Chapter 12. Fundamental Data Structures

Byoung-Tak Zhang
TA: Hanock Kwak
Biointelligence Laboratory
School of Computer Science and Engineering
Seoul National University

http://bi.snu.ac.kr
Stacks

- Collection of elements, with two principal operations
  - push
    - adds an element to the collection
  - pop
    - removes the last element that was added.
- First-In-Last-Out (FILO)
Stacks

- Stack with integer elements
  - int stack[MAX];
  - int top = -1; // -1 means that the stack is empty.

<table>
<thead>
<tr>
<th>i</th>
<th>stack[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>don’t care</td>
</tr>
<tr>
<td>5</td>
<td>don’t care</td>
</tr>
<tr>
<td>4</td>
<td>don’t care</td>
</tr>
<tr>
<td>3</td>
<td>don’t care</td>
</tr>
<tr>
<td>2</td>
<td>don’t care</td>
</tr>
<tr>
<td>1</td>
<td>don’t care</td>
</tr>
<tr>
<td>0</td>
<td>don’t care</td>
</tr>
</tbody>
</table>

top = -1
Stacks

- push(5):

<table>
<thead>
<tr>
<th>i</th>
<th>stack[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>don’t care</td>
</tr>
<tr>
<td>5</td>
<td>don’t care</td>
</tr>
<tr>
<td>4</td>
<td>don’t care</td>
</tr>
<tr>
<td>3</td>
<td>don’t care</td>
</tr>
<tr>
<td>2</td>
<td>don’t care</td>
</tr>
<tr>
<td>1</td>
<td>don’t care</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

\[\text{top} = 0\]
Stacks

- `push(7);`

```
<table>
<thead>
<tr>
<th>i</th>
<th>stack[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>don't care</td>
</tr>
<tr>
<td>5</td>
<td>don't care</td>
</tr>
<tr>
<td>4</td>
<td>don't care</td>
</tr>
<tr>
<td>3</td>
<td>don't care</td>
</tr>
<tr>
<td>2</td>
<td>don't care</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
```

top = 1
Stacks

- push(1);
- push(3);

\[
\begin{array}{c|c}
\text{i} & \text{stack[i]} \\
\hline
6 & \text{don’t care} \\
5 & \text{don’t care} \\
4 & \text{don’t care} \\
3 & 3 \\
2 & 1 \\
1 & 7 \\
0 & 5 \\
\end{array}
\]

top = 3
Stacks

- `pop();`

<table>
<thead>
<tr>
<th>i</th>
<th>stack[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>don’t care</td>
</tr>
<tr>
<td>5</td>
<td>don’t care</td>
</tr>
<tr>
<td>4</td>
<td>don’t care</td>
</tr>
<tr>
<td>3</td>
<td>don’t care</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

`top = 2`
The linked list is a data structure consisting of a group of nodes which together represent a sequence.

Each node is composed of data and a reference (pointer).
typedef char DATA;
struct linked_list {
    DATA d;
    struct linked_list *next;
};
typedef struct linked_list ELEMENT;
typedef ELEMENT *LINK;
Storing three characters n, e, w

```c
LINK head; /* ELEMENT *head */
head = malloc(sizeof(ELEMENT));
head->d = 'n';
head->next = NULL;
```
Storing three characters n, e, w

head->next = malloc(sizeof(ELEMENT));
head->next->d = 'e';
head->next->next = NULL;
Storing three characters n, e, w

```c
head->next->next = malloc(sizeof(ELEMENT));
head->next->next->d = 'w';
head->next->next->next = NULL;
```
List Operations

- Inserting an element
- Deleting an element
- Looking up an element
- Counting the elements
Inserting an element
Inserting an element

```c
void insert(LINK p1, LINK p2, LINK q)
{
    assert(p1 -> next == p2);
    p1 -> next = q; /* insert */
    q -> next = p2;
}
```
Deleting an element
Deleting an element

```c
void delete(LINK p)
{
    p->next=p->next->next;
    /*element containing B becomes a garbage*/
}
```
Looking up an element

```c
/* Looking up an element recursively. */
LINK find(LINK head, DATA d)
{
    if (head == NULL) {
        return 0; // not found
    }
    else if (head->d == d) {
        return head; // found!
    }
    else {
        return find(head->next); // find recursively
    }
}
```
Counting the elements

/* Count a list recursively. */
int count(LINK head)
{
    if (head == NULL) {
        return 0;
    }
    else {
        return (1 + count(head -> next));
    }
}
Queues

- Collection of elements, with two principal operations
  - enqueue
    - adds an element to the rear-end of the collection
  - dequeue
    - removes the front-end element in the collection.
- First-In-First-Out (FIFO)
Implementing a Queue with a Linked List

```c
LINK head;  // pt. to the front-end element
LINK rear;  // pt. to the rear-end element
```

```plaintext

<table>
<thead>
<tr>
<th>Linked List Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A --&gt; B --&gt; C</td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>head</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>rear</td>
</tr>
</tbody>
</table>
```

enqueue

head
D
A
B
C
NULL
rear

head
A
B
C
NULL
rear
D

A
B
C
D
NULL
A dequeue

head

A → B → C → D

rear

null

head

A

B → C → D

rear

null
Implementing a Stack with a Linked List

\[\text{LINK} \top; \quad \text{// pt. to the top element}\]
push('D')
pop

D -> C -> B -> A

D

C -> B -> A

NULL