#algorithms #datastructure

연결리스트

왜 나왔나?

- 메모리 위에는 수십 개의 프로그램이 돌아간다. 그리고 각 프로그램마다 스택 영역도 보유하고 있으며 힙 영역도 존재한다. 이러한 상황에서 테트리스처럼 배열로 메모리 공간을 점유하면 어떻게 될까?
- 테트리스처럼 중간에 빈 공간이 채워지지 않는다. 차라리 테트리스는 한 줄 채우면 없어지기라도 하지 프로그램은 종료할 때까지 유지된다. 그렇기에 효율적으로 저장공간을 활용하기 위해 배열과 달리 물리적으로 메모리를 연결하지 않는 방식을 만들었다.

배열과 다른 점

- 배열과 달리 값들이 순차적으로 메모리에 안 쌓여 있다. 그렇기에 중간 중간 빈 공간을 점유해서 효율적으로 활용 가능
- 별도의 주소 테이블이 없으면 값을 탐색하는 데 오래 걸림. 배열은 세 번째 원소 찾으라고 하면 첫번째 원소에서 주소값을 +2 하면 그만이지만 연결리스트는 주소값이 물리적으로 세번째 위치에 있을 지 보장해주지 않는다. 그렇기에 다음 원소의 주소값을 찾아가며 세 번째 위치까지 하나 하나 확인해보는 수 밖에 없다.

해시

왜 나왔나?

- 배열은 인덱스가 있고 인덱스를 찾으면 값을 얻는다. 그렇기에 내가 원하는 대상의 인덱스를 모르면 결국 순차적으로 뒤져가며 찾아가야 함. 또한, 안다고 해도 이를 전부 머릿속에 인지해야 하는 번거로움이 존재
- 내가 원하는 키 값을 넣으면 자동으로 함수가 처리해서 키 값에 맞는 인덱스가 나와서 값을 찾을 수 있으면 어떨까?

정의

- 특정 key 값을 입력하면 해시 함수를 통해 지정한 인덱스 값으로 변환하는 자료구조

특징

- 단방향 : 키를 통해 값을 찾을 수 있으나 값을 통해 키를 찾을 수는 없다
- 참조는 O(1) : 인덱스만 찾으면 바로 값을 참조할 수 있으니까

참고 용어

- 해시 테이블(hash table) : 값이 저장된 배열

- 버킷(bucket) : 해시 테이블에 있는 단일 데이터. 3번 인덱스 = 3번 버킷.

해시 함수 구현

- 고려사항
  - 해시 함수 변환값은 인덱스로 쓴다. 그렇기에 변환값이 해시 테이블(값이 저장된 배열) 크기보다 작아야 한다
  - 다른 키 값을 넣었는데 변환값(인덱스)이 동일하게 나오는 충돌 상황을 최대한 줄여야 한다
- 자주 사용하는 해시 함수
  - 나눗셈법 : 키를 소수 k로 나눈 나머지를 변환값으로 사용한다
    - 소수를 사용하는 이유는 약수의 개수가 적어서 동일한 나머지가 나오는 수가 적기 때문이다
    - 나머지는 아무리 크게 나와도 (k - 1)이므로 k값만 조절하면 해시 테이블 크기에 최대한 맞춰볼 수 있다. (물론, 해시 테이블 크기에 최대한 맞는 소수가 없으면...)
  - 곱셈법 : 나머지 연산에 황금비를 사용하여 변환값을 구한다
    - h(x) = (((x * A) mod 1) * m)
    - 키 값 x에 황금비 A를 곱하고 1로 나누어 나머지인 소수부만 가져간다. 소수부 값에 해시 테이블 크기를 곱하고 이 값에서 정수 부분만 취해 변환값을 만든다. 즉, (0 ~ 0.99999 ...) * m 이렇게 곱하는 셈이기에 0 ~ m - 1 사이의 값이 나온다.
  - 문자열 해싱 : 키의 자료형이 문자열일 때 해시 테이블 인덱스로 변환하는 방법
    - h(s) = (s[0] + s[1]*p + s[2]*p^2 + ... + s[n - 1] * p^(n - 1)) mod m
    - s는 문자 집합을 숫자로 변환하기 위해 만든 배열이다. 배열의 인덱스 값은 숫자기에 당연하게도 문자를 숫자로 변환해야 계산할 수 있다. 그렇다면 각 문자마다 어떤 숫자로 변환할지 표를 만들어야겠죠? 네 그렇습니다
    - p는 메르센 소수다. 메르센 소수는 2^n - 1 형식으로 표시할 수 있는 숫자 중 소수인 수이며 해시 충돌에 효과적이다.
    - 그렇다면 문자 집합 변환 배열이 커질 수록, 메르센 소수가 커질 수록 m으로 나누기 전에 나오는 결과값이 기하급수적으로 커질 것이다. 오버 플로우는 어떻게 막을 수 있을까?
- 충돌(collision)
  - 키 값이 다른 데 해시 함수를 통과한 변환값이 같은 경우를 이른다. 이러한 충돌을 막기 위한 방법은 아래와 같다.
  - 체이닝(chaining)
    - 변환값이 같으면 따로 연결 리스트로 연결하여 하나의 변환값이 복수의 데이터를 갖게 하는 방법
    - 간단한 방법이지만, 충돌이 많이질 수록 연결 리스트 길이도 길어져서 해시 테이블 공간 활용성이 떨어진다. 그리고 중복 데이터 중에 동일값 여부를 판정해서 연결 리스트 길이를 늘릴 지 말 지도 봐야하므로 중복 데이터가 많을 수록 탐색하는 데 더욱 시간이 걸린다.
  - 개방주소법(open addressing)
    - 충돌하면 빈 버킷 값을 찾아 충돌값을 삽입한다. 이 때, 빈 버킷을 찾는 방식은 선형 탐사와 이중 해싱이 있다.
    - 선형 탐사 : 충돌이 발생하면 다른 빈 버킷을 찾을 때까지 일정한 간격으로 이동하는 방식

        - h(k, i) = (h(k) + i) mod m
        - 키 값(k)에 해당하는 해시 테이블의 인덱스를 구한 뒤에 만약 충돌이 발생하면 일정 간격을 누적(i)으로 더해주고 테이블을 넘어갈 걸 고려해서 해시 테이블 크기(m)만큼 나머지 연산을 실시한다
        - 충돌 발생하면 간격만큼 이동해서 충돌이 발생하지 않는 곳까지 가면 되는 간단한 방법이지만, 해시 충돌이 일어나는 값끼리 군집(cluster)을 이룰 가능성이 있다. 군집이 커질 수록 해시값이 겹칠 확률이 올라가는 게 단점.
    - 이중 해싱 : 해시 함수를 N개(여기서는 이중이니까 2개) 사용하여 군집 형성을 피하고자 만들어진 방식
        - h(k, i) = (h1(k) + i * h2(k)) mod m
        - 첫번째 해시 테이블(h1)에서 충돌이 발생하면 i 만큼 간격을 벌리되, 두번째 해시테이블(h2) 값을 곱하여 최대한 군집이 모이는 걸 회피한다. 이 때, m은 제곱수나 소수를 채택하여 클러스터 형성을 더욱 줄일 수 있다.

사용 예시

- 비밀번호 관리, 데이터베이스 인덱싱, 블록체인

풀어보자

- 완주하지 못한 선수, https://school.programmers.co.kr/learn/courses/30/lessons/42576
- 메뉴 리뉴얼, https://school.programmers.co.kr/learn/courses/30/lessons/72411

트리

왜 나왔나?

- 디렉터리나 파일 시스템을 생각해 보자. 디스크에서 시작해서 각 폴더로 나뉘고 또 폴더 내부에 다른 폴더와 데이터가 이어지는 연속적인 구조를 어떻게 나타낼까? 배열로? 연결리스트로?
- 위와 같이 최상위 폴더를 중심으로 마치 가지가 뻗듯이 폴더 안에 폴더와 데이터들이 있는 걸 계층 구조라 한다. 이러한 계층 구조를 나타내기 위해서는 배열이든 연결리스트든 사용할 수 있다. 그렇다면 이를 사용해서 어떻게 계층 구조를 표현할 수 있을까?

정의

- 최상위 단일 원소를 기점으로 하여 가지가 뻗듯이 퍼져 나가는 비순환 자료구조.

특징

- 트리는 최상위 원소는 단 1개다.
- 트리는 순환하지 않는다. 즉, 단계가 낮은 원소가 자신보다 높은 단계에 있는 원소로 가는 길이 있으면 순환이기에 트리가 아니다.

참고 용어

- 노드(node) : 트리에 저장된 각각의 원소들을 말한다.
- 간선(edge) : 노드와 노드를 잇는 선. 배열이든 연결리스트든 다음 노드로 가기 위한 주소값이라 생각하면 편하다.
- 최상위 노드(root node) : 트리의 정점. 모든 가지가 뻗어나가기 시작하는 노드
- 부모 노드(parent node) : 어느 노드보다 한 단계 위에 있는 노드
- 자식 노드(child node) : 어느 노드보다 한 단계 아래에 있는 노드
- 형제 노드(sibling node) : 같은 부모를 갖는 노드
- 이파리 노드(leaf node) : 트리의 말단. 자식 노드가 없는 노드
- 차수(degree) : 최상위 노드를 기점으로 몇 단계에 위치하는 지 나타내는 수

이진 트리

- 이진 트리는 부모 노드에게 주어진 자식 노드가 최대 2개인 트리이다. 컴퓨터의 이진수 체계와 같이 어떠한 기준에 대하여 적합/부적합으로 나뉘기에 적용하기 쉽고 특정값 탐색 시 O(log n)이 걸리는 장점이 있다.
- 표현 방식
  - 배열 : 최상위 노드를 배열 인덱스 1에 저장. 왼쪽 자식 노드는 부모 노드 배열 인덱스 * 2, 오른쪽 자식 노드는 부모 노드 배열 인덱스 * 2 + 1로 구분한다.
  - 연결 리스트 : 각 노드마다 왼쪽과 오른쪽 주소값을 저장하면 된다. 본인이 자신있어 하는 프로그래밍 언어의 STL에서 특별히 연결리스트 방식을 지원하는 게 없으면 직접 만들어야 한다.
  - 인접 리스트
    - 인접 리스트는 인덱스별 배열 원소 개수가 다를 때 사용한다. 즉, 인덱스 주소는 배열처럼 이어져 있지만 각 인덱스가 가리키는 값들은 길이가 다른 배열의 주소값이다.
    - 트리에서 인덱스는 각 차수를 뜻하며 차수별로 형제 노드의 주소값들을 배열처럼 저장한 형태다.
- 순회 방식
  - 순회란 한 바퀴를 돈다는 의미다. 트리에 속한 모든 원소들을 다 방문하는 방법으로는 전위, 중위, 후위 순회가 있다.
  - 전위(preorder)
    - 현재 노드 -> 왼쪽 자식 노드 -> 오른쪽 자식 노드 순서로 원소를 확인한다
    - 트리를 복제할 때 많이 사용한다
  - 중위(inorder)
    - 왼쪽 자식 노드 -> 부모 노드 -> 오른쪽 자식 노드 순서로 원소를 확인한다
    - 정렬된 순서대로 값을 가져올 때 사용한다
  - 후위(postorder)
    - 왼쪽 자식 노드 -> 오른쪽 자식 노드 -> 부모 노드 순서로 원소를 확인한다
    - 트리를 삭제할 때 자주 사용한다

```
// 노드 방문 여부 확인 배열
visited_node[node_num]
```

```
// 전위 순회
preorder(cur_node) {
    // 노드를 방문했으면 넘긴다
    if (visited_node[cur_node])
        return ;
    // 노드 방문 확인 표시
    visited_node[cur_node] = true;
    // 왼쪽 자식 노드 확인
    preorder(left_child_node);
    // 오른쪽 자식 노드 확인
    preorder(right_child_node);
}

// 중위 순회
inorder(cur_node) {
    // 왼쪽 자식 노드 방문 여부 확인
    if (!visited_node[left_child_node])
        inorder(left_child_node);
    // 현재 노드 방문 확인 표시
    visited_node[cur_node] = true;
    // 오른쪽 자식 노드 방문 여부 확인
    if (!visited_node[right_child_node])
        inorder(right_child_node);
}

// 후위 순회
postorder(cur_node) {
    // 왼쪽 자식 노드 방문 여부 확인
    if (!visited_node[left_child_node])
        postorder(left_child_node);
    // 오른쪽 자식 노드 방문 여부 확인
    if (!visited_node[right_child_node])
        postorder(right_child_node);
    // 현재 노드 방문 확인 표시
    visited_node[cur_node] = true;
}
```

- 이진 트리 구축 방법
  - 이진 트리는 현재 노드를 기준값으로 하여 크다/작다로 구분하여 자식 노드를 배치한다. 보통은 작은 값을 왼쪽 자식 노드에 놓고 크거나 같은 값을 오른쪽 자식 노드에 놓는 경향이 있다.
- 이진 트리 탐색 방법
  - 이진 트리 구축 시, 현재 노드를 기점으로 크거나 작거나를 구분했다. 그렇기에 탐색할 목표값이 있으면 해당값이 현재 노드보다 크냐 작냐를 구분해서 그에 따른 자식 노드로 옮겨가면서 찾으면 된다.

이진 트리 탐색과 시간복잡도

- 이진 트리에서 자식 노드가 균일하게 배치된 경우에는 어느 한 쪽을 택할 때마다 나머지 절반의 경우의 수가 줄어든다. 그렇기에 O(log n)의 시간복잡도를 가진다.

```
이해가 안 되시면 봅시다.
84개의 노드가 균일하게 분포한 이진 트리가 있다고 가정하자.
1. 최상위 기준 왼/오른쪽 택하기(오른/왼쪽 42개가 제외)
2. 현재 노드 기준 왼/오른쪽 택하기(오른/왼쪽 21개가 제외)
3. 현재 노드 기준 왼/오른쪽 택하기(오른/왼쪽 10개가 제외)
...
84 -> 42 -> 21 -> 11 -> 6 -> 3 -> 2 -> 1 (최대 7번 만에 찾는다)
2^6 < 84 < 2^7
위 식을 이진로그로 나타내보자
6 < log 84 < 7 (컴퓨터에서는 이진로그 표현할 때 보통 log 밑에 2를 생략합니다)
```

- 그렇다면 한 쪽에 전부 치우쳐져 있을 때는? 불행하게도 O(N)이다. 왜냐하면 전부 왼쪽 자식 노드로 이어져 있으면 일렬로 세운 거랑 다를 게 없기 때문이다. (1 -> 2 -> 3 -> ... -> 84)

균형 이진 탐색 트리

- O(N)이 되는 걸 막고자 최대한 균일하게 이진 트리를 구성하기 위해 조성한 방식
- 종류 : AVL 트리, 레드 블랙 트리 등
- 이 부분은 난이도가 높아서 참고자료로 별도 파일로 배포함

풀어보자

- 예상 대진표, https://school.programmers.co.kr/learn/courses/30/lessons/12985
- 다단계 칫솔 판매, https://school.programmers.co.kr/learn/courses/30/lessons/77486
- (난이도 상) 길 찾기 게임, https://school.programmers.co.kr/learn/courses/30/lessons/42892