CS 187: Programming with Data Structures (Spring 2010)

Lecture 10: Linked List

Rui Wang
Summary of Stacks and Queues

• Stack:
  – LIFO (Last-in-first-out)
  – Modify via two operations: push, pop
  – Both at the top of the stack

• Queue:
  – FIFO (First-in-first-out)
  – Modify via two operations: insert, remove
  – Insert to the rear, remove from the front
  – Circular queue
Summary of Stacks and Queues

• Priority Queue:
  – Elements are ordered by priority (importance)
  – Remove always deletes the element with the highest priority.

• Priority queue follows the FIFO rule. → True or False?
A Different Data Structure

• **Arrays** have certain disadvantages:
  – Search is slow in unordered array
  – Insertion is slow in ordered array
  – Deletion is slow in both cases
  – 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
Today

• Linked List
  – Links
  – Simple Linked List
  – Search, Insert, Delete
  – Implement Stacks and Queues using Linked list
Links

- **Link** is a fundamental element in a linked list.
- Contains **data**, and **reference** to another link.
- We define it as a class.
Links

• A Linked List contains many **Link objects** chained together.
Links

• Class definition:

```java
class Link {
    public int iData;
    public double dData;
    ......
    public Link next;
}
```

Data

Reference to the next link object.
Links

• The data portion can be wrapped in a class.

```java
class Link {
    public Record data;  // Data
    public Link next;    // Reference to the next link object.
}
```

• This is called **self-referential**.
  – A class containing a reference to itself.
Self-Referential

• What is going on here? Is this a recursive definition?

• How does the compiler know how much space to allocate for the next variable?
Self-Referential

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

• A reference merely stores 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
Self-Referential

• Consider:

```java
Link ref;
ref = new Link(); // ref.next = ?
ref.next = new Link();
```

• FYI, in C/C++, reference is defined in a different way.
Difference with Arrays

• What are the main differences of Linked List vs. Array?
Difference with Arrays

• The major difference of Linked List with Array is that **Array stores elements continuously in memory** while linked list does **not**.

• There is no simple indexing in Linked List.

• Linked List incurs some memory overhead, because of the need to store references.
Difference with Arrays

• The only way to find an element in Linked List is to follow the chain of elements.
  – Think about an example where you need to reach a person who is a friend of friend of friend ... yours.

• This is called **relational** (vs. **positional**).

• Workshop applet.
Linked List

• **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**
  – Note how the links are changes
Linked List

- **The Link class**

```java
class Link {
    public int iData;        // data item
    public double dData;     // data item
    public Link next;        // next link in list

    // constructor
    public Link(int id, double dd) {  // constructor
        iData = id;  // initialize data
        dData = dd;  // ('next' is automatically
                      // set to null)
    }

    // display ourself
    public void displayLink() {
        System.out.print("{" + iData + ", " + dData + "} ");
    }
}  // end class Link
```
Linked List

• The **Link** class

```java
class Link {
    public int iData;       // data item
    public double dData;    // data item
    public Link next;       // next link in list

    public Link(int id, double dd) // constructor
    {
        iData = id;           // initialize data
        dData = dd;           // ('next' is automatically
        // set to null)
    }

    public void displayLink() // display ourself
    {
        System.out.print("{" + iData + ", " + dData + "} ");
    }
} // end class Link
```
Linked List

• The **LinkedList** class

```java
class LinkList {
    private Link first; // ref to first link on list

    public void LinkList() { // constructor
        first = null; // no items on list yet
    }

    public boolean isEmpty() { // true if list is empty
        return (first==null);
    }
}
```
Linked List

- The **LinkedList** class

```java
class LinkedList
{
    private Link first; // ref to first link on list

    public void LinkList() // constructor
    {
        first = null; // no items on list yet
    }

    public boolean isEmpty() // true if list is empty
    {
        return (first==null);
    }
}
```

Also called **root**
Linked List

• The `insertFirst()` method
Linked List

- The `insertFirst()` method
Linked List

• The **insertFirst()** method

```java
public void insertFirst(int id, double dd)
{
    // make new link
    Link newLink = new Link(id, dd);
    newLink.next = first;  // newLink --> old first
    first = newLink;       // first --> newLink
}
```

• Step by step demonstration
Linked List

- The `deleteFirst()` method
Linked List

- The `deleteFirst()` method
Linked List

• The **deleteFirst()** method

```java
public Link deleteFirst() { // delete first item
    Link temp = first; // (assumes list not empty)
    first = first.next; // save reference to link
    return temp; // delete it: first-->old next
    // return deleted link
}
```

• Step by step demonstration

• Java’s **garbage collection**
Linked List

• The `displayList()` method
  - Start from the first, and follow the chain of elements using the `next` variable.

```java
public void displayList()
{
    System.out.print("List (first-->last): ");
    Link current = first;   // start at beginning of list
    while(current != null)   // until end of list,
    {
        current.displayLink();   // print data
        current = current.next;   // move to next link
    }
    System.out.println(" ");
}
```
Find (Search)

• Find an element with a given key value
  – Follow the chain, and return when key found.
• An improved version over the textbook code:

```java
public Link find(int key) {
    Link current = first;
    while (current != null && current.idata != key)
        current = current.next;
    return current;
}
```
Find (Search)

• Find an element with a given key value
  – Follow the chain, and return when key found.

• An improved version over the textbook code:

```java
public Link find(int key) {
    Link current = first;
    while (current != null && current.idata != key)
        current = current.next;
    return current;
}
```

Note the order of the two conditions
Delete

• Delete an element with a given key.
  – Similar to search, but think about how to delete?
  – What happens when the element to delete is the first vs. not the first?
Delete

• Delete an element with a given key.
  – Similar to search, but think about how to delete?
  – What do we need to consider for delete?

• In order to delete, we need to keep the reference to the previous element as well as current element.

• Must consider the case where the element to delete is the first element.
Linked List Efficiency

- Insertion and deletion at the front?
- Find, delete, and insert with a given key?
- How do these compare with an Array?
Linked List Efficiency

• Insertion and deletion at the front? 
  \(O(1)\)

• Find, delete, and insert with a given key? 
  \(O(N)\)

• How do these compare with an Array?
Linked List Efficiency

• Insertion and deletion at the front?
  \( O(1) \)
• Find, delete, and insert with a given key?
  \( O(N) \)
• How do these compare with an Array?
  – Same cost in Big-O notation, but is potentially faster because nothing needs to be moved.
Linked List Efficiency

• Naturally support dynamic list size.
• This compares favorably even with expandable arrays. Why?
• What are the downsides?
Using Linked List

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

• Example of Stack
  – Note that the underlying storage using a linked list instead of an array is hidden.
  – The stack interface methods are exactly the same with before.
Using Linked List

• Abstraction
  Description of a data structure without regard to its detailed implementation.
  1. data and implementation are hidden
  2. common operations are public
Announcements

• Quiz 3 is due tomorrow.
• Assignment 4 is due on Thursday.

• Code in the textbook can be downloaded online. The link is provided on the class webpage.
• Link to the workshop applets is also on the class webpage.