Name: Shivangi Singh
UFID: 66734099
UF Email account: shivangisingh91@ufl.edu

B+ tree Implementation in Java

Implemented using three classes namely: node, bplustree, treesearch.

Class 'node' contains the structure of each node of tree.

Class 'bplustree' contains the actual implementation.

Class 'treesearch' contains the main function to create object of bplustree class and checks the implementation.

Program Structure

Node class has following structure:

- **Leaf**: Boolean to check if index or leaf node
- **Left, right**: pointer to left/right node for doubly linked list (only for leaf node)
- **Parent**: pointer to its parent node
- **Keys**: List of double type for storing keys in a node. List size of keys should be less than the order.
- **Values**: List of List of String type for storing values in a node. List size of list of string should be less than the order. Each key at an index corresponds to the same index of List of List of string. That is key at index 'i' of node have a value(or list of values) of List<String> at 'i' index of 'values' variable. ***Duplicates are handled by keeping List of strings for each key.***
- **Children**: List of nodes for storing left and right child of a key with key at index 'i' of node having left child at index 'i' of children and right child at index 'i+1' of children.

Bplustree class has following structure:

Global static variable:

- **root** : root of tree
- **order** : order of tree
- **ch1** : index of last element of left half after split
- **ch2** : index of first element of right half after split

Methods:

- **initialise** : to initalise order, ch1, ch2.
- **addontononleaf:** It add the key after split to its parent. And if its parent node (index node) size also is increased to 'order' after key adding, it recursively calls "splitnonleaf" and itself ('addontononleaf') until parent node size is less than 'order'.
- **splitleaf** : it splits the leaf node in two halves and maintains the B+ tree property of each node having minimum 'ceil(order/2) keys' and maximum 'order-1' keys.
  After splitting the leaf node it calls "addontononleaf" function, which add the key to its parent. And if its parent node (index node) size also is increased to 'order' after key adding, it recursively calls "splitnonleaf" and itself ('addontononleaf').
- **splitnonleaf** : it splits the index node (after being called by addontononleaf in case index size == order) in two halves and maintains the B+ tree property.
  After splitting the index node it calls "addontononleaf" function, which add the key to its parent. And if its parent node (index node) size also is increased to 'order' after key adding, it recursively calls "splitnonleaf" and itself('addontononleaf').
- **inssearch:** It takes in root/non-leaf node and returns the appropriate leaf-node.
  It is called by both insert and search to reach desired leaf node.
- **Insert**:
  a) insert new root (first insert)
  b) insert to leaf-node i.e. when root is leaf node.
  c) Insert when root is non-leaf node. (this internally calls '*inssearch*' to get from root to appropriate leaf node)

  Whenever after insert, number of keys in node becomes equal to order, node is split by calling "*splitleaf*" function. And the key is added to index parent node.
- **Search**:
  a) When root is leaf, it searches for the key in the root node.
  b) When root is index, it calls *"inssearch"* to get to desired leaf node, where it looks for key.
  Since values are stored in List<List<String>> , key at index 'i' of List of keys stores its respective values in List<String> at index 'i' of List of List<String> , hence handling duplicates.
- **Search(key1, key2):** Same as search of key =key1, after finding key in leaf-node with key= key1 or just greater than key1, it

outputs all the keys and values by traversing the doubly linked list until key <= key2.

Treesearch class is for testing the B+ tree implementation by reading input from input file and writing the output to output text file.

Source Code:

- **Class 1: node**

```
package bplustree;
import java.util.*;

public class node {
	public boolean leaf=false;                                    //
true for leaf nodes
	List<Double> keys= new ArrayList<Double>();            //
arraylist to store keys in each node
	List<List<String>> values= new ArrayList<List<String>>();
	// arraylist of list of string to store values key.get(i)
corresponds to values.get(i)'s list of values and hence storing
duplicates in list
	List<node> children = new ArrayList<node>();
	// list of class 'node' type to store children ---- key.get(i)'s
left child is children.get(i)  and key.get(i)'s right child is
children.get(i+1)
	public node left=null, right=null;
	// left & right for double linked list, null for index nodes
	public node parent=null;
	// parent pointer for each node
	//constructor
	public node(){
		leaf= false;
		left=null;
		right=null;
		parent=null;
	}
}
```

- **Class 2: bplustree**

```
package bplustree;

import java.util.*;
import java.util.*;
import java.io.File;

public class bplustree {
      private static int order;
      private static node root;
      private static int ch1, ch2 ;

      // initialize with order 'm'
      public void initialise(int m)
      {
            order=m;
            ch1= (int)m/2-1;                  // calculating 'ch1' for
split node index during node split L= 0 to ch1 & R= ch2 to order,
equivalent to ceil function
            ch2= ch1+1;                                    // calculating
'ch2' for split node index
      }


      public void insert(double key, String value)
      {
            insert_r(key, value, root);                        //calling
insert_r function to insert in tree rooted at 'root'
      }



      public String search(double key)
      {
            String s=search_r(key, root);                      //calling
search_r function to insert in tree rooted at 'root'
            return s;
      }


      // function for search with single key

      public String search_r(double key,  node r)
      {
            String s="";                       // storing all values in
String
            if(r.leaf==false){                 // when root is not leaf
                  int j=0;
                  while(j<r.keys.size() && r.keys.get(j)< key) j++;
      // traversing keys in node
                  if(j<r.keys.size() && r.keys.get(j)== key)
      // since r.keys.get(j) == key, its right child
```

```
(r.children.get(j+1)) is to be checked as it contains keys >=
r.keys.get(i)
                        r=inssearch(key, r.children.get(j+1));
                  else
                        r=inssearch(key, r.children.get(j));
      //since r.keys.get(j) > key, its left child (r.children.get(j))
is to be checked as it contains keys < r.keys.get(i)
            }

            //  r is leaf node either after returning from 'inssearch'
func or root was leaf
                  int i=0;
                  while(i<r.keys.size() && r.keys.get(i)< key) i++;
                  if( i<r.keys.size() && r.keys.get(i) == key){
      // if key found, storing the list of all the values
                        for(int y=0; y< r.values.get(i).size(); y++){
                              s+= r.values.get(i).get(y)+",";
                        }
                  }
                  else{
                  // else return null as key not found
                        return "Null";
                  }

                  return s.substring(0,s.length()-1) ;
      }




      public String search(double key1, double key2)
      // search range of keys
      {
            String s="";
            node r=root;

            if(r.leaf==false){
            // root is not leaf
                  int j=0;
                  while(j<r.keys.size() && r.keys.get(j)< key1) j++;
      // traversing keys in node
                  if(j<r.keys.size() && r.keys.get(j)== key1)
                        r=inssearch(key1, r.children.get(j+1));
      // since r.keys.get(j) == key, its right child
(r.children.get(j+1)) is to be checked as it contains keys >=
r.keys.get(i)
                  else
                        r=inssearch(key1, r.children.get(j));
      //since r.keys.get(j) == key, its left child (r.children.get(j))
is to be checked as it contains keys < r.keys.get(i)
            }
```

```
          // here r is leaf node either after returning from
'inssearch' func or root was leaf
                int i=0;
                while(i<r.keys.size() && r.keys.get(i)< key1) i++;
                double a= i<r.keys.size() ? r.keys.get(i):
r.right.keys.get(0);       // checking corner case when
i==r.keys.size()
                node h= i<r.keys.size() ? r: r.right;
                i=i<r.keys.size() ? i:0;
                r=h;
                node p=r;
                // traversing and storing values using doubly
linkedlist of 'node'
                while(a<= key2){

                      for(int y=0; y< r.values.get(i).size(); y++){

                            s+="("+ r.keys.get(i)+
","+r.values.get(i).get(y) + "),";          // storing key and values
in string
                      }
                      if(i+1 >= r.keys.size() && r.right==null){
                            break;
                      }
                      node k= i+1<r.keys.size() ? r: r.right;
                                  //checking right node when all keys in
node are checked
                      i=i+1<r.keys.size() ? i+1:0;

                      r=k;
                      a= r.keys.get(i);
                }
          return s.length()<=0? "Null" : s.substring(0, s.length()-
1);
     }



     public void insert_r(double key, String val, node r)
     {
          if(r==null)                 // calling during 'first' insert
when root==null   & initializing tree
          {
                node k= new node();          // allocating memory to
node
                k.leaf=true;
                k.parent=null;
                k.keys.add(key);           //adding key at '0' index
of list
                List<String> o= new ArrayList<String>();
                o.add(val);
```

```
                k.values.add(o);                        //adding
List<values for a key> at '0' index of list<List<>>
                root=k;                                          //
root is initialized
                return;
           }
           if(r.leaf==false)                    // when root is not
leaf, exploring child nodes
           {
                while(r.leaf== false){          // can be if
                     r= inssearch( key,r);      // returns
corresponding leaf node where key can be inserted
                }
           }
           // here r is leaf node either after returning from
'inssearch' func or root was leaf
           if(r.leaf==true)
           {
                int i=0;
                while(i<r.keys.size() && r.keys.get(i)< key) i++;
     // for finding index where key can be added
                if(i<r.keys.size() && r.keys.get(i)== key){
     // when key already existed (duplicate case) , add value to the
list
                     r.values.get(i).add(val);
                }
                else{
           // when key was not found (non-duplicate case) , create a
list and add value to the list
                r.keys.add(i,key);
                List<String> o= new ArrayList<String>();
                o.add(val);
                r.values.add(i,o);
                }
                //after insert if number of keys in nodes < 'order'
return as no need of splitting
                if(r.keys.size()<=order-1){
                     return;
                     }
                else{      //after insert if number of keys in nodes
== order split the node/leaf
                node carryup= splitleaf(r);      // calling
'splitleaf' function
                }
           }
           return;
     }



     // adding the key and left & right children to non-leaf node in
case of split
```

```
     public node addontononleaf(double key, node l, node r, node par)
// can be void return

     {
          if(par==null)                                   // when
splitting root, root.parent is null, therefore creating new root
          {
               node temp= new node();                     // allocating
memory
               temp.leaf=false;
               temp.left=temp.right=temp.parent=null;
               temp.keys.add(key);
               temp.children.add(l);                      // adding to
nodes children at index '0'
               temp.children.add(r);                      // adding to
nodes children at index '1'
               root=temp;
               l.parent=r.parent=root;                         //
assigning parent pointer of left and right children to present
node/root
               return temp;
          }
          else
          {                                                         //
when index node is not root, adding key after split
               int i=0;
               while(i<par.keys.size() && par.keys.get(i)<= key) i++;
// adding the key to the parent node at appropriate index
               par.keys.add(i,key);

               par.children.remove(i);
               // removing overflow child (which was split)
               par.children.add(i,l);
          // adding left child after split
               par.children.add(i+1,r);
          // adding right children after split
               l.parent=r.parent=par;
               // assigning parent pointer of left and right children
to present node/root
               if(par.keys.size()<=order-1){
          // if index node size < order after adding key   return

                    return par;
               }
               else{
                    node k = splitnonleaf(par);
          // if index node size == order after adding key, split
index node by calling 'splitnonleaf' func
                    return k;
               }
          }
     }
```

```
     // called when splitting index node as node does not contain
values & left, right pointers are null
     public node splitnonleaf(node r){                        // can be
void return
          node a= new node();                                      //
splitting index node into node 'a' & 'b'
          node b= new node();
          a.right=b.left=b.right= a.left= null;
          a.leaf=b.leaf=false;
          a.parent=b.parent= r.parent;                        //
redundant

          int s= r.keys.size();
          a.keys= new ArrayList<>(r.keys.subList(0, ch2));       //
assigning left 'keys' of index node to a
          b.keys= new ArrayList<>(r.keys.subList(ch2+1, s));
     // assigning left 'keys' of index node to a

          a.children=new ArrayList<>(r.children.subList(0, ch2+1));
     // assigning corresponding left children List to a.children
          b.children=new ArrayList<>(r.children.subList(ch2+1,s+1));
     // assigning corresponding right children List to b.children
          for(int n=0; n<a.children.size(); n++){
               // changing parent pointers of all the children nodes
of node 'a' to 'a'
               a.children.get(n).parent=a;
          }
          for(int n=0; n<b.children.size(); n++){
               // changing parent pointers of all the children nodes
of node 'b' to 'b'
               b.children.get(n).parent=b;
          }
          node g=r.parent;
          double x=r.keys.get(ch2);

          node kk = addontononleaf(x, a,b, g);
     // add the key after split to parent
          return kk;
     }



     // called when splitting leaf node as node contains values &
doubly linked list to be maintained
     public node splitleaf(node r){
     // can be void
          node a= new node();
          node b= new node();
```

```
            // manages doubly linked list pointer
            a.right= b;
            b.left=a;
            b.right= r.right;
            a.left=r.left;
            if(r.left!=null){
                  r.left.right= a;
            }
            if(r.right!=null){
                  r.right.left= b;
            }
            a.leaf=b.leaf=true;
            a.parent=b.parent= r.parent;  /// redundant
            int s= r.keys.size();
            a.keys= new ArrayList<>(r.keys.subList(0, ch2));
      // assigning left 'keys' of leaf node to a
            b.keys= new ArrayList<>(r.keys.subList(ch2, s));
      // assigning right 'keys' of leaf node to b
            a.values= new ArrayList<>(r.values.subList(0, ch2));
      // assigning left 'values' of leaf node to a
            b.values= new ArrayList<>(r.values.subList(ch2, s));
      // assigning left 'values' of leaf node to a
            node g= r.parent;
            r=null;
            node kk = addontononleaf(b.keys.get(0), a,b, g);       //
add the key after split to parent

            return kk;
      }


// for traversing tree from root to leaf for appropriate leaf-node
public node inssearch(double key,  node r)
{
      if(r.leaf==false){
            int j=0;
            while(j<r.keys.size() && r.keys.get(j)< key) j++;
      // traversing node
            if(j<r.keys.size() && r.keys.get(j)== key)
                  r=inssearch(key, r.children.get(j+1));      // since
r.keys.get(j) == key, its right child (r.children.get(j+1)) is to be
checked as it contains keys >= r.keys.get(i)
            else
                  r=inssearch(key, r.children.get(j));        // since
r.keys.get(j) > key, its left child (r.children.get(j)) is to be
checked as it contains keys < r.keys.get(i)
      }
            return r;
      //return leaf-node
}
```

```
}
```

- **Class 3: treesearch (containing main method)**

```
package bplustree;
import java.util.*;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;

public class treesearch {

      public static void main(String[] args) {
            // TODO Auto-generated method stub
            bplustree b=new bplustree();                        //
creating object of class bplustree
            String f=args[0];

         try {

              File file = new File(f);                          //creating
file object
              Scanner input = new Scanner(file);                // Scanner
object for input from file
              b.initialise(Integer.parseInt(input.nextLine()));
      // intialising tree with degree/order
              while (input.hasNextLine()) {
                  String line = input.nextLine();
      //reading lines from file
                  System.out.println(line);
                  if(line.charAt(0)== 'S'){
      // for Searching key
                      String res="";
                   String[] st= line.split(",");
                   if(st.length > 1)
      // for range search
                   {
                        int i=0;
                        while(st[0].charAt(i)!='(') i++;
                        String ss=st[0].substring(i+1);
                        double key1= Double.parseDouble(ss);
                        ss=st[1].substring(0,st[1].length()-1 );
                        double key2= Double.parseDouble(ss);
                        res+=b.search(key1, key2) + "\n";
      //result in string format
                   }
```

```
                else{
      //for searching single key
                    int i=0;
                    while(st[0].charAt(i)!='(') i++;
                    String ss=st[0].substring(i+1, st[0].length()-1);
                    double key= Double.parseDouble(ss);
                    res+=b.search(key)+"\n";
      //result in string format

                }
                    try{
                        File ofile =new File("output_file.txt");
//creating output file object
                        if(!ofile.exists()){
                            ofile.createNewFile();
                        }
                            FileWriter fileWritter = new
FileWriter(ofile,true);      // opening in append mode
                            BufferedWriter bufferWritter = new
BufferedWriter(fileWritter);
                            bufferWritter.write(res);
                                 // writing to file
                            bufferWritter.close();
                            fileWritter.close();
                    }catch(Exception e){
                        e.printStackTrace();
                    }
               }
                if(line.charAt(0)== 'I'){        // for inserting key
and value
                int i=0;
                String[] st= line.split(",");
                while(st[0].charAt(i)!='(') i++;
                double key= Double.parseDouble(st[0].substring(i+1));
                String val= st[1].substring(0,st[1].length()-1);
                b.insert(key, val);
                }
            }
            input.close();

        } catch (Exception ex) {
            ex.printStackTrace();
        }
      }
}
```