Tutoraggio Informatica Generale
Inserimento e cancellazione in un ABR
Counting Sort

A.Festa
festa@mat.uniroma1.it

20-5-2010

1 Inserimento e cancellazione in ABR

Quando si inserisce o si rimuove un elemento la struttura dell'albero cambia. L'albero modificato deve mantenere le proprietà di un albero binario di ricerca.

La struttura dell'albero varia a seconda della sequenza di dati da inserire o rimuovere. L'inserimento risulta essere un'operazione immediata mentre la rimozione di un elemento è più complicata, proprio perché bisogna essere certi che l'albero rimanga un albero binario di ricerca.

Per inserire z si usano due puntatori q e t. Il puntatore a q scende l'albero , mentre t punta al padre di q. Nel ciclo while i due puntatori (q e t) scendono l'albero. q scende al figlio sinistro o destro a seconda dell'esito del confronto di z con q.info. Ci si ferma quando q = NIL e q occupa la posizione in cui z verrà inserito. Il tempo necessario per l'inserimento è O(h), ossia non più del cammino massimo tra la radice e una foglia (cioè h l'altezza).

```
void ins(nodo *q, int z)
{nodo *t;
  while (q)
    {t=q;
     if (q->info>z)  //cerco il posto giusto
         q=q->fs;    //dove effettuare l'inserimento
     else q=q->fd;
    }
           // sono arrivato ad una foglia dell'albero.
           // t conserva l'indirizzo di tale foglia.

    if (t->info>z)
```

```
       {t->fs=new nodo;
        t->fs->info=z;     //inserisco l'elemento
        t->fs->pd=t;
        t->fs->fs=t->fs->fd=NULL;
       }
    else {t->fd=new nodo;
          t->fd->info=z;
          t->fd->pd=t;
          t->fd->fs=t->fd->fd=NULL;
         }
}
```

1.1 Cancellazione con puntatore al padre

Nella cancellazione ci sono tre casi:

- Se p non ha figli, allora si modifica p->pd che punta non più a p, ma a NULL.
- Se p ha un unico figlio, allora si taglia fuori p dall'albero, facendo puntare p->pd all'unico figlio di p.
- Se p ha due figli, allora si individua il successore, ossia il minimo del suo sottoalbero destro. Il successore y ha nessun figlio o 1 figlio. Quindi y prende il posto di p, riconducendosi al caso 1 e 2. Alla fine i dati in y vengono copiati in p.

(Provate a scivere il codice).

1.2 Cancellazione senza puntatore al padre

Questo è il codice commentato.

Sono contemplati i molti casi possibili.

Bisogna tenere a mente che nella cancellazione l'ABR che otteniamo *non è univocamente determinato*, due algoritmi di cancellazione differenti possono restituire alberi differenti, l'importante è che sia avvenuta la cancellazione e che il risultato sia ancora un ABR.

```
nodo *elimina(nodo *q,int x)
{if(q->info==x&&!q->fs&&!q->fd)
           return NULL;   //nel caso l'ABR sia costituito
 nodo *pd=q,*p=q;         //da un unico nodo da eliminare ritorno a NULL
 while(p&&p->info!=x)     //cerco un nodo contenente x
   {pd=p;                 //mantenendo un puntatore al padre
```

```
   if(p->info>x) p=p->fs;
     else p=p->fd;       //a questo punto ho p che punta il nodo
  }                      //da eliminare e pd al padre di p
if(!p) return q;         //se x non c'è nell'albero non faccio
                         //nulla e ritorno la root
if(p!=pd)
   {if(!p->fs)            //Se p ha un unico figlio,
    {if(p->info<pd->info) //allora si taglia fuori p dall'albero,
    pd->fs=p->fd;         //facendo puntare pd (dalla parte giusta)
     else pd->fd=p->fd;   //all'unico figlio di p.
     return q;            //è compreso anche il caso
    }                     //in cui p non ha figli
    if(!p->fd)
    {if(p->info<pd->info) pd->fs=p->fs;
     else pd->fd=p->fs;
     return q;
    }
   }
if(p->fs)             //ho entrambi i figli del nodo che devo eliminare
  {pd=p->fs;
   if(!pd->fd)           //se il figlio sx di p non ha fd lo
     {p->info=pd->info;  //metto al posto del nodo eliminato
      p->fs=pd->fs;      //comprendendo anche sottoalbero sx
      return q;
     }
   while(pd->fd->fd)   //altrimenti cerco il max del sottoalbero sx
      pd=pd->fd;       //che ha come root il figlio sx
   p->info=pd->fd->info;  //una volta trovato lo sostituisco
   pd->fd=pd->fd->fs;     //al nodo da eliminare, e "tappo"
   return q;              //il buco lasciato nel s.a. sx
  }
   else          //se il figlio sx di p ha fd guardo nel fd di p
     {pd=p->fd;
      if(!pd->fs)   //se tal nodo non ha fs lo sostituisco in p
{p->info=pd->info;
        p->fd=pd->fd;
        return q;
}
      while(pd->fs->fs)  //altrimenti cerco il min del s.a. dx
         pd=pd->fs;      //ed effettuo la stessa operazione di
      p->info=pd->fs->info; //sostituzione e "tappo buchi"
      pd->fs=pd->fs->fd;
      return q;
     }
```

```
}
```

2 Counting sort

Il Counting sort è un algoritmo di ordinamento per valori numerici interi con complessità lineare. L'algoritmo si basa sulla conoscenza a priori dell'intervallo in cui sono compresi i valori da ordinare.

L'algoritmo conta il numero di occorrenze di ciascun valore presente nell'array da ordinare, memorizzando questa informazione in un array temporaneo di dimensione pari all'intervallo di valori. Il numero di ripetizioni dei valori inferiori indica la posizione del valore immediatamente successivo.

Si calcolano i valori massimo, max(A), e minimo, min(A), dell'array e si prepara un array ausiliario C di dimensione pari all'intervallo dei valori con C[i] che rappresenta la frequenza dell'elemento i + min(A) nell'array di partenza A. Si visita l'array A aumentando l'elemento di C corrispondente. Dopo si visita l'array C in ordine e si scrivono su A, C[i] copie del valore i + min(A).

L'algoritmo esegue tre iterazioni, due di lunghezza n (pari alla lunghezza dell'array da ordinare) per l'individuazione di max(A) e min(A) e per il calcolo delle occorrenze dei valori, e una di lunghezza k (pari a max(A) - min(A) + 1) per l'impostazione delle posizioni finali dei valori: la complessità totale è quindi O(n + k).

Non è basato su confronti e scambi e conviene utilizzarlo quando il valore di k è O(n), nel qual caso l'algoritmo è O(n), altrimenti risulterebbero più veloci altri algoritmi.

```
void countingSort(int A[],int n)
{
  int i, min, max;

 //Calcolo degli elementi max e min
 max =  A[0];
 min = A[0];
 for (i = 0 ; i<n-1;i++) {
    if (A[i] > max)
       max = A[i];
    else if(A[i] < min)
       min = A[i] ;  }

    int m=max-min;
```

```
      int t;
     int C[m];
   for (i = 0 ; i<=m;i++)
      C[i]=0;
      C[m] = C[m] + 1;
   for (i = 0 ; i<n-1;i++)
      {t=A[i] - min;
      C[t] = C[t] + 1;  }

       //aumenta il numero di volte che si è incontrato il valore
       //Ordinamento in base al contenuto dell'array delle frequenze C

   int k = 0  ;


   for (i = 0 ; i<=m; i++)  {
      while (C[i] > 0) {        //scrive C[i] volte il valore
         A[k] = i + min ;      //(i + min) nell'array A
         k = k + 1;
         C[i] = C[i] - 1;
                     }
         }
}
```