Capitolo 4

Pile, code e alberi

4.1 Pile

La *pila* è una struttura dati, realizzabile sia con strutture indicizzate, sia collegate, che può essere descritta dal seguente schema generale:

Dati: una sequenza S di n elementi.

Operazioni:

`isEmpty()` $\rightarrow$ *booleano*
Restituisce *true* se S è vuota, *false* altrimenti.

`push(`*elem e*`)` $\rightarrow$ *void*
Aggiunge e come ultimo elemento di S.

`pop()` $\rightarrow$ *elem*
Toglie da S l'ultimo elemento e lo restituisce.

`top()` $\rightarrow$ *elem*
Restituisce l'ultimo elemento di S, senza rimuoverlo.

4.2 Code

La *coda*, come la pila, è una struttura dati realizzabile sia mediante strutture indicizzate, sia con strutture collegate; la realizzazione di una coda segue il seguente schema generale:

Dati: una sequenza S di n elementi.

Operazioni:

`isEmpty()` $\rightarrow$ *booleano*
Restituisce *true* se S è vuota, *false* altrimenti.

`enqueue(`*elem e*`)` $\rightarrow$ *void*
Aggiunge e come ultimo elemento di S.

`dequeue()` $\rightarrow$ *elem*
Toglie da S il primo elemento e lo restituisce.

`first()` $\rightarrow$ *elem*
Restituisce il primo elemento di S, senza rimuoverlo.

4.3 Alberi

Elenchiamo le principali caratteristiche degli alberi.

- Un *albero* è una coppia $T = (N, A)$ costituita da un insieme N di *nodi* e da un insieme $A \subseteq N \times N$ di coppie di nodi, dette *archi*.
- Definiamo cammino nell'albero una sequenza di nodi $< v_0, v_1, \ldots, v_n >$ tali che $(v_{i-1}, v_i) \in A$ per ogni $i = 1, \ldots, n$. Il valore n (ovvero i numero di archi) rappresenta anche la lunghezza del cammino.
- Un nodo, e solo uno, viene eletto *radice* dell'albero.
- Per ogni nodo v dell'albero esiste uno e un solo cammino dalla radice a v.
- In un albero, ogni nodo v (esclusa la *radice*) ha uno e un solo *padre* tale che $(u, v) \in A$.
- Ogni nodo può avere un certo numero di figli w tali che $(v, w) \in A$, ed il loro numero è detto *grado* del nodo.
- Un nodo senza figli è chiamato *foglia* e tutti i nodi che non sono foglie sono detti *nodi interni*.
- La *profondità* di un nodo è definita come segue:
  - la radice ha profondità zero;
  - se un nodo ha profondità k, i suoi figli avranno profondità $k + 1$.
- I nodi che hanno lo stesso padre sono detti *fratelli*, e dunque avranno la stessa profondità.
- L'*altezza* di un albero è definita come la massima profondità tra quelle delle varie foglie.
- Un albero *d-ario* è un albero in cui tutti i nodi tranne le foglie hanno grado d. Se inoltre tutte le foglie hanno medesima profondità, l'albero d-ario si dice che *completo*.

Lo schema generale delle operazioni eseguibili su un albero è il seguente:

Dati: un insieme di nodi e un insieme di archi

Operazioni:

`numNodi()` $\rightarrow$ *intero*
Restituisce il numero di nodi presenti nell'albero.

`grado(`*nodo v*`)` $\rightarrow$ *intero*
Restituisce il numero di figli del nodo v.

`padre(`*nodo v*`)` $\rightarrow$ *nodo*
Restituisce il padre del nodo v nell'albero, *null* se v è la radice.

`figli(`*nodo v*`)` $\rightarrow \langle nodo, nodo, ..., nodo \rangle$
Restituisce i figli del nodo v.

`aggiungiNodo(`*nodo u*`)` $\rightarrow$ *nodo*
Inserisce un nuovo nodo $v$come figlio di u e lo restituisce. Se v è il primo nodo ad essere inserito nell'albero, diventa la radice.

`aggiungiSottoalbero(`*albero a, nodo u*`)` $\rightarrow$ *albero*
Inserisce nell'albero il sottoalbero a in modo che la sua radice diventi figlia di u.

`rimuoviSottoalbero(`*nodo v*`)` $\rightarrow$ *albero*
Stacca e restituisce l'intero sottoalbero radicato in v.

4.3.1 Rappresentazioni

Le modalità di rappresentazione di un albero possono essere essenzialmente suddivise in due categorie: le *rappresentazioni indicizzate* e le *rappresentazioni collegate*. Le prime, nonostante risultino essere di facile realizzazione, rendono difficoltoso l'inserimento e la cancellazione di nodi nell'albero, mentre le seconde risultano essere decisamente più flessibili, nonostante siano leggermente più complesse da implementare.

4.3.2 Rappresentazioni indicizzate

Vettore padri

La più semplice rappresentazione possibile per un albero $T = (N, A)$ con n nodi è quella basata sul *vettore padri*: è un array di dimensione n le cui celle contengono coppie (*info*, *parent*), dove *info* è il contenuto informativo del nodo e *parent* il riferimento al padre (o *null* se si tratta della radice). Con questa implementazione, da ogni nodo è possibile risalire al padre in tempo $O(1)$, ma la ricerca dei figli richiede tempo $O(n)$.

Vettore posizionale

Consideriamo un albero d-ario completo con n nodi, dove $d \geq 2$; un *vettore posizionale* è un array P di dimensione n tale che $P[v]$ contiene l'informazione associata al nodo v e i figli sono memorizzati nelle posizioni $P[d \cdot v + i]$, con $0 \leq i \leq d - 1$. Da ciascun nodo è possibile risalire in tempo costante sia al proprio padre (indice $\lfloor v/d \rfloor$ se v non è la radice), sia a uno qualsiasi dei propri figli.

4.3.3 Rappresentazioni collegate

Puntatori ai figli

Se ogni nodo dell'albero ha al più grado d, è possibile mantenere in ogni nodo un puntatore a ciascuno dei possibili figli (se il figlio non è presente, si imposta a *null* il riferimento).

Lista figli

Se il numero massimo di figli non è noto a priori, si può mantenere per ogni nodo una lista di puntatori ai figli.

Primo figlio - fratello successivo

Variante della soluzione precedente, prevede di mantenere per ogni nodo un puntatore al primo figlio e uno al fratello successivo (*null* se non è presente, rispettivamente, il figlio o fratello); per scandire tutti i figli di un nodo, è sufficiente visitare il primo figlio e poi tutti i suoi fratelli.

4.3.4 Visite di alberi binari

Consideriamo alberi binari in cui ogni nodo può avere al massimo due figli, destro e sinistro.

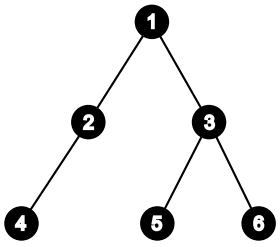

Figura 4.1: Esempio di albero binario

4.3.5 Visita in profondità

In una visita in profondità, si prosegue la visita dall'ultimo nodo lasciato in sospeso: può essere realizzata mediante l'utilizzo di una *pila* o, in maniera più semplice, usando la ricorsione. Tre varianti classiche della visita in profondità sono le seguenti:

Visita in preordine: si visita prima la radice, poi vengono eseguite le chiamate ricorsive sul figlio sinistro e destro (Applicato all'albero in Figura 4.1 produce: $1, 2, 4, 3, 5, 6$)

algoritmo `visitaPreordine` (*nodo* r) → *void*

```
1 if(r != null) then
2   visita il nodo r
3   visitaSimmetrica(figlio sinistro di r)
4   visitaSimmetrica(figlio destro di r)
```

Visita simmetrica: si effettua prima la chiamata sul figlio sinistro, poi si visita la radice e infine si esegue la chiamata ricorsiva sul figlio destro (applicata all'albero in Figura 4.1 produce $4, 2, 1, 5, 3, 6$)

algoritmo `visitaSimmetrica` (*nodo* r) → *void*

```
1 if(r != null) then
2   visitaSimmetrica(figlio sinistro di r)
3   visita il nodo r
4   visitaSimmetrica(figlio destro di r)
```

Visita in postordine: si effettuano prima le chiamate ricorsive sul figlio sinistro e destro, poi viene visitata la radice (applicata all'albero in Figura 4.1 produce $4, 2, 5, 6, 3, 1$)

algoritmo `visitaPostordine` (*nodo* r) → *void*

```
1 if(r != null) then
2   visitaSimmetrica(figlio sinistro di r)
3   visitaSimmetrica(figlio destro di r)
4   visita il nodo r
```

4.3.6 Visita in ampiezza

La visita in ampiezza è realizzata tramite l'uso di una *coda* e la sua caratteristica principale è il fatto che i nodi vengono visitati per livelli: l'ordine di visita dell'albero rappresentato in Figura 4.1 è $1, 2, 3, 4, 5, 6$.

algoritmo `visitaAmpiezza` (*nodo r*) → *void*

```
1 Coda c
2 c.enqueue(r)
3 while(!c.isEmpty()) do
4   u = c.dequeue()
5   if(u != null)
6     visita il nodo u
7     c.enqueue(figlio sinistro di u)
8     c.enqueue(figlio destro di u)
```