Strutture che contengono strutture:

Se la definizione di un tipo `struct` contiene elementi di un altro tipo `struct`, allora la definizione di quest'ultimo tipo deve essere già nota al compilatore.

```
struct a {
    ...
    struct b sb;  /* ERRORE: struct b tipo sconosciuto */
};

struct b {
    ...
};
```

La definizione di `struct b` deve precedere quella di `struct a`:

```
struct b {
    ...
};

struct a {
    ...
    struct b sb;  /* OK */
};
```

Strutture autoreferenziali

Poiché la definizione di un tipo `struct` deve essere nota per potere includere membri di quel tipo in altri tipi `struct`, non è possibile che un tipo `struct` contenga membri dello stesso tipo:

```
struct r {
    ...
    struct r next; /* ERRORE: struct r tipo sconosciuto */
};
```

Ma poiché i tipi puntatori a dati hanno tutti la stessa dimensione in memoria, è possibile includere membri di tipo puntatore al tipo `struct` che stiamo dichiarando:

```
struct r {
    ...
    struct r *next; /* OK: next e' puntatore a struct r */
};
```

Le strutture autoreferenziantisi tramite puntatori sono lo strumento principe del C per costruire strutture dati dinamiche.

Strutture con riferimenti mutui

Usando i puntatori a tipi `struct` è possibile la mutua referenza:

```
struct a {
    ...
    struct b *pb;
};

struct b {
    ...
    struct a *pa;
};
```

In questo caso, `struct b` avrebbe potuto contenere una variabile di tipo `struct a` come membro, al posto del puntatore `struct a *pa`, poiché `struct b` conosce la definizione di `struct a`, mentre `struct a` non conosce la definizione di `struct b`.

Liste, Alberi, Grafi

Liste concatenate

La lista concatenata è una struttura che organizza i dati in maniera sequenziale.

Mentre gli elementi di un array sono accessibili direttamente, gli elementi di una lista devono essere acceduti sequenzialmente: se voglio accedere all'$i$esimo elemento devo scandire la lista dal primo all'$(i-1)$esimo elemento.

Mentre le dimensioni di un array sono rigide, le dimensioni di una lista variano dinamicamente, tramite le operazioni di inserzione e cancellazione di elementi.

Elementi logicamente adiacenti in una lista possono essere allocati in posizioni di memoria non adiacenti.

L'implementazione di liste in C può avvenire in vari modi: il modo standard prevede l'uso di strutture e puntatori.

Ecco la definizione del tipo elemento di una lista di interi.

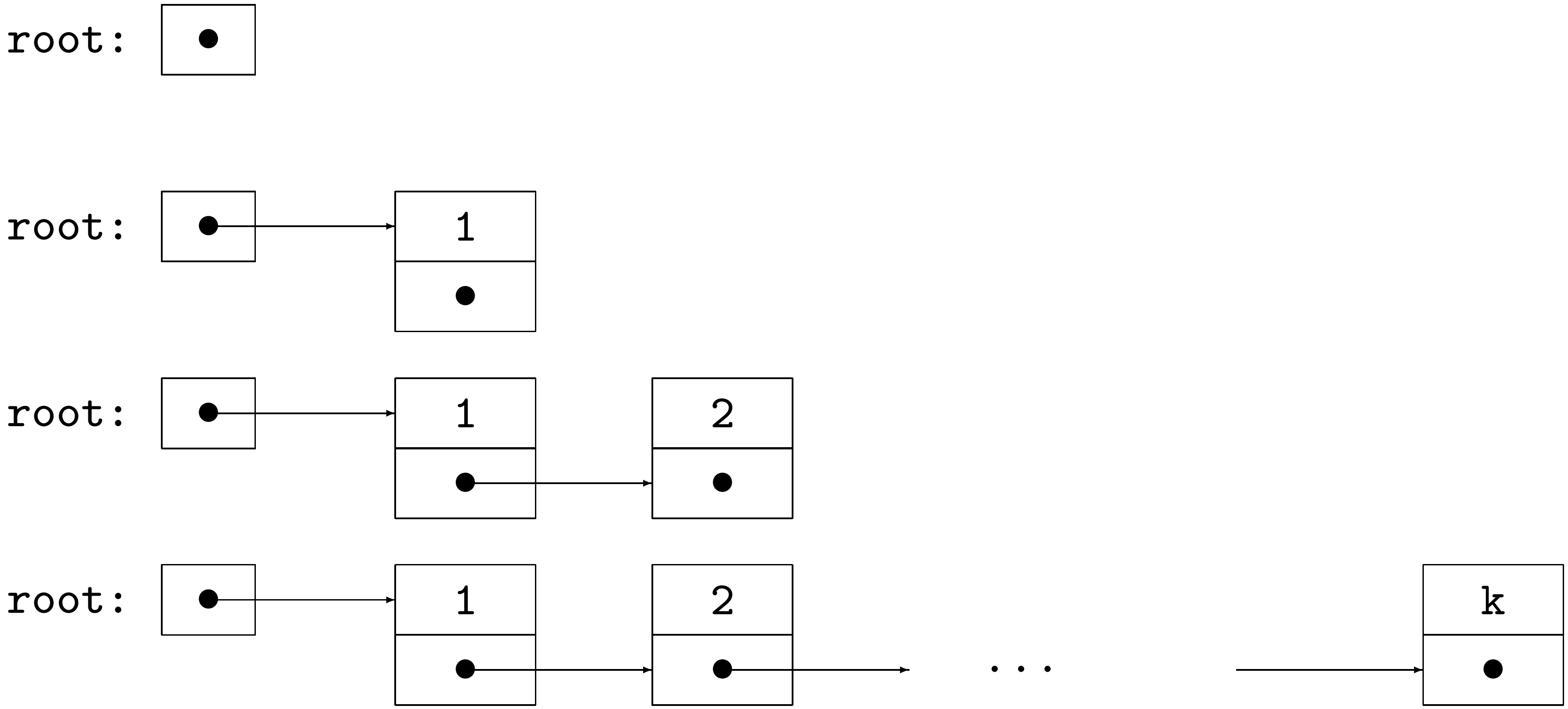
```
struct intlist {
    int dato;
    struct intlist *next;
};
```

Una siffatta lista deve essere acceduta tramite un puntatore al primo elemento della lista:

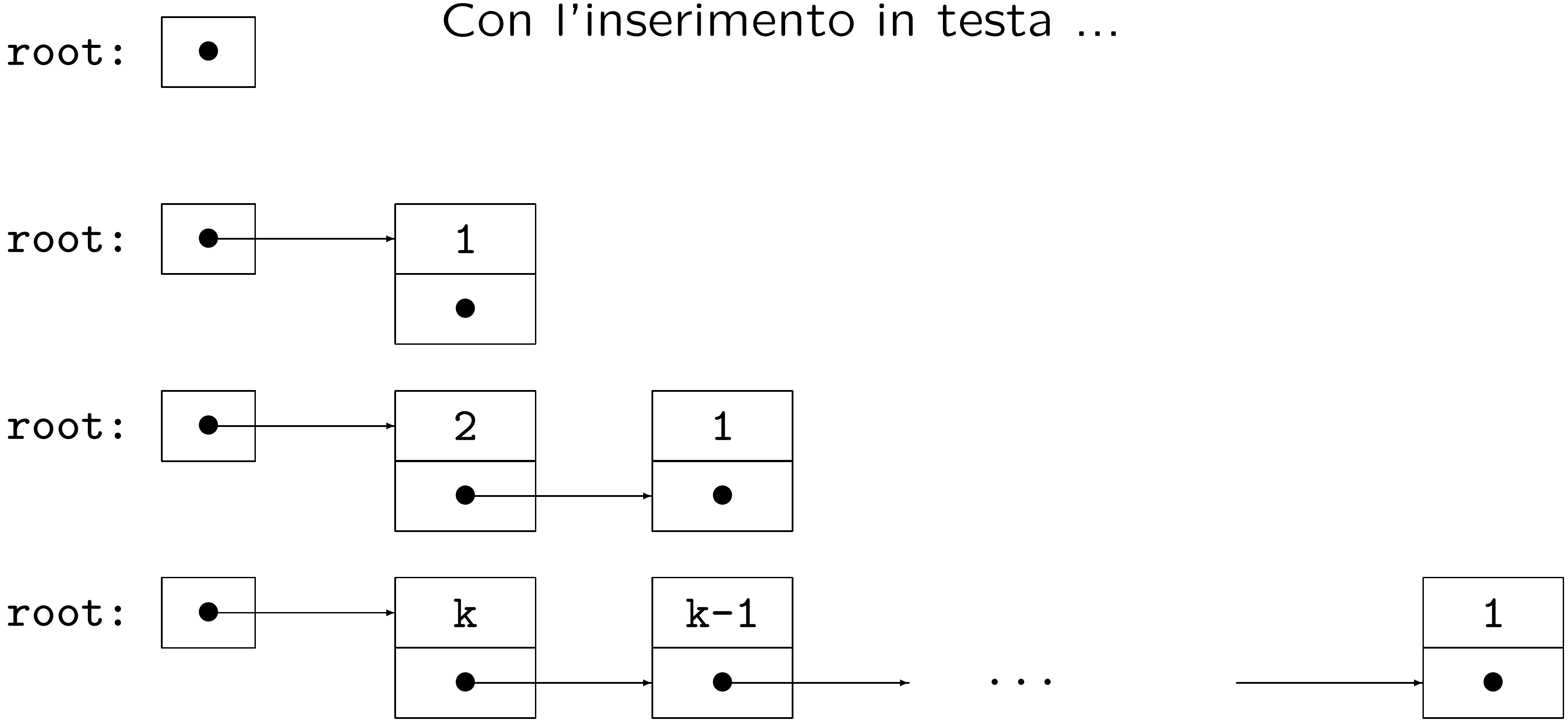
```
struct intlist *root = NULL; /* root punta a una lista vuota */

root = malloc(sizeof(struct intlist));
root->dato = 1;
root->next = malloc(sizeof(struct intlist));
root->next->dato = 2;
root->next->next = NULL; /* ora root punta a una lista di due elementi */
```

L'ultimo nodo della lista deve avere il suo campo `next` posto a `NULL`.


root:
root:
1
root:
1
2
root:
1
2
. . .
k

Con l’inserimento in testa ...



Operazioni sulle liste

Creiamo i file `intlist.h` e `intlist.c`.

In primo luogo introduciamo:

```
typedef struct intlist intlist;
```

Le operazioni: **Creazione**:
Con l'implementazione presentata la funzione per creare una lista è banale:

```
/* crea una lista vuota e ne restituisce il puntatore radice */
intlist *createlist(void)
{
    return NULL;
}
```

Se rivedessimo la nostra implementazione, potremmo dover fare operazioni più complesse per creare una lista inizialmente vuota: in tal caso riscriveremmo la funzione `createlist`.

Attraversamento:
L'attraversamento si ottiene seguendo i puntatori `next` fino a raggiungere il valore `NULL`.

```
/* visita una lista e esegue su ogni elemento la funzione op */
void traverse(intlist *p, void (*op)(intlist *))
{
    intlist *q;

    for(q = p; q; q = q->next)
        (*op)(q);
}
```

Esempio: per stampare il contenuto della lista puntata da `root`:

```
/* stampa l'elemento puntato */
void printelem(struct intlist *q)
{
    printf("\t-------\n\t|%5d|\n\t-------\n\t|  %c  |\n\t---%c---\n\t",
            q->dato, q->next ? '.' : 'X', q->next ? '|' : '-');
    if(q->next)
        printf("   |   \n\t   V\n");
}
```

Si richiama: `traverse(root,printelem);`

Inserimento in testa:
L'inserimento di un elemento in testa alla lista richiede solo l'aggiornamento di un paio di puntatori:

```
/* inserisce un elemento in testa alla lista */
intlist *insert(intlist *p, int elem)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n");
        exit(-1);
    }
    q->dato = elem;
    q->next = p;
    return q;
}
```

Per inserire un elemento di valore `15` in testa alla lista puntata da `root`:

```
root = insert(root,15);
```

Cancellazione:
La cancellazione di un elemento da una lista, con questa implementazione, risulta più complessa dell'inserzione, richiedendo la scansione della lista.

```
/* cancella l'elemento puntato da q dalla lista */
intlist *delete(intlist *p, intlist *q)  /* si assume q != NULL */
{
    intlist *r;

    if(p == q)
        p = p->next;
    else {
        for(r = p; r && r->next != q; r = r->next);
        if(r && r->next == q)
            r->next = r->next->next;
    }
    free(q);
    return p;
}
```

Per cancellare un elemento: `root = delete(root, q);`

Ricerca di un elemento:
Anche la ricerca del primo elemento di valore dato richiede la scansione della lista:

```
/* restituisce il primo elemento per cui check e' vera oppure NULL */
intlist *getcheckelem(intlist *p, char (*check)(intlist *, int), int a)
{
    intlist *q;

    for(q = p; q; q = q->next)
        if((*check)(q,a))
            return q;
    return NULL;
}
/* restituisce il primo elemento q per cui q->dato == elem */
intlist *geteleminlist(intlist *p, int elem) { return getcheckelem(p,checkexist,elem); }

char checkexist(intlist *p, int elem) { return p->dato == elem; }
```

Per cancellare il primo elemento di valore 25:

```
intlist *p = geteleminlist(root, 25);
root = delete(root,p);
```

Questo richiede due scansioni della lista: Esercizio: migliorare.

Distruzione della lista:
Si tratta di liberare tutta la memoria allocata alla lista:

```
/* distrugge la lista */
void destroylist(intlist *p) /* si assume p != NULL */
{
    while(p = delete(p,p));
}
```

Concatenazione di due liste:
Si scandisce la prima lista fino all'ultimo elemento, poi si collega il primo elemento della seconda all'ultimo elemento della prima.

```
/* concatena la lista con radice q alla lista con radice p */
intlist *listcat(intlist *p, intlist *q)
{
    intlist *r;

    if(!p)
        return q;
    for(r = p; r->next; r = r->next);
    r->next = q;
    return p;
}
```

Conteggio del numero di elementi:

```
/* ritorna il numero di elementi nella lista */
int countlist(intlist *p)
{
    int i;

    for(i = 0; p; p = p->next, i++);
    return i;
}
```

Inserimento di elementi in ordine:

```
/* inserisce un elem nella lista prima del primo elemento >= di elem */
intlist *insertinorder(intlist *p, int elem)
{
    intlist *q;

    if(!p || p->dato >= elem)
        return insert(p, elem);
    for(q = p; q->next && q->next->dato < elem; q = q->next);
    q->next = insert(q->next, elem);
    return p;
}
```

Provare l'esempio: `gcc -o list1 list1.c intlist.c`

Poiché la lista è una struttura dati definita ricorsivamente, è possibile riformulare le funzioni che effettuano una scansione dandone un'implementazione ricorsiva.

```
/* ritorna il numero di elementi nella lista: versione ricorsiva */
int rcountlist(intlist *p)
{
    return p ? rcountlist(p->next) + 1 : 0;
}

/* visita una lista e esegue su ogni elemento la funzione op: versione ricorsiva */
void rtraverse(intlist *p, void(*op)(intlist *))
{
    if(p) {
        (*op)(p);
        rtraverse(p->next,op);
    }
}

/* concatena la lista con radice q alla lista con radice p: versione ricorsiva */
intlist *rlistcat(intlist *p, intlist *q)
{
    if(p)
        p->next = rlistcat(p->next,q);
    return p ? p : q;
}
```

Modifichiamo la nostra implementazione di `intlist` aggiungendo un puntatore `prev` all'elemento precedente:

```
struct intlist {
    int dato;
    struct intlist *next, *prev;
};
```

In questo modo alcune funzioni di manipolazione risultano semplificate:

```
/* cancella l'elemento puntato da q dalla lista */
intlist *delete(intlist *p, intlist *q) /* si assume q != NULL */
{
    if(q->prev)
        q->prev->next = q->next;
    else
        p = q->next;
    if(q->next)
        q->next->prev = q->prev;
    free(q);
    return p;
}
```

Cancelliamo il primo elemento di valore 25:

```
intlist *p = geteleminlist(root, 25);

root = delete(root,p);
```

La funzione `geteleminlist` richiede una scansione lineare, mentre `delete` lavora in tempo costante.

La funzione `insert` va anch'essa modificata, ma continuerà a lavorare in tempo costante.

```
/* inserisce un elemento in testa alla lista */
intlist *insert(intlist *p, int elem)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n");
        exit(-1);
    }
    q->dato = elem;
    if(q->next = p)
        p->prev = q;
    q->prev = NULL;
    return q;
}
```

Un'ulteriore semplificazione si ottiene utilizzando un elemento *sentinella* (*dummy*) che non contiene informazione, ma serve a segnalare la fine (e l'inizio) di una lista.

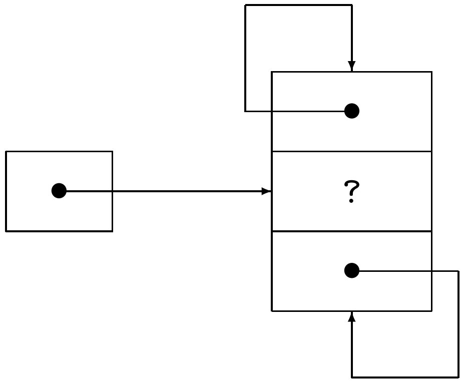
Questa soluzione va contemplata quando le lunghezze stimate dalle liste usate sono significativamente maggiori delle dimensioni di un elemento.

Bisogna in primo luogo allocare la sentinella al momento della creazione della lista.

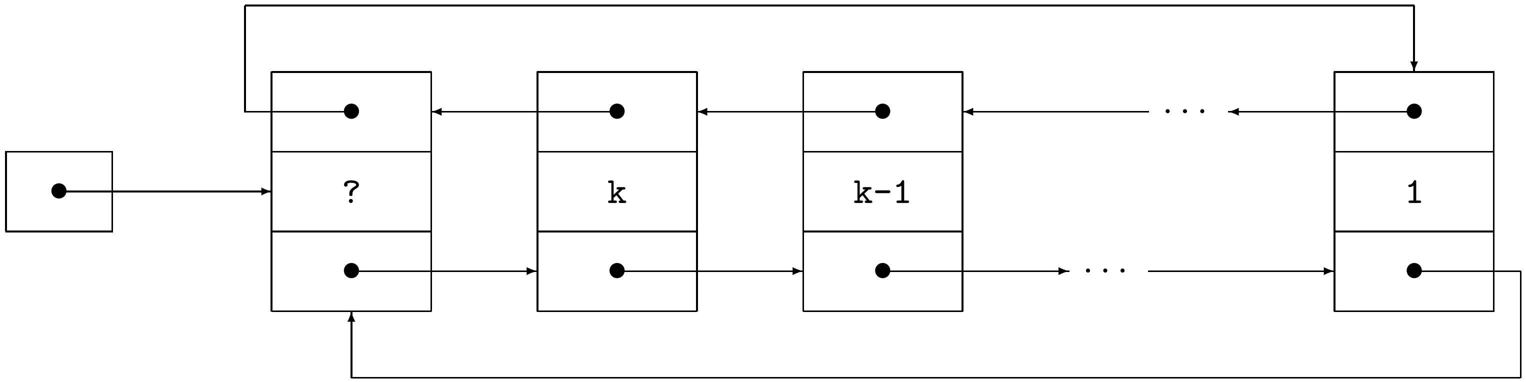
```
intlist *createlist(void)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore di allocazione nella creazione della lista\n");
        exit(-1);
    }
    q->next = q->prev = q;
    return q;
}
```

`root = createlist(); /* implementazione con sentinella */`

root:


dopo aver inserito in testa gli elementi $1, \ldots, k$:

root:


Con l'introduzione della sentinella, inserimento e cancellazione diventano:

```
void insert(intlist *p, int elem)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n");
        exit(-1);
    }
    q->dato = elem;
    q->next = p->next;
    p->next->prev = q;
    p->next = q;
    q->prev = p;
}

void delete(intlist *q)
{
    q->prev->next = q->next;
    q->next->prev = q->prev;
    free(q);
}
```

Anche altre funzioni sono state modificate:

```
void listcat(intlist *p, intlist *q)
{
    if(q != q->next) {
        if(p == p->next) {
            p->next = q->next;
            q->next->prev = p;
        } else {
            p->prev->next = q->next;
            q->next->prev = p->prev;
        }
        p->prev = q->prev;
        q->prev->next = p;
    }
    free(q);
}
```

La funzione `listcat` non attua più la scansione per raggiungere la fine della prima lista, ma lavora in tempo costante.

L'inserimento in coda alla lista risulta facile come l'inserimento in testa:

```
/* inserisce un elemento in coda alla lista */
void insertatend(intlist *p, int elem)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n");
        exit(-1);
    }
    q->dato = elem;
    q->prev = p->prev;
    p->prev->next = q;
    p->prev = q;
    q->next = p;
}
```

Si consideri questa implementazione con l'esempio rivisitato:

`gcc -o list3 list3.c intlistdummy.c`

Vi sono ancora funzioni che scandiscono la lista inutilmente:

```
int countlist(intlist *p)
{
    int i;
    intlist *q;

    if(p == p->next)
        return 0;
    for(i = 1, q = p->next; q->next != p; q = q->next, i++);
    return i;
}
```

In questo caso, per rimediare possiamo memorizzare l'informazione sul numero di elementi in un tipo `struct` che contiene anche il puntatore al primo elemento:

```
struct intlistheader {
    intlist *root;
    int count;
};
```

Per esercizio: modificare le funzioni di manipolazione di liste per implementare questa versione.

Poiché gli *stack* sono collezioni di dati per i quali l'accesso può avvenire solo attraverso la politica LIFO (Last In, First Out) è facile implementare stack attraverso liste concatenate:

```
struct intstack {
    intlist *top;
    int count;
};

typedef struct intstack intstack;

intstack *createstack(void)
{
    intstack *st = malloc(sizeof(intstack));

    if(!st) {
        fprintf(stderr,"Errore di allocazione nella creazione dello stack\n");
        exit(-1);
    }
    st->top = createlist();
    st->count = 0;
    return st;
}
```

Usiamo l'implementazione di `intlist` con sentinella che è contenuta in `intlistdummy.c`

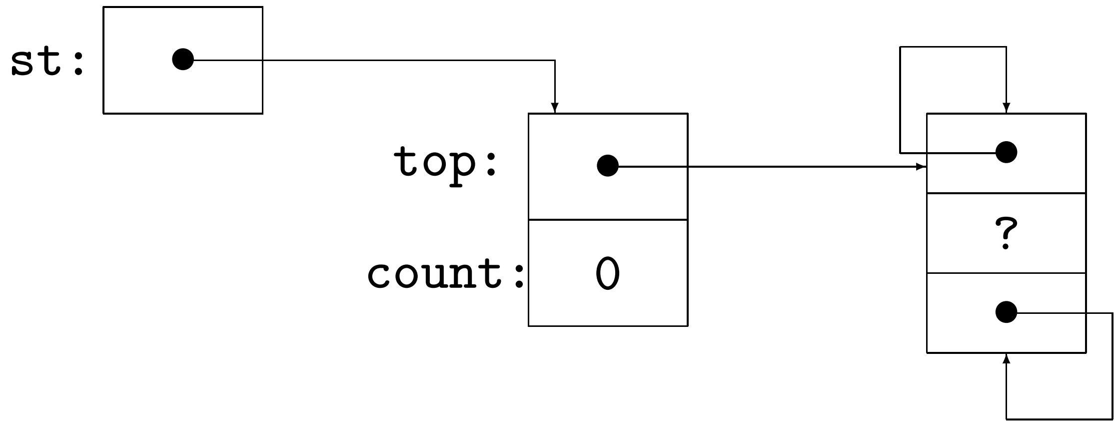
```
void push(intstack *st, int elem)
{
    insert(st->top, elem);
    st->count++;
}

int pop(intstack *st)
{
    int e;

    if(!st->count){
        fprintf(stderr,"Errore: pop su stack vuoto\n");
        exit(-2);
    }
    e = head(st->top);
    deletehead(st->top);
    st->count--;
    return e;
}

char empty(intstack *st)
{
    return !st->count;
}
```

```
st = createstack();
```



Una *coda* (*queue*) è un tipo di dati astratto analogo allo *stack*.

Le code seguono una politica FIFO (First In, First Out).

Le code si possono implementare attraverso liste concatenate, a patto di avere un'implementazione in cui l'inserimento in fondo alla lista sia efficiente.

Usiamo ancora l'implementazione contenuta in **`intlistdummy.c`**

```
struct intqueue {
    intlist *head;
    int count;
};

typedef struct intqueue intqueue;
```

L'implementazione di `intqueue` è quasi identica a quella di `intstack`:
Solo l'inserimento di dati in `intqueue` differisce:

Si confrontino `push` ed `enqueue`:

```
void push(intstack *st, int elem)
{
    insert(st->top, elem);
    st->count++;
}

void enqueue(intqueue *q, int elem)
{
    insertatend(q->head, elem);
    q->count++;
}
```

Proviamo l'esempio di confronto:

```
gcc -o cfrstq cfrstq.c intstack.c intqueue.c intlistdummy.c
```

Alberi

La definizione in termini di teoria dei grafi:
Un grafo (non orientato) senza cicli e connesso è detto *albero*.
Un albero *radicato* è una coppia $< T, r >$ dove T è un albero e r è un suo vertice, detto *radice*.

La definizione ricorsiva:
Un albero radicato (non vuoto) è:
– o un singolo nodo
– o una radice connessa a un insieme di alberi, dove ogni albero è connesso tramite un unico arco alla radice.

Dalla semplicità della definizione ricorsiva consegue la semplicità di scrittura di algoritmi ricorsivi per la manipolazione di alberi.

Mentre **array, liste, stack, code** sono strutture dati intrinsecamente *monodimensionali* (un elemento ne segue un altro), gli **alberi** strutturano l'informazione in modo più complesso.

La modellizzazione dell'informazione tramite alberi è molto diffusa e spesso naturale e intuitiva:

– gerarchie, genealogie, organigrammi, tornei, espressioni, frasi

Moltissimi algoritmi usano gli alberi per strutturare i dati.

La soluzione più comune per l'implementazione di alberi in C è attraverso l'uso di strutture autoreferenziali tramite puntatori.

Alberi binari

Un *albero binario* è un albero radicato in cui ogni nodo interno ha al più due figli. Ogni figlio è distinto come figlio *sinistro* oppure figlio *destro*.

Definiamo la struttura corrispondente a un vertice (*nodo*) di un albero di elementi di tipo `int`:

```
struct inttree {
    int dato;
    struct inttree *left, *right;
};

typedef struct inttree inttree;
```

La *radice* di un albero con siffatti nodi sarà un puntatore a `inttree`. Ecco un albero vuoto:

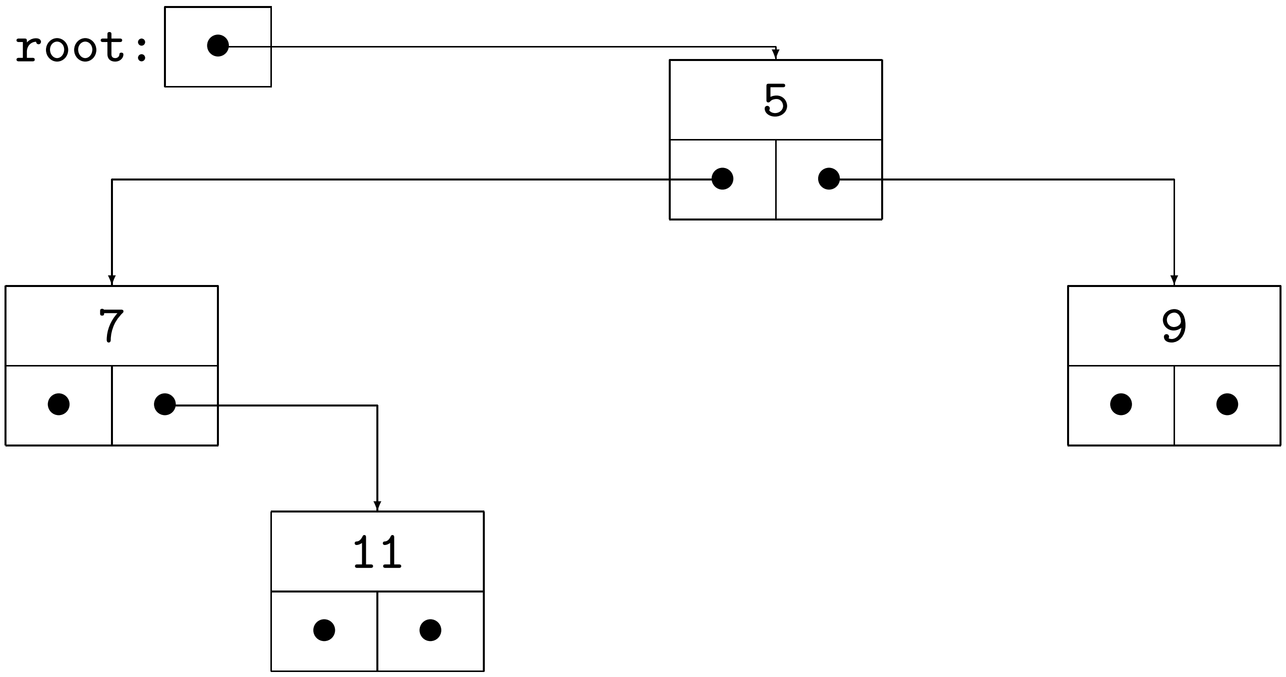
```
inttree *root = NULL;
```

I campi `left` e `right` della struttura `inttree` sono le radici dei sottoalberi sinistro e destro del nodo costituito dall'istanza della struttura.

Se un nodo non ha figlio sinistro (resp. destro), il suo campo `left` (resp. `right`) sarà posto a `NULL`.

Costruiamo manualmente un albero la cui radice sarà `root`:

```
root = malloc(sizeof(inttree));
root->dato = 5;
root->left = malloc(sizeof(inttree));
root->right = malloc(sizeof(inttree));
root->left->dato = 7;
root->right->dato = 9;
root->left->left = root->right->left = root->right->right = NULL;
root->left->right = malloc(sizeof(inttree));
root->left->right->dato = 11;
root->left->right->left = root->left->right->right = NULL;
```


root:
5
7
9
11

Attraversamento di alberi:
Un albero può essere attraversato in vari modi.
I tre modi più comuni hanno una semplice e diretta implementazione ricorsiva:

Visita in ordine **simmetrico** (**inorder**):
– sottoalbero sinistro
– radice
– sottoalbero destro

Visita in ordine **anticipato** (**preorder**):
– radice
– sottoalbero sinistro
– sottoalbero destro

Visita in ordine **posticipato** (**postorder**):
– sottoalbero sinistro
– sottoalbero destro
– radice

Visita **inorder**

```
void inorder(inttree *p)
{
    if(p) {
        inorder(p->left);
        dosomething(p);
        inorder(p->right);
    }
}
```

Visita **preorder**

```
void preorder(inttree *p)
{
    if(p) {
        dosomething(p);
        preorder(p->left);
        preorder(p->right);
    }
}
```

Visita **postorder**

```
void postorder(inttree *p)
{
    if(p) {
        postorder(p->left);
        postorder(p->right);
        dosomething(p);
    }
}
```

Per gestire in modo parametrico `dosomething(p)` almeno per funzioni di tipo `void f(inttree *p)` implementiamo gli attraversamenti con un secondo parametro di tipo puntatore a funzione:

```
void postorder(inttree *p, void (*op)(inttree *))
{
    if(p) {
        postorder(p->left,op);
        postorder(p->right,op);
        (*op)(p);
    }
}
```

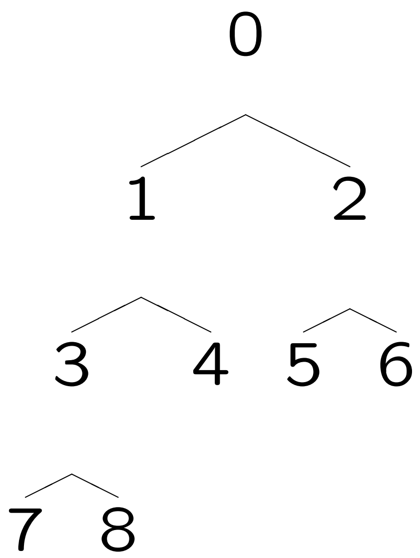
Costruzione di un albero binario a partire da un array

Un array di interi può essere visto come un albero binario in cui il figlio sinistro dell'elemento i-esimo è memorizzato nell'elemento di indice $2i+1$ e analogamente l'indice del figlio destro è $2i+2$.

```
inttree *buildtree(int *array, int i, int size)
{
    inttree *p;

    if(i >= size)
        return NULL;
    else {
        if(!(p = malloc(sizeof(inttree)))) {
            fprintf(stderr,"Errore di allocazione\n");
                exit(-1);
        }
        p->left = buildtree(array, 2 * i + 1, size);
        p->right = buildtree(array, 2 * i + 2, size);
        p->dato = array[i];
        return p;
    }
}
```

```
int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
```



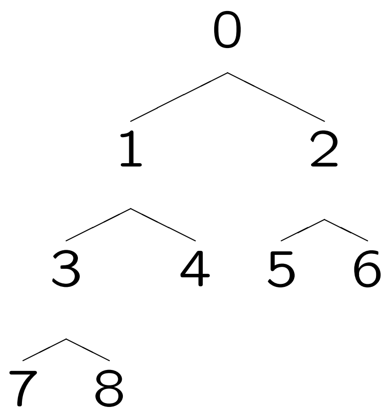
Esercizio: stampa di un albero

Vogliamo visualizzare la struttura dell'albero a video:

Poiché stampiamo una riga per volta, a ogni riga facciamo corrispondere un livello dell'albero.
Occorre stabilire:
– per ogni livello dell'albero quali nodi vi occorrono
– dove posizionare sulla riga questi nodi
La posizione del nodo sulla riga corrisponde al posto che il nodo ha nell'ordine simmetrico.

Utilizziamo un array di liste, indicizzato dai livelli per memorizzare le posizioni di stampa di ogni nodo.

Un elemento dell'i-esima lista contiene un puntatore a un nodo di livello i e la posizione di stampa per quel nodo.



					0			
			1				2	
	3			4		5		6
7		8						

Per ogni elemento, la sua collocazione (i, j) nella matrice dello schermo è così determinata:

- la riga i coincide con la profondità (livello) del nodo. Useremo una visita level-order per ottenere e gestire tale dato.

- il valore j per la colonna è stabilito dalla visita in-order dell'albero: l'elemento in esame è visitato per $(j + 1)$-esimo.

```c
struct spaces {
    int col;
    inttree *ptr;
    struct spaces *prev, *next;
};

struct spaces *createlist(void)
{
    struct spaces *q = malloc(sizeof(struct spaces));

    if(!q) {
        fprintf(stderr,"Errore di allocazione nella creazione della lista\n");
        exit(-1);
    }
    q->next = q->prev = q;
    return q;
}

void delete(struct spaces *q)
{
    q->prev->next = q->next;
    q->next->prev = q->prev;
    free(q);
}
```

```
void destroylist(struct spaces *p)
{
    while(p->next != p) delete(p->next);
    free(p);
}

void insertatend(struct spaces *p, inttree *ptr, int col)
{
    struct spaces *q = malloc(sizeof(struct spaces));

    if(!q) { fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n"); exit(-1); }
    q->ptr = ptr; q->col = col;
    q->prev = p->prev; p->prev->next = q; p->prev = q; q->next = p;
}

void levelvisit(inttree *p, int lev, struct spaces **sp)
{
    static int count = 0;

    if(!lev) count = 0;
    if(p) {
        levelvisit(p->left,lev + 1,sp);
        insertatend(sp[lev],p,count++);
        levelvisit(p->right,lev + 1,sp);
    }
}
```

```
void printtree(inttree *p)
{
    int lev,i,j;
    struct spaces *q;

    struct spaces **sp = calloc(lev = countlevels(p),sizeof(struct spaces *));

    for(i = 0; i < lev; i++)
        sp[i] = createlist();
    levelvisit(p,0,sp);
    for(i = 0; i < lev; i++) {
        j = 0;
        for(q = sp[i]->next;q != sp[i];q = q->next,j++) {
            for(;j < q->col;j++)
                printf("    ");
            printf("[%2d]",q->ptr->dato);
        }
        putchar('\n');
    }
    for(i = 0; i < lev; i++)
        destroylist(sp[i]);
    free(sp);
}
```

Commenti:

```
struct spaces {
    int col;
    inttree *ptr;
    struct spaces *prev, *next;
};
```

Creiamo un array di liste i cui elementi sono di tipo `struct spaces`:
`col` memorizza la colonna in cui stampare il dato puntato da `ptr`.
Abbiamo adattato le funzioni per l'implementazione di liste bidirezionali di interi (`intlistdummy.c`) al tipo `struct spaces`.

```
void levelvisit(inttree *p, int lev, struct spaces **sp)
{
    static int count = 0;

    if(p) {
        levelvisit(p->left,lev + 1,sp);
        insertatend(sp[lev],p,count++);
        levelvisit(p->right,lev + 1,sp);
    }
}
```

Usiamo una visita inorder dell'albero per assegnare le posizioni `col` dei nodi nella lista ausiliaria, *livello per livello*. La variabile `static int count` memorizza la posizione in ordine simmetrico del nodo correntemente visitato.

Per l'implementazione di `printtree.c` abbiamo utilizzato due semplici funzioni ricorsive, collocate in `inttree.c`, per il conteggio di nodi e livelli di un `inttree`:

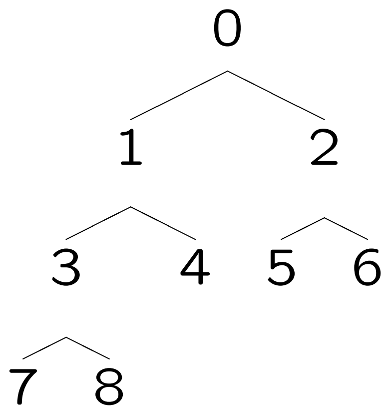
```
int countnodes(inttree *p)
{
    return p ? countnodes(p->left) + countnodes(p->right) + 1 : 0;
}

#define max(a,b)    ((a) > (b) ? (a) : (b))

int countlevels(inttree *p)
{
    return p ? max(countlevels(p->left),countlevels(p->right)) + 1 : 0;
}
```

Notare che la parentesizzazione nella definizione della macro `max` è necessaria per un uso corretto della macro stessa.

Esercizio: Se ci accontentiamo di visualizzare l'albero in modo trasposto:



			7
		3	
			8
	1		
		4	
0			
		5	
	2		
		6	

allora, non c'e' bisogno di memorizzare la riga e la colonna dove stampare ogni elementi. Non appena si è in grado di calcolare riga e colonna di un elemento, tale elemento può essere stampato. Perché?