1월 3일  
가장 먼저 들어간게 가장 먼저 나와야하는데, 현재 deq해야할 데이터를 뽑아서 확인을해야하는데 뽑으면 돌이킬수없다. 루프를 돌며 출력을 하는 이 경우에도? 배열을 직접 접근할 수는 없다. Front rear라는 변수도 전역으로 존재하지만 건들여서는 안된다. 오직 컨텐츠에서 호출가능한건 밑의 3개의 함수 뿐. 뽑을순 없으니 deq는 안되고 peek를 쓴다. Front를 이동시키지 않으면서.   
pos는 0이 들어가면 바로 나올 값. 1을넣으면 deq2번했을 떄 나온 값. front기준으로 몇칸을 가서 뽑아줄건지의 위치를 지정해주는 것. peek에서 랜더링, 100%가 되었는지 확인. 로직에서 100-%확인이 되었으면 deq. 랜더 위치에서는 무조건 랜더만. 10-0%가 되었다고 해서 deq를 랜더부에서 하면 안된다.

Bool Enqueue (Indata);  
bool Dequeue (&Outdata);  
Bool Peek(&Outdata, pos);

Getasynckeystate : 이 키가 눌렸는지, 우리가 원하는 키를 지정할 수 있음.   
0x8000 : 최상위 1비트가 지금 눌려져있다는 상태. 지금 눌렸냐  
0x0001 : 눌린적이 있었냐. 근데 msdn에서는 쓰지 않길 권장함. 왜? 내 어플리케이션에서만 검사하는게 아니기 때문에. 그래서 정확하지 않다는 것.

방법은 2가지.

1. If(Getasynckeystate(VK\_RETURN))  
   else
2. If(Getasynckeystate(VK\_RETURN) & 0X8000) : 마스킹 해서 눌렸는지 확인

배열에 생성 요청 시간이 들어있으면 루프를 돌면서 퍼센테이지가 증가함. 올라감으로 인해서 내 보관 데이터에 변화를 줄까? 그렇지 않고 처음 저장한 시간만 놔두고 현재시간에서 빼서 체크하면 된다. 반복적으로 += 를 통해 누적치를 구하게 되면 조금씩 오차가 나는게 누적이 된다. 그래서 매번 새로 구해야한다. 그래서 퍼센테이지를 ++로 올리는게 아니라 매번 계산해야한다.

해쉬테이블

해쉬 : 입력이 어떻게 되던 자기만의 공식에 의해 자기의 규칙대로 통일된 결과가 나온다. 몇바이트를 넣든지. 똑 같은 동일한 규칙에 맞춰진, 틀로 찍은 것처럼. 데이터는 달라지겠지만.   
0~99까지의 범위로 나오는 해쉬를 만들고 싶다면 어떤값이든 %100을하면 된다. 근데 1을 넣든 101을 넣든 나머지가 1이 나오면 이건 충돌했다고 말한다. 출력하는 결과가 동일하기 때문에 해쉬는 충돌할 수 있다. 알고리즘에 대한 해쉬코드를 공개함. 이건 무슨말이냐, 파일다운받고 알고리즘으로 이 파일 전체를 해쉬시켜봐라. 일치한다면 파일이 맞는 것. 변조를 방지하기위해. 아니면 게임같은경우엔 자체적으로 정해진 알고리즘으로 내 모든파일을 해쉬해봄. 다 해쉬알고리즘에 넣음. 그리고 결과가 64바이트나 또 다른 바이트가 나옴. 서버에도 원본에대한 해쉬가 있을ㅇ거고. 해쉬값만 최종적으로 일치하면 같다고 믿음. 또 회원가입할떄 비밀번호. 해쉬코드가 1이 있다 치자. 이 1의 원본은? 모른다. 뭐를 나머지 한 값이 1이 나오는지 원본을 모르는거랑 같음. 암호화가 아니다. 이건 복호화가 불가능하다. 복호화가 될수도 없다. Input에 대한 길이가 정해져있지 않고, 길에 따라 output이 달라지지 않는다. 원본으로 복구하는걸 고려하지 않고 만든다. 복호화가 안된다. 그럼 어떻게 쓰나? 우리가 입력한 데이터가 해쉬코드만 입력되어있으면 로그인할 때 같은 해쉬가 나오면 일치한다고 봄. 해쉬충돌은 이럴경우 0에 가까움.

모든 해쉬의 결과를 저장하는 것이 레인보우 테이블 DB.

이 기능을 해쉬테이블로 쓴다고 하면 이렇게 복잡하게 쓰지는 않음. 자료구조라고 하긴 뭐하고 목적은 빠른 검색을 위해서 있는 것. 검색을 한다는건 이걸 찾아서 실질적인 뭔가를 찾기위한 것. KEY VALUE 형태의 무언가를 쓸거임. Id를 넣었더니 객체 포인터가 나온다던가. 그래서 테이블이니까 배열로 만들 것. 한 1000칸 있다면 우리만의 해쉬함수가 있고, 파라미터로 char\* key가 들어감. 그럼 배열의 인덱스가 튀어나옴. 거기로 가면 데이터가 있음. 원래라면 배열이면 순회해야하는데, 우리가 만든 해쉬를 넣었더니 인덱스가 나오는 것. 엄청난 속도. 이건 o(1) 이다. 문자열이 있을 떄 abc가 있으면 하나의 변수에 ‘a’ ‘b’ ‘c’ 하나씩 아스키코드를 더하는 것. 이 규칙은 우리가 만드는것. 이 문자열을 가지고 인덱스가 나오게 하고싶은거니까. 그리고 다 더한 수를 %1000 해서 나온 인덱스값에다가 abc 유저의 값을 넣는 것. 빠른 검색이 필요하다면 해쉬테이블이 가장 좋은 방법. 단점은, 전체 순회자체가 불필요하다. 적합하지 않다. 그러나 어떤 key를 대상으로 검색하는것은 탁월한 선택이다. 서버에서도 많이 쓴다. 검색이 빠르기 때문에. 서버에서 자주 쓰는걸로 레드블랙트리, 해쉬가 있다. 레드블랙트리는 정렬이 되어있어서 좋은데 삽입삭제가 느리다.

근데 배열이 1000개라면 들어갈공간이 1000개인데, 해쉬 충돌이 난다면? 완벽하게 충돌이 절대 안나는건 사실없다. 아웃풋 데이터의범위가 한정적이기 때문에. 충돌하면 어떻게 해결할것이냐? 충돌하면 몇칸 옆으로 가자. 또는 대중적으로 그 안에서 리스트로 붙이는 것. 한칸한칸의 요소마다 linked list가 들어가는 구조. 하다보니 운없게 한 칸에 데이터가 몰렸다? 그럼 해쉬테이블로서의 가치가 없다. 그럼 거기서는 또 next를 보면서 순회해야하기 때문에. 최대한 충돌을 적게 만드는게 해쉬테이블의 중요한 요소. 충돌이 적으려면 배열 사이즈가 커져야할거고. 그래서 보통 소수로 간다. 소수로 가지고 나머지 연산을 해야 중복되는 패턴이 거의 없기 때문.

Map의 내부 알고리즘은 레드블랙트리