ALGO LAB MUST KNOW

TYPES IN C

Tipo	Dimensione in byte	
	32 Bit	64 bit
char	1	1
short	2	2
int	4	5
long	4	8
long long	8	8
float	4	4
double	8	8
Pointer	4	8

- `sizeof(a)` rappresenta il numero di byte necessari a memorizzare la variabile a.
- i tipi in virgola mobile sono
  - float = single-precision
  - double = double-precision
  - long double = extended-precision
  - Il valore di un char cambia a seconda della charset della macchina.

- `typedef [tipo] [nuovo_tipo]` serve a definire nuovi tipi

- per stampare variabili si usano delle stringhe di formato che usano delle specifiche di formato:
  - %d = interi
  - %f = float in notazione decimale
  - %e = float in notazione esponenziale
  - %c = per i char

ARRAY

- Dimensione di un array = `sizeof(array)/sizeof(a[0])`
- Array costante = `const int Array[5] = {1,2,3,4,5}`
- la dimensione dell'array va fissata in fase di compilazione
- Le stringhe in C sono array di char, terminati dal carattere di fine stringa (\0)
  - La lunghezza di una stringa `e data dal numero dei suoi caratteri + 1

STRUTTURE

- Ogni variabile strutturata è composta da membri
- Ci sono due modi per definire tipi strutturati:
  - Tramite tag:

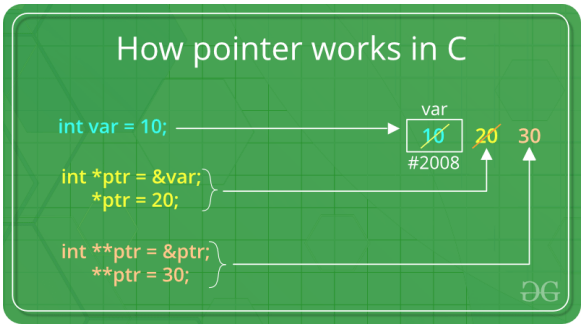
```
struct studente {
    char nome [10];
    char cognome [10];
    int anno ;
};
struct studente stud1 , stud2 ;
```

  - Tramite typedef

```
typedef struct {
    char nome [10];
    char cognome [10];
    int anno ;
}Studente ;

Studente stud1,stud2;
```

PUNTATORI



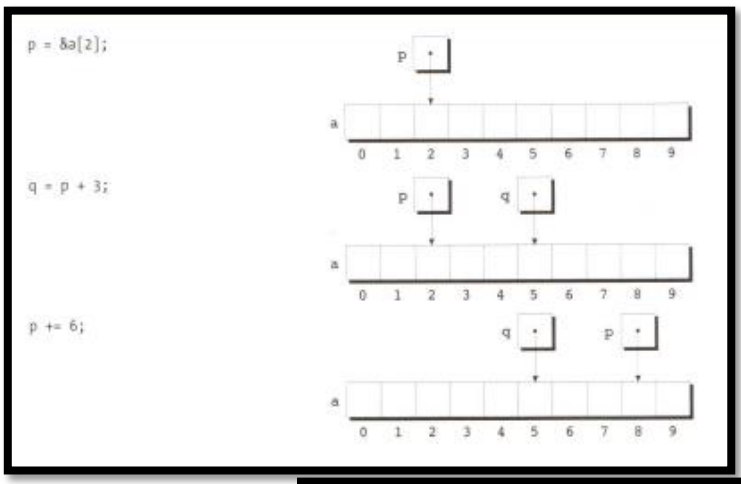
- variabili che hanno come valore un indirizzo di memoria
- & serve ad ottenere l'indirizzo di memoria di una variabile:
  `int i, * p;` = dichiaro il puntatore a intero p
  `p = & i;` = assegno a p l'indirizzo di i

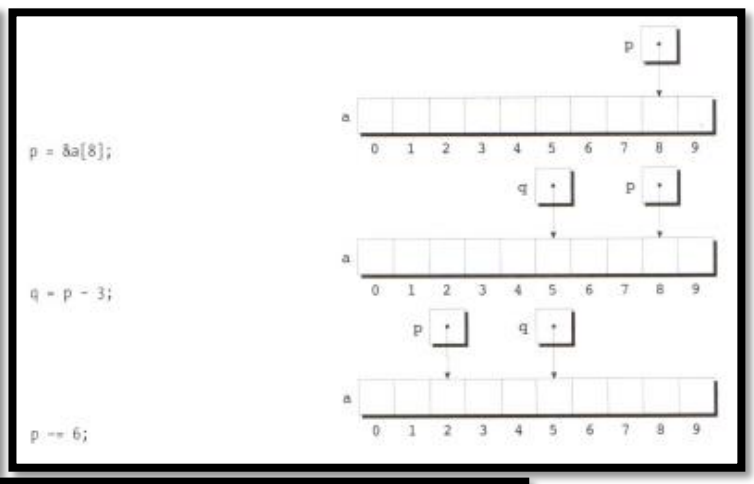
- Se un puntatore p non è inizializzato, il suo valore non è definito, quindi non è definito nemmeno *p; p potrebbe puntare ad uno spazio di memoria qualsiasi
  → **Segmentation Fault**

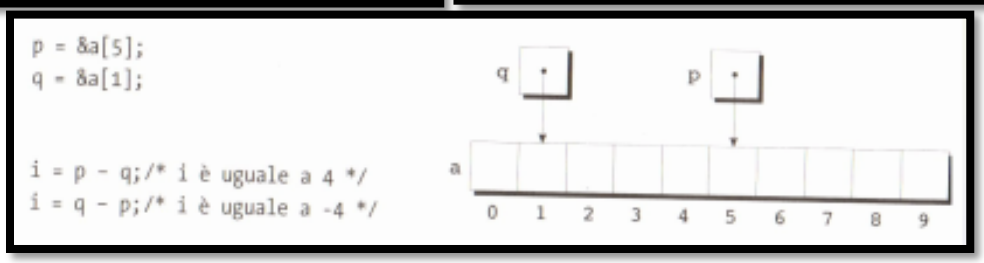
- il seguente codice effettua una copia di puntatori

```
int i, j, *p, * q;
p = &i;
q = p; // modificando il val di *p si cambia anche quello di *q
```

- per accedere ai membri di una struttura usando un puntatore si utilizza " -> "
- **Aritmetica dei puntatori**:







ALLOCAZIONE DINAMICA

- le variabili con storage duration **static** sono allocate in memoria principale all'inizio dell'esecuzione del programma e persistono per tutta l'esecuzione del programma (es: variabili globali)
- le variabili con storage duration **automatic** sono allocate sullo stack (es: variabili di una funzione)
- la memoria può essere allocata anche in modo dinamico, viene allocata in questo caso nello **Heap**
    - è possibile accedere a tali blocchi di memoria attraverso l'uso di puntatori
    - bisogna deallocarla manualmente
    - 4 funzioni fondamentali:
        - `void* malloc(size_t size);`
        - `void* calloc(size_t nmemb, size_t size);`
        - `void* realloc(void *p, size_t size);`
        - `void free(void * p);`

    - **Errore dangling pointer** = Dopo la chiamata free(p), il blocco di memoria puntato da p viene deallocato, ma il valore del puntatore p non cambia; eventuali usi successivi di p possono causare danni
    - **Memory leak** = se ho p = malloc (…) e q=malloc(…) e ad un certo punto faccio p=q, l'oggetto puntato da p prima dell'ultimo assegnamento non è piu raggiungibile. Quel blocco è allocato ma non utilizzabile.

LISTE CONCATENATE

- catena di strutture chiamate nodi. Ogni nodo contiene un puntatore al prossimo nodo.
- l'ultimo nodo contiene il puntatore nullo
- Ha la seguente struttura:

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

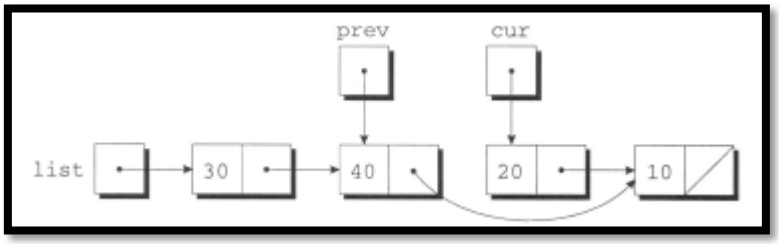
- **Inserimento** in testa: *Costo: O(1)*

```
struct node* add_to_list (struct node* list, int n ){
    struct node * new_node;
    new_node = my_malloc (sizeof(struct node));
    new_node -> value = n;
    new_node -> next = list;
    return new_node;
}
```

- **Ricerca** di un nodo: *Costo: O(n)*

```
struct node * search_list (struct node* list, int n){
    struct node * p;
    for(p = list ; p != NULL ; p = p -> next)
        if(p -> value == n)
            return p;
    return NULL;
}
```

- **Cancellazione** di un nodo: *Costo: O(n)*
  - Trovare il nodo da eliminare
  - Modificare il nodo precedente in modo che punti al nodo successive a quello da cancellare
  - liberare con free lo spazio occupato dal nodo cancellato



LISTE DOPPIAMENTE CONCATENATE

- come le liste concatenate singole, sono dei nodi, con unica differenza che hanno anche un puntatore al nodo precedente.
- **Inserimento** in testa: *Costo: O(1)*

```
struct node* add_to_list(int n, struct node* list){
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data=n;
    new_node->prev=NULL;
    new_node->next = head;
    head->prev=new_node;
    head=new_node;
    return new_node;
}
```

- **Ricerca** elemento: *Costo: O(n)*

```
struct node* search(struct node* list, int item){
    while (list!=NULL)  {
        if(list->data == item)
            return list;
        list = list -> next;
    }
    return NULL;
}
```

- **Cancellazione** all'inizio: *Costo: O(1)*

```
void beginning_delete(struct node* first){
    if(first == NULL)
        return;
    if(first->next == NULL)
        free(first);
    else{
        first = first->next;
        free(first->prev);
        first->prev = NULL;
    }
}
```

- **Cancellazione** alla fine: *Costo: O(1)*

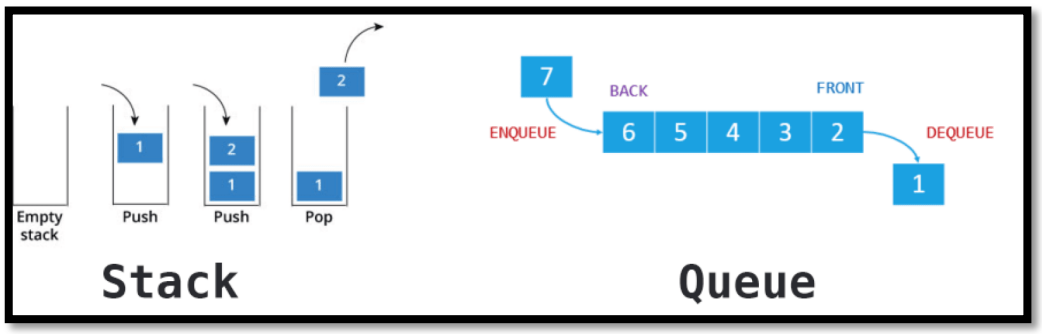
```
void last_delete(struct node* last){
    if(last == NULL)
        return;
    else if(last->prev == NULL)
        free(last);
    else(last->next == NULL){
        last=last->prev;
        free(last->next);
        last->next=NULL;
    }
}
```

- **Cancellazione** dato un valore *Costo: O(n)*

```
void delete_specified(int item)  {
    struct node *ptr, *temp;
    temp=search(first,item);    //O(n)
    if(temp==NULL)
        return;
    if(temp->prev==NULL && temp->next==NULL)
        free(temp);
    if(temp->prev==NULL){
        temp=temp->next;
        free(temp->prev);
        temp->prev=NULL;
    }
    if(temp->next==NULL){
        temp=temp->prev;
        free(temp->next);
        temp->next=NULL;
    }
    else{
        ptr=temp->prev;
        ptr->next=temp->next;
        ptr=temp->next;
        ptr->prev=temp->prev;
        free(temp);
    }
}
```

PILA E CODA

- Possono essere implementate con array oppure con le liste concatenate / doppiamente concatenate
- Pila e Coda funzionano in questo modo:
- **PILA: LIFO** **CODA: FIFO**



- La lista quindi permette di ottenere i valori nell'ordine inserito all'interno
- lo stack invece permette di ottenere i valori nell'ordine inverso di quando li abbiamo inseriti
- **STACK** fornisce le seguenti operazioni:
    - **POP** = rimuove l'elemento in cima allo stack
    - **PUSH** = inserisce un elemento nello stack
    - **TOP** = restituisce l'elemento in cima allo stack senza rimuoverlo
- **QUEUE** fornisce le seguenti operazioni:
    - **ENQUEUE** = Inserisce un elemento nella coda
    - **DEQUEUE** = elimina un elemento dalla coda
    - **FRONT** = restituisce il primo elemento della coda (quello che verrebbe rimosso con dequeue)
    - **REAR** = restituisce l'ultimo elemento della coda (l'ultimo elemento inserito)
- per le **queue** conviene utilizzare le **liste doppiamente concatenate**
- per lo **stack** conviene utilizzare invece le **liste concatenate singolarmente** (vanno bene anche le doubly linked list ma sarebbe uno spreco visto che sono strutture LIFO)

ALBERI

- **Albero** = collezione non vuota di nodi e archi. Ricorsivamente un albero è una foglia o una radice connessa ad un insieme di alberi.
- Terminologia:
    - **Cammino** = sequenza di nodi collegati da archi, esiste esattamente un cammino da un nodo ad un qualsiasi altro nodo
    - **Radice** = Nodo al di sopra della "gerarchia"
    - **Sottoalbero** = albero definito scegliendo un nodo interno che comprende tale nodo e tutti i suoi discendenti
    - **Alberi ordinati** = alberi in cui i figli hanno un ordine

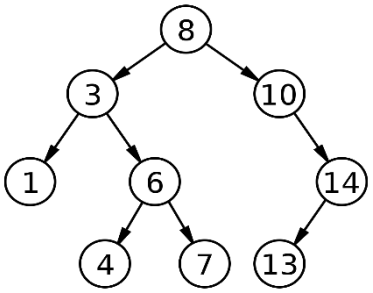
- Gli **Alberi Binari** sono importanti tipologie di **alberi ordinati.** un albero binario è una foglia oppure una radice connessa ad un albero binario destro e ad un albero binario sinistro.
    - un albero binario con **N nodi ha N-1 Archi**
    - un albero binario con **N nodi ha altezza di circa $\log_2 N$**
    - Vengono rappresentati nel seguente modo:

```
struct treenode {
    Item item;
    struct treenode *left;
    struct treenode *right;
};
typedef struct treenode *Treenode;
```

    - Si possono effettuare operazioni di attraversamento dell'albero:
        - **inorder** / **simmetric** = **SRD** = Sinistro Radice Destro
        - **Preorder** = **RSD** = Radice Sinistro Destro
        - **Postorder SDR** = Sinistro Destro Radice

- Gli **Alberi Binari di Ricerca**
    - sono strutture dati che consentono di rappresentare un insieme totalmente ordinato. Le operazioni previste sono:
        - Ricerca
        - Verifica appartenenza di un elemento
        - Inserimento
        - Cancellazione
        - Minimo e Massimo
        - Successore e Predecessore di un elemento

    - Se S è un insieme totalmente ordinato, lo rappresento come un albero binario T avente per etichette gli elementi di S e tale che per ogni nodo v di T:
        - *se u appartiene al sottoalbero di sinistra di v, allora l'etichetta di u è minore dell'etichetta di v*
        - *se u appartiene al sottoalbero di destra di v, allora l'etichetta di u è maggiore dell'etichetta di v*



CODA CIRCOLARE

- Strutture dati FIFO in cui l'ultima posizione è collegata alla prima posizione per renderla circolare.
- La struttura dati è composta da:
  - **Front** = tiene la posizione del primo elemento della coda
  - **Rear** = tiene la posizione dell'ultimo elemento della coda
  - **size** = dimensione della coda fino a quel momento
  - **max** = dimensione massima della coda
  - **array** = array che contiene i valori

```
typedef struct{
        int rear;
        int front;
        int size;
        int *arr;
}*Queue;
```

- Permette di effettuare normali operazioni di Enqueue e Dequeue
- Si implementano di seguito le principali funzioni:

1) enqueue

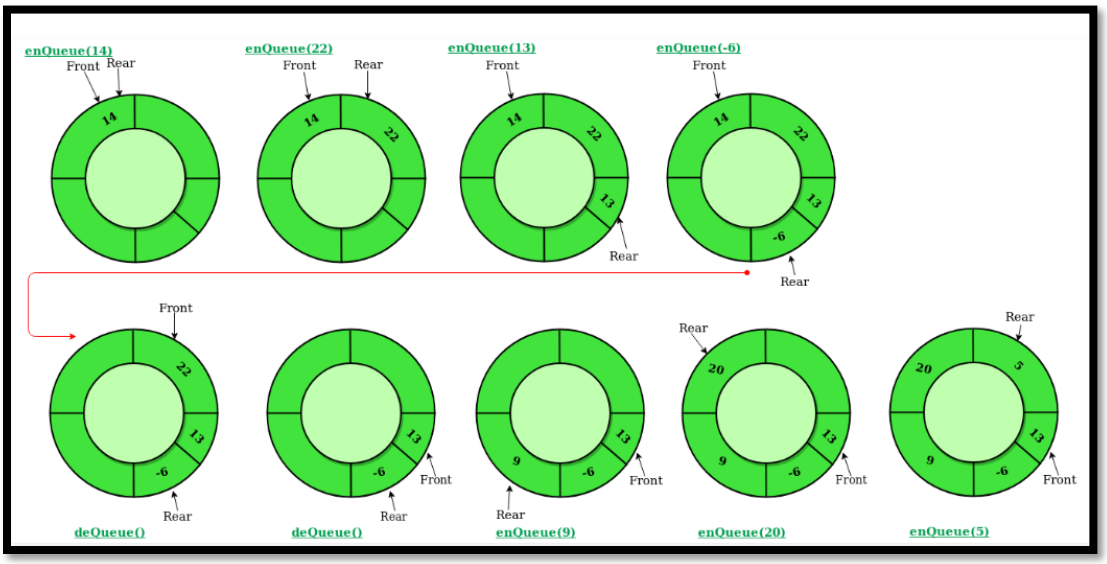
```
void enqueue(Queue queue,int value){
        if(isFull(queue)){
                printf("Queue Full!\n");
                return;
        }
        queue->arr[queue->rear]=value;
        queue->rear=(queue->rear+1)%MAX;
        queue->size++;
}
```

2) dequeue

```
int dequeue(Queue queue){
        if(isEmpty(queue)){
                exit(EXIT_FAILURE);
        }
        int n=queue->arr[queue->front];
        queue->front=(queue->front+1)%MAX;
        queue->size--;
        return n;
}
```

3) print queue

```
void printQueue(Queue queue){
        if(isEmpty(queue)){
                printf("QUEUE IS EMPTY!\n");
                return;
        }
        for(int i=queue->front;i<=queue->size;i++)
                printf("[%d]",queue->arr[i%MAX]);
        printf("\n");
}
```



- di seguito implementazione della prof. di una coda circolare:

```
typedef struct{
    Key *array;
    int lo;
    int hi;
    int len;
    int n;
}Ds;
```

```
Ds *ds_new(){
    Ds *new = malloc(sizeof(DS));
    new->len = 1;
    new->n = new->lo = new->hi = 0;
    new->array = malloc(new->len * sizeof(Key));
    return new;
}
```

```
void ds_rebuild(Ds *y){
    Key *tmp = malloc(2 * y-> len * sizeof(Key));
    for(int i=0; i< y->n; i++)
        tmp[i] = y->array[(i+y->lo)%y->len];
    free(y->array);
    y->array=tmp;
    y->lo=0;
    y->hi=y->n;
    y->len=y->len*2;
}
```

```
void ds_insert(Ds *y, Key in){
    y->array[y->hi]=in;
    y->hi+=1;
    if(y->hi == y->len)
        y->hi=0;
    y->n+=1;
    if(y->n == y->len)
        ds_rebuild(y);
}
```

```
Key ds_remove (Ds *y) {
    // assumes y is not empty
    Key out = y->array[y->lo];
    y->lo+=1;
    if(y->lo == y->len)
        y->lo=0;
    y->n-=1;
    return out;
}
```

PRIORITY QUEUE

- Considerando S un insieme dinamico di n elementi, ciascuno dei quali è dotato di una chiave, o valore di proprietà. una coda con priorità ha le seguenti caratteristiche:
    - minore è la chiave, più alto è il suo valore di priorità
    - le chiavi sono ordinate totalmente
    - bisogna effettuare in modo efficiente le seguenti operazioni:
        - inserimento
        - scegliere elemento di S con massima priorità
        - cancellare elemento di S con massima priorità
- vengono impiegate per esempio nello scheduling di processi.
- avendo a disposizione una coda di priorità è possibile effettuare questo algoritmo di ordinamento:

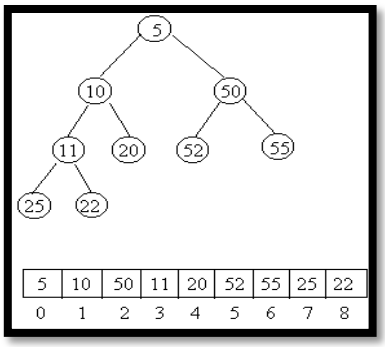
```
crea una nuova coda di priorità Q
inserisci in Q un elemento di S alla volta
finchè Q non è vuota
      estrai il minimo m da Q
      stampa m
```

- se le operazioni di inserimento e estrazione del minimo si possono fare in tempo O(logn) allora otterremmo un algoritmo di ordinamento ottimale, ovvero di costo O(nlogn) infatti:
    - per ogni elemento di S l'inserimento in coda costa O(logn) quindi l'inserimento degli n elementi costa O(nlogn)
    - l'estrazione del minimo costa O(logn) quindi il ciclo finale costa O(nlogn)

- possiamo pensare a due implementazioni abbastanza stupide ma veloci:
    1) **Usando una lista con un puntatore all'elemento minimo**
        - inserimento in testa = O(1)
        - ricerca del minimo = O(1)
        - per estrarre il minimo devo aggiornare il puntatore quindi scorrere la lista e quindi O(n)

    2) **Usando una struttura ordinata**
        - ricerca del minimo = O(1)
        - estrazione del minimo = O(1)
        - inserimento = O(n)
            - se si usa un array, con la ricerca dicotomica si trova la posizione in cui inserire con un costo O(logn) ma poi bisogna spostare tutti gli elementi più grandi e questo nel caso peggiore ha costo O(n)
            - Se si usa una lista l'inserimento ha costo O(1) ma la ricerca della posizione in cui effettuarlo ha costo O(n). Nel caso peggiore si scorre tutta la lista

HEAP



- uno **Heap** è un albero binario completo bilanciato dove le chiavi rispettano questa proprietà: → ***key(father(i)) ≤ key(i)***
- Essendo lo heap un albero binario completo è comodo rappresentarlo semplicemente come un array (In questo caso l'array parte da 0 ma per coerenza con le slide, il libro e le operazioni per la gestione dello heap nell'array bisognerebbe farlo partire da 1, quindi si consideri l'array in figura come tale.)
- Considerando la gestione dello heap tramite array, abbiamo:
  - *∀i ≥ 1* ***left(i) = 2i*** *se* $2i \leq n$
  - *∀i ≥ 1* ***right(i) = 2i + 1*** *se* $2i + 1 \leq n$
  - *∀i ≥ 2* ***father(i) = bi/2*** *(approssimato per difetto) se* $1 < i \leq n$
- Operazioni con Heap:

1) *inserimento*

- si inserisce in posizione n e poi si chiama la procedura heapify_up per correggere eventualmente la posizione del valore nello heap
- richiede **O(log n)**: al più effettuo tanti confronti/scambi quanta è l'altezza del nodo i nell'albero.

```
void heapify_up (Heap h, int i){
    if (i > 1){
        int j = father(i);
        if(key(h[i]) < key(h[j])){
            swap (h,i,j); // Swap h[i] e h[j]
            heapify_up (h, j);
        }
    }
}
```

2) *Cancellazione*

- consideriamo i la posizione dell'elemento da cancellare, allora si sposta h(n) in h(i) e si decrementa la lunghezza n.
- Potrebbe essere sbagliato lo heap quindi
  - se **key(i) < key(father(i))**
    → si esegue **heapify_up(h,j)**
  - se **key(i) > key(left(i)) or key(i) > key(right(j))**
    →si esegue **heapify_down(h,I,n)**

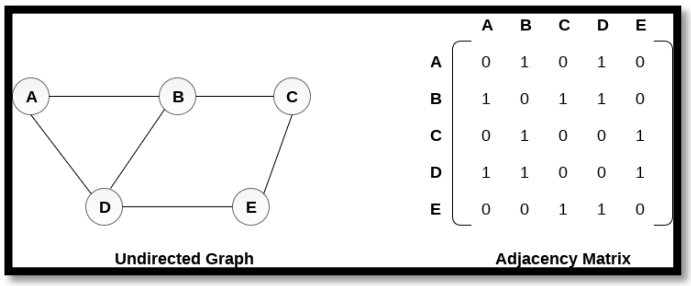
```
void heapify_down(Heap h,int i,int n){
    if (2*i<=n){     // i ha almeno un figlio
        int j;       // indice del figlio di i con chiave minore
        if(2*i==n)   // i ha solo il figlio sinistro
            j=2*i;
        else         // i ha due figli
            if(key(h[2*i])<key(h[2*i+1]))
                j=2*i;
            else
                j=2*i+1;
        if(key(h[j])<key(h[i])){
            swap (h,i,j);
            heapify_down (h,j,n );
        }
    }
}
```

- richiede O(log n): al più effettuo tanti confronti/scambi quanto è lungo il cammino dal nodo i fino ad una foglia.

GRAFI - IMPLEMENTAZIONE

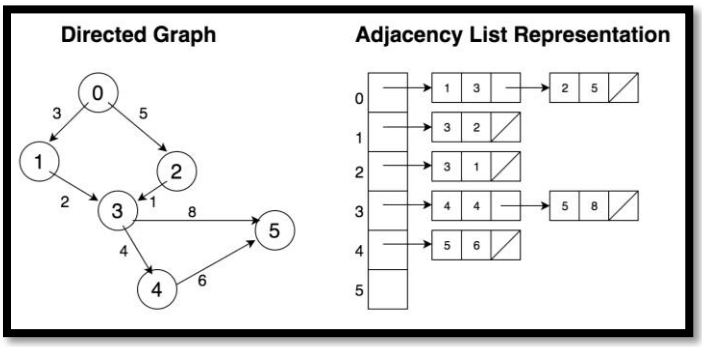
- sostanzialmente si utilizzano due rappresentazioni
    - matrice di adiacenza
    - liste di adiacenza
- I grafi possono essere
    - **sparsi** = hanno "pochi archi"
    - **densi** = hanno "molti archi"

1) Matrice di Adiacenza



- nel caso in cui il grafo fosse non orientato, la matrice avrà una simmetria.
- con un grafo non orientato si avra' uno spazio di **O(n^2)**
- con un grafo orientato si avra' uno spazio di **O(n^2)**
- utile con grafi **densi**
- nel caso in cui si avessero dei pesi tra gli archi si metterà al posto di 1 il peso.
- Costo operazioni:
    - **Verificare se esiste arco da u a v:** O(1), basta andare a prendere riga e colonna e vedere se != 0
- Implementazione: **array bidimensionale**, nulla di che quindi.

2) Lista di adiacenza



- utilizzate con i grafi **sparsi**
- con un grafo non orientato si avrà un costo di **O(n+2m)** dove n è il numero dei nodi e m è il numero degli archi
- con un grafo orientato si avrà un costo di **O(n+m)** dove n è il numero dei nodi e m è il numero degli archi
- nel caso in cui si avessero dei pesi tra gli archi si metterà (il peso) di fianco al nodo come in figura.
- Costo operazioni:
    - **Verificare se esiste arco da u a v:** O(n+m) si scorre la lista dei vertici e quando si trova si scorre la sua lista.

- Implementazione: ci sono diversi modi:
    - **I. array di liste concatenate**

```c
struct node {            /* nodo di lista di adiacenza */
    int v;
    struct node *next;
};
struct graph {           /* struttura associata a ogni grafo */
    int V;               /* numero nodi */
    int E;               /* numero archi */
    struct node **A;     /* array di liste di adiacenza */
};
```

- la prima struct definisce la lista degli archi e quindi dei nodi adiacenti a ciascun nodo in posizione i del vettore definito nella struttura graph
- la seconda struttura è la struttura del grafo che è un array di strutture node. Come detto prima ogni node è un nodo che contiene un valore e un puntatore al nodo successivo.
- struttura non adatta se si fanno molti cambiamenti nei nodi,

- Funzione per **creare** un grafo:

```
// crea un grafo con n nodi
struct graph *creategraph(int n){
    struct graph *g = malloc(sizeof(struct graph));
    if(g==NULL)
        exit(EXIT_FAILURE);
    g->n = n;
    g->A = calloc(n, sizeof(struct listnode*));
           //alloca n elementi di sizeof(...)
    if(g->A==NULL)
        exit(EXIT_FAILURE);
    return g;
}
```

- Funzione per **inserire** un nodo nella lista di adiacenza di un nodo

```
struct listnode *insertnode(struct listnode *p, int v){
    // riceve per parametro la lista di adiacenza di uno dei nodi
    struct node *newnode = malloc(sizeof(struct listnode));
    if(newnode==NULL)
        exit(EXIT_FAILURE);
    newnode->v = v;
    newnode->next = p;
    return newnode;
}
```

- Funzione per inserire un arco (v, w) in un grafo g sfruttando la funzione scritta sopra

```
void inserisci_arco(Graph g, int v, int w){
    g->A[v]=insertnode(g->A[v],w);
    g->A[v]=insertnode(g-A[w],v);
    // essendo NON orientato bisogna farlo in questo modo
    // se voglio inserire (C,F) in un grafo non orientato
    // bisognerà fare collegamento da C a F e da F a C nella lista
}
```

Domande Orale Cattivelle

DOMANDA 1:

```
int *g(void){
  int *px;
  px=(int*)malloc(sizeof(int));
  *px=10;
  return px;
}

int *i(void){
  int x= 10;
  return (&x);
}

int *h(void){
  int* px;
  *px= 10;
  return px;
}
```

Cosa fa ciascuna funzione e se ci sono problematiche.

Risposta:

- l'istruzione *px=10 nella terza funzione non funziona perchè il puntatore px non sta puntando a nulla di accessibile. Punta a qualcosa ma non a qualcosa che conosciamo, errore quindi a runtime, non a compile time. Non sappiamo dove prova a scrivere quindi potrebbe dare problemi come no.
- nella seconda funzione, c'è un errore, perche x c'è locale, e io restituisco l'indirizzo di una variabile locale che al termine viene svuotato completamente perchè nello stack.
- la prima non crea problemi perchè viene invece allocata nello heap e quindi se faccio un return risulta accessibile anche dopo che la funzione termina.
- Concetto generale: **tutto quello che viene allocato nello stack viene cancellato quando la procedura termina, se invece si fa un malloc, la variabile in questione viene allocata nello heap il quale persiste nonostante la procedura termini**.

DOMANDA 2:

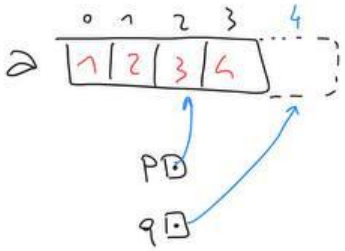
move to front dell'esame

Risposta:

```
void move_to_front(List list){
    Node tmp=list->last->prev;      //penultimo futuro last
    tmp->next=NULL;                 //penultimo next diventa NULL
    list->first->prev=list->last;   //prev del primo diventa last
    list->last->next=list->first;   //next del last diventa il primo
    list->first=list->last;         //il primo della lista diventa l'ultimo
    list->first->prev=NULL;         //il prev del first diventa NULL
    list->last=tmp;                 //l'ultimo diventa il penultimo (tmp)
}
```

ALTRE DOMANDE (a cura di uno studente a cui faccio ringraziamenti)

10/02/2021



q - p = 2

ho vettore di int
voglio ritorno segno se ∃ sottovettore
che ha somma = 17

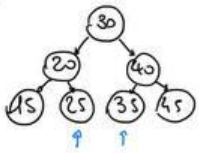
esempio:
1 4 6 2 7 19 3

trovo sottovett. la cui somma è 20

FUNZ. RICORSIVA che riceve il vettore
- indice di inizio
- target da raggiungere

10/02/2021 h 12:00

ho albero bin di ricerca
e ho una funzione che cerca un elemento.
per prob valori faccio il test per vedere se la mia funzione è corretta, dato l'albero in disegno?



ad esempio: 25, 35, 41, 40
- 25: vado a sx poi dx
- 35: vado a dx poi sx
- 41: non viene trovato
- 40: valore intermedio

---

ho un vettore di int e un valore int
scrivi funz che dato vett e int stabilisce se ∃ sottovettore la cui somma sia pari al valore indicato

esempio 5 23 7 11

Sum deve essere 11

SUGGERIM. LUNARI
Ragiono in maniera ricorsiva
per ogni vettore che ho valuto, calcolo
la funzione sul sottovettore che parte
dall'indice successivo

5 7 11 23

Scrivere ciò visto PRG.

5
7
11
23

Scrivere definizione di un tipo adatto ad
esprimere un grafo -- può essere lista di adiacenti, - - -