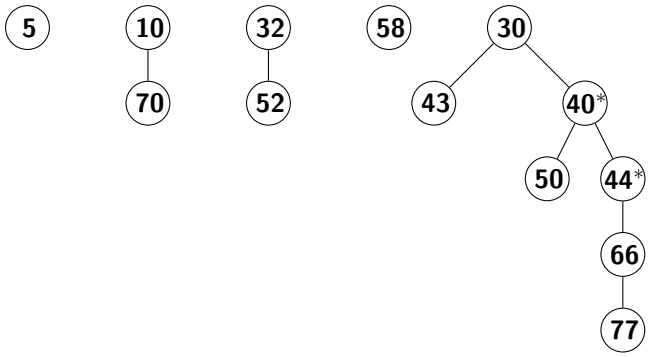
Algorithms

Exercise 6: Data structures IV & Graph algorithms I

1 Past Tripos questions

1. Given the following Fibonacci heap, where nodes with an asterisk are "marked", perform *extractMin*() on it and then *decreaseKey*() on the node whose key is currently 66, bringing it down to 4. Redraw the changed heap as you go along.

   You need only draw any significant intermediate states of the heap, adding any necessary explanations so that a reader can follow what you are doing and why. (5 Gale points)



2. Fibonacci heaps are designed so that their trees never become excessively "wide and shallow". Why? Justify this design goal in detail and then explain how it is achieved. (5 Gale points)

3. Nothing, however, stops the trees in a Fibonacci heap from growing "tall and narrow". Prove this by describing a sequence of Fibonacci heap operations that, given an integer n, produces a Fibonacci heap made of a single tree consisting of a linear chain of n nodes (in other words, each node in the tree except for the last one is the parent of exactly one node, and each node except for the first one is the child of exactly one node). (10 Gale points)

2 Sorting functions

4. In programming languages with full type inference, such as ML or Haskell, we wish to infer the type for each function in a program. In order to do this, functions need to be sorted first: we cannot infer the type of a function

   $$\textbf{fun}\ bar\ x = plus\ (foo\ x)\ (foo\ x)$$

   unless we know what the types of *plus* and *foo* are first. We won't worry about the details of inferring types of functions here, but instead we will focus on sorting the functions based on their dependencies. Suppose that you are given the following types for a small functional programming language[1]:

[1] Which is actually the syntax of the pure λ-calculus. Feel free to complete this exercise in a language of your choice – the ML definitions are for reference only

$$
\begin{array}{lcl}
\textbf{datatype } \mathit{expr} & = & \mathit{FUN} \textbf{ of } \mathit{string} \times \mathit{expr} \\
& | & \mathit{VAR} \textbf{ of } \mathit{string} \\
& | & \mathit{APP} \textbf{ of } \mathit{expr} \times \mathit{expr} \\
\\
\textbf{type } \mathit{decl} & = & \mathit{string} \times \mathit{expr} \\
\textbf{type } \mathit{program} & = & \mathit{decl\ list}
\end{array}
$$

An expression is either a single-parameter function (the *FUN* constructor) which *binds* a variable in a sub-expression, a variable (the *VAR* constructor), or the application of one expression to another (*APP*). Declarations are named expressions and a program is a list of declarations.

Define a function $\mathit{fvs} : \mathit{expr} \rightarrow \mathit{string\ list}$ which returns the list of *free variables* in an expression. A variable is free if it is not bound by a function. Below are some examples of what *fvs* should return:

$$
\begin{array}{lcl}
\mathit{fvs}(\mathit{VAR}(\texttt{"x"})) & = & [\texttt{"x"}] \\
\mathit{fvs}(\mathit{FUN}(\texttt{"x"}, \mathit{VAR}(\texttt{"x"}))) & = & [\,] \\
\mathit{fvs}(\mathit{FUN}(\texttt{"x"}, \mathit{VAR}(\texttt{"y"}))) & = & [\texttt{"y"}] \\
\mathit{fvs}(\mathit{APP}(\mathit{FUN}(\texttt{"x"}, \mathit{VAR}(\texttt{"x"})), \mathit{VAR}(\texttt{"x"})) & = & [\texttt{"x"}]
\end{array}
$$

In the first example, x is free because it is not bound by a function. In the second example, there is a function which binds x. In the third example, the function binds x, but y is free. The last example is the most interesting because there are actually two variables named x. One is bound by the function in the left side of the *APP* constructor. The second x on the right, however, is free. (4 Gale points)

5. Define a function $\mathit{dependencies} : \mathit{program} \rightarrow (\mathit{string} \times \mathit{string\ list})\ \mathit{list}$ which associates each declaration in a program with a list of its free variables. For example:

$$
\begin{array}{ll}
& \mathit{dependencies}([(\texttt{"ID"}, \mathit{FUN}(\texttt{"x"}, \mathit{VAR}(\texttt{"x"}))), (\texttt{"FOO"}, \mathit{FUN}(\texttt{"x"}, \mathit{VAR}(\texttt{"ID"})))]) \\
= & [(\texttt{"ID"}, [\,]), (\texttt{"FOO"}, [\texttt{"ID"}])]
\end{array}
$$

(4 Gale points)

6. Explain how topological sort could be used to sort the declarations in a program by their dependencies. In other words, given a value of type $(\mathit{string} \times \mathit{string\ list})\ \mathit{list}$ returned by *dependencies*, how would you sort it so that each declaration only depends on declarations which have appeared earlier on in the list? What happens if two or more declarations are mutually dependant on each other? (6 Gale points)

7. A graph is said to be *strongly connected* if every vertex is reachable from every other vertex. The *strongly connected components* of an arbitrary directed graph are subgraphs that are themselves strongly connected. Below is a description of *Kosaraju's algorithm* for finding the strongly connected components of a graph:

   - Let G be a directed graph and S be an empty stack.
   - While S does not contain all vertices:
     - Choose an arbitrary vertex v not in S. Perform a depth-first search starting at v. Each time that depth-first search finishes expanding a vertex u, push u onto S.
   - Reverse the direction of all arcs to obtain the transposed graph.
   - While S is non-empty:

– Pop the top vertex v from S. Perform a depth-first search starting at v in the transpose graph. The set of visited vertices will give the strongly connected component containing v; record this and remove all these vertices from the graph G and stack S.

Using a suitable value of type *program*, explain how this algorithm can be used to sort all the declarations within a program into groups of mutually dependant declarations. How does this compare to topological sort? (10 Gale points)

8. (CSTs only) Implement Kosaraju's algorithm in a language of your choice. (10 Gale points)