**Homework 4\_1 [Double\_linked\_stack]**

**Variable analysis**

**Doubly linked stack**

: A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. which is “head” of the stack where pushing and popping items happens at the head of the list. first node have null in link field and second node link have first node address in link field and so on and last node address in “top” pointer.  
The main advantage of using linked list in stack is that it is possible to implements a stack that can shrink or grow as much as needed(dynamic allocation), while using array will put a restriction to the maximum capacity of the array which can lead to stack overflow.

|  |  |
| --- | --- |
| **DlistNode** | |
| **type** | **name** |
| element | data |
| DlistNode\* | llink |
| DlistNode\* | rlink |

- data : value

- llink : address to left node

- rlink : address to right node

|  |  |
| --- | --- |
| **DlistStackType** | |
| **type** | **name** |
| DlistNode\* | top |

- top : points the top of stack

**Function analysis**

1)init  
: initialize top pointer to NULL.

void init(DlistStackType\* s)

{

s->top = NULL;

}

2) is\_empty

: check if stack is empty or not. If empty, return 1.

int is\_empty(DlistStackType\* s)

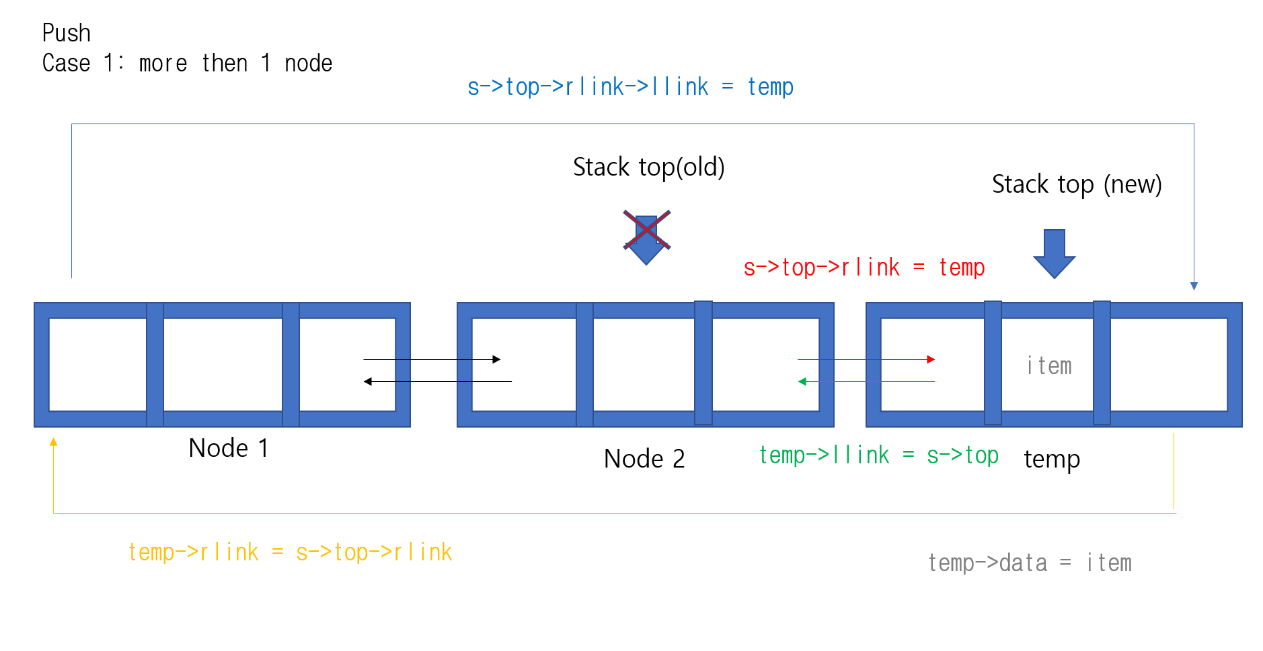
{

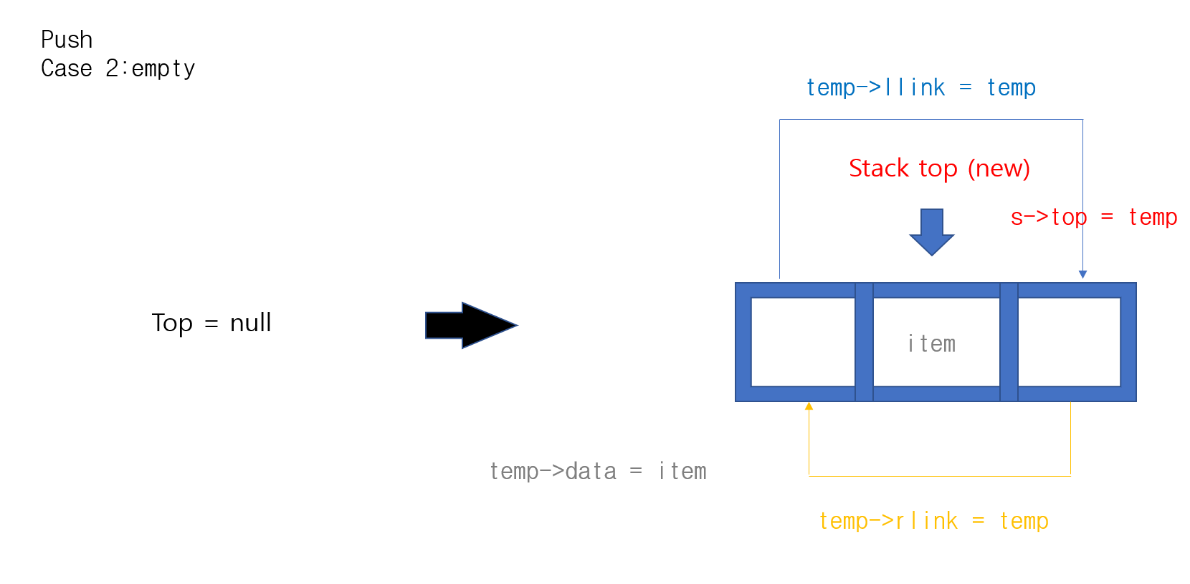
return (s->top == NULL);

}

3)push

: Add new element at the top of the stack. There are two cases in push()





implementing the function as shown in the figure above.

void push(DlistStackType\* s, element item)

{

DlistNode\* temp = (DlistNode\*)malloc(sizeof(DlistNode)); //make new node to insert

if (temp == NULL) {

fprintf(stderr, "Memory allocation error\n");

return;

}

else {

temp->data = item; //item assign

if (is\_empty(s)) {

// make self pointing node

temp->rlink = temp;

temp->llink = temp;

}

else {

temp->rlink = s->top->rlink; // last ->first

temp->llink = s->top; // old last -> new last

s->top->rlink->llink = temp; // first -> new last

s->top->rlink = temp; // old last -> new last

}

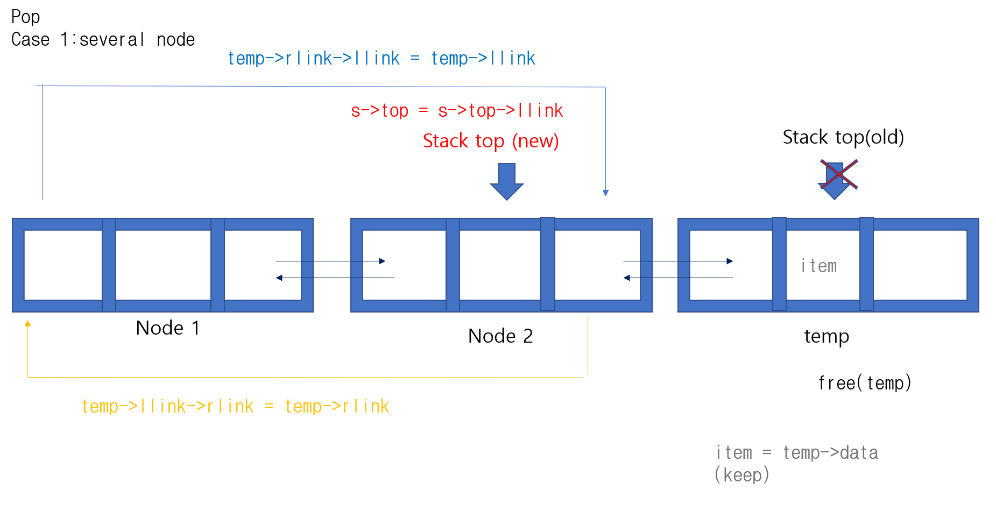
s->top = temp; //top renewal (common)

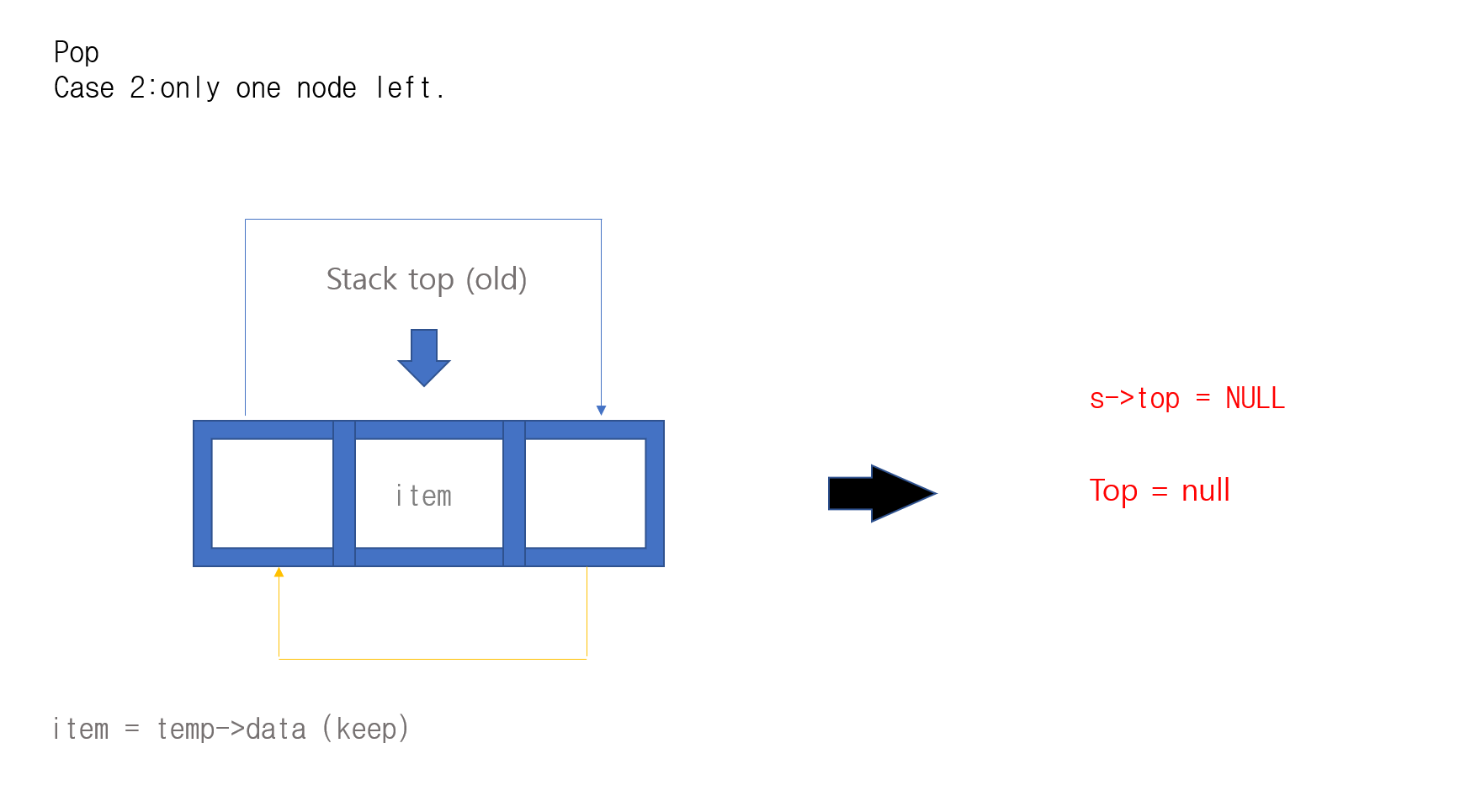
}

}

4)pop

: Return top element from the stack and move the top pointer. There are two cases in pop().





implementing the function as shown in the figure above.

element pop(DlistStackType\* s)

{

if (is\_empty(s)) { //if stack is empty

fprintf(stderr, "Stack is empty\n"); //there is no element to take out

exit(1);

}

else {

DlistNode\* temp = s->top; //pointer to the removed

int item = temp->data; //keep data

if (s->top == s->top->llink) //if there is only one node exist

{

s->top = NULL;

}

else {

temp->llink->rlink = temp->rlink; //new last ->fisrt

temp->rlink->llink = temp->llink; //first-> new last

s->top = s->top->llink; //top renewal

}

free(temp); //deallocate the removed

return item;

}

}

5)peek

: Returns the element at the top of the stack without removing it. It just return data that stack top points.

element peek(DlistStackType\* s)

{

if (is\_empty(s)) {

fprintf(stderr, "Stack is empty\n");

exit(1);

}

else {

return s->top->data;

[Result]

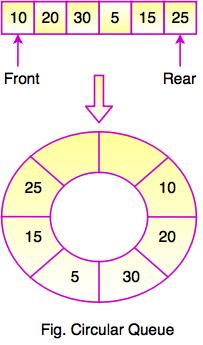


**Homework 4\_2 [Simulation]**

**Variable analysis**

**Circular Queue**

Circular Queue works by the process of circular increment i.e. when we we reach the end of the queue, we start from the beginning of the queue.

****

|  |  |
| --- | --- |
| **Queuetype** | |
| **type** | **name** |
| element [MAX\_QUEUE\_SIZE] | queue |
| int | front |
| int | rear |

- element : information about customers

- front : before index of fist element

- rear: index of last element

**Customer structure**

**:** used as element of circular queue

|  |  |
| --- | --- |
| **element** | |
| **type** | **name** |
| int | id |
| int | arrival\_time |
| int | service\_time |

- id: order of arrival

- arrival\_time : the time when customer arrived

- service\_time : time taking to complete service for each customer.

**Simulation variables**

- Duration : Simulation time

- arrival\_prob : probability of new customer to arrive each time.

- max\_serv\_time 5: maximum service time assigned for one customer

- waited\_time : Number of customers served

- customers : Total number of customers

- served\_customers : Number of customers served

- clock : used to check current time

**Function analysis**

**Circular Queue function**

**:** We use a % (modular operator) to reset the index when we circulate more than once.

1)is\_empty

// Empty state detection function

int is\_empty(QueueType \* q)

{

return (q->front == q->rear); // front = rear

}

: When front and rear point to the same index, the queue is empty. The reason % is not used here is that gap between the front and rear can not be more than one lap. It is because we defined the full-state queue as leaving one blank blank.

2)is\_full

// Full state detection function

int is\_full(QueueType \* q)

{

return ((q->rear + 1) % MAX\_QUEUE\_SIZE == q->front); //rear+1=front

}

: We define the full state in this way to distinguish between the full state and the empty state. When the front is one step ahead of the rear, queue is full.

3)enqueue

1) // Insert function

void enqueue(QueueType \* q, element item)

{

if (is\_full(q))

printf("Queue is full\n");

q->rear = (q->rear + 1) % MAX\_QUEUE\_SIZE; // rear ++.

q->queue[q->rear] = item; // insert item at rear

}

: (=push\_back) This function is used to insert the new value in the Queue. The new element is always inserted from the rear end. First, check if the queue is full. . If queue is full, the elements can not be added anymore. Then, circularly increase the REAR index by 1 . and add the new element in the position pointed to by REAR

4)dequeue

4) // delete function

element dequeue(QueueType \* q)

{

if (is\_empty(q))

printf("Queue is empty\n");

q->front = (q->front + 1) % MAX\_QUEUE\_SIZE; //front++

return q->queue[q->front]; //return item at front

}

: (=pop\_front) This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end. First, check if the queue is empty. If queue is full, there is nothing to be dequeued. Then, circularly increase the FRONT index by 1. And return the value pointed by FRONT. If the rear reaches the end, next it would be at the start of the queue.

**Simulation function**

1) double random

double random() { // Real random number generation function between 0 and 1

return rand() / (double)RAND\_MAX;

}

: RAND\_MAX is always bigger or equal to rand(). Thus it generates number between 0 and 1.

2)is\_customer\_arrived

int is\_customer\_arrived()

{

if (random() <arrival\_prob)

return true; { //arrived

else return false; { //not arrived

}

**:** It determines whether a new guest has arrived or not. If random number generated is smaller than ‘arrival\_prov’, assume that new customer comes in the bank.

3) insert\_customer

void insert\_customer(int arrival\_time)

{

element customer;

customer.id = customers++; { //set customer id

customer.arrival\_time = arrival\_time;

customer.service\_time = (int)(max\_serv\_time \* random()) + 1; //

enqueue(&queue, customer); //push customer into queue

printf("Customer %d comes in %d minutes. Service time is %d minutes.\n", customer.id, customer.arrival\_time, customer.service\_time);

}

: It inserts newly arrived customer into queue. It sequentially set the ID of the guest who arrives at the corresponding time and sets the service time randomly. Service time should be set between 1 ~max\_serv\_time +1.(not zero) Service time required by the customer is generated using a random number. When allocation of member variables in the element is completed, put it into the queue. And print out information about new customer arrived.

4) remove\_customer

int remove\_customer()

{

element customer;

int service\_time = 0;

if (is\_empty(&queue))//if there is no customer waiting

{

printf("there is no customer waiting to service\n");

return 0;

}

customer = dequeue(&queue);

service\_time = customer.service\_time - 1; // set service\_time

served\_customers++; ; // served\_customers increases by one

waited\_time += clock - customer.arrival\_time; // waited\_time= current - arrival

printf("starts service new customer in %d minutes. Wait time was %d minutes.\n", customer.id, clock, clock - customer.arrival\_time);

return service\_time;

}

: It retrieves the customer waiting in the queue and return the customer's service time. Once we take out the customer from the queue, the service time decrease by one immediately. So set service\_time as service\_time-1 and return renewed service\_time.

5) print\_stat

void print\_stat()

{

printf("Number of customers served = %d\n", served\_customers);

printf("Total wait time =% d minutes\n", waited\_time);

printf("Average wait time per person = %f minutes\n",

(double)waited\_time / served\_customers);

printf("Number of customers still waiting = %d\n", customers - served\_customers);

}

: It prints statistics about bank waiting system.

6)main

: Inorder to increase the number of bank employees from one to two, we can simply call function remove\_customer() twice per clock. To know when each staff can receive new guests, we need a variable `service\_time`. So declare two variables, service\_time1 and service\_time2. This variable shows whether the staffs is currently in service or not at each clock. When service\_time remaining is zero, which means previous customer’s service time is ended, staffs receive new quest.

void main()

{

int service\_time1 = 0; // staff1’s service time remaing

int service\_time2 = 0; // staff2’s service time remaing

// require for users to input simulation value

printf("input the simulation duration: ");

scanf\_s("%d", &duration);

printf("input the average number of customers arriving in one time unit (0.0 ~ 1.0): ");

scanf\_s("%lf", &arrival\_prob);

printf("input the max serve time: ");

scanf\_s("%d", &max\_serv\_time);

clock = 0; //initialize clock

while (clock < duration) { //during duration.

clock++; //time increases by one every loop

printf("Current time=%d\n", clock);

if (is\_customer\_arrived()) {

insert\_customer(clock);

}

// Check if the customer who is receiving the service is finished.

if (service\_time1 > 0) // if staff1 is giving service

service\_time1--; //service that customer.

else //if previous customer ‘s service time is over,

{// staff1 receives new customer

printf("%s", "staff1: ");

// So, staff1 take out a customer from the queue and start the service. service\_time1 = remove\_customer();

}

if (service\_time2 > 0) //if staff2 is giving service

service\_time2--; //service that customer.

i

else //if previous customer ‘s service time is over,

{ // staff2 receives new customer

printf("%s", "staff2: ");

// So, staff2 take out a customer from the queue and start the service.

service\_time2 = remove\_customer();

}

printf("\n\n");

}

print\_stat(); //print statistics.

}