COP 5536 - ADVANCED DATA STRUCTURES SPRING 2021

PROGRAMMING PROJECT ON B+ TREES

Submitted by:
Name- Budigi Sreelalitha
UFID - 36717336
Email- sbudigi@ufl.edu

PROBLEM DESCRIPTION

- The main aim of a B+ tree is to store data in a block-oriented storage context, such as files structures, for efficient retrieval.
- We build and evaluate a limited degree B+ tree for internal-memory dictionaries as part of this research (i.e. the entire tree resides in main memory).
- The data is given in the form (key, value) with no duplicates, and we store the data pairs in an m-way B+ tree.
- Only the leaf nodes in a B+ tree contain real values, and the leaves can be connected into a doubly-linked list.

FUNCTIONS IMPLEMENTED

1. Initialization of the order of the B+ Tree.
2. Insertion of the new key value pair in the tree.
3. Delete the key value pair in the tree.
4. Search the given key's value.
5. Search the values between given keys.

PROJECT STRUCTURE

The whole project is made up of 3 files which are:

1. ***Source Code (bplustree.java):*** Source Code contains all the required code for the B+ Tree implementation. It contains all functions like
    - insert()
    - delete()
    - search(key)
    - search(key1, key2)

2. ***Input File:*** Input file contains the {key, value} pairs of the data to be acted upon. The first line in the input file Initialize(m) means creating a B+ tree with the order m where m may be different depending on input file. Each of the remaining lines specifies a B+ tree operation.
    - Insert{key, value}
    - Delete{key}
    - Search{key}
    - Search{key1, key2}

Example:

Initialize(3)
Insert(21,0.3534)
Insert(108,31.907)
Insert(56089, 3.26)
Insert(234, 121.56)
Insert(4325, -109.23)
Delete(108)
Search(234)
Insert(102, 39.56)
Insert(65, -3.95)
Delete(102)
Delete(21)
Insert(106, -3.91)
Insert(23, 3.55)
Search(23, 99)
Insert(32, 0.02)
Insert(220, 3.55)
Search(33)
Delete(234)
Search(65)

3. ***Output File:*** Output file contains all the output generated when the source code is executed. For Initialize, Insert and Delete query we do not produce any output. For a Search query we output the results on a single line using commas to separate values. The output for each search query will be on a new line. All output is written to "output_file.txt". If a search query does not return anything we output "Null".

Example:

121.56
3.55, -3.95
Null
-3.95

FUNCTION PROTOTYPES:

INSERTION:

In the B+ tree, the Insert function is used to insert the specified key-value pair. If the given key-value pair is the first insertion in the tree, we will establish a leaf node that will be the tree's first node; otherwise, we will insert the given key-value pair into the parent and change the parent's siblings to match the whole tree.

```
public void insert(int key, double value)
	{
		if (checkIfEmpty())
		{
			/* For first insert only */
			// Create first leaf node
			LeafNodeClass list_dictionary = new LeafNodeClass(this.order_m, new DictionaryPairClass(key, value));
			this.first_leaf_node = list_dictionary;
		}
		else
		{
			// Find leaf node to insert
			LeafNodeClass list_dictionary = (this.root_node == null) ? this.first_leaf_node : searchForLeafNode(key);
			// If node becomes overfull
			if (!list_dictionary.insert(new DictionaryPairClass(key, value)))
			{
				// Sort the pair to be inserted
				list_dictionary.dictionary_list[list_dictionary.number_of_pairs] = new DictionaryPairClass(key, value);
				list_dictionary.number_of_pairs++;
				sortDictionaryList(list_dictionary.dictionary_list);
				// Divide the sorted pairs
				int center_point = findCenterPoint();
				DictionaryPairClass[] copy_half_of_dict = splitDictionaryList(list_dictionary, center_point);
				if (list_dictionary.parent_node == null)
				{
					/* Already one node present */
					// Create node to serve as parent
					Integer[] parent_keys = new Integer[this.order_m];
					parent_keys[0] = copy_half_of_dict[0].key;
					InternalNodeClass parent_node = new InternalNodeClass(this.order_m, parent_keys);
					list_dictionary.parent_node = parent_node;
					parent_node.appendingToChildPointer(list_dictionary);
				}
				else
				{
					/* Parent exists */
					// Add new key for proper indexing
					int new_parent_key = copy_half_of_dict[0].key;
					list_dictionary.parent_node.keys_collection[list_dictionary.parent_node.degree_value - 1] = new_parent_key;
					Arrays.sort(list_dictionary.parent_node.keys_collection, 0, list_dictionary.parent_node.degree_value);
				}
				// To hold the other half
				LeafNodeClass new_leaf_node = new LeafNodeClass(this.order_m, copy_half_of_dict, list_dictionary.parent_node);
				int index_of_pointer = list_dictionary.parent_node.findIndexOfPointer(list_dictionary) + 1;
				list_dictionary.parent_node.insertionWithinChildPointer(new_leaf_node, index_of_pointer);
				// Make leaf nodes as siblings
				new_leaf_node.rightside_ibling = list_dictionary.rightside_ibling;
				if (new_leaf_node.rightside_ibling != null)
				{
					new_leaf_node.rightside_ibling.leftside_sibling = new_leaf_node;
				}
				list_dictionary.rightside_ibling = new_leaf_node;
				new_leaf_node.leftside_sibling = list_dictionary;
				if (this.root_node == null)
				{
					// Make root as parent
```

```
                                        this.root_node = list_dictionary.parent_node;

                                }
                                else
                                {
                                        /* If parent is full we repeat till no deficiencies are found */
                                        InternalNodeClass internal_node = list_dictionary.parent_node;
                                        while (internal_node != null)
                                        {
                                                if (internal_node.checkIfOverfull())
                                                {
                                                        dividingInternalNode(internal_node);
                                                }
                                                else
                                                {
                                                        break;
                                                }
                                                internal_node = internal_node.parent_node;
                                        }
                                }
                        }
                }
        }
```

DELETION:

Based on the input key given to the function, this method will delete a key-value pair from the B+ tree. The delete operation is not performed if the tree is empty or if the key provided as input cannot be found. If the given key is valid, the key-value pair in question is removed from the tree. We must search for deficiencies after the deletion process is completed.

```
public void delete(int key)
        {
                if (checkIfEmpty())
                {
                        /* If tree empty */
                        System.err.println("The tree is empty.");
                }
                else
                {
                        // Find index of key to delete
                        LeafNodeClass list_dictionary = (this.root_node == null) ? this.first_leaf_node : searchForLeafNode(key);
                        int dict_pair_index = binarySearch(list_dictionary.dictionary_list, list_dictionary.number_of_pairs, key);
                        if (dict_pair_index < 0)
                        {
                                /* When key not found */
                                System.err.println("Key not found.");
                        }
                        else
                        {
                                list_dictionary.delete(dict_pair_index);
                                // Check deficiency
                                if (list_dictionary.checkIfDeficient())
                                {
                                        LeafNodeClass sibling_node;
                                        InternalNodeClass parent_node = list_dictionary.parent_node;
                                        // Borrow
                                        if (list_dictionary.leftside_sibling != null &&
                                                list_dictionary.leftside_sibling.parent_node == list_dictionary.parent_node &&
                                                list_dictionary.leftside_sibling.checkIfLendable())
                                                {
                                                sibling_node = list_dictionary.leftside_sibling;
                                                DictionaryPairClass borrowed_dict_pair = sibling_node.dictionary_list[sibling_node.number_of_pairs - 1];
                                                /* Insert borrowed pair, sort and delete the pair */
                                                list_dictionary.insert(borrowed_dict_pair);
                                                sortDictionaryList(list_dictionary.dictionary_list);
                                                sibling_node.delete(sibling_node.number_of_pairs - 1);
                                                // Update key
                                                int index_of_pointer = findIndexOfPointer(parent_node.pointers_for_child_nodes, list_dictionary);
                                                if (!(borrowed_dict_pair.key >= parent_node.keys_collection[index_of_pointer - 1]))
                                                {
                                                        parent_node.keys_collection[index_of_pointer - 1] = list_dictionary.dictionary_list[0].key;
                                                }
                                        }
                                        else if (list_dictionary.rightside_ibling != null &&
                                                        list_dictionary.rightside_ibling.parent_node == list_dictionary.parent_node &&
                                                        list_dictionary.rightside_ibling.checkIfLendable())
                                                        {
                                                sibling_node = list_dictionary.rightside_ibling;
                                                DictionaryPairClass borrowed_dict_pair = sibling_node.dictionary_list[0];
                                                /* Insert borrowed pair, sort and delete the pair */
                                                list_dictionary.insert(borrowed_dict_pair);
                                                sibling_node.delete(0);
```

```
                                        sortDictionaryList(sibling_node.dictionary_list);
                                        // Update key
                                        int index_of_pointer = findIndexOfPointer(parent_node.pointers_for_child_nodes, list_dictionary);
                                        if (!(borrowed_dict_pair.key < parent_node.keys_collection[index_of_pointer]))
                                        {
                                                parent_node.keys_collection[index_of_pointer] = sibling_node.dictionary_list[0].key;
                                        }
                                }
                                // Merge
                                else if (list_dictionary.leftside_sibling != null &&
                                                list_dictionary.leftside_sibling.parent_node == list_dictionary.parent_node &&
                                                list_dictionary.leftside_sibling.checkIfMergeable())
                                                {
                                        sibling_node = list_dictionary.leftside_sibling;
                                        int index_of_pointer = findIndexOfPointer(parent_node.pointers_for_child_nodes, list_dictionary);
                                        // Remove key and pointer from parent
                                        parent_node.removingTheKey(index_of_pointer - 1);
                                        parent_node.removingThePointer(list_dictionary);
                                        // Update sibling
                                        sibling_node.rightside_ibling = list_dictionary.rightside_ibling;
                                        // Check for deficiency
                                        if (parent_node.checkIfDeficient())
                                        {
                                                deficiencyHandling(parent_node);
                                        }
                                }
                                else if (list_dictionary.rightside_ibling != null &&
                                                list_dictionary.rightside_ibling.parent_node == list_dictionary.parent_node &&
                                                list_dictionary.rightside_ibling.checkIfMergeable())
                                                {
                                        sibling_node = list_dictionary.rightside_ibling;
                                        int index_of_pointer = findIndexOfPointer(parent_node.pointers_for_child_nodes, list_dictionary);
                                        // Remove key from parent
                                        parent_node.removingTheKey(index_of_pointer);
                                        parent_node.removingThePointer(index_of_pointer);
                                        // Update sibling
                                        sibling_node.leftside_sibling = list_dictionary.leftside_sibling;
                                        if (sibling_node.leftside_sibling == null)
                                        {
                                                first_leaf_node = sibling_node;
                                        }
                                        if (parent_node.checkIfDeficient())
                                        {
                                                deficiencyHandling(parent_node);
                                        }
                                }

                        }
                        else if (this.root_node == null && this.first_leaf_node.number_of_pairs == 0)
                        {
                                /* When the only pair within the tree is deleted */
                                // Set first leaf null to indicate tree is empty
                                this.first_leaf_node = null;
                        }
                        else
                        {
                                sortDictionaryList(list_dictionary.dictionary_list);
                        }
                }
        }
}
```

SEARCHING (SINGLE KEY):

From the set of key-value pairs that exist within the B+ tree, this method returns the value associated with the given key. The search operation is not performed if the tree is empty or the key given is invalid. If the key is true, we perform a binary search and return the value that corresponds to it.

```
public Double search(int key)
        {
                // If empty return null
                if (checkIfEmpty())
                {
                        return null;
                }
                // Find leaf node for the key
                LeafNodeClass list_dictionary = (this.root_node == null) ? this.first_leaf_node : searchForLeafNode(key);
                // Perform binary search to find index
                DictionaryPairClass[] sorted_dictionary_pairs = list_dictionary.dictionary_list;
                int index = binarySearch(sorted_dictionary_pairs, list_dictionary.number_of_pairs, key);
                // If index negative return null
                if (index < 0)
                {
```

```
                return null;
            }
            else
            {
                return sorted_dictionary_pairs[index].value;
            }
        }
```

SEARCHING (RANGE OF KEYS):

This method traverses the B+ tree, whose nodes are stored as a doubly connected array, and returns all the values in a list whose corresponding keys are within the lower bound key and upper bound key ranges. When we come across a null value, we know that the doubly connected array of tree nodes cannot be traversed any further.

```
public ArrayList<Double> search(int lower_bound_range, int upper_bound_range)
        {
            ArrayList<Double> values = new ArrayList<Double>();
            LeafNodeClass current_node = this.first_leaf_node;
            while (current_node != null)
            {
                DictionaryPairClass sorted_dictionary_pairs[] = current_node.dictionary_list;
                for (DictionaryPairClass dict_pair : sorted_dictionary_pairs)
                {
                    /* Stop searching if null value is encountered */
                    if (dict_pair == null) { break; }
                    // Include value if within the range
                    if (lower_bound_range <= dict_pair.key && dict_pair.key <= upper_bound_range)
                    {
                        values.add(dict_pair.value);
                    }
                }
                /* Update current node to right sibling node and leaf traversal is from left to right */
                current_node = current_node.rightside_ibling;
            }
            return values;
        }
```

main():

Based on the lines read from the input files <input.txt>, this feature calls all the functions (operations that can be performed in a B+ tree). It reads line by line from the input file, executes all of the required operations, and then generates an output file to which the program will write the results.

- Initialize (order)
- insert(key, value)
- delete(key)
- Search(key)
- Search(key1,key2)

```
public static void main(String[] args)
        {
            if (args.length != 1)
            {
                System.err.println("Provide a input file");
                System.exit(-1);
            }
            String name_of_file = args[0];
            try
            {
                // Read input file
                File file_pointer = new File(System.getProperty("user.dir") + "/" + name_of_file);
                Scanner input_scanner = new Scanner(file_pointer);
                // Create output file to store results
                FileWriter generate_output = new FileWriter("output_file.txt", false);
                boolean line_first = true;
                bplustree bplustree_object = null;
                while (input_scanner.hasNextLine())
                {
                    String current_line = input_scanner.nextLine().replace(" ", "");
                    String[] get_tokens = current_line.split("[(,)]");
                    switch (get_tokens[0])
                    {
                        // Initialize
                        case "Initialize":
                            bplustree_object = new bplustree(Integer.parseInt(get_tokens[1]));
```

```
                                        break;
                                // Insert into the tree
                                case "Insert":
                                        bplustree_object.insert(Integer.parseInt(get_tokens[1]), Double.parseDouble(get_tokens[2]));
                                        break;
                                // Delete from the tree
                                case "Delete":
                                        bplustree_object.delete(Integer.parseInt(get_tokens[1]));
                                        break;
                                case "Search":
                                        String search_result = "";
                                        // Perform search across a range
                                        if (get_tokens.length == 3)
                                        {
                                                ArrayList<Double> values = bplustree_object.search(Integer.parseInt(get_tokens[1]),
Integer.parseInt(get_tokens[2]));
                                                if (values.size() != 0)
                                                {
                                                        for (double v : values)
                                                        {
                                                                search_result += v + ", ";
                                                        }
                                                        search_result = search_result.substring(0, search_result.length() - 2);
                                                }
                                                else
                                                {
                                                        search_result = "Null";
                                                }
                                        }
                                        // Perform search operation
                                        else
                                        {
                                                Double value = bplustree_object.search(Integer.parseInt(get_tokens[1]));
                                                search_result = (value == null) ? "Null" : Double.toString(value);
                                        }
                                        // Write search result to file
                                        if (line_first)
                                        {
                                                generate_output.write(search_result);
                                                line_first = false;
                                        }
                                        else
                                        {
                                                generate_output.write("\n" + search_result);
                                        }
                                        generate_output.flush();
                                        break;
                                default:
                                        throw new IllegalArgumentException("\"" + get_tokens[0] + "\"" + " is an unacceptable input.");
                                }
                        }
                        // Close file pointer
                        generate_output.close();
                }
                catch (IllegalArgumentException e)
                {
                        System.err.println(e);
                }
                catch (FileNotFoundException e)
                {
                        System.err.println(e);
                }
                catch (IOException e)
                {
                        System.err.println(e);
                }

        }
```

OTHER HELPER FUNCTIONS:

searchForLeafNode():

This approach begins at the top of the B+ tree and works its way down using key comparisons to the leaf node that has 'key' in its dictionary.

binarySearch():

If a dictionary pair with target key t is found, this approach performs a regular binary search on sorted dictionary pairs and returns the index of the dictionary pair with target key t. Otherwise, a negative value is returned by this process.

findCenterPoint():

This is a basic method that returns the B+ tree's max degree m's midpoint (or lower limit, depending on the sense of the method invocation).

findIndexOfPointer():

This method returns the index of the pointer that points to the given node object from a list of pointers to Node objects.

deficiencyHandling():

This procedure, given a deficient node, corrects the problem by borrowing and merging.

linearSearchForNull():

This approach returns the index of the first null entry found after performing a regular linear search on sorted dictionary pairs. Otherwise, a -1 is returned. When the target t = 0, this procedure is usually used instead of binarySearch().

checkIfEmpty():

This is an easy approach for evaluating whether or not the B+ tree is empty.

shiftPointersDown():

This procedure is used to shift down a list of pointers that have null values prepended to them.

sortDictionaryList():

This is a specialized sorting process for dictionary lists of null values strewn throughout.

splitDictionaryList():

This approach divides a single dictionary into two dictionaries of equal length, but each of the resulting dictionaries contains half of the non-null values from the original dictionary. This strategy is most often used to break a node in the B+ tree.

dividingInternalNode():

This approach is called when an entry into the B+ tree results in an overfull node, and it splits the overfull node.

checkIfDeficient():

This basic approach decides whether or not a node is defective by comparing the actual degree of children to the allowed minimum.

checkIfLendable():

This basic method checks if a node's current degree is greater than the prescribed minimum to see whether it can lend one of its dictionary pairs to a deficient node.

checkIfMergeable():

This basic method tests if a node can be merged by seeing whether it has the appropriate number of children.

dividingChildPointers():

After the defined break, this method modifies the node by deleting all pointers within the child pointers. The method returns the deleted pointers in their array, which can be used to create a new sibling node.