CSN 102: DATA STRUCTURES

Linked List: List, Dynamic Memory Allocation, SLL, Polynomial Addition, Sparse Matrices, Garbage Collection, Josephus Problem, Circular LL

Abstract Data Type (ADT)

- An Abstract Data Type is:
  - Set of values
  - Set of operations which can be uniformly applied to these values
  - Set of Axioms

What is List?

- Countable number of ordered values
- Each occurrence of a value is a distinct item
- Implemented differently in different programming languages
- Linked list is an implementation of list

List Abstract Data Type

- List ADT has:
  - Values based on what type of data list stores
  - Main operations:
    - new() : creates a new list
    - prepend(L, key): add element key to front of list L
    - append(L, key): add element key to end of list L
    - remove(L, key): removes element key from list L
    - search(L, key): find location of element key in L
    - head(L): returns the first object in L
    - isEmpty(L): checks whether list L is empty or not
  - Axioms:
    - Based on implementation

Drawbacks with Array

- Is array also a list?

Drawbacks with Array

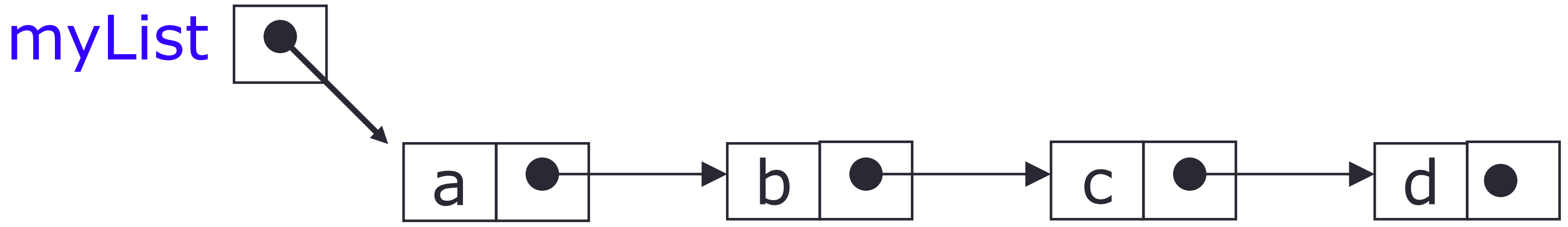
- Is array also a list?
  - Yes, certainly.

- Problem: An array has a limited number of elements
  - Routines inserting a new value have to check that there is room

- Solution: Multiple solutions exist
  - Increase the size of array with some constant each time array is full
  - Double the size of array each time array is full
  - Use Linked List data structure

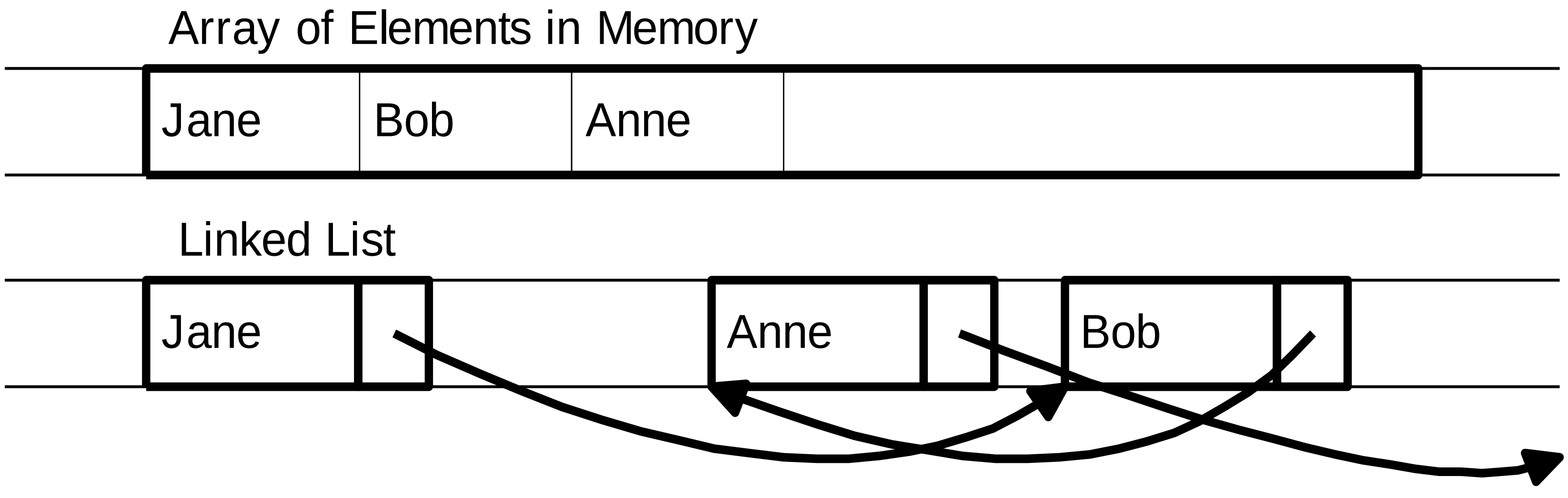
Linked List(LL)

- Nodes stores the data/values
- Sequence of nodes; each node points to next node in list
- Add node dynamically when required
- Can store infinite data until memory of system is exhausted
- LL might have a header to start node. Eg. myList
- Last node contains a null link



Dynamic Memory Allocation(1)

- Allocate memory to elements one at a time as needed, have each element keep track of the *next* element
- Result is referred to as linked list of elements, track next element with a pointer



Dynamic Memory Allocation(2)

- malloc(total_size) and calloc(count, size_of_one_block) are two functions for dynamic memory allocation
- Both returns a void pointer to start of memory is returned if allocated, else NULL is returned
- calloc initialize memory block with 0 but slower then malloc
- TypeCast void pointer to use memory as required data type
- int *ptr = (int*)malloc(10*sizeof(int))

Typecast to int pointer

Finds size of input data type

- int *ptr = (int*)calloc(10,sizeof(int))

malloc & variable length array

- Similarity: Both int arr[size] and malloc() allocates memory of dynamic size
- Difference: Memory allocated by int arr[size] is limited by the scope in which it is defined while malloc() allocated memory is accessible throughout the program

More Terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)
- Linked list in context of C are sometime referred as List
- Until not specified, linked list refers to Singly-Linked List

Creating a (Singly) Linked List

- **Type declaration:**

```
typedef struct ESTRUCT {
  DataType data;   /*DataType is type for element of list */
  struct ESTRUCT *next;  /*Pointer to struct ESTRUCT*/
} Estruct ;
typedef struct ESTRUCT * Eptr;
```

- **Pointer to node:**

```
struct ESTRUCT *ListStart;
Estruct *ListStruct;
EPtr ListStart;
```

Create LL with 3 Nodes

```
void main() {           /*Assume data is int*/
        Estruct first, second, third;
        first.data = 5;
        first.next = &second;
        second.data = 9;
        second.next = &third;
        third.data = 6;
        third.next = NULL;
        Eptr ListStart = &first;
}
```

Deletion in LL

- Delete a node from the LL from previous implementation
- Delete first node
  - Set header to point second node
  - ListStart=&second;
- Delete from middle
  - Set previous node to point node next to middle
  - first.next = &third;
- Delete Last node
  - Set pointer of second last node to point NULL
  - second.next = NULL;

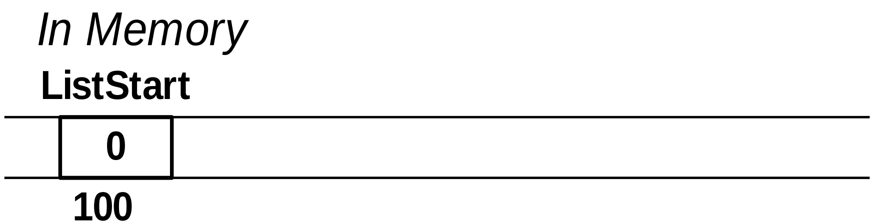
Issue with Implementation

- Above strategy for node creation will work in main function
- What would happen if we create a node in some function other than main?
  - Could not be accessible by other functions as limited by scope
- Solution: use dynamic memory allocation

LL with 3 nodes (dynamic)

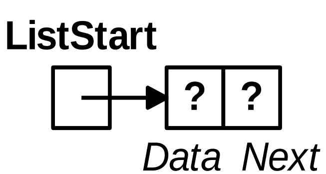
void main() { /*assume data is int */

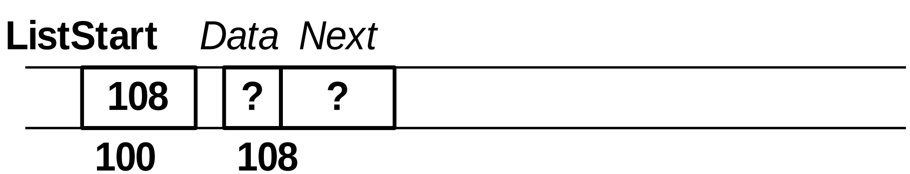
Eptr ListStart = NULL; /*safe to give a legal value*/



ListStart = (Eptr)malloc(sizeof(Estruct));

/* ListStart points to memory allocated at location 108 */





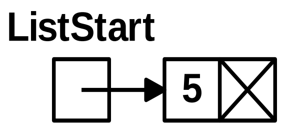
}

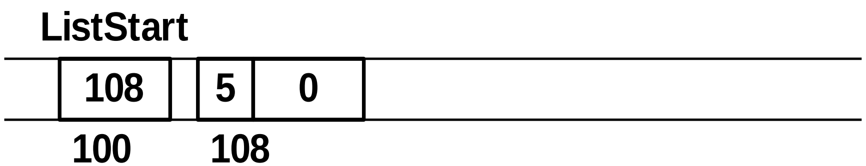
LL with 3 nodes (dynamic)

ListStart->data = 5;

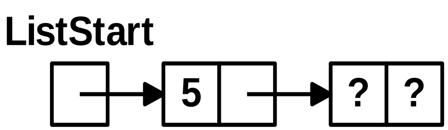
/*ListStart->data is equivalent to (*ListStart).data */

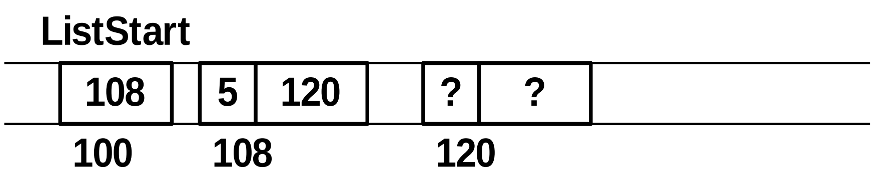
ListStart->next = NULL;





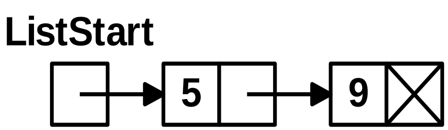
ListStart->next = (Eptr)malloc(sizeof(Estruct));

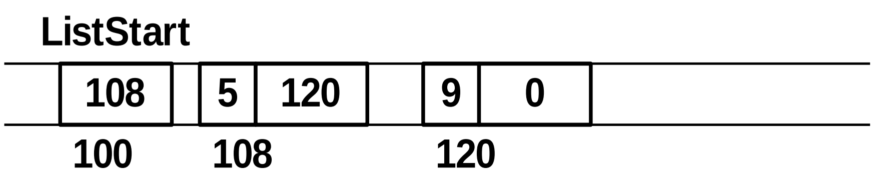




ListStart->next->data = 9;

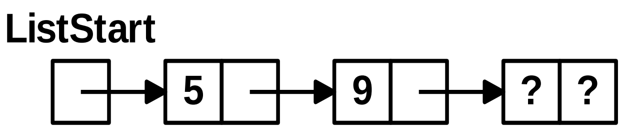
ListStart->next->next = NULL;

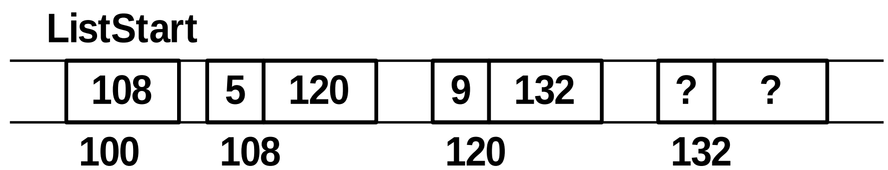




LL with 3 nodes (dynamic)

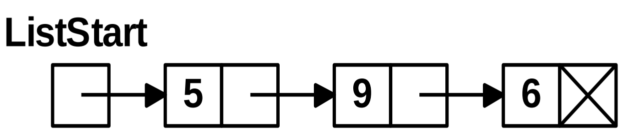
ListStart->next->next = (Eptr)malloc(sizeof(Estruct));

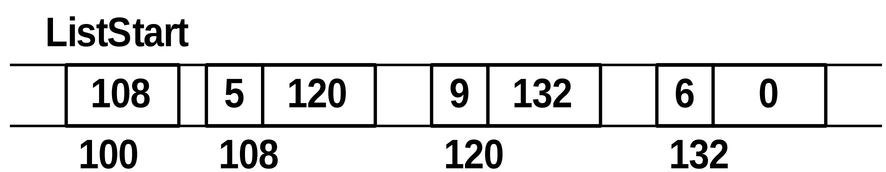




ListStart->next->next->data = 6;

ListStart->next->next->next = NULL;





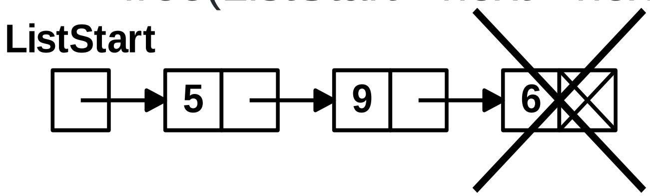
/* Linked list of 3 elements (count data values):
ListStart points to first element
ListStart->next points to second element
ListStart->next->next points to third element
and ListStart->next->next->next is NULL to indicate there is no fourth element */

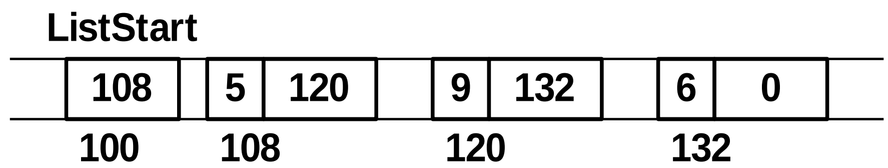
Deletion from LL(dynamic)

- Delete from LL in dynamic implementation strategy
- Delete 6 from the previous list

/* To eliminate element, start with free operation */

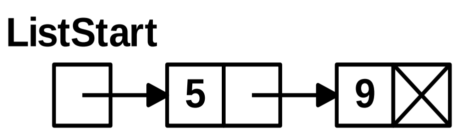
free(ListStart->next->next);

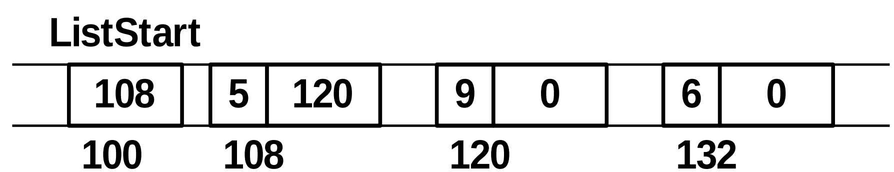




/* NOTE: free not enough -- does not change memory
Element still appears to be in list but C might give memory away
in next request. Need to reset the pointer to NULL */

ListStart->next->next = NULL;





/* Element at 132 no longer part of list (safe to reuse memory) */

Common Mistakes

- Dereferencing a NULL pointer
  - ListStart = NULL;
  - ListStart->data = 5;   /* ERROR */

- Using a freed element
  - free(ListStart->next);
  - ListStart->next->data = 6; /* PROBLEM */

- sUsing a pointer before set
  - ListStart = (EPtr) malloc(sizeof(EStruct));
  - ListStart->next->data = 7; /* ERROR */

LL Using Functions

- Certain linked list operations (init, insert, etc.) may change element at start of list (what ListStart points at)
- To change what ListStart points to could pass a pointer to ListStart (pointer to pointer)
- Alternately, in each such routine/function, always return a pointer to ListStart and set ListStart to the result of function call (if ListStart doesn't change it doesn't hurt)
- pseudocode:

```
Eptr <function_name>(<input_parameters>) {
        <some_processing>
        return <pointer_to_struct>
}
```

Initialization of LL

- Create an empty linked list pointing to null
- Return a header pointer

```
Eptr initializeLL () {
        return NULL;
}

main() {
        Eptr ListStart = initializeLL();
}
```

Create New Node

- Each time function is called, dynamically create a node and return pointer to it
- Pass element and address to be stored

```
Eptr newNode (DataType ndata, Eptr nnext) {
   Eptr newN = (Eptr) malloc(sizeof(Estruct));  /*new node*/
   newN->data = ndata;  /*set data of new node to ndata*/
   newN->next = nnext;  /*set next of new node to nnext*/

   return newN;  /*return pointer to new node*/
   }
```

Insertion in Linked List

- Involves two step:
  - Finding the correct position
  - Doing the work to add the node

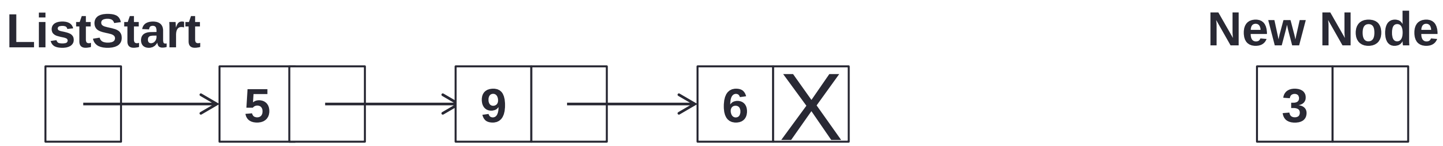
- Three possible positions:
  - Front
  - End
  - Somewhere in the middle

Insertion in Linked List (Front)

- Finding the correct location
  - No work required as already known
  - Irrespective of list is empty or not, header will always point to correct location

Insertion in Linked List (Front)

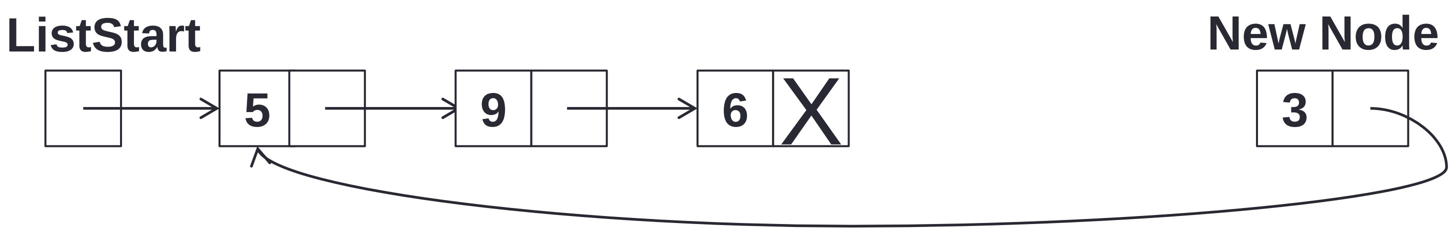
- Add new node to list
  - Save element in data field of new node




```
Eptr insertF(Eptr start, DataType nData) {
        return newNode(ndata, start);
}
```

Insertion in Linked List (Front)

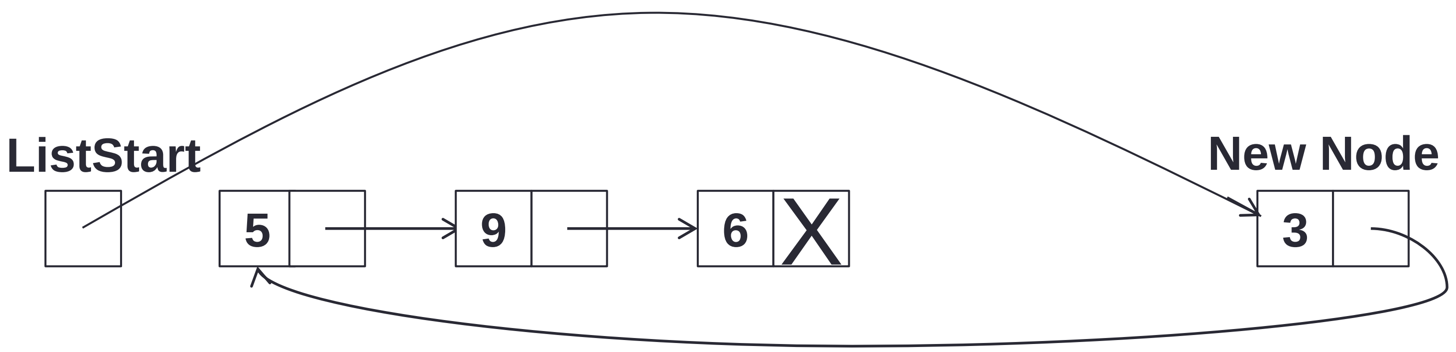
- Add new node to list
  - Save element in data field of new node
  - Make new node's next pointer to point start of existing list




```
Eptr insertF(Eptr start, DataType nData) {
        return newNode(ndata, start);
}
```

Insertion in Linked List (Front)

- Add new node to list
  - Save element in data field of new node
  - Make new node’s next pointer to point start of existing list



- Make start of list point to new node in main

  ListStart = insertF(ListStart, ndata);

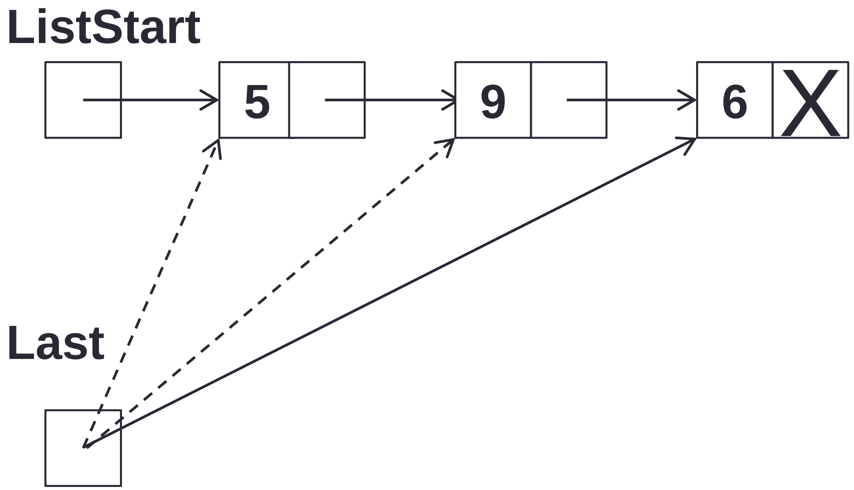
Insertion in Linked List (End)

- Finding the correct location
  - Need a “walker” to “walk” down the list till end is reached

```
Eptr insertE(Eptr start, dataType ndata) {
        Eptr last = start;   /*last is walker variable*/
        if (last == NULL)  /*list is empty, add at start*/
                return newNode(ndata, NULL);
        else {
                while (last->next != NULL)  /*find last node*/
                        last = last->next;
                last->next = newNode(ndata, NULL);
                return start;  /*start doesn't change*/
        }
}
```

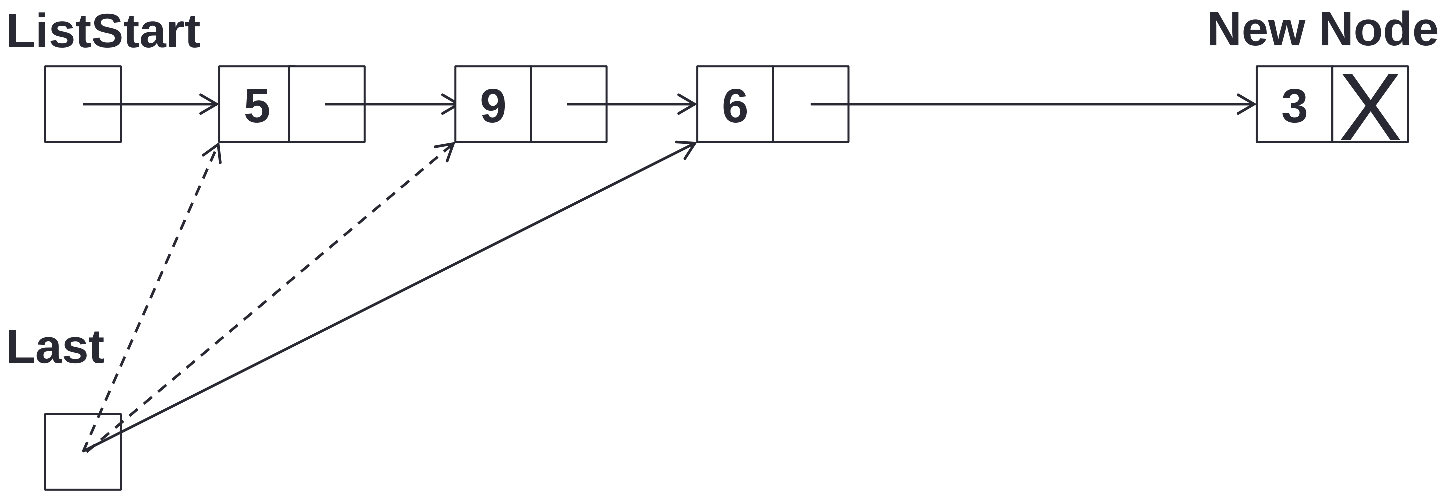
Insertion in Linked List (End)

- Find last of list



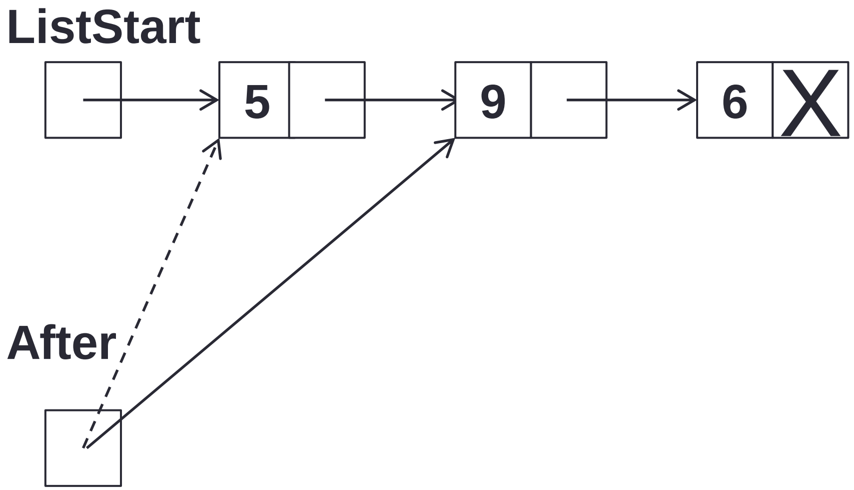
Insertion in Linked List (End)

- Find last of list
- Add new node to list
  - Save element in data field of new node
  - Save NULL in next field of new node
  - Make last node’s next pointer to point new node



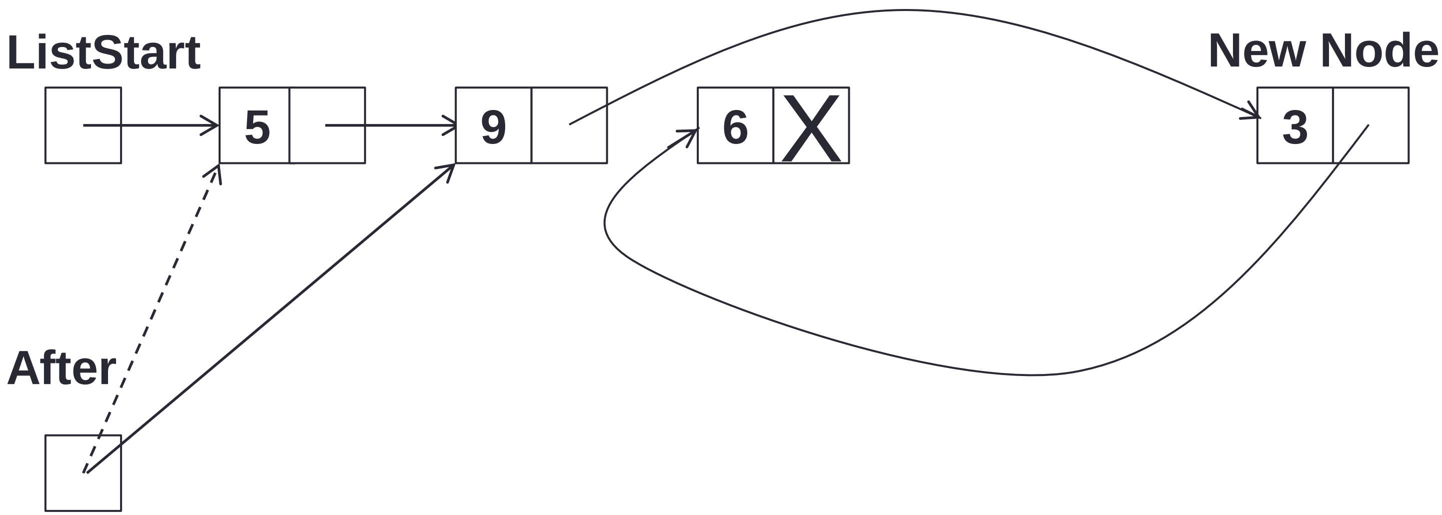
Insertion in Linked List (Between)

- Find the node you want to insert after



Insertion in Linked List (Between)

- Find the node you want to insert after
- Add new node to list
  - Save element in data field of new node
  - Save address of following node in next field of new node
  - Save address of new node in "after" node



Print a Linked List

- Use a “walker” to examine list from start to end

```
void printList(Eptr start) {
        Eptr temp = start;   /*temp is walker variable*/
        while (temp != NULL) {
                print(temp->data);
                temp = temp->next;
        }
}
```

- Why we need “walker” when we already have “start”?
  - Try to avoid start, could be required later in the function

Search in Linked List

- Compare every element in list; return pointer to node if found else return NULL

```
Eptr findE(Eptr start, dataType findData) {
        Eptr findP = start;   /*walker*/
        while(findP != NULL && findP->next == findData)
                findP = findP->next;
        retunr findP;
}
```

- Can we perform binary search given elements are sorted?

Search in Linked List

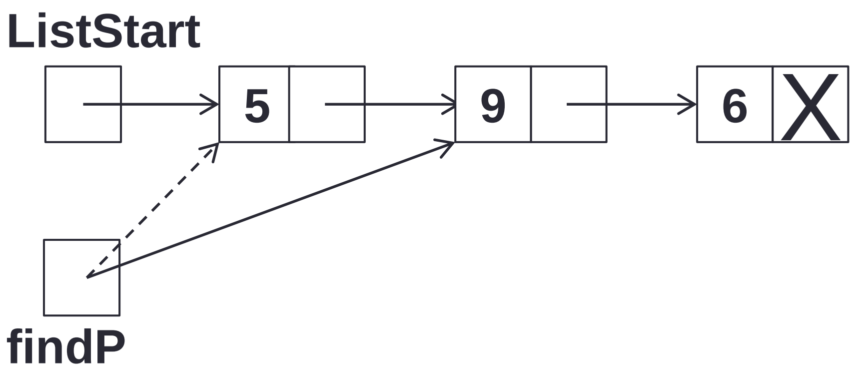
- Compare every element in list; return pointer to node if found else return NULL

```
Eptr findE(Eptr start, dataType findData) {
        Eptr findP = start;   /*walker*/
        while(findP != NULL && findP->next != findData)
                findP = findP->next;
        retunr findP;
}
```

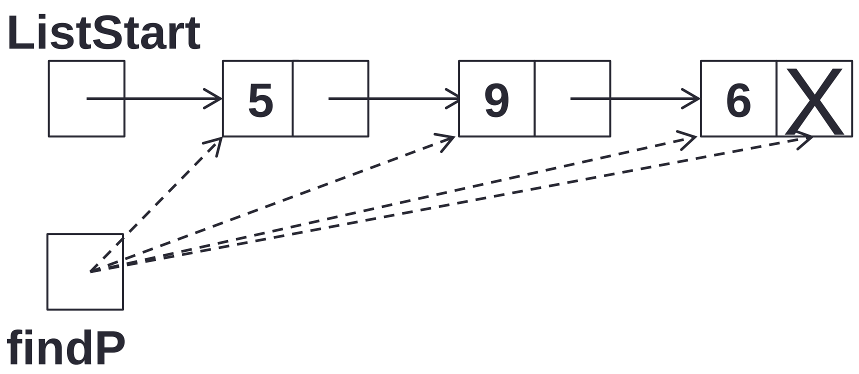
- Can we perform binary search given elements are sorted?
  - Yes, but the performance will be similar to linear search

Search in Linked List

- Search 9



- Search 4



Deletion in Linked List

- Involves two steps:
  - Find the node to be deleted
  - Change its predecessor to point its successor

- Deletion from front
  - Change header to point to second node

- Deletion from elsewhere
  - Change predecessor to point successor

Deletion in Linked List (Front)

- Change the header to point to second node.
- Set first node free

```
Eptr deleteF(Eptr start) {
        Eptr temp = start;
        temp = temp->next;
        free(start);
        return temp;
}
```

- Set start to returned pointer after deletion

```
        ListStart = deleteF(ListStart);
```

Deletion in Linked List (Except Front)

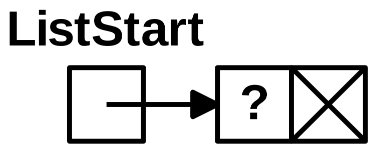
- Change the predecessor to point to successor
- Set deleted node free

```
Eptr deleteA(Eptr start, Eptr after) {
        Eptr temp = after;
        temp->next = temp->next->next;
        free(after);
        return start;
}
```

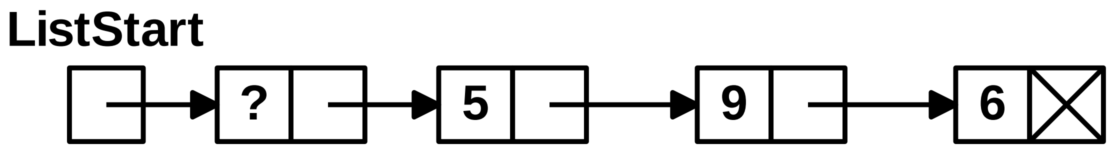
Linked List Variation: Dummy Head Node

Using "dummy" first (head) node:

Empty linked list



Sample list:



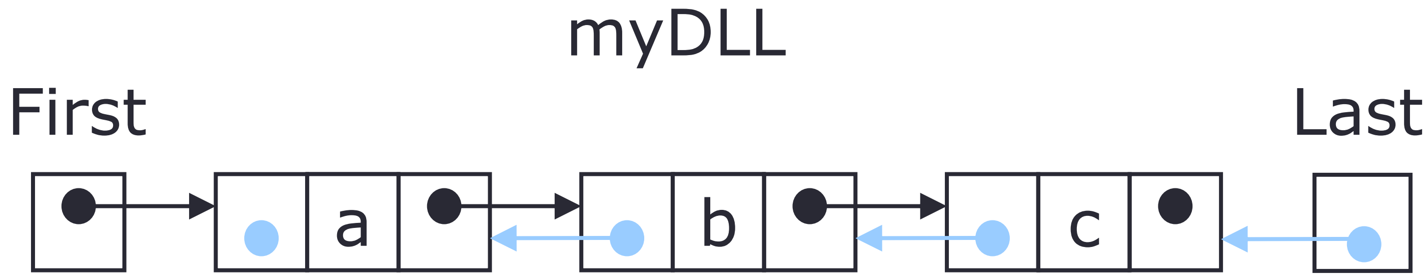
- Why?
  - No special case for inserting/deleting at beginning
  - Header (ListStart) does not change after it is initialized
- Disadvantage
  - cost of one extra element

Linked List Variation: Sorted List

- Idea: Keep the items on the list in a sorted order
- sort based on data value in each node
- Advantages:
  - already sorted
  - operations such as delete, find, etc. need not search to the end of the list if the item is not in list
- Disadvantages
  - insert must search for the right place to add element (slower than simply adding at beginning)

Doubly Linked List

- Each node contain data, link to its successor and a link to its predecessor
- Two headers pointing to first and last node respectively or pointing to NULL (if list is empty)



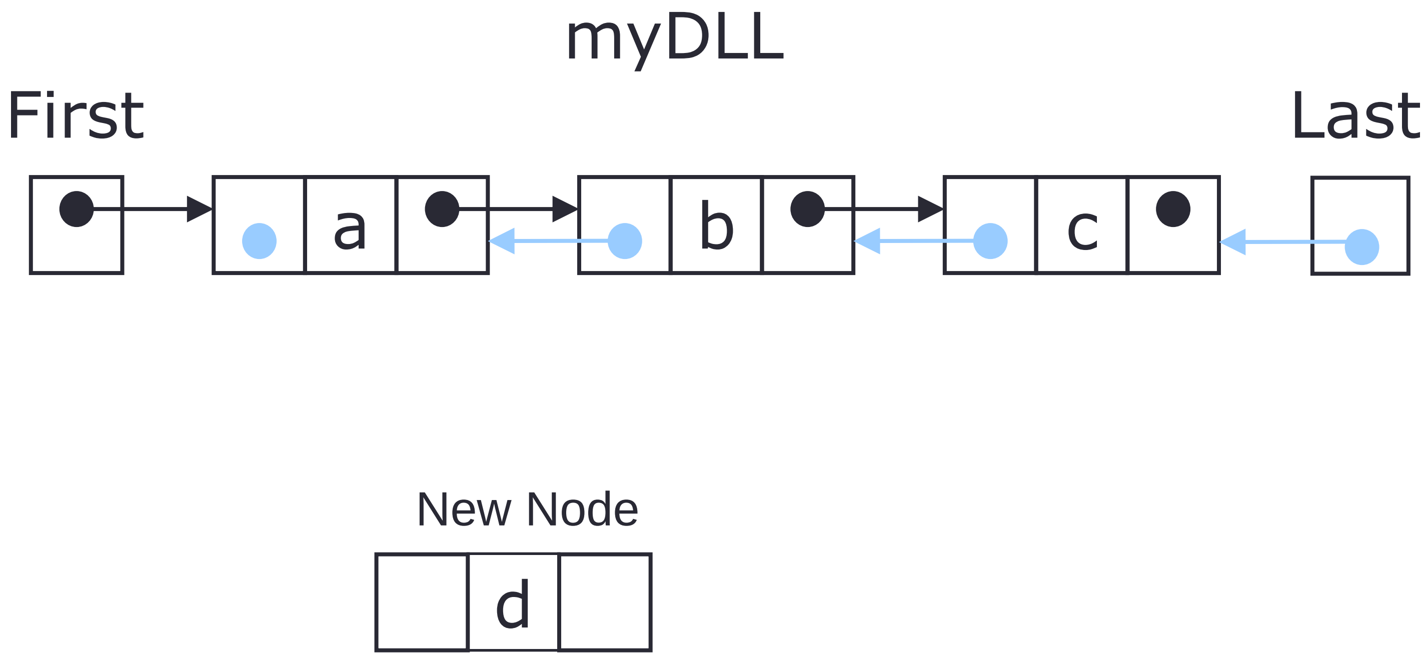
Doubly Linked List

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier

- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

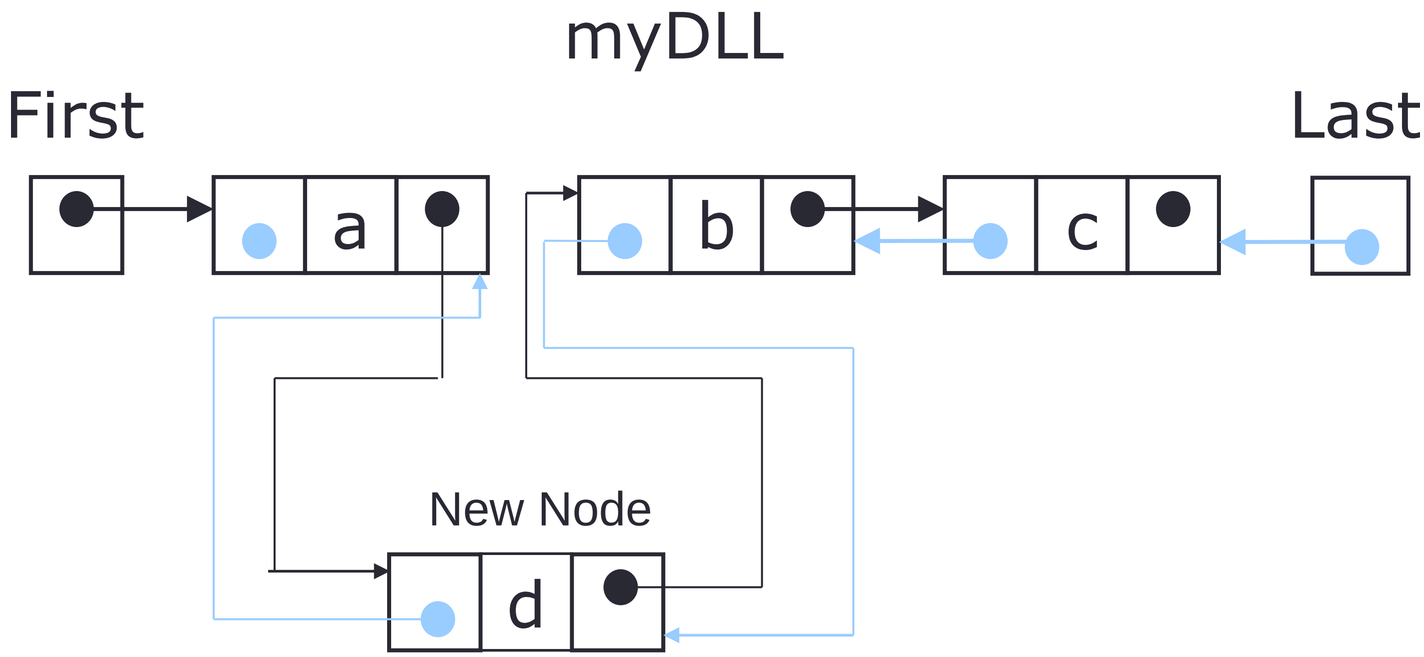
Insertion in DLL

- Change forward and backward pointers accordingly
- Insert element “d” after “a”



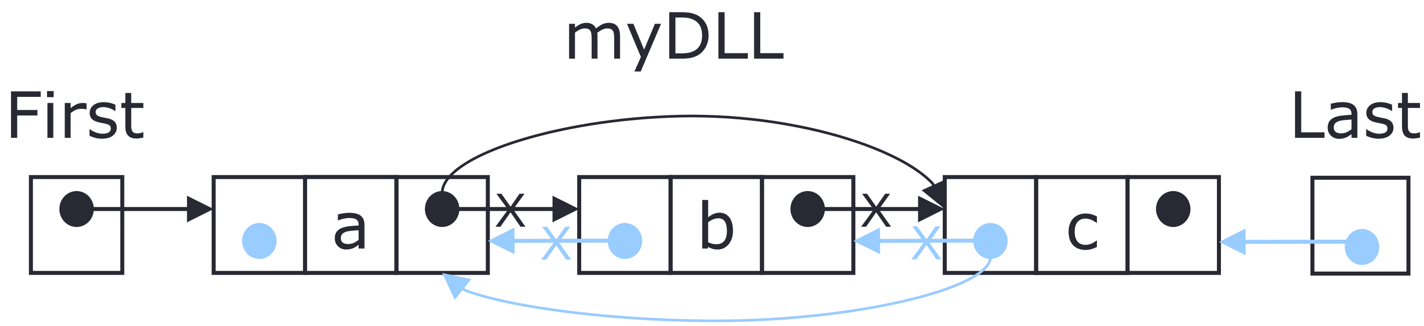
Insertion in DLL

- Change forward and backward pointers accordingly
- Eg. Insert element “d” after “a”



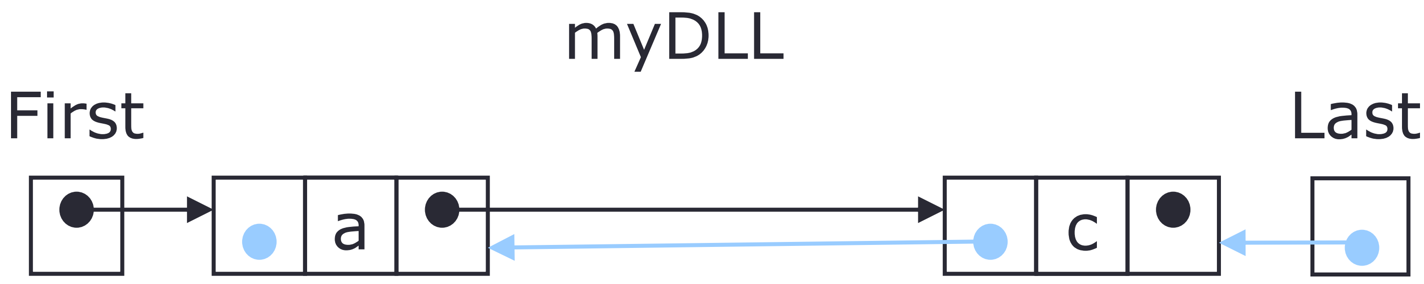
Deletion in DLL

- Change forward and backward pointers accordingly
- Insertion and Deletion in beginning and last are special cases and should be handled differently
- Eg. Delete “b” from previous list



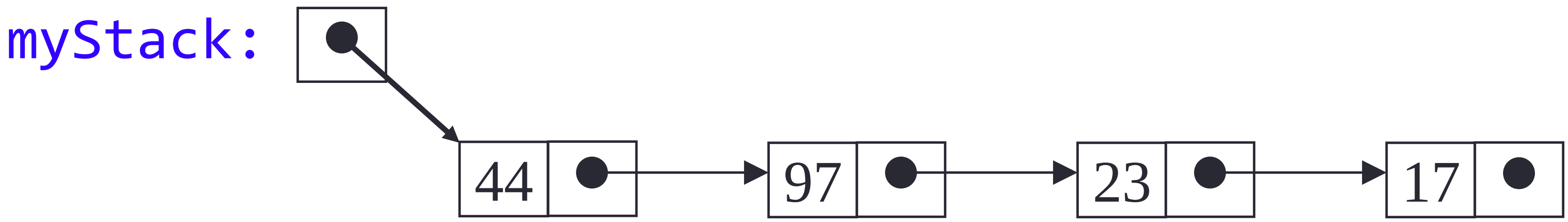
Deletion in DLL

- Change forward and backward pointers accordingly
- Insertion and Deletion in beginning and last are special cases and should be handled differently
- Eg. Delete “b” from previous list



Stack using LL

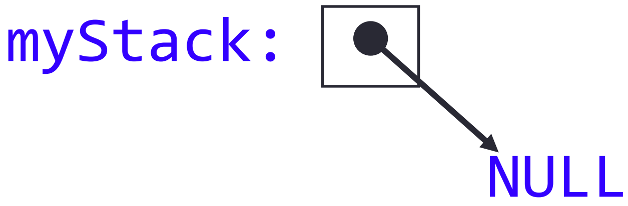
- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it
- The header of the list points to the top of the stack



- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

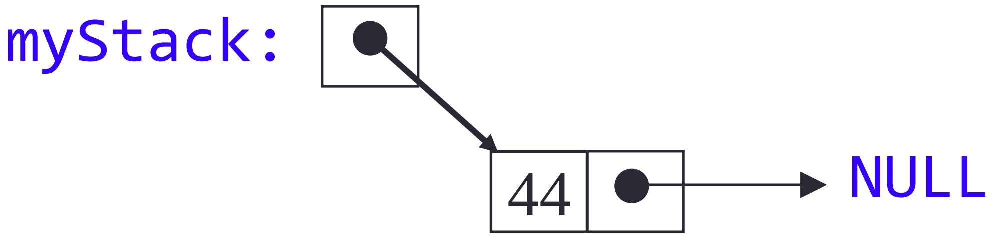
Stack using LL

- Initialize stack; set header to NULL



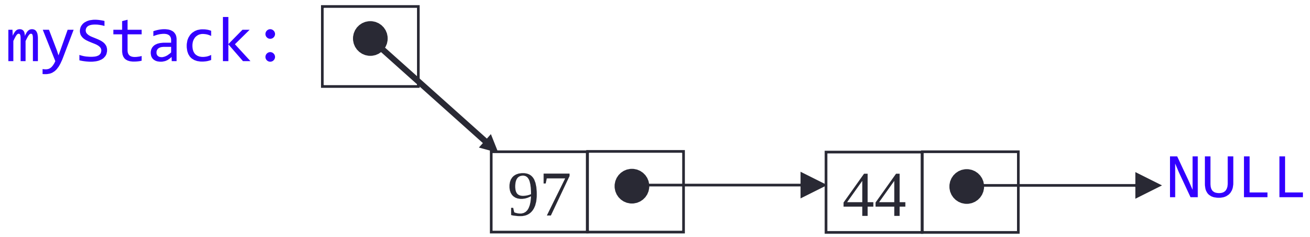
Stack using LL

- Initialize stack; set header to NULL
- push(44)



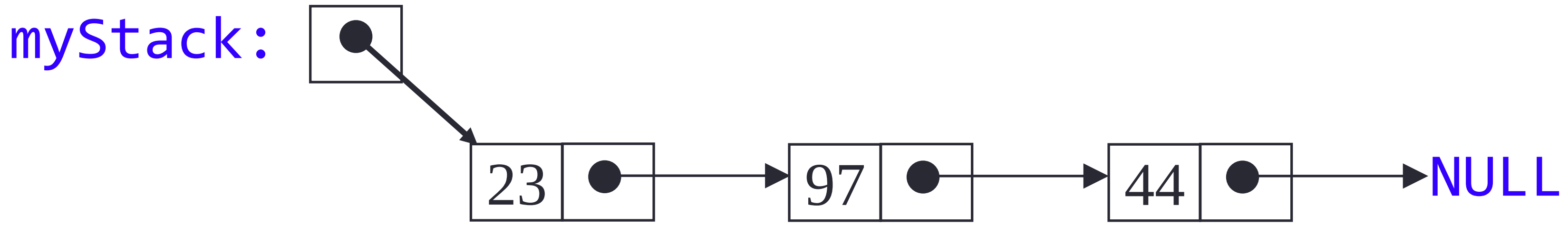
Stack using LL

- Initialize stack; set header to NULL
- push(44)
- push(97)



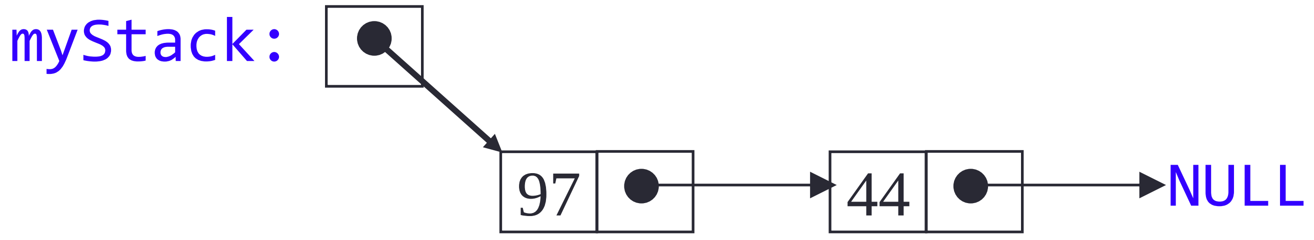
Stack using LL

- Initialize stack; set header to NULL
- push(44)
- push(97)
- push(23)



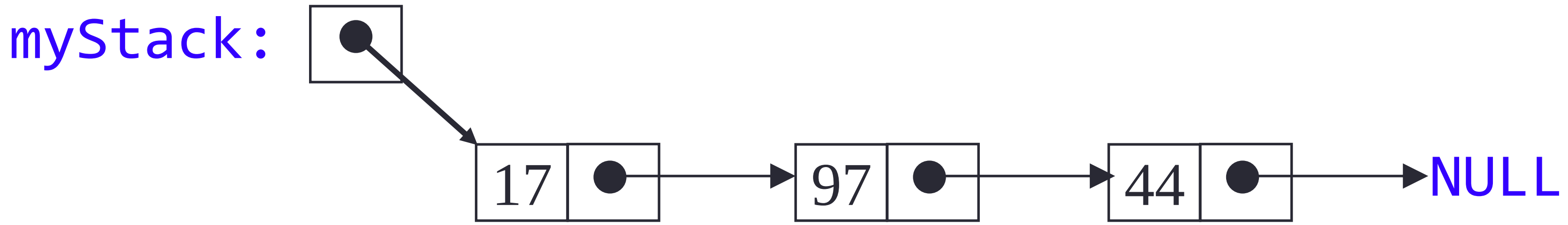
Stack using LL

- Initialize stack; set header to NULL
- push(44)
- push(97)
- push(23)
- pop()



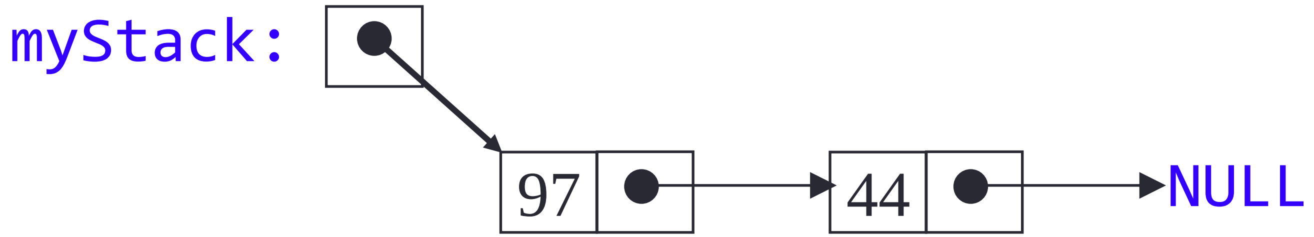
Stack using LL

- Initialize stack; set header to NULL
- push(44)
- push(97)
- push(23)
- pop()
- push(17)



Stack using LL

- Initialize stack; set header to NULL
- push(44)
- push(97)
- push(23)
- pop()
- push(17)
- pop()



Stack using LL details

- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is popped from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to `NULL`
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the linked list

Queue using LL

- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are O(1), but at the other end they are O(n)
  - Because you have to find the last element each time
- In order to implement both insertions and deletions in O(1) time in a SLL
  - You always need a pointer to the first thing in the list
  - You can keep an additional pointer to the *last* thing in the list

Queue using LL

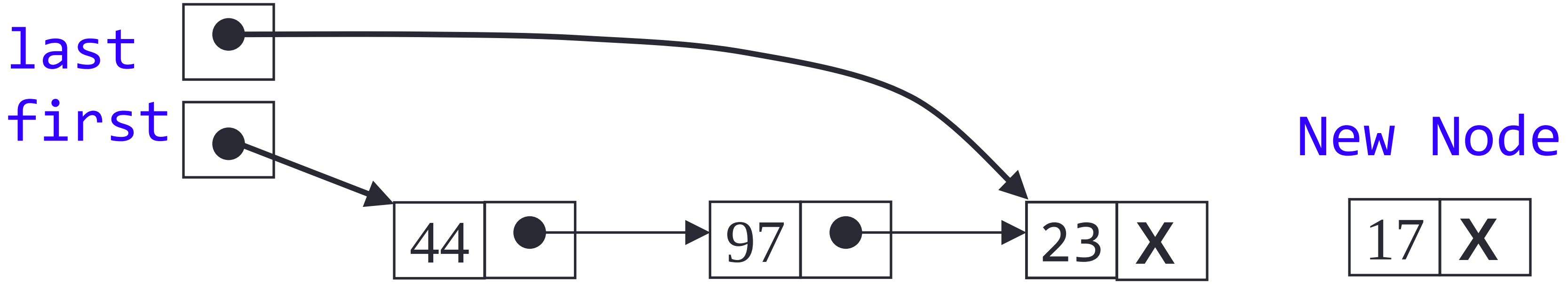
- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way

- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it

- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue
  - Keep pointers to *both* the front and the rear of the SLL

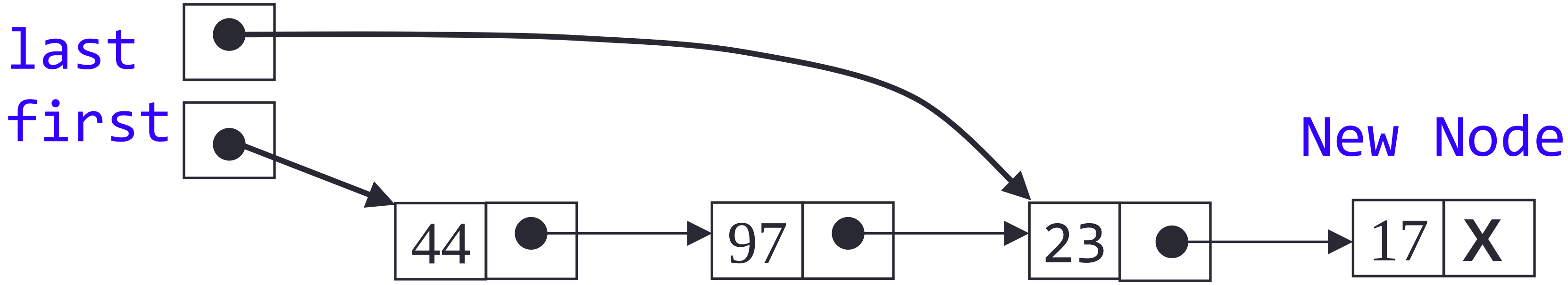
Enqueueing a Node

- Create a new node



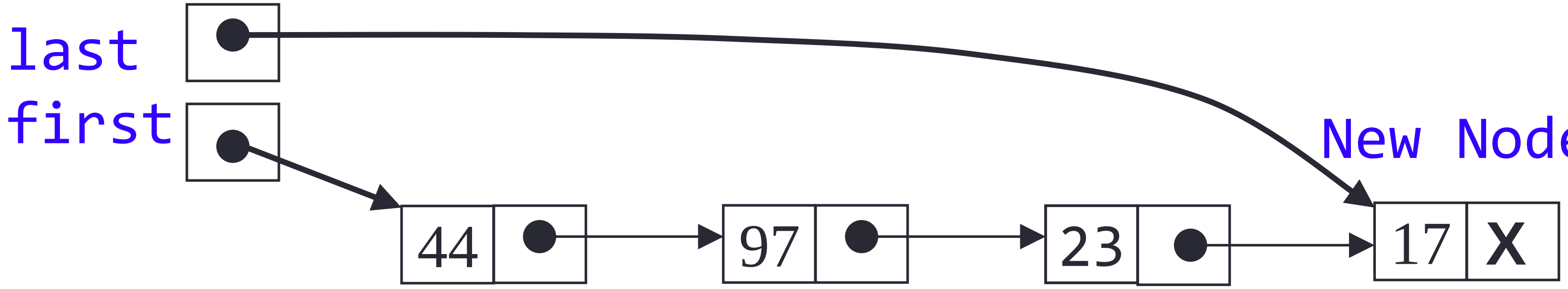
Enqueueing a Node

- Create a new node
- Change pointer of last node to point new node



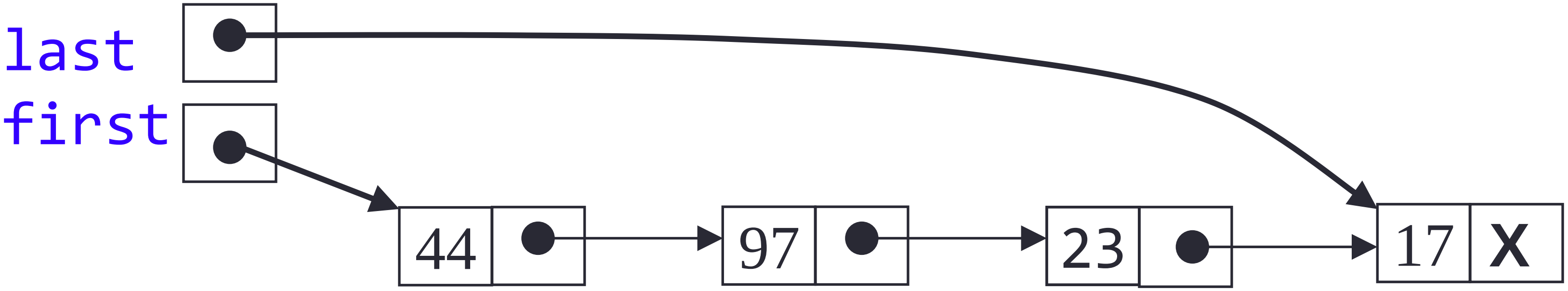
Enqueueing a Node

- Create a new node
- Change pointer of last node to point new node
- Change *last* pointer to point to new node



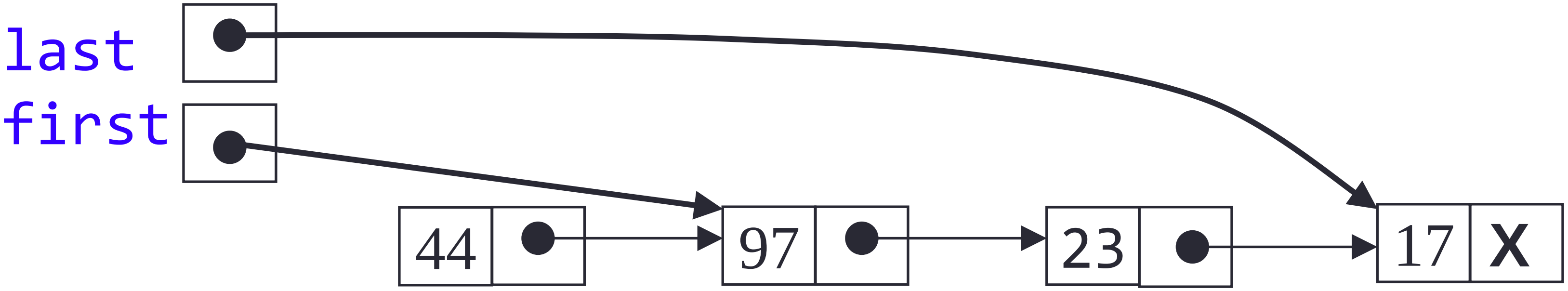
Dequeueing a Node

- Change "first" to point to second node



Dequeueing a Node

- Change “first” to point to second node
- Optionally, set deleted node free



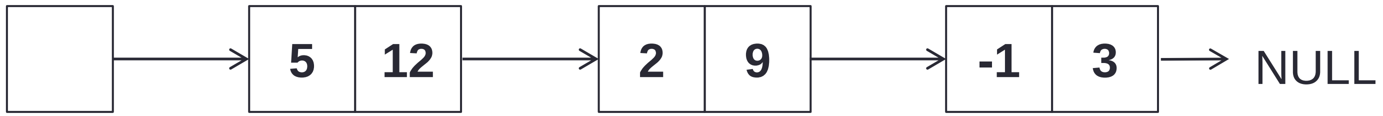
Queue using LL details

- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is dequeued from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to `NULL`
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the linked list

Polynomial Representation in LL

- Represent polynomial expression using Linked List
- A node in linked list stores coefficient and exponent of each term in polynomial expression

- Eg. $5X^{12} + 2X^9 - X^3$



Polynomial Addition

- Consider Two polynomial expressions

$$5X^{12} + 2X^{9} - X^{3}$$

$$5X^{11} - 4X^{9} + 2X^{3} - \mathrm{X}$$

- Addition of above two expression is

$$5X^{12} + 5X^{11} - 2X^{9} + X^{3} - X$$

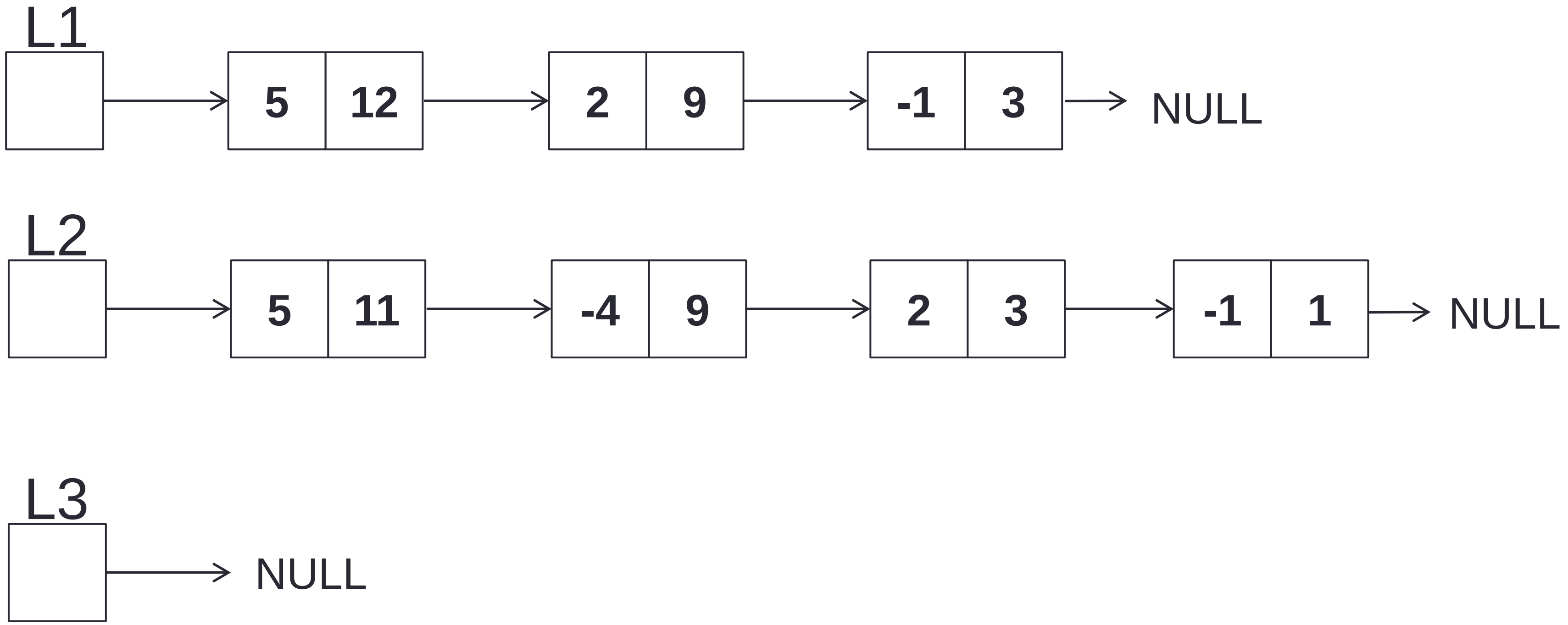
Algo for Polynomial Addition

- Represent two polynomials in two linked lists L1 and L2
- Create a third empty linked list L3
- Compare the items in L1 with the items in L2
  - If there is no item having the same exponent, append these items to the third list.
  - If there are two items with the same exponent *exp* and coefficient coff1 and coff2, append an item with exponent *exp* and coefficient *coff1+coff2* to L3

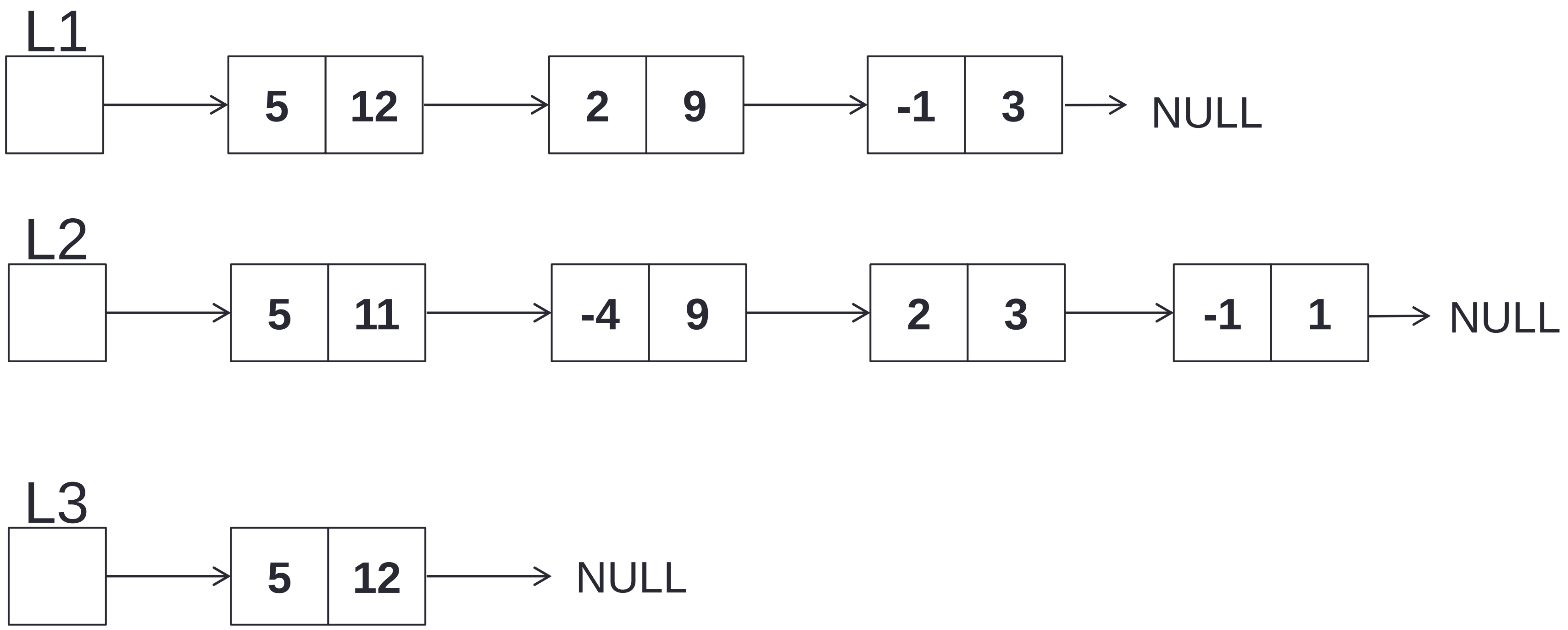
Pseudocode for Polynomial Addition

```
Polyadd(list L1, list L2, list L3)
   while (L1 != NULL and L2 != NULL)  //While both list are not empty
      if (L1->pow > L2->pow)  //if power of L1 is greater than L2, append L1 to L3
         L3->coff = L1->coff; L3->pow = L1->pow;
         L1 = L1->next;
      if (L1->pow < L2->pow)  //if power of L2 is greater than L1, append L1 to L3
         L3->coff = L2->coff; L3->pow = L2->pow;
         L2 = L2->next;
      if (L1->pow = L2->pow)  //if power of L1 & L2 is equal, add coefficient
         L3->coff = L1->coff + L2->coff; L3->pow = L1->pow;
         L1 = L1->next; L2= L2->next;

   while (L1 != NULL)  //if L2 has reached end, append remaining terms of L1
      L3->coff = L1->coff; L3->pow = L1->pow; L1 = L1->next;

   while (L2 != NULL)  //if L1 has reached end, append remaining terms of L2
      L3->coff = L2->coff; L3->pow = L2->pow; L1 = L2->next;
```

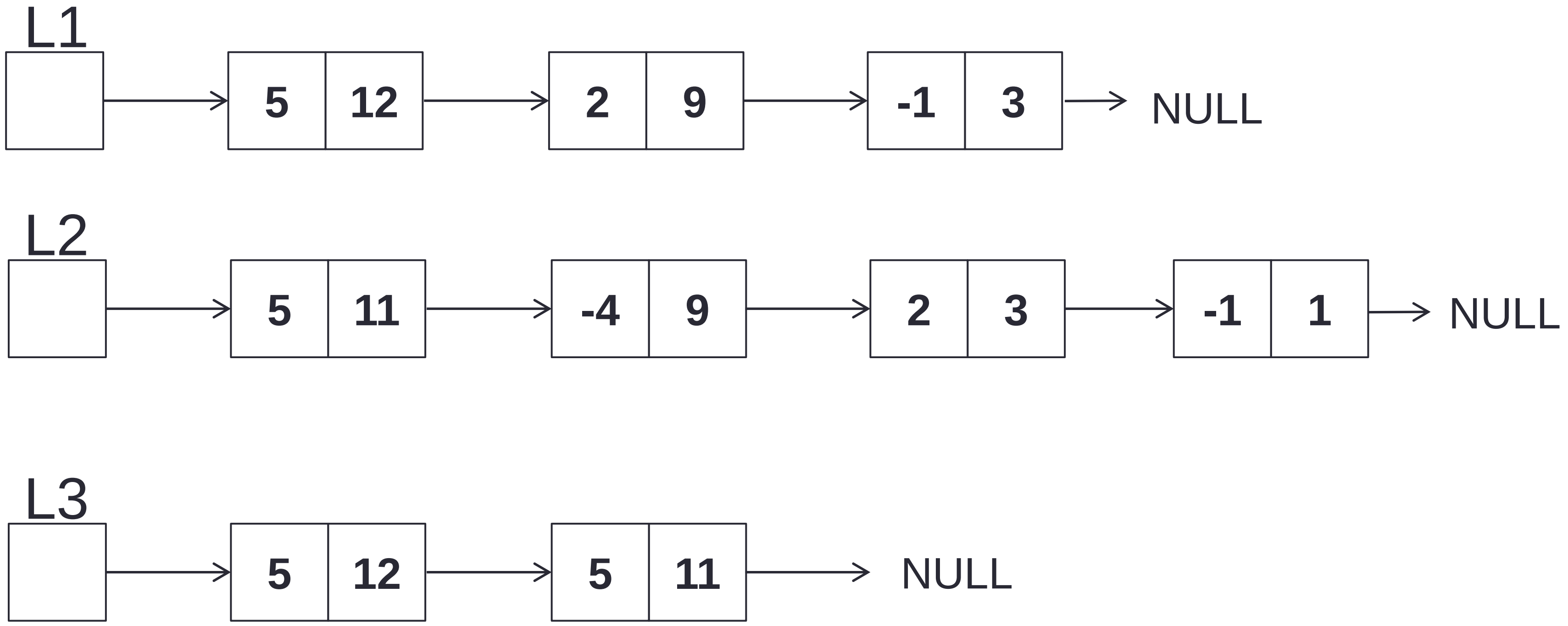
Example Polynomial Addition



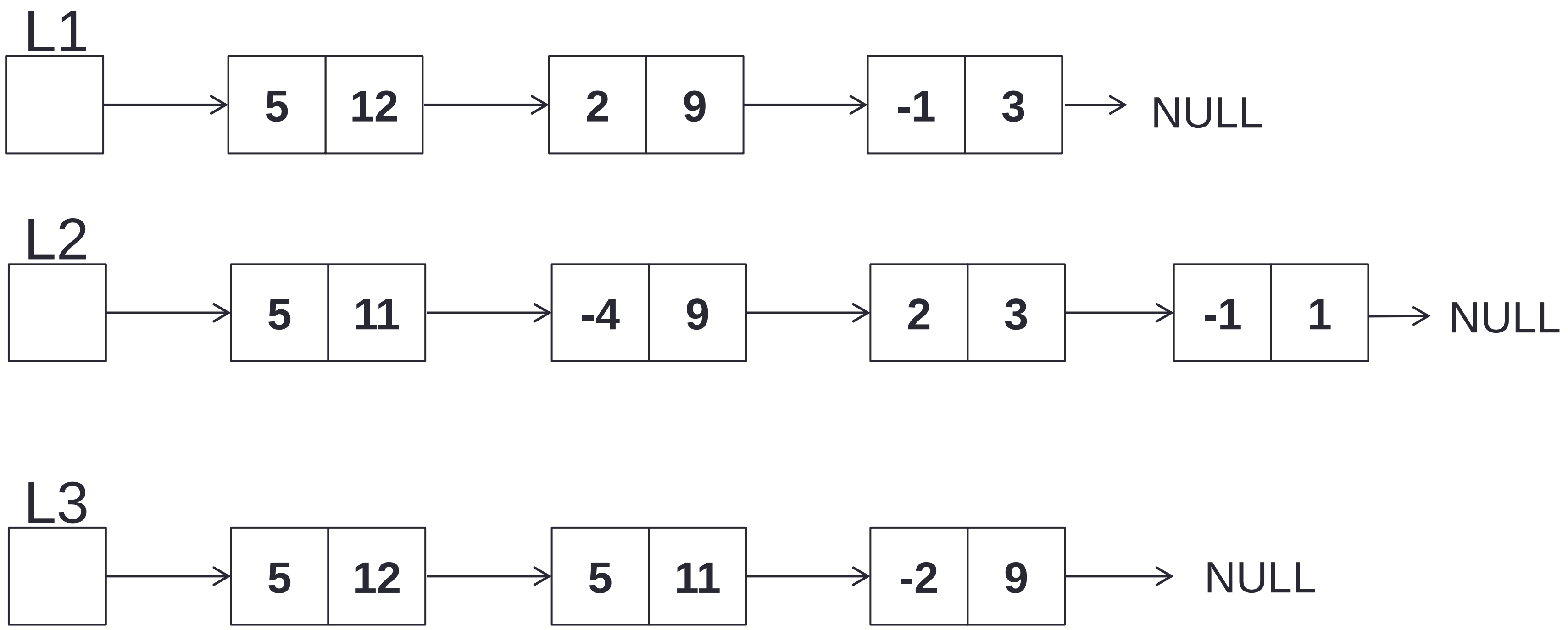
Example Polynomial Addition



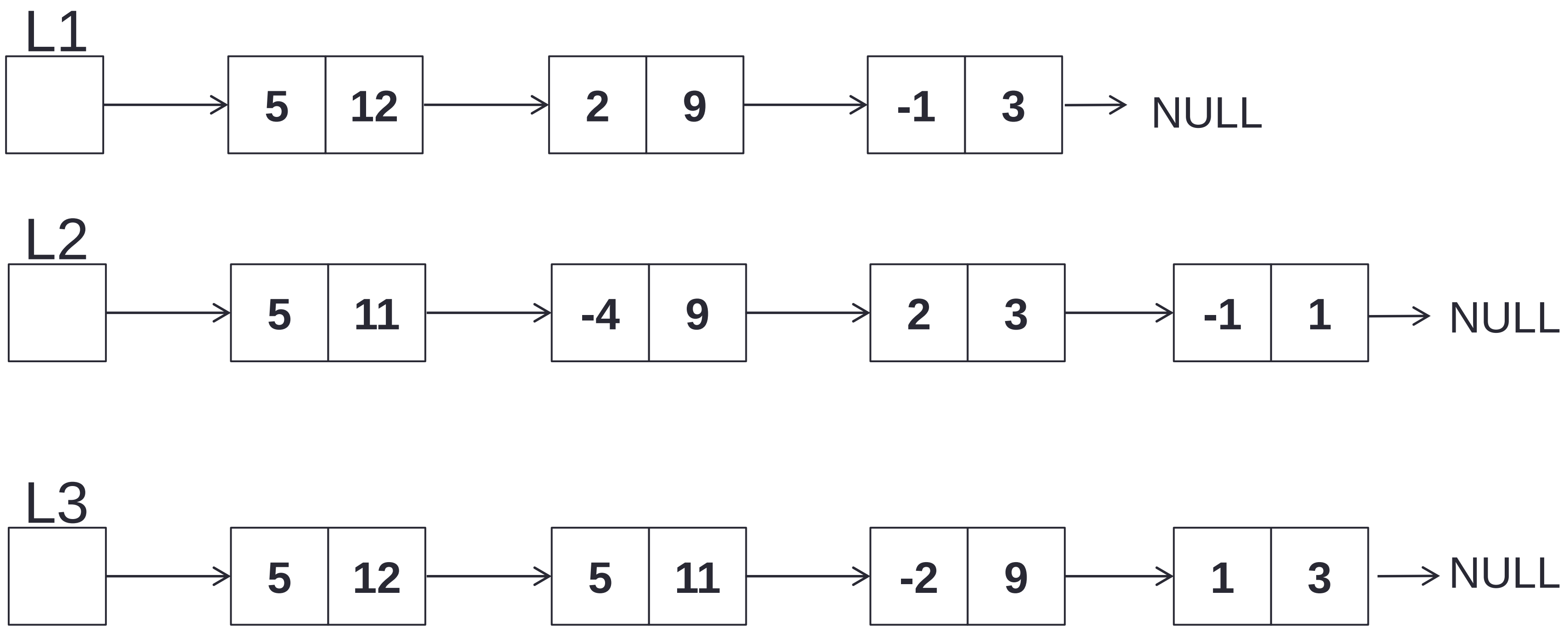
Example Polynomial Addition



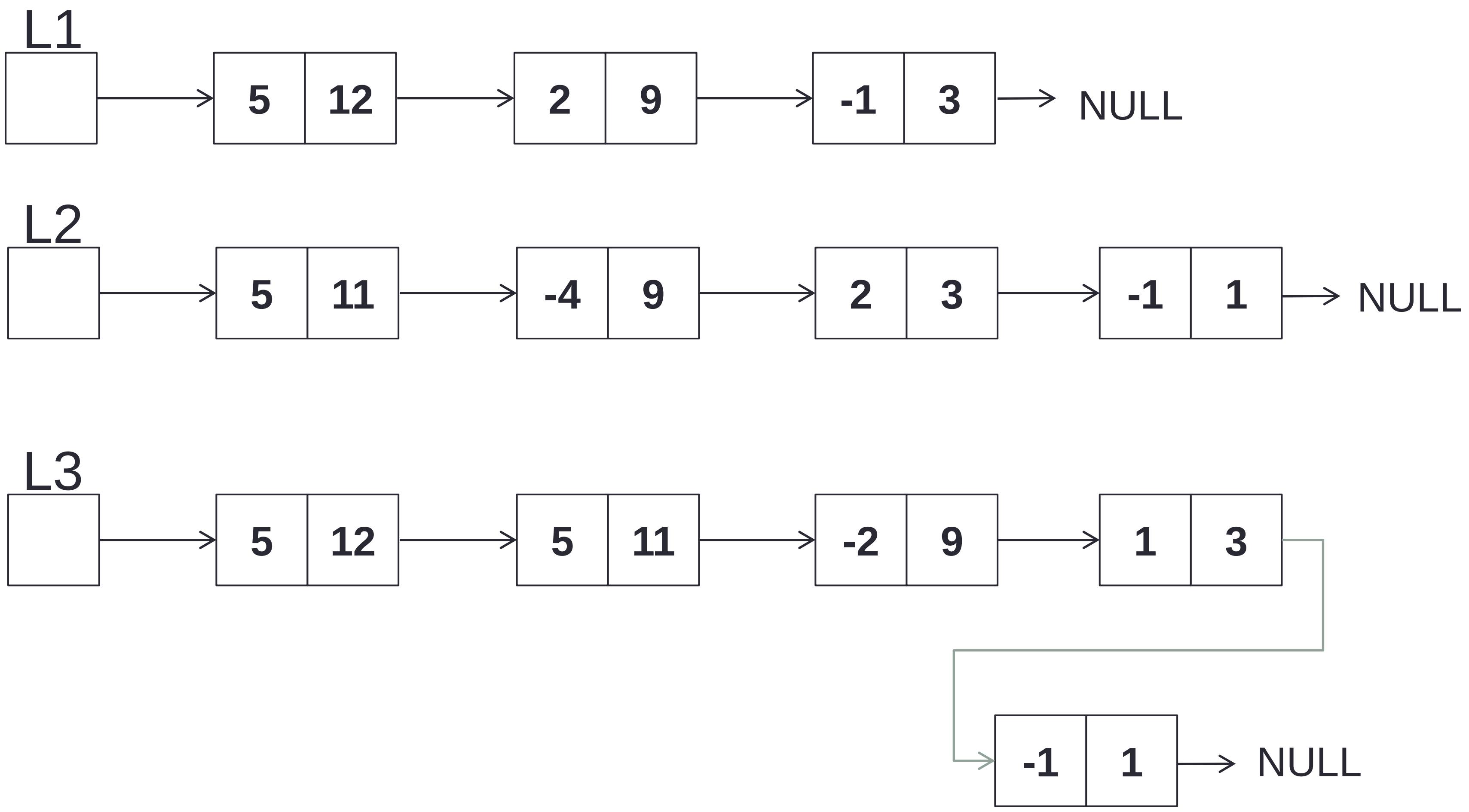
Example Polynomial Addition



Example Polynomial Addition



Example Polynomial Addition



Algo for Polynomial Subtraction

- Represent two polynomials in two linked lists L1 and L2
- Create a third linked list L3, with coefficient of L2 negated
- Perform the algo for addition of list L1 and L3

Sparse Matrices

- Sparse: Many elements are zero
- Dense: Many elements are non-zero

- A matrix is a collection of relation between two entities of same of different groups
- Eg. Airline Flight Matrix between cities

$$
\begin{array}{ll}
\text{city} \rightarrow & \\
\downarrow & \begin{array}{ccccc}
0 & 0 & 3 & 0 & 4 \\
0 & 0 & 5 & 7 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 2 & 6 & 0 & 0 \\
4 & 1 & 0 & 3 & 0
\end{array}
\end{array}
$$

Structured Sparse Matrices

- Has a proper structure of zero and non-zero elements
- Eg. Diagonal, tridiagonal, lower triangular

Diagonal	Tridiagonal
a_1 0 0 0	a_1 b_1 0 0
0 a_2 0 0	c_1 a_2 b_2 0
0 0 a_3 0	0 c_2 a_3 b_3
0 0 0 a_4	0 0 c_3 a_4

- May be mapped into a 1D array so that a mapping function can be used to locate an element.

Unstructured Sparse Matrices

- Airline flight matrix.
  - airports are numbered 1 through n
  - flight(i, j) = **list** of nonstop flights from airport i to airport j
  - n = 1000 (say)
  - n x n array of list references (assuming 4Bytes for one reference) => 4 million bytes
  - total number of flights = 20,000 (say)
  - need at most 20,000 list references => at most 80,000 bytes

Representation of Unstructured Sparse Matrix(USM)

- Single linear list in row-major order.
  - scan the nonzero elements of the sparse matrix in row-major order
  - each nonzero element is represented by a triple (row, column, value)
  - the list of triples may be an array list or a linked list (chain)

Example:

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0

Array Representation of USM

Example:

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

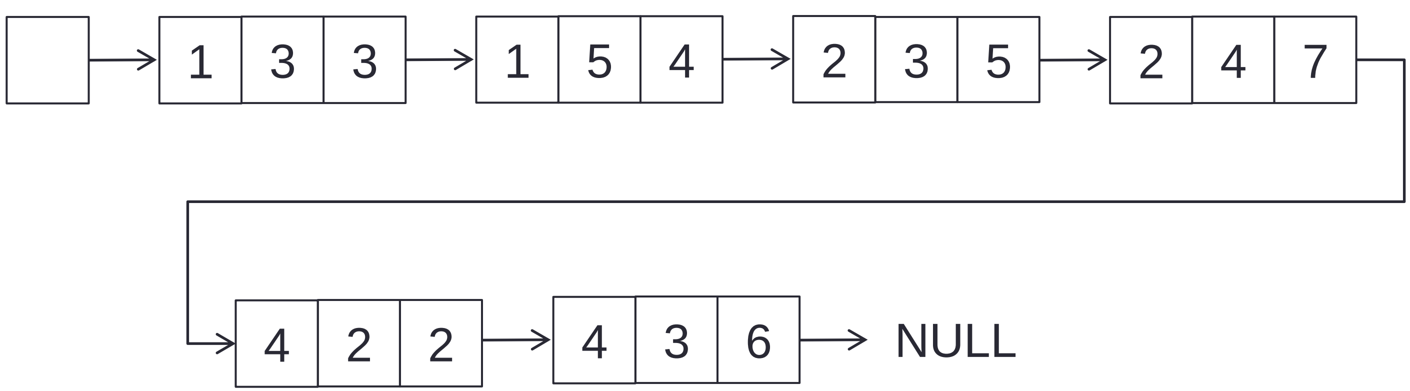
$$\begin{array}{l c} \text{Element} & \begin{matrix}0 & 1 & 2 & 3 & 4 & 5\end{matrix} \\ \begin{matrix}\text{Row} \\ \text{Column} \\ \text{Value}\end{matrix} & \begin{bmatrix}1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6\end{bmatrix}\end{array}$$

LL Representation of USM

Example:

$$\begin{matrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{matrix}$$

$$LL = \begin{array}{l} \text{Row} \\ \text{Column} \\ \text{Value} \end{array} \begin{bmatrix} 1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6 \end{bmatrix}$$



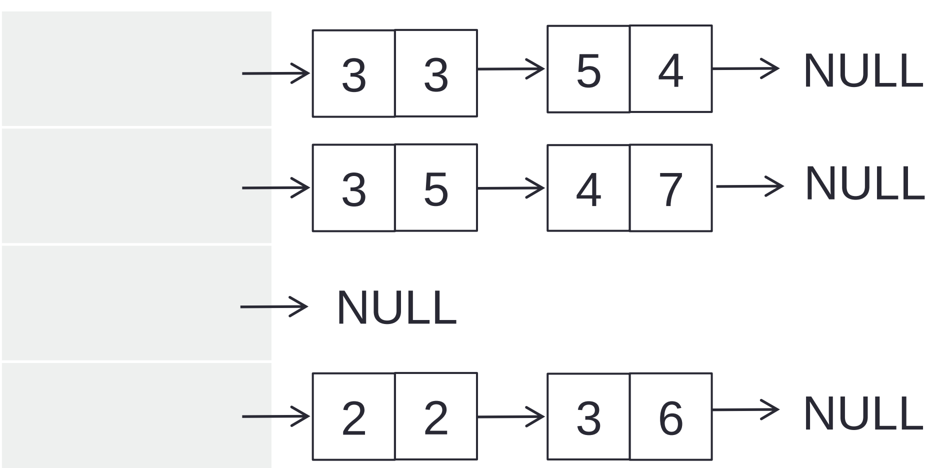
Array of LL Representation of USM

Example: 0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0



Row[]

Memory Requirements (Approx.)

- 500 x 500 matrix with 1994 nonzero elements

- 2D Array $\quad$ 500*500*4 = 10^6 Bytes
- Single Array List $\quad$ 3*1994*4 = 23,928 Bytes
- Array of LL $\quad$ 23,928 + 500*4 = 25,928 Bytes

Circular Linked List(CLL)

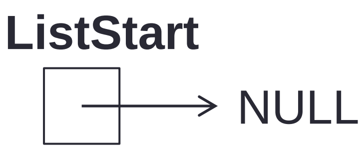
- Extension of Linear/Singly Linked List where last node points to beginning of list
- Can be implemented in two ways

1. **Header Pointer:** An extra pointer pointing to start of list
2. **Header Node:** A special node in the beginning of list
   - Header node stores some special value, like a negative number if list contains only positive numbers
   - We can use flag to specify header and non-header nodes

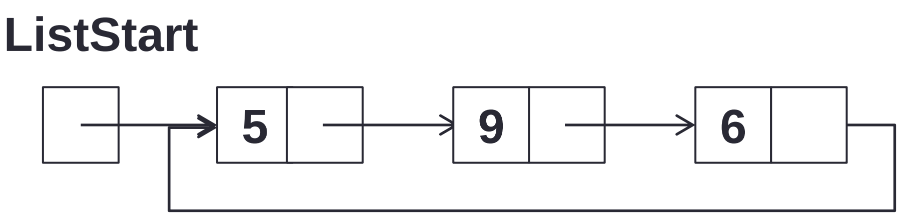
- CLL does not have any NULL pointer except for empty list

Circular Linked List (CLL)

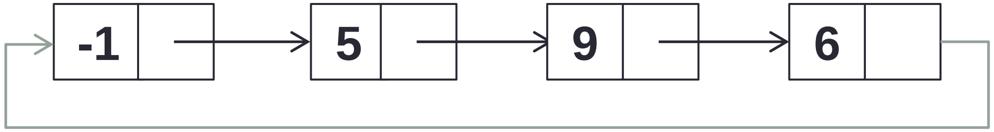
- Empty CLL



- Non-Empty CLL



- Header Node



Traversing in CLL

```
void print(Eptr ListStart) {
        Eptr temp = ListStart;
        print (temp->data);
        while (temp->next != ListStart) {
                print (temp->data);
                temp = temp->next;
        }
}
```

Advantages of CLL

- Some operations can be made efficient in CLL compare to Singly LL like search multiple entries subsequently
- CLL are useful when element of lists are to be visited in “Loop” fashion

Insertion in CLL

Insert in Empty CLL

- Change Header to point new node and new node to point itself

Insert at Start

- Create new node; point to existing first node
- Change last node to point to new node; change header

Insert in middle

- Change new node to point successor of the node after which node is inserted
- Change predecessor to point to new node

Deletion in CLL

Delete Last node

- If node points to itself, set header to point to null

Delete from Front

- Change last node to point second node in list
- Update header to point to second node

Delete from middle

- Change predecessor to point successor