Binomial Heap

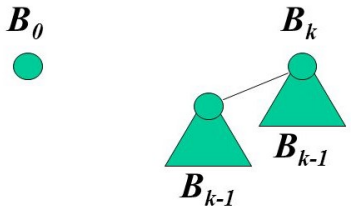
Rosario Forte

Gennaio 2020

1. Introduzione

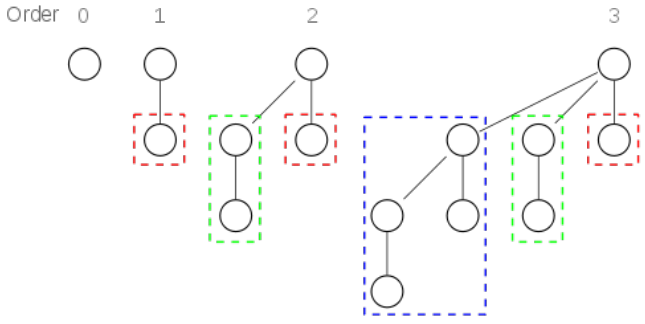
Gli alberi binomiali sono particolari alberi così definiti:

1. B_0 è l'albero formato da un solo nodo.
2. Dato B_{k-1} definiamo B_k combinando 2 copie di B_{k-1}.



Inoltre essi godono delle seguenti proprietà:

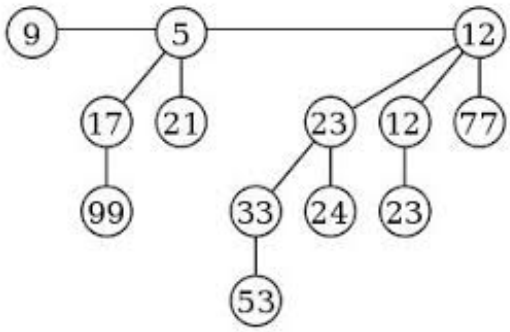
1. Sono presenti 2^K nodi.
2. L'altezza dell'albero è k.
3. Sono presenti esattamente $\binom{k}{i}$ nodi a profondità i, per i =0,1,. . . ,k.
4. Un albero di ordine k ha come figli gli alberi di ordine k-1,k-2,. . . ,0, nell'ordine dato.



Gli heap binomiali sono strutture dati utilizzate per l'implementazione di code di priorità, basate su alberi binomiali, nelle quali l'operazione di merge ha una complessità inferiore rispetto agli heap classici, questo li classifica come mergeable heap.
Un heap binomiale gode delle seguenti proprietà:

1. Ogni albero binomiale all'interno dell'heap gode della proprietà del min-heap.

2. Nell'heap per qualsiasi intero k positivo è presente al più un solo albero di ordine k.



2. Metodi e Complessità

Le complessità e i successivi pseudocodici fanno riferimento all'implementazione di un min Binomial Heap.

Procedure	Bynary Heap (caso pssimo)	Binomial Heap (caso pessimo)
MakeHeap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\mathcal{O}(\log n)$
Minimum	$\Theta(1)$	$\mathcal{O}(\log n)$
ExtractMinimun	$\Theta(\log n)$	$\Theta(\log n)$
UnionHeap	$\Theta(n)$	$\mathcal{O}(\log n)$
DecreaseKey	$\Theta(\log n)$	$\Theta(\log n)$
DeleteKey	$\Theta(\log n)$	$\Theta(\log n)$

3. Pseudocodice

3.1 Creazione di un Binomial Heap

La seguente procedura si occupa di allocare un heap, con la testa che punta a Nil.

MAKEHEAP()

1. H←allocateHeap()
2. head[H]←Nil
3. **return** H

Complessità $\mathcal{O}(1)$

3.2 Ricerca del minimo

La procedura si occupa di scorrere tutte le radici degli alberi, conservando un puntatore al nodo minimo e restituendolo.

MINIMUM(H)

1. y←Nil
2. x←head[H]
3. min← $-\infty$
4. **while** x≠Nil
5. **do if** key[x]<min
6. **then** min←key[x]
7. y←x
8. x←sibling[x]
9. **return** y

Complessità $\mathcal{O}(\log n)$

3.3 Unione di due Binomial Tree

La procedura unisce due binomial tree di grado k-1, formandone uno di grado k, che continua a rispettare le proprietà dei min-heap, questa procedura sarà utilizzata in seguito come supporto a procedure più complesse.

MERGETREE(y,z)

1. sibling[y]←child[z]

2. parent[y]←z
3. child[z]←y
4. degree[z]←degree[z]+1
5. **return** z

Complessità $\mathcal{O}(1)$

3.4 Merge di due Binomial Heap

La procedura di merge di due binomial heap è distruttiva, infatti essa unisce due heap distruggendoli e restituendo un puntatore alla testa del nuovo heap, l'heap ritornato potrebbe non rispettare le proprietà dei binomial-heap, data la presenza di due radici dello stesso grado. Si noti che dall'unione di due heap che rispettano le proprietà, avremo in output un heap ordinato secondo il grado delle radici, nella quale vi sono al più 2 radici dello stesso grado.

MergeHeap(H1,H2)

1. it1←head[h1]
2. it2←head[h2]
3. **if** it1=Nil
4. **then return** it2
5. **if** it2=Nil
6. **then return** it1
7. **if** degree[it1]<degree[it2]
8. **then** merged←it1
9. it1←sibling[it1]
10. **else**
11. **then** merged←it2
12. it2←sibling[it2]
13. it3←merged
14. **while** it1≠ Nil and it2≠Nil
15. **do if** degree[it1]<degree[it2]
16. **then** sibling[it3]←it1
17. it1←sibling[it1]

18. **else**
19. **then** sibling[it3]←it2
20. it2←sibling[it2]
21. **if** it1≠Nil
22. **then** sibling[it3]←it1
23. **if** it2≠Nil
24. **then** sibling[it3]←it2
25. **return** merged

Complessità $\mathcal{O}(\log n)$

3.5 Unione di due Binomial Heap

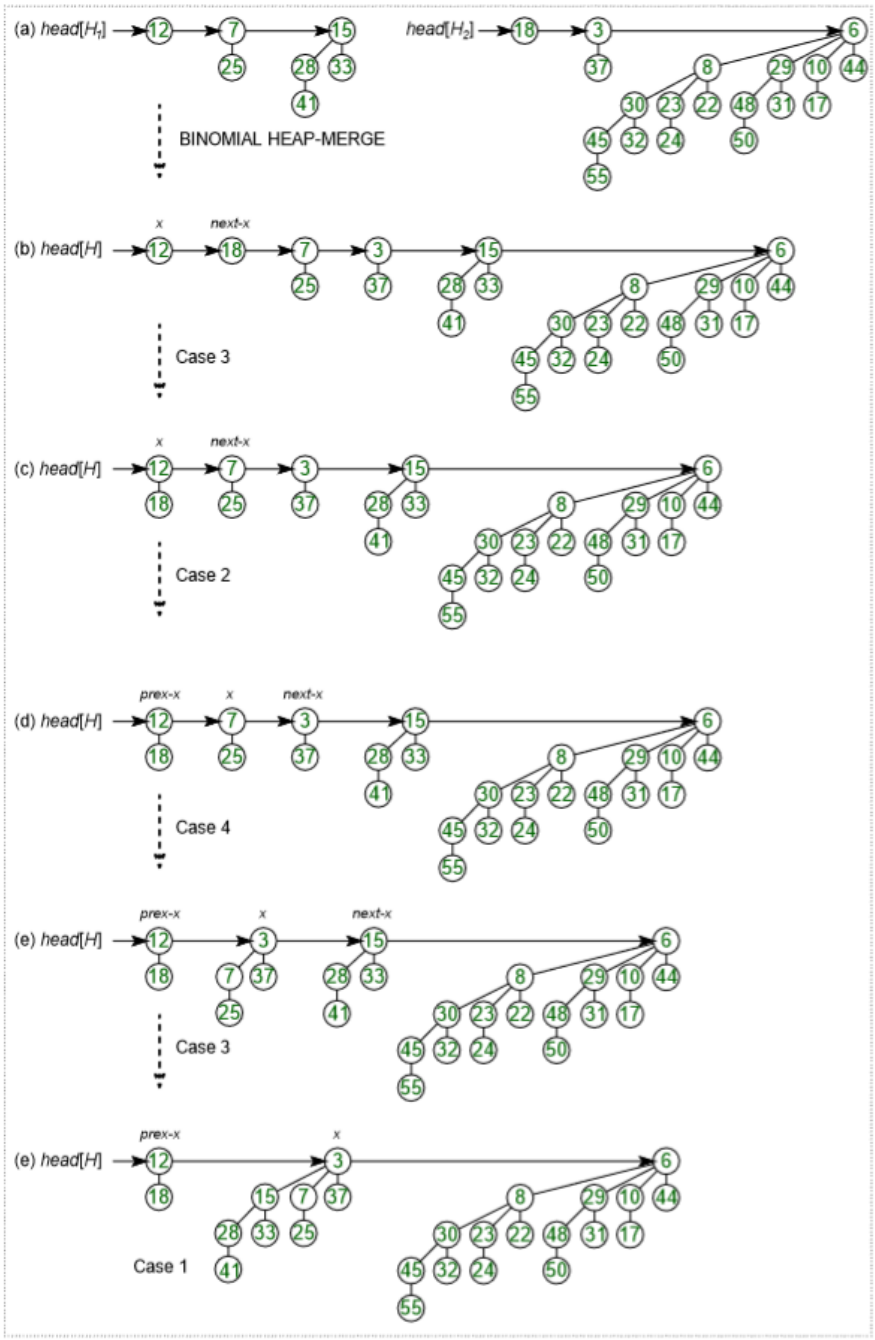
La procedura di unione unisce due heap in uno solo, dopo il merge dei due heap è necessario aggiustare la fusione facendo si che l'heap risultante rispetti anch'esso le proprietà degli heap. Si osservi che nella struttura risultante saranno presenti al più tre alberi di grado K, due di essi risultato del merge, mentre il terzo potrebbe essere frutto dell'unione di due alberi di grado k-1, di conseguenza, se due radici consecutive hanno grado diverso o vi sono tre radici di grado uguale scegliamo di proseguire nella lista, altrimenti procederemo con l'unione delle radici, fatti i dovuti accorgimenti.

UNIONHEAP(H1,H2)

1. H←MakeHeap()
2. head[H]←MergeHeap(H1,H2)
3. **if** head[H]=Nil
4. **then return** H
5. prev←Nil
6. x←head[H]
7. next←sibling[x]
8. **while** next≠Nil
9. **do if** degree[x]≠degree[next] or sibling[next]≠Nil and degree[sibling[next]]=degree[x]
10. **then** prev←x
11. x←next

```
12.        else if key[x]≤key[next]
13.           then sibling[x]←sibling[next]
14.              MergeTree(next,x)
15.        else if prev=Nil
16.           then head[H]←next
17.              MergeTree(x,next)
18.              x←next
19.        else then sibling[prev]←next
20.           MergeTree(x,next)
21.           x←next
22.     next←sibling[x]
23. return H
```

Complessità $\mathcal{O}(\log n)$


(a) head[H1]
head[H2]
BINOMIAL HEAP-MERGE
(b) head[H]
x
next-x
Case 3
(c) head[H]
x
next-x
Case 2
(d) head[H]
prex-x
x
next-x
Case 4
(e) head[H]
prex-x
x
next-x
Case 3
(e) head[H]
prex-x
x
Case 1

3.6 Inserimento di un nodo in un Heap

La procedura di inserimento crea un heap che ha come unico elemento il nodo da inserire, successivamente effettuerà l'unione con l'heap nella quale si vuole inserire l'elemento, poiché entrambi gli heap rispettano le proprietà l'unione sarà un heap valido. Si noti che un heap vuoto è comunque un heap valido.

INSERT(H,x)

1. H1←MakeHeap()
2. parent[x]←Nil
3. child[x]←Nil
4. sibling[x]←Nil
5. degree[x]←0
6. head[H1]←x
7. H←UnionHeap(H,H1)

Complessità $\mathcal{O}(\log n)$

3.7 Estrazione del minimo

La procedura si occupa di estrarre il minimo da un heap rimuovendolo dalla lista delle radici, dopodichè i suoi figli andranno inseriti in un nuovo heap facendo attenzione a mantenere inalterate le proprietà, infine verrà fatta l'unione di questi due heap, affinchè nessun elemento venga perso.

EXTRACTMINIMUM(H)

1. Trovare il nodo x con la key minore e rimuoverlo dalla lista delle radici
2. H1←MakeHeap()
3. Inserire i figli di x in H1 in modo che siano ordinati rispetto al grado
4. H←UnionHeap(H,H1)
5. **return** x

Complessità $\Theta(\log n)$

3.8 Decremento di una key

La procedura decrementerà il nodo specificato, in seguito alla diminuzione del valore è necessario accertarsi che le proprietà dei min-heap vengano rispettate, di conseguenza il nodo dovrà risalire l'albero finchè il suo valore è minore di quello del padre.
Si presupponga che la funzione swap utilizzata scambi le chiavi dei nodi e i loro dati satellite, se presenti.

DECREASEKEY(H,x,newk)

1. **if** newk>key[x] **return**
2. key[x]←newk
3. y←x
4. z←parent[x]
5. **while** z≠Nil and key[y]<key[z]
6. **do** swap(z,y)
7. y←z
8. z←parent[y]

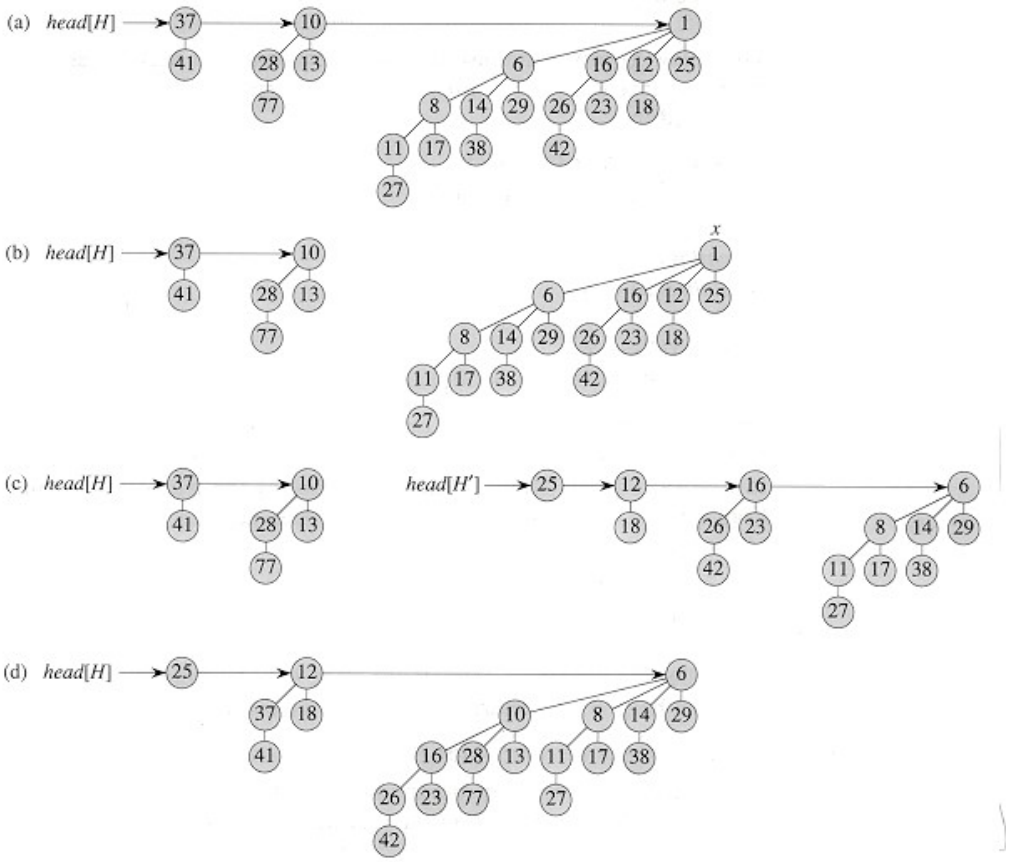
Complessità $\Theta(\log n)$

3.9 Cancellazione di un nodo

La procedura sfrutta la funzione di ExtractMinimum rendendo il nodo da eliminare il minore all'interno dell'heap, e poi estraendolo.
DELETEKEY(H,x)

1. DecreaseKey(H,x,-∞)
2. ExtractMinimum(H)

Complessità $\Theta(\log n)$



4. Implementazione

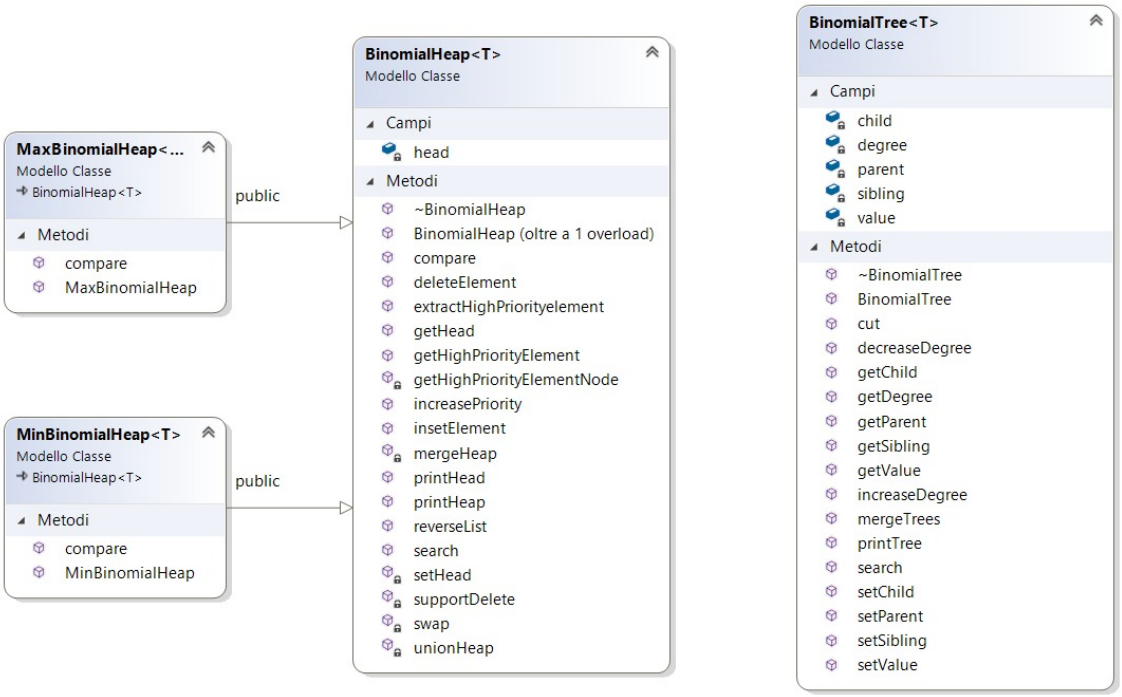
Si sceglie di utilizzare un approccio orientato agli oggetti, scomponendo il problema in più classi, questo tipo di approccio consente di rendere il codice più sicuro e stabile e inoltre permette di sfruttare l'ereditarietà per poter specializzare le classi in max-heap o min-heap.
Si sceglie inoltre di rappresentare un albero binomiale con una singola classe, poiché la natura dell'albero è di tipo ricorsivo, e quindi un nodo in se è un albero, evitando di creare una classe nodo ed una classe albero, rendendo il codice più leggibile e più semplice da scrivere e modificare.
La classe albero sarà dotata della key e dei puntatori a parent, child e sibling, e di tutti i metodi getter e setter necessari.
Nella classe binomial heap oltre tutte le funzioni già elencate sarà presente un puntatore alla testa dell'heap che formerà una lista concatenata con tutti i

nodi radice, in questo approccio si sceglie di concatenare direttamente le radici utilizzando il puntatore al fratello rendendo più facile lo scorrimento dell'heap e l'utilizzo dei nodi, di fatti tutte le radici nell'heap avranno il puntatore a padre nullo.
L'utilizzo della classe nodo rende molto semplice il merge di due alberi, poiché l'approccio utilizzato collega solamene le radici, di conseguenza quando si vorrà eliminare un nodo sarà sufficiente lavorare solo sui suoi figli, poiché i suoi nipoti non faranno parte di un'unica lista concatenata, questo semplifica di molto la procedura di delete, a differenza di molte implementazioni differenti nella quale vengono collegate tutte le liste di nodi, ciò comporta la necessità di aggiornare i puntatori ad ogni unione e cancellazione, una procedura troppo complicata con un vantaggio minimo ai fini della struttura dati.

4.1 Ulteriori scelte implementative

Poichè appare chiara la possibilità di generalizzare tutte le procedure all'interno degli heap, per così fare si sceglie di adottare uno standard che verrà anche ribadito in sede di scrittura di codice, ovvero si sceglie di vincolare tutte le procedure alla procedura di compare, così facendo oltre a generalizzare il codice, si consente a chiunque di implementare una versione diversa dell'heap semplicemente con una modifica o una riscrittura della procedura di compare.
lo standard che si sceglie di seguire è il seguente: se il primo parametro della compare che chiameremo X ha una priorità maggiore rispetto Y allora la compare ritornerà un intero k<0 altrimenti k>0.
Inoltre si sceglie di rendere la classe padre una classe astratta implementando la procedura di compare come virtuale pura, infine nel codice vengono riportati due esempi di ereditarietà con l'implementazione di un max-heap ed un min heap.

4.2 UML


MaxBinomialHeap<...
Modello Classe
BinomialHeap<T>
Metodi
compare
MaxBinomialHeap
public
MinBinomialHeap<T>
Modello Classe
BinomialHeap<T>
Metodi
compare
MinBinomialHeap
public
BinomialHeap<T>
Modello Classe
Campi
head
Metodi
~BinomialHeap
BinomialHeap (oltre a 1 overload)
compare
deleteElement
extractHighPrioritye lement
getHead
getHighPriorityElement
getHighPriorityElementNode
increasePriority
insetElement
mergeHeap
printHead
printHeap
reverseList
search
setHead
supportDelete
swap
unionHeap
BinomialTree<T>
Modello Classe
Campi
child
degree
parent
sibling
value
Metodi
~BinomialTree
BinomialTree
cut
decreaseDegree
getChild
getDegree
getParent
getSibling
getValue
increaseDegree
mergeTrees
printTree
search
setChild
setParent
setSibling
setValue

Riferimenti

Gli appunti e il materiale utilizzato sono stati reperiti in parte da queste fonti.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms, Third Edition" 2010, McGraw-Hill.

"Binomial Heap", Wikipedia, l'enciclopedia libera.

Prof. Cantone Domenico, corso di laurea magistrale, "Heap Binomiali" A.A. 2018/19.