CSE 1030

Yves Lespérance

Lecture Notes

Week 12 — Linked Data Structures

Recommended Readings:
Savitch Ch. 15

Motivation: Issues Using Arrays

Arrays store their elements contiguously in one memory block and the location of each element is known in advance.

Any element can be accessed in constant time.

But, inserting an element requires shifting subsequent elements, which is $O(n)$ in general.

When array gets full, insertion requires getting a larger array and copying content of old array into the new one.

Linked Data Structures

Linked data structures provide an alternative.

They store each data item in a small packet called a *node*.

Nodes are linked to other nodes by storing references to the related nodes in the source node.

Storage is allocated one piece at a time, as needed.

Linked Lists

A list of items can be represented by a linked list data structure.

E.g. a shopping list as in Savitch's Display 15.1.

A simple linked list can be represented by an object with an attribute `head` that stores a reference to the first node in the list.

Each node in the list contains a data item and a link/reference to the next node in the list. Nodes are instances of an inner class:

```
private class Node<T>
{
   private T item;
   private Node<t> link;

   // constructors omitted

}
```

The link of the last node in the list is set to `null` to indicate the end of the list.

To represent an empty list, we set the `head` attribute to `null`.

See `LinkedList2` code in Savitch's Display 15.7.

Adding a New Item at the Front of a Linked List

To add a new item at the front of a linked list you create a new node, set its data, make its link point to the current head node, and make the `head` of the list point to it.

```
Node n = new Node();
n.item = data;
n.link = head;
head = n;
```

Note that this works even if the list is empty.

To Delete the Item at the Front of a Linked List

To delete the item at the front of a linked list you simply make the `head` point to it's successor:

```
head = head.link;
```

The list must not be empty!

Traversing a Linked List

To traverse a linked list, you set a reference to the head, process a node, and then advance the reference to the next node, and so on, until you reach the end of the list (indicated by `null`).

```
Node position = head;
while (position != null)
{
   <PROCESS NODE AT position>
   position = position.link;  %advance to next node
}
```

Iterators for Linked Lists

To allow the client to traverse and manipulate a linked list at will, we can define a linked list iterator class.

The iterator has an attibute `position` that points to the current node in the linked list. Initially, `position` points to the head node.

One advances the iterator by calling the method `next()`, which does `position = position.link`.

This is only allowed if there is a next node. One can check this by calling the boolean method `hasNext()`.

To make deletion and insertion easier, it is convenient to keep a reference to the predecessor of the current node in an attribute `previous`. In `next()`, we do `previous = position` before advancing `position`.

See Savitch's Display 15.17.

Deleting the Node at the Current Position

To delete the node at the current position you simply make the current node's predecessor point to its successor, i.e.

```
previous.link = position.link;
```

The successor of the current node becomes the new current node.

Deletion of the head node must be handled as a special case, as we have seen earlier.

Inserting a New Item at the Current Position

To add a new item at the current position, you create a new node, set its data, make its link point to the current node, and make the curent node's predecessor point to it:

```
Node n = new Node();
n.item = data;
n.link = position;
previous.link = n;
```

Insertion at the head must be handled as a special case, as seen earlier.

Making a Copy

We traverse the existing list, making a copy of each node, and linking them up.

```
public LinkedList(LinkedList otherList)
{
   if (otherList == null)
      throw new NullPointerException( );
   if (otherList.head == null)
      head = null;
   else
      head = copyOf(otherList.head);
}
```

Making a Shallow Copy

```
private Node copyOf(Node otherHead)
{
   Node position = otherHead;//moves down other’s list.
   Node newHead; //will point to head of the copy list.
   Node end = null; //positioned at end of new growing list.

   newHead = new Node(position.data, null); // copy 1st node
   end = newHead;
   position = position.link;
   while (position != null)
   {//copy node at position to end of new list.
       end.link = new Node(position.data, null);
       end = end.link;
       position = position.link;
   }
   return newHead;
}
```

Making a Deep Copy

```
private Node copyOf(Node otherHead)
{
   Node position = otherHead;//moves down other’s list.
   Node newHead; //will point to head of the copy list.
   Node end = null; //positioned at end of new growing list.

   newHead = new Node(new DataClass(position.data), null);
   end = newHead;   //use appropriate data class e.g. Student, Date
   position = position.link;
   while (position != null)
   {//copy node at position to end of new list.
       end.link = new Node(new DataClass(position.data), null);
       end = end.link;
       position = position.link;
   }
   return newHead;
}
```

Generic Linked List

Can also define a generic `LinkedList<T>` where the data item type is a parameter `T`. See Savitch Display 15.8.

Variations on Linked Lists

There are many!

A *doubly linked list* stores links to both the successor and predecessor nodes in each node. This means that they can be traversed forwards and backwards. Insertion and deletion also become easier.

A *Stack* is a list where one can only insert (push) and remove (pop) at the front. This can be implemented using a linked list with restricted operations. It can also be implemented using an array.

A *Queue* is a list where one can only insert at the back and remove at the front. This can be implemented using a linked list that keeps both head and tail references. An array implementation is also possible.

Trees

A tree is a data structure made of nodes.

Each node stores a data item.

Each node has zero or more children nodes.

Each node has exactly one parent, except the root node, which has no parent.

Trees have a recursive structure. Each child of a node is also a tree. Tree algorithms are naturally defined by recursion.

Binary Trees

A binary tree is a tree where each node has at most two children.

Three common ways of traversing a binary tree:

preorder: process root, then traverse left child, then traverse right child;

inorder: traverse left child, then process root, then traverse right child;

postorder: traverse left child, then traverse right child, then process root.

Binary Search Trees

A binary search treee is a special kind of binary tree where:

1. all data items in the left child are $\leq$ the data item in the root;
2. all data items in the right child are $\geq$ the root data item;
3. the left child and the right child are binary search trees, i.e., also satisfy 1. and 2.

Can be used to represent a set.

Searching such a tree can be done very efficiently if all branches are more or less the same length.

Adding an data item is also straightforward and can be done efficiently. Deleting an item is more complicated, as the tree must be rearranged while preserving the binary search tree property.

See `IntTree` example in Savitch Display 15.40.

Much more on data structures in CSE 2011!