Design and Analysis of Data Structures and Algorithms

Warin Wattanapornprom Ph.D.

Classes and Objects

- หัดเขียนโปรแกรมเชิงวัตถุมากขึ้นมาอีกหน่อยนะ

```
#include <iostream>
using namespace std;
class Box
{
   public:
      double length;          // Length of a box
      double breadth;         // Breadth of a box
      double height;          // Height of a box

      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};
```

Classes and Objects

```
double Box::getVolume(void){
    return length * breadth * height;
}

void Box::setLength( double len ){
    length = len;
}

void Box::setBreadth( double bre ){
    breadth = bre;
}

void Box::setHeight( double hei ){
    height = hei;
}
```

Classes and Objects

```
int main( ){
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);


        volume = Box1.getVolume();
        cout << "Volume of Box1 : " << volume <<endl;

        volume = Box2.getVolume();
        cout << "Volume of Box2 : " << volume <<endl;
       return 0;
 }
```

Classes and Objects

```
#include <iostream>
using namespace std;

class Line{
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration
   private
      double length;
};
```

public สามารถเรียกได้ทุกที่ ทั้งภายในและภายนอกคลาส

private สามารถเรียกได้เฉพาะในคลาส

protected สามารถเรียกได้เฉพาะในคลาส และคลาสที่ขยายคลาสนี้

Classes and Objects

```
Line::Line(void){
    cout << "Object is being created" << endl;
}

Line::~Line(void){
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len ){
    length = len;
}

double Line::getLength( void ){
    return length;
}
```

- constructor และ destructor เป็น method ที่พิเศษ ใน class หนึ่งๆ เราอาจจะกำหนดให้มี constructor, destructor หรือไม่มีก็ได้
- โดย constructor จะเป็น method ที่มักจะใช้เซตค่าเริ่มต้น เมื่อเราทำการเรียก class นั้นๆ
- ส่วน destructor เป็น method ที่ให้ class ทำอะไรบางอย่างก่อนที่ class นั้นจะถูกทำลายลง

Classes and Objects

```
int main( )
{
   Line line;
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;
   return 0;
}
```

```
Object is being created
Length of line : 6
Object is being deleted
```

Template

- ในการสร้างฟังก์ชัน บางครั้งเราก็ต้องการให้ฟังก์ชันของเราใช้งานได้กับตัวแปรต่างชนิดกัน กล่าวคือใช้กับตัวแปรชนิด int ก็ได้ ใช้กับตัวแปรชนิด float ก็ได้ ใช้กับตัวแปรชนิด double ก็ได้
- ในภาษา C++ เราจะสามารถสร้างฟังก์ template เพื่อแก้ปัญหาดังกล่าวได้ รูปแบบของฟังก์ชัน template จะเป็นแบบนี้

  - **`template<class`** ชนิดของตัวแปร **`>`**

Template

```
#include <iostream>
using namespace std;

template<class T>
T product(T a,T b){
   return a*b;
}

int main(){
   int a=10,b=20;
   double c=1.5,d=2.5;

   cout << "The product of a and b ="<<product(a,b)<<endl;
   cout << "The product of c and d ="<<product(c,d)<<endl;

  return(0);
}
```

Template

- การสร้าง class Template ใน C++
  - ```
    class rectangle{
      public:
        double area(double a, double b);
    }
    ```
- เมื่อพิจารณา class ที่ชื่อว่า rectangle ข้างบน จะเห็นว่า method area จะต้องรับค่าเป็นตัวแปรชนิด double เท่านั้น
- ทีนี้เราอาจจะต้องการให้ method ใน class ของเรารับตัวแปรชนิดอื่นๆ เช่น int, float ได้ด้วย เราสามารถทำได้โดยการสร้าง template สำหรับ class ขึ้นมา
    ```

Template

```
template<class T>
class rectangle{
  public:
     T area(T a, T b);
};

template<class T>
T rectangle<T>::area(T a,T b){
     return a*b;
};
```

Template

```
int main()
{
    rectangle<double> rec1;
    rectangle<int> rec2;

    cout << "area rec1 =" << rec1.area(1.5,2.5)<<endl;
    cout << "area rec2 =" << rec2.area(5,2)<<endl;
return 0;
}
```

List - รายการ

- คือลำดับรายการของข้อมูล
- มีตัวระบุตำแหน่งหรือคั่นหน้า
- ก็คือมีอะไรมาเรียงกันนั่นแหละ เช่น
- $A_1, A_2, A_3, \ldots, A_n$

ตัวแรก ตัวสุดท้ายของลิสต์

List Concepts

ตัวระบุตำแหน่ง

เลื่อนซ้ายและขวาได้

หน้าตาของลิสต์

<20, 23 | 12, 15>

<20, 23, 12 | 15>

เราควรจะทำอะไรกับ list ได้บ้าง

- Find - หาตำแหน่งที่อยู่ของสมาชิกตัวหนึ่งๆ
- Insert – ใส่สมาชิกใหม่ลงในตำแหน่งที่กำหนด
- findKth – รีเทิร์นสมาชิกตัวที่ k
- Remove – เอาสมาชิกที่กำหนดออกจากลิสต์
- Head – รีเทิร์นสมาชิกตัวแรกในลิสต์
- Tail – รีเทิร์นลิสต์ที่เอาสมาชิกตัวแรกออกไปแล้ว
- Append - ใส่สมาชิกใหม่ที่ท้ายรายการ

List ภาพรวม

```
template <class Elem> class List {
public:
  virtual void clear() = 0;
  virtual bool insert(const Elem&) = 0;
  virtual bool append(const Elem&) = 0;
  virtual bool remove(Elem&) = 0;
  virtual void setStart() = 0;
  virtual void setEnd() = 0;
  virtual void prev() = 0;
  virtual void next() = 0;
   virtual int leftLength() const = 0;
  virtual int rightLength() const = 0;
  virtual bool setPos(int pos) = 0;
  virtual bool getValue(Elem&) const = 0;
  virtual void print() const = 0;
};
```

List ตัวอย่าง

```
List: <12 | 32, 15>

MyList.insert(99);

Result: <12 | 99, 32, 15>
```

เพิ่มรายการลงลิสต์

List Find Function

```
// Return true iff K is in list
bool find(List<int>& L, int K) {
  int it;
  for (L.setStart(); L.getValue(it); L.next())
    if (K == it) return true;  // Found it
  return false;                // Not found
}
```

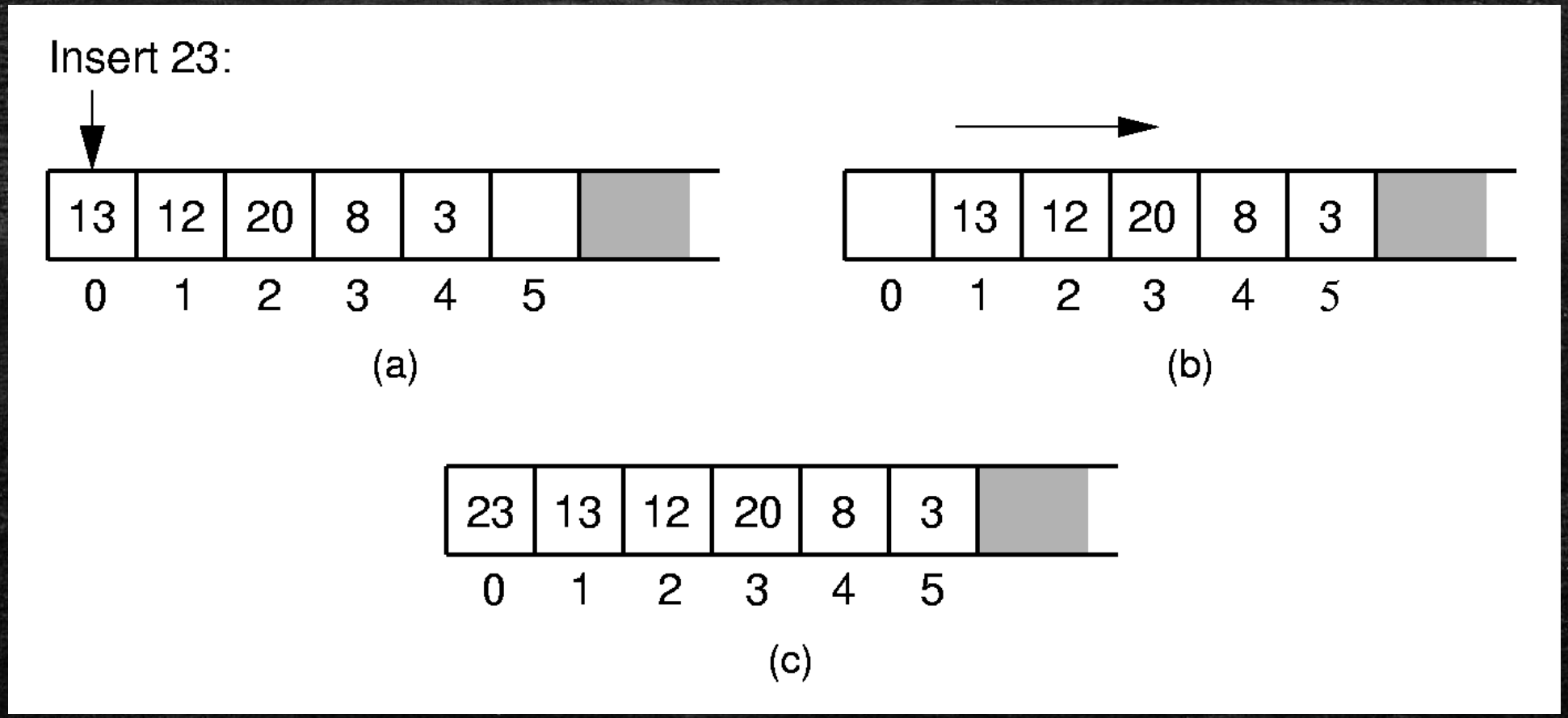
อะไรจะเกิด ถ้าใช้อาร์เรย์ทำ list

- อย่าลืมว่า ต้องบอกขนาดอาร์เรย์ล่วงหน้า
- Find ใช้เวลา $O(n)$
  - Find นั้นต้องหาเรียงตัว นับจากตัวแรกของอาร์เรย์
- findKth ใช้เวลาคงที่ เพราะใช้ index หาได้ทันที
- Insert กับ remove จะใช้เวลานาน
  - เพราะอาจต้องเลื่อนทุกๆสมาชิกในอาร์เรย์

Array List

```
template <typename E> // Array-based list implementation
class AList : public List<E> {
  private:
   int maxSize; // Maximum size of list
   int listSize; // Number of list items now
   int curr; // Position of current element
   E* listArray; // Array holding list elements
  public:
   AList(int size=defaultSize) { // Constructor
   maxSize = size;
   listSize = curr = 0;
   listArray = new E[maxSize];
  }
   ~AList() { delete [] listArray; } // Destructor
   void clear() { // Reinitialize the list
     delete [] listArray; // Remove the array
     listSize = curr = 0; // Reset the size
     listArray = new E[maxSize]; // Recreate array
   }
```

Array-Based List Insert



Array List

```
.....
// Insert "it" at current position

 void insert(const E& it) {
  Assert(listSize < maxSize, "List capacity exceeded");
  for(int i=listSize; i>curr; i--) // Shift elements up
     listArray[i] = listArray[i-1]; // to make room
  listArray[curr] = it;
  listSize++; // Increment list size
 }
 void append(const E& it) { // Append "it"
  Assert(listSize < maxSize, "List capacity exceeded");
  listArray[listSize++] = it;
 }
```

```
//listArray[listSize+1]=it;
//listSize=listSize+1;
```

Array List

```
.....
// Remove and return the current element.
  E remove() {
   Assert((curr>=0) && (curr < listSize), "No element");
   E it = listArray[curr]; // Copy the element
   for(int i=curr; i<listSize-1; i++) // Shift them down
      listArray[i] = listArray[i+1];
   listSize--; // Decrement size
   return it;
  }

  void moveToStart() { curr = 0; } // Reset position
  void moveToEnd() { curr = listSize; } // Set at end
  void prev() { if (curr != 0) curr--; } // Back up
  void next() { if (curr < listSize) curr++; } // Next
```

Array List

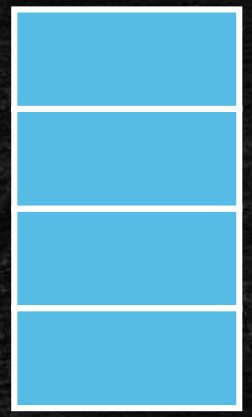
- .....

```
    // Return list size
    int length() const { return listSize; }
    // Return current position
    int currPos() const { return curr; }
    // Set current list position to "pos"
    void moveToPos(int pos) {
     Assert ((pos>=0)&&(pos<=listSize), "Pos out of range");
     curr = pos;
    }
    const E& getValue() const { // Return current element
     Assert((curr>=0)&&(curr<listSize),"No current element");
     return listArray[curr];
    }
};
```

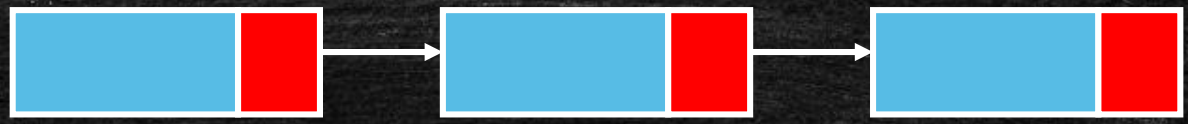
ลิงค์ลิสต์ (Linked List)

- เป็นการจัดเก็บชุดข้อมูลเชื่อมโยงต่อเนื่องกันไปตามลำดับ โครงสร้างข้อมูลแบบลิงค์ลิสต์จะประกอบไปด้วยส่วนที่เรียกว่าสมาชิก (Node) ส่วนเก็บข้อมูล (Data) และตำแหน่งของสมาชิกตัวถัดไป (Link)

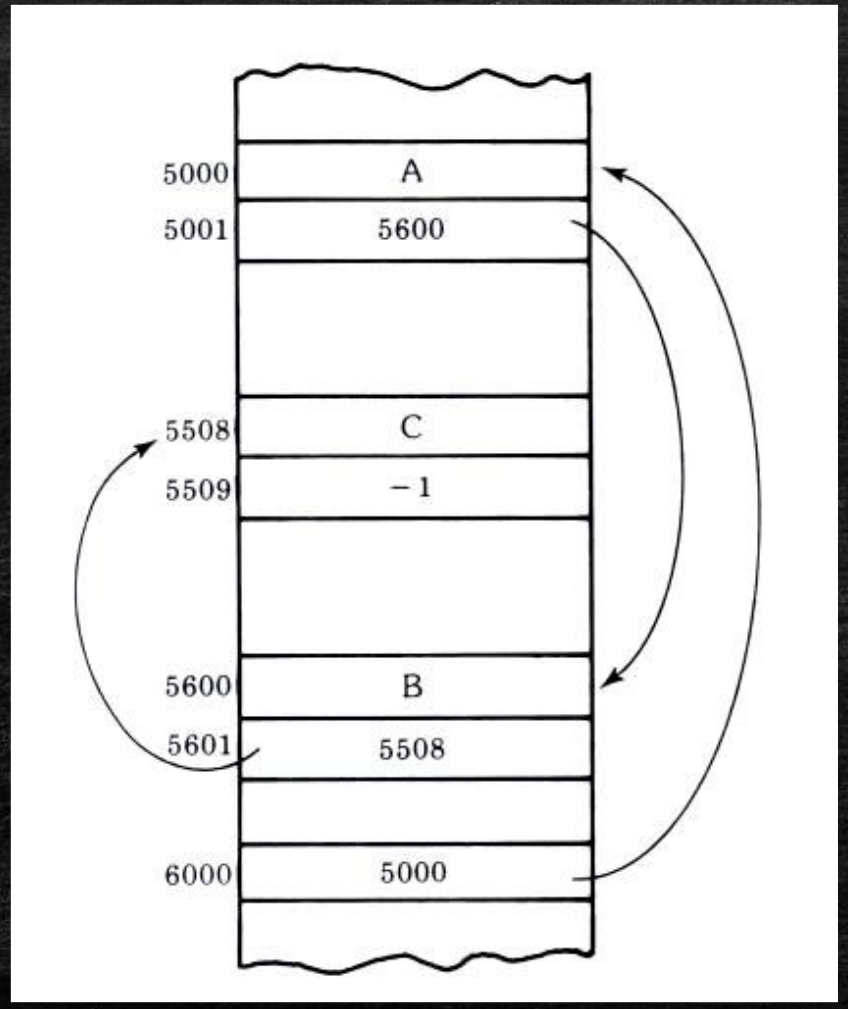
อาร์เรย์(Array)



ลิงค์ลิสต์(Linked List)

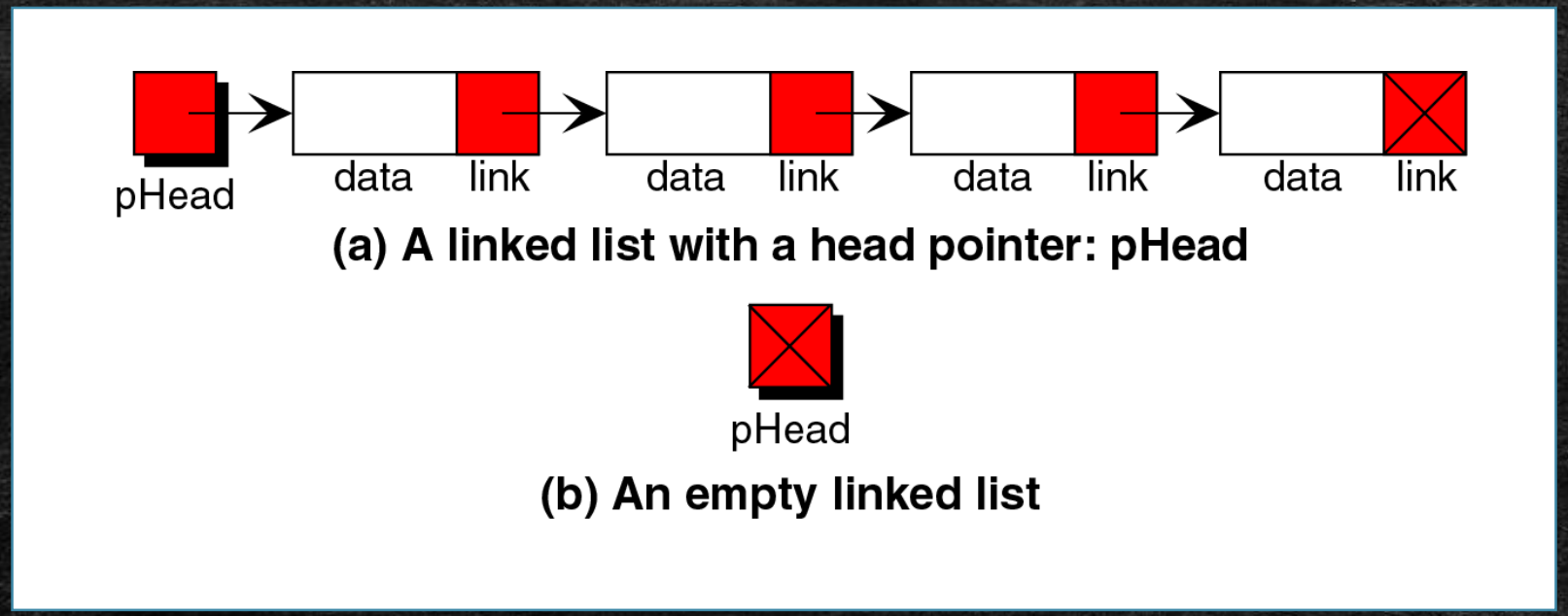


โครงสร้างข้อมูลลิงค์ลิสต์ (Linked List)



การแทนลิงค์ลิสต์
ในพื้นที่หน่วยความจำ

โครงสร้างข้อมูลลิงค์ลิสต์

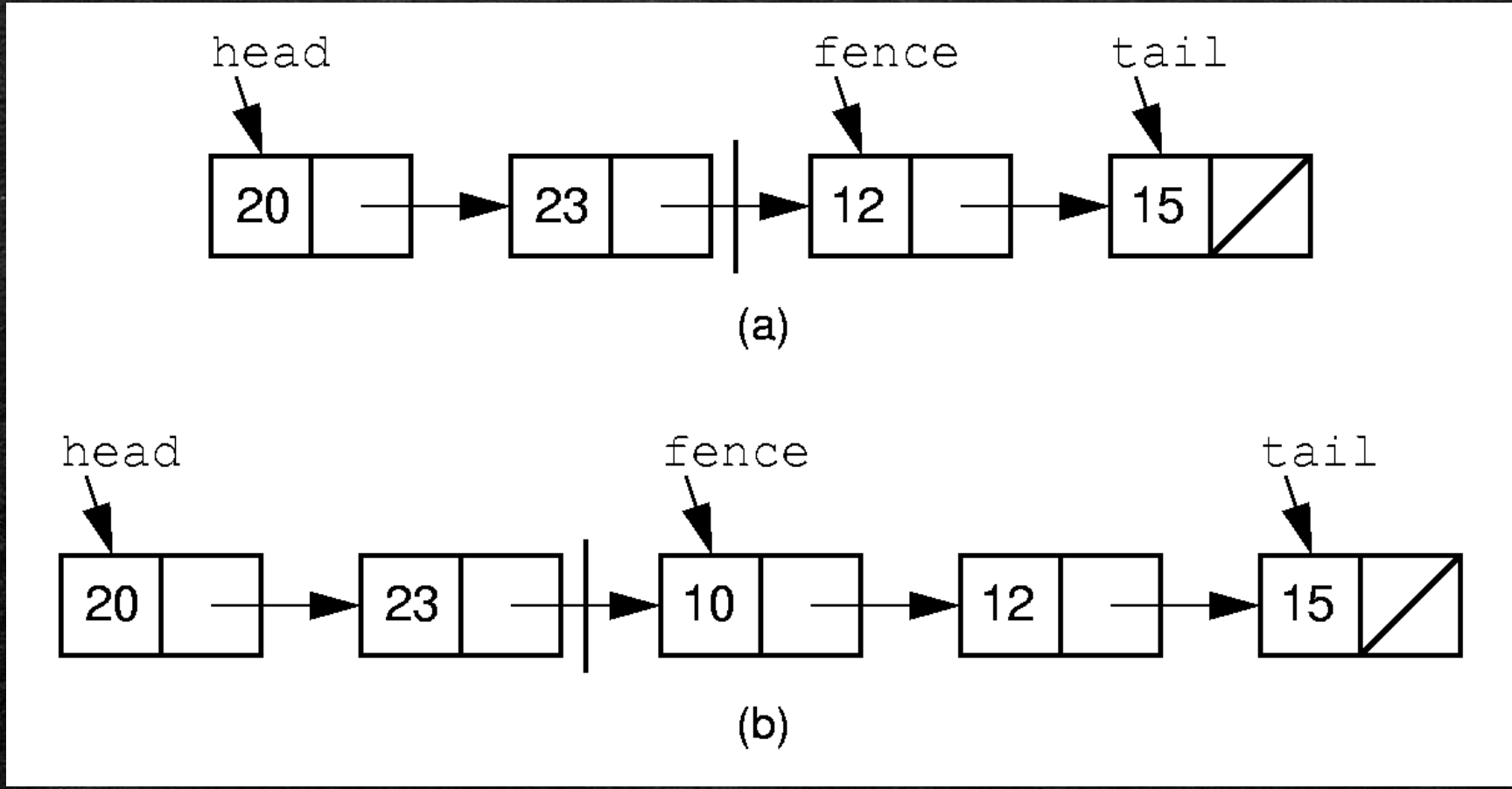


Link Class

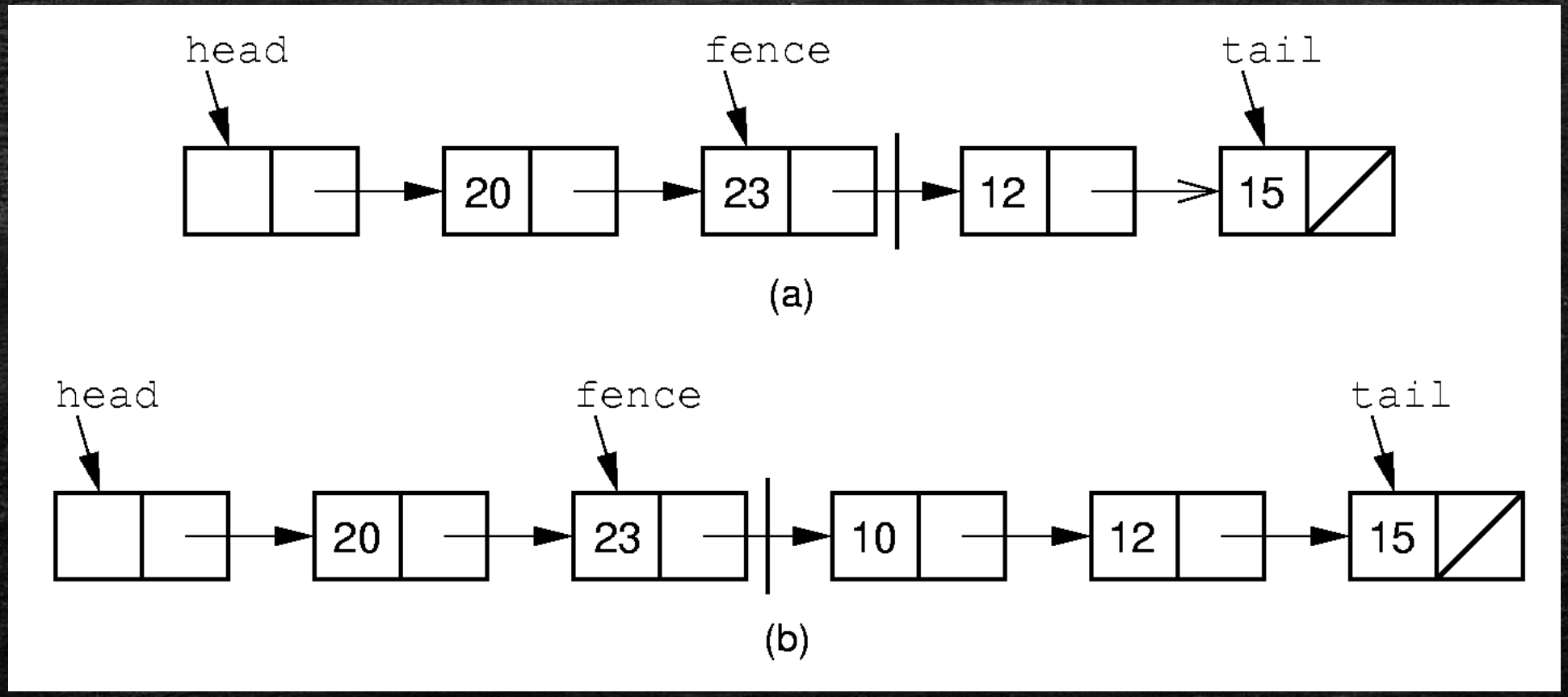
Dynamic allocation of new list elements.

```
// Singly-linked list node
template <class Elem> class Link {
public:
  Elem element; // Value for this node
  Link *next;   // Pointer to next node
  Link(const Elem& elemval,
       Link* nextval =NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval =NULL)
    { next = nextval; }
};
```

Linked List Position (1) ไม่ควรใช้



Linked List Position (2) แบบนี้ดีกว่า



Linked List Class (1)

```
/ Linked list implementation
template <class Elem> class LList:
      public List<Elem> {
private:
  Link<Elem>* head; // Point to list header
  Link<Elem>* tail; // Pointer to last Elem
  Link<Elem>* fence;// Last element on left
  int leftcnt;      // Size of left
  int rightcnt;     // Size of right
  void init() {     // Intialization routine
    fence = tail = head = new Link<Elem>;
    leftcnt = rightcnt = 0;
  }
```

Linked List Class (2)

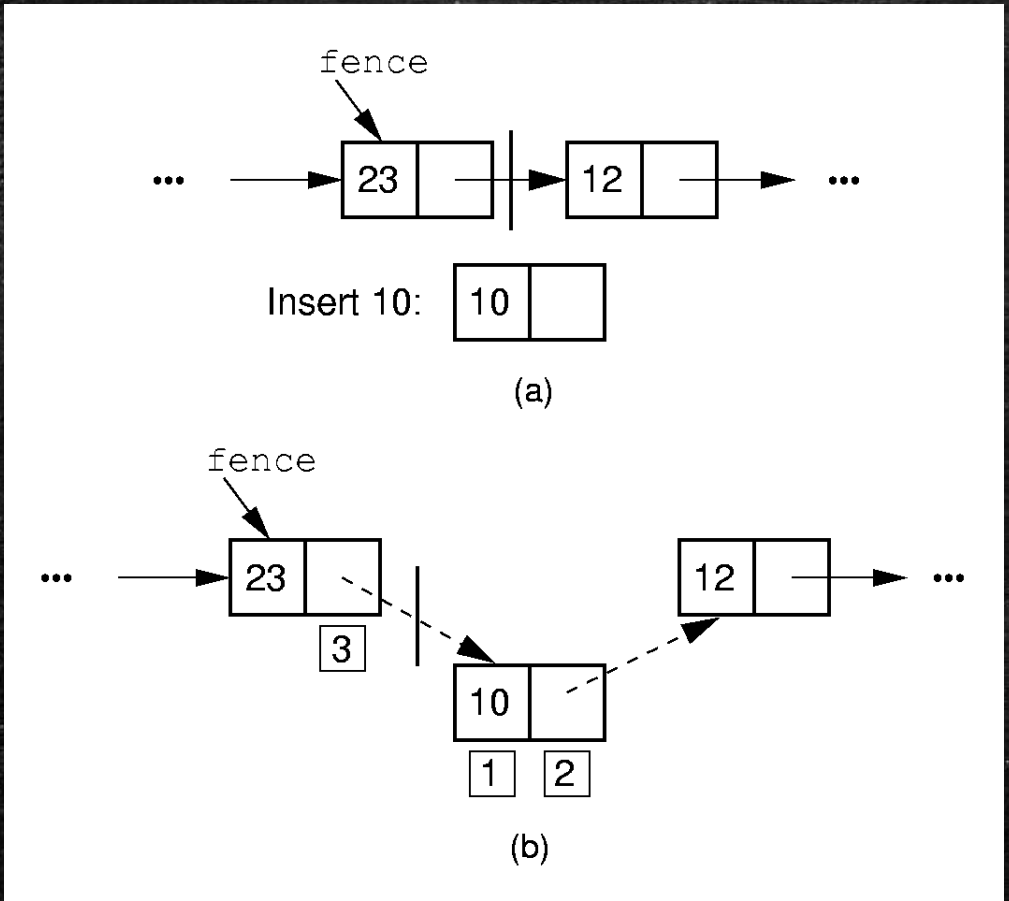
```
void removeall() {   // Return link nodes to free store
    while(head != NULL) {
      fence = head;
      head = head->next;
      delete fence;
    }
  }
public:
  LList(int size=DefaultListSize)
    { init(); }
  ~LList() { removeall(); }  // Destructor
  void clear() { removeall(); init(); }
```

```
void removeall() {head = NULL; }
```

Linked List Class (3)

```
void setStart() {
  fence = head; rightcnt += leftcnt;
  leftcnt = 0; }
void setEnd() {
  fence = tail; leftcnt += rightcnt;
  rightcnt = 0; }
void next() {
 // Don't move fence if right empty
 if (fence != tail) {
   fence = fence->next; rightcnt--;
      leftcnt++; }
}
int leftLength() const  { return leftcnt; }
int rightLength() const { return rightcnt; }
bool getValue(Elem& it) const {
  if(rightLength() == 0) return false;
  it = fence->next->element;
  return true; }
```

Insertion



1. Create new link
2. Point new to fence->next
3. Point fence to new

Insert/Append

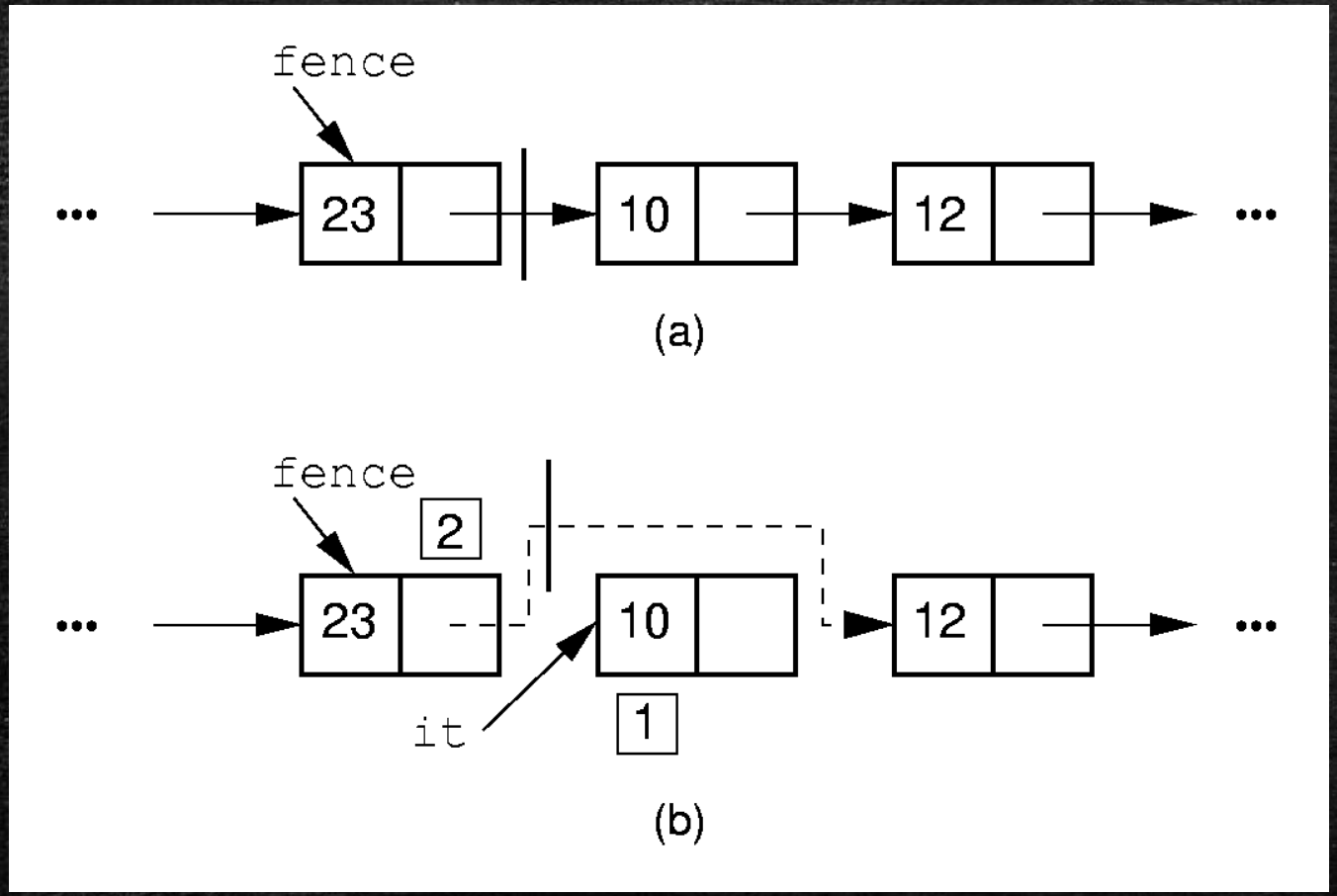
```
// Insert at front of right partition
template <class Elem>
bool LList<Elem>::insert(const Elem& item) {
  fence->next =
    new Link<Elem>(item, fence->next);
  if (tail == fence) tail = fence->next; rightcnt++;
  return true;}

//เขียนให้อ่านง่ายขึ้น
// Simpler Insert at front of right partition
template <class Elem>
bool LList<Elem>::insert(const Elem& item) {
  temp = new Link<Elem>(item,null);
  temp->next = fence->next;
  fence->next = temp;
if (tail == fence) tail = fence->next; rightcnt++;
  return true;}
```

Insert/Append

```
// Append Elem to end of the list
template <class Elem>
bool LList<Elem>::append(const Elem& item) {
  tail = tail->next =
    new Link<Elem>(item, NULL);
  rightcnt++;
  return true;}
```

Removal



1. Create temp link
2. Relink fence to temp->next

Remove

```
// Remove and return first Elem in right
// partition
template <class Elem> bool LList<Elem>::remove(Elem& it) {
  if (fence->next == NULL) return false;
  it = fence->next->element; // Remember val
  // Remember link node
  Link<Elem>* ltemp = fence->next;
  fence->next = ltemp->next; // Remove
  if (tail == ltemp) // Reset tail
    tail = fence;
  delete ltemp;      // Reclaim space
  rightcnt--;
  return true;
}
```

Prev & Next

```
// Move fence one step left;
// no change if left is empty
template <class Elem> void
LList<Elem>::prev() {
  Link<Elem>* temp = head;
  if (fence == head) return; // No prev Elem
  while (temp->next!=fence)
    temp=temp->next;
  fence = temp;
  leftcnt--;
  rightcnt++;
}

LList<Elem>::next() {
  if (curr != tail) curr = curr->next;
}
```

Setpos

```
// Set the size of left partition to pos
template <class Elem>
bool LList<Elem>::setPos(int pos) {
  if ((pos < 0) || (pos > rightcnt+leftcnt))
    return false;
  fence = head;
  for(int i=0; i<pos; i++)
    fence = fence->next;
  return true;
}
```

เปรียบเทียบประสิทธิภาพ

Array-Based Lists:

- Insertion and deletion are $\Theta(n)$.
- Prev and direct access are $\Theta(1)$.
- Array must be allocated in advance.
- No overhead if all array positions are full.

Linked Lists:

- Insertion and deletion are $\Theta(1)$.
- Prev and direct access are $\Theta(n)$.
- Space grows with number of elements.
- Every element requires overhead.

เปรียบเทียบพื้นที่จัดเก็บ

"Break-even" point:

$$DE = n(P + E);$$

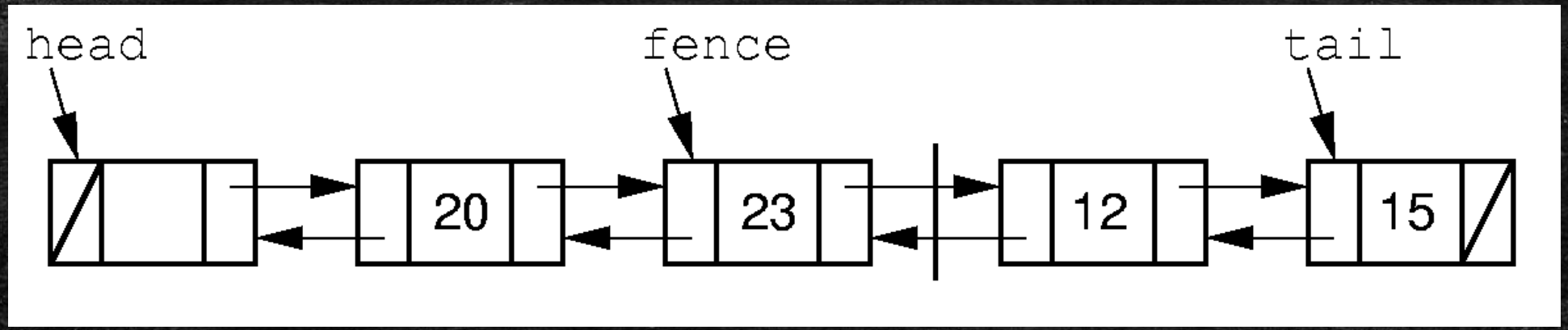
$$n = \frac{DE}{P + E}$$

E: Space for data value.

P: Space for pointer.

D: Number of elements in array.

Doubly Linked Lists

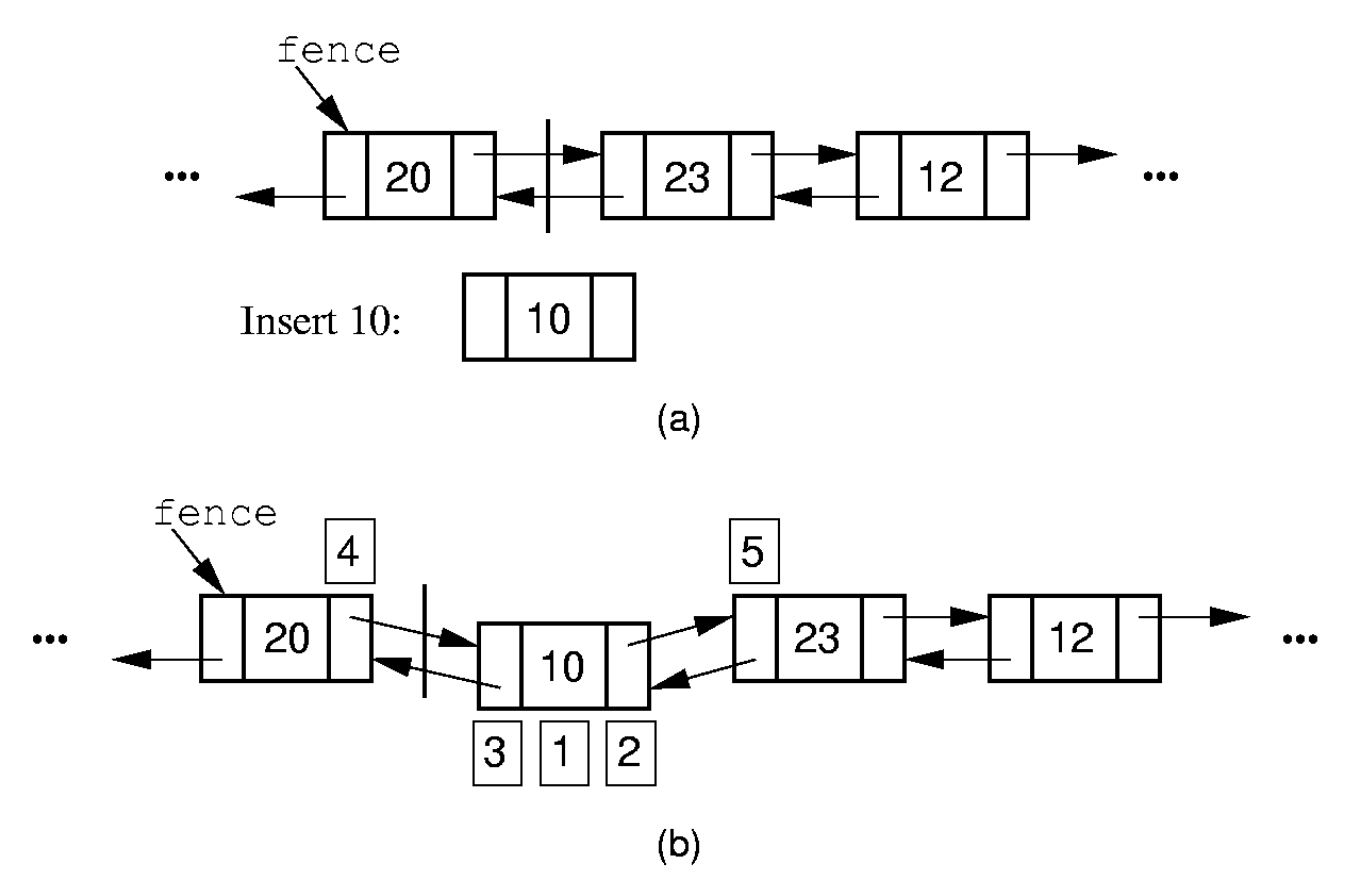


Doubly Linked Lists

Simplify insertion and deletion: Add a **prev** pointer.

```
// Doubly-linked list link node
template <class Elem> class Link {
public:
  Elem element;  // Value for this node
  Link *next;    // Pointer to next node
  Link *prev;    // Pointer to previous node
  Link(const Elem& e, Link* prevp =NULL,
                      Link* nextp =NULL)
    { element=e;  prev=prevp;  next=nextp; }
  Link(Link* prevp =NULL, Link* nextp =NULL)
    { prev = prevp;  next = nextp; }
};
```

Doubly Linked Insert



1. Create new
2. Link fence->next to new
3. Link new to fence
4. Link fence to new
5. Link new to fence->next

Doubly Linked Insert

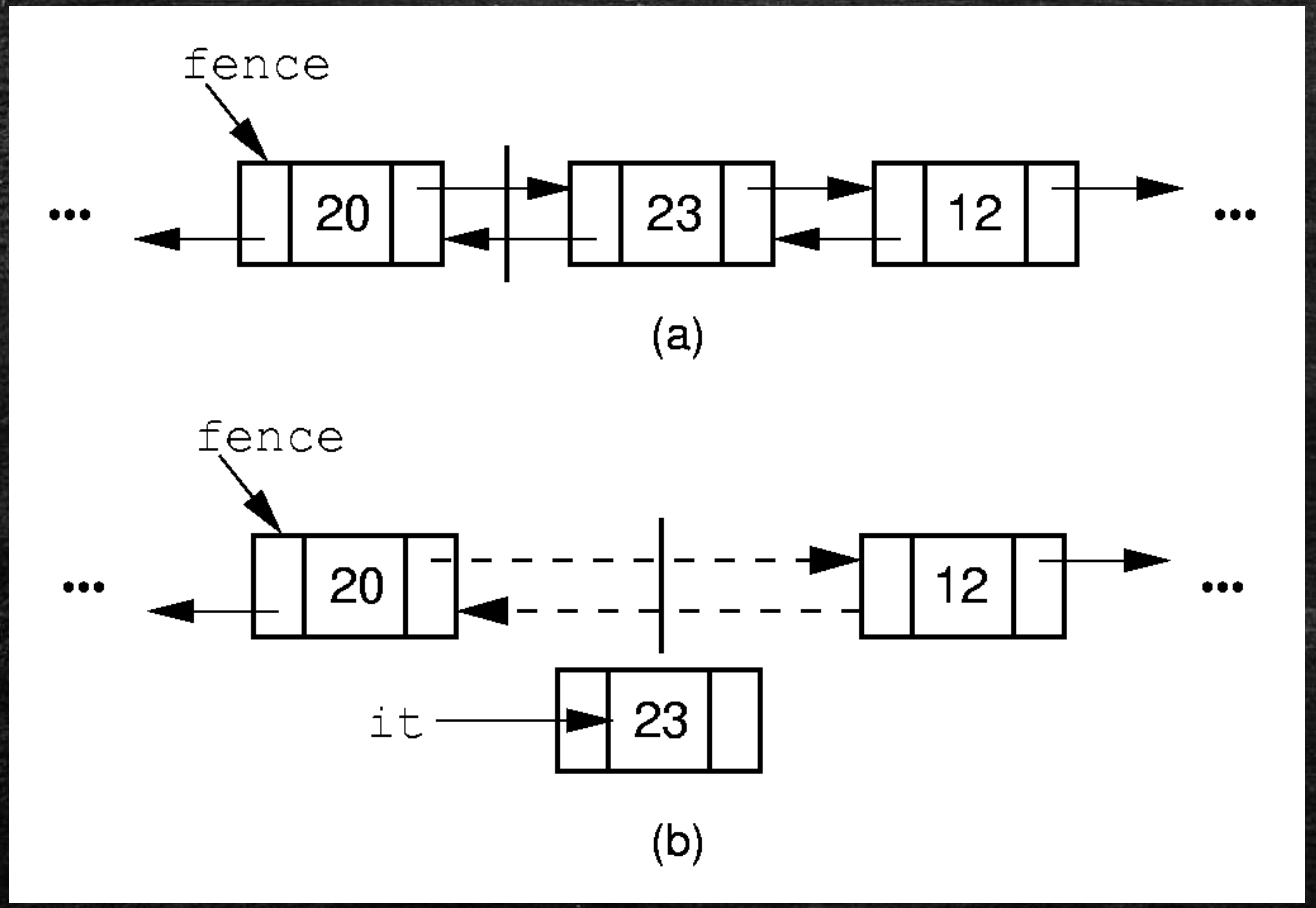
```
// Insert at front of right partition
template <class Elem>
bool LList<Elem>::insert(const Elem& item) {
  fence->next =
   new Link<Elem>(item, fence, fence->next);
  if (fence->next->next != NULL)
    fence->next->next->prev = fence->next;
  if (tail == fence)   // Appending new Elem
    tail = fence->next; //   so set tail
  rightcnt++;           // Added to right
  return true;
}
```

Doubly Linked Insert

```
//เขียนให้อ่านง่ายขึ้น
// Simpler Insert at front of right partition
template <class Elem>
bool LList<Elem>::insert(const Elem& item) {
  temp = new Link<Elem>(item, null, null);
  fence->next->prev=temp;
  temp->prev=fence;
  temp->next=fence->next;
  fence->next=temp;

  if (fence->next->next != NULL)
    fence->next->next->prev = fence->next;
  if (tail == fence)    // Appending new Elem
    tail = fence->next; //   so set tail
  rightcnt++;           // Added to right
  return true;
}
```

Doubly Linked Remove



Doubly Linked Remove

```
// Remove, return first Elem in right part
template <class Elem>
bool LList<Elem>::remove(Elem& it) {
  if (fence->next == NULL) return false;
  it = fence->next->element;
  Link<Elem>* ltemp = fence->next;
  if (ltemp->next != NULL)
    ltemp->next->prev = fence;
  else tail = fence;          // Reset tail
  fence->next = ltemp->next; // Remove
  delete ltemp;     // Reclaim space
  rightcnt--;       // Removed from right
  return true;
}
```

โบนัส 2 คะแนน

- ให้ทำ score board ของเกมขนาดไม่เกินสิบอันดับ
- โครงสร้างข้อมูลมี ชื่อ และ คะแนน
- เรียงลำดับคะแนนอัตโนมัติ
- เมื่อมี new score ให้นำไปแทรกในตำแหน่งที่ถูกต้องโดยตัดคนที่อยู่อันดับสุดท้ายออกจากลิสต์
- กำหนดให้ใช้ doubly linked list

ตัวอย่าง กำหนดให้ลิสใหญ่ไม่เกิน 4

- ```
  Ryu       100
  Ken       98
  Chunli    95
  Sagat     94
  ```
- ```
  Vaga      got new score = 97
  ```
- ```
  Ryu       100
  Ken       98
  Vega      97
  Chunli    95
  ```
- ```
  Sagat     was remove from the score board
  ```

จาก break-even point ของ singly linked list จงคำนวณหา break-even point ของ doubly linked list เมื่อเปลี่ยนสถาปัตยกรรมจาก 32 เป็น 64 bits

"Break-even" point:

$$DE = n(P + E);$$

$$n = \frac{DE}{P + E}$$

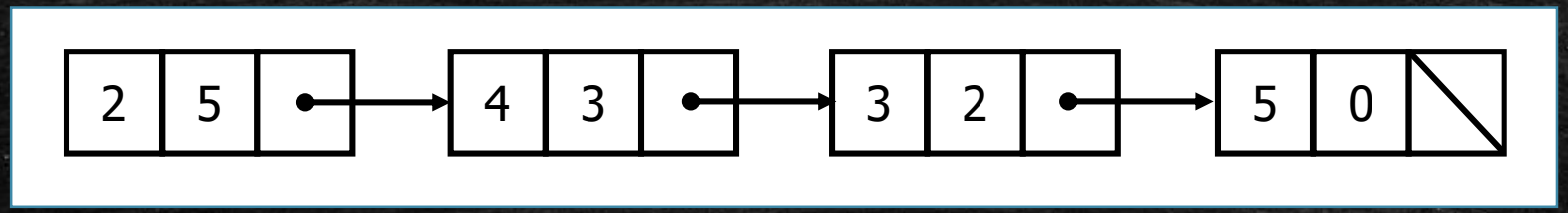
E: Space for data value.

P: Space for pointer.

D: Number of elements in array.

การบ้าน 5 คะแนน

- จงใช้ linked list สำหรับสร้าง Polynomial (พหุนาม)
- Eg. $2x^5 + 4x^3 + 3x^2 + 5$



- จงเขียน method add และ multiply
  พร้อมวิเคราะห์เวลาที่ใช้ด้วย Big O notation