DATA STRUCTURE AND ALGORITHMS

Deepali Londhe
Information Technology,
PICT, Pune

1

UNIT-I INTRODUCTION

Linked Organization

2

2

UNIT-I INTRODUCTION

Introduction to Data Structures: Concept of data, Data object, Data structure, Concept of Primitive and non-primitive, linear and Nonlinear, static and dynamic, persistent and ephemeral data structures, Definition of ADT

Analysis of algorithm: Frequency count and its importance in analysis of an algorithm, Time complexity & Space complexity of an algorithm Big 'O', 'Ω' and 'Θ' notations,

Sequential Organization: Single and multidimensional array and address calculation.

Linked Organization: Concept of linked organization, Singly Linked List, Doubly Linked List, Circular Linked List (Operations: Create, Display, Search, Insert, Delete).

CONTENTS

- **Linked Organization:**
- **Concept of linked organization**
- **Singly Linked List**
- **Doubly Linked List**
- **Circular Linked List**
- **(Operations: Create, Display, Search, Insert, Delete).**

AGENDA

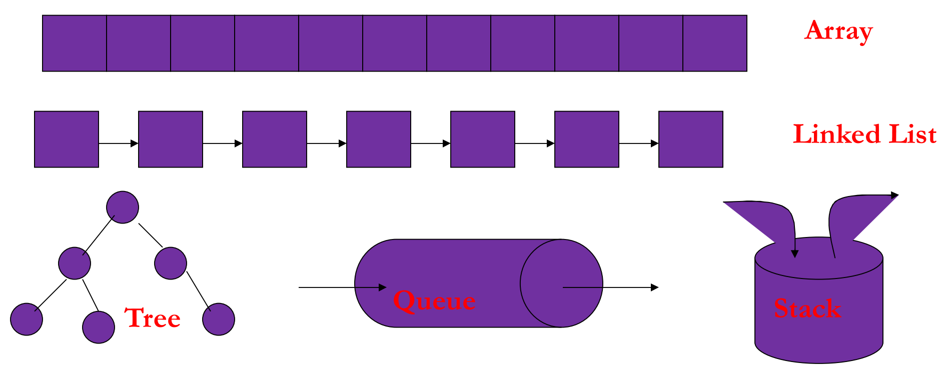
- **Linked Organization:**
- **Concept of linked organization**
- **Singly Linked List**
- **Doubly Linked List**
- **Circular Linked List**
- **(Operations: Create, Display, Search, Insert, Delete).**

REVISION-WHAT IS DATA STRUCTURE? WHY DS?

- A way of organizing, storing, accessing and updating data is data structure.
- So that it can be used efficiently and effectively.
- E.g. Array, lists, stacks, queues, tree, graphs
- Data structure is the logical or mathematical model of a particular organization of data.
- A group of data elements grouped together under one name.
  - For example, an array of integers

Program design depends crucially on how data is structured for use by the program
- Implementation of some operations may become easier or harder
- Speed of program may dramatically decrease or increase
- Memory used may increase or decrease
- Debugging may be become easier or harder

REVISION-Types of data structures

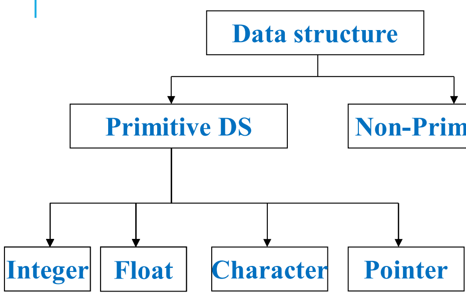


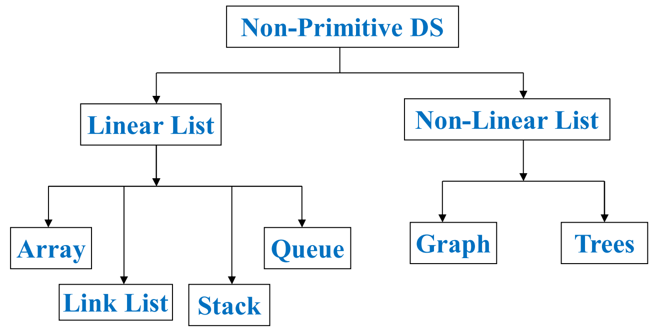
There are many, but we named a few. We'll learn these data structures in great detail!

REVISION

- Scalar Data Structure (primitive data structure)
- Integer, Character, Boolean, Float, Double, etc.
- Vector or Linear Data Structure (Non- primitive data structure)
- Array, List, Queue, Stack, Priority, Queue, Set, etc.
- Linear DS-the elements form a one to one correspondence between the elements
- Non-Linear data structures-Elements form a one to many correspondence between the elements e.g.Tree, Graph etc.

REVISION





- <u>**Linear Data structure**</u>
- Data elements are arranged sequentially or linearly.
- Single level
- Traverse all the elements in single run only.
- Easy to implement because computer memory is arranged in a linear way.

- <u>**Non-Linear Data structure**</u>
- Data elements are not arranged sequentially or linearly.
- Single level is not involved
- Can’t traverse all the elements in single run
- Not easy to implement.
- Efficient Memory utilization

OPERATIONS ON LINEAR STRUCTURES

Create :Create the respective data structure for a specific data type, with size, storage allocation for that size.

Traversal: Travel through the data structure

Search: Traversal through the data structure for a given element

Insertion: Adding new elements to the data structure

Deletion: Removing an element from the data structure

Sorting: Arranging the elements in some type of order

Merging: Combining two similar data structures into one

LIST OPERATIONS

1. Create an empty list
2. Determine whether a list is empty
3. Get the item at a given position in a list (retrieve)
4. Determine the number of items on a list
5. Determine the index of an element
6. Traverse the list
7. Add an item at a given position in a list
8. Remove the item at a given position in a list
9. Remove all the items from a list
10. Other operations?

THE LIST ADT

A sequence of zero or more elements

$A_1, A_2, A_3, \ldots A_N$

N: length of the list

A_1: first element

A_N: last element

A_i: position i

If N=0, then empty list

Linearly ordered
- A_i precedes A_{i+1}
- A_i follows A_{i-1}

OPERATIONS

printList: print the list

makeEmpty: create an empty list

find: locate the position of an object in a list
- list: 34,12, 52, 16, 12
- find(52) → 3

insert: insert an object to a list
- insert(x,3) → 34, 12, 52, x, 16, 12

remove: delete an element from the list
- remove(52) → 34, 12, x, 16, 12

findKth: retrieve the element at a certain position

IMPLEMENTATION OF AN ADT

Choose a **data structure** to represent the ADT

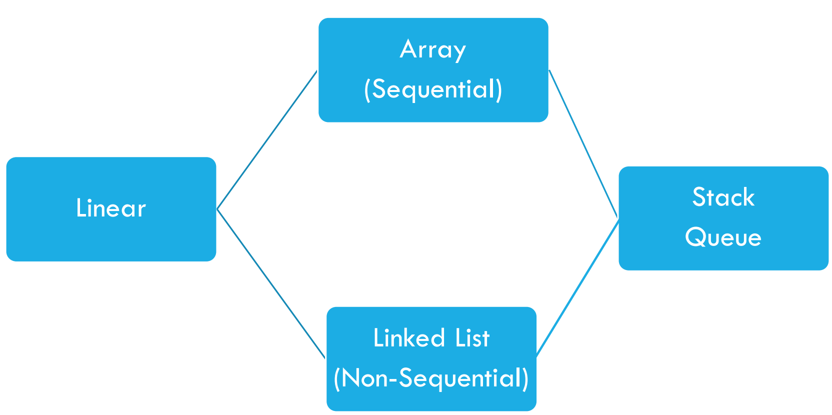
- E.g. arrays, records, etc.

Each operation associated with the ADT is implemented by one or more subroutines

Two standard implementations for the list ADT

- Array-based
- Linked list

LINEAR DATA STRUCTURE(LINEAR LIST)



REVISION- ARRAY

14	12	3	445	511	50	.	.	.	.	.

Array

- An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.
- An array is collection of items stored at contiguous memory locations.
- An *array* is a finite, ordered and collection of homogeneous data elements
- **Finite:** because it contains only limited number of elements
- **Ordered:** as all the elements are stored one by one in contiguous locations of computer memory in a linear ordered fashion
- **Homogeneous:** all elements of an array are of the same data type only

ARRAY IMPLEMENTATION

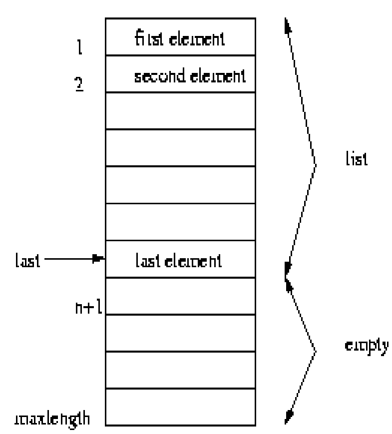
Elements are stored in contiguous array positions

Requires an estimate of the maximum size of the list
➢ waste space

printList and find: O(n)

findKth: O(1)

insert and delete: O(n)
- e.g. insert at position 0 (making a new element)
  - requires first pushing the entire array down one spot to make room
- e.g. delete at position 0
  - requires shifting all the elements in the list up one
- On average, half of the lists needs to be moved for either operation



OPTIONS FOR IMPLEMENTING AN ADT LIST

Arrays:

+ Fast element access.

- Impossible to resize.

Solution: Linked List

- Linked list is able to grow in size as needed. It does not require the shifting of items during insertions and deletions

LIST OVERVIEW

Linked lists

- Abstract data type (ADT)

Basic operations of linked lists

- Insert, find, delete, print, etc.

Variations of linked lists

- Circular linked lists
- Doubly linked lists

LIST ADT

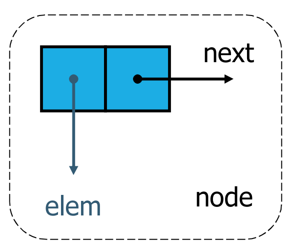
ADT with position-based methods

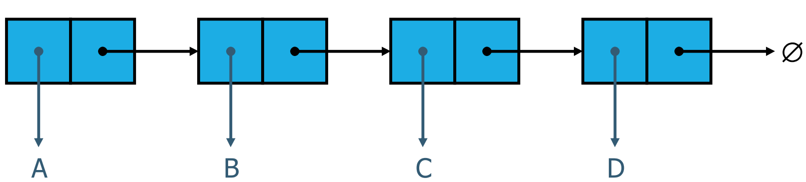
generic methods	size(), isEmpty()	
query methods	isFirst(p), isLast(p)	
accessor methods	first(), last()	
	before(p), after(p)	
update methods	swapElements(p,q),	insertLast(e)
	replaceElement(p,e)	insertBefore(p,e)
	insertFirst(e)	insertAfter(p,e)
		removeAfter(p)

SINGLY LINKED LIST

A singly linked list is a concrete data structure consisting of a sequence of nodes

Each node stores
- **element**
- **link to the next node**





POINTER-BASED LINKED LISTS

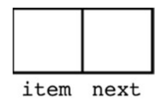
A node in a linked list is usually a **struct**

A structure to store two data items and a pointer to another node of the same type.this type of structure is known as **self referential structure**

```
struct Node
{
   int item
  Node *next;
}; //end struct
```

A node is dynamically allocated

```
Node *p;
p =new Node;
```



Node

Here, head node will point to first node of Linked List. where,

- If head is *NULL*, the linked list is empty
- Else head contains address of first node of link-list.
- Thus, initially

```
Node *head;
head = NULL;
```


POINTER-BASED LINKED LISTS

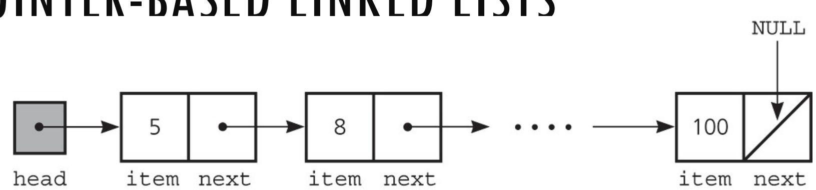


figure : **A head pointer to a list**

INSERTING A NEW NODE

Possible cases of InsertNode

1. Insert into an empty list
2. Insert in front
3. Insert at back
4. Insert in middle

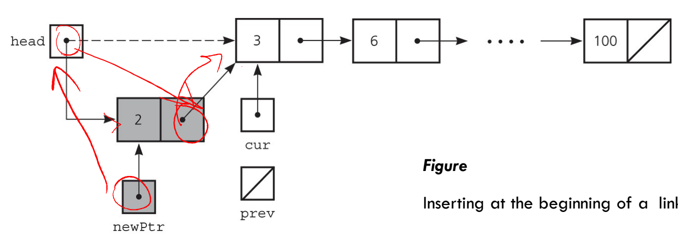
But, in fact, only need to handle two cases

- Insert as the first node (Case 1 and Case 2)
- Insert in the middle or at the end of the list (Case 3 and Case 4)

INSERTING A NODE INTO A SPECIFIED POSITION OF A LINKED LIST

To insert a node at the beginning of a linked list

```
newPtr->next = head;
head = newPtr;
```



Figure

Inserting at the beginning of a linked list

INSERTING A NODE INTO A SPECIFIED POSITION OF A LINKED LIST

To insert a node between two nodes

newPtr->next = cur;

prev->next = newPtr;

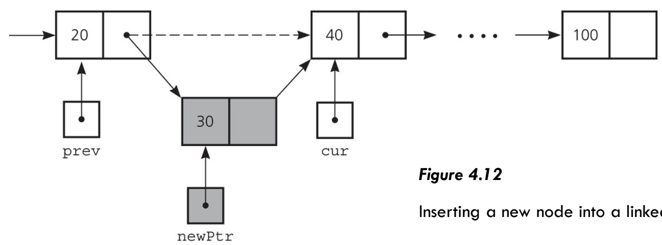


Figure 4.12

Inserting a new node into a linked list

PROCEDURE INSERT(FIRST,POS,X):

1. NEV <= NODE
2. P←1
3. INFO(NEV) ← X

4. IF FIRST = NULL (Insert at beginning as empty list)
   THEN LINK(NEV) ← NULL
   FIRST ←NEV
   RETURN(FIRST)

5. IF POS = 1 (insert at beginning)
   THEN LINK(NEV) ← FIRST
   FIRST ←NEV
   RETURN(FIRST)

CONT..

6. TEMP ← FIRST (Insert in between the list OR at last position)

7. REPEAT WHILE LINK(TEMP) != NULL AND P < POS – 1
   TEMP ← LINK(TEMP)
   P ← P+1

8. LINK(NEV) ← LINK(TEMP)
   LINK(TEMP) ← NEV

9. RETURN

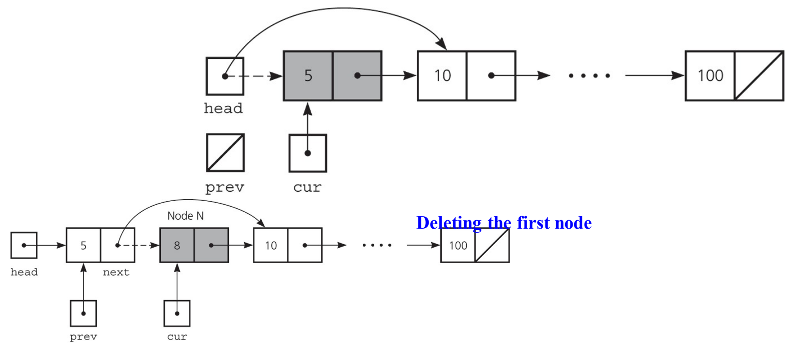
PROCEDURE DISPLAY (FIRST):

1. TEMP ← FIRST

2. REPEAT WHILE (TEMP) != NULL
   WRITE (DATA(TEMP))
   TEMP ← LINK(TEMP)

3. RETURN

DELETING A SPECIFIED NODE FROM A LINKED LIST



Deleting a node from a linked list

DELETING A SPECIFIED NODE FROM A LINKED LIST

Deleting an interior node

```
prev->next=cur->next;
```

Deleting the first node

```
head=head->next;
```

Deleting last

```
cur->next = NULL;
 free(cur); //return memory to system
```

PROCEDURE DELETE (X, FIRST):

1. IF FIRST = NULL
   THEN WRITE('UNDEFLOW')
   RETURN

2. TEMP ← FIRST

2. REPEAT WHILE DATA(TEMP) != X AND LINK(TEMP) != NULL
   PRED ← TEMP
   TEMP ← LINK(TEMP)

4. IF DATA(TEMP) != X
   THEN WRITE('NODE NOT FOUND')
   RETURN

CONT...

5. IF X= DATA(FIRST)
   THEN FIRST ← LINK(FIRST) (delete first element)
   ELSE LINK(PRED) ← LINK(TEMP) (delete last & betn. element)

6. FREE(TEMP)

7. RETURN(FIRST)

LINKED IMPLEMENTATION...

Requires no estimate of the maximum size of the list
- No wasted space

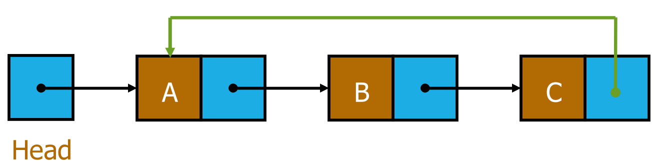
printList and find: O(n)

findKth: O(n)

insert and delete: O(1)
- e.g. insert at position 0 (making a new element)
  - Insert does not require moving the other elements
- e.g. delete at position 0
  - requires no shifting of elements
- Insertion and deletion becomes easier, but finding the Kth element moves from O(1) to O(n)

VARIATIONS OF LINKED LISTS

Circular linked lists
- The last node points to the first node of the list



- How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

PROCEDURE INSCIRLIST(HEAD,X,POS):

1. NEV <= NODE
2. P← 1
3. INFO(NEV) ← X

4. IF Head = NULL (Insert at beginning as empty list)
   THEN LINK(NEV) ←NEV
   Head←NEV
   RETURN(Head)

5. IF POS =1 (Insert at first position)
6. THEN TEMP = Head
   REPEAT WHILE LINK(TEMP) != Head // locate last
   TEMP=LINK(TEMP)
   LINK(NEV) ← Head
   Head ← NEV
   LINK(TEMP) = Head
   RETURN (Head)

CONT…

6. TEMP ← Head (Insert in between the list)

7. REPEAT WHILE LINK(TEMP) != Head AND P < POS – 1
   TEMP ← LINK(TEMP)
   P ← P+1

8. LINK(NEV) ← LINK(TEMP)
   LINK(TEMP) ← NEV

9. RETURN

PROCEDURE CIRDELETE (X, HEAD):

1. IF Head = NULL
   THEN WRITE('UNDER FLOW')
   RETURN

2. TEMP ← Head
3. REPEAT WHILE DATA(TEMP) != X AND LINK(TEMP) != Head
   PRED ← TEMP (Search Data Position Previous & current)
   TEMP ← LINK(TEMP)

4. IF DATA(TEMP) != X
   THEN WRITE('NODE NOT FOUND')
   RETURN

CONT…

5. IF X= DATA(Head)
   THEN REPEAT WHILE LINK(TEMP)!=Head
   TEMP=LINK(TEMP)
   Head ← LINK(Head) (delete Head element)
   LINK(TEMP)← Head

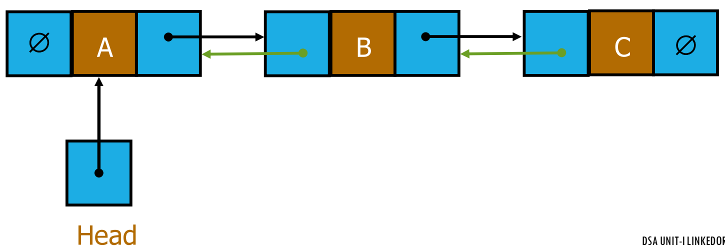
   ELSE
   LINK(PRED) ← LINK(TEMP) (delete last & betn. element)

6. FREE(TEMP)

7. RETURN(Head)

VARIATIONS OF LINKED LISTS
DOUBLY LINKED LISTS

- Each node points to not only successor but the predecessor
- There are two `NULL`: at the first and last nodes in the list
- Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



39

DLL

```
Struct dll
{ int data;
  dll *lptr;
  dll *rptr;
};
```

lptr	data	rptr

40

PROCEDURE INSDOUBLY (HEAD,X,POS):

1. NEV <= NODE
2. P← 1
3. INFO(NEV) ← X
4. IF Head = NULL //AND RPTR(Head) =NULL
   (Insert in Empty list)
   THEN LPTR(NEV) ←RPTR(NEV) ← NULL
   Head ←NEV
5. IF POS =1 (Insert at first position)
   THEN LPTR(NEV) ← NULL
   RPTR(NEV) ← Head
   LPTR(Head) ← NEV
   Head ← NEV

CONT...

6. TEMP ← Head (Insert in between the list)
7. REPEAT WHILE RPTR(TEMP) != NULL AND P < POS – 1
   TEMP ← RPTR(TEMP)
   P ← P+1
8. TEMP1 ← RPTR(TEMP)
   RPTR(NEV) ← TEMP1
   LPTR(NEV) ← TEMP
   RPTR(TEMP) ← NEV
   LPTR(TEMP1) ← NEV
9. RETURN

PROCEDURE DELDOUBLY(HEAD, LAST, X):

1. IF Head = NULL
   THEN WRITE('UNDER FLOW')
   RETURN
2. TEMP ← Head
3. REPEAT WHILE DATA(TEMP) != X AND RPTR(TEMP) != NULL
   TEMP ← RPTR(TEMP)
4. IF DATA(TEMP) != X
   THEN WRITE('NODE NOT FOUND')
   RETURN

5. IF X=DATA(Head) AND X=DATA(Last) (delete single element)
   THEN Head← Last← NULL

   ELSE IF X= DATA(Head)
   THEN Head ← RPTR(Head) (delete first element)
   LPTR(Head) ← NULL

   ELSE IF X= DATA(Last) (delete last element)
   THEN Last ← LPTR(Last)
   RPTR(Last) ← NULL

   ELSE (delete from between)
   PRED ← LPTR(TEMP)
   SUC ← RPTR(TEMP)
   RPTR(PRED) ← SUC
   LPTR(SUC) ← PRED
6. FREE(TEMP)
   RETURN

ARRAY VERSUS LINKED LISTS

Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.

- **Dynamic**: a linked list can easily grow and shrink in size.
  - We don't need to know how many nodes will be in the list. They are created in memory as needed.
  - In contrast, the size of a array is fixed at compilation time.
- **Easy and fast insertions and deletions**
  - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
  - With a linked list, no need to move other nodes. Only need to reset some pointers.

Applications of Linked Lists

1.They are mainly used for the implementation of other data structures like stacks, queues, trees and graphs.

2.To manipulate polynomials.

3.To represent sparse matrices.

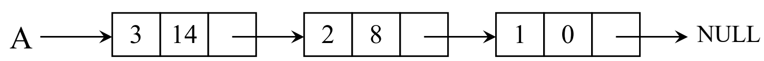
Polynomial Representation

$A=3x^{14}+2x^8+1$

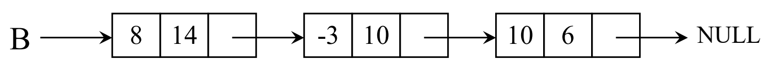
$B=8x^{14}-3x^{10}+10x^6$

```
typedef struct polynode {
      int coef;
      int expon;
       link;
}polypointer;
polypointer *A, *B ;
```

$A=3x^{14}+2x^8+1$



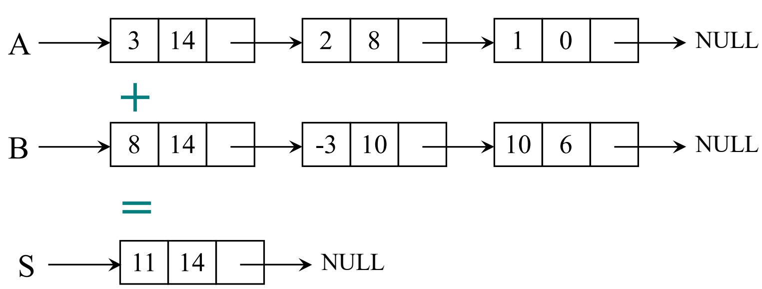
$B=8x^{14}-3x^{10}+10x^6$



47

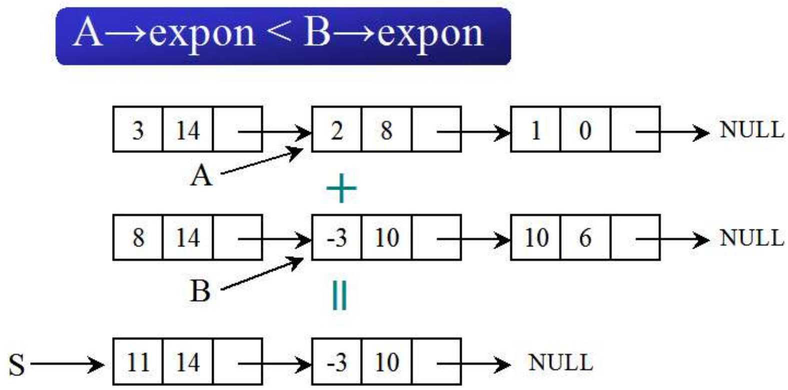
Adding Polynomials: CASE (I)

A→expon = = B→expon

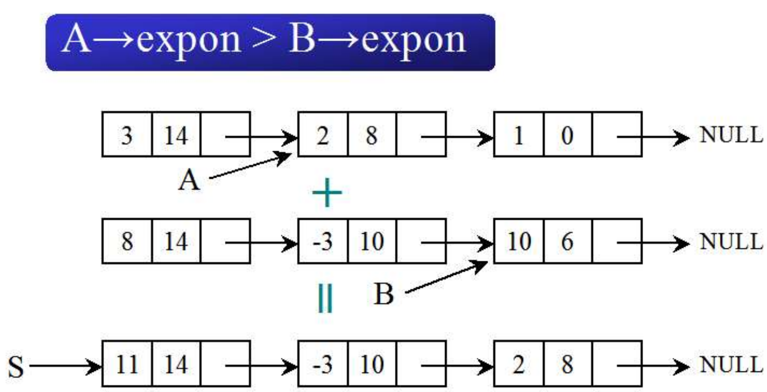


48

Adding Polynomials: CASE (II)



Adding Polynomials: CASE (III)



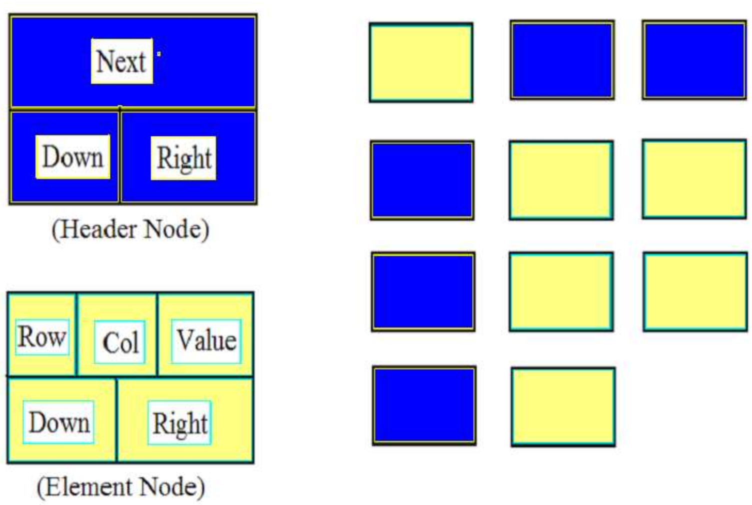
This process is continued until we reach the end of a list.Then the remaining elements of the other list should be copied to the result.

Sparse Matrix

- A sparse matrix is a matrix populated primarily with zeros.
- Huge sparse matrices often appear in science or engineering when solving partial differential equations.
- Operations using standard matrix structures and algorithms are slow and consume large amounts of memory when applied to large sparse matrices.

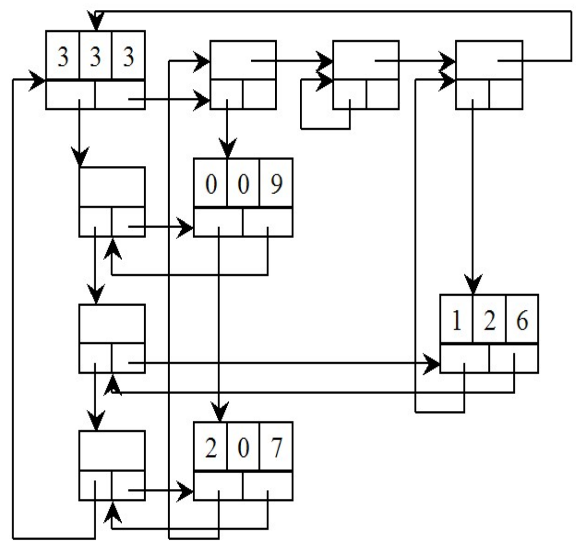
Example: $\begin{bmatrix} 9 & 0 & 0 \\ 0 & 0 & 6 \\ 7 & 0 & 0 \end{bmatrix}$

Sparse Matrix



Sparse Matrix

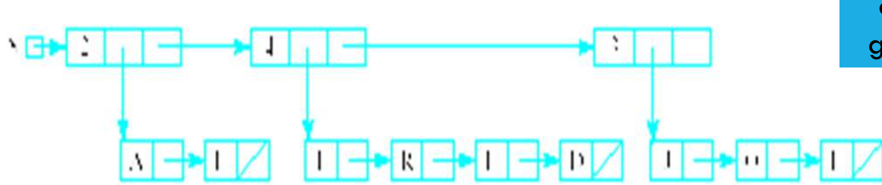
$$\begin{bmatrix} 9 & 0 & 0 \\ 0 & 0 & 6 \\ 7 & 0 & 0 \end{bmatrix}$$



GENERALIZED LISTS

Examples so far have had atomic elements
- The nodes are not themselves lists

Consider a linked list of strings
- The strings themselves can be linked lists of characters

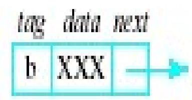


This is an example of a generalized list

GENERALIZED LISTS

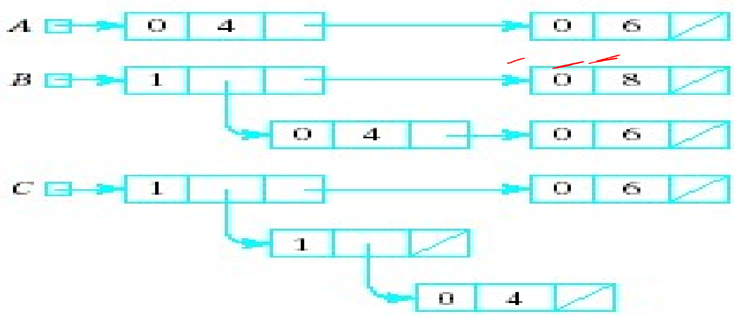
Commonly represented as linked lists where

- Nodes have a tag field along with data and link

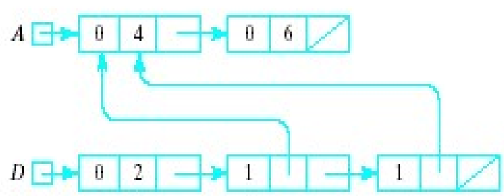


Tag used to indicate whether data field holds

- Atom
- Pointer to a list



GENERALIZED LISTS

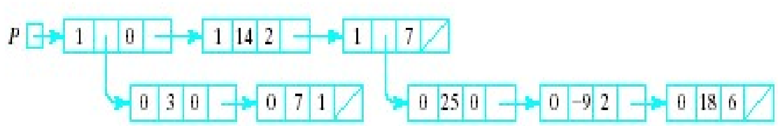


Lists can be shared

- To represent (2, (4,6), (4,6))

For polynomials in two variables

$P(x,y) = 3 + 7x + 14y^2 + 25y^7 - 7x^2y^7 + 18x^6y^7$



REFERENCES

Books

D. E. Knuth, *The Art of Computer Programming: Vol. 3: Sorting and Searching, 2d ed., Addison-* Wesley, Reading, Mass., 1998.

SORTING AND SEARCHING ALGORITHMS: A COOKBOOK BY THOMAS NIEMANN

Robert Sedgewick, Kevin Wayne, "Algorithms", 4th edition, Addison-Wesley Professional

Samanta Debasis, "**CLASSIC DATA STRUCTURES", PHI, 2nd ed.**

Ellis Horowitz and Sartaj Sahni , "Fundamentals of Data Structures", Computer Science Press, 1983

R. Gilberg, B. Forouzan, "Data Structures: A pseudo Code Approach with C++", Cengage Learning, ISBN 9788131503140.

E. Horowitz, S. Sahni, D. Mehta, "Fundamentals of Data Structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN 16782928

Dinesh P. Shah, Sartaj Sahani , "Handbook of DATA STRUCTURES and APPLICATIONS", CHAPMAN & HALL/CRC

Bayer B. et al. (2015) Electro-Mechanical Brake Systems. In: Winner H., Hakuli S., Lotz F., Singer C. (eds) Handbook of Driver Assistance Systems. Springer, Cham

Web

- http://statmath.wu.ac.at/courses/data-analysis/itdtHTML/node55.html
- https://en.wikipedia.org/wiki/Persistent_data_structure

57

THANK YOU !!!

58