Lecture 01/20/17

Lecturer: Xiaodi Wu

Reading: Chapter 10.1, 10.2
Abstract Data Type (ADT)

Stack
A **stack** is a container of objects that are inserted and removed according to the **last-in first-out (LIFO)** principle.

Stack: operations
- `push(o)`: insert object `o` at the top of the stack.
- `pop()`: remove and return the top of the stack.
- `size()`: return the number of objects in the stack.
- `isEmpty()`: return a Boolean indicating if the stack is empty.
- `top()`: return the top of the stack.
Abstract Data Type (ADT)

Stack
A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

Stack: operations

- push(o): insert object o at the top of the stack.
- pop(): remove and return the top of the stack.
Abstract Data Type (ADT)

Stack
A **stack** is a container of objects that are inserted and removed according to the **last-in first-out (LIFO)** principle.

Stack: operations

- **push(o):** insert object o at the top of the stack.
- **pop():** remove and return the top of the stack.
- **size():** return the number of objects in the stack.
- **isEmpty():** return a Boolean indicating if the stack is empty.
- **top():** return the top of the stack.
Definition
The $n$th Fibonacci number $F(n)$ is defined recursively as $F(n) = F(n-1) + F(n-2)$ for $n > 1$ with $F(0) = 0$, $F(1) = 1$. 

Algorithm
```python
def Fib(n):
    if n > 1:
        return Fib(n-1) + Fib(n-2)
    else:
        return n;
```
Definition
The $n$th Fibonacci number $F(n)$ is defined recursively as
$F(n) = F(n - 1) + F(n - 2)$ for $n > 1$ with $F(0) = 0, F(1) = 1$.

Algorithm Fib($n$)

if $n > 1$ then
    return Fib($n-1$)+Fib($n-2$)
else
    return $n$;
end if

Test run Fib(4).
Definition
The $n$th Fibonacci number $F(n)$ is defined recursively as
$F(n) = F(n - 1) + F(n - 2)$ for $n > 1$ with $F(0) = 0$, $F(1) = 1$.

Algorithm $\text{Fib}(n)$
if $n > 1$ then
    return $\text{Fib}(n-1) + \text{Fib}(n-2)$
else
    return $n$;
end if

- Test run $\text{Fib}(4)$. 
Implementation with an $N$-element array $S$, with elements stored from $S[0]$ to $S[t]$, where $t$ is the index of the top element. **Note:** arrays start at index 0 and thus $t$ is initialized to -1.
Implementation with an $N$-element array $S$, with elements stored from $S[0]$ to $S[t]$, where $t$ is the index of the top element. **Note:** arrays start at index 0 and thus $t$ is initialized to -1.

- `size()`: return $t+1$;
- `isEmpty()`: return True if $t=-1$; else return False;
- `top()`: return $S[t]$;
Stack: Array-based Implementation

**Algorithm** push(o)

if size() = N then
    stack-full exception
end if

\[ t \leftarrow t + 1 \]

\[ S[t] \leftarrow o \]

**Algorithm** pop()

if isEmpty() then
    stack-empty exception
end if

\[ e \leftarrow S[t] \]

\[ S[t] \leftarrow \text{null} \]

\[ t \leftarrow t - 1 \]

return \( e \).
Abstract Data Type (ADT)

Queue

A queue is a container of objects that are inserted and removed according to the first-in first-out (FIFO) principle. Enter at the rear and remove from the front.
Abstract Data Type (ADT)

Queue
A queue is a container of objects that are inserted and removed according to the first-in first-out (FIFO) principle. Enter at the rear and remove from the front.

Stack: operations

- enqueue(o): insert object o at the rear of the queue.
- dequeue(): remove and return the front of the queue.
Abstract Data Type (ADT)

Queue
A queue is a container of objects that are inserted and removed according to the first-in first-out (FIFO) principle. Enter at the rear and remove from the front.

Stack: operations

- enqueue(o): insert object o at the rear of the queue.
- dequeue(): remove and return the front of the queue.
- size(): return the number of objects in the queue.
- isEmpty(): return a Boolean indicating if the queue is empty.
- front(): return the front of the queue.
Queue: Array-based Implementation

Implementation with an $N$-element array $Q$, with elements stored from $S[f]$ to $S[r - 1]$, where $f$, $r - 1$ refer to the indices of the front and the rear of the queue. $f == r$ implies an empty queue.
Queue: Array-based Implementation

Implementation with an $N$-element array $Q$, with elements stored from $S[f]$ to $S[r - 1]$, where $f$, $r - 1$ refer to the indices of the front and the rear of the queue. $f == r$ implies an empty queue. Q: what if $r$ gets bigger than $N$?
Queue: Array-based Implementation

Implementation with an \( N \)-element array \( Q \), with elements stored from \( S[f] \) to \( S[r - 1] \), where \( f, r - 1 \) refer to the indices of the front and the rear of the queue. \( f == r \) implies an empty queue. 

Q: what if \( r \) gets bigger than \( N \)?

- **size():** return \((N + (r - f)) \mod N\).
Queue: Array-based Implementation

Implementation with an $N$-element array $Q$, with elements stored from $S[f]$ to $S[r - 1]$, where $f$, $r - 1$ refer to the indices of the front and the rear of the queue. $f == r$ implies an empty queue. Q: what if $r$ gets bigger than $N$?

- size(): return $(N + (r - f)) \mod N$.
- isEmpty(): return True if $r == f$; else return False;
- front(): return $S[f]$;
Queue: Array-based Implementation

**Algorithm** enqueue(o)
if size() = N-1 then
    queue-full exception
end if
Q[r] ← o
r ← (r + 1) mod N

**Algorithm** dequeue()
if isEmpty() then
    queue-empty exception
end if
e ← Q[f]
Q[f] ← null
f ← (f + 1) mod N
return e.
Linked-List Implementation

Linked-List
ADT with objects in a linear order determined by pointer(s) in each object.

Search key \( k \) in the list \( L \):
Algorithm List-Search \( (L, k) \)
\[
x \leftarrow L.\text{head}
\]
\[
\text{while } x \neq \text{NIL} \text{ and } x.\text{key} \neq k \text{ do}
\]
\[
x \leftarrow x.\text{next}
\]
\[
\text{end while}
\]
\[
\text{return } x. \quad \text{(Key found if } x \neq \text{NIL} \text{)}
\]
Linked-List Implementation

Linked-List
ADT with objects in a linear order determined by pointer(s) in each object.

Components: (doubly linked-list)

- Each object $x$ has a key and pointers next and prev.
- **head** when $x.prev = \text{NIL}$; **tail** when $x.next = \text{NIL}$.
- many variants: singly/doubly linked, circular or not, w/ or w/o sentinels.

Search key $k$ in the list $L$:

Algorithm List-Search ($L, k$)

$x \leftarrow L.head$

while $x \neq \text{NIL}$ and $x.key \neq k$

$x \leftarrow x.next$

end while

return $x$. (Key found if $x \neq \text{NIL}$)
Linked-List Implementation

Linked-List
ADT with objects in a linear order determined by pointer(s) in each object.

Components: (doubly linked-list)
- Each object \( x \) has a key and pointers next and prev.
- head when \( x.\text{prev}=\text{NIL} \); tail when \( x.\text{next}=\text{NIL} \).
- many variants: singly/doubly linked, circular or not, w/ or w/o sentinels.

Search key \( k \) in the list \( L \):

Algorithm List-Search \((L,k)\)

\[
x \leftarrow L.\text{head} \\
\text{while } x \neq \text{NIL} \text{ and } x.\text{key} \neq k \text{ do} \\
\quad x = x.\text{next} \\
\text{end while} \text{return } x. \text{ (Key found if } x \neq \text{NIL)}
\]
Linked-List Insertion and Deletion

**Algorithm** List-Insert($L, x$)

Insert $x$ to the front of $L$

$x.next = L.head$

if $L.head \neq NIL$ then

$L.head.prev = x$

end if

$L.head = x$

$x.prev = NIL$

**Algorithm** List-Delete($L, x$)

remove $x$ from $L$

if $x.prev \neq NIL$ then

$x.prev.next = x.next$

else

$L.head = x.next$

end if

if $x.next \neq NIL$ then

$x.next.prev = x.prev$

end if
FIFO vs LIFO

FIFO implemented by 2 LIFOs
FIFO vs LIFO

FIFO implemented by 2 LIFOs

- enqueue(o): stack2.push(o).

- dequeue():
  - if (!stack1.isEmpty()) then return stack1.pop();
  - else while (!stack2.isEmpty()) do {
    - o = stack2.pop(); stack1.push(o);
  }
  - return stack1.pop();
FIFO vs LIFO

FIFO implemented by 2 LIFOs

- enqueue(o): stack2.push(o).
- dequeue(): if (! stack1.isEmpty()) then return stack1.pop(); else while (! stack2.isEmpty()) do { o=stack2.pop(); stack1.push(o); } return stack1.pop();
FIFO vs LIFO

FIFO implemented by 2 LIFOs

- enqueue(o): stack2.push(o).
- dequeue(): if (! stack1.isEmpty()) then return stack1.pop();
  else while (! stack2.isEmpty()) do
    { o=stack2.pop(); stack1.push(o); }
  return stack1.pop();

Question: LIFO implemented by 2 FIFOs?