Strutture Dati elementari e Alberi

Le Strutture Dati sono un modo per gestire delle collezioni di oggetti.

Tipo di dato

Specifica una collezione di oggetti e delle operazioni che posso svolgere su tale collezione.

Struttura Dati

Organizzazione dei dati che permette di memorizzare la collezione e supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile.

Esempi di tipi di dato

```
tipo Dizionario

dati: Un insieme S di coppie (elem, chiave).

Operazioni:
        insert(elem e, chiave k)
                Aggiunge a S una nuova coppia(e,k)
        delete (chiave k)
                Cancella da S la coppia con chiave k.
        search(chiave k) -> elem
                Se la chiave k è presente in S restituisce l'elemento ad
essa associato      e null altrimenti.
```

```
tipo Pila

dati: Una sequenza S di n elementi

Operazioni:
        isEmpty() -> result
                Restituisce true se S è vuota, e false altrimenti.
```

```
        push(elem e)
                Aggiunge e come ultimo elemento e lo restituisce.
        pop() -> elem
                Toglie da S l'ultimo elemento e lo restituisce.
        top() -> elem
                Restituisce l'ultimo elemento di S(Senza toglierlo da S).
```

```
tipo Coda

dati: Una sequenza S di n elementi

Operazioni:
        isEmpty() -> result
                Restituisce true se S è vuota, e false altrimenti.
        enqueue(elem e)
                Aggiunge e come ultimo elemento e lo restituisce.
        dequeue() -> elem
                Toglie da S l'ultimo elemento e lo restituisce.
        first() -> elem
                Restituisce l'ultimo elemento di S(Senza toglierlo da S).
```

Possiamo rappresentare questi dati in due maniere:

- **Rappresentazioni indicizzate**: I dati sono contenuti principalmente in array.

> **Proprietà delle Rappresentazioni indicizzate**
>
> - Gli indici delle celle di un array sono numeri consecutivi (Forte);
> - **Non** è possibile aggiungere nuove celle ad un array (Debole);

- **Rappresentazioni collegate**: I dati sono contenuti in record collegati fra loro mediante puntatori.

> **Record**
>
> - Sono numerati tipicamente con il loro indirizzo di memoria.
> - Vengono creati e distrutti individualmente e dinamicamente.
> - Il collegamento tra due record avviene tramite un puntatore.

Proprietà delle rappresentazioni collegate

I costituenti base sono i record e:

- E' possibile aggiungere o togliere record a una struttura collegata (Forte);
- Gli indirizzi dei record non sono necessariamente consecutivi (Debole);

Alcuni esempi di strutture collegate sono:

- Liste semplici.
- Liste doppiamente collegate.
- Liste circolari doppiamente collegate.

Quindi ricapitolando pro e contro di questi tipi di rappresentazione abbiamo che:

- **Rappresentazioni indicizzate** --> Possiamo accedere direttamente mediante indici, ma abbiamo una dimensione fissa.
- **Rappresentazzioni collegate** --> Abbiamo una dimensione variabile, ma l'accesso ai reord è necessariamente sequenziale.

Realizzazione di un dizionario

Esistono diverse maniere per creare un dizionario, tramite:

1. **Array non ordinato**

Costi delle operazioni

Insert -> $O(1)$ - Inserisco dopo l'ultimo elemento;
Search -> $O(n)$ - Devo scorrere tutto l'array;
Delete -> $O(n)$ - Delete = Search + Cancellazione;

2. **Array ordinato**

Costi delle operazioni

Search -> $O(log(n))$ - Ricerca binaria;
Insert -> $O(n)$ - Trovo la posizione in cui inserire l'elemento $O(log(n))$ e faccio i

trasferimenti $O(n)$;
Delete -> $O(n)$ - Uguale ad Insert;

3. **Lista non Ordinata**

Costi delle operazioni

Search -> $O(n)$
Insert -> $O(1)$
Delete -> $O(n)$

4. **Lista Ordinata**

Costi delle operazioni

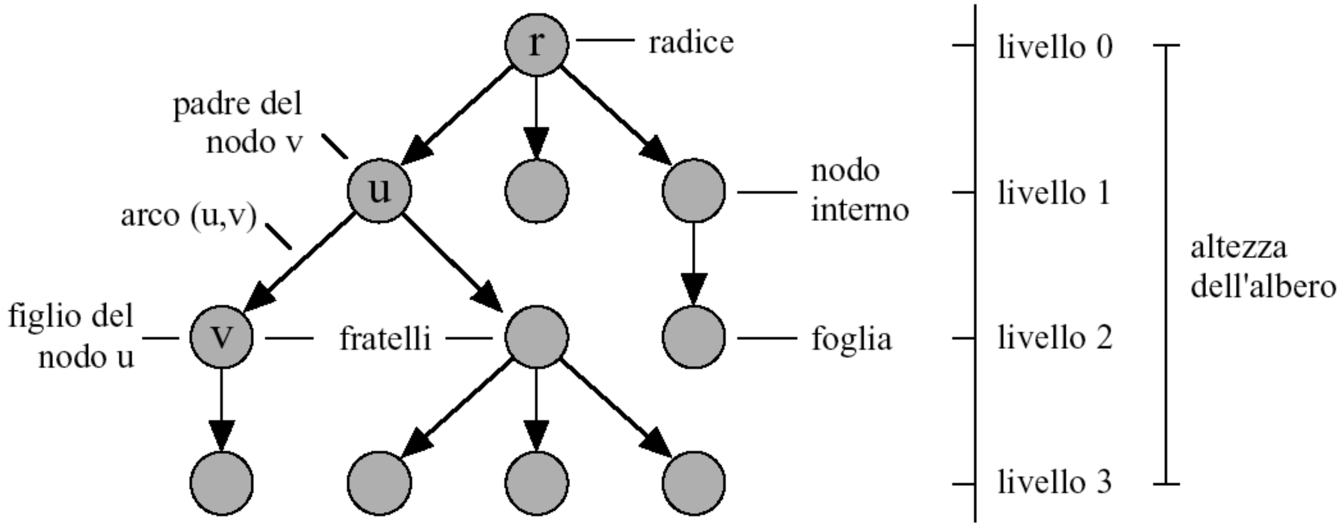
Search -> $O(n)$ - Non posso usare la ricerca binaria;
Insert -> $O(n)$ - devo mantenere ordinata la lista;
Delete -> $O(n)$

Quindi come possiamo realizzare un dizionario che ha tutte le operazioni al massimo $O(log(n))$?

E' abbastanza complicato e per farlo dobbiamo prima introdurre alcuni concetti e algoritmi sugli alberi.

Alberi

Ricordiamo inanzitutto che i dati sono contenuti all'interno dei nodi, mentre invece le relazioni gerarchiche sono definite dagli archi.



Grado di un nodo -> Numero dei suoi figli.

Inoltre un nodo u è antenato di v se u è raggiungibile da v risalendo di padre in padre, viceversa v è discendente di u se u è antenato di v.

Possiamo rappresentare gli alberi in diverse maniere:

- **Rappresentazione indicizzata:** Ogni cella dell'array contiene: le informazioni di un nodo, indici per raggiungere gli altri nodi.

Vettore dei padri

Rappresentazione di un albero tramite un array, per un albero di n nodi avremo bisogno di un array P di dimensione almeno n.
Ogni cella conterrà una coppia di informazioni (info,parent) dove:

- Info-->Contenuto informativo del nodo i
- parent --> Indice nell'array del nodo padre di i

Con questo vettore il tempo che ci mettiamo per trovare uno dei padri è costante, mentre per trovare un figlio è lineare.

Vettore posizionale per alberi d-ari quasi completi

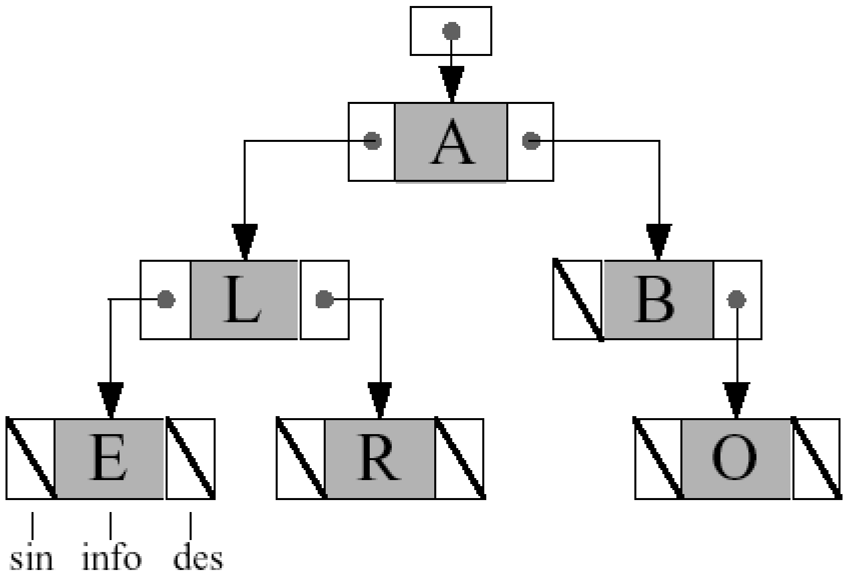
I nodi sono arrangiati per livelli, partendo dall'indice 0:

- Il $j-esimo$ figlio di i si troverà in posizione $d*i+j$
- Il padre di i si trova in posizione $\lfloor(i-2/d)\rfloor+1$

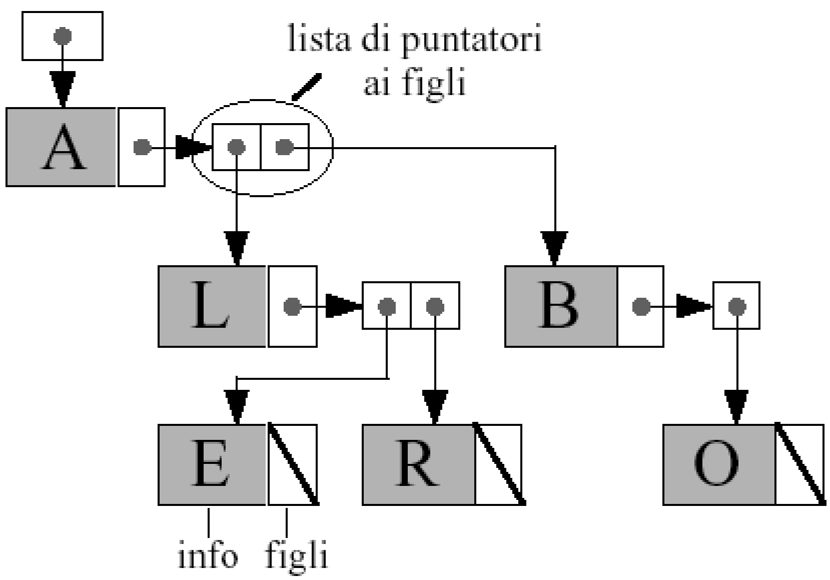
Con questo vettore possiamo individuare il padre di un nodo e il figlio di un nodo in tempo costante.

- **Rappresentazioni collegate:**

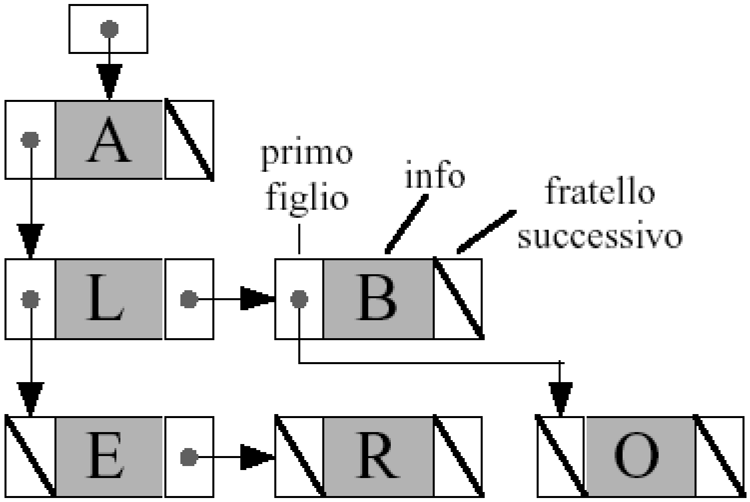
Rappresentazione con puntatori ai figli (Nodi con numero limitato di figli).



Rappresentazione con liste di puntatori ai figli (Nodi con numero arbitrario di figli)



Rappresentazione di tipo primo figlio - fratello successivo (Nodi con numero arbitrario di figli).



> **Visite di alberi**
>
> Le visite degli alberi sono algoritmi che permettono l'accesso sismtematico ai nodi e agli archi di un albero. Si distinguono in base all'ordine di accesso ai nodi.

Algoritmo di visita in profondità (DFS)

Partiamo dalla radice dell'albero e procede a visitare i nodi di figlio in figlio finché non raggiunge una fogli.
Poi torna al primo antenato che ha ancora figli da visitare e ripete il procedimento a partire da uno di quei figli.

```
algoritmo visitaDFS(nodo r)
        Pila S
        S.push(r)
        while(not S.isEmpty()) do
                u <-S.pop()
                if(u != null) then
                        visita il nodo u
                        S.push(figlio destro di u)
                        S.push(figlio sinistro di u)
```

Quindi inserisco ogni nodo all'interno della pila una sola volta, il tempo speso su ogni nodo è costante, poiché so individuare i figli in tempo costante, quindi la complessità è $O(n)$.

Algoritmo ricorsivo di visita in profondità:

```
algoritmo visitaDFSRicorsiva(nodo r)
        if(r != null) then
        visita il nodo r
        visitaDFSRicorsiva(figlio sinistro di r)
        visitaDFSRicorsiva(figlio destro di r)
```

Con questo algoritmo possiamo visitare l'albero in tre maniere diverse:

- **Visita in preordine**: Radice -> Sottoalbero Sinistro -> Sottoalbero Destro
- **Visita Simmetrica**: Sosttoalbero SInistro -> Radice -> Sottoalbero Destro
- **Visita in postordine**: Sottoalbero Sinistro -> Sottoalbero Destro -> Radice

Algoritmo di visita in ampiezza (BFS)

Partiamo dalla radice e procediamo a visitare i vari nodi per livello.
Un nodo sul livello i può essere visitato solo se abbiamo visitato tutto il livello $i-1$.

```
algoritmo visitaBFS(nodo r)
        Coda C
        C.enqueue(r)
        while(not C.isEmpty()) do
                u <-C.deque()
                if(u != null) then
                        visita il nodo u
                        S.enqueue(figlio destro di u)
                        S.enqueue(figlio sinistro di u)
```

La complessità temporale dell'algoritmo è identica a quella del DFS: $O(n)$.