Unit 5 Circularly Linked Lists & Doubly Linked List

College of Computer Science, CQU

Circularly Linked Lists

- **Singly Linked Lists**

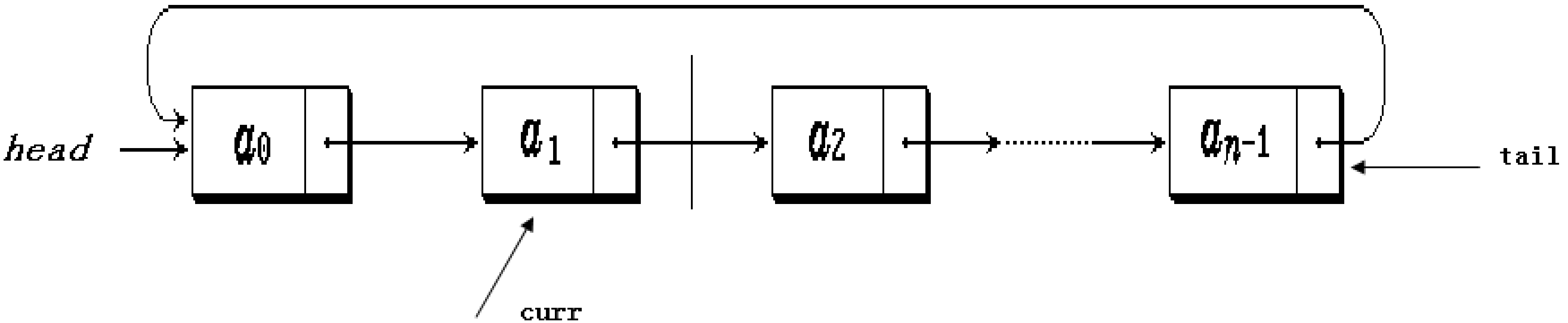
  the last node contain a NULL pointer

- **Circularly Linked Lists**
  - **the last node contains a** pointer to the first node

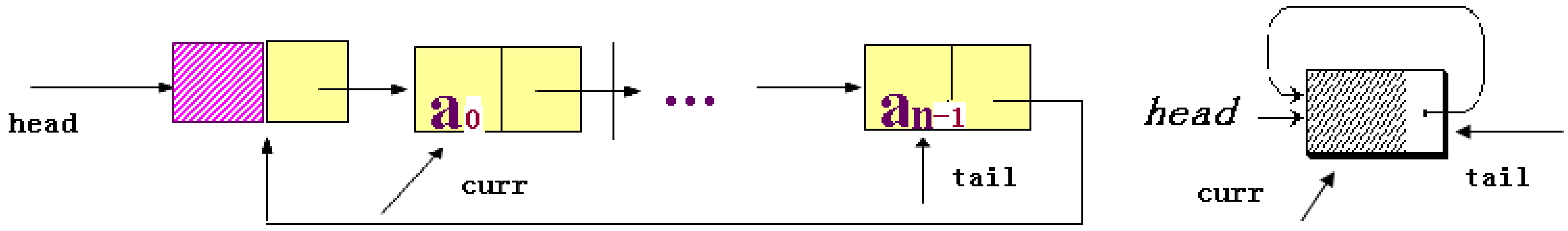
- **Advantage**

  start from any node, can access the others.

Example of circular linked list



Nonempty list & Empty list



Example:Josehus problem

- **A description of the problem are: 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 acomplish.**
- **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

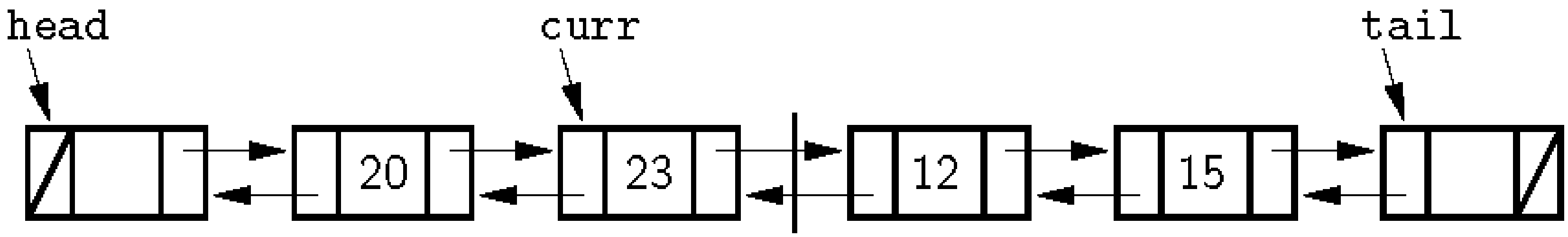
- **Singly Linked Lists**

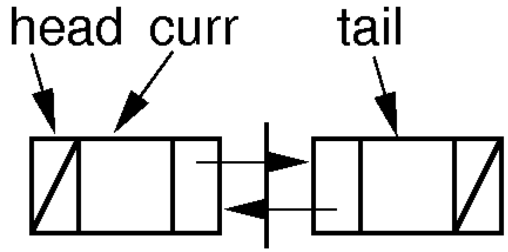
  The singly linked list allows for direct access from a list node only to the next node in the list.

- **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;
}
```

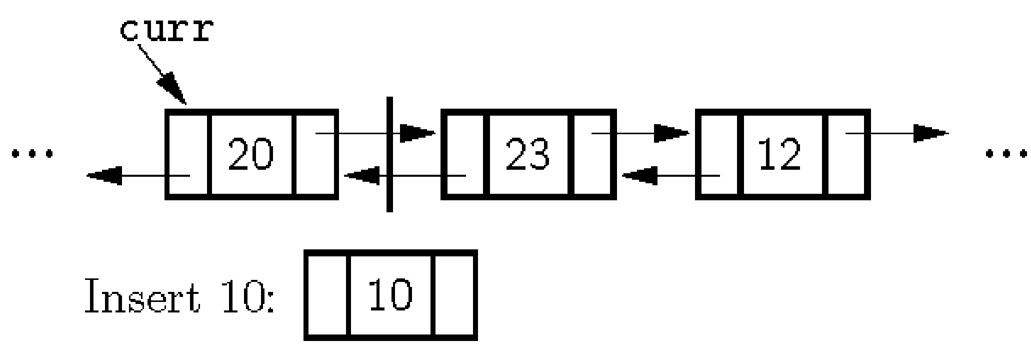
Doubly Linked Lists

```
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
}
```

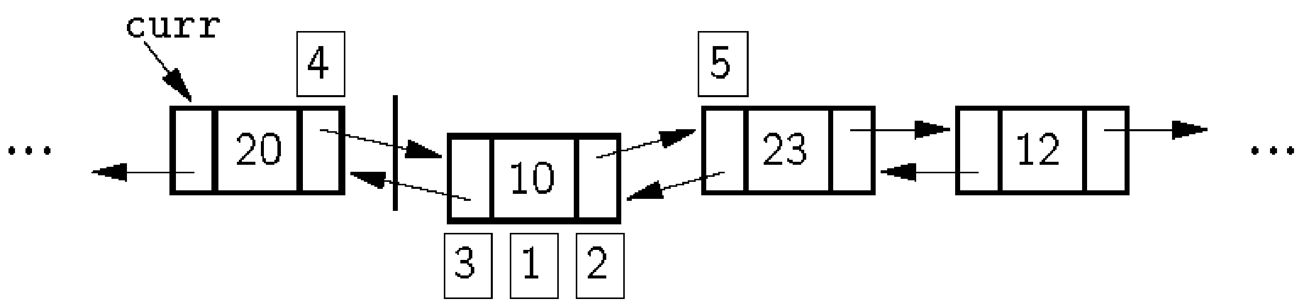
Doubly Linked Lists

```
// 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)

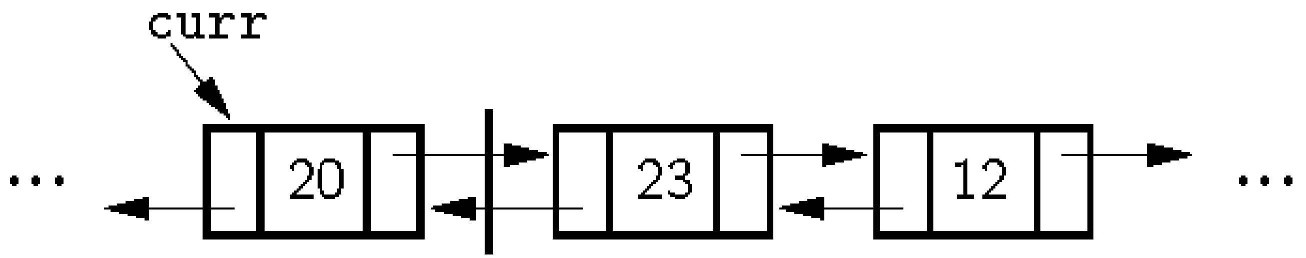


(b)

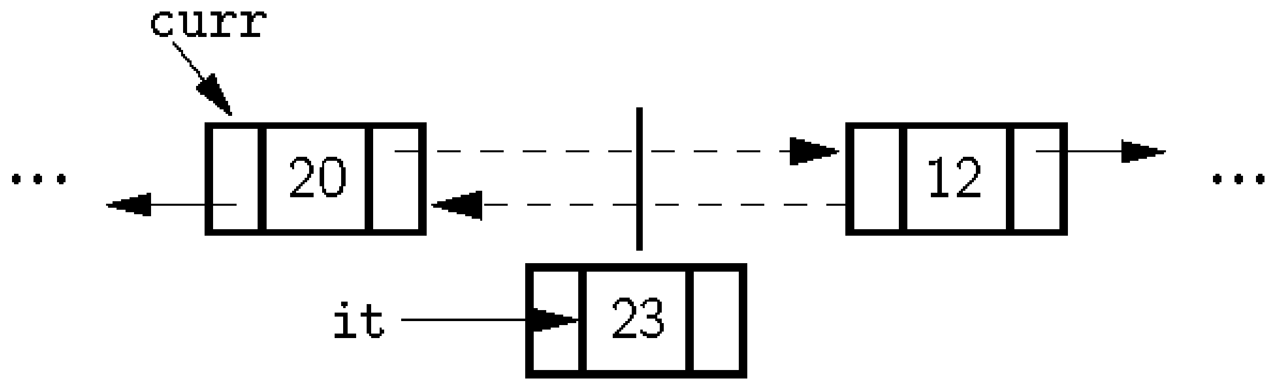
Doubly Linked Insert

```
// 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



(a)



(b)

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

（（p1, e1）, (p2, e2), ┅, (pm,em) ）

- How about the defects

The link expressing

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

$data = Term$

coef	exp	next

- Strong point is：

  The number of item of polynomial may rise dynamicly。

  It is convenient to insert,delete the element。

Polynomial type definition

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

Data relationship： R1＝{ <a_{i-1} ,a_i >|a_{i-1} ,a_i∈The index number value of A_{i-1} ＜The index number value of A_i， i=2,...,n }
Basic opertions：

CreatPolyn (&P)
DestroyPolyn (&P)
PrintPolyn (&P)
AddPolyn (…)
SubtractPolyn (…)
MultiplyPolyn (…)
PolynLength (P)

} ADT Polynomial

Polynomial node definition

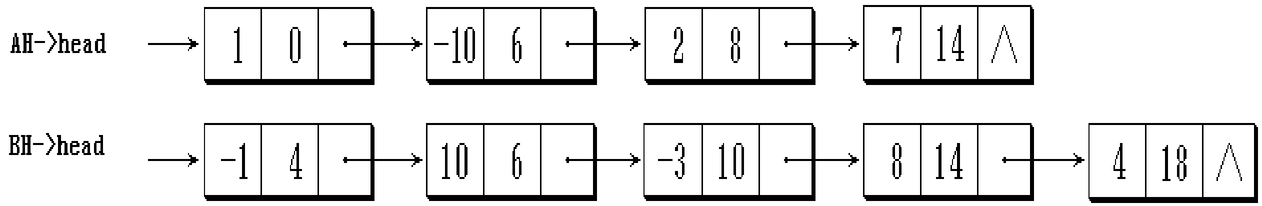
```
Class term{
   private
     int coef;
     int exp;
      term * link;
}
```

Polynomial node definition

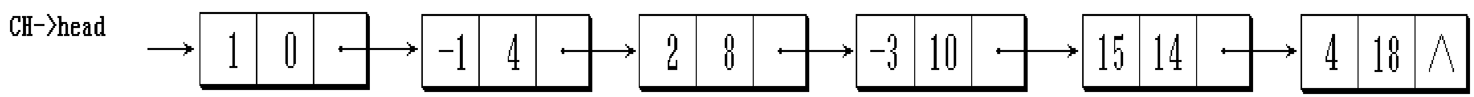
```
Class Term{
   public
     int coef;
     int exp;

     Term(const int c_t=1, const int e_t=0)
      {
            coef = c_t;
            exp = e_t;
       }
      Term(const Term& t)
      {
            coef = t.coef;
            exp = t.exp;
       }
}
```

Polynomial adding to of chained list

$$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$$
$$BH = - x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$$



(a) 两个相加的多项式



(b) 相加结果的多项式

```
void addPoly(Llist<Term> *p1,  Llist<Term> *p2)
{
        p1->moveToStart();
        p2->moveToStart();
        while(p2->length() > 0)
        {
          if(  p1->currPos()==p1->length() ||
               p1->getValue().exp>p2->getValue().exp )
          {    p1->insert(p2->remove()); p1->next(); continue;    }

          if( p1->getValue().exp == p2->getValue().exp )
          {
               Term t2 = p2->remove();
               Term t1 = p1->remove();
               if(t1.coef + t2.coef != 0)
                     p1->insert(Term(t1.coef+t2.coef, t1.exp));
          }
          p1->next();
        }
}
```

```
Llist<Term>* addPoly(const Llist<Term> *p1,  const Llist<Term> *p2)
{
        p1->moveToStart(); p2->moveToStart();
        Llist<Term>* pp = new Llist<Term>;
        while(p1->currPos()< p1->length()  && p2->currPos()< p2->length() )
        {
            if( p1->getValue().exp == p2->getValue().exp) {
                if(p1->getValue().coef+p2->getValue().coef != 0)
                    pp->append(Term(p1->getValue().coef+p2->getValue().coef,
                    p1->getValue().exp));
                p1->next();     p2->next();
             } else{
                Llist<Term>* tt = (p1->getValue().exp<p2->getValue().exp? p1  : p2);
                pp->append(tt->getValue);
                tt->next();     }
        } // end of while
        Llist<Term>* tmp = (p1->currPos()<p1->length()? p1 : p2);
        while( tmp->currPos()<tmp->length())
            { pp->append(tmp->getValue()); tmp->next();}
        return pp;
}
```

Polynomial

Polymial.cpp

Reference

- **P115----P120**

-End-