APS105
Winter 2012

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

Lecture 29
March 28, 2012

Today

- Linked Lists

Data Structures

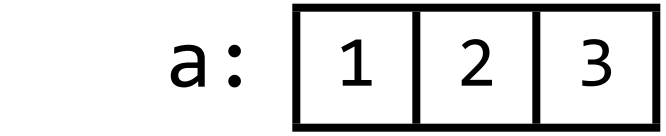
Data Structures

- How we represent and store data in our programs

- Individual variables
- Arrays
- Arrays of pointers
- `structs`

Arrays

```
int a[] = {1,2,3};
```

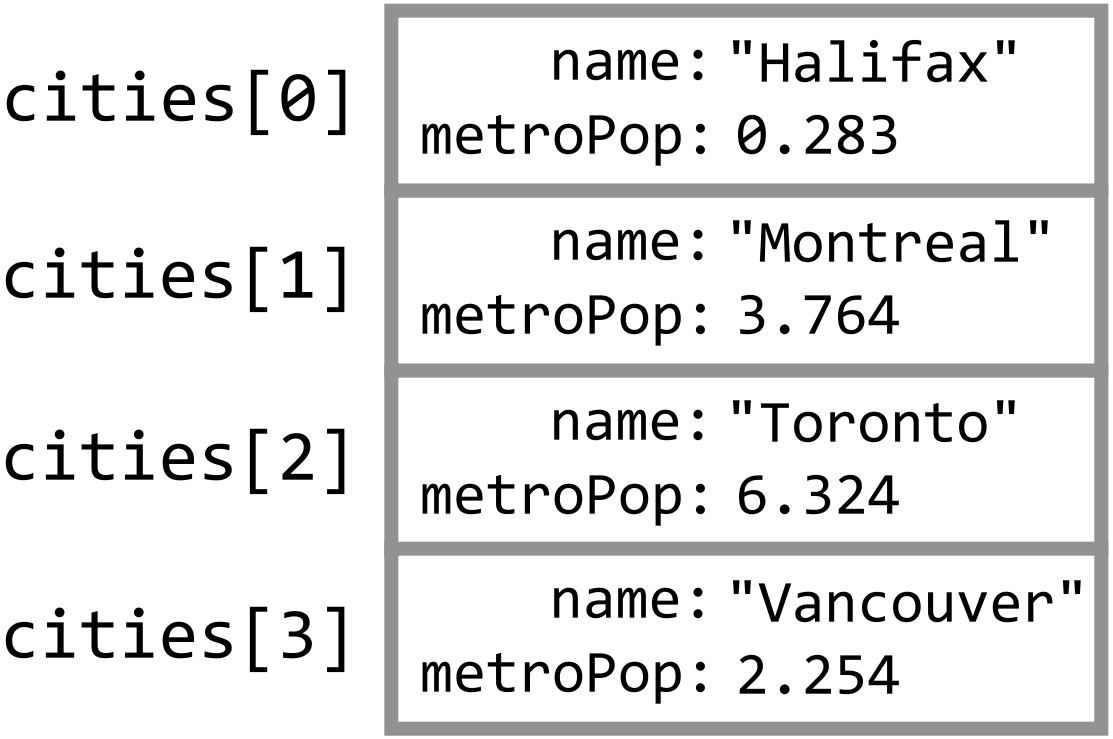


- Pro:
  - Simple
  - Easy (and fast) access to each element
- Con:
  - Only stores one type of data
  - Fixed size
  - Needs to be contiguous

Arrays of `structs`

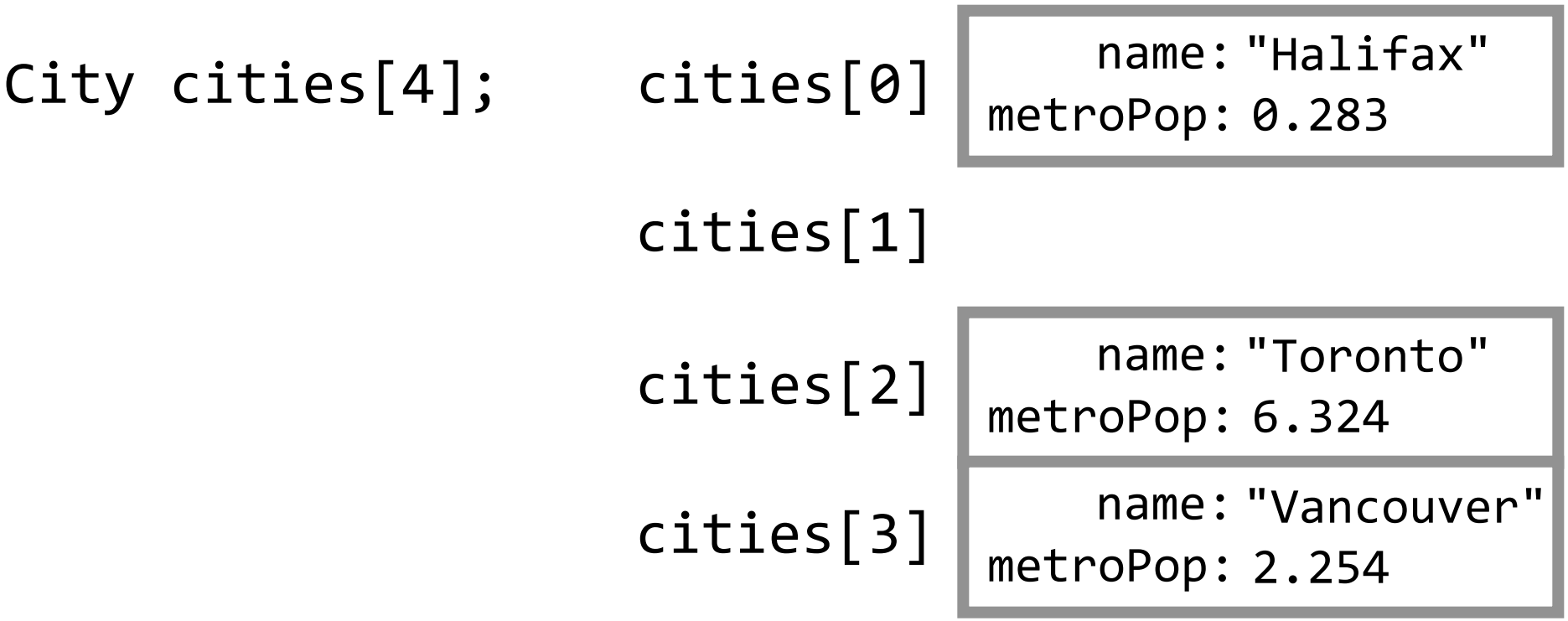
- `structs` let us group multiple variables together
- Arrays let us group multiple `structs` together
- Fixes the "single type" problem

```
City cities[4];
```



Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements



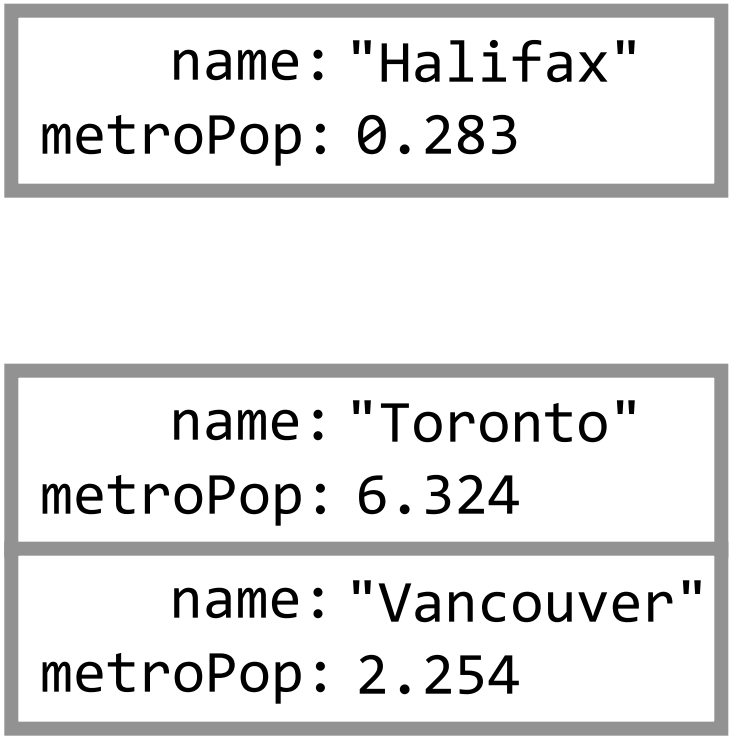
Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];
```

```
cities[1] = NULL;
```

Error



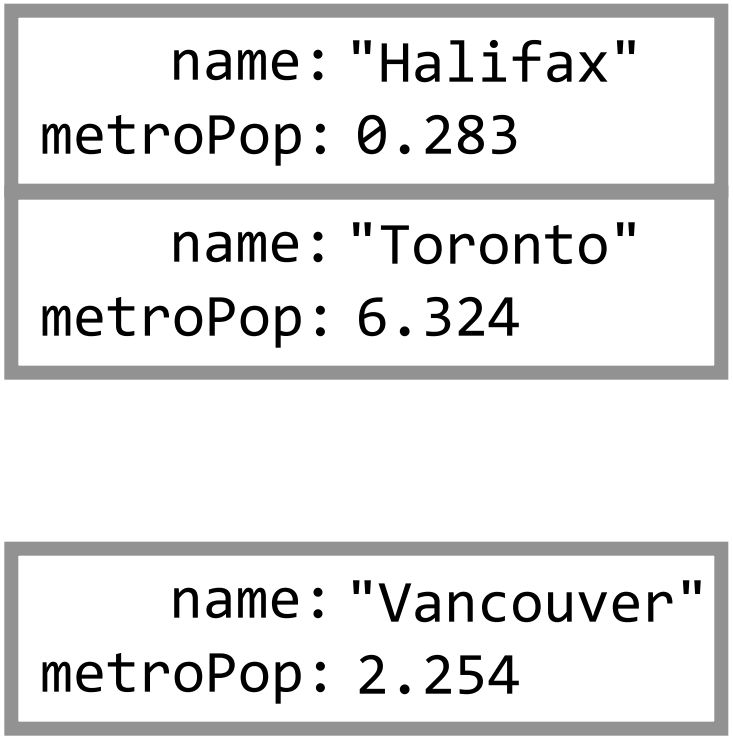
Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];
```

```
cities[1] = NULL;
```

Error

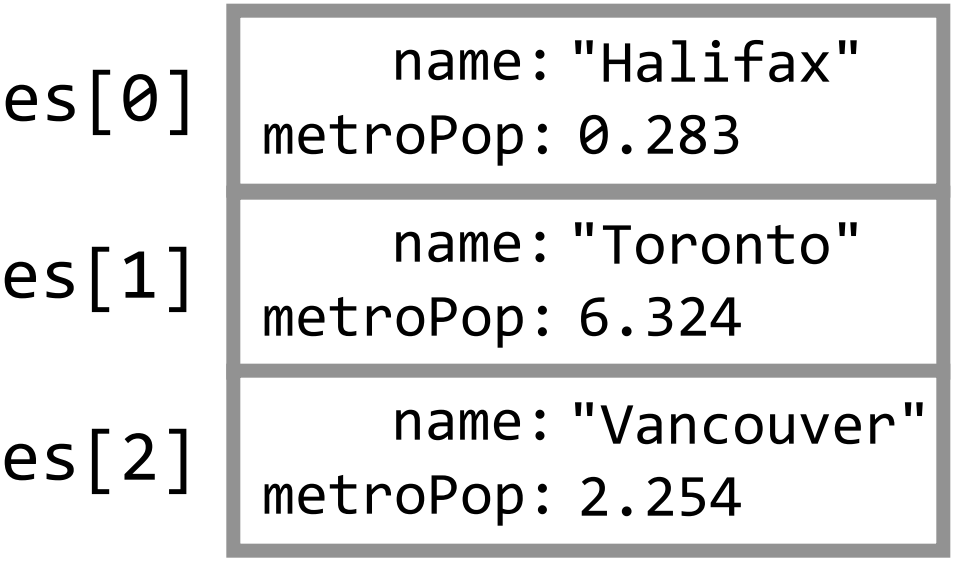


Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];

cities[1] = NULL;
```

Error



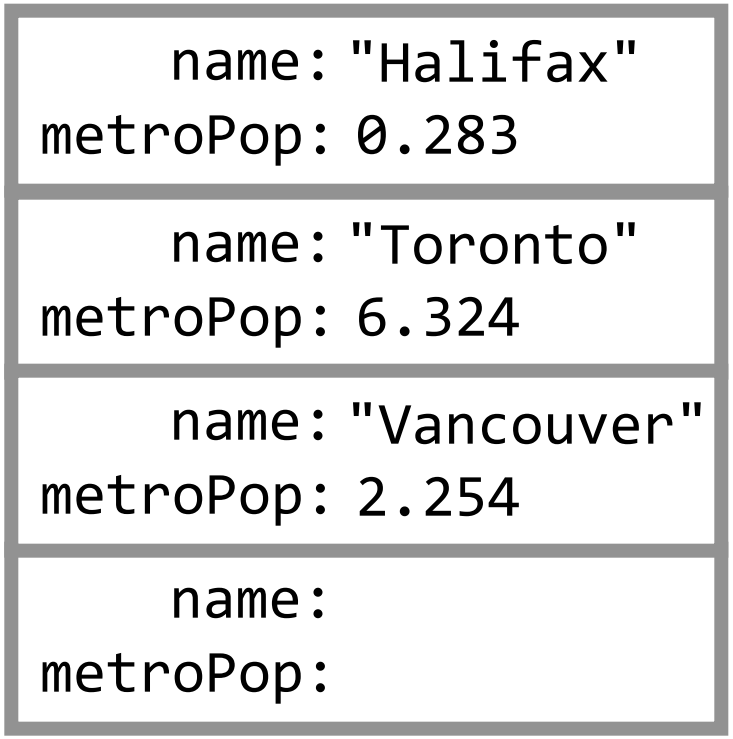
Arrays of `structs`

- Still need to be contiguous
- Hard to delete/move elements

```
City cities[4];
```

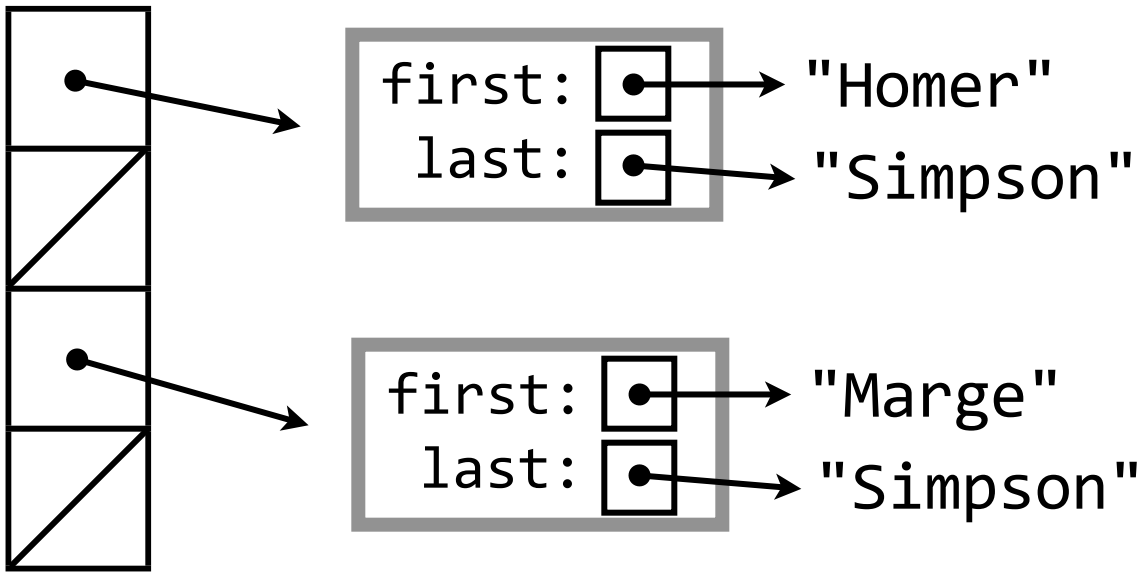
```
cities[1] = NULL;
```

Error



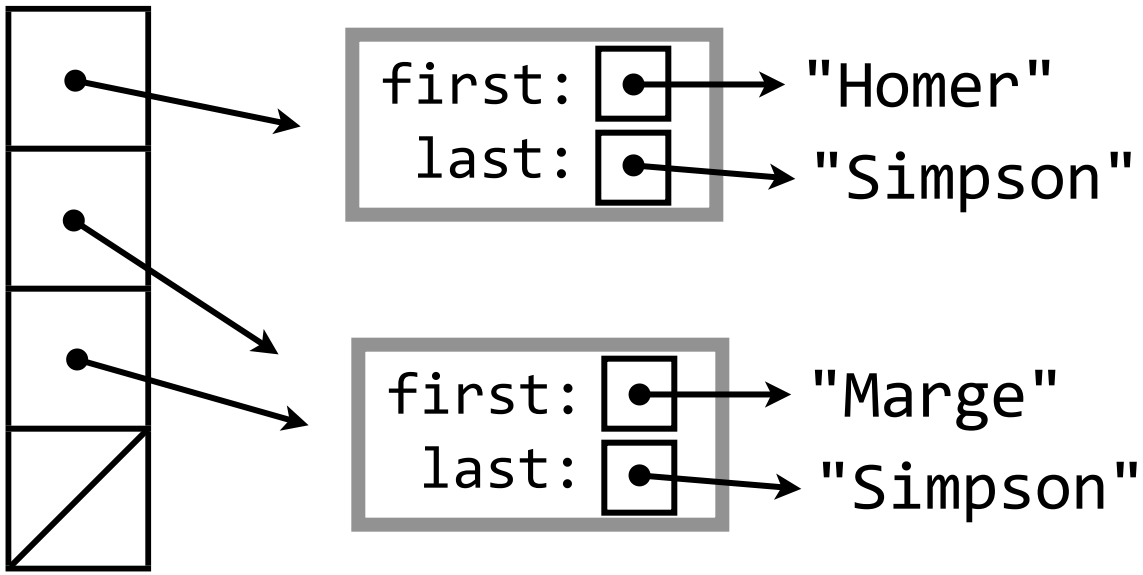
Arrays of pointers to `structs`

- Array elements are much smaller
- Array elements can be `NULL`



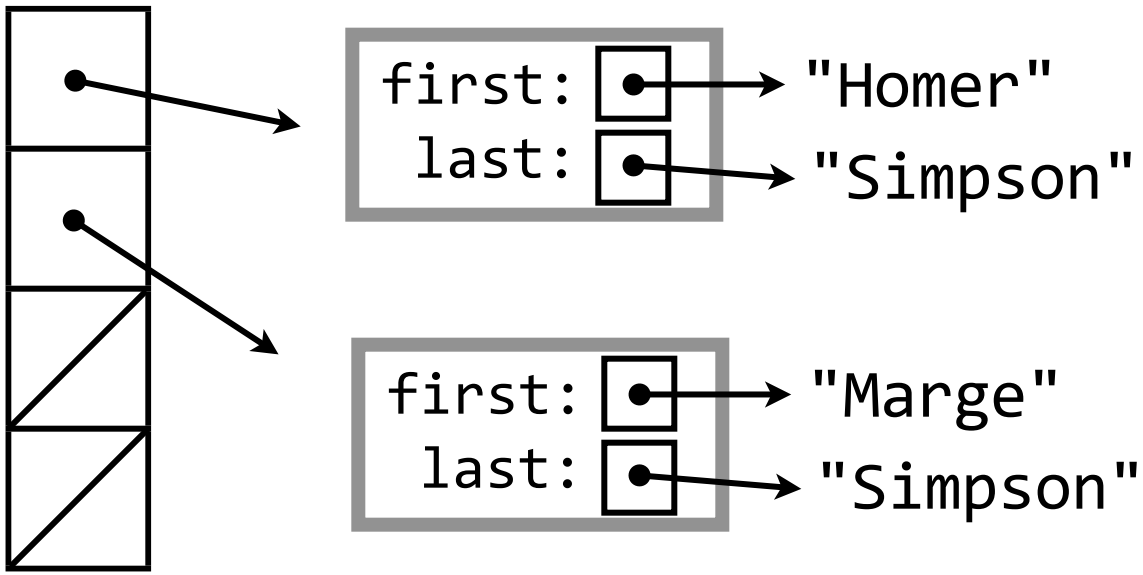
Arrays of pointers to `structs`

- Array elements are much smaller
- Array elements can be `NULL`

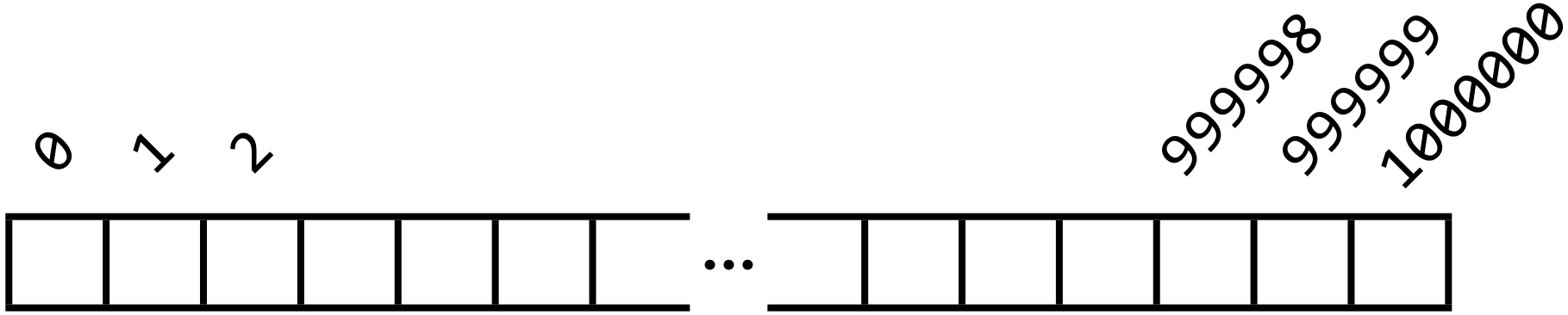


Arrays of pointers to `structs`

- Array elements are much smaller
- Array elements can be `NULL`

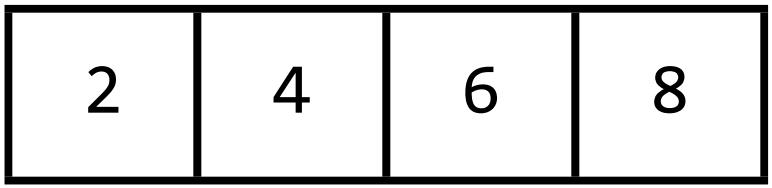


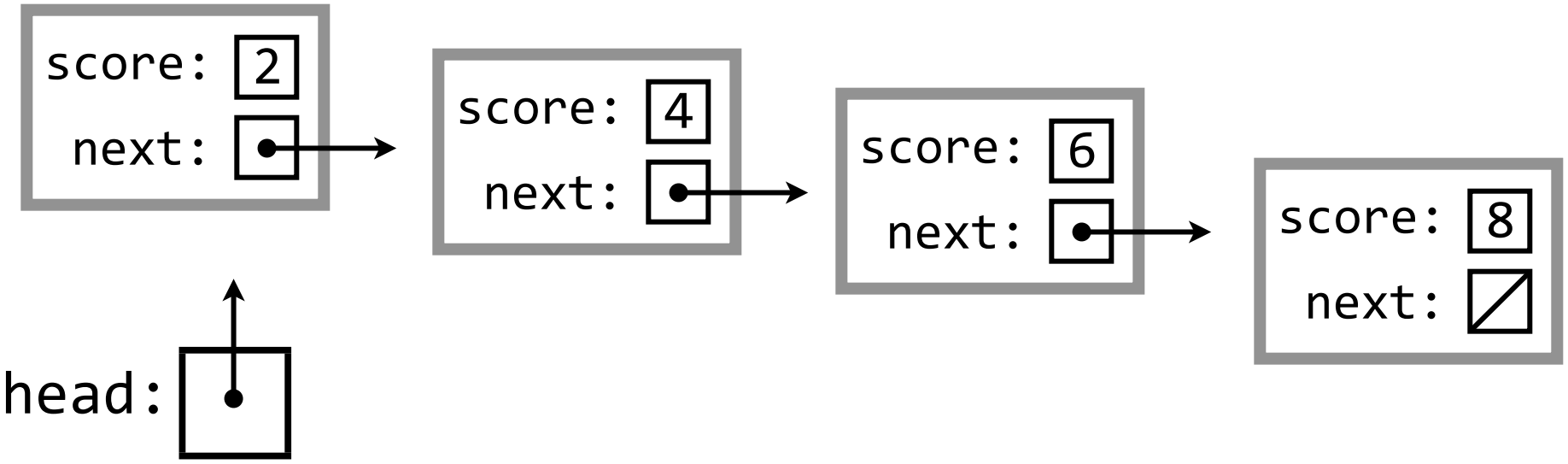
Shifting Can be Slow



Linked Lists

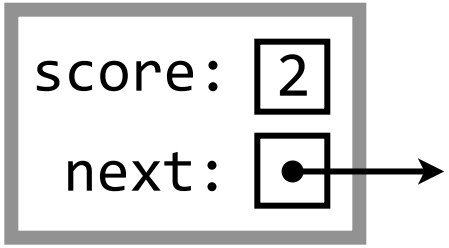
- A different way to store multiple chunks of data
- (Simplest) example of a whole category of data structures, built on indirection (i.e., using pointers)
- We have to build them ourselves

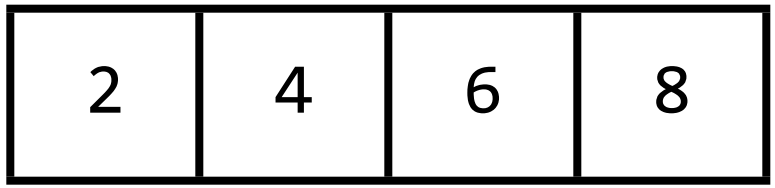

2
4
6
8

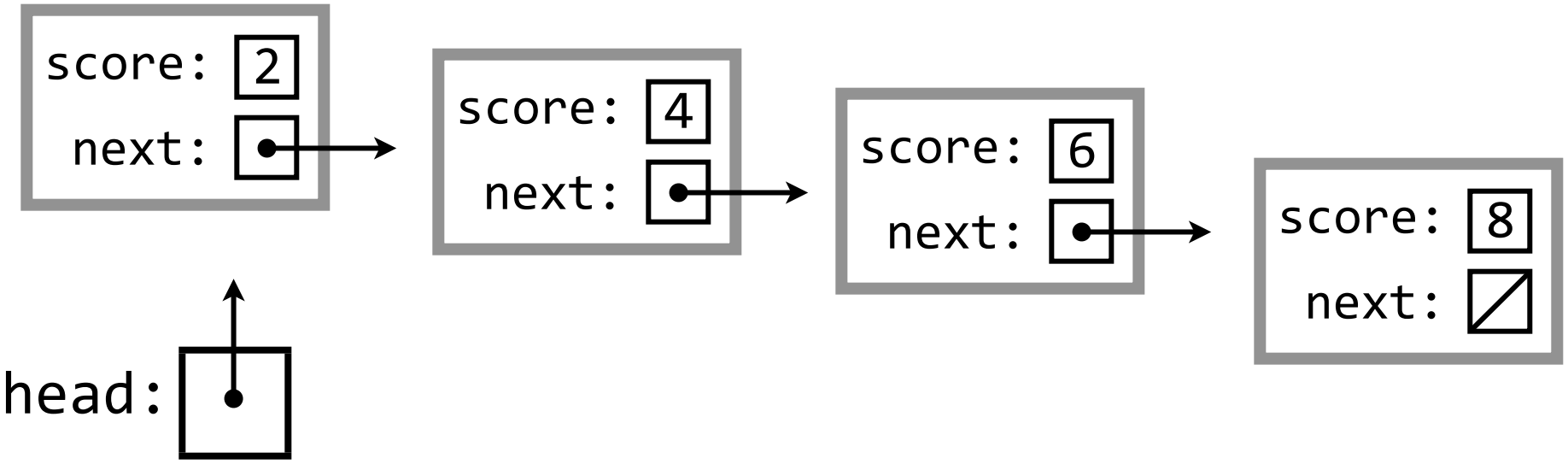

score: 2
next:
score: 4
next:
score: 6
next:
score: 8
next:
head:

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




2
4
6
8


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

A Pointer to Another Node?

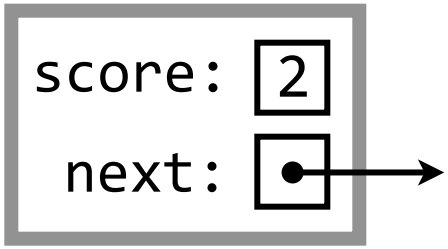
```
typedef struct node
{
   int score;
   Node next;
} Node;
```

Error

A Pointer to Another Node?

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

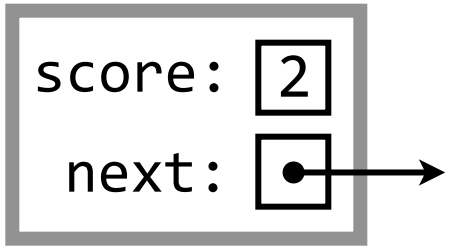
Error




```
error: syntax error before ‘Node’
warning: no semicolon at end of struct or union
warning: type defaults to ‘int’ in declaration of ‘Node’
warning: data definition has no type or storage class
```

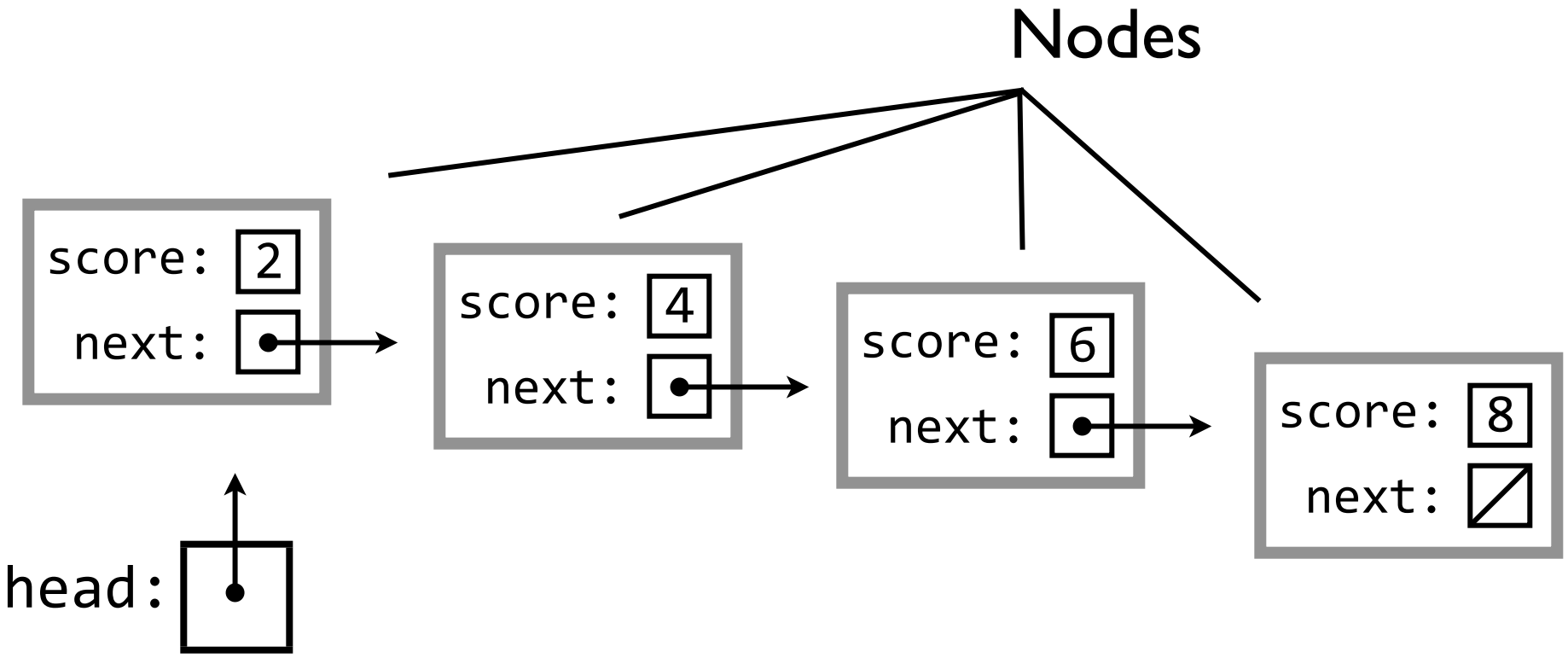
A Pointer to Another Node?

```
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

Arrays vs. Linked Lists

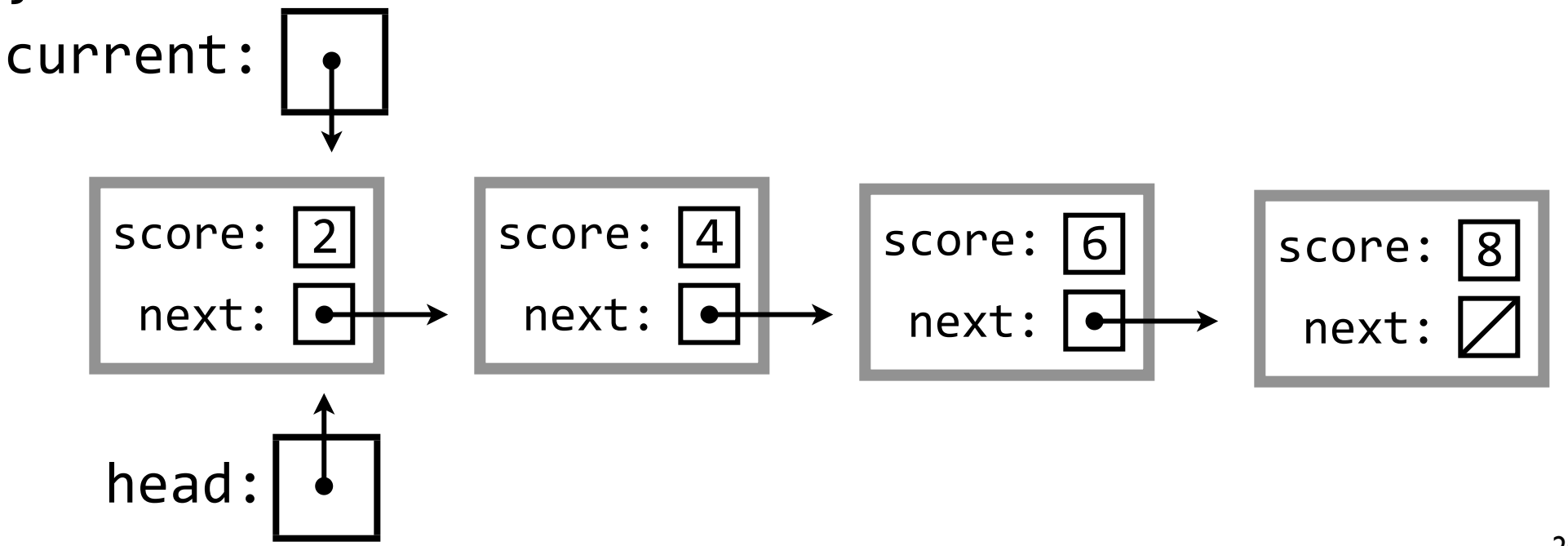
- Some things get easier
- Some things get harder
- Some things are about the same

Traversing Arrays

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

Traversing Linked Lists

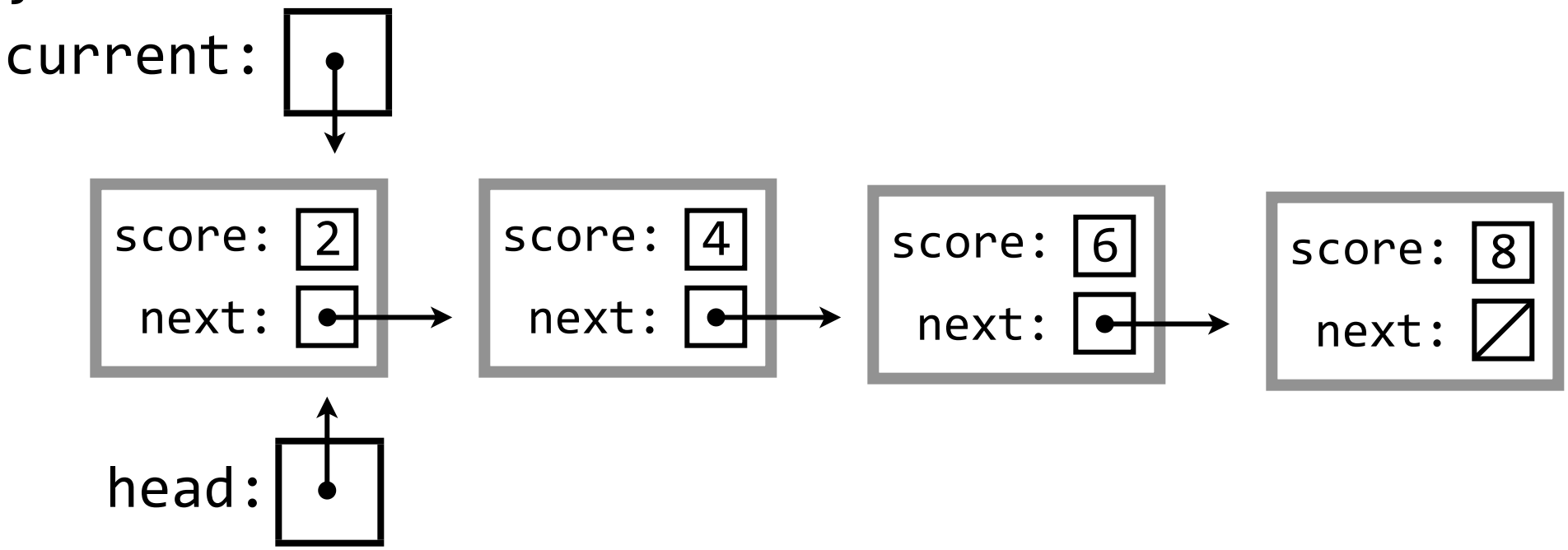
```
void printList(Node *head)
{
    Node *current = head;
    while (current != NULL)
    {
        printf("%d\n", current->score);
        current = current->next;
    }
}
```



Traversing Linked Lists

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

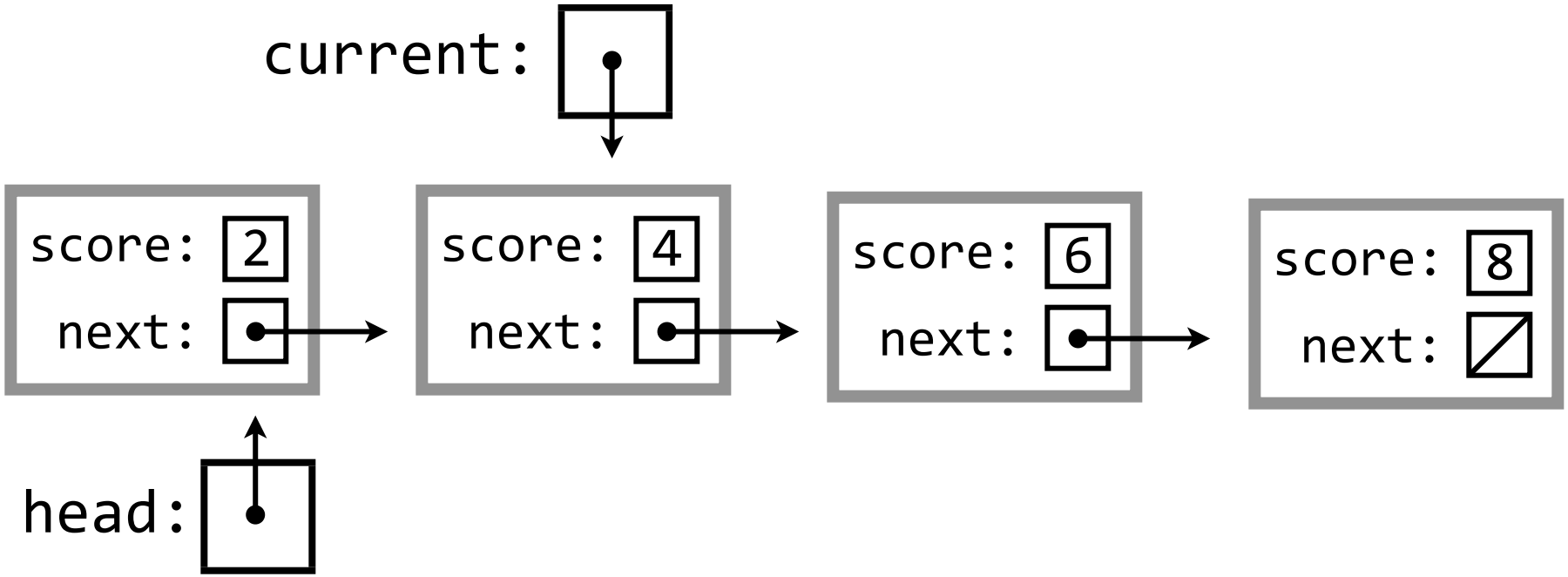
2



Traversing Linked Lists

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

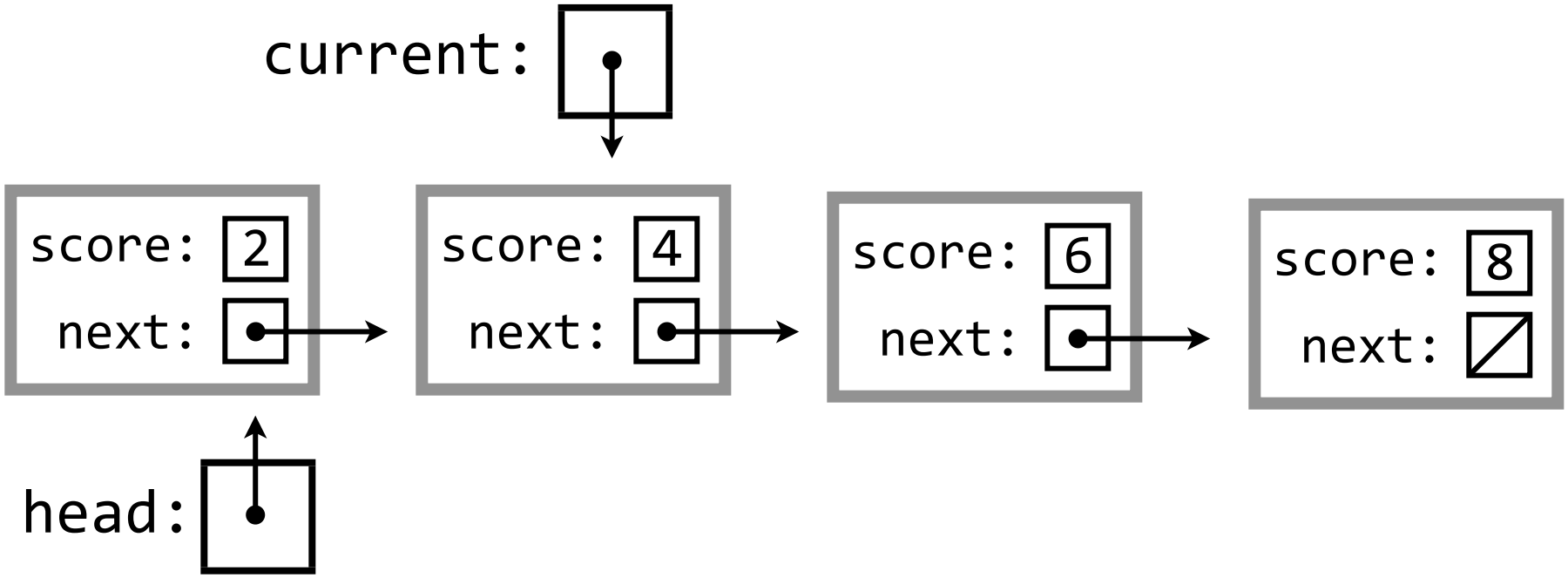
2



Traversing Linked Lists

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

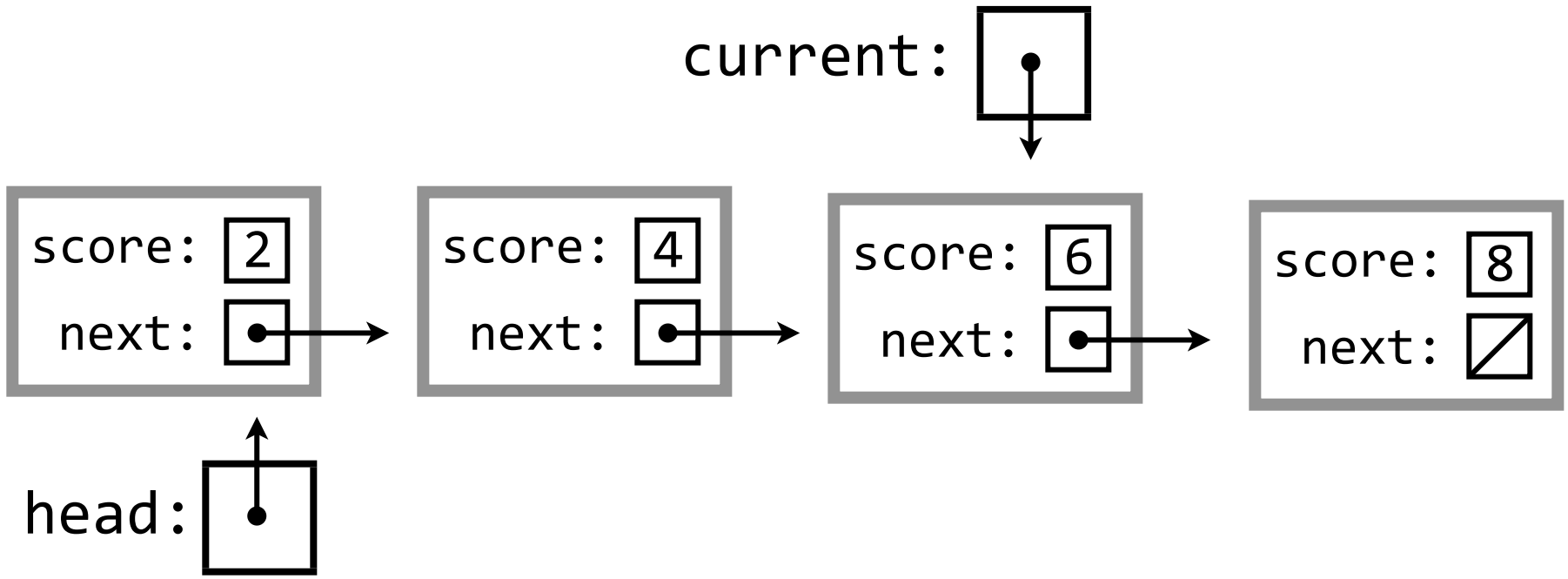
```
2
4
```



Traversing Linked Lists

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

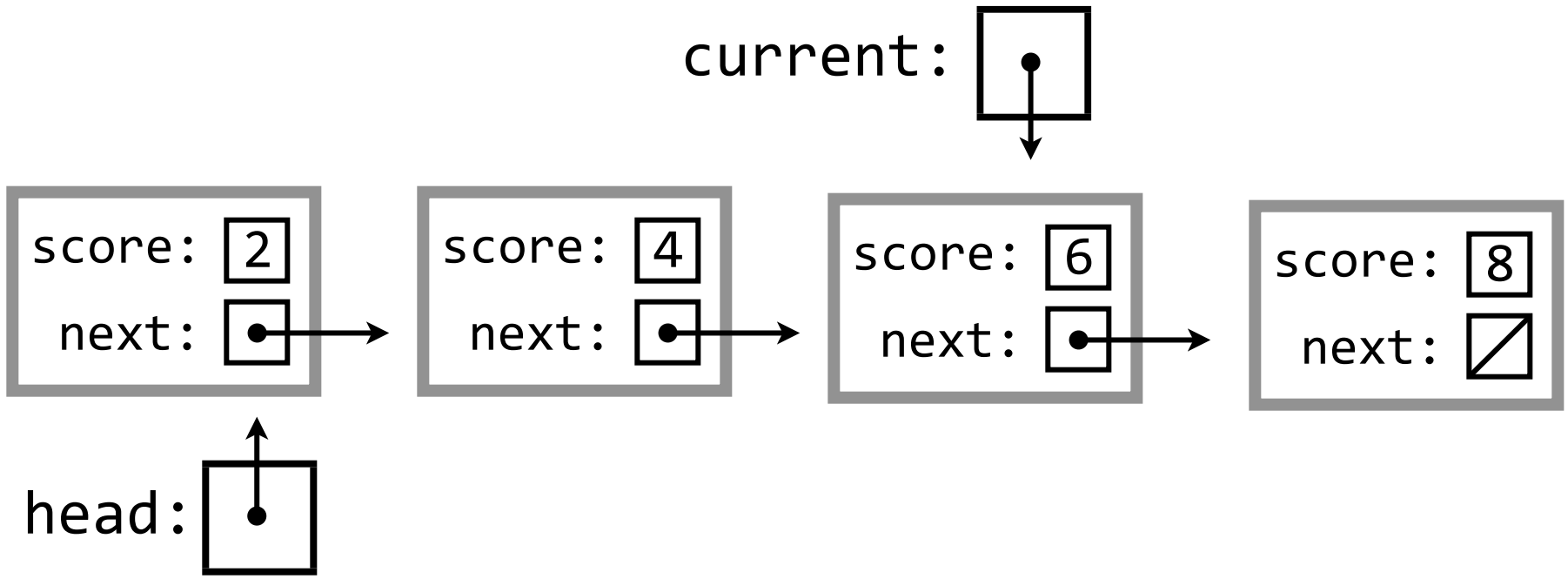
```
2
4
```



Traversing Linked Lists

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

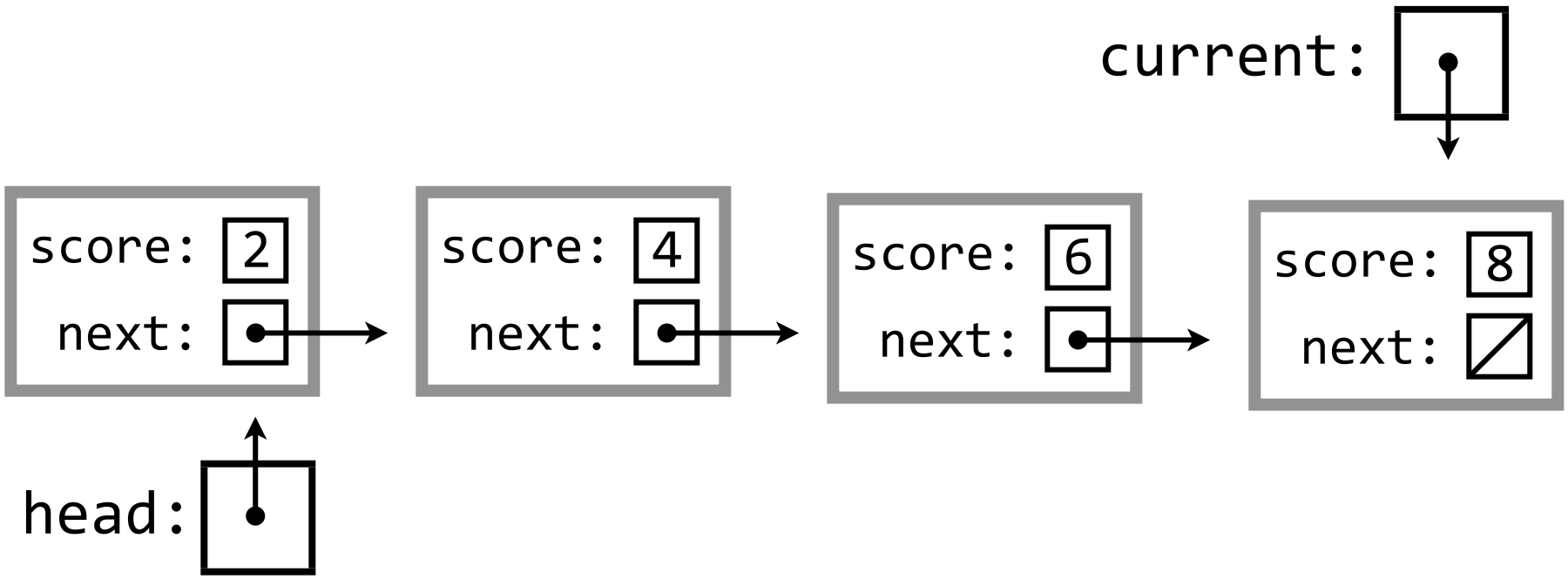
```
2
4
6
```



Traversing Linked Lists

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

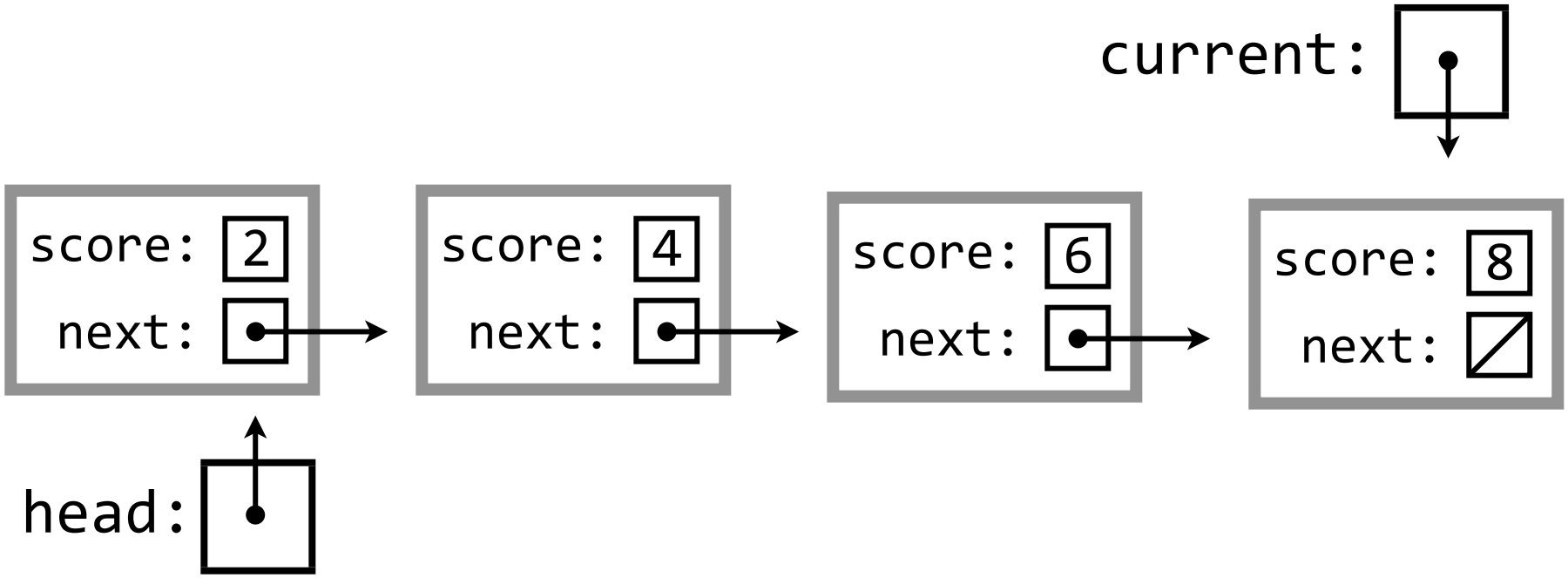
```
2
4
6
```



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
```

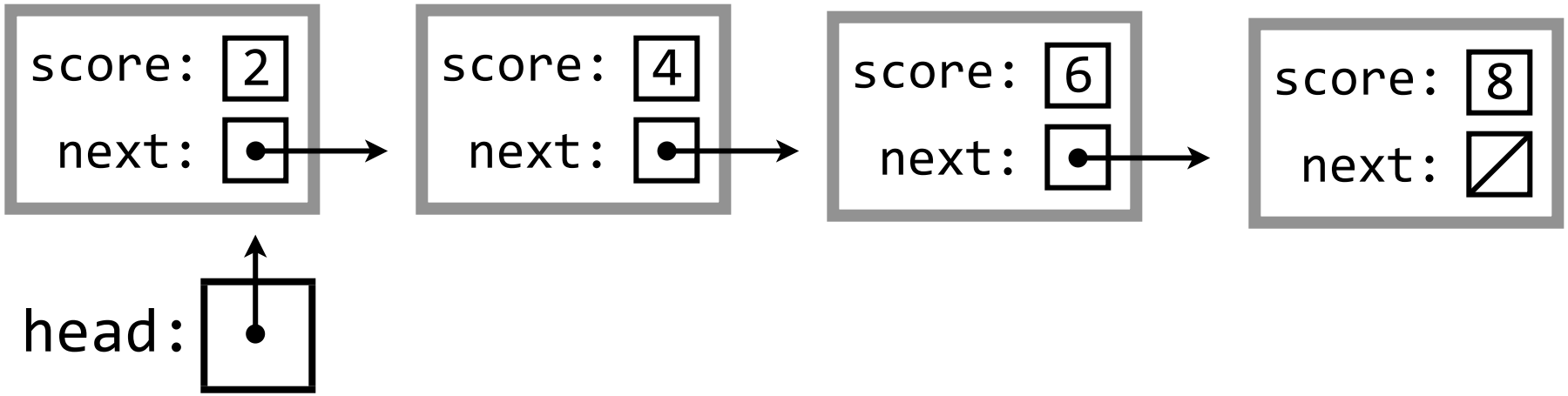


Traversing Linked Lists

```
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

Length of Arrays

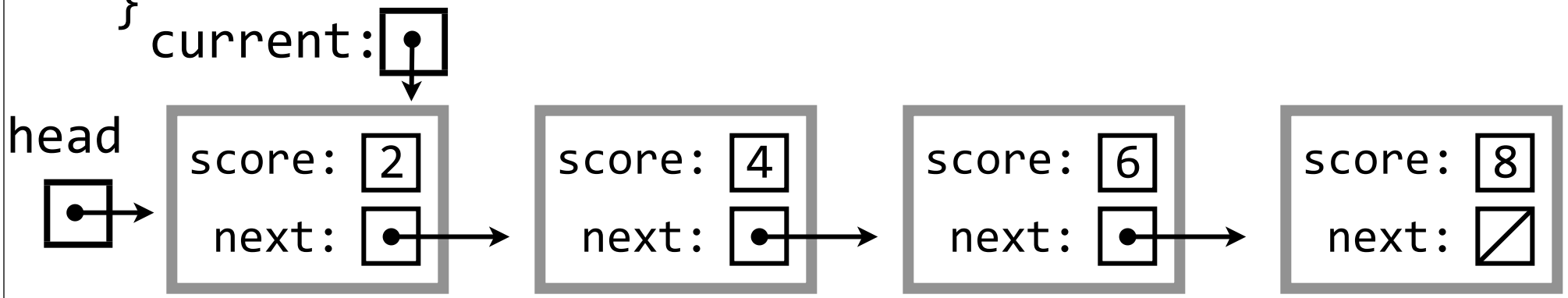
- No general technique
  - `sizeof()` under some circumstances
  - "Sentinel value" (e.g., `\0`) under some circumstances
  - Keep another variable around

Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 0

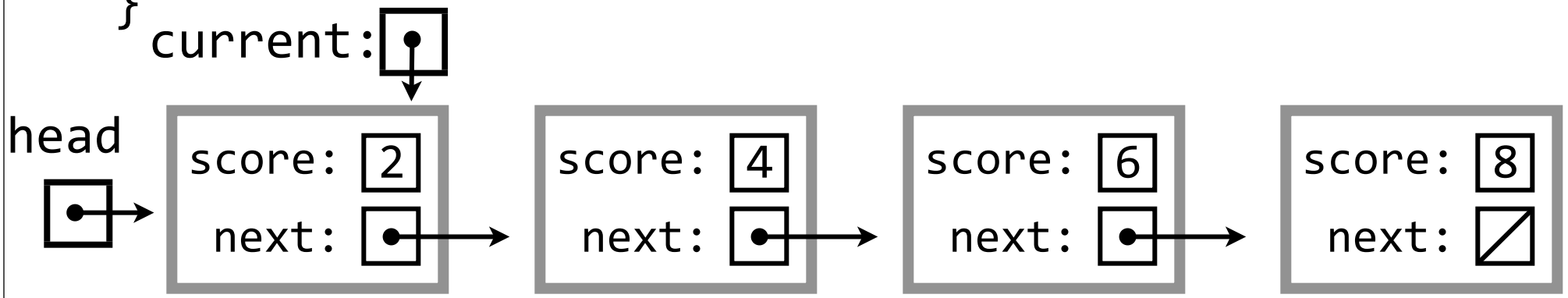


Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 1

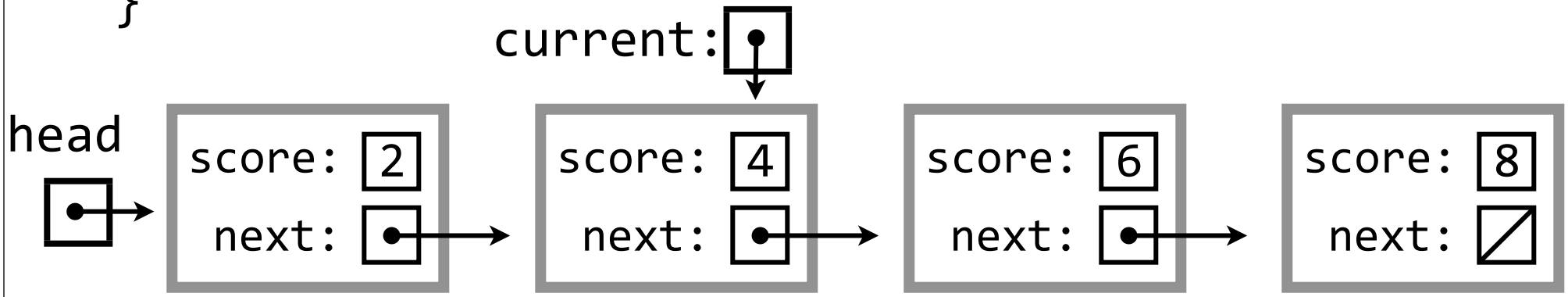


Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 1

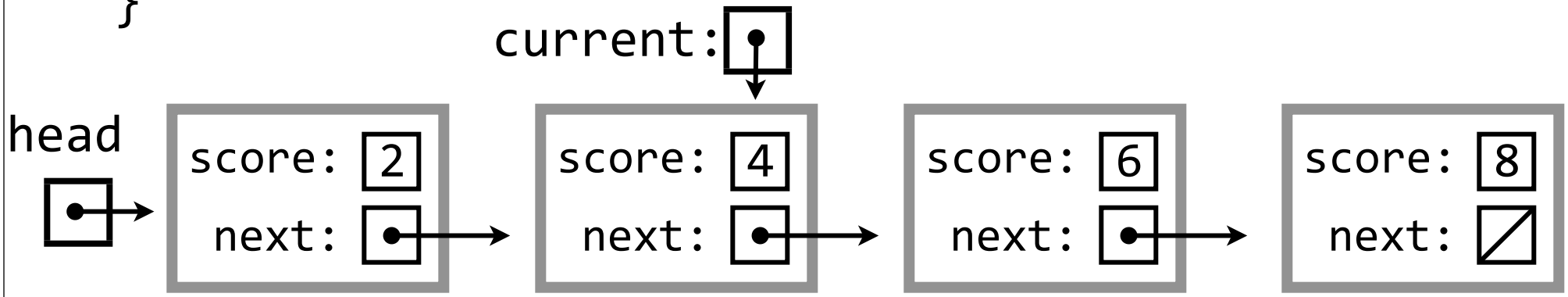


Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 2

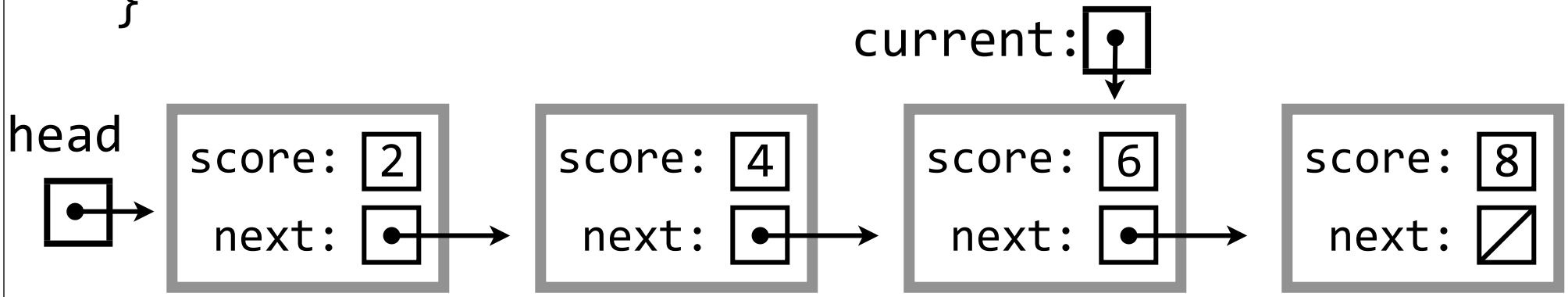


Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 2

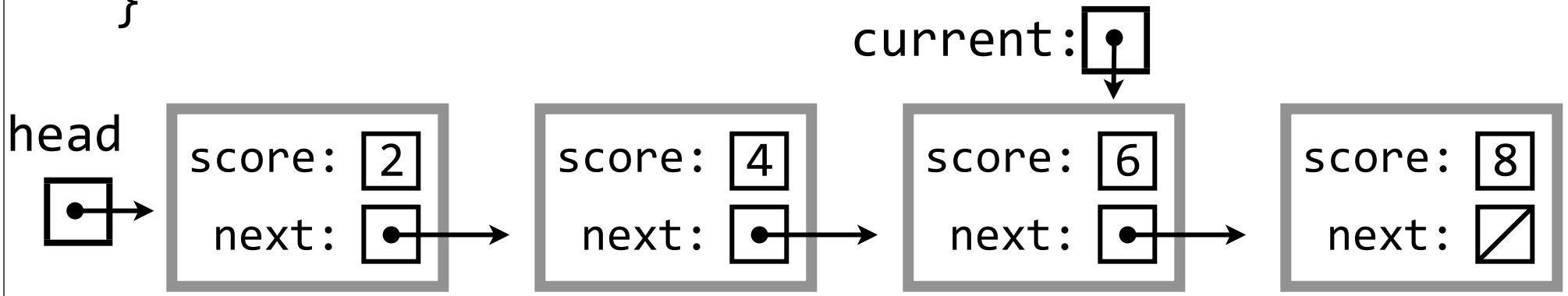


Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 3

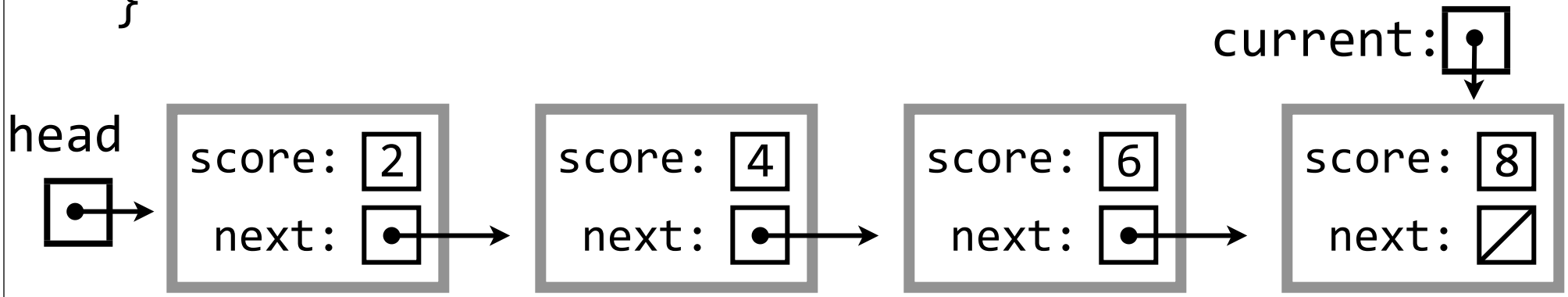


Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 3

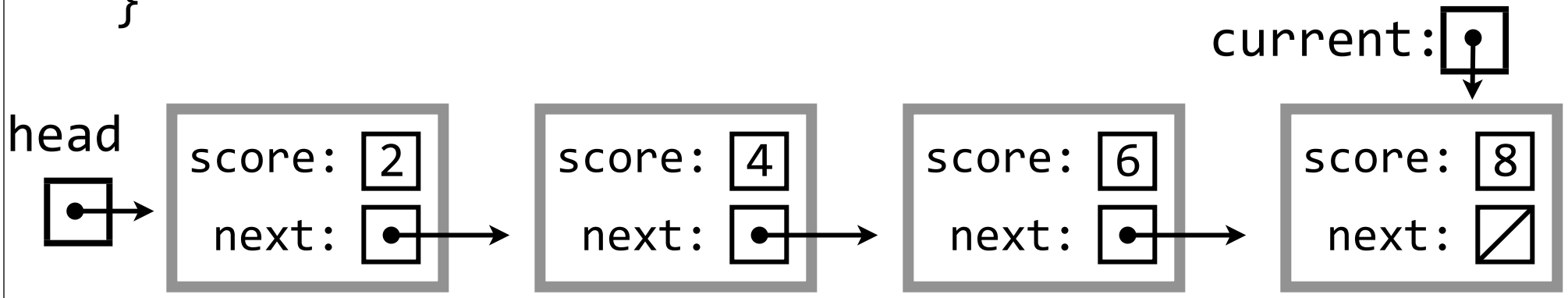


Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 4



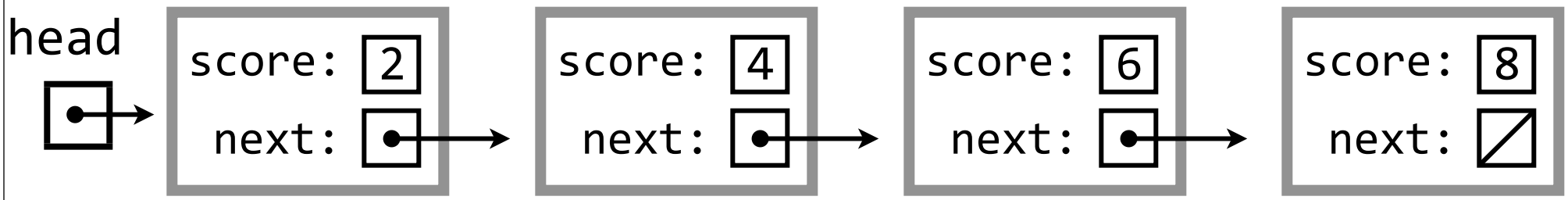
Length of Linked Lists

- Need to traverse it, but that will always work

```
int length(Node *head)
{
    int length = 0;

    Node *current = head;
    while (current != NULL)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

length: 4

current:



Verdict?

- You can always figure out the length of a linked list, but it might take you a while
- Different

`i`th Element of Array

- Really easy
- Really fast (basic math to find address)
  - Same speed for any `i`
- Potential for buffer overrun if not careful

```
City toronto = cities[2];
City vancouver = cities[29348];
```

Wrong?

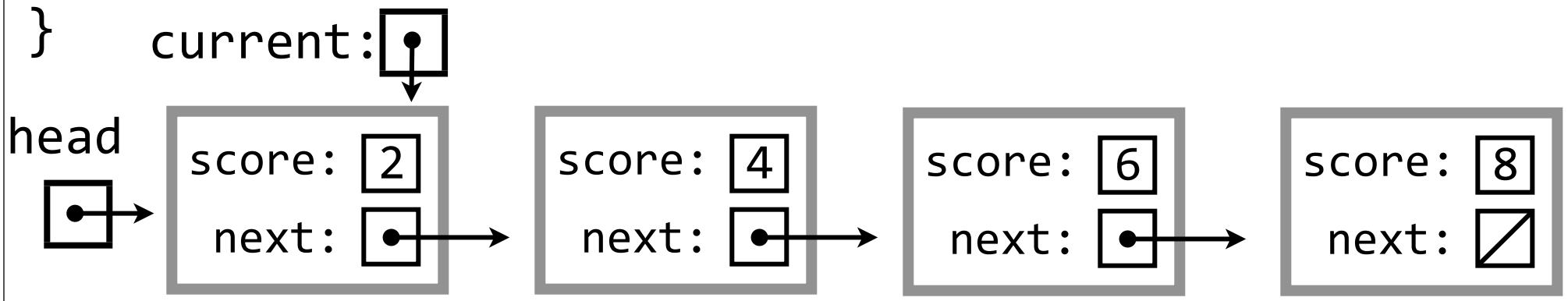
i^{th} Element of Linked List

- Need to traverse until we find it

index: 1

i: 0

```
Node *getElement(Node *head, int index)
{
  Node *current = head;
  for (int i = 0; i < index; i++)
  {
    current = current->next;
  }
  return current;
}
```



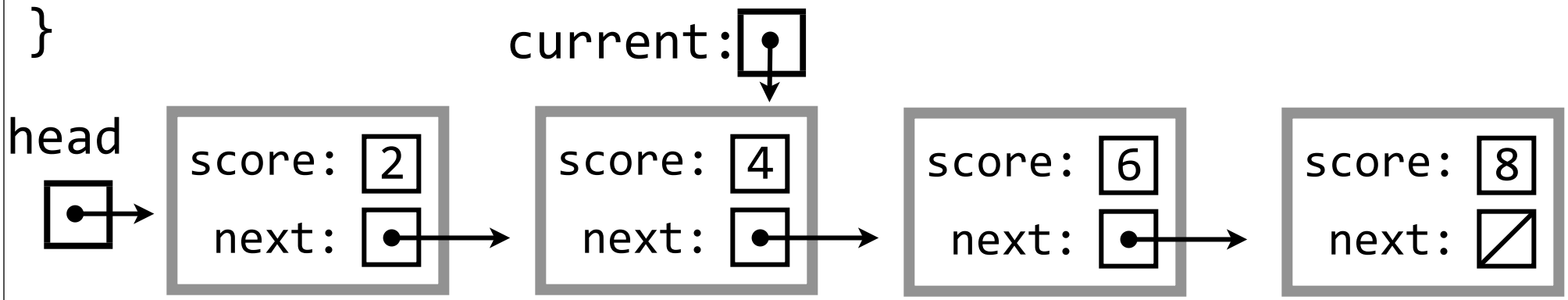
i^{th} Element of Linked List

- Need to traverse until we find it

index: 1

i: 1

```
Node *getElement(Node *head, int index)
{
  Node *current = head;
  for (int i = 0; i < index; i++)
  {
    current = current->next;
  }
  return current;
}
```



i^{th} Element of Linked List

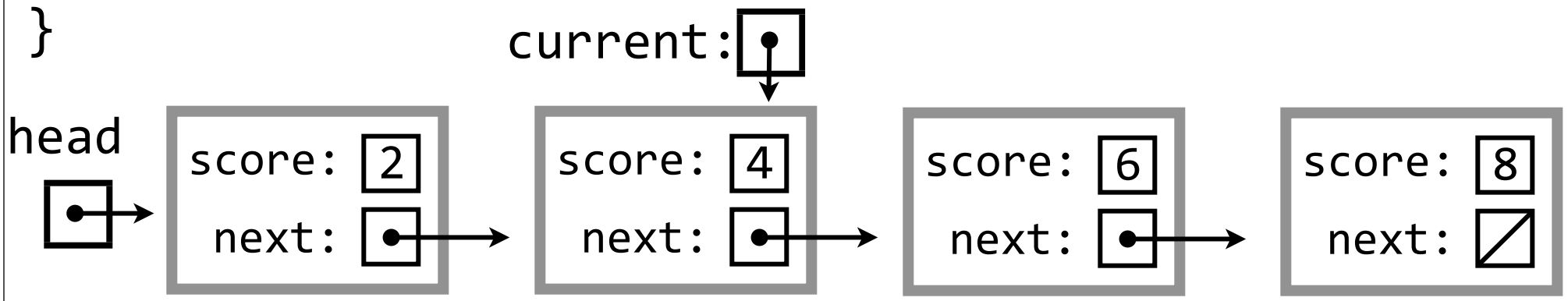
- Need to traverse until we find it

index: 10

i: 1

```
Node *getElement(Node *head, int index)
{
  Node *current = head;
  for (int i = 0; i < index; i++)
  {
    current = current->next;
  }
  return current;
}
```

What if index is larger than the number of elements in the list?



i^{th} Element of Linked List

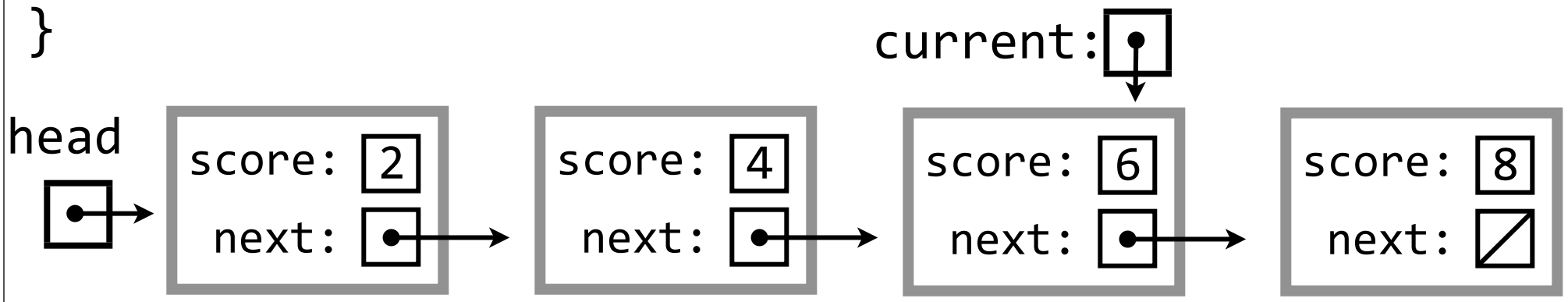
- Need to traverse until we find it

index: 10

i: 2

```
Node *getElement(Node *head, int index)
{
  Node *current = head;
  for (int i = 0; i < index; i++)
  {
    current = current->next;
  }
  return current;
}
```

What if index is larger than the number of elements in the list?



i^{th} Element of Linked List

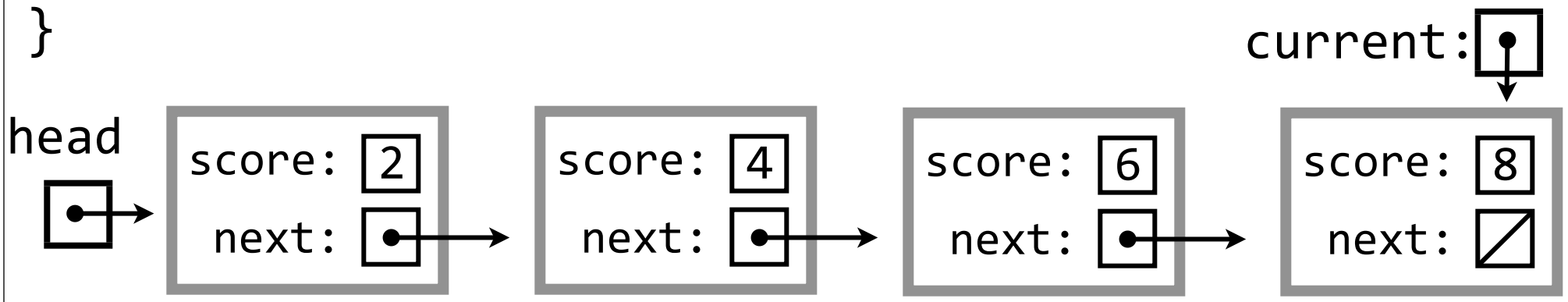
- Need to traverse until we find it

index: 10

i: 3

```
Node *getElement(Node *head, int index)
{
  Node *current = head;
  for (int i = 0; i < index; i++)
  {
    current = current->next;
  }
  return current;
}
```

What if index is larger than the number of elements in the list?



i^{th} Element of Linked List

- Need to traverse until we find it

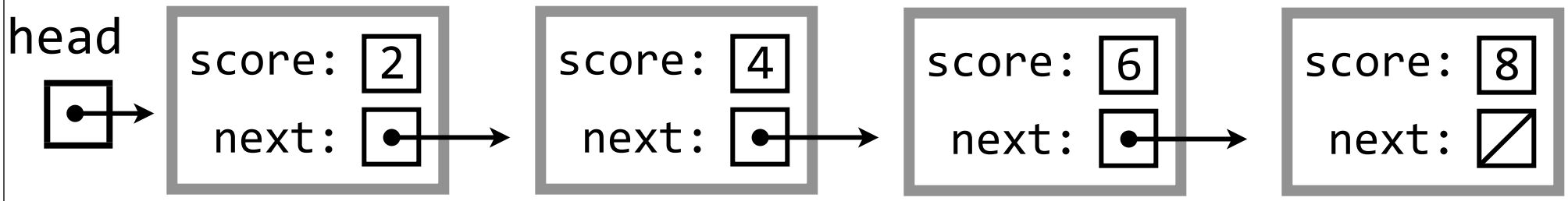
index: 10

i: 4

```
Node *getElement(Node *head, int index)
{
  Node *current = head;
  for (int i = 0; i < index; i++)
  {
    current = current->next;
  }
  return current;
}
```

What if index is larger than the number of elements in the list?

current: ⊘



i^{th} Element of Linked List

- Need to traverse until we find it

index: 10

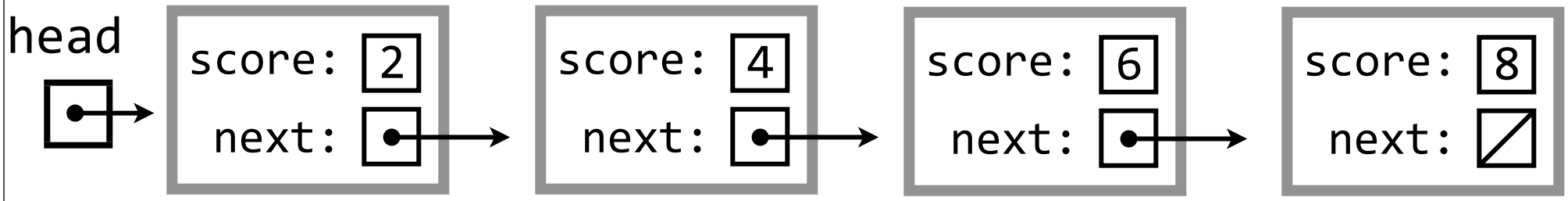
i: 4

```
Node *getElement(Node *head, int index)
{
  Node *current = head;
  for (int i = 0; i < index; i++)
  {
    current = current->next;
  }
  return current;
}
```

Wrong

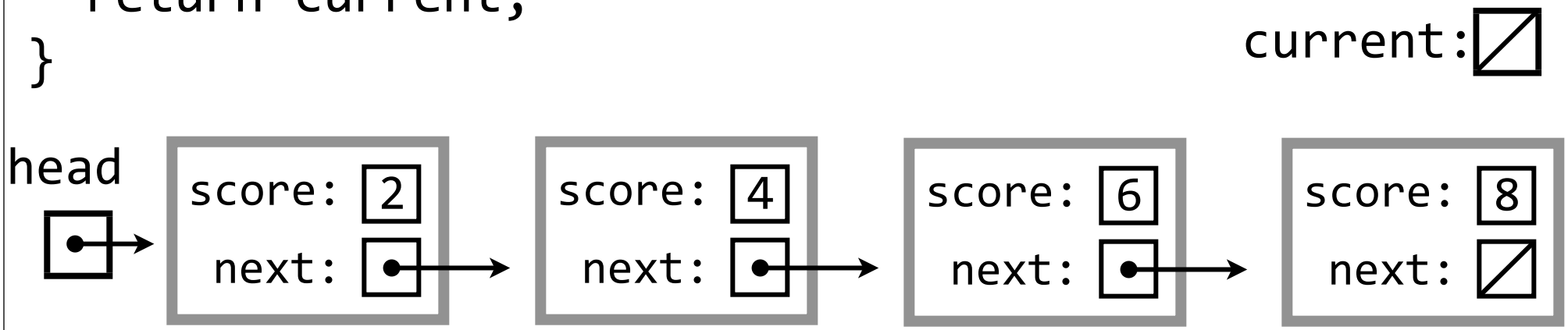
What if index is larger than the number of elements in the list?

current: ⊘



i^{th} Element of Linked List

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



Verdict?

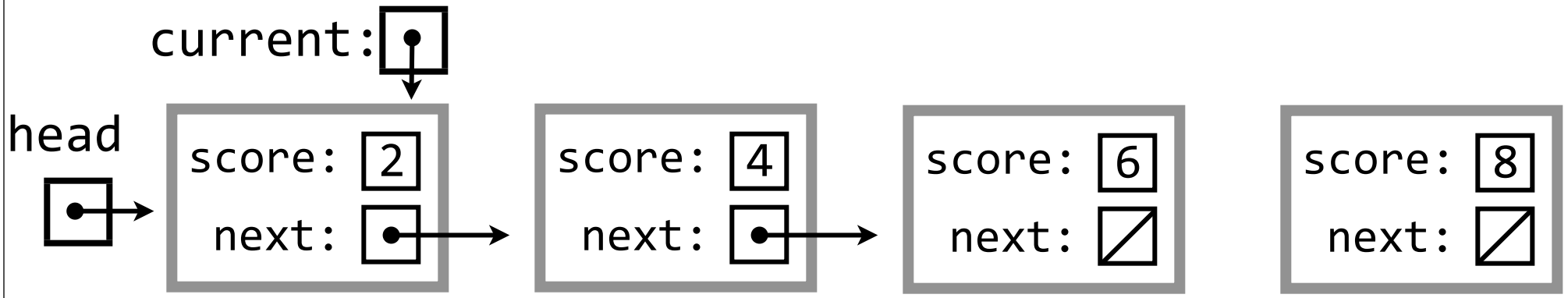
- Arrays are much easier
- Arrays are much faster, and have constant run time

Insert at End of Array

```
// If the array has space
list[numElements] = newItem;
numElements++;


// If the array is full but dynamically allocated
list = realloc(list, (numElements+1) * sizeof(list[0]);
list[numElements] = newItem;
numElements++;


// If the array is full but automatically allocated
// Out of luck!
```

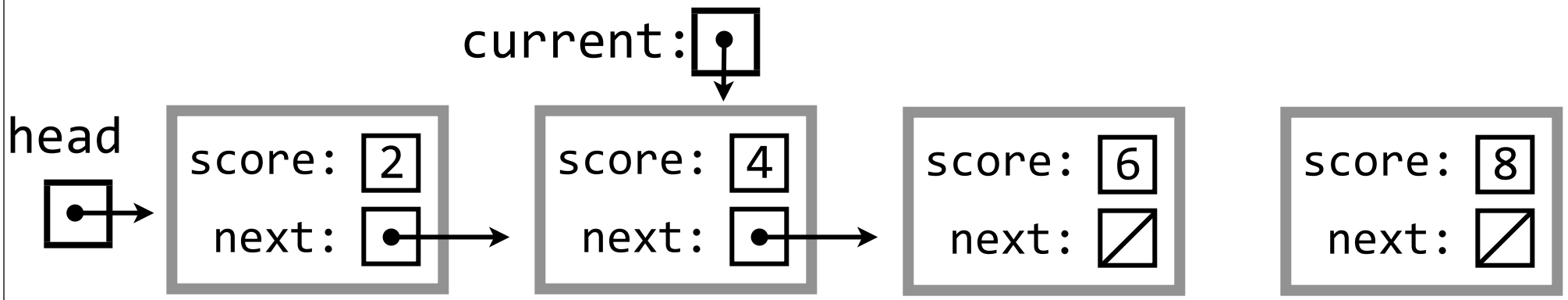
Insert at End of Linked List

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



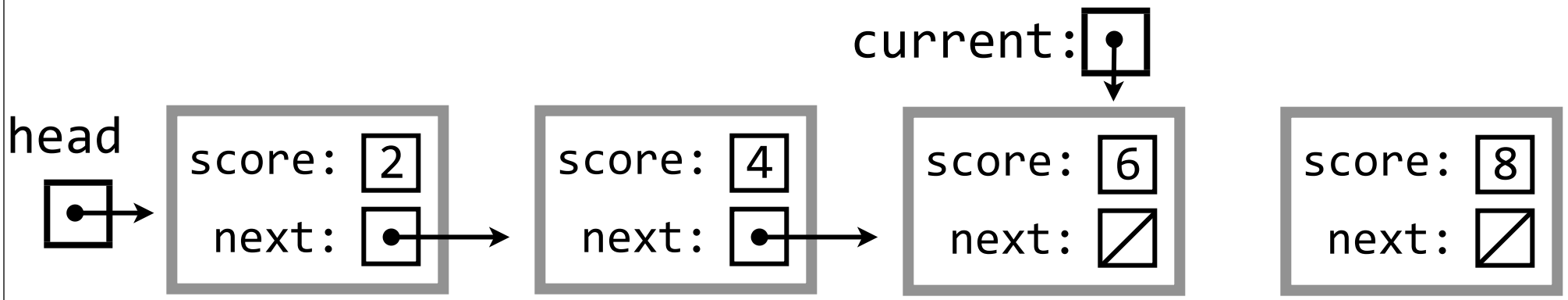
Insert at End of Linked List

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



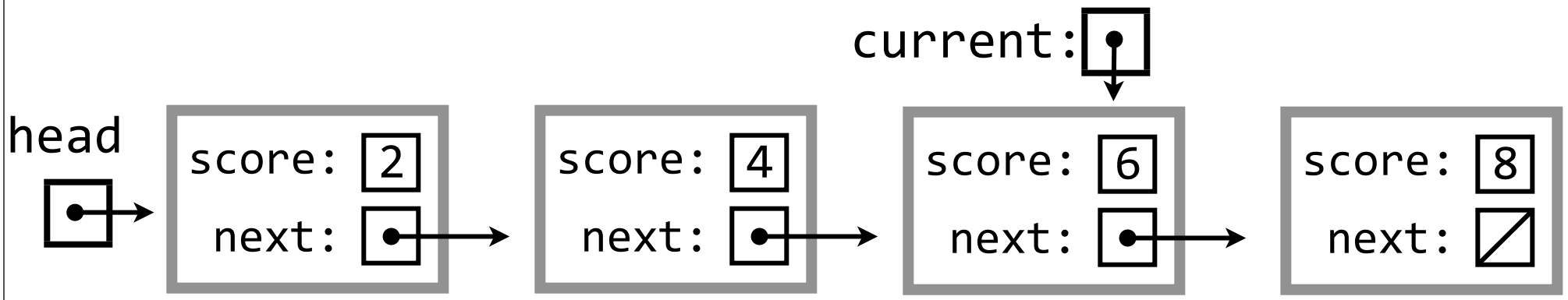
Insert at End of Linked List

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 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



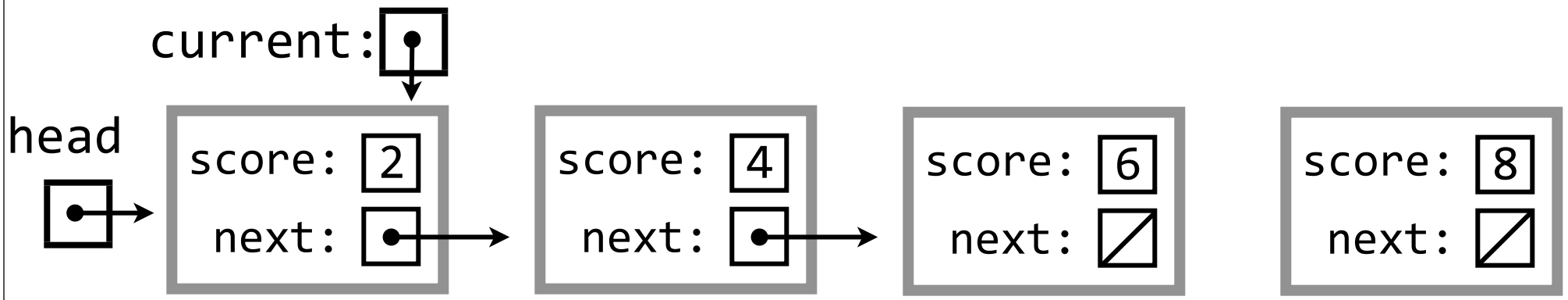
Insert at End of Linked List

```
void addToEnd(Node *head, Node *newNode)
{
    Node *current = head;
    while (current != NULL)
    {
        current = current->next;
    }

    current->next = newNode;
    newNode->next = NULL;
}
```

Wrong

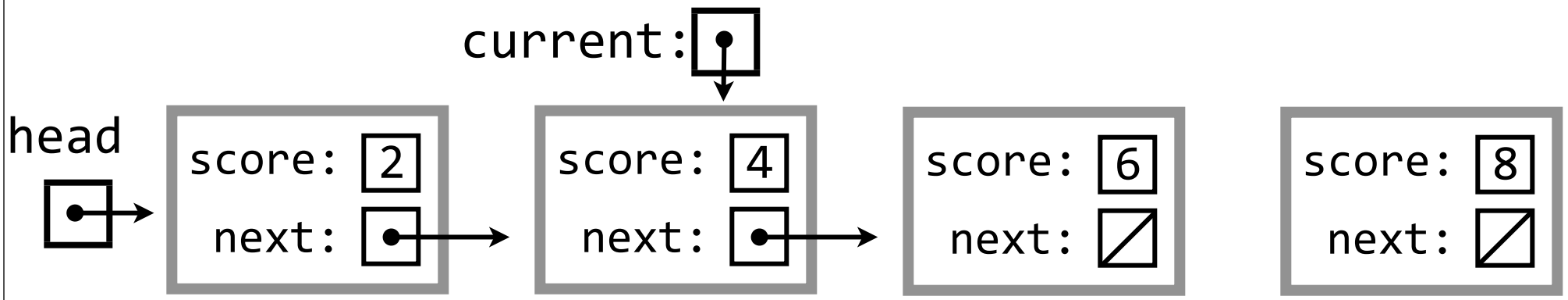
Insert at End of Linked List

```
void addToEnd(Node *head, Node *newNode)
{
    Node *current = head;
    while (current != NULL)
    {
        current = current->next;
    }

    current->next = newNode;
    newNode->next = NULL;
}
```



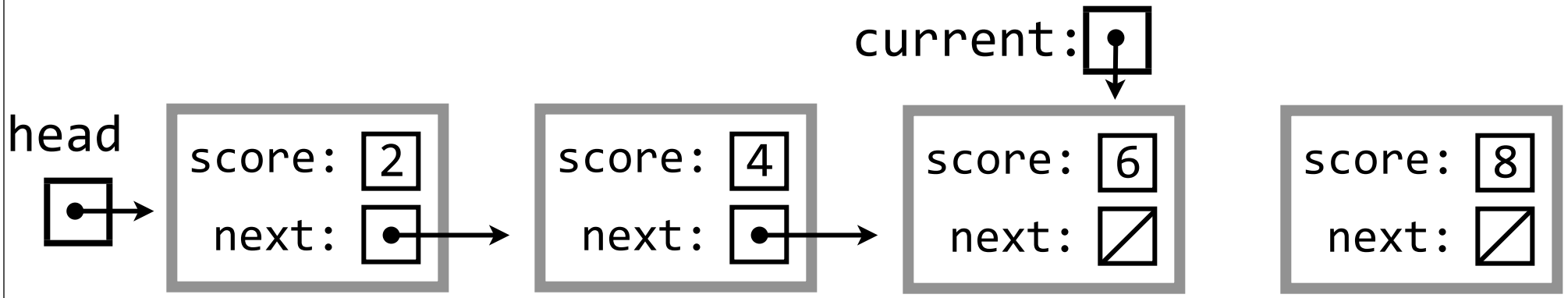
Insert at End of Linked List

```
void addToEnd(Node *head, Node *newNode)
{
    Node *current = head;
    while (current != NULL)
    {
        current = current->next;
    }

    current->next = newNode;
    newNode->next = NULL;
}
```



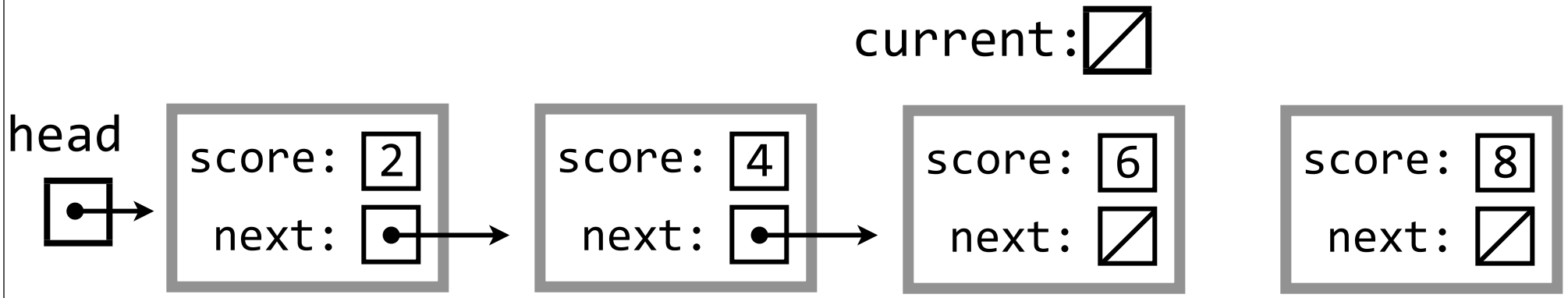
Insert at End of Linked List

```
void addToEnd(Node *head, Node *newNode)
{
    Node *current = head;
    while (current != NULL)
    {
        current = current->next;
    }

    current->next = newNode;
    newNode->next = NULL;
}
```



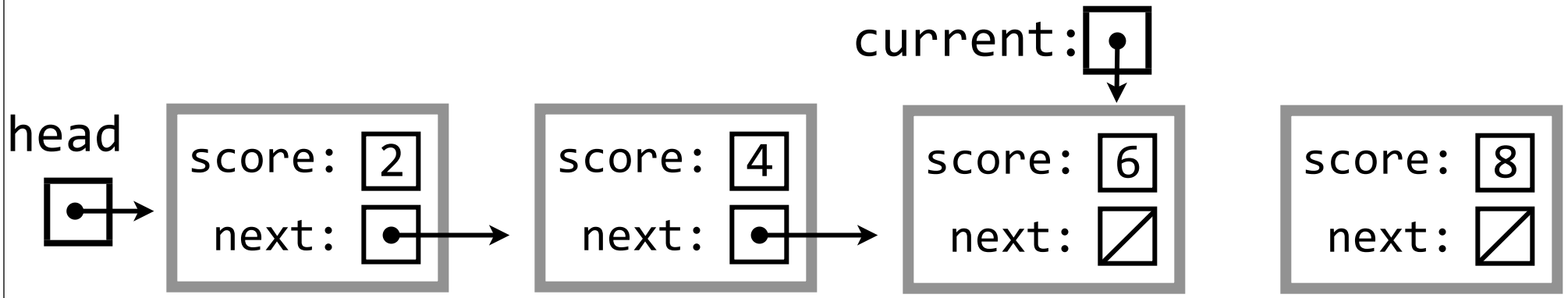
Insert at End of Linked List

```
void addToEnd(Node *head, Node *newNode)
{
    Node *current = head;
    while (current != NULL)
    {
        current = current->next;
    }

    current->next = newNode;
    newNode->next = NULL;
}
```



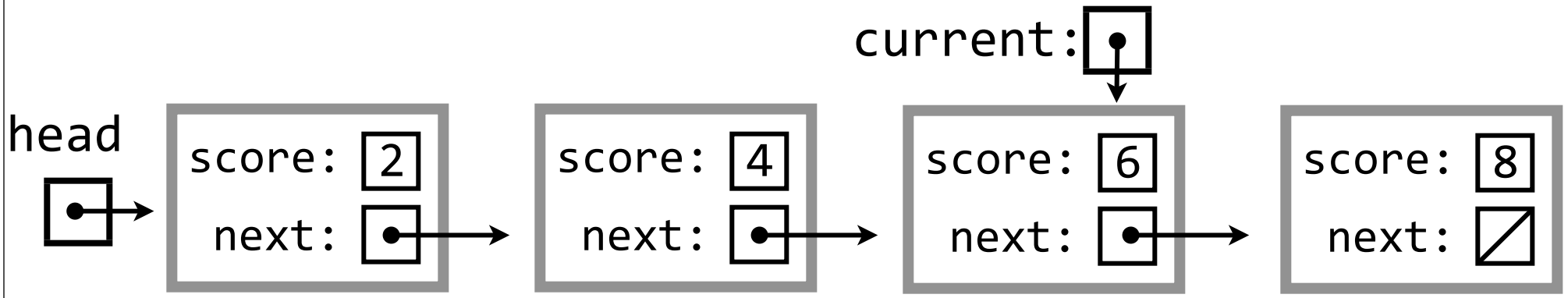
Insert at End of Linked List

```
void addToEnd(Node *head, Node *newNode)
{
    Node *current = head;
    while (current->next != NULL)
    {
        current = current->next;
    }

    current->next = newNode;
    newNode->next = NULL;
}
```



Insert at End of Linked List

```
void addToEnd(Node *head, Node *newNode)
{
    Node *current = head;
    while (current->next != NULL)
    {
        current = current->next;
    }

    current->next = newNode;
    newNode->next = NULL;
}
```

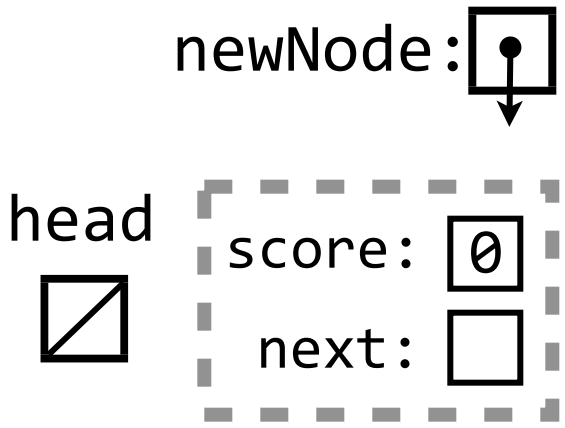


Insert at End of Linked List

```
Node *addToEnd(Node *head, Node *newNode)
{
    if (head == NULL)
    {
        newNode->next = NULL;
        return newNode;
    }

    Node *current = head;
    while (current->next != NULL)
    {
        current = current->next;
    }

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

    return head;
}
```

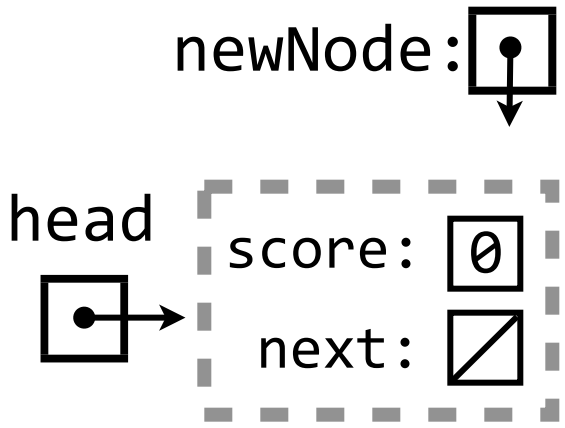
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToEnd(head, newNode);
}
```



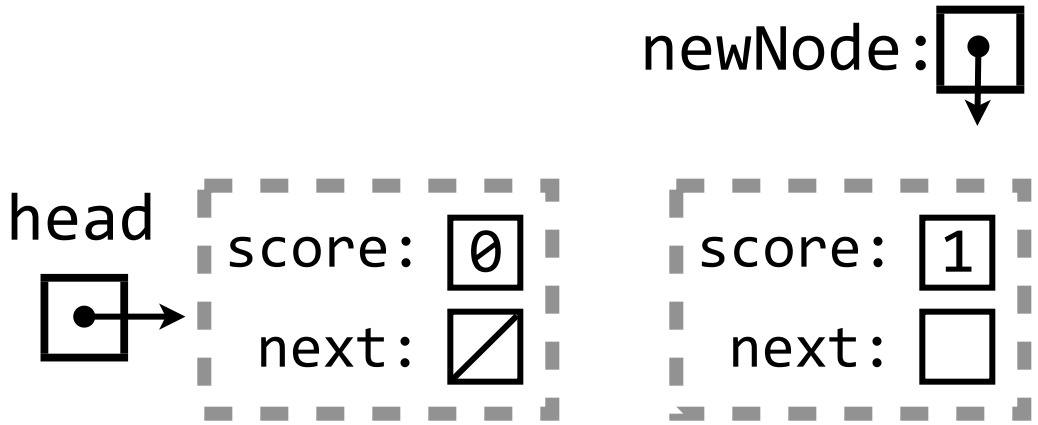
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToEnd(head, newNode);
}
```



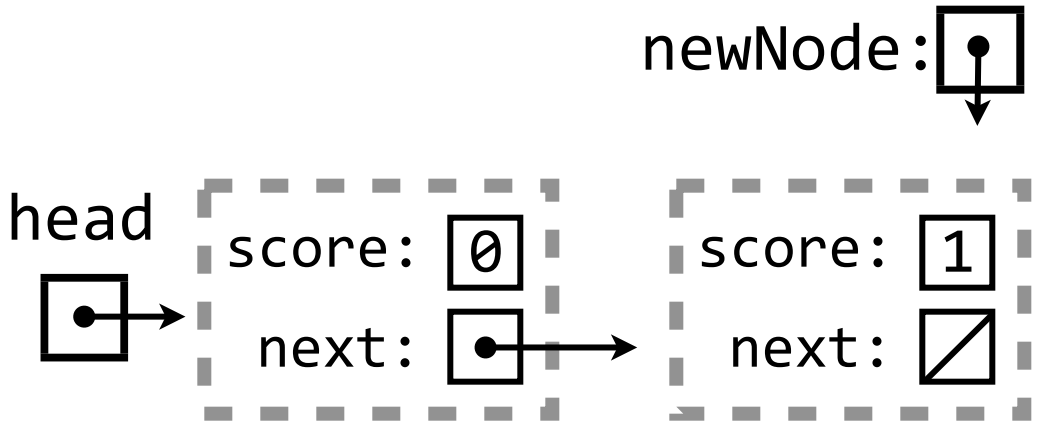
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToEnd(head, newNode);
}
```



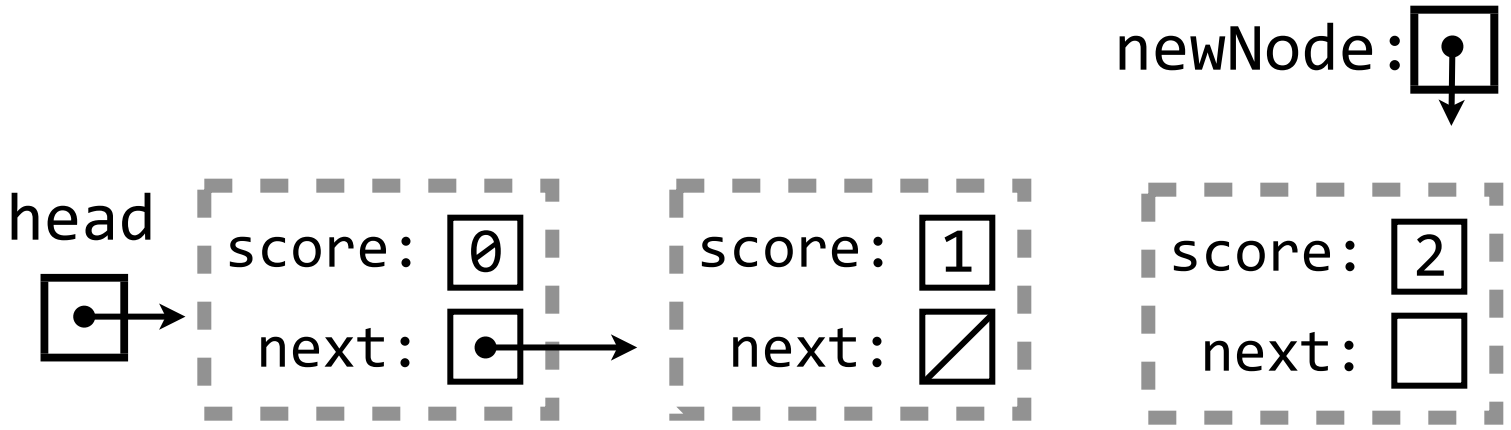
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToEnd(head, newNode);
}
```



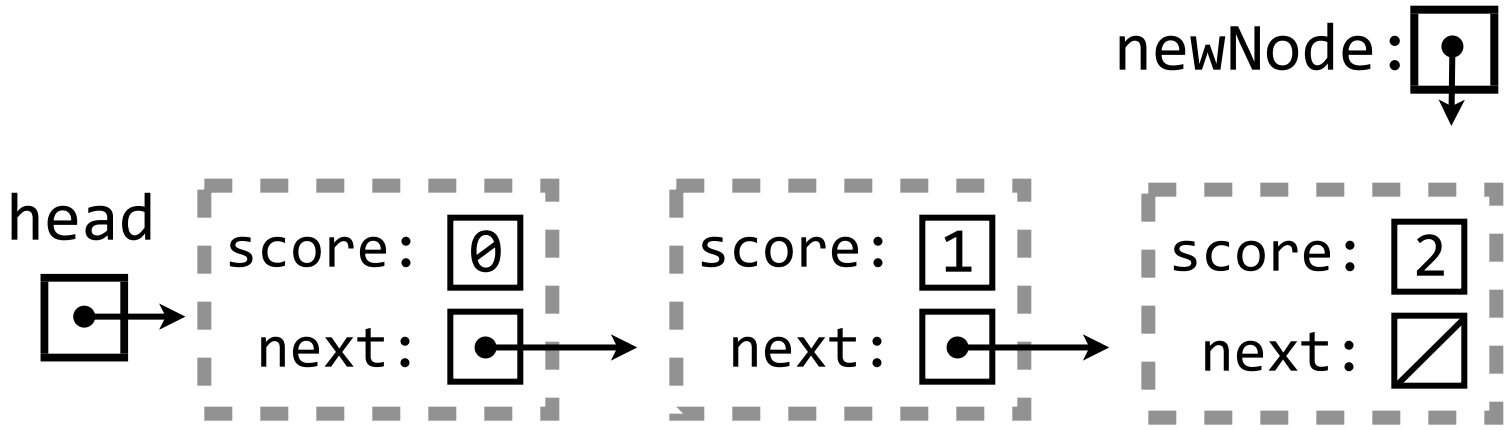
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToEnd(head, newNode);
}
```



Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToEnd(head, newNode);
}
```



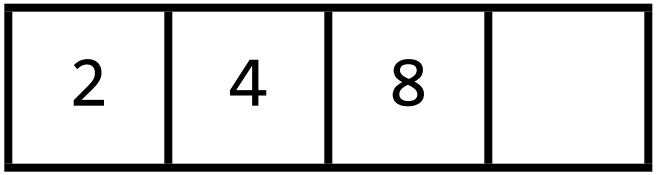
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToEnd(head, newNode);
}
```

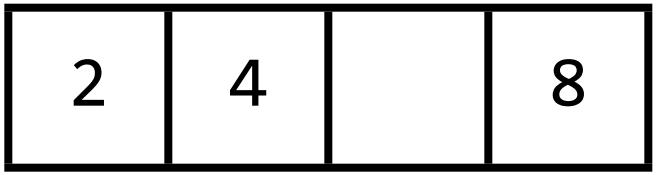
Verdict?

- Array easier, if it has space
- Array (much?) harder, if it doesn't
- Linked list never runs out of space
- Linked list needs to traverse entire list

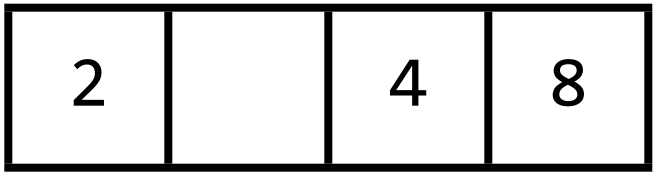
Insert at Beginning of Array



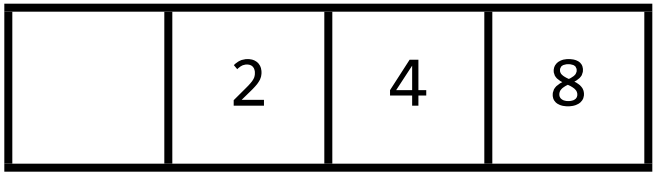
Insert at Beginning of Array



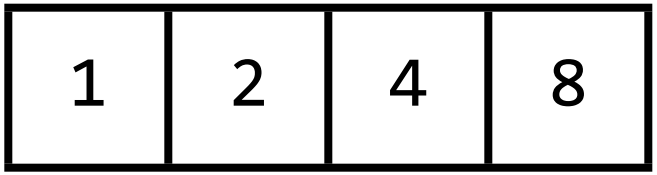
Insert at Beginning of Array



Insert at Beginning of Array

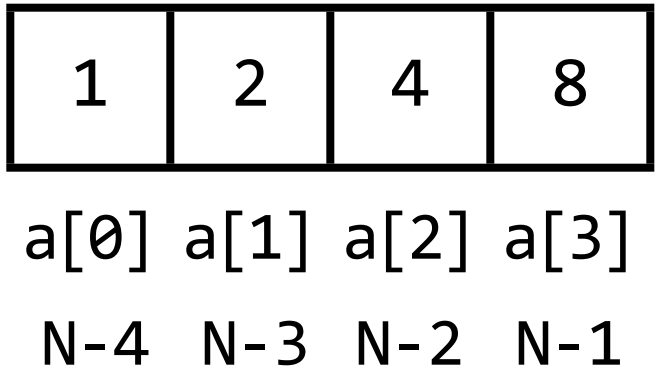


Insert at Beginning of Array



Insert at Beginning of Array

```
// Assuming the array is large enough
for (int i = N - 2;i >= 0; i--)
{
  a[i + 1] = a[i];
}
a[0] = newItem;
```



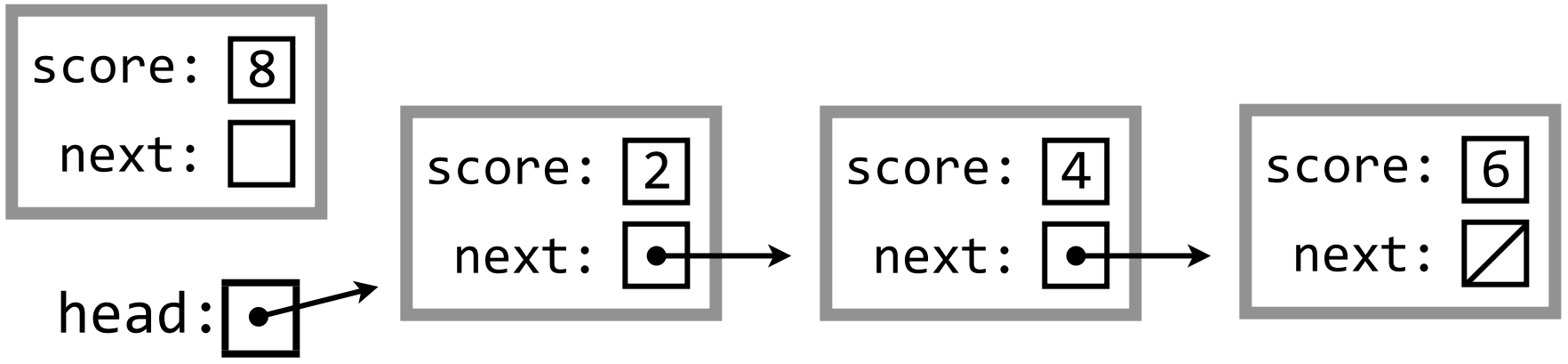
Insert at Beg. of Linked List

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



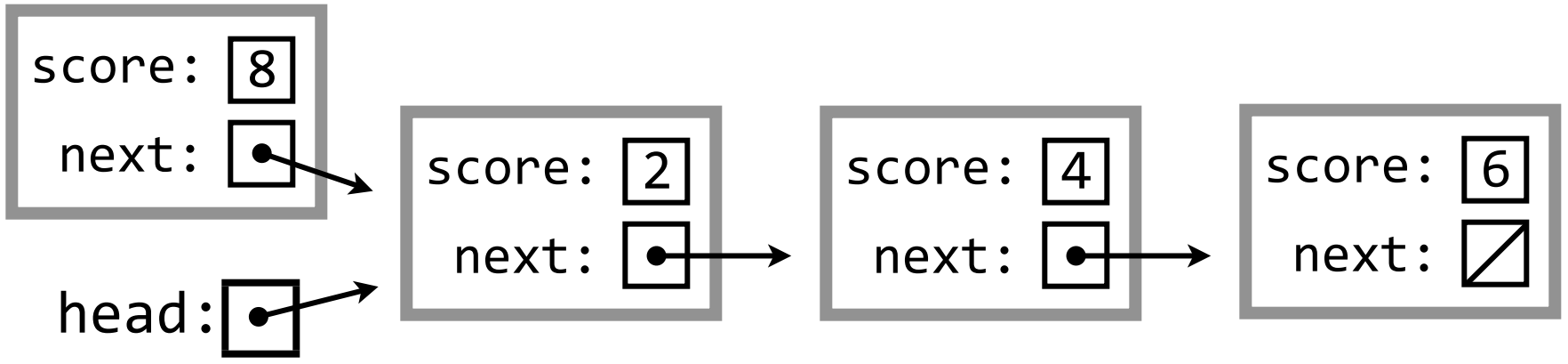
Insert at Beg. of Linked List

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



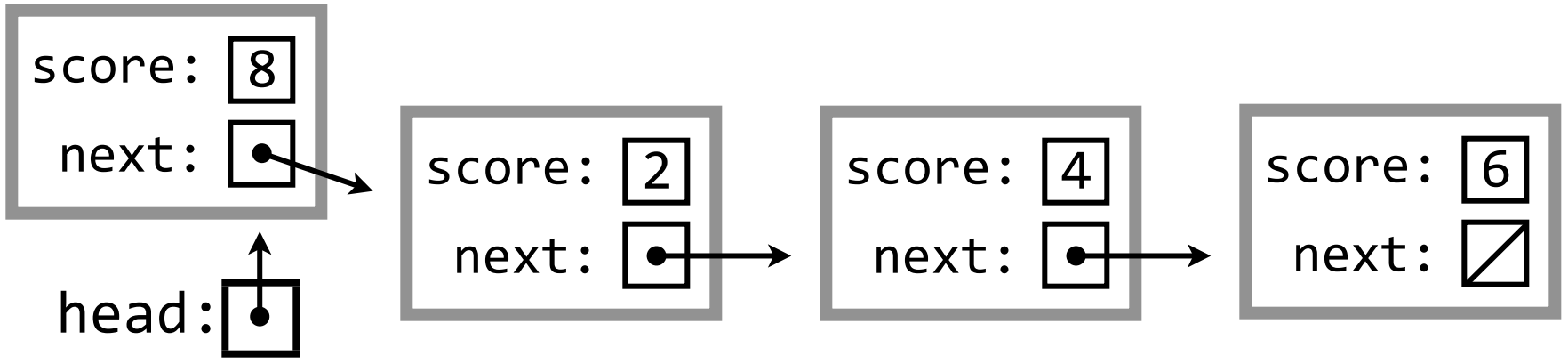
Insert at Beg. of Linked List

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

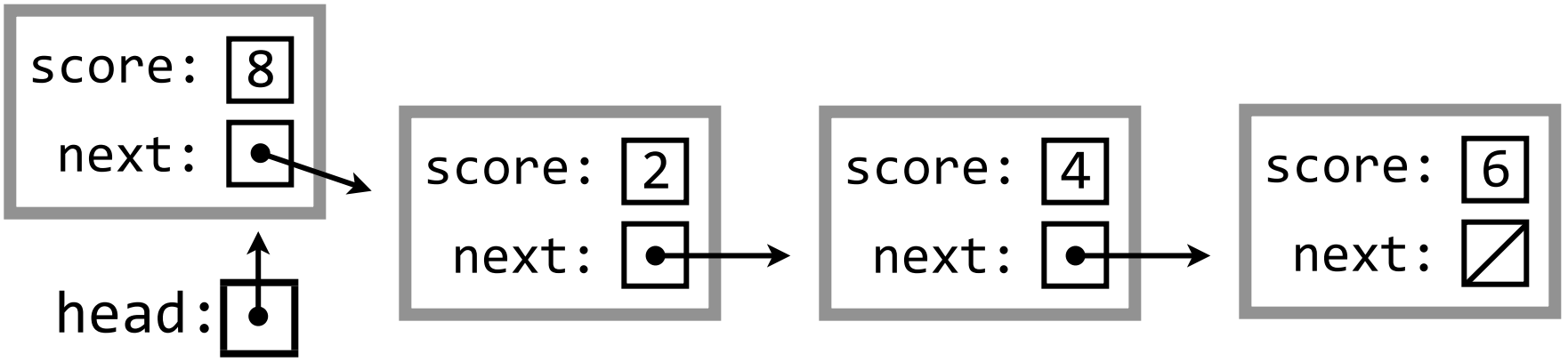


Insert at Beg. of Linked List

```
newNode->next = head;
head = newNode;
```

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

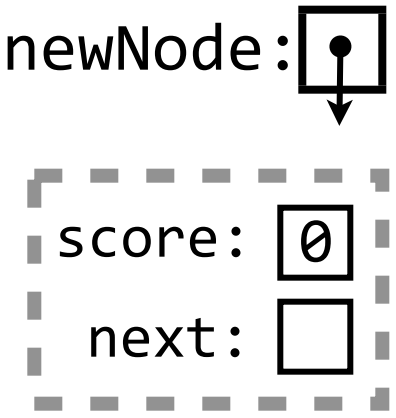


Insert at Beg. of Linked List

```
Node *addToFront(Node *head, Node *newNode)
{
    newNode->next = head;

    return newNode;
}
```

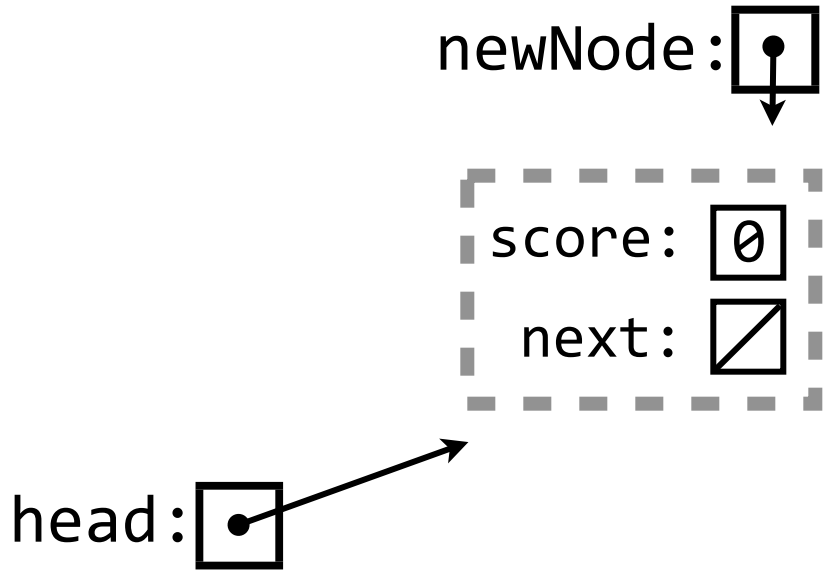
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToFront(head, newNode);
}
```



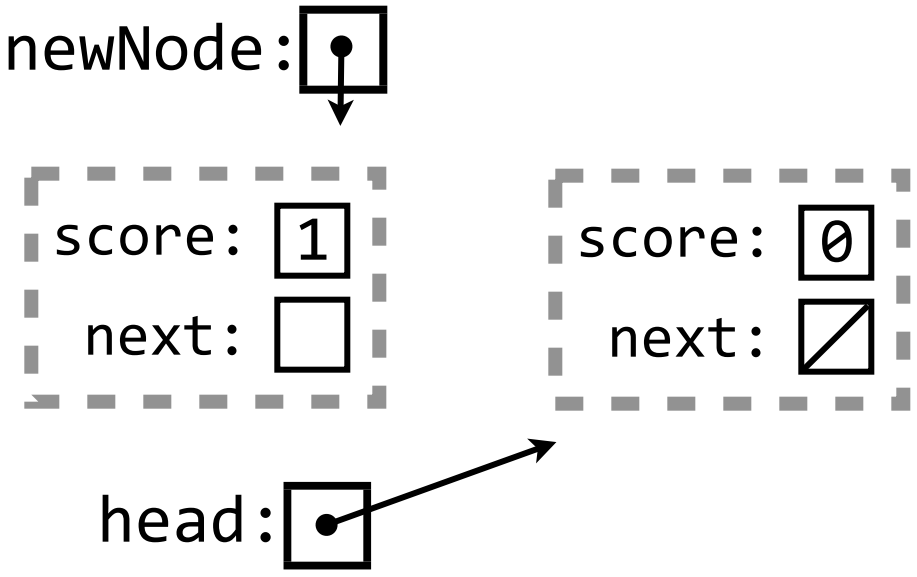
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToFront(head, newNode);
}
```



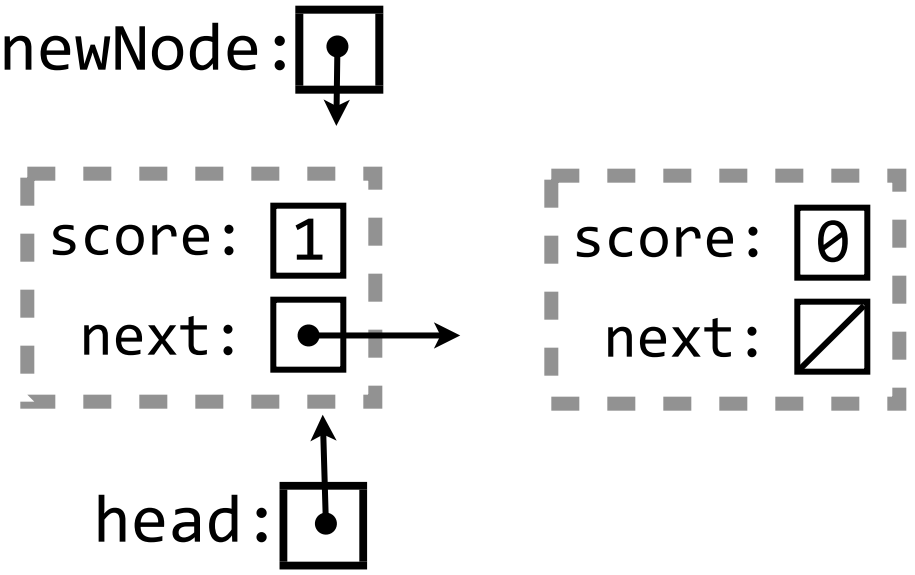
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToFront(head, newNode);
}
```



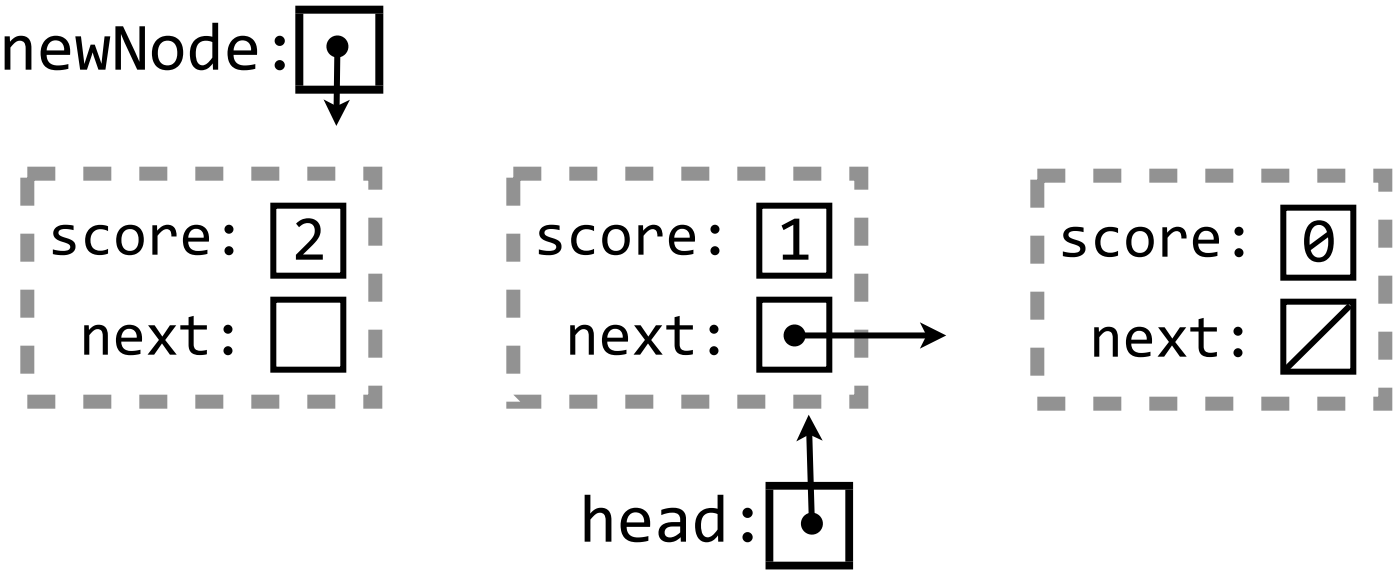
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToFront(head, newNode);
}
```



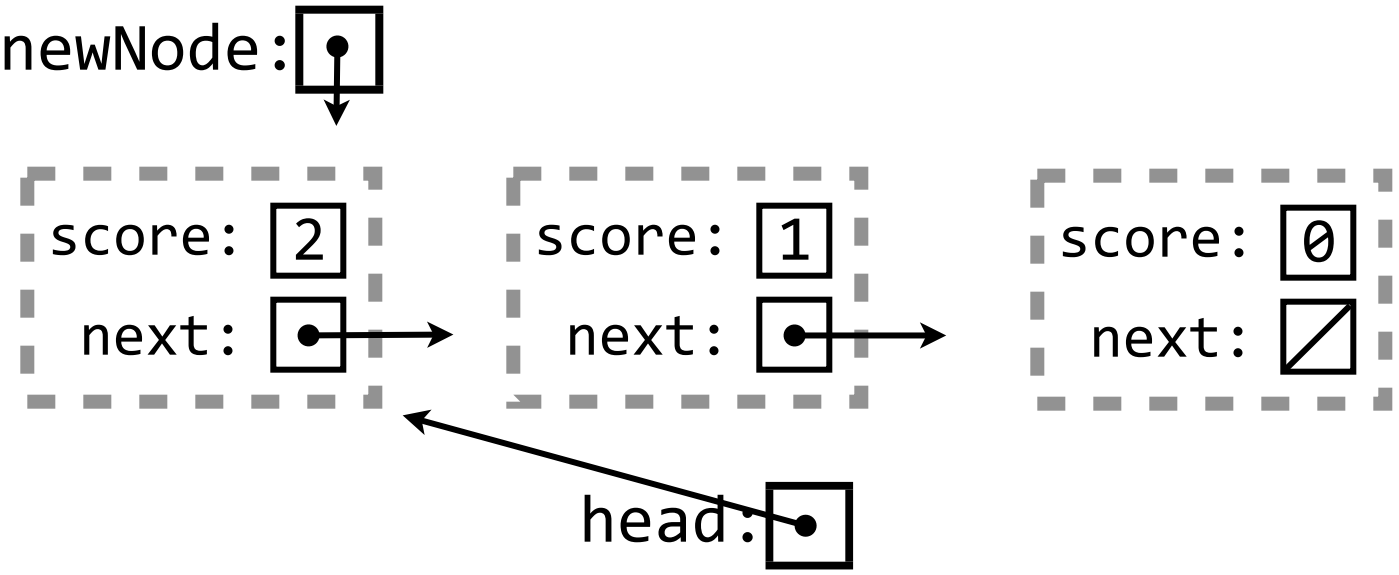
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToFront(head, newNode);
}
```



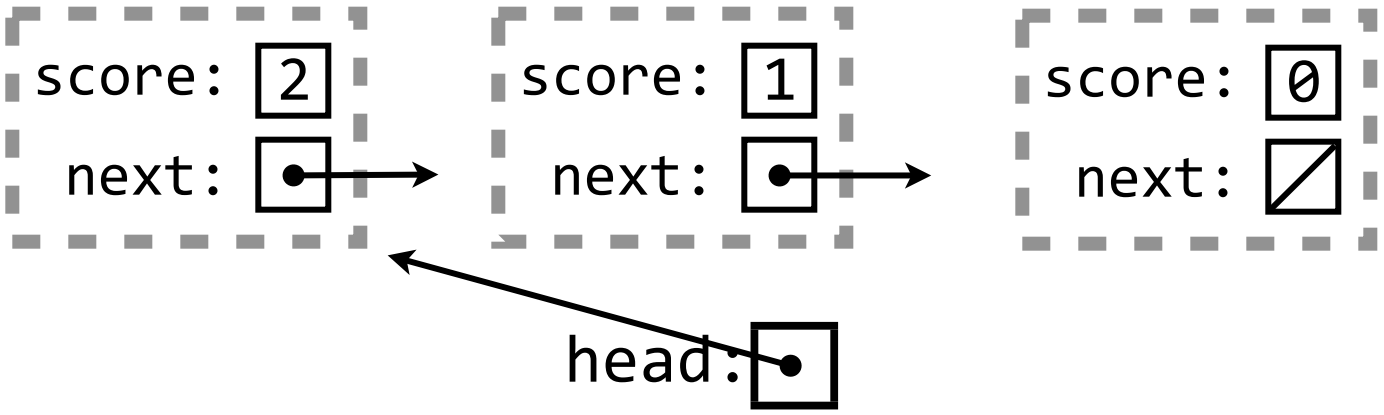
Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToFront(head, newNode);
}
```



Building a List

```
Node *head = NULL;
for (int i = 0; i < N; i++)
{
    Node *newNode = malloc(sizeof(Node));
    newNode->score = i;
    head = addToFront(head, newNode);
}
```

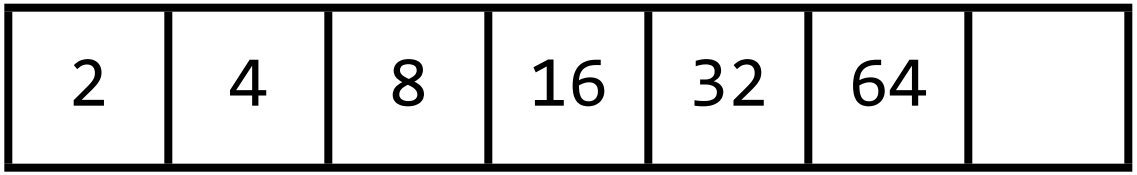


Verdict?

- Array time consuming, if it has space
  - Gets more time consuming as the array gets larger
- Array (much?) harder, if it doesn't
- Linked list never runs out of space
- Linked list is very fast
  - Always takes the same time, no matter the length

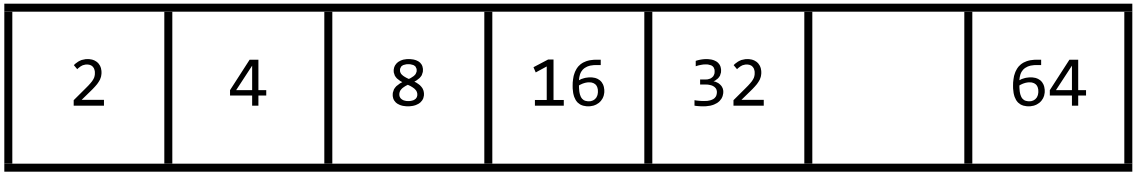
Insert in Middle of Array

Want to insert 1 at a[3]



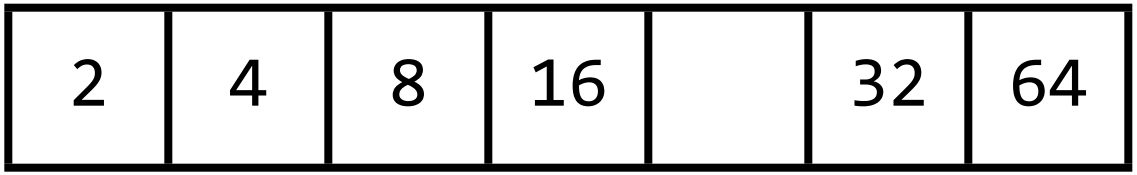
Insert in Middle of Array

Want to insert 1 at a[3]



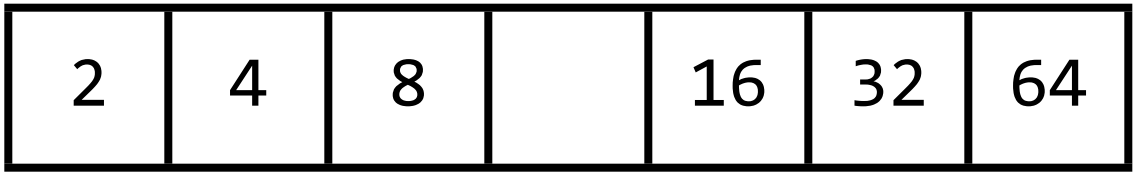
Insert in Middle of Array

Want to insert 1 at a[3]



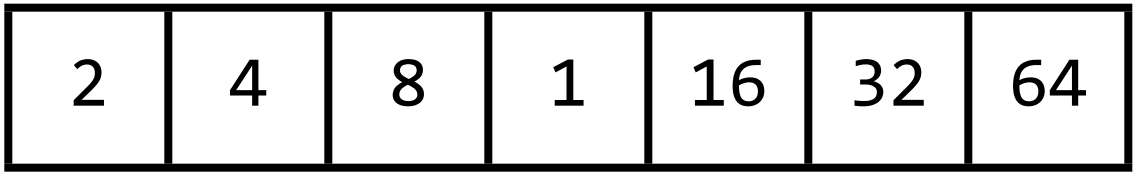
Insert in Middle of Array

Want to insert 1 at a[3]



Insert in Middle of Array

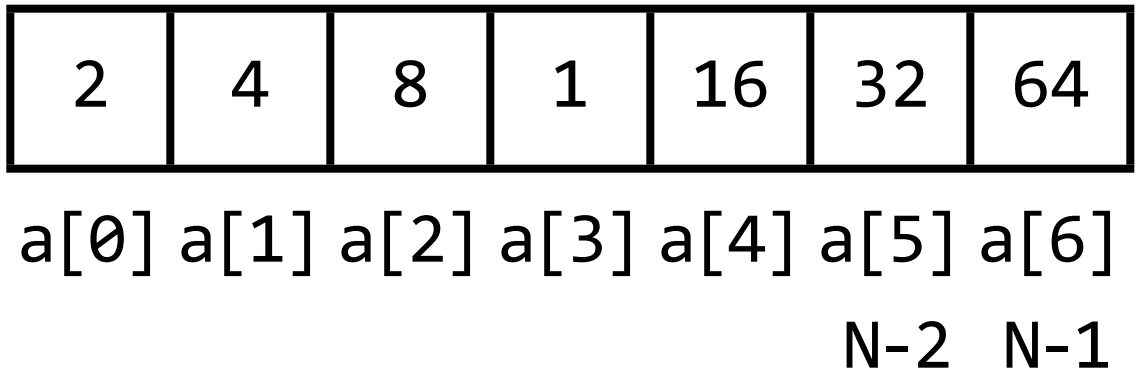
Want to insert 1 at a[3]



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]