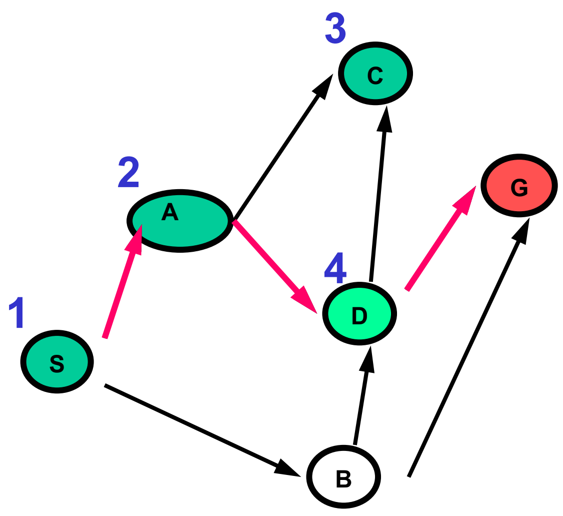
Lecture C8: Trees part II

Response to 'Muddiest Part of the Lecture Cards'

(16 respondents)

1) *Is the tree in the "Depth first" example actually a tree? (It has 6 vertices and 8 edges – a tree has N verticed and N-1 edges)*?
The algorithm/example is finding the 'minimum spanning tree' for a connected weighted graph (this graph has in example used in lecture slides, 6 vertices and 8 edges).

I believe you mean the following **Graph** (shown in figure below), which has 8 Edges and 6 Vertices?



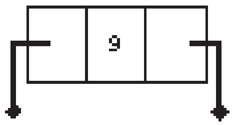
We were trying to find a path from start vertex **S** to goal vertex **G** using the Depth-First-Search algorithm. The result suggested visiting the vertices in order **S**, **A**, **D**, **G**. These four vertices, and the three edges in-between them are a Tree, with N (4) vertices, and N-1 (3) edges.

2) *When implementing trees in code, should loops be used to go through the tree? It seems as though there might be special cases where loops would not work… or not*?

When you are creating a tree, you can use either recursion or iteration. The recursive solution is concise and abstracts a lot of the implementation detail. In the example shown below,

```
procedure Insert (
      Root    : in out Nodeptr;
55.           Element : in     ElementType) is
56.         New_Node : Nodeptr;
57.       begin
58.         if Root = null then
59.           New_Node:= new Node;
60.           New_Node.Element := Element;
61.           Root := New_Node;
62.         else
63.           if Root.Element < Element then
64.             Insert(Root.Right_Child, Element);
65.           else
66.             Insert(Root.Left_Child, Element);
67.           end if;
68.         end if;
69.       end Insert;
```

Assume that only the root node exists, with the element = 9 as shown in figure below.



Main program

Insert(root, 6).

In line 63, the Root.Element > 6 so,

line 66, Insert(Root.Right_Child, 6) is called.

Recursive Insert(null, 6)

Line 58, Root = null

Line 59, create a new node

Line 60, New_Node.Element := 6

Line 61, Root := New_Node

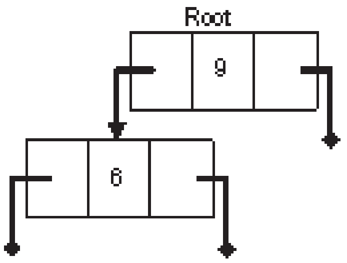
(sets Root.Right_Child to New_Node)

Line 68, end if (does nothing)

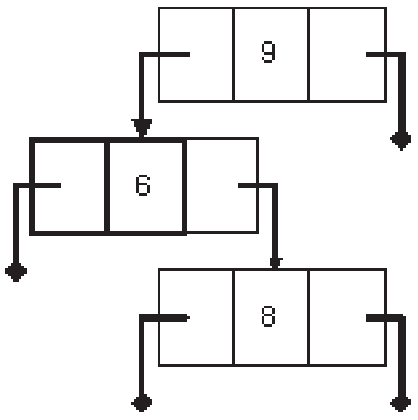
Line 69, end recursive insert (returns control to Insert)

Line 68, end if

Line 69, end insert



Try inserting 8 into the tree. The final tree is shown below.



The same insertion operation can be implemented using iteration as shown below.

```
procedure Insert (
    Root    : in out Nodeptr;
    Element : in     ElementType) is
  Temp,
  New_Node : Nodeptr;
  Inserted : Boolean;

begin

  New_Node:= new Node;
  New_Node.Element := Element;
  New_Node.Left_Child := null;
  New_Node.Right_Child := null;

  if Root = null then
    Root := New_Node;
  else
    Inserted := False;
    Temp := Root;
    loop
      exit when Inserted = True;

      if Temp.Element < Element then
        if Temp.Right_Child /= null then
          Temp:= Temp.Right_Child;
        else
          Temp.Right_Child:= New_Node;
          Inserted := True;
        end if;
      else
        if Temp.Left_Child/= null then
          Temp := Temp.Left_Child;
        else
          Temp.Left_Child:= New_Node;
          Inserted := True;
        end if;
      end if;
    end loop;
  end if;
end Insert;
```

When traversing the tree, recursion helps because you do not have to keep track of the parent node.

3) *How do you choose an algorithm for sorting/trees*?

You have to choose an algorithm based on memory and computation time. For example both bubble sort and insertion sort take O(n^2), but insertion sort takes less memory because sorting occurs in place.

Between bubble sort and merge sort, merge sort takes less computation time because it is an O($n\log n$) algorithm.

4) *How do you determine which one of these search algorithms is more efficient? In other words, could you give examples of when to use each algorithm*?
There are three different ways to figure this out:

1. Experience
2. Study the algorithmic complexity
3. Implement a number of different algorithms and test run them using different sizes of input

There exists a huge number of different sorting algorithms, some are simple and intuitive to implement (e.g., bubble sort), and others are very complicated (e.g., using recursion, advanced data structures, multiple arrays, …) to implement (e.g., an algorithm called quick sort), but they produce answers very quickly.

The most common sorting algorithms can be split into 2 classes based on their complexity. Algorithmic complexity is a very complex topic, but will be covered in upcoming lectures. There is a correlation between an algorithms complexity and how efficient the algorithm is. Algorithmic complexity is written using the Big-O notation. E.g., O(n) means that an algorithm has a linear complexity (i.e., it takes 10 times longer time to run the algorithm on an input size n=100 that it does on an input size n=10), and the value n represents the size of input that the algorithm is run on. One class of sorting algorithms is O(n^2) (e.g., bubble, insertion and selection sort), and the other class has the complexity O($n \log n$) (e.g., heap, and merge sort)

Bubble sort - O(n^2)
Straight forward and simple to implement, but a very inefficient algorithm. Mostly just used in teaching as a first example of a sorting algorithm.

Insertion sort - O(n^2)
Similar to Bubble sort, easy to implement, and somewhat more efficient since the number of elements having to be compared is reduced somewhat with each pass thorugh the algorithm. The algorithm is inefficient for large values of n, but still a good choice for small n (lets say, < 25) or for nearly sorted lists.

Merge sort - O(*n* log *n*)
Megrge sort is slightly faster than heapsort for large values of *n*, but uses recursion and uses at least twice the memory as required by heapsort.

Heapsort - O(*n* log *n*)
Is the slowest of the O(*n* log *n*) algorithms, but it does not require a large number of recursive calls or extra arrays. This makes heapsort a very attractive alternative for very large values of *n*.

In addition to study the algorithmic complexity of algorithms, we can study the speed of various implementations by comparing them with empirical data.

5) *What is this 'weight'? Why would minimizing the weight of the tree be helpful*?
Weight could be 'cost' as in kilometers, minutes, dollars or any other unit. If you want to visit all the nodes (e.g., airports) in Massachusetts (where the edges represent which 2 airports are connected with a flight route), you might be interested in figuring out what order all airports should be visited in order to minimize time taken, dollars spent or some other unit.

6) *How will we know which is the best/most efficient sorting algorithm to use*?
By experience, or by knowledge of how to calculate the Big-O of the algorithms, or by using a good reference, e.g.,

- Donald E. Knuth "*Volume 3: Sorting and Searching*." Second Edition (Reading, Massachusetts: Addison-Wesley, 1998), xiv+780pp.+foldout. ISBN 0-201-89685-0
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, "Introduction to Algorithms, Second Edition" September 2001 ISBN 0-262-03293-7, 8 x 9, 1184 pp. The MIT Press

7) *Was the point of the spanning trees to find the cheapest way to visit all the nodes*?

The main idea behind the spanning tree is to find the shortest distance between two nodes in a graph. The problem of finding the cheapest path to visit all nodes is called the *traveling salesman* problem.

8) *What is Ada.Unchecked_Deallocation*?
If you want to deallocate (return) memory to the "storage pool" that your program no longer is needing/using, then you can instantiate the generic package Unchecked_Deallocation. The word 'Unchecked' comes from that the procedure does no live object analysis before it deallocates the memory.

```
with Ada.Unchecked_Deallocation;
```

```
package body Lists is

-- Routine for deallocating nodes
procedure Free_Node is new Ada.Unchecked_Deallocation
               (object => node, name => pointer);
```

`Free_Node` could then be used in the following way

```
Head := new Node;
…
Free_Node (Head);
```

9) *No mud* (5 student)