COMP 250

Lecture 10

singly linked lists

Jan. 28, 2022
Lists

• array list  (last lecture)

• singly linked list  (today)

• doubly linked list  (next lecture)
Linked lists are a special case of both of the following which are core topics to be covered later this course:
array list

linked list

array

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

size = 4

sequence of “node” objects

null

null
Array slots are in *consecutive locations* (addresses) in memory. Objects can be anywhere.

Linked list “nodes” (and objects that they reference) can be *anywhere in memory*. 
Singly linked list node ("S" for singly)

class SNode<E> {

    SNode<E> next;
    E element;

}

e.g. E might be Shape

As we saw last lecture, a generic type E must be a reference type.

The next field references an SNode object, namely the "next" node in the list.
A linked list consists of a sequence of nodes, along with a reference to the first (head) and last (tail) node.
Example: a singly linked class

class SLinkedList<E> {

    SNode<E> head;
    SNode<E> tail;
    int size;

    :

    private class SNode<E> {

        // inner class

        SNode<E> next;
        E element;

        :
    }
}

“Inner” (versus “Outer”) Class

class SLinkedList<E> {
    SNode<E>    head;
    SNode<E>    tail;
    int         size;

    : 

    private  class SNode<E> {
        // inner class

        SNode<E>    next;
        E           element;

        :
    }
}

Outer classes have either public or package visibility, but not private.

Often inner classes are private.
How many objects?

$1 + 4 + 4 = 9$

SLinkedList  SNode  Shape
Linked list operations

- addFirst (e) - adds new item to front of list
- removeFirst()
- addLast (e) - adds new item to back (end) of list
- removeLast()
- .......
addFirst ( e )

BEFORE

AFTER

head

next element

null

tail
addFirst (e)
addFirst (e)

What to do first?
addFirst (e)

construct newNode
newNode.element = e
newNode.next = head
addFirst ( e )

```plaintext
construct newNode
newNode.element = e
newNode.next = head

head = newNode
size = size+1
```
addFirst ( e )

construct newNode
newNode.element = e
newNode.next = head

head = newNode
size = size+1

// special case: list was empty
if size == 1
    tail = head
What about `removeFirst()`?
removeFirst ()

BEFORE

tmp = head
if (size == 0)
    throw exception
head = head.next
size = size – 1
if (size == 0)
tail = null
return tmp.element
removeFirst()
removeFirst ( )

tmp = head
if (size == 0)
    throw exception

head = head.next
size = size – 1

// special case: list is empty
if (size == 0)
    tail = null

return tmp.element
removeFirst ( )

When removeFirst() finishes, the tmp variable is out of scope, and the gray node is garbage (no longer accessible).
addLast ( e )

BEFORE

next element

head

null
tail

e
addLast ( e )

tmp = new Node

tmp.element = e

tail.next = tmp

head

tail

tmp

next element

null
addLast ( e )

tmp = new Node
tmp.element = e
tail.next = tmp

tail = tail.next // or tail = tmp

size = size + 1
What about `removeLast()`?

**BEFORE**

Problem: we have no direct way to access the node before `tail`.

Problem: we have no direct way to access the node before `tail`. 

26
removeLast ( ) \rightarrow \text{ case of size == 1}

```java
if (size == 1){
    head = null
    tail = null
}
else {
    tmp = head
    while (tmp.next != tail)
        tmp = tmp.next
    tail = tmp
    tail.next = null
}
size = size - 1
```
removeLast( ) general case

if (size == 1) {
    head = null
    tail  = null
}
else {
    tmp = head
    while (tmp.next != tail)
        tmp = tmp.next
    tail = tmp
    tail.next = null
    size = size - 1
removeLast()

if (size == 1){
    head = null
    tail = null
}
else {
    tmp = head
    while (tmp.next != tail)
        tmp = tmp.next
    tail = tmp
    tail.next = null
    size = size - 1
}

removeLast ( )

if (size == 1){
    head = null
    tail = null
}
else {
    tmp = head
    while (tmp.next != tail)
        tmp = tmp.next
    tail = tmp
    tail.next = null
    size = size - 1
removeLast()

if (size == 1){
    head = null
    tail = null
}
else {
    tmp = head
    while ( tmp.next != tail )
        tmp = tmp.next
    tail = tmp
    tail.next = null
}
size = size - 1
removeLast()
removeLast ( )

e = tail.element
if (size == 1){
    head = null
    tail = null
}
else {
    tmp = head
    while ( tmp.next != tail )
        tmp = tmp.next
    tail = tmp
    tail.next = null
}
size = size – 1
return e
Computational Complexity

Let $N$ be the size of the input.

If an operation takes “constant time” (independent of $N$), then we say it has complexity $O(1)$.

If an operation takes time proportional to $N$, then we say it has complexity $O(N)$.

If an operation takes time proportional to $N^2$, then we say it has complexity $O(N^2)$. 
Computational Complexity  \((N = \text{list size})\)

<table>
<thead>
<tr>
<th></th>
<th>array list</th>
<th>SLinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFirst</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>removeFirst</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>addLast</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>removeLast</td>
<td>(O(1))</td>
<td>(O(N))</td>
</tr>
</tbody>
</table>
### Computational Complexity

$(N = \text{list size})$

<table>
<thead>
<tr>
<th>List Type</th>
<th>addFirst</th>
<th>removeFirst</th>
<th>addLast</th>
<th>removeLast</th>
</tr>
</thead>
<tbody>
<tr>
<td>array list</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>SLinkedList</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$?$</td>
<td>$?$</td>
</tr>
</tbody>
</table>

*Diagram shows computational complexity for different list operations.*
<table>
<thead>
<tr>
<th>Operation</th>
<th>Array List</th>
<th>SLinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFirst</td>
<td>?</td>
<td>O(1)</td>
</tr>
<tr>
<td>removeFirst</td>
<td>?</td>
<td>O(1)</td>
</tr>
<tr>
<td>addLast</td>
<td></td>
<td>O(1)</td>
</tr>
<tr>
<td>removeLast</td>
<td></td>
<td>O(N)</td>
</tr>
</tbody>
</table>
### Computational Complexity  \( (N = \text{list size}) \)

<table>
<thead>
<tr>
<th></th>
<th>array list</th>
<th>SLinkedlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFirst</td>
<td>( O(N) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>removeFirst</td>
<td>( O(N) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>addLast</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>removeLast</td>
<td>( O(N) )</td>
<td></td>
</tr>
</tbody>
</table>

An array list would use \( \text{add}(0, \ e) \) rather than \( \text{addFirst}(\ e) \) and \( \text{remove}(0) \) instead of \( \text{removeFirst()} \), ....
Computational Complexity \((N = \text{list size})\)

<table>
<thead>
<tr>
<th></th>
<th>array list</th>
<th>SLinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFirst</td>
<td>O( N )</td>
<td>O( 1 )</td>
</tr>
<tr>
<td>removeFirst</td>
<td>O( N )</td>
<td>O( 1 )</td>
</tr>
<tr>
<td>addLast</td>
<td>O( 1 )</td>
<td>O( 1 )</td>
</tr>
<tr>
<td>removeLast</td>
<td>O( 1 )</td>
<td>O( N )</td>
</tr>
</tbody>
</table>

But only if there is available space. Otherwise it takes time O(N).

If the array is full, then it still takes time O(N). We need to make a bigger array and then write N+1 elements to the bigger array.
### Coming up...

#### Lectures

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon. Jan. 31</td>
<td>Doubly Linked Lists</td>
</tr>
<tr>
<td>Wed. Feb.  2</td>
<td>Quadratic Sorting</td>
</tr>
<tr>
<td>Fri. Feb.  4</td>
<td>Object Oriented Design 1</td>
</tr>
<tr>
<td></td>
<td>(Inheritance)</td>
</tr>
</tbody>
</table>

#### Assessments

<table>
<thead>
<tr>
<th>Date</th>
<th>Assessment Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Today</td>
<td>Quiz 1 (lectures 1-7)</td>
</tr>
<tr>
<td></td>
<td>Assignment 1</td>
</tr>
<tr>
<td></td>
<td>- you will have 2 weeks to do it</td>
</tr>
</tbody>
</table>