b)

1.

if (isEmpty()) {

return -1;

}

int x = a[head];

if (head == capacity - 1) {

head = 0;

}

else {

head = head + 1;

}

--length;

return x;

Like the code shown above, for remove(), since I have a data member called head in my class Data, so every time the remove is called, just return a[head] where a is an array as data member in the class. This only take O(1). And, every time when an key is removed, I will re-set head value which depends on whether the head just removed is the end of the inner array.

int getValue(int i) {

if (i == tail||i >= capacity) {

return -1;

}

if (head < tail) {

if (i < head||i>=tail)

return -1;

}

if (tail > head) {

if (i < head && i >= tail)

return -1;

}

return a[i];

}

Like code shown above, the getValue will only take O(1) time since there is no iteration inside the function. One thing to note is that since there would be a lot of different kinds of distribution of elements in the FIFO data structure, I set three cases to prevent an empty space of the array being returned. Where the empty space should be behind the tail and before the head.

2.

void add(int x) {

if (length == capacity - 1) {

expansion();

}

a[tail] = x;

if (tail == capacity - 1) {

tail = 0;

}

else {

tail = tail + 1;

}

++length;

}

void expansion() {

capacity \*= 2;

int\* a2 = new int[capacity];

for (int i = 0; i <= length; i++) {

a2[i] = a[i];

}

delete[]a;

a = a2;

}

Here, I used a expansion function to solve the case that the number of elements added has been equal to capacity – 1. It means that there will be always one entry of the data structure being kept empty which is pointed by the data member tail. So every time the case is met, I will double the size of array by using dynamic memory allocation and freeing the original memory space and replacing a by new address a2. This process is going to take O(N) since I will need to copy the contents of original array to the new array.