Data Structure

#1

linked list

linked list는 서로 떨어져 있는 데이터를 메모리 주소를 참조함으로써 이어진 것처럼 사용할 수 있다. linked list는 구조체가 이어진 형태로 존재하며, 이 구조체를 `노드`라고 부른다. 노드는 값을 담고 있는 `데이터 필드`와 다음 구조체를 가리키는 `링크 필드(포인터)`로 구성된다. 정확히는, 포인터가 다음 구조체의 주소를 담고 있다. 보통, linked list의 맨 첫 원소를 가리키는 head 포인터와 맨 마지막 원소를 가리키는 tail 포인터를 통해, 리스트의 요소에 접근하거나 수정한다. linked list는 구현 방법에 따라, single linked list와 double linked list, circular linked list 등으로 구분할 수 있다.

Array vs. Linked list

array에서 중간에 값을 삽입하고 싶다면, 삽입할 위치 뒤의 모든 데이터가 한 칸씩 이동해야 한다는 단점이 있다. 또한, array가 할당받은 공간이 부족한데, 메모리상 뒤쪽의 메모리가 비어있지 않으면, 모든 데이터가 더 큰 홀로 이사를 가야 한다는 단점도 존재한다. linked list는 연속된 공간을 사용하지 않아도 되기 때문에 앞서 언급했던 array의 단점을 모두 해소할 수 있다. 그러나 N번째 데이터를 참조하고 싶을 때, 계속해서 다음 주소를 참조하는 형식으로 따라가야 하므로 메모리 참조에 시간이 오래 걸린다. 따라서 삽입과 삭제가 빈번한 경우에는 linked list를 사용하고, 참조가 빈번하게 일어날 때는 array를 쓰는 것이 바람직하다.

Time Complexity 비교

구분, Array, Linked List

`insert`, $O(n)$, $O(1)$

`find`, $O(1)$, $O(n)$

삽입, 삭제, 접근 방법

- 접근: 원하는 원소가 나올때까지 링크 필드(포인터, 다음 노드)를 계속해서 탐색한다.
- 삽입: 삽입할 노드의 next에 현재 위치의 next를 연결한 후, 현재 위치의 next에 삽입할 노드의 주소를 넣어준다.
- 삭제: 삭제할 노드의 next를 앞선 노드의 next에 연결해준다.

기본적인 linked list 구조 (=Single linked list 구조)

#2

single linked list

#1 Linked list에서 언급한 내용은 모두 Single linked list에 해당한다. Single linked list는 linked list 중에서도 가장 기본적인 구조로 되어 있으며, head에서 tail까지 단방향으로 포인터가 이어져 있으므로 N 번째 노드에서 N-1 번째 노드에 접근할 수 없다. 대신, 다시 head로부터 N-1 번의 탐색을 통해 접근해야 한다.

References

- 연결 리스트의 개념과 종류 - suitepotato : https://velog.io/@suitepotato/00007

#3

double linked list

#1-1 Single linked list은 단방향 연결이기 때문에 한번 다음 노드로 이동하면, 이전 노드로 돌아가기 힘들다는 단점이 있었다. 그러나 Double linked list는 뒤의 노드의 주소뿐만 아니라, 이전 노드의 주소도 담고 있다. 하나의 노드는 하나의 데이터와 두 개의 링크를 가지고 있으며, 각각의 링크를 prev와 next라고 부른다. 다음 노드를 참조하고 싶다면 next 링크가 담고 있는 주소를 확인하면 되고, 이전의 노드를 참조하고 싶다면 prev 링크가 가지는 주소를 확인하면 된다.

References

- 연결 리스트의 개념과 종류 - suitepotato : https://velog.io/@suitepotato/00007

#4

circular linked list

앞서 언급했던 linked list 유형들과는 다르게, tail이 다시 head를 가리키는 구조를 가지고 있다. 따라서, tail 노드의 next에는 NULL이 들어가는 것 대신, head의 주소가 들어간다.

References

- 연결 리스트의 개념과 종류 - suitepotato : https://velog.io/@suitepotato/00007

#5

hash table

해시 테이블은 (Key, Value)로 데이터를 저장하는 자료구조 중 하나로 빠르게 데이터를 검색할 수 있는 자료구조이다. 해시 테이블이 빠른 검색속도를 제공하는 이유는 내부적으로 배열(버킷)을 사용하여 데이터를 저장하기 때문이다. 해시 테이블은 각각의 Key값에 해시함수를 적용해 배열의 고유한 index를 생성하고, 이 index를 활용해 값을 저장하거나 검색하게 된다. 여기서 실제 값이 저장되는 장소를 버킷 또는 슬롯이라고 한다.

예를 들어, `(Key, Value)쌍` 구조를 가지는 데이터 `("John Smith", "521-1234")`를 크기가 16인 해시 테이블에 저장한다고 하자. 그러면 먼저 `index = hash_function("John Smith") % 16` 연산을 통해 index 값을 계산한다. 그리고 `array[index] = "521-1234"` 로 value를 저장하게 된다. 이러한 구조로 데이터를 저장하면 Key값으로 데이터를 찾을 때 해시 함수를 1번만 수행하면 되므로 매우 빠르게 데이터를 저장/삭제/조회할 수 있다. 해시테이블의 평균 시간복잡도는 O(1)이다.

해시(Hash)값이 충돌하는 경우

만약 "John Smith"를 해시 함수를 돌려 나온 값과 "Sandra Dee"를 해시 함수를 돌려 나온 값이 동일하다면, 아래와 같이 해결할 수 있다.

해결방법 1: Separate Chaining(분리 연결법)

동일한 버킷의 데이터에 대해 자료구조를 활용해 추가 메모리를 사용하여 다음 데이터의 주소를 저장하는 방법이다. 동일한 해시 값을 가지면, 동일한 버킷 안에 엔트리를 할당해줘야 한다. 이 때, 버킷 내부의 엔트리 값들은 linked list 형태로 이어준다. 이러한 Chaining 방식은 해시 테이블의 확장이 필요없고 간단하게 구현이 가능하며, 손쉽게 삭제할 수 있다는 장점이 있다. 하지만 데이터의 수가 많아지면 동일한 버킷에 chaining되는 데이터가 많아지며 그에 따라 캐시의 효율성이 감소한다는 단점이 있다.

해결방법 2: Open Addressing(개방주소법)

Open Addressing이란 추가적인 메모리를 사용하는 Chaining 방식과 다르게 비어있는 해시 테이블의 공간을 활용하는 방법이다. Open Addressing을 구현하기 위한 대표적인 방법으로는 3가지 방식이 존재한다.

- **Linear Probing**: 현재의 버킷 index로부터 고정폭 만큼씩 이동하여 차례대로 검색해 비어 있는 버킷에 데이터를 저장한다.
- **Quadratic Probing**: 해시의 저장순서 폭을 제곱으로 저장하는 방식이다. 예를 들어 처음 충돌이 발생한 경우에는 1만큼 이동하고 그 다음 계속 충돌이 발생하면 2^2, 3^2 칸씩 옮기는 방식이다.
- **Double Hashing Probing**: 해시된 값을 한번 더 해싱하여 해시의 규칙성을 없애버리는 방식이다. 해시된 값을 한번 더 해싱하여 새로운 주소를 할당하기 때문에 다른 방법들보다 많은 연산을 하게 된다.

충돌을 방지하는 방법들은 데이터의 규칙성(클러스터링)을 방지하기 위한 방식이지만 공간을 많이 사용한다는 치명적인 단점이 있다. 만약 테이블이 꽉 차있는 경우라면 테이블을 확장해주어야 하는데, 이는 매우 심각한 성능의 저하를 불러오기 때문에 가급적이면 확정을 하지 않도록 테이블을 설계해주어야 한다. (통계적으로 해시 테이블의 공간 사용률이 70% ~ 80%정도가 되면 해시의 충돌이 빈번하게 발생하여 성능이 저하되기 시작한다고 한다.) 또한 해시 테이블에서 자주 사용하게 되는 데이터를 Cache에 적용하면 효율을 높일 수 있다. 자주 hit하게 되는 데이터를 캐시에서 바로 찾음으로써 해시 테이블의 성능을 향상시킬 수 있다.

시간 복잡도

삽입, 삭제, 탐색에 대해 해시 충돌이 일어나지 않는 경우에 $O(1)$, 충돌이 일어난다면 최악의 경우에 $O(N)$의 시간 복잡도를 가진다.

왜냐하면 해시 충돌로 인해서 하나의 버킷에 여러 엔트리가 연결되어있는 경우에 모든 엔트리를 탐색해야할 수 있기 때문이다.

References

- [자료구조] 해시테이블(HashTable)이란? - MangKyu's Diary : https://mangkyu.tistory.com/102

#6

stack

LIFO (Last In First Out) 구조의 자료형으로 한 쪽으로만 데이터를 넣고 뺄 수 있다. `push` 명령으로 데이터를 넣고, `pop` 명령으로 가장 마지막에 들어간 데이터를 빼낸다.

stack 은 브라우저의 뒤로가기 기능, ctrl + z (되돌리기), 지역 변수와 매개변수를 저장하는 stack 메모리 등에 사용된다. 이외에도 DFS 알고리즘 등 다양한 곳에 사용되는 자료형이다.

stack 에 데이터가 꽉 차서 더 넣을 공간이 없는데 데이터를 push 하는 경우 `overflow`, 반대로 데이터가 없는데 pop 하는 경우를 `underflow` 라고 한다.

References

- [자료구조] 스택, 큐는 무엇인가? - 마이구미 : https://mygumi.tistory.com/357

#7

queue

FIFO (First In First Out) 구조의 자료형으로 출구(front)와 입구(rear or back)가 따로 존재하여 먼저 입력된 데이터가 먼저 반환된다.

`push` 명령으로 rear 에 자료를 넣는다. rear += 1 되어 다음에 데이터를 받을 메모리를 가리켜야 한다. `pop` 명령으로 front 에서 데이터를 빼낸다. front += 1 되어 다음에 데이터를 반환할 메모리를 가리켜야 한다.

queue 는 CPU 연산처리 작업대기, 프린터 인쇄, 프로세스 관리 등 들어온 순서를 보장해야 하는 경우 사용된다. 이외에도 BFS 알고리즘 등에 사용된다.

queue 의 rear 가 기리키는 공간에 데이터가 있는데 데이터를 push 하는 경우 `overflow`, 반대로 front 가 가리키는 공간에 데이터가 없는데 pop 하는 경우를 `underflow` 라고 한다.

References

- [자료구조] 스택, 큐는 무엇인가? - 마이구미 : https://mygumi.tistory.com/357

#8

circular queue

크기가 N 인 queue 에서 모든 원소를 다 채우면 rear 는 N-1 을 가리킨다. 이 때, pop 으로 제일 처음 원소를 제거하면 queue 에 남은 공간 1개가 생긴다. 하지만 rear 는 마지막을 가리키고 있기 때문에 더이상 원소를 추가할 수 없다. 이 문제를 해결하기 위해 원형 형태의 `circular queue` 를 사용한다. queue 와 같이 FIFO 구조의 자료형이다.

동작 방식은 다음과 같다.

- 처음에는 front 와 rear 가 같은 메모리를 가리킨다.
- 데이터를 입력하기 위해 rear 는 메모리가 꽉찼는지 검사한다. 꽉찬 경우는 rear 다음 번의 메모리가 front 를 가리키는 경우 (rear + 1 == front) 인데, 꽉차지 않았다면 데이터를 입력하고 rear 는 다음 메모리로 이동한다.
- 데이터를 반환하기 위해 front 는 메모리가 비었는지 검사한다. 빈 경우에는 현재 front 위치와 rear 위치가 같은 경우 (rear == front) 인데, 비지 않았다면 데이터를 반환하고 front 는 다음 메모리로 이동한다.

References

- [자료구조] 큐(QUEUE)와 원형큐(CIRCULAR QUEUE) 개념과 구현 - reakwon : https://reakwon.tistory.com/30

#9

graph

그래프는 정점과 간선으로 이루어진 자료구조이다. 정점 간의 연결관계는 간선으로 나타낸다.

그래프의 종류

간선이 담고있는 정보와 연결 상태에 따라 그래프의 종류가 나뉜다. 두 정점을 연결하는 간선에 방향이 없다면 `무방향 그래프`, 두 정점을 연결하는 간선에 방향이 존재하면 `방향 그래프`라고 부른다. 방향 그래프는 간선의 방향으로만 이동할 수 있다. 두 정점을 이동할 때 비용이 발생하면 `가중치 그래프`로 나타낼 수 있다. 모든 정점이 간선으로 연결된 경우, `완전 그래프`라고 부른다.

그래프 구현 방식

첫 번째로 **인접행렬 방식**이 있다. 노드를 인덱스로 삼는 2차원 배열을 만들어 각 노드가 간선으로 연결되어있으면 배열에 1을 넣어주고, 연결되지 않았다면 0을 넣어주면 된다.

두 노드의 연결관계를 조회할 때, $O(1)$ 시간이 걸린다. 그러나 모든 정점에 대해, 간선 정보를 입력해야하므로 초기화에 $O(N^2)$ 시간이 소요된다.

노드의 수가 많고, 간선의 수가 적은 그래프의 경우에, 공간을 낭비하게 된다.

두 번째로 **인접리스트 방식**이 있다. 그래프의 노드들을 리스트로 표현한다. head 노드와 연결된 노드들을 링크에 달아주면 된다.

한 정점에 연결된 노드들의 정보를 얻기 위해서 $O(M)$ 시간이 걸린다.(M은 간선의 수) 간선 정보만 유지하므로, 공간 낭비가 적으나 두 정점이 연결되었는지 확인하기 위해서 $O(M)$ 시간이 걸리며, 구현이 비교적 어렵다.

그래프 용어

- `정점(vertice)` : 노드(node)라고도 하며 정점에는 데이터가 저장된다.
- `간선(edge)` : 링크(arcs)라고도 하며 노드간의 관계를 나타낸다.
- `인접 정점(adjacent vertex)` : 간선에 의해 직접 연결된 정점이다.
- `단순 경로(simple-path)` : 경로 중 반복되는 정점이 없는것, 같은 간선을 자나가지 않는 경로이다.
- `차수(degree)` : 무방향 그래프에서 하나의 정점에 인접한 정점의 수이다.
- `진출 차수(out-degree)` : 방향그래프에서 사용되는 용어로 한 노드에서 외부로 향하는 간선의 수를 뜻한다.
- `진입차수(in-degree)` : 방향그래프에서 사용되는 용어로 외부 노드에서 들어오는 간선의 수를 뜻한다.

References

- [Algorithm] 자료구조 그래프(Graph)란 무엇인가? - 코딩팩토리 : https://coding-factory.tistory.com/610

#10

tree

tree는 그래프의 일종으로, 부모 노드 밑에 여러 자식 노드가 연결되고, 자식 노드 각각에 다시 자식 노드가 연결되는 재귀적 형태의 자료구조이다. 노드들은 서로 다른 자식 노드를 가지며 이때 각 노드는 재사용 되지 않는다. 트리는 다음과 같은 특징을 갖는다.

- 반드시 하나의 루트 노드만이 존재한다.
- 모든 자식 노드는 한 개의 부모 노드만을 가진다.
- 서로 다른 임의의 두 노드에 대해 두 노드를 연결하는 경로는 유일하다.
- 사이클을 가지는 노드 집합이 존재하지 않는다.
- 노드가 N개인 트리는 항상 N-1개의 간선을 가진다.

트리 용어

- `노드(node)` : 트리를 구성하는 기본 원소
- `루트 노드(root node/root)` : 트리에서 부모가 없는 최상위 노드, 트리의 시작점
- `부모 노드(parent node)` : 루트 노드 방향으로 직접 연결된 노드
- `자식 노드(child node)` : 루트 노드 반대 방향으로 직접 연결된 노드
- `형제 노드(siblings node)` : 같은 부모 노드를 갖는 노드들
- `잎 노드(leaf node)/단말 노드(terminal node)` : 자식이 없는 노드
- `경로(path)` : 한 노드에서 다른 한 노드에 이르는 길 사이에 있는 노드들의 순서
- `길이(length)` : 출발 노드에서 도착 노드까지 거치는 노드의 개수
- `깊이(depth)` : 루트 경로의 길이
- `레벨(level)` : 루트 노드(level=1)부터 노드까지 연결된 링크 수의 합
- `높이(height)` : 가장 긴 루트 경로의 길이
- `차수(degree)` : 각 노드의 자식의 개수
- `트리의 차수(degree of tree)` : 트리의 최대 차수 = max[deg1, deg2, ..., degn]
- `크기(size)` : 노드의 개수

- `너비(width)`: 가장 많은 노드를 갖고 있는 레벨의 크기

References

- [자료구조] 트리(Tree)란 - HeeJeong Kwon : https://gmlwjd9405.github.io/2018/08/12/data-structure-tree.html
- [자료구조] 트리? + 이진 트리 (Binary Tree) - Suyeon's Blog : https://suyeon96.tistory.com/29
- 트리(그래프) - 나무위키 : https://namu.wiki/w/트리(그래프)

#11

binary tree

이진 트리(binary tree)는 각각의 노드가 최대 두 개의 자식 노드를 가지는 트리를 말한다. 즉, 모든 노드의 차수(degree)가 2 이하인 트리를 말한다. 이진 트리의 모든 서브 트리들은 모두 이진 트리이다.

순회(Traversal) 방법

- 전위 순회(preorder)
  1. 노드를 방문한다.
  2. 왼쪽 서브 트리를 전위 순회한다.
  3. 오른쪽 서브 트리를 전위 순회한다.
- 중위 순회(inorder)
  1. 왼쪽 서브 트리를 중위 순회한다.
  2. 노드를 방문한다.
  3. 오른쪽 서브 트리를 중위 순회한다.
- 후위 순회(postorder)
  1. 왼쪽 서브 트리를 후위 순회한다.
  2. 오른쪽 서브 트리를 후위 순회한다.
  3. 노드를 방문한다.
- 레벨 순서 순회(level-order)
  - 모든 노드를 낮은 레벨부터 차례대로 순회한다. 레벨 순서 순회는 너비 우선 순회(breadth-first traversal)라고도 한다.

References

- 이진 트리 - 위키백과 : https://ko.wikipedia.org/wiki/이진_트리
- 이진 탐색 트리: 이론과 소개 - 오늘도 MadPlay! : https://madplay.github.io/post/binary-search-tree
- 트리 순회 - 위키백과 : https://ko.wikipedia.org/wiki/트리_순회

#12

full binary tree

full binary tree는 단말 노드들을 제외한 모든 노드들이 2개의 자식 노드를 가지는 binary tree이다.

References

- Binary Tree 종류 - Heap 구현 사전지식 - YABOONG : https://yaboong.github.io/data-structures/2018/02/10/1_binary-tree-1/

#13

complete binary tree

완전 이진 트리(complete binary tree)는 마지막 level을 제외한 나머지 level에 노드들이 가득 차있고, 마지막 level에서 노드는 가장 왼쪽부터 채워지는 형태의 binary tree이다.

References

- Binary Tree 종류 - Heap 구현 사전지식 - YABOONG : https://yaboong.github.io/data-structures/2018/02/10/1_binary-tree-1/

#14

bst(binary search tree)

이진 탐색 트리(binary search tree)는 아래의 성질을 갖고 있는 이진 트리이다.

- 각각의 모든 노드들의 값(key)은 중복된 값이 아니다.
- 노드의 왼쪽 서브트리에는 그 노드의 값보다 작은 값들을 지닌 노드들로 이루어져 있다.
- 노드의 오른쪽 서브트리에는 그 노드의 값보다 큰 값들을 지닌 노드들로 이루어져 있다.
- 좌우 서브트리는 각각이 다시 이진 탐색 트리여야 한다.

> 탐색(Search)

검색하고자 하는 값을 루트 노드와 먼저 비교하고, 일치할 경우 루트 노드를 리턴한다.

- 불일치하고 검색하고자 하는 값이 루트 노드의 값보다 작을 경우 왼쪽 서브트리에서 재귀적으로 검색한다.
- 불일치하고 검색하고자 하는 값이 루트 노드의 값보다 큰 경우 오른쪽 서브트리에서 재귀적으로 검색한다.

삽입(Insert)

삽입을 하기 전, 탐색을 수행한다. 트리를 탐색한 후 키와 일치하는 노드가 없으면 마지막 노드에서 키와 노드의 크기를 비교해서 왼쪽이나 오른쪽에 새로운 노드를 삽입한다.

삭제(Delete)

삭제하려는 노드의 자식 수에 따라

- **자식 노드가 없는 노드(리프 노드) 삭제**: 해당 노드를 단순히 삭제한다.
- **자식 노드가 1개인 노드 삭제**: 해당 노드를 삭제하고 그 위치에 해당 노드의 자식 노드를 대입한다.
- **자식 노드가 2개인 노드 삭제**: 삭제하고자 하는 노드의 값을 해당 노드의 왼쪽 서브트리에서 가장 큰 값으로 변경하거나, 오른쪽 서브트리에서 가장 작은 값으로 변경한 뒤, 해당 노드(왼쪽 서브트리에서 가장 큰 값을 가지는 노드 또는 오른쪽 서브트리에서 가장 작은 값을 가지는 노드)를 삭제한다.

시간 복잡도

BST의 탐색, 삽입, 삭제의 복잡도는 모두 $O(h)$이다. (h는 BST의 높이) BST는 평균 시간 복잡도가 $O(\log_2 n)$이지만 최악의 경우 $O(n)$이다. (skewed tree 이면 node의 수만큼 시간이 소요됨)

트리가 complete binary tree 거나 full binary tree 이면 $O(\log_2 n)$, skewed tree 이면 $O(n)$의 시간복잡도를 갖는다.

References

- 이진 탐색 트리 - 위키백과 : https://ko.wikipedia.org/wiki/이진_탐색_트리
- Binary Search Tree - 불곰 : https://brownbears.tistory.com/392
- 6. Binary Search Tree - JuHy_ : https://ju-hy.tistory.com/90

#15

heap(binary heap)

최대값 및 최소값을 찾아내는 연산을 빠르게 하기 위해 고안된 완전 이진 트리를 기본으로 한 자료구조로서 다음의 속성을 만족한다.

> A가 B의 부모 노드이면, A의 키값과 B의 키값 사이에는 대소관계가 성립한다.

heap의 종류에는 min heap, max heap이 있다.

각 노드의 자식 노드의 최대 개수는 힙의 종류에 따라 다르지만, 대부분의 경우는 자식 노드의 개수가 최대 2개인 이진 힙(binary heap)을 사용한다.

힙에서는 가장 높은(혹은 가장 낮은) 우선순위를 가지는 노드가 항상 루트 노드에 오게 되는 특징이 있으며, 이를 응용하여 우선순위 큐와 같은 추상적 자료형을 구현할 수 있다.

> 이진 힙(binary heap)

이진 힙은 다음과 같은 두 가지 특징을 갖는다. 트리를 T, 임의 내부 노드를 v라고 하면 다음과 같다.

1. 루트 노드를 제외한 각 내부 노드는 `key(T.parent(v)) < key(v)` 또는 `key(T.parent(v)) > key(v)` 이다. (즉, 키 값은 오름차순이거나 내림차순이다.)
2. 마지막 왼쪽 결합 노드들의 레벨을 제외한 다른 모든 레벨들은 완전 이진 트리를 형성한다.

힙 리스트(heap list)로 표현할 때 i번째 노드의 왼쪽 자식 노드의 위치는 2i가 되며, i번째 노드의 오른쪽 자식 노드의 위치는 2i+1이고, 또한 i번째 노드의 부모 노드의 위치는 i/2가 된다.

이진 힙의 시간복잡도는 $O(\log n)$이다.

References

- 힙 (자료 구조) - 위키백과 : https://ko.wikipedia.org/wiki/힙_(자료_구조)

#16

min heap

최소 힙(min heap)은 부모 노드의 키 값이 자식 노드의 키 값보다 작거나 같은 완전 이진 트리이다.

key(부모노드)≤key(자식노드)*key*(부모노드)≤*key*(자식노드)

References

- [자료구조] 힙(heap)이란 - HeeJeong Kwon : https://gmlwjd9405.github.io/2018/05/10/data-structure-heap.html

---

#17

max heap

최대 힙(max heap)은 부모 노드의 키 값이 자식 노드의 키 값보다 크거나 같은 완전 이진 트리이다.

key(부모노드)≥key(자식노드)*key*(부모노드)≥*key*(자식노드)

References

- [자료구조] 힙(heap)이란 - HeeJeong Kwon : https://gmlwjd9405.github.io/2018/05/10/data-structure-heap.html

---

#18

Red-black Tree

레드-블랙트리의 정의

레드-블랙 트리(Red-Black Tree)는 이진탐색트리(Binary Search Tree)의 한 종류로, 삽입(insert), 삭제(delete), 검색(retrieval) 연산을 $O(\log N)$에 수행하도록 보장하는 **균형 잡힌 트리**를 말한다. 즉, 트리의 높이가 $\log N$이 되도록 한다.

레드-블랙 트리는 **다음의 조건**을 만족한다.

- 모든 노드는 빨간색 혹은 검은색이다.
- 루트 노드는 검은색이다.
- `NULL` 혹은 `NIL`로 표기된 리프노드는 검정색이다.
- 빨간색 노드의 자식 노드는 검정색이다. 즉, 빨간색 노드가 연속적으로 나올 수 없다.
- 리프노드에서 루트노드까지 가는 경로에서 만나는 검은색 노드의 개수는 같다.

레드-블랙트리가 균형 잡힌 트리인 이유

레드-블랙 트리의 **5번째 조건** 때문인데, 검은색 노드의 개수가 B이고 빨간색 노드가 최소가 되는 경우와 최대가 되는 경우를 생각해보자. 빨간색 노드가 최소가 되려면, 빨간색 노드 자

체가 없어야 하고 총 노드의 개수는 B개이다. 빨간색 노드가 최대가 되려면, `검정-빨강-검정-빨강-...`으로 반복되어야 한다. 이 경우 총 노드의 개수는 2B이다.

그러므로 최소 경로와 최대 경로의 차이는 2배보다 크지 않으므로 레드-블랙 트리는 균형 잡힌 트리라고 말할 수 있다.

레드-블랙 트리의 연산

레드-블랙 트리의 연산으로 `검색`, `삽입`, `제거`가 있다. 자세한 내용은 레드-블랙 트리/동작 - 위키백과를 참고!

References

- 알고리즘) Red-Black Tree - ZeddiOS : https://zeddios.tistory.com/237
- 고급 주제 - 코딩인터뷰 완전분석 : http://www.yes24.com/Product/Goods/44305533
- [레드-블랙 트리 - 위키백과] : https://ko.wikipedia.org/wiki/레드-블랙_트리#동작 :

#19

B-Tree

B-트리의 정의

B-트리는 이진 트리(Binary Tree)를 확장해 모든 리프 노드들이 같은 높이를 갖도록 하는 트리이다. 노드 내에 여러 개의 key가 있을 수 있으며, 최대 key의 개수에 따라 2개이면 2차 B-트리, N개면 N차 B-트리라고 부른다.

B-트리는 **다음의 조건**을 만족한다.

- 노드의 key의 개수가 N이면, 자식 노드의 개수는 N+1이다.
- 노드 내의 key는 오름차순으로 정렬되어 있다.
- 루트 노드는 2개 이상의 자식을 가져야 한다.
- 루트 노드를 제외한 나머지 노드들은 적어도 최대 M/2개의 key를 가져야 한다.
  - M은 B-트리의 차수를 말한다.
- 리프 노드는 모두 같은 레벨에 있어야 한다.

B-트리의 연산

B-트리의 연산은 `검색`과 `삽입`, `제거`가 있다. 다음 연산은 B-트리 연산을 이해할 수 있는 자료로 이것을 참고!

- B-트리 연산 시뮬레이션: B-Tree Algorithm Visualizations : https://www.cs.usfca.edu/~galles/visualization/BTree.html
- B-트리 연산 개념 정리: [자료구조] 그림으로 알아보는 B-Tree - emplam27.log : https://velog.io/@emplam27/자료구조-그림으로-알아보는-B-Tree

B-트리 vs B+ 트리

B+ 트리는 B-트리와 비슷하지만 리프노드가 연결리스트의 형태를 띄어 선형 검색이 가능한 트리이다. 모든 노드에 key와 data가 있는 B 트리와는 달리 B+ 트리는 리프 노드에만 data가 존재한다. 또한 `삽입`과 `제거` 연산 모두 리프 노드에서만 이루어진다.

References

- [자료구조] 그림으로 알아보는 B-Tree - emplam27.log : https://velog.io/@emplam27/자료구조-그림으로-알아보는-B-Tree
- [자료구조] 그림으로 알아보는 B+Tree - emplam27.log : https://velog.io/@emplam27/자료구조-그림으로-알아보는-B-Plus-Tree
- B트리 - 위키백과 : https://ko.wikipedia.org/wiki/B_트리
- B-Tree 개념 정리 - Jlog : https://hyungjoon6876.github.io/jlog/2018/07/20/btree.html