DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

Progettazione algoritmo per creazione Tableau di Young. Utilizzato **IDE CodeBlock**.

Per maggiori dettagli vedere codice con relativi commenti inseriti.

Federico Maglione N86001405
Daniele Punziano N86001504

Anno accademico 2016-2017

Docenti: Murano Aniello
Di Stasio Antonio

INDICE

1 CAPITOLO : DESCRIZIONE TABLEAU DI YOUNG

Un Tableau di Young è una matrice m x n nella quale gli elementi di ogni riga sono ordinati da sinistra a destra e gli elementi di ogni colonna sono ordinati dall'alto verso il basso.

Alcuni elementi di una tableau di Young possono essere infiniti, che noi consideriamo elementi inesistenti.

Quindi un tableau di Young può essere utilizzato per contenere r <= m*n numeri finiti.

2	4	9	∞
3	8	16	∞
5	14	∞	∞
12	∞	∞	∞

CAPITOLO 2 : STRATEGIE PER RISOLUZIONE PROBLEMI E PRINCIPALI STRUTTURE DATI UTILIZZATE

Strategia per creazione Tableau di Young:

La strategia applicata si basa sulla creazione di una matrice di dimensioni n x m la quale permette di inserire un minimo di 1 ad un massimo di n*m elementi.

In maniera più specifica:

1. Si richiede la dimensione della matrice che si vuol creare,
   dimensione orizzontale e poi verticale.
2. Si richiede posizione per posizione il valore che si vuol inserire nella matrice , con la possibilità di inserirlo o meno (con opportuni controlli che permettano di rispettare le proprietà del tableau di Young)
3. Ad ogni valore inserito si esegue un ordinamento (basato sul heapsort) , così da rispettare le proprietà del tableau di Young

Strategia per estrazione minimo:

La strategia applicata si basa sull'estrazione della elemento di posizione [0] [0] , il quale (dato l'ordinamento eseguito precedentemente) corrisponderà al minimo valore possibile presente nella matrice

In maniera più specifica:

1. Si controlla il primo valore della matrice [0][0]
2. Se tale valore è infinito (elemento nullo) si ritorna un messaggio di errore
3. Altrimenti si assegna tale valore ad una variabile che verrà stampata a schermo.

Strategia per l'inserimento di un nuovo valore nel Tableau:

La strategia applicata si basa sull'acquisizione di un valore numerico da standard input che a sua volta verrà confrontato con gli elementi già presenti nella tabella, e nel caso in cui non sia presente verrà inserito nell'opportuna prima posizione libera.

In maniera più specifica:

1. Si richiedere un valore numerico da standard input.
2. Si esegue un controllo per determinare che il valore appena inserito non sia presente nella matrice.
3. In caso affermativo si posiziona tale valore nella prima posizione disponibile che risulterà essere quella contenente il valore infinito (nullo).
4. Una volta inserito il valore si esegue un ordinamento per ripristinare il Tableau di Young.
5. In caso negativo si riporterà un messaggio di "errore tabella piena" su standard output.
6. Si eseguono opportuni controlli per valori non accetti.

Strategia per la creazione di un tableau N x N da un Tableau N x M

La strategia applicata si basa sull'estrapolazione dei valori presenti nella table n x m poi analizzati, per determinare la minima table quadrata che li può contenere, così da crearla e copiarne i valori all'interno.

In maniera più specifica:

1. Si contano gli elementi della matrice n x m .
2. Si crea una matrice nxn che può contenere gli elementi contati.
3. Si estrae l'i-esimo elemento della matrice n x m (con i che parte dalla prima posizione della matrice
4. Tale elemento lo si copia nella matrice nxn e si esegue un incremento della posizione
5. Quando sono stati inseriti tutti gli elementi della matrice n x m si termina la computazione.

Strategia per la creazione di un Tableau N x N di N^2 elementi

La strategia applicata si basa sulla richiesta su std input di un numero quadratico di elementi , i quali andranno inseriti in una determinata matrice quadratica (calcolata in base al numeri di elementi inseriti).

In maniera più specifica:

1. Si richiedono m = n^2 numeri da stdInput, inserendoli in un vettore
2. Si realizza una matrice nxn in base agli m elementi inseriti.
3. Si copiano gli elementi dal vettore ad una nuova matrice di dimensione nxn.

Strategia per la creazione di una tabella N x M efficiente in base al numero di elementi inseriti.

La strategia applicata si basa sull'acquisizione di un numero di elementi da stdInput , usati per calcolare la dimensione orizzontale e verticale migliore associata (controllo se il numero inserito è un quadrato oppure eseguo la radice quadrata del numero ed in base al valore dato determino le dimensioni.

In maniera più specifica:

1. Richiedo da stdInput il numero di elementi da inserire e li assegno ad un vettore
2. Eseguo un controllo sul numero di elementi inseriti determinando se sia un quadrato o meno
3. Se un quadrato utilizzo la radice di quest'ultimo come dimensione sia orizzontale che verticale
4. Se non è un quadrato eseguo la radice quadrata di quest'ultimo e mi determino il numero di righe = parte intera del quadrato, numero di colonne che dipenderà dalla parte decimale della radice
5. Se la parte decimale della radice è >= 0.5 eseguo un incremento di 2 valori della parte intera.
6. Se la parte decimale della radice è < 0.5 eseguo un incremento di 1 valore della parte intera.
7. Eseguo una copia degli elementi dal vettore alla matrice appena creata.

Strutture dati utilizzate:
Le principali strutture dati utilizzate sono **matrici** e **vettori**.

3 CAPITOLO: DETTAGLI IMPLEMENTATIVI (I/O & FUNCTION)

Ogni funzione descritta e stata dotata di un link ipertestuale che riporta al codice C della funzione che si sta esaminando.

Variabili Globali:

- Int min: variabile che contiene il valore minimo della tabella
- Int tableNew[MAX][MAX] : nuova tabella usata per eseguire varie operazioni
- Int pot: variabile usata per determinare una matrice quadrata
- Int hzDim, vtDim: variabili usate per ritornare le dimensioni delle matrici modificate al main.
- #define MAX 10
- #define MAXVET 100

Librerie usate:

- <stdlib.h>
- <stdio.h>
- <limits.h>
- <math.h>

Ordinamento Tableau di Young(function):

void buildTableau(int verticalDim, int horizontalDim, int table[MAX][MAX]

I/O:

- Int table[MAX][MAX] : tabella di Young su cui lavorare.
- Int verticalDim: dimensione verticale della tabella di Young
- Int horizontalDim: dimensione orizzontale della tabella di Young:
- Int hzIndex, vtIndex: indici usati per la table.

Descrizione funzione:

Si eseguono due cicli for con vtIndex = verticalDim-1 finche vtIndex >=0 si incrementa vtIndex e hzIndex = horizontalDim-1 finche hzIndex >=0 si incrementa hzIndex
In questo costrutto si richiama la funzione heapTableau(table, hzIndex, vtIndex, horizontalDim, verticalDim) che permette di eseguire l'ordinamento dei valori passati alla table.

void heapTableau(int table[MAX][MAX], int hzIndex, int vtIndex, int horizontalDim, int verticalDim)

I/O:

- Int table[MAX][MAX] : tabella di Young su cui lavorare.
- Int verticalDim: dimensione verticale della tabella di Young
- Int horizontalDim: dimensione orizzontale della tabella di Young:
- Int leftVt, leftHz : indici per il figlio sinistro verticale e posizione orizzontale
- Int rightVt, rightHz: indici per il figlio destro orizzontale e posizione verticale

Descrizione funzione:

Si esegue un controllo con il costrutto if dove

a) Se leftVt < verticalDim && leftHz < horizontalDim) && table[leftVt][leftHz] < table[vtIndex][hzIndex]
ovvero che la posizione del figlio sinistro verticale sia minore della dimensione verticale e che la posizione del leftHz sia minore della dimensione orizzontale e che il figlio sinistro sia minore del valore dato dalla posizione di table[vtIndex][hzIndex], se tale condizione viene rispettata (equivale a dire che il figlio sinistro è più grande del valore dato da vtIndex e hzIndex) si assegna a largestVt = leftVt; largestHz = leftHz;

a1) altrimenti si assegna a largestVt = vtIndex; largestHz = hzIndex (come valore più grande)

b) Se rightVt < verticalDim && rightHz < horizontalDim) && table[rightVt][rightHz] < table[largestVt][largestHz] ovvero che la posizione del figlio destro orizzontale sia minore della dimensione orizzontale e che la posizione del rightVt sia minore della dimensione verticale e che il figlio destro sia minore del valore fino adesso più grande ovvero quello dato da table[largestVt][largestHz], si assegna a largestVt = rightVt e largestHz = rightHz.
c) A questo punto se largestVt != vtIndex || largesthz != hzIndex (ovvero controlla che largest non sia la posizione stessa di hzIndex e vtIndex) eseguo un swap tra la posizione contenente il valore più grande e vtIndex , hzIndex. Eseguo l'heapTableau (questa volta passandoci come valori (table,largestHz,largestVt,horizontalDim,verticalDim, quindi largestHz e largestVt) e rieseguo l'heap partendo da questi ultimi valori passati.

Inizializzazione Tableau di Young (function) :

void PopulateFristTimeTable(int table[MAX][MAX], int verticalDim, int horizontalDim)**:**

I/O **:**

- Int Vertical : Indice Verticale
- Int Horizontal : Indice orizzontale
- Int table[MAX][MAX] : tabella di Young su cui lavorare.
- Int verticalDim: dimensione verticale della tabella di Young
- Int horizontalDim: dimensione orizzontale della tabella di Young:

Descrizione Funzione: Due cicli for innestati che permettono l'inizializzazione di tutta la matrice N x M a valori nulli (infinito).

Inserimento valori nella Tabella(function):

void InsertValueIntoTable(int table[MAX][MAX], int verticalDim, int horizontalDim, int choice):

I/O :

- Int Check : variabile di controllo
- Int TableZero : variabile di controllo per controllo elementi nulli
- Int SaveIndex : variabile per salvare un indice quando inserito come valore -1.
- Int Insert : elemento da inserire.
- Int table[MAX][MAX] : tabella di Young su cui lavorare.
- Int verticalDim: dimensione verticale della tabella di Young
- Int horizontalDim: dimensione orizzontale della tabella di Young:
- Int Choise: variabile usata per determinare che operazione eseguire.

Descrizione Funzione:

Si cicla la tabella prima in senso verticale e poi orizzontale e ad ogni iterazione eseguiamo opportuni controlli:

a) Se check è uguale a 0 (controlla che non si stato inserito un -1 nella riga su cui lavoriamo) e se tableZero è uguale a 0 (controlla che non sia stato inserito un -1 come prima valore della riga su cui lavoriamo) e se saveIndex è uguale a 0 (controlla che non è mai stata salvata la posizione orizzontale della table dato che non è stato inserito mai il valore -1) oppure che l'indice orizzontale deve essere minore di saveIndex.
    1. Si eseguo un ciclo dove l'utente inserisce il valore da assegnare alla tabella.
        a. Se l'utente inserisce un valore = -1 e choice è diverso da 3 (quindi non stiamo eseguendo la terza operazione del menu del programma) si assegna a table[vertical] [horizontal] il valore nullo (infinito)
        Si pone check = 1. E si salva la posizione orizzontale in saveIndex
        Si esegue un controllo su horizontal (nel caso fosse = 0) si pone tableZero = 1.
        b. Se l'utente inserisce un valore >=0 lo si assegna a table[vertical][horizontal].
        c. Se l'utente inserisce un valore < -1 viene stampato su stdOutput un messaggio di errore "Il valore inserito deve essere un intero o -1 per elemento nullo"
        d. Se l'utente inserisce -1 e si sta eseguendo la 3 operazione del menu viene stampato su stdOutput un messaggio di errore "Il valore inserito deve essere un intero positivo.
        E si stampa una tabella per tener traccia degli elementi inseriti.

b) Se una di queste condizioni non viene rispettata viene inserito nella table in quella posizione il valore nullo(infinito).
c) Si esegue un ordinamento per ripristinare le proprietà del tableau di Young.
d) Si stampa la tabella appena creata
e) Si azzera check = 0.

void callInsertValeu(int table[MAX][MAX], int horizontalDim, int verticalDim)

I/O:

- Int newValue: Variabile usata per l'assegnazione del nuovo valore inserito.
- Int table[MAX][MAX]: tabella di Young su cui lavorare
- Int horizontalDim: dimensione orizzontale della tabella
- Int verticalDim: dimensione verticale della tabella

Descrizione funzione:

Si esegue una stampa della tabella su cui lavorare per avere una traccia degli elementi presenti.

Si esegue un ciclo iterativo do { }while() dove si richiede l'inserimento di un nuovo valore attraverso una funzione chiamata checkAndGetInteger(table,verticalDim,horizontalDim) che andrà ad acquisire da stdInput un valore, ed eseguirà un controllo su di esso per verificare che non sia già presente nella matrice. Una volta acquisito lo si assegna alla variabile newValue.

Il ciclo do{ } while () termina quando il valore inserito (newValue) non sia un numero negativo.

Si richiama la funzione newValueTableau() che permetterà di copiare il valore nella tabella.

Void newValueTableau(int table[MAX][MAX], int horizontalDim, int verticalDim, int newValue)

I/O:

- Int hzIndex: indice orizzontale
- Int vtIndex: indice verticale
- Int check : variabile di controllo.
- Int table[MAX][MAX]: tabella di Young su cui lavorare
- Int horizontalDim: dimensione orizzontale della tabella
- Int verticalDim: dimensione verticale della tabella

Descrizione funzione:

Si eseguono 2 cicli for innestati (con indici hzIndex e vtIndex) .
Si controlla con un if che il valore dato dalla posizione degli indici della matrice(table[vtIndex][hzIndex]) sia uguale ad un valore nullo.

a) Se si trova un valore nullo si pone check = 1 (così da tener traccia se troviamo un elemento nullo o meno)

I cicli for si fermano in 2 condizioni:

1. Il primo è quando abbiamo analizzato l'intera matrice ma non è stato trovato nessun elemento nullo. (Conseguentemente non è possibile inserire un nuovo valore)
2. Il secondo è quando è stato trovato un elemento nullo e quindi essendo check settato ad 1 si fuoriesce dai for poiché non si soddisfa più una delle due condizioni in and.

A questo punto si controlla con un if se ci troviamo o meno alla fine della matrice e se l'ultimo elemento non è nullo

a) In caso che si rispetti la condizione del if verrà riportato un messaggio di errore tabella piena.
b) In caso contrario alla condizione del if si assegna newValue alla posizione dove il ciclo for si è fermato -1. Successivamente si esegue il buildHeap per eseguire l'ordinamento e ripristinare le proprietà del Tableau di Young.

Estrazione minimo(function):

callMinTableau(int table[MAX][MAX], int verticalDim, int horizontalDim):

I/O:

- Int Min: variabile globale usata per l'assegnazione del minimo elemento della matrice
- Int table[MAX][MAX]: tabella di Young su cui lavorare
- Int horizontalDim: dimensione orizzontale della tabella
- Int verticalDim: dimensione verticale della tabella

Descrizione funzione:

Si richiama la funzione minTableau che calcola il minimo e ritorna un valore pari o al minimo o nel caso di matrice vuota , o pari a -2 e lo assegna a min.
Viene eseguito un controllo sul min che se uguale a -2 viene stampato un messaggio di errore per avvertire che la tabella e' vuota.

MinTableau(int table[MAX][MAX], int verticalDim, int horizontalDim):

I/O:

- Int MinValueTable: valore usato per assegnare il minimo elemento della matrice

Descrizione funzione:

Si esegue un controllo su il primo elemento della matrice (table[0][0] che corrisponde al valore più piccolo) :

a) se ques'ultimo è diverso dal valore nullo (infinito) lo si assegna alla variabile minValueTable. Successivamente viene posto il valore nullo in posizione table[0][0] e si riesegue il buildTableau per ripristinare l'ordinamento dato dalle proprietà del Tableau di Young. Si ritorna minValueTable
b) Se quest'ultimo è uguale al valore nullo (infinito) si ritorna -2.

Creazione di un tableau N x N da un Tableau N x M && creazione di un tableau N x N partendo da N^2 elementi(function):

void callTableauNxN(int table[MAX][MAX], int horizontalDim, int verticalDim, int controll);

I/O:

- Int table[MAX][MAX]: tabella di Young su cui lavorare
- Int horizontalDim: dimensione orizzontale della tabella
- Int verticalDim: dimensione verticale della tabella
- Int controll: variabile usata per determinare che operazione eseguire sulla matrice (se creazione di un tableau di N x N da un N x M(controll = 0) oppure se creazione di un tableau N x N da N^2 elementi (controll = 1))

Descrizione funzione:

Si richiama la funzione tableauNxN che permetterà di eseguire una delle due operazioni differenziate da controll(creazione di un tableau di N x N da un N x M oppure creazione di un tableau N x N da N^2 elementi)

Se controll == 0 si stampa un messaggio inerente al primo tipo di operazione.

Se controll == 1 si stampa un messaggio inerente al secondo tipo di operazione.

Si stampa la tabella ricavata.

void tableauNxN(int table[MAX][MAX], int horizontalDim, int verticalDim, int controll)

I/O

- Int table[MAX][MAX]: tabella di Young su cui lavorare
- Int horizontalDim: dimensione orizzontale della tabella
- Int verticalDim: dimensione verticale della tabella
- Int controll : variabile usata per determinare che operazione eseguire sulla matrice (se creazione di un tableau di N x N da un N x M oppure se creazione di un tableau N x N da N^2 elementi)
- Int VtIndex, hzIndex : indici usati per scorrere la table
- Int indexHzNew, indexVtNew : indici usati per scorrere la newTable
- Int array[MAXVET]: array di appoggio.
- Int arrayIndex: indice dell'array
- Int arrayIndexCheck : variabile usata per il controllo dell indice dell'array
- Int value: variabile contenete un valore preso da StdInput
- Int count, countNew: variabile usata per il conteggio degli elementi.
- Int check, checkCount, checkDouble: variabili di controllo

Descrizione Funzione:

Si esegue un controllo sulla variabile controll,

a) Se uguale a 0 si eseguono 2 cicli for(con un controllo su ogni valore della table[vtIndex][hzIndex] per determinare se ci sono elementi nulli) contando gli elementi non nulli della matrice.
b) Se uguale a 1 si richiede quanti elementi si vuole inserire .
   Successivamente si esegue un ciclo do{ }while( ) nel quale si determina se il numero inserito è un quadrato (in caso contrario si stampa su stdOutput un messaggio di errore e si richiede l'inserimento)
   Se un quadrato si pone check = 1 che permette di uscire dal ciclo do{ } while (check ==0)

   Si eseguo un ciclo for che permette di inserire nel vettore array il numero di elementi appena definiti andando a controllare che i valori inseriti non siano già presenti o non siano negativi. (questo controllo lo si esegue ponendo check = 1 nel caso si trovi un valore uguale, conseguentemente tramite un costrutto if con condizione checkDouble == 1 stampa su StdOutput un messaggio di errore "Valore già esistente" e permette di reinserire il valore. Si azzera check.

   A questo punto si esegue un cilco do{ } while (check == 0)
   Si esegue un controllo if (pot*pot >= count) dove count è il numero di elementi da inserire nella nuova matrice e pot è un numero che viene incrementato ciclo dopo ciclo fino a trovare il valore corretto tale da contenere count.

   Si inizializza la nuova tableNew con tutti valori nulli (due ciclo for innestati) e si azzera indexHzNew e IndexVtNew

   A questo punto si esegue un controllo su controll:
   a. Se uguale a 0 ciclo la table e ad ogni iterazione delle colonne controllo che il valore su cui lavoriamo sia diverso dal nullo (infinito). Se diverso si esegue un ulteriore controllo if (indexHzNew < pot && countNew < count) dove la prima condizione controlla se l'indice orizzontale si trova alla fine della dimensione (pot) della nuova matrice (newTable) mentre la seconda condizione controlla se non abbiamo ancora copiato tutti gli elementi dalla matrice table alla matrice tableNew (difatti ad ogni copia, countNew viene incrementato)
      Si assegna alla posizione [indexVtNew][indexHzNew] della tableNew il valore della table[vtIndex][hzIndex] si esegue un buildTableau per ripristinare le proprietà del tableau di Young.
      Si incrementa il numero di elementi copiati.
      Si incrementa la posizione orizzontale dell'indice della tableNew.

      Quando si arriva alla fine della dimensione orizzontale della tableNew(pot) si esegue lo stesso costrutto appena descritto però sul indice verticale incrementandolo, azzerando l'indice orizzontale, copiando di nuovo il nuovo

elemento nella posizione corretta, incrementando il numero di elementi inseriti e incrementando la posizione dell'indice orizzontale
Se countNew == count (equivale a dire che ho copiato tutti i valori) pongo checkCount = 1 ed esco dal for (che ha come altra condizione checkCount !=1)

b. Se uguale a 1 ciclo l'array fino a count, e ad ogni iterazione controllo se l'indice orizzontale della tabella nuova (indexHzNew) < pot (dimensione nuova tabella) se soddisfa tale condizione vado a copiare l'elemento dell'array nella tableNew
tableNew[indexVtNew][indexHzNew] = array[arrayIndex]
eseguo un buildTableau per ripristinare le proprietà del tableau di Young ed incremento l'indice orizzontale della nuova tabella indexHzNew.

Creazione di una tabella N x M efficiente in base al numero di elementi inseriti(function):

void newTableauOp(int table[MAX][MAX])

I/O:

- Int table[MAX][MAX]: tabella su cui lavorare
- Int hzIndex, vtIndex: indice usati per la table
- Int vector[MAXVET] : vettore di appoggio
- Int indexVector : indice del vettore
- Int arrayIndexcheck, checkDouble: variabili usate per controllo elementi sul vettore
- Int numberElement: numero di elementi inseriti
- Int value: variabile a cui si assegna un valore
- Double nValeuFract: variabile che contiene la parte decimale della radice del numero di elementi
- Double nValueInt: variabile che contiene la parte intera della radice del numero di elementi
- Double nValue: variabile che contiene la radice del numeri di elementi

Descrizione Funzione:

Si richiede di quanti elementi si vuol creare la tabella (numberElement)
Si esegue un controllo con un do{ }while( ) per inserire elementi compresi tra 1 e MAXVET

Si cicla con un for il vettore fino a numberElement inserendo il valore nella posizione data dal for, controllando che il valore appena inserito non sia già presente nel vettore (stesso funzionamento del controllo nella function tableauNxN)

Si calcola la radice quadrata di numberElement e la si assegna a nValue
Si calcola tramite la funzione modf(contenuta nella lib math.h) la parte decimale di Nvalue assegnandola a nValueFract e assegnando la parte intera nValueInt
Si esegue un cast da double a int di NvalueInt e lo si assegna a value.
Si eseguono una serie di controlli per determinare la più efficiente (in termini di dimensione) matrice tale da contenere gli elementi prima inseriti

a) Se il numero di elementi (numberElement) e uguale a value*value (ovvero numberElement è un quadrato) si assegna a vtDim e hzDim = value come dimensione.

b) Se la parte decimale di nValue (nValueFract) è >= 0.5 si assegna a hzDim = value+2 e vtDim = value
c) Se la parte decimale di nValue(nValueFract) è < 0.5 si assegna a hzDim = value+1 e vtDim = value.

A questo punto si eseguono due cicli for della matrice per inizializzarla con tutti gli elementi nulli(infinito)
Si azzera hzIndex e vtIndex

Si esegue un ciclo for sul vettore fino a numberElement(che è la sua dimensione) e si eseguono due controlli:

a) Se hzIndex < hzDim ovvero se l'indice orizzontale è minore della dimensione orizzontale della nuova matrice si assegna a table[vtIndex][hzIndex] = vector[indexVector]
Si esegue un buildTableau per ripristinare le proprietà del tableau di Young
Si incrementa l'indice orizzontale
b) Se vtIndex < vtDim ovvero nel momento in cui l'indice orizzontale ha superato la dimensione orizzontale eseguo questo controllo e verifico che l'indice verticale è minore della dimensione verticale , se soddisfo tale condizione , incremento vtIndex ,

azzero hzIndex , ed assegna a table[vtIndex][hzIndex] = vector[indexVector]
eseguo nuovamente un buildTableau per ripristinare le proprietà del tableau di Young
ed in fine incremento l'indice orizzontale.

CAPITOLO 4: COMPLESSITA' COMPUTAZIONALE (CASO MIGLIORE CASO PEGGIORE)

Definizione termini:

h = dimensione orizzontale.
v = dimensione verticale.
n = quante volte l'utente inserisce un valore.
m = valore inserito dall'utente.
pot = dimensione verticale o orizzontale della matrice in alcuni casi specifici.
K = somma di tutte le operazioni con peso O(1).

Minimo:

Caso peggiore: O((v*h)*log(v*h))

Sapendo che il valore minimo nella table di Young si trovi in prima posizione, il caso migliore corrisponde al caso peggiore, poiché va a fare le stesse operazioni. Se invece consideriamo la table di Young vuota per il caso migliore avremo che esso corrisponde a: k.

Inserimento nuovo valore:

Caso peggiore: O((2*(v*h))+((v*h)*log(v*h))+(n*v*h))

Caso migliore: O(2(v*h)*log(v*h))

Da una table di N*M a N*N:

Caso peggiore: O((2*(v*h))+(pot^2)+((v*h)*log(v*h)))

Caso migliore: O((pot^2)+(4*log(v*h)))

Dati N^2 elementi, creare una table N*N che li contenga:

Caso peggiore: O(n+m^2+m+mlogm)

Caso migliore: O(m^2+mlogm)

Dati n valori, qual è la matrice che li può contenere?

Caso peggiore: O(n+(m^2)*n+(v*h)+m*log(v*h))

Caso migliore: k

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

Progettazione algoritmo per creazione e la gestione di Alberi. Utilizzato **IDE CodeBlock** - **Compilato sotto sistema UNIX**

Per maggiori dettagli vedere codice con relativi commenti inseriti.

Federico Maglione N86001405
Daniele Punziano N86001504

Anno accademico 2016-2017

Docenti: Murano Aniello
Di Stasio Antonio

INDICE

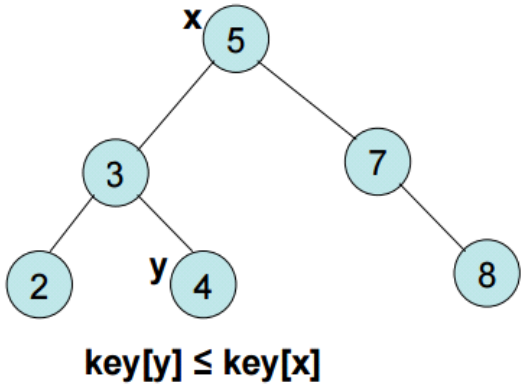
1 CAPITOLO : DESCRIZIONE ALBERI BINARI DI RICERCA (ABR)

Un albero binario è un albero dove ogni nodo ha al massimo due figli. Tutti i nodi tranne la radice hanno un nodo padre e ogni nodo ha una chiave (key).

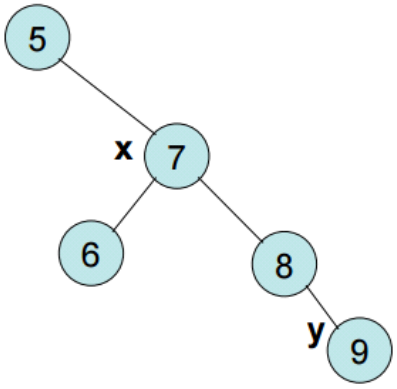
Sia x un nodo di un albero binario di ricerca.

1) Se y è un nodo appartenente al sottoalbero sinistro di x allora si ha key[y] ≤ key[x].

2) Se y è un nodo appartenente al sottoalbero destro di x allora si ha key[x] ≤ key[y]



key[y] ≤ key[x]



key[x] ≤ key[y]

CAPITOLO 2 : STRATEGIE PER RISOLUZIONE PROBLEMI E PRINCIPALI - STRUTTURE DATI UTILIZZATE

Strategia per la creazione di un Albero binario di ricerca:

La strategia applicata si basa sull'utilizzo di una struttura dati che noi chiamiamo structTree, nella quale sono indicati due puntatori corrispondenti al nodo destro e sinistro, ed un campo intero utilizzato per l'assegnazione di un valore.

Tale struttura viene utilizzata per creare n instanze di quest'ultima , allocandole dinamicamente tramite un richiamo alla funzione malloc, assegnando un valore al campo infoRoot e ponendo il puntatore left e right a NULL, successivamente tale "nodo" appena creato tramite la funzione insert verrà posto nella posizione corretta a fin che l'albero rispetti le condizioni di un ABR.

CREAZIONE MANUALE

In maniera più specifica:

- Si richiede il valore da inserire nel nodo,
- Si richiama la funzione insert che permette di inizzializzare la nuova struttura allocata inserendo il valore richiesto in input.
- Successivamente si richiederà un ulteriore valore da inserire in un nuovo nodo, fin quando l'utente non inserirà il valore -1 corrispondente all'uscita dalla funzione di creazione dell'ABR

CREAZIONE RANDOM:

- Si richiede la dimensione dell'abr da creare
- Si richiama la funzione insert che prenderà come argomento un valore dato dalla funzione rand() % N data dalla librerira time.h
- Una volta inseriti in maniera randomica il numero di valori definiti in input si uscirà dalla funzione.

Strategia per l'inserimento di un nuovo nodo:

La funzione di inserimento Insert permette di inserire un nuovo nodo nell'albero specificato avente come argomento un valore da inserire al suo interno. Tale funzione controlla che tale elemento sia presente o meno nell'albero ed asseconda dei casi lo posiziona nella posizione corretta.

In maniera più specifica:

- Confronta ogni nodo con l'elemento inserito in input, se tale elemento è più grande del nodo si passa a controllare il nodo destro , viceversa si passa a controllare il nodo sinistro. Se il valore e' uguale ad un nodo allora si esegue una swap tra il nodo sinistro successivo ed il nodo attuale creando un nuovo nodo e ponendolo tra questi due.
- Se invece il nodo non è presente si scende tutto l'albero fino ad arrivare alla foglia che corrisponde alla posizione corretta si crea un nuovo nodo dove inserire questo valore.
- Una volta inserito il valore, la funzione ricorsiva , ricorsivamente torna alla funzione precedente fino ad uscire definitivamente.

Strategia per la cancellazione di un nodo dall'albero:

La strategia applicata si basa sull’acquisizione di un valore numerico da standard input che a sua volta verrà confrontato con gli elementi già presenti nell'albero, e nel caso in sia presente verrà cancellato e verranno eseguite tutte le operazioni per ripristinare le proprietà di un abr.

In maniera più specifica:

- Si richiede un valore numerico da standard input.

- Si esegue un controllo per determinare che il valore appena inserito sia presente nell'albero.
- In caso affermativo si determina se il nodo contenente il valore da cancellare non abbia un nodo destro e sinistro, oppure un solo nodo destro o sinistro, oppure entrambi
- Si esegue una funzione differente per ogni caso descritto, così da poter cancellare il nodo e ripristinare le proprietà dell'abr

PS: In input verrà richiesto se cancellare tutte le occorrenze del valore indicato, in caso positivo si eseguira un ulteriore funzione che permetta di cancellare non solo il primo valore uguale trovato ma anche tutti i successivi.

Strategia per il calcolo della media dati n alberi random con dimensione d:

La strategia applicata si basa sulla creazione di n alberi random (il numero di alberi viene richiesto in input) ad ogni creazione di un albero, viene calcolata l'altezza di quest'ultimo e salvata in una apposita variabile, si incrementa una variabile che tiene conto di quanti alberi vengono creati, una volta creati tutti gli n alberi viene stampata la media aritmetica.

In maniera più specifica:

- Si richede il numero di alberi da creare e la dimensione di questi ultimi.
- Ad ogni creazione random degli alberi viene calcolata l'altezza dell'albero e salvata in una variabile, in più viene stampato a video l'albero creato con la sua altezza
- In fine viene eseguita una media aritmetica tra il numero di alberi e l altezza complessiva, la quale viene stampata.

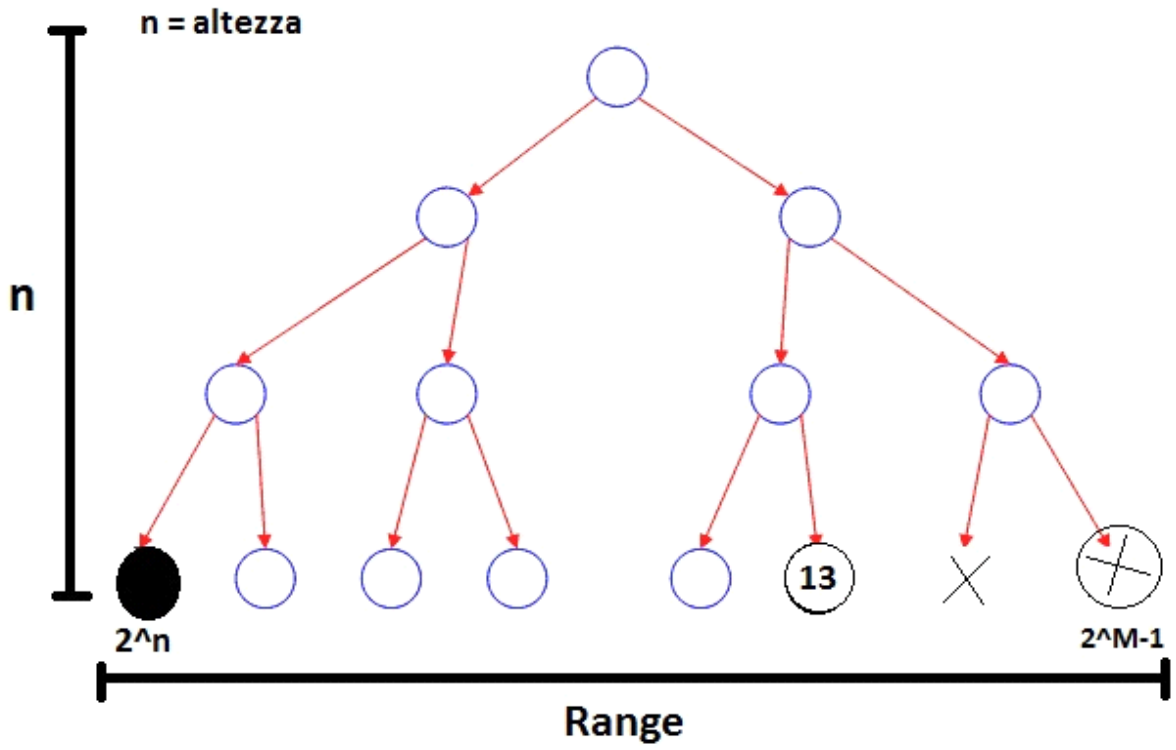
Studio della media di n alberi binario di ricerca random:

Il tools sviluppato permette di studiare l'andamento di una serie di successioni di alberi, dove per ogni successione verrà chiesto il numero di alberi da inserire e la loro dimensione, cosi da poter visualizzare le differende di altezza media tra una serie ed un altra. Di seguito alcune osservazioni fatte su un albero binario di ricerca:

Si può osservare che prendendo sotto analisi una sola serie con n alberi di dimensione d . Ogni albero avrà un altezza media che oscillerà in un "range" dato da alcune proprietà dell'albero stesso , ovvero:

Dati un numero di **nodi d** (dimensione). Si può osservare come **l'altezza massima hM** di un albero con d Nodi sia pari a **hM = D - 1** , mentre l'altezza minima di un albero con d nodi avrà alcune caratteristiche:dato che un abr e un albero binario , la condizione per la quale avremo un albero binario minimo è che ogni nodo abbia tutti e due i nodi associati ,e quindi d nodi saranno presenti un in range cioè,in termini matematici , definita n altezza minima **2^n <= d <= 2^N -1** ovvero **2^n è il numero di nodi minimo** per un altezza h mentre **2^N-1 è il numero di nodi massimi** per un altezza h.

Esempio : 13 --> 2^3 <= 13<2^4-1 ovvero 8<=13<=15 dove il range 8 e 16 è il numero di nodi per avere un albero con altezza 3.



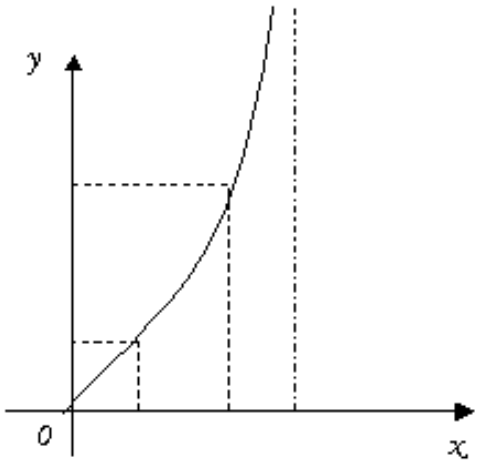
Detto ciò osservando tali considerazioni fatte su un numero n di alberi di dimensione crescente si può osservare che :

all'aumentare del numero di nodi (dimensione) il range definito diventa sempre più grande mentre le altezze hanno una variazione sempre più piccola. Es:

7 alberi alberi con dimensione crescente da 10 a 300:

10 nodi -> 2^3<= 10 <= 2^4-1 - 8<= 10 <=15 h: 3
50 nodi -> 2^5<= 50 <= 2^6-1 - 32<= 50 <= 63 h: 5
100 nodi -> 2^6<= 100 <= 2^7-1 - 64<= 100 <=127 h: 6
150 nodi -> 2^7<= 150 <= 2^8-1 - 128<= 150 <= 255 h: 7
200 nodi -> 2^7<= 200 <= 2^8-1 - 128<= 200 <= 255 h: 7
250 nodi -> 2^7<= 250 <= 2^8-1 - 128<= 250 <= 255 h. 7
300 nodi -> 2^8<= 300 <= 2^9-1 - 256<= 150 <= 511 h: 8
... h: 8
... h: 8
... h: 8
. . .

Graficamente risulterebbe molto simile all'immagine di seguito:



Tale grafico è la rappresentazione di un ***limite che tende all'infinito***, ovvero all'aumentare della dimensione dell'albero le medie si concentrarannò sempre di più verso un unico punto fino all'infinito.

Strategia per il merge tra due alberi binari di ricerca:

La strategia applicata si basa sulla creazione di due alberi (in maniera random o manuale), i quali vengono "filtrati" per determinare gli elementi in comune o valori doppi in ogni singolo albero cosi da eliminarli e rimanere solo un occorrenza di ogni valore, successivamente viene eseguito il merge tra i due alberi dove ogni nodo del secondo albero , partendo dalle foglie, viene "staccato" e "riattaccato" tramite l utilizzo di puntatori, al primo albero nella posizione corretta.

In maniera più specifica:

- Si creano due alberi (random o manuale)
- Si richiama una funzione che permette di ricercare nel secondo albero valori già presenti nel primo albero così da eliminarli, in più vengono eliminati anche valori doppi dal primo albero salvando solo un occorenza.
- Successivamente viene richiamata una piccola funzione che elimina anche più occorenze di uno stesso valore dal secondo albero (salvando solo un occorenza)
- In fine viene richiamata una funzione merge che visita il secondo albero fino ad arrivare alla foglia corretta (data da opportuni controlli) successivamente si visita il primo albero fino ad arrivare alla posizione corretta dove andare ad inserire il nodo ricercato nel secondo albero, così da unirli con l'uso opportuno dei puntatori.

PS: In input viene richiesto se utilizzare l'albero creato in precedenza o se crearne due nuovi in maniera random o manuale.

Strategia la rotazione dell'albero (verso destro o sinistra) un numero n di volte:

La strategia applicata permette di eseguire una rotazione dell'albero in un determinato verso , per un determinato numero di volte. L'algorimo che permette di eseguire una rotazione per volta , esegue ulteriori controlli sull'integrità dell'abr, affinchè dopo l'esecuzione rispetti tutte le sue proprietà.

In maniera più specifica:

- Si esegue un controllo sul numero di volte da eseguire la rotazione e il verso in cui effettuarla.
- Viene estratto il nodo sinistro , il figlio destro di questl'ultimo che viene fatto puntare al figlio finistro del padre (rotazione verso destra), inversamente viene estratto il nodo destro, il figlio sinistro di quest'ultimo che viene fatto puntare al figlio destro del padre (rotazione verso sinistra).
- Di seguito viene eseguita un ulteriore rotazioni fino al raggiungimento del valore stabilito.

PS: In input si richede il numero di volte ed il verso in cui ruotare l'albero.

Strategia per il bilanciamento di un albero:

La strategia applicata si basa sull utilizzo della funzione balanceTree che prende in input solo l'albero che deve essere bilanciato. Partendo dalla radice ci calcoliamo l'altezza dei due sotto alberi e facciamo una media per calcolarci l'altezza dell'albero bilanciato. Eseguiamo opportuni controlli per verificare che l'albero non sia già bilanciato oppure non sia possibile bilanciarlo. successivamente determinamo qual'è l altezze minore e quindi in quale direzione bisogna ruotare l'albero ad es: se l'altezza dell'albero sinistro è maggiore dell'albero destro allora ci calcoliamo il numero di rotazioni da effettuare per bilanciare l'albero verso destra, sottraendo all'altezza sinistra l'altezza opportuna per l'albero bilanciato. Infine richiamiamo la funzione rotation per bilanciare l'albero.

In maniera più specifica:

- Viene calcolata l'altezza partendo dalla radice dell'albero desto e sinistro
- Viene eseguita una media tra le due altezze
- Si controlla tramite la media e le due altezze se l'albero è gia bilanciato
- In caso contrario si controlla se è possibile bilanciare l'albero
- In caso controlario verifichiamo quale tra le due altezze è la maggiore così da poter determinare in che lato ruotare l'albero, una volta verificato salviamo in una apposita variabile la differenza tra l'altezza data e la media che corrisponde al numero di rotazioni da esegure, difatti successivamente richiamiamo la funzione rotation con il numero di rotazioni da eseguire e il verso.

Strategia per la stampa grafica di un abr:

La strategia applicata si basa sull'utilizzo di un array di struct allocato dinamicamente per ogni nodo dell'abr da stampare, nel quale vengono passati volta per volta i puntatori ad ogni nodo dell'abr. Successivamente si passa ad eseguire una stampa direttamente sull'array che tramite un calcolo dell'altezza, e di ogni singolo "livello" dell'abr (primo livello = radice , secondo livello = figli della radice ec..) ci permette di eseguire una stampa (formattata tramite un uso corretto di spazi " ") del vettore sotto forma di albero binario.

In maniera più specifica:

- Viene creato il vettore con ogni puntatore ad ogni nodo dell'abr
- Si eseguono calcoli per determinare il livello e l altezza dell'abr
- Si prosegue stampando tramite un apposito algoritmo ogni nodo nella posizione corretta
- Si dealloca il vettore creato liberando memoria.

Strutture dati utilizzate:
Le principali strutture dati utilizzate sono struct , e vettori (per la stampa). Utilizzo di puntatori e doppi puntatori.

3 CAPITOLO: DETTAGLI IMPLEMENTATIVI (I/0 & FUNCTION)

Variabili Globali:

- #define LIMIT 500

Librerie usate:

- <stdlib.h>
- <stdio.h>
- <limits.h>
- <time.h>

Funzione di inserimento.

void insert(tree *root,int value)

I/O:

- tree root: albero su cui lavorare
- int value: valore da inserire nell'albero

***Descrizione funzione: (Funzione ricorsiva*)**

Tale funzione ha come argomenti la radice (nodo) all'albero nel quale aggiungere un nuovo nodo "root" e il valore da inserire nel nuovo nodo.

Tale funzione ricorsiva richiama se stessa quando vengono rispettate varie condizioni:

- Se il nodo (root) è uguale è NULL "vuoto" andiamo ad inserire il nuovo valore in tale posizione richiamando la funzione initNode che permette l'allocazione di un nuovo nodo e l'inserimento del valore dato in input.
- Se il valore in input è minore del valore presente sul nodo richiamiamo la funzione insert sul nodo sinistro.
- Se il valore in input è maggiore del valore presente sul nodo richiamiamo la funzione insert sul nodo destro
- Se il valore in input è uguale al valore presente viene allocato un nuovo nodo per tale valore e viene posto tra il nodo attuale ed il nodo sinistro tramite un semplice swapping tra puntatori.

void insertNoEquals(tree *root,int value)

Tale funzione esegue la stessa cosa della precedente con la piccola differenza di non poter accettare valori uguali.

Funzione di cancellazione:

void deleteV(tree *root, int value, int choice);

I/O:

- tree *root:puntatore a puntatore all' albero su cui lavorare
- int value: valore cancellare dell'albero

Descrizione funzione: (Funzione ricorsiva)

Tale funzione ha come argomento la radice (nodo) dell'albero, il valore da eliminare ed un flag che permette di scegliere se eliminare tutte le occorenze di un valore o meno.

Tale funzione essendo ricorsiva richiama se stessa se vengono rispettate alcune condizioni:

- Si visità nodo per nodo l'albero fino a trovare o meno il valore da eliminare:
- Se il valore è presente si eseguono 3 operazioni differenti date da 3 situazioni differenti:
    - Se il nodo non ha nessun nodo sinistro ne destro (foglia) si libera la memoria e si pone il puntatore a tale nodo a NULL.
    - Se il nodo da eliminare ha un sono nodo (sinistro o destro) il puntatore tmp viene posto sul nodo da eliminare e il nodo (sinistro o destro) viene fatto puntare al posto del nodo da eliminare , successivamente viene liberata memoria del nodo puntato da tmp (che equvale a quello da eliminare)
    - Se il nodo ha entrambi i nodi sinistro e destro, viene eseguita una ricerca del nodo con valore minimo sull'albero sinistro del nodo corrente, il quale viene copiato al posto del valore del nodo corrente. Successivamente viene richiamata la funzione di cancellazione sul

nodo sinistro, e con valore minimo, che a sua volta asseconda dei casi esegue le opportune cancellazioni.

- La funzione equals permette , quando l'utente lo richiede in input, di andare ad eliminare tutte le occorenze del valore cercato.

PS: Si noti come il punto c non elimini un nodo ma permetta di riportare il nodo da eliminare in una posizone corretta che rispetti le condizioni a o b.

Funzione per il calcolo della media tra n alberi con dimensione d:

void calculateAVG(int dimension);

I/O:

- tree randRoot : usato per creare alberi random
- int number: numero di alberi su cui fare la media
- int array: array contente tutte le medie calcolate.
- int numer1: numero di serie di alberi da realizzare
- int j , index , count
- int h : altezza dell'albero
- float avg : variabile usata per contenere la media tra gli n alberi di ogni serie.

Descrizione funzione:

Viene inizializzato un puntatore a struttura a NULL, che ci permetterà di creare volta per volta un numero n di alberi . La funzione richiede all'utente quante serie di alberi studiare (cosi da visualizzare la differenza tra le medie calcolate alla fine su ogni serie), successivamente viene richiesta su quanti alberi eseguire la media e la dimensione di tali alberi, tutto questo eseguito in un ciclo for per ogni serie inserita.
Viene eseguito un ulteriore ciclo for per creare volta per volta un abr ranom, calcolare l'altezza dell'albero ed eseguire la somma tra tutte le altezze degli alberi creati.
La media di ogni serie verrà salvata in un apposito array che poi verrà stampato.

Funzione per il merge tra due alberi.

*void mergeTree(tree *T1, tree *T2);*

I/O:

- tree *T1, *T2 puntatore a puntatori degli alberi su cui fare il merge
- tree temp: puntatore di appoggio.

Descrizione funzione: (ricorsiva)

mergeTree permette di eseguire tre funzioni per il merge di due alberi. Di seguito:

searchAndDelete permette di ricercare gli elementi in comune dall'albero T2 rispetto l'albero T1 tramite una funzione di ricerca che ritorna 1 se trova un valore in comune.
Se vi è un valore in comune si esegue la funzione deleteV (che permette di cancellare il nodo associato al valore in comune trovato), successivamente viene eseguita una funzione double che permette di cancellare , se presenti, il valori in comune , dato quello trovato, che si ripetono più volte da T1 (salvando una sola occorrenza per ogni valore), in fine si controlla che l'albero T1 non sia nullo e che il nodo sinistro o destro non sia nullo richiamando ricorsivamente il searchAndDelete sul nodo sinistro o destro così da poter controllare la presenza o meno di altri valori in comune.

deleteDouble permette di ricercare valori uguali nell'albero T2, dato che due valori uguali (per come realizzato l'inserimento) verranno posti l'uno a sinistra dell'altro , ogni qual volta il nodo sinistro non è vuoto richiamiamo la funzione double sul nodo sinistro per verificare che non ci sia un ulteriore elemento uguale, se è presente viene eliminato salvandone una sola occorrenza.
Quindi la funzione, con chiamate ricorsive, visita nodo per nodo l'albero verificando che ogni nodo sinistro sia uguale o meno al precedente (padre).

Merge permette di eseguire una vera e propria unione tra i due alberi, staccando ogni nodo del secondo albero e per poi farlo puntare alla posizione corretta del primo albero

- Si controlla che i nodi dei due alberi siano non vuoti
- 1. Si verifica che il valore del nodo di T1 sia minore del valore del nodo di T2
- 2. Se lo è ci si salva in una variabile temporanea il nodo sinistro di T2
- 2. Si pone il nodo sinistro di T2 a null
- 2. E ci si richiama il merge passando T1 e tmp.

1. Si verifica che il valore del nodo di T2 sia minore del valore del nodo di T1

   2. Se lo è ci si salva in una variabile temporanea il nodo destro di T2

   2. Si pone il nodo destro di T2 a NULL

   2. E ci si richiama il merge T1 e tmp.

Quando T2 punterà a NULL si ritornerà indietro nella chiamata ricorsiva della funzione merge , dove T2 sarà proprio il nodo da far puntare a T1.

1. Si controlla se il nodo sinistro sia non vuoto o il nodo destro sia non vuoto:

   4. Se non è vuoto ci si richiama la funzione merge sul nodo sinistro o destro di T1 per n volte fino ad arrivare alla posizione nella quale il nodo sinistro o destro è NULL(posizione corretta dove inserire il nodo di T2

4.In tale posizione viene inserito T2.

Questa operazione verrà eseguita per ogni nodo di T2 fin quando tutti i nodi non siano stati collegati a T1.

Funzione per la rotazione di un albero verso destra o sinistra per un numero n di volte:

tree rotation(tree T, int to, int number);

I/O:

- tree T,tmp: puntatori ad alberi usati per ruotare
- inr to: direzone di rotazione
- int numero : numero di rotazioni
- int app: variabile di appoggio

Descrizione funzione:

Rotation prende in input un albero da ruotare, il verso in cui lo ruotiamo e quante volte lo si deve ruotare. Si esegue un controllo per verificare se sia vuoto o meno, in caso positivo viene stampato su stdOutput un messaggio di errore. Essendo rotation una funzione ricorsiva, si esegue un ulteriore controllo per verificare che il numero di rotazioni date in input siano state eseguite o meno, con relativo messaggio , se non eseguite si esegue un algoritmo di rotazione basato sulla direzione inserita in input. Esempio:

Eseguiamo una rotazioni verso sinistra, controlliamo , partendo dalla radice R, che l'albero sinistro S sia non vuoto, in caso di risposta positiva "stacchiamo il nodo destro" con l utilizzo di puntatori a struct, e lo "riattacchiamo" con il nodo sinistro della radice, e come nodo destro del nodo sinistro di S viene attaccata la radice.

Funzione per il bilanciamento di un albero binario di ricerca.

tree balanceTree(tree T);

I/O:

- int heightSX, int heaightDX : variabili per salvare l'altezza destra e sinistra dell'albero
- tree T puntatore all'albero da bilanciare
- int avg : variabile contente la media della somma delle altezze

Descrizione funzione:

La funzione balanceTree prende in input solo l'albero che deve essere bilanciato. Partendo dalla radice viene calcolata l'altezza dei due sotto alberi e la media tra la somma di queste due altezze, cosi da calcolarci l'altezza a cui portare l'albero da bilanciare. Se l'altezza dell'albero sinistro è maggiore dell'altezza dell'albero destro viene calcolato il numero di rotazioni da effettuare per bilanciare l'albero, sottraendo l'altezza di come dovrebbe essere l'albero bilanciato, infine richiamiamo la funzione rotation per bilanciare l'albero passando avg che adesso conterrà il numero esatto di rotazioni da eseguire.

Funzione per la stampa grafica di un abr:

printLevel(tree t);

I/O:

- tree T puntatore all'albero da bilanciare

Descrizione funzione:

La funzione printLevel prende in input solo l'albero che deve essere stampato. Ci calcoliamo l'altezza dell'albero e il numero massimo di elementi che possiamo trovare con quell'altezza. Come prima cosa dopo aver dichiarato l'array creiamo la prima struttura di posizione 0 in modo dinamico ed essa punterà alla radice. Partono due cicli for, dove il primo ciclo si basa sui livelli dell'albero mentre il secondo ciclo sulle potenze di 2 poiché ogni livello avrà 2^livello elementi (Es: livello 1 avremo 2^1 elementi, livello 2 avremo 2^2 elementi, ecc..). Usiamo due indici, indice i per scorrere e salvare le strutture nell'array e l'indice j per controllare gli elementi inseriti nell'array se hanno il sottoalbero sinistro e destro e posizionare i puntamenti nell'array di posizione i. Se invece non hanno figli allora verrà creata una struttura con valore -1 e con i due figli a NULL. Finiti i due cicli, abbiamo creato l'array da stampare. Per la stampa abbiamo usato altre variabili:

spazioRadice: che ci dice quanto dista la stampa della radice rispetto all'inizio riga.

spazioFigliPadre: indica la differenza di spazio che ci sta tra un nodo padre e i due figli.

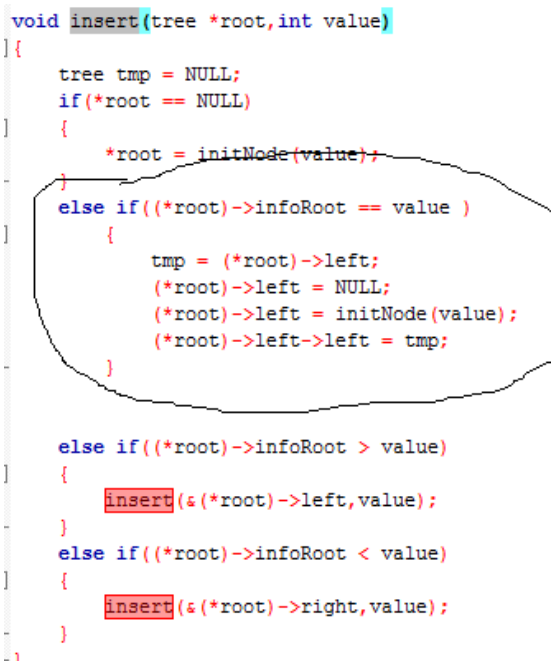
spazioFigli: e lo spazio che c'è tra i figli.

Usiamo la funzione printSpace per stampare tanti spazi bianchi in base alla variabile spazioRadice o spazioFigli. Stampiamo prima la radice e poi effettuiamo di nuovo i due cicli usati per la creazione dell'array. Ogni volta che avanza il livello allora dimezziamo lo spazioFigliPadre e lo assegniamo a spazioFigli. Nel secondo ciclo chiamiamo la funzione printSpace su spazioFigli e poi stampiamo il valore, e se il valore corrisponde a -1 stampiamo X. Dopo ci calcoliamo la lunghezza del numero inserito, (ES: se il numero inserito è 10, la lunghezza è 2) per inserire l'altro figlio dobbiamo cambiare la distanza tra di loro, e lo facciamo moltiplicando la distanza che c'è tra il padre e il figlio * 2 –la lunghezza del numero inserito (calcolato precedentemente).

Capitolo 4: Complessità computazionale

INSERIMENTO - CANCELLAZIONE:

- La complessità computazionale per **l'inserimento** non varia dalla solità è sempre O(h) con h altezza dell'albero poichè in aggiunta al solito inserimento ,troviamo le seguenti righe di codice le quali hanno costo tutte 1.



```
void insert(tree *root,int value)
{
    tree tmp = NULL;
    if(*root == NULL)
    {
        *root = initNode(value);
    }
    else if((*root)->infoRoot == value )
        {
            tmp = (*root)->left;
            (*root)->left = NULL;
            (*root)->left = initNode(value);
            (*root)->left->left = tmp;
        }

    else if((*root)->infoRoot > value)
    {
        insert(&(*root)->left,value);
    }
    else if((*root)->infoRoot < value)
    {
        insert(&(*root)->right,value);
    }
}
```

- La complessità per la **cancellazione** varia a seconda del tipo di cancellazione da eseguire, se vogliamo eseguire una cancellazione di un solo elemento , la complessità sarà sempre O(h) con h altezza dell'albero, mentre se vogliamo cancellare tutte le occorenze di un valore la complessità sarà O(h)+h poichè tramite la funzione equals ci richiamiamo la funzione di cancellazione un massimo di h volte massimo h volte (dato che ogni elemento uguale viene posto alla sinistra del primo elemento uguale trovato).

ROTATION:

- La complessità per la **rotazione** dipende dal numero di rotazioni da eseguire (n) , poichè è formulata da solo operazioni che hanno costo 1 e la chiamata ricorsiva in base al numero di rotazioni da eseguire quindi il caso peggiore sarà O(1)*n

MERGE:

- La complessità per il **merge** si suddivide in 3 sottofunzioni:
    - SearchAndDelete: Complessità caso peggiore 2*O(n)+O(h)+h (con 2*O(n) tra search e chiamate ricorsive della funzione, mentre O(h)+h tra deleteV e doubleN)

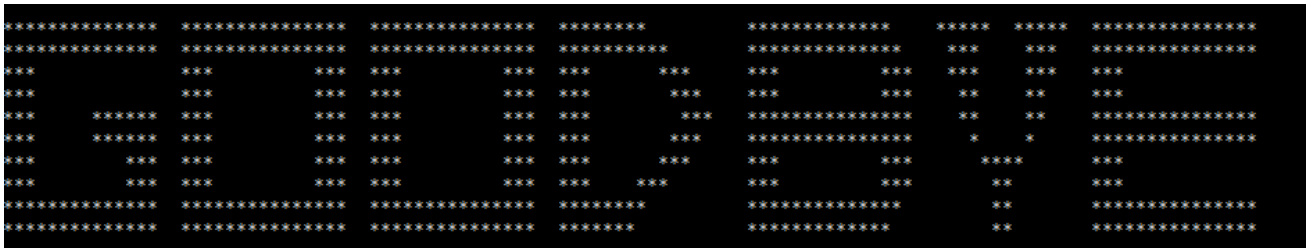
deleteDouble: O(n)+h ,O(n) per chiamate ricorsive + h per DoubleN

merge : Complessità 2*O(n) con n numero nodi

La complessità totale di merge tree è 4*O(n)+O(h)+O(n)+2h

BALANCE TREE:

- La complessità di balanca tree caso peggiore è log (n^2) + O(1)*m con n numero nodi ed m rotazioni.



DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

Progettazione algoritmo per creazione Tableau di Young. Utilizzato **IDE CodeBlock**.

Per maggiori dettagli vedere codice con relativi commenti inseriti.

Federico Maglione N86001405
Daniele Punziano N86001504

Anno accademico 2016-2017

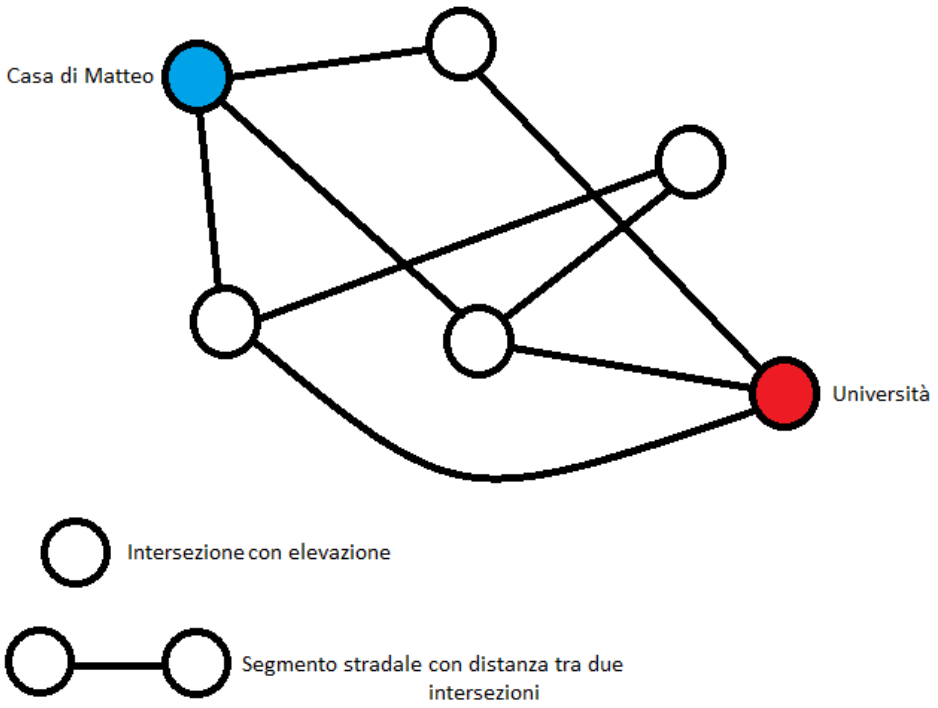
Docenti: Murano Aniello
Di Stasio Antonio

INDICE

CAPITOLO I: DESCRIZIONE DEL “CAMMINO DI MATTEO”

Maggio, l'estate è alle porte. Matteo, promettente studente universitario, viene afflitto da un pensiero che non gli dà respiro, lo soffoca addirittura di più di come succedeva quando pensava all'esame di algoritmi e strutture dati, poi superato felicemente al primo appello: la prova costume. Per rimettersi in forma per questo asfissiante evento Matteo ha deciso di andare in università correndo, e pianifica il percorso che ritiene il migliore per lo scopo. Decide che la soluzione migliore è scegliere una strada che vada per una parte in salita e poi una seconda parte completamente in discesa, in modo tale da bruciare e sudare di più nella parte in salita, e poi prendere una leggera brezza nella parte in discesa correndo velocemente fino l'università. La corsa inizierà dalla sua abitazione e terminerà in università, e tutto il suo percorso è dettagliato su una mappa dove ha segnato le strade con m segmenti stradali (cioè una strada tra due intersezioni) e n intersezioni. Ogni segmento stradale ha una lunghezza positiva e ogni intersezione ha un valore che indica la sua elevazione. Inoltre non esistono due intersezioni che si trovano allo stesso livello.

Punto 1:
Assumendo che ogni segmento stradale può essere classificato come segmento in salita oppure in discesa, sviluppare un algoritmo efficiente per trovare la strada più breve che soddisfi i vincoli descritti sopra.



CAPITOLO II: STRATEGIE PER RISOLUZIONE DEL PROBLEMA, DESCRIZIONE FUNZIONI PRINCIPALI

Premesse:

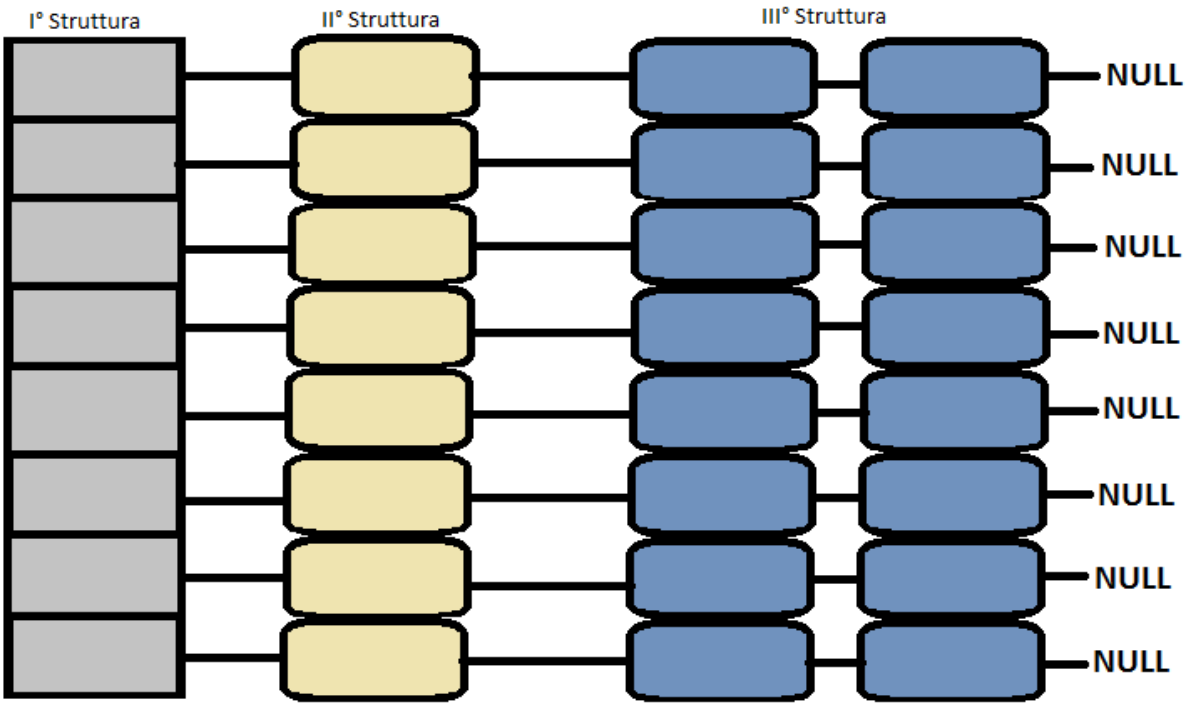
Per il calcolo del percorso minimo è stato creato ed utilizzato un **grafo aciclico direzionato**. Quindi non vi è la presenza di cicli tra due nodi, e il passaggio da un nodo ad un altro è di tipo monodirezionale.

Strategia per la creazione del grafo:

La strategia utilizzata si basa sulla creazione di 3 principali strutture:

- Una prima struttura che rappresenta un vettore di liste, che identifica il numero di vertici (intersezioni) del grafo.
- Una seconda struttura che rappresenta una sola lista che identifica il numero del vertice, l'elevazione (elevazione intersezione), ed un puntatore a liste.
- Una terza struttura che rappresenta una lista che identifica il punto di destinazione "target" la "distanza" tra due vertici ed un puntatore ad una lista dello stesso tipo.

Ad ogni creazione di un grafo viene allocata memoria per ogni struttura e vengono calcolate tutte le informazione (tramite l'ausilio di funzioni random o in maniera manuale) affinché si possa popolare il grafo. Il grafo risultante sarà di tipo **aciclico direzionato.**

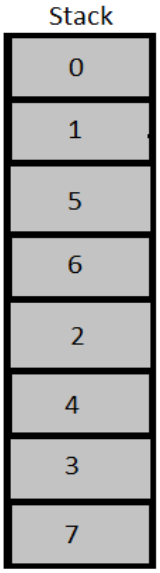


Strategia per l'ordinamento di un grafo tramite il DFS:

Data la definizione di ordinamento topologico per il quale i nodi di un grafo si definiscono **ordinati topologicamente** se sono disposti in modo tale che ogni nodo viene prima di tutti i nodi collegati ai suoi archi uscenti, allora è possibile eseguire tale ordinamento affinché sia possa identificare un percorso minimo in **tempo lineare**.

Tale definizione si basa sull'uso del **DFS**, il quale modificato opportunamente permette di eseguire l'ordinamento.

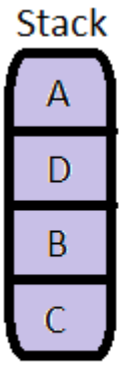
Esaminando il grafo attraverso il DFS, ogni qual volta si arriva ad un nodo che non presenta nodi in uscita, viene salvato il nodo in uno Stack (LIFO) e tornando indietro in maniera ricorsiva si va ad analizzare tutti i nodi, salvandoli.
Dopo l'esecuzione dell'ordinamento tramite il DFS su un generico grafo aciclico direzionato:

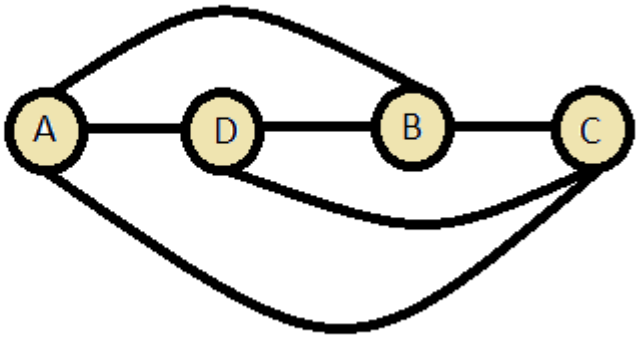


Ordine Topologico:
0 1 5 6 2 4 3 7

Strategia per la ricerca del percorso minimo:

La strategia applicata per la ricerca del percorso minimo si basa sull'utilizzo dell'ordinamento topologico. Utilizzando uno stack dove immagazzinare tale ordinamento è possibile tener traccia di ogni percorso proveniente da ogni singolo nodo:





Estraendo elemento per elemento dallo stack, tramite operazioni di pop, vi è la possibilità di controllare (in base all'ordine topologico) i nodi "target" provenienti da quello appena estratto verificandone e salvandone la distanza ed elevazione.

Ad esempio se volessi calcolare la distanza tra A e C eseguirei i seguenti passaggi:

Analizzo la lista di A ed ad ogni nodo ne scrivo la distanza:

A->B = 20
A->C =50
A ->D =15
Una volta esauriti i nodi passo al successivo nodo estratto dalla pop.
Analizzo la lista di D:

A -> D -> B = 25 poi eseguo un confronto tra i due percorso che arrivano a B:

A->D->B è minore di A->B ? No quindi mi salvo A->B in B poiché è quello più corto.
A->D->C =35. Chi è minore tra A -> D -> C e A->C? Mi salvo A->D->C in C
Analizzo la lista di B:
A->B->C = 40. Chi è minore tra A->B->C-> e A->D->C ? Mi salvo A->D->C in C
Analizzo la lista di C:

E' nulla quindi ho finito di elaborare. E mi sono trovato il percorso minimo che va da A a C cioè **A->D->C**

Il tutto scorrendo in maniera lineare gli archi e nodi O(V+E).

CAPITOLO III: DETTAGLI IMPLEMENTATIVI (I-O FUNCITON)

Direttive a preprocessore:

MAIN.C: NMAX Numero massimo di Nodi

GRAPH.C: ELEVATION Range massimo dell'elevazione random creabile.

GRAPH.H STACK_MAX Dimensione massima della stack.

Librerie utilizzate:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include <limits.h>
#include "graph.h"
```

Strutture utilizzate:

Terza struttura: Lista con nodo target e distanza e puntatore alla lista successiva.

```
struct listEdge {
   int target;
   int distanza;
   struct listEdge* next;
};
typedef struct listEdge *Elist;
```

Seconda struttura: Lista con elevazione e numero nodo collegato, con puntamento ad una terza struttura di tipo **toEdge(listEdge)**

```
struct listNode {
   int elevation;
   int nodeNumber;
   struct listEdge *toEdge;
};
typedef struct listNode *lNode;
```

Prima struttura: array di liste con puntatore a lista di tipo **lNode(listNode)**

```
struct GraphTab {
   lNode *adj;
   int n_nodes;
};
```

Struttura utilizzata per il salvataggio delle elevazioni già inserite, così da poter eseguire un controllo su elevazioni uguali:

```
struct listRand {
   int randNumber;
   struct listRand* next;

};
typedef struct listRand *randL;
```

Creazione del grafo Random:

graph randomGraph()
I/O

- n_nodes Numero nodi del grafo.

graph initGraph(int n_nodes) Allocazione prima struttura ed inserimento numero nodi.
I/O

- n_nodes Numero nodi del grafo.

void initializerGraph (graph G,int indexNode, randL numberList) permette di inizializzare la seconda struttura ed inserire l'elevazione tramite la funzione:
addElevation(&numberList,&numberList,rand()%ELEVATION)

- numberList lista utilizzata per il salvataggio di elevazioni già inserite.

Tale funzione ogni qual volta viene inserita una nuova elevazione verifica che non sia già presente all'interno della lista (scorrendola). In caso di risposta negativa si richiede una nuova elevazione.

addEdge(G); Tale funzione alloca memoria per la terza struttura, e permette il popolamento dei percorsi all'interno del grafo. Viene utilizzata una principale funziona dal nome *isCycle* che permette il controllo e modifica di possibili cicli.

Descrizione funzione ***RandomGraph:***
Quindi la funzione *randomGraph* restituisce un grafo creato con valori in modo random, ed utilizza due funzioni per inserire i valori:

1) *addElevation* che creato un' elevazione in modo randomico viene salvata all'interno di una lista che ogni volta viene ciclata per vedere se l'elevazione che vogliamo inserire già è stata inserita in un altro nodo e in quel caso si richiama la funzione ricorsivamente fino a quando non trova un'elevazione diversa che può essere inserita.
2) *isCycle* è una funzione per capire se l'arco che vogliamo aggiungere non crei cicli. Questa funzione chiama la funzione *isCycleTrue* che è una funzione ricorsiva. Essa controlla, a partire dalla destinazione tutti i nodi collegati a esso e per ogni nodo collegato controlla i nodi collegati a quest'ultimo nodo, e se trova un collegamento con il nodo sorgente (attraverso la funzione *checkInList*) allora vuol dire che stiamo creando un ciclo.

La funzione **manualGraph** esegue le stesse operazioni del *randomGraph* con l'unica differenza che tutti i valori (ovvero: elevazione di ogni nodo, per la creazione dell'arco il nodo sorgente, destinazione e la distanza che c'è tra i due archi) sono inseriti manualmente dall'utente.

Ordinamento grafo tramite il DFS:

Stack dfs(graph G,Stack S)

- Grafo G
- Stack S: Stack dove salvate l'ordinamento.

*void dfsOrdering(graph G, int index, int *array, Stack S*)

- Grafo G
- Stack S
- Index: nodo da analizzare
- Array: utilizzato per tener traccia dei nodi non ancora visitati.

Descrizione funzione: Si esegue un DFS , su ogni nodo del grafo, se non già visitato, si esegue il dfsOrdering che permette di analizzare tutto il nodo fin quando non si arriva ad un nodo senza archi in uscita dal quale tornare indietro ricorsivamente.

Ogni qual volta si arriva su un nodo senza archi in uscita o archi che portano a nodi già visitati ci si salva tramite un operazione di push quel nodo all'interno di uno stack , il quale dopo l' esecuzione di tutto il dfs conterrà l'ordinamento topologico del grafo.

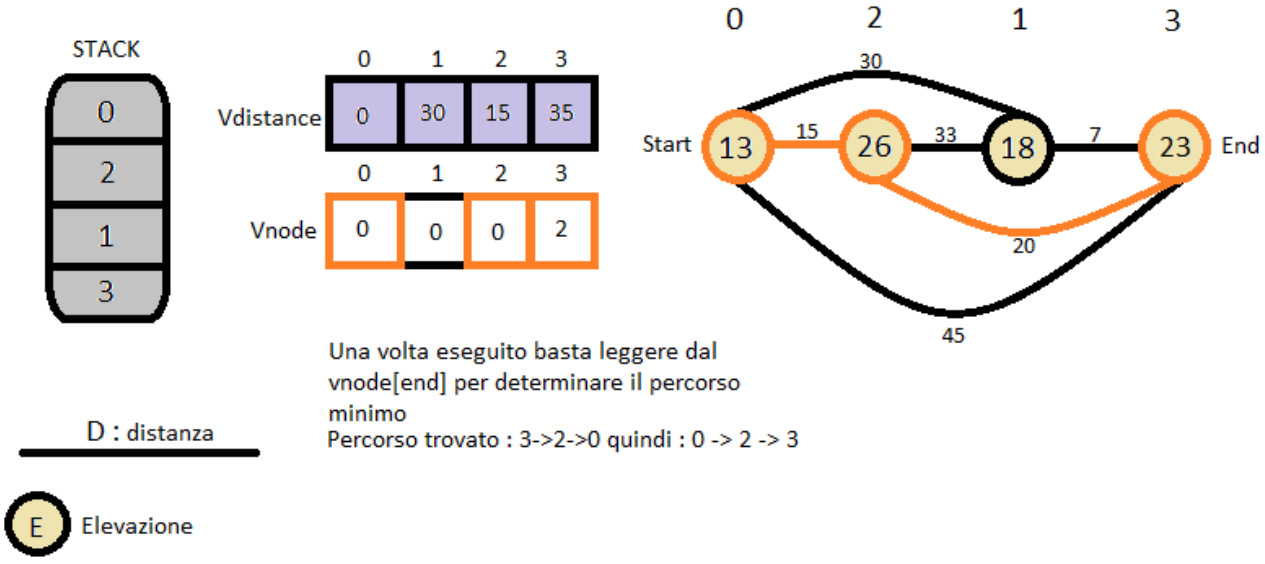
Calcolo percorso minimo:

void shortestPath(graph G, Stack S):

- Vertex: indice utilizzato per analizzare il nodo del elemento dello stack.
- Up,Down flag che verificano le condizioni di elevazione imposte dal progetto.
- VectorDistance: vettore che salva le distanze tra nodi.
- VectorNode: vettore che salva i nodi.

initializeNumber(vectorDistance, G->n_nodes,start): Permette di inizializzare il vettore delle distanze ponendo 0 nella posizione indicata dall'indice (nodo) di partenza e le altre posizioni a INTMAX.
Verificando che lo stack non sia vuoto si esegue un pop per estrarre il valore contenuto nell'ordinamento topologico dello stack. Si eseguono opportuni controlli per determinare che il nodo di partenza non sia posto dopo il nodo di arrivo (in base all'ordinamento topologico, che equivale ad un percorso non esistente).

Successivamente partendo dal nodo di start si eseguono i controlli sulle elevazioni e sulle distanze, salvando opportunamente le distanze inferiori (Vdistance)con i nodi (Vnode) che determinano quelle distanze:



Una volta eseguito lo shortest path leggendo il vettore Vnode dall'end è possibile estrapolare il percorso trovato.
Nell'esempio sopra: VectorNode[3] = 2 → VectorNode[2] = 0 Quindi :
Start : 0 -> 2 -> 3 End. Percorso minimo che soddisfa le proprietà di distanza ed elevazione.

CAPITOLO IV: COMPLESSITA' COMPUTAZIONALE

Percorso minimo:

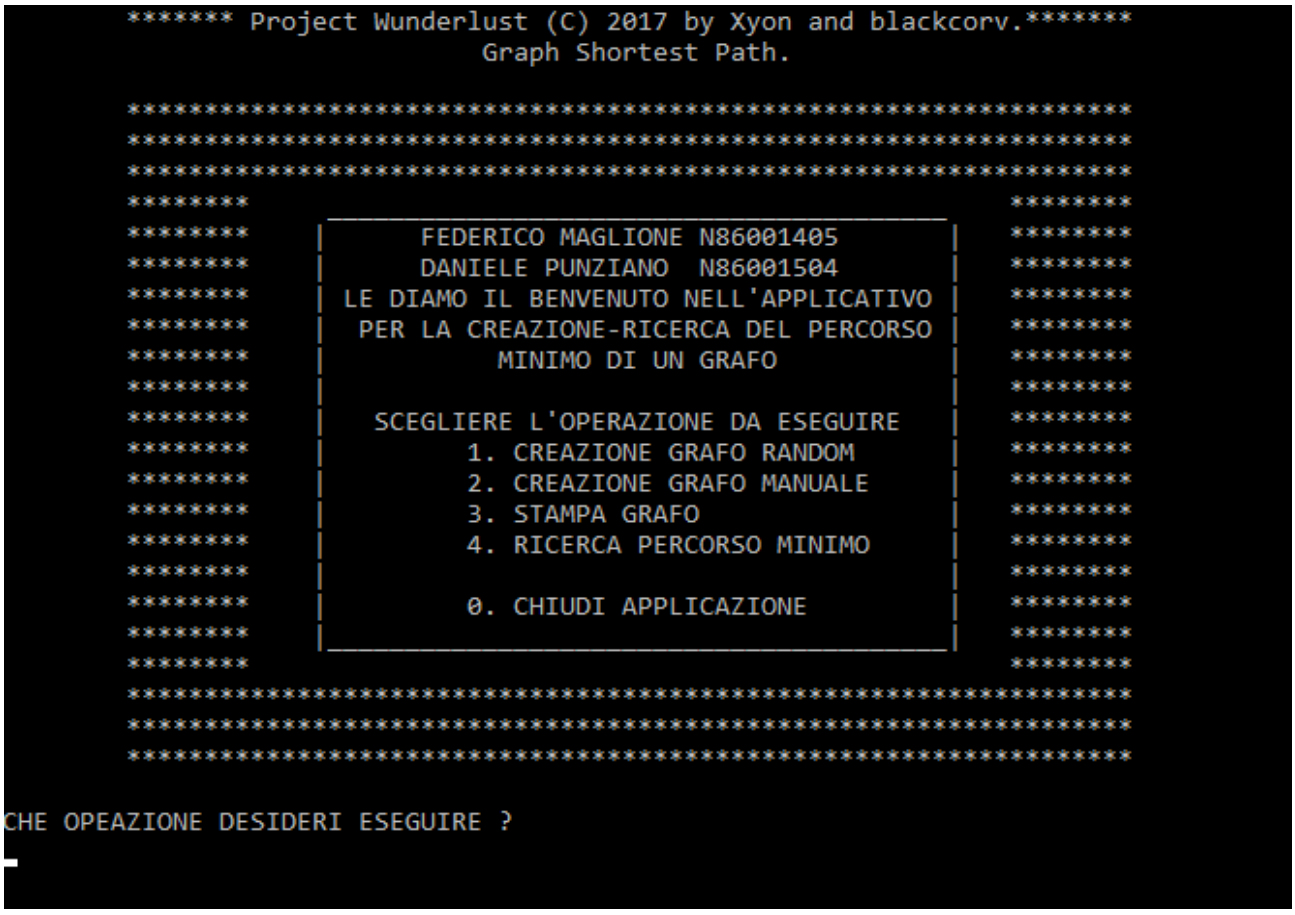
O(V+E) è la complessità delle seguenti funzioni:

- isCycleTrue
- dfs →dDfsOrdering
- shortestPath

PS: La complessità dello shortestPath è O(V+E) poiché , il calcolo del ordinamento topologico (DFS) ha costo O(V+E) successivamente si eseguono due cicli uno per tutti i vertici O(V) ed un altro per tutti i vertici adiacenti a quest'ultimo(archi) . La complessità per gli archi è di O(E) quindi il tempo totale è O(V+E).

CAPITOLO V: MANUALE D'USO APPLICATIVO

MENU:



Vi è la possibilità di scegliere una tra le 4 operazioni disponibili digitando il numero associato.

1. CREAZIONE GRAFO RANDOM:

Verrà richiesto il numero di nodi da creare tutto eseguito in maniera randomica.

```
INSERISCI IL NUMERO DI NODI DEL GRAFO
>
```

Successivamente sarà stampato il grafo creato con la possibilità di tornare al menù inziale:

```
0(54)-> 7|26|
1(59)-> 2|33| 0|5| 5|21|
2(19)-> 0|31| 6|15| 9|31| 7|41|
3(73)-> 8|4| 7|32| 5|3| 2|21| 1|40| 4|28|
4(64)-> 1|32| 0|5| 5|21| 7|23|
5(50)-> 7|40|
6(75)-> 9|8| 5|2|
7(27)->
8(5)-> 7|44| 4|48| 2|16|
9(4)-> 0|0| 7|26|

DIGITA 's' PER TORNARE INDIETRO
```

2. CREAZIONE GRAFO MANUALE:

Verrà richiesto l'inserimento del numero di nodi per il grafo, elevazione ,distanza per ogni nodo, e nodo target:

```
INSERISCI IL NUMERO DEI NODI: > 5

INSERISCI L'ELEVAZIONE PER IL 0 NODO > 22

INSERISCI L'ELEVAZIONE PER IL 1 NODO > 33

INSERISCI L'ELEVAZIONE PER IL 2 NODO > 44

INSERISCI L'ELEVAZIONE PER IL 3 NODO > 55

INSERISCI L'ELEVAZIONE PER IL 4 NODO > 66

VUOI INSERIRE UN'ARCO?  (1 = SI, 0 = NO) --> 1


DA QUALE NODO FAR PARTIRE L'ARCO? > 1

A QUALE NODO DESIDERI FAR ARRIVARE L'ARCO? > 3

QUALE E' LA DISTANZA TRA DI DUE ARCHI? > 22

VUOI INSERIRE ALTRI ARCHI?  (1 = SI, 0 = NO) -->
```

PS: Essendo un grafo aciclico direzionato non vi è la possibilità di inserire cicli , ne di inserire elevazioni uguali in base alle proprietà da rispettare del problema dato.

3. STAMPA GRAFO:

```
0(62)->
1(28)->
2(90)-> 6|20| 8|49| 9|22|
3(2)-> 0|12| 8|21| 2|3|
4(99)-> 9|1| 7|12| 3|17| 8|33| 6|40|
5(37)-> 4|12| 0|42| 7|12| 8|24|
6(68)-> 1|32| 0|0|
7(45)-> 2|11| 3|47| 9|46| 1|7|
8(7)-> 6|13| 1|16| 0|7|
9(77)-> 8|27|

DIGITA 's' PER TORNARE INDIETRO
```

4. CALCOLO PERCORSO MINIMO.

Viene richiesto il nodo di partenza ed arrivo per il quale cercare il percorso minimo, in base alle proprietà del grafo esso verrà ricercato:

```
0(7)-> 6|6| 3|8|
1(55)-> 6|27| 8|29|
2(88)-> 5|19| 9|16| 6|19|
3(98)-> 9|12|
4(63)-> 7|26| 1|6|
5(90)-> 3|42| 4|24| 9|31|
6(3)-> 8|28|
7(27)-> 0|36| 9|42| 8|46|
8(89)->
9(96)-> 6|27| 8|17| 1|8|

DI SEGUITO IL PERCORSO MINIMO TROVATO:

n:0(e:7 - d:0)   n:3(e:98 - d:8)   n:9(e:96 - d:20)   n:1(e:55 - d:28)

VUOI EFFETTUARE UN'ALTRA RICERCA?  (1 == SI, 0 == NO) -->
```

5. GOOD BYE