CS 171: Introduction to Computer Science II

Linked List

Li Xiong
What we have learned so far

• Basic data structure
  – Arrays

• Abstract data types
  – Stacks
    • Last-In-First-Out (LIFO)
    • Operations: push, pop
  – Queues
    • First-In-First-Out (FIFO)
    • Operations: enqueue, dequeue
Arrays

- **Arrays** have certain disadvantages:
  - Search is slow in unordered array
  - Insertion is slow in ordered array
  - Deletion is slow in both cases
  - Both insertion into ordered array and deletion require moving elements
  - Difficult to support dynamic size
A Different Data Structure

• **Linked List**
  – A general-purpose storage structure
  – Can replace arrays in many cases
  – Insertion and deletion are fast
  – Truly supports dynamic size
Linked list

- Linked list concept
- Linked list operations
- Different versions of linked list
- Re-implementing stacks and queues using linked list
Linked List

• A Linked List is a sequence of nodes chained together.

• Each node, element, or link contains a data item, and a reference to next node.
Node

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

• This is called \textit{self-referential}.
  \quad A class containing a reference to itself.
Self-Referential

• In Java, an object type variable stores a reference, a **pointer**, to an object, it does **not** contain the object.

• A reference is a memory address to the actual object.

• All references are of the same size: (regardless of what they point to)
  – 4 bytes in a 32-bit program
  – 8 bytes in a 64-bit program
Object vs. Object Reference

object

object reference

R1
1600 Pennsylvania Ave
Washington, D.C.
Linked List

A chain of objects

First object in the chain

What is actually stored in memory
Difference with Arrays

• The major difference of Linked List with Array is that **Array stores elements continuously in memory** while **linked list does not**.
• Linked list supports dynamic size
• There is no simple indexing in Linked List.
• Linked List incurs some memory overhead, because of 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

```java
Node first = new Node();
first.item = "to";

Node second = new Node();
second.item = "be";
first.next = second;

Node third = new Node();
third.item = "or";
second.next = third;
```
Linked List Operations

• **Insert**
  – Inserts an element at the front.
  – It’s possible to insert at the end as well.

• **Find (search)**
  – Find an element with a specific key.

• **Delete**
  – Delete an element at the front
  – Delete an element with a specific key
Insert at the beginning

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

• Example: insert “not” at the beginning
Remove from the beginning

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

• Example: remove “to” at the beginning
• Set the root to the next node in the list
Insert at the end

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

• Example: insert "not" at the end
• Maintain a link to the last node in the list

```
save a link to the last node
Node oldlast = last;

create a new node for the end
Node last = new Node();
last.item = "not";

link the new node to the end of the list
oldlast.next = last;
```
Double-ended Linked List

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

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

• Traversing an array

    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

• Traversing a linked list

```java
for (Node x = first; x != null; x = x.next) {
    if (x.item.equals("be")) return x;
}
```
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 in the list, then remove it

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

• Example: remove “be” from the linked list
• Search the item, then remove it
• Need to keep the reference to the previous element as well as 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;
Remove a given item

• Example: remove “be” from the linked list
• Search the item, then remove it
• Need to keep the reference to the *previous* element as well as current element.
• Need to consider the case when current is first

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
    previous.next = current.next;
Doubly Linked List

- A doubly linked list has **bidirectional** references, one pointing to the **next** link, and one pointing to the **previous** link.
Doubly Linked List

- **Pros**: flexibility
- **Cons**: complexity, memory consumption
- For clarity, we often call the ordinary linked list explicitly as **singly linked list**.
- **Do not confuse** Doubly Linked List with Double-ended List!
Linked List vs. Arrays

• Both are general purpose data structures
• Linked list support faster delete
• Linked list truly support dynamic size (compares favorably even with expandable arrays)
• Linked list does occur memory overhead
• Linked list does not support index based access
• (Singly) linked list
• Double ended linked list
• Doubly linked list
Halloween Costume – Linked List
Doubly Linked List
Circularly Linked List
Binary Tree
Null Pointer
Linked list

• Linked list concept
• Linked list operations
• Different versions of linked list
• Re-implementing stacks and queues using linked list
Using Linked List

• Linked List is **interchangeable** with array in many cases, we can re-implement Stacks and Queues using Linked List.

• Implementing Stack using Linked List
  – The underlying storage using a linked list instead of an array
  – The stack interface methods are exactly the same with before.
Maintain pointer to first node in a linked list; insert/remove from front.

<table>
<thead>
<tr>
<th>StdIn</th>
<th>StdOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>to</td>
</tr>
<tr>
<td>first</td>
<td>null</td>
</tr>
<tr>
<td>be</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>to</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>to</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>be</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>be</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>be</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
</tr>
</tbody>
</table>

insert at front of linked list
remove from front of linked list
Stack pop: 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;
Stack push: linked-list implementation

inner class

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

**save a link to the list**

```java
Node oldfirst = first;
```

**create a new node for the beginning**

```java
first = new Node();
```

**set the instance variables in the new node**

```java
first.item = "not";
first.next = oldfirst;
```
public class LinkedStackOfStrings {
    private Node first = null;

    private class Node {
        String item;
        Node next;
    }

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

    public void push(String item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop() {
        String item = first.item;
        first = first.next;
        return item;
    }
}
Generic stack: linked-list implementation

public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

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

    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}

public class Stack<Item>
{
    private Node first = null;

    private class Node
    {
        Item item;
        Node next;
    }

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

    public void push(Item item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
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;
        }
    }
}
Linked list stack implementation performance

• Every operation takes constant time
• No array resizing cost
Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list; insert/remove from opposite ends.
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().
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;
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;
    }
}