enqueue(queue, element)

element = 20

1. if (queue -> count = queue -> capacity)
   return e_false
2. if (queue -> front = -1)
   queue -> front = 0
3. rear = (rear + 1) % queue -> capacity
4. queue -> items[queue -> rear] = element
5. queue -> count ++
6. return e_true

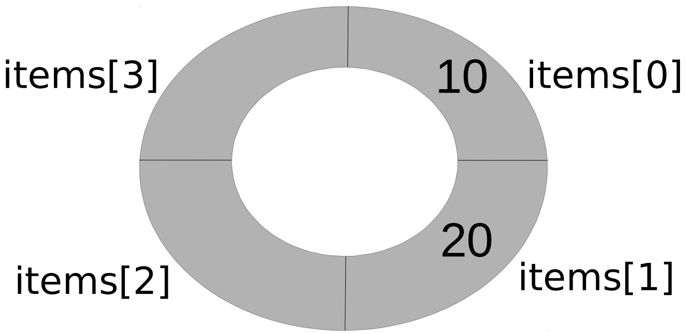
capacity = 4

count = 2

front = 0

rear = 1

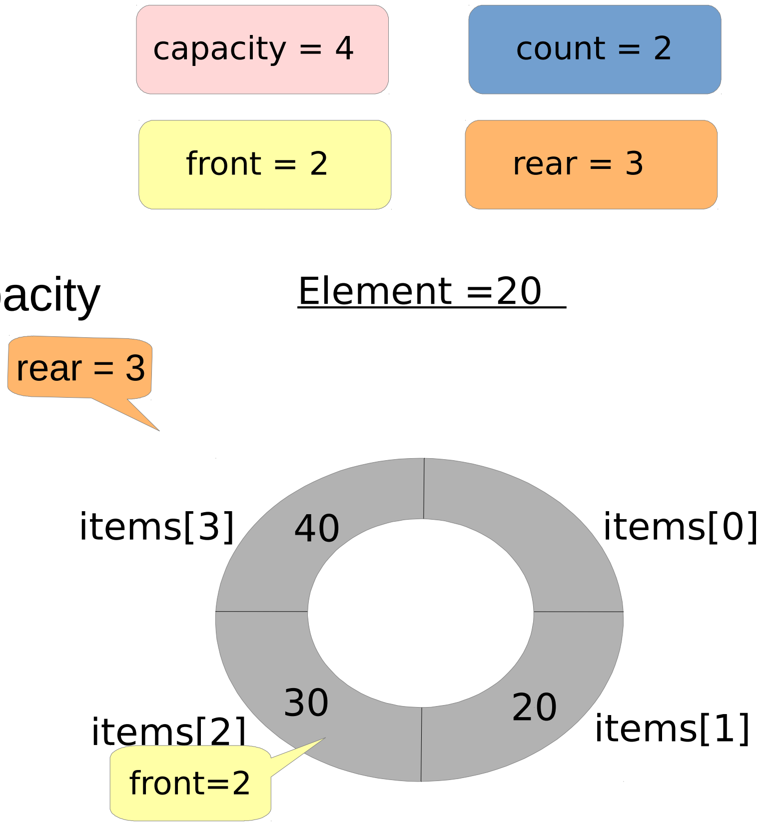
element = 10

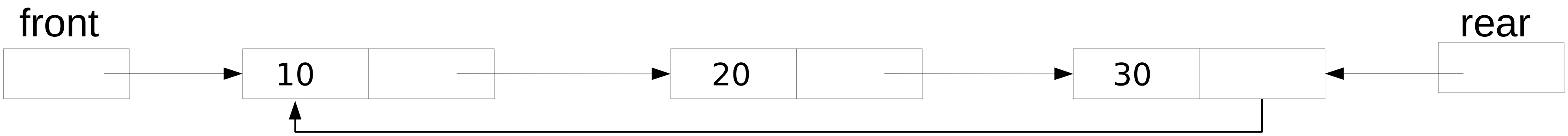


dequeue(queue, element)

1. if (queue -> count = 0)
   return e_false

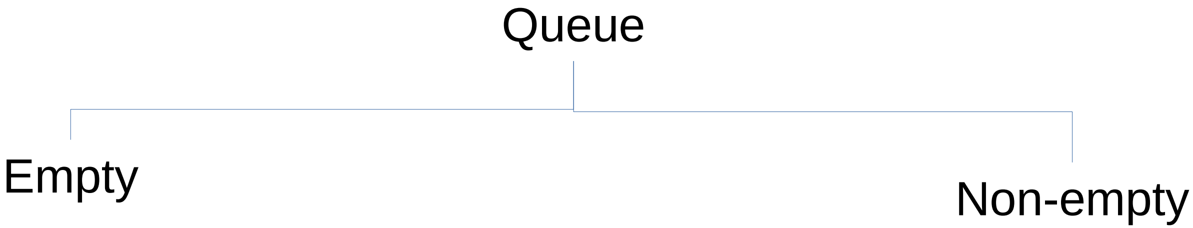
2. element = queue -> item[queue -> front]
3. queue -> front = (queue -> front + 1) % queue -> capacity
4. queue -> count --
5. return e_true



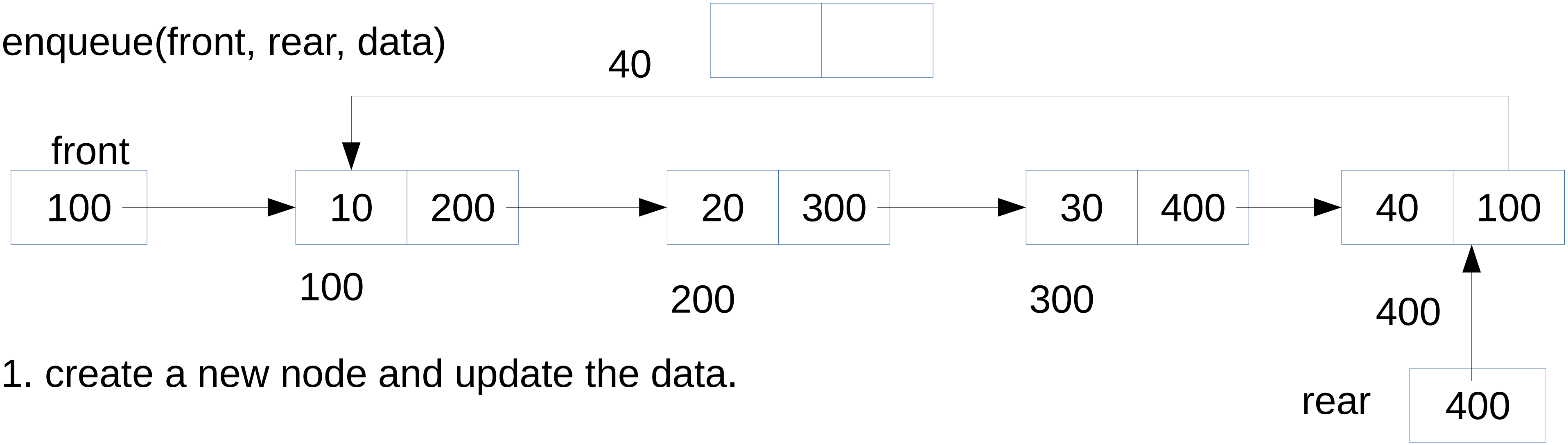


enqueue(front, rear, data)

data=10



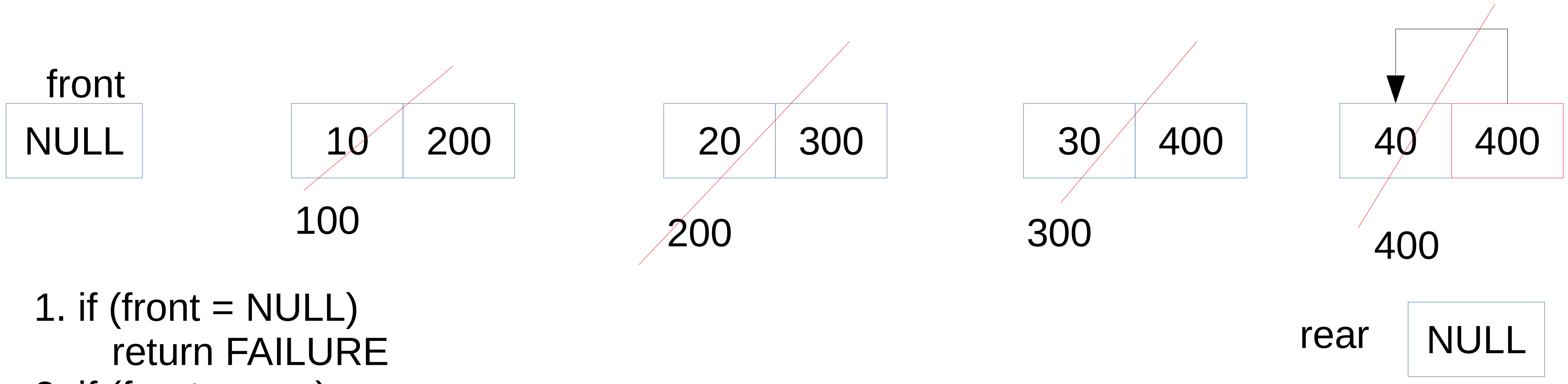
enqueue(front, rear, data)



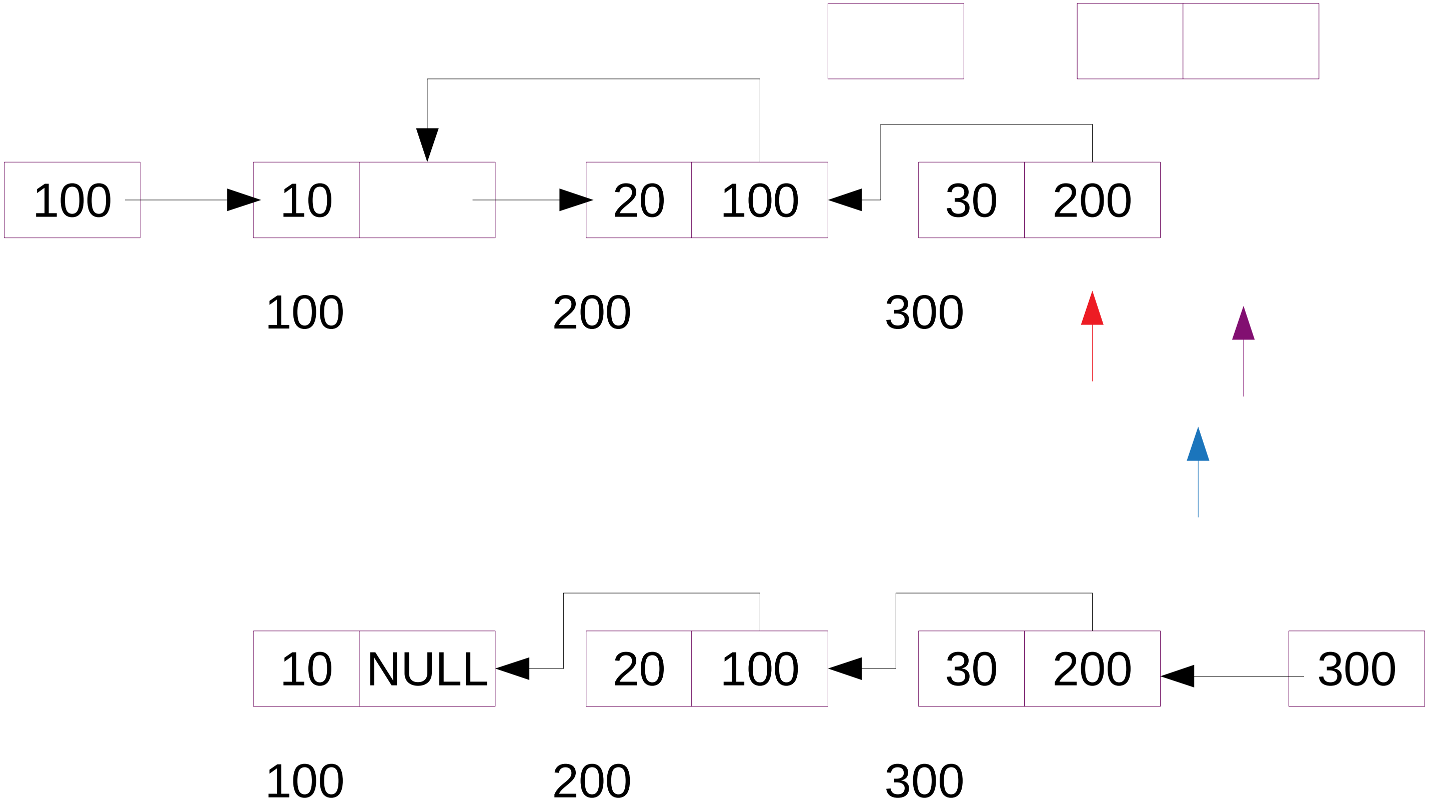
1. create a new node and update the data.

2. if (front = NULL)
    front = new
3. else
    rear -> link = new
4. rear = new
5. new -> link = front
6. return SUCCESS

dequeue(front, rear)



1. if (front = NULL)
   return FAILURE
2. if (front = rear)
   free(front)
   front = rear = NULL
3. else
   front = front -> link
   free(rear -> link)
   rear -> link = front
4. return SUCCESS



3 pointer

Current
Next
Prev

Current -> link = prev
Prev = current
Current = next
Next = current -> link

30 -> 20 -> 10 -> NULL

10 -> 20 -> 30 -> 40 -> 50 -> 60 -> NULL

sptr

fptr

Car1 -> 25 km/h

Car2 -> 50km/h

100KM

sptr = sptr -> link
fptr = fptr -> link -> link

mid = sptr -> data

get_nth_last(head, n, data)

N = 7

10 -> 20 -> 30 -> 40 -> 50 -> 60 -> NULL

NOTE : TRAVERSE THE LIST ONLY ONCE

COUNT = 6, N = 4

6 – 4 = 2

6 – 1 = 5

Searching technique:

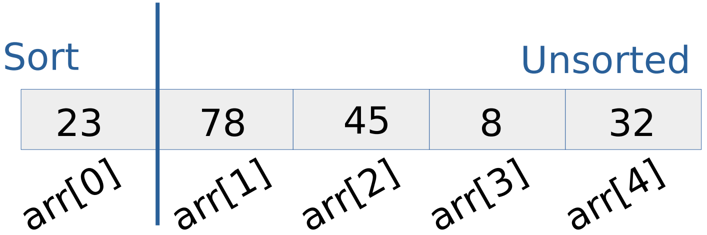
```
Binary_search(arr, size, key)

 high = size – 1, low = 0

 While (low <= high)
 {
       mid = (low + high) / 2
       if (arr[mid] = key)
             return mid / DATA_FOUND
       else if (key > arr[mid])
             low = mid + 1
       else
             high = mid - 1
 }
 Return DATA_NOT_FOUND / -1
```

Key = 2.5

0	1	2	3	4
1	2	3	4	5

Low	high	mid
0	4	2
0	1	0
1	1	1

insertion_sort(arr, size)



Key = arr[1]
Sort = 0
If (key < arr[sort])

insertion_sort(arr, size)

I = 3

Sort Unsorted

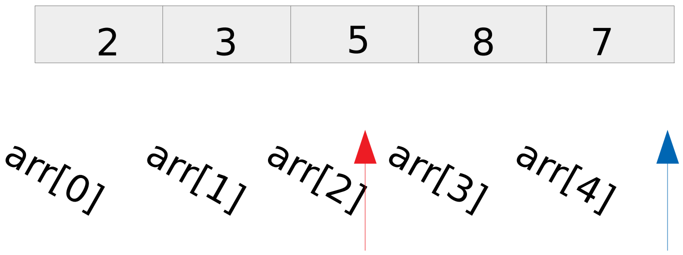
8	23	45	78	32
arr[0]	arr[1]	arr[2]	arr[3]	arr[4]

Key = 8
Sort = -1

```
For (i = 1; i < n; i++)
{
        Key = arr[i]
        Sort = i - 1
        while (sort >= 0 AND key < arr[sort])
        {
                arr[sort + 1] = arr[sort]
                sort--
        }
        arr[sort + 1] = key
}
```

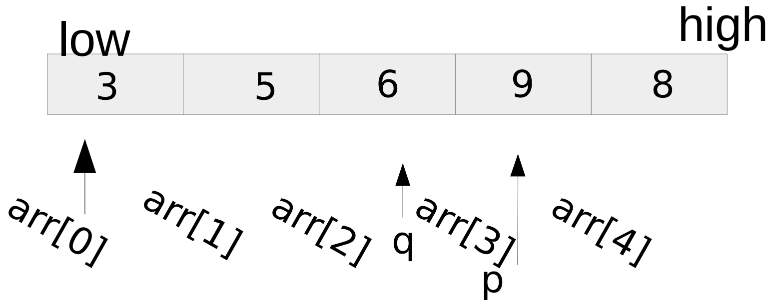
Selection_sort(arr,size)

I = 2




```
for(i = 0; i < size; i++)
{
        cur_min = i;
        for(cur_item = i + 1; cur_item < size; cur_item++)
        {
                if (arr[cur_min] > arr[cur_item])
                        cur_min = cur_item
        }
        if (i != cur_min)
                swap(arr[i], arr[cur_min])
}
```

```
partition(arr, low, high)
{
	pivot = low, p = low + 1, q = high

	while (p <= q)
	{
		while(arr[pivot] > arr[p])
		{
			p++
		}
		while (arr[pivot] < arr[q])
		{
			q--
		}
		if (p < q)
			swap(arr[p], arr[q])

	}
	swap(arr[pivot], arr[q])
	return q
}
```

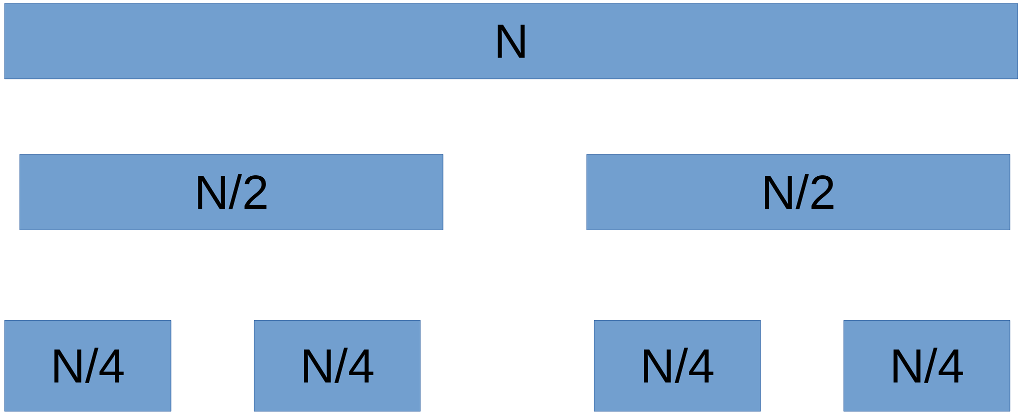


0 1

3 5

pivot Q

```
quick_sort(arr, low, high)
{
	if(low <= high)
	{
		index = partition(arr, low, high)
		quick_sort(arr, low, index - 1) //LSA
		quick_sort(arr, index + 1, high) //RSA
	}
}
```



(n * log n)

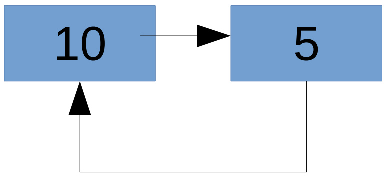
O(n log n)

Stable sort

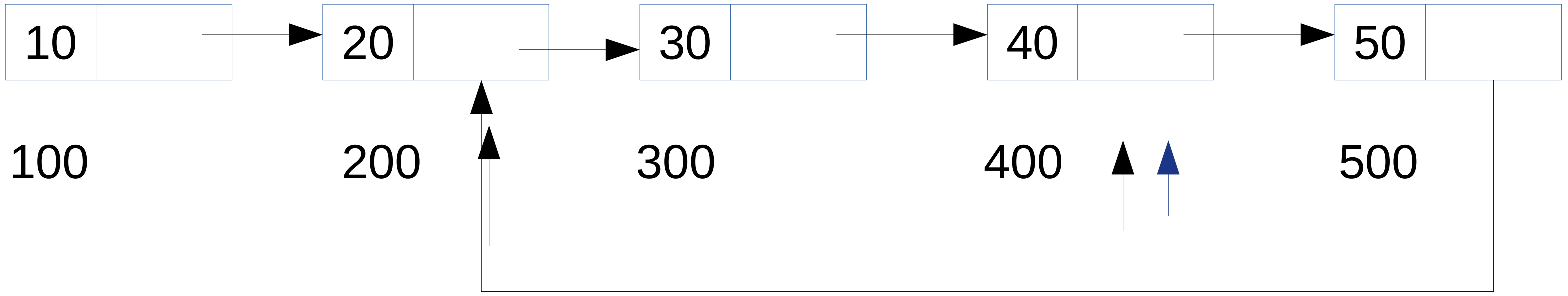
External sort

Ndata = 5

5 -> 10 -> 20 -> 30 -> 40 -> 45 -> 50 -> 55 -> NULL

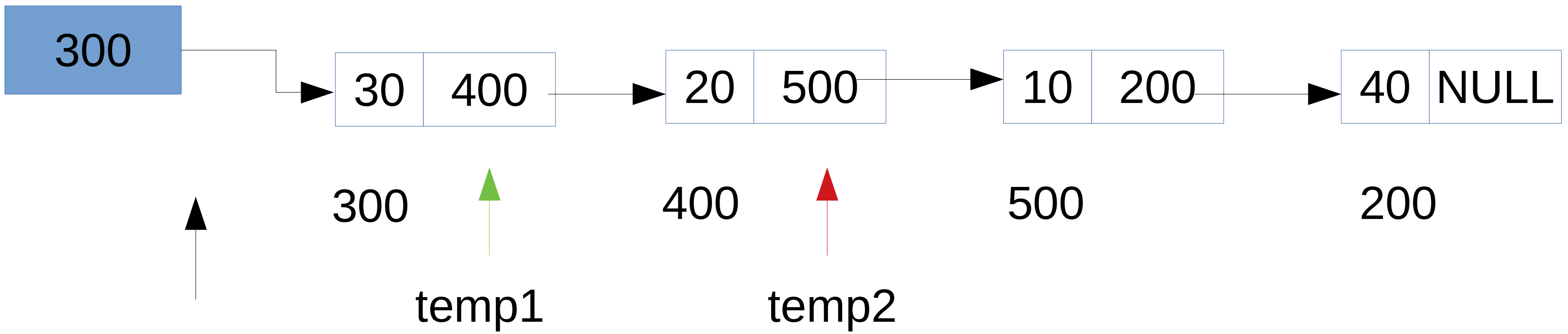


If (first node)
    head = new
else
    Prev -> link = new
New -> link = temp



create_loop(head, data)

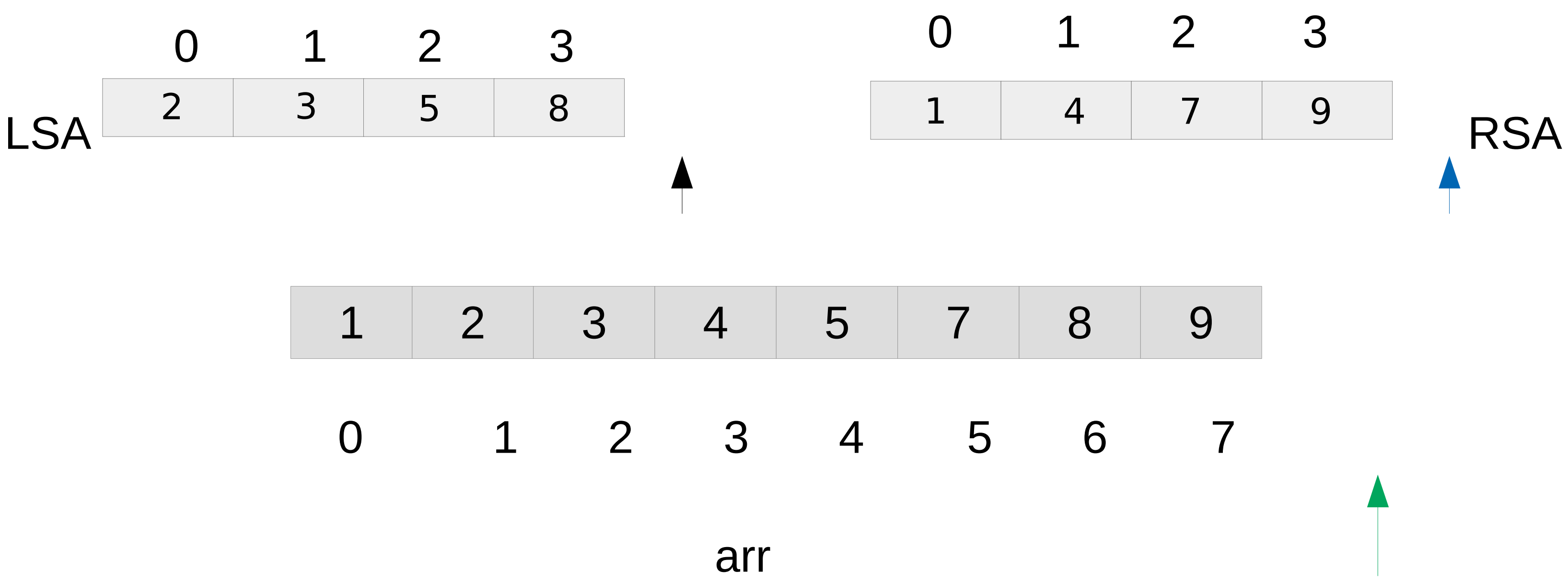
Data = 20



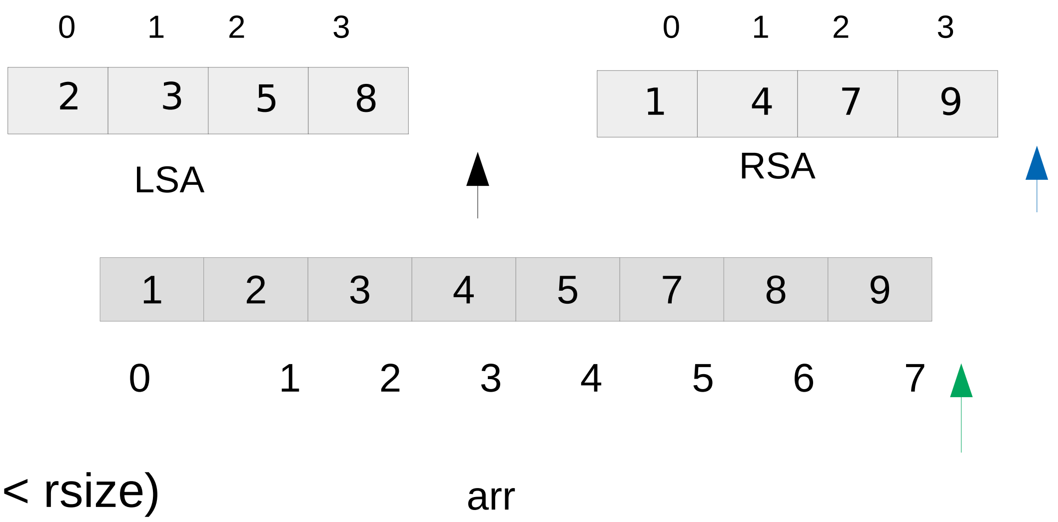
1. Bubble sort

NOTE : Swap nodes

```
For (i = 0; i < no.of_nodes; i++)
{
        for(j = 0; j < no.of_nodes – i – 1; j++)
        {
                //Logic
                if(temp1 -> data > temp2 -> data)
        }
}
```


0 1 2 3
2 3 5 8
LSA
0 1 2 3
1 4 7 9
RSA
1 2 3 4 5 7 8 9
0 1 2 3 4 5 6 7
arr

```
Merge(arr, size, LSA, lsize, RSA, rsize)

1. i = j = k = 0

2. loop(i < lsize AND j < rise)
   {
       if (LSA[i] > RSA[j])
       {
          arr[k] = RSA[j++]
       }
       else
       {
          arr[k] = LSA[i++]
       }
       k++
   }
3. while (i < lsize)
   {
     arr[k++] = LSA[i++]
   }
```

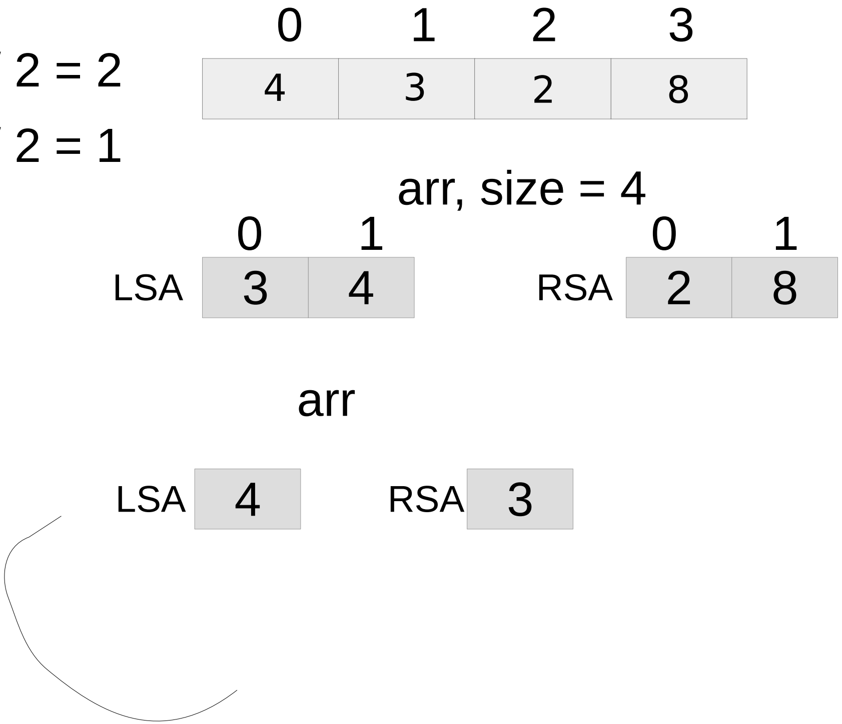



```
4. while (j < rsize)
   {
     arr[k++] = RSA[j++]
   }
```

```
merge_sort(arr, size)
{
	if (size = 1)
		return
	mid = size / 2
	LSA = Memalloc(sizeof(int) * mid)
	//Check memory allocated
	for (i = 0 upto mid – 1)
		LSA[i] = arr[i]
	RSA = Memalloc(sizeof(int) * (size - mid))
	//Check memory allocated
	for (i = 0 upto (size - mid) – 1)
		RSA[i] = arr[i + mid]
	merge_sort(LSA, mid)
	merge_sort(RSA, (size – mid))
	merge(arr, size, LSA, mid, RSA, (size - mid))
	free(LSA)
	free(RSA)
}
```

Mid = 4 / 2 = 2

Mid = 2 / 2 = 1



Assignment – 10, Merge and sort two linked list.

Head1 10 -> 30 -> 20 -> 70 -> 40 -> 60 -> 50 -> NULL

Head2 40 -> 60 -> 50 -> NULL

10 -> 20 -> 30 -> 70 -> 40 -> 50 -> 60

Cases:
1. Head1 = NULL
   Head2 = NULL -> LIST_EMPTY
2. head1 = NULL / empty
   Head2 = not NULL / not empty -> Update list2 1st node address in head 1, sort(head1)
3. head1 = not NULL / not empty
   Head2 = NULL / empty -> don't merge, sort(head1)

10
13 -> class

18th april -> deadline for 23 assignments
5 more are pending