EECS 402 Discussion 10!

Big 3, Linked Lists, Stacks and Queues

Big 3

Overview

- **Rule**: If any class implements a destructor, that class should also implement a copy constructor and overloaded assignment operator.
  - Destructor: Responsible for deleting dynamically allocated data
  - Copy constructor: Called when a new object is made based on another object
  - Assignment operator: Called when one object is set to the value of another

Deep Copy vs. Shallow Copy

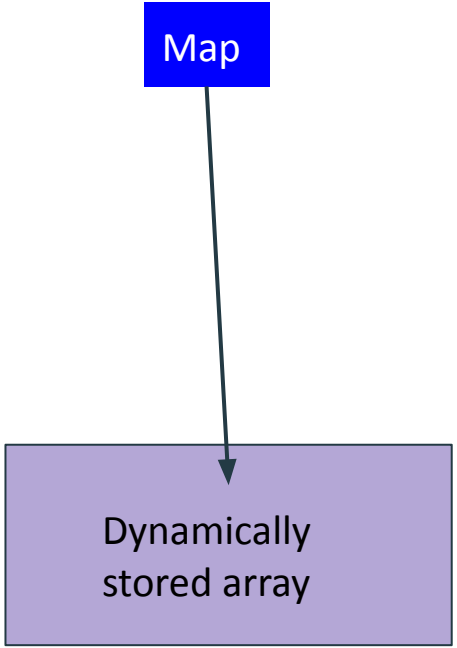
Shallow copy does not copy dynamically allocated memory pointed to by attributes

Sometimes shallow copies are okay

Sometimes we need to make deep copies

Why do we need deep copy?

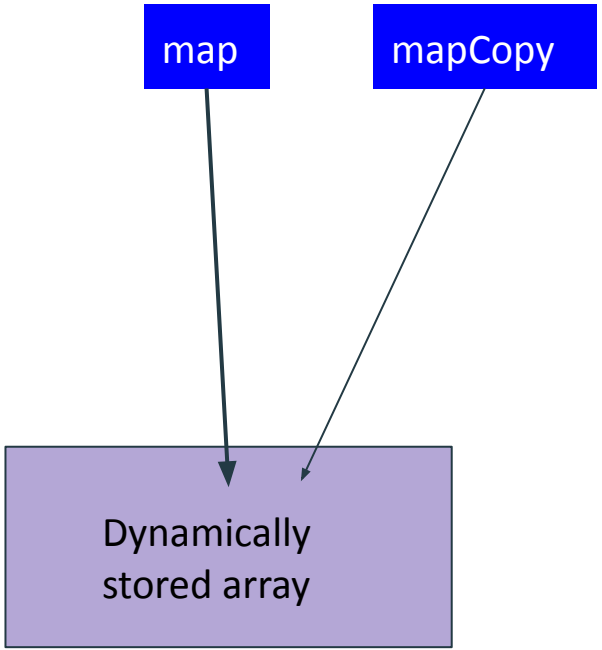
```
class MapClass {
    private:
        int **map;

};
```



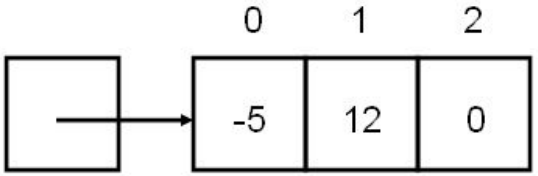
Why do we need deep copy?

MapClass mapCopy(map);

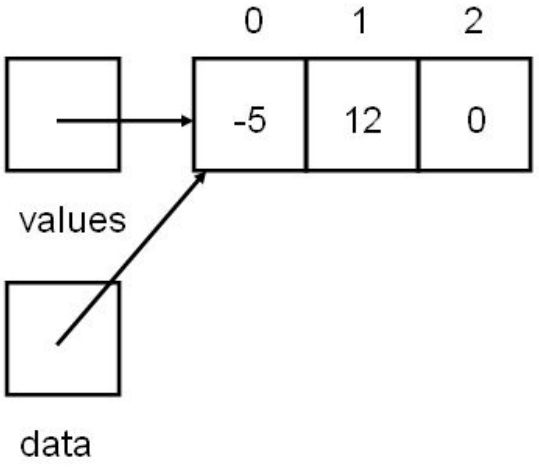
```
class MapClass {
    private:
        int **map;

};
```



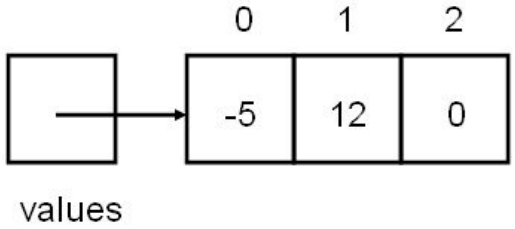
Shallow copy



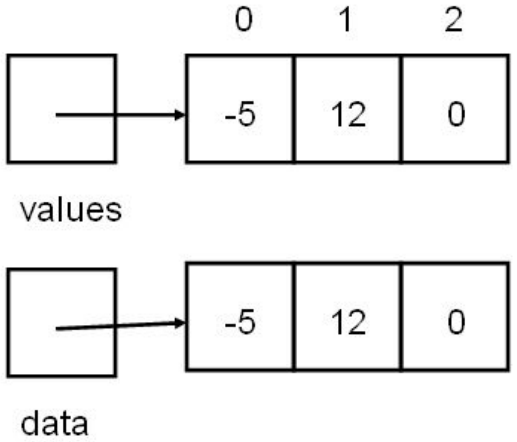

```
data = values;
```



Deep copy

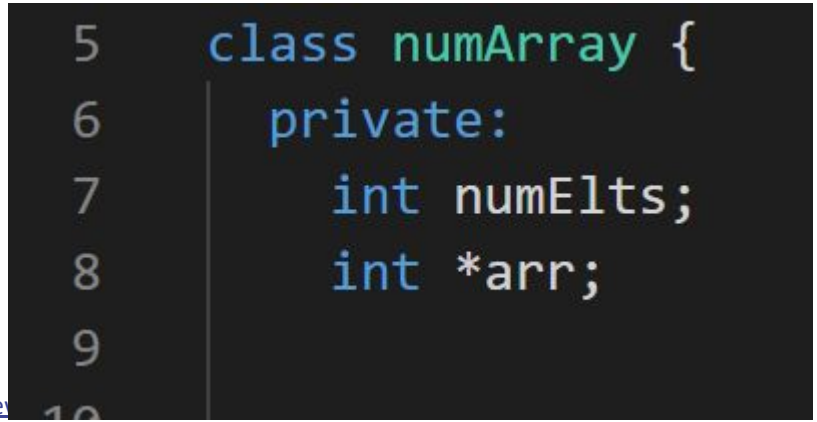



```
//code for deep copy
```



Example Copy Constructor

Create a copy constructor for this class



https://drive.google.com/file/d/1tKzXoD1Rsx9MV5iS2x_P30yFq4NOeBnU/vie

https://drive.google.com/file/d/1NJXYX65NbEBuT1gu2UzKl2bFZQJZvBdN/view?usp=sharing

git clone https://github.com/emolson16/big3.git

Example Solution

```
numArray(numArray & other){
  numElts = other.numElts;

  arr = new int[numElts];

  for(int i = 0; i < numElts; ++i){
    arr[i] = other.arr[i];
  }

}
```

Overload =

- Check for self assignment
- Copy non-dynamic attributes
- Clean old dynamic data
- Copy dynamically allocated data

Example Assignment Overload

Implement the overloaded operator= function

```
void operator=(numArray& rhs) {

}
```

Example Assignment Operator Solution

```
29      void operator=(numArray& rhs) {
30        if(this == &rhs) { // check self assignment
31          return;
32        }
33
34        numElts = rhs.numElts; // copy non-dynamic attributes
35
36        delete[] arr; // clean old dynamic data
37
38        arr = new int[numElts]; // copy dynamically allocated data
39        for(int i = 0; i < numElts; ++i) {
40          arr[i] = rhs.arr[i];
41        }
42
43        return;
44      }
```

Example Dtor

```
~numArray(){
}
```

Write the destructor for the class

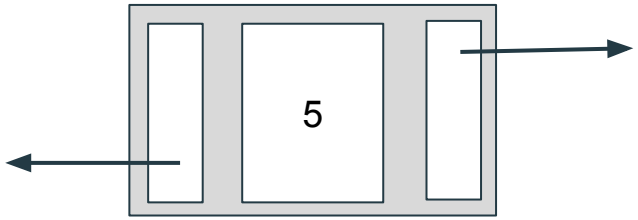
Example dtor Solution

```
~numArray(){
    delete[] arr;
}
```

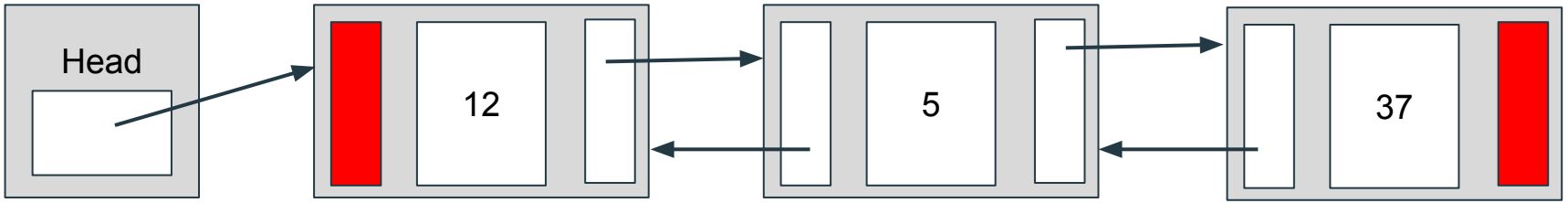
Lists

Node

- Stores a value (can be anything really)
- Has a next pointer
- Sometimes has a previous pointer
- Typically stored as a struct or class



Linked Lists (or just list)



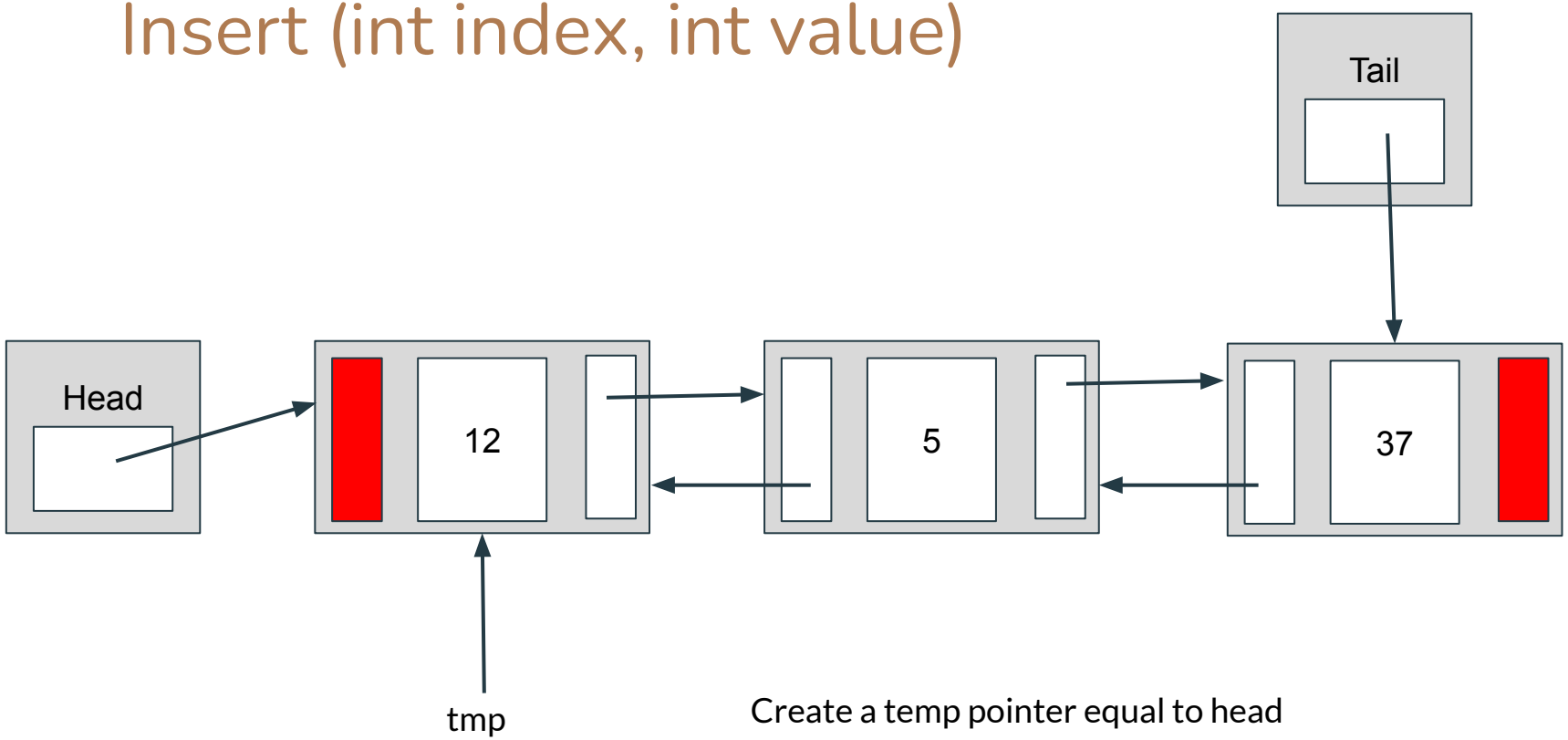
= NULL

Each pointer points to the **node** as a whole

Functions we might want to include

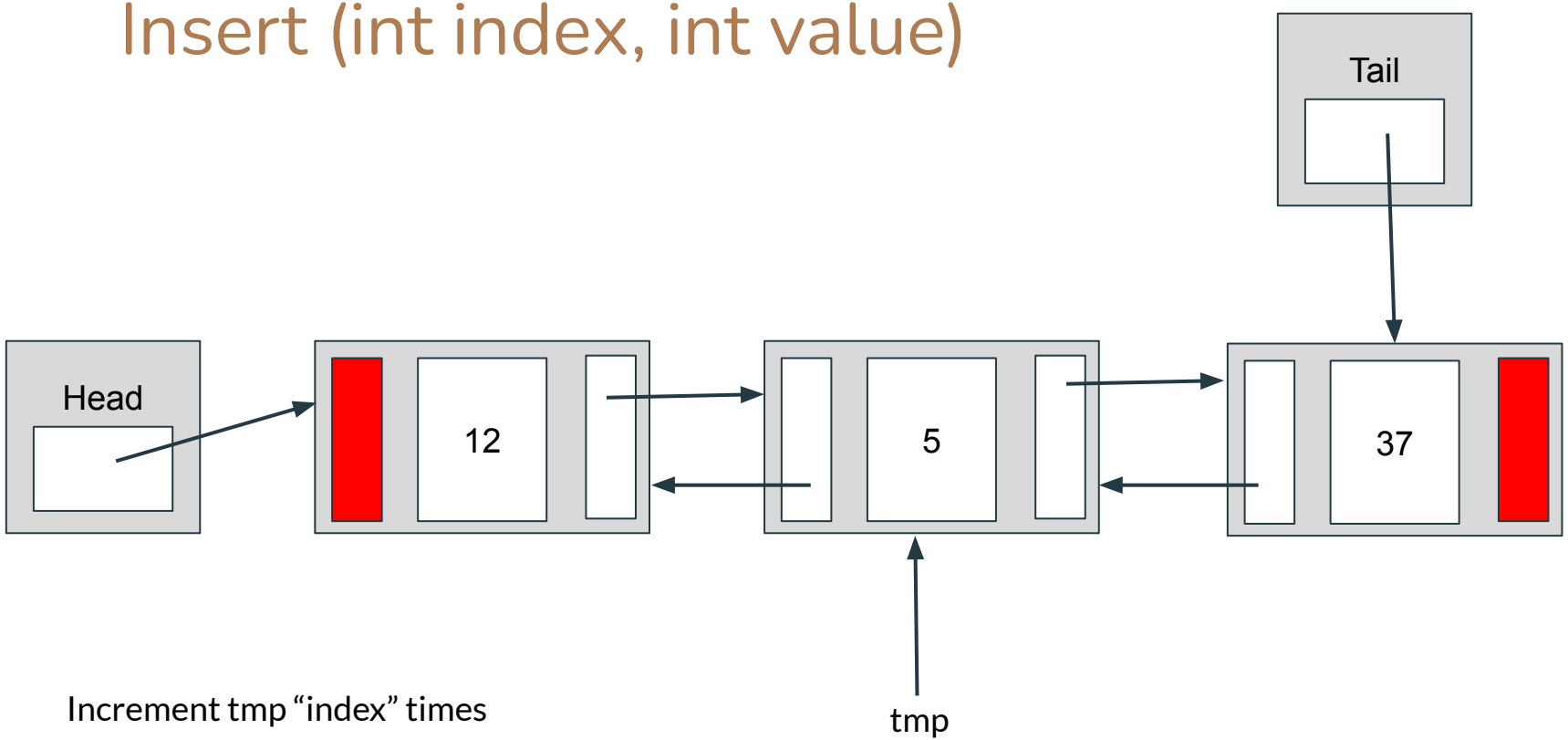
- appendFront(int val)
- appendBack(int val)
- insert(int index, int val)
- popFront()
- popBack()
- remove(int index)
- valueAt(int index)
- print()
- clear()
- size()

Insert (int index, int value)



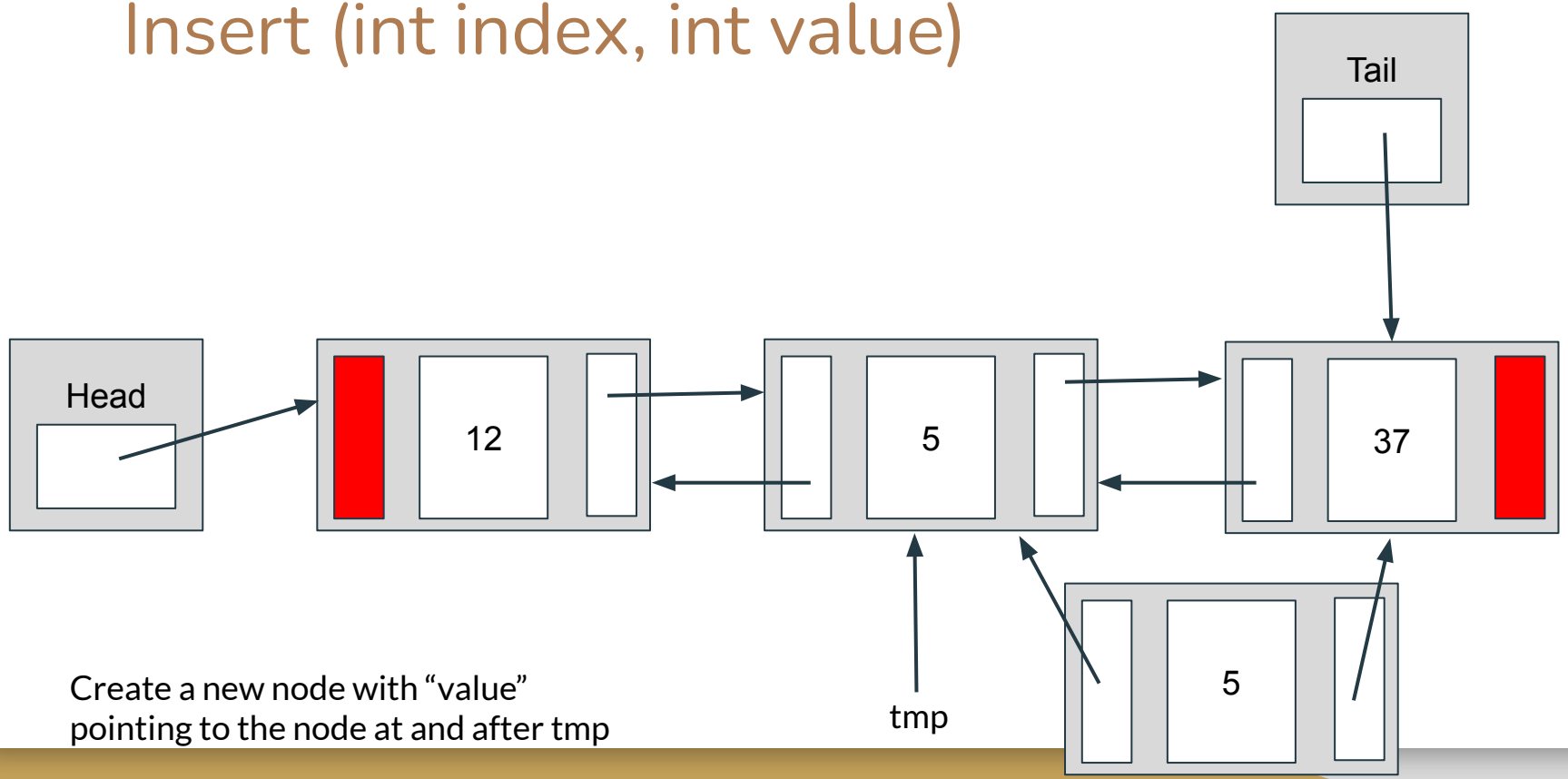
Create a temp pointer equal to head

Insert (int index, int value)



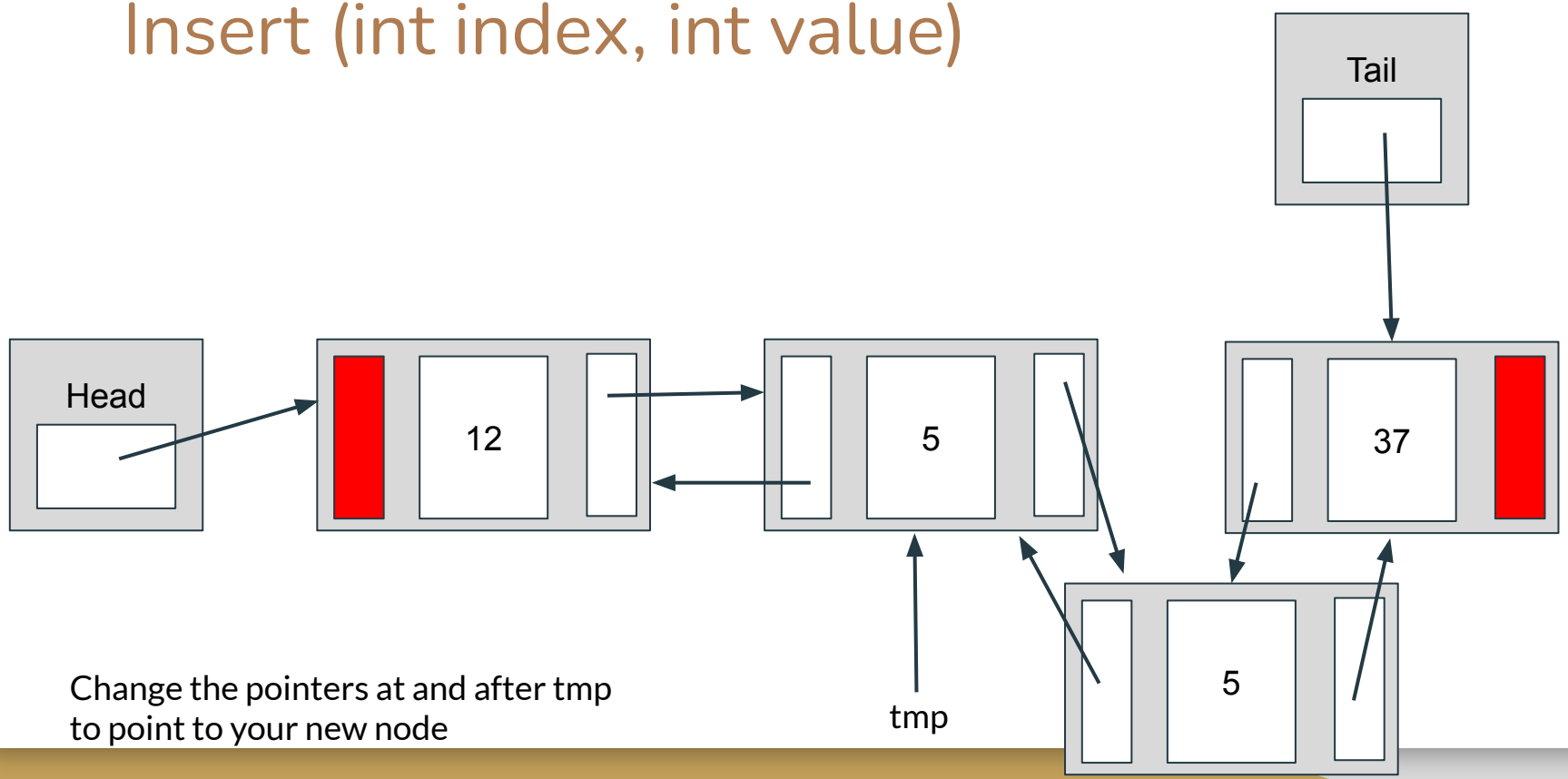
Increment tmp “index” times

Insert (int index, int value)



Create a new node with “value” pointing to the node at and after tmp

Insert (int index, int value)



Change the pointers at and after tmp
to point to your new node

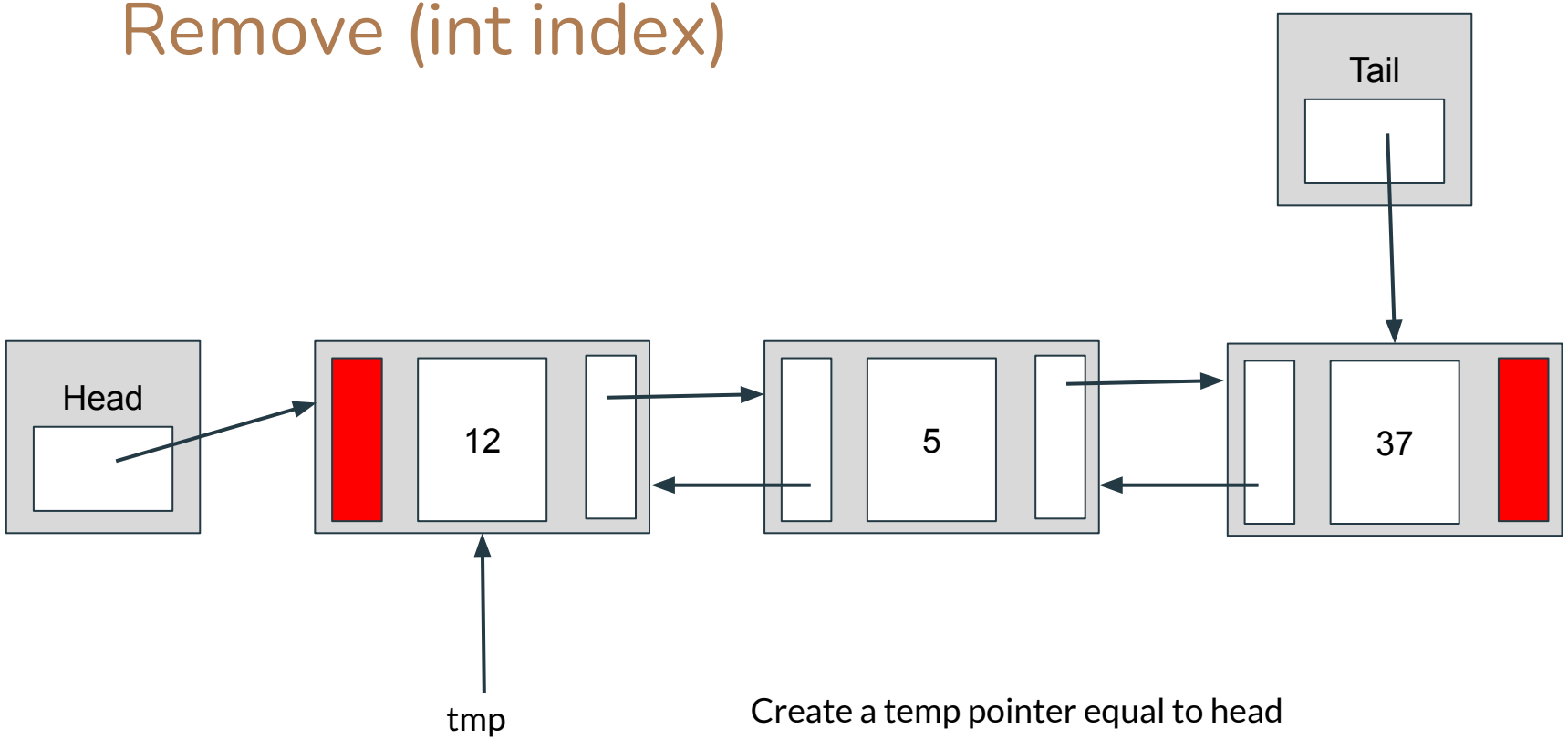
Edge cases

- Edge case: a specific input that is more difficult to handle than others
- Here, index = 0 is an edge case
    - Why?

Edge cases

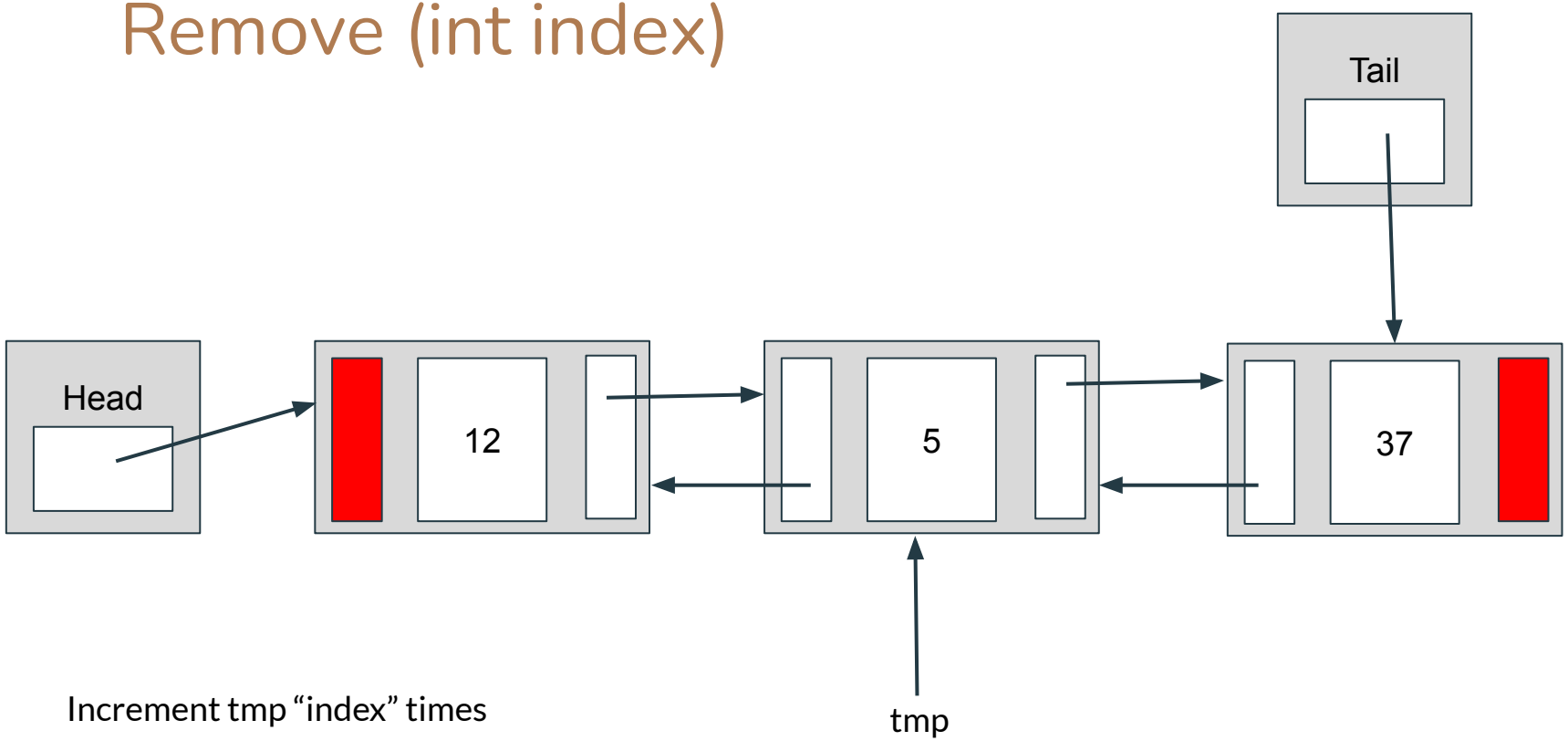
- Edge case: a specific input that is more difficult to handle than others
- Here, index = 0 is an edge case
  - Why?
  - Your algorithm now needs to change the value of the “head” pointer
  - You will fix this algorithm on P4

Remove (int index)



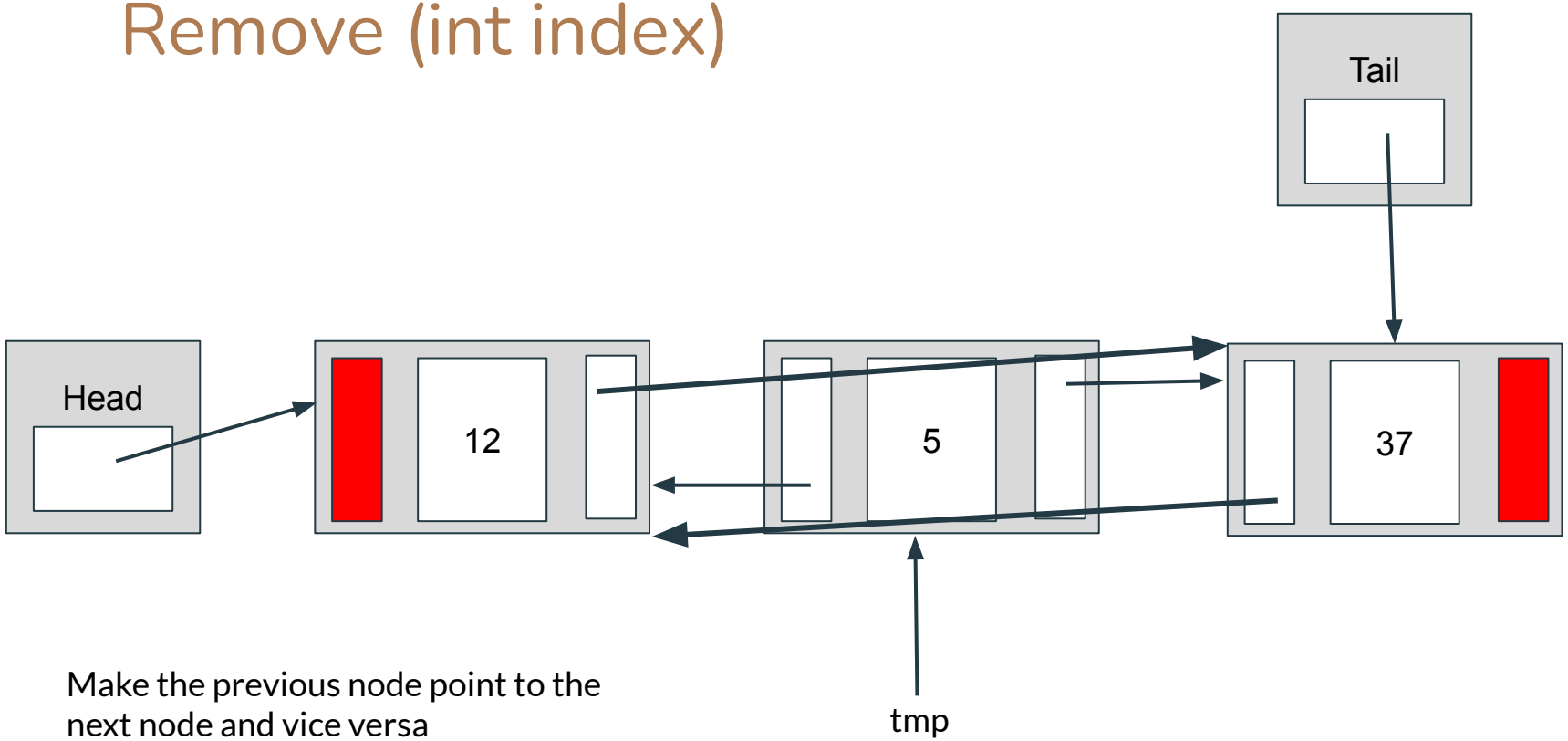
Create a temp pointer equal to head

Remove (int index)



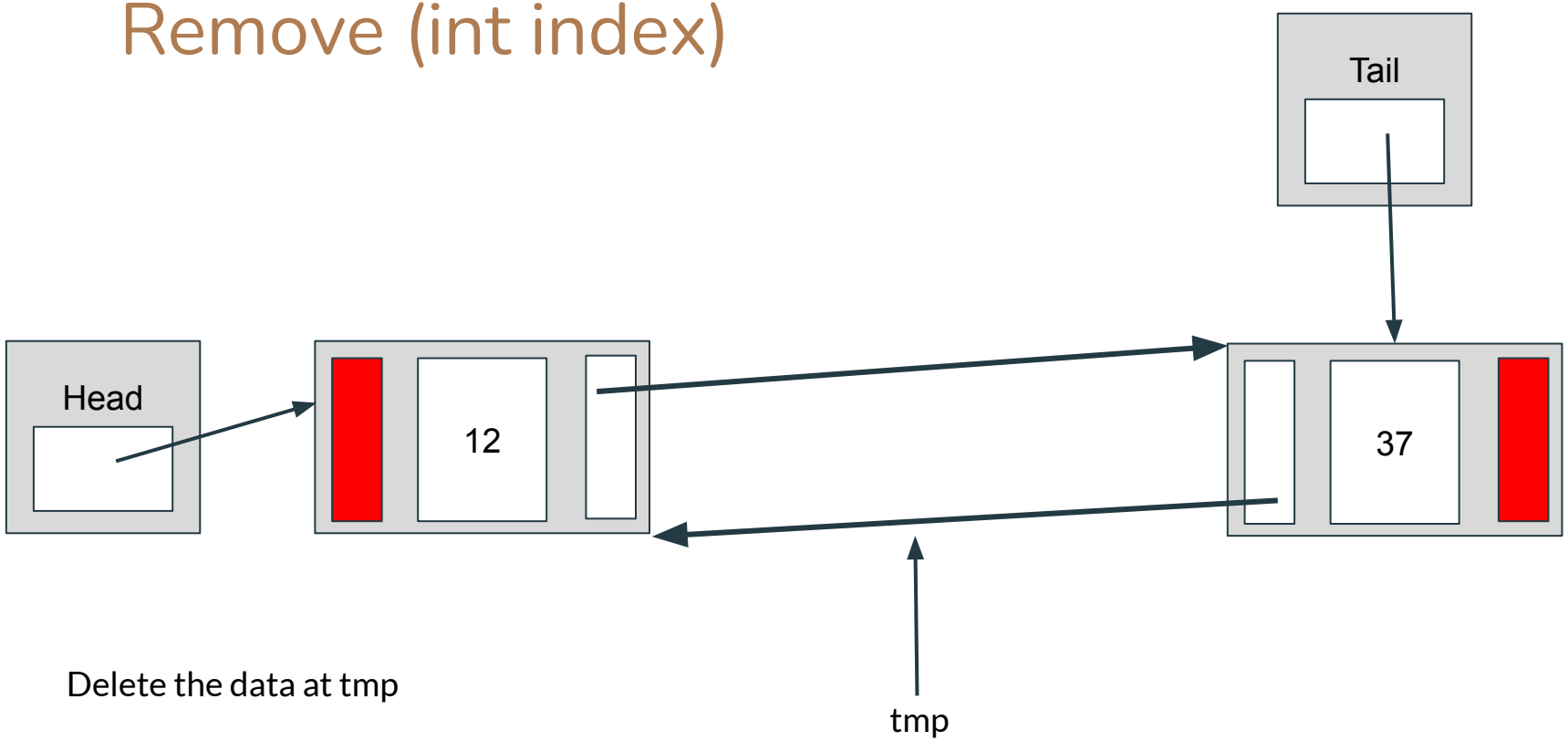
Increment tmp “index” times

Remove (int index)



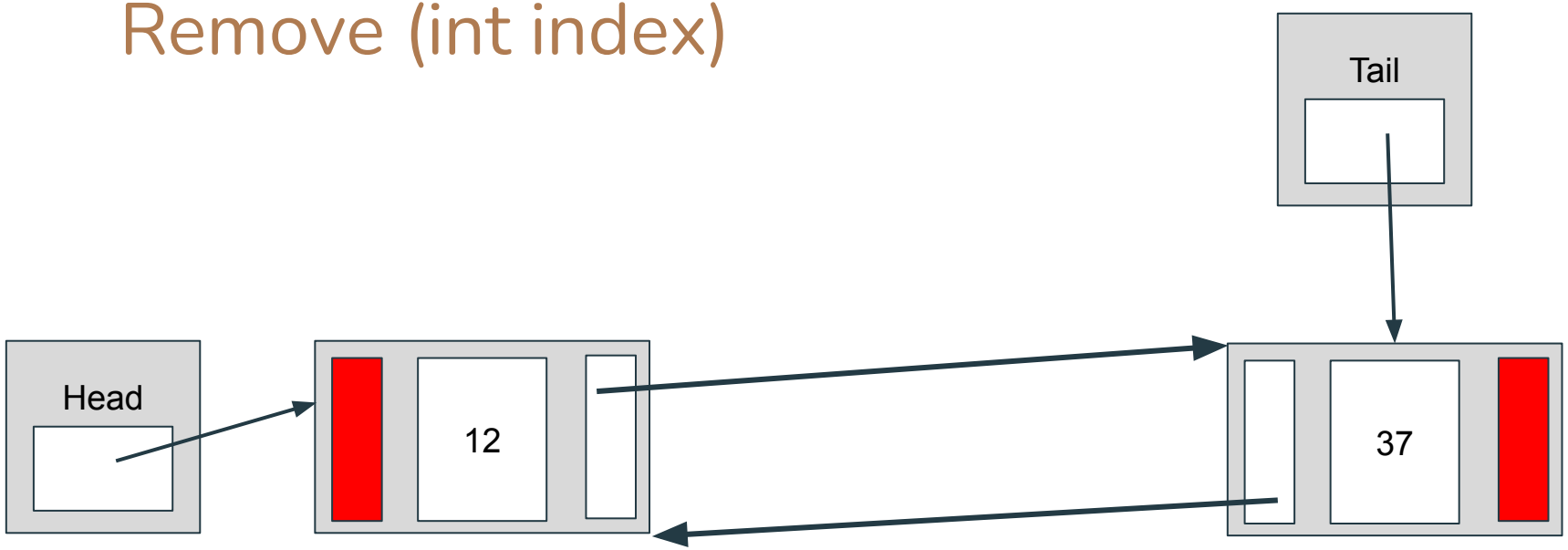
Make the previous node point to the next node and vice versa

Remove (int index)



Delete the data at tmp

Remove (int index)



Set tmp to NULL to avoid
“dangling pointers”

tmp 0

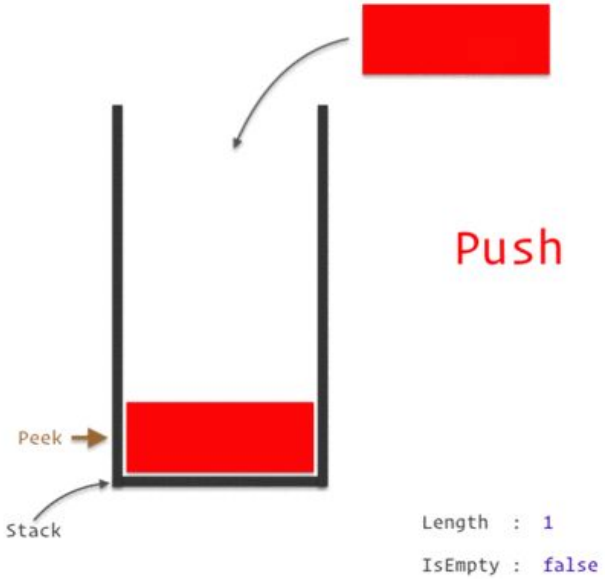
Stacks, Queues

Stacks

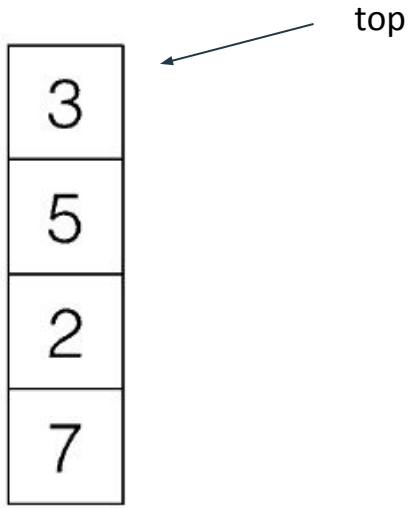
Items are placed “on top” of the stack when added

Items are popped off the top of the stack when removed

Last in First Out (LIFO)

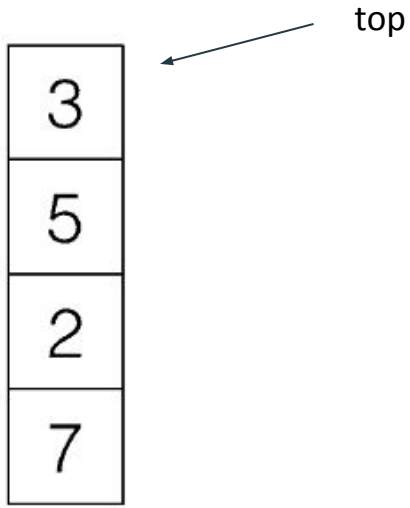


In what order were these elements pushed?



In what order were these elements pushed?

Answer: 7, 2, 5, 3

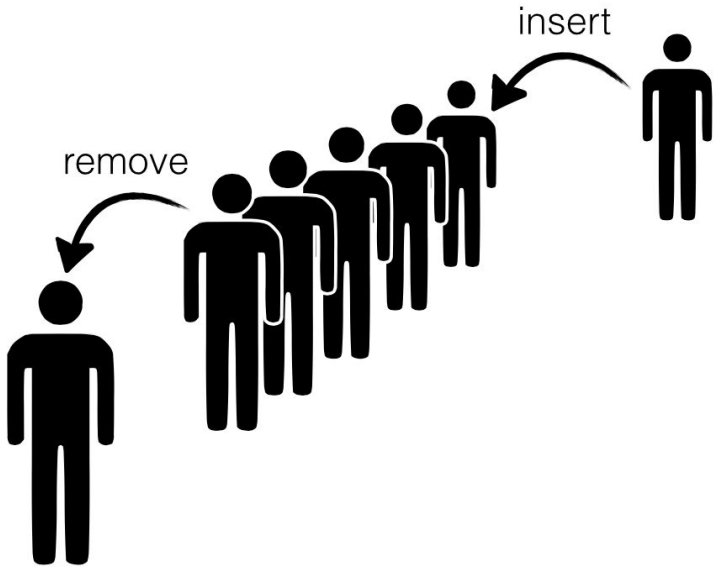


Queue

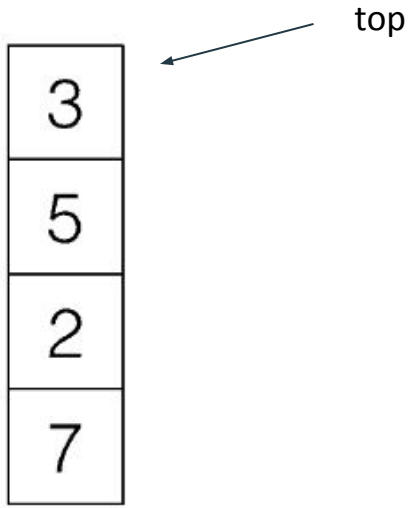
Items are placed “at the back” of the queue when inserted

Items are popped off the front of the queue when deleted

First In First Out (FIFO)

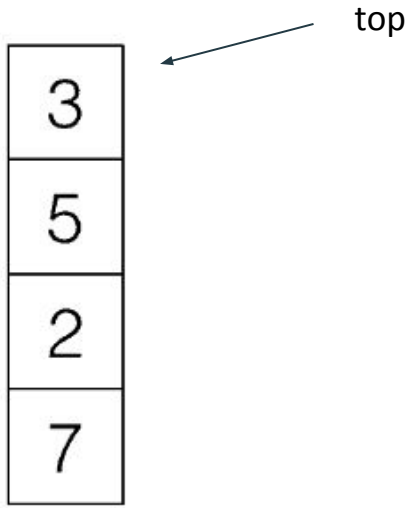


In what order were these elements pushed?



In what order were these elements pushed?

Answer: 3, 5, 2, 7

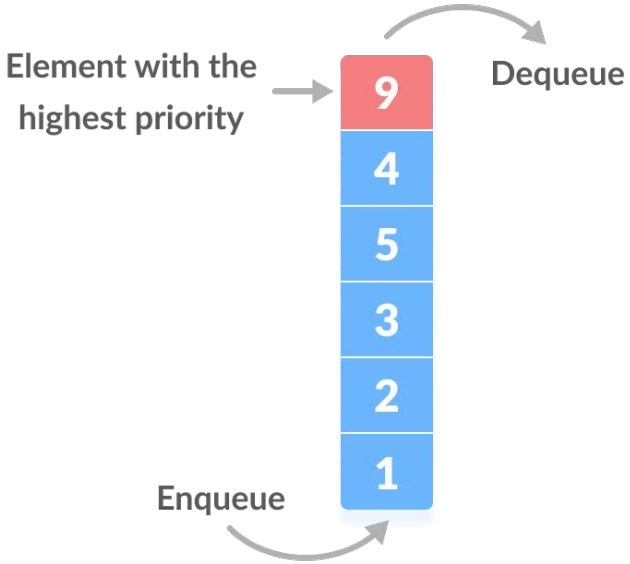


Priority Queue

Items are placed according to some sorted order

Items with “highest priority” are always at the head of the queue

The sorted order is left up to the user’s discretion



Example

Items are sorted by default in *ascending* order

Note that this example uses C++'s priority queue implementation

- You are **not allowed** to use this for project 4!!!

```
int main() {
    priority_queue<int> pq;
    printPQueue(pq);

    pq.push(3);
    printPQueue(pq);
    pq.push(5);
    printPQueue(pq);
    pq.push(2);
    printPQueue(pq);
    pq.push(7);
    printPQueue(pq);

    while (!pq.empty()) {
        pq.pop();
        printPQueue(pq);
    }
}
```

```
[]
[3]
[5, 3]
[5, 3, 2]
[7, 5, 3, 2]
[5, 3, 2]
[3, 2]
[2]
[]
```