

Algoritmi e Strutture Dati 2023-24

(M. Benerecetti)

Contents

1	Lezione 02 - 15/09/2023	4
1.1	Algoritmo di Conteggio	4
1.1.1	Algoritmo di conteggio v2	5
1.1.2	Algoritmo di conteggio v3	6
1.2	Algoritmo della massima sottosequenza contigua	6
1.2.1	Algoritmo v2	7
2	Lezione 03 - 21-09-2023	8
2.1	Algoritmo v3	8
2.2	Strutture Dati - Insieme Dinamico	8
3	Lezione 04 - 23/09/2023	9
3.1	Array non ordinato	9
3.1.1	Ricerca	9
3.1.2	Inserimento	9
3.1.3	Cancellazione	10
3.2	Array Ordinato	10
3.2.1	Ricerca Binaria	10
3.2.2	Inserimento/Cancellazione nella ricerca binaria	11
4	Lezione 05 - 26-09-2023	12
4.1	Ricerca Binaria Iterativa	12
4.2	Somma	12
4.3	Lista Linkata	13
4.3.1	Ricerca (iterativo)	13
4.3.2	Ricerca (ricorsiva)	13
4.3.3	Ragionamento Ricerca Binaria su Lista Ordinata	14
5	Lezione 06 - 29/09/2023	15
5.1	Inserimento	15
5.2	Inserimento Ordinato (iterativo)	15
5.3	Inserimento Ordinato (ricorsivo)	16
5.4	Cancellazione (iterativa)	16
5.5	Cancellazione (ricorsiva)	17

6	Lezione 07 - 29/09/2023	18
6.1	Albero Binario	18
6.2	Altezza di un Albero	18
6.3	Complessità Computazione	18
6.4	Visite	19
6.4.1	PreOrder	19
6.4.2	InOrder	19
6.4.3	PostOrder	20
6.5	Ricerca Albero Binario	20
7	Lezione 08 - 03/10/2023	21
7.1	Nomi Utilizzati delle visite	21
7.2	Visita In Ampiezza Albero	21
7.3	Albero Binario di Ricerca	21
7.3.1	Search Ricorsiva - BST	22
7.3.2	Min - BST	22
7.3.3	Algoritmo del Successore Ricorsivo - BST	23
7.3.4	Algoritmo del Successore Iterativo - BST	23
8	Lezione 9 - 05/10/2023	25
8.1	Successore Iterativo - BST	25
8.2	Predecessore Ricorsivo - BST	25
8.3	Insert Ricorsiva - BST	25
8.4	New Node - Generico	26
8.5	Insert Iterativa - BST	26
8.6	DeleteR - BST	27
9	Lezione 10 - 06/10/2023	29
9.1	Alberi Perfettamente Bilanciati	29
9.2	Alberi AVL	29
9.3	Alberi AVL Minimi	30
10	Lezione 11 - 10/10/2023	32
10.1	Altezza	32
10.2	Inserimento AVL	32
10.3	Bilanciamento	33
10.3.1	Bilanciamento Sinistro	33
11	Lezione 12 12/10/2023	34
11.1	Cancellazione albero AVL	34
11.1.1	DeleteAVL	34
11.1.2	DeleteAVL	34
11.1.3	Stacca_min	35
12	Lezione 13 - 13/10/2023	36
12.1	Alberi Red - Black	36
12.1.1	Altezza Nera di un albero R-B	36
12.1.2	Considerazioni sull'altezza nera di un albero	38

12.1.3 Inserimento Albero R-B	38
13 Lezione 14 17/10/2023	40
13.1 Bilanciamento All'inserimento in Albero R-B	40
13.2 Controllo Violazione a Sinistra Inserimento R-B	40
13.3 Risoluzione Violazione Inserimento a sinistra	40
13.4 Caso 1	40
13.5 Caso 2	41
13.6 Caso 3	42
13.7 Delete Albero R-B	42
14 Lezione 17 24/10/2023	44
14.1 Grafi	44
14.2 Grafo Orientato	44
14.3 Alberi e Grafi	45
14.4 Grafi Pesati	45
14.5 Grado di un Vertice	45
14.6 Percorsi	45
14.7 Raggiungibilita di un nodo	45
14.8 Percorsi Ciclici	46
14.9 Sottografo	46
15 Lezione 18 26/10/2023	47
15.1 Complessità sui grafi	47
15.2 Come Rappresentiamo un Grafo?	47
15.2.1 Matrice di bit	48
15.2.2 Liste di adiacenza	48
16 Lezione 19 - 27/10/2023	49
16.1 Visite Grafi	49
16.1.1 BFS	49
16.1.2 Variante BFS Tempo	50
17 Lezione 21 - 02/11/2023	51
17.1 Visita in Profondità - DFS	51
18 Lezione 22 - 03/11/2023	52
18.1 Dimostrazione Teorema delle parentesi	52
18.2 Teorema del percorso bianco	52
18.2.1 Dimostrazione Teorema - Condizione Necessaria \rightarrow	52
18.2.2 Dimostrazione Teorema - Condizione Sufficiente \leftarrow	52
18.3 I problemi dell'algoritmo DFS	53
18.4 Come aggiungere queste informazini all'algoritmo DFS	53
18.5 Proprieta archi di ritorno	54
18.6 Algoritmo del grafico Aciclico	54

1 Lezione 02 - 15/09/2023

1.1 Algoritmo di Conteggio

Descrivere un algoritmo che accetta come input un intero $N \geq 1$ e produce in output il numero di coppie ordinate $i, j \in \mathbb{N} \quad (i, j) : 1 \leq i \leq j \leq N$

Esempio:

- Input: $N=4$
- Output: 10 $\{(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)\}$

```
1 Conta(N):  
2   ris = 0; //Assegnamento costante (1 operazione elementare)  
3   for i=1 to N do //Assegnamento/Incremento + confronto (2 op.  
4       elementari)  
5       for j=1 to N do //idem for di sopra  
6           if i<=j then //2 letture + confronto (3 op. elementari)  
7               ris = ris+1 //lettura+scrittura+assegnamento (3 op. elementari)  
           return ris //1 op. elementare
```

Ogni riga ha un costo che corrisponde alle operazioni elementari effettuate, mediamente ogni op. elementari ha un costo di 1 unità di tempo, prendiamo come esempio il for al primo giro: Assegnamento + Confronto (2 op. elementari), invece i successivi giri: Incremento+confronto (2 op. elementari).

Ognuna di queste operazioni (righe) vengono eseguite più di una volta, quindi il costo sarà maggiore, andiamo ad esprimerlo:

- 2) Costo = 1 (fuori dal ciclo)
- 3) La testa viene eseguita $n + 1$ volte poiché abbiamo anche l'ultima operazione per uscire dal ciclo, quindi Costo = $2 * (n + 1) = 2 * \sum_{i=1}^{N+1} 1$
- 4) Questo for verrà ripetuto N volte poiché il corpo del for viene eseguito N volte, quindi il suo costo sarà:

$$\underbrace{2}_{\text{costo dell'operazione}} * \underbrace{\sum_{i=1}^N}_{\text{for esterno}} \underbrace{\sum_{j=1}^{N+1} 1}_{\text{for interno}}$$

- 5) L'if stando in entrambi i for avrà un costo di: $3 * \sum_{i=1}^N \sum_{j=1}^{N+1} 1$
- 6) Questa operazione non ha un numero fisso di volte di esecuzione. Pertanto è necessario stabilirne un algoritmo per decretarne il numero. Pensandoci il numero di volte che questa operazione esegue dipende da N e dall' i fissate in precedenza. Calcolando, anche banalmente a mano, quante operazioni vengono eseguite ci troveremo con:

$N - i + 1$ volte che l'operazione viene eseguita.

- 7) Costo = 1 (fuori dal ciclo)

Dopo che viene effettuata l'analisi, possiamo andare a sommare tutti i risultati che abbiamo ottenuto in termini di unita (correggere accento) di tempo. La funzione $T(n)$ e(correggere) la funzione che ci tiene traccia della complessita dell'algoritmo.

Andiamo semplicemente a sommare i nostri risultati di ogni riga:

$$T(n) = 1 + 2 * (N) + 2 * (N^2 + N) + 3 * \frac{N(N+1)}{2}$$

Questo risultato e ottenuto semplificando le nostre sommatorie:

- 3) $2 * \sum_{i=1}^{N+1} 1 = 2 * (N + 1)$
- 4) $2 * \sum_{i=1}^N \sum_{i=1}^{N+1} 1 = 2 * \sum_{i=1}^N N + 1 = 2 * (N^2 + N)$
- 5) $3 * \sum_{i=1}^N \sum_{i=1}^N 1 = 3 * \sum_{i=1}^N N = 3N^2$
- 6) $\sum_{i=1}^N (N-i+1) = N - (k-1)$ cioe ad ogni ciclo il numero delle volte che viene eseguita questa operazione diminuisce costantemente di 1 (cioe dipendente dal salire di i).

$$T(n) = \frac{13}{2}N^2 = \frac{9}{2}N + 4$$

Come vediamo questa funzione e quadratica, quindi cresce esponenzialmente nel tempo, molto pesante e lenta come funzione.

1.1.1 Algoritmo di conteggio v2

Dopo aver ottenuto i risultati dell'analisi sopra, possiamo dire che e sicuramente possibile semplificare il nostro codice in modo tale da far eseguire meno operazioni al nostro processore e quindi utilizzare meno tempo.

```

1 Conta(N):
2   ris = 0;
3   for i=1 to N do
4     ris = ris + (N-i+1)
5   return ris

```

Così facendo abbiamo semplicemente detto al nostro codice che deve sommare soltanto gli elementi che nel momento in cui i e fissato, sono \leq di se stesso.

Così facendo si dovrebbero eliminare molte operazioni inutili, analizziamo.

- 2) sempre 1 operazione
- 3) $2 * \sum_{i=1}^{N+1} 1$
- 4) $\sum_{i=1}^N 7 = 7 \cdot N$

Come possiamo osservare abbiamo eliminato il secondo for, dunque abbiamo eliminato la quadraticita, ora l'operazione a riga 4 viene eseguita solamente N volte, e il numero di operazioni semplici che esegue e fissato.

$$T(n) = 1 + 2(N + 1) + 7N + 1 = 9N + 4$$

1.1.2 Algoritmo di conteggio v3

Da come possiamo notare è possibile di nuovo semplificare l'ultima sommatoria della riga 4 dello scorso algoritmo.

Da

$$\sum_{i=1}^N N - i + 1 \rightarrow \sum_{i=1}^N i$$

Il risultato della sommatoria è lo stesso, se andiamo a semplificarlo.

$$\sum_{i=1}^N N - i + 1 = \sum_{i=1}^N i = \frac{N(N+1)}{2}$$

```
1 Conta(N):  
2   ris = 0  
3   ris = ris+(N-i+1)  
4   return ris
```

In questo modo abbiamo eliminato qualsiasi ciclo e quindi il risultato sarà un numero fisso di operazioni.

- 1) 1 operazione
- 2) 5 operazioni elementari

$$T(n) = 5 + 1 = 6$$

In questo caso la funzione tempo per eseguire queste operazioni è costante, non dipende da nessun N , dunque è la migliore soluzione possibile per questo algoritmo.

1.2 Algoritmo della massima sottosequenza contigua

Preso un array di n elementi, vorremmo provare a trovare la sottosequenza la cui somma di tutti i valori è massima.

Come fare? La soluzione più naïve possibile è effettuare 3 cicli for innestati:

```
1 int Max_seq_sum_1(int N, array a[])  
2   maxsum = 0  
3   for i=1 to N  
4     for j=i to N  
5       sum = 0  
6       for k=i to j  
7         sum = sum + a[k]  
8       maxsum = max(maxsum, sum)  
9   return maxsum
```

Come possiamo notare l'aver 3 for innestati ci fa avere una complessità computazionale di N^3 , comunemente rappresentato dalla formula della notazione asintotica $O(N^3)$.

1.2.1 Algoritmo v2

Come è facile notare, nel terzo for innestato c'è una ripetizione abbastanza inutile di operazioni che effettuiamo per andarci a sommare i valori delle sottosequenze. In particolare andiamo a ripetere scorrere più volte lo stesso numero di celle, solamente per trovare la sottosequenza poco più grande (a volte anche di una cella).

Gli stessi valori delle sottosequenze possono essere semplicemente trovati scorrendo avanti l'indice e sommando alla sequenza precedente il valore successivo. Analiticamente ci troveremmo che:

$$\sum_{k=i}^{j+1} a_k = a_{j+1} + \sum_{k=i}^j a_k$$

Il valore sum rimarrà inalterato, ma verrà solamente aggiornato del valore successivo. Il codice si presenterà in questo modo:

```
1 int Max_seq_sum_1(int N, array a[])
2   maxsum = 0
3   for i=1 to N
4     sum = 0
5     for j=i to N
6       sum = sum + a[j]
7       maxsum = max(maxsum, sum)
8   return maxsum
```

2 Lezione 03 - 21-09-2023

2.1 Algoritmo v3

L'algoritmo può essere anche migliorato, riuscendo ad arrivare ad una complessità **lineare**, nel seguente modo:

```
1 int Max_seq_sum_3(int N, array a[])
2   maxsum = 0
3   sum = 0
4   for j=1 to N
5     if (sum + a[j] > 0) then
6       sum = sum + a[j]
7     else
8       sum = 0
9     maxsum = max(maxsum, sum)
10  return maxsum
```

Il ragionamento è il seguente: Se prendiamo un insieme di numeri da sommare, (da i ad a), possiamo controllare se esso è positivo o negativo. Nel caso in cui $\sum_{e=i}^a A[e]$ risultasse positiva, andiamo a espandere il nostro range fintanto che il risultato della sommatoria rimanga positivo. Nel caso in cui invece il risultato fosse negativo, non ci conviene tenere traccia dei numeri più piccoli di quel range, dato che se quella sommatoria è minore del numero successivo alla sommatoria, non ha senso tenerne conto. E quindi invece ha senso tenere traccia del numero successivo. Da quel numero poi sommare i numeri successivi continuando il processo sopracitato.

2.2 Strutture Dati - Insieme Dinamico

Vediamo come rappresentare un insieme di dati dinamico S (con insieme dinamico si intende una collezione di elementi variabile nel tempo, quindi è possibile aggiungere o rimuovere elementi);

$$S = \{a_1, a_2, \dots, a_n\} \quad n \geq 0$$

Andiamo a definire alcune operazioni:

- $\text{Insert}(S, a) \rightarrow S'$ ($S' = S \cup \{x\}$)
- $\text{Deletes}(S, a) \rightarrow S'$ ($S' = S \setminus \{x\}$)
- $\text{Search}(S, a) \rightarrow \{True, False\}$
- $\text{Massimo}(S) \rightarrow a$
- $\text{Minimo}(S) \rightarrow a$
- $\text{Successore}(S, a) \rightarrow a'$
- $\text{Predecessore}(S, a) \rightarrow a'$

3 Lezione 04 - 23/09/2023

3.1 Array non ordinato

Abbiamo un insieme S che vogliamo rappresentare in A .

$$|S| = A.free$$

La funzione $A.free$ ci restituirà la posizione della prima cella libera.

3.1.1 Ricerca

Dato un array e un elemento da cercare, restituisce la posizione in cui è presente il valore

```
1 Search(A,e) //A: array in input, e: elemento da cercare
2   pos = A.free-1
3   while A[pos] != e and pos >= 0 do
4       pos = pos-1
5   return pos
```

Ci posizioniamo all'ultima cella piena e mano a mano tornando all'indietro andiamo a cercare il valore che abbiamo in input, se non è presente nell'array andiamo a restituire il valore -1 . La complessità computazione sarà **lineare** nello specifico:

$$T_s(n) = c \cdot n$$

Dove c è il costo fisso delle operazione e n è quante volte si ripete il ciclo.

3.1.2 Inserimento

Dato un array e un elemento da inserire, andiamo a verificare tramite la funzione **ricerca** se l'elemento non sia già presente poiché non vogliamo elementi duplicati.

```
1 Inserimento(A,e) //A: array in input, e: elemento da cercare
2   pos = search(A,e) //-1: non trovato, >=0: indice del valore
3   if pos = -1 then
4       if(A.free < length(A)) then
5           A=resize(A)
6           A[A.free]=e
7           A.free=A.free+1
```

La complessità computazione sarà **lineare** nello specifico:

$$T_i(n) = 2c \cdot n + c'$$

3.1.3 Cancellazione

Dato un array e un elemento da cancellare, andiamo a verificare tramite la funzione **ricerca** se l'elemento sia presente in modo da portelo eliminare.

```
1 Delete(A,e) //A: array in input, e: elemento da cercare
2   pos = search(A,e) //-1: non trovato, >=0: indice del valore
3   if pos >= 0 then //Elemento trovato
4       A[pos]=A[A.free-1]
5       A.free=A.free-1
```

3.2 Array Ordinato

Uno dei grosso problemi dell'array visto in precedenza era la ricerca, poiché al più avevo costo **lineare** cioè la lunghezza di tutto l'array, con l'array ordinario andiamo a sopperire a questo problema ma ovviamente aggiungendone degli altri.

3.2.1 Ricerca Binaria

La ricerca binaria è un algoritmo applicabile solo ad array ordinato che permette una ricerca molto rapida.

L'algoritmo è simile al metodo usato per poter trovare una parola sul dizionario: sapendo che il vocabolario è ordinato alfabeticamente, l'idea è quella di iniziare la ricerca non dal primo elemento, ma da quello centrale, cioè a metà del dizionario. Si confronta questo elemento con quello cercato:

- se corrisponde, la ricerca termina indicando che l'elemento è stato trovato;
- se è superiore, la ricerca viene ripetuta sugli elementi precedenti (ovvero sulla prima metà del dizionario), scartando quelli successivi;
- se invece è inferiore, la ricerca viene ripetuta sugli elementi successivi (ovvero sulla seconda metà del dizionario), scartando quelli precedenti.

Andiamo a definirlo per ricorsione:

```
1 BinSearch(A, e, i, j) //i: punto inizio, j: punto fine
2   if i <= j then
3       q=(i+j)/2 //punto centrale
4       if A[q] > e then
5           i=BinSearch(A, e, i, q-1) //ricerca a "sinistra"
6       else if A[q] < e then
7           i=BinSearch(A, e, q+1, j) //ricerca a "destra"
8       else return q //trovato
9   else return -1 //non trovato
```

Per studiare la complessità di questo algoritmo (ricorsivo), andiamo a rappresentare le varie chiamate tramite un "albero degenero" in cui ogni nodo ha un solo figlio e così via, il costo di ogni nodo sarà c , ad ogni chiamata andiamo a dimezzare la lunghezza dell'array $n \dots n/2 \dots n/4 \dots$ il massimo delle chiamate possono essere $h + 1$ dove h è

l'altezza dell'albero andiamo a generalizzare con:

$$\frac{n}{2^h} \quad h : \text{è il numero dei richiami}$$

Dobbiamo trovare il valore di h per il quale l'intervallo si riduce a 1 elemento, in quanto l'elemento desiderato sarà stato trovato. Quindi, dobbiamo risolvere l'equazione:

$$\frac{n}{2^h} = 1$$

Per risolvere questa equazione per h , possiamo moltiplicare entrambi i lati per 2^h :

$$n = 2^h$$

Ora, per isolare h , possiamo applicare il logaritmo in base 2 ad entrambi i lati:

$$h = \log_2(n)$$

Quindi, il numero di chiamate ricorsive necessarie per trovare l'elemento desiderato è logaritmico rispetto alla dimensione dell'array n . Pertanto, la complessità computazionale dell'algoritmo di ricerca binaria è $O(\log n)$ nel caso peggiore.

N.B Una complessità computazionale logaritmica è più efficiente di una lineare.

3.2.2 Inserimento/Cancellazione nella ricerca binaria

Abbiamo visto come in un array ordinato la ricerca (binario) è molto efficiente ma abbiamo come contro che l'inserimento/cancellazione hanno costo peggiore poiché dobbiamo mantenere l'ordinamento, quindi se vogliamo inserire un elemento dobbiamo andare a spostare i valori per creare un posto disponibile.

4 Lezione 05 - 26-09-2023

4.1 Ricerca Binaria Iterativa

Proviamo a riscrivere la ricerca binaria su un array ordinato in maniera iterativa:

```
1 BinSearchIterative(A,e) //A: array in input, e: elemento da cercare
2   ret=-1
3   i=0
4   j=length(A)-1
5   while i <=j AND ret=-1 do
6     q=(i+j)/2 //punto medio
7     if A[q] < k then
8       i=q+1 //ricerca a "destra"
9     else if A[q] > k then
10      j=q-1 //ricerca a "sinistra"
11   else
12     ret=q //trovato
13   return ret
```

Possiamo notare che a meno di piccoli cambiamenti il funzionamento di questo algoritmo rispetto alla sua versione ricorsiva è pressoché identico.

4.2 Somma

Andiamo a creare un algoritmo per sommare tutti i valori presenti in un array tramite ricorsione.

Come tutti gli algoritmi ricorsivi dobbiamo andare a trovare una base di induzione (caso base) e poi un passo di induzione, rappresentiamolo tramite sommatorie:

$$\sum_{i=1}^n i = \begin{cases} 0 & \text{se } n = 0 \text{ base induzione} \\ n + \sum_{i=1}^{n-1} i & \text{se } n \geq 1 \text{ passo induttivo} \end{cases}$$

Ora che abbiamo capito il ragionamento andiamo a scrivere l'algoritmo

```
1 Sum(A,i,n) // i=inizio, n=fine
2   if n=0 then
3     ret = 0
4   else
5     x=sum(A,i,n-1)
6     ret=n+x
7   return ret
```

4.3 Lista Linkata

La lista rappresenta una struttura dati dinamica dove le operazioni di inserimento e cancellazione sono meno dispendiose, a differenza di quanto accade negli array (che sono implementati come struttura statica, il che rende problematiche le suddette operazioni).

La lista condivide con l'array la proprietà di linearità (o sequenzialità) ma è una struttura più flessibile poiché non richiede la contiguità in memoria come l'array. La lista permette di avere gli elementi in una qualsiasi area di memoria rendendo le operazioni di inserimento e cancellazione eseguibili in tempo costante; infatti, la sua struttura è composta da nodi, i quali contengono sia un certo dato (key), sia un'informazione su dove si trovi il nodo successivo (next).

4.3.1 Ricerca (iterativo)

Andiamo a definire un algoritmo di ricerca su lista linkata tramite iterazione

```
1 Search(L,k) // L=lista, k=elemento da cercare
2   ris=-1
3   temp=L // "puntatore" al primo elemento della lista
4   while temp != NIL and ris=-1 do
5       if temp->key = k then
6           ris=temp
7       else
8           temp = temp->next
9   return ris
```

L'algoritmo è molto semplice tramite la variabile "temp" andiamo a scorrerci tutta la lista tramite i puntatori **next**, se troviamo il valore lo restituiamo il nodo in cui è presente altrimenti restituiamo -1.

4.3.2 Ricerca (ricorsiva)

Andiamo a rifare lo stesso algoritmo ma tramite la ricorsione

```
1 Search(L,k) // L=lista, k=elemento da cercare
2   ris=NIL
3   if L != NIL then
4       if L->Key = k then
5           ris=L
6       else
7           ris=search(L->Next, k)
8   return ris
```

In questo caso ad ogni prossima chiamata di "search" andiamo a passare una "sottolista" cioè la stessa ma partendo da un nodo più avanti fino ad arrivare alla sua fine.

Andiamo a calcolare la complessità computazionale, dato che ad ogni chiamata andiamo a passare la lista ma ridotta di un nodo avremo $n, n-1, n-2, n-i$ dove i sarà il numero di nodi, quindi abbiamo la sequenza dei primi i numeri naturali (formula di

Gauss) quindi l'algoritmo ha complessità **lineare**.

4.3.3 Ragionamento Ricerca Binaria su Lista Ordinata

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

5 Lezione 06 - 29/09/2023

Continuiamo i nostri algoritmi sulle liste linkate

5.1 Inserimento

Andiamo a scrivere un algoritmo per l'inserimento in testa, cioè andiamo ad "attaccare" il nostro nuovo nodo all'inizio della lista, è il tipo di inserimento più semplice perché non prevede particolari modifiche alla struttura.

Funzione Ausiliaria Useremo una funzione ausiliare per aiutarci in tutte le situazioni di creazioni di un nuovo nodo:

```
1 newNode(K,L) //K: Elemento, L: Lista
2   temp = allocanodo() //tipo una malloc
3   temp->key = k //assegnamento valore
4   temp->next = L //attacciamo il nodo al primo elemento della lista
   in input
5   return temp
```

Ora che abbiamo definito la nostra funzione ausiliaria andiamo a scrivere l'algoritmo di inserimento

```
1 Insert(L,K)
2   ret = search(L,K) //cerchiamo il valore
3   if ret=NIL then //se non e' presente lo inseriamo
4       L=newNode(K,L)
5   return L
```

5.2 Inserimento Ordinato (iterativo)

Per strutturare al meglio le liste possiamo prevedere un inserimento ordinato, fare questo nelle liste è molto più efficiente rispetto a un array poiché non dobbiamo spostare tutti i valori per fare spazio al valore da inserire ma possiamo banalmente staccare i puntatori e "riattaccarli" nel modo corretto:

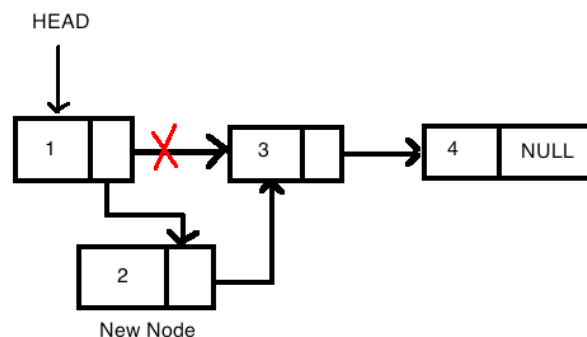


Fig. Insert after a given Node

Andiamo a definire l'algoritmo in maniera iterativa:

```
1 InsertO(L,K)
2   temp=L //salviamo il puntatore al primo elemento
3   P=NULL //creiamo una variabile di appoggio
4   while [temp != NULL and temp->key < k] do
5       P=temp //mettiamo P sul nodo "precedente" a dove inseriamo
6       temp=temp->next //mettiamo temp sul nodo "successivo"
7
8   if [temp = NULL or temp->key > k] then
9       new=newNode(k,temp) //creiamo il nodo attaccandolo al "successivo"
10      if P != NULL then //se "esiste" il precedente allora
11          P->next = new //attacciamo il "precedente" al nuovo
12      else //se non esiste
13          L=new //spostiamo la testa al nuovo valore
14
15  return L
```

5.3 Inserimento Ordinato (ricorsivo)

Come sempre per scrivere un buon algoritmo ricorsivo bisogna ragionare per casi, andiamo ad esaminare i vari possibili:

- Inserimento in testa (valore minimo)
- Inserimento "centrale" (valore compreso tra due numeri)
- Inserimento in coda (valore massimo)

Sulla base di ciò andiamo a scrivere il nostro algoritmo:

```
1 InsertOR(L,K)
2   if L=NULL then //Inserimento in coda
3       L=newNode(K,L)
4   else if [L->key > K] then //Inserimento in testa
5       temp=L
6       L=newNode(K, temp)
7   else //Inserimento "centrale"
8       L->next = InsertOR(L->next, K) //da esaminare
9   return L
```

5.4 Cancellazione (iterativa)

Andiamo a scrivere un algoritmo di cancellazione simile a quello visto per l'inserimento, cioè andiamo a cercare il valore e ci posizioniamo sia "prima" che "dopo" il valore da cancellare.


```

1 Delete(L,K)
2   temp=L //salviamo il puntatore al primo elemento
3   P=NULL //creiamo una variabile di appoggio
4   while [temp != NULL and temp->key != k] do
5       P=temp //mettiamo P sul nodo "precedente" a quello da cancellare
6       temp=temp->next //mettiamo temp sul nodo da cancellare
7
8   if temp != NULL then
9       if P != NULL then
10          P->next=temp->next
11      else
12          temp=L
13          L=L->next
14      dealloca(temp)
15  return L

```

5.5 Cancellazione (ricorsiva)

Ragioniamo per casi anche se in questo caso sono solo i due banali, cioè il valore è presente oppure no, nello specifico noi andiamo a considerare sempre la testa della lista che piano a piano a decrementarsi fino ad arrivare ad essere vuota

```

1 Delete(L,K)
2   if L!=NULL then //La lista ha almeno un valore
3       if L->key=k then //elemento in testa
4           temp=L
5           L=L->next
6           dealloca(temp)
7       else //elemento non in testa, "spostiamo" la testa in avanti
8           L->next = delete(L->next, k)
9   return L

```

6 Lezione 07 - 29/09/2023

6.1 Albero Binario

L'albero binario è una particolare struttura dati composto da nodi, ogni nodo ha tre campi:

- Valore
- Figlio Sinistro (puntatore)
- Figlio Destro (puntatore)

Ogni nodo può avere al più due figli, il primo nodo è chiamato **radice** e i nodi finali (senza figli) sono chiamati **foglie**.

Immagine

Andiamo a dare una definizione più corretta:

$$T \begin{cases} 1) T = NIL \\ 2) T = \text{nodo} + \text{sottoalbero} \end{cases}$$

6.2 Altezza di un Albero

Definiamo l'altezza di un albero come la **quantità** di nodi che scorreremo per raggiungere il nodo desiderato. Nota bene: Ad ogni livello (di altezza) il numero di nodi dell'albero aumenta esponenzialmente con un ritmo di

$$2^i$$

. Questo vale solo per un albero bilanciato. Osserveremo la sua complessità tra poco.

6.3 Complessità Computazione

Un albero binario può essere degenerare, cioè un albero i cui nodi hanno solo un figlio. Questo tipo di albero, come si vede in foto, creano in un certo senso una lista dinamica. Infatti questo tipo di albero condividerà la stessa complessità computazionale di una lista ordinata.

Analizziamo l'equazione che definisce quanti nodi ha ogni livello di un albero.

$n = \sum_{i=0}^n 2^i$ Questa sommatoria è risolvibile con una serie.

$$n = \sum_{i=0}^k x^i = \frac{x^{k+1} - 1}{x - 1}$$

Diventerà nel nostro caso:

$$n = \sum_{i=0}^n 2^i = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1$$

Il risultato che abbiamo ottenuto è il valore della tabellina del 2^{n+1} , a meno di un numero per il -1 .

Per andare dunque a calcolare la complessità computazione per la creazione di ogni nodo per ogni grado d'altezza dell'albero è dunque utile andare a prendere il risultato di questa funzione qui sopra scritta e risolverlo.

$$n = 2^{n+1} - 1 \rightarrow \log_2 n = \log_2 2^{n+1} - 1 \rightarrow \rightarrow$$

Sapendo che $n = 2^{n+1} - 1$ allora potremmo sostituirlo all'interno della seconda parte dell'equazione, facendo in modo che l'equazione diventi

$$\log_2 n = \log_2 n$$

Andiamo a ragionare su quando possa valere $2^{n+1} - 1$. Andiamo a considerare in quale spazio esso può essere compreso (a seconda dell'altezza h): $2^h \leq 2^{n+1} - 1 \leq 2^{n+1}$. In questo caso potremmo dire che sicuramente esso sarà più grande di 2^h quindi andiamo a suddividere la disequazione per trovarci la h .

$$\log_2 2^h \leq \log_2 2^{h+1} - 1 \rightarrow h = \log_2 2^{h+1} - 1$$

Ma per quanto abbiamo visto prima:

$$(n = 2^{h+1} - 1) \text{ e in questo modo avremo che : } \rightarrow n = \log_2 n$$

E questa sarà la nostra complessità dell'algoritmo.

6.4 Visite

Esistono vari tipi di visita ognuna con i suoi pregi e difetti

6.4.1 PreOrder

L'algoritmo di visita pre-order è un particolare algoritmo usato per l'esplorazione in profondità dei nodi di un albero. L'esplorazione dell'albero parte dalla radice per poi scendere alle foglie, prima il sotto albero sinistro poi quello destro, che sono gli ultimi nodi ad essere visitati.

```

1 PreOrder(T) :
2   T != NIL then
3     Visita(T->key) //funzione qualsiasi
4     PreOrder(T->sx)
5     PreOrder(T->dx)
```

6.4.2 InOrder

L'algoritmo di visita in-order è un particolare algoritmo usato per l'esplorazione in profondità dei nodi di un albero binario. In questo tipo di visita, per ogni nodo, si esplora prima il sottoalbero sinistro poi si visita il nodo corrente ed infine si passa al sottoalbero destro. Più precisamente, l'algoritmo esplora i rami di ogni sottoalbero fino ad arrivare alla foglia più a sinistra dell'intera struttura, solo a questo punto si accede al nodo. Terminata la visita del nodo corrente si procede poi con l'esplorazione del sottoalbero a destra, visitando sempre i nodi a cavallo dell'esplorazione del sottoalbero sinistro e quello destro.

```

1 InOrder(T) :
2   T != NIL then
3     InOrder(T->sx)
4     Visita(T->key) //funzione qualsiasi
5     InOrder(t->dx)

```

6.4.3 PostOrder

L'algoritmo esplora i rami dell'albero fino ad arrivare alle foglie prima di accedere ai singoli nodi, ad esempio si supponga di essere alla ricerca di alcuni oggetti molto pesanti e che per trovarli si debba esplorare un sentiero che comporta diverse diramazioni (albero).

Essendo gli oggetti pesanti non conviene raccogliarli subito e portarli con sé lungo il cammino ma conviene prima esplorare tutto il territorio quindi prelevarli quando si torna indietro.

L'algoritmo post-order visita un albero nello stesso modo, arrivando prima più in fondo possibile ad ogni diramazione ed accedendo agli elementi solamente al ritorno, il che corrisponde ad un'implementazione ricorsiva al fronte di risalita della ricorsione.

```

1 PostOrder(T) :
2   T != NIL then
3     PostOrder(T->sx)
4     PostOrder(t->dx)
5     Visita(T->key) //funzione qualsiasi

```

6.5 Ricerca Albero Binario

Andiamo a definire la ricerca su l'albero binario, la miglior visita per eseguire una ricerca è la **preorder** poiché è la visita che subito controlla i nodi senza aspettare la visita dei sottoalberi.

```

1 Search(T,k) :
2   ret=T
3   if T != NIL then
4     if T->key != k then //se il valore non c'è in testa
5       ret=Search(T->sx, k) //allora cerchiamo nel sottoalbero sinistro
6     if ret=NIL then //se non c'è a sinistra
7       ret=Search(T->dx,k) //cerchiamo a destra
8   return ret

```

7 Lezione 08 - 03/10/2023

7.1 Nomi Utilizzati delle visite

Vi sono due tipo di visita che analizziamo per gli alberi:

- **DFS** o visite in **profondità** sono i tipi di visite che, in generale, tendono a visitare prima i figlio dei padri, si sviluppano "in verticale". Sono intrinsecamente ricorsivi.
- **BFS** o visite in **ampiezza** sono i tipi di visite che tendono ad andare per livelli, visitando i nodi che sono tra loro "fratelli" e "cugini".

7.2 Visita In Ampiezza Albero

Come già accennato, la visita in ampiezza di un albero prevede la visita dei nodi in modo "orizzontale". Questa visita di prim'occhio potrebbe sembrare piu complessa, poiche rispetto alle altre che abbiamo visto fino ad ora non è possibile effettuarla con un algoritmo ricorsivo/iterativo e basta. Infatti, per questa visita sarà necessaria una struttura dati astratta già studiata in precedenza: **la coda**. Iniziando la visita dalla radice, inseriremo i figli del nodo corrente all'interno della coda (se presenti), per poi proseguire la visita con un nodo estratto dalla coda, che diventa così il nodo corrente, e così via, fin quando la coda non sarà vuota.

```
1 BFS(T) :  
2   Q=NIL  
3   Q=Accoda(Q,T)  
4   while(Q!=NIL) DO  
5       X=Testa(Q)  
6       "Visita del nodo"  
7       Q=Accoda(Q,X->sx)  
8       Q=Accoda(Q,X->dx)  
9       Q=Decode(Q) //Toglie della coda il primo
```

Ma quanto può essere complesso fare una cosa del genere? Le operazioni all'interno e prima del while sono a tempo costante, poichè sono operazioni di assegnazione. Il numero di volte in cui viene eseguito il while, però, dipende dalla quantità di nodi all'interno dell'albero, dunque è **lineare sul numero di nodi**. Per quanto riguarda la memoria occupata dalla coda, invece:

- Nel caso migliore l'albero è degenere (ogni nodo ha un solo figlio ad eccezione della foglia), nella coda sarà presente al più un nodo per volta, quindi è un costo di memoria **costante**.
- Nel caso migliore l'albero è bilanciato (ogni nodo ha 2 figli ad eccezione delle foglie), il momento in cui vi sarà più memoria utilizzata sarà quando tutte le foglie si troveranno in coda: circa $n/2$ nodi. Cioè nel caso più sfortunato si occuperà memoria **lineare sul numero di nodi**.

7.3 Albero Binario di Ricerca

E' un tipo di albero ordinato. Preso qualsiasi nodo, si ha che il suo dato è:

- Maggiore o uguale a tutti i dati contenuti nei nodi del suo sottoalbero sinistro
- Minore o uguale a tutti i dati contenuti nei nodi del suo sottoalbero destro

Il suo acronimo è **BST** (Binary Search Tree). L'algoritmo di ricerca per questo albero è uno dei più efficienti.

7.3.1 Search Ricorsiva - BST

```

1 SearchBSTr(T,k)
2   ret=T
3   if T!= NIL then
4     if T->key < k then
5       ret = SearchBSTr(T->dx, k)
6     else if T->key > k then
7       ret = SearchBSTr(T->sx, k)
8   return ret

```

Questo algoritmo sicuramente ci permetterà di cercare in modo più efficiente un dato, poichè, un po' come la ricerca binaria (ma non propriamente), ci permette di andare a "dimezzare" l'insieme di ricerca ad ogni confronto.

Esiste anche la versione iterativa di questo algoritmo:

```

1 SearchBSTi(T,k)
2   Tmp = T
3   while Tmp != NIL && Tmp-> key != k do
4     if Tmp->key < k then
5       Tmp=Tmp->dx
6     else
7       Tmp=Tmp->sx
8   return Tmp

```

Proprio come l'algoritmo ricorsivo andremo a dividere l'albero in due, ma ci avvarremo di un puntatore temporaneo per risalire nell'albero (questo puntatore viene memorizzato implicitamente nella versione ricorsiva all'interno dello stack di attivazione).

7.3.2 Min - BST

La ricerca del minimo in un albero binario di ricerca è molto semplice. Seguendo la regola secondo cui "l'elemento più piccolo si trova a sinistra", andremo a scendere a sinistra finchè sarà possibile, l'elemento che non ha figlio sinistro conterrà il dato più piccolo dell'albero.

```

1 MinR(T)
2   ret = T
3   if ret->sx != NIL then
4     ret= MinR(ret->sx)
5   return ret

```

7.3.3 Algoritmo del Successore Ricorsivo - BST

Il successore di un numero è il primo numero più grande dello stesso. Non è detto che un successore sia presente all'interno dell'albero (caso in cui tutti i valori nell'albero sono minori o uguali al valore di cui si cerca il successore) L'algoritmo diventa un po' più complesso: vi sono le 3 casistiche:

- Caso **T->key = k**. L'elemento successore si troverà sicuramente nel sottoalbero destro, poichè cerchiamo un valore più grande. In particolare, è il più piccolo elemento del sottoalbero destro, quindi **Min(T->dx)**
- Caso **T->key < k**. L'elemento successore si troverà sicuramente nel sottoalbero destro, poichè cerchiamo un valore più grande. Non sappiamo nient'altro però, quindi eseguiamo l'algoritmo sulla radice del sottoalbero destro: **Succ(T->dx)**.
- Caso **T->key > k**. L'elemento successore potrebbe trovarsi nel sottoalbero sinistro, poichè il valore corrente è più grande, quindi eseguiamo l'algoritmo sulla radice del sottoalbero sinistro **Succ(T->sx, k)**, se la ricerca non dovesse andare a buon fine, allora il valore corrente T è il successore.

```
1 SuccR(T,k)
2 ret = NIL
3 if ret != NIL then
4     if ret->key = k then
5         ret = Min(ret->dx)
6     else if ret->key < k then
7         ret = SuccR(ret->dx,k)
8     else
9         ret = SuccR(T->sx,k)
10    if ret= NIL then
11        ret = T
12 return ret
```

7.3.4 Algoritmo del Successore Iterativo - BST

Per questo tipo di algoritmo, dobbiamo ragionare in modo diverso. E' necessario fare controlli che nella versione ricorsiva vengono omessi poichè queste informazioni vengono ricavate dal punto di ripresa dell'esecuzione di ogni chiamata. Bisogna fare particolare attenzione al caso in cui il valore di cui vogliamo il successore non ha figli destri, poichè è necessario risalire fino a trovare un padre "da sinistra" (la terza casistica). Nell'algoritmo il valore tmp viene usato per visitare l'albero. Quando si scende a sinistra, viene salvato prima il nodo corrente nella variabile ret, poichè se non si trova il successore in quel sottoalbero sinistro, allora è il nodo salvato in ret il successore.

```
1 SuccI(T,k)
2 Tmp = T
3 ret = NIL
4 while Tmp != NIL andd Tmp->key != k then
5     if Tmp->key < k then
6         Tmp = Tmp->dx
```

```
7     else
8         ret = Tmp
9         Tmp = Tmp->sx
10    if Tmp != NIL && Tmp->dx != NIL then
11        ret = Min(Tmp->dx)
12    return ret
```


8 Lezione 9 - 05/10/2023

8.1 Successore Iterativo - BST

Per questo tipo di algoritmo, dobbiamo ragionare in modo diverso. In questo caso non iterativo non possiamo permetterci di omettere determinati controlli a posteriori. In particolare il controllo nel caso in cui il valore di cui vogliamo il successore non ha figli destri ed è una foglia. In questo caso particolare non abbiamo la possibilità di risalire a ritroso ricorsivamente ma dobbiamo tenere traccia ogni volta che il nodo scende a sinistra, segnandoci il puntatore di quest'ultimo.

```
1 SuccI(T,k)
2   Tmp = T
3   ret = NIL
4   while Tmp != NIL and Tmp->key != k then
5     if Tmp->key < k then
6       Tmp = Tmp->dx
7     else
8       ret = Tmp
9       Tmp = Tmp->sx
10    if Tmp != NIL && Tmp->dx != NIL then
11      ret = Min(Tmp->dx)
12  return ret
```

8.2 Predecessore Ricorsivo - BST

L'algoritmo del predecessore è simile al successore strutturalmente parlando ma invertendo segni e qualche operazione

```
1 PredR(T,k)
2   ret = NIL
3   if ret != NIL then
4     if ret->key = k then
5       ret = Max(ret->sx)
6     else if ret->key < k then
7       ret = PredR(ret->dx,k)
8     else
9       ret = PredR(T->sx,k)
10    if ret = NIL then
11      ret = T
12  return ret
```

8.3 Insert Ricorsiva - BST

L'algoritmo dell'inserimento in un albero in un albero binario di ricerca può vantare del fatto che è più facile trovare il nodo nel quale si può aggiungere il valore che abbiamo in input alla funzione. Ci basterà semplicemente scorrere a destra o a sinistra

il nostro puntatore per poi arrivare nel primo punto NIL favorevole e "returnare" a cascata i puntatori dei padri.

```
1 InsertR(T,k)
2 ret = T
3   if T = NIL then
4       ret = new_node(k)
5   else if T->key < k then
6       T->dx = InsertR(T->dx,k)
7   else if T->key > k then
8       T->sx = InsertR(T->sx,k)
9   return ret
```

8.4 New Node - Generico

La funzione new node va a creare un nuovo nodo dinamico all'interno dell'albero.

```
1 new_node(k)
2   ret = alloca_nodo() //andiamo a restituire il puntatore a nuovo
3                       nodo allocato in memoria a ret
4   ret->key = k
5   ret->sx = NIL
6   ret->dx = NIL
7   return ret
```

8.5 Insert Iterativa - BST

Versione iterativa della insert prevede dei controlli in piu per quanto riguarda la ricerca e inserimento. In questo caso specifico abbiamo bisogno di un puntatore in piu che ci segue nello scorrimento, chiamato **P** e sta a indicare il Padre del nodo a cui stiamo scorrendo.

```
1 InsertI(T,k)
2 ret = T
3 P = NIL
4 Tmp = T
5 while Tmp != NIL && Tmp->key != k do
6     P = Tmp
7     if Tmp->key < k then
8         Tmp = Tmp->dx
9     else
10        Tmp = Tmp->sx
11
12 if Tmp = NIL then
13     x = new_node(k)
14     if P = NIL then
15         ret = x
```

```

16  else if P->key < k then
17      P->dx = x
18  else
19      P->sx = x
20  return ret

```

8.6 DeleteR - BST

La delete prevede la delete del nodo e la restituzione dell'albero con quel nodo mancante. A primo acchitto non sembra un'operazione così difficile ma dobbiamo come sempre andare a ragionare per casi.

- Caso albero vuoto. In questo caso dobbiamo semplicemente restituire T, il puntatore (vuoto) alla radice dell'albero
- Caso albero non vuoto. In questo caso la radice del sottoalbero ha un sottoalbero destro e sinistro.
 - Se $T \rightarrow \text{key} < k$ then Delete($T \rightarrow \text{dx}, k$)
 - Se $T \rightarrow \text{key} > k$ then Delete($T \rightarrow \text{sx}, k$)
 - Se $T \rightarrow \text{key} = k$, dobbiamo distinguere dei casi
 - * Nel caso in cui il nodo non ha figli allora, si può procedere all'eliminazione del nodo.
 - * Nel caso in cui il nodo o ha figlio destro o figlio sinistro e basta, questo diventerà la nuova radice.
 - * Nel caso in cui il nodo ha entrambi i figli, deleghiamo la distruzione del nodo ad un'altra funzione chiamata **StaccaMin($T \rightarrow \text{dx}, T$)**.

Dunque per il richiamo della funzione Delete ricorsiva abbiamo bisogno di due funzioni ausiliarie. La prima è utilizzata nel caso in cui vogliamo eliminare un nodo che ha entrambi i figli.

StaccaMin($T \rightarrow \text{sx}, T$)

```

1  StaccaMin(T,P)
2      ret = T
3      If T != NIL then
4          ret = StaccaMin(T->sx,T)
5          if ret = NIL then
6              if P != NIL then
7                  if P->sx = T then
8                      P->sx = T->dx
9                  else
10                     P->dx = T->dx
11  return ret

```

La seconda è chiamata per l'eliminazione del nodo andando a operare sull'intero sottoalbero dove quel nodo è radice.

DeleteRoot(T)

```

1  DeleteRoot(T)
2      ret = T

```

```

3  if T!= NIL then
4      Tmp = T
5      if T->sx = Nil then
6          ret = T->dx
7      else if T->dx = NIL then
8          ret = T->sx
9      else
10         Tmp = StaccaMin(T->dx,T)
11         T->key = Tmp->key
12
13     dealloca(tmp)
14     return ret

```

In questo modo la chiamata a questa funziona delega il compito di capire in quale caso dei tre possibili si è dentro e di conseguenza deallocare in modo sicuro. Infine andiamo a scrivere la funzione generale per la distruzione di un nodo nel modo seguente:

```

1  DeleteR(T,k)
2      ret = T
3      if T!= NIL then
4          if T->key < k then
5              T->dx = Delete(T->dx,k)
6          else if T->key > k then
7              T->sx = Delete(T->sx,k)
8          else
9              ret = DeleteRoot(T)
10     return ret

```

9 Lezione 10 - 06/10/2023

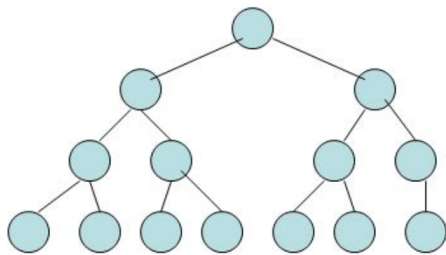
9.1 Alberi Perfettamente Bilanciati

Gli alberi perfettamente bilanciati sono particolare tipi di albero binario in cui vale la seguente condizione:

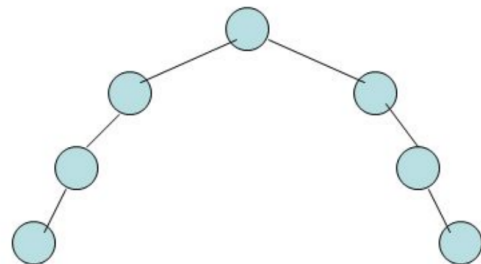
$$||T- > sx| - |T- > dx|| \leq 1$$

La cardinalità (numeri di elementi) del sottoalbero sinistro deve differire di **al più 1** elemento del sottoalbero destro.

Non tutti gli alberi completi sono perfettamente bilanciati ma tutti gli alberi perfettamente bilanciati sono pieni.



(a) È un APB



(b) NON è un APB

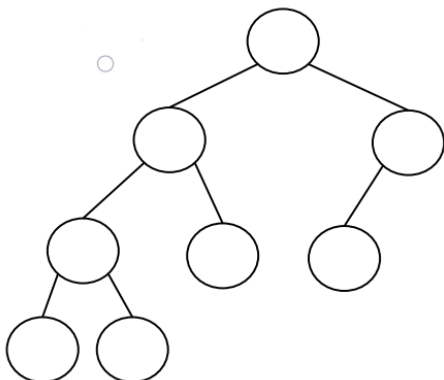
9.2 Alberi AVL

Gli alberi AVL sono particolare tipi di albero binario di ricerca in cui vale la seguente condizione:

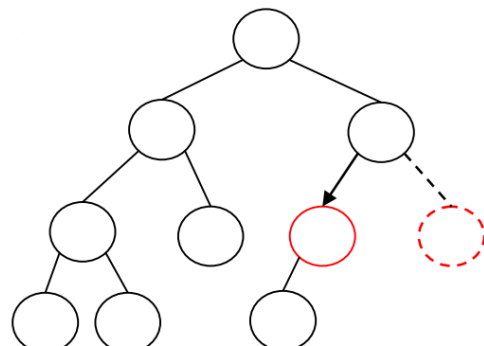
$$|h(T- > sx) - h(|T- > dx)| \leq 1$$

Quindi l'altezza del sottoalbero sinistro di T e quella del sottoalbero destro di T differiscono al più di uno, ovviamente si applica ad ogni sottonodo.

A differenza degli alberi perfettamente bilanciati, non si pone un limite sulla cardinalità dell'insieme ma bensì sull'**altezza** dei sottoalberi.



(a) È un AVL



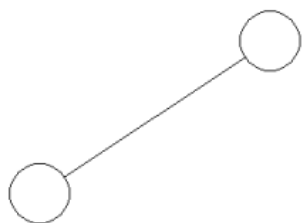
(b) NON è un AVL

Un albero pieno è sia ABL che AVL

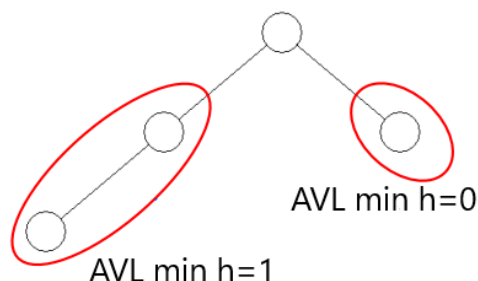
9.3 Alberi AVL Minimi

Fissato h , l'albero AVL minimo di altezza h è l'albero AVL di altezza h col minor numero di nodi possibile.

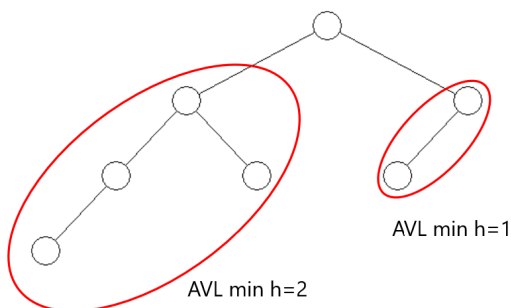
Per ogni altezza andiamo a mostrare un possibile albero:



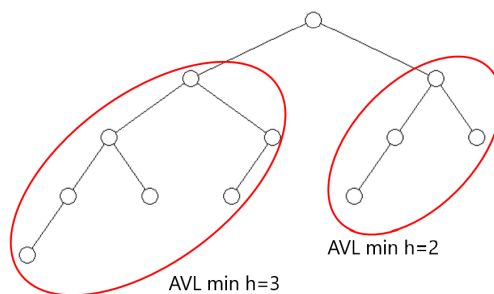
(a) AVL minimo di altezza 1



(b) AVL minimo di altezza 2



(c) AVL minimo di altezza 3



(d) AVL minimo di altezza 4

Possiamo notare un certo pattern che si ripete, nello specifico dato un albero di altezza h il sottoalbero sinistro sarà $h - 1$ e il sottoalbero destro $h - 2$.

Andiamo a generalizzare questa osservazione:

$$N(h) = \begin{cases} h + 1 & \text{se } h = 0, 1 \\ 1 + N(h - 1) + N(h - 2) & \text{e poniamo per assurdo } h \geq 2 \end{cases}$$

DIM $h \geq 2$:

Prediamo un generico AVL T minimo, e poniamo per assurdo che il suo sottoalbero sinistro è un **AVL non minimo**, dunque esisterà un albero T' con sottoalbero sinistro che sarà **AVL minimo**. Quindi è un assurdo il fatto che esisterà un sottoalbero di T' di altezza $h - 1$ con un numero minore di nodi rispetto al sottoalbero di T . In generale dunque se andiamo a dire che T è un Albero AVL minimo, non è possibile che esista un T' con un numero di nodi **minore** di un albero AVL minore.

Data la formula precedente facciamo una considerazione:

Altezza	0	1	2	3	4	5	6	7	8
Numeri Nodi	1	2	4	7	12	20	33	54	88
Fibonacci	0	1	1	2	3	5	8	13	21

Possiamo notare come ci sia un certo collegamento tra i numeri di nodi e la

sequenza di fibonacci, nello specifico notiamo come:

$$N(h) = F(h + 3) - 1$$

Facciamo un ragionamento su come ricaverci l'altezza dato la formula chiusa di Fibonacci:

$$F(x) = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^x - \left(\frac{1 - \sqrt{5}}{2} \right)^x \right]$$

Andiamo a rimuovere la seconda parte poiché tende a zero.

$$F(x) = c \cdot k^x$$

$$N(h) = F(h + 3) - 1$$

$$N(h) = c \cdot k^{h+3} - 1$$

$$\frac{N(h) - 1}{c} = k^{h+3}$$

$$h = \log_k \left(\frac{N(h) - 1}{c} \right) - 3$$

Abbiamo dimostrato che l'altezza è logaritmica sul numero di nodi.

Adesso andiamo a dimostrare che la formula dei nodi vale per ogni h

DIM:

- **Caso base:** $N(0) = F(0 + 3) - 1 = 1$
- **Caso induttivo:**

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

Per ipotesi:

$$N(h - 1) = F(h + 2) - 1$$

$$N(h - 2) = F(h + 1) - 1$$

Quindi:

$$1 + (F(h + 2) - 1) + (F(h + 1) - 1) = F(h + 3) - 1$$

10 Lezione 11 - 10/10/2023

10.1 Altezza

Per aiutarci nel bilanciamento andiamo a scrivere una funzione ausiliaria che ci sarà molto utile:

```
1 Altezza(T)
2   ret=-1
3   if T != NIL then
4     sx=Altezza(T->sx)
5     dx=Altezza(T->dx)
6     ret=1+max(sx,dx)
7 return ret
```

Questa funzione è **lineare** ma nel nostro caso voglio che sia logaritmo, quindi per ovviare a questo problema andiamo a inserire un valore h all'interno di ogni nodo.

```
1 Altezza(T)
2   if T = NIL then
3     ret=-1
4   else
5     ret=T->h
6 return ret
```

10.2 Inserimento AVL

Andiamo a vedere come scrivere un algoritmo per l'inserimento, essendo l'AVL un albero binario di ricerca sfruttiamo un ragionamento simile, ma con l'aggiunta del bilanciamento per rispettare la condizione degli AVL.

```
1 InsertAVL(T, k)
2 if T != NIL then
3   if T-key < k then
4     T->dx=InsertAVL(T->dx, k)
5     T=Bilanciadx(T)
6   else if T->key > k then
7     T->sx=InsertAVL(T->sx, k)
8     T=Bilanciasx(T)
9 else
10  T=newnodo(k)
11  T->h=0
12 return T
```


10.3 Bilanciamento

Dato che l'inserimento/cancellazione può sbilanciare un albero abbiamo bisogno di bilanciare l'albero in modo da far rispettare sempre la condizione, abbiamo diversi casi di bilanciamento andiamo ad esaminarli:

10.3.1 Bilanciamento Sinistro

```
1 Bilanciasx(T)
2 if T != NIL then
3     if Altezza(T->sx) - Altezza(T->dx) = 2 then
4         if Altezza(T->sx->sx) > Altezza(T->dx->dx) then
5             T=Rotazionesx(T)
6         else
7             T->sx=Rotazionedx(T->sx)
8             T=Rotazionesx(T)
9     else
10        T->h=1+max(Altezza(T->sx), Altezza(T->dx))
11 return T
```

11 Lezione 12 12/10/2023

11.1 Cancellazione albero AVL

Per la cancellazione di un elemento nell'albero AVL dobbiamo sempre fare delle considerazioni simili a quelle dell'aggiunta di un elemento nell'AVL.

Nel caso specifico pero c'è bisogno di considerare che la DeleteAVL del nodo. Nella diminuzione di un elemento K del sottoalbero, **il padre** dell'elemento K può essere attaccato facilmente a uno dei due sottoalberi di sinistra o destra. Nel farlo andiamo a diminuire di pesantezza uno dei due sottoalberi rendendo il successivo più pesante. In quel caso la strategia che potremmo attuare è quella del ribilanciamento, ma condizionato dal fatto che ci troviamo in un AVL.

I tre algoritmi sono concettualmente gli stessi che usiamo per il bilanciamento di un BST.

11.1.1 DeleteAVL

Questa funzione è pressoché simile a quella vista negli Alberi binari di ricerca con l'aggiunta del bilanciamento.

```
1 DeleteAVL(T,k)
2   if T != NIL then
3     if T->key > k then
4       T->sx = DeleteAVL(T->sx,k)
5       T = BilanciaDx(T)
6     else if T->key < k then
7       T->dx = DeleteAVL(T->dx,k)
8       T = BilanciaSx(T)
9     else
10      T = DeleteRootAVL(T)
11   return T
```

11.1.2 DeleterootAVL

La delete Root, chiamata quando viene trovato il nodo da eliminare, non solo copia il valore del più piccolo elemento del sottoalbero destro di T, ma andrà a bilanciare nuovamente a sinistra (perché più pesante) il sottoalbero di T.

```
1 DeleteRootAVL(T)
2   if T != NIL then
3     tmp = T
4     if T->sx = NIL then
5       T = T->dx
6     else if T->dx = NIL then
7       T = T->sx
8     else
9       tmp = stacca\_minAVL(T->dx,T)
```

```

10     T->key = tmp->key
11     T = BilanciaSx(T)
12     dealloca(tmp)
13     return T

```

11.1.3 Stacca_min

Per lo staccamin abbiamo bisogno di una variabile in piu in questo caso perche non ci e possibile andare a salvare l'elemento da cancellare dopo il bilanciamento, che si perderebbe nell'albero. In tal caso l'uso di una variabile e il controllo successivo se si trova nel nodo figlio di destra o sinistra ci verra in aiuto.

```

1  Stacca\_minAVL(T,P)
2  if T != NIL then
3      If T->sx != NIL then
4          ret = Stacca\_minAVL(T->sx, T)
5          newt = BilanciaDx(T)
6      else
7          ret = T
8          newt = T->dx
9      if T = P->sx then
10         P->sx = newt
11     else
12         P->dx = newt
13     return ret

```

Le operazioni di cancellazione andranno a effettuare una operazione di diminuzione dell'altezza di un albero e conseguentemente un'altra a cascata per il bilanciamento, che solitamente cambia le dimensioni dell'albero finale. Questo solitamente non ci dara problemi, ma solo in caso di alberi AVL ci dara problemi, poiche ci fara perdere le proprieta di quest'ultimo. In un esempio di AVL minimo la cancellazione di un nodo nel sottoalbero meno pesante andrebbe a comportare un ribilanciamento che va a peggiorare la differenza delle altezze tra alberi, portandola a 2, e quindi perdendo la proprieta di albero.

12 Lezione 13 - 13/10/2023

12.1 Alberi Red - Black

Gli alberi Red - Black, sono alberi binari di ricerca che associano dei colori ai loro nodi. La colorazione ovviamente andrà a braccetto con alcune proprietà (vincoli) che di seguito andremo a definire.

- 1) Ogni nodo deve essere rosso o nero.
- 2) **I nodi foglie possono essere solo neri (NIL)**, quindi i nodi rossi potranno essere soltanto all'interno.
- 3) Ogni nodo rosso ha **solo** figli neri.
- 4) Per ogni nodo X preso all'interno dell'albero, ogni percorso da X al nodo foglia contiene **lo stesso numero** di nodi neri.

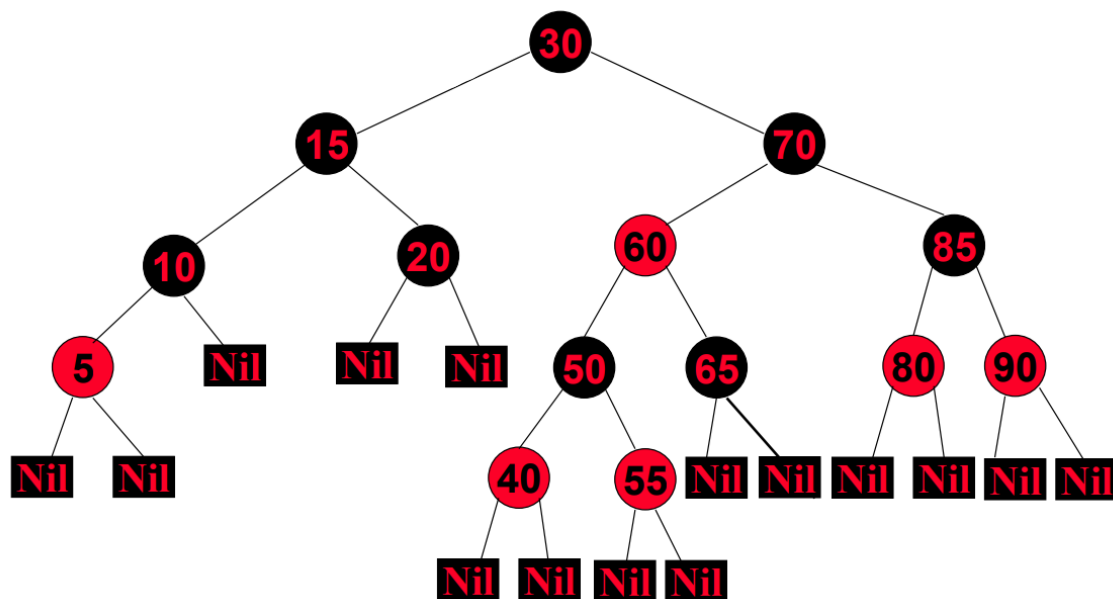


Figure 4: Questo è un albero RB perché soddisfa tutti e 4 i vincoli

Non tutti gli alberi possono essere colorabili

Per vedere se un albero è colorabile ci sono delle considerazioni da fare:

- Colorare subito le foglie e la radice di nero.
- Osservare se esiste un sottoalbero è visibilmente più pesante di un altro. In tal caso l'albero è quasi sicuramente non colorabile. Teoricamente se la differenza di altezza di alberi è maggiore di due allora probabilmente non è colorabile.

12.1.1 Altezza Nera di un albero R-B

L'altezza nera, di un albero R-B, è il numero di nodi neri che, preso un nodo X, si contano da X fino alle foglie escludendo X.

L'altezza nera è sicuramente minore dell'altezza dell'albero e al massimo uguale.

Dimostriamo dunque che l'altezza è sicuramente:

$$h \leq 2^{bh(x)} - 1$$

Preso un nodo all'interno di un albero il numero di nodi interni non può essere più piccolo di un albero completamente nero.

$$ni(x) \geq 2^{bh(x)} - 1$$

Dimostriamo per induzione

- Base Induttiva: Albero di altezza zero, quindi il numero di nodi interni di un albero di $h = 0$ è **zero**, andiamo a svolgere l'equazione con la base induttiva:

$$0 \geq 2^{bh(x)} - 1 \Rightarrow 2^0 - 1 = 0 \Rightarrow \text{VERO}$$

- Caso Induttivo $h > 0$: L'albero contiene almeno un nodo interno. Andiamo a scomporre il nostro albero come sottoalbero sx del figlio sinistro e sottoalbero dx del figlio destro.

$$ni(y) \geq 2^{bh(y)} - 1$$

$$ni(z) \geq 2^{bh(z)} - 1$$

Noi sappiamo che

$$ni(x) = 1 + ni(y) + ni(z)$$

In questo caso, ragionando analiticamente possiamo dire che l'altezza nera di X, il nostro nodo padre del sottoalbero dipende dal fatto che y, il suo sottoalbero sinistro, sia nero o rosso.

- Nel caso in cui il nodo sia rosso, allora l'altezza nera di x e y sono uguali.
- Nel caso in cui il nodo sia nero, allora l'altezza nera di x è uguale a quella di y + 1.

Esplicitiamo $bh(y)$ dalle due equazioni perché ci interessa esplicitare tutto per $bh(x)$

In questo caso vedremo che $bh(y) \geq bh(x) - 1$ poiché o è uguale, o è sicuramente maggiore di $bh(x) - 1$.

Questo vale anche per z, dunque avremo:

$$bh(y) \geq bh(x) - 1$$

$$bh(z) \geq bh(x) - 1$$

Questo vuol dire che scendendo di altezza, andrò a diminuire al massimo di uno l'altezza del sottoalbero. Grazie a queste equazioni possiamo ritornare a ritroso alla tesi.

Usiamo questo ragionamento matematico: Se io so che $n \geq m$, allora avrò anche che $2^n \geq 2^m$ poiché l'esponenziale è crescente e non andiamo a modificare il risultato comunque finale.

Applichiamo dunque la stessa proprietà alle stesse equazioni scritte sopra. In tal caso avremo

$$bh(y) \geq bh(x) - 1 \rightarrow 2^{bh(y)} \geq 2^{bh(x)-1}$$

sottraiamo una stessa quantità a entrambi i membri

$$2^{bh(y)} - 1 \geq 2^{bh(x)-1}$$

Dunque entrambi vedremo che la somma tra $2^{bh(y)} + 2^{bh(z)}$ sono maggiori o uguali di $2^{bh(x)-1}$.

Il numero di nodi interni di X è dato da $1 + ni(y) + ni(z)$. Sostituendo abbiamo che $1 + 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 \rightarrow 2 * 2^{bh(x)-1}$. Il "-1" può essere semplificato portando dentro il 2 moltiplicato avanti all'espressione. In tal modo avremmo che indipendentemente dall'altezza che io ho in entrata, il numero di nodi interni di quel sottoalbero è almeno uguale a $2^{bh(x)-1}$, quindi la tesi iniziale è dimostrata.

12.1.2 Considerazioni sull'altezza nera di un albero

Se sappiamo che il numero di nodi n è maggiore o uguale a $2^bh - 1$, possiamo intuitivamente e approssimativamente andare a trovare l'altezza nera dell'albero.

Nel caso in cui avesse tutti i nodi neri allora l'altezza nera è $\leq h$, mentre se ha alternati rossi e neri, abbiamo il limite minimo dell'altezza nera, cioè $\frac{h}{2}$.

Dunque l'altezza nera è compresa tra : $\frac{h}{2} \leq bh \leq h$.

Usando la matematica e le nozioni della dimostrazione precedente...

$$bh \geq \frac{h}{2} \rightarrow 2^{bh} - 1 \geq 2^{\frac{h}{2}-1}$$

Se sappiamo che $n \geq 2^{bh} - 1$ allora,

$$n \geq 2^{\frac{h}{2}-1}$$

$$n + 1 \geq 2^{\frac{h}{2}}$$

$$\log_2 h + 1 \geq \frac{h}{2}$$

$$h \geq 2 \log_2 n + 1$$

12.1.3 Inserimento Albero R-B

Gli algoritmi di inserimento e bilanciamento usati fino ad ora non andranno più bene per questo tipo di struttura. Nonostante abbiamo più libertà da un certo punto di vista, dobbiamo considerare che la proprietà 4 degli alberi Red-Black ci impedisce di fare degli inserimenti nella struttura dati in modo efficiente.

In particolare nell'inserimento di un valore in un nodo NIL, l'algoritmo deve occuparsi di creare il nodo, colorarlo e di creare e colorare a sua volta i figli NIL (di nero ovviamente).

Successivamente la colorazione del nodo k non è immediata e semplice, va considerato il colore del padre e non va rotto il vincolo dello stesso numero di nodi neri per ogni percorso dei sottoalberi.

Abbiamo due possibilità di colore all'inserimento. Solitamente per non creare problemi con il padre del sottoalbero a cui dobbiamo inserire, si inserisce nero. Anche in quel caso non è detto che l'inserimento del nero non abbia creato problemi per la proprietà 4 dei R-B.

Nei Red - Black la violazione di una di questi due algoritmi può essere scoperta solo ricorsivamente.

```

1  InsertRB(T,k)
2      if T != NIL then
3          if T->key < k then
4              T->dx = InsertRB(T->dx, k)
5              T = BilanciaRBdx(T)
6          else if T->key > k then
7              T->sx = InsertRB(T->sx, k)
8              T = BilanciaRBsx(T)
9          else
10             T = new_nodeRB(k,r) //Creazione nodo rosso (r) e figli a NIL
11     return T

```

Questa funzione verrà supportata dalle funzioni di bilanciamento che sono specificatamente scritte apposta per la R-B. In tal caso abbiamo 3 casistiche generali di problemi. Nel caso in cui il nodo che andiamo a inserire sia rosso...

- Caso 1) Il padre rosso e il fratello rosso. Sia a destra che a sinistra del padre rosso.
- Caso 2) Inserimento a destra del sottoalbero il cui padre è rosso e il fratello nero.
- Caso 3) Inserimento a sinistra del sottoalbero il cui padre è rosso e il fratello è nero.

La risoluzione del Caso 1, si scambia il nodo padre rosso con il nonno nero, in tal caso abbiamo che i figli diventeranno per forza di cose neri. In questo modo non andiamo a violare la proprietà 4, poiché i percorsi sia a destra che a sinistra avranno lo stesso numero di nodi neri, ma andiamo soltanto ad aumentare l'altezza nera.

Il Caso 2 si risolve ruotando in modo tale da arrivare al caso 3.

Per il caso 3 ci conviene ruotare l'albero a destra e sostituire il nodo radice (precedentemente nero) con un nodo rosso (visto che era figlio di nero). In tal modo abbiamo la radice nera, i figli rossi e i sottoalberi non cambiano.

Aggiustare assolutamente

13 Lezione 14 17/10/2023

13.1 Bilanciamento All'inserimento in Albero R-B

```
1  BilanciaInsSx(T)
2    if !NIL(T->sx) and (!NIL(T->sx->sx) or !NIL(T->sx->dx)) then
3      v = ViolazioneSxInsRB(T->sx,T->dx)
4      Case V of :
5        1: T = Caso1(T)
6        2: T = Caso2(T)
7        3: T= Caso3(T)
8    return T
```

La violazione verifica se esiste una violazione di regola numero 4 (degli R-B), mandando il sottoalbero sinistro e destro rispettivamente all'interno di una funzione controllo che restituirà il tipo di caso di violazione che stiamo riscontrando in quel sottoalbero.

13.2 Controllo Violazione a Sinistra Inserimento R-B

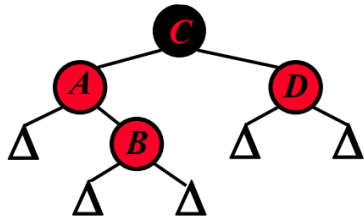
```
1  ViolazioneSxInsRB(s,d)
2    v=0
3    if s->col = R then
4      if d->col = R then
5        if s->sx->col = R or s->dx->col = R then
6          v=1
7        else if s->dx->col = R then
8          v=2
9        else if s->sx->col = R then
10         v=3
11    return v
```

13.3 Risoluzione Violazione Inserimento a sinistra

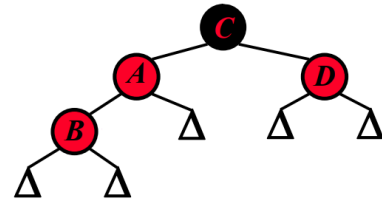
Quando andiamo ad inserire un nodo in un albero RB è facile violare le proprietà, quindi andiamo a indentificare i 3 casi in cui avvengono e come andare a risolverli. Generalmente il problema principale è non rompere la proprietà 4. Per far sì che non si rompa la proprietà diamo al nodo appena creato il colore rosso, che potrebbe compromettere i sottoalberi destro e sinistro.

13.4 Caso 1

Possiamo indentificare il **Caso 1**, osservando solamente se il figlio destro è un nodo rosso. La violazione può avvenire sia sul sottoalbero sinistro del nodo sinistro della radice, sia nel destro.



(a) Albero RB Caso 1 (n1)



(b) Albero RB Caso 1 (n2)

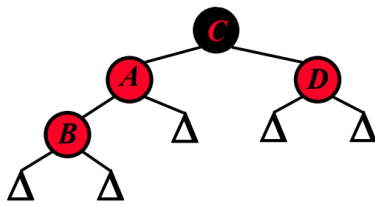
Per corregere questa violazione andiamo a colorare la "radice" di rossa (anche se per convenzione dovrebbe essere nera), e poi di conseguenza andando a colorare i figli sinistri e destri di nero.

```

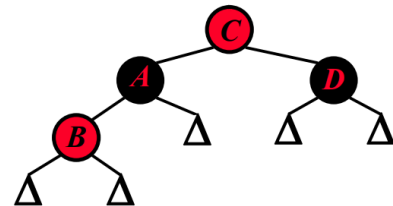
1 Caso1(T)
2   T->dx->col=black;
3   T->sx->col=black;
4   T->col=red;
5   return T;

```

In questo tipo di soluzione eliminiamo la violazione sui sottofigli, facciamo in modo che l'altezza nera sia rispettata in tutti i nodi, ma andiamo a "spostare" il problema verso l'alto, che poi verrà corretto a cascata, fino eventualmente al nodo radice.



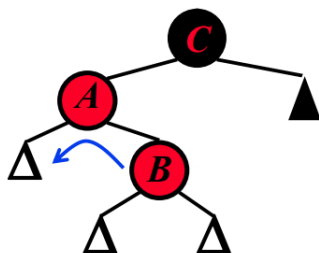
(a) Violazione Caso 1



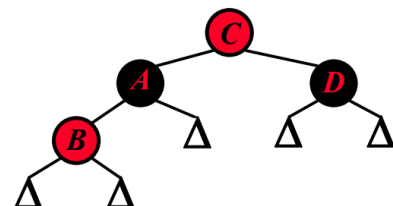
(b) Correzione Violazione

13.5 Caso 2

Il caso 2 (vedi foto) comporta il fatto che il **figlio destro** del sottoalbero su cui stiamo operando è **nero**. La strategia di risoluzione per il caso 2 prevede di far "ruotare" il problema a sinistra (quindi far salire di altezza la violazione) e far sì che diventi caso 3, in questo modo andiamo semplicemente a richiamare la strategia di risoluzione del caso 3.



(a) Violazione Caso 1



(b) Correzione Violazione

```

1 Caso2(T)
2   T->sx = rotate(T->sx)
3   t = Caso3(T)
4   return T

```

13.6 Caso 3

Il caso 3, come nel caso 2, si verifica nel momento in cui, il figlio destro del sottoalbero su cui stiamo operando è nero. La differenza col caso 2 è che il figli del sottoalbero sinistro non sono entrambi rossi, ma il rosso si trova soltanto nel sottoalbero destro (come si nota in foto). La strategia di risoluzione è dunque andare a ruotare sulla sinistra l'albero e scambiare i colori della radice e del figlio destro.

```

1 Caso3(T)
2   T = rotazioneSx(T)
3   T->col = N
4   T->dx->col = R
5   return T

```

Ovviamente queste stesse violazioni sono possibili sul sottoalbero destro. Analogamente le soluzioni sono le stesse con opportune inversioni di nodi su cui devono essere fatte le operazioni.

13.7 Delete Albero R-B

Per la distruzione del nodo di un albero R-B ci vogliono molte più accortezze della creazione poiché:

- La distruzione di un nodo nero comporterebbe una failure nella proprietà 4 dell'altezza nera.
- Un nodo da distruggere potrebbe avere figli, quindi bisogna ragionare su come disporli e colorarli.

Gli unici nodi da distruggere sono i nodi interni.

L'elemento da sostituire cambierà colore in base a quale colore sia stato cancellato.

- La distruzione di un nodo **rosso** comporterà che il nodo acquisterà il colore di nero.
- La distruzione di un nodo **nero** comporterà che il nodo acquisterà il colore di **doppio nero**, per non perdere la proprietà 4.

```

1 DeleteRB(T,k)
2 if !NIL(T) then
3   if T->key > k then
4     T->sx = DeleteRB(T->sx,k)
5     T = BilanciaDelsxRB(T)
6   else if T->key < k then
7     T->dx = DeleteRB(T->dx,k)
8     T = BilanciaDeldxRB(T)
9   else

```

```
10     T = DeleteRootRB(T)
11     return T
```

Parallelamente alla delete degli AVL la funzione di DeleteRB hanno la stessa struttura di funzioni.

```
1 DeleteRootRB(T)
2     if !NIL(T) then
3         TMP = T
4         if NIL(T->sx) then
5             T = T->dx
6             if TMP->col = N then
7                 PropagateBlack(T)
8         else if NIL(T->dx) then
9             T = T->sx
10            if TMP->col = N then
11                PropagateBlack(T)
12        else
13            TMP = StaccaMinRB(T->dx, T)
14            T = BilanciaDeldxRB(T)
15        dealloca(TMP)
16    return T
```

14 Lezione 17 24/10/2023

14.1 Grafi

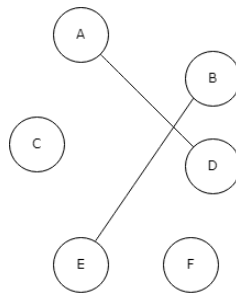
I grafi sono una coppia di insiemi definita come: (V, E) (chiamato oggetto), dove V è un insieme finito di elementi che compongono il grafo chiamati **vertici**. L'insieme E rappresenta l'insieme degli **archi**.

L'insieme E è costituito da una coppia composta così definita: $E \subseteq V \times V$, quindi una coppia formata da due vertici dell'insieme V .

L'insieme E è in relazione simmetrica.

I grafi non posseggono foglie, ma esistono grafi con archi uscenti soltanto, verso nodi che non hanno altri collegamenti.

Un grafo si dice **completo** quando il suo numero di archi è uguale a n .



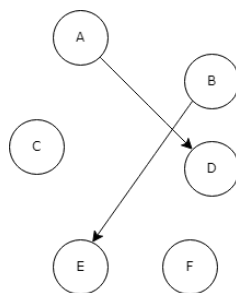
$$V = \{A, B, C, D, E, F\}$$

$$E = \{(A, D), (B, E)\}$$

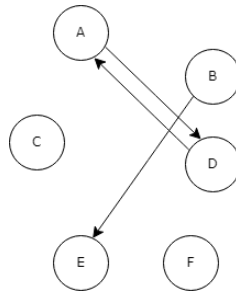
14.2 Grafo Orientato

Un grafo orientato è un grafo che ha gli archi direzionati, cioè che vanno da una direzione all'altra, solitamente nella rappresentazione grafica a coppie, la direzione va dal primo elemento verso il secondo.

(I secondi elementi nella coppia sono i cosiddetti "Carrier della relazione").



Per "tornare" al caso in cui il grafo non è orientato, bisogna avere aggiungere degli archi orientati nel verso opposto a quelli che già esistono:



14.3 Alberi e Grafi

Gli alberi sono particolari tipi di grafi dove la relazione che vige tra i nodi è quella di parentela (padre figli). In esso i tragitti non sono infiniti come possono essere in un grafo. Un grafo non può essere ricostruito come un albero nel caso in cui un nodo si perde, poiché non sappiamo quanti e quali archi abbia perso a differenza di un nodo di un albero che può avere un solo padre e uno o 2 figli.

14.4 Grafi Pesati

I grafi pesati sono un'applicazione dei grafi ordinati e non, dove ad ogni arco viene associato un numero (peso).

14.5 Grado di un Vertice

Il grado di un vertice può dividersi in

- Grado entrante, cioè il numero di archi che entrano nell'arco
- Grado uscente, cioè il numero di archi che escono dall'arco

P.S. un cappio è un arco che esce e entra nello stesso nodo.

Ogni vertice può avere come massimo numero di archi (entranti e uscenti), il numero di vertici del grafo.

$$n = |V|$$

14.6 Percorsi

Sia $v, w \in V$ e definito π = il percorso da v in w nel grafo G . Per percorso si intende la sequenza di vertici che si impegnano per arrivare al vertice stabilito.

$$\pi = v_0, v_1, v_2, \dots, v_k$$

tale che

- $v_0 = u$
- $v_k = v$
- $\forall 0 \leq i \leq k - 1 = (v_i, v_{i+1})$

La coppia di valori tali che il secondo è il successivo del primo.

14.7 Raggiungibilità di un nodo

Un nodo v è detto raggiungibile da u in un grafico $G \iff \exists \pi | \pi$ è un percorso da u in v .

Quindi solo se esiste un percorso (insieme di nodi raggiungibili attraverso degli archi) da u in v .

La raggiungibilit  puo essere scritta come funzione $Reach \in V \times V$. La funzione $Reach$ possiede delle proprieta:

- Riflessiva v e raggiungibile da v poiche $(v, v) \in E$, ed e chiamato percorso senza archi.
- Transitiva Se $(u, z) \in Reach$ e $(z, v) \in Reach$ allora avremo che sicuramente $(u, v) \in Reach$. Se esiste un percorso che da u va in z e da z in v , allora esistera un percorso che raggiunge (u, z) , concatenando i percorsi dei due.

14.8 Percorsi Ciclici

Un grafo orientato in cui non esistono cicli semplici e detto **Grafo Ciclico**, dove per ciclo semplice si intende una "ripetizione di un vertice in una sequenza".

14.9 Sottografo

Dato $G = (V, E)$, allora $G' = (V', E')$ e detto sottografo di G . La condizione principale per far si che G' sia sottografo e che $V' \subseteq V$. In questo caso il numero degli archi e : $E' \subseteq E \cap (V' \times V')$. Nel caso in cui $E' = E$, allora ci troviamo in un **sottografo indotto**.

15 Lezione 18 26/10/2023

15.1 Complessità sui grafi

I grafi hanno una ciclicità variabile, ovviamente dal numero di archi che ognuno di essi avrà. In un grafo completo ad esempio c'è una probabilità certa di trovare delle ripetizioni (ciclicità) in un percorso.

Data una famiglia di grafi $\{G_i\}_{i \leq N}$ (grafo finito) e dove i è il numero di vertici del grafo. Ciclicamente avremo che i grafi completi con " i " numeri di vertici avranno: Il grafo con 2 vertici:

Il grafo con 3 vertici:

Il grafo con 4 vertici:

Generalmente il numero dei percorsi che si hanno per un grafo è uguale alla somma del numero di percorsi che si hanno nei sottografi più piccoli insieme al suo, che avrà i percorsi.

$$P(i) = \sum_{k=1}^{i-1} P(k)$$

ovviamente questo vale se $i > 2$.

Se andiamo a esprimere questa somma avremo che:

$$P(i) = \begin{cases} 1 & \text{se } i \leq 2 \\ 2P(i-1) & \text{se } i > 2 \end{cases}$$

Questa equazione è chiamata **Equazione di ricorrenza**, equazione che si risolve con algoritmo ricorsivo per verificare il numero di percorsi in un grafo completo.

Questa stessa equazione è utilizzabile per un albero binario di ricerca completo poiché, ogni nodo ha due figli e quindi due sottoalberi dove al massimo possono essere di 2^{i-1} nodi. Anche in questo caso dunque avremo che:

$$P(i) = \begin{cases} 1 & \text{se } i \leq 2 \\ 2P(i-1) & \text{se } i > 2 \end{cases}$$

Quindi il numero di percorsi in un grafo è uguale a 2^{i-1} .

Questo risultato ci rende impossibile l'esplorazione totale del grafo (come anche in un caso analogo di un albero completo), poiché è esponenziale sul numero di nodi.

15.2 Come Rappresentiamo un Grafo?

Come possiamo andare a "disegnare un grafo" e quali informazioni minime abbiamo bisogno per farlo? Un modo abbastanza semplice e intuitivo è quello di andarlo a descrivere come lo abbiamo definito in precedenza quindi come $E \subseteq V \times V$. In questo modo abbiamo coppie di nodi a cui abbiamo associato i propri archi.

Un altro modo di rappresentare un grafo è attraverso l'uso della **funzione caratteristica**, cioè quella funzione "booleana" che dato in input un dato ci restituirà 0 o 1 a seconda della presenza o meno di quel dato nella struttura dati su cui vogliamo cercare.

15.2.1 Matrice di bit

Possiamo dunque andare a creare una specie di scacchiera (matrice) dove dal risultato della casella potremmo capire se quel nodo è collegato o meno con un altro.

La matrice è chiamata **Matrice di Bit**, che ci permette di avere un **vantaggio** nella ricerca di un eventuale arco (a tempo costante). Lo **svantaggio** invece è quello che dovremmo creare una matrice con $n \times n$ con l'eventuale utilizzo di memoria indesiderata in più. Inoltre non sappiamo dapprima quanti archi potremmo trovare nella matrice, non abbiamo un'informazione a tempo costante ma a tempo **lineare** in questo caso. E nella costruzione non c'è da dimenticarsi che è quadratica sul numero intero di bit. Alternativamente un altro modo per visualizzare il nostro grafo è quello di sfruttare la proprietà di adiacenza dei grafi, e quindi creare una struttura dati che ci tiene traccia di tutte le adiacenze che ogni vertice ha.

In questo caso ci viene in aiuto un array unidimensionale dove ogni cella punta a ogni vertice del grafo e da lì poi tutte le adiacenze saranno messe all'interno di ogni struttura.

15.2.2 Liste di adiacenza

Questa struttura dati è chiamata Lista di adiacenza. Lo **svantaggio** rispetto alla matrice è che per trovare un arco bisogna scorrere linearmente sul numero dei vertici della lista. Il **vantaggio** è che lo spazio impiegato per la lista intera è logaritmico sul numero di vertici $\log_2 n$.

Solitamente la soluzione scelta è la lista di adiacenza per convenzione e comodità. Le liste hanno una migliore esplorazione dei grafi, se non sono densi. In generale però è una ottima scelta di compattezza.

16 Lezione 19 - 27/10/2023

16.1 Visite Grafi

Come per gli alberi anche sui grafi esistono vari tipi di visite, avendo maggiore libertà di "movimento" dato che ogni vertice può essere raggiunto da più percorsi dobbiamo a definire alcuni concetti importati.

Vertice Sorgente Dato che in un grafo non esiste un punto di partenza come la radice negli alberi assegniamo questo ruolo ad un cosiddetto **vertice sorgente** s che avrà il ruolo di farci esplorare tutti gli altri nodi del grafo (frontiera).

Colorazione Come abbiamo detto ogni nodo può essere raggiunto da diversi percorsi per evitare di visitare più volte lo stesso vertice assoceremo ad ognuno di essi un colore che ci permetterà di capire lo "stato" di esplorazione:

- Bianco: Vertice non ancora scoperto (caso base)
- Grigio: Vertice scoperto ma non visitato
- Nero: Vertice scoperto e visitato (caso finale)

16.1.1 BFS

Rispetto agli alberi la visita in ampiezza non può sfruttare il concetto di "livello" poiché in un grafo non abbiamo questo tipo di concetto ma sfrutteremo la **distanza**, quindi verranno esplorati prima i vertici più vicini al **vertice sorgente** e poi quelli più distanti.

```
1 Init(G) //G:grafo
2   for each v in V do //per ogni vertice del grafo
3       color[v]=b //coloriamo di bianco il vertice
```

La funziona **init** ha lo scopo di colorare tutti i nodi di bianco, il costo di questa funzione è $|V|$ cioè il numeri dei vertici del grafo.

```
1 BFS(G, s) //G:grafo, s: vertice sorgente
2   Init(G)
3   Q={s} //creiamo una coda con dentro s
4   color[s]=g //coloriamo di grigio
5   while Q != NIL do
6       x=Testa(Q) //prendiamo l'elemento in testa
7       for each v in ADJ(x) do //scorriamo gli adiacenti
8           if color[v] = b then //se e' bianco
9               Q=Accoda(Q,v) //mettiamo il nodo bianco nella frontiera
10              color[v]=g //mettiamo a grigio
11          //operazione su x
12          Q=Decoda(Q) //x ha finito gli adiacenti e quindi lo togliamo
13          color[x]=n //lo mettiamo a nero (stato finale)
```

La funzione **BFS** ha un costo $|V| + (|E| + |V|) = |V| + |E| = |G|$

16.1.2 Variante BFS Tempo

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

17 Lezione 21 - 02/11/2023

17.1 Visita in Profondità - DFS

La Depth-First Search (DFS) è un algoritmo di esplorazione dei grafi il cui funzionamento consiste nel visitare prima i nodo più lontani dal nodo sorgente e poi tornando progressivamente verso il punto di partenza.

Rispetto alla BFS vista in precedenza, la DFS viene impiegata per scopi diversi. Mentre la BFS è utilizzata per scoprire percorsi minimi, la DFS può essere usato per verifica la ciclicità di un Grafo. Come per la BFS useremo di "strutture ausiliarie":

- $Col[]: V \rightarrow \{b, g, n\}$ (array dei colori)
- $Pred[]: V \rightarrow V$ (array dei predec., cioè quale vertice è stato scoperto da chi)
- $d[]: V \Rightarrow \mathbb{N}[\{1, 2, \dots, 2n\}]$ (il tempo in cui il vertice è stato scoperto)
- $f[]: V \Rightarrow \mathbb{N}[\{1, 2, \dots, 2n\}]$ (il tempo in cui il vertice è stato visitato)

Dato che per la DFS vogliamo visitare tutti i vertici di un grafo, dato che non sempre un vertice sorgente ci permette di farlo, andiamo a definire due funzioni:

```
1 Init(G) //G:grafo
2   for each v in V do //ogni vertice del Grafo
3       col[v]=b
4       prev[v]=NIL
5   tempo=1 //variabile globale
```

La funzione DFS ci permette di far ripartire la visita da un nuovo vertice sorgente (bianco) in modo da poter esplorare ogni parte del grafo.

```
1 DFS(G) //G:grafo
2   Init(G)
3   for each v in V do //ogni vertice del Grafo
4       if col[v]=b then
5           DFS_Visit(G,v)
```

Ogni volta che troviamo un nuovo nodo (bianco) andiamo a richiamare ricorsivamente la visita in modo da scendere in Profondità nel grafo e poi successivamente "risalire".

```
1 DFS_Visit(G, s) //G:grafo, s: vertice sorgente
2   col[s]=g //coloriamo di grigio
3   d[s]=tempo++ //scoperto a tempo corrente (+1)
4   for each v in ADJ[s] do
5       if col[v]=b then
6           pred[v]=s
7           DFS_Visit(G,v) //visita ricorsiva
8   f[s]=tempo++ //visitato a tempo corrente (+1)
9   col[s]=n
```

Costo Come visto per la BFS possiamo anche qua ogni nodo viene visitato una sola volta quindi il costo sarà sempre $|V|+|E|=|G|$

18 Lezione 22 - 03/11/2023

18.1 Dimostrazione Teorema delle parentesi

La dimostrazione della condizione di sufficienza (\rightarrow) del teorema delle parentesi. Per induzione sulla lunghezza di π (che rappresenta la lunghezza del percorso tra u e v):

Base d'induzione: $|\pi| = 1$. In questo caso esiste un arco che va direttamente da v a u . Essendo che u è figlio di v e $v = \text{prev}[u]$, allora sicuramente avremo che $d[v] < d[u]$. Ora ci rimane da capire quando vengono chiuse entrambe. Per coerenza con il nostro algoritmo di DFSvisit, sappiamo che sicuramente verranno chiusi prima i discendenti e poi gli antenati a ritroso, in questo caso dunque verrà chiuso prima u che v . In questo modo abbiamo dimostrato la nostra tesi, cioè che $d[v] < d[u] < f[u] < f[v]$.

Caso $|\pi| > 1$: In questo caso abbiamo che $\exists z \in V : v \rightarrow z \rightarrow u$. In questo caso allora abbiamo un sotto-percorso π' costituito da un solo elemento, come nella base induttiva. In tal senso abbiamo dunque che $d[v] < d[z] < d[z] < d[v]$. Possiamo estendere questo ragionamento agli n nodi che dividono v da u e creare quindi una catena di *dedif*, ma quello che vedremo semplificando tutti i nodi che non interessano alla dimostrazione che $d[v] < d[u] < f[u] < f[v]$.

18.2 Teorema del percorso bianco

Dato un grafo $G = (V, E)$ ed eseguita una DFS su G per ogni coppia di vertici $v, u \in V$ e con $v \div u$: u è detto discendente di v nella FDF *seesolose* al tempo $d[v]$ esiste un percorso (da u a v) fatto da solo vertici bianchi.

18.2.1 Dimostrazione Teorema - Condizione Necessaria \rightarrow

La foresta ha già almeno un percorso con vertici bianchi, quindi quel percorso può stare nella foresta. Se prendiamo un arbitrario vertice $z \in V$ del percorso, vedremo che z è discendente di v solo se $d[v] < d[z] < f[z] < f[v]$. E questo è verificato dal teorema precedente. Se la condizione è rispettata, tutti i vertici fino a z erano bianchi (a tempo v), e quindi rispetta una parte della tesi. Lo stesso ragionamento si può applicare a tempo z per tutti i vertici fino a u .

18.2.2 Dimostrazione Teorema - Condizione Sufficiente \leftarrow

Abbiamo un percorso fatto da tutti i vertici bianchi, dimostreremo per assurdo che non esista discendenza di u su v .

Ipotizziamo dunque che esiste un vertice che non è discendente da v , nel percorso di tutti bianchi fino a u e lo chiameremo t .

L'ultimo vertice discendente da v lo chiameremo z e si troverà prima di t , quindi $d[v] < d[z] < f[z] < f[v]$. Ci chiediamo ora quando verrà scoperto t , non essendo discendente di v o di z . Non potrebbe essere scoperto prima di z poiché a tempo di $d[v]$, sono tutti bianchi per la tesi del teorema. Non potrebbe essere scoperto al ritorno (durante la chiusura di v), perché dovrebbe essere già stato scoperto in chiusura, senno andrebbe in contrasto con il teorema delle parentesi. Il punto più probabile di scoperta è tra $d[z]$ e $f[z]$, e ciò andrebbe in tesi con ciò che abbiamo detto

poiche diventerebbe $d[v] < d[z] < d[t] < d[u] < f[u] < f[t] < f[z] < f[v]$. Andando a semplificare i nodi che non ci interessano, abbiamo che il nodo t rispetta il teorema delle parentesi e quindi il concetto di discendenza.

18.3 I problemi dell'algoritmo DFS

L'algoritmo DFS non prende in considerazione un ordine di scoperta dei nodi, ma fa come piu gli piace, o "scopre il primo che trova". In questo modo possiamo incorrere in problemi sugli archi. In tal senso andiamo a distinguere vari tipi di archi che possiamo trovare all'interno dell'albero. Tra i vari tipi abbiamo:

- Archi dell'albero, cioe archi che scoprono un nodo **bianco** a partire da un nodo **grigio**. E il normale metodo di scoperta dei nodi durante una DFS.
- Archi all'indietro, cioe archi che connettono un discendente appena scoperto (quindi **grigio**) a un vertice gia scoperto **grigio**.
- Archi in avanti, sono archi che connettono un antenato in fase di scoperta Deep (**grigio**), e un discendente gia scoperto e usato nero.
- Archi di attraversamento, sono archi che connettono 2 nodi di FDF differenti, quindi non si trovano mai nello stesso albero, ma hanno gli stessi colori degli archi in avanti : da **grigio** a **nero**.

Gli archi all'indietro sono archi che rispettano questa formula: $d[v] < d[u] < f[u] < f[v]$. E la scoperta dell'arco all'indietro avviene al centro della disequazione.

I tipi di archi 3 e 4 hanno lo stesso vertice su cui puntano di colore nero, per distinguerli abbiamo bisogno di discriminare i loro tempi: >Per un arco in avanti (da v a u), la scoperta dello stesso avviene dopo che u e terminato (quindi dopo $f[u]$) e sempre prima della terminazione di v (prima di $f[v]$). >Per un arco di attraversamento i tempi di discovery possono essere disgiunti : $d[v] < f[v] < d[u] < f[u]$.

18.4 Come aggiungere queste informazini all'algoritmo DFS

Per far si che questi nostri ragionamenti sui tipi di archi dobbiamo aggiungere delle righe di codice alla nostra DFS visit.

```

1 DFSvisit(G,s)
2   Col[s] = g
3   d[s] = tempo++ // il tempo viene prima incrementato e poi
                   aggiunto a d[s]
4   for each v in Adj[s] do
5     if col[v] = b then
6       pred[v] = s
7       DFS_visit(G,v)
8   else
9     if Col[v] = g then
10      print '(s,v) e un arco all'indietro'
11    else if d[s] < d[v] then
12      print '(s,v) e arco in avanti'
13    else}
14      print '(s,v) e un arco di attraversamento'
15  f[s] = tempo++

```

18.5 Proprieta archi di ritorno

(Questa sezione ha fonte solo da valentina che era presente a lezione li)

- Se all'interno di una DFS incontriamo un arco di ritorno, allora c'è necessariamente un ciclo. Dimostriamola prendendo π che rappresenta il percorso tra (v, u) . Se in questo percorso esiste un arco (u, v) , allora c'è un ciclo.
- La non esistenza di un arco all'indietro implica che non esiste ciclo all'interno del percorso. Per dimostrare questa proprietà consideriamo $G = (V, E)$, cioè un grafo che contiene un ciclo. Supponiamo che v_1 sia il primo vertice del percorso e che per il teorema dei percorsi bianchi allora tutti i suoi vertici discendenti ancora dovranno essere scoperti. A seconda di quale dei vertici discendenti abbia un arco di ritorno allora avremo che esisterà ciclo. Quindi è dimostrato che se esiste almeno 1 arco all'indietro esiste un ciclo e viceversa se non esiste.

18.6 Algoritmo del grafico Aciclico

```

1  Acicliclo(G)
2    For Each v ∈ V do
3      if Col[v]=b then
4        ret = Acicliclo_Visit(G,v)
5        if ret = false then
6          return false
7    return true

```

```

1  Aciclico_Visit(G,s)
2    Col[s] = g
3    For Each v in Adj[s] do
4      if Col[v] = b then
5        ret = Aciclico_Visit(G,v)
6        if ret = false then
7          return False
8      else if Col[v] = g then
9        return false
10   return true

```