COSC 2436: Queues

Queues: What is a queue?

A queue is a data structure that has a first-in-first-out (FIFO) characteristic. This means that elements are inserted to the rear of the queue and removed from the front of a queue.

A queue can be implemented using:

- **an array**
- **a linked list**
- **the C++ STL queue**

There are three main types of queues you will come across:

- **Regular Queue**
- **Priority Queue**
- **Circular Queue**

Queues: Different Types of Queues

Regular Queue:

- **A regular queue is a queue that has no special property to it. Elements are added to the rear and removed from the front.**
- **Some examples of a regular queue include: a checkout line at a grocery store, a line in a coffee shop, an input stream.**

Priority Queue:

- **A priority queue is a queue where elements with a higher priority are closer to the front of the queue than elements with a lower priority.**
- **Some examples of a priority queue include: an emergency room line, a help service, boarding onto an airplane, customer support (a website being down is more urgent than the wrong color font on a page).**

Circular Queue:

- **A circular queue is a queue where the rear points back to the front of the queue.**
- **An example of a circular queue is round robin scheduling.**

Queues: C++ STL queue

C++ has a built in queue. You can include it in your code by writing:
`#include<queue>`
in your code's header files

Some of the C++ STL queue functions include:

- `queue<T> queueName;` **//constructor**
- `void push(T);` **//inserts value into the rear of queue**
- `void pop();` **//removes the front of the queue (does not return a value)**
- `bool empty();` **//returns true if queue is empty, false otherwise**
- `T front();` **//returns the value at the front of the queue**
- `int size();` **//returns the number of elements in the queue**

Queues Practice: Regular Queue

Write the regularQueue class functions `void enqueue(int d)` and `void dequeue()`. Your functions should have a time complexity of O(1).

```
struct node{
  int data;
  node *next;
  node(int d) : data(d), next(nullptr) {}
};

class regularQueue{
  private:
    node *front;
    node *rear;
  public:
    regularQueue(){front = rear = nullptr;}
    void enqueue(int d);
    void dequeue();
};
```

Queues Practice: Regular Queue

```
void regularQueue::enqueue(int d){
  node *temp = new node(d);
  if(front = nullptr){
    front = rear = temp;
  }
  else{
    rear->next = temp;
    rear = temp;
  }
}

void regularQueue::dequeue(){
  if(front != nullptr){
   node *temp = front;
   front = front->next;
   delete temp;
  }
}
```

Queues Practice: Priority Queue

Write the priorityQueue class function void enqueue(string d, int p) which enqueues data with a higher priority closer to the front of the queue.

```
struct node{
  string data;
  int priority;
  node *next;
  node(string d, int p) : data(d), priority(p), next(nullptr) {}
};

class priorityQueue{
  private:
    node *front;
  public:
    priorityQueue(){front = nullptr;}
    void enqueue(string d, int p);
};
```

Queues Practice: Priority Queue

```
void priorityQueue::enqueue(string d, int p){
  node *temp = new node(d, p);
  if(front = nullptr)
    front = temp;
  else if(p > front->priority){
    temp->next = front;
    front = temp;
  }
  else{
    node *cu = front;
    while(cu->next != nullptr && p <= cu->priority){
      cu = cu->next;
    }
    cu->next = temp;
  }
}
```

Queues Practice: Priority Queue

Write the priorityQueue class function void enqueue(string d, int p) which enqueues data with a higher priority closer to the front of the queue. Your function should use recursion.

```
struct node{
  string data;
  int priority;
  node *next;
  node(string d, int p) : data(d), priority(p), next(nullptr) {}
};

class priorityQueue{
  private:
    node *front;
  public:
    priorityQueue(){front = nullptr;}
    void enqueue(node *n, string d, int p);
};
```

Queues Practice: Priority Queue

```
void priorityQueue::enqueue(node *n, string d, int p){
  if(front = nullptr)
    front = new node(d, p);
  else if(p > front->priority){
    node *temp = new node(d, p);
    temp->next = front;
    front = temp;
  }
  else if(p < n->priority && n->next = nullptr)
    n->next = new node(d, p);
  else if(p > n->priority && p > n->next->priority){
    node *temp = new node(d, p);
    temp->next = n->next;
    n->next = temp;
  }
  else
    enqueue(n->next, d, p);
}
```

Queues Practice: Print Level Order

Write the function `void levelOrder(int heap[], int size)` which prints a heap in level order. Example:

```
levelOrder([10, 9, 6, 5, 1, 2, 3], 7)
```

will produce:

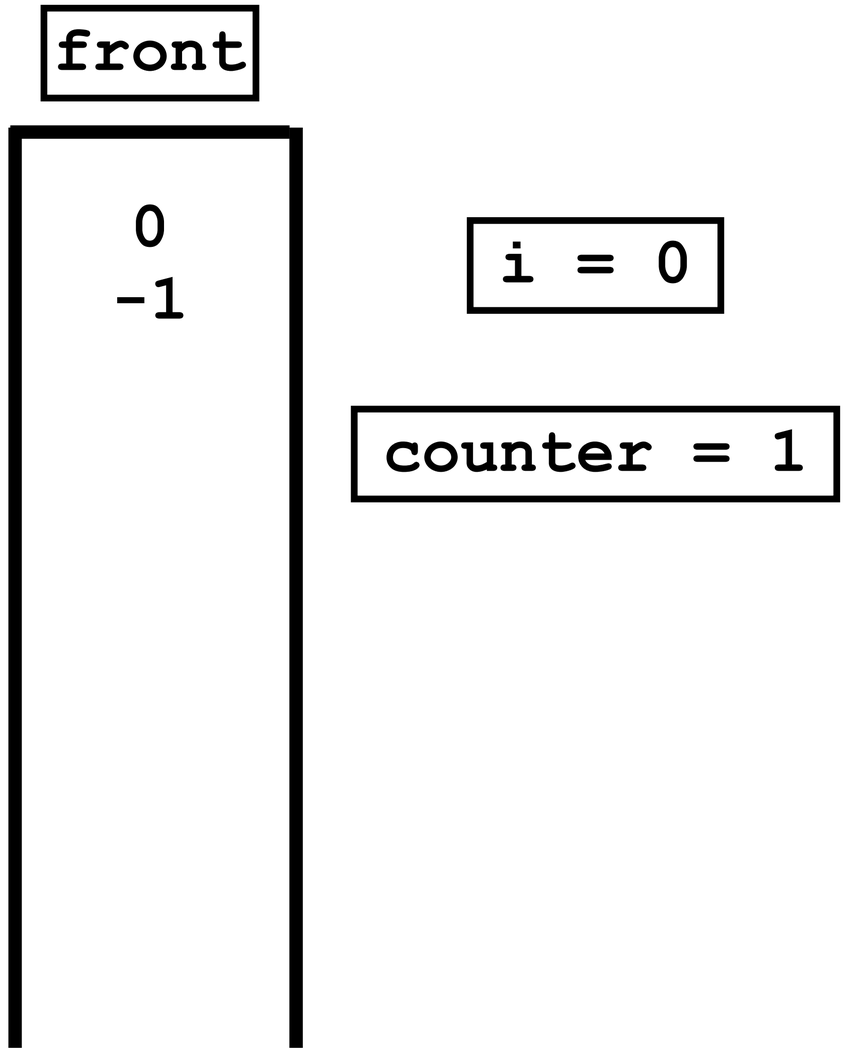
```
10
9 6
5 1 2 3
```

```
void levelOrder(int heap[], int size){


}
```

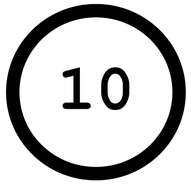
Queues Practice: Print Level Order

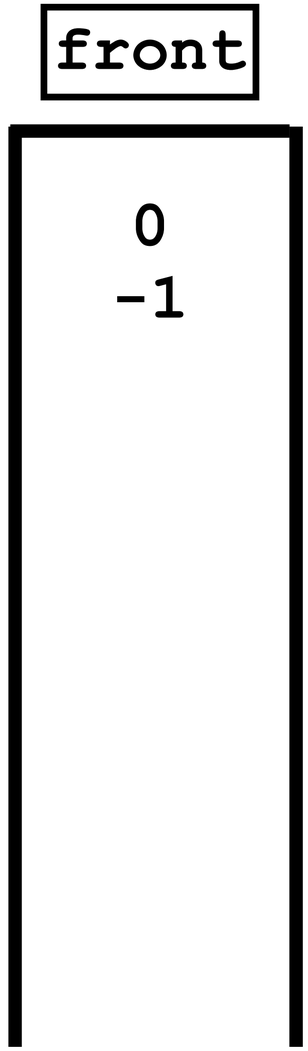
```
void levelOrder(int heap[], int size){
  queue<int> q;
  int counter = 1;
  q.push(0);
  q.push(-1);
  while(!q.empty()){
    int i = q.front(); q.pop();
    if(i != -1){
      cout << heap[i] << " ";
      if((2*i+1) < size){
        q.push(2*i+1);
        counter++;
      }
      if((2*i+2) < size){
        q.push(2*i+2);
        counter++;
      }
    }
    else{
      cout << endl;
      if(counter < size)
        q.push(-1);
    }
  }
}
```

Heap = {10, 9, 6, 5, 1, 2, 3}



Heap = {10, 9, 6, 5, 1, 2, 3}



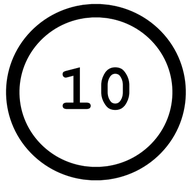


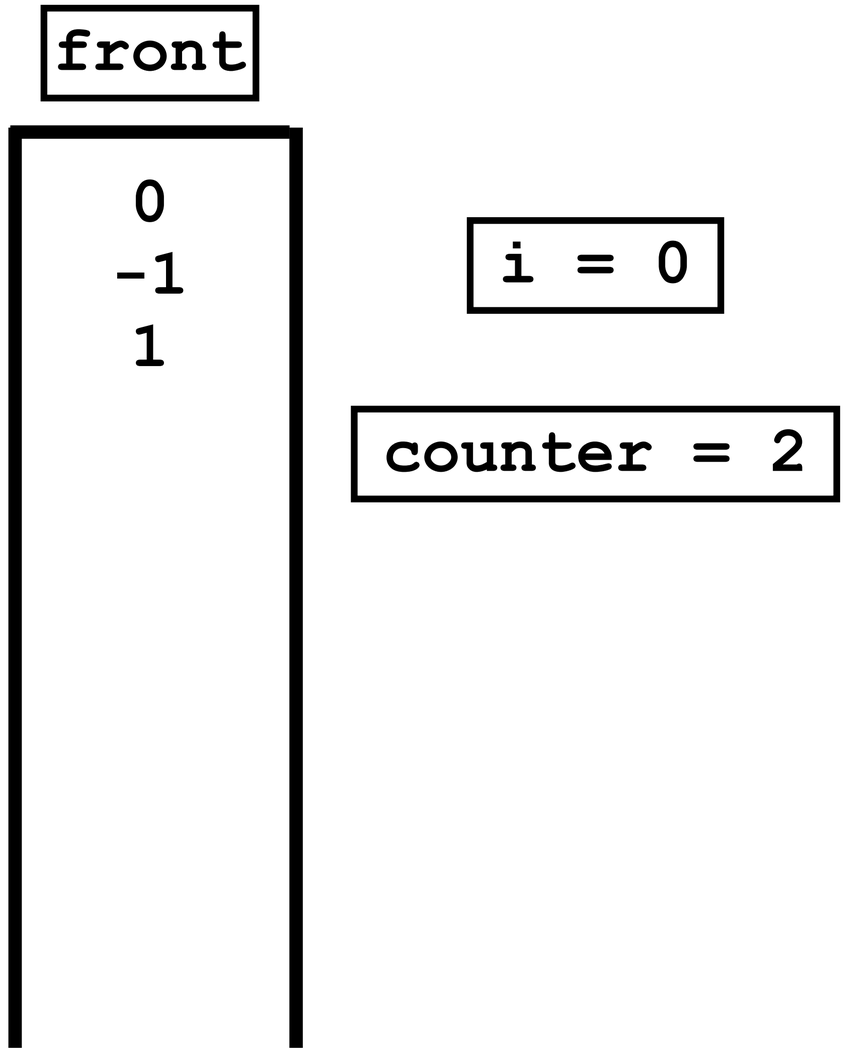
i = 0

counter = 1

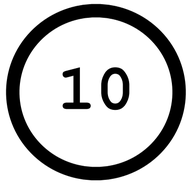
cout heap[0]

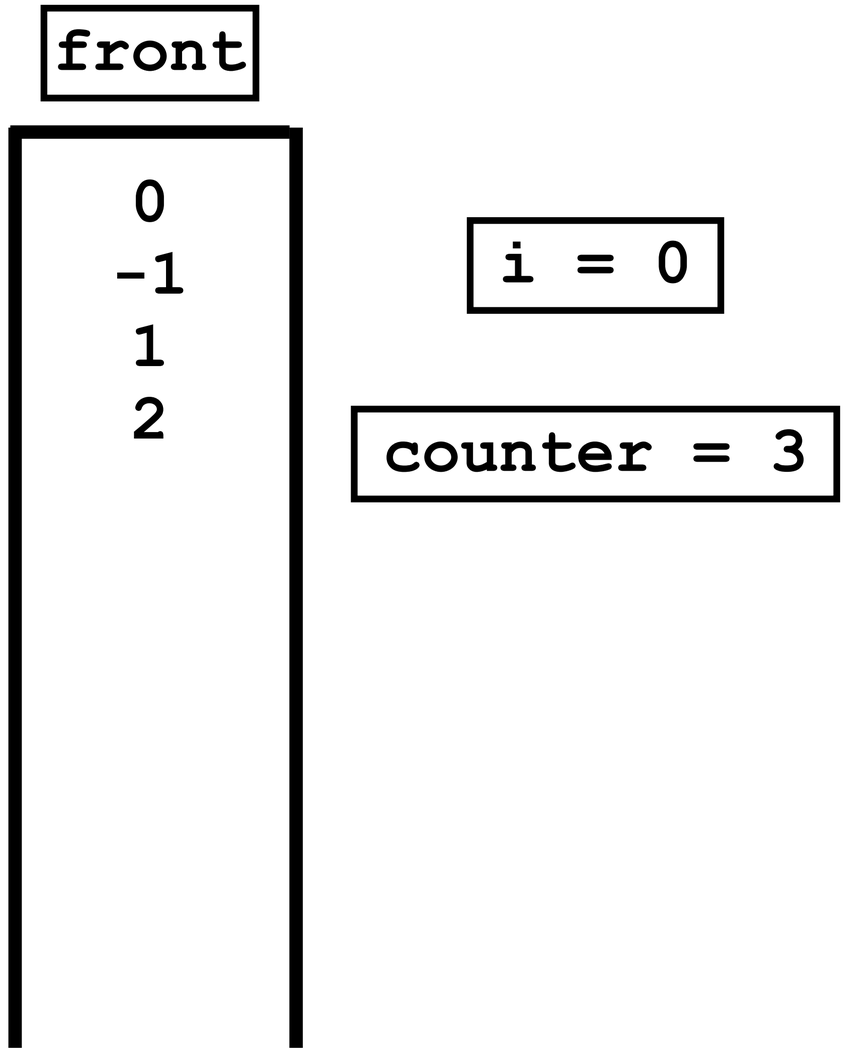
Heap = {10, 9, 6, 5, 1, 2, 3}



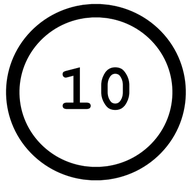


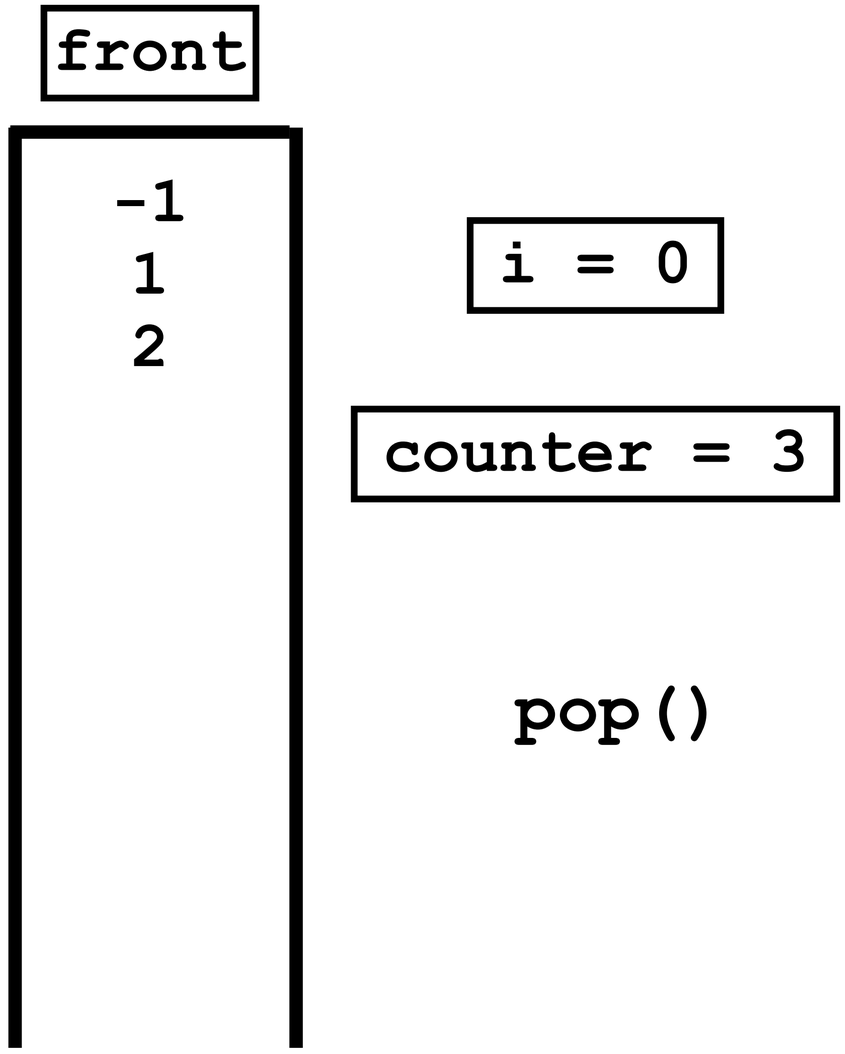
Heap = {10, 9, 6, 5, 1, 2, 3}



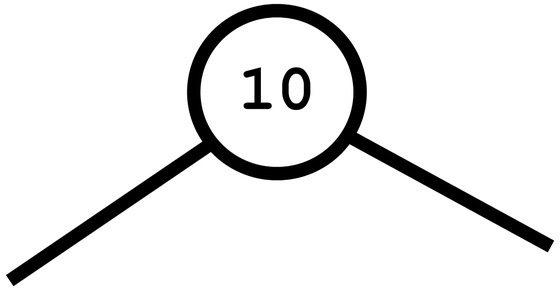


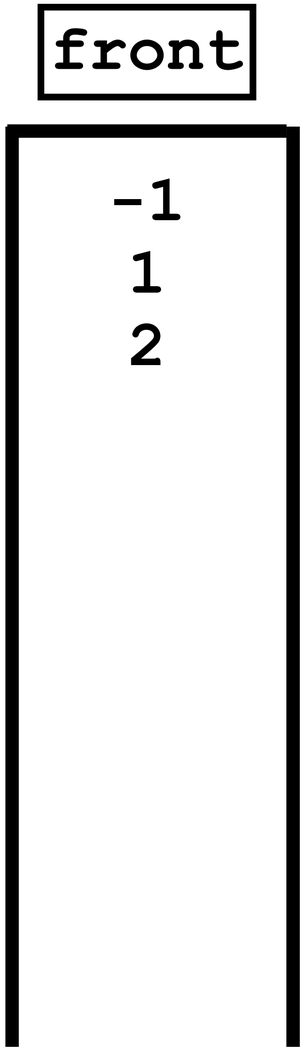
Heap = {10, 9, 6, 5, 1, 2, 3}





Heap = {10, 9, 6, 5, 1, 2, 3}



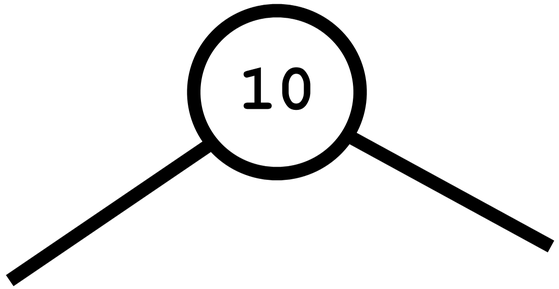


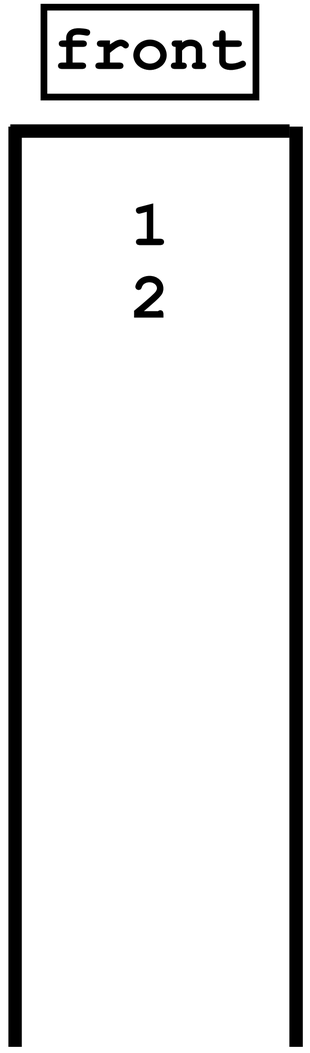
i = -1

counter = 3

cout newline

Heap = {10, 9, 6, 5, 1, 2, 3}



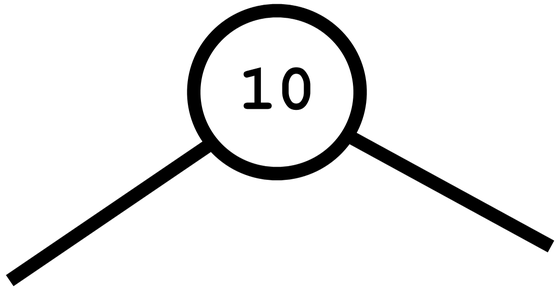


i = -1

counter = 3

pop()

Heap = {10, 9, 6, 5, 1, 2, 3}



front

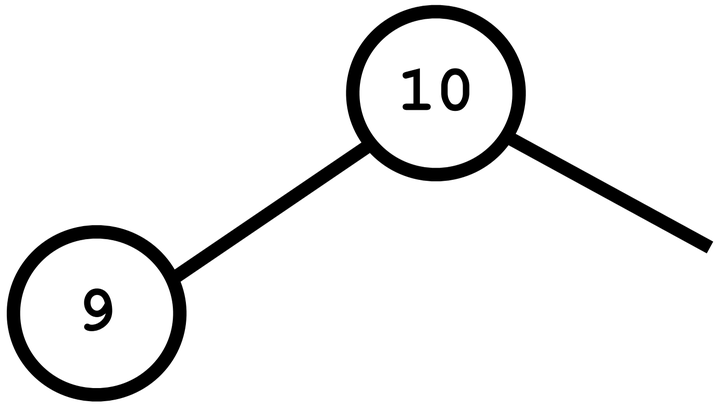
1
2
-1

i = -1

counter = 3

push(-1)

Heap = {10, 9, 6, 5, 1, 2, 3}



front

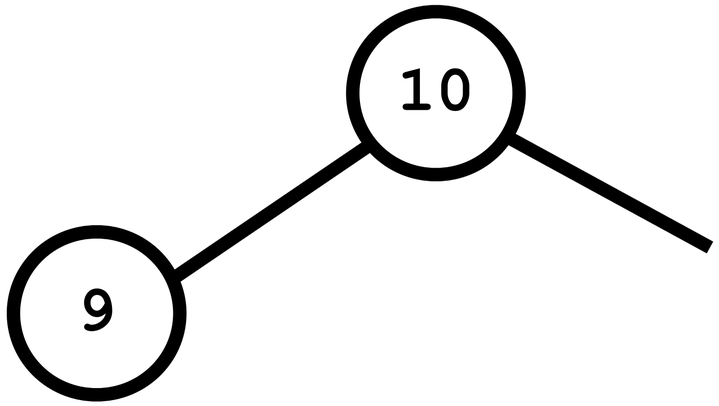
1
2
-1

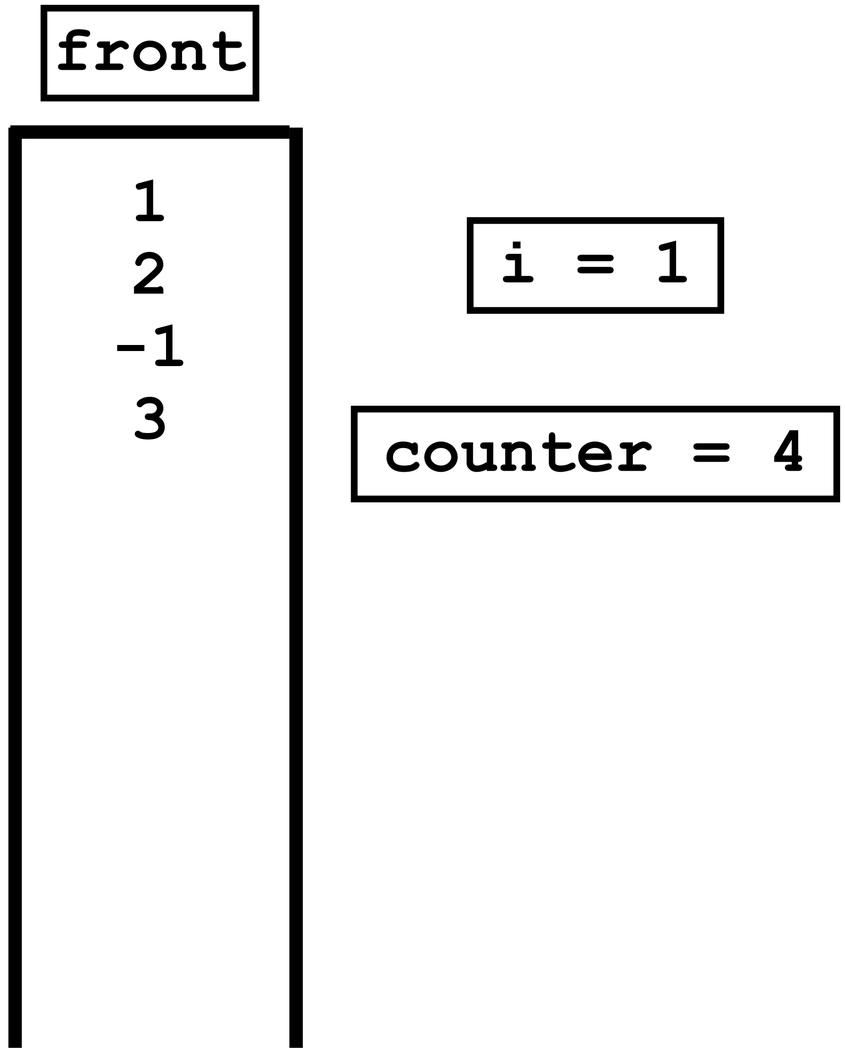
i = 1

counter = 3

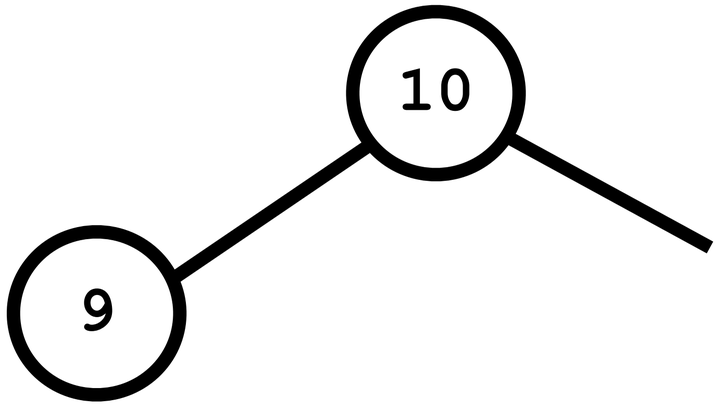
cout heap[1]

Heap = {10, 9, 6, 5, 1, 2, 3}





Heap = {10, 9, 6, 5, 1, 2, 3}



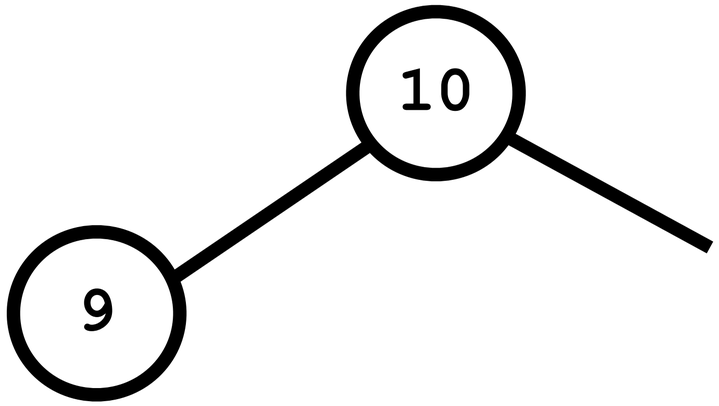
front

1
2
-1
3
4

i = 1

counter = 5

Heap = {10, 9, 6, 5, 1, 2, 3}



front

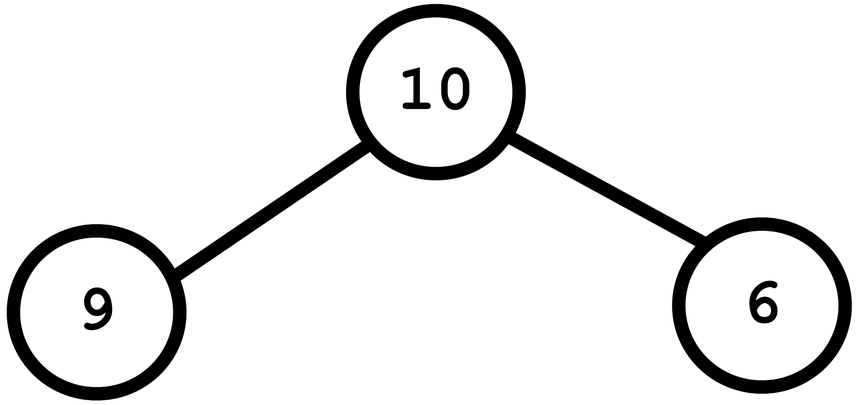
2
-1
3
4

i = 1

counter = 5

pop()

Heap = {10, 9, 6, 5, 1, 2, 3}



front

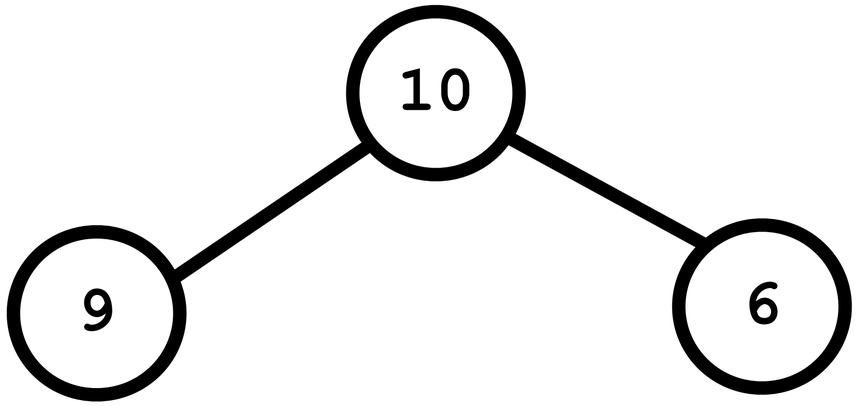
2
-1
3
4

i = 2

counter = 5

cout heap[2]

Heap = {10, 9, 6, 5, 1, 2, 3}



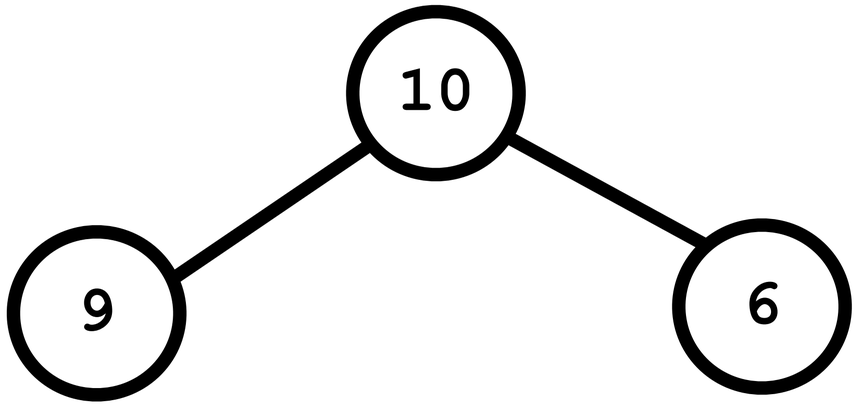
front

2
-1
3
4
5

i = 2

counter = 6

Heap = {10, 9, 6, 5, 1, 2, 3}



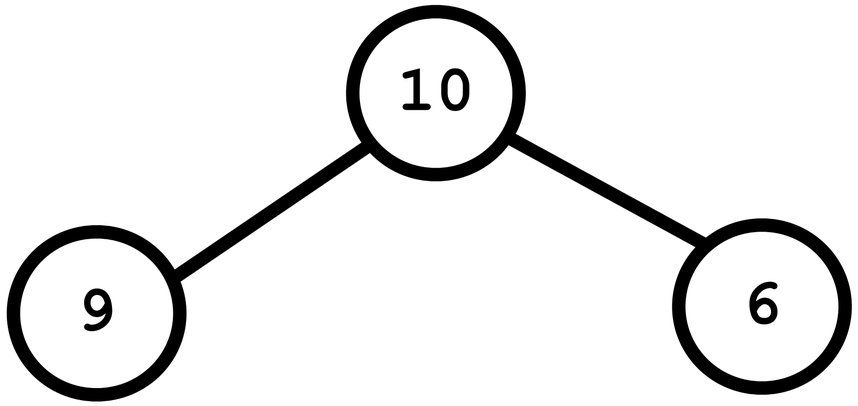
front

2
-1
3
4
5
6

i = 2

counter =7

Heap = {10, 9, 6, 5, 1, 2, 3}



front

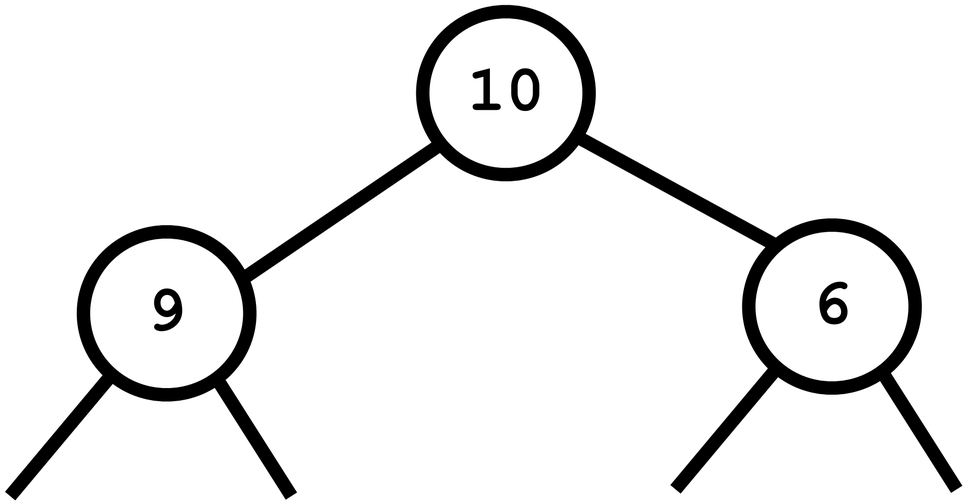
-1
3
4
5
6

i = 2

counter =7

pop()

Heap = {10, 9, 6, 5, 1, 2, 3}



front

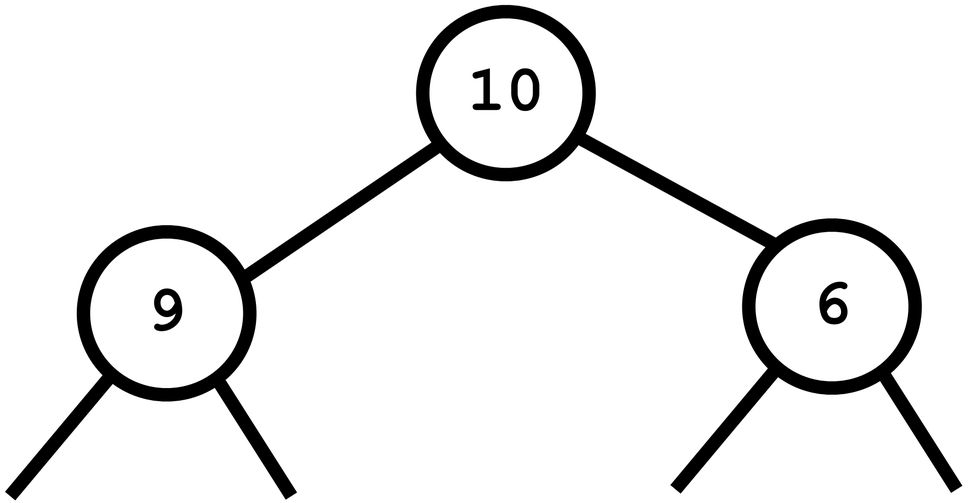
-1
3
4
5
6

i = -1

counter =7

cout newline

Heap = {10, 9, 6, 5, 1, 2, 3}



front

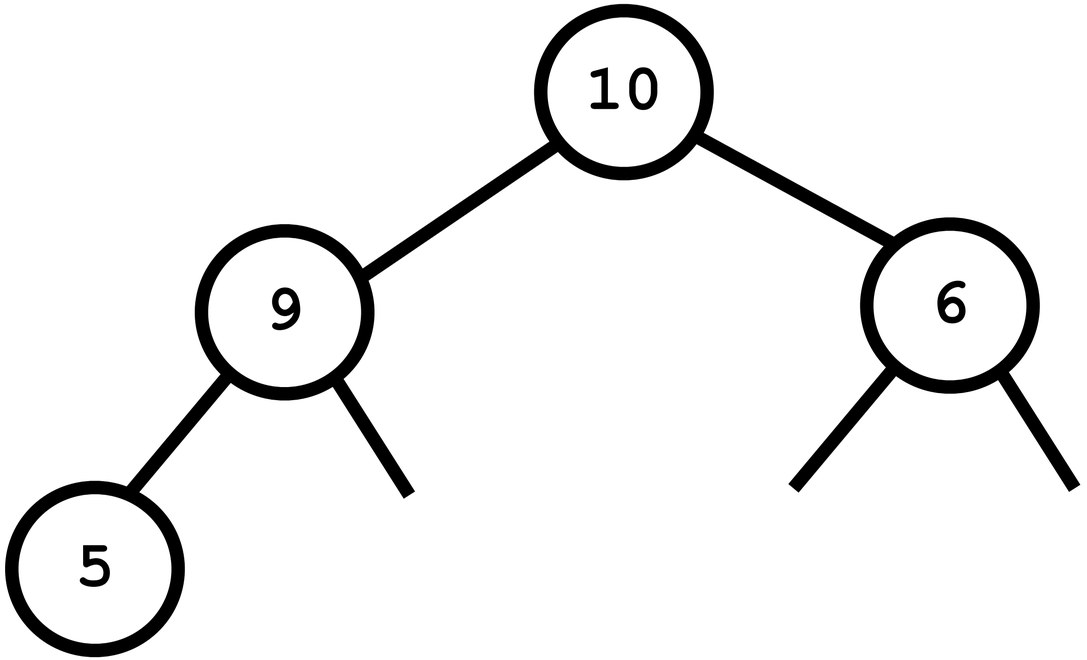
3
4
5
6

i = -1

counter =7

pop()

Heap = {10, 9, 6, 5, 1, 2, 3}



front

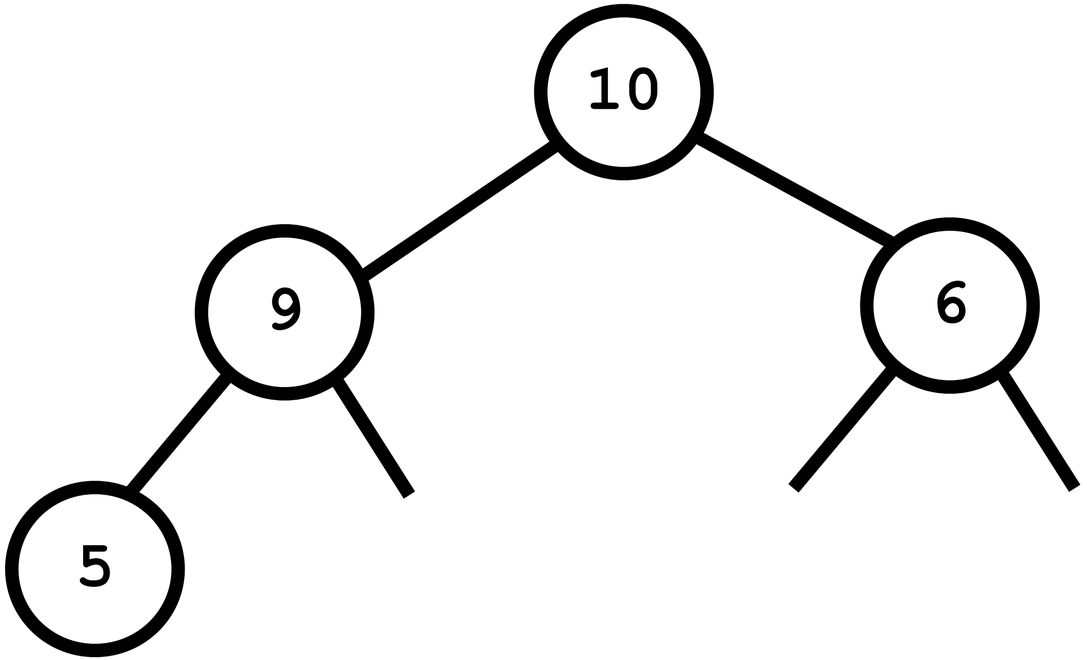
3
4
5
6

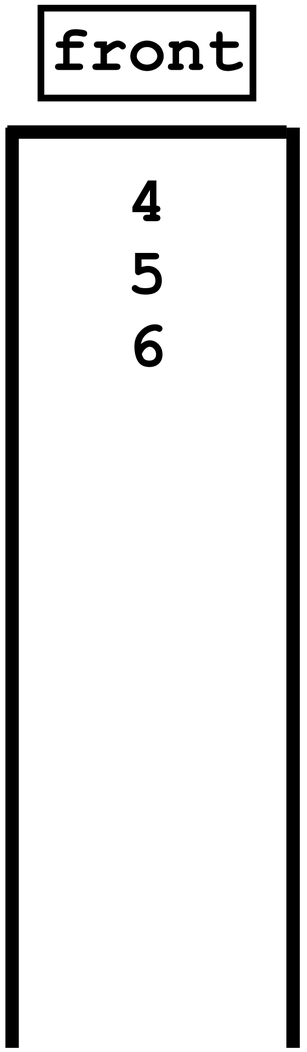
i = 3

counter =7

cout heap[3]

Heap = {10, 9, 6, 5, 1, 2, 3}



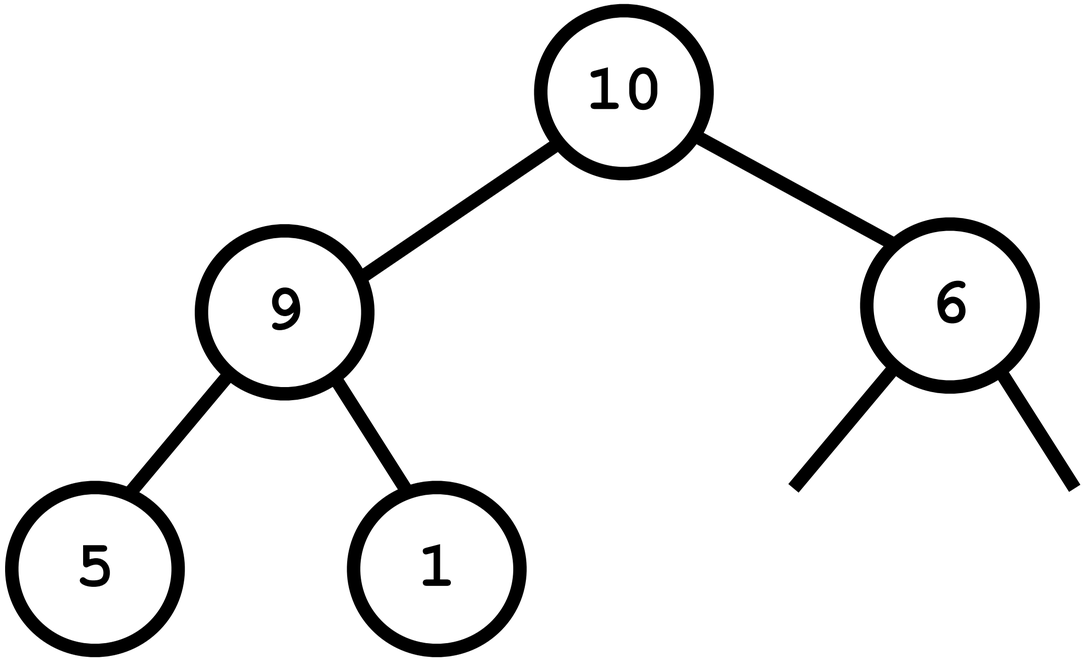


i = 3

counter =7

pop()

Heap = {10, 9, 6, 5, 1, 2, 3}



front

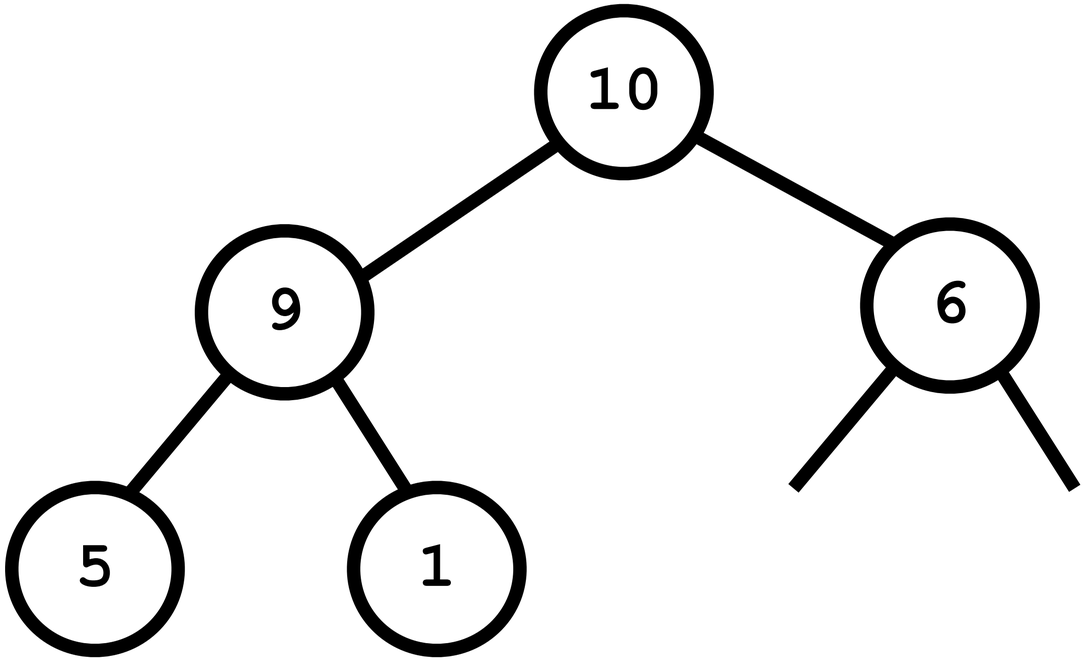
4
5
6

i = 4

counter =7

cout heap[4]

Heap = {10, 9, 6, 5, 1, 2, 3}



front

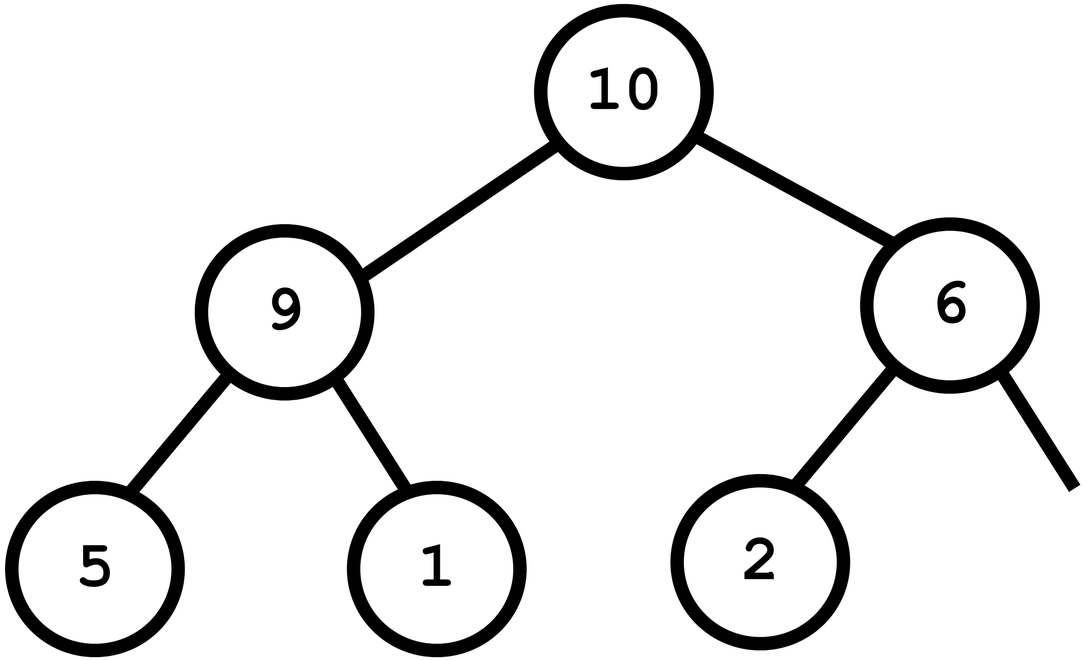
5
6

i = 4

counter =7

pop()

Heap = {10, 9, 6, 5, 1, 2, 3}



front

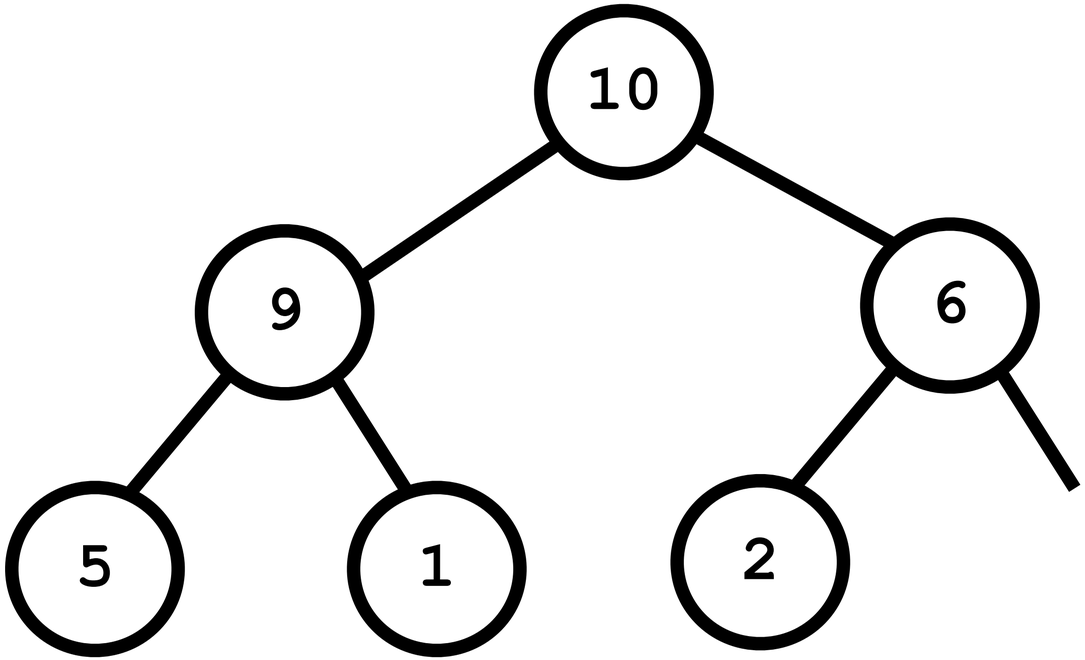
5
6

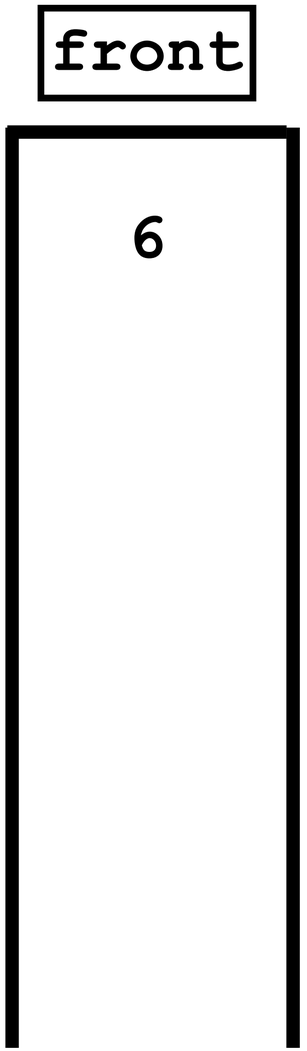
i = 5

counter =7

cout heap[5]

Heap = {10, 9, 6, 5, 1, 2, 3}



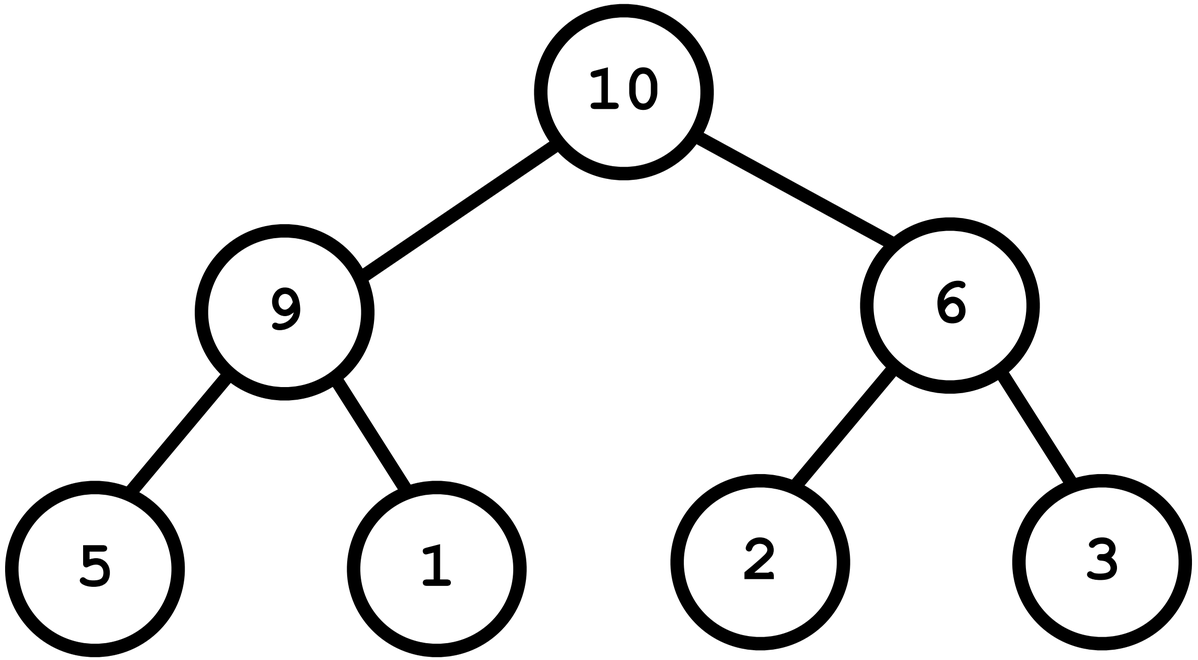


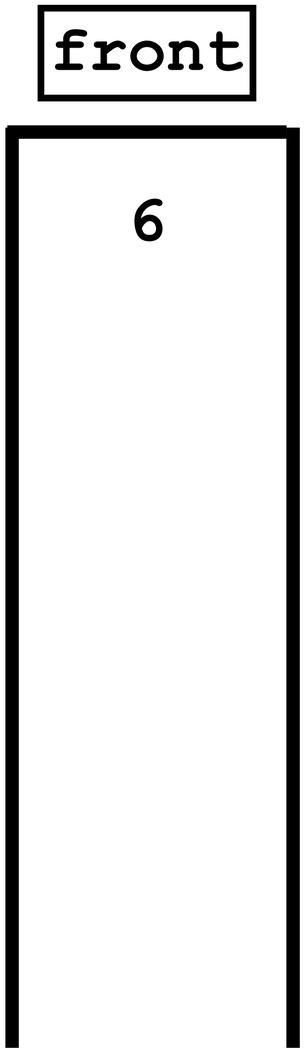
i = 5

counter =7

pop()

Heap = {10, 9, 6, 5, 1, 2, 3}



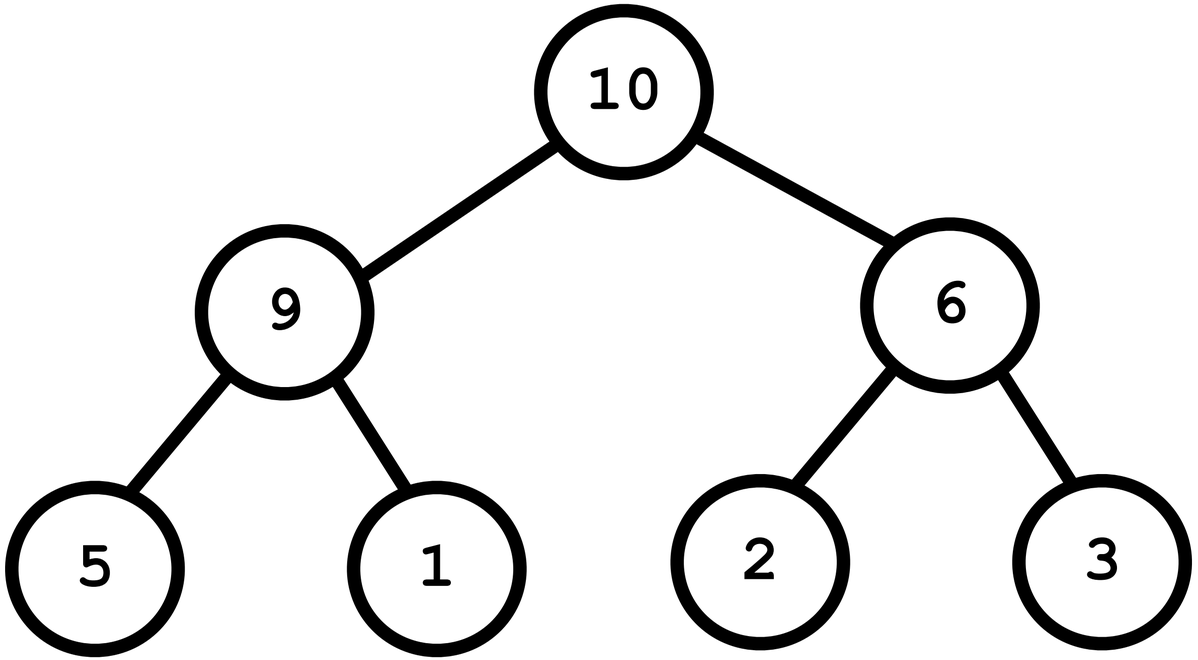


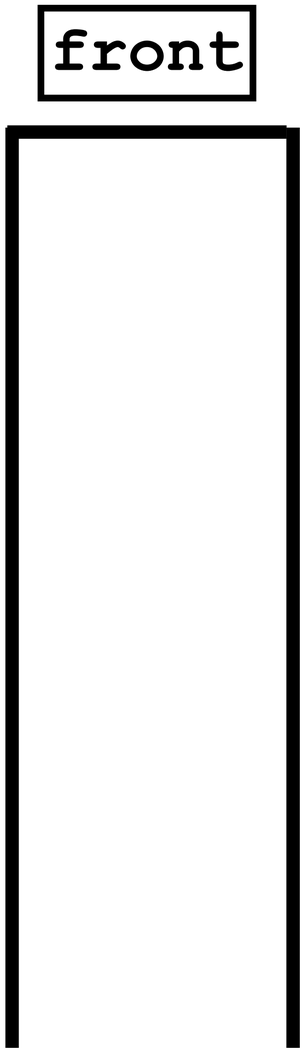
i = 6

counter =7

cout heap[6]

Heap = {10, 9, 6, 5, 1, 2, 3}





i = 6

counter =7

pop()

Queues Practice: Round Robin

Write the function `void roundRobin(process arr[], int n, int qt)` which executes round robin scheduling on the processes in `arr[]`. `n` is the number of processes in `arr[]` and `qt` is the quantum time. Your function should print the processes in the order in which they are finished (this means their time has reached 0). Example:

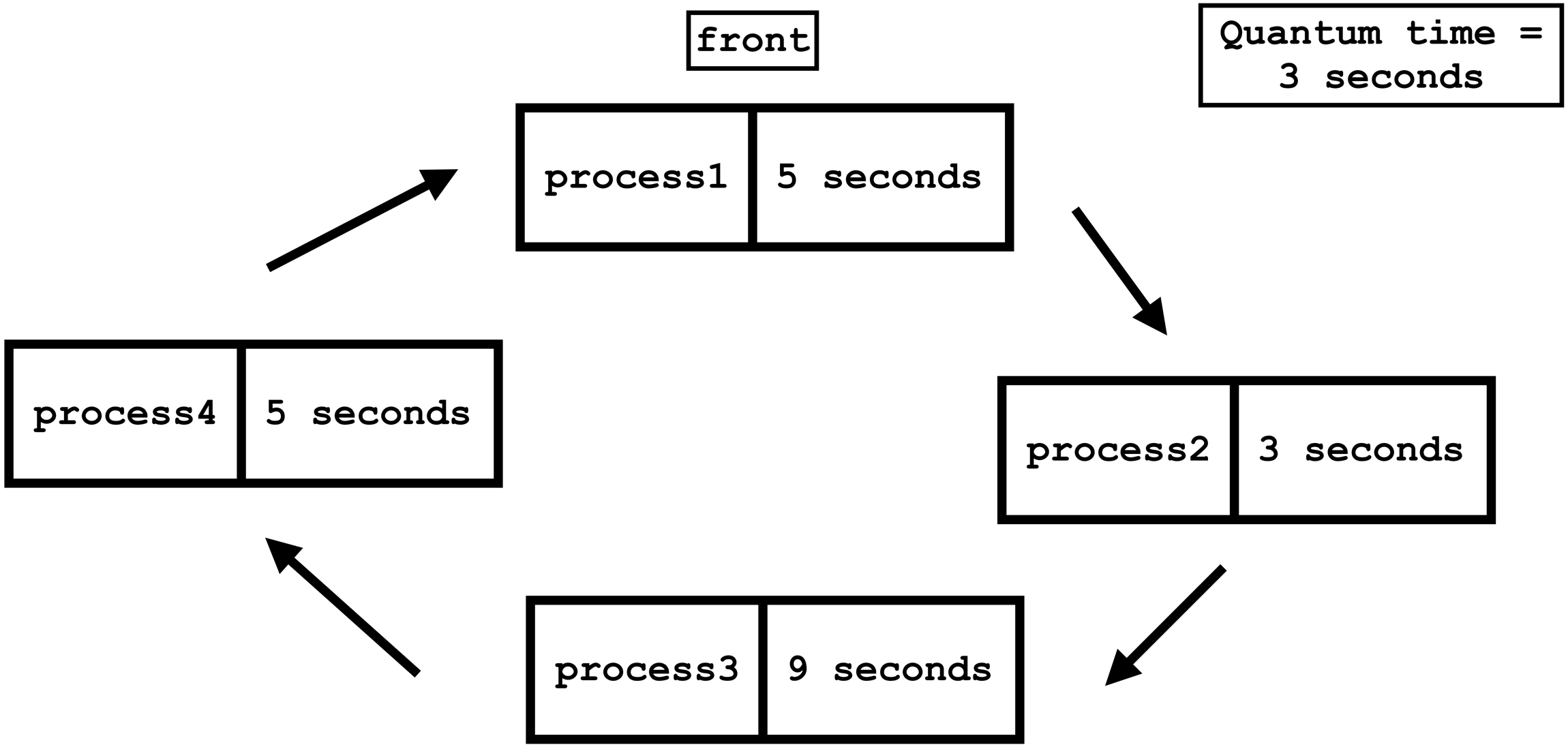
`roundRobin([P1(5), P2(3), P3(9), P4(5)], 4, 3)`
will produce:
`P2 P1 P4 P3`

```
struct process{
  string ID;
  int time;
};

void roundRobin(process arr[], int n, int qt){



}
```

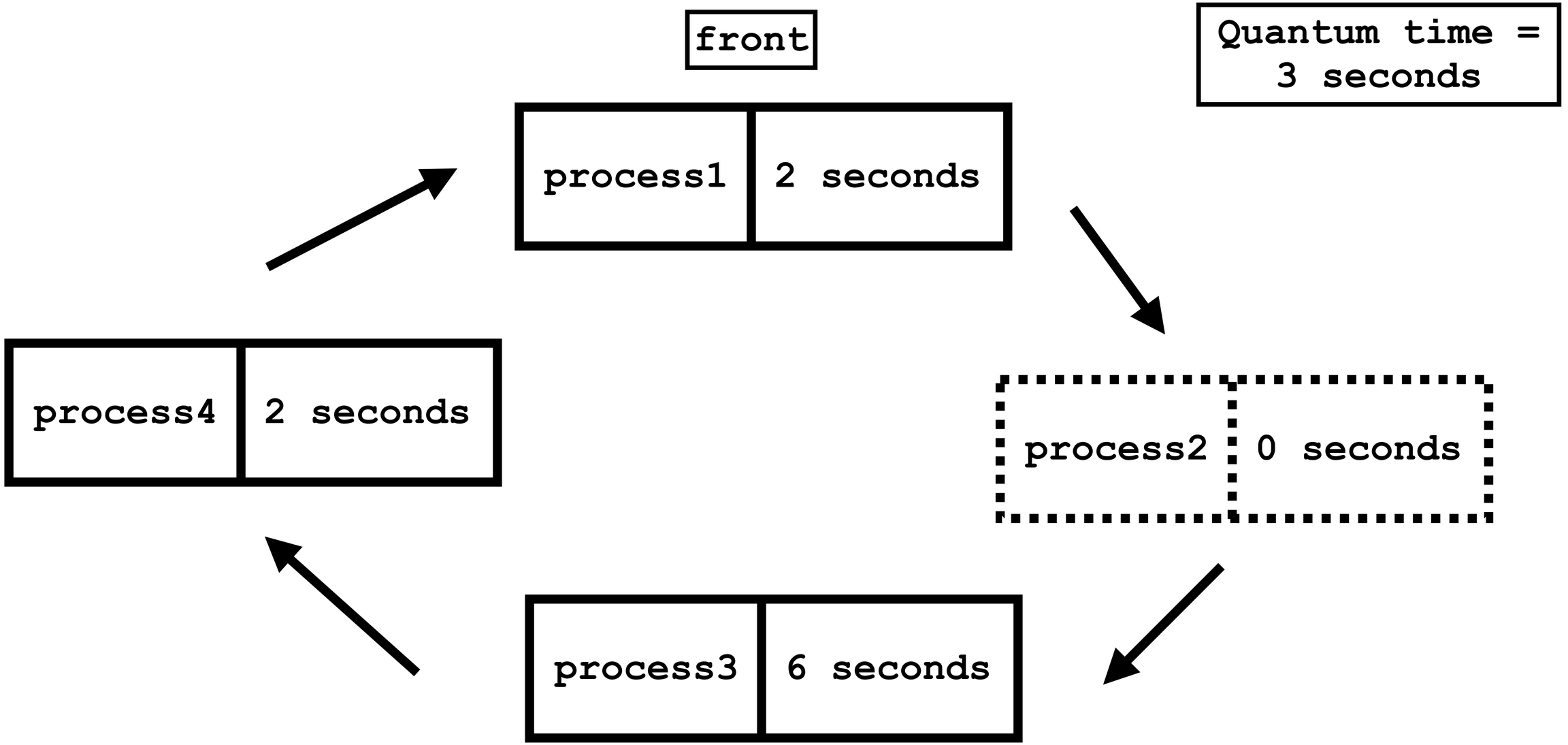
Queues Practice: Round Robin

```
void roundRobin(process arr[], int n, int qt){
  queue<process> q;
  process curP;
  for(int i = 0; i < n; i++){
    q.push(arr[i]);
  }
  while(!q.empty()){
    curP = q.front();
    q.pop();
    curP.time = curP.time - qt;
    if(curP.time <= 0)
      cout << curP.ID << " ";
    else
      q.push(curP);
  }
  cout << endl
}
```

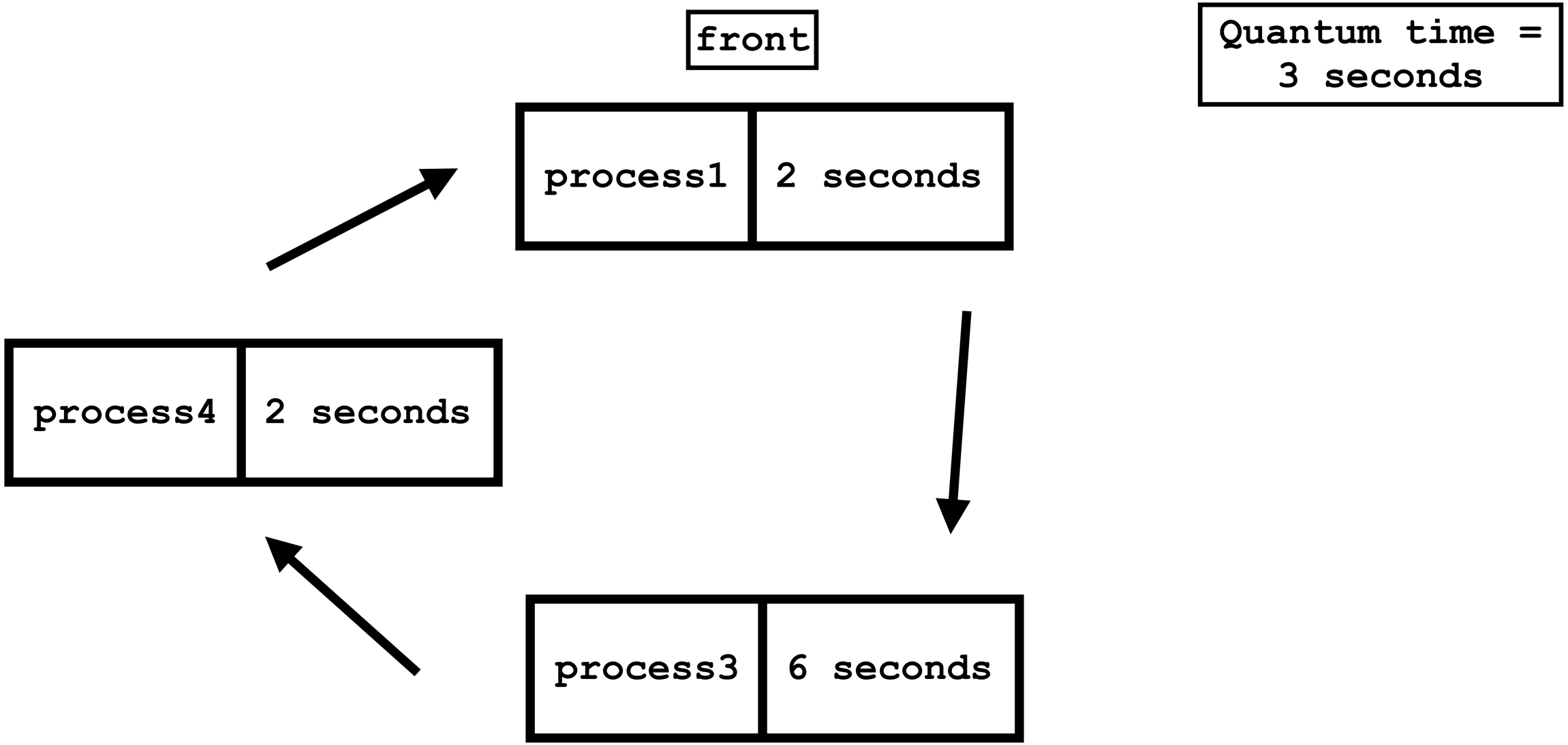
Initial State



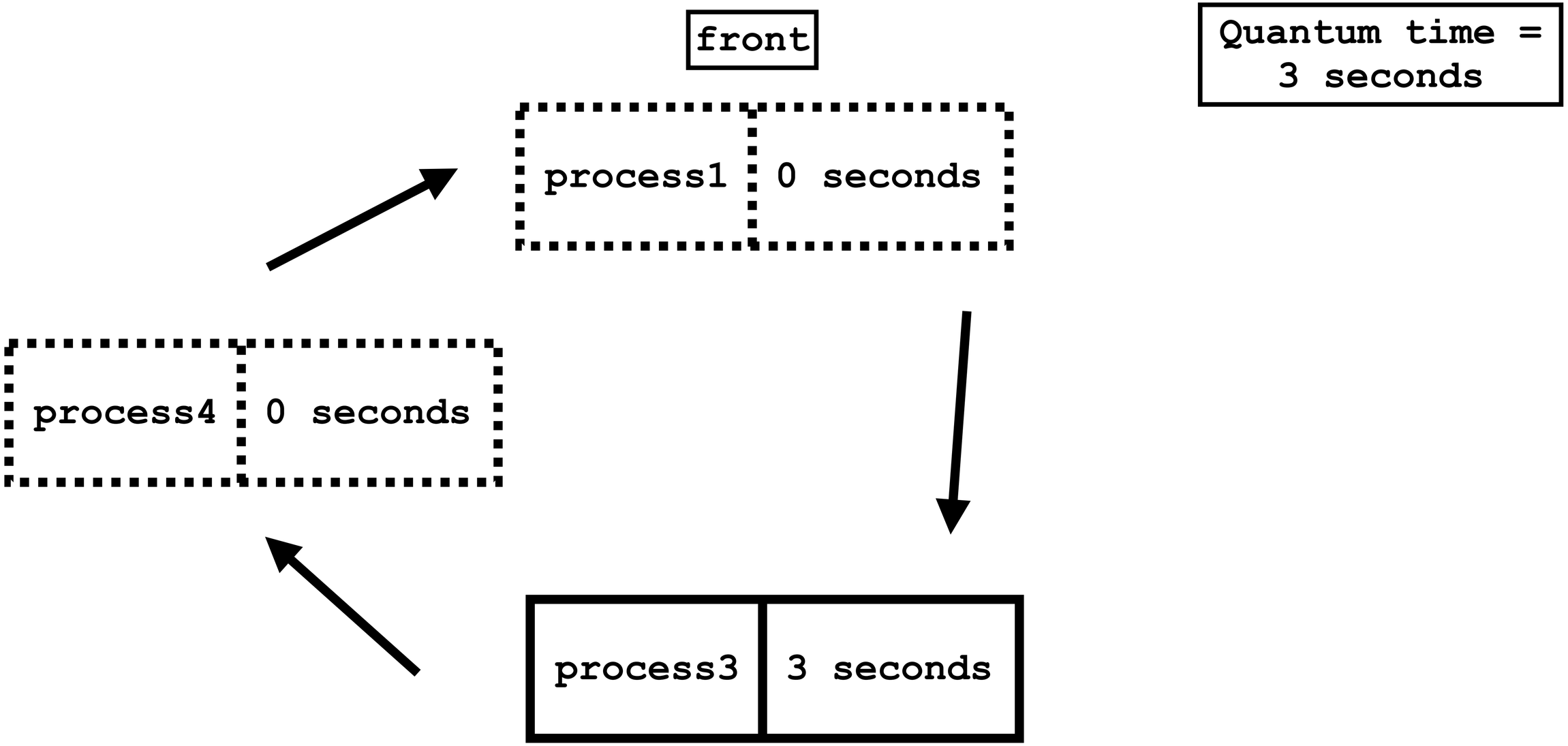
1st Round


front
Quantum time = 3 seconds
process1
2 seconds
process2
0 seconds
process3
6 seconds
process4
2 seconds

1st Round

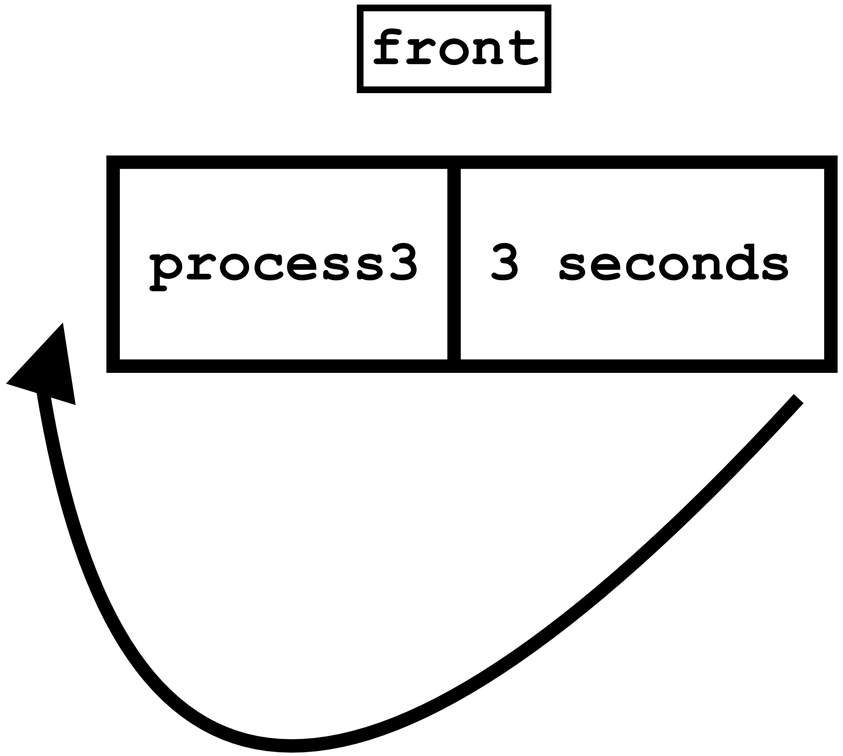

front
process1
2 seconds
process2
2 seconds
process3
6 seconds
process4
2 seconds
Quantum time =
3 seconds

2nd Round


front
process1
0 seconds
process4
0 seconds
process3
3 seconds
Quantum time =
3 seconds

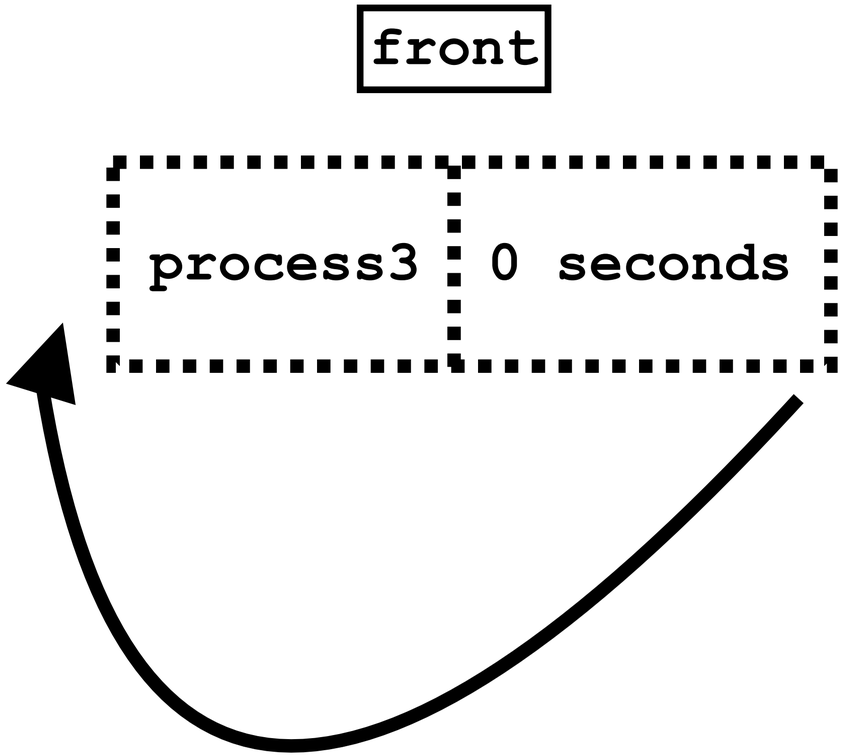
2nd Round

Quantum time =
3 seconds

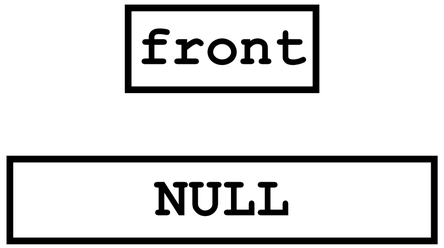


3rd Round

Quantum time =
3 seconds



3rd Round



We are now done with round robin.