CS 401 project

Problem Specification

You must construct a list (from a data file or from user input) and then implement two types of sorting algorithms: simple (such as selection sort, insertion sort, or bubble sort) and sophisticated (such as quick sort, merge sort, or heap sort). The purpose is to quantify the number of comparisons made by each algorithm in order to identify which one is more efficient, while also matching these results with theoretical assumptions based on their Big-O notation complexity.

Similarly, when it comes to searching algorithms, you should compare linear search with binary search tree (BST) and hashing. The efficiency of various searching algorithms will be assessed based on their complexity in comparison to linear, binary, and hash function searching.

Software specification

BSTNode.java: This file defines a class BSTNode for a node in a Binary Search Tree (BST). It includes the following functions:

Constructor to initialize a node with an item.

Getters and setters for the item and the left and right child nodes.

contains method to check if a certain item is present in the subtree rooted at this node.

DLLNode.java: This file likely defines a class for a node in a Doubly Linked List (DLL). Common functions in such a class include:

Constructor for initializing the node with an item.

Getters and setters for the item, and the previous and next nodes in the list.

DLLToBSTConverter.java: This file is expected to contain methods for converting a Doubly Linked List into a Binary Search Tree. Key functions might include:

A method to convert the entire list to a BST.

Helper methods to handle the conversion process, possibly involving recursion.

HashNode.java: This file probably contains a class definition for a node in a hash table. Typical functions might include:

Constructor for initializing the node with a key-value pair.

Getters and setters for the key, value, and possibly a reference to the next node (in case of collision handling).

Main.java: This is likely the main driver class of your application. It could contain:

The main method to run the application.

Code to demonstrate or test the functionalities of the other classes (like creating and manipulating BSTs, DLLs, and hash tables).

Search.java: This file likely contains various search algorithms. Functions could include:

Methods for linear search, binary search, and hash-based search.

Possibly helper methods for setting up data structures for these searches.

Sort.java: This file is expected to contain sorting algorithms. It might include functions for:

Different sorting methods like bubble sort, quicksort, merge sort, etc.

Possibly utility functions for partitioning arrays, merging arrays, etc.

UML Diagram

A computer screen shot of a computer screen

Description automatically generated

Pseudo code for sorting

**public** **void** SelectionSort(){

DLLNode current = **this**.list;

**while** (current != **null**) {

// Find the minimum value node in the remaining unsorted part of the list

DLLNode minNode = findMinNode(current);

// Swap the values of the current node and the minimum value node

T temp = (T) current.getItem();

current.setItem(minNode.getItem());

minNode.setItem(temp);

// Move to the next node in the unsorted part

current = current.getNext();

}

}

**private** DLLNode<T> findMinNode(DLLNode<T> startNode) {

**if** (startNode == **null**) {

**return** **null**;

}

DLLNode<T> minNode = startNode;

DLLNode<T> current = startNode.getNext();

**while** (current != **null**) {

**if** (current.compareTo(minNode) < 0) {

minNode = current;

selectionSortComparisons++;

}

current = current.getNext();

}

**return** minNode;

}

**public** **void** InsertionSort() {

DLLNode<T> sorted = **null**;

DLLNode<T> current = **this**.list;

**while** (current != **null**) {

DLLNode<T> next = current.getNext();

sorted = insert(sorted, current);

current = next;

}

**this**.list = sorted;

}

**private** DLLNode<T> insert(DLLNode<T> sorted, DLLNode<T> newNode) {

**if** (sorted == **null** || sorted.compareTo(newNode) >= 0) {

newNode.setNext(sorted);

**if** (sorted != **null**) {

sorted.setBack(newNode);

}

**return** newNode;

}

DLLNode<T> current = sorted;

**while** (current.getNext() != **null** && current.getNext().compareTo(newNode) < 0) {

current = current.getNext();

insertionSortComparisons++;

}

newNode.setNext(current.getNext());

**if** (current.getNext() != **null**) {

current.getNext().setBack(newNode);

}

current.setNext(newNode);

newNode.setBack(current);

**return** sorted;

}

**public** **void** BubbleSort() {

DLLNode<T> current = **this**.list;

DLLNode<T> last = lastNode(**this**.list);

**while** (current != **null**) {

DLLNode<T> next = current.getNext();

**while** (next != **null**) {

T currentItem = current.getItem();

T nextItem = next.getItem();

**if** (currentItem != **null** && nextItem != **null** && currentItem.compareTo(nextItem) > 0) {

// Swap elements

T temp = current.getItem();

current.setItem(next.getItem());

next.setItem(temp);

bubbleSortComparisons++;

}

next = next.getNext();

}

current = current.getNext();

}

}

**public** **void** QuickSort() {

DLLNode<T> head = **this**.list;

DLLNode<T> tail = lastNode(list);

quickSort(head, tail);

}

**private** **void** quickSort(DLLNode<T> low, DLLNode<T> high) {

**if** (low != **null** && high != **null** && low != high && low != high.getNext()) {

DLLNode<T> pivot = partition(low, high);

quickSort(low, pivot.getBack());

quickSort(pivot.getNext(), high);

}

}

**private** DLLNode<T> partition(DLLNode<T> low, DLLNode<T> high) {

T pivot = high.getItem();

DLLNode<T> i = **new** DLLNode<>(); // Temporary dummy node

i.setNext(low);

**for** (DLLNode<T> j = low; j != high; j = j.getNext()) {

T jItem = j.getItem();

// Assuming null values are treated as less than non-null values

**if** (jItem == **null** || (pivot != **null** && jItem.compareTo(pivot) <= 0)) {

i = i.getNext(); // Move i to the next node

swap(i, j);

quickSortComparisons++;

}

}

swap(i.getNext(), high); // Place pivot in the correct position

**return** i.getNext(); // Return the position of the pivot

}

**private** **void** swap(DLLNode<T> a, DLLNode<T> b) {

T temp = a.getItem();

a.setItem(b.getItem());

b.setItem(temp);

}

**private** DLLNode<T> lastNode(DLLNode<T> node) {

**while** (node != **null** && node.getNext() != **null**) {

node = node.getNext();

}

**return** node;

}

**public** **void** HeapSort() {

buildMaxHeap();

**for** (**int** i = size; i > 1; i--) {

swap(getNode(1), getNode(i)); // Swap root with the last element

size--;

heapify(getNode(1), 1);

}

}

**private** **int** calculateSize() {

**int** count = 0;

DLLNode<T> current = list;

**while** (current != **null**) {

count++;

current = current.getNext();

}

**return** count;

}

**private** **void** buildMaxHeap() {

size = calculateSize();// Determine the size of the list

**for** (**int** i = size / 2; i >= 1; i--) {

heapify(getNode(i), i);

}

}

**private** **void** heapify(DLLNode<T> node, **int** i) {

**int** leftIndex = 2 \* i;

**int** rightIndex = 2 \* i + 1;

**int** largest = i;

DLLNode<T> left = getNode(leftIndex);

DLLNode<T> right = getNode(rightIndex);

**if** (leftIndex <= size && isGreater(left.getItem(), node.getItem())) {

largest = leftIndex;

heapSortComparisons++;

}

**if** (rightIndex <= size && isGreater(right.getItem(), getNode(largest).getItem())) {

largest = rightIndex;

heapSortComparisons++;

}

**if** (largest != i) {

swap(node, getNode(largest));

heapify(getNode(largest), largest);

}

}

**private** **boolean** isGreater(T a, T b) {

**if** (a == **null** && b == **null**) **return** **false**;

**if** (a == **null**) **return** **false**; // Assuming null is considered less than non-null

**if** (b == **null**) **return** **true**; // Assuming non-null is considered greater than null

**return** a.compareTo(b) > 0;

}

**private** DLLNode<T> getNode(**int** index) {

**if** (index <= 0) {

**return** **null**;

}

DLLNode<T> current = list;

**int** count = 1;

**while** (current != **null** && count < index) {

current = current.getNext();

count++;

}

**return** current;

}

**public** **void** MergeSort() {

**this**.list = mergeSort(**this**.list);

}

**private** DLLNode<T> mergeSort(DLLNode<T> head) {

**if** (head == **null** || head.getNext() == **null**) {

**return** head;

}

DLLNode<T> middle = getMiddle(head);

DLLNode<T> nextOfMiddle = middle.getNext();

middle.setNext(**null**);

DLLNode<T> left = mergeSort(head);

DLLNode<T> right = mergeSort(nextOfMiddle);

**return** sortedMerge(left, right);

}

**private** DLLNode<T> getMiddle(DLLNode<T> head) {

**if** (head == **null**) **return** head;

DLLNode<T> slow = head, fast = head;

**while** (fast.getNext() != **null** && fast.getNext().getNext() != **null**) {

slow = slow.getNext();

fast = fast.getNext().getNext();

}

**return** slow;

}

**private** DLLNode<T> sortedMerge(DLLNode<T> a, DLLNode<T> b) {

**if** (a == **null**) **return** b;

**if** (b == **null**) **return** a;

DLLNode<T> result;

T aItem = a.getItem();

T bItem = b.getItem();

// Handling null items

**if** (aItem == **null** && bItem == **null**) {

result = a;

result.setNext(sortedMerge(a.getNext(), b));

} **else** **if** (aItem == **null**) {

result = a;

result.setNext(sortedMerge(a.getNext(), b));

} **else** **if** (bItem == **null**) {

result = b;

result.setNext(sortedMerge(a, b.getNext()));

} **else** **if** (aItem.compareTo(bItem) <= 0) {

result = a;

result.setNext(sortedMerge(a.getNext(), b));

mergeSortComparisons++;

} **else** {

result = b;

result.setNext(sortedMerge(a, b.getNext()));

}

**return** result;

}

Pseudocode for searching

**public** **boolean** linearSearch(T item) {

DLLNode<T> current = **this**.list;

**while** (current != **null**) {

**if** (item == **null**) {

**if** (current.getItem() == **null**) {

**return** **true**;

}

} **else** **if** (item.equals(current.getItem())) {

**return** **true**;

}

current = current.getNext();

}

**return** **false**; // Return false if the item is not found

}

**public** **boolean** binarySearch(T item) {

DLLNode<T> start = list;

DLLNode<T> last = getLastNode(list);

**while** (start != **null** && last != **null** && start != last && last.getNext() != start) {

DLLNode<T> mid = getMiddleNode(start, last);

**if** (item == **null**) {

**if** (mid.getItem() == **null**) {

**return** **true**;

}

} **else** **if** (item.equals(mid.getItem())) {

**return** **true**;

} **else** **if** (mid.getItem() != **null** && item.compareTo(mid.getItem()) > 0) {

start = mid.getNext();

} **else** {

last = mid;

}

}

**return** **false**; // Return false if the item is not found

}

**private** DLLNode<T> getLastNode(DLLNode<T> node) {

**while** (node != **null** && node.getNext() != **null**) {

node = node.getNext();

}

**return** node;

}

**private** DLLNode<T> getMiddleNode(DLLNode<T> start, DLLNode<T> last) {

DLLNode<T> slow = start;

DLLNode<T> fast = start;

**while** (fast != last && fast.getNext() != last) {

fast = fast.getNext().getNext();

slow = slow.getNext();

}

**return** slow;

}

**private** **void** initializeHashTable() {

DLLNode<T> current = **this**.list;

**while** (current != **null**) {

addNodeToHashTable(current);

current = current.getNext();

}

}

**private** **void** addNodeToHashTable(DLLNode<T> node) {

**if** ((**double**) size / capacity >= ***LOAD\_FACTOR\_THRESHOLD***) {

resizeHashTable();

}

**int** index = getHashIndex(node.getItem());

HashNode<T> head = hashTable[index];

HashNode<T> newNode = **new** HashNode<>(node.getItem(), node);

newNode.next = head;

hashTable[index] = newNode;

size++;

}

**private** **void** resizeHashTable() {

capacity \*= 2;

HashNode<T>[] newTable = **new** HashNode[capacity];

**for** (HashNode<T> head : hashTable) {

**while** (head != **null**) {

HashNode<T> next = head.next;

**int** newIndex = getHashIndex(head.key);

head.next = newTable[newIndex];

newTable[newIndex] = head;

head = next;

}

}

hashTable = newTable;

}

**private** **int** getHashIndex(T key) {

**if** (key == **null**) {

**return** 0;

}

**int** hashCode = key.hashCode();

**return** Math.*abs*(hashCode % capacity);

}

**public** **boolean** hashSearch(T item) {

**int** index = getHashIndex(item);

HashNode<T> head = hashTable[index];

**while** (head != **null**) {

**if** ((item == **null** && head.key == **null**) || (item != **null** && item.equals(head.key))) {

**return** **true**;

}

head = head.next;

}

**return** **false**;

}

Pseudo code for BST

**package** CS401;

**public** **class** BSTNode<T **extends** Comparable<T>> {

**private** T item;

**private** BSTNode<T> left;

**private** BSTNode<T> right;

**public** BSTNode(T item) {

**this**.item = item;

**this**.left = **null**;

**this**.right = **null**;

}

**public** T getItem() {

**return** item;

}

**public** **void** setItem(T item) {

**this**.item = item;

}

**public** BSTNode<T> getLeft() {

**return** left;

}

**public** **void** setLeft(BSTNode<T> left) {

**this**.left = left;

}

**public** BSTNode<T> getRight() {

**return** right;

}

**public** **void** setRight(BSTNode<T> right) {

**this**.right = right;

}

**public** **boolean** contains(T item) {

**if** (item == **null**) {

**return** **false**;

}

**if** (item.compareTo(**this**.item) == 0) {

**return** **true**;

} **else** **if** (item.compareTo(**this**.item) < 0) {

**return** (left != **null**) && left.contains(item);

} **else** {

**return** (right != **null**) && right.contains(item);

}

}

}

Converting DLL to BST

**package** CS401;

**public** **class** DLLToBSTConverter<T **extends** Comparable<T>> {

**private** DLLNode<T> head; // Head of the DLL

**public** DLLToBSTConverter(DLLNode<T> head) {

**this**.head = head;

}

**private** **int** getSize(DLLNode<T> node) {

**int** size = 0;

DLLNode<T> current = node;

**while** (current != **null**) {

size++;

current = current.getNext();

}

**return** size;

}

**public** BSTNode<T> convertToBST() {

**int** size = getSize(head);

**return** convertToBSTRecursive(size);

}

**private** BSTNode<T> convertToBSTRecursive(**int** size) {

**if** (size <= 0) {

**return** **null**;

}

BSTNode<T> left = convertToBSTRecursive(size / 2);

BSTNode<T> root = **new** BSTNode<>(head.getItem());

head = head.getNext();

BSTNode<T> right = convertToBSTRecursive(size - size / 2 - 1);

root.setLeft(left);

root.setRight(right);

**return** root;

}

}