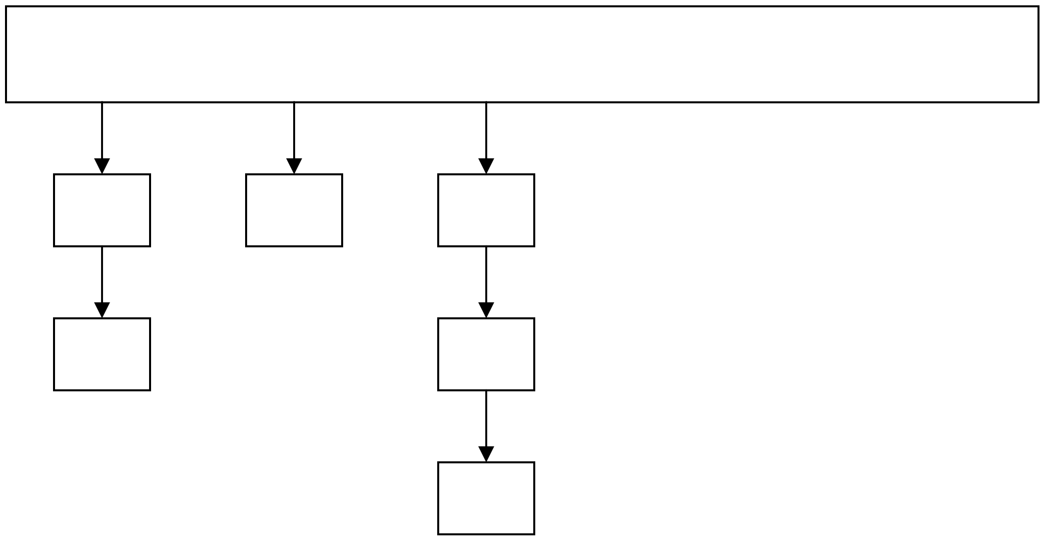
Lecture 11
Array of Linked Lists

In this lecture

- Array of Linked Lists
- Creating an Array of Linked Lists
- Representing a Sparse Matrix
- Defining a Node for Sparse Matrix
- Exercises
- Solutions

An Array of Linked Lists

A linked list is defined as a collection of nodes that can be traversed starting at the head node. It is important to note that head is not a node, rather the address of the first node of the list. Linked lists are very useful in situations where the program needs to manage memory very carefully and a contiguous block of memory is not needed. An array of linked lists is an important data structure that can be used in many applications. Conceptually, an array of linked lists looks as follows.



An array of linked list is an interesting structure as it combines a static structure (an array) and a dynamic structure (linked lists) to form a useful data structure. This type of a structure is appropriate for applications, where say for example, number of categories is known in advance, but how many nodes in each category is not known. For example, we can use an array (of size 26) of linked lists, where each list contains words starting with a specific letter in the alphabet.

The following code can be used to create an array of linked lists as shown in the figure above. Assume that all variables are declared.

```
node* A[n] ;    // defines an array of n node pointers
for (i=0; i<n; i++)  A[i] = NULL;  // initializes the array to NULL
```

Creating an Array of Linked Lists

Suppose that a linked list needs to be created starting at A[i]. The first node can be created as follows.

```
A[i] = (node*)malloc(sizeof(node)); // allocate memory for  node
A[i] → size = 10;
A[i] → name = (char*) malloc(strlen(“guna”+1));
strcpy(A[i] → name, “guna\0”);
A[i] → next = NULL;
```

Now to insert more nodes into the list let us assume that we have a function insertnodes with the following prototype.

```
int inserrnodes(node*** arrayhead, int index, node* ptr);
```

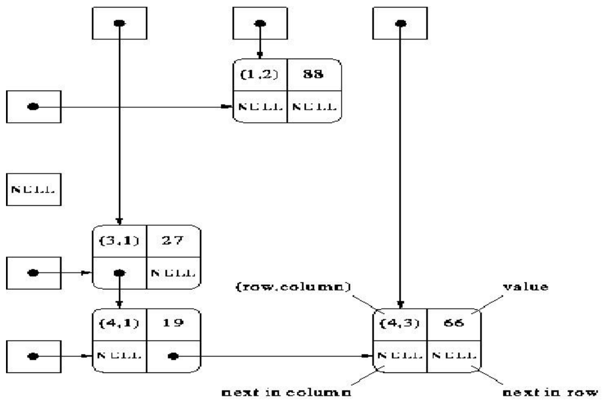
Here we pass the address of the array of node*, a pointer to a node, and the array index.

A call to the function can be as follows.

```
node* ptr = (node*)malloc(sizeof(node));
ptr → size = 10;
ptr → name = (char*) malloc(strlen(“guna”+1));
strcpy(ptr → name, “guna\0”);
ptr → next = NULL;
insertnodes(&A, ptr, 3);   // insert node ptr to array location 3.
```

In lab 4, we will implement a sparse matrix, a matrix where most entries are zero using a structure as follows.

Representing a Sparse Matrix



A suggested data structure to implement the above is given by two structs, node and matrix.

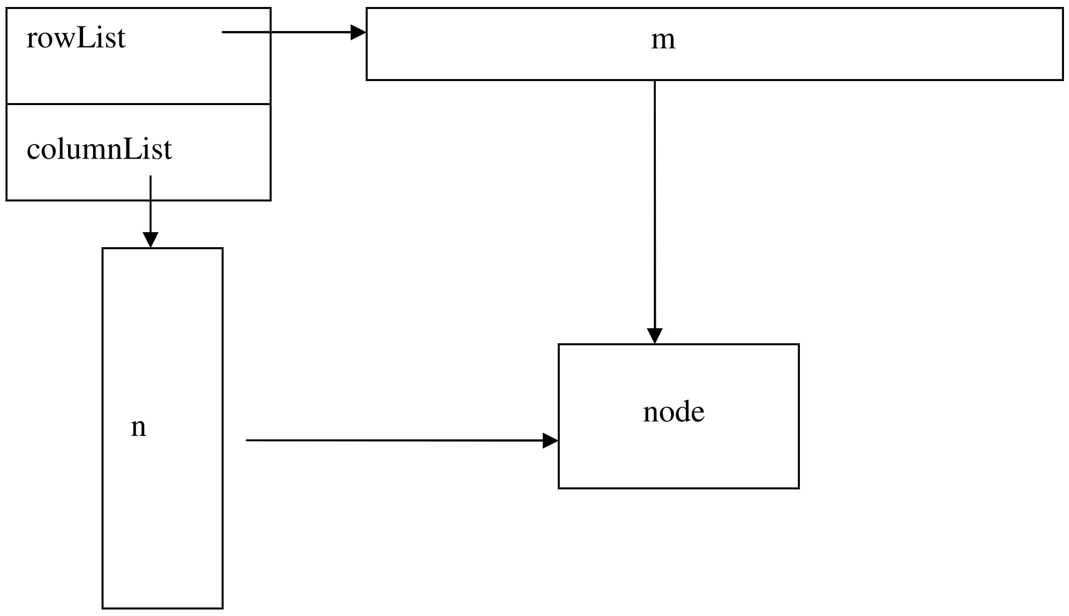
```
typedef struct node {
        int row, column,
        double value;
        struct node*  rowPtr;
        struct node*  colPtr;
} node;
```

The node is a self-referencing structure that can be used to form nodes in a linked list.

The matrix component of the data structure is a struct that contains two arrays of node pointers, each pointing to first element in a row or column. The overall matrix is stored in a structure as follows.

Defining a Sparse Matrix Node

```
typedef struct matrix {
     node**  rowList;     // rowList is a pointer to the array of rows
     node** columnList; // column list is a pointer to the array of columns.
     int  rows, columns;  // store the number of rows and columns of the matrix
} matrix;
```



In the structure above, we are using two arrays of linked nodes to create a structure that can be traversed starting from any row index or column index. Although the above structure seems complicated to implement, once we understand that each linked list is a separate singly linked list, it becomes easy to think about this and implement this structure. When a new node needs to be inserted, we must traverse the list from corresponding row index and corresponding column index and then link the node into the structure. You can use same logic in both cases, except that row indices are traversed and linked using the row_ptr and columns are traversed and linked using the column_ptr.

EXERCISES

1. Suppose M is a matrix*, where matrix is as defined above. Write code to allocate enough space to initialize a matrix of n by m.
2. Given a pointer to a node called ptr (assume all memory is allocated and node initialized), write code to insert the node to the beginning of the each list.
3. Write a function **int duplicatevalue(matrix* M, double value)** that returns 1 if a node with the value exists in the matrix. Return 0 if not.
4. Write a function **int resize(matrix**)** that doubles the rows and columns of the matrix. The old nodes need to be copied to the new matrix. Return 0 if success, 1 if failure.
5. Write a function **int transpose(matrix**)** that takes the transpose of the matrix. Transpose of a matrix M is defined as a matrix M1 where rows of M are equivalent to columns of M1 and columns of M are equivalent to rows of M1. For example the transpose of M = {{1,2},{3,4}} is M1 = {{1,3},{2,4}}

SOLUTIONS

1. Suppose M is a matrix*, where matrix is as defined above. Write code to allocate enough space to initialize a matrix of n by m.

```
matrix* M = malloc(sizeof(matrix));
M->rowList = malloc(n*sizeof(node*));
M->colList = malloc(m*sizeof(node*));
```

2. Given a pointer to a node called ptr (assume all memory is allocated and node initialized), write code to insert the node to the beginning of the each list.

```
for (i=0; i<n;i++) {
  ptr -> next = A[i] ;
  A[i] = ptr;
 }
```

6. Write a function **int duplicatevalue(matrix* M, double value)** that returns 1 if a node with the value exists in the matrix. Return 0 if not.

```
int duplicatevalue(matrix* M, double value) {
      int i=0;
      for (i=0; i<M->rows; i++) {
            node* head = M->rowList[i];
            while (head != NULL)
                { if (head->value == value) return 1;
                  head = head -> next;
                }
      }
      return 0;
 }
```

3. Write a function **int resize(matrix**)** that doubles the rows and columns of the matrix. The old nodes need to be copied to the new matrix. Return 0 if success, 1 if failure.

```
int resize(matrix** M){
      (*M)->rowList = realloc((*M)->rowList, 2*M->rows);
      (*M)->colList = realloc((*M)->colList, 2*M->cols);

}
```

4. Write a function **int transpose(matrix**)** that converts the matrix to its transpose. Transpose of a matrix M is defined as a matrix M1 where rows of M are equivalent to columnsof M1 and columns of M are equivalent to rows of M1. For example the transpose of M = {{1,2},{3,4}} is M1 = {{1,3},{2,4}}

```
int transpose(matrix**  M) {
        node** tmp = (*M)->rowList;
        (*M)->rowList = (*M)->colList;
        (*M)->colList = tmp;
        int temp = (*M)->rows;
        (*M)->rows = (*M)->cols;
}
```