Chapter 4
Linked Lists

4.1 Singly Linked Lists and Chains

The representation of simple data structure using an array and a sequential mapping has the property:

- Successive nodes of the data object are stored at fixed distance apart.
- This makes it easy to access an arbitrary node in O(1).

Disadvantage of sequential mapping:

It makes insertion and deletion of arbitrary elements expensive.

For example:

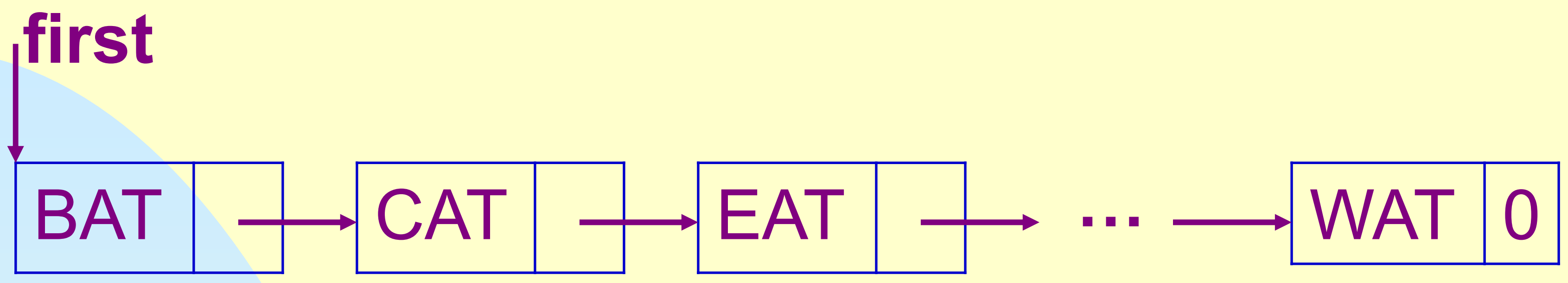
Insert "GAT" into or delete "LAT" from

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, TAT, VAT, WAT)

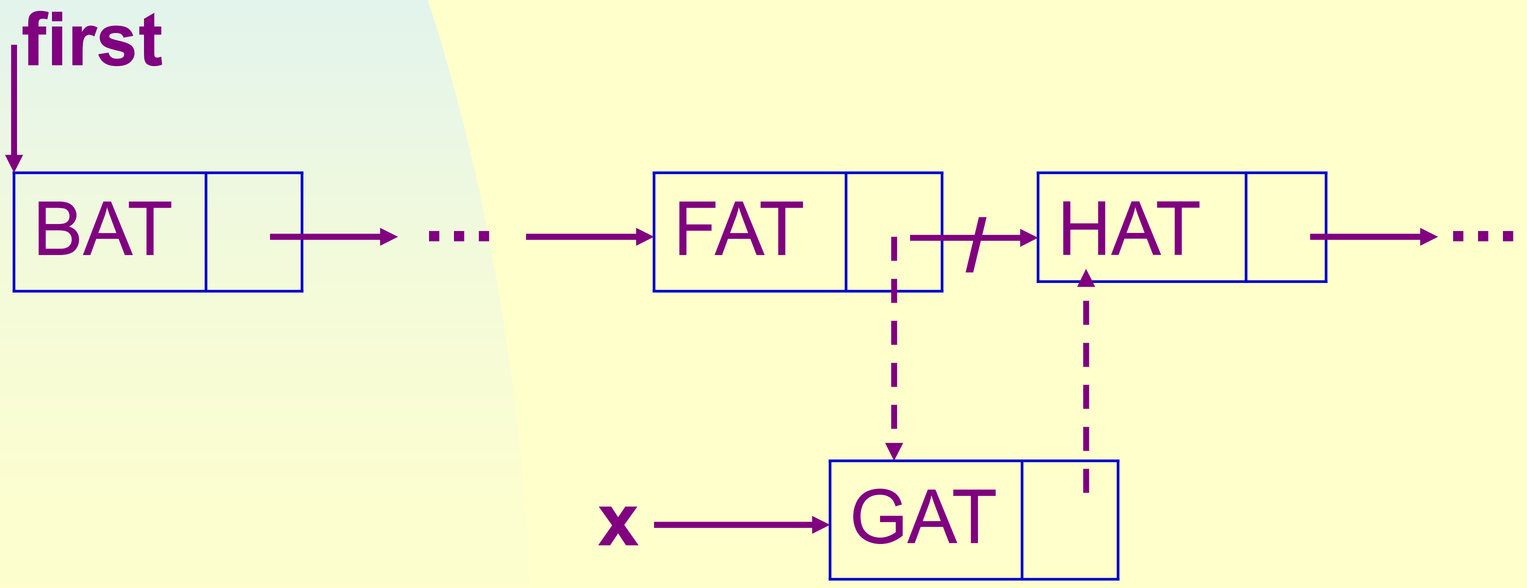
need **data movement**.

Solution---linked representation:

- Items of a list may be placed **anywhere** in the memory.
- Associated with each item is a point (link) to the next item.



In linked list, insertion (deletion) of arbitrary elements is much easier:

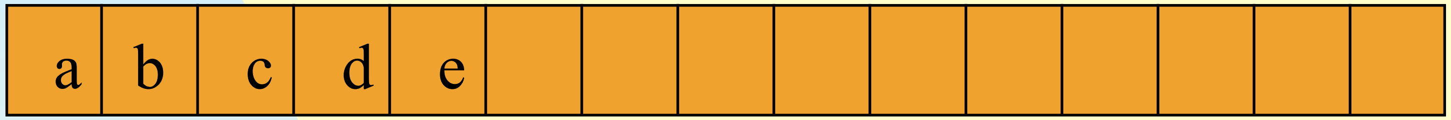


The above structures are called singly linked lists or chains in which each node has exactly one pointer field.

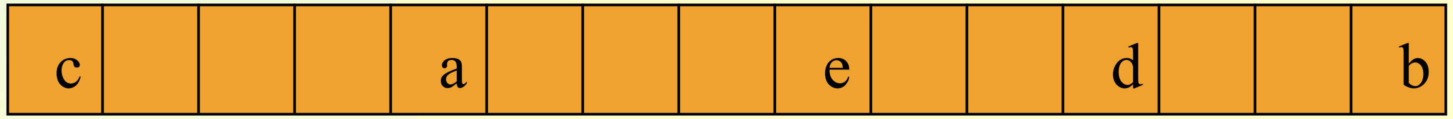
- list elements are stored, in memory, in an arbitrary order
- explicit information (called a link) is used to go from one element to the next

Memory Layout

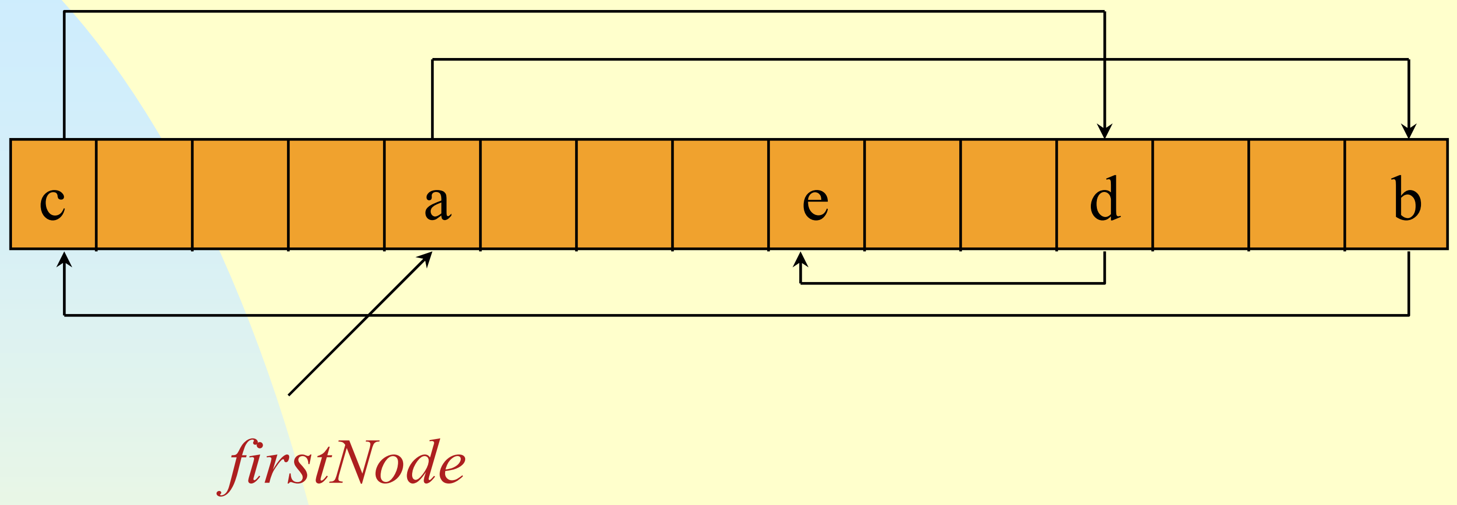
Layout of $L = (a, b, c, d, e)$ using an array representation.



A linked representation uses an arbitrary layout.



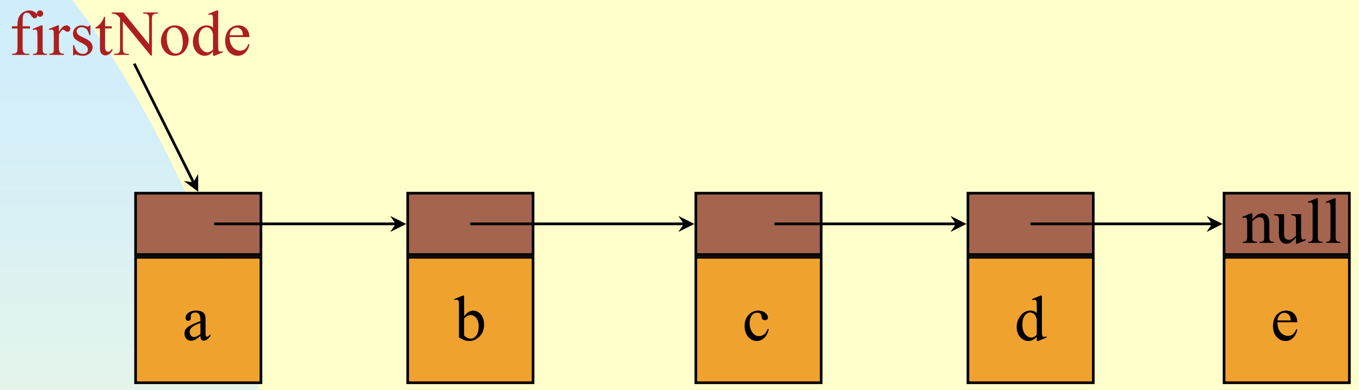
Linked Representation



pointer (or link) in *e* is *null*

use a variable *firstNode* to get to the first element *a*

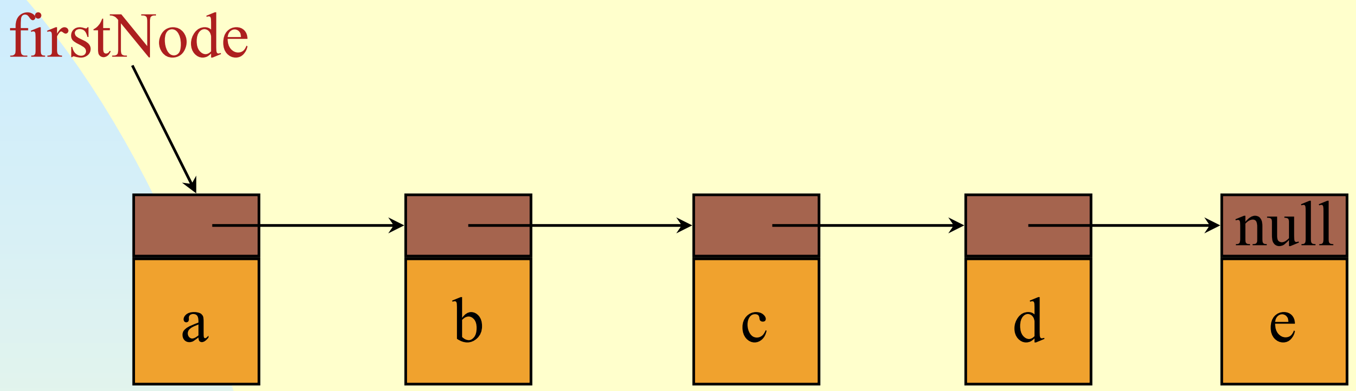
Normal Way To Draw A Linked List



🟫 link or pointer field of node

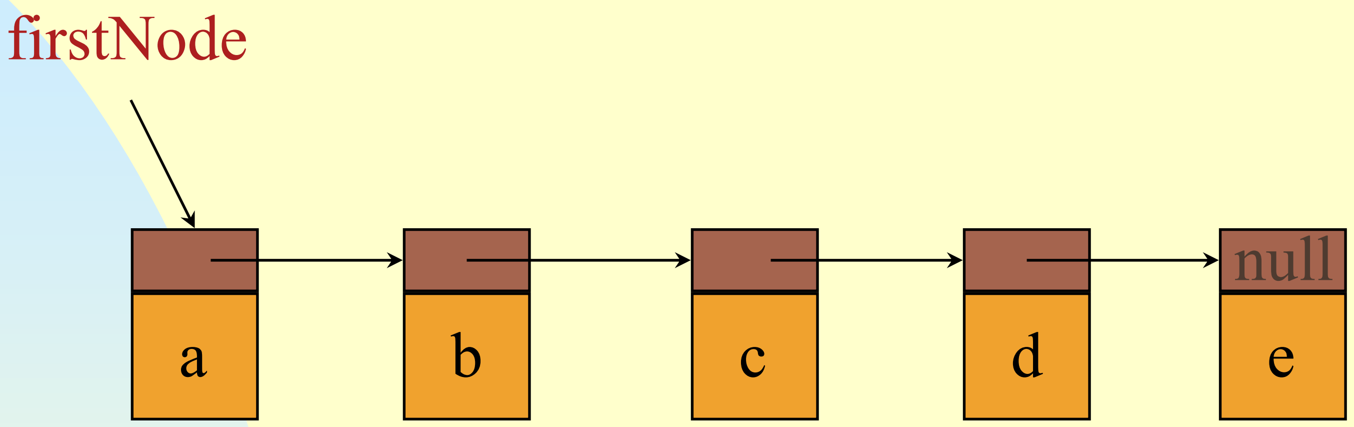
🟧 data field of node

Chain



- A chain is a linked list in which each node represents one element.
- There is a link or pointer from one element to the next.
- The last node has a *null* pointer.

get(0)

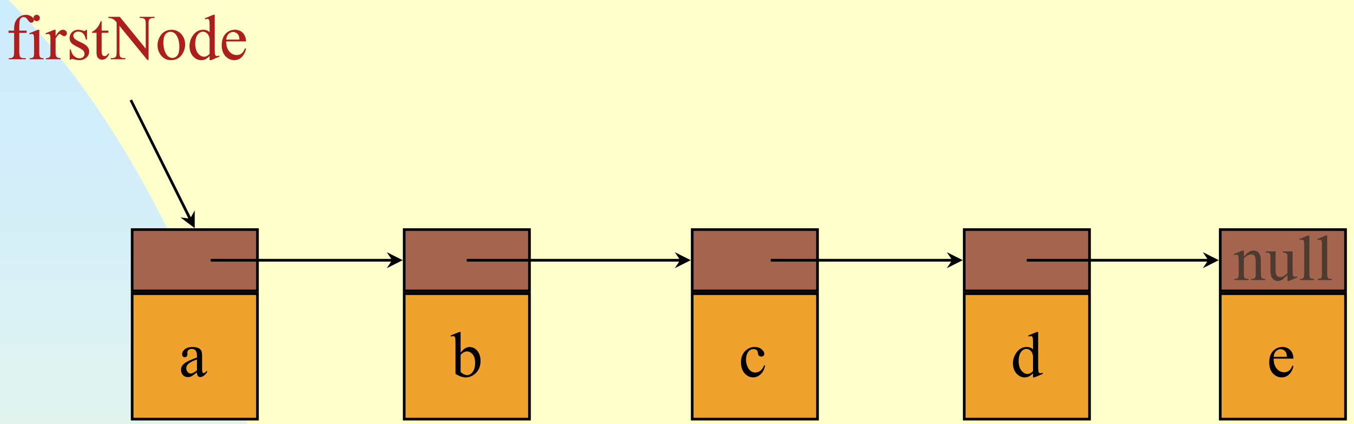


checkIndex(0)

desiredNode = *firstNode*; // gets you to first node

return *desiredNode* → *element*;

get(1)

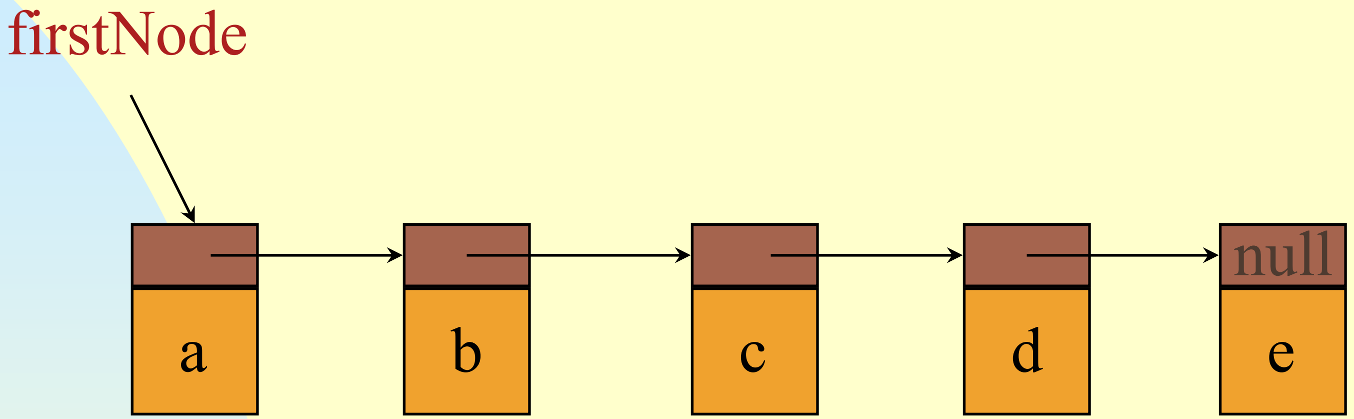


checkIndex(1)

desiredNode = *firstNode* → *next*; // gets you to second node

return *desiredNode* → *element*;

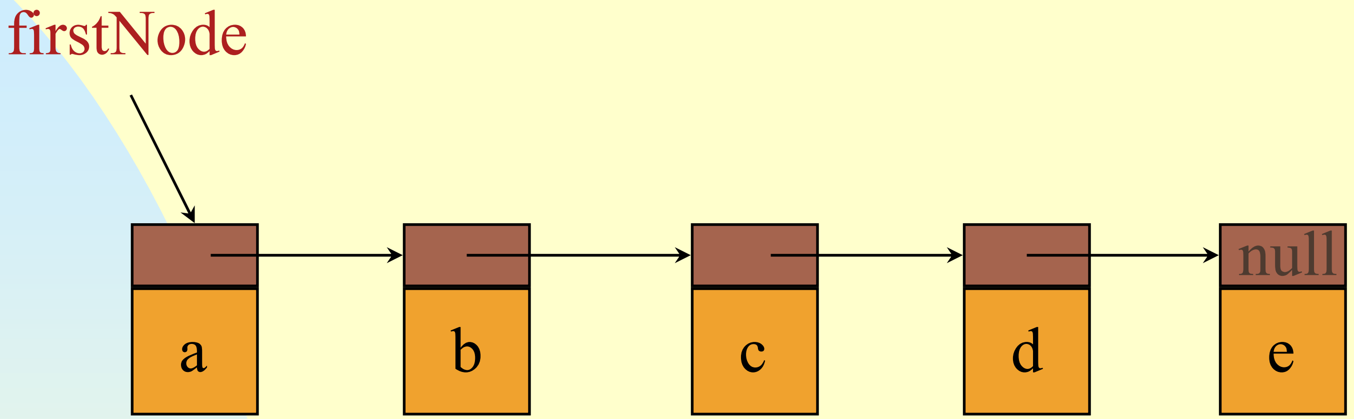
get(2)



checkIndex(2)

desiredNode = *firstNode* → *next* → *next*; // gets you to third node
return *desiredNode* → *element*;

get(5)

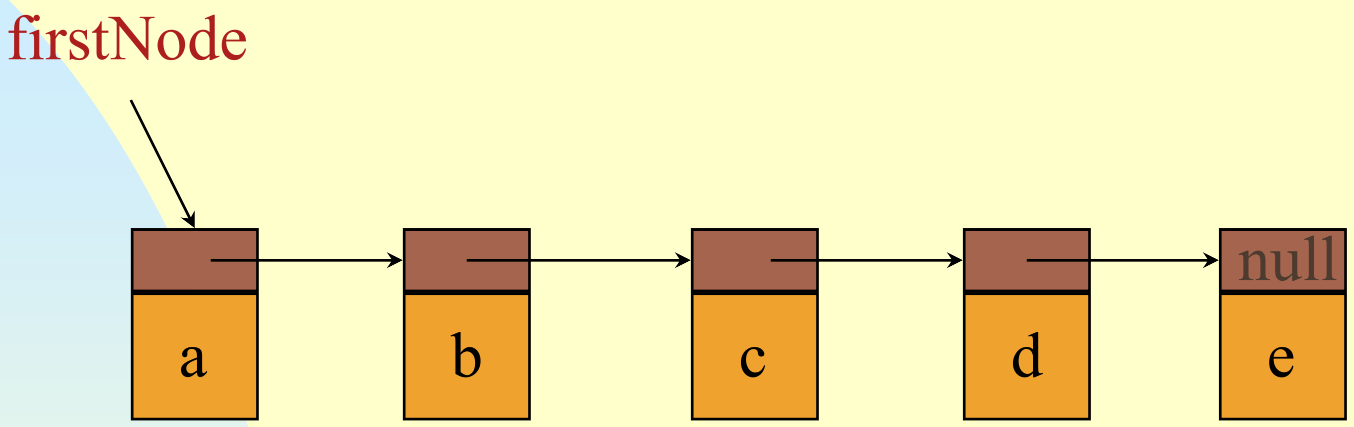


checkIndex(5)   // throws exception

desiredNode = firstNode → next → next → next → next → next;
  // desiredNode = null

return *desiredNode → element*; *// null→element*

NullPointerException

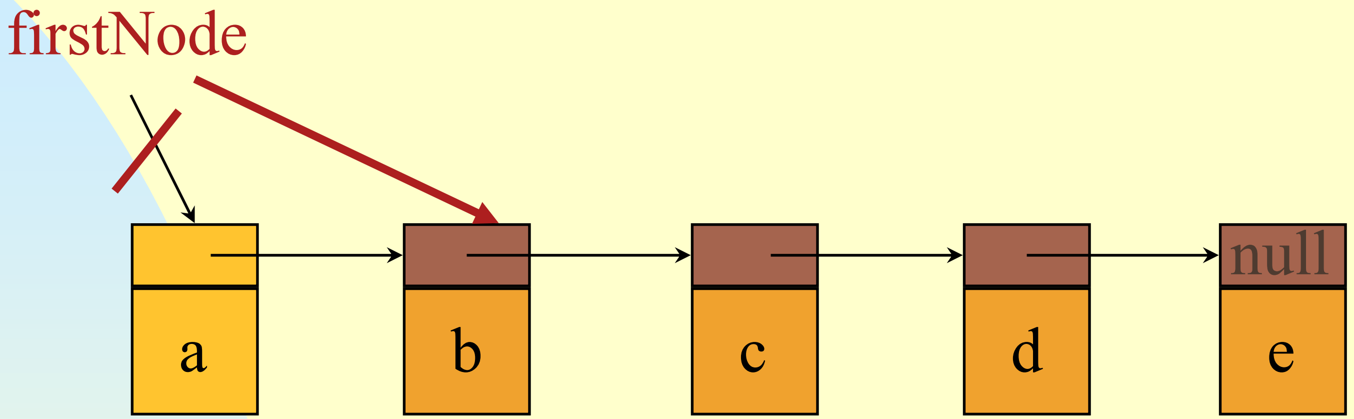


desiredNode = *firstNode* → *next* → *next* → *next* → *next* → *next*;

// gets the computer mad

// you get a NullPointerException

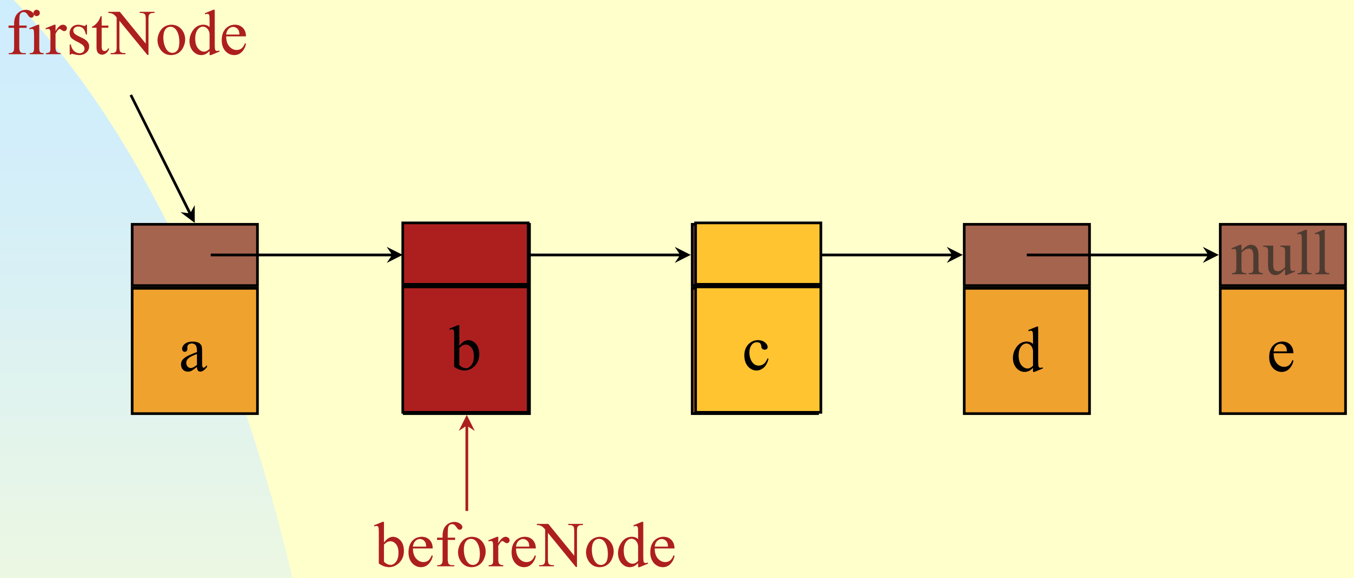
Remove An Element



remove(0)

firstNode = firstNode → next;

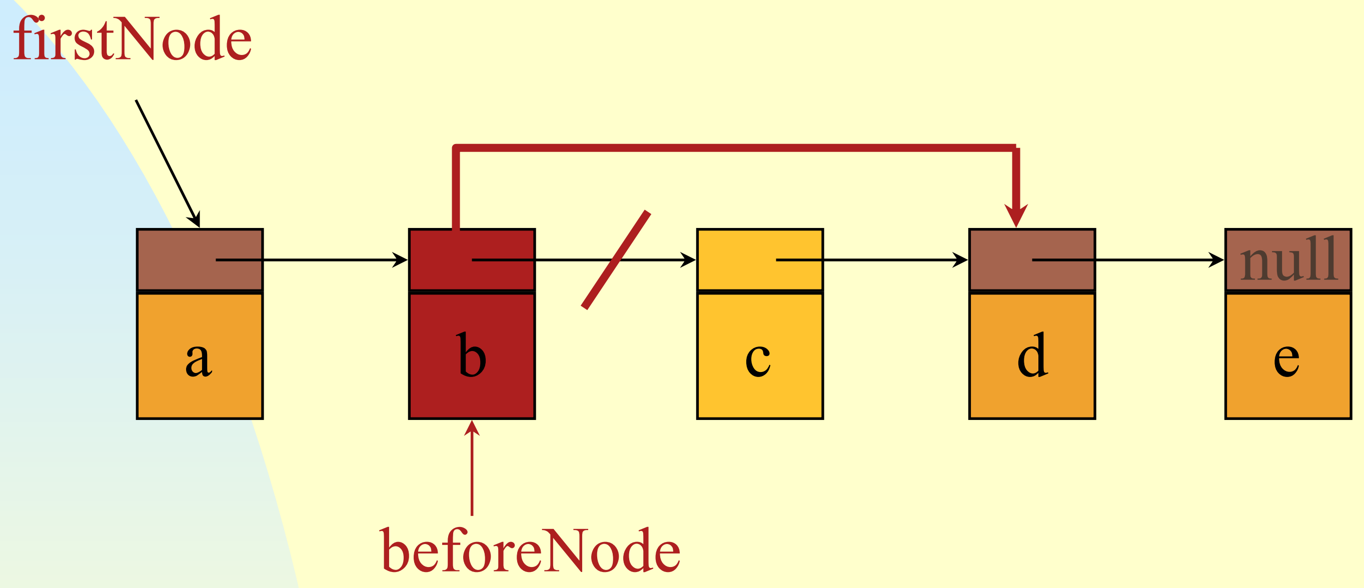
remove(2)



first get to node just before node to be removed

beforeNode = firstNode → next;

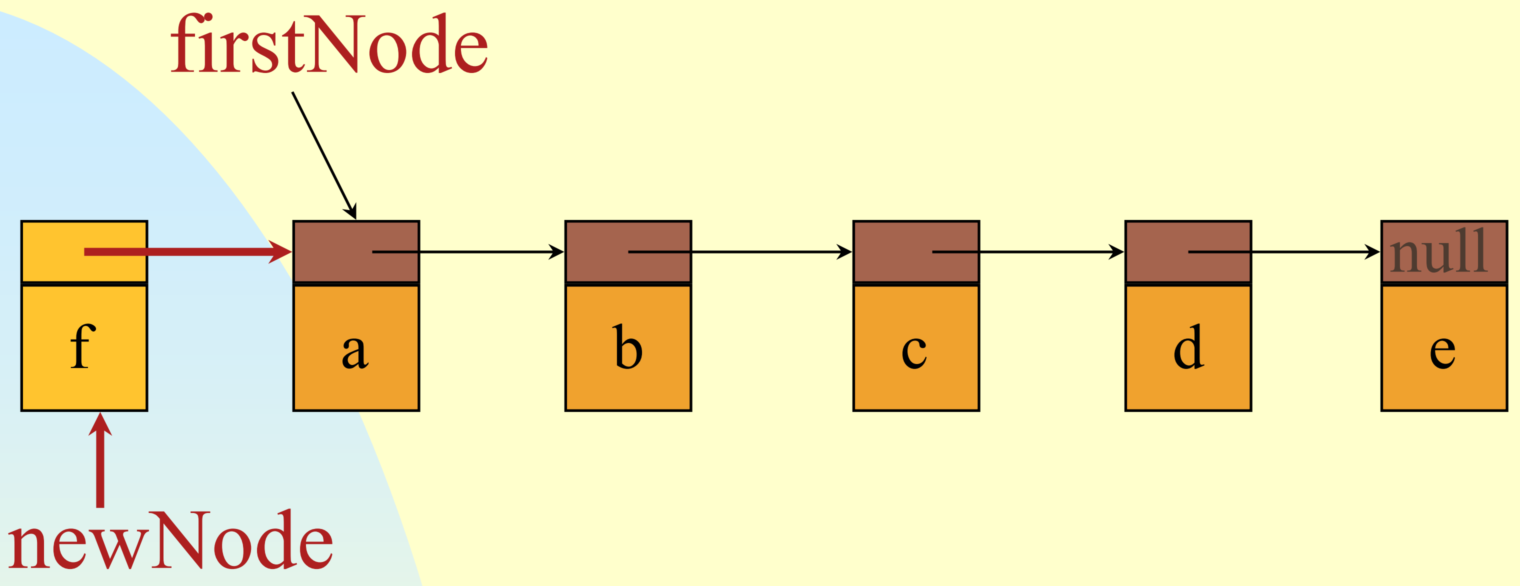
remove(2)



now change pointer in *beforeNode*

beforeNode → *next* = *beforeNode* → *next* → *next*;

add(0,’f’)

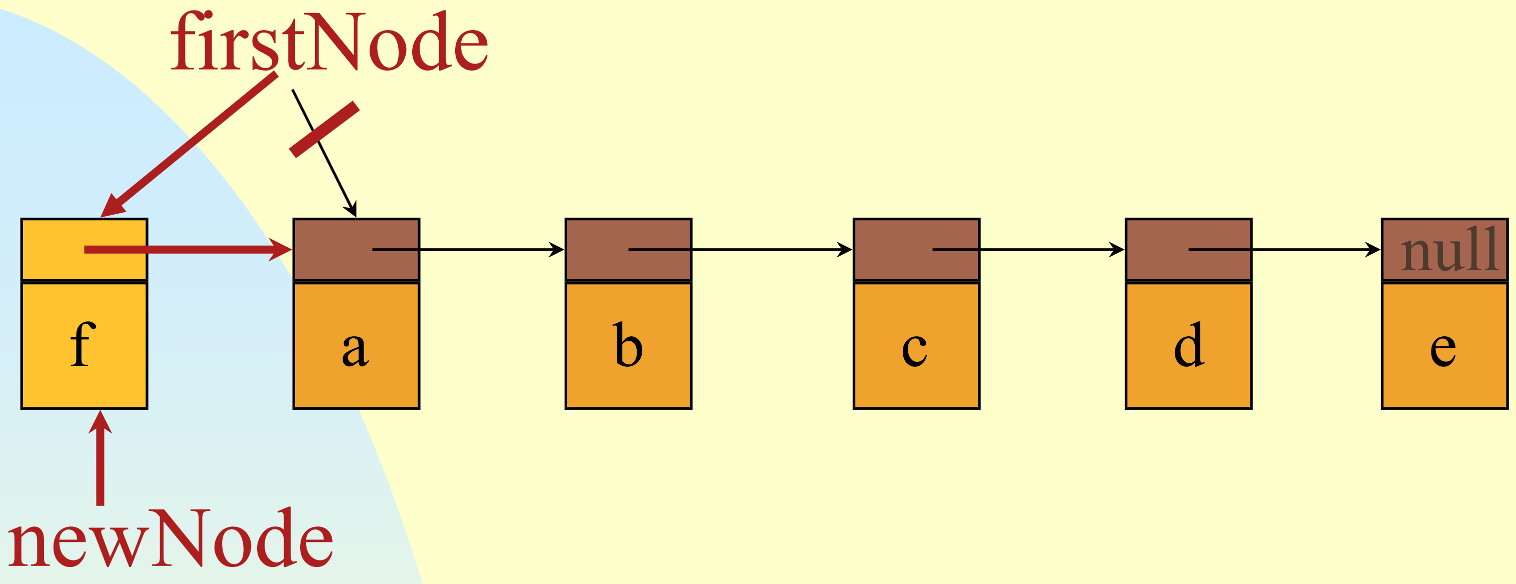


Step 1: get a node, set its data and link fields

ChainNode newNode =

new *ChainNode*(**new** *Character*(‘f’), *firstNode*);

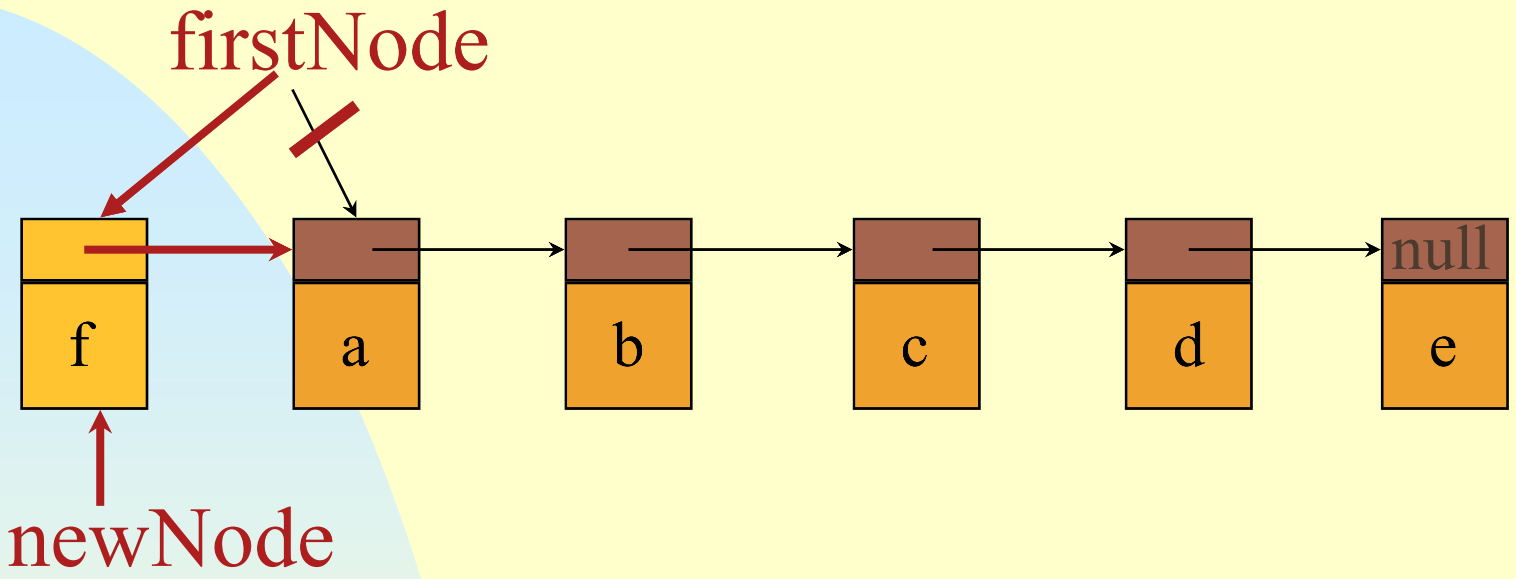
add(0,’f’)



Step 2: update *firstNode*

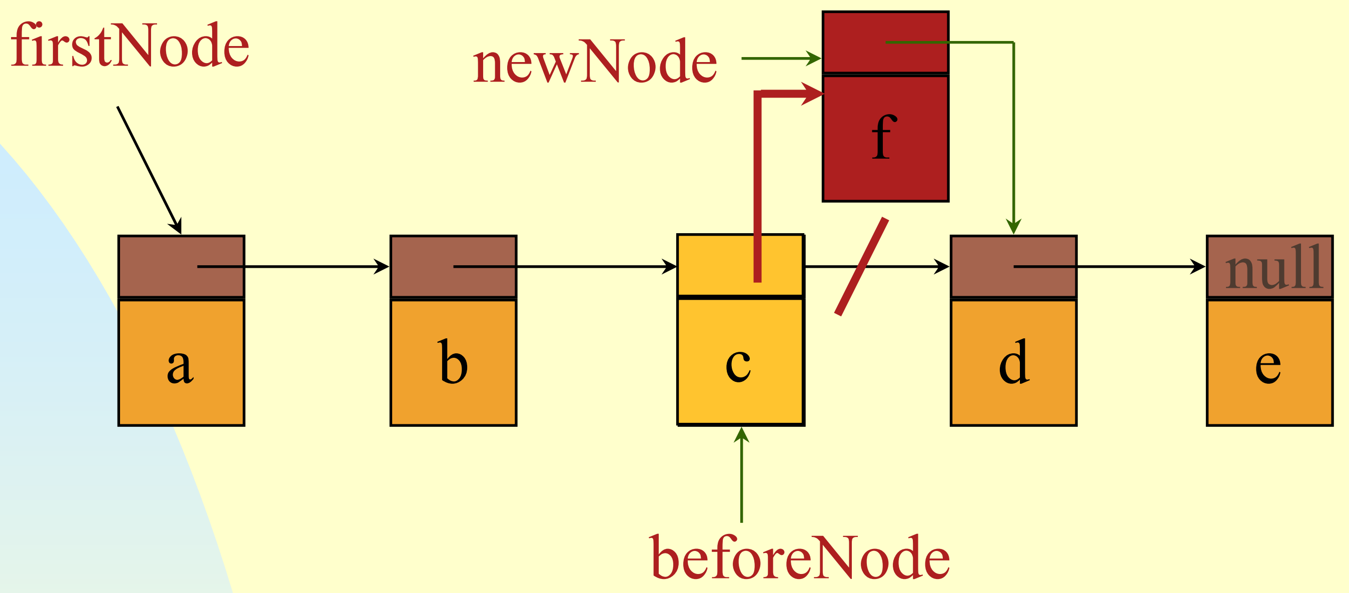
firstNode = newNode;

One-Step add(0,’f’)



firstNode = **new** *ChainNode*(
new *Character*(‘f’), *firstNode*);

add(3,’f’)



- first find node whose index is 2
- next create a node and set its data and link fields

ChainNode newNode = **new** *ChainNode*(**new** *Character*(‘f’),

beforeNode → *next*);

- finally link beforeNode to newNode

beforeNode → *next* = *newNode*;

4.2 Representing Chains in C++

Assume a chain node is defined as:

```
class ChainNode {
private:
    int data;
    ChainNode *link;
};
```

ChainNode **f*;

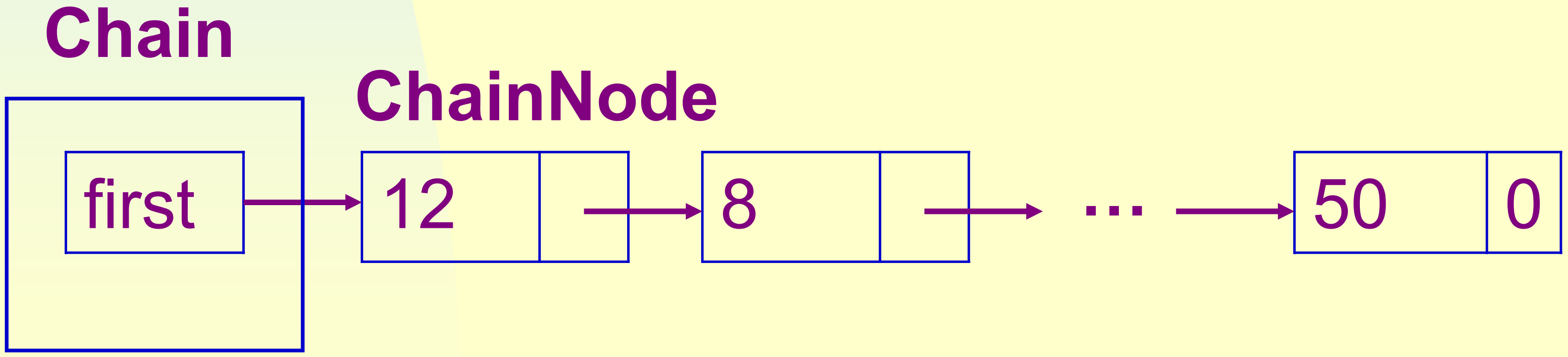
f→data

will cause a compiler error because a private data member cannot be accessed from outside of the object.

Definition: a data object of Type *A* HAS-A data object of Type *B* if *A* conceptually contains *B* or *B* is a part of *A*.

A composite of two classes: *ChainNode* and *Chain*.

***Chain* HAS-A *ChainNode*.**

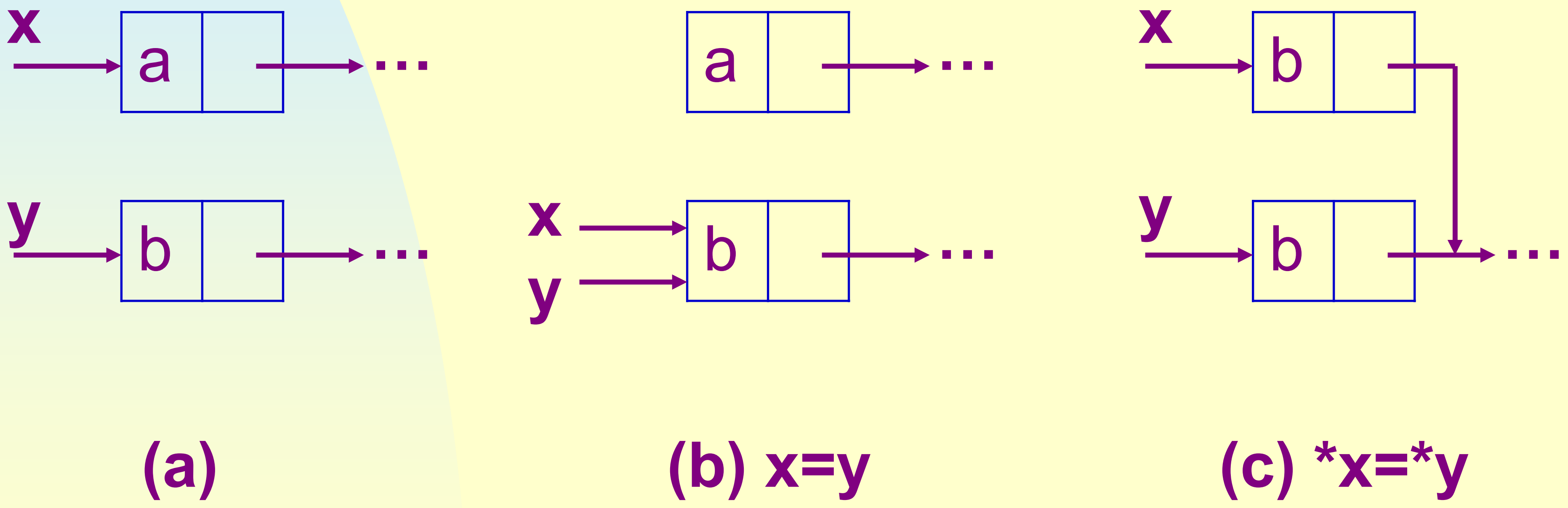


```
class Chain;  // forward declaration
class ChainNode {
friend class Chain; // to make functions of Chain be able to
                    // access private data members of ChainNode
Public:
   ChainNode(int element = 0, ChainNode *next = 0)
      {data = element; link = next;}
private:
   int data;
   ChainNode *link;
};
class Chain {
public:
   // Chain manipulation operations
…
private:
   ChainNode *first;
};
```

```
class Chain {
public:
    // Chain manipulation operations
…
private:
    class ChainNode {   //nested class
    public:
          int data;
          ChainNode *link;
     };
    ChainNode *first;
};
```

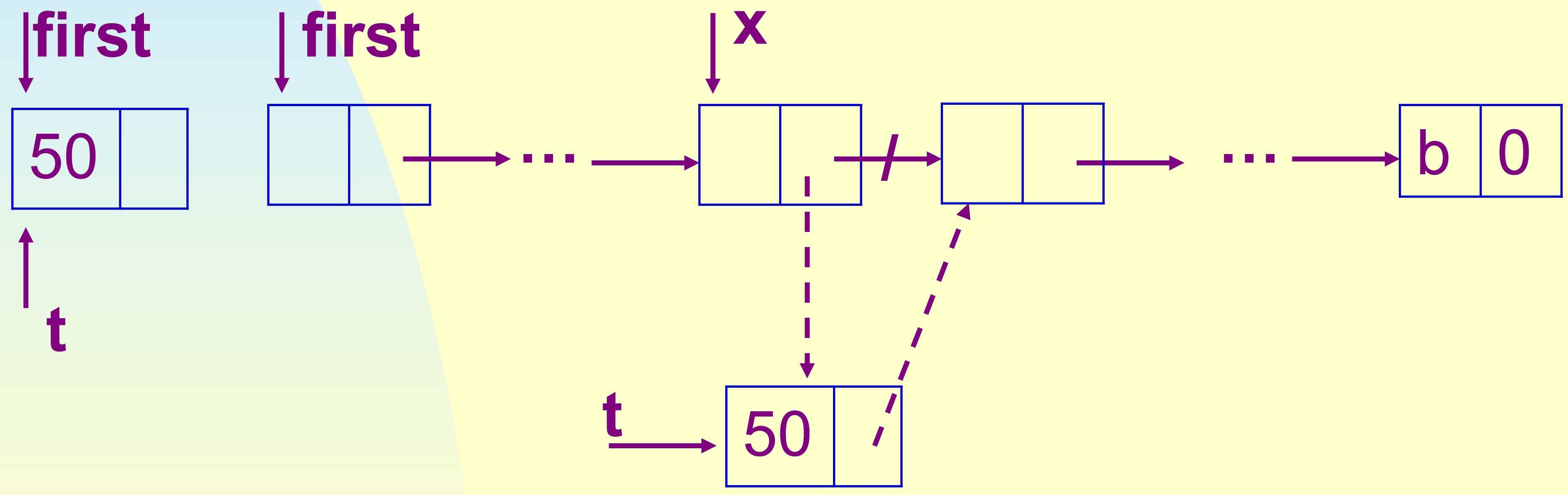
Null pointer constant 0 is used to indicate no node.

Pointer manipulation in C++:



Chain manipulation:

Insert a node with data field 50, following the node x.



(a) first=0

(b) first !=0

```
void Chain::Insert50 (ChainNode *x)
{
  if (first)
     // insert after x
    x → link = new ChainNode(50, x → link);
  else
     // insert into empty chain
     first = new ChainNode(50);
}
```

4.3 The Template Class Chain

We shall enhance the chain class of the previous section to make it more reusable.

4.3.1 Implementing Chains with Templates

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

template <class T>
class ChainNode {
friend class Chain<T>;
public:
   ChainNode(T element, ChainNode *next = 0)
      { data = element; link = next;}
private:
   T data;
   ChainNode *link;
};
```

```
template <class T>
class Chain {
public:
   Chain() { first=0;}; // constructor initializing first to 0
   // Chain manipulation operations
…
private:
   ChainNode<T> *first;
};
```

A empty chain of integers *intchain* would be defined as:

```
   Chain<int> intchain ;
```

4.3.2 Chain Iterators

A container class is a class that represents a data structure that contains or stores a number of data objects.

An iterator is an object that is used to access the elements of a container class one by one.

Why we need an iterator?

Consider the following operations that might be performed on a container class *C*, all of whose elements are integers:

(1) Output all integers in *C*.

(2) Obtain the sum, maximum, minimum, mean, median of all integers in *C*.

(3) Obtain the integer *x* from *C* such that *f*(*x*) is maximum.

Examine all the elements of the container class

These operations have to be implemented as member functions of *C* to access its private data members.

Consider the container class Chain<*T*>, there are, however, some drawbacks to this:

(1) Operations that make sense for one instantiation of *T* may not for another instantiation.

(2) The number of operations of Chain<*T*> can become too large.

(3) Even if it is acceptable to add member functions, the user would have to learn how to sequence through the container class.

These arguments suggest that container classes be equipped with iterators that provide systematic access the elements of the object.

User can employ these iterators to implement their own functions depending upon the particular application.

Typically, an iterator is implemented as a nested class of the container class.

A forward Iterator for Chain

A forward Iterator class for *Chain* may be implemented as in the next slides, and it is required that *ChainIterator* be a public nested member class of *Chain*.

```
class ChainIterator {
public:
   // typedefs required by C++ omitted

   // constructor
   ChainIterator(ChainNode<T>* startNode = 0)
      { current = startNode; }

   // dereferencing operators
   T& operator *() const { return current→data;}
   T* operator →() const { return &current→data;}
```

```
// increment
ChainIterator& operator ++() // preincrement
  {
     current = current → link;
     return *this;
  }
ChainIterator& operator ++(int) // postincrement
  {
     ChainIterator old = *this;
     current = current → link;
     return old;
  }
```

```
    // equality testing
    bool operator != (const ChainIterator right) const
        { return current != right.current; }
    bool operator == (const ChainIterator right) const
        { return current == right.current; }

    private:
        ChainNode<T>* current;
};
```

Additionally, we add the following public member functions to Chain:

```
ChainIterator begin() {return ChainIterator(first);}
ChainIterator end() {return ChainIterator(0);}
```

We may initialize an iterator object *yi* to the start of a chain of integers *y* using the statement:

```
Chain<int>::ChainIterator yi = y.begin();
```

And we may sum the elements in *y* using the statement:

```
sum = accumulate(y.begin(), y.end(), 0);
// note sum does not require access to private members
```

Write an algorithm to print all data of a Chain.

Write an algorithm to print all data of a Chain using the iterator mechanism.

4.3.3 Chain Operations

Operations provided in a reusable class should be enough but not too many.

Normally, include: constructor, destructor, operator=, operator==, operator>>, operator<<, etc.

A chain class should provide functions to insert and delete elements.

Another useful function is reverse that does an "in-place" reversal of the elements in a chain.

To insert efficiently at the end of a chain, we add a private member *last* to *Chain<T>*, which points to the last node in the chain.

InsertBack

```
template <class T>
void Chain<T>::InsertBack(const T& e)
{
   if (first) { // nonempty chain
      last → link = new ChainNode<T>(e);
      last = last → link;
    }
   else first = last = new ChainNode<T>(e);
}
```

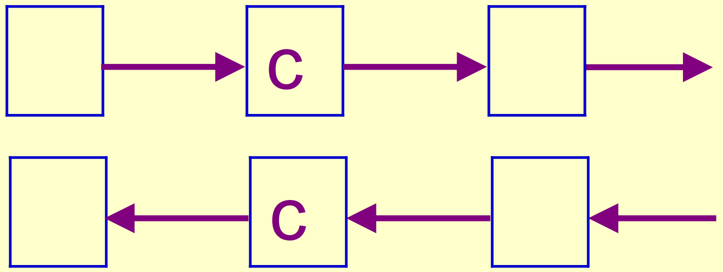
The complexity: O(1).

Concatenate

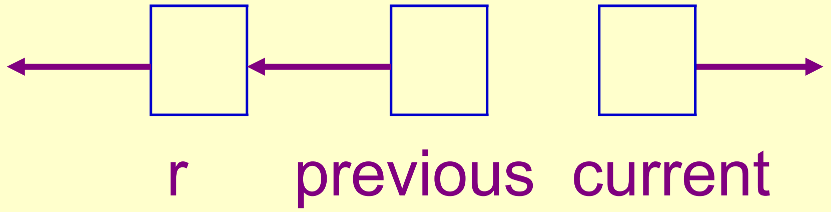
```
template <class T>
void Chain<T>::Concatenate(Chain<T>& b)
{ // b is concateneted to the end of *this
    if (first)
    { last→link = b.first; last = b.last;}
    else
    { first = b.first; last = b.last; }
    b.first = b.last = 0;
}
```

The complexity: O(1).

Reverse



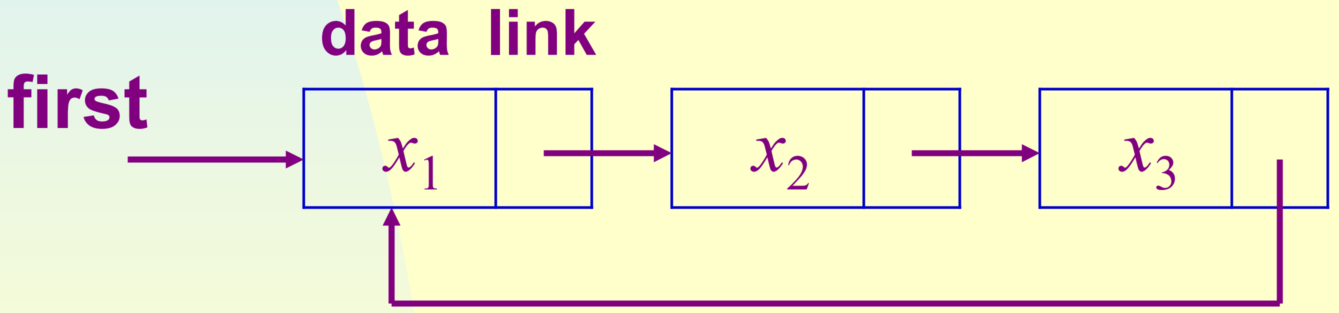
```
template <class T>
void Chain<T>::Reverse()
{ // make ($a_1$, …, $a_n$) becomes ($a_n$, …, $a_1$).
   ChainNode<T> *current = first, *previous = 0;
   while (current) {
      ChainNode<T> *r = previous; // r trails previous
      previous = current;
      current = current→link;
      previous→link = r;
   }
   first = previous;
}
```



For a chain with $m \geq 1$ nodes, the computing time of Reverse is O(m).

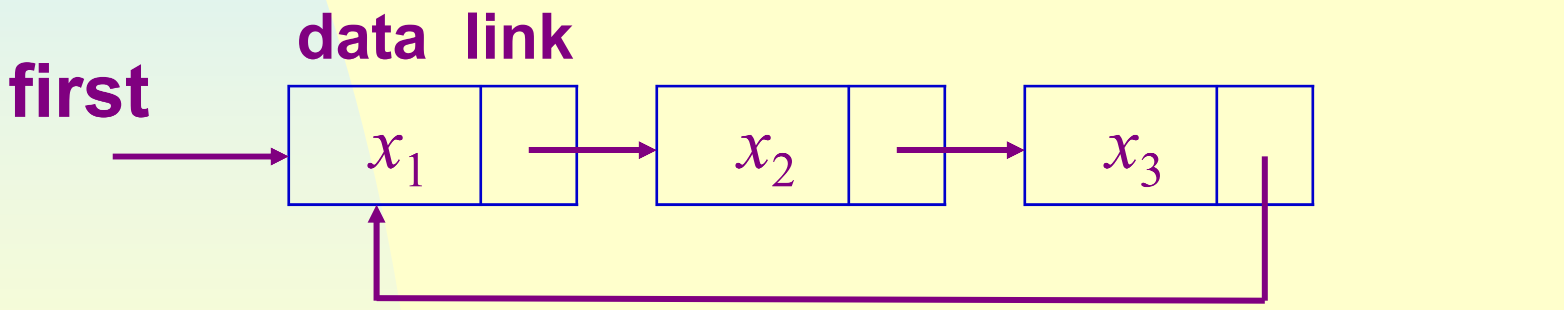
4.4 Circular Lists

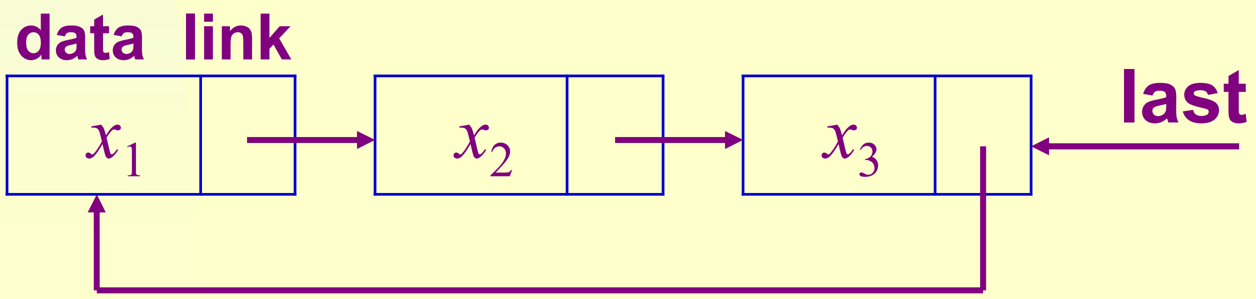
A circular list can be obtained by modifying a chain so that the link field of the last node points to the first node.



Consider inserting a new node at the front, we need to change the link field of the node containing x_3.

It is more convenient if the access pointer points to the **last** rather than the first.





Now we can insert at the front in O(1):

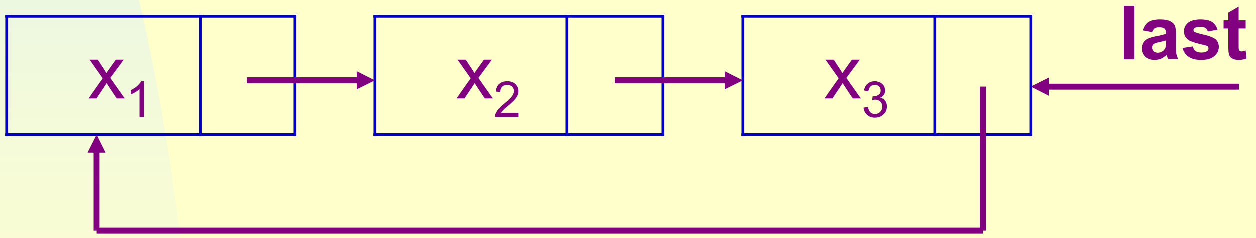
```
template <class T>
void CircularList<T>::InsertFront(const T& e)
{ // insert the element e at the "front" of the circular list *this,
  // where last points to the last node in the list.
    ChainNode<T>* newNode = new ChainNode<T>(e);
    if (last) { // nonempty list
        newNode→link =
                                    last→link;
        last→link =
                                   newNode;
     }
    else { last = newNode; newNode→link = newNode;}
}
```

To insert at the back,

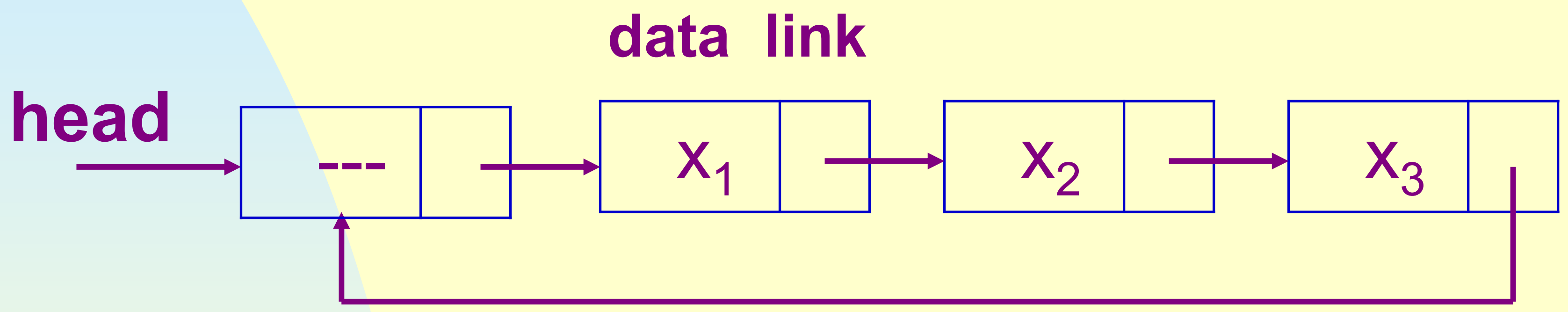
we only need to add the statement

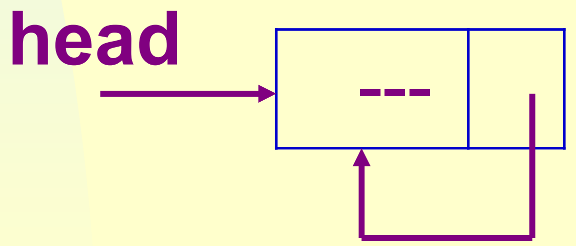
***last* = *newNode*;**

to the if clause of *InsertFront*, the complexity is still O(1).



To avoid handling empty list as a special case, a dummy (header) node is introduced.





empty circular list

4.5 Available Space lists

- **The time of destructors for chains and circular lists is linear in the length of the chain or list.**
- **It may be reduced to O(1) if we maintain our own chain of free nodes.**
- **The available space list is pointed by *av*.**
- ***av* be a static class member of *CircularList<T>* of type *ChainNode<T>* *, initially, *av* = 0.**
- **Only when the *av* list is empty do we need use new.**

We shall now use *CircularList<T>::GetNode* instead of using new:

```
template <class T>
ChainNode<T>* CircularList<T>::GetNode( )
{ //Provide a node for use.
    ChainNode<T> * x;
    if (av) { x = av; av = av→link;}
    else x = new ChainNode<T>;
    return x;
}
```

And we use *CircularList<T>::RetNode* instead of using delete:

```
template <class T>
 void CircularList<T>::RetNode(ChainNode<T>*& x)
{ // Free the node pointed to by x.
    x→link = av;
    av = x;
    x = 0;
}
```

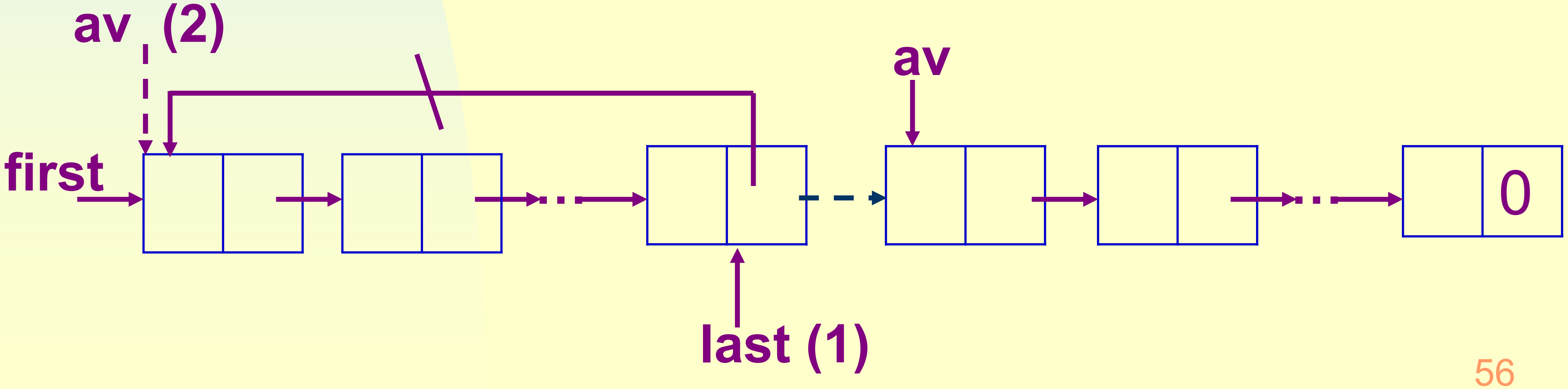
A circular list may be destructed in O(1):

```
template <class T>
void CircularList<T>::~CircularList()
{ // Delete the circular list.
    if (last) {
        ChainNode <T> * first = last→link;
        last→link = av;   // (1)
        av = first; // (2)
        last = 0;
    }
}
```

As shown in the next slide:

A circular list may be deleted in O(1):

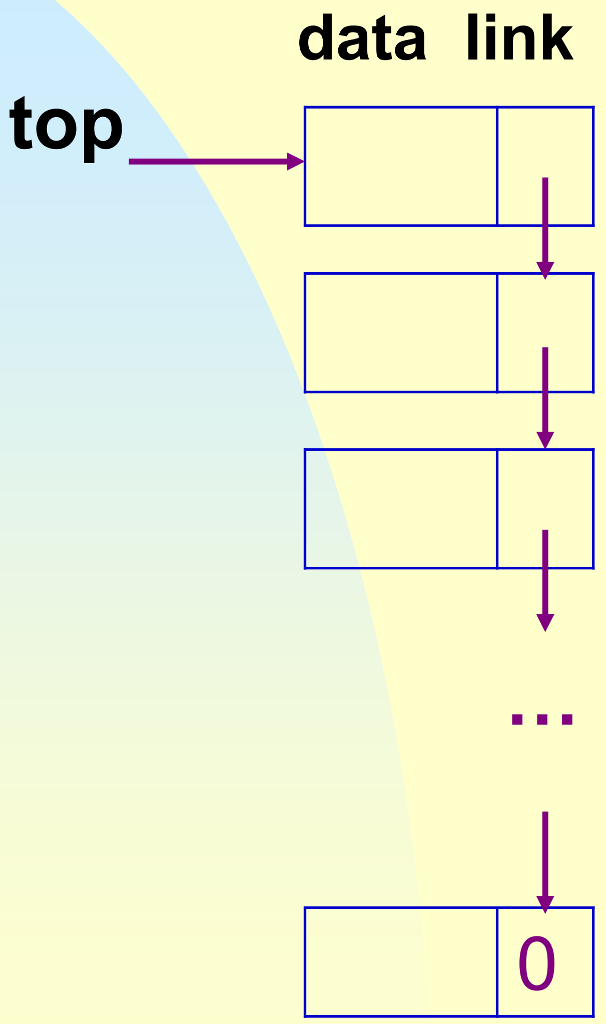
```
template <class T>
void CircularList<T>::~CircularList()
{ // delete the circular list.
   if (last) {  ChainNode <T> * first = last→link;
          last→link = av;    // (1)
          av = first; // (2)
          last = 0;
    }
}
```



A chain may be deleted in O(1) if we know its *first* and *last* nodes:

```
template <class T>
Chain<T>::~Chain()
{ // delete the chain
   if (first) {
       last→link = av;
       av = first;
       first = 0;
   }
}
```

4.6 Linked Stacks and Queues



linked stack

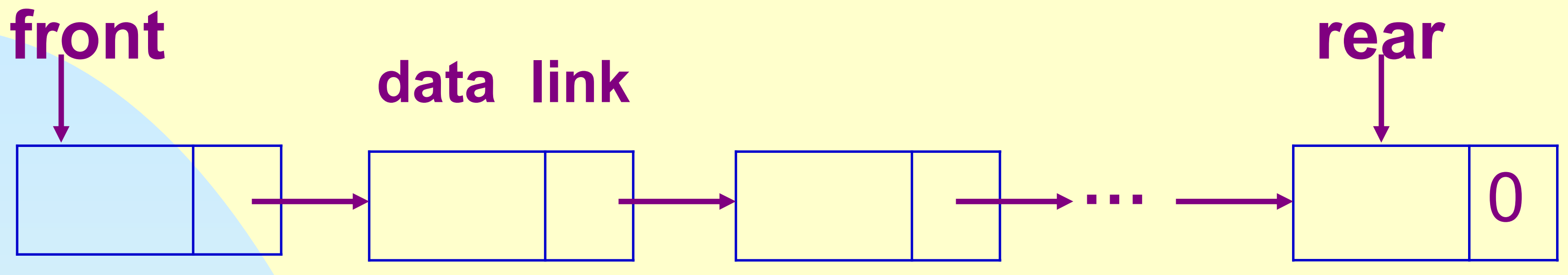
Assume the *LinkedStack* class has been declared as friend of *ChainNode<T>*.

```
template <class T>
class LinkedStack {
public:
   LinkedStack() { top=0;}; // constructor initializing top to 0
   // LinkedStack manipulation operations
…
private:
   ChainNode<T> *top;
};
```

```
template <class T>
void LinkedStack<T>::Push(const T& e) {
    top = new ChainNode(e, top);
}

template <class T>
void LinkedStack<T>::Pop()
{ // delete top node from the stack.
    if (IsEmpty()) throw "Stack is empty. Cannot delete.";
    ChainNode<T> * delNode = top;
    top = top→link;
    delete delNode;
}
```

The functions *IsEmpty* and *Top* are easy to implement, and are omitted.



linked queue

The functions of LinkedQueue are similar to those of LinkedStack, and are left as exercises.

4.7 Polynomials

4.7.1 Polynomial Representation

Since a polynomial is to be represented by a list, we say Polynomial is IS-IMPLEMENTED-BY List.

Definition: **a data object of Type A IS-IMPLEMENTED-IN-TERMS-OF a data object of Type B if the Type B object is central to the implementation of Type A object. ---Usually by declaring the Type B object as a data member of the Type A object.**

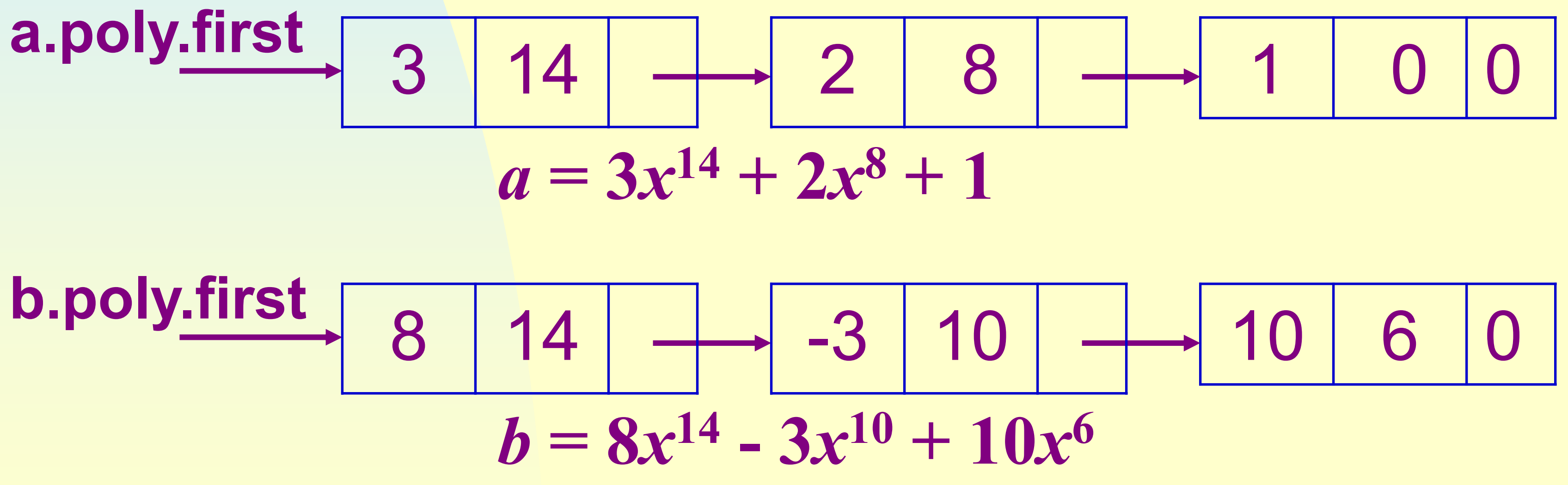
$$A(x) = a_m x^{em} + a_{m-1} x^{em-1} + \ldots + a_1 x^{e1}$$

Where $a_i \neq 0, \quad e_m > e_{m-1} > \ldots > e_2 > e_1 \geq 0$

- Make the chain ***poly*** a data member of ***Polynomial***.
- Each ***ChainNode*** will represent a term. The template ***T*** is instantiated to struct ***Term***:

```
struct Term
{ // All members of Term are public by default.
    int coef;
    int exp;
    Term Set(int c, int e) { coef = c; exp = e; return *this;};
};
```

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



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

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

4.7.2 Adding Polynomials

To add two polynomials *a* and *b*, use the chain iterators *ai* and *bi* to move along the terms of *a* and *b*.

```
Polynomial Polynomial::operaor+ (const Polynomial& b) const
{ // *this (a) and b are added and the sum returned
  Term temp;
  Chain<Term>::ChainIterator ai = poly.begin(),
                             bi = b.poly.begin();
  Polynomial c;
```

```
while (ai != poly.end() && bi != b.poly.end()) { //not null
   if (ai→exp == bi→exp) {
      int sum = ai→coef + bi→coef;
      if (sum) c.poly.InsertBack(temp.Set(sum, bi→exp));
       ai++; bi++; // to next term
   }
   else if (ai→exp < bi→exp) {
         c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
         bi++;  // next term of b
   }
   else {
         c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
         ai++;  // next term of a
    }
 }
```

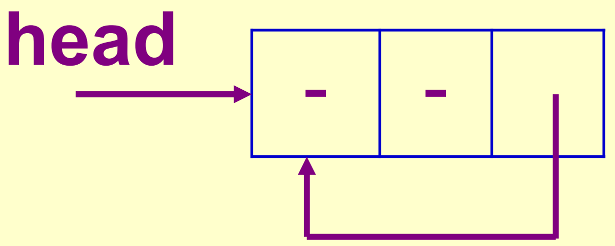
```
    while (ai != poly.end()) { // copy rest of a
        c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
        ai++;
    }
    while (bi != b.poly.end()) { // copy rest of b
        c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
        bi++;
    }
    return c;
}
```

Analysis:

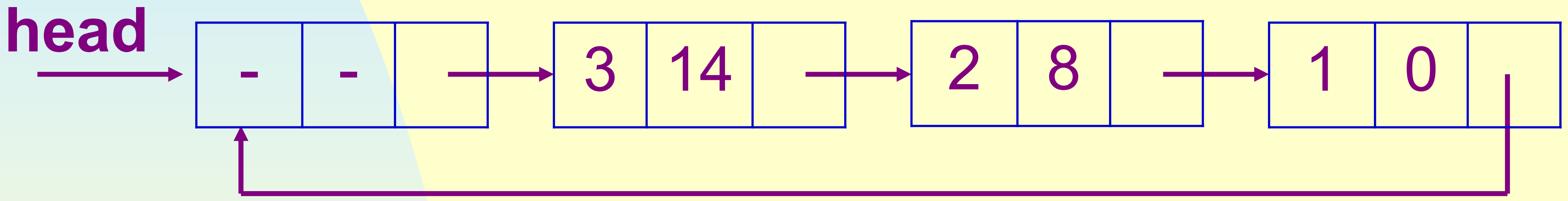
Assume *a* has *m* terms, *b* has *n* terms. The computing time is O(*m*+*n*).

4.7.3 Circular List Representation of Polynomials

Polynomials represented by circular lists with head node are as in the next slide:



(a) Zero polynomial



(b) $3x^{14} + 2x^8 + 1$

Adding circularly represented polynomials

- **The *exp* of the head node is set to –1 to push the rest of *a* or *b* to the result.**

- **Assume that the *begin*() function for class *CircularListWithHead* returns an iterator with its *current* that points to the node *head→link*.**

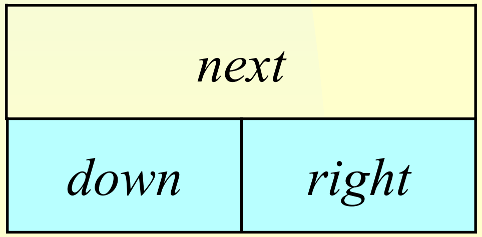
```
Polynomial Polynomial::operaor+(const Polynomial& b) const
{ // *this (a) and b are added and the sum returned
   Term temp;
   CircularListWithHead<Term>::Iterator ai = poly.begin(),
                                        bi = b.poly.begin();
   Polynomial c; //assume constructor sets head→exp = -1
   while (1) {
      if (ai→exp == bi→exp) {
         if (ai→exp == -1) return c;
         int sum = ai→coef + bi→coef;
         if (sum) c.poly.InsertBack(temp.Set(sum, ai→exp);
           ai++; bi++; // to next term
      }
```

```
    else if (ai→exp < bi→exp) {
        c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
        bi++;  // next term of b
    }
    else {
        c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
        ai++;  // next term of a
    }
  }
}
```

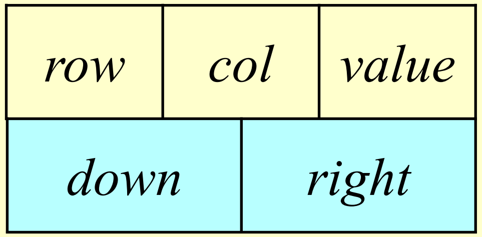
4.9 Sparse Matrices

4.9.1 Sparse Matrix Representation

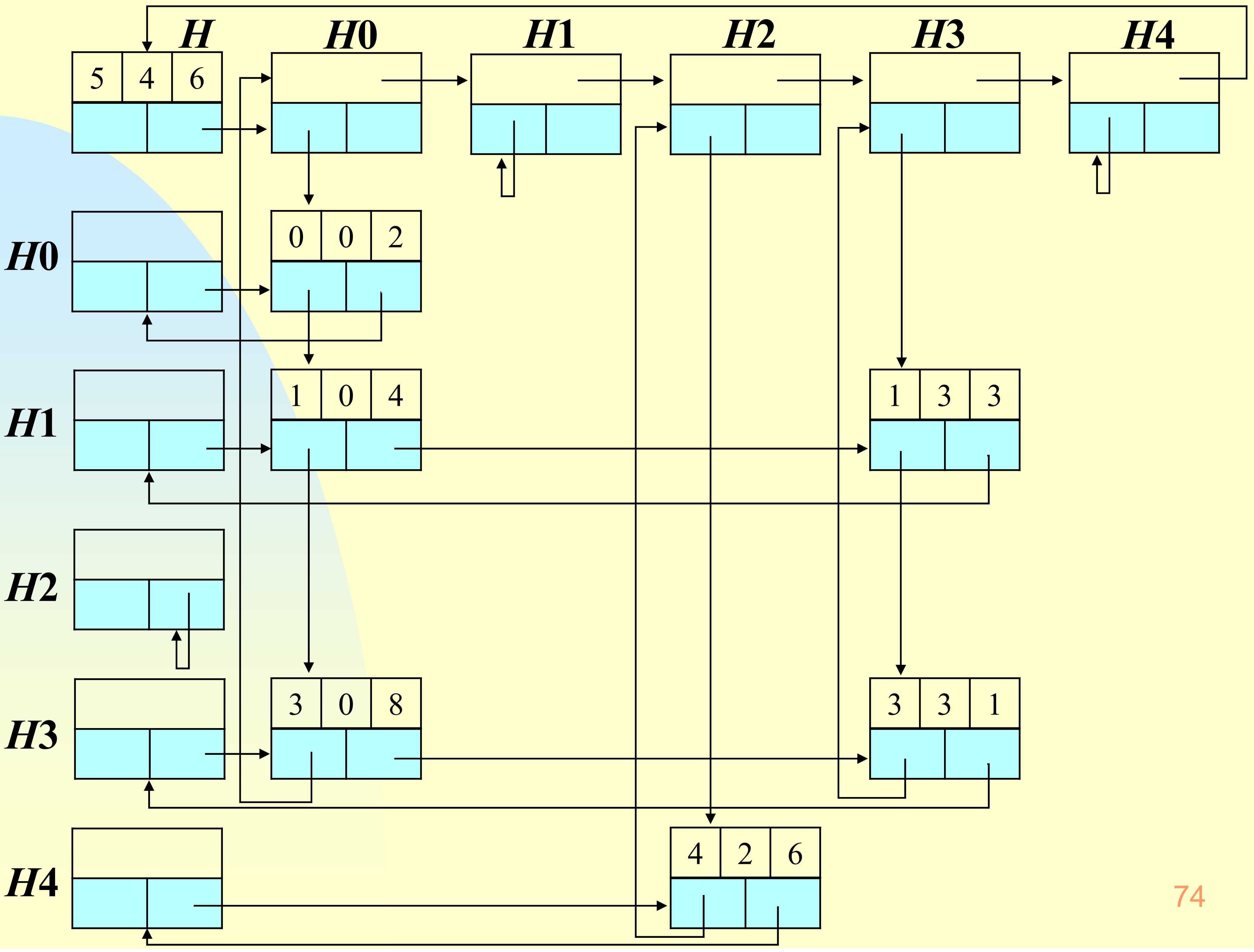
Each **header node** has a Boolean field *head* and three additional fields: *down*, *right*, and *next*. The total number of header nodes is max{number of rows, number of columns}.
Each **element node** has a Boolean field *head* and five additional fields: *row*, *col*, *value*, *down*, and *right*.
H is the header node of the list of header nodes, using the structure of element node.



header node



element node


H
5
4
6
H0
H1
H2
H3
H4
H0
0
0
2
H1
1
0
4
1
3
3
H2
H3
3
0
8
3
3
1
H4
4
2
6

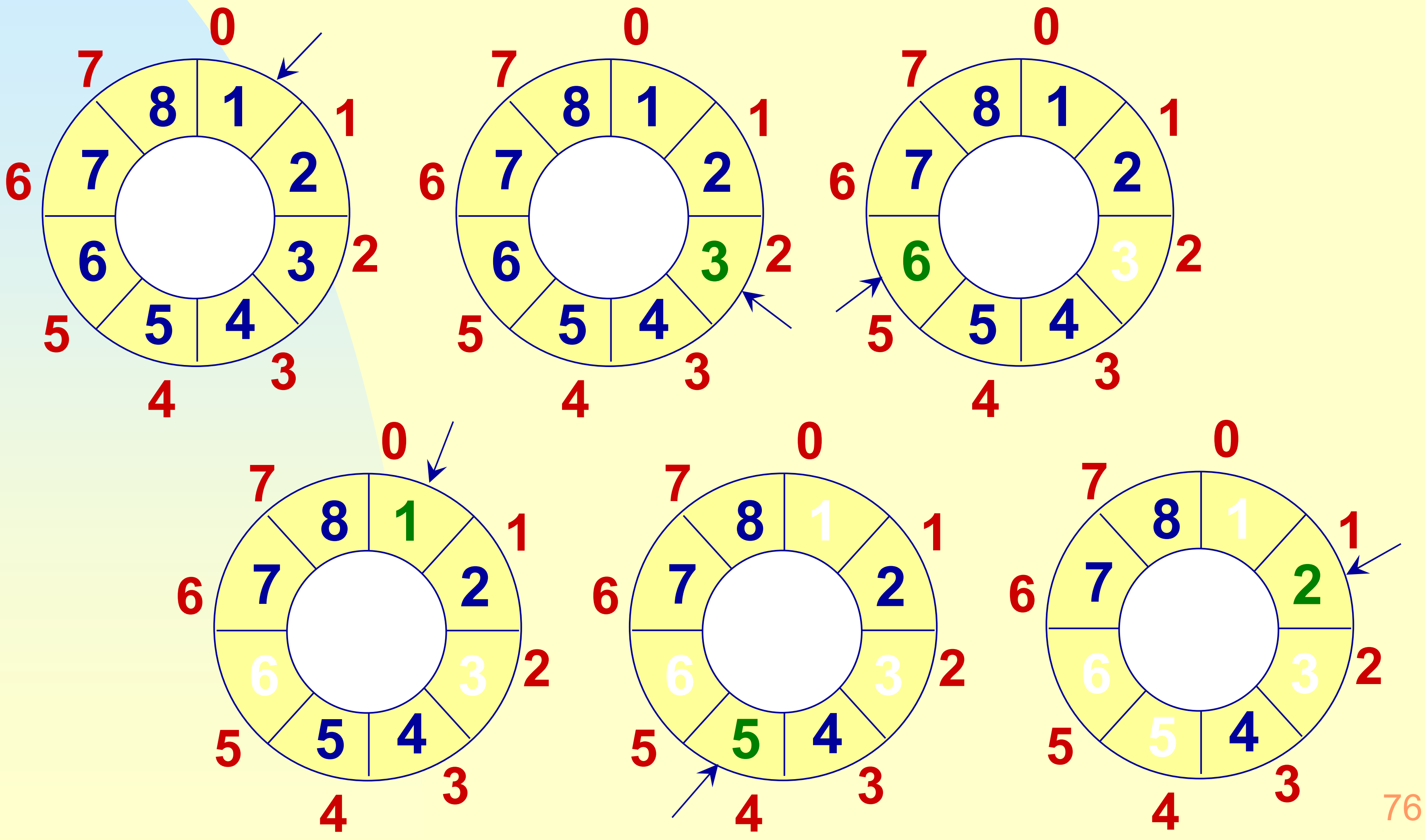
Josephus Problem

Consider the following children's game:

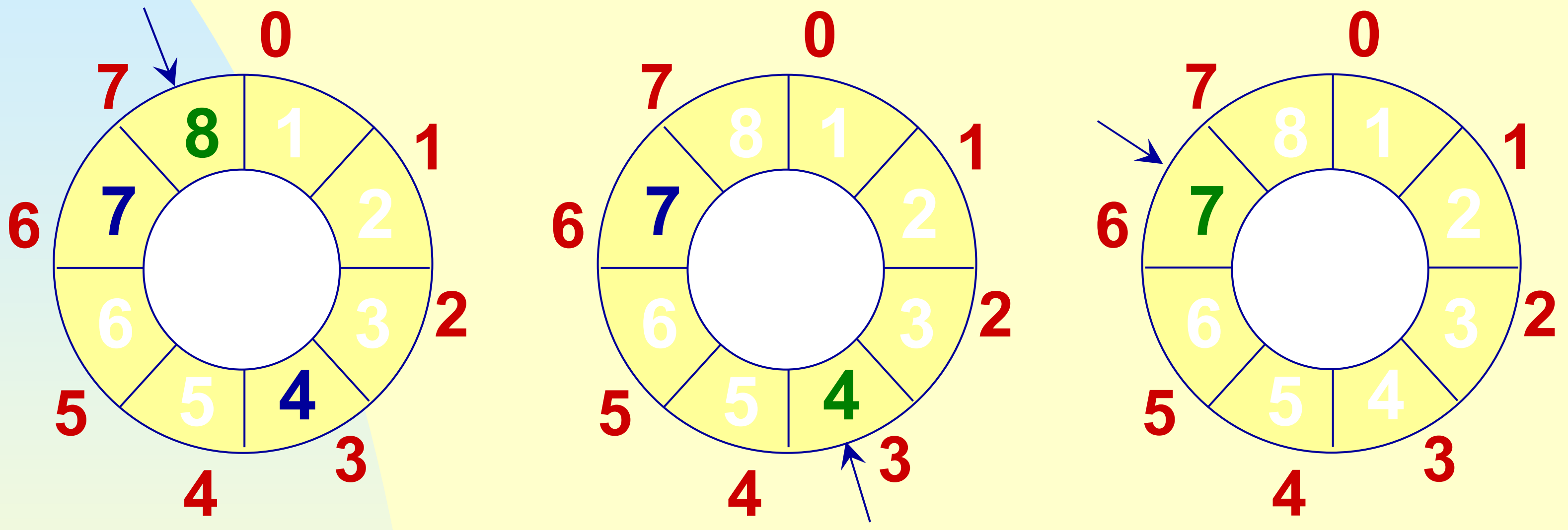
- ***n*** children stand around a circle.
- Starting with a given child and working clockwise, each child gets a sequential number, that is, his ID.
- Then starting with the first child, they count out from 1 until ***k***. The ***k***th child is now out and leaves the circle. The count starts again with the child immediately next to the eliminated one.
- Children are so removed from the circle one by one. The winner is the last child left standing.

Josephus Problem

For example, *n* = 8, *k* = 3.



Josephus Problem



n **= 8,** ***k*** **= 3**

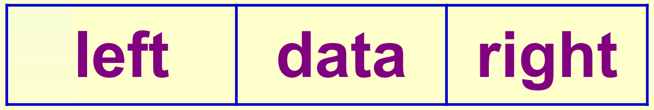
We can simulate a solution to the Josephus problem using a circular linked list.

4.10 Doubly Linked Lists

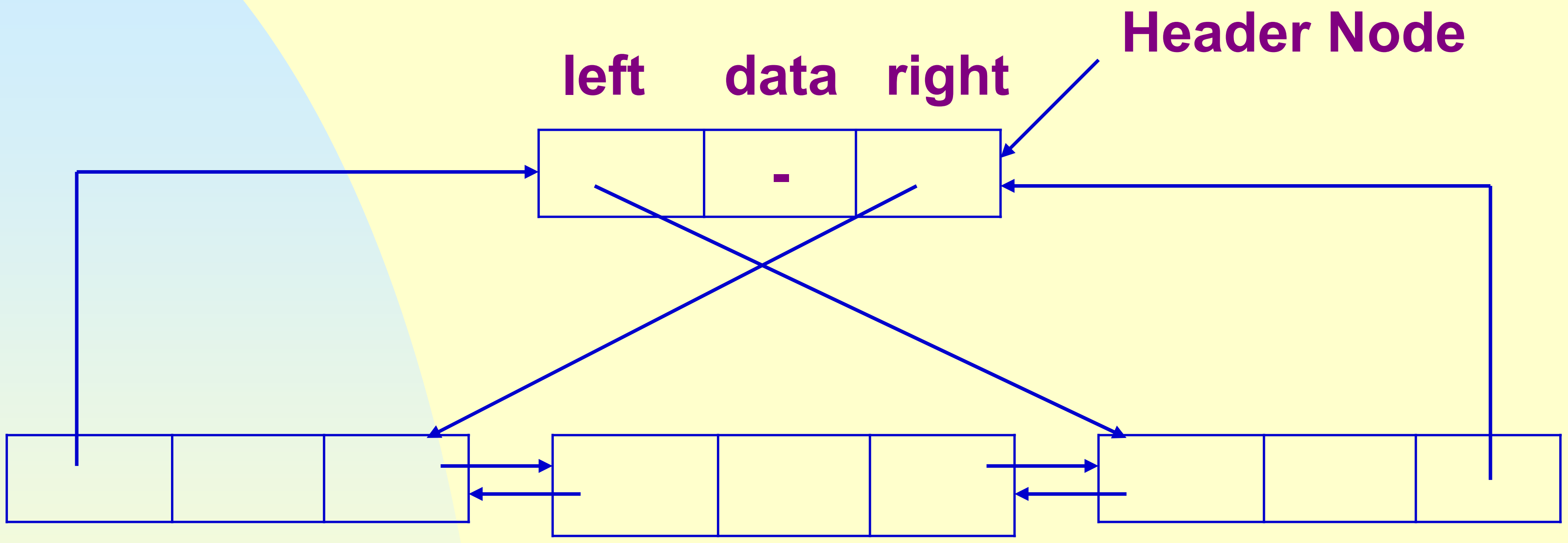
Difficulties with singly linked list:

- **can easily move only in one direction**
- **not easy to delete an arbitrary node**
    - **requires knowing the preceding node**

A node in doubly linked list has at least 3 fields: data, left and right, which makes it easy to move in both directions.



A doubly linked list may be circular. The following is a doubly linked circular list with header node:



Suppose *p* points to any node, then

p* == *p→left→right* == *p→right→left

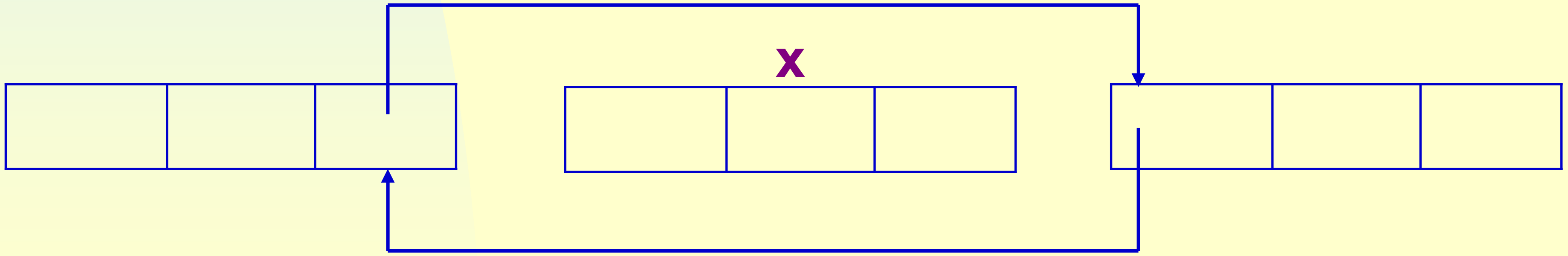
```
class DblList;

class DblListNode {
friend class DblList;
private:
   int data;
   DblListNode *left, *right;
};

class DblList {
public:
   // List manipulation operations
   …
private:
   DblListNode *first; // points to head node
};
```

Delete

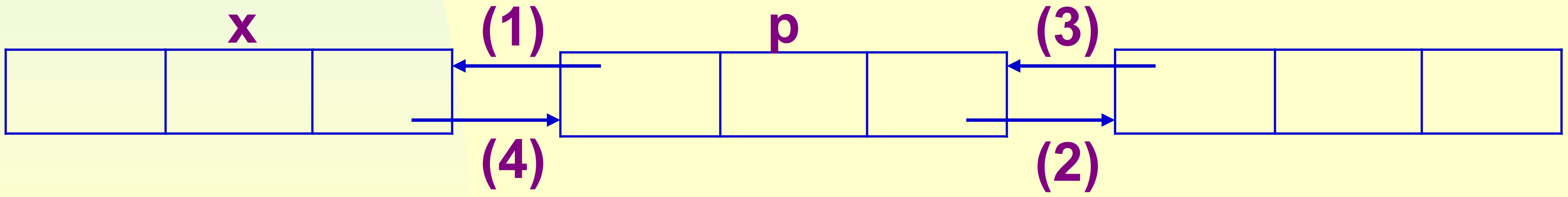
```
void DblList::Delete(DblListNode *x )
{
   if(x == first) throw “Deletion of head node not permitted”;
   else{
      x→left→right = x→right;
      x→right→left = x→left;
      delete x;
   }
}
```



Insert

```
void DblList::Insert(DblListNode *p, DblListNode *x )
{ // insert node p to the right of node x

      p→left = x;              // (1)
      p→right = x→right;  // (2)
      x→right→left = p;    // (3)
      x→right = p;             // (4)
}
```



4.11 Generalized Lists

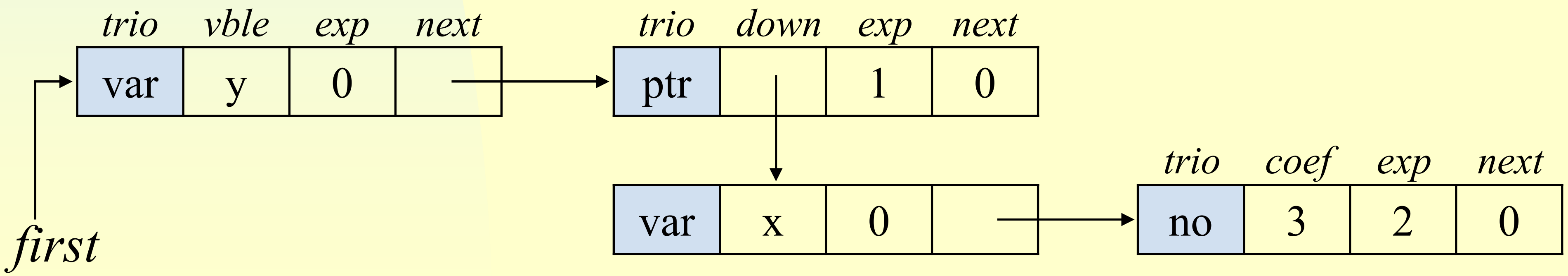
A generalized list is a linear list ($a_0, \ldots, a_{n-1}$) where element a_i ($0 \leq i \leq$ n-1) is either an **atom** or a **list**.

Examples:

- A = (): the null or empty, list; its length is zero.
- $B = (a, (b, c))$: a list of length two; its first element is the atom a, and its second element is the linear list (b, c).
- $C = (B, B, ())$: a list of length three whose first two elements are the list B, and the third element is the null list.
- $D = (a, D)$: a recursive list of length two; D corresponds to the infinite list $(a, (a, (a, \ldots)))$.

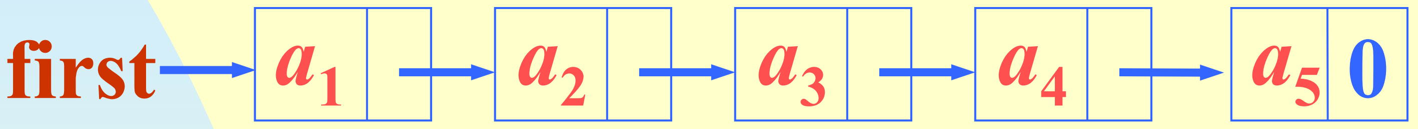
Polynomials in several variables

```
enum Triple {var, ptr, no};
class PolyNode
{
            PolyNode *next;
            int exp;
            Triple trio;
             union {
                        char vble;
                         PolyNode *down;
                         int coef;
            };
};
```



Static List

Can linked list be implemented using arrays?



	0	1	2	3	4	5
data		a_3	a_4	a_1	a_5	a_2
link	3	2	4	5	−1(0)	1

```
template <class T>
struct SLinkNode{
        T data;
        int link;    //index of next element
};
```