Chapter 8

STRUCTURES IN C

Introduction

- Structures
  - Collections of related variables (aggregates) under one name
    - Can contain variables of different data types
  - Commonly used to define records to be stored in files
  - Combined with pointers, can create linked lists, stacks, queues, and trees

Structure Definitions

- Example

```
struct card {
    char *face;
    char *suit;
 };
```

  - **`struct`** introduces the definition for structure card
  - **`card`** is the structure name and is used to declare variables of the structure type
  - **`card`** contains two members of type **`char *`**
    - These members are **`face`** and **`suit`**

Structure Definitions

- `struct` information
  - A `struct` cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type
  - A structure definition does not reserve space in memory
    - Instead creates a new data type used to declare structure variables
- Declarations
  - Declared like other variables:

```
struct card oneCard, deck[ 52 ], *cPtr;
```

  - Can use a comma separated list:

```
struct card {
    char *face;
    char *suit;
} oneCard, deck[ 52 ], *cPtr;
```

Structure Definitions

- Valid Operations
  - Assigning a structure to a structure of the same type
  - Taking the address (**`&`**) of a structure
  - Accessing the members of a structure
  - Using the **`sizeof`** operator to determine the size of a structure

Initializing Structures

- Initializer lists
  - Example:

```
struct card oneCard = { "Three", "Hearts" };
```

- Assignment statements
  - Example:

```
struct card threeHearts = oneCard;
```

  - Could also declare and initialize `threeHearts` as follows:

```
struct card threeHearts;
threeHearts.face = "Three";
threeHearts.suit = "Hearts";
```

Array of structures

```
struct time
{
  int hour;
  int minutes;
  int seconds;
} testTimes[] =
{ { 11, 59, 59
}, { 12, 0, 0 },
{ 1, 29, 59 },
{ 23, 59, 59 },
{ 19, 12, 27 }};
```

- Give the size of the array in []
- Or we could use sizeof
  - Compile-time unary operator
  - Use in #define

```
#define NTIMES (sizeof testTimes
/ sizeof(struct time))
```

  - Don’t assume that the size of a structure is the sum of the sizes of its members.

SHANGHAI JIAO TONG UNIVERSITY | ParisTech INSTITUT DES SCIENCES ET TECHNOLOGIES PARIS INSTITUTE OF TECHNOLOGY

Accessing Members of Structures

- Accessing structure members
  - Dot operator (`.`) used with structure variables

```
struct card myCard;
printf( "%s", myCard.suit );
```

  - Arrow operator (`->`) used with pointers to structure variables

Operator ->

- Structure could be pointed
  - Assign the pointer to structure

```
struct card *myCardPtr = &myCard;
```

  - Access structure variable with (.)

```
(*myCardPtr).suit
```

  - **`myCardPtr->suit`** is equivalent to

```
( *myCardPtr ).suit
```

Precedence of (.) and ->

- They are at the top level (should be firstly considered)
  - Before * and ++/--
- Example:
  - ++p->len, (++p)->len, p++->len
  - *p->str, *p->str++, (*p->str)++, *p++->str
  - @A@

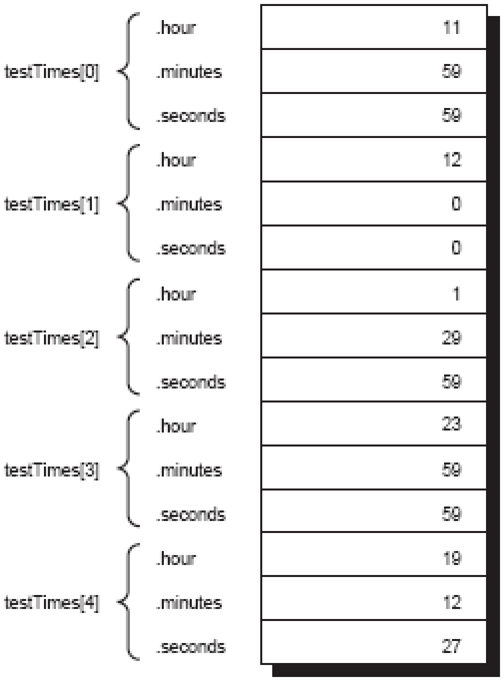
```
Struct {
   int len;
   char *str;
} *p;
```

Using Structures With Functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- Example: Time structure
  - struct time{ …}
  - struct time timeUpdate(struct time now)

Using Structures With Functions

- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
  - `struct time *getTimebyIndex(int index, struct time *tab, int n)`
- Working with array of structures
  - `struct time testTimes[5]`



- A list of sth
  - Examples: cards

typedef

- **typedef**
  - Creates synonyms (aliases) for previously defined data types
  - Use **typedef** to create shorter type names
    - `typedef int Length`
    - `typedef char *String`

`typedef`

- Example:

```
typedef struct Card *CardPtr;
```

- Defines a new type name **`CardPtr`** as a synonym for type **`struct Card *`**
  - **`-> everywhere`**
- **`typedef`** does not create a new data type
  - Only creates an alias

Example: High-Performance Card-shuffling and Dealing Simulation

- Pseudocode:
  - Create an array of card structures
  - Put cards in the deck
  - Shuffle the deck
  - Deal the cards

```c
/* Fig. 10.3: fig10_03.c
   The card shuffling and dealing program using structures */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct card {
   const char *face;
   const char *suit;
};

typedef struct card Card;

void fillDeck( Card * const, const char *[],
               const char *[] );
void shuffle( Card * const );
void deal( const Card * const );

int main()
{
   Card deck[ 52 ];
   const char *face[] = { "Ace", "Deuce", "Three",
                          "Four", "Five",
                          "Six", "Seven", "Eight",
                          "Nine", "Ten",
                          "Jack", "Queen", "King"};
   const char *suit[] = { "Hearts", "Diamonds",
                          "Clubs", "Spades"};

   srand( time( NULL ) );
```

1. Load headers

1.1 Define `struct`

1.2 Function prototypes

1.3 Initialize `deck[]` and `face[]`

1.4 Initialize `suit[]`

```c

   fillDeck( deck, face, suit );
   shuffle( deck );
   deal( deck );
   return 0;
}

void fillDeck( Card * const wDeck, const char * wFace[],
               const char * wSuit[] )
{
   int i;

   for ( i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = wFace[ i % 13 ];
      wDeck[ i ].suit = wSuit[ i / 13 ];
   }
}

void shuffle( Card * const wDeck )
{
   int i, j;
   Card temp;

   for ( i = 0; i <= 51; i++ ) {
      j = rand() % 52;
      temp = wDeck[ i ];
      wDeck[ i ] = wDeck[ j ];
      wDeck[ j ] = temp;
   }
}
```

Put all 52 cards in the deck. **face** and **suit** determined by remainder (modulus).

Select random number between **0** and **51**. Swap element **i** with that element.

2. fillDeck

2.1 shuffle

3. Function definitions

```c

void deal( const Card * const wDeck )
{
   int i;

   for ( i = 0; i <= 51; i++ )
      printf( "%5s of %-8s%c", wDeck[ i ].face,
              wDeck[ i ].suit,
              ( i + 1 ) % 2 ? '\t' : '\n' );
}
```

Cycle through array and print out data.

```
Eight of Diamonds        Ace of Hearts
Eight of Clubs          Five of Spades
Seven of Hearts        Deuce of Diamonds
  Ace of Clubs           Ten of Diamonds
Deuce of Spades          Six of Diamonds
Seven of Spades        Deuce of Clubs
 Jack of Clubs           Ten of Spades
 King of Hearts         Jack of Diamonds
Three of Hearts        Three of Diamonds
Three of Clubs          Nine of Clubs
  Ten of Hearts        Deuce of Hearts
  Ten of Clubs         Seven of Diamonds
  Six of Clubs         Queen of Spades
  Six of Hearts        Three of Spades
 Nine of Diamonds        Ace of Diamonds
 Jack of Spades         Five of Clubs
 King of Diamonds      Seven of Clubs
 Nine of Spades         Four of Hearts
  Six of Spades        Eight of Spades
Queen of Diamonds       Five of Diamonds
  Ace of Spades         Nine of Hearts
 King of Clubs          Five of Hearts
 King of Spades         Four of Diamonds
Queen of Hearts        Eight of Hearts
 Four of Spades         Jack of Hearts
 Four of Clubs         Queen of Clubs
```

Program Output

Unions

- **`union`**
  - Memory that contains a variety of objects at different times
  - Only contains one data member at a time
  - Members of a **`union`** share space
  - Conserves storage
  - Only the last data member defined can be accessed
- **`union`** declarations
  - Same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```

Unions

- Valid `union` operations
  - Assignment to `union` of same type: `=`
  - Taking address: `&`
  - Accessing union members: `.`
  - Accessing members using pointers: `->`
- Access to members of a union
  - Once a member at a time

```c
/* Fig. 10.5: fig10_05.c
   An example of a union */
#include <stdio.h>

union number {
   int x;
   double y;
};

int main()
{
   union number value;

   value.x = 100;
   printf( "%s\n%s\n%s%d\n%s%f\n\n",
          "Put a value in the integer member",
          "and print both members.",
          "int:   ", value.x,
          "double:\n", value.y );

   value.y = 100.0;
   printf( "%s\n%s\n%s%d\n%s%f\n",
          "Put a value in the floating member",
          "and print both members.",
          "int:   ", value.x,
          "double:\n", value.y );
   return 0;
}
```

1. Define union

1.1 Initialize variables

2. Set variables

3. Print

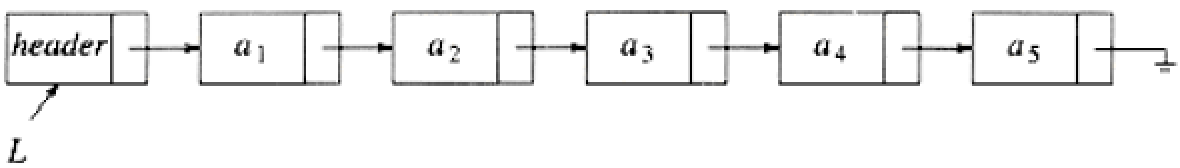
```
Put a value in the integer member
and print both members.
int:   100
double:
-9255959211743313600000000000000000000000000000000000000000000.00000

Put a value in the floating member
and print both members.
int:   0
double:
100.000000
```

Program Output

Self-referential Structure

- Self-referential structure is widely used to generate data structure (realization of ADT)
- Use of pointers to structures to create the links between objects
  - Use pointers to the same type of structure
  - Use pointers as members of the sturcture
- `typedef` and `struct`

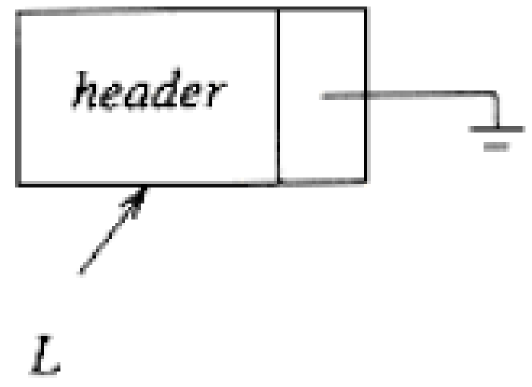
Example: Linked List




```
typedef struct node *node_ptr;
struct node
{
  int element;
  node_ptr next;
};
typedef node_ptr List;
List thislist=malloc(sizeof(struct node));
```

Use ->

```
int is_empty( LIST L )
{
  return( L->next == NULL );
}
```



Function malloc

- void *malloc(size_t size)
  - tab =(int *) malloc (n *sizeof (int)) ;

- free(void*);
  - Do not forget!

Write a cons function

- A cons function add a node at the beginning of the list, with a given element.

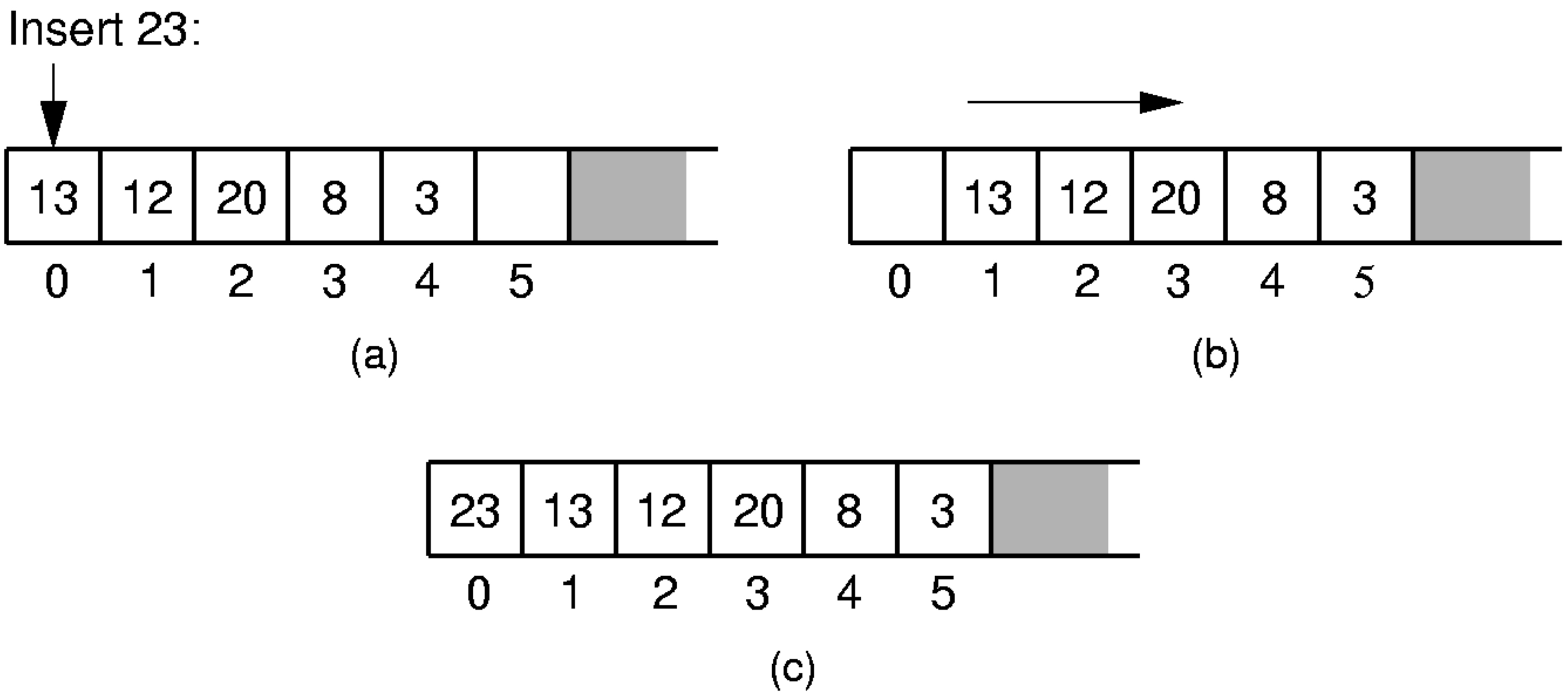
```
void cons(int e, List l)
{




}
```

But also: freelist(), printlist(), init().

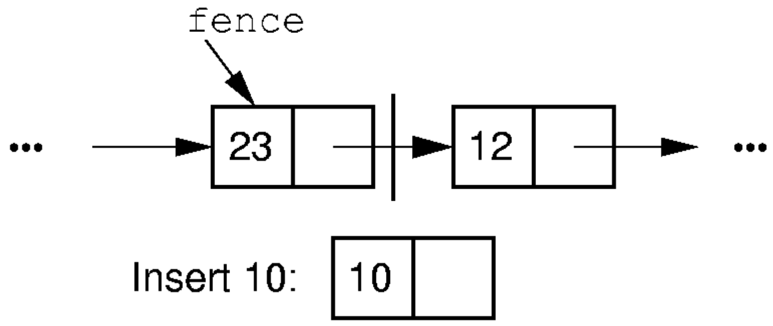
Lists

- A list is a finite, ordered sequence of data items.
- Important concept: List elements have a position.
- Notation: $<a_0, a_1, \ldots, a_{n-1}>$
- Size of a list:
  - n
  - Null list: size 0

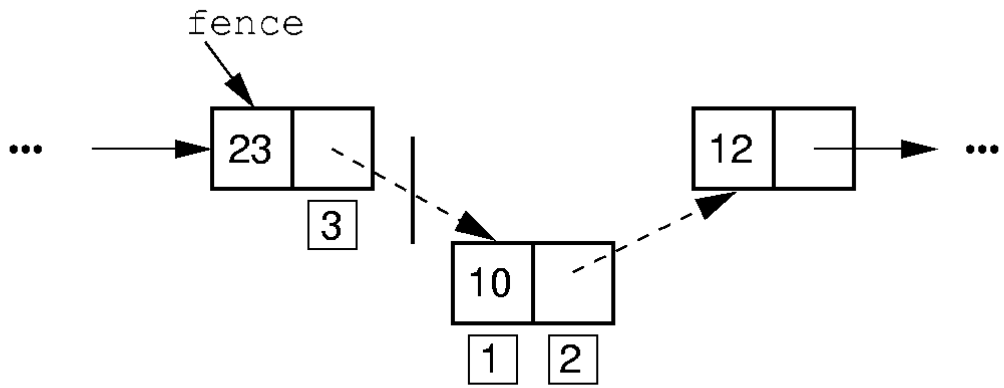
Array-Based List Insert



Linked List Insertion



(a)



(b)

Comparisons

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.

What operations should we implement with a List?

- clear
- print
- find
- insert
- getValue
- delete

List Implementation Concepts

Our list implementation will support the concept of a current position.

We will do this by defining the list in terms of left and right partitions.

- Either or both partitions may be empty.

Partitions are separated by the fence.

<20, 23 | 12, 15>

List ADT

```
void clear(List);
int insert(List, Elem);
int append(List, Elem);
int remove(List, Elem*);
void setStart(List);
void setEnd(List);
void prev(List);
void next(List);
```

List ADT (cont)

```
  int leftLength(List);
  int rightLength(List);
  int setPos(List,int);
  int getValue(List,Elem*);
  void print(List);
};
```

List Find Function

```
Pseudocode
// Return true iff K is in list
int find(List L, int K) {
  int it;
  for (setStart(L); getValue(L, &it);
            next(L))
    if (K == it) return 0;
    return 1;
}
```

List ADT Examples

List: <12 | 32, 15>

```
insert(myList,99);
```

Result: <12 | 99, 32, 15>

Iterate through the whole list:

```
for (setStart(myList); getValue(myList,&it);
     next(myList))
  DoSomething(it);
```

List data structure

- Array-based list
  - Use an array with maxSize to store elements
  - Use listSize to specify the length of the list
  - Use fence to control the current position
  - "List" is the name of the array (pointer)
- Linked list
  - Each node is a structure with self-referential
  - MaxSize and listSize is not necessary
  - head, tail, fence are used as node pointer
  - "List" is the node pointer head

Array-Based List

- All operation could be realized using array
- Need an estimation of maximum size
  - Limitation for unknown size list
  - Space consumming
- getValue in *O(1)*
- find in *O(n)*
- Insertion and deletion are expensive
  - Move all the element on the “right”
- Worst case *O(n)*

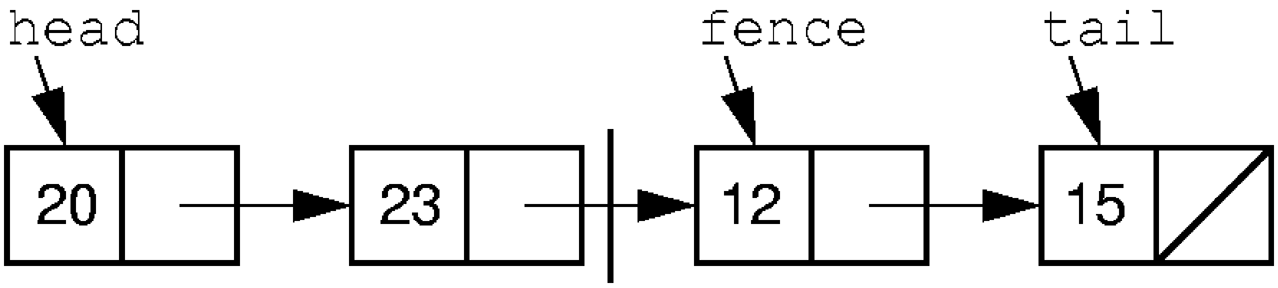
Linked List

Definition of an element of list

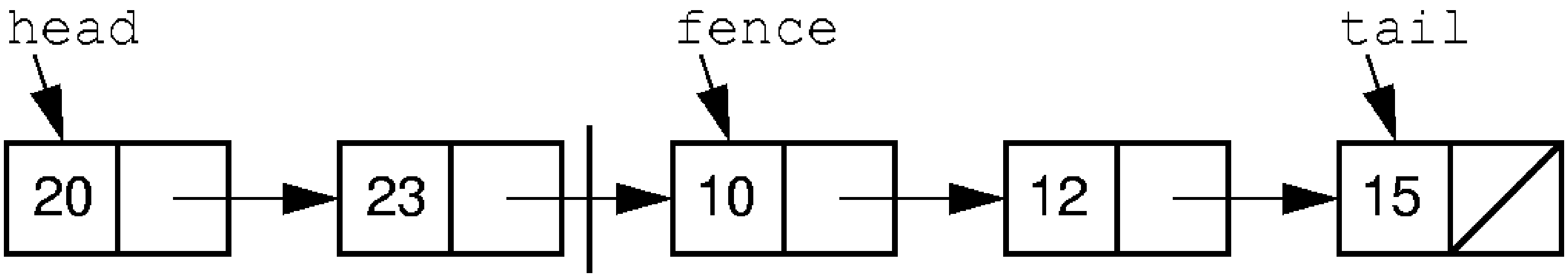
```
#define TRUE 1
#define FALSE 0
typedef int BOOL ;
typedef int element_type;
typedef struct STRUCTNODE* node_ptr;

// Singly-linked list node

typedef struct STRUCTNODE
{
  element_type element;
  struct STRUCTNODE* next;
}node;
```

Dynamic allocation of new list elements using `malloc()`

Linked List Position (1)

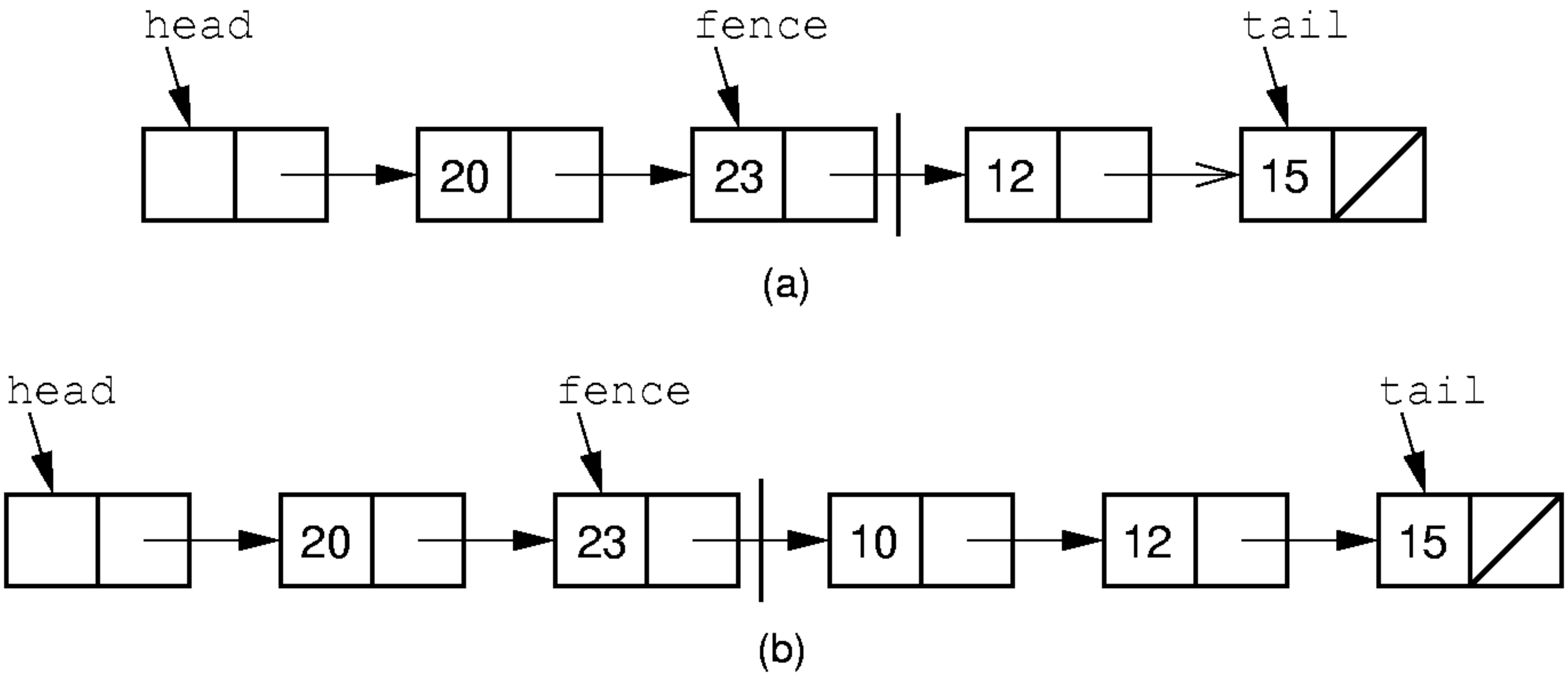


(a)



(b)

Linked List Position (2)



Linked List (1)

```
// Linked list implementation
typedef struct LINKLIST *LList{
node_ptr head; // Point to list header
node_ptr tail; // Pointer to last
node_ptr fence;// Last element on left
int leftcnt;        // Size of left
int rightcnt;       // Size of right
};
```

Linked List (2)

```
void init(LList listLink) {      // Intialization
    node_ptr headNode = (node_ptr)malloc(sizeof(node));
    headNode->element = 0;
    headNode->next = NULL;

    listLink->head = headNode;
    listLink->fence = listLink->tail = listLink->head;
    listLink->leftcnt = listLink-> rightcnt = 0;
}

void removeall(LList listLink) {//free store
    while(listLink->head != NULL) {
      listLink->fence = listLink->head;
      listLink->head = listLink->head->next;
      free(listLink->fence);
    }
  }

void clear(LList listLink) {      removeall(listLink);
   init(listLink);     }
```

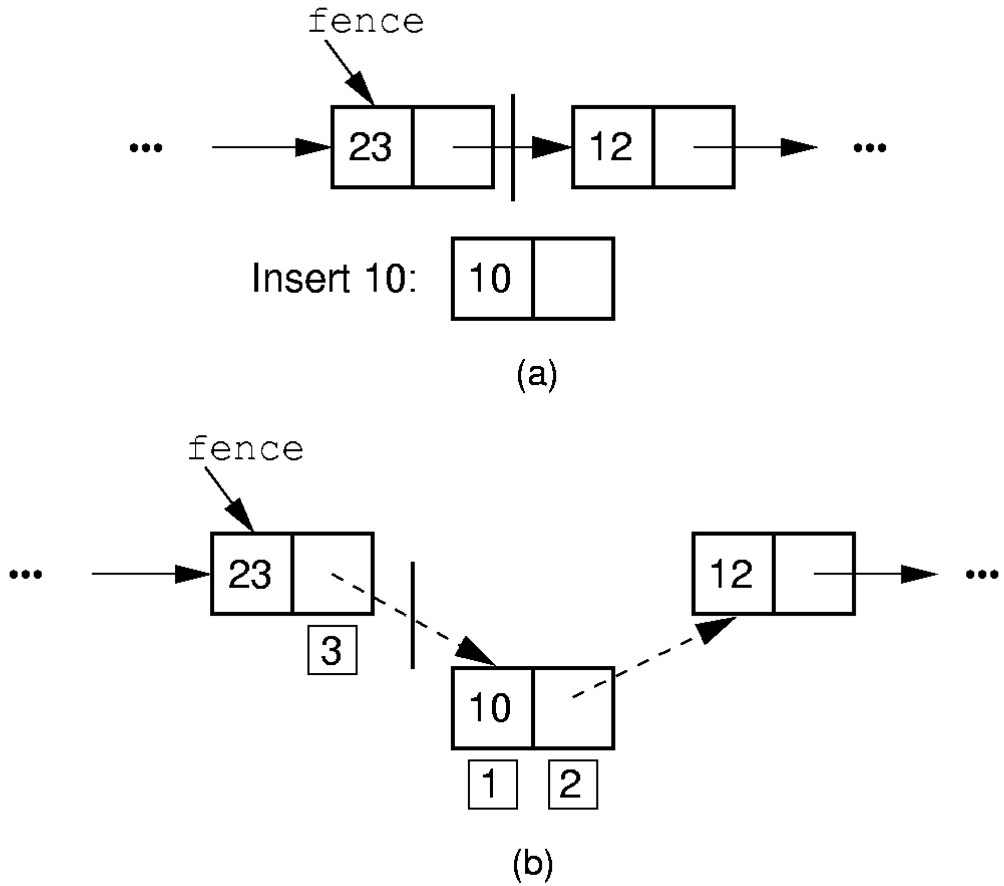
Linked List (3)

```
void setStart(LList listLink) {
  listLink->fence = listLink-> head;
  listLink->rightcnt += listLink->leftcnt;
  listLink->leftcnt = 0;
  }

void setEnd(LList listLink) {
  listLink->fence = listLink->tail;
  listLink->leftcnt += listLink->rightcnt;
  listLink->rightcnt = 0; }

void next(LList listLink) {
  // Don't move fence if right empty
  if (listLink->fence != listLink->tail) {
  listLink->fence = listLink->fence->next;
  listLink->rightcnt--;
  listLink->leftcnt++;
  }
}
```

Linked List (4)

```
int leftLength(LList listLink)
  { return listLink->leftcnt; }

int rightLength(LList listLink)
  { return listLink-> rightcnt; }

BOOL getValue(LList listLink, element_type* it){
  if(rightLength(listLink) == 0) return FALSE;
  *it = listLink->fence->next->element;
  return TRUE;
}
```

Insertion



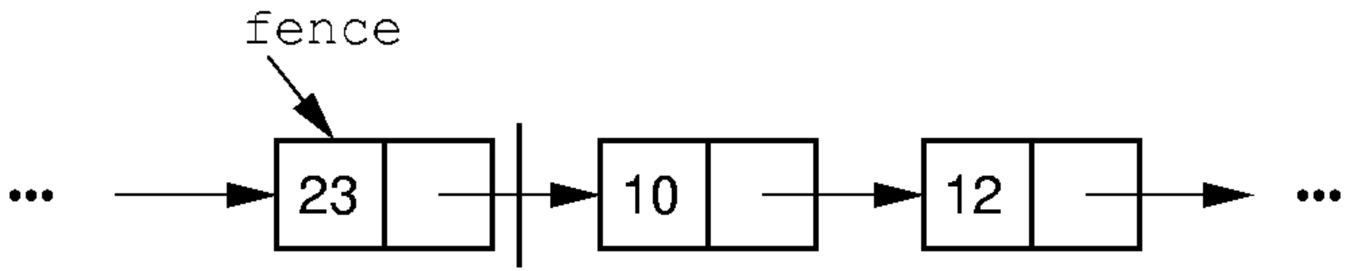
Insert

```
// Insert at front of right partition
BOOL insert(LList listLink,  element_type item)
{
  // new a node
  node_ptr tmpNode = (node_ptr)malloc(sizeof(node));
  tmpNode->element = item;

  tmpNode->next = listLink->fence->next;
  listLink->fence->next = tmpNode;

  if (listLink->tail == listLink->fence)
    listLink->tail = listLink->fence->next;
  listLink->rightcnt++;
  return TRUE;

 }
```

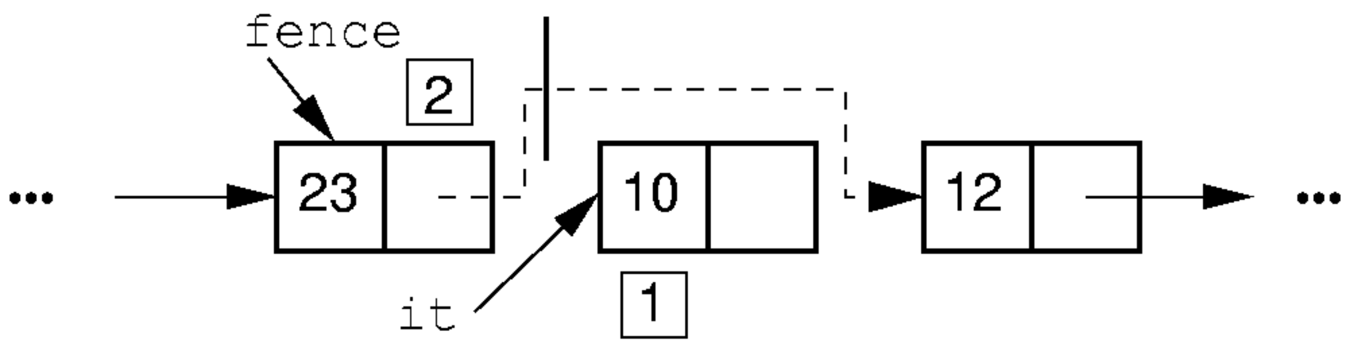
Append

```
// Append Elem to end of the list
BOOL append(LList listLink,   element_type item) {
  // new a node
  node_ptr tmpNode = (node_ptr)malloc(sizeof(node));
  tmpNode->element = item;
  tmpNode->next = NULL;

   listLink->tail->next = tmpNode;
   listLink->tail = listLink->tail->next;

  listLink->rightcnt++;

  return TRUE;

}
```

Remove



(a)



(b)

Remove

```
// Remove and return first Elem in right
// partition
BOOL Remove(LList listLink,   element_type* it)
{
  if (listLink->fence->next == NULL) return FALSE;

*it = listLink->fence->next->element; //Remember val
  // Remember link node
  node_ptr ltemp = listLink->fence->next;
  listLink->fence->next = ltemp->next; // Remove

  if (listLink->tail == ltemp) // Reset tail
    listLink->tail = listLink->fence;

  free(ltemp);       // Reclaim space
   listLink->rightcnt--;
  return TRUE;
}
```

Prev

```
// Move fence one step left;
// no change if left is empty
void prev(LList listLink) {

  node_ptr temp = listLink->head;
  if (listLink->fence == listLink->head) return; //
  No prev Elem
  while (temp->next != listLink->fence)
    temp=temp->next;
  listLink->fence = temp;
  listLink->leftcnt--;
  listLink->rightcnt++;
}
```

Setpos

```
// Set the size of left partition to pos
BOOL setPos(LList listLink, int pos) {
  if ((pos < 0) || (pos > listLink->rightcnt+listLink->leftcnt))
    return FALSE;
  listLink->fence = listLink->head;
  int i;
  for( i=0; i<pos; i++)
     listLink->fence =  listLink->fence->next;
  return TRUE;
}
```

Print （problem）

```
// print the list
void print(LList listLink) const{
   node_ptr temp=listLink->head;
   printf("[");
   while (temp!= NULL){
        print("%d ", temp->next->element);
        if (temp==listLink->fence){
              printf("| ");
        }
        temp=temp->next;
   }
   printf("]");

}
```

print

```
// print the list
void print(LList listLink) const{
   node_ptr temp=listLink->head;
   printf("[");
   temp=temp->next;
   while (temp!= NULL){
       printf("%d ", temp->element);
       if (temp==fence){
             printf("| ");
       }
       temp=temp->next;
   }
   printf("]");

}
```

Ex 2

- Given a singly linked list with head, which of the following statement signifies the list is empty

A .head==NULL   B .head->next==NULL
C. head->next==head   D. head!=NULL