1.0 Class Design

1.1 Deque

1.1.1 Data Structures

- The queue is composed of an array, as specified by the assignment.
- Track size and capacity as ints, could be unsigned ints but int for convenience. Tracking as ints means getting capacity or size is always O(1)
- All data is private since we don't want a user to be able to change the queue's size without adding or removing elements or trying to insert data into the middle of the queue.

1.1.2 Methods

- 4 methods for pushing or popping to the front or the back of the queue, 2 Methods for getting the capacity and size of the queue, all methods are public.
- During push or pops, array is doubled if the deque is full, size is incremented and decremented as items are added and removed

1.2 CPU

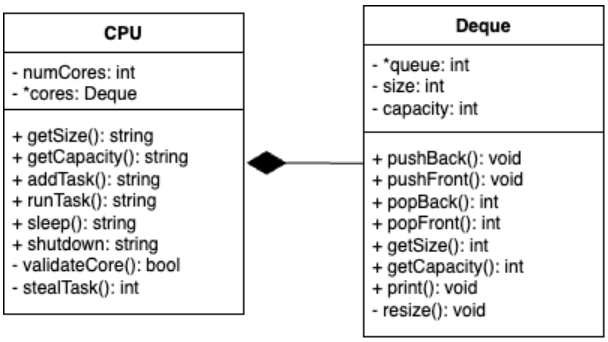
1.2.1 Data Structures

- Cores of the CPU is an array of deques. Chose to use an array of deques to enable iterating over the cores and cores only need to be accessed, never inserted or deleted so an array has good time complexity.
- Number of cores stored as an int, could be unsigned ints but int for convenience
- Both are private, user should not be able to change the number of cores or manipulate the tasks assigned to the cores.
- From the perspective of the CPU, the back of the deque for each core is the front of the core's task queue. This is to ensure the required time complexity.

1.2.2 Methods

- runTask() if a task is present will pop a task from the back of the deque, which since it is an array is O(1).
- addTask() will add tasks to the front of the deque, which since it is an array is O(C)
- All methods are public except the input validation methods and stealTask() since it is only called by the run method.
- All public methods return strings to be printed to the console.

2.0 UML Diagram



3.0 Runtime

3.1RUN

RUN N calls cpu.runTask(N). If core N is not empty, runTask calls popBack() on that core's deque. Deque is implemented as an array, accessing the last item in an array is O(1) since we know the number of elements in the list. "Removing" the item from the deque is done by decrementing size, making the value inaccessible.

```
string result = "core " + to_string(core) + " is running task " + to_string(cores[core].popBack());
```

```
int Deque::popBack()
{
    if (size == 0)
    {
        return -1;
    }
    size--;
    int task = queue[size];
    if (size > 0 && size <= capacity / 4 && capacity > 2)
    {
        resize(capacity / 2);
    }
    return task;
}
```

3.2 SPAWN

SPAWN calls addTask() which adds the task to the core with the fewest number of tasks. addTask adds a task to the core's Deque by calling pushFront(). Since the Deque is an array, to insert at the front of the list, each element in the list must be moved over which is O(C) where c is the number of elements in the core's Deque.

```
string CPU::addTask(int task)
{
    if (task <= 0)
    {
        return "failure";
    }

    int minSize = INT_MAX;
    int minIndex = -1;
    for (int i = 0; i < numCores; i++)
    {
        if (cores[i].getSize() < minSize)
        {
            minSize = cores[i].getSize();
            minIndex = i;
        }
    }
    cores[minIndex].pushFront(task);

    return "core " + to_string(minIndex) + " assigned task " + to_string(task);
}
```

```
void Deque::pushFront(int value)
{
    for (int i = size; i > 0; i--)
    {
        queue[i] = queue[i - 1];
    }
    queue[0] = value;
    size++;

    if (size == capacity)
    {
        resize(capacity * 2);
    }
}
```