**2020 B+ tree implementation assignment**

컴퓨터소프트웨어학부 2019067429 한승우

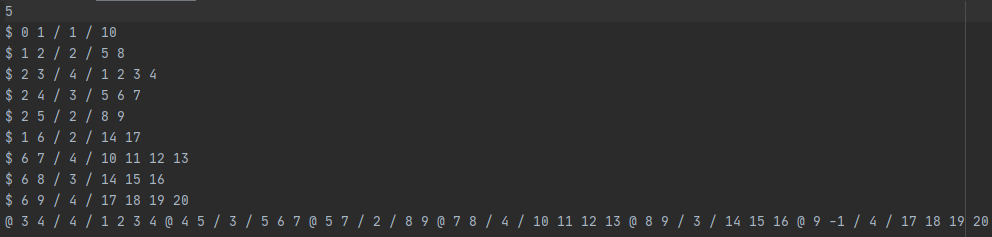
1. **전체 코드 요약**
2. **트리 저장**

* 차수를 가장 첫 줄에 저장한다.
* 깊이 우선 탐색 (DFS)를 이용해서 root부터 leaf까지 모든 노드들을 저장한다. 이때, DFS를 이용해서 만나는 노드들에게 각각 index를 부여한다.
* **트리를 저장하는 양식**

Non\_leaf\_node 일때 $ parent\_index now\_index / now\_m / now\_m의 크기 만큼의 key

Leaf\_node 일때 @ parent\_index r\_index / now\_m / now\_m의 크기 만큼의 key를 총 leaf\_node의 수만큼 반복해준다.

Ex)



1. **트리 만들기**

* 첫 줄이 차수 이므로 degree에 첫 줄을 저장해준다.
* 한 줄 씩 읽으면서 ArrayList를 사용하여 노드를 index 순서대로 저장해준다.
* 마지막 줄은 모두 leaf노드 이므로 각 leaf노드의 r을 r\_index를 이용해주어서 연결해주고 value값을 저장해준다.

1. **삽입**
2. 삽입할 leaf node를 찾아서 삽입을 실행시킨다. 이때, 찾는 과정은 log n 의 시간 복잡도를 가지는 이분 탐색을 사용하여 어느 child에 들어갈지 찾는다.
3. 삽입한 leaf node의 m값이 deg보다 작다면 그대로 둔다.
4. 삽입한 leaf node의 m값이 deg보다 크다면 leaf\_split를 실행한다.

3-1. leaf\_split의 과정은 한 노드를 두 개의 노드로 split하며 m/2 번째 키 값을

부모 노드에 올리는 것이다.

1. Split을 한 노드의 부모 노드의 m값과 deg를 비교하여 작다면 그대로 둔다.
2. Split을 한 노드의 부모 노드의 m값과 deg를 비교하여 크다면 parent\_split를 실행한다.

5-1. parent\_spit의 과정은 leaf\_spilit와 거의 동일하나 m/2 번째 키 값을 현재 노드에 남기지 않는 것이 다르다.

1. Root 까지 올라가면서 parent\_split이 일어나는 경우에 parent\_split를 실행시켜 준다.

(만약 root노드에서 parent\_split을 실행시켰을 때에는 새로 만든 부모 노드가 r oot가 된다.)

1. **삭제**
2. 삭제할 key가 존재하는 leaf node를 삽입과 마찬가지로 이분 탐색을 이용하여 찾아 내려간다.
3. 삭제한 key가 non\_leaf\_node에도 존재할 경우 삭제하고 난 leaf\_node에서 가장 작은 key로 그 값을 대체 해준다.
4. Leaf\_node의 m을 (deg-1)/2와 비교한다.
5. Leaf\_node의 m이 (deg-1)/2 이상이라면 끝낸다.
6. Leaf\_node의 m이 (deg-1)/2 이상이 아닐경우

5-1. 왼쪽 혹은 오른쪽 형제 node의 m을 확인한다.

5-2. 만약 둘 중 하나가 (deg-1)/2 이상 일 경우 key 값을 하나 가져온다.

5-3. 그렇지 않다면 leaf\_merge를 실행한다.

5-4. 왼쪽 형제 노드와 현재 노드가 merge 될 경우 왼쪽 노드에 현재 노드의 key값을 모두 넣고 현재 노드를 삭제시켜 준다.

5-5. 오른쪽 형제 노드와 현재 노드가 merge 될 경우 현재 노드에 오른쪽 노드의 key값을 모두 넣고 오른쪽 노드를 삭제시켜 준다.

5-6. 이때, 노드들 사이에 존재하는 부모노드의 key 값을 삭제시켜 준다.

Ex) 왼쪽 노드가 부모노드의 1번째 인덱스이고 현재 노드가 부모노드의 2번째 인덱스일 때, 부모노드에 1번째에 해당하는 key 값을 삭제시켜 준다.

1. 삭제한 node의 부모노드 m이 (deg-1)/2 보다 큰지 확인한다.
2. 삭제한 node의 부모노드 m이 (deg-1)/2 보다 작으면 parent\_merge를 실행한다.

7-1. 노드들 사이에 존재하는 부모노드의 key 값을 현재 노드로 내려 준다.

Ex) 왼쪽 노드가 부모노드의 1번째 인덱스이고 현재 노드가 부모노드의 2번째 인덱스일 때, 부모노드에 1번째에 해당하는 현재 노드에 내려 준다. (이점이 leaf\_merge 와의 가장 큰 차이이다.)

7-2. 왼쪽 형제 노드와 현재 노드가 merge 될 경우 왼쪽 노드에 현재 노드의 key값과 child를 모두 넣고 현재 노드를 삭제시켜 준다.

7-3. 오른쪽 형제 노드와 현재 노드가 merge 될 경우 현재 노드에 오른쪽 노드의 key값과 child를 모두 넣고 오른쪽 노드를 삭제시켜 준다.

1. Parent\_merge 후 현재 노드의 m 값이 deg보다 더 큰경우 split를 실행해준다.
2. Root로 계속해서 올라가면서 노드의 m이 (deg-1)/2 보다 큰지 확인해주면서 7,8번 과정을 반복한다.
3. 만약 merge 후 root의 m이 0이 되었을 때에는 root의 첫번째 자식 노드가 root가 된다.’
4. **Single key search**
5. Root node 에서 시작하여 insert와 마찬가지로 이분 탐색을 사용하여 알맞은 node로 내려간다.
6. 지나온 경로를 String 변수에다가 모두 저장한다.
7. 만약 leaf node에 찾으려는 key 값이 존재할 경우 저장한 경로를 모두 출력 해준 후 key값에 대응하는 value값을 출력해준다.
8. **Range key search**
9. Root node에서 시작하여 min\_key 값에 대하여 이분탐색을 사용하여 알맞은 leaf\_node 로 내려간다.
10. 찾은 leaf\_node 에서 min\_key 값보다 크고 max\_key 값보다 작은 값에 대해서 출력해준다.
11. 한 노드의 모든 key를 출력했을 경우 다음 노드로 이동하여 2번 과정을 반복해준다.
12. **코드 설명**

* **전역 변수**
* **Public Node root -> root 노드의 위치를 저장한다.**
* **Public static int deg -> 차수를 저장한다.**
* **Public static int count -> 파일 쓰기에 필요한 변수로 dfs 과정에서 필요하다.**
* **Public ArrayList<Pair> leaf -> leaf노드의 value와 r의 index를 저장해준다.**
* **Public ArrayList<Node> save\_node -> 파일을 읽어서 트리를 만들 때 필요한 변수이다.**
* **Pair class**
* key값과 value값을 변수로 가지는 클래스 이다.
* **Node class**
* 트리의 각각 한 노드가 되는 클래스 이다.
* **Int degree -> 차수**
* **Int m -> key값의 개수**
* **Node[] p\_child -> 노드의 child를 저장하는 배열**
* **Pair[] key\_value -> 노드의 key값과 value값을 저장하는 배열**
* **Node r -> 다음 노드**
* **Node parent -> 부모 노드**
* **생성자**
* **Key 값 하나를 받아오는 생성자**
* **복사 생성자**
* **기본 생성자**
* **binarySearch(Pair[] A, int n)**
* 해당 노드의 key 값 중 n과 같은 것이 있는지 확인하는 알고리즘이다. 같은 key 값을 찾는 알고리즘 중 가장 효율이 좋은 알고리즘이라서 사용했다.

**시간복잡도 log(n)**

* **Insert(int x, int value)**
* 삽입하려는 key값과 value값을 변수로 받아오는 함수이다.
* 이분 탐색을 이용하여 삽입하려는 key 값에 대응하는 leaf\_node를 찾아가는 것이 목적이며, 찾은 후에는 insertkey함수를 실행시킨다.
* 또한, 찾는 과정 중 중복 키를 발견하였을 때, 중복되었다는 메시지를 출력한 후 함수를 종료한다.
* **InsertKey(Node root, int x, int value)**
* Insert함수에서 받아온 leaf\_node에다가 직접적으로 key와 value를 넣는 함수이다.
* Key 값을 넣어 준 후 m의 값이 deg보다 클 경우 split을 실행시킨다.
* while문을 사용하여 root노드로 올라가면서 split을 실행해야 할 경우 split를 실행시켜 준다.
* **Non\_leaf\_split(Node left)**
* InsertKey 함수에서 실행 중 non\_leaf\_node에서 m이 deg보다 커질 경우 실행시켜 주는 함수이다.
* 부모가 존재할 때의 경우와 부모가 존재하지 않을 때, 두가지 경우로 나누어서 구현을 해주었다. (부모가 없을 경우는 root노드 일 경우이다.)
* 부모노드에 m/2번째 키 값을 올려주고 노드를 두개로 나누어 준다. 이때, 나누어준 두 노드는 값을 올려준 부모노드의 자식들이 된다.
* **Leaf\_split(Node left, int x, int value)**
* InsertKey 함수 실행 중 leaf\_node에서 m이 deg보다 커질 경우 실행시켜 주는 함수이다.
* Non\_leaf\_split함수와 구현 방식은 같으나 **값을 올리는 것이 아닌 복사를 하는 것이 차이점이다.**
* **Node Goto\_root(Node p)**
* 현재 노드의 위치를 root로 옮겨 주는 함수이다.
* **Node Find\_leaft\_node(Node s\_temp, int p\_to\_child)**
* Delete의 merge과정 중에 필요한 함수이다.
* 현재 노드의 왼쪽 형제 노드를 찾아 오는 함수이다.
* **Node Find\_right\_node(Node s\_temp, int p\_to\_child)**
* Delete의 merge과정 중 필요한 함수이다.
* 현재 노드의 오른쪽 형제 노드를 찾아 오는 함수이다.
* **Void delete(int x)**
* Insert 함수와 마찬거지로 while문과 이분 탐색을 이용하여 delete가 이루어 져야할 leaf\_node를 찾아 내려간다
* 만약 delete하려는 key 값이 트리에 존재하지 않을 경우 “NOT FOUND” 메시지를 출력한다.
* Key 값을 삭제 해준 후 해당 노드의 m이 (deg -1)/2를 넘는지 확인한다.
* 넘을 경우 Goto\_root함수를 실행시켜 다시 root노드로 돌아간 후 함수를 종료한다.
* 넘지 않을 경우 Find\_left\_node 함수와 Find\_right\_node 함수를 사용하여 양 옆 형제 노드를 찾아온다.
* 왼쪽과 오른쪽 형제 노드의 m-1 이 (deg-1)/2를 넘는지 확인한다.
* 넘을 경우 왼쪽 또는 오른쪽 형제 노드에서 값을 하나 가져 온 후 함수를 종료한다.
* 이 때, 삭제한 key와 값은 key값이 non\_leaf\_node에 존재 할 경우 현재 노드의 가장 작은 값으로 대체하여 준다.
* 왼쪽 형제와 오른쪽 형제 노드 모두 m-1값이 (deg-1)/2를 넘지 않는다면 leaf\_merge를 실행시켜 준다.
* **Leaf\_merge**

1. 오른쪽 형제 노드가 존재 할 경우 현재 노드에 오른쪽 형제 노드의 모든 key 값을 넣어준다.
2. 이 때, 부모노드에서 현재 노드와 오른쪽 형제 노드 사이에 존재하는 key 값을 삭제시켜 준다.
3. 만약 non\_leaf\_node 중에서 삭제한 key값과 같은 key값이 존재 할 경우 현재 노드의 가장 작은 값으로 대체 하여 준다.
4. 왼쪽 형제 노드가 존재 할 경우에도 위와 같은 과정을 하여준다.

* Leaf\_merge를 실행시켰을 때, 다시 root 노드로 올라가는 과정 중 노드의 m 값이 (deg-1)/2보다 작다면 parent\_merge를 실행시켜 준다.
* **Node parent\_merge(Node temp)**
* Non\_leaf\_node 이면서 merge를 해주어야 할 경우 실행시키는 함수이다.
* Find\_left\_node 함수와 Find\_right\_node 함수를 이용하여 현재 노드의 왼쪽 형제와 오른쪽 형제를 찾아온다.
* 왼쪽 형제가 존재 할 경우, 부모노드에서 현재 노드와 왼쪽 형제 노드 사이에 존재하는 key 값을 현재 노드에 내려준다.
* 이때, 왼쪽 형제 노드와 현재 노드의 m값 합이 deg보다 클 경우와 작은 경우로 나누어 주었다.
* 왼쪽 형제 노드와 현재 노드의 값이 deg 보다 클 경우 왼쪽 형제 노드의 가장 왼쪽 key 값을 부모 노드에 올려주고, 왼쪽 형제 노드의 가장 오른쪽 자식을 현재 노드의 가장 왼쪽에 할당시켜 준다.
* 왼쪽 형제 노드와 현재 노드의 값이 deg 보다 작을 경우 merge 과정을 실행시킨다.
* Merge를 할 때, 현재 노드의 모든 key값을 왼쪽 형제 노드에 넣어 준 후 현재 노드를 삭제 시킨다.
* 만약 왼쪽 형제가 존재하지 않을 경우 부모노드에서 오른쪽 형제 노드와 현재 노드 사이에 존재하는 key값을 현재 노드에 내려준다.
* 왼쪽 형제 노드가 존재하는 경우와 마찬가지로 오른쪽 노드와 현재 노드의 m값이 deg보다 클 경우와 작은 경우로 나누어 주었다.
* 두 노드의 m값 합이 deg보다 클 경우 현재 노드에서 가장 오른쪽 key 값을 부모 노드에 올려주고, 현재 노드의 가장 오른쪽 자식을 오른쪽 형재 노드의 가장 왼쪽에 할당시켜 준다.
* 두 노드의 m 값 합이 deg보다 작을 경우 왼쪽 형제 노드의 경우와 마찬가지로 merge를 실행시켜 준다.
* **Void single\_key\_search(int f\_key)**
* Single\_key\_search를 실행시켜 주는 함수이다.
* while문과 이분 탐색을 사용하여 값을 찾으려는 key가 존재하는 leaf\_node로 내려간다.
* 내려가는 과정 중에 만나는 모든 노드를 String 변수인 a에 저장한다.
* 만약 내려온 leaf\_node에 찾으려는 key값이 존재할 경우 변수 a를 출력해 준 후, key값에 상응하는 value값을 출력해 준다.
* key값이 존재하지 않을 경우 “NOT FOUND” 메시지를 출력한 후 종료한다.
* **Void range\_key\_search(int start\_key, int end\_key)**
* while문과 이분 탐색을 사용하여 start\_key보다 크거나 같은 가장 작은 key값을 가진 leaf\_node를 찾아 내려간다.
* 찾은 위치부터 end\_key값보다 커질 때까지 leaf\_node의 key값과 value값을 출력해준다.
* 이때, while문을 사용하여 한 노드의 모든 key값을 출력 해주었으면 다음 노드로 이동한다.
* 만약 아무런 출력도 하지 않았다면 “NOT FOUND”를 출력한 후 종료한다.
* **Static void create(String index\_file)**
* Index\_file의 이름을 가진 파일을 만들고 차수를 저장해준다.
* **void save(FileWriter w, Node now, int parent)**
* dfs를 응용하여 파일에 트리를 저장해주는 함수이다.
* 모든 노드를 탐색하면서, 각 노드의 index값과 부모 노드의 index값, 그리고 현재 노드의 m값, key값들을 파일에 입력하여 준다.
* 만약 해당 노드가 leaf\_node일 경우 전역변수 leaf에 해당 노드의 index값과 r값을 저장해 준다.
* **void save\_layout(String index\_file)**
* save 함수를 실행시켜 주는 함수이다.
* Save 함수를 실행 시킨 후 파일의 마지막 줄에 leaf\_node들의 index와 다음 노드의 index, value들을 입력하여 준다.
* **Void make\_tree(String index\_file)**
* Index\_file을 한줄 씩 읽는다.
* “/”를 기준으로 split해주어 각 노드의 parent\_index, index, m을 저장해 준다.
* 마지막 줄에서는 모두 leaf\_node 이므로, r과 value를 해당하는 노드에 저장해 준다.
* **Void read\_data(String filename, String act)**
* 한 줄씩 읽으면서 Insert와 delete 과정에서 넣어주거나 삭제시켜야 할 key 값이 들어있는 파일을 인자로 받아 온후 insert와 delete함수를 실행시켜 주는 함수이다.

1. **컴파일 하는 방법**

terminal이나 cmd등 에서 파일이 있는 경로까지 이동 후 해당 명령어를 실행한다. 

1. **후기**

선배들한테 익히 비플트리의 악명을 들어와서 조금 겁에 질린 상태로 과제를 시작하였습니다. 예상대로 쉽지 않았고, 특히 평소 c나 c++로만 알고리즘 코딩을 해오다가 자바로 하니 조금 어색하였습니다. 제 코드를 보시면 느끼겠지만, java를 잘 알지 못하여서 java의 이점을 살릴 수 있는 Arrayslist나 여러가지 좋은 함수들을 사용하지 않았습니다. 사실은 이미 전체적인 틀과 어느정도의 코딩을 마친 후에 해당 사실들을 알게 되어서 고치기에는 늦었습니다… 그 결과 자바보다는 c++에 더 가까운 코드가 완성 되었고, 덕분에 조금 더 머리를 쓰면서 비플트리를 작성 한 것 같습니다.

하지만 약 2주간의 시간동안 많은 시간을 할당하였지만 delete를 완벽하게 구현하는 데 실패한 것이 아쉽습니다. 수많은 디버깅 후에 문제가 merge과정 중 child 노드와 부모 노드를 연결 하는 과정 중 연결이 안된 경우가 있다는 것을 알았지만, 수많이 고쳐 보아도 많은 숫자를 넣고 어떠한 경우에 마주하였을 때, delete가 제대로 되지 않는 경우를 계속하여 마주하였습니다. 약 1200줄에 달하는 코드를 한줄 한줄 곱씩으면서 디버깅하고, 코딩하는 과정 중 제 자신의 실력이 많이 성장하는 것을 느꼈고, delete 경우 중 한가지 경우 떄문에 delete가 계속 실패하는 모습을 보며 오기가 생겨서 더 많이 시간을 비플트리 구현에 할당한 것 같습니다. 비록 코드를 100% 구현시켜 제출하지는 못하였지만, 제출 이후에도 delete를 완전히 완성시켜 볼 생각입니다.

긴 글 읽어주셔서 감사합니다.