# 알고리즘 스터디: 2주차 자료구조와 문자열

## 지난 시간 복습 (1주차 요약)

* **개발 환경 및 코드 실행:** C++/Java/Python 개발 도구(예: VS Code, IntelliJ 등) 설정과 효율적인 코드 실행 방법 정리[[1]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=C%2B%2B%3A%20Visual%20Studio%20%EB%98%90%EB%8A%94%20Visual,%ED%8C%8C%EC%9D%BC%20%EB%8B%A8%EC%9C%84%20%EC%8B%A4%ED%96%89%EC%9D%B4%20%EC%89%AC%EC%9A%B4%20%ED%8E%B8%EC%9E%85%EB%8B%88%EB%8B%A4)[[2]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=VS%20Code%EB%A5%BC%20%ED%99%9C%EC%9A%A9%ED%95%9C%20%ED%86%B5%ED%95%A9%20%ED%99%98%EA%B2%BD%3A,%ED%95%98%EB%8A%94%20%EA%B2%83%EC%9D%84%20%EC%9E%90%EB%8F%99%ED%99%94%ED%95%A0%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4). 문제별로 폴더/파일을 관리하고 VS Code Tasks 등 자동화로 편의성 향상.
* **학습 목표 및 방향:** 3개월 단기 집중을 통해 **코딩 테스트 대비 알고리즘 실력 향상**이 궁극적 목표[[3]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=%EC%9D%B4%EB%B2%88%20%EC%8A%A4%ED%84%B0%EB%94%94%EC%9D%98%20%EA%B6%81%EA%B7%B9%EC%A0%81%EC%9D%B8%20%EB%AA%A9%ED%91%9C%EB%8A%94%20%EA%B8%B0%EC%97%85,%EB%AA%A9%ED%91%9C%EC%99%80%20%EB%B0%A9%ED%96%A5%EC%84%B1%EC%9D%84%20%EA%B0%96%EA%B3%A0%20%ED%95%99%EC%8A%B5%ED%95%A0%20%EC%98%88%EC%A0%95%EC%9E%85%EB%8B%88%EB%8B%A4). 다양한 문제 풀이로 문제분석 능력 배양, 세 언어(C++14, Java17, Python3) 문법/특성 숙달, 그래프 탐색 등 중급 주제까지 목표. 다른 사람의 코드 리뷰를 통한 시야 확대도 강조[[4]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=%EB%AC%B8%EC%A0%9C%20%ED%95%B4%EA%B2%B0%20%EB%8A%A5%EB%A0%A5%20%ED%96%A5%EC%83%81%3A%20%EB%8B%A4%EC%96%91%ED%95%9C,%EC%BD%94%EB%93%9C%EB%A5%BC%20%EC%9D%BD%EA%B3%A0%20%EC%9D%B4%ED%95%B4%ED%95%98%EB%8A%94%20%EB%8A%A5%EB%A0%A5%EC%9D%84%20%ED%82%A4%EC%9B%8C%EB%B4%85).
* **알고리즘이란 무엇인가:** 문제를 해결하기 위한 **명확한 절차**나 방법의 집합. 주어진 입력에 대해 원하는 출력을 얻는 단계적 방법으로, 정확성·효율성 등이 요구됨 (예시: 정렬, 탐색 알고리즘 등).
* **알고리즘 문제 풀이 플랫폼 활용:** 주로 백준(BOJ) 및 solved.ac 활용, 필요에 따라 프로그래머스 등 참고[[5]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=%EC%9A%B0%EB%A6%AC%20%EC%8A%A4%ED%84%B0%EB%94%94%EC%9D%98%20%EA%B8%B0%EB%B3%B8%20%EB%AC%B8%EC%A0%9C%ED%92%80%EC%9D%B4%20%ED%94%8C%EB%9E%AB%ED%8F%BC%EC%9D%80,%EB%AC%B8%EC%A0%9C%EB%93%A4%EC%9D%84%20%ED%92%80%EC%96%B4%EB%B3%B4%EB%8A%94%20%EA%B2%83%EC%9D%84%20%EA%B6%8C%EC%9E%A5%EB%93%9C%20%EB%A6%BD%EB%8B%88%EB%8B%A4)[[6]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=%EC%A0%95%EB%A6%AC%3A%20%EC%97%AC%EB%9F%AC%20%ED%94%8C%EB%9E%AB%ED%8F%BC%EC%9D%B4%20%EC%9E%88%EC%A7%80%EB%A7%8C%2C%20%ED%95%B5%EC%8B%AC%EC%9D%80,%EB%8B%A4%EC%96%91%20%ED%95%9C%20%EC%9C%A0%ED%98%95%EC%97%90%20%EC%9D%B5%EC%88%99%ED%95%B4%EC%A7%80%EB%8F%84%EB%A1%9D%20%ED%95%A9%EC%8B%9C%EB%8B%A4). 다양한 플랫폼의 문제를 풀며 꾸준히 연습하는 것이 중요.
* **시간 복잡도와 공간 복잡도:** 알고리즘의 효율성을 평가하는 척도. 입력 크기에 따른 연산 횟수 증가율을 Big-O로 표현하며 (예: O(N), O(N log N), O(2^N) 등), 메모리 사용량 역시 고려. 1주차에서는 **시간/공간 복잡도의 개념과 계산 방법**을 소개하고, O(n^2) vs O(n log n) 사례 비교 등 **성능 차이**를 직접 확인.
* **기본 문법 및 구현 요소:** 각 언어별 자료형, 연산자, 조건문/반복문 문법 리뷰. 예제 코드를 통해 **입출력 처리, 배열/리스트 사용, 함수 구현** 등을 짚어봄. C++에선 <iostream> 입출력과 vector 사용법, Java에선 Scanner/System.out과 ArrayList, Python에선 input()/print()와 리스트 등의 기초 문법 정리를 진행.
* **표준 입출력 심화 및 팁:** 빠른 입력을 위한 C++의 ios::sync\_with\_stdio(false);와 cin.tie(NULL); 사용, Java의 BufferedReader/Writer, Python의 sys.stdin 활용 등 **입출력 최적화 기법**을 소개. 여러 케이스 입력 처리, large input 처리 시 팁 등을 공유.

## 자료구조 기본

알고리즘 효율적인 구현을 위해서는 **자료구조(data structure)**에 대한 이해가 필수입니다. 자료구조란 데이터를 저장하고 조직화하는 방식으로, 선택한 구조에 따라 **연산 속도와 메모리 사용량**이 크게 좌우됩니다[[7]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Sequence%20Containers%EB%8A%94%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%88%9C%EC%B0%A8%EC%A0%81%EC%9C%BC%EB%A1%9C%20%EC%A0%80%EC%9E%A5%ED%95%98%EB%8A%94,%EC%9C%A0%EC%A7%80%ED%95%A0%20%ED%95%84%EC%9A%94%EA%B0%80%20%EC%97%86%EB%8B%A4%EB%A9%B4%20%EC%A2%8B%EC%9D%80%20%EC%84%A0%ED%83%9D%EC%9E%85%EB%8B%88%EB%8B%A4). 이번 섹션에서는 C++ 표준 라이브러리(STL)에 구현된 주요 자료구조들을 중심으로, **Java의 컬렉션 프레임워크와 Python의 내장 구조**와 비교하며 살펴보겠습니다. 각 자료구조의 동작 원리, 주요 연산의 Big-O 복잡도, **사용 사례(왜 중요한지)**를 강조하고, **세 언어로 사용하는 방법**도 함께 소개합니다.

### 시퀀스 컨테이너 (Sequence Containers)

**시퀀스 컨테이너**는 데이터를 순차적(sequence)으로 저장하는 구조로, 구현이 단순하고 고속으로 동작합니다[[8]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Sequence%20Containers). 배열과 유사하게 **원소들이 순서대로 나열**되며, **정렬 유지가 필요 없는 경우**에 유용합니다. C++의 대표적인 시퀀스 컨테이너로 vector, deque, list 등이 있고, Java에서는 ArrayList, LinkedList 등이, Python에서는 리스트(list)나 collections.deque 등이 해당됩니다.

* **벡터(Vector)와 배열(Array):** C++의 std::vector는 **동적 배열**로 런타임에 크기를 변경할 수 있는 배열입니다[[9]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%A9%94%EB%AA%A8%EB%A6%AC%EC%83%81%EC%97%90%EC%84%9C%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EA%B0%80%20%EC%97%B0%EC%86%8D%EC%A0%81%EC%9C%BC%EB%A1%9C%20%EC%9C%84%EC%B9%98%ED%95%98%EB%8A%94%20%EB%B0%B0%EC%97%B4%EC%9E%85%EB%8B%88%EB%8B%A4). 반면 std::array는 고정 크기 배열로 **컴파일 타임에 크기가 결정**되어 이후 변경이 불가능합니다[[10]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Avector%EC%9D%98%20%EA%B2%BD%EC%9A%B0%20%EB%B0%B0%EC%97%B4%EC%9D%98%20%ED%81%AC%EA%B8%B0%EB%A5%BC%20%EB%9F%B0%ED%83%80%EC%9E%84%EC%97%90%EC%84%9C,%EA%B7%B8%EB%A0%87%EC%A7%80%20%EC%95%8A%EB%8B%A4%EB%A9%B4%20std%3A%3Avector%EB%A5%BC%20%EC%84%A0%ED%83%9D%ED%95%98%EC%8B%9C%EB%A9%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4). 기본적으로 크기를 알 수 없거나 변동 가능하다면 vector를, 크기가 고정되어 있다면 array를 사용합니다. std::vector는 내부적으로 **세 개의 포인터로 관리되는 연속 메모리**로 구현됩니다[[11]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Avector%EB%8A%94%20%ED%8F%AC%EC%9D%B8%ED%84%B0%20%EC%84%B8%20%EA%B0%9C%EB%A1%9C%20%EA%B5%AC%ED%98%84%EB%90%98%EC%96%B4,%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4). 첫 번째 포인터는 할당된 메모리 시작 주소, 두 번째는 **다음 삽입 위치**(끝 요소의 다음)를 가리키며, 세 번째는 할당된 메모리의 끝을 가리킵니다[[11]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Avector%EB%8A%94%20%ED%8F%AC%EC%9D%B8%ED%84%B0%20%EC%84%B8%20%EA%B0%9C%EB%A1%9C%20%EA%B5%AC%ED%98%84%EB%90%98%EC%96%B4,%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4). 이렇게 **capacity(용량)**와 **size(사용 중인 크기)**를 관리하여, 동적으로 확장하면서도 배열처럼 임의 접근이 가능합니다.

*벡터의 메모리 구조: 세 포인터로 관리되는 연속 메모리 블록. 빨간 화살표는 다음에 데이터가 삽입될 위치를 나타낸다*[*[12]*](file://file-Edsys4einV4shB8KZQcFA1#:~:text=1,%EA%B0%80%EB%A6%AC%ED%82%A4%EB%8A%94%20%ED%8F%AC%EC%9D%B8%ED%84%B0)[*[13]*](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EB%92%A4%EC%97%90%20%EC%82%BD%EC%9E%85%ED%95%98%EB%A9%B4%20%EB%B9%A8%EA%B0%84%20%ED%8F%AC%EC%9D%B8%ED%84%B0%EA%B0%80,%EC%82%BD%EC%9E%85%EB%90%98%EA%B3%A0%2C%20%EB%B9%A8%EA%B0%84%20%ED%8F%AC%EC%9D%B8%ED%84%B0%EA%B0%80%201%20%EC%A6%9D%EA%B0%80%ED%95%A9%EB%8B%88%EB%8B%A4)*.*

벡터에 새로운 원소를 추가할 때는 주로 **뒤쪽에서 push\_back**으로 삽입하며, 암묵적으로 두 번째 포인터(빨간색, next 위치)가 한 칸 늘어납니다. 벡터의 push\_back 연산은 암호화폐가 *대부분* 평균적으로 O(1) 상수 시간에 수행되지만, 현재 capacity가 가득 찬 상태에서 추가로 삽입하면 **재할당(reallocation)**이 발생하여 더 큰 메모리 블록을 확보하고 기존 요소들을 모두 복사하게 됩니다[[14]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=vector%20%EB%8D%B0%EC%9D%B4%ED%84%B0%20%EC%82%BD%EC%9E%85%20%EC%8B%9C%20%EC%A3%BC%EC%9D%98%EC%82%AC%ED%95%AD%3A). 이 경우 **시간 복잡도가 O(n)**으로 급증할 수 있으므로 주의해야 합니다. 따라서 문제 풀이 시 **필요한 만큼 미리 벡터 용량을 예약**(reserve)해 두면 reallocation 빈도를 줄여 성능을 높일 수 있습니다[[15]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EC%9D%BC%EB%B0%98%EC%A0%81%EC%9C%BC%EB%A1%9C%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EB%92%A4%EC%97%90%20%EC%82%BD%EC%9E%85%ED%95%98%EB%8A%94%20push_back%EC%97%B0%EC%82%B0%EC%9D%80,%EC%A7%81%EC%A0%91%20%ED%99%95%EC%9D%B8%ED%95%B4%20%EB%B3%BC%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4).

**예시:** C++에서 벡터 사용

std::vector<int> v;   
v.push\_back(10); // 원소 추가   
int x = v[0]; // 인덱스로 접근 (0번 원소)   
std::cout << v.size(); // 크기 출력 (결과: 1)

**Java:**

ArrayList<Integer> list = new ArrayList<>();   
list.add(10); // 원소 추가   
int x = list.get(0); // 인덱스로 접근   
System.out.println(list.size()); // 크기 출력 (결과: 1)

**Python:**

arr = []   
arr.append(10) # 원소 추가   
x = arr[0] # 인덱스로 접근   
print(len(arr)) # 크기 출력 (결과: 1)

※ C++에서 std::vector는 **임의 접근**(random access)이 O(1)으로 매우 빠르지만, 중간에 삽입/삭제 시에는 해당 위치 이후 요소들을 한 칸씩 밀거나 당겨야 해서 **O(n)** 시간이 걸립니다. 반면 **맨 끝**에서의 삽입/삭제는 amortized O(1)입니다. Java의 ArrayList와 Python의 리스트도 내부적으로 동적 배열로 구현되어 이와 **동일한 특성**을 지닙니다.

* **데크(Deque, Double-Ended Queue):** C++의 std::deque는 **양쪽 끝에서 O(1)** 시간에 삽입/삭제가 가능한 **양방향 큐**입니다[[16]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Container%20%EC%95%9E%2C%20%EB%92%A4%EC%97%90%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EB%B9%A0%EB%A5%B4%EA%B2%8C,ended%20queue%EC%9E%85%EB%8B%88%EB%8B%A4). 내부 구현이 벡터와 다르게 **여러 개의 고정 크기 버퍼를 연결**하여 전체 시퀀스를 표현하므로, 앞쪽에 새로운 공간을 할당할 수 있어 맨 앞 원소 삽입/삭제도 빠릅니다[[17]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Adeque%EB%8A%94%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%8D%BC%EC%97%90%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC,deque%20%EB%8A%94%20%EC%9C%A0%EC%9A%A9%ED%95%9C%20%EC%84%A0%ED%83%9D%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4). 벡터처럼 공간이 가득 찼다고 한 번에 큰 배열을 재할당하는 대신, 덱은 **필요할 때 새로운 버퍼만 할당**하여 연결하므로 확장이 효율적입니다[[17]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Adeque%EB%8A%94%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%8D%BC%EC%97%90%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC,deque%20%EB%8A%94%20%EC%9C%A0%EC%9A%A9%ED%95%9C%20%EC%84%A0%ED%83%9D%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4).

*Deque의 구조: 여러 버퍼에 걸쳐 요소들을 저장. 그림은 두 개의 버퍼(buffer1, buffer2)에 데이터를 나눈 상황으로, 각 버퍼에 연속된 데이터 블록이 저장되어 있다*[*[18]*](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Alist%20)[*[19]*](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8A%90%EB%A6%BD%EB%8B%88%EB%8B%A4,%EB%B9%A0%EB%A5%B8%20std%3A%3Aforward_list%EA%B0%80%20%EC%A2%8B%EC%9D%80%20%EC%84%A0%ED%83%9D%EC%9D%BC%20%EA%B2%83%EC%9E%85%EB%8B%88%EB%8B%A4)*.*

덱은 **앞뒤 모두 상수 시간 연산**이 가능하여, 컨테이너의 양 끝에서 자주 삽입/삭제가 일어나는 경우 적합합니다[[18]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Alist%20). 예를 들어 **슬라이딩 윈도우** 문제에서 일정 범위의 요소를 큐처럼 다루되 양쪽에서 효율적으로 조작할 때 유용합니다. 다만 덱의 내부 요소들은 메모리상 **연속적이지 않기 때문에** 벡터처럼 C 배열과 호환되는 포인터를 얻을 수 없으며, 캐시 지역성 측면에서도 벡터보다 불리할 수 있습니다[[20]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%ED%95%B4%EB%8B%B9%20%EB%82%B4%EC%9A%A9%EB%A7%8C%20%EB%B3%B4%EB%A9%B4%20deque%EA%B0%80%20vector%EB%B3%B4%EB%8B%A4,%EC%83%81%ED%99%A9%EC%97%90%EC%84%9C%EB%8A%94%20deque%EA%B0%80%20%EB%B6%88%EB%A6%AC%ED%95%9C%20%EC%A0%90%EC%9D%B4%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4).

*새 버퍼 추가 후 Deque 상태: 버퍼 2까지 차있던 덱에 새로운 값 7을 맨 뒤에 push\_back하면, 세 번째 버퍼(buffer3)가 할당되고 연결된다*[*[19]*](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8A%90%EB%A6%BD%EB%8B%88%EB%8B%A4,%EB%B9%A0%EB%A5%B8%20std%3A%3Aforward_list%EA%B0%80%20%EC%A2%8B%EC%9D%80%20%EC%84%A0%ED%83%9D%EC%9D%BC%20%EA%B2%83%EC%9E%85%EB%8B%88%EB%8B%A4)*. 덱은 앞뒤로 버퍼를 추가하여 확장하므로, 처음부터 메모리 연속성을 요구하지 않는다.*

**예시:** C++에서 데크 사용

std::deque<int> dq;   
dq.push\_front(1); // 앞에 삽입   
dq.push\_back(2); // 뒤에 삽입   
dq.pop\_front(); // 앞에서 제거

**Java:** (Java는 Deque 인터페이스 활용)

Deque<Integer> dq = new ArrayDeque<>();   
dq.addFirst(1); // 앞에 삽입   
dq.addLast(2); // 뒤에 삽입   
dq.pollFirst(); // 앞에서 제거

**Python:** (collections.deque 사용)

from collections import deque   
dq = deque()   
dq.appendleft(1) # 앞에 삽입   
dq.append(2) # 뒤에 삽입   
dq.popleft() # 앞에서 제거

* **리스트(List) 및 연결 리스트:** C++의 std::list는 **이중 연결 리스트**(doubly linked list) 구현체이며, std::forward\_list는 **단일 연결 리스트**(singly linked list)입니다[[21]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=linked%20list%EC%9E%85%EB%8B%88%EB%8B%A4,1%29%EC%97%90%20%EC%A0%91%EA%B7%BC%29%EB%8A%94%20%EB%B6%88%EA%B0%80%EB%8A%A5%ED%95%A9%EB%8B%88%EB%8B%A4). 리스트는 **포인터로 노드들을 연결**하여 구성되므로, **어느 위치에서든 노드 삽입/삭제가 포인터 조작으로 O(1)**에 가능합니다[[21]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=linked%20list%EC%9E%85%EB%8B%88%EB%8B%A4,1%29%EC%97%90%20%EC%A0%91%EA%B7%BC%29%EB%8A%94%20%EB%B6%88%EA%B0%80%EB%8A%A5%ED%95%A9%EB%8B%88%EB%8B%A4). 단, 배열처럼 인덱스로 바로 접근하는 random access는 지원하지 않아 **특정 위치를 찾는 데 O(n)**이 소요됩니다[[22]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=linked%20list%EC%9E%85%EB%8B%88%EB%8B%A4,1%29%EC%97%90%20%EC%A0%91%EA%B7%BC%29%EB%8A%94%20%EB%B6%88%EA%B0%80%EB%8A%A5%ED%95%A9%EB%8B%88%EB%8B%A4). 이로 인해 **순차적으로 탐색**하거나 중간에 데이터 추가/삭제가 빈번할 때 유용하지만, **탐색 작업이 많다면 비효율적**입니다. 또한 포인터를 많이 사용하므로 **메모리 오버헤드**와 **캐시 비효율**이 있다는 점도 고려해야 합니다.

std::forward\_list는 단일 연결 리스트로 구현되어 이중 리스트보다 구조가 가벼우나, prev 포인터가 없어 **뒤쪽으로의 이동이 불편**하고 제공 기능이 일부 제한됩니다[[23]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Alist%EB%8A%94%20doubly,%EB%B9%A0%EB%A5%B8%20std%3A%3Aforward_list%EA%B0%80%20%EC%A2%8B%EC%9D%80%20%EC%84%A0%ED%83%9D%EC%9D%BC%20%EA%B2%83%EC%9E%85%EB%8B%88%EB%8B%A4). 예를 들어 forward\_list는 push\_front는 빠르지만 push\_back이 느리며, 노드 자체를 삭제하는 함수가 제공되지 않아 이전 노드를 통해 다음 노드를 지워야 합니다. 그 대신 **메모리 사용이 적고 연산이 약간 빠르므로**, 양방향 접근이 불필요한 경우 forward\_list가 더 효율적일 수 있습니다[[23]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Alist%EB%8A%94%20doubly,%EB%B9%A0%EB%A5%B8%20std%3A%3Aforward_list%EA%B0%80%20%EC%A2%8B%EC%9D%80%20%EC%84%A0%ED%83%9D%EC%9D%BC%20%EA%B2%83%EC%9E%85%EB%8B%88%EB%8B%A4).

자바의 LinkedList 클래스는 이중 연결 리스트로 구현되어 있어 C++ list와 유사한 동작을 합니다. 파이썬은 별도의 연결 리스트 타입을 제공하지 않지만, 일반 리스트로 대부분의 기능을 커버합니다. 연결 리스트는 알고리즘 문제에서는 직접 구현보다는 **스택/큐 등으로 활용**되거나, 트리/그래프의 **인접 리스트 표현** 등에 응용됩니다. 다만, 인터뷰나 코테에서 **직접 연결 리스트를 구현**하거나 노드 포인터를 조작하는 문제도 출제될 수 있어, 개념을 이해하고 간단한 구현은 할 수 있도록 하는 것이 좋습니다.

*배열과 연결 리스트의 특성 비교*[*[24]*](file://file-JBLUSFLNtvecfTsT2Xsuci#:~:text=Array%EC%99%80%20%EB%B9%84%EA%B5%90%20%EB%B0%8F%20%EA%B3%84%EC%82%B0%EB%B3%B5%EC%9E%A1%EB%8F%84)[*[25]*](file://file-JBLUSFLNtvecfTsT2Xsuci#:~:text=Insert%20element%20at%20here)*. 배열은 인덱스로 즉시 접근 가능한 반면 삽입/삭제 비용이 크고, 연결 리스트는 순차 접근만 가능하지만 노드 삽입/삭제가 빠르다.*

### 연관 컨테이너 (Associative Containers) – 정렬맵/셋

**연관 컨테이너**는 데이터를 키(key)에 대한 **정렬된 상태로 저장**하는 자료구조입니다. 내부적으로 **레드-블랙 트리(Red-Black Tree)**와 같은 **자가 균형 이진 탐색 트리**로 구현되어, 원소 삽입/삭제/탐색이 모두 **O(log n)**에 동작합니다[[26]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Associative%20Containers)[[27]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EC%9E%90%EA%B0%80%20%EA%B7%A0%ED%98%95%20%EC%9D%B4%EC%A7%84%20%ED%83%90%EC%83%89%20%ED%8A%B8%EB%A6%AC%28self,%EA%B2%BD%EC%9A%B0%EC%97%90%EB%8F%84%20%EC%9A%B0%EC%88%98%ED%95%9C%20%EC%8B%A4%ED%96%89%20%EC%8B%9C%EA%B0%84%EC%9D%84%20%EA%B0%80%EC%A7%91%EB%8B%88%EB%8B%A4). 정렬 상태를 유지한 채로 데이터에 접근할 수 있다는 점이 강력하지만, 상수가 커서 작은 데이터셋에서는 오히려 단순한 배열보다 느릴 수 있고, 트리 노드들이 분산되어 있어 메모리 사용량이 많다는 단점이 있습니다[[26]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Associative%20Containers).

* **레드-블랙 트리 개요:** 레드-블랙 트리는 삽입/삭제 시 스스로 **균형을 잡는 이진 탐색 트리**로, 최악의 경우에도 O(log n)의 성능을 보장합니다[[28]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Red). 각 노드가 빨간색 또는 검은색으로 색칠되어 특정 균형 조건을 유지함으로써, **트리의 높이가 균형적으로 유지**됩니다. 이러한 트리 구조 덕분에, 데이터가 항상 **정렬된 상태**로 삽입/관리되며, 중위 순회(inorder traversal)를 하면 오름차순으로 데이터를 얻을 수 있습니다.
* **집합(Set)과 맵(Map):** C++의 std::set은 **값 그 자체를 키로** 하여 저장하는 **중복 없는(sorted) 집합**이며, std::map은 (키, 값) 쌍을 저장하는 **연관 배열**입니다[[29]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Aset%20). 예를 들어 set<int>에 정수를 넣으면 자동으로 오름차순 정렬되어 저장되고, map<string,int>에 넣으면 키인 문자열 기준으로 정렬되어 (문자열 사전순) (키, 값)이 관리됩니다. **중복 허용이 필요**하면 std::multiset과 std::multimap을 사용합니다. 동일 키의 여러 원소를 저장할 수 있지만, 대신 **동일 키 원소를 모두 탐색하거나 삭제할 때 비용 증가**를 감안해야 합니다. (예: multiset.erase(key)는 해당 키의 모든 원소를 지우므로 단일 원소 삭제보다 시간이 더 걸림). Java에서는 TreeSet, TreeMap이 이와 대응되며, 내부 구현도 이진균형트리(RB트리)로 동작합니다.

연관 컨테이너는 **항상 정렬된 데이터를 필요로 할 때** 유용합니다. 예를 들어 **우선순위가 아닌 정렬된 순회**가 필요한 경우나, **중간 값 계산, 범위 쿼리** 등에 활용할 수 있습니다. 다만 코딩 테스트에서는 정렬이 필요없다면 굳이 O(log n) 구조를 쓰기보다 **해시 구조(평균 O(1))**를 사용하는 편이 더 효율적인 경우가 많습니다.

**예시:** C++에서 Set/Map 사용

std::set<int> s;   
s.insert(5);   
s.insert(1); // 자동 정렬: {1,5}   
if(s.count(1)) { ... } // 원소 존재 확인   
std::map<string,int> mp;   
mp["apple"] = 3; // 삽입 (키 "apple"->값 3)

**Java:**

TreeSet<Integer> s = new TreeSet<>();   
s.add(5);   
s.add(1); // 자동 정렬: {1,5}   
if(s.contains(1)) { ... }   
TreeMap<String,Integer> mp = new TreeMap<>();   
mp.put("apple", 3); // 삽입 ("apple"->3)

**Python:** (내장에 TreeMap은 없으며 일반적으로 정렬 필요 시 정렬 리스트 사용)

s = set() # 파이썬 set은 해시(set과 내용 다름)   
# 대신 sorted 함수나 bisect 모듈로 정렬 관리

※ Python에는 기본 정렬맵/셋 자료구조가 없어 필요 시 bisect 모듈로 정렬된 리스트에 이분 검색으로 삽입하거나, sortedcontainers 같은 외부 라이브러리를 사용합니다. 기본 set/dict는 해시 기반이므로 삽입 순서와는 별개로 내부 순서는 일정하지 않습니다.

### 해시 컨테이너 (Unordered Associative Containers) – 해시셋/맵

**Unordered 컨테이너**는 데이터의 **해시값(hash value)**을 사용하여 **평균적으로 O(1)** 복잡도로 검색/삽입/삭제를 제공하는 구조입니다[[30]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%ED%95%B4%EC%8B%9C%EA%B0%92%EC%9D%84%20%EC%82%AC%EC%9A%A9%ED%95%B4%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%A0%80%EC%9E%A5%ED%95%98%EB%8A%94%20%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0%EC%9E%85%EB%8B%88%EB%8B%A4,Associated%20Container%EB%A5%BC%20%EC%82%AC%EC%9A%A9%ED%95%98%EB%8A%94%20%EA%B2%8C%20%EC%A2%8B%EC%8A%B5%EB%8B%88%EB%8B%A4). C++의 std::unordered\_set, std::unordered\_map 등이 이에 해당하며, 자바의 HashSet, HashMap, 파이썬의 set, dict도 모두 **해시 테이블**을 기반으로 동작합니다. 해시 컨테이너는 대체로 정렬 컨테이너보다 성능이 우수하지만, **데이터가 자동 정렬되지 않는다는 점**을 유의해야 합니다[[30]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%ED%95%B4%EC%8B%9C%EA%B0%92%EC%9D%84%20%EC%82%AC%EC%9A%A9%ED%95%B4%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%A0%80%EC%9E%A5%ED%95%98%EB%8A%94%20%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0%EC%9E%85%EB%8B%88%EB%8B%A4,Associated%20Container%EB%A5%BC%20%EC%82%AC%EC%9A%A9%ED%95%98%EB%8A%94%20%EA%B2%8C%20%EC%A2%8B%EC%8A%B5%EB%8B%88%EB%8B%A4). 또한 **해시 충돌**의 경우 체인 처리를 위해 **연결 리스트 등의 구조**를 사용하므로, **최악의 경우 O(n)**까지 성능이 떨어질 수 있습니다[[31]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%82%B7%EC%97%90%20%EB%82%98%EB%88%A0%EC%84%9C,n%29%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4). 대부분의 상황에서는 충돌이 심하지 않아 상수 시간에 가깝게 동작하지만, 최악 시나리오(특히 키가 편향되게 작동하거나 악의적 입력)에서는 성능 열화가 생길 수 있음을 알아둬야 합니다[[31]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%82%B7%EC%97%90%20%EB%82%98%EB%88%A0%EC%84%9C,n%29%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4).

*해시맵(unordered\_map)의 내부 구조 예시: 해시 함수로 계산된 값에 따라 여러* *버킷(bucket)* *중 하나에 (키, 값) 쌍을 저장한다. 서로 다른 키가 같은 버킷에 할당될 수 있는데*[*[32]*](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%82%B7%EC%97%90%20%EB%82%98%EB%88%A0%EC%84%9C,n%29%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4)*, 이러한* *해시 충돌이 발생하면 해당 버킷 내에서 연결 리스트 등으로 여러 데이터를 관리한다*[*[32]*](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%82%B7%EC%97%90%20%EB%82%98%EB%88%A0%EC%84%9C,n%29%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4)*.*

C++의 unordered\_map은 기본 해시 함수로 std::hash를 사용하며, int, double, string 같은 기본 타입에 대한 해시가 미리 정의되어 있습니다[[33]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EA%B8%B0%EB%B3%B8%EC%9C%BC%EB%A1%9C%20%EC%82%AC%EC%9A%A9%ED%95%98%EB%8A%94%20hash%20function%EC%9D%80%20std%3A%3Ahash%EC%9D%B4%EA%B3%A0%2C,primitive%20type%EA%B3%BC%20std%3A%3Astring%20%EB%93%B1%EC%9D%B4%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4). Java의 각 객체는 hashCode() 메서드를 통해 해시값을 제공하며, HashMap은 이것을 활용합니다. Python의 dict/set은 객체의 \_\_hash\_\_()를 사용하며, 문자열과 숫자는 각각 해시값이 정의되어 있습니다.

일반적으로 **빈도 계산, 존재 여부 체크, 데이터 분류** 등 **순서와 상관없이 빠른 탐색**이 요구되는 상황에서 해시 자료구조가 널리 쓰입니다. 예를 들어 **단어 빈도수 세기**, **이미 등장한 원소 체크**, **두 개 집합의 교집합/합집합 연산** 등에 유용합니다. 다만 데이터가 매우 크고 해시 충돌이 우려될 때는, 혹은 정렬이 필요할 때는 앞서 언급한 트리 기반 구조를 고려해야 합니다[[30]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%ED%95%B4%EC%8B%9C%EA%B0%92%EC%9D%84%20%EC%82%AC%EC%9A%A9%ED%95%B4%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%A0%80%EC%9E%A5%ED%95%98%EB%8A%94%20%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0%EC%9E%85%EB%8B%88%EB%8B%A4,Associated%20Container%EB%A5%BC%20%EC%82%AC%EC%9A%A9%ED%95%98%EB%8A%94%20%EA%B2%8C%20%EC%A2%8B%EC%8A%B5%EB%8B%88%EB%8B%A4).

**예시:** C++에서 Unordered Map/Set 사용

std::unordered\_set<int> us;   
us.insert(3);   
if(us.find(3) != us.end()) { ... } // 원소 존재 확인   
std::unordered\_map<string,int> ump;   
ump["one"] = 1; // (키 "one", 값 1) 저장

**Java:**

HashSet<Integer> us = new HashSet<>();   
us.add(3);   
if(us.contains(3)) { ... }   
HashMap<String,Integer> ump = new HashMap<>();   
ump.put("one", 1); // (키 "one", 값 1) 저장

**Python:**

us = set()   
us.add(3)   
if 3 in us: ...   
ump = {}   
ump["one"] = 1 # (키 "one", 값 1) 저장

### 컨테이너 어댑터 (Container Adaptors) – 스택, 큐, 우선순위 큐

**컨테이너 어댑터**는 말 그대로 기존 컨테이너에 기능적인 **인터페이스를 입힌 래퍼**들입니다[[34]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Container%20Adaptors). STL에서는 stack, queue, priority\_queue 세 가지가 제공되며, 실제 내부 구현체로 기본 deque를 사용하지만, 원한다면 다른 컨테이너로 변경도 가능합니다[[35]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Astack%2C%20std%3A%3Aqueue). 예를 들어 std::stack은 deque를 기본으로 하되 vector나 list로도 변경 가능하며, 인터페이스는 LIFO 스택 연산(push/pop/top 등)만 노출합니다. 자바에서는 Stack 클래스(예전부터 존재)나 Deque 인터페이스로 스택/큐를 구현하고, PriorityQueue 클래스로 우선순위 큐를 제공합니다. 파이썬은 전용 Stack 자료형은 없지만 리스트나 deque로 스택/큐 기능을 수행하고, heapq 모듈로 최소 힙(min-heap)을 이용하여 우선순위 큐를 구현합니다.

* **스택(Stack)** – **후입선출(LIFO)** 자료구조의 대표격입니다. C++ std::stack은 push, pop, top 등의 연산을 제공하며, Java는 java.util.Stack (또는 권장되는 Deque 활용), Python은 리스트의 append, pop을 사용하여 스택처럼 쓰거나 collections.deque를 이용합니다. **함수 호출의** 호출 스택(메모리 스택)**,** 수식의 괄호 검사**,** DFS(깊이 우선 탐색) **등에 활용되며, 재귀 알고리즘은 내부적으로 함수 호출 스택을 사용하므로 스택 구조와 밀접합니다. 또한** 중위표기법->후위표기법 변환**이나** 후위표기식 계산\*\* 등에서도 스택이 필수적으로 쓰입니다.

**예시:** 세 언어에서의 스택 사용 비교

std::stack<int> st;   
st.push(5); // 스택에 5 추가   
int a = st.top(); // top 조회 (a=5)   
st.pop(); // top 제거

Stack<Integer> st = new Stack<>();   
st.push(5); // 스택에 5 추가   
int a = st.peek(); // top 조회 (a=5)   
st.pop(); // top 제거

st = []   
st.append(5) # 스택에 5 추가   
a = st[-1] # top 조회 (a=5)   
st.pop() # top 제거

* **큐(Queue)** – **선입선출(FIFO)** 구조로, C++ std::queue는 push, pop, front, back 등을 제공합니다. Java에서는 Queue 인터페이스(LinkedList나 ArrayDeque 구현 사용)로 add/offer, poll, peek 등을 사용하고, Python에서는 collections.deque를 활용하여 append(뒤로 넣기), popleft(앞에서 꺼내기)로 큐 동작을 구현합니다. 큐는 **너비 우선 탐색(BFS)**에서 방문 대기열로 쓰이며, **운영체제의 작업 스케줄링**, **이벤트 처리 루프**(event loop) 등 **선입선출 대기열**이 필요한 상황에서 폭넓게 활용됩니다.
* **우선순위 큐(Priority Queue)** – **최대/최소 힙(heap)** 자료구조로 구현되며, **가장 우선순위 높은 요소를 빠르게 추출**할 수 있는 구조입니다. C++ std::priority\_queue는 기본 max-heap으로 동작 (가장 큰 값이 top)하며, push, pop, top 연산이 모두 **O(log n)**에 실행됩니다[[36]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Apriority_queue). Java의 PriorityQueue는 기본 min-heap (가장 작은 값 우선)이고, Python은 heapq 모듈을 통해 min-heap 기능을 제공합니다. 우선순위 큐는 **그래프 알고리즘의 Dijkstra** 최단경로에서 최소 거리 정점 선택에 사용되거나, **허프만 코딩**에서 최소 가중치 간 병합 등에 활용되는 등, **Greedy 알고리즘**에서 자주 등장합니다. 또한 **실시간 데이터 스트림에서 top-N 추출**이나 **이벤트 우선 처리** 등에도 응용됩니다.

**예시:** C++ vs Java vs Python 우선순위 큐 (최소값 기준)

// C++에서 최소힙: greater를 컴퍼레이터로 사용   
std::priority\_queue<int, vector<int>, greater<int>> pq;   
pq.push(10); pq.push(3); pq.push(5);   
cout << pq.top(); // 출력: 3 (최소값)   
pq.pop(); // 최솟값 제거

PriorityQueue<Integer> pq = new PriorityQueue<>(); // 기본 최소힙   
pq.add(10); pq.add(3); pq.add(5);   
System.out.println(pq.peek()); // 출력: 3 (최소값)   
pq.poll(); // 최솟값 제거

import heapq   
pq = []   
for x in [10,3,5]:   
 heapq.heappush(pq, x)   
print(pq[0]) # 출력: 3 (최소값)   
heapq.heappop(pq) # 최솟값 제거

위의 자료구조들의 **시간 복잡도**를 요약하면 다음과 같습니다:

* *배열 기반 시퀀스 (vector/ArrayList 등):* 접근 O(1), 맨뒤 삽입/삭제 amortized O(1), 중간 삽입/삭제 O(n).
* *연결 리스트 (list/LinkedList):* 접근 O(n), 임의 위치 삽입/삭제 O(1) (사전에 해당 노드 가리킬 경우).
* *덱 (deque):* 양 끝 삽입/삭제 O(1), 임의 접근 평균 O(1) (연속 메모리 블록 내 계산), 중간 삽입/삭제는 해당 위치 탐색 필요.
* *정렬 트리 맵/셋:* 삽입/삭제/탐색 O(log n).
* *해시 맵/셋:* 평균 삽입/삭제/탐색 O(1), 최악 O(n).
* *스택/큐:* (덱 기반) 삽입/삭제 O(1), 조회 O(1).
* *우선순위 큐:* 삽입 O(log n), 삭제(최고 우선순위) O(log n), 조회(최고 우선순위 값) O(1).

자료구조를 선택할 때는 위 복잡도 특성과 데이터 규모, 연산 패턴을 모두 고려해야 합니다. **왜 해당 구조가 필요한지**를 항상 생각해보세요. 예를 들어, 데이터가 입력 순서는 상관없고 **빠른 검색**만 필요하면 해시를, **항상 정렬된 상태**로 유지하며 중간값이나 범위 쿼리가 필요하면 트리 구조를, **후입선출**이 필요하면 스택을, **선입선출**이 필요하면 큐를 쓰는 식입니다.

## 비트 연산 (Bit Operations)

컴퓨터는 결국 모든 데이터를 **이진수(bit)**로 표현하므로, 비트 단위의 조작은 알고리즘에서 매우 강력한 도구가 됩니다[[37]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=2). 비트 연산을 활용하면 논리적인 **참/거짓 상태를 집합처럼 표현**하거나, 연산을 **병렬화 및 최적화**할 수 있습니다. 이번 장에서는 비트의 개념과 기본 연산자부터, 알고리즘 문제에서 흔히 쓰이는 **비트마스크 기법**까지 살펴보겠습니다.

### 비트와 기본 연산자

**비트(bit)**는 0 또는 1의 값을 갖는 가장 작은 정보 단위이며, 8비트가 모이면 1바이트(byte)가 됩니다[[38]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%EC%BB%B4%ED%93%A8%ED%84%B0%EC%97%90%EC%84%9C%20%EC%9E%90%EB%A3%8C%EB%A5%BC%20%ED%91%9C%ED%98%84%ED%95%98%EA%B8%B0%20%EC%9C%84%ED%95%B4%20%EB%B9%84%ED%8A%B8%EB%A5%BC,%EC%82%AC%EC%9A%A9%ED%95%A9%EB%8B%88%EB%8B%A4). 정수는 이진 비트들의 집합으로 표현되고, CPU는 비트 단위 논리연산을 효율적으로 수행할 수 있습니다. 주요 **비트 연산자**는 다음 여섯 가지입니다[[39]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%EB%8B%A4%EC%9D%8C%EC%9D%80%206%EA%B0%80%EC%A7%80%20%EB%B9%84%ED%8A%B8%20%EC%97%B0%EC%82%B0%EC%9E%90%EC%9E%85%EB%8B%88%EB%8B%A4):

* & (AND): 두 비트가 모두 1일 때 결과가 1 (교집합 역할). 예) 1010 & 0100 = 0000[[40]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=a%20%3D%200b1010%2C%20b%20%3D,0b0100)
* | (OR): 두 비트 중 하나라도 1이면 결과 1 (합집합 역할). 예) 1010 | 0100 = 1110[[41]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=a%20%26%20b%20%3D%C2%A00b0000)
* ^ (XOR): 두 비트가 다를 때만 1 (배타적 논리합). 예) 1010 ^ 0100 = 1110 (한쪽에만 있는 비트 추출)[[42]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=a%20)
* ~ (NOT): 모든 비트를 반전 (1->0, 0->1)[[43]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=). (C++/Java에서는 32비트 정수 기준으로 비트를 뒤집어 음수 표현, Python에서는 무한 정수에서 2의 보수로 표현)
* << (Left Shift): 비트들을 왼쪽으로 N칸 이동 (2의 N제곱 곱과 동일). 예) a << 1은 a \* 2와 같음[[44]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=)
* >> (Right Shift): 비트들을 오른쪽으로 N칸 이동 (2의 N제곱으로 나눔). 예) a >> 1은 a / 2와 같음[[45]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=) (부호비트 처리: C++에선 구현체마다 다를 수 있고, Java는 산술 시프트 >>와 논리 시프트 >>> 구분)

**주의:** 비트 연산자의 우선순위는 비교 연산자(<, > 등)보다 낮고 논리 연산자(&&, ||)보다는 높습니다[[46]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%EB%B9%84%ED%8A%B8%20%EC%97%B0%EC%82%B0%EC%9D%98%20%EC%9A%B0%EC%84%A0%EC%88%9C%EC%9C%84%EC%97%90%20%EC%A3%BC%EC%9D%98%EA%B0%80%20%ED%95%84%EC%9A%94%ED%95%A9%EB%8B%88%EB%8B%A4,%EC%9A%B0%EC%84%A0%EC%88%9C%EC%9C%84%EA%B0%80%20%EB%86%92%EC%9C%BC%EB%82%98%20%EB%B9%84%EA%B5%90%20%EC%97%B0%EC%82%B0%EB%B3%B4%EB%8B%A8%20%EB%82%AE%EC%8A%B5%EB%8B%88%EB%8B%A4). 따라서 if (x & y == 0)와 같은 코드는 if (x & (y == 0))로 해석되므로 의도와 다르게 작동합니다[[47]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=if%20,%EA%B3%BC%20%EA%B0%99%EC%9D%8C). 비트 연산을 조건식에 사용할 때는 반드시 괄호로 우선순위를 명시하세요.

### 비트 연산 응용 팁

비트 연산은 간단한 규칙이지만 이를 응용하면 다양한 트릭과 최적화를 구현할 수 있습니다:

* **집합 연산으로서의 AND/OR:** 정수의 이진 표현을 **집합의 포함 여부**로 해석하면, AND(&)는 **교집합**, OR(|)는 **합집합** 연산이 됩니다[[48]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%26%20AND%2C%20). 예를 들어 비트열 1010과 1100을 AND 하면 1000이 되어, 두 집합에 공통으로 있는 원소에 해당하는 비트만 1로 남습니다. OR는 두 집합의 전체 원소 비트를 1로 설정합니다. (아래 **비트마스킹**에서 자세히 다룸)
* **XOR의 특성:** ^ (XOR)은 입력이 같으면 0, 다르면 1을 결과로 내므로 **toggle(토글)** 용도로 쓰입니다[[49]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=). 어떤 불(Boolean) 값을 XOR 1 하면 0->1, 1->0으로 바뀌고, 다시 XOR 1 하면 원래 값으로 돌아옵니다. 또한 **짝이 맞는 두 값의 XOR 결과는 0**이 되는 성질이 있습니다[[50]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%E2%97%86%20%EA%B0%99%EC%9D%80%20%EA%B0%92%EB%81%BC%EB%A6%AC%20XOR%ED%95%98%EB%A9%B4%200%EC%9D%B4,%EB%A7%8E%EC%9D%80%20%EA%B3%B3%EC%97%90%20%EC%A0%81%EC%9A%A9%ED%95%A0%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4). 이 덕분에, **한 번만 등장하는 원소 찾기**(예: 모든 숫자가 두 번씩 존재하고 하나만 한 번 있을 때 XOR 전체 하면 그 하나가 남음) 같은 문제에 활용됩니다. 재미있는 예로, **좌표가 쌍으로 주어지고 하나만 홀로 있는 점 찾기**에서 x좌표들 XOR, y좌표들 XOR 하면 각각 남은 하나의 좌표를 구할 수 있습니다[[50]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%E2%97%86%20%EA%B0%99%EC%9D%80%20%EA%B0%92%EB%81%BC%EB%A6%AC%20XOR%ED%95%98%EB%A9%B4%200%EC%9D%B4,%EB%A7%8E%EC%9D%80%20%EA%B3%B3%EC%97%90%20%EC%A0%81%EC%9A%A9%ED%95%A0%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4).

또한 XOR은 **비트 패턴을 원하는 대로 뒤집는 마스크**를 만들 때도 쓰입니다. 예를 들어 **영문 대문자<->소문자 변환**에서는 ASCII 코드에서 대소문자가 단지 한 비트 차이(0x20)인 것을 이용하여, 해당 비트를 XOR함으로써 간단히 변경할 수 있습니다[[51]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%E2%97%86%20%EC%96%B4%EB%96%A4%20%EC%88%98%EC%97%90%EC%84%9C%20%EB%AA%87%20%EA%B0%9C%EC%9D%98,%EA%B8%B0%EB%B2%95%EC%9D%84%20%EC%9D%B4%EC%9A%A9%ED%95%9C%20%EB%8C%80%EC%86%8C%EB%AC%B8%EC%9E%90%20%EB%B3%80%ED%99%98%20%ED%95%A8%EC%88%98%EC%9E%85%EB%8B%88%EB%8B%A4). 아래는 문자의 5번째 비트를 뒤집어 대소문자를 변환하는 함수입니다:

char caseConvert(char c) {  
 return c ^ 32; // 'A'(65, 1000001) <-> 'a'(97, 1100001) 변환[51]   
}

(※ 모든 문자에 적용할 수 있는 보편적 방법은 아니며, 영어 알파벳에만 해당)

* **NOT과 음수 인덱스:** 비트 NOT(~)은 **모든 비트를 뒤집어** 보수 표현에서는 -x-1의 값을 얻습니다. C++에서 ~i는 -i-1과 같고, Python에서는 무한 정수 기준 2의 보수로 처리됩니다. 활용 예로, C++에서 v.end() 이터레이터에서 ~i를 사용하면 편리하게 **뒤에서부터 i번째** 요소에 접근할 수 있습니다[[52]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%2F%2F%20%EC%95%9E%EC%97%90%EC%84%9C%EB%B6%80%ED%84%B0%20i%EB%B2%88%EC%A7%B8%20%EC%9B%90%EC%86%8C%EC%99%80%20%EB%92%A4%EC%97%90%EC%84%9C%EB%B6%80%ED%84%B0,i%EB%B2%88%EC%A7%B8%20%EC%9B%90%EC%86%8C%EB%A5%BC%20%EC%B6%9C%EB%A0%A5%ED%95%98%EB%8A%94%20%EC%BD%94%EB%93%9C)[[53]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=printf%28%22). 예를 들어 vec.end()[~0]는 마지막 요소, vec.end()[~1]은 뒤에서 두 번째 요소를 가리킵니다 (Python의 음수 인덱싱과 비슷한 트릭).
* **시프트 연산으로 곱셈/나눗셈 최적화:** 왼쪽 시프트 << n은 2^n 배 곱셈, 오른쪽 시프트 >> n은 2^n로 나눗셈과 같으므로 계산 최적화에 쓰입니다[[54]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%E2%97%86%202%EC%9D%98%20%EA%B1%B0%EB%93%AD%EC%A0%9C%EA%B3%B1%20%EA%B3%B1%EC%85%88%2F%EB%82%98%EB%88%97%EC%85%88). 특히 **2의 거듭제곱으로 나누기**는 일반 나눗셈 연산보다 훨씬 빠르므로, 예를 들어 8로 나누는 연산은 x >> 3으로 대체 가능합니다. 나머지 연산(%)도 2^n으로 mod 할 경우 비트 AND로 변환 가능합니다: x % 8 == x & 7 (단, **음수에 적용하면** 언어마다 결과가 다를 수 있어 주의[[55]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%2F%2F%20WARNING%20%EC%9D%8C%EC%88%98%EC%9D%BC%20%EB%95%8C%20%EC%A0%9C%EB%8C%80%EB%A1%9C,%EB%8F%99%EC%9E%91%ED%95%98%EC%A7%80%20%EC%95%8A%EC%9D%8C)). 이러한 트릭은 성능이 중요한 경우에 고려되지만, 고급 컴파일러는 이러한 최적화를 자동으로 해주기도 합니다.
* **SWAP 트릭 (XOR Swap):** XOR를 사용하면 추가 변수 없이 두 값을 교환(swap)하는 코드도 구현할 수 있습니다[[56]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=void%20xor_swap%28int,). 예:
* // 경고: 실제로는 권장되지 않음  
  void xor\_swap(int &x, int &y) {  
   x ^= y;  
   y ^= x;  
   x ^= y;  
  }
* 그러나 이 방법은 가독성이 떨어지고, 두 포인터가 같은 주소를 가리킬 때 문제가 생기는 등 실용적이지 않아 **현업에서는 임시 변수를 사용하는 일반 swap이 더 안전**합니다[[57]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=).

### 비트마스킹과 활용

**비트마스킹(bit masking)**은 정수의 비트 한 개 한 개를 **불리언 배열처럼 간주**하여 사용함으로써, **여러 불 값들을 한 변수에 압축**해서 표현하는 기법입니다. 예를 들어 32비트 정수 하나는 0/1로 표현되는 상태 32개를 담을 수 있으므로, 최대 32개의 요소로 구성된 **부분집합의 상태** 등을 표시할 때 유용합니다[[58]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=2,%ED%91%9C%ED%98%84%EC%9D%84%20%EC%89%BD%EA%B2%8C%20%ED%95%A0%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4).

* **부분집합 표현:** 어떤 집합에서 원소의 포함 여부를 1(포함) 또는 0(미포함)으로 나타내어, n개의 원소에 대해 2^n개의 부분집합을 0부터 2^n-1까지의 정수로 대응시킬 수 있습니다. 예를 들어 원소 3개짜리 집합의 부분집합을 나타내는 3비트 값을 생각하면, 101(5)은 {0번 원소, 2번 원소}의 부분집합을 뜻합니다. 이 기법을 활용하면 **모든 부분집합 순회**를 0부터 (1<<n)-1까지 루프 도는 것으로 구현할 수 있고, 특정 부분집합의 부분집합을 구하는 등의 **비트 DP** 기법도 사용됩니다.
* **집합 연산:** 앞서 소개한 대로, 두 비트마스크 값에 대해 AND, OR, XOR를 적용하면 각 비트 위치를 **집합의 원소로 해석**했을 때 교집합, 합집합, 대칭차 등을 구할 수 있습니다[[48]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%26%20AND%2C%20). 이를 통해 두 집합 간 공통 원소가 있는지, 모든 원소가 다 다른지 등의 검사가 상수 시간에 가능합니다.

예를 들어, 사람 A와 B의 친구 목록을 비트마스크로 표현해보겠습니다. A의 친구가 {0,3,6,7,10,...}이라면 해당 번호에 해당하는 비트를 1로 둔 정수로 나타내고, B도 마찬가지로 표현합니다[[59]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%EC%82%AC%EB%9E%8C%EC%97%90%200,%EC%9D%B4%EB%9D%BC%EA%B3%A0%20%ED%95%A9%EC%8B%9C%EB%8B%A4). 이때 **두 사람이 모두 친구로 가진 사람**은 A마스크 AND B마스크를 하면 바로 구해지고, **둘 중 한 명이라도 친구인 사람**은 OR로 구할 수 있습니다[[60]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=A%2C%20B%20%EB%AA%A8%EB%91%90%EC%99%80%20%EC%B9%9C%EA%B5%AC%EC%9D%B8%20%EC%82%AC%EB%9E%8C%EC%9D%80%3F). 이러한 비트 연산을 사용하면 반복문으로 일일이 확인하는 것보다 훨씬 빠르게 계산할 수 있습니다.

* **비트 개수 세기(popcount):** 비트마스크를 쓰다 보면 **1로 설정된 비트의 개수**(즉, 부분집합의 원소 수)를 알아야 할 때가 많습니다. 이를 계산하는 여러 방법이 있는데, C++17에는 std::popcount 함수가, **GNUC** 계열에는 \_\_builtin\_popcount 등이 있습니다. 파이썬은 3.8부터 정수 메서드로 .bit\_count()를 제공하며, Java도 Integer.bitCount() 메서드를 제공합니다. 직접 구현하는 경우, **Brian Kernighan의 알고리즘**이 유명합니다[[61]](https://advenoh.tistory.com/18#:~:text=,1%29%EB%A5%BC%20%EC%A0%95%EC%88%98%EA%B0%92%EC%9D%B4%200%EC%9D%B4%20%EB%90%A0%EB%95%8C%EA%B9%8C%EC%A7%80%20%EB%B0%98%EB%B3%B5%ED%95%9C%EB%8B%A4). 이 방법은 다음과 같습니다: **주어진 수 N에 대해 반복해서 N = N & (N-1) 연산을 수행하면, N이 0이 될 때까지 딱 1의 개수만큼 반복**됨을 이용합니다[[61]](https://advenoh.tistory.com/18#:~:text=,1%29%EB%A5%BC%20%EC%A0%95%EC%88%98%EA%B0%92%EC%9D%B4%200%EC%9D%B4%20%EB%90%A0%EB%95%8C%EA%B9%8C%EC%A7%80%20%EB%B0%98%EB%B3%B5%ED%95%9C%EB%8B%A4). 한 번의 N = N & (N-1)은 N의 가장 오른쪽에 있는 1비트를 제거하는 효과가 있으므로, 이 과정을 1의 개수만큼 수행하면 N이 0이 됩니다. 결과적으로 루프를 돈 횟수가 1의 개수와 같게 되므로 이를 카운트하면 됩니다[[61]](https://advenoh.tistory.com/18#:~:text=,1%29%EB%A5%BC%20%EC%A0%95%EC%88%98%EA%B0%92%EC%9D%B4%200%EC%9D%B4%20%EB%90%A0%EB%95%8C%EA%B9%8C%EC%A7%80%20%EB%B0%98%EB%B3%B5%ED%95%9C%EB%8B%A4). 이 알고리즘의 복잡도는 1의 개수가 k일 때 O(k)로, 최악의 O(n) 비트수만큼 돌리는 것보다 평균적으로 빠릅니다.

**예시:** C++/Java/Python에서 1비트 개수 세기

unsigned int n = 23; // 23의 이진수: 10111 (1이 4개)   
int count = 0;   
while(n) {   
 n &= (n-1); // 최하위 1비트 제거   
 count++;   
}   
cout << count; // 출력: 4

int n = 23;   
int count = Integer.bitCount(n); // 자바 내장 함수 사용 (결과: 4)   
System.out.println(count);

n = 23   
count = n.bit\_count() # 파이썬 3.8+ 내장 (결과: 4)   
print(count)

이외에도 비트마스크는 **DP(Bit DP)**에서 상태를 compact하게 표현하거나, **그래프의 모든 경우 탐색**(예: 외판원 문제 상태표현) 등에 활발히 쓰입니다. 비트 연산 트릭은 알고리즘 문제 풀이의 **테크닉**으로 자주 등장하므로, 다양한 패턴을 익혀두면 유용합니다.

## 문자열 기본

문자열(String)은 **문자(char)들의 나열**로, 시퀀스 컨테이너의 일종입니다. 사실 C++에서 std::string은 basic\_string<char> 형태로 벡터와 유사하게 동작하며, Java의 String 클래스는 불변 객체(immutable)지만 문자 배열 기반으로 관리되고, Python의 str도 불변 시퀀스 타입입니다. 문자열은 프로그래밍에서 **가장 흔하게 다루는 데이터 타입 중 하나**로, 알고리즘 문제에서도 **문자열 처리**는 매우 중요합니다 (예: 회문 검사, 패턴 매칭, 파싱 등). 이번 장에서는 문자열을 다루는 방법과 주의점, 세 언어별 특징을 살펴보겠습니다.

### 문자열 선언과 초기화, 입출력

* **C++:** <string> 헤더의 std::string 클래스를 사용합니다. 리터럴은 "로 감싸 표현하며, string s = "Hello";처럼 선언합니다. C 스타일 문자열(널종료 char 배열)과 호환이 가능하며, std::getline으로 공백 포함 입력을 받을 수 있습니다. 주의할 점은 cin >> s는 공백 기준으로 토큰 단위 입력을 받으므로, 한 줄 전체를 입력받으려면 getline(cin, s)를 써야 합니다.
* **Java:** String str = "Hello";처럼 리터럴로 초기화하거나, new String("Hello")로 생성합니다. 표준 입력은 Scanner sc.nextLine()으로 한 줄 읽을 수 있습니다. Java의 문자열은 **immutable**이라 한 번 값이 정해지면 변경시 새로운 객체가 생성됩니다. 입출력 시엔 인코딩(UTF-8 기본) 등을 유의해야 하지만, 일반적인 영어/숫자 입력은 신경쓰지 않아도 됩니다.
* **Python:** s = "Hello"로 간단히 선언하며, 작은따옴표나 큰따옴표 둘 다 사용 가능합니다. input() 함수로 한 줄 입력받아 문자열로 저장하고, print()로 출력합니다. Python 문자열은 Unicode 문자열을 기본으로 취급하여, 한글과 같은 다국어도 기본 지원됩니다. (C++에서는 wstring이나 UTF-8 처리 필요, Java는 내부적으로 UTF-16 사용)

### 문자열의 주요 연산과 메서드

**길이 및 접근:** 세 언어 모두 문자열 길이를 구하는 연산을 제공합니다. C++은 s.size() 또는 s.length() (동일 기능)를 쓰고, Java는 s.length() 메서드, Python은 전역 함수 len(s)를 사용합니다. 문자열의 각 문자에 접근할 때, C++은 s[i] (참조 반환, 읽기/쓰기 가능)를, Java는 s.charAt(i) (char 반환, 읽기 전용)를, Python은 s[i] (char 유사 객체 반환, 읽기 전용)를 사용합니다. Python과 Java에서는 문자열이 불변(immutable)이라 s[i] = 'X'와 같이 **직접 수정이 불가능**합니다[[62]](https://www.upgrad.com/tutorials/software-engineering/java-tutorial/strings-in-java-vs-strings-in-cpp/#:~:text=Strings%20in%20Java%20vs%20Strings,contrast%2C%20C%2B%2B%20std%3A%3Astring%20is). C++ std::string은 내부적으로 가변(mutable) 시퀀스여서 s[0] = 'X'; 식으로 수정이 가능합니다[[63]](https://stackoverflow.com/questions/28442719/are-c-strings-mutable-unlike-java-strings#:~:text=).

**추가와 연결:** C++에서는 s += "world" 혹은 s.append("world") 로 다른 문자열을 이어붙일 수 있으며, 단일 문자 추가는 push\_back()도 제공합니다. Java에서는 str1 + str2으로 간단히 연결이 가능하지만 내부적으로 StringBuilder를 사용하여 새 문자열을 만듭니다. 반복된 연결이 많은 경우 StringBuilder sb = new StringBuilder(); sb.append(...) 방식이 성능에 유리합니다. Python에서는 s1 + s2로 연결하거나, 문자열 리스트를 다 합쳐야 할 경우 '.join(list)` 방법을 권장합니다 (여러 문자열을 반복 연결할 때는 join이 효율적).

**부분 문자열 (substring):** 문자열의 일부분을 추출하는 연산입니다. C++ string.substr(pos, len) 메서드는 pos 위치부터 len 길이만큼의 부분 문자열을 반환합니다. Java String.substring(beginIndex, endIndex)는 beginIndex부터 endIndex-1까지를 새로운 String으로 만듭니다. Python은 **슬라이싱(slicing)** 구문 s[a:b]으로 a부터 b-1까지 잘라낸 문자열을 얻을 수 있습니다 (omit 시 시작이나 끝까지). 음수 인덱스도 사용 가능하여, s[-3:]는 끝에서 3글자 등을 쉽게 구할 수 있습니다.

**검색 및 비교:** C++의 s.find("lo")는 문자열 내에 "lo"가 처음 등장하는 인덱스를 반환하고, 없으면 npos를 반환합니다. Java의 s.indexOf("lo")가 같은 역할을 합니다 (lastIndexOf는 뒤에서부터 찾기). Python은 s.find("lo") 또는 s.index("lo") (못 찾으면 에러) 메서드가 있고, 'lo' in s 구문으로 포함 여부를 빠르게 확인 가능합니다. 문자열 비교는 C++에선 == 연산자가 오버로딩되어 값 비교를 합니다 (compare 메서드도 있음), Java는 equals() 메서드를 써야 값 비교가 되며 ==는 참조동일성 비교이므로 주의합니다. Python은 ==로 값 비교가 가능합니다.

**대소문자 변환 등 편의:** 세 언어 모두 문자열 대소문자 변경 등의 함수를 제공합니다. C++은 <cctype>의 toupper, tolower (단일 문자) 함수를 사용하거나 Boost 등 활용, Java는 s.toUpperCase(), s.toLowerCase(), Python은 s.upper(), s.lower() 메서드를 제공합니다. 이 외에 양쪽 공백 제거(trim), 문자열 분할(split), 포맷(format) 등 풍부한 함수들이 있지만 여기서 모두 다루지는 않습니다. 알고리즘 문제에서 자주 쓰이는 몇 가지만 언급하면: Python s.split() (구분자로 분리), s.join(list) (리스트를 구분자로 연결), Java String.split(regex), C++ getline으로 한 줄 읽은 뒤 istringstream으로 파싱 등입니다.

### 이터레이션과 변경 (문자열을 다루는 방식)

문자열을 다룰 때 **문자 단위 반복**은 기본입니다. C++에서는 다음과 같이 두 가지 방법을 흔히 씁니다:

string s = "Hello";  
for(char c : s) {  
 cout << c << '\n';  
}  
for(int i=0;i<s.size();++i) {  
 cout << s[i] << '\n';  
}

첫 번째는 범위 기반 for (range-for)으로, 두 번째는 인덱스를 사용한 접근입니다. Java는 enhanced-for에 직접 문자열을 넣을 수 없지만 s.toCharArray()로 char 배열을 얻어 for(char c : s.toCharArray())로 순회하거나, 인덱스 기반으로 for(int i=0;i<s.length();++i)와 charAt(i)를 사용합니다. Python은 문자열도 iterable이므로 for c in s: 형태로 한 문자씩 순회할 수 있습니다.

**문자열 수정**은 C++에선 직접 가능하지만, Java/Python에서는 불변성 때문에 **새 문자열 생성**이 수반됩니다[[62]](https://www.upgrad.com/tutorials/software-engineering/java-tutorial/strings-in-java-vs-strings-in-cpp/#:~:text=Strings%20in%20Java%20vs%20Strings,contrast%2C%20C%2B%2B%20std%3A%3Astring%20is). 예를 들어 Python에서 문자열을 효율적으로 수정하려면 **리스트로 변환 후 수정**하거나, **문자열 슬라이싱**으로 잘라서 새로운 문자열을 만들어야 합니다. Java도 많은 문자 조작이 필요하면 StringBuilder나 char[]로 변환 후 작업하는 것이 좋습니다. 불변 객체의 장점은 **보안성과 멀티스레드 안전성, 해시값 캐싱 등**이 있지만, 반복 수정이 필요한 알고리즘에서는 오버헤드가 될 수 있으니 알고 있어야 합니다.

**예시:** 문자열의 문자 하나 바꾸기  
- C++: s[0] = 'Y'; // 가능  
- Java: char[] arr = str.toCharArray(); arr[0] = 'Y'; str = new String(arr); (또는 StringBuilder 이용)  
- Python: lst = list(s); lst[0] = 'Y'; s = ''.join(lst) (파이썬 문자열 직접 수정 불가)

### 문자열 관련 자주 나오는 문제 예시

문자열은 알고리즘 문제에서 매우 빈번하게 등장하며, 아래와 같은 고전적인 문제들이 있습니다:

* **회문(Palindrome) 검사:** 회문은 앞뒤로 읽어도 같은 문자열을 말합니다. 예: "level", "abba". 회문 검사 코드는 양 끝에서 좁혀오며 문자가 모두 일치하는지 확인하거나, 반대를 뒤집어 비교하는 방식으로 구현합니다. 대소문자 무시나 영문자 외 제거 등이 가미될 수 있습니다.
* *접근:* s == reverse(s)인지 보거나, 투 포인터로 i=0, j=n-1에서 while(i<j) 비교. Python에서는 s == s[::-1] 한 줄로 쉽게 검사 가능합니다.
* **아나그램(Anagram) 판별:** 두 문자열이 **철자 구성**이 동일하지만 순서만 다른 경우 서로 아나그램 관계라고 합니다. 예: "listen"과 "silent". 문제에서는 주로 주어진 두 문자열이 아나그램인지 묻거나, 한 문자열을 아나그램으로 만들기 위해 제거/추가해야 할 최소 문자 수 등을 묻습니다.
* *접근:* 각각 정렬해서 같은지 비교 (O(n log n)), 또는 알파벳 빈도수를 세어 비교 (O(n)). Python에선 collections.Counter로 빈도 딕셔너리 비교도 편리합니다.
* **문자열 뒤집기:** 주어진 문자열을 반대로 만드는 문제는 구현 연습으로 자주 등장합니다. Python에선 슬라이싱 s[::-1]로 바로 해결되고, C++/Java도 라이브러리 (reverse 함수나 StringBuilder.reverse())가 있습니다. 직접 구현하려면 투포인터로 swap 하거나 스택을 이용하기도 합니다.
* **빈도수 계산:** 문자열에서 **각 문자의 등장 횟수**를 세는 문제는 기본적이지만, 자료구조 활용을 물어볼 수 있습니다. 예를 들어 알파벳 소문자로만 구성된 문자열에서 가장 많이 등장하는 문자 찾기 등. 해시맵(map<char,int> or unordered\_map)을 사용하거나, 문자 범위가 작으면 길이 26짜리 배열을 사용하여 count[c - 'a']++ 처럼 풀 수 있습니다. 이러한 **Direct Address Table(DAT)** 방식은 공간을 조금 더 쓰는 대신 해시를 쓰지 않아도 돼 빠르고, 메모리가 허용된다면 코딩 테스트에서 자주 쓰입니다. Python에서는 collections.Counter를 쓰면 한 줄로 빈도수를 얻을 수 있습니다 (물론 내부적으로 해시 사용).

이 외에도 문자열 처리는 **문자열 검색(KMP 알고리즘 등)**, **트라이(Trie)**, **정규표현식** 등 심화 주제가 많지만, 이러한 고급 주제는 뒤에서 다룰 예정입니다. 우선은 문자열을 다루는 기본 방법과, 구현 시 주의사항(불변 객체 처리, 인코딩 등)을 확실히 익혀두어야 합니다.

### 문자와 코드값 (문자 vs 정수)

프로그래밍에서 문자와 정수는 밀접한 관계가 있습니다. **문자에는 코드값(아스키 혹은 유니코드 값)**이 부여되어 있어, 내부적으로는 정수로 저장되기 때문입니다. 따라서 '1'과 1은 타입도 값도 완전히 다릅니다. '1'은 문자 '1'의 코드(아스키 49)에 해당하고, 정수 1은 이진수 0001로 표현된 값일 뿐입니다. 실수로 문자를 정수로 혼동하면 엉뚱한 결과가 나오므로 유의해야 합니다.

* '1' (문자형) vs 1 (정수형): 전자는 문자 **'1'의 코드값** 49를 의미하고, 후자는 정수 1 그 자체입니다. '2'는 50, 'A'는 65 등 표준 ASCII 코드 체계가 있습니다. 따라서 '1' + '1' (문자 덧셈)은 49+49=98이 되어 문자 'b'에 해당하는 코드값이 됩니다. 파이썬에서 "1" \* 3은 문자열 "111"이 되지만, C++에서 '1' \* 3은 정수 49\*3=147이 되어 해당 코드(147)에 해당하는 문자로 바뀔 수 있습니다.

**아스키(ASCII)와 유니코드(Unicode):** ASCII는 영문자, 숫자, 특수기호 등을 7비트(128개)로 표현한 초기 표준입니다. 유니코드는 전 세계 모든 문자를 표현하기 위한 확장 체계로, ASCII를 포함하여 한글, 한자 등도 코드값이 할당되어 있습니다. Java와 Python의 str은 기본적으로 유니코드 문자 단위를 사용하며, C++도 char 외에 wchar\_t(유니코드), 또는 UTF-8 인코딩을 string에 담아 처리하는 경우가 많습니다. 알고리즘 문제에서는 주로 ASCII 범위(영문/숫자)가 등장하므로, 문자 코드를 다룰 때 ASCII 코드 값을 염두에 두고 연산합니다.

**문자->숫자 변환:** '0'~'9' 문자는 연속된 ASCII 코드(48~57)를 가지므로, **문자 '0'를 빼는 연산**으로 해당 숫자로 쉽게 변환 가능합니다. 예를 들어 C++에서 '7' - '0'은 55-48 = 7이 됩니다[[62]](https://www.upgrad.com/tutorials/software-engineering/java-tutorial/strings-in-java-vs-strings-in-cpp/#:~:text=Strings%20in%20Java%20vs%20Strings,contrast%2C%20C%2B%2B%20std%3A%3Astring%20is)[[63]](https://stackoverflow.com/questions/28442719/are-c-strings-mutable-unlike-java-strings#:~:text=). Java도 char 간 뺄셈이 가능하여 같은 결과를 얻습니다. Python에서는 문자에 대해 뺄셈이 지원되지 않으므로, 내장 함수 ord()를 사용하여 코드값을 얻고, ord('7') - ord('0')로 계산하거나, 더 간단히 int("7")으로 문자열을 숫자로 바꾸면 됩니다. 반대로 숫자를 문자로 바꾸는 것은 C++에서 char('0' + x) (x는 0~9), Java에서는 Character.forDigit(x, 10) 또는 (char)('0'+x), Python에서는 chr(ord('0')+x)나 str(x)를 사용합니다.

* **문자인지 판별:** C++ <cctype>의 isdigit(ch) 함수는 해당 문자가 숫자 문자('0'~'9')인지 확인합니다. Java는 Character.isDigit(ch) 메서드, Python은 문자열 메서드 str.isdigit()을 제공합니다. 이들 함수를 활용하여 입력 문자가 숫자인지 여부, 대문자인지 여부(isupper 등) 등을 쉽게 판단할 수 있습니다.
* **예시:** 'A'를 소문자로, '9'를 숫자로 변환
* C++: char lower = ch + 32; // 대문자->소문자 (대문자 'A'(65) + 32 = 'a'(97))  
    int num = ch - '0'; // '9'(57) - '0'(48) = 9
* Java: char lower = Character.toLowerCase(ch);  
    int num = Character.getNumericValue(ch); // '9' -> 9
* Python: lower = ch.lower()  
    num = int(ch)

### 언어별 문자열 메서드 요약

마지막으로, 알고리즘 구현 시 유용한 **문자열 관련 메서드 및 기능**을 세 언어 기준으로 정리합니다:

* **길이 구하기:** C++ s.size()/s.length(), Java s.length(), Python len(s)
* **문자 접근:** C++ s[i], Java s.charAt(i), Python s[i] (인덱스로 접근)
* **연결(Concatenate):** C++ s1 + s2 또는 s.append(s2), Java s1 + s2 또는 s1.concat(s2), Python s1 + s2 또는 ''.join(list)
* **부분문자열:** C++ s.substr(pos, n), Java s.substring(a, b), Python s[a:b]
* **검색:** C++ s.find("sub"), Java s.indexOf("sub"), Python s.find("sub") 또는 'sub' in s
* **비교:** C++ == (사전순 비교는 s1 < s2 등 가능), Java s1.equals(s2) (사전순: s1.compareTo(s2)), Python == (사전순: s1 < s2)
* **대소문자:** C++ toupper(c), tolower(c) (문자 단위), Java s.toUpperCase(), Python s.upper()
* **공백제거:** C++ boost::trim(s) (부스트 사용 또는 수동 구현), Java s.trim(), Python s.strip()
* **분할:** C++ (없음, istream이나 strtok 등 활용), Java s.split("delim"), Python s.split("delim")
* **포맷:** C++ sprintf (C스타일) 또는 std::format(C++20), Java String.format, Python f-string f"{var}"

지금까지 자료구조와 문자열 기본에 대해 살펴보았습니다. 정리하면, **상황에 맞는 자료구조의 선택과 활용**이 알고리즘 효율에 큰 영향을 미치며, 비트 연산과 문자열 처리 등의 기초 테크닉 역시 문제 해결에 자주 사용됩니다. 이번 주차의 내용을 통해 다양한 구조의 장단점을 이해하고, 구현 방법을 C++/Java/Python 각각으로 익혀두세요. 다음 시간에는 이러한 자료구조들을 활용한 알고리즘 문제와 고급 주제들에 대해 다룰 예정입니다.

[[1]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=C%2B%2B%3A%20Visual%20Studio%20%EB%98%90%EB%8A%94%20Visual,%ED%8C%8C%EC%9D%BC%20%EB%8B%A8%EC%9C%84%20%EC%8B%A4%ED%96%89%EC%9D%B4%20%EC%89%AC%EC%9A%B4%20%ED%8E%B8%EC%9E%85%EB%8B%88%EB%8B%A4) [[2]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=VS%20Code%EB%A5%BC%20%ED%99%9C%EC%9A%A9%ED%95%9C%20%ED%86%B5%ED%95%A9%20%ED%99%98%EA%B2%BD%3A,%ED%95%98%EB%8A%94%20%EA%B2%83%EC%9D%84%20%EC%9E%90%EB%8F%99%ED%99%94%ED%95%A0%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4) [[3]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=%EC%9D%B4%EB%B2%88%20%EC%8A%A4%ED%84%B0%EB%94%94%EC%9D%98%20%EA%B6%81%EA%B7%B9%EC%A0%81%EC%9D%B8%20%EB%AA%A9%ED%91%9C%EB%8A%94%20%EA%B8%B0%EC%97%85,%EB%AA%A9%ED%91%9C%EC%99%80%20%EB%B0%A9%ED%96%A5%EC%84%B1%EC%9D%84%20%EA%B0%96%EA%B3%A0%20%ED%95%99%EC%8A%B5%ED%95%A0%20%EC%98%88%EC%A0%95%EC%9E%85%EB%8B%88%EB%8B%A4) [[4]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=%EB%AC%B8%EC%A0%9C%20%ED%95%B4%EA%B2%B0%20%EB%8A%A5%EB%A0%A5%20%ED%96%A5%EC%83%81%3A%20%EB%8B%A4%EC%96%91%ED%95%9C,%EC%BD%94%EB%93%9C%EB%A5%BC%20%EC%9D%BD%EA%B3%A0%20%EC%9D%B4%ED%95%B4%ED%95%98%EB%8A%94%20%EB%8A%A5%EB%A0%A5%EC%9D%84%20%ED%82%A4%EC%9B%8C%EB%B4%85) [[5]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=%EC%9A%B0%EB%A6%AC%20%EC%8A%A4%ED%84%B0%EB%94%94%EC%9D%98%20%EA%B8%B0%EB%B3%B8%20%EB%AC%B8%EC%A0%9C%ED%92%80%EC%9D%B4%20%ED%94%8C%EB%9E%AB%ED%8F%BC%EC%9D%80,%EB%AC%B8%EC%A0%9C%EB%93%A4%EC%9D%84%20%ED%92%80%EC%96%B4%EB%B3%B4%EB%8A%94%20%EA%B2%83%EC%9D%84%20%EA%B6%8C%EC%9E%A5%EB%93%9C%20%EB%A6%BD%EB%8B%88%EB%8B%A4) [[6]](file://file-Jk4ezPBXEwEvzrpm863pVL#:~:text=%EC%A0%95%EB%A6%AC%3A%20%EC%97%AC%EB%9F%AC%20%ED%94%8C%EB%9E%AB%ED%8F%BC%EC%9D%B4%20%EC%9E%88%EC%A7%80%EB%A7%8C%2C%20%ED%95%B5%EC%8B%AC%EC%9D%80,%EB%8B%A4%EC%96%91%20%ED%95%9C%20%EC%9C%A0%ED%98%95%EC%97%90%20%EC%9D%B5%EC%88%99%ED%95%B4%EC%A7%80%EB%8F%84%EB%A1%9D%20%ED%95%A9%EC%8B%9C%EB%8B%A4) study-note-week01.pdf

<file://file-Jk4ezPBXEwEvzrpm863pVL>

[[7]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Sequence%20Containers%EB%8A%94%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%88%9C%EC%B0%A8%EC%A0%81%EC%9C%BC%EB%A1%9C%20%EC%A0%80%EC%9E%A5%ED%95%98%EB%8A%94,%EC%9C%A0%EC%A7%80%ED%95%A0%20%ED%95%84%EC%9A%94%EA%B0%80%20%EC%97%86%EB%8B%A4%EB%A9%B4%20%EC%A2%8B%EC%9D%80%20%EC%84%A0%ED%83%9D%EC%9E%85%EB%8B%88%EB%8B%A4) [[8]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Sequence%20Containers) [[9]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%A9%94%EB%AA%A8%EB%A6%AC%EC%83%81%EC%97%90%EC%84%9C%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EA%B0%80%20%EC%97%B0%EC%86%8D%EC%A0%81%EC%9C%BC%EB%A1%9C%20%EC%9C%84%EC%B9%98%ED%95%98%EB%8A%94%20%EB%B0%B0%EC%97%B4%EC%9E%85%EB%8B%88%EB%8B%A4) [[10]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Avector%EC%9D%98%20%EA%B2%BD%EC%9A%B0%20%EB%B0%B0%EC%97%B4%EC%9D%98%20%ED%81%AC%EA%B8%B0%EB%A5%BC%20%EB%9F%B0%ED%83%80%EC%9E%84%EC%97%90%EC%84%9C,%EA%B7%B8%EB%A0%87%EC%A7%80%20%EC%95%8A%EB%8B%A4%EB%A9%B4%20std%3A%3Avector%EB%A5%BC%20%EC%84%A0%ED%83%9D%ED%95%98%EC%8B%9C%EB%A9%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4) [[11]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Avector%EB%8A%94%20%ED%8F%AC%EC%9D%B8%ED%84%B0%20%EC%84%B8%20%EA%B0%9C%EB%A1%9C%20%EA%B5%AC%ED%98%84%EB%90%98%EC%96%B4,%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4) [[12]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=1,%EA%B0%80%EB%A6%AC%ED%82%A4%EB%8A%94%20%ED%8F%AC%EC%9D%B8%ED%84%B0) [[13]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EB%92%A4%EC%97%90%20%EC%82%BD%EC%9E%85%ED%95%98%EB%A9%B4%20%EB%B9%A8%EA%B0%84%20%ED%8F%AC%EC%9D%B8%ED%84%B0%EA%B0%80,%EC%82%BD%EC%9E%85%EB%90%98%EA%B3%A0%2C%20%EB%B9%A8%EA%B0%84%20%ED%8F%AC%EC%9D%B8%ED%84%B0%EA%B0%80%201%20%EC%A6%9D%EA%B0%80%ED%95%A9%EB%8B%88%EB%8B%A4) [[14]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=vector%20%EB%8D%B0%EC%9D%B4%ED%84%B0%20%EC%82%BD%EC%9E%85%20%EC%8B%9C%20%EC%A3%BC%EC%9D%98%EC%82%AC%ED%95%AD%3A) [[15]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EC%9D%BC%EB%B0%98%EC%A0%81%EC%9C%BC%EB%A1%9C%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EB%92%A4%EC%97%90%20%EC%82%BD%EC%9E%85%ED%95%98%EB%8A%94%20push_back%EC%97%B0%EC%82%B0%EC%9D%80,%EC%A7%81%EC%A0%91%20%ED%99%95%EC%9D%B8%ED%95%B4%20%EB%B3%BC%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4) [[16]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Container%20%EC%95%9E%2C%20%EB%92%A4%EC%97%90%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EB%B9%A0%EB%A5%B4%EA%B2%8C,ended%20queue%EC%9E%85%EB%8B%88%EB%8B%A4) [[17]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Adeque%EB%8A%94%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%8D%BC%EC%97%90%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC,deque%20%EB%8A%94%20%EC%9C%A0%EC%9A%A9%ED%95%9C%20%EC%84%A0%ED%83%9D%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4) [[18]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Alist%20) [[19]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8A%90%EB%A6%BD%EB%8B%88%EB%8B%A4,%EB%B9%A0%EB%A5%B8%20std%3A%3Aforward_list%EA%B0%80%20%EC%A2%8B%EC%9D%80%20%EC%84%A0%ED%83%9D%EC%9D%BC%20%EA%B2%83%EC%9E%85%EB%8B%88%EB%8B%A4) [[20]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%ED%95%B4%EB%8B%B9%20%EB%82%B4%EC%9A%A9%EB%A7%8C%20%EB%B3%B4%EB%A9%B4%20deque%EA%B0%80%20vector%EB%B3%B4%EB%8B%A4,%EC%83%81%ED%99%A9%EC%97%90%EC%84%9C%EB%8A%94%20deque%EA%B0%80%20%EB%B6%88%EB%A6%AC%ED%95%9C%20%EC%A0%90%EC%9D%B4%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4) [[21]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=linked%20list%EC%9E%85%EB%8B%88%EB%8B%A4,1%29%EC%97%90%20%EC%A0%91%EA%B7%BC%29%EB%8A%94%20%EB%B6%88%EA%B0%80%EB%8A%A5%ED%95%A9%EB%8B%88%EB%8B%A4) [[22]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=linked%20list%EC%9E%85%EB%8B%88%EB%8B%A4,1%29%EC%97%90%20%EC%A0%91%EA%B7%BC%29%EB%8A%94%20%EB%B6%88%EA%B0%80%EB%8A%A5%ED%95%A9%EB%8B%88%EB%8B%A4) [[23]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Alist%EB%8A%94%20doubly,%EB%B9%A0%EB%A5%B8%20std%3A%3Aforward_list%EA%B0%80%20%EC%A2%8B%EC%9D%80%20%EC%84%A0%ED%83%9D%EC%9D%BC%20%EA%B2%83%EC%9E%85%EB%8B%88%EB%8B%A4) [[26]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Associative%20Containers) [[27]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EC%9E%90%EA%B0%80%20%EA%B7%A0%ED%98%95%20%EC%9D%B4%EC%A7%84%20%ED%83%90%EC%83%89%20%ED%8A%B8%EB%A6%AC%28self,%EA%B2%BD%EC%9A%B0%EC%97%90%EB%8F%84%20%EC%9A%B0%EC%88%98%ED%95%9C%20%EC%8B%A4%ED%96%89%20%EC%8B%9C%EA%B0%84%EC%9D%84%20%EA%B0%80%EC%A7%91%EB%8B%88%EB%8B%A4) [[28]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Red) [[29]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Aset%20) [[30]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%ED%95%B4%EC%8B%9C%EA%B0%92%EC%9D%84%20%EC%82%AC%EC%9A%A9%ED%95%B4%20%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%A0%80%EC%9E%A5%ED%95%98%EB%8A%94%20%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0%EC%9E%85%EB%8B%88%EB%8B%A4,Associated%20Container%EB%A5%BC%20%EC%82%AC%EC%9A%A9%ED%95%98%EB%8A%94%20%EA%B2%8C%20%EC%A2%8B%EC%8A%B5%EB%8B%88%EB%8B%A4) [[31]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%82%B7%EC%97%90%20%EB%82%98%EB%88%A0%EC%84%9C,n%29%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4) [[32]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%A5%BC%20%EC%97%AC%EB%9F%AC%20%EA%B0%9C%EC%9D%98%20%EB%B2%84%ED%82%B7%EC%97%90%20%EB%82%98%EB%88%A0%EC%84%9C,n%29%EC%9D%B4%20%EB%90%A9%EB%8B%88%EB%8B%A4) [[33]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=%EA%B8%B0%EB%B3%B8%EC%9C%BC%EB%A1%9C%20%EC%82%AC%EC%9A%A9%ED%95%98%EB%8A%94%20hash%20function%EC%9D%80%20std%3A%3Ahash%EC%9D%B4%EA%B3%A0%2C,primitive%20type%EA%B3%BC%20std%3A%3Astring%20%EB%93%B1%EC%9D%B4%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4) [[34]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=Container%20Adaptors) [[35]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Astack%2C%20std%3A%3Aqueue) [[36]](file://file-Edsys4einV4shB8KZQcFA1#:~:text=std%3A%3Apriority_queue) 1.+STL.docx

<file://file-Edsys4einV4shB8KZQcFA1>

[[24]](file://file-JBLUSFLNtvecfTsT2Xsuci#:~:text=Array%EC%99%80%20%EB%B9%84%EA%B5%90%20%EB%B0%8F%20%EA%B3%84%EC%82%B0%EB%B3%B5%EC%9E%A1%EB%8F%84) [[25]](file://file-JBLUSFLNtvecfTsT2Xsuci#:~:text=Insert%20element%20at%20here) 3.+연결+리스트.docx

<file://file-JBLUSFLNtvecfTsT2Xsuci>

[[37]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=2) [[38]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%EC%BB%B4%ED%93%A8%ED%84%B0%EC%97%90%EC%84%9C%20%EC%9E%90%EB%A3%8C%EB%A5%BC%20%ED%91%9C%ED%98%84%ED%95%98%EA%B8%B0%20%EC%9C%84%ED%95%B4%20%EB%B9%84%ED%8A%B8%EB%A5%BC,%EC%82%AC%EC%9A%A9%ED%95%A9%EB%8B%88%EB%8B%A4) [[39]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%EB%8B%A4%EC%9D%8C%EC%9D%80%206%EA%B0%80%EC%A7%80%20%EB%B9%84%ED%8A%B8%20%EC%97%B0%EC%82%B0%EC%9E%90%EC%9E%85%EB%8B%88%EB%8B%A4) [[40]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=a%20%3D%200b1010%2C%20b%20%3D,0b0100) [[41]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=a%20%26%20b%20%3D%C2%A00b0000) [[42]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=a%20) [[43]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=) [[44]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=) [[45]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=) [[46]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%EB%B9%84%ED%8A%B8%20%EC%97%B0%EC%82%B0%EC%9D%98%20%EC%9A%B0%EC%84%A0%EC%88%9C%EC%9C%84%EC%97%90%20%EC%A3%BC%EC%9D%98%EA%B0%80%20%ED%95%84%EC%9A%94%ED%95%A9%EB%8B%88%EB%8B%A4,%EC%9A%B0%EC%84%A0%EC%88%9C%EC%9C%84%EA%B0%80%20%EB%86%92%EC%9C%BC%EB%82%98%20%EB%B9%84%EA%B5%90%20%EC%97%B0%EC%82%B0%EB%B3%B4%EB%8B%A8%20%EB%82%AE%EC%8A%B5%EB%8B%88%EB%8B%A4) [[47]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=if%20,%EA%B3%BC%20%EA%B0%99%EC%9D%8C) [[48]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%26%20AND%2C%20) [[49]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=) [[50]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%E2%97%86%20%EA%B0%99%EC%9D%80%20%EA%B0%92%EB%81%BC%EB%A6%AC%20XOR%ED%95%98%EB%A9%B4%200%EC%9D%B4,%EB%A7%8E%EC%9D%80%20%EA%B3%B3%EC%97%90%20%EC%A0%81%EC%9A%A9%ED%95%A0%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4) [[51]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%E2%97%86%20%EC%96%B4%EB%96%A4%20%EC%88%98%EC%97%90%EC%84%9C%20%EB%AA%87%20%EA%B0%9C%EC%9D%98,%EA%B8%B0%EB%B2%95%EC%9D%84%20%EC%9D%B4%EC%9A%A9%ED%95%9C%20%EB%8C%80%EC%86%8C%EB%AC%B8%EC%9E%90%20%EB%B3%80%ED%99%98%20%ED%95%A8%EC%88%98%EC%9E%85%EB%8B%88%EB%8B%A4) [[52]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%2F%2F%20%EC%95%9E%EC%97%90%EC%84%9C%EB%B6%80%ED%84%B0%20i%EB%B2%88%EC%A7%B8%20%EC%9B%90%EC%86%8C%EC%99%80%20%EB%92%A4%EC%97%90%EC%84%9C%EB%B6%80%ED%84%B0,i%EB%B2%88%EC%A7%B8%20%EC%9B%90%EC%86%8C%EB%A5%BC%20%EC%B6%9C%EB%A0%A5%ED%95%98%EB%8A%94%20%EC%BD%94%EB%93%9C) [[53]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=printf%28%22) [[54]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%E2%97%86%202%EC%9D%98%20%EA%B1%B0%EB%93%AD%EC%A0%9C%EA%B3%B1%20%EA%B3%B1%EC%85%88%2F%EB%82%98%EB%88%97%EC%85%88) [[55]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%2F%2F%20WARNING%20%EC%9D%8C%EC%88%98%EC%9D%BC%20%EB%95%8C%20%EC%A0%9C%EB%8C%80%EB%A1%9C,%EB%8F%99%EC%9E%91%ED%95%98%EC%A7%80%20%EC%95%8A%EC%9D%8C) [[56]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=void%20xor_swap%28int,) [[57]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=) [[58]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=2,%ED%91%9C%ED%98%84%EC%9D%84%20%EC%89%BD%EA%B2%8C%20%ED%95%A0%20%EC%88%98%20%EC%9E%88%EC%8A%B5%EB%8B%88%EB%8B%A4) [[59]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=%EC%82%AC%EB%9E%8C%EC%97%90%200,%EC%9D%B4%EB%9D%BC%EA%B3%A0%20%ED%95%A9%EC%8B%9C%EB%8B%A4) [[60]](file://file-4RyuT8646SpnRn1nCv7H6L#:~:text=A%2C%20B%20%EB%AA%A8%EB%91%90%EC%99%80%20%EC%B9%9C%EA%B5%AC%EC%9D%B8%20%EC%82%AC%EB%9E%8C%EC%9D%80%3F) 2.+비트연산.docx

<file://file-4RyuT8646SpnRn1nCv7H6L>

[[61]](https://advenoh.tistory.com/18#:~:text=,1%29%EB%A5%BC%20%EC%A0%95%EC%88%98%EA%B0%92%EC%9D%B4%200%EC%9D%B4%20%EB%90%A0%EB%95%8C%EA%B9%8C%EC%A7%80%20%EB%B0%98%EB%B3%B5%ED%95%9C%EB%8B%A4) Algorithm : 정수값에서 1이 설정된 bit를 카운트하기

<https://advenoh.tistory.com/18>

[[62]](https://www.upgrad.com/tutorials/software-engineering/java-tutorial/strings-in-java-vs-strings-in-cpp/#:~:text=Strings%20in%20Java%20vs%20Strings,contrast%2C%20C%2B%2B%20std%3A%3Astring%20is) Strings in Java vs Strings in C++: Key Differences Explained - upGrad

<https://www.upgrad.com/tutorials/software-engineering/java-tutorial/strings-in-java-vs-strings-in-cpp/>

[[63]](https://stackoverflow.com/questions/28442719/are-c-strings-mutable-unlike-java-strings#:~:text=) Are C++ strings mutable UNLIKE Java strings? - Stack Overflow

<https://stackoverflow.com/questions/28442719/are-c-strings-mutable-unlike-java-strings>