Estruturas de Dados / Programação 2
Listas Duplamente Encadeadas

Márcio Ribeiro
marcio@ic.ufal.br
twitter.com/marciomribeiro

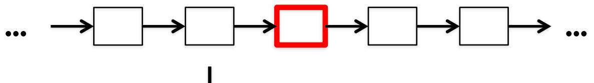
Linked Lists

- Node
  - Next node
  - Item (data)

- Can you see problems?

- Take 3 minutes to discuss with your friends about them

Keeping the previous node when removing

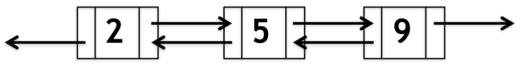


Previous

Doubly Linked List

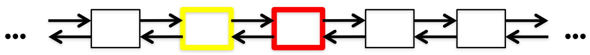
Doubly Linked Lists

```
struct node {
  int item;
  node *next;
  node *previous;
};
```



Now…

✓ No need to traverse the list to discover the previous node



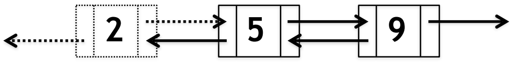
Abstract Data Type: Doubly Linked List

Doubly Linked List ADT

```
node* create_doubly_linked_list();

node* add(node *head, int item);

node* search(node *head, int item);

node* remove(node *head, int item);

int is_empty(node *head);

void print_doubly_linked_list_forward(node *head);

void print_doubly_linked_list_backward(node *tail);
```

Adding elements… (at the beginning)

```
node* add(node *head, int item)
{
  node *new_node = (node*) malloc(sizeof(node));
  new_node->item = item;
  new_node->next = head;
  new_node->previous = NULL;
  if (head != NULL)
    head->previous = new_node;
  return new_node;
}
```



Exercise 1: write the *print_doubly_linked_list_backward*

```
void print_doubly_linked_list_backward(node *tail)
{
  if (tail != NULL) {
    printf("%d\n", tail->item);
    print_doubly_linked_list_backward(tail->previous);
  }
}
```

Exercise 2: write the *remove* function

```
node* remove(node *head, int item)
{
  node *current = head;
  while (current != NULL && current->item != item)
    current = current->next;

  if (current == NULL) return head;

  if (head == current) {
    head = current->next;
  } else {
    current->previous->next = current->next;
  }

  if (current->next != NULL) {
    current->next->previous = current->previous;
  }
  free(current);
  return head;
}
```

Application

Least Recently Used - LRU

Least Recently Used (LRU) Cache

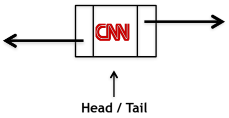
- Cache algorithm
  - Operating systems
  - Web browsers
    - Images
    - Pages

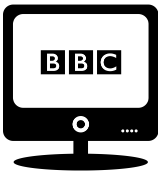
Least Recently Used (LRU) Cache

- After a request…
  - If it is present in cache, it is moved to front of the list
  - If it is not present , a new mapping is done. If cache is not full, a new entry is added to front. Otherwise, the least recently used entry is removed and then a new entry to front is added.

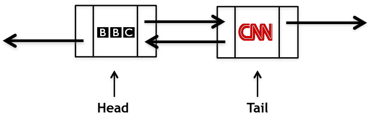
Cache size = 3

```
if (contains(key)) {

  node = map.get(key);
  bring_to_head(node);

} else {

  node* new_node = create_node();

  if (current_size < capacity) {
    add_to_head(new_node);
    current_size++;
  } else {
    remove_tail_node();
    add_to_head(new_node);
  }
}
```

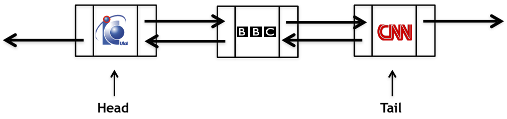




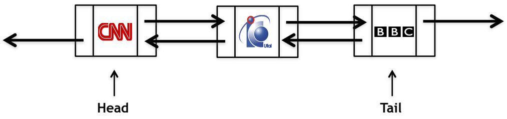
```
if (contains(key)) {

  node = map.get(key);
  bring_to_head(node);

} else {

  node* new_node = create_node();

  if (current_size < capacity) {
    add_to_head(new_node);
    current_size++;
  } else {
    remove_tail_node();
    add_to_head(new_node);
  }
}
```



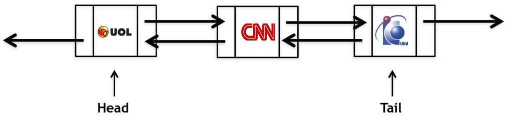
```
if (contains(key)) {

  node = map.get(key);
  bring_to_head(node);

} else {

  node* new_node = create_node();

  if (current_size < capacity) {
    add_to_head(new_node);
    current_size++;
  } else {
    remove_tail_node();
    add_to_head(new_node);
  }
}
```



```
if (contains(key)) {

  node = map.get(key);
  bring_to_head(node);

} else {

  node* new_node = create_node();

  if (current_size < capacity) {
    add_to_head(new_node);
    current_size++;
  } else {
    remove_tail_node();
    add_to_head(new_node);
  }
}
```



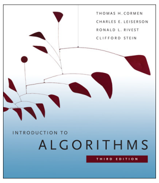
```
if (contains(key)) {

  node = map.get(key);
  bring_to_head(node);

} else {

  node* new_node = create_node();

  if (current_size < capacity) {
    add_to_head(new_node);
    current_size++;
  } else {
    remove_tail_node();
    add_to_head(new_node);
  }
}
```



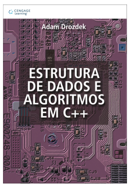
```
void add_to_head(node* new_node)
{
  new_node->next = head;
  new_node->previous = null;
  if (head != null)
    head->previous = new_node;
  head = new_node;
  if (tail == null)
    tail = new_node;
  cache.put(new_node->item, new_node);
}

void bring_to_head(node* node)
{
  node* previous = node->previous;
  node* next = node->next;

  if (previous != null)
    previous->next = next;
  else
    head = next;

  if (next != null)
    next->previous = previous;
  else
    tail = previous;

  add_to_head(node);
}
```

put / get: Hash Tables

References



Chapter 10



Chapter 3