CS5800: Algorithms — Virgil Pavlu

Homework 8

Name: Aaron
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1 Hash Table:

1. (50 points)

Implement a hash for text. Given a string as input, construct a hash with words as keys, and word counts as values. Your implementation should include:

- a hash function that has good properties for text
- storage and collision management using linked lists
- operations: insert(key,value), delete(key), increase(key), find(key), list-all-keys

Output the list of words together with their counts on an output file. For this problem,you cannot use built-in-language datastuctures that can index by strings (like hashtables). Use a language that easily implements linked lists, like C/C++.
You can test your code on "Alice in Wonderland" by Lewis Carroll, at link.
The test file used by TA will probably be shorter.
Try these three values for $m = MAXHASH$: $30, 300, 1000$. For each of these m values, produce a histogram over the lengths of collision lists. You can also calculate variance of these lengths.
If your hash is close to uniform in collisions, you should get variance close to zero, and almost all list-lengths around $\alpha = n/m$.
If your hash has long lists, we want to know how many and how long, for example print the lengths of the longest 10% of the lists.
(Extra Credit) Find a way to record not only word counts, but also the positions in text. For each word, besides the count value, build a linked list with positions in the given text. Output this list together with the count.

Solution:

For sake of brevity/organization I'm just including the .cpp for the Hash Table structure, the other files can be found here: asoiceNU/CS-5800

```
\#include <iostream>\#include <fstream>\#include <sstream>\#include <string>\#include <cctype>\

\#include "hash\_table.hpp"

extern int \__Z10free\_tableP9HashTable;

using namespace std;

// Node structure for the linked list

// Hash table structure//class HashTable \{//public:HashTable::HashTable(int size) : size(size)

// Simple hash function to hash words into a range of [0, size-1]unsigned int HashTable::hash(c

// Insert a key-value pair into the hash tablevoid HashTable::insert(const string\& word) \{uns
```

```
// Check if the word already exists in the linked listwhile (current != nullptr) \{if (current-
// Word not found, insert it at the beginning of the listNode* new\_node = new Node(word);new\_
// Find a word and return its count, or -1 if not foundint HashTable::find(const string\& word)
while (current != nullptr) \{if (current->word == word) \{cout << word << " frequency: " << cu
// Delete a word from the hash tablevoid HashTable::deleteWord(const string\& word) \{unsigned
while (current != nullptr) \{if (current->word == word) \{if (prev == nullptr) \{table[index] =
// Increase the count of a wordvoid HashTable::increase(const string\& word) \{unsigned int in
while (current != nullptr) \{if (current->word == word) \{current->count++;return;\}current = c
// Print all words and their countsvoid HashTable::list\_all\_keys(ofstream\& output\_file) \{i
// Count the number of collisions and list statisticsvoid HashTable::collision\_statistics() \{
// Collect collision list sizesfor (int i = 0; i < size; i++) \{int length = 0;Node* current =
// Print basic statisticscout << "Total collisions: " << total\_collisions << endl;cout << "Num
// Variance of list lengthsdouble mean = 0;for (int i = 0; i < size; i++) \{mean += bucket\_siz
double variance = 0;for (int i = 0; i < size; i++) \{variance += (bucket\_sizes[i] - mean) * (b
cout << "Variance of list lengths: " << variance << endl;
// Generate and print histogram of collision list lengthsprint\_histogram(bucket\_sizes);\}
//private:// int size;// vector<Node*> table; // Hash table with linked lists
// Function to print the histogram of collision list lengthsvoid HashTable::print\_histogram(co
// Print the histogramcout << "\textbackslash{}nHistogram of Collision List Lengths:\textbacksl
// Clean up the hash tablevoid free\_table(HashTable* ht) \{delete ht; // C++ will call the des
```

2 Red Black Tree:

2. (50 points)

Implement a red-black tree, including binary-search-tree operations *sort, search, min, max, successor, predecessor* and specific red-black procedures *rotation, insert, delete.* The *delete* implementation is **Extra Credit** (but highly recommended).

Your code should take the input array of numbers from a file and build a red-black tree with this input by sequence of "inserts". Then interactively ask the user for an operational command like "insert x" or "sort" or "search x" etc, on each of which your code rearranges the tree and if needed produces an output. After each operation also print out the height of the tree.

You can use any mechanism to implement the tree, for example with pointers and struct objects in C++, or with arrays of indices that represent links between parent and children. You cannot use any tree built-in structure in any language.

You can also find the code here: asoiceNU/CS-5800 **Solution:**

rbt.py

```
class Node:def \__init\__(self, data):self.data = dataself.color = "RED" \# All new nodes are

class RedBlackTree:def \__init\__(self):self.TNULL = Node(0) \# Sentinel node (TNULL) to repres

\# Left rotatedef left\_rotate(self, x):y = x.rightx.right = y.leftif y.left != self.TNULL:y.le

\# Right rotatedef right\_rotate(self, x):y = x.leftx.left = y.rightif y.right != self.TNULL:y

\# Insert a node into the treedef insert(self, key):node = Node(key)node.parent = Nonenode.data

y = Nonex = self.root

\# Standard BST insertionwhile x != self.TNULL:y = xif node.data < x.data:x = x.leftelse:x = x

node.parent = yif y == None:self.root = nodeelif node.data < y.data:y.left = nodeelse:y.right =

if node.parent == None:node.color = "BLACK"return

if node.parent.parent == None:return

\# Fix the red-black tree properties after insertionself.fix\_insert(node)

\# Fix the red-black tree properties after insertiondef fix\_insert(self, k):while k.parent.col

\# In-order traversal (to print the sorted elements)def inorder(self, node, result):if node !=

def sort(self):result = []self.inorder(self.root, result)return result

\# Search for a node with a given keydef search(self, node, key):if node == self.TNULL or key =
```

```
\# Find the minimum nodedef minimum(self, node):while node.left != self.TNULL:node = node.leftr
\# Find the maximum nodedef maximum(self, node):while node.right != self.TNULL:node = node.righ
\# Find the successor of a given nodedef successor(self, x):if x.right != self.TNULL:return sel
\# Find the predecessor of a given nodedef predecessor(self, x):if x.left != self.TNULL:return
\# Delete a node from the treedef delete(self, key):node = self.search(self.root, key)if node =
y = nodey\_original\_color = y.colorif node.left == self.TNULL:x = node.rightself.transplant(no
if y\_original\_color == "BLACK":self.fix\_delete(x)
\# Fix the red-black tree after deletiondef fix\_delete(self, x):while x != self.root and x.col
\# Transplant helper function for deletiondef transplant(self, u, v):if u.parent == None:self.r
\# Print the tree structure for verification (simple tree visualization)def print\_tree(self, n
\# Read values from a file and insert them into the treedef read\_values\_from\_file(filename)
\# Interactive User Inputdef main():rbt = RedBlackTree()
\# Read values from input.txt and insert into the Red-Black Treevalues = read\_values\_from\_fi
if command.startswith("insert"):try:value = int(command.split()[1])rbt.insert(value)print(f"\te
elif command.startswith("delete"):try:value = int(command.split()[1])rbt.delete(value)print(f"
elif command.startswith("search"):try:value = int(command.split()[1])node = rbt.search(rbt.root
elif command == "sort":sorted\_values = rbt.sort()print("\textbackslash{}nSorted Tree:", sorted
elif command == "min":min\_node = rbt.minimum(rbt.root)print("\textbackslash{}nMin:", min\_node
elif command == "max":max\_node = rbt.maximum(rbt.root)print("\textbackslash{}nMax:", max\_node
elif command.startswith("successor"):try:value = int(command.split()[1])node = rbt.search(rbt.r
elif command.startswith("predecessor"):try:value = int(command.split()[1])node = rbt.search(rbt
elif command == "print":rbt.print\_tree(rbt.root)
```

```
elif command == "exit":print("Exiting the program.")break

else:print("Unknown command. Try again.")

if \__name\__ == "\__main\__":main()
```

3 Skiplist:

3. (50 points)

Study the skiplist data structure and operations. They are used for sorting values, but in a datastructure more effficient than lists or arrays, and more guaranteed than binary search trees. Review Slides skiplists.pdf and Visualizer. The demo will be a sequence of operations (asked by TA) such as for example insert 20, insert 40, insert 10, insert 20, insert 5, insert 80, delete 20, insert 100, insert 20, insert 30, delete 5, insert 50, lookup 80, etc

Solution:

Here we simply include the .cpp for the skiplist structure itself. The other files were shown in the code demo and can be found here: asoiceNU/CS-5800

```
\#include "SkipList.hpp"\#include <iostream>\#include <cstdlib>

SkipList::SkipList(int maxLevel, float probability) \{this->maxLevel = maxLevel;this->probabili

// Create a header node with a maximum level and no valueheader = new SkipListNode(-1, maxLevel

int SkipList::randomLevel() \{int lvl = 1;while (static\_cast<float>(rand()) / RAND\_MAX < prob

SkipListNode* SkipList::search(int value) \{SkipListNode* current = header;for (int i = level -

void SkipList::insert(int value) \{std::vector<SkipListNode*> update(maxLevel, nullptr);SkipLis

// Find the insertion point for each levelfor (int i = level - 1; i >= 0; i--) \{while (current

// Move to the next node at level 0current = current->forward[0];

// If the node already exists, don't insert itif (current \&& current->value == value) \{return

// Randomly determine the level of the new nodeint newLevel = randomLevel();

// Update the level of the skip list if necessaryif (newLevel > level) \{for (int i = level; i

// Create the new nodeSkipListNode* newNode = new SkipListNode(value, newLevel);

// Update the forward pointers for the new node and the nodes in the update listfor (int i = 0
```

```
void SkipList::deleteNode(int value) \{std::vector<SkipListNode*> update(maxLevel, nullptr);Ski

// Find the node to deletefor (int i = level - 1; i >= 0; i--) \{while (current->forward[i] \&&

// Move to the next node at level 0current = current->forward[0];

// If the node is not found, returnif (current == nullptr || current->value != value) \{std::co

// Update the forward pointers to remove the nodefor (int i = 0; i < level; i++) \{if (update[i

// Decrease the level of the skip list if necessarywhile (level > 1 \&& header->forward[level -

// Free the memory for the nodedelete current;\}

void SkipList::printList() \{for (int i = 0; i < level; i++) \{SkipListNode* current = header->
```

4 Binomial Heap:

This wasn't included in the LaTex template which meant that I nearly didn't see it–Boooooooo.

Again, this is the primary .cpp that defines the structure, the other files in the repo can be found here: asoiceNU/CS-5800

```
\#include "BinomialHeap.hpp"\#include <iostream>\#include <climits>\#include <vector>\#include

using namespace std;

// Merges two binomial trees of the same degreeBH\_Node* BinomialHeap::mergeTrees(BH\_Node* t1

t2->parent = t1;t2->sibling = t1->child;t1->child = t2;t1->degree++;

return t1;\}

// Merges two binomial heapsBH\_Node* BinomialHeap::mergeHeaps(BH\_Node* h1, BH\_Node* h2) \{i

BH\_Node *dummy = new BH\_Node(INT\_MIN), *tail = dummy;BH\_Node *t1 = h1, *t2 = h2;

while (t1 \&& t2) \{if (t1->degree < t2->degree) \{tail->sibling = t1;t1 = t1->sibling;\} else

// Append remaining treesif (t1) tail->sibling = t1;if (t2) tail->sibling = t2;

BH\_Node* newHead = dummy->sibling;delete dummy;

return newHead;\}
```

```
// Consolidates the heap to ensure there are no two trees of the same degreevoid BinomialHeap:
vector<BH\_Node*> degrees(32, nullptr);BH\_Node* prev = nullptr;BH\_Node* curr = head;BH\_Node*
while (curr) \{next = curr->sibling;int d = curr->degree;
while (degrees[d]) \{BH\_Node* other = degrees[d];if (other->key < curr->key) \{swap(curr, othe
if (curr->sibling == other) \{curr->sibling = other->sibling;\}
mergeTrees(curr, other);degrees[d] = nullptr;d++;\}
degrees[d] = curr;prev = curr;curr = next;\}
head = nullptr;for (int i = 0; i < 32; i++) \{if (degrees[i]) \{if (!head) \{head = degrees[i]
// Insert a new key into the heapvoid BinomialHeap::insert(int key) \{// Create a new nodeBH\_
// Extract the minimum key from the heapint BinomialHeap::extractMin() \{if (!head) return INT
BH\_Node* minNode = head;BH\_Node* minPrev = nullptr;BH\_Node* prev = nullptr;BH\_Node* curr =
while (curr) \{if (curr->key < minNode->key) \{minNode = curr;minPrev = prev;\}prev = curr;curr
if (minPrev) minPrev->sibling = minNode->sibling;else head = minNode->sibling;
BH\_Node* child = minNode->child;BH\_Node* reversedChild = nullptr;
while (child) \{BH\_Node* nextChild = child->sibling;child->sibling = reversedChild;child->pare
head = mergeHeaps(head, reversedChild);int minValue = minNode->key;delete minNode;return minVal
// Get the minimum key without removing itint BinomialHeap::getMin() \{if (!head) return INT\_M
while (curr) \{minVal = min(minVal, curr->key);curr = curr->sibling;\}
return minVal;\}
// Union of two binomial heapsvoid BinomialHeap::unionHeaps(BinomialHeap\& other) \{head = merg
// Decrease the key of a nodevoid BinomialHeap::decreaseKey(int oldKey, int newKey) \{if (newKe
BH\_Node* node = findNode(head, oldKey);if (!node) \{std::cout << "Key not found!\textbackslash
```

```
// Decrease the keynode->key = newKey;

// Percolate up the tree to restore heap propertyBH\_Node* parent = node->parent;while (parent

// Delete a key from the heapvoid BinomialHeap::deleteKey(int key) \{decreaseKey(key, INT\_MIN

// Helper function to print binomial heap (in order)void BinomialHeap::printHeap(BH\_Node* roo

BH\_Node* current = root;while (current) \{std::cout << current->key << " ";printHeap(current-

// Helper function to find a node by key (for Decrease Key and Delete)BH\_Node* BinomialHeap::

BH\_Node* res = findNode(root->child, key);if (res) return res;

return findNode(root->sibling, key);\}

BinomialHeap::BinomialHeap() : head(nullptr) \{\}

void BinomialHeap::print() \{printHeap(head);std::cout << std::endl;\}
```

5 Exercise 16.3-3:

4. Consider an ordinary binary min-heap data structure supporting the instructions INSERT and EXTRACT-MIN that, when there are items in the heap, implements each operation in lg worst-case time. Give a potential function such that the amortized cost of INSERT is lg and the amortized cost of EXTRACT-MIN is , and show that your potential function yields these amortized time bounds. Note that in the analysis, is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever be stored in the heap.

Solution:

5.1 Answer:

5.1.1 Potential Function

Here we can do this by assigning the following potential function:

$$\Phi(H) = \Sigma_i^n(lg(k)) = nlg(n)$$

Where n is the number of elements in the heap.

5.1.2 Insert Analysis:

Insert is allowed to take O(lg(n)) time, and this pans out.

From the problem, the actual cost of inserting an element is O(lg(n)). Inserting an element, naturally, adds another element to the heap–resulting in a potential difference of:

$$\Delta_n \Phi(H) = \Sigma_i^{n+1}(lg(k)) - \Sigma_i^n(lg(k)) = lg(n+1) = 2 * lg(n) = O(lg(n))$$

Combined together then we get:

$$Amor = T(n) + \Delta_n \Phi(H) = O(lg(n)) + O(lg(n)) = O(lg(n))$$

5.1.3 Extract-Min Analysis:

We need this to happen in O(1) time amortized–this is going to require some compensation as, we have from the problem that, Extract-Min takes O(lg(n)) time. However, because Extract-Min subtracts an element (the minimum) from an array–our potential function allows this to happen in O(1) amortized.

$$\Delta_n \Phi(H) = \Sigma_i^n(lg(k)) - \Sigma_i^{n-1}(lg(k)) = -lg(n)$$

All in we get:

$$Amor = T(n) + \Delta_n \Phi = O(lg(n)) - lg(n) = O(1)$$

5.1.4 Conclusion:

Here we arrive at a potential function such that the amortized cost of Insert is O(lg(n))–though admittedly, at much higher coefficient–and Extract-Min is amortized to O(1).

In a more general sense this makes sense that we can do this, since removing an element would require first inserting it–the insert-remove pair takes O(lg(n)) with or without the potential function, which makes a lot of sense. Further, the requirement to insert an element without removing gives a more intuitive reason for why the potential is positive definite (which is a requirement).

□