Algorithms and Imperative Programming

COMP26120

Todd Davies

May 3, 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, smooth sort, selection sort and insertion sort. The different type of sort depends on how the priority queue is implemented. 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;
}
```

3.1 Heaps

Priority queues are often implemented using a heap as the backing datastructure. A heap is a binary tree that stores a collection of values. The type of values stored by the heap must have a total order relationship, since otherwise, the heap cannot be constructed correctly. The two properties that make heaps different from normal binary trees are:

Heap order property:
For every node v in the tree, the value of v is greater than or equal to the value of its parent. The only exception is the root, since that doesn't have a parent.

This means that if you start from the root, and move towards any leaf, then the nodes you encounter will be in a non-decreasing order. It also means that the minimum key is at the root.

Complete binary tree property:
This means that if heap has a depth of d, then all levels of the tree from 0 to $d-1$ must be completely full (i.e. they have 2^{level} nodes in), and the last level is filled up from left to right with the remaining elements.

There are no operations on a heap run that in worse than $O(log(n))$ time. The operations are performed in a time proportional to the height of the tree rather than the number of its elements.

3.1.1 Insertion

To store a new element in the heap, we add a new node to the heap at the next available position (the last position in the tree). If the heap is backed by an array, then the index of the array will be $n+1$, where n is the current heap size.

After we've inserted the new element we must check that the heap properties aren't violated. The *complete binary tree* property will be fine since we added the element at the end of the array (which is always the last element in the current level, or the first element in a new level), but the *heap order* property may be violated.

To maintain the *heap order* property, we need to 'bubble' the newly inserted item up the heap until the property is restored. This is achieved by comparing the key of the newly inserted node n with its parent p. If $k(u) > k(p)$, then they are swapped around (which is literally caused by indexes the two items in the array). In the worst case, you would insert the (new) smallest element into the heap, and the element would bubble all the way up to the top, taking $log(n)$ time. This is called 'up heap bubbling'.

3.1.2 Removal

If we want to remove the smallest element from the heap (as is required for a priority queue), then that will be the root element of the heap. However, if we just removed that element, then we would have two binary trees not one (a left and a right tree).

Instead, we move the last element in the array into the first element, and return the original first element (the last element is deleted). The *complete tree* property is now satisfied again. Now we have the element previously at the bottom of the heap at the top, so we need to move it into a position that will satisfy the *heap order* property.

To restore the *heap order* property, we must 'down heap bubble' the root element. If the root element is the only node, then the property is already satisfied. If the only child is on the left, then let s be the left child, and otherwise, let s be the smallest (direct) child of the root node.

If $k(root) > k(s)$ then the heap order property is restored by swapping r and s. You continue down the tree, swapping what was the route node until no violation of the heap order property

occurs. Since we may have to move all the way down the tree from root to leaf, the runtime of this is also $O(log(n))$.

3.1.3 Heap sort

Since insertion and removal are both $O(log(n))$, and we're inserting n elements, and then removing n elements, then the runtime of a heap sort will be $n \times n\log(n)$. Since constant factors are removed from the Big-O notation, then the runtime is $O(n\log(n))$. This is a large improvement from selection and insertion sort that take n^2 time. If the sequence was implemented as an array, then the sort can be done in place too!

4 Dictionaries and Hash Tables

In a dictionary, the user assigns keys to elements, and the keys can be used to look up elements in the datastructure (also known as a map in other languages such as `java`). Dictionaries can be unordered, or ordered. When keys are unique, then the dictionary is referred to as an associated store.

If the keys are unique, then a good way to implement the dictionary is a hash table. Here, the key is hashed into an integer using a hashing function. The result is used as the index of an array of 'buckets'.

Each bucket stores a collection of values that have the same hashed key. Ideally, each bucket will hold one element. This is easy to ensure for some data, for example, if we had a Hash Table of the integers from 0-50, then we could use the integer as the index to an array of fifty elements. In general though, it is not possible to ensure that no collisions will occur, and one bucket will probably end up with multiple elements. In this case, we need a collision resolution strategy.

If we have a collision resolution strategy, then we don't need to have an array that is as big as the number of items we're going to hold. We could just rely on the collision resolution strategy to let us have a smaller array and therefore need less contiguous memory space. Even if we do have the memory to spare, its wasteful to have a massive array if we're only going to store a few items in there.

If our hash function produces integers as indexes that are greater than the number of buckets, then we can either take the modulus of that number by the size of the array, or use the MAD method:

$$hash = |ak + b|modN$$

Where N is a prime, and a, b are random non-negative integers such that $amodN \neq 0$. The MAD (Multiply Add and Divide) method means the chance of a collision is $\frac{1}{N}$.

4.1 Collision handling

There are different collision handling schemes, and each has its own benefits and drawbacks. Here are just three:

Separate chaining:
: This is most often implemented using a Linked List for the buckets. Basically, when you want to add an item to the bucket, you just append it to the existing Linked List of items.

Open addressing:
: In this strategy, if the index of the hash value is already occupied, then the buckets are examined in some sequence (the easiest is linear probing, which just looks at consecutive elements in the array), until a free slot is found.

Double hashing:

Here, a second hashing algorithm is used to find an alternative slot when the first shows a collision. If we did j attempts to find a new slot, the new index would be $hash1+(j \times h(key))$, where h is the second hashing algorithm.

Different collision resolution schemes exhibit different performance as the load factor changes. The load factor is the number of entries divided by the number of buckets.

If the load factor becomes too high, then we could *rehash* the table. This involves increasing the number of buckets, and re-inserting all the elements currently in the table so that they are distributed according to the new size of the array.

4.1.1 Probing

Probing is part of the open addressing collision resolution strategy. Linear probing is easiest, you just walk through the array one by one until you find an empty slot. Quadratic probing is harder, the next index to try and insert the element into is found using an equation such as $|index + j^2|modN$, where j is the j^{th} attempt at finding a slot.

Removing elements from a open addressed hash table is slightly more complicated, but can be resolved by marking elements as 'deactivated' rather than removing them.