Pràctica 4

Heaps y hashing

Niub – Nom
16799506 - Rubén Ballester Bautista
16856733 – Oriol Rabasseda Alcaide
Grup de Pràctiques F **– Parella** 1
Professora:
Maria Salomó

Exercici 1:

Es tracta de crear un TAD CuaPrioritaria amb una estructura de dades de Heap implementada a la seva vegada mitjançant vectors (que funcionen d'una forma similar als arrays però amb mida variable).

Aquest vector l'hem plenat de punters a objectes del TAD Position, subTAD de Heap que hem representat d'aquesta forma (Nota, hem implementat directament els exercicis 1 i 2 i per tant donem la implementació final del cercador de paraules):

- E key → Representa la key, que seria la prioritat de l'element.
- vector<pair<N, N>> values → Vector amb valors de línea i número de paraula en la línea on s'ha trobat la paraula al text. Utilitzem un vector per la seva flexibilitat i velocitat i pair per que és el TAD natural per representar un conjunt parell d'elements ordenats.

Funcions comentades TAD MinHeap:

- const int size() → O(1) Ja que retornem variable.
- const bool empty() → O(1) Comprovem si el size és 0
- void insert() → O(log n) Ja que depén de la altura degut a que és representa com un arbre quasi complet (degut al upHeap)
- const E& min() → O(1) El mínim sempre és l'arrel.
- const Position <E,N>* getPosition() → O(n) En el pitjor dels casos haurà de visitar tots els nodes.
- const vector<pair<N, N>> minValues() → O(1) El mínim sempre és l'arrel i retornar els seus valors és O(1) [O(1) + O(1) = O(1)]
- void removeMin() → O(log n) Com és un arbre quasi complet depén de la altura. Els passos bàsics (intercanvi arrel per últim element insertat i eliminació del mateix) són O(1) i després el procés de downHeap és O(log n) (a continuació)
- void printHeap() → O(n) Ha de recorrer tots els elements
- int height() → O(1) Al ser un arbre quasi complet és una operació aritmètica O(1)
- void upHeap() → O(log n) Depén de la altura ja que és un arbre quasi complet i el que fa és anar pujant de nivell (és a dir, recorrent com a màxim la altura del arbre.)
- void downHeap() → O(log n) Depén de la altura ja que és un arbre quasi complet i el que fa és anar baixant de nivell (és a dir, recorrent com a màxim la altura del arbre.)

Funcions comentades TAD Position:

- void newValue() → O(1) Algorisme que depén de push_back del vector que és O(1)
- void toString() → O(#values) Depén del valors afegits. (For Scheme)
- const vector<pair<N,N>> getValues() → O(1) Únicament retorna un atribut.

Exercici 2:

Implementat com a la pràctica anterior, només que ara cridem als mètodes equivalents dels arbre (recordem que els heaps segueixen una representació d'arbres quasi complets i per tant tenen quasi els mateix mètodes).

Nota: Cal destacar que hem implementat exercici 1 i 2 al mateix projecte.

Exericici 3:

Es tracta de crear un TAD HashMap per a la cerca d'elements exhaustiva, de manera que es prioritzi l'accés a elements i la seva cració en poc temps que la seva ordenació dins de la representació del TAD.

La seva representació és aquesta:

- LinkedHashEntry<E,N>* arrayElems[MAX_TABLE] → On MAX_TABLE és una constant entera i primera (que nosaltres hem definit per a la cerca específica en largeText). La plenem de punters al TAD especificat al document adjunt a la pràctica que representen nodes individuals encadenats per fer una taula de Hash oberta.

Les funcions del TAD HashMap són les següents:

- int getHashCode(string) → O(Mida paraula inserida) Utilitza totes les lletres de la paraula inserida i per tant fa un recorregut de mida = paraula inserida. Les operacions aritmètiques associades a cada iteració són O(1) => (Mida Paraula Inserida no fixa) * O(1) = O(mida paraula inserida). Utilitzem el mètode de hash fet a classe de la transformació de strings en polinomis.
- int getHashCode (int) → O(1). No va a passar mai, per tant suposarem que en aquest cas la millor implementació és de la familia de hash que creixen uniformement i únicament apliquem el mòdul de la taula.
- void put() → O(Mida paraula inserida) + O(tassa de col·lisions mitjana) Depén de la funció getHasCode(…) ja què l'accés a un array és O(1). A més a més, com es poden produir col·lisions (hash obert), hem d'agafar el nombre mitjà de col·lisions per al cost del cas promig.
- const bool get() → O(Mida paraula inserida) + O(tassa de col·lisions mitjana) Depén de la funció getHasCode(…) ja què l'accés a un array és O(1). A més a més, com es poden produir col·lisions, hem d'agafar el nombre mitjà de col·lisions per al const del cas promig.
- LinkedHashEntry <E,N>* getPosition() → O(Mida paraula inserida) + O(tassa de col·lisions mitjana) Mateix que get.

Nota: Normalment tassa de col·lisions mitjana tendeix a O(1) i el màxim de lletres d'una paraula és un nombre molt petit que es pot considerar O(1), ja que si prenem com a referència l'idioma castellà, la seva paraula amb més mida (lletres) és *"ciclopentanoperhidrofenantreno"* amb 30 lletres.

(Com a curiositat, cal dir que si haguèssim tingut més temps ens hagués agradat implementar un algorisme de HASH com SHA).

Exercici 4:

Implementat com a l'exercici 2, només que ara cridem als mètodes equivalents del hash.

Nota: Cal destacar que hem implementat exercici 3 i 4 al mateix projecte.

Exercici 5:

LargeText	Temps d’accés (cerca)	Temps generació estructura
HEAP	17.3692	0.156875
TAULES HASH	0.098456	0.006114

smallText	Temps d’accés (cerca)	Temps generació estructura
HEAP	0.862761	0.000483
TAULES HASH	0.094354	0.000333

Els resultats obtinguts són els esperats degut als ordres de ambdós TADs (hem comentat els costos al anàlisi de mètodes de cadascún dels TADs).

Com era d’esperar, per afegir i per a cercar el TAD Hash és molt millor, ara bé, si volguèssim ordenar o trobem mínims o qualsevol cosa que pugui requerir una seqüència, el TAD HashMap romanarà molt més de temps que el TAD Heap.