APS105
Winter 2012

Jonathan Deber
jdeber -at- cs -dot- toronto -dot- edu

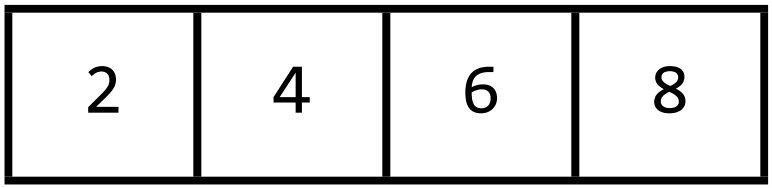
Lecture 30
March 30, 2012

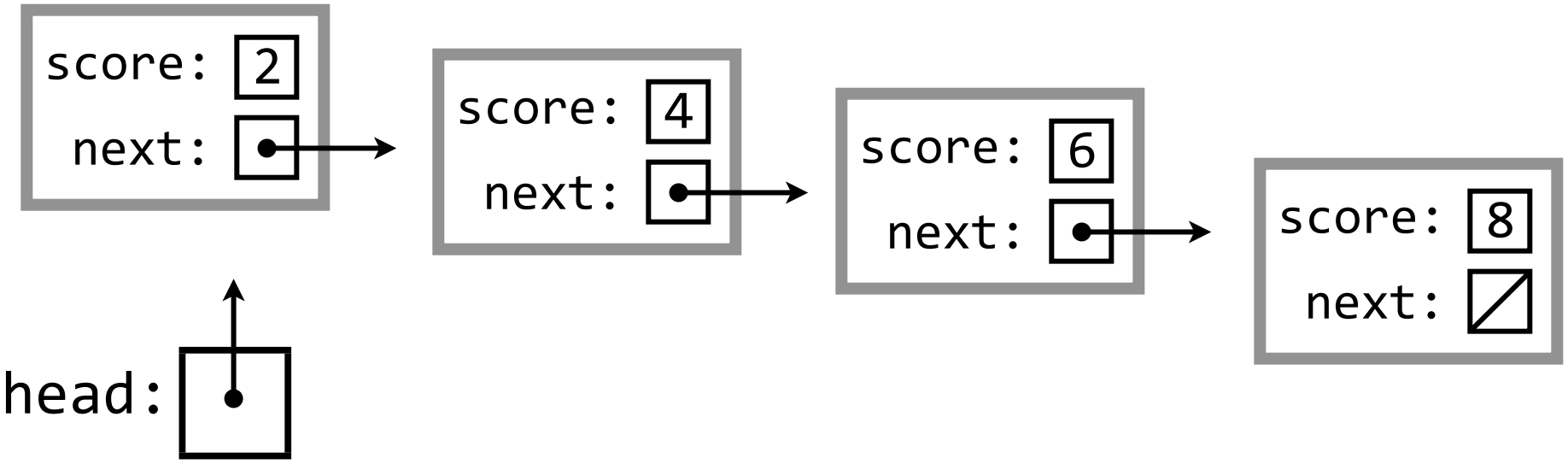
Today

- Linked Lists `->next`

Linked Lists

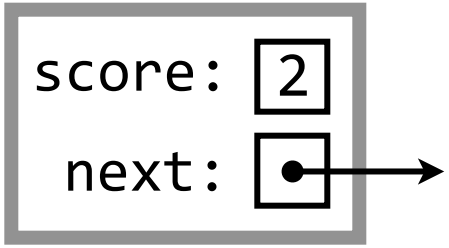
- A different way to store multiple chunks of data
- (Simplest) example of a whole category of data structures, built on indirection (i.e., using pointers)
- We have to build them ourselves


2
4
6
8


score: 2
next:
score: 4
next:
score: 6
next:
score: 8
next:
head:

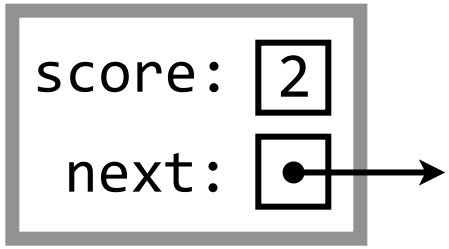
Linked Lists

- A recursive data structure
- Made up of nodes
  - Actual data
  - A pointer to another node
- Each node points to the next node
- Last node points to NULL



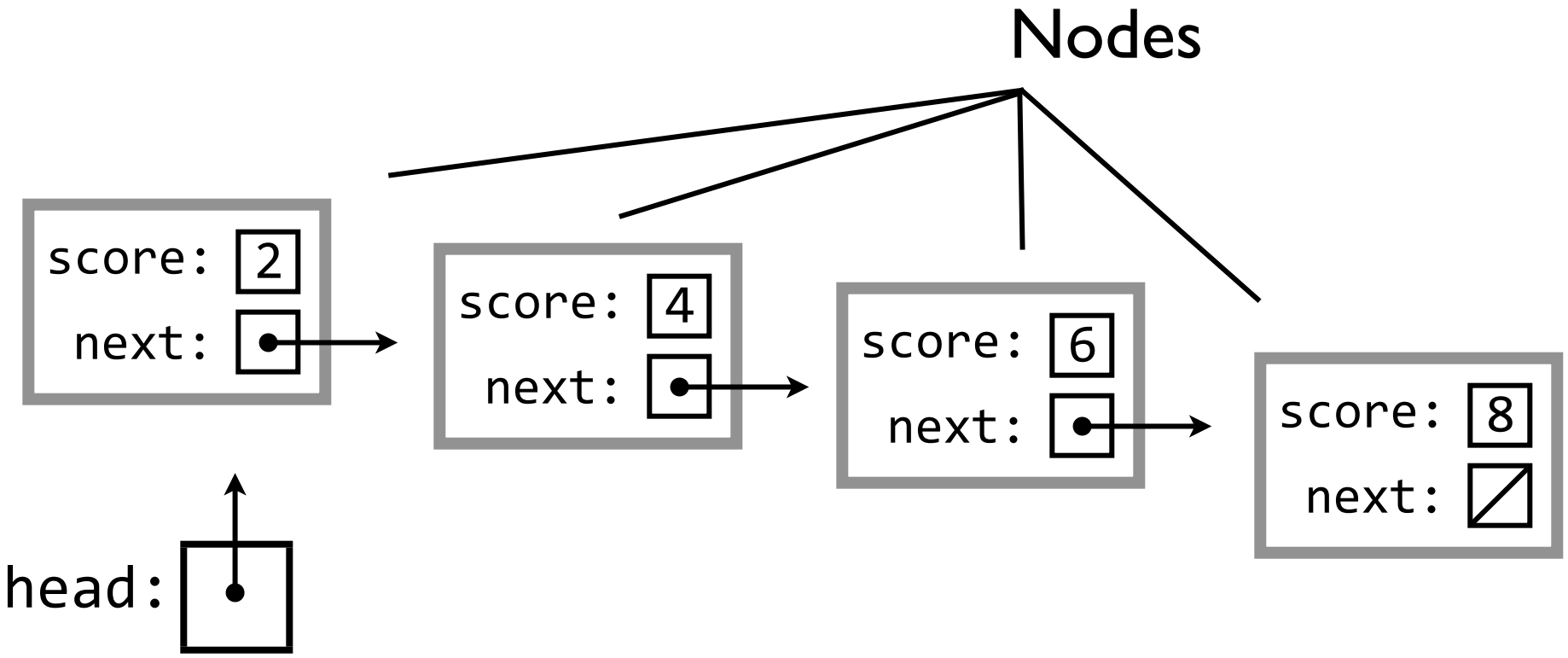
A Pointer to Another Node?

```
typedef struct node
{
   int score;
   struct node *next;
} Node;
```



Terminology

Whole thing is a linked list



First node is the head
(what you normally have a pointer to)

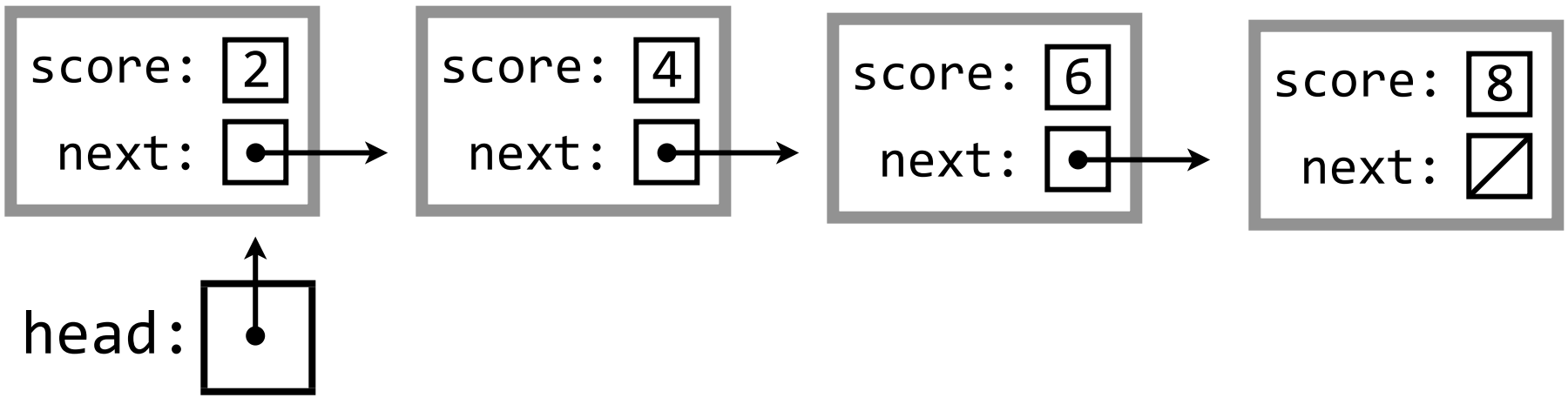
Last node is the tail

Traversing Arrays

```
void printArray(int a[], int n)
{
  for (int i = 0; i < n; i++)
  {
    printf("%d\n", a[i]);
  }
}
```

Traversing Linked Lists

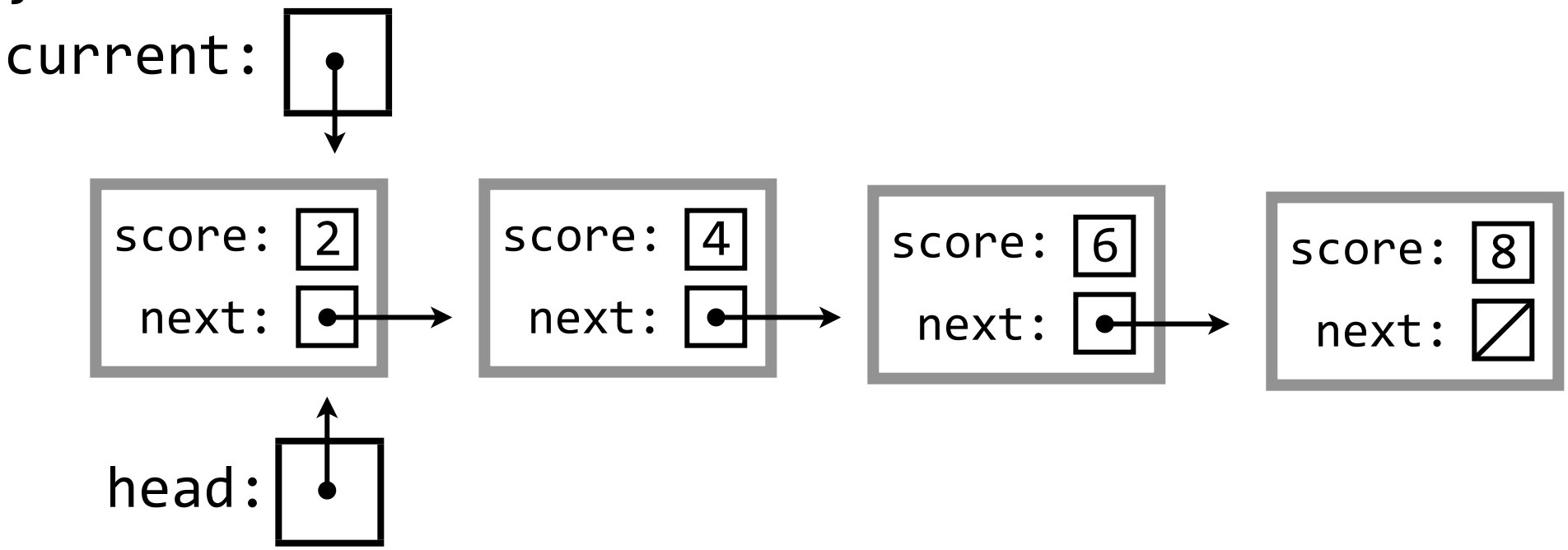
```
void printList(Node *head)
{
    Node *current = head;
    while (current != NULL)
    {
        printf("%d\n", current->score);
        current = current->next;
    }
}
```



Traversing Linked Lists

```
void printList(Node *head)
{
    Node *current = head;
    while (current != NULL)
    {
        printf("%d\n", current->score);
        current = current->next;
    }
}
```

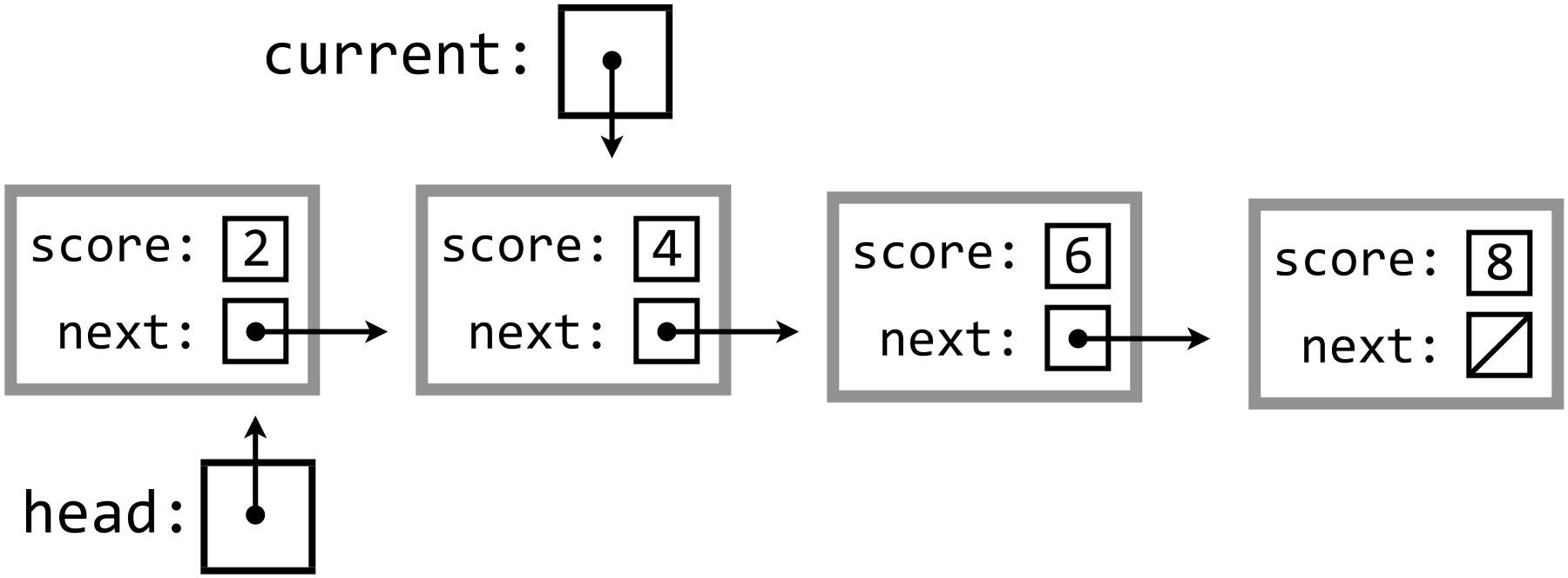
2



Traversing Linked Lists

```
void printList(Node *head)
{
    Node *current = head;
    while (current != NULL)
    {
        printf("%d\n", current->score);
        current = current->next;
    }
}
```

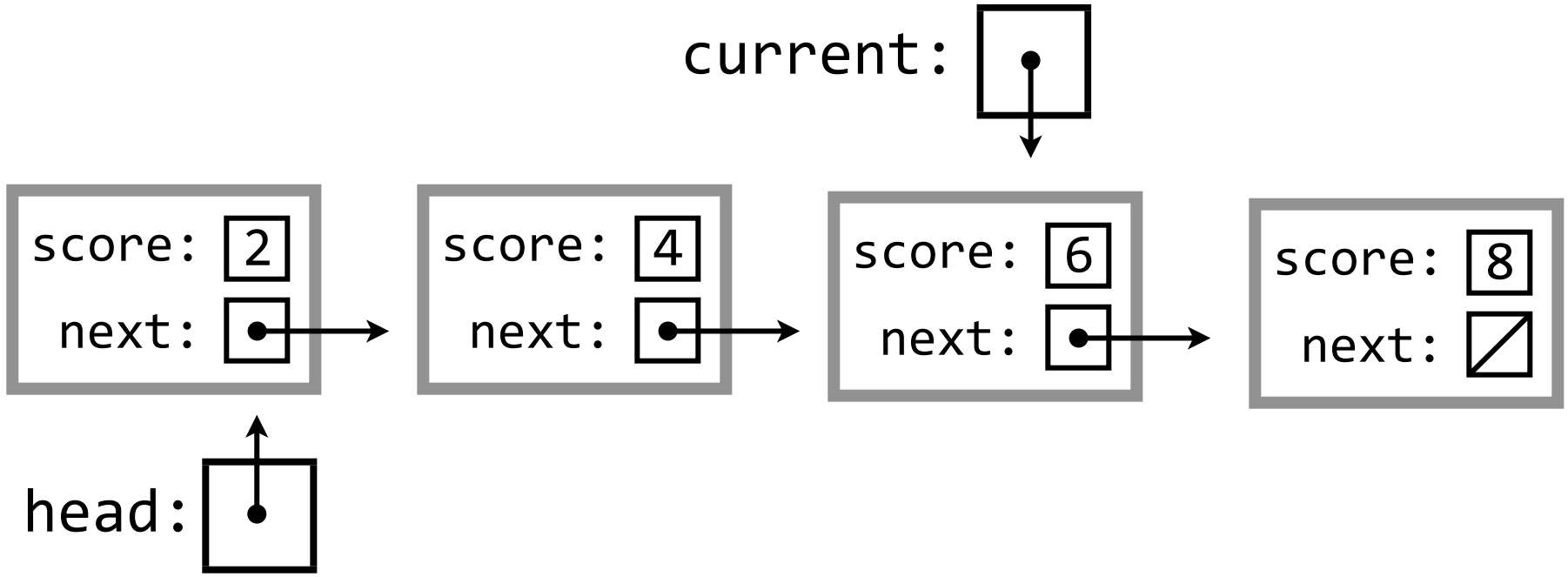
```
2
4
```



Traversing Linked Lists

```
void printList(Node *head)
{
    Node *current = head;
    while (current != NULL)
    {
        printf("%d\n", current->score);
        current = current->next;
    }
}
```

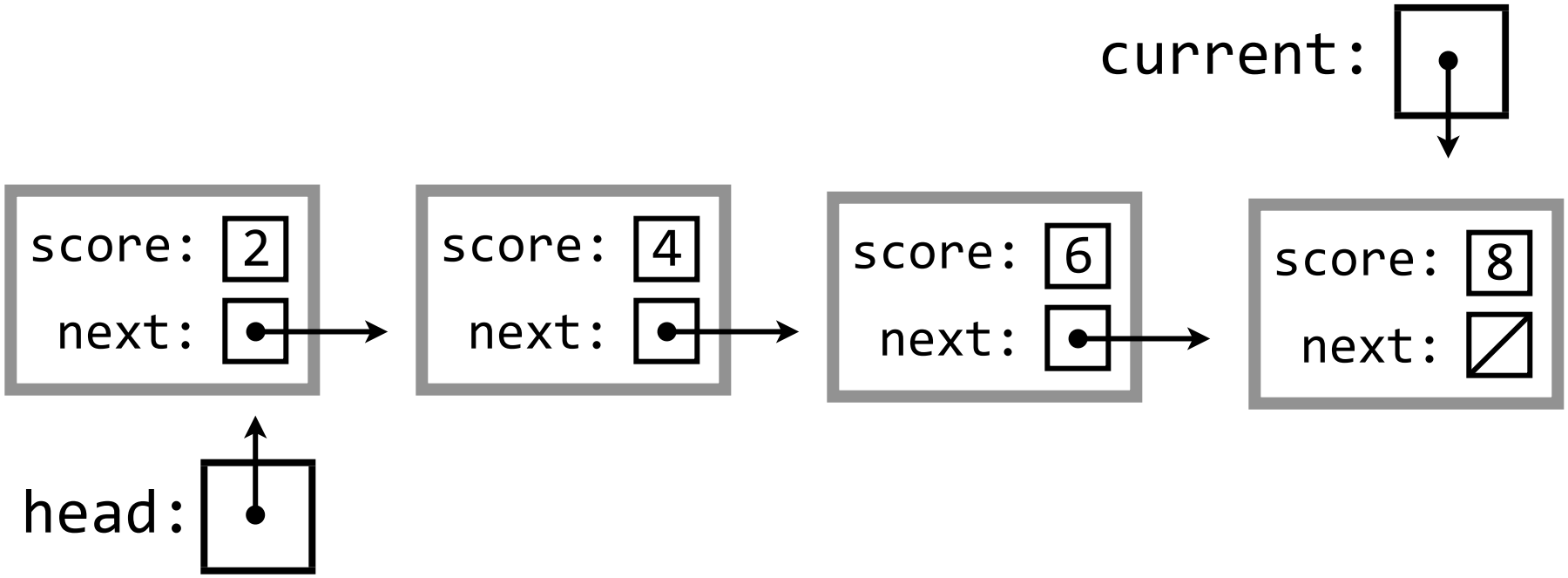
```
2
4
6
```



Traversing Linked Lists

```
void printList(Node *head)
{
    Node *current = head;
    while (current != NULL)
    {
        printf("%d\n", current->score);
        current = current->next;
    }
}
```

```
2
4
6
8
```

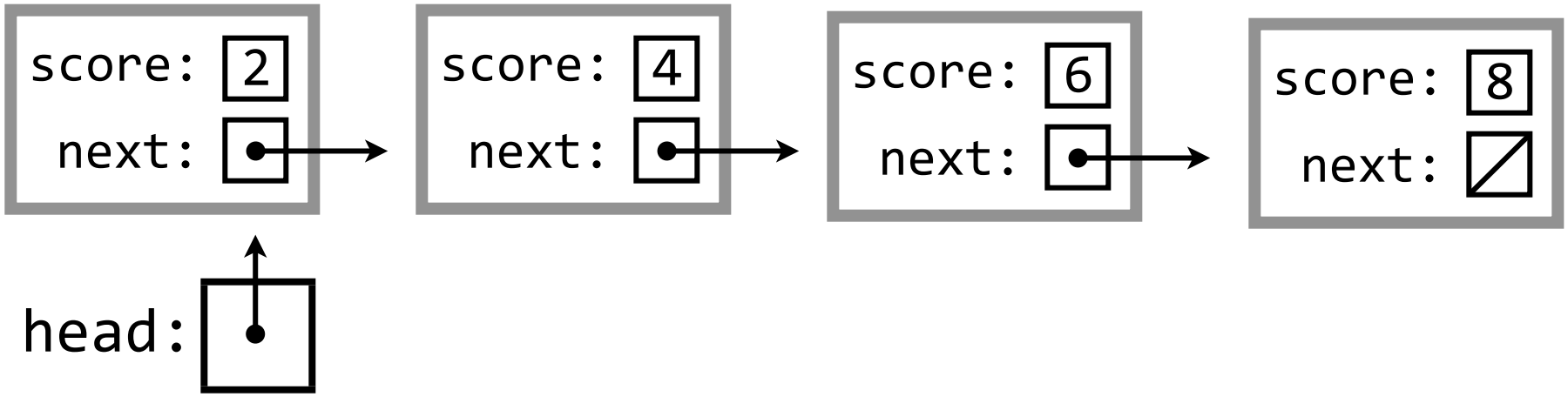


Traversing Linked Lists

```
void printList(Node *head)
{
    Node *current = head;
    while (current != NULL)
    {
        printf("%d\n", current->score);
        current = current->next;
    }
}
```

```
2
4
6
8
```

current:



Verdict?

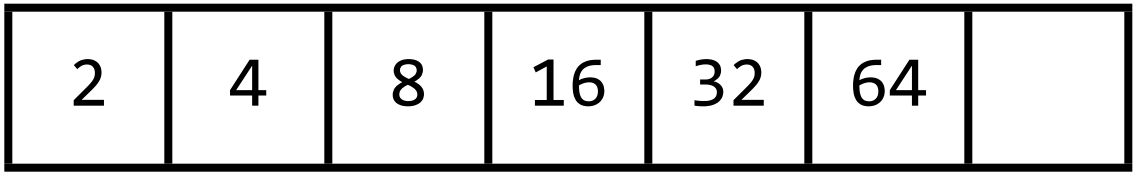
- Basically the same

Arrays vs. Linked Lists

- Calculating length of list
- Accessing `i`th element
- Insert at end of list
- Insert at beginning of list

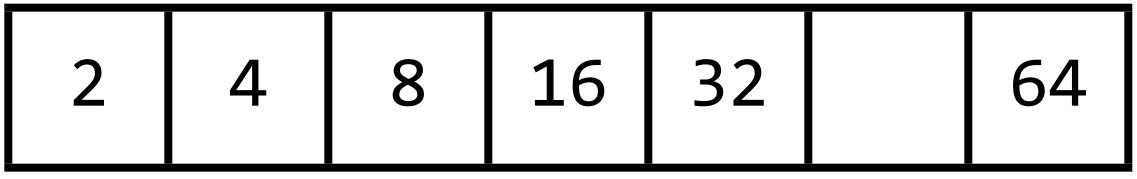
Insert in Middle of Array

Want to insert 1 at a[3]



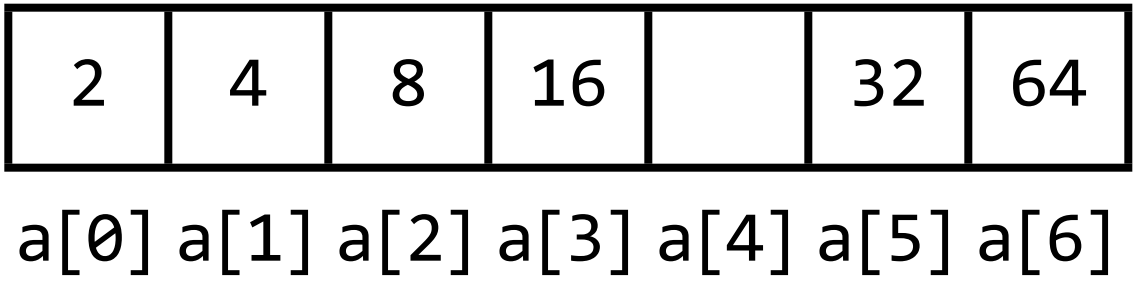
Insert in Middle of Array

Want to insert 1 at a[3]



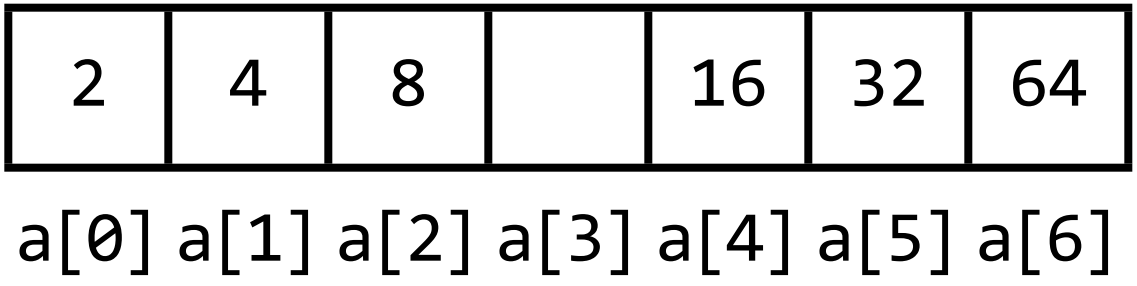
Insert in Middle of Array

Want to insert 1 at a[3]



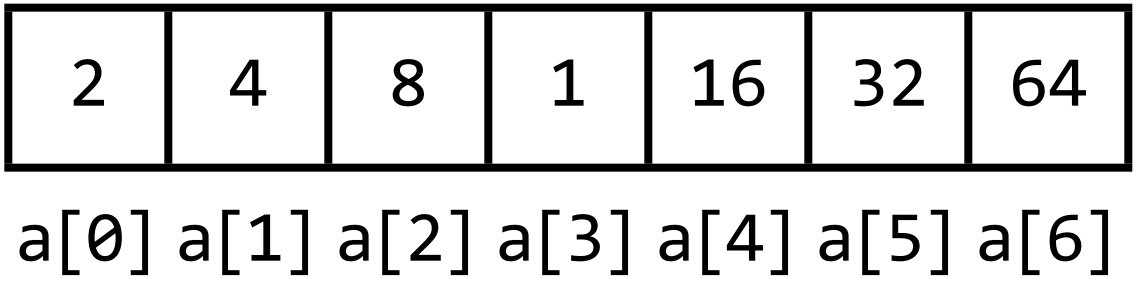
Insert in Middle of Array

Want to insert 1 at a[3]



Insert in Middle of Array

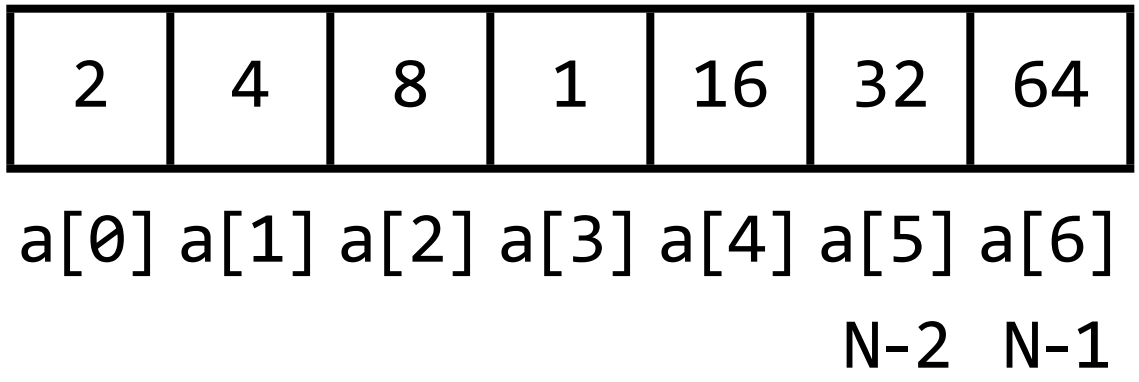
Want to insert 1 at a[3]



Insert in Middle of Array

```
// Assuming the array is large enough
for (int i = N - 2; i >= 3; i--)
{
  a[i + 1] = a[i];
}
a[3] = 1;
```

Want to insert 1 at a[3]

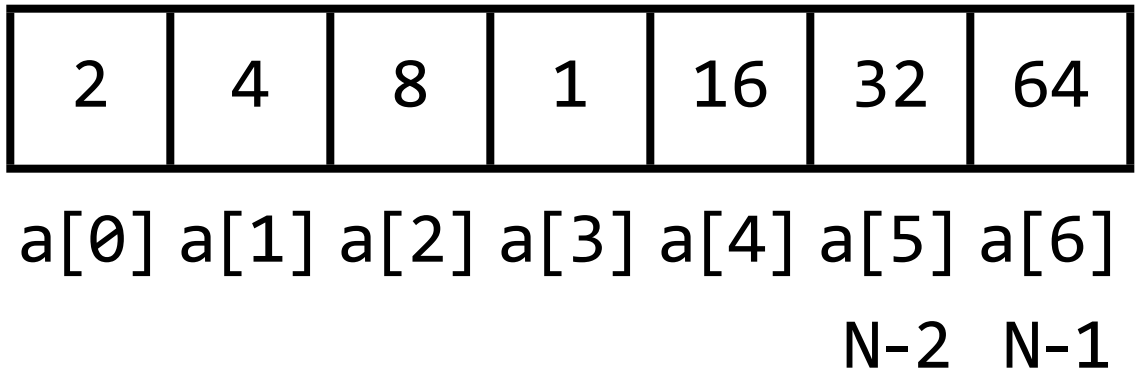


Insert in Middle of Array

```
// Assuming the array is large enough
for (int i = N - 2; i >= k; i--)
{
  a[i + 1] = a[i];
}
a[k] = x;
```

Want to insert 1 at a[3]

Want to insert x at a[k]



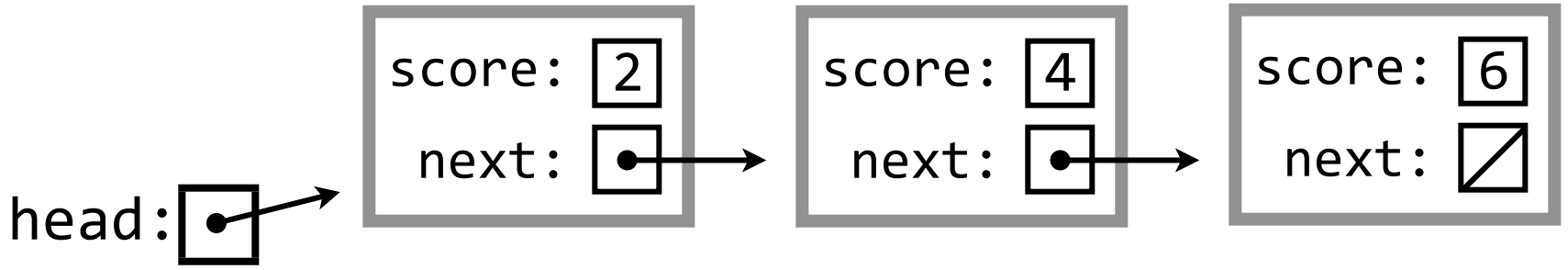
Insert at Middle of Linked List

Insert at End of Linked List:

- Find the end of the list
- Update that node's pointer to point at the new node
- Make sure the new node's pointer is NULL

Insert at Beginning of Linked List:

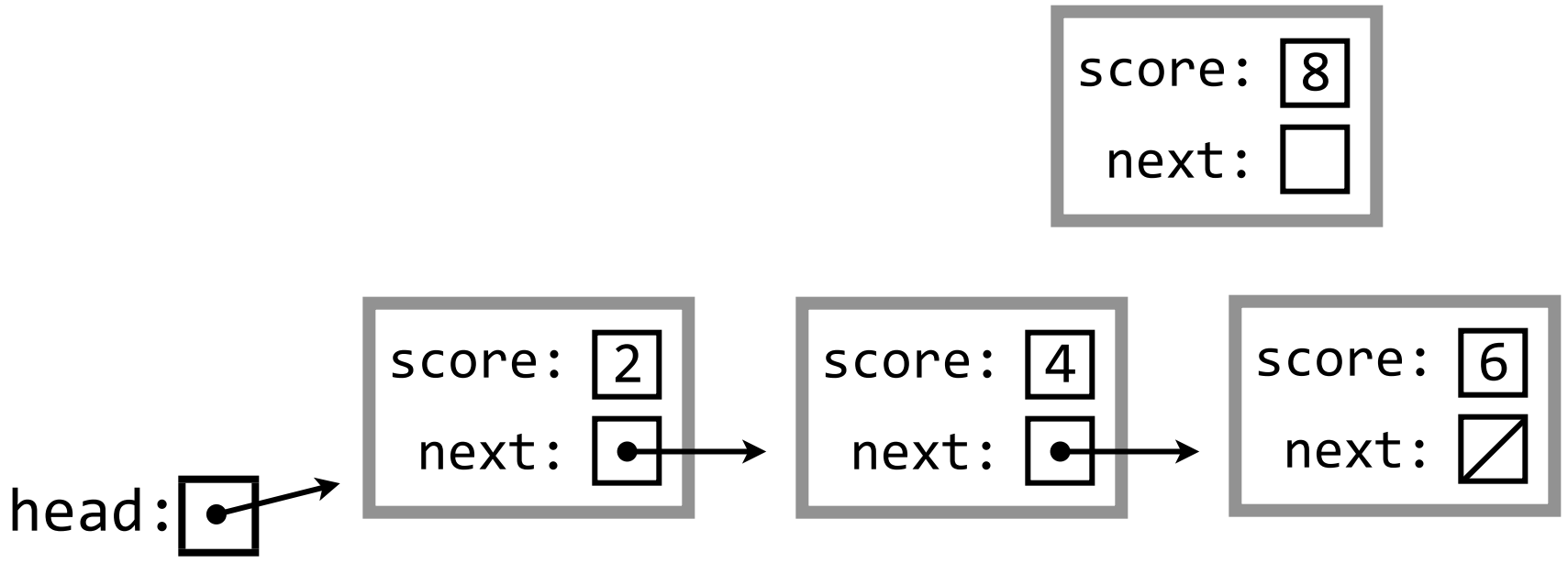
- Find the beginning of the list
- Point the new node's pointer at that node
- Make sure head points to the new node



Insert at Middle of Linked List

Insert at Middle of Linked List:

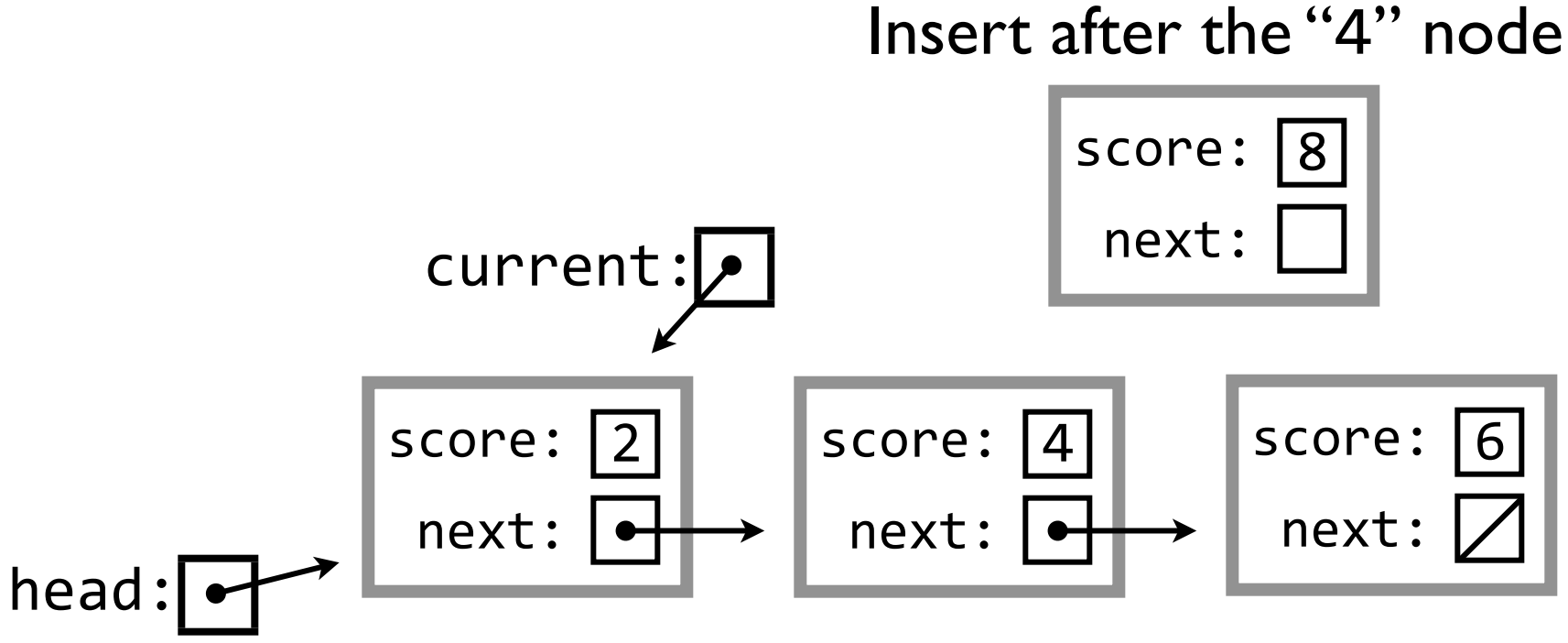
- Find the node immediately before the insertion point
- Point the new node's pointer at that node's next node
- Point that node's pointer at the new node



Insert at Middle of Linked List

Insert at Middle of Linked List:

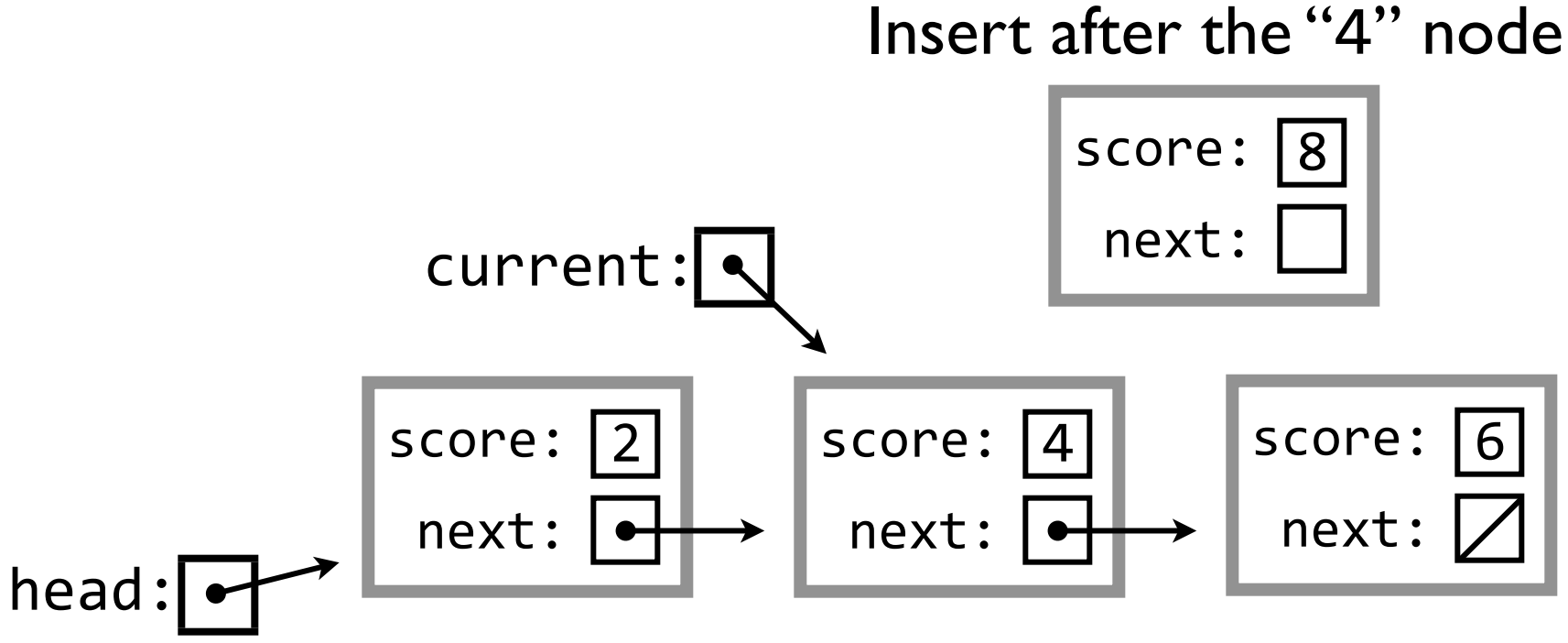
- Find the node immediately before the insertion point
- Point the new node's pointer at that node's next node
- Point that node's pointer at the new node



Insert at Middle of Linked List

Insert at Middle of Linked List:

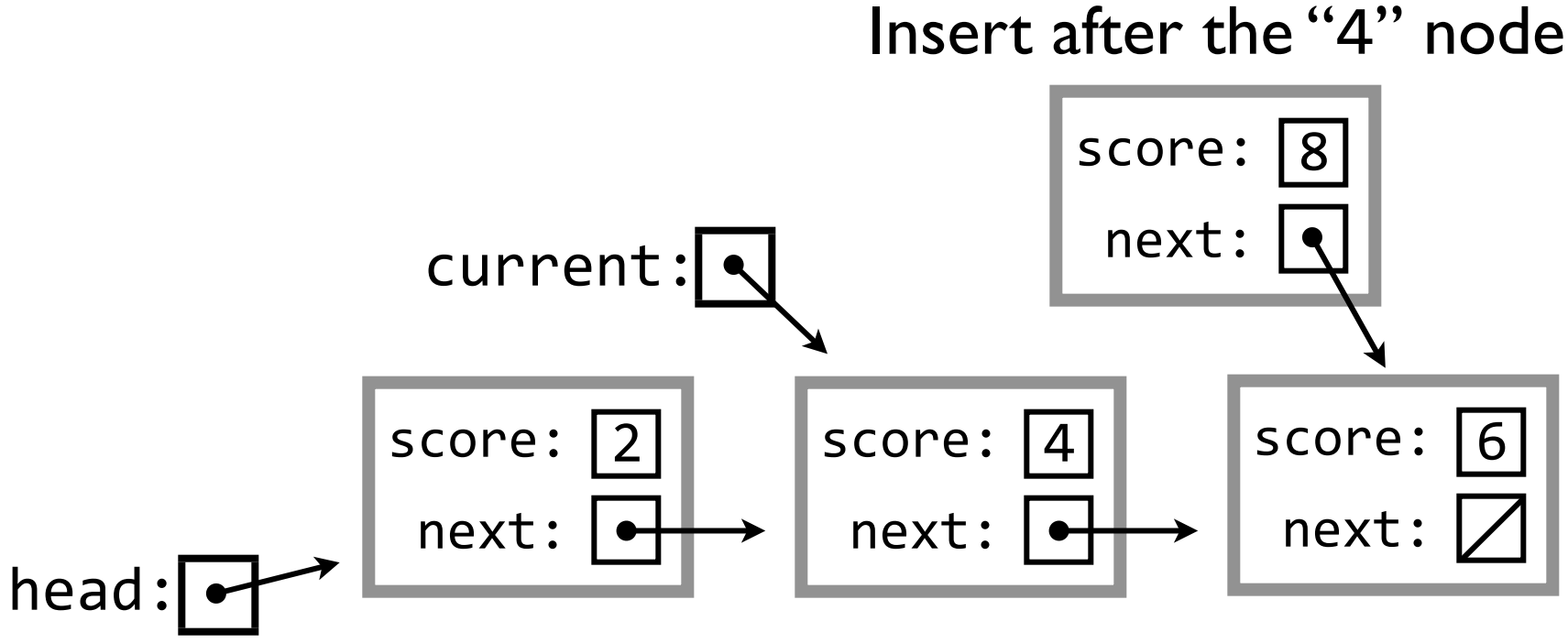
- Find the node immediately before the insertion point
- Point the new node's pointer at that node's next node
- Point that node's pointer at the new node



Insert at Middle of Linked List

Insert at Middle of Linked List:

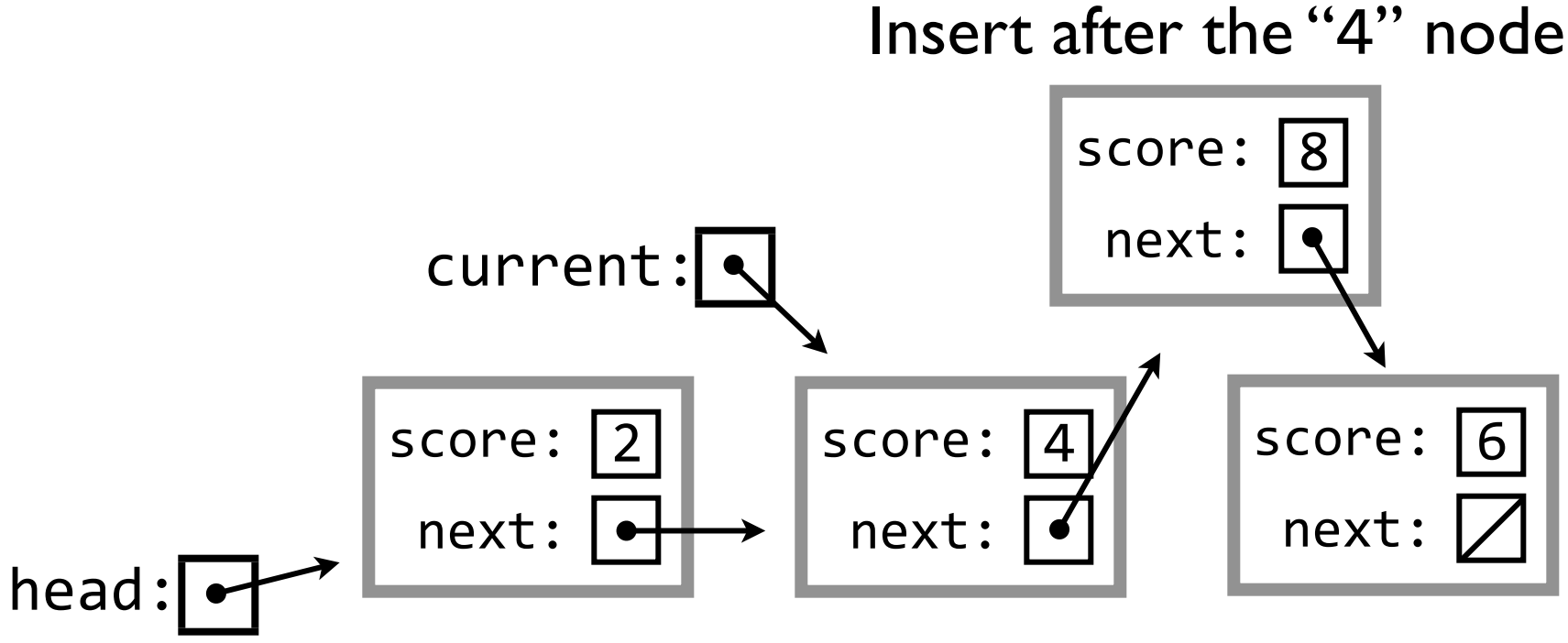
- Find the node immediately before the insertion point
- Point the new node's pointer at that node's next node
- Point that node's pointer at the new node



Insert at Middle of Linked List

Insert at Middle of Linked List:

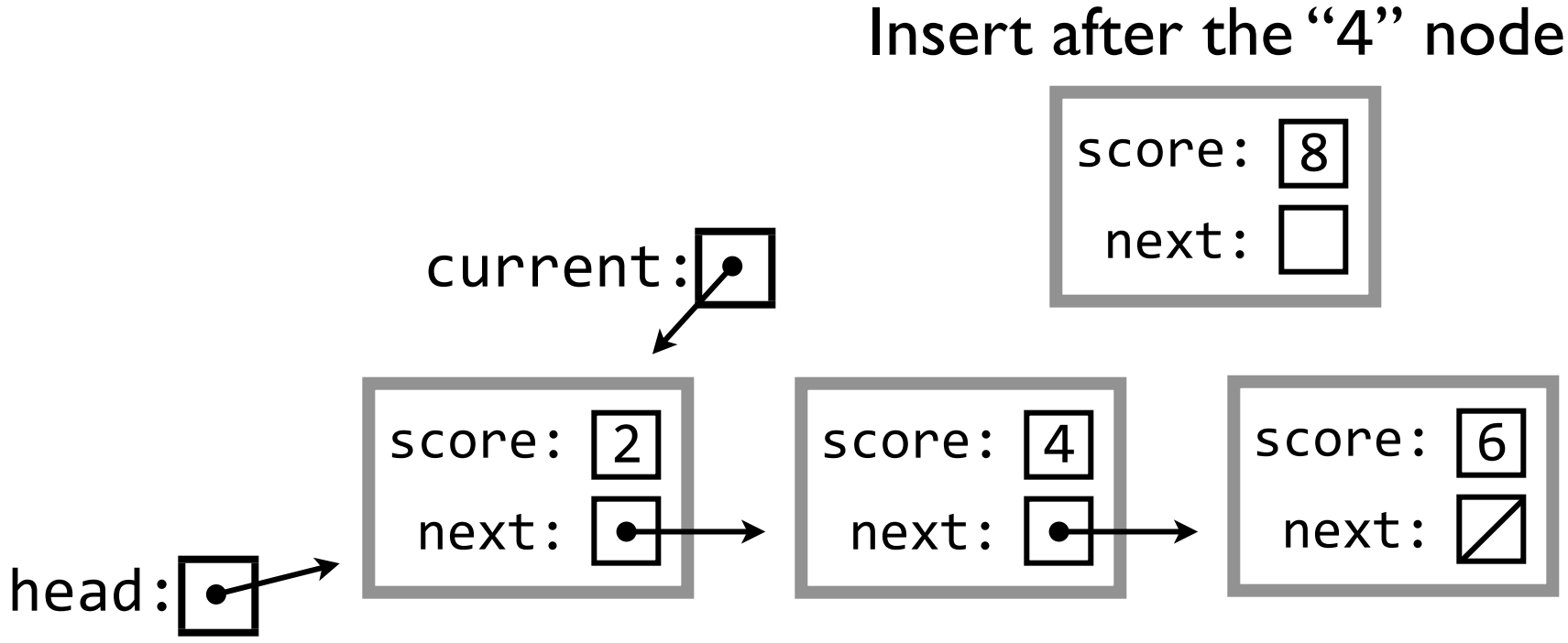
- Find the node immediately before the insertion point
- Point the new node's pointer at that node's next node
- Point that node's pointer at the new node



Order Matters

Insert at Middle of Linked List:

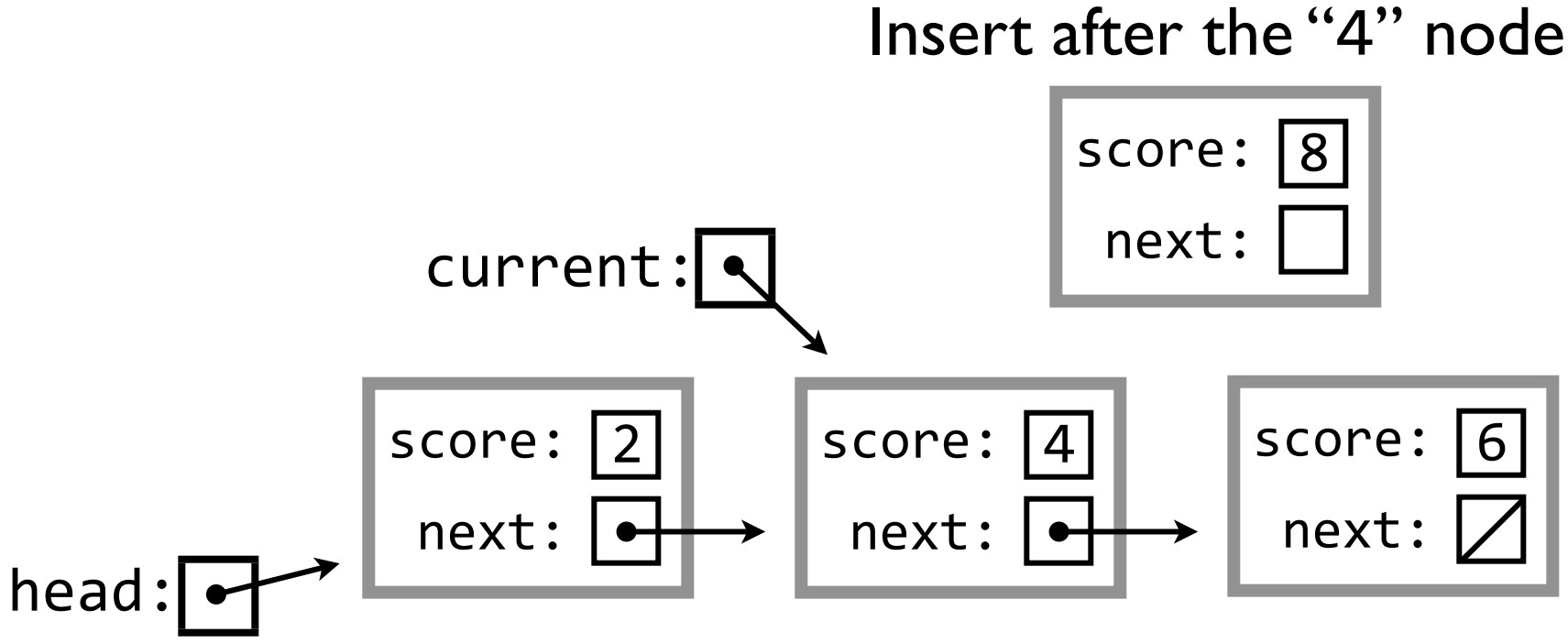
- Find the node immediately before the insertion point
- Point that node's pointer at the new node
- Point the new node's pointer at that node's next node



Order Matters

Insert at Middle of Linked List:

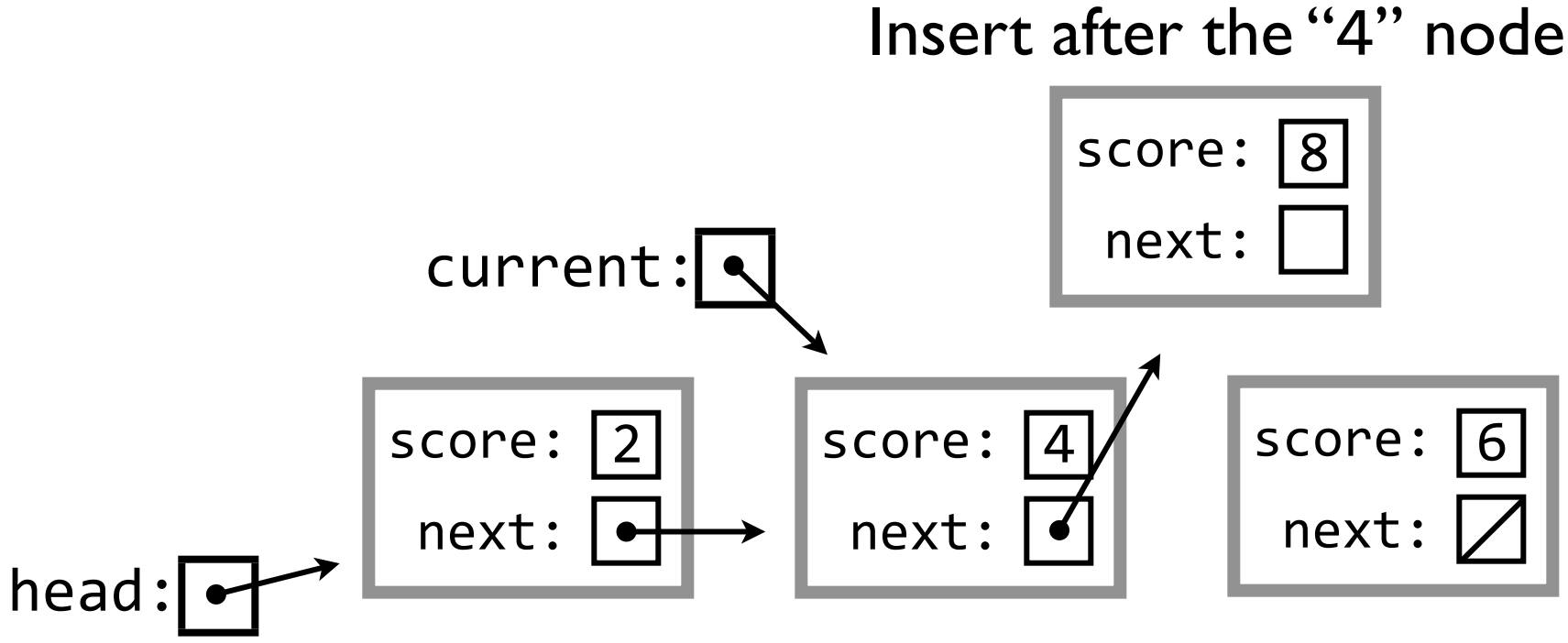
- Find the node immediately before the insertion point
- Point that node's pointer at the new node
- Point the new node's pointer at that node's next node



Order Matters

Insert at Middle of Linked List:

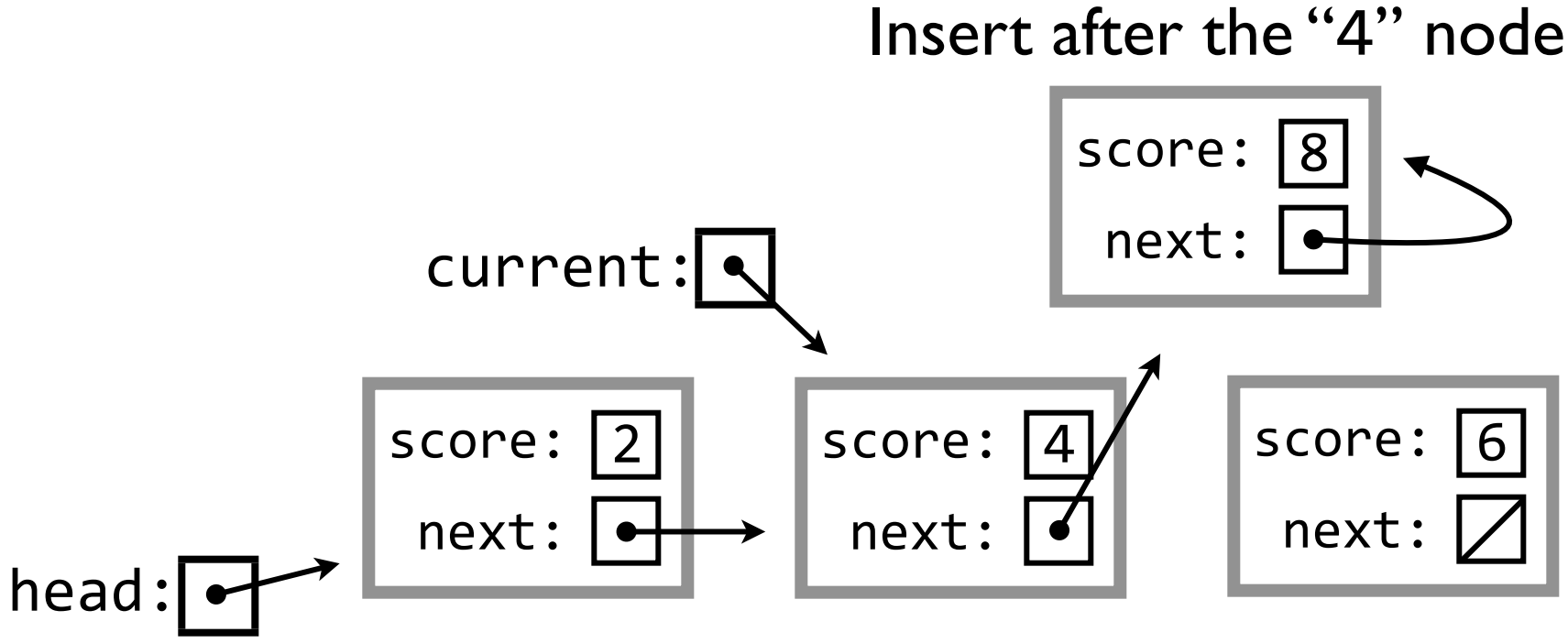
- Find the node immediately before the insertion point
- Point that node's pointer at the new node
- Point the new node's pointer at that node's next node



Order Matters

Insert at Middle of Linked List:

- Find the node immediately before the insertion point
- Point that node's pointer at the new node
- Point the new node's pointer at that node's next node



Order Matters

- For linked list operations, be very careful about the order you do things
- It's like swap
- Draw pictures

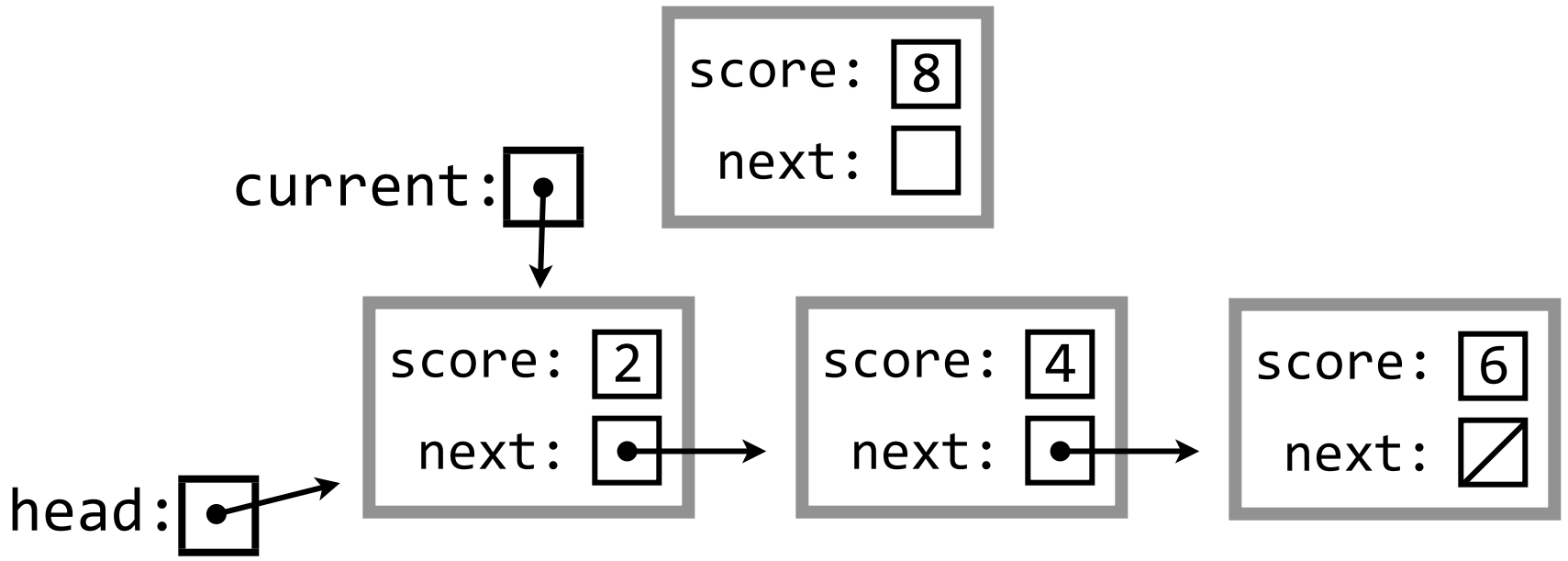
Insert at Middle of Linked List

```
Node *insert(Node *head, int index, Node *newNode)
{
  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

Wrong

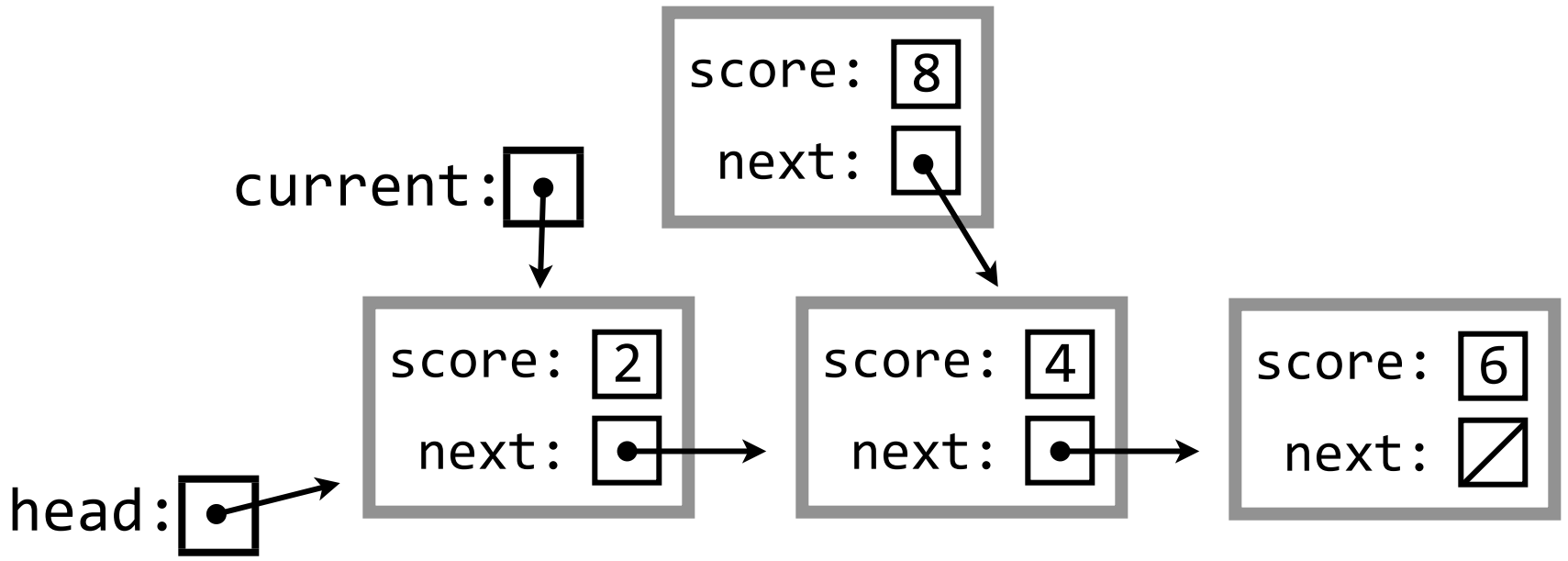
```
Node *insert(Node *head, int index, Node *newNode)
{
  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

```
head = insert(head, 0, newNode);
```



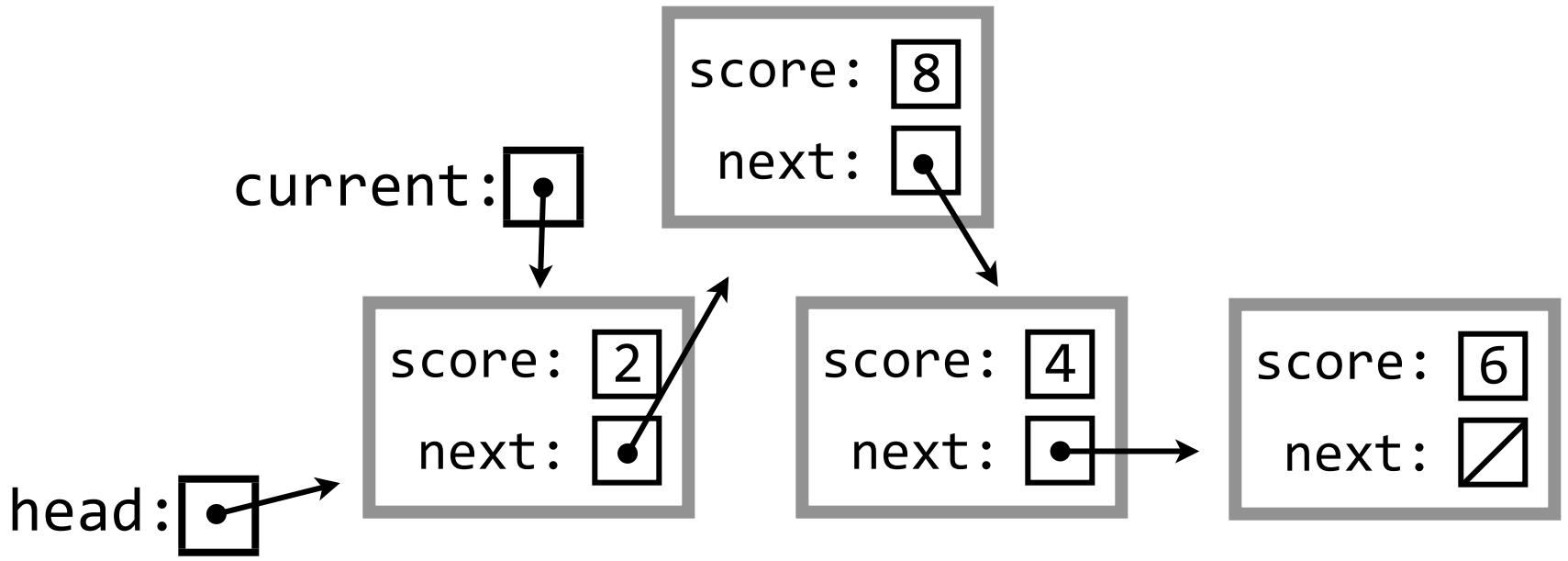
```
Node *insert(Node *head, int index, Node *newNode)
{
  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

```
head = insert(head, 0, newNode);
```



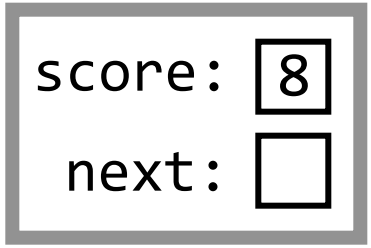
```
Node *insert(Node *head, int index, Node *newNode)
{
  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

```
head = insert(head, 0, newNode);
```



```
Node *insert(Node *head, int index, Node *newNode)
{
  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

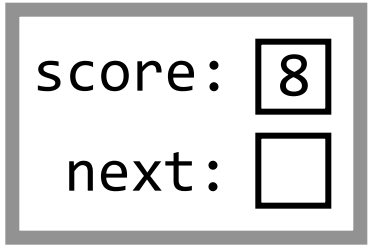
```
head = insert(head, 0, newNode);
```



head:

```
Node *insert(Node *head, int index, Node *newNode)
{
  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

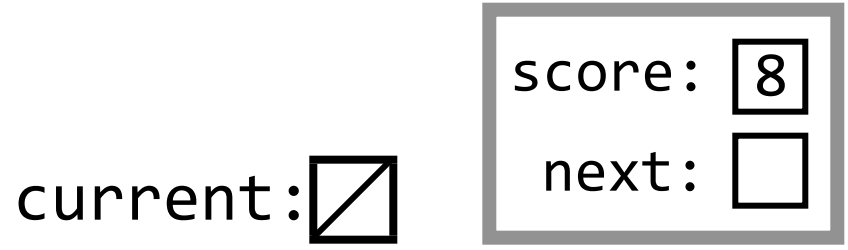
```
head = insert(head, 0, newNode);
```



head:

```
Node *insert(Node *head, int index, Node *newNode)
{
  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

```
head = insert(head, 0, newNode);
```

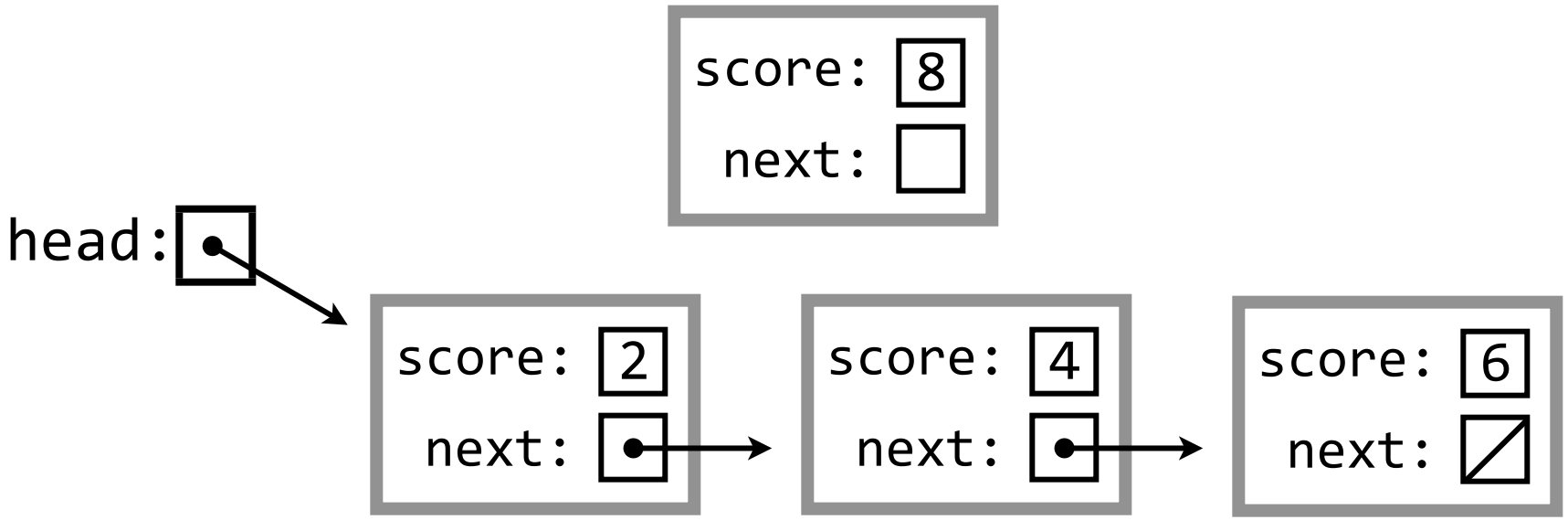


```
Node *insert(Node *head, int index, Node *newNode)
{
  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

```
Node *insert(Node *head, int index, Node *newNode)
{
  if (index == 0 || head == NULL)
  {
    newNode->next = head;
    return newNode;
  }

  Node *current = head;
  for (int i = 0; i < index-1 && current->next != NULL; i++)
  {
    current = current->next;
  }

  newNode->next = current->next;
  current->next = newNode;

  return head;
}
```

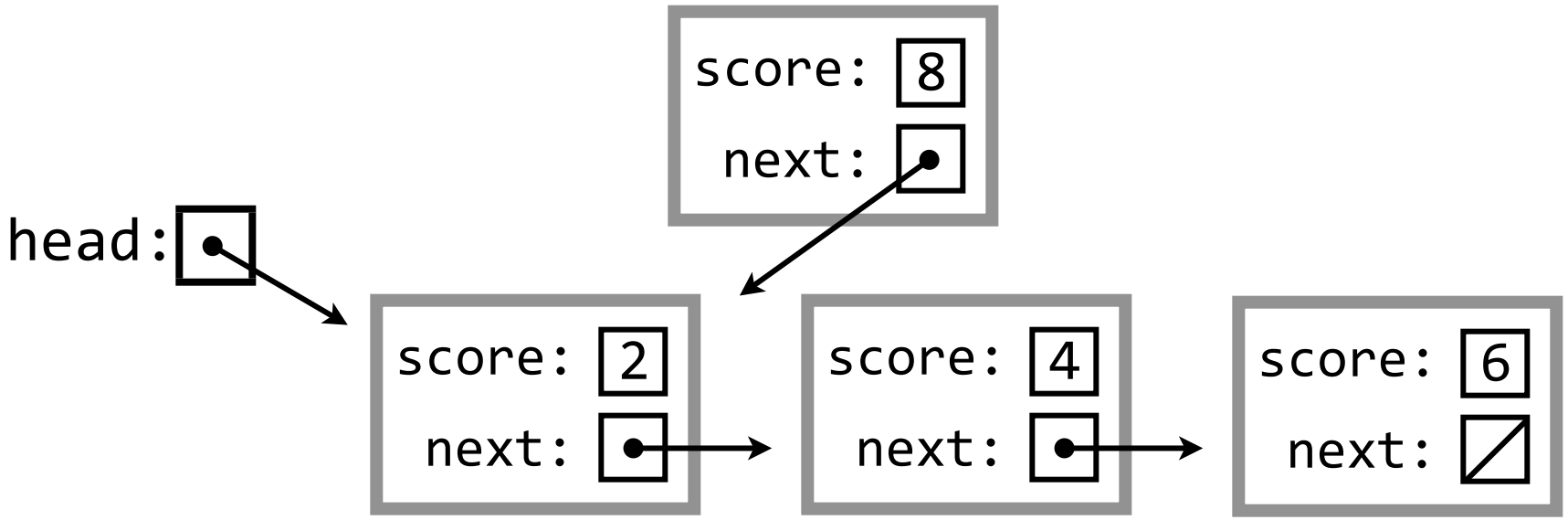
```
if (index == 0 || head == NULL)
{
  newNode->next = head;
  return newNode;
}
```

```
head = insert(head, 0, newNode);
```



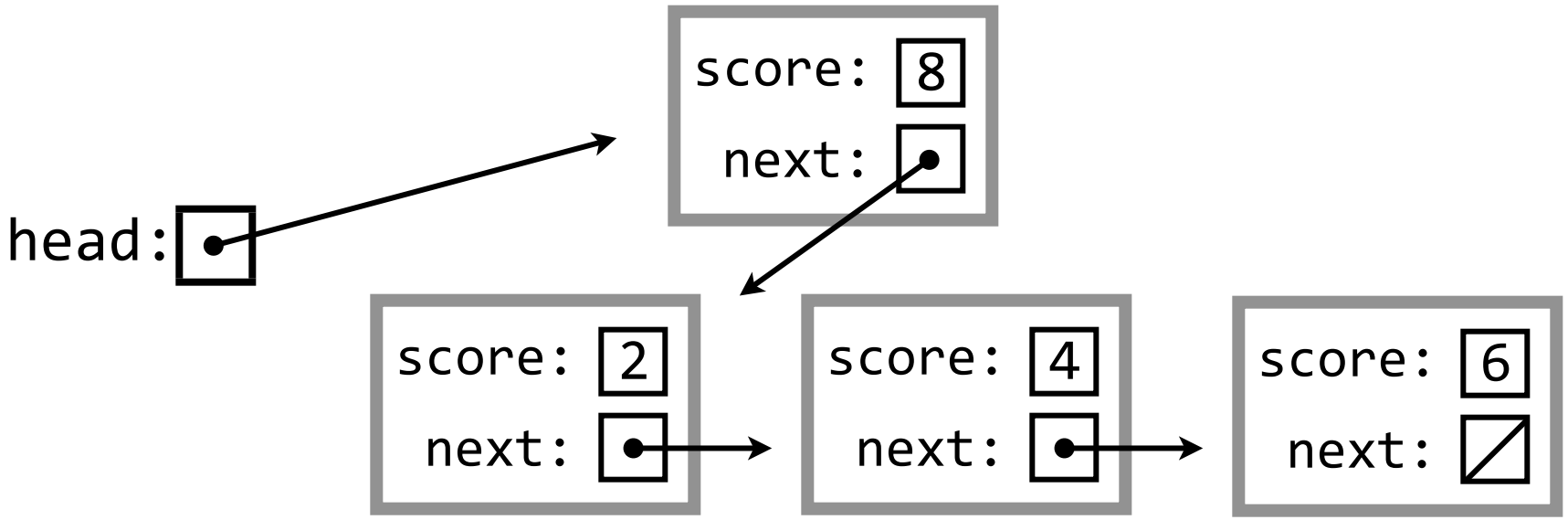
```
if (index == 0 || head == NULL)
{
  newNode->next = head;
  return newNode;
}
```

```
head = insert(head, 0, newNode);
```



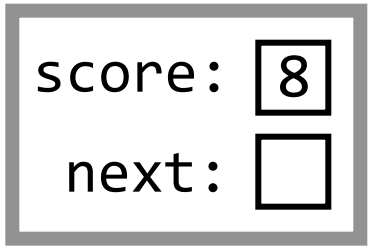
```
if (index == 0 || head == NULL)
{
  newNode->next = head;
  return newNode;
}
```

```
head = insert(head, 0, newNode);
```



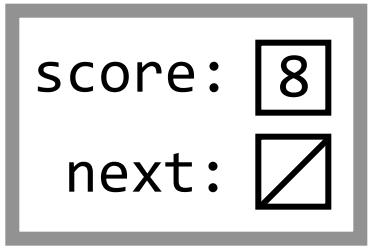
```
if (index == 0 || head == NULL)
{
  newNode->next = head;
  return newNode;
}
```

```
head = insert(head, 0, newNode);
```



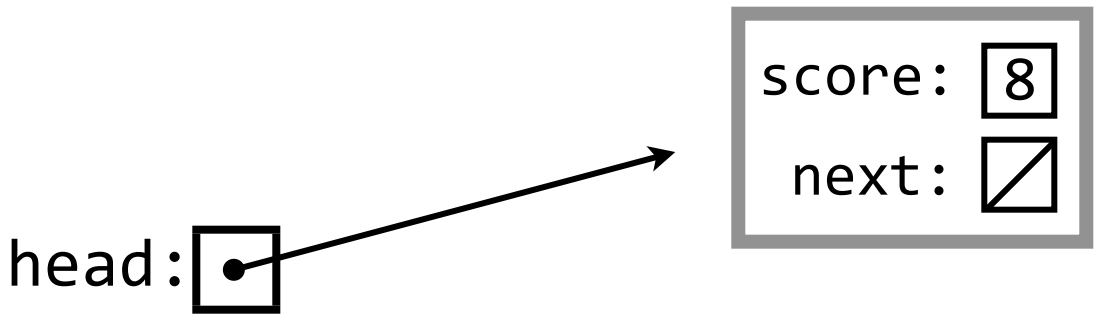
```
if (index == 0 || head == NULL)
{
  newNode->next = head;
  return newNode;
}
```

```
head = insert(head, 0, newNode);
```



```
if (index == 0 || head == NULL)
{
  newNode->next = head;
  return newNode;
}
```

```
head = insert(head, 0, newNode);
```



Verdict?

- Somewhere in between
- In some cases arrays could be better
- In general, linked lists are better

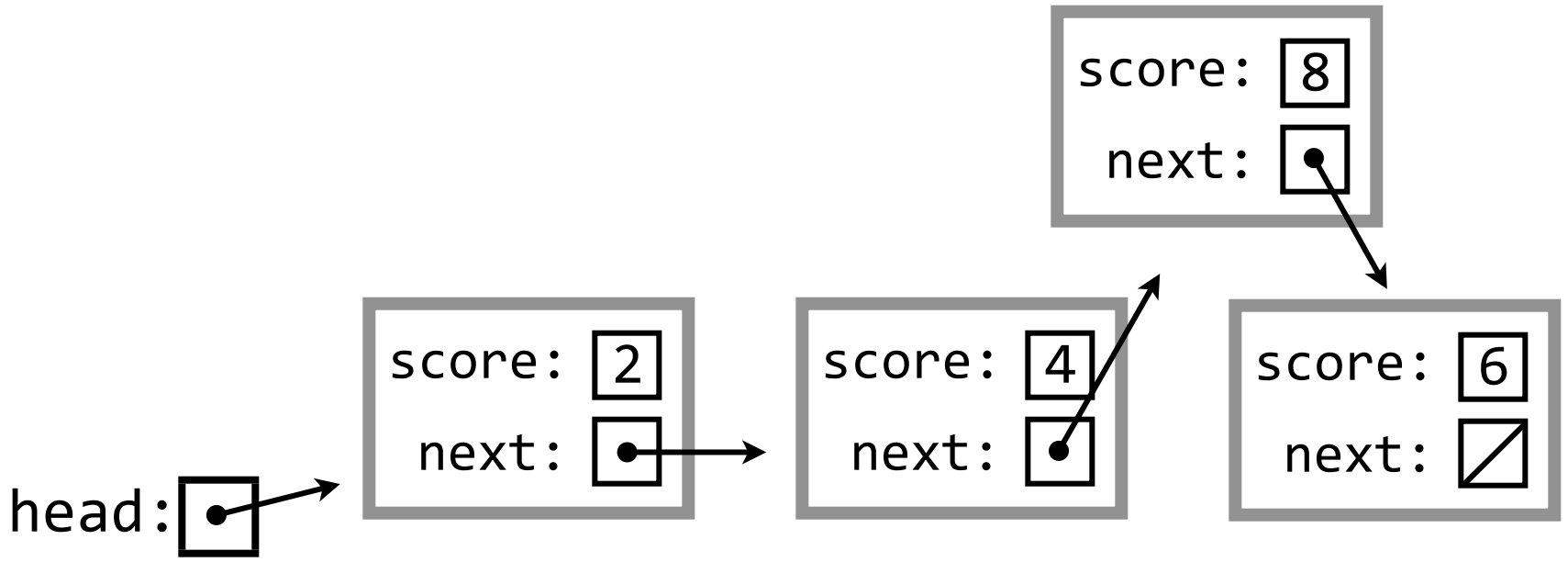
Delete From Array

- Trivial if we can leave and mark holes
- Otherwise need to shift everything

Delete From Linked List

Delete from Linked List:

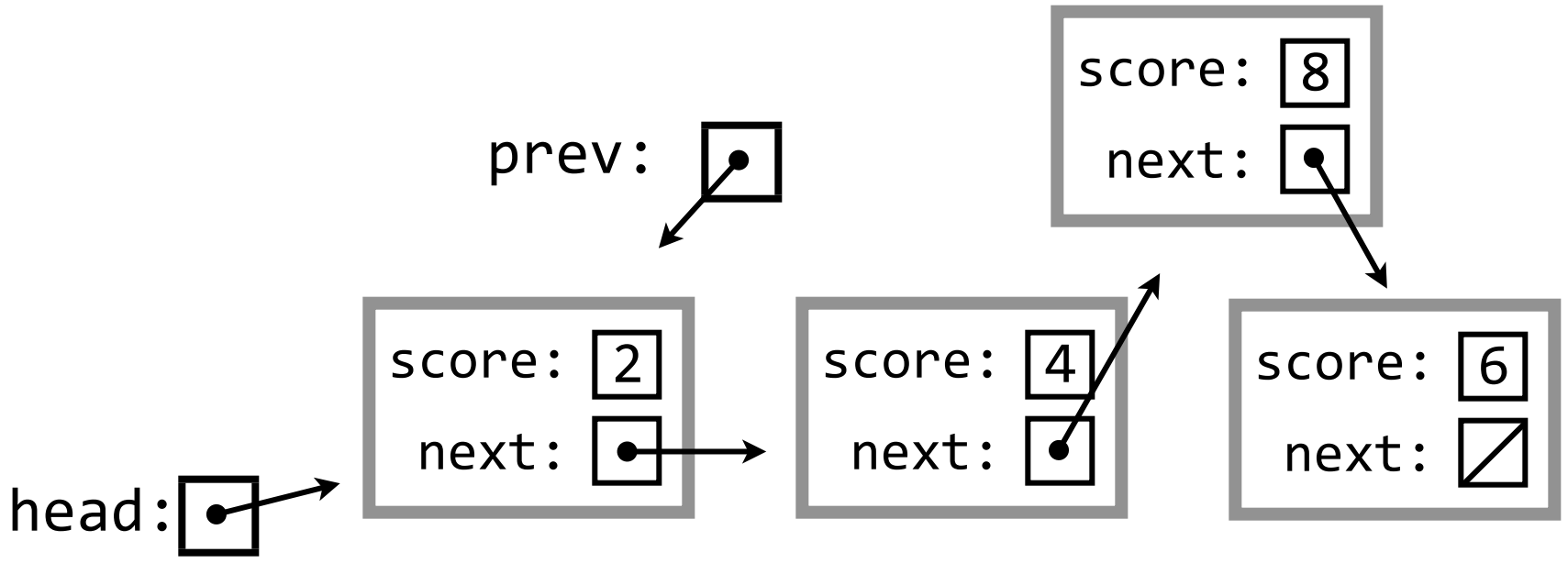
- Find the node immediately before the node being deleted
- Point that node's pointer at the deleted node's next node
- Clean up the deleted node, if necessary



Delete From Linked List

Delete from Linked List:

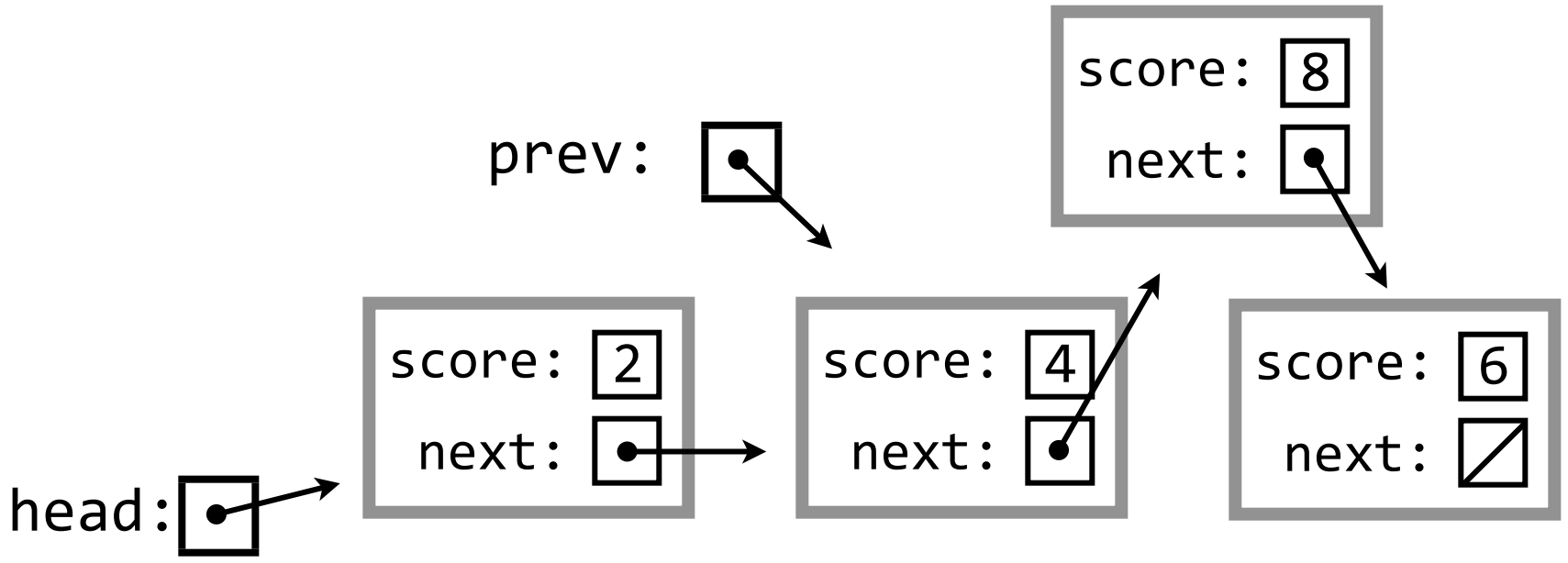
- Find the node immediately before the node being deleted
- Point that node's pointer at the deleted node's next node
- Clean up the deleted node, if necessary



Delete From Linked List

Delete from Linked List:

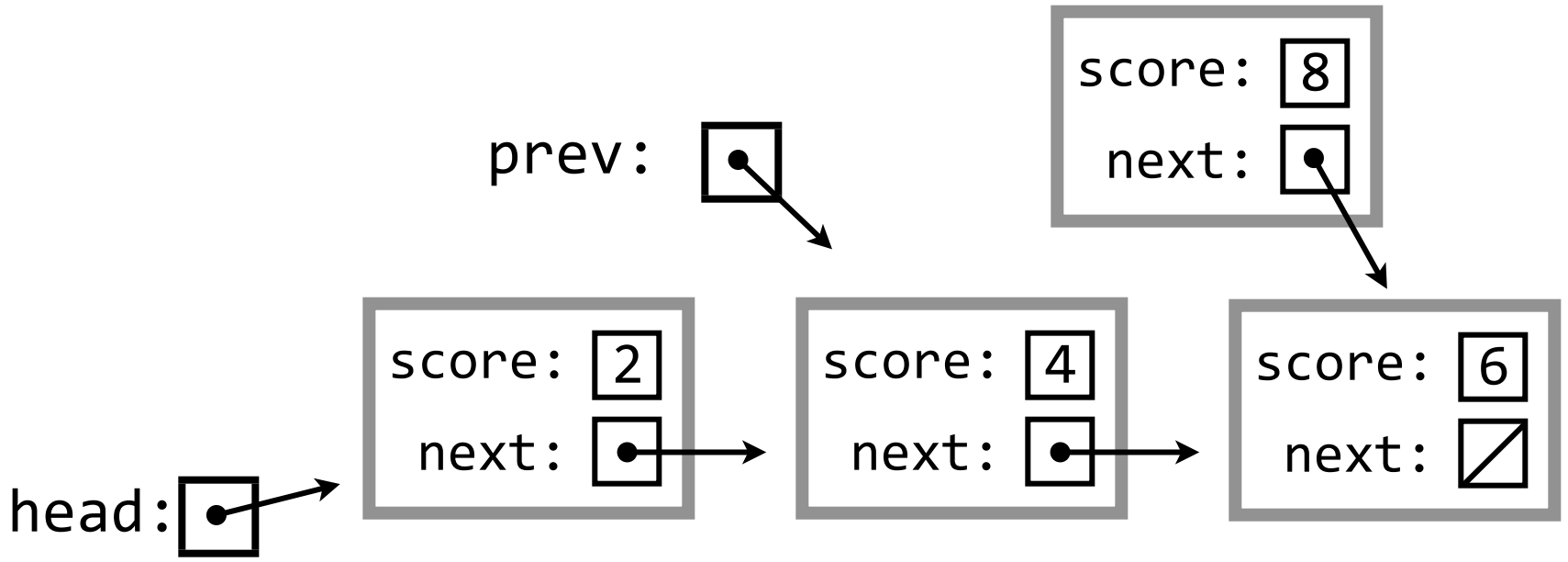
- Find the node immediately before the node being deleted
- Point that node's pointer at the deleted node's next node
- Clean up the deleted node, if necessary



Delete From Linked List

Delete from Linked List:

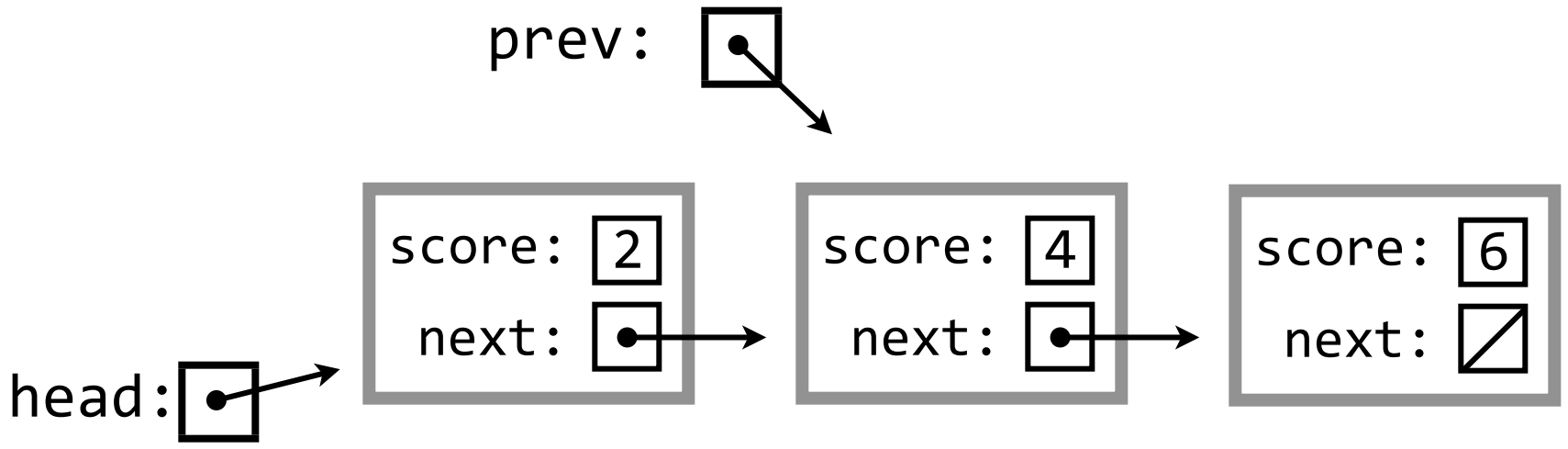
- Find the node immediately before the node being deleted
- Point that node's pointer at the deleted node's next node
- Clean up the deleted node, if necessary



Delete From Linked List

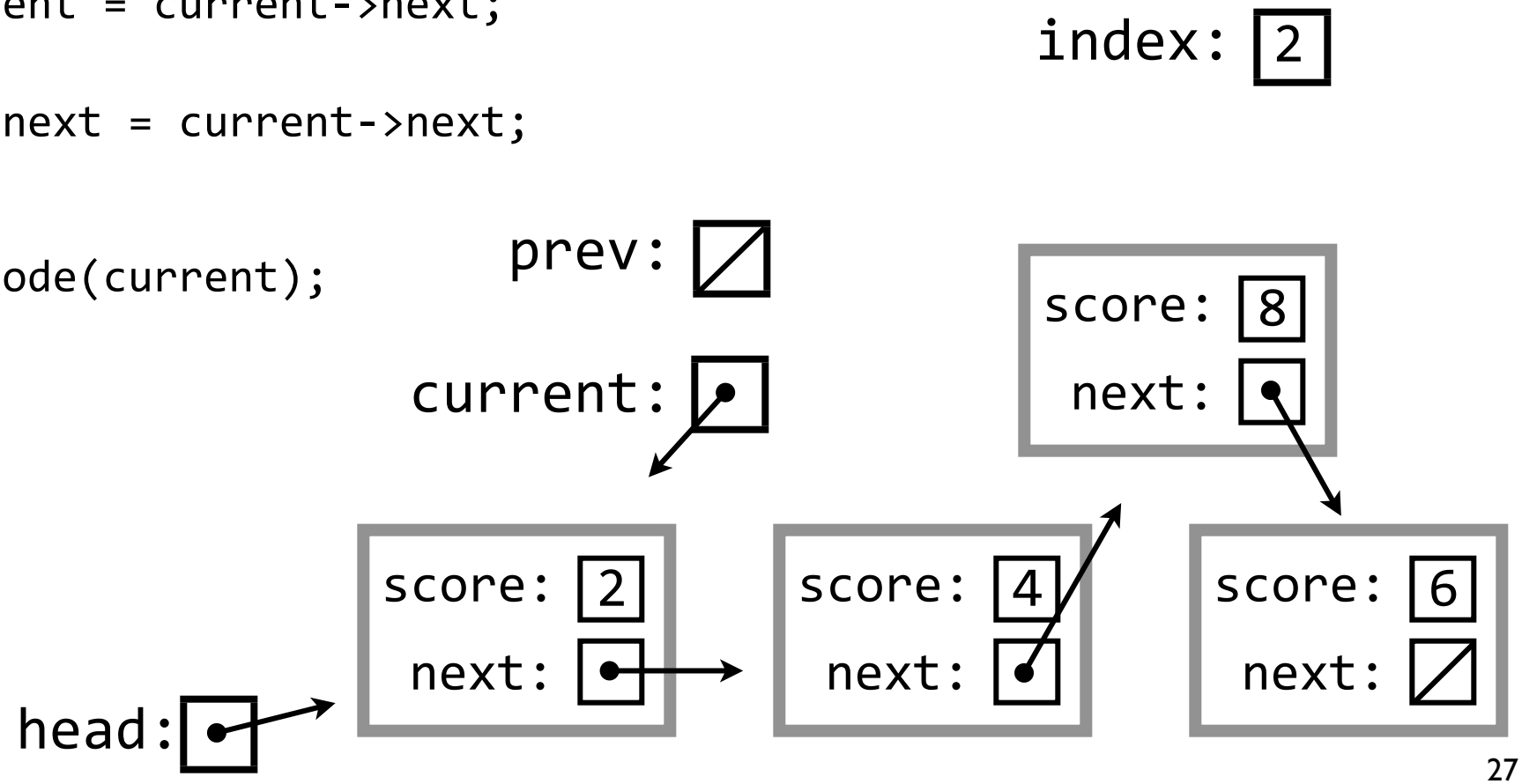
Delete from Linked List:

- Find the node immediately before the node being deleted
- Point that node's pointer at the deleted node's next node
- Clean up the deleted node, if necessary

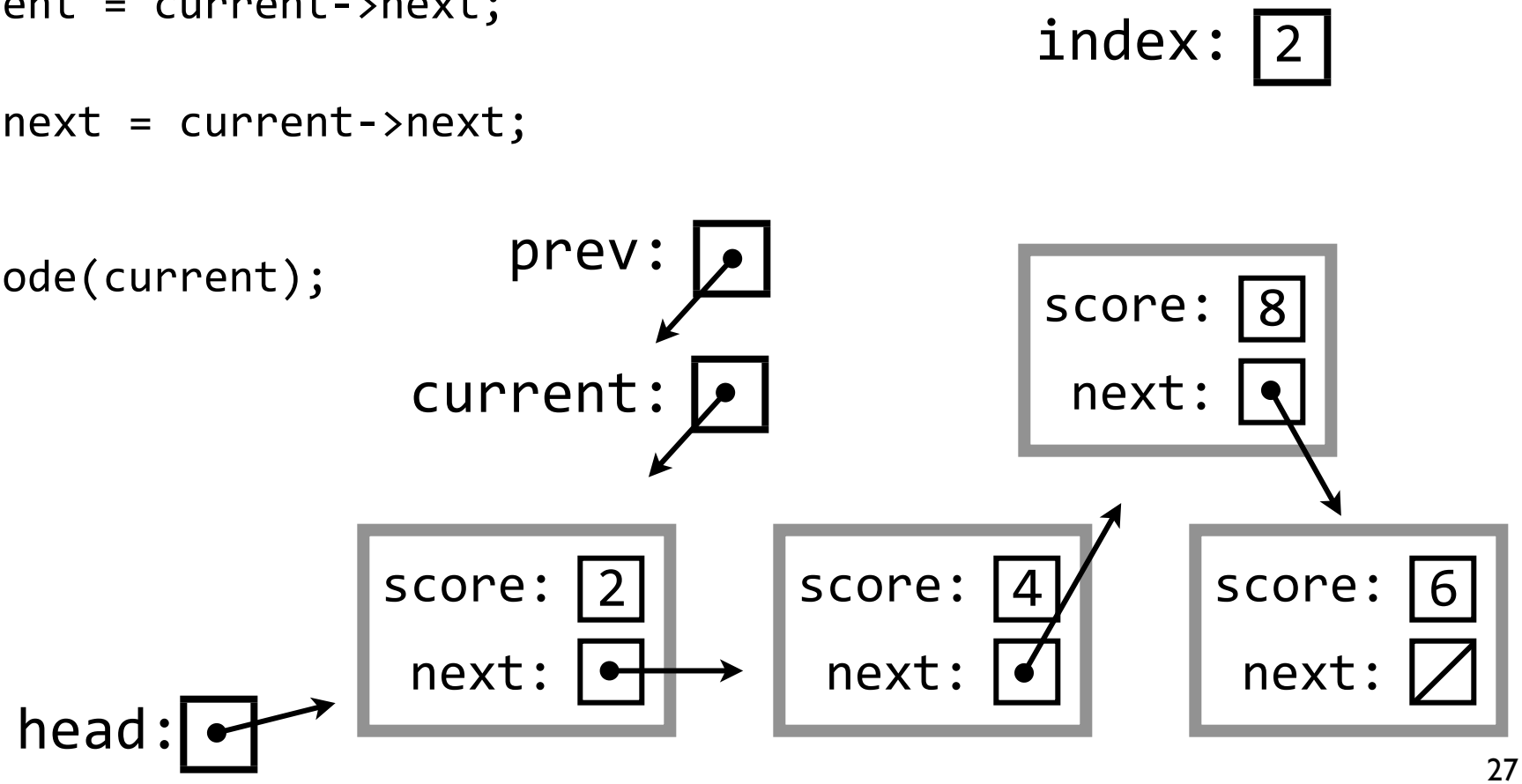


```
Node *delete(Node *head, int index)
{
  Node *current = head;
  Node *prev = NULL;

  if (index == 0)
  {
    head = head->next;
  }
  else
  {

    for (int i = 0; i < index && current != NULL; i++)
    {
      prev = current;
      current = current->next;
    }

    prev->next = current->next;

  }

  destroyNode(current);

  return head;
}
```

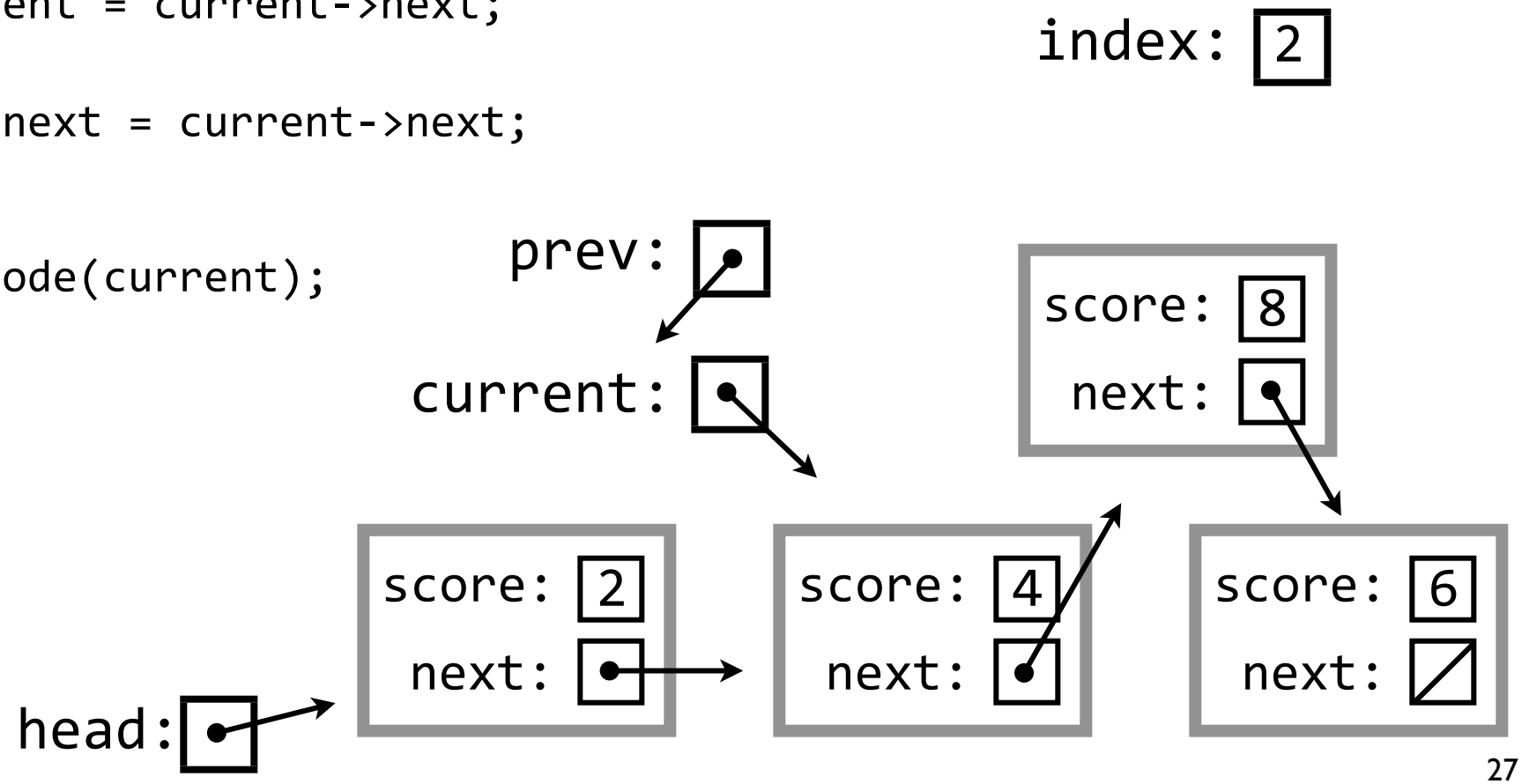
```
Node *current = head;
Node *prev = NULL;

if (index == 0)
{
  head = head->next;
}
else
{

  for (int i = 0; i < index && current != NULL; i++)
  {
    prev = current;
    current = current->next;
  }

  prev->next = current->next;

}

destroyNode(current);
```



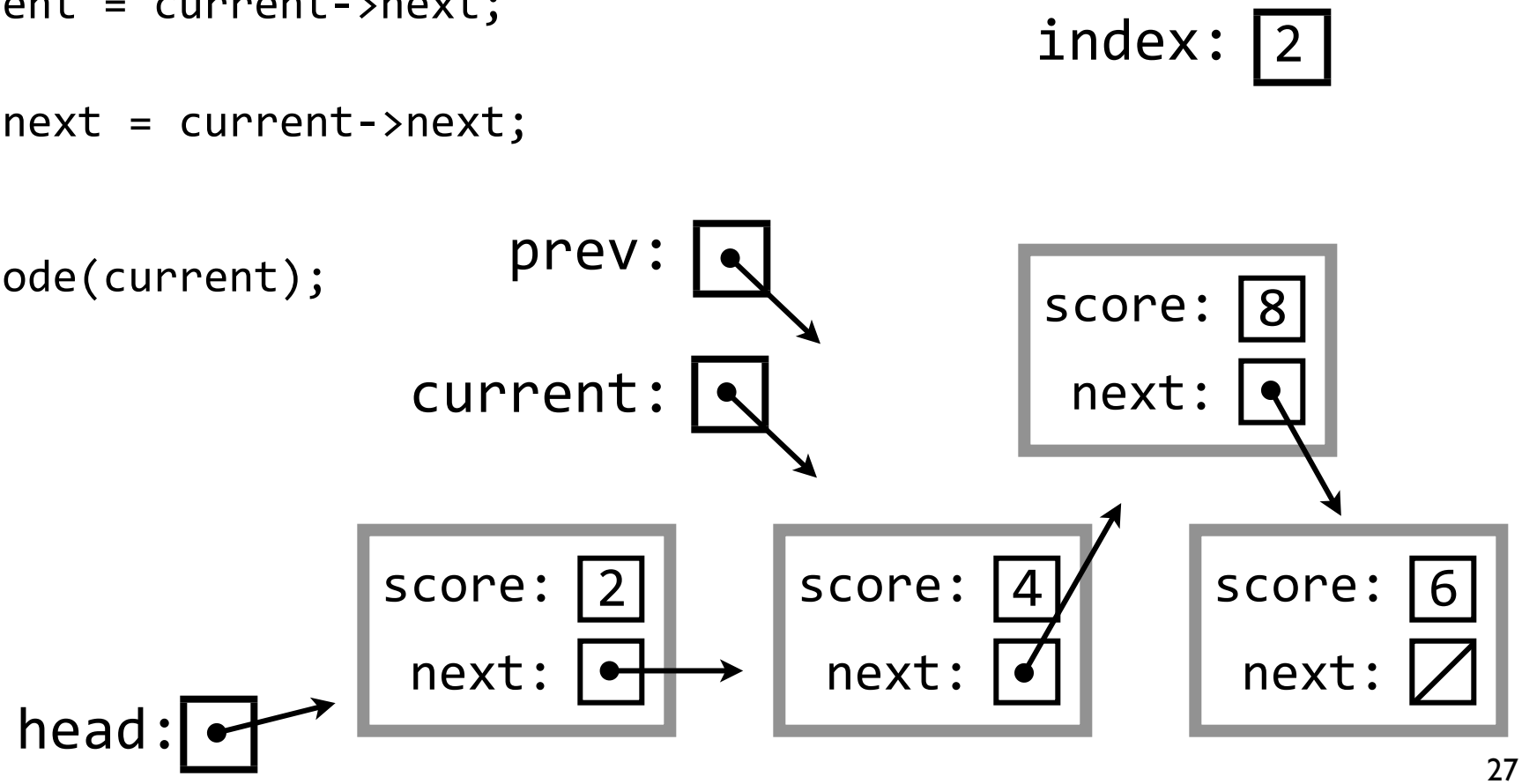
```
Node *current = head;
Node *prev = NULL;

if (index == 0)
{
  head = head->next;
}
else
{

  for (int i = 0; i < index && current != NULL; i++)
  {
    prev = current;
    current = current->next;
  }

  prev->next = current->next;

}

destroyNode(current);
```



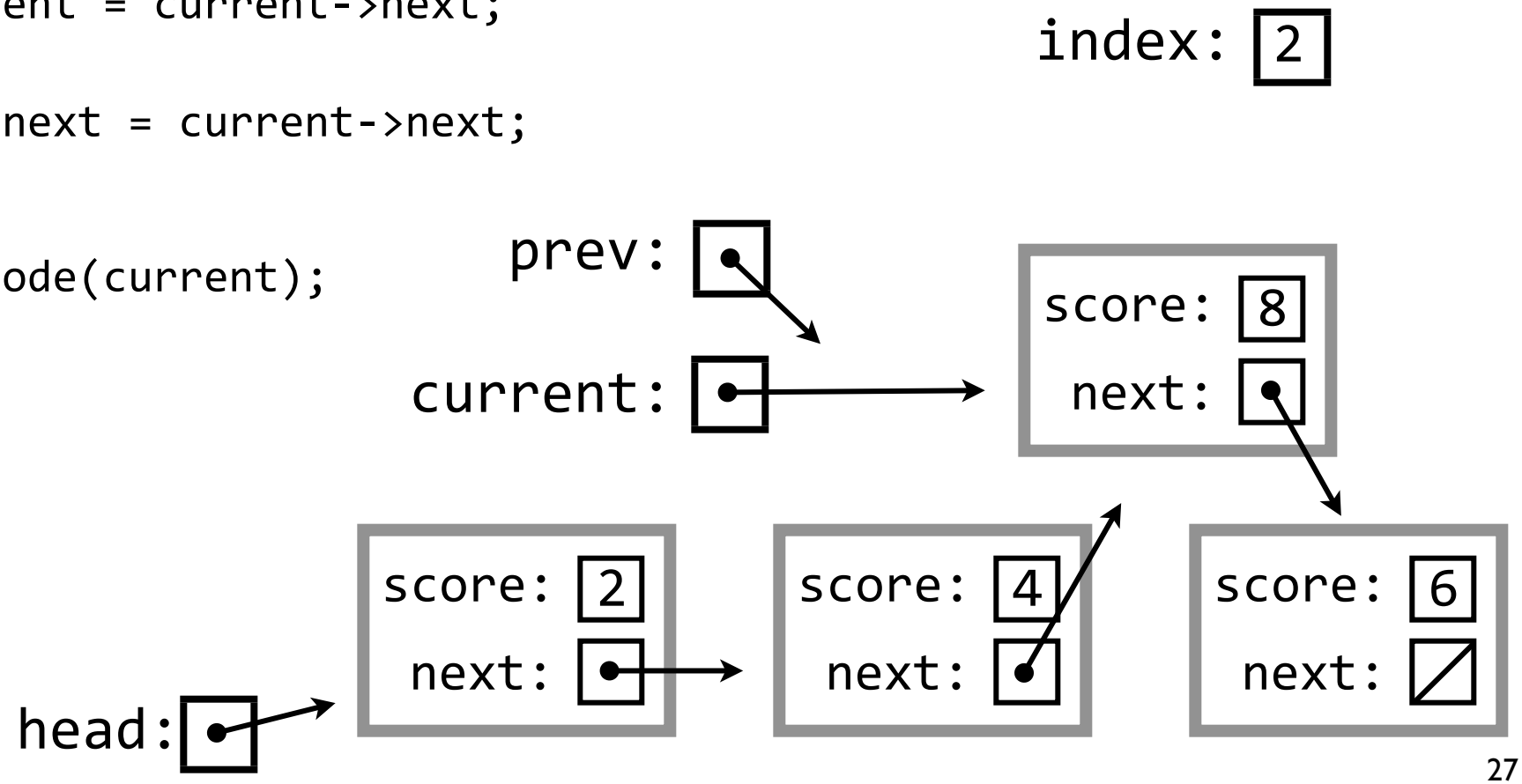
```
Node *current = head;
Node *prev = NULL;

if (index == 0)
{
  head = head->next;
}
else
{

  for (int i = 0; i < index && current != NULL; i++)
  {
    prev = current;
    current = current->next;
  }

  prev->next = current->next;

}

destroyNode(current);
```



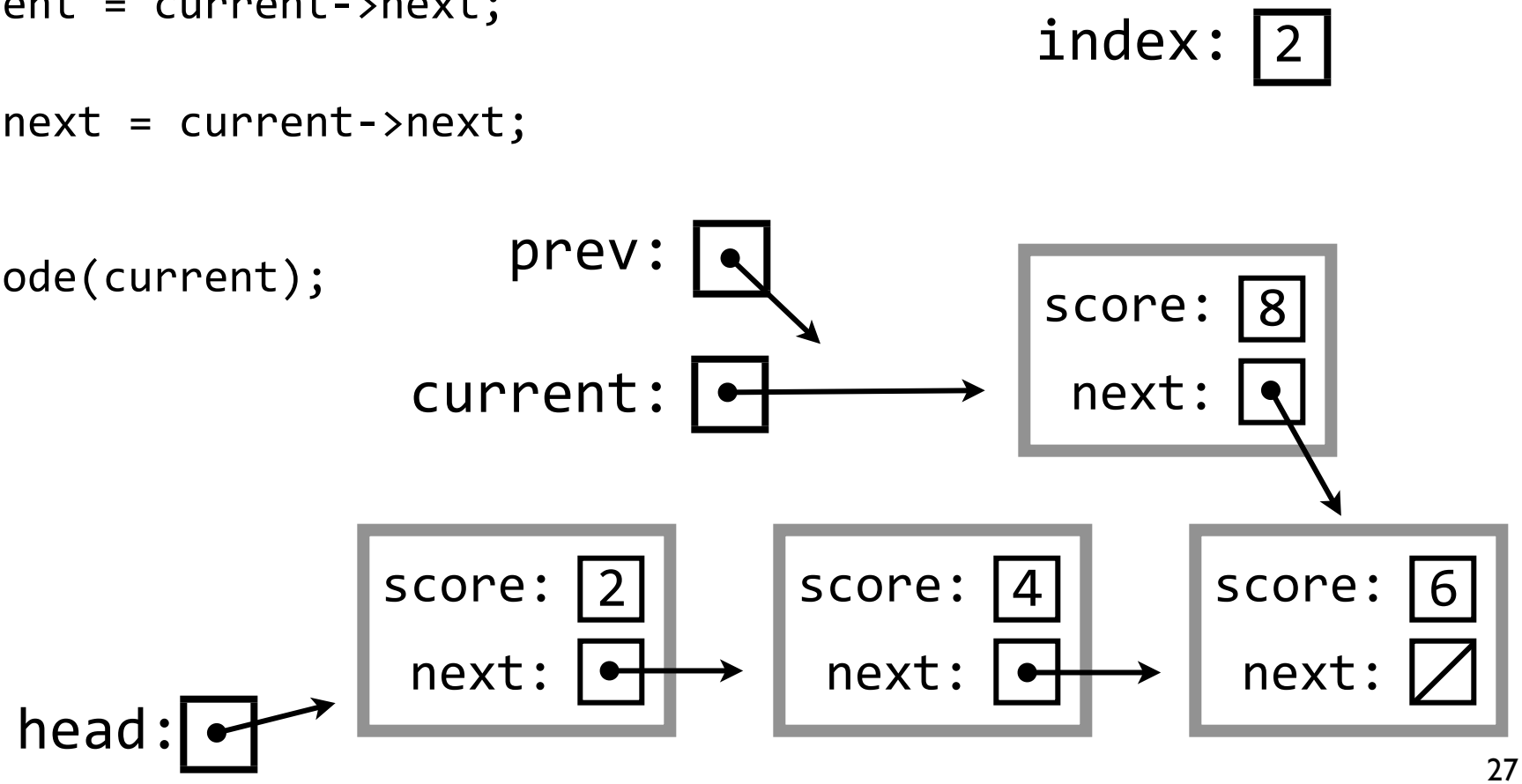
```
Node *current = head;
Node *prev = NULL;

if (index == 0)
{
  head = head->next;
}
else
{

  for (int i = 0; i < index && current != NULL; i++)
  {
    prev = current;
    current = current->next;
  }

  prev->next = current->next;

}

destroyNode(current);
```



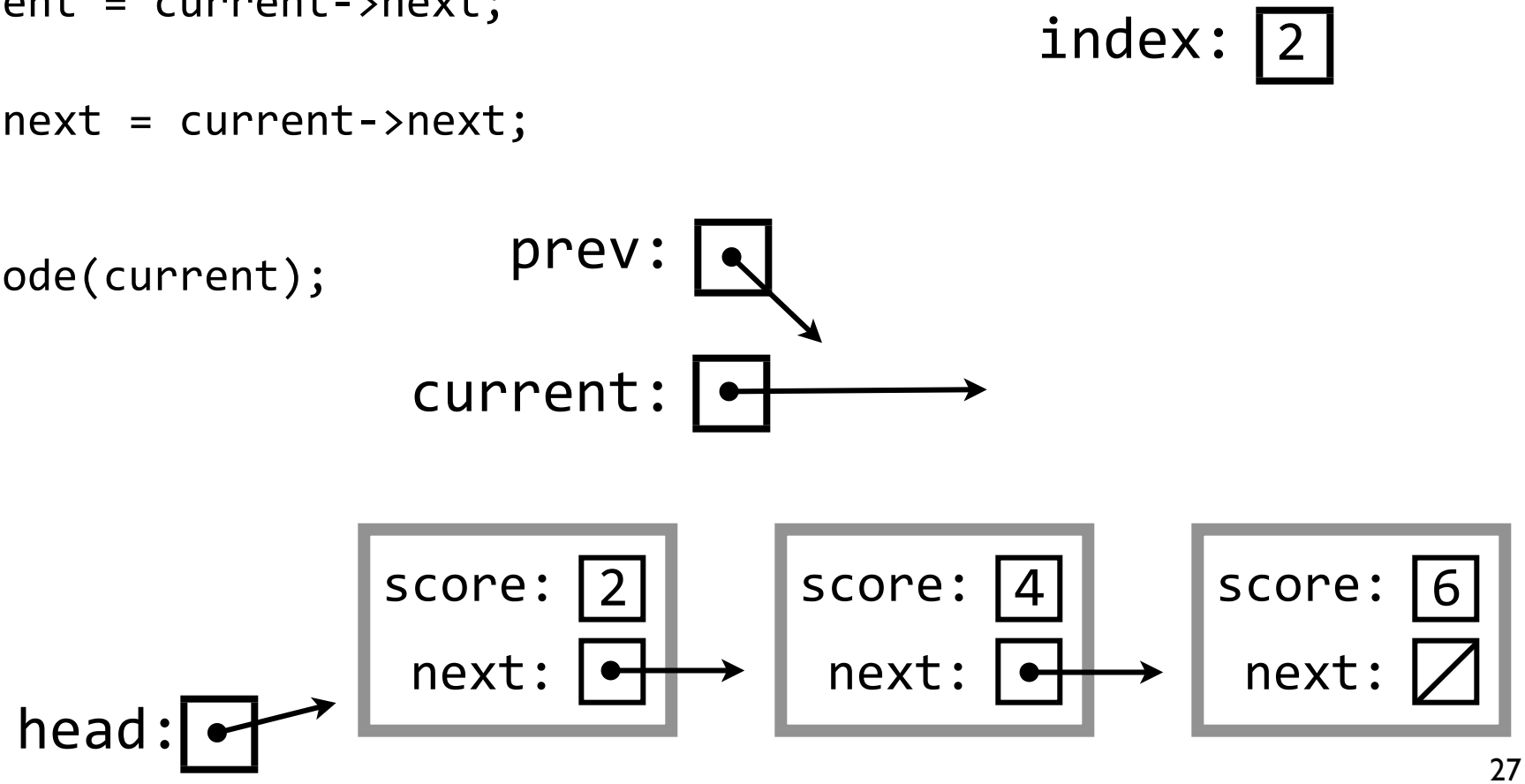
```
Node *current = head;
Node *prev = NULL;

if (index == 0)
{
  head = head->next;
}
else
{

  for (int i = 0; i < index && current != NULL; i++)
  {
    prev = current;
    current = current->next;
  }

  prev->next = current->next;

}

destroyNode(current);
```



```
Node *current = head;
Node *prev = NULL;

if (index == 0)
{
  head = head->next;
}
else
{

  for (int i = 0; i < index && current != NULL; i++)
  {
    prev = current;
    current = current->next;
  }

  prev->next = current->next;

}

destroyNode(current);
```



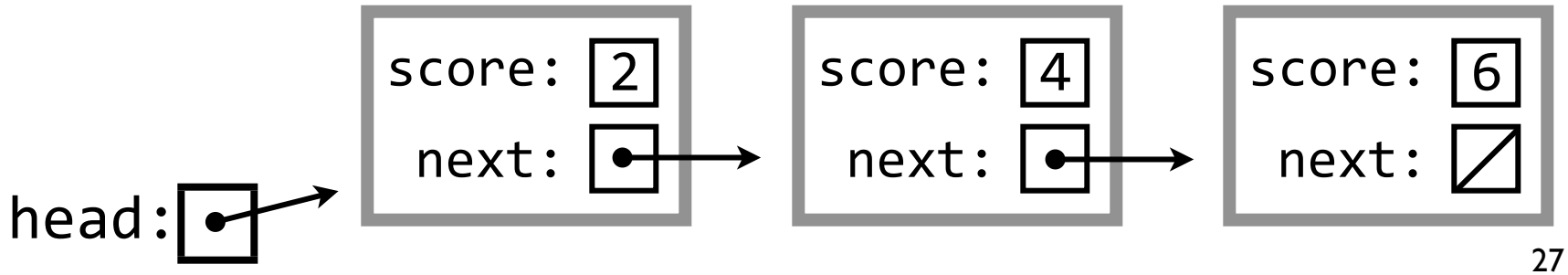
```
Node *current = head;
Node *prev = NULL;

if (index == 0)
{
  head = head->next;
}
else
{

  for (int i = 0; i < index && current != NULL; i++)
  {
    prev = current;
    current = current->next;
  }

  prev->next = current->next;

}

destroyNode(current);
```



```
Node *current = head;
Node *prev = NULL;

if (index == 0)
{
  head = head->next;
}
else
{

  for (int i = 0; i < index && current != NULL; i++)
  {
    prev = current;
    current = current->next;
  }

  prev->next = current->next;

}

destroyNode(current);
```

This doesn't work if index > length of list

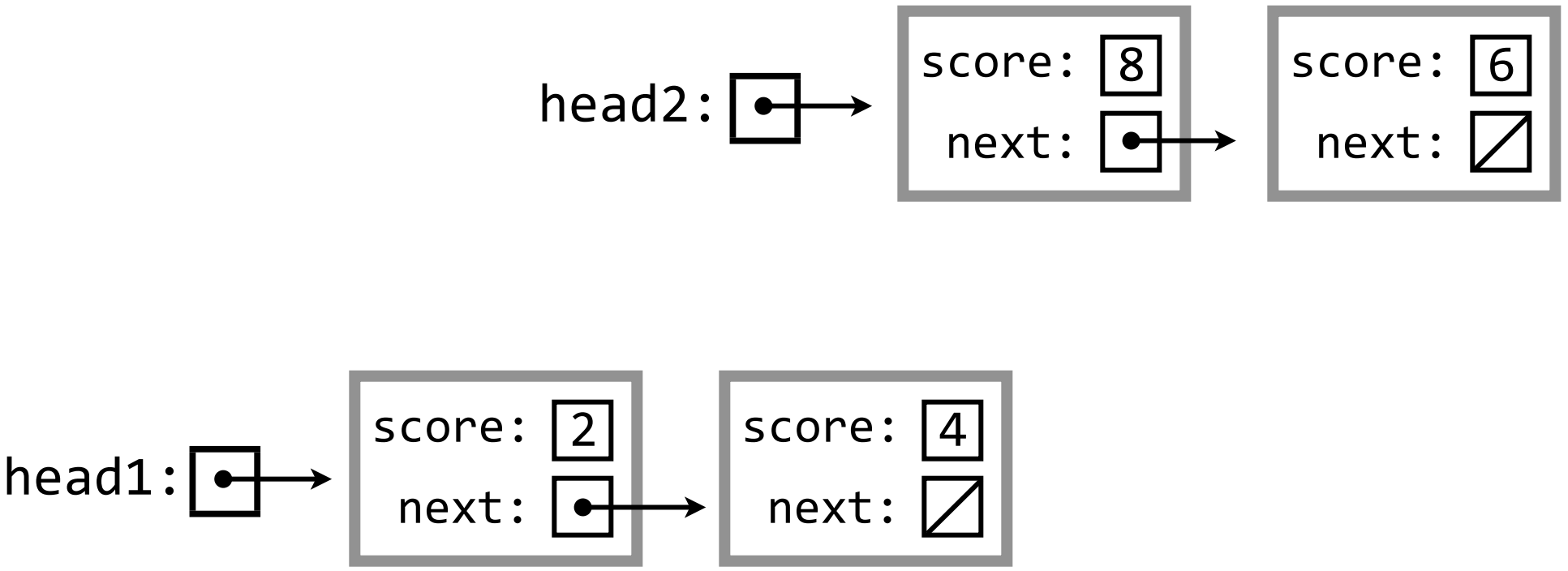
index: 2



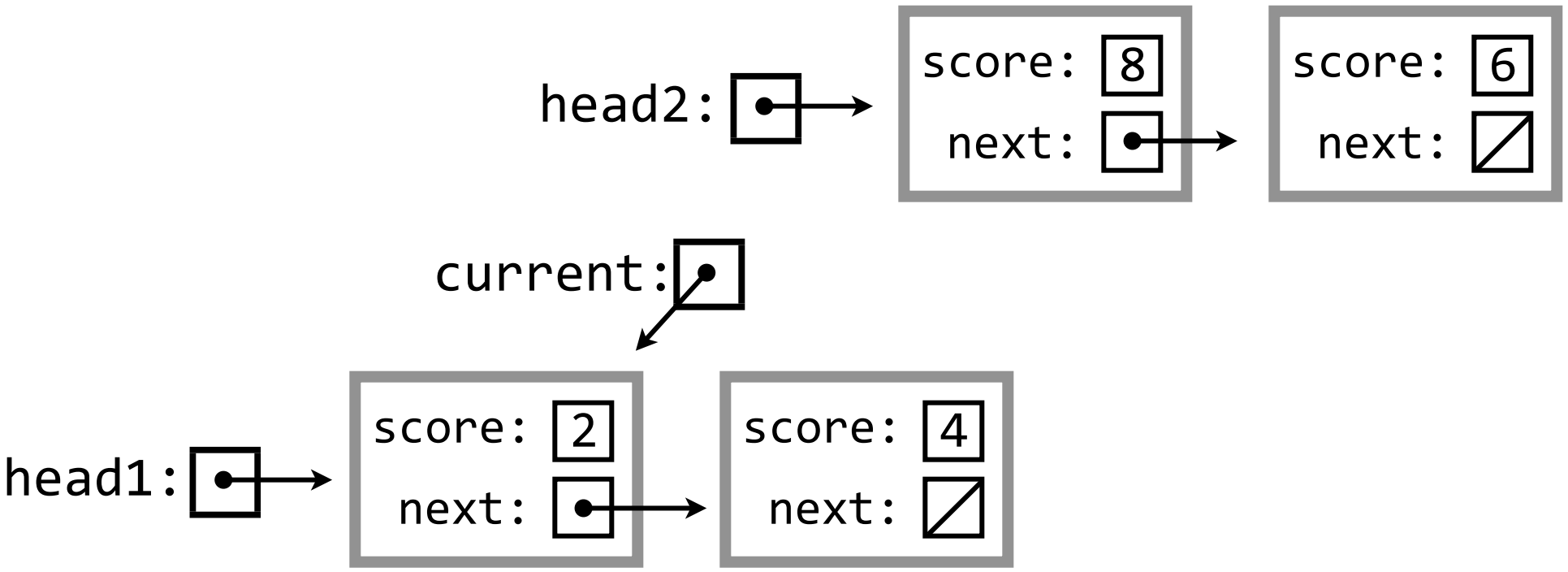
Verdict?

- Somewhere in between
- In some cases arrays could be better
- In general, linked lists are better

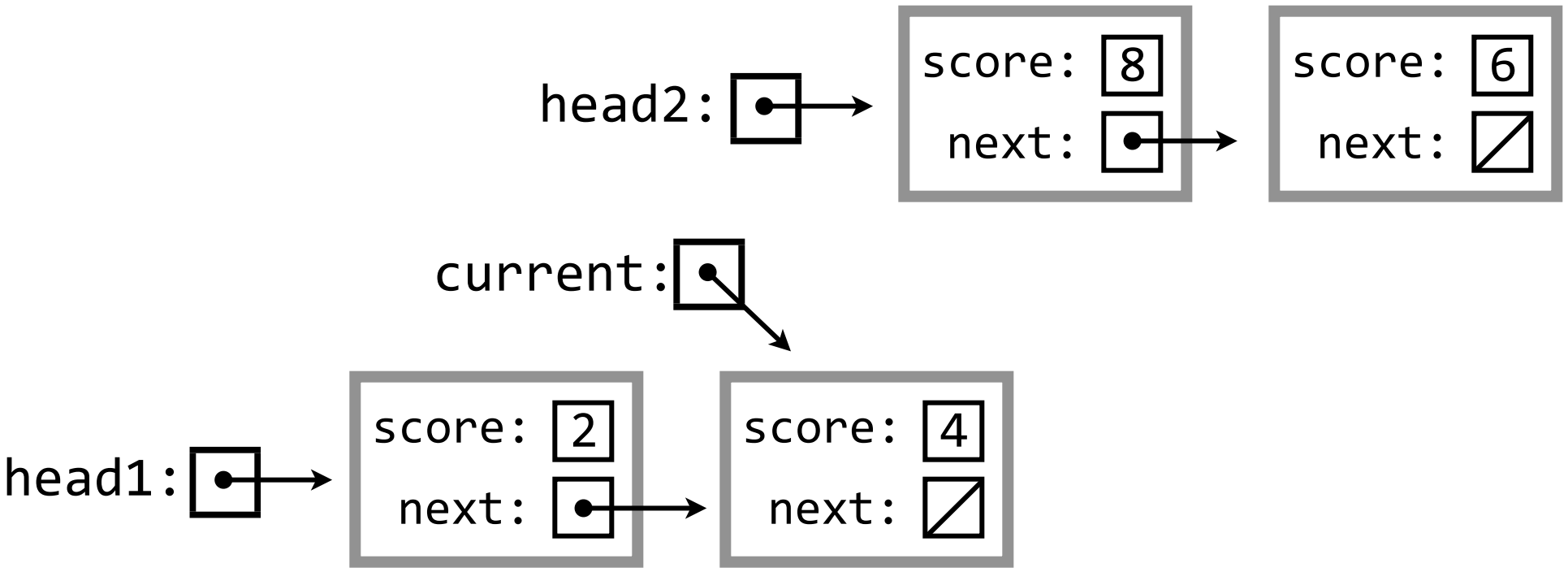
Concatenate Two Lists



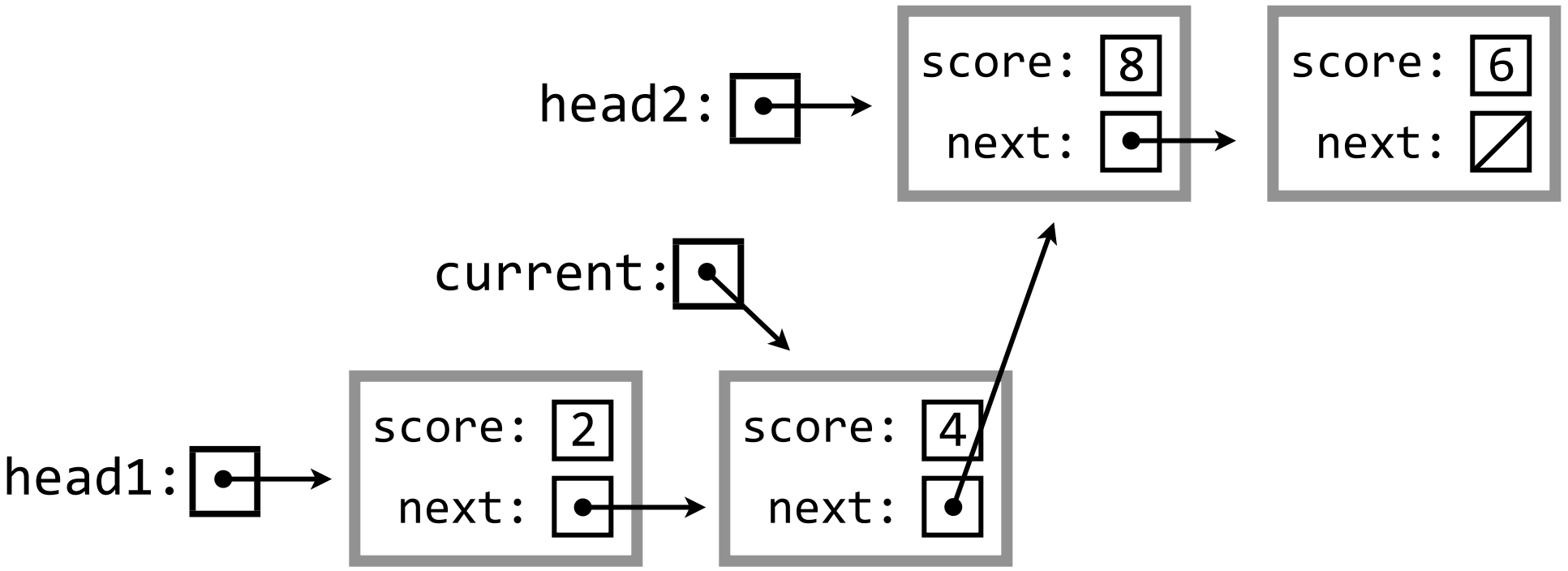
Concatenate Two Lists



Concatenate Two Lists



Concatenate Two Lists

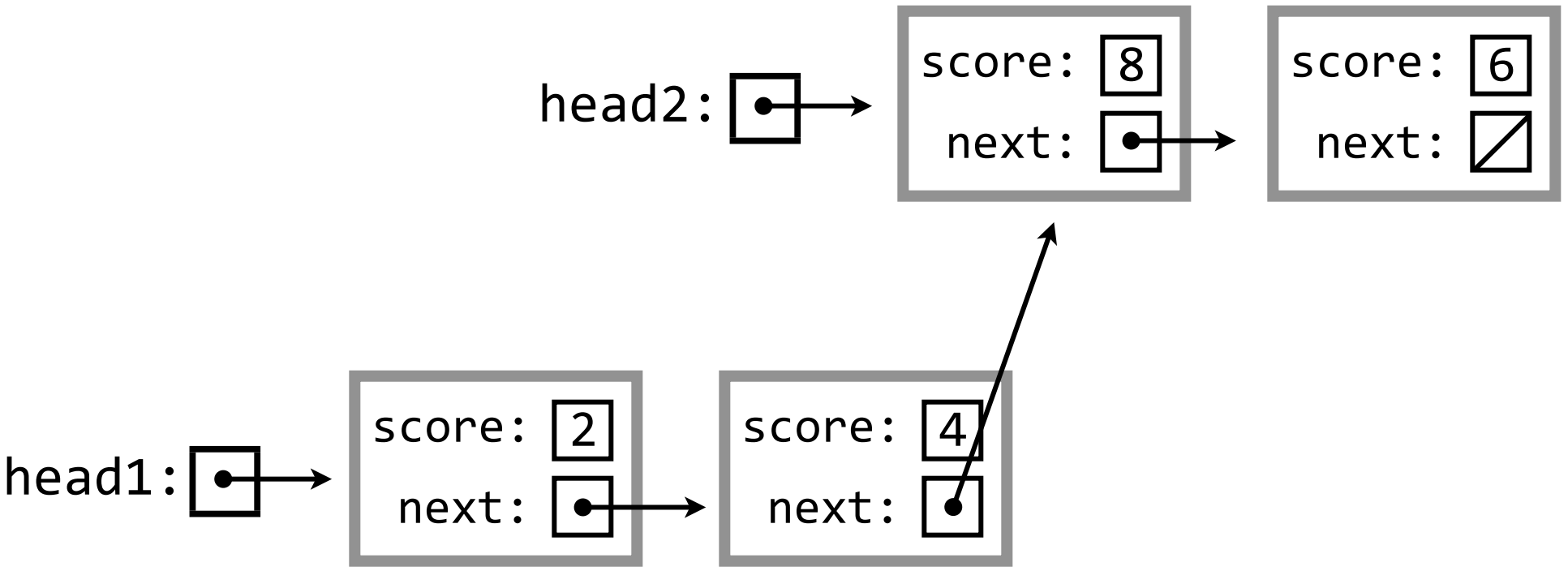


Concatenate Two Lists

head2 is still a valid pointer

head2 is still a linked list

These facts can be useful, dangerous, or irrelevant



Recursively Processing Lists

To Process a List:

If head is NULL,

we're done

Otherwise:

Process the first node

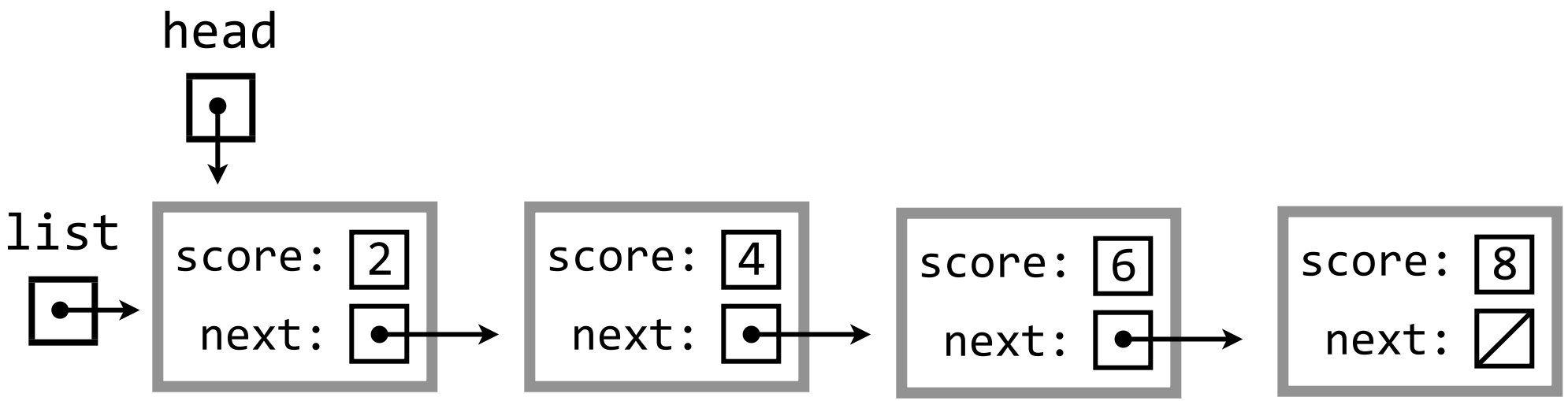
Process the remainder of the list

```
void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

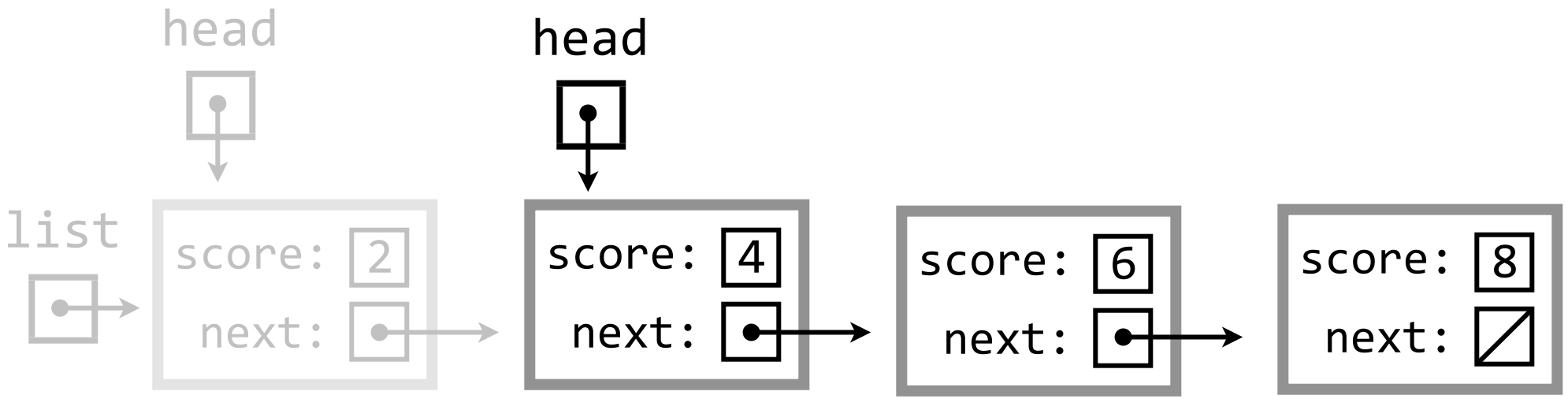
2



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

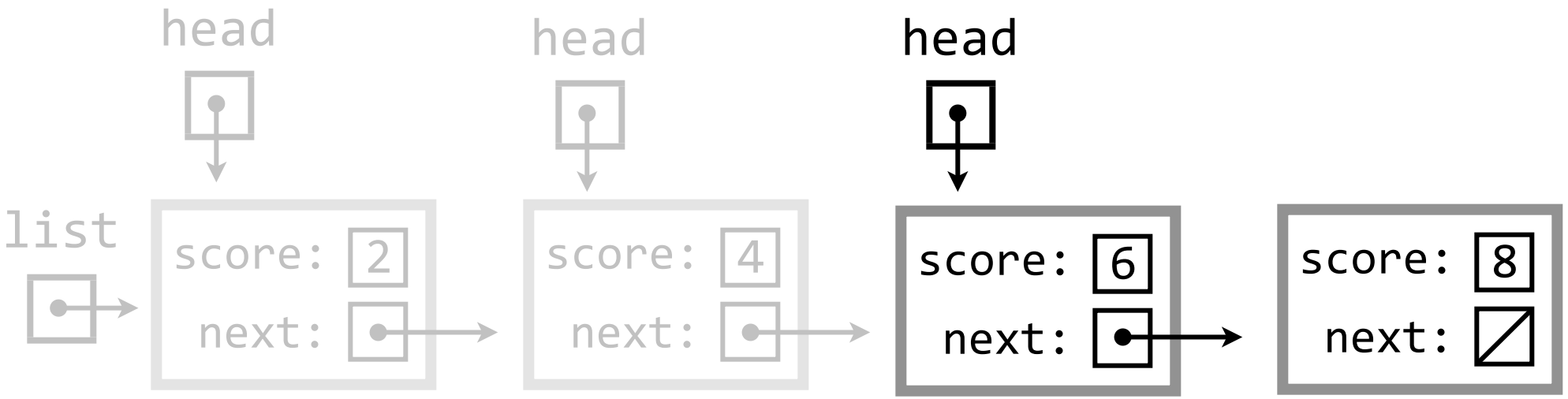
2
4



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

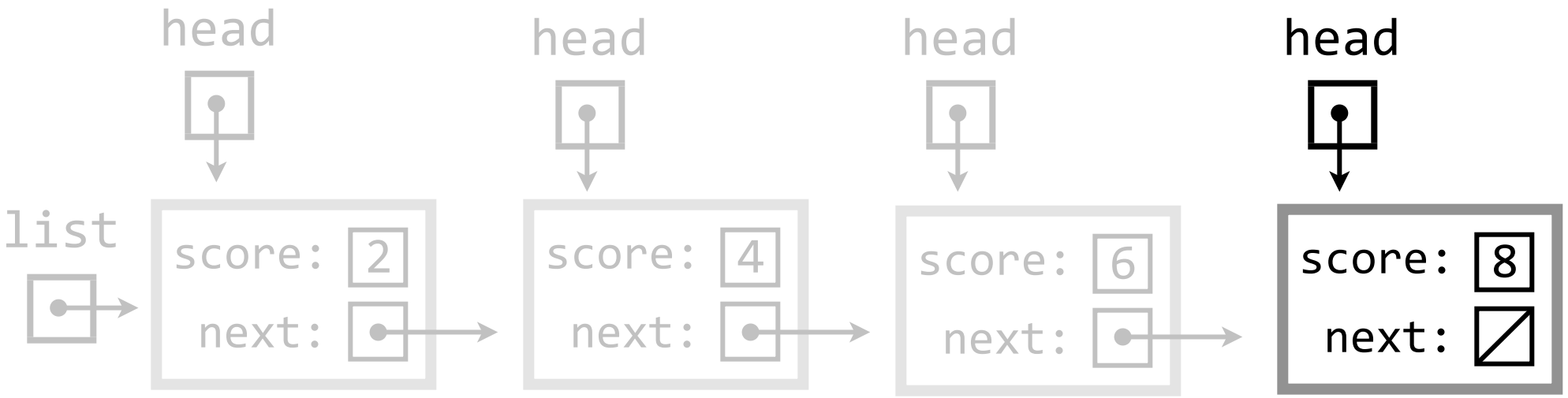
2
4
6



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

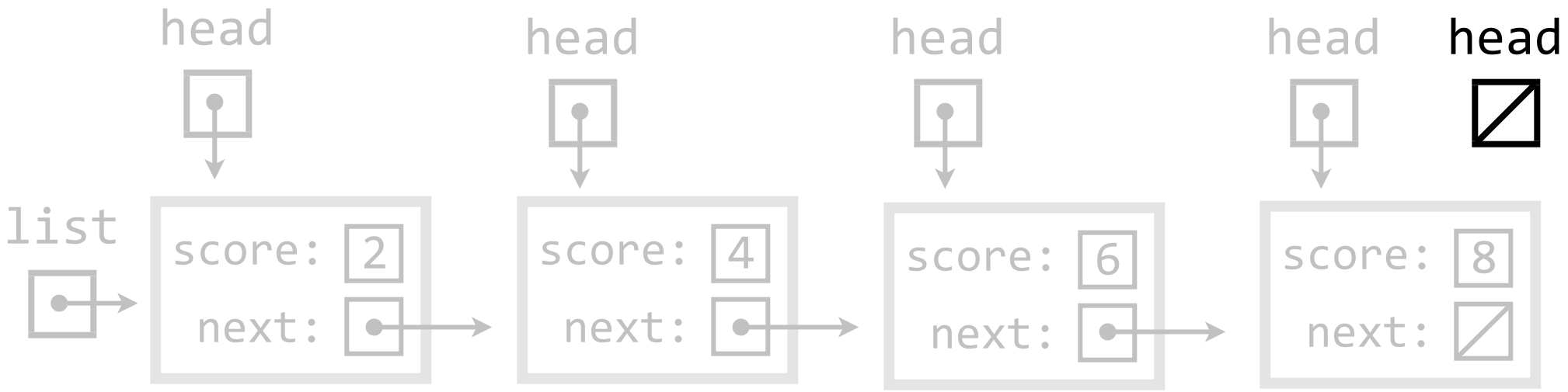
2
4
6
8



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

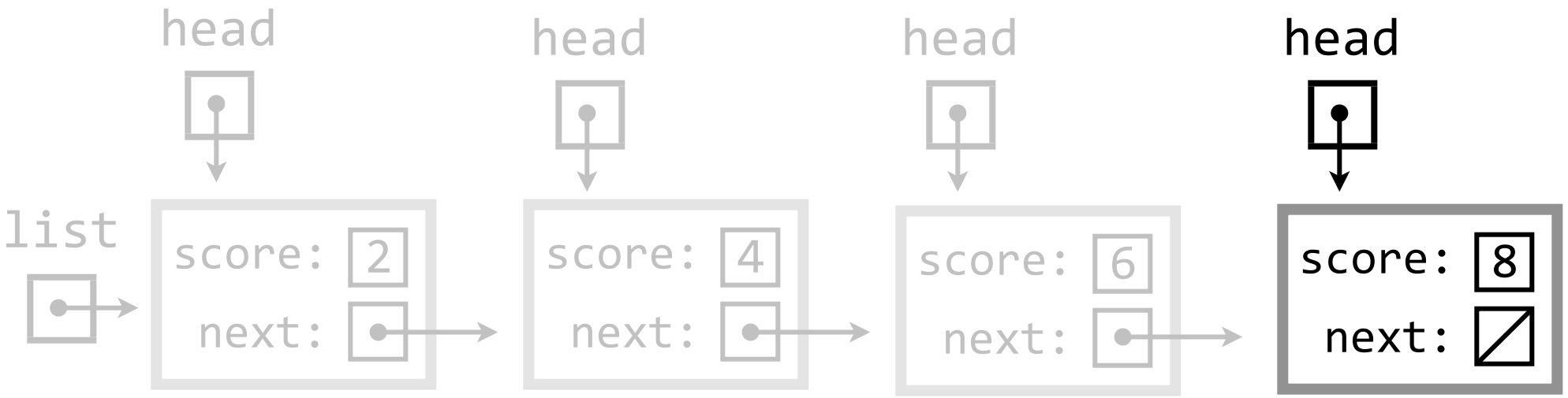
2
4
6
8



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

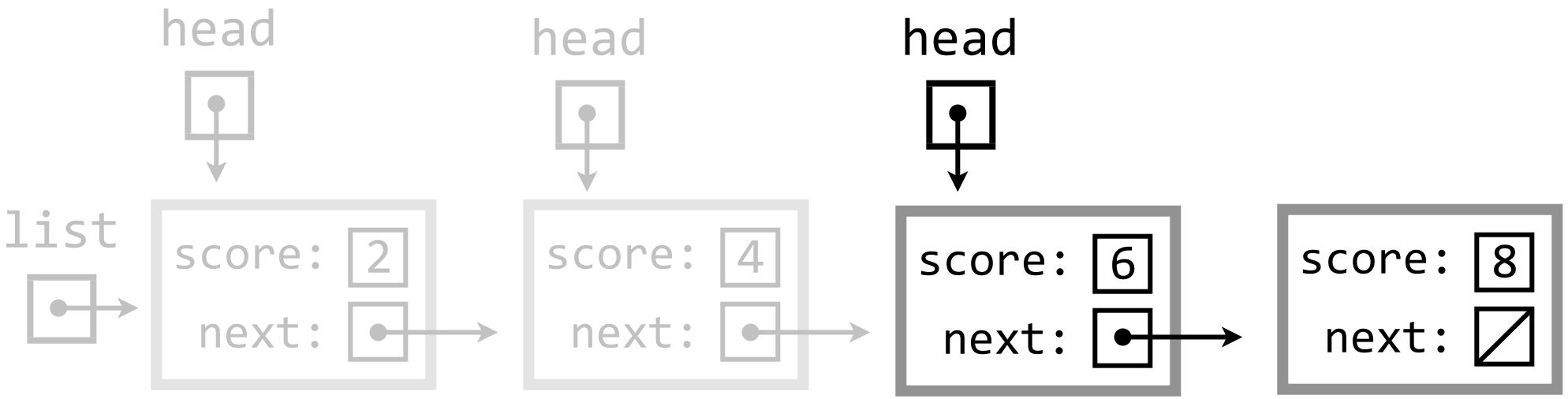
2
4
6
8



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

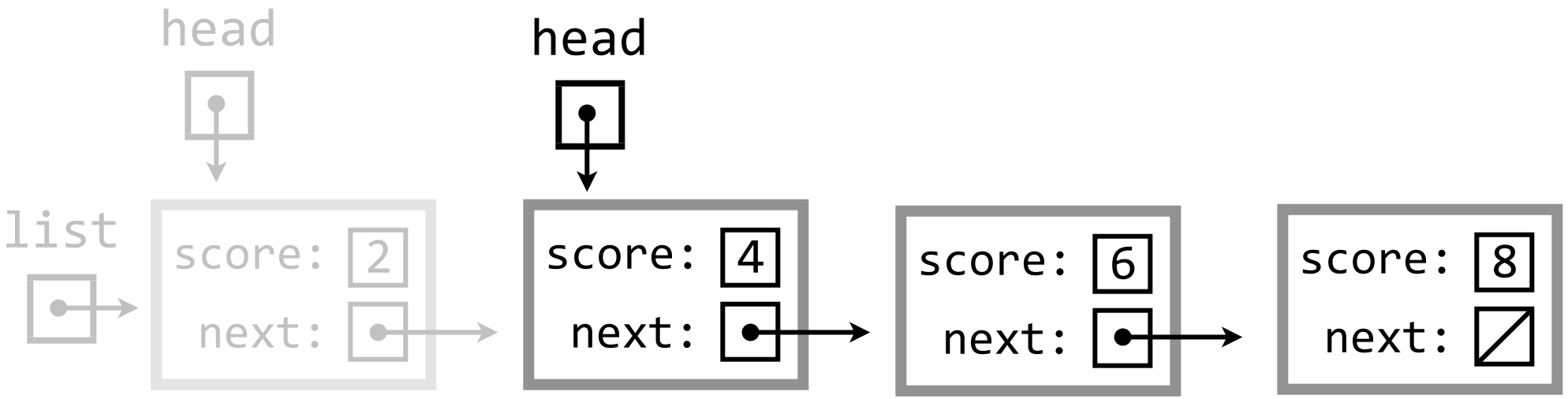
2
4
6
8



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

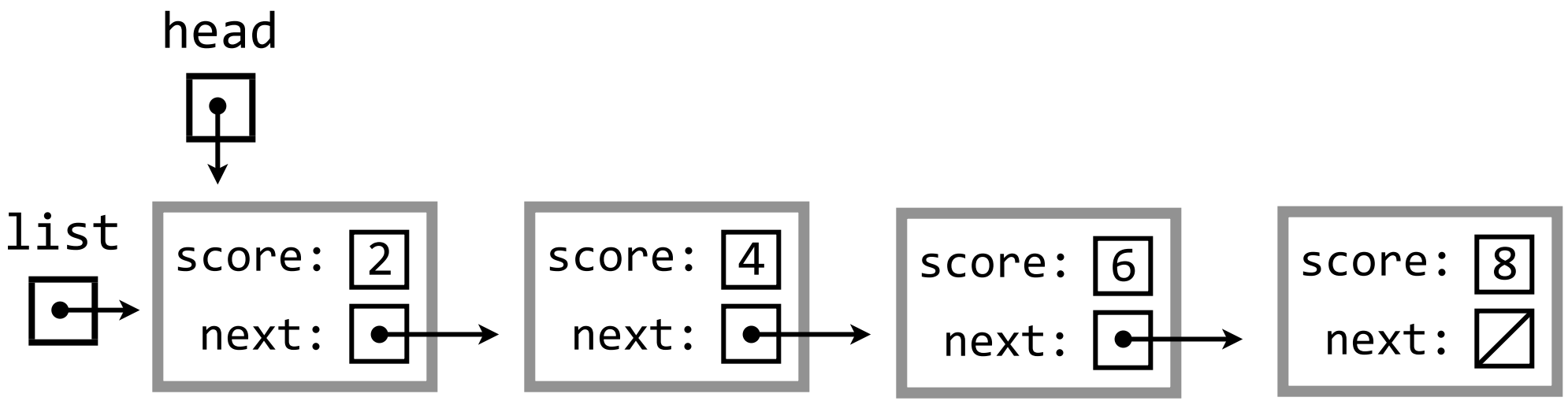
2
4
6
8



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

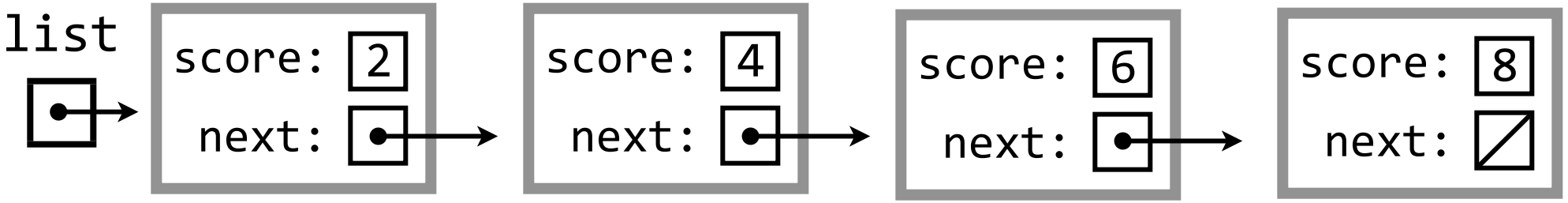
2
4
6
8



Recursively Processing Lists

```
printList(list);


void printList(Node *head)
{
    if (head != NULL)
    {
        printf("%d\n", head->score);
        printList(head->next);
    }
}
```

2
4
6
8

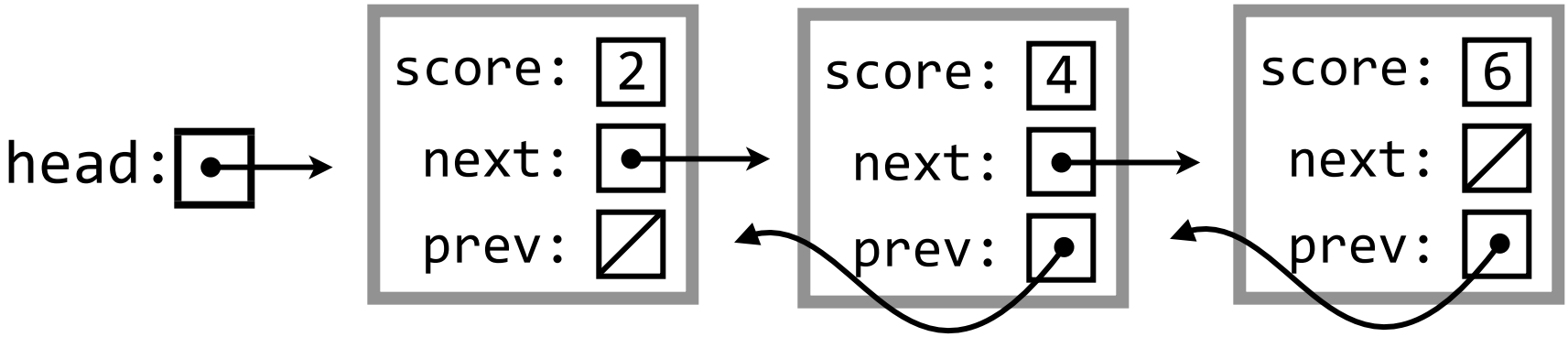


Linked List Variations

- Doubly linked lists
- Branched linked lists (i.e., trees)

(You are not being tested on these)

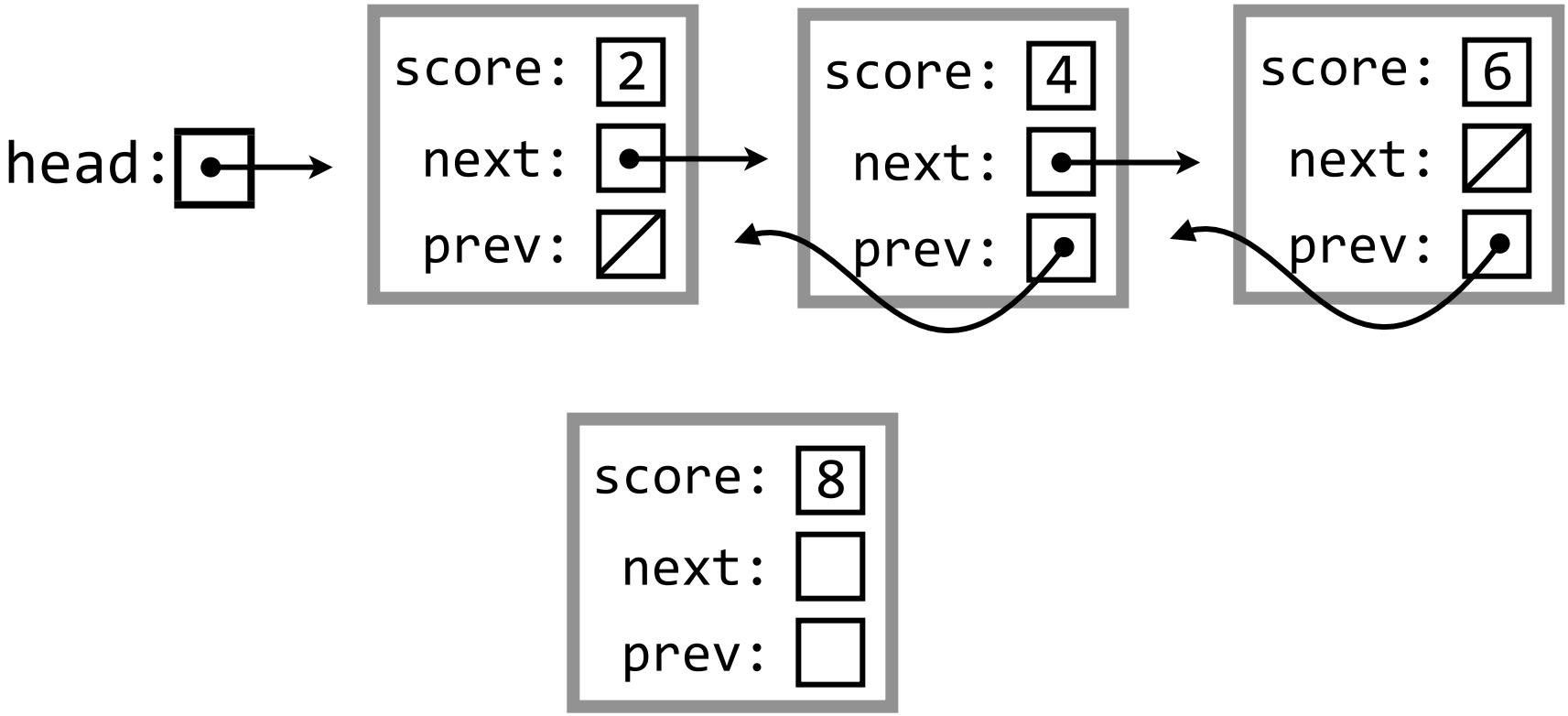
Doubly Linked Lists



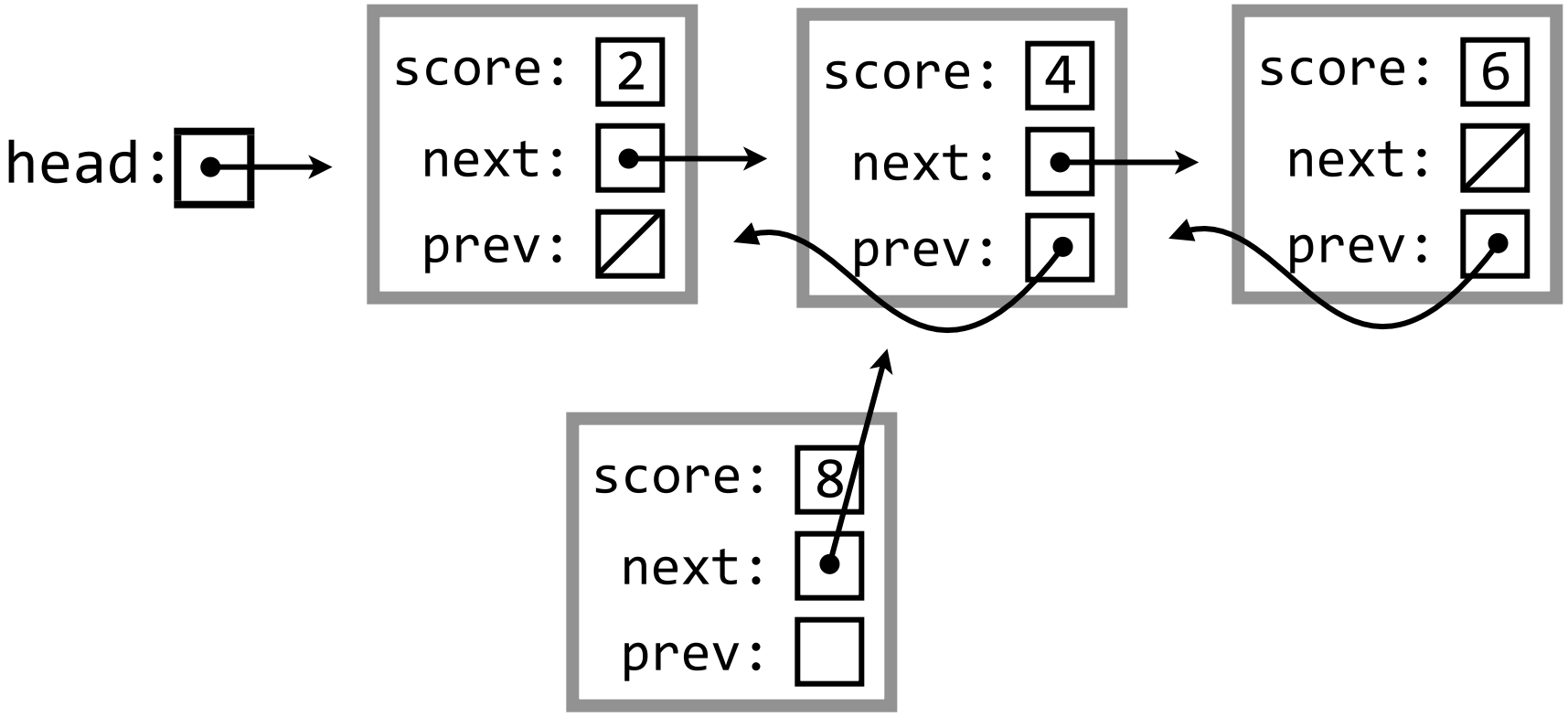
Doubly Linked Lists

- Pointers go in both directions
  - We can go forward and backward
  - There are two NULL pointers
- This seems useful, why not use them all the time?
  - More complexity

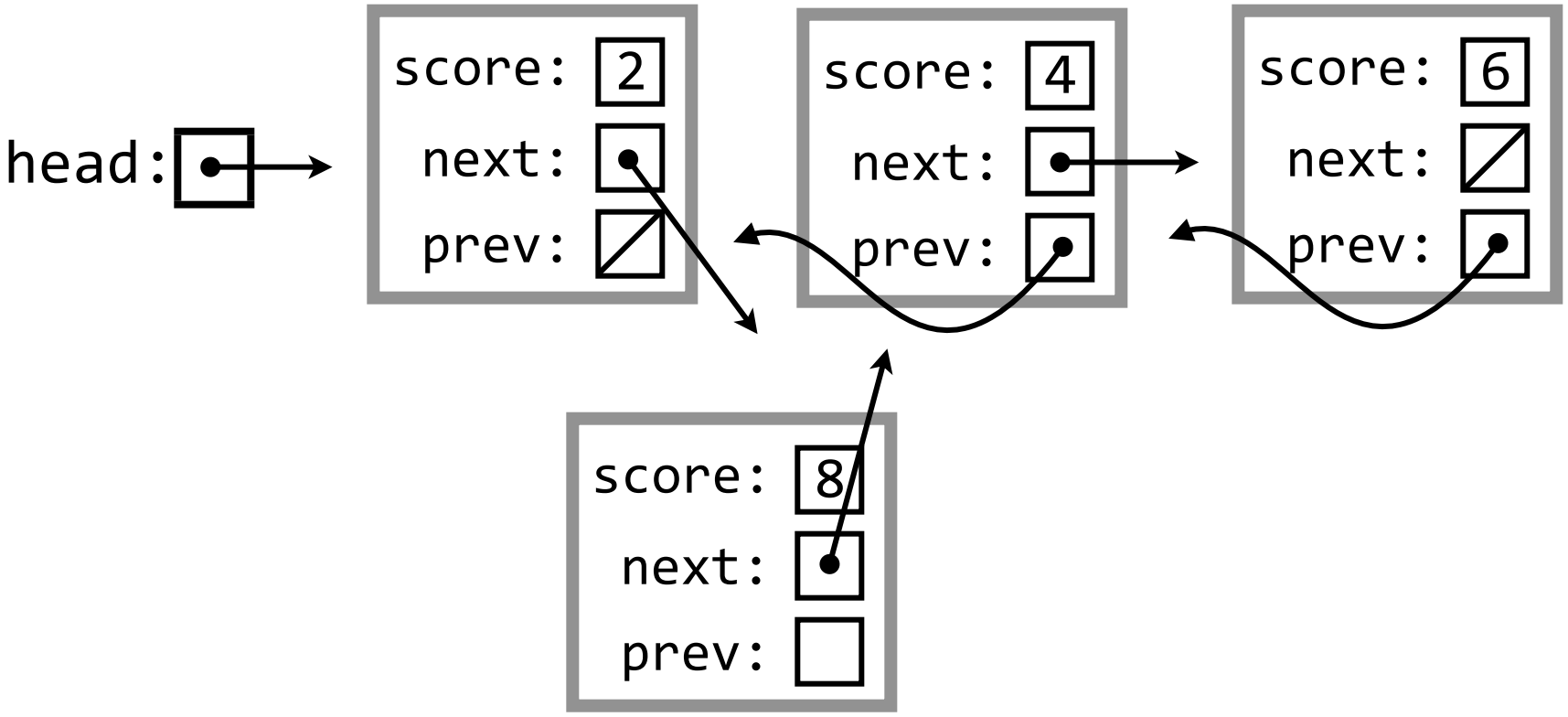
Doubly Linked Lists



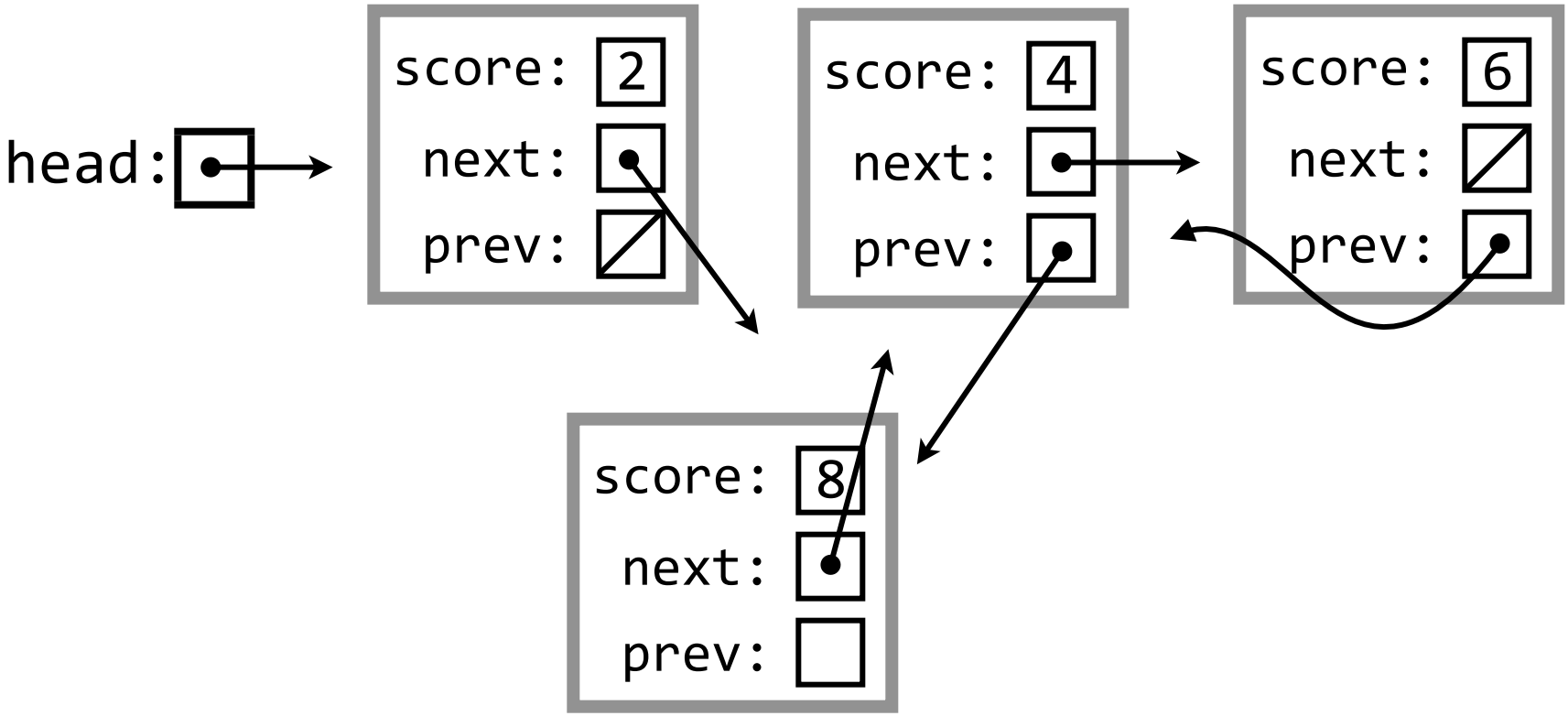
Doubly Linked Lists



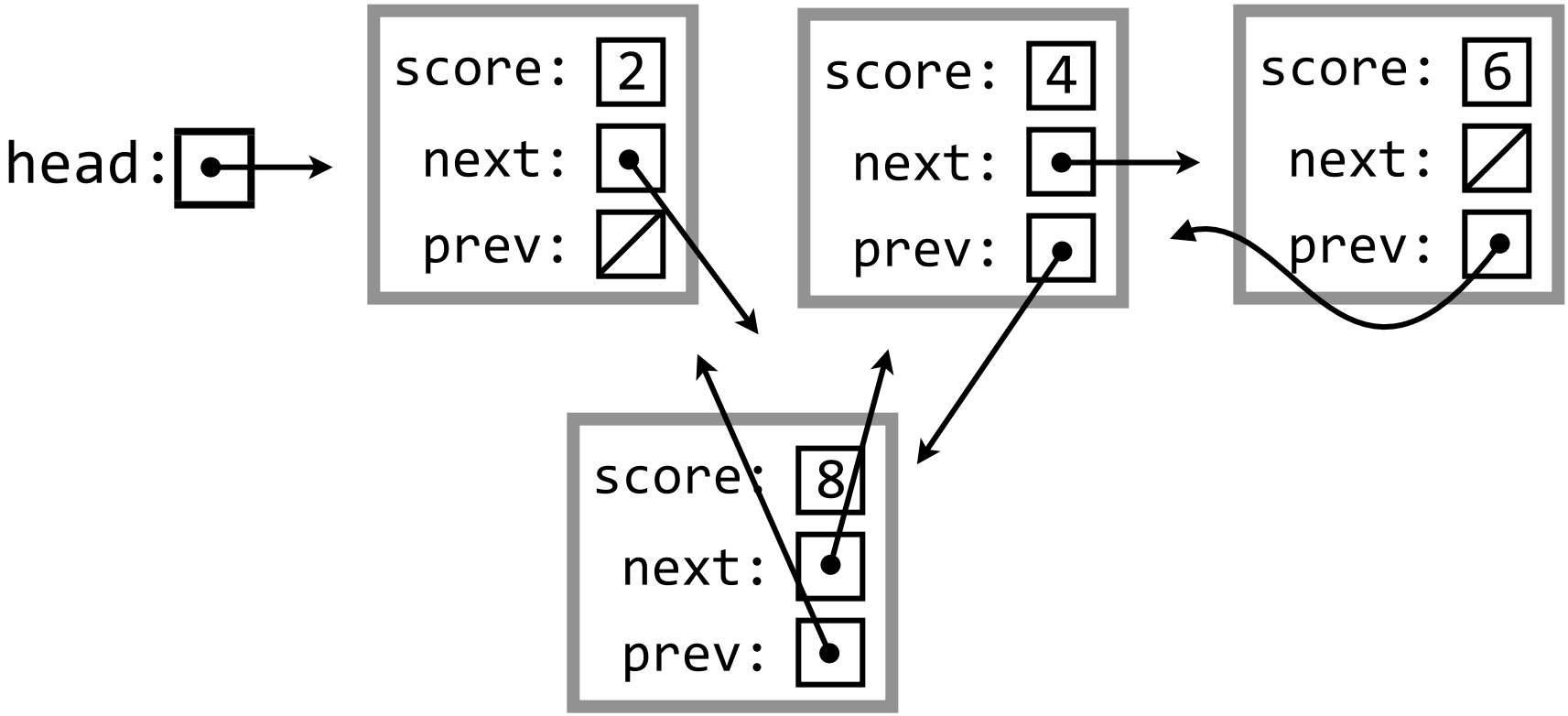
Doubly Linked Lists



Doubly Linked Lists



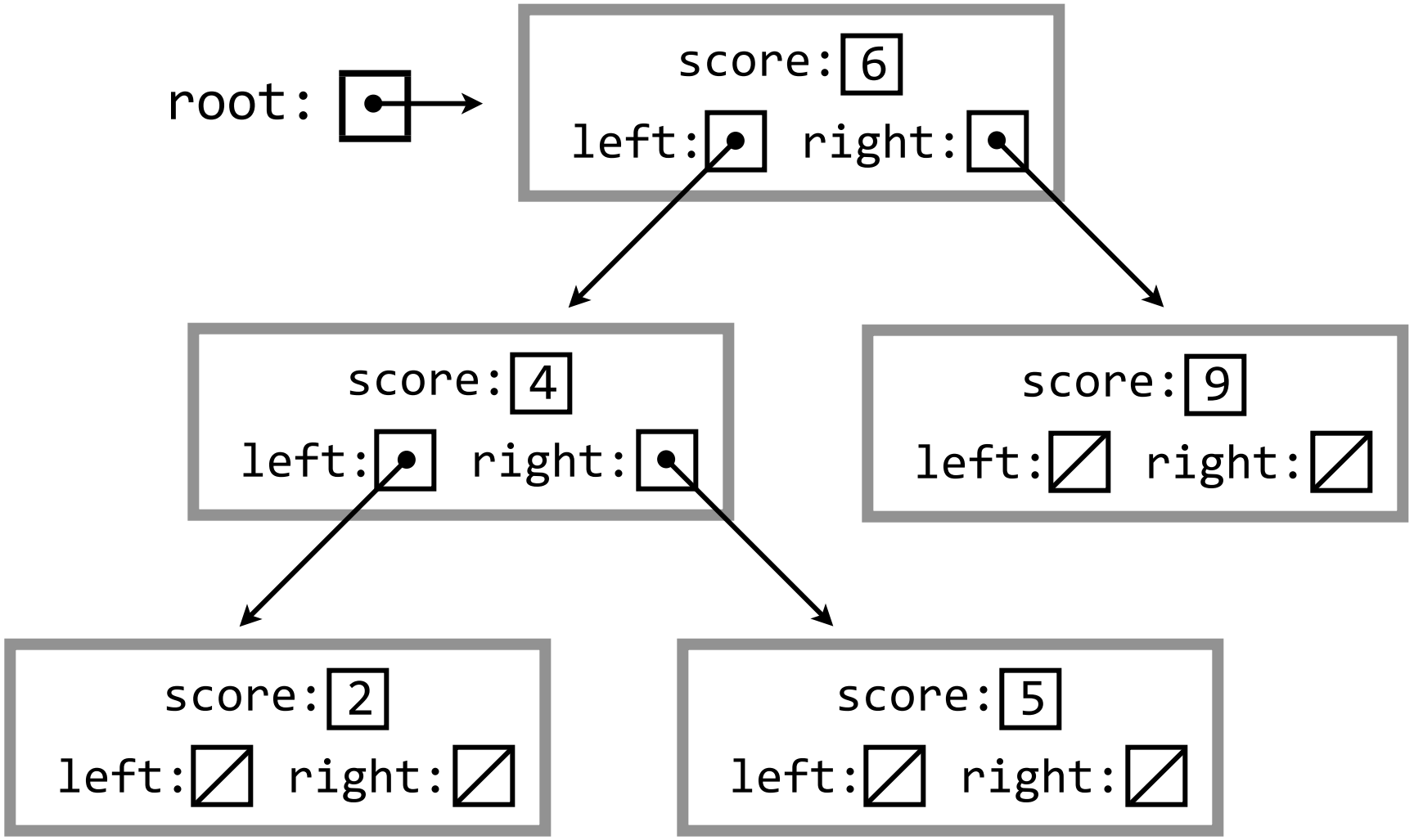
Doubly Linked Lists



Doubly Linked Lists

- Pointers go in both directions
  - We can go forward and backward
  - There are two NULL pointers
- This seems useful, why not use them all the time?
  - More complexity
  - More space (almost never an issue these days)

Trees


root:
score: 6
left:
right:
score: 4
left:
right:
score: 9
left:
right:
score: 2
left:
right:
score: 5
left:
right:

Linked List Summary

- Different approach to building a data structure (using indirection)
- Series of nodes, each has a pointer to the next node
- Arrays are better at accessing `i`th element
- Linked lists are usually better at inserting and deleting