4. Lists

h. choi

hchoi@handong.edu

Agenda

- Singly Linked Lists
- Linked Stacks and Queues
- Circularly Linked Lists
- Sparse Matrices
- Doubly Linked Lists

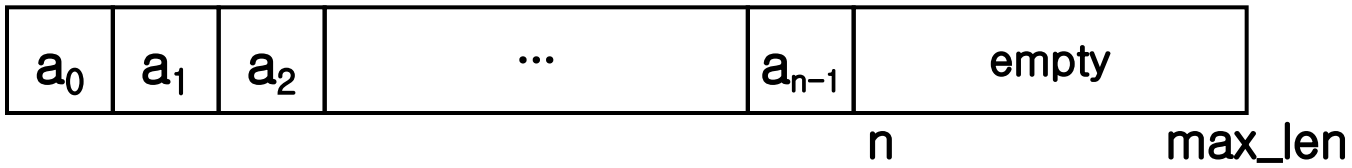
What is a List?

- A list is a varying-length, linear collection of homogeneous elements
- Linear means:
  - Each list element has a unique predecessor (except the first)
  - Each element has a unique successor (except the last)

e.g.) Array[3] Array[4] Array[5]

Introduction

- Problems of sequential representation (array)
  - Fixed size
  - Inefficiency in insertion, deletion



- Alternative: linked representation
  - Self-referential pointer

```
typedef struct t_list_node {
   string data;
   struct t_list_node *link;   // pointer to next element
} list_node;
```

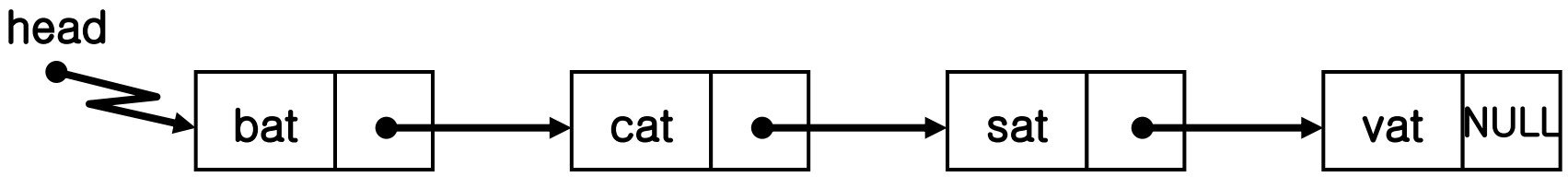
Logical link
(actual location
is not important)

Singly Linked Lists

- Linked list: connection of nodes
  - **Node**: element of linked list
    - Composed of **data** and **pointer to the next node**
  - Nodes don't need to reside in physically sequential locations

Ex)

```
typedef struct t_list_node {
   string data; //  char data[4];      // data, could be a structure
   struct t_list_node *link;          // pointer to the next node
} list_node;
list_node *head;                      // pointer to the first element
```



class

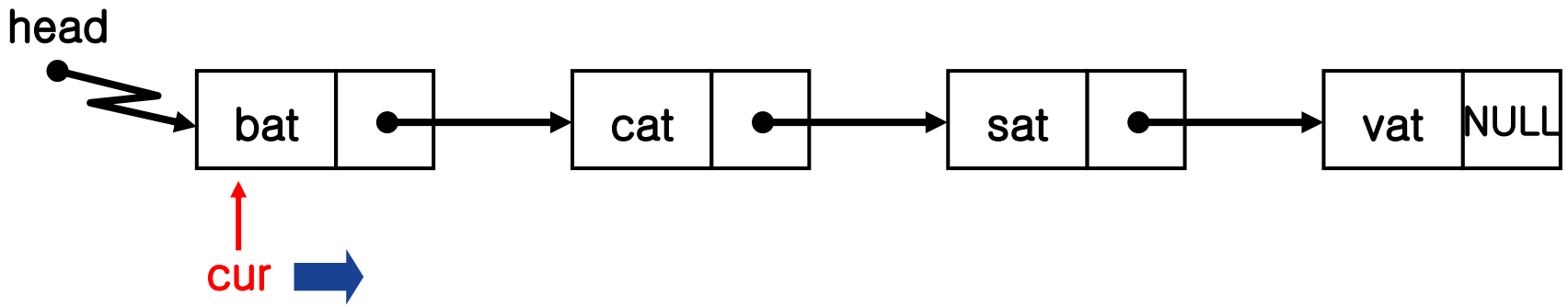
```
typedef struct t_list_node {
    string data;
    struct t_list_node *link;
} list_node;

class DList
{
  private:
    list_node *head;

  public:
    DList();
    ~DList();
    void delete_list(list_node* node_ptr); // for ~Dlist()
    list_node* Retrieve(string data);
    list_node* Retrieve(int index);
    void Insert(string data);
    void Delete(string data);
    void Invert();
    void Print();
    bool IsEmpty();
};
```

implement the methods

Traversal

- Printing all nodes in a linked list

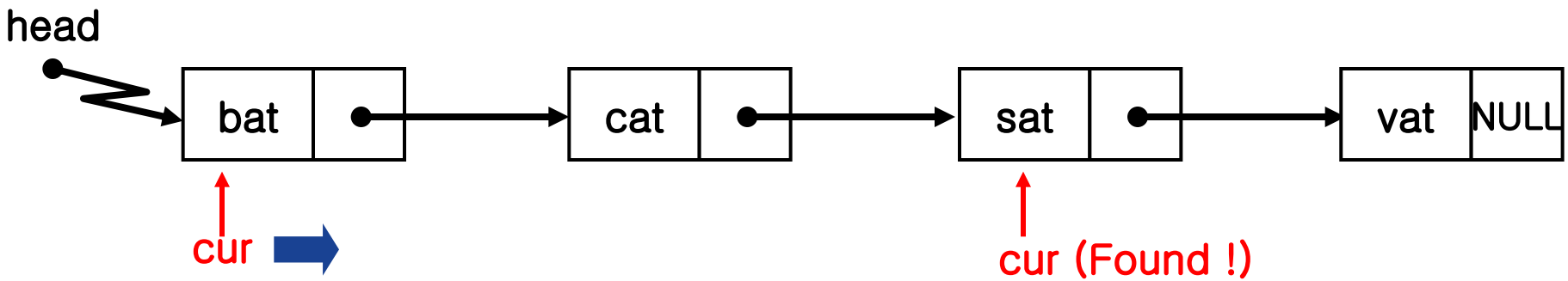
```
list_node *cur = NULL;
for(cur = head; cur != NULL; cur = cur->link){
   cout << cur->data << end;
}
```



Finding An Element

- Follow links from head pointer *head* until the target node is found → O(len)

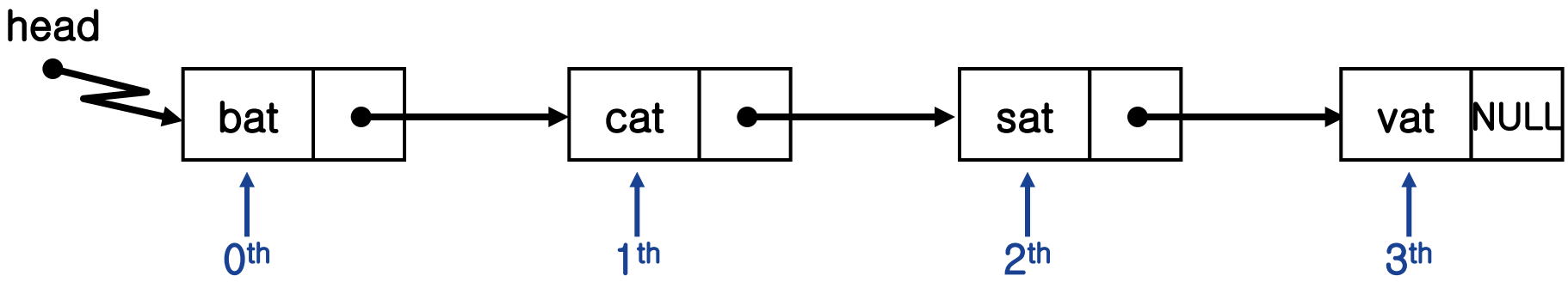
```
list_node *cur = NULL;
for(cur = head; cur != NULL; cur = cur->link){
   if(cur->data.compare("sat") == 0)
      break;                    // found!!
}
// if cur != NULL, found!
```



Finding *n*-th Element

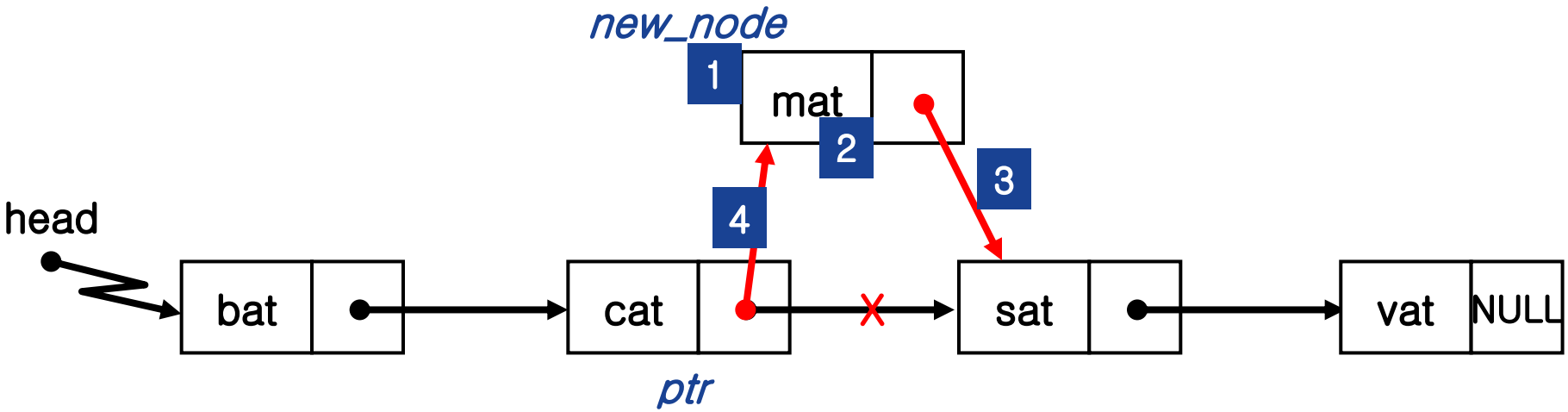
- Follow links from head pointer *head* n times → O(n)

```
list_node *cur = NULL;
int i;
for(cur=head, i=0; cur!=NULL && i<n; cur=cur->link, i++){
   // empty body
}
// if cur != NULL, n-th node is found
```



Insertion

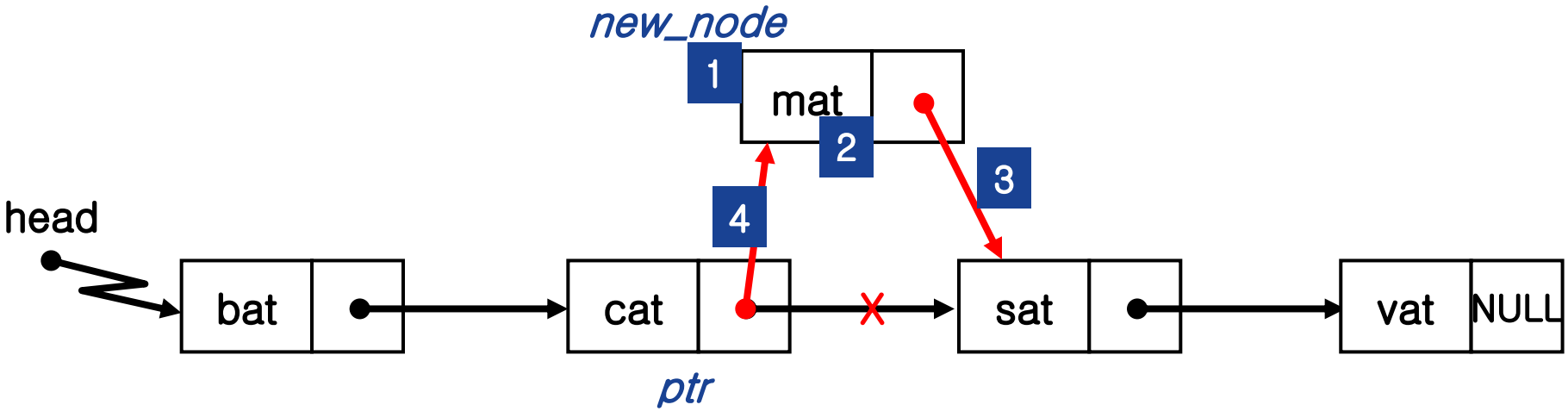
- Inserting a *new_node* after a node *ptr* → O(1)
  1. Get (or allocate) *new_node*
  2. Set data fields of *new_node*
  3. Set *new_node->link* to address found in link of *ptr*
  4. Set *link* of *ptr* node to *new_node*



Insertion

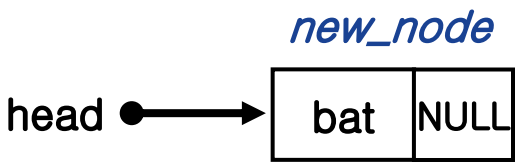
- Insert *new_node* after *ptr*

```
// list_node *new_node = (list_node *)malloc(sizeof(list_node));
list_node *new_node = new list_node; // C++ style
new_node->data = “mat”;
new_node->link = ptr->link;
ptr->link = new_node;
```

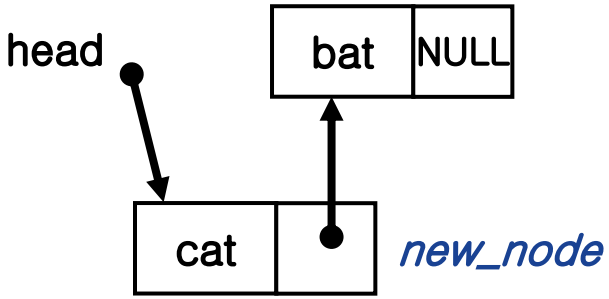


Insertion at the beginning of the list

```
void DList::Insert(string data){
    //list_node *new_node = (list_node*) malloc(sizeof(list_node));
    list_node *new_node = new list_node;
    new_node->data = data;
    new_node->link = head;
    head = new_node;
}
```



add a node when it is empty

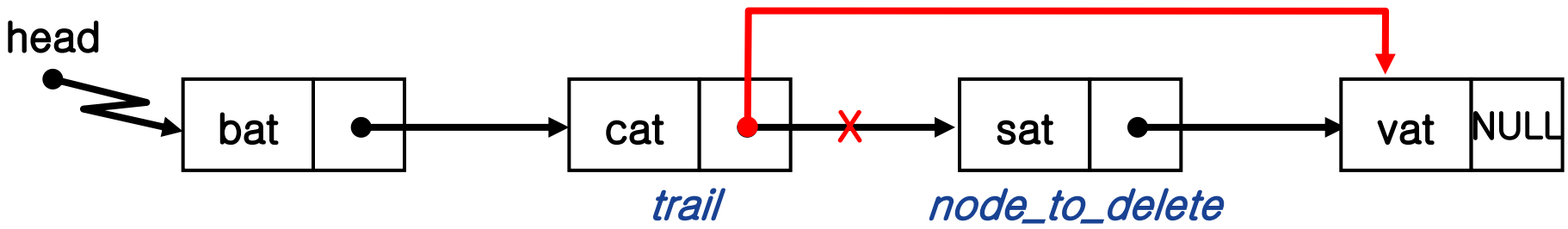


add a node when it has some nodes.

Deletion

- Deleting a node → given trail, O(1)
  1. Find element immediately precedes *node_to_delete*, say *trail*
     → O(len)
  2. Set *trail ->link* to link of *node_to_delete*
  3. Discard (or deallocate) *node_to_delete*

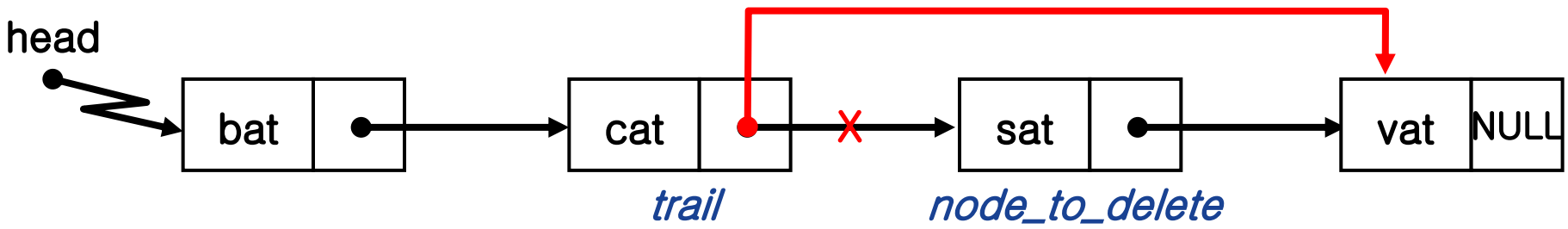
Note: pointer to previous node is required



Deletion

- Delete *node_to_delete* after *trail*

```
trail->link = node_to_delete->link;
// free(node_to_delete);
delete node_to_delete; // C++ style
```



Deletion

```
void DList::Delete(string data){
    list_node* prev;
    list_node* curr = head;

    while(curr && curr->data.compare(data) != 0){
        prev = curr;                          // loop to find the target
        curr = curr->link;
    }

    if(curr){  // if there is the target
        if(curr == head) head = curr->link;  // target is the first one
        else             prev->link = curr->link;

        // free(curr);
        delete curr;
    }
}
```

Creation / Empty Check

- Creating an empty linked list

```
list_node *head = NULL;
```

- Empty check

```
int IsEmpty(list_node *ptr)
{
    return (ptr == NULL);
}
```

```
bool DList::IsEmpty() // as a member function of class DList
{
    return (head == NULL);
}
```

Destructor

recursive approach

```
DList::~DList(){
    delete_list(head);
}
void DList::delete_list(list_node* node_ptr){
    if (node_ptr != NULL) {
        delete_list(node_ptr->link);
        delete node_ptr;
    }
}
```

iterative approach

```
DList::~DList(){
    list_node *curr=head, *next;
    while(curr){
        next = curr->link;
        delete curr;
        cur = next;
    }
}
```

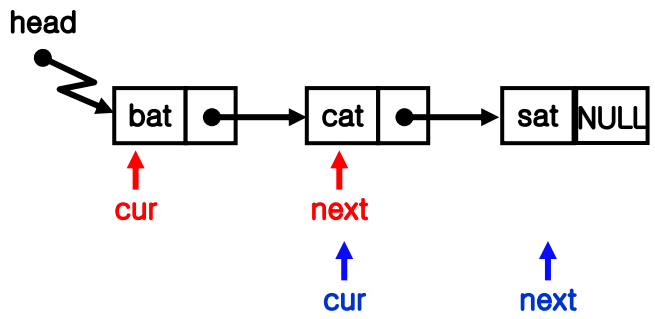
GetLength and Invert

- GetLength → O(len)

```
int n=0;
list_node *cur = NULL;
for(cur=head, n=0; cur!=NULL; cur=cur->link, n++);
```

- inverting the list

```
void DList::Invert () {
    if (IsEmpty()) return;
    list_node *cur = head, *next = head->link;
    while(cur){
        if (cur==head) cur->link = NULL;
        else {
            cur->link = head;
            head = cur;
        }
        cur = next;
        if (cur) next = next->link;
    }
}
```



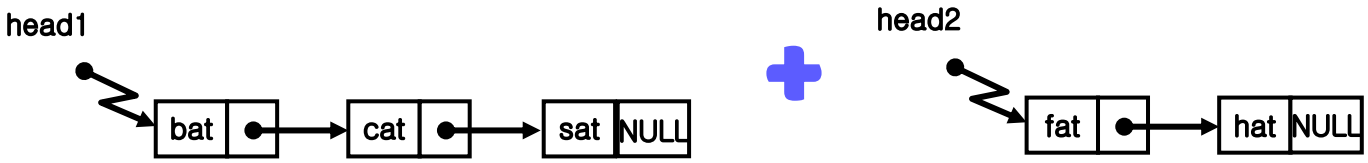
main for DList

```
#include <iostream>
#include "DList.h"

using namespace std;

int main()
{
    DList* list = new DList();
    list_node* node;

    list->Print();
    list->Insert("111");
    list->Insert("222");
    list->Insert("333");
    list->Print();

    list->Invert();
    list->Print();

    node = list->Retrieve(3);
    node = list->Retrieve("444");
    list->Delete("333");
    list->Print();

    delete list;
}
```

Additional List Operations

- Concatenation of linked lists

practice



- Copying linked lists

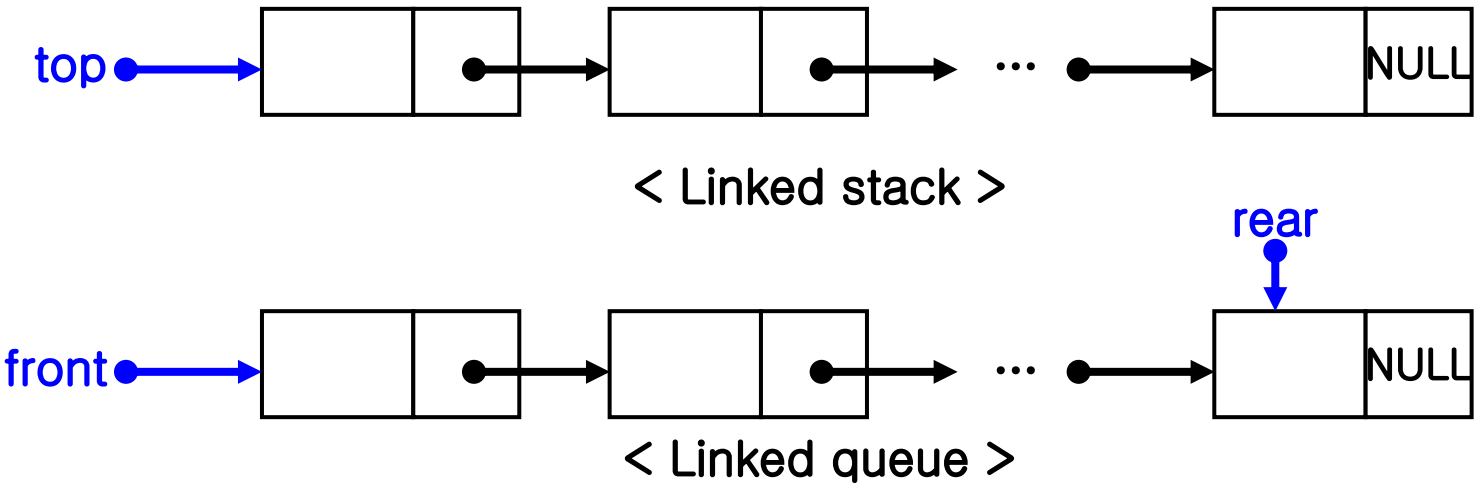
Agenda

- Singly Linked Lists
- Linked Stacks and Queues
- Circularly Linked Lists
- Sparse Matrices
- Doubly Linked Lists

Linked Stacks and Queues

- Problems of sequential stacks/queues
  - Maximum size is fixed
  - If multiple stacks and queues coexist, memory is utilized inefficiently

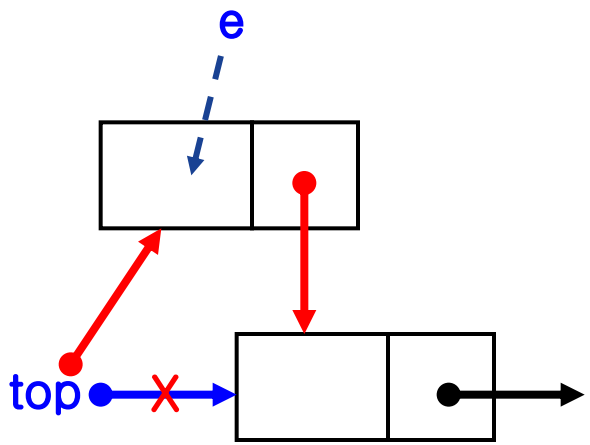
- Alternative: linked stacks/queues



Dynamically Linked Stacks

- Push into linked stack

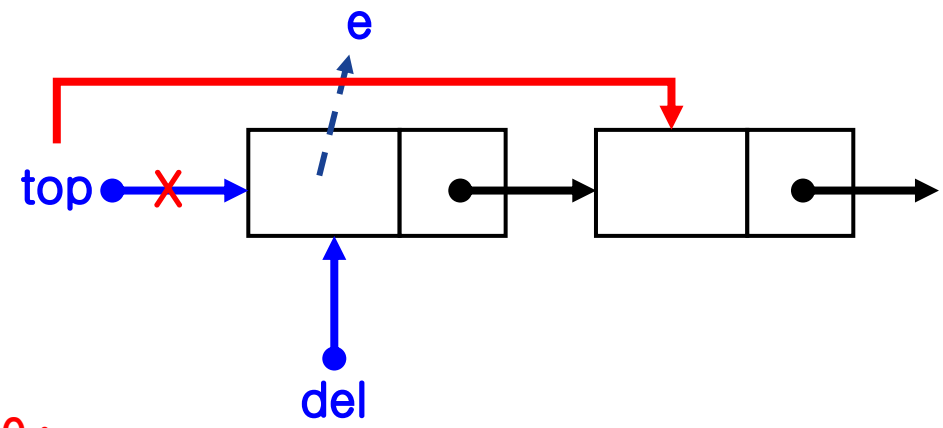
```
void Stack::Push(Element e)
{
    stack_node *new_node = new stack_node;

    if(new_node == NULL)
        return;

    new_node->data = e;
    new_node->link = top;
    top = new_node;
}
```



Element can be any built-in or user-defined data type

Dynamically Linked Stacks

- Pop from linked stack

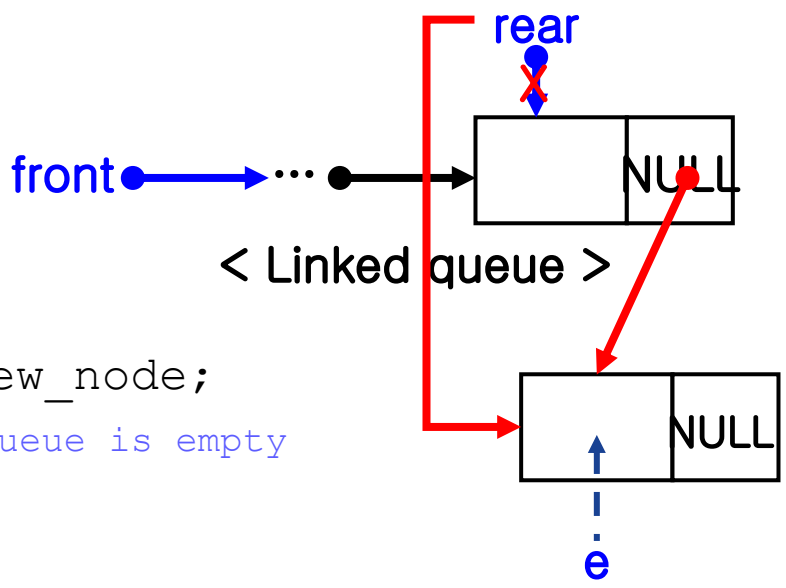



```
Element Stack::Pop()
{
    Element e;
    stack_node *del;

    if(top == NULL)
        return (Element)0;

    e = top->data;    // data to return
    del = top;        // delete node
    top = top->link;
    delete del;       // deallocate the node
    return e;
}
```

Dynamically Linked Queues

- AddQ into linked queue

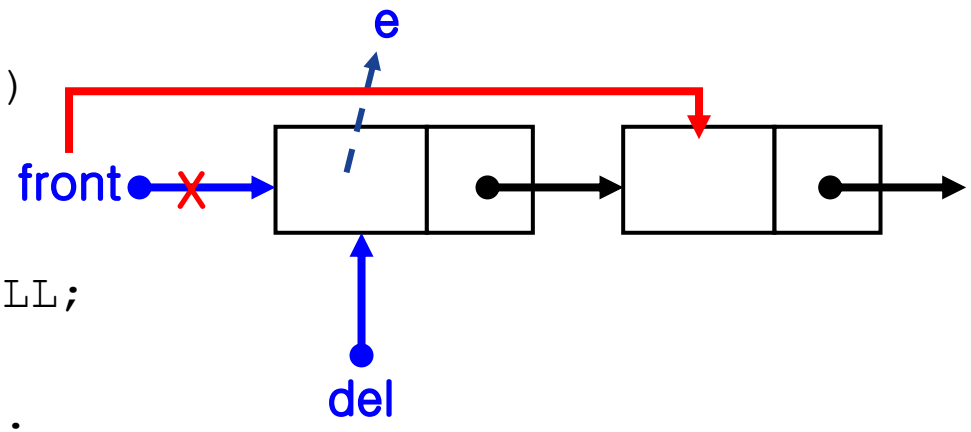
```
void Queue::AddQ(Element e)
{
   queue_node *new_node = new queue_node;

   if(new_node == NULL)
       return;

   new_node->data = e;
   new_node->link = NULL;
   if(front) rear->link = new_node;
   else front = new_node;//queue is empty
   rear = new_node;
}
```



Dynamically Linked Queues

- DeleteQ from linked queue

```
Element Queue::DeleteQ()
{
    Element e;
    queue_node *del = NULL;
    if(front == NULL)
        return (Element)0;

    e = front->data;
    del = front;
    front = front->link;
    if(front==NULL) rear = NULL; // the only one is deleted
    delete del;
    return e;
}
```



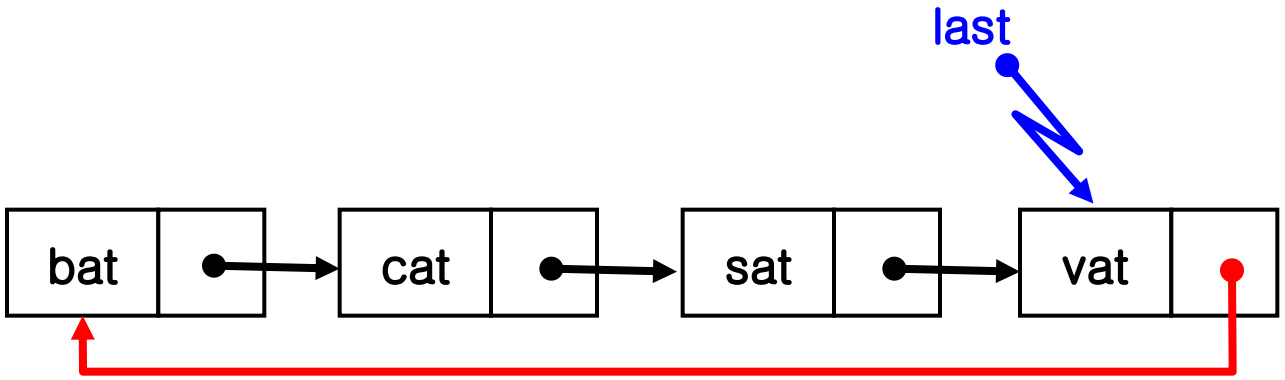
Agenda

- Singly Linked Lists
- Linked Stacks and Queues
- Circularly Linked Lists
- Sparse Matrices
- Doubly Linked Lists

Circularly Linked Lists

- Circularly linked list: a linked list in which the last node points the first node

  ➔ We can traverse whole list starting from any node.



- Usually, a circular linked list is handled by pointer to last element.
  - *head* can be obtained from *last->link.*

Traversal on Circularly Linked List

- Traversal on null-terminated linked list

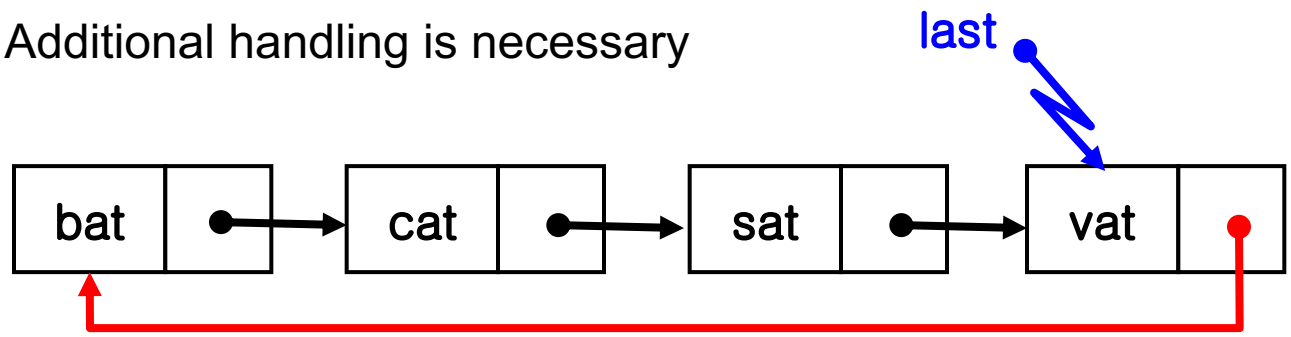
```
list_node *cur = NULL;
for(cur = last->link; cur != NULL; cur = cur->link){
   cout << cur->data << endl;                    // infinite loop
}
```

- Traversal on circularly linked list

```
list_node *cur = NULL;
for(cur = last->link; cur != last; cur = cur->link){
   cout << cur->data << endl;
}
```

➡ Correct ?

  - Additional handling is necessary

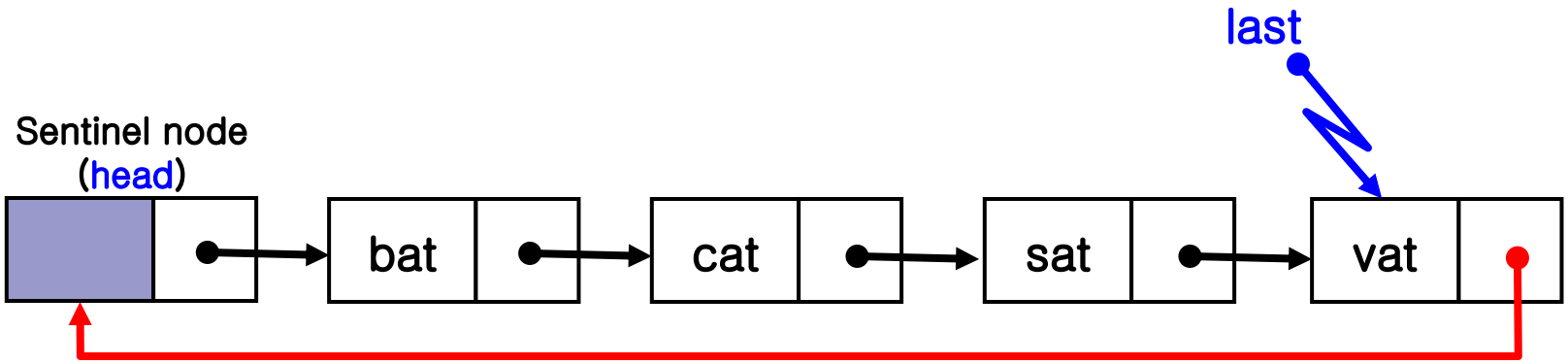


Traversal on Circularly Linked List

- Sentinel(dummy) node
  - Node at first or last, not used to store data, but to simplify or speed up some operations

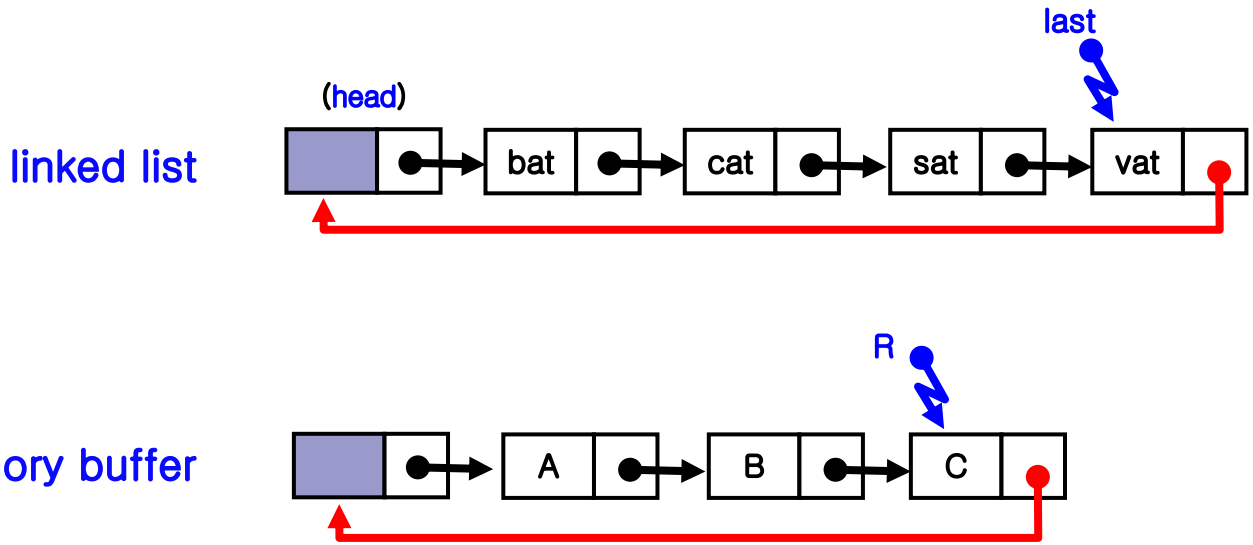
- Traversal on circularly linked list with sentinel node

```
list_node *cur = NULL;
for(cur = head->link; cur != head; cur = cur->link){
    cout << cur->data << endl;
}
```



maintaining “freed” nodes

- we can keep another linked list, R, as a memory buffer.
  - when we need a new node, we get one from R.
  - when we delete one, we return it back to R.
- only when R is empty, do we need to use malloc (or new) to create a new node.



Agenda

- Singly Linked Lists
- Linked Stacks and Queues
- Circularly Linked Lists
- Sparse Matrices
- Doubly Linked Lists

Representation of Matrix

- Matrix

$$\begin{pmatrix} a_{0,0}, a_{0,1}, \ldots, & a_{0,n-1} \\ a_{1,0}, a_{1,1}, \ldots, & a_{1,n-1} \\ \ldots, & a_{i,j}, \ldots \\ a_{m-1,0}, a_{m-1,1}, \ldots, & a_{m-1,n-1} \end{pmatrix}$$

- Representation of (dense) matrix: array
  - What if matrix is sparse?

Sparse Matrix

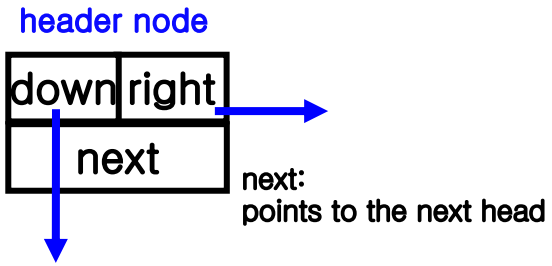
- Sparse matrix: A matrix whose non-zero elements are sparse
  - Most of elements have meaningless values (zeroes)
  - Array representation is **inefficient** for sparse matrix

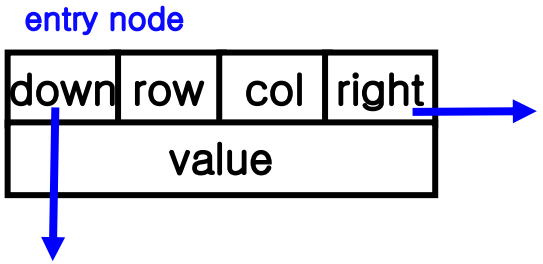
	col0	col1	col2	col3	col4	col5
row0	15	0	0	22	0	−15
row1	0	11	3	0	0	0
row2	0	0	0	−6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

Linked Representation of Sparse Matrix

- Representation using 2D linked list
- Node for non-zero element

```
typedef struct tMatrix_node{
    int value;
    int row, col;
    struct tMatrix_node *down, *right;
} matrix_node;
```



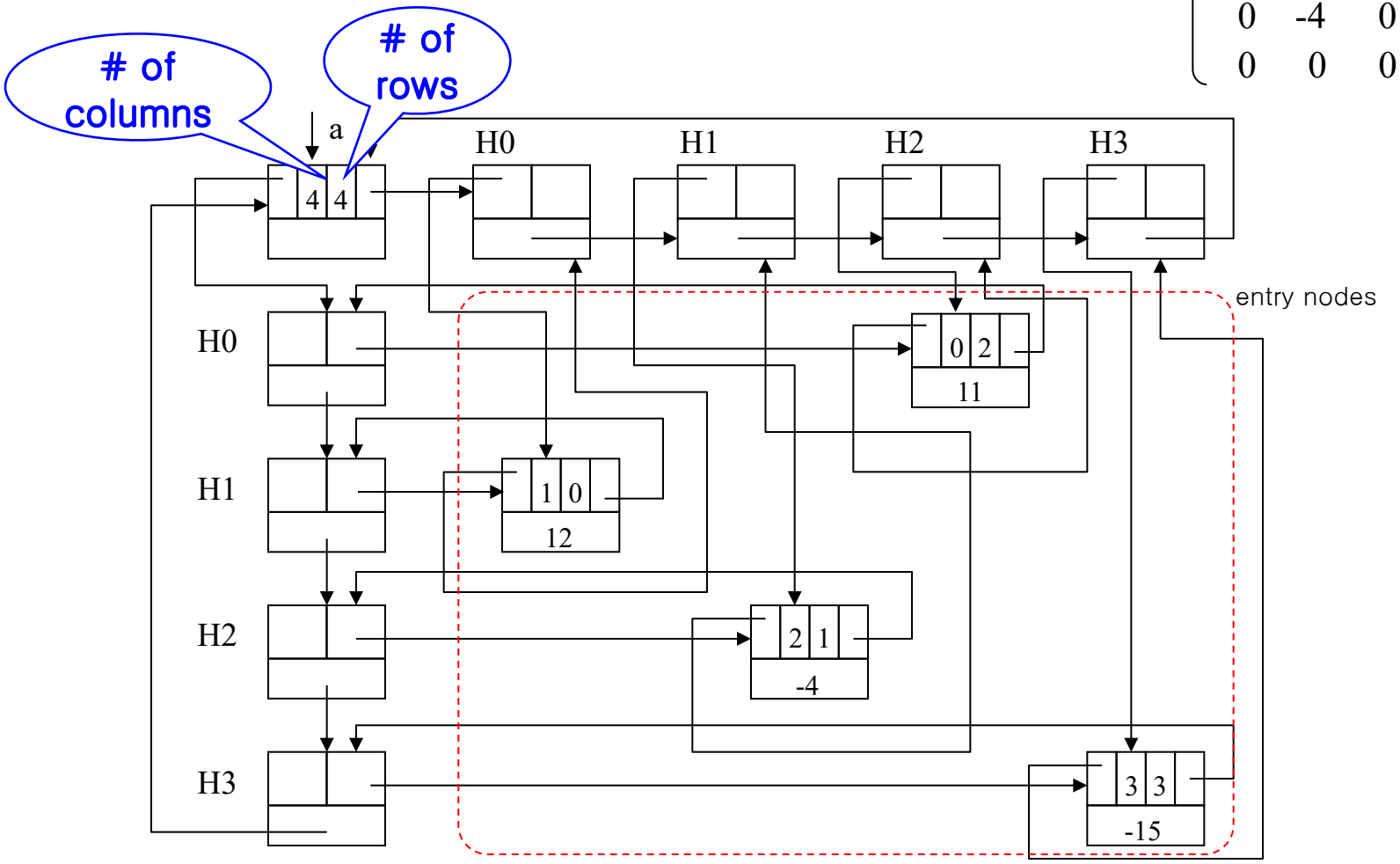


	col0	col1	col2	col3	col4	col5
row0	15	0	0	22	0	−15
row1	0	11	3	0	0	0
row2	0	0	0	−6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

Linked Representation of Sparse Matrix

- Example

$$\begin{pmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{pmatrix}$$

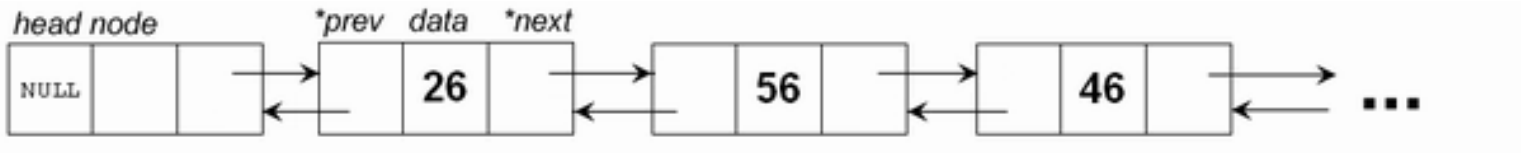


Agenda

- Singly Linked Lists
- Linked Stacks and Queues
- Circularly Linked Lists
- Sparse Matrices
- Doubly Linked Lists

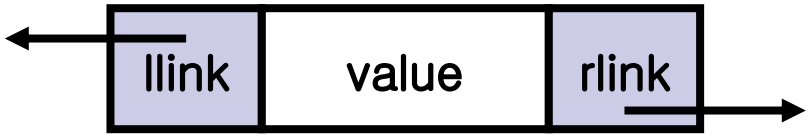
Doubly Linked Lists

- Doubly linked list: linked list, in which each node has two links, one to the previous node and one to the next node.



- Node of doubly linked list

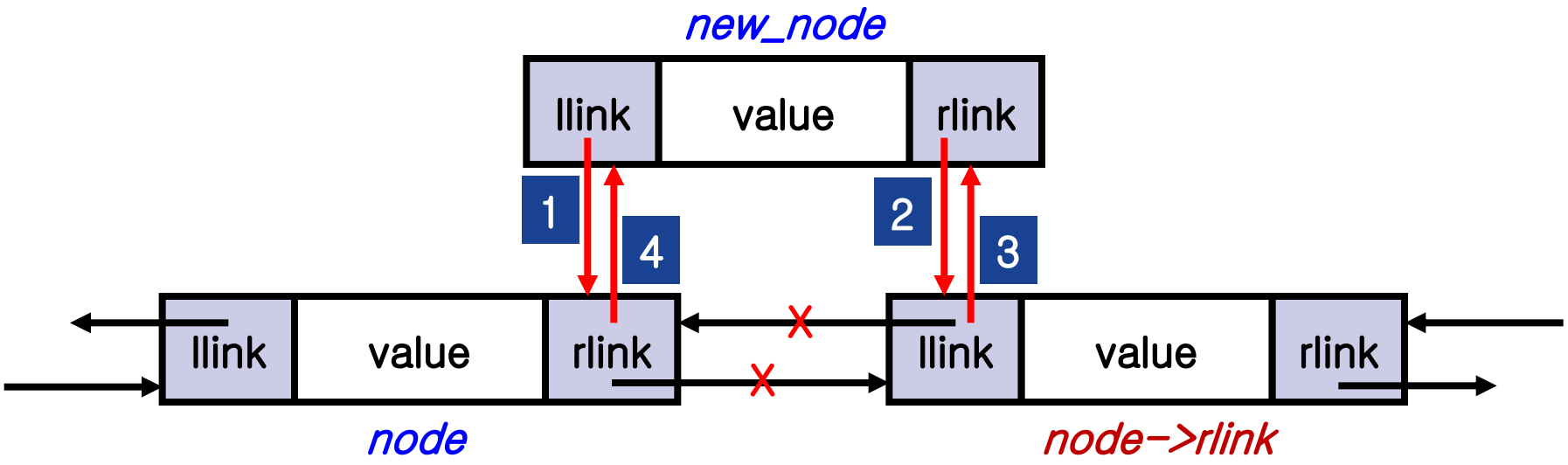
```
typdef struct tNode {
    Element value;
    struct tNode *llink, *rlink;
} Node;
```



Insertion in Doubly Linked Lists

- Inserting *new_node* to right of *node*

```
new_node->llink = node;
new_node->rlink = node->rlink;
node->rlink->llink = new_node;
node->rlink = new_node;
```



Doubly Linked Lists

- Advantage
  - Backward traversal is efficient
  - Insertion / deletion is possible in constant time
- Disadvantage
  - Requires more space and time

questions or comments?

hchoi@handong.edu