Allocation

Memory is assigned before the execution of the program begins i.e., during compilation time in the stack region. The compiler allocates the required memory space.

Advantages of Static Memory Allocation

1. Memory is allocated in a more structured area of memory, called the stack region.
2. No need of pointers to access the data
3. Faster execution than Dynamic memory allocation

Disadvantages of Static Memory Allocation

1. The memory is allocated during compilation time. Hence, the memory allocated is fixed and cannot be altered during run time.
2. This leads to under utilization of memory if more memory is allocated
3. This leads to over utilization of memory if less memory is allocated
4. Useful only when the data is fixed and known before processing
5. Memory cannot be deleted explicitly only overwriting takes place
6. If elements are to be added to or deleted from intermediate positions then shifting of elements is required to be done

Dynamic Memory Allocation

Consider a particular application for which we need to change the size of array at run time, if array is allocated statically, extending or shrinking of the memory during run time is not possible. Thus by using static memory allocation, we cannot utilize the memory efficiently instead we can use dynamic memory allocation in this scenario

The process of allocating memory at runtime is known as dynamic memory allocation. Memory is allocated or deallocated during run-time (during the

execution of the program) in the heap region. Library routines defined in **stdlib.h** known as "memory management functions" are used for allocating and releasing also termed as freeing memory during execution of a program. These functions are as follows

malloc(): Allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space

calloc(): Allocates space for an array of elements, initialize them to zero and then return a void pointer to the memory

realloc(): Modifies the size of previously allocated space using above functions

free(): Releases the allocated memory

Advantages of Dynamic Memory Allocation

1. Memory is allocated only when the program unit is active.
2. Memory is utilized efficiently.
3. Allocated memory can be resized during run time based on the dynamic requirement of the user/program.

Disadvantages of Dynamic Memory Allocation

1. Memory is allocated in the heap region which is less structured.
2. Execution is comparatively slower.
3. Pointers are required to work with dynamically allocated memory region.

Difference between static and Dynamic Memory Allocation

Static Memory Allocation)	Dynamic Memory Allocation
The memory is allocated during compile time.	The memory is allocated during run time
The size of the memory to be allocated is fixed during compile time and cannot be altered during run time.	As and when memory is required, memory can be allocated. If memory is not required it can be deallocated.
Memory is allocated in stack area	Memory is allocated in heap area
Used only when the data size is fixed and known in advance before processing	Used for unpredictable memory requirement
Execution is faster, since memory is already allocated and data manipulation is done on these allocated memory locations	Execution is slower since memory has to be allocated during run time. Data manipulation is done only after allocating the memory
Example: Arrays	Example: Linked List

Dynamic Memory Management Functions

malloc()

Prototype: void *malloc(size_t size);
Description: malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then malloc() returns either NULL, or a unique pointer value. The malloc() function returns a void pointer to the allocated memory. On error, function return NULL.

calloc()

Prototype: void *calloc(size_t num_blocks, size_t size);
Description: The calloc() function allocates memory for an array of num_blocks elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If num_blocks or size is 0, then calloc() returns either NULL, or a unique pointer value. The calloc() function return a void pointer to the allocated memory.

realloc()

Prototype: void *realloc(void *ptr, size_t size);
Description: This function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If ptr is NULL, then the call is equivalent to malloc(size) for size>0; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). If the area pointed by ptr gets moved, a free(ptr) is done.The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr, or NULL if the request fails. If realloc() fails the original block is left untouched, and it is not freed or moved.

free()

Prototype: void free(void *ptr);
Description: The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed. The free() function returns no value.

Differences between malloc and calloc

malloc()	calloc()
stands for memory allocation	Stands for contiguous allocation
This function takes single argument. Syntax: void *malloc(size_t size); size: total number of bytes to be allocated	This function takes two arguments Syntax: void *calloc(size_t n, size_t size); n :number of blocks to be allocated. Size: number of bytes to be allocated for each block
Allocates a block of memory of size bytes	Allocates multiple blocks of memory, each block with the same size
Allocated space will be initialized to junk values	Each byte of allocated space is initialized to zero
malloc is faster than calloc.	calloc takes little longer(not recognizable in practical scenario) than malloc because of the extra step of initializing the allocated memory.

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

SINGLY LINKED LIST

Abstract

Singly linked list overview with its advantages and disadvantages. Types of singly linked list and pseudo code for various operations

Vandana M Ladwani
vandanamd@pes.edu

PES UNIVERSITY

Contents

Overview....2
Disadvantages of Array....2
Advantages of linked lists....2
Disadvantages of linked lists....3
Types of Linked Lists....3
Singly Linked List....4
Creating a node....4
Insertion of a Node....5
Inserting a node at front....5
Inserting a node at the end....5
Inserting a node at intermediate position....5
Deletion of a node....6
Deleting a node from front of linked list....6
Deleting a node at the end....6
Deleting a node at Intermediate position....7
Traversal and displaying a list (Left to Right)....7

Singly Linked list

Overview

Linked lists and arrays are similar in the sense that they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy access to elements through index. Once the array is set up, access to any element is convenient and fast.

Disadvantages of Array

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Deleting an element from an array is not possible, deletion just overwrites the content in the memory location that too at the expense of shifting elements

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Array allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked list allocates memory for each element separately and only when necessary. A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the data item/items to which it is linked in the list. The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the linked data item.

Advantages of linked lists

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (released) when it is no longer needed.

3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deleting a data item from the given position.
4. Linked list is used as an important data structure for implementing many complex applications

Disadvantages of linked lists

1. It consumes more space because every node requires an additional pointer to store address of the link node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists

Basically we can put linked lists into the following four items:

1. Singly Linked List.
2. Doubly Linked List.
3. Circular Singly Linked List.
4. Circular Doubly Linked List.

A singly linked list is the one in which all nodes are linked together in sequential manner. Each node holds a single pointer which points to the next item. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by two links which helps in accessing both the successor node (link node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (link). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular doubly linked list is one, which has both the successor pointer and predecessor pointer and last node and first node are connected

Comparison between array and linked list:

ArrayList	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient	Insertions and Deletions are efficient
Elements are in contiguous memory locations	Elements are not in contiguous memory locations
May result in memory wastage if all the allocated space is not used	Since memory is allocated dynamically (as per requirement) there is no wastage of memory.
Sequential and random access is faster	Sequential and random access is slow

Singly Linked List

A linked list allocates space for each element separately in a block of memory called “node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "link" field which is a pointer used to link to the link node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). Head/start is a pointer to the first node in the linked list.

The basic operations in a singly linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing
- Reversing a list
- Concatenating two lists

Creating a node

- Get a node using malloc
- If memory is allocated successfully
    1. Set data part
    2. Set link part

Insertion of a Node

Memory is to be allocated for the new node. The new node will contain empty data field and empty link field. The data field of the new node is set to the value given by the user. The link field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.
- Inserting a node at the beginning:

Inserting a node at front

- Get the new node using createnode().
- new_node = createnode();
- If the list is empty then head = new_node.
- If the list is not empty, follow the steps given below:
  - new_node -> link = head;
  - head = new_node;

Inserting a node at the end

- Get the new node using createnode()
  new_node = createnode();
- If the list is empty then head = new_node.
- If the list is not empty follow the steps given below:
  - temp = head;
  - while(temp -> link != NULL)
    - temp = temp -> link;
    - temp -> link = new_node;

Inserting a node at intermediate position

- Get the new node using createnode().
  new_node = createnode();
- If the position is greater than length of linked list+1, specified position is invalid.
- Store the heading address (which is in head pointer) in temp and prev pointers. Then traverse the temp pointer unto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:
  If position is 1

Insert at front

If position=length of linked list+1

Insert at end

Else

if intermediate position

- prev -> link = new_node;
- new_node -> link = temp;

Deletion of a node

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.
- Deleting a node at the beginning:

Deleting a node from front of linked list

If list is empty then display 'Empty List' message.

- If the list is not empty, follow the steps given below:

```
temp = head;
head = head -> link;
free(temp);
```

Deleting a node at the end

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = prev = head;
while(temp -> link != NULL)
{
 prev = temp;
 temp = temp -> link;
}
prev -> link = NULL;
free(temp);
```

Deleting a node at Intermediate position

- If list is empty then display 'Empty List' message
- If the list is not empty (has atleast two nodes), follow the steps given below.

if(pos > 1 && pos < nodectr)

- temp = prev = head;
- ctr = 1;
- while(ctr < pos)
    - prev = temp;
    - temp = temp -> link;
    - ctr++;
- prev -> link = temp -> link;
- free(temp);

Traversal and displaying a list (Left to Right)

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached.

Traversing a list involves repeating following steps till head becomes NULL

- Assign the address of head pointer to a temp pointer.
- Display the information from the data field of each node
- Advance head pointer

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

CIRCULAR DOUBLY LINKED LIST

Abstract

Circular Double Linked List – overview and pseudo code of various operations

Vandana M Ladwani

vandanamd@pes.edu

Contents

Overview 2

Operations 2

Inserting a node at the beginning 2

Inserting a node at the end 2

Inserting a node at an intermediate position 3

Deleting a node at the beginning 3

Deleting a node at the end 4

Deleting a node at Intermediate position: 4

Overview

A circular doubly linked list has both successor pointer and predecessor pointer in circular manner that is the last and first node are connected. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the rlink link of the right most node points back to the head node and llink link of the first node points to the last node.

The basic operations in a circular double linked list are:

- Creation
- Insertion
- Deletion
- Traversing

Operations

Inserting a node at the beginning

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using createnode().
  new_node=createnode();
- If the list is empty, then
  head = new_node;
  new_node -> llink = head;
  new_node -> rlink = head;
- If the list is not empty, follow the steps given below:
  new_node -> llink = head -> llink;
  new_node -> rlink = head;
  head -> llink -> rlink = new_node;
  head -> llink = new_node;
  head = new_node;

Inserting a node at the end

The following steps are followed to insert a new node at the end of the list:

- Get the new node using createnode()
  new_node=createnode();
- If the list is empty, then
  head = new_node;

new_node -> llink = head;
new_node -> rlink = head;

- If the list is not empty follow the steps given below:
  new_node -> llink = head -> llink;
  new_node -> rlink = head;
  head -> llink -> rlink = new_node;
  head -> llink = new_node;

Inserting a node at an intermediate position

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using createnode().
  new_node=createnode();
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
- Store the heading address (which is in head pointer) in temp. Then traverse the temp pointer upto the specified position.
- After reaching the specified position, follow the steps given below:
  new_node -> llink = temp;
  new_node -> rlink = temp -> rlink;
  temp -> rlink -> llink = new_node;
  temp -> rlink = new_node;
  nodectr++;

Deleting a node at the beginning

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
  temp = head;
  head = head -> rlink; // second node
  temp -> llink -> rlink = head;// temp->llink is last node
  head -> llink = temp -> llink;
  free(temp)

Deleting a node at the end

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below:

```
cur=head->llink          // last node address
cur->llink->rlink=head // cur->llink is second last node
head->llink=cur->llink
free(cur)
```

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contains more than two node).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
- Get the position of the node to delete.
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
- Then perform the following steps:

```
if(pos > 1 && pos < nodectr)
    temp = head;
    i = 1;
    while(i < pos)
        temp = temp -> rlink ;
        i++;
    temp -> rlink -> llink = temp -> llink;
   temp -> llink -> rlink = temp -> rlink;
   free(temp);
   nodectr--;
```

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS202: Data Structures and its Applications (4-0-0-4-4)

SPARSE MATRIX

Abstract

SPARE MATRIX – overview and its representations

Vandana M Ladwani

vandanamd@pes.edu

Contents

Overview 2

Representations 2

Triple notation 2

Linked Representation 3

SPARSE MATRIX

Overview

A matrix is represented as 2 dimensional array where every element is accessed by row and column index. Real world data such as image, spectrogram and graph can be modelled using matrices.
A matrix for which most of values are zero is termed as the sparse matrix. If a sparse matrix is stored in a memory as a two dimensional matrix it wastes lot of space.
So alternate representations are preferred for sparse matrix
Alternate representations are:

- Triple notation
- Linked representation

Representations

1. Triple notation

In triple notation sparse matrix is represented as an array of tuple values. Each tuple consists of
<rowno columnno Value>

The first block in array block holds information regarding
<total no of rows, total no of columns ,value>

Declaration

```
typedef struct
{
        int col;
        int row;
        int value;
} term;
term a[10];
```

Various operations that can be performed on sparse matrix are
Create_SparseMatrix()
Transpose_of_SparseMatrix()
Add_SparseMatrices()

Multiple_SparseMatrices()

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

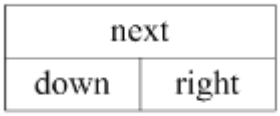
Triple Notation

Row No	Column No	Value
5	4	6
0	0	2
1	0	4
1	3	3
3	0	8
3	3	1
4	2	6

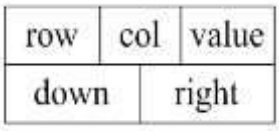
2. Linked Representation

Two types of nodes are used

Header Node

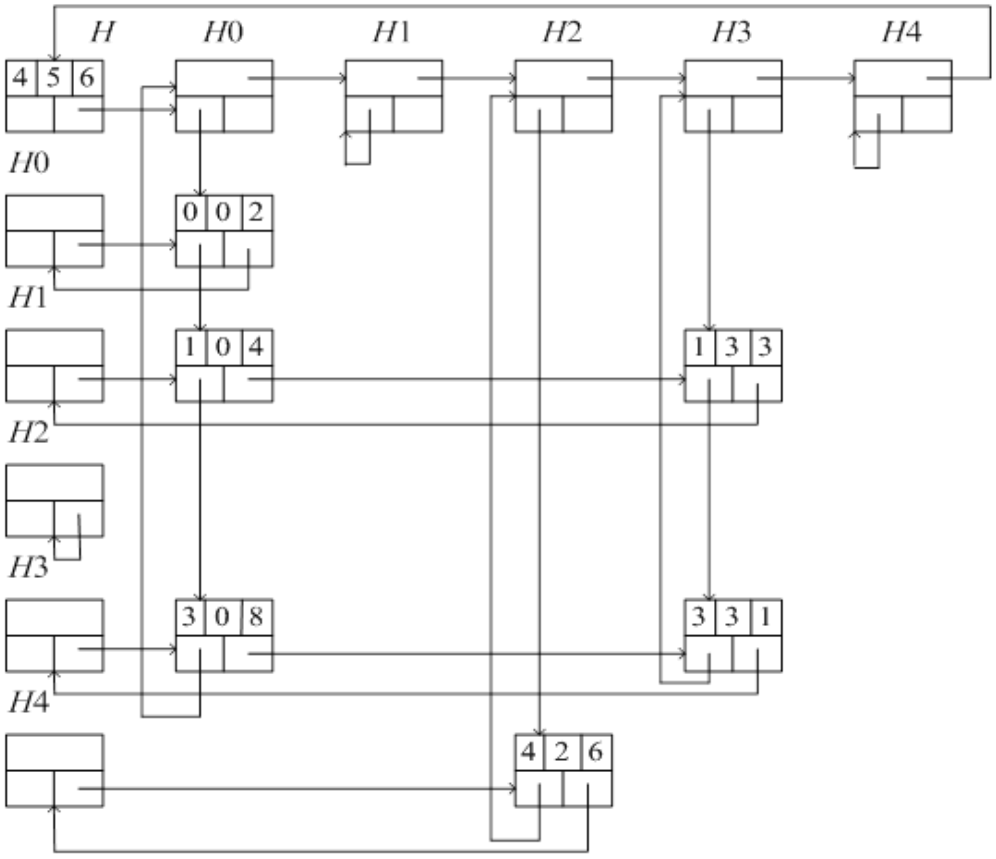


Data Node




```
#define MAX_SIZE 50 /* size of largest matrix */
typedef enum {head, entry} tagfield;
typedef struct matrixNode * matrixPointer;
typedef struct entryNode {
int row;
int col;
int value; };
typedef struct matrixNode {
        matrixPointer down;
        matrixPointer right;
        tagfield tag;
        union
        {
        matrixPointer next;
        entryNode entry;
        } u;
};
```

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$



Sparse Matrix representation using Linked Nodes

Courtesy: “Fundamentals of Data Structures” By Ellis Horowitz and Sartaj Sahni