SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220411<A|D>

Replace <A|D> with this section's letter

Linked Lists

—

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- Array limitations
- Linked lists
- A linked list toolkit

Array limitations

Limitations of array type

- no out-of-bounds checking

```
const int NUM_INTS = 6;

int main() {
    int arr[NUM_INTS];

    arr[100] = 3;
    cout << arr[100] << endl;

    for (size_t i = 0; i <= NUM_INTS; i++) {
        cout << arr[i] << endl;
    }
}
```

```
warning: array index 100 is past the end of the array
(which contains 6 elements) [-Warray-bounds]
 arr[100] = 3;
 ^  ~~~
```

generate warnings

i equal to NUM_INTS exceeds array bounds

able to access (and modify) memory locations outside of array

arr

0	0
0	1
0	2
0	3
-404154144	4
32766	5

```
% g++ -std=c++11 array_tests.cpp -o array_tests.o
% ./array_tests.o
3
0
0
0
0
-404154144
32766
672727067
```

Limitations of array type

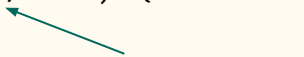
- no out-of-bounds checking
- need to know size
  - array arguments to function

```
const int NUM_INTS = 6;

void init_array(int in_arr[]);



int main() {
    int arr[NUM_INTS];
    init_array(arr);

    for (size_t i = 0; i < NUM_INTS; i++) {
        cout << arr[i] << endl;
    }
}
```

Limitations of array type

- no out-of-bounds checking
- need to know size
  - array arguments to function

```
const int NUM_INTS = 6;

void init_array(int in_arr[]) {
    for (size_t i = 0; i < ??; ++i) {
        in_arr[i] = 0;
    }
}

int main() {
    int arr[NUM_INTS];
    init_array(arr);

    for (size_t i = 0; i < NUM_INTS; i++) {
        cout << arr[i] << endl;
    }
}
```



when does iteration stop?

Limitations of array type

- no out-of-bounds checking
- need to know size
  - array arguments to function

```
const int NUM_INTS = 6;

void init_array(int in_arr[], size_t arr_size) {
    for (size_t i = 0; i < ??; ++i) {
         in_arr[i] = 0;
    }
}
int main() {
    int arr[NUM_INTS];
    init_array(arr);

    for (size_t i = 0; i < NUM_INTS; i++) {
        cout << arr[i] << endl;
    }
}
```

Limitations of array type

- no out-of-bounds checking
- need to know size
  - array arguments to function

```
const int NUM_INTS = 6;

void init_array(int in_arr[], size_t arr_size) {
    for (size_t i = 0; i < arr_size; ++i) {
         in_arr[i] = 0;
    }
}
int main() {
    int arr[NUM_INTS];
    init_array(arr);

    for (size_t i = 0; i < NUM_INTS; i++) {
        cout << arr[i] << endl;
    }
}
```

Limitations of array type

- no out-of-bounds checking
- need to know size
  - array arguments to function

```
const int NUM_INTS = 6;

void init_array(int in_arr[], size_t arr_size) {
    for (size_t i = 0; i < arr_size; ++i) {
         in_arr[i] = 0;
    }
}
int main() {
    int arr[NUM_INTS];
    init_array(arr, NUM_INTS);

    for (size_t i = 0; i < NUM_INTS; i++) {
        cout << arr[i] << endl;
    }
}
```

Limitations of array type

- no out-of-bounds checking
- need to know size
  - array arguments to function
  - declaring local array variable

```
int main() {

    int arr[];  compilation error!

}
```

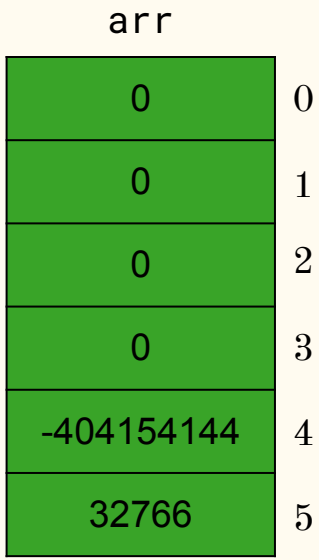
```
array_tests3.cpp:5:7: error: definition of variable with
array type needs an explicit size or an initializer
  int arr[];
      ^
```

Limitations of array type

- no out-of-bounds checking
- need to know size
  - array arguments to function
  - declaring local array variable
- no ability to add or remove elements

```
const int NUM_INTS = 6;

int main() {

    int arr[NUM_INTS];

}
```

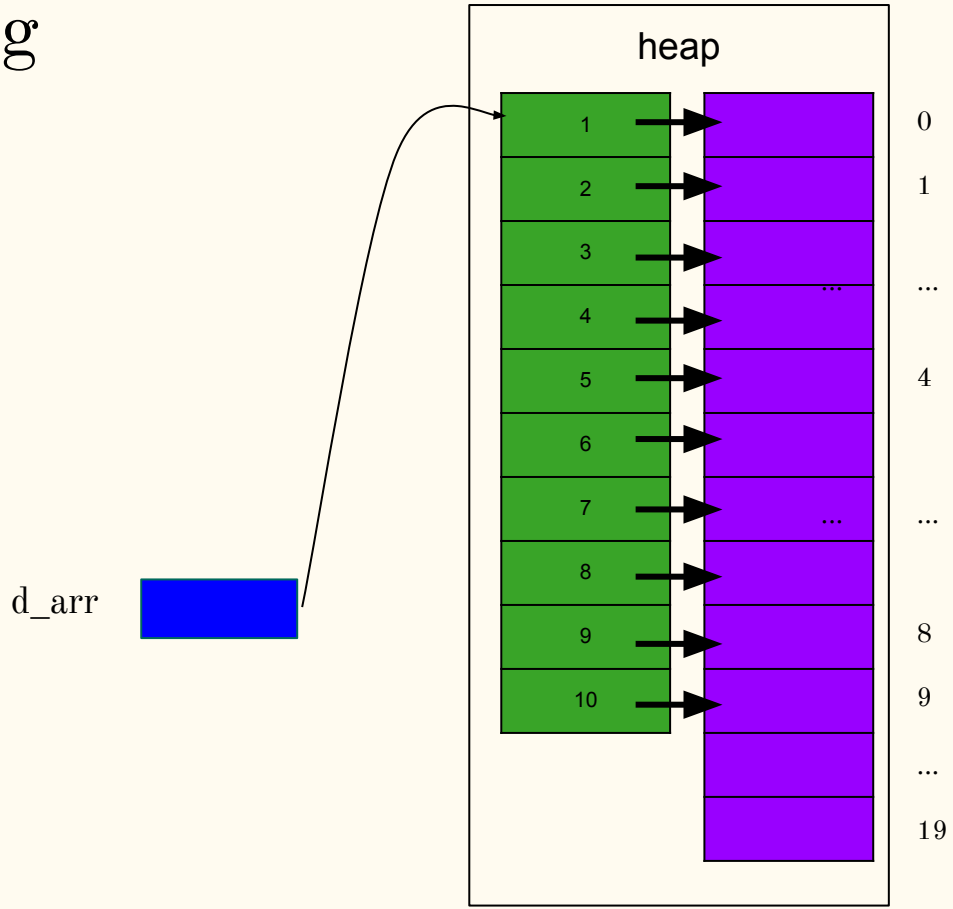
array size can never change



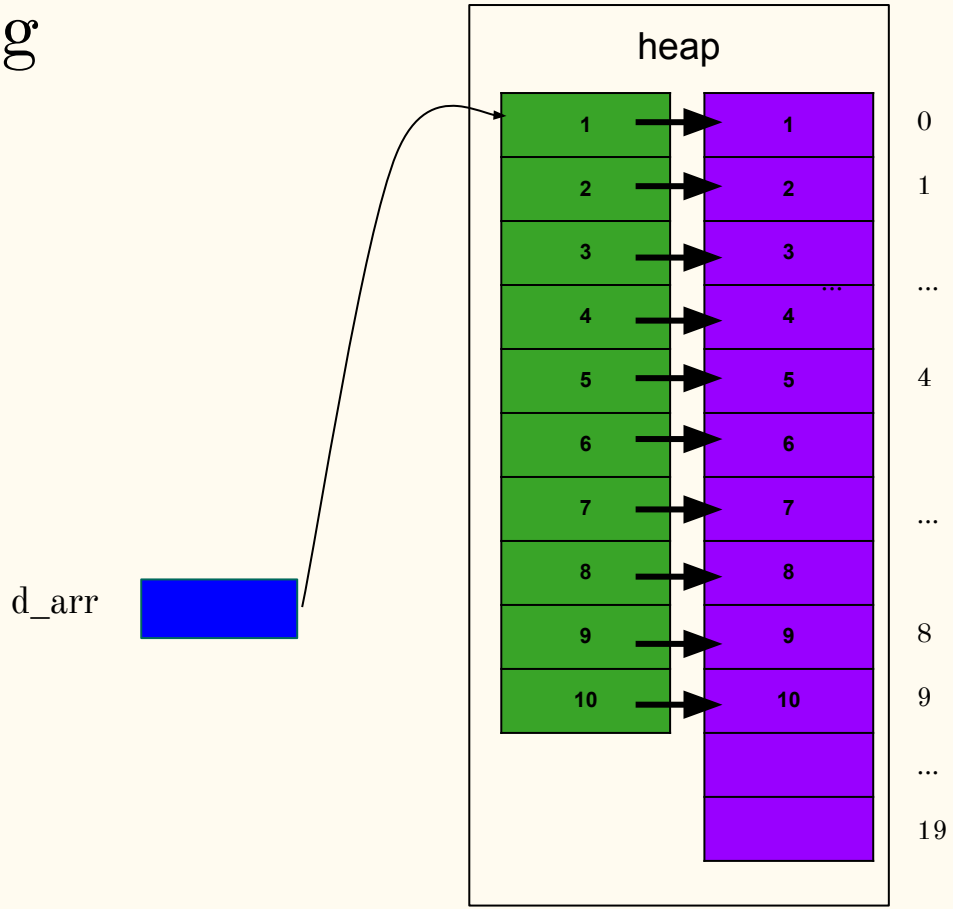
Dynamic arrays

- solve some problems with static arrays
  - determine size at runtime
  - increase size
    - make new array
    - copy values from old to new
    - add new values as needed
  - decrease size
    - make new array
    - copy values to retain from old to new
- problems remain
  - inserting values
  - removing values
  - frequent resizing

Dynamic array resizing

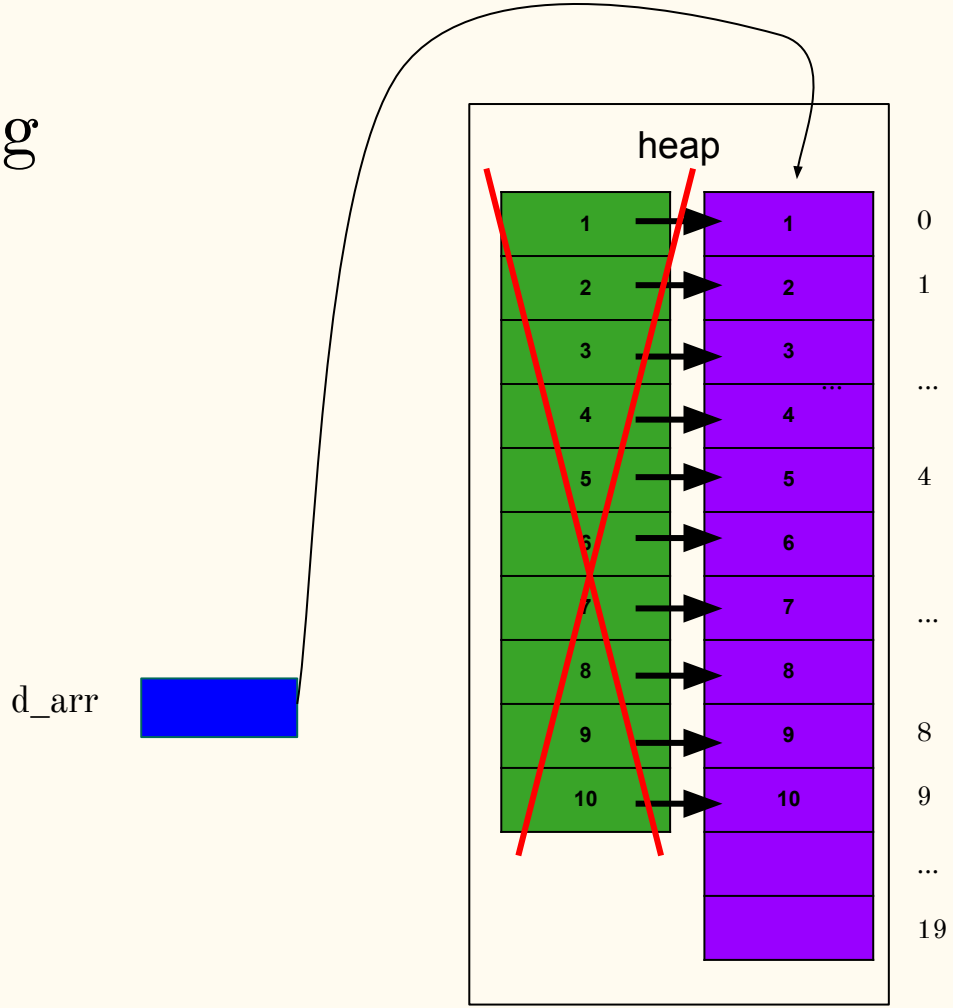


Dynamic array resizing

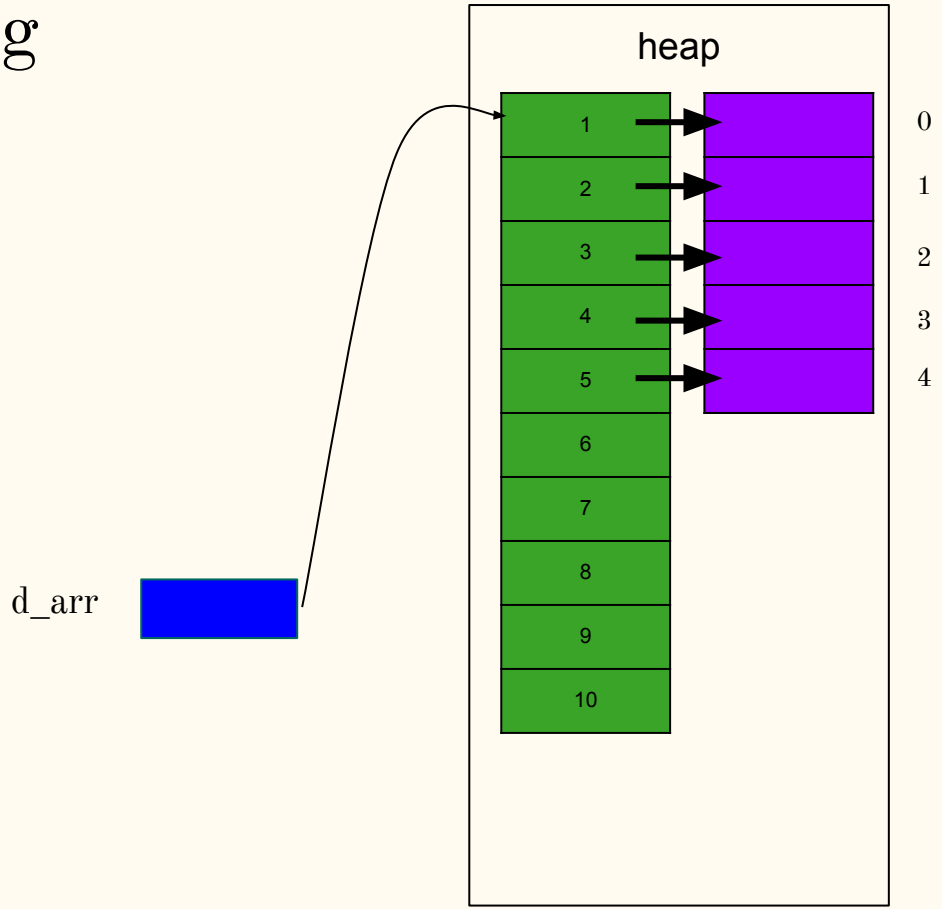


Dynamic array resizing

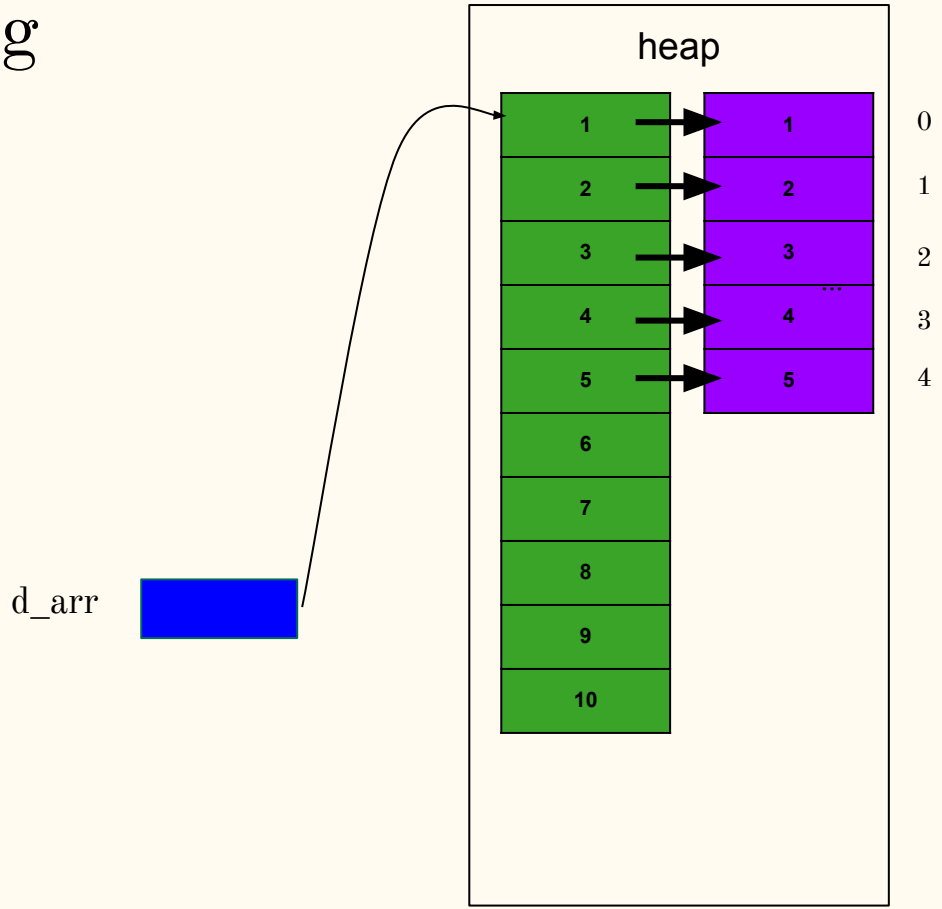
overhead costs from memory allocation and copying



Dynamic array resizing

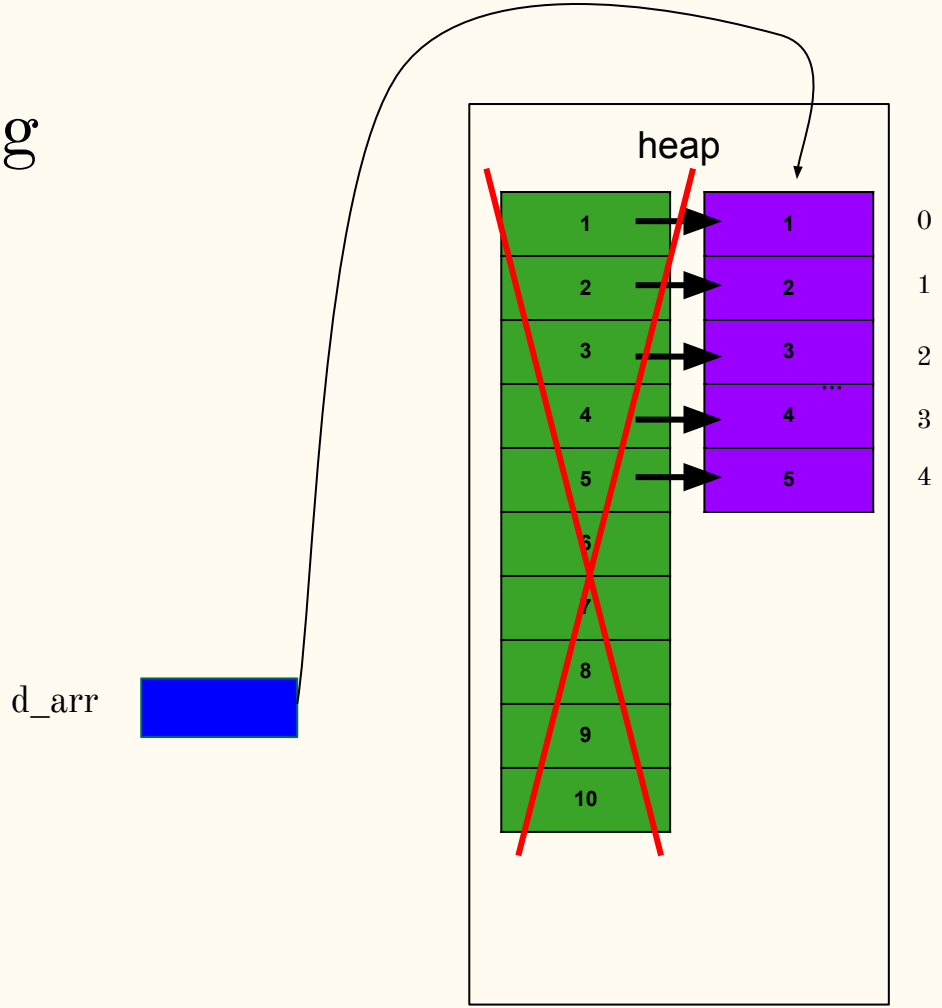


Dynamic array resizing

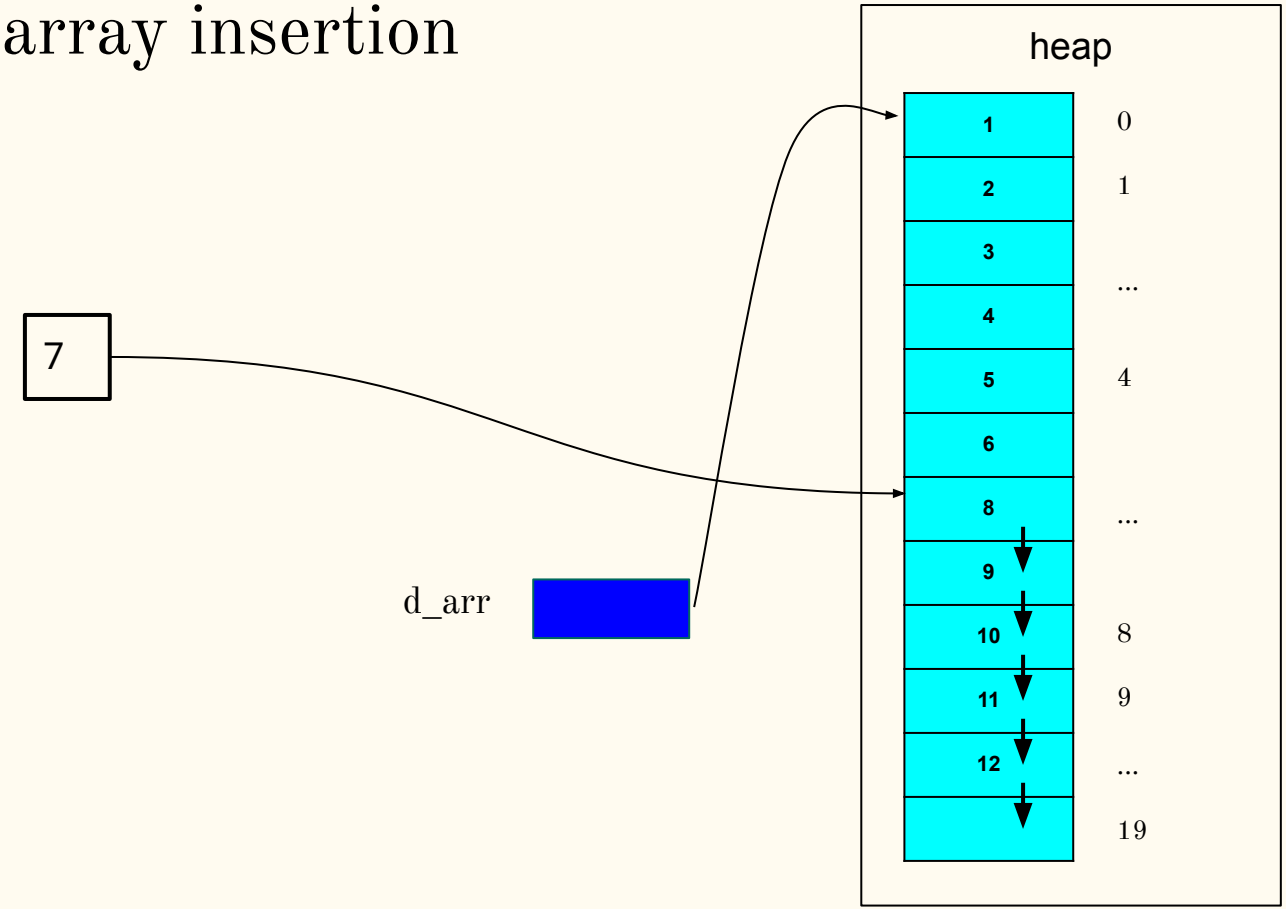


Dynamic array resizing

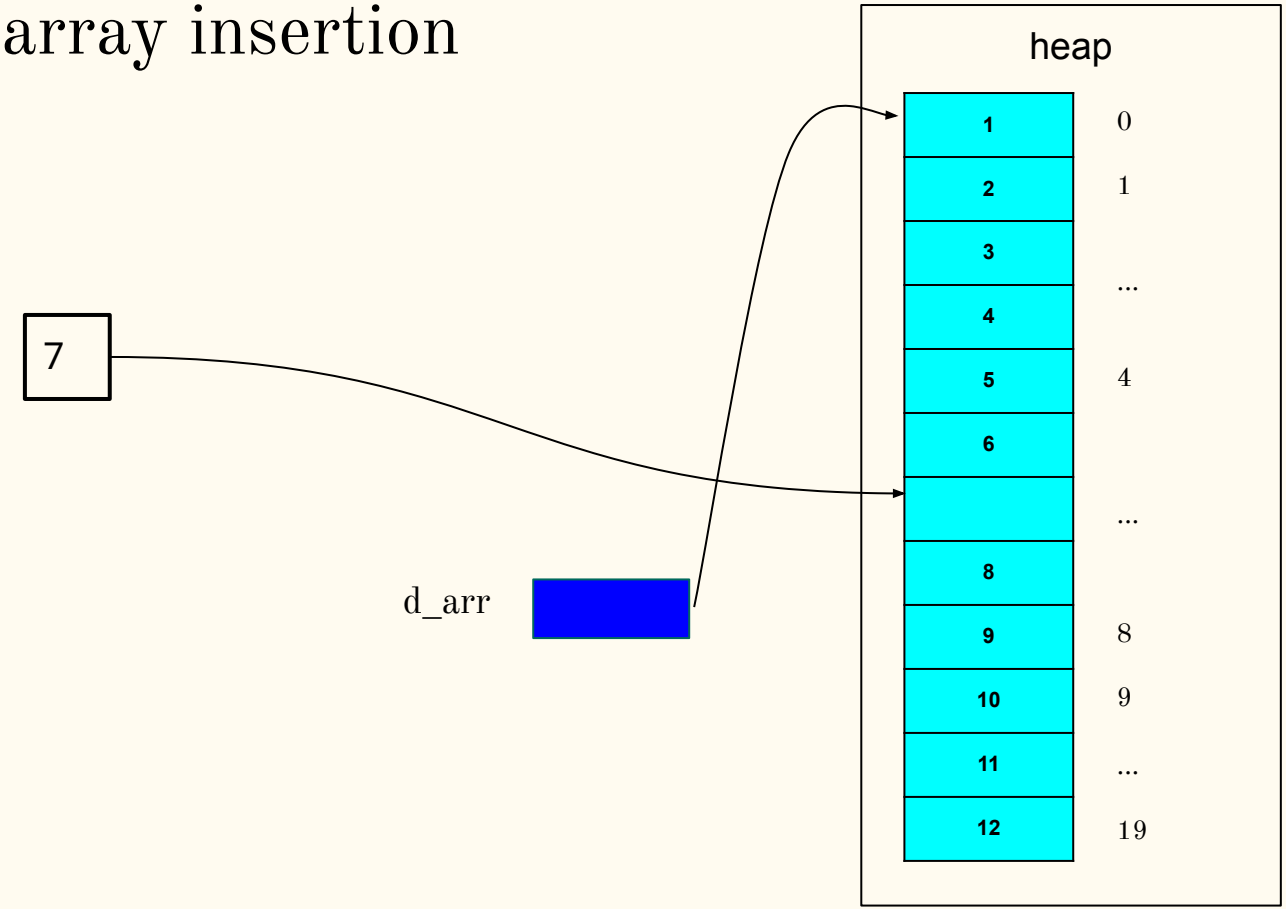
overhead costs from memory allocation and copying



Dynamic array insertion

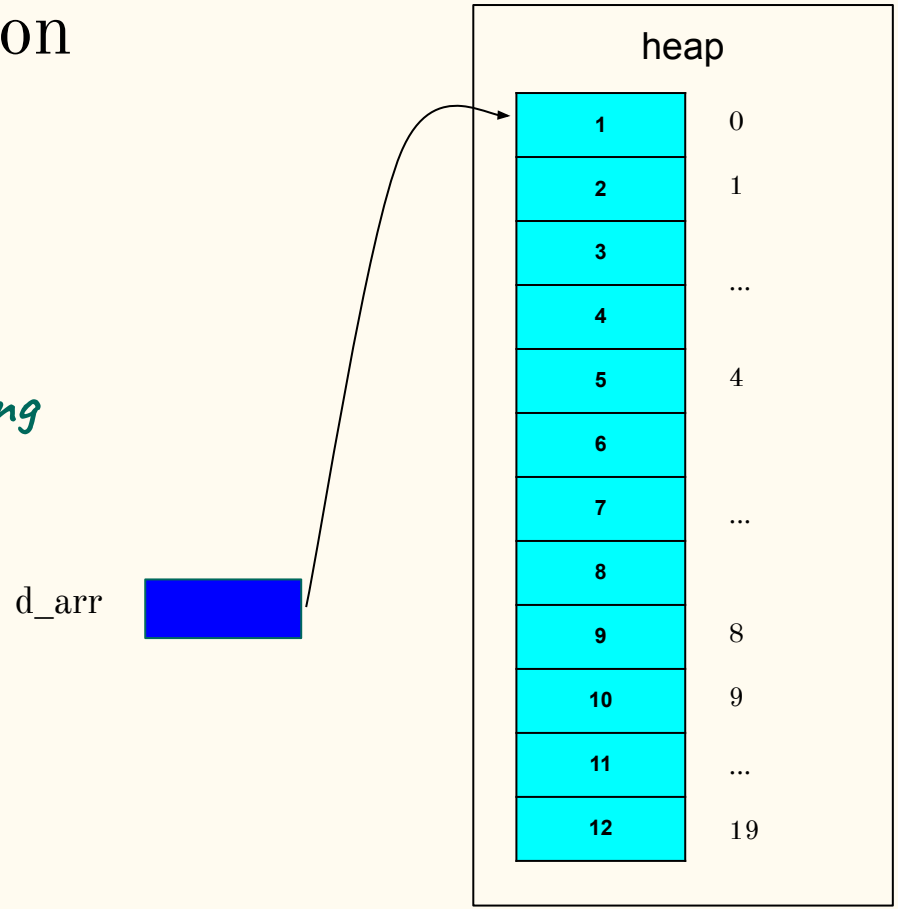


Dynamic array insertion

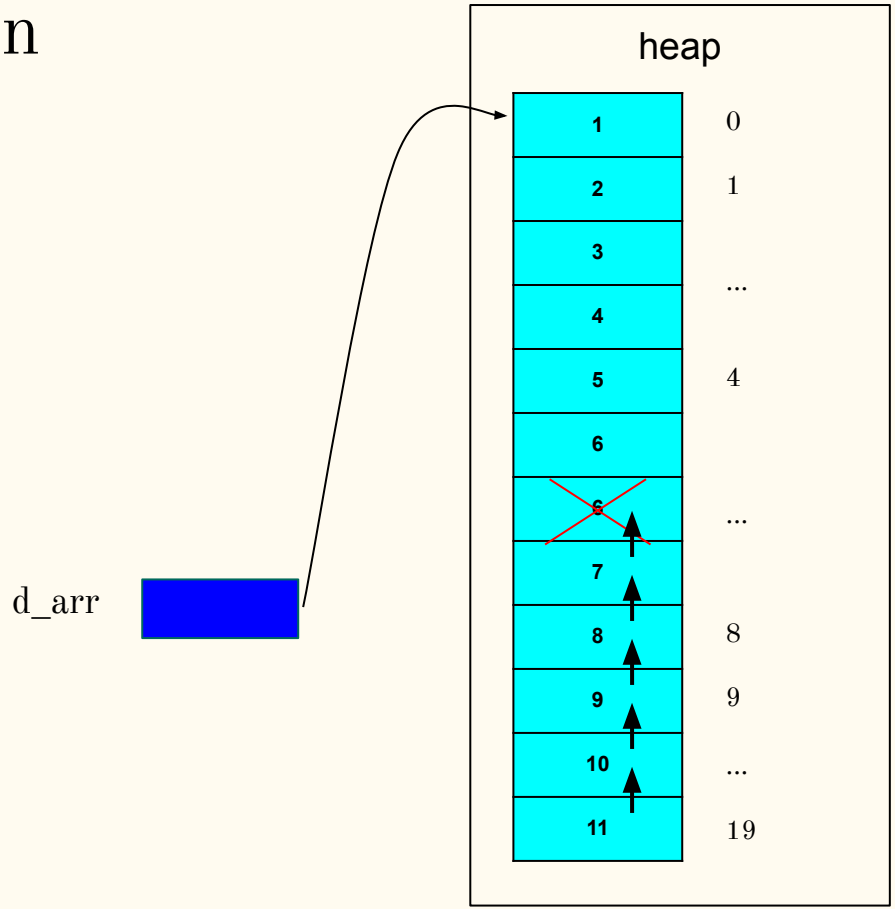


Dynamic array insertion

overhead costs from relocating array elements

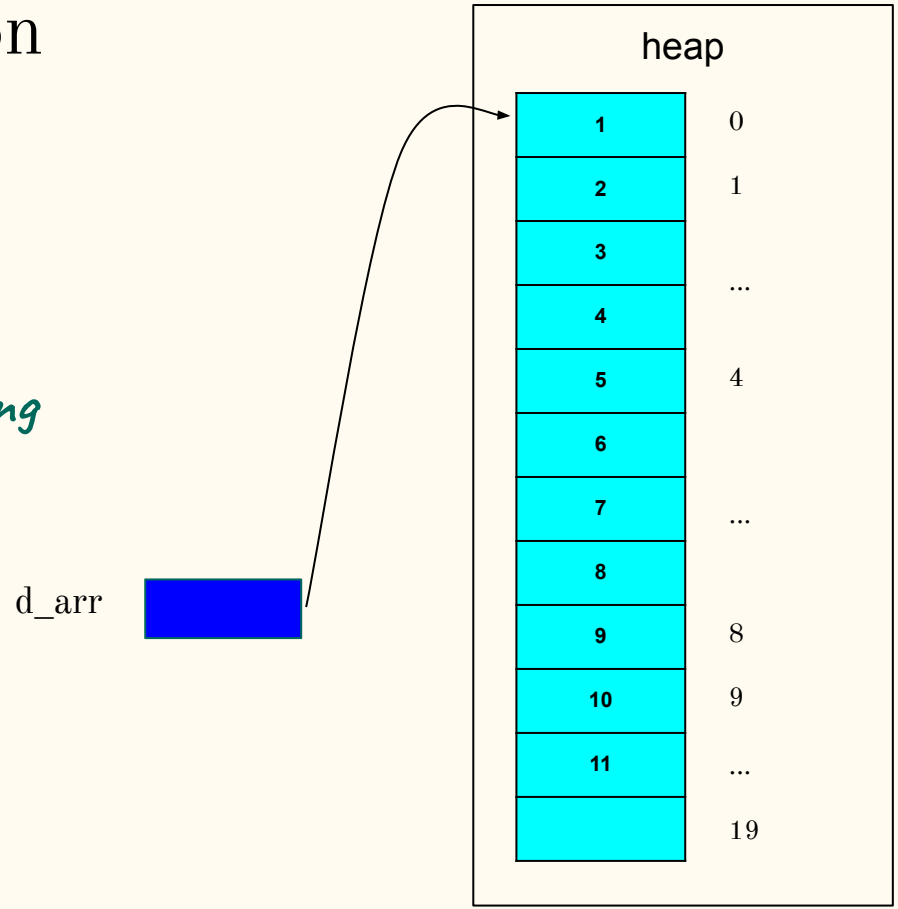


Dynamic array deletion



Dynamic array deletion

overhead costs from relocating array elements

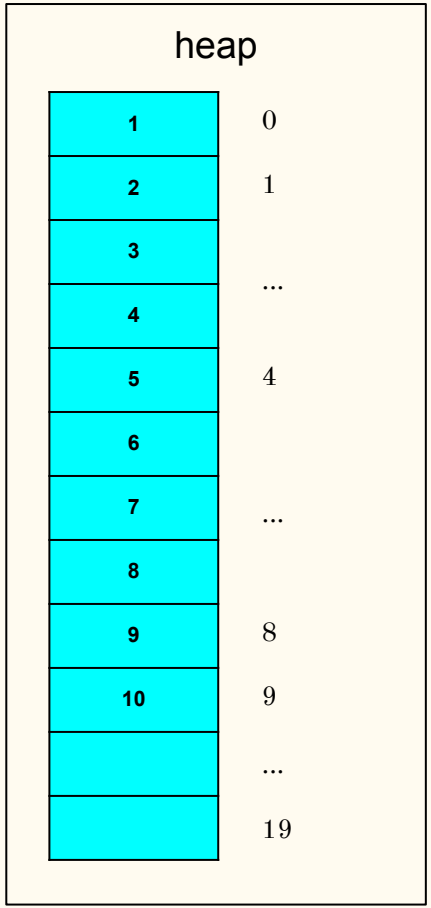


Array limitations

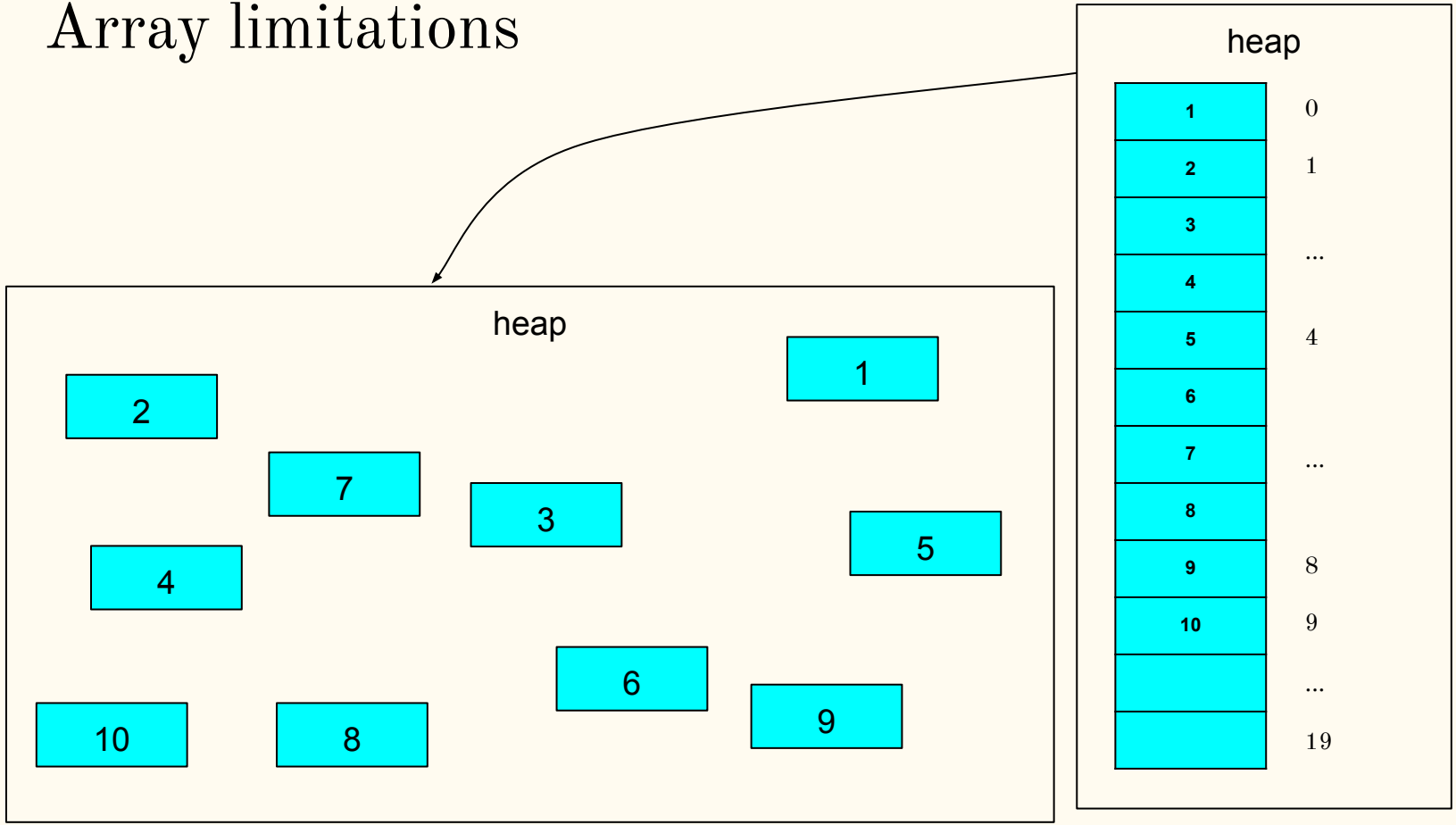
Arrays arranged in contiguous memory

- advantage: allows for index-based access
- disadvantages
  - fixed size
  - copying
    - resizing
    - insertion
    - deletion

what if values need not be contiguous?

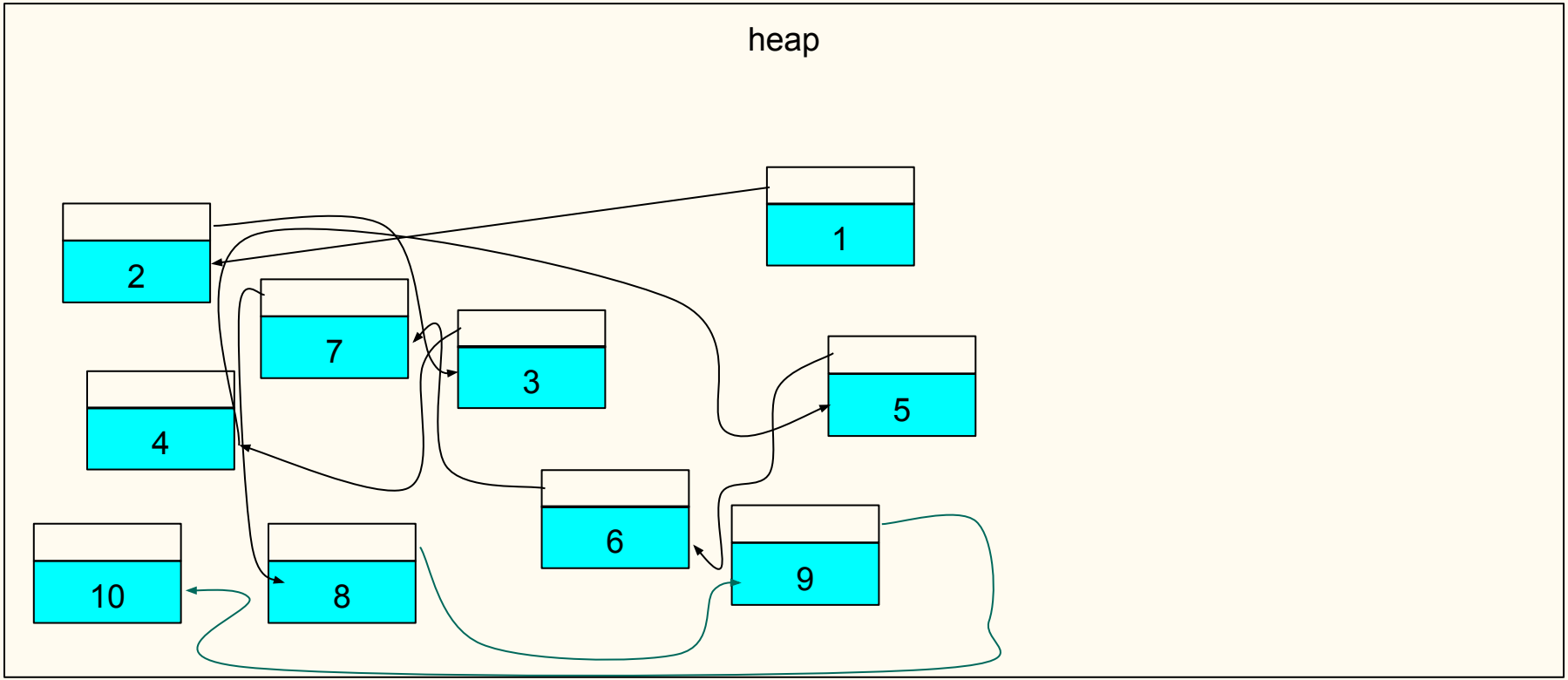


Array limitations

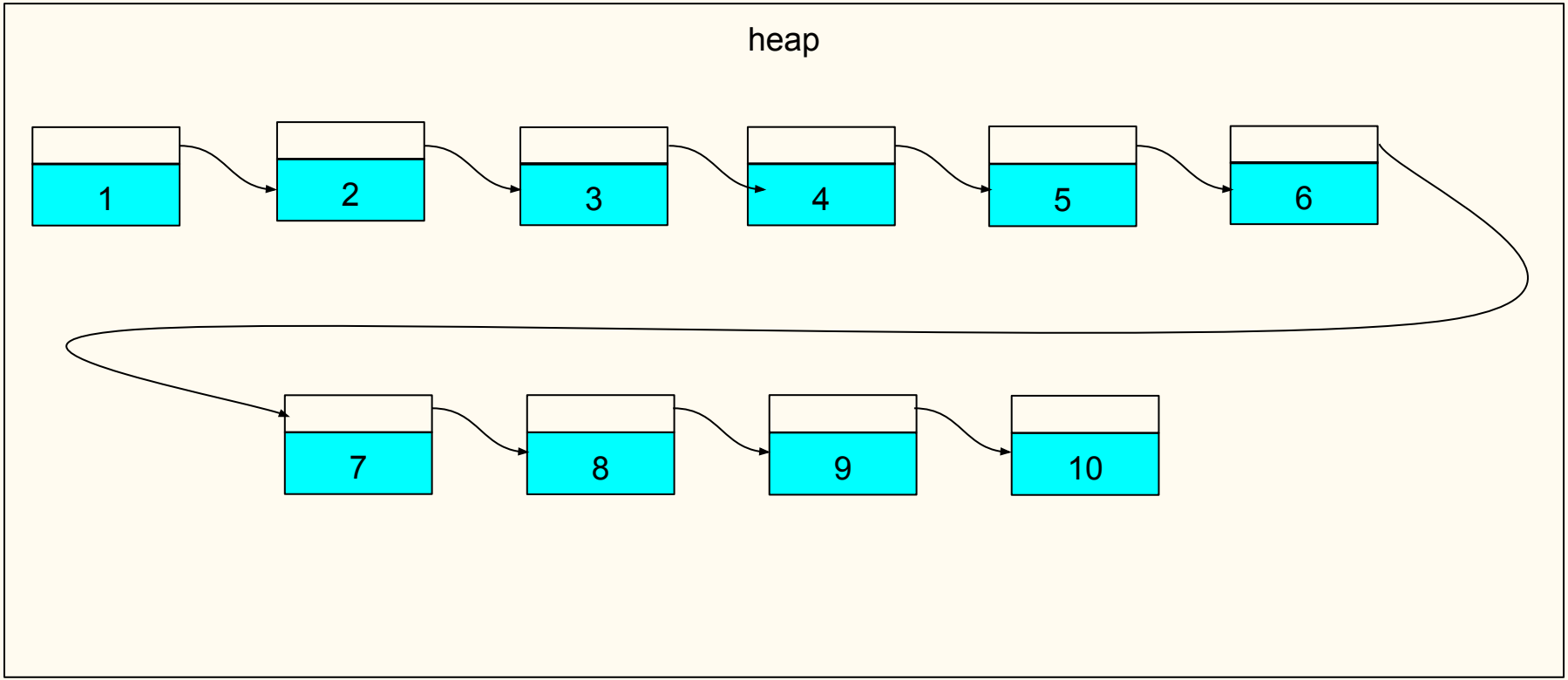


Linked lists

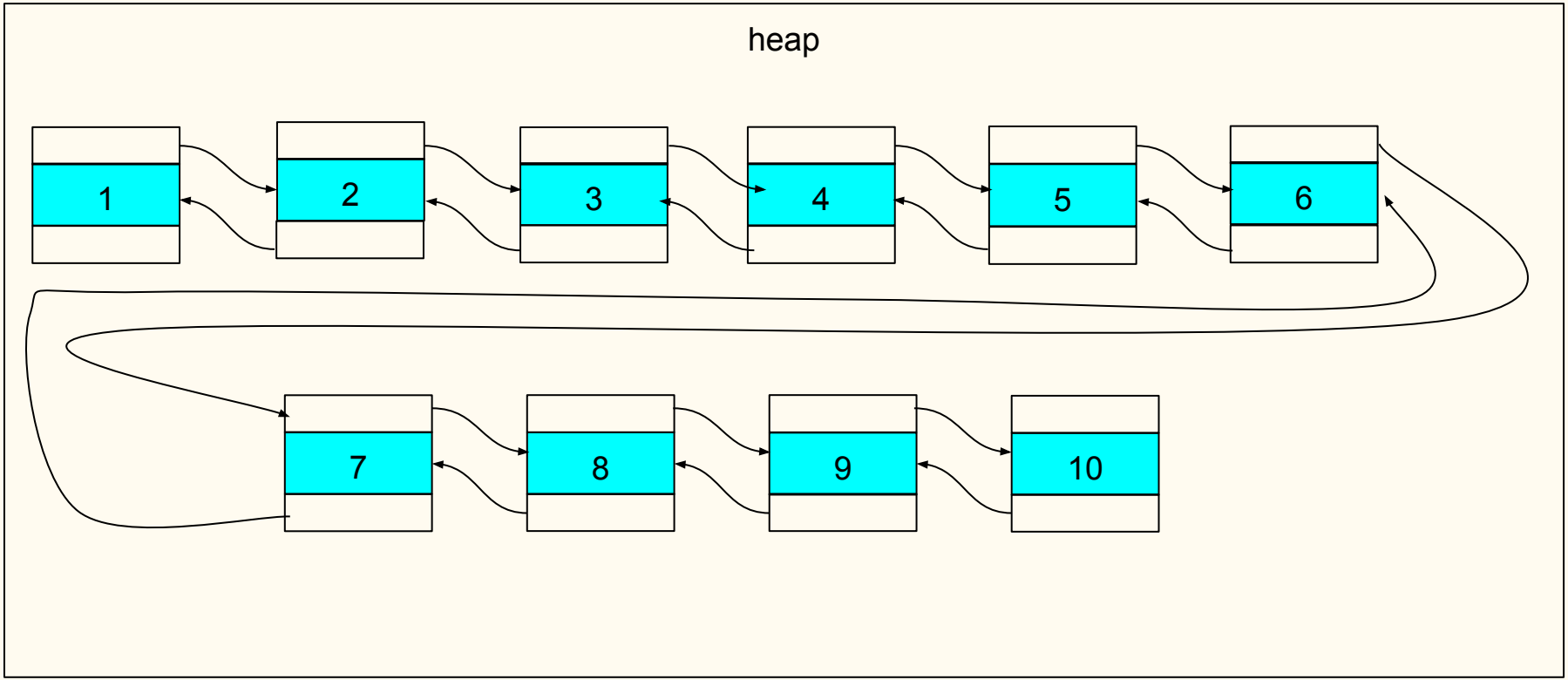
Linked lists



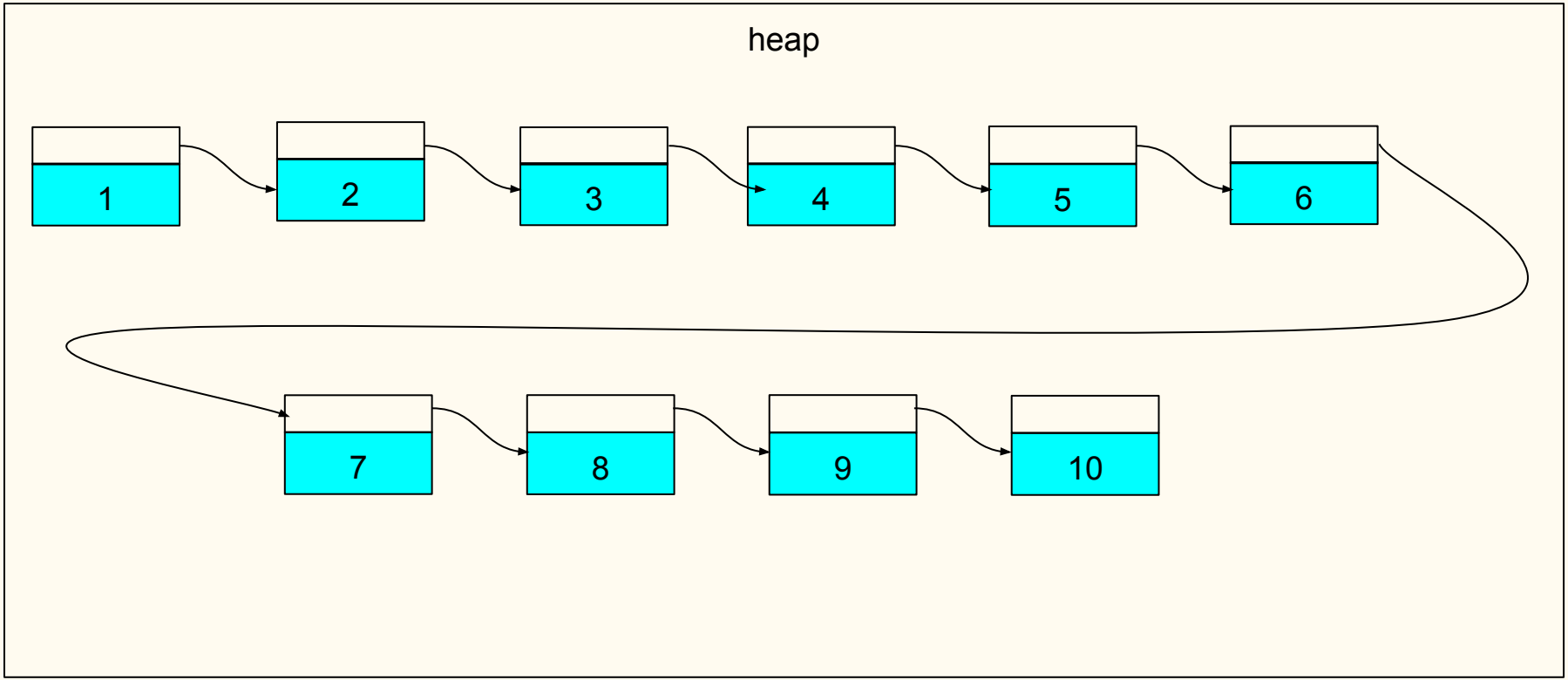
Singly linked list



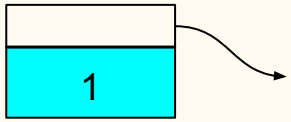
Doubly linked list



Singly linked list

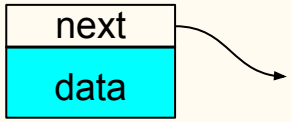


Code representations



How do we represent an element of the linked list in code?

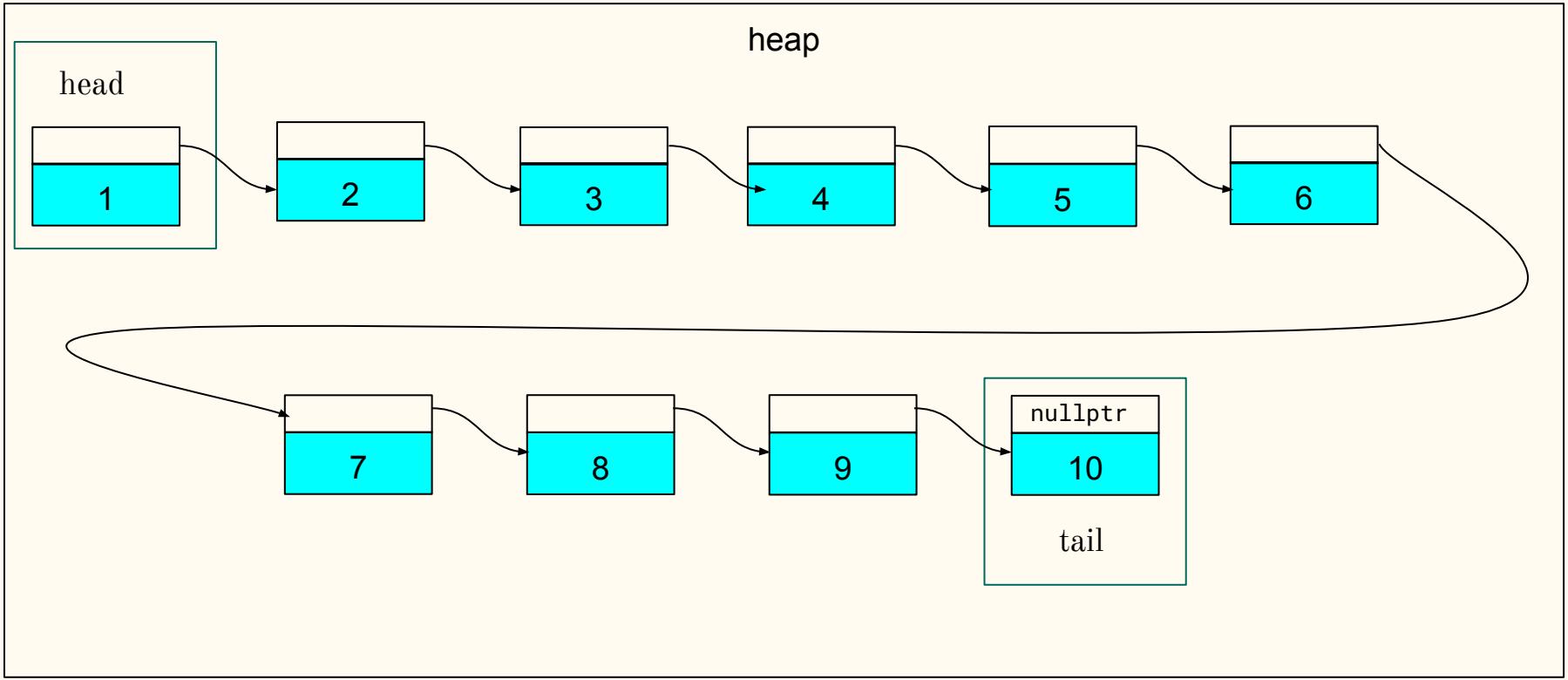
Code representations



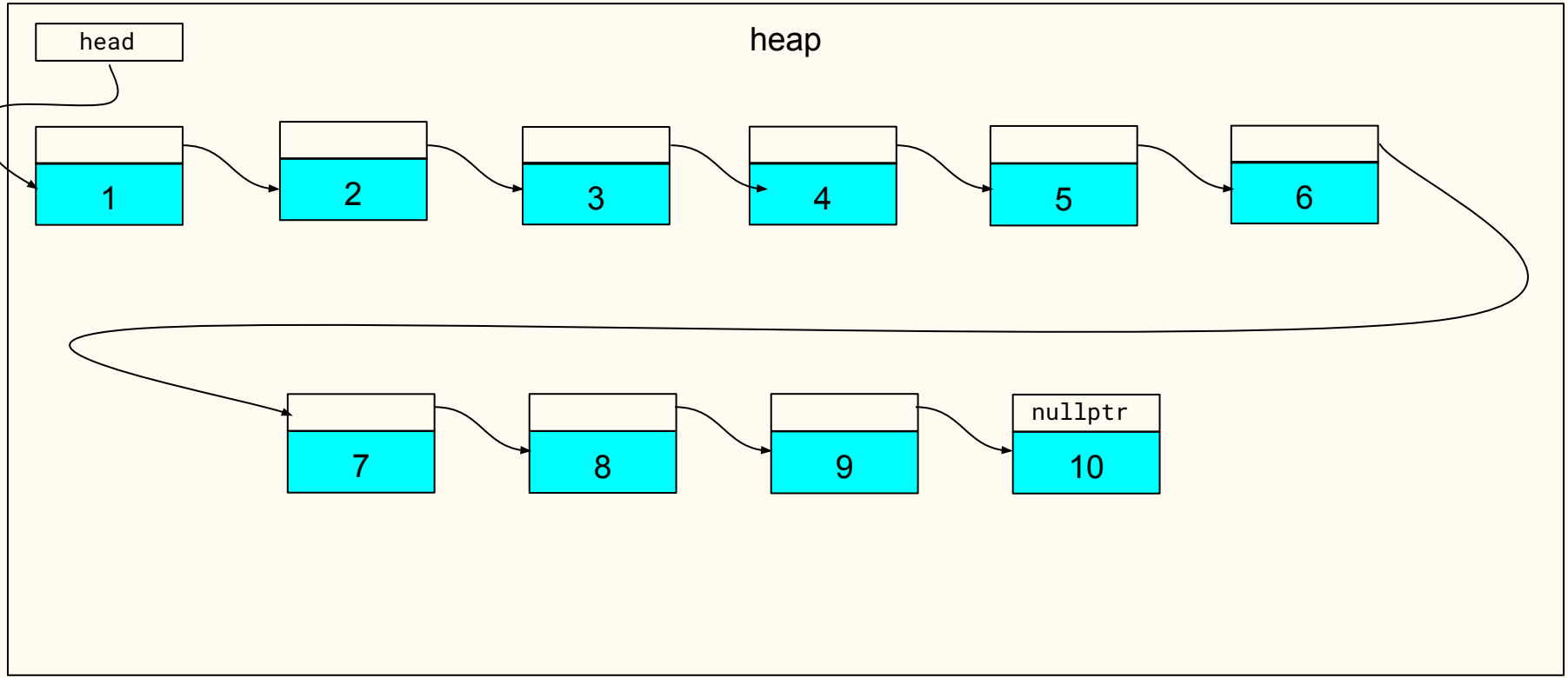

```
struct Node {
public:
    int data;
    Node* next;
};
```

A linked list toolkit

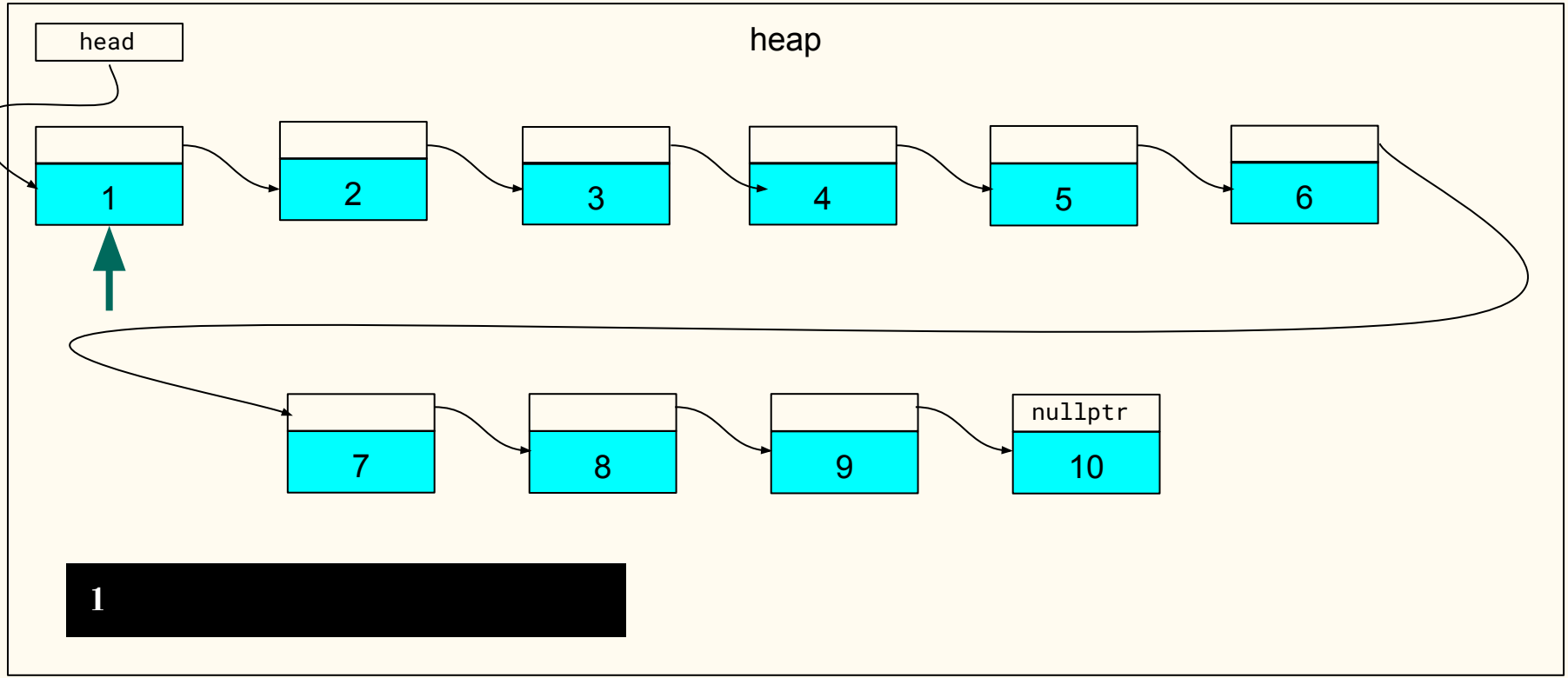
Singly linked list



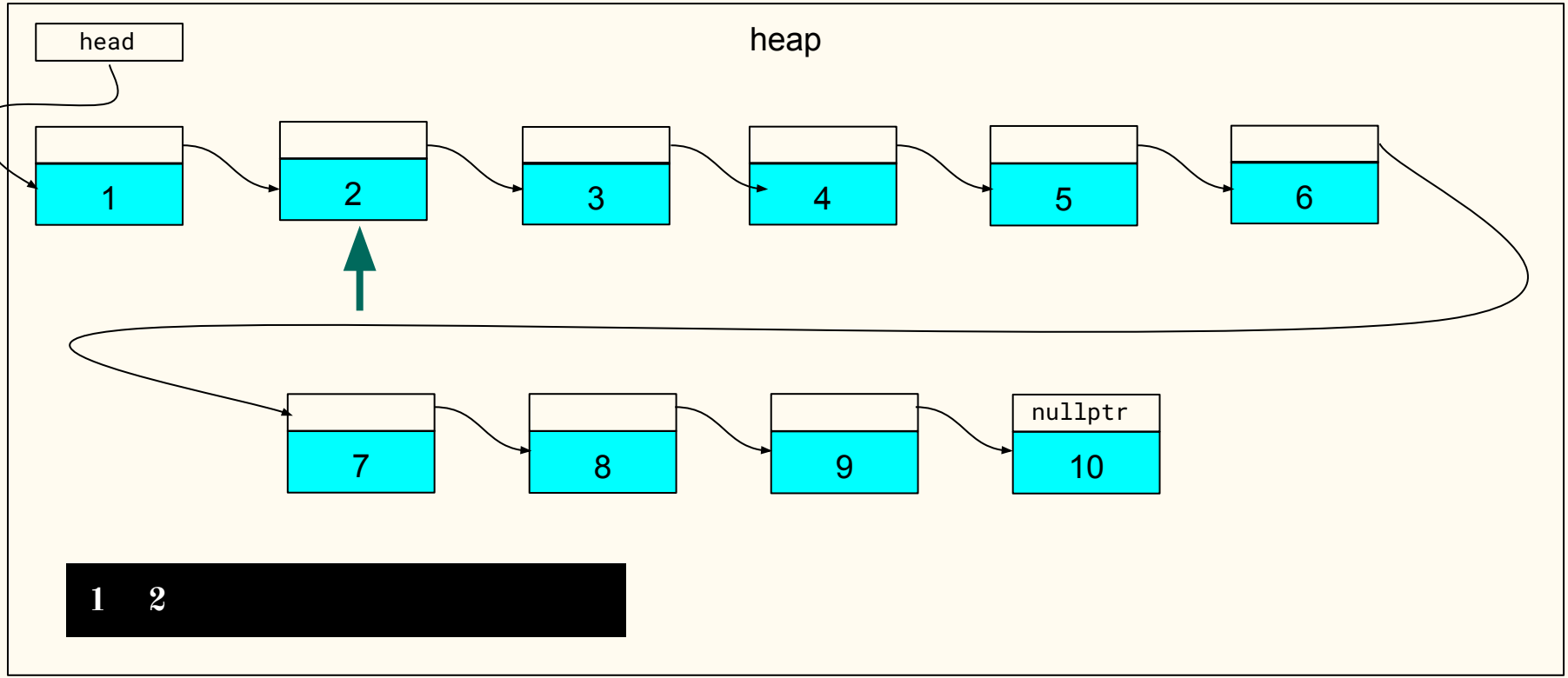
Singly linked list



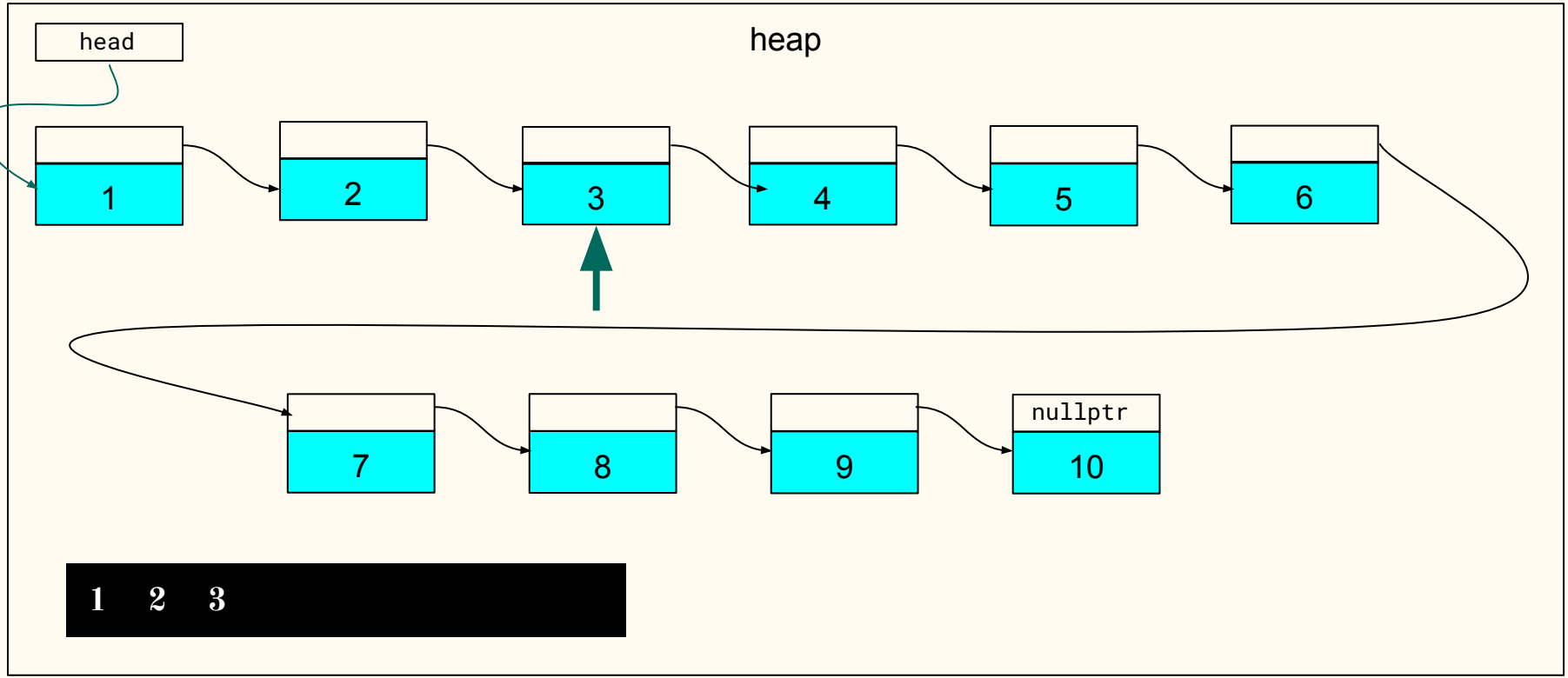
Displaying a linked list



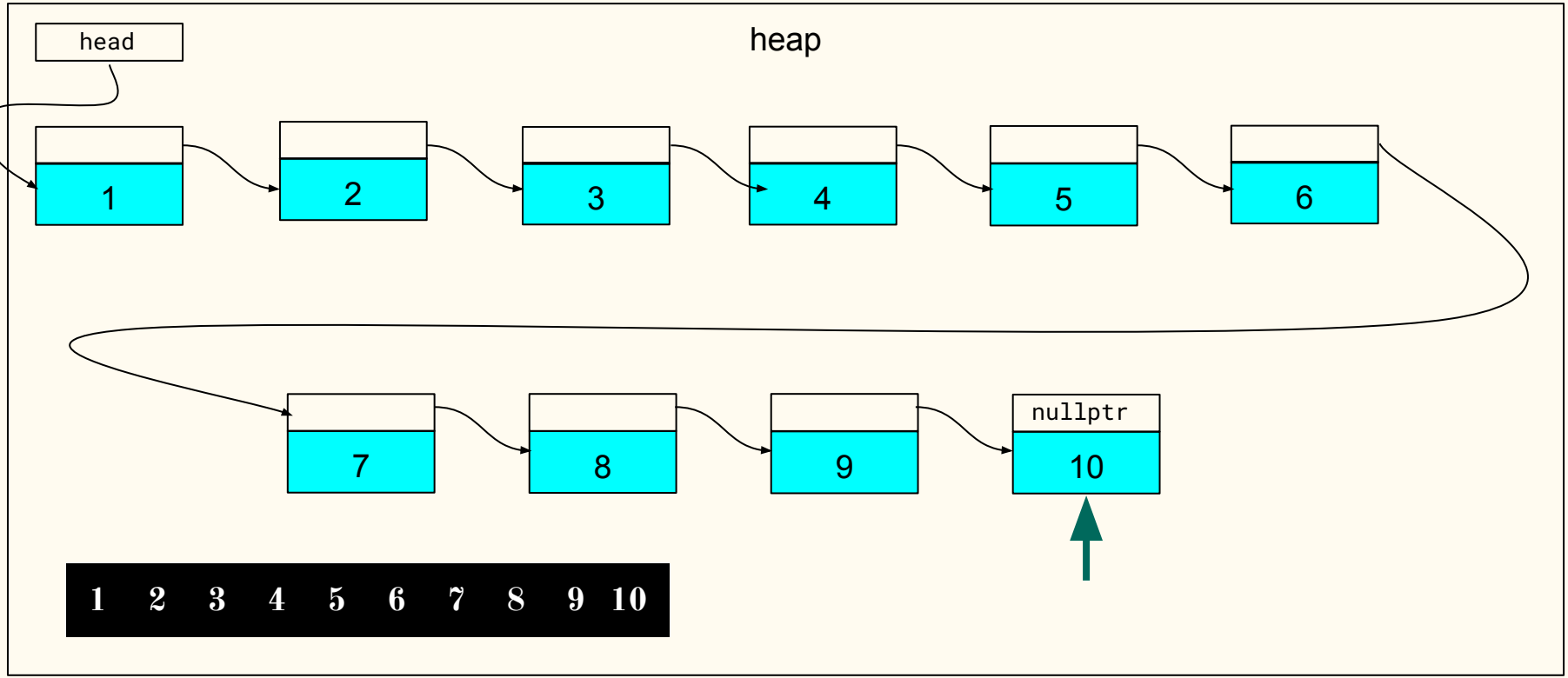
Displaying a linked list



Displaying a linked list



Displaying a linked list

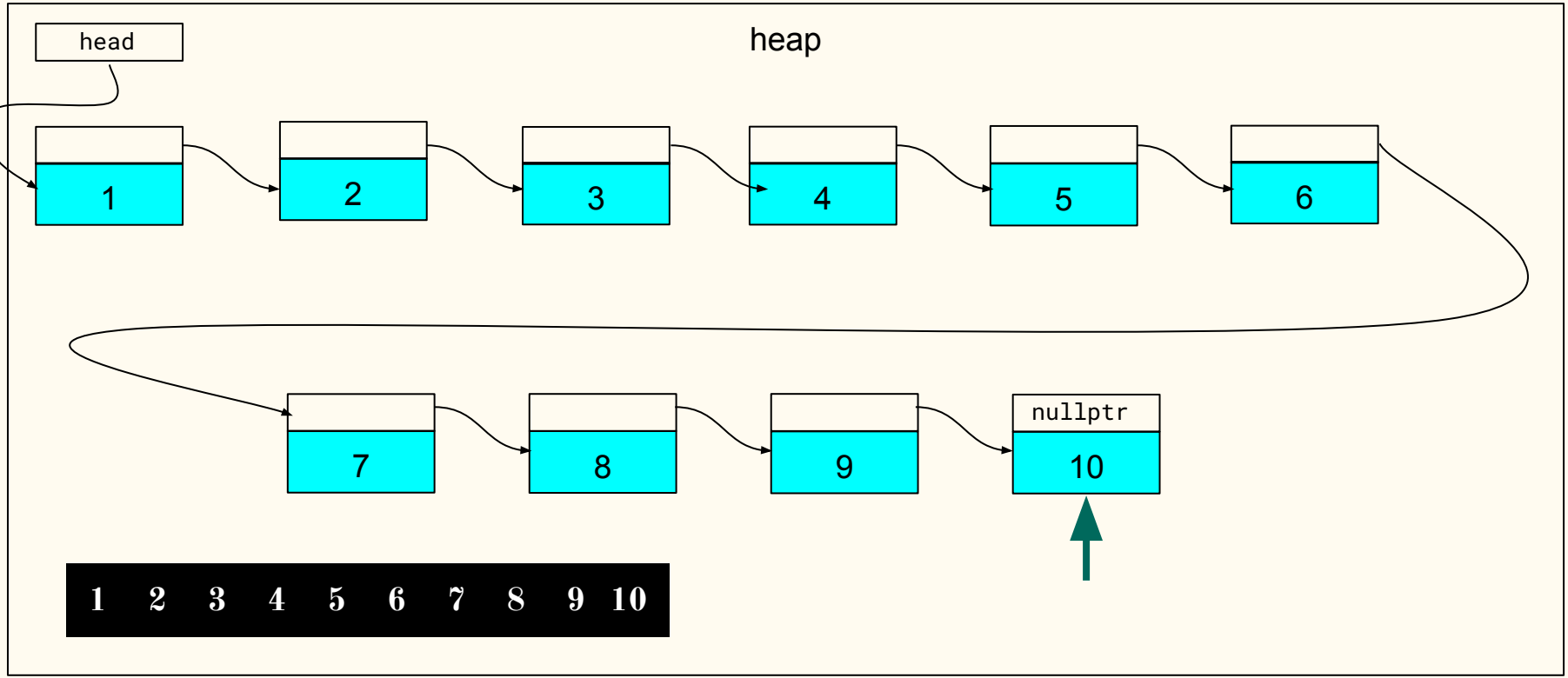


TurningPoint

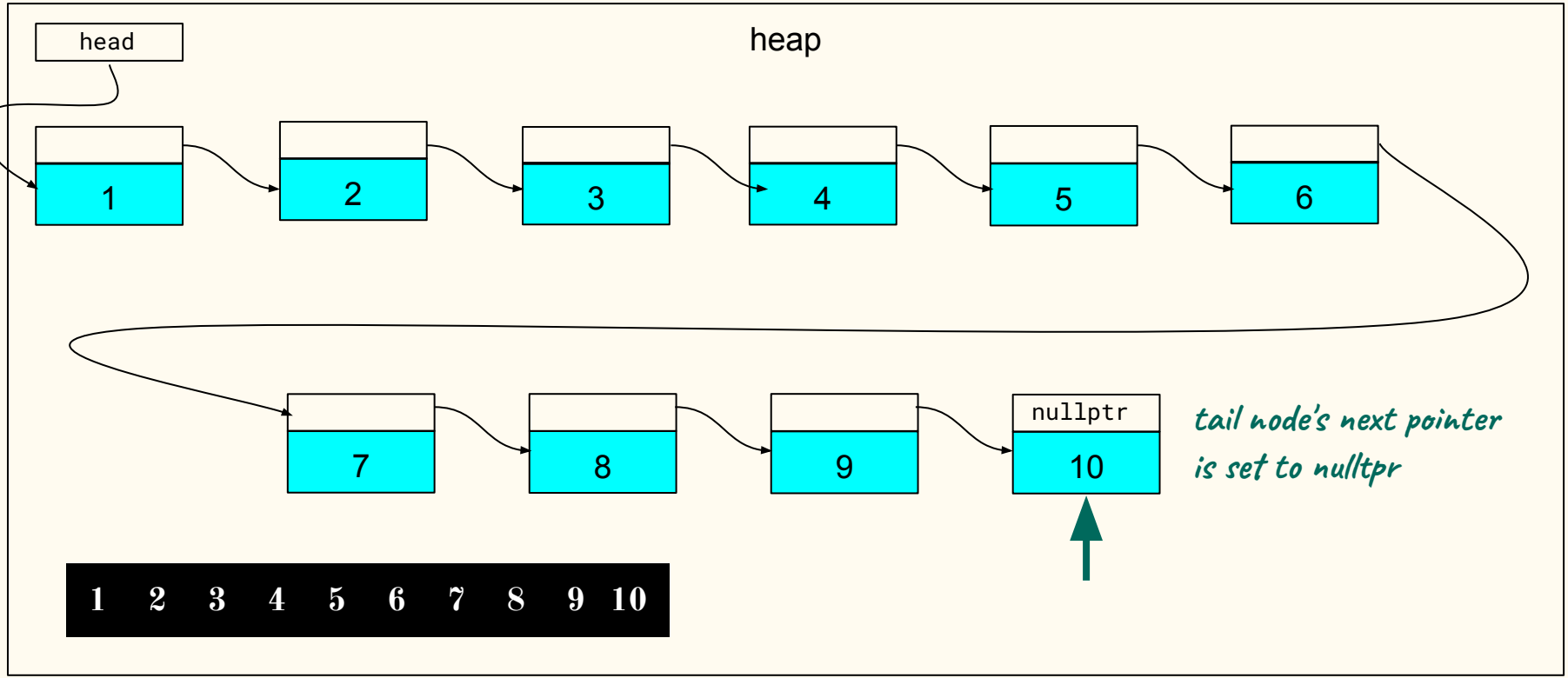
SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220411<A|D>

Replace <A|D> with this section's letter

Which condition indicates that no additional nodes need to be output?



Displaying a linked list



Displaying a linked list

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


void display_list(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    while (___) {
        cout << ptr->data << ' ';
        ptr = ___;
    }
    cout << endl;
}
```

Displaying a linked list

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


void display_list(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    while (_1_) {
        cout << ptr->data << ' ';
        ptr = ___;
    }
    cout << endl;
}
```

Which condition replaces blank #1 so that the loop terminates when the list has been fully traversed?

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


void display_list(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    while (_1_) {
        cout << ptr->data << ' ';
        ptr = ___;
    }
    cout << endl;
}
```

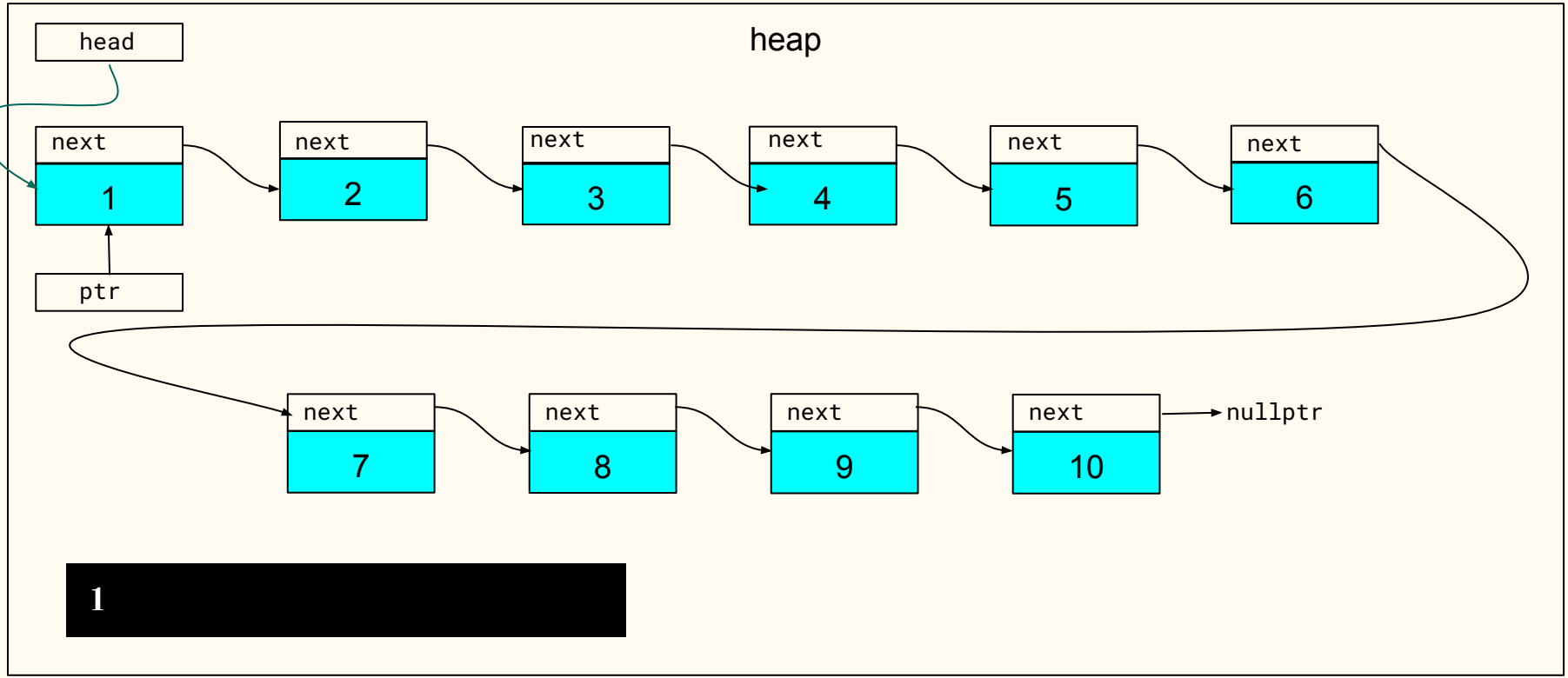
Displaying a linked list

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


void display_list(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    while (ptr != nullptr) {
        cout << ptr->data << ' ';
        ptr = ___;
    }
    cout << endl;
}
```

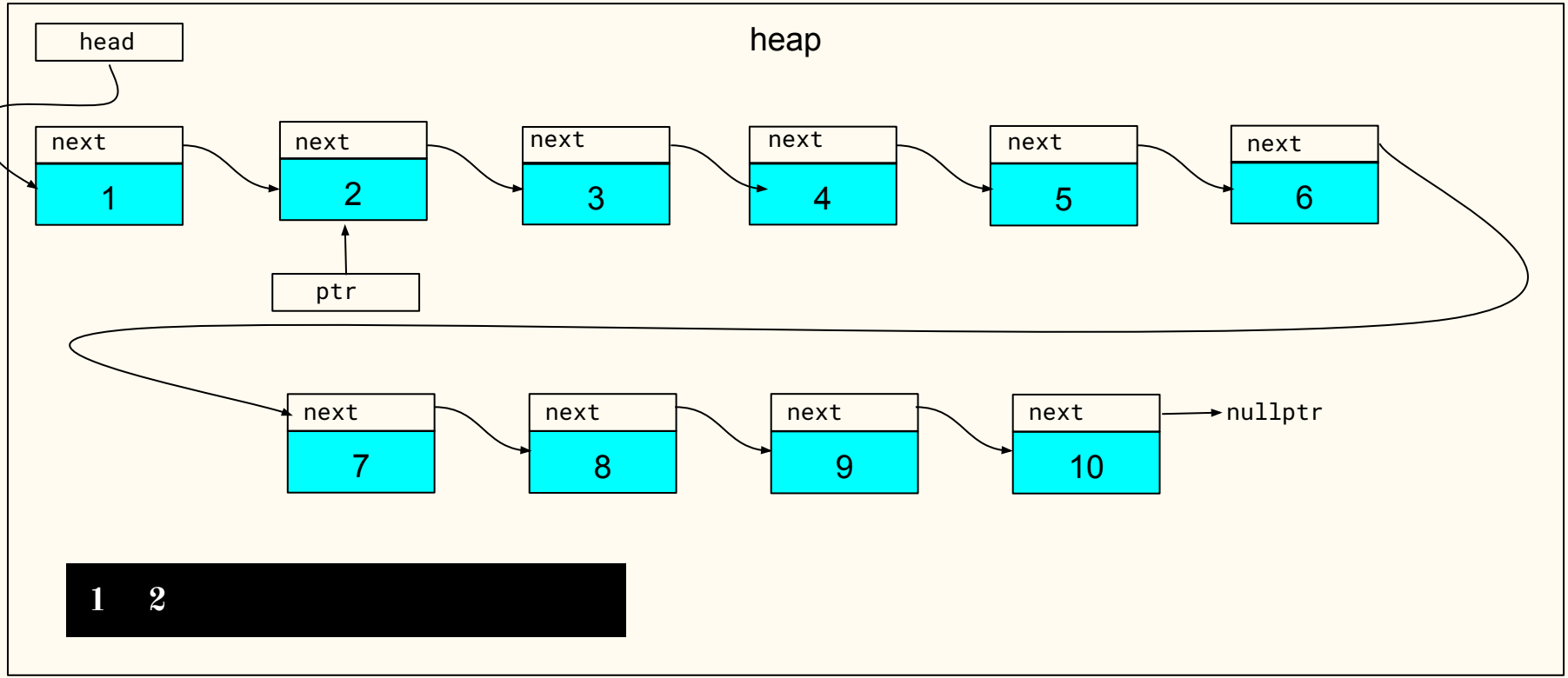
Displaying a linked list

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


void display_list(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    while (ptr != nullptr) {
        cout << ptr->data << ' ';
        ptr = _2_;
    }
    cout << endl;
}
```

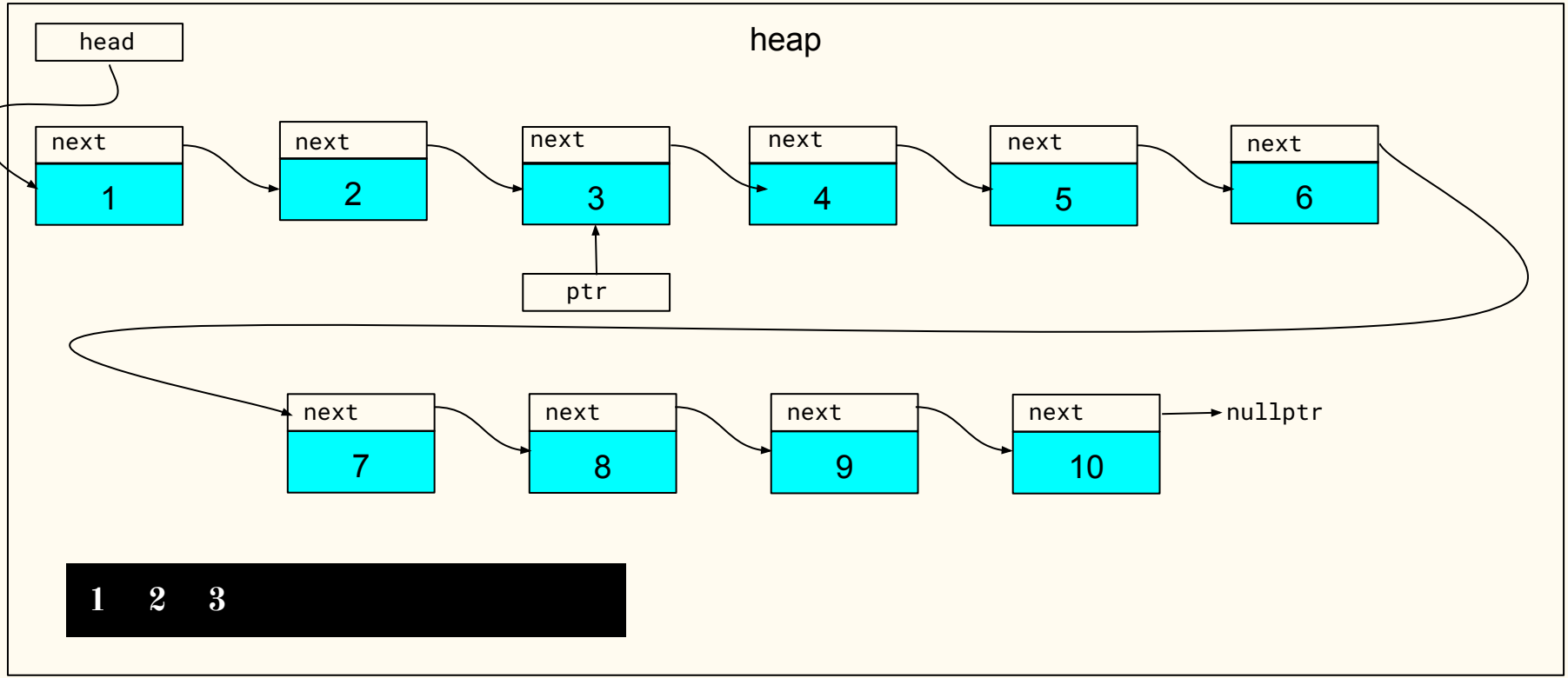
Displaying a linked list



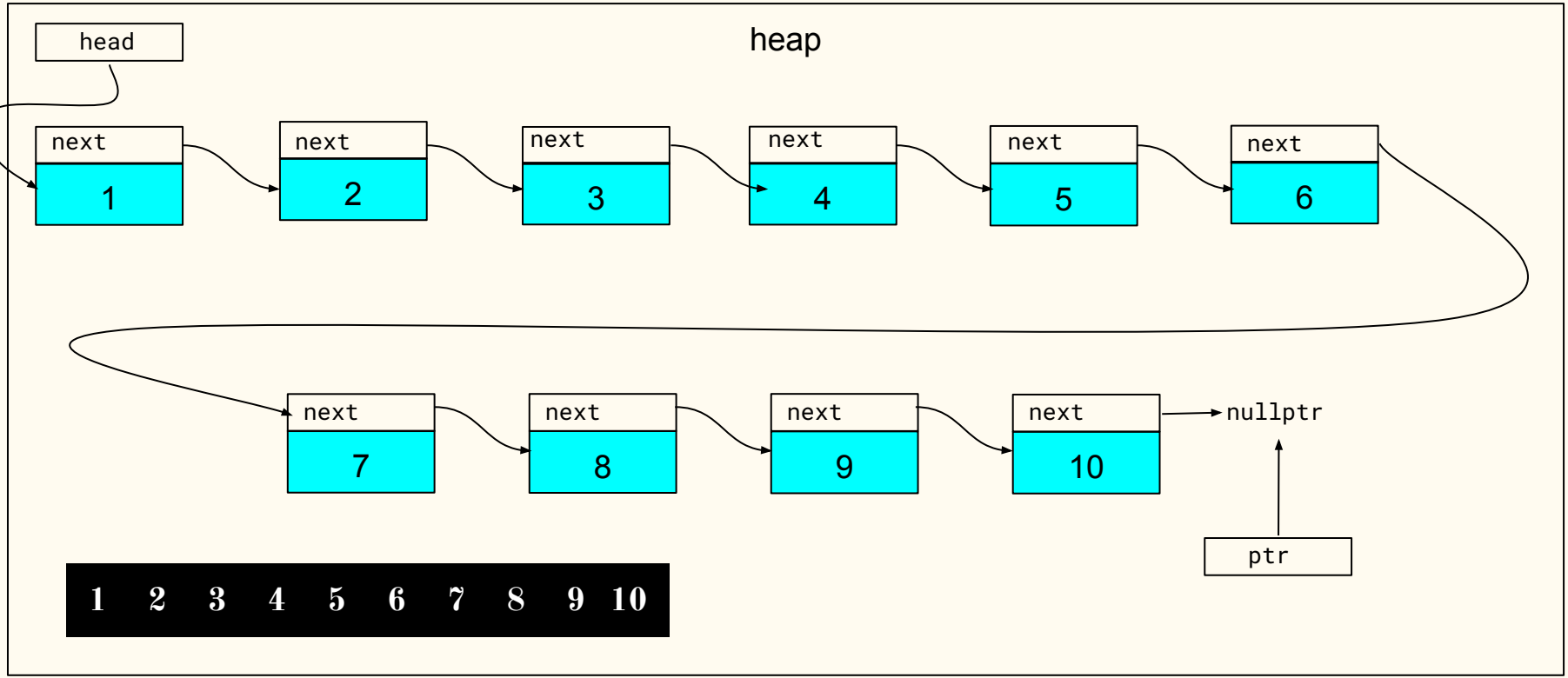
Displaying a linked list



Displaying a linked list



Displaying a linked list



Which expression replaces blank #2 in order to use `ptr` to traverse the list?

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


void display_list(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    while (ptr != nullptr) {
        cout << ptr->data << ' ';
        ptr = _2_;
    }
    cout << endl;
}
```

Displaying a linked list

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


void display_list(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    while (ptr != nullptr) {
        cout << ptr->data << ' ';
        ptr = ptr->next;
    }
    cout << endl;
}
```

Calculating a list's length

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = ___;









}
```

Calculating a list's length

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = _3_;









}
```

Which address do we assign to `ptr` (replacing blank #3) to traverse the list whose head `Node` is located at the address pointed at by `head_ptr`?

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = _3_;









}
```

Calculating a list's length

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        ---

    }

}
```

Calculating a list's length

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        _4_

    }

}
```

Which statement replaces blank #4 to update the current length of the list on each iteration of the while loop?

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        _4_

    }

}
```

Calculating a list's length

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        ++counter;
        ---
    }

}
```

Calculating a list's length

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        ++counter;
        _5_
    }

}
```

Which statement replaces blank #5 to point `ptr` at the next node in the list?

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        ++counter;
        _5_
    }

}
```

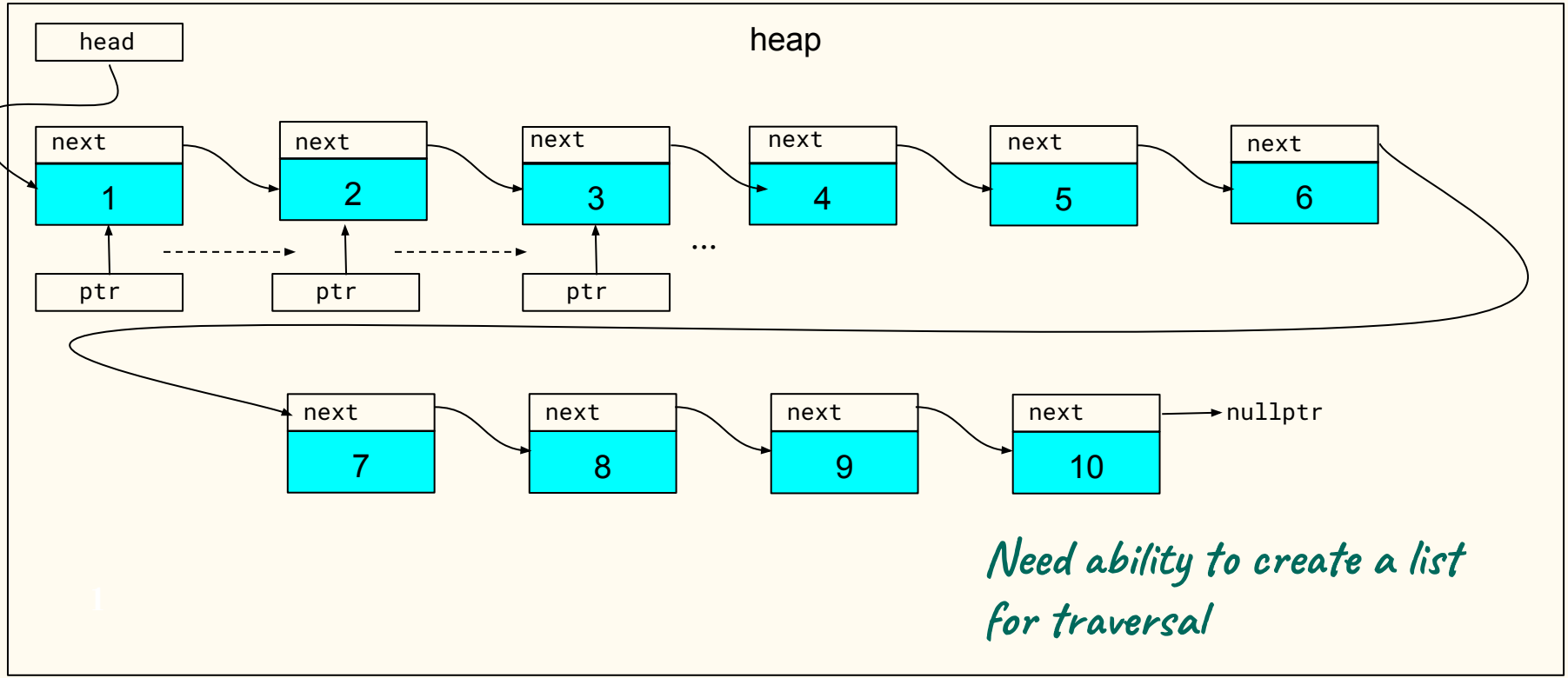
Calculating a list's length

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        ++counter;
        ptr = ptr->next;
    }
    return counter;
}
```

Calculating a list's length

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


int calc_list_length(const Node* head_ptr) {
    const Node* ptr = head_ptr;
    int counter = 0;
    while (ptr != nullptr) {
        ++counter;
        ptr = ptr->next;
    }
    return counter;
}
```

Building a list



Building a list

Start with a single node...

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

Building a list

Start with a single node...

```
struct Node {
    // define constructor
    int data;
    Node* next;
};
```

Building a list

Start with a single node...

```
struct Node {
    // define constructor
    ___
    int data;
    Node* next;
};
```

Building a list

Start with a single node...

```
struct Node {
    // define constructor
    Node() {}
    int data;
    Node* next;
};
```

Building a list

Start with a single node...

```
struct Node {
    // define constructor
    Node(int data, Node* next) {}
    int data;
    Node* next;
};
```

Building a list

Start with a single node...

```
struct Node {
    // define constructor
    Node(int data = ___, Node* next) {}
    int data;
    Node* next;
};
```

Building a list

Start with a single node...

```
struct Node {
    // define constructor
    Node(int data = _6_, Node* next) {}
    int data;
    Node* next;
};
```

Building a list

Start with a single node...

```
struct Node {
    // define constructor
    Node(int data = 0, Node* next) {}
    int data;
    Node* next;
};
```

Building a list

Start with a single node...

```
struct Node {
    // define constructor
    Node(int data = 0, Node* next = nullptr) {}
    int data;
    Node* next;
};
```

Building a list

Start with a single node...

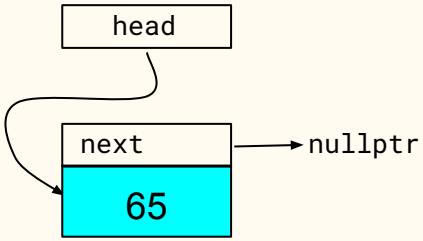
```
struct Node {
    // define constructor
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

Building a list

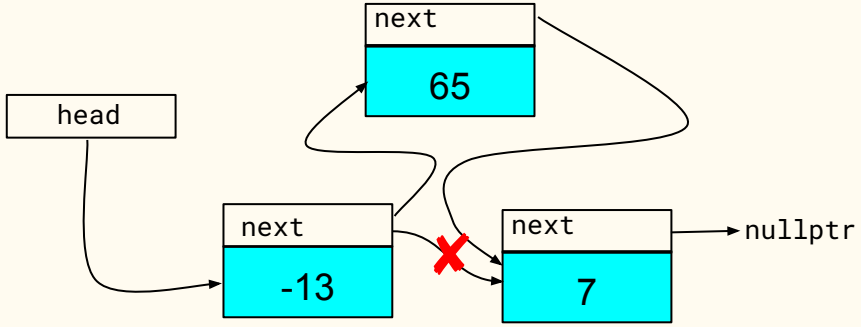
Start with a single node...

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

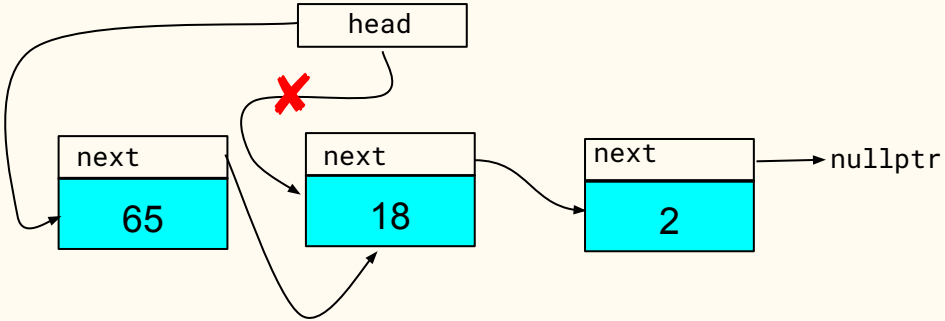
Building a list



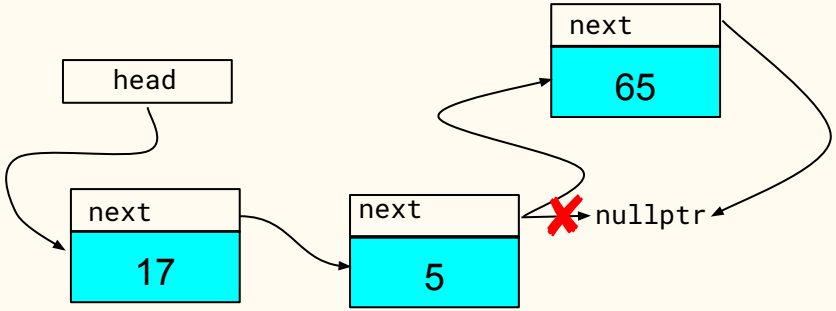
creating a list with one Node



inserting a Node into list

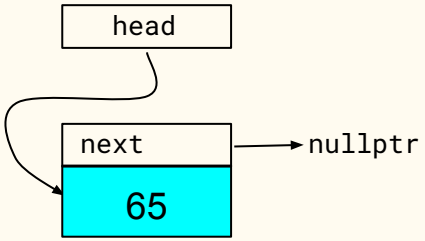


adding a Node to beginning of list



adding a Node to end of list

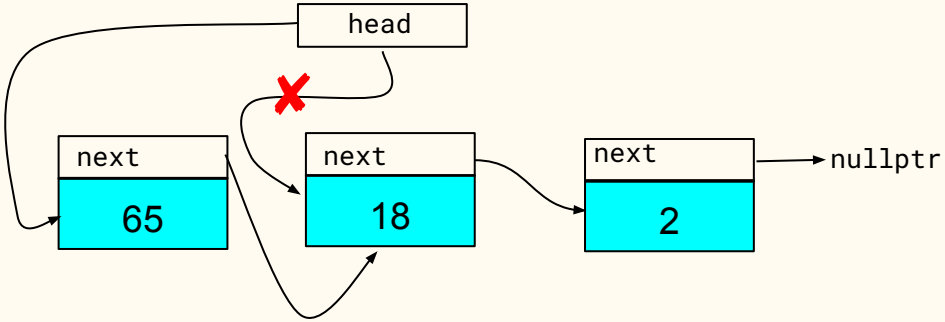
Building a list



creating a list with one Node

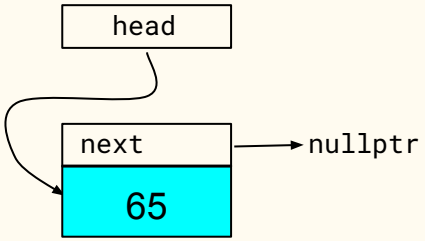
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_head_to_list(___) {

}
```



adding a Node to beginning of list

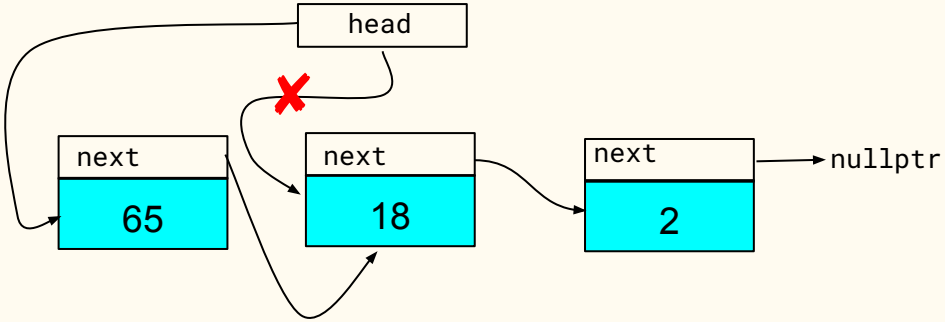
Building a list



creating a list with one Node

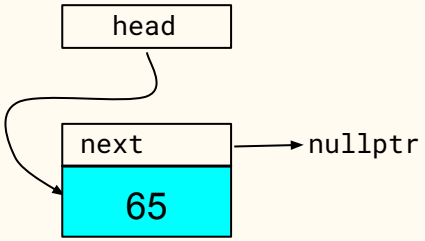
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_head_to_list(___, ___) {

}
```

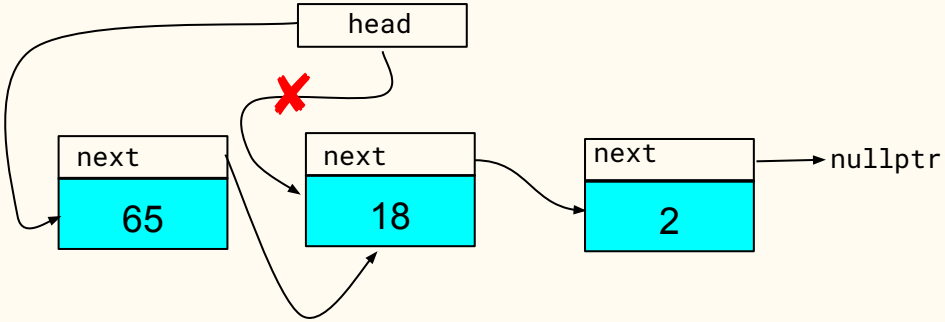


adding a Node to beginning of list

Building a list



creating a list with one Node

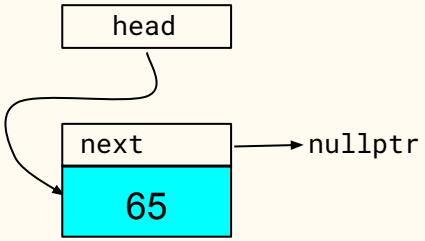


adding a Node to beginning of list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_head_to_list(___, int data) {

}
```

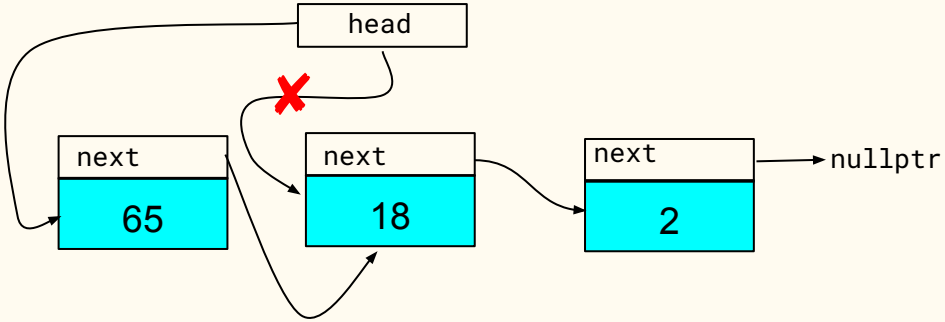
Building a list



creating a list with one Node

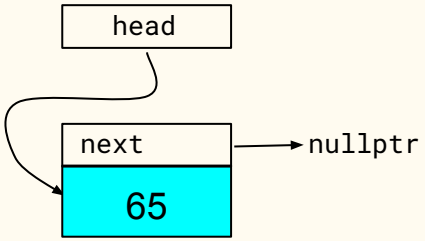
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_head_to_list(___ head_ptr, int data) {

}
```

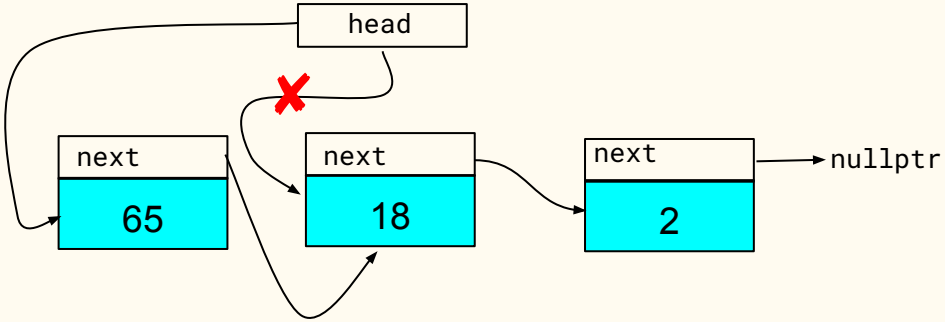


adding a Node to beginning of list

Building a list



creating a list with one Node

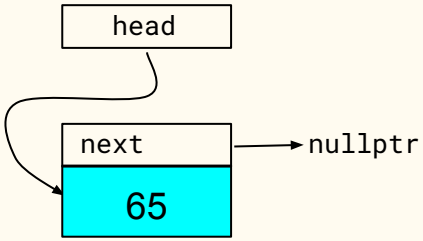


adding a Node to beginning of list

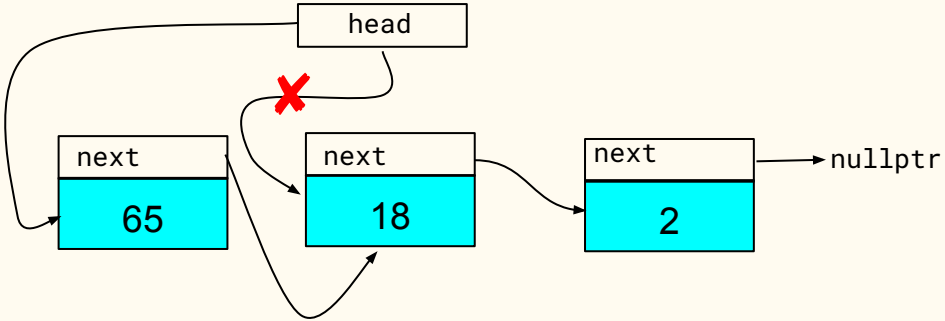
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
void add_head_to_list(_6_ head_ptr, int data) {

}
```

To ensure that we update the head pointer properly in `add_head_to_list`'s calling function, as which type should `head_ptr` be declared (replacing blank #6)?



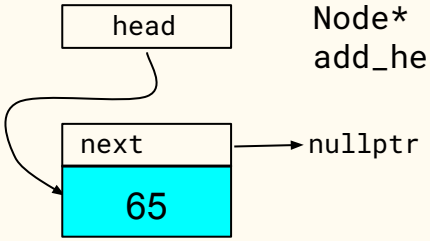
creating a list with one Node



adding a Node to beginning of list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};


void add_head_to_list(_6_ head_ptr, int data) {

}
```

Building a list



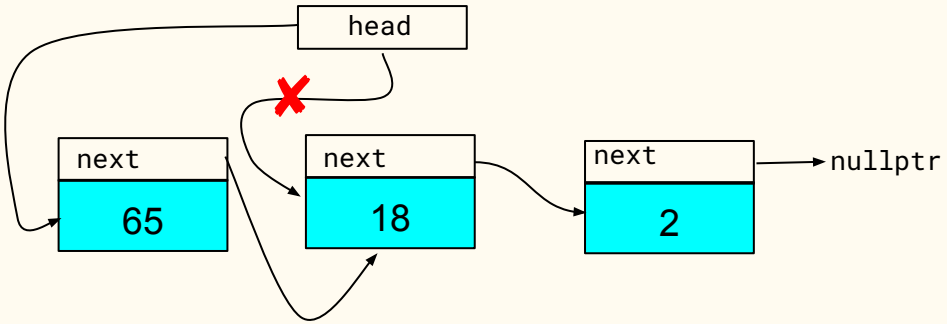

```
Node* head = nullptr;
add_head_to_list(head, 65);
```

creating a list with one Node

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

Node*& ensures changes reflected in calling function

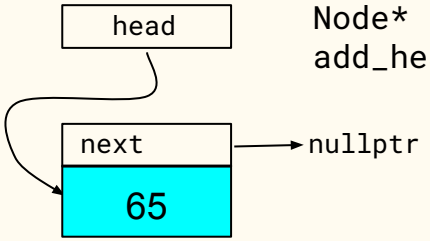
```
void add_head_to_list(Node*& head_ptr, int data) {
    // make a node for the data
    // have the node "point to" the old head
    // have the head_ptr point to the new node
}
```



adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

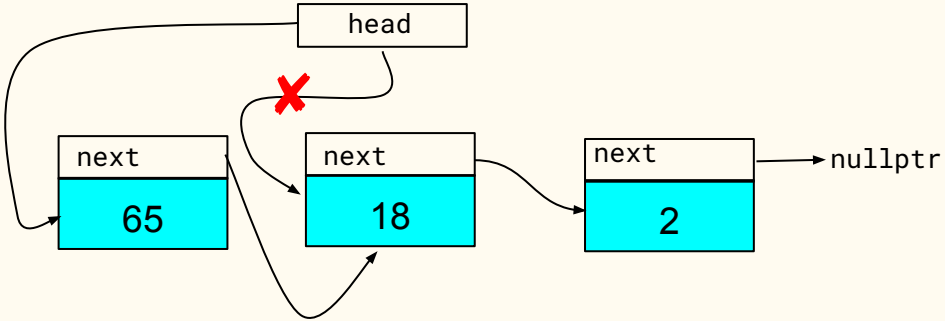
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

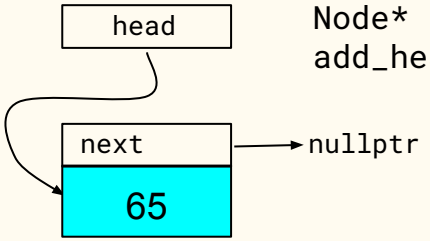



```
void add_head_to_list(Node*& head_ptr, int data) {
    // make a node for the data
    ___
    // have the node "point to" the old head
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

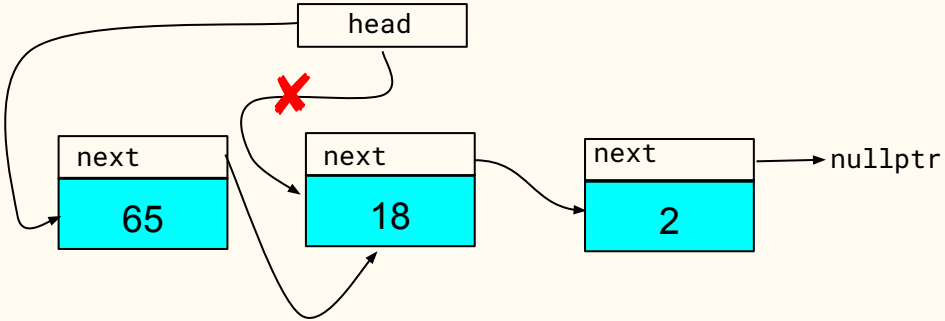
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

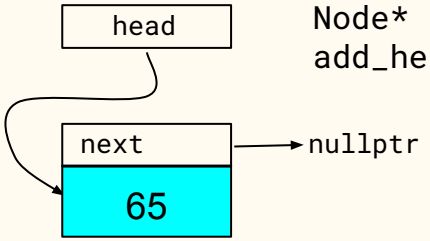



```
void add_head_to_list(Node*& head_ptr, int data) {
    // make a node for the data
    Node* ptr = ___;
    // have the node "point to" the old head
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

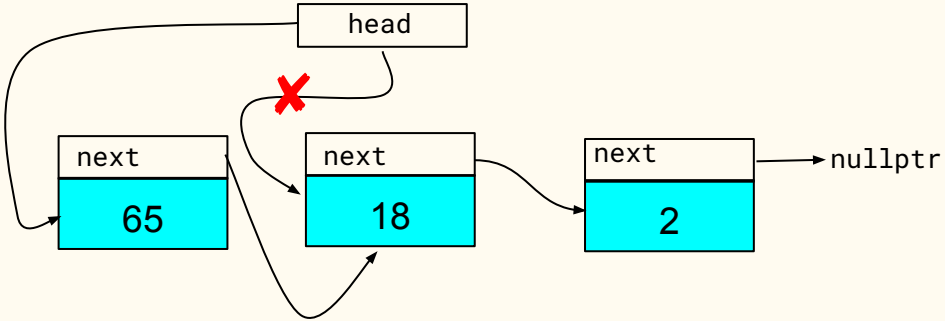
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

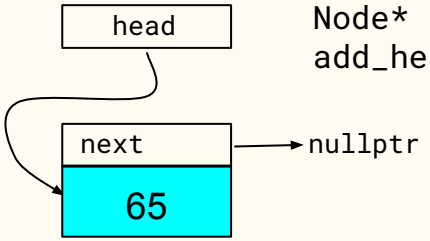



```
void add_head_to_list(Node*& head_ptr, int data) {
    // make a node for the data
    Node* ptr = _7_;
    // have the node "point to" the old head
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

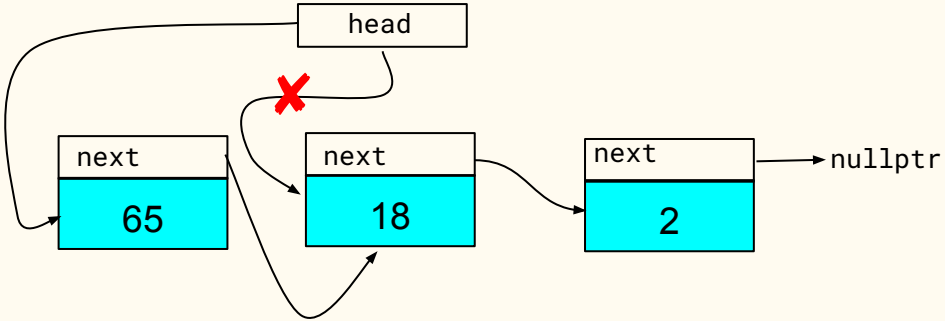
Which expression replaces blank #7 to instantiate a `Node` with `data` as the initial value of the `Node`'s `data` member?




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

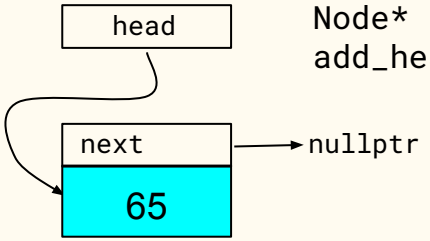



```
void add_head_to_list(Node*& head_ptr, int data) {
    // make a node for the data
    Node* ptr = _7_;
    // have the node "point to" the old head
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

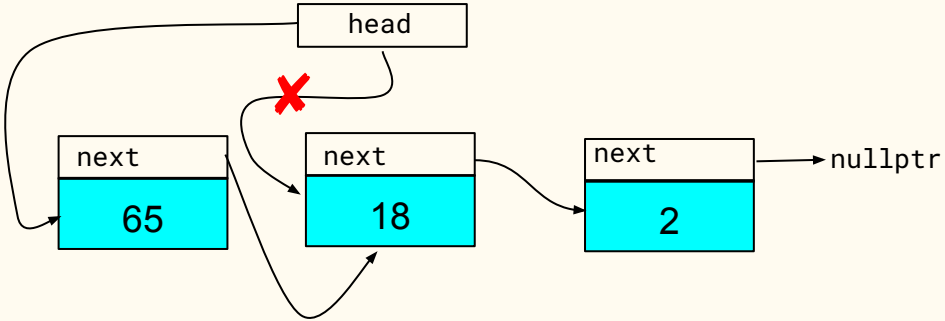
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

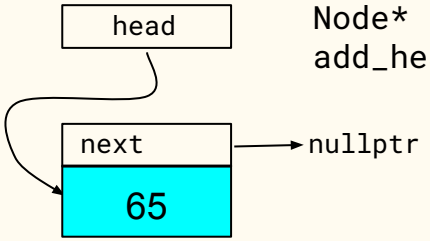



```
void add_head_to_list(Node*& head_ptr, int data) {
    // make a node for the data
    Node* ptr = new Node(data);
    // have the node "point to" the old head
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

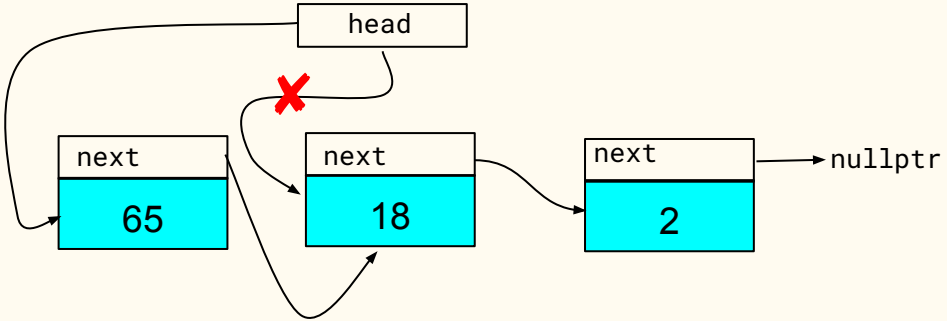
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

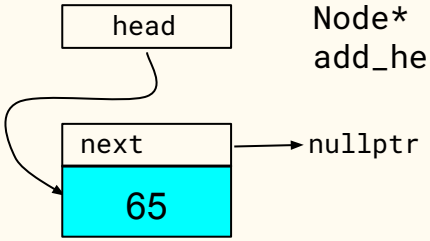



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    // have the node "point to" the old head
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

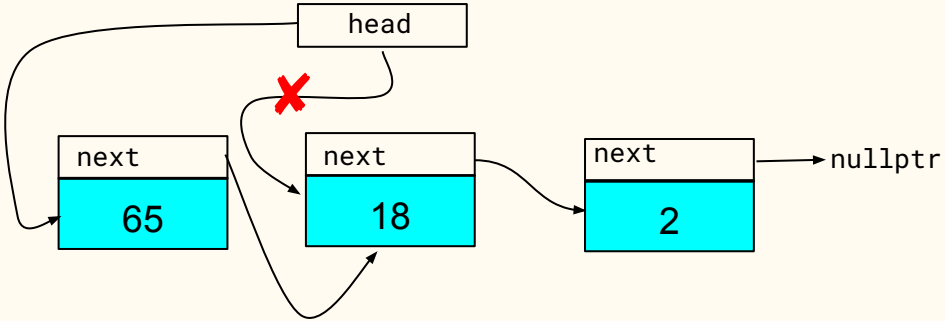
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

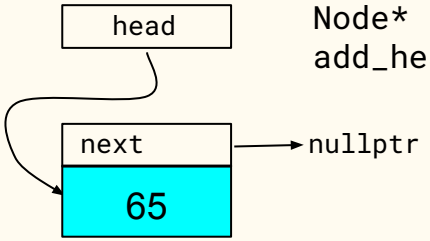



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    // have the node "point to" the old head
    ___
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

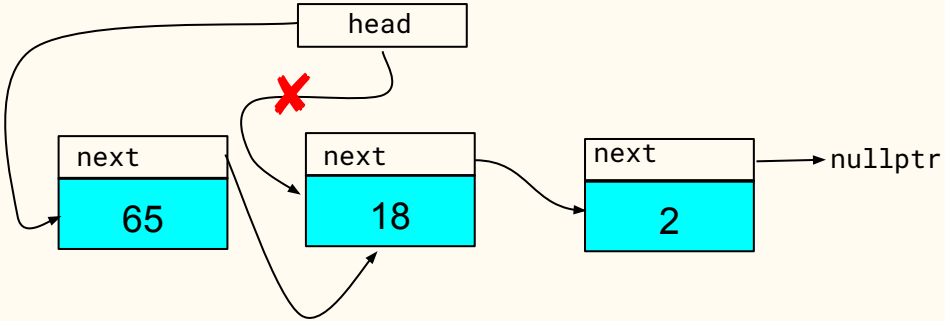
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

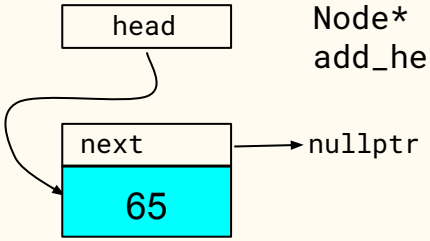



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    // have the node "point to" the old head
    _8_
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

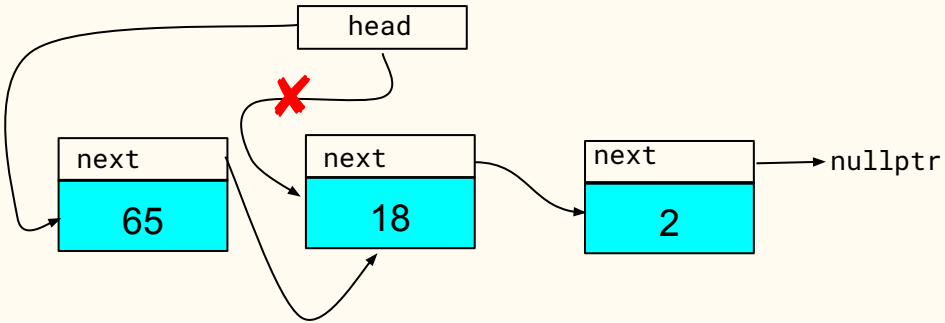
Which statement replaces blank #8 so that the old head `Node` of the list follows the new **Node**?




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

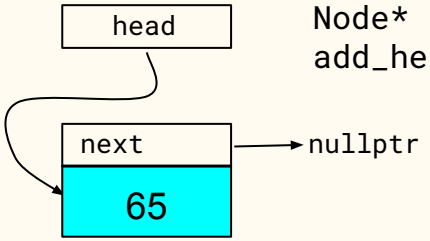



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    // have the node "point to" the old head
    _8_
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

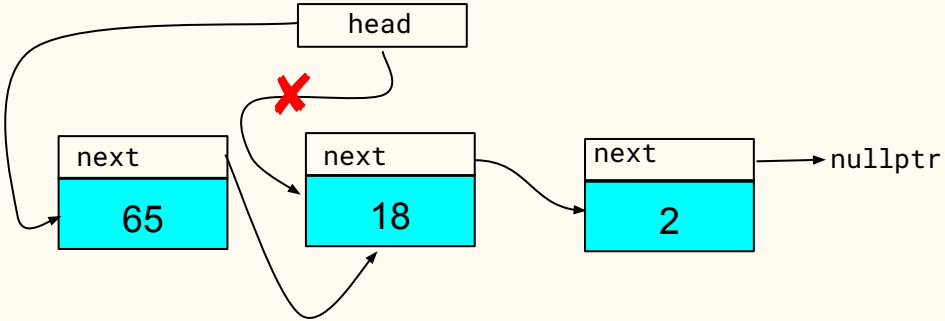
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

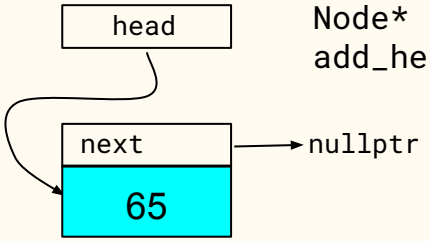



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    // have the node "point to" the old head
    ptr->next = head_ptr;
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

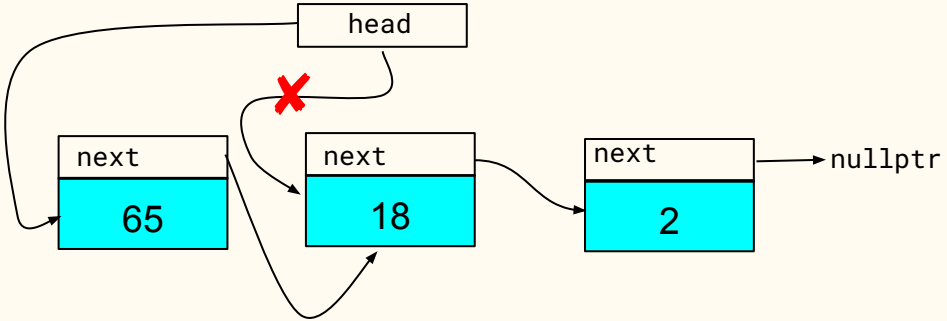
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

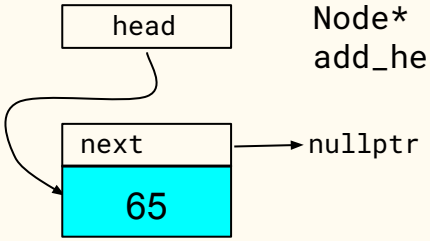



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    ptr->next = head_ptr;
    // have the head_ptr point to the new node
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

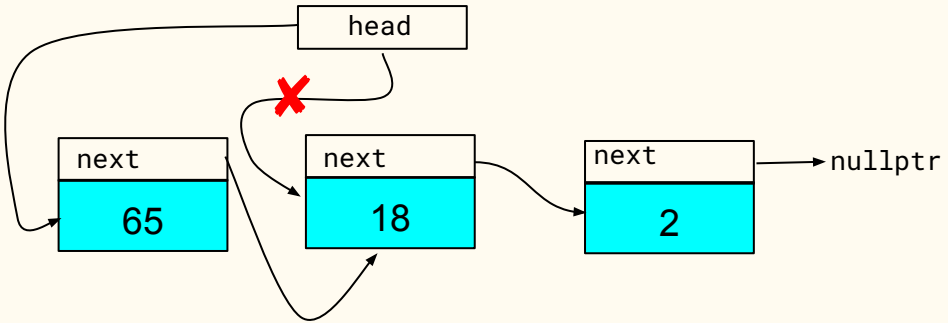
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

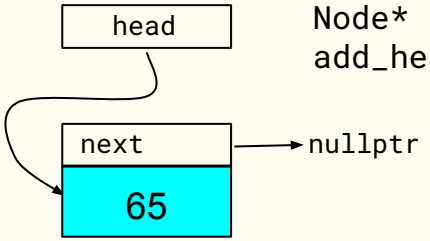



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    ptr->next = head_ptr;
    // have the head_ptr point to the new node
    ---
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

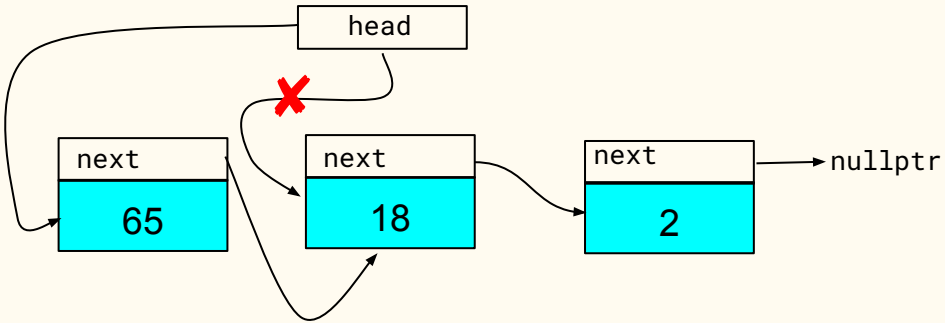
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

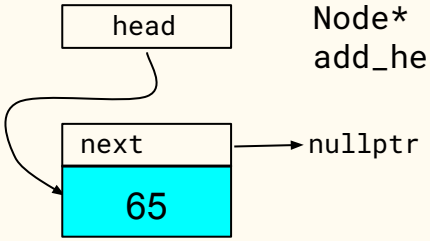



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    ptr->next = head_ptr;
    // have the head_ptr point to the new node
    _9_
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

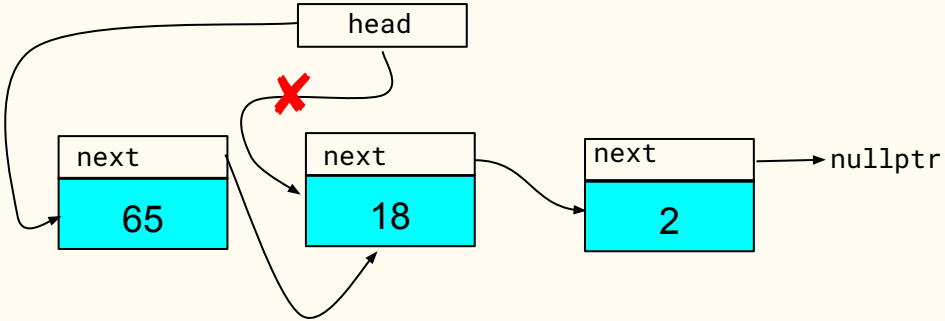
Which statement replaces blank #9 to point `head_ptr` to the newly created `Node`?




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

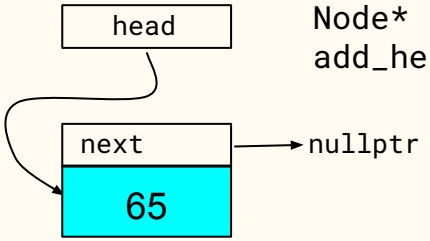



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    ptr->next = head_ptr;
    // have the head_ptr point to the new node
    _9_
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

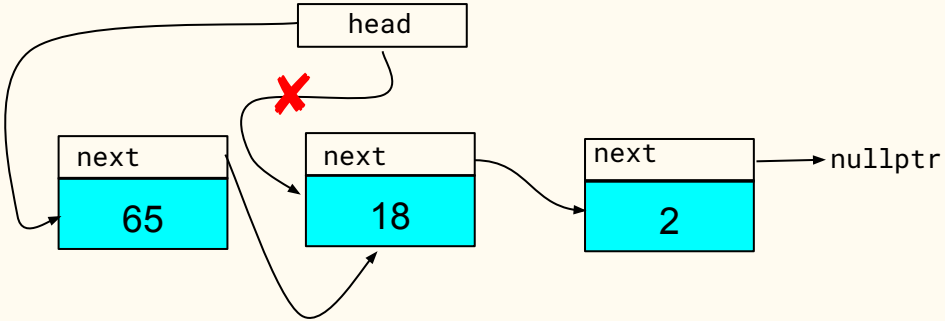
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

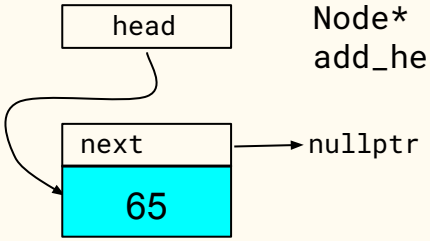



```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    ptr->next = head_ptr;
    // have the head_ptr point to the new node
    head_ptr  = ptr;
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

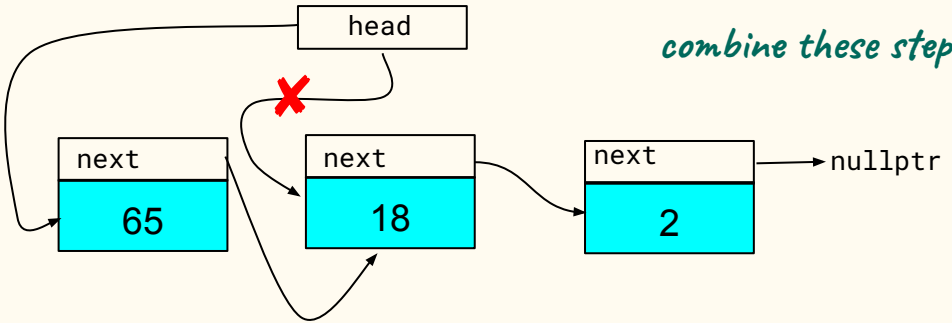
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node



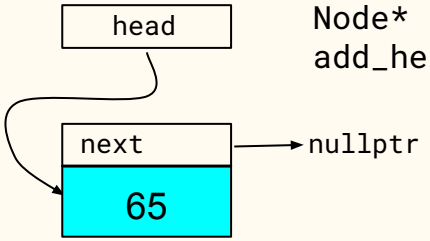
combine these steps

```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data);
    ptr->next = head_ptr;
    head_ptr  = ptr;
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

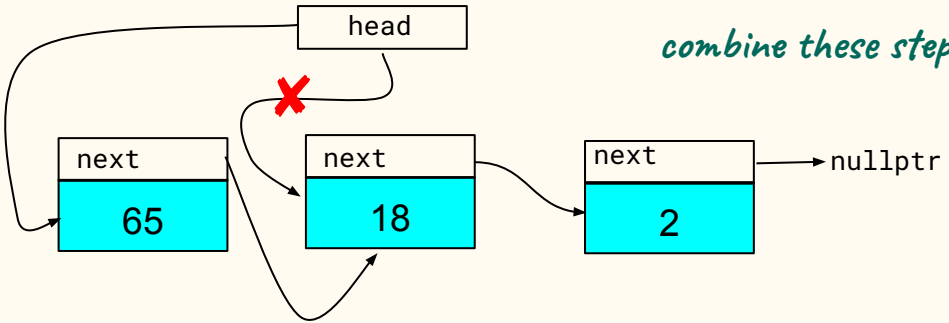
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node



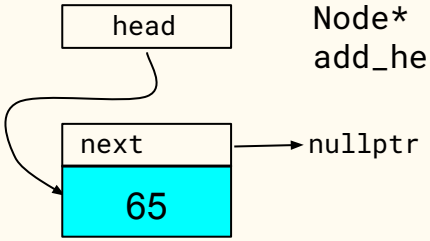
combine these steps

```
void add_head_to_list(Node*& head_ptr, int data) {
    Node* ptr = new Node(data, head_ptr);
    head_ptr  = ptr;
}
```

adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

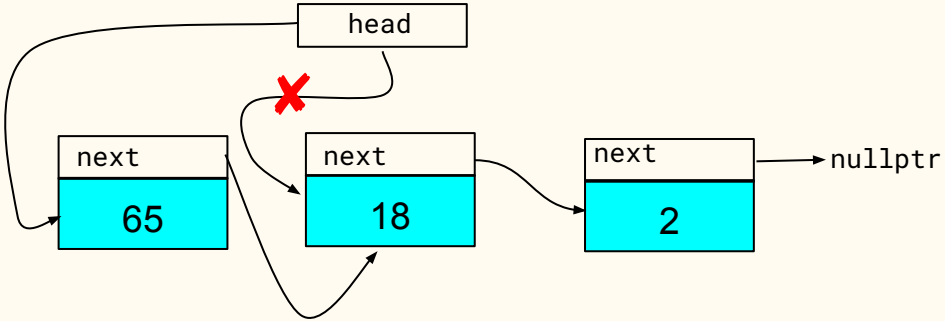
Building a list




```
Node* head = nullptr;
add_head_to_list(head, 65);
```

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};
```

creating a list with one Node

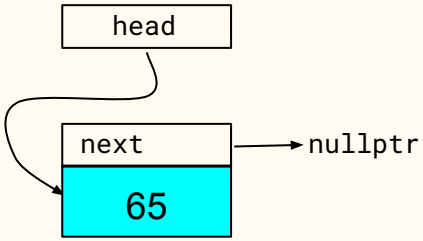



```
void add_head_to_list(Node*& head_ptr, int data) {
    head_ptr  = new Node(data, head_ptr);
}
```

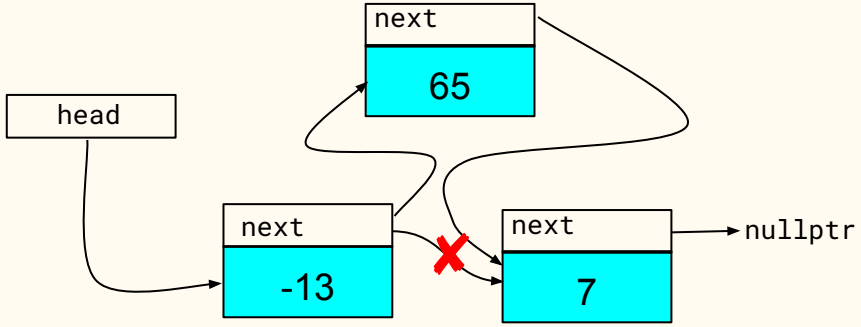
adding a Node to beginning of list

```
add_head_to_list(head, 65);
```

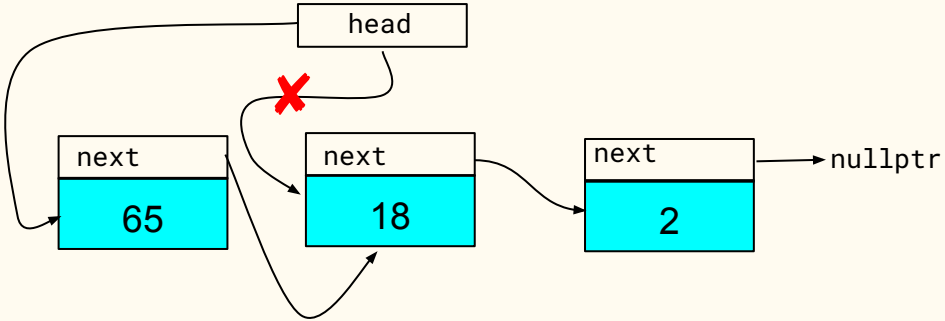
Building a list



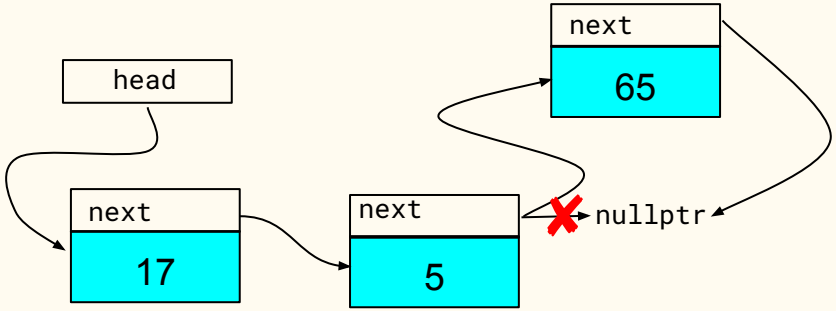
creating a list with one Node



inserting a Node into list



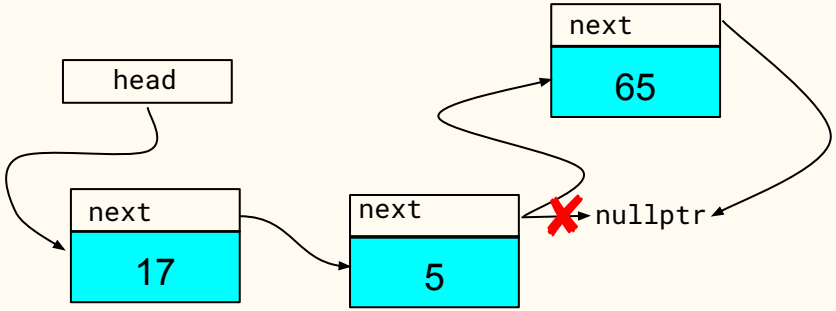
adding a Node to beginning of list



adding a Node to end of list

Building a list

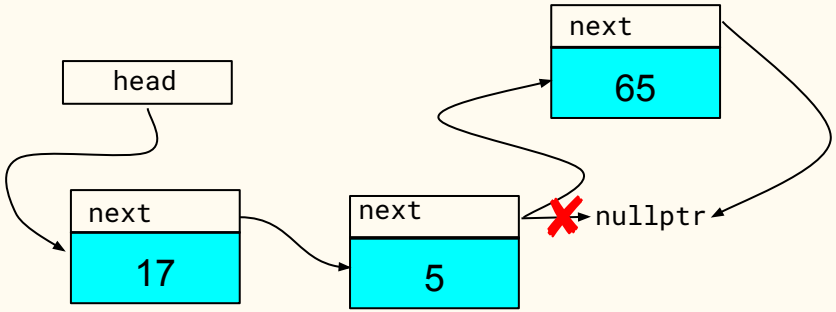
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    // 2. list contains at least one Node
}
```



adding a Node to end of list

Building a list

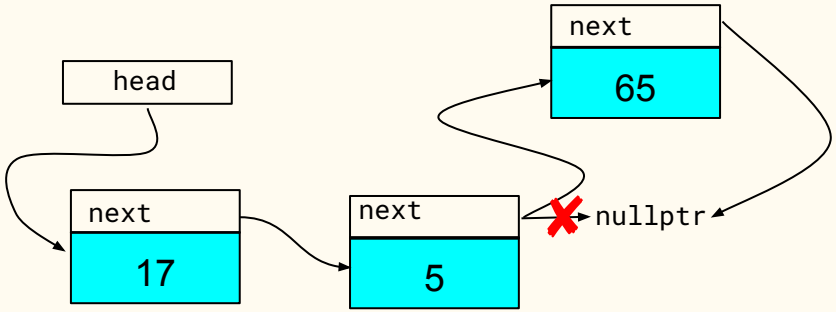
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (___) {

    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list

Building a list

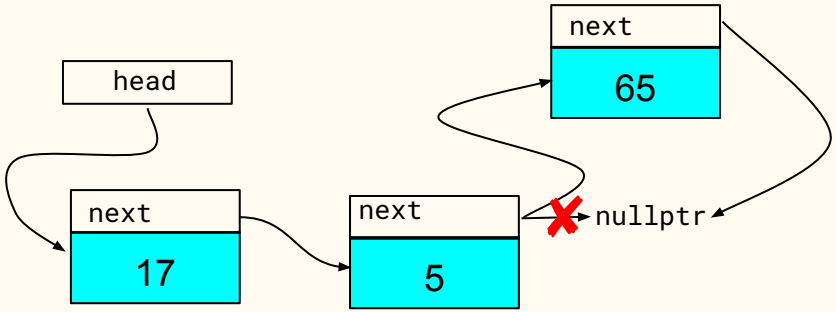
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (_12_) {

    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list

Which expression (replacing blank #12) evaluates to `true` when the list accessed through `head_ptr` is empty (contains no `Node`s)?

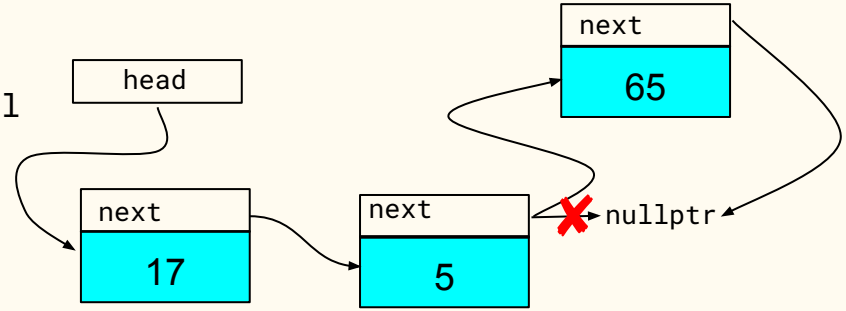
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (_12_) {

    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list

Building a list

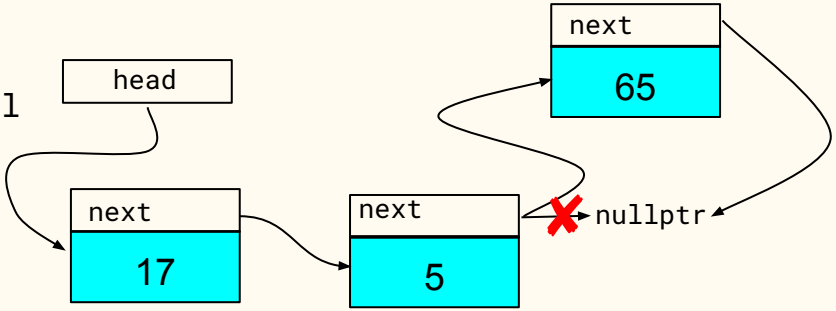
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (head_ptr == nullptr) {
        // create a new Node as both head and tail
        ___
    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list

Building a list

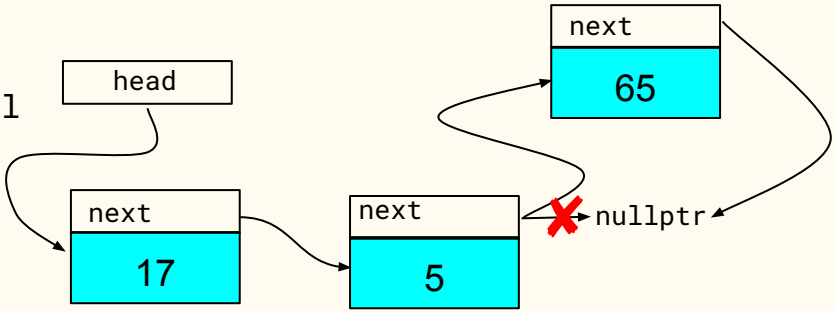
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (head_ptr == nullptr) {
        // create a new Node as both head and tail
        ___ = ___;
    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list

Building a list

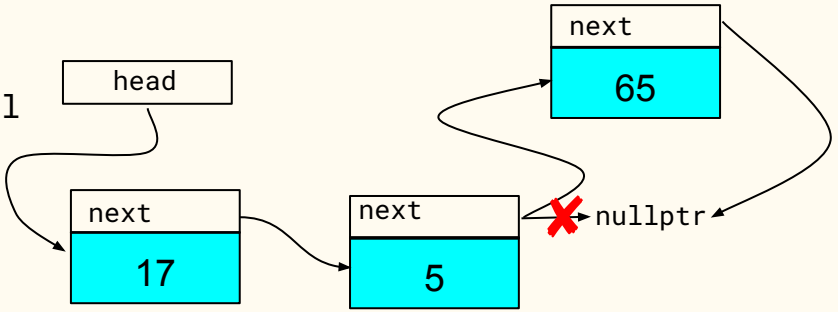
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (head_ptr == nullptr) {
        // create a new Node as both head and tail
        ___ = _13_;
    } else {
        // 2. list contains at least one Node
    }
}
```



adding a Node to end of list

Which expression replaces blank #13 to instantiate a `Node` with `data` as the initial value of the `Node`'s `data` member?

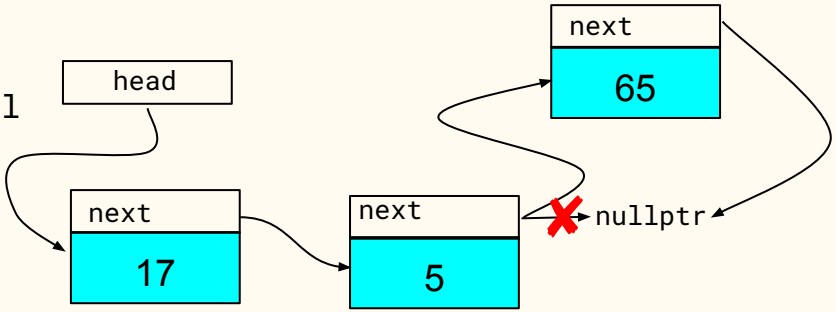
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (head_ptr == nullptr) {
        // create a new Node as both head and tail
        ___ = _13_;
    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list

Building a list

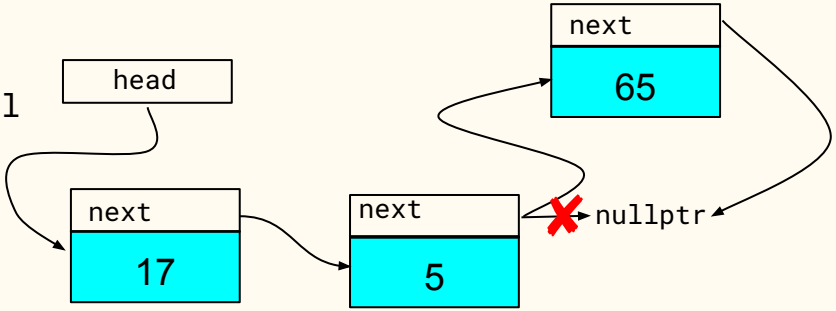
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (head_ptr == nullptr) {
        // create a new Node as both head and tail
        ___ = new Node(data);
    } else {
        // 2. list contains at least one Node
    }
}
```



adding a Node to end of list

Building a list

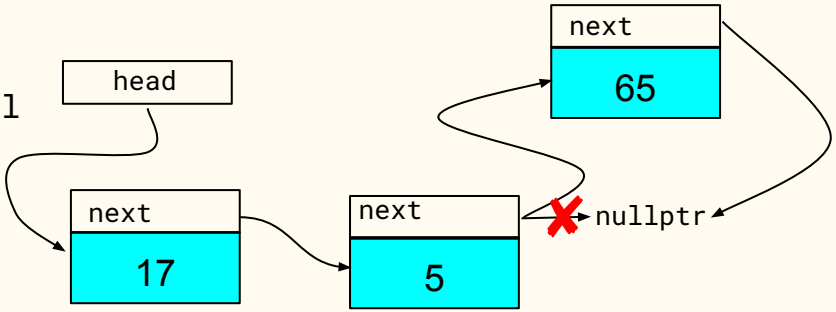
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (head_ptr == nullptr) {
        // create a new Node as both head and tail
        _14_ = new Node(data);
    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list

To which variable (replacing blank #14) do we assign the address of the newly instantiated Node?

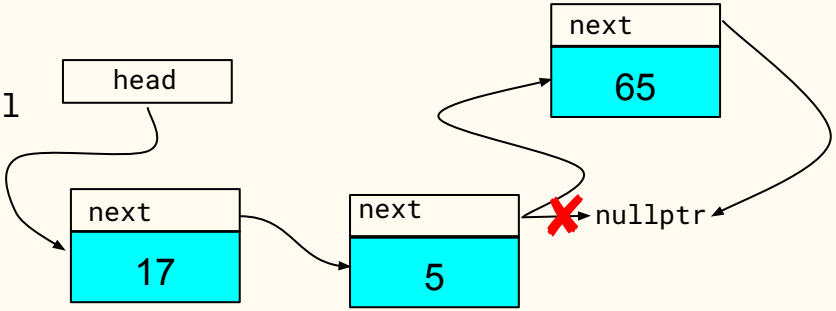
```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (head_ptr == nullptr) {
        // create a new Node as both head and tail
        _14_ = new Node(data);
    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list

Building a list

```
struct Node {
    Node(int data = 0, Node* next = nullptr)
        : data(data), next(next) {}
    int data;
    Node* next;
};

void add_tail_to_list(Node*& head_ptr, int data) {
    // two possibilities
    // 1. list is empty
    if (head_ptr == nullptr) {
        // create a new Node as both head and tail
        head_ptr = new Node(data);
    } else {
        // 2. list contains at least one Node
    }

}
```



adding a Node to end of list