2020253091\_김호중

C++ Standard Template Library (Container, iterator, functor, Algorithm)

컨테이너 알고리즘 반복자 함수자로 구성되며 보다 효율적인 프로그래밍에 도움을 준다. 템플릿을 사용하기에 데이터 타입선택이 자유롭다. ++os에 따라 사용이 비효율적일 수 있다고 한다. 일반적인 알고리즘 풀이시에는 효율적이다.

C++ Memory Layout

C++의 가상 메모리는 다음과 같은 구조로 이루어져 있다. Code, Data, BSS, Heap, Stack 각각의 역할에 대해서 알아보면 Code영역은 함수를 정의하고 저장하며 Data영역은 초기화된 전역 및 정적 변수들을 저장하며 프로그램의 시작과 함께 할당되며 프로그램 종료시에 소멸한다. BSS는 초기화 되지 않은 전역 및 정적 변수들의 저장공간을 의미하고, Stack(LFIO last in first out 이렇게 부르는지 처음 알았다. 수업에서는 후입선출)은 함수와 연관되는 지역변수와 매개변수가 저장되는 영역이다. Heap은 메모리가 동적으로 할당되고 해제 되는 영역으로 메모리의 낮은 주소에서 높은 주소의 방향으로 할당된다. 설명에서 알 수 있듯이 메모리 영역에 따라 호출되는 순서와 목적이 각각 다르며 효율적인 배분을 도와준다.

C++ malloc/free

Malloc는 함수는 프로그램이 실행 중일 때 사용자가 직접 힙영역에 메모리를 할당할 수 있게 해준다. 할당 받고자 하는 인수를 바이트단위로 받으며 할당되지 않은 블록의 주소를 반환한다. 할당에 실패시에 null을 반환하며 주소 값을 반환하기에 포인터를 사용해야 하며 메모리 누수를 방지하기위해 할당된 메모리 해제가 필요하다. Free는 힙영역에 할당된 메모리를 운영체제로 반환해 주는 함수이다. 위에서 다루었듯이 힙영역에는 메모리가 동적으로 할당되며 런 타임 내내 메모리가 변한다 malloc 메모리 해제를 해주기 위해 사용한다.

Memory

메모리에는 두가지 종류가 있다. 첫째는 물리적 메모리 그리고 둘째는 가상 메모리 먼저 물리적 메모리에 대해서 다뤄보면 물리적 메모리란 RAM과 같이 컴퓨터의 하드웨어 파트를 담당하며 cpu와 gpu에 의해 읽고 써지는 메모리를 뜻한다. 가상 메모리란 컴퓨터공학에서 굉장히 중요한 개념으로 프로그래밍에 쓰이는 논리적인 구조다. 이렇게 두가지로 나뉘는 이유는 효율적인 분배를 위해서이다. 우리가 프로그램을 실행할시에 발생하는 프로세스의 크기는 물리적 메모리의 양을 한참 초과한다. 하지만 우리는 경험으로써 프로세스들의 처리가 가능함을 알고 있다. 이와 같은 일이 가능한 이유는 paging(위키 문서 읽어보았을 때 이해가 잘 되지 않았다. 6개월뒤에 다시 돌아오자) 덕분이다. Paging은 메모리 관리 체계로서 주메모리에서 사용하기 위해 보조 저장소에서 데이터를 저장하고 검색 하게한다. 이러한 이유로 우리는 프로그래밍시에 가상메모리에 조금 더 많은 신경을 쓰게된다. 실제 메모리 공간에서 작업을 진행하는 것이 아닌 프로세스마다 할당된 연속된 메모리 공간에서 작업을 진행하게 되며 이러한 역할을 해결하는 것이 바로 os이다.

C++ Vector

동적배열로 객체를 삽입하거나 제거할 때 자동으로 자신의 크기를 조정해준다. 일반적인 정적배열에 비하여 더많은 공간을 차지하며 추가 메모리가 소진될경우 재할당이 필요하다. 벡터연산에 대한 시간 복잡도는 다음과 같은데 random access의 경우 O(1) 끝 요소에 대한 입력과 출력 O(1) 끝이 아닌 요소에 대한 삽입 제거 O(n)이다.

C++ Map

노드 기반의 균형 이진 트리 구조를 띈다. Key와 Map로 구성되어 있으며 pair객체로 저장된다. Key와 value가 연결되는 것을 매핑 된다고 하며 하나의 value에 하나의 key만 매핑된다.(하나의 key에 여러 개의 value가능) 순서보다 key와 상호작용하는 자료구조를 원할 때 사용되며 대표적인 종류에는 HashMap TreeMap LinkedHashMap등이 있다. HashMap에 대해서만 간략하게 알아보면 key와 value가 mapping될뿐 쌍들에 대해서 순서를 보장할 수 없다.

Queue

먼저 들어간 데이터가 먼저 나오는 선입선출 자료구조이다.(FIFO first in first out 라고도 부른다) 스택과는 반대되는 개념이며 입력을 Enqueue 출력을 Dequeue라고 한다. Deque라는 특이한 구조가 있는데 이는 queue와는 다르게 양쪽에서 삽입 인출이 가능하며 톱니바퀴 같은 문제를 풀 때 유용하게 사용될 듯하다. 다른 특이한 형태의 queue중 원형 queue가 존재한다. 원형 queue는 기존의 큐의 공간이 꽉 차면 더 이상 요소를 추가할 수 없다는 점을 개선한 형태의 자료구조로서 enqueue를 하며 rear포인터가 앞으로 이동하고dequeue를 하면 front가 앞으로 이동한다. 솔직히 어디에 쓰일지 잘 모르겠다.

Tree

나무를 거꾸로 뒤집어 놓은 모양이며 노드로 이루어진 자료구조이다. 트리는 하나의 루트노드와 거기에 연결되는 0개 이상의 자식노드를 가진다 루트노드에서 파생되어 또 다시 0개 이상의 자식노드가 생기며 이를 반복하는게 트리 자료구조이다. 노드들은 edge로 연결되어 있으며 다른 자료구조에 비해 형태가 자유롭다 생각한다. 각 노드에는 하위노드에 대한 포인터를 가지고 있으며 고유의 값 또한 존재한다. 한 노드 A를 생각해보자 이 노드에서 파생된 2개의 노드 B,C에 대하여 A는 B,C의 부모이며 B,C는 A의 자식이다. B와 C는 형제이며 루트노드에서 어떤 특정한 노드를 만나기 위해 거쳐야하는 edge를 depth라고 하며 어떤 특정한 노드에서 다른 가장 먼노드 까지의 거리를 height이라고한다. 이 밖에도 루트에서 어떤 특정한 노드들까지의 간선수를 Level이라 하며 한 레벨에 있는 노드의 수를 넓이 두 노드사이의 최단 경로를 거리라고 부른다. 트리 자료구조의 특징은 linked된 무 방향 그래프 구조이며 모든 자식은 하나의 부모를 가지고 n개의 노드로 구성되어 있으면 n-1개의 간선을 가진다. 데이터를 계층적으로 저장할 때 사용되며 효율적인 검색을 위해 사용되기도 한다.

Greedy Algorithm

현재 상황 t에서 목표에 대해 가장 가까운 것을 선택하는 알고리즘을 의미한다. Greedy라는 이름에 걸맞은 알고리즘이며 순간순간의 상황에서 가장 좋은 선택을 했을 때가 결국에 최적의 결과로 이어질 때 사용하는 알고리즘이다. 현실의 상황에서는 그리 좋은 알고리즘이라 말할 수 없지만 알고리즘에서 만큼은 강력한 알고리즘이다. 몇가지 예를 들어보자 N원을 거슬러 주어야 할 때 최소로 사용한 동전의 개수를 구하는 문제를 살펴보았을 때 N에 대하여 500원으로 나누고 그 나머지를 100원으로 나누고 그 나머지를 마지막으로 50원으로 나누는 것으로 문제 해결이 가능하다. 또 다른 예시로는 어떤 자연수 n에 대하여 2가지 행동을 취할 수 있다 1을 빼거나 k로 나누거나 이때에 최소 횟수를 구하는 문제에서 또한 그리드 알고리즘으로 접근할 수 있다. N에 대하여 일단 나누어 보고 안되면 1을 빼는 식으로 알고리즘을 구현 가능하다.

DFS Algorithm (Depth first search Algorithm)

깊이우선탐색 알고리즘을 의미하며 그래프에서 가장 깊은 부분을 먼저 탐색하는 알고리즘이다. 대표적으로 스택을 활용하는 방법 재귀 함수를 활용하는 방법이 존재한다. DFS의 과정을 알아보면 탐색 시작 노드를 스택에 삽입하고 방문 처리한다. 시작 노드와 연결된 노드를 선택하고 방문처리한다. 다시 선택한 노드와 연결할 노드를 선택 그 노드를 방문처리 한다. 위 과정에서 특정 노드에 대해 인접한 노드 중 더 이상 방문할 노드가 없다면 스택에서 pop연산을 진행한후 전 과정을 반복한다.

BFS Algorithm (Breadth first search Algorithm)

DFS와 비슷한 느낌의 알고리즘으로 너비우선탐색 알고리즘이다. 과정을 한번 살펴보면 시작 노드를 하나 정하고 그 노드를 queue에 넣는다. 그후 큐에서 노드를 꺼내고 해당 노드와 인접해 있는 노드 중 방문하지 않은 노드를 queue에 집어넣는다. 위 과정을 반복이 불가능 할 때까지 반복하면 된다. BFS DFS 모두 코딩테스트에서 자주 출제되는 알고리즘이라 한다.

BIG-O

Big o는 알고리즘의 시간 복잡도를 표현한다. 천천히 예시와 함께 알아보자 O(1)의 경우 데이터의 수와 종류에 관계없이 항상 일정한 시간이 걸리는 알고리즘을 말하며 예로 앞에서 서술한 vector의 접근이 여기에 속한다. O(N)은 그냥 for문 한번 돌리면 O(N)이다. 순차적으로 입출력 할때에 이런 복잡도를 가진다. O(N\*\*2)도 비슷한데 for문 2번돌리면 O(N\*\*2)이다. O(M\*\*N)피보나치 등등 이있다. O(log n) 처음에 공부했을 때 조금 아리까리 헀던 표기다. 바이너리 서치로 쉽게 이해해보자 16개의 데이터가 있을 때 반으로 나눈다. 나눈 후에 타깃과 비교하며 반토막난 덩어리 중 하나를 선택 위 과정을 반복한다. 16개의 데이터가 들어왔을 때 최대 log2 16 번 연산을 진행하며 여기에서 나타난 것이 O(logn)이다. 쉽게 정리해보면 1회 연산을 진행했을 때 로그의 밑 만큼 데이터가 줄어드는 효과를 낸다면 시간 복잡도가 logn 이다.

Hash Table

이론은 굉장히 간단하다 인풋값을 해쉬화 시키고 그것을 저장한다. 저장 값을 리턴하고 싶을 때 키를 입력으로 저장 값을 얻어온다. 굉장히 간단한 원리이지만 프로그래밍에서 굉장히 중요한 요소다. Hash Table에 대해 자세히 알아보자 인풋 데이터를 문자라고 가정하자 apple이 들어 갔을 때 우리는 해쉬함수를 통하여 해쉬화를 진행한다. 그후 결과값을 나머지 연산을 통해 index에 분류하게 되는데 이런 식으로 할당할 때에 key값에 대응하는 데이터를 쉽게 찾을 수 있으며 삭제와 저장연산 또한 O(1)으로 굉장히 빠르다. 이런 Hash Table에도 치명적인 문제가 존재하는데 충돌문제다. Hash된 값을 index에 할당하시에 이미 데이터가 존재하는 곳에 할당이 될 수 있는데 이런 문제를 충돌이라고 하며 이를 해결하기 위해 몇 가지 방법이 존재한다. 첫째는 충돌하는 index에 트리, linked list등의 자료구조를 사용하여 index를 할당하는 방법 둘째는 충돌하는 index 이후 index를 서치해 가며 비어 있는 곳을 찾아 그곳에 할당하는 방법 이 두가지가 존재한다.

Dynamic Programming

하나의 문제를 작은 여러 문제로 나누어 해결할 수 있을 때 다이나믹 프로그램을 활용하며 간단하게 시간 복잡도를 개선할 수 있는 하나의 아이디어다. 대표적으로 피보나치 문제가 존재하는데 일반적인 피보나치를 구현하기 위해서는 기하급수적인 즉 2\*\*n의 시간 복잡도를 가지지만 피보나치 수열의 수를 그때그때 계산하는 것이 아닌 전단계의 수를 기억해 둠으로써 복잡도를 개선해 나갈 수 있다. f(n-1)+f(n-2)=f(n)이라는 식에서 f(n)값과 f(n-1)값을 저장함으로써 다음단계의 계산을 보다 쉽게 하고자 하는 아이디어이다. 결국에 이런 방식으로 문제를 해결했을 때에 시간 복잡도는 O(n)으로 비교도 못할 정도로 빨라졌다.

Dijkstra Algorithm

최단경로 알고리즘중 대표적인 알고리즘으로 특정한 노드에서 출발하여 다른 모든 노드로 가는 최단경로를 계산한다. 음의 간선이 포함되지 않았을 때 정상적으로 작동하며 Greedy 알고리즘 중 하나이다. 동작 과정을 살펴보면 임의의 시작 노드에서 가장 짧은 거리의 노드를 선택 그 노드에서 다시 가장 가까운 노드를 선택해 나가며 목적 노드까지의 거리를 테이블에 저장하며 위과정을 반복한다. 하나하나 예시를 들어보자 시작노드에서 갈 수 있는 주변 노드의 거리를 업데이트 한다. 이제 주변노드를 차례대로 탐색하기 시작한다 예를 들어 시작노드 a 주변에 노드 b, c, d가 있다 가정했을 때 a->b를 통해서 이동 할 수 있는 노드들에 대한 거리를 측정하고 테이블에 저장된 값들과 비교하여 만약 새로 측정된 값이 더 작을 경우 값을 새로운 값으로 업데이트한다. 다음으로는 a->b->?를 통해 연산을 진행하고 더 이상 방문하지 않은 노드가 없을 때에 전 단계로 돌아와 연산은 계속 진행하게 된다. 결론적으로 우리는 모든 경우의 수를 통해 시작노드에서 갈수 있는 모든 노드에 대한 최단거리를 알 수 있다. Priority queue를 사용하여 알고리즘의 시간 복잡도를 줄일 수 있다. 기존의 프로세스대로 처리를 할 경우 o(n\*\*2)의 복잡도를 가지는 반면에 우선순위 queue를 활용할 경우 O(Elogv)를 가진다. 처음 안 사실인데 log N\*\*2==lon n 이다.

Algorithm

문제에서 자주 사용하는 라이브러리와 핵심원리의 초석을 깔았다. 기본 예제를 통해서 어떤 느낌으로 사용되는지는 감을 얻었지만 이것을 실제 문제에서 떠올리고 사용할 수 있도록 문제를 유형화 시키는 작업이 필요할 듯하다. 특히 BFS&DFS는 예제 수준의 문제는 어렵지 않게 구현 할 수 있었지만 조금 난이도가 올라가자 기겁을 할 정도의 난이도를 보여 주었다. 이와 별개로 메모리에 대한 이해와 정리하지 않은 자료구조 또한 굉장히 중요함을 깨달았다. 테스트 문제에 자주 사용되는 수학적인 개념들과 아이디어에 대한 정리가 1순위로 필요할 듯 하다.

출처

<https://tcpschool.com/c/c_memory_malloc>

https://ko.wikipedia.org/wiki/%ED%91%9C%EC%A4%80\_%ED%85%9C%ED%94%8C%EB%A6%BF\_%EB%9D%BC%EC%9D%B4%EB%B8%8C%EB%9F%AC%EB%A6%AC

<https://cpp.tech-academy.co.uk/memory-layout/>

<https://yoongrammer.tistory.com/68>

<https://blockdmask.tistory.com/87>

이것이 코딩테스트다

YouTube channel

동빈나

코드없는 프로그래밍

포프Tv