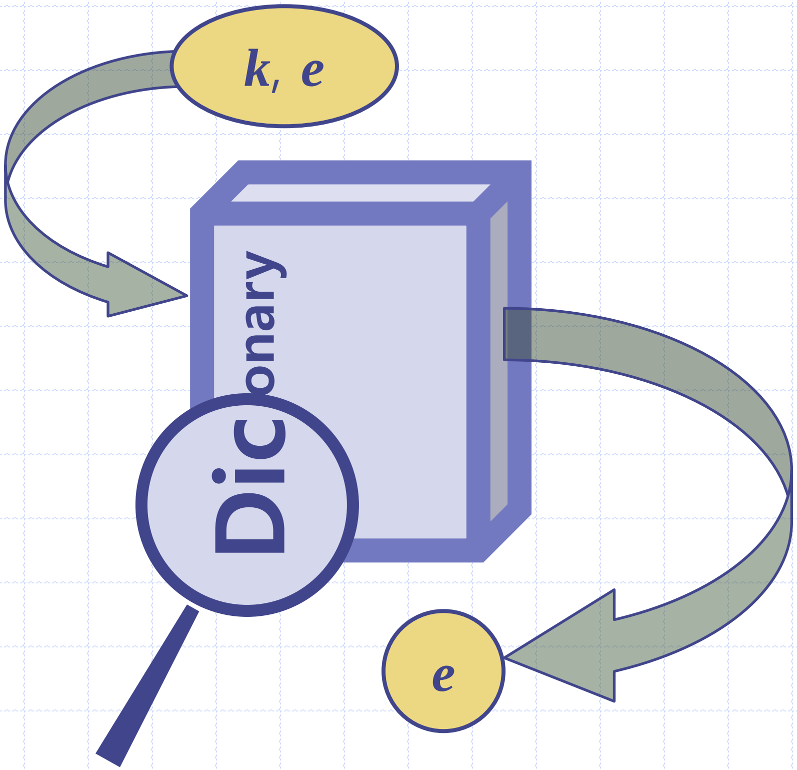
사전

Outline

- 10.1 사전 ADT
- 10.2 사전 ADT 메쏘드
- 10.3 사전 ADT 구현
- 10.4 응용문제

사전 ADT

- 사전 ADT는 탐색 가능한 형태의 (**키**, **원소**) 쌍 항목들의 모음을 모델링
- 사전에 관한 주요 작업
  - **탐색**(searching)
  - **삽입**(inserting)
  - **삭제**(deleting)
- 두 종류의 사전
  - 무순사전 ADT
  - 순서사전 ADT

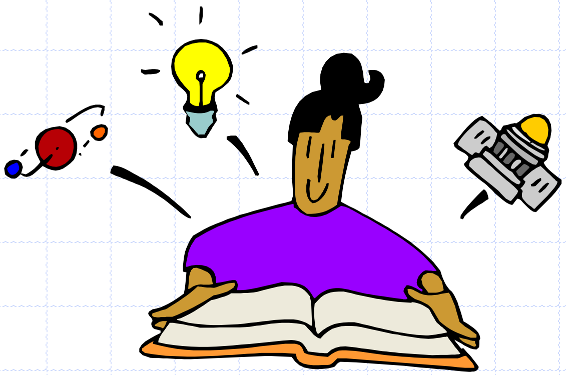


사전 ADT 메쏘드

- 일반 메쏘드
  - integer size(): 사전의 항목 수를 반환
  - boolean isEmpty(): 사전이 비어 있는지 여부를 반환
- 접근 메쏘드
  - element findElement(k): 사전에 키 ***k***를 가진 항목이 존재하면 해당 원소를 반환, 그렇지 않으면 특별 원소 ***NoSuchKey***를 반환
- 갱신 메쏘드
  - insertItem(k, e): 사전에 (***k***, ***e***) 항목을 삽입
  - element removeElement(k): 사전에 키 ***k***를 가진 항목이 존재하면 해당 항목을 삭제하고 원소를 반환, 그렇지 않으면 특별 원소 ***NoSuchKey***를 반환

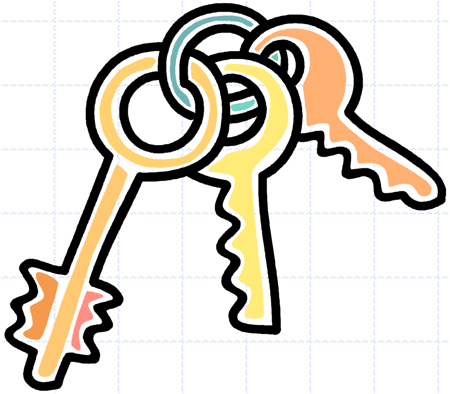
사전 응용

- 직접 응용
  - 연락처 목록
  - 신용카드 사용승인
  - 인터넷주소 매핑
    - 호스트명(**예:** www.sejong.ac.kr)을 인터넷 주소(**예:** 128.148.34.101)로 매핑
- 간접 응용
  - 알고리즘 수행을 위한 보조 데이터구조
  - 다른 데이터구조를 구성하는 요소



탐색

- **비공식적으로, 탐색**(search)은 데이터 집단으로부터 특정한 정보를 추출함을 말한다
- **공식적으로, 탐색**은 사전으로 구현된 데이터 집단으로부터 지정된 키(key)와 연관된 데이터 원소를 반환함을 말한다

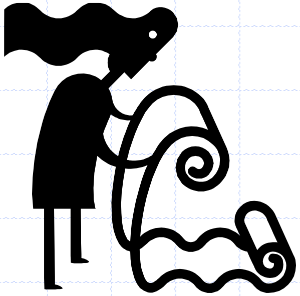


- 상이한 전제
  - **유일 키:** 한 개의 키에 대해 하나의 데이터 항목만 존재
    - **예:** 학번, 은행계좌, login ID
  - **중복 키:** 한 개의 키에 대해 여러 개의 데이터 항목 존재
    - **예:** 이름, 나이, 계좌개설일자

사전 구현에 따른 탐색기법

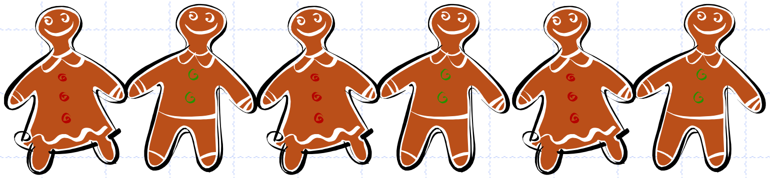
구현 형태	구현 종류	예	주요 탐색 기법
리스트	무순사전 ADT	기록파일	선형탐색
	순서사전 ADT	일람표	이진탐색
트리	탐색트리	이진탐색트리, AVL 트리, 스플레이 트리	트리탐색
해시테이블			해싱

무순사전 ADT: 기록파일



- **기록파일**(log file): **무순리스트**를 사용하여 구현된 사전
  - 사전 항목들을 임의의 순서로 리스트에 저장(이중연결리스트 또는 원형배열을 이용)
- 성능
  - insertItem: 새로운 항목을 기존 리스트의 맨 앞 또는 맨 뒤에 삽입하면 되므로 **O**(1) 시간 소요
  - findElement 및 removeElement: 최악의 경우(즉, 항목이 존재하지 않을 경우), 주어진 키를 갖는 항목을 찾기 위해 리스트 전체를 순회해야 하므로 **O**(***n***) 시간 소요
- 기록파일의 사용이 효과적인 경우
  - 소규모의 사전, 또는
  - 삽입은 빈번하지만 탐색이나 삭제는 드문 사전(**예:** 서버의 로그인 기록)

선형탐색



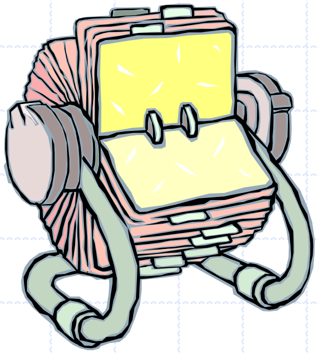
- findElement 작업은 사전에 대해 지정된 키 ***k***에 관한 **선형탐색**(linear search)을 수행하여 ***k***를 가진 원소를 반환

```
Alg findElement(k)                  {generic}
    input list L, key k
    output element with key k

1. L.initialize(i)
2. while (L.isValid(i))
       if (L.key(i) = k)
           return L.element(i)
       else
           L.advance(i)
3. return NoSuchKey
```

선형탐색 분석

- 시간
  - 입력크기(즉, 데이터항목의 수)를 n이라 하면 최악의 경우는 찾고자 하는 키가 맨 뒤에 있거나 아예 없는 경우다
  - 따라서 **O**(n) 시간에 수행
- 공간
  - 입력 데이터구조에 대해 읽기 작업만 수행하므로 **O**(1) 공간으로 수행

순서사전 ADT: 일람표



- **일람표**(lookup table): 순서리스트를 사용하여 구현된 사전
  - 사전 항목들을 배열에 기초한 리스트에 키로 정렬된 순서로 저장
- 성능
  - findElement: **이진탐색**을 사용하면 **O**(log ***n***) 시간 소요
  - insertItem: 새로운 항목을 삽입하기 위한 공간 확보를 위해 최악의 경우 ***n***개의 기존 항목들을 이동해야 하므로 **O**(***n***) 시간 소요
  - removeElement: 항목이 삭제된 공간을 기존 항목들로 메꾸기 위해 최악의 경우 ***n***개의 기존 항목들을 이동해야 하므로 **O**(***n***) 시간 소요
- 일람표 사용이 효과적인 경우
  - 소규모 사전, 또는
  - 탐색은 빈번하지만 삽입이나 삭제는 드문 사전(**예:** 신용카드 사용승인, 전화번호부)

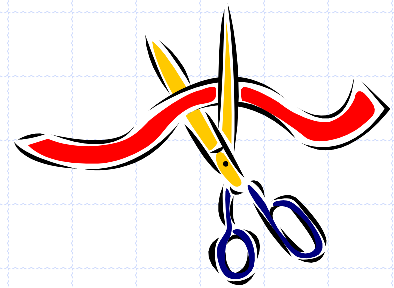
선형탐색

- 실패가 예정된 선형탐색의 경우, 평균적으로 입력크기의 절반 정도만 탐색하고 정지(elseif 절 참조)
- **O**($\boldsymbol{n}$) 시간 소요

Alg ***findElement*(*k*)** {generic}
  input list ***L***, key ***k***
  output element with key ***k***

1. ***L*.*initialize*(*i*)**
2. **while (*L*.*isValid*(*i*))**
    if (*L*.*key*(*i*) = *k*)
      return *L*.*element*(*i*)
    elseif (*L*.*key*(*i*) > *k*)
      return *NoSuchKey*
    else
      ***L*.*advance*(*i*)**
3. **return *NoSuchKey***

이진탐색



- **이진탐색**(binary search): 키로 정렬된 배열에 기초한 리스트로 구현된 사전에 대해 findElement 작업을 수행
- 재귀할 때마다 후보 항목들의 수가 반감
- 입력크기의 로그 수에 해당하는 수의 재귀를 수행한 후 정지
- **참고:** 스무고개

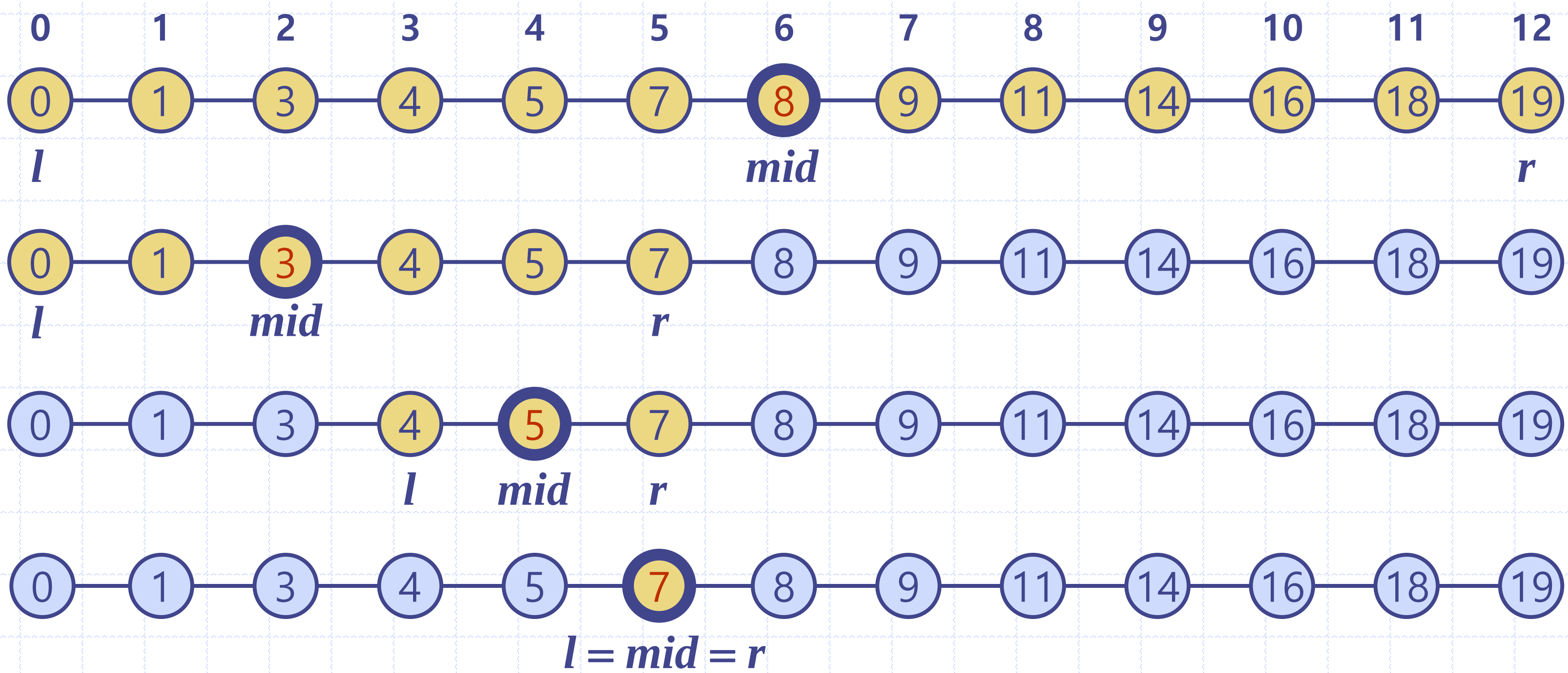
Alg ***findElement*****(*****k*****)** {driver}
  input sorted array $A[0..n-1]$, key k
  output element with key k

1. **return** $rFE(k, 0, n-1)$

Alg $rFE(k, l, r)$ {recursive}
1. **if** $(l > r)$
   return *NoSuchKey*
2. $mid \leftarrow (l + r)/2$
3. **if** $(k = key(A[mid]))$
   return $element(A[mid])$
  elseif $(k < key(A[mid]))$
   return $rFE(k, l, mid-1)$
  else $\{k > key(A[mid])\}$
   return $rFE(k, mid+1, r)$

이진탐색 수행 예

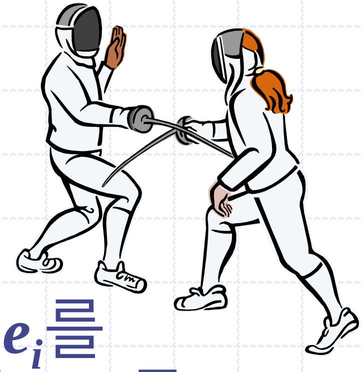
- findElement(7)



이진탐색 분석

- 입력 순서리스트가 **배열**로 구현된 경우
  - 총 비교회수는 최악의 경우 $\mathbf{O}(\log \boldsymbol{n})$
  - 따라서, $\mathbf{O}(\log \boldsymbol{n})$ 시간에 수행
- 입력 순서리스트가 **연결리스트**로 구현된 경우
  - 가운데 위치로 접근하는 데만 $\mathbf{O}(\boldsymbol{n})$ 시간 소요되므로 전체적으로 $\mathbf{O}(\boldsymbol{n})$ 시간에 수행
- **이진탐색의 힘**을 보여주는 예: **스무고개**
  - 2^{20}(약 100만)개의 후보 범위에서 출발하더라도 20회의 이등분이 가능한 질문을 통해 1개 후보(즉, 답)로 압축 가능
- **분할통치** vs. **이진탐색**
  - **분할통치:** 이등분된 두 개의 범위 **양쪽**을 모두 고려
  - **이진탐색:** 이등분된 두 개의 범위 중 **한쪽**만 고려

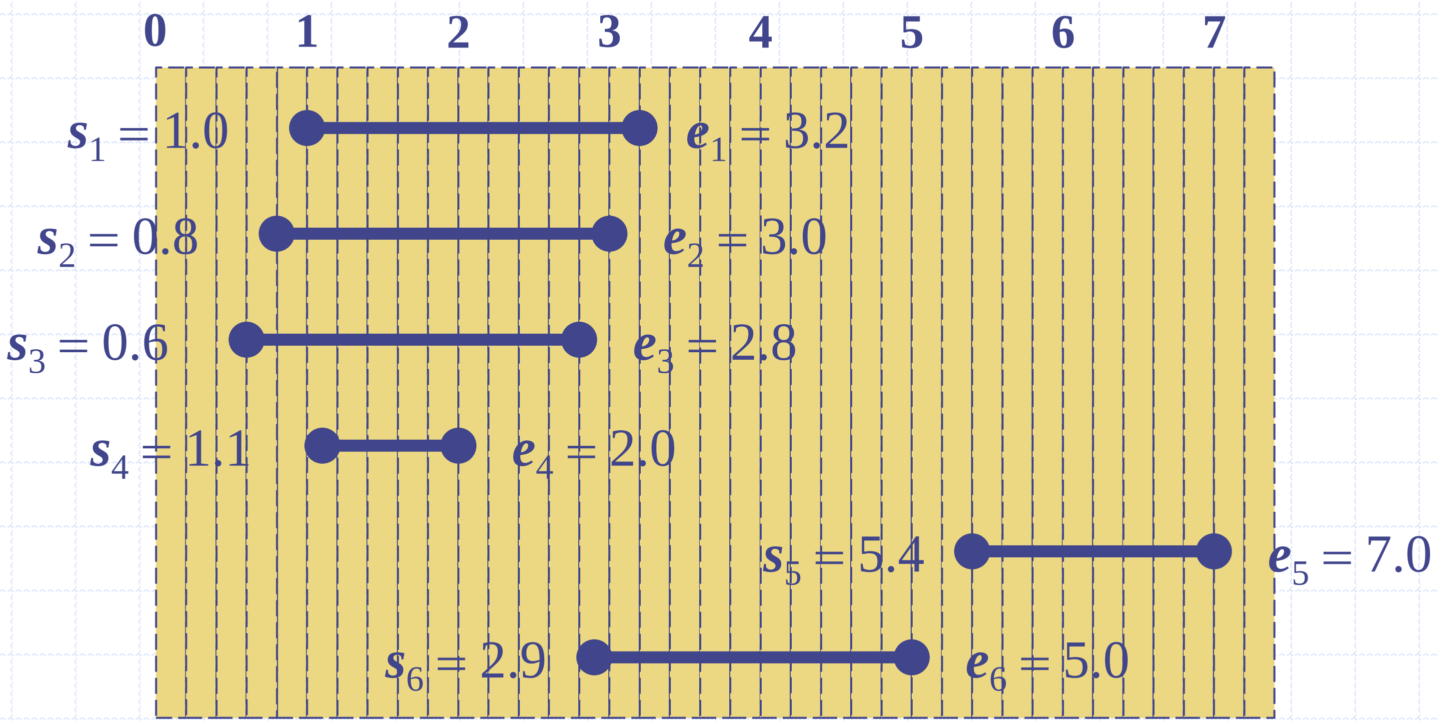
응용문제: 교차 선분



- n개의 **선분**이 있다 – 각 선분 i는 양끝점 s_i와 e_i를 나타내는 실수 쌍 (s_i, e_i)으로 표현되며 s_i와 e_i는 모두 **유일**
- **문제**
  - 교차하는 선분들이 있는지 찾아보고 **몇 번**이나 교차하는지 구하라
  - 교차하는 선분들의 **쌍** (i, j)**의 집합**을 구하라
- **힌트: O(*n* log *n*)**-시간 정렬 함수 sort 사용 가능
- **참고:** 선분에 대한 관점에 따라 다양한 문제에 적용 가능
  - 각 항공편의 운항시간
  - 각 사용자의 서버 접속시간

응용문제: 교차 선분 (conti.)

- **예:** 그림 예에 보인 6개의 선분에는 아래 8개의 교차 쌍이 있다
  (1, 2) (1, 3) (1, 4) (2, 3) (2, 4) (3, 4) (1, 6) (2, 6)



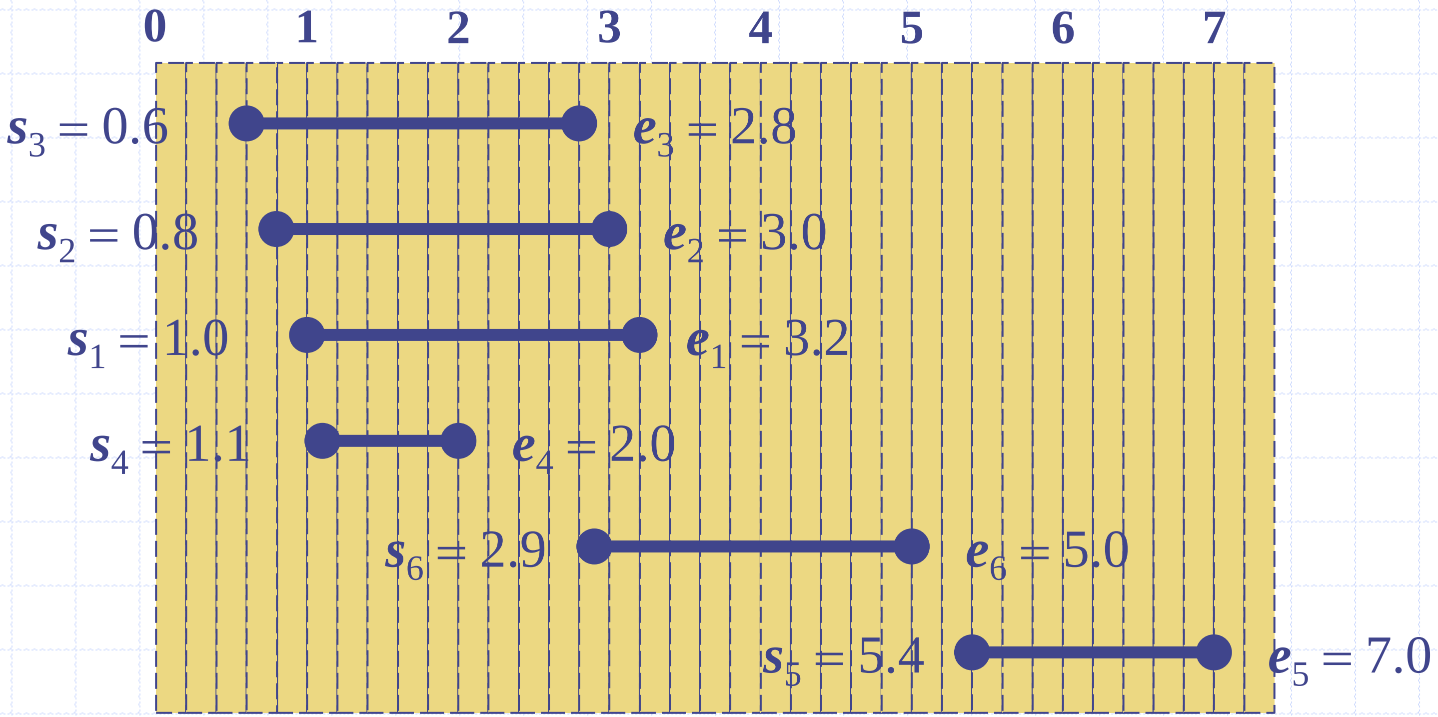
해결

- 각 선분에 대해 두 개의 **이벤트**를 설정
  - **시작**(Start)
  - **끝**(End)
- 각 이벤트 구성 항목: ((좌표, 이벤트코드), 선분 ID)
  - **좌표:** Start 이벤트의 경우 시작점, End의 경우 끝점
  - **이벤트코드: S**(tart) 또는 **E**(nd)
  - **선분 ID**
- **좌표**에 의해 정렬된 **순서사전**을 생성
- 사전의 이벤트들을 차례로 처리
  - 선분의 시작 이벤트에서는,
    교차 선분 ← 교차 선분 ∪ {(열린 선분, 현재 선분)}
    열린 선분 ← 열린 선분 ∪ {현재 선분}
  - 선분의 끝 이벤트에서는,
    열린 선분 ← 열린 선분 – {현재 선분}

해결 (conti.)

- n개의 선분을 $2n$개의 이벤트로 재구성하고 좌표를 기준으로 정렬 – 즉,

((0.6 ***S***), 3), ((0.8 ***S***), 2), ((1.0 ***S***), 1), ((1.1 ***S***), 4), ((2.0 ***E***) 4), ((2.8 ***E***) 3), …)



해결 (conti.)

Alg ***findIntersectingSegments*(*n*)**
input ***n*** line segments with start and end coordinates each
output intersecting segments

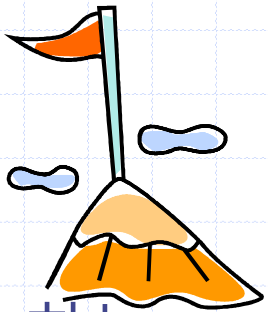
1. $D \leftarrow$ *build a dictionary of 2***n** *events of ((***coordinate***,* ***eventcode***)*, ***ID***)*
pairs sorted by ***coordinate****, where an* ***eventcode*** *is either* ***S***(*tart*) *or* ***E***(*nd*),
and the ***coordinate*** *is a real number* {**O**(***n*** log ***n***)}
2. ***interSegments***, ***openSegments*** $\leftarrow \varnothing$
3. **for each** ***e*** $\in$ ***D*** {**O**(***n***)}
    if (*e*.eventcode = S)
        for each *s* $\in$ ***openSegments*** {**O**(***n***^{2})}
            interSegments $\leftarrow$ ***interSegments*** $\cup$ {(***s***, ***e*****.ID**)}
        openSegments $\leftarrow$ ***openSegments*** $\cup$ {***e*****.ID**}
    else {***e*****.eventcode = E**}
        openSegments $\leftarrow$ ***openSegments*** – {***e*****.ID**}
4. **return** ***interSegments***

{Total **O**(***n***^{2})}

해결 (conti.)

e(vent)	*coordinate*	*openSegments*	*openSegments* 수	*interSegments* 수	*interSegments*
s_3	0.6	$\{s_3\}$	1		{}
s_2	0.8	$\{s_3, s_2\}$	2	+1	$\cup$ {(3, 2)}
s_1	1.0	$\{s_3, s_2, s_1\}$	3	+2	$\cup$ {(3, 1) (2, 1)}
s_4	1.1	$\{s_3, s_2, s_1, s_4\}$	4	+3	$\cup$ {(3, 4) (2, 4) (1, 4)}
e_4	2.0	$\{s_3, s_2, s_1\}$	3		
e_3	2.8	$\{s_2, s_1\}$	2		
s_6	2.9	$\{s_2, s_1, s_6\}$	3	+2	$\cup$ {(2, 6) (1, 6)}
e_2	3.0	$\{s_1, s_6\}$	2		
e_1	3.2	$\{s_6\}$	1		
e_6	5.0	{}	0		
s_5	5.4	$\{s_5\}$	1		
e_5	7.0	{}	0		

응용문제: 단일모드 배열의 최대 원소



- 어떤 배열 $A[0..n-1]$의 원소가 증가하다가 감소하는 원소들로 구성되어 있으면, 배열 A를 **단일모드**(unimodal)라고 한다
- 구체적으로, 다음을 만족하는 배열 A의 첨자 m이 존재하는 경우를 말한다
  - 모든 $0 \le i < m$에 대해, $A[i] < A[i+1]$
  - 모든 $m \le i < n$에 대해, $A[i] > A[i+1]$
- $A[m]$은 최대 원소가 되며, 자신보다 작은 원소들(즉, $A[m-1]$과 $A[m+1]$)로 둘러싸인 단 하나의 **지역 최대**(local maximum) 원소가 된다
- 주어진 단일모드 배열 $A[0..n-1]$의 최대 원소를 $\mathbf{O}(\log n)$ 시간에 찾는 알고리즘을 **의사코드**로 작성하라

응용문제: 단일모드 배열의 최대 원소 (conti.)

- **예:** 아래 배열 A는 단일모드 배열이다
- 배열 A의 최대원소는 $A[5] = 41$

A

0	1	2	3	4	5	6	7	8
–21	8	12	13	35	41	23	20	17

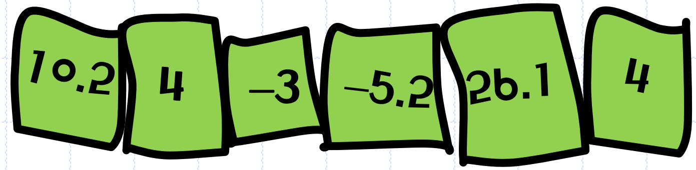
해결

- 단일모드 배열의 정의에 의해, $0 \le i < n$에 대해 $A[i] < A[i+1]$이거나 $A[i] > A[i+1]$이다
  - 이 두 가지 경우를 구분하는 데만 집중하면 된다
- 만약 $A[i] < A[i+1]$이면 $A[0..\,n-1]$의 최대 원소는 $A[i+1..\,n-1]$에 존재
- 반대로, 만약 $A[i] > A[i+1]$이면 $A[0..\,n-1]$의 최대 원소는 $A[0..\,i]$에 존재

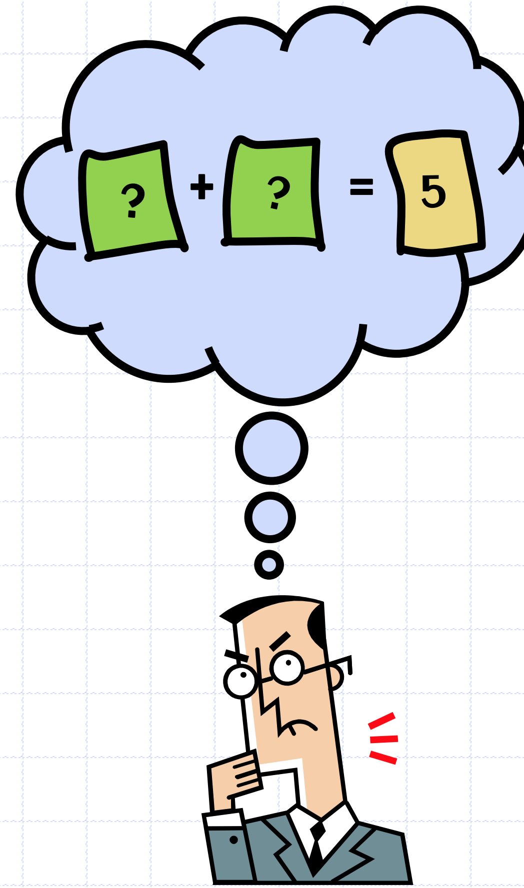
```
Alg findMaxOfUnimodalArray(A, n)
    input unimodal array A of size n
    output the maximum element of A

1. a ← 0
2. b ← n – 1
3. while (a < b)
       mid ← ⌊(l + r)/2⌋
       if (A[mid] < A[mid + 1])
           a ← mid + 1
       if (A[mid] > A[mid + 1])
           b ← mid
4. return A[a]
```

- **이진탐색**과 유사한 절차로 수행하므로, 실행시간: $\mathbf{O}(\log n)$

응용문제: 배열의 두 수 덧셈



- 정수가 아닐 수도 있는 수들의 **무순배열** $A[0..n-1]$가 있다 – A의 원소는 중복이 있을 수 있다
- 역시 정수가 아닐 수도 있는 주어진 수 s에 대해, $A[i_1] + A[i_2] = s$를 만족하는 (i_1, i_2) 쌍을 찾는 알고리즘을 작성하라 – 그런 쌍이 여러 개 있더라도 하나만 찾으면 된다
- **힌트**
  - $O(n^2)$보다 빨라야 한다
  - $O(n \log n)$-시간 정렬 함수 sort 사용 가능



응용문제: 배열의 두 수 덧셈 (conti.)

- 예

A

	0	1	**2**	3	**4**	5	6	7	
s = 13	2	21	**8**	3	**5**	1	13	1	→ (2, 4)

	0	1	2	3	4	**5**	6	**7**	
s = 2	2	21	8	3	5	**1**	13	**1**	→ (5, 7)

	0	1	2	3	4	5	6	7	
s = 12	2	21	8	3	5	1	13	1	→ ***NotFound***

해결

Alg ***buildDictionaryForPairs***(A)
    input array $A[0..n-1]$ of numbers
    output an ordered dictionary D with $(A[i], i)$ pairs for i in $0..n-1$,
        i.e., maps numbers in A to the indices of the numbers

1. $D \leftarrow$ *empty dictionary*
2. **for** $i \leftarrow 0$ **to** $n-1$ $\{\mathbf{O}(n)\}$
        D.***insertItem***$(A[i], i)$ $\{\mathbf{O}(1)\}$
3. **return** ***sort***(D) $\{\mathbf{O}(n \log n)\}$
    {Total $\mathbf{O}(n \log n)$}

- 우선, buildDictionaryForPairs 알고리즘을 수행하여, 배열로부터 **순서사전** D를 생성
- 생성된 D의 각 항목은 (A **배열원소**, A **배열첨자**)로 이루어진 **키-원소** 쌍
- 다음, findIndexPair 알고리즘을 수행하여 배열의 각 원소를 순회하면서 각 원소의 대응수(즉, 더해서 s가 되는 수)를 D에서 **이진탐색**에 의해 찾는다

해결 (conti.)

```
Alg findIndexPair(A, s)
    input array A[0..n – 1] of numbers, number s
    output an index pair (i0, i1), s.t. A[i0] + A[i1] = s

1. D ← buildDictionaryForPairs(A)            {O(n log n)}
2. for i ← 0 to n – 1                        {O(n)}
       s′ ← s – A[i]
       j ← D.findElement(s′)                 {O(n log n)}
       if (j ≠ NoSuchKey)
           return i, j
3. return NotFound
                                             {Total O(n log n)}
```

답 (conti.)

- **수행 예**

	0	1	2	3	4	5	6	7
A	2	21	8	3	5	1	13	1

D (1, 5), (1, 7), (2, 0), (3, 3), (5, 4), (8, 2), (13, 6), (21, 1)

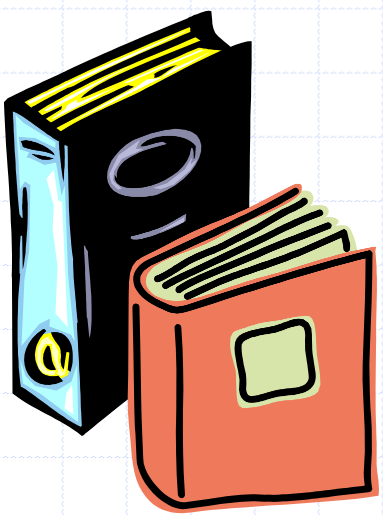
s = 13 인 경우, i = 2 에 대해

	0	1	2	3	4	5	6	7
A	2	21	**8**	3	**5**	1	13	1

D (1, 5), (1, 7), (2, 0), (3, 3), **(5, 4)**, (8, 2), (13, 6), (21, 1)

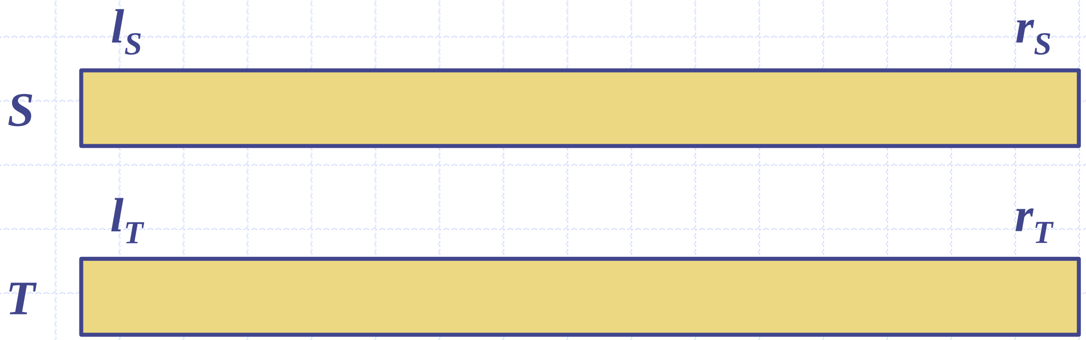
응용문제: 두 개의 사전에서 k-번째 작은 키

- 두 개의 순서사전 ***S***와 ***T***가 있다
- ***S***와 ***T*** 모두 ***n***개의 항목으로 구성되었으며 배열에 기초한 순서리스트로 구현되어 있다
- ***S***와 ***T*** 전체에서 ***k***번째 작은 키를 찾는 **O**(log ***n***)-시간 알고리즘을 **의사코드**로 작성하라
- **전제:** 중복 키는 없음

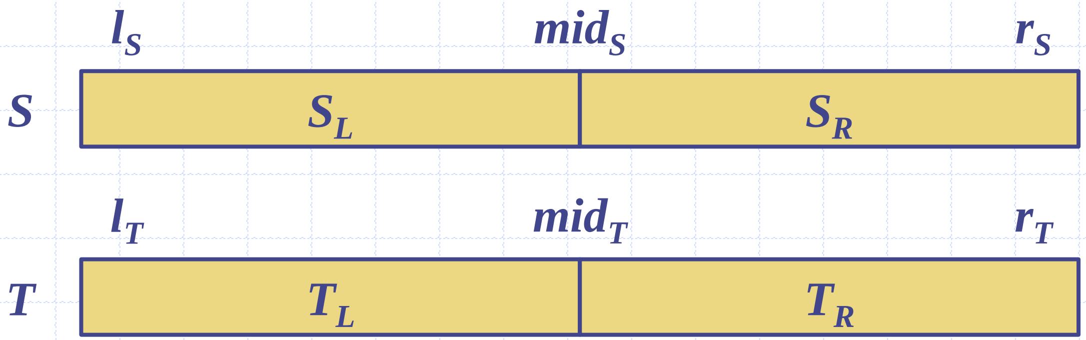


해결

- **이진 탐색**으로 해결
- 초기에는 S와 T의 전체 구역에 목표 키가 존재할 가능성이 있으므로 전체 구역을 고려범위에 포함
- S와 T를 각각 이등분하여 그림과 같이 전체를 4개의 구역, S_L, S_R, T_L, T_R 로 나누어 고려
- k와 S_L과 T_L 두 구역을 합한 크기를 비교
- 비교 결과 =, <, >의 각 경우마다 mid_S와 mid_T의 키를 비교하여 4개의 구역 가운데 고려 대상에서 제외할 구역을 찾아 걸러낸다



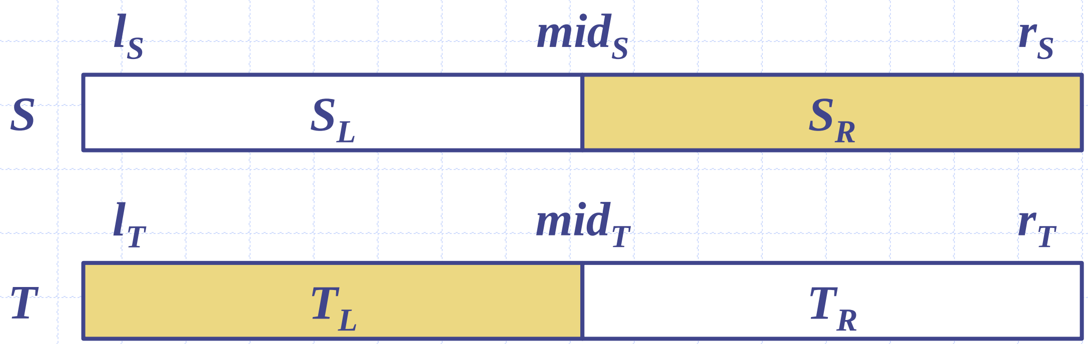
(a) 초기



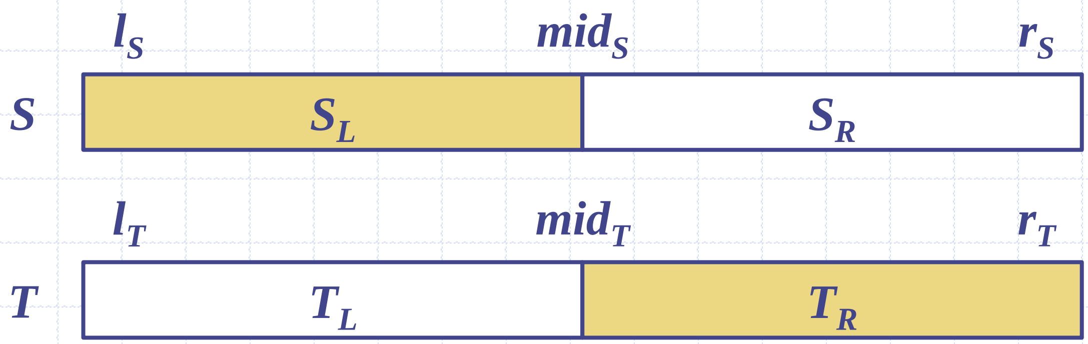
(b) $mid_S \leftarrow (l_S + r_S)/2$, $mid_T \leftarrow (l_T + r_T)/2$

해결 (conti.)

- 첫번째 경우로, k가 S_L과 T_L을 더한 크기와 같은 $k = |S_L| + |T_L|$인 경우, 양쪽의 중앙 키를 비교
- 만약 $key(S[mid_S]) < key(T[mid_T])$이면, 목표 키는 S_R, T_L을 합한 구역에서 $k - |S_L|$번째 큰 키
- 그렇지 않으면, 목표 키는 S_L, T_R을 합한 구역에서 $k - |T_L|$번째 큰 키



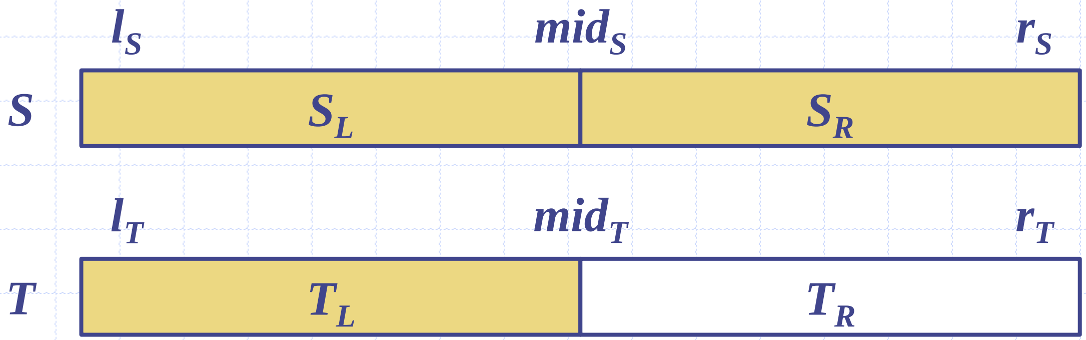
(c) $k = |S_L| + |T_L|$, $key(S[mid_S]) < key(T[mid_T])$



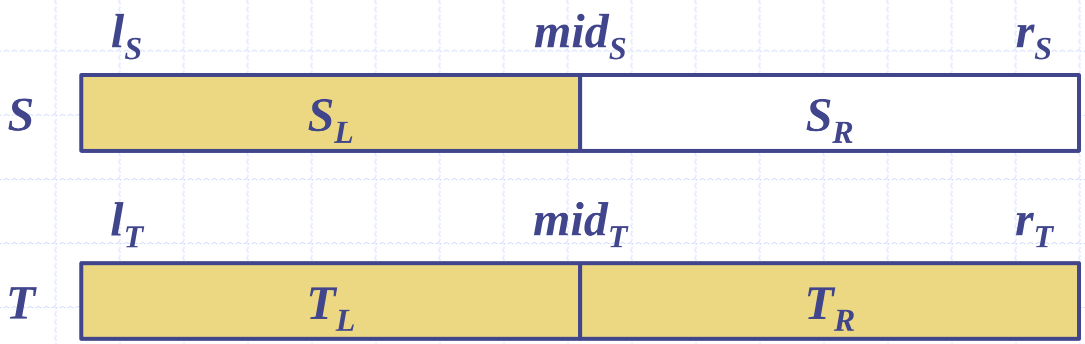
(d) $k = |S_L| + |T_L|$, $key(S[mid_S]) > key(T[mid_T])$

해결 (conti.)

- 두번째 경우로, k가 S_L과 T_L을 더한 크기보다 작은 $k < |S_L| + |T_L|$인 경우, 양쪽의 중앙 키를 비교
- 만약 $key(S[mid_S]) < key(T[mid_T])$이면, 목표 키는 S_L, S_R, T_L을 합한 구역에서 k번째 큰 키
- 그렇지 않으면, 목표 키는 S_L, T_L, T_R을 합한 구역에서 k번째 큰 키



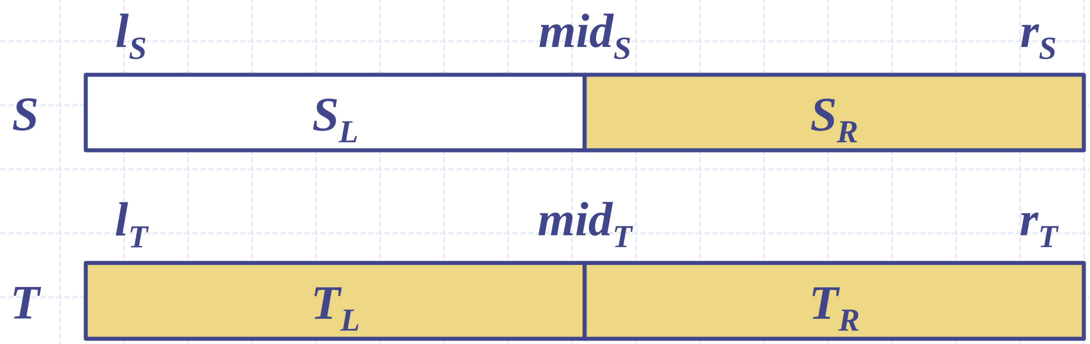
(e) $k < |S_L| + |T_L|$, $key(S[mid_S]) < key(T[mid_T])$



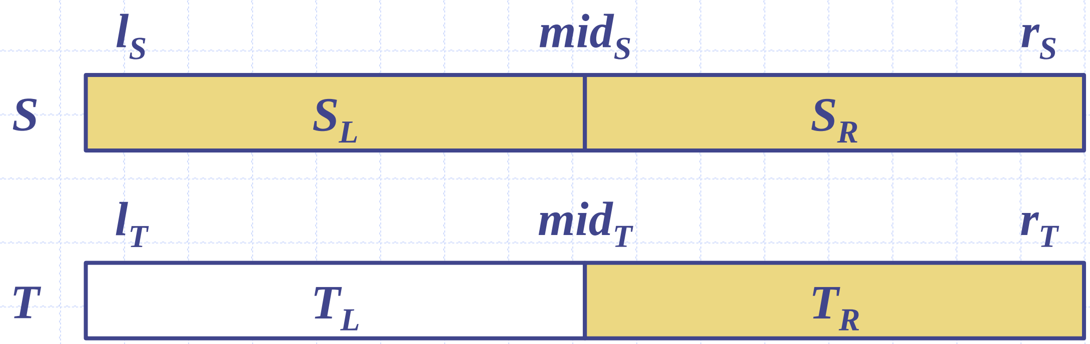
(f) $k < |S_L| + |T_L|$, $key(S[mid_S]) > key(T[mid_T])$

해결 (conti.)

- 마지막 경우로, k가 S_L과 T_L을 더한 크기보다 큰 $k > |S_L| + |T_L|$인 경우, 양쪽의 중앙 키를 비교
- 만약 $key(S[mid_S]) < key(T[mid_T])$이면, 목표 키는 S_R, T_L, T_R을 합한 구역에서 $k - |S_L|$번째 큰 키
- 그렇지 않으면, 목표 키는 S_L, S_R, T_R을 합한 구역에서 $k - |T_L|$번째 큰 키



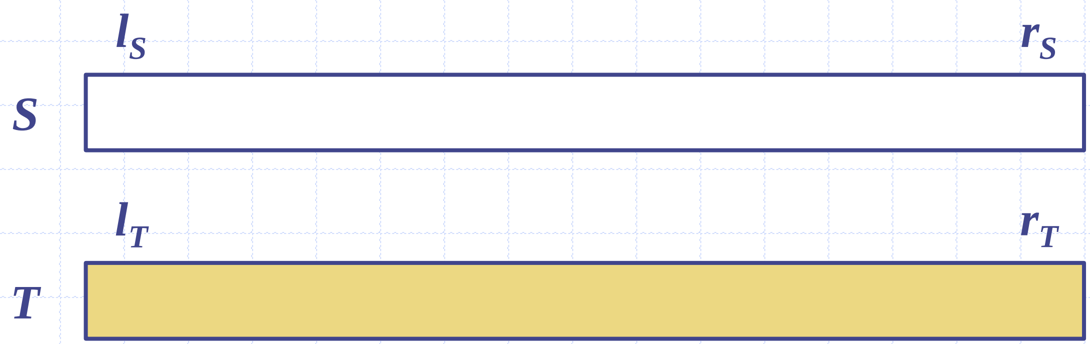
(g) $k > |S_L| + |T_L|$, $key(S[mid_S]) < key(T[mid_T])$



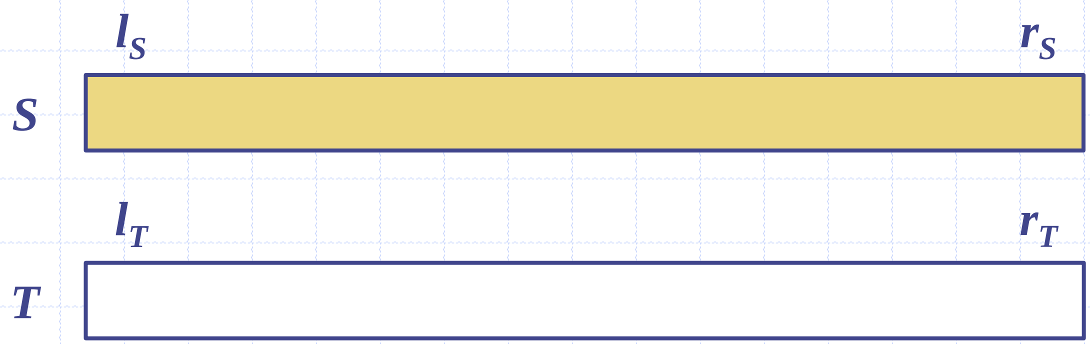
(h) $k > |S_L| + |T_L|$, $key(S[mid_S]) > key(T[mid_T])$

해결 (conti.)

- S나 T 어느 한쪽에 고려할 구역이 비게 되면 반대쪽 구역에서 k 번째 원소를 찾아 반환



(i) $l_S > r_S$



(j) $l_T > r_T$

답 (conti.)

Alg ***findKth*(k)** {driver}
  input sorted array $S[0 .. n-1]$, $T[0 .. n-1]$,
    integer k
  output k-th smallest key in $S \cup T$

1. **return** $rFK(k, 0, n-1, 0, n-1)$

답 (conti.)

Alg $rFK(k, l_S, r_S, l_T, r_T)$
{recursive}

input sorted array $S[l_S .. r_S]$, $T[l_T .. r_T]$, integer k, index l_S, r_S, l_T, r_T

output k-th smallest key in $S[l_S .. r_S] \cup T[l_T .. r_T]$

1. **if** $(l_S > r_S)$
   return $key(T[l_T + k - 1])$
2. **if** $(l_T > r_T)$
   return $key(S[l_S + k - 1])$
3. $mid_S \leftarrow \lfloor (l_S + r_S)/2 \rfloor$
4. $mid_T \leftarrow \lfloor (l_T + r_T)/2 \rfloor$
5. $|S_L| \leftarrow mid_S - l_S + 1$
6. $|T_L| \leftarrow mid_T - l_T + 1$
7. **if** $(k = |S_L| + |T_L|)$
   if $(key(S[mid_S]) < key(T[mid_T]))$
      return $rFK(k - |S_L|, mid_S + 1, r_S, l_T, mid_T)$
   else $\{key(S[mid_S]) > key(T[mid_T])\}$
      return $rFK(k - |T_L|, l_S, mid_S, mid_T + 1, r_T)$
   elseif $(k < |S_L| + |T_L|)$
   if $(key(S[mid_S]) < key(T[mid_T]))$
      return $rFK(k, l_S, r_S, l_T, mid_T - 1)$
   else $\{key(S[mid_S]) > key(T[mid_T])\}$
      return $rFK(k, l_S, mid_S - 1, l_T, r_T)$
   else $\{k > |S_L| + |T_L|\}$
   if $(key(S[mid_S]) < key(T[mid_T]))$
      return $rFK(k - |S_L|, mid_S + 1, r_S, l_T, r_T)$
   else $\{key(S[mid_S]) > key(T[mid_T])\}$
      return $rFK(k - |T_L|, l_S, r_S, mid_T + 1, r_T)$