Algorithms and Imperative Programming

COMP26120

Todd Davies

May 2, 2015

Introduction

This is a two-semester practical introduction to algorithms and data structures, concentrating on devising and using algorithms, including algorithm design and performance issues as well as 'algorithmic literacy' - knowing what algorithms are available and how and when to use them.

To reflect the emphasis on practical issues, there are two practical (laboratory) hours to each lectured hour. Lectures serve to motivate the subject, orient students, reflect on practical exercises and impart some basic information. A range of practical applications of algorithms will also be presented in the lectures. Other information resources will be important, including a set textbook, which will provide essential support.

The course-unit starts with a 5-week primer on the C programming language, enabling students to become competent programmers in this language as well as in Java (and, possibly, in other languages). This teaching is supported by an on- line C course and extensive laboratory exercises.

There is a follow-up course unit on Advanced Algorithms in the Third Year. This presents the foundational areas of the subject, including (1) measures of algorithmic performance and the classification of computational tasks by the performance of algorithms, (2) formulating and presenting correctness arguments, as well as (3) a range of advanced algorithms, their structure and applications.

Aims

- To make best use of available learning time by encouraging active learning and by transmitting information in the most effective ways.
- To give students a genuine experience of C.
- To make students aware of the importance of algorithmic concerns in real-life Computer Science situations.
- To emphasise practical concerns, rather than mathematical analysis.
- To become confident with a range of data structures and algorithms and able to apply them in realistic tasks.

Additional reading

Algorithm design: foundations, analysis and internet examples - Goodrich, Michael T. and Roberto Tamassia

Contents

1 Keys and total orders

It is often important to be able to implement some kind of comperator between data items. This is often achieved in the from of a total order relation, which has the following three properties:

Reflexive property $k \leq k$

Antisymmetric property $(k_1 \leq k_2 \wedge k_2 \leq k_1) \implies k_1 = k_2$

Transitive property $k_1 \leq k_2 \wedge k_2 \leq k_3 \implies k_1 \leq k_3$

A comperator that has the above three properties defines a linear ordering relationship between data items. This means there will be a smallest item k_{min}, where $k_{min} \leq K$ for all K in the collections of data items.

2 Trees

2.1 Definition

A tree is an abstract data type for hierarchical storage of information. Each element in a tree has a parent element, and zero or more children elements. The node at the top of the tree is called the root.

A sub-tree is the tree consisting of all the descendents of a child of a tree, including the child itself.

A tree is said to be *ordered* if a linear ordering relation is defined for the children of each node, that is to say that if we wanted to, we could apply this relation to sort the children into an ordered list.

A binary tree is one where each node can have a maximum of two children. A binary tree is *proper* if each node as two (or zero) children. At each level of a binary tree, the number of nodes in that level is at most 2^d where d is the level of the tree (starting from 0).

The depth of a node is the number of ancestors of the node excluding the node itself.

2.2 Tree algorithms

2.2.1 Depth of a node

The depth of a node in a tree is the number of ancestors of the node, excluding the node itself.

```
int depth() {
  if(parent == null) {
    // We've got no parent; we are the root!
    return 0;
  } else {
    return parent.depth() + 1;
  }
}
```

This algorithm runs in $O(n)$ time and space, since it's dependent on the depth of the tree, and is recursive.

Note that recursive algorithms always use space proportional to the number of times they have recursed, since (unless tail recursion is used), each recursion will use a stack frame.

2.3 Height of a tree

A simple way to find the height of the tree, would be to iterate over every node (maybe using a tree traversal algorithm mentioned in section **??**), and find the depth of each, keeping track of the maximum depth. This would be a $O(n^2)$ algorithm.

A better approach is to use a recursive definition, and start from the root of the tree. We can find the height of all of the child nodes, and return that plus one.

```
int height() {
  if(children.size() == 0) return 0;
  else {
    int max = 0;
    for(Tree child : children) {
      int childHeight = child.height();
      if(childHeight > max) max = childHeight;
    }
    return max + 1;
  }
}
```

2.3.1 Tree traversal

There are two different traversal schemes for trees; pre-order and post-order. Each visits the elements in the tree in a different order. The following code shows two different map functions, iterating in pre-order and then post order.

```
void mapPreOrder(Tree* root, void (*action)(Tree*)) {
  action(root);
  for(int i = 0; i < root->numChildren; i++) {
    root->children[i].mapInOrder(action);
  }
}
void mapPostOrder(Tree* root, void (*action)(Tree*)) {
  for(int i = 0; i < root->numChildren; i++) {
    root->children[i].mapInOrder(action);
  }
  action(root);
}
```

You can also iterate in-order if your tree is a binary tree, you visit the left child first, call the function on the current node, and then visit the right child.

All the traversal algorithms take $O(n)$ time.

2.4 Tree datastructures

There are two *main* ways to store binary trees in memory; using a list of nodes (in a heap style), or by using a linked datastructure, having nodes point to other nodes.

2.4.1 Using a vector based datastructure for trees

In the vector (list/array/whatever you want to call it) style, the root of the tree is stored at the start of the array. To calculate the index of the left child of a node, you multiply the index of the current node by two. To find the index of the right child of a node, you multiply its index by

two and add one. This numbering function is known as *level numbering*, and can be implemented like so:

```
int left(int n) { return 2 * n; }
int right(int n) { return (2 * n) + 1; }
```

The running times of a vector-backed binary tree are good. Iteration can be done in $O(n)$ time with a low overhead, swapping elements is $O(1)$ as is replacing them.

2.4.2 Using linked nodes to form a tree datastructure

The trouble with using a vector datastructure, is you need to initialise an area of memory equal to $2^{depth} \times sizeof(Tree)$. This means that for deep trees, you will be wasting very large amounts of memory. This is a rather extreme case of a memory-speed trade off.

In a linked data structure, each node points to all of its children. A really simple example of a binary tree one could be:

```
public class Tree<T> {
  public Tree<T> left, right;
  public T value;
}
```

If wanted to represent general (i.e. not binary) trees, we would have to modify the datastructure so that we could have any number of children:

```
public class Tree<T> {
  public List<Tree<T>> children;
  public T value;
}
```

3 Priority Queues

A priority queue is a datastructure capable of ordering items based on some associated key. The two most important operations implemented on it are `insertItempriority, item` and `removeMin()`.

Priority queues are the basis of some sorting algorithms, for example heap sort (heaps are the datastructure behind most priority queues). To sort a list using this method, add all the items to the priority queue in any order, fill up the array again in the order that the elements come out of the queue.

```
public <T implements Comparable<T>> List<T> sort(List<T> list) {
  PriorityQueue<T> pQueue = new PriotityQueue<T>();
  while(!list.isEmpty()) {
    pQueue.add(list.remove(0));
  }
  while(pQueue.isEmpty()) {
    list.add(pQueue.poll());
  }
  return list;
}
```