CECS 274: Data Structures
Linked Lists

Oscar Morales Ponce

California State University Long Beach

Pointer-Based Lists

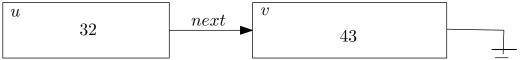
- Using references (pointers), the nodes are linked together into a sequence
- Node for Singly-linked list (SLList)
  - State:
    - $next$: next node in the sequence or nil if it is the last node.
    - x: The data to store.

u

Variable	Memory Address	value
$next$	0x01FFFFBB	0x04AFCD0A
$data$	0x02FFFFBA	32

v

Variable	Memory Address	value
$next$	0x04AFCD0A	0x00000000
$data$	0x05AFCD09	43



- Disadvantage?
- Advantages?

SLList: A Singly-Linked List

- SLList (singly-linked list) is a sequence of Nodes.
  - State:
    - $head$: First Node in the list.
    - $tail$: Last Node in the list.
    - n: The number of elements in n.
  - Invariant: $\begin{cases} \text{If } n = 0\text{, then } head = tail = nil \\ \text{If } n = 1\text{, then } head = tail \neq nil \\ \text{If } n > 1\text{, then } head \neq tail \neq nil \end{cases}$

SLList: The Basics

initialize()

Set $head$ and $tail$ to nil

Initialize the number of elements stored to zero

SLList: $initialize()$

initialize()

$n \leftarrow 0$

$head \leftarrow nil$

$tail \leftarrow nil$

SLList: Python

```
class SLList(Stack):
    class Node() :
        def __init__(self, x):
            self.next = None
            self.x = x

    def __init__(self):
        self.head = None
        self.tail = None
        self.n = 0
```

SLList: Java

```
public class SLList implements Stack  {
    class Node {
       Node next;
       Object x;

       Node(Object x)
       {
          this.x = x;
       }
     }
     protected int n;
     protected Node head, tail;

     public SLList()  {
        n = 0;
        head = tail = null;
     }
```

SLList: Implementing $push(x)$ of a Stack

- $push(x)$: Insert a node in the head of the list

$\text{push}(x)$
  Create a node u storing x
  The head becomes the second node and u the head
  Ensure that the invariant still holds
  Increment n by one

SLList: Implementing $push(x)$ of a Stack

$\text{push}(x)$
$\quad u \leftarrow \text{new\_node}(x)$
$\quad u.next \leftarrow head$
$\quad head \leftarrow u$
$\quad$ **if** $n = 0$ **then**
$\qquad tail \leftarrow u$
$\quad n \leftarrow n + 1$
$\quad$ **return** x

- How many operations?

SLList: Implementing $pop()$ of a Stack

- $pop(x)$: Return the value of the head and remove it from the list

pop()
    Check Precondition
    Let x be the value of the head
    Remove the head
    Decrement n by one
    Ensure that the invariant still holds

SLList: Implementing $pop()$ of a Stack

pop()

if $n = 0$ **then return** nil

$x \leftarrow head.x$

$head \leftarrow head.next$

$n \leftarrow n - 1$

if $n = 0$ **then**

  $tail \leftarrow nil$

return x

- How many operations?

SLList: $pop()$ (Discussion Activity)

Implement the function $pop()$ in Python

SLList: Implementing $remove()$ of a FIFO

- Removals are done from the head of the list, and are identical to the $\mathrm{pop}()$ operation:

```
remove()
    return pop()
```

SLList: Implementing $add(x)$ of a FIFO (Discussion Activity: Write the step)

- $add(x)$: Insert a node in the tail of the list.

$\mathrm{add}(x)$

SLList: Implementing $add(x)$ of a FIFO

$\mathrm{add}(x)$
    $u \leftarrow \mathrm{new\_node}(x)$
    if $n = 0$ **then**
        $head \leftarrow u$
    else
        $tail.next \leftarrow u$
    $tail \leftarrow u$
    $n \leftarrow n + 1$
    return $true$

- How many operations?

SLList: Summary

Theorem

An SLList implements the Stack and FIFO Queue interfaces. The $\mathrm{push}(x)$*,* $\mathrm{pop}()$*,* $\mathrm{add}(x)$ *and* $\mathrm{remove}()$ *operations run in* $O(1)$ *time per operation.*

- An SLList nearly implements the full set of Deque operations.
- The only missing operation is removing from the tail of an SLList.
- Removing from the tail of an SLList is difficult because it requires updating the value of $tail$ so that it points to the node w that precedes $tail$ in the SLLis.
- Unfortunately, the only way to get to w is by traversing the SLList starting at $head$ and taking $n - 2$ steps.

SLList: $get\_node(i)$

- $get\_node(i)$. Get the i-th node in the list

> $\mathrm{get\_node}(i)$
> Check precondition
> Start at the $head$ of the list and work forward

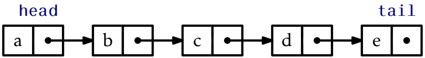
- How many operations?

SLList: $get\_node(i)$

```
get_node(i)
    if n < 0 or  i ≥ n then return nil
    u = head
    repeat i times
        u = u.next
    return u
```

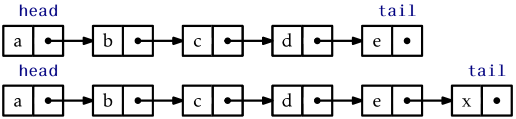
- How many operations?

SLList: Example



add(x)

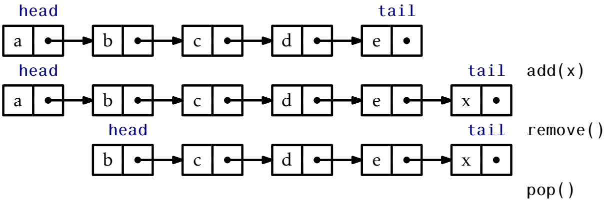
SLList: Example



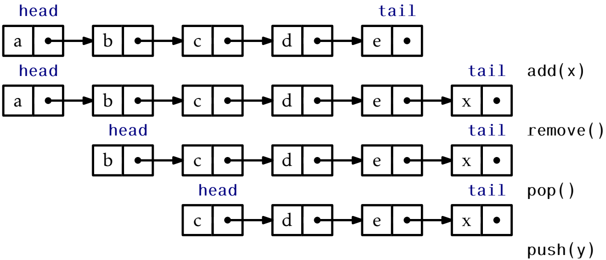
add(x)

remove()

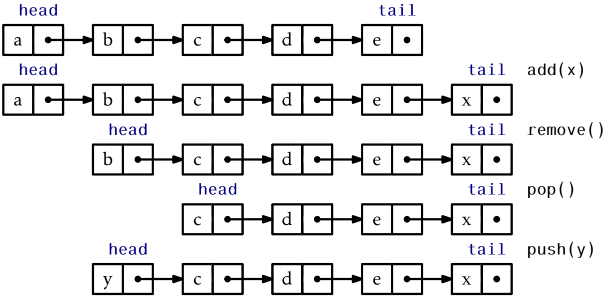
SLList: Example



SLList: Example



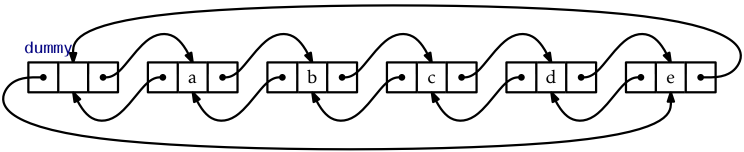
SLList: Example



DLList: A Doubly-Linked List

- DLList is a sequence of Nodes pointing to the next and previous node.
  - State:
    - $dummy$: A dummy node that does not contain data.
    - n: The number of elements in n.
  - Invariant:

$$\begin{cases} \text{If } n = 0\text{, then } dummy.next = dummy.prev = dummy \\ \text{If } n > 0\text{, then } dummy.next \text{ is the head} \\ \qquad \text{and } dummy.prev \text{ is the tail} \end{cases}$$



DLList: Initialize

$\text{initialize}()$

Create the node $dummy$ and initialize it

Initialize the number of elements to zero

DLList: Initialize

initialize()

$n \leftarrow 0$

$dummy \leftarrow new\_node(nil)$

$dummy.prev \leftarrow dummy$

$dummy.next \leftarrow dummy$

DLList: $get\_node(i)$

- $get\_node(i)$. Get the i-th node in the list

$\mathrm{get\_node}(i)$
    Check preconditions
    if $i < n/2$ start at the head of the list and work forward
    if $i \geq n/2$ start at the tail of the list and work backwards

DLList: Implementing $get\_node(i)$ a node (Discussion Activity: Pseudo-code)

$\text{get\_node}(i)$

- How many operations?

DLList: Implementing $get(i)$ and $set(i, x)$

- $get(i)$: Return the value of the i-th node
- $set(i, x)$: Set the value of the i-th node and returns the old value

$\mathrm{get}(i)$
    Check Preconditions
    Find the i-th node and then return its value x

$\mathrm{set}(i, x)$
    Check Preconditions
    Find the i-th node
    Store its value in y, set the new value and return y

DLList: Implementing $get(i)$ and $set(i, x)$

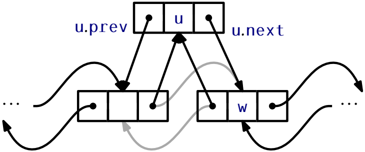
$\text{get}(i)$
    if $i < 0$ **or** $i >= n$ **then** Exception
    return $\text{get\_node}(i).x$

$\text{set}(i, x)$
    if $i < 0$ **or** $i >= n$ **then** Exception
    $u \leftarrow \text{get\_node}(i)$
    $y \leftarrow u.x$
    $u.x \leftarrow x$
    return y

- The running time of these operations is dominated by the time it takes to find the ith node,: $O(1 + \min\{i, n - i\})$.

DLList: $add\_before(w, x)$

- $add\_before(w, x)$ (Helper function): Given a node w, inserts a new node before w.



$\text{add\_before}(w, x)$
Check Preconditions
Create a new node u
Update $next$, $prev$ of u and w
Increment the number of elements by one

DLList: $add\_before(w, x)$

$\text{add\_before}(w, x)$

if w is nil **then return** Exception

$u \leftarrow \text{new\_node}(x)$

$u.prev \leftarrow w.prev$

$u.next \leftarrow w$

$u.next.prev \leftarrow u$

$u.prev.next \leftarrow u$

$n \leftarrow n + 1$

return u

DLList: $add(i, x)$

- $add(i, x)$: Add a new node containing x at position i.

```
add(i, x)
    Check Preconditions
     Find the ith node in the DLList
     Insert a new node u that contains x just before it
```

DLList: $add(i, x)$

$\mathrm{add}(i, x)$
  if $i < 0$ **or** $i \geq n$ **then return** Exception
  $\mathrm{add\_before}(\mathrm{get\_node}(i), x)$

- $\mathrm{add}(i, x)$ runs in $O(1 + \min\{i, n - i\})$ time.

DLList: $remove(i)$

- $remove(i)$: Remove the ith node

remove(i)
  Check Preconditions
  Let w be the ith node in the DLList
  Adjust the pointers at $w.next$ and $w.prev$ to skip over w

DLList: $remove(i)$ (Discussion Activity Pseudo-code)

$\text{remove}(i)$

- How many operations?

DLList: Summary

Theorem

A DLList implements the List interface. In this implementation, the $\mathrm{get}(i)$, $\mathrm{set}(i, x)$, $\mathrm{add}(i, x)$ *and* $\mathrm{remove}(i)$ *operations run in* $O(1 + \min\{i, n - i\})$ *time per operation.*

- Queue operations (constant time)
  - $add(x)$: $add(0, x)$
  - $remove()$: $remove(n - 1)$
- Stack operation (constant time)
  - $push(x)$: $add(0, x)$
  - $pop()$: $pop(0)$
- Deque operation (constant time)
  - $add\_fist(x)$: $add(0, x)$
  - $add\_last(x)$: $add(n, x)$
  - $remove\_first(x)$: $remove(0)$
  - $remove\_last(x)$: $remove(n - 1)$

DLList: Notes

- The only expensive part of operations on a DLList is finding the relevant node.
- In array-based List implementations the relevant array item can be found in constant time. However, addition or removal requires shifting elements in the array and, in general, takes non-constant time.
- For this reason, linked list structures are well-suited to applications where references to list nodes can be obtained through external means.