

# **RACCOLTA PROCEDURE E FUNZIONI PER L'ESAME PRATICO DI ALGORITMICA**

A cura di Salvo Firera

## **INDICE**

<b>Sezione 1: Array</b> .....	<b>pag.2</b>
<b>Sezione 2: Stringhe</b> .....	<b>pag.4</b>
<b>Sezione 3: Algoritmi di sorting</b> .....	<b>pag.5</b>
<b>Sezione 4: Liste</b> .....	<b>pag.8</b>
<b>Sezione 5: Hashing</b> .....	<b>pag.16</b>
<b>Sezione 6: Grafi</b> .....	<b>pag.27</b>
<b>Sezione 7: Alberi</b> .....	<b>pag.37</b>
<b>Sezione 8: Esercizi Svolti</b> .....	<b>pag.50</b>
<b>Sezione 9: Utility e Promemoria</b> .....	<b>pag.57</b>

## 1.ARRAY

### Inversione array in place:

```
1. void revArray (int a[],int dim){
2.     for (i=0;i<dim/2;i++){
3.         tmp=a[i];
4.         a[i]=a[dim-i-1];
5.         a[dim-i-1]=tmp;}
6. }
```

### Array con allocazione dinamica:

```
1. // L'UTENTE MI DA LA DIMENSIONE DA INPUT
2.
3. scanf("%d",&n);
4. int *a=malloc(sizeof(int)); //alloco dinamicamente la memoria che mi serve
5. for (i=0;i<n;i++){
6.     scanf("%d",&arr[i]);
7. }
8.
9. // NON CONOSCO MA DIMENSIONE PRECISA, QUINDI ALLOCO UN TETTO MASSIMO,RIEMPIO QUELLO CHE SERVE E SCRIVO
10. // LA DIMENSIONE SU UN PUNTATORE RITORNATO O PASSATO COME PUNTATORE
11.
12. void readArray (int* arr,int *length){
13.     int end=0; // serve per interrompere il loop di acquisizione
14.     int i=0, val;
15.     while (!end){
16.         scanf("%d",&val);
17.         if (val>=0){
18.             arr[i]=val; // esempio di terminazione appena si incontra un valore negativo
19.             i++;
20.         }
21.         else end++;
22.     }
23.     *length=i; // scrivo la dimensione sul puntatore passato
24. }
25.
26. // nel blocco main dichiarero un array allocando uno spazio massimo (es.100),il puntatore con la
27. // lunghezza mi dirà fin dove è riempito
28.
29. int *arr= malloc(100*sizeof(int)); //poi lo passo come parametro alla funzione di lettura
```

## Sottosequenza di somma massima:

```
1. int max_subsum(int a[],int len){
2.     int i=0;
3.     int sum=0;
4.     int max=0;
5.     while (i<len ){
6.         if (a[i]<0){
7.             sum=sum+a[i];
8.             if (sum<0)      // se la soma diventa negativa posso azzerarla
9.                 sum=0;
10.        }
11.        else{
12.            sum=sum+a[i];
13.            if (sum>max)
14.                max=sum;
15.        }
16.        i++;
17.    }
18.    return max;
19. }
```

## Intersezione Array ordinati (modo efficiente)

```
1. int inters_ord(int a[],int len_a,int b[],int len_b){
2.     int i=0;
3.     int j=0;
4.     int count=0;
5.     while (i<len_a && j<len_b){    // scorro gli array contemporaneamente
6.         if (a[i] < b[j])
7.             i++;
8.         else if (a[i]>b[j])
9.             j++;
10.        else{
11.            i++;
12.            j++;
13.            count++;
14.        }
15.    }
16.    return count;
17. }
```

## 2.STRINGHE

### Creazione stringa

```
1. // Viene gestita come un array di caratteri
2.
3. char* str=malloc(MAXLEN*sizeof(char));
4. scanf ("%s",str);
5.
6. printf ("%s\n",str);
```

### Comparazione stringhe

```
1. // Uso la funzione strcmp nella libreria string.h
2.
3. return strcmp(str1,str2);
4.
5. // restituisce 0 se le due stringhe sono uguali
6. //          1 se la prima è lessicograficamente maggiore della seconda
7. //          -1 se la seconda '' '' '' prima
```

### Array di stringhe

```
1. // Gestito come array di array (matrice)
2.
3. char** legge(int *len) {
4.     int i;
5.     scanf("%d", len);
6.     char **a = malloc(*len * sizeof(char*));
7.     for( i = 0; i < *len; i++ ){
8.         *(a+i)=malloc(MAX_LEN*sizeof(char)); //per ogni stringa alloco la lunghezza massima (definita)
9.         scanf("%s", *(a+i));
10.    }
11.    return a;
12. }
13.
14. void main(){
15.     int len;
16.     char** sequenza=malloc(n*sizeof(char*)); //alloco spazio per n stringhe
17.     sequenza=legge(&len);
18. }
```

### 3.ALGORITMI DI SORTING

#### Insertion Sort ( $O(n^2)$ )

```
1. void insertion_sort (int *a,int dim){
2.     int i;
3.     int j;
4.     int tmp;
5.     for (i=1;i<dim;i++){
6.         tmp=*(a+i);
7.         j=i-1;
8.         while (*(a+j) > tmp && j>=0){
9.             *(a+j+1)=*(a+j);
10.            j--;
11.        }
12.        *(a+j+1)=tmp;
13.    }
14. }
```

#### Selection Sort ( $O(n^2)$ )

```
1. void selection_sort (int *a,int dim){
2.     int i;
3.     int j;
4.     int tmp;
5.     for (i=0;i<dim-1;i++){
6.         for (j=i+1;j<dim;j++){
7.             if (*(a+j) < *(a+i)){
8.                 tmp=*(a+i);
9.                 *(a+i)=*(a+j);
10.                *(a+j)=tmp;
11.            }
12.        }
13.    }
14. }
```

## Merge Sort ( $\Theta(n \log n)$ )

```
void mergeSort(int a[], int p, int r) {
    int q;
    if (p < r) {
        q = (p+r)/2;
        mergeSort(a, p, q);
        mergeSort(a, q+1, r);
        merge(a, p, q, r);
    }
    return;
}
```

---

```
1. void merge(int a[], int p, int q, int r) {
2.     int i, j, k=0, b[max];
3.     i = p;
4.     j = q+1;
5.     while (i<=q && j<=r) {
6.         if (a[i]<a[j]) {
7.             b[k] = a[i];
8.             i++;
9.         } else {
10.            b[k] = a[j];
11.            j++;
12.        }
13.        k++;
14.    }
15.    while (i <= q) {
16.        b[k] = a[i];
17.        i++;
18.        k++;
19.    }
20.    while (j <= r) {
21.        b[k] = a[j];
22.        j++;
23.        k++;
24.    }
25.    for (k=p; k<=r; k++)
26.        a[k] = b[k-p];
27.    return;
28. }
```

## QuickSort ( $\Theta(n \log n)$ caso medio, $O(n^2)$ caso pessimo )

```
1. // La procedura quicksort è già presente nella libreria stdlib.h
2. // in base al tipo di dati da ordinare dobbiamo soltanto creare una funzione che dia
3. // alla procedura un criterio di ordinamento
4.
5. // per INTERI
6.
7. int int_compare (const void*a,const void *b){
8.     int v1= *((int*)a);
9.     int v2= *((int*)b);
10.    return v1 - v2;
11. }
12.
13. qsort (arr,dim,sizeof(int),int_compare);
14.
15. // per STRINGHE
16.
17. int compare_string (const void*a,const void *b){
18.     char* st1=*(char**)a;
19.     char* st2=*(char**)b;
20.    return strcmp(st1,st2);
21. }
22.
23. // per STRUCT
24.
25. int struct_compare (const void *a,const void *b){
26.     string_el aa= *(string_el*)a;    // in questo caso ordina le stringhe in ordine di lunghezza
27.     string_el bb= *(string_el*)b;    // nel caso fossero uguali le ordina in ordine alfabetico
28.     if (aa.len != bb.len){
29.         return aa.len-bb.len;}
30.     else{
31.         return strcmp(aa.str,bb.str);}
32. }
```

## 4.LISTE CONCATENATE

### Inserimento in testa – PushHead ( $\Theta(1)$ )

```
1. void pushHD (node **head,int valore){
2.     *node new=malloc(sizeof(node)); // creazione e allocazione del nuovo nodo
3.     node->key=valore;
4.     if (*head==NULL){
5.         new->next=NULL;
6.         *head=new;
7.     }
8.     else{
9.         new->next=*head;
10.        *head=new;
11.    }
12. }
```

### Inserimento in coda – PushTail ( $\Theta(n)$ )

```
1. void pushTail (node **head,int valore){
2.     node *new=malloc(sizeof(node));
3.     new->key=valore;
4.     new->next=NULL;
5.     if (*head==NULL)
6.         *head=new;
7.     else{
8.         *node cur=*head;
9.         while (cur->next!=NULL){
10.             cur=cur->next;
11.         }
12.         cur->next=new;
13.     }
14. }
```



## Inserimento ordinato – Push Order ( $O(n)$ )

```
1. void add_ordered (node **head,int valore){
2.     node new=malloc(sizeof(Node));
3.     new->key=valore;
4.     if (*head==NULL){
5.         new->next=NULL;
6.         *head=new;
7.     }
8.     else{
9.         node *cur=*head;
10.        node *prev=NULL;
11.        while (cur != NULL && cur->value < v){
12.            prev=cur;
13.            cur=cur->next;
14.        }
15.        if (prev == NULL){
16.            new->next=*head;
17.            *head=new;}
18.        }
19.        else{
20.            prev->next=new;
21.            new->next=cur;
22.        }
23.    }
24. }
```

## Cancella Duplicati (lista ordinata)

```
1. void deleteDup (node **head){
2.     if (*head==NULL || (*head)->next==NULL)
3.         return;
4.     if ( (*head)->key == (*head)->next->key ){
5.         node *tmp=*head;
6.         *head= (*head)->next;
7.         free(tmp);
8.         deleteDup ( &(*head))
9.     }
10.    deleteDup ( &(*head)->next );
11. }
```

## Filtrare lista (secondo un qualsiasi criterio)

```
// VERSIONE RICORSIVA

void filter (node **head){
    if (head==NULL) return;
    if (head->key == **CONDIZIONE**){
        node *tmp=*head;
        *head= *head->next;
        filter( &(*head) );
    }
    filter( &(*head)->next );
}
```

## versione iterativa

```
15. void filter (node **head){
16.     if (*head==NULL) return;
17.     node *prev=NULL;
18.     node *x=*head;
19.     node *fw=x->next;
20.     while (fw!=NULL){
21.         if (x->key == **CONDIZIONE**){
22.             node *tmp=x;
23.             if (prev==NULL){
24.                 *head=*head->next;
25.                 x=*head;
26.                 free(tmp);
27.             }
28.             prev->next=x->next;
29.             x=fw;
30.             fw=fw->next;
31.             free(tmp);
32.         }
33.         else{
34.             prev=cur;
35.             cur=fw;
36.             fw=fw->next;
37.         }
38.     }
39.     if (x->key == **CONDIZIONE**){ // esamino l'ultimo elemento separatamente
40.         if (prev==NULL){
41.             *head=NULL;
42.             free(x);
43.         }
44.         else{
45.             prev->next=NULL;
46.             free(x);
47.         }
48.     }
```

## Invertire una lista (senza usarne una d'appoggio)

```
1. void inverti_lista(Listadiementi* list){
2.     Listadiementi aux=*list;
3.     Listadiementi aux2;
4.     *list=NULL;
5.     while (aux!=NULL){           //finche' ci sono elementi nella lista R
6.         aux2=aux;                //prendi il primo elemento di R,
7.         aux=aux->next;            //sgancialo da R e
8.         aux2->next=*list;         //mettilo in testa alla lista invertita.
9.         *list=aux2;
10.    }
11. }
```

## Stampare lista

```
1. void printList (node *head){
2.     if (head==NULL) return;
3.     printf("%d ",head->key);
4.     printList(head->next);
5. }
```

## Lista versione con puntatore alla coda (inserimento e cancellazione coda in tempo costante)

```
struct node {
    int key;
    struct node* next;
};
```

```
typedef struct node node;
```

```
struct list {
    node* head;
    node* tail;
    int size;
};
```

```
typedef struct list list;
```

```
list* newList() {
    list* lst = malloc(sizeof(list));
```

```

    lst->head = NULL;
    lst->tail = NULL;
    lst->size = 0;
    return lst;
}

void destroyList(list* lst) {
    while(lst->head != NULL) {
        node* tmp = lst->head; // Salva l'elemento corrente
        lst->head = lst->head->next; // Avanza nella lista
        free(tmp); // Dealloca l'elemento
    }
    free(lst); // Free della struct che conteneva la lista
}

void pushHead (list *lst,int el){
    node *new= malloc(sizeof(node));
    new->key=el;
    if (lst->size==0) {
        lst->head=new;
        new->next=NULL;
        lst->tail=lst->head;
        lst->size++;
        return;
    }
    new->next=lst->head;
    lst->head=new;
    lst->size++;
}

void pushTail(list *lst,int el){
    node *new=malloc(sizeof(node));
    new->next=NULL;
    new->key=el;
    if (lst->size==0){
        lst->head=new;
        new->next=NULL;
        lst->tail=lst->head;
        lst->size++;
        return;
    }
    lst->tail->next=new;
    lst->tail=new;
    lst->size++;
}

void dropHead(list *lst){
    if (lst->size==1){
        free(lst->head);
        lst->head=NULL;
    }
}

```

```

    lst->tail=NULL;
    lst->size--;
    return;
}
if (lst->size>1){
    node *tmp= lst->head;
    lst->head=lst->head->next;
    free(tmp);
    lst->size--;
}
}

```

```

void dropTail(list *lst){
    if (lst->size==1){
        free(lst->tail);
        lst->head=NULL;
        lst->tail=NULL;
        lst->size--;
        return;
    }
    if (lst->size>1){
        node* cur=lst->head;
        node* prev=NULL;
        while (cur->next!=NULL) {
            prev=cur;
            cur=cur->next;
        }
        free(cur);
        prev->next=NULL;
        lst->tail=prev;
        lst->size--;
    }
}

```

```

void delNode(list *lst,int el){
    node *cur=lst->head;
    node *prev=NULL;
    while (cur!=NULL && cur->key!=el){
        prev=cur;
        cur=cur->next;
    }
    if (prev==NULL){
        dropHead(lst);
        return;
    }
    if (cur!=NULL && cur->next==NULL){
        dropTail(lst);
        return;
    }
    if (cur!=NULL){

```

```

    prev->next=cur->next;
    lst->size--;
    free(cur);
}
}

int member(list *lst,int el){
    node *cur=lst->head;
    while (cur!=NULL && cur->key!=el){
        cur=cur->next;
    }
    if (cur!=NULL) return 1;
    return 0;
}

void printList(list *lst){
    node* curr = lst->head;
    printf("Elementi presenti: %d\n",lst->size);
    printf("Contenuto della lista:\n");
    while(curr != NULL) {
        printf("%d -> ", curr->key);
        curr = curr->next;
    }
    printf("NULL\n");
}

void main(){
    list *l1= newList();

```

## **Filtraggio con lista 2.0 (condizione: elementi minori della media)**

```

void filterAvg(list *lst,int avg){
    node *cur=lst->head;
    node *prev=NULL;
    while (cur!=NULL){
        if (cur->key <= avg){
            if (prev==NULL){
                dropHead(lst);
                cur=lst->head;
            }
            else if (cur!=NULL && cur->next==NULL){
                dropTail(lst);
                return;
            }
            else if (cur!=NULL){
                node *tmp=cur;
                prev->next= cur->next;

```

```

    cur=prev->next;
    free(tmp);
}
}
else{
    prev=cur;
    if (cur!=NULL)
        cur=cur->next;
}
}
}

```

## Somme suffisse (ogni elemento è la somma dei successivi)

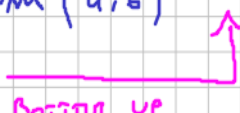
```

void suffixSum(node *head, int* s){
    int f;
    if(head==NULL) return;
    *s= *s + head->key;
    f= *s - head->key;
    suffixSum(head->next, s);
    head->key= *s - f - head->key;
}

```

**SPIEGAZIONE SOMME SUFFISSE:**

INPUT: 1 → 2 → 3 → 4

1 <sup>a</sup> CHIAMATA:	$S = \emptyset$	$h = 1$	$\text{suffixSum}(1, 0)$	$10 - 1 - 0 = 9$	} OUTPUT
2 <sup>a</sup> CHIAMATA:	$S = 1$	$h = \emptyset$	$\text{suffixSum}(2, 1)$	$10 - 1 - 2 = 7$	
3 <sup>a</sup> CHIAMATA:	$S = 3$	$h = 1$	$\text{suffixSum}(3, 3)$	$10 - 3 - 3 = 4$	
4 <sup>a</sup> CHIAMATA:	$S = 6$	$h = 3$	$\text{suffixSum}(4, 6)$	$10 - 6 - 4 = 0$	
	$S = 10$	$h = 6$			

**/\* prima calcolo la somma di tutti, poi per ottenere l'elemento in una posizione, sottraggo alla somma l'elemento stesso e la somma dei precedenti \*/**

## 5.HASHING

### Inserimento con chaining (liste di trabocco)

```
1. void hash_insert (node **tab,int n,int x){ // x valore da inserire
2.     int i
3.     int hx = (x*c) % (2*n); // calcolo la funzione hash che mi restituisce la soluzione
4.     push_hd(&tab[hx],x); // inserisco nella lista puntata dalla posizione
5. }
6.
7. // stesso procedimento per la ricerca, calcolo la funzione per individuare la lista, e cerco nella lista
8. // con una normale ricerca su lista
```

---

### Inizializzazione nel main

```
1. node **tab=malloc(2*n*sizeof(node));
2. for (i=0;i<2*n;i++){
3.     tab[i]=NULL}
```

---

### Esempio Rubrica telefonica con Hashing

```
struct contatto{
    char* nome;
    char* numero;};
typedef struct contatto contatto;

struct node{
    contatto record;
    struct node* next;};
typedef struct node node;

int fHash (char* str, int n){
    int i, sum=0;
    for (i=0;i<strlen(str);i++){
        sum= (unsigned int)str[i] + sum;
    }
    return sum % (2*n);
}

void insert (node **tab, int n, char* name, char* num){
    contatto new;
    new.nome=name;
    new.numero=num;
    node *newNode= malloc(sizeof(node));
    newNode->record= new;
```



```

int pos = fHash(name,n);
node *cur= tab[pos];
node *prev= NULL;
if (tab[pos]==NULL){
    tab[pos]=newNode;
    tab[pos]->next=NULL;
    return;
}
while (cur!=NULL && strcmp(cur->record.name,name)<0){
    prev= cur;
    cur= cur->next;
}
if (prev==NULL){
    newNode->next= tab[pos];
    tab[pos]=newNode;
    return;
}
prev->next=newNode;
newNode->next=cur;
}

void printList(node *head){
    if (head==NULL) return;
    printf("%s %s\n",head->record.name,head->record.numero);
    printList(head->next);
}

void main(){
    int n, i, k;
    scanf("%d",&n);
    node **tab= malloc(2*n*sizeof(node));
    for (i=0;i<(2*n);i++){
        tab[i]=NULL;
    }
    for (i=0;i<n;i++){
        char* nom= malloc(MAXNOME*sizeof(char));
        char* num= malloc(MAXNUM*sizeof(char));
        scanf("%s",nom);
        scanf("%s",num);
        insert(tab,n,nom,num);
    }
    scanf("%d",&k);
    printList(tab[k]);
}

```

## Esempio 15 oggetti Alice&Bob (problema dello zaino)

```
#define MAXLEN 101
```

```
// oggetto formato da nome e valore
```

```
struct item {  
    char* nome;  
    int val;};
```

```
typedef struct item item;
```

```
// lista di oggetti
```

```
struct node {  
    item key;  
    struct node* next;};
```

```
typedef struct node node;
```

```
void push (node **head, char *nome, int val) {  
    node *new = malloc(sizeof(node));  
    new->key.nome = nome;  
    new->key.val = val;  
    if (*head == NULL) {  
        new->next = NULL;  
        *head = new;  
    }  
    else {  
        new->next = *head;  
        *head = new;  
    }  
}
```

```
// calcola la funzione hash e inserisce
```

```
void hash_insert (node **tab, int n, char *nome, int val) {  
    int hx, str_val = 0, i;  
    for (i = 0; i < strlen(nome); i++) {  
        str_val = str_val + (unsigned int)nome[i];  
    }  
    hx = str_val % (2*n);  
    node *cur = tab[hx];  
    while (cur != NULL) {  
        if (!strcmp(cur->key.nome, nome)) {  
            if (cur->key.val < val) {  
                cur->key.val = val;  
                return;  
            }  
            else  
                return;  
        }  
        else {  
            return;  
        }  
    }  
}
```

```

        cur=cur->next;
    }
}
push(&(tab[hx]),nome,val);
}

```

// trasforma la tabella in un array per poter ordinare

```

void tableToArr (node **tab,int n,item *arr,int *len){
    int i, j=0;
    for (i=0;i< 2*n;i++){
        node *cur=tab[i];
        while (cur!=NULL){
            arr[j]=cur->key;
            j++;
            cur=cur->next;
        }
    }
    *len=j;
}

```

// funzione ausiliaria per qsort

```

int struct_compare (const void *a,const void *b){
    item it_a= *(item*)a;
    item it_b= *(item*)b;
    if ( it_a.val == it_b.val)
        return strcmp(it_a.nome,it_b.nome);
    return it_b.val - it_a.val;
}

```

```

int main(){
    int i, n, valore, len;
    scanf("%d",&n);
    item *arr=malloc(n*sizeof(item));
    node **tab=malloc(2*n*sizeof(node));
    for (i=0;i< 2*n;i++){
        tab[i]=NULL;
    }
    for(i=0;i<n;i++){
        char *name=malloc(MAXLEN*sizeof(char));
        scanf("%s",name);
        scanf("%d",&valore);
        hash_insert(tab,n,name,valore);
    }
    tableToArr(tab,n,arr,&len);
    qsort(arr,len,sizeof(item),struct_compare);
    if (len > 15){
        for (i=0;i<15;i++){
            printf("%s\n",arr[i].nome);
        }
    }
}

```

```

else{
    for (i=0;i<len;i++){
        printf("%s\n",arr[i].nome);
    }
}
return 0;
}

```

## Esami studente (appello 10set19) Hashing

```
#define HASH_CONS 941
```

```

struct node{
    char mat;
    int fail;
    struct node *next;};

```

```
typedef struct node node;
```

```

void push_head (node **head,int mat){
    node *new=malloc(sizeof(node));
    new->mat=mat;
    new->fail=0;
    if (*head==NULL){
        *head=new;
        (*head)->next=NULL;
    }
    else{
        new->next=*head;
        *head=new;
    }
}

```

```

void hash_insert (node **tab,int n,int mat){
    int i;
    int hx = (mat*HASH_CONS)%(2*n);
    push_head(&tab[hx],mat);
}

```

```

void hash_aggiorna (node **tab,int n,int mat,int *conta){
    int hx = (mat*HASH_CONS)%(2*n);
    node *cur=tab[hx];
    node *prev=NULL;
    while (cur!=NULL){
        if (cur->mat==mat){
            cur->fail++;
            if (cur->fail==2){
                *conta=*conta-1;
                if (prev==NULL){
                    node *tmp=cur;

```

```

        tab[hx]=tab[hx]->next;
        cur=tab[hx];
        free(tmp);
        return;
    }
    else{
        node *tmp=cur;
        prev->next=cur->next;
        free(tmp);
        return;
    }
}
return;
}
else{
    prev=cur;
    cur=cur->next;
}
}
}
}

```

**/\* VERSIONE RICORSIVA:**

```

void hash_aggiorna (node *tab,int mat,int *conta){
    if (tab==NULL)return;
    if (tab->mat==mat && tab->fail==1){
        *conta=*conta-1;
        node *tmp=tab;
        tab=tab->next;
        free(tmp);
        return;
    }
    if (tab->mat==mat && tab->fail==0){
        tab->fail++;
        return;
    }
    hash_aggiorna(tab->next,mat,conta);
} */

```

```

int main(){
    int n,i,m1,m2,mat,conta;
    scanf("%d",&n);
    conta=n;
    node **tab=malloc(2*n*sizeof(node));
    for (i=0;i<2*n;i++){
        tab[i]=NULL;
    }
    for (i=0;i<n;i++){
        scanf("%d",&mat);
        hash_insert(tab,n,mat);
    }
}

```

```

scanf("%d",&m1);
for (i=0;i<m1;i++){
    scanf("%d",&mat);
    hash_aggiorna(tab,n,mat,&conta);
}
scanf("%d",&m2);
for (i=0;i<m2;i++){
    scanf("%d",&mat);
    hash_aggiorna(tab,n,mat,&conta);
}
printf("%d\n",conta);
return 0;
}

```

## Esempio K-occorrenze con Hash

```

struct node {
    int key;
    int freq;
    struct node* next;
};
typedef struct node node;

struct table {
    node** table; /* Tabella Hash (array di liste) */
    int m; /* Dimensione della tabella */
};
typedef struct table table;

int hashF(int x,int n){
    return x%(2*n);
}

table* newTable(int m) {
    int i;
    table* T = malloc(sizeof(table));
    T->table = malloc(m * sizeof(node*));
    T->m = m;
    for(i=0;i<m;i++) {
        T->table[i] = NULL;
    }
    return T;
}

void hash_destroy(table* T) {
    int i;
    int m = T->m;
    for(i=0;i<m;i++) {
        node* lista = T->table[i];
        while(lista != NULL) {

```

```

        node* tmp = lista;
        lista = lista->next;
        free(tmp);
    }
}
free(T->table);
free(T);
}

void insert(table* T, int k, int fr) {
    /* Calcolo dell'hash */
    int posizione = hashF(k, T->m);
    node* new = malloc(sizeof(node));
    new->key = k;
    new->freq = fr;
    new->next = T->table[posizione]; // NULL se la lista era vuota
    T->table[posizione] = new; // Inserimento in testa
}

int search(table* T, int k, int* result) {
    node* lista = T->table[hashF(k, T->m)];
    while(lista != NULL && lista->key != k)
        lista = lista->next;
    if(lista == NULL) { // Non trovato
        result = NULL;
        return 0;
    }
    else {
        *result = lista->freq;
        return 1; // Esito positivo
    }
}

void delete(table* T, int k) {
    int pos = hashF(k, T->m);
    node* current = T->table[pos];
    node* previous = NULL;
    while(current != NULL && current->key != k) {
        previous = current;
        current = current->next;
    }
    /* Current punterà al blocco della lista che contiene la chiave k, se c'è */

    if(current != NULL) { // Se k è effettivamente presente nella tabella
        if(previous == NULL) { // Cancellazione in testa alla lista
            /* Avanziamo il puntatore alla testa della lista
             * e deallochiamo la "vecchia" testa (puntata da current)
             */
            T->table[pos] = current->next;
            free(current);
        }
    }
}

```

```

    }
    else {          // Cancellazione in mezzo alla lista
        previous->next = current->next; // "Salta" il blocco da cancellare
        free(current);
    }
}
}

```

```

void main(){
    int el, n, k, i;
    scanf("%d%d\n",&n,&k);
    int *arr= malloc(n*sizeof(int));
    table *tab=newTable(2*n);
    for (i=0;i<n;i++){
        int tmp;
        scanf("%d",&el);
        arr[i]=el;
        if ( !search(tab,el,&tmp) )
            insert(tab,el,1);
        else{
            delete(tab,el);
            insert(tab,el,tmp+1);
        }
    }
    for (i=0;i<n;i++){
        int tmp;
        search(tab,arr[i],&tmp);
        if (tmp>=k) printf("%d ",arr[i]);
    }
}

```

## **Inserimento con contatore conflitti e max lunghezza lista**

```

void insert (node **tab, int n, int a, int b, int x, int *conf, int *maxlen){
    int position= fHash(n,x,a,b);
    int len=0;
    if (tab[position]==NULL){
        node *new= malloc(sizeof(node));
        new->key=x;
        tab[position]=new;
        tab[position]->next=NULL;
        len++;
        if (len > *maxlen) *maxlen=len;
        return;
    }
    *conf=*conf + 1;
    len++;
}

```



```

node *new=malloc(sizeof(node));
new->key=x;
new->next=NULL;
node *cur=tab[position];
while (cur->next!=NULL) {
    len++;
    cur=cur->next;
}
len++;          // incremento di nuovo perchè il while termina al penultimo elemento
if (len > *maxlen) *maxlen=len;
cur->next=new;
}

void main(){
    int n, i, a, b, x, conflicts, maxLength;
    scanf("%d",&n);
    node **tab=malloc((2*n)*sizeof(node));
    for (i=0;i<(2*n);i++){
        tab[i]=NULL;
    }
    conflicts=0;
    maxLength=0;
    scanf("%d%d",&a,&b);
    for (i=0;i<n;i++){
        scanf("%d",&x);
        insert(tab,n,a,b,x,&conflicts,&maxLength);
    }
    printf("%d\n%d\n",maxLength,conflicts);
}

```

## **Inserimento senza duplicati con conteggio max lunghezza lista, conflitti e elementi distinti**

```

void insert (node **tab, int n, int a, int b, int x, int *conf, int *maxlen, int *unique){
    int position= fHash(n,x,a,b);
    int len=0;
    if (tab[position]==NULL){
        node *new= malloc(sizeof(node));
        new->key=x;
        tab[position]=new;
        tab[position]->next=NULL;
        *unique= *unique + 1;
        len++;
        if (len > *maxlen) *maxlen=len;
        return;
    }
    *conf=*conf + 1;
    len++;
    node *new=malloc(sizeof(node));

```

```

new->key=x;
new->next=NULL;
node *cur=tab[position];
while (cur->next!=NULL && x!=cur->key) {
    len++;
    cur=cur->next;
}
len++;
if (cur->key==x){           // controllo se mi sono fermato perchè arrivato alla fine
    *conf = *conf - 1;      // oppure perchè sono incappato in un doppione
    return;
}
if (len > *maxlen) *maxlen=len;
cur->next=new;
*unique= *unique + 1;
}

```

## 6.GRAFI

### Visita in ampiezza (BFS) (Esempio Cammino Minimo)

```
struct grafo{
    int grado; //grado nodo
    int *adj;}; // nodi raggiungibili
typedef struct grafo grafo;

struct queue {
    int capacity;
    int head;
    int tail;
    int *valori;};
typedef struct queue queue;

//costruzione grafo
grafo* readGraph (int n){
    int i,g,j;
    grafo* E=malloc(n*sizeof(grafo));
    for (i=0;i<n;i++){
        scanf("%d",&g);
        E[i].grado=g;
        E[i].adj=malloc(E[i].grado*sizeof(int));
        for (j=0;j<g;j++){
            scanf("%d",E[i].adj+j);} }
    return E;
}

void init(queue *q, int n){
    q->capacity=n;
    q->head=0;
    q->tail=0;
    q->valori=malloc(sizeof(int)*q->capacity);
}

void accoda(queue* q, int k){
    q->valori[q->tail++]=k;
}

int decoda(queue* q){
    return q->valori[q->head++];
}

void deinit(queue* q){
    free(q->valori);
    q->capacity=0;
    q->head=0;
    q->tail=0;
}
```

```

}

int codavuota(queue* q){
    if(q->head==q->tail)
        return 1;
    else
        return 0;
}

int bfs (grafo* g,int start,int end,int n){
    int *colori=malloc(n*sizeof(int));
    int *distanza=malloc(n*sizeof(int));
    int i,u,v;
    queue q;

    for (i=0;i<n;i++){
        colori[i]=0;
        distanza[i]=-1;
    }
    init(&q,n);
    accoda(&q,start);
    colori[start]=1;
    distanza[start]=0;
    // estraggo dalla coda
    while (!codavuota(&q)) { // finche ho elementi in coda
        u=decoda(&q);
        for (i=0;i<g[u].grado;i++){ //esamino la lista id adiacenza del nodo
            v= g[u].adj[i];
            if (colori[v]==0){
                colori[v]=1;
                distanza[v]=distanza[u]+1;
                if (v==end) return distanza[v]; // se è il nodo finale posso terminare
                accoda(&q,v); // metto il nodo in coda
            }
        }
    }
    deinit(&q);
    return distanza[end];
}

```

## Visita in profondità (DFS) Trovare componenti connesse

```

struct vert{
    int degree;
    int *adj;};

typedef struct vert vertex;

```

// costruzione del grafo

```

vertex* readGraph (int n){
    int i,deg,j;
    vertex *g=malloc(n*sizeof(vertex));
    for (i=0;i<n;i++){
        scanf("%d",&deg);
        g[i].degree=deg;
        g[i].adj=malloc((g[i].degree)*sizeof(int));
        for (j=0;j<deg;j++){
            scanf("%d",g[i].adj+j);
        }
    }
    return g;
}

void dfs_visit (vertex *g,int src,int **colori){
    int i,cur;
    for (i=0;i<g[src].degree;i++){
        cur=g[src].adj[i];
        if ( (*colori)[cur] == 0){
            (*colori)[cur]=1;
            dfs_visit(g,cur,&(*colori));
        }
    }
}

int dfs_connesse (vertex *g,int n){
    int i;
    int *colori=malloc(n*sizeof(int));
    for (i=0;i<n;i++){
        colori[i]=0;
    }
    int count=0;
    for (i=0;i<n;i++){
        if (colori[i]==0){
            colori[i]=1;
            count++;
            dfs_visit(g,i,&colori);
        }
    }
    return count;
}

void main(){
    vertex *g;
    int n,i;
    scanf("%d",&n);
    g=readGraph(n);
    printf("%d\n",dfs_connesse(g,n));
}

```

## Esercizio Vertici-Quartieri

```
struct grafo{
    int grado;
    int *adj;};

struct queue{
    int capacity;
    int head;
    int tail;
    int *valori;};

typedef struct grafo grafo;
typedef struct queue queue;

void init (queue *q,int n){
    q->capacity=n;
    q->head=0;
    q->tail=0;
    q->valori=malloc(q->capacity*sizeof(int));
}

void enqueue (queue *q,int k){
    q->valori[q->tail++]=k;
}

int dequeue (queue *q){
    return q->valori[q->head++];
}

void deinit(queue *q){
    free(q->valori);
    q->capacity=0;
    q->head=0;
    q->tail=0;
}

int isEmpty (queue *q){
    if (q->head==q->tail)
        return 1;
    else
        return 0;
}

grafo* readGraph (int n){
    int i,deg,j;
    grafo* g=malloc(n*sizeof(grafo));
    for (i=0;i<n;i++){
        scanf("%d",&deg);
        g[i].grado=deg;
```

```

    g[i].adj=malloc(g[i].grado*sizeof(int));
    for (j=0;j<deg;j++){
        scanf("%d",&g[i].adj[j]);}
    return g;
}

void bfs (grafo* g,int start,int n,int colore,int *colori){
    int i,u,v;
    queue q;
    init (&q,n);
    enqueue(&q,start);
    while (!isEmpty(&q)){
        u=dequeue(&q);
        for (i=0;i<g[u].grado;i++){
            v=g[u].adj[i];
            if (colori[v] == 0){
                colori[v]=colore;
                enqueue(&q,v);} }
        deinit(&q);
    }
}

```

// l'idea è di far partire una bfs da un nodo,poi verificare i nodi rimasti "bianchi"  
 // e da questi lanciare un'altra bfs ma colorando di un diverso colore creando dei "quartieri"  
 // presi i nodi in input basta verificare se appartengono allo stesso "quartiere".

```

void main(){
    int i,n,a,b;
    int colore=1, end=0;
    int *colori=malloc(n*sizeof(int));
    scanf("%d",&n);
    grafo *g=readGraph(n);
    for (i=0;i<n;i++){
        colori[i]=0;}
    for (i=0;i<n;i++){
        if (colori[i]==0){
            bfs(g,i,n,colore,colori);
            colore++;} }
    // il loop si interrompe dando uno dei due nodi negativo.
    while (!end){
        scanf("%d%d",&a,&b);
        if (a==-1 || b==-1)
            end++;
        else{
            if (colori[a]==colori[b])
                printf("Connessi\n");
            else
                printf("Non Connessi\n");} }
}

```

## Grafo connesso

```
int isConnected(grafo *g,int src,int n){
    int i, u, v;
    queue q;
    int *colore=malloc(n*sizeof(int));
    for (i=0;i<n;i++){
        colore[i]=0;
    }
    init(&q,n);
    accoda(&q,src);
    colore[src]=1;
    while (!codavuota(&q)) {
        u=decoda(&q);
        for (i=0;i<g[u].grado;i++){
            v=g[u].adj[i];
            if (colore[v]==0){
                colore[v]=1;
                accoda(&q,v);
            }
        }
    }
    for (i=0;i<n;i++){
        if (!colore[i]) return 0;
    }
    deinit(&q);
    return 1;
}
```

## Verificare se un grafo contiene cicli

*/\* sfrutto la regola che in un grafo aciclico la il numero degli archi  
deve essere MINORE del numero dei nodi -1 ( sommatoria gradi < (n-1) ) \*/*

```
int isAcyclic (grafo *g,int n){
    int i, sum=0;
    for (i=0;i<n;i++){
        sum= sum + g[i].grado;
    }
    if (sum> (n-1)) return 0;  // 0 ciclico, 1 aciclico
    return 1;
}
```

*// versione con BFS*

```
int Acyclic(grafo *g,int src,int n){
    int i, u, v;
    queue q;
```



```

int *colore=malloc(n*sizeof(int));
for (i=0;i<n;i++){
    colore[i]=0;
}
init(&q,n);
accoda(&q,src);
colore[src]=1;
while (!codavuota(&q)) {
    u=decoda(&q);
    for (i=0;i<g[u].degree;i++){
        v=g[u].adj[i];
        if (colore[v]==0){
            colore[v]=1;
            accoda(&q,v);
        }
        else return 0;
    }
}
deinit(&q);
return 1;
}

```

## Numero di nodi a distanza massima dalla sorgente (selezionata)

*// Sfrutto la BFS*

```

int maxDistanceVertex(grafo *g,int src,int n){
    int i, u, v, max, count=0;
    queue q;
    int *colore=malloc(n*sizeof(int));
    int *distanza=malloc(n*sizeof(int));
    for (i=0;i<n;i++){
        colore[i]=0;
    }
    init(&q,n);
    accoda(&q,src);
    colore[src]=1;
    distanza[src]=0;
    max=distanza[src];
    while (!codavuota(&q)) {
        u=decoda(&q);
        for (i=0;i<g[u].grado;i++){
            v=g[u].adj[i];
            if (colore[v]==0){
                colore[v]=1;
                distanza[v]=distanza[u]+1;
                max=distanza[v];
                accoda(&q,v);
            }
        }
    }
}

```

```

}
for (i=0;i<n;i++){
    if (distanza[i]==max) count++;
}
deinit(&q);
return count;
}

```

## Stampa nodi a distanza K dalla sorgente (selezionata)

```

void printDistK(grafo *g,int src,int n,int k){
    int i, u, v, exit=0;
    queue q;
    int *colore=malloc(n*sizeof(int));
    int *distanza=malloc(n*sizeof(int));
    for (i=0;i<n;i++){
        colore[i]=0;
    }
    init(&q,n);
    accoda(&q,src);
    colore[src]=1;
    distanza[src]=0;
    while (!codavuota(&q) && !exit) {
        u=decoda(&q);
        for (i=0;i<g[u].grado;i++){
            v=g[u].adj[i];
            if (colore[v]==0){
                colore[v]=1;
                distanza[v]=distanza[u]+1;
                accoda(&q,v);
                if (distanza[v]==k){
                    printf("%d\n",v);
                    exit=1;          // se sono arrivato a distanza k posso anche uscire dalla procedura
                }
            }
        }
    }
    deinit(&q);
}

```

## Grafo bipartito

```

int dfs (edges *E,int s,int* colore){
    int i,v;
    for (i=0;i<E[s].grado;i++){
        v=E[s].adiacenti[i];
    }
}

```

```

    if (colore[v]==0){
        colore[v]=-colore[s];
        if (dfs(E,v,colore) == 0)
            return 0;
    }
    else if (colore[s]==colore[v])
        return 0;
}
return 1;
}

int bipartito (edges *E,int n){
    int i;
    int* colore=malloc(n*sizeof(int));
    for(i=0;i<n;i++){
        colore[i]=0;
    }
    for(i=0;i<n;i++){
        if(colore[i]==0){
            colore[i]=1;
            if(dfs(E,i,colore)==0)
                return 0;
        }
    }
    return 1;
}

```

## **Diametro grafo (cammino minimo più lungo)**

```

int bfs (edges *E,int start,int n){
    int *colori=malloc(n*sizeof(int));
    int *distance=malloc(n*sizeof(int));
    int i,u,v,max_d=0;
    queue q;
    for (i=0;i<n;i++){
        colori[i]=0;
        distance[i]=-1;
    }
    init(&q,n);
    accoda(&q,start);
    colori[start]=1;
    distance[start]=0;
    while (!codavuota(&q)){
        u=decoda(&q);
        for (i=0;i<E[u].grado;i++){
            v=E[u].adiacenti[i];
            if (colori[v]==0){
                distance[v]=distance[u]+1;
                colori[v]=1;
            }
        }
    }
    return max_d;
}

```

```

        accoda(&q,v);
    }
}
}
deinit(&q);
for (i=0;i<n;i++){
    if (!colori[i])
        return -1;
    if (distance[i]> max_d)
        max_d=distance[i];
}
return max_d;
}

```

// lancio la bfs da ogni nodo per vedere qual è il cammino minimo piu lungo

```

int diametro(edges *E,int n){
    int i,diametro=-1;
    for (i=0;i<n;i++){
        int tmp=bfs(E,i,n);
        if (tmp == -1)
            return -1;
        if (tmp > diametro)
            diametro=tmp;
    }
    return diametro;
}

```

## 7.ALBERI ABR

### Costruzione albero:

```
struct btree{
    int key;
    struct btree* left;
    struct btree* right;
};

typedef struct btree tree;

void insert (tree **t,int n){
    if (*t==NULL){
        tree *new= malloc(sizeof(tree));
        new->key= n;
        new->left=NULL;
        new->right=NULL;
        *t=new;
        return;
    }
    if (n <= (*t)->key)
        insert(&(*t)->left,n);
    else
        insert (&(*t)->right,n);
}
```

### Albero con stringhe:

```
struct btree{
    int key;
    char* name;
    struct btree* left;
    struct btree* right;
};

typedef struct btree tree;

void insert (tree **t,int n,char *nome){
    if (*t==NULL){
        tree *new= malloc(sizeof(tree));
        new->key= n;
        new->name= nome;
        new->left=NULL;
        new->right=NULL;
        *t=new;
        return;
    }
}
```

```

if (n < (*t)->key)
    insert(&(*t)->left,n,nome);
else
    insert (&(*t)->right,n,nome);
}

```

### **Deallocare albero:**

```

void free_tree(tree_node_t* root) {
    if(root == NULL)
        return;
    /* Deallocazione del figlio sinistro */
    if(root->left != NULL)
        free_tree(root->left);
    /* Deallocazione del figlio destro */
    if(root->right != NULL)
        free_tree(root->right);
    /* Deallocazione del nodo corrente */
    free(root);
}

```

### **Inorder traversal (stampa in ordine crescente)**

```

void inorderVisit(tree *t){
    if (t==NULL) return;
    inorderVisit(t->left);
    printf("%d - %s\n",t->key,t->name);
    inorderVisit(t->right);
}

```

### **Preorder traversal:**

```

void preOrderVisit(tree *t){
    if (t==NULL) return;
    printf("%d - %s\n",t->key,t->name);
    inorderVisit(t->left);
    inorderVisit(t->right);
}

```

### **Postorder traversal:**

```

void postOrderVisit(tree *t){
    if (t==NULL) return;
    inorderVisit(t->left);
    inorderVisit(t->right);
    printf("%d - %s\n",t->key,t->name);
}

```

InOrder(root) visits nodes in the following order:

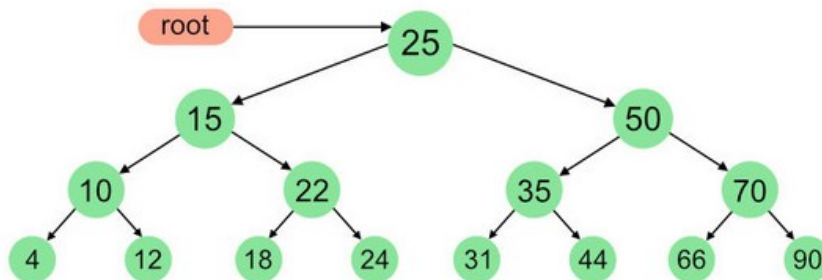
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



### LCA (antenato comune):

// visito dalla radice, il primo nodo che incontro compreso tra n1 e n2 è il LCA

```
int lca(tree *t,int n1,int n2){
    if (t->key > n1 && t->key > n2)
        return lca(t->left,n1,n2); // se n1 e n2 sono più piccoli della radice vado a sinistra
    if (t->key < n1 && t->key < n2)
        return lca(t->right,n1,n2); // se n1 e n2 sono più grandi della radice vado a destra
    return t->key;
}
```

### Secondo massimo:

```
int secondMax(tree *t){
    tree *cur=t;
    tree *prev=NULL;
    while (cur->right!=NULL){
        prev=cur;
        cur=cur->right;
    }
    if (cur->left==NULL) return prev->key;
    cur=cur->left;
    while (cur->right!=NULL) {
        cur=cur->right;
    }
    return cur->key;
}
```

### **Sottofoglia di valore massimo (data una chiave):**

```
int srcMaxSub (tree *t, char* s){
    if (t==NULL) return -1;
    tree *cur= t;
    int found=0;
    while (cur->left!=NULL || cur->right!=NULL && !found) {
        if ( !strcmp(cur->name,s)) found++;
        else{
            if ( strcmp(cur->name,s) > 0 && cur->left!=NULL)
                cur= cur->left;
            else if (cur->right!=NULL)
                cur=cur->right;
        }
    }
    int dir; // uso una variabile per capire la direzione del cammino
    if (cur->right!=NULL && cur->right->key > cur->key){
        cur= cur->right;
        dir=0;
    }
    else if (cur->left!=NULL){
        cur= cur->left;
        dir=1;
    }
    if (!dir) {
        while (cur->right!=NULL){
            cur=cur->right;
        }
    }
    else{
        while (cur->left!=NULL) {
            cur=cur->left;
        }
    }
    return cur->key;
}
```

### **Ricerca Binaria con profondità:**

```
int binarySearch (tree *t, int x){
    int depth=0;
    tree *cur=t;
    while (cur!=NULL) {
        if (cur->key==x)
            return depth;
        else if (x <= cur->key)
            cur=cur->left;
        else
            cur= cur->right;
    }
```



```

    depth++;
}
return -1;
}

```

### **Altezza di un ABR:**

```

int altezza (tree *t){
    if (t==NULL) return 0;
    int hsx= altezza(t->left);
    int hdx= altezza(t->right);
    return max(hsx,hdx) + 1 (va implementato con if-else)
}

```

### **Minima chiave Cammino di somma massima:**

```

solution minMaxPath (tree *t){
    if (t==NULL){
        solution res;
        res.min=0;
        res.path=0;
        return res;
    }
    if (t->left==NULL && t->right==NULL){
        solution leaf;
        leaf.min= t->key;
        leaf.path= t->key;
        return leaf;
    }
    solution sx= minMaxPath(t->left);
    solution dx= minMaxPath(t->right);

    // scelgo il cammino maggiore
    solution max;
    if (sx.path > dx.path)
        max=sx;
    else if (sx.path < dx.path)
        max=dx;
    else if (sx.min < dx.min)
        max=sx;
    else
        max=dx;
    max.path= max.path + t->key;
    if (t->key < max.min)
        max.min=t->key;
    return max;
}

```

## Aggiungere figlio sinistro "0" a tutti i nodi foglia

```
void add0Leaf(tree *t){
    if (t==NULL) return;
    add0Leaf(t->right);
    add0Leaf(t->left);
    if ( t->left==NULL && t->right==NULL){
        tree *zero= malloc(sizeof(tree));
        zero->key=0;
        zero->left=NULL;
        zero->right=NULL;
        t->left=zero;
    }
}
```

## Esiste un cammino root-leaf di valore k?

```
int existPathK(tree *t,int k){
    if (t==NULL)
        return 0;
    int sub= k - t->key;
    int b= 0; // booleano da ritornare
    if (sub==0 && t->left==NULL && t->right==NULL)
        return 1;
    b= (b || existPathK(t->left,sub));
    b= (b || existPathK(t->right,sub));
    return b;
}
```

## Contare nodi non foglia

```
int nonLeafCount(tree *t){
    if (t==NULL)
        return 0;
    if (t->left==NULL && t->right==NULL) return 0;
    return 1 + nonLeafCount(t->right) + nonLeafCount(t->left);
}
```

## Invertire un ABR

```
void invertT(tree *t){
    if (t==NULL) return;
    if (t->left==NULL && t->right==NULL) return;
    tree *tmp= t->left;
    t->left=t->right;
    t->right=tmp;
    invertT(t->right);
    invertT(t->left);
}
```

## Verificare se un albero è un ABR

```
void isABR(tree *t,int *ans){
    if (t==NULL) {
        return;
    }
    int tmp= t->key;
    isABR(t->left,ans);
    if (t->key < tmp){
        *ans=0;
        return;
    }
    isABR(t->right,ans);
}
```

## Calcolare prodotto dei nodi foglia

```
void leafProd(tree *t,int *prod){
    if (t==NULL)
        return;
    if (t->left==NULL && t->right==NULL)
        *prod= (*prod) * t->key;
    leafProd(t->right,prod);
    leafProd(t->left,prod);
}
```

## Verificare se la somma dei nodi è uguale alla radice

```
void SumTree(tree *t,int *s){
    if (t==NULL) {
        return;
    }
    SumTree(t->left,s);
    *s= *s + t->key;
    SumTree(t->right,s);
}
```

nel main....

```
SumTree(tr,&sum);
printf("%d\n", (tr->key==(sum-tr->key)) );
```

## Stampa chiavi appartenenti all'intervallo [a,b] in un ABR

```
void intervallo(tree *t,int a,int b){
    if (t==NULL) {
        return;
    }
    intervallo(t->left,a,b);
    if (t->key <= b && t->key >=a) printf("%d\n",t->key);
    intervallo(t->right,a,b);
}
```

## Stampa sottoalbero di una chiave k (se presente)

```
void subTreeK(tree *t,int k){
    if (t==NULL) return;
    if (t->key==k)
        inorderTrav(t);
    if (k < t->key)
        subTreeK(t->left,k);
    else
        subTreeK(t->right,k);
}
```

## Costruire ABR da array ordinato in O(n)

```
tree* arrayToABR(int *arr,int sx,int dx){
    if (sx>dx) return NULL;
    tree *t=malloc(sizeof(tree));;
    int cx = (sx+dx)/2;
    t->key=arr[cx];
    t->left= arrayToABR(arr,sx,cx-1);
    t->right= arrayToABR(arr,cx+1,dx);
    return t;
}
```

*/\* se non fosse ordinato devo prima ordinarlo impiegando nlogn quindi non ha senso questa procedura, faccio semplicemente un inserimento elemento per elemento \*/*

## Visita SX > DX (stampa nodi con sottoalbero sinistro più profondo del destro)

```
typedef struct ris {
    int sx;
    int dx;
}ris;
```

```
typedef struct tree {
```

```

    int info;
    int ok;
    struct tree *right;
    struct tree *left;
} tree ;

void visita_simm(tree *t){
    if(t!=NULL){
        visita_simm(t->left);
        if(t->ok ==1) printf("%d\n",t->info);
        visita_simm(t->right);
    }
}

.....

ris * stampanodi ( tree *t ) {
    if(t==NULL) {
        ris *risultato=malloc(sizeof(ris));
        risultato->sx=0;
        risultato->dx=0;
        return risultato;
    }
    else {
        ris *left=stampanodi(t->left);
        ris *right=stampanodi(t->right);
        ris *risultato=malloc(sizeof(ris));
        if(left->sx > right->dx) {
            t->ok=1;
        }
        risultato->sx=left->sx+1;
        risultato->dx=right->dx+1;
        return risultato;
    }
}

int main() {
    tree *tr=NULL;
    int numel,i;
    scanf("%d",&numel);
    for(i=0;i<numel;i++) {
        int valore;
        scanf("%d",&valore);
        tr=inserisci(&tr,valore);
    }
    stampanodi(tr);
    visita_simm(tr);
    return 0;
}

```

### Dato un albero e un intero k, contare nodi $\leq k$

```
int countGT (tree *t, int k){
    if (t==NULL) return 0;
    if (t->key > k)          // se k è minore della radice vado solo a sinistra
        return countGT(t->left,k);
    return 1 + countGT(t->left,k) + countGT(t->right,k);
}
```

### Stampare il minimo del sottoalbero radicato in ogni nodo

```
void minSubtree (tree *t){
    if (t==NULL) return;
    if (t->left==NULL){
        printf("%d\n",t->key);
        return;
    }
    minSubtree(t->left);
    minSubtree(t->right);
}
```

### Dimensione ABR (numero nodi)

```
int dimTree(tree *t){
    if (t==NULL) return 0;
    return dimTree(t->left) + dimTree(t->right) + 1;
}
```

### Verificare se ABR è completamente bilanciato

```
res compBilanciato(tree *t){
    res bil;
    if (t==NULL){
        bil.bool=1;
        bil.h=-1;
        return bil;
    }
    if (t->left==NULL && t->right==NULL){
        bil.bool=1;
        bil.h=0;
        return bil;
    }
}
```

```

    res dx;
    res sx;
    if (t->right!=NULL)
        dx = compBilanciato(t->right);
    if (t->left!=NULL)
        sx = compBilanciato(t->left);
    if (dx.bool==1 && sx.bool ==1 && sx.h==dx.h){
        bil.bool=1;
        bil.h= sx.h;
        return bil;
    }
    bil.bool=0;
    bil.h=-1;
    return bil;
}

```

## Stampa nodi cardine ABR

// p è la profondità, si chiama con p=0 (se non richiesto diversamente)

```

int NodiCardine(tree *t, int p){
    if (t==NULL) return -1;
    int hdx = NodiCardine(t->right,p+1);
    int hsx = NodiCardine(t->left,p+1);
    int h;
    if (hdx > hdx) h=1+hdx;
    else h= 1+hsx;
    if (h==p) printf("%d\n",t->key);
    return h;
}

```

## Alberi ternari

```

void insert (tree **t,int n){
    if (*t==NULL){
        tree *new= malloc(sizeof(tree));
        new->key= n;
        new->left=NULL;
        new->center=NULL;
        new->right=NULL;
        *t=new;
        return;
    }
    if (n < (*t)->key)
        insert(&(*t)->left,n);
    else if (!(n%(*t)->key))
        insert(&(*t)->center,n);
    else

```

```

insert (&(*t)->right,n);
}

```

**// conta i nodi che hanno 3 figli**

```

int ThreeSonsNodes(tree *t){
    if (t==NULL) return 0;
    if (t->right!=NULL && t->left!=NULL && t->center!=NULL)
        return 1 + ThreeSonsNodes(t->left) + ThreeSonsNodes(t->center) +
ThreeSonsNodes(t->right);
    return ThreeSonsNodes(t->left) + ThreeSonsNodes(t->center) + ThreeSonsNodes(t->right);
}

```

### **Confronta ABR (controlla se il percorso da root a k è lo stesso in entrambi)**

```

int samePath (tree *t1, tree *t2, int k){
    if (t1->key==k && t2->key==k)
        return 1;
    if (k < t1->key && k < t2->key && t1->key == t2->key)
        return samePath(t1->left,t2->left,k);
    else if (k > t1->key && k > t2->key && t1->key == t2->key)
        return samePath(t1->right,t2->right,k);
    else return 0;
}

```

### **Controllare se un ABR è completo (non ha nodi con un figlio solo)**

```

int isComplete (tree *t){
    if (t==NULL) return 0;
    if (t->left==NULL && t->right==NULL) return 1;
    int sx= isComplete(t->left);
    int dx= isComplete(t->right);
    if (sx==1 && dx==1) return 1;
    return 0;
}

```

### **Mediana ABR ( $n/2$ elementi $< k < n/2$ elementi)**

// sfrutto la inorder visit contando i nodi che visito

```

int mediana(tree *t,int count,int n,int *med){
    if (count > n/2 || n<0) return;

    if (t->left!=NULL)
        count= mediana(t->left,count,n,med);
    if (count== n/2){
        *med= t->key;
    }
}

```



```

    return 1+count;
}

count++;
if (t->right!=NULL)
    count= mediana(t->right,count,n,med);
return count;
}

```

## Stampare chiave e profondità di ogni nodo

```

void depthVisit (tree *t,int d){ // nel main la chiamo con d=0
    if (t==NULL) return;
    depthVisit(t->left,d+1);
    printf("%d - %d\n",t->key,d);
    depthVisit(t->right,d+1);
}

```

## 8.ESERCIZI COMPLETI SVOLTI

### K stringhe più frequenti

```
#define MAXLEN 101
```

```
struct str_occ{      // struct con stringa e occorrenze
    char *s;
    int occ;};
typedef struct str_occ str_occ;
```

```
// compare per stringhe
```

```
int strCompare (const void* a,const void* b){
    char *st1 = *(char**)a;
    char *st2 = *(char**)b;
    return strcmp(st1,st2);
}
```

```
// compare per struct rispetto alle occorrenze
```

```
int occCompare (const void* a, const void* b){
    str_occ s1 = *(str_occ*)a;
    str_occ s2 = *(str_occ*)b;
    return -(s1.occ - s2.occ);
}
```

```
// compare per struct rispetto alla stringa
```

```
int nameCompare (const void* a, const void* b){
    str_occ s1 = *(str_occ*)a;
    str_occ s2 = *(str_occ*)b;
    return strcmp(s1.s,s2.s);
}
```

```
str_occ* contaOcc (char** arr,int n,int *unique){ // lavora su array ordinato
    int i=1, count, j=0;
    str_occ *stringhe_occorrenze = malloc(n*sizeof(str_occ));
    while (i<n){
        count=1;
        while (i<n && !(strcmp(arr[i],arr[i-1]))) {
            count++;
            i++;
        }
        str_occ coppia;
        coppia.s= arr[i-1];
        coppia.occ= count;
        stringhe_occorrenze[j]=coppia;
        j++;
        i++;
    }
}
```

```

    *unique=j; // ritorno la lunghezza effettiva dell'array di struct
    return stringhe_occorrenze;
}

void main(){
    int n, k, i, unique;
    scanf("%d%d",&n,&k);
    str_occ* coppie;
    char** stringhe= malloc(n*sizeof(char*));
    for (i=0;i<n;i++){
        char* str= malloc(MAXLEN*sizeof(char));
        scanf("%s",str);
        stringhe[i]=str;
    }
    qsort(stringhe,n,sizeof(char*),strCompare); // ordino l'array di stringhe da passare alla funzione
    coppie=contaOcc(stringhe,n,&unique);
    qsort(coppie,unique,sizeof(str_occ),occCompare); //ordino per occorrenze decrescenti
    qsort(coppie,k,sizeof(str_occ),nameCompare); //prendo le prime k e le ordino lessicograficamente
    for (i=0;i<k;i++){
        printf("%s\n",coppie[i].s);
    }
}

```

## Punti colorati

```

struct point{
    int x;
    int y;
    int c;};
typedef struct point point;

int colCompare (const void* a,const void* b){
    point p1= *(point*)a;
    point p2= *(point*)b;
    return p1.c - p2.c;
}

int query(point*arr,int n,point inq1,point inq2){
    int i, last=-1, count=0;
    for (i=0;i<n;i++){
        if (arr[i].x >= inq1.x && arr[i].y >= inq1.y && arr[i].x <= inq2.x && arr[i].y <= inq2.y){
            if (arr[i].c != last){
                count++;
                last= arr[i].c;
            }
        }
    }
    return count;
}

```

```

}

void main(){
    int i, n, m, asc, ord, col, qx1, qy1, qx2, qy2;
    scanf("%d%d",&n,&m);
    point* points= malloc(n*sizeof(int));
    for (i=0;i<n;i++){
        scanf("%d%d%d",&asc,&ord,&col);
        points[i].x=asc;
        points[i].y=ord;
        points[i].c=col;
    }
    qsort(points,n,sizeof(point),colCompare); // ordino per colore
    for (i=0;i<m;i++){
        scanf("%d%d%d%d",&qx1,&qy1,&qx2,&qy2);
        point p1,p2;
        p1.x=qx1;
        p1.y=qy1;
        p1.c=-1; // non ha importanza il colore del punto in cui cerco
        p2.x=qx2;
        p2.y=qy2;
        p2.c=-1;
        printf("%d\n",query(points,n,p1,p2));
    }
}

```

## Scelta esami studente (prendo prima quelli con piu CFU e meno difficoltà)

```

struct esame{
    char* sigla;
    int cfu;
    int diff;};
typedef struct esame esame;

int examCompare(const void* a,const void* b){
    esame s1 = *(esame*)a;
    esame s2 = *(esame*)b;
    if (s1.cfu!=s2.cfu)
        return -(s1.cfu - s2.cfu);
    else if (s1.diff!=s2.diff)
        return s1.diff - s2.diff;
    else
        return strcmp(s1.sigla,s2.sigla);
}

int strCompare (const void* a,const void* b){
    char* st1 = *(char**)a;
    char* st2 = *(char**)b;

```

```

        return strcmp(st1,st2);
    }

void main(){
    int i, k, n, cr, d, acc=0;
    scanf("%d%d",&k,&n);
    esame *arr= malloc(n*sizeof(esame));
    char **scelti= malloc(n*sizeof(char*));
    for (i=0;i<n;i++){
        char* sig= malloc(MAXLEN*sizeof(char));
        scanf("%s%d",&sig,&cr,&d);
        arr[i].sigla=sig;
        arr[i].cfu=cr;
        arr[i].diff=d;
    }
    qsort(arr,n,sizeof(esame),examCompare);
    i=0;
    int countScelti=0;
    while (i<n){
        if ( (arr[i].cfu + acc) > k )
            i++;
        else if ( (arr[i].cfu + acc) == k ){
            char* esameScelto= malloc(strlen(arr[i].sigla)*sizeof(char));
            esameScelto=arr[i].sigla;
            scelti[countScelti]=esameScelto;
            countScelti++;
            i=n;
        }
        else {
            acc= acc + arr[i].cfu;
            char* esameScelto= malloc(strlen(arr[i].sigla)*sizeof(char));
            esameScelto=arr[i].sigla;
            scelti[countScelti]=esameScelto;
            countScelti++;
            i++;
        }
    }
    qsort(scelti,countScelti,sizeof(char*),strCompare);
    for (i=0;i<countScelti;i++){
        printf("%s\n",scelti[i]);
    }
}

```

## Gestione Ambulatorio (coda implementata con liste)

```

struct node {
    char* key;
    struct node* next;
}

```

```

};
typedef struct node node;

struct list {
    node* head;
    node* tail;
    int size;
};
typedef struct list list;

int compare (const void* a, const void* b){
    char* s1 = *(char**)a;
    char* s2 = *(char**)b;
    return strcmp(s1,s2);
}

list* newList() {
    list* lst = malloc(sizeof(list));
    lst->head = NULL;
    lst->tail = NULL;
    lst->size = 0;
    return lst;
}

void destroyList(list* lst) {
    while(lst->head != NULL) {
        node* tmp = lst->head; // Salva l'elemento corrente
        lst->head = lst->head->next; // Avanza nella lista
        free(tmp); // Dealloca l'elemento
    }
    free(lst); // Free della struct che conteneva la lista
}

void pushTail(list *lst,char* el){
    node *new=malloc(sizeof(node));
    new->next=NULL;
    new->key=el;
    if (lst->size==0){
        lst->head=new;
        new->next=NULL;
        lst->tail=lst->head;
        lst->size++;
        return;
    }
    lst->tail->next=new;
    lst->tail=new;
    lst->size++;
}

void dropHead(list *lst){

```

```

if (lst->size==1){
    free(lst->head);
    lst->head=NULL;
    lst->tail=NULL;
    lst->size--;
    return;
}
if (lst->size>1){
    node *tmp= lst->head;
    lst->head=lst->head->next;
    free(tmp);
    lst->size--;
}
}

void main(){
    int closed=0, op;
    list *ambulatorio=newList();
    while (!closed){
        scanf("%d",&op);
        if (op==1){
            char *new= malloc(MAXLEN*sizeof(char));
            scanf("%s",new);
            pushTail(ambulatorio,new);
        }
        else if (op==2)
            dropHead(ambulatorio);
        else if (op==0)
            closed++;
    }
    if (ambulatorio->head!=NULL){
        int i=0;
        int len=ambulatorio->size;
        char** arr= malloc(len*sizeof(char*));
        node *cur= ambulatorio->head;
        while (cur!=NULL){
            char* el= malloc(MAXLEN*sizeof(char));
            el= cur->key;
            arr[i]=el;
            i++;
            cur=cur->next;
        }
        free(cur);
        destroyList(ambulatorio); // ora la lista non serve più, l'ho trasferita su array
        qsort(arr,len,sizeof(char*),compare);
        for (i=0;i<len;i++){
            printf("%s\n",arr[i]);
        }
        for (i=0;i<len;i++){ // libero l'array
            free(arr[i]);
        }
    }
}

```

```
    }  
    free(arr);  
}  
printf("$\n");  
}
```



## 9.PROMEMORIA UTILITY

### **Compilazione semplice**

```
cd Desktop  
gcc programma.c -o a.out  
./a.out
```

### **Compilazione per Valgrind**

```
gcc programma.c -g  
valgrind a.out
```

### **Esecuzione con acquisizione diretta file input.txt**

```
./a.out < input01.txt  
./a.out < input01.txt | diff - output01.txt (per evidenziare le differenze degli output)
```