CS3310 Data and File Structures with Java

Instructor: Ajay K Gupta

Lab TA: Yu Guo, Zijiang J Yang

Assignment: 5

Jonah Kubath

**Software Life Cycle Report – Assignment 5**

**Phase 1: Specification**

* Practice building B trees
* Practice searching B trees
* Develop high-performance solutions

Program Output is at the end of this file.

**Phase 2 Design:**

**Modules and Basic structure**

Module 1: Class Main

Fields:

1. Final int ASCII\_LENGTH – number of ascii values to generate
2. Characters characters[] – holds the characters
3. Int decimal[] – holds the decimals
4. String octal[] – holds the octal values as strings
5. String hex[] – holds the hex values as strings
6. Tree23 – a tree is made for each value
7. Tree234 – a tree is made for each value
8. Long total[] – holds the total time for 2-3 and 2-3-4 trees

Methods:

1. Int getSearchValue() – receives the search value from the user
2. Long[] searchBothTrees(8 binary trees) – main search manager for searching the trees for the given value from the user
3. Long[] Search(4 b trees) – perform the actual search of the b trees
4. Long[] search(4 b trees) – perform the actual search for the 2-3 trees
5. Integer[] getValue(String) – converts the given value to searchable values
6. writeFiles(8 binary trees) – writes the level traversal to files
7. writeAverage(long[]) – writes the average search time to the console

Module 2: LeafNode

Fields:

1. private Key ldata – only used on actual leaf nodes
2. private Key lkey – largest value in the left subtree
3. private Key rkey – largest value in the middle subtree
4. private LeafNode<Key> parent – pointer to the parent node
5. private LeafNode<Key>lNode – pointer to the left node
6. private LeafNode<Key>mNode – pointer to the middle node
7. private LeafNode<Key>rNode – pointer to the right node
8. boolean isLeaf – if the node is being inserted as a leaf or internal node

Methods:

1. Standard getters and setters for the fields

Module 3: LeafNode234

Fields:

1. Private Key ldata – only used in leaf nodes
2. Private key lkey – largest value in left subtree
3. Private key mkey – largest value in the middle left subtree
4. Private key rkey – largest value in the middle right subtree
5. Private LeafNode234<Key> parent – pointer to the parent node
6. Private LeafNode234<Key> lNode – pointer to the left subtree
7. Private LeafNode234<Key> M0Node – pointer to the middle left subtree
8. Private LeafNode234<Key> mNode – pointer to the middle right subtree
9. Private LeafNode234<Key> rNode – pointer to the right subtree
10. Boolean isLeaf – if the node is being inserted as a leaf or internal node

Methods:

1. Standard getters and setters for the fields

Module 4: NodeQueue23

Fields:

1. NodeQueue23<Key> next – pointer to the next item in the list
2. LeafNode<Key> data – B Tree node held

Methods

1. Standard getters and setters for the fields

Module 5: NodeQueue234

Fields:

1. Private NodeQueue234<Key> next – pointer to the next item in the list
2. Private LeafNode234<Key> data – B Tree node held

Methods:

1. Standard getters and setters for the fields

Module 6: Queue23

Fields:

1. Private NodeQueue next – the next Node in the queue
2. Private Node data – the data held

Methods:

1. Getters and setters for the fields

Module 7: Queue23

Fields:

1. Private NodeQueue23<Key> head – the head node to the queue
2. Private NodeQueue23<Key> tail – the tail node to the queue

Methods:

1. Getters and setters for the fields
2. Void add(LeafNode<Key> newNode) – adds the new Node to the end of the queue
3. NodeQueue23<Key> delete() – deletes the head node of the queue

Module 8: Queue234

Fields:

1. Private NodeQueue234<Key> head – the head node to the queue
2. Private NodeQueue234<Key> tail – the tail node to the queue

Methods:

1. Getters and setters for the fields
2. Void add(LeafNode234<Key> newNode) – adds the new Node to the end of the queue
3. NodeQueue234<Key> delete() – deletes the head node of the queue

Modele 9: Tree23

Fields:

1. LeafNode<Key> root – root of the tree

Methods:

1. Void insert(LeafNode<Key> current, LeafNode<Key> newNode) – insert a new node into the tree – either leaf node or internal node
2. LeafNode<Key> search(LeafNode<Key> current, LeafNode<Key> newNode) – finds the parent node to add the new node to
3. Int addNode(LeafNode<Key> parent, LeafNode<Key> newNode) – attempts to add the new node to the parent node. 0 is return on failure
4. Int add(LeafNode<Key> current, LeafNode<Key> newNode) – attemps to add the new leaf node to the parent node. 0 is returned on failure
5. Void split(LeafNode<Key parent, LeafNode<Key> newNode) – takes the newNode and adds it to the newly created, split node
6. Void makeNewRoot(LeafNode<Key> left, LeafNode<Key> right) – used to split the root node
7. Void printInOrder(LeafNode<Key> current) – prints the tree with an in-order traversal
8. Key search(LeafNode<Key> current, Key value) – searches the tree for the given value – null is returned if the value is not found
9. Void traversal(FileWriter file) – performs a level traversal and writes the data to the given file

Module 10 – Tree234

Same information as Tree23

1. Method – void setKeys(LeafNode<Key> parent) – sets the keys of the given parent node

**Pseudocode**

**Class - Main**

Main()

Make a variable to determine the number of ascii values to generate

Make 4 arrays to hold the generated ascii values

Run a for loop to generate the ascii values

Use Integer.toOctalString and Integer.toHexString to convert the decimal values to octal and hex string representations

Save the values to the appropriate arrays

Make 4 variables to hold the 4 2-3 Binary Trees

Run a for loop to insert the given ascii values to the appropriate B Tree

Call the B Tree object.insert() method

Make 4 variables to hold the 4 2-3-4 Binary Trees

Run a for loop to insert the given ascii values to the appropriate B Tree

Call the B Tree object.insert() method

Write the level traversal to files with writeFiles()

Allow the user to search for values with searchBothTrees()

Print the average serach time with writeAverage()

getSearchValue()

Make a variable to hold the user input

Make a boolean to determine if good input was given

Use a while loop to continue to prompt for data until good data in given

Use a try catch to catch an error thrown

Use the global scanner object to retrieve input from the user

Throw an error if no input is given

Catch the error

Tell the user that no input was given and re prompt for input

searchBothTrees()

Make a variable and call the getSearchValue() to get the value the user wants to search for

Make an array to hold the total search time and number of searches for both data structures

Run a while loop until the user inputs quit

Create a temporary array to hold the runtime and count for the individual B Tree

Search the B tree by calling search23()

Add the temp array to the total time and total count

Search the 2-3-4 tree by calling search()

Add the temp array to the total time and total count for the 2-3-4 tree

Prompt the user for more input by calling getSearchValue()

Search()

Initialize variables to hold the converted input data to Decimal, octal, hex, and char

Initialize variables to hold the total run time

Search for the decimal value of the input

Call search on hex, octal, and char to get the information for the given decimal

Search for the hex value

Call search on decimal, octal, and char to get the information for the given hex

Search for the octal value

Call search on the octal, hex, and char to get the information for the given octal

Search for the char value

Call search on the decimal, octal, and hex to get the information for the given char

Return the total time spent searching and the total searches

Search234()

Same as search(), but performed on the 2-3-4 B Tree

getValues()

Converts the given string input to the needed decimal, octal, hex, and char

Null is set if the string cannot be converted to the given value

If statements are used to catch values of chars that cannot be typed

writeFiles()

Create 8 files to write to

Create 8 WriteFile objects to write to the given files

Call the traversal() method for each Tree to print the level traversals to the files

Close the files

writeAverage()

Write the average search time to the console for the given 2-3 and 2-3-4 Tree

**Class – Queue23 and Queue234**

Standard getters and setters

Add()

Create a new node with the given data

Add the new node to the tail of the queue

Delete()

Delete a node from the queue and return it

**Class – Tree23**

Insert()

If the node being inserted is a leaf node

If the root is null

Create a new node and set it as root

If root is a leaf node

Create a new node and add the two nodes to it

Set the new node as the root

IF the root is not a leaf node

Call search() to find the parent node to add the new node to

Call add() to add the new node to the parent

If the parent doesn’t have space for the new node

Call split()

If the new node is not a leaf node

Call addNode() to insert the new tree into the main tree

If error is returned

Call split to make room

Search()

Base case – we are at the leaf nodes, return the parent of the leaf node

Otherwise move through the B Tree accordingly

If the new node is less than the left key – move left

If the new node is less than the right key – move to the middle

Else move to the right

addNode() – used for adding a subtree

if either of the middle or right nodes are null, then we have a spot to add the new node

determine which spot the new node should be in and move the keys and node pointers accordingly

else

return an error so split() can be called

add()

if either of the middle or right nodes are null, then we have a spot to add the new node

Determine the spot to add the new node and move the keys and node pointers accordingly

Else

Return an error so split() can be called

Split()

Split the left node into 2 nodes

Create a new node

Add the left node’s right child to the left of the new node

Add the node to be inserted into the middle of the new node

Set the keys accordingly

Set the right subchild of the left node to null as the child was moved

If the left node was the root node, then the new node cannot be added to its parent

Call makeNewRoot()

Else

Re insert the new node into the parent of the left node

makeNewRoot()

Take in the left and right subtrees

Make a new root node

Set the left child to be the left subtree and middle child to be the right subtree

Set the keys accordingly

Set root as the new root

Search()

Base case – current node is null, return null

Base case – current value is a leaf node

If it equals the searched value then return the value

Else return null

Otherwise move down the tree

If value is less the left key, move left

If the value is less than the right key, move to the middle

Else move to the right

Traversal()

Create a queue object to add the nodes to

Call list.add(root) to add all the nodes to the queue

Use a while loop that stops when there are no more nodes in the queue

Remove an object from the queue – print its data

**Class – Tree234**

Methods are the same as Tree23, but the middle keys must be checked when traversing

When nodes are split, the left node will have 3 children and the new right node will have 2 children

**Class: LeafNode, LeafNode234, NodeQueue23, NodeQueue234**

These classes only has standard getters and setters for the fields

**Summary Analysis**

The binary trees allow for quick searching on many records. The 2-3 Tree can have up to log base 3 height with a minimum of log base 2 height. The 2-3-4 Tree can have as many as log 2 height, if all nodes had 2 children, or as few as log base 4 height. For my implementation, I used 128 ascii values as the searchable range. This allows for 5-7 as the height for the 2-3 tree and 4-7 as the height for the 2-3-4 tree. The number of leaves for both the trees will be the same as the leaves are the records being added.

One of the main differences in the 2-3 tree and the 2-3-4 tree is the requirement for an addition key in the 2-3-4 tree. This means that when adding a new node more values must be checked and eventually moved to keep the sorting the tree offers. It also means that when a node is split that either the left node can have 3 children and the right 2 or the the left can have 2 and the right 3. This would be determined on how the data would be added to the tree, either close to in-order or reverse-order.

**Time Complexity**

Main

Main – adds the nodes to the tree O(n)

Tree

insert – O(h)

search – O(h)

addNode – O(1)

add – O(1)

split – O(1)

makeNewRoot – O(1)

search for value – O(h)

traversal – O(n)

Time complexity is difficult to measure with such a small table as the ascii table. The searching is performed very quickly as the trees only have a few levels the pass through. This brings variability into the consistency of the computer running the application. I kept the computer as consistent as possible for the search to have the best outcome. Over a large data set, the 2-3-4 tree would start to show the signs of faster searching as it would have less levels.

**Phase 3: Risk Analysis**

The traversal files are overwritten each time the program is run. This may cause data loss of previous runs if the user is not careful.

**Phase 4: Verification**

I searched for multiple values many times.

**Phase 5: Coding**

The code of this program is included in the zip file. The code is explained with comments and line breaks to make reading easier. A Javadoc is also made to explain the use of each method.

**Phase 6: Testing**

Tree traversals were output as files.

**Phase 7: Refining the program**

Refinements can be made after the client has used the application

**Phase 8: Production**

A zip file including the source files from eclipse and the output of the program have been included.

**Phase 9: Maintenance**

Changes can be made once feedback is received.