CS3310 Data and File Structures with Java
Instructor: Ajay K Gupta
Lab TA: Yu Guo, Zijiang J Yang
Assignment: 4

Jonah Kubath

Software Life Cycle Report – Assignment 4

Phase 1: Specification

- Practice building trees and heaps
- Practice traversing trees and heaps
- 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. Int timesToRun – controls the average time
2. ArrayList<Student> - holds the original file data
3. Node minHeap – head node to the min heap
4. BST binarySearch – Object that holds the binary search tree
5. MaxHeap heap – object that holds the max heap
6. Long start, end, total – holds the run time of each search
7. Long[] totalArr – holds the cumulative run time of each search
8. Int index – used to copy data from the original to new data structures
9. String first, last – the first and last name to search for
10. Queue queue – used for breadth first traversal
11. Stack stack – used for depth first traversal
12. Node found – holds the data if the node is found in the search

Methods:

1. Void printNodeData(Node data) – prints the wanted information for the found node – parent, level, position in level, left / right child
2. String getSearchName() – used if the user wants to input a different name other than the default
3. Void printAverage(long[] array, int timesRun) – prints the average run time for the searches

Module 2: BinaryTree
Fields: none
Methods:

1. Void insert(Node head, Node insert) – inserts the insert Node using the head Node as the root
2. Void increment(Node leaf) – starts at the given leaf node and increments the size of each node above it until the root is found
3. Void decrement(Node leaf) – starts at the given leaf node and decrements the size of each node above it until the root is found
4. Void minHeap(Node leaf) – heapify for min heaps
5. Void minHeapTopDown(Node head) – start at the root node and heapify from the root node down to the leaves
6. Void printPreOrder(Node head) – prints the tree in pre-order starting at the given head node
7. Void printPostOrder(Node head) – prints the tree in post-order starting at the given head node
8. Void printInOrder(Node head) – prints the tree in in-order starting at the given head node
9. Student delete(Node head) – deletes the given node from the tree and re-arranges the children nodes. The deleted data is returned
10. Void enqueue(Node head, Queue insert) – inserts the tree nodes to the given queue starting at the head node as root
11. Void enqueue(MaxHeap maxHeap, Queue insert, int start) – overloaded method to insert the nodes into the queue using the maxHeap.heap array
12. Void pushStack(Node head, Stack stack) – pushes the tree nodes to the given stack starting at the head node as root
13. Node breadthFirst(Queue queue, String last, String first) – performs a breadth first search using the queue comparing data to the first and last name
14. Node depthFirst(Stack stack, String last, String first) – performs a depth first search using the stack comparing data to the first and last name
15. Void setPosition(Node head, HashMap<Integer, Integer> map) – sets the position of the nodes in the tree using the HashMap

Module 3: BST
Fields:

1. Private Node head – head of the BST tree
2. Private HashMap<Integer, Integer> map – holds the positions of the nodes

Methods:

1. Void insert(Node head, Student insert, int level) – inserts the new Student data into the tree using the head Node as the root. Level is passed recursively to set the level of the node
2. Student delete(Node head, String name) – deletes the node with the given name and returns the Student object that was deleted
3. Node getHead() – returns the head of the tree
4. Node setHead(Node head) – sets the head node of the tree
5. Void setPosition(Node head) – traverses the tree and sets the position of each node in the level that it is in starting at the head node as the root
6. Node searchLast(Node head, String last, String first) – searches the tree for the given last and first name

Module 4: MaxHeap

Fields:

1. Node[] heap – holds a max heap using an array
2. Int last – keeps track of the last node index in the array

Methods

1. MaxHeap(ArrayList<Student> list) – initializes the array using the size of the array list. Sets the last index to 0
2. Void insert(Student record) – inserts the new student data as a node into the array
3. Void maxHeapify(int last) – heapify the last node data
4. Void printInOrder(int print) – prints the nodes of the array in in-order of their tree traversal
5. Void printPreOrder(int print) – prints the nodes of the array in pre-order of their tree traversal
6. Void printPostOrder(int print) – prints the nodes of the array in post-order of their tree traversal
7. Node delete() – deletes that last node in the array and returns it
8. Void heapify(int index) – heapify starting at the given node

Module 5: Node

Fields:

1. Private Student data – the first and last name of the student
2. Private int size – the size of the sub trees + 1
3. Private Node parent – Node pointer to the parent node
4. Private Node leftChild – Node pointer to the left child
5. Private Node rightChild – Node pointer to the right child

6. Int level – holds the level the node was inserted at
7. Int position – holds the position in the level relative to siblings

Methods:

1. Node(String head) – makes the node the root not by setting parent to null and size to 1
2. Getters and Setters for all the fields

Module 6: NodeQueue

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: NodeStack

Fields:

1. Node data – the data held
2. NodeStack next – the next node in the stack

Methods:

1. Getters and setters for the fields

Module 8: Queue

Fields:

1. Private NodeQueue head – the head node to the queue
2. Private NodeQueue tail – the tail node to the queue

Methods:

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

Module 9: ReadFile

Fields:

1. Static Scanner scan – Used to read data
2. Static String fileName – the file to read the data in from

Methods:

1. Void readFile(Node headNode, ArrayList<Student> list) – reads from the filename field and saves the data to the headNode tree and also the list Array list
2. Void getFile() – asks the user for a new data file if the default file is not found

Module 10: Stack

Fields:

1. Private NodeStack head – the head node of the stack
2. Private NodeStack tail – the tail node of the stack

Methods:

1. Void insert(Node newNode) – inserts the new node to the head of the stack
2. NodeStack delete() – deletes the head node of the stack and returns it
3. Getters and setters for the fields

Module 11: Student

Fields:

1. Private String last – The last name of the student
2. Private String first – The first name of the student

Methods:

1. Getters and setters for the fields

Pseudocode

Class - Main

Main()

Make a variable to hold the number of times the program is run
Create an array list to hold the original data
Create a Node minHeap and a BST binarySearch object
Create 3 variables to hold the time data – start, end, total
Create an array of longs to hold the cumulative time for the searches
Call ReadFile.getFile() to set the data file
Call ReadFile.readFile() to read the data in and set the array list and minHeap data
Iterate the array list and set the Max heap data
Call binarySearch.setPosition to set the position of the nodes relative to their siblings
Call BinaryTree.setPosition to set the position of the minHeap relative to their siblings
Set the first and last name to search for – there is a method if this should be dynamic
Name = getSearchName()
Run a for loop for the designated number of runs
Enqueue the data from the MaxHeap
Call BinaryTree.breadthFirst to search the MaxHeap using breadthFirst search
Print the data if it is the first time the program was run
Add the total time to the array
Enqueue the minHeap
Call BinaryTree.breadthFirst to search the MinHeap using breadthFirst search
Add the total time to the array

Push the MaxHeap nodes to the stack
Call BinaryTree.depthFirst() to search the MaxHeap using depth search
Add the total time to the array
Push the MinHeap to the stack
Call BinaryTree.depthFirst() to search the MinHeap using depth search
Add the total time to the array
Search the binary search tree – binarySearch.searchLast()
Add the total time to the array

After the number of runs are done – print the average time
printAverage()

printNodeData()
print the data from the given node

getSearchName
Initialize a scanner for System.in reading
use a try / catch to ask the user for a name to search the trees for
return the String from the user

Class – BinaryTree

Insert()
Create a temp node to iterate the list until a child is null
Use a while loop to traverse the tree until a child is null
If the left child is null – set the new node to the left child's next
If the right child is null – set the new node to the right child's next
Call increment() to add 1 to the size of all the parent nodes
Call minHeap() to move the data and maintain a MinHeap

Increment()
While the current node.parent != null
Add one to the size

Decrement()
While the current node.parent != null
Subtract one from the size

minHeap()
create a temp node to traverse the tree
while the temp node's parent is not null and the data is less than the parent data
swap the two student objects

minHeapTopDown()

Create a temp node
Create a temp student object to hold the head data
If the head node's children are both null, then it is a leaf node
    Return
Iterate the tree until the head node's children are null

printPreOrder()

traverse the list
visit the node
if != null call printPreOrder() on the node.leftChild
if != null call printPreOrder() on the node.rightChild

printPostOrder()

traverse the list
if != null call printPreOrder() on the node.leftChild
if != null call printPreOrder() on the node.rightChild
visit the node

printInOrder()

traverse the list
if != null call printPreOrder() on the node.leftChild
visit the node
if != null call printPreOrder() on the node.rightChild

delete()

Create a temp node
Create a temp student object
Iterate the list until either of the children are null
    Iterate the tree checking the size of each sub tree
Swap this node with the root node
Delete the new leaf node
Heapify from the top down by calling minHeapTopDown on the new root
Return the deleted leaf node data

Enqueue() - tree

If head == null then the tree is empty or a leaf node is found
    Return
Else
    Insert the head node to the queue
    Recursively call enqueuer on the left and right child

```
Enqueue() – array
        Add the node data to the queue
        Recursively call the enqueue() until the index is out of bounds for the array

pushStack() – tree
        if the head == null, then the tree is empty or a leaf node is found
        else
                insert the head node to the stack
                recursively call the pushStack with the left and right child

pushStack() – array
        run a for loop to push every node in the array to the stack

breadthFirst()
        if the queue.getHead() == null, then the queue is empty
                return
        else
                test the head data with the first and last name, if equal return the data
                else
                        delete the head and move to the next node in the queue

depthFirst()
        if the stack.getHead() == null, then the stack is empty
                return
        else
                test the head data with the first and last name, if equal return the data
                else
                        delete the head node and move to the next node in the stack

setPosition()
        if the current node has both children as null
                if the hash map doesn't have a key for the current level
                        set the level -> position as 1
                        set the current node position to 1
                        iterate the position
                else
                        get the position and set the current node to it
                        iterate the position
        if the left child is not null
                call setPosition on the left child node
        if the right child is not null
                call setPosition on the right child node

        set the position of the current node
```

if the map contains the key and position
    get the position and set the current node to it
    iterate the position
else
    create the key and position
    set the current node position to 1
    iterate the position

Class: BST

Insert()
    If the head node of the BST is null, then the tree is empty
        Set the head node data to the input data
    Else
        Add one to the current node
        Compare the insert data to the left child
        If the insert data is >
            Move to the right
                If the data is null, create node and set the node to the right child
                Else
                    Recursively call insert() on the right child
        If the insert data is <
            Move to the left
                If the data is null, create node and set the node to the left child
                Else
                    Recursively call insert() on the left child

Delete()
    If the head data is null, then the tree is empty
        Return
    If the data to delete is less than the current node, move to the right
        Call delete() on the left child node
    Else
        Call delete() on the right child node
    If it is not less than or greater than, then the node to delete has been found
        Check if both children of the parent are null
        Create a temp student object to hold the data
        Get the data
        Set the parent pointer to the child to null
        Call BinaryTree.decrement() on the current node to decrease the size by one

        Else if the node to be deleted has only 1 child
            Find which node of the parent is the node to be deleted
            Set the parent’s child node to the child node of the nod to be deleted

Else the node to be deleted has two children
Swap the deleted node data with a child node until it becomes a leaf node
Call BinaryTree.decrement() to decrease the size of all the parent nodes by one
Delete the leaf node by setting the parent node's child pointer to null

Return the deleted data

Class: MaxHeap

Insert()
Create a new node to add to the array
Save the given student data to the new node
Find where the parent should be in the array
Make sure the parent is in the bounds of the array
If the node should be set to the left child of the parent – set it
If the node should be set to the right child of the parent – set it
If the parent is out of bounds of the array
Set the parent to null
Calculate the level of the node by dividing the current index position by 2 until 0 is found
Set the position in the level by subtracting the current index from the total of the inner nodes (2 ^ level – 1)
Call maxHeapify() to move the student data until the parent node is greater

maxHeapify()
hold the parent index by taking the current (index – 1) / 2
if the parent is < 0, the array will be out of bounds
return;
else compare the parent node data to the current node data
if the parent is less than the current data – swap the data
if the parent is greater than the current data
return

call maxHeapfiy() recusively until pass the array bounds

printInOrder()
in-order traversal of the array

printPreOrder()
pre-order traversal of the array

printPostOrder()
post-order traversal of the array

delete() – delete the head node

    create a temp node to hold the node to be deleted
    swap the head node with the last node
    delete by new last node by setting it to null
    call heapfiy on the first node to keep the MaxHeap ordr
    return the temp node

heapify()

    create a temp node
    create an integer to hold the index of the left child of the current node
    if the left child index is past the array length
        return as the array will be out of bounds
    compare the current node to the left child
        if the left child is greater than current node – swap
        call heapify() on the left child index
    compare the current node to the right child
        if the right child is greater than the current node – swap
        call heapfiy() on the right child index

Class: Node, NodeQueue, NodeStack

These classes only have standard getters and setters

Class: Stack

Insert()

    Create a new node to the push to the stack with the Node data set
    If the head of the stack is null – the stack is empty
        Set the head and tail of the stack to the new node

Delete()

    If the head node is null
        Return as the stack is empty
    Else
        Create a temp NodeStack object to hold the head node
        Set the head of the stack to Head.getNext()
        Return the temp NodeStack object

Standard getters and setters for the fields

Class: ReadFile

readFile()

    set a counter to keep track of the current level and position of the trees
    if the file has data to read – read the first line
        create a new student object to hold the data

set the first and last name of the new student by scan.nextLine()
add the student object to the array list to hold the original data
add the student object to the minHeap
set the minHeap size to 1
set the minHeap level to 0
set the minHeap position to 1

increment the position
use a while to read through the remaining data – scan.hasNext()
if the position is greater that what the current level can hold
reset the position to 0
increment the levels
increment the position
Create a new student object to hold the data
Read and set the student first and last name – scan.next()
Read the new line character with a blank scan.nextLine()
Create a new node to insert into the tree
Set the data to the created student object
Add the node to the tree with BinaryTree.insert()

Close the scanner

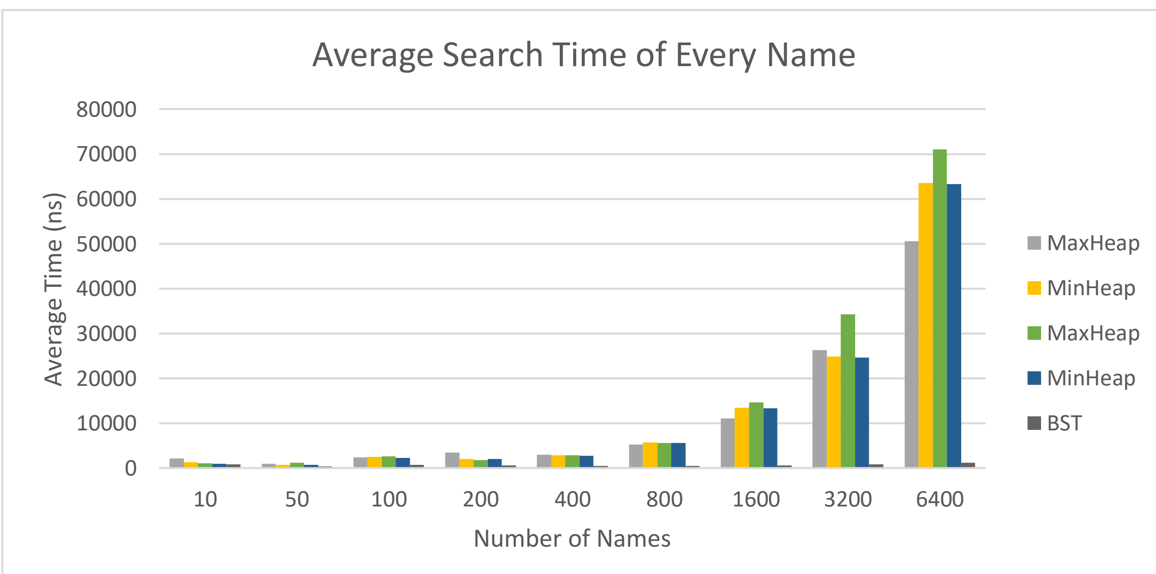
getFile()
create a boolean to determine if the file is found
use a while loop to prompt the user for an input file until the file is found
use a try / catch to catch errors from opening a file that does not exist
try {
Create a file object with the default file name – namelist.txt
Create a scanner object with the opened file
Set the boolean to true because if we are this far an error has not be thrown
}
catch (FileNotFoundException e) {
Print to the user that the file was not found and they need to enter a new file name
Change the scanner to read from the System.in
Set the filename string to scan.nextLine()
Trim the filename for any leading white spaces
Set the goodFile to false to run the test again
}

Class: ReadFile
This class only has standard getters and setters for the fields

Summary Analysis

The binary search tree performed well in the actual tests. As the number of names double for each the average search time stayed very low. The graphs of the binary search tree look nearly linear. The binary search tree had a high performance because, unlike the heaps, there is a definite decision of whether to move left or right by the comparison of the data. The issue that the binary search tree could have is if the data would come in a sorted order. This would cause the binary search tree to be a simple linked list and the searching would be linear. Assuming this doesn't happen, the binary search tree can cut off approximately half the data with the first comparison and so on through the tree.

The heaps did not perform very well with the data and name changes. The issue that the heaps have is that we cannot guarantee what the siblings of a node look like other than that they are either less than or greater than the parent depending on if a max or min heap is implemented. This causes us to search through the nodes on the level before moving to the next level. This issue can be somewhat eliminated by improving the enqueue process. If a max heap is implemented and the current node is less than the searched name, we are guaranteed by the max heap that there are only going to lesser names below it. This would allow us to not enqueue those nodes for searching.



Time Complexity

BinaryTree.depthFirst() – iterate through the stack O(n)
BinaryTree.breadthFirst() – iterate through the queue O(n)
BinaryTree.setPosition() – traverse the tree and set the position of the node O(n)
BinaryTree.pushStack() – push the item on the stack O(n)
BinaryTree.enqueue() – add the head to the queue O(n)
Print the binary tree – O(n)
BinaryTree.minHeapTopDown() – heapfiy from the root to the leaves O(h)
BinaryTree.minHeap() – start and leaf and swap until parent is lesser O(h)
BinarySearch.decrement() – traverse from leaf node to root O(h)
BinarySearch.increment() – traverse from leaf node to root O(h)
BinarySearch.insert() – traverse until a leaf node is found O(h)
BST.searchLast() – compare and move left or right O(h) *H could be linear*
BST.setPosition – set the position of the node relative to siblings O(n)
BST.insert() – navigate to a leaf, insert O(h) *H could be linear in worst case*
MaxHeap.heapfiy() – heapify from top down O(h)
Print max heap – O(n)
MaxHeap.maxHeapfiy() – compare to the parent, swap if greater O(h)
MaxHeap.insert() – insert at the end of the array O(1)
ReadFile.readFile() – read every line of the file O(n)
Node, NodeQueue, NodeStack, and Student Class – only getters and setters O(1)

Phase 3: Risk Analysis

The only risk for this application is having a file that is not found. If this happens, the user is prompted for a new file.

Phase 4: Verification

The application will search every name and output the average time to a file

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

Test runs have been included at the end of this file.

Phase 7: Refining the program

The tree structure of the binary search tree could be improved by finding an average name to start the root with. This will allow the data to more likely be split in half.

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.

Program Output