자료구조

Chap 4: Linked list

2018년 1학기

컴퓨터과학과
민 경 하

Contents

4. Linked list

1. Introduction
2. Two types of list implementation
3. Data structure of linked list
4. Operations of singly linked list
5. Doubly linked list
6. Operations of Doubly linked list
7. Performance

0. struct VS class

- struct in C VS class in C++
  - struct + typedef == class

```
typedef struct _node node;
struct _node {
    char ats[3];
    node *link;
};
```

```
class node {
    char ats[3];
    node *link;
};
```

1. Introduction

- Types of data structure

Organization	Data structure		Implementation	
			Index-based	**Pointer-based**
Linear	List	Array	O	
		Linked list		O
	Stack		O	O
	Queue		O	O
Hierarchical	Tree			O
Arbitrary	Graph		O	O

1. Introduction

- What is list?
  - A fundamental linear data structure
  - A mapping of an element and an index

  - Two types of implementations
    - Index-based → array
    - Pointer-based → linked list

2. Two types of list implementation

- Array
  - The successive elements of data object are stored in **a fixed distance** apart
  - Example
    - a_i is stored at L_i ➔ a_{i+1} is stored at $L_i + d$
      ➔ a_{i-1} is stored at $L_i - d$
- Linked list
  - The successive elements of data object are stored at **arbitrary position** of memory
  - Each element has a pointer to the next element

2. Two types of list implementation

Array

VS

Linked

data + link

➔ **node**

address	*data*
0	BAT
1	CAT
2	EAT
3	FAT
4	HAT
5	JAT
6	LAT
7	MAT
8	OAT
9	PAT
10	RAT
11	SAT
12	VAT
13	WAT

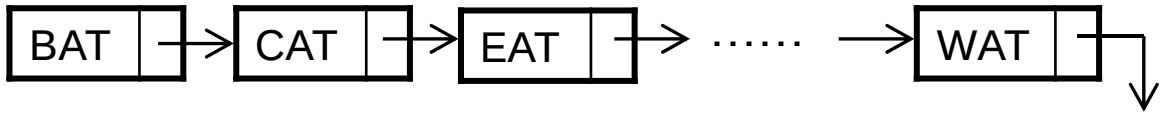
array

address	*data*	*link*
0	HAT	15
1		
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	-1
8	BAT	3
9	FAT	0
10		
11	VAT	7
12	:	:
13	:	:

linked

3. Data structure of linked list

- Linked list
  - Nodes are stored in an arbitrary position in memory
  - Each node possesses a pointer to its next node

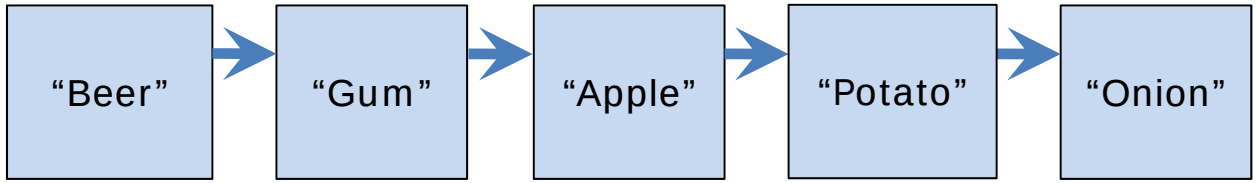


address	*node*	
0	HAT	15
1		
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	-1
8	BAT	3
9	FAT	0
10		
11	VAT	7
12	:	:
13	:	:

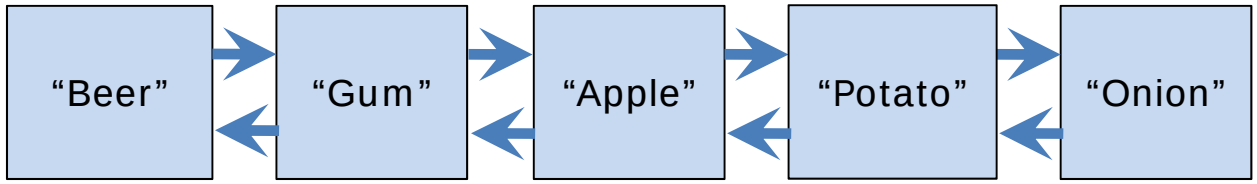
3. Data structure of linked list

- Singly linked list
  - Each node has exactly one pointer field
  - Chain
    - A singly linked list that is comprised of zero or more nodes
- Doubly linked list
  - Each node has two pointer fields

Singly linked list



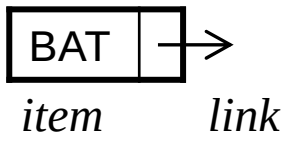
Doubly linked list

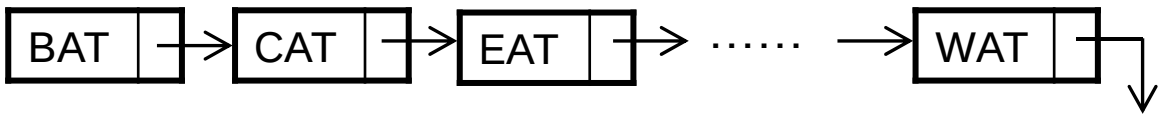


3. Data structure of linked list

- Data structure for a node of singly linked list

```
class node {
    data_type item;
    node *link;
};
```

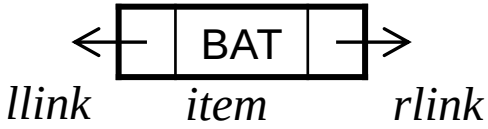


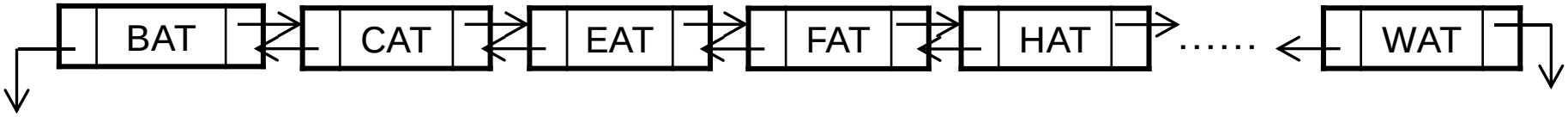


3. Data structure of linked list

- Data structure for a node of doubly linked list

```
class node {
    data_type item;
    node *llink, *rlink;
};
```



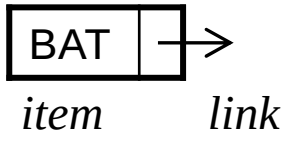


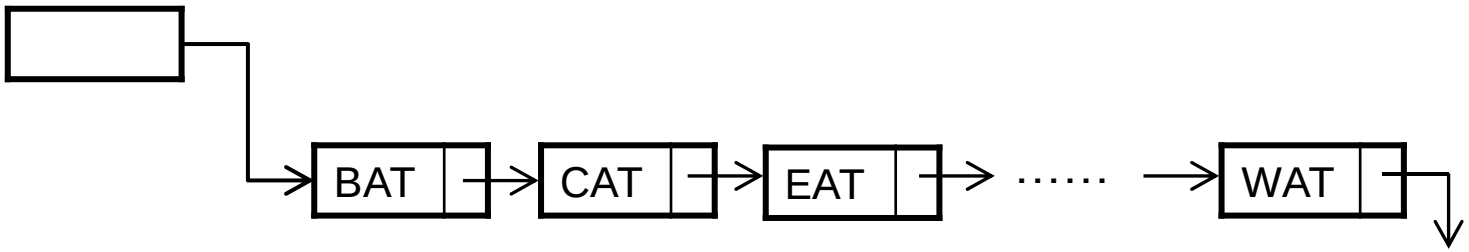
3. Data structure of linked list

- Singly linked list with head node

```
class node {
    data_type item;
    node *link;
};
```

```
class hnode {
    node *link;
};
```



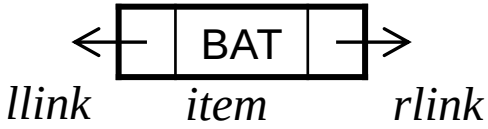


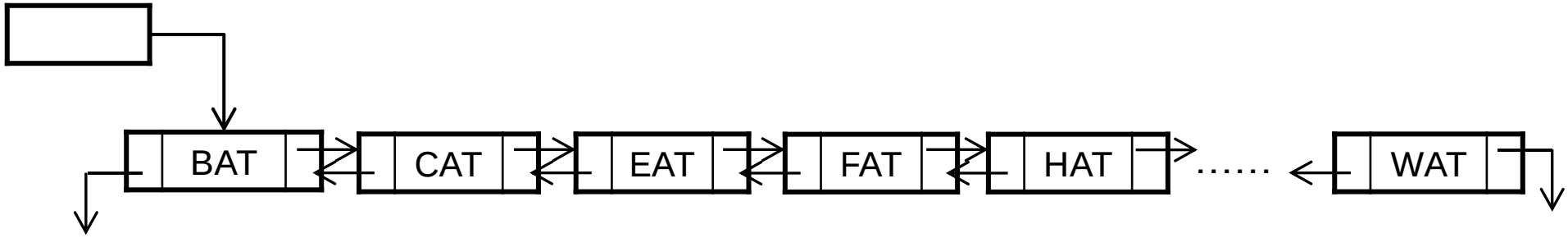
3. Data structure of linked list

- Data structure for a node of doubly linked list

```
class node {
    data_type item;
    node *llink, *rlink;
};
```

```
class hnode {
    node *link;
};
```





4. Operations of Singly Linked List

- Operations on linked list
  (1) **Create**
  (2) **Search**
  (3) **Insert**
  (4) **Delete**
  (5) Modify
  (6) Length
  (7) Next
  (8) Previous
  (9) etc

4. Operations of Singly Linked List

Operation	Sorted linked list 2 → 5 → 7 → 9 → 10	Unsorted linked list 5 → 2 → 7 → 10 → 9
Search	Linear_search (A, x)	Linear_search (A, x)
	~~Binary_search (A, x)~~	
Insert	Insert_by_value (A, x) **(A, 8):** **2 → 5 → 7 → 8 → 9 → 10**	Insert (A, x) **(A, 8):** **8 → 5 → 2 → 7 → 10 → 9**
		Insert_by_index (A, i, x) **(A, 3, 8):** **5 → 2 → 7 → 8 → 10 → 9**
Delete	Delete_by_value (A, x) **(A, 5): 2 → 7 → 9 → 10**	Delete_by_value (A, x) **(A, 5): 2 → 7 → 10 → 9**
	Delete_by_index (A, i) **(A, 3): 2 → 5 → 7 → 10**	Delete_by_index (A, i) **(A, 3): 5 → 2 → 7 → 9**

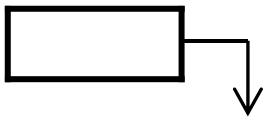
4. Operations of Singly Linked List

(1) Create

– Build an empty linked list

```
void main ( )
{
    hnode first;
}
```

first

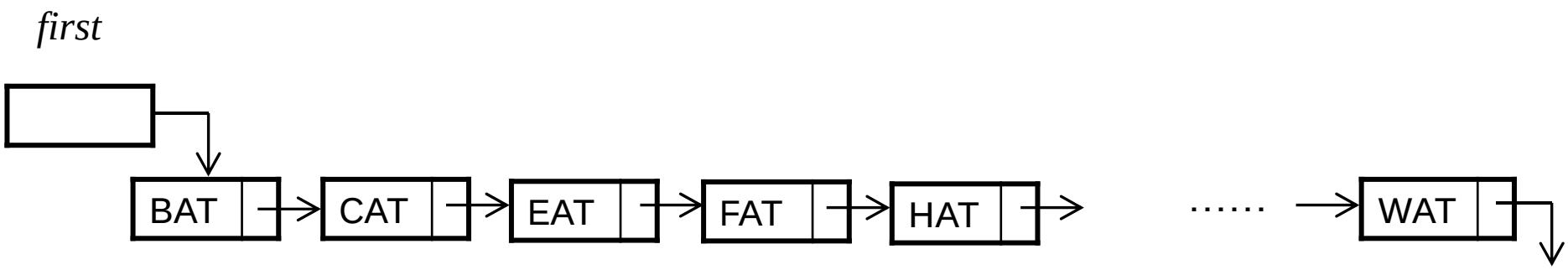


4. Operations of Singly Linked List

(2) Search

– Find a node that contains a data item to be retrieved

```
void main ( )
{
     node *temp = first.search ( “HAT” );
}
```

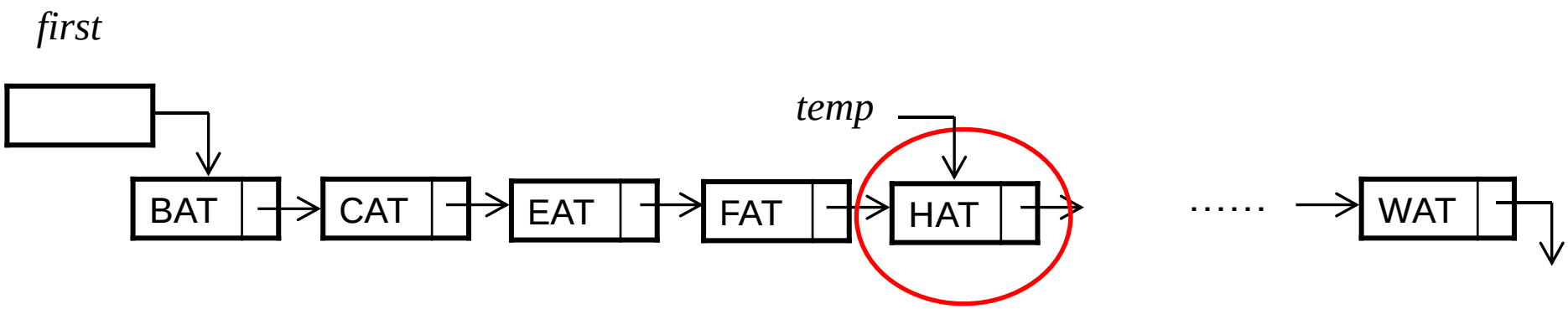


4. Operations of Singly Linked List

(2) Search

– Find a node that contains a data item to be retrieved

```
void main ( )
{
    node *temp = first.search ( “HAT” );
}
```



4. Operations of Singly Linked List

(2) Search

1. search ( ) at hnode

```
node *hnode::search( data_type item )
{
    return this->link->search ( item );
}
```

4. Operations of Singly Linked List

(2) Search

1. search ( ) at node

```
node *node::search( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr != NULL ) {
        if ( curr->item == item )
            return curr;  // Found
        curr = curr->link;
    }

    return NULL;          // Not found
}
```

4. Operations of Singly Linked List

(2) Search

```
node *curr = this;
while ( curr != NULL ) {
    if ( curr->item == item )
        return curr;  // Found
    curr = curr->link;
}
```

```
for ( node *curr = this; curr != NULL; curr = curr->next )
    if ( curr->item == item )
        return curr;
```

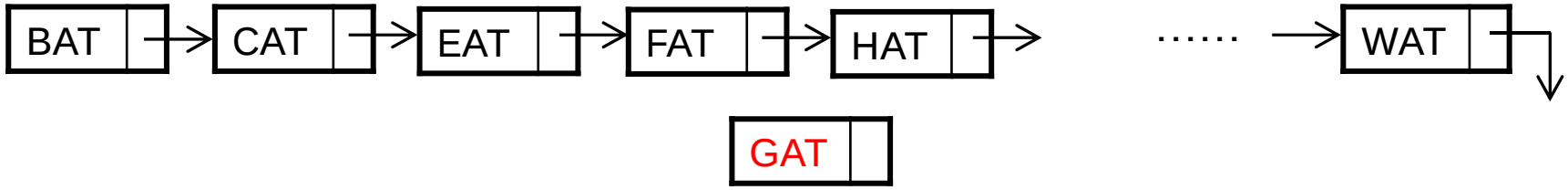
```
for ( int i = 0; i < n; i = i + 1 )
    if ( list[i] == item )
        return list[i];
```

4. Operations of Singly Linked List

(3) Insert

- Make a node with a data item to insert
- Add the node at a proper position

```
void main ( )
{
    first.insert ( “GAT” );
}
```

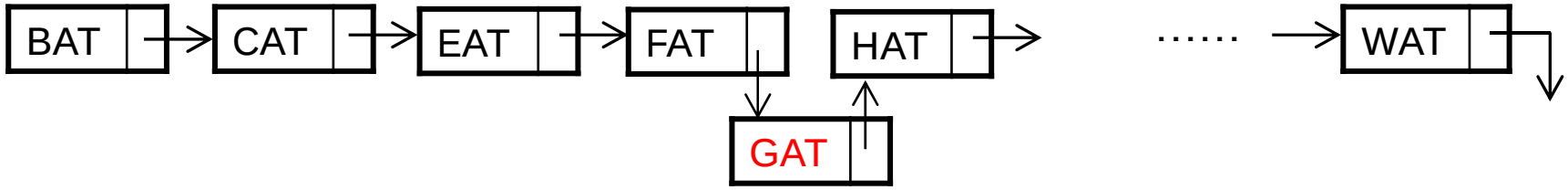


4. Operations of Singly Linked List

(3) Insert

- Make a node with a data item to insert
- Add the node at a proper position

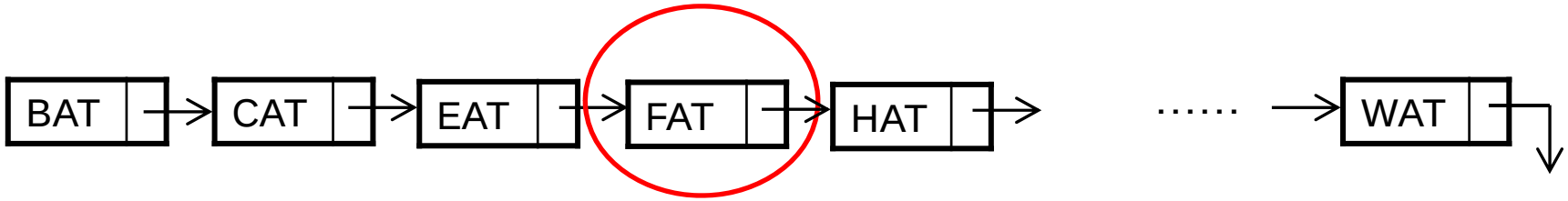
```
void main ( )
{
    first.insert ( “GAT” );
}
```



4. Operations of Singly Linked List

(3) Insert

1. Find a proper position to insert an item

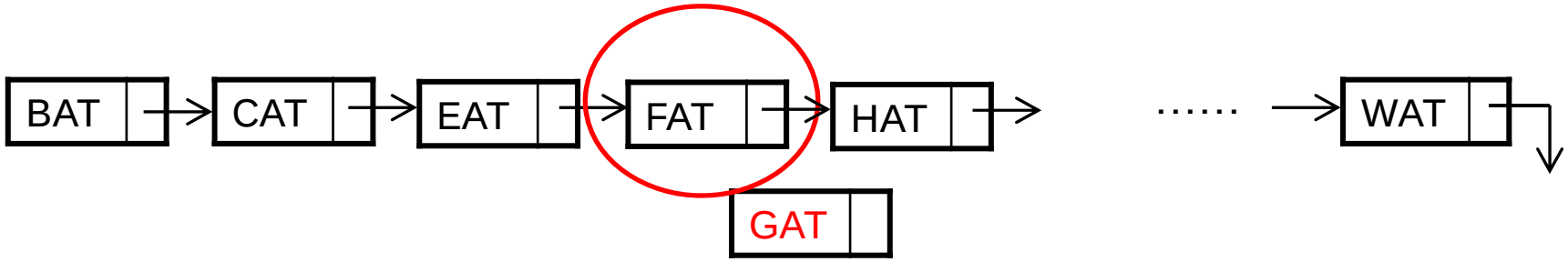


4. Operations of Singly Linked List

(3) Insert

1. Find a proper position to insert an item

2. Build a new node for the item



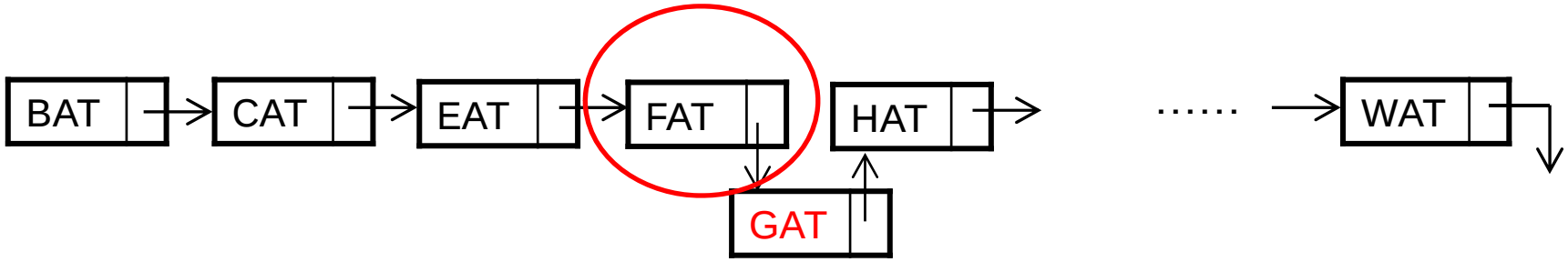
4. Operations of Singly Linked List

(3) Insert

1. Find a proper position to insert an item

2. Build a new node for the item

3. Modify the pointers of the list to contain the new node in the list



4. Operations of Singly Linked List

(3) Insert

```
void hnode::insert ( data_type item )
{
    this->link->insert ( item );
}
```

4. Operations of Singly Linked List

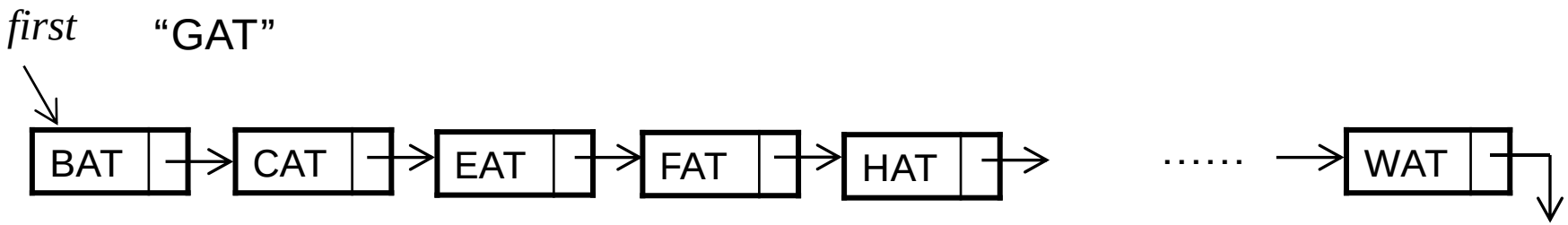
(3) Insert

```
void node::insert ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item > item )
            break;
        curr = curr->link;
    }
//  2. Build a new node
    node *nnode = new node;
    nnode->item = item;
//  3. Modify the pointers
    nnode->link = curr->link;
    curr->link = nnode;
}
```

4. Operations of Singly Linked List

(3) Insert

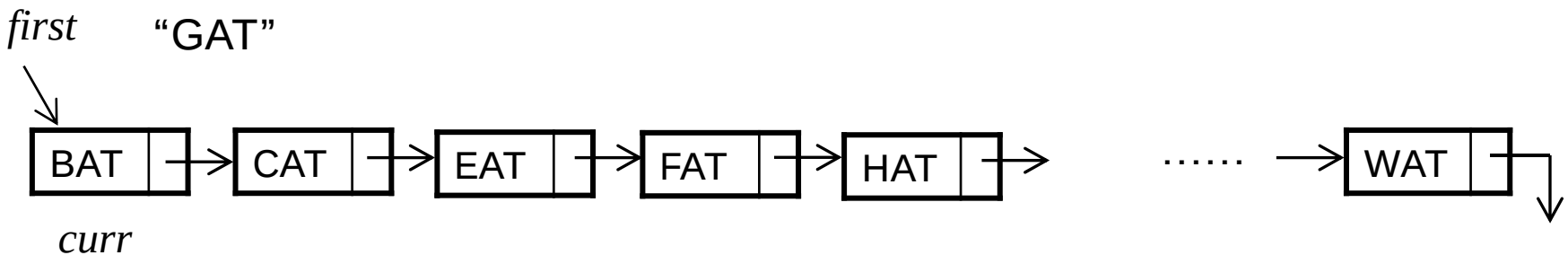
```
void node::insert ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item > item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(3) Insert

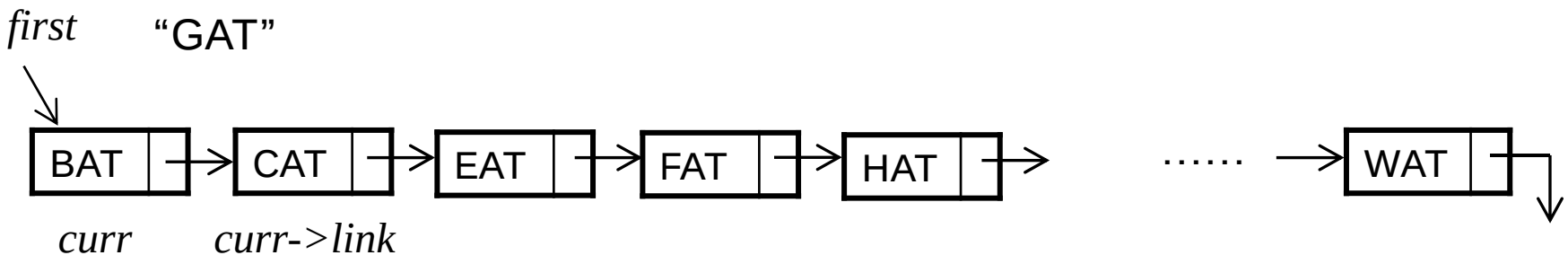
```
void node::insert ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item > item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(3) Insert

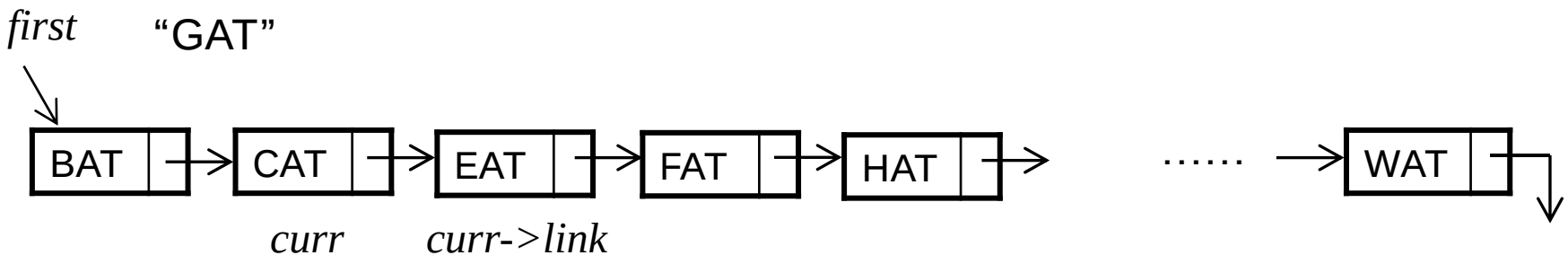
```
void node::insert ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item > item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(3) Insert

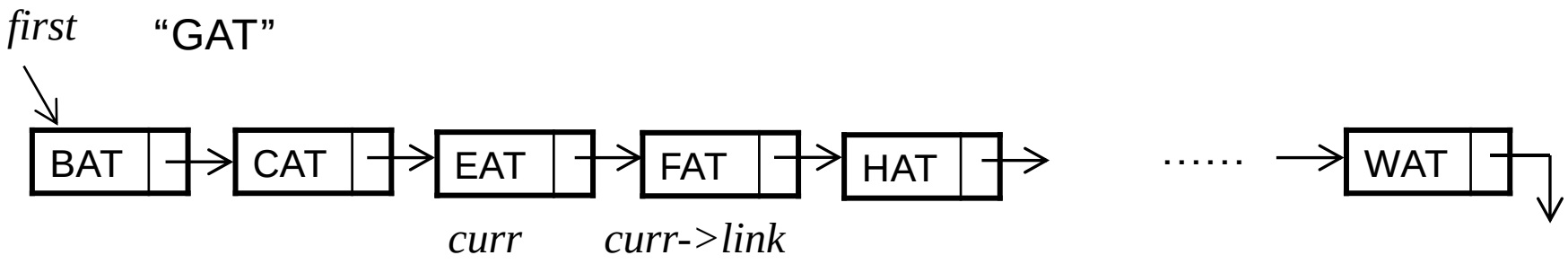
```
void node::insert ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item > item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(3) Insert

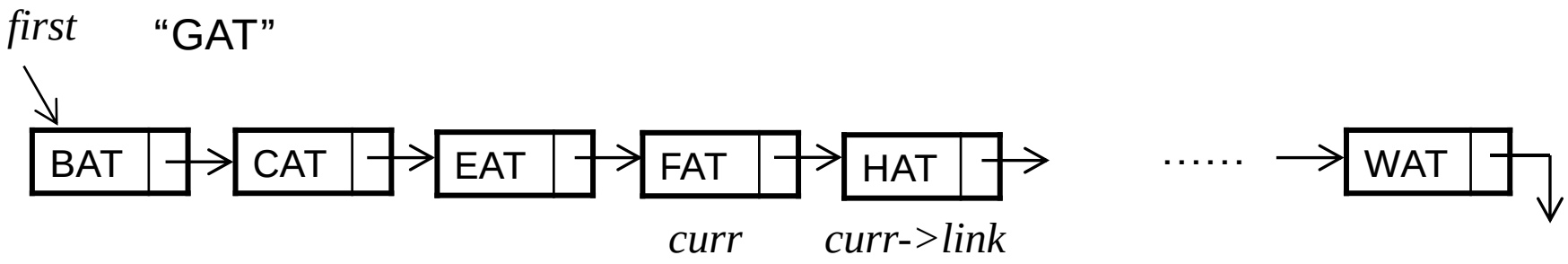
```
void node::insert ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item > item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(3) Insert

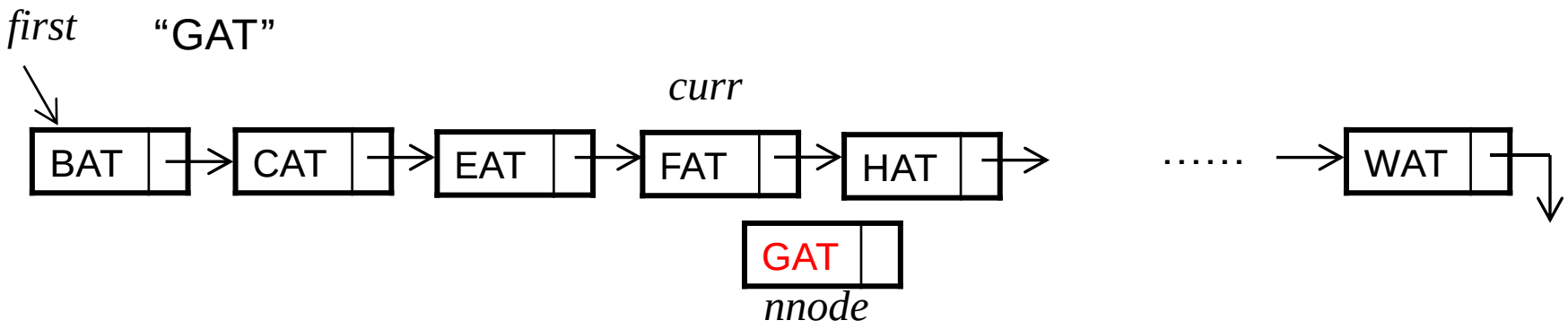
```
void node::insert ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item > item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(3) Insert

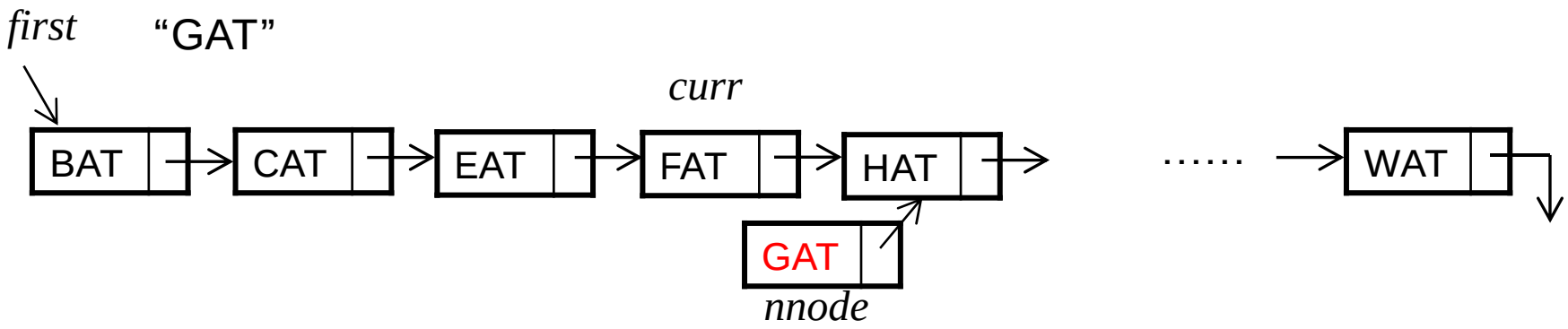
```
void node::insert ( data_type item )
{
//  2. Build a new node
    node *nnode = new node;
    nnode->item = item;
}
```



4. Operations of Singly Linked List

(3) Insert

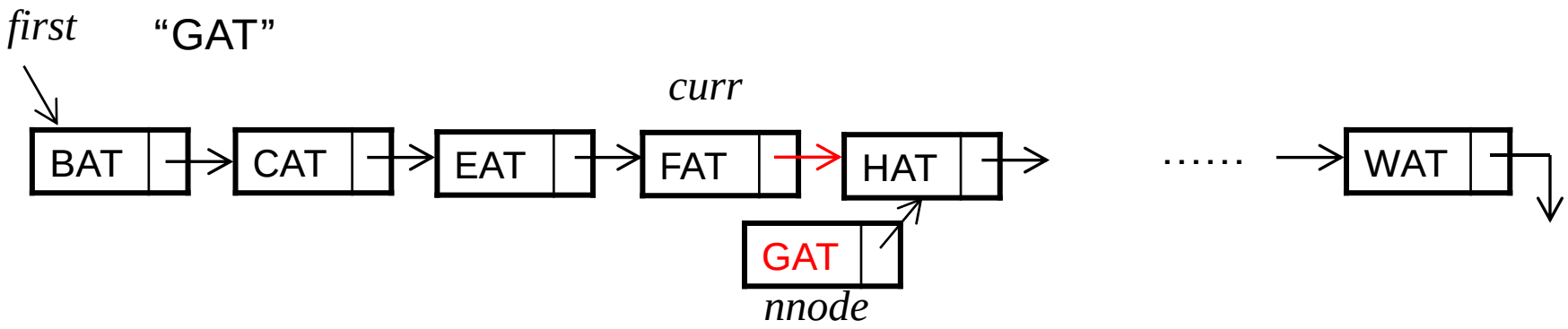
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
    nnode->link = curr->link;
    curr->link = nnode;
}
```



4. Operations of Singly Linked List

(3) Insert

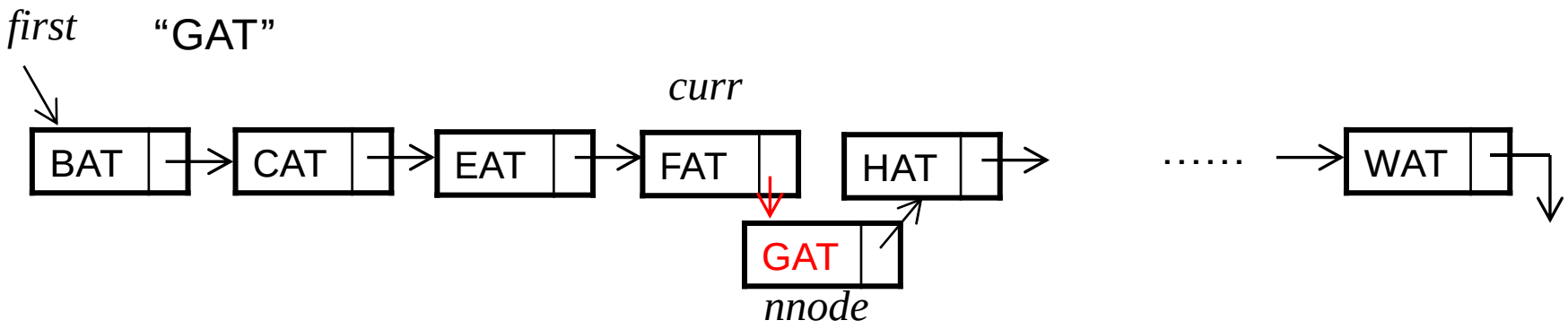
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
    nnode->link = curr->link;
    curr->link = nnode;
}
```



4. Operations of Singly Linked List

(3) Insert

```
void node::insert ( data_type item )
{
//  3. Modify the pointers
    nnode->link = curr->link;
    curr->link = nnode;
}
```



4. Operations of Singly Linked List

(3) Insert

- Degenerate cases?
  - What happens if the item is to be inserted before the first node
  - What happens if `first` points NULL
  - What else?

4. Operations of Singly Linked List

(3) Insert

– Degenerate cases?

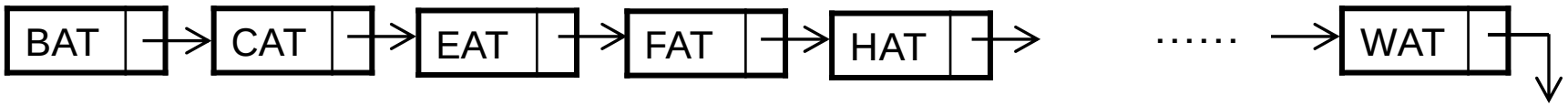
```
void hnode::insert ( data_type item )
{
    // 1. degenerate case 1

    // 2. degenerate case 2

    // ……

    this->link->insert ( item );
}
```

4. Operations of Singly Linked List

(4) Delete

– Delete a node that contains the data item to be deleted

```
void main ( )
{
    first.delete ( “FAT” );
}
```

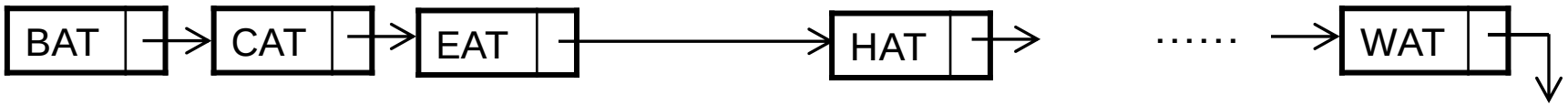


4. Operations of Singly Linked List

(4) Delete

– Delete a node that contains the data item to be deleted

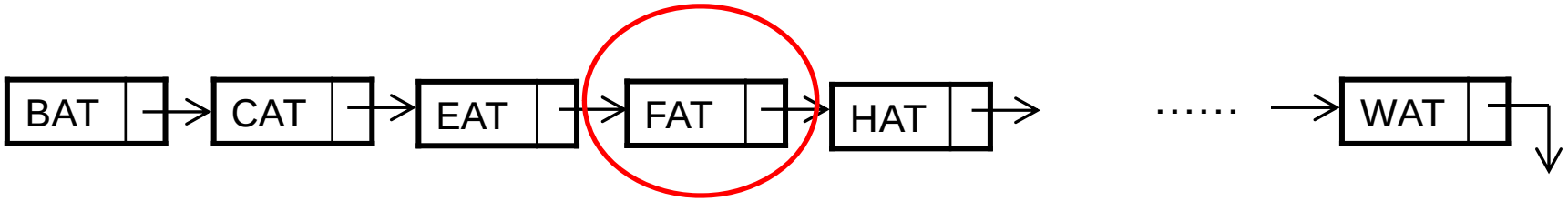
```
void main ( )
{
    first.delete ( “FAT” );
}
```



4. Operations of Singly Linked List

(4) Delete

1. Find a node that contains the item to delete

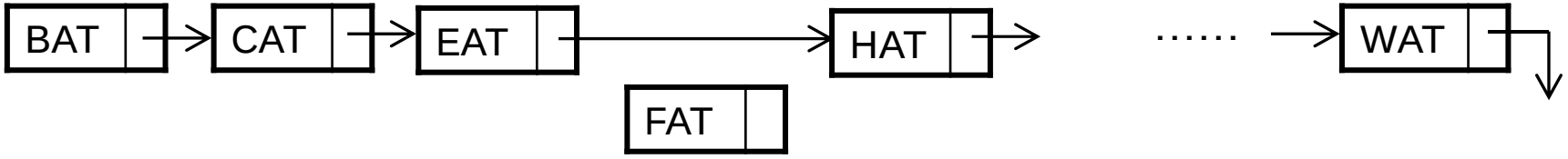


4. Operations of Singly Linked List

(4) Delete

1. Find a node that contains the item to delete

2. Modify the pointers of the list to remove the node



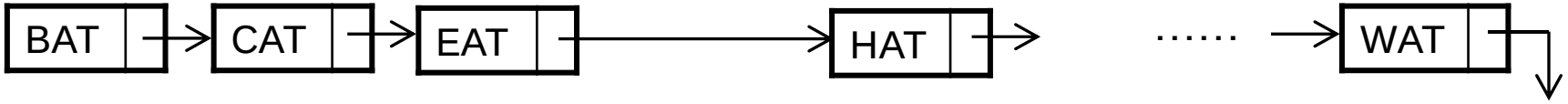
4. Operations of Singly Linked List

(4) Delete

1. Find a node that contains the item to delete

2. Modify the pointers of the list to remove the node

3. Free the deleted node



4. Operations of Singly Linked List

(4) Delete

```
void hnode::delete ( data_type item )
{
    this->link->delete ( item );
}
```

4. Operations of Singly Linked List

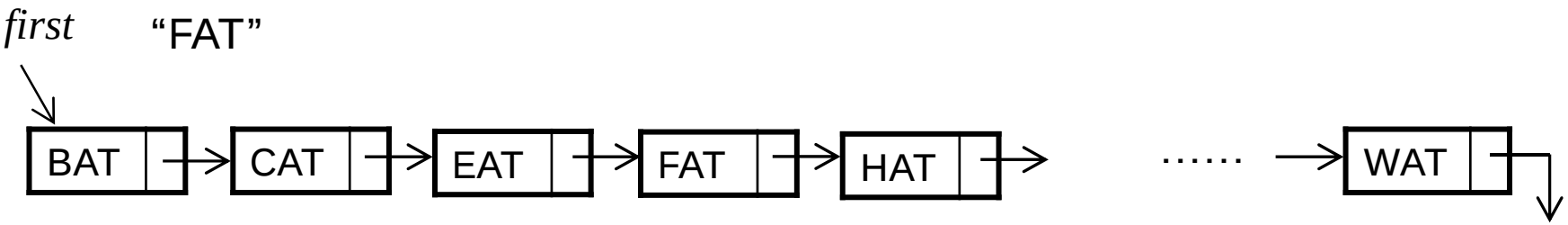
(4) Delete

```
void node::delete ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item == item )
            break;
        curr = curr->link;
    }
//  2. Modify the pointers
    node *dnode = curr->link;
    curr->link = dnode->link;
//  3. Free the deleted node
    free ( dnode );
}
```

4. Operations of Singly Linked List

(4) Delete

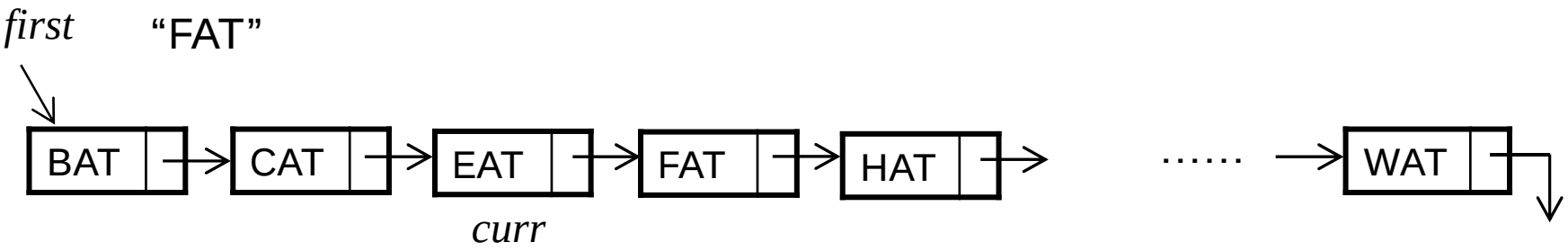
```
void node::delete ( data_type item )
{
//  1. Find a proper position
    node *curr = first;
    while ( curr->link != NULL ) {
        if ( curr->link->item == item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(4) Delete

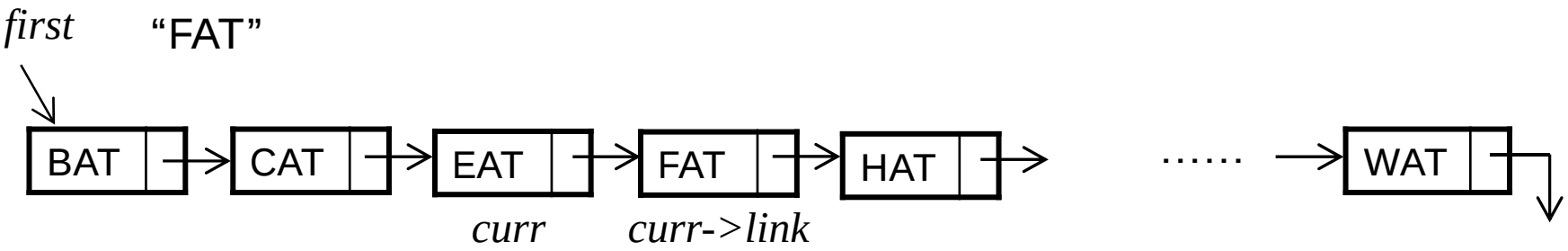
```
void node::delete ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item == item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(4) Delete

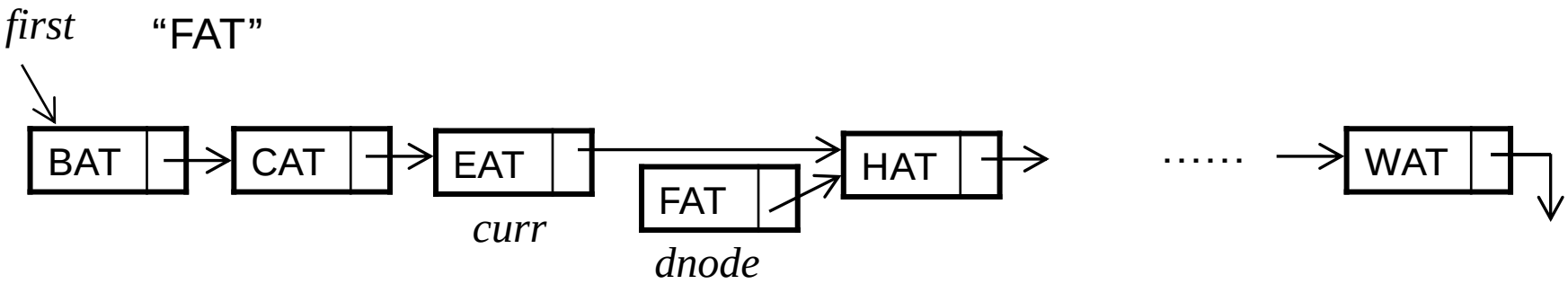
```
void node::delete ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->link != NULL ) {
        if ( curr->link->item == item )
            break;
        curr = curr->link;
    }
}
```



4. Operations of Singly Linked List

(4) Delete

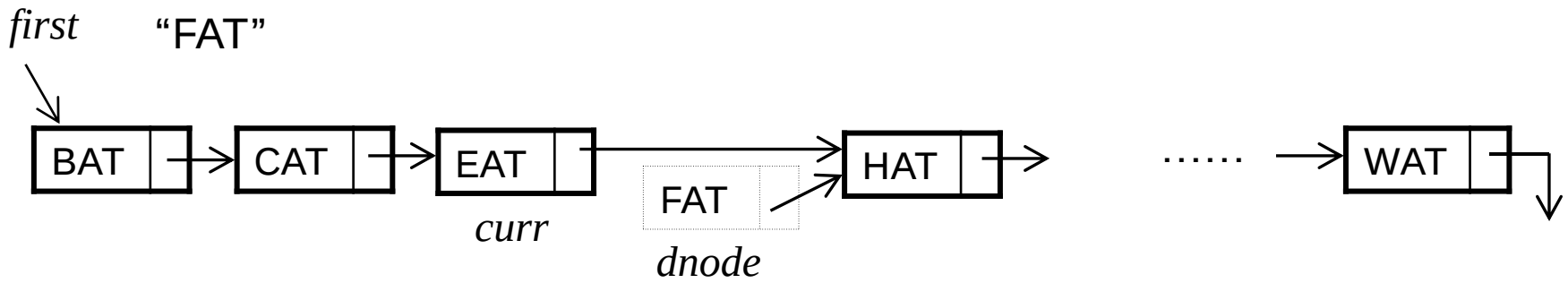
```
void node::delete ( data_type item )
{
//  2. Modify the pointers
    node *dnode = curr->link;
    curr->link = dnode->link;
}
```



4. Operations of Singly Linked List

(4) Delete

```
void node::delete ( data_type item )
{
//  3. Free the deleted node
    free ( dnode );
}
```



4. Operations of Singly Linked List

(4) Delete

- Degenerate cases?
  - What happens if `first` is to delete
  - What happens if `first` is NULL
  - What else?

4. Operations of Singly Linked List

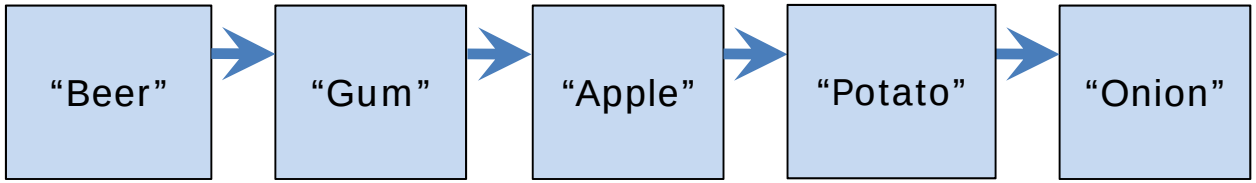
(4) Delete

– Degenerate cases?

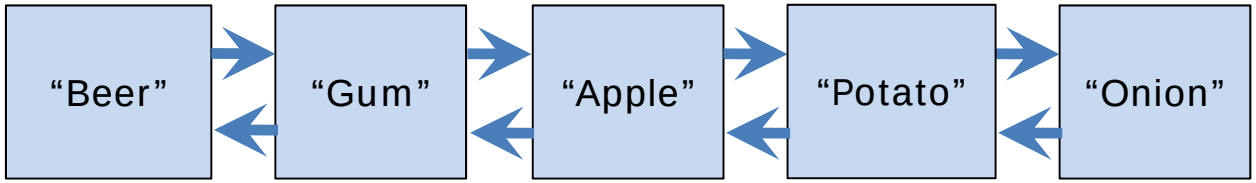
```
void hnode::delete ( data_type item )
{
    // 1. degenerate case 1

    // 2. degenerate case 2

    // ……

    this->link->delete ( item );
}
```

5. Doubly linked list

- Singly-linked list



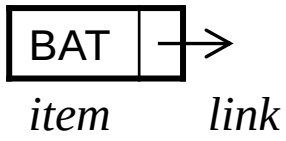
- Doubly-linked list



5. Doubly linked list

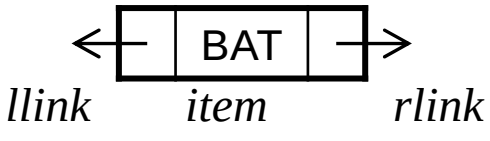
- Singly-linked list

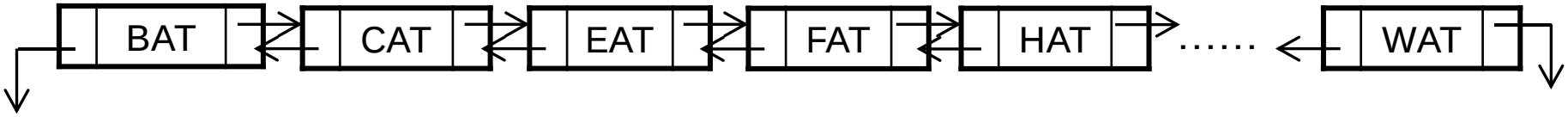
```
class node {
    data_type item;
    node *link;
};
```



- Doubly-linked list

```
class node {
    data_type item;
    node *llink, *rlink;
};
```





6. Operations of Doubly linked list

- Operations

  (1) Insert

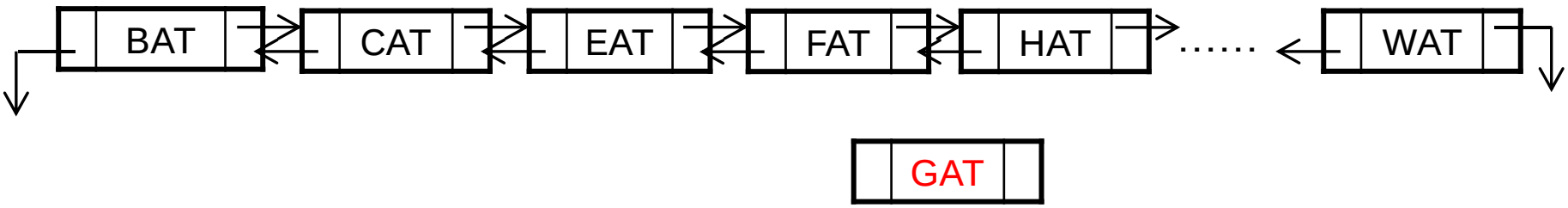
  (2) Delete

6. Operations of Doubly linked list

(1) Insert

- Make a node with a data item to insert
- Add the node at a proper position

```
void main ( )
{
    first.insert ( “GAT” );
}
```

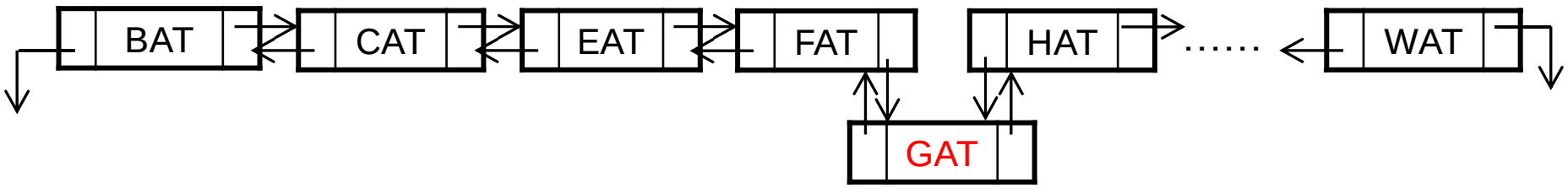


6. Operations of Doubly linked list

(1) Insert

- Make a node with a data item to insert
- Add the node at a proper position

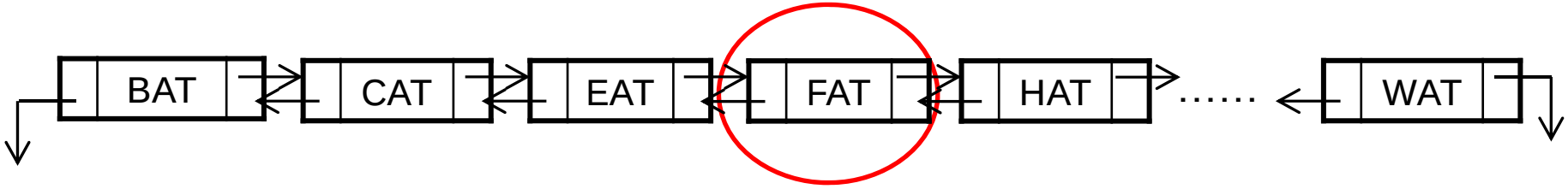
```
void main ( )
{
    first.insert ( “GAT” );
}
```



6. Operations of Doubly linked list

(1) Insert

1. Find a proper position to insert an item

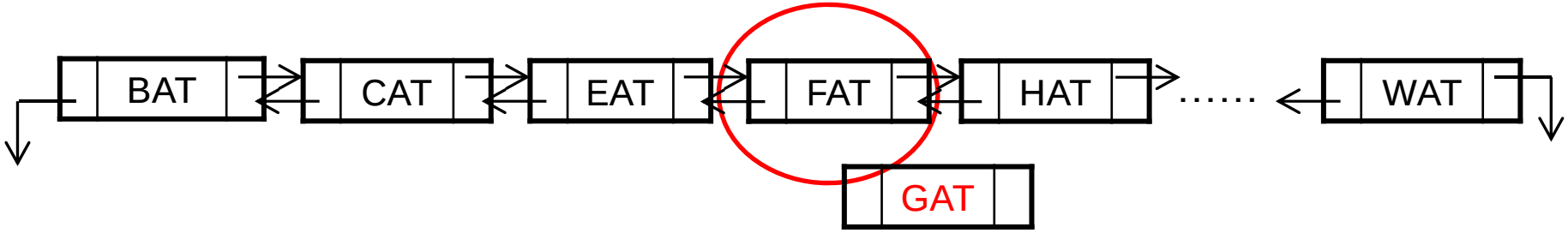


6. Operations of Doubly linked list

(1) Insert

1. Find a proper position to insert an item

2. Build a new node for the item



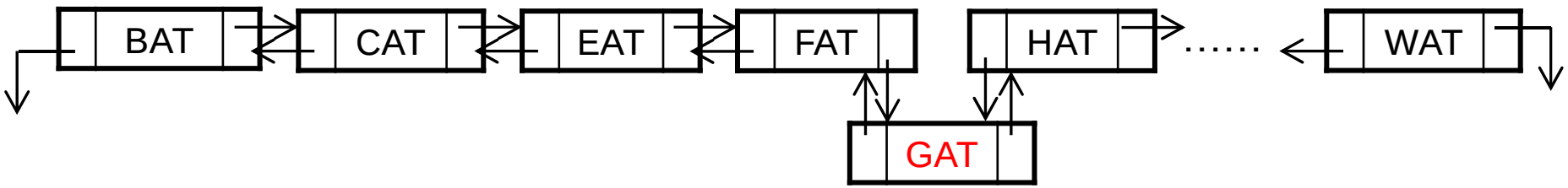
6. Operations of Doubly linked list

(1) Insert

1. Find a proper position to insert an item

2. Build a new node for the item

3. Modify the pointers of the list to contain the new node in the list



6. Operations of Doubly linked list

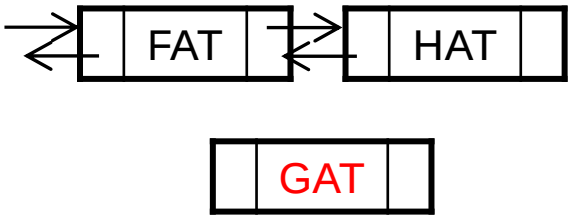
(1) Insert

```
void node::insert ( data_type item )
{
//  1. Find a proper position
    node *curr = this;
    while ( curr->rlink != NULL ) {
        if ( curr->rlink->item > item )
            break;
        curr = curr->rlink;
    }
//  2. Build a new node
    node *nnode = new node;
    nnode->item = item;
//  3. Modify the pointers
}
```

6. Operations of Doubly linked list

(1) Insert

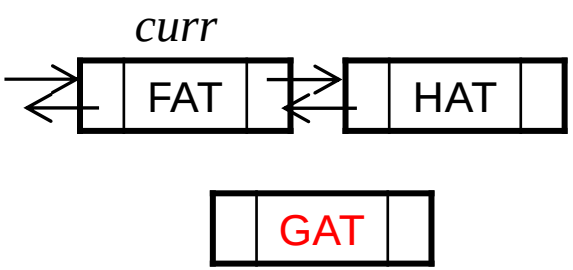
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
}
```



6. Operations of Doubly linked list

(1) Insert

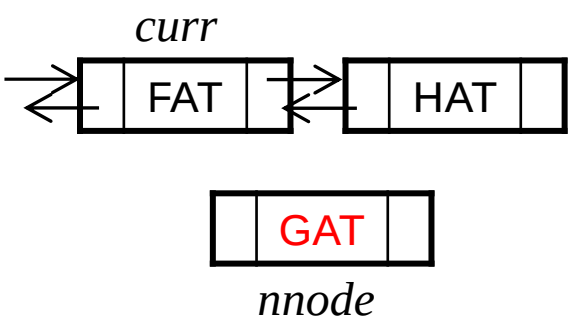
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
}
```



6. Operations of Doubly linked list

(1) Insert

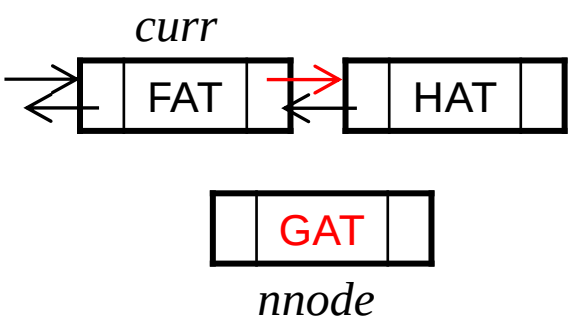
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
}
```



6. Operations of Doubly linked list

(1) Insert

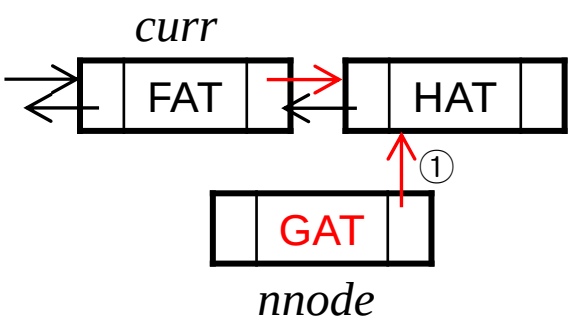
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
//  3.1 forward direction ( rlink: → )
}
```



6. Operations of Doubly linked list

(1) Insert

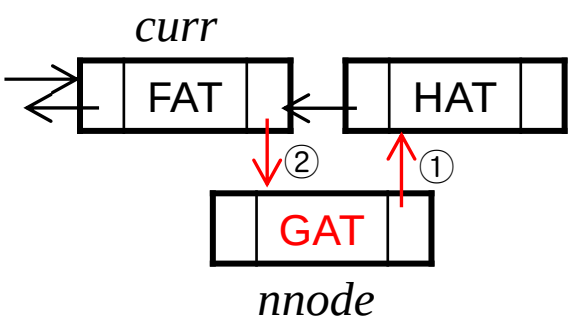
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
//  3.1 forward direction ( rlink: → )
    nnode->rlink = curr->rlink;
}
```



6. Operations of Doubly linked list

(1) Insert

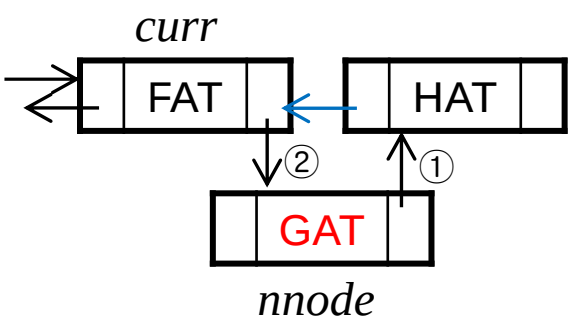
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
//  3.1 forward direction ( rlink: → )
    nnode->rlink = curr->rlink;
    curr->rlink = nnode;
}
```



6. Operations of Doubly linked list

(1) Insert

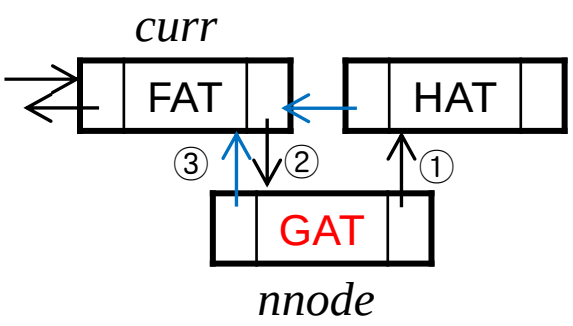
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
//  3.1 forward direction ( rlink: → )
    nnode->rlink = curr->rlink;
    curr->rlink = nnode;
//  3.2 backward direction ( llink: ← )
}
```



6. Operations of Doubly linked list

(1) Insert

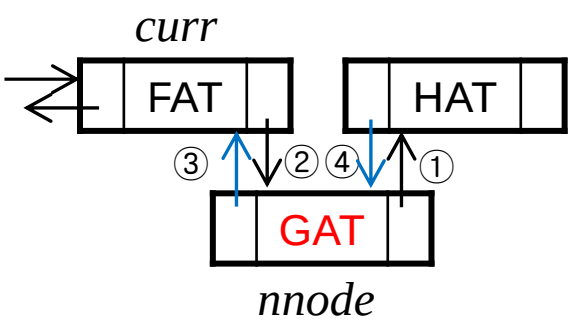
```
void node::insert ( data_type item )
{
//  3. Modify the pointers
//  3.1 forward direction ( rlink: → )
    nnode->rlink = curr->rlink;
    curr->rlink = nnode;
//  3.2 backward direction ( llink: ← )
    nnode->llink = curr;
}
```



6. Operations of Doubly linked list

(1) Insert

```
void node::insert ( data_type item )
{
//  3. Modify the pointers
//  3.1 forward direction ( rlink: → )
    nnode->rlink = curr->rlink;
    curr->rlink = nnode;
//  3.2 backward direction ( llink: ← )
    nnode->llink = curr;
    nnode->rlink->llink = nnode;
}
```



6. Operations of Doubly linked list

(1) Insert

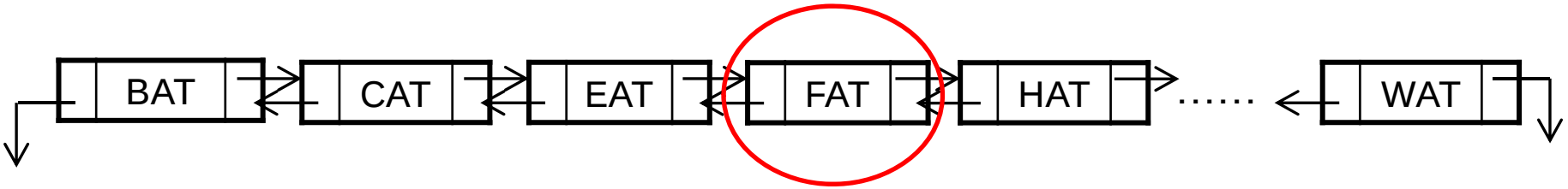
- Degenerate cases?
  - What happens if the item is to be inserted before `first`
  - What happens if `first` is NULL
  - What else?

6. Operations of Doubly linked list

(2) Delete

– Delete a node that contains the data item to be deleted

```
void main ( )
{
    first.delete ( “FAT” );
}
```

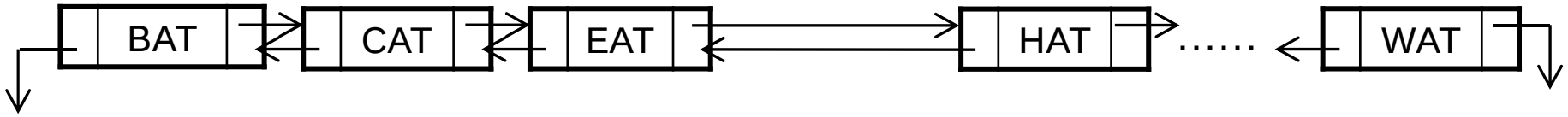


6. Operations of Doubly linked list

(2) Delete

– Delete a node that contains the data item to be deleted

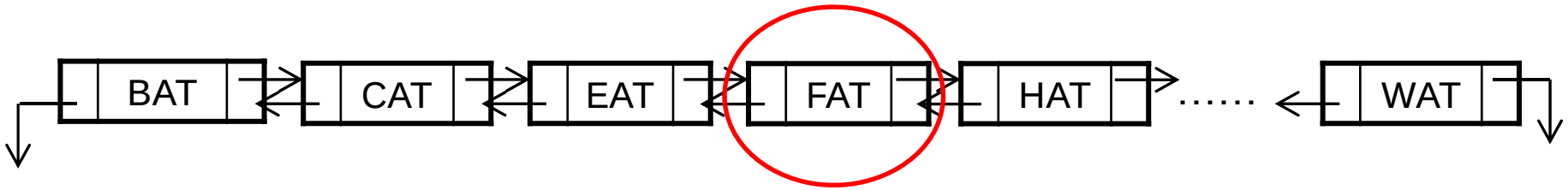
```
void main ( )
{
    first.delete ( “FAT” );
}
```



6. Operations of Doubly linked list

(2) Delete

1. Find a node that contains the item to delete

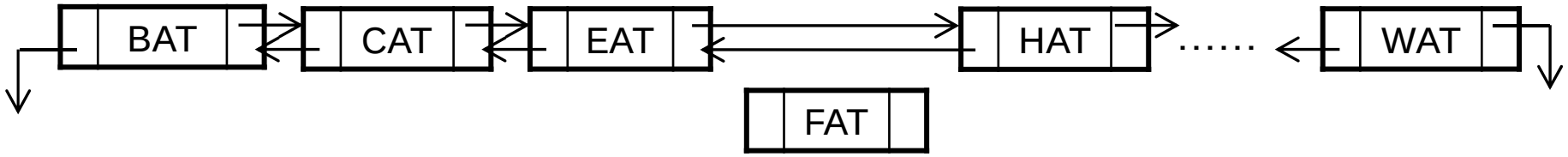


6. Operations of Doubly linked list

(2) Delete

1. Find a node that contains the item to delete

2. Modify the pointers of the list to remove the node



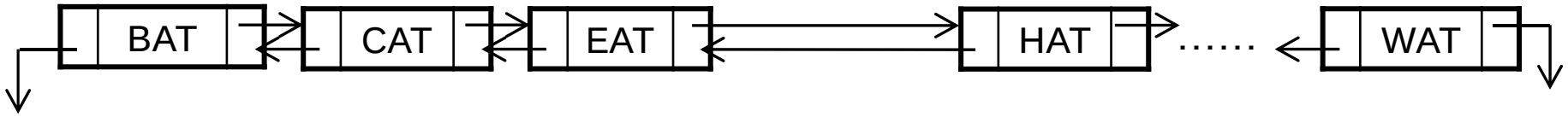
6. Operations of Doubly linked list

(2) Delete

1. Find a node that contains the item to delete

2. Modify the pointers of the list to remove the node

3. Return the deleted node



6. Operations of Doubly linked list

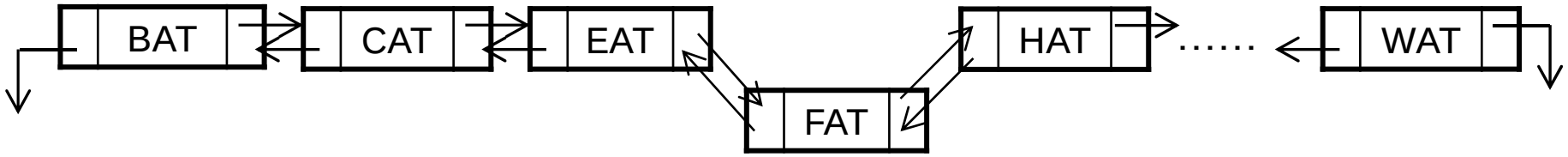
(2) Delete

```
void node::delete ( data_type item )
{
//  1. Find a proper position
    node *curr = first;
    while ( curr->rlink != NULL ) {
        if ( curr->rlink->item == item )
            break;
        curr = curr->rlink;
    }
//  2. Modify the pointers

//  3. Return the deleted node
    free ( dnode );
}
```

6. Operations of Doubly linked list

(2) Delete

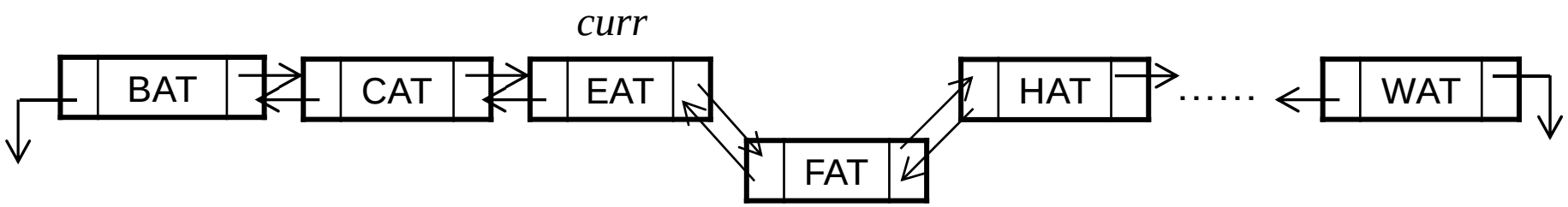
```
void node::delete ( data_type item )
{
//  2. Modify the pointers
    node *dnode = curr->rlink;
}
```



6. Operations of Doubly linked list

(2) Delete

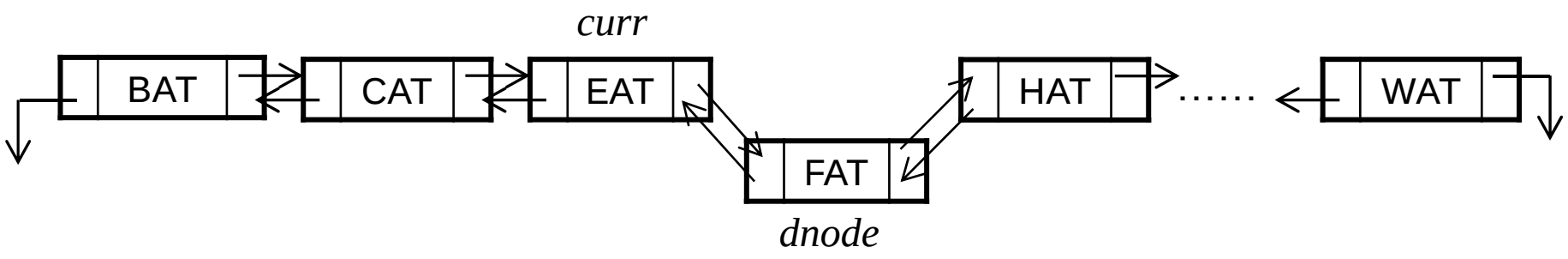
```
void node::delete ( data_type item )
{
//  2. Modify the pointers
    node *dnode = curr->rlink;
}
```



6. Operations of Doubly linked list

(2) Delete

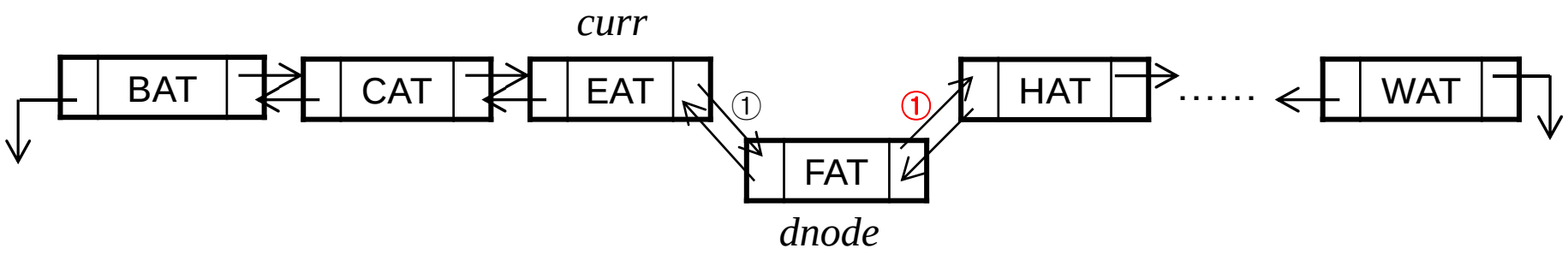
```
void node::delete ( data_type item )
{
//  2. Modify the pointers
    node *dnode = curr->rlink;
}
```



6. Operations of Doubly linked list

(2) Delete

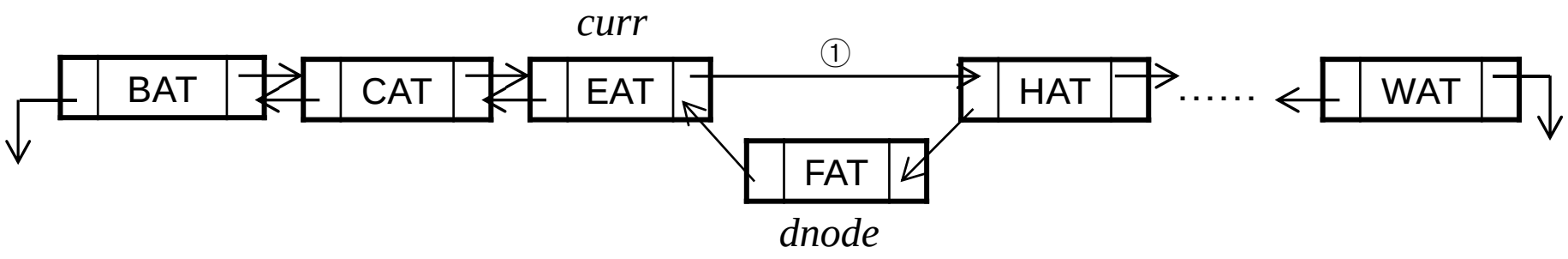
```
void node::delete ( data_type item )
{
//  2. Modify the pointers
    node *dnode = curr->rlink;
    curr->rlink = dnode->rlink;
}
```



6. Operations of Doubly linked list

(2) Delete

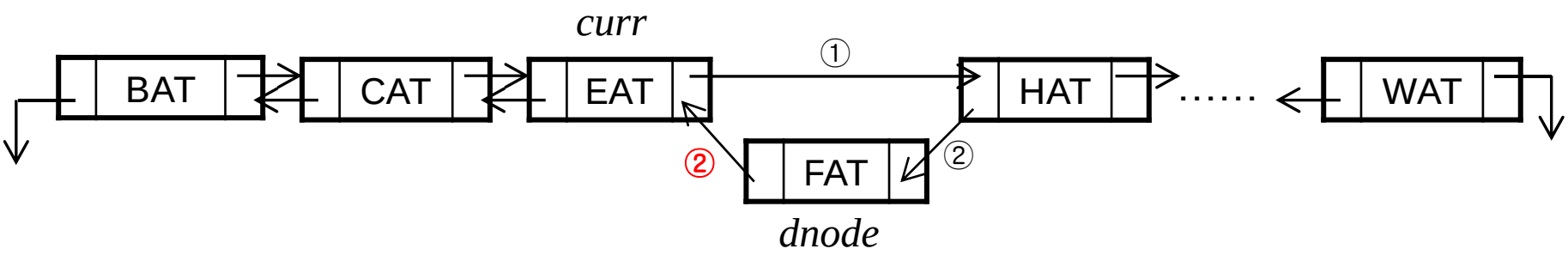
```
void node::delete ( data_type item )
{
//  2. Modify the pointers
    node *dnode = curr->rlink;
    curr->rlink = dnode->rlink;
}
```



6. Operations of Doubly linked list

(2) Delete

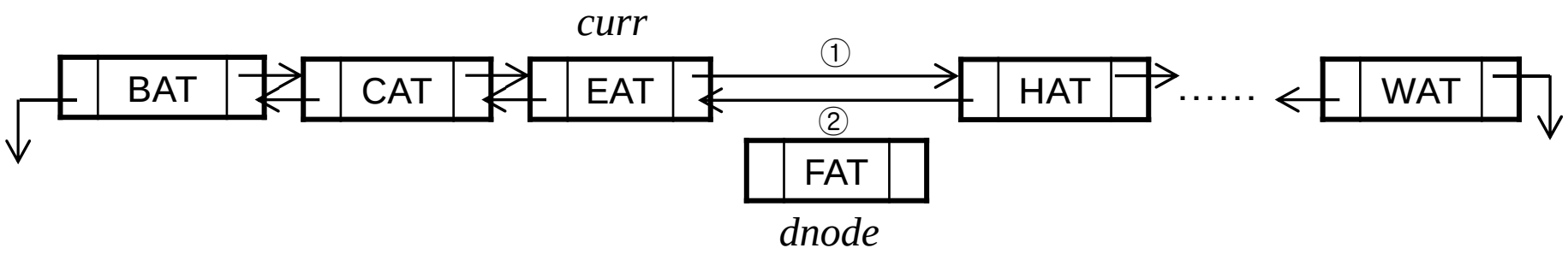
```
void node::delete ( data_type item )
{
//  2. Modify the pointers
    node *dnode = curr->rlink;
    curr->rlink = dnode->rlink;
    curr->rlink->llink = dnode->llink;
}
```



6. Operations of Doubly linked list

(2) Delete

```
void node::delete ( data_type item )
{
//  2. Modify the pointers
    node *dnode = curr->rlink;
    curr->rlink = dnode->rlink;
    curr->rlink->llink = dnode->llink;
}
```



6. Operations of Doubly linked list

(2) Delete

- Degenerate cases?
  - What happens if `first` is to delete
  - What happens if `first` is NULL
  - What else?

7. Performance

- Time complexity

Operation	Singly linked list	Doubly linked list
search	O (n)	O (n)
insert	O (n)	O (n)
delete	O (n)	O (n)

Summing up (only for sorted list)

- Search

List		search

array	Linear search	`for ( int i = 0; i < n; i++ )`
		`  if ( A[i] == x )`
		`  return i;`

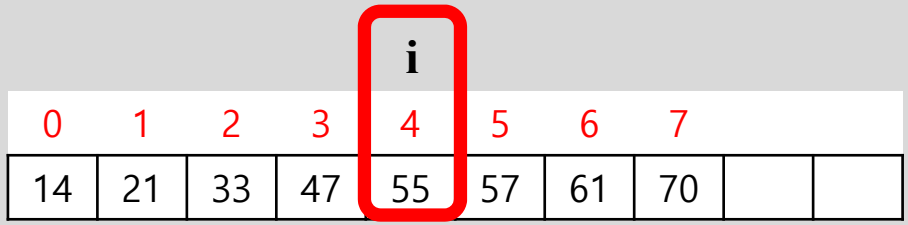
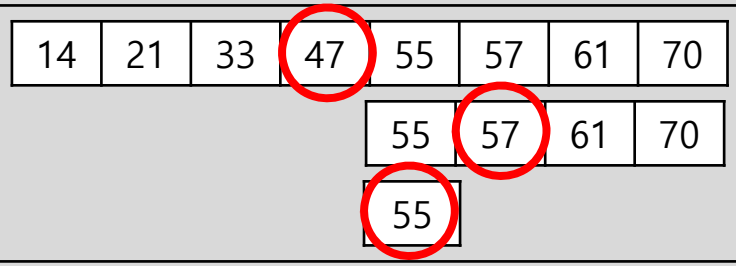
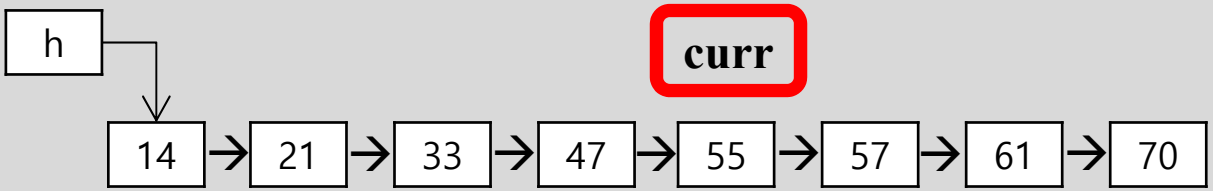
	Binary search	`int mid = (s + e)/2;`
		`if (x == A[mid])  return mid;`
		`else if (x < A[mid])  return bs ( A, s, mid-1, x );`
		`else  return bs ( A, mid+1, e, x );`

singly linked list	Linear search	`node *curr = this;`
		`while ( curr != NULL ) {`
		`  if ( curr->item == item )`
		`  return curr;`
		`  curr = curr->link;`
		`}`

doubly linked list		`for ( node *curr = this; curr != NULL; curr = curr->link )`
		`  if ( curr->item == item )`
		`  return curr;`

Summing up (only for sorted list)

- Search

List		search 55
array	Linear search	
	Binary search	
singly linked list	Linear search	
doubly linked list		

Summing up (only for sorted list)

- insert (insert_by_value)

list	insert

array	`// 1. Find a proper position`
	`for ( i = 0; i < n; i++ )`
	`  if ( A[i] > x )  break;`
	
	`// 2. Move the items to the next positions ( j → j+1 )`
	`for ( j = n-1; j >= i; j-- )`
	`  A[j+1] = A[j];`
	
	`// 3. Insert the item`
	`A[i] = x;`
	`n++;`

singly linked list	`// 1. Find a proper position`
	`for ( node *curr = this; curr->link != NULL; curr = curr->link )`
	`  if ( curr->link->item > item )  break;`
	
	`// 2. Build a new node`
	`node *nnode = new node;`
	`nnode->item = item;`
	
	`// 3. Modify the pointers`
	`nnode->link = curr->link;`
	`curr->link = nnode;`

Summing up (only for sorted list)

list	insert 50
array	// 1. Find a proper position i 14 | 21 | 33 | 47 | 55 | 57 | 61 | 70 | | // 2. Move the items to the next positions (j → j+1) i 14 | 21 | 33 | 47 | 55 | 55 | 57 | 61 | 70 | // 3. Insert the item 14 | 21 | 33 | 47 | **50** | 55 | 57 | 61 | 70 |
singly linked list	// 1. Find a proper position curr 14 → 21 → 33 → 47 → 55 → 57 → 61 → 70 // 2. Build a new node curr 14 → 21 → 33 → 47 → 55 → 57 → 61 → 70 50 // 3. Modify the pointers curr 14 → 21 → 33 → 47 55 → 57 → 61 → 70 50

Summing up (only for sorted list)

- delete (delete_by_value)

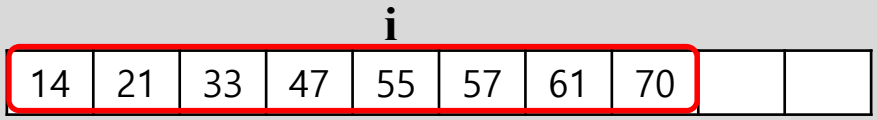
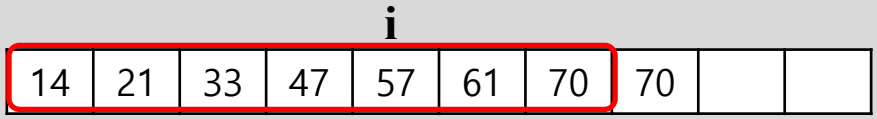
list	delete

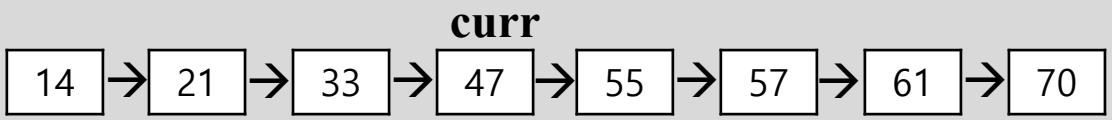
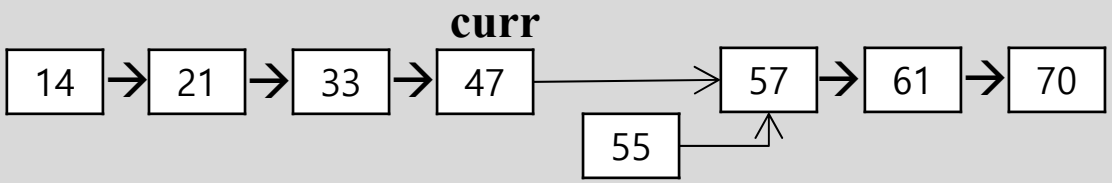
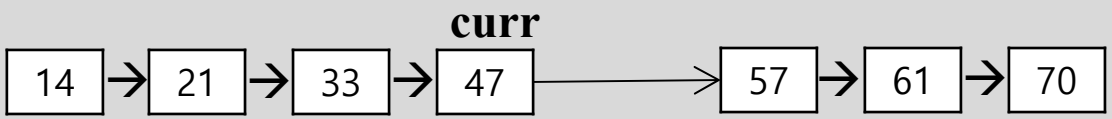
array	`//  1. Find a proper position`
	`    for ( int i = 0; i < n; i++ )`
	`        if ( A[i] == x )      break;`
	`    if ( i == n )    return;`
	
	`//  2. Move the items to the previous positions ( j+1 → j )`
	`    for ( int j = i; j < n-1; j++ )`
	`        A[j] = A[j+1];`
	`    n--;`

singly linked list	`//  1. Find a proper position`
	`    for ( node *curr = this; curr->link != NULL; curr = curr->link )`
	`        if ( curr->link->item == item )     break;`
	
	`//  2. Modify the pointers`
	`    node *dnode = curr->link;`
	`    curr->link = dnode->link;`
	
	`//  3. Free the deleted node`
	`    free ( dnode );`

Summing up (only for sorted list)

list	delete 55

array	`//  1. Find a proper position`
	 
	
	`//  2. Move the items to the previous positions ( j+1 → j )`
	 

singly linked list	`//  1. Find a proper position`
	 
	
	`//  2. Modify the pointers`
	 
	
	`//  3. Free the deleted node`
	 

Contents

Contents