Linked Lists 1

Elyse Cornwall

July 31, 2023

Contributions made from previous CS106B Instructors

Stanford University

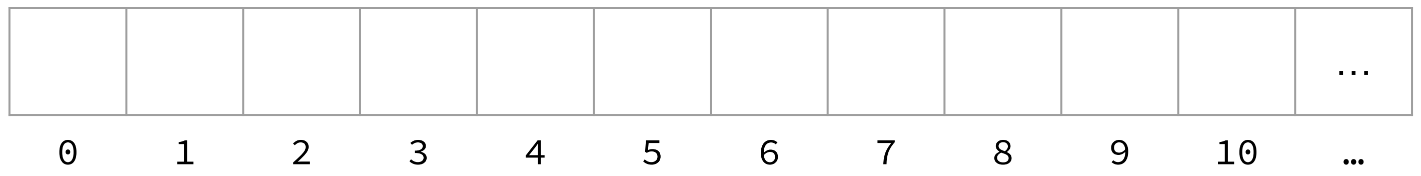
Announcements

- Assignment 3 IGs this week
  - IG attendance is part of section participation grade
- Assignment 4 due this Wednesday at 11:59pm
  - Draws heavily from last week’s lectures
  - Lecture 17 is a conceptual walkthrough of what you’ll be implementing

Recap: Pointers

How is computer memory organized?

- Memory in your computer is just a giant array!
  - Can think of it as a long row of boxes, with each box having a value in it and an associated index



- How can we communicate with the computer to find exactly which box we want to access/store information in?
  - We'll give each box an associated numerical location, called a **memory address**

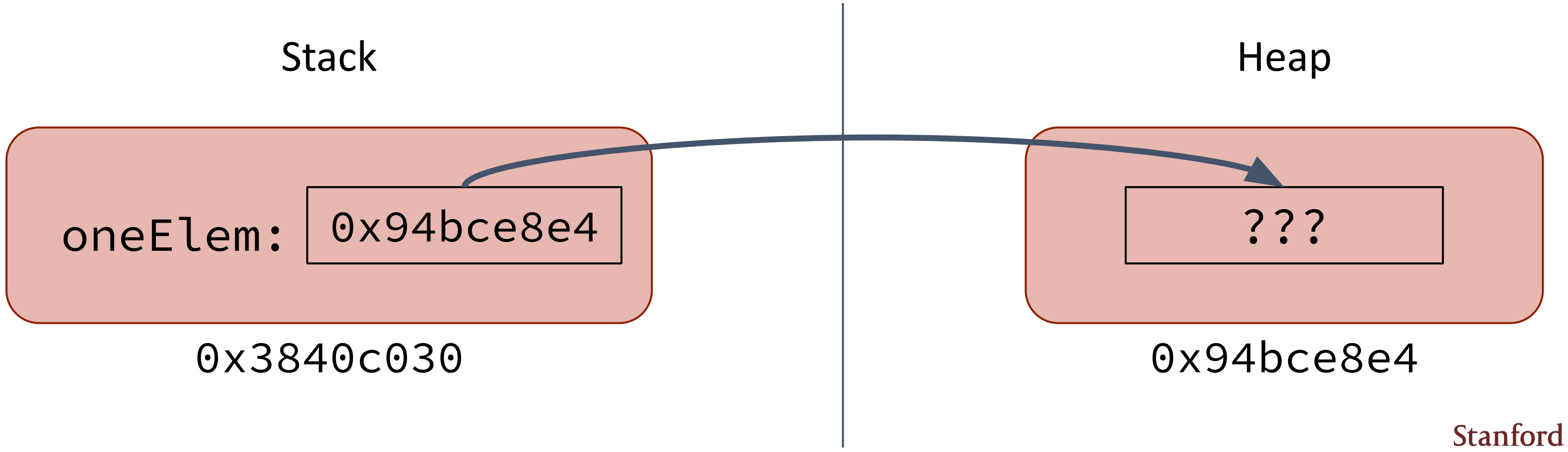
What is a pointer?

A memory address!!

Pointer Syntax

- Pointers are necessary to store the value generated by the new keyword (which is just a memory address on the heap)

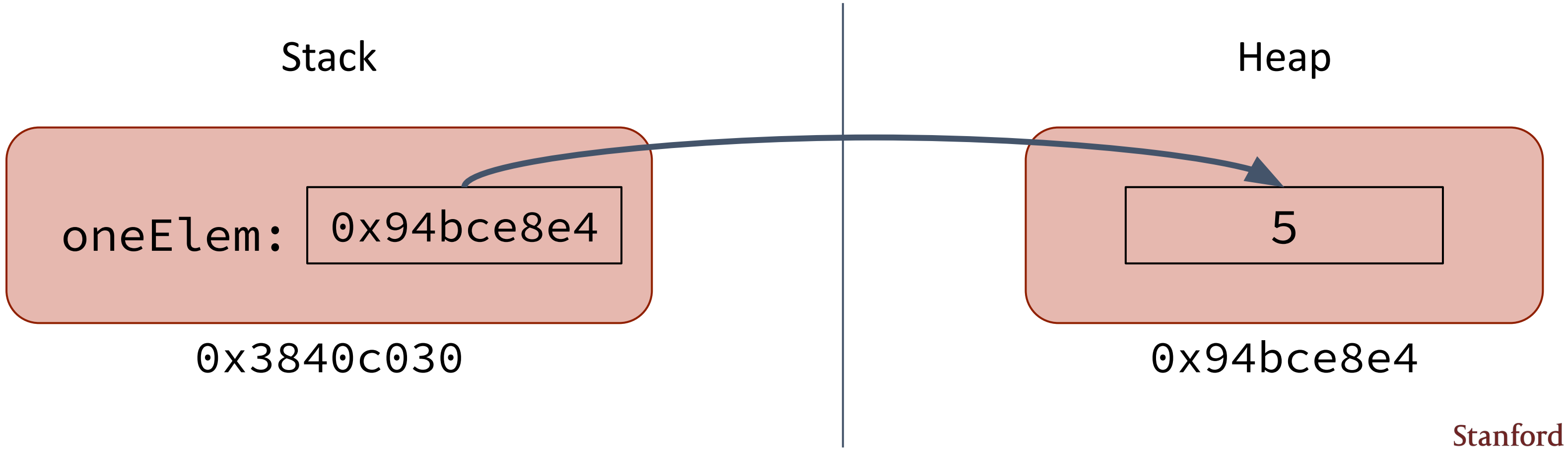
```
int* oneElem = new int;
```



Pointer Syntax

- To read or modify the variable that a pointer points to, we use the `*` (asterisk) operator (in a different way than before!)
- Known as **dereferencing the pointer**
- Follow the arrow to the memory location

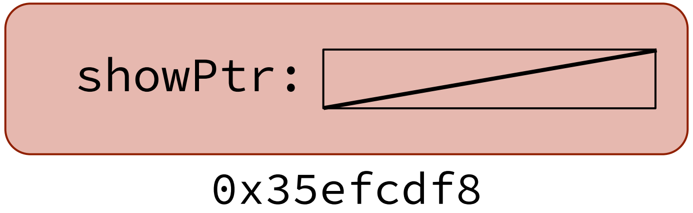
```
*oneElem = 5;
```



`nullptr`

- When we declare/initialize a pointer but don’t have anything to point it at yet, that can be dangerous and unpredictable
- To ensure that we can tell if a pointer has a valid address or not, set your declared pointer to `nullptr`, which means "no valid address"

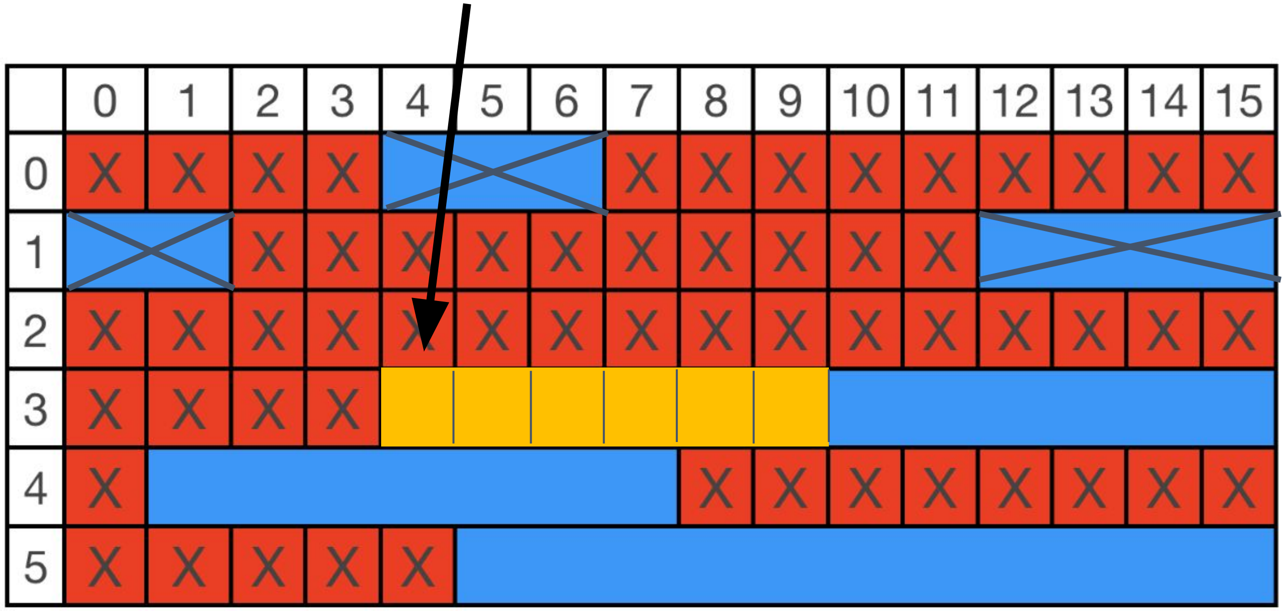
```
string* showPtr = nullptr;
```



Under the Hood

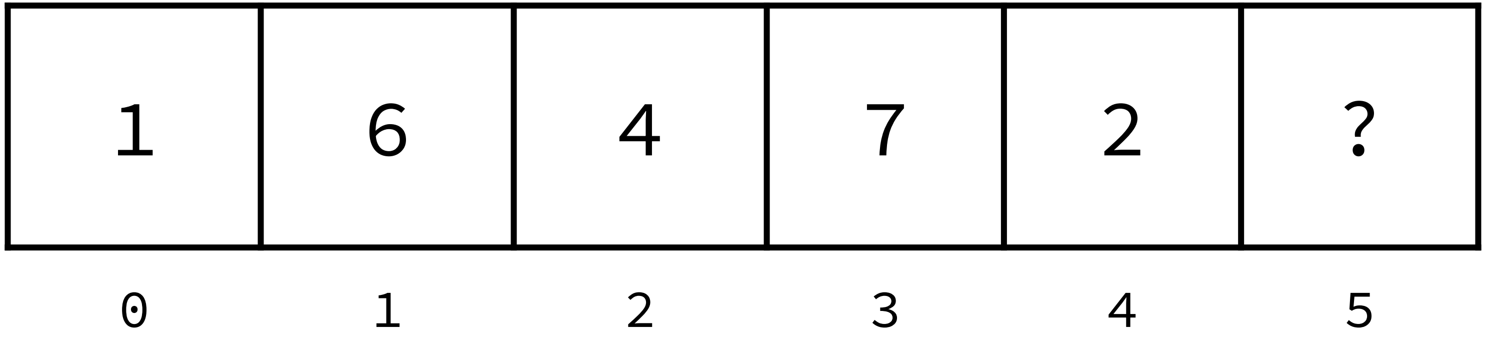
Arrays are ***contiguous*** chunks of space in the computer's memory

```
int* sixInts = new int[6];
```



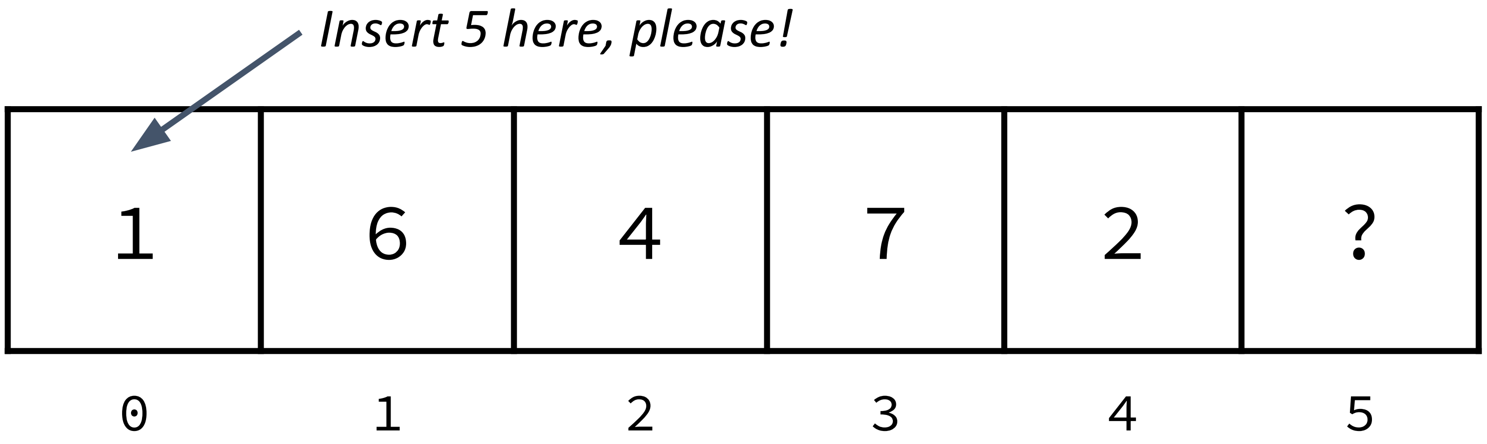
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



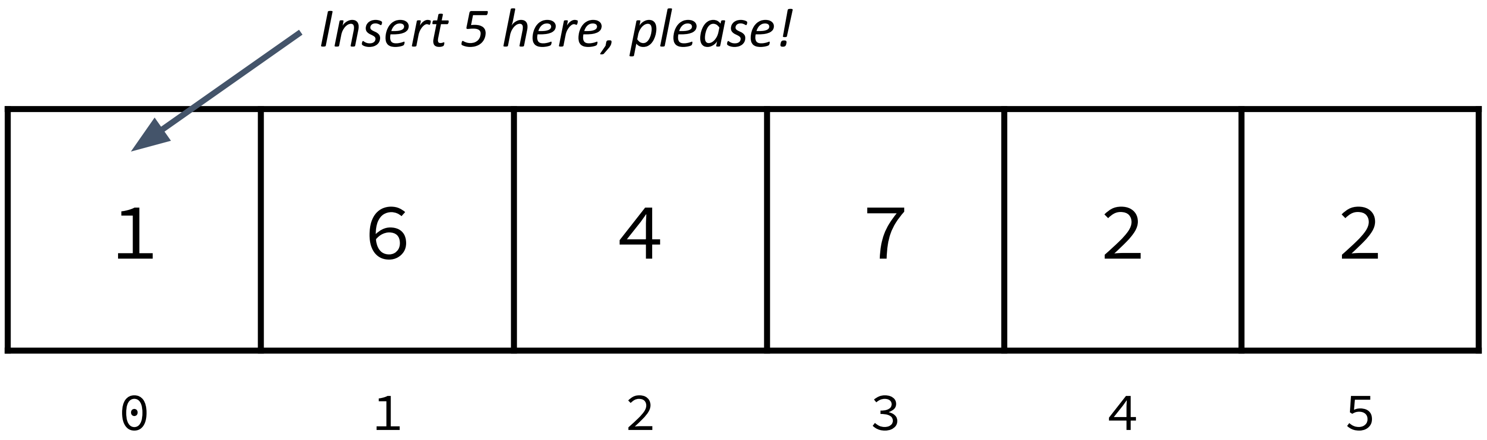
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



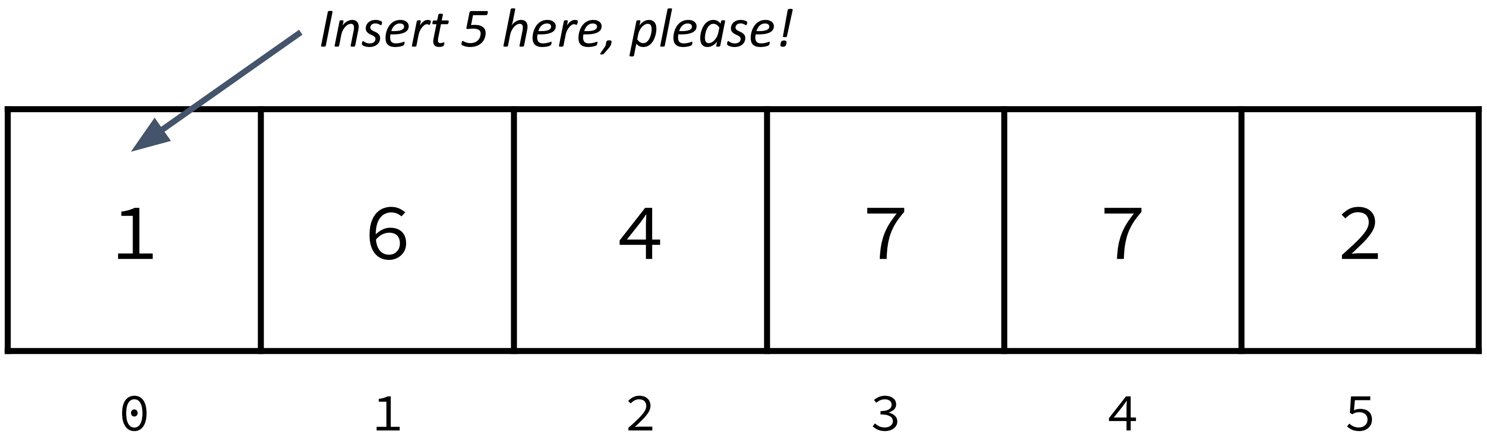
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



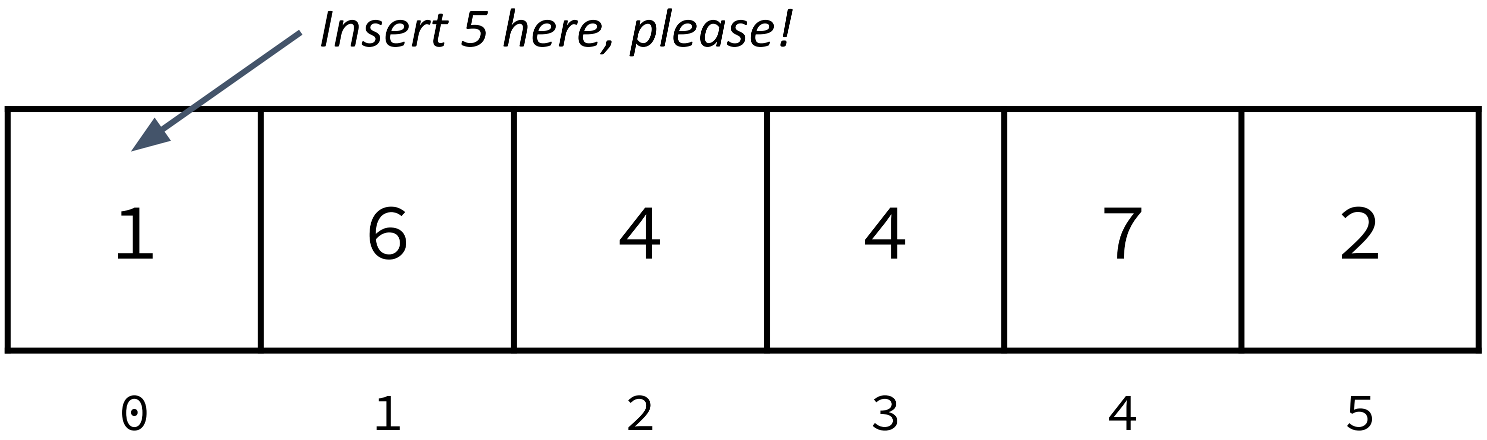
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



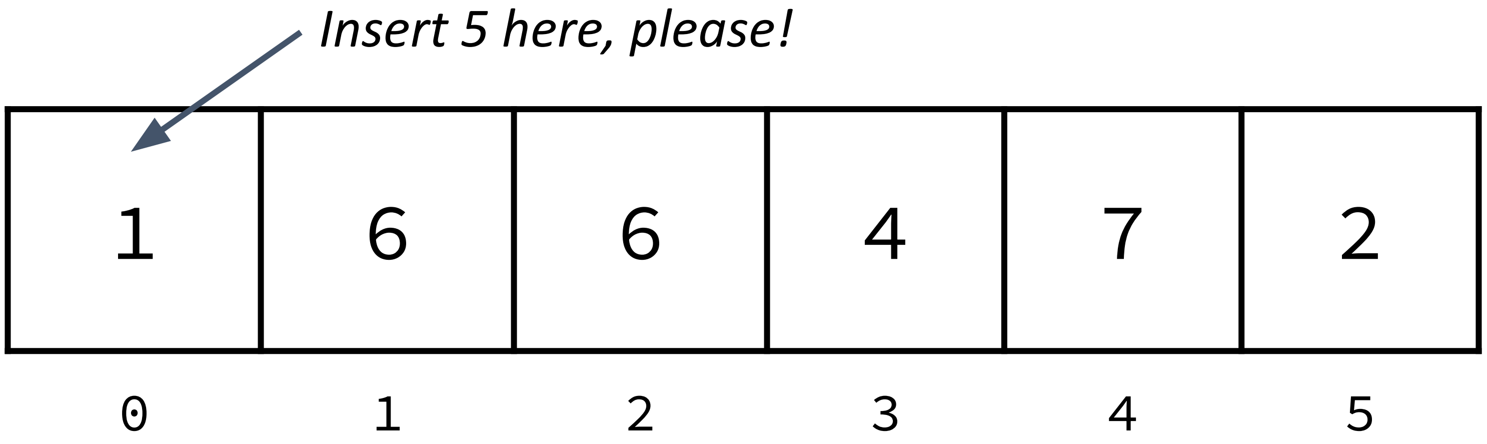
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



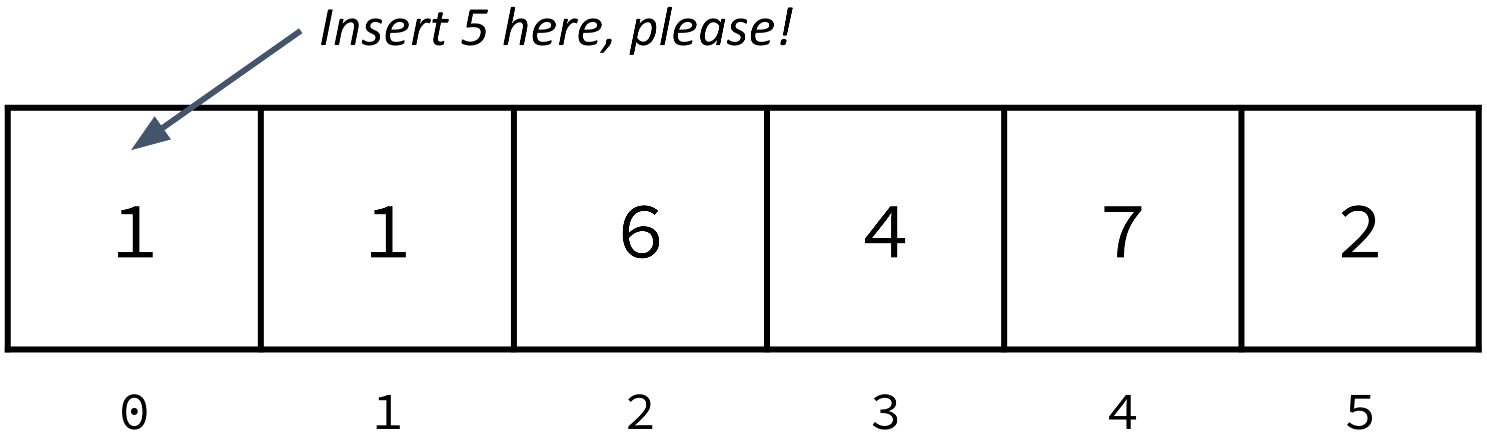
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



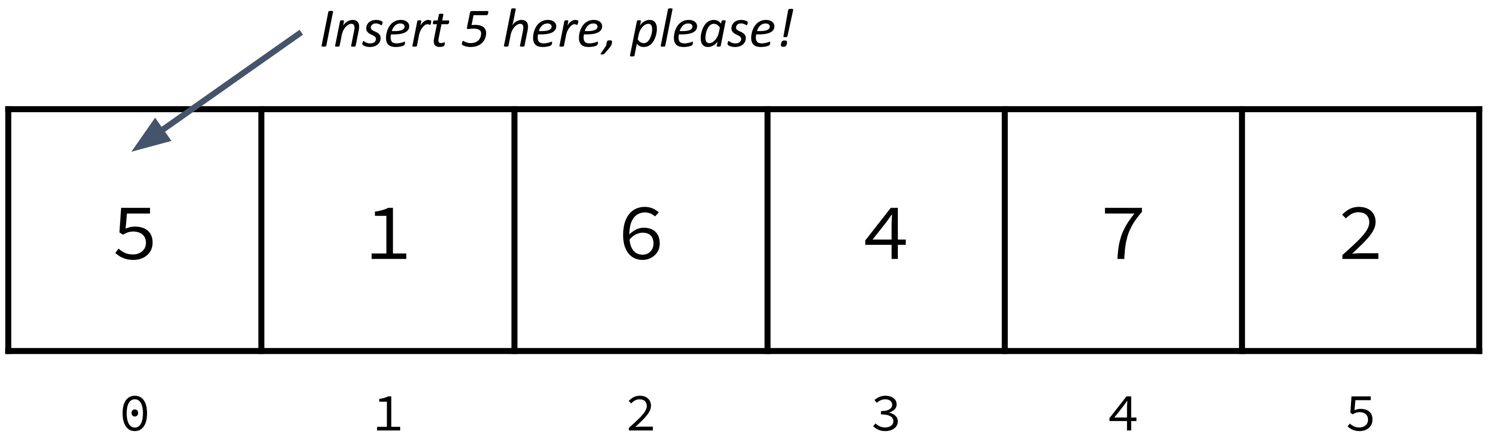
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



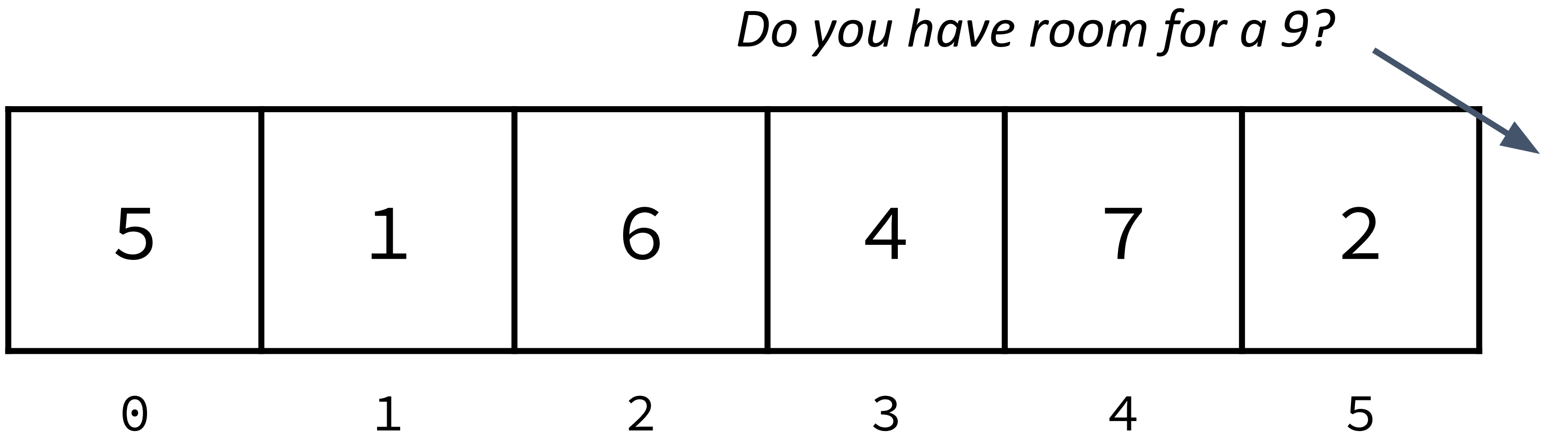
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



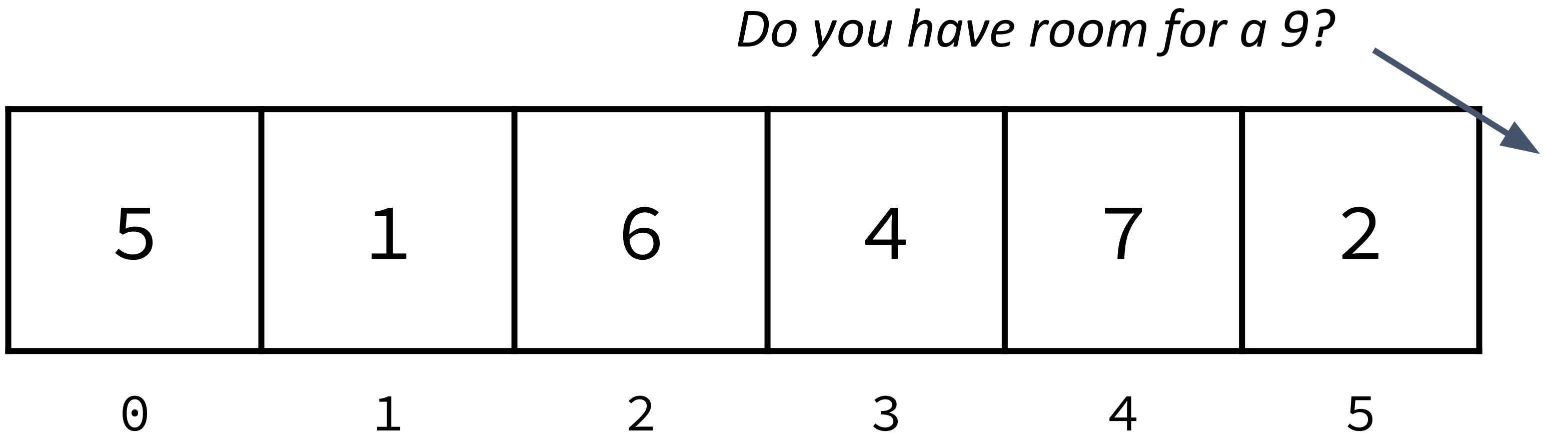
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



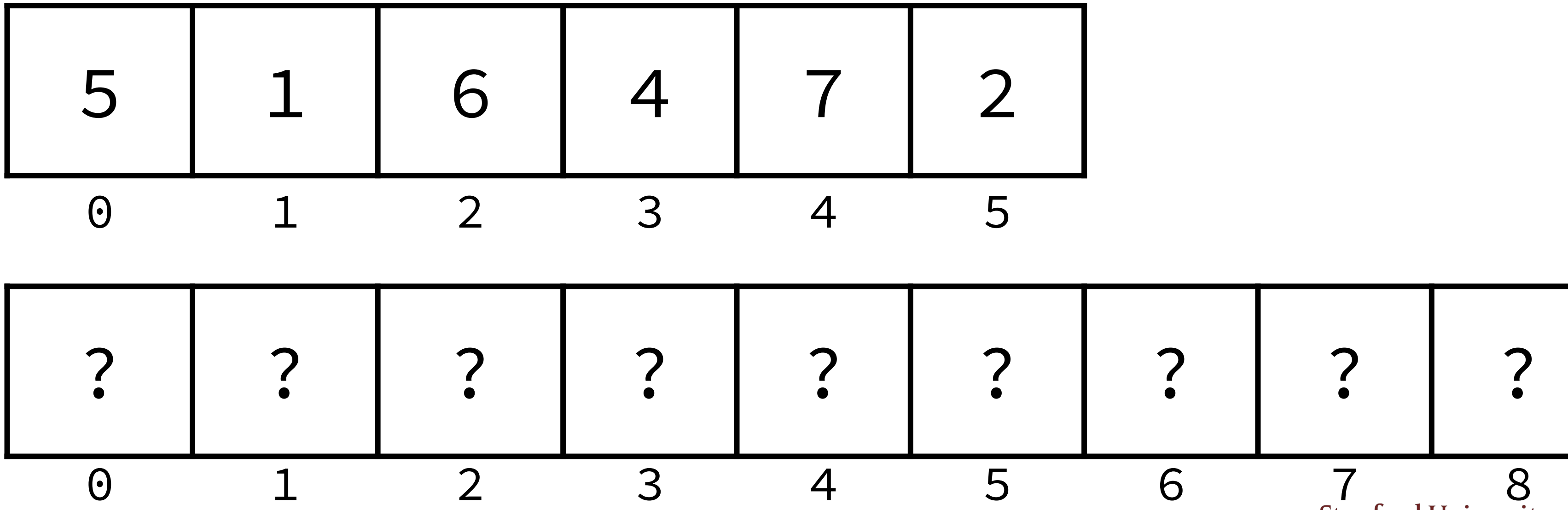
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



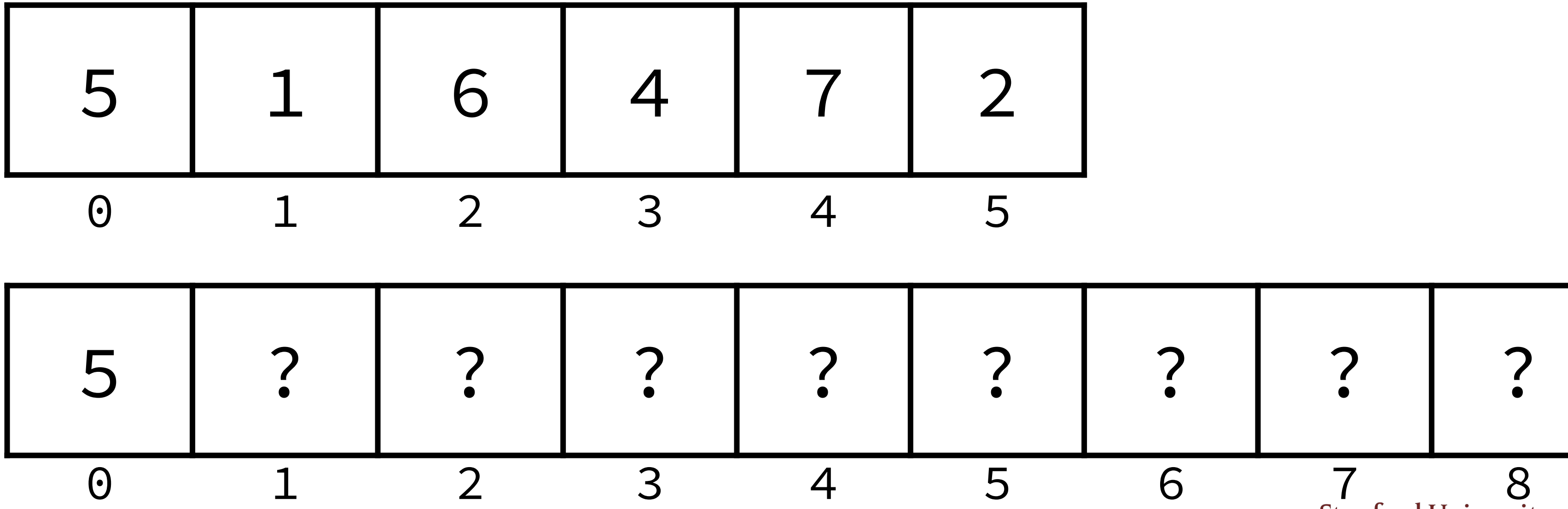
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



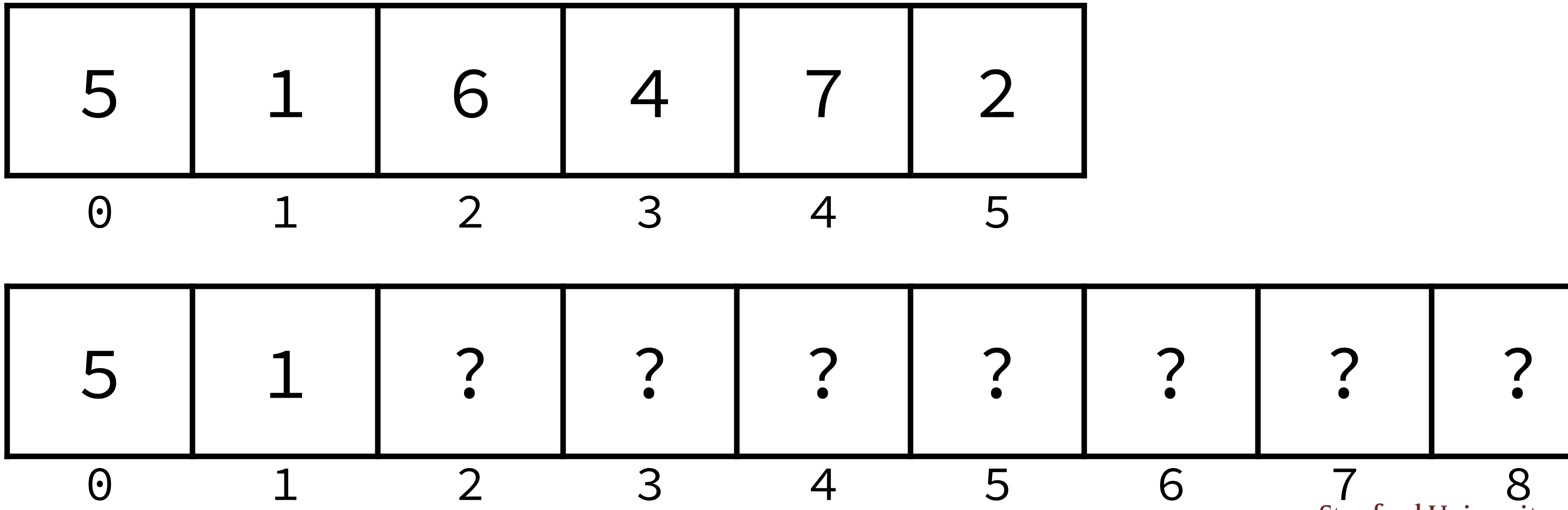
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



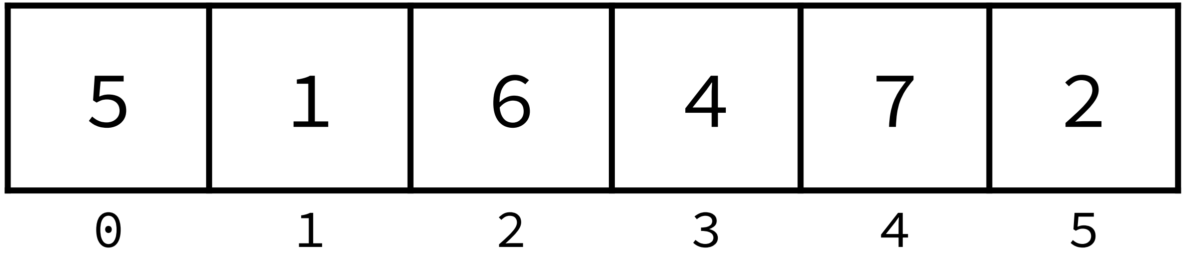
Frustrations with Arrays

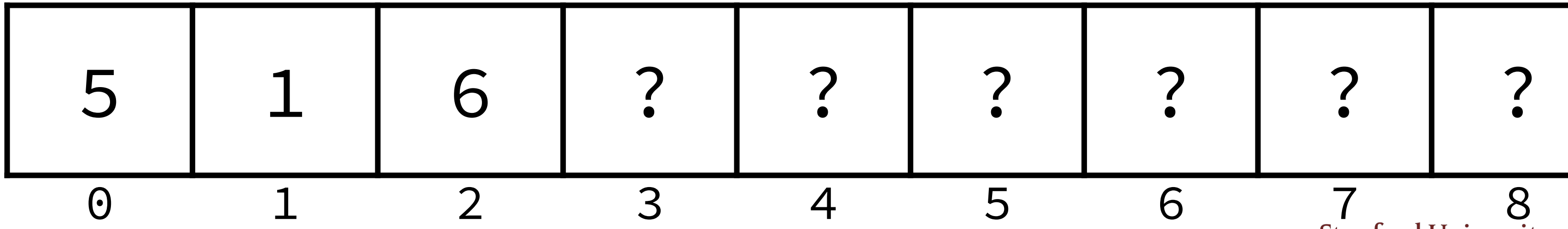
- Not easily resizable
- Not efficient to insert elements at the beginning



Frustrations with Arrays

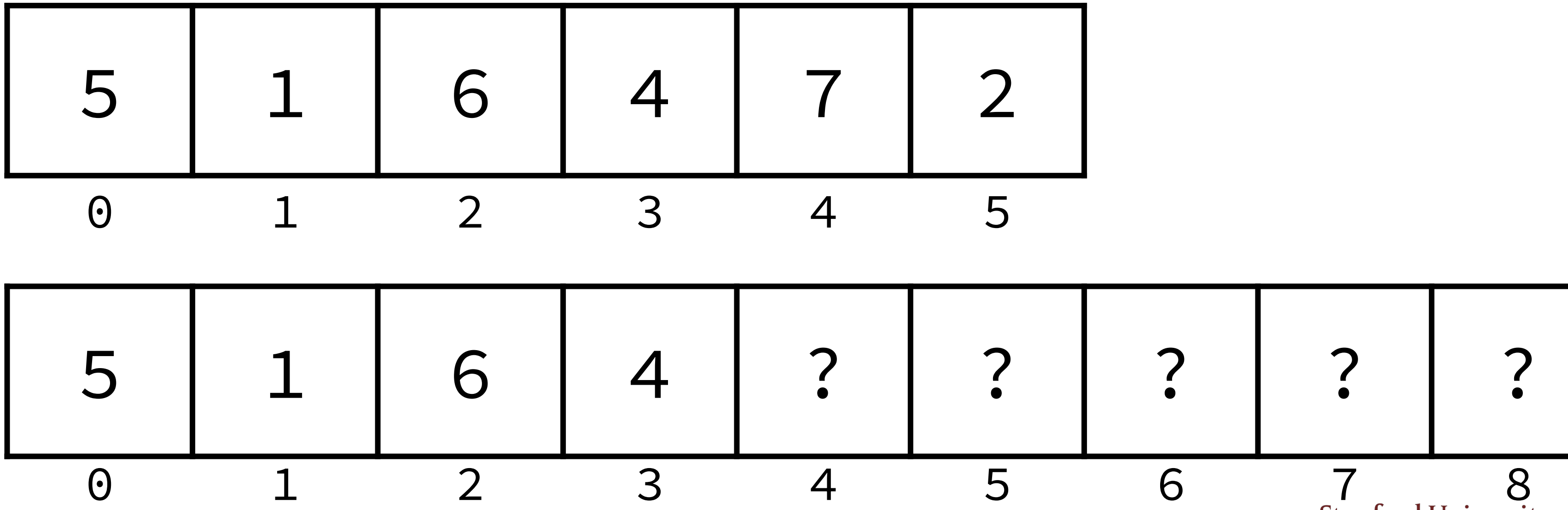
- Not easily resizable
- Not efficient to insert elements at the beginning





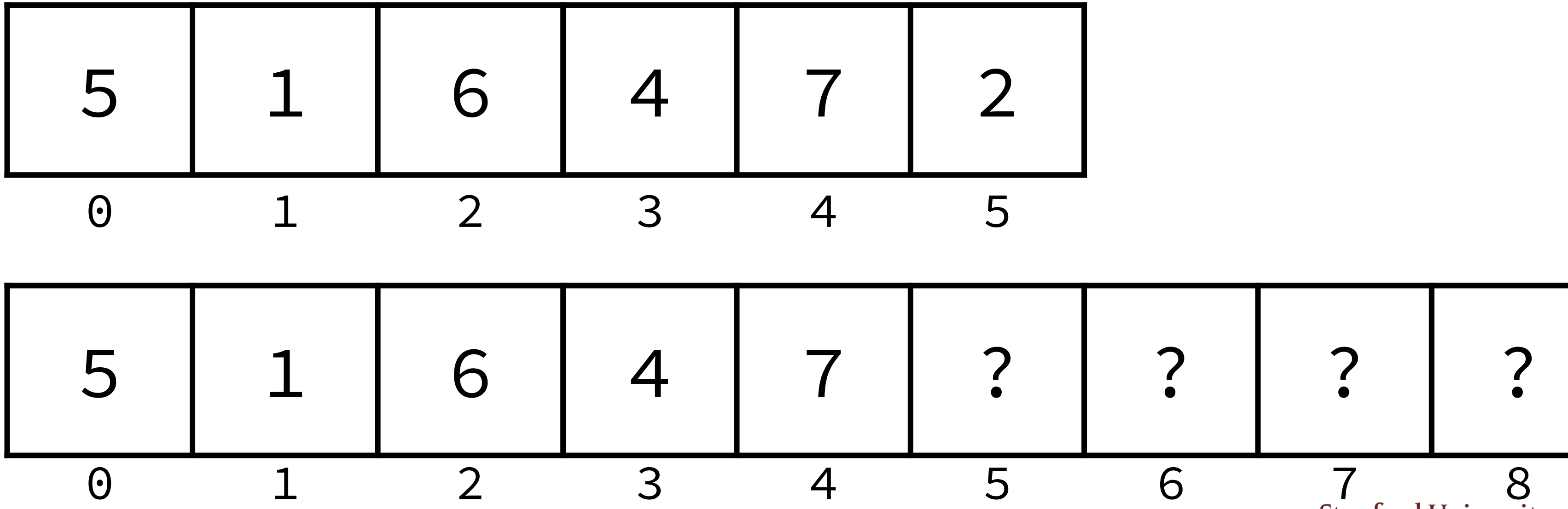
Frustrations with Arrays

- Not easily resizable
- Not efficient to insert elements at the beginning



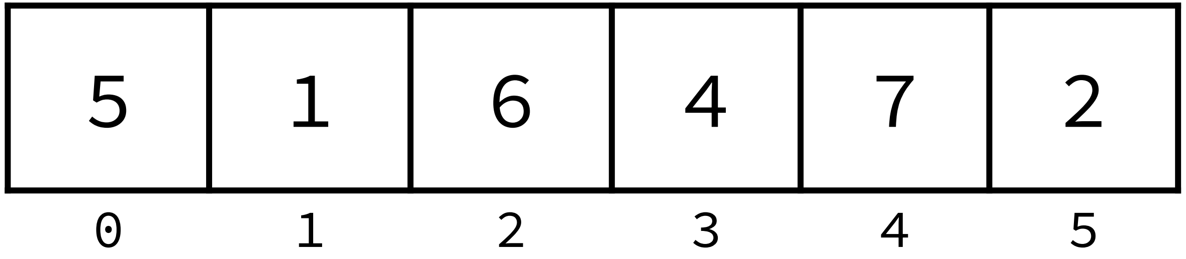
Frustrations with Arrays

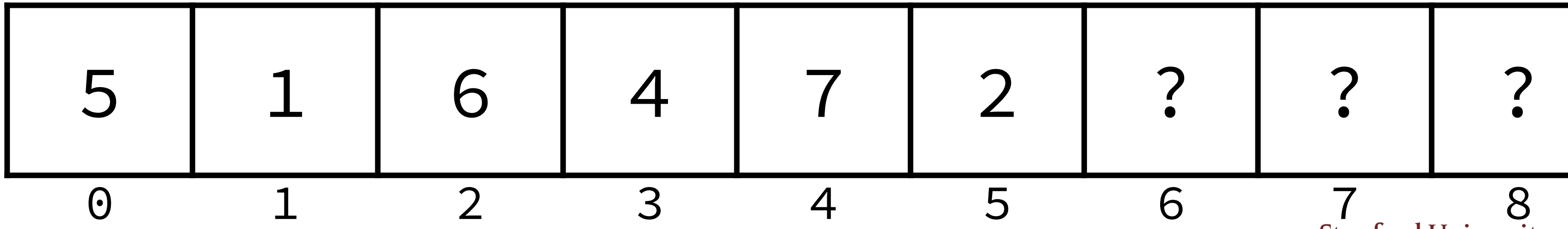
- Not easily resizable
- Not efficient to insert elements at the beginning



Frustrations with Arrays

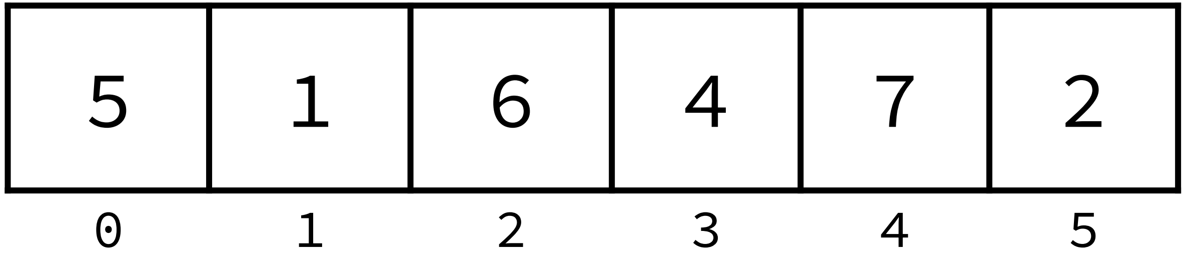
- Not easily resizable
- Not efficient to insert elements at the beginning

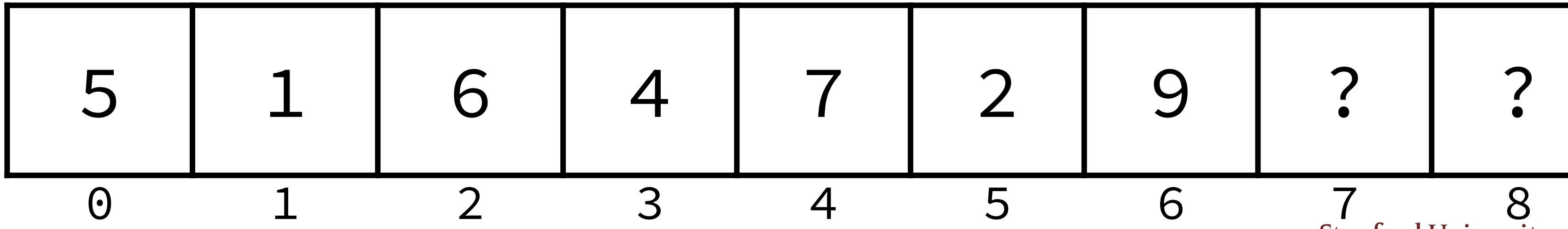




Frustrations with Arrays

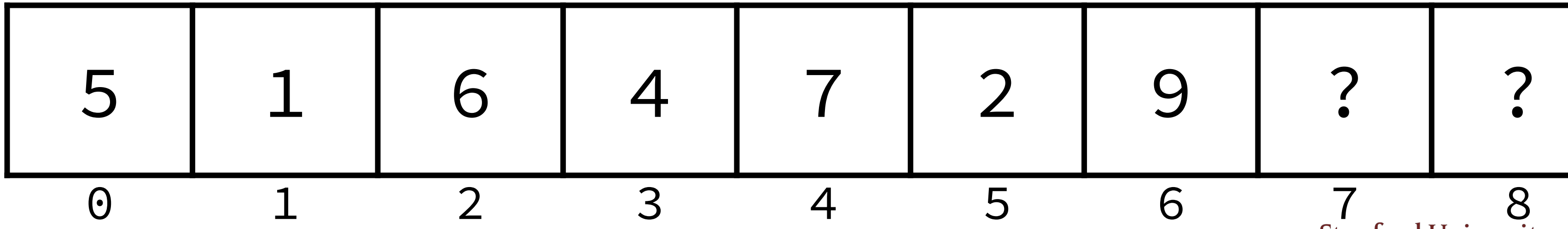
- Not easily resizable
- Not efficient to insert elements at the beginning





Frustrations with Arrays

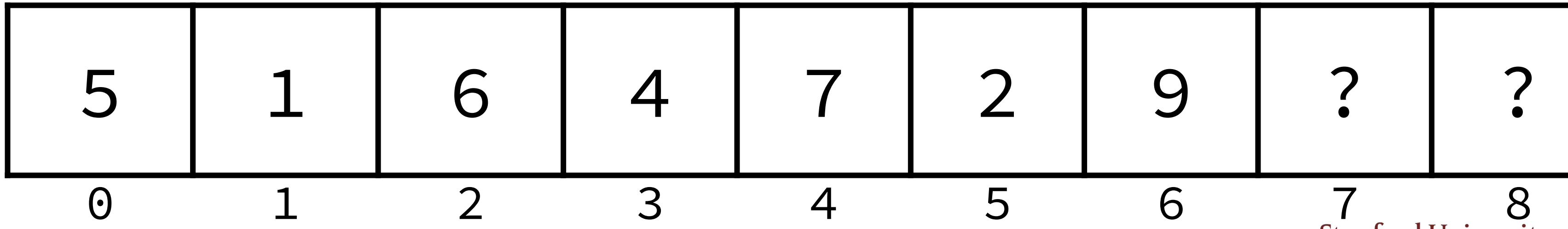
- Not easily resizable
- Not efficient to insert elements at the beginning



Frustrations with Arrays

Can we do better?

- Not easily resizable
- Not efficient to insert elements at the beginning



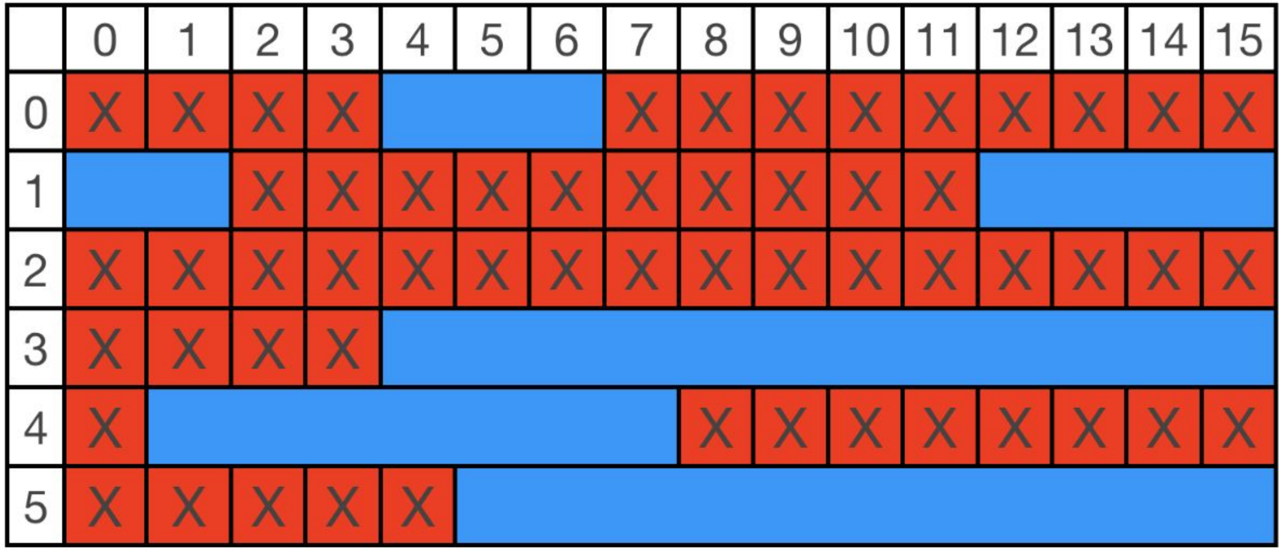
Linked Lists

What are Linked Lists?

- A way we can use pointers to organize non-contiguous memory on the heap

What are Linked Lists?

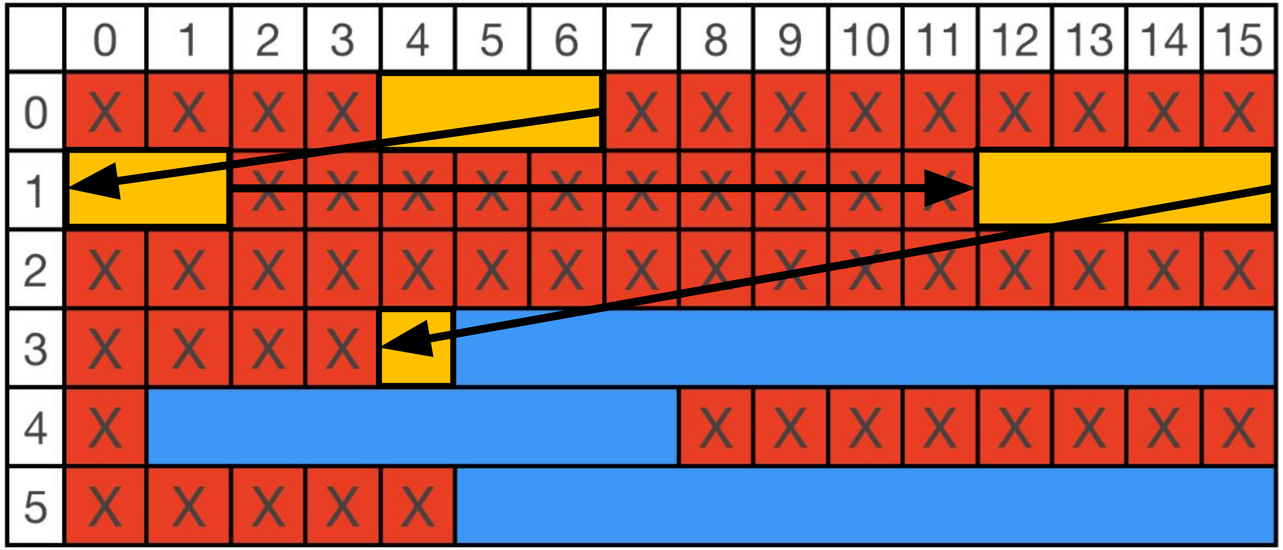
- A way we can use pointers to organize non-contiguous memory on the heap



What are Linked Lists?

- A way we can use pointers to organize non-contiguous memory on the heap

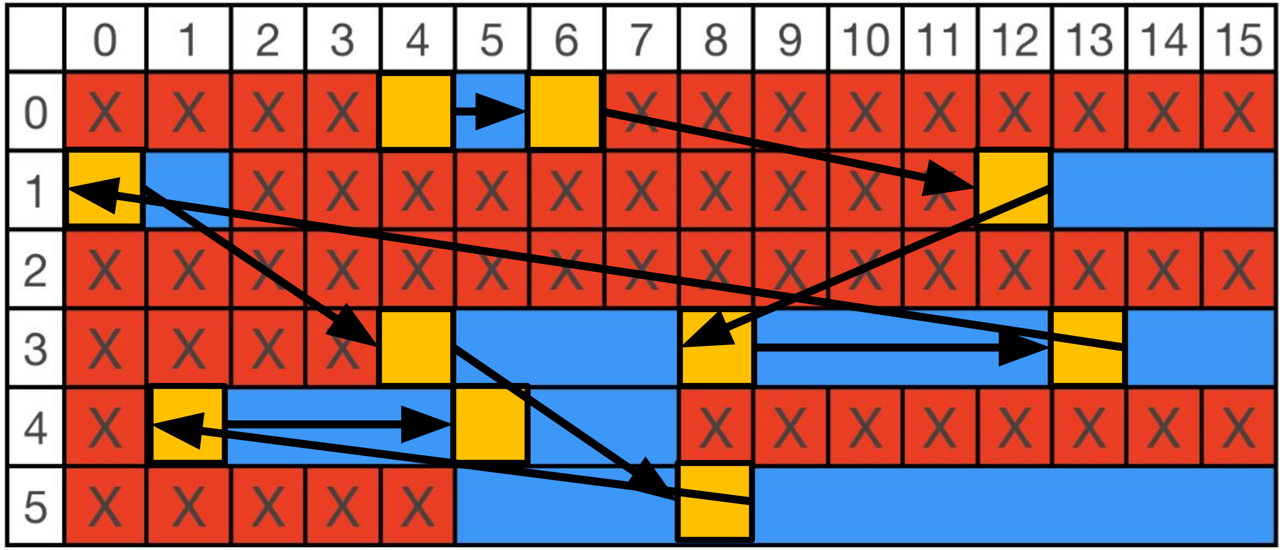
Could we store 10 integers like this?



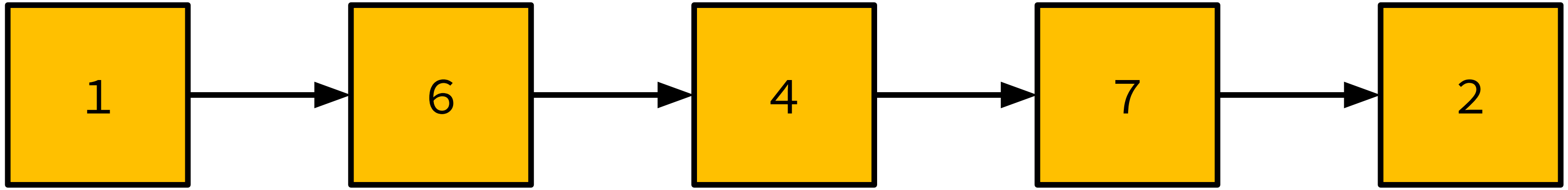
What are Linked Lists?

- A way we can use pointers to organize non-contiguous memory on the heap

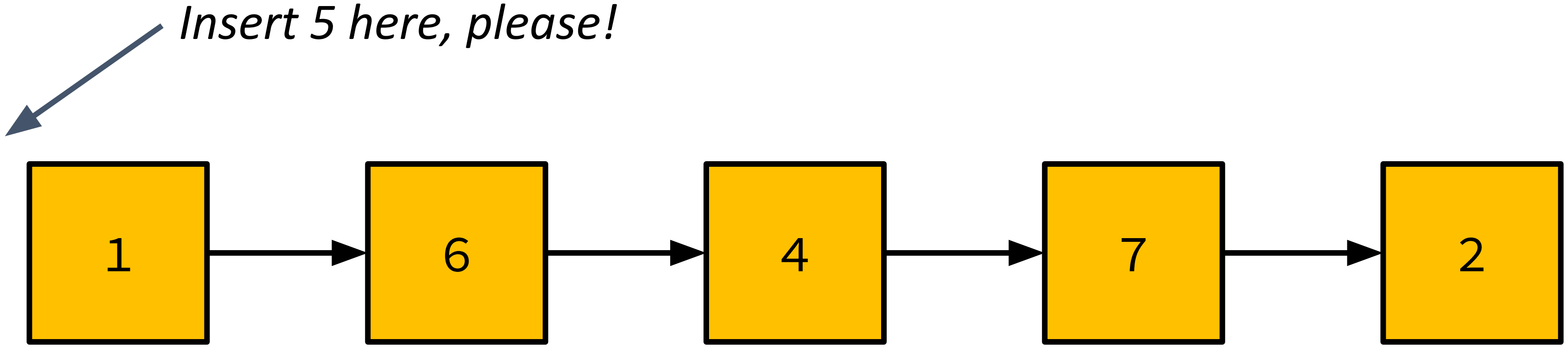
Or this?



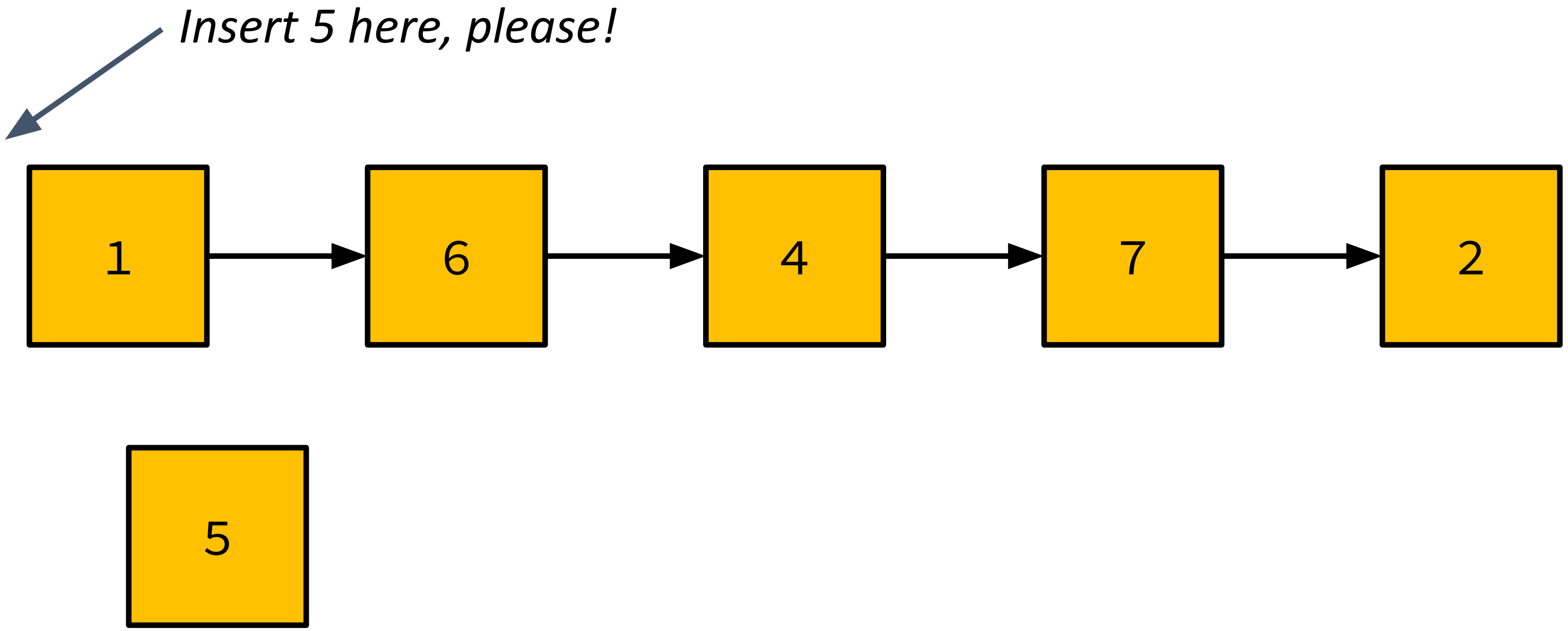
Benefits of Linked Lists



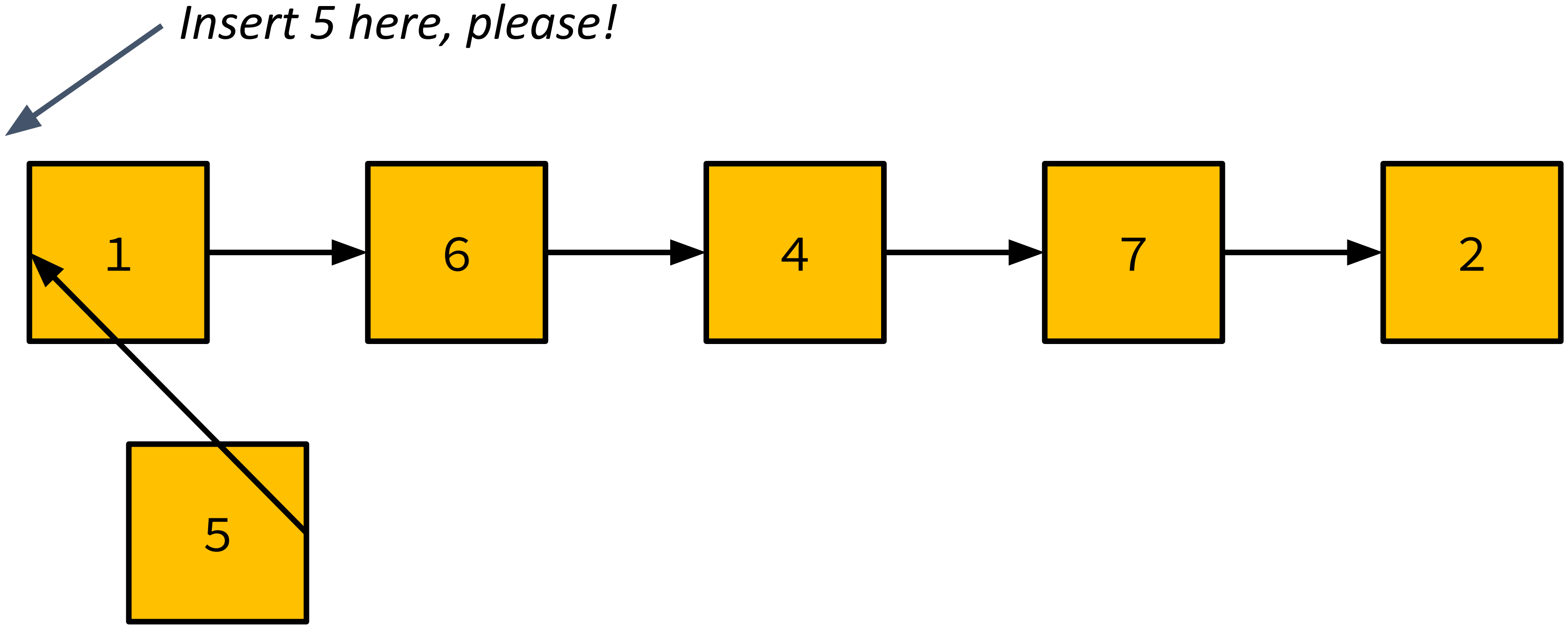
Benefits of Linked Lists



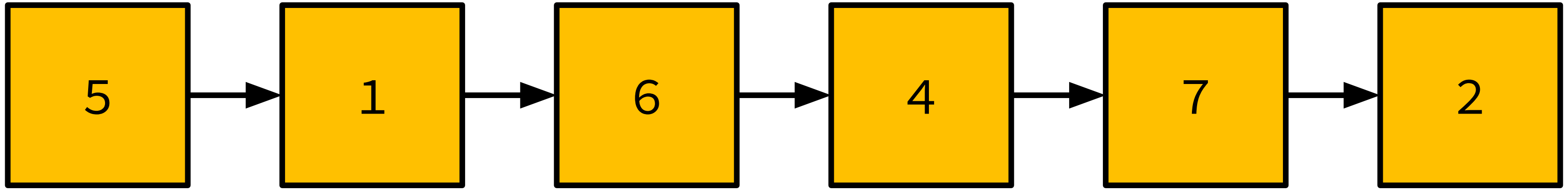
Benefits of Linked Lists



Benefits of Linked Lists

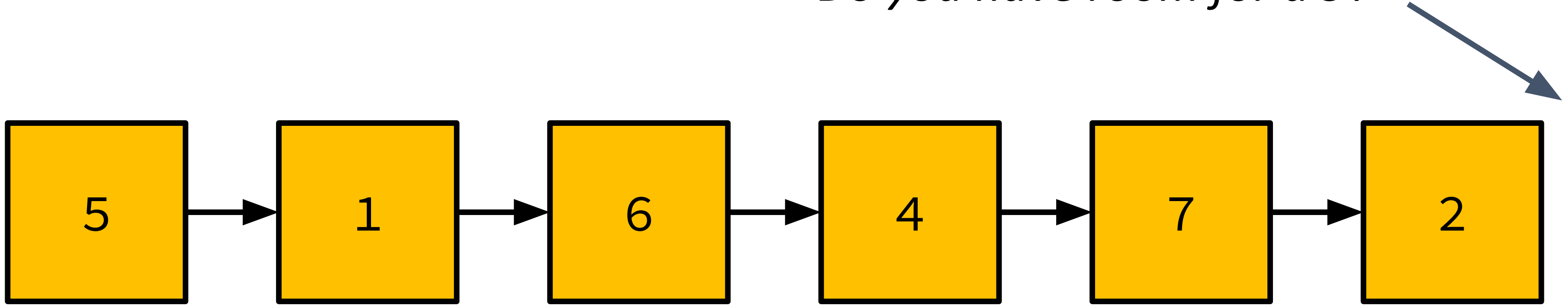


Benefits of Linked Lists

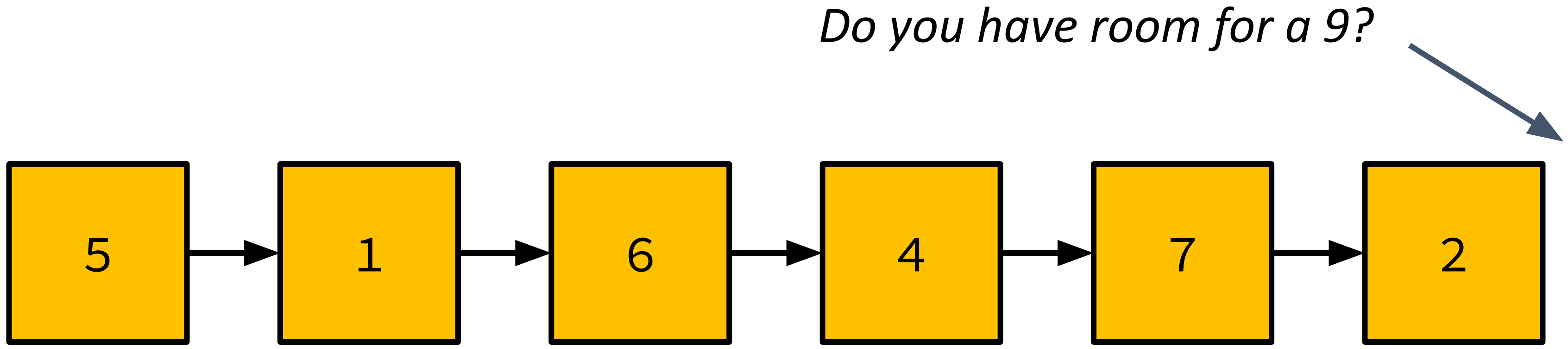


Benefits of Linked Lists

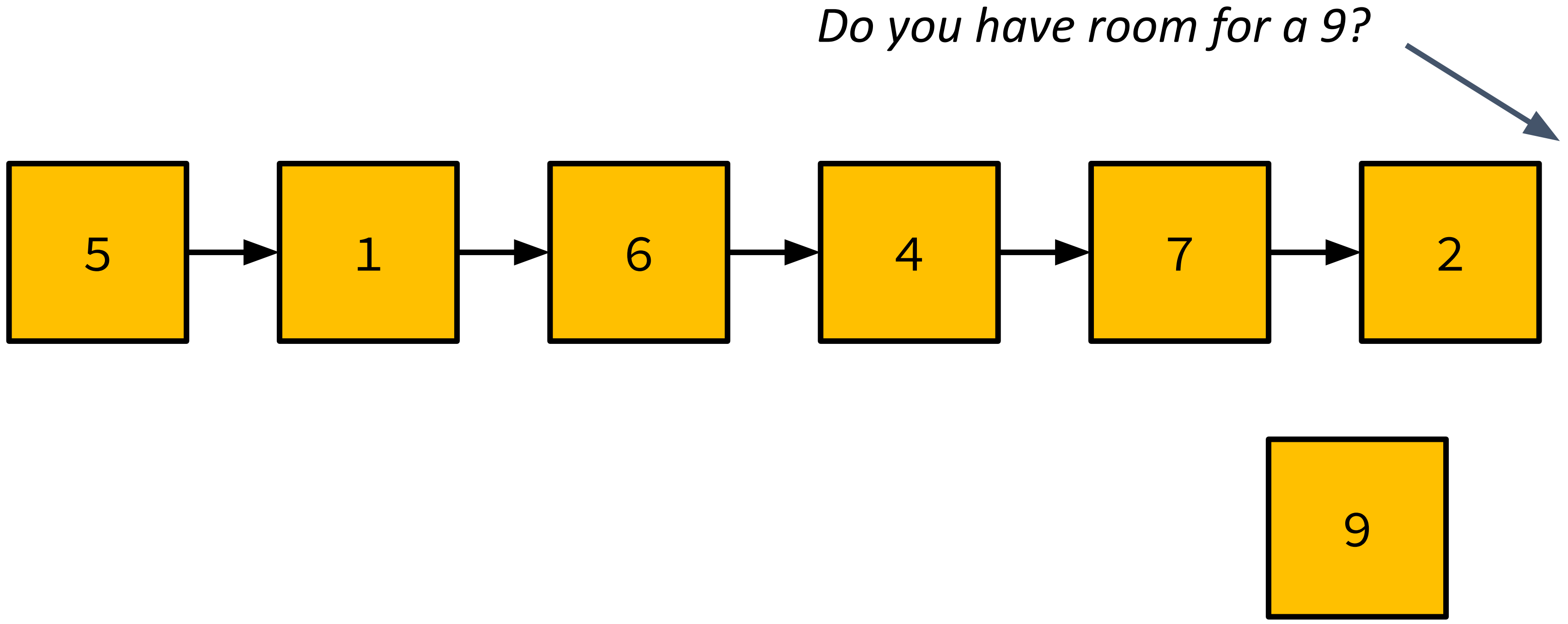
Do you have room for a 9?



Benefits of Linked Lists

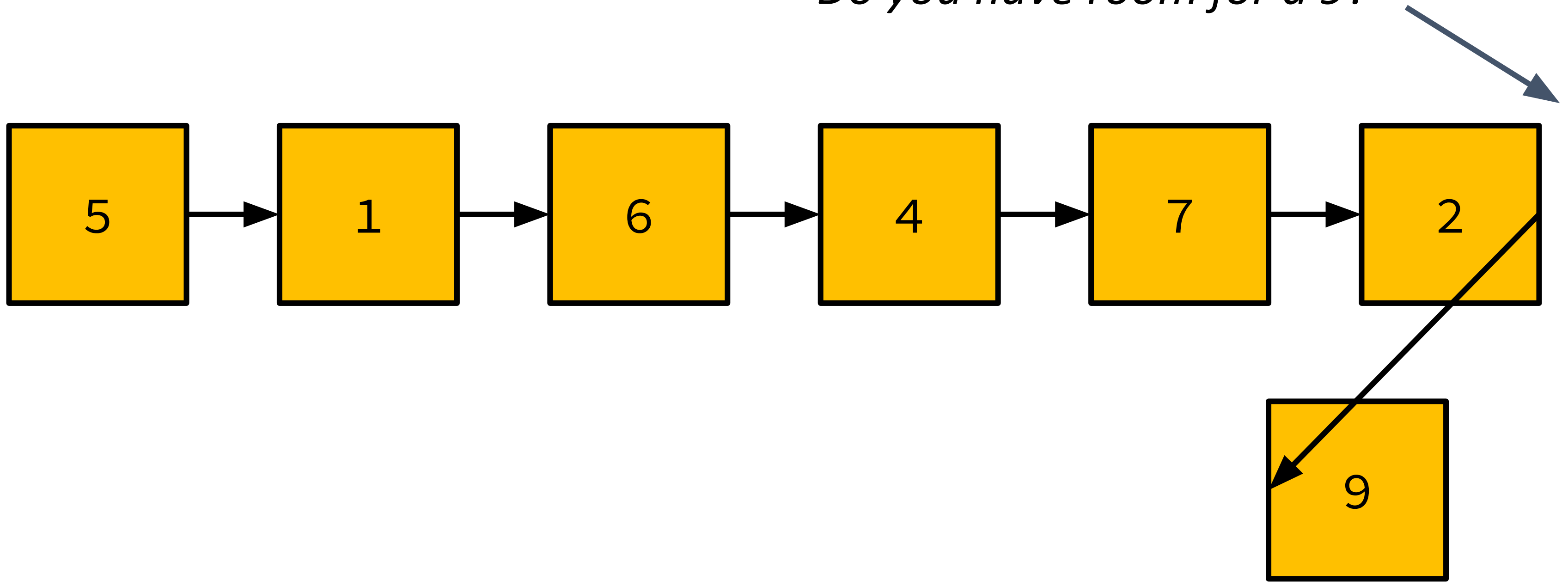


Benefits of Linked Lists



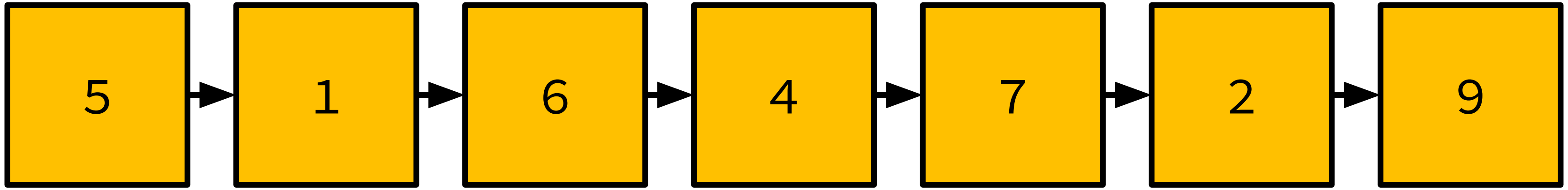
Benefits of Linked Lists

Do you have room for a 9?



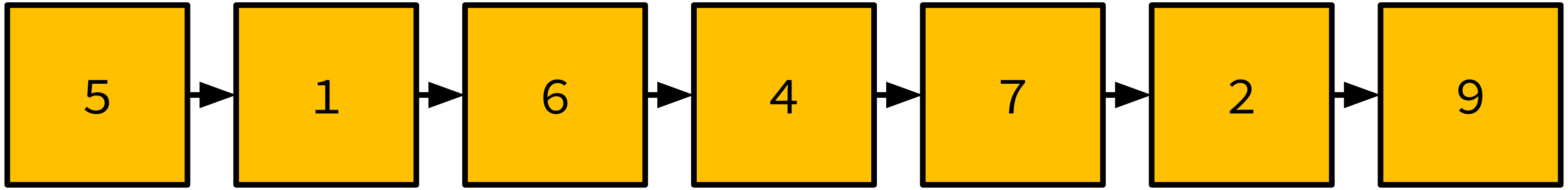
Benefits of Linked Lists

- Easily resizable
- Efficient to insert elements at the beginning



Benefits of Linked Lists

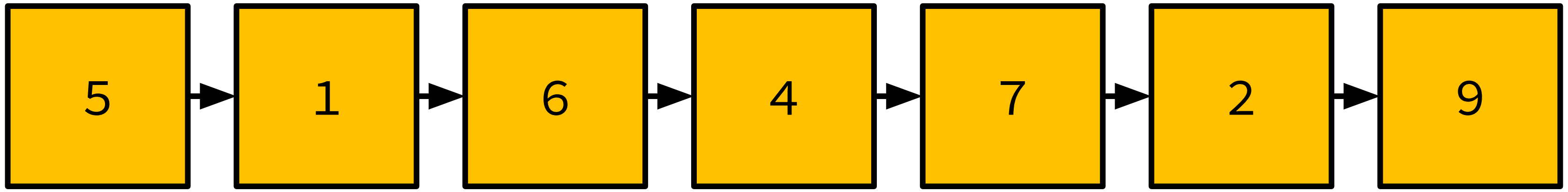
- Easily resizable
- Efficient to insert elements at the beginning



Okay, but what are these little boxes?

Benefits of Linked Lists

- Easily resizable
- Efficient to insert elements at the beginning



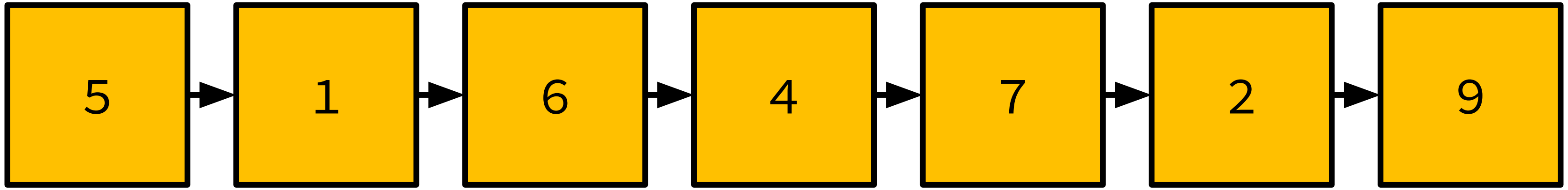
Ints?

Okay, but what are these little boxes?

Benefits of Linked Lists

- Easily resizable
- Efficient to insert elements at the beginning

Length 1 arrays?

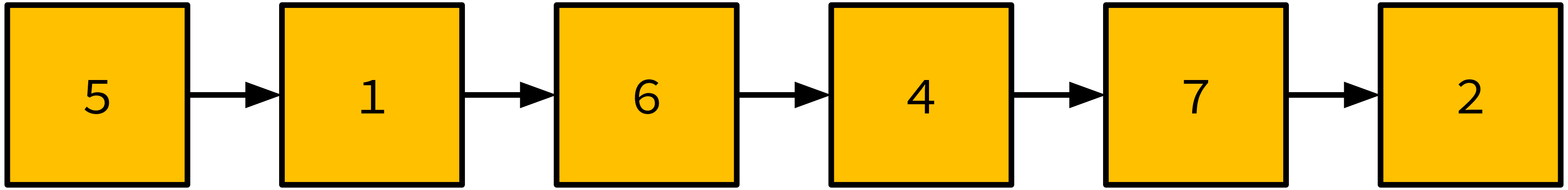


Ints?

Okay, but what are these little boxes?

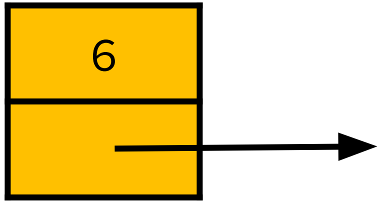
Linked Lists, Structurally

- A linked list is a chain of **nodes**


$$\begin{tikzcd} \boxed{5} \arrow[r] & \boxed{1} \arrow[r] & \boxed{6} \arrow[r] & \boxed{4} \arrow[r] & \boxed{7} \arrow[r] & \boxed{2} \end{tikzcd}$$

Linked Lists, Structurally

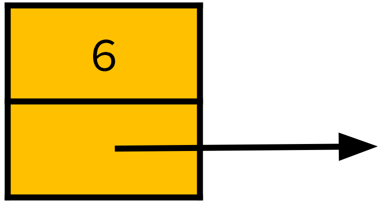
- A linked list is a chain of **nodes**
- Each node contains:
  - A piece of data (like an int, or string)
  - A link to the next node



Linked Lists, Structurally

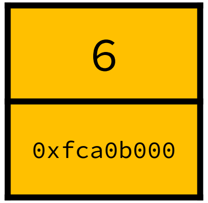
- A linked list is a chain of nodes
- Each node contains:
  - A piece of data (like an int, or string)
  - A **pointer** to the next node

What are pointers again?



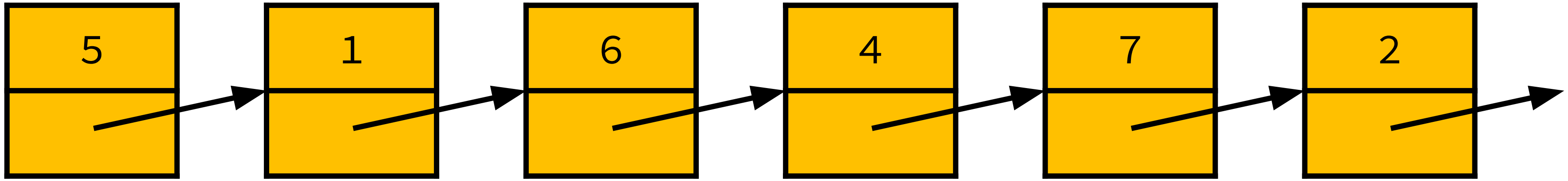
Linked Lists, Structurally

- A linked list is a chain of nodes
- Each node contains:
  - A piece of data (like an int, or string)
  - A **pointer** to the next node



Linked Lists, Structurally

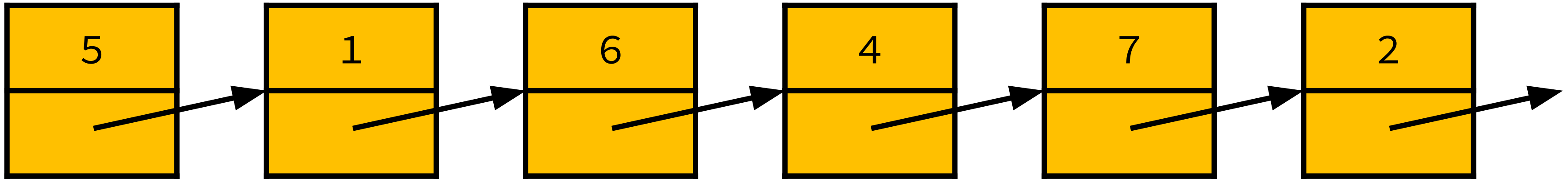
- A linked list is a chain of nodes
- Each node contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node



Linked Lists, Structurally

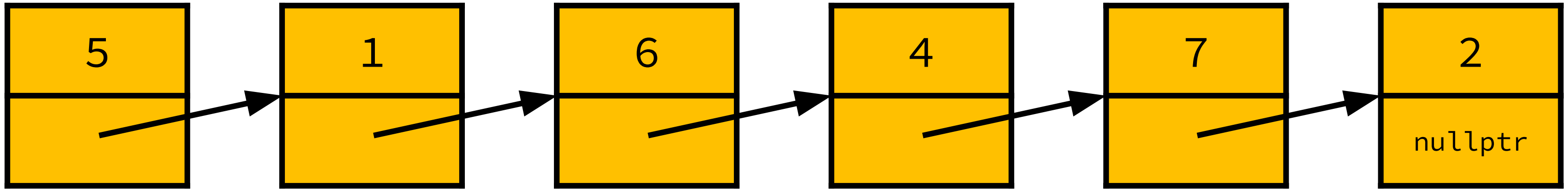
- A linked list is a chain of nodes
- Each node contains:
    - A piece of data (like an int, or string)
    - A pointer to the next node

👥 *What should the last node point to?*



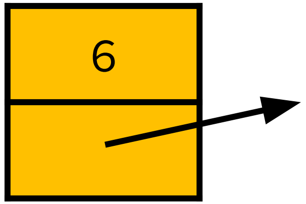
Linked Lists, Structurally

- A linked list is a chain of nodes
- Each node contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node



Linked Lists, Structurally

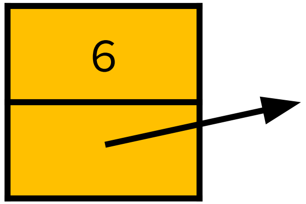
- A linked list is a chain of nodes
- Each node contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node



👥 *How can we implement a node in C++? How do we store two or more pieces of data together?*

Linked Lists, Structurally

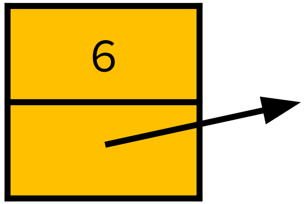
- A linked list is a chain of nodes
- Each node **is a struct** that contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node



```
struct Node {
   // data
   // pointer
};
```

Linked Lists, Structurally

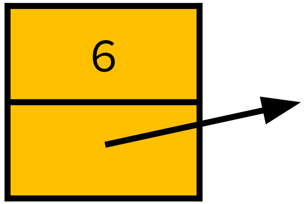
- A linked list is a chain of nodes
- Each node **is a struct** that contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node



```
struct Node {
    int data;
    // pointer
};
```

Linked Lists, Structurally

- A linked list is a chain of nodes
- Each node **is a struct** that contains:
  - A piece of data (like an int, or string)
  - A pointer to the next node

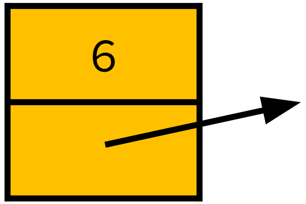


```
struct Node {
   int data;
   Node* next;
};
```

Yes, this recursive definition is allowed!

Node*

- Each Node contains a pointer to another Node, or nullptr
- A pointer to a Node is of type Node*



```
struct Node {
   int data;
   Node* next;
};
```

Creating a Linked List

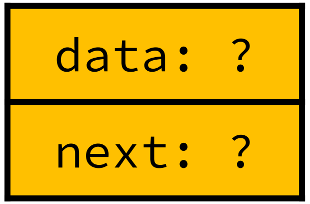
- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
```

Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
    Node* list = new Node;
```



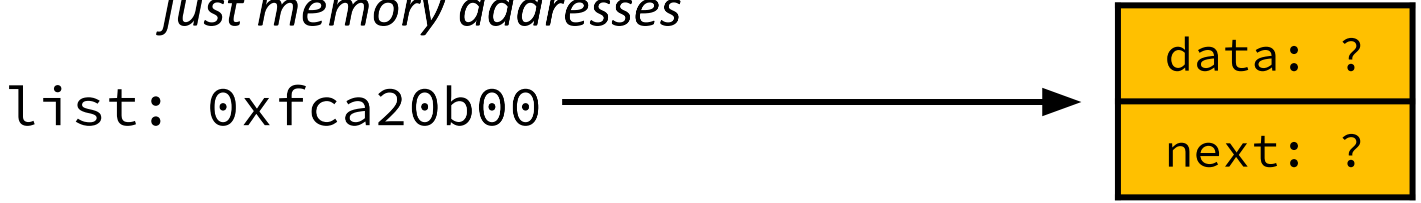
Lives at `0xfca20b00` on the heap

Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
```

Remember, pointers are just memory addresses



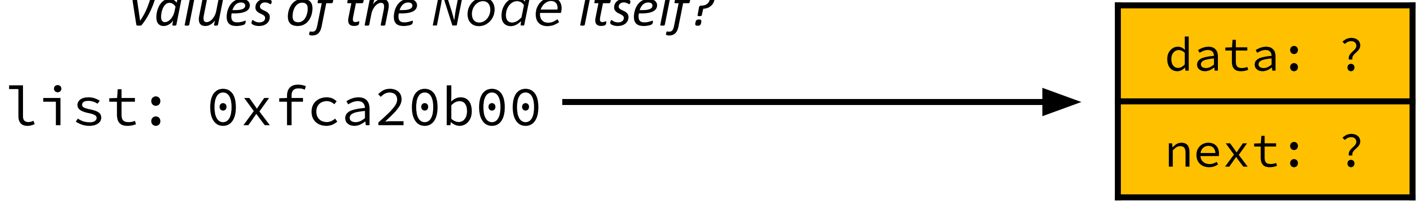
Lives at 0xfca20b00 on the heap

Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
```

How do we update the values of the Node itself?

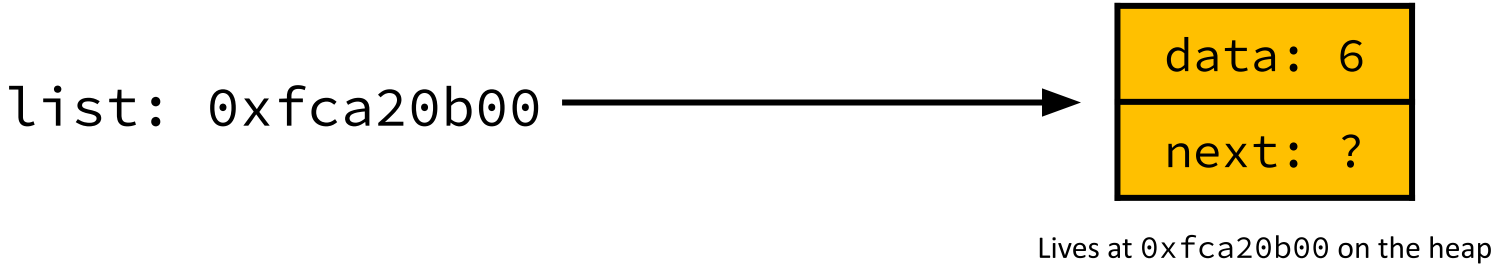


Lives at `0xfca20b00` on the heap

Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
(*list).data = 6;
```

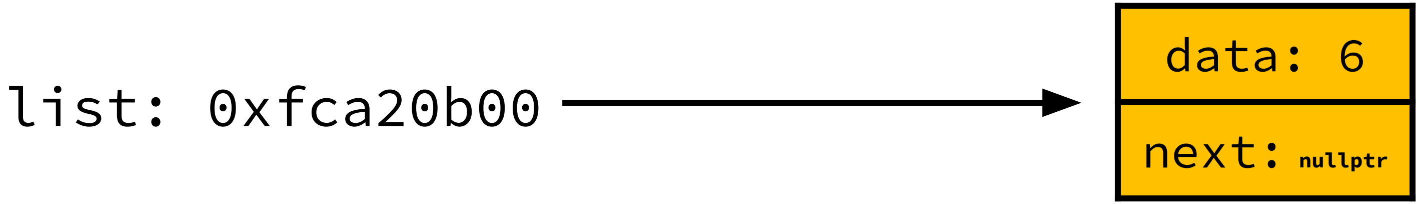


Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
(*list).data = 6;
(*list).next = nullptr;
```

*Dereference with *, access field with .*



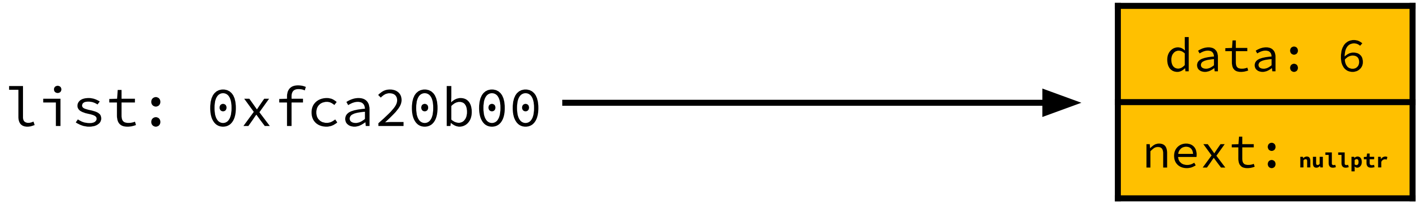
Lives at `0xfca20b00` on the heap

Creating a Linked List

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
```

Dereference AND access the field for struct pointers using ->



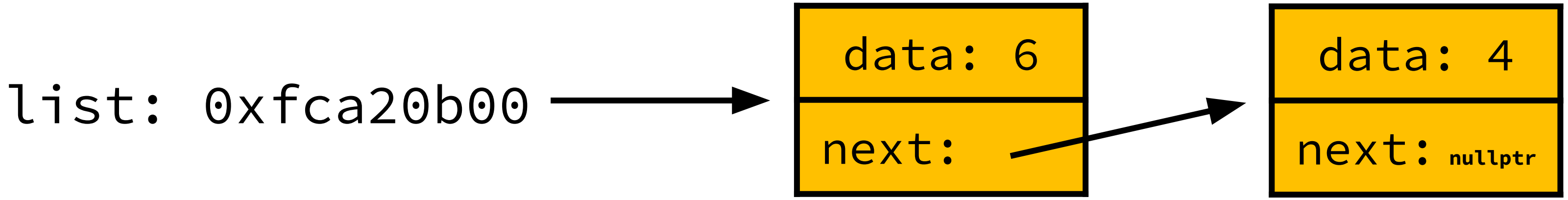
Lives at `0xfca20b00` on the heap

Appending Nodes

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
```

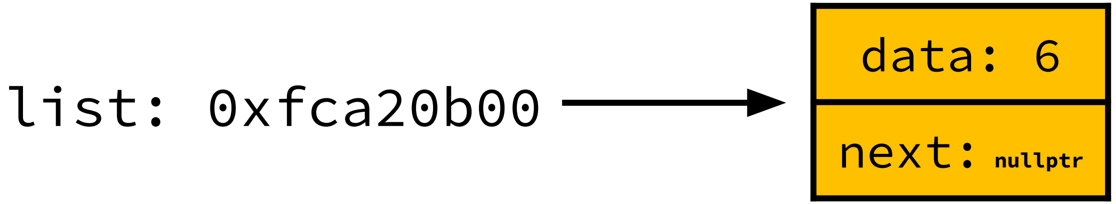
👥 *How could we build a list that looks like this?*



Appending Nodes

- Create a new Node on the heap and store a pointer to it

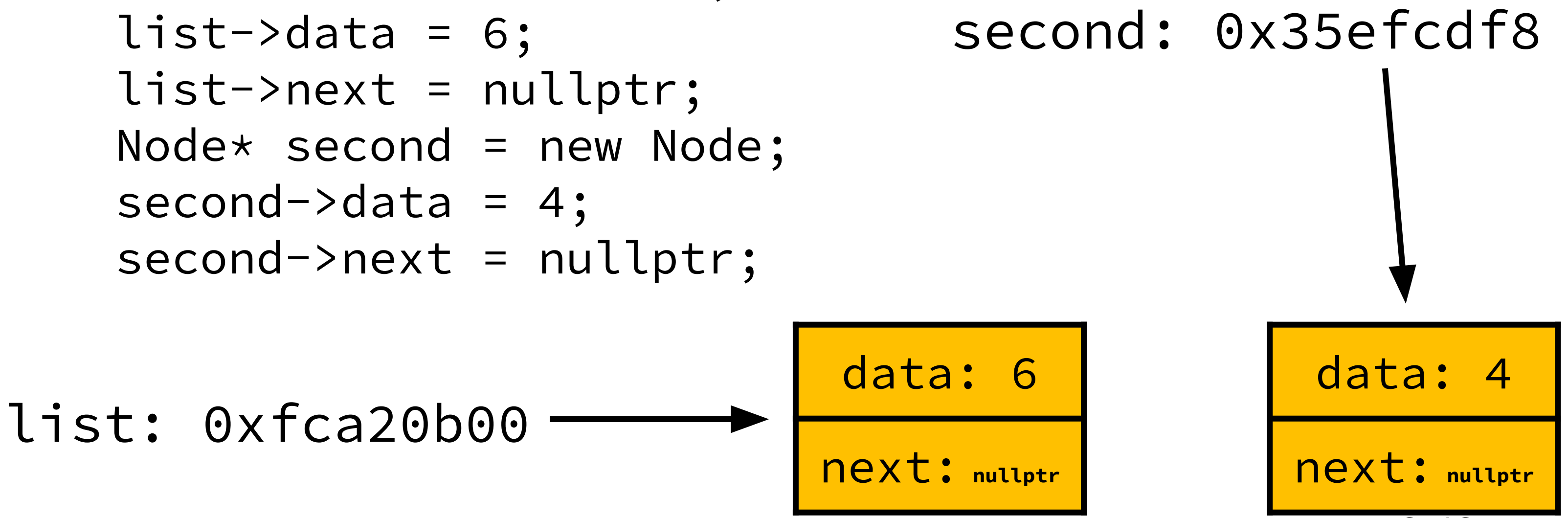
```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
```



Appending Nodes

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
Node* second = new Node;
second->data = 4;
second->next = nullptr;
```

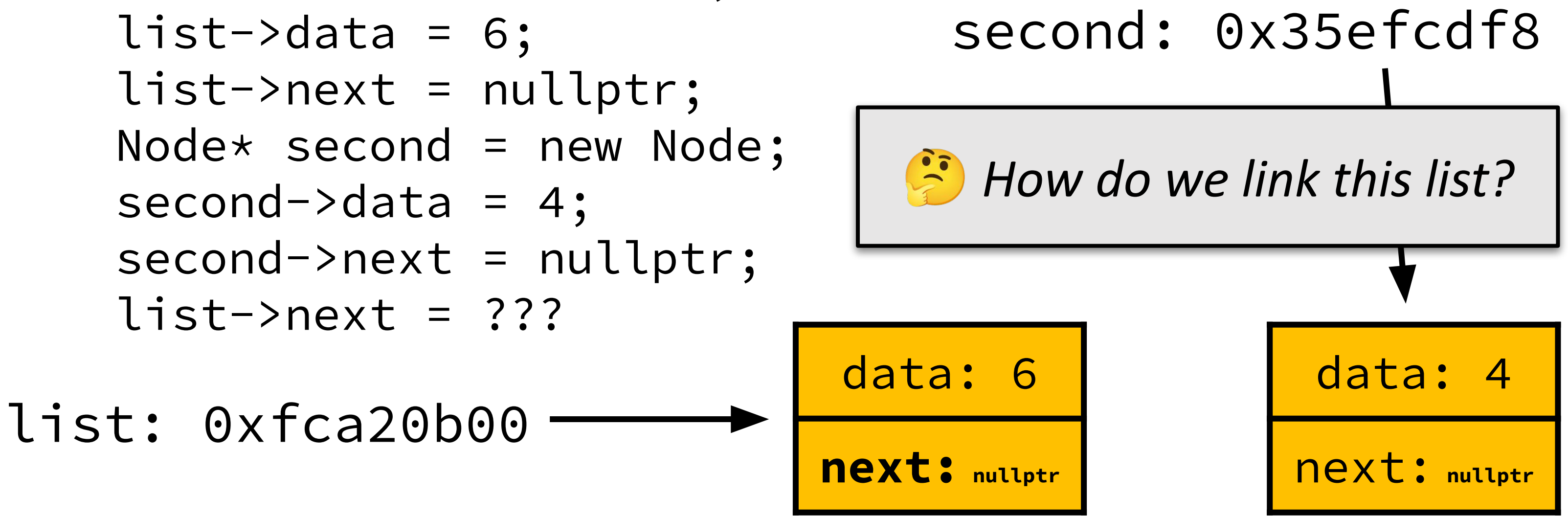


Lives at `0x35efcdf8` on the heap

Appending Nodes

- Create a new Node on the heap and store a pointer to it

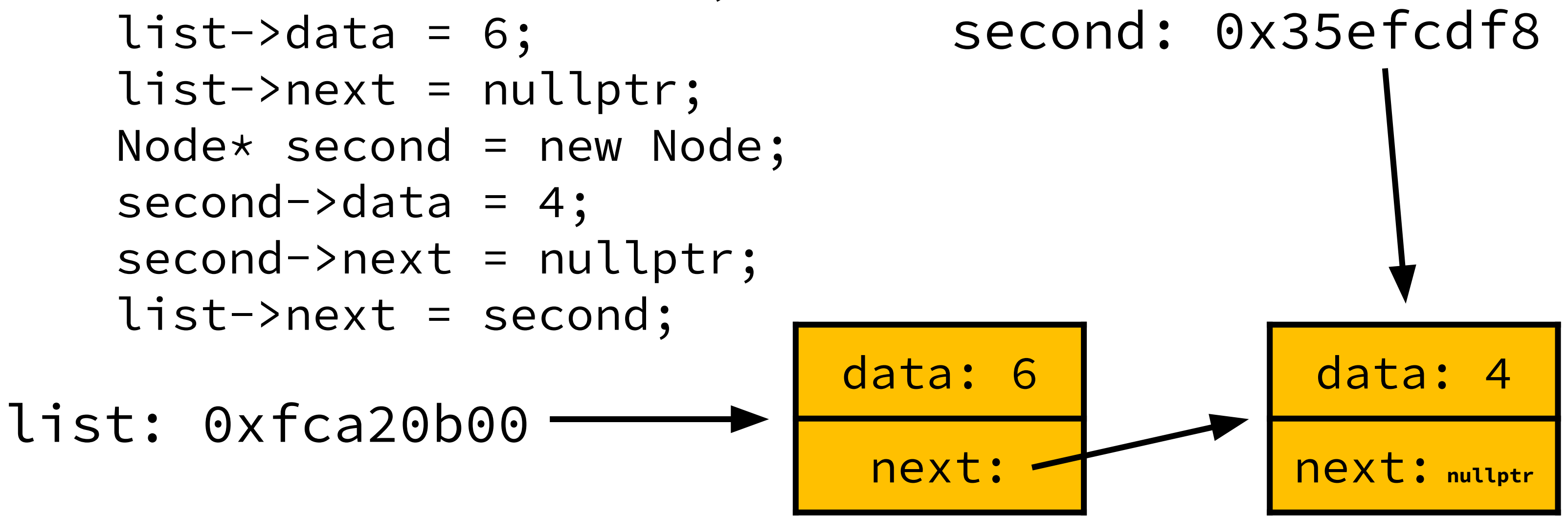
```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
Node* second = new Node;
second->data = 4;
second->next = nullptr;
list->next = ???
```



Appending Nodes

- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
Node* second = new Node;
second->data = 4;
second->next = nullptr;
list->next = second;
```

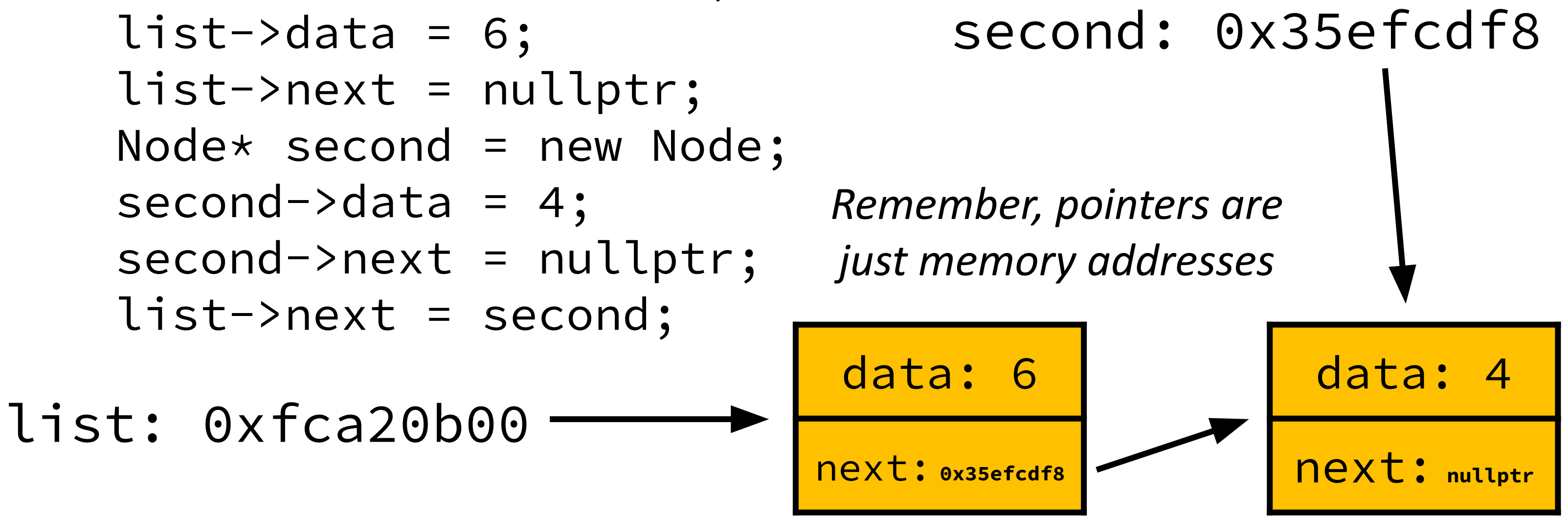


Appending Nodes

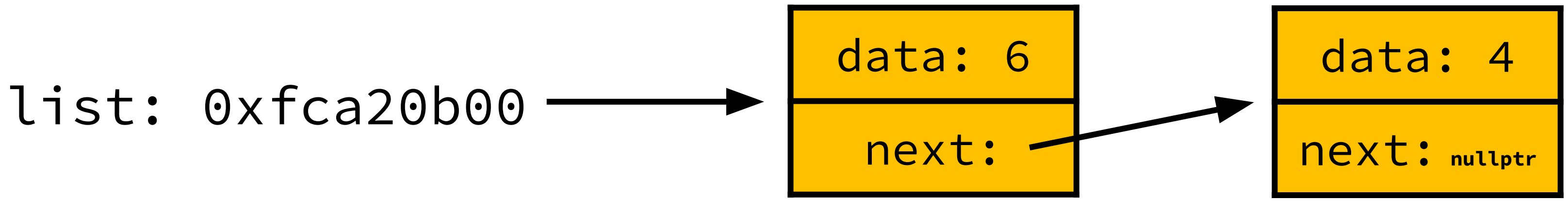
- Create a new Node on the heap and store a pointer to it

```
Node* list = new Node;
list->data = 6;
list->next = nullptr;
Node* second = new Node;
second->data = 4;
second->next = nullptr;
list->next = second;
```

Remember, pointers are just memory addresses

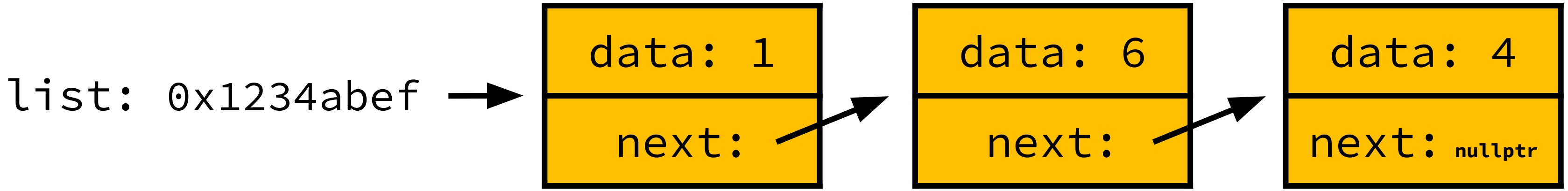


Prepending Nodes



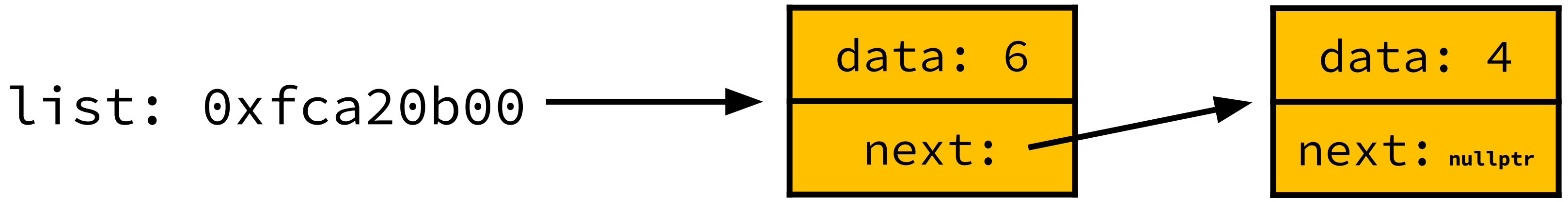
How would we go from this…

Prepending Nodes

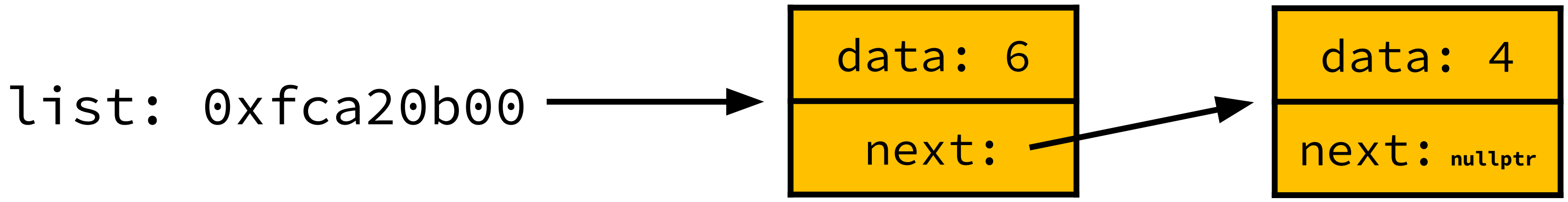


… to this?

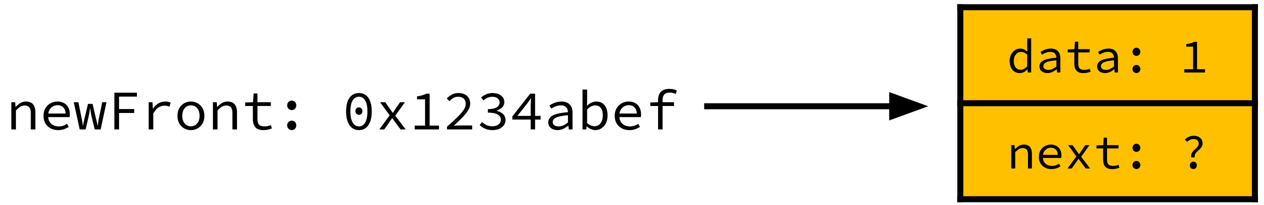
Prepending Nodes



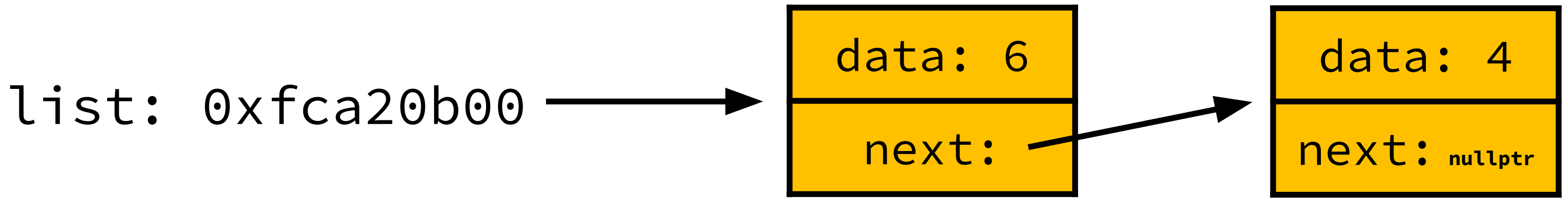
Prepending Nodes



```
Node* newFront = new Node;
newFront->data = 1;
```

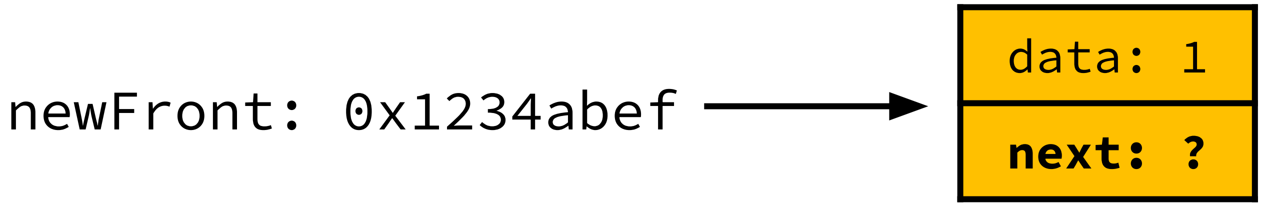


Prepending Nodes

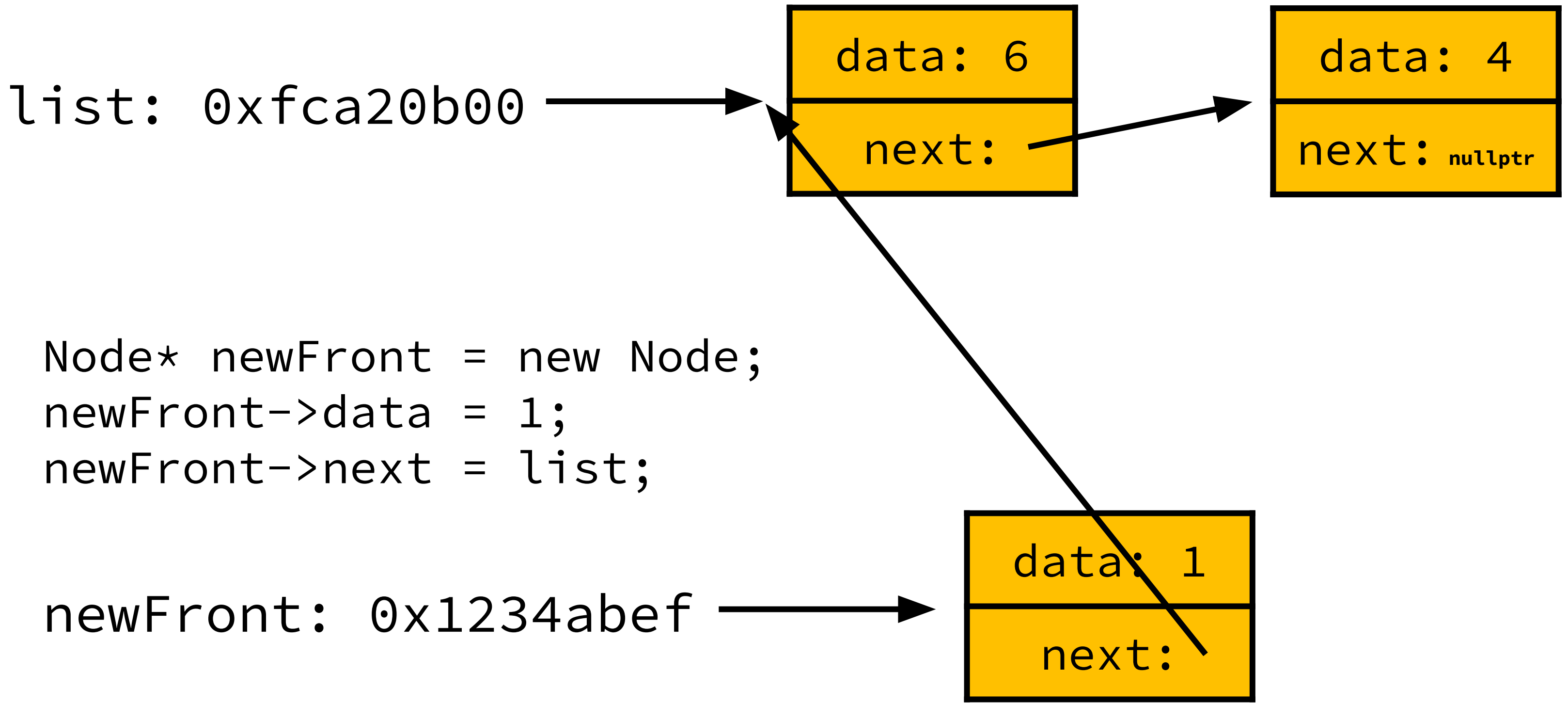


```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = ???
```

🤔 *Help me out here…*

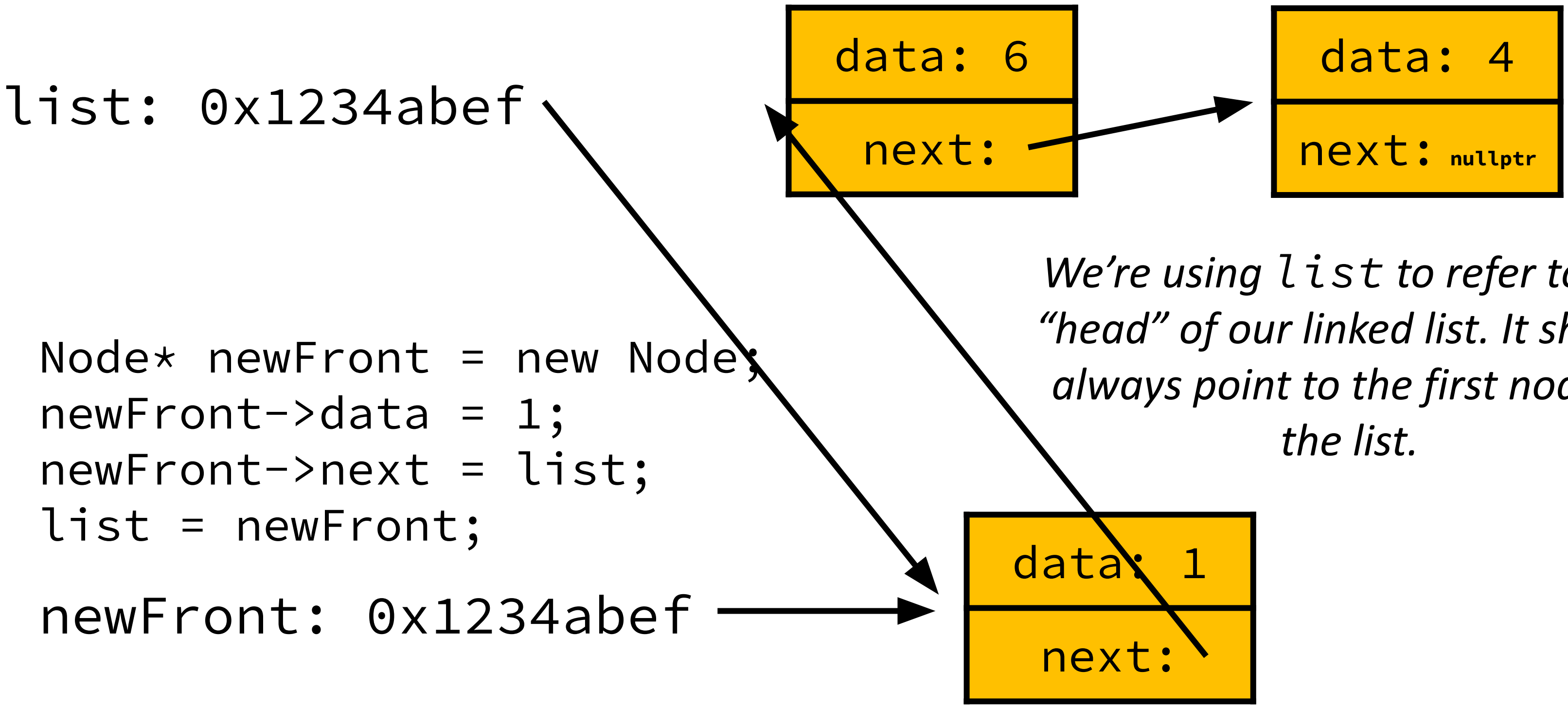


Prepending Nodes



```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = list;
```

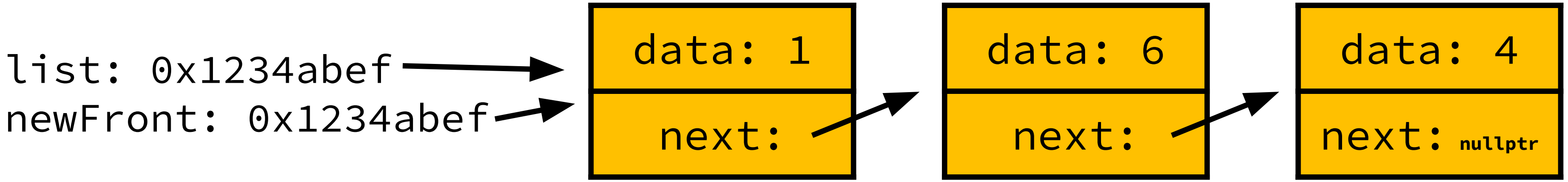
Prepending Nodes



```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = list;
list = newFront;
```

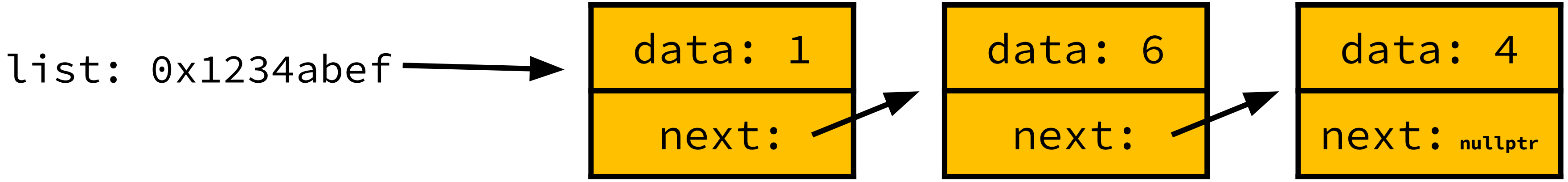
We're using `list` to refer to the "head" of our linked list. It should always point to the first node in the list.

Prepending Nodes



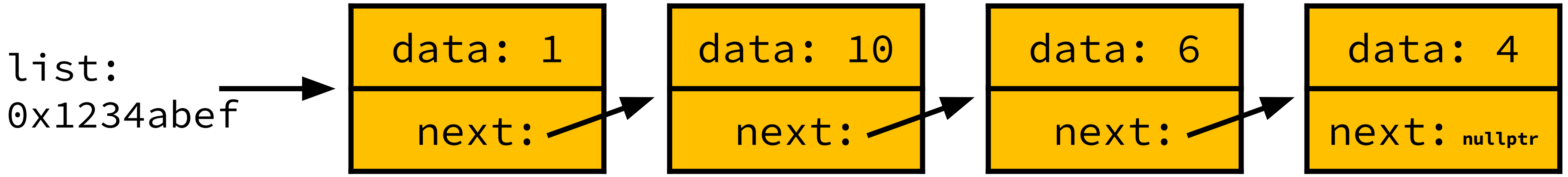
```
Node* newFront = new Node;
newFront->data = 1;
newFront->next = list;
list = newFront;
```

Let’s Trace Some Code



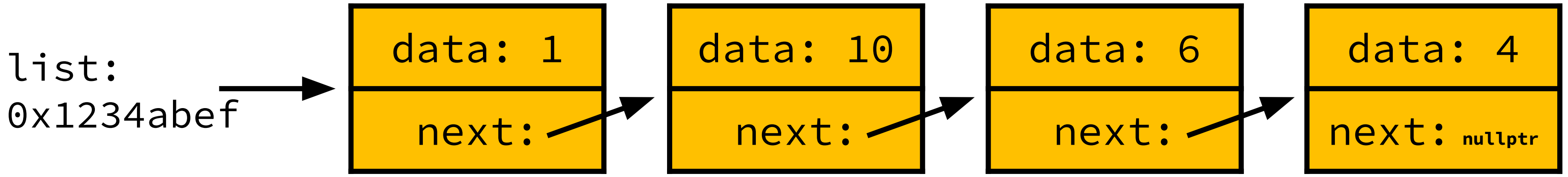
```
Node* mystery = new Node;
mystery->data = 10;
mystery->next = list->next;
list->next = mystery;
```

Let’s Trace Some Code



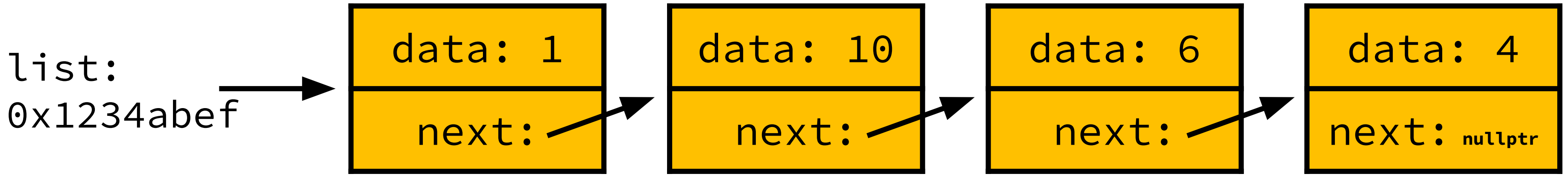
```
Node* mystery = new Node;
mystery->data = 10;
mystery->next = list->next;
list->next = mystery;
```

Let’s Trace Some Code (Inserting Nodes)

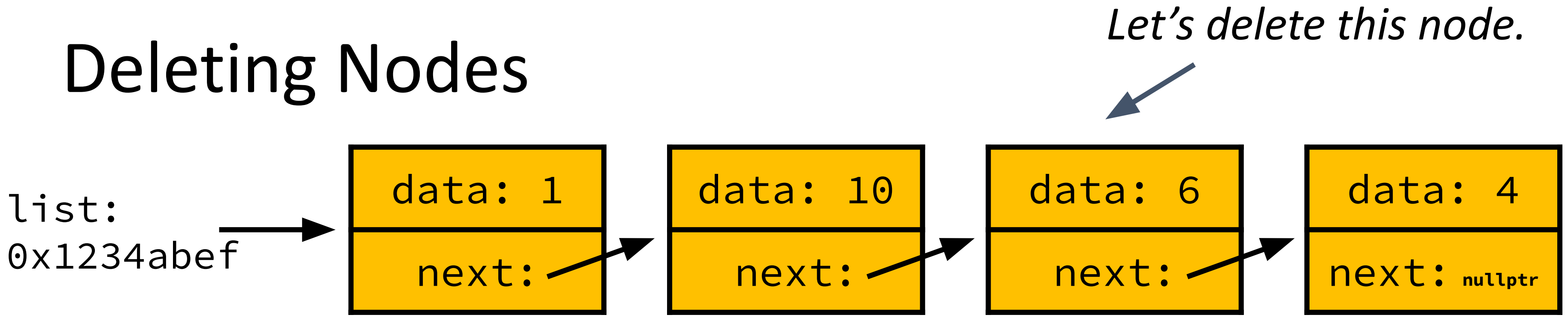


```
Node* mystery = new Node;
mystery->data = 10;
mystery->next = list->next;
list->next = mystery;
```

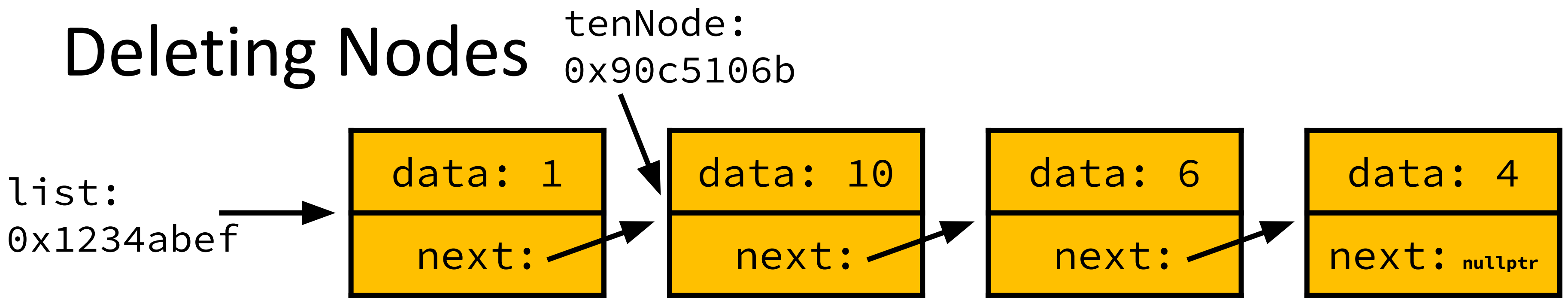
Deleting Nodes



Deleting Nodes

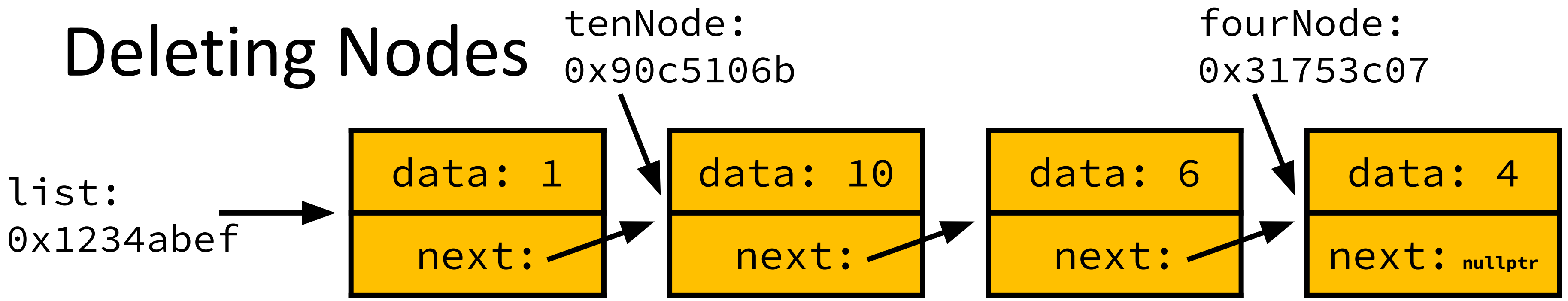


Deleting Nodes



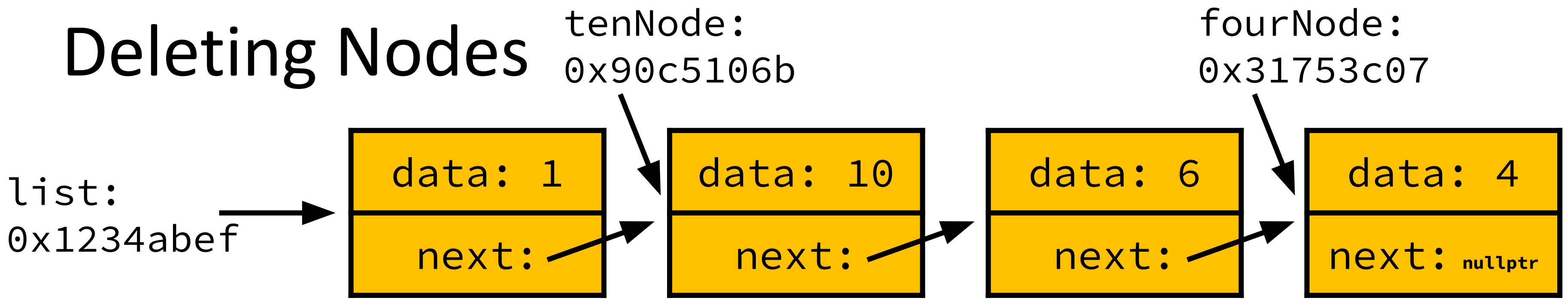
```
Node* tenNode = list->next;
```

Deleting Nodes



```
Node* tenNode = list->next;

Node* fourNode = list->next->next->next;
```

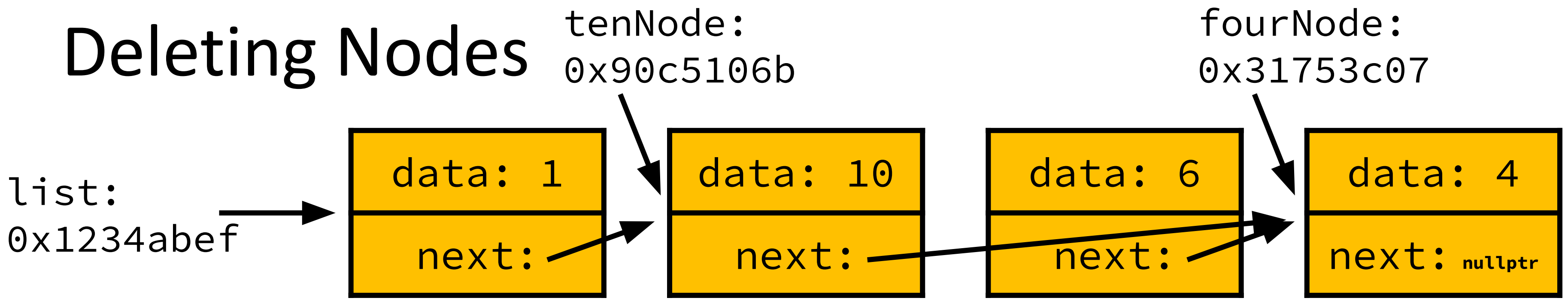
Deleting Nodes



```
Node* tenNode = list->next;

Node* fourNode = list->next->next->next;
```

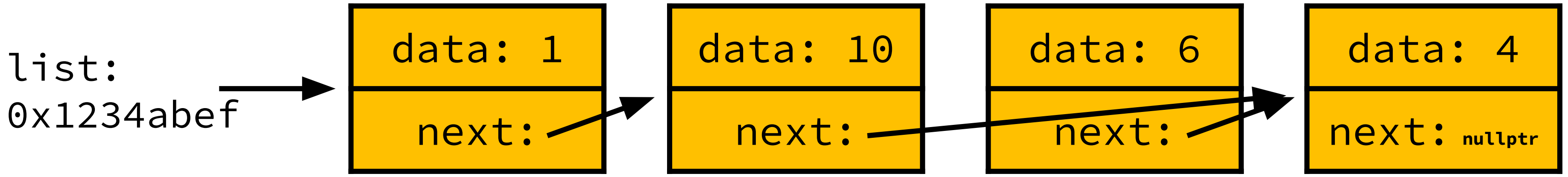
In practice, we wouldn't hard-code the number of `->next`*s like this…*
We'll see linked list traversal shortly!

Deleting Nodes



```
Node* tenNode = list->next;

Node* fourNode = list->next->next->next;

tenNode->next = fourNode;
```

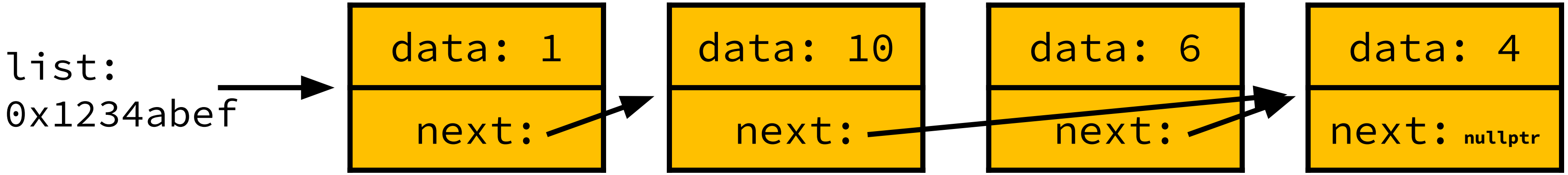
Deleting Nodes *BUGGY



```
Node* tenNode = list->next;

Node* fourNode = list->next->next->next;

tenNode->next = fourNode;
```

👥 *What's wrong with this approach?*

Deleting Nodes *BUGGY

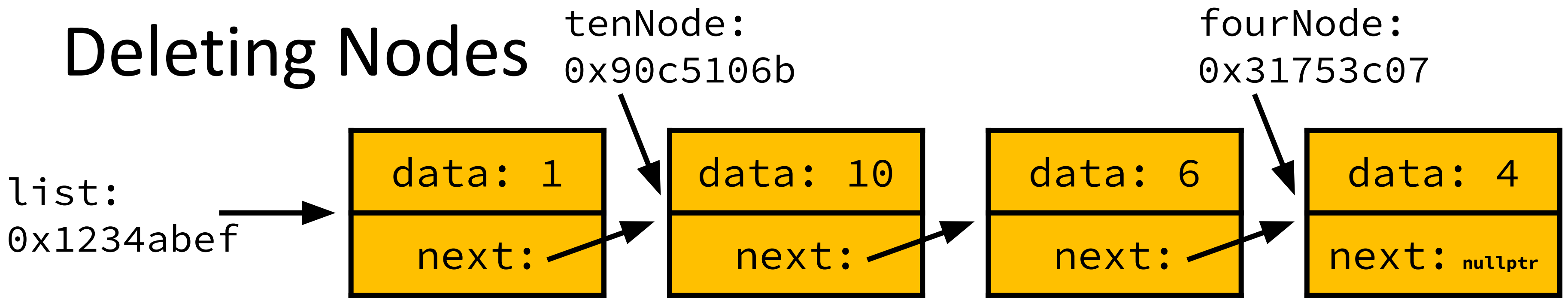


```
Node* tenNode = list->next;
Node* fourNode = list->next->next->next;
tenNode->next = fourNode;
```

MEMORY LEAK 👎

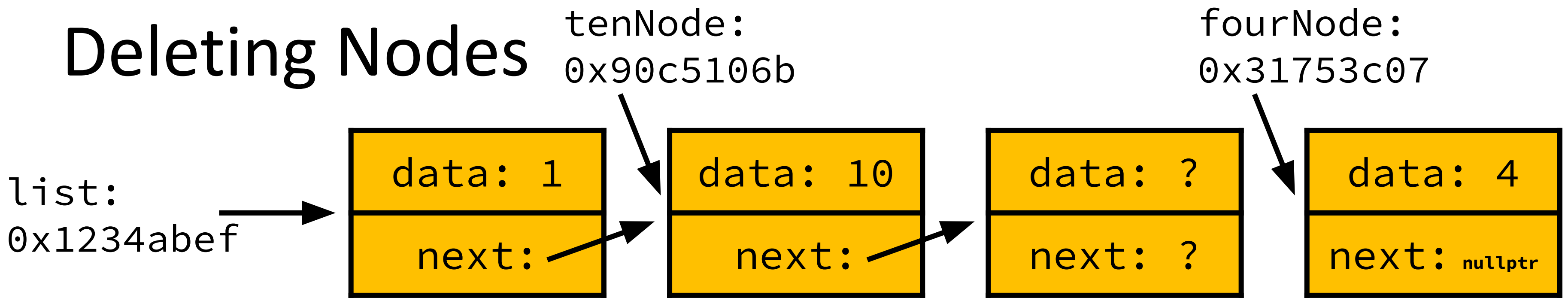
Now, we have no way of referring to the node that contains 6!
We'd like to `delete` it, but we don't have a pointer to it.

Deleting Nodes



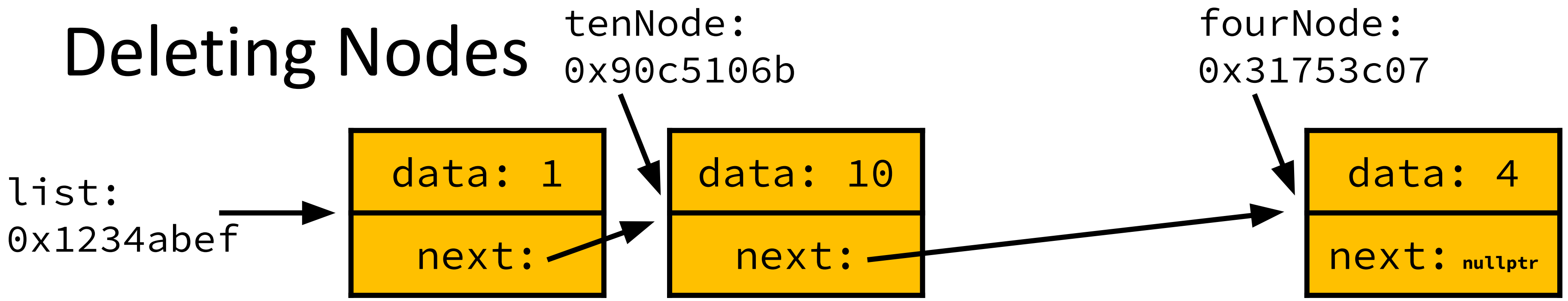
```
Node* tenNode = list->next;

Node* fourNode = list->next->next->next;
```

Deleting Nodes



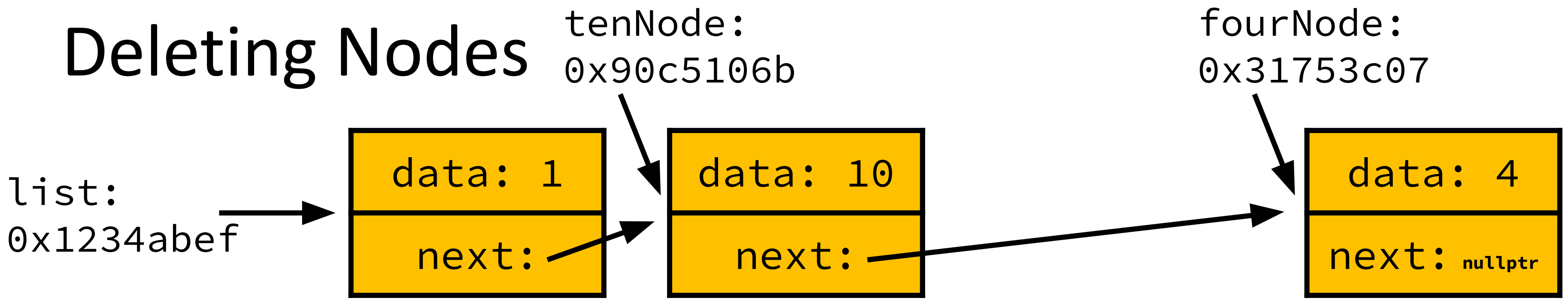
```
Node* tenNode = list->next;

Node* fourNode = list->next->next->next;

delete tenNode->next;
```

Deleting Nodes



```
Node* tenNode = list->next;
Node* fourNode = list->next->next->next;
delete tenNode->next;
tenNode->next = fourNode;
```

Deleting Nodes



```
Node* tenNode = list->next;

Node* fourNode = list->next->next->next;

delete tenNode->next;

tenNode->next = fourNode;
```

When deleting a node, we need to free its memory AND rewire the other nodes.

Demo: Traversing a Linked List

🎟️ *Attendance ticket: applications of linked list traversal*

Solution: Traversing a Linked List

```
void printList(Node* list) {
    while (list != nullptr) {
        cout << list->data << endl;
        list = list->next;
    }
}

int measureList(Node* list) {
    int count = 0;
    while (list != nullptr) {
        count++;
        list = list->next;
    }
    return count;
}
```

```
void freeList(Node* list) {
    while (list != nullptr) {
        Node* temp = list->next;
        delete list;
        list = temp;
    }
}
```

Recap

- Downsides of arrays
- Benefits of linked lists
- Basic linked list operations
    - Initializing nodes
    - Adding nodes: Append / Prepend / Insert
    - Deleting nodes
- Traversing a linked list

Thank you!