Esercizi in C:

Esercizi sugli array:

1.Bandiera

// pre: 0 <= n == dim di B[]

// post: in B[0..n-1] i valori verde precedono i bianco ed i rosso, i bianco precedono i rosso

void Bandiera(Color B[], int n)

{

int i = 0;

int j = 0;

// inv. B[0..i-1] sono verde, B[i..j-1] bianco e B[j..k-1] rosso

for(int k = 0; k < n; k++) {

if(B[k] == bianco) {

swap(B, j, k);

j++;

}

else if(B[k] == verde) {

swap(B, j, k);

swap(B, i, j);

j++; i++;

} // altrimenti B[k] == rosso dunque basta incrementare k ad opera del for

}

}

2.BubbleSortCount

// pre: 0 <= n <= dimensione di A[]

// post: A[0..n-1] è ordinato in senso non decrescente;

// ritorna il numero di confronti

int countBubbleSort(int A[], int n) {

int count = 0;

bool sw = true;

for (int i = n - 1; i > 0 && sw; i--) {

// inv. A[i + 1..n - 1] è ordinato e

// per ogni x in A[0..i], y in A[i + 1..n - 1] si ha x <= y

sw = false;

for (int j = 0; j < i; j++) {

// inv. A[j] è il minimo in A[j..n - 1]

count = count + 1;

if(A[j] > A[j + 1]) {

swap(A, j, j + 1);

sw = true;

}

}

}

return count;

}

Sort:

1.Quicksort (PARTITION)

// pre: 0 <= p < r <= dimensionse di A[] e sia x = A[p]

// post: ritorna q dove A[p..q - 1] <= A[q] = x < A[q+1..r]

int Partition(int A[], int p, int r) {

int pivot = A[r];

int i = p-1;

for(int j = p; j<r;j++){

if(A[j] <= pivot){

i++;

swap(A,i,j);

}

}

swap(A,i+1,r);

return i+1;

}

2.Mergesort (merge)

// pre: l, m ordinate in senso non decrescente

// post: fonde distruttivamente l m in una lista ordinata

list merge(list l, list m) {

if (l == NULL) return m;

else if (m == NULL) return l;

else if (l->info <= m->info) {

l->next = merge(l->next, m);

return l;

}

else { // l->info > m->info

m->next = merge(l, m->next);

return m;

}

}

Esercizi sulle liste:

1.deleteAll:

// post: ritorna la lista as da cui sono state rimosse

// tutte le occorrenze di n

list deleteAll(int n, list as) {

if (as == NULL)

return NULL;

else if (as->info == n) {

list t = as->next;

free(as);

return deleteAll(n, t);

}

else {

as->next = deleteAll(n, as->next);

return as;

}

}

2.symmDiff:

// pre: xs e ys sono ordinate in senso crescente

// post: restituiesce una nuova lista degli el.

// in una ma non nell'altra delle due liste,

// ordinata in senso crescente

list symmDiff(list xs, list ys) {

if (xs == NULL)

return copyList(ys);

else if (ys == NULL)

return copyList(xs);

else if (xs->info == ys->info)

return symmDiff(xs->next, ys->next);

else if (xs->info < ys->info)

return Cons(xs->info, symmDiff(xs->next, ys));

else return Cons(ys->info, symmDiff(xs, ys->next));

}

3.insert:

// pre: 0 <= n < lunghezza di bs

// post: inserisce distruttivamente as in bs dopo n elementi, seguita

// dai rimanenti el. di bs

list insert(list as, list bs, int n) {

if (n == 0)

return concat(as, bs);

else {

bs->next = insert(as, bs->next, n - 1);

return bs;

}

}

4.pari/dispari:

recordList Foo(){

recordList record = {NULL, NULL, NULL};

return record;

}

void auxOddEven(list head, recordList \*record)

{

if (head == NULL)

{

return;

}

auxOddEven(head->next, record);

if (head->value % 2 == 1)

{

head->next = record->odd;

if (record->odd == NULL)

{

record->last = head;

}

record->odd = head;

}

else

{

head->next = record->even;

record->even = head;

}

return;

}

list oddEven(list head)

{

recordList record = Foo();

auxOddEven2(head, &record);

if (record.odd == NULL)

{

return record.even;

}

else

{

record.last->next = record.even;

return record.odd;

}

}

5.corank:

// post: ogni nodo di l contiene il suo co-rango

int corank(list l) {

if(l==NULL){

Return 0;

}else{

int sum=0;

while(l){

sum+=l->info;

l->info=sum;

l=l->next;

}

}

return sum;

}

6.fastreverse:

// post: inverte non distruttivamente l ritornando la lista invertita (int tempo lineare, non quadratico)

list fast\_reverse(list l){

if(l==NULL){

return NULL;

}

list reversed=NULL;

list current=l;

while(current){

reversed=Cons(current->info, reversed):

current=current->next;

}

return reversed;

}

Alberi k-ari:

1.Altezza:

// pre: t non è vuoto

// post: ritorna l'altezza di t

int height(kTree t) {

if(t->child == NULL)

return 0;

else {

int ht = 0;

kTree c = t->child;

while(c != NULL) {

ht = max(ht, height(c));

c = c->sibling;

}

return ht + 1;

}

}

2.Grado:

// pre: t non è vuoto

// post: ritorna il grado di t, ossia

// il massimo numero dei figli di un vertice in t

int degree(kTree t) {

if(t->child == NULL){

return 1;

}

else{

int m = 0;

kTree c = t->child;

while(c != NULL){

m = max(m,degree(c));

c = c->sibling;

}

return m+1;

}

}

3.SumLeaf:

// post: ritorna la somma delle etichette

// delle foglie di t

int sumLeaf (kTree t) {

if (t == NULL) return 0;

else if (t->child == NULL) // t è una foglia

return t->key;

else {

kTree c = t->child;

int sum = 0;

while (c != NULL) {

sum = sum + sumLeaf(c);

c = c->sibling;

}

return sum;

}

}

4.Bfs:

// post: ritorna la lista delle etichette (chiavi) di t visitato in ampiezza

list kTreeBFS(kTree t) {

if (t == NULL)

return NULL;

list l = NULL;

queue q = NewQueue();

EnQueue(t, q);

while(!isEmptyQueue(q)) {

kTree node = DeQueue(q);

l = Cons(node->key, l); // visita

// l = Add(node->key, l); // versione con Add

node = node->child;

while (node != NULL) {

EnQueue(node, q);

node = node->sibling;

}

}

return reverse(l);

// return l; // versione con Add

}

5.Massima Somma Etichette Ramo:

// post: ritorna la più grande somma delle etichette tra i rami

int maxSumBranch(kTree t){

if(t==NULL){

return 0;

}

if(t->child==NULL){

return t->key;

}else{

int maxSum=maxSumBranch(t->child);

kTree current=t->child->sibling;

while(current){

maxSum=max(maxSum,maxSumBranch(current));

current=current->sibling;

}

return t->key+maxSum;

}

}

6.Etichette dispari:

// pre: albero etichettato con interi

//post: ritorna true se tutte le etichette sono dispari, false altrimenti

bool isOdd(kTree t){

if(t==NULL){

return false;

}

if(t->child==NULL){

if(t->key%2!=0){ //dispari

return true;

}else{ //pari

return false;

}

}

bool odd=true;

kTree current=t->child;

while(current){

odd=odd&&isOdd(current);

current=current->sibling;

}

return odd;

}

Alberi ordinati:

1.isOrdered:

// pre: bt != NULL

// post: restituisce la tripla (min, isOrdered, max)

// dove isOrdered == true se bt e' ordinato, false altrimenti

// se isOrdered == true allora min e max sono il minimo ed il massimo

// delle chiavi in bt

Triple isOrderedBTreeAux(btree bt)

{

Triple t;

if(isLeaf(bt)) // true se bt->left == NULL && bt->right == NULL

{

t.isOrdered = true;

t.min = t.max = bt->key;

}

else if(bt->right == NULL) // quindi bt->left != NULL

{

Triple tleft = isOrderedBTreeAux(bt->left);

t.isOrdered = tleft.isOrdered && bt->key > tleft.max;

t.max = bt->key;

t.min = tleft.min;

}

else if(bt->left == NULL) // quindi bt->right != NULL

{

Triple tright = isOrderedBTreeAux(bt->right);

t.isOrdered = tright.isOrdered && bt->key < tright.min;

t.max = tright.max;

t.min = bt->key;

}

else // bt->left != NULL && bt->right != NULL

{

Triple tleft = isOrderedBTreeAux(bt->left);

Triple tright = isOrderedBTreeAux(bt->right);

t.isOrdered = tleft.isOrdered &&

tright.isOrdered &&

bt->key > tleft.max && bt->key < tright.min;

t.min = tleft.min;

t.max = tright.max;

}

return t;

}

// post: decide se bt e' ordinato (di ricerca)

bool isOrdered(btree bt)

{

if (bt == NULL)

return true;

else

{

Triple t = isOrderedBTreeAux(bt);

return t.isOrdered;

}

}

2. DescList:

// pre: bt albero bunario di ricerca

// post: ritorna la lista delle chiavi in bt ordinate in senso decrescente

// davanti ad l

list DescList\_aux(btree bt, list l) {

if (bt == NULL)

return l;

else {

l = DescList\_aux(bt->left, l);

l = Cons(bt->key, l);

return DescList\_aux(bt->right, l);

}

}

// pre: bt albero bunario di ricerca

// post: ritorna la lista delle chiavi in bt ordinate in senso decrescente

list DescList(btree bt) {

return DescList\_aux(bt, NULL);

}

3.Successor:

// pre: bt != NULL è un albero binario di ricerca

// post: ritorna il puntatore al nodo con chiave minima in bt

btree minInBtree(btree bt)

{

if(bt->left == NULL)

return bt;

else

return minInBtree(bt->left);

}

// pre: nd != NULL è un nodo di un albero binario di ricerca

// post: ritorna l'avo destro più prossimo a nd se esiste; NULL altrimenti

btree rightAncestor(btree nd)

{

if(nd->parent == NULL)

return NULL;

else if(nd == nd->parent->left)

return nd->parent;

else // nd == nd->parent->right

return rightAncestor(nd->parent);

}

// pre: nd != NULL punta ad un nodo di un albero binario di ricerca

// post: ritorna il puntatore al predecessore di nd se esiste

// NULL altrimenti

btree successor(btree nd)

{

if(nd->right != NULL)

return minInBtree(nd->right);

else

return rightAncestor(nd);

}

Heap:

1.maxHeapify:

HEAPIFYMAX(array hd, int i, int d){

int max = i;

int lefty = leftchild (i,hd);

int righty = rightchild (i,hd);

if (i <= d && hd[i] < hd[lefty]){

max = lefty;

}

if (i <= d && hd[i] < hd[righty]){

max = righty;

}

if(i!=max){

swap(hd,i,max);

HEAPIFYMAX(hd,max,d);

}

}

1.heapiSort:

HEAPSORT(array hd, int d){

BUILDHEAP(hd);

for (int i = d-1; i>=0,; i--){

swap(hd[0],hd[i]);

d--;

HEAPIFYMAX(hd,0,d);}

}

1.buildHeap:

BUILDHEAP(array hd){

for (int i = (hd->size/2)-1 ; i>=0; i--){

HEAPIFYMAX(hd,i,hd->dim);

}

}

1.increaseKey:

void increase\_key(Heap H, int key, int i){

if(key < H->arr[i]) error();

H->arr[i] = key;

While(i>0 && H->arr[parent(H,i)] < H->arr[i]{

Swap(H->arr,I,parent(H,i));

I = parent(H,i);

}

}

1.extractMax: