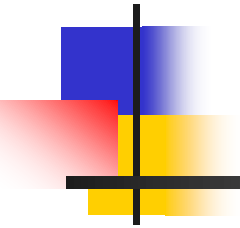
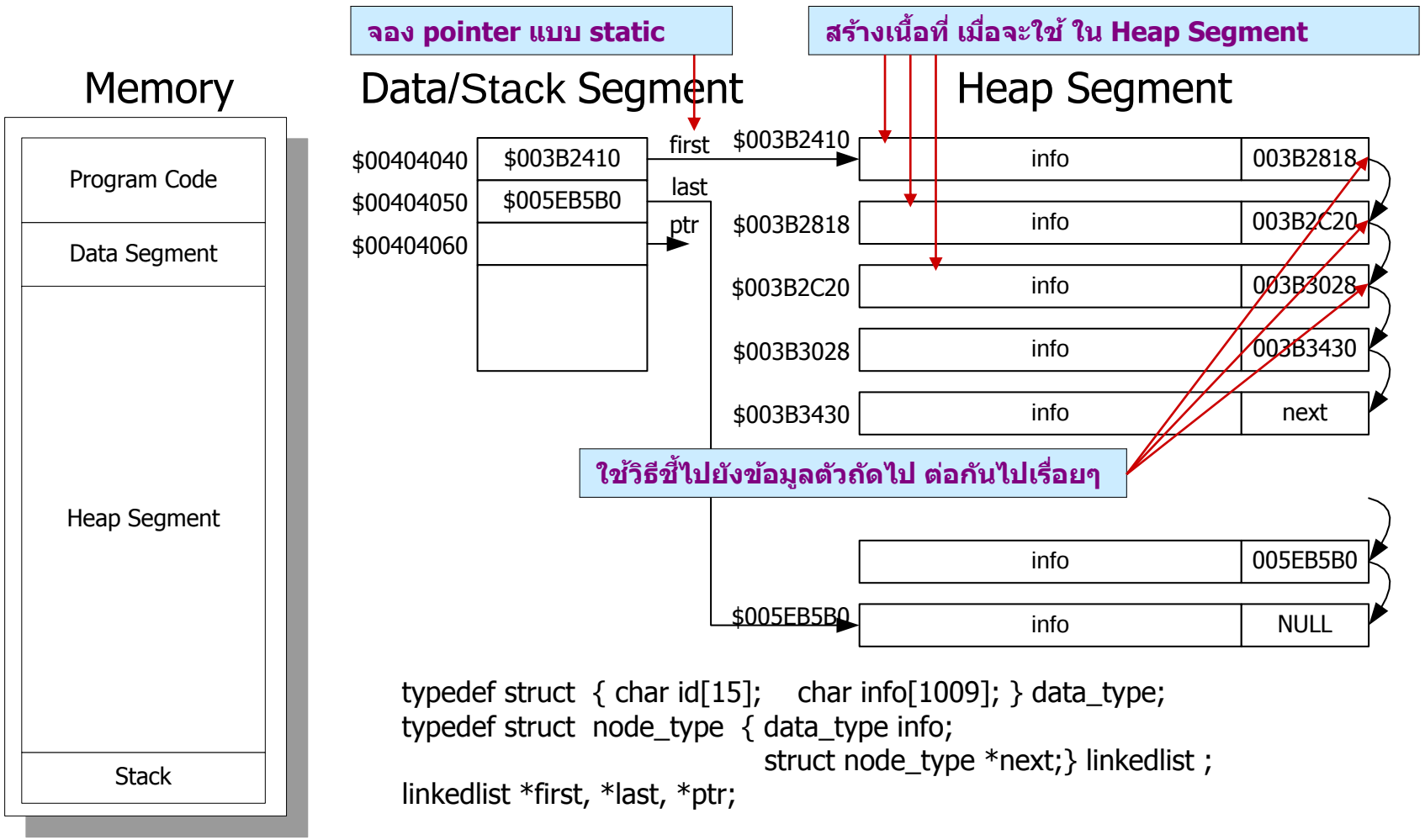


Chapter 03

Linked List

Pointer & Linked List

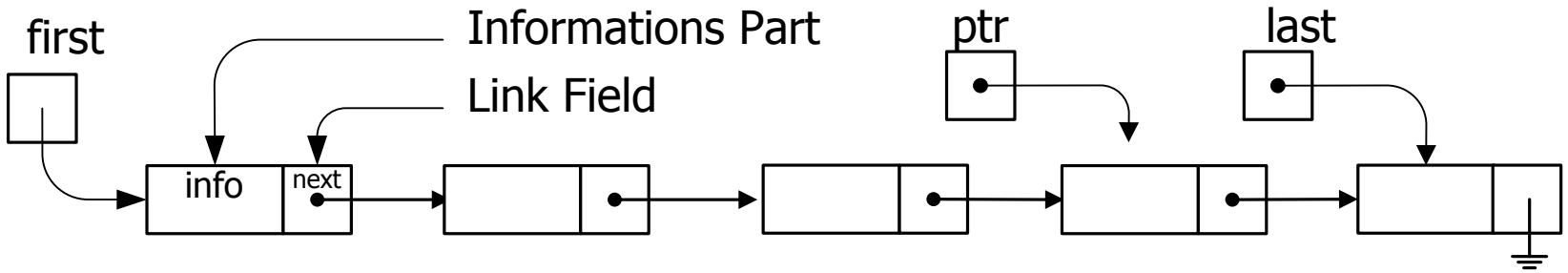
- โครงสร้างข้อมูลแบบ **linked list** จะต้องมี **pointer** ไว้ชี้ไปยังข้อมูลตัวที่อยู่ถัดไป
  - จองตัวแปรพอยต์เตอร์ไว้ สำหรับเก็บข้อมูลตัวแรก และ ตัวสุดท้าย
  - สร้างข้อมูลใน **Heap (Dynamic)** ทีละตัว แล้วใช้พอยต์เตอร์ชี้ต่อๆ กัน




```
typedef struct  { char id[15];    char info[1009]; } data_type;
typedef struct  node_type  { data_type info;
                             struct node_type *next;} linkedlist ;
linkedlist *first, *last, *ptr;
```

Linked Lists

- **linked list คือโครงสร้างที่มีการใช้ pointer ชี้ไปยังข้อมูลตัวที่อยู่ถัดไป**
  - **ต้องมี pointer สำหรับชี้ไปยังข้อมูลตัวแรก(first)**
  - **สร้างข้อมูลใน Heap (Dynamic) ทีละตัว แล้วใช้พอยต์เตอร์ชี้ต่อๆ กัน**




```
typedef  struct   {  long  id;
                     char  name[30];
                     double  gpa; } data_type; //กำหนดชื่อโครงสร้าง
data_type  item;  //จองตัวแปรใช้งาน
```

Self Reference เพื่อให้รู้จักตัวเอง

```
typedef  struct  node_type { data_type  info ;
                             struct  node_type  *next;} linkedlist;
linkedlist  *first=NULL, *last=NULL, *ptr;   //จองตัวแปร pointer
```

ptr ใช้เป็น pointer ชั่วคราวสำหรับชี้ไปยังตัวที่สนใจ

first , last ใช้เป็น pointer ชี้ไปยังโครงสร้าง Linked list

Linked List in C (dynamic Memory)

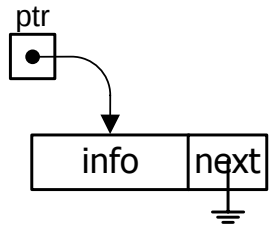
- C ใช้ตัวแปรประเภท pointer ในการจัดการเกี่ยวกับ linked list
  - first เป็น pointer สำหรับข้อมูลตัวแรก
  - ใช้ last เป็น pointer สำหรับข้อมูลตัวสุดท้ายเพื่อสะดวกในการเพิ่มข้อมูล

```
typedef struct  node_type {  double  info ;
                             struct node_type  *next; } linkedlist;
linkedlist  *first=NULL, *last=NULL;
```

- เมื่อต้องการสร้างข้อมูล จะต้องสร้างเนื้อที่ใน dynamic memory ก่อน

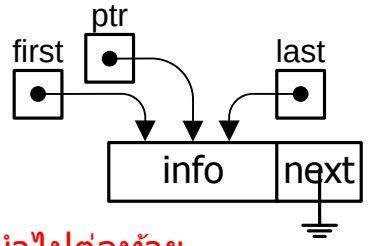
```
linkedlist *ptr;
    ptr = (linkedlist *) malloc(sizeof(linkedlist)) ; //สร้างโหนดเปล่า
    ptr->next = NULL; //เคลียร์พอยต์เตอร์ next
    ptr->info= info; //นำข้อมูลใส่ในโหนด
```

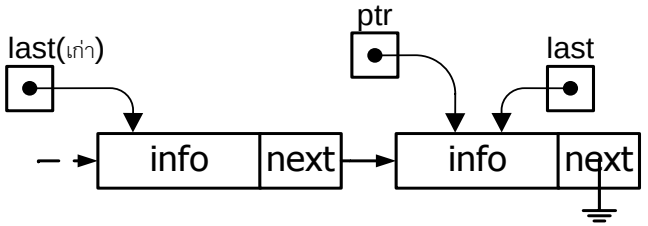
เขียนแบบ pointer to structure (ptr->info แทน (*ptr).info)



- การนำโหนดไปสร้างลิงค์ลิสต์ จะใช้วิธีนำข้อมูลไปต่อๆกัน โดจต้องคำนึงถึงลิงค์ลิสต์ที่มีอยู่เดิมด้วย

```
if  (first == NULL) // ถ้าลิงค์ลิสต์ยังไม่ได้สร้าง
    { first = last = ptr; } // สร้างเป็นลิงค์ลิสต์ตัวแรก
else
    { last->next = ptr; // ถ้าลิงค์ลิสต์สร้างไว้แล้ว ให้นำไปต่อท้าย
      last = ptr; } //ปรับใหม่ให้เป็นตัวสุดท้าย
```

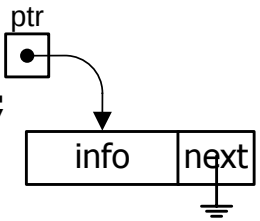


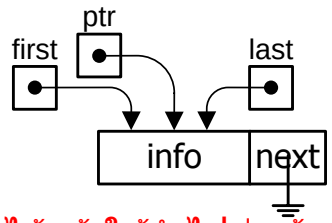


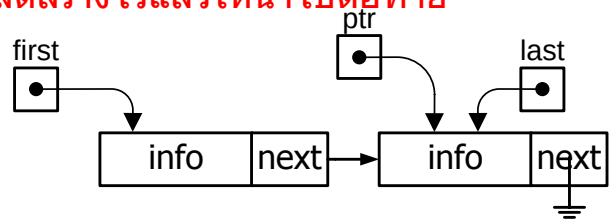
linked lists in dynamic memory

- การสร้างลิงค์ลิสต์ (เพิ่มทีละตัว)

```
void add_linked(double item) // ต้องการนำ item สร้างเป็นลิงค์ลิสต์
{ linkedlist *ptr;
// สร้างโหนดใหม่
   ptr = (linkedlist *) malloc(sizeof(linkedlist)) ;
   ptr -> next = NULL;
   ptr -> info = item;
//นำโหนดไปต่อในลิงค์ลิสต์
   if  (first == NULL) //ถ้าลิงค์ลิสต์ยังไม่ได้สร้าง
       {first = last = ptr;  } //สร้างเป็นลิงค์แรก
   else  { last -> next = ptr; //ถ้าลิงค์ลิสต์สร้างไว้แล้วให้นำไปต่อท้าย
           last = ptr;}
}
```







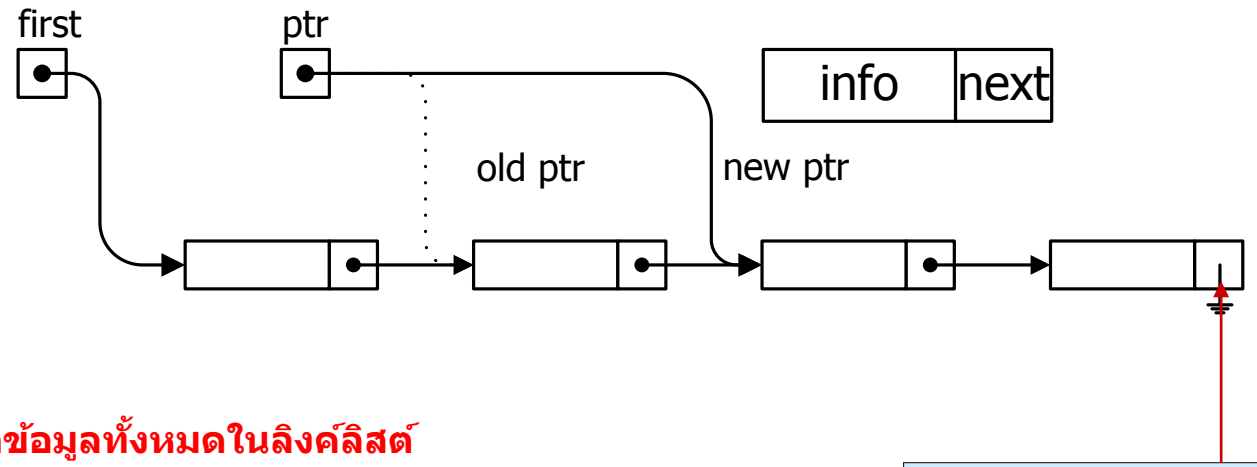
สมมุติให้ *first และ *last เป็น global

```
// ตัวอย่างนำข้อมูลในอาร์เรย์ทั้งหมดสร้างเป็นลิงค์ลิสต์
for (i = 0; i < count ; i++)
     add_node (data[i]);
```

สมมุติข้อมูลที่จะใส่ใน node เป็น double

Traversing a Linked List

- การท่องไปในลิงค์ลิสต์ (การเข้าถึงข้อมูลที่อยู่ในลิงค์ลิสต์)



ข้อมูลตัวสุดท้าย จะชี้ไปยัง NULL
ไม่จำเป็นต้องสนใจ last

```
// แสดงผลข้อมูลทั้งหมดในลิงค์ลิสต์
void print_all()
{  linkedlist *ptr;  // ใช้ ptr เป็นตัววนรอบ
   int count = 0;
   ptr  =  first;  // เริ่มที่ตัวแรก
   while (ptr != NULL)  //ใช้ ptr วนรอบจนกว่าจะหมด ไม่ต้องสนใจ last
   {  count++;  // นับจำนวน
      printf ("data[%d] = %d\n", count, ptr -> info);  //แสดงผลข้อมูลที่ชี้โดยptr
      ptr  =  ptr -> next; }   //เลื่อน ptr เป็นตัวถัดไป
}
```

Searching a Linked List

- การค้นหาข้อมูลที่อยู่ในลิงค์ลิสต์ที่ไม่ได้เรียงลำดับ

```
linkedlist * search_unordered(double item)
{   linkedlist *ptr;  //ใช้ ptr ในการค้นหา
    ptr = first;  // เริ่มต้นที่ตัวแรก
     while  ((ptr  != NULL) && (ptr -> info != item))
            ptr = ptr -> next; //เลื่อนเป็นตำแหน่งถัดไป
    return ptr;
}
```

ค้นจนกว่าจะเจอ หรือหมดข้อมูล

หลุดจากวนรอบ
เมื่อค้นเจอ ptr จะชี้ที่โหนด
ถ้าค้นไม่เจอ ptr จะมีค่าเป็น NULL

example

```
if (search_unorder(item) != NULL)
      printf("Search found");
else
     printf("Search not found");
```

Search in ordered data

- การค้นหาข้อมูลที่อยู่ในลิงค์ลิสต์ที่เรียงลำดับแล้ว

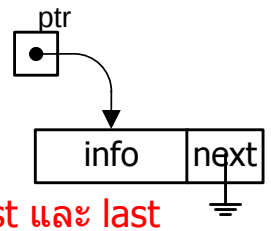
```
linkedlist *search_ordered_data(int item)
{ linkedlist *ptr ;
    ptr = first;
     while  ((ptr != NULL) && (ptr -> info < item))
            ptr = ptr -> next; //หลุดจากวนรอบ เมื่อ info >= item หรือ ptr เป็น NULL
     if  (ptr -> info == item)
        return ptr; //ถ้าค้นเจอ
    else
        return  NULL; //ถ้าค้นไม่เจอ
}

//ตัวอย่างการค้นหา
   if (search_ordered_data(item) != NULL)
        printf("Search found");
   else
        printf("Search not  found");
```

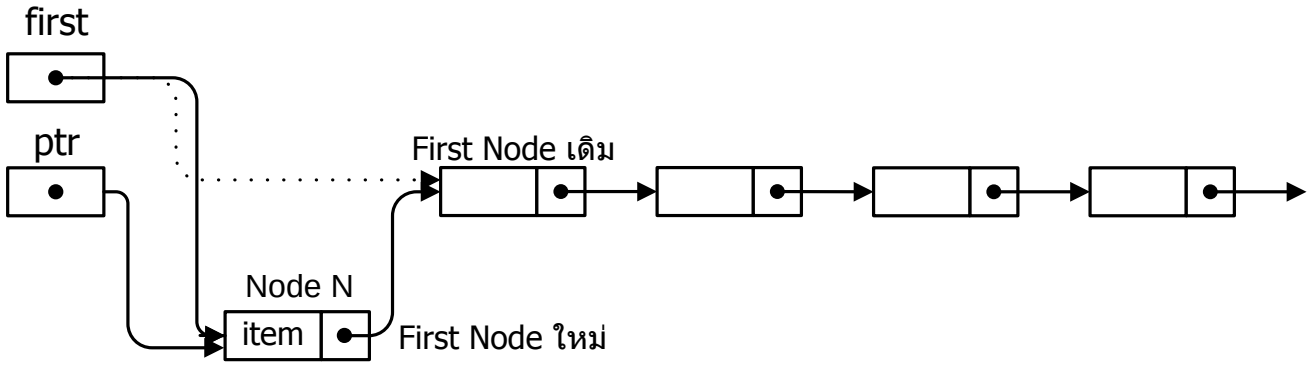
ขณะที่ยังไม่หมดข้อมูล และยังค้นข้อมูลไม่เจอ

Add node at the beginning

- ส่งโหนดที่มีข้อมูล ptr มาเป็นตัวแรกสุดของลิงค์ลิสต์

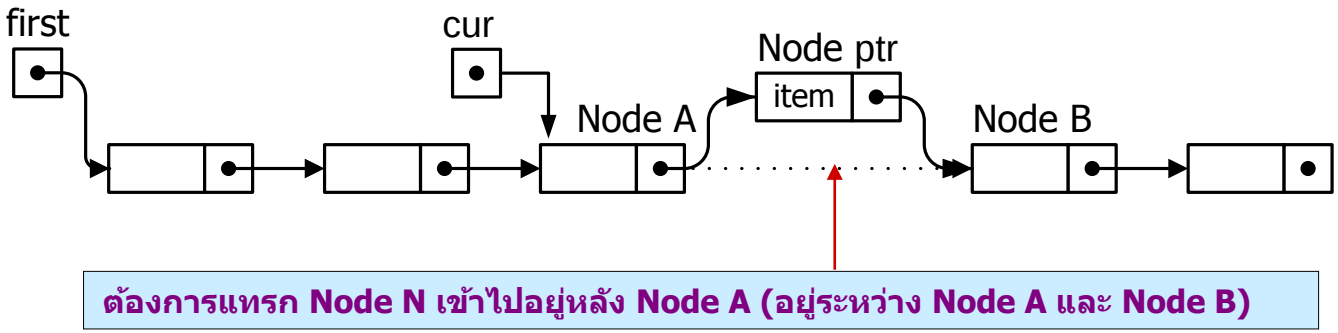
```
void  push_node(linkedlist *ptr) // ใส่ ptr ไว้ที่โหนดแรก
{   if (first==NULL) // กรณียังไม่มีลิงค์ลิสต์
    {  ptr->next = NULL;  //ไม่จำเป็น ถ้าส่ง ptr ที่มี next เป็น NULL
       first = last = ptr; //กรณียังไม่สร้างลิงค์ลิสต์ ต้องกำหนดทั้ง first และ last
    }
    else
    {  ptr->next = first; // ถ้ามี linked list สร้างอยู่แล้ว ให้ปรับ next ชี้ไปยัง first เก่า
       first = ptr;   // ปรับ first ให้ชี้ไปยังตัวใหม่ที่ส่งมา
    }
}
```





Insertion data after a node

- นำข้อมูลมาแทรกต่อท้ายโหนดที่กำหนด(current node)




```
void insert_after_node(linkedlist *cur, linkedlist * ptr)
{           ptr -> next = cur -> next; //ให้โหนดใหม่ชี้ไปยังโหนดถัดไปก่อน
            cur -> next = ptr; //นำโหนดที่กำหนดชี้มายังโหนดใหม่
            if (cur==last) last = ptr; //กรณีต่อหลังโหนดสุดท้าย ต้องปรับค่า last ใหม่
}
```

นำ ptr มาต่อท้าย cur

Insert in order node

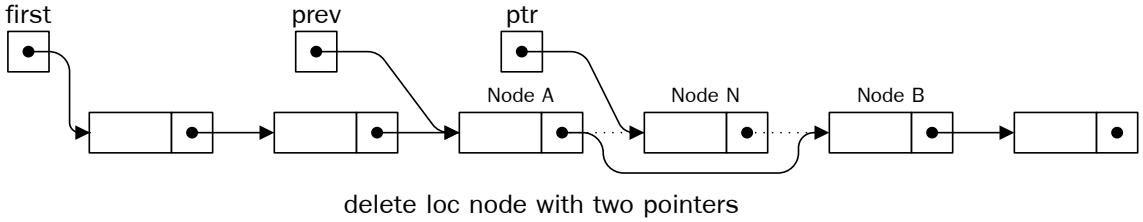
- ค้นหาตำแหน่ง และแทรกโหนดเข้าไปในลิงค์ลิสต์ที่เรียงลำดับ (สร้าง node ไว้แล้ว)
  - กรณีใส่หลังสุดต้องปรับค่า last

```
void insertOrderNode (linkedlist *new_node)
{ linkedlist *ptr;
   if (first==NULL) // ถ้าลิงค์ลิสต์ยังไม่ได้สร้าง
      first = last = new_node; // กำหนดให้เป็นตัวแรก (และตัวสุดท้าย)
  else if (first->info>new_node->info) // ถ้าข้อมูลน้อยกว่าตัวแรกที่มี
  {  new_node->next = first; // โหนดใหม่ชี้ที่ first
     first=new_node; // กำหนด first ใหม่
  }
  else // ถ้าลิงค์ลิสต์สร้างไว้แล้ว
  {  ptr = first; //ค้นตั้งแต่ตัวแรก
     while  ( (ptr->next != NULL) && ((pt                    node->info) )
          ptr = ptr->next; //จนกว่าจะเจอ หรือถึงตัวสุดท้าย (ptr ไม่ใช่ NULL)
     new_node->next = ptr->next; //นำ new_node มาต่อหลัง ptr (ถ้าไม่เจอ ptr จะเป็นตัวสุดท้าย)
     ptr->next = new_node;
     if (ptr==last) //ถ้า ptr เป็นตัวสุดท้าย ต้องปรับ last ตัวสุดท้ายใหม่
       last = new_node;
  }
}
```

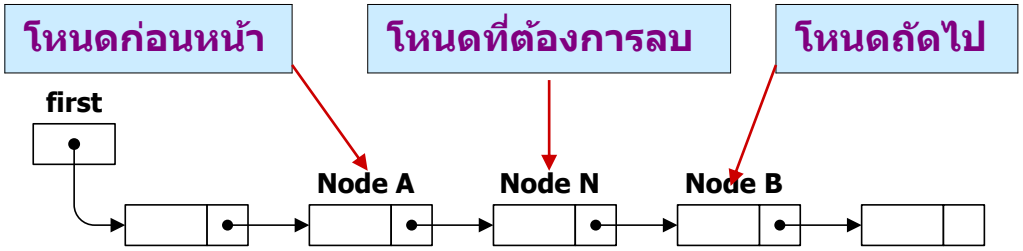
หาตำแหน่งที่จะต้องแทรก

Deletion node

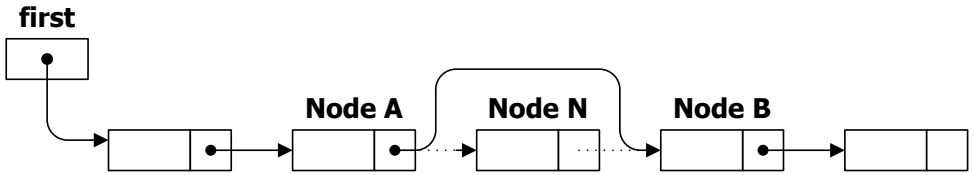
- การลบโหนดออกจากลิงค์ลิสต์ ต้องให้โหนดที่อยู่ก่อนหน้า(prev) ชี้ข้ามไปยังตำแหน่งโหนดถัดไป (ต้องรู้ตำแหน่งโหนดที่อยู่ก่อนหน้าโหนดที่จะลบ)



delete loc node with two pointers



(a) before Deletion



(b) after Deletion

```
ptr  =  prev-> next;
prev->next  = ptr->next ; // prev->next = (prev->next)->next;
```

ต้องการลบ Node N (ptr) ต้องรู้ตำแหน่ง Node A ซึ่งเป็นตำแหน่ง ของโหนดที่อยู่หน้า(prev) ก่อนจึงจะลบได้

Deletion node (with two pointers)

- ต้องการลบโหนดออกจากลิงค์ลิสต์
  - กรณีลบโหนดแรก
  - กรณีลบโหนดกลาง
  - กรณีลบโหนดสุดท้าย
  - กรณีมีโหนดเดียว

```
void  delete_node(linkedlist *prev, linkedlist *ptr)
{     if ((ptr == first) && (first == last)) // ถ้ามีโหนดเดียว
            first = last = NULL;
      else  if (ptr==first) // ถ้ามีหลายโหนด กรณีลบโหนดแรก
               first = first->next;
      else  if (ptr ==last)  // ถ้ามีหลายโหนด กรณีลบโหนดสุดท้าย
              { last = prev;   // กรณีลบโหนดสุดท้าย ต้องใช้ prev
                last->next = NULL; }
      else  prev -> next = ptr -> next; //กรณีลบโหนดทั่วไป ต้องใช้ prev
      free(ptr); //คืนหน่วยความจำให้กับระบบ
}
```

Search (return two pointer)

- ค้นหาโหนดที่ต้องการ โดยส่งกลับ pointer 2 ตัว (เพื่อจะใช้ในการลบ)

double pointer เพื่อเปลี่ยนค่า pointer ที่ส่งกลับ

```
int search_prev_pointer( double  item, linkedlist **prev, linkedlist **ptr);
{   *ptr = first; //เริ่มที่โหนดแรก
    if  ((*ptr)->info == item) //ถ้าเจอที่โหนดแรก ให้ prev เป็น NULL
    {  (*prev) = NULL;
       return 1;
     }
    else  {  while  ((*ptr != NULL) && ((*ptr)->info != item)) //วนรอบค้นต่อ
             { *prev = *ptr;  // ก่อนเปลี่ยนตำแหน่ง ให้จำค่า ptr ไว้ใน prev
               *ptr = (*ptr)->next;
              }
             if (*ptr != NULL) //กรณีค้นเจอ
                { return 1 ; }  //cur และ prev จะถูกกำหนดค่าไว้แล้ว
             else  { *ptr = NULL; //กรณีค้นไม่เจอ ตั้งค่า cur และ prev ให้เป็น NULL
                     *prev = NULL;
                     return 0 ; }
          }
}
```

```
linkedlist *prev, *ptr ;
Found = search_with_two_pointer(item, &prev, &ptr);
```

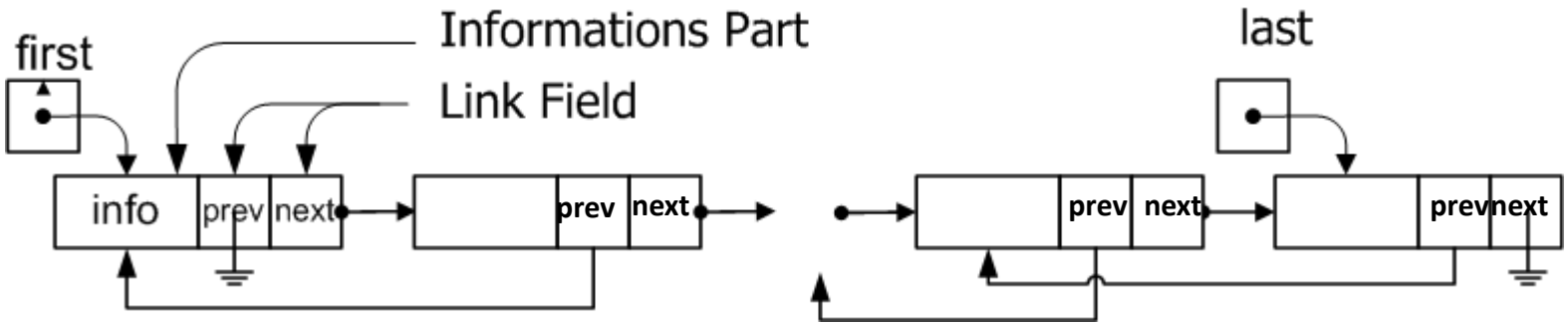
Search and Delete Node

- ค้นหาข้อมูลพร้อมกับลบทิ้ง

```
int searchAndDeleteNode (double info)
{  linkedlist  *prev , *ptr ;
   ptr = first; //เริ่มที่โหนดแรก
   if  ( ptr->info == info)  //ถ้าเจอที่โหนดแรก
   {  first = first->next ;
      free(ptr); //คืนหน่วยความจำให้กับระบบ
      return 1;  }
   else
   {  while  ((ptr != NULL) && (ptr->info != info)) //ค้นโหนดถัดไป
       {  prev = ptr; //ทุกครั้งที่เปลี่ยนตำแหน่ง ptr จะต้องจำเอาไว้เป็น prev ก่อน
          ptr = ptr->next; }
       if  (ptr != NULL) //ถ้าเจอที่โหนดที่จะลบ
       {  if (ptr==last) //ถ้าเจอที่โหนดสุดท้าย
              last = prev; //ปรับพอยต์เตอร์ที่ชี้โหนดสุดท้าย
          prev->next = ptr->next; //ลบโหนด (ปรับพอยต์เตอร์ข้ามโหนด)
          free(ptr); //คืนหน่วยความจำให้กับระบบ
          return 1;  }
       else
          return 0; //ค้นไม่เจอที่โหนดที่จะลบ
   }
}
```

Double Linked List

- **Double Linked list**
  - **Linked List ที่มีตัวชี้ถัดไป และตัวชี้ย้อนกลับไปยังตัวก่อนหน้า**
  - **การลบโหนดใช้ pointer ตัวเดียว ไม่จำเป็นต้องรู้ pointer ที่อยู่ก่อนหน้า prev**




```
typedef struct Node_type {  data_type info ;
                            struct Node_type *prev;
                            struct Node_type *next; } doublelist;
doublelist  *first , *last, *ptr;
```

Add node in Double Linked List

- เพิ่มโหนดเข้าไปใน double linked list

```
doublelist  *ptr;
      ptr  =  (doublelist *) malloc( sizeof(doublelist)) ;
      ptr->info = info;
      ptr->prev = ptr->next = NULL;
```

สร้างโหนดใหม่

```
void addNode (doublelist *ptr)
{     if  (first == NULL)   // ถ้าลิงค์ยังไม่ถูกสร้าง ให้กำหนดเป็นโหนดแรก
      {  first = ptr;
         last = ptr;
      }
      else
      {  ptr->prev = last; // ถ้าลิงค์สร้างแล้ว prev ชี้ไปยังโหนดสุดท้ายเดิม
         last->next = ptr;   //ให้โหนดสุดท้ายเดิม ชี้มายังโหนดนี้
         last = ptr;          //ปรับตำแหน่งโหนดสุดท้ายใหม่
       }
}
```

Delete node in double Linked List

- การลบโหนดของ double linked list

```
void deleteNode(doublelist *ptr)  // รู้เพียงโหนดที่ต้องการลบ
{
    if ((ptr == first) && (first == last)) // ถ้าเหลืออยู่โหนดเดียว
          {  first = last = NULL; }
    else if (ptr==first) // กรณีลบโหนดแรก
          {  first = first->next;
             first->prev = NULL; }
    else if (ptr ==last) {// กรณีลบโหนดสุดท้าย
          {   last = last->prev;
              last->next = NULL; }
    else { (ptr->prev)->next = ptr->next; // กรณีลบโหนดทั่วไป
           (ptr->next)->prev = ptr->prev ;  }
    free(ptr) ;
}
```

Scan Sort in Linked List

- เรียงลำดับข้อมูลในลิงค์ลิสต์

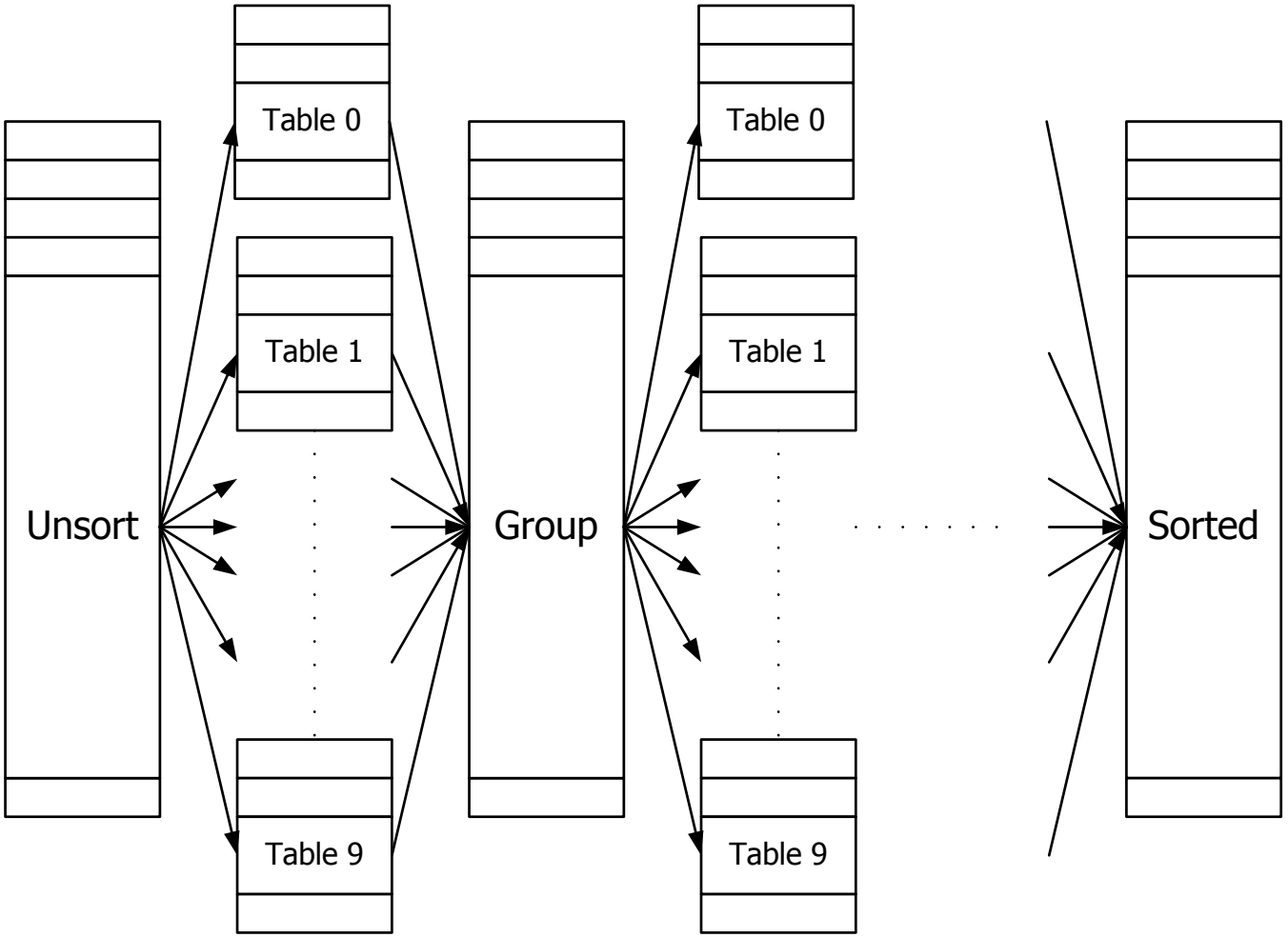
```
void  scanSortLinkedList()
{linkedlist  *ptr_i, *ptr_j ;
  double  x ;
    ptr_i = first;
    while (ptr_i->next != NULL)  { //วนรอบจนกว่าจะเรียงลำดับเสร็จ
          ptr_j = ptr_i->next;  // ใช้ ptr_j เป็นตัว เปรียบเทียบ
          while (ptr_j != NULL)  { //วนรอบจนกว่าจะเปรียบเทียบครบทุกตัว
                if (ptr_j->info < ptr_i->info) { //เปรียบเทียบและสลับค่า
                    x = ptr_i->info;
                    ptr_i->info = ptr_j->info;
                    ptr_j->info = x;
                  }
                ptr_j = ptr_j->next; // เปรียบเทียบตัวถัดไป
          }
          ptr_i  = ptr_i->next; // ตั้งตัว reference ถัดไป
    }
}
```

ตั้ง ptr_i เป็นreference

สลับค่าข้อมูล info ระหว่างโหนด (ไม่ได้สลับโหนด)

Radix sort

- **Divided & Merged**



Algorithm of Radix sort

- ใช้ตารางช่วยในการเรียงลำดับ
- สร้างอาร์เรย์ของตาราง เพิ่มอีกจำนวนเท่ากับค่าที่เป็นไปได้ของแต่ละหลัก (เหมาะสำหรับกรณีที่คีย์เป็นตัวเลข : ถ้าเป็นตัวเลขจะมี (0 .. 9 รวม 10 ตาราง) ทำให้สิ้นเปลืองหน่วยความจำมาก เพื่อเก็บตัวเลขที่มีค่าในหลักที่กำหนด ลงไปในตารางของแต่ละหลัก
- ถ้าใช้ linked list มาช่วยในการสร้างตารางจะลดหน่วยความจำไปได้
- เรียงลำดับได้เร็วถ้าข้อมูลมีเป็นจำนวนมาก และมีคีย์เป็นตัวเลข

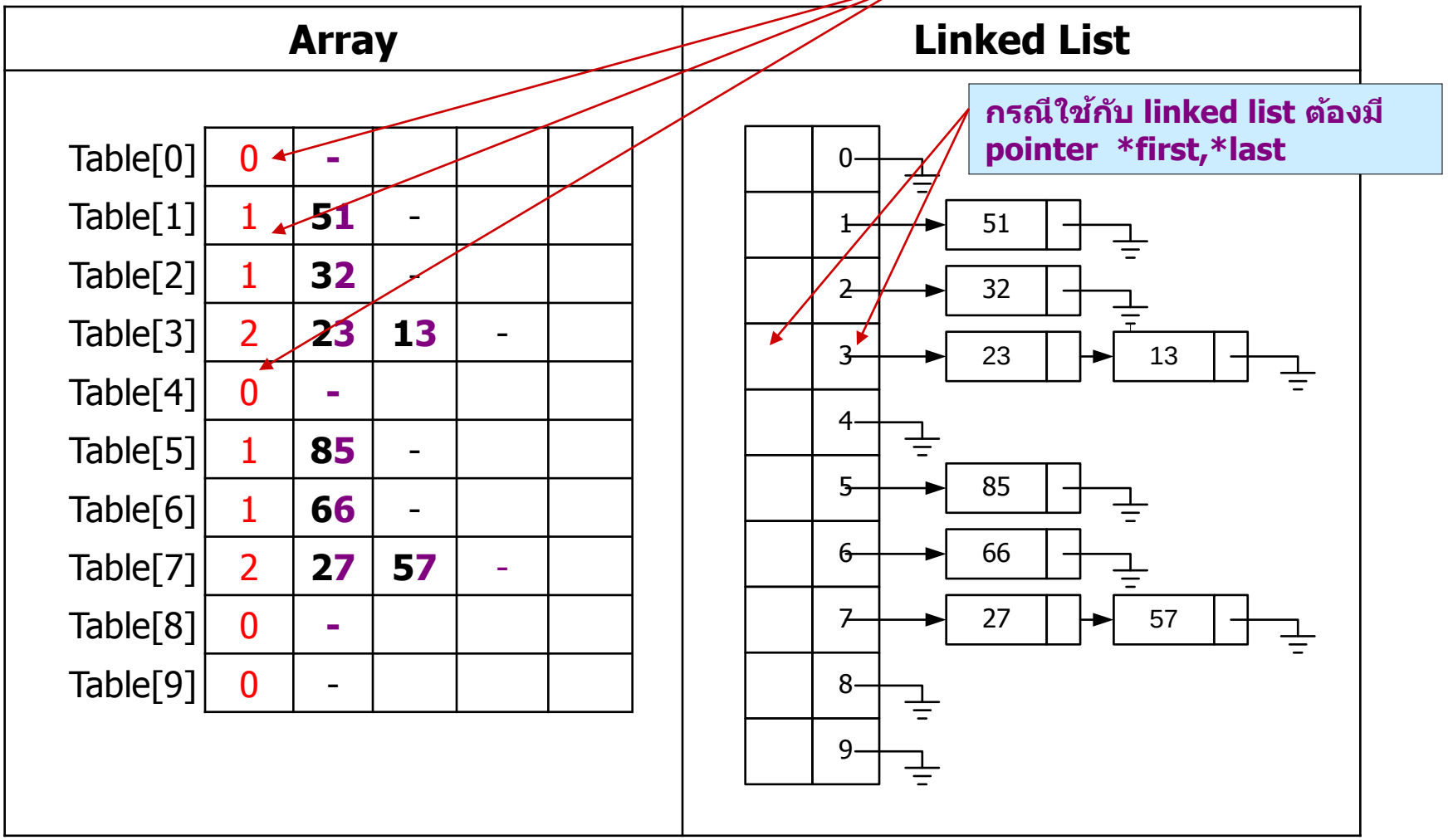
  O(d*n) เมื่อ d เป็นจำนวนหลักของคีย์
- วิธีการเรียงลำดับ
  - พิจารณาตัวเลขทีละหลัก เริ่มที่หลักสุดท้าย(digit 0) จนถึงหลักแรก
  - วนรอบ นำข้อมูลที่ต้องการเรียงลำดับมาใส่ไว้ในตารางที่เตรียมไว้สำหรับเก็บข้อมูลของหลักนั้น
  - เมื่อใส่ข้อมูลครบทุกตัวลงในตารางแล้ว ให้นำข้อมูลในแต่ละตาราง มารวมกันเป็นอาร์เรย์ชุดใหม่
  - เลื่อนหลักที่สนใจมาข้างอีก 1 หลัก แล้วทำซ้ำจนกระทั่งครบทุกหลัก

Example 32, 51, 27, 85, 66, 23, 13, 57

- Loop1 selected decimal digit = 0

กรณีใช้กับอาร์เรย์ ต้องมีที่เก็บจำนวนข้อมูลในตาราง

กรณีใช้กับ linked list ต้องมี pointer *first,*last



Merge : 51, 32, 23, 13, 85, 66, 27, 57

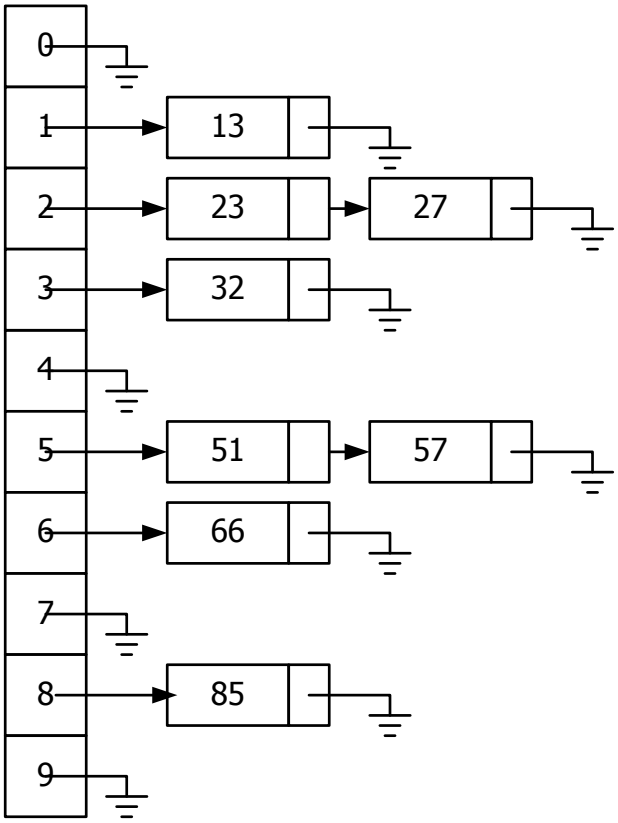
51, 32, 23, 13, 85, 66, 27, 57

- Loop 2 selected decimal digit = 1

Array / Queue

	[0]	[1]	[2]	[3]	...
Table[0]	0	-			
Table[1]	1	13	-		
Table[2]	2	23	27	-	
Table[3]	1	32	-		
Table[4]	0	-			
Table[5]	2	51	57	-	
Table[6]	1	66	-		
Table[7]	0	-			
Table[8]	1	85	-		
Table[9]	0	-			

Linked List



Merge 13, 23, 27, 32, 51, 57, 66, 85 (Sorted Data)

Radix Sort (Linked List)

```
typedef struct node_tag { long int info;
                          struct node_tag  *next;
                        } node_type;
struct table_type {node_type *first,*last;} table[10];
node_type   *first=NULL , *last=NULL;
int maxdigit = 8;  //เรียงลำดับตัวเลข 8 หลัก
void radix_sort()
{ int  digit;
   for (digit=0; digit < maxdigit; digit++)    //ทำรอบซ้ำ เท่ากับจำนวนหลักสูงสุดของข้อมูล
   {   devided_linked_listed_to_table (digit); //แบ่งตัวเลขลง 10 ตาราง
       merge_table_to_linked_listed ();         //รวม 10 ตารางกลับมาเป็นตารางเดียว
    }
}
```

ตัวอย่างนี้ใช้ตัวแปรเป็น Global ทั้งหมด

ตารางของ linked list แบ่งเป็น table[i].first, table[i].last

Divide to Table

```
void devided_linked_listed_to_table(int digit)
{ int i, tablenum ;
  node_type *ptr;
  for(i=0; i<=9; i++) //เคลียร์ตารางลิงค์ย่อยของแต่ละหลักเลข
        table[i].first = table[i].last = NULL;
  ptr=first; //เริ่มต้นที่ตัวแรกของลิงค์หลัก
   while (ptr != NULL) //วนจนครบทุกตัว
   {   tablenum = (int)(ptr -> info / pow(10,digit)) % 10; //คำนวนเพื่อเลือกหมายเลขตารางลิงค์
       if (table[tablenum].first==NULL)  //ถ้าตารางย่อยยังไม่มีข้อมูล
        {  //  ถ้าเป็น double  linked list  ให้ ptr->prev = NULL;
           table[tablenum].first=ptr;
           table[tablenum].last=ptr;}
      else { // ถ้าเป็น double  linked list  ให้ ptr->prev = table[tablenum].last;
                table[tablenum].last->next=ptr;
                table[tablenum].last=ptr;}
      ptr = ptr->next; // เลือกโหนดใหม่จากลิงค์หลัก
      table[tablenum].last -> next= NULL; // ต้องตัด pointer ไม่ให้ชี้ไปยัง linked เก่า
    }
}
```

นำโหนด ptr ไปสร้างเป็นตัวแรกของตาราง

นำโหนด ptr ไปต่อตัวสุดท้าย

Merge Table

```
void merge_table_to_linked_listed( )
{int i;
   first = last = NULL; //เตรียมลิงค์ลิสต์ ใหม่สำหรับรวมลิงค์ลิสต์ในตาราง
   for (i=0; i<=9;i++) //วนรอบสำหรับตารางของหลักเลข 0-9
  { if (table[i].first != NULL)  //ถ้ามีลิสต์ในตารางย่อยของหลักนั้น ให้นำลิงค์ไปต่อในลิงค์รวม
      { if (first == NULL)       //กรณีถ้ายังไม่สร้างลิงค์ลิสต์รวม ให้ตารางนั้นเป็นชุดแรก
          { first = table[i].first; //ตัวแรกของลิงค์รวม ชี้มาที่ตัวแรกตารางย่อย
            last = table[i].last;  //ตัวสุดท้ายตารางรวม ชี้มาที่ตัวสุดท้ายตารางย่อย
          }
        else  //กรณีมีลิงค์รวมอยู่แล้ว  นำลิงค์ของตารางย่อยไปต่อท้าย
          {  last -> next = table[i].first;  //ตัวสุดท้ายลิงค์รวมชี้มาที่ตัวแรกของลิงค์ย่อย
             last = table[i].last; //เปลี่ยนตัวสุดท้ายลิงค์รวมชี้มาที่ตัวสุดท้ายของลิงค์ย่อย
          }
      }
  }
}
```