CS201
Vectors and Dynamic Data Structures

A data structure is a software construct used to organize our data in a particular way. Some common data structures include lists, stacks, queues, and heaps. Dynamic data structures are those data structures that can grow as needed while a program is running. First we will look at a useful class provided by Java called a vector. Then we will see how to create our own data structures, in particular, how to create dynamically linked lists.

Vectors

The vector class provided by Java is essentially a way to create an array that can grow or shrink in size during the execution of a program. If you recall the section on arrays, once we declared an array to be a particular size we were not able to change that size. For example, in our inventory problem we declared the array to be of size 10. This means we have a maximum of 10 items and can never exceed that amount. This is because Java allocates a specific amount of memory to hold exactly the number of items we initially specified. We could get around that problem by creating a new array of the size we like and copy the elements we want into it, but this approach is time consuming. The vector class does the dirty work for us to give us the same effect as a dynamic array.

If vectors do the same thing as arrays but are dynamic, why not use them all the time? We could (and some people do) but there are two good reasons to use arrays over vectors:

1. Arrays are more efficient than vectors
2. The elements in a vector must be objects (which contributes to #1 for simple items)

If we wanted a vector of int’s, then we instead we would have to make a vector of class Integer, which is a wrapper class for objects of type int (or we could make our own class).

Here is how we use a vector. To access the vector code we can import the class via:

    import java.util.*;

To create a variable object of type Vector use the following:

    Vector varName = new Vector();

If you know how many items the vector will likely need, execution can be a bit faster by specifying an initial capacity. Note however that we can still change the size later during runtime if we want to. The example below initialized the vector with 50 elements:

    Vector v = new Vector(50);
Here are vector methods to manipulate data in a vector:

public void addElement(Object newElement)
    Adds a new object to the end of the vector.

    Recall that all classes are derived from class Object, therefore all classes are supported as parameters for this method.

public void setElementAt(Object newElement, int index)
    Sets an existing element at position index to newElement.
    Index starts at 0.
    An element at this index must previously exist (i.e. can’t use to add to a new position)

public Object elementAt(int index)
    Returns the object at position index
    Index starts at 0.

public void insertElementAt(Object newElement, int index)
    Inserts the newElement at position index and pushes everything else after this object down by one in the vector.

public void removeElementAt(int index)
    Deletes the element at position index and moves everything else after this object up by one in the vector.

public void removeAllElements()
    Removes everything from the vector

public int indexOf(Object target)
    Returns the index of the first element equal to target or -1 if not found.
    This method invokes the equals() method of the object, so if your object does not implement and override equals() inherited from class object, most likely this method will not work properly!

public int indexOf(Object target, int startIndex)
    Returns the index of the first element equal to target after position startIndex. Can be used to find subsequent matches in a vector.

public int size()
    Returns the number of elements placed in the vector

Here is a simple example. Let’s say we would like to make a vector of integers. To drive home the point that the vector only works with objects, we’ll make our own Integer class (but we could have used the built-in Integer class too).
import java.util.*;

// Simple integer class with just two constructors.
// A more robust version would make the int m_value private with
// accessor methods instead
public class MyIntClass
{
    public int m_value;

    public MyIntClass()
    {
        m_value = 0;
    }
    public MyIntClass(int i)
    {
        m_value = i;
    }
}

// This class illustrates common uses of the vector
public class VectorTest
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        int i;

        // First add 4 numbers to the vector
        for (i=0; i<4; i++) {
            v.addElement(new MyIntClass(i));
        }
        // Print out size of the vector, should be 4
        System.out.println("Size: " + v.size() + "\n");

        // Print vector
        System.out.println("Original vector:");
        PrintVector(v);

        // Remove the second element
        v.removeElementAt(2);
        // Print out the elements again
        System.out.println("After remove:");
        PrintVector(v);

        // Insert the number 100 at position 1
        v.insertElementAt(new MyIntClass(100), 1);
        // Print out the elements again
        System.out.println("After insert:");
        PrintVector(v);
    }
}
// This method prints out the elements in the vector
public static void PrintVector(Vector v)
{
    MyIntClass temp;
    int i;
    for (i=0; i<v.size(); i++) {
        // Notice typecast below.
        // Necessary since we are going from
        // a more general object to more specific
        temp = (MyIntClass) v.elementAt(i);
        System.out.println(temp.m_value);
    }
    System.out.println();
}

The output from this program is:

Size: 4

Original vector:
0
1
2
3

After remove:
0
1
3

After insert:
0
100
1
3

A few comments of note: Notice how we had to typecast the object back to our class of
MyIntClass after we get it out of the vector. This is because the vector returns a value of
type Object. While all classes are of type Object, not all Objects are of type MyIntClass
(in fact, most of them aren’t!). So we must use the typecast to inform Java that it is
actually correct to assign a variable of type Object to the variable of type MyIntClass.

Do you remember the Superhero Supply Store we created using arrays? The store kept
an inventory of store items and printed out the list of names with product ID’s. It was a
bit of work to make the store work so that we could keep everything in order. However,
this application is a snap using vectors. Here is a complete example of the store re-
implemented with vectors.

First here is the Product class. It is the same as before except for the equals() method:
Here is the class for Products that uses a vector of class Product. We also use various vector methods to keep things easy to put the data in sorted order and to find matches.
class Products
{
    private Vector m_vProds;

    public Products()
    {
        // Constructor just allocates new vector
        m_vProds = new Vector();
    }

    public void AddProduct(int newID, String newName)
    {
        int i;
        Product p;

        // Find new position to add element to keep it in
        // sorted order by ID
        for (i=0; i < m_vProds.size(); i++) {
            p = (Product) m_vProds.elementAt(i);
            if (p.GetID() > newID) {
                break; // insert at position i
            }
        }

        if (i==m_vProds.size()) {
            // In this case we went all the way to the end
            // so just add new element at the end
            m_vProds.addElement(new Product(newID, newName));
        } else {
            // In this case insert new element at position i
            m_vProds.insertElementAt(new Product(newID, newName), i);
        }
    }

    public void DeleteProduct(int targetID)
    {
        int i;
        // Use the indexOf method to find matching position
        // It will use the equals method which just checks
        // the ID's, so we can make a dummy Product with the target
        i = m_vProds.indexOf(new Product(targetID,""));
        if (i<0) return; // Not found
        m_vProds.removeElementAt(i); // Remove found element
    }

    public String FindProductName(int targetID)
    {
        int i;
        String s = "";
        i = m_vProds.indexOf(new Product(targetID,""));
        if (i>=0) s = ((Product) m_vProds.elementAt(i)).GetName();
        return s;
    }
}
public void PrintProducts()
{
    Product p;
    for (int i=0; i<m_vProds.size(); i++) {
        p = (Product) m_vProds.elementAt(i);
        p.PrintValues();
    }
}

// Main method to test the stuff
public static void main(String[] args)
{
    String s;
    Products p = new Products();
    p.AddProduct(55, "Spandex Suit");
    p.AddProduct(23, "Golden Lasso");
    p.AddProduct(91, "Utility Belt");
    p.AddProduct(45, "Web Fluid");

    p.PrintProducts(); // Prints list in order of ID
    s = p.FindProductName(23);
    System.out.println(s); // Prints Golden lasso
    s = p.FindProductName(24);
    System.out.println(s); // Prints blank
    p.DeleteProduct(23);
    p.PrintProducts(); // Prints list with Lasso removed
}

The output from this program is:

23 Golden Lasso
45 Web Fluid
55 Spandex Suit
91 Utility Belt
Golden Lasso

45 Web Fluid
55 Spandex Suit
91 Utility Belt

The complexity of this solution is simpler than that of the previous one using arrays. It is also more flexible since we can make our vector as large as we like, up to the size of available memory.
Linked Structures

Classes can also be linked together to form dynamic lists. Recall that a variable that has a
datatype of a class really corresponds to a pointer in memory to the address where the
class object is stored. This is called a pointer or a reference variable. We define a class
(called a node when used in a dynamic structure) that has at least two members: next (a
pointer or reference to the next node in the list) and component (the type of the items on
the list). For example, let's make a class to construct a list of integers.

```java
public class IntNode
{
    int num;
    IntNode next = null;

    public IntNode(int i) { num = i; }   // Constructor

    // We could add other methods here too
    // but we'll leave it at this to keep it simple for now
}
```

To form dynamic lists, we link variables of type IntNode together to form a chain using
the member variable called “next”. Since this variable is of type IntNode, it is really a
pointer to an IntNode object in memory. This could be a pointer to any IntNode!

Let’s start to create our list by creating two separate IntNode’s. We also make a variable
called head, which is not set to any new object yet, but is referencing null. This will be
used to remember the beginning of our list.

```java
IntNode n1 = new IntNode(51);
IntNode n2 = new IntNode(55);
IntNode head = null;
```

In memory we have something that looks like the following:

Next, let’s connect the nodes together

```java
head = n1;
n1.next = n2;
```
Here is the new picture:

We just built a linked list consisting of two elements! The end of the list is signified by the next field holding NULL. Note that even if the variables n1 and n2 go away, we can still access the nodes they were pointing to by following the links from head.

We can get a third node and have the next variable of the second node point to it. This process continues until the list is complete. Since we are constructing the list node by node, our list can grow as large as we like up to the amount of free memory. The following code fragment reads and stores integer numbers into a list until the input is –1:

```java
String s;
int i;
BufferedReader in = new BufferedReader(
    new InputStreamReader(System.in));
IntNode head = null, tail = null, temp = null, newNode = null;

System.out.println("Enter value for first node.");
s = in.readLine();
head = new IntNode(Integer.parseInt(s));
tail = head;  // Track end of the list

System.out.println("Enter values for other nodes, -1 to stop.");
s = in.readLine();
i = Integer.parseInt(s);
while (i != -1) {
    // Make new node
    newNode = new IntNode(i);
    // Add it to the end of the list
    tail.next = newNode;
    // Set tail to the new tail
    tail = newNode;
    // Get next value
    s = in.readLine();
i = Integer.parseInt(s);
}
```

This program (it is incomplete, we’ll finish it below) first inputs a number and allocates memory for head and stores the value into it. It then sets tail equal to head. tail will be used to track the end of the list while head will be used to track the beginning of the list. For example, let’s say that initially we enter the value 10:
Before entering the loop, let’s say that we enter 50 which is stored into i. First we create a new node, pointed to by newNode, and store the value of i into it:

Then we link tail.next to newNode:

Finally we update tail to point to newNode since this has become the new end of the list:

Let’s say that the next number we enter is 23. We will repeat the process, first allocated a new node pointed to by newNode, and filling in its values:
Then we link tail.next to newNode:

Then we update tail to point to the new end of the list, newNode:

The process shown above continues until the user enters –1. Note that this allows us to enter an arbitrary number of elements, up until we run out of memory! This overcomes limitations with arrays where we need to pre-allocate a certain amount of memory (that may turn out later to be too small).

Lists of dynamic variables are traversed (nodes accessed one by one) by beginning with the first node and accessing each node until the next member of a node is NULL. The following code fragment prints out the values in the list.

```java
// List is entered, now print it out
System.out.println("\nList contains: \n");
temp = head;
while (temp != null) {
    System.out.println(temp.num);
    temp = temp.next;
}
```

temp is initialized to head, the first node. If temp is NULL, the list is empty and the loop is not entered. If there is at least one node in the list, we enter the loop, print the member component of temp, and update temp to point to the next node in the list. temp is NULL when the last number has been printed, and we exit the loop.
Note that we are allocating memory for each node, but we never tell Java to de-allocate that memory when we’re done. Java has a method called garbage collection that watches when any allocated memory has nothing referencing it. Periodically Java will “collect” all of this allocated-but-unused memory and mark it as free and available for use. As mentioned before, with many programming languages the programmer is required to explicitly de-allocate any allocated memory to prevent “memory leaks” (what happens when a program grows and grows and keeps using more and more memory until eventually the program crashes).

Searching A Linked List

Let’s say that we have constructed the following linked list:

```
head
  ─── num: 32 ─── num: 31 ─── num: 55 ─── num: 14 ─── next: null
```

If we would like to search the list for some target, we merely traverse the list starting from the head:

```
temp = head;
boolean found = false;
while ((temp != null) && (!found)) {
  if (temp.num == target) {
    System.out.println("Found match, num = " + temp.num);
    found = true;
  } else {
    temp = temp.next;
  }
} if (!found) System.out.println("No match found.");
```

This code snippet merely walks through the linked list starting from the beginning until a match is found, at which point we stop by setting the found flag to true.

Deletion From A Linked List

What if we would like to delete a node from the list? We could start by modifying our code to search the list:
temp = head;
boolean found = false;
while ((temp != null) && (!found)) {
    if (temp.num == target) {
        // We want to delete this node
        found = true;
    }
    else {
        temp = temp.next;
    }
}

This is a start, but what does it mean to delete a node? Let's say we want to delete the node with value 55 from the linked list below:

```
<table>
<thead>
<tr>
<th>num: 32</th>
<th>num: 31</th>
<th>num: 55</th>
<th>num: 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>next:</td>
<td>next:</td>
<td>next:</td>
<td>next:</td>
</tr>
</tbody>
</table>
```

This would result in:

```
<table>
<thead>
<tr>
<th>num: 32</th>
<th>num: 31</th>
<th>num: 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>next:</td>
<td>next:</td>
<td>next:</td>
</tr>
</tbody>
</table>
```

To delete the node with 55 what we really want to do is set the next pointer for the previous node to the next of the node we want to delete! But with the code we had written so far, we didn't keep track of the previous node. Here is modified code that tracks the previous:

```
currentPtr = head;
previousPtr = null;
boolean found = false;
while ((temp != null) && (!found)) {
    if (temp.num == target) {
        // To delete this one, set the previous next
        // to the current next
        previousPtr.next = currentPtr.next;
        found = true;
    }
    else {
        previousPtr = currentPtr; // Set previous to current
        currentPtr = currentPtr.next; // Advance current
    }
}
```
Note that we have really created the following picture in deleting node 55:

Since the node with the value 55 was referencing the last node, we actually have two nodes referencing the same last node. But starting from the head of the list, there is no way to access the node with 55. So this node is effectively deleted. During garbage collection, Java will reclaim the memory that was allocated to this node so that it can be reused for something else.

Before we exit this section, the code we have just written has a subtle flaw. Does it work correctly if we try to delete the last node in the list? What if we try to delete the first node in the list? If there is a problem, how could we fix it?

**Optional Example: Animal Guessing Game**

Let’s do a more complex example with pointers and classes. In this example we will write a program to play a guessing game. The player thinks of an animal and the program will ask yes or no questions and try to guess what the player is thinking of.

Initially we’ll only have two types of animals that the program knows about. But each time the player is done, if the program is incorrect it will ask the player to input a new question and a new animal which will be incorporated into its knowledge base.

The plan is to start with a data structure as shown below:

```
Q: have stripes?

<table>
<thead>
<tr>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr: NULL</td>
<td>Ptr: NULL</td>
</tr>
<tr>
<td>Ans: Bear</td>
<td>Ans: Zebra</td>
</tr>
</tbody>
</table>
```

This structure contains the following knowledge:

- Ask player if the animal has stripes
- If player says Y, the null pointer means guess “Zebra”
- If player says N, the null pointer means guess “Bear”

Let’s say the player is thinking of a tiger. We’ll answer Y to “has stripes” but N to the animal being a zebra. The program will then ask the player to enter a new question to
identify the tiger. Let’s say the question is “has hooves?”. We’ll now have the knowledge structure shown below:

If we start over, the program will ask “have stripes?” and if the player types “Y” then we go directly to the copy of the node below with “have hooves?”. Depending on the answer, the program will guess Tiger or Zebra.

If we continue, each time the game is played and the program is wrong, it gains some new knowledge for later. The structure that is created is called a binary tree because it resembles an upside-down tree with the root at the top. There are at most two branches per node in the tree. A slightly larger tree with additional knowledge is shown below.
We’ll represent each node with the following class:

class Animal
{
    String question;
    String yesGuess, noGuess;
    Animal noPtr = null, yesPtr = null;
}

The “yesGuess” and “noGuess” variables will only be used if the corresponding yesPtr or noPtr variables are equal to NULL.

The rest of the program follows:

class AnimalGame
{
    // root points to the base of our tree
    private Animal root = null;
    private BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));

    public AnimalGame()
    {
        // Put in our initial knowledge
        root = new Animal();
        root.question = "Does it have black stripes?";
        root.yesGuess = "zebra";
        root.noGuess = "bear";
    }

    // Ask the user for their guess
    private boolean AskGuess(String sGuess) throws Exception
    {
        String s;
        System.out.println("Are you thinking of a " +
            sGuess + "?");
        s = in.readLine();
        if ((s.charAt(0)=='y') || (s.charAt(0)=='Y'))
            return true;
        return false;
    }

    // Adds a new node below 'ptr' to the tree of knowledge
    // cYesOrNo indicates if we should add to the Yes or No branch
    private void AddNewAnimal(Animal ptr, char cYesOrNo)
        throws Exception
    {
        String sAnimal, s;
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        Animal newAnimal = new Animal();
        if (cYesOrNo == 'y') {
            // Link new node to old yes guess
            newAnimal.yesGuess = ptr.yesGuess;
        }
    }
}
newAnimal.noGuess = ptr.yesGuess;
ptr.yesPtr = newAnimal;
}
else {
  // Link new node to old no guess
  newAnimal.yesGuess = ptr.noGuess;
  newAnimal.noGuess = ptr.noGuess;
  ptr.noPtr = newAnimal;
}
System.out.println("What is the correct answer?");
sAnimal = in.readLine();
System.out.println("Enter a question to identify your new animal.");
newAnimal.question = in.readLine();
System.out.println("Is the answer 'Y' or 'N'? ");
s = in.readLine();
if ((s.charAt(0)=='y') || (s.charAt(0)=='Y')) {
  newAnimal.yesGuess = sAnimal;
}
else {
  newAnimal.noGuess = sAnimal;
}
}

// Main method to play the game
public static void main(String[] args) throws Exception {
  AnimalGame g = new AnimalGame();
  Animal curPtr = null;
  String sGuess, sQuestion;
  boolean playgame = true, madeguess = false;
  String s;
  System.out.println("Guess the Animal!
";
  curPtr = g.root;
  while (playgame) {
    // Travel down tree until we have a guess
    madeguess = false;
    while (!madeguess) {
      // Ask the question
      System.out.println(curPtr.question);
      s = g.in.readLine();
      if (s.charAt(0)=='y') {
        // Check if we reached bottom of tree
        // if so, guess an animal
        if (curPtr.yesPtr==null) {
          if (!g.AskGuess(curPtr.yesGuess)) {
            // We were wrong, ask
            // for answer
            g.AddNewAnimal(curPtr,'y');
            madeguess = true;
          }
        }
        else curPtr = curPtr.yesPtr;
      }
      else curPtr = curPtr.noPtr;
    }
  }
}
// Check if we reached bottom of tree
// if so, guess an animal
if (curPtr.noPtr==null) {
    if (!g.AskGuess(curPtr.noGuess)) {
        // We were wrong, ask
        // for answer
        g.AddNewAnimal(curPtr,'n');
    }
    madeguess = true;
} else curPtr = curPtr.noPtr;
}

System.out.println("Play again? (y/n)");
s = g.in.readLine();
if (s.charAt(0)=='n') playgame=false;
curPtr = g.root;
}

This program is split up into several methods. AskGuess prompts the user with an animal and returns true or false if that is the animal the player is thinking of.

AddNewNode takes the current pointer, which should be pointing to a leaf in the tree. It takes another value that indicates if we are to add to the “yes” or “no” links of the tree. A new node is created and the question and one of the guesses is filled with input by the user. The other guess is copied from the parent, so we remember the old guess that was made as a default for the new guess if the new question is wrong.

The main method controls the action, looping continually until a leaf node is reached (the pointer is NULL). When this happens, the player is asked to guess and if the program is wrong, it asks the player to enter new information via AddNewNode.

This complete program is available in the Java Code Samples directory from the webpage if you would like to run it.