Algoritmi e strutture dati

Roberto Cordone

A. A. 2015-16

Capitolo 3

Implementazioni dei dizionari ordinati

Nota: queste dispense sono un rapido riassunto delle lezioni svolte nel dicembre 2015 e gennaio 2016. In buona parte ripetono gli argomenti delle dispense del prof. Goldwurm, con alcune aggiunte, qualche piccola modifica nella presentazione, qualche esempio. Chi notasse errori, incoerenze, oscurità o avesse dubbi è invitato a contattarmi per segnalarli.

3.1 Introduzione

- I dizionari ordinati rappresentano sottoinsiemi di insiemi totalmente ordinati. Rispetto ai dizionari, forniscono in più la funzione `MIN`, che restituisce il primo elemento (in base all'ordinamento totale) del sottoinsieme rappresentato.
- Un'implementazione banale potrebbe usare una lista con in aggiunta un puntatore al primo elemento (non è necessario mantenere ordinata l'intera lista). Ma si è già detto che le funzioni `MEMBER` e `DELETE` sono inefficienti sulle liste. Per di più, la funzione `DELETE` dovrebbe gestire anche l'aggiornamento del puntatore al primo elemento (banale per la `INSERT`).
- Un'implementazione semplice potrebbe usare una tabella hash, in cui la funzione `MIN` si limiti a scorrere la tabella. Tutte le funzioni avrebbero la complessità già nota, ma la funzione `MIN` avrebbe una complessità pessima, $O(m)$, proporzionale al numero di caselle della tabella hash, anche per dizionari praticamente vuoti. Questo sarebbe molto peggio che usare una lista. Se la funzione `MIN` è di uso frequente, questa implementazione non va affatto bene.

3.2 Alberi binari di ricerca

- Un albero binario si definisce come
$$T = \begin{cases} \emptyset \text{ se è vuoto} \\ (a, T_s, T_d) \text{ altrimenti} \end{cases}$$
dove $a \in A$ racchiude l'informazione e T_s e T_d sono alberi binari detti *sottoalbero sinistro* e *sottoalbero destro*. La terna $v = (a, T_s, T_d)$ si dice anche *nodo* e in genere contiene anche la chiave $k(a)$ per consentire confronti veloci.
- In un *albero di ricerca* le informazioni a_v legate ai nodi appartengono a un insieme A totalmente ordinato e la loro distribuzione rispetta le seguenti proprietà:

1. esiste uno e un solo nodo $v \in T$ tale che $a_v = a$ per ogni elemento a del dizionario;
2. per ogni $v \in T$ e per ogni $u \in T_s(v)$, è $a_u \prec a_v$;
3. per ogni $v \in T$ e per ogni $u \in T_d(v)$, è $a_u \succ a_v$.

- Un'implementazione diretta è quella con puntatori:

```
struct _elemento {
  A *a;
  int E;
  struct _elemento *Ts;
  struct _elemento *Td;
  struct _elemento *father;
};

typedef struct _elemento *node;
typedef struct _elemento *tree;

tree T;
```

  dove un albero `T` è visto come un puntatore al nodo radice, e i nodi contengono un puntatore `a` all'informazione di tipo `A`, una chiave intera `E`, un sottoalbero sinistro, uno destro e un puntatore `father` che punta il nodo padre e serve a rendere efficiente la funzione `DELETE`.

- Una rappresentazione alternativa, equivalente, usa un vettore di strutture con gli stessi campi, dove i puntatori sono sostituiti da indici numerici.

```
struct _elemento {
  A *a;
  int E;
  int Ts;
  int Td;
  int father;
};

typedef int node;
typedef struct _elemento *tree;

tree T;
```

- Esercizio: costruire la rappresentazione a vettore dell'albero in Figura 3.1.
- Nel seguito useremo la rappresentazione a puntatori. È un buon esercizio tradurre le funzioni nella rappresentazione a vettore, oppure creare un'interfaccia che renda l'implementazione effettivamente usata invisibile all'esterno, attraverso funzioni che accedano in lettura e scrittura alle singole informazioni (per esempio, `SIN(T)` e `DES(T)` per le radici dei due sottoalberi, `ROOT(T)` per il nodo radice, `E(v)` per la chiave dell'elemento `v`, ecc...
- Si definisce la funzione $\emptyset$ che crea un albero vuoto come `return NULL;` (nel seguito, scriveremo direttamente `T = NULL;`).

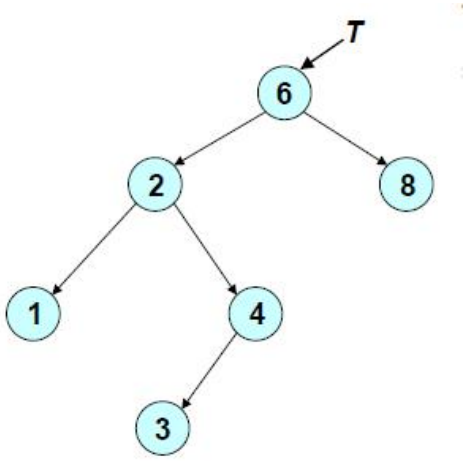

Figura 3.1: Esempio di albero binario di ricerca

- La visita "in ordine" dell'albero riproduce l'ordine totale del dizionario

```
void INORDER(tree T)
{
  if (T != NULL)
  {
    INORDER(T->Ts);
    PRINT(T->a);
    INORDER(T->Td);
  }
}
```

  Considerati due elementi qualsiasi a_i e a_j con $a_i \prec a_j$, ci sono cinque casi:

  1. $a_i \in Ts(T)$ e $a_j \in Td(T)$;
  2. $a_i \in Ts(T)$ e $a_j = a_{ROOT(T)}$;
  3. $a_i = a_{ROOT(T)}$ e $a_j \in Td(T)$;
  4. $a_i, a_j \in Ts(T)$;
  5. $a_i, a_j \in Td(T)$.

  I primi tre casi implicano che a_i viene stampato prima di a_j, gli ultimi due non sono risolutivi, ma indicano che si può studiare un solo sottoalbero. Prima o poi, si ricadrà certamente in uno dei primi tre casi. Quindi, la visita "in ordine" considera gli elementi nell'ordine totale predefinito.

- La funzione `MIN` sfrutta questa proprietà: il primo elemento è il primo a venir stampato. Non sta certamente nel sottoalbero destro il fatto che l'elemento minimo certamente non sta in alcun sottoalbero destro e non ha sottoalbero destro, per cui si può trovare scorrendo i sottoalberi sinistri finché ce ne sono:

```
A *MIN(tree T)
{
  if (T == NULL)
    return NULL;
```

```
  else if (T->Ts == NULL)
    return T->a;
  else
    return MIN(T->Ts);
}
```

Ne esiste una semplice versione iterativa:

```
A *MIN(tree T)
{
  if (T != NULL)
    while (T->Ts != NULL)
      T = T->Ts;

  return T;
}
```

che funziona anche nel caso di albero vuoto (restituisce `NULL`). Ovviamente, è possibile realizzare una funzione `MAX` sfruttando allo stesso modo i sottoalberi destri.

- Esercizio: si esegua la funzione `MIN` sull'albero in Figura 3.1.
- La funzione `MEMBER(a,T)` valuta la radice dell'albero; se l'elemento cercato ha chiave inferiore, lo cerca nel sottoalbero sinistro, se superiore nel sottoalbero destro. Mostriamo una funzione più sofisticata, `SEARCH(a,T)`, che restituisce `NULL` se l'elemento non appartiene al dizionario e un puntatore ad esso se vi appartiene.

```
A *SEARCH (A *a, tree T)
{
  if (T == NULL) return NULL;
  k = KEY(a);
  if (k == T->E;)
    return T->a;
  else if (k < T->E)
    return SEARCH(a,T->Ts);
  else /* (k > T->E) */
    return SEARCH(a,T->Td);
}
```

- Esercizio: si esegua la funzione `SEARCH` sull'albero in Figura 3.1 cercando gli elementi 5 e 8.
- La funzione `INSERT(a,T)` deve preservare la struttura binaria dell'albero e l'ordinamento delle etichette. A tale scopo, si deve cercare l'elemento nel dizionario: se lo si trova, non bisogna fare nulla; se invece non si trova, l'ultimo elemento visitato è utile per l'inserimento. Infatti:
  1. la ricerca si ferma in un sottoalbero vuoto;
  2. se l'ultima radice considerata aveva chiave superiore all'elemento da inserire, si è fermata nel sottoalbero sinistro;

3. se l'ultima radice considerata aveva chiave inferiore all'elemento da inserire, si è fermata nel sottoalbero destro.

In entrambi i casi, il sottoalbero vuoto è la posizione corretta per il nuovo nodo, perché ad ogni passo siamo scesi nel sottoalbero corretto per garantire che il nuovo nodo e i suoi antenati soddisfino le proprietà di ordinamento richieste sulle etichette. Per inserire il nuovo elemento, basta creare un nuovo nodo e appenderlo in quella posizione. Ovviamente, bisogna considerare a parte il caso di albero vuoto.

```
tree INSERT(A *a, tree T)
{
  if (T == NULL) T = CREATE_TREE(a);
  k = KEY(a);
  r = ROOT(T);
  if (k < r->E)
  {
    if (r->Ts == NULL)
      APPEND_SINISTRA(a,r);
    else
      INSERT(a,r->Ts);
  }
  else /* (k > r->E) */
  {
    if (T->Td == NULL)
      APPEND_DESTRA(a,r);
    else
      INSERT(a,r->Td);
  }
  return T;
}
```

- Le tre funzioni ausiliarie sono estremamente simili: tutte consistono nel creare un nodo associato all'elemento a, con sottoalberi vuoti.
  - `CREATE_TREE` lo restituisce come albero isolato;
  - `APPEND_SINISTRA` lo appende come sottoalbero sinistro a un nodo dato;
  - `APPEND_DESTRA` lo appende come sottoalbero destro a un nodo dato.

```
tree CREATE_TREE (A *a)
{
  r = malloc(sizeof(node));
  r->a = a;
  r->E = KEY(a);
  r->Ts = NULL;
  r->Td = NULL;
  r->father = NULL;
  return r;
}


void APPEND_SINISTRA (A *a, nodo v)
{
```

```
  T = CREATE_TREE(a);
  T->father = v;
  v->Ts = T;
}


void APPEND_DESTRA (A *a, nodo v)
{
  T = CREATE_TREE(a);
  T->father = v;
  v->Td = T;
}
```

- La funzione `DELETE(a,T)` si basa anch'essa, ovviamente sulla ricerca, dato che per cancellare un elemento bisogna sapere dov'è. Rispetto all'inserimento, presenta alcuni casi complicati.
  - è molto semplice se l'elemento è una foglia: basta aggiornare il puntatore ad esso nel nodo padre;
  - è piuttosto semplice se l'elemento ha un solo sottoalbero, perché si può collegare direttamente tale sottoalbero al nodo padre, cortocircuitando il nodo cancellato;
  - è complessa se il nodo ha due sottoalberi, ma si può sfruttare una proprietà: *il predecessore di un nodo con due figli non ha sottoalbero destro*, dato che in tale sottoalbero starebbero nodi compresi fra lui e il nodo con due figli, contraddicendo l'ipotesi. Allora si può cancellare tale predecessore come nel caso precedente e sostituirlo al nodo che va effettivamente cancellato.

```
tree DELETE (A *a, tree T)
{
  r = SEARCH(a,T);
  if (r != NULL)
  {
    if ( (r->Ts == NULL) || (r->Td == NULL) ) /* 0 o 1 sottoalbero */
      DELETE_NODE(r);                         /* 0 cancella la radice */
    else
    {
      v = NODE_MAX(r->Ts);
      COPY_NODE(v,r);
      DELETE_NODE(v);
    }
  }
  return T;
}
```

- È un buon esercizio scrivere la funzione `DELETE` senza usare esplicitamente la funzione `SEARCH`. Semplicemente, si tratta di applicare le stesse chiamate ricorsive:

```
tree DELETE (A *a, tree T)
```

```
{
  if (T == NULL) return NULL;

  k = KEY(a);
  r = ROOT(T);
  if ( (k < r->E) && (r->Ts != NULL) )
    DELETE(a,r->Ts);   /* prosegue la ricerca a sinistra */
  else if ( (k > r->E) && (r->Td != NULL) )
    DELETE(a,r->Td);   /* prosegue la ricerca a destra */
  else if (k == r->E)  /* elemento trovato */
  {
    if ( (r->Ts == NULL) || (r->Td == NULL) ) /* 0 o 1 sottoalbero */
      DELETE_NODE(r);                         /* 0 cancella la radice */
    else
    {
      v = NODE_MAX(T->Ts);
      COPY_NODE(v,r);
      DELETE_NODE(v,T);
    }
  }
  return T;
}
```

- La funzione ausiliaria `DELETE_NODE(v)` cancella un nodo, sotto l'ipotesi che esso abbia almeno un sottoalbero vuoto. Le prime due istruzioni individuano il nodo padre e il sottoalbero figlio, le altre due aggiornano i puntatori di padre e figlio per agganciarli direttamente fra loro.

```
void DELETE_NODE (node v, tree T)
{
  fv = v->father;
  if (v->Ts != NULL)
    ST = v->Ts;
  else
    ST = v->Td;

  if (T == fv->Ts)
    fv->Ts = ST;
  else
    fv->Td = ST;
  ST->father = fv;

  free(v);
}
```

- La funzione `NODE_MAX` non è altro che una variante della funzione `MAX` che restituisce il nodo contenente l'elemento massimo di un albero, anziché l'informazione. Infine, la funzione `COPY_NODE` copia i campi informazione del primo nodo (`a` e `E`) nei corrispondenti campi del secondo nodo (nota: non scriviamo `v2 = v;` perché `v2` e `v` sono puntatori o indici nelle due rappresentazioni sopra presentate).

```
A *MIN(tree T)
```

```
{
  if (T == NULL)
    return NULL;
  else if (T->Ts == NULL)
    return T->a;
  else
    return MIN(T->Ts);
}
```

- Tutte queste operazioni hanno una complessità pari a quella della ricerca, con eventualmente un numero costante di operazioni in più per gestire inserimenti, copie, cancellazioni. La ricerca dipende dalla profondità h dell'albero. Questa può variare fra due casi estremi:
  - albero perfettamente bilanciato: $h = \log_2 n$;
  - albero perfettamente sbilanciato: $h = n$.
- In particolare, inserendo elementi in ordine si ottiene un albero completamente sbilanciato (verso destra se gli elementi sono in ordine crescente, verso sinistra se in ordine decrescente). Un altro esempio molto sbilanciato si ottiene inserendo i valori 1, 2n, 2, 2n-1, 3, 2n-2 ... n, n+1 in quest'ordine.
- Nel caso medio, però, le cose vanno molto meglio. Se tutti gli ordinamenti sono equiprobabili e inseriamo n elementi, vediamo che ciascun elemento a_k del dizionario ($k = 1, \ldots, n$) ha probabilità $1/n$ di essere inserito per primo, e quindi di finire nella radice e restarci. Tale elemento subisce $n-1$ confronti durante il programma. Se indichiamo con T_n il numero medio di confronti richiesto per inserire gli n elementi nell'intero albero, i confronti richiesti nel sottoalbero di sinistra saranno T_{k-1} e quelli nel sottoalbero di destra T_{n-k}. Quindi il numero medio di confronti richiesto per costruire l'albero è:

$$T_n = \frac{1}{n} \sum_{k=1}^{n} (n - 1 + T_{k-1} + T_{n-k}) =$$

- Questa espressione è la stessa che si ritrova nello studio di complessità nel caso medio dell'algoritmo QuickSort. Infatti, tale algoritmo consiste nella chiamata di una procedura di partizione che scorre $n-1$ elementi per dividerli in due sottovettori e nella chiamata ricorsiva dell'algoritmo sul primo sottovettore (di $k-1$ elementi) e sul secondo (di $n-k$ elementi), dove k è l'indice dell'elemento pivot, non noto a priori, che si ipotizza abbia distribuzione uniforme fra 1 e n, dato che si ipotizzano equiprobabili tutte le permutazioni di elementi.
- Lo studio dell'equazione attraversa i seguenti passi:
  1. $\sum_{k=1}^{n} (n-1) = n\,(n-1)$;
  2. $\sum_{k=1}^{n} T_{k-1} = T_0 + T_1 + \ldots + T_{n-1} = \sum_{k=0}^{n-1} T_k$;
  3. $\sum_{k=1}^{n} T_{n-k} = T_{n-1} + T_{n-2} + \ldots + T_0 = \sum_{k=0}^{n-1} T_k$;
  4. Quindi $T_n = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} T_k$;

5. $T_{n-1} = n - 2 + \dfrac{2}{n-1} \sum_{k=0}^{n-2} T_k$;
6. Combiniamo $nT_n - (n-1)\,T_{n-1}$:

$$nT_n - (n-1)\,T_{n-1} = n\,(n-1) + 2\sum_{k=0}^{n-1} T_k - (n-1)\,(n-2) - 2\sum_{k=0}^{n-2} T_k$$

7. Raccogliamo separatamente i termini polinomiali e le sommatorie:

$$\begin{cases} n\,(n-1) - (n-1)\,(n-2) = 2n-2 \\ 2\sum_{k=0}^{n-1} T_k - 2\sum_{k=0}^{n-2} T_k = 2T_{n-1} \end{cases}$$

8. Riaccorpiamo i due pezzi:

$$nT_n - (n-1)\,T_{n-1} = 2n - 2 + 2T_{n-1} \Rightarrow nT_n = (n+1)\,T_{n-1} + 2\,(n-1)$$

9. Introduciamo una sostituzione: $u_n = \dfrac{T_n}{n+1}$,
   da cui $T_n = (n+1)\,u_n$ e $T_{n-1} = nu_{n-1}$
10. Quindi l'equazione ricorrente $nT_n = (n+1)\,T_{n-1} + 2\,(n-1)$ diventa

$$n\,(n+1)\,u_n = (n+1)\,nu_{n-1} + 2\,(n-1) \Rightarrow u_n = u_{n-1} + \frac{2}{n}\frac{n-1}{n+1}$$

11. È possibile calcolare una soluzione esatta (vedi Cormen), ma ci "accontentiamo" di osservare che $n - 1 < n + 1$, per cui

$$u_n < u_{n-1} + \frac{2}{n}$$

12. Ciò implica che $u_{n-1} < u_{n-2} + \dfrac{2}{n-1}$, $u_{n-2} < u_{n-3} + \dfrac{2}{n-2}$, ecc...
13. Quindi

$$u_n < u_{n-1} + \frac{2}{n} < u_{n-2} + \frac{2}{n-1} + \frac{2}{n} < u_{n-3} + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} < \ldots < u_0 + \sum_{k=1}^{n} \frac{2}{k}$$

14. Infine, $u_0 = T_0/\,(n+1) = 0$, perché l'inserimento in un albero vuoto non comporta confronti, e la somma è la classica somma armonica.

$$T_n < (n+1)\log n$$

- In conclusione, costruire un dizionario di n elementi comporta $O\,(n \log n)$ operazioni di confronto, cioè mediamente $O\,(\log n)$ operazioni per ogni inserimento.