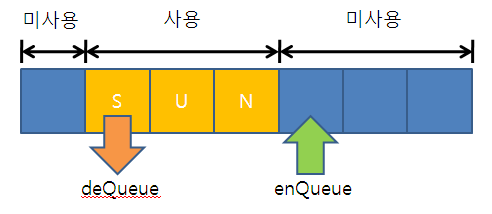
**———————————————————————————————————————————————————**

[1] 기초 문제

———————————————————————————————————————————————————

1. **다음 그림에서 찾을 수 있는 큐의 용어들에 어떤 것이 있는지 나열하시오.**



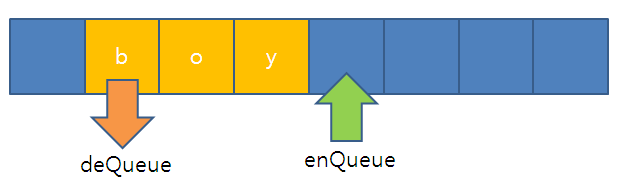
1. **큐 용어와 관련된 문제다. 빈 칸에 들어가는 단어를 맞춰보도록 한다.**
   1. ① FIFO ② 먼저 ③ 먼저 ④ 선입선출
   2. ① rear ② 추가될 ③ 뒤쪽
   3. ① dequeue ② 꺼내는 ③ 앞쪽 ④ pop
2. **큐를 사용하고 있는 곳을 세 가지씩만 나열해 보자.**
   1. 컨베이어 벨트, 자동 판매기, 번호표, 일방통행, 리프트
   2. 스풀러, 입력 장치, 매크로 명령, 복기, 콜 센터

|  |
| --- |
| 해답  정답으로 간주할 수 있는 것들이 너무 많아서 이번 장에서 설명한 것만 실었습니다. |

1. **최대 7개의 문자를 저장할 수 있는 큐가 있다. 아래와 같은 입출력이 발생한다면, 꺼낸 데이터를 순서대로 보여주고 최종적인 큐의 내부를 그려라.**

출력 : cleanroom

큐는 (크기-1)개를 저장할 수 있으므로 크기는 8이 됩니다. 실수하기 쉬운 부분입니다.



1. **큐의 반대말은 스택이 아니다. 스택과 큐는 상호 보완적인 관계로, ‘견우와 직녀’ 또는 ‘로미오와 줄리엣’이라고 할 수 있다. 빈 칸에 들어가는 내용을 단답형으로 채워보자.**

|  |  |  |
| --- | --- | --- |
| 구분 | 스택 | 큐 |
| 방식 | LIFO(후입선출) | FIFO(선입선출) |
| 용어 | push, pop | enQueue, deQueue |
| 포인터 | top | front, rear |
| 원리 | top이 오르락내리락 | front와 rear가 빙글빙글 |
| 주변 사례 | 프링글스, CD 케이크 | 은행 번호표, 스키장 리프트 |
| 컴퓨터 사례 | 함수 호출, 미로 찾기 | 프린터 스풀러, 파일 송수신 |

|  |
| --- |
| 해답  표에 대한 완전한 설명은 앞에서 배웠던 스택 2장의 심화 학습에 있습니다. 설명이 너무 길어 기본적인 답안만 제공합니다. |

**———————————————————————————————————————————————————**

[2] 기본 실습 문제

———————————————————————————————————————————————————

1. **본문에서 구현했던, 큐를 구성하는 핵심 함수들이다. 직접 구현해 보자.**
   1. void enQueue(QUEUE\* pq, char data)

{

pq->array[pq->rear] = data;

pq->rear = (pq->rear+1) % QUEUE\_SIZE;

}

* 1. char deQueue(QUEUE\* pq)

{

int save = pq->front;

pq->front = (pq->front+1) % QUEUE\_SIZE;

return pq->array[save];

}

* 1. int isEmpty(QUEUE\* pq)

{

return pq->front == pq->rear;

}

1. **본문에서 배우지 않았거나 구현을 했어도 조금은 어려운 함수들이다. 구현하고 검증까지 하도록 하자.**
   1. int count(QUEUE\* pq)

{

int c = pq->rear - pq->front;

return (c >= 0) ? c : QUEUE\_SIZE+c;

}

* 1. 성능을 고려하지 않는다면 아래처럼 구현하는 것이 좋습니다.

void arrange(QUEUE\* pq)

{

char temp[QUEUE\_SIZE];

int i, j = 0;

for(i = pq->front; i != pq->rear; i=(i+1)%QUEUE\_SIZE)

temp[j++] = pq->array[i];

for(i = 0; i < j; i++)

pq->array[i] = temp[i];

pq->front = 0;

pq->rear = j;

}

이 코드는 정상과 뒤집힌 경우를 따로 처리합니다. 뒤집힌 경우에 평균 25% 정도의 성능 향상이 기대됩니다. 정상인 경우에는 임시 배열을 사용하지 않아도 되는 장점도 있습니다. 추천하진 않습니다.

void arrange(QUEUE\* pq)

{

int i, count = pq->rear - pq->front;

if(pq->front <= pq->rear)

{

for(i = 0; i < count; i++)

pq->array[i] = pq->array[i+pq->front];

}

else

{

char temp[QUEUE\_SIZE];

int j = QUEUE\_SIZE - pq->front;

// 0부터 rear까지, 큐의 뒷부분을 임시 배열에 복사

for(i = 0; i < pq->rear; i++)

temp[i] = pq->array[i];

// front부터 마지막까지 앞쪽으로 이동

for(i = pq->front; i < QUEUE\_SIZE; i++)

pq->array[i-pq->front] = pq->array[i];

// 임시 배열을 큐로 복사

for(i = 0; i < pq->rear; i++)

pq->array[i+j] = temp[i];

// 요소 개수

count += QUEUE\_SIZE;

}

pq->front = 0;

pq->rear = count;

}

* 1. 최근 데이터를 검색하기 때문에 rear에서 front 방향으로 반복합니다. 삭제 이후의 결과는 rear쪽으로 한 칸씩 당겨진 형태가 됩니다. i를 이동시킬 땐 QUEUE\_SIZE-1, 복사할 땐 QUEUE\_SIZE+1을 사용한다는 것만 주의하면 되겠습니다. flag가 0일 땐 찾기, 1일 땐 복사를 수행합니다.

int deleteData(QUEUE\* pq, char del)

{

int i = pq->rear,

flag = 0,

end = (pq->front+QUEUE\_SIZE-1)%QUEUE\_SIZE;

while(1)

{

i = (i+QUEUE\_SIZE-1)%QUEUE\_SIZE;

if(i == end)

break;

if(flag == 1) pq->array[(i+QUEUE\_SIZE+1)%QUEUE\_SIZE] = pq->array[i];

else if(pq->array[i] == del) flag = 1;

}

if(flag == 1)

pq->front = (pq->front+QUEUE\_SIZE+1)%QUEUE\_SIZE;

return flag;

}

**———————————————————————————————————————————————————**

[3] 응용 실습 문제

———————————————————————————————————————————————————

1. 큐를 적용한 함수입니다. 함정이 있다고 말한 것처럼, 절반만 큐에 넣어야 검증할 수 있습니다. 전체를 다 넣으면 문자열을 거꾸로 반복해야 합니다. 회문이니까, 반만 반복하는 게 좋은 코드 같습니다.

int palindrome1(QUEUE\* pq, const char\* s)

{

size\_t i, len = strlen(s);

for(i = 0; i < len/2; i++)

enQueue(pq, s[i]);

for(i = len-1; isEmpty(pq) == 0; i--)

{

if(s[i] != deQueue(pq))

return 0;

}

return 1;

}

큐를 사용하지 않는, 그래서 훨씬 더 좋은 코드입니다.

int palindrome2(const char\* s)

{

int i = 0, j = (int) strlen(s)-1;

while(i < j)

{

if(s[i++] != s[j--])

return 0;

}

return 1;

}

1. 본문과 똑같이 다중 파일 프로젝트로 처리했습니다. 크기를 1부터 시작하지만, 하나는 사용하지 못하기 때문에 출력은 2개부터 시작됩니다. 달라진 부분은 추가하는 enQueue 함수 정도입니다. QUEUE\_SIZE 상수 대신 size 멤버를 추가했고, % 연산자를 사용할 때 size 멤버로 대체했습니다.

**// Queue.h**

#pragma once

typedef struct

{

char\* array;

int size;

int front, rear;

}

QUEUE;

void initQueue(QUEUE\* pq);

void enQueue(QUEUE\* pq, char data);

char deQueue(QUEUE\* pq);

int isEmpty(QUEUE\* pq);

int isFull(QUEUE\* pq);

void printQueueData(QUEUE\* pq);

**// Queue.c**

#include <stdio.h>

#include <stdlib.h>

#include "Queue.h"

void initQueue(QUEUE\* pq)

{

pq->front = pq->rear = 0;

pq->size = 1;

pq->array = malloc(sizeof(char)\*pq->size);

}

void enQueue(QUEUE\* pq, char data)

{

if(isFull(pq) == 1)

{

int i, j;

char\* big = malloc(sizeof(char)\*pq->size\*2);

for(i = pq->front, j = 0; i != pq->rear; i = (i+1)%pq->size)

big[j++] = pq->array[i];

free(pq->array);

pq->array = big;

pq->size \*= 2;

pq->front = 0;

pq->rear = j;

}

pq->array[pq->rear] = data;

pq->rear = (pq->rear+1) % pq->size;

}

char deQueue(QUEUE\* pq)

{

int save = pq->front;

pq->front = (pq->front+1) % pq->size;

return pq->array[save];

}

int isEmpty(QUEUE\* pq)

{

return pq->front == pq->rear;

}

int isFull(QUEUE\* pq)

{

return (pq->rear+1)%pq->size == pq->front;

}

void printQueueData(QUEUE\* pq)

{

int i;

printf("크기: %3d -- ", pq->size);

for(i = pq->front; i != pq->rear; i=(i+1)%pq->size)

printf("%c", pq->array[i]);

printf("\n");

}

**// main.c**

#pragma warning(disable:4996)

#include <stdio.h>

#include "Queue.h"

void main(void)

{

QUEUE q;

char s[] = "merry christmas!";

int i;

initQueue(&q);

for(i = 0; s[i]; i++)

{

enQueue(&q, s[i]);

printQueueData(&q);

if(i%3 == 2)

{

printf("\*\*\* delete \*\*\*\n");

deQueue(&q);

}

}

}

**// 출력 결과**

크기: 2 -- m

크기: 4 -- me

크기: 4 -- mer

\*\*\* delete \*\*\*

크기: 4 -- err

크기: 8 -- erry

크기: 8 -- erry

\*\*\* delete \*\*\*

크기: 8 -- rry c

크기: 8 -- rry ch

크기: 8 -- rry chr

\*\*\* delete \*\*\*

크기: 8 -- ry chri

크기: 16 -- ry chris

크기: 16 -- ry christ

\*\*\* delete \*\*\*

크기: 16 -- y christm

크기: 16 -- y christma

크기: 16 -- y christmas

\*\*\* delete \*\*\*

크기: 16 -- christmas!

**// 코드 개선**

enQueue 함수를 처리하는 더 좋은 방법이 있어 소개합니다. 큐가 꽉 찰 때는 front가 rear보다 작을 때와 클 때의 두 가지가 있습니다. 그런데, front가 작은 경우는 딱 한 가지로, front가 0번째, rear가 size-1번째에 위치한 경우입니다. 반복문을 쉽게 구성하기 위해 항상 rear가 front보다 크게 만듭니다. 이렇게 하면 임시 변수들이 필요 없고, 코드가 간결해집니다.

void enQueue(QUEUE\* pq, char data)

{

if(isFull(pq) == 1)

{

int i;

char\* big = malloc(sizeof(char)\*pq->size\*2);

// rear는 언제나 front보다 큽니다

if(pq->front > 0)

pq->rear += pq->size;

for(i = pq->front; i < pq->rear; i++)

big[i] = pq->array[i%pq->size];

free(pq->array);

pq->array = big;

pq->front = pq->front;

pq->rear = pq->front + pq->size-1;

pq->size \*= 2;

}

pq->array[pq->rear] = data;

pq->rear = (pq->rear+1) % pq->size;

}