RELAZIONE DI PROGRAMMAZIONE AD OGGETTI

TREEKALK

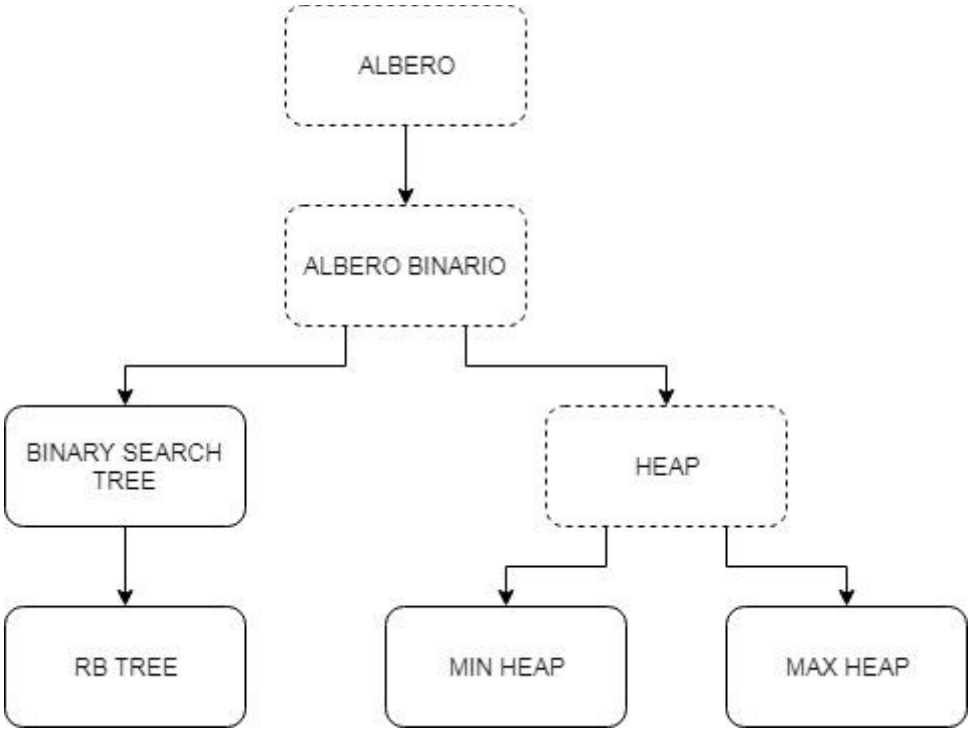
Mazzalovo Diego 1142519

ABSTRACT:

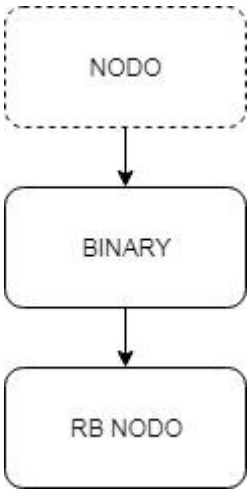
TreeKalk è una calcolatrice che consente all'utente di eseguire calcoli su alberi(in questa versione sono stati implementati alberi binari)

NEL PROGETTO SONO PRESENTI 2 GERARCHIE DI TIPO LOGICO:

Albero:



Nodo:



Gerarchia Nodo:

- La base della gerarchia è la classe nodo, che è caratterizzata dal contenere un'informazione di tipo double, un vettore di nodi contenente puntatori ai figli del nodo e un ulteriore puntatore al padre del nodo.
- Binary deriva da nodo ma non ha nessun campo in più. Ho fatto questa classe al solo scopo di semplificare le operazioni sugli alberi binari.
- Rb Nodo deriva da binary ed in più ha il campo colore che può essere rosso,nero o dn(doppio nero).

Gerarchia Albero:

- La base della gerarchia è la classe Albero, classe astratta, caratterizzata semplicemente dalla radice dell'albero che è un puntatore a nodo e da 2 campi di tipo QString nome e tipo.
- Albero binario deriva da Albero ed è ancora astratto ed utilizza nodi di tipo binary.
- Binary Search Tree deriva da albero binario ed implementa le funzioni astratte di albero e albero binario.
- Rb Tree deriva da Binary Search Tree ed utilizza nodi di tipo rb.
- Heap deriva da albero binario, risulta ancora essere astratta ed in più aggiunge 2 metodi astratti heapify e heapifyup.
- Max Heap e Min Heap derivano entrambi da Heap ed implementano i suoi metodi.

Metodi polimorfi:

Classe Albero:

- **Virtual ~albero();** // distruttore virtuale
- **Virtual void insert(double i);** // consente di inserire un dato nell'albero
- **Virtual void cancella(nodo* n); e virtual void cancella(double i);** // consentono di cancellare rispettivamente il nodo specifico all'interno dell'albero o la prima ricorrenza del dato richiesto dall'albero.
- **Virtual bool search(double i)const;** // consente di ricercare un dato all'interno dell'albero. Il metodo è virtuale in quanto anche se la ricerca di base nodo a nodo andrebbe bene in ogni albero, essa è raffinabile nei derivati
- **Virtual albero* copy();** // restituisce una copia profonda dell'albero senza condivisione di memoria.
- **Virtual void modifica(nodo* n,double i);** // modifica l'info del nodo n (qualora presente) in i
- **Virtual double massimo()const e minimo()const**; // trovano massimo e minimo nell'albero.Vale lo stesso discorso precedentemente fatto per search.
- **Virtual Qstring ordina()const**; // ritorna una stringa contenente i dati dell'albero ordinati in maniera crescente
- **Virtual albero* operazione(albero* a,char tipo)const;** // consente di eseguire un tipo di operazione tra [+,-,*,/] tra 2 alberi e ritorna un nuovo albero del tipo dell'albero che invoca la funzione,lasciando invariati i 2 operandi.(le 4 operazioni si intendono di somma,sottrazione,moltiplicazione e divisione nodo a nodo).
- **Virtual albero* unione(albero* a)const;** // ritorna un nuovo albero contenente tutti i nodi dei 2 alberi.
- **Virtual albero* intersezione(albero* a)const;** // ritorna un nuovo albero contenente tutti e soli i nodi presenti in entrambi gli alberi.
- **Virtual albero* operator+(albero* a)const, Virtual albero* operator+(albero* a)const, Virtual albero* operator+(albero* a)const, Virtual albero* operator+(albero* a)const;** // sono gli overload degli operatori usati per fare le rispettive operazioni.

Classe Heap:

- **Virtual void heapify(binary* b);** // fa scendere l'info nell'albero se necessario, scambiandola di volta in volta con uno dei suoi figli (se maxheap scende qualora un figlio abbia info maggiore di b, se minheap scende qualora un figlio abbia info minore di b)
- **Virtual void heapifyUp(binary* b);** // fa salire l'info nell'albero se necessario, scambiandola di volta in volta con suo padre (se maxheap sale qualora il padre abbia info maggiore di b, se minheap sale qualora il padre abbia info minore di b)

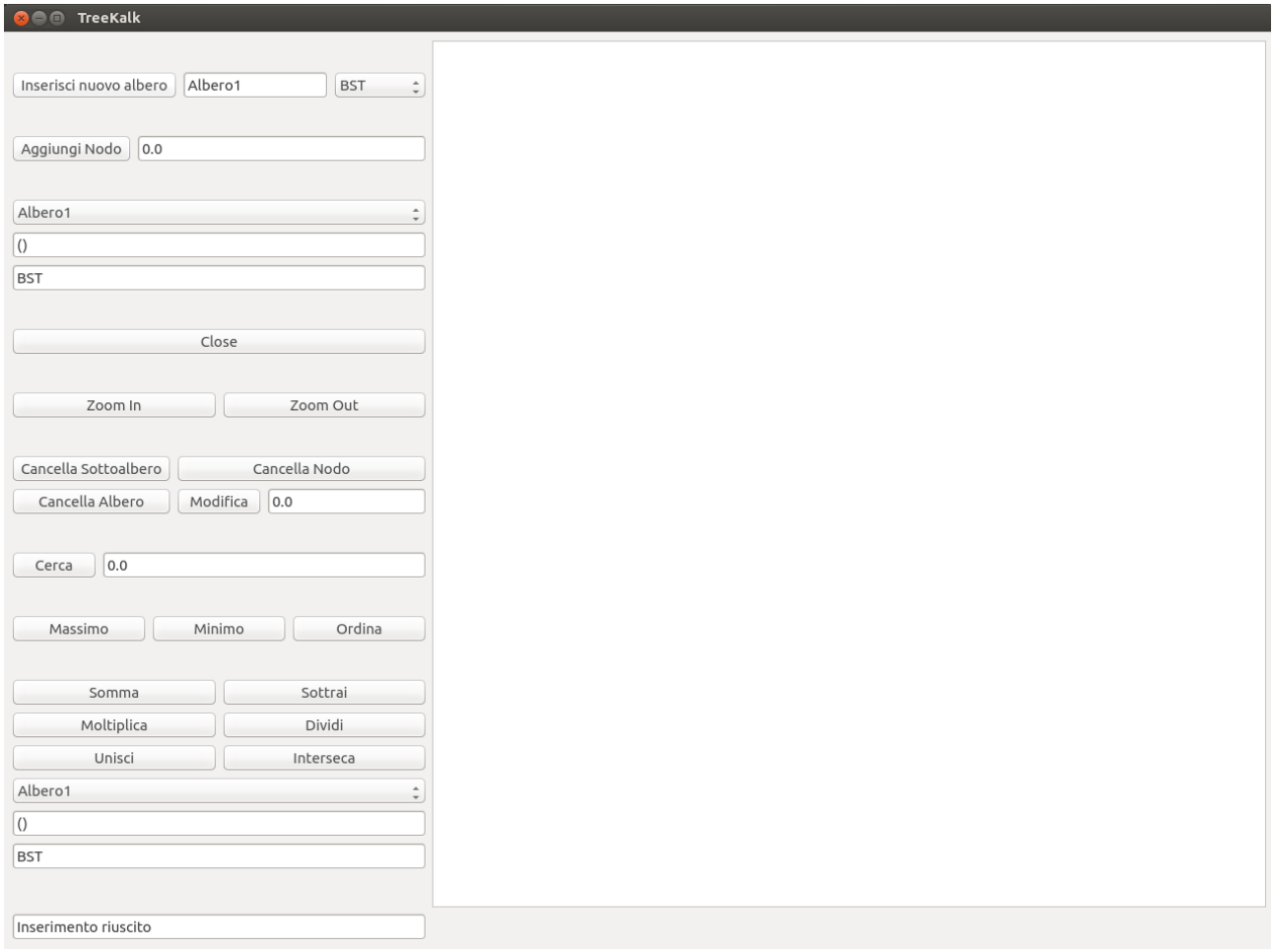
Classe nodo:

- **Virtual ~nodo();** //distruttore virtuale
- **Virtual nodo* getPadre()const;** // ritorna il padre del nodo
- **Virtual nodo* copy()const** //ritorna una copia profonda del sottoalbero radicato in this

Classe binary:

- **Virtual binary* left()const e right()const;** // ritornano rispettivamente figlio sinistro e destro del nodo

Manuale GUI:



Per utilizzare treekalk, per prima cosa bisogna inserire almeno un albero:

- Scegliere il nome per l'albero e scriverlo nel riquadro dove nell'immagine appare scritto Albero1
- Scegliere un tipo per l'albero selezionandolo nel menù a tendina dove c'è scritto BST
- Premere su inserisci nuovo albero.

Una volta inserito un albero, per inserirne altri,scegliere nomi divesi. Quando viene inserito un nuovo albero esso diventa in automatico quello selezionato.

Per inserire un nodo nell'albero, basta scrivere il dato che si vuole inserire a fianco di inserisci nuovo nodo e premere nel tasto. Ad ogni inserimento di nodi, si aggiornerà l'immagine dell'albero a destra e d anche la stampa dell'albero sotto al suo nome(dove al momento ci sono le parentesi ()). La stampa è composta da (info(figlio sx),(figlio dx)).

Per vedere l'info contenuta in un nodo è sufficiente andarci sopra col mouse senza cliccare.

Per cambiare albero basta selezionarlo dal menù a tendina cliccando sul suo nome e poi di possono aggiungere nodi.

In ogni momento è possibile selezionare un nodo dell'albero cliccandoci sopra e poi premendo i rispettivi tasti : cancellarlo,cancellare il sottolabero radicato nel nodo(non disponibile negli rb),modificare l'info del nodo(ad ognuna di queste operazioni corrisponde un automatico aggiustamento dell'albero). È anche possibile cancellare un albero selezionandolo e premendo su cancella albero(non è possibile cancellare l'ultimo albero rimasto).

Dopo aver selezionato un albero è possibile premere su massimo, minimo,ordina o cerca(per trovare l'info nel corrispondente riquadro a dx) e il risultato apparirà nel riquadro in basso.

Per eseguire un'operazione, bisogna selezionare il primo albero(albero1) dal menù a tendina sotto al tasto di inserimento nodo ed il secondo albero(albero2) dal menù a tendina sotto ai bottoni unisci ed interseca e premere uno dei sei tasti [somma,sottrai,moltiplica,dividi,unisci,interseca]. Verrà così invocato il comando albero1.operator[+,-,*,/](albero2) o albero1.unisci/interseca(albero2), che restituiscono sempre un nuovo albero del tipo di albero1, il quale diventa subito l'albero selezionato. Per poter eseguire le prime 4 operazioni, è necessario che i 2 alberi selezionati abbiano la stessa forma(stessi nodi con stessi figli non nulli ad esempio un minheap (3(5)(8)) e un maxheap(10(7)(4)) possono essere sommati, mentre un bst (6(2(1)())())) e un maxheap(10(7)(4)) no. Non serve che i 2 alberi siano dello stesso tipo).

Sono disponibili inoltre i tasti zoom in e zoom out per poter meglio visualizzare l'albero.

Eccezioni:

dal momento che il progetto è di piccole dimensioni non ho ritenuto necessario sviluppare classi per le eccezioni, limitandomi ad utilizzare delle QString il cui scopo è di modificare il display della calcolatrice.

CONSIDERAZIONI:

- Il progetto è sviluppato con il pattern MVC
- Sarebbe stato interessante sviluppare la calcolatrice templatizzata per qualsiasi tipo, ma purtroppo visto il numero di ore concesse non è stato possibile.
- Gli alberi utilizzano le 3 classi di nodo come radici e nodi dell'albero. Ho optato per questa separazione, principalmente per il fatto che non in tutti i tipi di albero, se si prende un nodo interno all'albero, esso risulta ancora essere un albero di quel tipo (come negli alberi rb, dove la radice deve essere nera, quindi se si prendesse un nodo rosso, esso non sarebbe a sua volta albero rb).
- Nonostante le classi nodo e albero vengano utilizzate solo per alberi binari all'interno del progetto, mi è sembrato necessario aggiungerle comunque, in quanto sarebbero state la base per altri tipi di alberi, come alberi ternari o quaternari(che avrei voluto aggiungere ma non c'era abbastanza tempo), dal momento che garantiscono i metodi di base per poter gestire qualsiasi tipo di nodo(come getNodo(int i)const che restituisce il figlio i-esimo se presente, che in binary viene sostituito da binary*left()const e right()const e

setFiglio(int i,nodo*n) per settare il figlio i-esimo del nodo che poi diventa setLeft(binary*n) e setRight(binary* n)).

COMPILAZIONE :

- Entrare nella cartella TreeKalk.
- qmake -project “QT+=widgets”
- qmake
- make
- ./TreeKalk

JAVA:

Nella cartella java è presente il file Use.java con il main di prova delle funzionalità di Treekalk avviabile coi comandi javac Use.java e java Use.

ORE RICHIESTE :

- Analisi e progettazione modello: 13 ore
- Apprendimento strumenti qt: 10 ore
- Implementazione parte grafica: 17 ore
- Debug e test: 12 ore
- Codifica in java: 4 ore
- Relazione: 2 ore

Totale ore:58

Ho sforato il limite di 50 ore in quanto lo sviluppo di certe classi ha richiesto più tempo del previsto(in particolare rb alberi) e all’inizio era difficile capire quanto sarebbe stato il tempo per apprendere ed implementare la parte grafica.

VERSIONE DI QT USATA:

Qt Creator 4.4.1 Based on Qt 5.9.2

GCC 5.3.1