List.h Quando devi gestire una collezione in modo dinamico,con inserimenti e rimozioni senza conoscere a priori la dimensione

link NEWnode(Item val, link next); crea un nuovo nodo con valore val e puntatore next (puntatore al nodo succ.)
void LISTinshead(LIST l, Item val); inserisce un nuovo elemento val all'inizio della lista l
link LISTinstail(link head, Item val); inserisce un nuovo nodo val in coda alla lista, restituisce la testa
link LISTinstail(link head, Item val);
ITEM LISTsearch(link head, Key k); restituisce l'elemento con chiave k, se presente | head testa della lista
link LISTdelhead(link h); elimina il primo nodo e restituisce la nuova testa | h testa della testa
ITEM LISTextrheadP(link *hp); estrae e restituisce l'elemento in testa, aggiornando la lista, | hp puntatore alla testa
link LISTdelkey(link h, Key k); elimina il nodo con chiave k e restituisce la nuova testa
link LISTdelkeyR(link x, Key k); versione ricorsiva della link LISTdelkeyR | x nodo corrente
ITEM LISTextrkeyP(link *x, Key k); estrae l'elemento con chiave k, aggiornando la lista | x puntatore alla testa
link LISTsortins(link h, ITEM item); inserisce l'elemento item mantenendo l'ordine della lista
ITEM LISTsortsearch(link h, Key k); cerca un elemento in una lista ordinata
link LISTsortdel(link h, Key k); cancella un nodo in una lista ordinata
void LISTshow(link h); stampa il contenuto della lista
void LISTfree(link h); libera la memoria occupata dalla lista

BST.h Memorizzare dati ordinabili chiave-valore, struttura dinamica e ordinata

BST BSTinit(); inizializza un albero vuoto
void BSTfree (BST bst); libera la memoria usata dall'albero bst
int BSTcount (BST bst); restituisce il numero di nodi del bst
int BSTempty (BST bst); restituisce 1 se albero vuoto, 0 altrimenti
Item BSTsearch(BST bst, Key k); cerca l'elemento con chiave k nell'albero
Item BSTmin(BST bst); restituisce l'item con chiave minima
Item BSTmax(BST bst); restituisce l'item con chiave massima
void BSTinsert_leafR(BST bst, Item x); inserisce x come foglia (ricorsivo)
void BSTinsert_leafI(BST bst, Item x); inserisce x come foglia (iterativo)
void BSTinsert_root(BST bst, Item x); inserisce x e lo porta alla radice con rotazioni
void BSTvisit (BST bst, int strategy); visita l'albero, con strategy : inorder,post ecc.
link rotR(link h); rotazione a destra attorno al nodo h radice locale
link rotL(link h); rotazione a sinistra attorno al nodo h radice locale
ext link partR (link h, int r); riporta in radice l'elemento con rango r
ext void BSTdelete (BST bst, Key k); elimina l'elemento con chiave k
ext Item BSTselect (BST bst, int r); restituisce l'elemento con rango r
ext Item BSTsucc (BST bst, Key k); restituisce il successore di k (valore più grande)
ext Item BSTpred (BST bst, Key k); restituisce il predecessore di k (valore più piccolo)

IBST.h Variante del BST in cui ogni nodo rappresenta un intervallo

void IBSTinit(IBST ibst); inizializza un albero di ibst vuoto
void IBSTfree (IBST ibst); libera la memoria dell'albero
void BSTinsert (IBST ibst, Item x); inserisce un nuovo intervallo item x
void IBSTdelete (IBST ibst, Item x); cancella un item x
Item IBSTsearch (IBST ibst, Item x); cerca se esiste un intervallo che si sovrappone a x
Int IBSTcount (IBST ibst); ritorna il numero di nodi dell'albero
int IBSTempty (IBST ibst); 1 se albero vuoto
void IBSTvisit (IBST ibst, int strategy); visita tuti i nodi con strategia

PQ.h Collezione di elementi in cui vogliamo estrarre l'elemento con la priorità più alta

PQ PQinit(int maxN); Crea una PQ vuota, maxN elementi
void PQfree (PQ pq); Libera memoria allocata alla PQ
int PQempty(PQ pq); 1 se coda vuota
void PQinsert(PQ pq, Item val); Inserisce un nuovo elemento nella coda mantenendo la priorità
Item PQextractMax(PQ pq); Rimuove e restituisce l'elemento massimo (con priorità più alta)
Item PQshowMax(PQ pq); Restituisce l'elemento massimo, senza rimuoverlo
void PQdisplay(PQ pq); Stampa (stdout) tutti gli elementi nella coda
int PQsize(PQ pq); Restituisce il numero di elementi presenti attualmente in coda
void PQchange(PQ pq, Item val); Cerca un elemento e aggiorna la sua priorità
void PQchange(PQ pq, int pos, Item val); Aggiorna la priorità di un elemento che si trova in una posizione nota (pos)

Heap.h Basata su un albero binario completo, permette di estrarre rapidamente max,min e inserire elementi in modo efficiente, si può usare per implementare una coda a priorità o per un algoritmo di ordinamento Heapsort

Heap HEAPinit(int maxN); crea, alloca in memoria e inizializza una struttura Heap vuota di maxN dimensione
void HEAPfill(Heap h, Item val); inserisce un nuovo elemento nell'heap
void HEAPsort(Heap h); esegue algoritmo di Heapsort sull'array interno dell'Heap
void HEAPdisplay(Heap h); stampa il contenuto dell'heap
void HEAPfree(Heap h); dealloca la memoria dell'heap
void HEAPify(Heap h, int i); ripristina le proprietà di heap facendo affondare l'elemento i
void HEAPbuild(Heap h); trasforma l'array disordinato in un heap valido chiamando heapify
int PARENT(int i); trova genitore di i
int RIGHT(int i); trova figlio destro di i
int LEFT(int i); trova figlio sinistro di i

ST.h Memorizzare e cercare elementi tramite una Key

ST STinit(int maxN); Inizializza ST vuota con dimensione massima maxN
void STdisplay(ST st); stampa l'intero contenuto della tabella
int STsize(int N); non sono sicuro
int STinsert(ST st, Item val); inserisce un nuovo elemento (1 se successo)
int STcount(ST st); restituisce il numero di elementi attualmente in tabella
int STempty(ST st); 1 se vuota
int STselect(ST st, int r); restituisce l'indice di rango r
int STcount (ST st);
void STinsert (ST st, Item val); inserisce elemento senza return
int STgetindex(ST tabella, ITEM item); cerca item e restituisce l'indice
Item STsearch(ST st, Key k); cerca l'elemento tramite chiave e restituisce l'item
int STsearch (ST st, Key k); cerca l'elemento tramite chiave, 1 se successo
Key STsearchByIndex (ST st, int id); trova l'elemento tramite l'indice e restituisce la key
void STdelete(ST st, Key k); rimuove un elemento dalla tabella tramite key
void STfree(ST st); dealloca spazio dello st
void STdisplay (ST st);
int hashU(char *v, int M); calcola l'hash per una key di tipo stringa *v, HELPER
int hash (Key k, int M); calcola l'hash della tabella per una data chiave | M dimensione tabella
int full((ST st, int i); controlla se la cella i della ST è occupata
void STchangePrio (ST st, Item val, int i); aggiorna priorità di un elemento all'indice i (per PQ indirette)

Queue.h Collezioni di elementi in cui il primo inserito è anche il primo a uscire (FIFO)

QUEUE QUEUEinit(int maxN); Inizializza con dimensione massima maxN elementi
int QUEUEempty(QUEUE q); 1 se vuota

void QUEUEput(QUEUE queue, Item val); Inserisce un nuovo elemento in fondo alla coda
Item QUEUEget(QUEUE q); Rimuove e restituisce l'elemento in testa alla coda

Graph.h Insieme di vertici e archi

Graph GRAPHinit(int V); crea grafo vuoto con V vertici
void GRAPHfree(Graph G); dealloca
Graph GRAPHload(FILE *fin); crea grafo da file
void GRAPHstore(Graph G, FILE *fout); salva la struttura del grafo su file fout
void GRAPHgetIndex(Graph G, char*label); cerca un vertice tramite la sua label
void GRAPHinsertE(Graph G, int id1, int id2, int wt); inserisci un nuovo arco tra vertici id1 e id2 di peso wt
void GRAPHremoveE(Graph G, int id1, int id2); rimuovi arco che collega id1 e id2
void GRAPHshow(Graph G); stampa il grafo
void GRAPHedges(Graph G, Edge *a); estrai tutti gli archi del grafo e mettili nell'array A
void insertE(Graph G, Edge e); inserisci un arco (versione che accetta la struttura Edge e)
void removeE(Graph G, Edge e); rimuovi arco con struct Edge
int randV(Graph G); restituisce l'indice di un vertice scelto casualmente
Graph GRAPHrand1(Graph G, int V, int E); crea un grafo casuale con V vertici ed E archi
Graph GRAPHrand2(Graph G, int V, int E); come prima ma in un altro modo
int GRAPHpath(Graph G, int id1, int id2); 1 se esiste almeno un cammino tra id1 e id2
void GRAPHpathH (Graph G, int id1, int id2); trova se esiste un cammino Hamiltoniano (tocca tutti i vertici una sola volta) tra id1 e id2
void GRAPHbfs(Graph G, int id); Visita in ampiezza Breadth first search partendo da id
void GRAPHdfs(Graph G, int id); Visita in profondità Depth first search a partire da id
int GRAPHscc(Graph G); Conta le componenti fortemente connesse
int GRAPHcc(Graph G); Conta le componenti connesse
Graph reverse(Graph G); Restituisce un nuovo grafo con archi invertiti (utile per Kosaraju)
void GRAPHmstK(Graph G); Calcola l'albero di Copertura minimo usando algoritmo di Kruskal
void GRAPHmstP(Graph G); Calcola l'albero di Copertura minimo usando algoritmo di Prim
void GRAPHspD(Graph G, int id); Calcola i cammini minimi da id usando Dijkstra (pesi non negativi)
void GRAPHspBF(Graph G, int id); Calcola i cammini minimi usando Bellman-Ford (pesi negativi)

Arco.h Struttura di supporto per il graph, NON è un ADT

Edge EDGEcreate(int v, int w, int wt); Crea una struttura edge usando due vertici e il loro peso
Edge EDGEalloc(int v, int w, int wt); Alloca la memoria necessaria per una Edge

IterativeSort.h Ordinamenti basati su Iterazione

void BubbleSort(Item A[], int N); A array da ordinare, N elementi | O(N^2)
void OptBubbleSort(Item A[], int N);
void SelectionSort(Item A[], int N);
void InsertionSort(Item A[], int N);
void ShellSort(Item A[], int N);
void CountingSort(Item A[], Item B[], int C[], int N, int k); A input, B output, C appoggio, N elementi, k valore max

RecursiveSort.h Ordinamenti basati su Ricorsione
void QuickSort (Item *A, int N); A array da ordinare, N elementi
void QuickSortR(Item *A, int l, int r); A array , l limite sinistro, r limite destro
int partition(Item *A, int l, int r); Funzione helper
void MergeSort (Item *A, int N);
void MergeSortR(Item *A, Item * B, int l, int r); B di appoggio
void Merge(Item *A, Item *B, int l, int q, int r); q centro
void BottomUpMergeSort (Item *A, int N);