ads-final 구현 및 실험

1. 서론

거리 비교의 계산양을 줄이기 위해 1 차원 range_tree 를 C 로 구현하여 과제를 해결하였다.

2. 명세

이번 과제 데이터는 점이 하나씩 입력되거나 삭제된다. 이에 대응하기 위해 range tree 를 적용하였다. 또한 힙 할당을 줄이고 캐시 히트를 높이기 위해 range tree 를 배열을 이용하여 구현하였다.

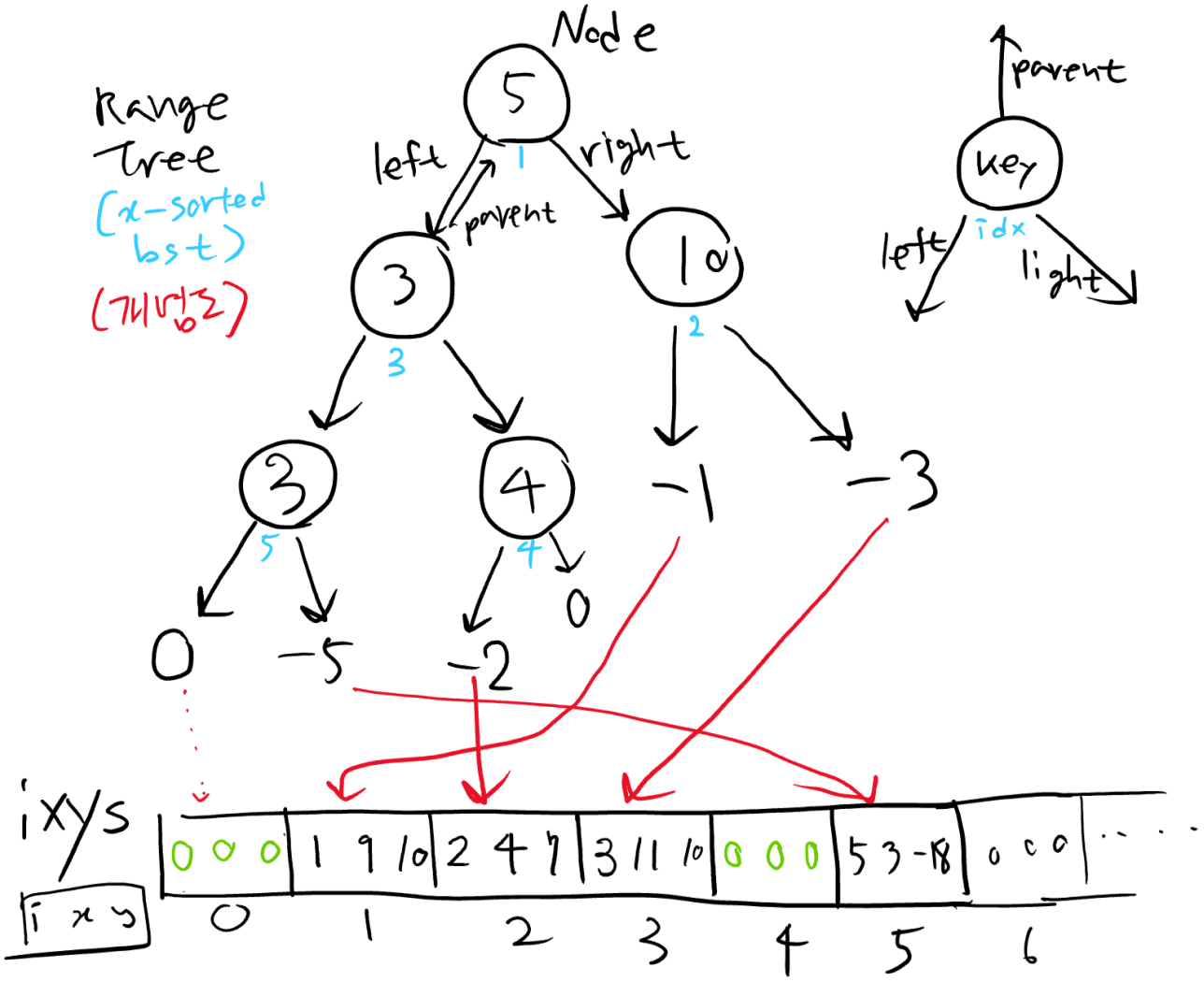

그림 1. Range Tree 개념도

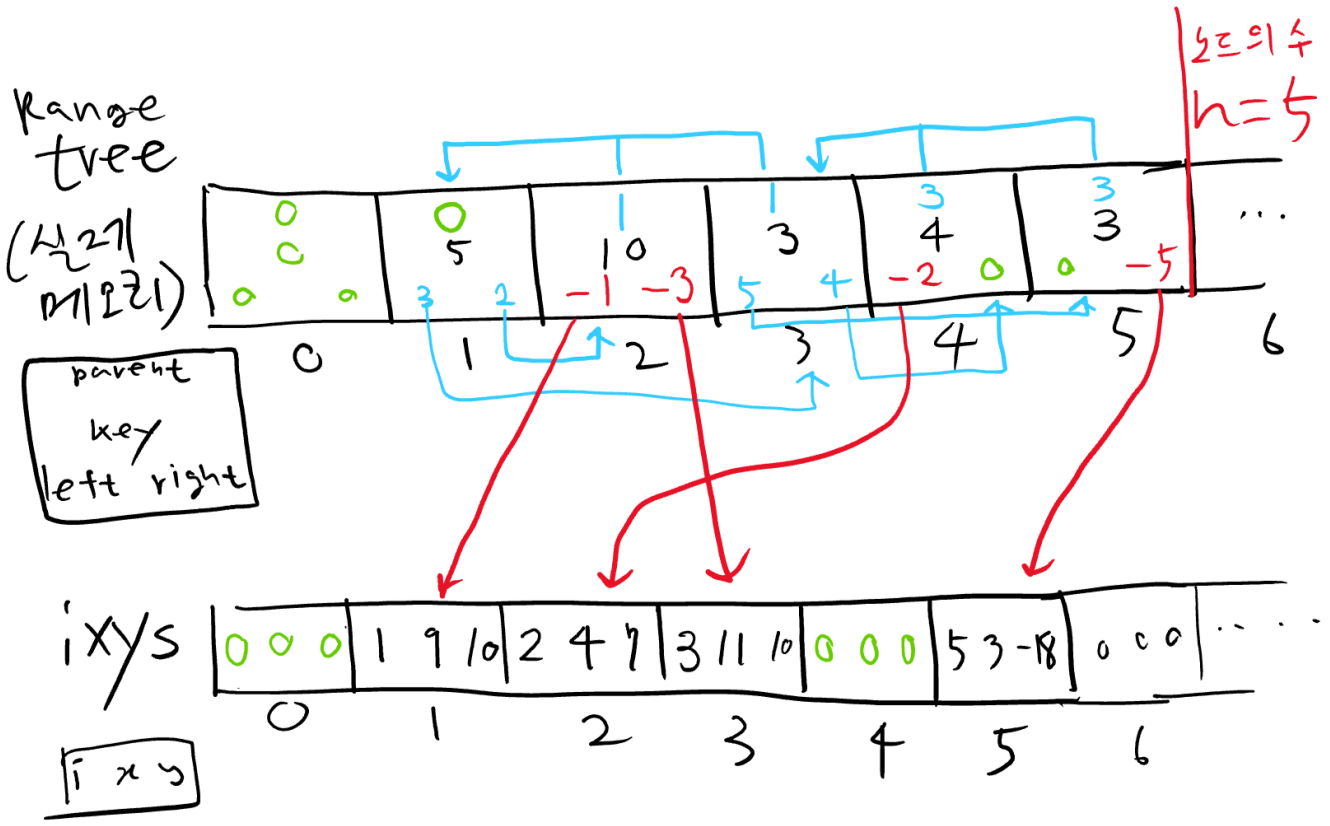

그림 2. 실제 메모리에서의 range tree

삽입되는 좌표는 3 개의 int: i,x,y 로 이루어진 구조체 IXY 로 정의된다. 데이터가 입력될 때마다 IXY 의 배열 ixys 에 저장되며 데이터에 명시된 위치에 인덱스값(i)과 좌표값(x,y)을 저장한다. 배열 ixys 는 항상 첫번째 위치(i = 0)를 비워둔다. i = 0 인 경우, 존재하지 않는 좌표로 취급한다.

range tree 는 4 개의 int: key, parent, left, right 로 이루어지는 구조체 Node 의 배열로 정의하였다. range tree 는 각 key 의 값으로 정렬된 binary search tree 이며, parent, left, right 는 각각 부모, 왼쪽 서브트리, 오른쪽 서브트리를 가리킨다.

성능을 높이고 메모리 사용량을 줄이기 위해 range tree 는 배열을 이용해 구현하였으며, parent, left, right 는 노드를 가리키는 포인터가 아닌 range tree 배열의 인덱스이다. range tree 를 표현하는 배열의 첫번째 위치는 항상 비워둔다. parent, left, right 의 값이 0 일 경우 가리키는 노드가 없는 것을 의미한다. 양수이면 range tree 의 인덱스를 가리키는 것이며, 음수일 경우 ixys 의 인덱스, 즉 실제 좌표를 가리킨다. range tree 는 트리에 존재하는 노드 수 만큼의 연속된 메모리를 사용하며, 노드의 수는 사용자가 따로 기억해 두어야 한다.

이렇게 정의하였기 때문에, 사용자는 연속된 메모리를 할당하여 트리를 구성할 수 있게 된다. 이에 따라 더 나은 캐시 히트 확률을 기대할 수 있으며, 여러번의 힙 할당에 따른 성능

저하를 없앨 수 있다. 또한 포인터가 아닌 int 로 트리를 구성했기 때문에, 64 비트 환경에서는 메모리 사용량을 줄일 수 있다.

3. 연산

range tree 는 insert, delete, range_query1d, all_leaves 를 지원한다.

```
int insert(int n_node, Node tree[], char xORy, int iidx, IXY ixys[]);
```

n_node: 현재 트리에 존재하는 노드의 수

tree: 트리

xORy: 'x' 이면 x 로 정렬된 트리, 'y'이면 y 로 정렬된 트리.

iidx: 삽입하려는 좌표값의 ixys 상에서의 인덱스

ixys: ixy 가 저장된 배열

반환값: 변경된 n_node

트리에 새로운 좌표를 삽입한다. 사용자는 트리에 존재하는 노드의 수와 정렬된 기준을 미리 알고 있어야 한다. 이번 과제에서는 데이터가 무작위로 입력되기 때문에, bst 의 균형을 신경쓰지 않아도 트리의 높이는 좌표의 수가 N 일 때 근사적으로 O(logN)이 되며 이는 insert 의 시간복잡도이다.

먼저 ixy 의 인덱스 iidx 가 삽입되어야 하는 노드를 트리에서 찾아낸 뒤(O(log N)) iidx 를 적절한 위치(left or right)에 삽입한다. 만일 노드가 꽉 차 있다면(left 와 right 가 0 이 아니라면) 새로운 노드를 할당하여(n_node++) 연결하고 iidx 를 삽입한다. 마지막으로 n_node 를 반환한다.

```
int range_query1d(const Node tree[], const IXY ixys[], char xORy, int min, int max,
                  int ixy_idxes[], int stack[]);
```

tree: 트리

ixys: ixy 가 저장된 배열

xORy: 'x' 이면 x 로 정렬된 트리, 'y'이면 y 로 정렬된 트리

min, max: 범위, [min, max]를 결과로 반환한다.

ixy_idxes: [min, max]에 속하는 ixy 의 인덱스를 xORy 의 기준에 정렬하여 반환한다.

stack: inorder traverse 에 필요한 계산 공간.

반환값: 가져오는 좌표의 수, ixy_idxes 에서 좌표를 다룰 때 쓸 수 있다.

트리에서 원하는 범위에 속하는 모든 좌표값을 가져온다. 좌표의 수가 N 이고, 범위에 속하는 좌표가 k 일 때, 시간복잡도는 O(logN + k)이다. 교과서에 나온 알고리즘과 같이 먼저 split node 를 구하고, split node 에서부터 left path 와 right path 의 subtree 에 존재하는 좌표를 가져온다. subtree 에 존재하는 모든 좌표를 가져와야 하기 때문에, inorder traverse 가 수행된다. 또한 left path 에서 subtree 에서 좌표를 가져올 때, 좌표의 순서가 bst 의 기준가 반대가 되기 때문에, 내부적으로 가변 배열을 만들어 노드의 인덱스를 저장한 뒤, right path 의 인덱스를 모드 가져온 후 left path 에서 가져온 인덱스들의 순서를 뒤집고, 모든 노드에 대해서 inorder traverse 를 수행하여 ixy_idxes 에 저장하였다.

```
int delete(int n_node, Node tree[], int iidx, IXY ixys[],
           int ixy_idxes[], int prevs[], int stack[]);
```

n_node: 현재 트리에 존재하는 노드의 수

tree: 트리

iidx: 삭제하려는 좌표값의 ixys 상에서의 인덱스

ixys: ixy 가 저장된 배열

ixys_idxes, prevs, stack: 계산을 위한 임시 공간.

반환값: 변경된 n_node

트리에서 특정 좌표를 삭제한다. 좌표의 수가 N 일 때 시간복잡도는 O(logN + k)이다. 삭제해야하는 좌표가 존재하는 노드를 트리를 내려가면서 찾는다. 만일 원하는 좌표에 속하는 노드가 여러개 있는 경우, 이들을 모두 가져와서 확인한 후 iidx 와 동일한 자식을 가지는 노드를 삭제한다. 만일 노드가 텅 비게 되는 경우(left, right 모두 0) 노드를 삭제한다. range tree 는 연속된 메모리 상에 존재해야 하기 때문에 메모리의 마지막(인덱스는 n_node)이 아닌 곳에 존재하는 노드가 삭제될 경우, 마지막 노드를 삭제된 노드가 존재하는 위치에 덮어씌워야 한다. 이는 연속적으로 일어날 수 있으며, 이에 따라 트리의 크기는 1 이상 줄어들 수 있다.

3. 결과

range tree 를 이용하여 주어지는 좌표를 x 의 값으로 정렬해 두고, 계속해서 트리를 유지한다. 쿼리(cx,cy,r) 가 입력으로 들어올 경우 range tree 에서 cx – r, cx + r 범위로 range_query1d 를 수행하여 포함되는 ixy 를 찾아낸 뒤, cy – r, cy + r 범위에 존재하는 ixy 만 걸러낸다. 걸러진 좌표들에 대해 (cx,cy)까지의 거리를 체크하여 원 안에 포함된 점의 수와

최대 거리에 존재하는 좌표의 인덱스를 알아내었다. 그 결과 Intel **i7-8700K** (3.70 GHz) CPU 환경에서 **pin_1.txt** 를 **5.076 초**, **pin_2.txt** 를 **6.683 초** 만에 처리하였다.

4. 결론 및 소감

이번 과제에서는 range tree 를 배열을 이용하여 구현해 보았다. 높은 성능을 기대한다.

과제에 맞는 range tree 를 구현하는 의사 코드나 구현체를 찾을 수 없었기에 모든 알고리즘을 직접 생각해내고 적절히 변형해야 했다. 특히 온라인으로 좌표가 입력되며, 좌표에 중복이 있을 수 있고, 좌표가 온라인으로 삭제되는 조건을 만족하기 위해 고민을 많이 했다. 또한 성능을 최대한으로 짜내기 위해 힙 할당을 배제하고 배열과 인덱스만으로 트리를 구성해 보았다. 이론적으로는 가능한 것을 알고 있었지만 구현이 생각보다 까다로워 여러모로 고생을 했다. 특히 range search 에서 inorder traverse 로 가져온 좌표의 순서를 뒤집는 작업과, delete 에서 연속된 메모리를 보장하기 위해 배열의 마지막에 저장된 노드를 삭제된 노드의 위치에 덮어쓰는 작업을 구현하는 것이 굉장히 까다로웠다.

insert, delete, range_query 의 알고리즘을 직접 짜는 일은 매우 힘들었지만 한편으로는 정말 재미있는 경험이었다. 또한 배열을 이용하여 트리를 만드는 것을 처음 해보았는데, C 를 제대로 활용하는 방법이라고 생각한다. 마지막으로, range tree 를 이차원으로 만들어보고 성능을 비교해보고 싶었지만 시간이 촉박하여 하지 못한 점은 아쉽다.

한 학기동안 수고하셨습니다.