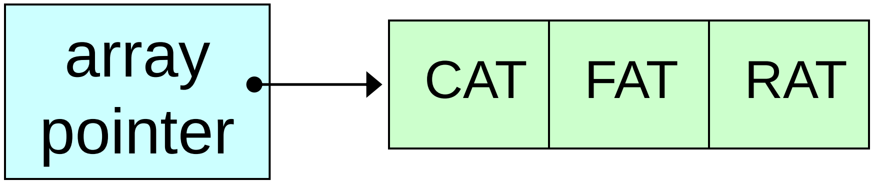
Linked Lists
(chapter 4)

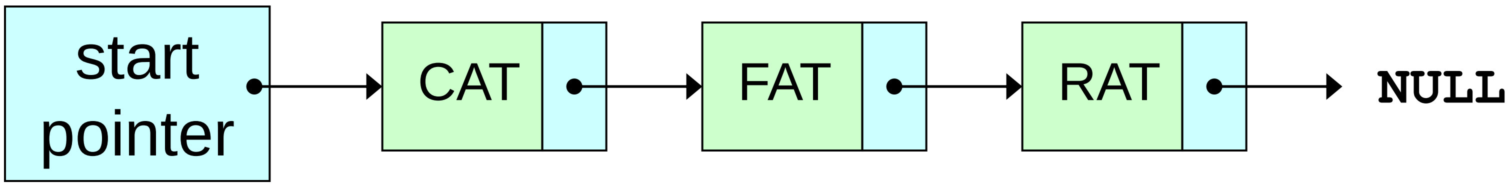
- Singly-linked lists
- Representation in C++
- Circular linked lists
- Linked-list representations of previous ADTs:
  - Stacks and queues
  - Polynomials
  - Sparse matrices
- Application problem: Equivalence classes
- Doubly-linked lists

Linked Lists vs. Arrays

- An array of three 3-letter words

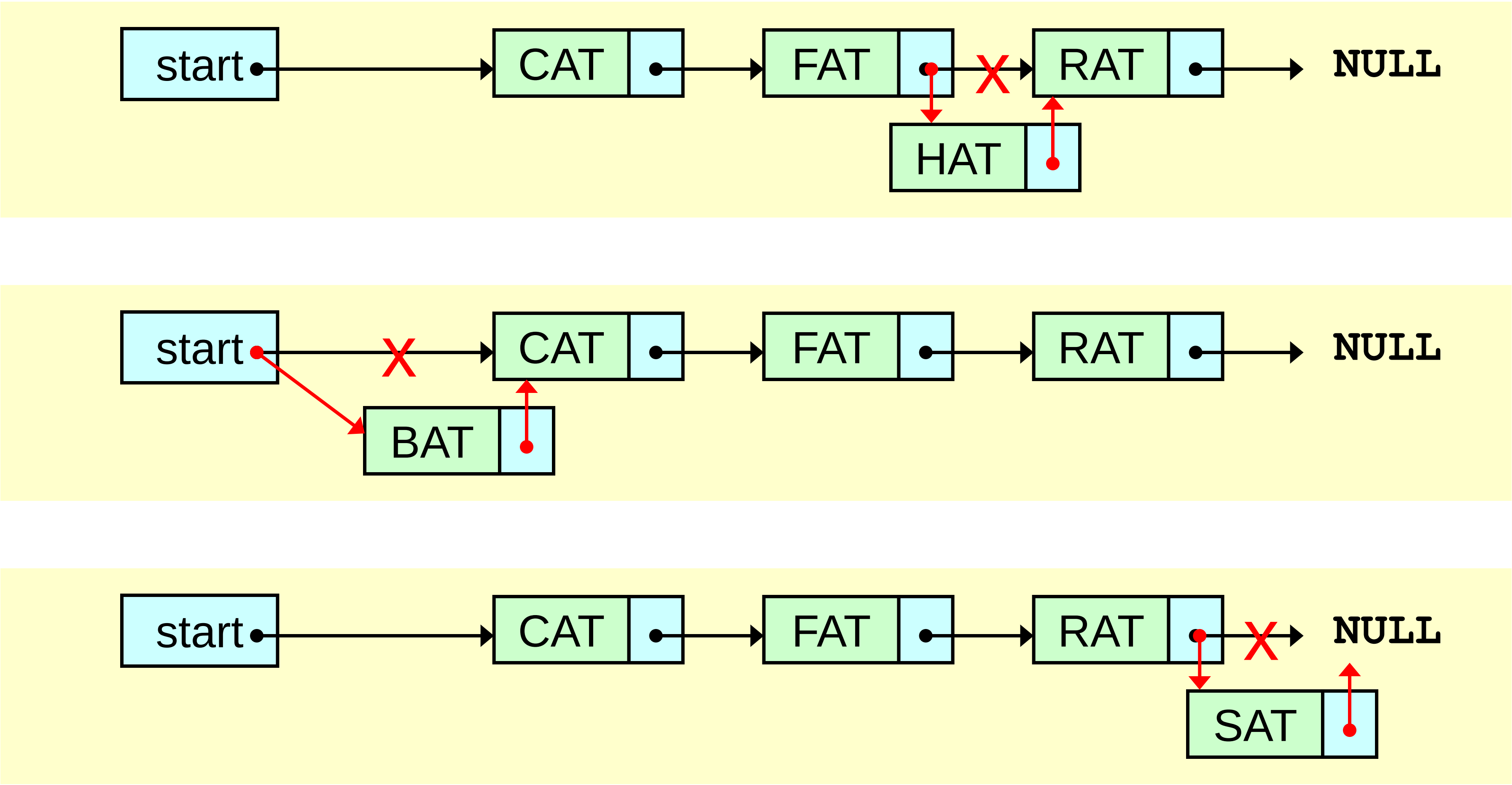


- A singly-linked list (chain) of three 3-letter words

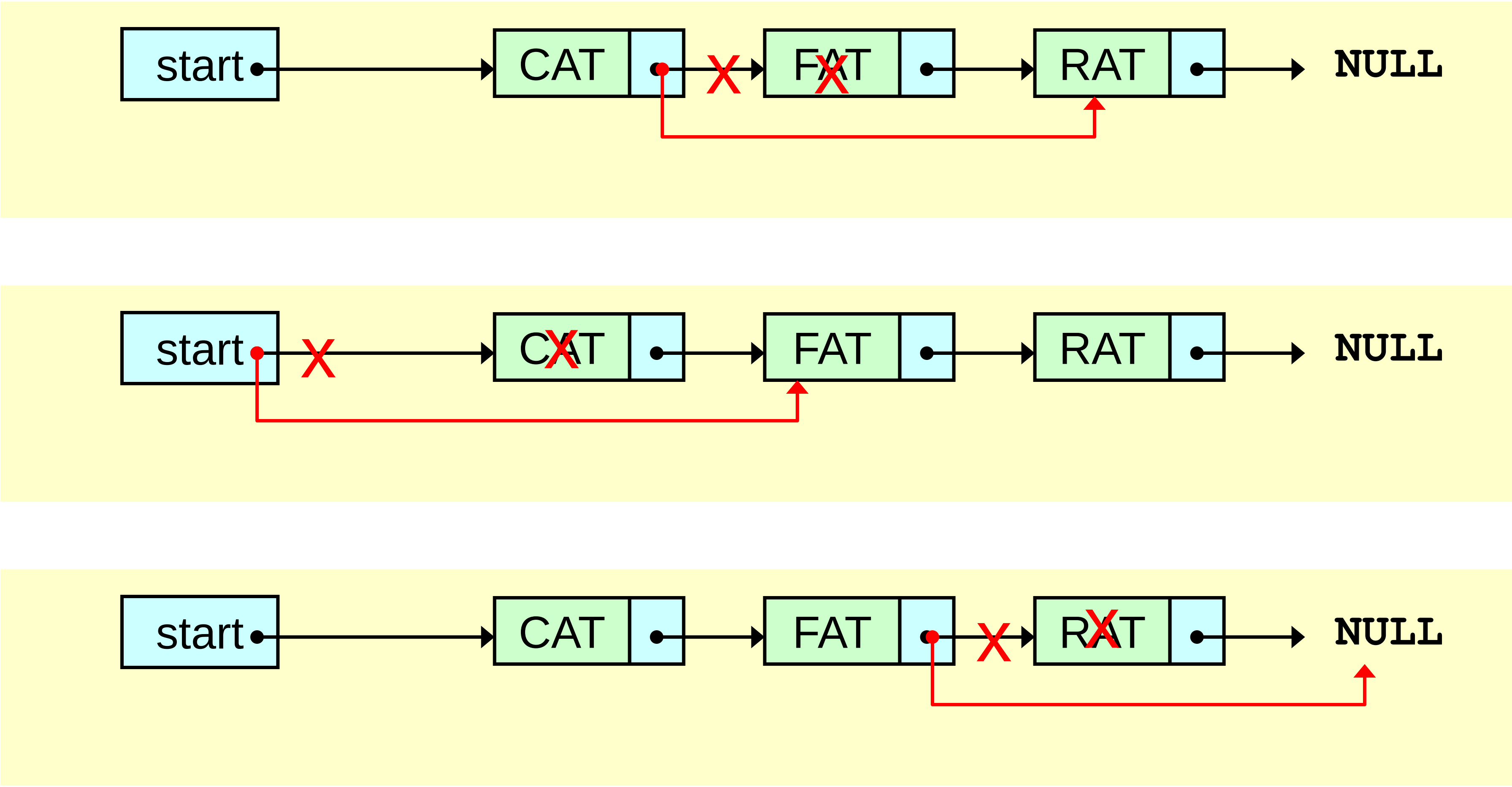


- Advantages of linked lists
  - No need to keep items in the memory in the correct order
  - No data movement during <u>deletion</u> and <u>insertion</u>
- Of course, you need to be really comfortable with pointers.

Linked Lists Operations: Insertion



Linked Lists Operations: Deletion



Representation of a List Node in C++

For our data item (3-letter words):

```
class ThreeLetterNode
{
private:
   char data[3];
   ThreeLetterNode * link;
}
```

data

pointer to another node

This is called a **self-referencing class**.

A list of such nodes is a **singly-linked list** (also called a **chain** in the textbook) because each node has one link to another node.

Representation of Linked Lists in C++

- Ideally, we should only be able to access contents of list nodes through list operations (such as insertion and deletion).
- For better data encapsulation: Use two classes.
  - A class for the nodes (e.g., `ThreeLetterNode`).
  - A class for the linked list, which contains objects of the node class. This is a **container class**. (Stacks and queues are also container classes.)

The Basic Chain Class Template

```
template<class T> class Chain;  // forward declaration

template<class T> class ChainNode {
friend class Chain<T>;
private:
  T data;
  ChainNode<T> * link;
};

template <class T> class Chain {
public:
  Chain() {first = 0;} // initialize to an empty chain
  // operations of the list
  …
private:
  ChainNode<T> *first;
};
```

Data members of node objects are only accessible via list operations.

Chain Operation: Insertion

```
/* Insert after node x (or at the first position if x
is NULL */
template<class T>
void Chain<T>::Insert(const T &e, ChainNode<T> *x)
{
  if (x) {
    x->link = new ChainNode<T>(e, x->link);
  }
  else {
    first = new ChainNode<T>(e, (first) ? first : NULL);
  }
}
```

Chain Operation: Deletion

```
/* Delete node x, assuming y->link points to x if x is
not first */
template<class T>
void Chain<T>::Delete(ChainNode<T> *x, ChainNode<T> *y)
{
  if ((x) && (x == first || ((y) && y->link == x))) {
    if (x == first) first == first->link;
    else y->link = x->link;
    delete x;
  }
  else { ... } // exception
}
```

More Chain Operations

Attaching an Item to the End of a Chain

To do this (efficiently), we need one more data member **`last`**, which points to the last node of the chain. It is initialized to **`NULL`** for an empty chain.

```
template<class T>
void Chain<T>::InsertBack(const T& e)
{
  if (first) { // non-empty list
    last->link = new ChainNode<T>(e);
    last = last->link;
  }
  else first = last = new ChainNode<T>(e);
}
```

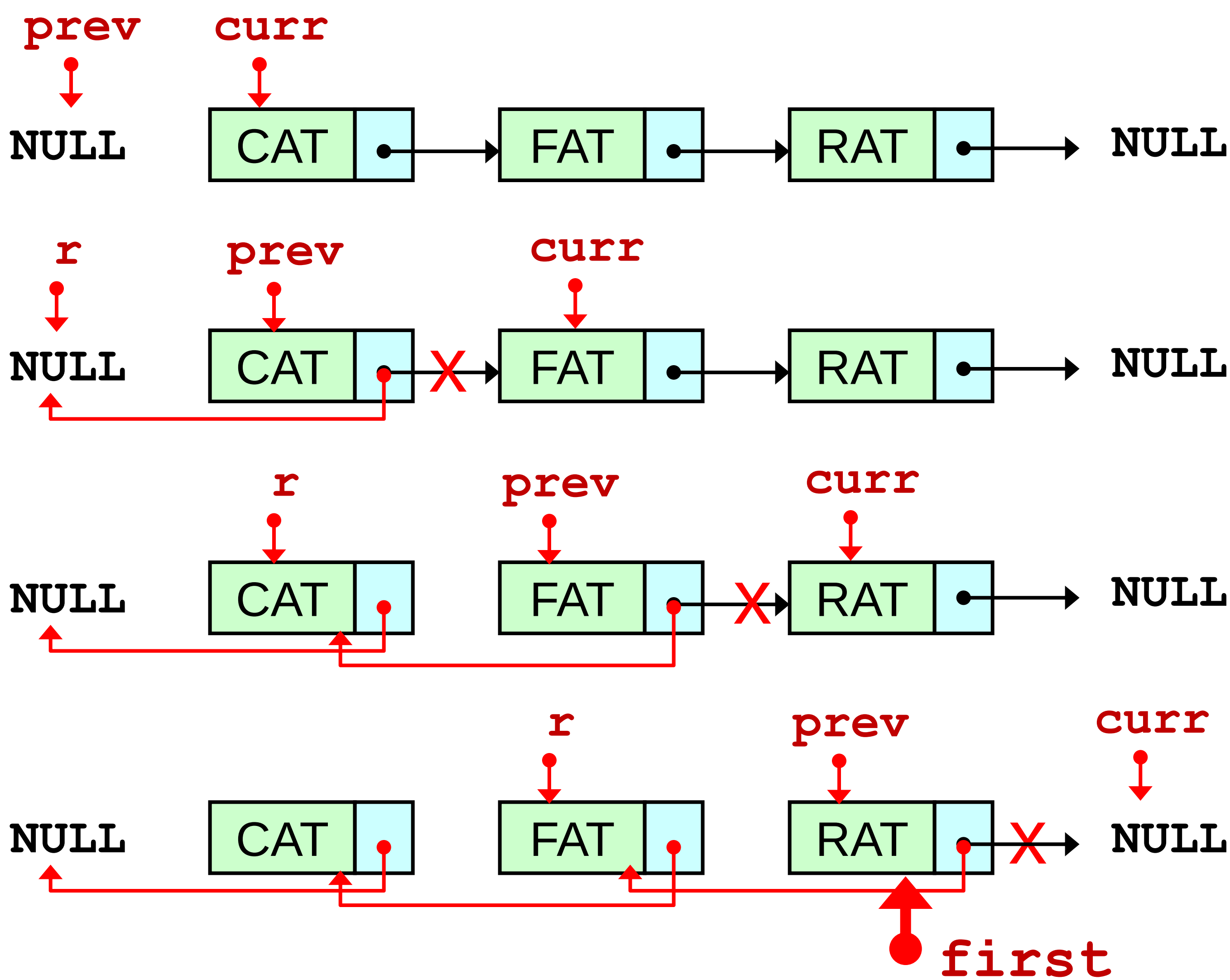
More Chain Operations

Concatenating Two Chains

```
template<class T>
void Chain<T>::Concatenate(Chain<T>& b)
{ // attach chain b to the end of *this
  if (first) { // *this is non-empty
    last->link = b.first;
    last = b.last;
  } else { // *this is empty; set *this to chain b
    first = b.first;
    last = b.last;
  }
  b.first = b.last = 0; // reset b to empty chain
}
```

More Chain Operations

Chain Reversal



More Chain Operations

Chain Reversal

```
template <class T>
void Chain<T>::Reverse()
{
  // curr is used to scan the nodes in the chain
  // prev points to the node after curr
  //    in the reversed chain
  ChainNode<T> *curr = first, *prev = 0;
  while (curr) {
    ChainNode<T> *r = prev;
    prev = curr;
    curr= curr->link;
    prev->link = r;
  }
  first = prev;
}
```

More Chain Operations

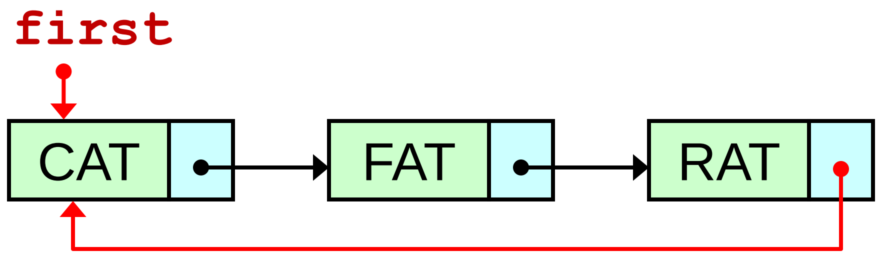
Deleting All Nodes

```
template<class T>
void Chain<T>::Clear()
{
  ChainNode<T> * next;
  while (first) {
    next = first->link;
    delete first;
    first = next;
  }
}
```

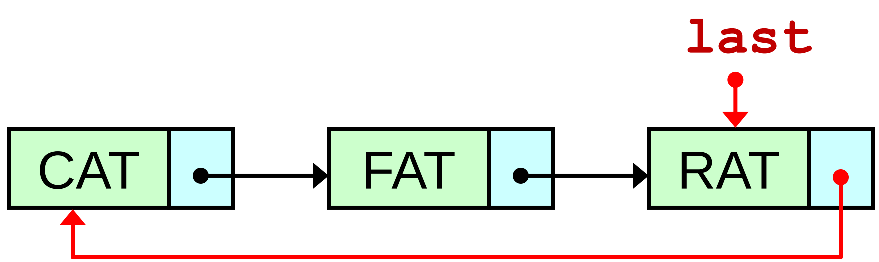
Q: For our chain operations, what modifications are necessary if we have the data member `last`?

Circular Lists

- The simple idea: The last node points back to the first node of the list.



- Useful when the nodes are circular in nature (such as the vertices of a polygon).
- To reduce the time complexity of insertion at the front, we can use the `last` pointer instead of `first` to access the list.

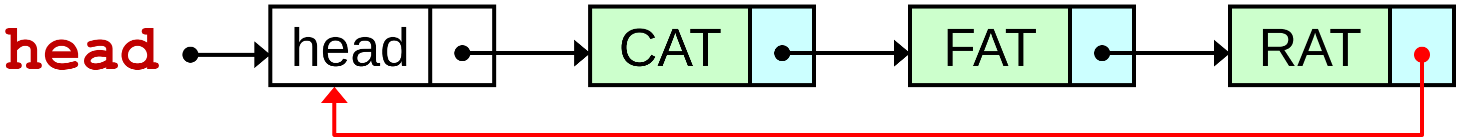


Head Nodes for Circular Lists

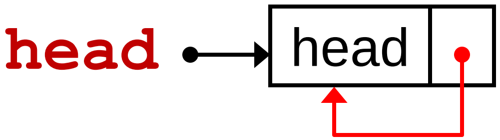
- When a circular list is empty:
  - This can be identified by `first==NULL` or `last==NULL`.
  - Handling empty circular lists requires special care in all operations.
- Alternative: Use a dummy head node that is never deleted.
  - Access the list via a `head` pointer to the head node.

Head Nodes for Circular Lists

A circular list with a head node:



An empty circular list with a head node:



Available-Space List

- Additions and deletions of nodes occur frequently for some lists in practice.
- The frequent use of **`new`** and **`delete`** is time consuming.
- Idea: Reuse deleted nodes instead of freeing them:
  - Inside the list class, keep a **static** chain **`av`**. (We can initialize it to **`NULL`** or pre-allocate a chunk of nodes.)
  - Deleted nodes are added to **`av`** (instead of using **`delete`**).
  - When a list object needs a new node and **`av`** is not empty, a node from **`av`** is used (instead of using **`new`**).
  - When a list object needs a new node and **`av`** is empty, use **`new`** to create a new node then.

Available-Space List

When inserting a node:

```
template <class T>
ChainNode<T>* CircularList<T>::GetNode()
{
  ChainNode<T>* x;
  if (av) {x = av;  av = av->link;}
  else x = new ChainNode<T>;
  return x;
}
```

When deleting a node:

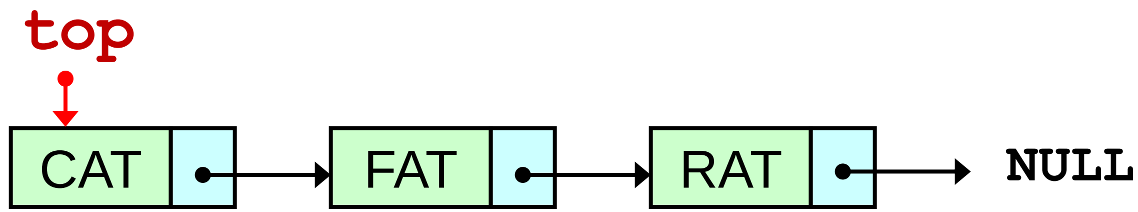
```
template <class T>
void CircularList<T>::RetNode(ChainNode<T>* &x)
{
  x->link = av;
  av = x;
  x = 0;
}
```

Q: Complexity of deleting a whole list now becomes $O(1)$. How is this done?

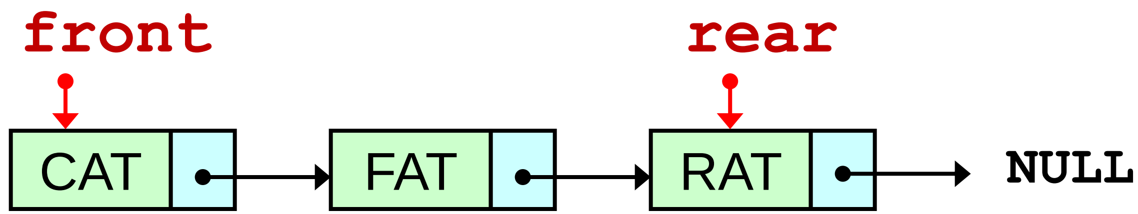
Linked Stacks and Queues

Using linked lists to represent stacks and queues. (Read textbook section 4.6 for implementations.)

Linked stack:



Linked queue:



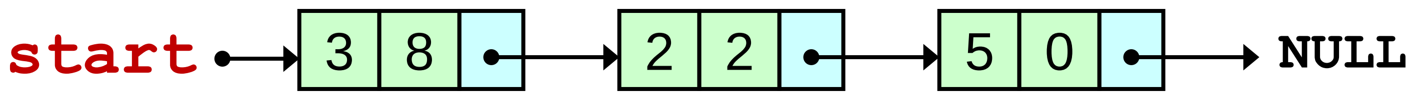
Revisiting Polynomials

Text: Use linked lists to represent polynomial.

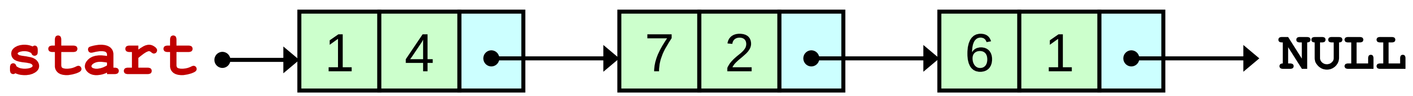
Node of a polynomial term:

coefficient	exponent	link

$$p_a(x) = 3x^8 + 2x^2 + 5$$



$$p_b(x) = x^4 + 7x^2 + 6x$$

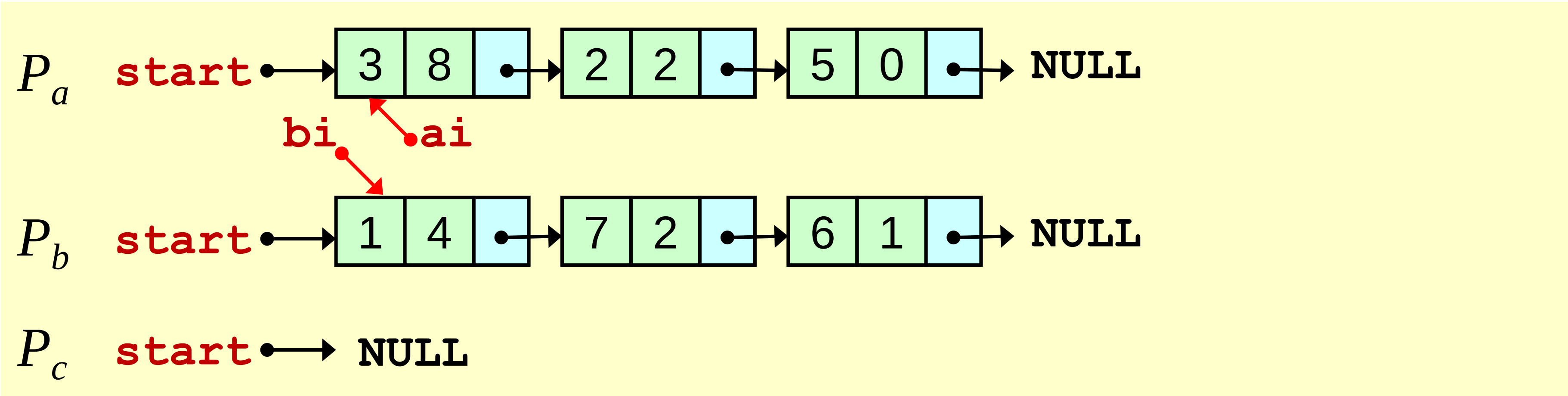


Polynomial Class

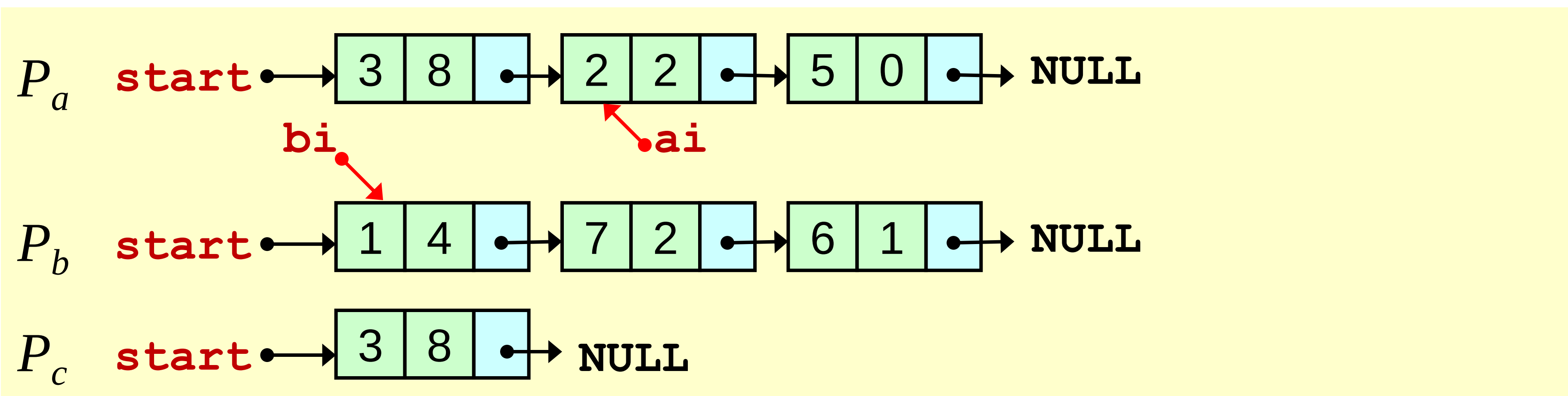
```
struct Term
{
  int coef;
  int exp;
  Term Set(int c,int e)
    {coef = c;  exp = e;  return *this;};
};


class Polynomial {
public:
  // declare public operations here
private:
  Chain<Term> poly;
};
```

Polynomial Addition

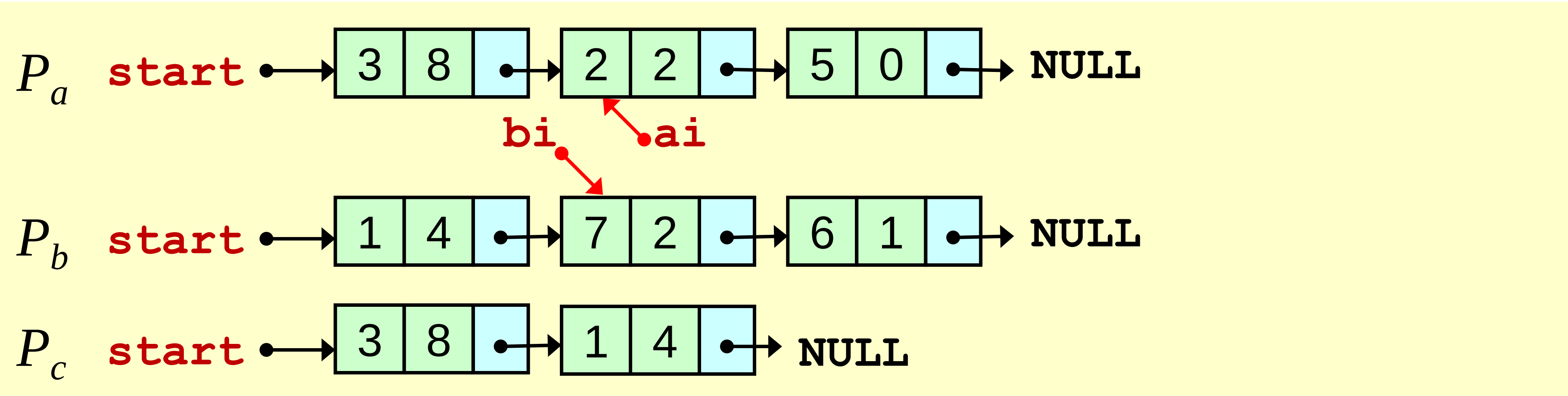


`ai->exp > bi->exp`

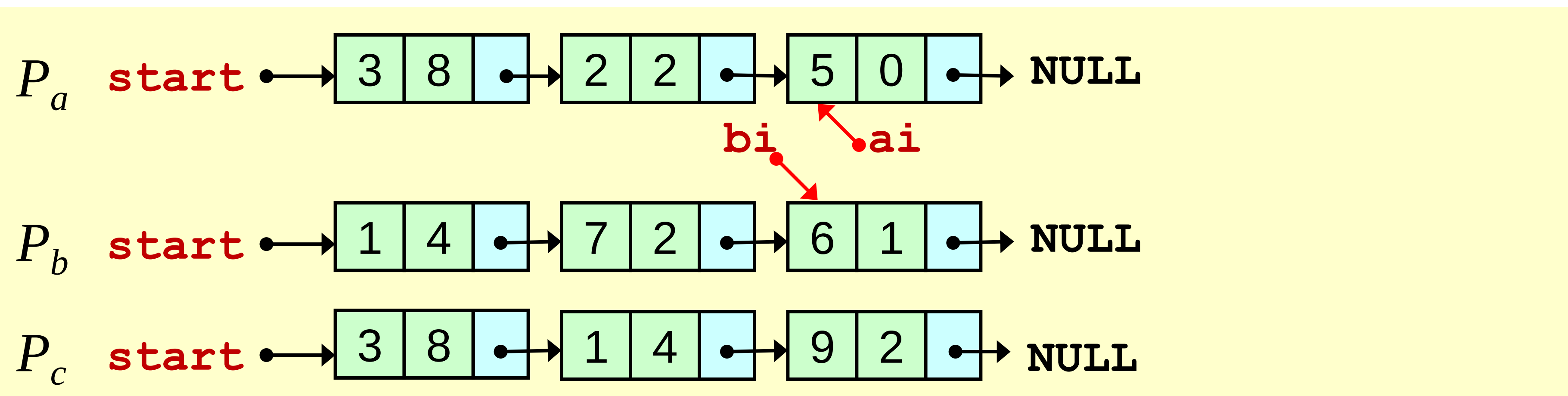


`ai->exp < bi->exp`

Polynomial Addition

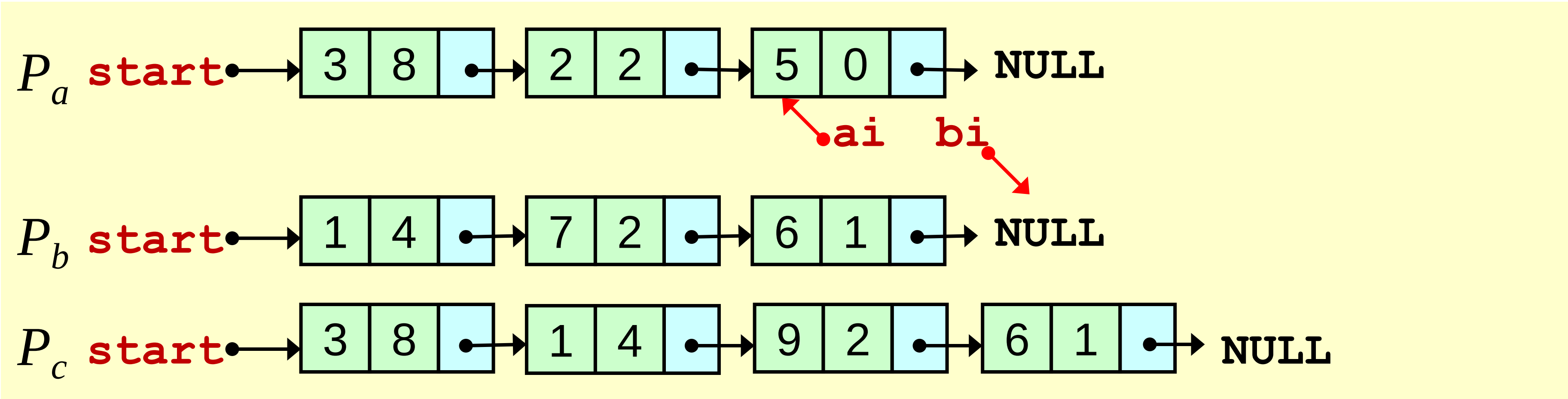


`ai->exp == bi->exp`

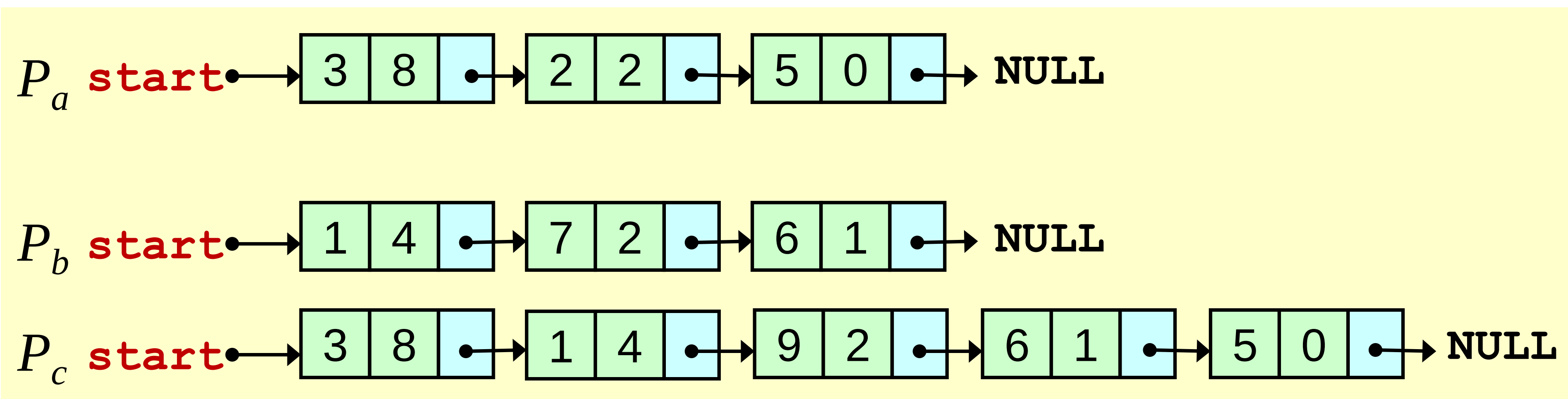


`ai->exp < bi->exp`

Polynomial Addition



bi is NULL



Sparse Matrix

- Our previous representation (chapter 2) of a sparse matrix is a linear representation ordered by row first.
  - Difficulty in finding or traversing elements by column.
  - Extra care in operations to keep the resulting elements in the correct order.
- Linked-list representation:
  - Each row or column is like a circular list with a header node.
  - Links in two directions (`down` and `right`) ➔ easy to access the next node on the same column or the same row.
  - Header node for each row and each column. (Actually, a header node is shared by a column and a row.)

Sparse Matrix Node

Header node:

<table>
<tr><td><code>head</code></td><td colspan="2"><code>next</code></td></tr>
<tr><td colspan="2"><code>down</code></td><td><code>right</code></td></tr>
</table>

Acts as a head node (in a circular list) **<u>both</u>** along a row and a column.

Element node:

<table>
<tr><td><code>head</code></td><td><code>row</code></td><td><code>col</code></td><td><code>value</code></td></tr>
<tr><td colspan="2"><code>down</code></td><td colspan="2"><code>right</code></td></tr>
</table>

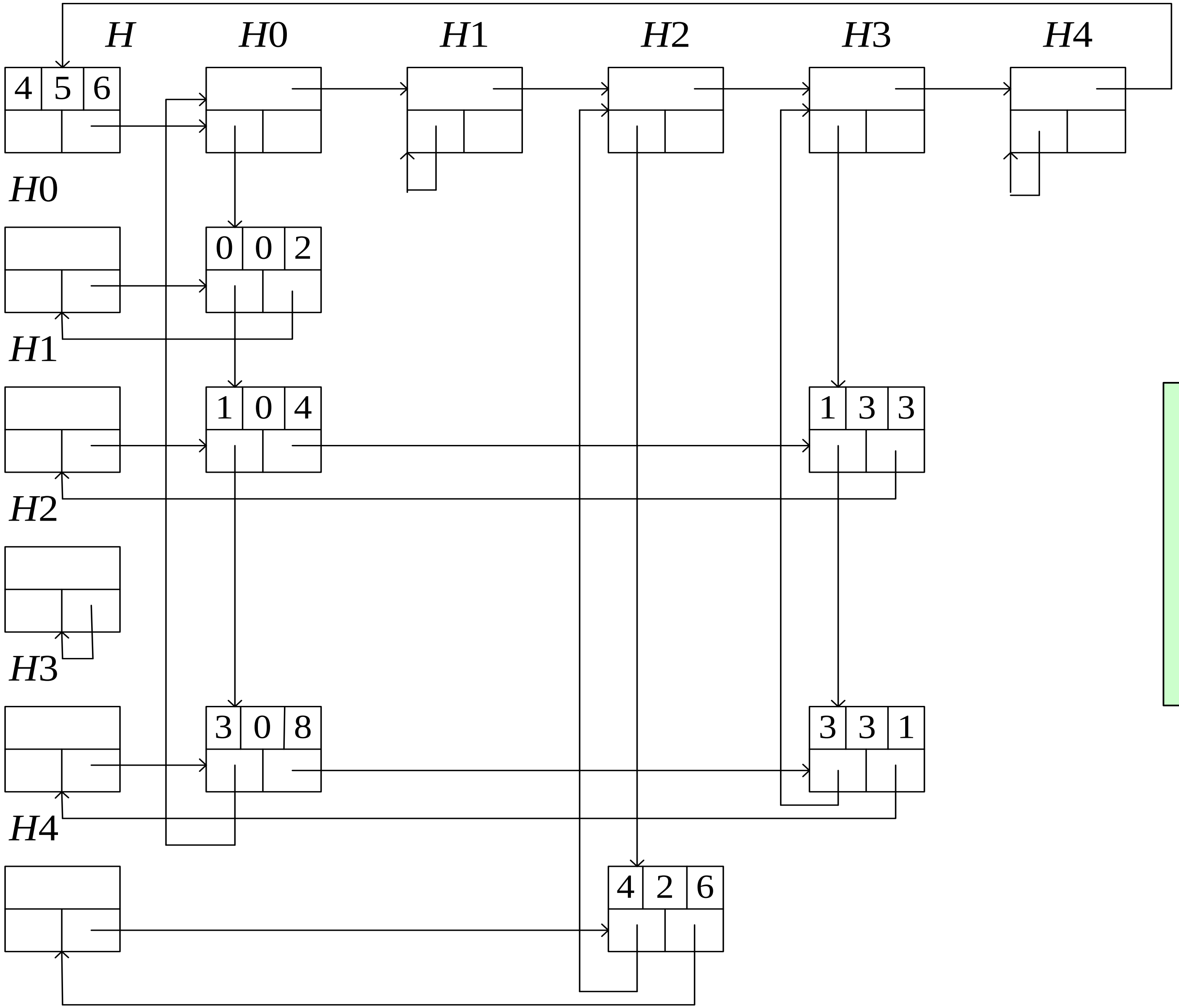
There is an overall head node for the whole matrix:

- Head node of the circular list of header nodes.
- Use the same members as an element node.
- **`row`**: number of rows
- **`col`**: number of columns

Sparse Matrix Node Class

```
struct Triple { int row, col, value; };
class Matrix;
class MatrixNode {
friend class Matrix;
friend istream& operator>>(istream&, Matrix&);
private:
  MatrixNode *down , *right;
  bool head; // flag of whether this is a header node
  union {
    MatrixNode *next; // this is a header node
    Triple triple; // this is a matrix element node
  };
  MatrixNode(bool, Triple*); // constructor
}
```

Sparse Matrix Example



$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

Equivalence Classes

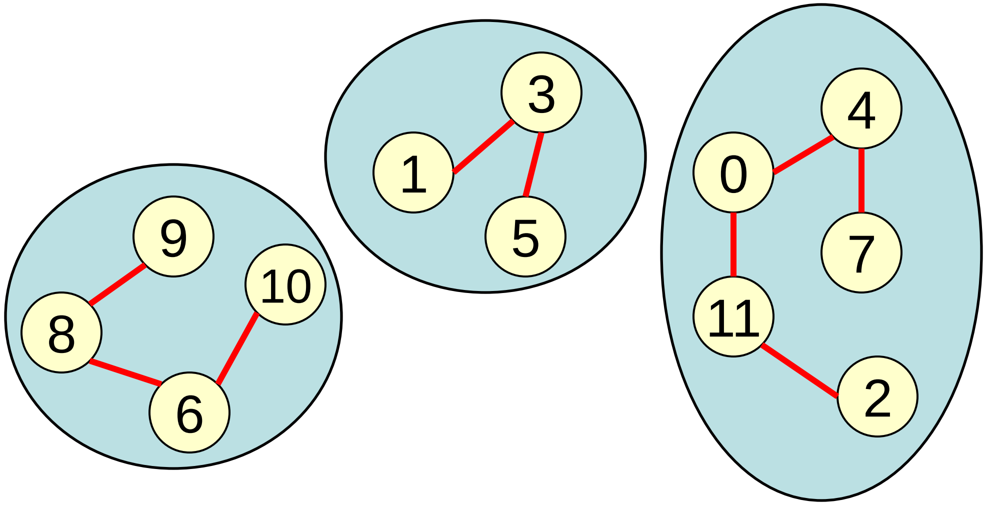
Equivalence relations:

- Represented by the symbol '$\equiv$'
- Required properties:
  - (Reflexive) $x \equiv x$
  - (Symmetric) $x \equiv y \Leftrightarrow y \equiv x$
  - (Transitive) $x \equiv y \text{ AND } y \equiv z \Rightarrow x \equiv z$
- Examples:
  - The "equality" relation
  - (Polygon vertices) "on the same polygon"

Equivalence Classes

An equivalence relation partitions a set into equivalence classes.

- Example set: {0, 1, 2, …, 10, 11}
- Known equivalent pairs:
  - 0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0
- Equivalent classes:
  - {0,2,4,7,11}, {1,3,5}, {6,8,9,10}

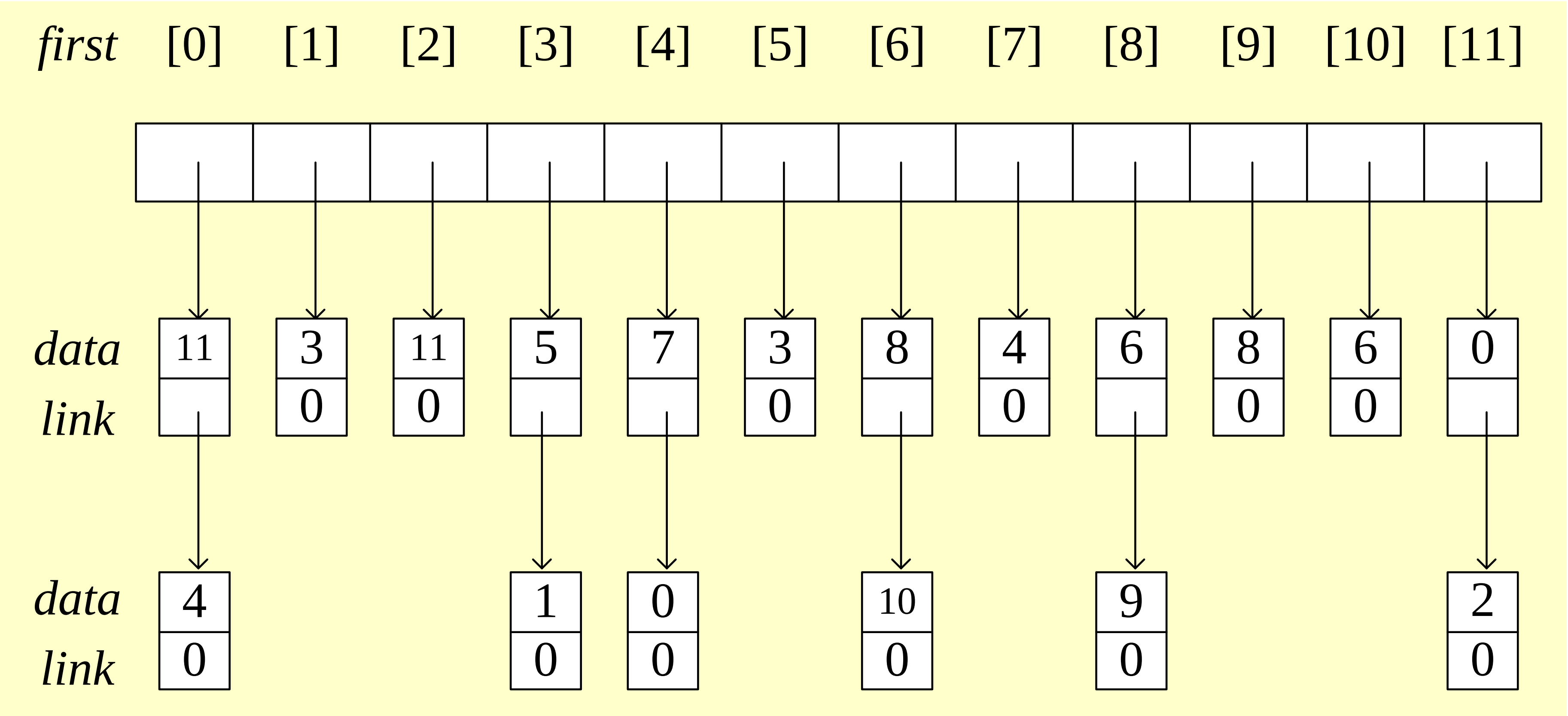


Building Equivalence Classes

- Inputs: Equivalent pairs
- Outputs: Equivalent classes
- Idea:
  - Read equivalent pairs one by one.
  - For each item, build a chain containing its directly linked (equivalent) items.
  - Output the equivalent classes, using a stack to handle transitivity.

Building Equivalence Classes

The chains of equivalence relations, after reading all the pairs (**Phase 1**).



`first` is an array of pointers, each pointing to the first item of a chain.

Building Equivalence Classes

Handling of transitivity (**Phase 2**).

An array of flags `out` (all initialized to false) is used to indicate whether an item is already in an equivalence class.

Procedure (minor modification from the code in textbook):

- For each item `i` not in an equivalence class
  - Start a new equivalence class
  - Initialize a stack containing only `i`
  - Repeat until the stack is empty
    - Pop the top item and assign it to this class
    - Push all the not-yet-assigned "neighbors" of the popped item to the stack

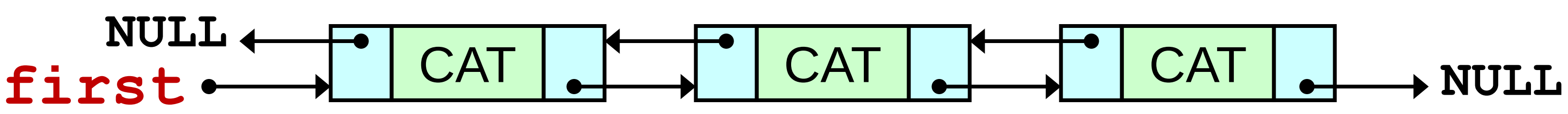
Building Equivalence Classes

Example of **Phase 2** (done in class):

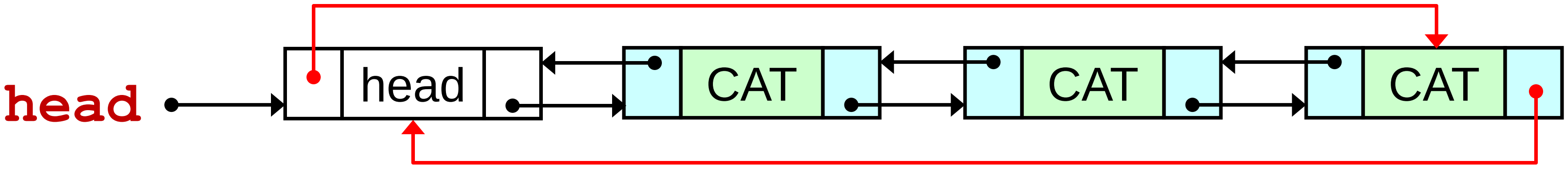
Note: This procedure is equivalent to the problem of finding a path in a maze, as "connectedness" among maze spaces is an equivalence relation. The array **`out`** plays the role of **`mark`** in the maze problem.

Doubly Linked Lists

- The main difficulty of singly linked lists:
  - Determining the proceeding (previous) node in the list, such as when deleting a node.
- This problem is solved if each node has a link to its previous node, in addition to a link to the next node.
- Linear doubly linked list:



- Circular doubly linked list (with a head node):



Operations of Doubly Linked Lists

- A node of a doubly linked list has two links: `left` and `right`. (See textbook for class definition).
- Insertion:

```
void DblList::Insert(DblListNode *p,
                     DblListNode *x)
{ // insert p to the right of x
  p->left = x;
  p->right = x->right;
  x->right->left = p;
  x->right = p;
}
```

- Deletion:

```
void DblList::Delete(DblListNode *x)
{
  if (x==head) { …; return; } // exception
  x->left->right = x->right;
  x->right->left = x->left;
  delete x;
}
```

Extra Reading Assignments

- From the textbook: Section 4.3.2, the part on C++ iterators.
- From the textbook: Section 4.6.