Imperial College London – Department of Computing

MSc in Advanced Computing
MSc in Computing Science (Specialism)

531: Prolog
'Heap' – Assessed

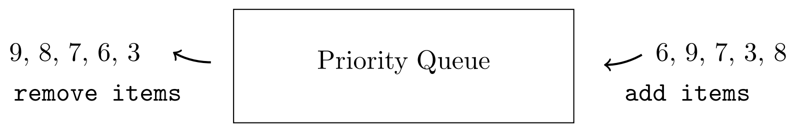
Issued: 31 October 2018 **Due:** 8 November 2018

Introduction

In this exercise you will implement a *binary max-heap* data structure in Prolog.

Priority Queues

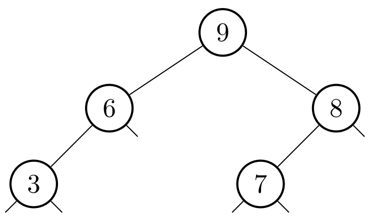
A binary heap is an efficient way to implement a *priority queue*, commonly used in AI in *best-first search.* A priority queue is a collection of items each of which is assigned a key (not necessarily unique) giving it a priority. A *max* priority queue ensures that the item(s) assigned the highest priority is always at the front, regardless of the order in which the items were added. A *min* priority queue implements the opposite ordering. This exercise involves a *max* queue. The operation of a max queue is illustrated below. Please note that only the keys are shown. The items themselves have been omitted for simplicity.



Heap Ordering

A binary max-heap is a binary tree that is *max heap-ordered.*

A binary tree is max heap-ordered iff, for every key-item pair (K, I) in the heap, the key K is greater than or equal to the keys of the children of (K, I).



A binary max-heap (only keys are shown).

Prolog Representation

In your program a heap should be represented by one of the following two terms:

- `empty` to represent an empty heap,
- `heap(K, I, LH, RH)` to represent a non-empty heap,

where, in a term `heap(K, I, LH, RH)`:

- `K` (an integer) is the key of the item on top of the heap (the root of the tree),
- `I` (a term) is the item on top of the heap,
- `LH` (a binary heap) is the left sub-heap,
- `RH` (a binary heap) is the right sub-heap.

What To Do

Obtain the Exercise Files

You will need to clone the skeleton respository to your DoC home directory in order to work on it. Later you will need to *push* your changes back to the server.

- You can get your skeleton repository by issuing the following command (all on one line, replacing the occurrence of *login* with your DoC login):

  ```
  git clone https://gitlab.doc.ic.ac.uk/lab1819_autumn/531_heap_login.git
  ```

  Or, if you have set up ssh key access you can use:

  ```
  git clone git@gitlab.doc.ic.ac.uk:lab1819_autumn/531_heap_login.git
  ```

- This will create a new directory called `531_heap_`*login*. Inside you will find the following files / directories:
  - `heap.pl` — this is the source file you should edit to implement the procedures required for this exercise.
  - `support.pl` — this file contains procedures to help you debug your solution.
  - `.git` and `.gitignore`

Write Your Program

Task 1

Write a Prolog program for `is_heap(+H)`. A call `is_heap(H)` should succeed iff `H` is a binary max-heap as defined above.

Balancing the Heap The next tasks require you to write procedures that will add nodes to a heap and remove nodes from it. The performance of these procedures can both be kept to $O(\log N)$ by minimising the height of the heap (height corresponds to the length of the longest branch). This can be achieved by maintaining the following *balance invariant*. A heap `heap(K, I, LH, RH)`, where `LH` contains N_L nodes and `RH` contains N_R nodes, satisfies the balance invariant iff:

- $N_L = N_R$ or $N_L = N_R + 1$, and
- `LH` is either `empty` or satisfies the balance invariant, and
- `RH` is either `empty` or satisfies the balance invariant.

NOTE. Your `is_heap` procedure in Task 1 does *NOT* need to check if the heap satisfies the balance invariant.

Task 2

Write a Prolog program for `add_to_heap(+K, +I, +H, -HNew)`. A call `add_to_heap(K, I, H, HNew)` should add the node `(K,I)` with key `K` to max-heap `H`, and return the resulting max-heap `HNew`. You can assume that `I`, `K` and `H` are not variables, are the correct types, and that `H` satisfies the balance invariant. `HNew` must also satisfy this invariant.

HINT. The aim is to add a node in time proportional to $log_2(N)$, in the worst case. Therefore, any algorithm that might iterate over the whole heap (e.g. to count nodes) will be too slow. Your program should avoid this. One quite elegant solution uses a *rotation* within the heap. This kind of technique is used in other *self-balancing trees*, e.g. red-black trees. A rotation reorganises the tree by moving subtrees around. In this case a very simple rotation should suffice. The table below might help you discover it. For each possible configuration of the input heap `H`, the table shows the result of adding one node to the *right* sub-heap.

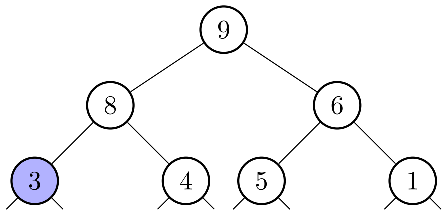
	Initially		After adding one node to `RH`	
	`LH`	`RH`	`LH`	`RH`
Case 1	N	N	N	$N+1$
Case 2	$N+1$	N	$N+1$	$N+1$

The file `support.pl` contains the program `portray_heap(+H)` which will print a visual representation of heap `H` to standard output. You may find this useful when debugging your program.

Task 3

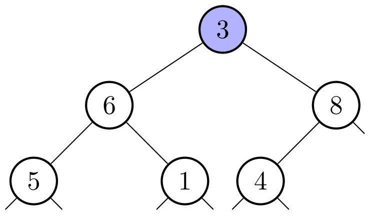
Write a Prolog program for `remove_max(+H, -K, -I, -HNew)`. A call `remove_max(H, K, I, HNew)` should remove the node `(K,I)` with the highest valued key `K` from max-heap `H`, returning `K`, `I` and the resulting max-heap `HNew`. You can assume that `H` is not a variable and is a valid, balanced heap. In order to ensure that `HNew` is correctly heap-ordered and satisfies the balance invariant you should implement the following algorithm.

1. Remove a node from the left sub-heap. (Write a sub-program to do this.) It is best to remove the *left-most leaf* since it will have no children that would need to be relocated. Make sure balance is maintained within `LH`.



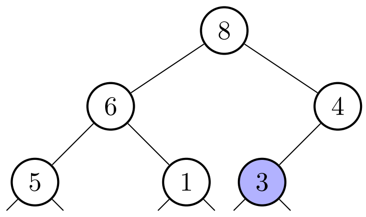
Remove the shaded leaf.

2. Replace the node at the top of the heap with the one you have removed from the left sub-heap and restore the overall balance.



This "heap" is now balanced, but not properly ordered.

3. Push the top node down the heap until the proper heap-ordering is restored.



The final heap.

Task 4

The *Heapsort* algorithm can be stated very simply. To sort a list, first create a heap from the list and then create a second list by removing each item from the heap.

Implement Heapsort as a program `heap_sort_asc(+L,-S)`. A call `heap_sort_asc(L,S)` should sort the list `L` of `(Key,Item)` pairs into *ascending* order by key, returning the sorted list `S`. You can assume that `L` is not a variable. Use your answers to Tasks 2 and 3 to build and then dismantle a max-heap. You can attempt this question by assuming that you have correct programs for Tasks 2 and 3, even if your own answers are not complete. For full credit your solution should be tail recursive and should not require any list to be reversed. You may *not* use any built-in predicates.

Task 5

To change the priority of an item in a queue, say, we cannot simply update the key, since heap ordering would be violated. Instead, we must delete the node, and then add a new one.

Write a Prolog program for `delete_from_heap(+I, +H, -NewH)`. A call `delete_from_heap(I, H, NewH)` should remove the node `(K,I)` from balanced max-heap `H` and return the resulting balanced max-heap `HNew`. The call should fail if there is no node containing `I`. You can assume that `H` and `I` are not variables, and that `I` occurs at most once in `H`.

There are three cases to consider:

1. `(K,I)` is the root node.
2. `(K,I)` is in the left sub-heap.
3. `(K,I)` is in the right sub-heap.

For Cases 2 and 3 you will need to consider what step (rotation?) or steps are needed to restore the balance. Use a case analysis like the one provided in Task 2. Case 2 should be the simpler one. For Case 3 an extra step is required. Consider how something similar is done in the algorithm you were given for Task 3. (If you were thinking there was a fourth case, the failure case, then remember this is Prolog, so we only care about what is true, right?)

Testing

You should test your program on a range of suitable examples before submission. You can automate your testing using the `plunit` Sicstus library. See the Sicstus documentation for details. The file `support.pl` contains the program `portray_heap(+H)` which will print a visual representation of heap `H` to standard output. You may find this useful when debugging your programs.

Your submitted work will be automatically tested, so you must ensure that:

- Your Prolog program is written in `heap.pl`.
- Your program *COMPILES WITHOUT ERRORS* on the **Linux Sicstus** system installed on the lab machines.
- You have *NOT* included any print or write statements in your submitted code. Use them by all means for tracing/debugging but do not include them in your submitted program.

You can verify that your code runs and produces sensible output by requesting an autotest online via `https://teaching.doc.ic.ac.uk/labts`. This is NOT the full set of tests that will be used in assessing your work, so you cannot assume that your program is correct if all the tests pass.

However, you will be able to check that your code was loaded and executed without errors. (In particular look at the log output as well as the test results.)

Submission

Submit By: 8 November 2018

Submission is a two stage process.

1. **Push To Gitlab.** Use `git add`, `git commit` and `git push` to update the Gitlab server with the changes you have made to the skeleton repository. Use `git status` to confirm that you have no local changes you have not pushed, and then inspect the files on Gitlab:

   `https://gitlab.doc.ic.ac.uk`

2. **Submit directly to CATE.** Go to LabTS (`https://teaching.doc.ic.ac.uk/labts`), find your list of commits for this exercise and click the **Submit to CATe** button for the commit you want to submit.

Assessment

This exercise is worth 15% of the marks allocated for coursework on this module. Your solutions will be marked on correctness, code design and readability.

Grades

`F-E:`	The answer shows a lack of understanding of how to write Prolog code. The behaviour of the program is likely to be incorrect.
`D-C:`	The submission shows that the student can write Prolog, but the behaviour of the program is incorrect. Solutions may have serious efficiency problems. Solutions may be grossly overcomplicated, or otherwise poorly designed. Code style may be poor.
`B-A:`	The behaviour of the program is mostly or entirely correct. Solutions may have efficiency problems, and/or some design flaws. Code style is good.
`A+:`	The behaviour of the program is correct. Some minor efficiency problems may exist in some predicates, but otherwise code is well designed, simple and elegant. Code style is good.
`A*:`	There are no obvious deficiencies in the solution, including efficiency of the program, program design and the coding style.

Return of Work and Feedback

This exercise will be marked and returned by **23rd November 2018**. Feedback on your solution will be given on the returned copy.