Chapter 10 Elementary Data Structures

By Y. Meng   gmail: y(dot)meng201011

Dynamic sets

△ Sets manipulated by algorithms can grow, shrink, or otherwise **change over time** are **dynamic**.
△ Operations on dynamic sets:
→ SEARCH(S, k)
returns i such that $S[i].key = k$, or NIL if no such element was found in S.
→ INSERT(S, x)
insert x into S (say, at i) such that $S[i-1].key \le x < S[i+1].key$.
→ DELETE(S, x)
remove x from S if found, otherwise do nothing.
→ MINIMUM(S)
returns i such that for all $j = 0, 1, \cdots, S.length$; $S[i].key \le S[j].key$.
→ MAXIMUM(S)
returns i such that for all $j = 0, 1, \cdots, S.length$; $S[i].key \ge S[j].key$.
→ SUCCESSOR(S, x)
returns i such that $S[i] = min(\{S[j] \mid S[j].key > x\})$, or NIL if no such element was found (i.e., x is larger than or equal to MAXIMUM(S)).
→ PREDECESSOR(S, x)
returns i such that $S[i] = max(\{S[j] \mid S[j].key < x\})$, or NIL if no such element was found (i.e., x is smaller than or equal to MINIMUM(S)).

Stacks and queues

★ Stacks.
- Stack implements a last-ine, first-out (i.e., LIFO) policy.
- Stack operations:
  - INSERT - PUSH; DELETE - POP.
- The stack consists of elements $S[1..S.top]$, where $S[1]$ is the element at the bottom and $S[S.top]$ is the element at the top.
  - $S.top = 0$ - the stack contains no element, i.e., it is empty.
  - Stack underflows - when attempt to pop an empty stack.
  - Stack overflows - S.top exceeds n.

STACK_EMPTY(S)
```
if S.top == 0
    return TRUE
else
    return FALSE
```

PUSH(S, x)
```
S.top = S.top + 1
S[S.top] = x
```

POP(S)
```
if STACK_EMPTY(S)
    error UNDERFLOW
else
    x = S.top
    S.top = S,top - 1
    return x
```

- Analysis: running time of PUSH and POP are $O(1)$.

a pystack

In [1]:

```python
# @author: meng (gmail: y(dot)meng201011)
class Stack:
    """A sample implement of stack with list"""
    def __init__(self):
        self.items = []

    def pop(self):
        if (self.is_empty()):
            return '>>>> UNDERFLOW <<<<'
        else:
            return self.items.pop()

    def push(self, item):
        self.items.append(item)

    def is_empty(self):
        return (self.items == [])
```

In [2]:

```python
stack = Stack()
stack.push('stack')
stack.push('sample')
stack.push('a')
stack.push(1)
stack.push(2)
stack.push(3)
while not stack.is_empty():
    print stack.pop()

print stack.pop()
```

```
3
2
1
a
sample
stack
>>>> UNDERFLOW <<<<
```

★ Queues

◇ Queue implements a first-in, first out (i.e., FIFO) policy.

◇ INSERT - ENQUEUE; DELETE - DEQUEUE

◇ Queue has a head and a tail. Every element will go in from tail and go out from head.

◇ When $Q.head = Q.tail$, the queue is empty.

ENQUEUE(Q, x)

$Q[Q.tail] = x$
$if\, Q.tail == Q.length$
$\quad Q.tail = 1$
$else$
$\quad Q.tail = Q.tail + 1$

DEQUEUE(Q)

$x = Q[Q.head]$
$if\, Q.head == Q.length$
$\quad Q.head = 1$
$else$
$\quad Q.head = Q.head + 1$
$return\, x$

◇ Analysis: Each of ENQUEUE and DEQUEUE takes $O(1)$ time.

a pyqueue

In [3]:

```
# @author meng (gmail: y(dot)meng201011)
class PyQueue:
    """Implement queue using list"""
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if (self.is_empty()):
            return '>>>> UNDERFLOW <<<<'
        else:
            item = self.items[0]
            self.items = self.items[1:]
            return item

    def is_empty(self):
        return (self.items == [])
```

In [4]:

```
queue = PyQueue()
queue.enqueue("a")
queue.enqueue("sample")
queue.enqueue("queue")
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
while not (queue.is_empty()):
    print queue.dequeue()

print queue.dequeue()
```

```
a
sample
queue
1
2
3
>>>> UNDERFLOW <<<<
```

Linked lists (linked list is also a dynamic set)

★ Objects in linked list are arranged in a linear order, which is determined by a pointer in each object.

- ⋄ Linked list supports all operations that are supported by dynamic sets.

★ **Doubly linked list** L has three attributes: key, $next$, and $prev$.

- ⋄ $L.element.next$ points to its successor; $L.element.prev$ points to its predecessor.
- ⋄ $L.head.prev = NIL$ and $L.tail.next = NIL$.
- ⋄ List is empty, if $L.head = NIL$

★ **Singly linked list** has two attributes: key and $next$.

★ **Circular list** has three attributes: key, $next$, and $prev$.

- ⋄ $L.head.prev = L.tail$ and $L.tail = L.head$.
- ⋄ Circular list is a ring.

LIST_SEARCH(L, k) *// Worst-case running time:* $\Theta(n)$

$x = L.head$

$while x != NIL and x.key != k$

$\quad x = x.next$

$return x$

LIST_INSERT(L, x) *// Constant running time.* $O(1)$

$x.next = L.head$

$if L.head != NIL$

$\quad L.head.prev = x$

$L.head = x$

$x.prev = NIL$

LIST_DELETE(L, x)

// Running time of remove one element: $O(1)$

// But we have to find the find the element first, by calling LIST_SEARCH, which costs $\Theta(n)$ *in worst case.*

// Thus, total running time, in worst case, is: $\Theta(n)$

$if x.prev != NIL$

$\quad x.prev.next = x.next$

$else$

$\quad L.head = x.next$

$if x.next != NIL$

$\quad x.next.prev = x.prev$

a singly linked list in python

In [5]:

```
# @author meng (gmail: y(dot)meng201011)
class Node(object):
    """Node in list"""
    def __init__(self, data = None, next = None):
        self.data = data
        self.next = next

class Singly_List(object):
    """singly linked list"""
    def __init__(self):
        self.head = None
        self.tail = None

    def __str__(self):
        return 'Node[%s]' % self.data, 'Next[%s]' % self.next.data

    def is_empty(self):
        return (self.head == None)

    def insert(self, data):
        """insert an item at the head"""
        node = Node(data, None)
        if (self.head is None):
            """insert a node to an empty list"""
```

```
            self.head = self.tail = node
        else:
            """insert new node before the head"""
            node.next = self.head
        """set the head to new node"""
        self.head = node

    def append(self, data):
        """append an item to the tail"""
        node = Node(data, None)
        if (self.head is None):
            """append a node to an empty list"""
            self.head = self.tail = node
        else:
            """append new node to the tail"""
            self.tail.next = node
        """set the tail to new node"""
        self.tail = node

    def printList(self):
        """list of the elements"""
        print 'elements in list are:'
        node = self.head
        while node is not None:
            print node.data + ' -> ',
            node = node.next
        print '(End)'

    def size(self):
        """get count of elements in list"""
        current = self.head
        count = 0
        while (current):
            """count elements thoughout the list"""
            current = current.next
            count += 1
        return count

    def search(self, data):
        """search for node containing given data"""
        current = self.head
        found = False
        while (current is not None) and (not found):
            """go though the list and check"""
            if (current.data == data):
                found = True
            else:
                current = current.next

        if not found:
            raise ValueError('data not found')
        return current

    def delete(self, data):
        """delete the node containing given data"""
        current = self.head
        previous = None

        while (current is not None):
            """go though the list and check"""
            if (current.data == data):
                if previous is not None:
                    """certain node is not the head"""
                    previous.next = current.next
                else:
                    """delete head node"""
                    self.head = current.next
            """node is not found yet"""
            previous = current
            current = current.next
```

In [6]:

```
# sample usage of singly_linked_list
slist = Singly_List()
slist.insert('linked')
slist.insert('singly')
slist.insert('single')
slist.insert('sample')
slist.insert('a')
slist.printList()
slist.append('list')
slist.delete('single')
slist.printList()
print slist.search('linked').data
```

```
elements in list are:
a ->  sample ->  single ->  singly ->  linked ->  (End)
elements in list are:
a ->  sample ->  singly ->  linked ->  list ->  (End)
linked
```

a doubly linked list in python

In [7]:

```
# @author meng (gmail: y(dot)meng201011)
class DNode(object):
    """Node in doubly linked list"""
    def __init__(self, data = None, prev = None, next = None):
        self.data = data
        self.prev = prev
        self.next = next

class Doubly_List(object):
    """doubly linked list"""
    def __init__(self):
        self.head = None
        self.tail = None

    def insert(self, data):
        """insert a node to head"""
        node = DNode(data)
        if (self.head is None):
            """original list is empty"""
            self.head = self.tail = node
        else:
            """list is not empty"""
            node.next = self.head
            self.head.prev = node
        """set new node as list.head"""
        self.head = node

    def printList(self):
        """list of the elements"""
        print 'elements in the doubly list are:'
        node = self.head
        while node is not None:
            print node.data + ' -> ',
            node = node.next
        print '(End)'

    def delete(self, data):
        """delete the node containing given data"""
        current = self.head
        """go through the list"""
        while (current is not None):
            if (current.data == data):
                if (current.prev is None):
                    """delete the head"""
                    self.head = self.next
                else:
                    """delete a node in the middle"""
                    current.prev.next = current.next
                    current.next.prev = current.prev
            """not found yet"""
            current = current.next
```

```
            current = current.next

    def append(self, data):
        node = DNode(data)
        if (self.head is None):
            """original list is empty"""
            self.head = self.tail = node
        else:
            """list is not empty"""
            self.tail.next = node
            node.prev = self.tail
        """set new node as list.tail"""
        self.tail = node

    def size(self):
        """return number of elements in the list"""
        count = 0
        current = self.head
        while (current is not None):
            current = current.next
            count += 1
        return count

    def search(self, data):
        current = self.head
        found = False
        """go through the list checking the data"""
        while (current is not None) and (not found):
            if (current.data == data):
                found = True
            else:
                current = current.next
        if (not found):
            raise ValueError('element not found!')
        return current

    def is_empty(self):
        return (self.head == None)
```

In [8]:

```
doubly_list = Doubly_List()
doubly_list.append('linked')
doubly_list.append('list')
doubly_list.insert('double')
doubly_list.insert('doubly')
doubly_list.insert('sample')
doubly_list.printList()
doubly_list.delete('double')
doubly_list.printList()
print doubly_list.search('sample')
```

```
elements in the doubly list are:
sample ->  doubly ->  double ->  linked ->  list ->  (End)
elements in the doubly list are:
sample ->  doubly ->  linked ->  list ->  (End)
<__main__.DNode object at 0x7faae12b3c10>
```

a circular linked list in python

In [9]:

```
class CNode(object):
    def __init__(self, data = None, next = None):
        self.data = data
        self.next = next

class CircularList(object):
    def __init__(self):
        self.head = None
        self.tail = None
        if (self.tail is not None):
            self.tail.next = self.head

    def insert(self, data):
        node = CNode(data)
        if (self.head is None):
```

```
        if (self.head is None):

            """list was empty"""
            self.head = node
            self.tail = node
            self.head.next = self.tail
        else:
            """list was NOT empty"""
            node.next = self.head
            self.head = node
        self.tail.next = self.head

    def append(self, data):
        node = CNode(data)
        if (self.head is None):
            """list was empty"""
            self.head = node
            self.tail = node
            self.head.next = self.tail
        else:
            """list was NOT empty"""
            self.tail.next = node
            self.tail = node
        self.tail.next = self.head

    def search(self, data):
        current = self.head
        found = False
        all_touched = False
        count = -1
        while (not all_touched) and (not found):
            if (current.next == self.head.next):
                count += 1
            if (count >= 1):
                all_touched = True
            if (current.data == data):
                found = True
            else:
                current = current.next
        if (not found):
            raise ValueError('element is not found')
        return current

    def delete(self, data):
        current = self.head
        prev = None
        all_touched = False

        while (not all_touched):
            if (current.next == self.head) and (not all_touched):
                all_touched = True

            """go though the list and check"""
            if (current.data == data):
                if (prev is not None):
                    """certain node is not the head"""
                    prev.next = current.next
                else:
                    """delete the head"""
                    self.head = current.next
                    self.tail.next = self.head
            else:
                prev = current

            current = current.next
            #self.print_list()


    def is_empty(self):
        return (self.head is None)

    def print_list(self):
        node = self.head
        all_touched = False

        while (not all_touched) and (node is not None):
            if (node.next == self.head) and (not all_touched):
```

```
            all_touched = True


        print node.data + ' -> ',
        node = node.next
    print '(End)'
```

In [10]:

```
circular_list = CircularList()
circular_list.insert('sample')
circular_list.append('circular')
circular_list.append('sample')
circular_list.insert('linked')
circular_list.insert('sample')
circular_list.insert('sample')
circular_list.insert('sample')
circular_list.insert('sample')
circular_list.append('list')
circular_list.append('sample')
circular_list.append('sample')
circular_list.insert('sample')
circular_list.insert('sample')
circular_list.insert('sample')
circular_list.print_list()
print '\n>>>> DELETE NODE[Data = sample]\n'
circular_list.delete('sample')
circular_list.print_list()
```

```
sample ->  sample ->  sample ->  sample ->  sample ->  sample ->  sample ->  linked ->  sample ->  circular
 ->  sample ->  list ->  sample ->  sample ->  (End)

>>>> DELETE NODE[Data = sample]

linked ->  circular ->  list ->  (End)
```

Binary tree

In [ ]:

```
```