CS 171: Introduction to Computer Science II

Linked List, Stacks and Queues

Li Xiong
Roadmap

• Basic data structure
  – Arrays

• Abstract data types
  – Stacks
  – Queues
  – Implemented using resizing arrays

• Linked List

• Re-implementing Stacks and Queues using Linked List
Linked List

- A linked list is a recursive data structure that is either empty (null) or a reference to a node having a data item and a reference to a linked list
  - A sequence of nodes chained together
  - Each node contains a data item and a reference to next node
Node

class Node {
    Item item;       \rightarrow Data
    Node next;      \rightarrow Reference to the next node
}

• This is **self-referential**: a class containing a reference to itself
• A linked list is referred by the reference to the first node
Object vs. Object Reference

object          object reference

1600 Pennsylvania Ave
Washington, D.C.
Linked List

A chain of objects

First object in the chain

What is actually stored in memory
Linked List vs. Arrays

• Arrays
  – stores elements continuously in memory
  – Fixed size
  – supports indexed access

• Linked list
  – Does not store elements continuously in memory
  – supports dynamic size (create a node as needed)
  – Does not support indexed access
  – incurs some memory overhead due to the need to store references
Building a linked list

• Example: to build a linked list that contains the items “to”, “be”, and “or”

• Create a Node for each item
  – set the item field to the desired value
  – set the next field to next node

• Maintains a link to the first node of the list, also called root, head
Linked List operations

• Traverse a linked list
• Search an item with a key
• Insert an item (at beginning and end)
• Delete an item (at beginning and end, with a given key)
Traversing a linked list

• Example: print out the values of the linked list
Traversing a linked list

- Example: print out the values of the linked list
- Traversing a linked list

```java
for (Node x = first; x != null; x = x.next) {
    // process x.item
}
```

- Traversing an array

```java
for (int i = 0; i < N; i++) {
    // process a[i]
}
```
Search in a linked list

• Example: search if there is “be” in the linked list
Search in a linked list

• Example: search if there is “be” in the linked list

    for (Node x = first; x != null; x = x.next) {
        if x.item.equals(“be”)
            return x;
    }

• Search in a linked list
Insert at the beginning

• Example: insert “not” at the beginning
Insert at the beginning

• Example: insert “not” at the beginning
• What if the list is empty, i.e. first is null?

```java
// create a new node
Node x = new Node();

// set the new node
x.item = "not";
```

```java
x.next = first;
```

```java
// update first
first = x;
```
Insert at the beginning – book version

• Example: insert “not” at the beginning

```java
// save a link to first
Node oldfirst = first;

// create a new first
first = new Node();

// set the new first
first.item = “not”;
First.next = oldfirst;
```
Remove from the beginning

- Example: remove “to” at the beginning
Remove from the beginning

- Example: remove “to” at the beginning
- What if the list is empty, i.e. first is null?
Insert/remove at the end

- Example: insert “not” at the end
- Example: remove “or” at the end
Insert/remove at the end

• Example: insert “not” at the end
• Example: remove “or” at the end
• Traverse the list to find last node, then insert/remove

```java
for (Node x = first; x!= null; x = x.next); //?
```
Insert/remove at the end

• Example: insert “not” at the end

• Example: remove “or” at the end

• Traverse the list to find last node, then insert/remove

```java
for (Node x = first; x!= null; x = x.next);  // x is null after the loop

for (Node x = first; x.next != null; x = x.next);  // x is the last node after the loop
```
Double-ended Linked List

• Similar to an ordinary linked list, but in addition to keep ‘first’, also keeps a reference to the ‘last’ element in the list.
• What happens when the list is empty? Has only one element?
Remove a given item

- Example: remove “be” from the linked list
Remove a given item

- Example: remove “be” from the linked list
- Search the item, then remove

```java
for (Node x = first; x != null; x = x.next) {
    if (x.item.equals("be")) {
        // how to remove x?
    }
}
```
Remove a given item

- Example: remove “be” from the linked list
- Search the item, then remove

Node x = first;
while (!x.next==null && !x.next.item.equals("be"))
    x = x.next;
node.next = x.next.next;
Remove a given item

- Example: remove “be” from the linked list
- Search the item, then remove
- Keep a reference to the previous and current element

Node current = first;
Node previous = first;
while (current != null && !current.item.equals("be")){
    previous = current;
    current = current.next;
}
// remove current
previous.next = current.next;

// What if the item is the first node?
// What if the item does not exist?
Remove a given item

- Example: remove “be” from the linked list
- Search the item, then remove it
- Need to keep a reference to the previous and current element.
- Need to consider the cases when item is the first and when item does not exist

Node current = first;
Node previous = first;
while (current != null && !current.item.equals("be")){
    previous = current;
    current = current.next;
}
// remove current
if (current == first)
    first = first.next;
else if (current != null)
    previous.next = current.next;
Doubly linked list

• A doubly linked list has bidirectional references, one to the next, one to the previous link
• Pros: flexibility
• Cons: complexity, memory overhead
Linked List

• (Singly) linked list
• Double ended linked list
• Doubly linked list
Halloween Costume – Linked List
Doubly Linked List
Circularly Linked List
Roadmap

• Linked List

• Re-implementing Stacks and Queues using Linked List

• Interface and Iterators

• Java collections classes

• Hw3 discussion
Stacks and queues

Fundamental data types.
- Value: collection of objects.
- Operations: *insert*, *remove*, *iterate*, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

**Stack.** Examine the item most recently added.  
**Queue.** Examine the item least recently added.

LIFO = "last in first out"  
FIFO = "first in first out"
Maintain pointer to first node in a linked list; insert/remove from front.

- `to` node: insert at front of linked list
- `be` node: first node
- `not` node: remove from front of linked list
- `or` node
- `to` node
- `-` node
- `not` node
- `be` node
Stack push: linked-list implementation

inner class

public class Node {
    String item;
    Node next;
    ...
}

save a link to the list

Node oldfirst = first;

create a new node for the beginning

first = new Node();

set the instance variables in the new node

first.item = "not";
first.next = oldfirst;
Stack pop: linked-list implementation

inner class

```java
public class Node {
    String item;
    Node next;
    ...
}
```

save item to return

```java
String item = first.item;
```

delete first node

```java
first = first.next;
```

return saved item

```java
return item;
```
Linked list stack implementation performance

• Every operation takes constant time
• No array resizing cost
Stacks and queues

Fundamental data types.
- Value: collection of objects.
- Operations: insert, remove, iterate, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

Stack. Examine the item most recently added.  LIFO = "last in first out"
Queue. Examine the item least recently added.  FIFO = "first in first out"
Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list; insert/remove from opposite ends.
Queue enqueue: linked-list implementation

inner class

public class Node {
    String item;
    Node next;
    ...
}

save a link to the last node

Node oldlast = last;

create a new node for the end

Node last = new Node();
last.item = "not";
last.next = null;

link the new node to the end of the list

oldlast.next = last;
Queue dequeue: linked-list implementation

inner class

public class Node
{
    String item;
    Node next;
    ...
}

save item to return
String item = first.item;

delete first node
first = first.next;

return saved item
return item;

Remark. Identical code to linked-list stack pop().
public class LinkedQueueOfStrings
{
    private Node first, last;

    private class Node
    { /* same as in StackOfStrings */ }

    public boolean isEmpty()
    { return first == null; }

    public void enqueue(String item)
    {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else oldlast.next = last;
    }

    public String dequeue()
    {
        String item = first.item;
        first = first.next;
        if (isEmpty()) last = null;
        return item;
    }
}
Roadmap

• Linked List
• Re-implementing Stacks and Queues using Linked List
• Interface and Iterators
• Java collections classes
• Hw3 discussion
Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.

Java solution. Make stack implement the `Iterable` interface.
Interface

- An interface in Java is an abstract type that specifies an interface that classes must implement.
- An interface only contains method signature and constant declarations, no method definitions.
- A class that implements an interface must implement all of the methods described in the interface, or be an abstract class.
- Simulate multiple inheritance.
  - All classes in Java must have exactly one super class (except Object); multiple inheritance of classes is not allowed.
  - A class may implement any number of interfaces.
Iterators

Q. What is an `Iterable`?
A. Has a method that returns an `Iterator`.

Q. What is an `Iterator`?
A. Has methods `hasNext()` and `next()`.

Q. Why make data structures `Iterable`?
A. Java supports elegant client code.

"foreach" statement

```java
for (String s : stack)
    StdOut.println(s);
```

equivalent code

```java
Iterator<String> i = stack.iterator();
while (i.hasNext())
    {
        String s = i.next();
        StdOut.println(s);
    }
```
Stack iterator: array implementation

```java
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove() { /* not supported */ }
        public Item next() { return s[--i]; }
    }
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>null</td>
</tr>
<tr>
<td>7</td>
<td>null</td>
</tr>
<tr>
<td>8</td>
<td>null</td>
</tr>
<tr>
<td>9</td>
<td>null</td>
</tr>
</tbody>
</table>
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() { return current != null; }
        public void remove() { /* not supported */ }
        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
Iterators

Q. What is an **Iterable**?
A. Has a method that returns an **Iterator**.

Q. What is an **Iterator**?
A. Has methods `hasNext()` and `next()`.

Q. Why make data structures **Iterable**?
A. Java supports elegant client code.

---

```
public interface Iterable<Item> {
    Iterator<Item> iterator();
}
```

```
public interface Iterator<Item> {
    boolean hasNext();
    Item next();
    void remove();  // optional; use at your own risk
}
```

---

```
for (String s : stack)  
    StdOut.println(s);
```

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```
Roadmap

• Linked List
• Re-implementing Stacks and Queues using Linked List
• Interface and Iterators
• Java collections classes
• Hw3 discussion
List interface. `java.util.List` is API for ordered collection of items.

```java
public interface List<Item> implements Iterable<Item>
```

- `List()`  
  - create an empty list
- `boolean isEmpty()`  
  - is the list empty?
- `int size()`  
  - number of items
- `void add(Item item)`  
  - append item to the end
- `Item get(int index)`  
  - return item at given index
- `Item remove(int index)`  
  - return and delete item at given index
- `boolean contains(Item item)`  
  - does the list contain the given item?
- `Iterator<Item> iterator()`  
  - iterator over all items in the list
  ...

Implementations. `java.util.ArrayList` uses resizing array;  
`java.util.LinkedList` uses linked list.
Java collections library

java.util.Stack.

• Supports `push()`, `pop()`, `size()`, `isEmpty()`, and iteration.
• Also implements java.util.List interface from previous slide, including, `get()`, `remove()`, and `contains()`.
Java Queues and Deques

• `java.util.Queue` is an interface
  – `add()`
  – `remove()`

• `java.util.Deque` is an interface that is a sub interface of Queue
  – Supports insertion and removal at both ends
  – `addFirst()`
  – `removeFirst()`: equivalent to `remove()`
  – `addLast()`: equivalent to `add()`
  – `removeLast()`
Java ArrayDeque class

- **java.util.ArrayDeque** implements Deque interface and supports both stack and queue operations
- **Queue methods**
  - `add()`, `addLast()`
  - `remove()`, `removeFirst()`
  - `peek()`, `peekFirst()`
- **Stack methods**
  - `push()`, `addFirst()`
  - `pop()`, `removeFirst()`
  - `peek()`, `peekFirst()`
Java ArrayDeque Example

```java
import java.util.ArrayDeque;
import java.util.Iterator;

public class DequeTest {
    public static void main(String[] args) {
        ArrayDeque<Integer> s = new ArrayDeque<Integer>();

        s.push(2);
        s.push(4);
        s.push(6);

        System.out.println(s);
        System.out.println(s.pop());

        // use iterator to access inner elements
        Iterator<Integer> iter = s.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```
Roadmap

• Linked List
• Re-implementing Stacks and Queues using Linked List
• Interface and Iterators
• Java collections classes
• Hw3 discussion
Maze Traversal
Hw 3: Maze Traversal

- A maze is a square space represented using two-dimensional array
  - Each cell has value 0 (passage) or 1 (internal wall).
  - Entrance at upper left corner, an exit at lower right corner
- Find a path from entrance to exit
Example Output

Path: ( [0][0], [1][0], [1][1], [1][2], [2][2],
   [3][2], [3][3], [4][3], [5][3], [6][3],
   [6][4], [6][5], [5][5], [4][5], [3][5],
   [2][5], [2][6], [1][6], [0][6], [0][7],
   [0][8], [1][8], [2][8], [3][8], [4][8],
   [5][8], [5][7], [6][7], [7][7], [8][7],
   [8][8], [8][9], [9][9])

ENTER --> X 1 1 1 0 0 X---X---X 0
          |      |
  x---x---x 1 0 0 X 1 x 0
          |      |
  0 1 x 1 1 X---x 1 X 0
          |      |
  0 1 X---x 1 x 1 1 X 0
          |      |
  0 1 0 x 1 x 1 1 X 0
          |      |
  1 1 1 x 1 x 1 X---x 0
          |      |
  0 0 1 X---x---x 1 x 1 1
          |      |
  0 0 1 0 0 0 0 1 X 1 1
          |      |
  0 1 1 0 1 0 1 X-- X-- X
          |      |
  0 0 0 0 1 0 1 1 0 X --> EXIT

2/26/2013
Path Finding

• Basic idea:
  – Starting from entrance, systematically search for all possible paths in the maze
  – If one path reach the exit, stop the search and construct the path

• Path search:
  – Depth-first search
    • At each choice point, follow one path until there is no further choice or exit reached
    • Back trace to previous choice point
  – Breadth-first search
    • Split at every choice point
Path Finding Algorithm

Create a search list (stack or queue) for positions yet to explore
Add the entrance position, (0,0), to the search list
while the list is not empty  {
    remove the next position from the search list
    if it is the exit position, [n-1, n-1]
        a path is found, construct the path and return the path
    else
        mark the position as visited, add all valid up, down, left, or right neighbor positions to the search list
}
if the list is empty and the method has not returned, there is no path
Path Construction

• Use linked lists to keep track of explored paths
  – Each position has a parent or predecessor reference
  – When adding the neighbor positions to the search list, the current position is the parent

• Reversely traverse the linked list to construct the path
Example

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>