Algoritmi e Strutture Dati[1]

Corso di Laurea in Ingegneria dell'Informazione
Sapienza Università di Roma – sede di Latina

Fabio Patrizi

Dipartimento di Ingegneria Informatica, Automatica e Gestionale (DIAG)
Sapienza Università di Roma – Italy
www.dis.uniroma1.it/~patrizi
patrizi@dis.uniroma1.it

[1]Slides prodotte a partire dal materiale didattico fornito con il testo *Demetrescu, Finocchi, Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione*.

Strutture Dati Elementari

Tipi di Dato Astratti e Strutture Dati

Definition (Tipo di Dato Astratto)

Un *tipo di dato (astratto)* è una descrizione formale (matematica) dei dati d'interesse e delle operazioni ad essi associate.

Definition (Struttura Dati)

Una *struttura dati* è una specifica organizzazione implementativa dei dati di un tipo astratto che supporta le operazioni del tipo astratto.

- Il tipo di dato astratto ci dice *cosa* fare, non *come*, né come organizzare i dati
- La struttura dati ci dice *come organizzare i dati* su cui implementeremo le operazioni
- Il modo in cui le operazioni sono implementate dipende dalla specifica struttura dati

Tipo di Dato Astratto *Dizionario*

tipo $Dizionario$:
dati: insieme finito $S \subseteq Chiave \times Elemento$ ($Chiave$ è totalmente ordinato)
operazioni:

- $insert(Chiave\ c, Elemento\ e)$:
  Se $\neg\exists e'.(c, e') \in S$, aggiunge (c, e) ad S
- $delete(Chiave\ c)$:
  Se $\exists e.(c, e) \in S$, rimuove (c, e) da S
- $search(Chiave\ c) \rightarrow Elemento$:
  Se $\exists e.(c, e) \in S$, restituisce e, altrimenti restituisce $null$

Osservazione: D non può contenere due coppie con stessa chiave.

Il Tipo di Dato Astratto $Pila$

tipo $Pila$:
dati: Sequenza S di n elementi $e \in Elemento$
operazioni:

- $isEmpty() \rightarrow Boolean$:
  Restituisce $true$ se S è vuota, $false$ altrimenti
- $push(Elemento\ e)$:
  Inserisce e come elemento affiorante di S
- $pop() \rightarrow Elemento$:
  Estrae l'elemento affiorante da S e lo restituisce
- $top() \rightarrow Elemento$:
  Restituisce l'elemento affiorante da S (senza estrarlo)

Politica di accesso LIFO (last-in-first-out)

Il Tipo di Dato Astratto $Coda$

tipo $Coda$:
dati: Sequenza S di n elementi $e \in Elemento$
operazioni:

- $isEmpty() \rightarrow Boolean$:
  Restituisce $true$ se S è vuota, $false$ altrimenti
- $enqueue(Elemento\ e)$:
  Inserisce e in S come elemento in ultima posizione
- $dequeue() \rightarrow Elemento$:
  Estrae da S l'elemento in prima posizione e lo restituisce
- $first() \rightarrow Elemento$:
  Restituisce l'elemento di S in prima posizione (senza estrarlo)

Politica di accesso FIFO (first-in-first-out)

Tecniche di Rappresentazione dei Dati

Vogliamo implementare un dizionario

Il primo problema che si presenta è: *come rappresentare l'insieme* S?

Due approcci:

- Rappresentazione *indicizzata*: dati contenuti in un array
  - Vantaggio: accesso indicizzato (tempo costante)
  - Svantaggio: dimensione fissa (riallocazione in tempo lineare)
- Rappresentazione *collegata*: dati contenuti in record collegati da puntatori
  - Vantaggio: dimensione variabile (tempo costante)
  - Svantaggio: accesso sequenziale (tempo lineare)

La rappresentazione scelta influenza il costo delle operazioni!

Implementazione Indicizzata di $Dizionario$

Classe $DizionarioIndicizzato$ **implementa** $Dizionario$:
dati: array D di n coppie $(chiave, elemento) \in Chiave \times Elemento$, ordinate secondo c.
operazioni:

- $search(Chiave\ c) \rightarrow E$:

  return $RicercaBinaria(D, c)$;

  $T(n) = \mathcal{O}(\log(n))$, $S(n) = \Theta(1)$.

- $insert(Chiave\ c, Elemento\ e)$:

  if $(search(c) \neq null)$ **then return** ;
  D' : array di dimensione $n + 1$;
  $i := 0$;
  while $(i < n$ e $D[i].chiave < c)$ **do**
      $D'[i] := D[i]$;
      $i := i + 1$;
  $D'[i] := (c, e)$;
  for $(j = i, \ldots, n - 1)$ **do** $D'[j + 1] := D[j]$;
  $D := D'$;

  $T(n) = \mathcal{O}(n)$, $S(n) = \Theta(n)$.

Implementazione Indicizzata di *Dizionario*

- $delete(Chiave\ c)$:

  if $(search(c) == null)$ **then return** ;
  D' : array di dimensione $n-1$;
  for $i = 0, \ldots, n-2$ **do**
      if $(D[i].chiave < c)$ **then** $D'[i] := D[i]$;
      if $(D[i].chiave \geq c)$ **then** $D'[i] := D[i+1]$;
  $D := D'$;

  $T(n) = \mathcal{O}(n)$, $S(n) = \Theta(n)$.

Si noti che le operazioni di inserimento e cancellazione preservano l'ordinamento dell'array. È pertanto possibile effettuare la ricerca tramite l'algoritmo di ricerca binaria.

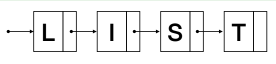
Strutture Collegate Lineari (SCL)

Una *struttura collegata* è una struttura dati formata da un insieme di record, in cui:

- I dati d'interesse sono contenuti nei campi dei record
- Ciascun record contiene uno o più riferimenti ad altri record
- Esiste un record da cui tutti i record della struttura sono accessibili

Una struttura collegata è detta *lineare* se i suoi riferimenti permettono di definire un ordine totale sui suoi elementi

Example (Lista Semplice)



Implementazione di $Dizionario$ con SCL

Classe $DizionarioCollegato$ **implementa** $Dizionario$:

dati:

- lista collegata D di n record $(chiave, elemento, next) \in Chiave \times Elemento \times Riferimento$
- riferimento $dizionario$ al primo record ($null$ se dizionario vuoto)

operazioni:

- $insert(Chiave\ c, Elemento\ e)$:

  $aux := dizionario$;
  while $(aux \neq null)$ **do**
      if $(aux \rightarrow chiave == c)$ **then return** ;
      $aux := aux \rightarrow next$;
  inserisci il record (c, e) in testa.

  $T(n) = \mathcal{O}(n)$, $S(n) = \Theta(1)$.

Implementazione di $Dizionario$ con SCL

- $delete(Chiave\ c)$:

  if $(dizionario == null)$ **then return** ;

  if $(dizionario \rightarrow chiave == c)$ **then**

  $\quad dizionario = dizionario \rightarrow next;$

  $\quad$ **return** ;

  $aux := dizionario;$

  while $(aux \rightarrow next \neq null$ e $aux \rightarrow next \rightarrow chiave \neq c)$ **do**

  $\quad aux := aux \rightarrow next;$

  if $(aux \rightarrow next \rightarrow chiave == c)$ **then**

  $\quad aux \rightarrow next := aux \rightarrow next \rightarrow next;$

  $T(n) = \mathcal{O}(n)$, $S(n) = \Theta(1)$.

- $search(Chiave\ c) \rightarrow E$:

  $aux := dizionario;$

  while $(aux \neq null)$ **do**

  $\quad$ **if** $(aux \rightarrow chiave == c)$ **then return** $aux \rightarrow elemento;$

  $\quad aux := aux \rightarrow next;$

  return $null;$

  $T(n) = \mathcal{O}(n)$, $S(n) = \Theta(1)$.

Confronto fra le due implementazioni di *Dizionario*

- Non osserviamo sostanziali differenze tra l'implementazione indicizzata e quella collegata di *Dizionario*, in termini di tempo
- Addirittura, l'implementazione indicizzata offre un'operazione di ricerca più efficiente (tempo logaritmico vs. lineare)
- Tuttavia, la rappresentazione collegata offre un risparmio significativo in termini di spazio (costante vs. lineare)

Implementazione Indicizzata di $Pila$

Classe $PilaIndicizzata$ **implementa** $Pila$:

dati: array S contenente n elementi dall'insieme $Elemento$ (la prima componente di S contiene l'elemento affiorante della pila);

operazioni:

- $isEmpty() \rightarrow Boolean$:

  return $(n == 0)$;

  $T(n) = \Theta(1)$, $S(n) = 0$.

- $push(Elemento\ e)$:

  S' : array di dimensione $n + 1$;
  $S'[0] := e$;
  for all $(i \in [1, n])$ **do** $S'[i] := S[i - 1]$;
  $S := S'$.

  $T(n) = \mathcal{O}(n)$, $S(n) = \Theta(n)$.

Implementazione Indicizzata di $Pila$

- $pop() \to E$:

  if $(n == 0)$ **then return** $null$;

  $r := S[0]$;

  S' : array di dimensione $n-1$;

  for all $(i \in [1, n-1])$ **do** $S'[i-1] := S[i]$;

  $S := S'$;

  return r.

  $T(n) = \mathcal{O}(n)$, $S(n) = \Theta(n)$.

- $top() \to E$:

  if $(n == 0)$ **then return** $null$;
  return $S[0]$;

  $T(n) = \Theta(1)$, $S(n) = 0$.

Implementazione di $Pila$ con struttura collegata

Classe $PilaCollegata$ **implementa** $Pila$:

dati:

- lista collegata S di n record $(elemento, next) \in Elemento \times Riferimento$;
- riferimento $pila$ al primo record della lista.

operazioni:

- $isEmpty() \to Boolean$:

  return $pila == null$;

  $T(n) = \Theta(1)$, $S(n) = 0$.

- $push(Elemento\ c)$:

  $nuovo := c$
  Inserisci nuovo in testa a $pila$

  $T(n) = \Theta(1)$, $S(n) = \Theta(1)$.

Implementazione di $Pila$ con struttura collegata

Classe $PilaCollegata$ **implementa** $Pila$:
dati:

- $pop() \to E$:

  if $(isEmpty())$ **then return** $null$;
  $result := pila \to elemento$;
  $pila := pila \to next$;
  return $result$;

  $T(n) = \Theta(1)$, $S(n) = \Theta(1)$.

- $top() \to E$:

  if $(isEmpty())$ **then return** $null$;
  return $pila \to elemento$;

  $T(n) = \Theta(1)$, $S(n) = 0$.

Confronto fra le due implementazioni di $Pila$

- In questo caso l'implementazione collegata offre maggiore risparmio in termini sia di spazio che di tempo

Alberi

Spesso è utile organizzare i dati in maniera *gerarchica* (esempio: file system)

Alberi: strutture matematiche che astraggono il concetto di *gerarchia*

Definition (Albero)

Albero $T = (N, A)$:

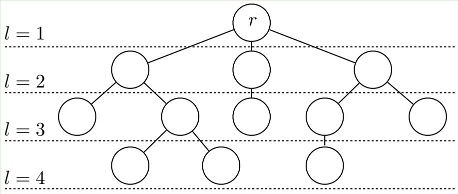
- N è l'insieme finito dei *nodi*
- $A \subseteq N \times N$ è l'insieme degli *archi*

tale che:

- Se $N \neq \emptyset$ esiste esattamente un nodo r, detto *radice*, che non ha padre (non esiste nessun arco $(n, r) \in A$)
- Tutti i nodi n diversi da r hanno esattamente un padre n' (esiste $n' \in N$ t.c. $(n', n) \in A$)
- Nessun nodo n può essere antenato di se stesso ($\{(n_0, n_1), (n_1, n_2), \ldots, (n_{\ell-1}, n_\ell), (n_\ell, n_0)\} \not\subseteq A$.)

Alberi

Example (Albero)



Associeremo un campo informativo $info$ ad ogni nodo
Notazione:

- profondità (depth) d di un nodo n: #archi nel percorso (unico) radice-n
- grado (degree) di un nodo n: #archi uscenti da n
- livello (level) l di un albero: insieme dei nodi con profondità $l - 1$
- altezza (height) h di un albero: livello massimo dell'albero

Il Tipo di Dato Astratto *Albero*

tipo $Albero$:
dati: Insieme $N \subseteq Nodo$ di nodi, Insieme $A \subseteq N \times N$ di archi
operazioni:

- $numNodi \rightarrow Intero$: Restituisce $|N|$
- $grado(Nodo\ n) \rightarrow Intero$: Restituisce il numero di archi uscenti da n
- $padre(Nodo\ n) \rightarrow Nodo$: Restituisce il padre di n
- $figli(Nodo\ n) \rightarrow Insieme[Nodo]$: Restituisce l'insieme dei figli di n
- $aggiungiNodo(Nodo\ n) \rightarrow Nodo$: Crea un nuovo nodo v, lo inserisce come figlio di n e lo restituisce. Se l'albero è vuoto, v ne diventa la radice (ed n viene ignorato)
- $aggiungiSottoalbero(Albero\ a, Nodo\ n)$: Inserisce l'albero a come sottoalbero, rendendo la radice di a figlio di n. Se l'albero è vuoto, a diventa l'albero
- $rimuoviSottoalbero(Nodo\ n) \rightarrow Albero$: Disconnette il sottoalbero con radice in n (compresa) dall'albero e lo restituisce.

Rappresentazione di Alberi

Anche per gli alberi si pone il problema della loro implementazione, in particolare di *come organizzare i dati*

Anche in questo caso, abbiamo due possibili alternative:

- Rappresentazione indicizzata
- Rappresentazione mediante struttura collegata (non lineare)

Rappresentazioni Indicizzate di Alberi

Vettore dei Padri

Le componenti di ciascun vettore contengono un record $(info, padre)$:

- $info$ è il contenuto informativo del nodo
- $padre$ è l'indice della componente corrispondente al nodo padre

Costo delle operazioni base:

- $padre(n)$: $\mathcal{O}(1)$
- $figli(n)$: $\mathcal{O}(|N|)$

Rappresentazioni Indicizzate di Alberi

Vettore Posizionale (per alberi d-ari completi, con $d \geq 2$

- Il nodo radice è in posizione 0
- Ciascuna componente contiene l'informazione relativa ad un nodo
- I figli del nodo in posizione v si trovano in posizione $d \cdot v + i$, per $i = 1, \ldots, d$

Costo delle operazioni base:

- $padre(n)$: $\mathcal{O}(1)$ (infatti: $v = (p-1)/d$)
- $figli(n)$: $\mathcal{O}(grado(n))$

Rappresentazioni Collegate di Alberi

Puntatori ai Figli (solo per alberi con grado limitato d)

Ciascun nodo è un record di tipo $(info, figlio_1, \ldots, figlio_m)$:

- $info$: contenuto informativo del nodo
- $figlio_i$: riferimento al figlio i-esimo ($null$ se assente)

Spazio occupato: $\mathcal{O}(|N|)$

Costo delle operazioni base:

- $padre(n)$: $\mathcal{O}(|N|)$
- $figli(n)$: $\mathcal{O}(grado(n))$

Lista dei Figli

Ciascun nodo è un record di tipo $(info, figli)$:

- $info$: contenuto informativo del nodo
- $figli$: riferimento ad una lista di riferimenti ai figli

Spazio occupato: $\mathcal{O}(|N|)$

Costo delle operazioni base:

- $padre(nodo)$: $\mathcal{O}(|N|)$
- $figli(nodo)$: $\mathcal{O}(1)$

Variante: ogni record contiene un riferimento al primo figlio ed al fratello successivo.

Visita di un Albero

Esplorazione esaustiva dei nodi di un albero

Visita generica

$Algoritmo\ visitaGenerica(nodo\ r)$

$S \leftarrow \{r\}$
while $(S \neq \emptyset)$ **do**
  estrai un nodo $u \in S$
  visita u
  $S = S \cup \{figli(u)\}$

- Struttura dati non lineare: visita possibile seguendo vari ordini
- La visita generica non impone un ordine specifico di visita
- La politica di gestione di S condiziona il tipo di visita

Visita di un Albero

Theorem

L'algoritmo $visitaGenerica(r)$ termina in al più $\mathcal{O}(n)$ passi, usa al più $\mathcal{O}(n)$ unità di memoria e visita tutti i nodi dell'albero con radice r.

Proof.

Ad ogni iterazione, viene estratto un nodo dall'insieme S e ne vengono inseriti i figli in S. Poiché l'albero non ha cicli, un nodo estratto non viene mai reinserito in S. Pertanto, dopo al più $\mathcal{O}(n)$ passi, S è vuoto e l'algoritmo termina.

Per l'occupazione di memoria, si noti che S contiene al più tutti gli n nodi dell'albero.

Mostriamo per contraddizione che tutti i nodi sono visitati, assumendo che un nodo u non lo sia. In tal caso, $padre(u)$ non sarebbe visitato, in quanto sappiamo che ogni volta che un nodo viene visitato, *tutti* i suoi figli sono inseriti in S (e quindi visitati). Ma allora anche $padre(padre(u))$ non lo sarebbe, e così via fino alla radice r che, invece, sappiamo essere visitata alla prima iterazione. L'assurdo nasce dall'aver supposto l'esistenza di un nodo non visitato.

Visita in profondità (DFT: depth-first traversal)

S viene gestito come una pila: si procede dalla radice verso le foglie, scendendo di livello appena possibile

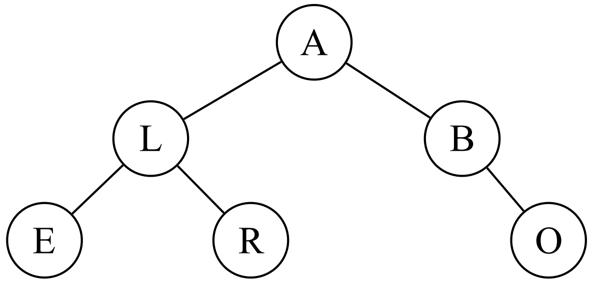
Visita in profondità

$Algoritmo\ visitaDF(nodo\ r)$

    Pila vuota S
    $S.push(r)$
    while $(\textbf{not}\ S.isEmpty())$ **do**
        $u = S.pop()$
        visita u
        for all $(v \in figli(u))$ **do**
            $S.push(v)$

Visita in profondità

Example (Visita in profondità)



Assumendo che i nodi vengano inseriti in S da destra verso sinistra, l'ordine di visita dell'albero è: A L E R B O

Visita in profondità ricorsiva

La visita in profondità può essere implementata ricorsivamente, senza l'uso della pila S (che viene costruita implicitamente nello stack)

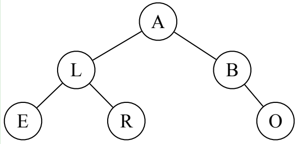
Visita in profondità ricorsiva

$Algoritmo\ visitaDFRicorsiva(nodo\ r)$

if $(r == null)$ **then**
  return
visita r
for all $(u \in figli(r))$ **do**
  $visitaDFRicorsiva(u)$

Implementazioni ricorsive naturali ed eleganti

Visita in profondità ricorsiva



Nella visita in profondità ricorsiva, la visita di un nodo si può eseguire:

- *in preordine*: prima della visita dei figli (A L E R B O)
- *simmetricamente* (per alberi binari): in mezzo alla visita dei figli (E L R A B O)
- *in postordine*: dopo la visita dei figli (E R L O B A)

Visita in ampiezza (BFT: breadth-first traversal)

S viene gestito come una coda: si procede per livelli, partendo dalla radice e scendendo solo quando il livello corrente è stato completato

Visita in ampiezza

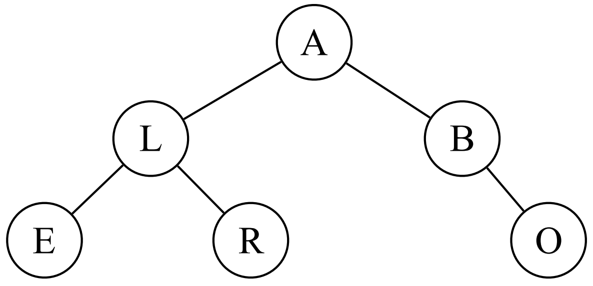
$Algoritmo\ visitaBF(nodo\ r)$

  Coda vuota S
  $S.enqueue(r)$
  while $(\mathbf{not}\ S.isEmpty())$ **do**
    $u = S.dequeue()$
    visita u
    for all $(v \in figli(u))$ **do**
      $S.enqueue(v)$

Implementazioni ricorsive possibili ma artificiose e inutilmente complesse

Visita in ampiezza

Example (Visita in ampiezza)



Assumendo che i nodi vengano inseriti in S da sinistra verso destra, l'ordine di visita dell'albero è: A L B E R O

Ricerca di un Nodo in un albero

La visita di un albero si presta a vari scopi, ad es.:

- *Ricerca*: per ogni nodo visitato, verifica se corrisponde al criterio di ricerca
- *Conteggio*: ogni volta che trovo un nodo che corrisponde al criterio di ricerca, incremento il risultato

Può essere conveniente procedere in ampiezza o in profondità

Ricerca: *breadth-first search* (BFS) o *depth-first search* (DFS)

Esercizi

Esercizio 1 (d'esame)

1. Specificare un algoritmo (pseudocodice) con segnatura: $quantiNodi(Albero\ T) \rightarrow Intero$ che, preso in input un albero T, ne restituisce il numero di nodi.
2. Indicare, motivando la risposta, il costo temporale dell'algoritmo definito.

Esercizio 2 (d'esame)

1. Specificare un algoritmo (pseudocodice) con segnatura: $presente(Albero\ T, Intero\ i) \rightarrow Boolean$ che, preso in input un albero T con nodi contenenti valori interi, ed un valore intero i, restituisce il valore $true$ se e solo T contiene un nodo con valore i.
2. Indicare, motivando la risposta, il costo temporale dell'algoritmo definito.