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

HEAP SORT

StudentiDIA Forum

http://forum.studentidia.org/

Dove trovare i lucidi presentati a lezione

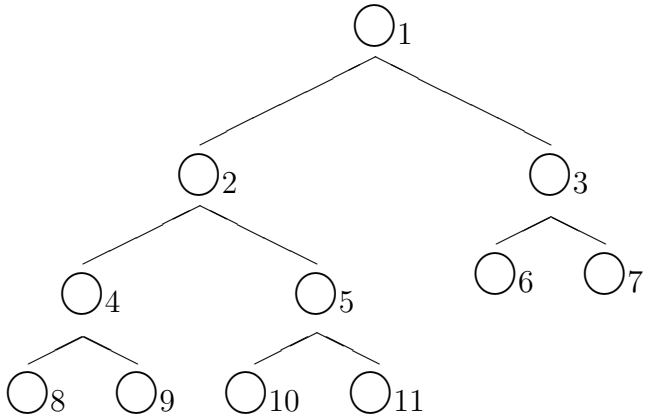
http://limongelli.dia.uniroma3.it/asd/

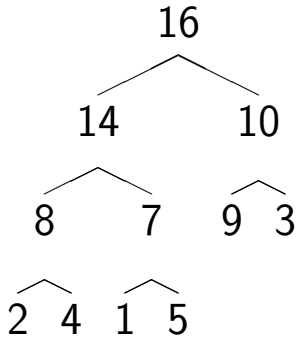
HEAP e HEAP SORT

- Presentiamo una possibile implementazione degli HEAP e dell'algoritmo di HEAP-SORT
- Utilizziamo un array per rappresentare l'HEAP
- Implementiamo le funzioni per la manipolazione degli HEAP
- Implementiamo le funzioni che vengono utilizzate per l'HEAP-SORT
- Esercizi

Rappresentazione degli HEAP

- un HEAP è un albero bilanciato rappresentato mediante un array. Ad ogni nodo dell'albero è associato un indice





16	14	10	8	7	9	3	2	4	1	5	
1	2	3	4	5	6	7	8	9	10	11	$\cdots$

Dettagli di Implementazione

- i valori dell'ARRAY sono i valori che dobbiamo ordinare
- gli indici dell'ARRAY corrispondono agli indici dell'HEAP

```
typedef int indice;

#define NumElementi 100
typedef int TipoElemVettore;
typedef TipoElemVettore TipoVettore[NumElementi];

TipoVettore HEAP;
```

- N.B. Problema degli indici: In questo caso scegliamo di usare la stessa indicizzazione.

Rappresentiamo anche i due attributi degli HEAP:

- LENGTH indica il numero di elementi contenuti nell'ARRAY
- HEAP-SIZE indica il numero di elementi dello HEAP

```
length = HEAP[NumElementi-1];
heap_size = HEAP[0];
```

hs	16	14	10	8	7	9	3	2	4		$length$
0	1	2	3	4	5	6	7	8	9	$\cdots$	99

- Vale sempre $heap\text{-}size \leq length$.
  Nessun elemento dopo l'indice $HEAP[heap\text{-}size]$ è nella struttura Heap, anche se presente nell'Array.

Funzioni per la manipolazione degli HEAP

- Funzioni per la gestione delle dimensioni

```
int length(TipoVettore HEAP)
{
  return HEAP[NumElementi-1];
}

void set_length(TipoVettore HEAP,int i)
{
  HEAP[NumElementi-1]=i;
}

int heap_size(TipoVettore HEAP)
{
  return HEAP[0];
}

void set_heap_size(TipoVettore HEAP,int i)
{
  HEAP[0]=i;
}
```

- Funzione INIT

```
void INIT(TipoVettore HEAP)
{
  int i;

  for (i=1;i<=length(HEAP);i++)
    {
      printf("HEAP[%d]=",i);
      scanf("%d",&HEAP[i]);
    }
}
```

- funzione PRINT-HEAP

```
void PRINT_HEAP(TipoVettore HEAP)
{
  int i;

  for (i=1;i<=heap_size(HEAP);i++)
    {
      printf("HEAP[%d]=%d\n",i,HEAP[i]);
    }
  printf("\n");
}
```

Le funzioni per implementare l'HEAP-SORT

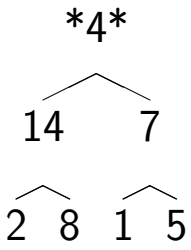
- PARENT(i), LEFT(i), RIGHT(i) e scambia

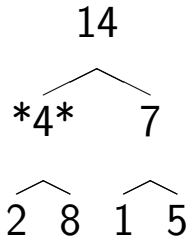
```
indice PARENT(indice i)
{
  return (i/2);
}
indice LEFT(indice i)
{
  return 2*i;
}
indice RIGHT(indice i)
{
  return (2*i)+1;
}

void scambia(TipoVettore HEAP, indice ind1, indice ind2)
{
  TipoElemVettore appo;

  appo = HEAP[ind1];
  HEAP[ind1]=HEAP[ind2];
  HEAP[ind2]=appo;
}
```

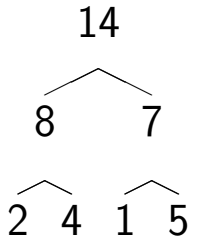
Algoritmo di HEAPIFY

- Verifica che un nodo dell'albero sia HEAP
- I figli sono HEAP. Ma potrebbero contenere valori più grandi del padre.

```
HEAPIFY(A,i)
 1.  l <- LEFT(i)
 2.  r <- RIGHT(i)
 3.  if l <= heap-size[A] e A[l] > A[i]
 4.     then largest <- l
 5.     else largest <- i
 6.  if r <= heap-size[A] e A[r] > A[largest]
 7.     then largest <- r
 8.  if largest <> i
 9.     then scambia A[i] <-> A[largest]
10.     HEAPIFY(A,largest)
```







- HEAPIFY

```
void HEAPIFY(TipoVettore HEAP, indice i)
{
  indice largest;
  indice l = LEFT(i);
  indice r = RIGHT(i);

  if ((l <= heap_size(HEAP)) && (HEAP[l] > HEAP[i]))
    largest = l;
  else
    largest = i;

  if ((r <= heap_size(HEAP)) && (HEAP[r] > HEAP[largest]))
    largest = r;

  if (largest != i)
    {
      scambia(HEAP,i,largest);
      HEAPIFY(HEAP,largest);
    }
}
```

Algoritmo di BUILD-HEAP

- Costruzione di un HEAP a partire da un ARRAY di valori
- BUILD-HEAP esegue HEAPIFY sui nodi rimanenti, dal basso verso l'alto.

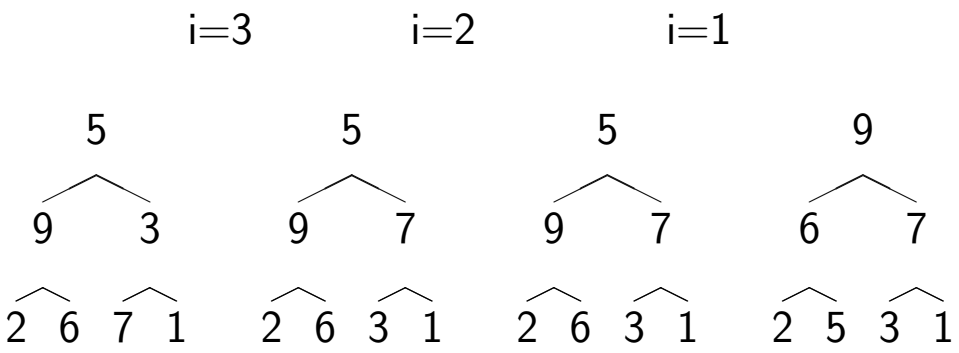
```
BUILD-HEAP(A)
1.  heap-size[A] <- length[A]
2.  for i <- (length[A] div 2) downto 1
3.       do HEAPIFY(A,i)
```

- Se $n = length$, gli elementi con indice $\geq \lfloor n/2 \rfloor + 1$ sono tutte foglie: ognuna è uno heap con un solo elemento.
- BUILD-HEAP

  Esempio:

5	9	3	2	6	7	1

  $\lfloor length[A]/2 \rfloor = 3$



```
void BUILD_HEAP(TipoVettore HEAP)
{
  int i;

  set_heap_size(HEAP,length(HEAP));
  for (i=(length(HEAP)/2);i>0;i--)
    HEAPIFY(HEAP,i);
}
```

Algoritmo di HEAP-SORT

- Costruzione di uno heap dall'array.
- L'elemento massimo è in $HEAP[1]$: si scambia con $HEAP[n]$ e si decrementa $heap\text{-}size$.
- Il nuovo valore $HEAP[1]$ viene "spinto in basso" con HEAPIFY(A,1).

```
HEAPSORT(A)
1.  BUILD-HEAP(A)
2.  for i <- length[A] downto 2
3.      do scambia A[1] <-> A[i]
4.         heap-size[A] <- heap-size[A] - 1
5.         HEAPIFY(A,1)
```

- HEAP-SORT

```
void HEAP_SORT(TipoVettore HEAP)
{
  int i;

  BUILD_HEAP(HEAP);
  for(i=length(HEAP);i>1;i--)
    {
      scambia(HEAP,1,i);
      set_heap_size(HEAP,heap_size(HEAP) - 1);
      HEAPIFY(HEAP,1);
    }
}
```

- Esempio HEAP-SORT

  Array Iniziale: | 5 | 9 | 3 | 2 | 6 | 7 | 1 |

  Heap heap-size=7: | 9 | 6 | 7 | 2 | 5 | 3 | 1 |

  Scambio,Heapify(1) heap-size=6: | 1 | 6 | 7 | 2 | 5 | 3 | 9 |

  New Heap: | 7 | 6 | 3 | 2 | 5 | 1 | 9 |

  Scambio,Heapify(1) heap-size=5: | 1 | 6 | 3 | 2 | 5 | 7 | 9 |

  New Heap: | 6 | 5 | 3 | 2 | 1 | 7 | 9 |

  ...

  Array Ordinato: | 1 | 2 | 3 | 5 | 6 | 7 | 9 |

Esercizio

1. Prendiamo da input n valori
2. Costruiamo una struttura Heap
3. Descrivere un algoritmo che dall'Heap appena costruito ne produca due composti da $n/2$ elementi (al massimo differenza di un elemento), tali che tutte le chiavi del primo siano minori di quelle del secondo

Ragioniamo Top-Down

- Leggere da input dei valori e costruire una struttura HEAP lo sappiamo fare. (punti 1 e 2).
- Dobbiamo "pensare" il punto 3.

Pseudo Codice

```
PROBLEMA(A,B,C)
    ;; x è un intero che indica la lunghezza dell'array A
    SET_LENGTH(A,x)
    INIT(A)
    BUILD_HEAP(A)
    SPLIT_HEAP(A,B,C) ;; Splitta A in due HEAP B e C
```

Cosa deve fare lo Split_Heap?

- Dividere gli elementi di A in due insiemi di $n/2$ elementi (Ricordiamo che gli elementi nel primo devono essere tutti minori di quelli del secondo)
- Inserire in B i primi $n/2$
- Inserire in C gli altri

Devo trovare una partizione degli elementi di A

- Posso ordinare gli elementi e poi dividere a metà l'Array.
- quindi costruire due Heap...

```
SPLIT_HEAP(A,B,C)
   HEAP_SORT(A);
   DIVIDI_IN_DUE(A,B,C)
```

Per dividere?

- Calcolo il valore mediano dell'indice (med)
- Copio i valori di A da 1 a med in B
- Copio i valori di A da med+1 a length(A) in C

```
DIVIDI_IN_DUE(A,B,C)
   med = LENGTH(A)/2
   LENGTH(B) = med
   LENGTH(C) = LENGTH(A) - med

   FOR I=1 TO med
      B[I] = A[I]
   FOR I=med+1 to LENGTH(A)
      C[I-med] = A[I]

   BUILD_HEAP(B)
   BUILD_HEAP(C)
```

Traduzione in codice C

```
void Problema()
{
  TipoVettore A,B,C;
  int lung;

  /* leggo da input il numero di elementi */
  printf("Lunghezza vettore A: ");
  scanf("%d",&lung);

  set_length(A,lung);
  INIT(A);

  BUILD_HEAP(A);
  split_Heap(A,B,C);

}

void split_Heap(TipoVettore A,TipoVettore B, TipoVettore C)
{
  HEAP_SORT(A);
  dividiIn2(A,B,C,length(A));
}
```

Codice C dividiIn2

```
void dividiIn2(TipoVettore A, TipoVettore B, TipoVettore C)
{
  int i;  int med = length(A)/2;

  set_length(B,med);
  set_length(C,length(A)-med);

  for (i=1;i<=med;i++)
      B[i] = A[i];
  for (i=med+1;i<=length(A);i++)
      C[i-med] = A[i];

  BUILD_HEAP(B);
  BUILD_HEAP(C);
}
```

- Possiamo costruire gli HEAP B e C in MODO DIFFERENTE???
- suggerimento: non usare la BUILD_HEAP, ma una funzione differente.

```
void REVERSE(TipoVettore V)
{
  int i;
  for (i=1;i<=length(V)/2;i++)
      scambia(V,i,length(V)+1-i);
}

void dividiIn2(TipoVettore A, TipoVettore B, TipoVettore C)
{
  int i;  int med = length(A)/2;

  set_length(B,med);
  set_length(C,length(A)-med);

  for (i=1;i<=med;i++)
      B[i] = A[i];
  for (i=med+1;i<=length(A);i++)
      C[i-med] = A[i];

  REVERSE(B); /* BUILD_HEAP(B); */
  REVERSE(C); /* BUILD_HEAP(C); */
}
```

Esercizio x casa:

Dato un Heap, costruire un altro Heap invertendo l'ordinamento. (Vale cioè: A[PARENT(i)]<A[i])