4/4

Linked List

Data Structures

C++ for C Coders

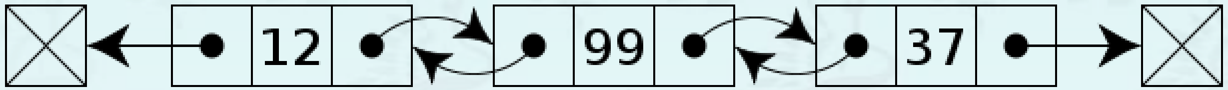
한동대학교 김영섭 교수

idebtor@gmail.com

Doubly Linked List with sentinel nodes

Doubly Linked lists

❖ **Doubly linked list:** each node contains, besides the **next**-node link, a second link field pointing to the ***previous*** node in the sequence. The two links may be called **forward** and **backward**, or **next** and **prev**(**ious**).



❖ **Type definition**

```
struct Node {
    int      item;
    Node*    prev;
    Node*    next;
};
using pNode = Node*;
```

Q. Doubly linked list, Why?

Doubly Linked lists

Q. Array vs. Singly linked list vs. Doubly linked list, Why?

Advantages of linked list:

- Dynamic structure (Memory Allocated at run-time)
- Have more than one data type.
- Re-arrange of linked list is easy (Insertion-Deletion).
- <u>It doesn't waste memory.</u>

Disadvantages of linked list:

- In linked list, if we want to access any node it is difficult.
- <u>It is occupying more memory.</u>

Advantages of doubly linked list:

- A doubly linked list can be **traversed in both directions** (forward and backward). A singly linked list can only be traversed in one direction.
- **most operations are O(1)** instead of O(n)

doubly linked list with sentinel nodes

```
void erase(pNode x){
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```
void pop(pList p, int val){
  erase(find(p, val));
}
```

This code may not work some cases.
How can you fix it?

doubly linked list with sentinel nodes

```
void erase(pNode x){
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```
void pop(pList p, int val){
  erase(find(p, val));
}
```

This code may not work some cases.
How can you fix it?

```
pNode find(pList p, int val){
  pNode curr = begin(p);
  while(curr != end(p)) {
    if (curr->item == val) return curr;
    curr = curr->next;
  }
  return curr;
}
```

```
pNode find(pList p, int val){
  pNode x = begin(p);
  for (; x != end(p); x = x->next;)
    if (x->item == val) return x;
  return x;
}
```

doubly linked list with sentinel nodes

```
void erase(pNode x){
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```
void pop(pList p, int val){
  erase(find(p, val));
}
```

This code may not work some cases.
How can you fix it?

```
void pop(pList p, int val){
  pNode node = find(p, val);
  if (node == p->tail || node == p->head) return;
  erase(node);
}
```

doubly linked list with sentinel nodes

```
void erase(pNode x){
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```
void erase(pList p, pNode x){
  if (x == p->tail || x == p->head) return;
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```
void pop(pList p, int val){
  erase(find(p, val));
}
```

This code may not work some cases.
How can you fix it?

```
void pop(pList p, int val){
  pNode node = find(p, val);
  if (node == p->tail || node == p->head) return;
  erase(node);
}
```

doubly linked list with sentinel nodes

```
void erase(pNode x){
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```
void pop(pList p, int val){
  erase(find(p, val));
}
```

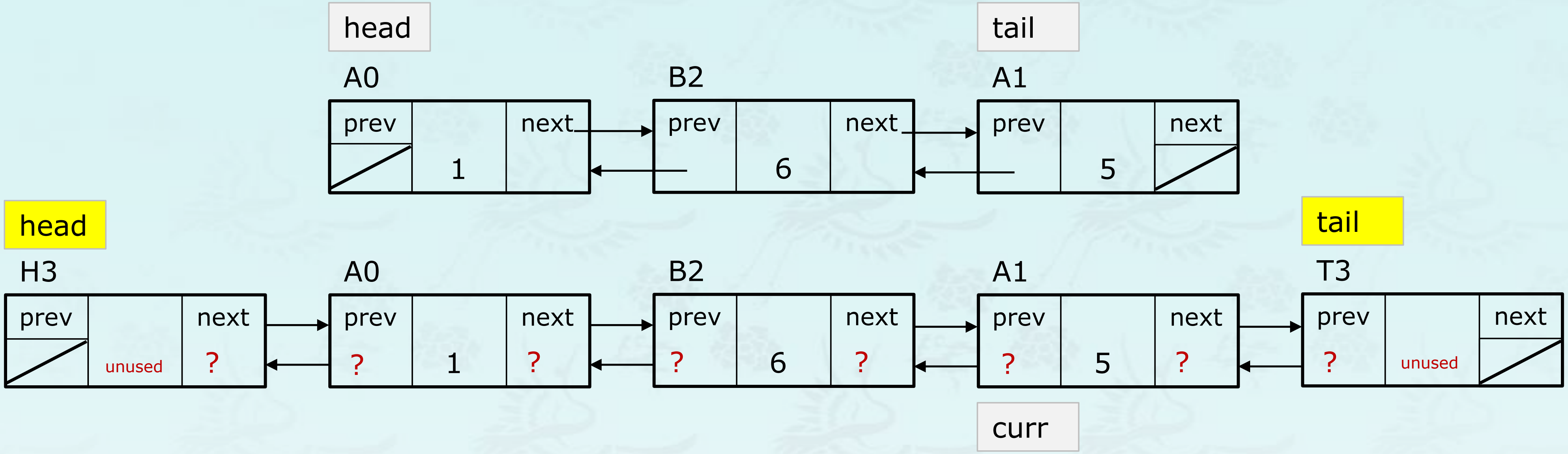
This code may not work some cases.
How can you fix it?

```
void erase(pList p, pNode x){
  if (x == p->tail || x == p->head) return;
  x->prev->next = x->next;
  x->next->prev = x->prev;
  delete x;
}
```

```
void pop(pList p, int val){
  erase(p, find(p, val));
}
```

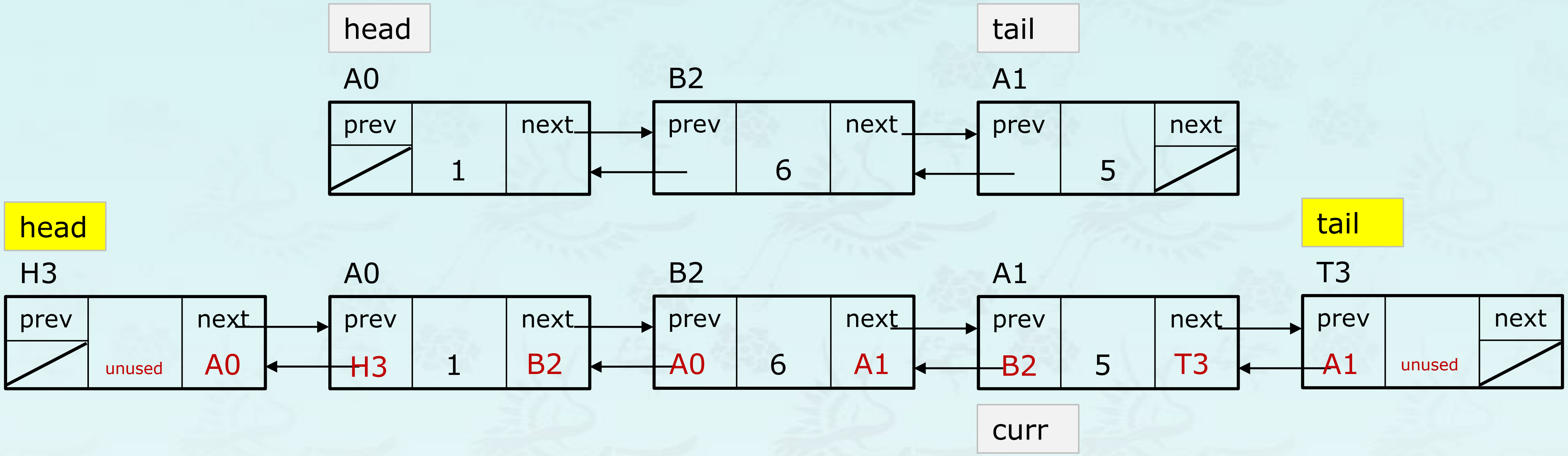
```
void pop(pList p, int val){
  pNode node = find(p, val);
  if (node == p->tail || node == p->head) return;
  erase(node);
}
```

Linked list **with sentinel nodes**

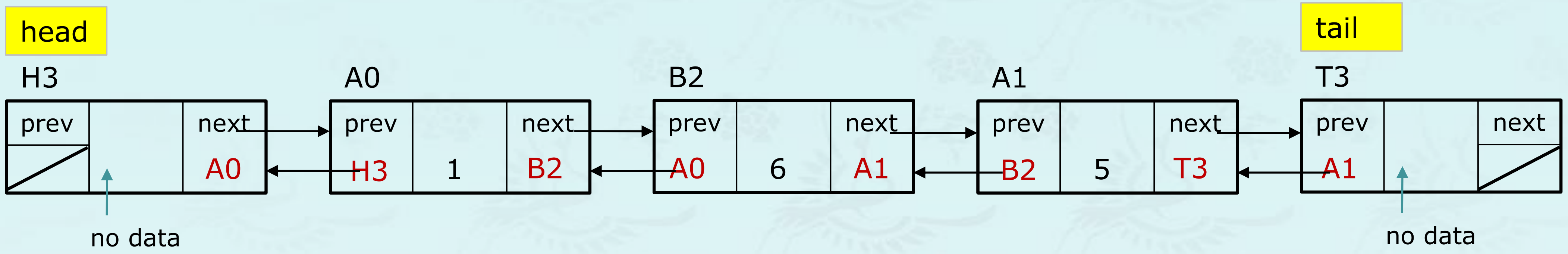


Fill the blanks(?) with mnemonics to form a linked list as shown.

doubly linked list **with sentinel nodes**



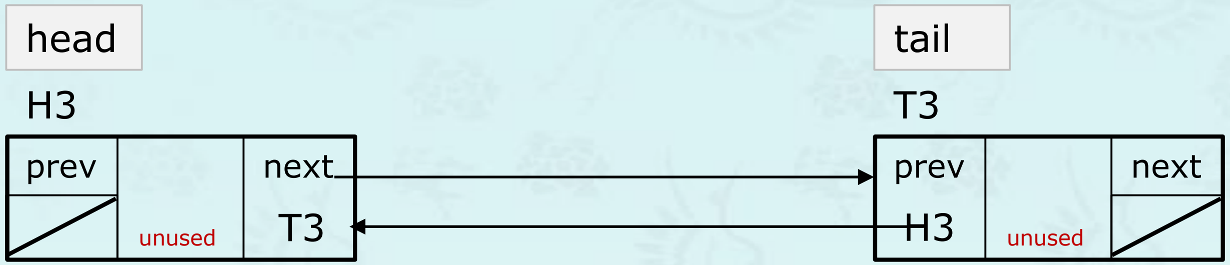
doubly linked list **with sentinel nodes**



- These extra nodes are known as **sentinel nodes.** The node at the front is known as **head node**, and the node at the end as a **tail node**. The head and tail nodes are created when the doubly linked list is initialized. The purpose of these nodes is to simply the insert, push/pop front and back, remove methods by eliminating all need for special-case code when the list empty, or when we insert at the head or tail of the list. This would **greatly** simplify the coding **unbelievably**.

- For instance, if we do not use a head node, then removing the first node becomes a special case, because we must reset the list's link to **the first node** during the remove and because the remove algorithm in general needs to access the node prior to the node being removed (and without a head node, the first node does not have a node prior to it).

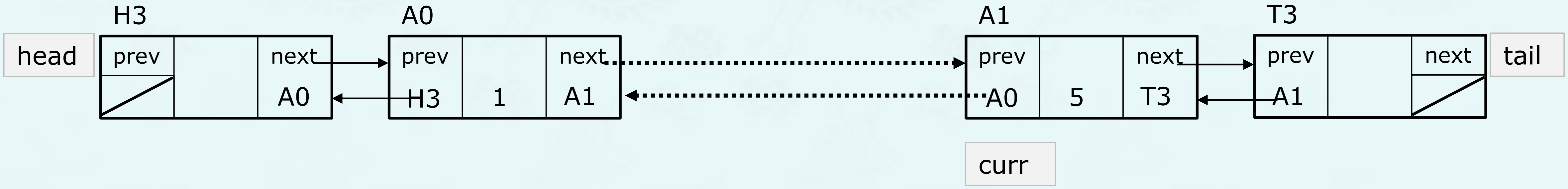
doubly linked list



An **empty** doubly linked list with sentinel nodes

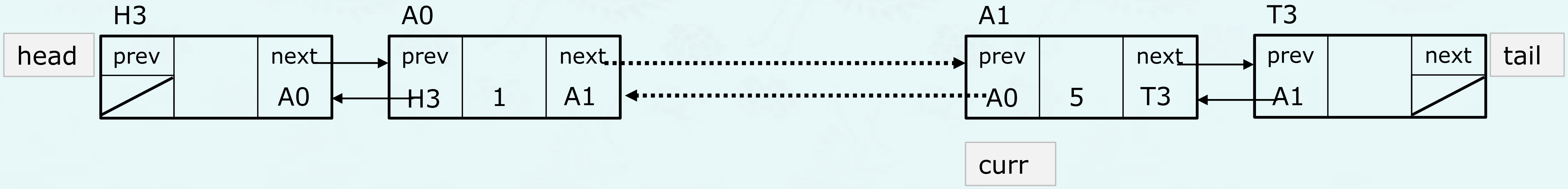
doubly linked list – begin() and end()

```
// returns the first node which list::head points to in the container.
pNode begin(pList p) {

}
```



doubly linked list – begin() and end()

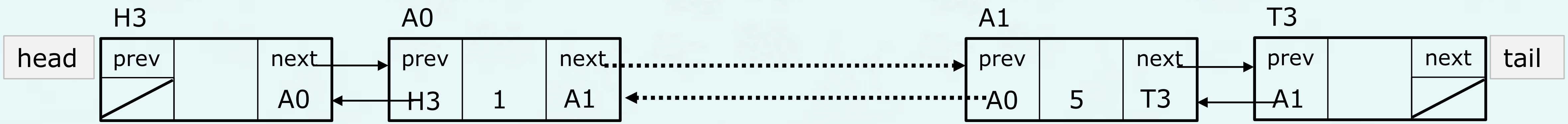
```
// returns the first node which list::head points to in the container.
pNode begin(pList p) {
  return p->head->next;
}
```



doubly linked list – begin() and end()

```
// returns the first node which list::head points to in the container.
pNode begin(pList p) {
  return p->head->next;
}
```

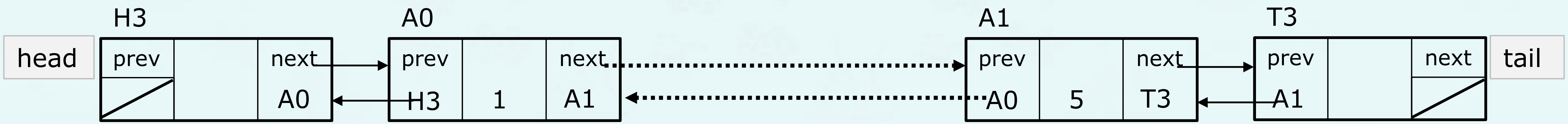
```
// returns the tail node referring to the past -the last- node in the list.
// The past -the last- node is the sentinel node which is used only as a sentinel
// that would follow the last node. It does not point to any node next, and thus
// shall not be dereferenced. Because the way we are going use during the iteration,
// we don't want to include the node pointed by this. this function is often used
// in combination with List::begin to specify a range including all the nodes in
// the list. This is a kind of simulated used in STL. If the container is empty,
// this function returns the same as List::begin.
pNode end(pList p) {

}
```



doubly linked list – begin() and end()

```
// returns the first node which list::head points to in the container.
pNode begin(pList p) {
  return p->head->next;
}
```

```
// returns the tail node referring to the past -the last- node in the list.
// The past -the last- node is the sentinel node which is used only as a sentinel
// that would follow the last node. It does not point to any node next, and thus
// shall not be dereferenced. Because the way we are going use during the iteration,
// we don't want to include the node pointed by this. this function is often used
// in combination with List::begin to specify a range including all the nodes in
// the list. This is a kind of simulated used in STL. If the container is empty,
// this function returns the same as List::begin.
pNode end(pList p) {
(1)  return p->tail;
(2)  return p->tail->next;
}
```

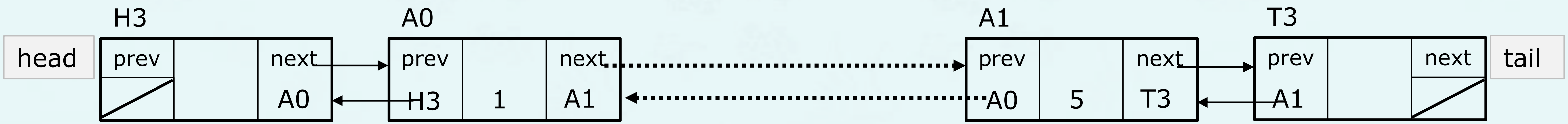


doubly linked list – begin() and end()

```
// returns the first node which list::head points to in the container.
pNode begin(pList p) {
  return p->head->next;
}
```

With the container given below, what does begin(p) and end(p) return respectively?
Answer in mnemonic:

```
// returns the tail no
// The past -the last-                                                    nel
// that would follow the last node. It does not point to any node next, and thus
// shall not be dereferenced. Because the way we are going use during the iteration,
// we don't want to include the node pointed by this. this function is often used
// in combination with List::begin to specify a range including all the nodes in
// the list. This is a kind of simulated used in STL. If the container is empty,
// this function returns the same as List::begin.
pNode end(pList p) {
  return p->tail;           // not tail->next
}
```



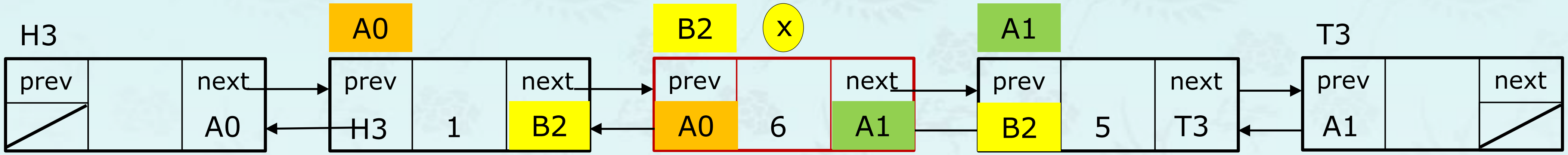
doubly linked list - **erase**

```
void erase(pNode x);
```

The node B2 is to be erased or removed.

- Let us supposed B2 is removed. Then,
- Which nodes are changed and where?

before:

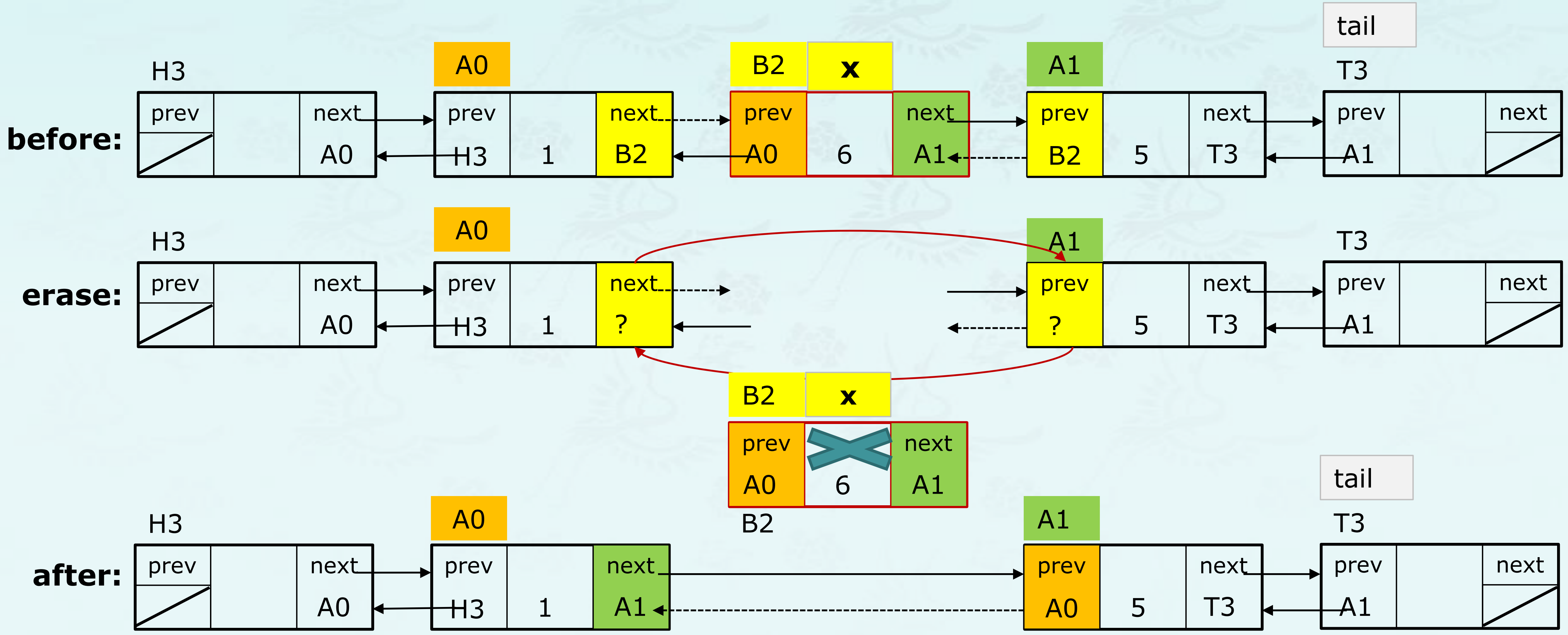


doubly linked list - erase

```
void erase(pNode x);
```

The node B2 is to be erased or removed.
- Let us supposed B2 is removed. Then,
- Which nodes are changed and where?

Express it in mnemonically or using node addresses (H3, A0, B2, A1 etc.):
- B2 at A0→next should be A1;
- B2 at A1→prev should be A0;



doubly linked list - erase

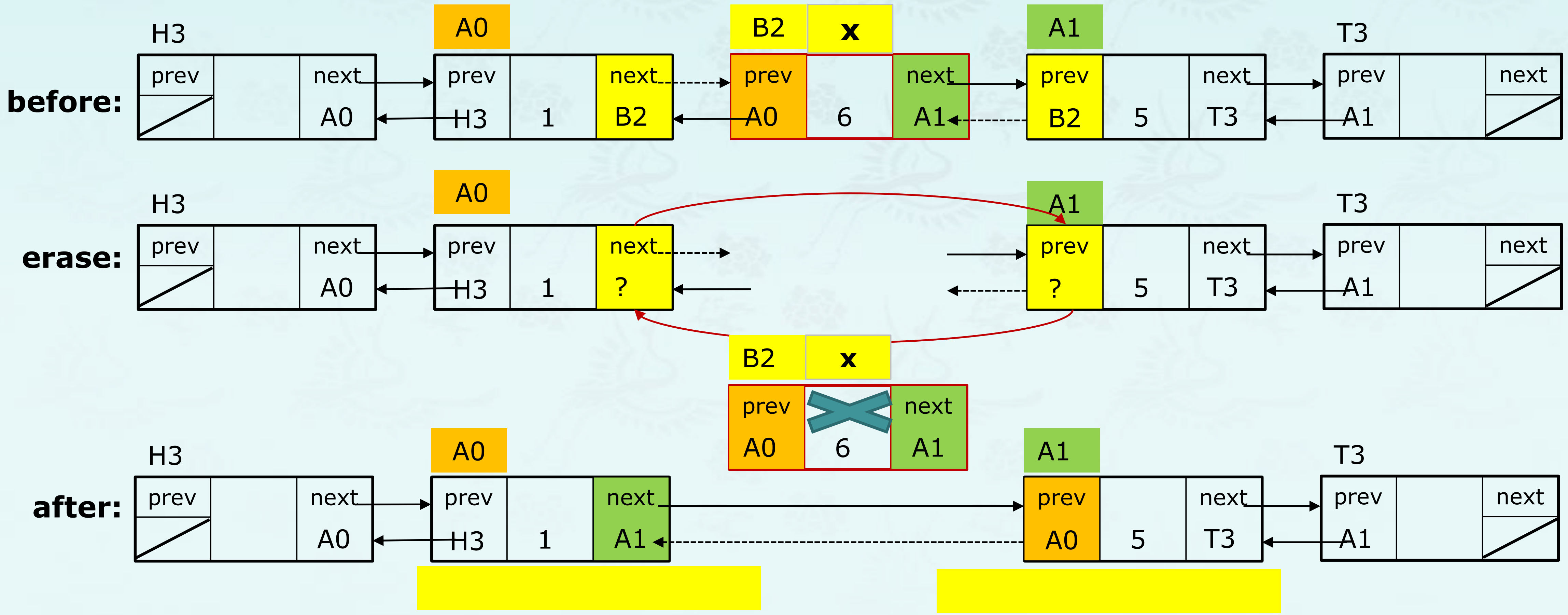
```
void erase(pNode x);
```

The node B2 is to be erased or removed.

- Let us supposed B2 is removed. Then,
- Which nodes are changed and where?

Express it in mnemonically or using node addresses (H3, A0, B2, A1 etc.):

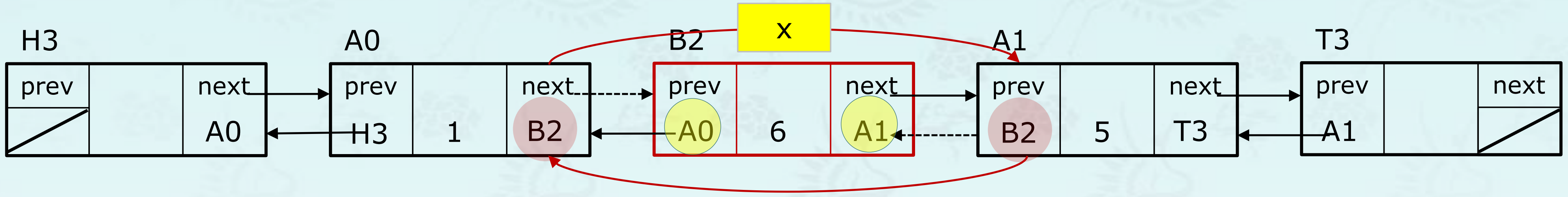
- B2 at A0→next should be A1;
- B2 at A1→prev should be A0;



doubly linked list - erase

The node B2 is to be erased or removed.
- Let us supposed B2 is removed. Then,
- Which nodes are changed and where?

Express it in mnemonically or using node addresses (H3, A0, B2, A1 etc.):
- B2 at A0→next should be A1;
- B2 at A1→prev should be A0;



In coding:

```
void erase(pNode x) {
    x->prev->next = x->next;
    x->next->prev = x->prev;
    delete x;
}
```

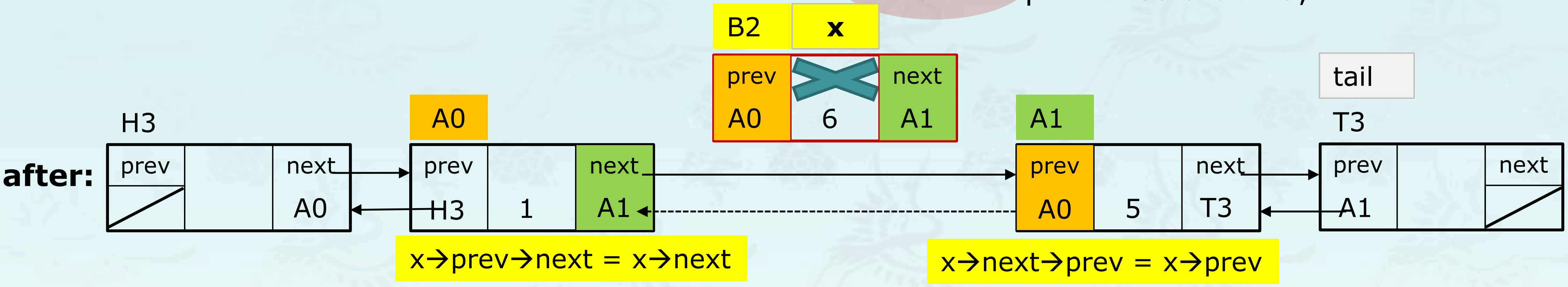
Rewrite the following mnemonics in coding while x is given:
- A0 : x→prev
- A1 : x→next
- B2 : x

Check the values these pointers points.
- x→prev→next : B2
- x→next→prev : B2

doubly linked list - erase

The node B2 is to be erased or removed.
- Let us supposed B2 is removed. Then,
- Which nodes are changed and where?

Express it in mnemonically or using node addresses (H3, A0, B2, A1 etc.):
- B2 at A0→next should be A1;
- B2 at A1→prev should be A0;



In coding:

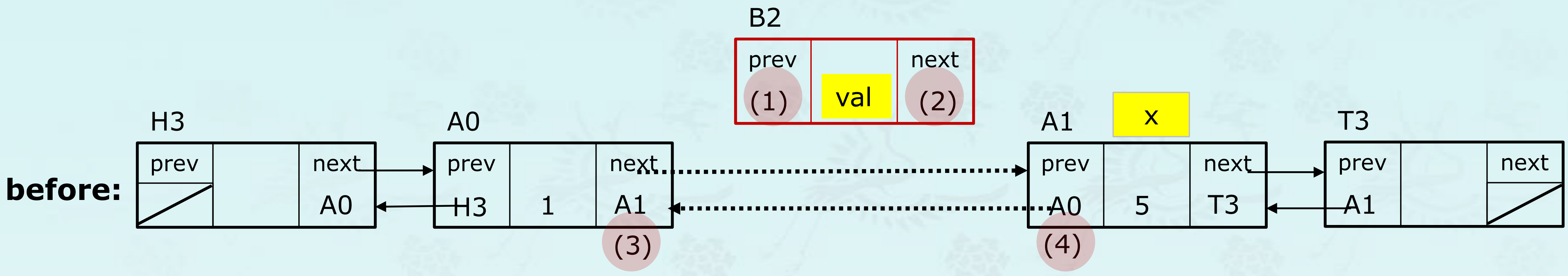
```
void erase(pNode x) {
    x->prev->next = x->next;
    x->next->prev = x->prev;
    delete x;
}
```

Rewrite the following mnemonics in coding while x is given:
- A0 : x→prev
- A1 : x→next
- B2 : x

Check the values these pointers points.
- x→prev→next : B2
- x→next→prev : B2

doubly linked list – insert()

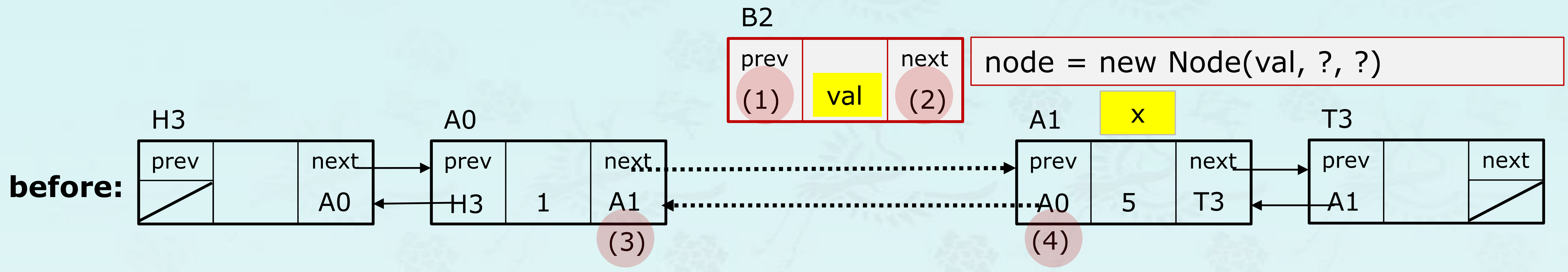
```
void insert(pNode x, int val)
```



Express (1) ~ (4) in mnemonics and real code:
(1) B2→prev
(2) B2→next
(3) A0→next or A1
(4) A1→prev or A0

doubly linked list – insert()

```
void insert(pNode x, int val)
```

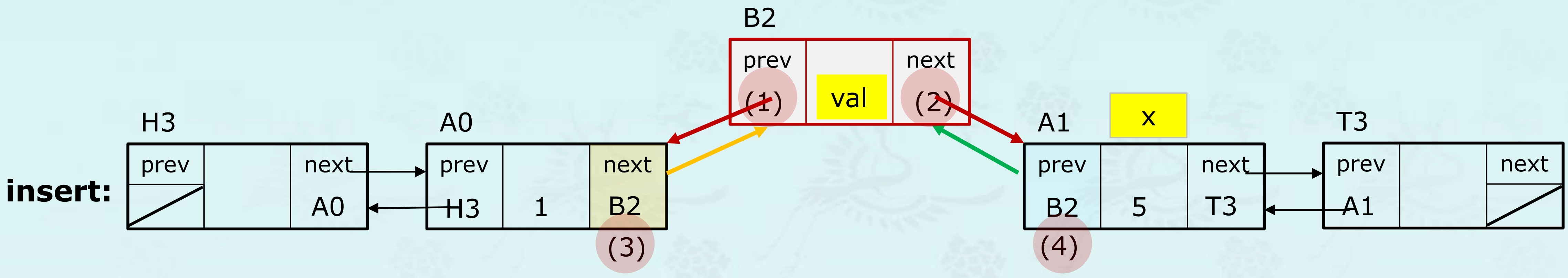


Express (1) ~ (4) in mnemonics and real code:

(1) B2→prev	(1) node→prev
(2) B2→next	(2) node→next
(3) A0→next or A1	(3) x or x→prev→next
(4) A1→prev or A0	(4) x→prev

doubly linked list – insert()

`void insert(pNode x, int val)`



Express (1) ~ (4) in mnemonics and real code:

(1) B2→prev
(2) B2→next
(3) A0→next or A1
(4) A1→prev or A0

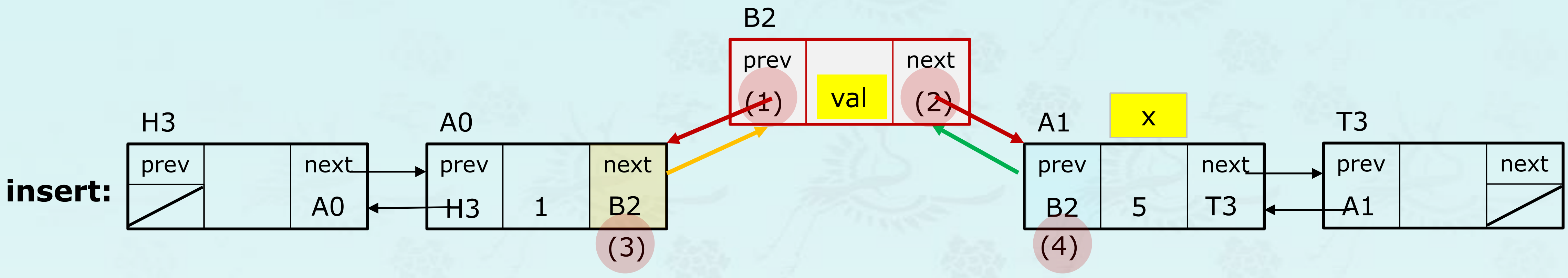
(1) node→prev
(2) node→next
(3) x or x→prev→next
(4) x→prev

Link 1,2: Link the new node to A0 and A1 nodes
`pNode node = new Node(val, x→prev, x);`

Link 3: Set A0→next to the new node
`x→prev→next = node;`

Link 4: Set A1→prev to the new node
`x→prev = node;`

doubly linked list – insert()



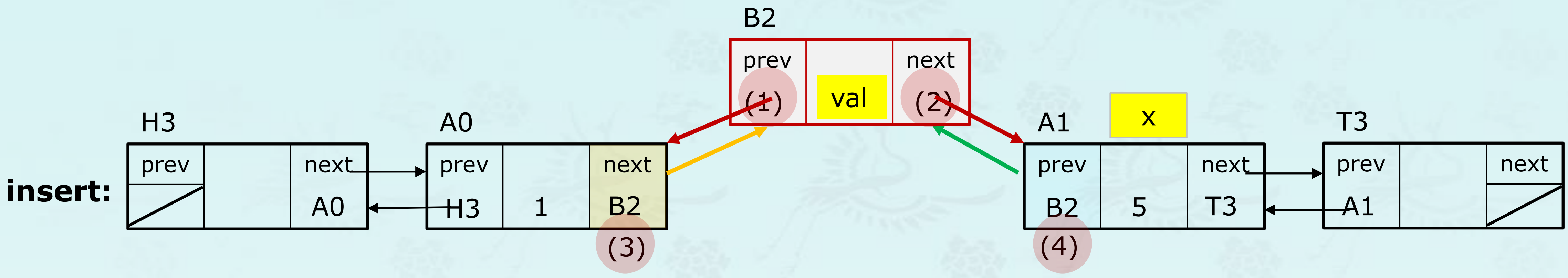

```
void insert(pNode x, int val) {
  pNode node = new Node{val, x->prev,  x};
  x->prev = x->prev->next = node;
}
```

Link 1,2: Link the new node to A0 and A1 nodes
pNode node = new Node(val, x→prev, x);

Link 3: Set A0→next to the new node
x→prev→next = node;

Link 4: Set A1→prev to the new node
x→prev = node;

doubly linked list – insert()



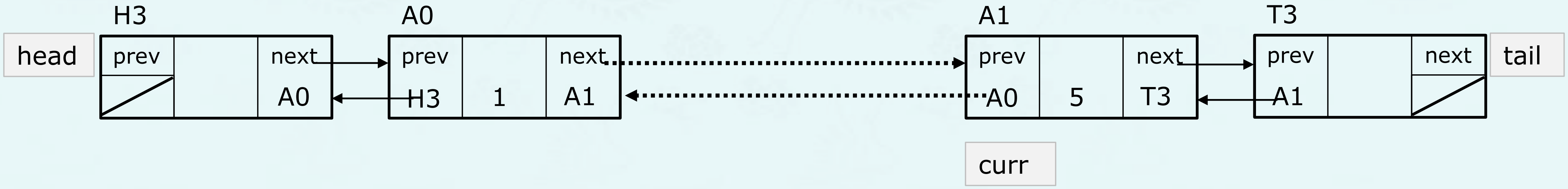

```
void insert(pNode x, int val) {
  pNode node = new Node{val, x->prev,  x};
  x->prev = x->prev->next = node;
}
```

insert() extends the list by inserting a new node with val **before** the node **at** the specified position **x**.

For example, if **begin(p)** is specified as an insertion position, the new node becomes the first one in the list.

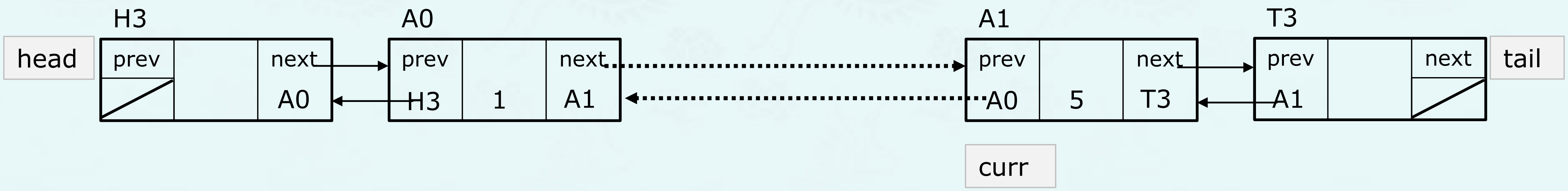
doubly linked list – **push_front()**

```
// Inserts a new node at the beginning of the list, right before its
// current first node. The content of item is copied(or moved) to the
// inserted node. This effectively increases the container size by one.
void push_front(pList p, int val) {
  insert(begin(p), val);
}
```



doubly linked list – **pop_front()**

```
// Removes the first node in the list container, effectively reducing
// its size by one. This destroys the removed node.
void pop_front(pList p) {
  if (!empty(p)) erase(begin(p));
}
```



doubly linked list – **find(), _more(), and _less()**

```
// returns the first node with a value,
// tail or end(p) node otherwise.
pNode find(pList p, int val) {
  pNode c = begin(p);
  for (; c != end(p); c = c->next)
    if (c->item == val) return c;
  return c;
}
```

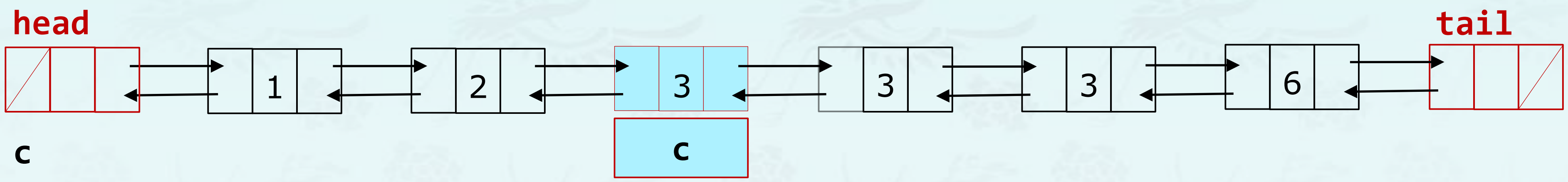
```
// returns the node of which val is greater
// than x firstly encountered.
pNode _more(pList p, int val) {
  pNode c = begin(p);
  for (; c != end(p); c = c->next)
    if (c->item > val) return c;
  return c;
}
```

```
// returns the node of which val is smaller
// than x firstly encountered
pNode _less(pList p, int val) {
  pNode c = begin(p);
  for (; c != end(p); c = c->next)
    if (c->item < val) return c;
  return c;
}
```

doubly linked list – **pop_all()***

```
// remove all occurrences of nodes with val given in the list.
void pop_all(pList p, int val) {
  while (find(p, val) != end(p)) {
    pop(p, val);
  }
}
```

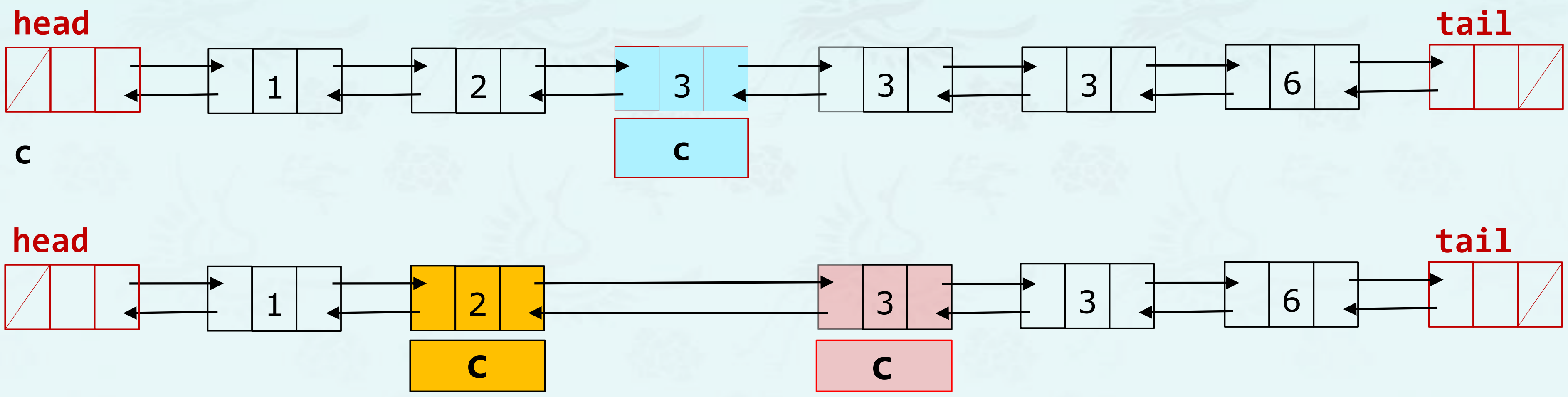
version 1 with a problem



What is the time complexity of the function?

doubly linked list – **pop_all()***

```
// remove all occurrences of nodes with val given in the list.
void pop_all(pList p, int val) {
  for (pNode c = begin(p); c != end(p); c = c->next)
    if (c->item == val)
       erase(c);
}
```
version 2 with a bug

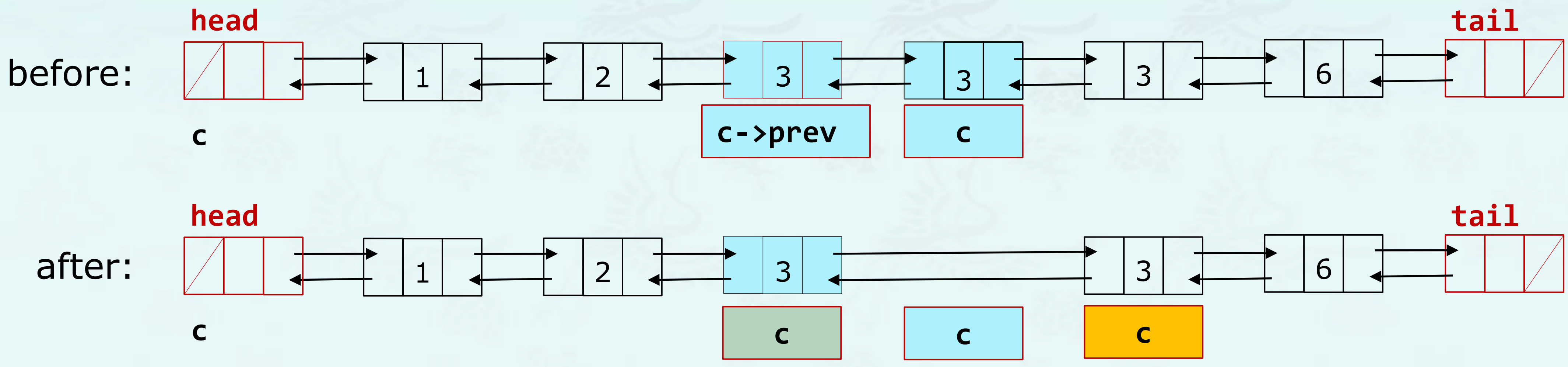


Where should c be pointing **right after erase(c)**, 2 or 4?

doubly linked list – **unique()***

```
// removes extra nodes that have duplicate values from the list.
void unique(pList p) {
  if (size(p) <= 1) return;
  for (pNode c = begin(p); c != end(p); c = c->next)
    if (c->item == c->prev->item)
        erase(c);
}
```

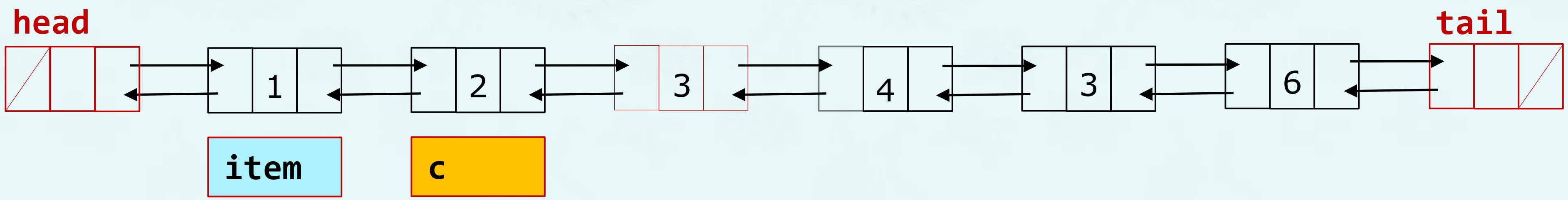
version 1 with a bug



Where should c be pointing **right after the first erase(c)**?

doubly linked list – **sorted()**

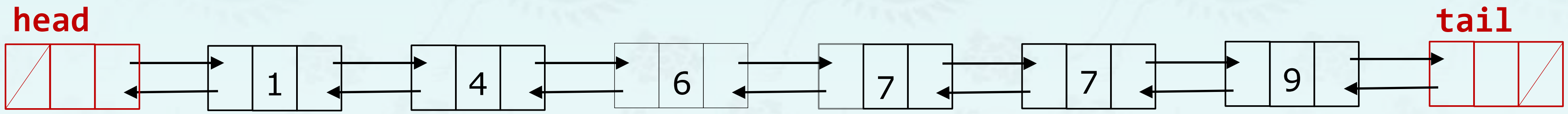
```
// returns true if the list is sorted either ascending or descending.
bool sorted(pList p) {
  return sorted(p, ascending) || sorted(p, descending);
}
bool sorted(pList p, int(*comp)(int a, int b)) {
  if (size(p) <= 1) return true;
  int item = "set it to 1st node value
  for (pNode c = "starts from second node"; c != end(p); c = c->next) {
    // your code here; return false as soon as out of order found
    // compare item and c->item
    item = c->item;
  }
  return true;
}
```



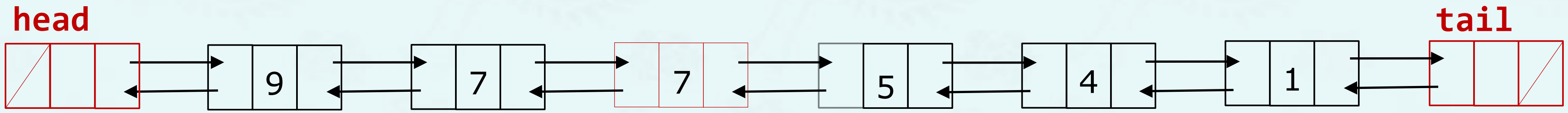
doubly linked list – push_sorted()

```
// inserts a new node with val in sorted order
void push_sorted(pList p, int val) {
  if sorted(p, "ascending order")
     insert("find a node _more() than val", val);
  else
     insert("find a node _less() than val", val);
}
```

Which node should be located to invoke insert() if val = 4?

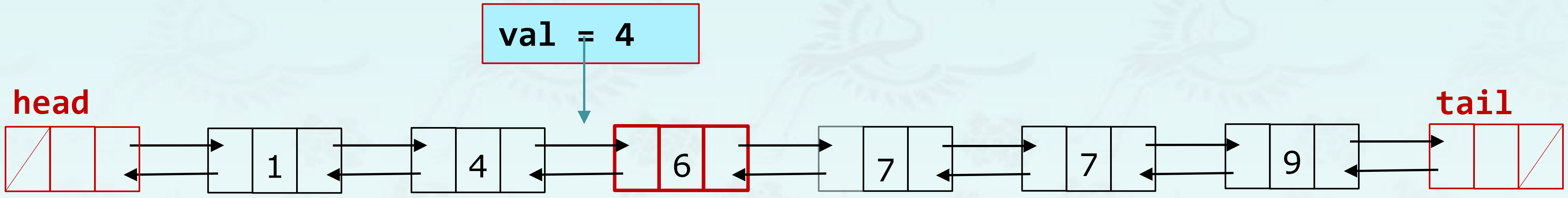


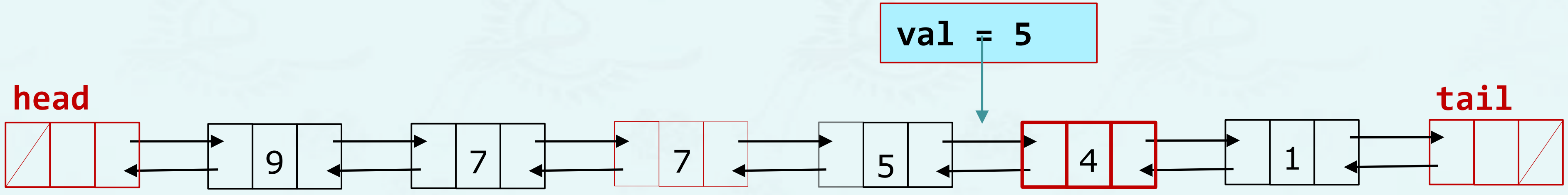
Which node should be located to invoke insert() if val = 5?



doubly linked list – **push_sorted()**

```
// inserts a new node with val in sorted order
void push_sorted(pList p, int val) {
  if sorted(p, "ascending order")
     insert("find a node _more() than val", val);
  else
     insert("find a node _less() than val", val);
}
```



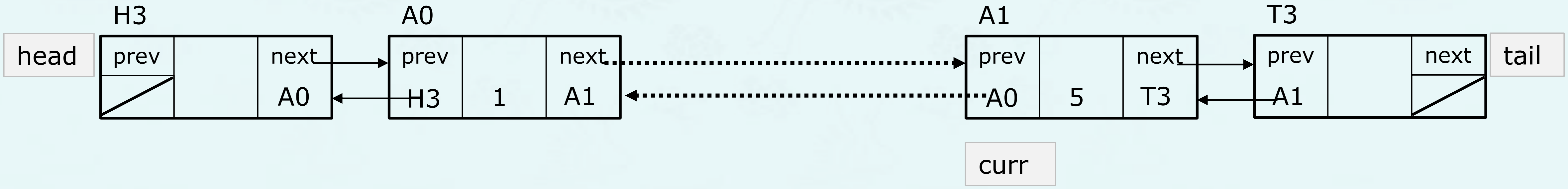


doubly linked list – **bubbleSort()**

```
void bubbleSort(pList p, int(*comp)(int, int)) {
  // if (sorted(p)) { reverse(p); return; }

  pNode curr;
  for (pNode i = begin(p); i != end(p); i = i->next) {
    for (curr = begin(p); curr->next != end(p); curr = curr->next) {
      if (comp(curr->item, curr->next->item) > 0)
         swap(curr->item, curr->next->item);
    }
    tail = curr;
  }
}
```

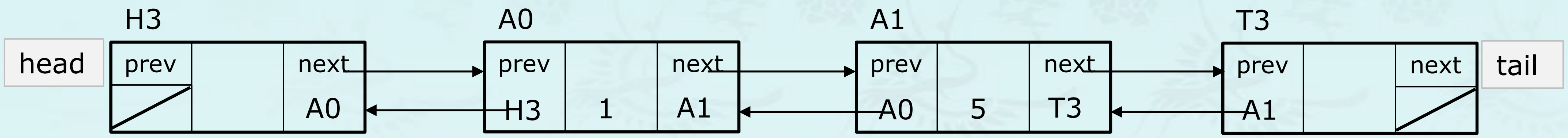
doubly linked list – **reverse()****

```
// reverses the order of the nodes in the list container. O(n)
void reverse(pList p) {
  if (size(p) <= 1) return;
  // hint: swap prev and next in every node including head & tail
  // then, swap head and tail.
  // hint: use while loop, don't use begin()/end()
  // your code here

}
```

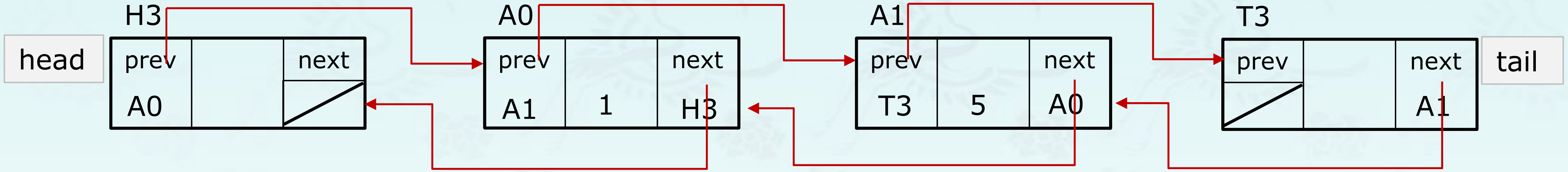


doubly linked list – **reverse()****

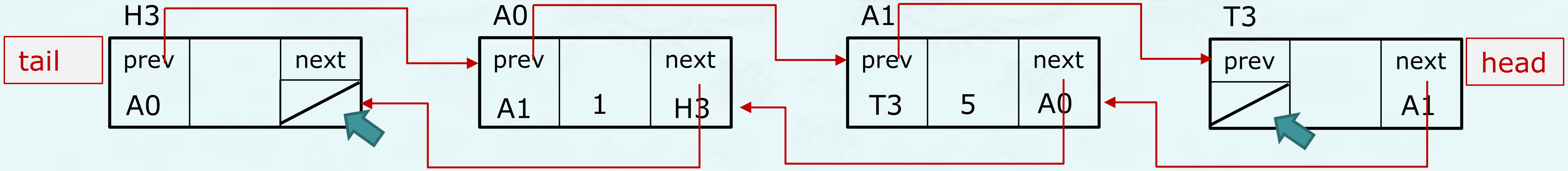
doubly linked list reverse() algorithm



step 1: swap prev and next in every node **including head & tail.**

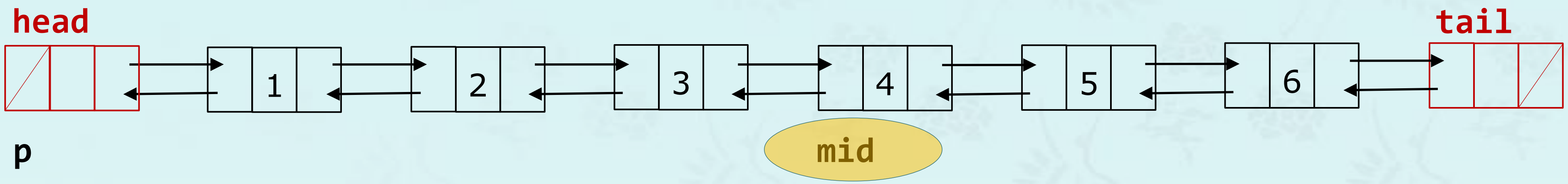


step 2: swap head and tail node.




```
What does begin() and end() from this list, respectively?
```

doubly linked list – **half()**



- **Method 1:** Count how many nodes there are in the list, then scan to the halfway point, breaking the last link followed.
- **Method 2:** It works by sending rabbit and turtle down the list: turtle moving at speed one, and rabbit moving at speed two. As soon as the rabbit hits the end, you know that the turtle is at the halfway point as long as the rabbit gets asleep at the halfway.

```
pNode half(pList p) {
  pNode rabbit = begin(p);
  pNode turtle = begin(p);

  while (rabbit != end(p)) {
    rabbit = rabbit->next->next;
    turtle = turtle->next;
  }
  return turtle;
} // buggy on purpose
```

This code will not work some cases.
Debug if you want to go for this code.

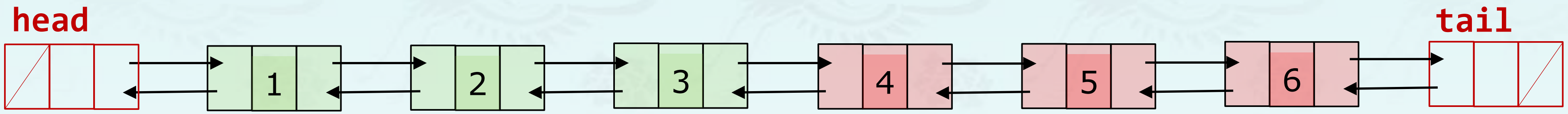
doubly linked list – **shuffle()*****

```
// returns so called "perfectly shuffled" list.
// The first half and the second half are interleaved each other.
// The shuffled list begins with the second half of the original.
// For example, 1234567890 returns 6172839405.

Algorithm:
1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
4) keep on interleaving nodes until the "que" is exhausted.
    •  save away next pointers of mid and que.
    •  interleave nodes in the "que" into "mid" in the list of p.
       (insert the first node in "que" at the second node in "mid".)
```

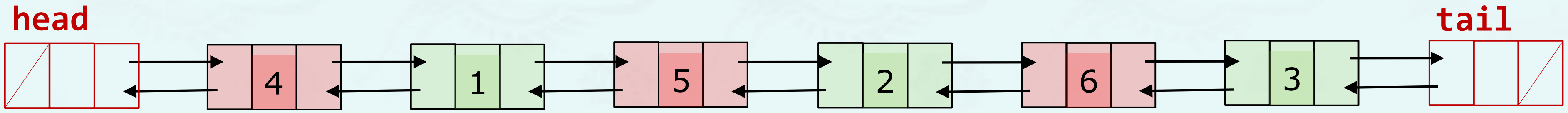
doubly linked list – **shuffle()*****

```
Algorithm:
1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
4) keep on interleaving nodes until the "que" is exhausted.
    •  save away next pointers of mid and que.
    •  interleave nodes in the "que" into "mid" in the list of p.
       (start inserting the first node in "que" at the second node in "mid".)
```

before:

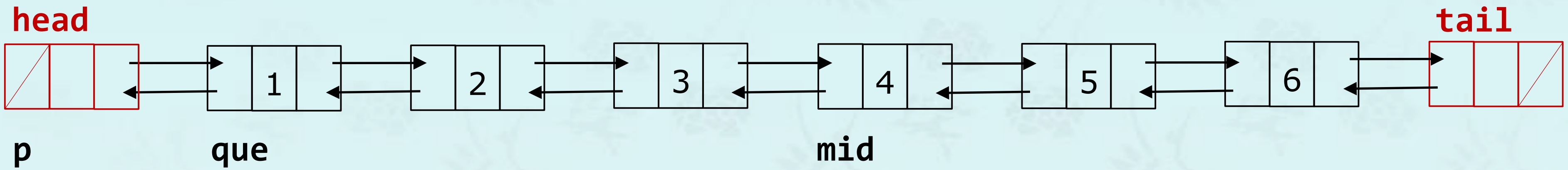


after:



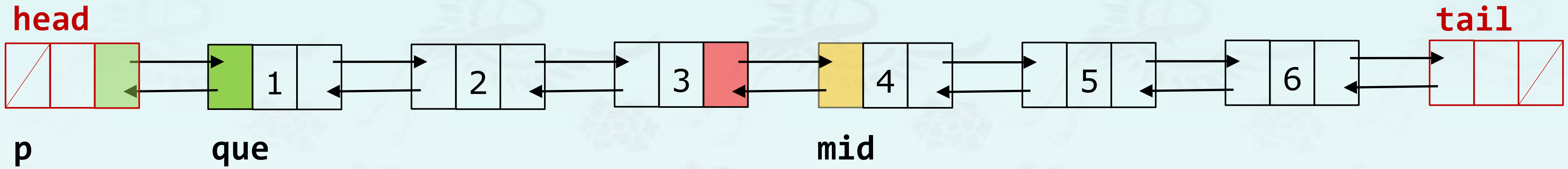
doubly linked list – **shuffle()*****

before:



1) find the mid node of the list p to split it into two lists at the mid node.

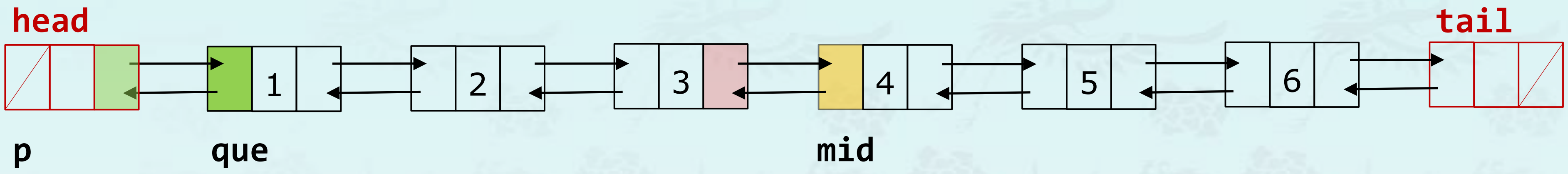
2) remove the 1st half from the list p, and keep it as a list "que" to add.

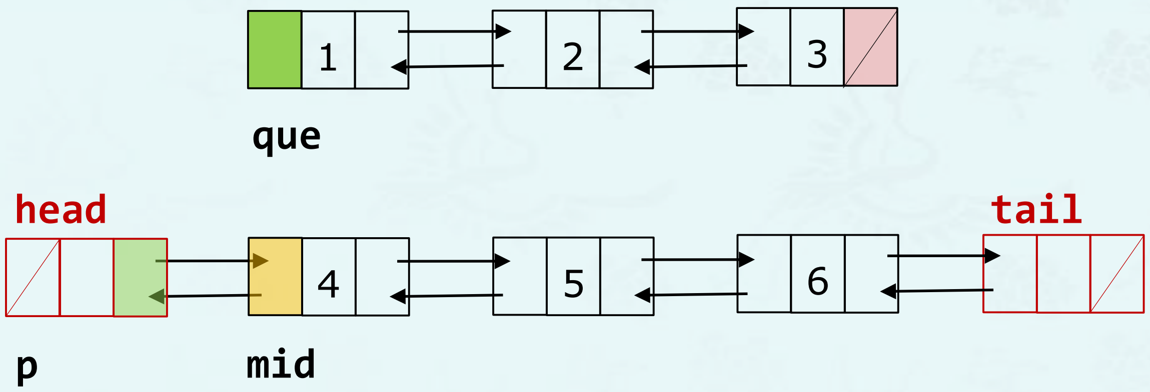



```
pNode mid = half(p);
pNode que = begin(p);
```

doubly linked list – **shuffle()*****

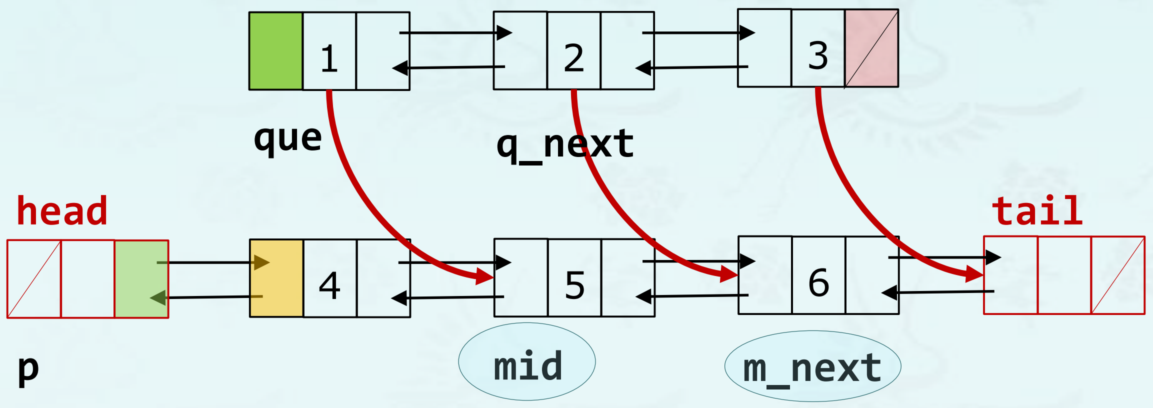
1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.





doubly linked list – **shuffle()*****

1) find the mid node of the list p to split it into two lists at the mid node.
2) remove the 1st half from the list p, and keep it as a list "que" to add.
3) set the list p head such that it points the "mid" of the list p.
4) keep on interleaving nodes until the "que" is exhausted.
   - save away next pointers of mid and que.
   - interleave nodes in the "que" into "mid" in the list of p.
     (start inserting the fist node in "que" at the second node in "mid".)




```
mid = begin(p)->next;
while (que != nullptr) {
  pNode q_next = que->next;
  pNode m_next = mid->next;

  // que is inserted at mid.
  ...

  mid = m_next;
  que = q_next;
}
```

Abstract in-place merge demo

Goal: push sorted N – O(n log n)

Given two sorted containers, **one sorted linked list and one sorted array**, merge the array into the list in sorted manner. O(n)

array:

A	C	E	R	T

v[j]

two containers to merge

array v

N

doubly linked list:

E	E	G	M	R

i

pNode i = begin(p)

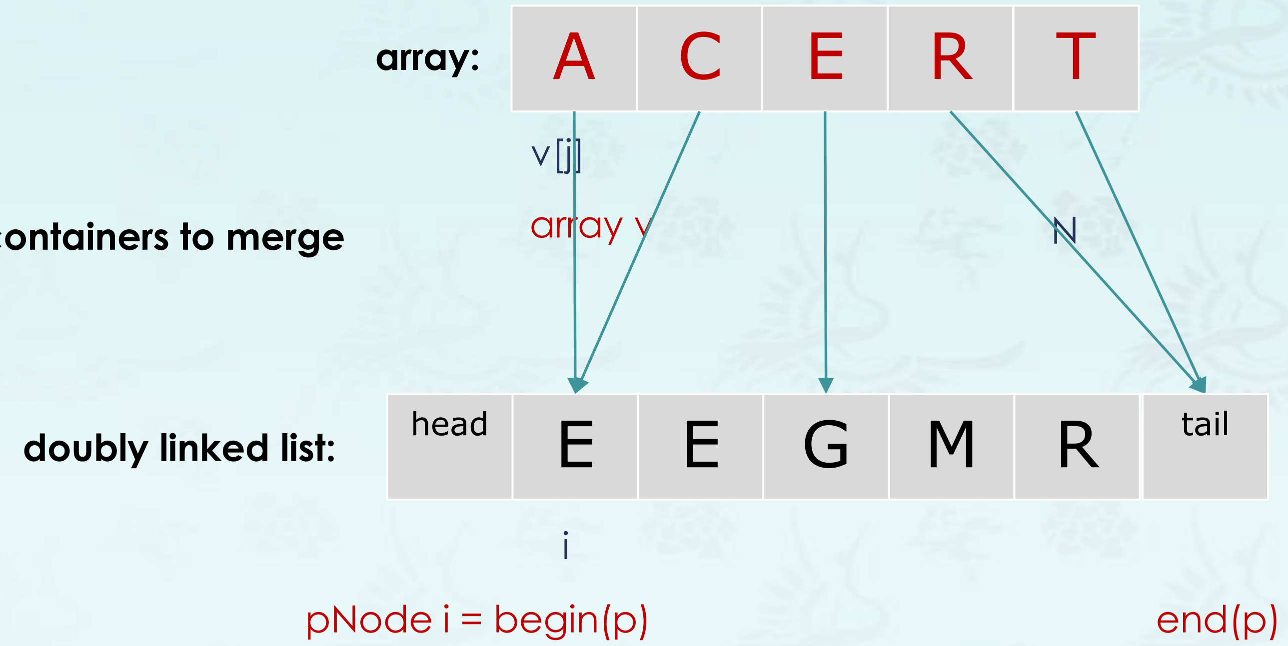
end(p)

Abstract in-place merge demo

Goal: **push sorted N – O(n log n)**

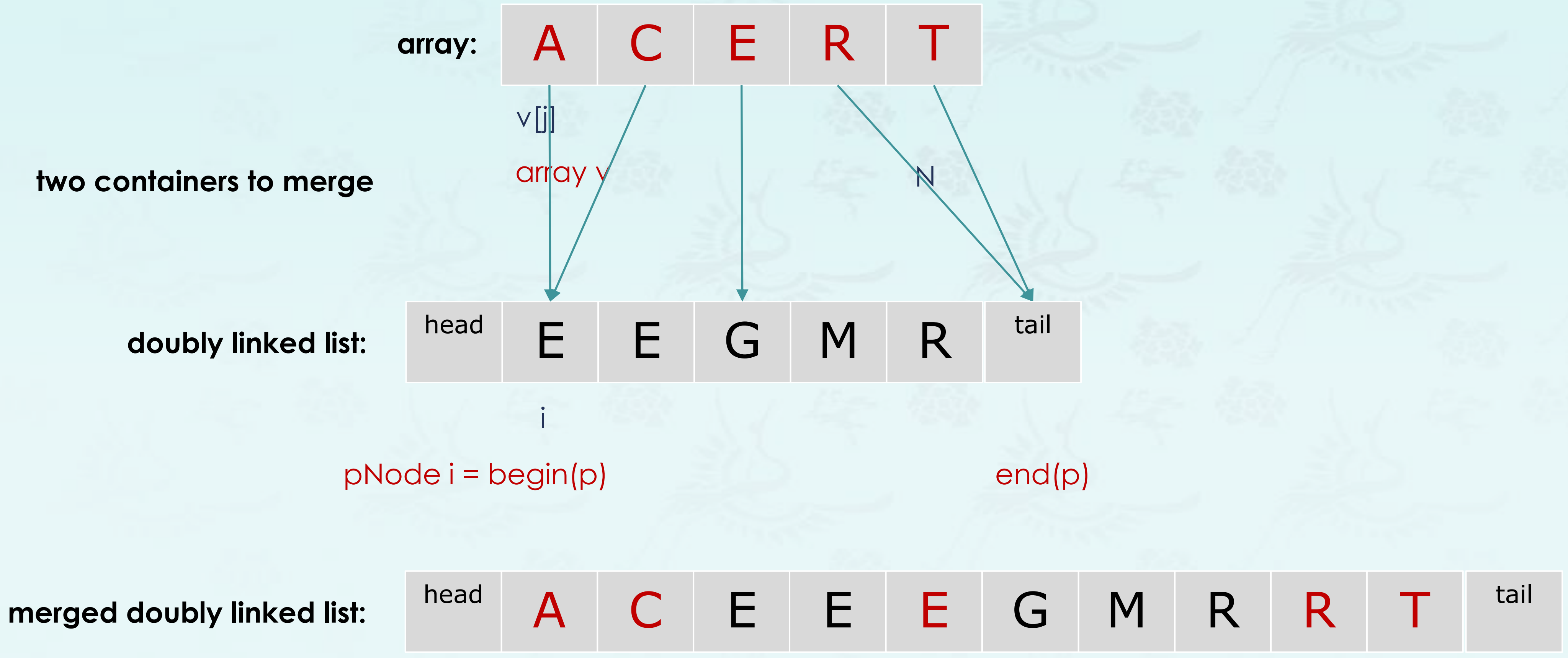
Given two sorted containers, **one sorted linked list and one sorted array**, merge the array into the list in sorted manner. O(n)

two containers to merge



Abstract in-place merge demo

Goal: **push sorted N – O(n log n)**
Given two sorted containers, **one sorted linked list and one sorted array**, merge the array into the list in sorted manner. O(n)



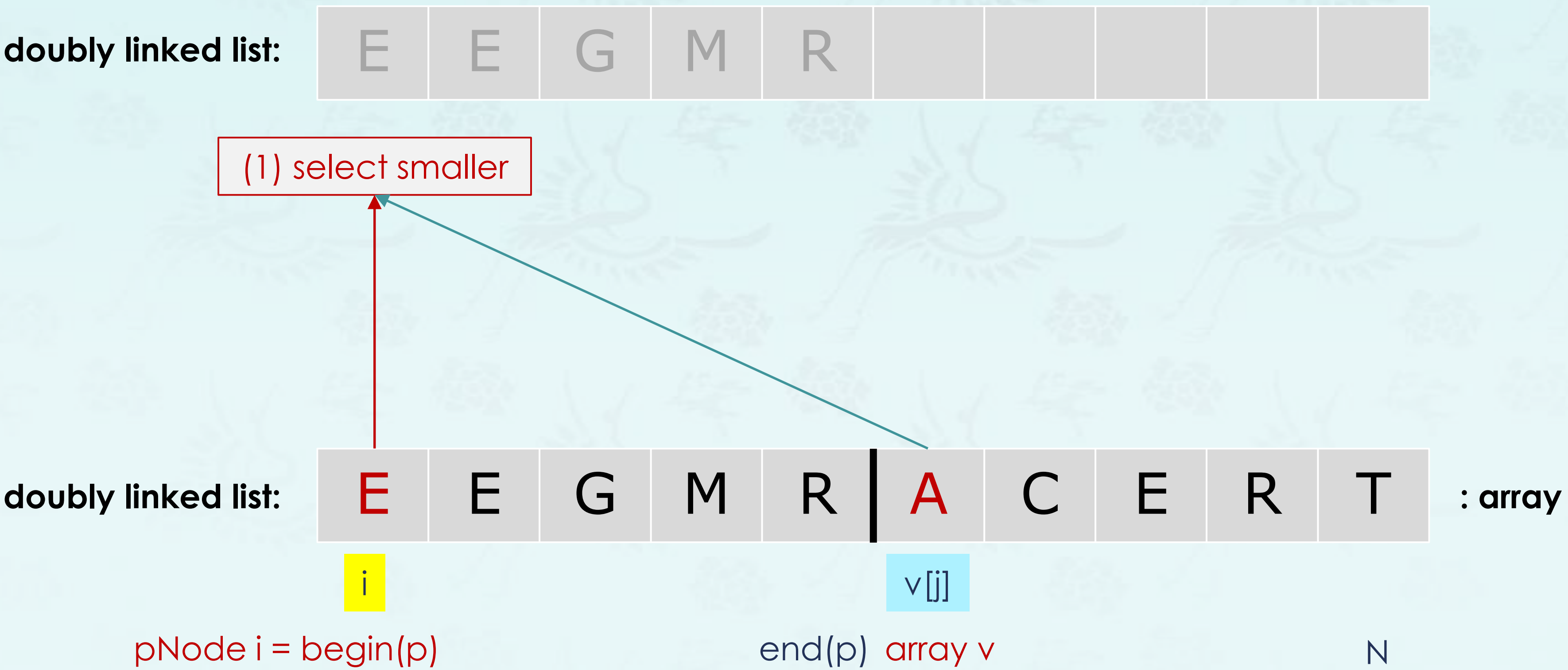
Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

Abstract in-place merge demo

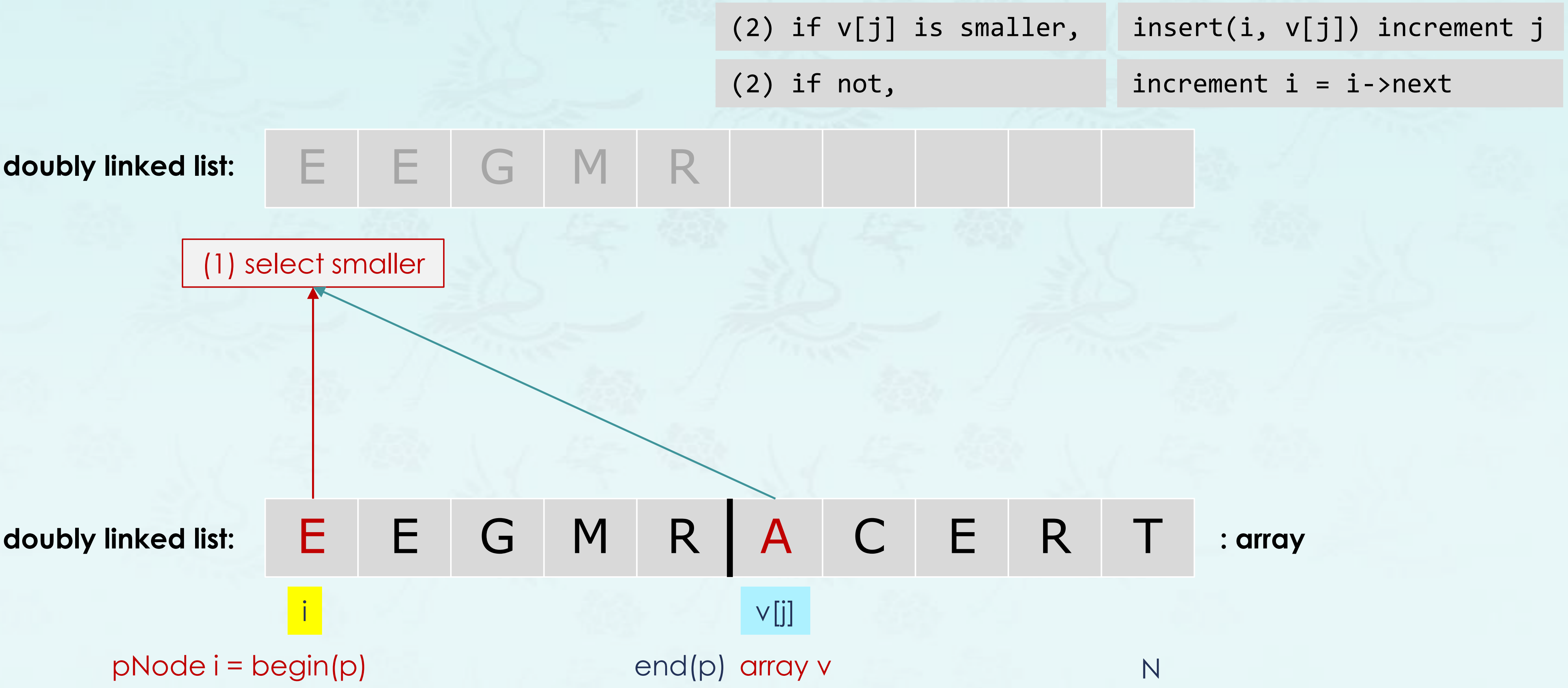
Goal: **push sorted N – O(n log n)**

Given two sorted containers, **one sorted linked list and one sorted array**, merge the array into the list in sorted manner. O(n)

Abstract in-place merge demo

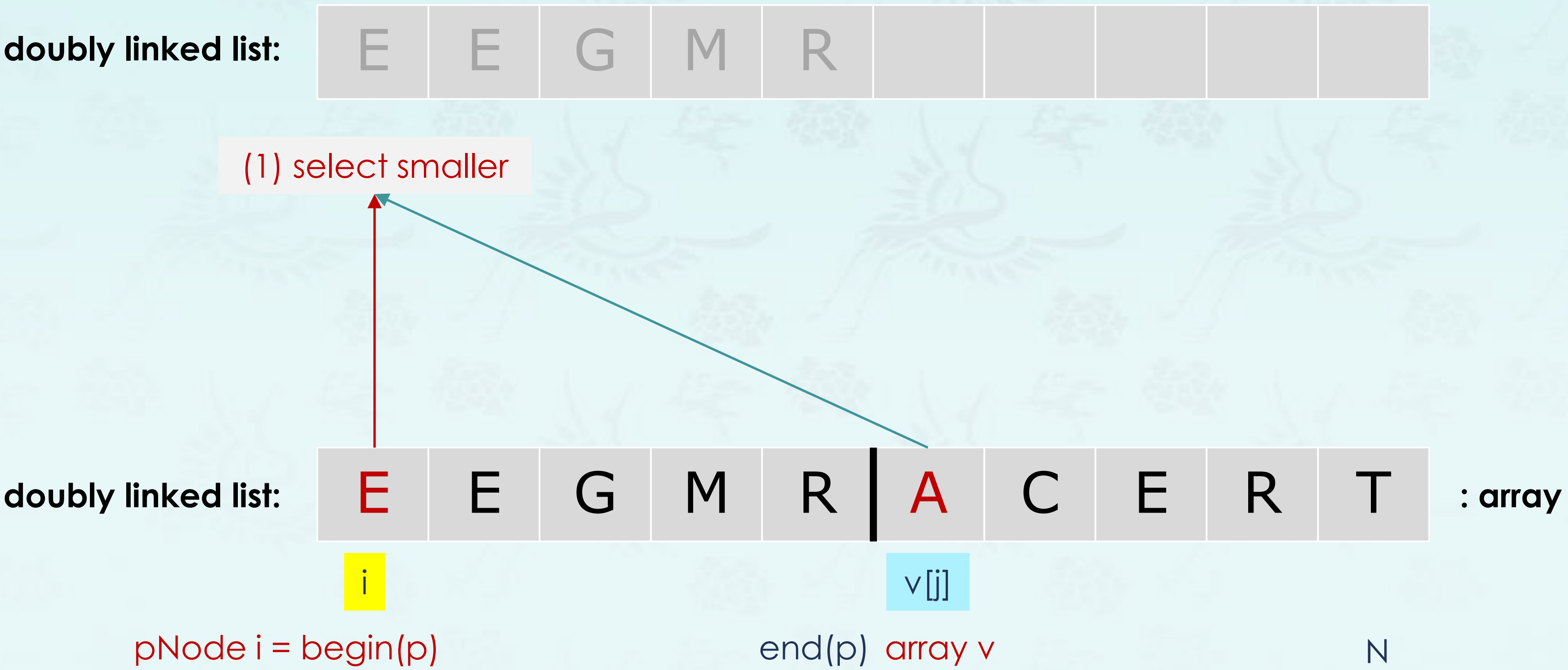


Abstract in-place merge demo



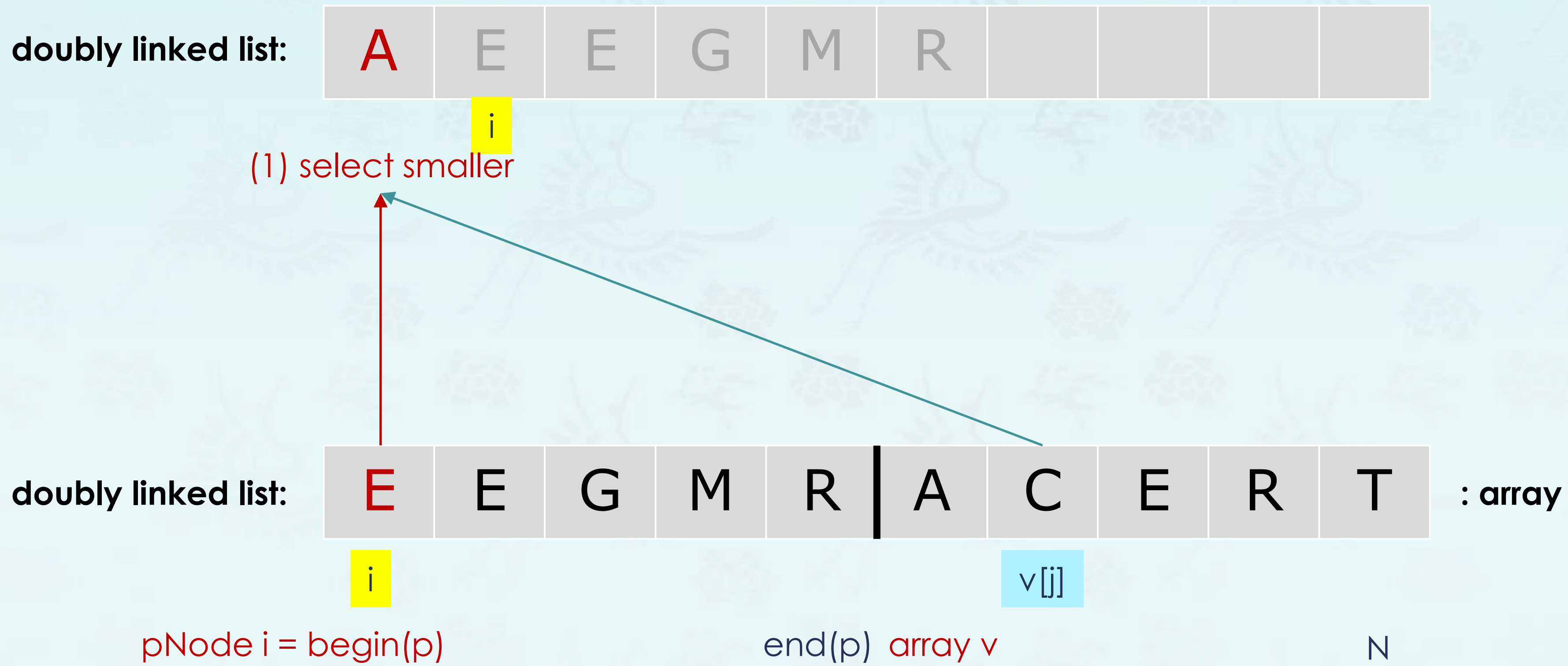
Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`



Abstract in-place merge demo

(2) if v[j] is smaller,	$\boxed{\texttt{insert(i, v[j]) increment j}}$
(2) if not,	increment i = i->next



Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	E	E	G	M	R				
	i								

(1) select smaller

doubly linked list:

E	E	G	M	R	A	C	E	R	T	**: array**
i						v[j]				

pNode i = begin(p) end(p) array v N

two containers to merge

Abstract in-place merge demo

`(2) if v[j] is smaller,`	$\boxed{\texttt{insert(i, v[j]) increment j}}$
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	C	E	E	G	M	R			
	i								

(1) select smaller

doubly linked list:

E	E	G	M	R	A	C	E	R	T
i						v[j]			

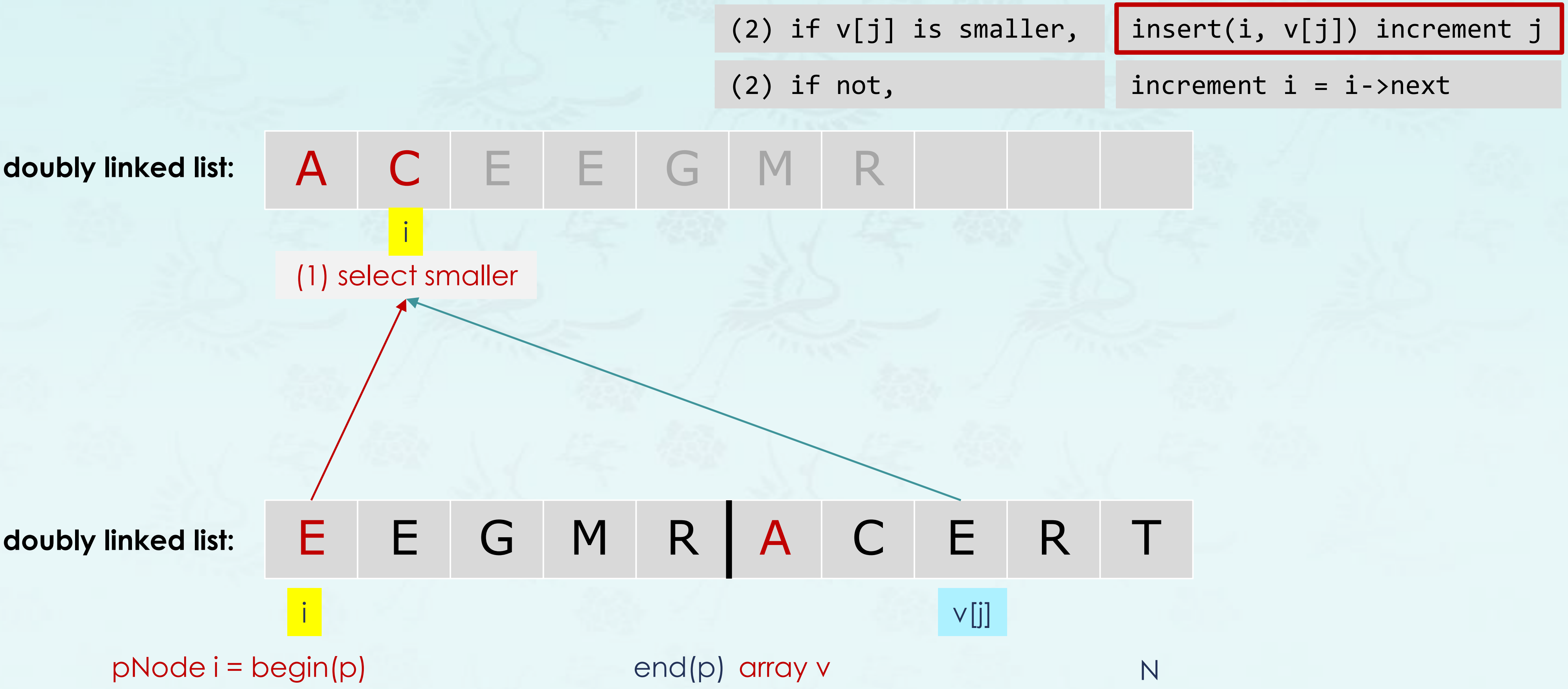
: array

pNode i = begin(p)

end(p) array v

N

Abstract in-place merge demo



Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	C	E	E	G	M	R			

i

(1) select smaller

doubly linked list:

E	E	G	M	R	A	C	E	R	T

: array

i

v[j]

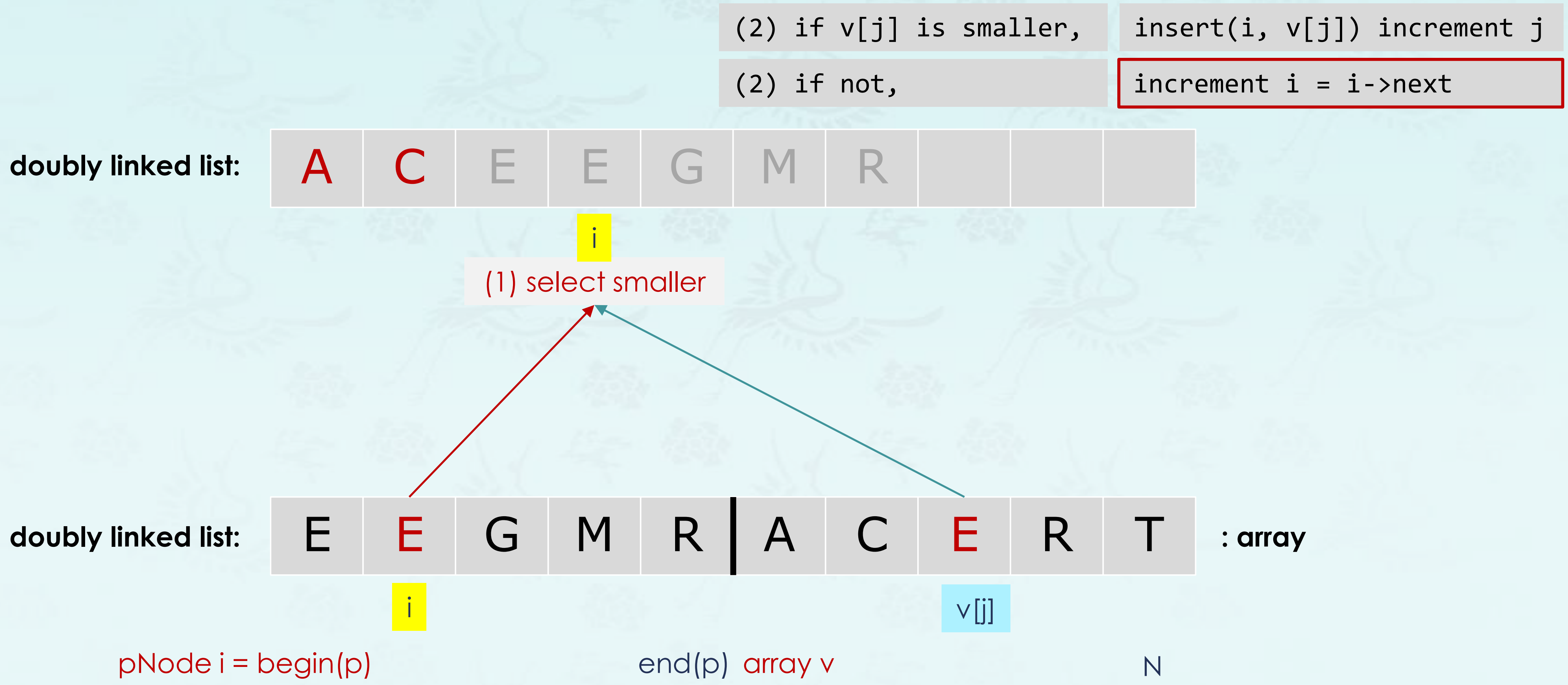
pNode i = begin(p)

end(p) array v

N

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept, Handong Global University

Abstract in-place merge demo



Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	C	E	E	G	M	R			

i

(1) select smaller

doubly linked list:

E	E	G	M	R	A	C	E	R	T

: array

i

v[j]

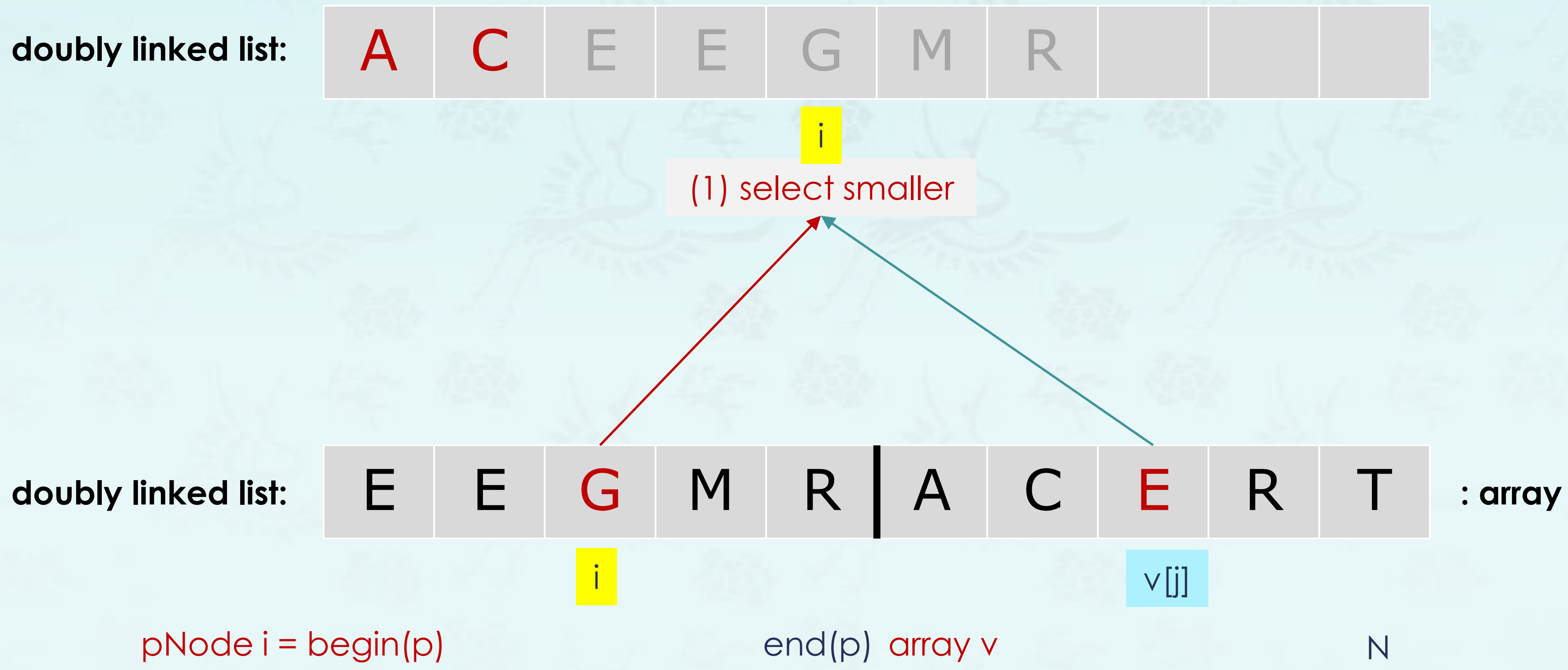
pNode i = begin(p)

end(p) array v

N

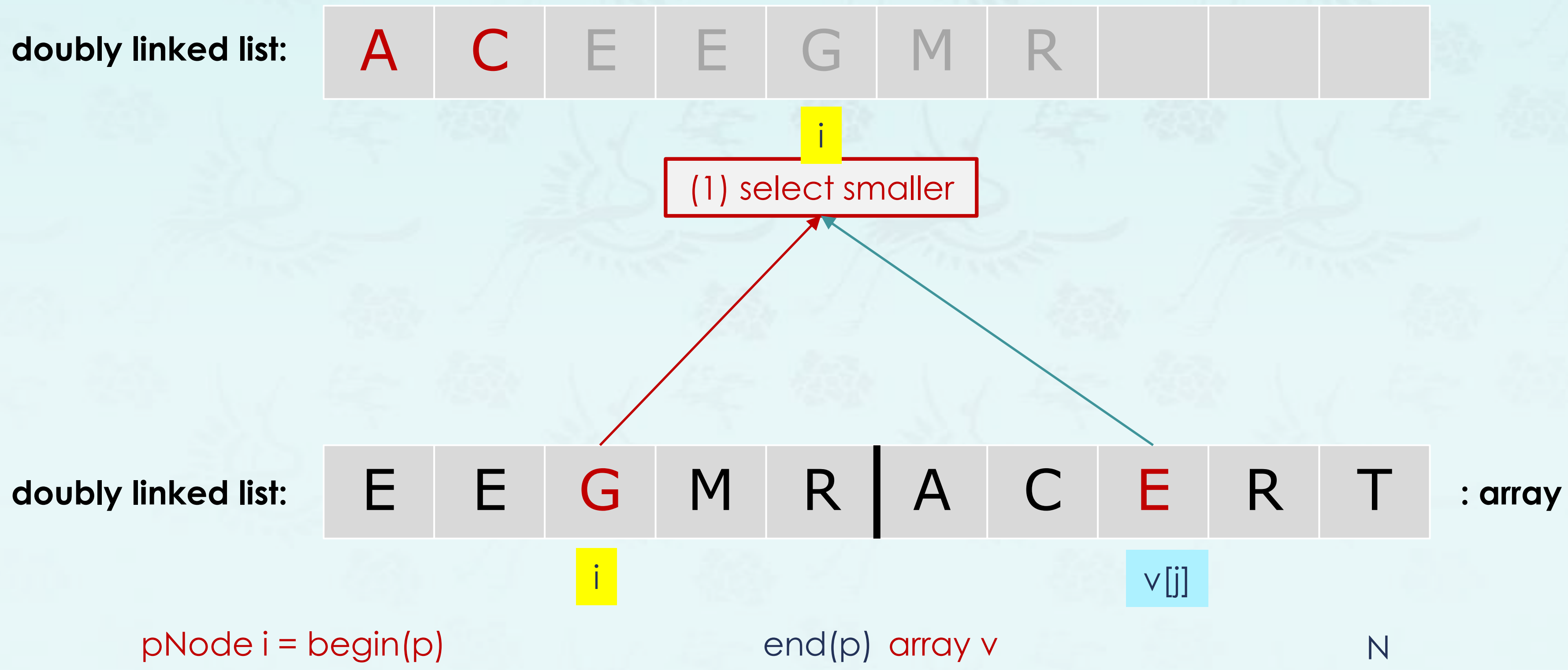
Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`



Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`



Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	C	E	E	G	M	R			

i

(1) select smaller

doubly linked list:

E	E	G	M	R	A	C	E	R	T

: array

i

v[j]

pNode i = begin(p)

end(p) array v

N

Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	C	E	E	E	G	M	R		

i

(1) select smaller

doubly linked list:

E	E	G	M	R	A	C	E	R	T

: array

i

v[j]

pNode i = begin(p)

end(p) array v

N

Abstract in-place merge demo

(2) if v[j] is smaller,	insert(i, v[j]) increment j
(2) if not,	increment i = i->next

doubly linked list:

A	C	E	E	E	G	M	R		

i

(1) select smaller

doubly linked list:

E	E	G	M	R	A	C	E	R	T

: array

i

v[j]

pNode i = begin(p)

end(p) array v

N

Abstract in-place merge demo

(2) if v[j] is smaller,	insert(i, v[j]) increment j
(2) if not,	increment i = i->next

doubly linked list:

A	C	E	E	E	G	M	R		

i

(1) select smaller

doubly linked list:

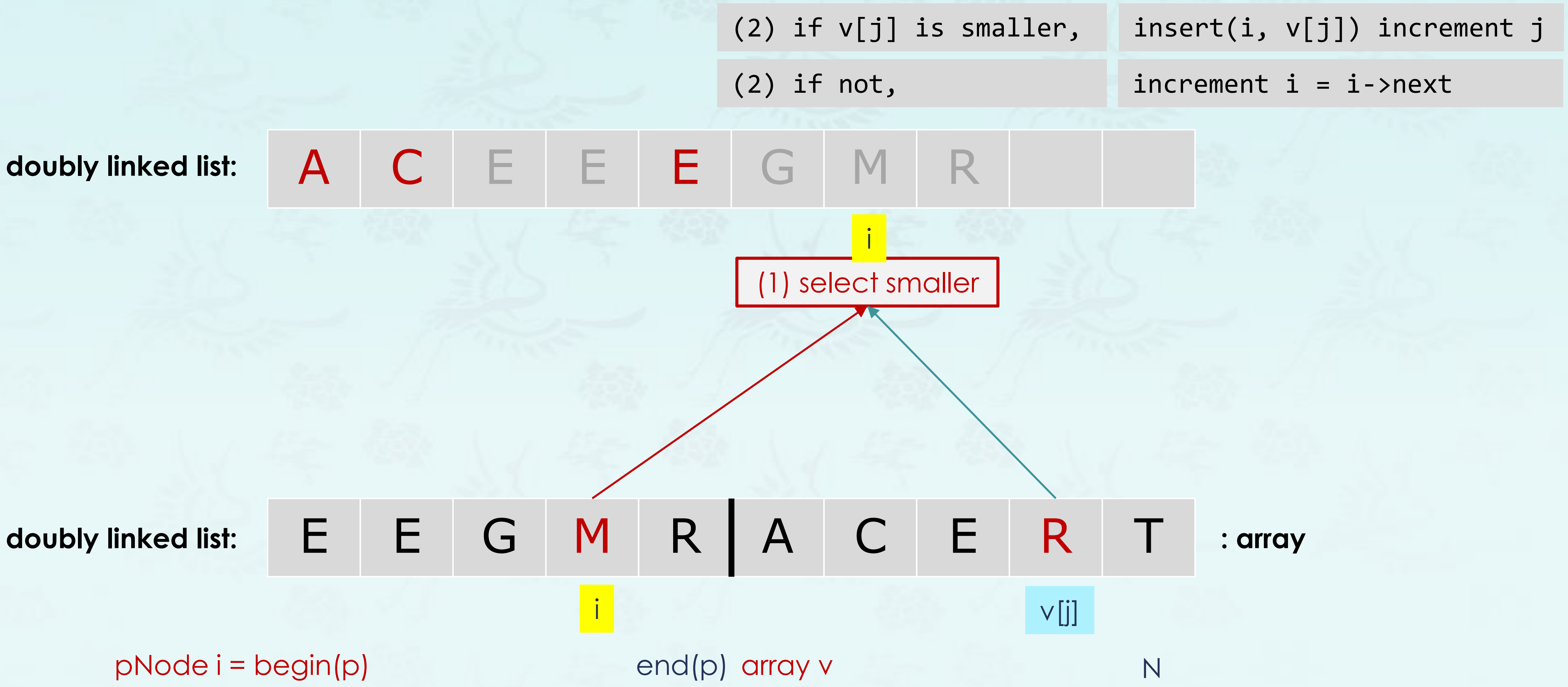
E	E	G	M	R	A	C	E	R	T

: array

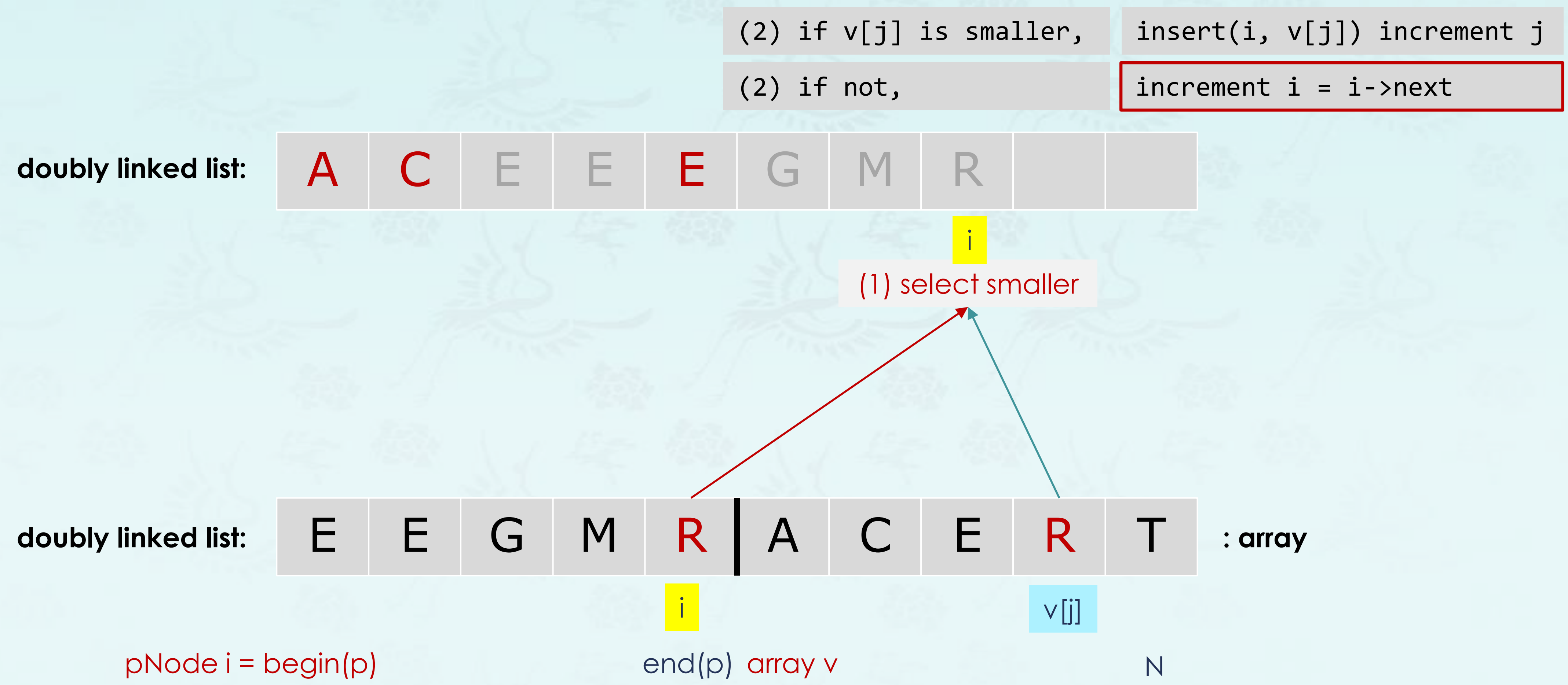
i   v[j]

pNode i = begin(p)   end(p) array v   N

Abstract in-place merge demo



Abstract in-place merge demo



Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	C	E	E	E	G	M	R		

i

(1) select smaller

doubly linked list:

E	E	G	M	R	A	C	E	R	T

: array

i

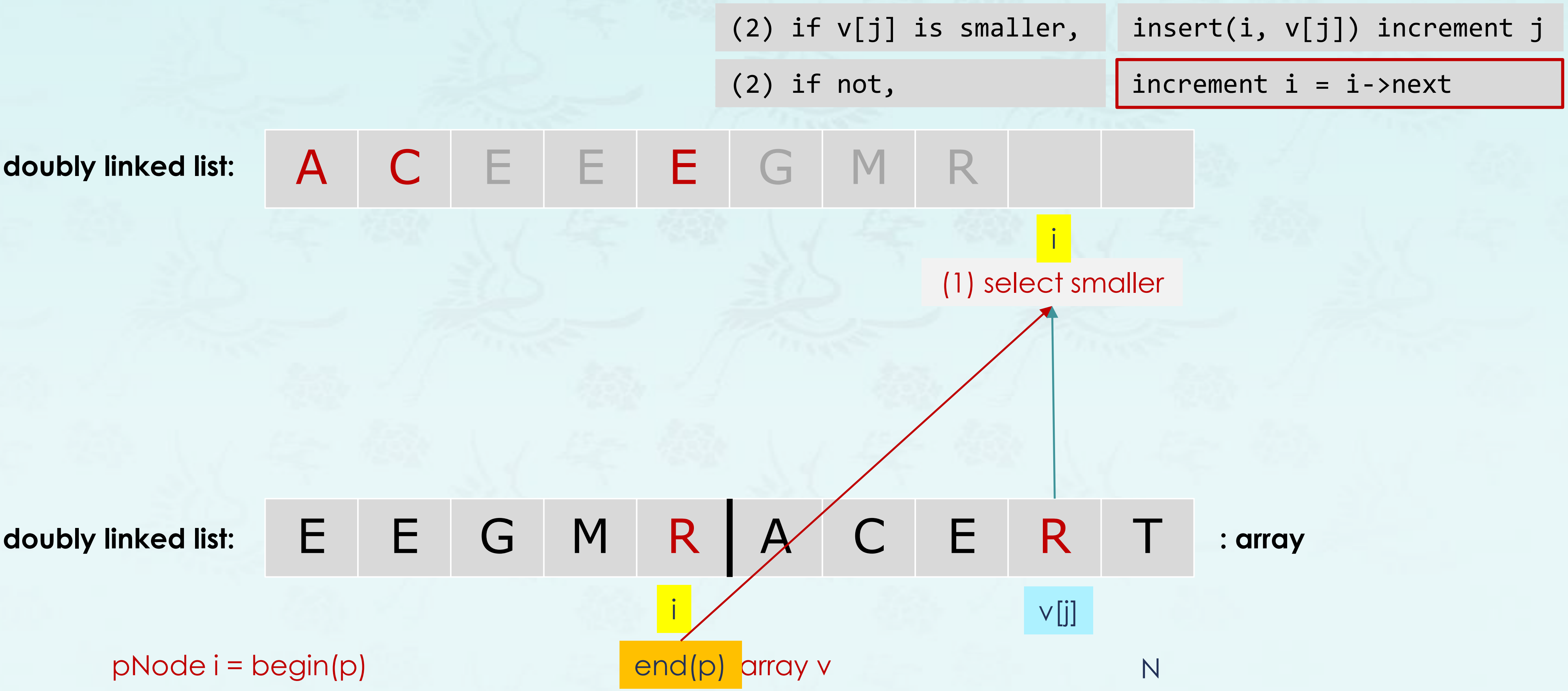
v[j]

pNode i = begin(p)

end(p) array v

N

Abstract in-place merge demo



Abstract in-place merge demo

(2) if v[j] is smaller,	insert(i, v[j]) increment j
(2) if not,	increment i = i->next

doubly linked list:

A	C	E	E	E	G	M	R		

i

(1) stop the merge,
if reached at the end
of one of containers.

doubly linked list:

E	E	G	M	R	A	C	E	R	T

: array

i

v[j]

pNode i = begin(p)

end(p) array v

N

Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	C	E	E	E	G	M	R		

i

(1) stop the merge,
if reached at the end
of one of containers.

(2) if array is exhausted,
the merge is over.
(2) if list is exhausted,
insert the rest of array v
into the list.

doubly linked list:

E	E	G	M	R	A	C	E	R	T

: array

i

v[j]

pNode i = begin(p)

end(p) array v

N

Abstract in-place merge demo

`(2) if v[j] is smaller,`	`insert(i, v[j]) increment j`
`(2) if not,`	`increment i = i->next`

doubly linked list:

A	C	E	E	E	G	M	R	R	T

i

(1) stop the merge,
if reached at the end
of one of containers.

(2) if array is exhausted,
the merge is over.
(2) if list is exhausted,
insert the rest of array v
into the list.

doubly linked list:

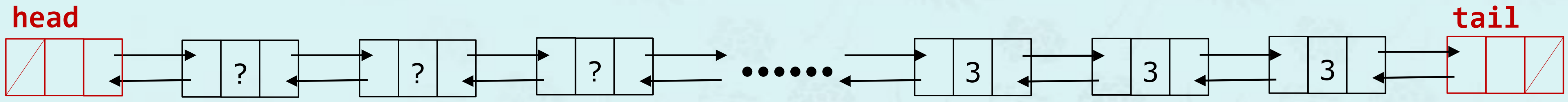
E	E	G	M	R	A	C	E	R	T

: array

i v[j]

pNode i = begin(p) end(p) array v N

doubly linked list – **testing**



1. Running push_backN() once will be enough for most cases. For "pop vals" or "unique", you may run it twice as shown below.
   ① create 50,000 nodes filled with random values at the first run.
   ② create 50,000 nodes filled with a fixed value such as 100,000 or 3 at the second run.
2. Now you are ready to use this vector to test O(n), O(n log n), and O(n^2)
3. You may test your code with 1,000,000 nodes and compare them with listdsx.exe.

doubly linked list – **timing test**

N		10,000	100,000	1 Million	
push sorted O(n)	my code				
	listdsx				
sort (selection) O(n^2)	my code				takes too long unless use quicksort
	listdsx				
reverse O(n)	my code				
	listdsx				
pop vals O(n)	my code				
	listdsx				
unique O(n)	my code				
	listdsx				
shuffle O(n)	my code				
	listdsx				
push sorted N O(n^2)	my code				N = 10, 000, 100,000, 1,000,000
	listdsx				
push sorted N O(n log n)	my code				N = 10, 000, 100,000, 1,000,000
	listdsx				

Summary

&