ESERCIZIO DI ASD DEL 30 MARZO 2009

B-Tree Join

Algoritmo. L'idea di base è la seguente:

- dobbiamo assicurarci di trovarci sempre su un nodo non pieno, quindi come prima cosa splittiamo le radici di T_1 e T_2 se sono piene. Ogni volta che scenderemo su un nuovo nodo lo splitteremo se è pieno;
- se T_1 e T_2 hanno la stessa altezza, allora creiamo una nuova radice che conterrà solo la chiave k ed avrà la radice di T_1 come primo figlio e la radice di T_2 come secondo figlio;
- se T_1 ha altezza maggiore di T_2, allora scendiamo in T_1 verso destra fino ad arrivare ad un nodo che ha altezza di uno superiore rispetto all'altezza di T_2. Aggiungiamo a questo nodo la chiave k come chiave più a destra e come nuovo figlio la radice di T_2;
- se T_2 ha altezza maggiore di T_1, allora procediamo analogamente al caso precedente, ma scendendo verso sinistra.

Algorithm 1 BTreeJoin(T_1, T_2, k, t)

```
1: x_1 ← root[T_1]
2: x_2 ← root[T_2]
3: if n[x_1] = 2t − 1 then
4:    BTreeRootSplit(T_1, x_1, t)
5:    x_1 ← root[T_1]
6: end if
7: if n[x_2] = 2t − 1 then
8:    BTreeRootSplit(T_2, x_2, t)
9:    x_2 ← root[T_2]
10: end if
11: h_1 ← Heigh(x_1)
12: h_2 ← Heigh(x_2)
13: if h_1 = h_2 then
14:    return BTreeJoinEqual(T_1, T_2, k, t, x_1, x_2)
15: else
16:    if h_1 > h_2 then
17:       return BTreeJoinRight(T_1, T_2, k, t, x_1, x_2)
18:    else
19:       return BTreeJoinLeftt(T_1, T_2, k, t, x_1, x_2)
20:    end if
21: end if
```

Date: 30 Marzo 2009.

Algorithm 2 BTreeJoinEqual$(T_1, T_2, k, t, x_1, x_2)$

```
1: z ←AllocateNewNode(t)
2: n[z] ← 1
3: key_1[z] ← k
4: c_1[z] ← x_1
5: c_2[z] ← x_2
6: if n[x_1] = 1 then
7:    BTreeMerge(z, x_1, 1)
8: end if
9: if n[x_2] = 1 then
10:   BTreeMerge(z, x_2, n[z] + 1)
11: end if
12: DiskWrite(z)
13: root[T_1] ← z
14: return T_1
```

Algorithm 3 BTreeJoinRight$(T_1, T_2, k, t, x_1, x_2)$

```
1: while h_1 > h_2 + 1 do
2:    y_1 ← DiskRead c_{n[x_1]+1}[x_1]
3:    if n[y_1] = 2t − 1 then
4:       BTreeSplit(x_1, y_1, n[x_1] + 1)
5:       y_1 ← DiskRead c_{n[x_1]+1}[x_1]
6:    end if
7:    x_1 ← y_1
8:    h_1 ← h_1 − 1
9: end while
10: n[x_1] ← n[x_1] + 1
11: key_{n[x_1]}[x_1] ← k
12: c_{n[x_1]+1}[x_1] ← x_2
13: DiskWrite(x_1)
14: return T_1
```

Correttezza.

Theorem 1. BTreeJoin(T_1, T_2, k, t) *termina sempre restituendo un B-tree di grado t che contiene tutte le chiavi di T_1, tutte le chiavi di T_2 e la chiave k.*

Dimostrazione. L'algoritmo termina sempre perchè gli unici due cicli while che compaiono nelle procedure BTreeJoinLeft e BTreeJoinRight terminano in quando h_1 ed h_2 vengono opportunamente decrementati.

Sicuramente al termine l'albero restituito contiene tutte le chiavi richieste in quanto non vengono cancellate chiavi, viene aggiunta la chiave k e tutte le chiavi di T_2 o tutte le chiavi di T_1 (a seconda dei casi).

Dobbiamo dimostrare che l'albero restituito è un B-tree di grado t.

Se $h_1 = h_2$, allora l'albero restituito è un B-tree perchè:

- ha una radice contenente solamente la chiave k ed avente come figli x_1 ed x_2;
- x_1 è radice di un B-tree di grado t contentente solo chiavi minori di k;

Algorithm 4 BTreeJoinLeft$(T_1, T_2, k, t, x_1, x_2)$

```
1: while h_2 > h_1 + 1 do
2:    y_2 ← DiskRead c_1[x_2]
3:    if n[y_1] = 2t − 1 then
4:       BTreeSplit(x_2, y_2, 1)
5:       y_2 ← DiskRead c_1[x_2]
6:    end if
7:    x_2 ← y_2
8:    h_2 ← h_2 − 1
9: end while
10: n[x_2] ← n[x_2] + 1
11: for i ← n[x_2] downto 2 do
12:    key_i[x_2] ← key_{i−1}[x_2]
13: end for
14: for i ← n[x_2] + 1 downto 2 do
15:    c_i[x_2] ← c_{i−1}[x_2]
16: end for
17: key_1[x_2] ← k
18: c_1[x_2] ← x_1
19: DiskWrite(x_2)
20: return T_2
```

Algorithm 5 NodeCheck(x, C, h)

```
1: if x = NIL then
2:    if C = RED or h ≠ 0 then
3:       return FALSE
4:    else
5:       return TRUE
6:    end if
7: else
8:    if C = RED then
9:       return NodeCheck(left[x], BLACK, h − 1) and
                NodeCheck(right[x], BLACK, h − 1)
10:   else
11:      return (NodeCheck(left[x], BLACK, h − 1) or
                 NodeCheck(left[x], RED, h)) and
                (NodeCheck(right[x], BLACK, h − 1) or
                 NodeCheck(right[x], RED, h))
12:   end if
13: end if
```

- x_2 è radice di un B-tree di grado t contentente solo chiavi maggiori di k;
- se x_1 e/o x_2 hanno solo una chiave vengono fusi con z.

Se $h_1 > h_2$, allora l'albero restituito è un B-tree perchè:

- al termine del ciclo while il nodo x_1 è un nodo non pieno di altezza $h_2 + 1$;
- il nodo x_1 si trova a destra nell'albero T_1;

- al nodo x_1 viene aggiunta a destra la chiave k, che è maggiore di tutte le altre di T_1;
- al nodo x_1 viene aggiunto a destra il figlio x_2 che contiene tutte chiavi maggiori di quelle di T_1 e di k e che ha altezza h_2.

Il caso $h1 < h_2$ è analogo. □

Complessità. Per determinare h_1 ed h_2 occorre scendere dalla radice ad una foglia in T_1 e T_2, rispettivamente. Tutte le altre istruzioni richiedono un numero di operazioni limitato superiormente dall'altezza dei due alberi. Quindi abbiamo complessità $\Theta(h(T_1) + h(T_2))$ in termini di operazioni di lettura/scrittura da disco ed una complessità $O(t * (h(T_1) + h(T_2)))$ in termini di operazioni di CPU.