가천대학교
Gachon University

06

Algorithm

탐색 알고리즘

2025-04-11

조윤실

목 차

※ 가천대학교 컴퓨터공학과 '알고리즘' 과정

탐색 알고리즘

탐색 알고리즘

‘탐색(Search)’ 하면 연상 되는 것은?

탐색

- 탐색(Search)이란?
  - 데이터의 집합에서 원하는 조건을 만족하는 데이터를 찾는 작업
  - 테이블에서 원하는 탐색키를 가진 레코드를 찾는 작업

[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

탐색 알고리즘

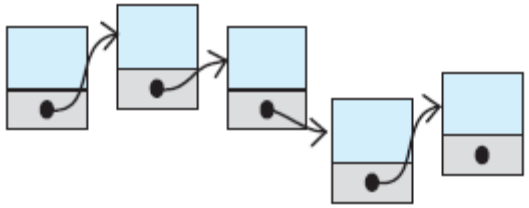
- 탐색 알고리즘(Search Algorithm)이란?
  - 주어진 데이터 집합에서 특정 값이나 조건을 만족하는 요소를 찾는 알고리즘
  - 데이터를 효율적으로 탐색하여 원하는 결과를 얻을 수 있도록 함
  - 각 알고리즘은 특정 상황과 데이터 구조에 따라 장단점이 있음
    - ➢ 데이터 정렬 여부, 데이터의 양이나 구조에 따라서 탐색 속도가 달라질 수 있음

(*적절한 알고리즘 선택하는 것이 중요*)

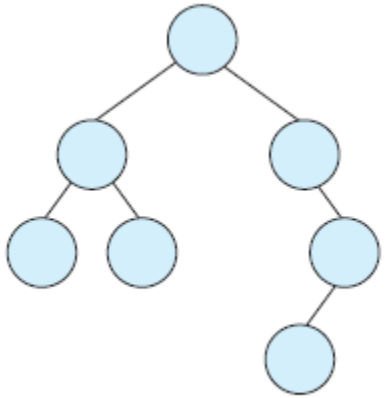
탐색 알고리즘

- 탐색 알고리즘에서 자주 사용하는 자료 구조
  - 테이블을 구성하는 방법(데이터 구조)에 따라 효율이 달라짐

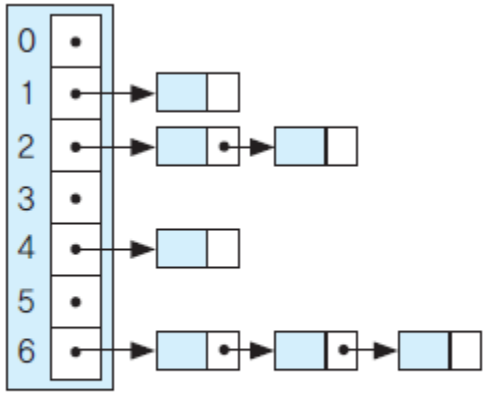
배열



연결 리스트



탐색 트리



해싱

[고림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

탐색 알고리즘으로 무엇을 할 수 있을까?

- 탐색 알고리즘의 주요 역할
  - 데이터 탐색(Data Search) : 데이터 구조에서 특정 값이나 키를 가진 요소 찾기
  - 경로 탐색(Path Finding) : 그래프 혹은 네트워크에서 두 노드 사이의 최단 경로나 최적 경로 찾기
  - 결정 문제 해결(Decision Problem Solving) : 주어진 제약 조건 하에서 해답의 존재 여부 결정
  - 최적화 문제 해결(Optimization Problem Solving) : 주어진 제약 조건 하에서 최적의 해답
  - 패턴 매칭(Pattern Matching) : 문자열, 이미지, 신호 등의 데이터에서 특정 패턴 찾기

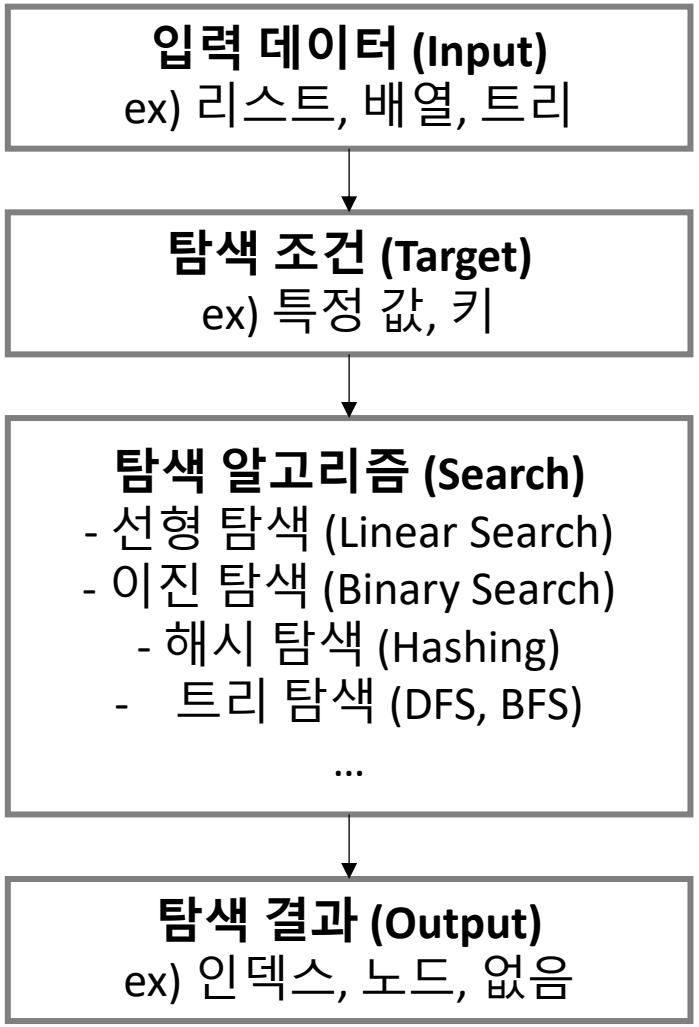
탐색 알고리즘의 응용 분야

- **탐색 알고리즘이 사용되는 대표적 기술과 응용 분야**
  - **인공지능 및 기계학습** : 그래프 탐색 알고리즘은 지식 그래프를 생성하고 분석하는 데 사용
  - **자연어 처리** : 문자열 탐색은 자연어 처리 분야에서 텍스트 분석, 정보 추출, 검색 엔진 등에 사용
  - **최적화 문제** : 유전 알고리즘(Genetic Algorithm)은 탐색 공간을 탐색하고 최적의 해를 찾는 데 사용
  - **데이터베이스 및 정보 검색** : B-트리와 같은 트리 기반의 탐색 알고리즘은 DB 인덱싱에 사용
  - **로봇 공학 및 자율 주행** : A* & 그래프 탐색 알고리즘이 로봇의 이동 경로를 계획하는 데 사용
  - **네트워크 및 라우팅** : 네트워크에서 데이터 패킷의 전송 경로를 결정하는 라우팅 알고리즘에 다익스트라와 같은 다양한 알고리즘이 사용

[Quiz]

- 파이썬 언어에서 탐색 기능이 내재(적용)된 부분은?

탐색 알고리즘 아키텍처 : (기초 알고리즘 예)



검색 방식 : 키워드 매칭

탐색 알고리즘 아키텍처 : (구글 검색 예)

Google 검색의 3단계 소개

Google 검색은 세 단계로 작동하며, 각 단계가 모든 페이지에 적용되는 것은 아닙니다.

1. **크롤링:** Google은 크롤러라는 자동화된 프로그램을 사용하여 인터넷에서 찾은 페이지로부터 텍스트, 이미지, 동영상을 다운로드합니다.
2. **색인 생성:** Google은 페이지의 텍스트, 이미지, 동영상 파일을 분석하고 대규모 데이터베이스인 Google 색인에 이 정보를 저장합니다.
3. **검색결과 게재:** 사용자가 Google에서 검색하면 Google에서는 사용자의 검색어와 관련된 정보를 반환합니다. 참고 링크

- TF-IDF(Term Frequency-Inverse Document Frequency)
  문서 내에서 **어떤 단어가 얼마나 중요한지를 수치화**하는 대표적인 방법
- BM25(Best Matching 25)
  TF-IDF의 약점을 개선한 **확률적 검색 모델 기반의 스코어링 알고리즘**
  현대 검색 엔진에서 가장 널리 쓰이는 랭킹 함수 중 하나

사용자 Query 입력

↓

쿼리 분석 (Query Parser)
ex) 형태소 분석, 정규화, 추천어

↓

색인(Index) 시스템
- 역색인(Inverted Index)
- 문서 ID, 단어 위치 정보

↓

검색 알고리즘 수행
- BM25 / TF-IDF 점수화
- PageRank, BERT rerank

↓

결과 정렬 및 필터링
- 관련도 순 정렬
- 사용자 맞춤 추천

↓

사용자에게 결과 반환
ex) 제목, 링크, 요약 문구

탐색 알고리즘 아키텍처 : (AI 검색 알고리즘)

검색 방식 : 의미 기반 유사도 매칭

토큰화, 형태소 분석, 불용어 제거 등

쿼리와 문서를 의미기반 벡터로 변환 (BERT, SBERT, E5 등)

수백만 개 문서 임베딩을 벡터 DB에 저장 (FAISS, Pinecone 등)

Cosine similarity, dot product 기반으로 가장 유사한 Top-k 문서 추출

BM25 기반 또는 BERT를 활용한 정밀한 재정렬

필요시 LLM이 요약하거나 자연어 응답 생성 (RAG 구조)

참고: https://www.elastic.co/kr/blog/understanding-ai-search-algorithms

주요 탐색 알고리즘

- 선형 탐색(Linear Search)
- 이진 탐색(Binary Search)
- 해시 탐색(Hash Search)
- 트리 기반 탐색
- 그래프 탐색 알고리즘
- 문자열 탐색 알고리즘

주요 탐색 알고리즘

분류 기준	알고리즘 유형	사용 자료구조	정렬 필요 여부	시간 복잡도 (평균/최악)	공간 복잡도	특징 및 용도
선형 탐색 여부	**선형 탐색 (Linear Search)**	**리스트, 배열**	**❌ 불필요**	**O(n) / O(n)**	**O(1)**	단순 반복으로 탐색, 데이터가 적을 때 유리
	이진 탐색 (Binary Search)	**정렬된 배열**	**✅ 필요**	**O(log n) / O(log n)**	**O(1)**	정렬된 데이터에서 빠르게 탐색 가능
해시 기반 탐색	**해시 탐색 (Hash Search)**	**해시 테이블 (dict)**	**❌ 불필요**	**O(1) / O(n)**	**O(n)**	키 기반 탐색, 충돌 시 성능 저하 가능
트리 기반 탐색	이진 탐색 트리 (BST)	트리 구조	❌ 불필요	O(log n) / O(n)	O(n)	정렬+탐색을 모두 지원, 불균형 트리의 경우 성능 저하
	AVL / Red-Black Tree	균형 이진 탐색 트리	❌ 불필요	O(log n) / O(log n)	O(n)	균형 유지로 최악의 경우에도 빠른 탐색 가능
그래프 기반 탐색	DFS (깊이 우선 탐색)	스택, 재귀, 그래프	❌ 불필요	O(V + E)	O(V)	미로, 백트래킹, 순열 문제 등에 유용
	BFS (너비 우선 탐색)	큐, 그래프	❌ 불필요	O(V + E)	O(V)	최단 거리 탐색, 그래프 레벨 탐색에서 유용
문자열 탐색	KMP, Rabin-Karp 등	문자열	❌ 불필요	O(n + m) 등	O(m)이상	문자열 패턴 매칭, 정보 검색 시스템에 활용
현대적 검색 알고리즘	BM25, BERT reranker 등	역색인, 신경망 등	❌ 불필요	수치 기반, 근사 검색	모델 의존	구글, 챗봇 등에서 자연어 기반 검색에 사용

탐색 알고리즘 속 연산 예시

연산 예시	연산 종류	사용 알고리즘/자료구조	탐색 연산의 핵심 질문 예:
3 in [1,2,3,4]	존재 여부 확인	선형 탐색	값이 존재하는가?
arr.index(5)	인덱스 탐색	선형/이진 탐색	어디에 있는가?
dict['name']	키 기반 값 검색	해시 탐색	값이 존재하는가? 어디에 있는가?
max(arr),min(arr)	최댓값/최솟값 탐색	선형 탐색 / 힙	무엇이 가장 큰가/작은가?
find("apple")	문자열 검색	KMP, Rabin-Karp 등	값이 존재하는가? 어디에 있는가?
A* 알고리즘	최단 경로 탐색	그래프 탐색 알고리즘	어디까지 가는 경로가 최선인가?
FAISS, BM25	유사도 기반 검색	Dense/Sparse 벡터 기반	어떤 것이 가장 유사한가?

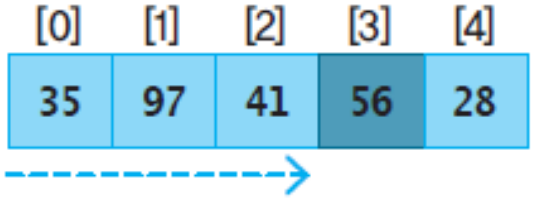
선형 탐색

선형 탐색

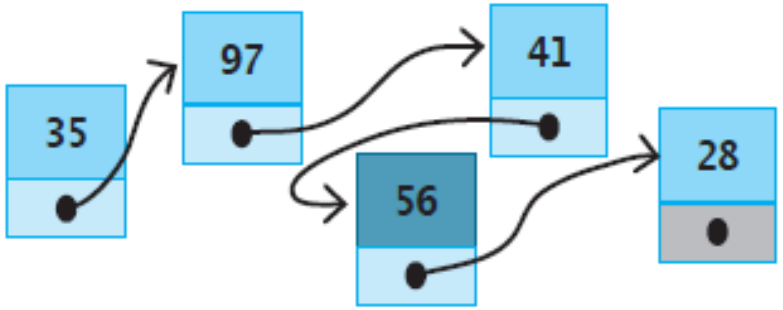
- ## 선형 탐색(Linear Search)이란?
  - 배열의 처음부터 끝까지 순차적으로 모든 요소를 확인 → 순차 탐색(Sequential search)
  - 정렬되지 않은 데이터에서도 사용 가능
  - 가장 단순한 탐색 방법
  - 시간 복잡도 : $O(n)$
    - 최선의 경우: O(1) - 첫 번째 요소가 찾는 값일 때
    - 평균/최악의 경우: O(n) - n은 배열의 크기

선형 탐색 알고리즘

- 선형 탐색(Linear Search) 알고리즘



배열 구조의 테이블(56 탐색)



연결된 구조의 테이블(56 탐색)

- 테이블이 배열인 경우

```
01:  def sequential_search(A, key, low, high) :
02:      for i in range(low, high+1) :          # i : low, low+1, ... high
03:          if A[i] == key :                   # 탐색 성공하면
04:              return i                       # 인덱스 반환
05:      return -1                              # 탐색에 실패하면 -1 반환
```

[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

선형 탐색 동작 방식

▪ 선형 탐색 동작 방식

① **시작**: 탐색할 데이터 구조의 첫 번째 요소부터 시작

② **요소 확인**: 현재 위치에서 요소를 하나씩 확인, 확인된 요소와 찾고자 하는 값 비교

③ **일치 여부 확인**: 현재 요소와 찾고자 하는 값이 일치하는지 확인

- **일치하는 경우**: 탐색을 종료하고 해당 위치를 반환
- **일치하지 않는 경우**: 다음 요소를 확인하기 위해 탐색을 계속 진행

④ **끝까지 탐색**: 위의 과정을 마지막 요소까지 반복. 만약 찾고자 하는 값이 존재하지 않는 경우, 탐색을 마친 후 '값을 찾을 수 없음'을 나타내는 메시지를 반환

선형 탐색 알고리즘

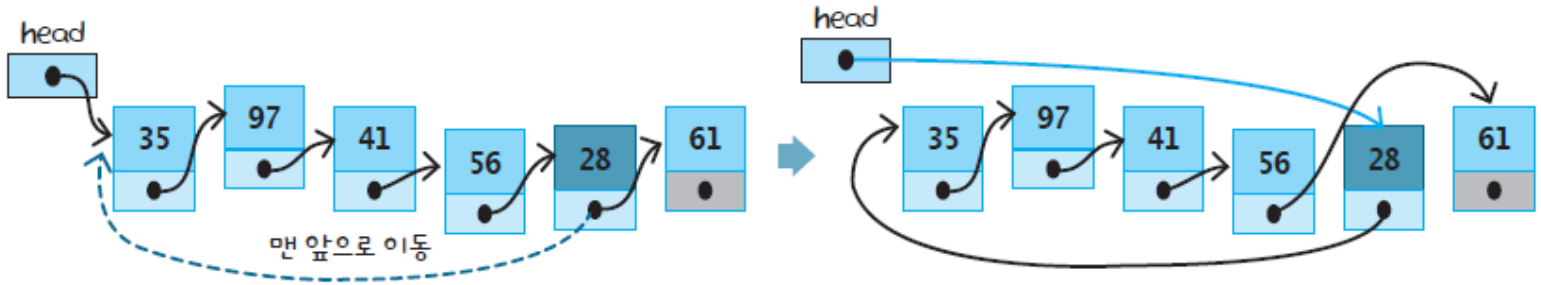
```
def linear_search(A, target):
    for i in range(len(A)):# 시작, 요소 확인
        if A[i] == target :      # 탐색 성공하면
            return i          # 값의 인덱스 반환
    return -1                 # 찾지 못한 경우 -1 반환


# 예시
arr = [ 3, 9, 15, 22, 31, 55, 67, 88, 91 ]
target = 22   # 찾는 값
print(f"#선형 탐색(idx): {linear_search(arr, target)}" )
```

선형 탐색 개선 방법

- 방법1: 빈도 기반 최적화 (Move-To-Front / Self-Organizing List)
  - 자주 사용되는 레코드를 앞쪽에 옮기는 방법
  - 맨 앞으로 보내기 : 탐색에 성공한 레코드를 리스트의 맨 앞으로 보내는 방법
  - 자기 구성(self-organizing) 순차 탐색

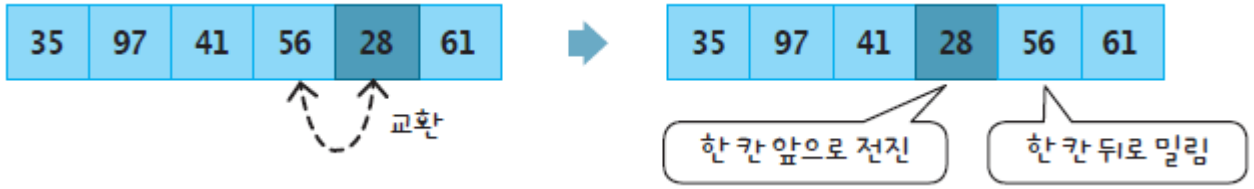
- 탐색된 레코드가 향후에도 많이 탐색될 가능성이 큰 경우 적용
- O(n) → O(1)



[그림 출처]: 자료구조와 알고리즘 with 파이썬 by 생능북스

선형 탐색 개선 방법

- 방법2 : 교환하기
  - 탐색된 레코드를 바로 앞의 레코드와 교환하는 전략




```
01:  def sequential_search_transpose(A, key, low, high) :
02:      for i in range(low, high+1) :
03:          if A[i] == key :
04:              if i > low :            # 맨 처음 요소가 아니면
05:                  A[i], A[i-1] = A[i-1], A[i] # 교환하기(transpose)
06:                  i = i-1             # 한 칸 앞으로 왔음
07:              return i                # 탐색에 성공하면 키 값의 인덱스 반환
08:      return -1                   # 탐색에 실패하면 -1 반환
```

- 탐색된 레코드가 향후에도 많이 탐색될 가능성이 큰 경우 적용
- O(n) → O(1)

[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

선형 탐색 개선 방법

- 방법3 : 정렬 + 이진 탐색으로 대체
  - 데이터가 정렬 가능하거나 자주 탐색이 필요한 경우,
    미리 정렬한 후 이진 탐색으로 대체

```
arr.sort()
index = binary_search(arr, key)
```

- 탐색된 레코드가 향후에도 많이 탐색될 가능성이 큰 경우 적용
- O(log n)

[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

실습문제 : 선형 탐색 알고리즘 구현하기

- 선형 탐색 알고리즘을 파이썬으로 구현하세요.
  1) 선형 탐색 알고리즘
  2) 선형 탐색 개선 방법

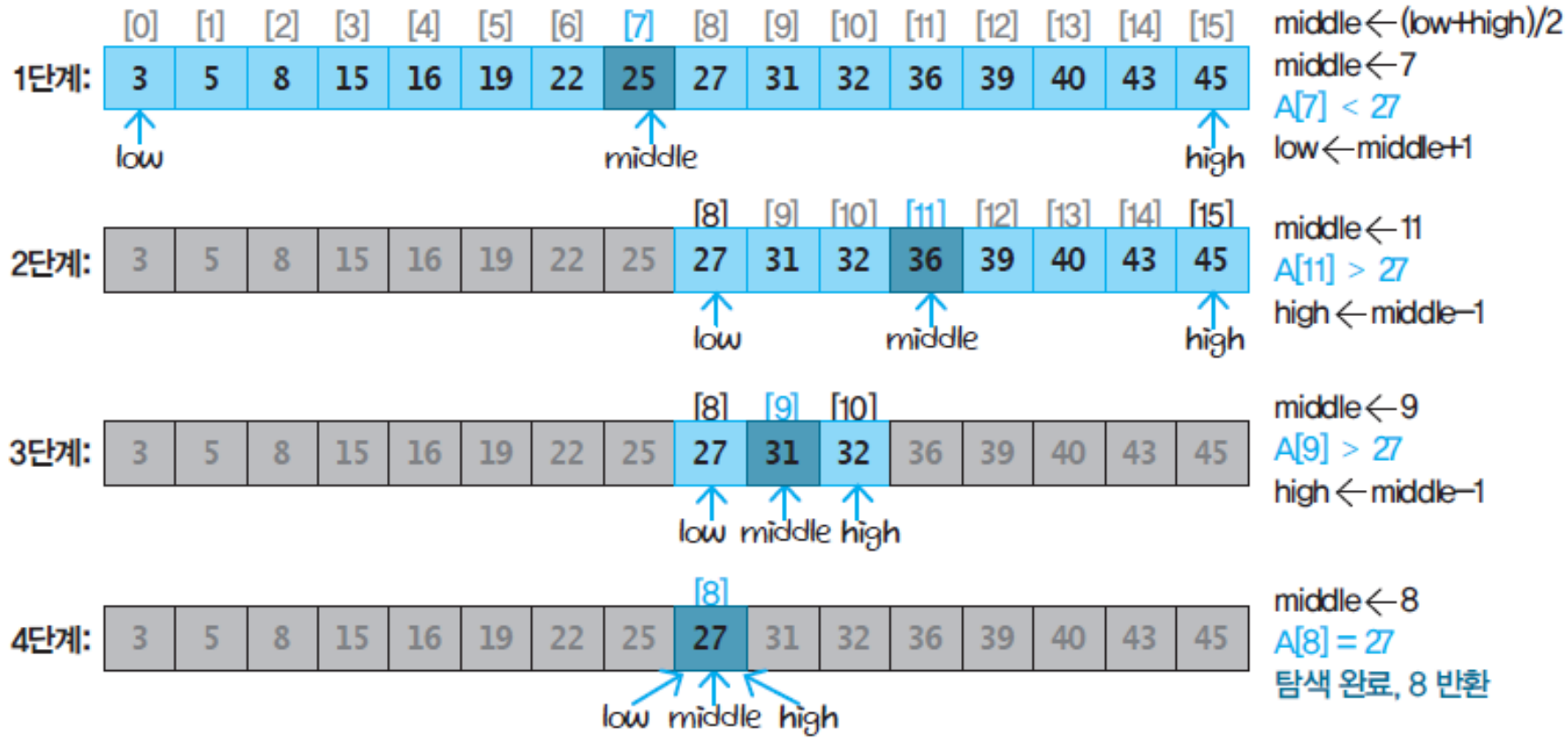
이진 탐색

이진 탐색

- 이진 탐색(Binary Search)이란?
  - 테이블이 정렬되어 있을 때 사용
  - 정렬된 테이블에서 중간 값(middle)을 찾아 찾고자 하는 값과 비교하여 검색 범위를 반으로 줄여가는 방식으로 동작
  - 재귀 / 반복 둘 다 구현 가능
  - 선형 탐색보다 효율적인 탐색 방법
  - 탐색 과정을 트리 구조로 시각화
  - 시간 복잡도: $O(\log n)$
    - ➢ 최선/평균/최악의 경우: O(log n) - 매 단계마다 탐색 범위가 절반으로 줄어듦

이진 탐색 동작 방식

- 이진 탐색 동작 방식



[그림 출처]: 자료구조와 알고리즘 with 파이썬 by 생능북스

이진 탐색 동작 방식

▪ 이진 탐색 동작 방식

① **시작:** 탐색할 리스트가 정렬되어 있다고 가정

② **시작과 끝 설정:** 탐색할 리스트의 시작과 끝 인덱스 설정, 초기에는 전체 리스트 대상

③ **중간값 찾기:** 시작과 끝 인덱스를 사용하여 리스트의 중간 요소의 인덱스 찾기

④ **중간값과 비교:** 중간값과 찾고자 하는 값을 비교

- 중간값==키값 : 탐색 종료, 중간값 인덱스 반환
- 중간값 > 키값 : 탐색 범위를 왼쪽 반으로 줄임
- 중간값 < 키값 : 탐색 범위를 오른쪽 반으로 줄임

⑤ **범위 줄이기:** 값이 있을 가능성이 있는 반쪽을 새로운 탐색 범위로 설정하고 과정 반복

⑥ **탐색 종료:** 탐색 범위 줄어들지 않거나 찾고자 하는 값이 발견되면 탐색을 종료하고 결과를 반환

이진 탐색 알고리즘 : 반복 구조

```
def binary_search_iter(A, target):
    left, right = 0, len(A)-1

    while left <= right:
        mid = (left + right) // 2

        if A[mid] == target :
            return mid              # 찾은 요소의 인덱스 반환
        elif A[mid] < target :
            left = mid + 1          # 오른쪽 반에서 탐색
        else:
            right = mid - 1         # 왼쪽 반에서 탐색

    return -1                   # 요소를 찾지 못한 경우

arr.sort()  # 배열을 정렬한다.
print(f"{msg}#binary_search(idx): {binary_search_iter(arr, target)}" )
```

이진 탐색 알고리즘 : 순환 구조(재귀)

```
def binary_search_recursive(A, target, left, right) :
    if left > right:                    # 기저(종료) 조건: 왼쪽 값이 크면
        return -1

    mid = (left + right) // 2           # 중간 인덱스 계산

    if A[mid] == target:
        return mid
    elif A[mid] < target:
        return binary_search_recursive(A, target, mid + 1, right)
    else:
        return binary_search_recursive(A, target, left, mid - 1)


arr.sort()  # 배열을 정렬한다.
print(f"{msg}#binary_search(idx): {binary_search_iter(arr, target)}" )
```

이진 탐색의 특징

- 탐색 범위

$$2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \cdots \rightarrow 2^1 \rightarrow 2^0 \qquad K = \log_2 n \;\rightarrow\; O(\log n)$$

- 이진 탐색의 특징
  - 반드시 배열이 정렬되어 있어야 사용할 수 있음
  - 탐색은 효율적이지만 삽입/삭제는 효율적이지 않음
    - 테이블이 한번 만들어지면 이후로 변경되지 않고 탐색 연산만 처리한다면 이진 탐색이 최고의 선택
    - 삽입과 삭제가 빈번하다면? → 이진탐색트리 등 다른 방법을 고려

이진 탐색 개선 방법

- 보간 탐색(Interpolation Search)
  - 이진 탐색의 일종
  - 정렬된 배열에서 데이터의 분포가 균등할 때 탐색 이진 탐색보다 더 빠르게 탐색
  - 값의 크기를 고려하여 예상 위치를 추정
  - 핵심 아이디어(예상 위치 계산)

    수치적 보간(선형 보간법)에 기반하여, target이 어느 위치에 있을지를 비례적으로 추정

    ex) 사전에서 'ㄱ'은 앞쪽에서 찾고 'ㅎ'은 뒤쪽에서 탐색

$$pos = low + \frac{(target - arr[low])\ X\ (hight - low)}{arr[high] - arr[low]}$$

이진 탐색 개선 방법 : 보간 탐색

```
def binary_interpolation_search(A, target):
    left, right = 0, len(A) - 1
    while left <= right and A[left] <= target <= A[right]:
        if A[right] == A[left]:       # 분모가 0이 되지 않도록 체크
            if A[left] == target:
                return left
            else:
                return -1
        # 보간 위치 계산
        pos = left + ((target - A[left]) * (right - left)) // (A[right] - A[left])
        if pos < 0 or pos >= len(A):
            return -1  # 예외 처리

        if A[pos] == target:
            return pos
        elif A[pos] < target:
            left = pos + 1
        else:
            right = pos - 1
    return -1  # 찾지 못한 경우
```

실습문제 : 이진 탐색 알고리즘 구현하기

- 이진 탐색 알고리즘을 파이썬으로 구현하세요.
  1) 이진 탐색 알고리즘 (반복 구조)
  2) 이진 탐색 알고리즘 (순환 구조:재귀)
  3) 이진 탐색 알고리즘 (보간 탐색 적용)

해시 탐색

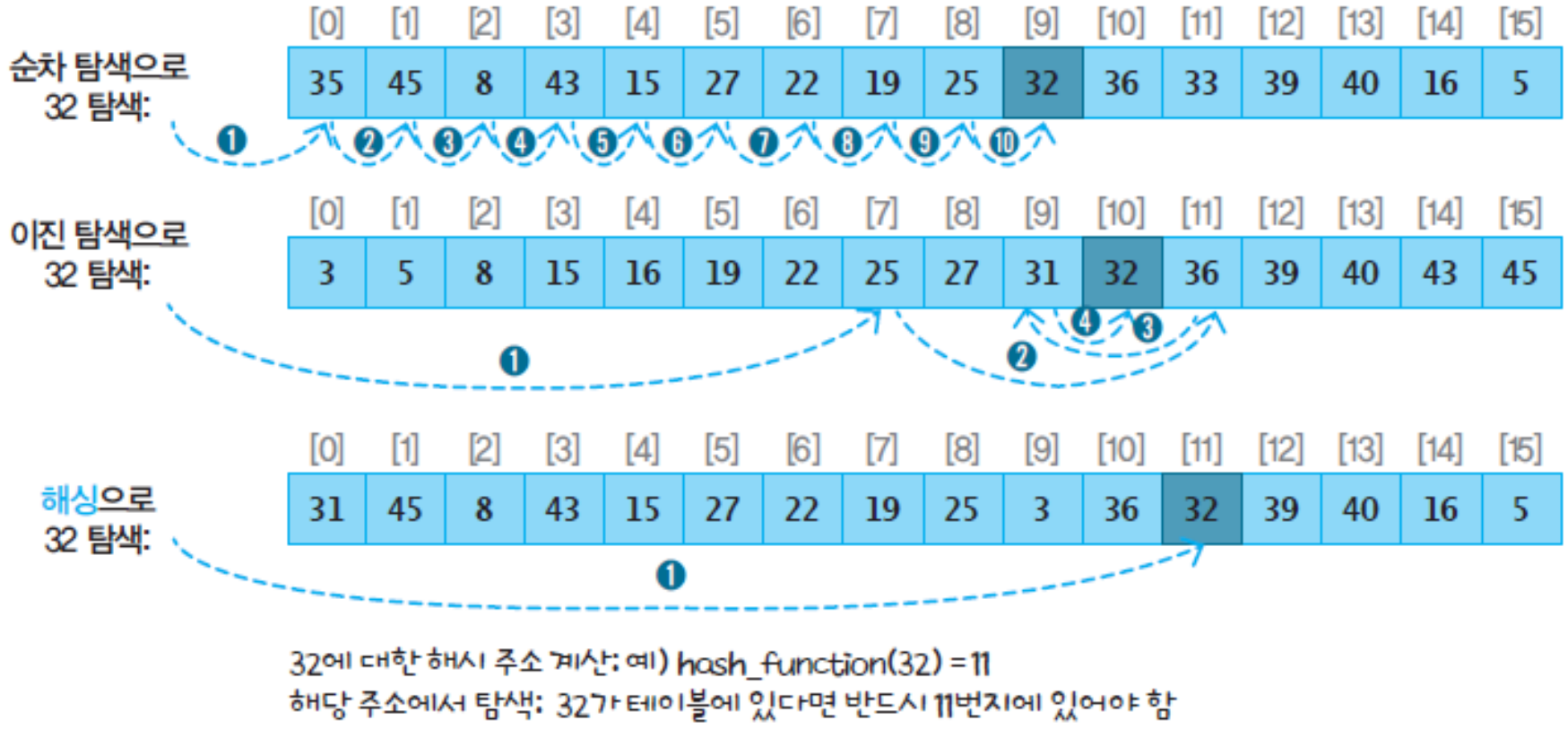
해시 탐색

- 해시 탐색(Hash Search, Hashing)
  - 해시 함수(해시 테이블)를 사용하여 데이터를 저장하고 검색하는 방법
  - 해시 테이블은 키(key)와 값(value)으로 이루어진 데이터를 저장하는 자료구조
  - 동작 방식 :

    key ——▶ [hash function] ——▶ index ——▶ 값 접근 (검색, 삽입, 삭제)
  - 평균 시간 복잡도: O(1)
  - 최악의 경우 O(n)이 될 수 있음
  - 충돌 해결 방법이 중요

해시 탐색

- 순차탐색, 이진탐색과 해싱(Hashing)의 비교



해시 탐색 구성 요소

- 해시 탐색을 위한 조건

구성 요소	설명
해시 함수 (Hash Function)	키를 정수 인덱스로 변환하는 함수 (hash(key) 등)
해시 테이블 (Hash Table)	데이터를 저장하는 배열 (슬롯 또는 버킷으로 구성)
충돌 해결 전략 (Collision Handling)	서로 다른 키가 같은 인덱스를 갖게 되는 경우 처리 방식

해시 탐색 동작 방식

▪ 해시 탐색 동작 방식

① **해시 함수 적용:** 검색할 키를 해시 함수에 입력하여 해시값 생성.

② **해시값을 인덱스로 사용:** 생성된 해시값을 해시 테이블의 인덱스로 사용하여 해당 인덱스에 데이터를 저장하거나 검색

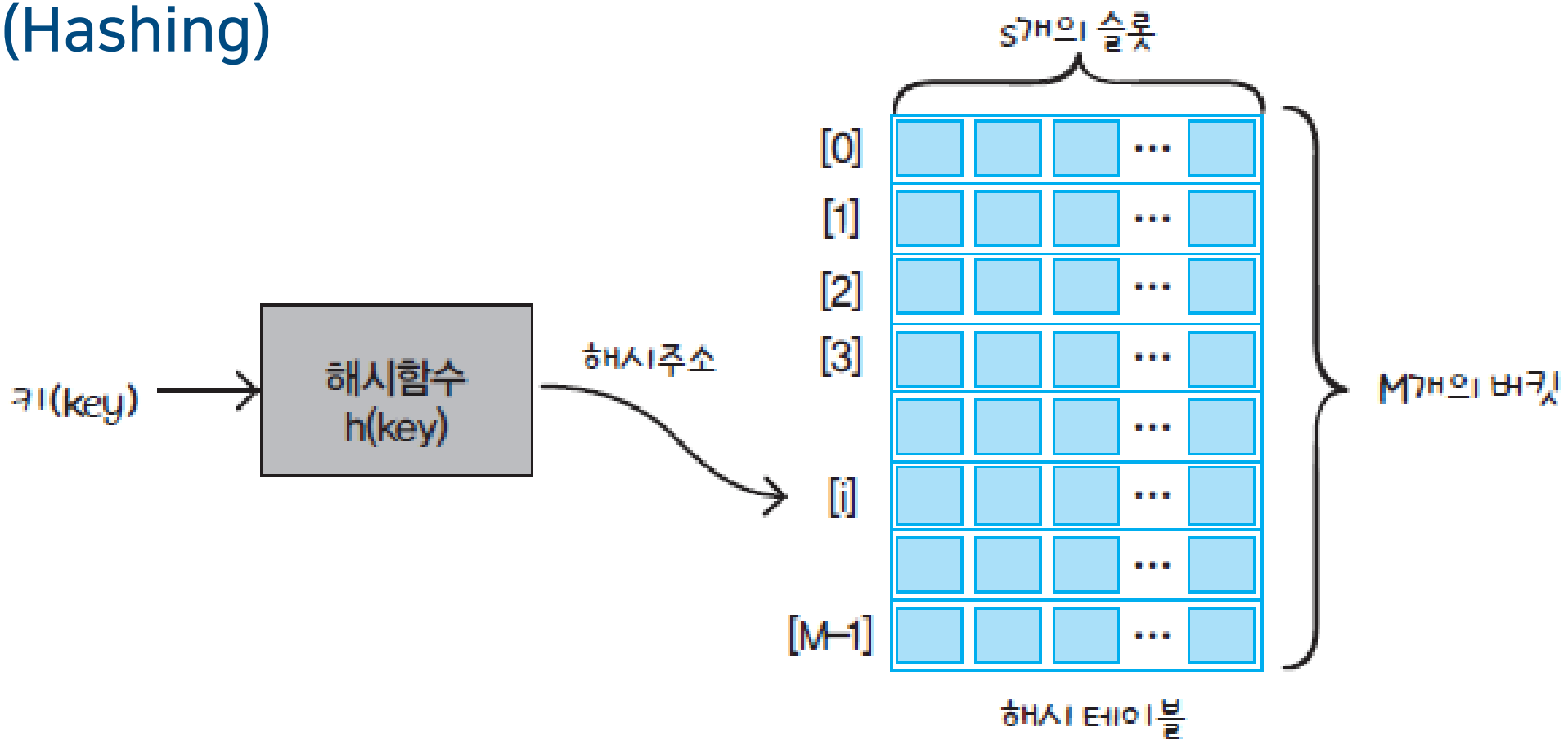
③ **충돌 처리:** 서로 다른 키들이 같은 해시값을 가질 수 있으므로, 이를 충돌이라고 함. 대표적인 충돌 해결 방법인 체이닝(chaining)과 개방 주소법(open addressing) 사용

- **체이닝**: 각 해시값에 대해 연결 리스트를 이용하여 충돌된 데이터들을 저장
- **개방 주소법**: 충돌이 발생한 경우, 다른 빈 버킷을 찾아 데이터를 저장.

  선형 탐색, 이차 탐색, 이중 해싱 등이 사용될 수 있음

해시 탐색 동작 방식

- 해싱(Hashing)



[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

해시 함수

- 해시 함수(Hash Function)
  - 키(key)를 일정한 크기의 정수 인덱스로 변환해주는 함수
    - ex) 키 → 고정된 범위(0 ~ N-1)의 정수 인덱스로 변환

      hash(key) = index ∈ [0, table_size-1]
  - 탐색 효율성과 충돌 최소화에 직접적인 영향을 준다. → 설계와 이해가 매우 중요
  - 좋은 해시 함수의 조건

조건	설명
균등 분포	모든 키가 가능한 인덱스에 고르게 분포해야 함
결정성(Deterministic)	같은 키에 대해 항상 같은 해시값 반환
빠른 계산	연산량이 적어야 함
충돌 최소화	서로 다른 키가 같은 해시값을 갖지 않도록 설계

해시 함수 예 : (정수, 문자열)

1. 정수형 키를 위한 단순 해시 함수

```
def hash_int(key, table_size):
    return key % table_size
```

2. 문자열 키를 위한 아스키 기반 해시 함수

```
def hash_string(key, table_size):
    total = 0
    for char in key:
        total += ord(char)  # 각 문자 → 유니코드 정수값  아스키 합
    return total % table_size
```

```
def _hash(self, key):
    return sum(ord(c) for c in str(key)) % self.size
```

해시 함수 예 : (정수, 문자열)

3. 문자열 해시 – Horner's Rule (문자 순서 고려)

```
def horner_hash(key, table_size, base=31):  ← base(기수) : 보통 소수 사용
    hash_val = 0
    for char in key:
        hash_val = (hash_val * base + ord(char)) % table_size
    return hash_val
```

- Horner's Rule은 실제로 많은 언어에서 문자열 해시 구현에 사용되는 방식
- 파이썬의 hash()도 이와 유사한 방식으로 동작

4. 파이썬 내장 해시 함수 사용(정수,문자열)

```
def builtin_hash(key, table_size):
    return hash(key) % table_size
```

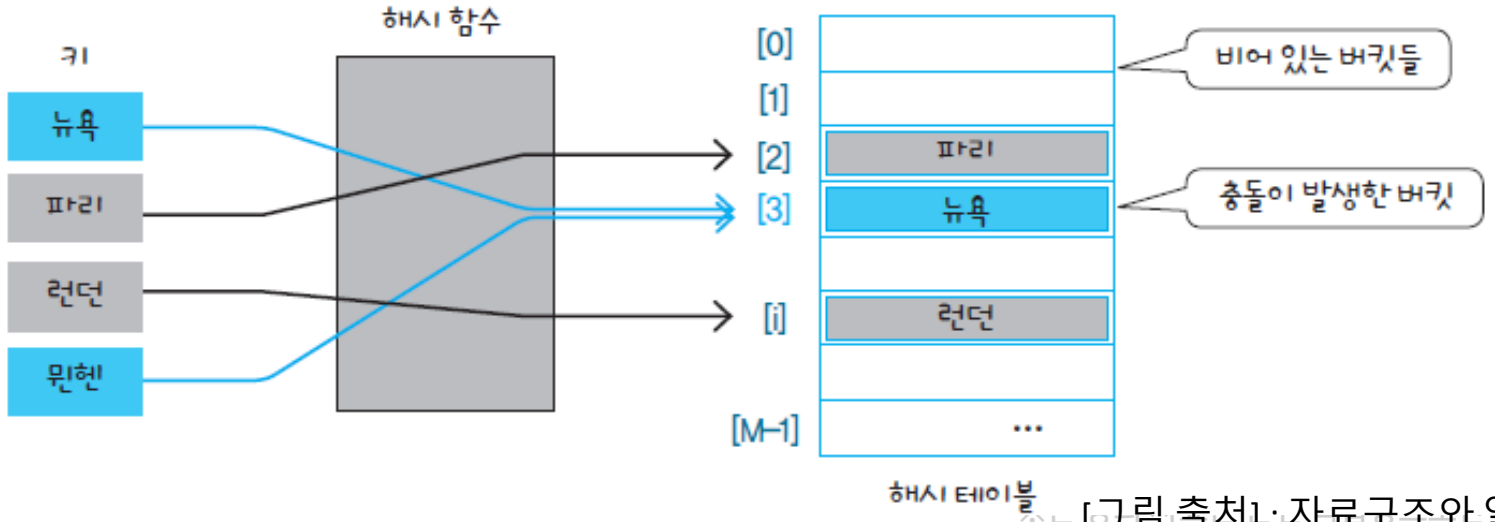
해시 테이블

- 해시 테이블 기본 연산
  - 삽입: *put(key, value)* , 검색: *get(key)*, 삭제: *remove(key)*

- 해시 테이블의 장점
  - 자료의 검색, 읽기, 저장 속도가 빠르다.
  - 자료가 중복되는지 확인하기 쉽다. (집합이 중복을 허용하지 않는다.)
- 해시 테이블의 단점
  - 보통 저장 공간이 더 필요하다.
  - 충돌을 해결할 방법이 필요하다.

해시 테이블 충돌

- 해시 충돌(Collision)
  - 서로 다른 키(key)가 같은 인덱스를 가지는 상황, 이것을 충돌(Collision)이라 함
  - 이상적인 해싱 vs 실제의 해싱
    - 충돌(collision) 발생
    - **오버플로 처리 필요 :** 충돌이 슬롯 수보다 더 많이 발생하는 경우



[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

해시 테이블 충돌

```
class BasicHashTable:

    def __init__(self, size=5):
        self.size = size
        self.table = [None] * size

    def _hash(self, key):
        return sum(ord(c) for c in str(key)) % self.size


    def put(self, key, value):
        index = self._hash(key)

        self.table[index] = (key, value)


    def display(self):
        for i, item in enumerate(self.table):
            print(f"[{i}] {item}")
```

파이썬 hash()함수로 대체 가능

```
index = hash(key) % self.size
```

충돌 발생!!!

충돌 해결 방법 예 :

1. 체이닝(Chaining, 분리 연결법)

- 충돌 시, 같은 해시 인덱스에 여러 데이터를 연결리스트로 연결하여 저장

2. 개방 주소법(Open Addressing)

- 충돌 시, 다음 빈 슬롯을 찾아 저장 (선형 탐색, 이차 탐색, 이중 해싱 등)

구분	개방 주소법 (dict)	체이닝 방식
충돌 처리	테이블 내에서 다른 슬롯을 찾음	연결 리스트로 연결
메모리 사용	단일 배열 내에 저장	별도의 포인터, 구조체 필요
구현 복잡도	상대적으로 복잡함	단순 구조
파이썬에서 사용 여부	✅ 사용됨	❌ 사용되지 않음

충돌 해결 방법 예 : 체이닝(Chaining)

```
class ChainingHashTable:

    def __init__(self, size=5):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return sum(ord(c) for c in str(key)) % self.size

    def put(self, key, value):
        index = self._hash(key)
        if self.table[index] != []:
            print(f"🔁 충돌! {key} → 슬롯 {index} 점유됨, 같은 인덱스에 리스트로 저장")

        self.table[index].append((key, value))

    def display(self):
        for i, bucket in enumerate(self.table):
            print(f"[{i}] {bucket}")
```

연결 리스트를 위한 초기화!

충돌 해결!!!

충돌 해결 방법 예 : 개방 주소법(Open Addressing)

```
class OpenAddressingHashTable:
    def __init__(self, size=5):
        self.size = size
        self.keys = [None] * size
        self.values = [None] * size

    def _hash(self, key):
        return sum(ord(c) for c in str(key)) % self.size

    def put(self, key, value):
        index = self._hash(key)
        start = index
        while self.keys[index] is not None and self.keys[index] != key:
            print(f"🔁 충돌! {key} → 슬롯 {index} 점유됨, 다음으로 이동")
            index = (index + 1) % self.size
            if index == start:
                print("❌ 테이블이 가득 찼습니다.")
                return
        self.keys[index] = key
        self.values[index] = value
```

key, value를 위한 초기화!

충돌 해결!!!

실습문제 : 해시 탐색 알고리즘 구현하기

- 해싱 알고리즘을 파이썬으로 구현하세요.

  포함 기능 : put, get, remove, display

  1) 충돌 방지 방법(체이닝, Chaining) 적용한 코드
  2) 에 충돌 방지 방법(개방 주소법, Open Addressing) 적용한 코드

Q & A

Next Topic

- 탐색 알고리즘 2

Keep learning, see you soon!