

# COMP 250 Assignment 4

Christopher Hundt (110220945)

November 18, 2002

## Question 1

It is possible to implement this data structure with an array  $A$  of size  $n$ , where  $n$  is the largest possible nonnegative integer allowed to be in a set, and one additional variable. This variable could easily be made the first element of the array (by increasing the size of the array by 1), but for this explanation we will assume that it lies alone. We call this additional variable  $x$ .

Both  $x$  and the elements of  $A$  are of some type that can hold any nonnegative integer up to  $n$ , as well as two additional possibilities, which we will represent in our explanation as the integers -1 and -2.

An empty set is represented by  $x$  having a value of -2 and every element of  $A$  having a value of -1.

To see if an element  $e$  is in the set, we look at the value of the element of  $A$  at index  $e$ , which we will represent as  $A[e]$ . If  $A[e]$  is not equal to -1, then the item is in the set.

To add an item  $e$  to the set, we first see if  $e$  is already in the set, as just described. If it is, we do nothing. If  $e$  is not already in the set, we set  $A[e]$  to the value of  $x$ , and we then set  $x$  to the value of  $e$ . Thus,  $x$  provides a kind of “pointer” to the most recently added item, and each item provides the same kind of pointer to the item added immediately before it.

Predictably, the removal of an item is what  $x$  is really used for. To remove an item, we first look at  $x$ . If  $x$  is -2, then the set is empty (as described above). Otherwise, we set  $A[x]$  to -1, and  $x$  to what the value of  $A[x]$  was before we changed it to -1. This is the exact opposite of the procedure that was followed to add that item to the set.

## Question 2

For the following algorithm, it is assumed that there is some manner of getting an element from the array such that `getElement(n, m)` returns the element in the  $n^{\text{th}}$  row (counting down from the top, starting with 1) and the  $m^{\text{th}}$  column (counting right from the left side, starting with 1). Also, the variable  $x$  is the number being searched for, and  $n$  is the dimension of the array.

```
currentRow = 1
currentCol = n
found = false
endLoop = false
while endLoop is false
  if currentCol < 1 or currentRow > n then endLoop = true
  else if getElement(currentRow, currentCol) = x then
    found = true
    endLoop = true
  else if getElement(currentRow, currentCol) > x then currentCol = currentCol - 1
  else currentRow = currentRow + 1
if found is true then return 'Found at position currentRow, currentCol.'
else return 'Not found.'
```

## Explanation

By restricting its movement to downwards and to the left, the algorithm assures that it will never look at numbers above or to the right of the current one. This is acceptable, because if the number being examined is greater than the number searched for, then so is everything below that number (because columns are sorted in increasing order), so it is safe to move to the left, putting all the lower numbers in that column to the right of the algorithm's "position" in the array, effectively discarding them. Similarly, if the number is smaller than the number searched for, then so too is everything to the left of it, so it is safe to move down one row.

In effect, the algorithm constantly reduces the array to one of a smaller size by discarding a row or column of elements that could not be the one searched for. Since the algorithm starts at the top right, never moves anywhere that improperly discards elements, and always moves, it will eventually be either forced out of the array (which obviously means the number is not in the array), or it will find the number.

## Question 3

This algorithm uses the previously defined method `append()`, which concatenates two sequences.

```
fringe(t)
  if t is null then return ()
  else if t is a leaf then return root(t).()
  else return append(fringe(left(t)), fringe(right(t)))
```

## Question 4

```
fringec(t1, t2, s1, s2)
  if t1 is null then
    if s1 is null then
      if t2 and s2 are null then return true
      else return false
    else return fringec(first(s1), t2, rest(s1), s2)
  else if t1 is a leaf then
    if t2 is null then
      return fringec(t2, t1, s2, s1)
    else if t2 is a leaf then
      if root(t1) = root(t2) then
        if s1 is null then
          if s2 is null then return true
          else return false
        else if s2 is null then return false
        else return fringec(first(s1), first(s2), rest(s1), rest(s2))
      else return false
    else return fringec(t2, t1, s2, s1)
  else return fringec(left(t1), t2, right(t1).s1, s2)
```

## Explanation

$t1$  and  $t2$  are the trees being compared, while  $s1$  and  $s2$  are sequences used to find more leaves after each one is found. They are sequences of the right trees found at each “junction” where the algorithm continued on down to the left.

Starting from the beginning of the algorithm:

If  $t1$  and  $s1$  are null, then all the leaves of  $t1$  have been found. If there is anything left of  $t2$  or  $s2$ , then  $t2$  has more leaves than  $t1$ , so the fringes are not the same. If all four variables are null, then there can be no differences left to find, so the fringes are the same. If  $t1$  is null but  $s1$  is not null, then we go back to the most recent right tree added to  $s1$ , because that is where the next leftmost leaf will be found.

If  $t1$  is not null, then it must be either a leaf, or a tree with other nodes attached to it.

If  $t1$  is a leaf, look at  $t2$ . If  $t2$  is null, there is still a possibility that some leaf will be found by going through  $s2$ , and the algorithm will do just that if we call it, switching  $t1$  and  $t2$  (and their respective sequences). If  $t2$  is a leaf, then we have found two leaves to compare. If they’re not equal, then the fringes are not equal. If they are equal, the fringes might be equal, but only if we find that the rest of the leaves are equal, which means going through the sequences. However, before we do that, we need to make sure the sequences are not null. If they’re both null, the fringes are equal and we’re done. If one is null and the other is not, then one tree has more leaves than the other, so the fringes aren’t equal. If neither is null, then we can move on to the next element in the sequences by calling `return fringec(first(s1), first(s2), rest(s1), rest(s2))`. Still under the condition of  $t1$  being a leaf, if  $t2$  is not a leaf, then we search through it to find one by calling the algorithm with  $t1$  and  $t2$  (and their sequences) switched.

Finally, if  $t1$  is neither null nor a leaf, we look down the left of  $t1$ , saving the right branch in  $s1$  so we can check it later.

## Question 5

1. Added code (entire file found on diskette and at <http://www.cs.mcgill.ca/~chundt/comp250>):

```
public static double recIntegrate(Function f,
                                double xmin, double xmax, double epsilon)
{
    double minVal, maxVal, midVal, midEst, xMiddle;

    xMiddle = (xmin + xmax)/2;

    minVal = f.yVal(xmin);
    maxVal = f.yVal(xmax);
    midVal = f.yVal(xMiddle);

    midEst = minVal + (maxVal-minVal)/2; //find point on estimation line

    //check against epsilon:
    if (Math.abs(midVal-midEst) <= epsilon) {
        return .5*(minVal + maxVal)*(xmax - xmin);
    }
    else {
        return (recIntegrate(f, xmin, xMiddle, epsilon)
                + recIntegrate(f, xMiddle, xmax, epsilon));
    }
}
```

output:

```
Integral of x from 0.0 to 1.0:
    Iterative method (step = 0.01): 0.5000000000000002
    Recursive method (epsilon = 1.0E-4): 0.5
Integral of x^2 from -1.0 to 1.0:
    Iterative method (step = 0.01): 0.6666500000000013
    Recursive method (epsilon = 1.0E-4): 0.666748046875
Integral of x^3 from 0.0 to 1.0:
    Iterative method (step = 0.01): 0.24998750000000045
    Recursive method (epsilon = 1.0E-4): 0.2500346712768078
Integral of x^4 - 2x^2 from -2.5 to 2.5:
    Iterative method (step = 0.01): 18.496988055631086
    Recursive method (epsilon = 1.0E-4): 18.229269626194796
Integral of x^5 - x^4 - 6x^3 + 4x^2 + 8x from -2.5 to 2.5:
    Iterative method (step = 0.01): 2.705393452571584
    Recursive method (epsilon = 1.0E-4): 2.6041584119557895
Integral of x^5 - x^4 - 6x^3 + 4x^2 + 8x from -2.5 to 2.5:
    Iterative method (step = 0.1): 2.6395687500000116
    Recursive method (epsilon = 0.01): 2.6105535773142465
Integral of sin(x) from 0.0 to 3.141592653589793:
    Iterative method (step = 0.01): 1.999972991681736
    Recursive method (epsilon = 1.0E-4): 1.9998761900442124
Integral of e^(-x^2) from -3.0 to 3.0:
    Iterative method (step = 0.01): 1.7724159002839825
    Recursive method (epsilon = 1.0E-4): 1.772525395771758
```

2. One advantage of the recursive algorithm is that a trapezoid is a more “flexible” shape, in that the side that is supposed to match the graph can have any slope. Another advantage of the recursive algorithm, as mentioned in its documentation, is that it tends to avoid wasting calculations by only making its interval smaller when it appears necessary. However, this also means that it is unclear how many recursive calls will be made in estimating the integral of a particular graph. The advantage of the iterative algorithm is that, provided the range of integration and the interval used, it is easy to tell exactly how many calculations will be performed, which translates easily into running time.

The iterative method will be relatively accurate most of the time, but is prone to slight over- or under-estimation. The recursive function can be very accurate in estimating functions which are well approximated by lines, which includes most functions. Its main flaw is that it can't truly tell when the line is close to the graph of the function; it can only tell when the midpoint is close. Thus, the midpoint could randomly end up close at some point where the line is not actually a good estimate.

This brings us to functions which will cause the methods to fail badly. For the iterative method, a function that happens to oscillate or change significantly at a frequency equal to the interval would cause problems. For instance, the integral of  $\sin(100\pi x)$  with an interval of 0.01 would be very badly estimated. For the recursive function, the line of the trapezoid could happen to match the function at the line's midpoint before the trapezoid is a good estimation. One example is  $\sin x + 1$  from  $\pi/2$  to  $9\pi/2$ , which evaluates to 2 at  $\pi/2$ ,  $9\pi/2$ , and  $5\pi/2$ , but whose integral is  $4\pi$ , not  $2(9\pi/2 - \pi/2) = 8\pi$ .