Giorno 4

1 Strutture dati

Una STRUTTURA DATI è un metodo per organizzare (logicamente e/o fisicamente) un insieme di dati e facilitare le operazioni su di esso.

- Una SD si dice *lineare* se i suoi elementi sono organizzati in sequenza (ogni elemento ha al più un precedente ed un successivo).
- Una SD si dice *omogenea* se i suoi elementi sono tutti dello stesso tipo.
- Una SD può essere ad accesso *diretto* o *sequenziale*; le SD ad accesso diretto permettono l'accesso a qualsiasi elemento in tempo $O(1)$, mentre quelle ad accesso sequenziale non lo permettono.

1.1 Array, Liste

1.1.1 Array

Un array è una SD lineare, omogenea, ad accesso diretto. Solitamente l'organizzazione fisica in memoria rispecchia quella logica, e gli elementi sono memorizzati in posizioni contigue.

L'accesso diretto è reso possibile proprio da questa proprietà; Per accedere all'elemento $A[i]$ dell'array, basta accedere all'elemento nell'indirizzo di memoria `indirizzo`$(A[0]) + i \cdot d$, dove d è la dimensione del tipo di dato elementare che compone l'array.

Se l'accesso diretto è un vantaggio degli array, un loro svantaggio è la rigidità della struttura; la dimensione di un array è definita alla dichiarazione e non varia mai.

1.1.2 Liste concatenate

Una lista è una SD lineare, omogenea, ad accesso sequenziale. Gli elementi non si trovano, in generale, in posizioni di memoria contigue.

Per muoversi tra un elemento e l'altro si utilizza l'attributo `x.next` di ogni elemento `x`, che punta al successivo.

La lista stessa ha un attributo `list.head` che punta al primo elemento della lista. Nelle liste *doppie* ogni elemento ha anche un attributo `x.prev` che punta al precedente. Tutti gli elementi hanno anche un attributo `key` che restituisce il loro valore.

Seguono le funzioni di ricerca, inserimento, cancellazione di un elemento nella lista.

Ricerca di elemento di chiave k

```
1  Search(L, k):
2      new nodo tmp = L.head;
3      while(tmp ≠ nil && tmp.key ≠
           k):
4          tmp = tmp.next;
5      return tmp;
```

Tempo $O(n)$

Inserimento nodo x in testa

```
1  Insert(L, x):
2      x.next = L.head;
3      L.head = x.next;
```

Tempo $O(1)$

Inserimento nodo x in testa, lista doppia

```
1  Insert(L, x):
2      if(L.head != nil): L.head.
           prev = x;
3      x.next = L.head;
4      L.head = x.next;
```

Tempo $O(1)$

Cancellazione nodo x (lista doppia)

```
1  Delete(L, x):
2      if(x.prev != nil): x.prev.
           next = x.next;
3      else: L.head = x.next;
4      if(x.next != nil): x.next.
           prev = x.prev;
```

Tempo $O(1)$, ma se si deve prima trovare il nodo x: tempo $O(n)$

Search and Delete k (lista singola)

```
1  SearchAndDelete(L, k):
2      if(L.head != nil && L.head.
           key == k): return L.head;
3      else if(L.head == nil):
           return nil;
4      else:
5          new nodo prev = L.head;
6          new nodo tmp = L.head.
               next;
7          while(tmp ≠ nil && tmp.
               key ≠ k):
8              prev = tmp;
9              tmp = tmp.next;
10         prev.next = tmp.next;
```

Tempo $O(n)$

1.2 Pile

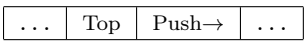
Una pila (stack) è una SD lineare che prevede due operazioni:

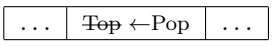
1. Push: inserisce un elemento in testa alla pila
2. Pop: rimuove un elemento dalla testa della pila e lo restituisce

Una struttura di questo tipo si dice FIFO, "First In First Out".

1.2.1 Implementazione su array

Si marca la prima posizione come `top`, e si inserisce nelle posizioni successive, aggiornando `top`. Si possono avere errori di overflow (se si eccede la lunghezza dell'array) e di underflow (se si esegue un pop quando l'array è vuoto.)





Push

```
1  Push(A, x):
2      if(A.top < A.length-1):
3          A[A.top + 1] = x;
4          A.top ++;
5      else: error <overflow>
```

Pop

```
1  Pop(A):
2      if(A.top > 0):
3          A.top --;
4          return A[A.top-1];
5      else: error <underflow>
```

1.2.2 Implementazione su Liste

Si usa head come top, il `push` è l'inserimento in testa, mentre il `pop` consiste nell'eliminazione in testa.

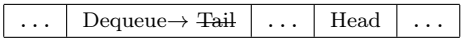
1.3 Code

Una coda è una struttura lineare che prevede due operazioni:

1. Enqueue: Inserisce un elemento dalla testa
2. Dequeue: Rimuove un elemento dalla coda

Una struttura di questo tipo si dice LIFO: "Last In First Out".

1.3.1 Implementazione su Array



1.3.2 Implementazione su Liste

$$[\text{Tail}] \longrightarrow [\ldots] \longrightarrow [\text{Head}] \longrightarrow [\text{Enqueue}\rightarrow]$$

$$[\text{Dequeue}\rightarrow \sout{\text{Tail}}] \longrightarrow [\ldots] \longrightarrow [\text{Head}]$$

Si tengono due riferimenti, uno alla testa ed uno alla coda della lista. L'enqueue corrisponde all'aggiunta di un elemento in coda alla lista, il dequeue corrisponde all'eliminazione in testa (controintuitivamente si ha **che la testa della coda è nella coda della lista, e la coda della coda è nella testa della lista**, che è uno scioglilingua eccellente che non dirò mai più).

1.4 Alberi

Un albero è una struttura non lineare. Ogni elemento `x` di un albero, detto nodo, ha un attributo che punta al nodo "padre", `x.p`, ed un certo numero di attributi che possono puntare ad un nodo "figlio", ad un array di nodi "figli" o ad una lista di nodi "figli".

- Nel caso di alberi *binari*, ossia tali che ogni nodo ha al più due figli, si hanno i due attributi `x.dx` e `x.sx`.
- Nel caso generale di alberi *cardinali*, che possono avere al più n figli, si ha un unico attributo che punta ad un array di nodi di lunghezza n.
- Nel caso di alberi *ordinali*, che non hanno un limite di figli, si ha un unico attributo che punta ad una lista concatenata di nodi.

1.5 Alberi binari

1.5.1 Visite

Se si vuole accedere ad uno ad uno a tutti i nodi di un albero binario, ad esempio se li si vuole stampare, si deve usare un algoritmo di *visita*.
I più usati sono: visita **anticipata**, visita **posticipata**, visita **simmetrica**.

Visita anticipata

```
1  VA(r):
2      if(r == nil) return;
3      print(r.key);
4      if(r.sx != nil) VP(r.sx);
5      if(r.dx != nil) VP(r.dx);
```

Visita posticipata

```
1  VP(r):
2      if(r == nil) return;
3      if(r.sx != nil) VP(r.sx);
4      if(r.dx != nil) VP(r.dx);
5      print(r.key);
```

Visita simmetrica

```
1  VS(r):
2      if(r == nil) return;
3      if(r.sx != nil) VP(r.sx);
4      print(r.key);
5      if(r.dx != nil) VP(r.dx);
```

Un'altra visita è la **visita per livelli**, che viene implementata con una coda d'appoggio:

```
1  VL(d):
2      coda = new coda;
3      enqueue(coda, d);
4      while(!isEmpty(coda)):
5          let nodo = first(coda);
6          if(nodo.sx ≠ nil) enqueue(coda, nodo.sx);
7          if(nodo.dx ≠ nil) enqueue(coda, nodo.dx);
8          print(dequeue(coda));
```

[notazione binarizzata]