The Transactions of the Korea Information Processing Society
TKIPS Vol.13, No.7, pp.311~318
eISSN: 3022-7011
https://doi.org/10.3745/TKIPS.2024.13.7.311

# Metadata-Based Data Structure Analysis to Optimize Search Speed and Memory Efficiency

Kim Se Yeon† · Lim Young Hoon††

## ABSTRACT

As the amount of data increases due to the development of artificial intelligence and the Internet, data management is becoming increasingly important, and the efficient utilization of data retrieval and memory space is crucial. In this study, we investigate how to optimize search speed and memory efficiency by analyzing data structure based on metadata. As a research method, we compared and analyzed the performance of the array, association list, dictionary binary tree, and graph data structures using metadata of photographic images, focusing on temporal and space complexity. Through experimentation, it was confirmed that dictionary data structure performs best in collection speed and graph data structure performs best in search speed when dealing with large-scale image data. We expect the results of this paper to provide practical guidelines for selecting data structures to optimize search speed and memory efficiency for the images data.

Keywords : Metadata, Data Structure, Time Complexity, Search Speed, Memory Efficiency

# 검색 속도와 메모리 효율 최적화를 위한 메타데이터 기반 데이터 구조 분석

김 세 연† · 임 영 훈††

## 요 약

인공지능과 인터넷의 발전으로 인한 데이터의 증가로 데이터 관리의 중요성이 부각되고 있는 상황에서, 데이터 검색과 메모리 공간의 효율적 활용이 매우 중요한 시대가 도래하였다. 본 연구에서는 메타데이터를 기반으로 데이터 구조를 분석하여 검색 속도와 메모리 효율을 최적화하는 방안을 연구한다. 연구방법으로는 사진 이미지의 메타데이터를 활용하여 배열, 연결리스트, 딕셔너리, 이진 트리, 그래프의 데이터 구조에 대한 성능을 시간적, 공간적 복잡도를 중심으로 비교하고 분석하였다. 실험을 통해 대규모의 이미지 데이터를 다루는 상황에서 딕셔너리 구조는 수집 속도에서, 그래프 구조는 검색 속도에서 가장 우수한 성능을 보여주는 것을 확인할 수 있었다. 본 논문의 결과는 이미지 데이터 검색 속도와 메모리 효율을 최적화하기 위한 데이터 구조를 선택하는데 실용적인 가이드라인을 제시할 것으로 기대한다.

키워드 : 메타데이터, 데이터 구조, 시간복잡도, 검색속도, 메모리 효율

## 1. 서 론

인공지능과 인터넷의 발전으로 인해 데이터 양이 급격히 증가함에 따라 데이터 관리에 대한 연구의 필요성이 대두되고 있다. 초거대 데이터 중에서도 원하는 정보를 신속하게 찾기 위해서는 효율적인 검색 작업이 매우 중요하다. 클라우드 서비스나 보조 저장장치의 용량은 증가하고 가격은 저렴해지고 있지만, 필요한 저장용량 또한 함께 증가하고 있기 때문에 저장장치 공간의 수요는 앞으로도 계속해서 증가할 것이다[1]. 대규모 데이터 처리를 위해서는 빠른 검색 속도가 필수적이며, 이는 데이터베이스 시스템, 웹 검색 엔진, 알고리즘 문제 해결 등 다양한 분야에서 중요한 역할을 한다. 또한 메모리 공간의 효율적인 확보 역시 매우 중요하다. 알고리즘의 성능 향상, 사용자 경험 개선, 서비스 품질 향상에 큰 영향을 미치며, 대용량 데이터의 처리와 분석에도 효율적인 검색 알고리즘과 데이터 구조의 선택이 필수적이다. 그러므로 다양한 상황에 적합한 데이터 구조의 선택과 설계가 검색 속도에 미치는 영향을 분석하고, 메모리 사용 효율을 향상시키기 위한 연구가 필요하다.

※ 이 논문은 2024년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 연구되었음(NRF-2022R1I1A1A0107117 독립단편영화를 위한 반응형 웹 기반 OTT 플랫폼 및 인공지능기반 영화 화질 개선기술 개발).
† 준 회 원 : 중앙대학교 소프트웨어학부 학사과정
†† 정 회 원 : 중앙대학교 첨단영상대학원 강사
Manuscript Received : March 28, 2024
First Revision : May 16, 2024
Accepted : June 6, 2024
* Corresponding Author : Lim Young Hoon(secretflasher@gmail.com)

기존의 연구들은 일반적인 텍스트 데이터를 기반으로 동적 데이터 구조나 한 가지의 자료구조를 사용한 성능 향상을 위주로 연구가 되었다[2,3]. 반면 본 연구는 메타데이터(Metadata)를 포함한 이미지의 검색 속도와 메모리 사용량을 분석하기 위해, 다양한 데이터 구조의 적합성을 평가하는 것을 목적으로 한다. 연구방법은 5000장과 100장의 사진 이미지 데이터를 활용하여 배열(Array), 연결 리스트(Linked List), 딕셔너리(Dictionary), 이진 트리(Binary Tree), 그래프(Graph)의 5가지 데이터 구조에 대한 실험을 진행한다. 메타데이터의 태그 정보는 카메라에서 제공하는 공통된 정보(GPS, 촬영일자, 해상도 등)를 활용하여 검색을 수행합니다. 실험은 대상 데이터 구조(Data Structure) 내에서 정보를 검색할 때의 시간 복잡도(Time Complexity)와 공간 복잡도(Space Complexity)를 측정하여 진행한다.

## 2. 이론적 배경

### 2.1 메타데이터

메타데이터(Metadata)는 일반적으로 구조화된 데이터로, 대량의 정보 중에서 찾고자 하는 정보를 효율적으로 검색하기 위해 원시 데이터(Raw Data)를 일정한 규칙에 따라 구조화하거나 표준화한 것을 의미한다[4]. 다시 말해, 메타데이터는 데이터의 세부 내용을 담고 있는 정보이다. 사진 이미지의 메타데이터는 카메라의 정보, 해상도, 크기, 촬영시간 등이 함께 저장된다. 또한, GPS를 사용하여 위치 정보까지 함께 기록할 수 있다. 카메라의 종류에 따라 저장되는 메타데이터는 다양하며, 본 연구에서는 다양한 메타데이터 중 공통된 정보만을 검색하여 실험을 진행했다.

### 2.2 데이터 구조

데이터 구조는 레코드(데이터)를 효율적으로 액세스하고 사용할 수 있도록 구성하고 저장하는 방법이다. 이는 데이터의 논리적 또는 수학적 표현만이 아니라 컴퓨터 프로그램의 구현을 의미한다. Fig. 1은 데이터 구조를 도식화하여 표현한 것이다. 데이터 구조는 크게 선형구조와 비선형구조로 분류할 수 있다. 선형 데이터 구조는 정적 자료구조와 동적 자료구조로 나뉘고, 데이터 요소가 순차적 또는 선형으로 배열되고 각 요소가 이전 및 다음 인접 요소에 연결되는 데이터 구조를 선형 데이터 구조라 하며, 배열(Array), 스택(Stack), 큐(Queue) 등이 있다. 비선형 데이터 구조는 데이터 요소가 순차적으로 또는 선형적으로 배치되지 않은 데이터 구조를 비선형 데이터 구조라고 한다. 트리(Tree)와 그래프(Graph)가 대표적인 구조이다[5]. 본 연구에서는 각 구조의 대표적인 예로 배열, 연결리스트, 딕셔너리, 이진트리, 그래프를 사용한다. 큐(Queue)는 요소가 끝에 추가되고 처음부터 제거되는 선입선출(First In First Out, FIFO) 구조이고, 스택(Stack)은 위에서부터 요소를 추가하고 제거하는 후입선출(Last In First Out, LIFO)의 특성 때문에 본 연구에는 적합하지 않기에 제외하였다. 해당 5가지 데이터 구조의 기본적인 개념은 다음과 같다.

배열은 번호(Index)와 번호에 대응하는 다수의 값들을 하나의 변수명 아래에 저장하는 정적 자료구조이다[6]. 일반적으로 배열에는 같은 종류의 데이터들이 순차적으로 저장되어, 값의 번호가 배열의 시작점으로부터 값이 저장되어있는 상대적인 위치가 된다[7]. Fig. 2는 배열의 구조를 표현하였다.

연결리스트는 각 노드가 데이터와 포인터를 가지고 한 줄로 연결된 방식으로 데이터를 저장하는 동적 자료구조이다. 연결리스트는 데이터를 담고 있는 노드들이 연결되어 있는데, 노드의 포인터가 다음이나 이전 노드와의 연결을 담당한다. 본 연구에서 사용한 연결리스트는 헤더(Head Node)를 가지는 단일 연결리스트로, 각 노드는 다음 노드를 가리키며 마지막 노드는 끝을 나타내기 위해 'None'으로 표현된다[8]. 연결리스트는 노드의 중간에 값을 추가하거나 삭제하는 것이 가능하며, 그 과정이 O(1)만큼 걸린다는 점이 장점이다[9]. Fig. 3은 연결리스트의 구조를 표현하였다.

딕셔너리는 데이터의 Key와 Value를 한 쌍으로 가지는 비선형 자료구조이다[10]. 각 키에 값 하나가 연관되어 있어 연관 배열(Associative Array) 또는 해시(Hash)라고 부르기도 한다. 배열과 다른 점은 순서가 없으며, 키를 사용해 값을 찾는다는 점이다. Table 1은 딕셔너리 구조이다.

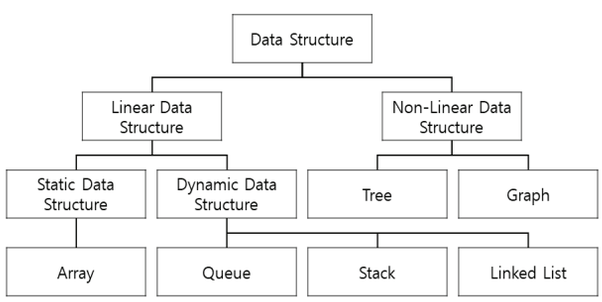


Fig. 1. Data Structure


Fig. 2. Array Structure

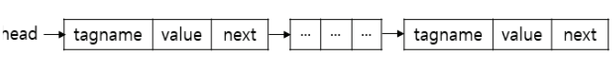


Fig. 3. Linked List Structure

Table 1. Dictionary Structure

| Key | Value |
|---|---|
| tagname | value |
| tagname | value |
| ... | ... |
| tagname | value |

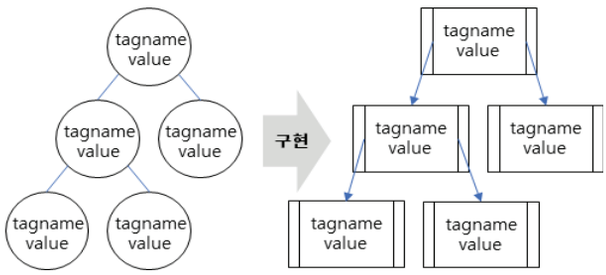


Fig. 4. Binary Tree Structure

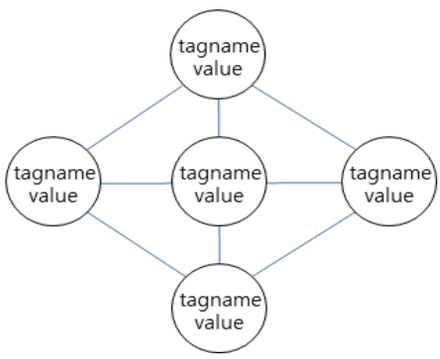


Fig. 5. Graph Structure

이진 트리는 각각의 노드가 최대 두 개의 자식 노드를 가지는 트리 구조로, 자식 노드를 각각 왼쪽 자식 노드와 오른쪽 자식 노드라고 한다[11]. 루트 노드를 기준으로 왼쪽 자식 노드에는 현재 노드의 값보다 작은 값들, 오른쪽 자식 노드에는 현재 노드의 값보다 큰 값들이 저장된다. Fig. 4는 이진 트리의 구조와 해당 구현 방법을 연결리스트로 나타낸 것이다.

그래프는 정점(Vertex)과 간선(Edge)으로 이루어진 데이터 구조이다. 정점은 노드로도 불리며, 간선은 그래프 내에서 두 정점을 연결하는 선이다. G(V,E)로 표시되며, V는 정점의 집합, E는 간선의 집합을 의미한다[12]. 본 논문에서 사용한 그래프는 가중치가 없는 무방향 그래프이다. Fig. 5는 그래프의 구조를 표현하였다.

### 2.3 점근 표기법

점근 표기법은 어떤 함수의 증가 양상을 다른 함수와 비교 표현하는 방법으로 알고리즘의 복잡도를 단순화하거나 무한대에서 간소화시키기 위해 사용된다[13]. 알고리즘의 복잡도를 판단하는 데에는 시간 복잡도와 공간 복잡도라는 두 가지 척도가 있다. 그중에서도 시간 복잡도는 알고리즘 내에서 수행되는 연산의 횟수와 밀접한 관련이 있다[14]. 시간 복잡도는 어떤 알고리즘이 실행되는 데 소요되는 시간을 나타내며, 이를 분석하는데, 대표적으로 사용되는 표기법은 빅 오 표기법(Big-O Notation), 빅 오메가 표기법(Big-$\Omega$ Notation), 빅 세타 표기법(Big-$\Theta$ Notation)이 있다.

빅 오 표기법은 알고리즘의 효율성을 분석할 때 주로 사용되며, 수식은 $f(n) \leq c*g(n)$으로 계산한다. f(n)은 입력 크기 n에 대한 알고리즘의 실행 시간을 나타낸다. 모든 $n(n \geq k)$에 대해 $f(n) \leq c*g(n)$ 인 조건을 만족하는 상수 계수 c와 상수 오프셋 k가 존재하기만 하면 $f(n) = O(g(n))$이다[15-18]. 이때 빅 오 표기법은 상수항과 계수를 무시하고 최고차항만을 고려하여 표기하며, 입력 데이터 크기에 따른 영향만을 주목하고 시간 복잡도에 작은 영향을 주는 것들은 배제하고 표기한다는 것이다. 예를 들어, O(1)은 언제나 일정한 실행 시간을 의미하고, O(n)은 입력 데이터 크기에 비례하여 실행 시간이 증가함을 나타낸다. O(nlogn)은 데이터가 많아질수록 로그의 배수만큼 더 늘어남을 의미하며, n은 입력되는 데이터 수를 의미한다. 따라서 빅 오 표기법은 알고리즘의 점근 상한선을 나타낸다고 할 수 있다. 그에 비해 빅 오메가 표기법은 알고리즘의 점근 하한선을 나타내고 $c*g(n) \leq f(n)$으로 정의한다. 이때 모든 $n(n \geq k)$에 대해 $c*g(n) \leq f(n)$ 인 조건을 만족하는 c와 k가 존재하기만 하면 $f(n) = \Omega(g(n))$이다. 빅 세타 표기법은 $O(f(n)) \cap \Omega(f(n))$이며 정의는 다음과 같다. $c1*g(n) \leq f(n) \leq c2*g(n)$, 모든 $n(n \geq k)$에 대해 $c1g(n) \leq f(n) \leq c2g(n)$인 조건을 만족하는 c1, c2와 k가 존재하기만 하면 $f(n) = \Theta(g(n))$이다[15-18]. 이때 c1과 c2는 f(n)이 g(n)의 상한과 하한으로 얼마나 근접하는지를 나타내는 상수 계수다.

공간 복잡도는 프로그램 실행 후에 필요한 자원 공간의 양을 나타낸다. 총 공간 요구는 고정 공간 요구와 가변 공간 요구의 합으로 나타낼 수 있으며, 이를 수식으로는 $S(P) = c + SP(n)$으로 표기한다[19],[20]. 여기서 c는 고정된 공간 요구를 나타내는 상수를, SP(n)은 입력 크기 n에 대한 알고리즘 P의 가변 공간 요구를 나타낸다. 고정 공간은 입력과 출력 크기, 알고리즘과 무관한 추가적인 공간을 의미하고, 가변 공간은 문제의 특정 인스턴스에 의존하는 구조화된 변수나 함수가 순환 호출할 경우, 요구되는 추가 공간, 즉 동적으로 필요한 공간을 의미한다.

## 3. 실험 방법

본 논문에서는 효율적인 검색 작업을 평가하기 위해, 메타데이터의 추출, 삽입 등의 전처리 과정을 수행하고, 시간 복잡도와 공간 복잡도를 빅 오 표기법, 빅 오메가 표기법, 빅 세타 표기법의 점근 표기법으로 평가한다. 시간 복잡도와 공간 복잡도는 각 자료구조의 선언, 검색, 추출 등 전체 코드를 대상으로 실험하였다.

### 3.1 전처리 과정

본 연구에서는 배열, 연결리스트, 딕셔너리, 이진 트리, 그래프의 5가지의 자료구조를 활용하여 메타데이터 정보를 저장하고 검색하는 알고리즘을 사용했다. Fig. 6은 이미지 파일에서 메타데이터를 읽어와 해당 정보를 각 자료구조에 삽입하는 알고리즘의 핵심 부분을 나타낸 것이다. 구현의 세부사항은 각 자료구조에 따라 약간씩 차이가 있을 수 있다. Exif 데이터는 이미지에 대한 다양한 정보를 포함하는 메타데이터로, getexif 함수를 사용하여 정보를 추출한다. 자료구조에 메타

```
def get_image_metadata(file_path, data_structure):
    img= Image.open(file_path)
    exifdata= img.getexif()
    img.close()

    if exifdatais not None:
        for tagid, value in exifdata.items():
            tagname= TAGS.get(tagid)
            data_structure.append((tagname, value))
return data_structure
```

Fig. 6. Metadata Extraction and Insertion Algorithm

Table 2. Notation

| | |
|---|---|
| exifdata | Exif data from image |
| tagname | the tagname you want to search |
| value | the value you want to search |
| result | an array to store results if the searched tagname and value exist |
| data_ structure | kind of data structure<br>array, linked list, dictionary, binary tree ∈ data_structure |
| ast.literal_ eval | the function that safely evaluates Python literal values |
| search_ value | value to be searched for the corresponding taga search value safely evaluated using the ast.literal_evalfunction |
| metadata_ tagname | tagidof exifdata |
| metadata_ value | value for the corresponding tag |
| append | the function that adds a new element to a list |
| getexif | a PIL function that returns the Exif metadata of an image |
| root | binary tree's root node (starting node) |
| get_image_ metadata | metadata extraction and insertion algorithm |
| new_node | a new metadata node to be added to the graph |

데이터 정보를 추가할 때에는 tagname과 value를 튜플 형태로 저장하기 위해 append 함수를 활용한다.

본 연구에서는 Table 2와 같은 표기를 사용하여 작성하였다.

본 연구에서는 프로그램 실행 중에 사용자로부터 검색하고자 하는 tagname과 value를 입력받은 후, 각 자료구조에 저장된 메타데이터 정보에서 입력값들을 찾는 알고리즘을 적용하고 있다. Fig. 7은 이러한 과정을 처리하기 위한 코드이다. 검색 결과를 저장하기 위해 빈 임시 배열인 “result”를 생성하고, 검색할 값을 파이썬 객체로 변환하기 위해“ast.literal_eval”함수를 사용한다. 그 다음으로 객체로 변환한 “search_value”와 “metadata_value”가 일치하고, “metadata_tagname”과 검색할 “tagname”이 일치하면“append” 함수를 사용하여 검색 결과를 “result” 배열에 추가된다. 이 과정에서 예외가 발생하면 문자열 형태로 비교를 수행한다.

```
def search_metadata_in_data_structure(tagname, value,
data_structure):
    result = []
    for metadata_tagname, metadata_valuein data_structure:
      try:
        search_value= ast.literal_eval(value)
        if metadata_tagname== tagnameand
metadata_value== search_value:
            result.append((metadata_tagname,
metadata_value))
      except(ValueError, SyntaxError):
        if metadata_tagname== tagnameand
str(metadata_value) == value:
          result.append((metadata_tagname, metadata_value))
    return result
```

Fig. 7. Metadata Search Algorithm

```
def insert(root, tagname, value):
    if root is None:
        return TreeNode(tagname, value)
    if tagname< root.tagname:
        root.left= insert(root.left, tagname, value)
    else:
        root.right= insert(root.right, tagname, value)
    return root
```

Fig. 8. Insertion Function in Binary Tree

이진 트리는 나머지 4개의 자료구조들과 달리 값을 저장하는 방법이 다르다. 이진 트리도 Fig. 5와 마찬가지로 getexif 함수를 사용해 메타데이터를 추출하고 tagname과 value를 저장한다. 다만 다른 자료구조들과 다른 부분은 메타데이터를 추가하기 위해 insert 함수를 호출한다는 것이다. 이진 트리의 특성상 이진 트리를 정의하는 클래스가 따로 필요하고, 이 클래스의 왼쪽 트리와 오른쪽 트리에 값을 저장하기 위해 insert 함수를 호출하게 된다. Fig. 8은 insert함수를 나타낸 것이다.

트리의 양쪽에 저장되는 값들은 작은 값부터 큰 값 순서로 저장되며, 루트 노드를 기준으로 왼쪽 서브트리에는 현재 노드의 값보다 작은 값들, 오른쪽 서브트리에는 현재 노드의 값보다 큰 값들이 저장된다. 값을 저장하는 기준은 tagname의 이름순이며 최종적으로 get_image_metadata 함수는 갱신된 root를 반환한다.

그래프는 정점과 간선으로 이루어진다. Fig. 9는 메타데이터 정보를 그래프에 삽입할 때, 간선을 나타내기 위한 코드로 추가적으로 필요한 부분이다. 해당 태그명이 이미 그래프에 존재하고 현재 “value”가 리스트에 없다면 새로운 노드를 생성한다. 그리고 새 노드를 기존 노드들과 연결시킨다. 즉, 같은 “tagname”을 가진 노드들 사이에서 연결이 이루어진다. 그러나 이미 그래프에 존재하는 특정 값과 동일한 새 값이 입력될 경우, 새로운 노드는 추가하지 않고 기존 노드를 사용한다.

```
if tagname in graph:
        found = False
        for node in graph[tagname]:
                if str(node.value) == str(value):
                        found = True
                        break
        if not found:
                new_node = GraphNode(tagname, value)
                for node in graph[tagname]:
                        node.add_neighbor(new_node)
                        new_node.add_neighbor(node)
                graph[tagname].append(new_node)
else:
        new_node = GraphNode(tagname, value)
        graph[tagname] = [new_node]
```

Fig. 9. Extra Insertion Function in Graph

```
def search(root, tagname, value):
    results = []
    if root is None:
        return results
    if tagname< root.tagname:
        return search(root.left, tagname, value)
    elif tagname> root.tagname:
        return search(root.right, tagname, value)
    else:
        if str(value) == str(root.value):
            results.append((root.tagname, root.value))
        results.extend(search(root.left, tagname, value))
        results.extend(search(root.right, tagname, value))
        return results
```

Fig. 10. Search Algorithm in Binary Tree

자료구조에 메타데이터를 저장하는 방식이 각각 다르기 때문에 검색 알고리즘 또한 달라야 한다. 또한 이미 저장할 때부터 값들을 정렬하여 저장하므로 검색 속도도 빠를 수밖에 없다. Fig. 10은 이진 트리 내의 검색 알고리즘을 나타낸 것이다. 주어진 tagname이 현재 노드의 tagname보다 작으면 왼쪽 서브트리에서 검색을 수행하고, 반대로 현재 노드의 tagname보다 크면 오른쪽 서브트리에서 검색을 수행한다. 사용자로부터 받은 입력 값을 해당 서브트리에서 재귀적으로 검색하여 다른 서브트리까지 검색을 수행할 수 있다. 검색하고자 하는 tagname과 value가 존재하는 경우 해당 노드의 정보를 result 배열에 추가한다.

### 3.2 시간 복잡도 분석

Table 3에서 N은 이미지 파일의 수이고, M은 한 이미지의 메타데이터 항목 수다. 모든 이미지 파일에 대해 루프를 돌며 메타데이터를 수집하고 배열, 연결리스트, 딕셔너리, 그래프에 추가하므로 메타데이터 수집에 걸리는 시간 복잡도는 N*M이다. 메타데이터 검색에 걸리는 시간 복잡도도 배열과 연결리스트, 딕셔너리를 순회하며 각 이미지에 대해 선형 검색을 수행하므로 O(N*M)으로 계산된다. 따라서 전체 시간 복잡도를 나타내면 O(N*M) + O(N*M) = O(2N*M)이 된다. 하지만 시간 복잡도를 표기할 때에는 계수와 상수를 모두 떼고 가장 큰 영향을 주는 항만을 고려하므로 결론적으로O(N*M)이 된다. 빅 오 표기법은 최악의 경우를 생각하고 계산한 것이라면 빅 오메가 표기법은 최선의 경우를 생각한다. 만약 모든 이미지에 정확히 하나의 메타데이터만 있다면, 즉 M이 상수라면 $\Omega$(N)으로 표현할 수 있다. 하지만 일반적인 경우, 각 이미지에는 두 개 이상의 메타데이터가 존재할 가능성이 높으므로 $\Omega$(N*M)으로 표현하는 것이 더 낫다. 이진 트리의 경우 get_image_metadata 함수에서 insert 함수를 호출하여 메타데이터를 삽입한다. 검색 속도는 트리 높이인 검색 자료 개수의 로그형에 비례한다[13]. 이진 트리의 높이는 트리의 균형 여부에 따라 결정되는데, 트리가 균형 잡혀 있다면 트리의 높이는 log(N*M)이고 시간 복잡도는 $\Omega$(log(N*M))이 된다. 하지만 트리가 한쪽으로 치우쳐져 있는 경우에는 높이가 N*M이고 O(N*M)의 시간 복잡도를 갖는다. 결론적으로 시간 복잡도의 효율은 N과 M의 크기에 따라 결정되며, 최악의 시간 복잡도는 O(N*M), 최선의 시간 복잡도는 $\Omega$(log(N*M))이다. 빅 세타 표기법은 빅 오 표기법과 빅 오메가 표기법이 같을 때 표기하므로 이진 트리에서는 제외한다. 실제로 코드가 실행되는 시간을 확인하기 위해 time.perf_counter() 함수를 사용하였다. 이 함수는 성능 카운터의 값을 반환한다. 다시 말해, 매우 짧은 기간을 측정하는 데 사용되는 가장 높은 해상도의 시계로, 수면 중에 경과된 시간을 포함하여 전체 시스템에서 작동한다[16].

Table 3. Time Complexity Comparison

| | Big-O | Big-Omega | Big-Seta |
|---|---|---|---|
| Array | O(N*M) | $\Omega$(N*M) | $\theta$(N*M) |
| Linked List | O(N*M) | $\Omega$(N*M) | $\theta$(N*M) |
| Dictionary | O(N*M) | $\Omega$(N*M) | $\theta$(N*M) |
| Binary Tree | O(N*M) | $\Omega$(log(N*M)) | - |
| Graph | O(N*M) | $\Omega$(N*M) | $\theta$(N*M) |

### 3.3 공간 복잡도 분석

공간 복잡도는 보통 빅 오 표기법으로 나타내며 각 자료구조의 공간 복잡도는 Table 4와 같다.

image_array 배열에 사용된 메모리는 각 이미지의 메타데이터를 저장하는 데 사용된 공간이다. 이미지의 수가 N이고,

Table 4. Space Complexity Comparison

| | |
|---|---|
| Array | O(N*M) |
| Linked List | O(N*M) |
| Dictionary | O(K*V) |
| Binary Tree | O(log(N*M)) ~ O(N*M) |
| Graph | O(N*M*p) |

각 이미지의 메타데이터 항목이 M이라고 할 때, 배열의 총 공간 복잡도는 O(N*M)이다. 연결리스트도 배열과 마찬가지로 이미지의 수가 N이고, 각 이미지의 메타데이터 항목이 M일 때, 각 이미지의 메타데이터를 저장하는 데 사용된 공간은 O(N*M)이다. 그러나 연결리스트는 노드에 tagname, value 및 next 값을 저장하기 때문에 노드의 공간까지 고려해야한다. 각 이미지에 대해 하나의 노드가 생성되기 때문에 노드를 저장하는 메모리는 O(N)이다. 따라서 총 공간 복잡도는 O(N*M + N)이며, 간소화하면 O(N*M)이 된다. 딕셔너리는 key와 value를 한 쌍으로 가지고 있기 때문에 각 이미지에 대한 tagname 수와 각 tagname당 value의 수가 중요하다. 따라서 각각을 K와 V라고 할 때 총 공간 복잡도는 O(K*V)이다. 이진 트리의 공간 복잡도는 주로 트리의 높이와 관련이 있다. 각 노드는 tagname, value, left child, right child로 이루어져 있다. 각 이미지에 대해 M개의 메타데이터가 만들어진다고 가정하면, 노드의 수는 N*M이고 이진 트리의 높이는 대략 log(N*M)이다. 그러나 트리가 편향되어 있어 모든 노드가 한쪽으로 치우쳐 있는 최악의 경우에는 O(N*M)이 될 수 있다. 따라서 이진 트리의 공간 복잡도는 O(N*M)에서 O(log(N*M)) 사이이다[21]. 그래프는 가장 결과가 안좋은 경우 N*M개의 노드가 생성될 수 있다. 또한 각 노드는 tagname과 value뿐만 아니라 이웃 노드의 리스트를 저장한다. 평균적으로 각 태그에 대해 p개의 유니크한 값이 있다면, 각 노드의 이웃 리스트의 크기는 최대 p-1이 된다. 따라서 그래프의 공간 복잡도는 O(N*M*(p-1))이며, 간소화하면 O(N*M*p)이다. 결론적으로 이진 트리의 공간 복잡도가 제일 낮고, 그래프의 공간 복잡도가 제일 높은 것으로 나타났다.

## 4. 실험 결과

본 논문에서는 효율적인 검색 작업을 평가하기 위해, 시간 복잡도와 공간 복잡도를 실험하였다. 실제로 코드가 실행되는 시간을 확인하기 위해 time.perf_counter() 함수를 사용하였다. 이 함수는 성능 카운터의 값을 반환한다. 다시 말해, 매우 짧은 기간을 측정하는 데 사용되는 가장 높은 해상도의 시계로, 수면 중에 경과된 시간을 포함하여 전체 시스템에서 작동한다[22]. 실험은 각 자료구조에 대해 5회씩 반복되었으며, 사진 데이터 양은 5000장과 100장으로 나누어 두 가지 조건에서 실험되었다. 실험결과는 Table 5와 같다.

메타데이터의 수집시간은 딕셔너리가 평균 1.042로 가장 짧았고, 메타데이터의 검색 시간은 그래프가 평균 0.001로 가장 짧았다. 이진 트리와 그래프 모두 빠른 검색 시간을 보여주었지만, 그래프의 검색 속도가 더 빠르게 나타났다. 이는 그래프가 tagname별로 edge를 데이터를 연결하고 각 연결된 노드를 효율적으로 탐색할 수 있게 구성되었기 때문이다. 그래프에서는 연결된 노드들을 통해 관련 데이터에 대한 접근 경로가 다양화되고 최적화됨으로써, 필요한 정보를 신속하게 찾을 수 있다. 반면, 이진 트리에서 특정 값을 찾을 때에는 현재 노드 값과 비교하여 작으면 왼쪽 트리로, 크면 오른쪽 트리로 이동한다. 이러한 특성으로 인해 이진 트리의 검색 속도가 매우 빠르다. 그러나 좌우 노드로 이동을 통해서만 데이터 접근이 가능하기 때문에 그래프보다 검색 경로가 길어질 수 있다. 연결리스트의 메타데이터 수집시간이 다른 자료구조보다 오래 걸리는 이유는 리스트에 노드를 추가할 때 발생하는 연산 비용 때문이다. 배열은 메모리 상의 연속된 위치에 데이터를 저장하기 때문에 인덱스로 바로 접근이 가능하다. 또한 딕셔너리는 해시 테이블을 사용하여 데이터를 저장하기 때문에 데이터를 효율적으로 추가할 수 있다. 반면에 연결리스트는 각 노드가 다음 노드의 위치를 기억해야하기 때문에 새로운 데이터를 추가할 때마다 이러한 연결 정보를 갱신해야 한다. 또한 연결리스트는 각 이미지의 메타데이터를 하나의 노드에 저장하는 구조이기 때문에, 한 이미지당 여러 메타데이터가 있는 경우에는 여러 노드가 생성되어야 한다. 이러한 구조는 이미지당 메타데이터가 하나일 때나 메타데이터의 개수가 적을 때는 큰 문제가 되지 않지만, 이미지당 메타데이터의 개수가 많아지면 각 이미지에 대해 노드를 많이 생성해야 하므로 연산 비용이 커진다. 이진 트리도 연결리스트의 구조를 가져다 쓰기 때문에 이미지 수집시간이 오래 걸리는 것이다.

100장의 이미지 데이터의 경우 메타데이터의 수집시간은 그래프가 평균 0.038으로 가장 짧았고, 메타데이터의 검색 시간은 이진 트리와 그래프가 평균 0.000로 가장 짧았다. 유의 깊게 살펴볼 부분은 연결리스트와 이진 트리의 수집시간이다. 데이터의 양이 상대적으로 많을 경우, 자료구조의 데이터 수집 시간이 배열과 딕셔너리에 비해 월등히 오래 걸렸다. 그러나 데이터의 양이 상대적으로 적을 경우, 자료구조들 간의 데이터 수집 시간은 큰 차이가 나지 않는다는 것을 알 수 있었다.

Table 5. Results of 5000 and 100 Images Data(Time Unit: Sec)

| | | 5000 Images Data | | | 100 Images Data | | |
|---|---|---|---|---|---|---|---|
| | | Min | Max | Avg | Min | Max | Avg |
| Array | Collection Time | 1.096 | 1.173 | 1.131 | 0.050 | 0.110 | 0.065 |
| | Search Time | 0.201 | 0.336 | 0.251 | 0.008 | 0.010 | 0.009 |
| Linked List | Collection Time | 16.669 | 17.851 | 17.142 | 0.065 | 0.076 | 0.070 |
| | Search Time | 0.019 | 0.023 | 0.020 | 0.001 | 0.001 | 0.001 |
| Dictionary | Collection Time | 0.870 | 1.170 | 1.042 | 0.049 | 0.066 | 0.056 |
| | Search Time | 0.014 | 0.028 | 0.019 | 0.001 | 0.001 | 0.001 |
| Binary Tree | Collection Time | 12.860 | 29.440 | 23.372 | 0.057 | 0.068 | 0.061 |
| | Search Time | 0.011 | 0.013 | 0.012 | 0.000 | 0.000 | 0.000 |
| Graph | Collection Time | 0.874 | 8.435 | 2.389 | 0.038 | 0.040 | 0.038 |
| | Search Time | 0.001 | 0.002 | 0.001 | 0.000 | 0.000 | 0.000 |

## 5. 결 론

본 연구에서는 사진 이미지의 메타데이터를 추출하고 5가지의 다른 자료구조에 삽입하여, 검색 성능을 비교하였다. 이를 통해 각 자료구조의 장단점과 성능 차이를 명확히 알 수 있다. 시간 복잡도를 살펴봤을 때, 이진트리의 빅 오메가 표기법을 제외한 각 자료구조들 간에는 큰 차이가 없었다. 이를 통해 이진트리의 균형이 잡혀있을 때 가장 적은 시간 복잡도를 가짐을 알 수 있다. 더 정확한 시간을 분석해보기 위해 데이터 수집 시간과 검색 시간을 모두 비교했을 때, 수집시간이 가장 빠른 것은 딕셔너리이며 검색 시간이 가장 빠른 것은 그래프로 나타났다. 특히 그래프는 연결된 노드들을 통한 빠른 검색 경로 제공으로 우수한 검색 효율을 나타냈다. 이는 데이터 수집과 검색 과정에서 발생하는 연산 비용과 메모리 사용 효율이 자료구조 선택에 중요한 고려 사항임을 보여준다. 해당 결과는 대규모 이미지 데이터베이스의 구축 및 관리에 특히 유용할 것으로 보인다. 이진 트리 역시 검색 성능이 매우 좋았으나, 최선의 시간 복잡도와 공간 복잡도를 봤을 때 트리의 균형 유지가 중요함을 시사한다. 다만, 공간 복잡도의 측면에서는 그래프 자료구조가 다른 자료구조들보다 상대적으로 높은 공간 사용을 보였다. 그래프는 연결된 노드 간의 관계를 유지하기 위해 추가적인 메모리를 소비하며, 특히 각 노드는 자신과 연결된 다른 노드들에 대한 참조를 저장해야 한다. 이는 태그 이름별로 데이터를 연결하는 구조에서 각 노드의 간선이 많아질수록 메모리 사용량이 기하급수적으로 증가할 수 있음을 의미한다. 이는 대규모 이미지 데이터베이스를 다룰 때 고려해야 할 중요한 요소다. 본 논문에서는 같은 태그 이름에 값이 다른 노드들을 추가할 때 기존 노드와 새로 생성된 노드를 모두 연결했지만, 더욱 효율적인 노드 연결 알고리즘을 채택하여 공간 사용을 줄이는 방법도 있다. 따라서 처리하고자 하는 데이터의 양에 따라 시간 복잡도와 공간 복잡도 중 더 중요하게 생각하는 부분에 초점을 두고 자료구조를 선택할 필요가 있다. 이러한 연구 결과를 고려해, 메타데이터 기반 데이터 구조의 최적화는 이미지 처리 및 관리 분야에서 활용될 수 있다. 특히 대규모 이미지 데이터베이스나 이미지 검색 엔진에서 빠른 검색 및 효율적인 데이터 관리를 위해 적용될 수 있다. 더 나아가, 이미지를 다루는 여러 산업 분야에서도 응용될 수 있다. 다만 본 연구에서는 주로 이미지 데이터의 메타데이터만을 고려되었다. 향후 연구에서는 더 많은 자료구조와 알고리즘을 포함하여 다양한 실험을 수행함으로써 정교한 비교 분석을 진행할 필요가 있다. 본 연구의 연구 결과는 신속하고 효율적인 이미지 데이터 검색을 위한 실용적인 가이드라인을 제시할 것으로 기대한다.

## References


[1] R. E. Neapolitan, “Foundations of algorithms,” 5th ed., Jones&Bartlett Learning, 2009.

[2] J. He et al., “Dynamic data structures for a direct search algorithm,” *Computational Optimization and Applications*, Vol.23, pp.5-25, 2002.

[3] R. A. Finkel and J. L. Bentley, “Quad trees a data structure for retrieval on composite keys,” *Acta Informatica*, Vol.4, pp.1-9, 1974.

[4] C. S. Kim, “The development of the application program generator based on meta-data,” *The KIPS Transactions: Part D*, Vol.13, No.1, pp.97-102, 2024.

[5] C. Althoff, “The self-taught computer scientist: The Beginner’s guide to data Structures&Algorithms,” 1st ed., Wiley, 2021.

[6] N. Parlante, “Linked list basics,” Stanford CS Education Library 1:25, 2001.

[7] V. P. Parmar, and C. K. Kumbharana, “Comparing linear search and binary search algorithms to search an element from a linear list implemented through static array, dynamic array and linked list,” *International Journal of Computer Applications*, Vol.121, No.3, pp.13-17, 2015.

[8] Parlante, Nick., Linked List Problems [Internet], http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

[9] P. F. Dietz, “Maintaining order in a linked list,” *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pp.122-127, 1982.

[10] W. D. Maurer and T. G. Lewis, “Hash table methods,” *ACM Computing Surveys*, Vol7, No.1, pp.5-19, 1975.

[11] S. Y. Kim, “Estimation of performance for random binary search trees,” *Journal of the Korea Computer Industry Society*, Vol.2, No.2, pp.203-210, 2001.

[12] J. A. Bondy, “Graph theory with applications,” 3rd ed., Elsevier Science Ltd, 1984.

[13] N. G. de Bruijn, “Asymptotic methods in analysis,” 4th ed., Courier Corporation, 1981.

[14] S. Kurgalin and S. Borzunov, “The discrete math workbook: A companion manual using python,” 2nd ed., Springer Nature, pp.357-375, 2020.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Introduction to algorithms,” McGraw-Hill, 2nd ed., 2001.

[16] M. T. Goodrich and R. Tamassia, “Algorithm design: Foundations, analysis, and internet examples,” Wiley, 2002.

[17] J. Kleinberg and E. Tardos. “Algorithm Design,” Addison Wesley, 2006.

[18] A. Levitin, “Introduction to the design & analysis of algorithms,” Addison Wesley, 2nd edition, 2007.

[19] W. Le Yi and G. G. Yin, “Time and space complexity in feedback systems: Recent progress and challenges,” *Proceedings of the 29th Chinese Control Conference*, pp. 6209-6214, 2010.

[20] M. J. North, "A time and space complexity analysis of model integration," *Proceedings of the Winter Simulation Conference*, pp.1644-1651, 2014.
[21] R. Lesuisse, "Some lessons drawn from the history of the binary search algorithm," *The Computer Journal*, Vol.26, No.2, pp.154-163, 1983.
[22] G. Micha and I. Ozsvald, "High performance python: Practical performant programming for humans," 2nd ed., O'Reilly, 2020.

김 세 연

https://orcid.org/0009-0001-0999-9936
e-mail : sy1024@cau.ac.kr
2023년 ~ 현 재 중앙대학교
소프트웨어학부 학사과정
관심분야 : Data Structure, Data Management

임 영 훈

https://orcid.org/0000-0002-4627-263X
e-mail : secretflasher@gmail.com
2007년 상명대학교 미디어학부(학사)
2010년 중앙대학교 첨단영상대학원
첨단영상학과 영상예술학-
영화제작전공(영화제작석사)
2019년 중앙대학교 첨단영상대학원 영상학과
영상공학-예술공학전공(영상학박사)
2012년 ~ 2016년 비트컴퓨터 전임연구원
2019년 ~ 현 재 중앙대학교 첨단영상대학원 강사
2020년 ~ 현 재 중앙대학교 첨단영상대학원 전임연구원
2020년 ~ 현 재 중앙대학교 영상콘텐츠융합연구소 선임연구원
관심분야 : Film Production, VFX, Generative AI & Front-end Design