ALGORITMI E STRUTTURE DATI

Esercitazioni

AndreA Orlandini

http://www.dia.uniroma3.it/~orlandin/asd/

e-mail: orlandin@dia.uniroma3.it

Orario di ricevimento: Martedì 14.00 - 16.00

Puntatori e Alberi, ABR

StudentiDIA Forum

http://forum.studentidia.org/

Dove trovare i lucidi presentati a lezione

http://limongelli.dia.uniroma3.it/asd/

Alberi

Un albero è un insieme di oggetti, chiamati nodi, su cui è definita una relazione binaria $G(x, y)$ – che leggiamo "x è genitore di y" – tale che:

1. esiste un unico nodo, chiamato radice, che non ha genitori;
2. ogni nodo x diverso dalla radice ha uno ed unico genitore;
3. per ogni nodo x diverso dalla radice esiste un cammino dalla radice a x (l'albero è *connesso*): esistono nodi $x_1, .., x_k$ $(k \geq 1)$ tali che x_1 è la radice dell'albero, $x_k = x$ e per ogni $i = 1, ..., k-1$ x_i è genitore di x_{i+1}.

- Se x è il genitore di y, allora y è un figlio di x.
- Un albero binario è un albero in cui ogni nodo ha al massimo due figli.

Terminologia di base

cammino: sequenza di nodi $x_1, ..., x_k, x_{k+1}$ tale che, per $i = 1, ..., k$, x_i è il genitore di x_{i+1};

lunghezza del cammino: k (numero degli archi);

antenato e discendente: dato un nodo x di un albero T con radice r, qualunque nodo y sul cammino (unico) da r a x è un antenato di x, mentre x è un discendente di y;

fratelli: nodi che hanno lo stesso genitore;

sottoalbero: insieme costituito da un nodo x e tutti i suoi discendenti; x è la radice del sottoalbero;

foglia: nodo senza figli;

nodo interno: nodo con uno o più figli;

grado: il numero di figli di un nodo x è il grado di x;

Profondità o Livello :

- La radice ha profondità 0
- Se un nodo ha profondità k, i suoi figli hanno profondità k+1

Altezza dell'albero : massima profondità a cui si trova una foglia ovvero numero di archi nel cammino dalla radice alla foglia più profonda

Operazioni di base

- Empty(): restituisce una struttura rappresentante l'albero vuoto
- Null(T): test albero vuoto.
- Root(T): restituisce il puntatore alla radice dell'albero; un errore se T è vuoto.
- Left(T): restituisce il sottoalbero sinistro di T; un errore se T è vuoto.
- Right(T): restituisce il sottoalbero destro di T; un errore se T è vuoto.
- Left-Child(T): restituisce il puntatore al figlio sinistro; un errore se T è vuoto.
- Right-Child(T): restituisce il puntatore al figlio destro; un errore se T è vuoto.
- Mktree(x,T1,T2): restituisce l'albero con radice x e sottoalberi T1 e T2.

Rappresentazione di alcune operazioni di base

```
EMPTY()
1. ;; eventuale allocazione
   ;; di memoria per T
2. root[T] <- NIL
3. return T
```

```
NULL(T)
1. return (root[T] = NIL)
```

```
ROOT(T)
1. if NULL(T)
2.    then error "albero vuoto"
3.    else return root[T]
```

```
LEFT(T)
1. if NULL(T) then error "albero vuoto"
2.    else ;; eventuale allocazione di memoria per T1
3.         root[T1] <- left[root[T]]
4.         return T1
```

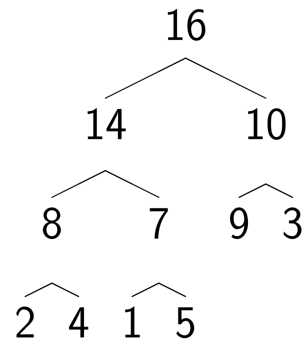
```
RIGHT(T)
1. if NULL(T) then error "albero vuoto"
2.    else ;; eventuale allocazione di memoria per T1
3.         root[T1] <- right[root[T]]
4.         return T1
```

```
LEFT-CHILD(T)
1. if NULL(T)
2.     then error "albero vuoto"
3.     else return left[root[T]]

MK-TREE(x,T1,T2)
1. ;; eventuale allocazione di memoria per T
2. root[T] <- x
3. left[x] <- root[T1]
4. right[x] <- root[T2]
5. return T
```

Operazioni sul tipo nodo

```
NKEY(x)
1. return key[x]

NLEFT(x)
1. return left[x]

NRIGHT(x)
1. return right[x]

SET-KEY(x,k)
1. key[x] <- k

SET-LEFT(x,y)
1. left[x] <- y

SET-RIGHT(x,y)
1. right[x] <-y
```

Diversi modi per visitare gli alberi



preordine (profondità)

visita il padre, visita il s.a. sinistro, visita il s.a. destro;

16,14,8,2,4,7,1,5,10,9,3

postordine (profondità)

visita il s.a. sinistro, visita il s.a. destro, visita il padre;

2,4,8,1,5,7,14,9,3,10,16

simmetrica (profondità)

visita il s.a. sinistro, visita il padre, visita il s.a. destro;

2,8,4,14,1,7,5,16,9,10,3

per livelli (ampiezza)

16,14,10,8,7,9,3,2,4,1,5

Visita in preordine

Versione ricorsiva:

```
PREORDER(T)                              SE VOGLIAMO ASTRARRE DAL TIPO NODO
1. if not NULL(T)
2.    then visita ROOT(T)
3.         PREORDER(LEFT(T))
4.         PREORDER(RIGHT(T))
```

oppure

```
PREORDER'(T')                            T SI RIFERISCE AD UN TIPO PRECISO DI NODO
1. if T'!= NIL
2.    then visita T'
3.         PREORDER(left[T'])
4.         PREORDER(right[T'])

T' <- ROOT(T)                            T' PUNTATORE IN LINGUAGGIO C
```

Visita in ampiezza

Per memorizzare i puntatori ai figli (non vuoti) dei nodi che via via vengono visitati, si usa una coda.

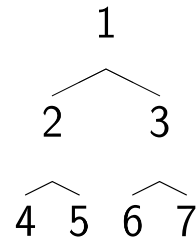
```
BREADTH-TREE(T)
1. EMPTY(Q)
2. if not NULL(T)
3.    then ENQUEUE(Q,ROOT(T))
4. while not EMPTY-QUEUE(Q)
5.       do x <- DEQUEUE(Q)
6.          visita x
7.          if NLEFT(x) != NIL
8.             then ENQUEUE(Q,NLEFT(x))
9           if NRIGHT(x) != NIL
10.            then ENQUEUE(Q,NRIGHT(x))
```

Esercizi

Scrivere programmi che realizzino le operazioni seguenti sugli alberi binari:

1. NODE(x): riporta l'albero costituito dall'unico nodo x.
2. IS-NODE(T): predicato che determina se l'albero è costituito da un unico nodo.
3. REMOVE-LEFT(T): riporta l'albero che si ottiene da T eliminando il primo sottoalbero, se esiste; un errore altrimenti.
4. INSERT-LEFT(T,TLEFT): riporta l'albero che si ottiene aggiungendo a T l'albero TLEFT come sottoalbero sinistro. Il programma riporta un errore se T è vuoto o se ha già il sottoalbero sinistro.
5. SEARCH(T,k): riporta il puntatore a un nodo di T con chiave k, se esiste, NIL altrimenti.
6. DELETE-SUBTREE(T,x): elimina da T il sottoalbero con radice x, se esiste. Altrimenti T rimane inalterato.
7. POSTORDER(T): visita in post-ordine dell'albero con radice T.
8. SIMM(T): visita simmetrica dell'albero con radice T.
9. Specializzare le visite dell'albero in modo tale che ogni nodo visitato venga memorizzato in una lista. Utilizzare le funzioni definite sulle liste.
10. H(T): riporta l'altezza dell'albero con radice T. Si ricordi che un albero costituito da un solo nodo ha altezza 0.

- Utilizziamo una rappresentazione GRAFICA per gli alberi



- Esiste anche una notazione testuale detta PARENTETICA

```
( 1
    ( 2
        ( 4 () () )
        ( 5 () () ) )
    ( 3
        ( 6 () () )
        ( 7 () () ) )
)
```

- Un nodo n che non ha figli è rappresentato da

```
( n () () )
```

Rappresentazione di Alberi Binari in C mediante puntatori

- Utilizziamo la solita tecnica di rappresentazione

```
typedef ... TipoInfoAlbero;

struct StructAlbero {
  TipoInfoAlbero info;
  struct StructAlbero *destro, *sinistro;
};

typedef struct StructAlbero TipoNodoAlbero;

typedef TipoNodoAlbero *TipoAlbero;
```

- Un puntatore nullo rappresenta un albero vuoto, mentre un puntatore ad un nodo rappresenta un albero che ha come radice l'elemento puntato
- Vediamo come creare un albero a partire da una sua rappresentazione parentetica

Creazione di Alberi letti da file

- Creazione di un albero binario a partire dalla rappresentazione parentetica letta da file

```
TipoAlbero LeggiAlbero(char *nome_file)
{
  TipoAlbero result;
  FILE *file_albero;

  file_albero = fopen(nome_file, "r");
  assert(file_albero);

  result = LeggiSottoAlbero(file_albero);

  fclose(file_albero);
  return result;
}
```

- Leggo sul file la struttura dell'albero tramite la funzione LeggiSottoAlbero

```
char LeggiParentesi(FILE *f) { /* funz. per leggere gli  spazi tra le () */
  char c = fgetc(f);

  while (c != '(' && c != ')') {c = fgetc(f);}
  return c;
}
```

- Questa funzione esegue la costruzione vera e propria

```
TipoAlbero LeggiSottoAlbero(FILE *file_albero)
{
  char c; TipoInfoAlbero i; TipoAlbero r;

  c = LeggiParentesi(file_albero);
  c = fgetc(file_albero);

  if (c == ')')
    return NULL;
  else {
    fscanf(file_albero, "%d", &i);

    r = (TipoAlbero) malloc(sizeof(TipoNodoAlbero));
    r->info = i;

    r->sinistro = LeggiSottoAlbero(file_albero);
    r->destro = LeggiSottoAlbero(file_albero);

    c = LeggiParentesi(file_albero);
    return r;
  }
}
```

- Stampa in forma parentetica non indentata

```
void StampaAlbero(TipoAlbero albero)
{
  if (albero == NULL) {
    printf("()");
    return;
  }
  printf("( %d ", albero->info);
  StampaAlbero(albero->sinistro);
  StampaAlbero(albero->destro);
  printf(" )");
}
```

- Esempio di main

```
main()
{
  char nomefile[80];
  TipoAlbero tree;

  strcpy(nomefile,"alb.par");
  tree = LeggiAlbero(nomefile);
  StampaAlbero(tree);
  printf("\n");
}
```

- il file di input ALB.PAR

```
( 1
    ( 2
        ( 4 () () )
        ( 5 () () )
     )
    ( 3
        ( 6 () () )
        ( 7 () () )
     )
)
```

- Il risultato della funzione

```
( 1 ( 2 ( 4 ()() )( 5 ()() ) ) ( 3 ( 6 ()() )( 7 ()() ) ) )
```

- Riproduco la rappresentazione parentetica
- La stampa può risultare poco chiara

- Stampa in forma parentetica indentata

```
void StampaLivello(TipoAlbero albero, int livello)
{
  if (albero == NULL) {
    Indenta(livello); /* 2 blank x livello */
    printf("()\n");
    return;
  }
  Indenta(livello);
  printf("( ");

  printf("%d\n", albero->info);
  StampaLivello(albero->sinistro, livello + 1);
  StampaLivello(albero->destro, livello + 1);

  Indenta(livello);
  printf(")\n");
}

void StampaAlberoIndentato(TipoAlbero albero)
{
  putchar('\n');
  StampaLivello(albero, 0);
}
```

- Nella funzione main cambio solo la funzione di stampa

```
main()
{
  char nomefile[80];
  TipoAlbero tree;

  strcpy(nomefile,"alb.par");
  tree = LeggiAlberoIndentato(nomefile);
  StampaAlbero(tree);
  printf("\n");
}
```

- il file di input è lo stesso e ottengo come risultato di esecuzione

```
( 1
  ( 2
    ( 4
      ()
      ()
    )
    ( 5
      ()
      ()
    .
    .
```

- Calcolare la profondità di un albero

```
int Massimo(int a, int b);

int Profondita(TipoAlbero albero)
{
  int s, d;

  if (albero == NULL)
    return -1;
  else {
    s = Profondita(albero->sinistro);
    d = Profondita(albero->destro);

    return (Massimo(s, d) + 1);
  }
}
```

- Test di presenza di un nodo

```
bool Presente(TipoAlbero albero, int elem)
{
  if (albero == NULL)
    return FALSE;
  else if (albero->info == elem) {
    return TRUE;
  } else {
    return (Presente(albero->sinistro, elem) || Presente(albero->destro, elem));
  }
}
```

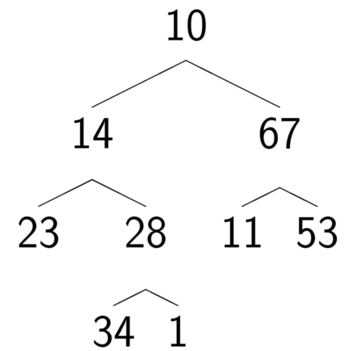
- Stampa Foglie

```
void StampaFoglie(TipoAlbero albero) {
  if (albero != NULL) {
    if (isFoglia(albero)) stampaNodo(albero->info);
    StampaFoglie(albero->sinistro);
    StampaFoglie(albero->destro);
  }
}

bool isFoglia(TipoAlbero) {
  return ((albero->sinistro == NULL) && (albero->destro == NULL))
}
```

- Test della proprietà di albero crescente. Un albero è crescente se:
  a) è vuoto
  b) esiste un cammino radice-foglia con valori sempre crescenti

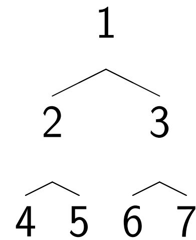
```
bool Crescente(TipoAlbero albero)
{

  if (albero == NULL)  return TRUE;
  if (albero->sinistro == NULL && albero->destro == NULL)
      return TRUE;

  if (albero->sinistro != NULL) {
    if ((albero->info < albero->sinistro->info) && Crescente(albero->sinistro))
      return TRUE;
  }

  if (albero->destro != NULL) {
    if ((albero->info < albero->destro->info) && Crescente(albero->destro))
      return TRUE;
  }

  return FALSE;
}
```

```
Es. ( 10
     ( 14 ( 23 () () )
          ( 28 ( 34 () () ) ( 1 () () ) ) )
         ( 67 ( 11 () () ) ( 53 () () ) ) )
```



- E' crescente questo albero?
- Per esercizio modificate la funzione Presente, ipotizzando di lavorare con alberi crescenti

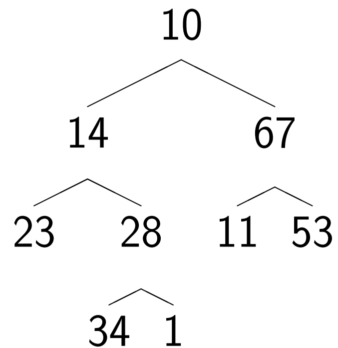
Algoritmi di visita



- Visita in preordine
  Si analizza la radice, poi si visita sottoalbero sinistro e sottoalbero destro
  Es. 1, 2, 4, 5, 3, 6, 7
- Visita in postordine
  Si visita prima sottoalbero sinistro, poi destro, e alla fine la radice
  Es. 4, 5, 2, 6, 7, 3, 1
- Visita Simmetrica
  Prima visito il figlio sinistro, poi la radice e quindi il figlio destro.
  Es. 4, 2, 5, 1, 6, 3, 7
- Visita per livelli
  Il primo nodo a essere analizzato è la radice. Poi si analizzano i figli della radice. E poi i figli dei figli... Es. 1, 2, 3, 4, 5, 6, 7
- Per tutte le visite: nel caso in cui l'abero sia vuoto non viene eseguita nessuna azione

```
void Analizza(int n)
{
  printf("%d ", n);
}

void VisitaPreordine(TipoAlbero a)
{
  if (a == NULL)
    return;
  Analizza(a->info);
  VisitaPreordine(a->sinistro);
  VisitaPreordine(a->destro);
}
```

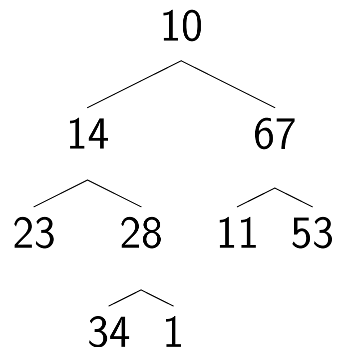
Come viene visitato l'albero?



Visita in postordine

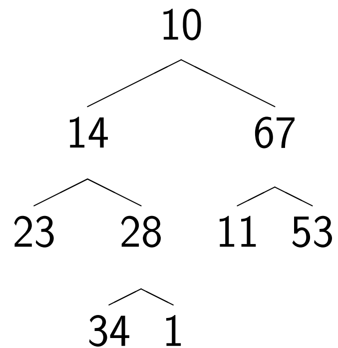
```
void Analizza(int n)
{
  printf("%d ", n);
}

void VisitaPostordine(TipoAlbero a)
{
  if (a == NULL)
    return;
  VisitaPostordine(a->sinistro);
  VisitaPostordine(a->destro);
  Analizza(a->info);
}
```

Come viene visitato l'albero?



```
void Analizza(int n)
{
  printf("%d ", n);
}

void VisitaSimmetrica(TipoAlbero a)
{
  if (a == NULL)
    return;
  VisitaSimmetrica(a->sinistro);
  Analizza(a->info);
  VisitaSimmetrica(a->destro);
}
```

Come viene visitato l'albero?



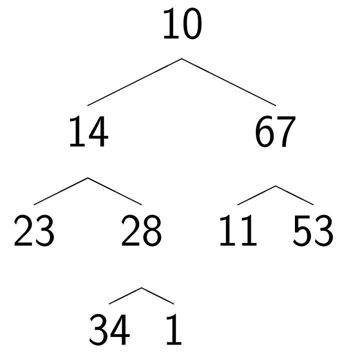
Visita per livelli

- Utilizziamo la struttura dati Coda (codapunt.c)

```
void VisitaLivelli(TipoAlbero albero)
{
  TipoCoda coda;
  TipoAlbero a;

  InitCoda(&coda);
  InCoda(&coda, albero);

  while (! TestCodaVuota(coda)) {
    OutCoda(&coda, &a);

    if (a != NULL) {
      Analizza(a->info);
      InCoda(&coda, a->sinistro);
      InCoda(&coda, a->destro);
    }
  }
  printf("\n");
}
```

- La coda è utilizzata per memorizzare i nodi in ordine di livello

  Esempio



- Come viene visitato l'albero?

Esercizio Lista-ABR (Appello Luglio 2003)

Dato un Albero Binario di Ricerca A etichettato con interi e data una lista L i cui nodi contengono valori interi non ripetuti e ordinati in ordine crescente, creare una nuova lista D i cui elementi sono tutti gli elementi dell'albero che non sono presenti nella lista L.

Soluzione "Facile"

- Dobbiamo visitare l'albero e chiederci per ogni elemento se questo appartiene o no alla lista L.
- Se non appartiene lo inseriamo in D, altrimenti continuiamo la visita.
- Inserisci Codice di Visita (qualsiasi) e Analizza con controllo!
- come posso migliorare questo algoritmo?
  - Sfrutto l'ordinamento di L
  - Sfrutto l'ordinamento di A

Cerchiamo di ragioniare sulla struttura dati

- Quale è la caratteristica dell'ABR?
- La lista L è ordinata.
- Quale visita per l'albero mi conviene utilizzare per sfruttare le proprietà delle strutture di input del problema? (Preordine, Postordine, Simmetrica, Per livelli)

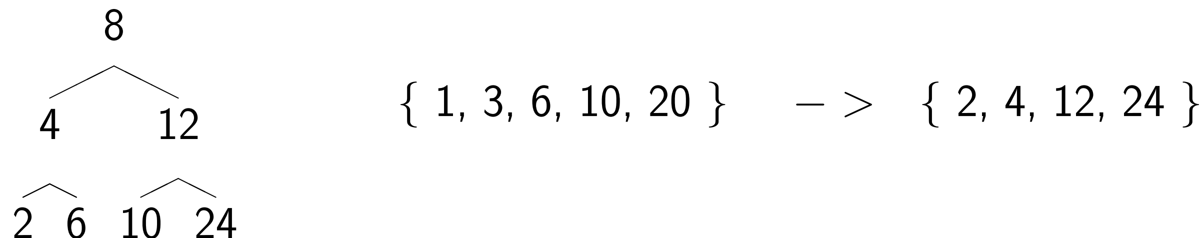
Visita Simmetrica

- Eseguiamo le visite di entrambe le strutture dati contemporaneamente
- Ogni volta che trovo un elemento di A che non appartiene a L lo inserisco in D
- La funzione ANALIZZA quindi è definita come segue:

```
VISITA(A,L,D)
   IF A <> NIL THEN
      VISITA(SX[A],L,D)
      ANALIZZA(KEY[A],L,D)
      VISITA(DX[A],L,D)
```

Concentriamoci sulla funzione ANALIZZA

- Scorro la lista L fino a quando finisce o INFO[L] $\geq$ KEY[A]
- Se INFO[L] $=$ NULL $->$ inserisco KEY[A] in D
- Se INFO[L] $\neq$ KEY[A] $->$ inserisco KEY[A] in D
- Quindi continuo la visita di A



Pseudo-Codice di ANALIZZA

```
ANALIZZA(K,L,D)
   WHILE ((L<>NIL) AND (INFO[L] < K))
      L <- NEXT[L]

   IF ((L = NIL) OR (INFO[L] <> K)) THEN
      INSERISCITESTALISTA(D,K)
```

Codifica in C

- L e D sono passate per riferimento

```
void analizza(TipoInfoAlbero k, TipoLista *L, TipoLista *D)
{

  while ((*L != NULL) && ((*L)->infoL < k))
    *L = (*L)->next;

  if ((*L == NULL) || ((*L)->infoL != k))
      InserisciTestaLista(D,k);

}
```

- Visita Simmetrica

```
void visita(TipoAlbero A,TipoLista *L, TipoLista *D)
{
  if (A!=NULL)
    {
      visita(A->sinistro,L,D);
      analizza(A->infoA,L,D);
      visita(A->destro,L,D);
    }
}
```

Per esercizio: Dato un albero A (rappresentato mediante puntatori) le cui etichette sono valori interi, costruire una lista L, contenente un nodo per ogni cammino in A, il cui valore è pari alla somma delle etichette del cammino.

Ancora per esercizio: Definire un tipo in C per un albero n-ario.