ASD RIPASSO LAB

Malchiodi Riccardo

Indice

Premessa

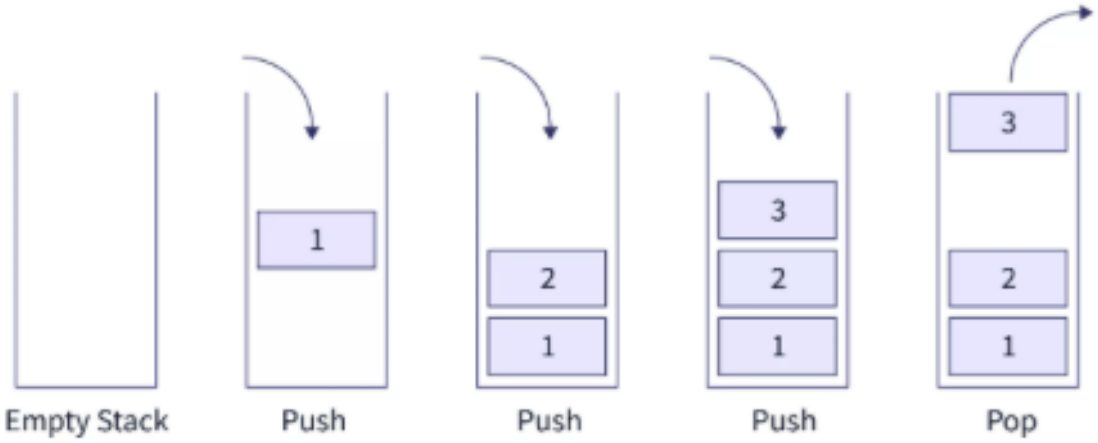
All’interno di questo pdf troverete la spiegazione generale di alcuni laboratori con qualche accenno teorico. Questo documento è utile solo per un Ripasso per le strutture dati più comuni. Se siete in cerca di spiegazione dettagliate per ciascun laboratorio con l’implementazione completa vi consiglio di guardare la repository dedicata su GitHub.

Buona lettura.

Stack

La pila è un insieme di elementi caratterizzata dal fatto che le operazioni che faccio avvengono sempre alla stessa estremità, chiamata “cima”.
Questo significa che l’ultimo elemento inserito sarà il primo ad essere rimosso, seguendo il principio LIFO (Last In, First Out).
Quando aggiungo un elemento alla mia pila, lo inserisco in cima, sovrapponendolo all'elemento precedente. Allo stesso modo, quando devo rimuovere un elemento, lo tolgo dalla cima della pila, cioè dall'estremità in cui è stato aggiunto per ultimo.
Per esempio, se immagino una pila di piatti, ogni nuovo piatto viene impilato sopra gli altri. Se voglio prendere un piatto, devo iniziare dall'ultimo che ho messo, poiché è quello più facile da raggiungere. Lo stesso concetto si applica alla struttura dati della pila.

Implementazione



Operazioni push e pop

Innanzitutto dobbiamo fornire la dimensione massima della nostra pila, essa viene chiamata (in questo caso) BLOCKDIM.
La pila nel nostro laboratorio è un array.

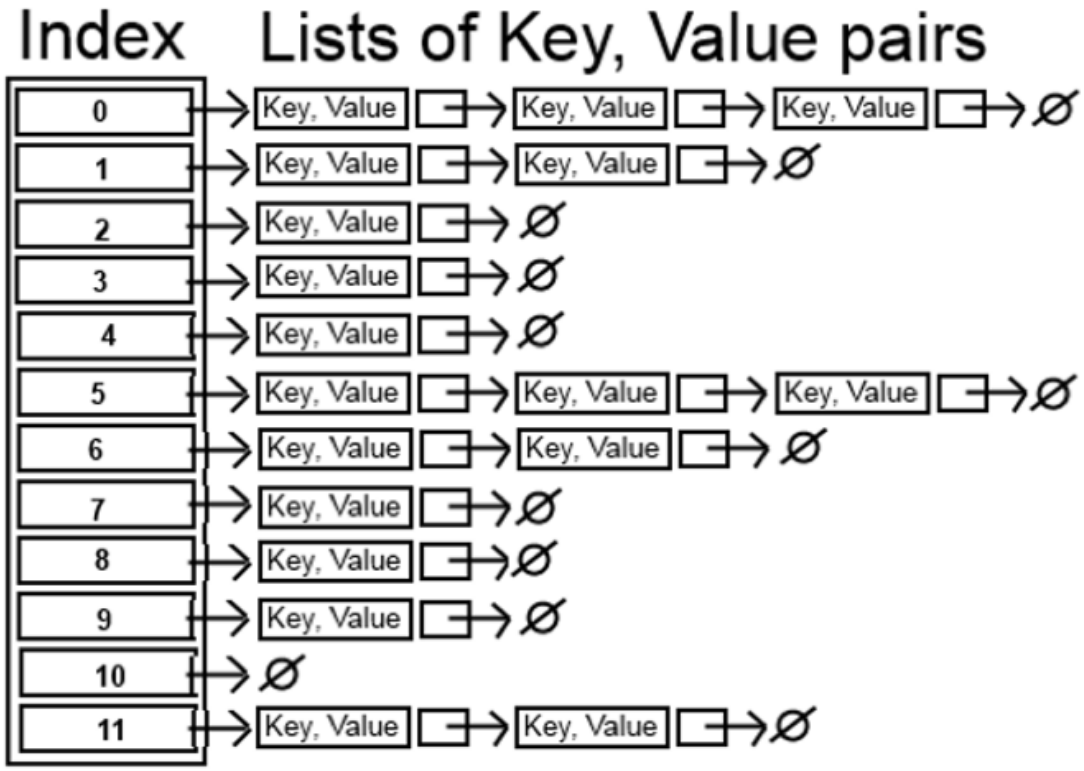
Creazione Pila Vuota: Per creare una pila vuota dobbiamo allocare un array di dimensione BLOCKDIM.
Questo array sarà utilizzato per memorizzare gli elementi della pila. L'array viene assegnato al membro data dell'oggetto sret.
La variabile membro maxsize di sret viene impostata al valore di BLOCKDIM mentre la variabile membro size viene impostata a 0.

Operazione PUSH: La funzione push serve per aggiungere un elemento (el) in cima allo stack (st). L'implementazione della funzione tiene conto del fatto che lo stack potrebbe raggiungere la sua capacità massima, e in tal caso, è necessario espanderlo per poter inserire ulteriori elementi. La funzione push serve per aggiungere un elemento (el) in cima allo stack (st). L'implementazione della funzione tiene conto del fatto che lo stack potrebbe raggiungere la sua capacità massima, e in tal caso, è necessario espanderlo per poter inserire ulteriori elementi. Se lo stack è pieno, la funzione procede a creare un nuovo array con una dimensione maggiore. La nuova dimensione (NewDimensione) è calcolata aggiungendo il valore di BLOCKDIM a st.maxsize, aumentando così la capacità dello stack. Successivamente, viene allocato un nuovo array (NuovoArray) di tipo Elem con la nuova dimensione. Tutti gli elementi presenti nel vecchio array vengono copiati nel nuovo array utilizzando un ciclo for.
Dopo aver copiato gli elementi, il vecchio array viene eliminato con delete[] st.data per evitare problemi di memory leak (perdita di memoria non più utilizzabile).
Infine, il puntatore st.data viene aggiornato per puntare al nuovo array,
e st.maxsize viene aggiornato per riflettere la nuova capacità dello stack.
Una volta garantito che c'è spazio sufficiente nello stack, la dimensione attuale dello stack (st.size) viene incrementata di 1.
L'elemento el viene quindi aggiunto alla nuova posizione in cima allo stack, che corrisponde all'indice st.size – 1 nell'array st.data.

Operazione POP: semplicemente posso andare a diminuire la size per potere eliminare l’ultimo elemento.

Dizionari e Tabelle di Hash

Un dizionario è una struttura dati che permette di memorizzare coppie di elementi sotto forma di chiave e valore. Questa struttura consente di accedere ai valori in modo rapido utilizzando la chiave corrispondente. Il dizionario è noto per il fatto che le chiavi sono univoche, il che significa che a ogni chiave può essere associato un solo valore, ma un valore può essere associato a più chiavi diverse.
I dizionari sono estremamente utili quando si ha bisogno di un accesso rapido ai dati basato su un'identificazione univoca (la chiave).
Una tabella di hash è una struttura dati utilizzata per implementare un dizionario. Questa struttura consente di memorizzare coppie chiave-valore in modo tale che il tempo di accesso a un valore, dato la chiave, sia approssimativamente costante, anche se il numero di elementi nel dizionario è grande.



Implementazione Hash

Search: questa funzione mi garantisce di andare a scorrere tutta la lista per cercare l'elemento con la chiave corrispondente.

Insert: L'operazione di Insert consente di inserire un nuovo elemento nella tabella di hash. Prima di procedere con l'inserimento, è buona pratica utilizzare la funzione *Search* per verificare se esiste già un elemento con la stessa chiave. Se la chiave non è già presente, l'inserimento può avvenire.

In generale è comune inserire il nuovo elemento *in testa* alla lista collegata nel bucket per semplicità e velocità (dato che questo non richiede l'attraversamento dell'intera lista).
Tuttavia, l'inserimento in coda potrebbe essere un'altra opzione, anche se meno comune, a seconda delle esigenze specifiche dell'implementazione.

Delete: La funzione Delete permette di rimuovere un elemento specifico dalla tabella di hash. Per eseguire questa operazione, è necessario gestire alcuni casi particolari, come la rimozione dell'elemento in testa o in coda della lista collegata nel bucket. Se l'elemento da eliminare si trova in testa alla lista, occorre aggiornare il puntatore del bucket per escludere l'elemento rimosso. Se l'elemento è in coda, bisogna aggiornare il puntatore del penultimo elemento per escludere quello rimosso. Anche la rimozione di un elemento al centro della lista richiede un aggiornamento accurato dei puntatori per mantenere l'integrità della lista.

BST

Un **Binary Search Tree** (BST), o **Albero Binario di Ricerca**, è una struttura dati ad albero che ha la proprietà fondamentale di facilitare la ricerca, l'inserimento e la cancellazione di elementi in modo efficiente.
Un BST è composto da nodi, dove ogni nodo ha al massimo due figli:

- **Nodo sinistro** (left child)
- **Nodo destro** (right child)

Ogni nodo in un BST contiene tre componenti principali:

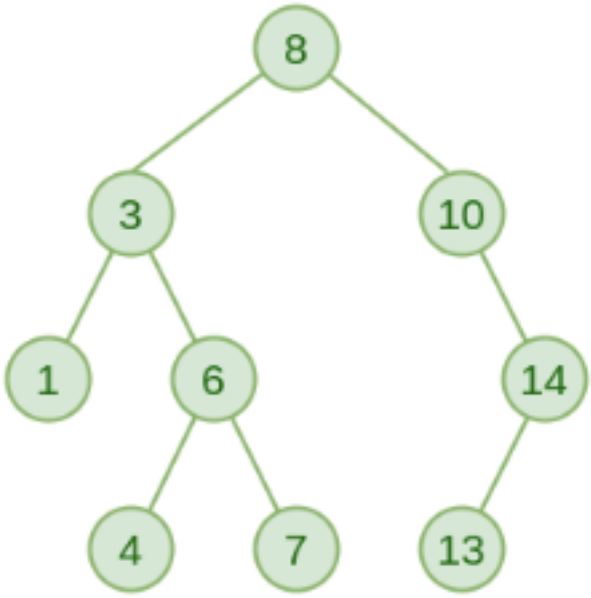
1. **Chiave/Valore**: Il dato memorizzato nel nodo.
2. **Puntatore al figlio sinistro**: Un riferimento a un altro nodo, o null se non esiste un figlio sinistro.
3. **Puntatore al figlio destro**: Un riferimento a un altro nodo, o null se non esiste un figlio destro.

Proprietà del BST
Il BST ha una proprietà di ordinamento che lo rende particolarmente utile per la ricerca:

- **Proprietà di ordinamento**: Per ogni nodo, tutti i valori contenuti nel sottoalbero sinistro sono minori o uguali alla chiave del nodo stesso, e tutti i valori contenuti nel sottoalbero destro sono maggiori o uguali alla chiave del nodo stesso.

Questa proprietà consente di effettuare ricerche binarie, il che rende il BST molto efficiente per la ricerca di elementi.

Implementazione



Search: Questa funzione mi permette di scorrere ricorsivamente l'albero per poter cercare l'elemento tramite la sua chiave. Se la chiave del nodo corrente è maggiore della chiave dell'elemento da cercare allora mi sposterò a sinistra altrimenti mi sposterò nel sottoalbero destro.

Insert: Questa funzione mi permette di inserire un nuovo nodo all'interno del mio albero. Anche questa funzione sfrutta il metodo ricorsivo. Il controllo che vado a fare è quello di verificare se il nodo corrente è vuoto. In caso affermativo allora creo il nodo nuovo, senno scorro ricorsivamente con la stessa logica usata nella search.

Delete: Potrebbe risultare la funzione più complessa in quanto ci sono diversi casi da considerare:

1) **Il nodo da eliminare è una foglia:**
   Elimino direttamente il nodo
2) **Il nodo da eliminare ha un solo figlio sinistro:**
   Rimuoviamo il nodo e colleghiamo il figlio del nodo eliminato al genitore del nodo stesso.
3) **Il nodo da eliminare ha due figli:**
   in questo caso devo utilizzare la funzione ausiliare deleteMin che mi permette di rimuovere l'elemento con chiave minore in un albero non vuoto e restituisce il dictionary Elem (ovvero la coppia (key, value)) corrispondente all'elemento rimosso.

Per una spiegazione piu dettagliata vi consiglio di guardare qui.

Grafi

Un **grafo** è una struttura dati utilizzata per rappresentare relazioni tra oggetti. Un grafo è composto da due insiemi principali:

- **Vertici (o nodi)**: Gli oggetti o i punti del grafo.
- **Archi (o spigoli)**: Le connessioni tra i vertici. Ogni arco può essere orientato (direzione specifica) o non orientato (nessuna direzione specifica).

Tipi di Grafi

1. **Grafi Non Orientati**: Gli archi non hanno una direzione. Se esiste un arco tra il vertice A e il vertice B, è possibile percorrerlo in entrambe le direzioni.
2. **Grafi Orientati (o Digrafi)**: Gli archi hanno una direzione specifica. Se esiste un arco da A a B, non implica automaticamente l'esistenza di un arco da B a A.

Implementazione

addVertex: Per aggiungere un vertice all'interno del mio grafo verifico inanziutto che il nodo non sia già presente.
Per farlo scorro tramite un ciclo while e verifico che la label dell'elemento corrrente sia diversa da quella dell'elemento da inserire.
Se non è già presente posso proseguire con un inserimento in coda come se si trattasse di una lista semplice.

addEdge: Aggiunge nuovo arco tra i due vertici con etichette le due stringe e peso l'intero. Fallisce se non sono presenti tutti e due i nodi o se l'arco tra i due e' gia' presente. Se fallisce ritorna false, altrimenti ritorna true.
La funzione inizia creando due puntatori, nodo1 e nodo2, inizialmente impostati su nullptr, che rappresentano i vertici di partenza e arrivo dell'arco da aggiungere. Utilizzando un puntatore cur, la funzione scorre la lista di vertici del grafo alla ricerca dei vertici con le etichette specificate dai parametri from e to. Se trova il vertice con l'etichetta from, assegna questo nodo a nodo1, e se trova il vertice con l'etichetta to, assegna questo nodo a nodo2.
Una volta che il ciclo termina, la funzione verifica se entrambi i vertici sono stati trovati; se uno dei due è nullptr, significa che uno o entrambi i vertici non esistono nel grafo e la funzione ritorna false. Se entrambi i vertici esistono, la funzione prosegue controllando se esiste già un arco tra nodo1 e nodo2. Scorrendo la lista di adiacenza di nodo1, verifica se c'è già un arco che punta a nodo2. Se trova un arco esistente, ritorna false per evitare duplicati.
Se non esiste già un arco, la funzione procede alla creazione di un nuovo arco. Per il grafo non orientato, crea due nodi di tipo halfEdgeVertex: il primo rappresenta l'arco da nodo1 a nodo2, e il secondo rappresenta l'arco da nodo2 a nodo1. Per l'arco da nodo1 a nodo2, viene creato un nuovo nodo arco1, dove si impostano l'etichetta, il puntatore al vertice di destinazione, e il peso dell'arco. Questo nuovo nodo viene inserito all'inizio della lista di adiacenza di nodo1. Successivamente, per mantenere la

simmetria dell'arco in un grafo non orientato, viene creato un secondo nodo arco2 con l'etichetta e il puntatore a nodo1, e questo viene inserito all'inizio della lista di adiacenza di nodo2.
Alla fine della funzione, se l'arco è stato aggiunto con successo a entrambe le liste di adiacenza, la funzione ritorna true, indicando che l'operazione è stata completata con successo.

numEdges: La funzione è progettata per calcolare il numero totale di archi in un grafo non orientato. La funzione inizia con un puntatore cur che scorre la lista dei vertici del grafo, inizializzato a “g”, che rappresenta la testa della lista di adiacenza. All'inizio, viene creato un contatore count inizializzato a zero.

La funzione entra in un ciclo while che continua finché cur non è nullo. All'interno di questo ciclo, un altro puntatore “arco” scorre la lista di adiacenza del vertice attuale cur. Per ogni arco trovato nella lista di adiacenza di cur, il contatore count viene incrementato di uno. Questo ciclo interno continua finché ci sono archi nella lista di adiacenza, spostandosi di nodo in nodo con il puntatore “arco”.

Una volta che tutti gli archi della lista di adiacenza del vertice corrente sono stati contati, il puntatore cur viene aggiornato per passare al vertice successivo nella lista di vertici del grafo.

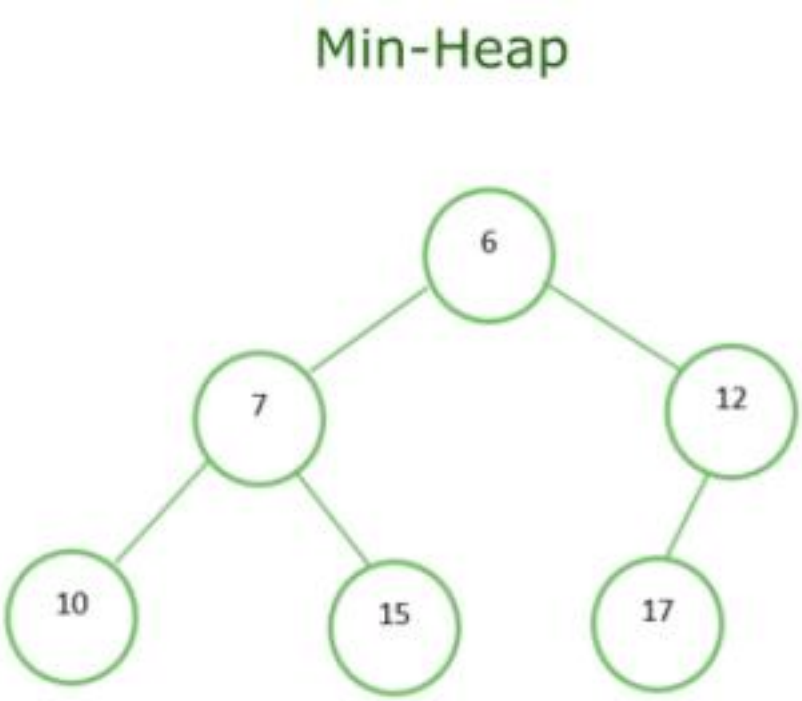
Dopo aver scansionato tutti i vertici e i loro archi, il contatore count contiene il numero totale di archi, ma questo valore include ogni arco due volte (una volta per ciascuna estremità dell'arco) a causa della natura non orientata del grafo. Per ottenere il numero corretto di archi distinti, il valore di count viene diviso per 2 prima di essere restituito.

Heap min

Un **heap minimo** è una struttura dati che rappresenta un albero binario completo in cui ogni nodo ha un valore inferiore o uguale ai valori dei suoi figli. Questa proprietà garantisce che il nodo con il valore minimo si trovi sempre alla radice dell'heap. Gli heap minimi sono utilizzati in vari algoritmi e applicazioni, tra cui gli algoritmi di ordinamento e le code di priorità.

Vi consiglio di guardare questo sito per capire il funzionamento di un heap min

Implementazione



moveDown: è utilizzata per ripristinare la proprietà dell'heap minimo dopo la rimozione dell'elemento minimo o dopo altre operazioni che possono violare l'ordinamento dell'heap. Partendo da un dato indice, calcola gli indici dei figli sinistro e destro. Confronta il valore dell'elemento corrente con i valori dei suoi figli e determina il figlio più piccolo. Se uno dei figli è più piccolo dell'elemento corrente, scambia l'elemento corrente con il figlio più piccolo e applica ricorsivamente moveDown al nuovo indice dell'elemento spostato. Questo processo continua finché l'elemento corrente non è minore o uguale ai suoi figli, mantenendo così la proprietà dell'heap minimo.

moveUp: ripristina le proprietà dell'heap dopo un'inserzione di un nuovo elemento.

Partendo da un dato indice, la funzione calcola l'indice del genitore. Se l'elemento corrente è più piccolo del suo genitore, scambia i due elementi e aggiorna l'indice dell'elemento corrente per essere il genitore. Questo processo viene ripetuto fino a quando l'elemento corrente è maggiore o uguale al suo genitore, garantendo che la struttura dell'heap rimanga valida.

CreateEmpty: La funzione createEmpty crea una nuova PriorityQueue di dimensione dim. Alloca dinamicamente un array per memorizzare gli elementi e inizializza la dimensione dell'heap a zero e la dimensione massima a dim. Restituisce la nuova struttura PriorityQueue, pronta per l'uso.

Insert: La funzione insert aggiunge un nuovo elemento alla PriorityQueue. Se l'heap è pieno (cioè la dimensione corrente è uguale alla dimensione massima), la funzione restituisce false. Altrimenti, inserisce l'elemento alla fine dell'array e incrementa la dimensione dell'heap. Dopo l'inserimento, chiama moveUp per assicurarsi che la proprietà dell'heap minimo sia mantenuta, spostando l'elemento verso l'alto se necessario. La funzione restituisce true se l'inserimento è avvenuto con successo.

FindMin: La funzione findMin trova l'elemento minimo dell'heap, che si trova sempre alla radice dell'heap (posizione 0 dell'array). Se l'heap è vuoto, la funzione restituisce false. Altrimenti, copia l'elemento radice in res e restituisce true.

deleteMin: La funzione deleteMin rimuove l'elemento minimo dell'heap, che si trova alla radice. Se l'heap è vuoto, la funzione restituisce false. Altrimenti, sostituisce la radice con l'ultimo elemento dell'heap, decrementa la dimensione dell'heap e chiama moveDown per ripristinare le proprietà dell'heap minimo. Questo processo assicura che l'elemento più piccolo venga ripristinato alla radice e che l'heap rimanga valido. La funzione restituisce true se la rimozione è avvenuta con successo.