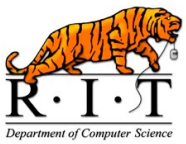


Linked lists

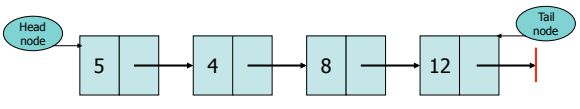
One of the classic "linear structures"

What are linked lists?

- Yet another Abstract Data Type
- Provides another method for providing space-efficient storage of data

What do they look like?

- Linked lists are made up of a series of elements that are arranged one after another, with each one connected to the next by a "link".
- A node is an element of the list that contains a value and the connection to the next element in the list (or null, if there is no next connection)
- The first node in the list is called the "head" of the list
- The last node in the list is called the "tail"



Things that use "nodes"

- Single linked lists
  - Nodes contain a value, and a link to the next node in the list
- Double linked lists
  - Nodes contain a value, and a link to the next and previous nodes in the list
- Trees and Graphs
  - Nodes contain a value, and may links to *multiple* child nodes (unlike in a list)
  - (We'll see these later....)

A starter linked list node class

```
/* See
   http://www.cs.rit.edu/~mjh/classes/cs3/samples/linkedLists/skeleton/
   LinkedObjectNode.java

   This class uses Object references to store data.
*/

public class LinkedObjectNode {
   private Object data;           // The data in this linked node
   private LinkedObjectNode next; // The next node in the list

   // contains constructors/methods
   public LinkedObjectNode() { ... }
   public LinkedObjectNode( Object data ) { ... }
   public LinkedObjectNode( Object data, LinkedObjectNode next ) { ... }
   public void setData( Object data ) { ... }
   public Object getData() { ... }
   public void setNext(LinkedObjectNode next ) { ... }
   public LinkedObjectNode  getNext() { ... }
}
```

Another linked list node class

```
/* See
   http://www.cs.rit.edu/~mjh/classes/cs3/samples/linkedLists/skeleton/
   LinkedNode.java

   Unlike the preceding example, this class uses the "generic" mechanisms
   introduced in Java 5 to store data.
*/

public class LinkedNode<E> {
   private E data;             // The data in this linked node
   private LinkedNode<E> next; // The next node in the list

   // contains constructors/methods
   public LinkedNode() { ... }
   public LinkedNode( E data ) { ... }
   public LinkedNode( E data, LinkedNode<E> next ) { ... }
   public void setData( E data ) { ... }
   public E getData() { ... }
   public void setNext( LinkedNode<E> next ) { ... }
   public LinkedNode<E> getNext() { ... }
}
```

Implementing LinkedNode

- [done in class, from the starter code using "generics"]

An important distinction

- What's the difference between a collection of (connected) linked node objects and a linked list?
- Answer: What the user of the data structure(s) needs to know/do to use it.
  - Linked node objects are an implementation detail: the user shouldn't be manipulating them directly.
  - A "linked list" class, which hides details from the user (like dealing with nodes) is called for here.

Designing a LinkedList class

- What needs to be in there?
  - What can the users *do*?
  - What do the users *see*?
  - What happens to the data while this goes on?

Implementing LinkedList

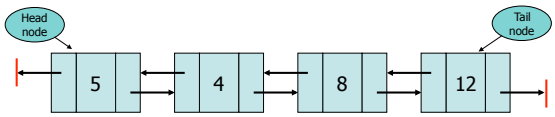
- [done in class; will be posted to the web site afterward]

Some linked list variants

- Double-linked lists
- Circular lists
- Sorted lists

Doubly-linked lists

- In a double-linked list, each node keeps track of the node before and after it in the sequence

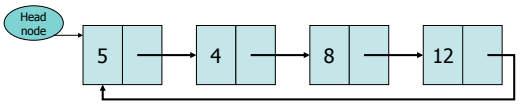


Doubly-linked list pros/cons

- Pros include:
  - You can easily traverse the list in either direction
- Cons include:
  - Twice as much "overhead" per node, in the form of an extra reference
  - Adding/removing nodes is a little more complicated, due to the additional references

Circular lists

- In a circular list, the "last" element in the list has the "first" element as its next node



Circular list pros/cons

- Pros include:
  - There can be fewer special cases to be dealt with (e.g., removing the first node), depending on your implementation
  - The "head" of the list can be changed as needed (e.g., if you're rotating through a set of data periodically)
- Cons include:
  - Iterating over the list means we need to remember where we started (since there's no null to signal the end)

Sorted lists

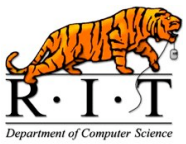
- The internal structure of a sorted list can be anything (single, double, circular, etc.)
- The only rule is one of placement

When to use what for storage (part 1)

- Arrays are good when:
  - You want to change a specific element in a specific location *quickly* (arrays are better at random access)
  - You aren't frequently changing the storage capacity (and especially when you know ahead of time how much space you'll need)
- Singly linked lists are good when:
  - You need to remove/add elements after the current node
  - You need frequent capacity changes in the data structure
- Doubly linked lists are good when:
  - You need to remove/add elements before the current node
  - You need frequent capacity changes in the data structure

Building on Linked Lists

- Some other "classical" data structures can be built up on top of linked lists
- Examples:
  - stacks
  - queues
  - priority queues



Some notes for later on....

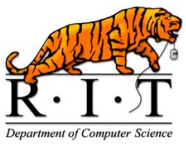
(...when we've looked at algorithmic complexity)

Some design constraints

- Our customers have spoken:
  - New data should go in at the end of the list
  - Adding data should be a constant-time operation (i.e., O(1))
  - Finding the size of the list should be O(1)
  - Removing data should be supported, either by value or by position in the list
  - Retrieving the "N^{th}" thing in the list should be O(1)
  - The list should allow collisions on values
  - The users want to be able to find out if data is in the list or not
- How well does our design hold up?

Some new demands

- Having seen the LinkedList class, our customers have some more requests:
  - They want to be able to add data:
    - after a specified value that's in the list
    - at a specified position (index)
  - This should be O(1) time, like the existing add().
- What needs to change in our design?



Choosing your ADTs

or, when to use what for storage when you have some other options

When to use what for storage

- Arrays are good when:
  - You want to change a specific element in a specific location *quickly* (arrays are better at random access)
  - You aren't frequently changing the storage capacity (and especially when you know ahead of time how much space you'll need)
- Singly linked lists are good when:
  - You need to remove/add elements after the current node
  - You need frequent capacity changes in the data structure
- Doubly linked lists are good when:
  - You need to remove/add elements before the current node
  - You need frequent capacity changes in the data structure

When to use what for storage

- Binary trees are good when:
  - You need frequent capacity changes in the data structure
  - High-speed manipulation (especially searches) is important
  - You're concerned with providing sorted access to data
  - You're unwilling to risk the "hiccups" involved with rehashing
- Hash tables are good when:
  - You need frequent capacity changes in the data structure
  - High-speed manipulation (especially searches) is important
  - You're concerned with key values, rather than positional data
  - You can deal with the risk of rehashing when the table changes