Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

Liste bidirezionali, struttura dati coda*

1 Liste, liste con puntatore a *last*, liste bidirezionali

Per i primi due esercizi considerate la seguente porzione di codice. Il tipo `List_with_tail` indica una lista concatenata di interi, dotata di due riferimenti al primo e all'ultimo elemento della lista. Quando la lista è vuota, sia `head` che `tail` sono `NULL`. La funzione `new_node` alloca lo spazio per un nuovo nodo e ne inizializza il valore con l'argomento passato.

```
1  struct node {
2    int info;
3    struct node *next;
4  };
5  typedef struct node *Node;
6
7  typedef struct {
8    Node head;
9    Node tail;
10 } *List_with_tail;
11
12 Node new_node ( int e ) {
13   Node new = malloc( sizeof (struct node) );
14   new -> info = e;
15   new -> next = NULL;
16   return new;
17 }
```

1.1 Inserimento alla fine

Considerate la seguente funzione (incompleta) che aggiunge un elemento `e` in fondo alla lista:

```
1  void addAtEnd( List_with_tail l, int e ){
2    if ( l -> tail == NULL ) {
3      l -> tail = new_node(e);
4      l -> head = l -> tail;
5    }
6    else {
7      // MISSING CODE
8    }
9  }
```

Quale dei seguenti frammenti di codice completa la funzione `addAtEnd`? Spiegate quali problemi si riscontrano scegliendo ciascuna delle altre opzioni.

*Ultima modifica 2 dicembre 2020

A)
```
l -> tail -> next = e;
l -> tail = e;
```

B)
```
Node temp = new_node(e);
l -> tail -> next = temp;
```

C)
```
Node temp = new_node(e);
l -> tail = temp;
```

D)
```
l -> tail -> next = new_node(e);
l -> tail = l -> tail -> next;
```

E)
```
Node temp = l -> head;
while ( temp -> next != NULL ) {
  temp = temp -> next;
}
temp -> next = new_node(e);
```

1.2 Lista semplice e lista con `tail`

Nella lista concatenata `List_with_tail` abbiamo i riferimenti all'inizio e alla fine della lista. Confrontate questa implementazione con quella di una lista semplice che non ha un riferimento alla fine della lista, cioè definita come segue:

```
typedef struct {
  Node head;
} *List;
```

Per quali delle seguenti operazioni si ha un tempo migliore con `List_with_tail` piuttosto che con `List`?

Scegliete tutte le opzioni corrette e giustificate la risposta.

A) restituisci 1 se la lista contiene un dato elemento
B) cancella l'ultimo elemento della lista
C) aggiungi un elemento all'inizio della lista
D) aggiungi un elemento alla fine della lista

1.3 Lista semplice o bidirezionale?

Avete l'implementazione di una lista, ma non sapete se si tratta di una lista semplice (con solo il riferimento all'inizio della lista) oppure una lista bidirezionale (con riferimenti all'inizio e alla fine della lista, e in cui ogni nodo ha riferimenti al nodo successivo e al precedente).

Avete accesso alla lista solo tramite questa interfaccia:

- `size()` restituisce il numero di elementi nella lista
- `contains(e)` restituisce 1 se `e` è nella lista
- `removeAtStart()` cancella l'elemento all'inizio della lista
- `removeAtEnd()` cancella l'elemento alla fine della lista
- `addAtStart(e)` inserisce l'elemento `e` all'inizio della lista
- `addAtEnd(e)` inserisce l'elemento `e` alla fine della lista

Indicate quali dei seguenti esperimenti consente di stabilire quale sia l'implementazione sconosciuta. Se ci sono più opzioni valide, scegliete la migliore. Giustificate la risposta.

A) Realizzo una mia implementazione della lista semplice e confronto i tempi di esecuzione su tutte le funzioni. Se i tempi della mia implementazione coincidono con quelli della implementazione data per tutte le funzioni, allora si tratta di una lista semplice, altrimenti di una lista bidirezionale.

B) Eseguo *N* operazioni `addAtEnd` seguite da *N* operazioni `addAtStart`. Se le operazioni `addAtEnd` ci impiegano molto di più di quelle `addAtStart`, allora si tratta di una lista semplice, altrimenti una lista bidirezionale.

C) Eseguo *N* operazioni `addAtEnd` seguite da *N* operazioni `removeAtEnd`. Se le operazioni `removeAtEnd` ci impiegano molto di più di quelle `addAtEnd`, allora si tratta di una lista semplice, altrimenti di una lista bidirezionale.

D) Creo due istanze della lista e confronto il tempo di esecuzione della prima rispetto alla seconda, per tutte le operazioni. Se i tempi sono simili per tutte le operazioni, allora si tratta di una lista semplice, altrimenti di una lista bidirezionale.

E) Nessuno degli esperimenti precedenti mi consente di rispondere alla domanda. Bisogna esaminare il codice per vedere se l'implementazione usa un riferimento `prev` al nodo precedente.

1.4 Funzione misteriosa

Considerate questa funzione per una lista concatenata semplice. Assumete che `head` sia una variabile che si riferisce al primo nodo della lista.

```
1 struct node *mystery( List l, int value ) {
2   struct node *current = l -> head;
3   struct node *temp = NULL;
4   while ( current != NULL && current -> item != value ) {
5     temp = current;
6     current = current -> next;
7   }
8   return temp;
9 }
```

Analizzate la funzione e descrive quale è lo "scopo" della funzione `mystery`, ovvero: quale è il senso/l'obiettivo complessivo della funzione? Prendete in considerazione tutti i casi possibili.

1.5 Annulla l'ultima operazione

Immaginate di dover implementare una funzionalità "annulla" (*undo*), che annulla l'ultima operazione svolta da un applicativo: volete salvare le azioni dell'utente in modo da poterle annullare nell'ordine inverso. Per esempio, se l'utente esegue le azioni a,b, e c, la funzionalità "annulla" andrebbe ad annullare prima c, poi b e infine a.

Avete una lista concatenata semplice (con solo il riferimento al primo nodo). Con l'obiettivo di avere la migliore prestazione per la funzionalità "annulla", come usereste la lista per memorizzare le azioni dell'utente?

Selezionate tutte le opzioni corrette e giustificate la risposta:

A) Aggiungo alla fine e tolgo dall'inizio
B) Aggiungo all'inizio e tolgo dalla fine
C) Aggiungo e tolgo dalla fine
D) Aggiungo e tolgo dall'inizio.

2 Implementazione di code con array circolari

Implementate la struttura dati *Queue* secondo il paradigma FIFO (*First In First Out*): il prossimo elemento da inserire viene messo alla fine della coda (*rear*) e il prossimo elemento da togliere viene preso dalla testa (*front*).

Rappresentate la coda tramite un array circolare, come spiegato a lezione. Usate l'operatore di modulo per ripartire da 0 quando si raggiunge l'ultima posizione dell'array. Usate come riferimento alla spiegazione e alle animazioni proposte da OpenDSA (`https://opendsa-server.cs.vt.edu/OpenDSA/Books/CS2/html/Queue.html#array-based-queues`).

Oltre all'array circolare avrete bisogno di due indici `front` e `rear` e di una variabile `n` che indica quanti elementi sono nella coda. È opportuno raggruppare queste variabili (incluso l'array) in una struttura, in modo da poter passare un solo argomento alle funzioni `enqueue`, `dequeue`, ecc.

Organizzate il vostro programma separando l'interfaccia `queue.h` dal file di implementazione `queue-circular.c`. Scrivete inoltre un file *client* (contenente una funzione `main`) per testare la vostra implementazione.

L'interfaccia conterrà ad esempio questi prototipi:

```
1  // Put element on rear
2  void enqueue( Queue, Item );
3
4  // Remove and return element from front
5  Item dequeue( Queue );
6
7  // Return front element
8  Item frontValue( Queue );
9
10 // Check if queue is empty
11 int is_empty_queue( Queue );
12
13 // Print queue content
14 void print_queue( Queue );
```

A seconda di come decidete di realizzare il vostro programma, potrebbe risultare necessario/comodo definire l'interfaccia diversamente. Ad esempio, potreste voler aggiungere all'interfaccia una "definizione incompleta" per il tipo `Queue` (vedi scheda su paradigma client-interfaccia-implementazione); in questo caso sarà opportuno aggiungere all'interfaccia anche due funzioni per la creazione e per la distruzione della coda.

3 Implementazione di code con liste concatenate

Realizzate un'implementazione alternativa per *queue* usando le liste concatenate. Potete usare una lista bidirezionale oppure una lista semplice con in aggiunta un puntatore all'ultimo elemento della lista. In questo secondo caso, valutate cosa tra `front` e `rear` è opportuno collocare all'inizio della lista, in modo da garantire che il costo in tempo di *enqueue* e *dequeue* sia costante.