#Summary

하나의 class이지만, bptree 클래스 안에 inner class로 Node class, Pair class, Sort class가 있다. Node는 각각의 node를 위한 class이고, Pair은 Node를 구성하는 ArrayList를 이루기 위한 class이다. Sort class는 각각의 Node 내에서 오름차순으로 key 값을 정렬하기 위해 필요한 class이다.

static으로 Node형 root를 선언하였고, 변수 parent, r, node에 의해 모든 node가 결국 root에 직간접적으로 연결되어 있게 된다. 따라서 Serialize를 이용하여 root 객체를 저장하고 읽어오는 방식으로 구현하였다.

Node가 leaf node일 경우, node는 value를 담은 또 다른 Node 객체를 가리키고, non-leaf node일 경우 left child node를 가리킨다. 또한 leaf node의 r은 right sibiling node에 대한 pointer이고, non-leaf node의 r은 rightmost child node에 대한 pointer이다. 따라서 Node가 leaf node인지 non-leaf node인지 구분하는 것은 필요하지만, 같은 자료형을 가지는 같은 변수이기 때문에 따로 구현할 필요가 없다. is_root 함수와 is_leaf 변수를 이용하여 Node 객체가 leaf node인지 아닌지 구분해줄 수 있다.

bptree.java를 크게 4가지 파트로 나눌 수 있다.

1. Create

   새로운 index file을 생성하고, size b의 node를 생성한다. 가장 초기에 필요하다.

2. Insert

   tree에 key와 value를 삽입한다. insertion을 위해 key값에 적절한 leaf node를 찾아가는 함수(Search_leaf)가 필요하다. Insert는 overflow 여부, root 여부를 고려한 총 4가지 경우로 나눌 수 있다. 삽입 후 결과에서 overflow가 발생하더라도, 편의상 일단 삽입을 한 뒤 split을 하는 방식으로 구현하였다.

3. Single_Key_Search

   특정 key값에 해당하는 value를 찾는다. root부터 시작하여 node에 있는 key값들과 찾고자 하는 key값을 비교하며 한번이라도 탐색하는 node의 key들을 모두 출력하고, leaf node에 도달했을 경우 해당하는 value만 출력한다. 존재하지 않는다면 예외처리한다.

4. Ranged_Search

   특정 범위를 만족하는 key들을 모두 찾는다. start key로 leaf node를 search하고, 그 leaf node에서부터 leaf node의 r로 값을 갱신해주며 node의 key들을 탐색하며 end key 이하라면 출력한다.

5. main

   command line을 입력받고, 각각을 필요한 변수에 할당하여 명령을 실행한다.

#bptree class

bptree 클래스 내부에 Node 클래스, Pair 클래스, Sort 클래스가 있다.

#Node class

```
boolean is_leaf;
int Max_size;
Node parent;
int m; //# of keys
ArrayList<Pair> p; //an array of b <key, value> pairs
Node r; //pointer to the right
int value;
```

Node 클래스는 tree를 구성하고 있는 각 node들의 정보를 담기 위한 클래스이다. 변수로는

1. 특정 node가 leaf node인지 판별하기 위한 `is_leaf`
2. 노드에 담을 수 있는 key의 최대 개수를 뜻하는 `Max_size`
3. 부모 노드를 가리키는 `parent`
4. 현재 key의 개수를 뜻하는 `m`
5. key와 value 값의 pair들을 담고 있는 `p`
6. non-leaf 일 경우에는 rightmost child node를 가리키고, leaf 일 경우에는 right sibling node를 가리키는 `r`
7. node가 leaf node가 가리키고 있는 value node일 경우 value값을 담기 위한 `value`

가 있다.

```
public Node()
public Node(int v)
public Node(int v, int size)
```

생성자는 총 세 종류가 있다. 첫 번째 생성자는 기본 생성자이다. 두 번째 생성자는 int형 v를 인자로 받아 node의 `value` 값을 v로 설정해주는 생성자이다. 이 생성자는 `node`가 value node 일 때, 즉, leaf node가 가리키는 node일 때 호출된다. 세 번째 생성자는 int형 v와 int형 size를 인자로 받는다. 이 생성자는 `node`가 value node가 아닐 때 호출된다. node의 `m` 0으로 설정해주고, `Max_size`를 인자로 받은 size로 설정해준다. 그리고 size+1 크기의 `p`를 생성한다. 이 때 size+1 인 이유는 insert 구현을 편리하게 하기 위해서이고, insert 함수 파트에서 구체적으로 설명 할 것이다. 초기 `parent`는 null 이다.

#Pair class

```
public int key;
public Node node;
```

Pair 클래스는 각 노드를 구성하고 있는 요소들을 관리하기 위한 클래스이다. 변수로는 int형 `key`, Node형 `node`가 있다. `node`가 non-leaf 일 경우에는 <key, left_child_node>를 의미하고, leaf 일 경우에는 <key, value(or pointer to the value)>를 의미한다. 이 경우 실제 `value`는 leaf node가 가리키고 있는 하위 node에 담겨 있기 때문에, non-leaf와 동일하게 pair의 두 번째 변수를 Node형으로 설정한다.

```
public Pair()
public Pair(int k, Node n)
```

생성자는 총 두 종류가 있다. 첫 번째 생성자는 기본 생성자이다. 두 번째 생성자는 int형 k와 Node형 n을 인자로 받아, 해당 pair의 `key`를 k로 설정해주고 `node`를 n으로 할당해준다.

#Sort class

```
public Sort()
```

Sort 클래스는 하나의 node 내에서 key 값을 기준으로 pair들을 삽입할 때 오름차순 정렬을 유지하기 위한 클래스이다. 변수는 따로 없고, 기본 생성자 하나만 가지고 있다.

```
public int add_index(Node nd, int k)
```

`add_index` 함수는실제 sort를 위해 사용된다. 인자로 Node형 `nd`, int형 `k`를 받아 `nd`에서 `k`가 들어갈 위치를 return한다. 만약 `nd`의 `m`이 0이라면 index 0에 삽입하면 되므로 0을 return한다. 그렇지 않으면 `nd`의 `m`번 for문을 돌며, `nd`의 i번째 pair의 `key`가 k보다 작을 경우 현재 반복을 중단하고 i를 증가시키며 다음 반복으로 넘어가고, `k`가 `key`와 같거나 작을 경우 break하여 현재 i 값을 return한다.

```
static int b;
static Node root = null;
static Sort sort = new Sort();
```

bptree 클래스의 가장 핵심이 되는 변수이다. int형 `b`는 child node의 최대 수, 즉, node의 최대 `key` 개수 + 1 을 의미한다. Node형 `root`는 tree의 기반이 되는 변수이다. `root`를 이용하면 tree 전체를 탐색할 수 있기 때문이다. Sort 클래스 내의 함수를 사용하기 위해 Sort형 `sort` 객체를 미리 생성한다.

```
public static boolean is_root(Node nd)
```

`is_root` 함수는 Node형 nd를 인자로 받아 nd가 root node인지 아닌지 판별하는 함수이다. nd의 `parent`가 null이면 true를 return하고, 그렇지 않으면 false를 return한다.

```
public static Node Create(int size)
```

`Create` 함수는 bptree 실행시 -c 명령을 수행한다. int형 size를 인자로 받아 size를 `Max_size`로 하는 새로운 node를 생성하고, `root`에 할당한다. tree에 하나의 node만 있을 경우 root이자 leaf node가 되기 때문에, `is_leaf`를 true로 설정한다. 현재 아무 key도 들어가 있지 않기 때문에 `m`은 0, `parent`는 null로 설정하고, 인자로 받은 size를 `b`에 할당한다. 모든 설정이 끝난 `root`를 return한다.

```
public static Node Search_leaf(Node n, int k)
```

`Search_leaf` 함수는 Node형 `n`과 int형 `k`를 인자로 받아 `k`가 key로 들어있는 leaf node를 return한다. `n`이 leaf node라면 `n`을 return하고, 그렇지 않으면 for문을 `n`의 `m`만큼 반복한다. 만약 `k`가 `n`의 마지막 key값보다 크면 `n`의 `r`을 인자로 하는 `Search_leaf`를 재귀적으로 호출한다. `k`가 `n`의 i번째 key값보다 작다면 `n`의 i번째 pair의 node를 인자로 하는 `Search_leaf`를 재귀적으로 호출한다. `k`가 `n`의 i번째 key값보다 크거나 같다면 현재 반복을 중단하고 i를 증가시켜 다음 반복을 수행한다. 모든 경우에 해당하지 않는다면 `null`을 return한다.

```
public static void Insert(Node nd, int k, int v){
    Node tmp = new Node(1, b-1);
        tmp = Search_leaf(nd, k);
        Node value_node = new Node(v);
        int addIndex;
    ...
}
```

`Insert` 함수는 Node형 `nd`, int형 `k`, int형 `v`를 인자로 받아 기존의 tree에 삽입하는 함수이다. Node형 `tmp`를 생성하고, `nd`와 `k`를 인자로 하는 `Search_leaf` 함수를 호출하여 return 값을 `tmp`에 할당한다. 그리고 `v`를 인자로 하는 Node형 `value_node`를 생성한다. 오름차순 삽입을 위한 int형 `addIndex` 변수를 선언한다.

Insert를 구현하기 위해 경우를 크게 네 가지로 나누었다.

```
1. leaf node에 삽입할 공간이 있는 경우

2. leaf node에 삽입할 공간이 없는 경우

   1) insert할 node가 root인 경우

   2) insert할 node가 root가 아닌 경우

   • a. insert할 node의 parent에 삽입할 공간이 있는 경우

   • b. insert할 node의 parent에 삽입할 공간이 없는 경우
```

모든 경우에서 "sort의 `add_index` 함수를 호출하여 return 값을 `addIndex`에 할당한 후, `tmp`의 `p`의 `addIndex`번째에 `k`와 `value_node`를 원소로 하는 새로운 pair을 add 한다. add 후 `tmp`의 `key` 개수가 1 증가했으므로 `tmp`의 `m`을 1 증가시킨다." 라는 작업이 반복되므로, 편의 상 "add를 수행한다"라는 말로 대체할 것이다. 이 때, ArrayList의 특성 상 맨 끝에 add되지 않는 경우에도 `addIndex` 뒤의 pair들이 뒤로 한 칸씩 밀릴 것이다.

1. leaf node에 삽입할 공간이 있는 경우

   `tmp`의 `m`이 b-1보다 작을 때, 즉 insert를 하여도 overflow가 나지 않을 때는 leaf node에 삽입하기만 하면 된다. 만약 `tmp`의 `parent`가 null이면, `root`를 `tmp`로 설정한다. 그리고 `tmp`에 add를 수행한다.

2. leaf node에 삽입할 공간이 없는 경우 - 1) insert할 node가 root인 경우

   먼저 `tmp`에 add를 수행한다. int형 `index`를 선언하여 `tmp.m`/2 값을 할당한다. 이 값은 node가 overflow 상황일 때 split을 하기 위한 기준이 되는 index이다. 이 경우 `tmp`는 root로 유지할 것이고, 새로운 node 두 개를 새로 생성할 것이다. Node형 `new_left`와 `new_right`를 생성하고, 각각 `is_leaf`를 `true`로 설정하고 `parent`에 `tmp`를 할당한다. 그리고 `new_left`의 `r`은 `new_right`가 된다 .

새 node들을 설정한 후 for문을 통해 `index` 전까지의 원소들을 `new_left`에 복사하여 차례대로 add하고, `m`을 증가시킨다. 마찬가지로 `index`부터 마지막 원소까지 `new_right`에 복사하여 차례대로 add하고, `m`을 증가시킨다. `tmp`에서 index전까지의 원소들을 remove하고 나면 `tmp`의 `p`의 0번째에 `index`에 해당하는 pair가 있기 때문에, 반복문을 통해 `tmp`의 `key` 값이 하나만 존재할 때까지 `tmp`의 `p`의 1번째 pair을 remove한다. remove 할 때마다 `m`을 감소시킨다.

모든 node에서 `p` 설정이 완료된 후, `tmp`의 `is_leaf`를 `false`로 한다. 그리고 `tmp`의 `p`의 0번째 원소의 node를 `new_left`로 설정하고 `tmp`의 `r`을 `new_right`로 설정한다. root는 `tmp`로 유지된다.

3. -2) - a. insert할 node의 parent에 삽입할 공간이 있는 경우

   먼저 `tmp`에 add를 수행한다. int형 `index`를 선언하여 `tmp.m`/2 값을 할당한다. 이 값은 node가 split될 때 parent에 add될 것이다. 다음으로 Node형 `new_right`를 생성한다. `new_right`는 `tmp`가 split된 후 원소들이 담길 node이다. for문을 통해 `index`번째 원소부터 마지막 원소까지 `new_right`에 add하고, `tmp`에서 `index` 이후의 원소들을 remove한다. 부모 node에 들어갈 새로운 Pair형 `newToParent`를 생성한다. 이 pair는 `new_right`의 0번째 원소의 `key`와 `node`를 가진다. 그리고 `tmp`의 parent에 add를 수행한다. 만약 `addIndex`가 `tmp.parent`의 마지막 자리에 add된다면, `tmp.parent`의 `r`은 `new_right`가 된다. 부모노드에 삽입이 끝나면, 각각의 `node`를 연결해준다. `newToParent`가 위치한 index를 int형 `ParentIndexOfNew`를 선언하여 할당한다. 그 값이 0이라면 `newToParent.node`를 `ParentIndexOfNew`+1의 node로 설정하고, `ParentIndexOfNew`+1의 node는 그것의 `r`로 설정한다. 0이 아니라면 `ParentIndexOfNew`-1의 `node`의 `r`로 설정한다.

   그 후 Node형 `tmpNd`에 `newToParent`의 `node`의 `r`값을 할당하여 `ParentIndexOfNew`+1부터 parent의 `key` 개수만큼 for문을 돌려 연결되지 않은 나머지 pair들에 `node`를 연결한다. 만약 i가 마지막 반복이라면, `node`를 가질 수 있는 pair가 더 이상 존재하지 않는 상태이므로 parent의 `r`을 `tmpNd`로 설정한다.

4. -2) - b. insert할 node의 parent에 삽입할 공간이 없는 경우

   먼저 `tmp`에 add를 수행한다. `tmp`의 key의 개수가 `b`와 같을동안, 즉, `tmp`가 overflow인 상태이면 다음 실행을 반복한다.

   Pair형 `newToParent`를 선언하고, int형 `index`를 선언하여 `tmp.m`/2 값을 할당한다. 그리고 split을 위해 Node형 `new_right`를 생성한다. `tmp`가 root가 아니라면, `new_right`의 `parent`에 `tmp`의 `parent` 값을 대입한다. `tmp`가 leaf node라면, `new_right`의 `is_leaf`를 `true`로 설정한다. 그 후 위에서의 split 작업과 유사하게 반복문을 통해 `tmp`와 `new_right`를 split된 형태로 만든다. 만약 `tmp`가 leaf node이고, `tmp`의 `r`이 존재한다면 그 값을 `new_right`의 `r`에 할당한다. 그리고 `tmp`의 `r`은 `new_right`가 된다. `tmp`가 leaf node가 아니라면, `new_right`의 `r`에 `tmp`의 `r`값을 할당하고 `tmp`의 `r`은 `new_right`의 첫번째 원소의 `node`의 값을 갖는다.

   그 후 `tmp`가 root이면, 새로운 root를 만드는 작업을 한다. Node형 `newParentNd`를 생성하여 `tmp`와 `new_right`의 parent로 설정하고, root로 설정한다.

   다음으로 위에서와 같이 `newToParent`를 생성하여, `tmp`의 `parent`에 add를 수행한다. `tmp`가 leaf node가 아닐 경우, split하는 기준이 되었던 key는 해당 node에서는 사라지고 그 부모 node로 올라가기 때문에 `new_right`의 첫번째 원소를 삭제한다. 그리고 반복문을 통해 `new_right`와 연결된 자식 node들과 `r`의 `parent`를 `new_right`로 설정한다.

   int형 `ParentIndexOfNew`를 생성하여 `tmp`의 `parent`에서 `newToParent`의 `index`값을 대입한다. `tmp`의 `parent`의 key의 개수가 1개일 경우 그 원소의 `node`가 `tmp`가 되고, `parent`의 `r`은 `new_right`가 된다. 그 외의 경우, `ParentIndexOfNew`가 0이라면 `newToParent`의 `node`가 `ParentIndexOfNew`+1의 `node`값으로 되고, 0이 아니라면 `ParentIndexOfNew`-1의 `node`의 `r`값으로 된다. `newToParent`의 `node`를 연결해준 후, Node형 `tmpNd`를 선언하고 `tmp`가 leaf node일 때와 아닐 때로 경우를 나눈다.

`tmp`가 leaf node가 아닐 경우, `tmp`의 `parent`의 `ParentIndexOfNew`번째 원소의 `node`는 `tmp`가 된다. 그리고 만약 `ParentIndexOfNew`가 `parent`의 마지막 index라면 `parent`의 `r`을 `new_right`로 설정하고, 그 외의 경우 `parent`의 `ParentIndexOfNew`+1의 `node`를 `new_right`로 설정한다.

`tmp`가 leaf node일 경우, `tmpNd`를 `newToParent`의 `node`의 `r`로 한다. `ParentIndexOfNew`+1부터 `tmp`의 `parent`의 key 개수만큼 for문을 돌려, 나머지 `node`들과 `r`을 설정해주는 작업을 한다. 마지막 반복의 경우 `parent`의 `i`번째 원소가 존재하지 않으므로 `parent`의 `r`을 설정해준 후 반복을 중단한다. 그 외의 반복에서는 `tmpNd`를 `tmpNd`의 `r`로 갱신하며 `parent`의 원소 각각 `node`를 연결한다.

해당하는 경우에서의 실행이 끝나고, 다음 반복 전 `tmp`를 `tmp`의 `parent`로 갱신한다.

```
public static int Single_Key_Search(Node root, int k)
```

`Single_Key_Search` 함수는 Node형 `root`와 int형 `k`를 인자로 받아, `root`에서부터 `k`가 존재하는 leaf node까지 탐색한 과정을 print하고, `k`의 `value`를 return한다. 만약 `root`가 leaf node라면, `root`의 0번째 원소부터 마지막 원소까지 for문을 돌며 `k`와 같은 `key`를 가지면 그 `value`를 return, 그렇지 않으면 다음 반복을 수행한다. `k`가 존재하지 않는다면 -1을 return한다. `root`가 leaf node가 아니라면, `tmp`가 leaf node가 되기 전까지 반복문을 돌며 `tmp`에 있는 모든 key를 차례대로 print하고, `tmp`의 key와 `k`의 값을 비교하며 다음 탐색 node를 결정한다. 이후 `tmp`는 `k`가 존재 가능한 node가 된다. `tmp`의 key 개수만큼 for문을 돌며 `k`와 같은 key를 가진 pair을 찾고, 그 pair의 `value`를 return한다. `k`가 존재하지 않는다면 -1을 return한다.

```
public static ArrayList<Integer> Ranged_Search(Node root, int start, int end)
```

`Ranged_Search` 함수는 Node형 `root`와 int형 `start`, `end`를 인자로 받아 `start` 이상 `end` 이하의 key들을 모두 탐색하여 value와 함께 출력하고, key의 array를 return한다. 범위에 해당하는 key들을 저장할 Integer형 ArrayList `ranged_value`를 생성한다. `tmp`가 leaf node가 되기 전까지 반복문을 돌며 range search를 하기 위한 start node를 찾는다. 그 다음 while문을 돌며 `tmp`의 첫번째 key가 `end`보다 크면 반복문을 중단하고, 그렇지 않으면 for문을 돌며 `start`와 `end` 사이에 있는 값들을 `ranged_value`에 add하고, 해당 key와 value를 print한다. 반복이 끝난 후 `ranged_value`를 return한다.

```
public static void main(String[] args)
```

`main` 함수는 command line argument로 `args`를 입력받는다. `args[0]`은 실행할 명령의 `type`이 되고, `args[1]`은 index file의 이름이 된다. 입력받은 `args`의 길이가 3이고 `type`이 "-c"일 경우, `args[2]` 값은 `b`, 즉 tree의 degree가 된다. `type`이 "i"이거나 "-d"일 경우에는 `args[2]` 값은 input이나 delete file을 의미하는 `file_name`이다. 그 외의 경우(`type`이 "s"일 경우) `args[2]`는 찾고자하는 키 값인 `searchKey_Num`이다. 입력받은 `args`의 길이가 4일 경우(`type`이 "r"일 경우), `args[2]`의 값은 range search의 start key인 `Range_start`, `args[3]`의 값은 `range search`의 end key인 `Range_end`가 된다.

argument를 각각 변수에 대입한 후, 입력받은 명령의 `type`에 따라 함수를 호출한다. `type`이 "c"일 때, `Create`함수의 인자로 `b`를 넘겨주어 호출하고 return값을 `root`에 대입한다. `type`이 "i"일 때, int형 2차원 배열 `index`를 생성하여 csv파일의 ,를 기준으로 key값인 왼쪽 정수는 해당 `[row][0]`에, value값인 오른쪽 정수는 해당 `[row][1]`에 저장한다. 그 후 `root`, `index[row][0]`, `index[row][1]`을 인자로 하여 `Insert` 함수를 호출한다. `type`이 "d"일 때,

"i" 명령 실행과 유사한 방법으로 key만을 저장한 후 `root`와 `index[row][0]`을 인자로 하는 `Delete`함수를 호출한다. `type`이 "-s"일 때, `root`와 `searchKey_Num`을 인자로 하는 `Single_Key_Search`를 호출하여 return 값을 int형 `value`에 저장한다. `value`가 -1이면, 찾는 key 값이 tree에 존재하지 않았다는 뜻이므로 "NOT FOUND"를 출력하고, 그렇지 않으면 `value`를 출력한다. `type`이 "-r"일 때, `root`와 `Range_start`, `Range_end`를 인자로 하는 `Ranged_Search` 함수를 호출한다. 그 외의 명령이 입력되었을 경우 "Not valid Command Type"을 출력한다.

#Instructions for compliling bptree

1. B-tree_Assignment_2018008331.zip 파일을 저장한다.
2. cmd 창에서 "cd <파일 저장 경로>" 를 입력한다.
3. "javac bptree3.java" 로 컴파일한다. (필요시)
4. .class 파일 생성 후 "java bptree3 "를 입력한다.

   ex) java bptree3 -c index.dat 3