JAVA NOTES

DATA STRUCTURES AND ALGORITHMS

Terry Marris July 2001

9 DYNAMIC STACKS

9.1 LEARNING OUTCOMES

By the end of this lesson the student should be able to

- compare and contrast static and dynamic data structures
- understand the concept of a node
- draw diagrams to show how a stack can be implemented as linked list of nodes
- appreciate the role of inner classes
- implement the standard stack operations on a linked list of nodes

9.2 INTRODUCTION

In the last lesson we saw how to implement a stack using an array. The problem with an array is that its capacity is fixed at compile time. If the capacity is too small, applications are compromised. If the capacity is too large, memory is wasted. We need a way of allowing the capacity of a stack to change during run-time, according to the requirements of the application.

We begin our study of dynamic data structures here.

9.3 NODES

The basic unit of storage is a node. A stack node always has two fields. One field refers to data; the other field contains a link to the next node.

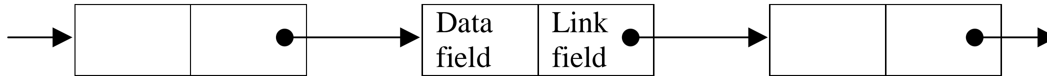


Figure 9.1 *Nodes are linked to form a data structure*

We can think of each node being located at some address in memory. Using numbers 64, 68, 72, ... to represent these (arbitrarily chosen) addresses, we can picture

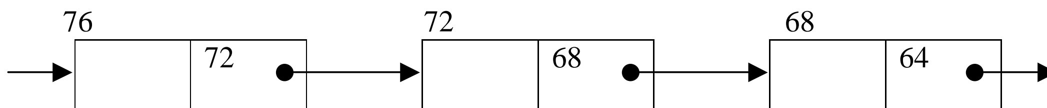
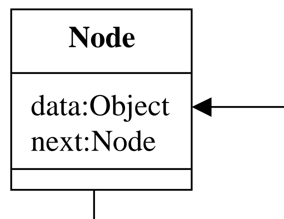


Figure 9.2 *Each link field contains the address of the next node in sequence*

Of course we can write a *Node* class.

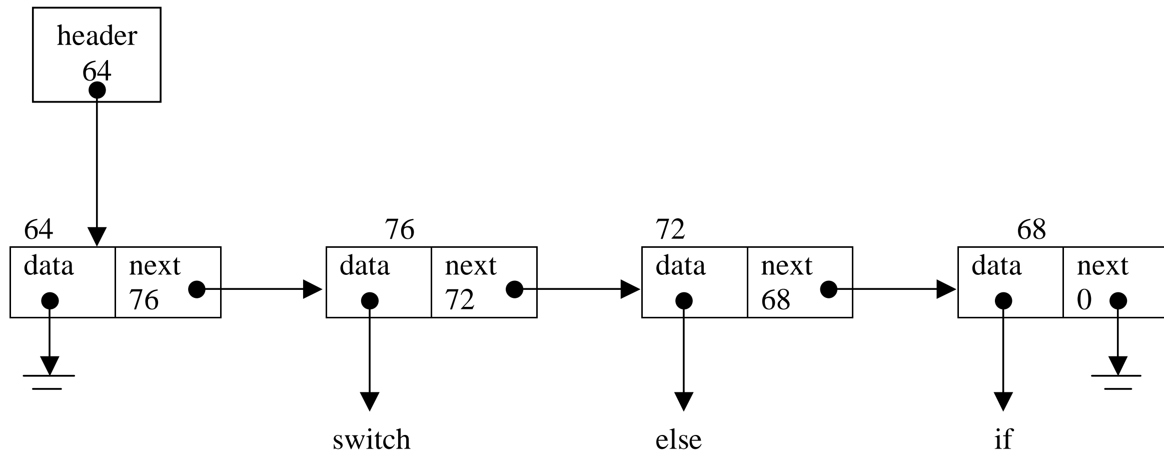
```
class Node {
  Object data;
  Node next;      // the link field
  Node(Object data, Node next)
  {
    this.data = data;
    this.next = next;
  }
}
```

The *Node* class has two fields and one constructor. *data* refers to the object being stored e.g. a *String*, an *Employee*, a *Character*. *next* refers to the next *Node* in sequence. In terms of a class diagram:



9.4 STACK

We picture a stack as a linear sequence of nodes. The following diagram represents a stack with three elements referring to the strings, *if*, *else* and *switch*.



A stack has a node referred to by *header*. This node (*64*) contains no data but does contain the address of the next node in sequence. This next node (*76*) contains a reference to the most recent data (*switch*) pushed onto the stack and so represents the top of the stack.

The node that has been in the stack for the longest (*68*) has no next node to refer to.

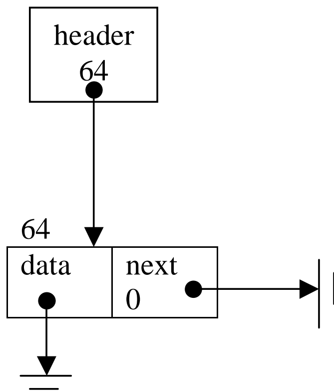
The *DynamicStack* class has two fields.

```
public class DynamicStack implements Stack {
     ...
  private Node header;
  private int size;
```

size represents the number of data objects stored in the stack.

9.5 AN EMPTY STACK

A newly created stack is an empty stack. It looks like this.



The constructor initialises *header* with a reference to a new node, and sets *size* to zero.

```
public DynamicStack()
{
  header = new Node(null, null);
  size = 0;
}
```

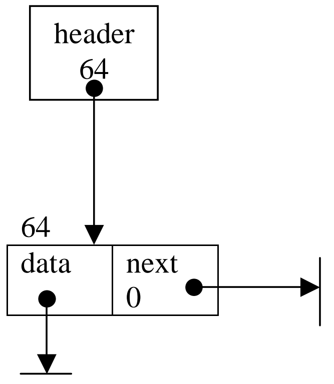
The new node has no data associated with it, and no next node to reference (*null, null*).

The *isEmpty()* method returns true if *size* is zero.

```
public boolean isEmpty()
{
  return size <= 0;
}
```

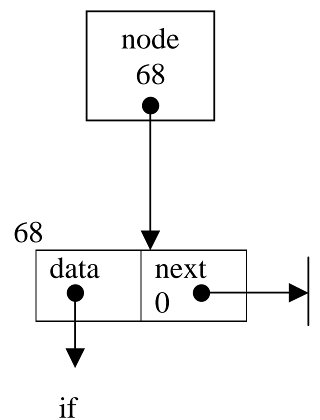
9.6 PUSH

We start with an empty stack.



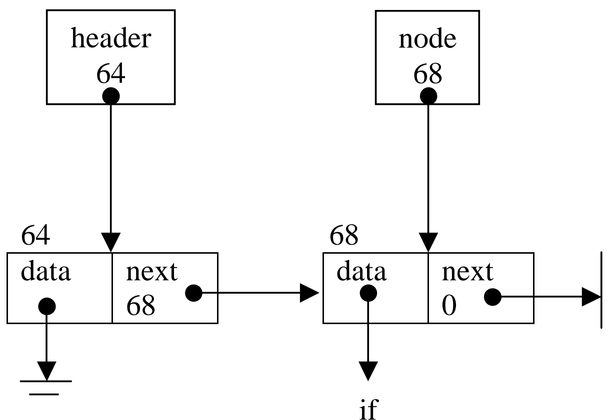
We create a new node with the given object (*if*) for its data value and the contents of *header.next* for its link field.

```
Node node = new Node(obj, header.next);
```



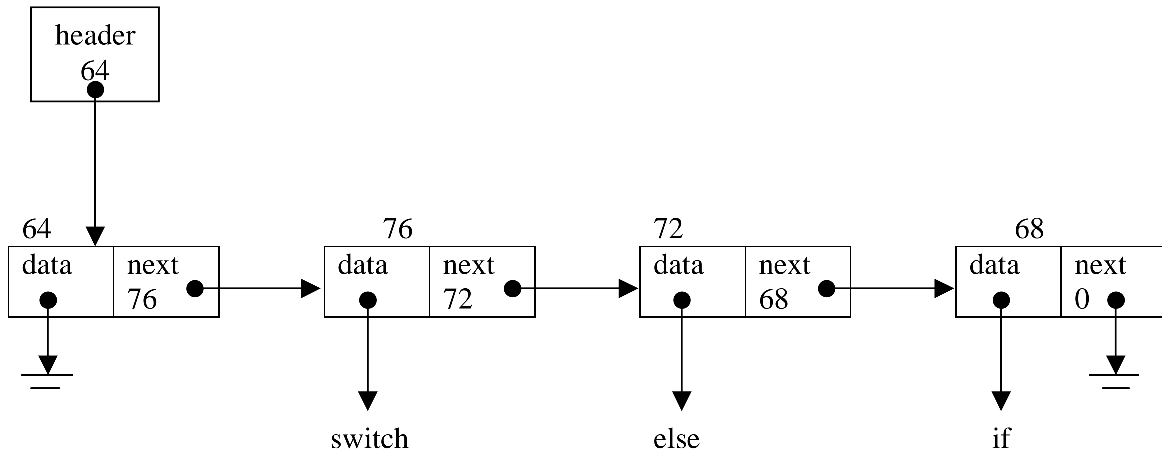
Then we update *header.next* to contain the address of the new node.

```
header.next = node;
```



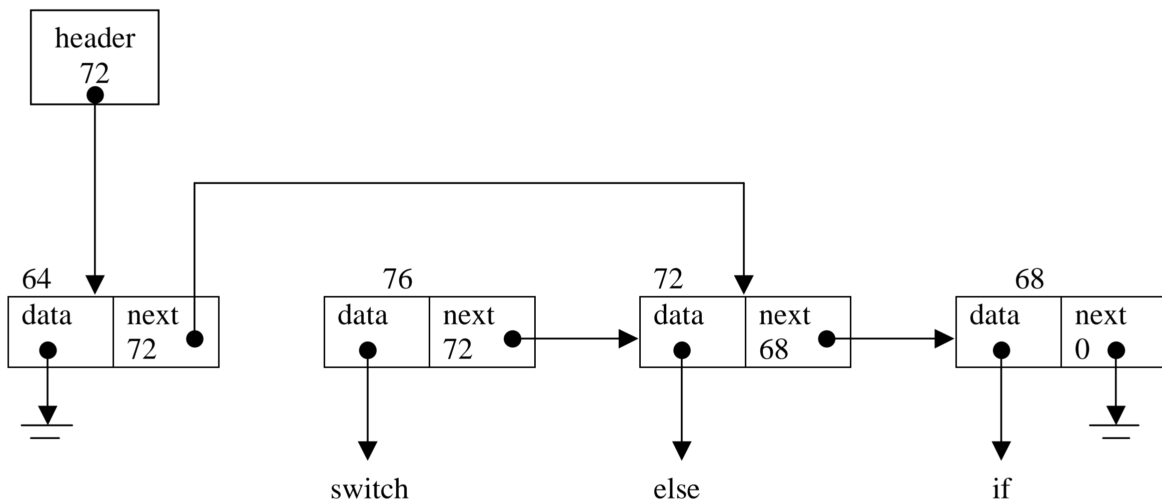
9.7 POP

We start with a stack containing three data objects.



We adjust *header.next* so that it bypasses the top node (76) in the stack.

```
header.next = header.next.next;
```



The node with the data reference to *switch* (76) is left dangling - it cannot be reached. We rely on the Java garbage collection system to dispose of it.

9.8 INNER CLASSES

We make *Node* a *private*, *static inner class* of *DynamicStack*.

```
public class DynamicStack implements Stack {
  private static class Node {
    Object data;
    Node next;

    Node(Object data, Node next)
    {
      this.data = data;
      this.next = next;
    }
  }


  private Node header;
  private int size;
     ...
```

This means that

- *Node* is a class inside the *DynamicStack* class
- *Node* helps implement *DynamicStack* behaviour
- You cannot create *Node* objects outside *DynamicStack* since *Node* is a *private* class within *DynamicStack*
- Only the *DynamicStack* class can create *Node* objects
- *Node* cannot access *DynamicStack*'s fields and methods since it is declared *static*.

Only inner classes can be *private* and *static*.

9.9 THE DYNAMIC STACK CLASS

The entire class implementation is shown below.

```
/* DynamicStack.java
   Terry Marris  25 July 2001
*/

public class DynamicStack implements Stack {
  private static class Node {
    Object data;
    Node next;

    Node(Object data, Node next)
    {
      this.data = data;
      this.next = next;
    }
  }


  private Node header;
  private int size;

  public DynamicStack()
  {
    header = new Node(null, null);
    size = 0;
  }


  public boolean isEmpty()
  {
    return size <= 0;
  }


  public String push(Object obj)
  {
    Node node = new Node(obj, header.next);
    header.next = node;
    size++;
    return "success";
  }
```

```
  public String pop()
  {
    if (isEmpty())
      return "failure - stack empty";
    header.next = header.next.next;
    size--;
    return "success";
  }


  public Object peek()
  {
    if (isEmpty())
      return null;
    else
      return header.next.data;
  }


  public int size()
  {
    return size;
  }
}
```

The test program is straightforward.

```
/* TestDynamicStack.java
   Terry Marris  25 July 2001
*/

public class TestDynamicStack {
  public static void main(String[] s)
  {
    DynamicStack stack = new DynamicStack();
    System.out.println("A new stack is an empty stack ... " +
                        stack.isEmpty());

    System.out.println(
                    "Adding five strings to the stack ...");
    System.out.println("if: " +
                        stack.push(new String("if")));
    System.out.println("else: " +
                        stack.push(new String("else")));
    System.out.println("switch: " +
                        stack.push(new String("switch")));
    System.out.println("case: " +
                        stack.push(new String("case")));
    System.out.println("default: " +
                        stack.push(new String("default")));

    System.out.println("Emptying the stack item by item ...");
    while (!stack.isEmpty()) {
      System.out.println(stack.peek());
      stack.pop();
    }
  }
}
```

Output

```
A new stack is an empty stack ... true
Adding five strings to the stack ...
if: success
else: success
switch: success
case: success
default: success
Emptying the stack item by item ...
default
case
switch
else
if
```

9.10 REVIEW

9.11 FURTHER READING

HORSTMANN & CORNELL Core Java 2 Volume 1 pp 238

9.12 EXERCISES

1 Explain the differences and similarities between static (i.e. array) and dynamic implementations of a stack.

2 Explain what a node is.

3 Explain the role of an inner class.

4(a) Draw a diagram to represent a stack as a linked list of nodes. Your stack should contain two data items.

(b) Draw a sequence of diagrams, supported by descriptive prose and snippets of Java code, to show how a new data item may be added to the stack.

(c) Draw a sequence of diagrams, supported by descriptive prose and snippets of Java code, to show an item may be removed from your stack.

5 Test all the methods defined in the *DynamicStack* class.