Data structure [A04]

김종규, PhD

2017-03-27

Reviews

- Quiz exercise

Remarks

- Index begins 0 or 1?
  - depends on the description
- 다음은 stack 의 push operation 이다. array index 가 0 에서 시작할 때 pop() operation 을 정의하시오.

Outline

- Queue in more depth
- Linked list
- Recursion
- Searching

Data structure: Queue

- Circular queue

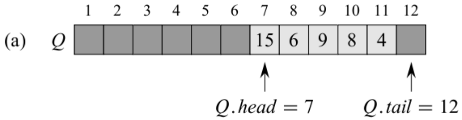

그림: Circular queue

Data structure: Queue

- Circular queue

ENQUEUE(Q, x)

```
Q[Q.tail] = x
if Q.tail == Q.length
      Q.tail = 1
else Q.tail = Q.tail + 1
```

DEQUEUE(Q)

```
x = Q[Q.head]
if Q.head == Q.length
      Q.head = 1
else Q.head = Q.head + 1
return x
```

Data structure: Empty

- What is empty?

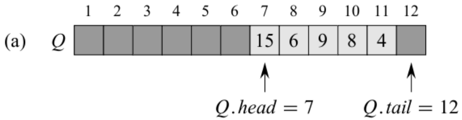

그림: Circular queue

$\longrightarrow$ `Q.head == Q.tail`

Data structure: Empty

- What is full?

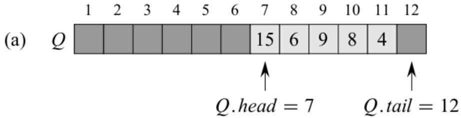

그림: Circular queue

$\longrightarrow$ `Q.head == Q.tail`?

$\longrightarrow$ Must be smaller than this

- The maximum number of elements is $n - 1$ where n is the size of the given array

Clock in

그림: Clock in

Time card


K. FRENCH
6/17
L. DIABLO
6/17
456
J. BURR
6/17
S. JORDAN
6/17
A. HICKS
S. MOORE
1st DAY
2nd DAY
3rd DAY
4th DAY
5th DAY
6th DAY
7th DAY
CAUTION

Clock-in data

```
eid,name,time,status
...
014,"Abraham Lincoln",08:59:51,on-time
023,"George Washington",09:00:01,late
...
```

$\longrightarrow$ No arbitrary limit on the size of entry

Clock-in data

- Common questions
  - List all employes late at work
- Common Operations
  - Delete the clock-in data for “George Washington”, who cheated the system

$\longrightarrow$ Abstract data type: list (insert, search, delete)

  - No arbitrary limit on the size of entry (i.e., not an array)
  - insert is the most common operation
  - delete is a quite common
  - search is a rare operation (once a day)

  $\longrightarrow$ fast insert, delete is required

Concept: Doubly Linked list

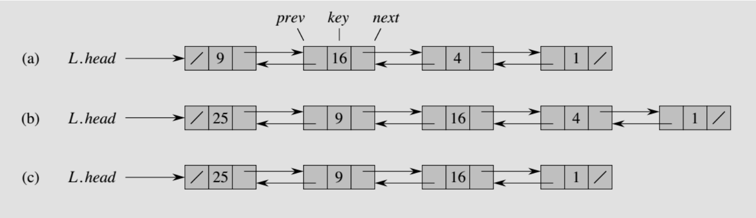

그림: Concept of doubly linked list

Search

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

그림: Searching a value

- $O(n)$

Insert

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

그림: Inserting a node

- $O(1)$

Delete

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

그림: Deleting a node

- $O(1)$

Concept: sentinel

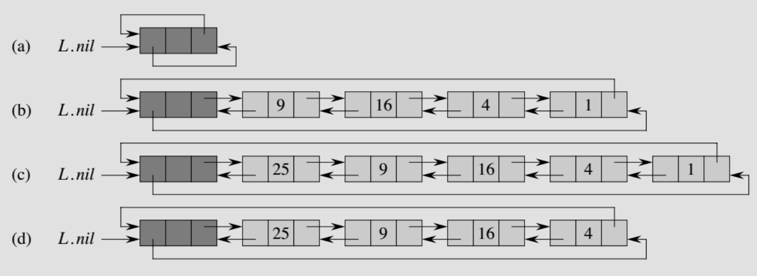

그림: Linked list with a sentinel

Search with sentinel

$\text{LIST-SEARCH}'(L, k)$

```
x = L.nil.next
while x ≠ L.nil and x.key ≠ k
    x = x.next
return x
```

그림: Searching with a sentinel

- $O(n)$

Insert with sentinel

LIST-INSERT$'(L, x)$

1. $x.next = L.nil.next$
2. $L.nil.next.prev = x$
3. $L.nil.next = x$
4. $x.prev = L.nil$

그림: Inserting with a sentinel

- $O(1)$

Comparison: search

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

그림: Searching a value

```
LIST-SEARCH′(L, k)
x = L.nil.next
while x ≠ L.nil and x.key ≠ k
    x = x.next
return x
```

그림: Searching with a sentinel

Comparison: insert

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

그림: Inserting a node

```
LIST-INSERT′(L, x)
x.next = L.nil.next
L.nil.next.prev = x
L.nil.next = x
x.prev = L.nil
```

그림: Inserting with a sentinel

Linked list

- Linked list is an abstract data type by which we can handle collection of objects in linear order
  - Retrieval: first, next, last
  - Modify: addFirst, deleteFirst, $\cdots$
- A singly linked list uses a single pointer to manage its data structure
  - Deleting an element in the middle $\longrightarrow O(n)$

    $\longrightarrow$ Doubly linked list
  - Adding behind the last element $\longrightarrow O(n)$

    $\longrightarrow$ Circular list

Concept: Doubly Linked list

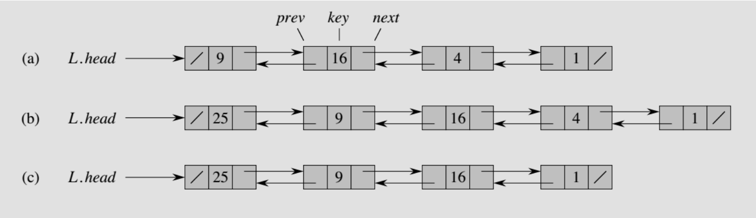

그림: Concept of doubly linked list

C language

```
struct Node {
  int key;
  struct Node* prev;
  struct Node* next;
};
struct List {
  struct Node* head;
};
```

Java

```
class Node {
  public int key;
  public Node prev;
  public Node next;
  public Node() {
    prev = next = null;
  }
}
class List {
  public Node head;
  public List() {
    head = null;
  }
}
```

Python

```
class Node:
  def __init__(self, key):
    self.key = key
    self.prev = None
    self.next = None
class List:
  def __init__(self):
    self.head = None
```

Concept: sentinel

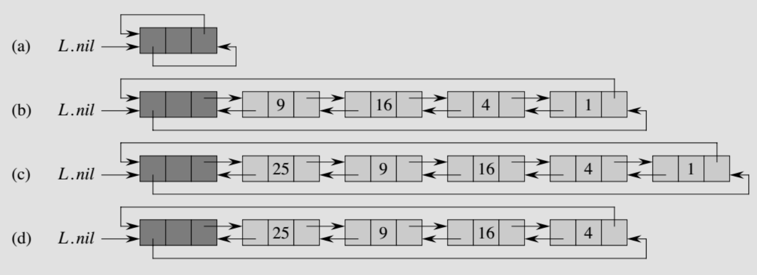

그림: Linked list with a sentinel

C

```c
struct List {
  struct Node* head;
  struct Node nil;
};
void list_init(struct List* L) {
  L->head = &L.nil;
}
```

Java

```
class List {
  public Node head;
  public Node nil;
  public List () {
    nil = new Node();
    head = nil;
  }
}
```

Python

```
class List:
  def __init__(self):
    nil = Node()
    self.head = nil
```

Singly Linked List

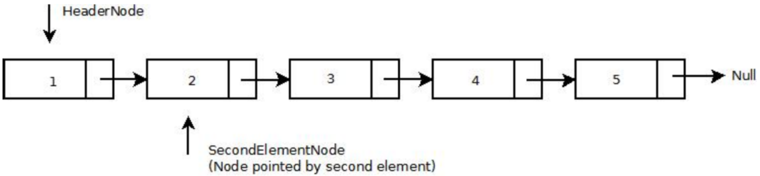

그림: Linked list

Implementation using array

```
struct Node {
  int key;
  int next;
};
#define NODE_BUFFER_SIZE 10
struct Node node_buffer[NODE_BUFFER_SIZE];
int free_ptr;
```

Implementation using array

```
void init_node_buffer() {
  for (int i = 0; i < NODE_BUFFER_SIZE; i++) {
    node_buffer[i].next = i + 1;
  }
  node_buffer[NODE_BUFFER_SIZE - 1].next = -1;
  free_ptr = 0;
}
```

Implementation using array

```
int count_free() {
  int cnt = 0;
  for(int p = free_ptr; p != -1; p = node_buffer[p].next)
    cnt++;
  return cnt;
}
```

Implementation using array

```
int alloc_node() {
  assert(free_ptr != -1);
  int node = free_ptr;
  free_ptr = node_buffer[node].next;
  node_buffer[node].next = -1;
  return node;
}
```

Implementation using array

```
int free_node(int node)
{
  node_buffer[node].next = free_ptr;
  free_ptr = node;
}
```

Implementation using pointer

```
struct Node {
  int val;
  int next;
};
struct Node {
  int val;
  struct Node* next;
};
```

Implementation using pointer

```
int alloc_node() {
  assert(free_ptr != -1);
  int node = free_ptr;
  free_ptr = node_buffer[node].next;
  node_buffer[node].next = -1;
  return node;
}
struct Node* alloc_node() {
  struct Node* node
    = (struct Node}malloc(sizeof(struct Node));
  node->next = NULL;
  return node;
}
```

Implementation using pointer


```
int free_node(int node) {
  node_buffer[node].next = free_ptr;
  free_ptr = node;
}
void free_node(struct Node* node) {
  free(node);
}
```

Python implementation

```
class Node:
    def __init__(self):
        self.val = 0;
        self.next = None
    def insert_first(self,val):
        n = Node()
        n.val = val
        n.next = self
        return n
    def delete_first(self):
        res = self.next
        return res
```

- No `free()`? $\longrightarrow$ garbage collection

Administriva

- Quiz: 03-29 (Wed) 30 min (max 40 min)

Remember the first question

- Reverse printing: "abcdefz" $\longrightarrow$ "zfedcba"

```
main()
{
  int ch;
  for (ch = getchar(); ch != EOF;ch = getchar()) {
    push(ch);
  }
  while(!empty()) {
    putchar(pop());
  }
}
```

$\longrightarrow$ With array implemented stack, we must know the size of inputs in advance

Linked list as a stack

```
int main() {
  struct Node* list = NULL;
  list = insert_node(list, 1);
  list = insert_node(list, 2);
  list = insert_node(list, 3);
  list = insert_node(list, 4);
  print_list(list);
}
```

Linked list as a stack

```
struct Node* stk = NULL;
void push(int val) {
  stk = insert_node(stk, val);
}
int pop() {
  int val = stk->val;
  stk = delete_first(stk);
  return val;
}
```

Linked list as a stack

```
push(1);
push(2);
push(3);
printf("%d\n", pop());
printf("%d\n", pop());
printf("%d\n", pop());
```

Linked list and queue

- Can we implement queue using linked list?

Further reading

- This week
  - Chapter 10: Stack, queue, linked list
- Next week
  - Chapter 6, 7

Wrap-up

- There are several important classes of complexities, i.e., $O(g(n))$
- Linked lists can manage an unlimited number of objects
  - Accessing n-th elements takes $O(n)$, which is much slower than an array
  - Searching a value takes $O(n)$, which is much slower than binary search of a sorted array
- By using linked list data structure, we can implement a stack by which we can push elements as far as memory allows