Universidad Peruana de Ciencias Aplicadas

Logotipo

Descripción generada automáticamente

Trabajo Final

“Lastpass”

Algoritmos y estructuras de datos

Sección: SX32

Profesor: Abraham Sopla Maslucan

Integrantes:

* Coronado Puicón, José Carlos U202018935
* Cueva Elera, Erick Armando U201910151
* Urbizagástegui Alvarez, Erick Gabriel U20201E465

2022

* **Explicación del caso de estudio:**

Para el presente proyecto se buscó implementar simulación de una aplicación de uso público. En esta oportunidad a nuestro grupo nos tocó representar un Gestor de contraseñas, para esto nos basamos en una aplicación conocida como Lastpass. Lo que permite esta aplicación es crear una cuenta de usuario, y poder guardad diversas contraseñas y códigos de manera segura. Haciendo uso de diversas estructuras y algoritmos de ordenamiento, se busca representar de manera lógica la aplicación anterior mencionada.

* **El diagrama de clases:**

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

* **Explicación de las estructuras necesarias:**

HashTable: Es una estructura de datos la cual asocia llaves o *keys* con valores. La operación principal que soporta de manera eficiente es la búsqueda. Permite el acceso a los elementos almacenados a partir de una clave generada. En pocas palabras, este funciona transformando la clave con una función hash en un hash, un número que identifica la posición donde la *HashTable* ubica el valor deseado.

Árbol Binario de Búsqueda: Es un árbol binario que cumple que el subárbol izquierdo de cualquier nodo (si este no está vacío), contiene valores menores que el que continue dicho nodo. Y el subárbol derecho (si no está vacío igualmente) continue valores mayores.

Árbol Biselado: Es un árbol binario de búsqueda auto-balanceable, con la propiedad adicional de que a los elementos accedidos recientemente se accederá más rápidamente en accesos posteriores. Este realiza operaciones básicas como la inserción, la búsqueda y el borrado.

* **Estudio de la complejidad.**

La complejidad algorítmica representa la cantidad de recursos temporales que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo. Algunos de nuestros métodos alcanzaban un tiempo asintótico de O(log n), O(n) y O(1).

A continuación, se presentará el código y el análisis big O de diversas funciones de nuestro proyecto.

Árbol Binario de Búsqueda:

1. #include <functional>
2. using namespace std;
4. template <class T>
5. class Nodo {
6. public:
7. T elemento; // 1
8. Nodo\* izq; // 1
9. Nodo\* der; // 1
10. };
12. template <class T>
13. class ArbolBB {
14. typedef function<int(T, T)> Comp; //Lambda
16. Nodo<T>\* raiz;
17. void(\*procesar)(T); //Puntero a funcion
19. Comp comparar; // lambda de criterio de comparación

22. private:
23. bool \_buscar(Nodo<T>\* nodo, T e) { // 6 + n -> 11 + n
24. if (nodo == nullptr) return false; // 1 + 1
25. else {
26. int r = comparar(nodo->elemento, e); // 3
27. if (r == 0) return true; // 1 + n -> 6 + n
28. else if (r < 0) {
29. return \_buscar(nodo->der, e);// n -> 6 + n
30. }
31. else {
32. return \_buscar(nodo->izq, e); // n -> 6 + n
33. }
34. }
35. }
36. Nodo<T>\* \_obtener(Nodo<T>\* nodo, T e) { // 6 + n
37. if (nodo == nullptr) return nullptr; // 1 + 1
38. else {
39. int r = comparar(nodo->elemento, e); // 3
40. if (r == 0) return nodo; // 1 + n
41. else if (r < 0) {
42. return \_buscar(nodo->der, e); // n
43. }
44. else {
45. return \_buscar(nodo->izq, e); // n
46. }
47. }
48. }
49. bool \_insertar(Nodo<T>\*& nodo, T e) { // 5 + n -> 10 + n
50. if (nodo == nullptr) { // 1 + 4 + n -> 1 + 9 + n
51. nodo = new Nodo<T>(); // 1
52. nodo->elemento = e; // 1
53. return true; // 1
54. }
55. else {
56. int r = comparar(nodo->elemento, e); // 3
57. if (r == 0) return false; // 1 + n -> 6 + n
58. else if (r < 0) {
59. return \_insertar(nodo->der, e); // n -> 5 + n
60. }
61. else {
62. return \_insertar(nodo->izq, e); // n -> 5 + n
63. }
64. }
65. }
66. void \_enOrden(Nodo<T>\* nodo) { // 2 + 2n -> 6 + 2n
67. if (nodo == nullptr) return; // 1 + 2n + 1 -> 6 + 2n
68. \_enOrden(nodo->izq); // n -> 2 + 2n
69. procesar(nodo->elemento); // 1
70. \_enOrden(nodo->der); // n -> 2 + 2n
71. }
73. void \_preOrden(Nodo<T>\* nodo) { // 2 + 2n -> 6 + 2n
74. if (nodo == nullptr) return; // 1 + 2n + 1 -> 6 + 2n
75. procesar(nodo->elemento); // 1
76. \_preOrden(nodo->izq); // n -> 2 + 2n
77. \_preOrden(nodo->der); // n -> 2 + 2n
78. }
80. void \_postOrden(Nodo<T>\* nodo) { // 2 + 2n -> 6 + 2n
81. if (nodo == nullptr) return; // 1 + 2n + 1 -> 6 + 2n
82. \_postOrden(nodo->izq); // n -> 2 + 2n
83. \_postOrden(nodo->der); // n -> 2 + 2n
84. procesar(nodo->elemento); // 1
85. }
86. bool \_vacio() { // 1
87. return raiz == nullptr; // 1
88. }
89. int \_cantidad(Nodo<T>\* nodo) {  // 8 + 2n
90. //La cantidad de nodos del árbol es:
91. //      0 si es vacío
92. //      1 + la cantidad de nodos por la izquierda + la cantidad de nodos por la derecha
94. if (nodo == nullptr) // 1 + 7 + 2n
95. return 0; // 1
96. else
97. {
98. int ci, cd; // 2
99. ci = \_cantidad(nodo->izq); // n + 1
100. cd = \_cantidad(nodo->der); // n + 1
101. return 1 + ci + cd; // 3
102. }
104. }
105. int \_altura(Nodo<T>\* nodo) { // 1 + 7 + 2n
106. //La altura del árbol es:
107. //      0 si es vacío
108. //      la mayor de las alturas por la izquierda y por la derecha, las cuáles son 0 si son vacías ó 1 + la altura por la izq(o der) en caso contrario
110. if (nodo == nullptr) // 1 + 7 + 2n
111. return 0; // 1
112. else
113. {
114. int ai, ad; // 2
115. ai = 1 + \_altura(nodo->izq); // n + 1
116. ad = 1 + \_altura(nodo->der); // n + 1
117. return ai > ad ? ai : ad; // 3
118. }
119. }
121. int \_minimo(Nodo<T>\* nodo) { // 2 + n
122. if (nodo->izq == nullptr) return nodo->elemento; // 2
123. else
124. return \_minimo(nodo->izq); // n
125. }
126. int \_maximo(Nodo<T>\* nodo) { // 2 + n
127. if (nodo->der == nullptr) return nodo->elemento; // 2
128. else
129. return \_maximo(nodo->der); // n
130. }
131. bool \_eliminar(Nodo<T>\*& nodo, T e) {
132. if (nodo == nullptr) return false;
133. else {
134. int r = comparar(nodo->elemento, e);
135. if (r < 0) {
136. return \_eliminar(nodo->der, e);
137. }
138. else if (r > 0) {
139. return \_eliminar(nodo->izq, e);
140. }
141. else { // r==0 es porque se encontró el elemento e en el arbol
142. if (nodo->der == nullptr && nodo->izq == nullptr) {//caso 1 nodo hoja
143. nodo = nullptr;
144. delete nodo;
145. return true;
146. }
147. else if (nodo->izq == nullptr) { //caso 2 nodo con hijo derecho
148. nodo = nodo->der;
149. return true;
150. }
151. else if (nodo->der == nullptr) { //caso 3 nodo con hijo izquierdo
152. nodo = nodo->izq;
153. return true;
154. }
155. else { //caso 4
156. Nodo<T>\* aux = nodo->der; //Se establece buscar el menor elemento por la derecha
157. while (aux->izq != nullptr)
158. {
159. aux = aux->izq; //Se obtiene la hoja menor
160. }
161. nodo->elemento = aux->elemento; //Se actualiza el elemento en el nodo raiz y
162. return \_eliminar(nodo->der, aux->elemento); //se envía a eliminar el elemento en el arbol por la derecha
163. }
164. }
165. }
166. }
167. public:
168. ArbolBB(void(\*otroPunteroAFuncion)(T)) {
169. this->procesar = otroPunteroAFuncion; // 1
170. this->comparar = [](T a, T b)->int {/\*return a - b;\*/if (a > b)return 1; else if (a < b)return -1; else return 0; }; // 2
171. raiz = nullptr;
172. }
173. bool insertar(T e) {
174. return \_insertar(raiz, e);
175. }
176. void enOrden() {
177. \_enOrden(raiz);
178. }
179. void preOrden() {
180. \_preOrden(raiz);
181. }
182. void postOrden() {
183. \_postOrden(raiz);
184. }
185. int cantidad() {
186. return \_cantidad(raiz);
187. }
188. int altura() {
189. return \_altura(raiz);
190. }
191. bool Buscar(T e) {
192. return \_buscar(raiz, e);
193. }
194. int Minimo() {
195. return \_minimo(raiz);
196. }
197. int Maximo() {
198. return \_maximo(raiz);
199. }
200. bool Eliminar(T e) {
201. return \_eliminar(raiz, e);
202. }
203. };

Árbol Biselado:

1. #include <functional>
2. using namespace std;
4. template<class T>
5. class SPLNodo
6. {
7. public:
8. T dato; // 1
9. SPLNodo\* izq; // 1
10. SPLNodo\* der; // 1
12. SPLNodo(T dato) {
13. this->dato = dato; // 1
14. this->der = nullptr; // 1
15. this->izq = nullptr; // 1
16. }
17. ~SPLNodo();
18. void setDato(T dato) {
19. this->dato = dato; // 1
20. }
21. void setIzq(SPLNodo<T>\* izq) {
22. this->izq = izq; // 1
23. }
24. void setDer(SPLNodo<T>\* der) {
25. this->der = der; // 1
26. }
27. };
29. template<class T>
30. class ArbolSPL
31. {
32. public:
33. ArbolSPL() { // 5
34. this->raiz = nullptr; // 1
35. this->comparar = [](T a, T b)->int { // 4
36. if (a < b) { // 1 + 1
37. return 1; // 1
38. }
39. else if (a > b) {
40. return -1; // 1
41. }
42. else {
43. return 0; // 1
44. }
45. };
46. }
47. ~ArbolSPL();
48. void insertar(T dato) { // 2
49. if (\_insertar(dato, this->raiz)) { // 2
50. cout << "Dato insertado exitosamente." << endl; // 1
51. }
52. else {
53. cout << "Hubo un error." << endl;  // 1
54. }
55. }
56. void buscar(T dato) { // 2
57. if (\_buscar(dato, this->raiz)) { // 1 + 1
58. moverNodoARaiz(dato, this->raiz); // 1
59. }
60. }
61. void eliminar(T dato) { // 2
62. if (\_eliminar(dato)) { // 1 + 1
63. cout << "Dato eliminado exitosamente." << endl; // 1
64. }
65. else {
66. cout << "Hubo un error." << endl; // 1
67. }
68. }
69. void preOrden() { // 1
70. \_preOrden(this->raiz); // 1
71. }
72. private:
73. SPLNodo<T>\* raiz;
74. typedef function<int(T, T)> Comp; //Lambda
75. Comp comparar; //criterio de comparación
76. void zig(SPLNodo<T>\*& nodo) { // 4
77. SPLNodo<T>\* aux = nodo; // 1
78. nodo = nodo->izq; // 1
79. aux->izq = nodo->der; // 1
80. nodo->der = aux; // 1
81. }
82. void zigZig(SPLNodo<T>\*& nodo) { // 2
83. zig(nodo); // 1
84. zig(nodo); // 1
85. }
86. void zigZag(SPLNodo<T>\*& nodo) { // 2
87. zag(nodo->izq); // 1
88. zig(nodo); // 1
89. }
90. void zag(SPLNodo<T>\*& nodo) { // 4
91. SPLNodo<T>\* aux = nodo; // 1
92. nodo = nodo->der; // 1
93. aux->der = nodo->izq; // 1
94. nodo->izq = aux; // 1
95. }
96. void zagZag(SPLNodo<T>\*& nodo) { // 2
97. zag(nodo); // 1
98. zag(nodo); // 1
99. }
100. void zagZig(SPLNodo<T>\*& nodo) { // 2
101. zig(nodo->der); // 1
102. zag(nodo); // 1
103. }
104. void rotaciones(T dato, SPLNodo<T>\*& nodo) { // n
105. if (nodo->dato > dato) { // 1 + 5
106. if (nodo->izq->dato == dato) { // 1 + 4
107. zig(nodo); // 4
108. }
109. else if (nodo->izq->dato > dato && nodo->izq->izq->dato == dato) {
110. zigZig(nodo); // 2
111. }
112. else if (nodo->izq->dato < dato && nodo->izq->der->dato == dato) {
113. zigZag(nodo); // 2
114. }
115. else {
116. rotaciones(dato, nodo->izq); // n
117. }
118. }
119. else if (nodo->dato < dato) { // 5
120. if (nodo->der->dato == dato) { // 1 + 4
121. zag(nodo); // 4
122. }
123. else if (nodo->der->dato > dato && nodo->der->izq->dato == dato) {
124. zagZig(nodo); // 2
125. }
126. else if (nodo->der->dato < dato && nodo->der->der->dato == dato) {
127. zagZag(nodo); // 2
128. }
129. else {
130. rotaciones(dato, nodo->der); // n
131. }
132. }
133. }
134. void moverNodoARaiz(T dato, SPLNodo<T>\*& nodo) { // n^2
135. while (nodo->dato != dato) { // n \* n
136. rotaciones(dato, nodo); // n
137. }
138. }
139. bool \_insertar(T dato, SPLNodo<T>\*& nodo) { // n
140. if (nodo == nullptr) { // 1 + 4 + n
141. nodo = new SPLNodo<T>(dato); // 1
142. return true; // 1
143. }
144. else {
145. int r = comparar(dato, nodo->dato); // 2
146. if (r == 0) return false; // 2 + n
147. else if (r < 0) { // 1 + n
148. return \_insertar(dato, nodo->der); // n
149. }
150. else {
151. return \_insertar(dato, nodo->izq); // n
152. }
153. }
155. moverNodoARaiz(dato, this->raiz); // n^2
156. }
157. bool \_buscar(T dato, SPLNodo<T>\* nodo) { // 3 + n
158. if (nodo != nullptr) { // 2 + n
159. if (nodo->dato == dato) { // 1 + n
160. return true; // 1
161. }
162. else if (nodo->dato > dato) {
163. return \_buscar(dato, nodo->izq); // n
164. }
165. else if (nodo->dato < dato) {
166. return \_buscar(dato, nodo->der); // n
167. }
168. }
169. return false; // 1
170. }
171. T obtenerMayorHijo(SPLNodo<T>\*& nodo) { // 3 + n
172. if (nodo->der == nullptr) { // 2
173. return nodo->dato; // 1
174. }
176. return obtenerMayorHijo(nodo->der); // n
177. }
178. bool \_eliminar(T dato) { // 2n^2 + n + 11
179. if (!\_buscar(dato, this->raiz)) { // 2
180. return false; // 1
181. }
182. moverNodoARaiz(dato, this->raiz); // n^2
183. if (this->raiz->izq == nullptr && this->raiz->der == nullptr) { // 9 + n + n^2
184. this->raiz = nullptr; // 2
185. }
186. else if (this->raiz->izq != nullptr && this->raiz->der != nullptr) { // 9 + n + n^2
187. T aux = obtenerMayorHijo(raiz->izq); // 3 + n
189. if (aux != this->raiz->izq->dato) { // 1 + n^2
190. moverNodoARaiz(aux, this->raiz->izq); // n^2
191. }
192. this->raiz->izq->der = this->raiz->der; // 3
193. this->raiz = this->raiz->izq; // 2
194. }
195. else if (this->raiz->izq != nullptr) {
196. this->raiz = this->raiz->izq; // 2
197. }
198. else {
199. this->raiz = this->raiz->der; // 2
200. }
202. return true; // 1
203. }
204. void \_preOrden(SPLNodo<T>\* nodo) { // 3 + 2n
205. if (nodo == nullptr) return; // 2
206. cout << nodo->dato << " "; // 1
207. \_preOrden(nodo->izq); // n
208. \_preOrden(nodo->der); // n
209. }
210. };

HashTable:

1. #include <iostream>
2. using namespace std;
4. template<class T>
5. class HashEntidad {
6. private:
7. T key; // 1
8. T value; // 1
9. public:
10. HashEntidad(T key, T value) { // 2
11. this->key = key; // 1
12. this->value = value; // 1
13. }
14. T getKey() { return key; } // 1
15. T getValue() { return value; } // 1
16. void imprimirHashEntidad() { // 1
17. cout << key << "," << value << endl; // 1
18. }
19. };
21. template <class T>
22. class HashTabla {
23. private:
24. HashEntidad<T>\*\* tabla; // 1
25. int numElementos; // 1
26. int TABLE\_SIZE; // 1
27. int deStringAInt(T palabra) { // 2 + 1 + 4n
28. int valor = 0; // 1
30. for (int i = 0; i < palabra.size(); i++) { // 1 + n(1 + 1 + 2) -> 1 + 4n
31. valor += palabra[i]; // 1
32. }
34. return valor; // 1
35. }
36. public:
37. HashTabla(int TABLE\_SIZE = 128) { // 4 + 4n
38. this->TABLE\_SIZE = TABLE\_SIZE; // 1
39. tabla = new HashEntidad<T> \*[TABLE\_SIZE]; // 1
40. for (int i = 0; i < TABLE\_SIZE; ++i) { // 1 + 4n
41. tabla[i] = nullptr; // 1
42. }
43. numElementos = 0; // 1
45. }
46. ~HashTabla()
47. {
48. for (int i = 0; i < TABLE\_SIZE; ++i) { // 1 + 5n
49. if (tabla[i] != nullptr) { // 1 + 1
50. delete tabla[i]; // 1
51. }
52. }
53. delete[] tabla; // 1
54. }
55. //Direccionamiento según Prueba Lineal
56. void insertar(T key, T value) { // 11 + 4n
57. //Hash prima
58. int base, step, hash; // 3
59. //validar si la tabla está llena
60. if (numElementos == TABLE\_SIZE) cout << "La tabla esta llena.**\n**"; // 2
61. //Función Hash1
62. base = deStringAInt(key) % TABLE\_SIZE; // 2
63. hash = base; // 1
64. //constante para Hash2
65. step = 0; // 1
66. while (tabla[hash] != nullptr) // n(4)
67. {
68. //Función Hash2
69. hash = (base + step) % TABLE\_SIZE; // 3
70. step++; // 1
71. }
72. //almacenarlo en la tabla
73. tabla[hash] = new HashEntidad<T>(key, value); // 1
74. numElementos++; // 1
75. }
76. int size() { // 1
77. return TABLE\_SIZE; // 1
78. }
79. int sizeactual() { // 1
80. return numElementos; // 1
81. }
82. T obtener(T key) { // 7 + 5n
83. int step = 0; // 1
84. int i, base; // 2
85. i = base = deStringAInt(key) % TABLE\_SIZE; // 3 (hash1 es = a hash2 cuando step=0;)
86. string error = "No se encontro.**\n**"; // 1
87. while (true) // 5(n)
88. {
89. if (tabla[i] == nullptr)return error;  // 1 + 1
90. else if (tabla[i]->getKey() == key) {
91. return tabla[i]->getValue(); // 1
92. }
93. else step++; // 1
95. i = (base + step) % TABLE\_SIZE; // 3
96. }
97. }
98. HashEntidad<T>\*\* getTabla() {
99. return tabla; // 1
100. }
101. void imprimirHT() { // 1 + 5n
102. for (int i = 0; i < TABLE\_SIZE; ++i) { // 1 + n(3 + 2)
103. cout << "POSICION " << i << ": "; // 1
104. tabla[i]->imprimirHashEntidad(); // 1
105. }
106. }
107. };

Ordenamientos QuickSort, QuickSelect y MergeSort:

1. void intercambiar(string& a, string& b) { // 3
2. string aux = a; // 1
3. a = b; // 1
4. b = aux; // 1
5. }
7. int particion(vector<string>& arreglo, int inicio, int final) { // 10 + 7n
8. string pivote = arreglo[inicio]; // 2
9. int i = inicio + 1; // 3
10. for (int j = i; j <= final; j++) { // 1 + n( 1 + 4 + 2) -> 1 + 7n
11. if (arreglo[j] < pivote) { // 1 + 3
12. intercambiar(arreglo[i], arreglo[j]); // 3
13. i++;
14. }
15. }
16. intercambiar(arreglo[inicio], arreglo[i - 1]); // 3
17. return i - 1; // 1 (pivote)
18. }
20. void quickSort(vector<string>& arreglo, int inicio, int fin) { // 12 + 7n + 2m1 -> m1
21. if (inicio < fin) { // 1 + 11 + 7n + 2m1
22. int pivote = particion(arreglo, inicio, fin); // 1 + 10 + 7n
23. quickSort(arreglo, inicio, pivote - 1); // m1
24. quickSort(arreglo, pivote + 1, fin); // m1
25. }
26. }
28. string quickselect(vector<string>& arreglo, int inicio, int fin, int k) { // 17 + 7n + m1 -> m1
29. if (inicio == fin) return arreglo[inicio]; // 1 + 1
30. int q = particion(arreglo, inicio, fin); // 1 + 10 + 7n
31. int l = q - inicio + 1; // 3
32. if (k == l) // 1 + m1
33. return arreglo[q]; // 1
34. else if (k < l) {
35. return quickselect(arreglo, inicio, q - 1, k); // m1
36. }
37. else {
38. return quickselect(arreglo, q + 1, fin, k - l); // m1
39. }
40. }
42. void merge(vector<string>& arreglo, int inicio, int mitad, int final) { // 18n + 15
43. int i, j, k;  // 3
44. int elementosIzq = mitad - inicio + 1; // 3
45. int elementosDer = final - mitad; // 2
47. vector<string>izquierda(elementosIzq); // 1
48. vector<string>derecha(elementosDer); // 1
50. for (int i = 0; i < elementosIzq; i++) { // 1 + 4n
51. izquierda[i] = arreglo[inicio + i]; // 1
52. }
53. for (int j = 0; j < elementosDer; j++) { // 1 + 4n
54. derecha[j] = arreglo[mitad + 1 + j]; // 1
55. }
57. i = 0; // 1
58. j = 0; // 1
59. k = inicio; // 1
61. while (i < elementosIzq && j < elementosDer) { // 4(n)
62. if (izquierda[i] <= derecha[j]) { // 1 + 2
63. arreglo[k] = izquierda[i]; // 1
64. i++; // 1
65. }
66. else {
67. arreglo[k] = derecha[j]; // 1
68. j++; // 1
69. }
70. k++; // 1
71. }
73. while (j < elementosDer) { // 3(n)
74. arreglo[k] = derecha[j]; // 1
75. j++; // 1
76. k++; // 1
77. }
79. while (i < elementosIzq) { // 3(n)
80. arreglo[k] = izquierda[i]; // 1
81. i++; // 1
82. k++; // 1
83. }
85. }
87. void mergeSort(vector<string>& arreglo, int inicio, int final) { // 19 + 18n + 2m1 -> m1
88. if (inicio < final) { // 1 + 18n + 15 + 3 + 2m1
89. int mitad = inicio + (final - inicio) / 2; // 3
90. mergeSort(arreglo, inicio, mitad); // m1
91. mergeSort(arreglo, mitad + 1, final); // m1
92. merge(arreglo, inicio, mitad, final); // 18n + 15
93. }
94. }

Video de exposición:

<https://youtu.be/wXmjaMNkerE>

Cuadro de participación:

|  |  |  |
| --- | --- | --- |
| Nombre | Tareas | Autovaloración |
| Erick Armando Cueva | QuickSelect  Clase ArbolBB  Adaptaciones del menú | 30% |
| Erick Gabriel Urbizagástegui | QuickSort  Clase ArbolSPL  Clase HashTable | 40% |
| José Carlos Coronado | MergeSort  Big O  Adaptaciones del menu  Informe | 30% |