B+tree

2019019016 서시언

<알고리즘 요약>

1. node의 구조

1) Pair class는 key, value 와 해당 key의 왼쪽 child 노드인 leftChild를 가진다.
2) Node class는 Pair의 목록인 p, Pair의 개수(key의 개수)인 m, 해당 노드의 가장 오른쪽 child(non leaf 노드일 때) 또는 오른쪽 노드(leaf 노드일 때)인 r을 가진다.

2. b+tree 저장

1) 해당 노드의 key 개수
   해당 노드가 leaf인지 여부(leaf이면 1, non leaf이면 0)
   key, value
   key, value
   :
   의 형태로 저장된다.
2) 노드는 root 노드부터 시작해서 recursive하게 저장된다.
   ① 현재노드 -> 가장 왼쪽 child노드 -> … ->가장 오른쪽 child 노드

3. b+tree 로드

1) root 노드부터 시작해서 recursive하게 로드한다.
   ① 현재노드 -> 가장 왼쪽 child노드 -> … ->가장 오른쪽 child 노드
   (단, 현재 노드가 leaf인 경우는 해당 노드의 모든 child를 null로 설정하고 return한다.)

4. key 삽입

1) root 노드부터 시작해서 recursive하게 이동한다.
2) 해당 노드의 child 노드가 leaf 노드라면,
   ① 해당 노드에서 삽입하고자 하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드를 Node 변수 'child'에 저장한다. (단, 해당 노드의 가장 오른쪽 key보다 삽입하고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드를 'child'에 저장한다.)
   ② 'child'에서 key가 삽입되어야 할 위치를 찾아 삽입한다.
3) 해당 노드의 child 노드가 leaf 노드가 아니라면,
   ① 해당 노드에서 삽입하고자 하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드로 이동하고 Node 변수 'child'에 저장한다. (단, 해당 노드의 가장 오른쪽 key보다 삽입하고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드로 이동하고 'child'에 저장한다.)
   ② ①과정 이후 'child'의 pair 개수가 degree이상이면 split을 해준다.

5. key 삭제

1) root 노드부터 시작해서 recursive하게 이동한다.
2) 해당 노드의 child 노드가 leaf 노드라면,

① 해당 노드에서 삽입하고자 하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드를 Node 변수 'child'에 저장한다. (단, 해당 노드의 가장 오른쪽 key보다 삽입하고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드를 'child'에 저장한다.)

② 'child'에서 key를 찾아 삭제한다.

③ ②과정 이후 'child'의 pair 개수가 (degree-1)/2보다 작다면 (underflow가 발생한 경우), 상황에 맞게 borrow 또는 merge를 한다.

3) 해당 노드의 child 노드가 leaf 노드가 아니라면,

① 해당 노드에서 삽입하고자 하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드로 이동하고 Node 변수 'child'에 저장한다. (단, 해당 노드의 가장 오른쪽 key보다 삽입하고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드로 이동하고 'child'에 저장한다.)

② ①과정 이후 'child'의 pair 개수가 (degree-1)/2보다 작다면 (underflow가 발생한 경우), 상황에 맞게 borrow 또는 merge를 한다.

6. key 검색

1) root 노드부터 시작해서 leaf 노드에 도달할 때까지 찾고자 하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드로 이동한다. (단, 해당 노드의 가장 오른쪽 key보다 찾고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드로 이동한다.)

2) leaf 노드에서 key를 찾는다.

<코드 상세 설명>

- bptree -

1. main

1) switch-case문을 이용해 args[0] 값에 따라 기능에 맞는 함수를 호출한다.

2. dataFileCreation

1) 입력받은 index file명을 이용해 indexFile을 만든다.

2) 입력받은 degree값을 indexFile에 입력한다.

3. saveInfo

1) ① index.dat 구조 :

degree
(해당 노드의 key 개수
해당 노드가 leaf인지 여부(leaf이면 1, non leaf이면 0)
key, value
key, value
:) - ⓐ

의 형태로 저장되는데, 괄호로 둘러싸인 ⓐ 부분은 하나의 노드에 대한 정보로, 같은 구조가 반복해서 저장된다.

② 노드는 root 노드부터 시작해서 recursive하게 저장된다.

- 현재노드 -> 가장 왼쪽 child노드 -> … ->가장 오른쪽 child 노드

2) saveInfo는 indexFile에 트리에 대한 정보를 저장하는 함수.
3) degree를 입력한 뒤 saveTree를 호출한다.

4) saveTree

① saveTree는 root노드부터 시작해서 '현재노드 -> 가장 왼쪽 child노드 -> … ->가장 오른쪽 child 노드' 순서로 ⓐ의 구조에 맞게 노드의 정보를 입력한다.
(단, 현재노드가 null이거나, key의 개수가 0이라면 아무것도 하지 않고 return한다.)

4. loadIndexFile

1) index file에 저장되어 있는 트리에 대한 정보를 로드한다.
2) degree를 입력받은 뒤, initTree를 호출한다.

3) initTree

① initTree는 root 노드부터 시작해서 '현재노드 -> 가장 왼쪽 child노드 -> … ->가장 오른쪽 child 노드' 순서로 indexFile에서 노드 정보를 로드한다.
(단, 현재 노드가 leaf인 경우는 해당 노드의 모든 child를 null로 설정하고 return한다.)
② 현재 노드가 leaf 노드가 아닌 경우, leaf node끼리 연결하기 위해 왼쪽 pair의 leftchild 중 가장 오른쪽에 있는 leaf 노드의 r을 오른쪽 pair의 leftchild 중 가장 왼쪽에 있는 leaf 노드로 설정한다.

5. insertion

1) data file에 있는 key, value 값을 가진 pair를 트리에 삽입한다.
2) loadIndexFile을 호출해 index file에 저장되어 있는 트리에 대한 정보를 로드한다.
3) data file의 key, value 정보를 한 줄씩 입력받고 insert를 호출한다.

4) insert

① 입력받은 key, value값을 tree에 삽입하는 함수
② root부터 시작해서 recursive하게 이동한다.
③ 해당 노드에 삽입하고자하는 key가 이미 존재한다면 "<Insertion Fail> key is already exist."라는 메시지를 출력하고 r을 return한다.
④ 해당 노드의 child 노드가 leaf 노드라면,
- 해당 노드에서 삽입하고자하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드를 Node 변수 'child'에 저장한다. (단, 해당 노드의 가장 오른쪽 key보다 삽입하고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드를 'child'에 저장한다.)
- 'child'에서 key가 삽입되어야 할 위치를 찾아 삽입한다.

⑤ 해당 노드의 child 노드가 leaf 노드가 아니라면,
- 해당 노드에서 삽입하고자하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드로 이동하고 Node 변수 'child'에 저장한다. (단, 해당 노드의 가장 오른쪽 key보다 삽입하고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드로 이동하고 'child'에 저장한다.)
  - 여기서 이동은 insert(key, value, child)를 호출하는 것을 말한다.
- 위의 과정 이후 'child'의 pair 개수가 degree이상이면 split을 호출한다.

⑥ 해당노드가 root 노드이면서 위의 과정이후 pair의 개수가 degree이상이라면 split을 호출한다.

5) split

① mid(==degree/2)를 기준으로 노드를 둘로 쪼갠다.
② c는 split할 노드, p는 c의 parent, c는 p의 index번째 pair의 leftChild이다.
③ c가 root 노드인 경우,
- 새로운 root 노드가 될 p에 c의 mid번째 pair를 추가한다. 이때 추가된 pair의 leftChild는 c로 설

정하고, p의 r의 r을 c의 r로 설정한다.
- c가 leaf 노드가 아닌 경우 c의 r을 c의 mid번째 pair의 leftChild로 설정한 뒤 mid번째 pair를 삭제하고, leaf 노드인 경우 c의 r을 p의 r로 설정한다.
- p의 r에 c의 mid번째 이후 pair들을 추가해준다.

④ c가 leaf node인 경우
- p의 index번째에 c의 mid번째 pair를 추가하고, 이때 추가한 pair의 leftChild는 c로 설정한다.
- tmpNode를 만들고, tmpNode의 r을 c의 r로 설정한다.
- tmpNode에 c의 mid번째 이후 pair들을 추가해준다.
- c의 r을 tmpNode로 설정하고, p의 index+1번째 pair의 leftChild를 tmpNode로 설정해준다.

⑤ c가 leaf가 아닌 경우,
- tmpNode를 만들고, tmpNode의 r을 c의 r로 설정한다.
- c의 mid+1번째 이후 pair들을 tmpNode에 추가해준다.
- c의 r을 c의 mid번째 pair의 leftChild로 설정한 뒤, c의 mid번째 pair를 p의 index번째에 추가해준다. 이때 추가한 pair의 leftChild를 c로 설정한다.
- p의 index+1번째 pair의 leftChild를 tmpNode로 설정하고 c의 mid번째 pair를 삭제한다.

+ 여기서 num번째 pair는 리스트에서 index가 num인 pair를 말한다.

6. deletion

1) data file에 있는 key 값을 가진 pair를 트리에서 삭제한다.
2) loadIndexFile을 호출해 index file에 저장되어 있는 트리에 대한 정보를 로드한다.
3) data file의 key를 입력받아 keyToDelete에 저장한다.
4) ver에 0을 저장하고, delete를 호출한다.
5) delete 호출 후 ver이 1이 되었다면 delete를 한 번 더 호출한다.

6) delete

① ver(전역변수, version)이 0인 경우 delete는 leaf노드에 있는 key를 삭제하고, ver이 1인 경우, leaf노드가 아닌 다른 노드에 있는 key를 삭제한다.

② root부터 시작해서 recursive하게 이동한다.

③ empty tree인 경우 “<Deletion Fail> tree is empty.”라는 메시지를 출력하고 return한다.

④ ver이 1일 때 해당노드의 i번째 pair에 삭제할 key가 있다면 노드의 오른쪽 sibling의 child 중 가장 작은 key를 가진 pair를 i번째에 추가하고 leftChild를 i+1번째 pair의 leftChild로 설정한 뒤 해당 노드에서 i+1번째 pair를 삭제한다.

⑤ 해당 노드가 root 노드이면서 leaf 노드라면 삭제할 key를 가진 pair를 찾아 삭제한다.
- 삭제할 key를 가진 pair가 존재하지 않는다면, “<Deletion Fail> key is not exist”라는 메시지를 출력하고 해당노드를 return한다.

⑥ 해당 노드의 child 노드가 leaf 노드라면,
- ver이 1인 경우 r을 return한다.
- 해당 노드에서 삭제하고자하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드를 Node 변수 ‘child’에 저장한다. (단, 해당 노드의 가장 오른쪽 key보다 삭제하고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드를 ‘child’에 저장한다.)
- ‘child’는 부모노드의 index번째 pair이다.
- ‘child’에서 삭제할 key를 가진 pair를 찾아 삭제한다.
    - 삭제할 key를 가진 pair가 ‘child’의 0번째 pair이고, ‘child’가 해당 노드(부모노드)의 0번째 pair의 leftChild가 아닌 경우,
        - ‘child’의 pair 개수가 1개라면 ‘child’에서 해당 pair를 삭제한다. 해당노드(부모노드)

의 pair 개수가 1이고, 'child'의 sibling의 pair 개수도 1이면 'child'의 sibling의 r을 child의 r로 설정하고, 부모노드의 r을 'child'의 sibling으로 설정한 뒤, 부모노드의 pair를 삭제한다.
- 'child'의 pair 개수가 1개가 아니라면 부모노드의 index번째에 'child'의 1번째 pair를 추가한다. 이때 추가된 pair의 leftChild를 부모노드의 index-1번째 pair의 leftChild로 설정한 뒤 index-1번째 pair를 제거한다. 그 후 'child'의 0번째 pair를 제거한다.

- 나머지 경우,
  - 'child'와 부모 노드와 'child'의 sibling 모두의 pair 개수가 1인 경우, 'child'에 sibling의 pair를 추가하고 'child'의 r을 sibling의 r로 설정한 뒤 부모노드의 r을 child로 설정한다. 부모노드와 'child'의 pair를 삭제하고 mergeLeafNode를 호출한다.
  - 아닌 경우, 'child'에서 삭제할 key를 가진 pair를 제거한다.
  - ver을 1증가시킨다.
- 위의 과정 이후 'child'의 pair개수가 (degree-1)/2 보다 작다면 상황에 맞게 borrowFromLeft, borrowFromRight, mergeLeafNode 중 하나를 호출한다.

- 'child'에서 삭제할 key를 가진 pair가 존재하지 않는다면 "<Deletion Fail> key is not exist"라는 메시지를 출력하고 부모노드를 return한다.

⑦ 해당 노드의 child 노드가 leaf 노드가 아니라면,
- 해당 노드에서 삭제하고자하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드로 이동하고 Node 변수 'child'에 할당한다. (단, 해당 노드의 가장 오른쪽 key보다 삭제하고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드로 이동하고 'child'에 할당한다.)
  - 여기서 이동은 insert(key, value, child)를 호출하는 것을 말한다.
- 위의 과정 이후 'child'의 pair 개수가 (degree-1)/2 보다 작다면 상황에 맞게 borrowFromLeft, borrowFromRight, mergeNonLeafNode 중 하나를 호출한다.

7) borrowFromLeft

① c의 왼쪽 sibling으로부터 pair를 빌려와 c에 추가한다. (c는 pair의 개수가 (degree-1)/2 보다 적은 노드이고, c는 p의 index번째 pair의 leftChild이다.)
② Node변수 sibling에 p의 index-1번째 pair의 leftChild를 저장한다.
③ c가 leaf node이면
- c의 0번째에 sibling의 마지막 pair를 추가하고 sibling에서 마지막 pair를 제거한다.
- p의 index번째에 c의 0번째 pair를 추가한다. 이때 추가된 pair의 leftChild는 sibling으로 설정한 뒤 p의 index-1번째 pair를 제거한다.

④ c가 non leaf node이면
- c의 0번째에 p의 index-1번째 pair를 추가한다. 이때 추가된 pair의 leftChild는 sibling의 r로 설정한다.
- p의 index번째에는 sibling의 마지막 pair를 추가한다. 이때 추가된 pair의 leftChild는 sibling으로 설정한다.
- p의 index-1번째 pair를 제거한 뒤 sibling의 r을 sibling의 마지막 pair의 leftChild로 설정한 후 이 pair를 sibling에서 제거한다.

8) borrowFromRight

① c의 오른쪽 sibling으로부터 pair를 빌려와 c에 추가한다. (c는 pair의 개수가 (degree-1)/2 보다 적은 노드이고, c는 p의 index번째 pair의 leftChild이다.)
② Node변수 sibling에 p의 index+1번째 pair의 leftChild를 저장한다.

③ c가 leaf 노드이면
- c에 sibling의 0번째 pair를 추가한 뒤, sibling에서 0번째 pair를 제거한다.
- p의 index번째에 sibling의 0번째 pair를 추가한다. 이때 추가된 pair의 leftChild는 c로 설정한다.
- p의 index+1번째 pair의 leftChild를 제거한다.
- c의 pair개수가 1개이고, index가 0이 아니라면 p의 index번째에 c의 0번째 pair를 추가한다. 이때 추가된 pair의 leftChild는 p의 index-1번째 pair의 leftChild로 설정한다. 그 후 p의 index-1번째 pair를 제거한다.

④ c가 leaf 노드가 아니면
- c에 p의 index번째 pair를 추가한다. 이때 추가된 pair의 leftChild는 c의 r로 설정한다.
- p의 index+1번째에 sibling의 0번째 pair를 추가한다. 이때 추가된 pair의 leftChild는 p의 index번째 pair의 leftChild로 설정하고 p의 index번째 pair를 제거한다.
- c의 r을 sibling의 0번째 pair의 leftChild로 설정하고, sibling에서 0번째 pair를 제거한다.

9) mergeLeafNode

① leaf 노드인 c를 sibling과 합친다. (c는 pair의 개수가 (degree-1)/2 보다 적은 노드이고, c는 p의 index번째 pair의 leftChild이다.)

② p가 root이고 p의 pair개수가 0 또는 1개인 경우, p의 child에 있는 모든 pair들을 p에 추가하고 p의 모든 child들을 null로 설정한다.

③ p가 root가 아닌데 p의 pair 개수가 0개인 경우 아무것도 하지 않고 p를 return한다. (p의 sibling으로부터 pair를 빌려오거나, sibling과 merge를 하도록)

④ index가 0이 아닌 경우,
- p의 index-1번째 child에 c의 모든 pair를 추가해준다.
- p의 index-1번째 pair의 leftChild의 r을 c의 r로 설정하고, p의 index번째 pair의 leftChild를 p의 index-1번째 pair의 leftChild로 설정해준 뒤 p의 index-1번째 pair를 제거한다.
- 위 과정 이후 p의 index-1번째 child의 pair의 개수가 degree이상이면 split을 호출한다.

⑤ index가 0인 경우,
- c에 p의 index+1번째 child의 모든 pair를 추가해준다.
- c의 r을 p의 index+1번째 child의 r로 설정하고, p의 index+1번째 pair의 leftChild를 c로 설정한 뒤 p의 index번째 pair를 제거한다.
- 위 과정 이후 p의 index번째 child의 pair의 개수가 degree이상이면 split을 호출한다.

10) mergeNonLeafNode

① non leaf 노드인 c를 sibling과 합친다. (c는 pair의 개수가 (degree-1)/2 보다 적은 노드이고, c는 p의 index번째 pair의 leftChild이다.)

② p가 root가 아니고 p의 pair 개수가 1개인 경우 p와 p의 0번째 child의 모든 pair를 p의 r에 추가한 뒤, p의 pair를 제거한다.

③ 나머지 경우
- index가 p의 pair개수와 같은 경우(c가 p의 r인 경우) c에 p의 index-1번째 child를 저장하고 index를 1감소시킨다.
- c에 p의 index번째 pair를 추가한다. 이때 추가된 pair의 r을 c의 r로 설정한다.
- p의 index+1번째 child의 모든 pair를 c에 추가하고 c의 r을 p의 index+1번째 child의 r로 설정한다.
- 위의 과정 이후 c의 pair 개수가 degree 이상이면 split을 호출한다.
- 그렇지 않은 경우,
    - p가 root이고, pair 개수가 1개이면 p에 c를 저장한다.
    - 위의 경우가 아니면 p의 index+1번째 pair의 leftChild를 c로 설정한 뒤 p의 index번째 pair

를 제거한다.

7. singleSearch

1) 입력받은 key값을 가진 pair를 찾아 value를 출력한다.
2) loadIndexFile을 호출해 index file에 저장되어 있는 트리에 대한 정보를 로드한다.
3) root가 null이라면 empty tree이므로 “NOT FOUND”라는 메시지를 출력하고 return한다.
4) search를 호출한다.

5) search

① ver이 0일 때(single key search) 입력받은 key값을 가진 leaf 노드의 pair를 찾아 value를 출력한다. pair를 찾는 과정에서 지나간 모든 노드의 key들을 출력한다. ver이 1일 때(range search) start key값을 가진 leaf 노드를 찾아 해당 노드를 return한다.
② while문을 이용해 Node 변수 n에 root 노드를 저장한 뒤 n이 leaf 노드가 될 때까지 찾고자 하는 key 이상의 key 중 가장 작은 key를 가진 pair의 left child 노드를 반복해서 저장한다. (단, 해당 노드의 가장 오른쪽 key보다 찾고자 하는 key가 더 크다면, node의 가장 오른쪽 child 노드를 저장한다.)
③ ver이 0이라면 ②의 과정에서 지나는 모든 노드의 모든 key들을 출력한다.
④ while문을 빠져나왔을 때(n에 leaf node가 저장되었을 때), ver이 0이라면 찾는 key가 있을 때 해당하는 value값을 출력하고, 찾는 key가 없을 때는 “NOT FOUND”라는 메시지를 출력한다.
⑤ n을 return한다.

8. rangeSearch

1) 입력받은 start key와 end key 사이에 존재하는 모든 key를 value와 함께 출력한다.
2) loadIndexFile을 호출해 index file에 저장되어 있는 트리에 대한 정보를 로드한다.
3) root가 null이라면 empty tree이므로 “<Ranged Search Fail> tree is empty.”라는 메시지를 출력하고 return한다.
4) ver을 1증가시키고 search를 호출해 return되는 값을 presentNode에 저장한다.
5) for문을 이용해 presentNode의 start key 이상, end key 이하의 key들을 value와 함께 출력한다. end key보다 큰 key가 나오면 0으로 초기화 되어있던 exit를 1 증가시키고 for문을 빠져나간다.
6) for문 종료 후 presentNode에 presentNode의 r을 저장하고 5)의 과정이 while문을 통해 반복된다.
① for문 종료 후, exit가 1이라면 while문을 빠져나간다.
② presentNode가 null이 되면 while문을 빠져나간다.

-Node-

1. addPair

1) p(pair 목록)에 Pair를 추가하고 m(pair 개수)를 1증가시킨다.
2) addPair(int key, int value) : key, value 값을 그대로 가지고 leftChild가 null인 Pair 객체를 p에 추가한다.
3) addPair(int key, int value, Node node) : key, value 값을 그대로 가지고 leftChild가 node인 Pair 객체를 p에 추가한다.
4) addPair(int key, int value, int index) : p의 index번째 위치에 key, value 값을 그대로 가지고 leftChild가 null인 Pair 객체를 추가한다.

2. removePair

1) p의 index번째에 위치한 pair를 제거하고 m을 1감소시킨다.

3. setLefhChild

1) index번째 pair의 leftChild를 node로 설정한다.
2) index가 m인 경우는 r을 node로 설정한다.

4. getLeftChild

1) index번째 pair의 leftChild를 리턴한다.
2) index가 m인 경우는 r을 리턴한다.

5. FindMostLeftLeaf

1) 현재노드의 child 중 가장 왼쪽에 있는 Leaf 노드를 return한다.

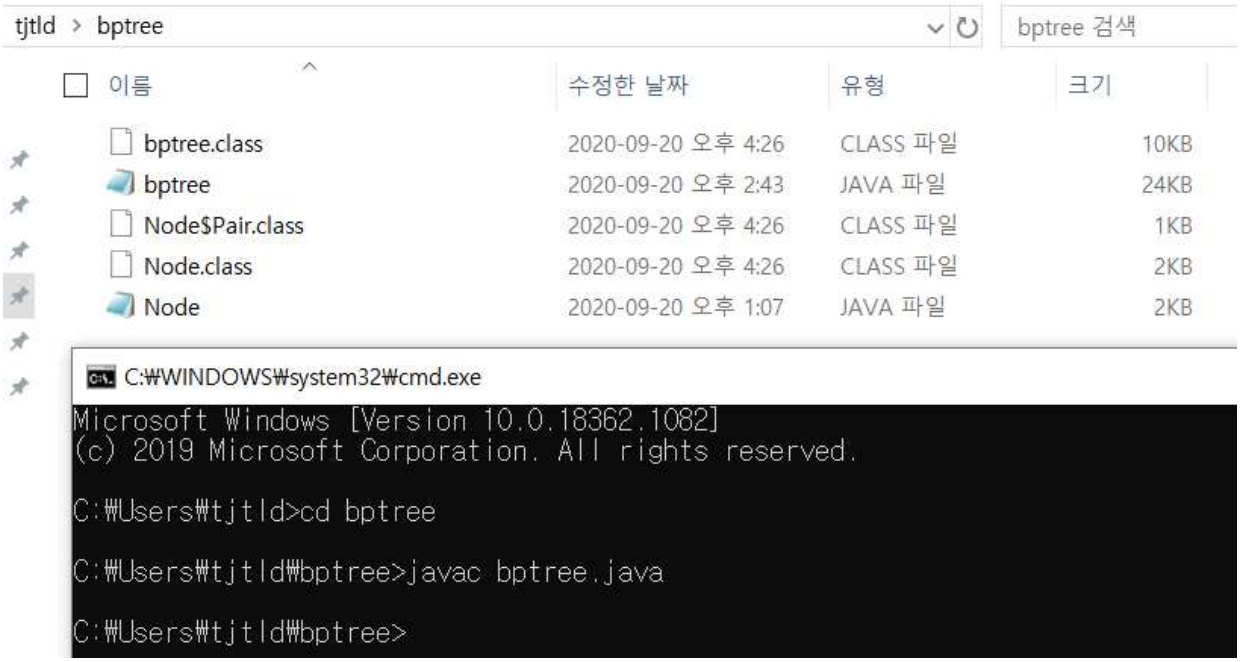
6. FindMostRightLeaf

1) 현재노드의 child 중 가장 오른쪽에 있는 Leaf 노드를 return한다.

<컴파일 및 실행 방법>

1. 실행파일(.jar) 만들기

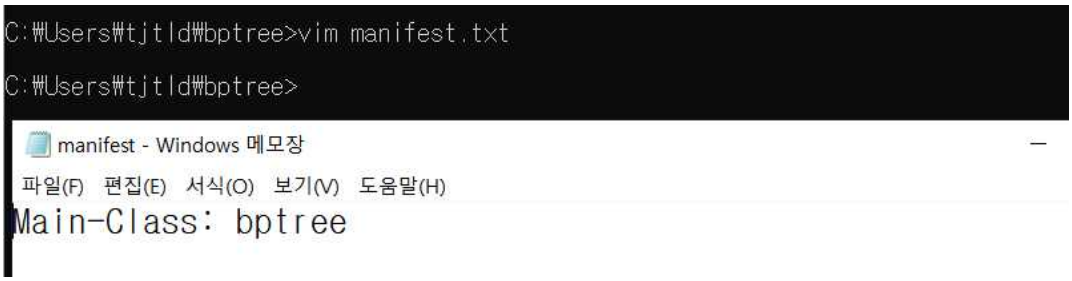
1) 소스코드가 있는 디렉토리에서 “javac bptree.java” 명령어 실행
   ① 아래 사진과 같이 bptree.class, Node.class, Node$Pair.class 파일이 생성된다.



2) “vim manifest.txt” 명령어 실행

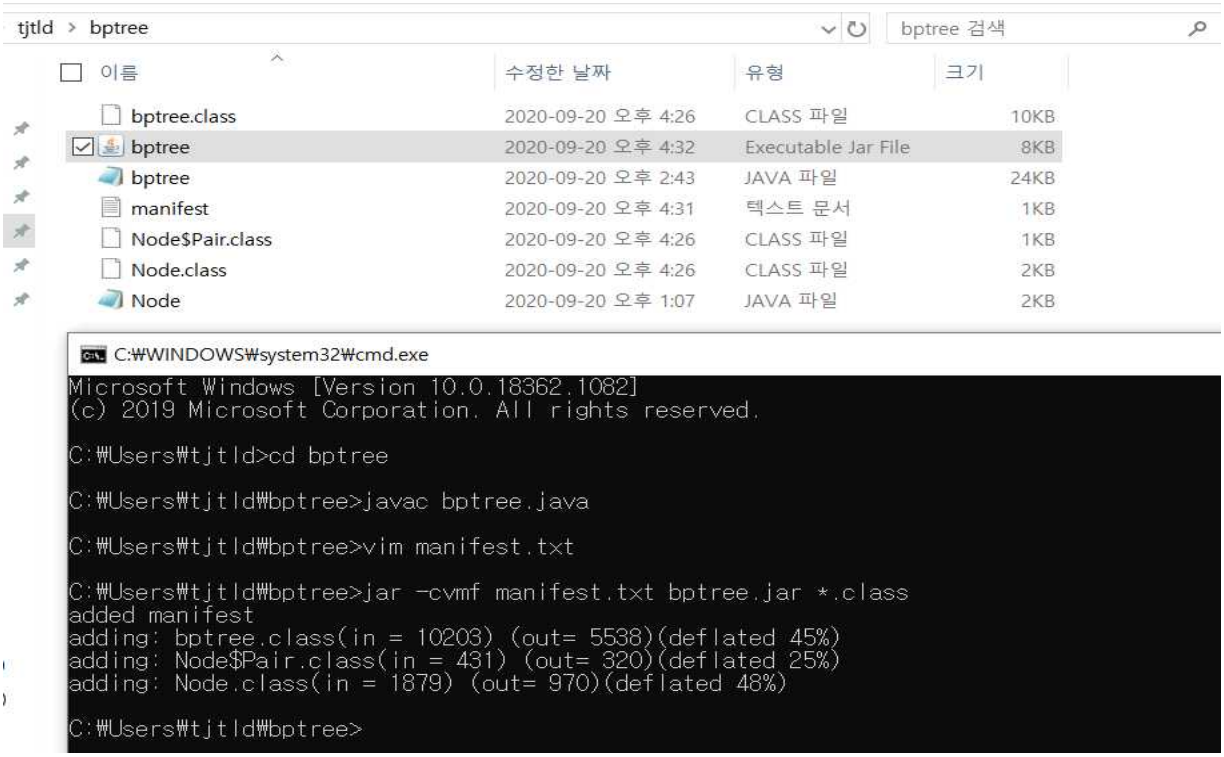
① manifest.txt는 main() 메소드가 어떤 클래스에 들어있는지 알려주는 파일

② manifest.txt에는 아래와 같이 [Main-Class: bptree]를 입력한다.



3) “jar -cvmf manifest.txt bptree.jar *.class” 명령어 실행

① 아래와 같이 bptree.jar 파일이 생성된다.



2. 실행

1) Data File Creation

① “java -jar bptree.jar -c [index_file] [b]” 명령어 실행

② ex) bptree -c index.dat 5

2) Insertion

① “java -jar bptree.jar -i [index_file] [data_file]” 명령어 실행

② ex) bptree -i index.dat input.csv

3) Deletion

① “java -jar bptree.jar -d [index_file] [date_file]” 명령어 실행

② ex) bptree -d index.dat delete.csv

4) Single Key Search

① “java -jar bptree.jar -s [index_file] [key]” 명령어 실행

② ex) bptree -s index.dat 10

5) Ranged Search

① “java -jar bptree.jar -r [index_file] [start_key] [end_key]” 명령어 실행

② ex) bptree -r index.dat 60 90

```
C:₩Users₩tjtld₩bptree>java -jar bptree.jar -c index.dat 3

C:₩Users₩tjtld₩bptree>java -jar bptree.jar -i index.dat input.csv

C:₩Users₩tjtld₩bptree>java -jar bptree.jar -d index.dat delete.csv

C:₩Users₩tjtld₩bptree>java -jar bptree.jar -s index.dat 10
68,86
NOT FOUND

C:₩Users₩tjtld₩bptree>java -jar bptree.jar -s index.dat 37
68,86
2132

C:₩Users₩tjtld₩bptree>java -jar bptree.jar -r index.dat 60 90
68,97321
84,431142
86,67945
87,984796
```