Laboratorio di Algoritmi e Strutture Dati

A.A. 2020-2021

Docente: F. Mogavero

Dispense tratte dal corso di Informatica a cura dello studente **S. Cerrone**
un po' di roba è stata rubata senza ritegno dal git di **V. Bocchetti**
https://github.com/luftmensch-luftmensch/Appunti-UNI

Elementi di Linguaggi C++

Tipi di dato astratti e implementazione in C++

Progettazione: libreria contenitore di dati

Strutture dati elementari

Alberi binari di ricerca & iteratori nei dati

Matrici & Grafi

Elementi di linguaggio C++

1. Struttura modulare di un programma

1.1 Le basi di un programma minimale

Ogni programma in C++ ha esattamente una funzione globale chiamata *main()* di tipo intero che fallisce se ritorna un valore diverso da zero. Di seguito un esempio minimale di programma:

```
#include <iostream>

int main() {
    std::cout << "Hello, World!\n";
    return 0;   //opzionale
}
```

La linea *#include<iostream>* introduce la libreria standard di C++ indispensabile per operazioni di I/O. Il namespace *std::* specifica che il nome *cout* si trova nella libreria standard *iostream*; se si aggiunge la linea di codice **using namespace std;** si rendono accessibili le funzionalità della libreria standard senza che sia necessario aggiungere *std::*.

1.2 Modularità

Generalmente è impensabile creare un programma in un singolo file, per cui viene strutturato in più file di diverse estensioni divise in file .hpp (header file) ed in file .cpp (implementation file) che vengono poi compilati per creare il file oggetto. Si assume che ogni header file (.hpp) che viene incluso come prototipo nelle altre librerie inizi e termini con le macro, dette anche header guards:

```
#ifndef TEST_HPP
#define TEST_HPP

    // ... Corpo

#endif
```

Così facendo si evitano di creare più copie dello stesso prototipo e che il compilatore si fermi per ridefinizioni dei file (è proprio un errore). Inoltre, queste macro sono risolte dal preprocessore. Il codice delle funzioni va scritto in un file .cpp separato che deve includere l'header.

1.3 Compilazione separata

La compilazione e linking devono essere fatti separatamente, per la compilazione sono due le modalità:

- File bash: Classico build.sh che ripete ogni volta la compilazione di tutti i file con la conseguenza di aumentare rapidamente i tempi per la compilazione;
- Makefile: ricompila solo i file che hanno avuto variazioni, rendendo la seconda compilazione molto più veloce rispetto alla prima (se ovviamente non cambio tutti i file).

La fase di Linking è l'ultimo passaggio nella creazione di un eseguibile ed aggiunge ai file oggetto il codice delle librerie esterne necessarie per il programma.

2. Tipi fondamentali, puntatori e riferimenti

2.1 Dichiarazione di variabili

Le variabili in C++ devono essere dichiarate prima del loro impiego e, a differenza del C, il C++ permette la dichiarazione delle variabili all'interno di qualsiasi blocco dove però la loro visibilità rimane ovviamente circoscritta. Per ogni variabile occorre indicare il tipo e il nome assegnato seguito dal punto e virgola, opzionalmente si può assegnare anche un valore alla variabile così dichiarata.

2.2 Tipi fondamentali

I principali tipi standard di variabili in C++ sono: ***int*** (numeri interi 32bit), ***float*** (numeri in virgola mobile in 32bit), ***double*** (numeri in virgola mobile in 64 bit), ***char*** (un carattere occupa 8 bit) e ***bool*** (può assumere solo i valori ***true*** o ***false***).

I qualificatori ***short*** e ***long*** si applicano al tipo int e cambiano i bit da 32 a, rispettivamente, 16 e 64, mentre ***signed*** e ***unsigned*** sono qualificatori che possono essere assegnati a ***char*** o ***int*** e determinano se le rispettive variabili possono assumere o meno valori negativi.

I tipi ***const*** non possono essere modificati durante l'esecuzione del programma, è utile quando si vuole avere la certezza che quel valore non venga modificato. Ovviamente vanno inizializzate durante la dichiarazione della variabile: ***const double pi = 3.141592654***

2.3 Puntatori

Il puntatore non è altro che una variabile numerica il cui valore è un indirizzo di memoria, la definizione del puntatore è ***tipo*variabile***, è possibile inizializzare il puntatore ma deve essere ovviamente l'indirizzo e non il valore puntato. La sua lunghezza è 32bit o 64bit e dipende dalla macchina (attualmente le macchine sono a 64bit). Si può rendere nullo un puntatore facendolo puntare all'indirizzo 0 della memoria (nullptr) che è semplicemente un indirizzo inesistente usato anche per "pulire" un puntatore.

Non posso puntare ad una variabile di tipo ***const*** poiché quest'ultime vengono memorizzate in un'area di memoria (area di testo) esterna all'area stack ed all'area heap dove lavora il puntatore, ammenoché non dichiaro un puntatore di tipo ***const tipo****, poiché è sicuro che non si alteri la variabile ***const*** tramite operazioni di scrittura, è possibile comunque cambiare l'indirizzo puntato ammenoché non si dichiari ***const*** anche il puntatore con, ad esempio, ***const char*const p***.

Con ***void*variabile*** si rende indefinito il puntatore, la differenza con gli altri puntatori sta nel fatto che con ***void*** non ho limiti sul tipo di variabile a cui posso puntare, nota bene che ***void*** definisce una variabile indirizzo di memoria. Per usare però una variabile di tipo ***void*** bisogna fare il cast di quella variabile dichiarandone il tipo, ad esempio ***cout << *(static_cast<char*>p) << endl;*** poiché solo ***cout << *p << endl*** mi darebbe errore. Lo static cast è prerogativa del C++ e si assicura che il casting che sto facendo sia possibile e non causi problemi come la frammentazione della memoria, a differenza del C che casta indipendentemente con ***(char*)p*** (cosa che si può fare anche in C++).

2.4 Riferimenti

Il passaggio dei parametri viene fatto di default per copia, ma al fine di evitare che ci sia un costo enorme di copia ogni volta che si chiami una funzione (con magari strutture di 200 elementi) posso passare la variabile per riferimento con il carattere ***&*** così da non dover copiare la variabile, inoltre in questo modo è possibile far modificare direttamente la variabile alla funzione, ovviamente nel caso si volesse evitare una modifica non voluta si può passare la variabile come riferimento di tipo costante con ***const tipo&***.

Un riferimento al valore di ritorno di una funzione non ha senso poiché è una locazione temporanea, più generalmente qualsiasi riferimento ad una locazione temporanea della memoria è logicamente scorretto. Il riferimento è la stessa locazione della variabile a cui mi riferisco, differentemente dai puntatori.

I riferimenti precedentemente descritti sono di tipo **Lvalue** (ha un indirizzo a cui il programma può accedere). I riferimenti Lvalue e Rvalue sono sintatticamente e semanticamente simili, ma seguono regole in qualche modo diverse. I riferimenti **Rvalue** si dichiarano con l'operatore ***&&***, e consentono di generare un valore temporaneo. Praticamente Rvalue sono tipi di riferimenti che è possibile inserire a destra dell'assegnazione.

In genere, i Rvalue sono valori il cui indirizzo non può essere ottenuto dereferenziandoli, perché sono di natura temporanea (come i valori restituiti da funzioni o chiamate esplicite al costruttore).

2.5 Funzioni standard Move e Swap

template <class T>
typename remove_reference<T>::type&& move (T&& arg) noexcept;
Restituisce un riferimento rvalue come argomento, questa è una funzione di supporto per forzare lo spostamento della semantica sui valori, anche se hanno un nome: l'uso diretto del valore restituito fa sì che *l'argomento* venga considerato un Rvalue. Passando un oggetto a questa funzione, si ottiene un valore che fa riferimento ad esso. Praticamente consente di spostare le proprietà di un oggetto senza doverle copiare.

template <class T> void swap (T& a, T& b);
Scambia semplicemente I valori di a e b. Nel caso gli argomenti fossero degli array allora avremo il seguente
template <class T, size_t N> void swap(T (&a)[N], T (&b)[N]);
N.B.: La swap di puntatori chiama internamente il move assignement (vedi capitolo 10).

3. Allocazione dinamica della memoria

3.1 Stack e Heap

Lo stack è un area di memoria che contiene i record di attivazione delle funzioni (parametri e variabili locali); ad ogni chiamata di funzione viene creato un blocco nello stack che termina con la terminazione della funzione stessa. L'area di memoria heap, invece, non è allocata automaticamente ma viene allocata solo su esplicita richiesta del programma (tramite puntatori), e occupa memoria fino ad una sua esplicita deallocazione.

3.1 Malloc e free

Un oggetto allocato dinamicamente nello heap rimane fintanto che non viene deallocato esplicitamente; nel linguaggio C si alloca con una chiamata di sistema, *malloc()* e si dealloca con la chiamata *free()*, questi vengono sostituiti nel linguaggio C++ dagli operatori new e delete, rispettivamente, che anche se equivalenti alla *malloc()* e alla *free()* hanno delle differenze e per ciò non vanno mai mischiate (C++ permette di utilizzare anche *malloc()* e *free()* ma è meglio evitare poiché gli operatori new e delete sono più sicure).

3.2 New e Delete

Solitamente questi due operatori vanno richiamati, per quanto possibile, all'interno dei costruttori (la new) e dei distruttori (la delete).

NEW: Alloca la memoria per un oggetto o un array di oggetti di tipo Type dall'archivio gratuito e restituisce un puntatore di tipo diverso da zero all'oggetto. La sintassi è la seguente: ***new-type-name [new-initializer];***
Quando new viene usato per allocare memoria, il costruttore dell'oggetto viene chiamato dopo che la memoria è stata allocata, se l'operatore ha esito negativo non restituisce mai un puntatore a null come la malloc() ma solleva un'eccezione (bad alloc) quando si eccede dalle dimensioni della memoria.

DELETE: la sintassi è ***delete ptr;*** nel caso di un elemento, mentre per gli array si usa ***delete [] ptr;*** l'argomento deve essere un puntatore a un blocco di memoria allocato in precedenza per un oggetto creato con l'operatore New. L'operatore delete ha un risultato di tipo void e pertanto non restituisce alcun valore. L'utilizzo dell'operatore delete su un oggetto dealloca la memoria. Un programma che dereferenzia un puntatore dopo l'eliminazione dell'oggetto può ottenere risultati imprevisti o un arresto anomalo. Quando delete viene usato per deallocare la memoria per un oggetto della classe C++, il distruttore dell'oggetto viene chiamato prima della deallocazione della memoria dell'oggetto (se l'oggetto ha un distruttore).

3.3 Memory Leak

Un memory leak è un particolare tipo di consumo non voluto di memoria dovuto alla mancata deallocazione dalla stessa, di variabili/dati non più utilizzati da parte dei processi.

4. Tipi definiti dall'utente

4.1 Strutture (Struct)

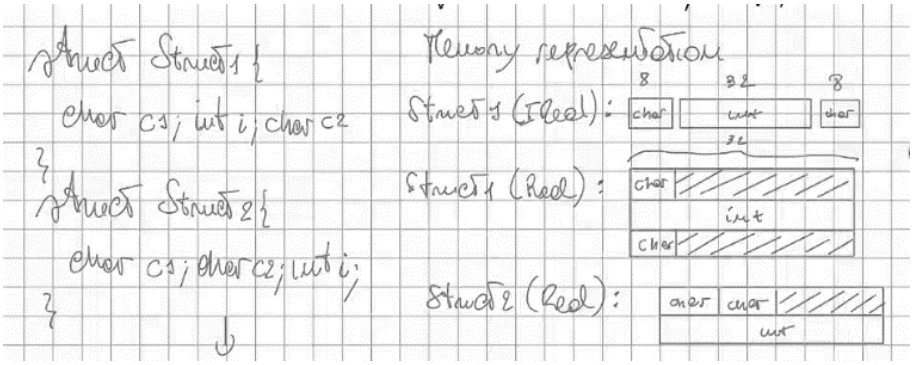
La Struct è un gruppo di variabili, anche di tipo diverso, aggregate insieme con un unico nome. La dichiarazione è la seguente:

```
struct Nome_struttura{
        tipoX Nome_campo1;
        …
        tipoY Nome_campoN;
};
```

L'accesso al campo si fa con ***Nome_struttura.Nome_campo;***. La struttura si dichiara con ***Struttura nome1;*** mentre per dichiarare e inizializzare la struttura si possono usare le seguenti linee:

Struttura Nome1 = {campo1, campo2, campo3};
Struttura Nome1{campo1, campo2, campo3}; ← si preferisce alla prima
Struttura Nome1(campo1, campo2, campo3); ← possibile se si definisce un costruttore

Le struct sono classi con accesso pubblico agli attributi e alle funzioni senza nessuna restrizione.



4.2 Enumerazione

Dichiarazione in C++ : ***enum class Nome { elenco di parole non chiavi separate da virgole }*** (nel linguaggio C si omette "class", ciò significa che i valori non sono locali alla classe definita ma globali per tutte le classi, quindi non posso definire parole uguali per enumerazioni diverse).
Gli elementi possono essere confrontati (==, !=, <,>,<=,>=) e assegnati (:=), Le variabili al tipo enumerativo possono ovviamente avere un valore iniziale di default (Il confronto tra due enumeration diverse non è possibile; non è definito).

5. Libreria iostream e funzioni di I/O

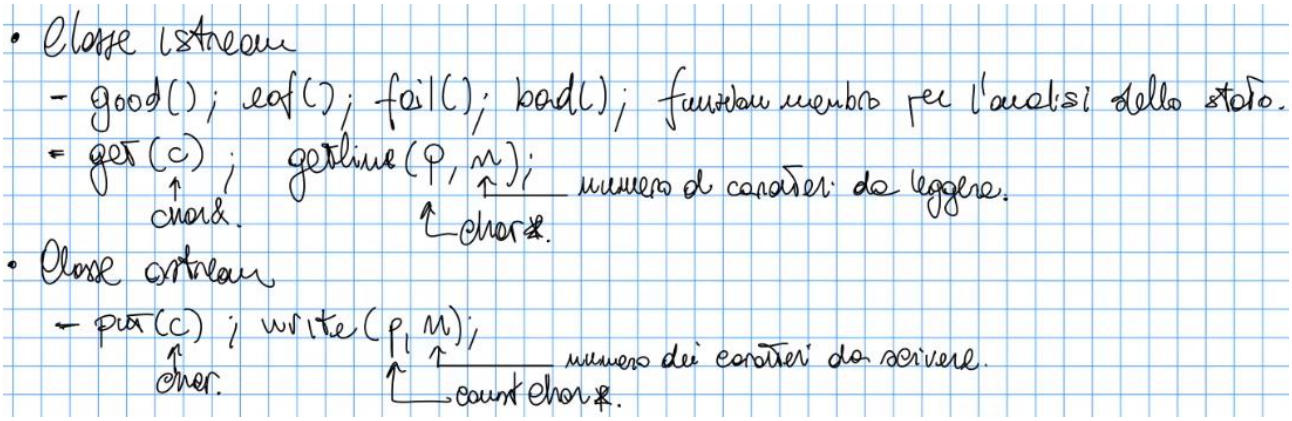
5.1 "put-to" e "get-trow"

Questa libreria permette l'accesso ad input e output di default con gli operatori ***cin >>*** e ***cout <<***, rispettivamente. Praticamente "<<" è un operatore chiamato put-to e si applica a oggetti di tipo "ostream" mentre ">>" get-trow si applica a oggetti di tipo "istream". Questi sono definiti per tutti i tipi di default mentre per i tipi definiti dall'utente bisogna gestirli tramite l'overloading.

5.2 Overloading

```
ostream& operator << (ostream& os, const Struttura st){
return os << st.campo1 << "-" << st.campo2 << "-" << st.campo3 << endl;
}
istream& operator >> (iostream& is, Struttura st){
return is >> st.campo1 >> st.campo2 >> st.campo3;
}
```

Le classi istream e ostream hanno molti metodi (non solo quei due operatori) in particolare funzioni membro per l'analisi dello stato:



Nota bene che nel caso di campi protected o private bisogna anteporre "***friend***" a ***i/ostream&***

6. Libreria string

6.1 Il tipo string

Il tipo di dato string non è un tipo di dato base, infatti bisogna includere la libreria *#include<string>* (essa è comunque compresa nella libreria iostream), questo dato semplifica le operazioni con le stringhe di caratteri, per dichiarare un oggetto string si possono usare *string Nome = "valore stringa"; string Nome ("valore stringa"); string Nome ={"valore stringa"};*. Gli spazi o i tab sono considerati separatori.

Come si può immaginare le stringhe sono confrontabili con i classici operatori e si assume un ordinamento lessicografico (dalla prima lettera a sinistra in ordine alfabetico secondo codice ASCII). Anche gli operatori put to e get trow sono già definiti. Per accedere ad un carattere specifico: *[i];*

6.2 Metodi principali

Tra le funzioni più comuni nella libreria string si ricordano:

- *size();* ritorna il numero di elementi del vettore stringa
- *empty();* ritorna vero se la lunghezza della stringa è 0, falso altrimenti
- *front();* ritorna un riferimento al primo carattere della stringa
- *back();*. ritorna un riferimento all'ultimo carattere della stringa

Utile, inoltre, è la funzione *var.substr(i, n);* che crea una sottostringa dalla posizione i alla posizione i+n-1; esempio: *string var="Alone"; cout << var.substr(1, 2);* stamperà a video "lo".

7. Generazione pseudo-casuale di numeri

7.1 Libreria <random>

#include<random> è una libreria che consente di generare numeri casuali tramite la funzione *default_random_engine nome_random (random_device{}());* che crea un seed random con cui posso definire un tipo di distribuzione specifico, ad esempio se voglio una distribuzione uniforme definisco la funzione:

```
uniform_int_distribution <type> nome_funzione (i,j);
for (int i=0, i<15, i++) {
cout << nome_funzione(nome_random); \\ nome_random è la funzione descritta precedentemente.
}
```

Questo tipo di generatori sono molto utili per testare il codice del programma in maniera più efficiente possibile.

7.2 Esempio in C++

```
// Random generation a la C++
default_random_engine gen(random_device{}());
uniform_int_distribution<uint> dist(7, 35);
for(uint i = 0; i < 15; i++) { cout << dist(gen) << " "; }; cout << endl;
```

Che stamperà a video numeri casuali: `35 24 12 16 15 19 15 25 31 7 24 30 30 19 27`
ovviamente se rieseguo la funzione i numeri saranno diversi (tranne se usi quella merda di windows che non aggiorna il cazzo di seed).

8. Eccezione e relativa gestione

8.1 Struttura di gestione elementare

Di seguito un codice elementare per la gestione di eccezioni:

```
try {
        …
        throw someException();
        …
}
catch (someException name)
{
        …Gestione dell'eccezione
}
catch (…) {
        …
        throw  ← Risolleva l'eccezione
        …
}
```

8.2 Eccezioni utili dello standard library

Le eccezioni sono un dato di tipo classe **exception** che poi è suddivisa ulteriormente in:

- logic_error
  - length_error
  - out_of_range
- runtime_error
  - overflow_error
  - underflow_error
- bad_alloc

→ **Exception**

Questi tipi di eccezione sono già presenti nella standard library, e noi utilizzeremo principalmente le eccezioni di tipo logico, ovvero i logic_error.

Posso dichiarare una funzione che non solleva alcuna eccezione scrivendo *noexcept* alla fine della funzione: *type nome_funzione* ***noexcept****;* praticamente la funzione nome_funzione non può generare alcuna eccezione.

Ovviamente ci sono eccezioni che non è possibile gestire come ad esempio il segmentation fault (si tenta di accedere ad una locazione di memoria a cui non è permesso accedere).

Il bad_alloc è un tipo di eccezione che non può mandare messaggi, in ogni caso è una funzione che dovrebbe evitare l'utente stesso, quindi non va trattata.

9. Puntatori a funzione

9.1 Possibili usi

I puntatori a funzione non puntano ad un dato bensì ad un codice eseguibile, quindi è un modo di chiamare più funzionalità che sintatticamente hanno lo stesso prototipo (stessi valori di input) che voglio chiamare tramite l'indirizzo. Praticamente, un puntatore a funzione permette la chiamata di funzioni diverse con lo stesso prototipo conoscendone l'indirizzo.

È utile, generalmente, per il passaggio di funzioni come parametro di altre funzioni (non note) oppure per le funzioni di callback (ossia le funzioni che astraggono sul tipo di dato o comportamento degli algoritmi) che sono alla base dell'implementazione della struttura ad oggetti.

9.2 Definizione

I puntatori a funzioni nel linguaggio C si definiscono con ***typedef type (*NomeFunzione) (lista parametri);*** mentre per la definizione in C++: ***typedef std:function <type(lista parametri)> NomeFunzione;*** dove std:function richiede #include <functional>

La chiamata della funzione così definita è: ****NomeFunzione(…)*** oppure, semplicemente, ***NomeFunzione(…)***

9.3 Esempio

+ **void sort (int* A, int n);** ← Ordinamento prefissato
+ **bool (*Compare) (int x, int y);**
+ **void sort (int* A, int n, Compare comp){**

…
... if (comp(x,y)){
…
}else{
…
}
}

Uso del confronto

9.4 Quick sort

Il quick sort è un algoritmo ricorsivo che viene utilizzato per ordinare i dati in un array, esso ha, generalmente, prestazioni migliori tra quelli basati sul confronto. In questo algoritmo la ricorsione viene fatta non dividendo il vettore in base agli indici ma in base al suo contenuto.

I passi ricorsivi sono:

1. Trovare il pivot che divide l'array in due parti;
2. Applicare il Quick Sort sulla parte sinistra;
3. Applicare il Quick Sort sulla parte destra.

Di seguito una possibile implementazione:

```
// low -> indice di partenza, high -> indice finale
void quickSort(int* Vec, int low, int high){
    int r;
    if (low < high) {
        r = partition(Vec, low, high);
        quickSort(Vec, low, (r - 1));
        quickSort(Vec, (r + 1), high);
    }
}
```

l'algoritmo partiziona sceglie un valore dell'array che fungerà da elemento "spartiacque" tra i due sotto-array, detto valore pivot. Partition sposta i valori maggiori del pivot verso l'estremità destra dell'array e i valori minori verso quella sinistra.

```
int Partition(int* Vec, int sx, int dx) {
    int pivot = Vec[sx];
    int i = sx - 1;
    int j = dx + 1;
    while (i >= j) {
        // cerca il primo elemento da destra <= al pivot
        while (Vec[j] <= pivot) { j--; }
        // cerca il primo elemento da sinistra >= al pivot
        while (Vec[i] >= pivot) { i++; }
        // scambia gli elementi se i != j
        if (i < j) {
            swap(Vec[i], Vec[j])
        }
    }
    return j; // indice mediano
}
```

10. Classi, oggetti e template

10.1 Concetti di base

Una **classe** rappresenta, sostanzialmente, una categoria particolare di oggetti e, dal punto di vista della programmazione, è anche possibile affermare che una classe funge da tipo per un determinato oggetto ad essa appartenente (dove con tipo si intende il tipo di dato, come lo sono gli interi o le stringhe). Diremo, inoltre, che un particolare oggetto che appartiene ad una classe costituisce un'istanza della classe stessa.

L'**ereditarietà** è un meccanismo che consente di scomporre il modello dati della nostra applicazione in una gerarchia di classi che definiscono strutture dati e comportamenti differenziati. Da un punto di vista pratico, tale decomposizione consente di trasmettere un insieme di caratteristiche comuni da una classe base ad una derivata senza che ciò comporti una duplicazione del codice, offrendo allo stesso tempo l'opportunità di adattare o estendere il comportamento a casi d'uso specifici.

Un problema ricorrente con l'**ereditarietà multipla** è la definizione di classi derivate a partire da classi base che condividono un antenato comune (il così detto *diamond problem*). La ridondanza può essere eliminata ricorrendo all'eredità virtuale. L'effetto della parola chiave ***virtual*** in una clausola di derivazione è quello di forzare il compilatore a includere la base virtuale una sola volta nella definizione degli oggetti derivati, anche se essa appare più volte nella catena di derivazione. In questo modo si ottimizza l'uso delle risorse, e si risolvono a monte eventuali conflitti di nomi.

10.2 Definizione di classe in C++

Il costrutto è **class NomeClasse {**

private:

opzionale, se non dichiarati private o public i metodi e le variabili sono sempre private; sono attributi utilizzabili solo all'interno della classe stessa; sono definiti attributi e funzioni membro privati

protected:

attributi e metodi visibili solo dalle classi figlie oltre, ovviamente, alla classe NomeClasse

public:

attributi e metodi visibili a tutti

}

Il seguente elenco puntato è stato scritto da M.G. Carofano.

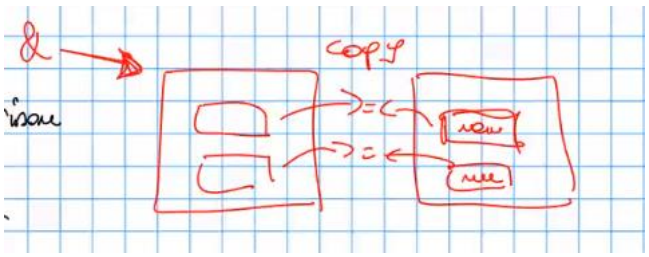
Le classi possono ereditare in maniera pubblica, protetta o privata:

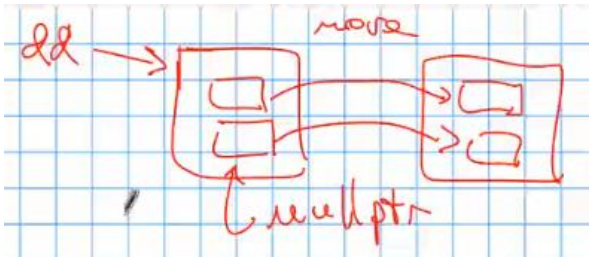
- Ereditarietà privata:
    - Tutto ciò che è private rimane private (non visibile)
    - Tutto ciò che è protected diventa private (non visibile)
    - Tutto ciò che è public diventa private (non visibile)
- Ereditarietà protetta:
    - Tutto ciò che è private rimane private (non visibile)
    - Tutto ciò che è protected rimane protected (visibile solo nella gerarchia e non agli utenti esterni)
    - Tutto ciò che è public diventa protected (visibile solo nella gerarchia e non agli utenti esterni)
- Ereditarietà pubblica:
    - Tutto ciò che è private rimane private (non visibile)
    - Tutto ciò che è protected rimane protected (visibile solo nella gerarchia e non agli utenti esterni)
    - Tutto ciò che è public rimane public (visibile anche da utenti esterni)

10.3 Costruttori

I costruttori principali di ogni classe sono quello di default (**default constructor**) `NomeClasse();` e il costruttore specifico (**specific constructor**) `NomeClasse (parametri);`.

Altri costruttori utili che di solto si trovano nelle classi sono il costruttore di copia (**copy constructor**) `NomeClasse (const NomeClasse&);` che ha lo scopo di creare un nuovo oggetto con lo stato del vecchio oggetto (i valori dei dati sono gli stessi dell'oggetto copiato ma sono appunto copie, quindi sono nuovi dati); ed il costruttore di spostamento (**move constructor**) `NomeClasse (NomeClasse&&) noexcept;` che crea una nuova classe con esattamente gli stessi stati dalla classe da cui si effettua lo spostamento (quest'ultima dovrebbe poi essere messa in uno stato vuoto, al fine di poterlo deallocare).





N.B.: nei move (constructor o assignment) posso aggiungere la signature noexcept poiché sono sicuro (avendo già oggetti completamente funzionanti) che nello spostamento non si possano generare eccezioni.

10.4 Distruttore

Metodo che viene automaticamente chiamato dal compilatore e si definisce con `~NomeClasse();` quindi i casi in cui bisogna invocare manualmente il distruttore sono rari. Il distruttore serve ovviamente a gestire l'allocazione dinamica, infatti nel caso in cui si hanno variabili allocate dinamicamente nella classe bisogna gestire la loro deallocazione nel distruttore.

I distruttori delle classi padri in una gerarchia vanno sempre definiti `virtual` per evitare problemi seri di segmentazione. Non facendolo, infatti, potrei rischiare di deallocare solo la classe padre e non l'istanza creata dalla classe figlio. Immaginando le classi come scatole sarebbe come togliere una scatola (classe padre) dall'interno di un'altra scatola più grande (classe figlia), lasciando però intatta quest'ultima.

10.5 Altri metodi utili

Oltre ai metodi classici che si possono definire in una classe, utili sono gli operatori seguenti:

- **Comparion:** `bool operator == (const NomeClasse&) const noexcept;`
  metodo per il confronto di due oggetti di tipo classe, ovviamente ci si aspetta che non viene modificato alcun dato, né da sinistra né da destra (da cui i const) e che questo metodo non provochi nessuna eccezione (noexcept).

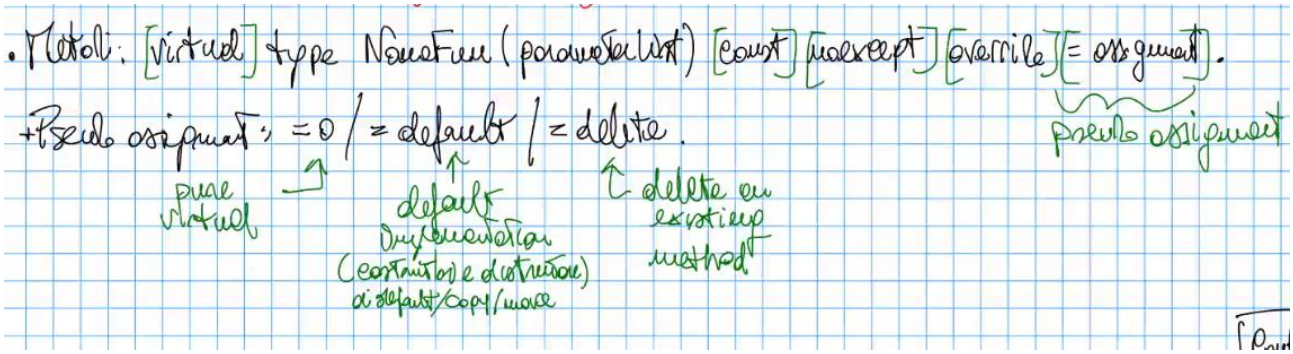
- **Copy assignement**: **NomeClasse& operator = (const NomeClasse&);**
- **Move assignement**: **NomeClasse& operator = (const NomeClasse&&) noexcept;**

Gli ultimi due sono operatori di assegnamento, il primo di tipo copia e il secondo di tipo move, praticamente corrispondono ai costruttori di tipo copy e move; con la differenza che non costruisco per la prima volta un oggetto NomeClasse ma qui ho già un oggetto NomeClasse e voglio semplicemente che assuma il valore di un altro oggetto o per copia oppure per spostamento.

10.6 Sintassi dettagliata

Tutto ciò scritto tra parentesi [] sono parametri opzionali.

In C++ per definire l'ereditarietà tra le classi uso la seguente sintassi al fine di rendere ben definita la gerarchia: **class NomeClasse: [virtual][protected/public] NomeClasseBase [,… ,AltreClassi, …] {..};** (la classe NomeClasse eredita dalla classe NomeClasseBase e dalle eventuali AltreClassi) tutto ciò che è *private* dalla superclasse non è utilizzabile dalla classe figlio, mentre se nella classi base ho protected e public allora anche nella figlia i metodi saranno protected o public ammenoché non abbassi il livello manualmente (posso passare da public a protected o a private, o da protected a private ma non il viceversa).



L'**override** sta ad indicare che il metodo della classe appartiene già alla classe padre ma si vuole dare una nuova implementazione. Lo pseudo assegnamento **default** dice al compilatore di andare a cercare la funzione nella classe superiore, mentre, con **= 0** indico una classe virtuale pura, ovvero una funzione che può essere specificata all'interno della gerarchia (so che ci deve essere almeno una classe figlio che deve avere questa funzione) ma che non so attualmente come implementarla, in modo tale che un'altra classe all'interno della gerarchia ne faccia semplicemente l'override. Infine, con **delete** indico al compilatore che quella funzione non è presente in quella classe (non ho modo di implementarla) ma esisterà all'interno della gerarchia.

In C++ non è possibile definire una interfaccia ma possiamo paragonare una classe astratta pura (con soli metodi virtuali) alle interfacce Java.

Per rendere accessibili le funzionalità di una libreria senza l'include posso definire concretamente di quale classe sto definendo tale metodo con: **type NomeClasse::NomeFunzione(parametri)[specifiche]{#code}**

10.7 Template

I template servono per definire modelli di classi e funzioni parametrizzando i tipi utilizzati: nelle classi, si possono parametrizzare i tipi dei dati-membro; nelle funzioni (e nelle funzioni-membro delle classi) si possono parametrizzare i tipi degli argomenti e del valore di ritorno. In questo modo si raggiunge il massimo di indipendenza degli algoritmi dai dati a cui si applicano: per esempio, un algoritmo di ordinamento può essere scritto una sola volta, qualunque sia il tipo dei dati da ordinare.

I template sono risolti staticamente (cioè a livello di compilazione) e pertanto non comportano alcun costo aggiuntivo in fase di esecuzione; sono invece di enorme utilità per il programmatore, che può scrivere del codice "generico", senza doversi preoccupare di differenziarlo in ragione della varietà dei tipi a cui tale codice

va applicato. Ciò è particolarmente vantaggioso quando si possono creare classi strutturate identiche, ma differenti solo per i tipi dei membri e/o per i tipi degli argomenti delle funzioni-membro.

Una classe template è definita dall'espressione *<typename NomeDato>* (significa che questa classe sarà sostituita dal tipo di dato NomeDato). Di seguito un esempio di template:

Vector.hpp

```
template <typename Data>
class Vector {
    private:
        uint size = 0;
        Data* Elements = nullptr:
    public:
        Vector() = default;
        Vector(uint n);
        ~Vector();
        ...
        Data& operator[](ulong);
};
#include "Vector.cpp"
```

Vector.cpp

```
template <typename Data>
Vector<Data>::Vector(uint n)
    Elements = new Data[n];
    size = n;
}
template <typename Data>
Vector<Data>::~Vector(){
    delete[] Elements;
}
```

È sempre preferibile separare le definizione dalle implementazioni (Vector.hpp e Vector.cpp) ma ciò comporta di dover inserire per i template alla fine del codice #include "codice.cpp" altrimenti non compila.

Tutte le operazioni new e delate che non sono dichiarate nell'implementazione di costruttori e distruttori vengono definite "nude", questo tipo di operazione va evitata perché potrebbe essere fonte di errore.

Per utilizzare il template precedentemente definito si usa *Vector<Data> Nome[x];*

11. Differenze e similitudini con i linguaggi C e Java

11.1 Differenza tra C++ e C

C++ è (quasi) un sovrainsieme del linguaggio C:

- C++ aggiunge classi, overloading, riferimenti, template, eccezioni e altro al linguaggio C
- A livello semantico è praticamente lo stesso, ha poche differenze (per costrutti validi in entrambi i linguaggi) ad esempio *int x = sizeof('a')* la variabile a è considerata intero in C ma di tipo char in C++.

11.2 Differenza tra C++ e Java

Java è un linguaggio orientato agli oggetti che si è ispirato al C++. Esistono svariate differenze semantiche e anche molte differenze a livello pratico, alcune differenze fondamentali sono rappresentate in tabella.

C++	Native executable	No Garbage Collector	Allocation on Stack & Heap	Operator Overloading	Multiple Inheritance (eredità)	Metaprogramming via Template
Java	Virtual Machine	Garbage Collector	Allocation (almost) on heap	No Operator Overloading	Single Inheritance	Parametrization via Generics

Tipi di Dato Astratti e Implementazione C++

12. Abstract Data Types (ADT)

12.1 Introduzione

Per comprendere il concetto di Tipo di Dato Astratto (Abstract Data Types nel seguito ADT) è utile confrontare ADT con la nozione di procedura (funzione).

- La procedura (funzione) generalizza il concetto di operazione. Oltre alle operazioni elementari definite in un particolare linguaggio, quali addizione e sottrazione, è possibile, definire proprie operazioni ed applicarle a dei dati, che non necessariamente sono dati di tipo elementare, dati cioè definiti dal linguaggio utilizzato.

- La procedura (funzione) permette di incapsulare parte di un algoritmo localizzando in una determinata porzione di programma gli aspetti che riguardano un determinato comportamento, o determinate operazioni.

12.2 Definizione di ADT

Possiamo pensare ad un ADT come ad un modello matematico con un insieme di operazioni definite sul modello. Ad esempio, un insieme di interi, con le operazioni di unione, intersezione, etc.. sono un semplice esempio di ADT.

In un ADT le operazioni possono avere come operando non solo una istanza dell'ADT ma anche altri tipi di operando. Ad esempio, nell'ADT "insieme di interi" il risultato di un'operazione può essere una nuova istanza di ADT ma anche un semplice intero.

Le due proprietà viste per le procedure valgono anche per un ADT.

- Gli ADT sono una generalizzazione di dato primitivo (real, integer, etc.), così come la procedure è una generalizzazione di operazione.

- Un ADT incapsula un tipo di dato nel senso che la definizione di dato e tutte le operazioni relative sono localizzate in un determinato segmento di programma. Se si vuole cambiare la definizione di un determinato ADT, modificare o aggiungere operazioni, sappiamo dove guardare, e siamo sicuri che una eventuale modifica non altera il comportamento del programma che lo utilizza. Per un programma un ADT è a tutti gli effetti un dato primitivo.

12.3 Astrazione

Astrazione dei dati: Separazione del concetto “dato” dalla sua rappresentazione.

“*Un’astrazione deve denotare le caratteristiche essenziali di un oggetto contraddistinguendolo da tutti gli altri oggetti e fornendo, in tal modo, dei confini concettuali ben precisi relativamente alla prospettiva dell’osservatore*” ~ Grady Booch.

Strutture Dati Astratte (ADT): strutture dati definite a prescindere dalla loro implementazione.

Opacità dei dati: ogni programma può accedere alla struttura esclusivamente tramite i servizi (concetto di interfaccia).

13. Implementazione

13.1 Stack

```
template <typename Data>
class Stack: virtual public Container {

public:

  // Destructor
  virtual ~Stack() = default;

  // Copy assignment
  Stack& operator=(const Stack&) = delete;

  // Move assignment
  Stack& operator=(Stack&&) noexcept = delete;

  // Comparison operators
  bool operator==(const Stack&) const noexcept = delete;
  bool operator!=(const Stack&) const noexcept = delete;

  // Specific member functions
  virtual void Push(const Data&) = 0;
  virtual void Push(Data&&) noexcept = 0;
  virtual Data& Top() const = 0;
  virtual void Pop() = 0;
  virtual Data TopNPop() = 0;

};
```

13.2 Queue

```
template <typename Data>
class Queue: virtual public Container {

public:

  // Destructor
  virtual ~Queue() = default;

  // Copy assignment
  Queue& operator=(const Queue&) = delete;

  // Move assignment
  Queue& operator=(Queue&&) noexcept = delete;

  // Comparison operators
  bool operator==(const Queue&) const noexcept = delete;
  bool operator!=(const Queue&) const noexcept = delete;

  // Specific member functions
  virtual void Enqueue(const Data&) = 0;
  virtual void Enqueue(Data&&) noexcept = 0;
  virtual Data& Head() const = 0;
  virtual void Dequeue() = 0;
  virtual Data HeadNDequeue() = 0;

};
```

13.3 Binary Tree

```
template <typename Data>
class BinaryTree: virtual public InOrderMappableContainer<Data>,
                  virtual public InOrderFoldableContainer<Data>,
                  virtual public BreadthMappableContainer<Data>,
                  virtual public BreadthFoldableContainer<Data> {

protected:

  using BreadthMappableContainer<Data>::size;

public:

  struct Node {

  // Destructor
  virtual ~BinaryTree() = default;

  // Copy assignment
  BinaryTree& operator=(const BinaryTree&) = delete;

  // Move assignment
  BinaryTree& operator=(BinaryTree&&) = delete;

  // Comparison operators
  virtual bool operator==(const BinaryTree&) const noexcept;
  virtual bool operator!=(const BinaryTree&) const noexcept;

  // Specific member functions
  virtual Node& Root() const = 0;

  // Specific member functions (inherited from MappableContainer)
  using typename MappableContainer<Data>::MapFunctor;
  virtual void MapPreOrder(const MapFunctor, void*) override;
  virtual void MapPostOrder(const MapFunctor, void*) override;

  // Specific member functions (inherited from FoldableContainer)
  using typename FoldableContainer<Data>::FoldFunctor;
  virtual void FoldPreOrder(const FoldFunctor, const void*, void*) const override;
  virtual void FoldPostOrder(const FoldFunctor, const void*, void*) const override;

  // Specific member functions (inherited from InOrderMappableContainer)
  virtual void MapInOrder(const MapFunctor, void*) override;

  // Specific member functions (inherited from InOrderFoldableContainer)
  virtual void FoldInOrder(const FoldFunctor, const void*, void*) const override;

  // Specific member functions (inherited from BreadthMappableContainer)
  virtual void MapBreadth(const MapFunctor, void*) override;

  // Specific member functions (inherited from BreadthFoldableContainer)
  virtual void FoldBreadth(const FoldFunctor, const void*, void*) const override;

};
```

13.4 Iterator

```
template <typename Data>
class Iterator {

public:

  // Destructor
  virtual ~Iterator() = default;

  // Copy assignment
  Iterator& operator=(const Iterator&) = delete;

  // Move assignment
  Iterator& operator=(Iterator&&) noexcept = delete;

  // Comparison operators
  bool operator==(const Iterator&) const noexcept = delete;
  bool operator!=(const Iterator&) const noexcept = delete;

  // Specific member functions
  virtual Data& operator*() const = 0;
  virtual bool Terminated() const noexcept = 0;

};
```

13.5 Matrix

```
template <typename Data>
class Matrix: virtual public MappableContainer<Data>,
              virtual public FoldableContainer<Data> {

public:

  // Destructor
  virtual ~Matrix() = default;

  // Copy assignment
  Matrix& operator=(const Matrix&) = delete;

  // Move assignment
  Matrix& operator=(Matrix&&) noexcept = delete;

  // Comparison operators
  bool operator==(const Matrix&) const noexcept = delete;
  bool operator!=(const Matrix&) const noexcept = delete;

  // Specific member functions
  virtual inline ulong RowNumber() const noexcept { return row; }
  virtual inline ulong ColumnNumber() const noexcept { return clm; }
  virtual void RowResize(const ulong) = 0;
  virtual void ColumnResize(const ulong) = 0;
  virtual bool ExistsCell(const ulong, const ulong) const noexcept = 0;
  virtual Data& operator()(const ulong, const ulong) = 0;
  virtual Data const& operator()(const ulong, const ulong) const = 0;

};
```

Progettazione: Libreria Contenitore di Dati

14. Container

14.1 Introduzione

Le classi container (contenitore) sono una delle applicazioni più usate del costrutto template in C++.

Una classe contenitore è una struttura dati che si presta alla gestione di oggetti dello stesso tipo, per mezzo di operazioni di inserimento, indicizzazione e cancellazione.

Essendo Container il concetto più alto livello di contenitore le informazioni sono poche, infatti è presente solo la size e delle funzioni Empty (chiede se è vuoto) e Clear (pulisce il contenitore).

Si noti che un contenitore non parla mai del tipo di dati che contiene, a differenza dei contenitori specifici.

14.2 Astrazione

```
class Container {

protected:

  ulong size = 0;

public:

  // Destructor
  virtual ~Container() = default;

  // Copy assignment
  Container& operator=(const Container&) = delete;

  // Move assignment
  Container& operator=(Container&&) noexcept = delete;

  // Comparison operators
  bool operator==(const Container&) const noexcept = delete;
  bool operator!=(const Container&) const noexcept = delete;

  // Specific member functions
  virtual inline bool Empty() const noexcept { return (size == 0); }
  virtual inline ulong Size() const noexcept { return size; }
  virtual void Clear() = 0;

};
```

15. Contenitori specifici

15.1 Linear Container

Dati ordinati linearmente secondo l'occupazione in memoria, e questo ci dà l'opportunità di poter aggiungere le funzioni Front (che ci dà il primo elemento in memoria) e Back (ultimo elemento).

In modo simile, avendo un organizzazione lineare posso assegnare un indice agli elementi e quindi prevedere un operatore di accesso all'elemento i-esimo.

```
template <typename Data>
class LinearContainer: virtual public Container {

public:

  // Destructor
  virtual ~LinearContainer() = default;

  // Copy assignment
  LinearContainer& operator=(const LinearContainer&) = delete;

  // Move assignment
  LinearContainer& operator=(LinearContainer&&) noexcept = delete;

  // Comparison operators
  bool operator==(const LinearContainer&) const noexcept = delete;
  bool operator!=(const LinearContainer&) const noexcept = delete;


  // Specific member functions
  virtual Data& Front() const = 0;
  virtual Data& Back() const = 0;
  virtual Data& operator[](const ulong) const = 0;

};
```

15.2 Testable Container

Rappresenta i contenitori testabili, e ciò significa che posso testare l'esistenza di un elemento (aggiunta di un metodo di tipo Exists). Il metodo Exist non è ovviamente implementabile a questo livello essendo il testable container una classe astratta.

```
template <typename Data>
class TestableContainer: virtual public Container {

public:

  // Destructor
  virtual ~TestableContainer() = default;

  // Copy assignment
  TestableContainer& operator=(const TestableContainer&) = delete;

  // Move assignment
  TestableContainer& operator=(TestableContainer&&) noexcept = delete;

  // Comparison operators
  bool operator==(const TestableContainer&) const noexcept = delete;
  bool operator!=(const TestableContainer&) const noexcept = delete;

  // Specific member functions
  virtual bool Exists(const Data&) const noexcept = 0;

};
```

15.3 Mappable Container

```
template <typename Data>
class MappableContainer: virtual public Container {

public:

  // Destructor
  virtual ~MappableContainer() = default;

  // Copy assignment
  MappableContainer& operator=(const MappableContainer&) = delete;

  // Move assignment
  MappableContainer& operator=(MappableContainer&&) noexcept = delete;

  // Comparison operators
  bool operator==(const MappableContainer&) const noexcept = delete;
  bool operator!=(const MappableContainer&) const noexcept = delete;

  // Specific member functions

  typedef std::function<void(Data&, void*)> MapFunctor;

  virtual void MapPreOrder(const MapFunctor, void*) = 0;
  virtual void MapPostOrder(const MapFunctor, void*) = 0;

};
```

Si noti il puntatore a funzione MapFunctor che si utilizza all'interno delle map. Quest'ultime sono funzioni che ci permettono di poter applicare ad ogni elemento del contenitore un tipo di funzione (MapFunctor).

15.4 Foldable Container

```
template <typename Data>
class FoldableContainer: virtual public TestableContainer<Data> {

public:

  // Destructor
  virtual ~FoldableContainer() = default;

  // Copy assignment
  FoldableContainer& operator=(const FoldableContainer&) = delete;

  // Move assignment
  FoldableContainer& operator=(FoldableContainer&&) noexcept = delete;

  // Comparison operators
  bool operator==(const FoldableContainer&) const noexcept = delete;
  bool operator!=(const FoldableContainer&) const noexcept = delete;

  // Specific member functions

  typedef std::function<void(const Data&, const void*, void*) noexcept> FoldFunctor;

  virtual void FoldPreOrder(const FoldFunctor, const void*, void*) const = 0;
  virtual void FoldPostOrder(const FoldFunctor, const void*, void*) const = 0;

  virtual bool Exists(const Data&) const noexcept override;

};
```

Estende testable container poiché grazie alle funzioni di fold si può implementare concretamente l'exists:

```
template<typename Data>
void AuxFoldExist(const Data& dat, const void* val, void* exists) {
  if (dat == *((Data*) val)) {
    *((bool*) exists) = true;
  }

}

template<typename Data>
bool FoldableContainer<Data>::Exists(const Data& dat) const noexcept {
  bool exists = false;
  FoldPreOrder(&AuxFoldExist<Data>, &dat, &exists);
  return exists;
}
```

Le fold differiscono dalle map per il fatto che queste non possono modificare la struttura e inoltre, hanno un accumulatore.

15.6 InOrder e Breadth Container

Rappresentano dei tipi di contenitori non lineari (come alberi e grafi), navigabili sia in profondità che in ampiezza, di seguito si riportano le classi per la navigazione InOrder e Breadth:

```
template <typename Data>
class InOrderMappableContainer: virtual public MappableContainer<Data> {
/* ... */
  using typename MappableContainer<Data>::MapFunctor;
  virtual void MapInOrder(const MapFunctor, void*) = 0;
};

template <typename Data>
class InOrderFoldableContainer: virtual public FoldableContainer<Data> {
/* ... */
  using typename FoldableContainer<Data>::FoldFunctor;
  virtual void FoldInOrder(const FoldFunctor, const void*, void*) const = 0;
};

template <typename Data>
class BreadthMappableContainer: virtual public MappableContainer<Data> { /
/* ... */
  using typename MappableContainer<Data>::MapFunctor;
  virtual void MapBreadth(const MapFunctor, void*) = 0;
};

template <typename Data>
class BreadthFoldableContainer: virtual public FoldableContainer<Data> {
/* ... */
  using typename FoldableContainer<Data>::FoldFunctor;
  virtual void FoldBreadth(const FoldFunctor, const void*, void*) const = 0;
};
```

Strutture Dati Elementari

16. Vettori e Liste

16.1 Vector

I vettori sono un insieme di variabili, tutte dello stesso tipo, identificate da un numero intero (indice). La classe Vector eredita da LinearConteiner, MappableContainer e FoldableContainer, di conseguenza sono contenitori lineari in cui è possibile navigare la struttura sia con operazioni di lettura che di scrittura.

16.2 List

Una lista è una successione finita di valori di uno stesso tipo. Come per il Vector eredita dalle stesse classi e quindi si possono svolgere operazioni di inserimento, cancellazione, visita, ricerca e inizializzazione. A differenza del vettore gli elementi di una lista sono rappresentati da nodi collegati tra loro. Un nodo ha in sé, oltre al dato un puntatore al nodo successivo (lista linkata) e, a volte, un puntatore al nodo precedente (lista doppiamente linkata).

Sulle liste linkate risulta particolarmente utile e semplice la ricorsione. Questo è dovuto al fatto che posso definire una lista come: "*una lista è: la lista vuota, oppure un elemento seguito da un'altra lista"*. Quindi ogni nodo è composto dal dato e dal resto della lista (se si ha solo il puntatore next).

17. Pile e Code

17.1 Stack

Uno stack è una collezione dinamica di elementi in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato, in uno Stack questo elemento è l'ultimo elemento inserito. Inoltre, uno stack può essere visto come una lista di tipo LIFO (last in, first out), ovvero i nuovi elementi vengono inseriti in testa e prelevati dalla testa. Di conseguenza avremo due operazioni di modifica: Push (aggiunge un elemento in cima allo stack) e Pop (rimuove un elemento dalla cima dello stack).

Vedremo due possibili implementazioni per un stack: vettoriale o lista linkata.

Le strutture stack devono poter accedere allo stack solo con le funzionalità dello stack, di conseguenza anche se eredita da lista o da vettore esternamente non deve essere visibile, quindi l'estensione sarà in modo protetto: class stack: ***protected list { };***. È possibile chiedere però il tipo di concretizzazione, se con rappresentazione vettoriale o di lista poiché possono essere più o meno efficienti in base all'utilizzo, ma l'utente non può avere la possibilità di sapere cosa chiamiamo internamente alle funzioni.

Lo stack non reimplementa i metodi della classe da cui eredita, ma viene implementata richiamando i metodi di quest'ultima, quindi non si ha overloading.

- **ListStack**: essendo dinamica basta usare le funzioni remove e front
- **VectorStack**: Lo Stack di vettore dovrebbe, ad ogni inserimento, fare ogni volta una ridimensione della size del vettore, di conseguenza per evitare ogni volta una chiamata di sistema si crea un'allocazione di memoria abbastanza ampia così da dover creare una nuova allocazione solo se la precedente allocazione viene saturata (si consiglia di raddoppiare ogni volta l'attuale dimensione). L'aumento di dimensione può avvenire sia se non ho più spazio a disposizione e devo inserire il valore, oppure se inserito il valore ho riempito la memoria (non c'è nessuna differenza tra i due, è a scelta del programmatore).

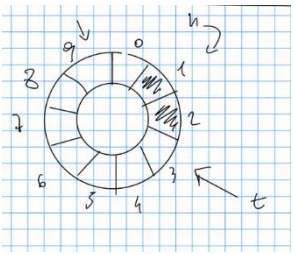
Ha senso, inoltre fare una resize del vettore solo se la dimensione del vettore è molto più libera che piena, ma, ovviamente, bisogna sempre lasciare un certo spazio per eventuali futuri inserimenti. Quindi dopo aver deciso di fare il ridimensionamento (dopo ad esempio ¾ di spazio libero se raddoppio la dimensione) bisogna lasciare una discreta parte di memoria così da non dover aumentare subito dopo la dimensione del vettore.
Si consiglia, inoltre, di non partire con un array vuoto al fine di non dover gestire il caso iniziale diversamente dagli altri casi (quindi partire con una size di almeno 1).

17.2 Queue

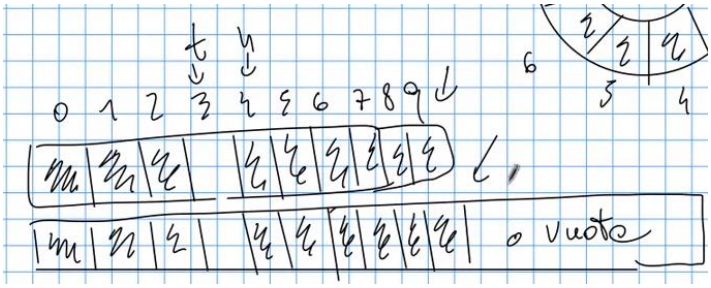
Una coda, come lo stack, è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato (l'elemento che per più tempo è rimasto nell'insieme). Una coda implementa una lista di tipo FIFO (first in, first out).

Anche per la Queue abbiamo un tipo di implementazione tramite lista che è banale ed una implementazione più complessa che è quella tramite rappresentazione vettoriale.

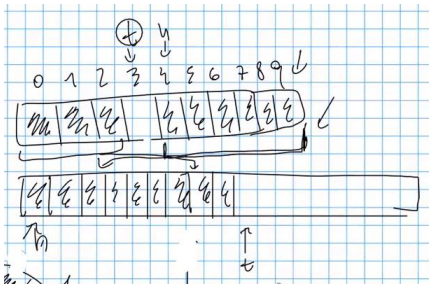
Nella queue il vettore non verrà utilizzato in maniera lineare (come nello stack) ma in modo circolare, l'idea è quella di avere un vettore che rappresenta i dati come se fosse chiuso su se stesso e questo necessita l'utilizzo di due indici: testa e coda. Quindi con la enqueue sposto l'indice di coda mentre con la dequeue è la testa a dover avanzare. È utile una locazione sentinella (terzo indice) da lasciare sempre libera così da poter gestire più semplicemente la saturazione del vettore.



Questo tipo di gestione implica un modo diverso di espandere o ridurre il vettore, infatti la stessa resize del vettore che è possibile usare nello stack non funziona bene poiché si avrebbe una situazione del genere:

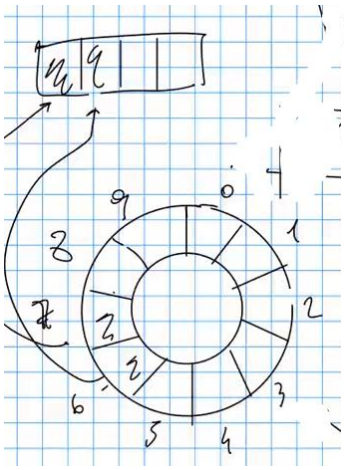


invece di



Quindi è necessaria una funzionalità opportuna che faccia una resize come mostrato nella figura sopra ⇑.

Ovviamente la riduzione non presenta questo tipo di problemi poiché si innesca quando c'è una parte di memoria occupata ed il resto è libero, quindi semplicemente:



Alberi Binari di Ricerca & Iteratori nei Dati

18. Alberi Binari

18.1 Caratteristiche

Un albero binario è una insieme dinamico che è vuoto oppure è composto da tre insiemi disgiunti di nodi:

- Un insieme di cardinalità uno, detto **nodo radice**
- Due sottoalberi: **sottoalbero sinistro** e **sottoalbero destro**.

Per gestire un albero la prima cosa che si deve avere è l'accesso alla Radice, inoltre, ogni albero è formato da nodi connessi tramite un collegamento ad un nodo figlio sinistro e/o destro. Si definisce nodo **foglia** un nodo dell'albero che non ha archi uscenti.

Un albero binario si dice **completo** se ogni livello, tranne eventualmente l'ultimo, è completamente pieno e tutti i nodi sono il più sinistra possibile ($h = \log n$). Quando abbiamo alberi completi sappiamo come poter accedere ai figli in base al solo indice dell'array (figlio sinistro: $2 * \text{curr\_idx} + 1$; figlio destro: $2 * \text{curr\_idx} + 2$) e quindi è evidente una sua rappresentazione vettoriale.

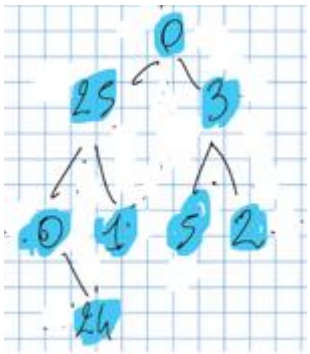
Per alberi non completi, invece, sarebbe difficile, o almeno molto dispensioso in spazio, usare una rappresentazione vettoriale poiché si dovrebbe indicare (tramite un valore accessorio) gli elementi vuoti dell'array (che corrispondono ai nodi inesistenti dell'albero).

Utile è avere la distinzione sinistra e destra tra i figli, quindi il nodo sarà un oggetto contenente sicuramente il Dato, puntatore al nodo left e puntatore a nodo rigth, inoltre è convenzione dare priorità al figlio sinistro e dopo a quello destro (necessario dare una priorità per le visite).

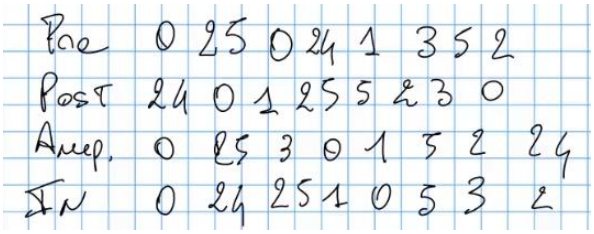
18.2 Visite negli alberi binari

Gli alberi possono essere attraversati (visitati) in diversi modi:

- **Visita in profondità**: ci sono diversi modi per visitare un albero in profondità:
  - **Pre-Order**: prima apro scopro e visito il nodo corrente, poi il sotto albero del figlio sinistro ed infine quelo destro
  - **In-Order**: scopro il nodo, visito il sottolabero del figlio sinistro, poi visito il nodo ed infine il sottoalbero destro. Questa visita è possibile solo per alberi binari poiché si perderebbe la sua proprietà per alberi con arità maggiore di due (ad esempio se ho tre figli, visito il mio nodo prima o dopo il sottoalbero centrale?)
  - **Post-Order**: prima scopro il nodo, visito il sottoalbero sinistro e poi quello destro ed infine il nodo stesso, questa è anche la modalità spesso usata per distruggere l'albero (se distruggo il nodo poi perdo le informazioni dei figli).
- **Visita in ampiezza**: si visita l'albero per livelli, prima tutti i nodi a livello 0, poi quell a livello 1, …, poi quelli a livello h



Ad esempio, dato l'albero a sinistra,
avremo i seguenti tipi di visita:



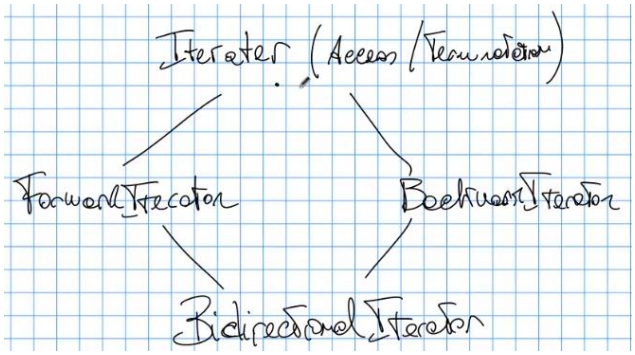
19. Iteratori

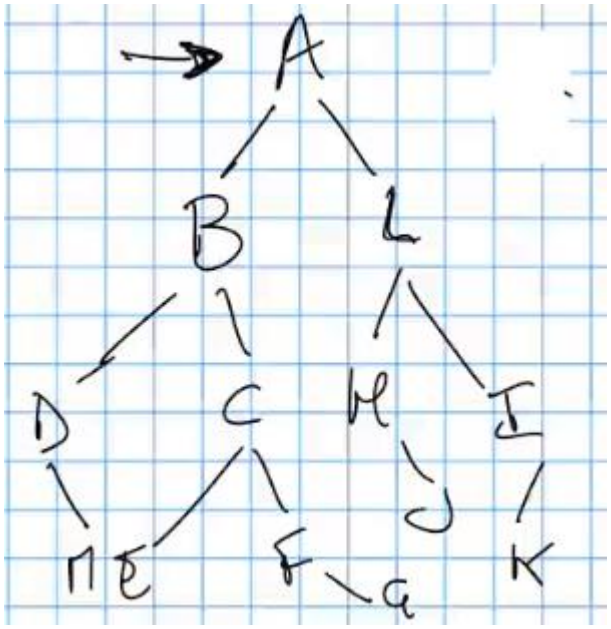
19.1 Operazioni richieste su un oggetto iteratore

Un iteratore è un oggetto che consente di visitare tutti gli elementi contenuti in un altro oggetto, tipicamente un contenitore, senza doversi preoccupare dei dettagli di una specifica implementazione.

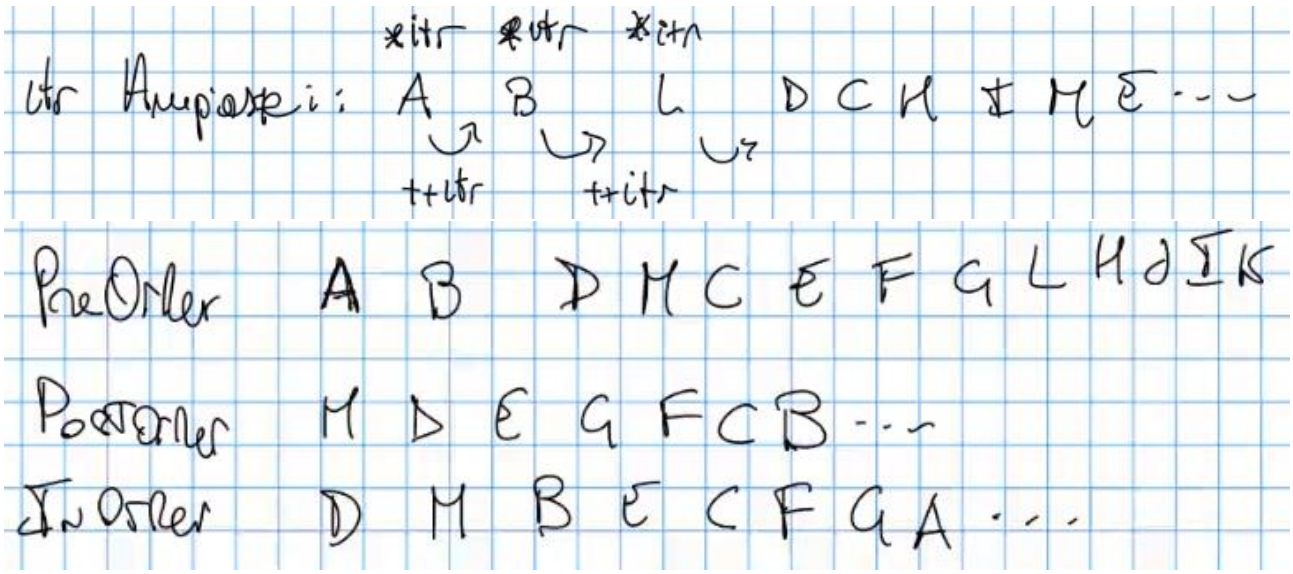
L'iteratore, oltre ad ovviamente distruttore e costruttore specifico (non avrebbe senso un iteratore vuoto), avrà un overleading di vari operatori: operatore di accesso (deferenzazione * e accesso a puntatore ->); un termination test (indica che l'iteratore ha terminato il test) con funzione booleana; ed infine, operatori Successor e precessor (rispetticamente ++, - -). Ovviamente non tutti gli iteratori devono avere tutte le funzionalità, ad esempio il ForwardIterator non avrà l'operatore di predecessore che sarà invece in BackwordIterator.

Vediamo come si comporta un oggetto iteratore su albero binario (implementeremo solo il ForwardIterator).





Sappiamo che una volta avuto accesso alla radice si ha accesso all'intero albero indipendentemente dalla struttura dello stesso, posso, quindi, implementare la ricerca a livello astratto. I metodi della struttura ad albero restituiranno opportuni iteratori (ampiezza, preorder, postorde, inorder) che saranno <u>oggetti</u> (non più metodi) utilizzabili tramite gli operatori ++, - -, *, ->. Vediamo il comportamento degli operatori ++ e *: quest'ultimo ci dà la possibilità di visitare il nodo corrente mentre l'operatore ++ ci fa avanzare al nodo successivo. Avremo dunque, per l'albero in figura, le seguenti visite:



19.2 Implementazione degli iteratori

```
template <typename Data>
class Iterator {

public:

  // Destructor
  virtual ~Iterator() = default;
  // Copy assignment
  Iterator& operator=(const Iterator&) = delete;
  // Move assignment
  Iterator& operator=(Iterator&&) noexcept = delete;
  // Comparison operators
  bool operator==(const Iterator&) const noexcept = delete;
  bool operator!=(const Iterator&) const noexcept = delete;

  // Specific member functions
  virtual Data& operator*() const = 0;
  virtual bool Terminated() const noexcept = 0;

};

template <typename Data>
class ForwardIterator: virtual public Iterator<Data> {

public:

  // ...

  // Specific member functions
  virtual ForwardIterator& operator++() = 0;
};
```

Per un albero avremo i seguenti tipi di iteratori concreti:

- **BTPreOrderIterator**: naviga l'albero in PreOrder
- **BTPostOrderIterator**: naviga l'albero in PostOrder
- **BTInOrderIterator**: naviga l'albero in InOrder
- **BTBreadthIterator**: naviga l'albero in ampiezza

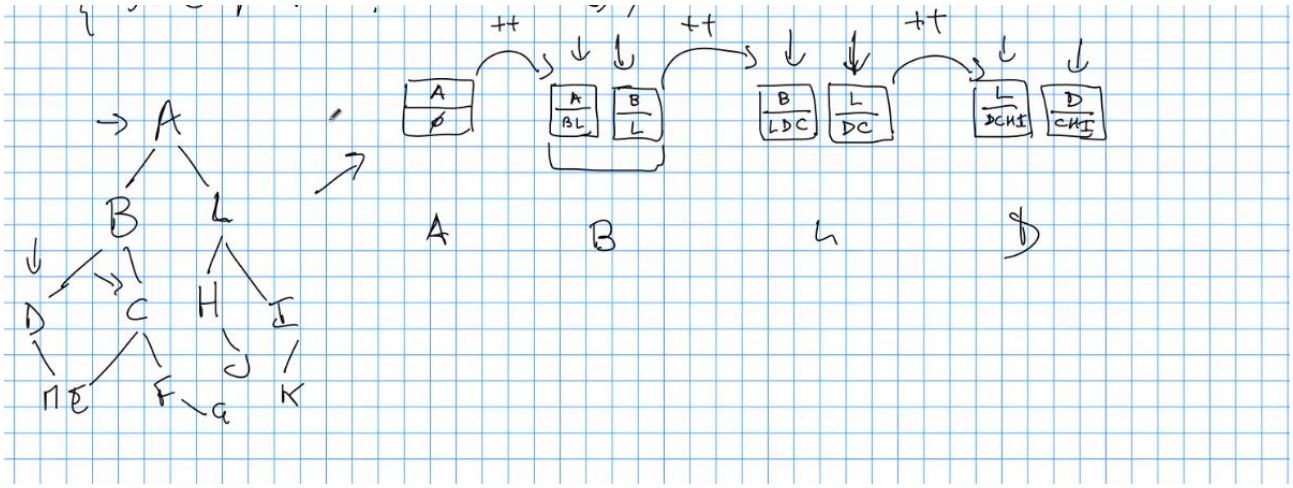
Un oggetto di tipo iteratore su albero deve contenere i seguenti dati:

- **Puntatore a nodo**: punterà al nodo corrente dell'iteratore
- **Struttura dati**: necessaria per salvare i nodi scoperti (allo scopo di arrivare al nodo giusto) ma non ancora visitati. Praticamente si usa una coda per l'iteratore in ampiezza ed uno stack per gli altri tipi di iteratore.

19.3 BTBreadthIterator

Costruttore specifico: dovendo visitare l'albero per livelli il nodo corrente sarà ovviamente la radice, inoltre bisogna mettere in coda i suoi eventuali figli e lo si potrebbe fare o a questo livello oppure nell'operatore ++.

Operator++: deve semplicemente (oltre a mandare errore se l'iteratore è già terminato) controllare se la coda non è vuota, e in quel caso far puntare al nodo corrente il nodo presente in testa alla coda (HeadNDequeue) ed infine encodare gli eventuali figli (prima ovviamente il sinistro, se presente, e poi il destro, se presente); altrimenti (se la coda è vuota) bisogna fare current = nullptr.



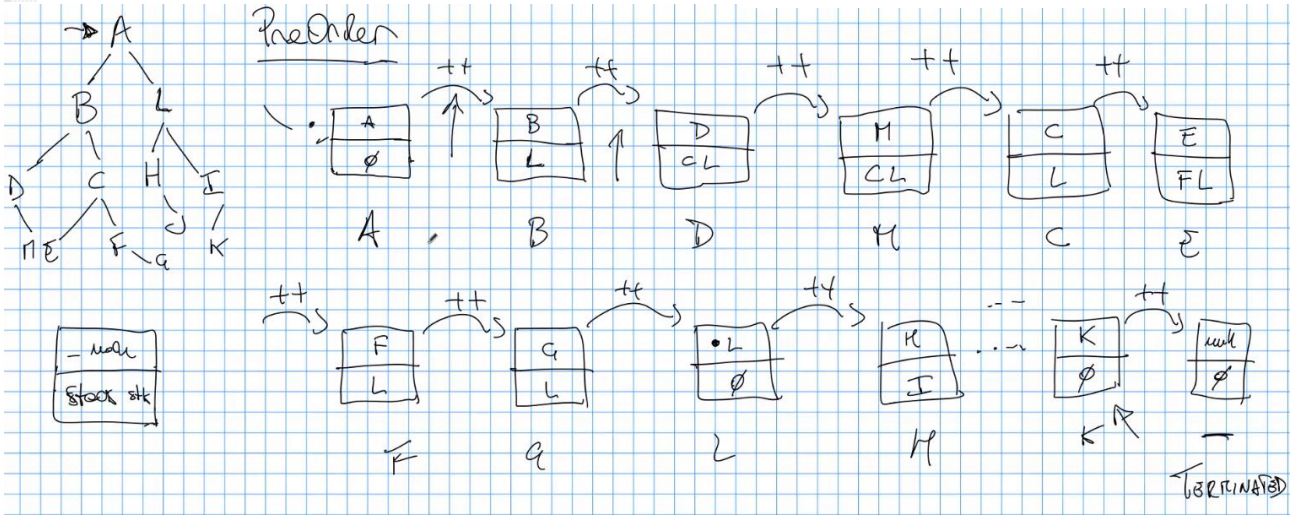
19.4 BTPreOrderIterator

Costruttore specifico: anche qui il nodo corrente punterà alla radice, e questa sarà anche l'unica operazione necessaria per il costruttore poiché, per il tipo di visita, non è necessario riempire attualmente lo stack.

Operator ++: Si procede nel seguente modo

```
if (current->HasLeftChild()) {
    if (current->HasRightChild()) {
        stk.Push(&(current->RightChild()));
    }
    current = &(current->LeftChild());
    return *this;
    }
if (current->HasRightChild()) {
    current = &(current->RightChild());
    return *this;
}
if (!stk.Empty()) {
    current = stk.TopNPop();
    return *this;
}
else {
    current = nullptr;
    return *this;
}
```

Un implementazione differente potrebbe essere per un albero statico implementato tramite vettori, dove è sufficiente avere soltanto un indice del nodo corrente. In questo caso è banale fare un iteratore in ampiezza ed anche un iteratore in backword.



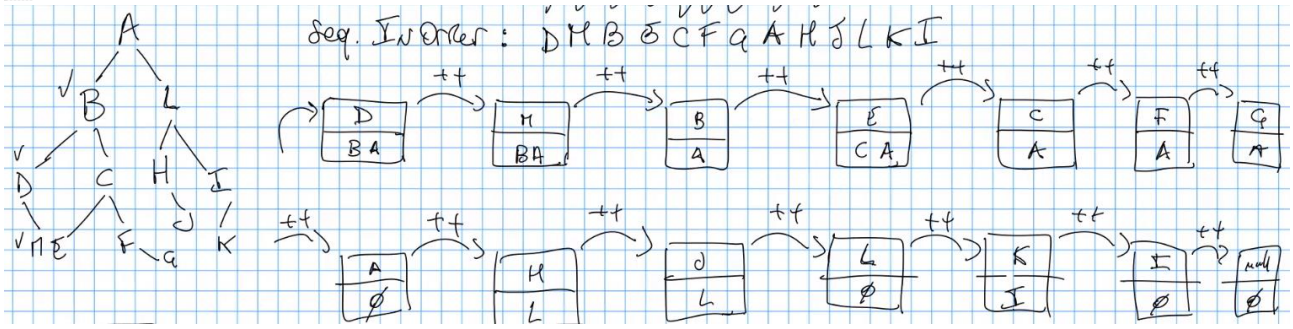
19.5 BTInOrderIterator

Costruttore specifico: A differenza dei precedenti qui il primo nodo da visitare non è la radice, di conseguenza (partendo sempre dalla radice) il nostro nodo corrente sarà il nodo più a sinistra, e durante la navigazione bisognerà anche riempire in maniera opportuna lo stack $\left(\text{LeftmostNode}(root, stk)\right)$.

Operator ++: Si procede nel seguente modo:

```
if (current->HasRightChild()) {
    current = LeftMostNode(&current->RightChild(), stk);
    return *this;
}
else {
    if (stk.Empty()) {
        current = nullptr;
        return *this;
    }
    else {
        current = stk.TopNPop();
        return *this;
    }
}
```

```
LeftMostNode(Node* curr, Stack& stk) {
    while (curr->HasLeftChild()) {
        stk.Push(curr);
        curr = &(curr->LeftChild());
    }
    return curr;
}
```



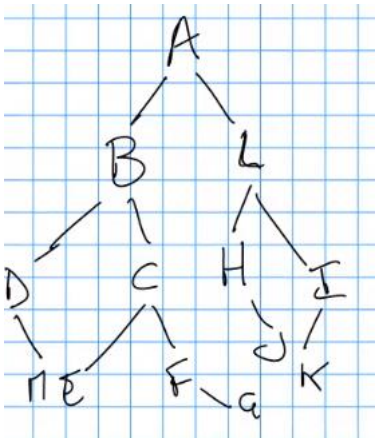
19.5 BTPostOrderIterator

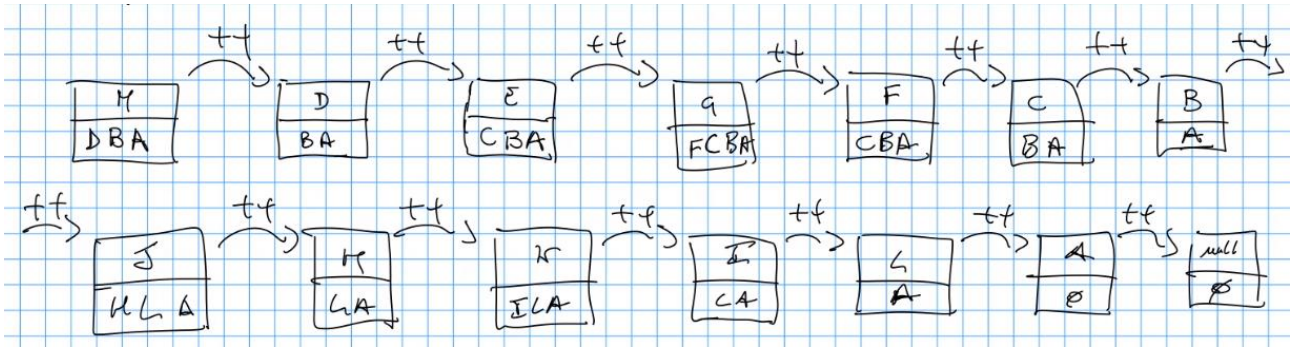
Costruttore specifico: Il nodo corrente punterà alla foglia più a sinistra, e lo stack sarà riempito con i nodi scoperti per arrivare alle foglie.

Operator ++: Si procede nel seguente modo:

```
if (stk.Empty()) {
    current = nullptr;
    return *this;
}
else {
    if ((stk.Top()->HasRightChild()) &&
        (&(stk.Top()->RightChild()) != current))
    {
        current = &(stk.Top()->RightChild());
        current = LeftMostLeaf(current, stk);
        return *this;
    }
    else {
        current = stk.TopNPop();
        return *this;
    }
}
```

```
LeftMostLeaf(Node* curr, Stack& stk) {
    while (!(curr->IsLeaf())) {
        while (curr->HasLeftChild()) {
            stk.Push(curr);
            curr = &(curr->LeftChild());
        }
        if (curr->HasRightChild()) {
            stk.Push(curr);
            curr = &(curr->RightChild());
        }
    }
    return curr;
}
```





20. Binary Search Tree (BST)

20.1 Alberi Binari di Ricerca

Un albero binario di ricerca è un albero binario che soddisfa la seguente proprietà: Se X è un nodo e Y un qualsiasi nodo del sottoalbero sinistro, mentre Z un qualsiasi nodo del sottoalbero destro allora $Y \to dato \leq X \to dato \leq Z \to dato$; inoltre per semplicità non considereremo copie di dati, dunque la nostra proprietà sarà semplicemente $Y \to dato < X \to dato < Z \to dato$.

Questa proprietà ci permette di poter ottimizzare di molto la ricerca dei valori nell'albero, di conseguenza usare il metodo exists implementato fino ad ora sarebbe da veri pagliacci, poiché visitando un nodo che ha dato x so per certo che il mio dato y sarà a destra (se $y > x$) o a sinistra (se $y < x$). Generalmente, la ricerca è confinata ai nodi posizionati lungo un singolo percorso dalla radice ad una foglia, dando quindi un tempo di ricerca pari massimo all'altezza dell'albero.

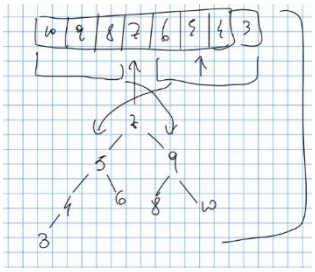
Anche il concetto di visita in un albero binario viene meno, poiché nei binari di ricerca ho un ordine dei nodi che corrispone esattamente all'ordine dei dati, inoltre anche il concetto di uguaglianza è diverso. Infatti, un albero binario è lo stesso se oltre a contenere gli stessi dati, i nodi sono nelle stesse posizioni, mentre, per un albero binario basta che entrambi hanno gli stessi dati e che questi rispettino la proprietà di albero binario.

Un albero binario di ricerca si potrebbe pensare di poterlo implementare solo grazie a BinaryTreeLink ma ciò lo rinderebbe poco efficiente infatti, anche se funzionerebbe, si perderebbero tutte le possibili ottimizzazioni descritte in precedenza, di conseguenza non avrebbe senso.

20.2 Costruzione

1) Costruzione di un albero binario di ricerca dato un linear container:
Potrei scorrermi il vettore ed ad ogni dato inserirlo nell'albero e/o modificarlo, ma ciò potrebbe portare ad un costo quadratico nel caso abbia un albero sbilanciato, in ogni caso questo tipo di implementazione resta la più semplice da fare.

Una migliore implementazione è quella di ordinare il vettore in maniera crescente (ovviamente è necessario un algoritmo di ordinamento con un costo basso, altrimenti conviene il metodo precedente) e prenderne il punto medio che diventerà la nostra radice, così facendo la parte sinistra del vettore sarà il sottoalbero sinistro e la parte destra il sottoalbero destro, questo procedimento in maniera ricorsiva costruirà un albero ben bilanciato (vedi figura a destra).

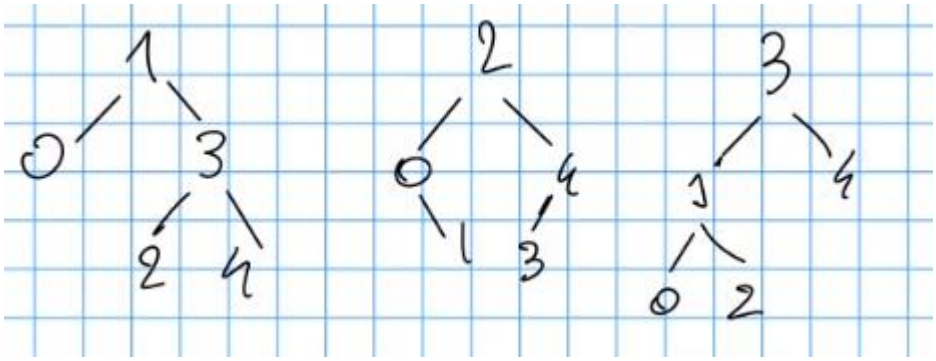


Se si parla di liste l'unica via percorribile è la costruzione tramite inserimento, poiché queste non hanno un metodo efficiente di ordinamento.

2) costruzione e asegnamento per copia e spostamento:
Si possono banalmente usare i cosntructor e gli assignement imlpementati in binarytree, poiché copiando/spostando l'albero si mantengono invariate le sue proprietà.

20.3 Operatori di confronto ed Exists

Non si possono ereditare i comparison operators da BinaryTreeLink poiché esso compara gli alberi strutturalmente ma negli alberi binari di ricerca il concetto di uguaglianza è quello di comparare i dati e non la struttura dell'albero. Ad esempio i seguenti alberi sono diversi per la comparazione binarytreelink ma uguali per l'ABR:



Quindi dopo il confronto della dimensione si può semplicemente usare l'iteratore InOrder (che preserva sia i dati che l'ordine nel nostro caso) per visitare e confrontare i singoli dati.

Per l'operatore di uguaglianza, funzionerebbe anche quello implementato da binarytree ma così facendo non si sfrutterebbero le proprietà dell'albero binario di ricerca va dunque reimplementato. Una implementazione concreta e semplice da implementare è quella di costruire due iteratori (uno per albero) e con un cilco for che incrementa entrambi gli iteratori ad ogni passo confrontare i loro dati.

20.4 Insert e Remove

Nell'implementazione dell'albero binario si nota l'importanza di modularizzare il codice, infatti nell' inserimento è in altre funzionalità si usa la stessa parte di ricerca che viene implementata nella exists. Di conseguenza è utile una funzione che ritorna il riferimento al puntatore a nodo chiamata FindPoinerTo(dato) che cerca il puntatore al nodo in cui è presente il dato che stiamo cercando.

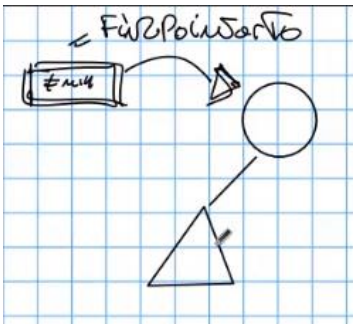
Di conseguenza nell'inserimento vado semplicemente a creare un nuovo nodo e a collegarlo grazie all'indirizzo trovato dalla FindPoinerTo. Questa funzione ausiliaria è utilissima non tanto in ricerca, ma in tutte le operazioni di modifica nella struttura, poiché semplifica di molto la programmazione (oltre ad una miglior modularizzazione del programma). La FindPointerTo potrebbe essere implementata nel seguente modo:

```
Node* const& FindPointerTo(const Data& dat, Node* const& key) const noexcept {
    Node* const*ptr = &key;
    if (*ptr != nullptr) {
        if (key->info == dat) {
            return *ptr;
        }
        else if (key->info > dat) {
            ptr = &(FindPointerTo(dat, key->left));
        }
        else /* key->info < dat */ {
            ptr = &(FindPointerTo(dat, key->right));
        }
    }
    return *ptr;
}
```

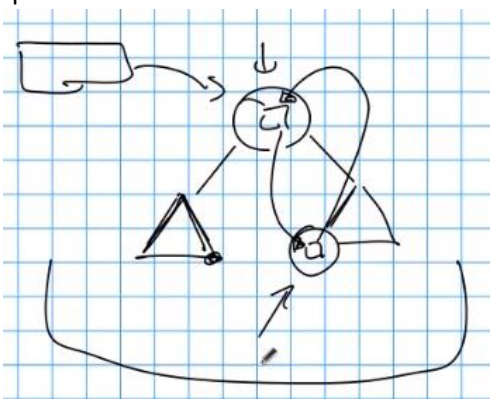
Più complessa invece è la remove, poiché nella sua implementazione bisogna portare particolare attenzione agli eventuali figli del nodo da cancellare.

Di conseguenza, sempre utilizzando la FindPointerTo, ho quattro possibili casi:

1. Non è presente il dato da rimuovere, quindi ovviamente non dovrò cancellare nessun nodo.
2. Il nodo da rimuovere ha solo sottoalbero sinistro:
    - una possibilità è quella di spostare il puntatore in modo tale da farlo puntare alla radice del sottoalbero sinistro e cancellare il nodo ricordando di staccare prima il riferimento dal nodo al figlio per evitare di cancellare tutto il sottoalbero.



3. Il nodo da rimuovere ha solo sottoalbero destro:
    - Si procede in maniera analoga al caso 2.
4. Il nodo da rimuovere ha entrambi i sottoalberi:
    - Si sostiuisce il dato con un nodo foglia così da dargli il dato del minimo nel sottoalbero destro (trovato tramite una funzione di TouchMin) e cancellare quest'ultimo:



Va bene anche scambiarlo con il massimo del sottoalbero sinistro (praticamente basta mantenere intatta la proprietà dell'albero binario).

20.5 Ricerca di massimo e minimo

Grazie alla proprietà del BST, la ricerca del minimo o massimo di un nodo sarà banalmente il nodo più a sinistra e il nodo più a destra rispettivamente. Di seguito una possibile implementazione:

```
Node*& FindPointerToMin(Node*& key) {
    Node* *ptr = &key;
    while ((*ptr)->left != nullptr) {
        ptr = &((*ptr)->left);
    }
    return *ptr;
}
```

```
Node*& FindPointerToMax(Node*& key) {
    Node* *ptr = &key;
    while ((*ptr)->right != nullptr) {
        ptr = &((*ptr)->right);
    }
    return *ptr;
}
```

20.6 Predecessor e Successor (lettura/rimozione)

L'accesso al dato deve essere di sola lettura, poiché dare la possibilità all'utente di modificare il dato implicherebbe dare la possibilità di distruggere le proprietà della struttura, infatti se si modificasse il valore del dato in maniera scorretta si perderebbe la proprietà dell'albero binario di ricerca. Di conseguenza, il predecessore avrà il seguente prototipo: $const\ Data\&\ \mathrm{Predecessor}(data)$, allo stesso modo il prototipo del successore sarà $const\ Data\&\ \mathrm{Successor}(data)$. Prima di dare un esempio di codice, andiamo a definire il predecessore ed il successore:

- Il predecessore di un nodo X è il più grande nodo minore del nodo X (il massimo dei minori di X)
- Il successore di un nodo X è il più piccolo nodo maggiore del nodo X (il minimo dei maggiori di X)

Essendo le due funzionalità complementari si riporta l'implementazione solo della ricerca del successore. Per un nodo X si hanno le seguenti casistiche:

- Se X ha un figlio destro, il successore è il minimo nodo di quel sottoalbero
- Se X non ha un figlio destro, e X è figlio sinistro del padre, il successore è il padre.
- Se X non ha un figlio destro, ed è figlio destro del padre, il successore è l'ultimo antenato per il quale X si trova nel suo sottoalbero sinistro.

Di conseguenza per la ricerca del successore è necessaria un puntatore inizializzato a null che mantenga l'indirizzo del potenziale candidato al successore, poi partendo dalla radice dell'albero:

- ogni volta che si segue il ramo sinistro per raggiungere il nodo, si aggiorna il successore al nodo padre;
- ogni volta che si segue un ramo destro per raggiungere il nodo, non si aggiorna il successore al nodo padre;

```
Node*& FindPointerToSuccessor(const Data& dat) {
    Node* *current = &root;
    Node* *estimate = nullptr;
    // aggiornamento standard della stima
    while (*current != nullptr && (*current)->info != dat) {
        if ((*current)->info < dat) {
            current = &((*current)->right);
        }
        else /* (*current)->info > dat */ {
            estimate = current;
            current = &((*current)->left);
        }
    }
    // aggiornamento della stima se trovo il dato ed ha un figlio destro
    if (*current != nullptr && (*current)->HasRightChild()) {
        estimate = &(FindPointerToMin((*current)->right));
    }
    return *estimate;
}
```

Il predecessore avrà esattamente lo stesso codice ma va invertita la sinistra con la destra:

- ogni volta che si segue il ramo destro per raggiungere il nodo, si aggiorna il predecessore al nodo padre;
- ogni volta che si segue un ramo sinistro per raggiungere il nodo, non si aggiorna il predecessore al nodo padre;
- Se trovo il dato in un nodo, e quel nodo ha figlio sinistro, il predecessore è il massimo del sottoalbero sinistro

Matrici & Grafi

21. Matrici

21.1 Tipi di rappresentazioni

Le matrici vengono suddivise principalmente in matrici dense e matrici sparse, ognuna delle quali ha diversi tipi di rappresentazioni concrete, di seguito ne descriveremo alcune:

- Serializzazione vettoriale
- Rappresentazione multivettoriale
- Liste ortogonali
- Rappresentazione di tipo Yale, conosciuta anche come compressed row/column rappresentation.
- Rappresentazione Coordinate List

Nell'analizzare le varie rappresentazioni considereremo la seguente matrice sparsa 8×6:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} A & B & - & - & - & E \\ C & - & - & - & - & - \\ - & D & - & - & - & - \\ - & - & - & F & G & - \\ - & - & - & - & - & - \\ - & H & - & - & - & I \\ - & - & - & - & - & - \\ L & - & M & - & - & - \end{pmatrix} \end{array}$$

21.2 Serializzazione vettoriale

Si crea un unico array con dimensione il prodotto tra righe e colonne (nel nostro caso $\text{size} = 48$), questo vettore conterrà tutti i dati (comprese le celle vuote, che saranno rappresentate con un determinato carattere), se si prediligono le righe (e quindi il vettore sarà ordinato con la prima riga, poi la seconta, etc…) avrò una rappresentazione in row-major order ($|A\ B - - - E||C - - - - - -|\ |- D - - - -|$ …) mentre se si prediligono le colonne avrò una rappresentazione in colum-major order ($|A\ C - - - - - L\ |\ |\ B - D - - H - - |$ …).

Il vantaggio di questa rappresentazione è la semplicità con cui si accede ai dati, infatti, essendo un semplice vettore, l'accesso all'elemento in posizione $(i\ , j)$ si farà semplicemente accedendo all'indice giusto del vettore, che dipenderà ovviamente dal tipo di rappresentazione che ho usato: per la row-major avrò $i \times column + j$ mentre per la colum-major l'indice sarà $j \times row + i$.

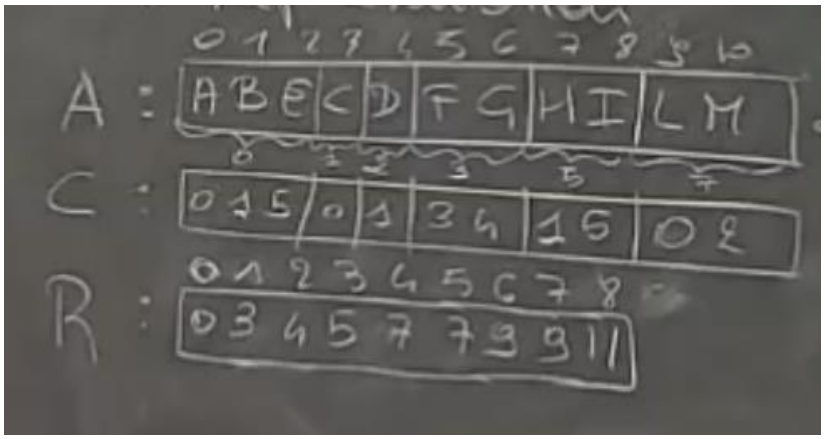
Anche modificare un cella già esistente è banale, però questo tipo di rappresentazione ha un costo enorme se si deve ridimensionare il numero di righe e/o di colonne; infatti per cambiare la dimensione della matrice si deve praticamente copiare tutto il vettore in un nuovo vettore con la nuova dimensione.

21.2 Formato Yale con soli vettori

La Yale representation, conosciuta anche come compressed sparse row/columm rappresentation, rispettivamente CSR e CSC ha diversi vantaggi per le matrici sparse, ci soffermeremo sulla row order ma il ragionamento è del tutto analogo anche per la column order.

Avremo 3 vettori, un vettore A contentente tutti e solo i dati mentre il vettore C e R conterranno rispettivamente gli indici di righe e colonne occupate dal dato. Dunque, tenendo presenta la matrice rappresentata nel paragrafo 21.1 il vettore $A = \{A, B, E, C, D, F, G, H, I, L, M\}$ ed avrà 11 locazioni (size = 11), poi abbiamo un altro vettore della stessa dimensione di A, che rappresenta gli indici colonna dei dati ovvero

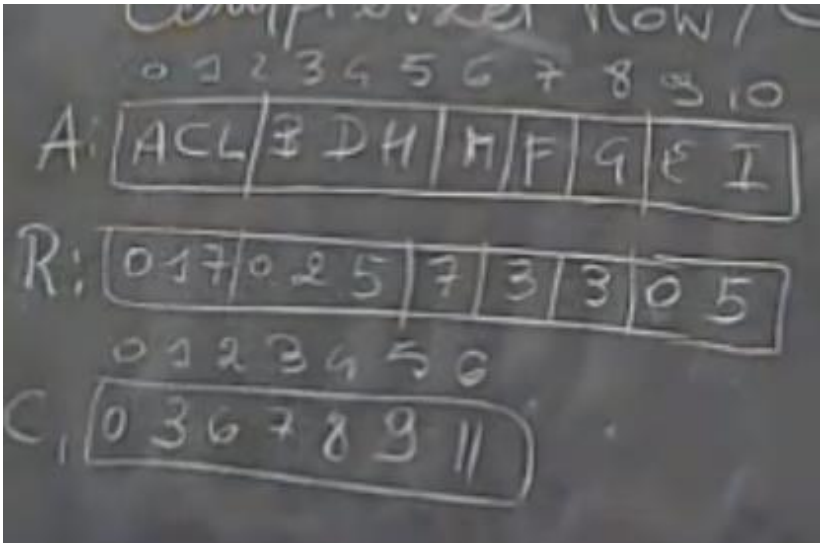
$C = \{0,1,5,0,1,3,4,1,5,0,2\}$, mentre il vettore righe avrà lunghezza pari al numero di righe + 1 (in questo caso 9) dunque se i dati rappresentati sono in numero maggiore del numero di righe questa rappresentazione costa addirittura meno della rappresentazione in liste ortogonali (ultimo paragrafo), l'indice del vettore riga indica appunto la riga e conterrà l'indice del vettore A in cui è presente il primo dato di quella riga, mentre l'ultimo elemento di R conterrà il valore della size di A; avremo dunque $R = \{0, 3,4,5, \_,7, \_,9,11\}$ (se per quella riga non ho alcun dato, allora la cella conterrà il valore della cella successiva):



$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} A & B & - & - & - & E \\ C & - & - & - & - & - \\ - & D & - & - & - & - \\ - & - & - & F & G & - \\ - & - & - & - & - & - \\ - & H & - & - & - & I \\ - & - & - & - & - & - \\ L & - & M & - & - & - \end{pmatrix} \end{array}$$

Il numero dei dati che contiene ogni riga sarà pari alla differenza tra la riga successiva e la stessa, ad esempio la riga 5 avrà $(9 - 7) = 2$ dati, la riga 0 avrà $(3 - 0) = 3$ dati e così via.

La rappresentazione colum major è praticamente identica ma avrò ovviamente il ruolo di C e R invertito:



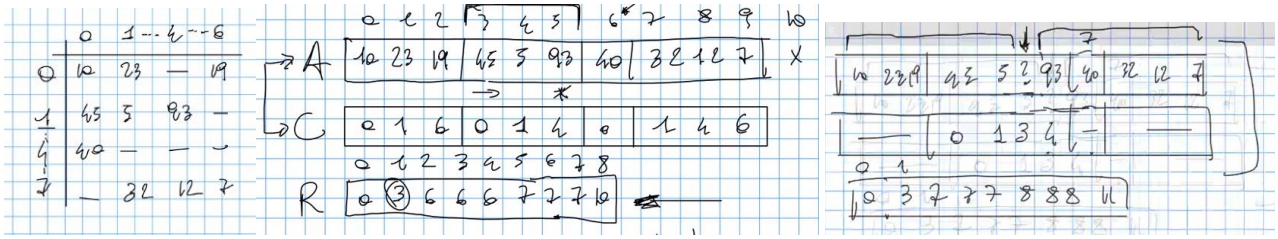
L'accesso ai dati in questa rappresentazione è più costoso ma compensa nella ridimensione del vettore.

- **Accesso ai dati**: supponiamo di voler accedere a D quindi riga 2 e colonna 1, nella serializzazione di dati rom-major order dovremmo identificare la posizione 4 del vettore A; accediamo nel vettore R alla riga 2 (nostro indice) quindi avrà valore 4 che è proprio il nostro dato, infatti accedendo alla posizione 4 delle colonne avrò il valore 1 che è proprio l'indice di colonna (per assicurarne la correttezza). Supponiamo ora di voler trovare il dato G in posizione (3,4), nella posizione 3 della riga avrò il valore 5, quindi vado in posizione 5 del vettore A, il rispettivo nel vettore C è 3, quindi controllo il successivo che è proprio G. Il numero di iterazioni è pari alla differenza spiegata precedentemente, in questo caso due poiché (7-5) = 2, di conseguenza se non trovo il dato dopo questo numero di iterazioni allora non esiste e lancio un'eccezione. Fare il ciclo for è semplice poiché potrei fare nel caso del punto G: for (ulong index = 5, index < 7, ++index)

Accesso ai dati in colum-major order: Supponiamo di voler cercare il dato (2,1) quindi nella colonna 1 ho il valore 3 ed andrò a cercare il vettore 3 nella posizione A, la riga in 3 è zero quindi mi sposto e trovo il macth e mi fermo con il dato D.
Si può ottimizzare la ricerca andando a fare una ricerca binaria nel vettore di colonna in raw-major (ricerca binaria nel vettore di riga in colum major).

- **Ridimensionamento del vettore**: Il ridimensionamento delle colonne non richiede operazione alcuna se non cambiare l'indice totale delle colonne. Se si deve ridurre il numero di righe l'unica cosa da fare è ridimenisonare il vettore righe con dimensione $newrow + 1$, e poi ridimensionare i vettori A e C con la dimensione contenuta in $R[newrow + 1]$. L'estensione del numero di righe è semplicemente una resize del solo vettore R.

Più complessa è l'aggiunta di un dato in una cella non presente, infatti si dovrebbe ridimensionare il vettore A e C, copiare i vecchi dati nella giusta posizione e poi aggiungere i nuovi:



21.3 Rappresentazione multivettoriale

Classica rappresentazione matriciale in memoria è la rappresentazione multivettoriale, è possibile, inoltre, una rappresentazione con vettori nidificati, nestervector, o con una rappresentazione raw/column major order di tipo $\text{vector}\langle vector\langle data\rangle\rangle$. Bisogna stare attenti alla consistenza dei dati in modo tale che i vettori contenuti all'interno dell'altro vettore abbiano tutti la stessa dimensione, altrimenti i dati non rappresenterebbero una vera e propria matrice.

Questa rappresentazione ha il vantaggio rispetto la precedente di poter sfruttare il ridimensionamento di vector per ridimensionare appunto al size della matrice.

21.4 Coordinate List

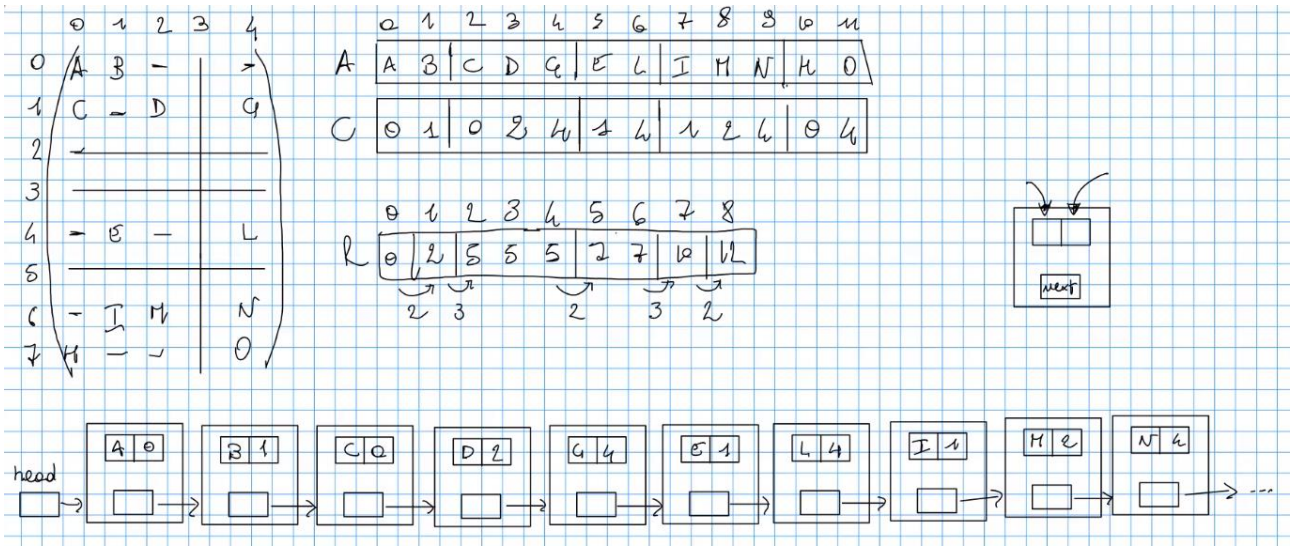
Anche in questa rappresentazione ci sono i due diversi approcci row/column major order in cui il primo dato è formato dalle triple (c_1^1, c_2^1, d^1) (coordinata 1, coordinata 2, dato); $(c_1^2, c_2^2, d^2), \dots, (c_1^n, c_2^n, d^n)$ (di solito sono liste ordinate in modo lessicogarfico, come prima tripla si preferisce quella con prima coordinata più bassa e a parità di coordinata c_1 si preferisce avere coordinata c_2 più bassa) e a seconda che c_1 sia raw o column abbiamo rispettivamente l'ordinamento raw-major o column major.

Questa rappresentazione ha il vantaggio di rappresentare solo i dati effettivi ma lo svantaggio di avere una ricerca lineare degli elementi, altrimenti con opportuni alberi si potrebbe avere una struttura dati più efficiente in ricerca ma al costo di una infrastruttura molto più pesante e spesso quindi, si preferisce avere una struttura più semplice a costo di spendere più tempo nella ricerca.

21.5 Yale CSR con lista

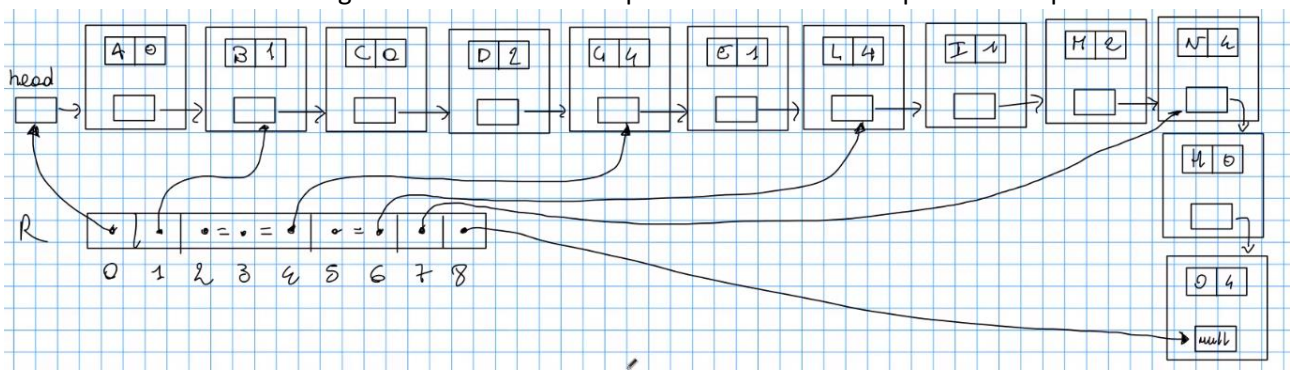
Per risolvere il problema visto nella rappresentazione Yale con soli vettori di aggiungere un dato all'interno della matrice bastra semplicemente cambiare i vettori A e C con una lista di coppie (o due liste diverse). Ovviamente avremo una modifica estremamente più rapida a costo di perdere però la possibilità di implementare la ricerca binaria dei dati (si può solo fare la ricerca lineare). I vettori saranno inglobati in una lista di coppie (o due liste diverse).

Si consideri la seuente rappresentazione CSR (tramite liste):



Nella rappresentazione vettoriale abbiamo già visto la complessità nell'aggiunta di un nuovo dato non già presente.

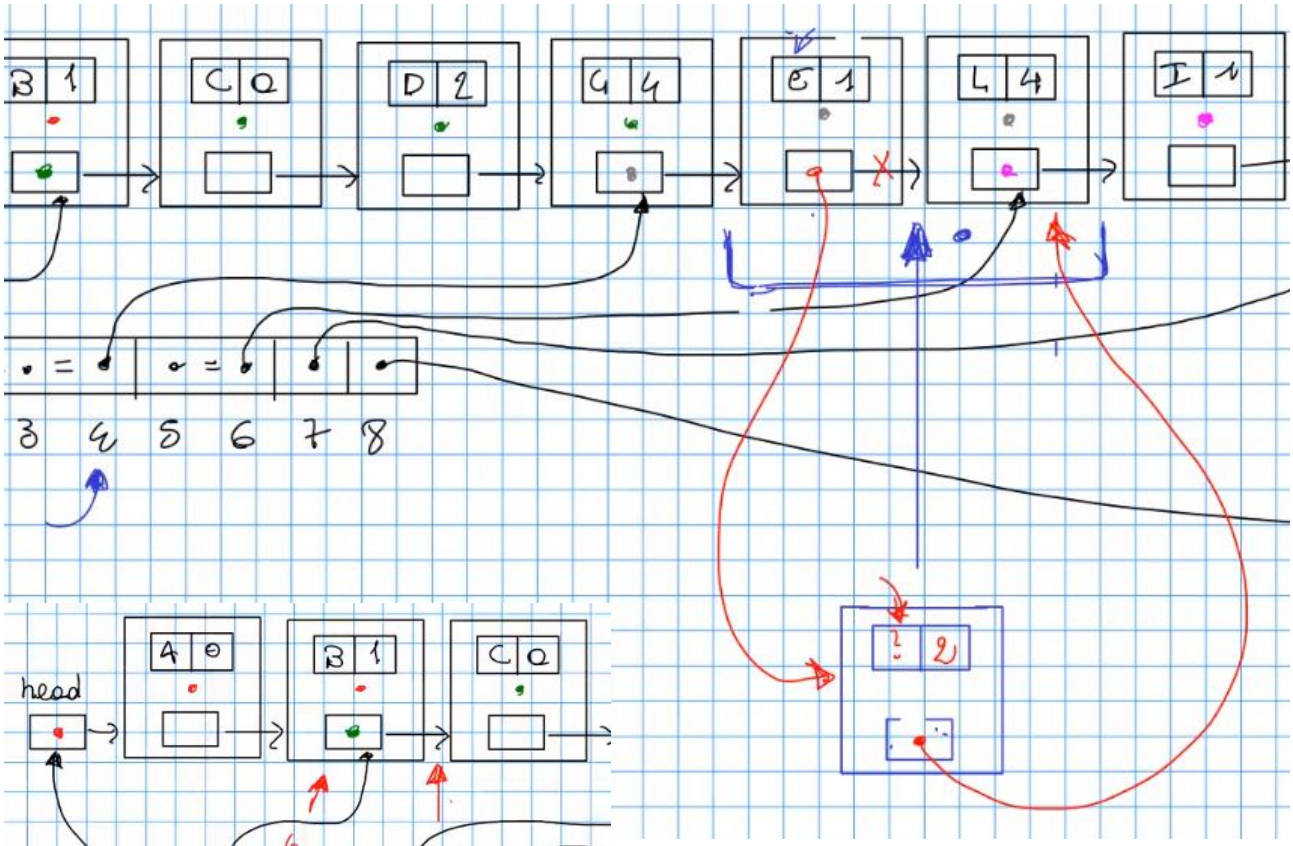
Nella rappresentazione di lista ho ogni nodo con un campo next ed una pair rappresentante la coppia $(dato, colonna)$. Le righe saranno rappresentate da un vettore della stessa lunghezza del vettore R indicizzato con i numeri di riga ma non conterranno più indici numeri bensì puntatori di puntatore a nodo:

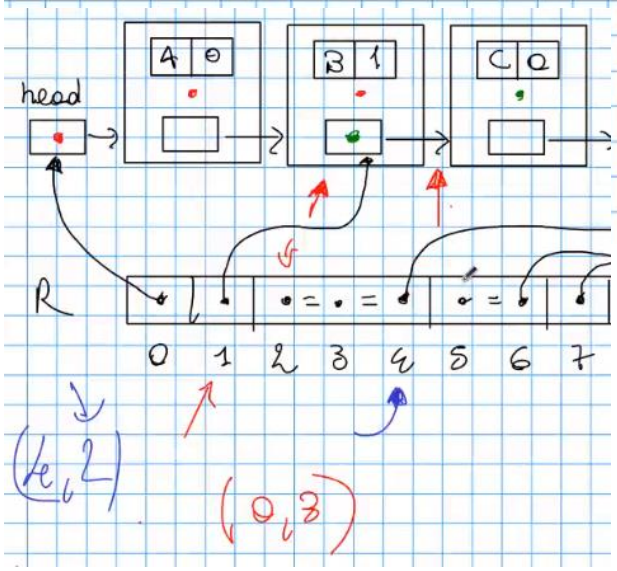


Ogni elemento del vettore è un puntatore al puntatore al nodo (non è un puntatore diretto al nodo). Quindi 0 punterà sempre alla testa (che punta al primo nodo della lista con il primo elemento della matrice), ora 1 punta al puntatore che punta alla prima cella della seconda riga (ogni elemento di R è l'indirizzo o di head o di next del nodo che precede il nodo in cui inizia la riga corrispondente); 2 e 3 avranno lo stesso indirizzo della riga 4 che punta al next (che punta ad E, quindi alla riga di indice 4). Il 5 avrà lo stesso indirizzo di 6. Infine la riga sette inizia con il dato H e di conseguenza punta al next del nodo precedente. L'indice 8 punta al next dell'ultimo nodo (ovvero all'indirizzo che contiene next = nullptr), questo è utile per operazioni in casi limite.

Nel caso si volesse inserire un elemento nella cella (4,2) bisogna innanzitutto accedere al dato, quindi si deferenzia 4 e si accede al next che deferenziato punta ad E; non essendo questo il nostro dato procediamo in avanti spostando il nsotro puntatore. Il dato successivo è effettivamente la colonna del dato che abbiamo noi ma non c'è un match della colonna, a questo punto, non avendolo trovato prima e appena ci si sposta siamo andati oltre 1<2<4. Allora mi fermo poiché so che il dato deve trovarsi in mezzo, di conseguenza si procede con l'**inserimento**:

Si va a creare un nuovo nodo da inserire tra E e L, quindi si aggiungono i valori dei dati (si può aggiungere immediatamente il valore della colonna non trovata e il puntatore next al nodo successore) mentre si stacca il precedente e lo si fa puntare al nodo appena aggiunto:

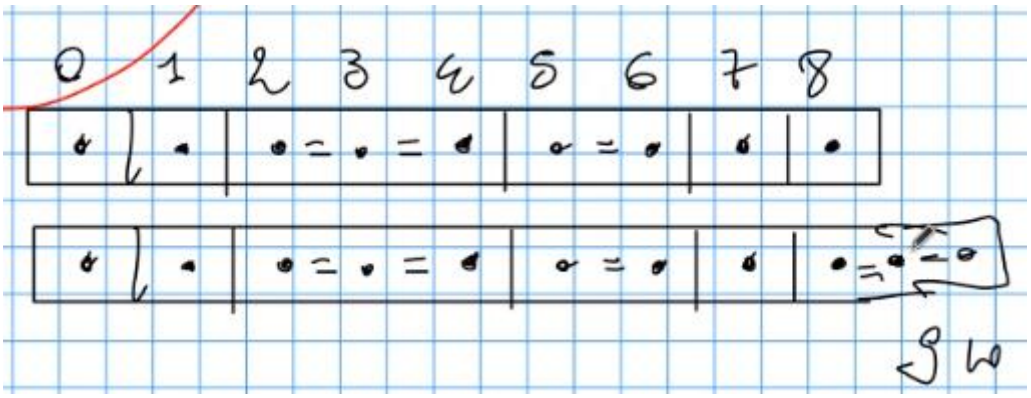




← Ci si può fermare anche se il puntatore che abbiamo è uguale alla riga sucessiva, ad esempio se volessimo inserire la cella (0,3), si parte con il puntatore che punta alla head che ci permette di accedere ad A; 0 è minore quindi ci spostiamo al nodo successivo che continene 1 che è minore di 3, spostiamoci quindi al puntatore di B, ma il nodo successivo non fa parte di quella riga poiché punto alla stessa cella puntata dalla riga 1.

Per estendere il numero di righe bisogna ridimensionare il vettore riga aumentandolo delle righe necessarie copiando i vecchi valori ed aggiungendo alle celle aggiunte lo stesso indirizzo di 8.

Ad esempio se volessi agigungerela riga 9 e 10 avrò:

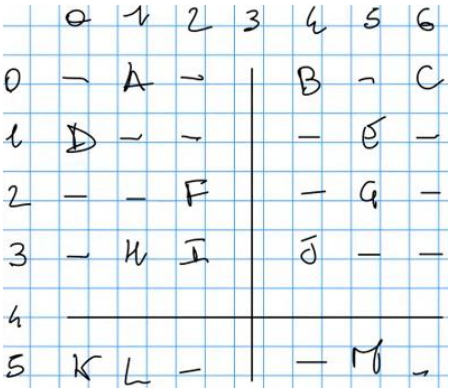


Per la riduzione di righe, invece, bisogna <u>prima</u> accedere ai nodi e cancellarli e poi ridimensionare il vettore.

Per aggiungere colonne non bisogna ovviamente fare nulla, mentre per eliminarle bisogna effettuare le opportune delete (facendo attenzione ovviamente anche al vettore riga).

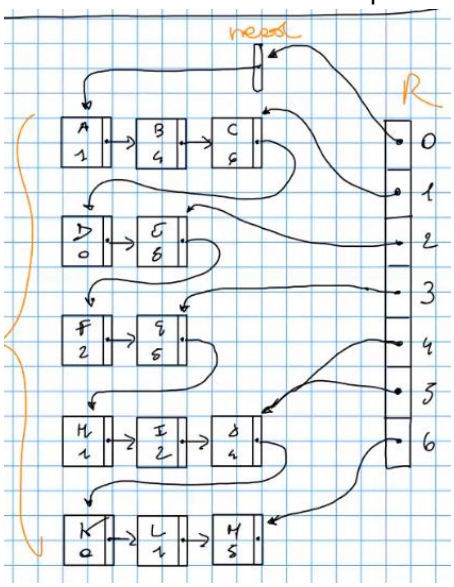
21.6 Rappresentazione liste ortogonali

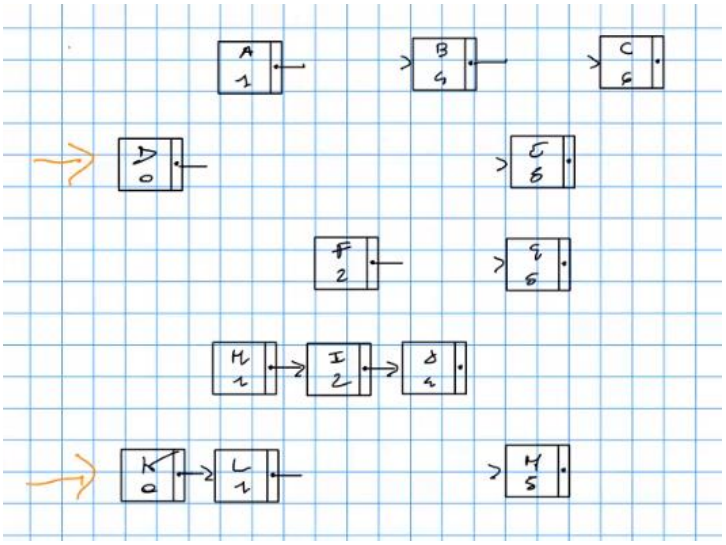
La rappresentazione in liste ortogonali è un tipo di rappresentazione che non predilige righe o colonne ed entrambe hanno lo stesso costo.



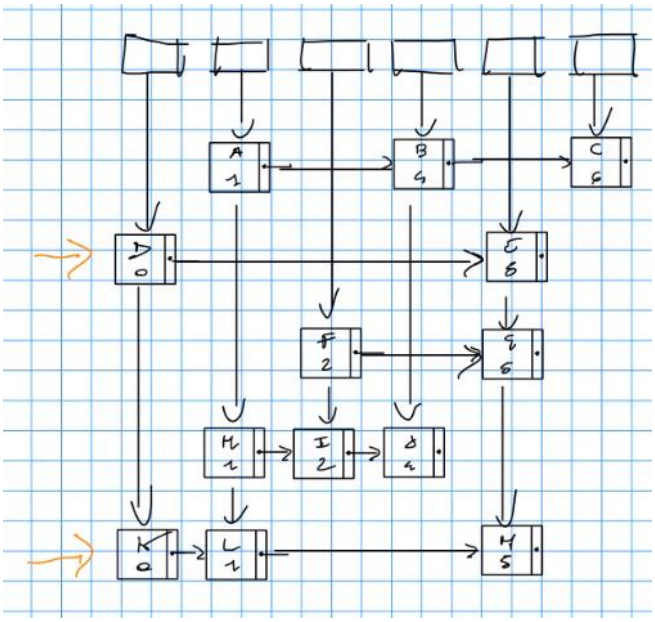
Prendiamo in considerazione la matrice a destra ed a fini di confronto diamo la rappresentazione di CSR del formato Yale, arrangiandola graficamente per far meglio notare le colonne (vedi figura a sinistra).

In questo caso abbiamo un accesso tramite righe, ma nel caso ci fosse nella parte superiore un vettore colonne potremmo anche andare in verticale, ma a differenza di come fatto nelle CSR non possiamo semplicemente ordinare le righe come se fossero banalmente consecutive, infatti c'è bisogno anche di puntatori in verticale per poter riorganizzare i dati. In particolare, in verticale dovrei avere i puntatori tra nodi con colonne corrispondenti, quindi va riorganizzata la struttura, ed intuitivamente si dovrebbero spostare i dati in modo tale da averli nella stessa colonna (figura a destra):

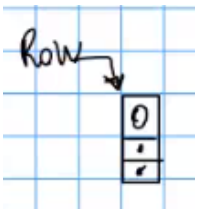




A questo punto, se avessi un array oppure una lista in cui abbiamo dei puntatori alle colonne potremmo intuitivamente andare a puntare agli inizi delle colonne e a questo punto, siccome le righe sono collegate tramite puntatori potremmo collegare anche le colonne con ulteriori puntatori:

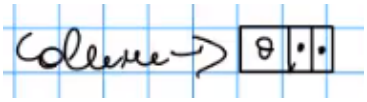


Necessariamente bisogna avere o due vettori o due liste, facciamo la rappresentazione in lista. Quest'ultima dovrà, per la lista delle righe, avere 3 campi per ogni nodo:



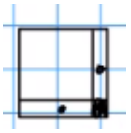
1. Numero della riga.
2. Un puntatore al primo nodo della lista dei dati per quella riga.
3. Un puntatore al prossimo nodo della lista delle righe.

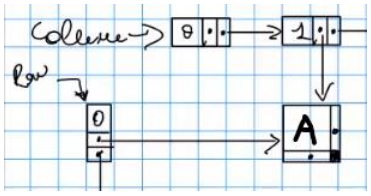
Di conseguenza abbiamo tante celle quante righe realmente presenti.
Dualmente avremo una lista per le colonne:



1. Numero della colonna.
2. Un puntatore al primo nodo della lista dei dati per quella colonna
3. Un puntatore al prossimo nodo della lista delle colonne.

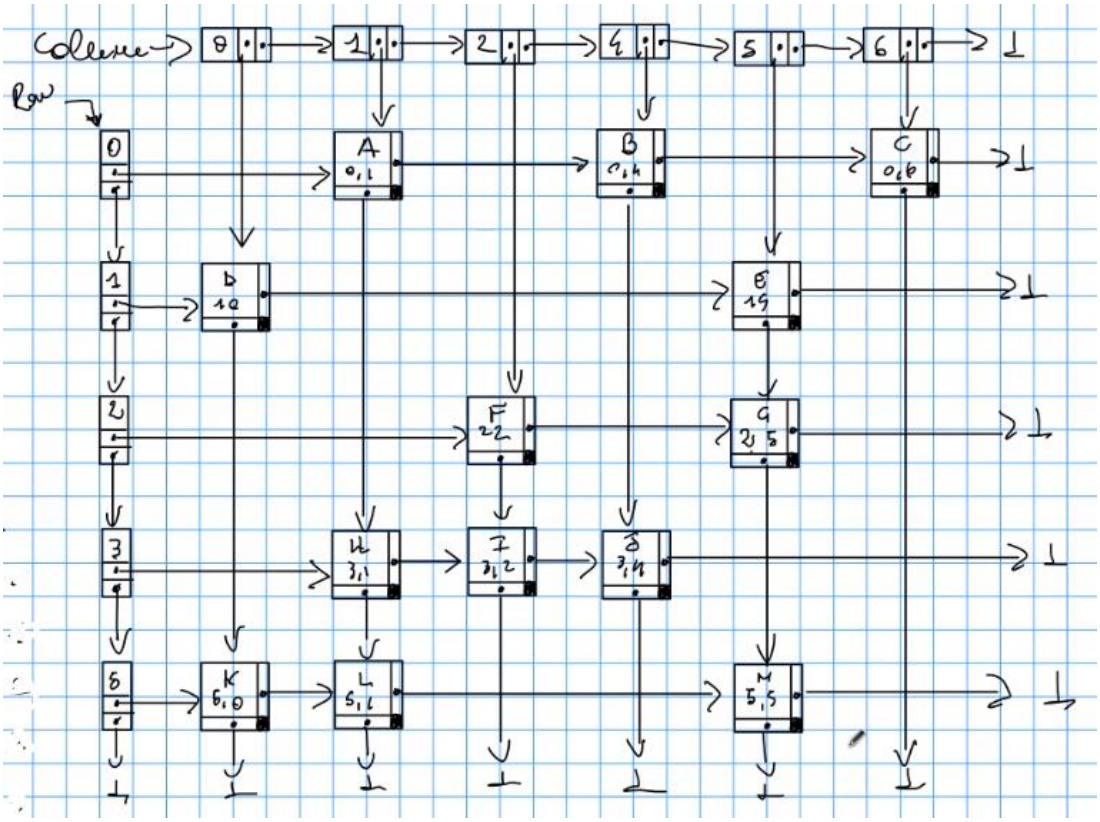
Anche i nodi degli effettivi dati avranno differenti informazioni rispetto a quanto visto con la CSR, infatti, oltre ad ovviamente il campo dato, sarà presente un puntatore alla colonna adiacente ed uno alla riga adiacente.





Prendendo la nostra matrice di esempio, il primo dato è A in posizione $(0,1)$, quindi farò puntare al primo elemento di Row il dato, e lo stesso vale per il secondo elemento di column. Quindi ogni dato avrà i suoi puntatori ai dati nella stessa riga e nella stessa colonna immediatamente adiacenti, ed un'altra informazione contenente gli indici di riga e colonna.

Procedento in questo modo avremo la seguente costruzione finale ($\perp = nullptr$):



Questo tipo di implementazione è più complessa rispetto alla CSR dovendo gestire più puntatori, ma ha il vantaggio di avere un accesso bidimensionale completo, quindi si accede a righe e colonne in maniera immediata con lo stesso costo, a differenza delle altre rappresentazioni che dipendono dal tipo di ordinamento (row-major o column-major). Praticamente potrei scegliere se accedere per riga o per colonna in base a quale tra le due liste ha il minor numero di elementi, o in base a quanto sia vicino il dato al bordo. Inoltre, il ridimensionamento è praticamente a tempo costante, infatti nell'espansione cambierò semplicemente il numero di righe e/o colonne, e in riduzione cancellerò i nodi opportuni.

22. Grafi e relative rappresentazioni

22.1 Definizione

I grafi sono uno strumento di rappresentazione (modellazione) di problemi. Sono tra le strutture dati più utilizzati poiché la soluzione di molti problemi può essere ricondotta alla soluzione di opportuni problemi su grafi, ad esempio creare una rete stradale o connettere un certo numero di abitazioni tramite tubature.

Un **grafo** G è definito come una coppia di elementi (V, U), dove V è l'insieme dei vertici e U è l'insieme degli archi (coppie di vertici). Di seguito sono rappresentati i diversi tipi di grafi:

- Un **grafo orientato** è una coppia (V, U), dove V è l'insieme dei vertici ed U l'insieme degli archi orientati; ovvero archi che hanno una coda (da dove esce l'arco) ed una testa (dove entra l'arco, rappresentata da una freccia).

- Un **grafo non orientato** è una coppia (V, U), dove V è l'insieme dei vertici ed U è l'insieme degli archi nei quali la connessione $i \rightarrow j$ ha lo stesso significato della connessione $j \rightarrow i$.

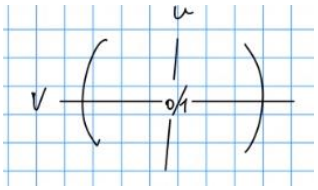
- Un **grafo pesato** è un grafo che associa l'informazione a un peso di ogni arco. Dunque, un grafo con archi che hanno dei pesi (ad esempio se si parla di strade il loro peso potrebbe variare a seconda della loro lunghezza in chilometri) è nominato grafo pesato (i pesi sono valori numerici).

- Un **grafo etichettato** è un grafo che presenta vertici e/o archi muniti di una informazione aggiuntiva (etichetta). Sono una superclasse dei grafi pesati.

21.2 Rappresentazioni

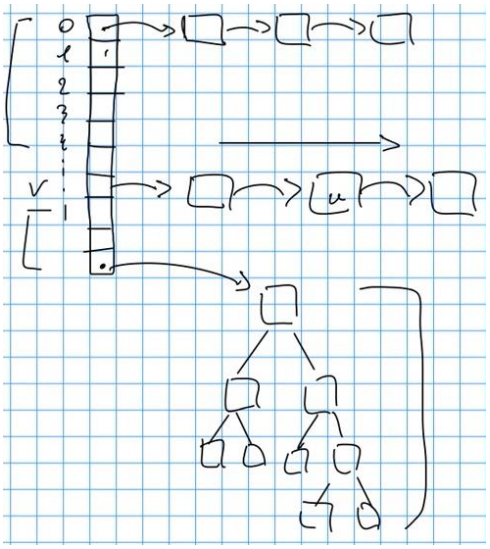
Un grafo può essere rappresentato con:

- **Matrice di adiacenza**: una matrice che ad una certa coppia di coordinate fa corrispondere il valore 0 oppure 1. Praticamente se nel nodo di indice v c'è un arco che va verso il nodo di indice u allora nella cella (v, u) ci sarà il valore 1 (lo zero rappresenta l'assenza dell'arco).



- **Lista di adiacenza**: Si ha praticamente un vettore lungo quanto i nodi presenti nel grafo ed ogni elemento del vettore (rappresentabile con indici numerici) non è altro che un puntatore ad una lsita dove sono memorizzati i vari archi di quel nodo. Nella gestione di un vettore si ha un accesso a tempo costante nella lista ma un potenziale spreco di spazio per grafi degeneri.

- **Albero di adiacenza**: analogo alla lista di adiacenza, solo che il vettore non punterà al primo nodo della lista bensì alla radice di un albero binario di ricerca. Questo consente di effettuare una ricerca binaria ma, ovviamente, si presuppone che l'albero sia bilanciato, poiché con un albero degenere non avrei alcun vantaggio rispetto alla rappresentazione con liste.



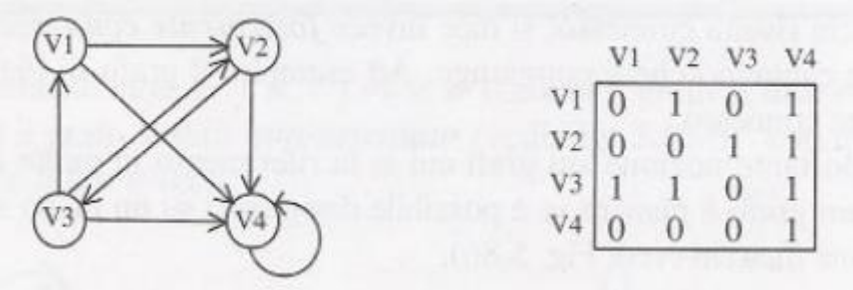
Sia per gli alberi che per la lista di adiacenza, si potrebbe pensare (nel caso di grafi dinamici) di usare invece del vettore una lista (o addirittura un albero binario), così da evitare sprechi di spazio poicé nel vettore si devono mettere i puntatori a NULL lasciando intatte le celle.

Nel caso di grafi etichettati o pesati, dove non interessa soltanto la struttura topologica ma anche l'informazione contenuta nei nodi e/o negli archi, la rappresentazione è leggermente differente da quelli appena descritti. Prendiamo, ad esempio, la matrice di adiacenza e supponiamo di voler rappresentare grafi etichettati su archi, potremmo banalmente avere, al posto di una matrice con i soli valori 0/1, una matrice di puntatori che puntano ad una struttura dati rappresentanti il dato (nullptr rappresenterà l'assenza del dato). Nelle altre rappresentazione basta modificare la struttura nodo aggiungiendo i campi etichetta.

Nel caso invece si volessero rappresentare grafi etichettati su vertici si potrebbe avere, al posto del vettore che punta ai dati, un vettore di coppie $(puntatore, etichetta)$ oppure due vettori che per ogni indice i, uno contiene il puntatore e l'altro l'etichetta. Nel caso di nodi, banalmente si modifica la struttura nodo.

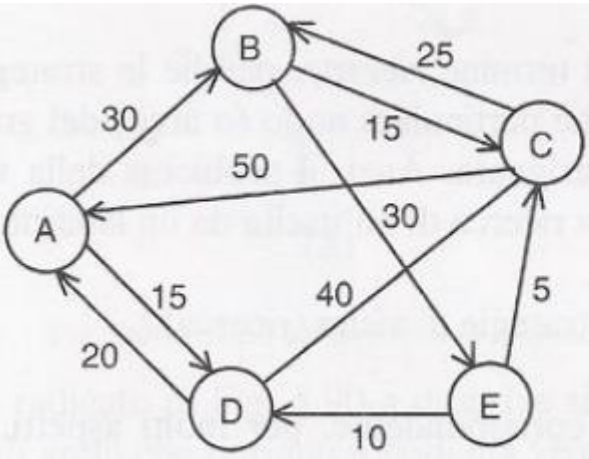
Esempi:

- Rappresentazione di un grafo orientato non pesato con matrice di adiacenza:



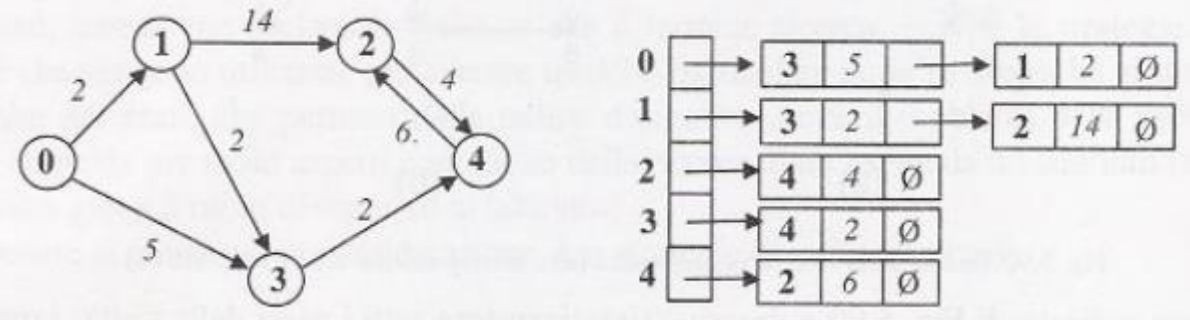
	V1	V2	V3	V4
V1	0	1	0	1
V2	0	0	1	1
V3	1	1	0	1
V4	0	0	0	1

- Rappresentazione di un grafo orientato pesato con matrice di adiacenza:



	A	B	C	D	E
A		30		15	
B			15		30
C	50	25			
D	20		40		
E			5	10	

- Grafo orientato etichettato con le liste di adiacenza:



21.3 Visita di un grafo

Per i grafi il concetto di visita o attraversamento consiste nell'esplorare, secondo una strategia, tutti i nodi e tutti gli archi di un grafo partendo da un nodo indicato come radice e percorrendo tutti gli archi presenti nella parte raggiungibile dalla radice. Un grafo può essere visitato in ampiezza e in profondità.

- **Visita in profondità**: Si parte dal nodo scelto come radice del grafo e si cerca di "fare più strada possibile" cercando sempre nuovi nodi prima di tornare indietro e prendere la più vicina diramazione.
- **Visita in ampiezza**: Si parte dal nodo scelto come radice e si visitano quindi tutti i nodi immediatamente raggiungibili da un arco, poi si visitano tutti i vertici raggiungibili con un arco da questi e così via.

In entrambi i tipi di visite è necessario ad un certo punto tornare indietro (backtracking) per visitare i vertici successivi; inoltre, essendo un grafo connesso (cioè sono presenti dei cicli), per evitare di entrare in un loop infinito, è necessario dotare il nodo di una enumeration di colori. Questa tecnica di "colorazione" del nodo permette di dotare il nodo con tre colori: bianco (nodo ancora da visitare, è anche il valore di inizializzazione di un nodo prima della visita), grigio (nodo scoperto ma non visitato), nero (nodo visitato).

Di conseguenza per grafi arbitrari la funzione di colorazione è fondamentale per la terminazione, mentre, nel caso di grafi aciclici (vedi paragrafo 21.8) non è importante per la terminazione, ma lo è per la complessità poiché senza colorazione si rischia di ottenere percorsi esponenziali.

Oltre a queste visite vedremo per i grafi anche i seguenti algoritmi:

- Algoritmi di **visita con iteratori** in ampiezza e in profondità.
- **Acyclicy Test**: test che ci consente di capire se il grafo è ciclico o aciclico.
- Iteratori per ordine topologico (**Iterator for Topological Order**): ci peremette di visitare il nodo successivo seguendo un ordinamento topologico (questo ordinamento esiste solo per grafi aciclici).
- **SCC** (componenti fortemente connesse).

21.4 Visita del grafo in ampiezza

Supponiamo di dover implementare la classica funzione map per i grafi. Innanzitutto bisogna avere una struttura accessoria per i colori. Al fine di evitare di costruire un vettore ad ogni chiamata di funzione, e poiché questo vettore lo utilizzeremo per tutti gli algoritmi è logico pensare di inserire la colorazione all'interno della struttura dati (ma bisogna ovviamente ristabilire i colori nelle funzioni).

Sicuramente necessito di una coda come funzione ausiliaria che va inizializzata con i potenziali vettori scoperti ma non visitati (quelli bianchi) oppure i vettori scoperti (neri). Si noti come in questo tipo di visita non è necessario avere tre colori ma ne bastano due. Di seguito una potenziale implementazione di una Map (supponiamo di avere un grafo G rappresentato con lista di adiacenza):

```
Map(fun) {
    Init(G); // colora tutti i nodi a bianco
    QueueVec Q;
    for (/* ogni vertice v */) {
        if (Color[v] == White) {
            Q.Enqueue(v);
            Color[v] = Grey;
            while (!Q.Empty()) {
                v = Q.Dequeue();
                fun(v);
                for (u € adiacenti(v) /* scorro la lista */) {
                    if (Color[u] == White) {
                        Q.Enqueue(u);
                        Color[u] = Grey;
                    }
                }
                Color[v] = Black;
            }
        }
    }
}
```

Questo for è necessario poiché non possiamo sapere che il primo nodo scelto ci permetta di visitare tutto il grafo (supponiamo di avere un vettore di vertici)

21.5 Visita in profondità (Pre/Post Order)

Supponiamo di voler implementare una map in pre o post order, innanzitutto settiamo i colori bianchi a tutto il grafo, poi per ogni vertice bianco visitiamo e mano mano coloriamo i vettori, e quando ho colorato tutto allora ho terminato. Di seguito un potenziale algoritmo:

```
Map(fun) {
    Init(G);
    for (v € VertexSet) {
        if (Color(v) == White) {
            DFSvisit(fun, v);
        }
    }
}
```

Per evitare un costo quadratico si potrebbe utilizzare una fold nel conteiner VertexSet (ovviamente se il container è un vettore allora non ha senso implementare una fold poiché basta scorrere con un for gli elementi del vettore).

Vediamo ora come va fatta la funzione $DFSvisit$:

```
DFSvisit(fun, v) {
    /*  Pre Order:
        fun(v);
        Color(v) = Black;
    */
    // Post: Color(v) = Gray;
    for (u € adiacenti(v)) {
        if (Color(u) == white) {
            DFSvisit(fun, u);
        }
    }
    /*  Post Order:
        fun(v);
        Color(v) = Black;
    */
}
```

Questa linea è necessaria per evitare di entrare in un ciclo durante la post, ma il tipo di colore è opzionale, infatti funzionerebbe anche messo a nero.

Il momento in cui è davvero necessario distinguere il colore grigio dal nero è per l'aciclicità.

21.6 Iteratore su grafo in ampiezza

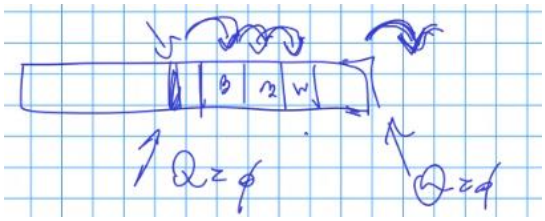
Una classe iteratore (ForwardIterator) avrà la seguente struttura:

- **Costruzione/distruzione**
  Sono necessari nella costruzione dell'iteratore i seguenti dati:
  - Puntatore o riferimento al grafo
  - Un array di colori (usare i colori all'interno del grafo rende l'iteratore meno flessibile, infatti senza un suo array di colore non si potrebbe usare ad esempio lo stesso iteratore in due cilci for innestati oppure usare l'iteratore in una funzione che colora il grafo) così da evitare i problemi di sovrapposizione di colori.
  - Una Coda (visto che siamo in ampiezza)
  - Puntatore al nodo corrente

  Il distruttore (ammenoché non si aggiungano dati dinamici) è quello di default.
  Andiamo a definire il costruttore $constructor(const\ G\&)$:
  - $curr\ =\ 0$; (supponendo che si tratti di un grafo con struttura array, quindi gli diamo il primo indice) oppure $curr\ =\ G.firstnode$
  - Inizializzazione dell'array dei colori a bianco
  - Mettere il vertice corrente a nero con $Color(curr) = Black$

- **Access operatore itr***

- **Terminazione**
  Il Termination check testa solo lo svuotamento della coda e se i successivi indici dell'array non sono bianchi. Di conseguenza è necessario controllare che la coda sia vuota e che tutti i nodi siano neri. Dobbiamo in qualche modo avere un'informazione sulla posizione all'interno della struttura (il linear container dei nodi) e controllare che tutti i nodi siano neri.



- **operator ++ ( )**
  Assumiamo che il current sia stato già visitato e che ovviamente l'iteratore non sia terminato:

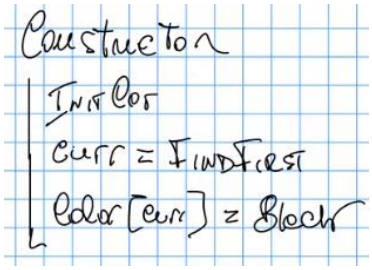
```
operator++() {
    for (u € adiacenti(curr)) {
        if (Color[u] == White) {
            Q.Enqueue(u);
        }
    }
    if (!Q.Empty) {
        curr = Q.HeadNDequeue();
        Color(curr) = Black;
    }
    else {
        for(; curr < size && Color(curr) != White; ++curr) { }
        if (curr < size) {
            Color(curr) = Black;
        }
    }
}
```

Questo for, che non necessita di un corpo, si ferma quando curr è bianco oppure quando curr = size, quindi si può semplificare il termination test semplicemente con al condizione curr == size.

21.7 Iteratore su grafo in PreOrder

Le informazioni necessarie sono il nodo corrente, uno stack ed un vettore di colori.

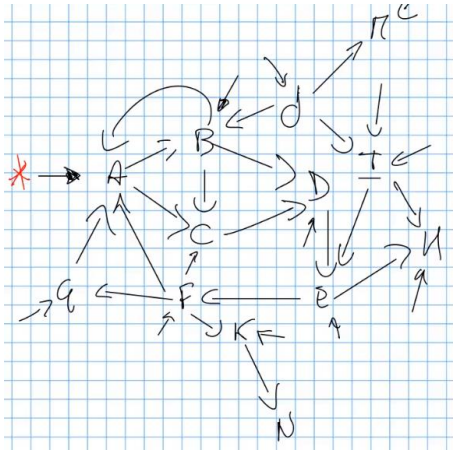
Per il costruttore di un iteratore in PreOrder bisogna innanzitutto reinizializzare i colori a bianco.



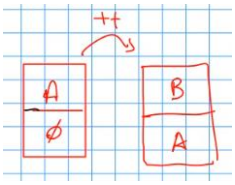
Il nodo corrente deve puntare al primo nodo (se necessario con una funzione FindFirst) ed ovviamente settare il suo colore a nero (poiché essendo in preorder assumo quel nodo già visitato).

Lo Stack è ovviamente vuoto e quindi ho completato il mio costruttore.

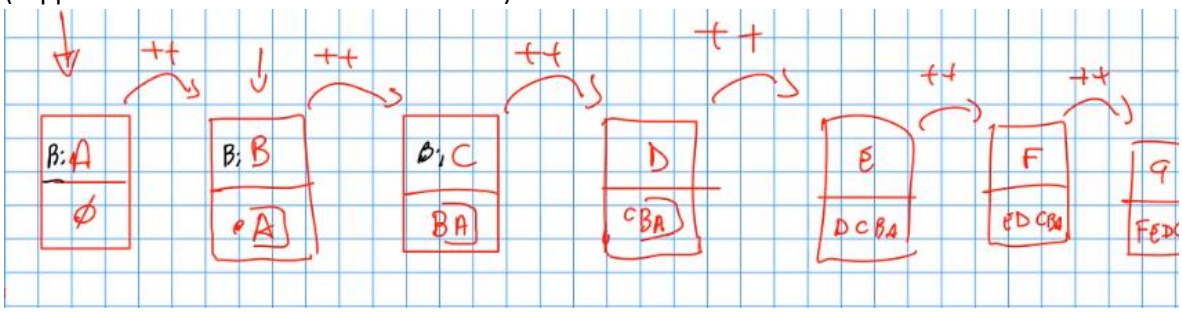
Esempio concreto di grafo (utilizzeremo questo grafo per spiegare la pre order):



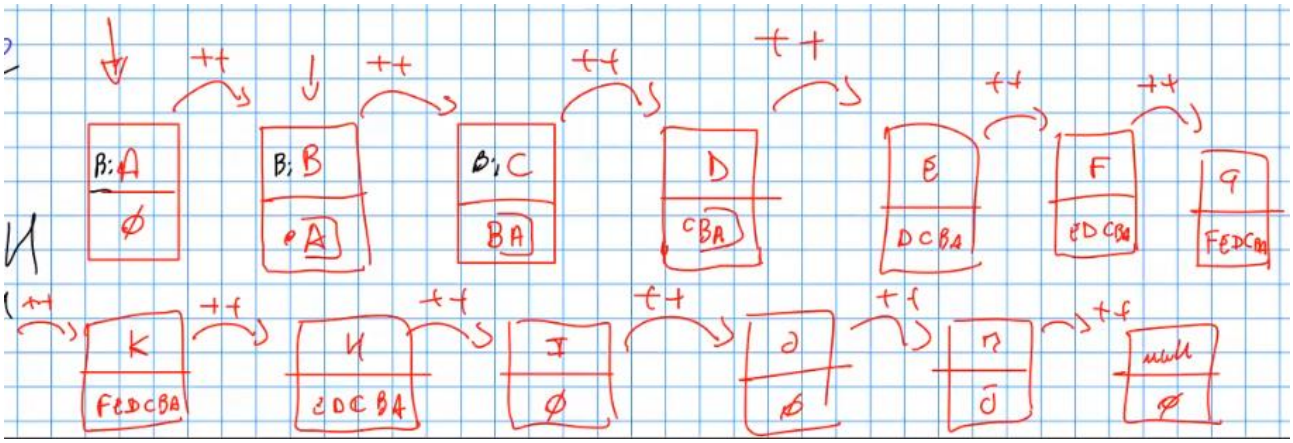
Supponiamo di iniziare con il nodo A; il ragionamento per il successor è il seguente: dopo A mi sposto al primo bianco successivo, ma al fine di non perdere l'informazione di A per trovare gli eventuali altri successivi devo preservarlo e quindi metterlo nello stack.



(con B che diventa nero), Nel nostro grafo un potenziale percorso è il seguente (supponendo la testa dello stack a sinistra):



A questo punto abbiamo tutti gli adiacenti neri (lo stesso succede nel caso non ci saranno aidacenti) e dunque posso procedere facendo la top (cercando potenzialmente i nodi di F):



È possibile notare che, arrivati al nodo G (che ha archi uscenti a nodi già visitati) si arriva ad una situazione di "stallo". Quindi si riparte da F (con una Top(), e inserisco K). Da qui ripeto l'operazione precedente e si continua con lo scorrimento del grafo. Nel caso si voglia avere un iteratore arbitrario (generico), che non vada a sfruttare l'informazione concreta, e che quindi possa essere realizzata su un concetto di grafo generico, la cosa più semplice da fare è tener traccia del nodo corrente e poi di accedere sempre agli adiacenti per andare a vedere il prossimo (o avere un'informazione su quale sia il prossimo adiacente per mezzo di un operatore, indice numerico o una qualche informazione che permetta di capire quale sia il prossimo nodo).

Quindi una possibile implementazione del successor è un while con la condizione "ho ancora qualcosa da visitare" e il corpo corrisponde al for del DFSvisit:

```
operator++() {
    while (/* ho qualcosa da visitare */) {
        DFSvisit(curr);
        // cerco una nuova radice
        SearchForNewRoot();
    }
}
```

21.7 Iteratore su grafo in PostOrder

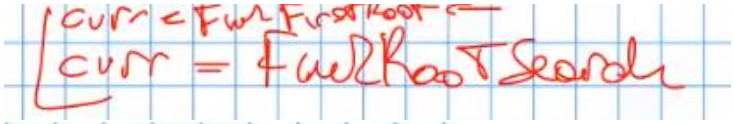
Come per la Pre Order abbiamo bisogno del nodo corrente, di uno stack e di un vettore di colori.

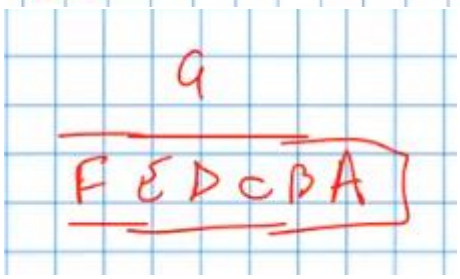
Andiamo innanzitutto a definire il costruttore:

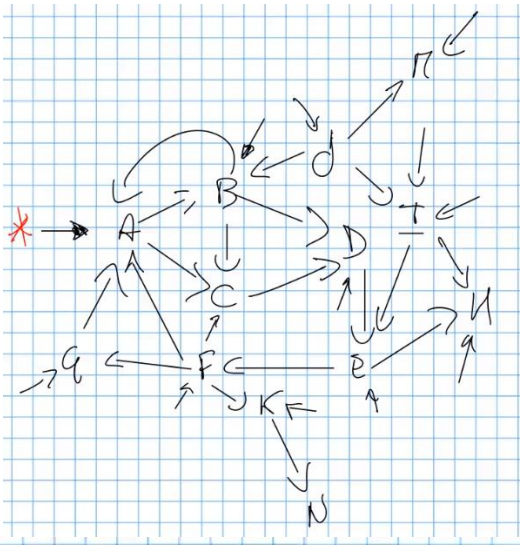
Tutto ciò che è sullo stack è Grigio mentre il current punta al nodo visitato che è nero.

Il nostro costruttore deve teoricamente cercare una foglia del primo nodo (in questo caso G).

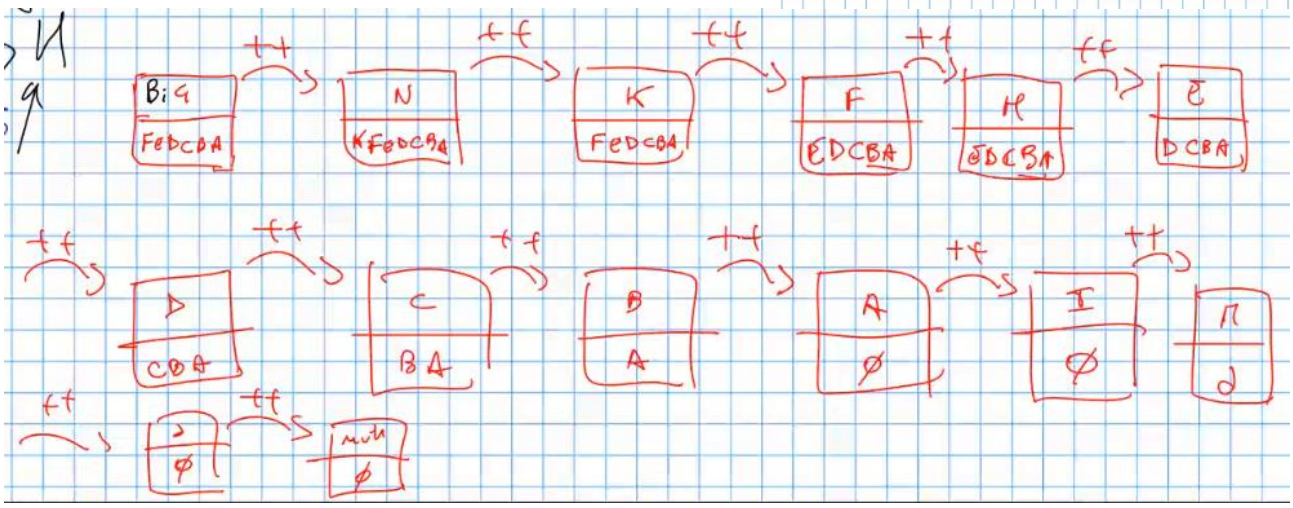
Si inizializza i colori a bianco, si cerca il primo vertice e a questo punto si costruisce lo stack:





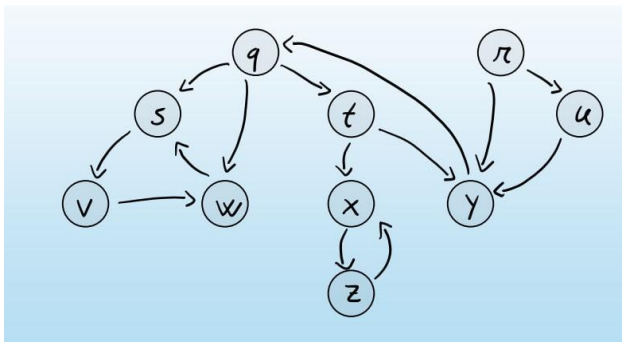


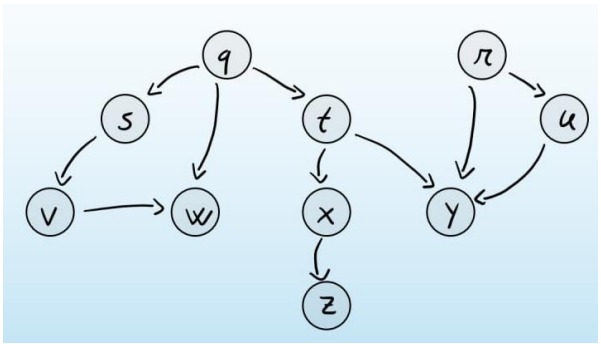
Il nostro successor sarà, supponendo di avere il grafo seguente:



21.8 Acyclicy test

Ovviamente sarà una funzione booleana che restituisce true per grafo aciclico e false per grafo non aciclico. Con grafo **ciclico** intendiamo un grafo che ha almeno un loop, con **aciclico**, invece, intendimo un grafo che nessun loop. Esempio di grafo ciclico a sinistra e aciclico a destra (forniti dal buon M.G. Carofano):





Al fine di capire l'aciclicità in un grafo è molto efficiente la visita in Post Order, infatti una condizione necessaria, ma non sufficiente, affinché il grafo sia aciclico è quello di avere una foglia, bisogna praticamente creare una DFS, partendo da tutti i nodi radice di ricerca, andando a cercarmi gli archi di ritorno (quelli che

chiudono il ciclo), ovvero quelli che terminano con colore grigio. Quindi è necessaria la tripla colorazione con i bianchi che rappresentano i nodi che non ho ancora visitato, i neri rappresentano parti potenzialmente completamente scollegate e i cicli con archi di ritorno (quindi ciclici) sono necessariamente grigi.

Praticamente per testare che un grafo sia aciclico bisogna seguire i seguenti passi:

1. Se il grafo non ha nodi è aciclico quindi mi fermo.
2. Se il grafo non ha foglie mi fermo perché ho un grafo ciclico.
3. Scegliere una foglia del grafo. Rimuovere la foglia e tutti gli archi interni nella foglia per ottenere un nuovo grafo (qui è evidente la necessità di usare la post order).
4. Ripetere il punto 1 con il nuovo grafo.

21.9 Ordinamento topologico

Innanzitutto dato un grafo orientato aciclico G, definiamo un **ordinamento topologico** su G, un ordinamento lineare dei suoi vertici tale che: se G contiene l'arco (u, v), allora u compare prima di v nell'ordinamento, più in generale, se esiste un percorso da u a v, allora u compare prima di v nell'ordinamento. Vediamo adesso come creare una funzione per il Topological Ordening per la nostra libreria:

Supponiamo di avere un grafo G ed una funzione $TopologicalOrdering()$ che non ha parametri e restituisce un opportuno iteratore, quest'ultimo ha ovviamente operatore di accesso *, successore ++ e un booleano di terminazione. Ovviamente questo iteratore nel momento in cui andrà a dereferenziare una certa posizione (ad es. la prima) indicherà un nodo del grafo che non ha archi entranti, e con l'operatore ++ dovrà creare una sequenza di elementi che una volta dereferenziati, creerà la permutazione (modo di ordinare in successione oggetti distinti) dei nodi.

Per la costruzione della $TopologicalOrdering()$si potrebbe usare l'algoritmo del **grado entrante**:

```
{
    ComputeInDegree();
    InitQueue(Vertex == 0);
    while (!Q.Empty()) {
        v = Q.HeadNDequeue();
        // applicare qui la funzione f(v) nel
        // caso si voglia fare una map/fold
        // seguendo l'ordinamento topologico
        for (u € adiacenti(v)) {
            // Grado entrante di u = D[u]
            D[u]--;
            if (D[u] == 0) {
                Q.Enqueue(u);
            }
        }
    }
}
```

Algoritmo che calcola i gradi del vertice: si inizializzano tutti i valori a 0 e poi vengono incrementati i vertici degli adiacenti

Inserisce in coda tutti i vertici pari a 0 della ComputeInDegree

In questo algoritmo non usiamo la colorazione, ma questa è implicitamente definita con il grado entrante, infatti i miei nodi scoperti e visitati saranno quelli con grado entrante pari a 0.

La $TopologicalOrdering()$ potrebbe essere implementata anche con l'algoritmo di **DFS**, dove ovviamente la colorazione è necessaria. Nello pseudo codice con Pi si intende una struttura dati che sia Stack oppure Queue, a seconda che si uso lo stack o la queue ho sempre lo stesso ordinamento ma con accesso differente. Infatti per lo stack ho i vertici meno vincolati in cima, invece in una queue l'ulitmo elemento sarà quello più vincolato (coda), ma essendo una coda accedo sempre prima ai meno vincolati (testa).

Di seguito una implementazione dell'algoritmo di DFS:

```
{
    InitColor();
    for (v € VertexSet) {
        if (Color(v) == White) {
            Pi <- DFSvisit(G,v, Pi);
        }
    }
}
```

```
DFSvisit(G,v, Pi) {
    Color(v) = Gray;
    for (u € adiacenti(v)) {
        if (Color(u) == white) {
            Pi <- DFSvisit(G,v, pi);
        }
    }
    Color(v) = Black;
}
```

Vediamo ora come sfruttare questi algoritmi per costruire il nostro iteratore, innanzitutto la nostra struttura avrà un $current$, un vettore $Degree$ sui gradi entranti ed una coda Q (ovviamente si ricorda che la Topological Ordering ha senso solo sui grafi aciclici):

- **Constructor**:

```
{
    ComputerInDegree(); // scorro l'insieme dei vertici
                        // e popolo il vettore Degree
                        // con i gradi entranti del Grafo
                        // così da poterli poi modificare
    for (v € Vertex) {
        if (Degree[v] == 0) {
            Q.Enqueue(v);
        }
    }
    current = Q.HeadNDequeue();
}
```

- **Successor Operator** (++):
  Essendo current già visitato va aggiornato il vettore Degree a partire da current.

```
{
    for (u € adiacenti(current)) {
        Degree[u]--;
        if (Degree[u] == 0) {
            Q.Enqueue(u);
        }
    }
    if (!Q.Empty()) {
        current = Q.HeadNDequeue();
    }
    else {
        current = NULL;
    }
}
```

Abbiamo sfruttato l'algoritmo del grado entrante, se volessimo invece sfruttare la DFS dovremmo usare un vettore dei colori al posto del $Degree$ ed al current dare quindi il primo bianco. Praticamente si usa l'algoritmo di iteratore in post order che fa esattamente quello con l'unica differenza che visita prima quello più vincolato, quindi usarlo così com'è andremmo ad utilizzare l'ordinamento topologico inverso, si potrebbe dunque costruire un iteratore offline.

21.10 Nozioni dell'ultima mezzora (rubate dal git di Valentino)

Iteratori Online e Offline:

- **OFFLINE** → nel momento in cui viene creato l'iteratore creiamo anche l'intera struttura che verrà poi sfruttata nella visita. Ritorna utile durante la visita dell'intero grafo (spendo più tempo nella costruzione per avere una maggiore velocità dopo).
- **ONLINE** → questa operazione viene invece fatta al momento. Ritorna utile nel diluire il tempo di calcolo a cavallo di tutte le operazioni di Successor (presenta un tempo di costruzione minore, ed è preferita nel caso in cui non debba o non sia sicuro di dover visitare l'intero grafo).

Memo di Valentino:

- Esempio di applicazione dell'ordinamento topologico inverso → Calcolo delle componenti fortemente connesse.
- Per avere il trasposto (InDegree e OutDegree) senza doverlo calcolare ogni volta (esplicitamente dichiarato nel grafo) si sfruttano gli indici (si usano dei flag che permettano di contenere il dato, che verrà usato da una funzione trasponi).
- Un caso di struttura che permette ciò è quella delle liste ortogonali. Con la lista di adiacenza questo non è possibile.

Capitata come domanda di esame (risposta di G. Aiello):

- la **RAII** è una tecnica di buona norma per la gestione di memoria dinamica, e cioè che facciamo le new e delete incapsulate nei costruttori e distruttori (si evitano quindi le naked)