

Algoritmi e Strutture Dati 2023-24

(M. Benerecetti)

Contents

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

6	Lezione 07 - 29/09/2023	17
6.1	Albero Binario	17
6.2	Altezza di un Albero	17
6.3	Complessità Computazione	17
6.4	Visite	18
6.4.1	PreOrder	18
6.4.2	InOrder	18
6.4.3	PostOrder	19
6.5	Ricerca Albero Binario	19
7	Lezione 08 - 03/10/2023	20
7.1	Nomi Utilizzati delle visite	20
7.2	Visita In Ampiezza Albero	20
7.3	Albero Binario di Ricerca	21
7.3.1	Search Ricorsiva - BST	21
7.3.2	Min - BST	21
7.3.3	Algoritmo del Successore Ricorsivo - BST	22
7.3.4	Algoritmo del Successore Iterativo - BST	22
7.4	Lezione 9 - 05/10/2023	23
7.4.1	Successore Iterativo - BST	23
7.4.2	Predecessore Ricorsivo - BST	23
7.4.3	Insert Ricorsiva - BST	23
7.4.4	New Node - Generico	24
7.4.5	Insert Iterativa - BST	24
7.4.6	DeleteR - BST	25
8	Lezione 10 - 06/10/2023	26
8.1	Alberi Perfettamente Bilanciati	26
8.2	Alberi AVL	26
8.3	Alberi AVL Minimi	27

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. elementari)  
4       for j=1 to N do //idem for di sopra  
5           if i<=j then //2 letture + confronto (3 op. elementari)  
6               ris = ris+1 //lettura+scrittura+assegnamento (3 op. elementari)  
7   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:

$$N - i + 1 \text{ volte che l'operazione viene eseguita.}$$

- 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 è 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'avere 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 fintantochè 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 .

$$|S| = |A|$$

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 "destra"
6       else if A[q] < e then
7           i=BinSearch(A, e, q+1, j) //ricerca a "sinistra"
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
7     if A[q] < k then
8       i=q-1
9     else if A[q] > k then
10      j=q+1
11   else
12     ret = q
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:

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

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,1,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=k
7       else
8           temp = temp->next
9   return ris
```

La variabile "temp" ci serve per evitare di andare a perdere "l'inizio" della lista in ingresso dato che la lista in questo caso può solo andare avanti, quindi se non ci salviamo il punto di inizio non possiamo più tornare all'inizio

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           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:

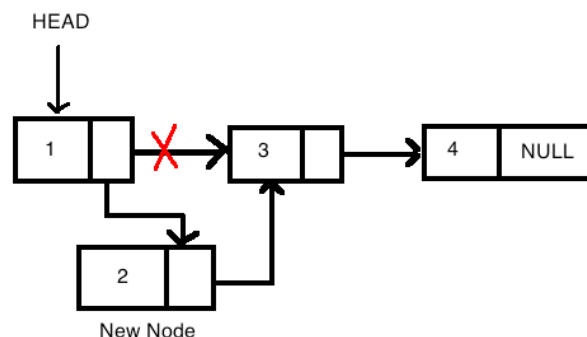


Fig. Insert after a given Node

Andiamo a definire l'algoritmo in maniera iterativa:

```
1 InsertO(L,K)
2   temp=L //salviamo il puntatore al primo elemento
3   P=NIL //creiamo una variabile di appoggio
4   while [temp != NIL 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 = NIL or temp->key > k] then
9       new=newNode(k,temp) //creiamo il nodo attaccandolo al "successivo"
10      if P != NIL 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=NIL 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 computazionale 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.

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

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

Questo piccolo inserto legenda servira per identificare i 2 particolari tipi di visita che effettueremo.

- **DFS** o visite in profondita sono i tipi di visite che prevedono prima la visita dei figli e poi l'operazione che si vuole effettuare.
- **BFS** o visite di **ampiezza** sono tipi di visite che prevedono l'utilizzo di una coda per l'esplorazione dei figli. Infatti questa visita prevedera prima l'esplorazione del nodo padre e poi la messa in coda dei figli.

7.2 Visita In Ampiezza Albero

La visita in ampiezza dell'albero e un tipo di visita che 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 e possibile effettuarla con un algoritmo ricorsivo/iterativo e basta. Infatti per questa visita ci sara utile l'utilizzo di una struttura dati astratta gia studiata in precedenza che ci consentira l'esplorazione. **La coda** ci sara utile per effettuare questi tipi di visite. La coda ci serve per inserire i figli del nodo selezionato all'interno di essa e poi prelevarli per andare a visitarli e a sua volta inserire i suoi figli.

```
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 puo essere complesso fare una cosa del genere? Le operazioni all'interno e prima del while sono a tempo costante perche sono operazioni di assegnazione. Il while pero dipende dalla quantita di nodi all'interno dell'albero. In particolare a seconda se l'albero e bilanciato o meno.

- Nel caso peggiore l'albero e un albero degenerare, in questo caso la coda avra tutti gli elementi della lista all'interno di essa. Il costo computazionale dunque, a seconda dell'altezza, sara il **tetto frazionario** di $n/2$. Dunque la memoria necessaria in questo caso potrebbe richiedere una quantita lineare (per la coda).
- Nel caso migliore l'albero e bilanciato, cioe ogni nodo ha 2 figli. A livello di memoria invece sara costante.

7.3 Albero Binario di Ricerca

E' un tipo di albero ordinato, con una relazione che lega tutti i nodi figli. Il vincolo sta a indicare che preso un qualsiasi nodo, esso sara sempre piu piccolo del suo sottoalbero destro, e piu grande del suo sottoalbero sinistro. L'acronimo per questo albero sara **BST**. Il motivo per cui e stato introdotto questo albero e per facilitare la ricerca di dati, infatti il suo algoritmo di ricerca e uno dei piu 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 permettera di cercare in modo piu efficiente il dato poiche, un po come la ricerca binaria (ma non propriamente), ci permette di andare a "dimezzare" l'area di ricerca ogni volta che si fa un 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 temporale per lo scorrimento dell'albero.

7.3.2 Min - BST

La ricerca del minimo in un albero binario di ricerca (dunque ordinato) e molto semplice. Se semplicemente seguendo la regola secondo cui "l'elemento piu piccolo si trova a sinistra". Allora andremo a scorrere continuamente a sinistra finche non troveremo l'elemento piu 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. In questo caso l'algoritmo diventa un po più complesso nelle casistiche:

- Caso in cui l'elemento di cui vogliamo il successore è uguale alla chiave dell'elemento in cui siamo. Caso $T \rightarrow \text{key} = k$. In questo caso l'elemento che sicuramente sarà successore si troverà nel sottoalbero destro. E il più piccolo elemento del sottoalbero destro sarà il nostro risultato che cerchiamo quindi $\rightarrow \text{Min}(T \rightarrow \text{dx})$
- Caso $T \rightarrow \text{key} < k$. Se il numero di cui cerchiamo il successore si trova ancora in un nodo più piccolo di esso allora dovremmo spostarci a destra con la stessa funzione. $\text{Succ}(T \rightarrow \text{dx})$.
- Caso $T \rightarrow \text{key} > k$. Se il numero sarà più grande di quello che cerchiamo allora dovremmo spostarci sulla sinistra. $(T \rightarrow \text{sx}, k)$

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

7.3.4 Algoritmo del 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
```

7.4 Lezione 9 - 05/10/2023

7.4.1 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
```

7.4.2 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
```

7.4.3 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

```

7.4.4 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 nodo
   allocato in memoria a ret
3  ret->key = k
4  ret->sx = NIL
5  ret->dx = NIL
6  return ret

```

7.4.5 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->key < k then
15          P->dx = x
16      else
17          P->sx = x
18  else
19      P-sx = x

```


7.4.6 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
 - * Il caso in cui il nodo o ha figlio destro o figlio sinistro e basta.
 - * Il caso in cui il nodo ha entrambi i figli collegati. In questo caso specifico deleghiamo la distruzione del nodo a un'altra funzione chiamata **Stacca($T \rightarrow \text{dx}, k$)**.

In particolare questa funzione sarà formata in questo modo:

```
1 StaccaMin(T,P)
2   ret = T
3   If T != NIL then
4     ret = StaccaMin(T->sx,P)
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
```

8 Lezione 10 - 06/10/2023

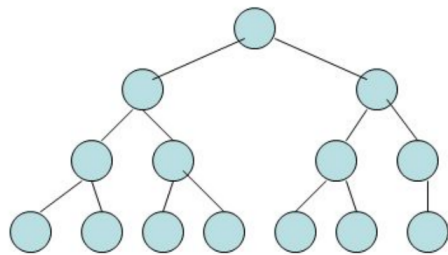
8.1 Alberi Perfettamente Bilanciati

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

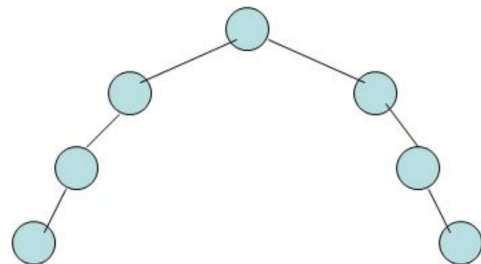
$$||T- > sx| - |T- > dx|| \leq 1$$

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

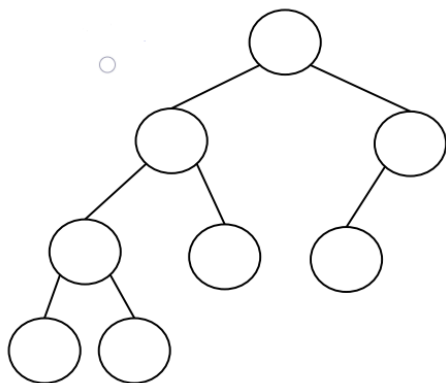
8.2 Alberi AVL

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

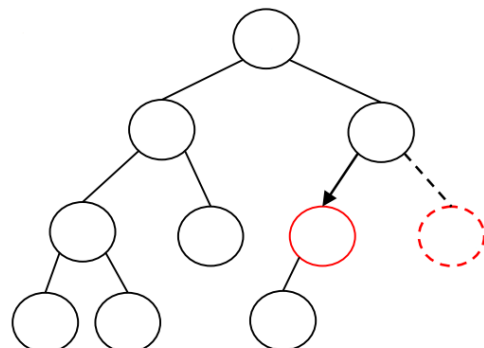
$$|h(T- > sx) - h(T- > dx)| \leq 1$$

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



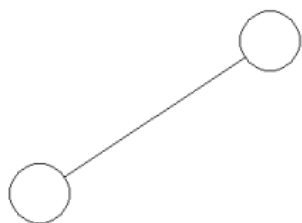
(b) NON è un AVL

Un albero pieno è sia ABL che AVL

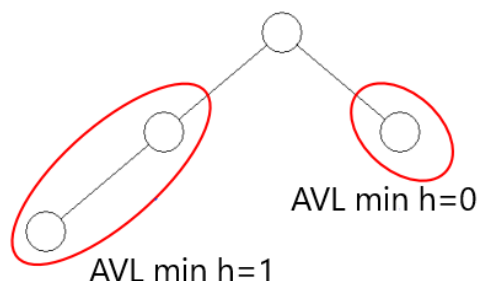
8.3 Alberi AVL Minimi

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

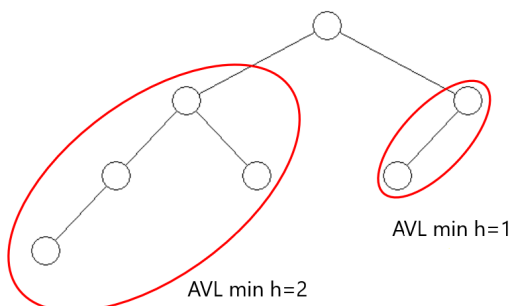
Per ogni altezza andiamo a mostrare un possibile albero:



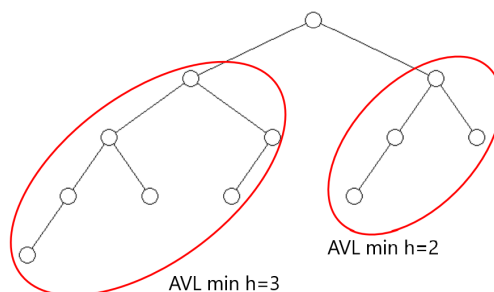
(a) AVL minimo di altezza 1



(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 ciasottoalbero 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 esiste un T' con un numero di nodi **minore** di un albero AVL minore.

Data la formula precendete 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 siano un certo collegamento tra i numeri di nodi e la sequenza di fibonacci, nello specifico notiamo come:

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

DIM: