CHAP 14

해싱(hashing)

탐색(retrieval, searching)이란?

- 탐색 방법의 기준 : 키의 크기 비교검색 vs. 키를 해석한 위치 값 검색
  - 키값 크기 비교검색 : 이진검색, 블록(보간)검색, 피보나치 검색, 순차검색 등
  - 키값 해석한 위치값 검색 : 해싱

- 탐색방법 성능
  - 키값 크기 비교 4가지 : $O(\log_2 n)$, $O(\sqrt{n})$, $O(n^{1.5})$, $O(n)$
  - 해싱 : 평균 O(1), 최악 O(n)

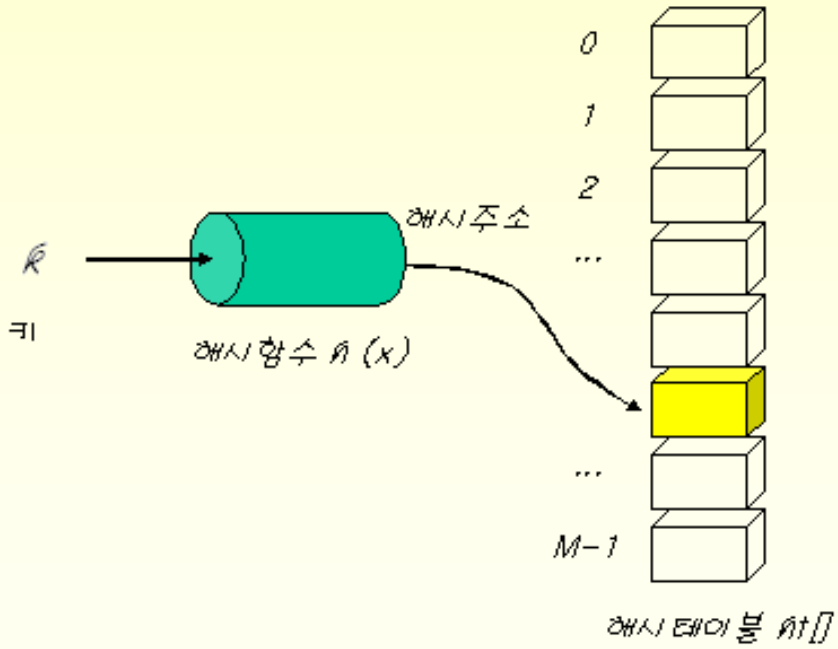
해싱(hashing)이란?

- **해싱**은 키 값을 연산한 값을 기준으로 저장 ➔ 테이블 주소에서 직접 탐색
  - 키 값 연산 결과를 이용하여 직접 접근하는 자료구조 : **해시 테이블**(hash table)
  - 해시테이블을 이용한 탐색기법 : **해싱**(hashing) **탐색**
- **해싱의 예 : 운영체제가 주로 사용(컴파일러가 항상 구성하는 symbol table)**

해싱의 구조

- **해시 함수**(hash function)에서 키값을 입력받아 **해시 주소**(hash address) 생성 ➔ **해시 테이블** (주로 배열로 구성)의 주소로 사용
- 예를 들어, 영어사전에서는 단어 "baby"가 탐색키가 되고, 단어 "baby"를 해싱 함수에서 계산한 값(예 : 123)으로 변환 ➔ 배열 table[123] 에 단어를 저장

  검색 과정 : "baby"를 입력 ➔ 해시함수에서 123 출력 ➔ 배열 table[123]을 직접 접근



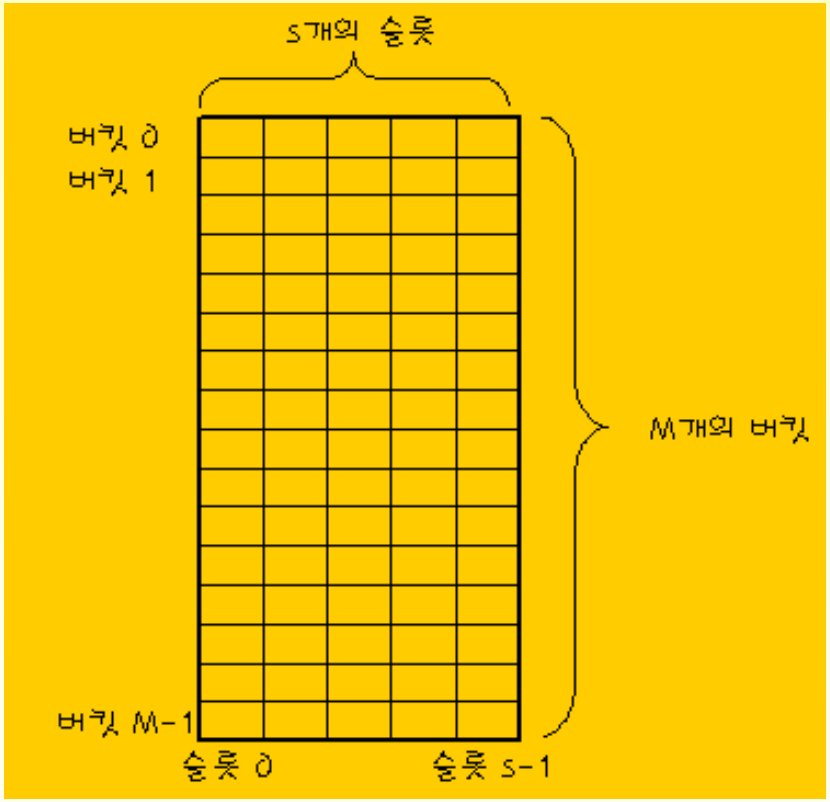
해시 테이블의 구조

- 해시테이블 table[ ]는 M개 **버켓**(bucket)으로 이루어지는 테이블 : table[0] ~ table[M-1]
- 버켓은 하나 이상의 데이터를 저장할 수 있음 : s개의 **슬롯**(slot) 가능

- 장점 : O(1) 시간에 위치 검색 가능
- 단점 : **충돌**(collision) 발생 가능

- 충돌 : 서로 다른 탐색키 k1과 k2의 해싱 결과 h(k1)와 h(k2)가 동일한 버켓주소를 발생
  ➔ 하나의 버켓이 여러 개의 슬롯을 가진다면 같은 버켓에 저장하면 해결

- 이런 상황이 자주 발생된다면?
  충돌 횟수가 버켓의 슬롯 개수보다 많이 발생하게 되면 버켓에 더이상 항목을 저장 불가한 **오버플로우(overflow)** 현상이 발생
  ➔ 오버플로우 해결방법 필요
  ➔ 이를 위한 추가시간이 소모

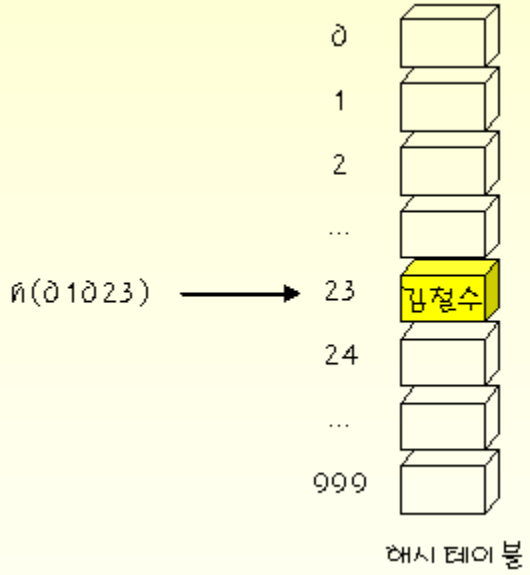


이상적(Ideal) 해싱

(예) 학생들에 대한 정보를 해싱으로 저장하고 탐색한다면…

- 탐색키로 학번(예 : 01023) 입력 : 앞의 2자리는 학과를 나타내고, 뒤의 3자리는 학생 번호로 사용한다면 ➔ 뒤의 3자리만으로 해시함수 결과로 사용(예 : 023)
- 거의 불가능하지만, 학번이 모두 고르게 버켓에 저장된다면 이상적 구조

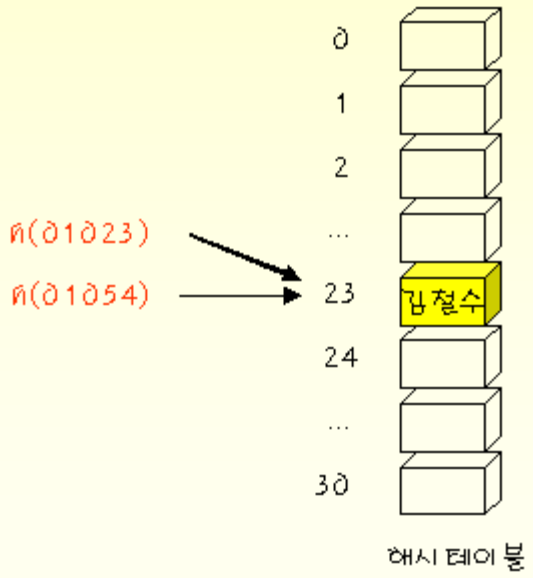
  - 학번이 01023 ➔ 학생정보는 해시테이블 table[23]에 저장 ➔ 검색도 같은 과정




```
저장 함수 add(key, value)
{
    index ← hash_function(key);
    ht[index] = value;
}
검색 함수 search(key)
{
    index ←hash_function(key);
    return ht[index];
}
```

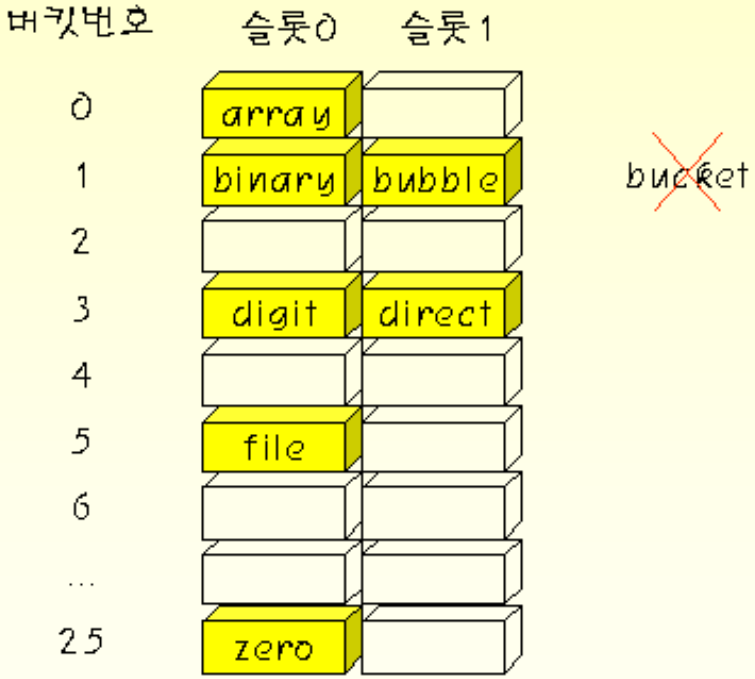
실제의 해싱

- 실제로 해시 테이블의 크기(버켓의 개수)가 제한 ➔ 탐색 키마다 해시테이블을 준비해둘 순 없음
- 좀 더 지능적인 해싱함수 필요 ➔ 예 : 해시함수 결과 = 키값 *mod* M
- 서로 다른 키값인데 같은 주소가 설정된다면?
  : 충돌 발생 ➔ 버켓 여유(여러 개 슬롯)가 있으면 저장 ➔ 반복 ➔ 오버플로우 발생



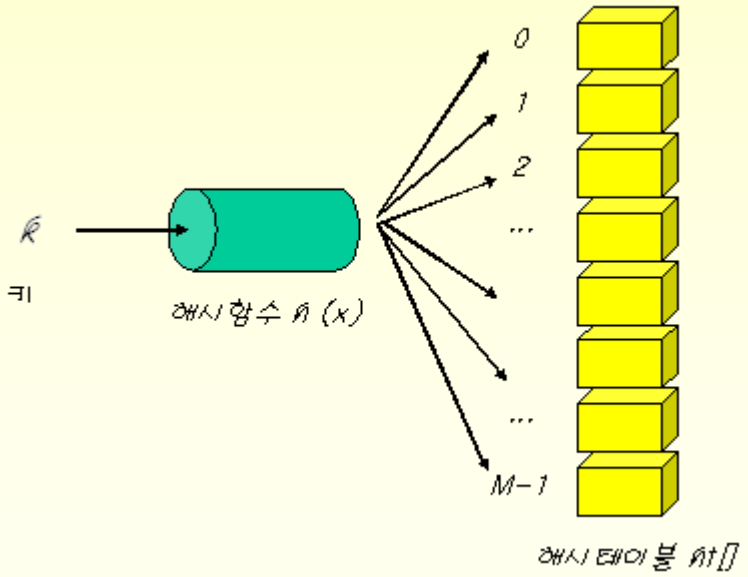
(두 번째 예제) 탐색키로 단어가 입력된다면...

- 해시함수는 키의 첫 번째 문자를 숫자로 바꾸는 함수를 적용
  h("array") ➔ 1 h("binary") ➔ 2 h("bubble") ➔ 2 h("digit") ➔ 4 ...
- 버켓마다 2개의 슬롯이 확보된 총 26개 버켓의 해싱 기법에서.
  입력이 (array, binary, bubble, file, digit, direct, zero, bucket)라면?



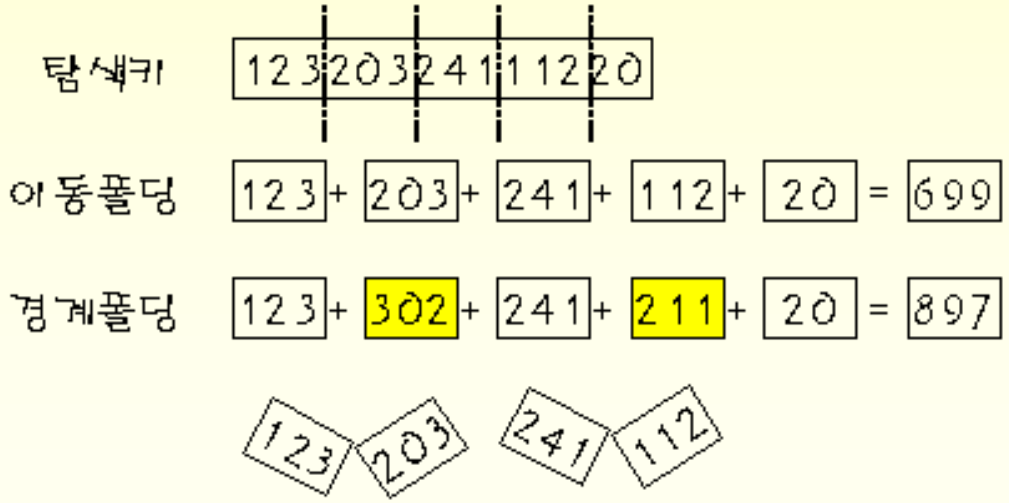
해싱 함수(hashing function)

- 좋은 해싱 함수의 조건
  - 서로 다른 키값을 해싱했을 때, 충돌이 최소화되어야 한다.
    (즉 해시주소가 해시테이블 내에서 고르게 분포되어야 한다)
  - 빠른 계산을 위해서 해싱함수가 단순해야 한다.



해시함수 유형

- **제산 함수**
  - h(k)=k mod M
  - 해시 테이블의 크기와 같은 M는 **소수**(prime number)가 절대적으로 유리...
- **폴딩 함수**
  - hash_index=(short)(key ^ (key>>16))
  - **이동 폴딩**(shift folding)과 **경계 폴딩**(boundary folding)



해시함수 유형(cont.)

- **중간제곱 함수**
  - 중간 제곱 함수는 탐색키를 제곱한 다음, 중간의 몇 비트를 취해서 해시 주소를 생성한다.
- **비트추출 함수**
  - 탐색키를 이진수로 간주하여 임의의 위치의 k개의 비트를 해시 주소로 사용하는 것이다.
- **숫자 분석 방법**
  - 키의 각각의 위치에 있는 숫자 중에서 편중되지 않는 수들을 해시 테이블의 크기에 적합한 만큼 조합하여 해시 주소로 사용

해싱의 충돌 해결책

- **아무리 좋은 해시 함수를 사용하더라도 충돌과 오버플로우는 발생**
  - Bucket 개수를 무한정 늘리면 해결되지만, 메모리 낭비가 매우 심해짐

- **충돌해결책**
  - **Open 해결책**
    - 선형 조사법 : 충돌이 일어난 항목을 해시 테이블의 다른 위치에 저장하는 방식
    - 이중 해싱법 : 해싱 & 재해싱으로 분산력 증가시키는 방식
  - **Close 해결책**
    - 체이닝(chaining) : 해시테이블 마다 여러 개의 항목을 저장할 수 있도록 해시 테이블의 구조를 연결리스트 형식으로 운영

1) 선형 조사법(개방법)

- ht[k]에서 충돌이 발생했다면, 바로 다음 위치의 ht[k+1]가 비어 있는지를 검색
  - 비어있으면 저장, 비어있지 않다면 그 다음 위치인 ht[k+2]를 검색
  - 이런 식으로 비어있는 공간이 나올 때까지 계속 검색해서 저장하는 방법
  - 테이블 끝이라면 처음으로 circular 검색 ➔ 시작 위치로 오면 테이블 FULL 상태

- 조사되는 위치 : h(k) ➔ h(k)+1 ➔ h(k)+2 , …
  (예) h(k) = k mod 7 ➔ 충돌 ➔ 여유 없으면 h(k) = (k+1) mod 7

1단계 (8) : $h(8) = 8 \bmod 7 = 1$(저장)
2단계 (1) : $h(1) = 1 \bmod 7 = 1$(충돌발생)
$(h(1)+1) \bmod 7 = 2$(저장)
3단계 (9) : $h(9) = 9 \bmod 7 = 2$(충돌발생)
$(h(9)+1) \bmod 7 = 3$(저장)
4단계 (6) : $h(6) = 6 \bmod 7 = 6$(저장)
5단계 (13) : $h(13) = 13 \bmod 7 = 6$(충돌 발생)

	1단계	2단계	3단계	4단계	5단계
[0]					13
[1]	8	8	8	8	8
[2]		1	1	1	1
[3]			9	9	9
[4]					
[5]					
[6]				6	6

2) 선형 이차 조사법(개방법)

- **이차 조사법**(quadratic probing)은 선형 조사법과 유사하지만, 다음 조사할 위치를 다양하게 선정하기 위하여 더 복잡한 식에 의하여 결정

  (예) $h_2(k) = (h_1(k) + i*i) \bmod M$

  따라서 조사되는 위치는 다음과 같이 된다.

  h(k) ➔ h(k)+1 ➔ h(k)+4 ➔ h(k)+9 ➔ …

- 선형 조사법에서의 문제점 "집중과 결합"을 크게 완화할 수 있음

  but, 완전한 해결책은 될 수 없음

3) 이중 해싱법(개방법)

- **이중 해싱법**(double hashing) 또는 **재해싱**(rehashing)은 오버플로우가 발생하면 저장할 다음 위치를 결정할 때, 처음의 해시 함수와 다른 해시 함수를 적용하는 방법

- 예 : 해시테이블에서 첫 번째 해시함수 결과인 h(k) = k mod M 위치가 오버플로우이면, 다른 해시함수 step=5-(5 mod 5) 적용
  - 테스트 추가 입력 : 8, 1, 9, 6, 13

- 약간은 개선되지만, 처리과정이 복잡함

	1단계	2단계	3단계	4단계	5단계
[0]					
[1]	8	8	8	8	8
[2]			9	9	9
[3]					13
[4]					
[5]		1	1	1	1
[6]				6	6

1단계 (8) : ·h(8) = 8 mod 7 = 1(저장)
2단계 (1) : ·h(1) = 1 mod 7 = 1(충돌발생)
·(h(1)+h'(1)) mod 7 = (1+5-(1 mod 5)) mod 7 = 5(저장)
3단계 (9) : ·h(9) = 9 mod 7 = 2(저장)
4단계 (6) : ·h(6) = 6 mod 7 = 6(저장)
5단계 (13) :·h(13) = 13 mod 7 = 6(충돌 발생)
·(h(13)+h'(13)) mod 7 = (6+5-(13 mod 5)) mod 7= 1(충돌발생)
·(h(13)+2*h'(13)) mod 7 = (6+2*2) mod 7= 3(저장)

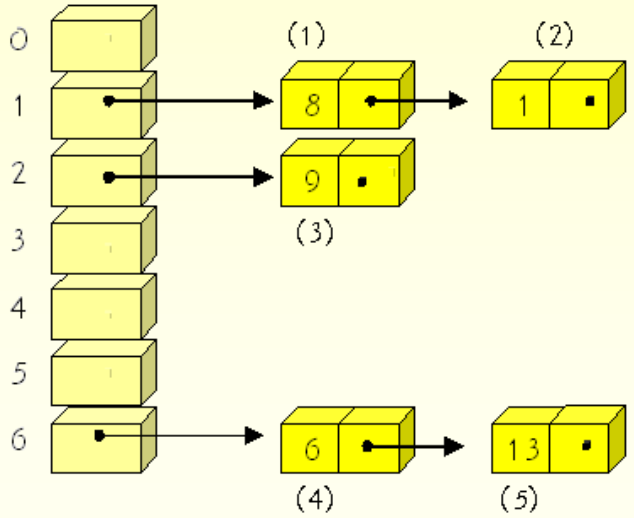
4) 체이닝 방식(폐쇄법)

- **개념 :** 오버플로우된 버켓에 저장할 추가 데이터를 연결 리스트로 해결
  - 각 버켓에 고정된 개수의 슬롯을 할당하는 것이 아님
  - 각 버켓에 삽입과 삭제가 가능한 연결 리스트를 저장

- 해싱주소로 찾아간 버켓에서는 원하는 항목을 찾기 위해 연결 리스트를 순차 탐색

(예) 크기가 7인 해시테이블에 h(k)=k mod 7 해시 함수를 이용하여 키값 8, 1, 9, 6, 13을 삽입할 때, 체이닝 방식에 의한 충돌 처리과정

1단계 (8) : h(8) = 8 mod 7 = 1(저장)
2단계 (1) : h(1) = 1 mod 7 = 1(충돌발생->새로운 노드 생성 저장)
3단계 (9) : h(9) = 9 mod 7 = 2(저장)
4단계 (6) : h(6) = 6 mod 7 = 6(저장)
5단계 (13) :h(13) = 13 mod 7 = 6(충돌 발생->새로운 노드 생성 저장)



해싱의 성능분석

- **적재 밀도**(loading density) 또는 **적재 비율**(loading factor)
  : 저장되는 데이터 개수 vs. 해시 테이블의 크기

$$\alpha = \frac{\text{저장된 항목의 개수}}{\text{해싱테이블의 버킷의 개수}} = \frac{n}{M}$$

- **오버플로우 발생빈도율**
  : 추가 저장시간, 추가 검색시간과 비례

- 해싱의 시간복잡도
  - 이상적 해싱 : O(1)
  - 실제적 해싱 : 평균의 경우 버켓수, 해싱함수, 입력키값 등에 종속되므로 예측불가
    최악의 경우 O(n)

해싱함수별 성능분석(cont.)

적재율	.50		.70		.90		.95	
해싱 함수	체인	선형조사	체인	선형조사	체인	선형조사	체인	선형조사
중간 제곱	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
제산	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
이동 폴딩	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
경제 폴딩	1.39	22.97	1.57	48.70	1.55	69.63	1.55	97.56
숫자 분석	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
이론적	1.25	1.50	1.37	2.50	1.45	5.50	1.45	10.50

해싱 *vs.* 다른 탐색방법

탐색방법		탐색	삽입	삭제
순차탐색		$O(n)$	$O(1)$	$O(n)$
이진탐색		$O(\log_2 n)$	$O(\log_2 n + n)$	$O(\log_2 n + n)$
이진탐색트리	균형트리	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
	경사트리	$O(n)$	$O(n)$	$O(n)$
해싱	최선의 경우	$O(1)$	$O(1)$	$O(1)$
	최악의 경우	$O(n)$	$O(n)$	$O(n)$

자료구조 교과내용 완료

자료구조 강의 마무리

- 2019-1학기 자료구조
- 강의시간 : (15주) 주 4시간 이론 및 실습
- 강의내용
  - 내용 : 프로그램 개발의 자료구조 이론 및 활용 실습
  - 시험 : 중간, 기말
  - 과제 : 응용실습 문제들
  - 교재 : 자료구조(생능출판사), 강의노트(e-class 업로딩)
  - 기타 : S/W설계과정, IT분야 동향, 대학생활 등

성적 및 기말시험 공지

- 시험 일자 : 다음주 중의 적절한 시간 공지 예정
- 시험 범위 : 8장 ~ 14장
- 문제 유형 : No 프로그램, Yes 수행결과
- 성적 산출(학칙 기준)
  - 총점 = 중간(30) + 기말(40) + 과제(20) + 출석(10)
  - 상대평가 : A학점(30%), B학점(40%), C~F학점(30%)