Chap 4. Linked Lists (1)

Contents

4.1 Singly Linked Lists and Chains

- Ordered list

  (BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, VAT, WAT)

  – Sequential representation: ***array***

  – Non-sequential representation: ***linked list***

Sequential Representation

- Sequential storage scheme
- Successive items of a list are located a fixed distance apart
- *The order of elements is the same as in the ordered list*
- Insertion and deletion of arbitrary elements become expensive
  - excessive data movement

Linked Representation

- Successive items of a list may be placed anywhere in memory
- *The order of elements need not be the same as in the ordered list*
- A linked list is comprised of ***nodes***
  - each node has zero or more *data fields* and one or more *link or pointer fields* to the next item
- Insertion and deletion of arbitrary elements become easier
  - no data movement

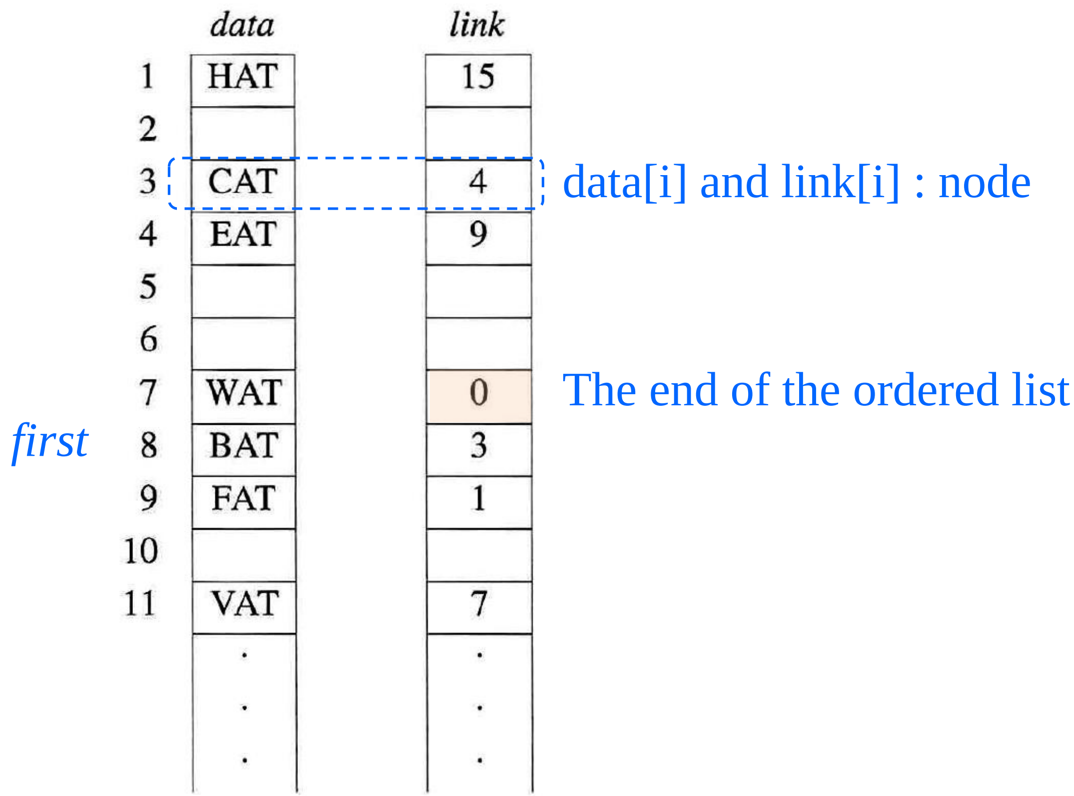


Figure 4.1: Nonsequential list-representation using two arrays

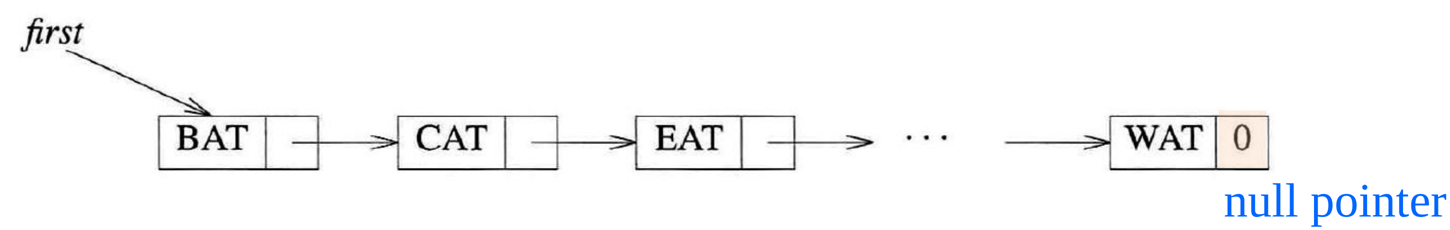


Figure 4.2: Usual way to draw a linked list

- In a singly linked list, each node has exactly one pointer field.
- A chain is a singly linked list that is comprised of zero or more nodes.

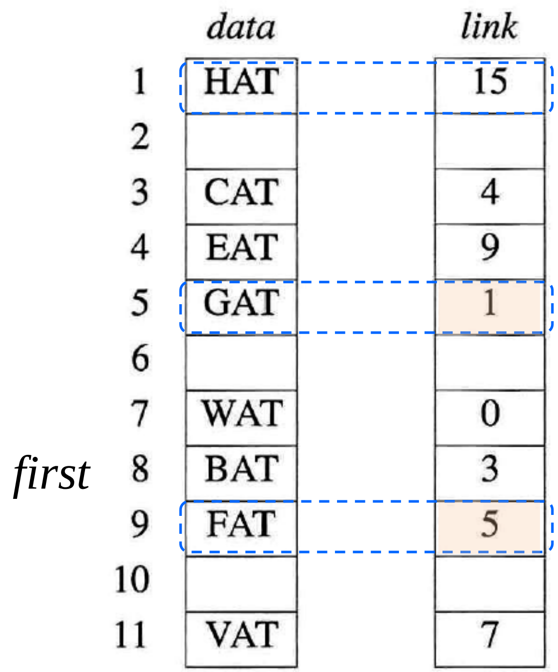
Linked List : Insert(GAT)

(1) Get a node *a* that is currently unused.

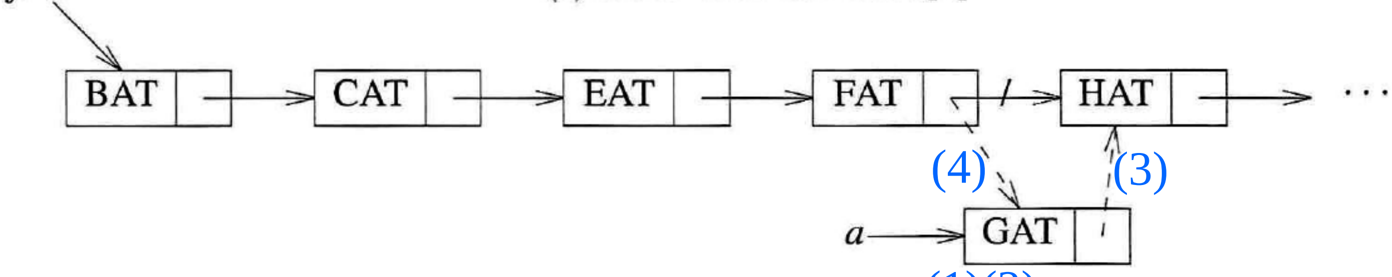
(2) Set the *data* field of *a* to GAT.

(3) Set the *link* field of *a* to point to the node after FAT, which contains HAT.

(4) Set the *link* field of the node containing FAT to *a*.



(a) Insert GAT into *data* [5]



(b) Insert node GAT into list

Figure 4.3: Inserting into a linked list

Linked List : Delete(GAT)

(1) Find the element that immediately precedes GAT

(2) Set its link filed to point to the node after GAT

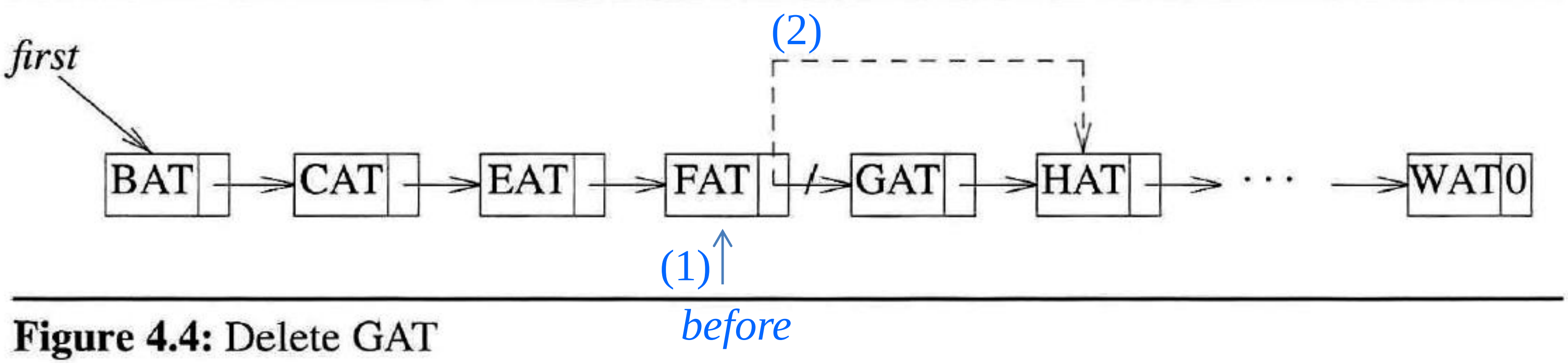


Figure 4.4: Delete GAT

4.2 Representing Chains in C

- **Example 4.1 [List of words]**
  - Defining a node's structure
    - ***self-referential structure***

```
typedef struct listNode *listPointer;
typedef struct listNode {
        char data[4];
        listPointer link;
        }listNode;
```

  - Creation of a new empty list

```
listPointer first = NULL;
```

  - Test for an empty list

```
#define IS_EMPTY(first) (! (first) )
```

Example 4.1 [List of words]

– Creation of a new node for the list

MALLOC(first, sizeof(*first));

– Assigning values to the fields of the node

strcpy(first→data, “BAT”);
first→link = NULL;

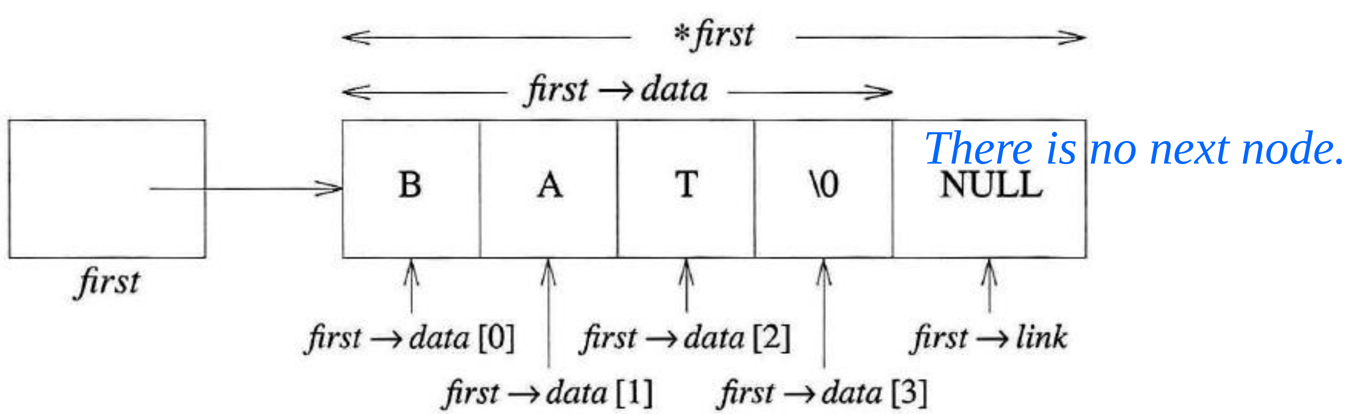


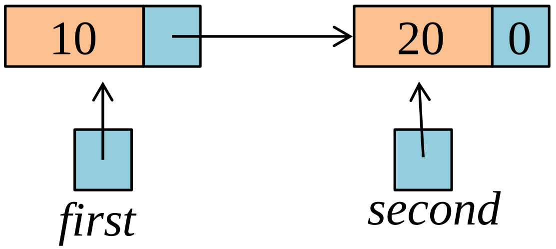
Figure 4.5: Referencing the fields of a node

Example 4.2 [Two-node linked list]

```
typedef struct listNode *listPointer;
typedef struct listNode {
        int data;
        listPointer link;
        }listNode;
```

```
listPointer create2()
{/* create a linked list with two nodes */
   listPointer first, second;
   MALLOC(first, sizeof(*first));
   MALLOC(second, sizeof(*second));
   second→link = NULL;
   second→data = 20;
   first→data = 10;
   first→link = second;
   return first;
}
```

Program 4.1: Create a two-node list

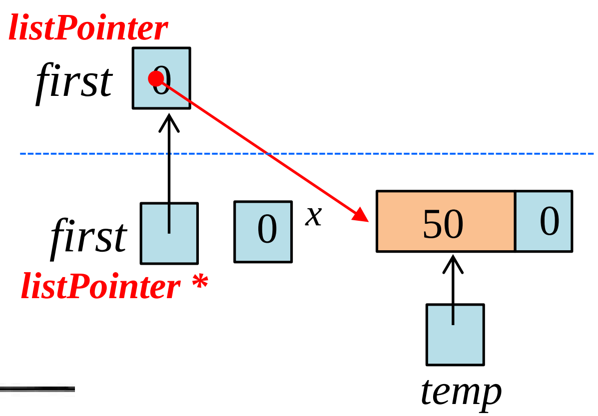


Example 4.3 [List insertion]

```
void insert(listPointer *first, listPointer x)
{/* insert a new node with data = 50 into the chain
    first after node x */
  listPointer temp;
  MALLOC(temp, sizeof(*temp));
  temp→data = 50;
  if (*first) {
     temp→link = x→link;
     x→link = temp;
  }
  else {
     temp→link = NULL;
     *first = temp;
  }
}
```

Program 4.2: Simple insert into **list**

(a) Inserting into an empty list
insert(&first, NULL)



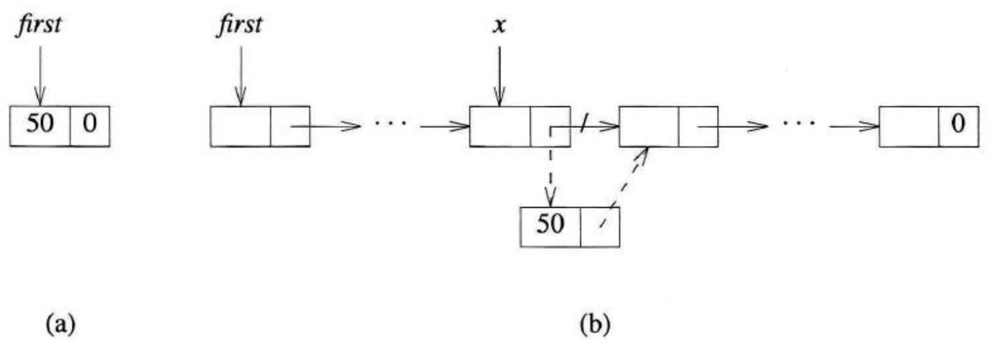


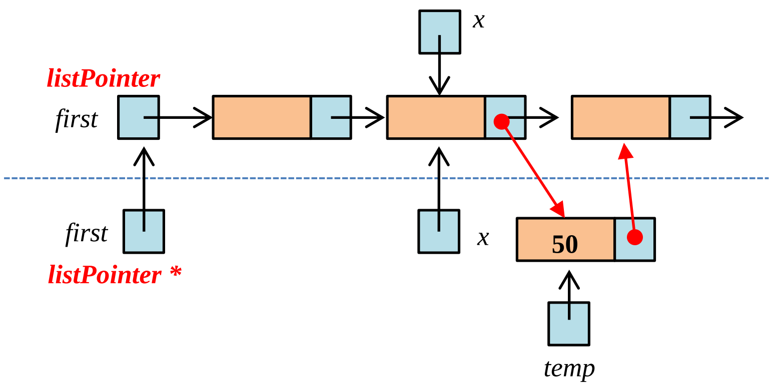
Figure 4.7: Inserting into an empty and nonempty list

Example 4.3 [List insertion]

```
void insert(listPointer *first, listPointer x)
{/* insert a new node with data = 50 into the chain
    first after node x */
   listPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→data = 50;
   if (*first) {
      temp→link = x→link;
      x→link = temp;
   }
   else {
      temp→link = NULL;
      *first = temp;
   }
}
```

Program 4.2: Simple insert into **list**

(b) Inserting into a nonempty list
insert(&first, x)



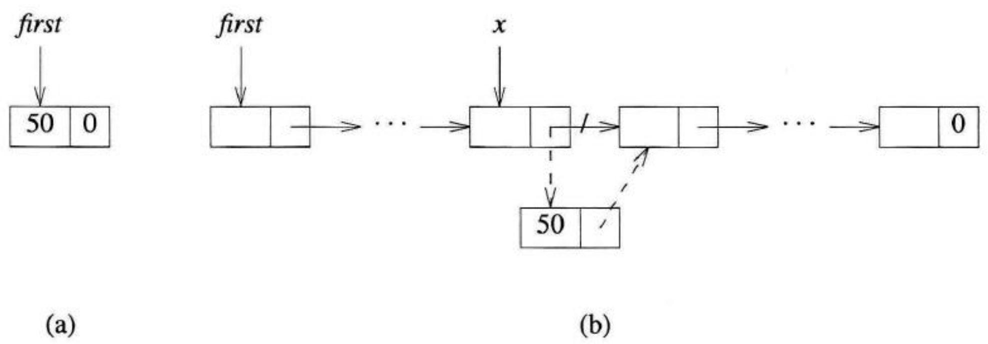
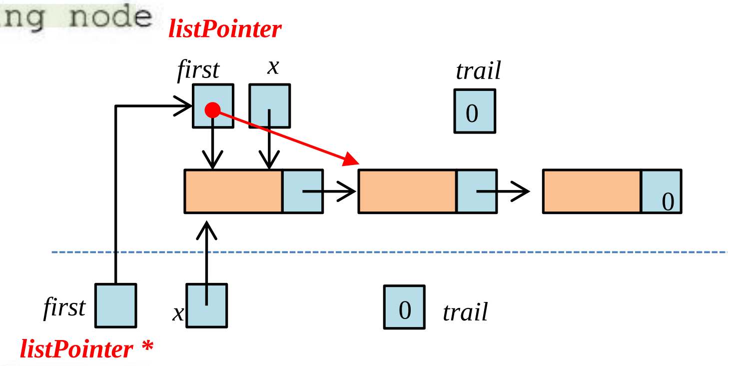


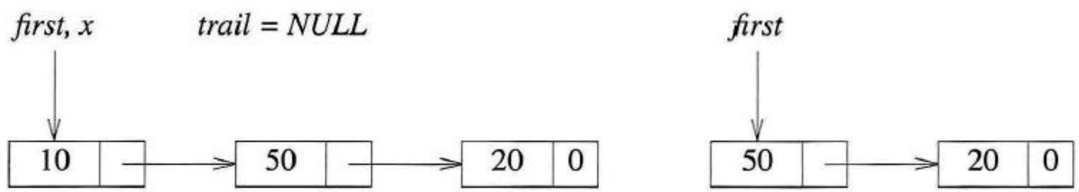
Figure 4.7: Inserting into an empty and nonempty list

Example 4.4 [List deletion]

```
void delete(listPointer *first, listPointer trail,
                                listPointer x)
{/* delete x from the list, trail is the preceding node
    and *first is the front of the list */
   if (trail)
     trail→link = x→link;
   else
     *first = (*first)→link;
   free(x);
}
```



Program 4.3: Deletion from a list

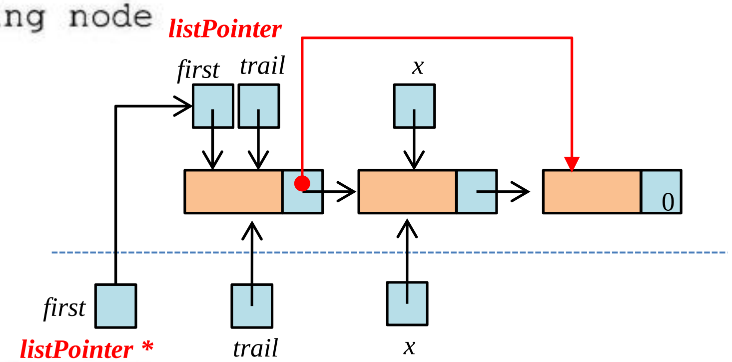


(a) Before deletion (b) After deletion

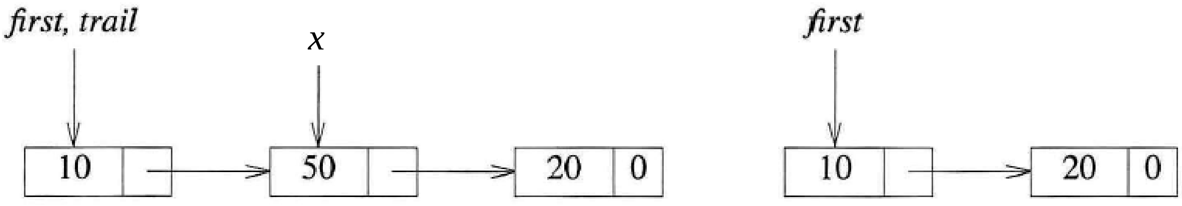
Figure 4.8: List before and after the function call *delete(&first, trail, x)*

Example 4.4 [List deletion]

```
void delete(listPointer *first, listPointer trail,
                                listPointer x)
{/* delete x from the list, trail is the preceding node
    and *first is the front of the list */
   if (trail)
     trail→link = x→link;
   else
     *first = (*first)→link;
   free(x);
}
```



Program 4.3: Deletion from a list



(a) Before deletion (b) After deletion

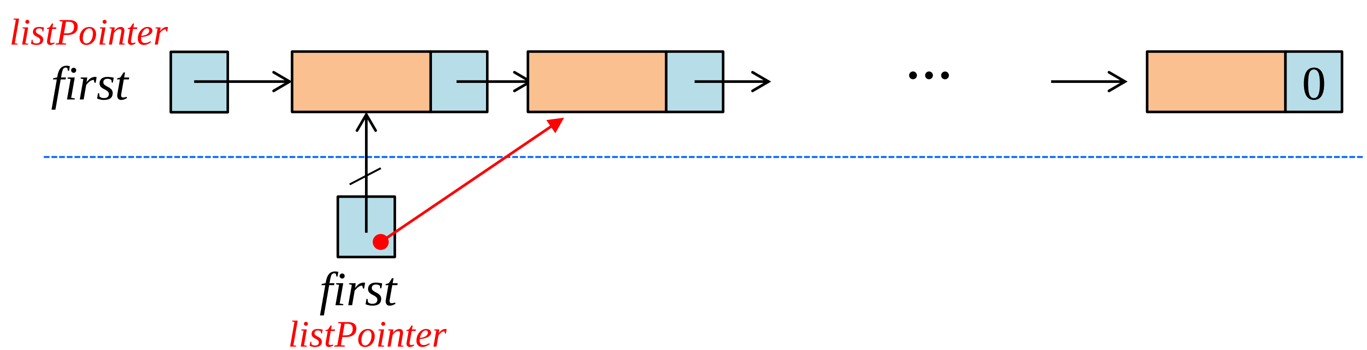
Figure 4.9: List after the function call ***delete(&first, trail, x)***

Example 4.5 [Printing out a list]

```
void printList(listPointer first)
{
   printf("The list contains: ");
   for (; first; first→link)
      printf("%4d",first→data);
   printf("\n");
}
```

Program 4.4: Printing a list

printList(first)



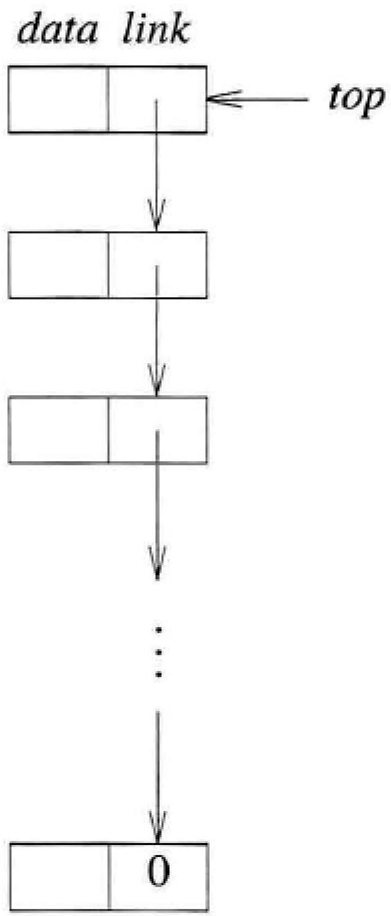
4.3 Linked Stacks And Queues

- Representing **$n \leq MAX\_STACKS$ stacks** simultaneously

```
#define MAX_STACKS 10 /* maximum number of stacks */
typedef struct {
        int key;
        /* other fields */
        } element;
typedef struct stack *stackPointer;
typedef struct stack {
        element data;
        stackPointer link;
        } node;
stackPointer top[MAX_STACKS];
```

$top[i] = NULL,\ 0 \leq i < MAX\_STACKS$ Initial conditions for the stacks

$top[i] = NULL$ iff the ith stack is empty



(a) Linked stack

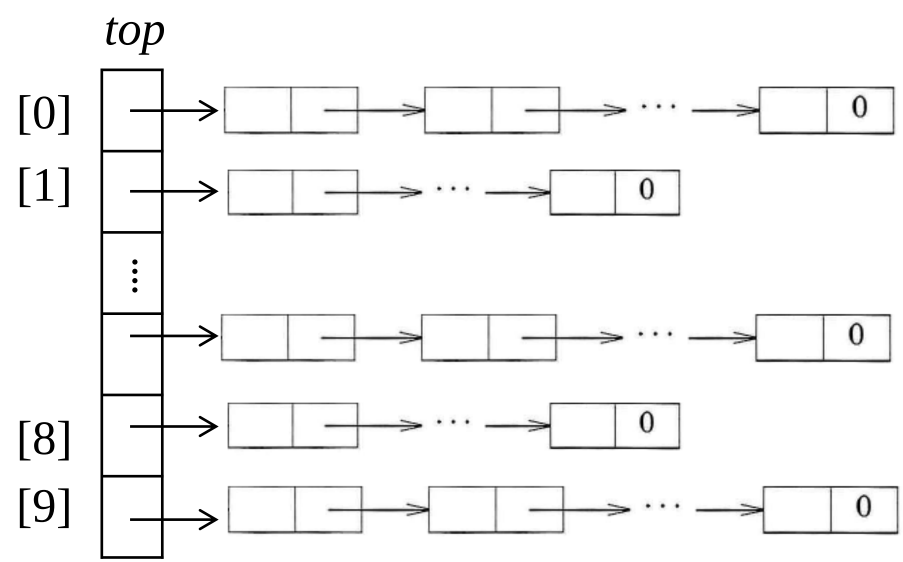
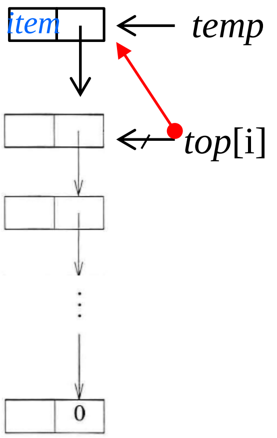


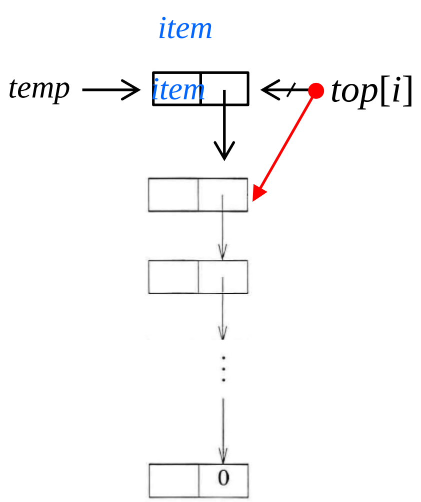
Figure 4.11: Linked stack and queue (1/2)

```
void push(int i, element item)
{/* add item to the ith stack */
   stackPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→data = item;
   temp→link = top[i];
   top[i] = temp;
}
```



Program 4.5: Add to a linked stack *push(i, item)*

```
element pop(int i)
{/* remove top element from the ith stack */
   stackPointer temp = top[i];
   element item;
   if (!temp)
     return stackEmpty();
   item = temp→data;
   top[i] = temp→link;
   free(temp);
   return item;
}
```



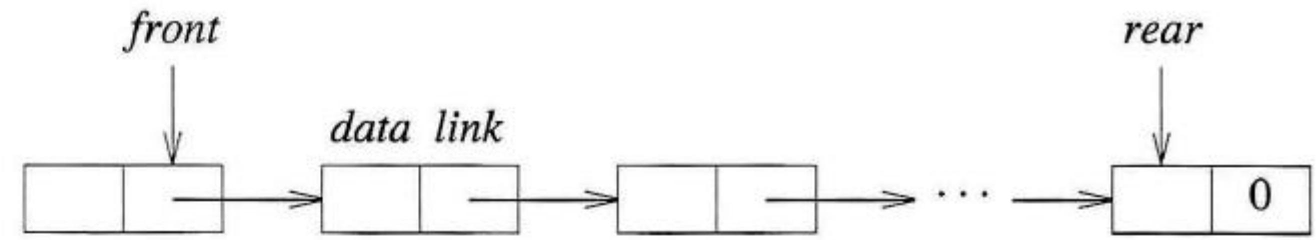
Program 4.6: Delete from a linked stack *item = pop(i)*

- Representing ***n≤MAX_QUEUES* queues** simultaneously

```
#define MAX_QUEUES 10 /* maximum number of queues */
typedef struct queue *queuePointer;
typedef struct queue {
        element data;
        queuePointer link;
        } node;
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

$front[i] = NULL, 0 \leq i < MAX\_QUEUES$ Initial conditions for the queues

$front[i] = \mathrm{NULL}$ iff the ith queue is empty



(b) Linked queue

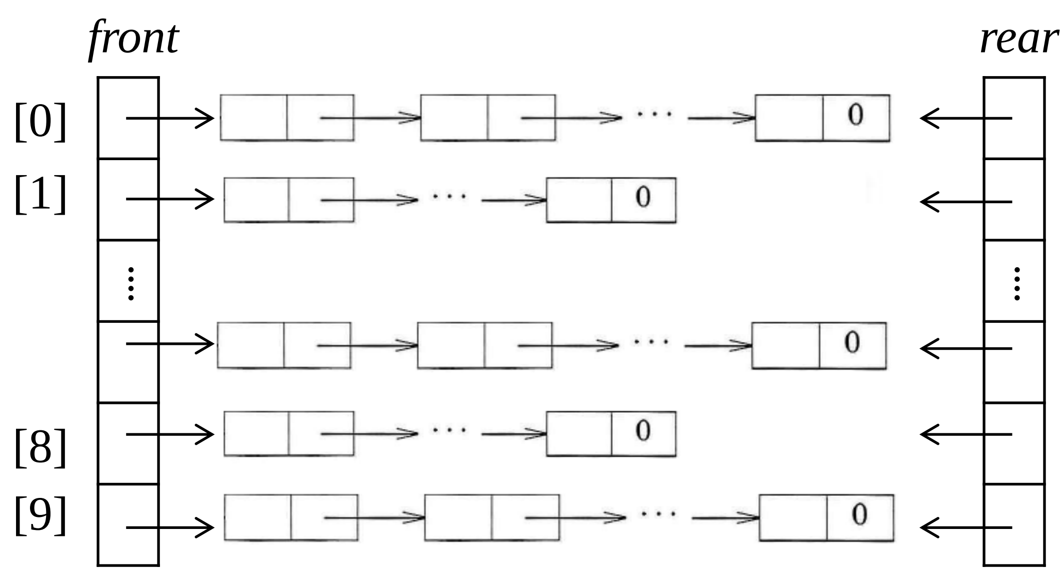
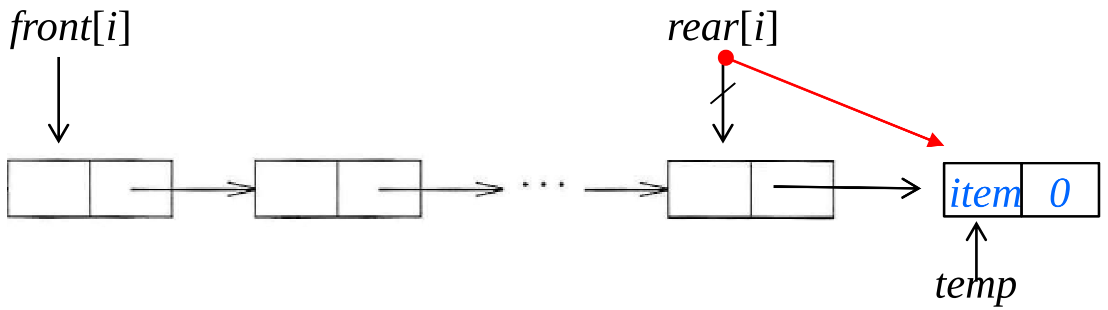


Figure 4.11: Linked stack and queue (2/2)

```
void addq(int i, element item)
{/* add item to the rear of queue i */
   queuePointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→data = item;
   temp→link = NULL;
   if (front[i])
       rear[i]→link = temp;
   else// addition to empty queue
       front[i] = temp;
   rear[i] = temp;
}
```

Program 4.7: Add to the rear of a linked queue *addq(i, item)*



```
element deleteq(int i)
{/* delete an element from queue i */
   queuePointer temp = front[i];
   element item;
   if (!temp)
      return queueEmpty();
   item = temp→data;
   front[i]= temp→link;
   free(temp);
   return item;
}
```

Program 4.8: Delete from the front of a linked queue *item = deleteq(i)*