알고리즘 2분반

[Btree 구현]

학번 : 32203660

성명 : 이현정

1. b-tree 알고리즘 구현

해당 내용은 아래의 github 주소에 올려 두었습니다.

GitHub 링크 : https://github.com/12hyeon/Hyeon-Algorithm/blob/main/lec_algorithm/btree.c

2. 설계

교수님께서 수업 시간에 C언어도 가능하다고 하셔서 C언어를 이용하였습니다.

btree는 하위 트리에 대한 키의 범위를 가집니다. 모든 단말노드는 동일한 높이를 가지고, 내부의 키의 개수는 2보다 큰 t에 대해 (t-1) ~ (2t-1)까지 값 중 1개여야하며, 루트의 자식은 2개 이상이라는 기본 조건을 가집니다. 검색은 중위순회로 root에서 시작해서 왼쪽 -> 오른쪽 서브 트리의 방문으로 진행합니다.

우선 minimum degree를 5,7,9인 경우에 대해 수행해야하므로 해당하는 내용은 t이라는 변수를 이용해서 해당 내용이 변경시키면서 다른 파일로 구성하였습니다.

0~9999 사이 서로 다른 1000개 데이터를 삽입하는 과정은 rand 함수를 생성해서 해당 내용이 수행될 수 있게 하였습니다. 삭제하는 내용 또한 랜덤으로 돌려서 해당 노드가 존재하는 경우와 존재하지 않는 경우가 출력이 되면서 500번 발생될 수 있게 하였습니다.

그리고 생성과 제거까지의 과정의 시간을 재서 마지막에 출력되게 하였습니다.

구조체 node : 최대가질 수 있는 자식의 개수만큼 key 배열과 자식노드를 가리킬 구조체 포인터, leaf인지와 해당 노드에 담긴 key의 개수를 확인할 값을 가지고 있습니다.

1) key 삽입

루트에 자리가 남아있으면 최대 개수까지 계속 넣다가 루트가 가득차면 루트의 중간 키가 새 루트가 되고, 나머지 키들은 서브 트리가 되는 과정을 거칩니다. 해당 과정에서 중간 키가 들어갈 자리가 가득 차 있다면 다시 가운데 키를 위로 이동시키고 나머지를 분할시킨 후에 노드에 값을 추가하는 과정을 하면서 해당 값이 들어갈 수 있을 때까지 위로 올라가면서 반복합니다.

랜덤으로 해당 값이 트리에 속해 있지 않으면, 키 값에 대한 삽입을 시작합니다. btree_Insert 함수에서 root 위치에 키가 가득 차있으면 새로 노드를 만들어서 root로 btree_Split_Child 함수에 해당 root 노드 위에 빈 root가 생성된 트리와 해당 노드의 개수인 0을 넘겨주고, btree_Insert_Nonfull 함수에는 해당 트리와 넣을 key값을 넘겨줍니다.

```
void btree_Split_Child(node* x, int i)
{
    node* z = malloc(sizeof(node));
    if (z == NULL)
    {
        printf("memory allocation falied");
        return;
    }
    node* y = x->child[i]; // 0 <= i
    z->leaf = y->leaf;
    z->n = t - 1;
    for (int j = 0; j < t - 1; j++)
    {
        z->key[j] = y->key[j + t];
    }
    if (y->leaf == FALSE)
    {
        for (int j = 0; j < t; j++)
        {
            z->child[j] = y->child[j + t];
        }
    }
    y->n = t - 1;
    for (int j = x->n; j > i; j--)
    {
        x->child[j + 1] = x->child[j];
    }
    x->child[i + 1] = z;
    for (int j = x->n - 1; j > i - 1; j--)
    {
        x->key[j + 1] = x->key[j];
    }
    x->key[i] = y->key[t - 1];
    x->n = x->n + 1;
}
```

btree_Split_Child 함수에서는 받은 노드x의 key가 들어가야할 자리의 key인 y가 leaf인지 확인한 내용을 새 노드z의 leaf에 넣고, 노드에 들어가야하는 t-1을 새 노드의 n에 넣습니다. 그리고 z의 0~t-2까지 인덱스에 y의 t~2t-2까지의 값들을 넣고, y가 leaf인 경우에는 z의 key의 인덱스 0~t-1까지에 t~2t-1까지의 y의 key 값을 넣습니다. 이것으로 가져온 x에서 들어갈 자리의 우측 부분에 대해서 z에 넣어지게 처리하였고, 이 과정을 좌측에 대해서도 수행을 합니다.

```
void btree_Insert_Nonfull(node* x, int k)
{
    int i = x->n;
    if (x->leaf)
    {
        i--;
        while (i >= 0 && k < x->key[i])
        {
            x->key[i + 1] = x->key[i];
            i--;
        }
        x->key[i + 1] = k;
        x->n = x->n + 1;
    }
    else {
        while (i >= 1 && k < x->key[i - 1])
        {
            i--;
        }
        if ((x->child[i])->n == 2 * t - 1)
        {
            btree_Split_Child(x, i);
            if (k > x->key[i]) {
                i++;
            }
        }
        btree_Insert_Nonfull(x->child[i], k);
    }
}
```

btree_Insert_Nonfull 함수에서는 해당 트리가 leaf인지 확인 후에 leaft이면 총 담겨있는 key의 수를 줄여가면서 값의 위치를 1개씩 뒤로 미루다가 key가 담겨질 위치를 찾으면 해당 위치에 key를 넣어서 알맞게 자신의 자리를 찾게 합니다. leaf가 아닌 경우에는 해당 위치를 찾은 후에 해당 자리의 개수가 2t-1로 찼으면 다시 btree_Split_Child 함수를 해당 위치에 대해 수행하고 재귀로 자신의 넣어진 노드에 대해서도 key 값과 함께 처리가 진행됩니다.

2) key 삭제

// 해당 함수가 너무 길어서 github 페이지의 코드로 확인

btree_Delete 함수에서 동작을 하게 되며, 키가 존재하고 리프에 키(t개)가 존재하면. 해당 키가 들어갈 위치를 해당 노드의 key값들에 대한 인덱스를 줄이면서 탐색을 해서 key가 존재하는지 여부에 따라서 해당 위치에 존재하며 키를 빼도 최소 개수(t-1개)를 만족하면 해당 키를 삭제합니다.

키가 존재하지 않지만 해당 값이 있다면 위치할 곳의 자식이 t-1개를 가지면, 계속 아래로 내려가면서 t개를 가지는 것을 찾아서 키를 포함하는 노드로 내려가는 과정을 진행합니다. 해당 키가 있을 자리의 자식이 t-1개인데, 형제가 t개 이상을 가지면 형제에서 남는 키를 해당 자식으로 옮기는 과정을 거칩니다. 해당 키가 있을 자리의 인접한 형제가 모두 최소인 t-1개만 가진다면, 해당 키 + 형제 + 부모를 결합해서 1개로 만드는 과정을 거칩니다.

다음은 키가 존재하고 내부 노드면, 해당 키의 자식이 t개 이상 키를 가지고 있는 경우의 처리입니다. 노드의 가장 오른쪽 값이 키가 되는 경우에는 키의 바로 이전 값 x'을 찾아서 해당 노드를 키의 자리에 두는데, x'가 오른쪽 자식가 있으면 해당 끌어올리는 과정을 반복합니다. 노드의 가장 왼쪽 값이 키가 되는 경우에는 키의 바로 이후 값을 찾아서 위와 같이 끌어올리는 과정을 반복합니다.위 2가지가 아닌 경우(키가 노드의 끝에 존재하는 값이 아닌 경우)에는아래쪽 값을 가져오는데, 아래쪽의 왼쪽 오른쪽 서브 트리가 모두 최소 개수를 가졌으면 아래의 2개 자식를 합치고 중간 값 1개를 올리는 과정을 거칩니다.

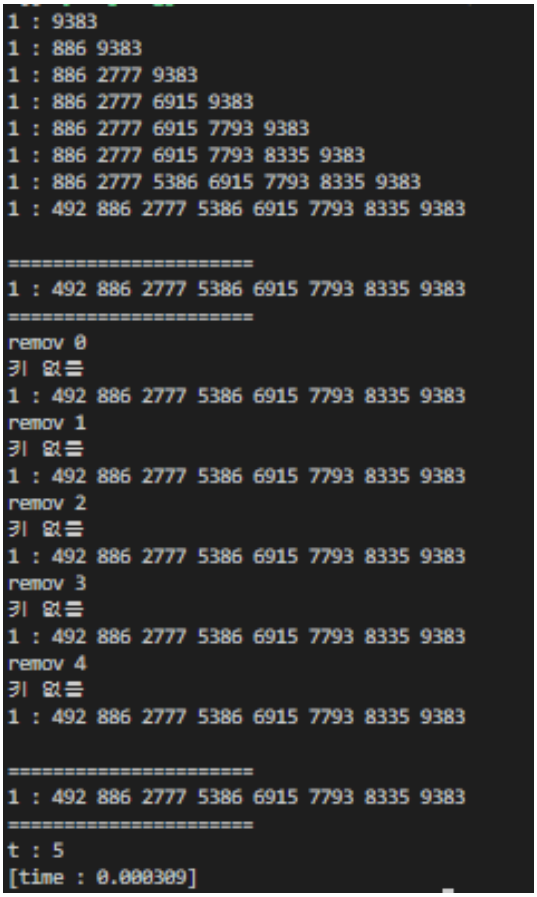
4. 결과

주어진 조건대로 실행을 시키면 이미 존재하는 값인지에 대해 search하는 시간, 노드 삽입과 제거의 과정에서 시간이 오래 걸려서 간단하게 삽입은 8개, 제거는 5개로 값이 랜덤하게 들어간 경우에 제대로 돌아가는지 확인한 내용입니다.

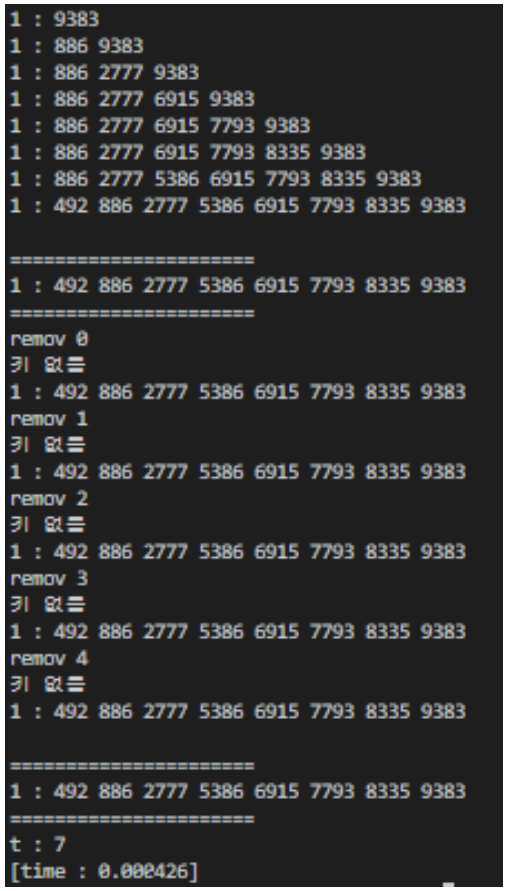
현재는 들어가는 노드가 모두 root의 최대 값을 넘어가지 않아서 비슷한 시간이 출력되지만, 더 t가 커질수록 한번에 많은 키값이 노드에 들어가서 아래로 내려가 보는 작업이 더 필요해서 t가 많을수록 더 효율적으로 동작하게 될 것 같습니다.

-> 다음 페이지에 해당 결과를 출력하였습니다.

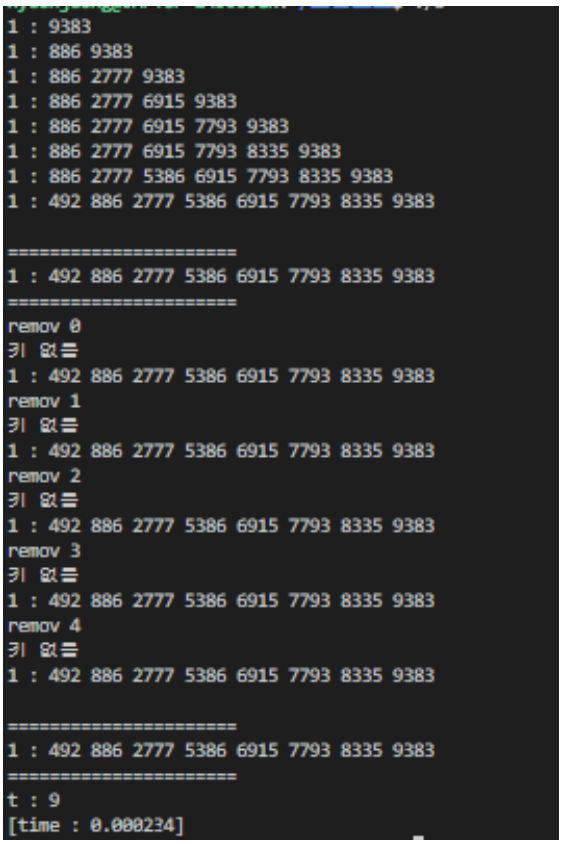
t = 5

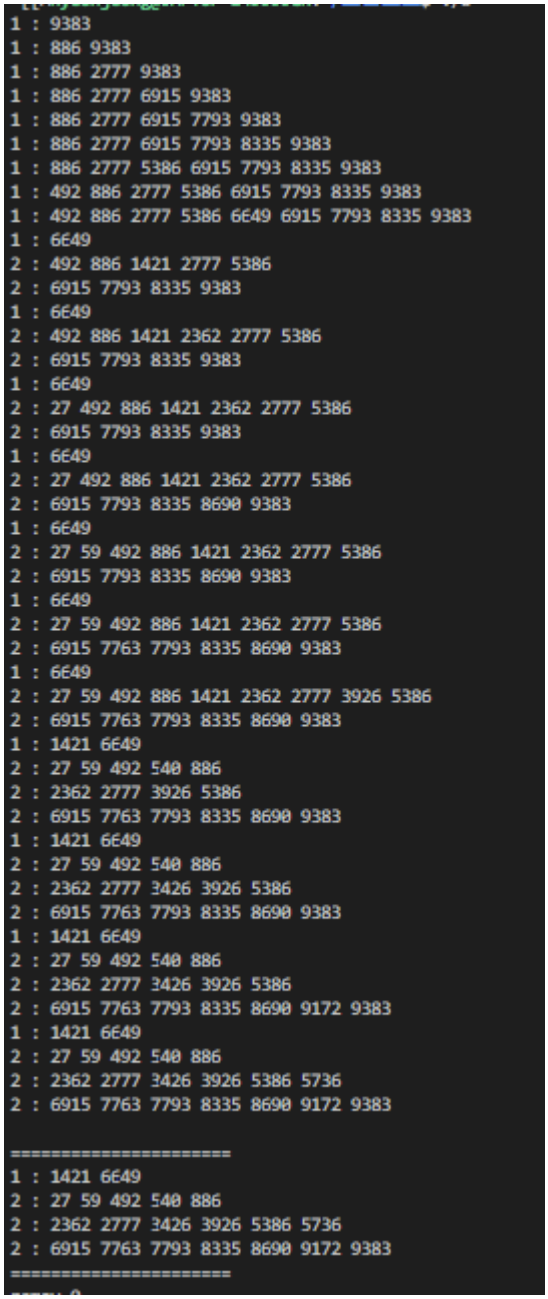

1 : 9383
1 : 886 9383
1 : 886 2777 9383
1 : 886 2777 6915 9383
1 : 886 2777 6915 7793 9383
1 : 886 2777 6915 7793 8335 9383
1 : 886 2777 5386 6915 7793 8335 9383
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
remov 0
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 1
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 2
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 3
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 4
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
t : 5
[time : 0.000309]

t = 7


1 : 9383
1 : 886 9383
1 : 886 2777 9383
1 : 886 2777 6915 9383
1 : 886 2777 6915 7793 9383
1 : 886 2777 6915 7793 8335 9383
1 : 886 2777 5386 6915 7793 8335 9383
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
remov 0
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 1
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 2
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 3
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 4
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
t : 7
[time : 0.000426]

t = 9


1 : 9383
1 : 886 9383
1 : 886 2777 9383
1 : 886 2777 6915 9383
1 : 886 2777 6915 7793 9383
1 : 886 2777 6915 7793 8335 9383
1 : 886 2777 5386 6915 7793 8335 9383
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
remov 0
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 1
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 2
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 3
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
remov 4
키 없음
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
1 : 492 886 2777 5386 6915 7793 8335 9383
==========
t : 9
[time : 0.000234]



-> 해당 내용은 20개을 삽입하는 경우 만들어지는 트리 모양을 보여줍니다.