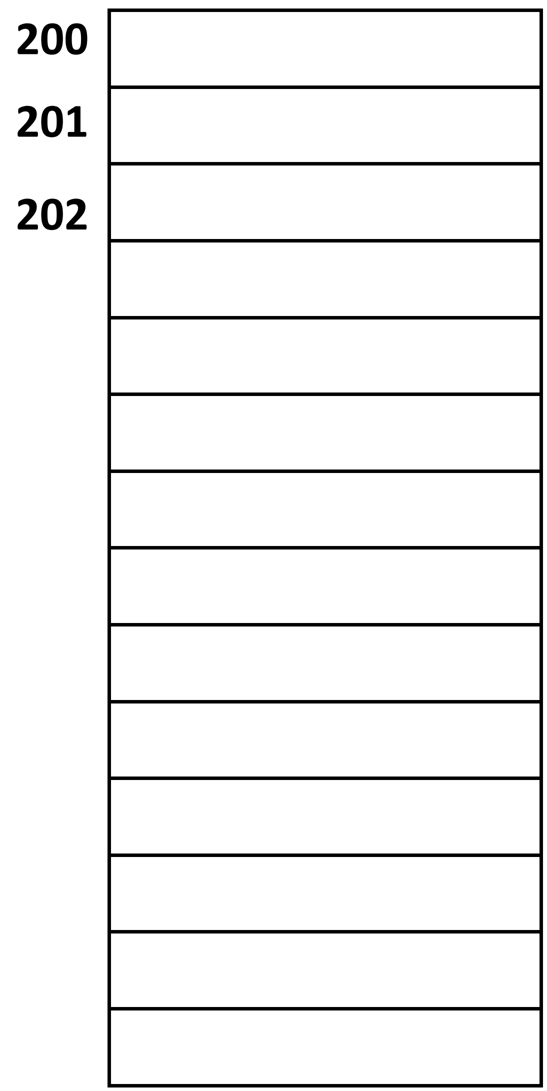
Linked List

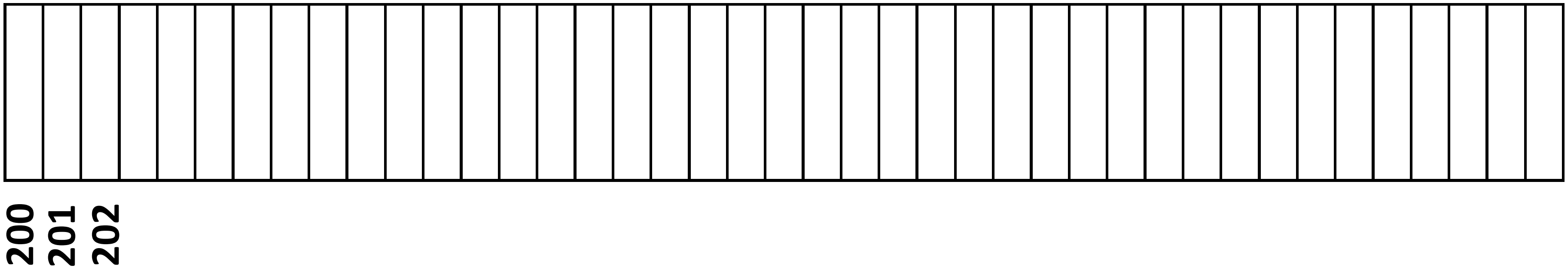
Limitations of Dynamic Array



Heap Memory

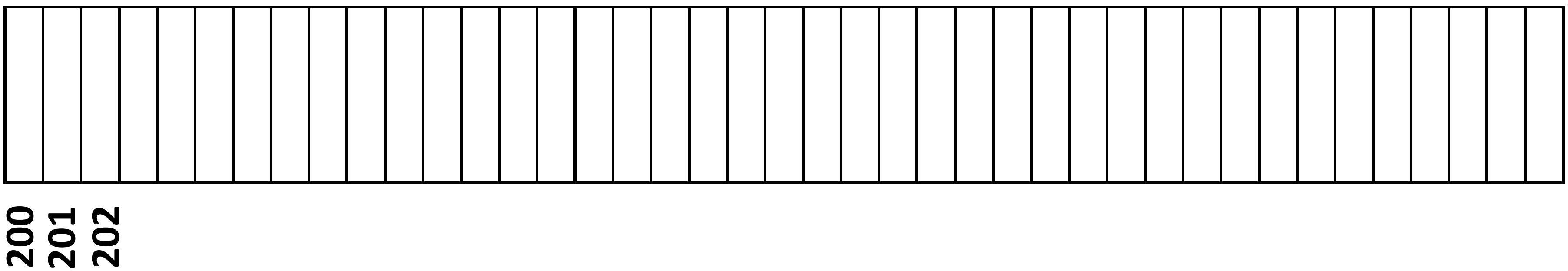
Limitations of Dynamic Array

Heap Memory (Horizontal View)



Limitations of Dynamic Array

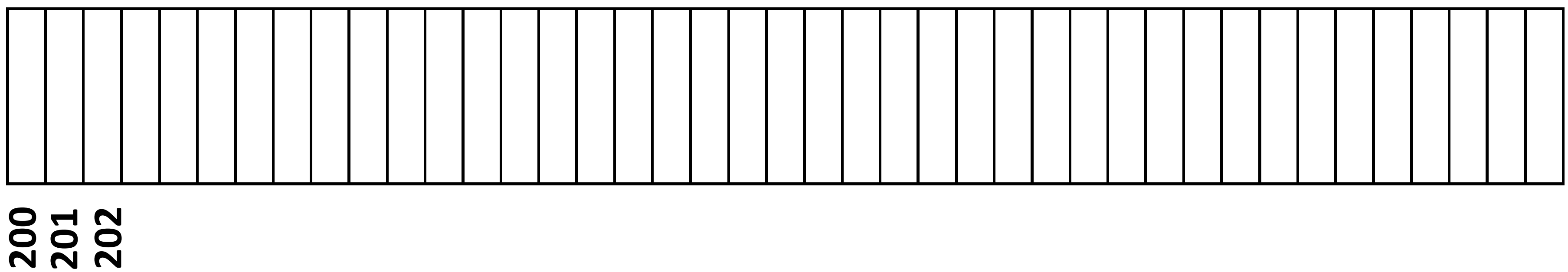
Heap Memory (Horizontal View)



- Memory Manager (A part of operating systems) manages the memory.
    - Keeps track of free space
    - Allocates space on request from the programs

Limitations of Dynamic Array

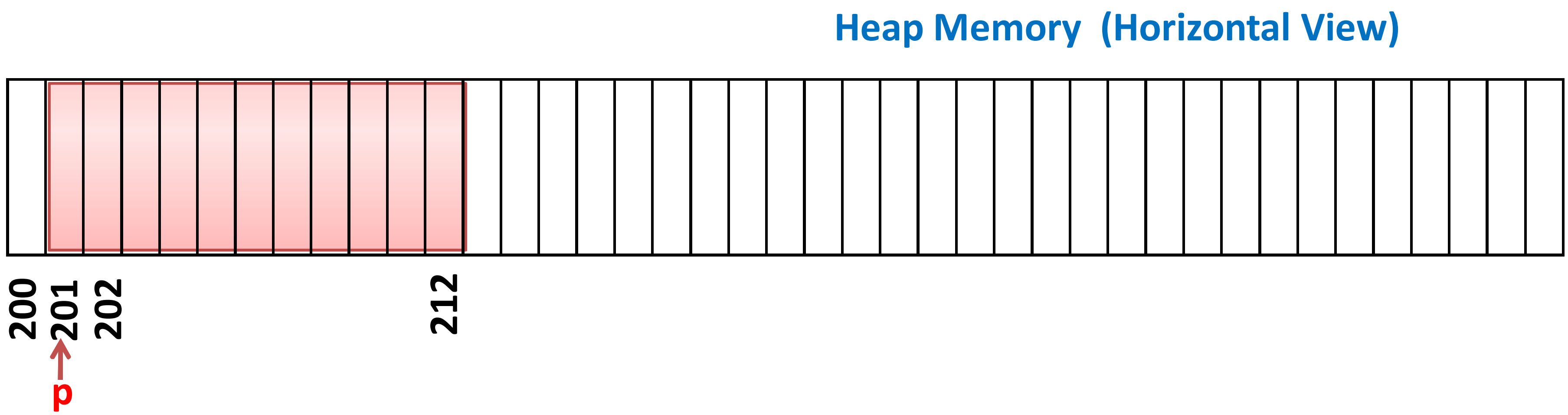
Heap Memory (Horizontal View)



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));
```

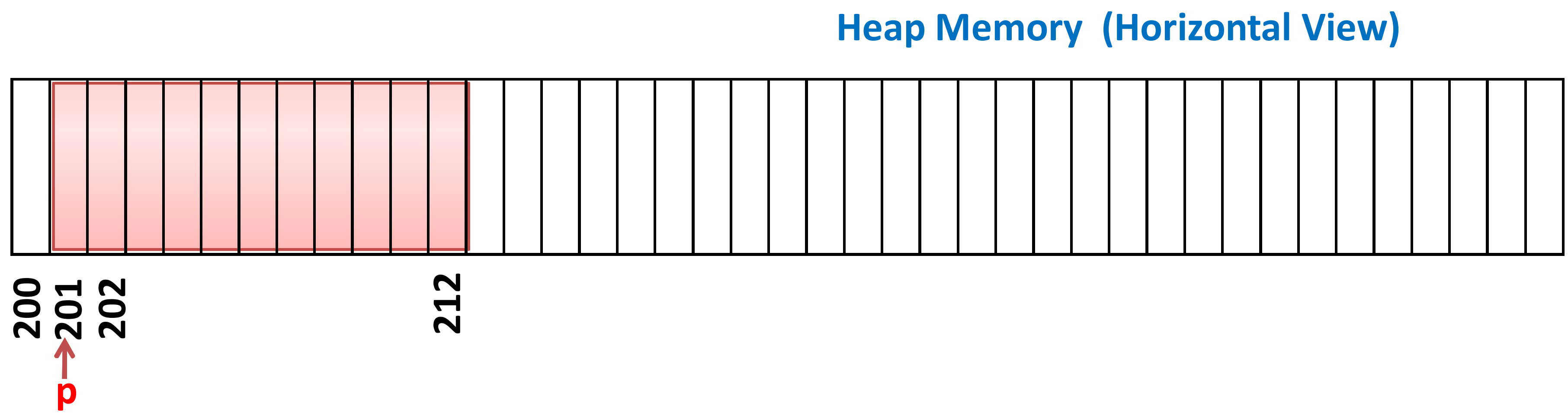
Limitations of Dynamic Array



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
```

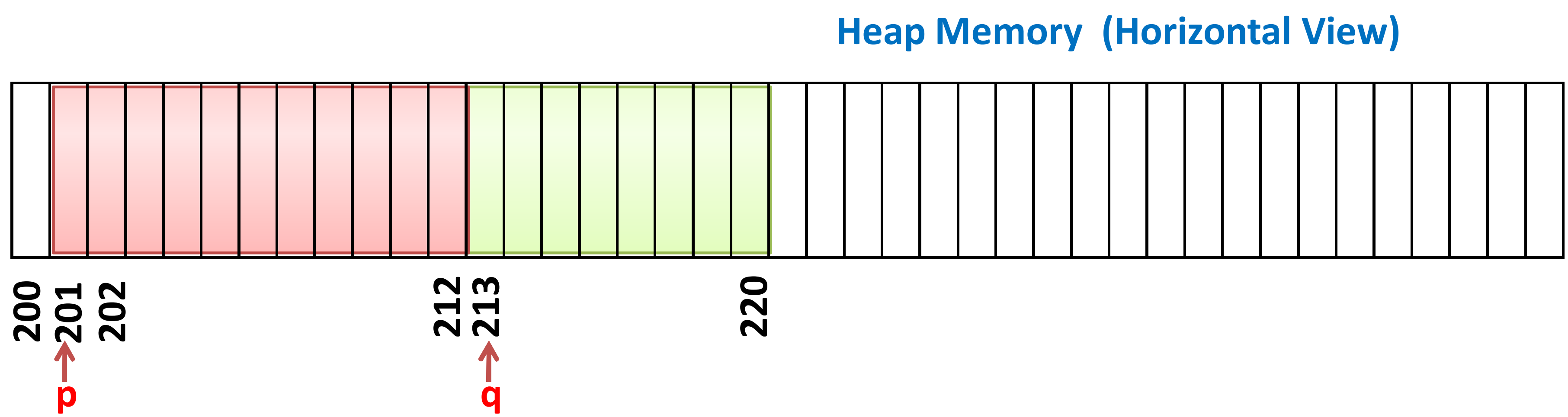
Limitations of Dynamic Array



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
float *q=(float*)malloc(sizeof(float));   // float size is 8 bytes
```

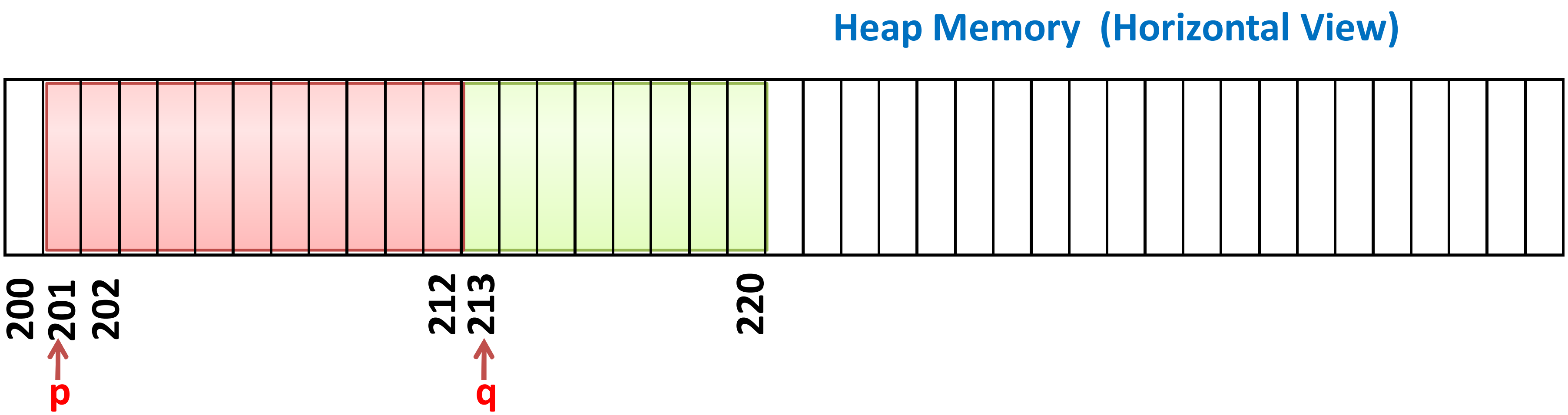
Limitations of Dynamic Array



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
float *q=(float*)malloc(sizeof(float));   // float size is 8 bytes
```

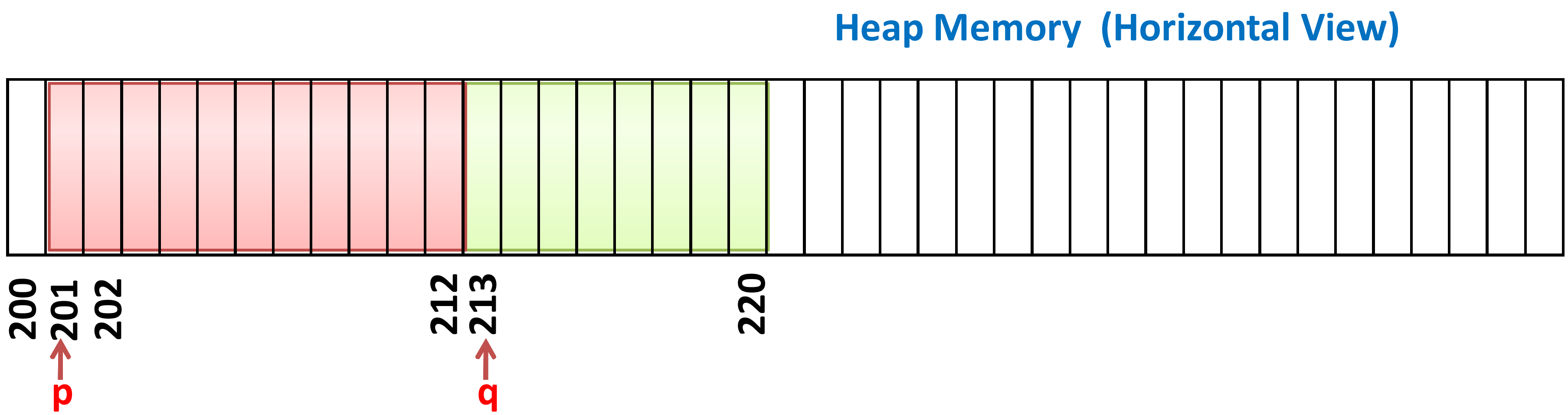
Limitations of Dynamic Array



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
float *q=(float*)malloc(sizeof(float));   // float size is 8 bytes
int *r=(int*)malloc(4*sizeof(int)); // int size is 4 bytes
```

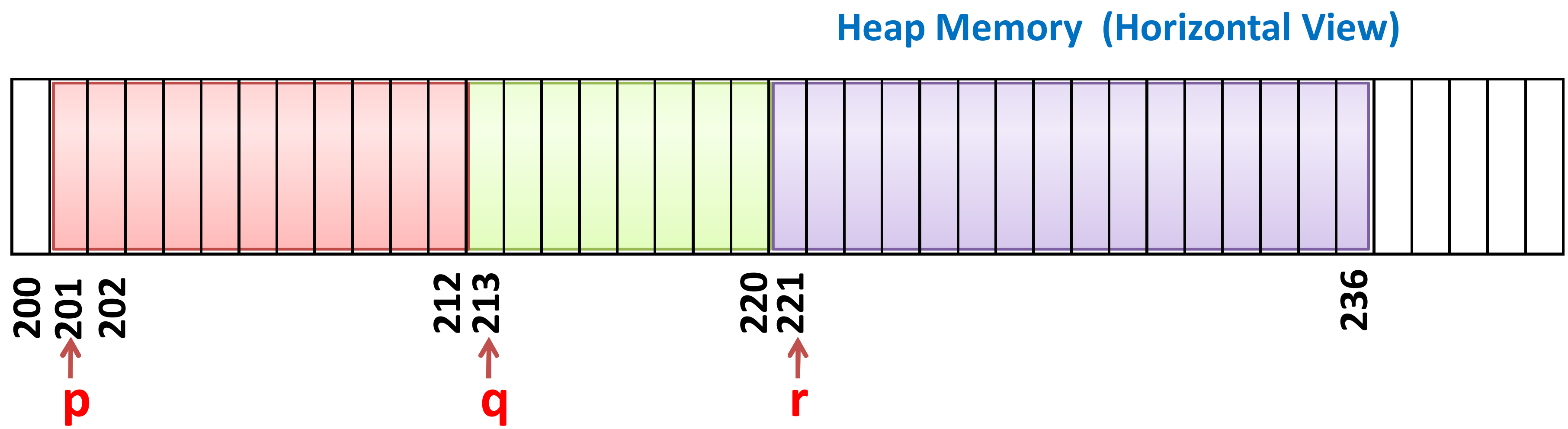
Limitations of Dynamic Array



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
float *q=(float*)malloc(sizeof(float));   // float size is 8 bytes
int *r=(int*)malloc(4*sizeof(int)); // int size is 4 bytes
```

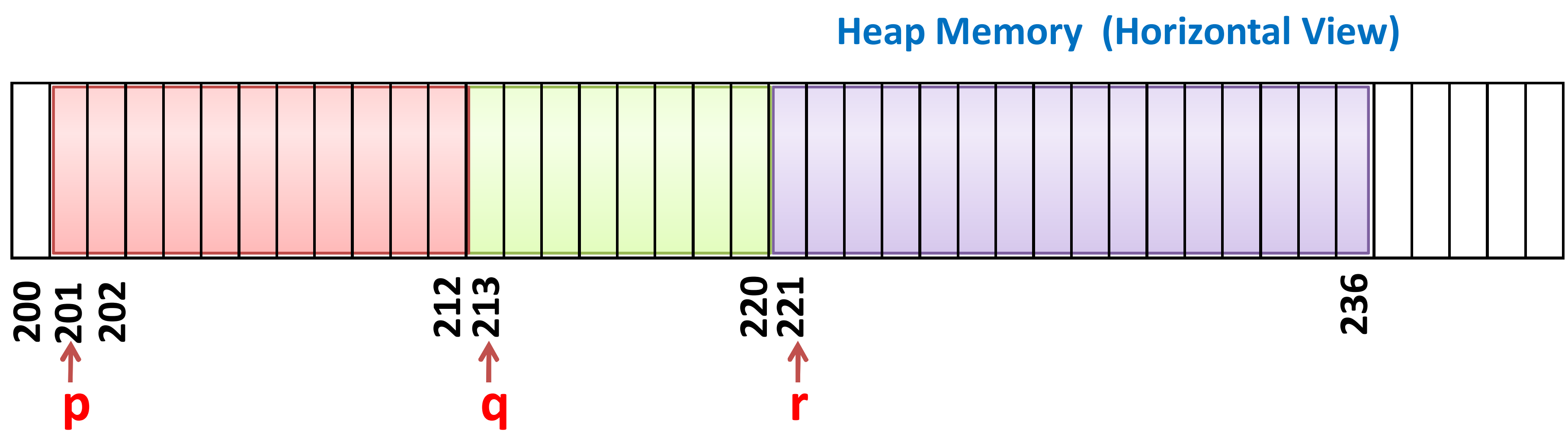
Limitations of Dynamic Array



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
float *q=(float*)malloc(sizeof(float));   // float size is 8 bytes
int *r=(int*)malloc(4*sizeof(int)); // int size is 4 bytes
```

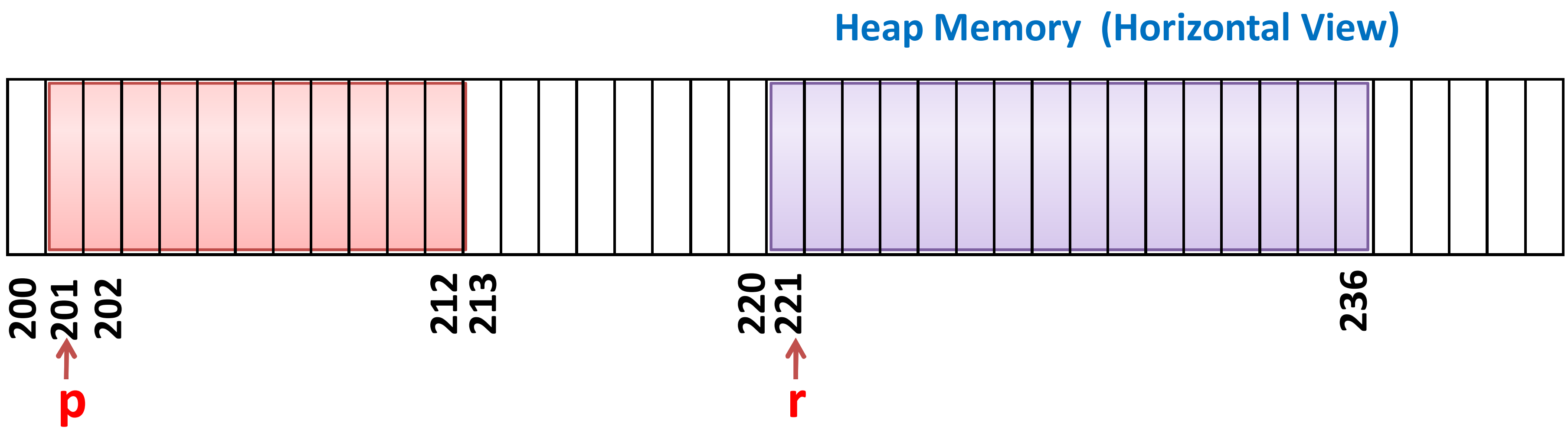
Limitations of Dynamic Array



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
float *q=(float*)malloc(sizeof(float));   // float size is 8 bytes
int *r=(int*)malloc(4*sizeof(int)); // int size is 4 bytes
free(q)
```

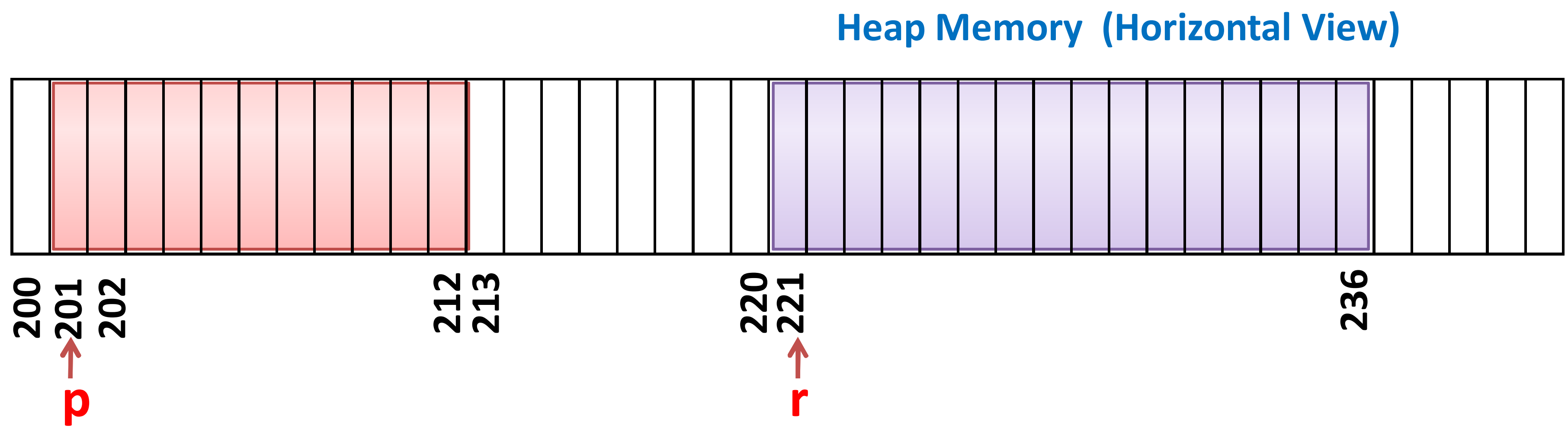
Limitations of Dynamic Array



Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
float *q=(float*)malloc(sizeof(float));   // float size is 8 bytes
int *r=(int*)malloc(4*sizeof(int)); // int size is 4 bytes
free(q)
```

Limitations of Dynamic Array

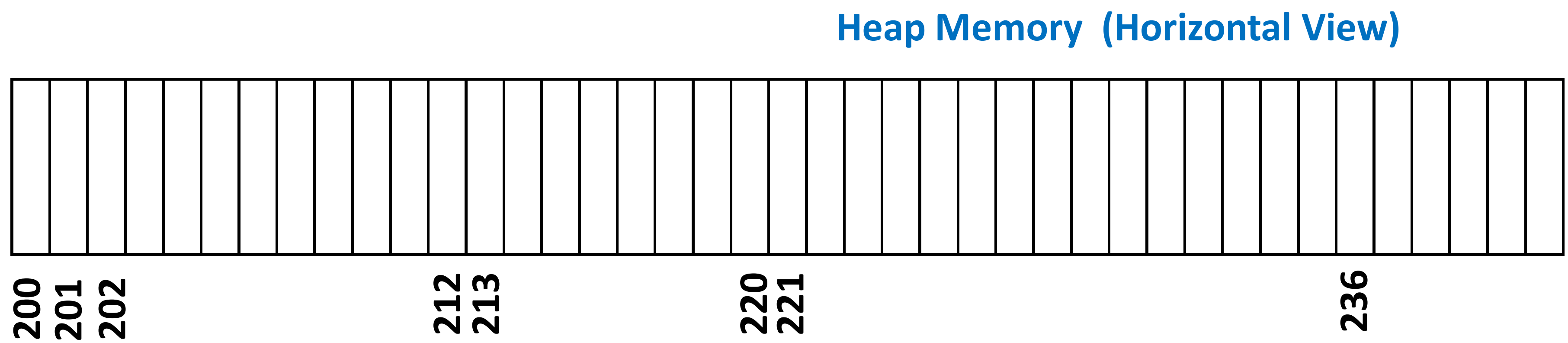


Consider a program segment:

```
int *p=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
float *q=(float*)malloc(sizeof(float));   // float size is 8 bytes
int *r=(int*)malloc(4*sizeof(int)); // int size is 4 bytes
free(q)
int *s=(int*)malloc(3*sizeof(int));   // int size is 4 bytes
```

Although free memory is more than required 3 x 4 = 12 bytes, yet it cannot be allocated as it is not contiguous. Returns a null pointer.

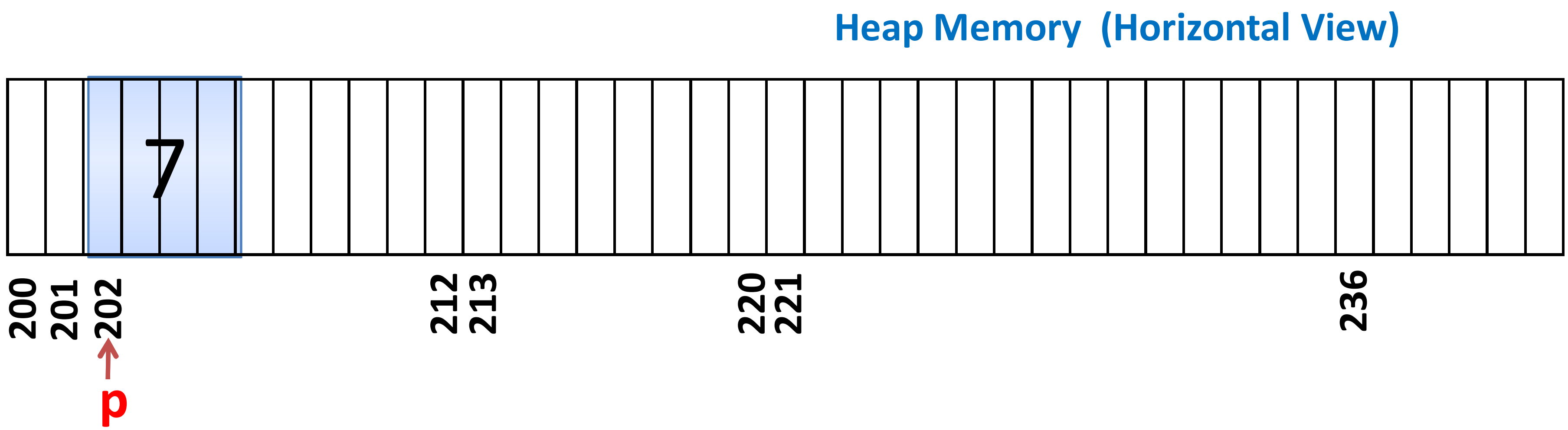
Solution: Linked List



Suppose we need to store a list of 4 integers: 7, 10, 5, 9.

Instead of asking memory manager for an integer array of 4 elements, these 4 integers can be stored one at a time in memory in different places.

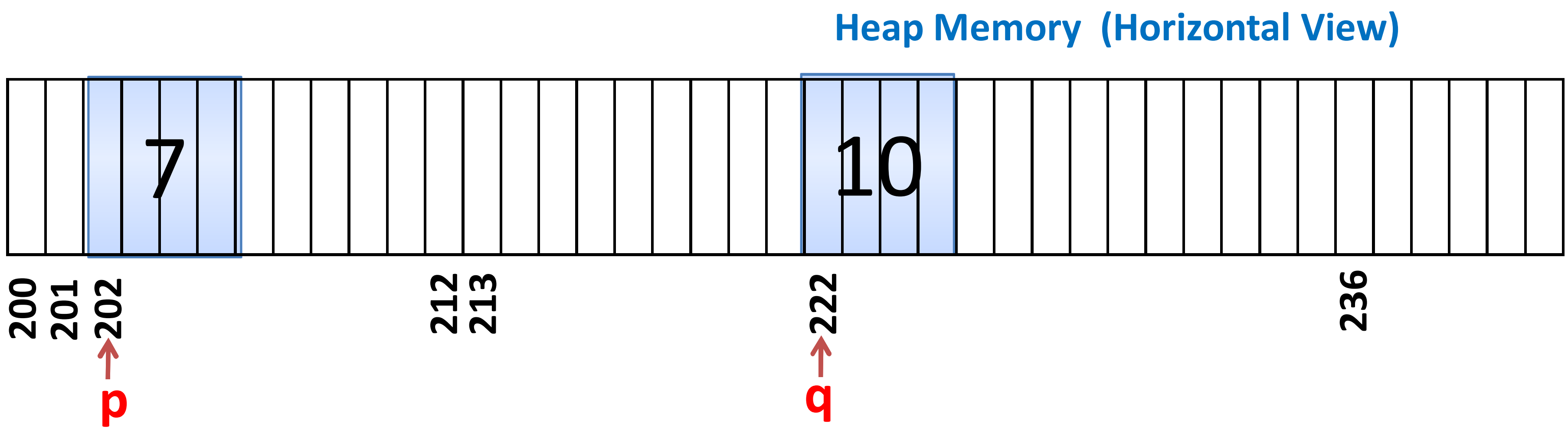
Solution: Linked List



Suppose we need to store a list of 4 integers: 7, 10, 5, 9.

```
int *p=(int*)malloc(sizeof(int));    *p=7;
```

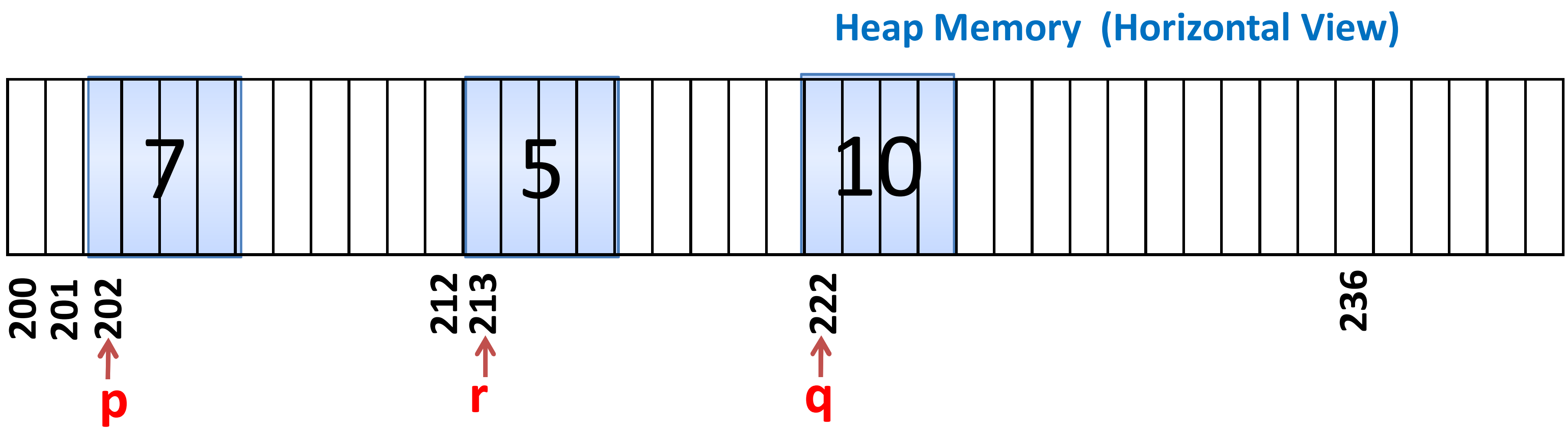
Solution: Linked List



Suppose we need to store a list of 4 integers: 7, 10, 5, 9.

```
int *p=(int*)malloc(sizeof(int));    *p=7;
int *q=(int*)malloc(sizeof(int));    *q=10;
```

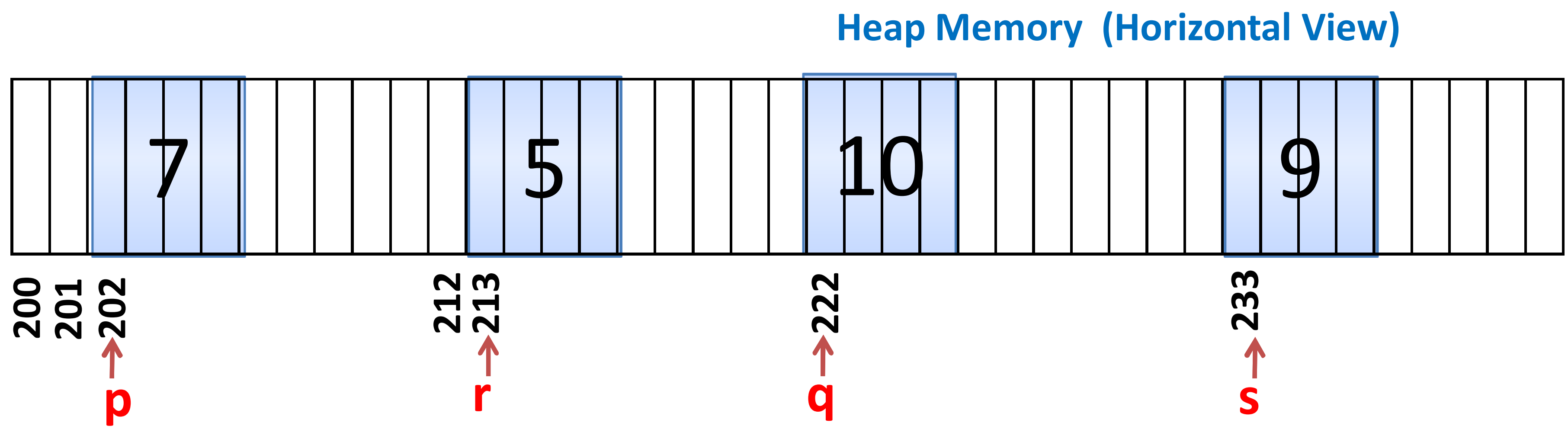
Solution: Linked List



Suppose we need to store a list of 4 integers: 7, 10, 5, 9.

```
int *p=(int*)malloc(sizeof(int));    *p=7;
int *q=(int*)malloc(sizeof(int));    *q=10;
int *r=(int*)malloc(sizeof(int));    *r=5;
```

Solution: Linked List

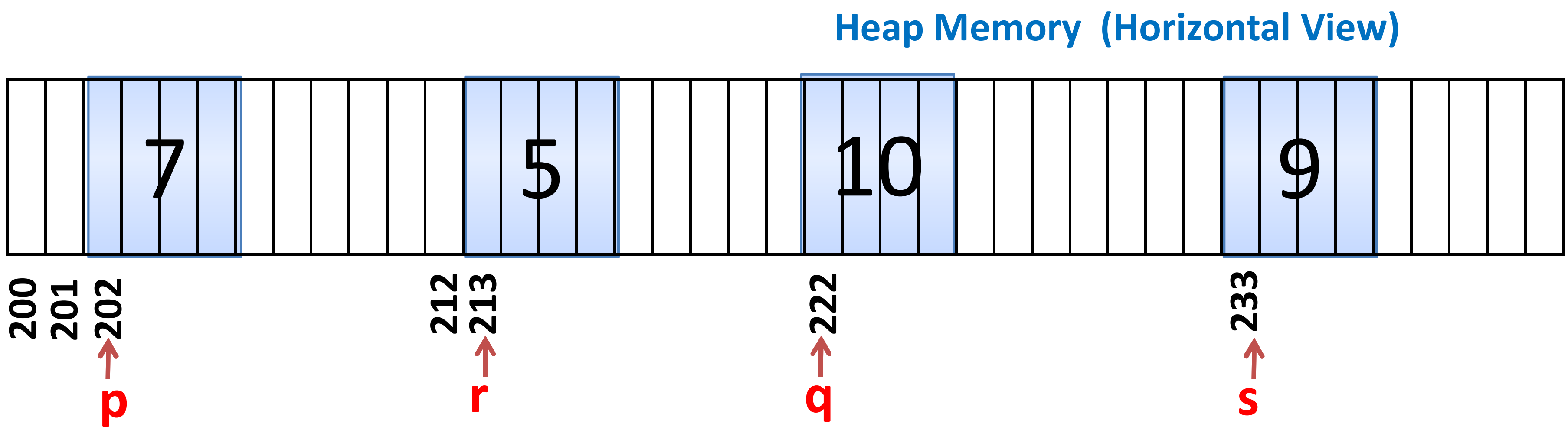


Suppose we need to store a list of 4 integers: 7, 10, 5, 9.

```
int *p=(int*)malloc(sizeof(int));    *p=7;
int *q=(int*)malloc(sizeof(int));    *q=10;
int *r=(int*)malloc(sizeof(int));    *r=5;
int *s=(int*)malloc(sizeof(int));    *s=9;
```

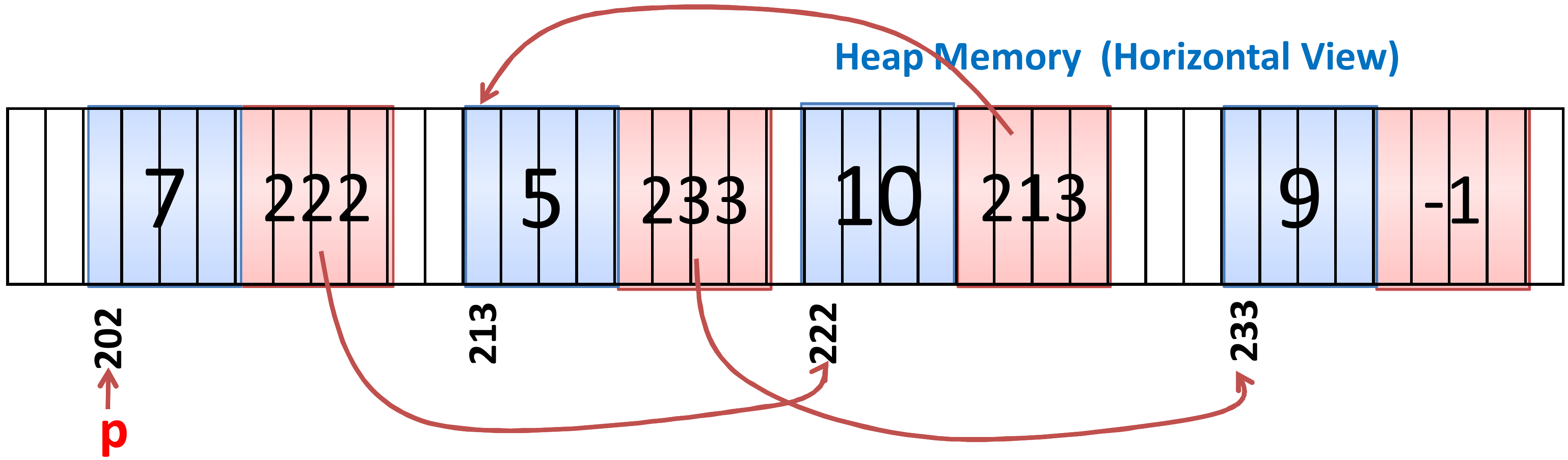
Non-contiguous allocation: Hence it is not possible to traverse the list directly as done in array.

Solution: Linked List



To traverse the elements of the list, one should store the address of the next element in the list with each element.

Solution: Linked List



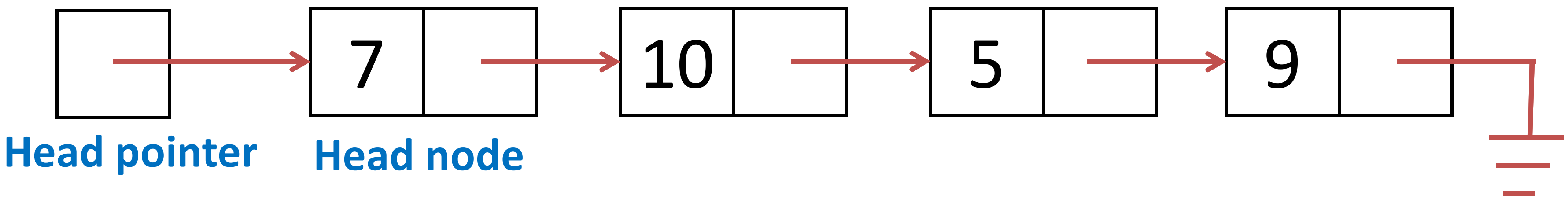
To traverse the elements of the list {7, 10, 5, 9}, one should store the address of the next element (**pointer to next element**) in the list with each element.

It is sufficient to know the address of (pointer to) the first node (also called as head node) to retrieve all the elements of the list.

The last node points to null (represented by -1 here).

Logical View of Singly Linked List

A linked list contains a sequence of nodes.



Each node contains a value.

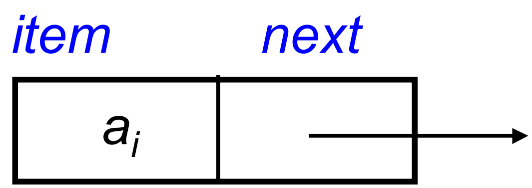
Each node contains the address (link) of the next node.

The last node contains a null link.

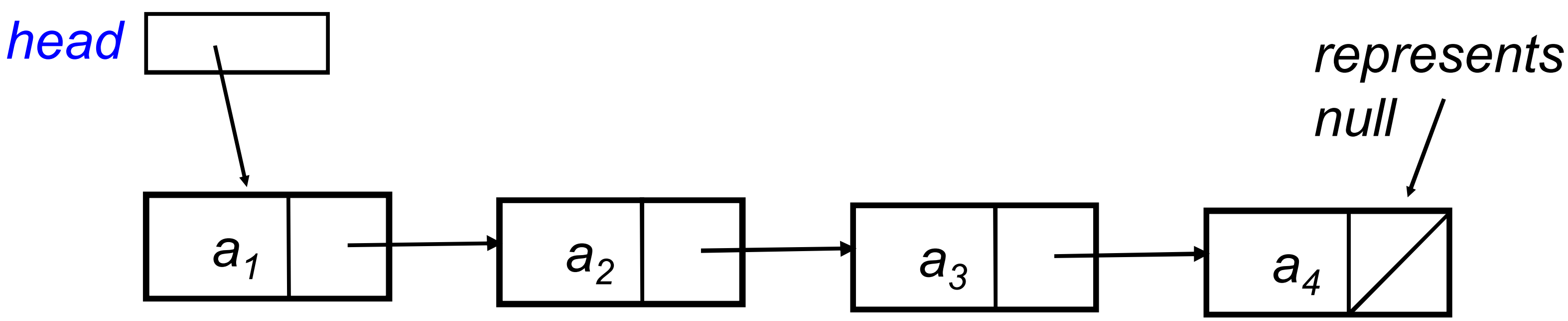
The list is identified by a head pointer (link to the head node)

Linked List Approach

- Main problem of array is the slow deletion/insertion since it has to shift items in its *contiguous* memory
- **Solution**: linked list where items need *not be contiguous* with nodes of the form



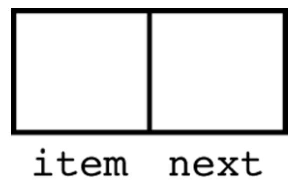
- Sequence (list) of four items < a_1, a_2, a_3, a_4 > can be represented by:



Pointer-Based Linked Lists

- A node in a linked list is usually a `struct`

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



A node

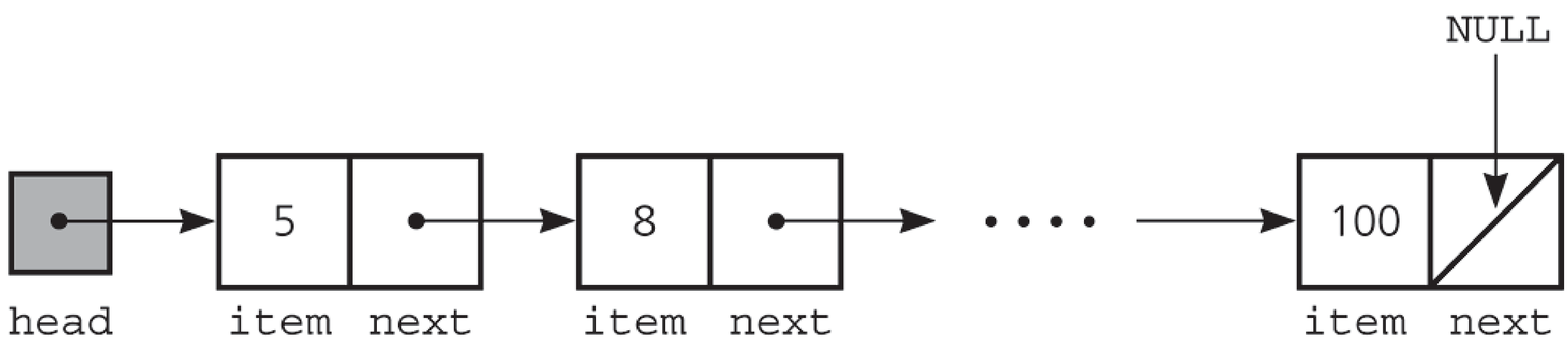
- A node is dynamically allocated

```
Node *p;
p = malloc(sizeof(Node));
```

Pointer-Based Linked Lists

- The head pointer points to the first node in a linked list
- If head is *NULL*, the linked list is empty
  - `head=NULL`
- `head=malloc(sizeof(Node))`

A Sample Linked List



Traverse a Linked List

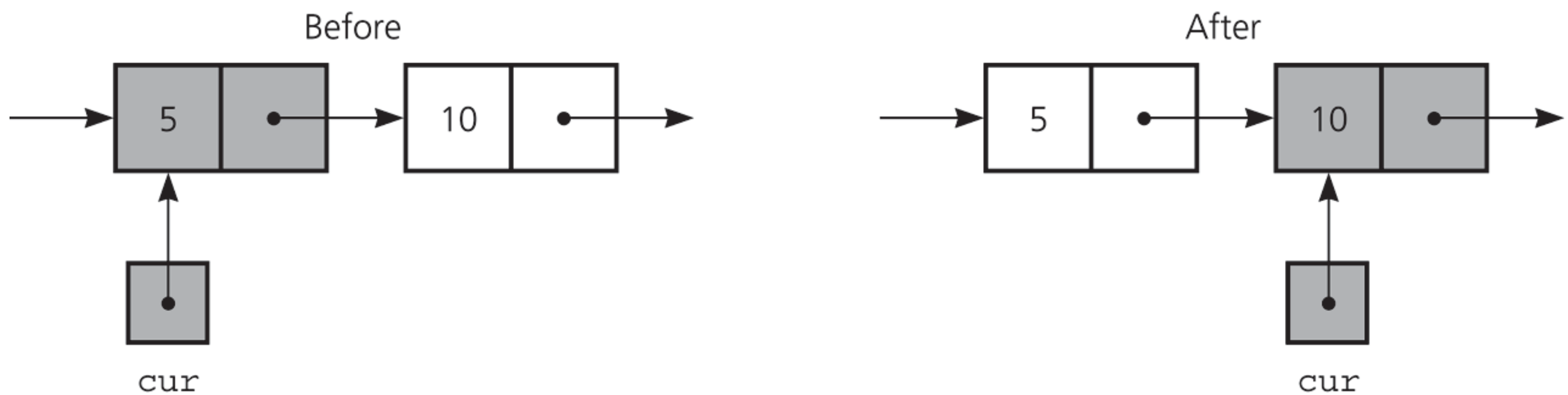
- Reference a node member with the -> operator

```
p->item;
```

- A traverse operation visits each node in the linked list
  - A pointer variable cur keeps track of the current node

```
for (Node *cur = head;
     cur != NULL; cur = cur->next)
    x = cur->item;
```

Traverse a Linked List



The effect of the assignment `cur = cur->next`

Delete a Node from a Linked List

- Deleting an interior/last node

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

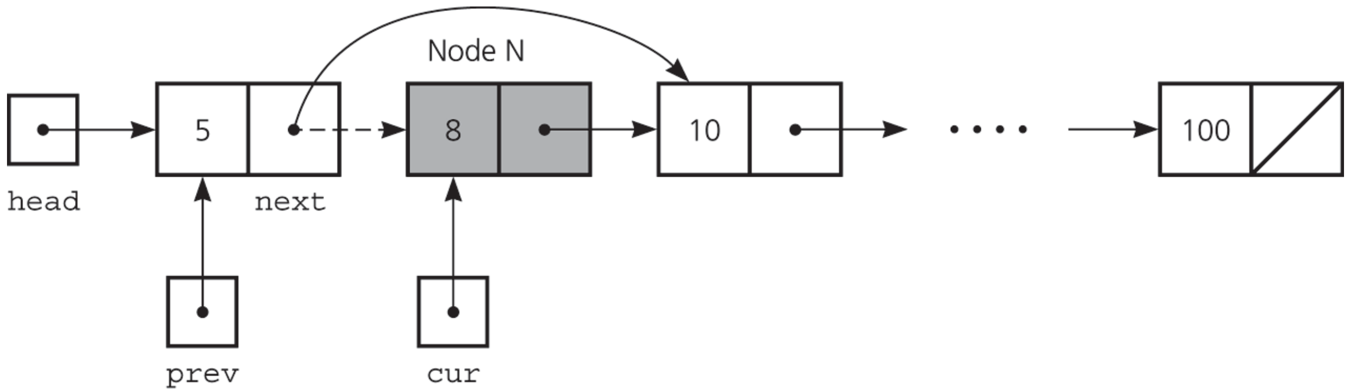
- Deleting the first node

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

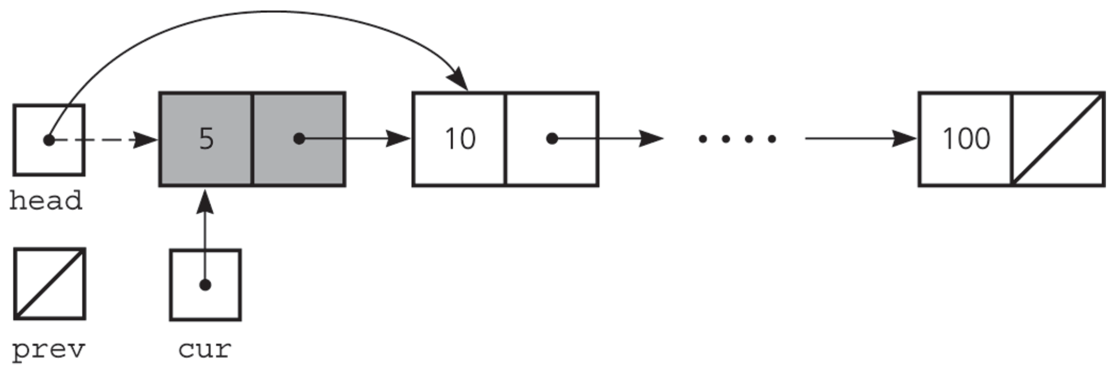
- Return deleted node to system

```
free(cur);
```

Delete a Node from a Linked List



Deleting a node from a linked list

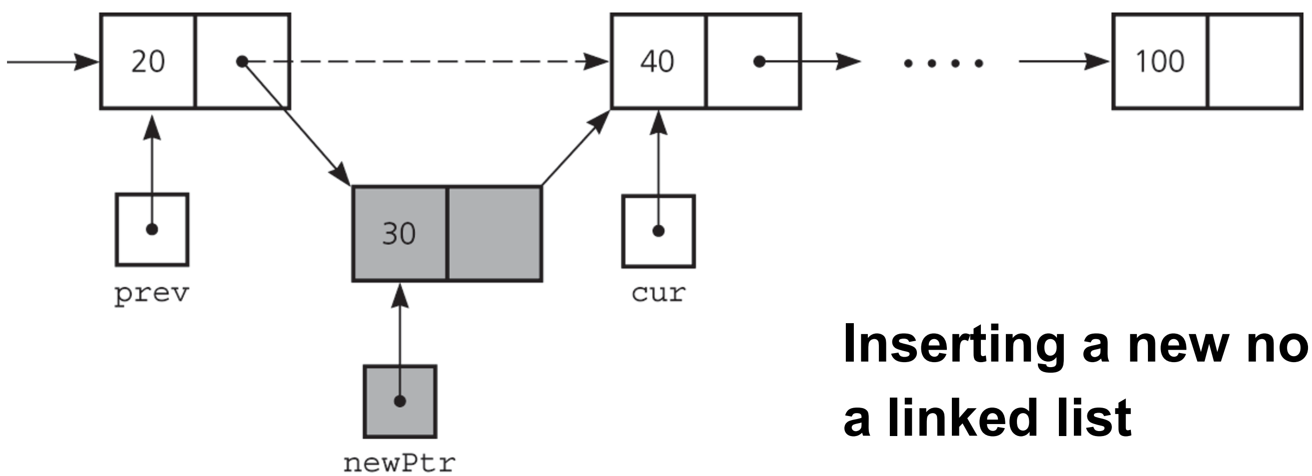


Deleting the first node

Insert a Node into a Linked List

- To insert a node between two nodes

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

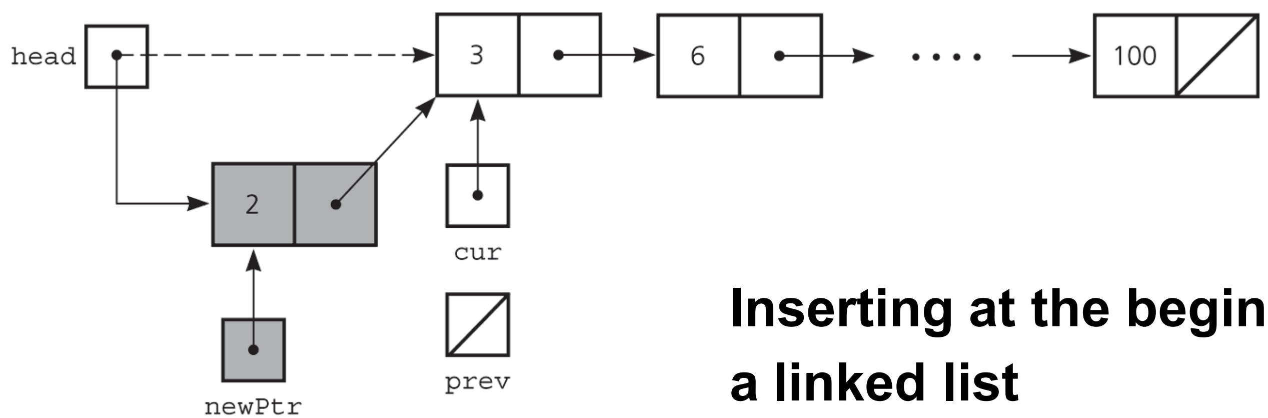


Inserting a new node into a linked list

Insert a Node into a Linked List

- To insert a node at the beginning of a linked list

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

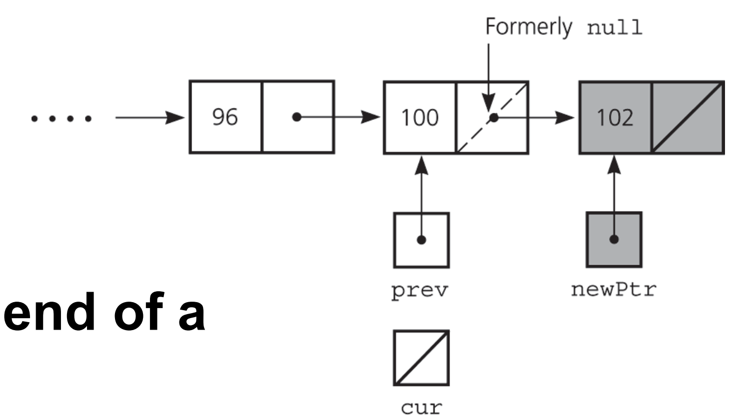


Inserting at the beginning of a linked list

Insert a Node into a Linked List

- Inserting at the end of a linked list

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



Inserting at the end of a linked list

Look up

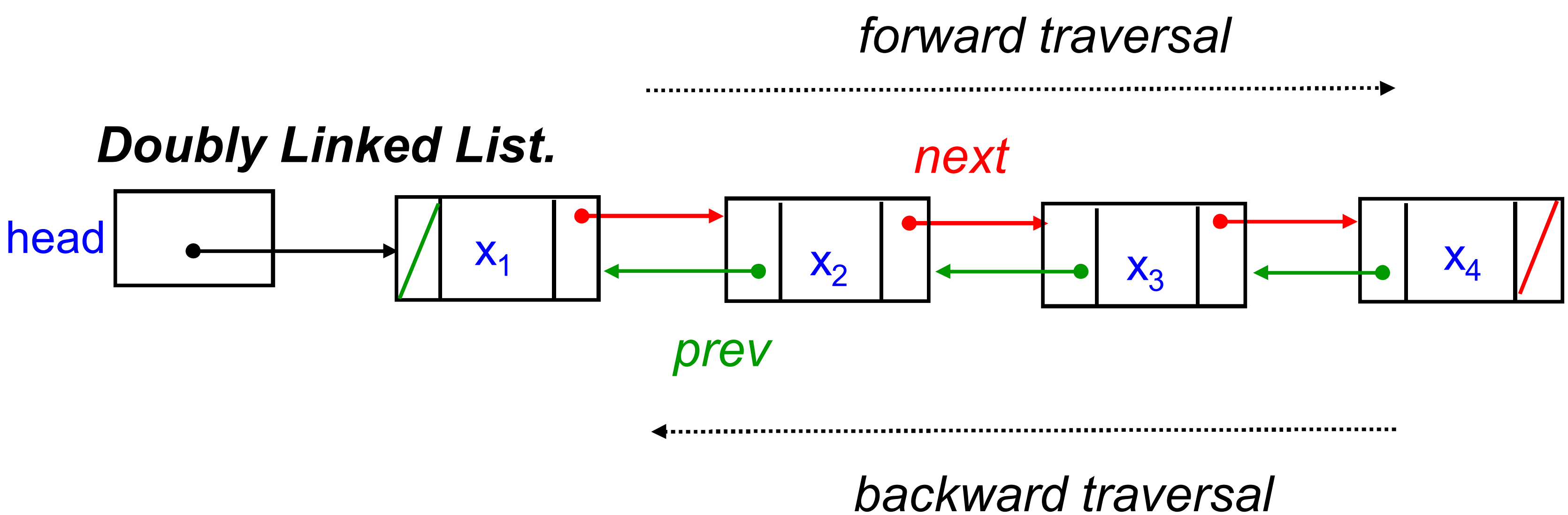
```
BOOLEAN lookup (int x, Node *L)
 {  if (L == NULL)
        return FALSE
    else if (x == L->item)
             return TRUE
    else
        return lookup(x, L-next);
 }
```

An ADT Interface for List

- Functions
  - `isEmpty`
  - `getLength`
  - `insert`
  - `delete`
  - `Lookup`
  - …
- Data Members
  - `head`
  - `Size`
- Local variables to member functions
  - `cur`
  - `prev`

Doubly Liked Lists

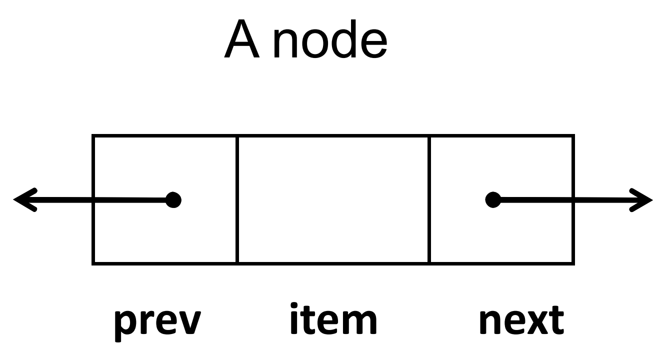
- Frequently, we need to traverse a sequence in BOTH directions efficiently
- ***Solution***: Use doubly-linked list where each node has two pointers



Doubly Linked Lists

- A node in a doubly linked list is usually a `struct`

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

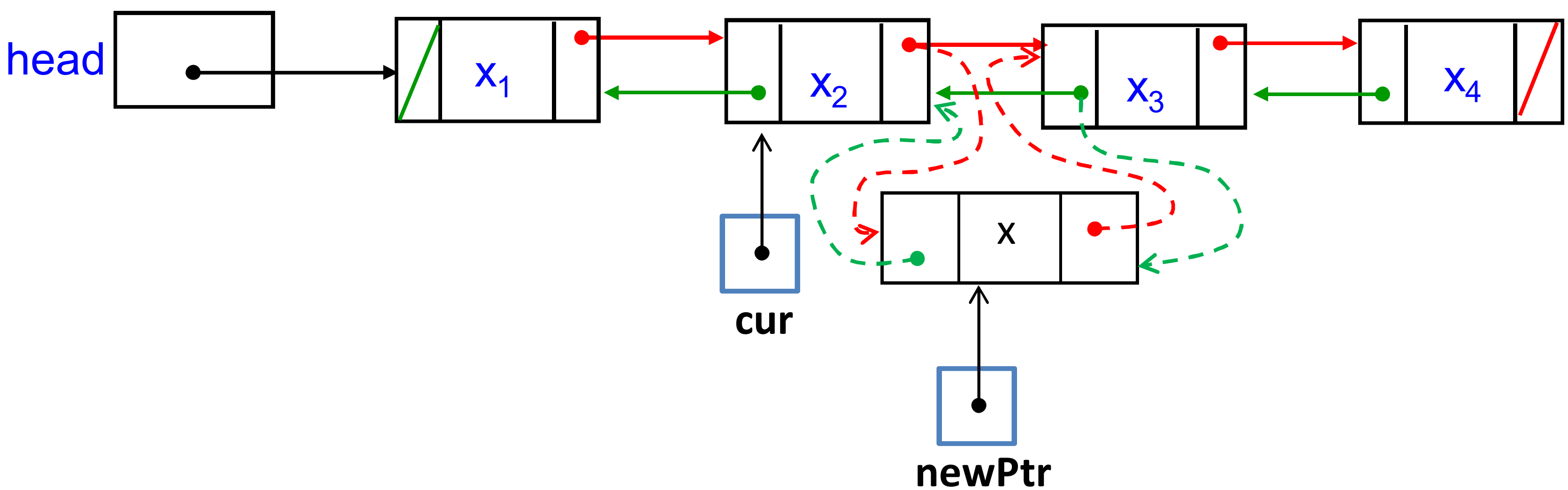


- A node is dynamically allocated

```
Node *p;
p = malloc(sizeof(Node));
```

Inserting a Node

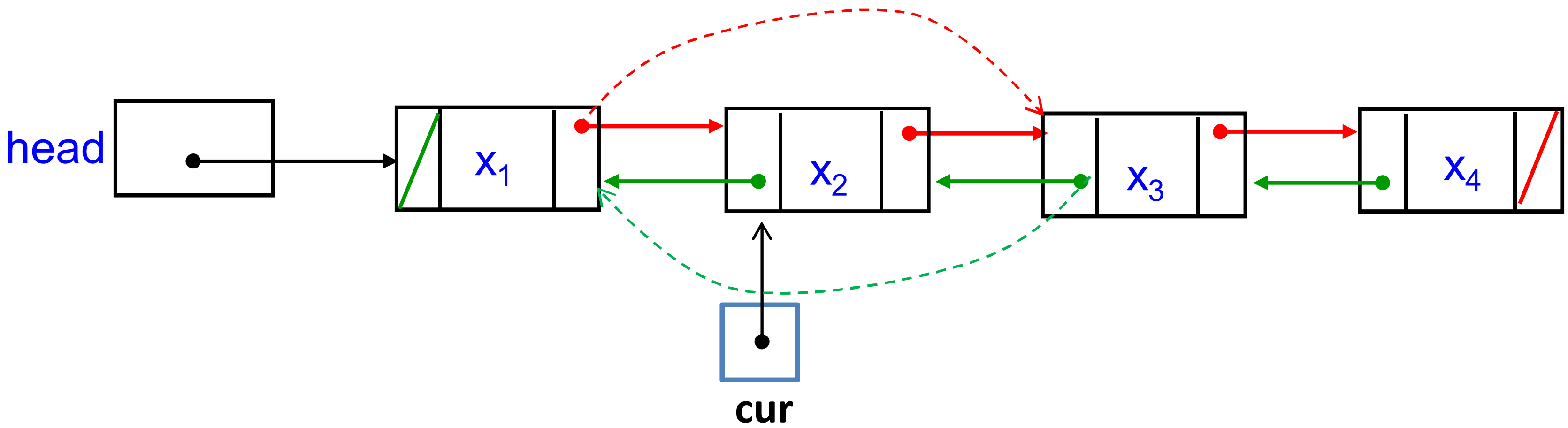
- Insert a node in the middle (After cur)




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

Deleting a Node

- Delete a node from middle



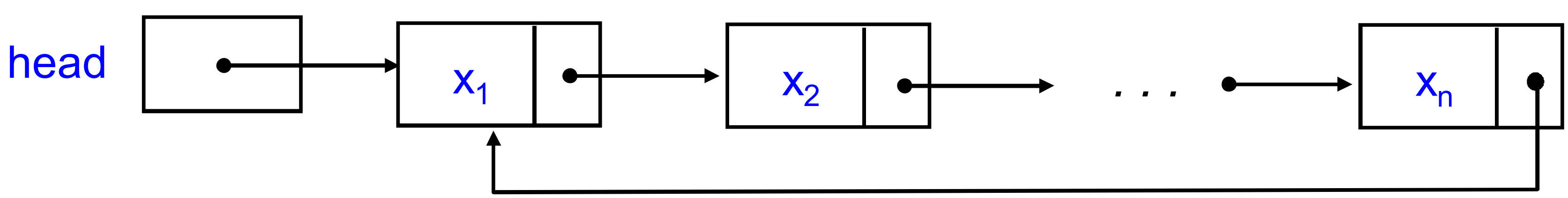

```
cur->prev->next = cur->next;
cur->next->prev = cur->prev;
free(cur);
```

Circular Linked Lists

- May need to cycle through a list repeatedly, e.g. round robin system for a shared resource.

- ***Solution***: Have the last node point to the first node (Head).

Circular Linked List.



Try in Lab

- Insert a node into a sorted linked lists at the correct position.
- Concatenate two singly linked lists.
- Concatenate two doubly linked lists.
- Concatenate two circular linked lists.

Array Vs. Linked LIst

Cost of Accessing i^{th} Element

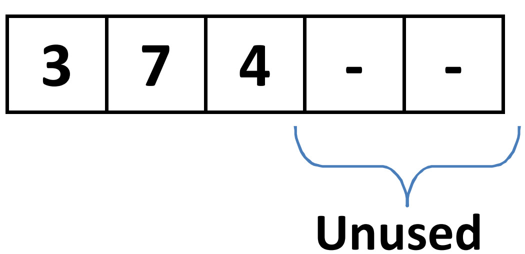
Array	**Linked List**
Accessed by index	Accessed by traversing
Constant	Best case: O(1)
O(1)	Average/Worst case: O(n)

Array Vs. Linked LIst

Memory Requirement

Array

Fixed size



Suffers from fragmentation.
- Contiguous Memory may not be available

Linked List

No unused memory

Extra space for pointers

- Negligible if data part is big

Suffers less from fragmentation.
- Small blocks are likely to be available

Array Vs. Linked LIst

Cost of inserting an element

Array	**Linked List**
No traversal but shifting	Traversal but no shifting
Beginning – O(n)	Beginning – O(1)
Middle – O(n)	Middle – O(n)

End – O(1) // If array is not full
O(n) // If array is full
// Reallocation required
	End – O(n)

Array Vs. Linked LIst

Cost of deleting an element

Array

No traversal but shifting

Beginning – O(n)
Middle – O(n)
End – O(1) // If array is not full

Linked List

Traversal but no shifting

Beginning – O(1)
Middle – O(n)
End – O(n)

Array Vs. Linked LIst

Ease of Use

Array	**Linked List**
Easy to implement	Complex use of pointers Care should be taken to free up the memory for deleted nodes.

Polynomial Representation

- What is it?

An example of a single variable polynomial:

$$\boxed{4x^6 + 10x^4 - 5x + 3}$$

$$4x^6 + 0.x^5 + 10x^4 + 0.x^3 + 0.x^2 + (-5)x^1 + 3x^0$$

Remark: the order of this polynomial is 6

(look for highest exponent)

Polynomial ADT

- Why call it an Abstract Data Type (ADT)?

A single variable polynomial can be generalized as:

$$\boxed{f(x) = \sum_{i=0}^{n} a_i x^i}$$

Polynomial ADT (continued)

…This sum can be expanded to:

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

Notice the two visible data sets namely: (C and E), where

- C is the coefficient set **[Real values: a_0 to a_n].**
- E is the exponent set object **[Integer values: 0 to n].**

Polynomial ADT (continued)

- Now what?

By definition of a data types:

A set of values and a set of allowable operations on those values.

We can now operate on this polynomial the way we like…

Polynomial ADT (continued)

- What kinds of operations?

Here are the most common operations on a polynomial:

- Add & Subtract
- Multiply
- Differentiate
- Integrate
- etc…

Polynomial ADT (continued)

- Why implement this?

Calculating polynomial operations by hand can be very cumbersome. Take differentiation as an example:

$$d(23x^9 + 18x^7 + 41x^6 + 163x^4 + 5x + 3)/dx$$

$$= (23*9)x^{(9-1)} + (18*7)x^{(7-1)} + (41*6)x^{(6-1)} + \ldots$$

Polynomial ADT (continued)

- How to implement this?

There are different ways of implementing the polynomial ADT:

- Array (not recommended)

- Linked List (preferred and recommended)

Polynomial ADT (continued)

- Array Implementation:
- $p1(x) = 8x^3 + 3x^2 + 2x + 6$
- $p2(x) = 23x^4 + 18x - 3$

p1(x)

6	2	3	8
0		2	

p2(x)

-3	18	0	0	23
0		2		4

Index represents exponents

Polynomial ADT (continued)

- This is why arrays aren’t good to represent polynomials:

- $p3(x) = 16x^{21} - 3x^{5} + 2x + 6$

6	2	0	0	-3	0	…………	0	16
0	1	2	3	4	5		20	21

WASTE OF SPACE!

Polynomial ADT (continued)

- Advantages of using an Array:

  - only good for **non-sparse** polynomials.
  - ease of storage and retrieval.

- Disadvantages of using an Array:

  - have to allocate array size ahead of time.
  - huge array size required for sparse polynomials. Waste of space and runtime.

Polynomial ADT (continued)

- Sparse Polynomial

- p4(x) = $6x^{102} - 13x^{14} + 22x^{2} - 2$

-2	0	22	…	…	-13	…	…	…	0	6
0	1	2	3 ---------13		14	15 ---------------100			101	102

Huge wastage of space

Polynomial ADT using Linked List

Node Representation

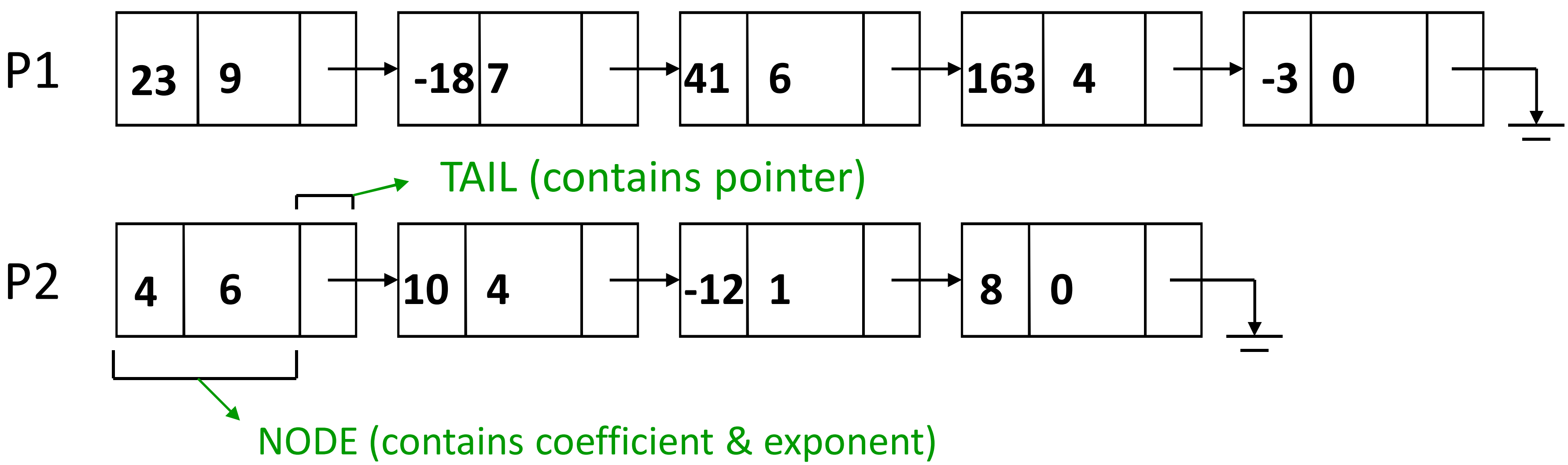
```
struct polynode
{
   double coeff;
   int expon;
   polynode* next;
};
typedef struct polynode *polyptr;
```

coeff	expon	next

Polynomial ADT (continued)

- Linked list Implementation:

- $p1(x) = 23x^9 - 18x^7 + 41x^6 + 163x^4 - 3$
- $p2(x) = 4x^6 + 10x^4 - 12x + 8$



Polynomial ADT (continued)

- Advantages of using a Linked list:

  - save space (don't have to worry about sparse polynomials) and easy to maintain

  - don't need to allocate list size initially and can declare nodes (terms) only as needed

Adding Two Polynomials (Linked Lists)

- Given two polynomials:

$p1(x) = 23x^9 - 18x^7 + 163x^4 - 3$

$p2(x) = 4x^8 + 10x^7 + 8$

After addition

$p3(x) = 23x^9 + 4x^8 - 8x^7 + 163x^4 + 5$

Adding Two Polynomials (Linked Lists)

- How it works

$p1(x) = 23x^{9} - 18x^{7} + 163x^{4} - 3$

$p2(x) = 4x^{8} + 10x^{7} + 8$

After addition

$p3(x) = 23x^{9}$

Adding Two Polynomials (Linked Lists)

- How it works

$p1(x) = 23x^{9} - 18x^{7} + 163x^{4} - 3$

$p2(x) = 4x^{8} + 10x^{7} + 8$

After addition

$p3(x) = 23x^{9} + 4x^{8}$

Adding Two Polynomials (Linked Lists)

- How it works

$p1(x) = 23x^{9} - 18x^{7} + 163x^{4} - 3$

$p2(x) = 4x^{8} + 10x^{7} + 8$

After addition

$p3(x) = 23x^{9} + 4x^{8} + (-18+10)x^{7}$

Adding Two Polynomials (Linked Lists)

- How it works

$p1(x) = 23x^{9} - 18x^{7} + 163x^{4} - 3$

$p2(x) = 4x^{8} + 10x^{7} + 8$

After addition

$p3(x) = 23x^{9} + 4x^{8} - 8x^{7}$

Adding Two Polynomials (Linked Lists)

- How it works

$p1(x) = 23x^{9} - 18x^{7} + 163x^{4} - 3$

$p2(x) = 4x^{8} + 10x^{7} + 8$

After addition

$p3(x) = 23x^{9} + 4x^{8} - 8x^{7} + 163x^{4}$

Adding Two Polynomials (Linked Lists)

- How it works

$p1(x) = 23x^9 - 18x^7 + 163x^4 - 3$

$p2(x) = 4x^8 + 10x^7 + 8$

After addition

$p3(x) = 23x^9 + 4x^8 - 8x^7 + 163x^4 + (-3+8)$

Adding Two Polynomials (Linked Lists)

- How it works

$p1(x) = 23x^9 - 18x^7 + 163x^4 - 3$

$p2(x) = 4x^8 + 10x^7 + 8$

After addition

$p3(x) = 23x^9 + 4x^8 - 8x^7 + 163x^4 + 5$

Adding Two Polynomials (Linked Lists)

- How it works

$p1(x) = 23x^9 - 18x^7 + 163x^4 - 3$

$p2(x) = 4x^8 + 10x^7 + 8$

After addition

$p3(x) = 23x^9 + 4x^8 - 8x^7 + 163x^4 + 5$

Polynomial ADT (continued)

- Adding polynomials using a Linked list representation: (storing the result in p3)

To do this, we have to break the process down to cases:

- **Case 1: exponent of p1 > exponent of p2**
  - Copy node of p1 to end of p3.
  - p1 = p1->next;
- **Case 2: exponent of p1 < exponent of p2**
  - Copy node of p2 to end of p3.
  - p2 = p2->next;
- **Case 3: exponent of p1 = exponent of p2**
  - Create a new node in p3 with the same exponent and with the sum of the coefficients of p1 and p2.
  - p1 = p1->next; p2=p2->next;

Polynomial ADT (continued)

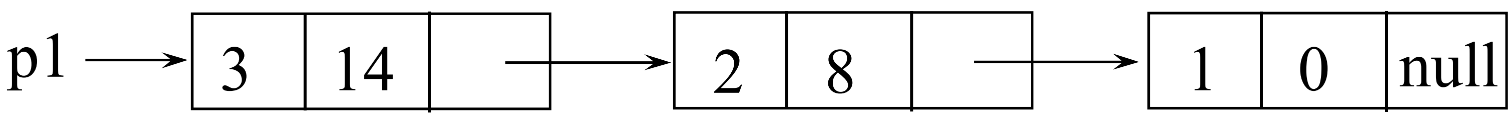
- Adding polynomials using a Linked list representation: (storing the result in p3)

To do this, we have to break the process down to cases:

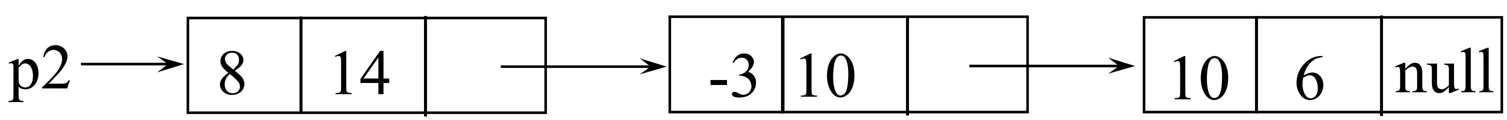
- **Case 4: p1 == NULL**
  - Copy entire p2 to end of p3.
- **Case 5: p2 == NULL**
  - Copy entire p1 to end of p3.

Example

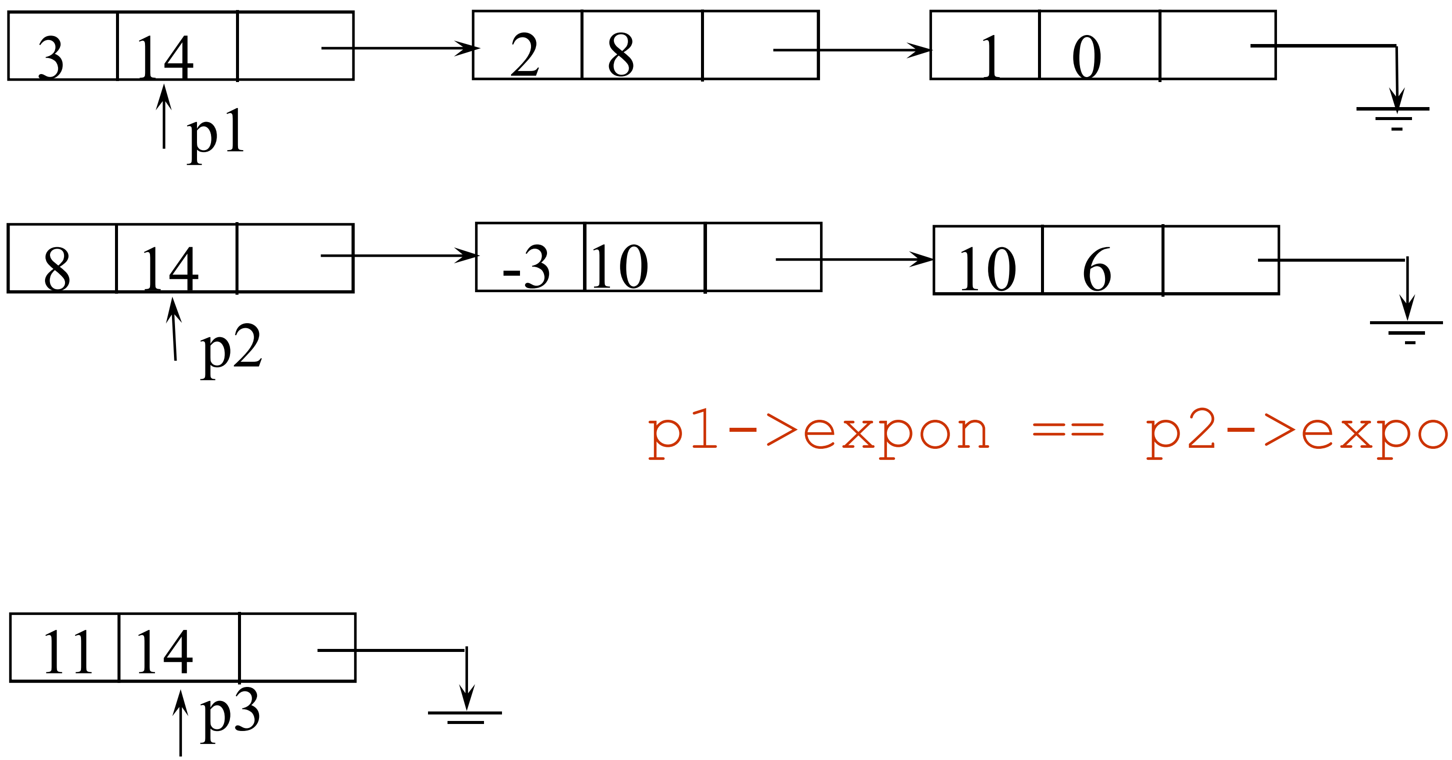
$$p1 = 3x^{14} + 2x^{8} + 1$$



$$p2 = 8x^{14} - 3x^{10} + 10x^{6}$$

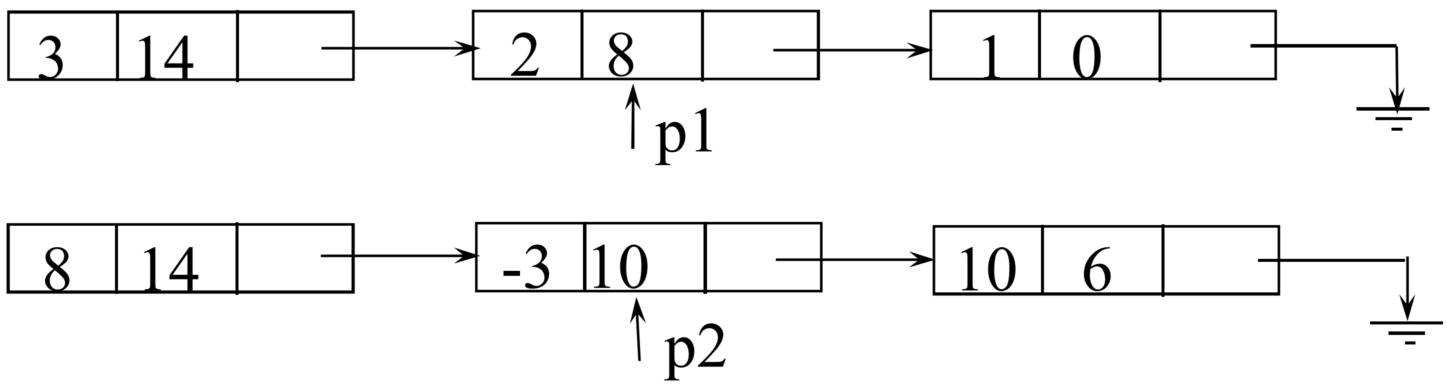


Adding Two Polynomials

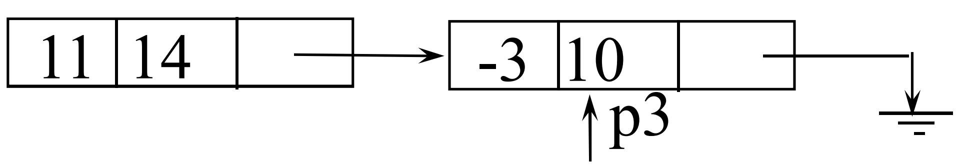


`p1->expon == p2->expon`

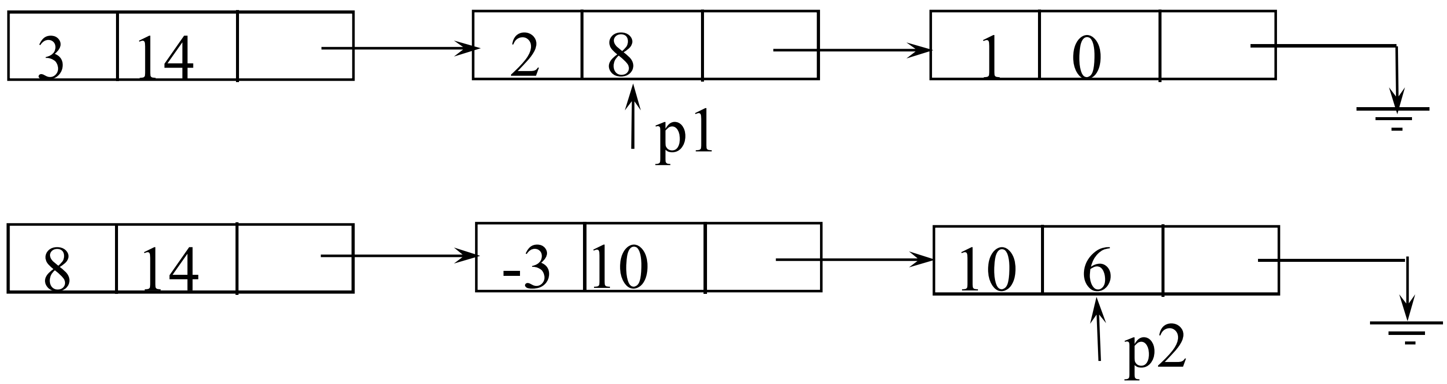
Adding Two Polynomials



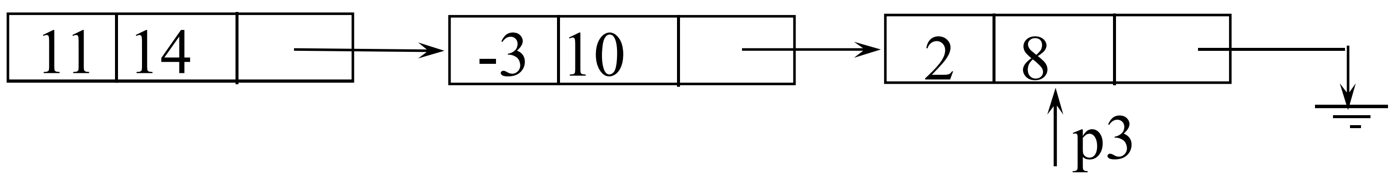
p1->expon < p2->expon



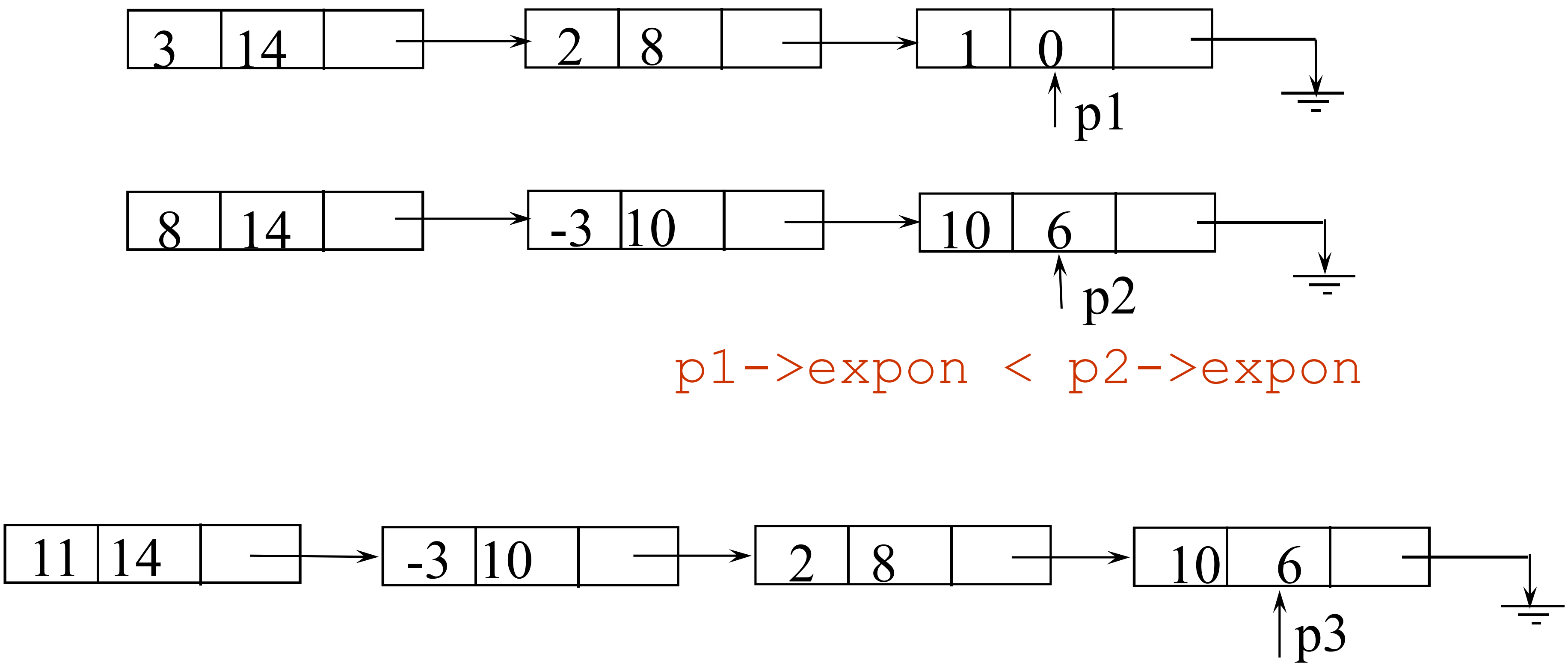
Adding Two Polynomials



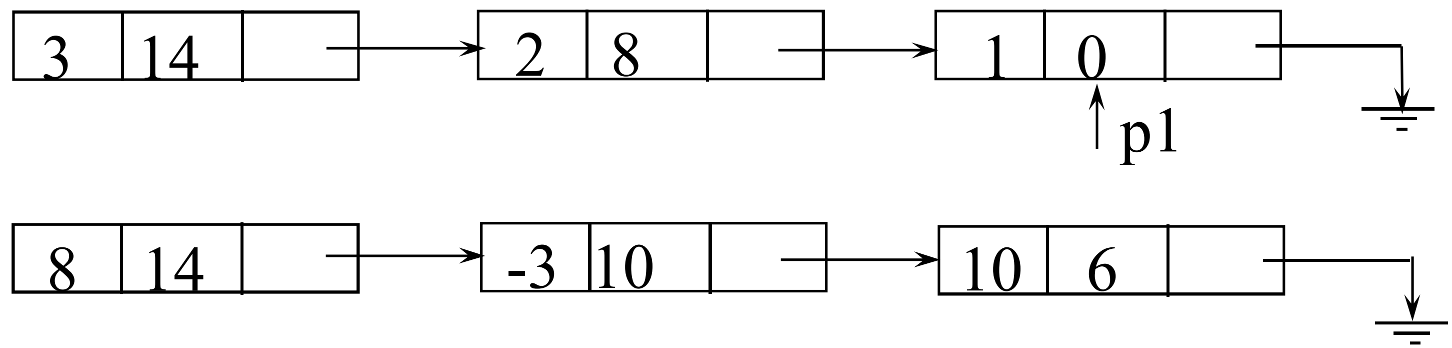
`p1->expon > p2->expon`



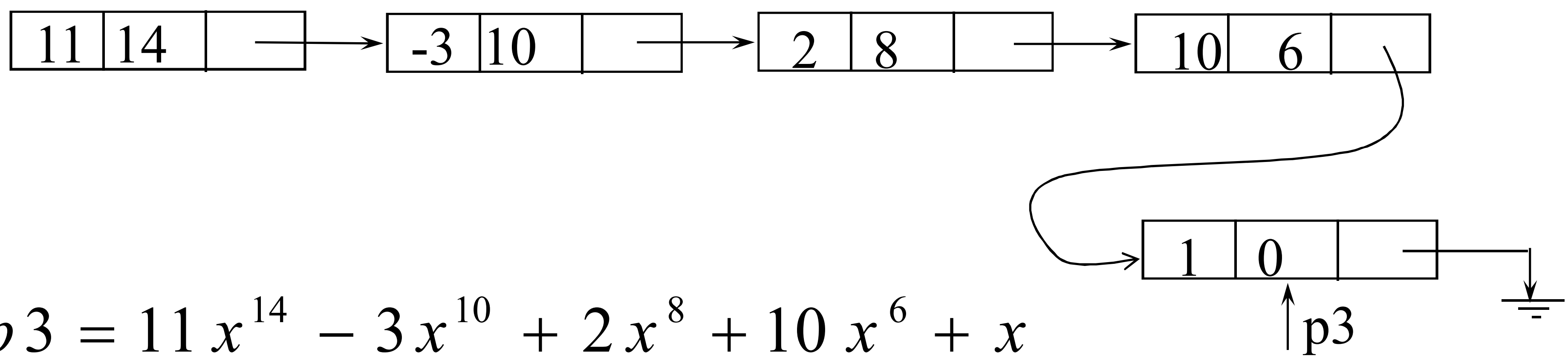
Adding Two Polynomials



Adding Two Polynomials



`p2 = NULL`



$$p3 = 11x^{14} - 3x^{10} + 2x^{8} + 10x^{6} + x$$

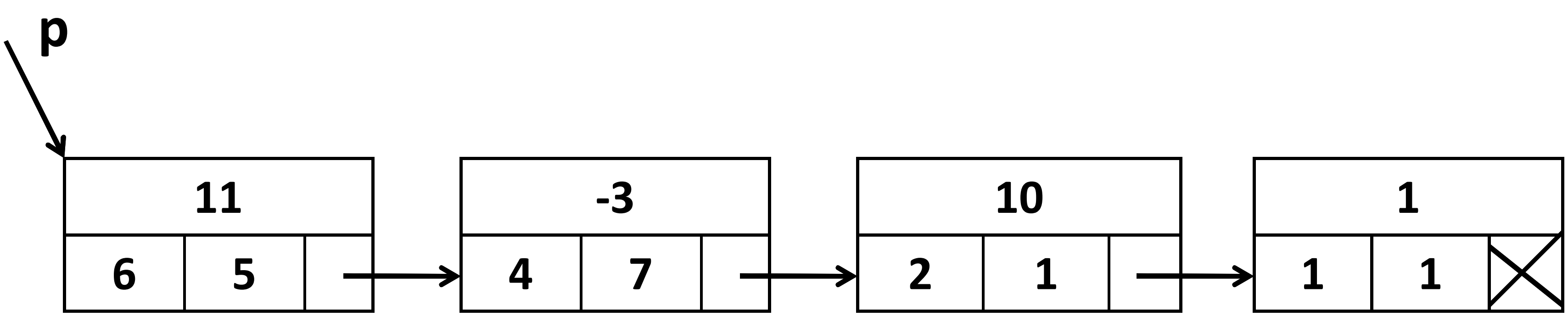
- **Home task**

  Represent a two-variable polynomial using linked list.

  $4x^{10}y^7 – 10x^7y^6 + 13x^6y^6 – 23xy^2 + 62$

Representing two-variable polynomial

$$p = 11x^6y^5 - 3x^4y^7 + 10x^2y + xy$$



Representing two-variable polynomial
Another way

$$p = 11x^6y^5 - 3x^4x^7 + 10x^2y + xy$$

x ↓ / y →	1	2	3	4	5	6	7
1	**1**	0	0	0	0	0	0
2	**10**	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	**-3**
5	0	0	0	0	0	0	0
6	0	0	0	0	**11**	0	0

Sparse Matrix

Sparse Matrix Representation

In a sparse matrix, most entries are zeroes:

$$\begin{bmatrix} -1 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 4 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & \ldots & 0 & 6 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 \end{bmatrix}$$

Sparse Matrix Representation

Many problems require large sparse matrices.

For example, a college may have 1000 students and 500 different courses.

We can have a matrix M of size 1000 x 500, where M(*i,j*) represents the marks obtained by student *i* in course *j*. If M(*i*,*j*)=0, it means student *i* has not taken course *j.*

Each student takes only few courses. This will be sparse matrix with most of the elements 0.

Store as an ordinary array?   1000 x 500 = 500000 entries
Impractical (usually).

Sparse Matrix Representation using Linked List

Idea: Only allocate memory for the entries that are non-zero.

Mechanism:

Store each non-zero entry in a node.

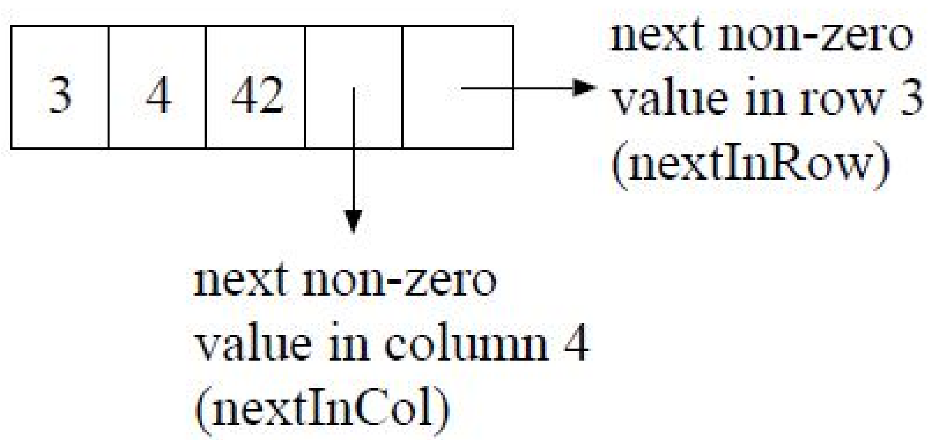
Use the heap to get the space for non-zero entries.

Link non-zero entries by row and by column.

Each node will need to know its row number and column number and (non-zero) value.

An Example Node

For example, row 3 and column 4 (i.e., entry [3,4]) contains 42:




```
struct Node{
   int row, col, value;
   Node* NextInCol;
   Node* NextInRow;
}
```

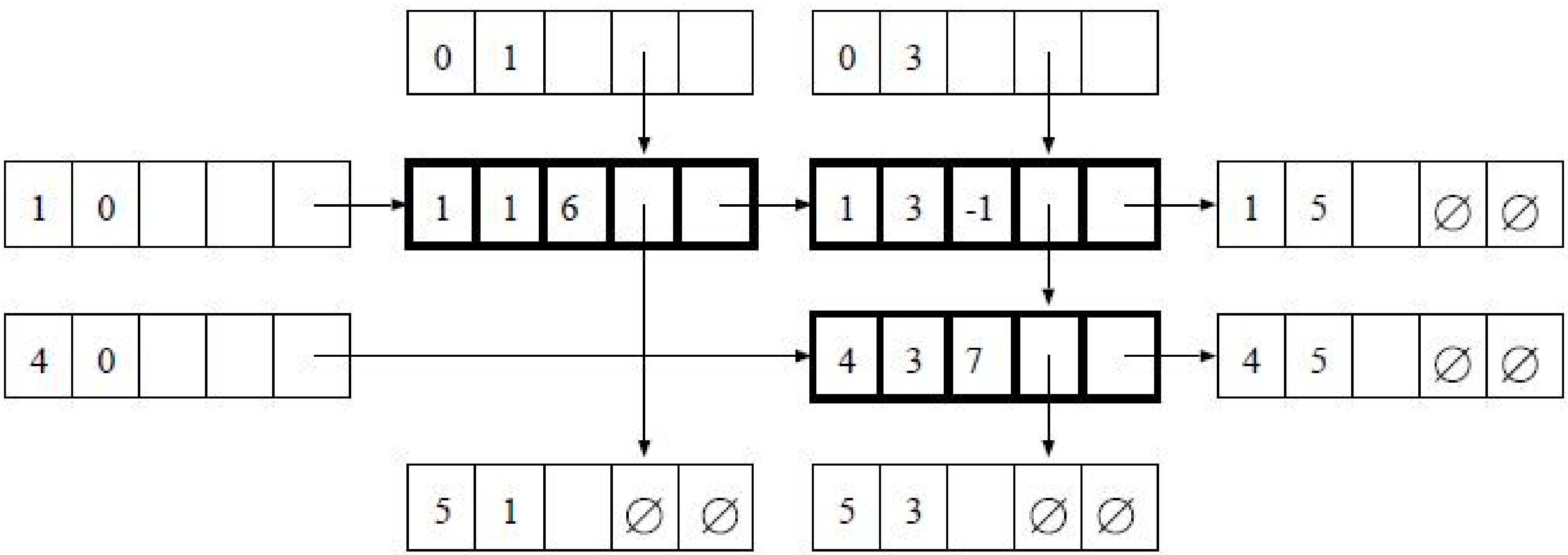
Row and Columns

- A row or a column is a linked list of non-zero entries.
- We will use plain linked lists (not circular or doubly-linked) with dummy nodes.
- The dummy header nodes would contain row or column number 0.
- The dummy trailer nodes would contain row number **numRows+1** or column number **numCols+1**.

Example

Example:

$$\begin{bmatrix} 6 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 \end{bmatrix}$$



Searching and Element

- We will need to search for particular rows or columns.

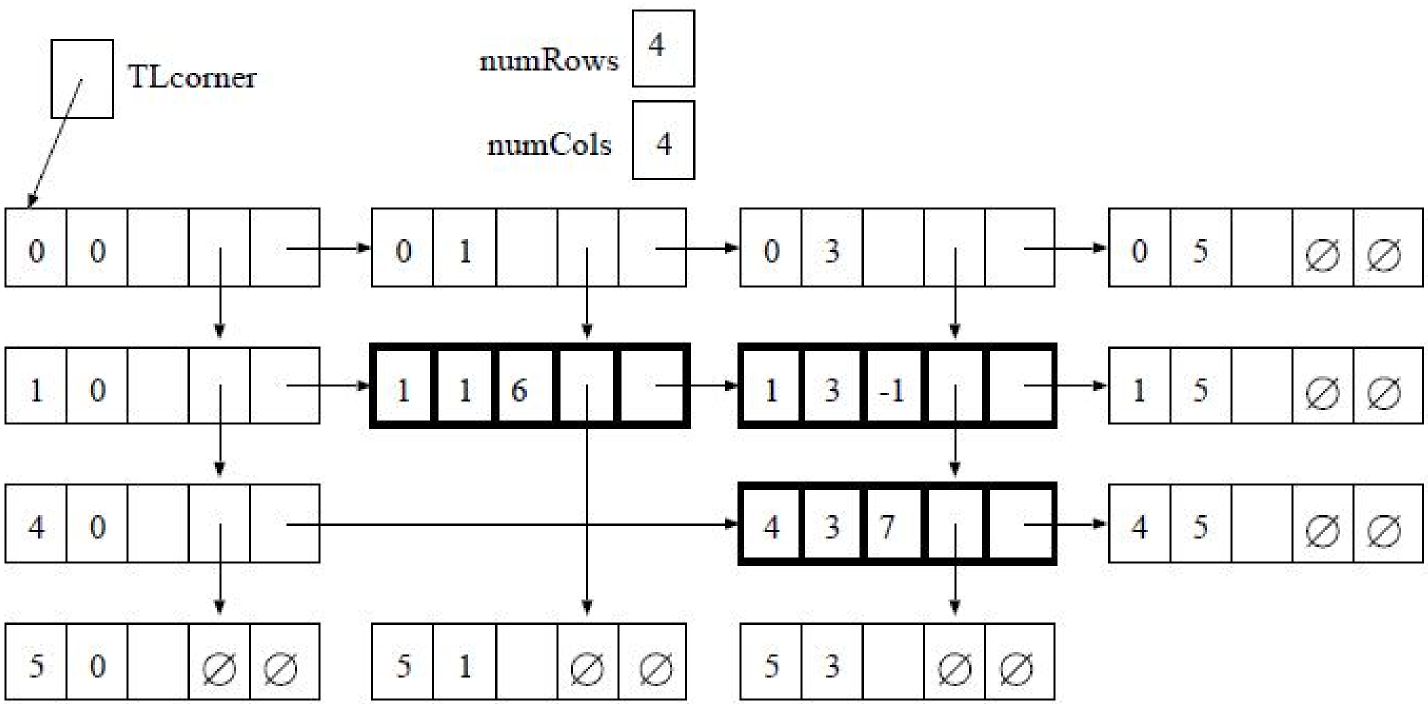
- **Idea:** Link together “row 0”, which is a list of all the available columns, and link together “column 0”, which is a list of all the available rows.

- How about dummy nodes for those lists? Good idea.

Example

Example (re-examined):

$$\begin{bmatrix} 6 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 \end{bmatrix}$$



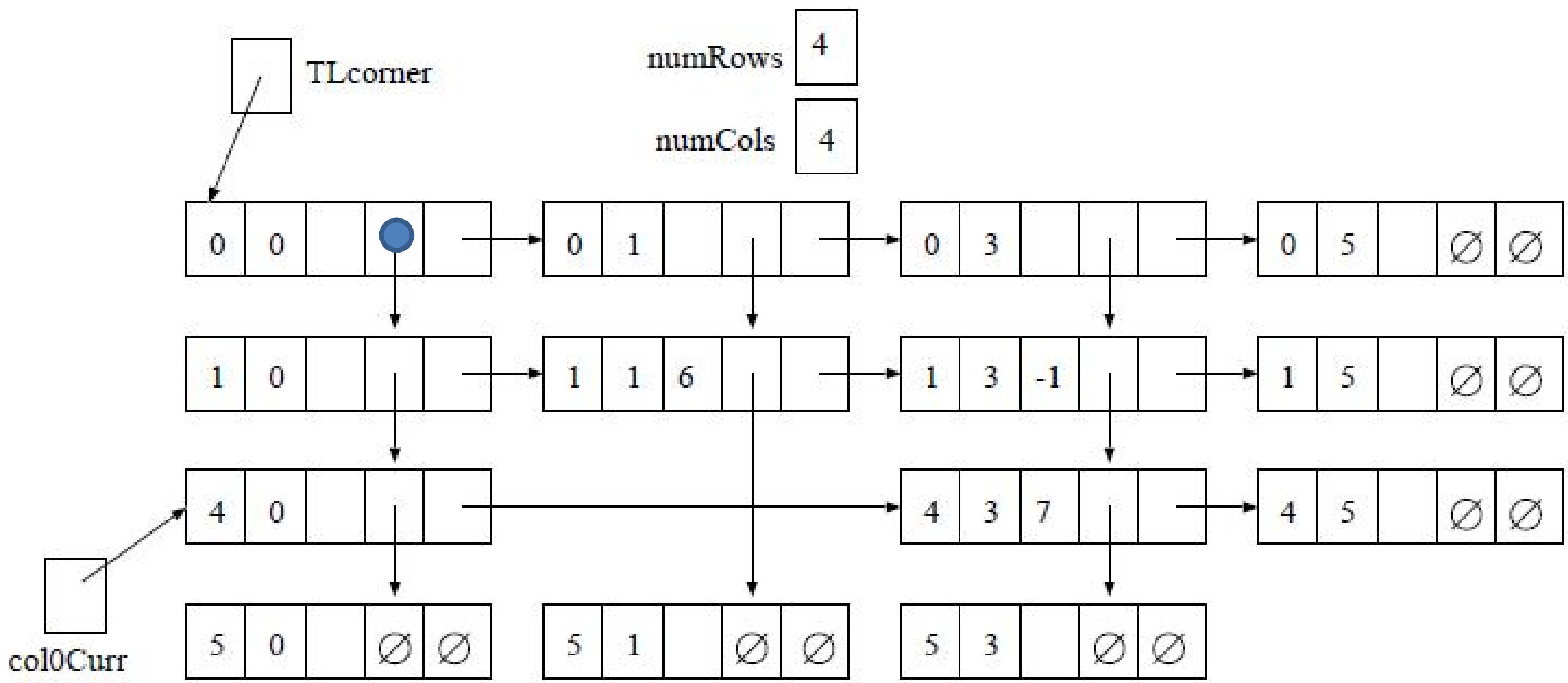
General Search Strategy

Search for [R,C] element

- Search down dummy column 0 looking for the row list for row R (or search across dummy row 0 looking for the column list for column C).
  - If the row list (or column list) doesn't exist, then all entries in that row (or column) are 0.
- Then search along row R looking for a node in column C (or search down column C looking for a node in row R).
- If the node you're looking for doesn't exist, then the value of that entry is 0.
- We always search an ordered list.

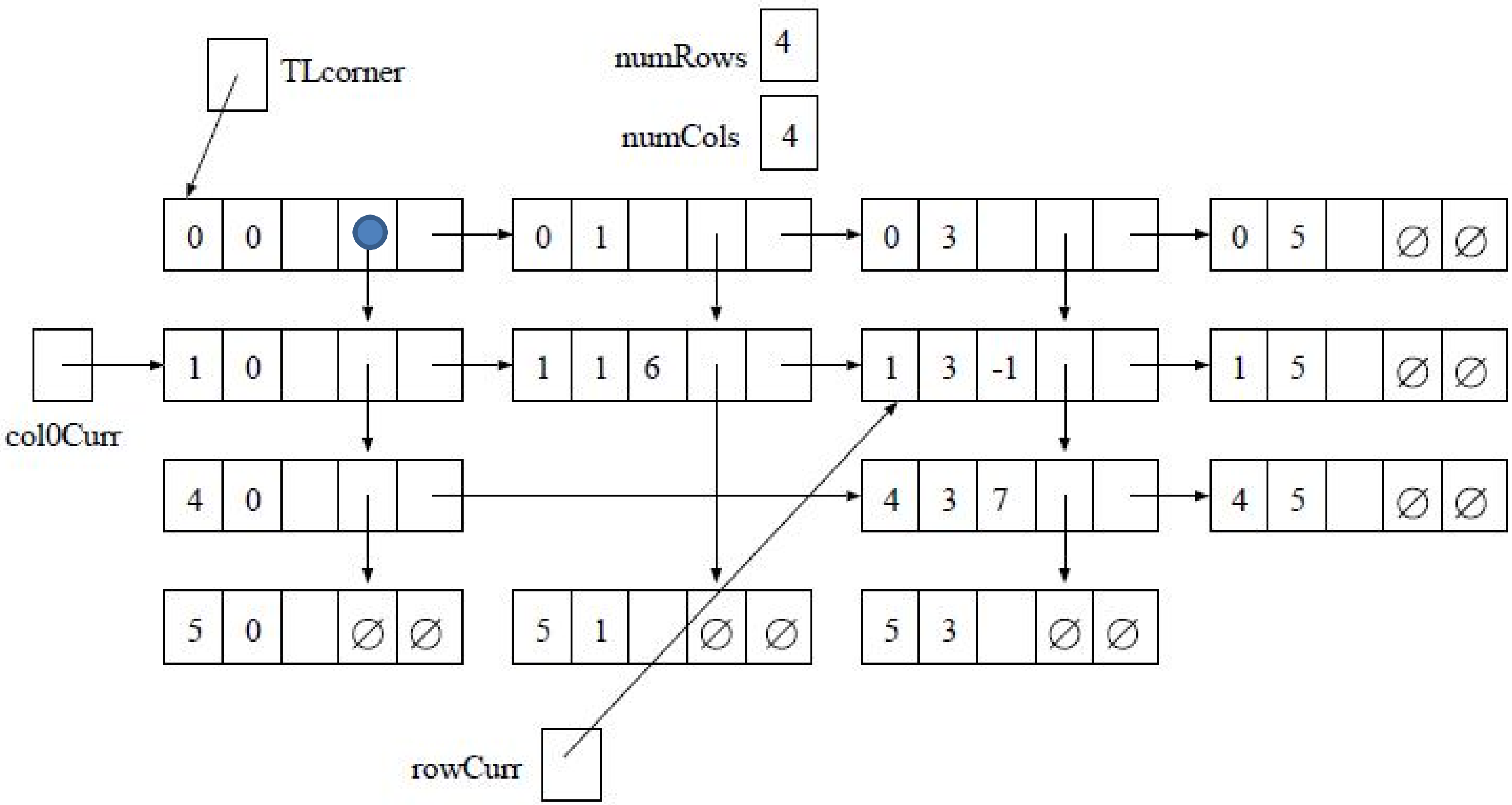
Example

We can see that the value of entry [3,2] is zero because there is no list for row 3:



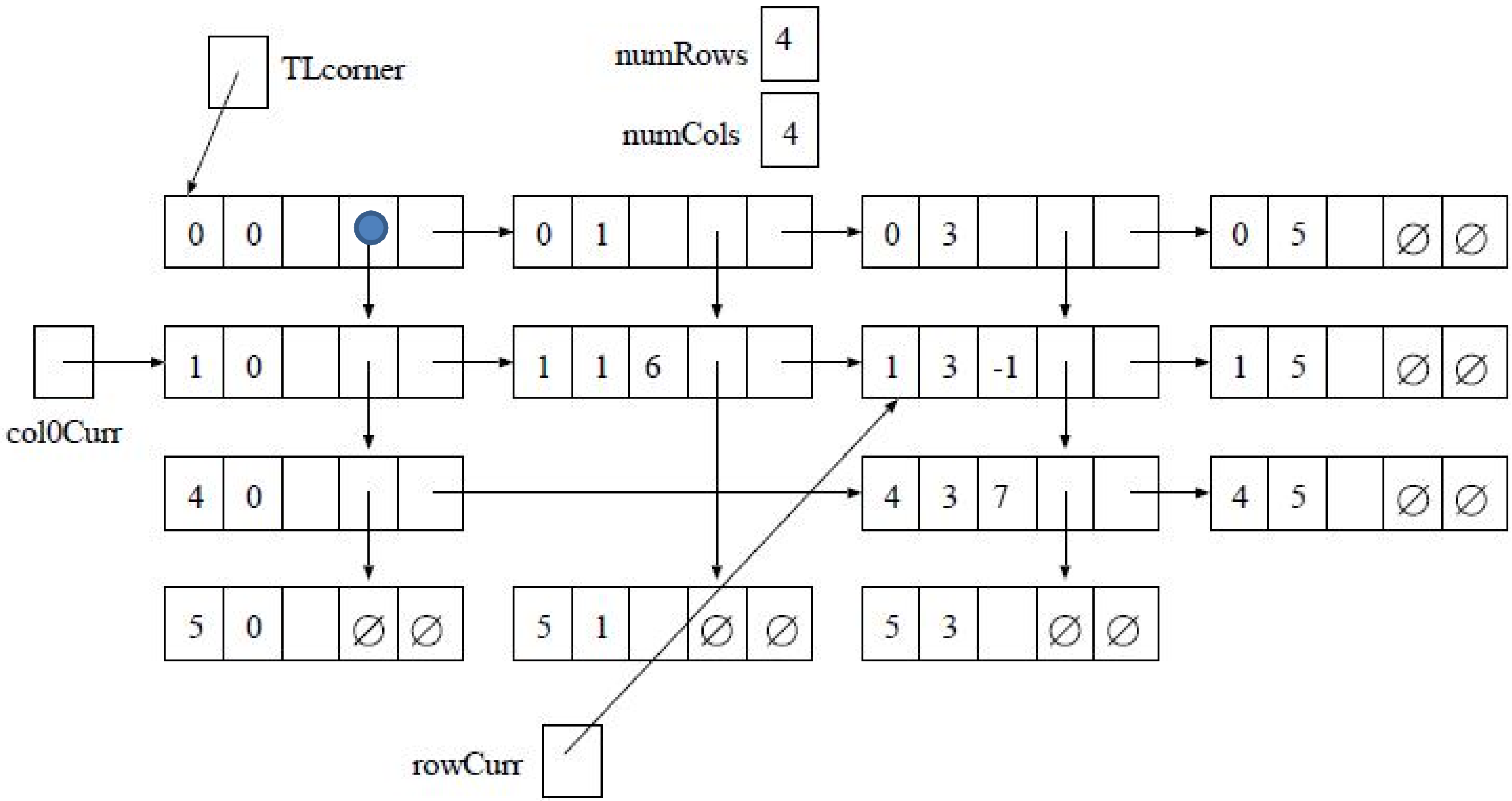
Example

We can see that the value of entry [1,2] is zero because, although there is a list for row 1, the list does not contain a node in column 2:



Example

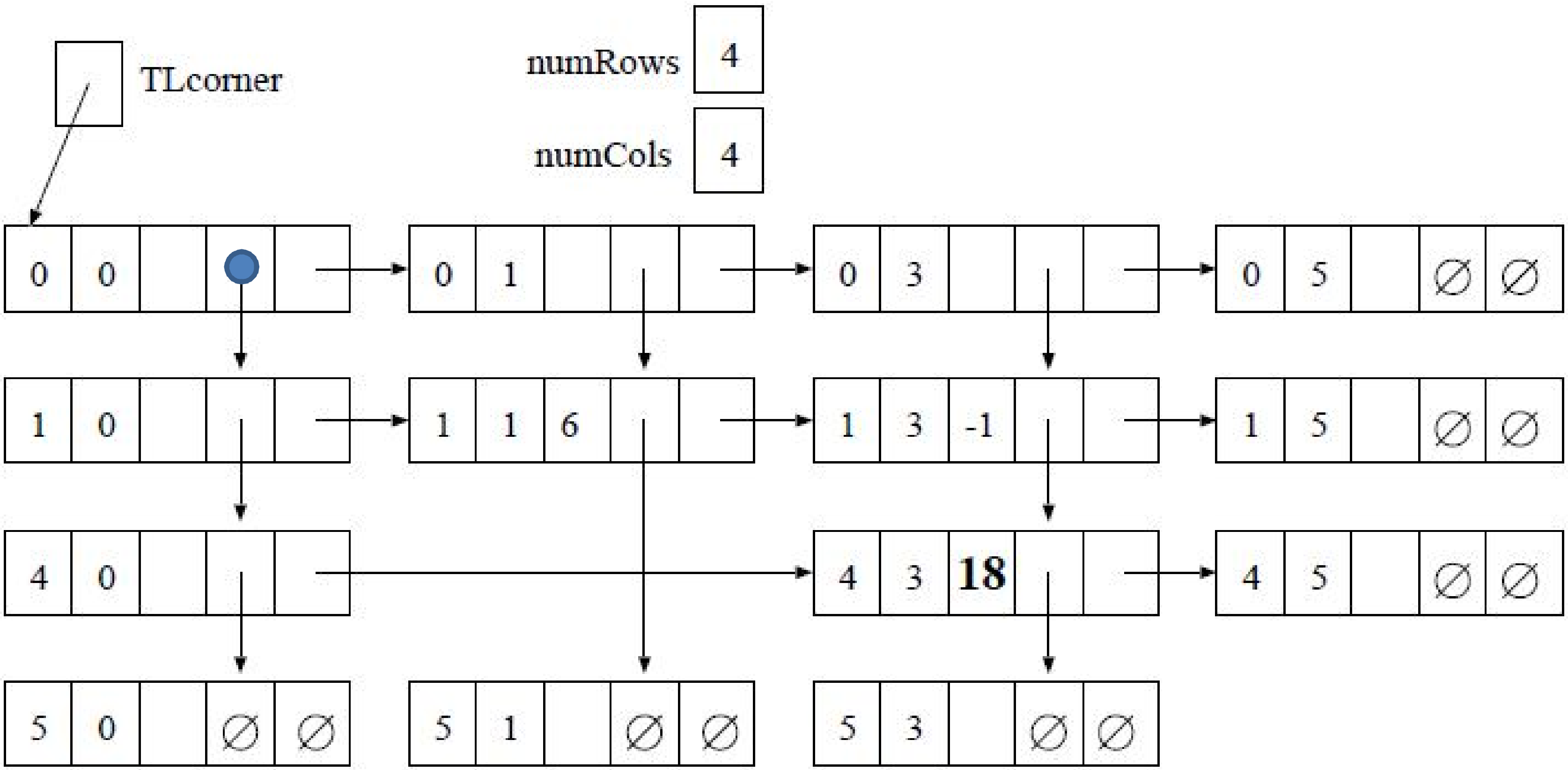
We can see that the value of entry [1,3] is -1 because there is a list for row 1, it contains a node in column 3 and that node contains the value -1.



Setting a new value

If `newValue` is not 0 and a node already exists for entry `[R,C]`, simply change the value in the existing node.

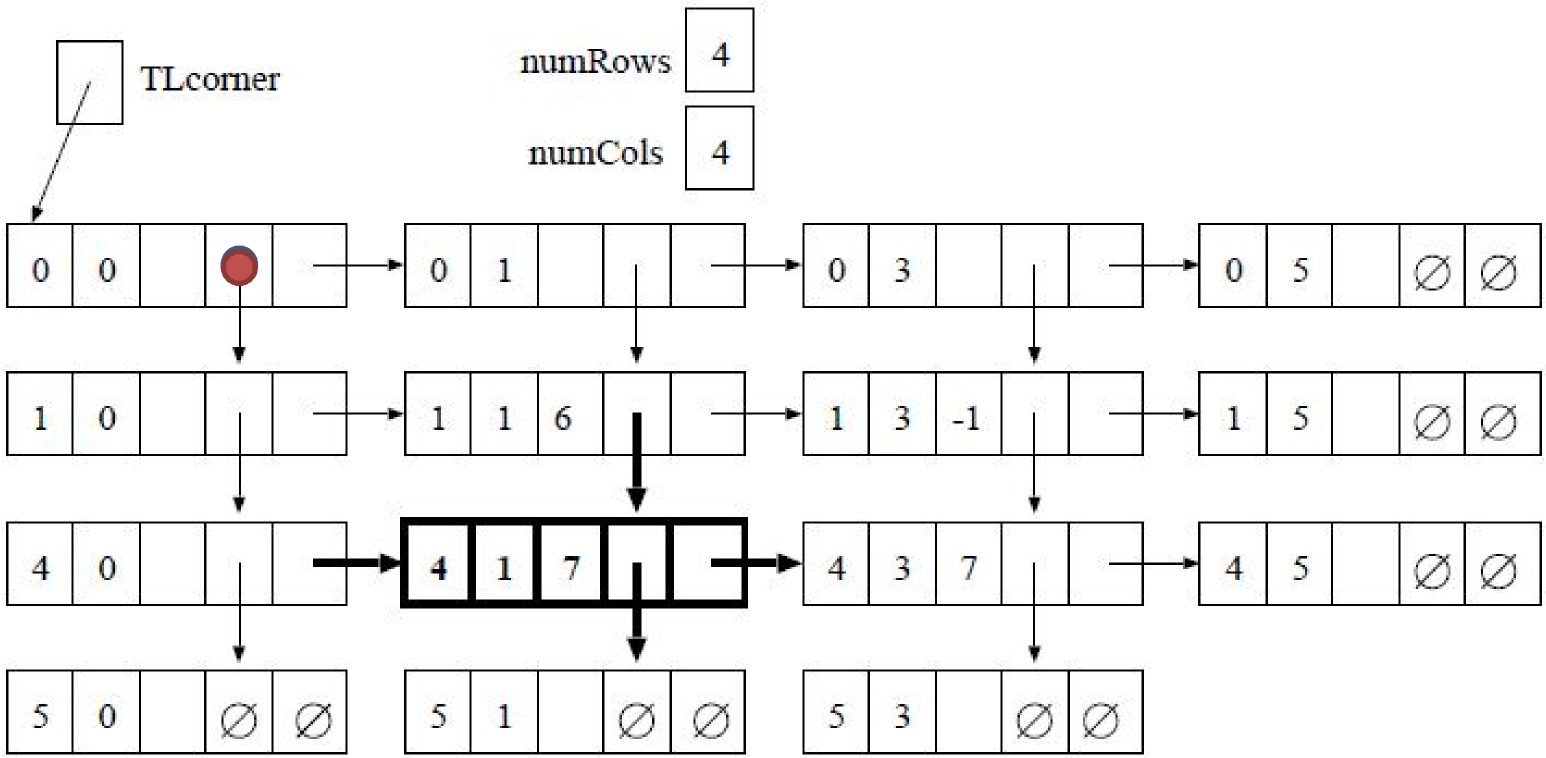
setValue(4,3,18) causes the following change:



More Example

If `newValue` is not 0, but no node exists for entry `[R,C]`, add a new node for entry `[R,C]`. If lists for both row `R` and column `C` already exist, simply link the new node into those lists in the appropriate positions:

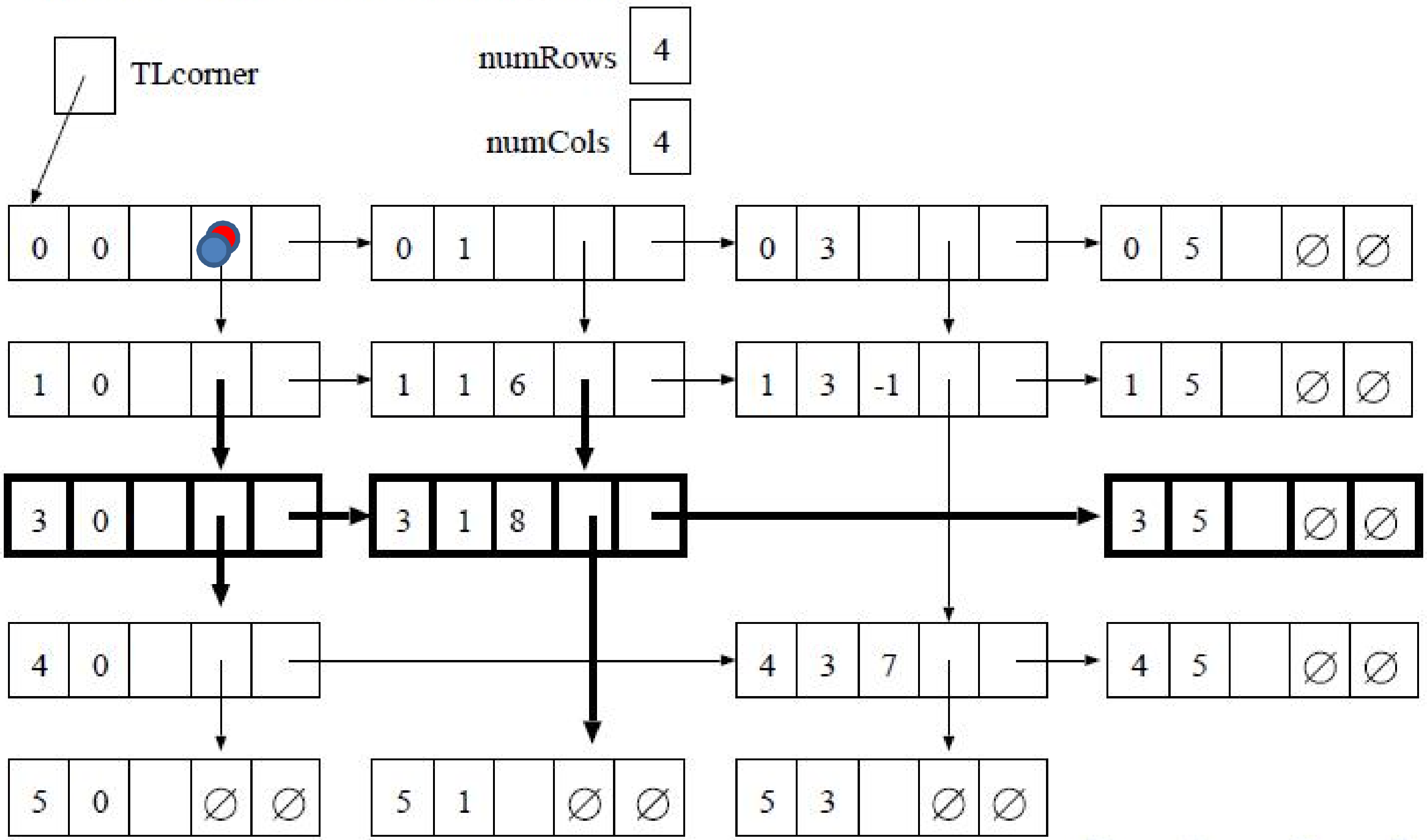
setValue(4,1,7) causes the following changes:



More Example

If `newValue` is not 0, no node exists for entry `[R,C]`, but no list exists for row R, then add a new node for entry `[R,C]` and create a new row list for row R.

setValue(3,1,8) causes the following changes:



Example

If `newValue` is not 0, you might have to add a new list for column `C`.

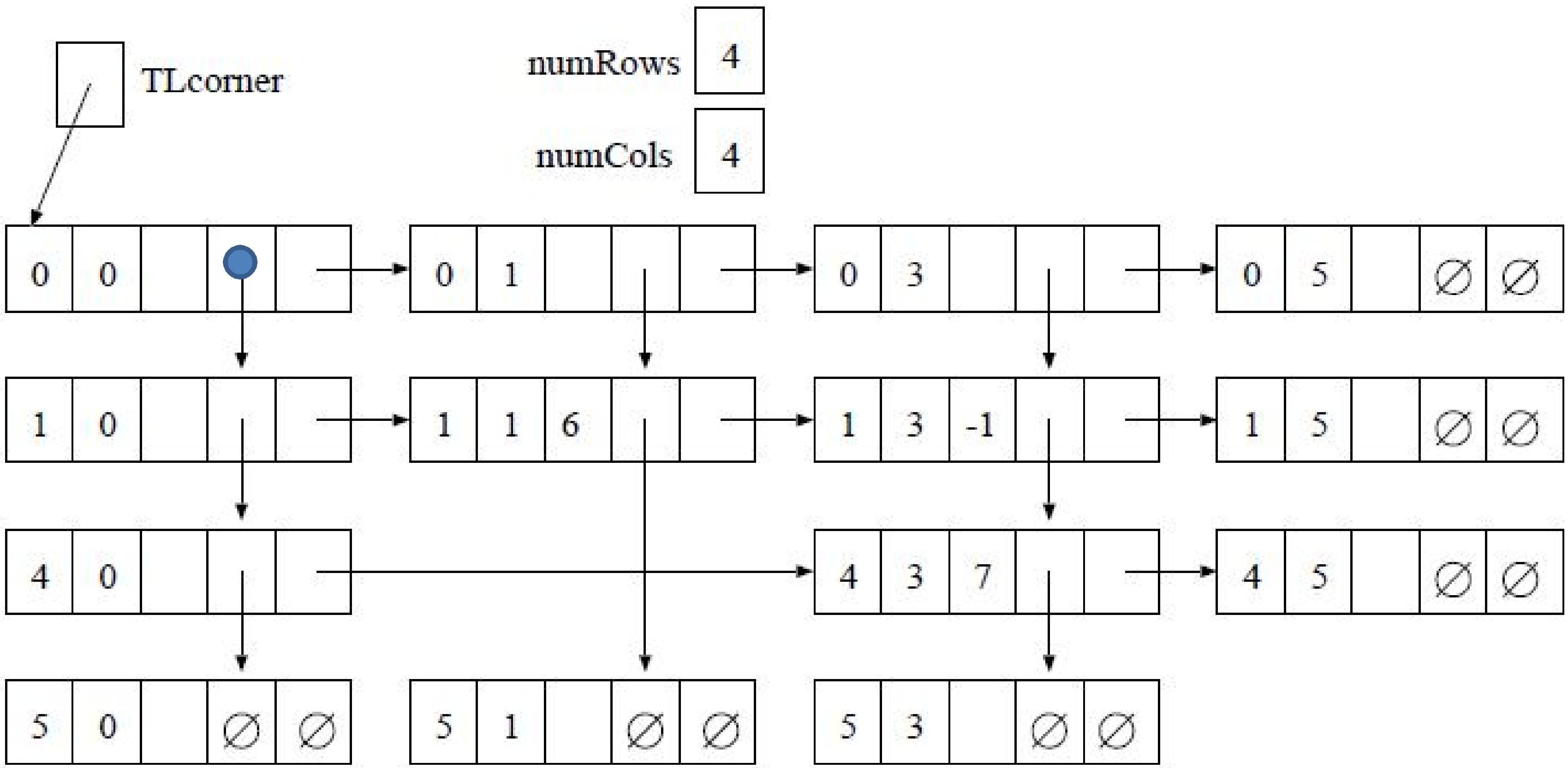
You might have to add new lists for both row `R` and column `C`.

(Pictures not shown.)

Example

If `newValue` is 0 and there is no node for entry `[R,C]`, then do nothing!

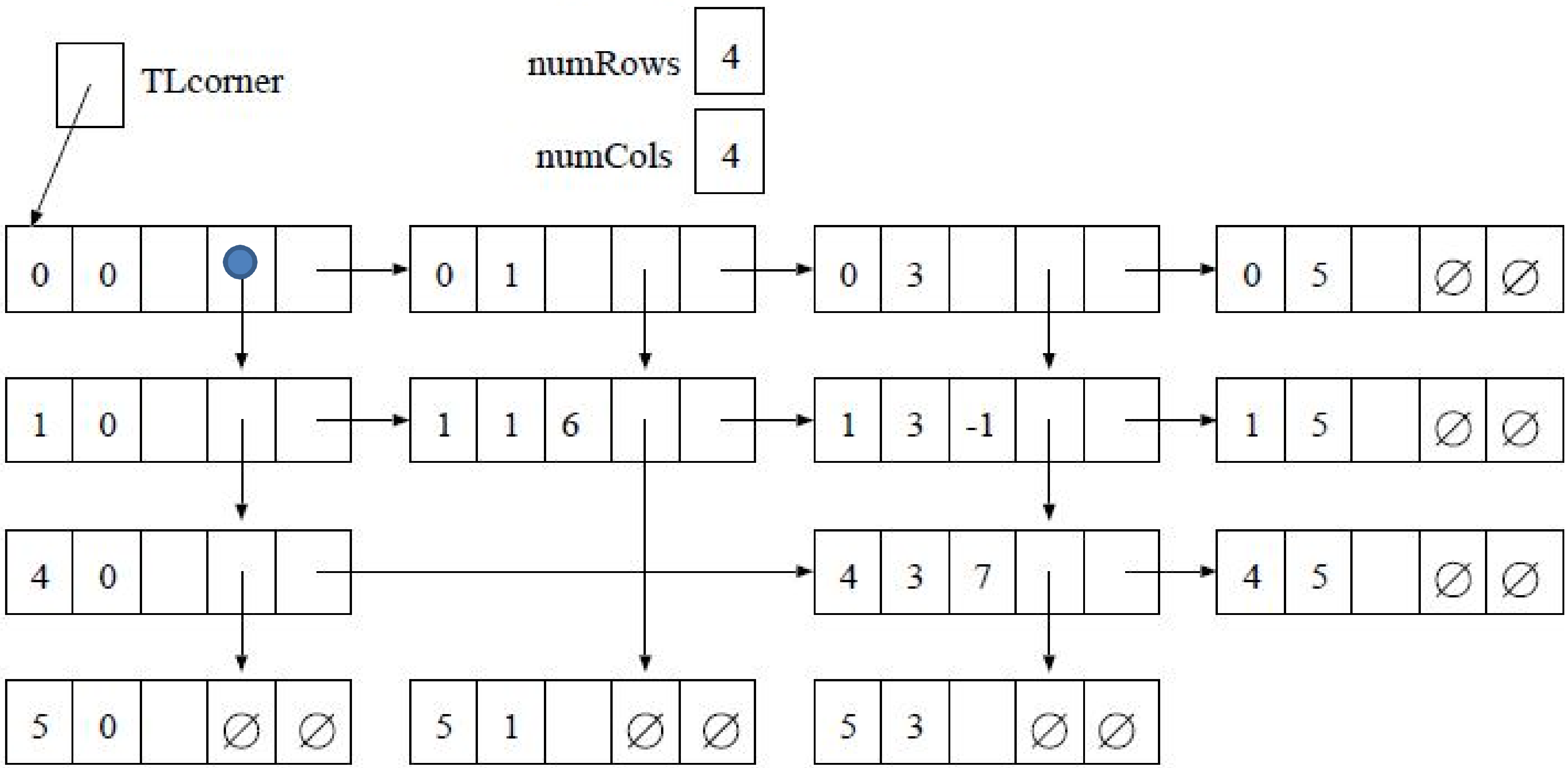
setValue(3,2,0) does nothing:



Example

If `newValue` is 0 and there is a node for entry `[R,C]`, then delete that node. If row `R` and column `[C]` both contain other non-zero entries, simply unlink that node from the row list and the column list.

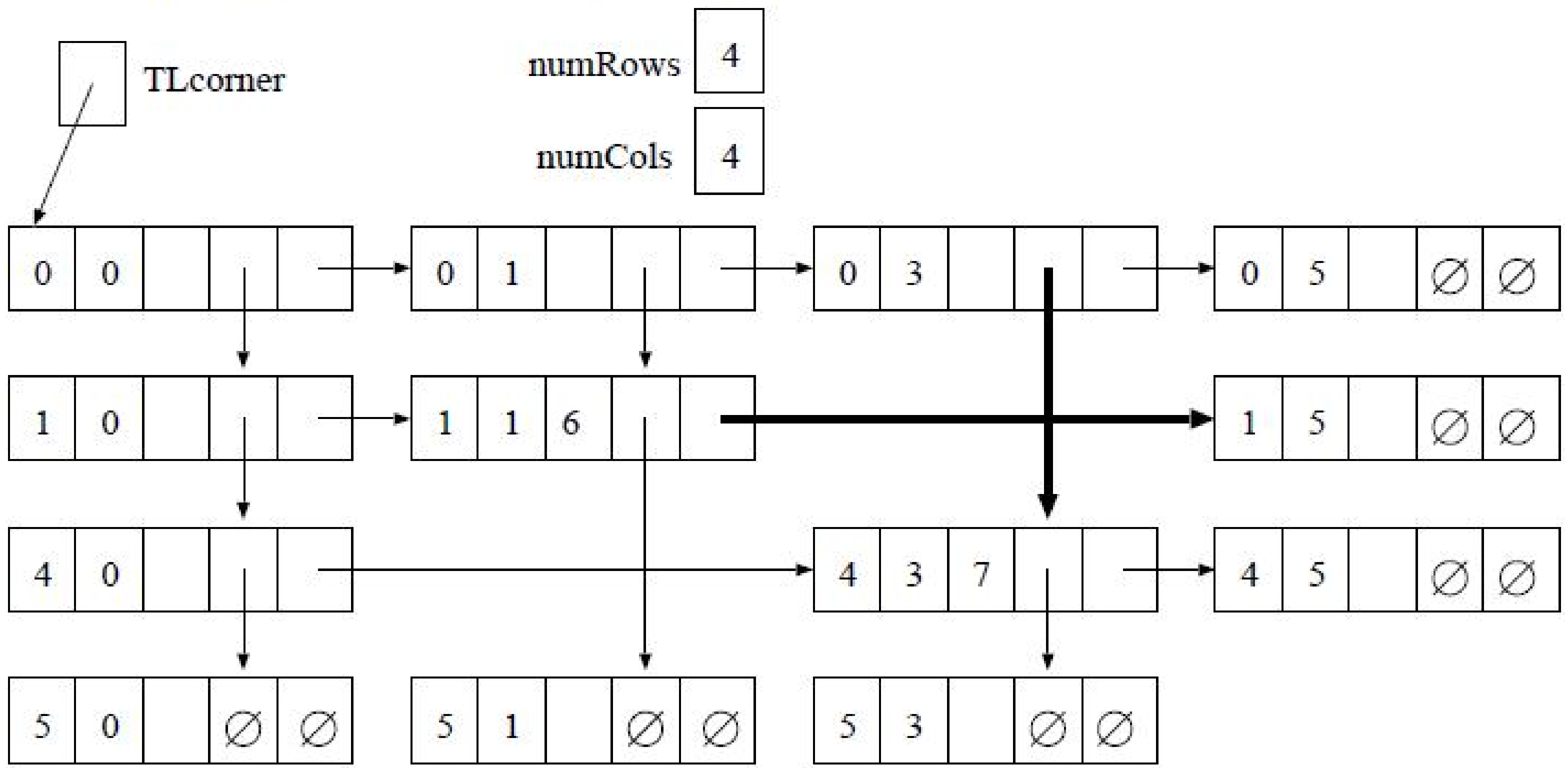
setValue(1,3,0) causes the following changes:



Example

If `newValue` is 0 and there is a node for entry `[R,C]`, then delete that node. If row `R` and column `[C]` both contain other non-zero entries, simply unlink that node from the row list and the column list.

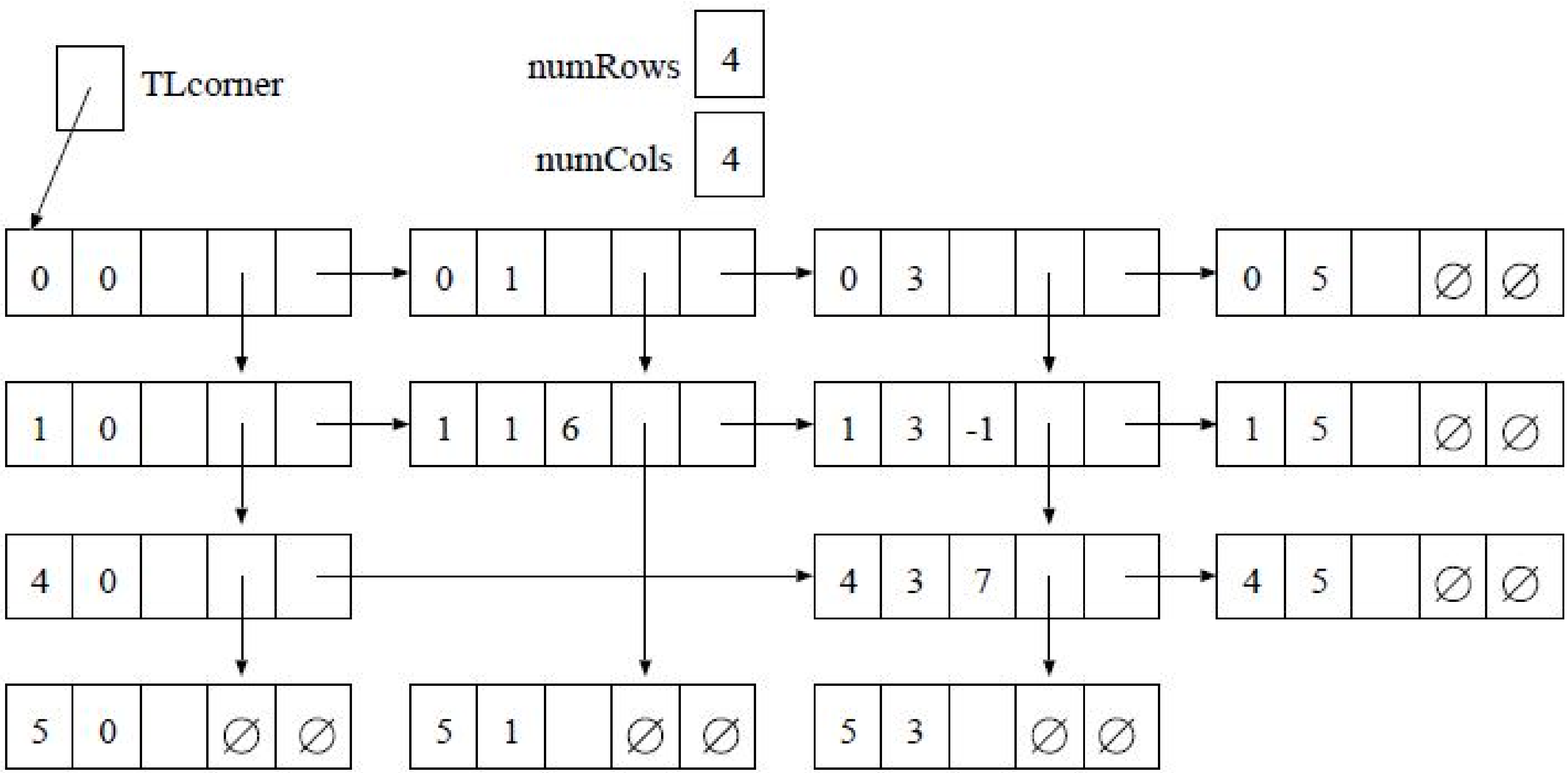
setValue(1,3,0) causes the following changes:



Example

If `newValue` is 0, a node for entry `[R,C]` exists, but row `R` contains no other non-zero entries, then you have to delete the entire row `R`.

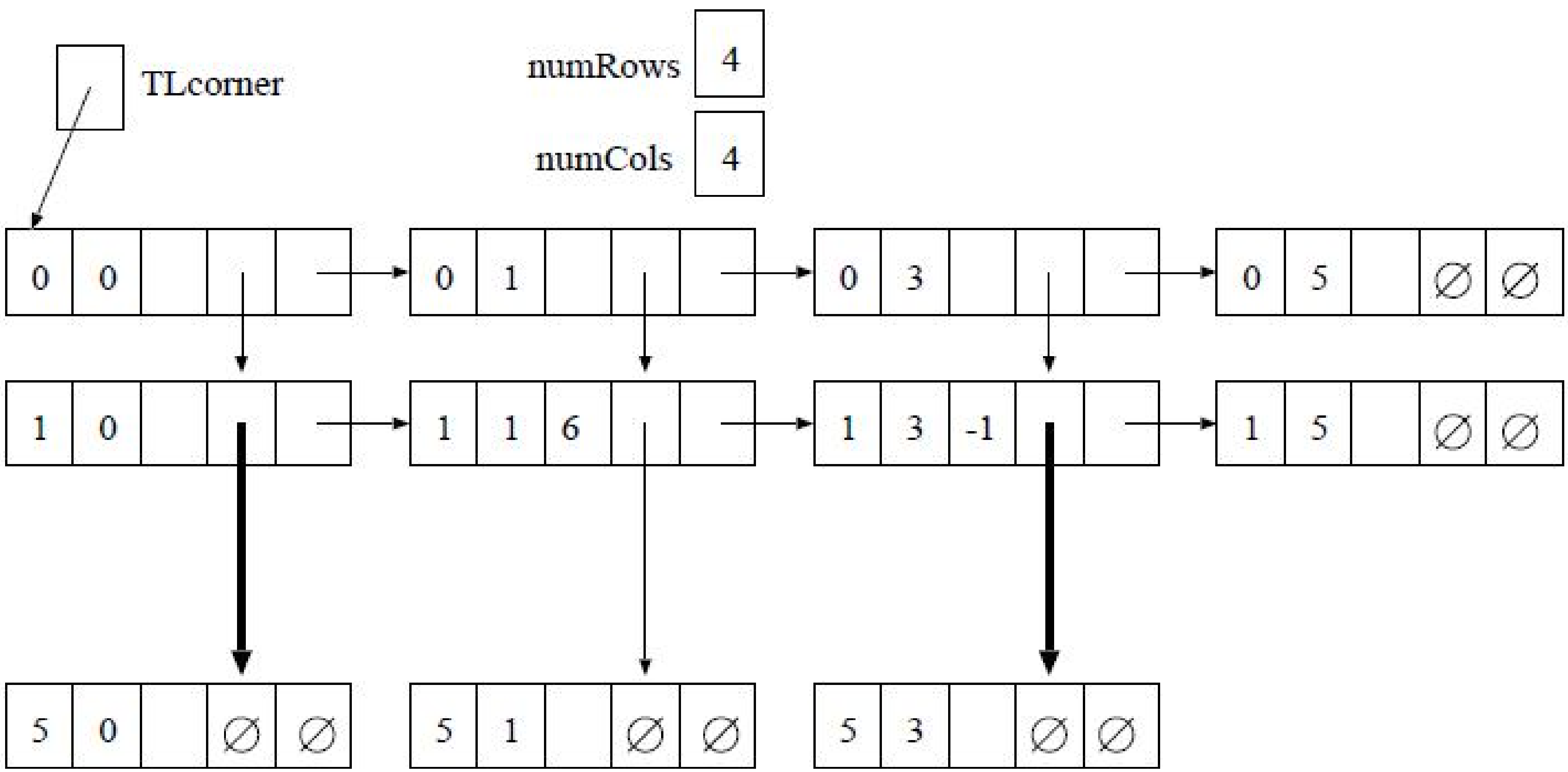
setValue(4,3,0)



Example

If `newValue` is 0, a node for entry `[R,C]` exists, but row `R` contains no other non-zero entries, then you have to delete the entire row `R`.

setValue(4,3,0) causes the following changes:



Example

- If newValue is 0 and you delete the only non-zero value in column C, then you have to delete the column list for column C.

- You might have to delete both the list for row R and the list for column C.

(Pictures omitted.)