# Unit 5 Circularly Linked Lists & Doubly Linked List

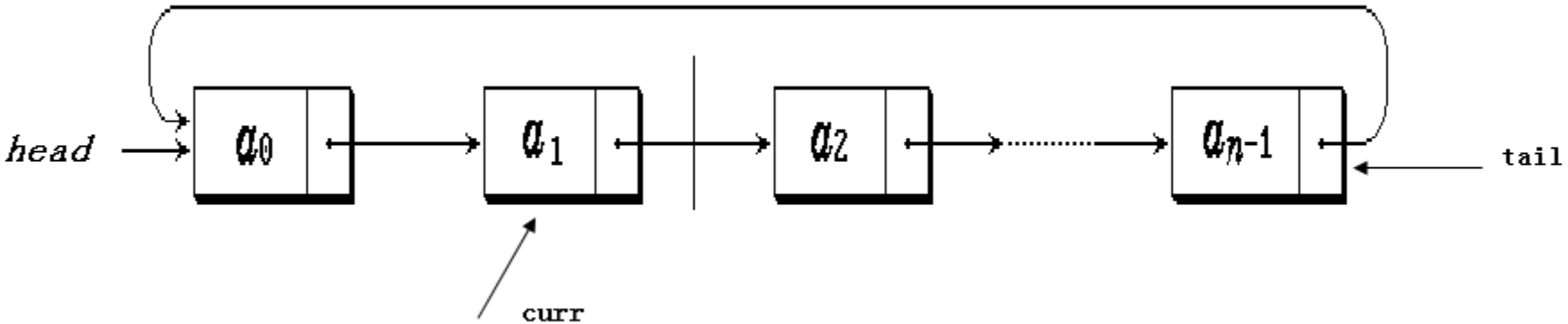
College of Computer Science, CQU

# Circularly Linked Lists

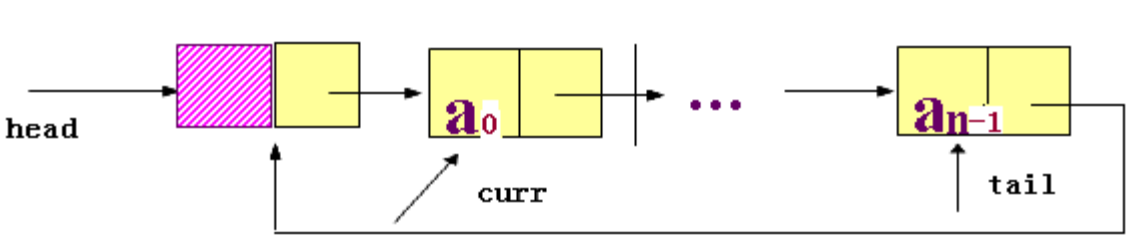
- **Singly Linked Lists**
  **the last node contain a NULL pointer**
- **Circularly Linked Lists**
  - **the last node contains a pointer to the first node**
- **Advantage**

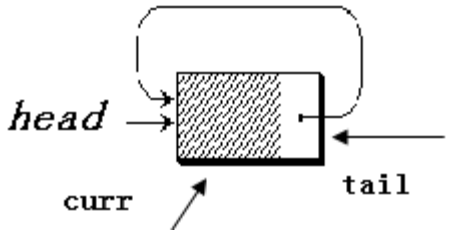
  **start from any node, can access the others.**

# Example of circular linked list




# Nonempty list & Empty list



# Example:Josehus problem

- **A description of the Josehus problem is: number 1,2, ..., n of n individuals sitting around a circle clockwise, each holding a password (positive integer). Choose a positive integer beginning as a limit on the number of reported m, starting from the first person to start a clockwise direction from a report number, report the number of reported m stop. Who reported m out of line, his password as the new m value, in a clockwise direction from the next person he began to re-reported from a number, it goes on until all the people all of the columns so far. Design a program, according to the column order prints each number.**

# Example:Josehus problem

- **Use circular link list to accomplish.**
- **Josehusproblem.cpp**

# Example:Josehus problem

- **Main function**
- **bool LList<Elem>::remove(Elem& it)**
- **void LList<Elem>::getOut(int &it,int& sum)**
- **bool LList<Elem>::append(const people& T)**

# Doubly Linked Lists

- **Singly Linked Lists**

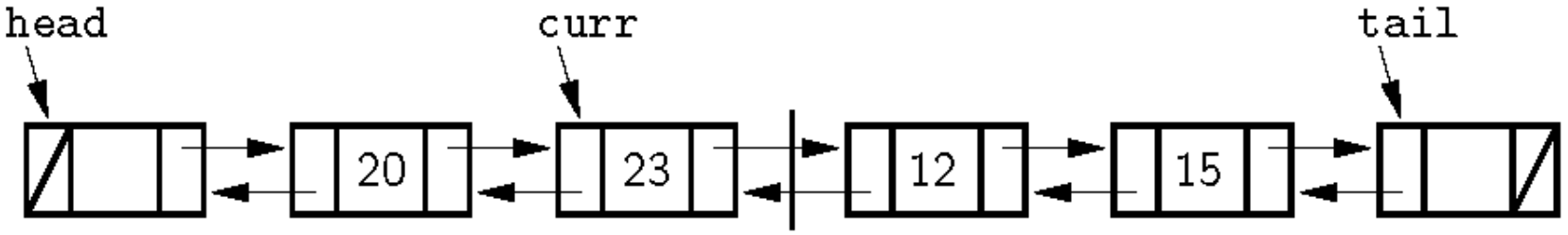
  **The singly linked list allows for direct access from a list node only to the next node in the list.**

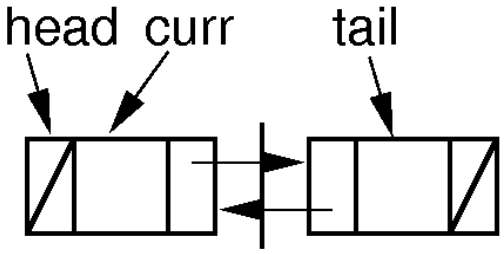
- **Doubly Linked Lists**
  - **A doubly linked list allows convenient access from a list node to the next node and also to the preceding node on the list.**

- **How to accomplish?**

**The doubly linked list node accomplishes this in the obvious way by storing two pointers: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it.**

# Doubly Linked Lists



# Doubly Linked Lists

```
// Doubly linked list link node with freelist support
template <typename E> class Link {
private:
  static Link<E>* freelist; // Reference to freelist head
public:
  E element;       // Value for this node
  Link* next;      // Pointer to next node in list
  Link* prev;      // Pointer to previous node
```

# Doubly Linked Lists

```
// Constructors
Link(const E& it, Link* prevp, Link* nextp) {
  element = it;
  prev = prevp;
  next = nextp;
}
Link(Link* prevp =NULL, Link* nextp =NULL) {
  prev = prevp;
  next = nextp;
}
```

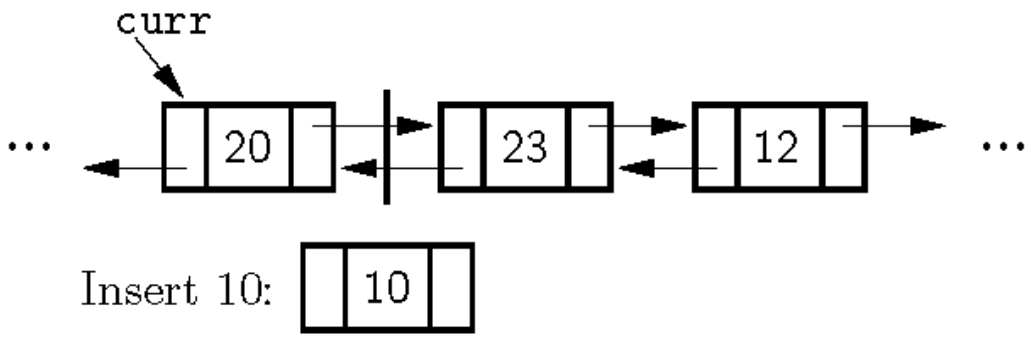
# Doubly Linked Lists

```
void* operator new(size_t) {  // Overloaded new operator
  if (freelist == NULL) return ::new Link; // Create space
  Link<E>* temp = freelist; // Can take from freelist
  freelist = freelist->next;
  return temp;                // Return the link
}
```

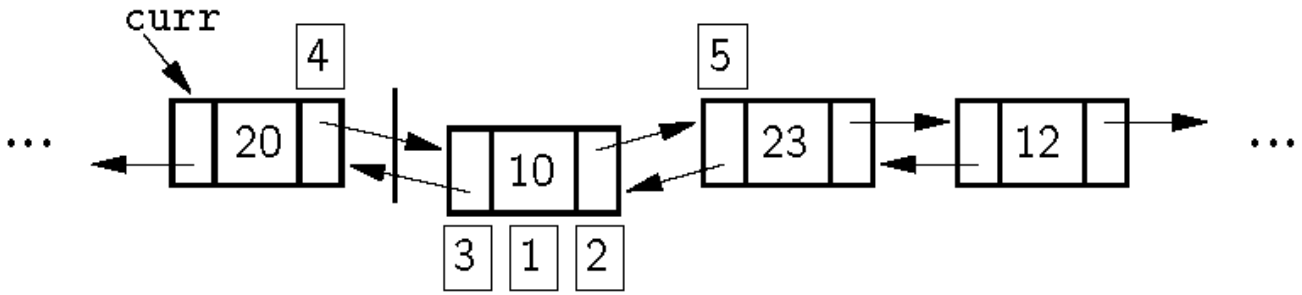
# Doubly Linked Lists

```
  // Overloaded delete operator
  void operator delete(void* ptr) {
    ((Link<E>*)ptr)->next = freelist; // Put on freelist
    freelist = (Link<E>*)ptr;
  }
};
// The freelist head pointer is actually created here
template <typename E>
Link<E>* Link<E>::freelist = NULL;
```

# Doubly Linked Insert




(a)

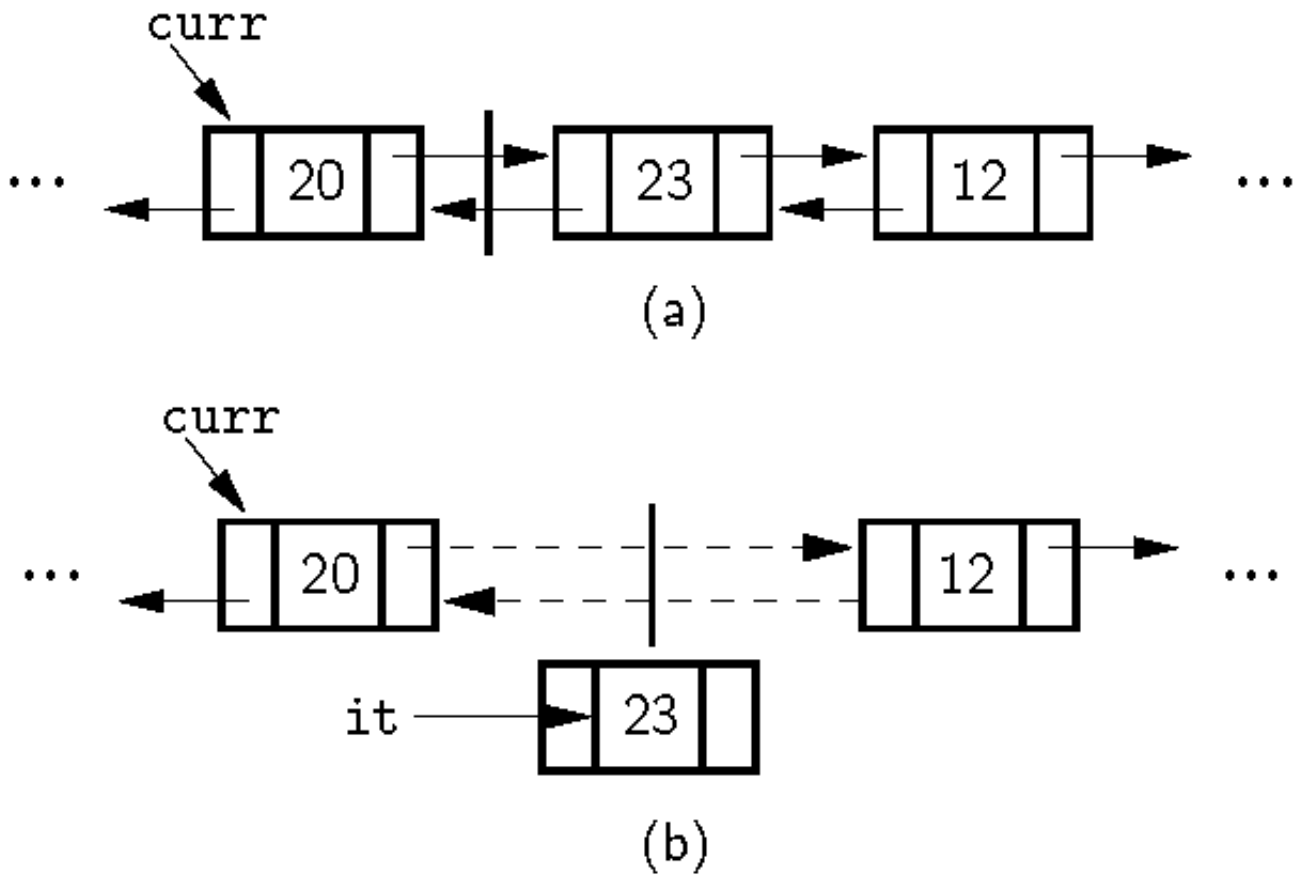



(b)

# Doubly Linked Insert

```
// Insert "it" at current position
void insert(const E& it) {
  curr->next = curr->next->prev =
    new Link<E>(it, curr, curr->next);
  cnt++;
}
```

# Doubly Linked Remove



# Doubly Linked Remove

```
// Remove and return current element
E remove() {
  if (curr->next == tail)        // Nothing to remove
    return NULL;
  E it = curr->next->element;    // Remember value
  Link<E>* ltemp = curr->next;   // Remember link node
  curr->next->next->prev = curr;
  curr->next = curr->next->next; // Remove from list
  delete ltemp;                  // Reclaim space
  cnt--;                         // Decrement cnt
  return it;
}
```

# Doubly Linked Append

```
// Append "it" to the end of the list.
void append(const E& it) {
  tail->prev = tail->prev->next =
    new Link<E>(it, tail->prev, tail);
  cnt++;
}
```

# Doubly Linked Prev

```
// Move fence one step left; no change if left is empty
void prev() {
  if (curr != head)  // Can't back up from list head
    curr = curr->prev;
}
```

# Doubly Linked List disadvantage

- **The only disadvantage of the doubly linked list as compared to the singly linked list is the additional space used.**

# Application: polynomial

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

$$= \sum_{i=0}^{n} a_i x^i$$

# Expressing the polynomial

- Express the linear list :

  P = (p0, p1, …， pn)

- It is also unsuitable to express the form like S ( X ) = 1 + $3x^{10000}$
- Writing factor and index number

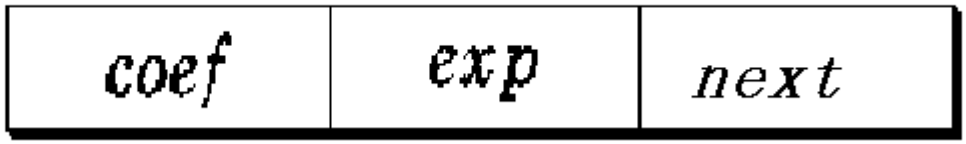
  （（p0, e0）, (p1, e1), ···, (pm,em) ）

- How about the defects

# The link expressing

- Every one node add data member Nexts during the polynomial chained list being living is expressed , As the link pointer 。

$data \equiv Term$




- Strong point is :

  The number of item of polynomial may rise dynamicly 。

  It is convenient to insert,delete the element 。

# Polynomial node definition

```
Class term{
    private
      int coef;
      int exp;
      term * next;
}
```

# Polynomial type definition

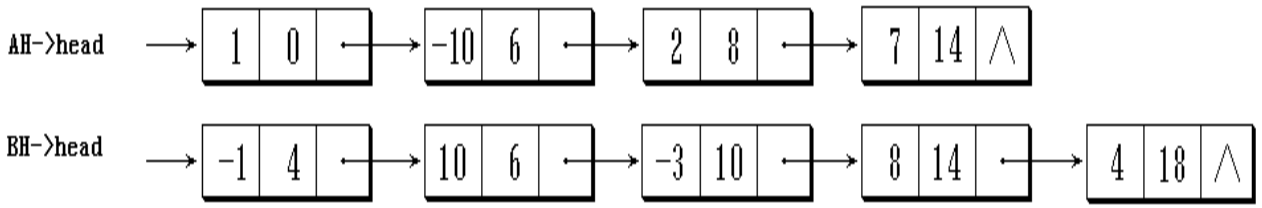
ADT Polynomial {
Data object： D＝{ ai | ai ∈TermSet, i=1,2,...,m, m≥0 }

Data relationship： R1＝{ <ai-1 ,ai >|ai-1 ,ai∈The index number value of ai-1 ＜ The index number value of ai， i=2,...,n }
Basic opertions：
CreatPolyn ( &P )
DestroyPolyn ( &P )
PrintPolyn ( &P )
AddPolyn (…)
SubtractPolyn (…)
MultiplyPolyn ( … )
PolynLength ( P )
} ADT Polynomial

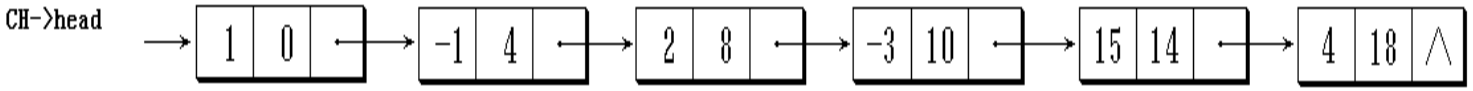
# Polynomial adding to of chained list

***AH = 1 - 10x6 + 2x8 +7x14***

***BH = - x4 + 10x6 - 3x10 + 8x14 +4x18***




(a) 两个相加的多项式




(b) 相加结果的多项式

# Polynomial

**Polymial.cpp**

# Reference

- **P115----P120**

# Preview

- **Chapter 4, pp.**

# End

# Thank you for listening!