Lecture 07

스택 및 큐

리뷰

- **`struct`**
  - 하난 또는 여러 개의 데이터형을 포함할 수 있음
  - **`struct`**의 **`struct`**도 있을 수 있음
  - **`struct`**의 크기는 모두 멤버가 들어갈 수 있는 만큼(패딩될 수 있음) 메모리는 할당됨
  - **`struct`**의 이름이 생략될 수 있음
- **`union`**
  - 한 기억장소 내에서 서로 다른 데이터형을 처리할 수 있음
  - **`union`**의 크기는 제일 큰 멤버가 들어갈 수 있는 만큼 메모리는 할당됨
  - **`union`**의 이름이 생략될 수 있음
- 비트 필드(bit fields)
  - 구조체 멤버의 크기는 비트 단위로 설정할 수 있음
  - 메모리 할당은 컴파일러와 하드웨어에 따름

리뷰

- 다음과 같은 구조체를 참고하셔서 질문을 답하시오.

```
struct A {
    short s;
    union {
        int k;
        char c;
    } u;
    unsigned char flag_s : 1;
    unsigned char flag_u : 2;
    unsigned int d;
};
```

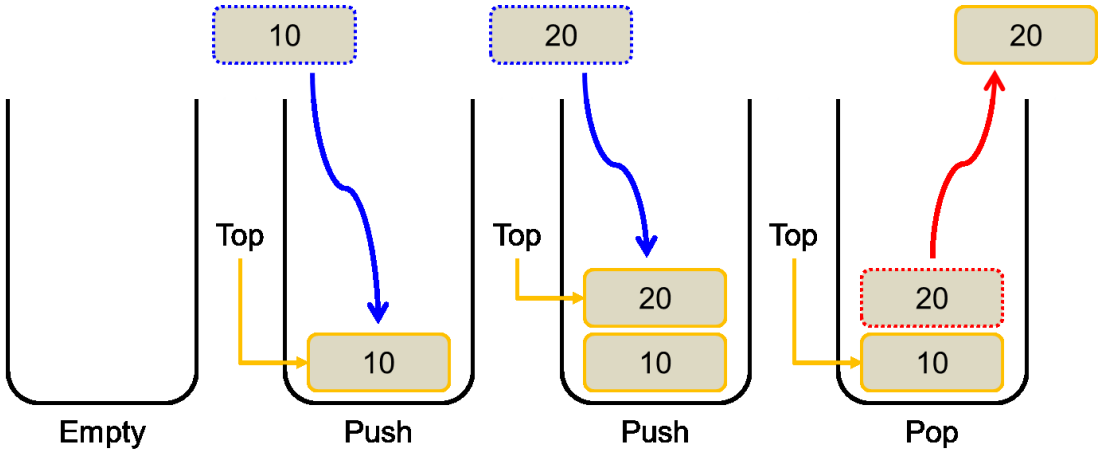
struct A의 크기를 구하시고 **struct** A의 크기를 최소화하기 위해 멤버의 순서를 다시 정렬하시오.

리뷰

- 연결리스트 및 트리
  - 동적 메모리 할당으로 데이터를 추가하거나 삭제할 수 있는 자료구조
  - 일반적으로 C에서 **`struct`**로 구현됨
  - 동적 메모리 할당을 위해서 `malloc()`, `free()` 등과 같은 함수를 사용함
  - 배열과 달리 연결리스트나 트리의 노드를 인덱스로 알아낼 수 없으며, 종주(traversal)를 통해 해야 함

스택(stack)

- 데이터를 후입선출(Last In First Out)하는 자료구조임
  - 가장 최근에 들어온 데이터가 가장 먼저 나감
  - 스택에 실행할 수 있는 연산
    - Push : 스택에 데이터를 추가한다는 의미
    - Pop : 스택에서 데이터를 삭제하고, 그 데이터를 리턴한다는 의미
    - Peek : 스택에서 데이터를 삭제하지 않고, 들여다보기만 한다는 의미



스택(stack)

- 배열 기반 스택 구현

```
#define SIZE 100
typedef struct {
    int data[SIZE];
    int top;
} stack;
```

  - 스택 만들기

```
stack *createStack() {
    stack *S = (stack*)malloc(sizeof(stack));
    S->top = -1; // 빈 스택이라는 의미
    return S;
}
```

스택(stack)

- 배열 기반 스택 구현
  - 스택이 비워 있는 지 검토

```
int is_empty(stack *S) {
    return S->top == -1;
}
```

  - 스택이 차 있는 지 검토

```
int is_full(stack *S) {
    return S->top == SIZE-1;
}
```

스택(stack)

- 배열 기반 스택 구현
  - 스택에 데이터 추가(push)

```
void push(stack *S, int data) {
    if (is_full(S)) printf("Overflow\n");
    else {
        S->top++;
        S->data[S->top] = data;
    }
}
```

스택(stack)

- 배열 기반 스택 구현
  - 스택에서 데이터 삭제(pop)

```
int pop(stack *S) {
    if (is_empty(S)) {
        printf("Empty\n");
        return -1;
    }
    else {
        int tmp = S->data[S->top];
        S->top--;
        return tmp;
    }
}
```

스택(stack)

- 배열 기반 스택 구현
  - 스택에서 데이터 들여보기(peek)

```
int peek(stack *S) {
    if (is_empty(S)) {
        printf("Empty\n");
        return -1;
    }
    else {
        return S->data[S->top];
    }
}
```

스택(stack)

- 배열 기반 스택 구현
  - 스택을 처음부터 끝까지 출력

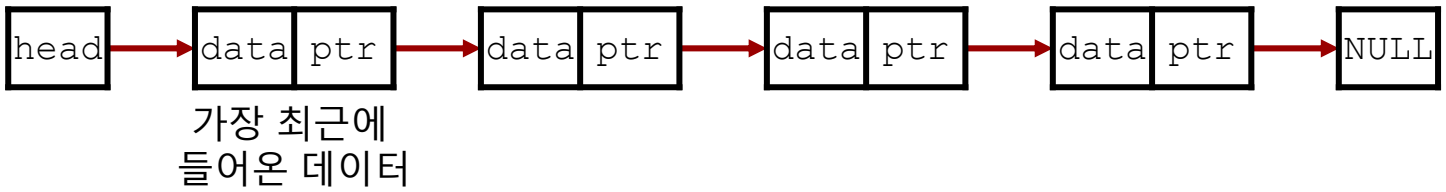
```
void printStack(stack *S) {
    int k;
    for (k=0; k<=S->top; k++)
        printf("%d\n", S->data[k]);
    printf("\n");
}
```

스택(stack)

- 연결리스트 기반 스택 구현

```
typedef struct Node {
    int data;
    struct Node *next;
} node;

node *stack = NULL; // 빈 스택 만들기
```



스택(stack)

- 연결리스트 기반 스택 구현
  - 스택에 데이터 추가(push)

```
node *push(node *S, int data) {
    node *newS = insertHead(S, data))
    return newS;
}
```

  - 스택에서 데이터 삭제(pop)

```
node *pop(node *S) {
    node *newS = deleteFront(S);
    return newS;
}
```

스택(stack)

- 연결리스트 기반 스택 구현
  - 스택에서 데이터 들여보기(peek)

```
int peek(node *S) {
    if (S == NULL) {
        printf("Empty\n");
        return -1;
    }
    else {
        return S->data;
    }
}
```

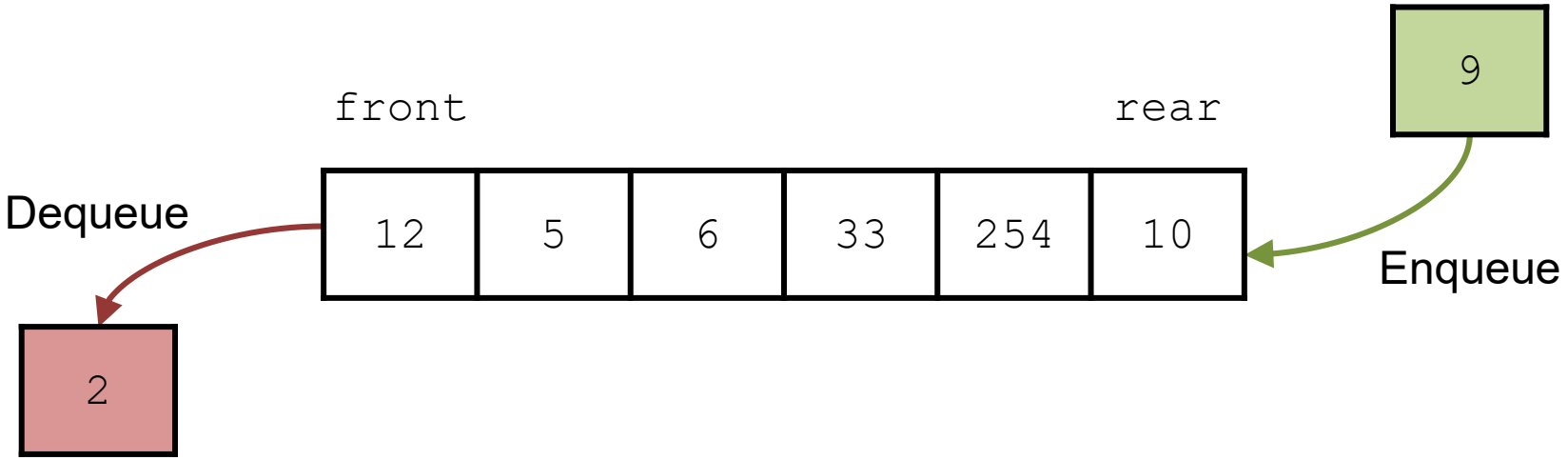
스택(stack)

- 연결리스트 기반 스택 구현
  - 스택을 처음부터 끝까지 출력

```
void printStack(node *S) {
    node *tmp = S;
    while (tmp != NULL) {
        printf("%d\n", tmp->data);
        tmp = tmp->next;
    }
    printf("\n");
}
```

큐(queue)

- 데이터를 선입선출(First In First Out)하는 자료구조임
  - 가장 먼저 들어온 데이터가 가장 먼저 나감
  - 큐에 실행할 수 있는 연산
    - Enqueue : 큐 끝에 데이터를 추가한다는 의미
    - Dequeue : 큐 앞에서 데이터를 삭제한다는 의미



큐(queue)

- 배열 기반 큐 구현

```
#define SIZE 100
typedef struct {
    int data[SIZE];
    int front;
    int rear;
} queue;
```

  - 큐 만들기

```
queue *createQueue() {
    queue *Q = (queue*)malloc(sizeof(queue));
    Q->front = -1;
    Q->rear = -1;
    return Q;
}
```

큐(queue)

- 배열 기반 큐 구현
  - 큐 비워 있는 지 검토

```
int is_empty(queue *Q) {
    return (Q->front == -1) || (Q->front == Q->rear);
}
```

  - 큐 차 있는 지 검토

```
int is_full(queue *Q) {
    return (Q->rear == SIZE-1);
}
```

큐(queue)

- 배열 기반 큐 구현
  - 큐 끝에 데이터 추가(enqueue)

```
void enqueue(queue *Q, int data) {
    if (is_full(Q)) {
        printf("Queue is full\n");
        return;
    }
    if (is_empty(Q)) Q->front = 0;
    Q->rear++;
    Q->data[Q->rear] = data;
}
```

큐(queue)

- 배열 기반 큐 구현
  - 큐 앞에서 데이터 삭제(dequeue)

```
int dequeue(queue *Q) {
    if (is_empty(Q)) {
        printf("Queue is empty\n");
        return -1;
    }
    int data = Q->data[Q->front];
    if (Q->front == Q->rear) { // dequeue 후 비워 된 경우
        Q->front = -1;
        Q->rear = -1;
    } else Q->front++;
    return data;
}
```

큐(queue)

- 배열 기반 큐 구현
  - 큐에서 저장된 데이터 출력

```
void printQ(queue *Q) {
    if (is_empty(Q)) {
        printf("Queue is empty\n");
        return;
    }
    int k;
    for (k=Q->front; k<=Q->rear; k++)
        printf("%d ", Q->data[k]);
    printf("\d");
}
```

큐(queue)

- 연결리스트 기반 큐 구현

```
typedef struct Node {
    int data;
    struct Node *next;
} node;

typedef struct {
    node *front;
    node *rear;
} queue;
```

```
node *createNode(int data) {
    node *n = (node*)malloc(sizeof(node));
    if (n == NULL) return NULL;
    n>data = data;
    n>next = NULL;
    return n;
}

queue *createQueue() {
    queue *q = (queue*)malloc(sizeof(queue));
    q->front = q->rear = NULL;
    return q;
}
```

큐(queue)

- 연결리스트 기반 큐 구현
  - 큐 비워 있는 지 검토

```
int is_empty(queue *Q) {
    return Q->front == NULL;
}
```

  - 큐 차 있는 지 검토
    - 연결리스트 기반으로 구현하기 때문에 큐에 데이터를 추가하기 위한 메모리 할당이 실패할 경우 큐가 차 있는 것임

```
void enqueue(queue *Q, int data) {
    ...
    node *n = createNode(data);
    if (n == NULL) // 메모리 할당 실패, 큐 차 있음
    ...
}
```

큐(queue)

- 연결리스트 기반 큐 구현
  - 큐 끝에 데이터 추가(enqueue)

```
void enqueue(queue *Q, int data) {
    node *n = createNode(data);
    if (!n) {
        printf("Queue is full\n");
        return;
    }
    if (Q->rear == NULL) {
        Q->front = Q->rear = n;
        return;
    }
    Q->rear->next = n;
    Q->rear = n;
}
```

큐(queue)

- 연결리스트 기반 큐 구현
  - 큐 앞에서 데이터 삭제(dequeue)

```
int dequeue(queue *Q) {
    if (is_empty(Q)) {
        printf("Queue is empty\n");
        return -1;
    }
    node *n = Q->front;
    Q->front = Q->front->next;
    if (Q->front == NULL) Q->rear = NULL;
    int data = n->data;
    free(n);
    return data;
}
```

큐(queue)

- 연결리스트 기반 큐 구현
  - 큐에서 저장된 데이터 출력

```
void printQ(queue *Q) {
    node *n = Q->front;
    while (n != NULL) {
        printf("%d ", n->data);
        n = n->next;
    }
    printf("\n");
}
```

계산기

- 스택과 큐를 이용하여 간단한 계산기를 구현할 수 있음
- Prefix(전위), infix(중위), postfix(후위) 표기법

Infix(중위)	Prefix(전위)	Postfix(후위)
A + B	+ A B	A B +
A * B - C	- * A B C	A B * C -
(A + B) * (C - D)	* + A B - C D	A B + C D - *

- 쓰기 및 해석하기 위해 infix를 사용하는 것 편하지만, 프로그래밍 언어로 구현하기에 postfix가 더 적합함

계산기

- 계산기 구현
  - Infix 표기법으로 입력을 받음
  - 입력을 postfix 표기법으로 변환함
  - Postfix 수식을 실행하여 결과를 리턴함
- 프로그램 구조

```
int main() {
    infix 입력 받기
    infix_to_postfix();
    postfix_eval();
    결과 출력
    return 0;
}
```

계산기

- Infix – Postfix 변환
  - 스택(연산자 저장) 및 큐(입출력 저장) 사용
  - Binary 연산자만 처리 가능
  - 알고리즘 흐름
    ① 입력 큐에서 dequeue하여 토큰(token)을 얻음
    ② 토큰은 피연산자가 일 경우 출력 큐에 enqueue함
    ③ 토큰은 연산자가 일 경우
      a. 스택의 탑에서 있는 연산자가 토큰보다 높거나 같은 우선순위를 갖을 경우, 스택에서 있는 연산자의 모두를 pop하여 출력 큐에 enqueue함
      b. 토큰을 스택에 push함
    ④ 입력 큐가 비워 있지 않을 경우, 첫 단계로 다시 실행함
    ⑤ 스택에서 남아 있는 연산자들을 pop하여 출력 큐에 enqueue함

계산기

- Infix – Postfix 변환
  - Infix 입력: A + B * C - D

토큰	출력 큐	스택
A	A	
+	A	+
B	A B	+
*	A B	+ *
C	A B C	+ *
-	A B C * +	-
D	A B C * + D	-
(끝)	A B C * + D -	

계산기

- Infix – Postfix 변환
  - Infix 입력: (A + B) * (C – D)

토큰	출력 큐	스택
(		(
A	A	(
+	A	(+
B	A B	(+
)	A B +	
*	A B +	*
(	A B +	* (
C	A B + C	* (
–	A B + C	* (–
D	A B + C D	* (–
)	A B + C D –	*
(끝)	A B + C D – *	

계산기

- Postfix 수식 실행
  - 알고리즘 흐름
    ① 출력 큐에서 dequeue하여 토큰을 얻음
    ② 토큰은 피연산자가 일 경우 스택에 push함
    ③ 토큰은 연산자가 일 경우
      a. 스택에서 2개의 피연산자를 pop함
      b. 수식을 실행하여 결과를 스택에 push함
    ④ 큐가 비워 있을 때까지 ①~③을 반복함
    ⑤ 스택에 남아 있는 것은 최종 결과임

계산기

- Postfix 수식 실행
  - Postfix 수식: `3 4 + 5 1 - *`

토큰	스택
3	3
4	3 4
+	7
5	7 5
1	7 5 1
-	7 4
*	28 (최종 결과)
(끝)	