

STRUTTURE DATI

- Sono particolari strumenti a disposizione di un programma per memorizzare, organizzare e accedere ai dati *efficientemente*
- Esistono diverse tipologie di strutture dati, ciascuna delle quali è più o meno adatta a un particolare programmi
 - Array, liste collegate, pile, code, alberi binari, alberi n-ari, hash table, skip list, rainbow table, heap, filtri di bloom
- L'effettivo utilizzo di qualsiasi struttura dati richiede l'implementazione di procedure per loro creazione e manipolazione
- Primo esempio: struttura dati statica basa su array (vedi slide seguente)

SEMPLICE RUBRICA TELEFONICA

```
#define MAX_NUM_CONTATTI 1024

struct contatto {
    int id;
    char nome[64];
    char numero[16];
};

struct rubrica {
    int num_inseriti;
    struct contatto db[MAX_NUM_CONTATTI];
};
```

INSERISCI IN RUBRICA

/* crea un contatto nella rubrica puntata da r al primo posto disponibile con nome e numero specificati. Ritorna zero in caso di successo, -1 altrimenti. Non controlla la lunghezza delle stringhe */

```
int inserisci(struct rubrica *r, char *nome, char *numero) {
    struct contatto *p;
    if (r->num_inseriti == MAX_NUM_CONTATTI) {
        printf("Rubrica piena");
        return -1; //errore
    }
    p = &r->db[r->num_inseriti]; //punta la prima area di
                                //memoria disponibile
    strcpy(p->nome, nome); //copia il nome
    strcpy(p->numero, numero); //copia il numero
    p->id = r->num_inseriti; //assegno l'indice
    r->num_inseriti++; //aumenta numero contatti
    return 0; //tutto OK
}
```

TROVA CONTATTO

/* ritorna il puntatore al numero legato al nome.
Ritorna null se il nome non è presente in rubrica */

```
char *trova_numero_da_nome(struct rubrica *r, char *nome) {
    int i;
    struct contatto *p;
    for (i=0; i<MAX_NUM_CONTATTI; i++) {
        if (i == r->num_inseriti)
            break;                //STOP se abbiamo raggiunto l'ultimo
        p = &r->db[i];             //punto l'i-esimo contatto

        if (strcmp(p->nome, nome) == 0)
            return p->numero;      //ritorno il num se i nomi coincidono
    }
    return NULL;
}
```

STAMPA

```
void stampa_rubrica(struct rubrica *r) {
    int i;
    struct contatto *p;

    printf("\nStampa rubrica: %d contatti\n", r->num_inseriti);

    for (i=0; i<MAX_NUM_CONTATTI; i++) {
        if (i == r->num_inseriti)
            break;                //STOP se abbiamo raggiunto l'ultimo
        p = &r->db[i];             //punto l'i-esimo contatto

        printf("%s, %s\n", p->nome, p->numero);
    }
}
```

POSSIBILI INEFFICIENZE?

- Stiamo gestendo male l'area allocata
 - Se mi servono più numeri? Se sto sprecando numeri?
 - Devo cambiare la macro e ricompilare
 - Questo può non essere un problema su PC con 4 GB di ram, ma su piccoli router da 16 MB?
- Non stiamo aiutando possibili algoritmi di ricerca
 - Nella funzione di insert abbiamo un ciclo che cresce linearmente con il numero di contatti inseriti
 - Stesso discorso nel caso in cui volessimo cercare se un nome è già presente in rubrica prima di inserirlo
- Non stiamo aiutando noi stessi in termini di flessibilità
 - La struttura dati è STATICA
 - Se non so quanti numeri servono?
 - Senza fare cambiamenti, in caso di cancellazione di un contatto (o inserimento ordinato) dovremmo riordinare tutto l'array (che significa copiare e shiftare ogni elemento)
 - Gli elementi di un'array devono essere contigui, quindi in caso di gestione dinamica dell'array (`malloc`, `realloc`) potremmo avere grosse inefficienze

COSA VEDREMO

- La scelta della corretta struttura dati è un passaggio fondamentale per la scrittura di qualsiasi programma
- In questo corso vedremo solo le seguenti strutture dati
 - Liste ordinate
 - Pile
 - Code
 - Alberi
- Prima di entrare nello specifico, introduciamo il concetto di struttura autoreferenziale

STRUTTURA AUTOREFERENZIALE

- E' una struttura al cui interno abbiamo un puntatore a un'altra struttura dello stesso tipo. Esempio:

```
struct node {  
    void *data;  
    struct node *next;  
};
```

- Il membro `next` è chiamato *link* perché può essere usato per collegare una struttura di tipo `node` ad un'altra
- Questo tipo di strutture è la base di liste, pile, code ed alberi (e altre...)



UN'OCCHIATA A COME OTTENIAMO UNA LISTA

L'utilizzo combinato di strutture autoreferenziali e gestione dinamica della memoria (`malloc`, `free`) ci aiuta ad ottenere delle strutture dati molto più flessibili

Inizializzazione

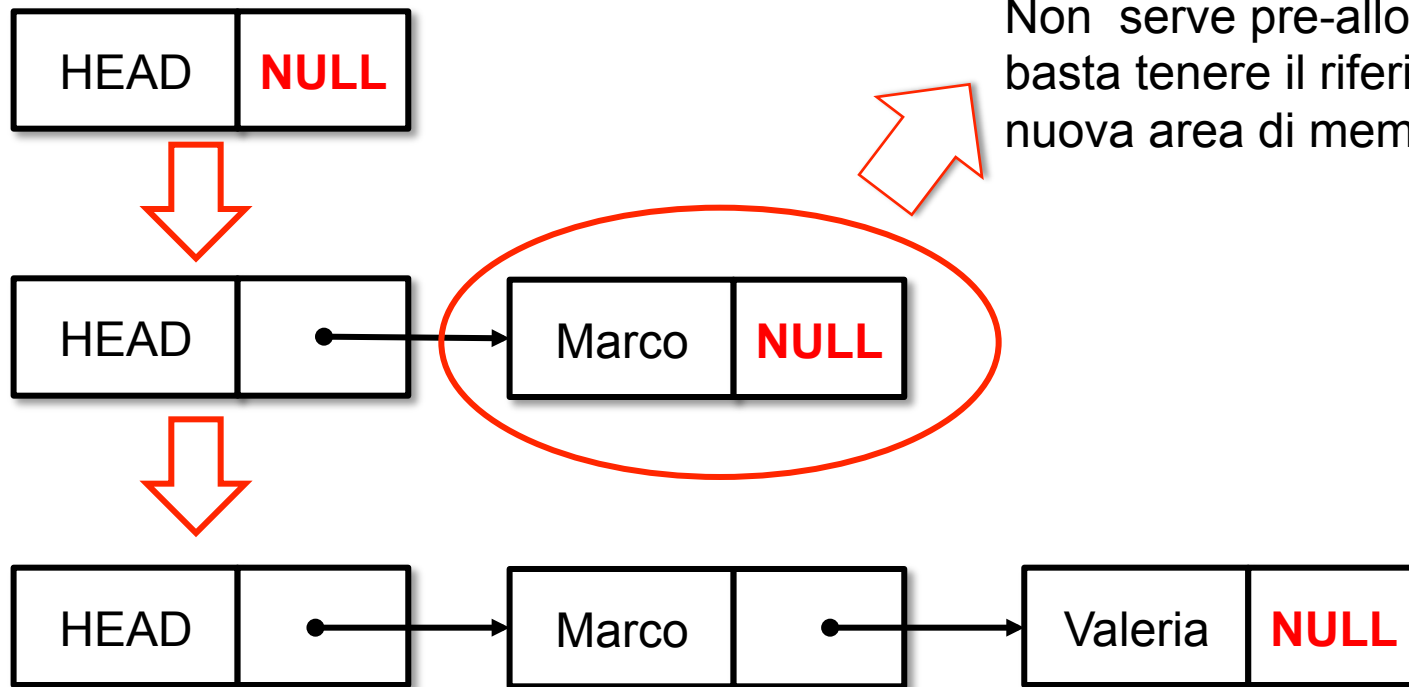


Lista Vuota

Il programma non deve sapere quanti elementi dovranno essere inseriti. Durante l'avvio allocherà semplicemente la memoria per un nodo di “testa” usato per tenere un riferimento all’inizio della lista

UN'OCCHIATA A COME OTTENIAMO UNA LISTA

Aggiunta elementi in coda

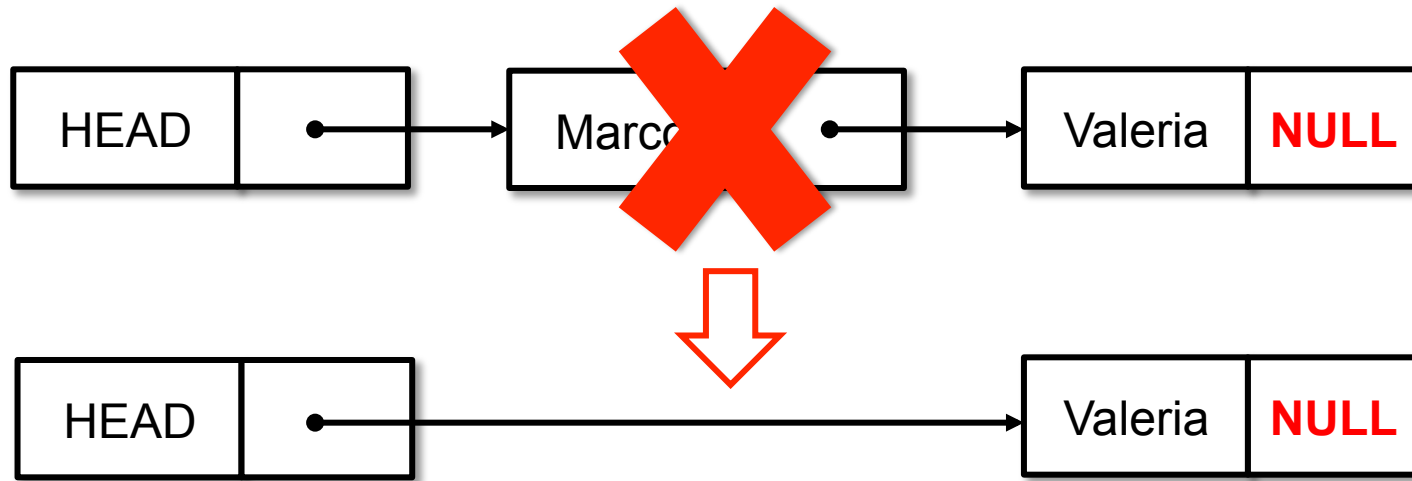


Ogni nuovo elemento non sappiamo dove starà in memoria. Non serve pre-allocare spazio ma basta tenere il riferimento alla nuova area di memoria

Ogni inserimento si riduce a (1) allocare un nuovo nodo con null come valore del puntatore link e (2) linkarlo all'ultimo

UN'OCCHIATA A COME OTTENIAMO UNA LISTA

Cancellazione elemento



Non mi devo preoccupare di “spostare” tutta la lista perché dovrò semplicemente:

- 1) Liberare la memoria del nodo cancellato
- 2) Aggiornare il puntatore (il link) del nodo precedente a quello cancellato

Lo stesso meccanismo si può applicare per inserimenti non in coda

LISTE

CONCETTI BASE

- Come visto graficamente nelle slide precedenti le liste sono una collezione organizzata lineare di strutture autoreferenziali (**nodi**), connesse tra loro attraverso puntatori (**link**).
- A una lista si accede attraverso il nodo di testa (**HEAD**) che punta a NULL in caso di lista vuota, o al primo elemento della lista
- Il link dell'ultimo nodo è per convenzione posto a NULL per indicare che non ci sono più elementi
- Vantaggi
 - Non richiede la conoscenza a priori della dimensione che può aumentare e diminuire a run time
 - Le cancellazioni e inserimenti (ordinati) richiedono non richiedono lo spostamento degli altri elementi
- **Implementiamo un programma che gestisce una lista di contatti ordinati per età**

PROGRAMMA LISTA CONTATTI

```
#define MAX_LEN 63
```

Ogni contatto avrà come campi (*user_id*, *mail*, *età*) e sarà salvato in una lista ordinata per età

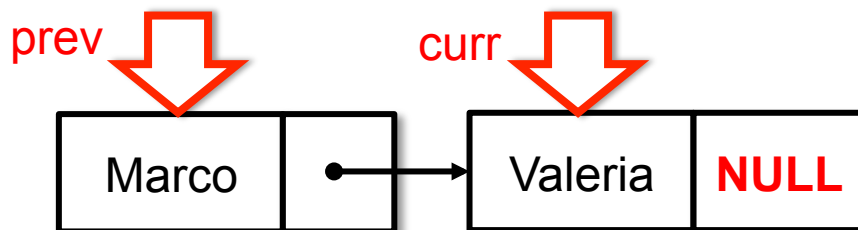
```
struct list_node {  
    struct list_node *next;  
    int age;  
    char user_id[MAX_LEN + 1];  
    char mail[MAX_LEN + 1];  
};
```

Il nostro programma implementerà le seguenti funzioni

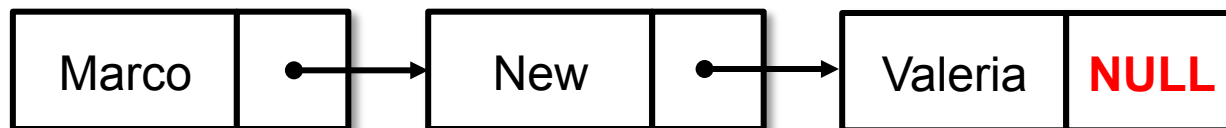
- Inserisci in lista ordinata
- Funzione predicato “nome utente già presente”
- Cancella per user id
- Stampa la lista
- Funzione predicato “lista vuota”

INSERISCI IN ORDINE DI ETÀ

- Per inserire un elemento eseguiamo le seguenti operazioni:
 - Allochiamo la memoria per il nuovo elemento
 - Scorriamo la lista con 2 puntatori a due nodi contigui
 - Una volta trovata la posizione corretta aggiorniamo i puntatori come in figura



L'aggiornamento è semplicemente $\rightarrow \text{prev.next} = \text{new}; \text{new.next} = \text{curr}$



INSERISCI IN ORDINE

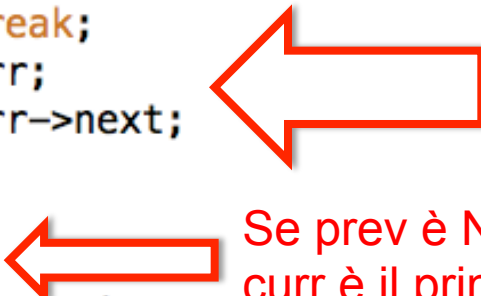
NOTA: l'allocazione è fatta fuori dalla funzione inserisci

```
void insert_ordered_by_age(struct list_node **head, struct list_node *element){
    struct list_node *curr=*head, *prev=NULL;

    while(curr != NULL) {
        if (element->age <= curr->age)
            break;
        prev = curr;
        curr = curr->next;
    }

    if (prev == NULL)
        *head = element;
    else
        prev->next = element;

    element->next = curr;
}
```



Se la lista non è vuota scorro i due puntatori. Il nuovo elemento verrà inserito tra prev e curr

Se prev è NULL stiamo nel caso lista vuota o curr è il primo elemento (vedi posizione break)

CANCELLA PER USER ID

```
void delete_by_user_id(struct list_node **head, char *user_id){
    struct list_node *curr=*head, *prev=NULL;

    while(curr != NULL) {
        if (!strcmp(curr->user_id, user_id)) {
            if (prev == NULL)
                *head = curr->next;
            else
                prev->next = curr->next;

            free(curr);
            return;
        }
        prev = curr;
        curr = curr->next;
    }
    printf("Elemento non trovato\n");
}
```

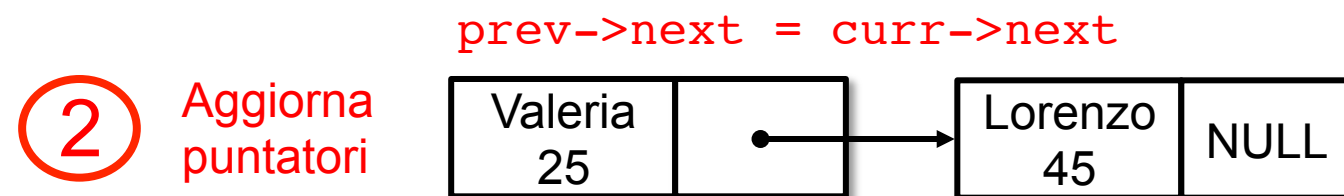
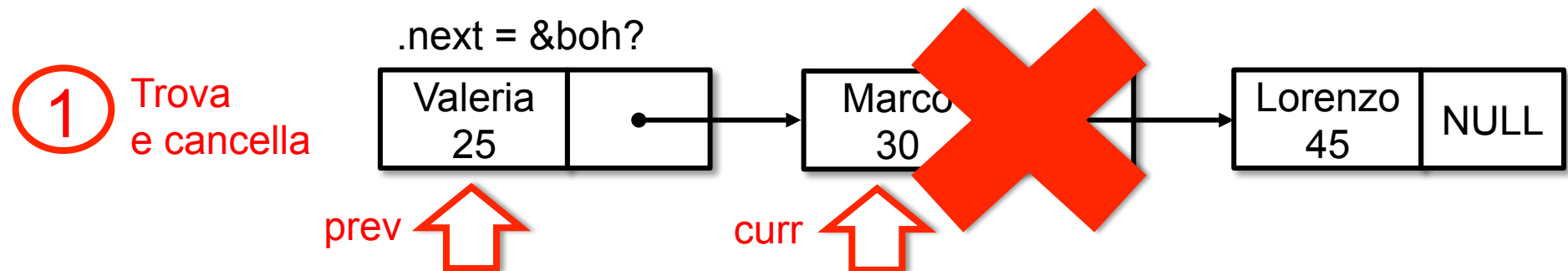
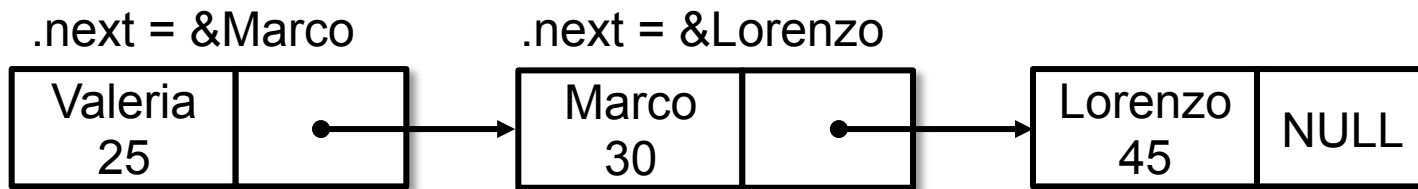
Se la lista è vuota non entra nel while

Se sì, curr è il primo elemento

Se arriva qui significa che non ha mai verificato la condizione

ESEMPIO CANCELLA PER USER ID

Cancelliamo Marco nella seguente lista



ALTRE FUNZIONI

- La funzione “**utente già presente**” si realizza facendo scorrere un puntatore lungo tutta lista. Non appena si trova l’elemento cercato si interrompe l’esecuzione della funzione e si ritorna 1 (TRUE). Se invece si scorre tutta la funzione senza trovare l’elemento si ritorna 0 (FALSE)
- La funzione “**stampa lista**” è simile alla precedente. Si fa scorrere un puntatore lungo tutta la lista e per ogni elemento si stampano i campi della struct. La funzione finisce quando la fine della lista è raggiunta
- La funzione “**lista vuota**” ritorna 1 (TRUE) se il puntatore alla testa della lista è NULL

UTENTE GIÀ IN LISTA

```
int user_id_already_in_list(struct list_node *head, char* user_id) {  
    struct list_node *tmp = head;  
  
    while(tmp != NULL) {  
        if (!strcmp(tmp->user_id, user_id))  
            return 1;  
        tmp = tmp->next;  
    }  
  
    return 0;  
}
```

Se arriva qui significa
che non ha mai verificato
la condizione

STAMPA LISTA

```
void print_list(struct list_node* head) {
    if (is_empty(head))
        printf("lista vuota\n");
    else {
        struct list_node *tmp = head;
        while (tmp != NULL) {
            printf("user id %s, mail %s, età %d\n",
                  tmp->user_id, tmp->mail, tmp->age);
            tmp = tmp->next;
        }
    }
}
```

LISTA VUOTA

```
int is_empty(struct list_node *head) {  
    if (head == NULL) return 1;  
    else return 0;  
}
```

Come si può scrivere in una sola riga
questa funzione?

PILE

CONCETTI BASE

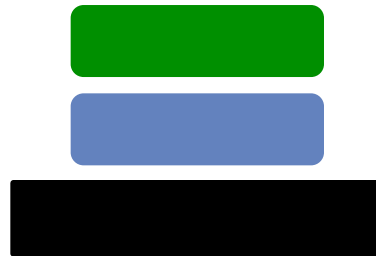
- Le pile (“stack”) sono dei casi particolari di liste in cui gli inserimenti e le cancellazioni possono avvenire soltanto in “cima” alla lista
- Le pile sono strutture dati di tipo **LIFO** (Last In First OUT) in cui i primi nodi a uscire sono gli ultimi inseriti
 - Immaginate una pila di piatti sporchi
 - Il primo piatto ad essere lavato e messo in dispensa è l’ultimo che abbiamo “impilato” nel lavandino
 - Esempio reale: STACK dei programmi in esecuzione
- Come per le liste ordinate, una pila mantiene un puntatore al primo elemento (**HEAD**)
- Le funzioni base implementate dalle pile sono
 - **PUSH**: inserisce un nodo in cima alla pila
 - **POP**: rimuove il primo nodo della pila

ESEMPIO OPERAZIONI BASE

PUSH



PUSH



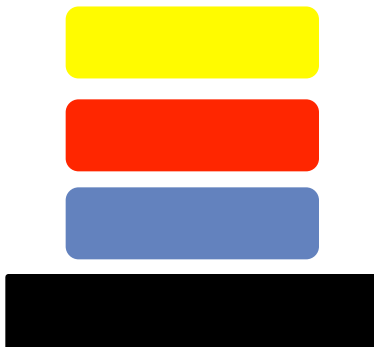
POP



PUSH



PUSH



PUSH



POP



POP



FUNZIONE PUSH

- Le funzione `push ()` prende il puntatore all'area di memoria allocata dinamicamente è contenente il nuovo elemento e lo posiziona in cima alla lista
 - Questo viene fatto esattamente come in un inserimento in cima alla lista
- Il puntatore `next` del nuovo elemento viene settato all'indirizzo puntato dal campo `next` dell'elemento puntato da `HEAD`
 - Se la lista è vuota il campo `next` del nuovo elemento è settato a `NULL`
- Il puntatore `HEAD` viene aggiornato con la posizione del nuovo elemento

FUNZIONE PUSH

NOTA: l'allocazione è fatta fuori dalla funzione push ()

```
void push(struct stack_node **head, struct stack_node *element) {  
    if (is_empty(*head)) {  
        *head = element;  
        element->next = NULL;  
    }  
    else {  
        element->next = *head;  
        *head = element;  
    }  
}
```

FUNZIONE POP

- Le funzione `pop()` ritorna il puntatore al primo elemento
- Il valore del puntatore `HEAD` viene aggiornato con l'indirizzo del secondo elemento
- Se la lista contiene un solo elemento `HEAD` viene settato a `NULL`
- L'area di memoria ritornata dalla funzione `pop()` viene così "staccata" dalla pila
- Una volta "processata", quest'area di memoria può essere liberata con una `free()`
 - Una volta uscita dalla funzione che ha invocato la `pop()` si perderà il riferimento a questa area di memoria
 - **ATTENZIONE AL MEMORY LEAK**

FUNZIONE POP

NOTA: la cancellazione dell'elemento ritornato dalla `pop()` deve essere fatta fuori dalla funzione chiamando `free()` sul puntatore ritornato

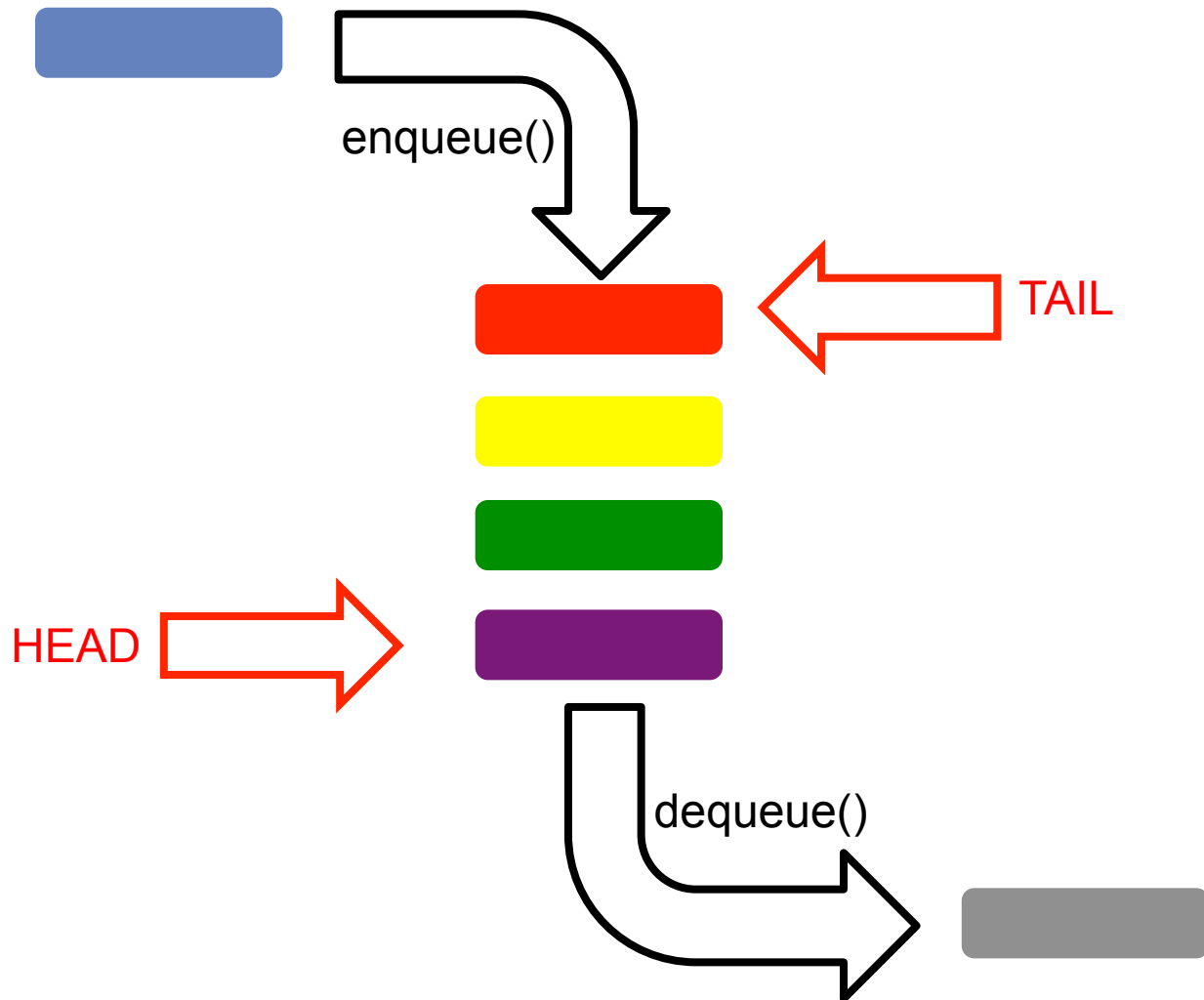
```
struct stack_node * pop(struct stack_node **head){  
    struct stack_node *ret = *head;  
  
    if (is_empty(*head))  
        return NULL;  
    else  
        *head = ret->next;  
  
    return ret;  
}
```

CODE

CONCETTI BASE

- Le code sono un'altra implementazione “vincolata” di liste
- Nell code gli inserimenti vengono effettuati in coda e le rimozioni in testa
- Le code sono struttura dati di tipo **FIFO** (First In First Out) in cui il primo elemento inserito è il primo a essere rimosso
- Esempio di code sono i buffer delle schede di rete dei computer oppure le liste di documenti da processare in una stampante
- A differenza delle pile, per le code vengono mantenuti due puntatori
 - HEAD: puntatore al primo elemento della coda
 - TAIL: puntatore all'ultimo elemento della coda
- Le funzioni base delle code sono:
 - ENQUEUE: inserimento di un elemento nella coda
 - DEQUEUE: estrazione di un elemento dalla coda

ESEMPIO OPERAZIONI BASE



FUNZIONE ENQUEUE

- Le funzione enqueue () prende il puntatore all'area di memoria allocata dinamicamente e contenente il nuovo elemento e lo posiziona alla fine alla coda
 - Il campo `next` dell'ultimo elemento viene settato all'indirizzo del nuovo elemento
 - Il campo `next` del nuovo elemento (che ora è diventato l'ultimo) viene settato a NULL
 - TAIL punterà al nuovo elemento
 - Se la lista è vuota HEAD punterà al nuovo elemento

FUNZIONE ENQUEUE

NOTA: l'allocazione è fatta fuori dalla funzione enqueue ()

```
void enqueue(struct queue_node **head,  
             struct queue_node **tail, struct queue_node *element) {  
  
    if (is_empty(*head))  
        *head = element;  
    else  
        (*tail)->next = element;  
  
    *tail = element;  
    element->next = NULL;  
  
}
```

FUNZIONE DEQUEUE

- Le funzione dequeue () ritorna il puntatore HEAD della lista
- Questa funzione equivale alla pop ()
 - Se la lista contiene più di un elemento HEAD punterà al secondo elemento della coda e TAIL rimane invariato
 - Se la lista contiene un solo elemento HEAD e TAIL punteranno a NULL
- L'area di memoria ritornata dalla funzione dequeue () viene così “staccata” dalla coda
- Una volta “processata”, quest'area di memoria può essere liberata con una free ()
 - **ATTENZIONE AL MEMORY LEAK (come per le pile)**

FUNZIONE DEQUEUE

NOTA: la cancellazione dell'elemento ritornato dalla `pop()` deve essere fatta fuori dalla funzione chiamando `free()` sul puntatore ritornato

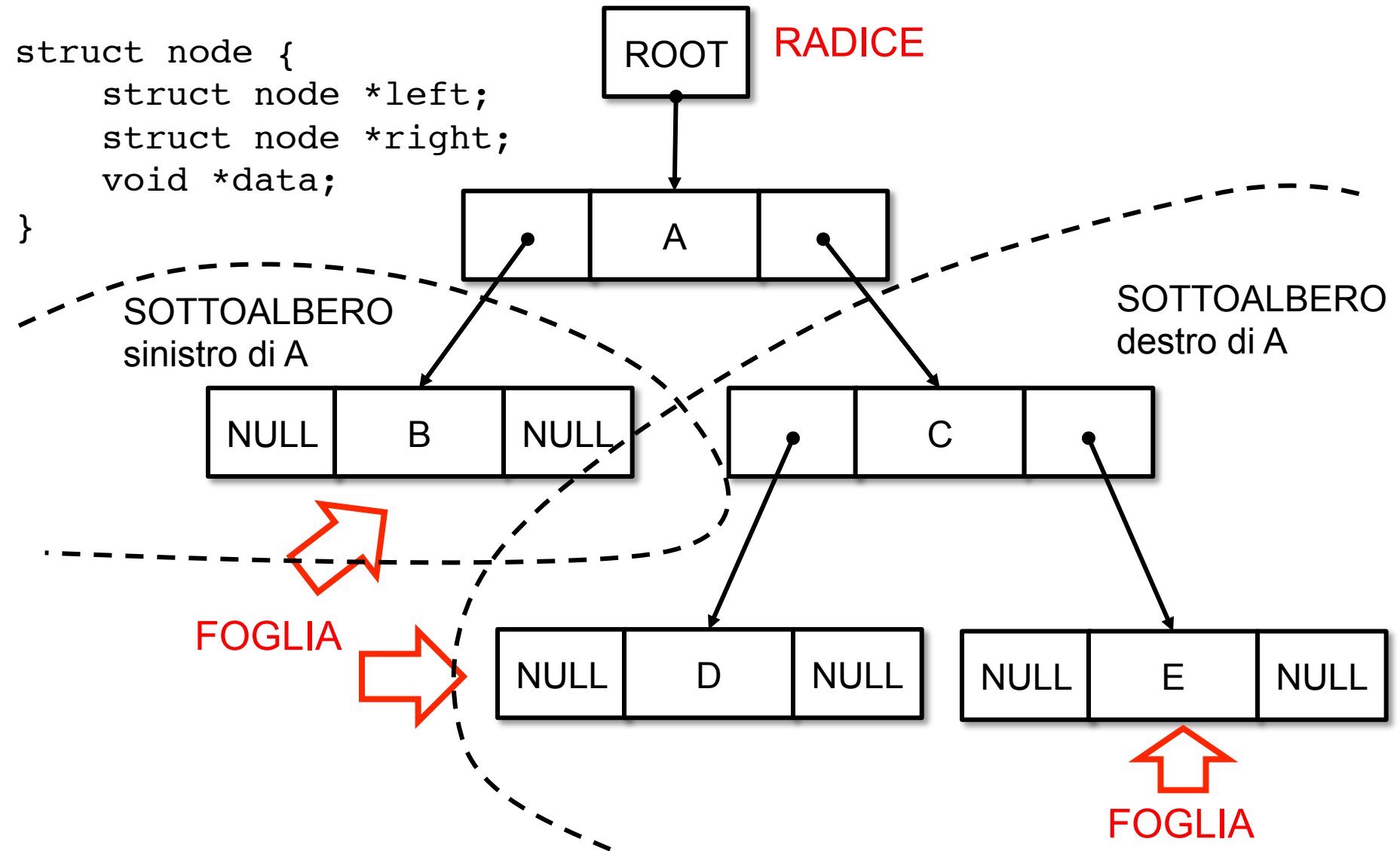
```
struct queue_node * dequeue(struct queue_node **head,  
                             struct queue_node **tail){  
    struct queue_node *ret = *head;  
  
    if (is_empty(*head))  
        return NULL;  
    else  
        *head = ret->next;  
  
    if (*head == NULL)  
        *tail = NULL;  
  
    return ret;  
}
```

ALBERI

CONCETTI BASE

- Un albero è una struttura dati non lineare
- Ogni nodo di un albero contiene due o più collegamenti
 - In questo corso ci **concentreremo solo su alberi binari**, ovvero alberi in cui i nodi hanno solo 2 collegamenti
- Il *nodo radice* (ROOT) è il primo nodo in un albero
- Ogni nodo può avere a due *figli* (due nodi collegati), uno sinistro (primo nodo del sotto albero sinistro) e uno destro (primo nodo del sottoalbero destro)
- Un nodo che non ha figli è detto *nodo foglia*
- Tutti i nodi tranne il ROOT hanno necessariamente un padre

ALBERO BINARIO

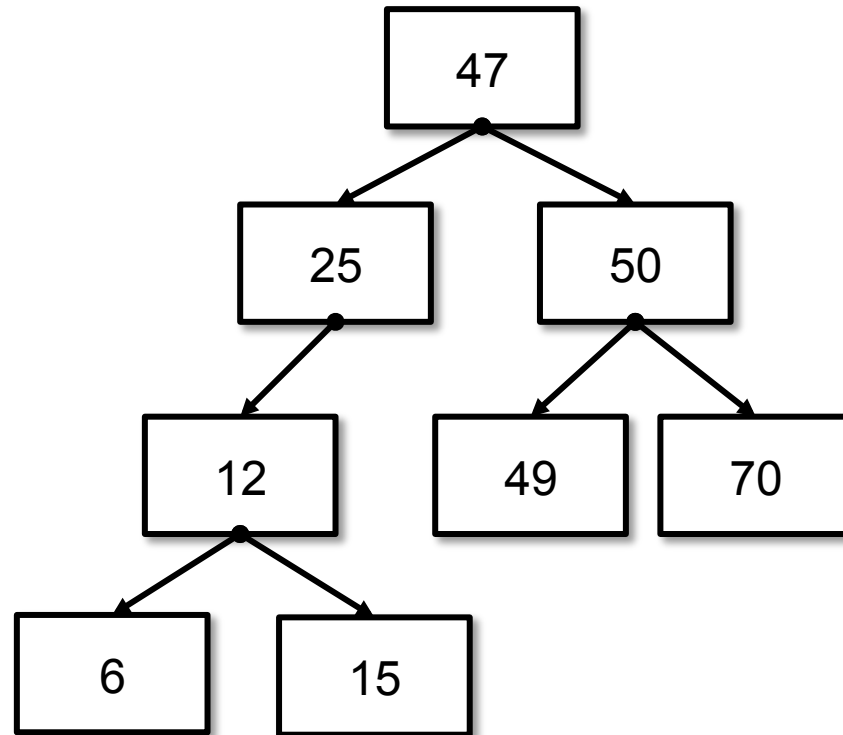


CONCETTI BASE

- In questo corso vedremo esclusivamente gli ***alberi di ricerca binaria***
 - Indicizzati da una chiave univoca (non ci sono duplicati)
 - Tutti i valori di un sotto albero sinistro sono minori del valore della chiave del nodo padre
 - Tutti i valori di un sotto albero destro sono maggiori del valore della chiave del nodo padre

ALBERO DI RICERCA BINARIO

```
struct tree_node {  
    struct node *left;  
    struct node *right;  
    int key;  
    void *data;  
}
```

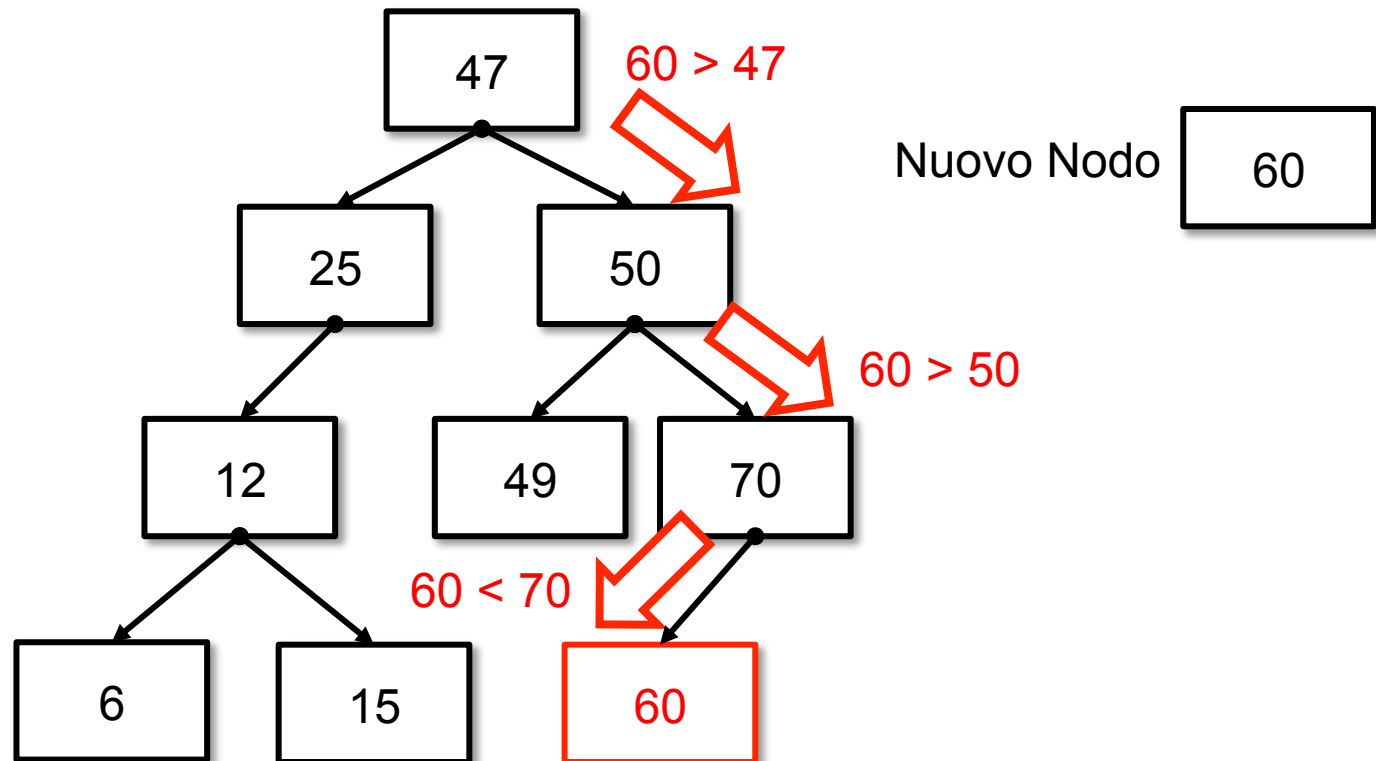


INSERIMENTO

A partire dal nodo radice si esplora l'albero prendendo ad ogni livello

- “la strada sinistra” se la nuova chiave è minore del valore del nodo corrente
- “la strada destra” viceversa

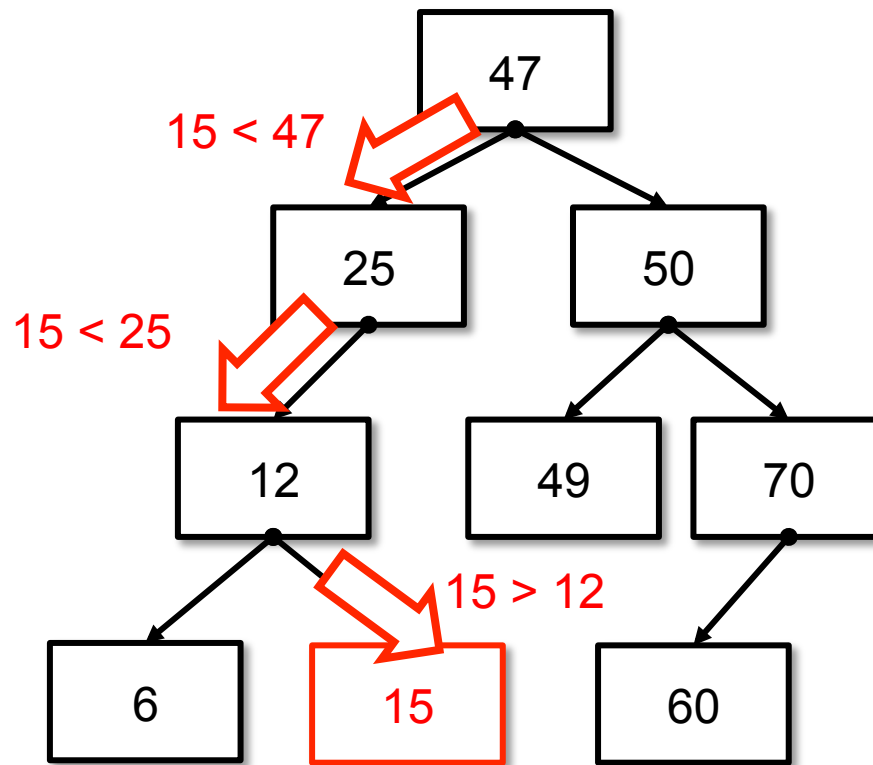
Il processo termina quando si raggiunge una foglia



RICERCA

In maniera del tutta analoga all'inserimento, la ricerca di una chiave avviene esplorando ricorsivamente il sottoalbero sinistro di ciascun nodo se la chiave è minore del nodo corrente, destro altrimenti

TROVA IL NODO CON CHIAVE 15



ALBERO BILANCIATO

- Se l'albero è BILANCIATO (vedi dopo) la complessità di inserimento è ricerca per un albero con N nodi è:

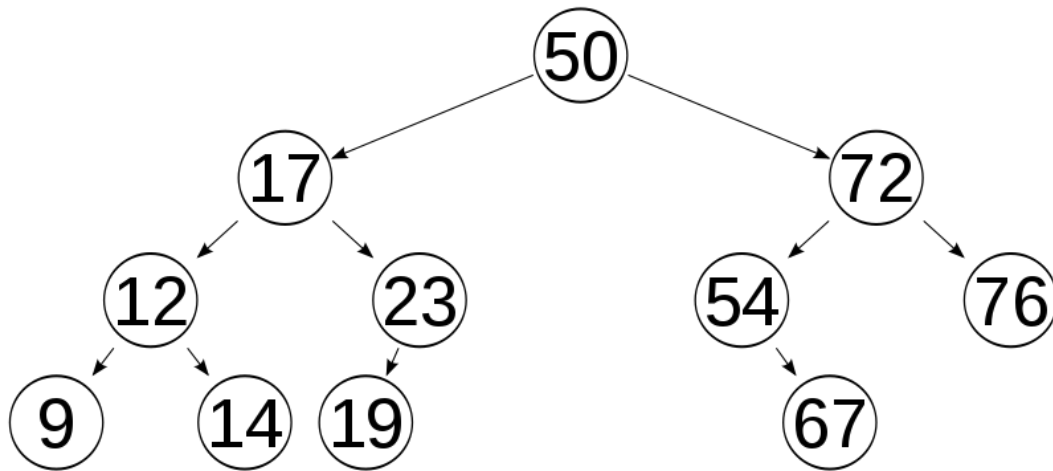
$$O(\log_2(N))$$

- Mentre per una lista collegata di N nodi la complessità è

$$O(N)$$

- Il problema è che inserzioni e cancellazioni possono sbilanciare l'albero
 - Provate a inserire in ordine: 7, 8, 9, 10, 11, 12
- Questo problema è risolto dagli alberi autobilanciati (che non saranno coperti da questo corso)
 - Per chi volesse interessarsi: http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

ALBERO BILANCIATO



Bilanciato: l'altezza è la minima altezza di un albero con N elementi, ovvero l'intero inferiore di $\log_2(N)$

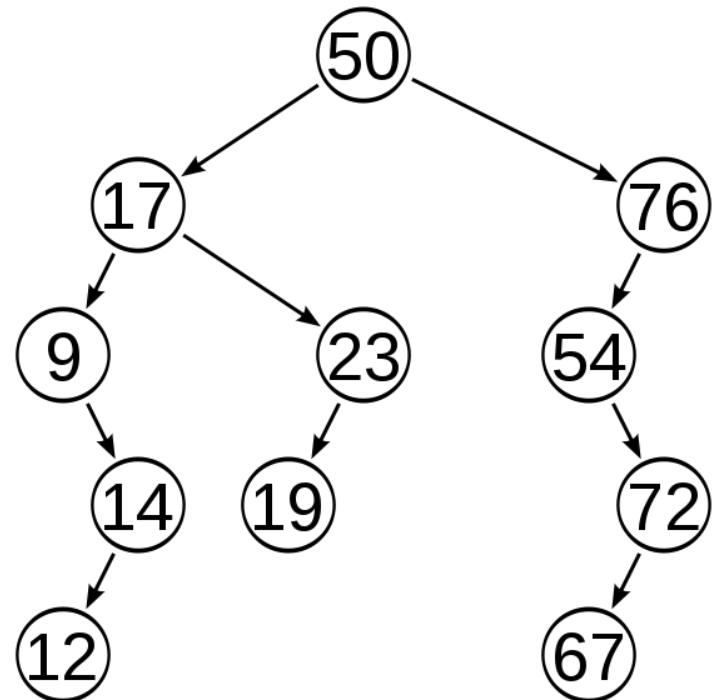
$$h = 3$$

$$\log_2(11) = 3,45$$

Non Bilanciato

$$h = 4$$

$$\log_2(11) = 3,45$$



INSERIMENTO

- Un nodo può essere inserito solo come foglia
- La funziona opera ricorsivamente a partire dalla radice
 - Se il puntatore corrente è NULL questo viene fatto puntare al nuovo elemento. I campi left e right del nuovo elemento saranno messi a NULL
 - Se l'albero non è vuoto viene ricorsivamente chiamata la stessa funzione di inserimento aggiornando il puntatore corrente con il valore del figlio sinistro se la chiave è minore, altrimenti con il valore del figlio destro
 - Le chiamate ricorsive terminano quando si raggiunge un nodo foglia

STRUTTURA BASE

```
struct tree_node {  
    struct tree_node *left;  
    struct tree_node *right;  
    int key;  
    int val;  
};
```



Chiave univoca di ricerca

INSERIMENTO

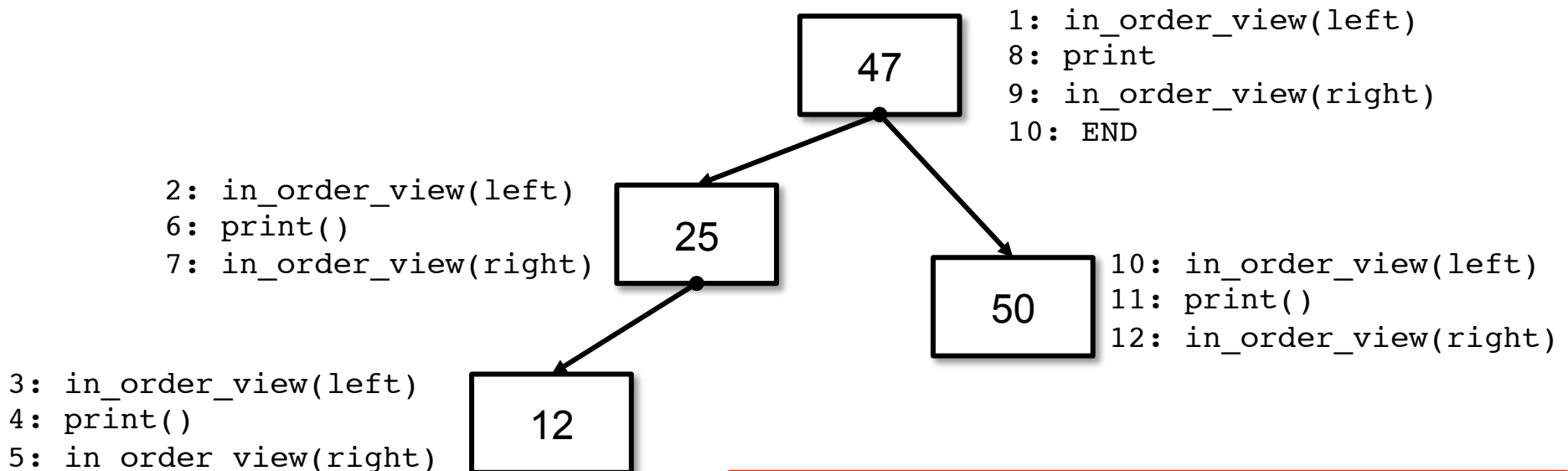
```
void insert(struct tree_node **tree,
            struct tree_node* new) {
    if (*tree == NULL) {
        *tree = new;
        (*tree)->left = NULL;
        (*tree)->right = NULL;
    }
    else {
        if (new->key < (*tree)->key)
            insert(&(*tree)->left, new);
        else if (new->key > (*tree)->key)
            insert(&(*tree)->right, new);
        else
            printf("Chiave duplicata\n");
    }
}
```


RICERCA

```
struct tree_node * find_by_key(struct tree_node *tree, int key){
    if (tree == NULL)
        return NULL;
    else {
        if (key < tree->key)
            return find_by_key(tree->left, key);
        else if (key > tree->key)
            return find_by_key(tree->right, key);
        else
            return tree;
    }
}
```

VISTA

```
void in_order_view(struct tree_node* tree) {  
    if (tree != NULL) {  
        in_order_view(tree->left);  
        printf("Key %d, value %d\n",  
              tree->key, tree->val);  
  
        in_order_view(tree->right);  
    }  
}
```



Sequenza di print: 12, 25, 47, 50

NOTA SU FUNZIONI RAND E TIME (USATE NEI PROGRAMMI DI TEST)

```
#include <time.h>

time_t time (time_t* timer);
```

Ritorna il current time e fa puntare timer all'oggetto tempo corrente in libreria. Questo valore può essere usato come seme della sequenza pseudo casuale (attenzione cambia con bassa frequenza)

```
#include <stdlib.h>

void srand(unsigned int seed);

int rand(void);
```

srand() setta il seme delle sequenza pseudocasuale
rand() ritorna un intero casuale nel range [0, RAND_MAX]

**NOTAZIONE “O
GRANDE”**

NOTAZIONE “O GRANDE”

- La notazione $O()$ indica come cresce il tempo di esecuzione di un algoritmo in relazione al numero di elementi elaborati, generalmente indicati con n
- In altre parole indica l'ordine di grandezza del numero di operazioni fondamentali al crescere di n
- Esempi, dato un array con n elementi: `int array[n];`
 - 1) Verificare se il primo elemento è uguale al secondo
 - 2) Verificare se il primo e il secondo elemento di un array sono uguali al terzo
 - 3) Verificare se il primo elemento è uguale a uno qualsiasi degli altri elementi dell'array
 - 4) Verificare se ci sono duplicati in un array

TEMPO DI ESECUZIONE COSTANTE

L'esempio (1) richiede 1 sola operazione, indipendentemente da quanti siano gli elementi

```
return array[0] == array[1]
```

Questo algoritmo ha complessità $O(1)$

L'esempio (2) richiede invece 2 confronti, indipendentemente da quanti siano gli elementi dell'array

```
if array[0] == array[1]
    if array[1] == array[2]
        return 1
return 0
```

Anche il secondo esempio ha complessità $O(1)$ anche se richiede 2 confronti. $O(1)$ indica che il tempo di esecuzione è indipendente dal numero di elementi

TEMPO DI ESECUZIONE LINEARE

L'esempio (3) richiede $n - 1$ operazioni di confronto

```
for (i=1; i<n; i++) {  
    if array[0] == array[i+1];  
        return 1;  
}
```

La costante è irrilevante (siamo interessati al comportamento “asintotico”, più cresce n , più “1 è piccolo”)

L'algoritmo è caratterizzato da un tempo di esecuzione $O(n)$, o *lineare*

TEMPO DI ESECUZIONE QUADRATICO

L'esempio 3 può essere realizzato con il seguente codice

```
for (i=0; i<n; i++) {  
    for (j=i+1; j<n; j++) {  
        if array[i] == array[j];  
            return 1;  
    }  
}  
return 0;
```

Che richiede il seguente numero di operazioni:

$$(n-1) + (n-2) + (n-3) + \dots + 2 + 1 = n^2 - (n^2+n)/2 = n^2/2 - n/2$$

Siccome siamo interessati all'andamento asintotico, la componente lineare è trascurabile rispetto a quella quadratica, e il coefficiente $\frac{1}{2}$ è trascurabile

Questo algoritmo ha un tempo di esecuzione $O(n^2)$, o quadratico

ALGORITMI DI ORDINAMENTO

ORDINAMENTO PER SELEZIONE (SELECTION SORT)

- Algoritmo semplice ma poco efficiente
- Complessità quadratica
- Al primo passo seleziona l'elemento più piccolo dell'array e lo scambia con il primo
- Al secondo passo selezione il secondo elemento più piccolo dell'array e lo scambia con il secondo
 - Il secondo più piccolo equivale al primo più piccolo del "sotto-array" che parte dal secondo elemento
- E così via...

Esempio: (5, 23, 56, 3, 4)

Passo 1: scambia 5 con 3 → (3, 23, 56, 5, 4)

Passo 2: scambia 23 con 4 → (3, 4, 56, 5, 23)

Passo 3: scambia 56 con 5 → (3, 4, 5, 56, 23)

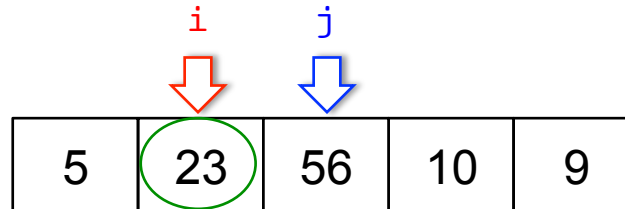
Passo 4: scambia 56 con 23 → (3, 4, 5, 23, 56)

SELECTION SORT: IMPLEMENTAZIONE

```
void selection_sort(int *array, int len) {  
    int i, j, idx_min;  
  
    for (i=0; i<len-1; i++) {  
        idx_min = i;  
  
        for (j=i+1; j<len; j++) {  
            if (array[j] < array[idx_min])  
                idx_min = j;  
        }  
  
        __swap_elements(array, i, idx_min);  
        print_algorithm_step(array, len, i, idx_min);  
    }  
}
```

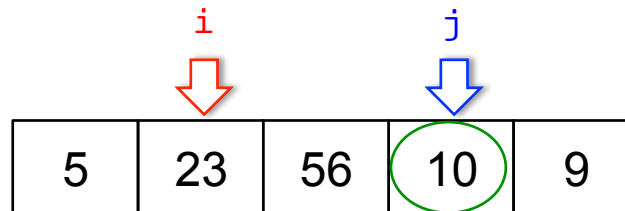
SELECTION SORT: N-ESIMO PASSO

$i=1, j=2$



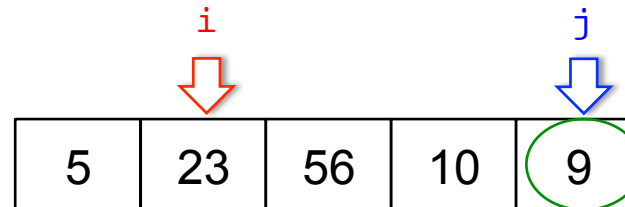
$idx_min = 1$

$i=1, j=3$



$idx_min = 3$

$i=1, j=4$



$idx_min = 4$

SWAP (1 , 4)



SELECTION SORT: IMPLEMENTAZIONE

```
static void __swap_elements(int *array, int idx1, int idx2) {  
    int tmp = array[idx1];  
  
    array[idx1] = array[idx2];  
    array[idx2] = tmp;  
}
```

SELECTION SORT: IMPLEMENTAZIONE

```
void print_algorithm_step(int *array, int len, int step, int index) {
    int i;

    printf("step %d:\n", step);

    for (i=0; i<len; i++) {
        if (i != index)
            printf("\t%d", array[i]);
        else
            printf("\t%d*", array[i]);
    }
    printf("\n");
}
```

ORDINAMENTO PER INSERZIONE (INSERTION SORT)

- Algoritmo semplice ma poco efficiente
- Complessità quadratica
- L'algoritmo parte dal secondo elemento e se $(array[1] < array[0])$ lo scambia
- Al secondo passo considera il terzo elemento e lo inserisce nella posizione corretta rispetto ai primi 2
- Ripete fino a che non viene raggiunto l'ultimo elemento

Esempio: (5, 23, 56, 3, 4)

Passo 1: resta invariato $\rightarrow (5, 23, 56, 3, 4)$

Passo 2: resta invariato $\rightarrow (5, 23, 56, 3, 4)$

Passo 3: scambia 5 con 3 $\rightarrow (3, 5, 23, 56, 4)$

Passo 4: scambia 56 con 23 $\rightarrow (3, 4, 5, 23, 56)$

INSERTION SORT: IMPLEMENTAZIONE

```
void inserction_sort(int *array, int len) {  
    int i, j, current;  
  
    for (i=1; i<len; i++) {  
        j = i;  
        current = array[i];  
  
        while((j > 0) && (array[j-1] > current)) {  
            array[j] = array[j-1];  
            j--;  
        }  
  
        array[j] = current;  
        print_algorithm_step(array, len, i, j);  
    }  
}
```


INSERTION SORT: N-ESIMO PASSO

```
while ((j>0) && (array[j-1]>current)){  
    array[j] = array[j-1]; //sposta a destra  
    j--;  
}
```

i = 5, j = 5, current = 32

12	20	50	55	68	32
----	----	----	----	----	----

Diagram illustrating the initial state of the array [12, 20, 50, 55, 68, 32]. The current element 32 is at index j=5. The element 68 at index j-1=4 is being compared to 32. Red and blue arrows indicate the comparison and the potential shift of elements to the right.

i = 5, j = 4, current = 32

12	20	50	55	68	68
----	----	----	----	----	----

Diagram illustrating the state of the array after the first shift. The element 68 has been shifted to index j=5. The current element 32 is still at index j=4. Red and blue arrows indicate the comparison and the potential shift of elements to the right.

i = 5, j = 3, current = 32

12	20	50	55	55	68
----	----	----	----	----	----

Diagram illustrating the state of the array after the second shift. The element 55 has been shifted to index j=4. The current element 32 is still at index j=3. Red and blue arrows indicate the comparison and the potential shift of elements to the right.

i = 5, j = 2, current = 32

12	20	50	50	55	68
----	----	----	----	----	----

Diagram illustrating the state of the array after the third shift. The element 50 has been shifted to index j=3. The current element 32 is still at index j=2. Red and blue arrows indicate the comparison and the potential shift of elements to the right.

INSERTION SORT: N-ESIMO PASSO

```
while ((j>0) && (array[j-1]>current)){  
    array[j] = array[j-1]; //sposta a destra  
    j--;  
}
```

i = 5, tmp = 2, current = 32

	tmp-1	tmp			
	↓	↓			
12	20	50	50	55	68

STOP perché $20 < 32$, ovvero
 $\text{array}[\text{tmp}-1] < \text{current}$

Assegnazione finale
 $\text{array}[\text{tmp}] = \text{current}$

12	20	32	50	55	68
----	----	----	----	----	----

ORDINAMENTO PER FUSIONE (MERGE SORT)

- Algoritmo più complicato ma anche più efficiente
- Basato sul paradigma Divide & Conquer
 - Si divide ricorsivamente il problema in sottoproblemi dello stesso tipo finché non risultano risolvibili in maniera diretta
- Merge sort si può concettualmente descrivere con i seguenti passi
 - Si suddivide la lista non ordinata in n sottoliste da 1 elemento
 - Si fondono ricorsivamente le sottoliste in liste ordinate fino a che non si ottiene una lista unica

MERGE SORT RICORSIVO (TOP DOWN)

