5-6. Diccionaris

La implementació dels diccionaris generalment es duu a terme amb un hash table.

5.1 Hash table

Un hash table es un tipus d'array que es pot considerar com la generalització del vector. En comptes d'indexar cada possible posició en el seu k valor com fa el vector, la hash table transforma el k valor usant una funció hash i l'indexa allà.

Per saber com d'eficientment s'ha repartit els elements en la taula de hash utilitzem el anomenat factor de carrega ($\boldsymbol{\alpha}$). Que s'obté al dividir el nombre d'elements a emmagatzemar entre el nombre de posicions disponibles.

Eficiència mitja de la cerca: O(1).

Eficiència cas pitjor: O(n).

Espai requerit: Equivalent al nombre de claus emmagatzemades, no al total.

5.1.1 Teoremes derivats

Una cerca sense èxit trigarà un temps equivalent a $\Theta(1+\alpha)$.

Una cerca exitosa trigarà un temps equivalent a $\Theta(1+\alpha)$.

5.2 Arbres de cerca

Es tracta d'una estructura que suporta moltes operacions de conjunts dinàmics. Pot ser usat tant com un diccionari com una cua de prioritats. La cerca d'un element en aquest arbre pren un temps proporcional a la alçada del arbre.

Per assolir aquest tipus de cerca es necessari que les claus dels nodes fills esquerres d'un node x siguin menors o iguals a x.key i que els nodes fills drets d'un node x siguin majors o iguals a x.key.

Per recórrer un arbre de cerca s'usa típicament la cerca en inordre.

Eficiència mitja de la cerca: $\Theta(\log(n))$.

Eficiència cas pitjor: $\Theta(n)$.

Eficiència inserció i esborrat: $\Theta(h)$. On h es la alçada de l'arbre.

6.1 Arbres de carca balancejats (AVL)

Es tracta d'un arbre de cerca habitual però que els dos sub-arbres dret i esquerra com a molt poden diferir en una unitat d'alçada. Això es aconsegueix mantenint en cada node un paràmetre extra que emmagatzema la alçada a la que es troba aquell node.

Per a mantenir el balanç quan s'insereix un nou element els AVL disposen d'una funció de balancejat.

Eficiència mitja de la cerca: $\Theta(\log(n))$.

6.2 Heaps

Un heap es com un arbre complert fins l'últim nivell que pot no estar complert. Hi ha dos tipus diferents d'organitzar els heaps:

Max-heap:

Per a cada node el element mes gran esta al pare, i tots els fills posteriors tenen un valor mes petit. Aquest sistema es el que s'usa en el heapsort.

Min-heap:

Es just l'oposat al max-heap, consisteix en el mateix però en aquest cas el valor mes petit es el que es troba al node pare.

Per a mantenir la propietat del heap s'implementa una funció:

```
MAX-HEAPIFY(A, i)
    ℓ = LEFT(i)
    r = RIGHT(i)
    largest = i
    if ℓ ⩽ A.heap-size and A[ℓ] > A[largest] then
        largest = ℓ
    if r ⩽ A.heap-size and A[r] > A[largest] then
        largest = r
    if largest ≠ i then
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

Per a construir un heap es pot utilitzar aquesta mateixa funció.

Eficiència mitja de la construcció: $\Theta(\log(n) \cdot n)$.

6.3 Cues de prioritat

Les cues de prioritat s'implementen típicament amb un Max-heap. Bàsicament es un set que manté uns valors key associats a cada element, i implementa una funció per veure el màxim de la cua de prioritat.

Eficiència de comprovar el màxim: O(1).

Eficiència de consultat el màxim i extreure'l: O(log n).

Eficiència d'incrementar un element de valor: O(log n).

Eficiència d'insertar un nou element: O(log n).

7-8. Grafs

Un graf pot ser dirigit o no dirigit. G=(V,E). Podem diferenciar dos tipus de grafs:

Graf dens:

Un graf dens es aquell que $|E|$ es proper a $|V|^2$.

Graf dispers:

Un graf dispers es aquell que $|E|$ es molt menys que $|V|^2$.

Moltes vegades un graf es representa amb la **llista d'adjacències** que es un array composat en cada posició per una llista amb tots els vèrtex que son adjacents al vèrtex del primer array.

Un graf també pot ser representat amb una **matriu d'adjacències**, aquesta representació te una avantatge respecte a la llista ja que per **a buscar si dos grafs estan connectats te cost constant**. A mes a mes es pot representar un **graf amb pesos**.

7.1 Busca en profunditat (DFS)

La busca en profunditat consisteix en prioritzar la cerca de nodes no visitats i anar visitant tots els nodes recursivament (o amb un stack iterativament).

7.2 Ordenació topològica

Consisteix en aplicar un DFS i veure si aquest retorna alguna aresta enredera.

7.3 Busca en amplada (BFS)

Serveix per a calcular la distancia de tot vèrtex al vèrtex des de el que s'ha començat el BFS.

7.4 Algoritme de Dijkstra

Es el mateix que el BFS però per grafs amb pesos.

7.5 Algoritme de Prim

Consisteix en crear un arbre en un graf amb pesos recorrent nomes les arestes de mínim pes que connecten un vèrtex amb la resta, això s'implementa amb una priority queue dels menors, implementada amb un min heap.

9. Recerca exhaustiva i generació

9.1 Backtracking

La tècnica del backtracking consisteix en recórrer l'arbre i seleccionar els estats que ens semblin prometedors i descartar els que no.