Assignment 1

Probabilistic data structures and linked lists: leap lists

Motivation

As you know: linked lists have a number of advantages compared to arrays. For example, they do not need contiguous memory and their size can dynamically grow at run time. Insertions and deletions are also easy to implement. However, one key disadvantage is that lookups are generally slow and linked lists cannot speed up the search process if a set of items is ordered. This is where a probabilistic data structure such as a *leap list* can help.

Instead of a single linked list, we keep several of them! At the base level, we keep our items in a singly linked list. The only difference is that the list is sorted. You could ask: how does this help if I still have to traverse each element in the (singly) linked list? To explain this, think of the Melbourne train network. In Melbourne we use express trains that do not stop at every station but only at a few of them. If your final destination is not among them, then you need to change to a slower (i.e., more frequently stopping) train at an exchange stop just before your final destination. You could -- of course -- also change to a train that stops at every station earlier but that would just slow down your journey as the train has to stop more often.

Introduction

Our additional leap lists do behave in just the same way: they do not store every item in the original list but provide long range pointers and store only a few of them (again these items are still sorted of course as they just leave out some elements from the base list). How do we select the items in the second list that we wish to keep? We choose them randomly! Let p denote the probability of keeping an item. For example, p might be 0.5 whether or not we keep an item in the second list. Of course, we can generalise this even further and have a third list that keeps even less items, again with the same probability p of drawing items from the second list. Each of these lists will be a proper subset of a list on a lower level. We can continue this process to a given maximum level.

Our data structure might have h lists in total, constructed as described above. We call h the height of our leap list data structure. Remember that if an item is in a list of height h, then every list in that data structure with a smaller height $h' < h$ also contains that item.

Implementation

In the next step, formalise the operations on our abstract data type, i.e., the leap lists. You will need to implement three key operations: *searching* for a key, *inserting* an element and *deleting* an element.

For simplicity, you may assume that we do not store any duplicates.

Searching for a key (findKey)

When you are searching for a key k, you start at the top layer and search for the largest element e in that list that is $\leq$ than the key. If $next(e)$ is the next element in the list (the top layer), then we have $e \leq k < next(e)$. This is called the *scan forward* step. If $k = e$, you have found the key. Whilst we could terminate here, we wish to return the position on the lowest level. This means we continue to drop down a level (whether or not we have found the key), and continue the search. This is called the *drop down* step.

Once you have found the key on a higher level, there are no further scan forward steps. The algorithm simply drops down to the lowest level and returns the position of the key at the lowest level.

If the key has not been found on a level l_i, we drop down to level l_{i-1} and continue to scan forward for the right position, i.e., the largest element e_{i-1} in the list at level l_{i-1} such that $e_{i-1} \leq k$.

On the lowest level, you continue to search for the key if it has not been found so far. There are three outcomes:

1. you find the key and return its position.
2. you find an element that is larger than the key. The search aborts and returns $NULL$.
3. you have reached the end of the list, i.e., the key is not in the list and the search returns $NULL$.

Note that it might happen that you need to immediately have to drop down 1 or more layers from the top list, for example, if the entries all greater than the key we are searching for.

Please note that you are asked for a slightly different variant of findKey in your actual implementation.

Inserting an element (insertKey)

To insert an element n, you perform findKey with the new element n and find the position of n. This will lead to the largest element e on the lowest list that satisfies $e < n$. We insert n directly after e. After inserting n, you insert n on the higher level with probability p (you could imagine a coin flip if $p = 0.5$). If you did not insert the element (because the coin flip was not successful or your random value was greater than p, then you stop and do not insert the item. Otherwise, you insert the item at the correct position (remember how we do a search). You repeat this process until your coin flip (probabilistic insertion) stops or you have reached the maximum height h. Note that a key insertion could lead to a larger height for an existing leap list (but of course not exceed the maximum height).

Deleting an element (deleteKey)

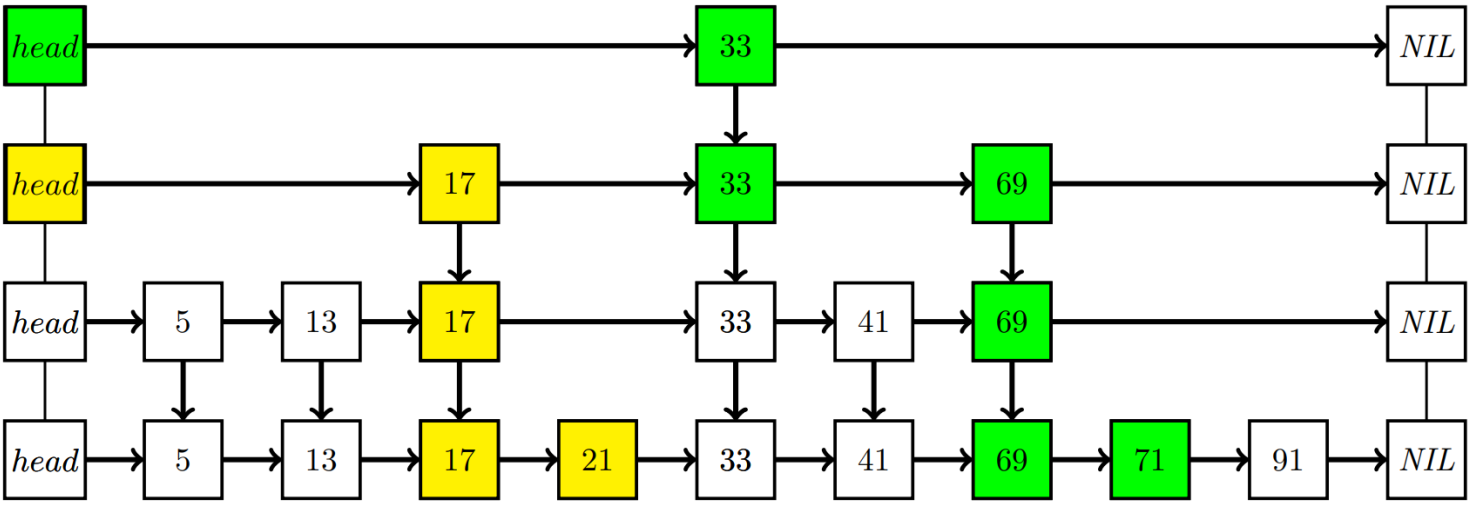
To delete an element d, you again search for it using findKey. If it does not exist, return $NULL$. If d does exist, simply delete the element d in the base list and in all lists with a higher level than the base list. Since the findKey function will also provide you with all occurrences of the key in all higher lists, it is easy to do this.

Creating and deleting leap lists

As always we need a function to create a leap list (leapList) and one to free all memory once we exit our program (freeList). You will need to create those functions in your program as well.

Example

The figure below visualizes two successful searches. First, we search for the value 21. Since 21 is less than 33 (the value in the top list), we drop down to the second highest level (technically the top-left node would have to be colored in yellow as well). Since 21 is larger than 17 but less than 33, we drop down one level at node 17. Since the next node is again 33 on that level, we drop down to the lowest level. We find 21 as the next node and the search terminates. Another run for the key 71 is shown by coloring the visited elements in green.



Problem 1 - Programming Leap Lists

This section comprises the programming for the implementation of leap lists.

Part A

The first part of the task will be to implement the handling of insertion of elements and querying those values by performing a search.

Input

The input to the program is delivered on stdin, it follows the following pattern:

```
<random seed>
<length> <query length>
<height> <p>
<elements>
<query elements>
```

- `random seed` is the number used to initialise the random number generation.
- `length` is the number of elements to be read in.
- `query length` is the number of elements which will be searched.
- `height` is the maximum height of the leap list.
- `p` is the probability of a particular element found on one level to be found on the next.
- `elements` is a space separated list of numbers comprising the leap list.
- `query elements` is a space separated list of numbers comprising the keys which will be searched for.

Use of Random

In order to have a single expected output, you should evaluate `(double) rand() / RAND_MAX < p` as the condition (rather than checking if the random value is *above* a particular threshold).

⚠ Make sure to only call `rand` as many times as needed, if you're already at the maximum possible level, don't check to see if the level should be increased further.

Output

The output from the program should be one line per query element, matching the following pattern:

```
<query element> (<found>): <base accesses> <required accesses>
```

- `query element` is the element being searched for.
- `found` is whether the element was found ("present") or not ("not found").
- `base accesses` is the position of the element being searched for in the base level of the express list - treating the roots of the express list as the 0th element. This could also be thought of as the number of pointer accesses which would be required to reach the element in a regular linked list (including the initial head pointer access).
- `required accesses` is the number of pointer accesses required to find the element in the express list.

For the purposes of counting required accesses consistently, `NULL` pointers will not add a pointer access. Additionally, where the same pointer is present in multiple levels , the pointer only needs to be counted *once*. This is because we don't need to follow the pointer in either case to perform a comparison, we already know its result.

As we don't perform any other operations with pointer accesses, the number of required pointer accesses will match the number of required key comparisons exactly, so if this is easier to understand you may equivalently consider this as the basic operation.

Part B

The second part of the task introduces deletion. This task has an additional step after creating the leap list where a set of elements are deleted from the list before the queries occur.

Input

The input to the program is delivered on stdin, it follows the following pattern:

```
<random seed>
<length> <query length>
<height> <p>
<elements>
<query elements>
<delete length>
<delete elements>
```

- `delete length` is the number of elements which will be deleted from the list.
- `delete elements` is a space separated list of numbers comprising the keys which will be deleted.

The remaining components of the input are the same as in the previous part.

Output - Part B

The output in this part will differ. It comprises two components, the first is a list of keys occurring at each level, the second is the same as the first part, a set of results of the queries performed on the

list.

An example of the format for the first component is:

```
3
1 3 5
1 2 3 4 5
```

where the keys 1, 2, 3, 4 and 5 have been inserted at levels 2, 1, 3, 1 and 2, respectively.

The second component is the same format as Part A.

Problem 1 - Theoretical Analysis

In this task you are asked analyse the probabilistic nature of leap lists. You have seen already this kind of analysis in the lectures for linear search.

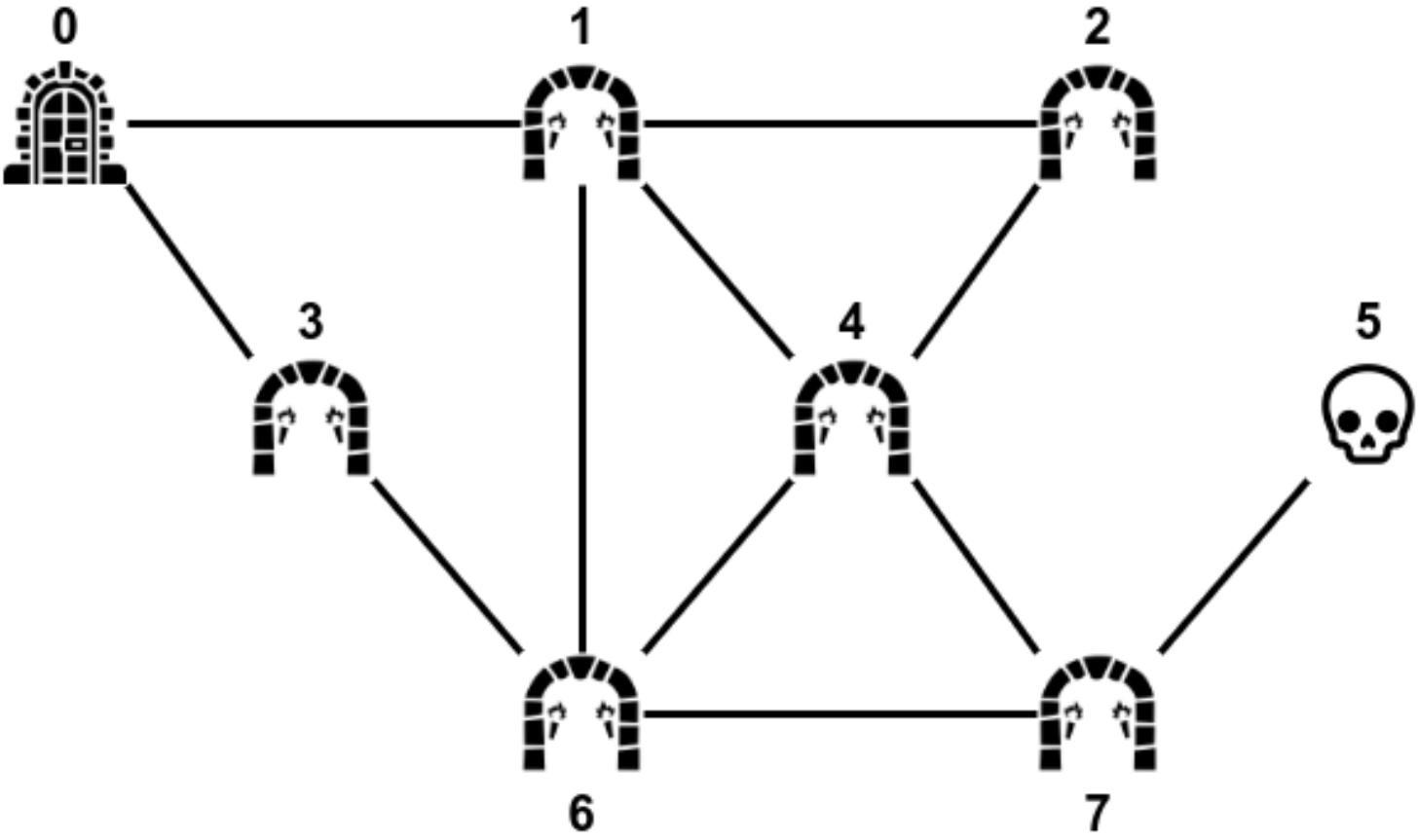
To simplify the discussion, assume that the probability for inserting an element to a higher level is always $p = 1/2$. You could visualise the decision by performing a coin flip: if the outcome is H (for head), we insert an element from a lower level to the next higher level; otherwise, i.e., the outcome is T (for tail), we do not insert the element.

1. Discuss how many elements do we expect *on average* on level i if the base level 1 has n elements. Hint: start by finding out how many elements the list at height 2 will have on average. Then you need to generalise this step. This will lead to a formula to derive the *average space* efficiency of leap lists (if we consider the storage costs across all levels). Your final result should use Big-Oh notation.
2. What is the relationship between the number of elements n in the base level and the total number of lists in a leap lists *on average* if we assume that there is no upper limit on the total levels in a leap list? Justify your answer (a proof is not necessary).
3. Use the result in step 2 (regarding the average number of lists in a leap lists) to discuss the *average time efficiency* when searching for a key. You may assume that the arrangements of elements across all levels reflects an *average case*.
4. Provide an argument for the *worst case efficiency* of `findKey` in leap lists given the number of elements n and the height h.

Problem 2 - The Legend of Zegend

The Legend of Zegend is a new dungeon exploration game that reached massive success. The protagonist, Lonk, has to explore dungeons and fight bosses at the end of each dungeon. The game however, was designed for experienced players who enjoy a challenge. In the Legend of Zegend, every time Lonk moves to new room in a dungeon, they lose 1 health (measured in hearts). Therefore, the best strategy is to reach the boss room with as much health as possible.

Here is an example of a dungeon, represented as a graph:



The entrance room is represented as a door (node "0" above) and the boss room as a skull (node "5" above). Connected rooms are represented as edges: the goal is to reach the boss room while losing the minimum amount of hearts.

Part A

Your task is to write a program in C that, given a dungeon, finds the route that reaches the boss room with maximum health possible. Remember that Lonk loses one heart every time they move to a new

room. The output of your program should be how many hearts you lose in total.

For example, the dungeon shown above would be represented in the following format:

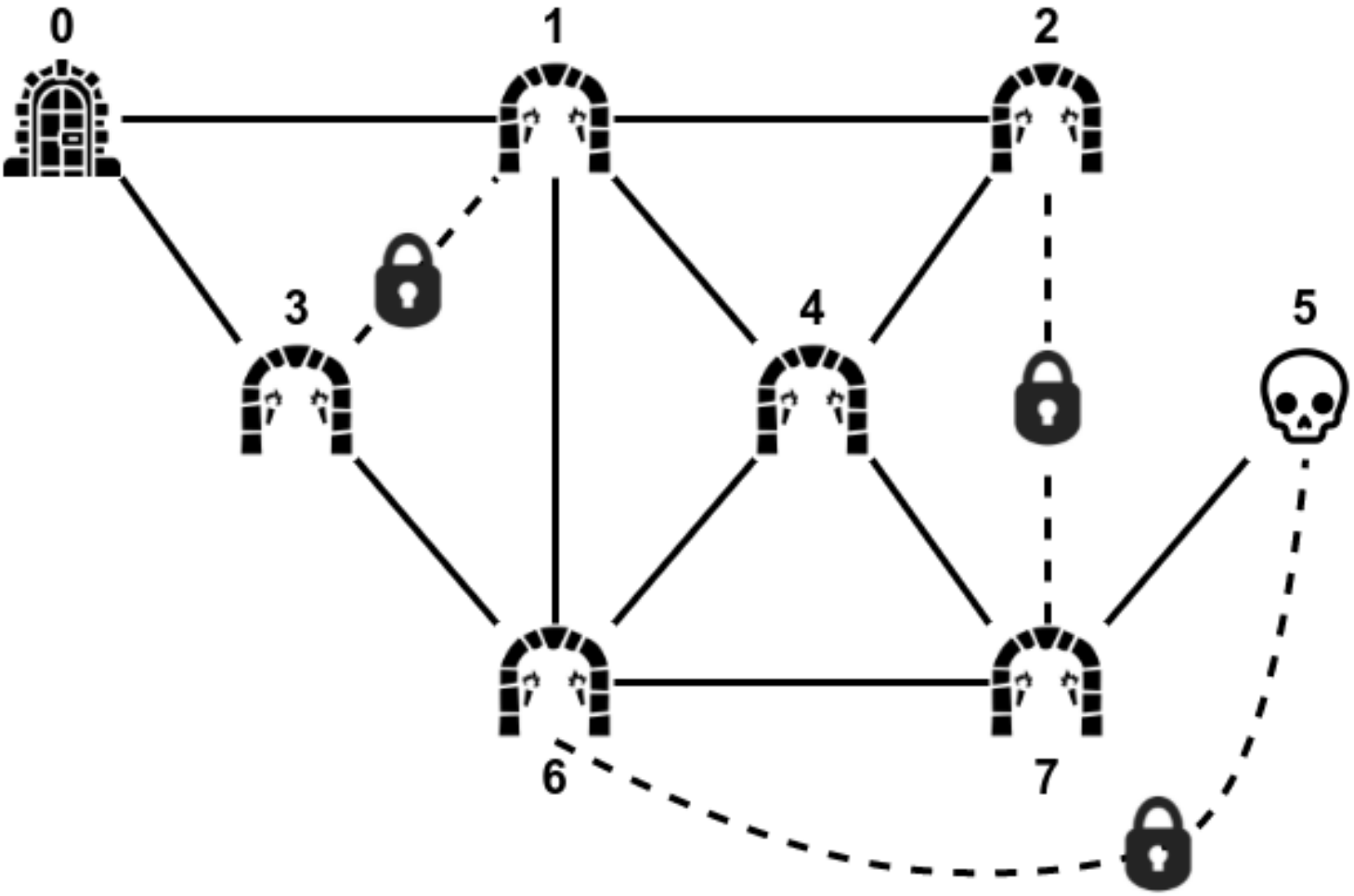
```
8
11
0
5
0 1
0 3
1 2
1 4
1 6
2 4
3 6
4 6
4 7
5 7
6 7
```

The first line shows how many rooms the dungeon has: 8
The second line shows how many connections between rooms there are in the dungeon: 11
The third line shows the starting room: 0
The fourth line shows the boss room: 5
The remaining lines represent all connections between rooms.

In this example, the output should be "4", as it is the minimum amount of hearts Lonk has to lose to reach the boss room from the starting room.

Part B

Now the game has a set of dungeons with shortcuts that connect rooms, but each shortcut requires a key. Here is an example:



This dungeon would be represented using the following format:

```
8
11
0
5
0 1
0 3
1 2
1 4
1 6
2 4
3 6
4 6
4 7
5 7
6 7
3
1 3
2 7
5 6
```

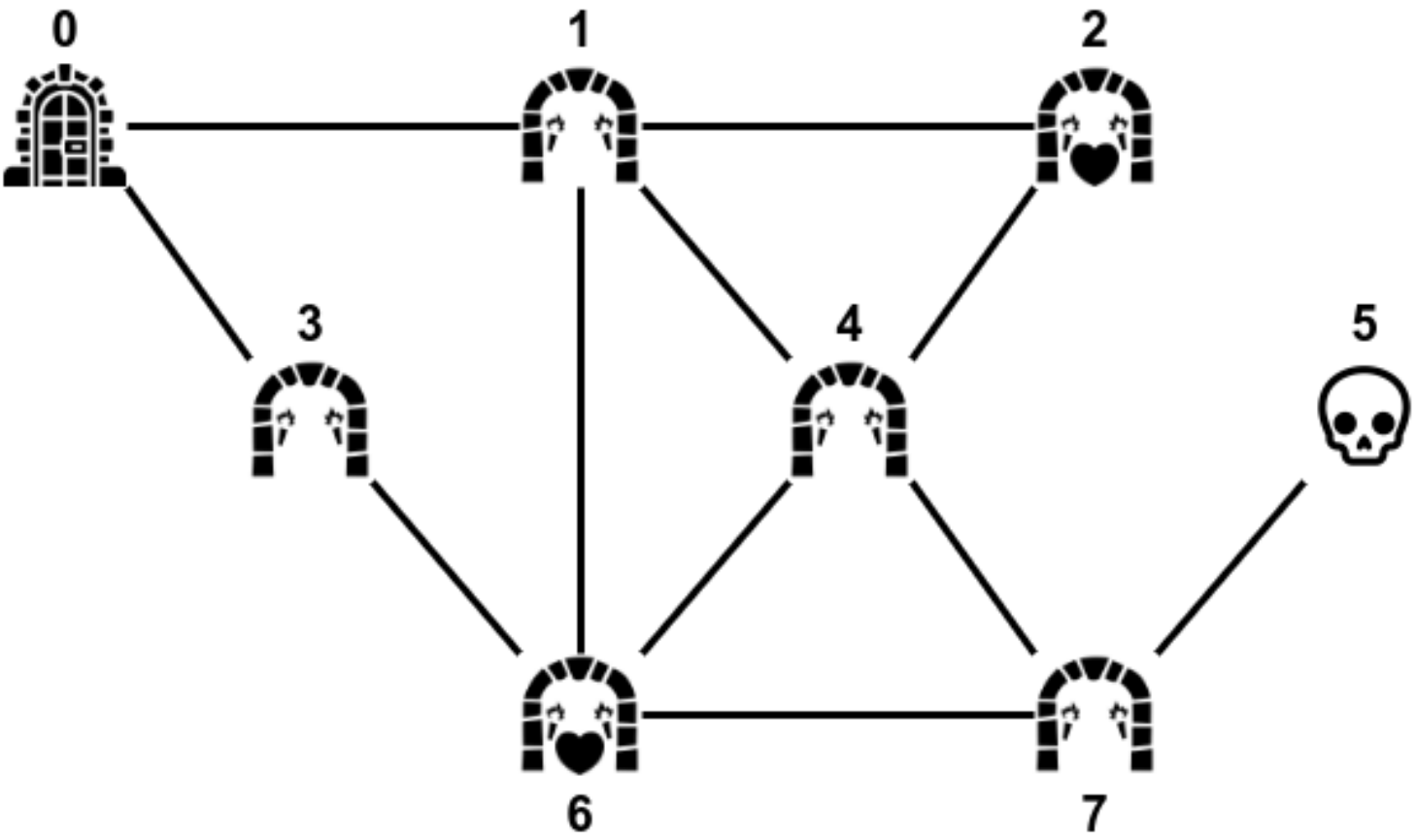
Which is the same as Part A's dungeon format, but with added information after the list of connections between rooms. The first line after the list of connections shows how many shortcuts the dungeon has: 3. The remaining lines after that shows the shortcuts.

Lonk enters the dungeon with **a single key**, which they can use to open any shortcut. Your task is to write a program in C that, given a dungeon, finds the route that reaches the boss room with maximum health possible. This should take into account any potential shortcuts, but using the key **is optional**. The output of your program should be how many hearts you lose in total.

In this example, the output should be "3", as it is the minimum amount of hearts Lonk has to lose to reach the boss room from the starting room. It is one less than Part A's dungeon because Lonk can open the shortcut between rooms 6 and 5.

Part C

In this last set of dungeons, some rooms have hearts that Lonk can use to replenish their health. Here is an example:



In this case, rooms 2 and 6 are "heart rooms". When Lonk enters a heart room, their health is replenished by 1 heart. Notice they still lose 1 heart from traversing rooms though. This dungeon would be represented using the following format:

```
8
11
0
5
0 1
0 3
```

```
1 2
1 4
1 6
2 4
3 6
4 6
4 7
5 7
6 7
2
2
6
```

Which is the same as Part A's dungeon format, but with added information after the list of connections between rooms. The first line after the list of connections shows how many heart rooms the dungeon has: 2. The remaining lines after that gives the IDs for the heart rooms.

Your task is to write a program in C that, given a dungeon, finds the route that reaches the boss room with maximum health possible. This should take into account any heart rooms Lonk might find. **Important:** you can assume two heart rooms are never connected to each other. The output of your program should be how many hearts you lose in total, **minus how many hearts were obtained through any potential heart rooms in the way**.

In this example, the output should be "3", as it is the minimum amount of hearts Lonk has to lose to reach the boss room from the starting room. It is one less than Part A's dungeon because Lonk can collect the heart in room 6.

Problem 2 - Extra Seen Test Cases

Requirements

The following implementation requirements must be adhered to:

- You must write your implementation in the C programming language.
- You must write your code in a modular way, so that your implementation could be used in another program without extensive rewriting or copying. This means that the leap list operations are kept together in a separate .c file, with its own header (.h) file, separate from the main program.
- Your code should be easily extensible to different leap lists. This means that the functions for modifying parts of your leap list should take as arguments not only the values required to perform the operation required, but also a pointer to the leap list.
- **Your implementation must read the input file once only.**
- Your program should store strings in a space-efficient manner. If you are using malloc() to create the space for a string, remember to allow space for the final end of string '\0' (NULL).
- A full Makefile is not provided for you - the Makefile provided with the scaffold will need to be modified if you make changes to the structure of the program. The Makefile should direct the compilation of your program. To use the Makefile, make sure it is in the same directory as your code, and type `make problem1a`, `make problem1b` , `make problem1c` for the first programming task and `make problem2a`, `make problem2b` and `make problem2c` for the second task. You must submit your Makefile with your assignment.

Hint: If you haven't used make before, try it on simple programs first. If it doesn't work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces. It is not a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

Programming Style

Below is a style guide which assignments are evaluated against. For this subject, the 80 character limit is a guideline rather than a rule - if your code exceeds this limit, you should consider whether your code would be more readable if you instead rearranged it.

```
/** ***********************
* C Programming Style for Engineering Computation
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
* Definitions and includes
* Definitions are in UPPER_CASE
* Includes go before definitions
* Space between includes, definitions and the main function.
* Use definitions for any constants in your program, do not just write them
* in.
*
* Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
* Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
* style. Both are very standard.
*/

/**
* GOOD:
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_STRING_SIZE 1000
#define DEBUG 0
int main(int argc, char **argv) {
 ...

/**
* BAD:
*/

/* Definitions and includes are mixed up */
#include <stdlib.h>
#define MAX_STING_SIZE 1000
/* Definitions are given names like variables */
#define debug 0
#include <stdio.h>
/* No spacing between includes, definitions and main function*/
int main(int argc, char **argv) {
 ...

/** ****************************
* Variables
* Give them useful lower_case names or camelCase. Either is fine,
```

```
* as long as you are consistent and apply always the same style.
* Initialise them to something that makes sense.
*/

/**
* GOOD: lower_case
*/

int main(int argc, char **argv) {

  int i = 0;
  int num_fifties = 0;
  int num_twenties = 0;
  int num_tens = 0;

 ...
/**
* GOOD: camelCase
*/

int main(int argc, char **argv) {

  int i = 0;
  int numFifties = 0;
  int numTwenties = 0;
  int numTens = 0;

 ...
/**
* BAD:
*/

int main(int argc, char **argv) {

  /* Variable not initialised - causes a bug because we didn't remember to
  * set it before the loop */
  int i;
  /* Variable in all caps - we'll get confused between this and constants
  */
  int NUM_FIFTIES = 0;
  /* Overly abbreviated variable names make things hard. */
  int nt = 0

  while (i < 10) {
    ...
    i++;
  }

 ...

/** ********************
* Spacing:
* Space intelligently, vertically to group blocks of code that are doing a
* specific operation, or to separate variable declarations from other code.
```

```
* One tab of indentation within either a function or a loop.
* Spaces after commas.
* Space between ) and {.
* No space between the ** and the argv in the definition of the main
* function.
* When declaring a pointer variable or argument, you may place the asterisk
* adjacent to either the type or to the variable name.
* Lines at most 80 characters long.
* Closing brace goes on its own line
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

 int i = 0;

 for(i = 100; i >= 0; i--) {
   if (i > 0) {
     printf("%d bottles of beer, take one down and pass it around,"
     " %d bottles of beer.\n", i, i - 1);
   } else {
     printf("%d bottles of beer, take one down and pass it around."
     " We're empty.\n", i);
   }
 }

 return 0;
}

/**
* BAD:
*/

/* No space after commas
* Space between the ** and argv in the main function definition
* No space between the ) and { at the start of a function */
int main(int argc,char ** argv){
 int i = 0;
 /* No space between variable declarations and the rest of the function.
 * No spaces around the boolean operators */
 for(i=100;i>=0;i--) {
 /* No indentation */
 if (i > 0) {
 /* Line too long */
 printf("%d bottles of beer, take one down and pass it around, %d
bottles of beer.\n", i, i - 1);
 } else {
 /* Spacing for no good reason. */

 printf("%d bottles of beer, take one down and pass it around."
 " We're empty.\n", i);
```

```
 }
 }
 /* Closing brace not on its own line */
 return 0;}

/** ****************
* Braces:
* Opening braces go on the same line as the loop or function name
* Closing braces go on their own line
* Closing braces go at the same indentation level as the thing they are
* closing
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

  ...

  for(...) {
    ...
  }

  return 0;
}

/**
* BAD:
*/

int main(int argc, char **argv) {

  ...

  /* Opening brace on a different line to the for loop open */
  for(...)
  {
    ...
    /* Closing brace at a different indentation to the thing it's
    closing
    */
           }

 /* Closing brace not on its own line. */
       return 0;}

/** **************
* Commenting:
* Each program should have a comment explaining what it does and who created
* it.
* Also comment how to run the program, including optional command line
```

```
* parameters.
* Any interesting code should have a comment to explain itself.
* We should not comment obvious things - write code that documents itself
*/

/**
* GOOD:
*/

/* change.c
*
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
13/03/2011
*
* Print the number of each coin that would be needed to make up some
change
* that is input by the user
*
* To run the program type:
* ./coins --num_coins 5 --shape_coins trapezoid --output blabla.txt
*
* To see all the input parameters, type:
* ./coins --help
* Options::
*   --help                Show help message
*   --num_coins arg       Input number of coins
*   --shape_coins arg     Input coins shape
*   --bound arg (=1)      Max bound on xxx, default value 1
*   --output arg          Output solution file
*
*/

int main(int argc, char **argv) {

  int input_change = 0;

  printf("Please input the value of the change (0-99 cents
  inclusive):\n");
  scanf("%d", &input_change);
  printf("\n");

  // Valid change values are 0-99 inclusive.
  if(input_change < 0 || input_change > 99) {
    printf("Input not in the range 0-99.\n")
  }

  ...

/**
* BAD:
*/

/* No explanation of what the program is doing */
int main(int argc, char **argv) {
```

```
  /* Commenting obvious things */
  /* Create a int variable called input_change to store the input from
  the
  * user. */
  int input_change;

  ...

/** ****************
* Code structure:
* Fail fast - input checks should happen first, then do the computation.
* Structure the code so that all error handling happens in an easy to read
* location
*/

/**
* GOOD:
*/
if (input_is_bad) {
  printf("Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}

/* Do computations here */
...

/**
* BAD:
*/

if (input_is_good) {
  /* lots of computation here, pushing the else part off the screen.
  */
  ...
} else {
  fprintf(stderr, "Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}
```

Some automatic evaluations of your code style may be performed where they are reliable. As determining whether these style-related issues are occurring sometimes involves non-trivial (and sometimes even undecidable) calculations, a simpler and more error-prone (but highly successful) solution is used. You may need to add a comment to identify these cases, so check any failing test outputs for instructions on how to resolve incorrectly flagged issues.

Additional support

Your tutors will be available to help with your assignment during the scheduled workshop times. Questions related to the assignment may be posted on the Ed discussion forum, using the folder tag Assignments for new posts. You should feel free to answer other students' questions if you are confident of your skills.

A tutor will check the discussion forum regularly, and answer some questions, but be aware that for some questions you will just need to use your judgment and document your thinking. For example, a question asking whether your answer is correct will not be answered; you must perform tests and see whether your solution is appropriate for the problem.

If you have questions about your code specifically which you feel would reveal too much of the assignment, feel free to post a private question on the discussion forum.

Submission

Your C code files (including your Makefile and any other files needed to run your code) should be submitted through Ed to this assignment. Your programs must compile and run correctly on Ed. You may have developed your program in another environment, but it still must run on Ed at submission time. For this reason, and because there are often small, but significant, differences between compilers, it is suggested that if you are working in a different environment, you upload and test your code on Ed at reasonably frequent intervals.

A common reason for programs not to compile is that a file has been inadvertently omitted from the submission. Please check your submission, and resubmit all files if necessary.

The theoretical analysis for Problem 1 should be submitted with your other files in the *Problem 1 - Programming Leap Lists* slide and should be submitted by filling in the text file `written-problem.txt` or as a PDF if more complex figures are required.

Assessment

There are a total of 10 marks given for this assignment.

Your C program will be marked on the basis of accuracy, readability, and good C programming structure, safety and style, including documentation (2 marks). Safety refers to checking whether opening a file returns something, whether mallocs do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting, by choosing descriptive variable names. The remainder of the marks will be based on the correct functioning of your submission.

Note that these correct functioning-related marks will be based on passing various tests. If your program passes these tests without addressing the learning outcomes (e.g. if you fully hard-code solutions or otherwise deliberately exploit the test cases), you may receive less marks than is suggested but your marks will otherwise be determined by test cases.

Mark Breakdown

Problem	Mark Distribution
Problem 1	2 Marks (Part A) + 2 Marks (Part B) + 2 Marks (Written Task)
Problem 2	2 Marks (Part A) + 1 Mark (Part B) + 1 Mark (Part C)

Marks will be given based on test case passing, a maximum of one mark may be deducted across the programming exercises for serious code style or structural errors. Note that not all marks will represent the same amount of work, you may find some marks are easier to obtain than others. Note also that the test cases are similarly not sorted in order of difficulty, some may be easier to pass than others.

Note that code style will be manually marked in order to provide you with the most meaningful feedback for the second assignment.

Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

“Borrowing” of someone else’s code without acknowledgment is plagiarism. **Plagiarism is considered a serious offence at the University of Melbourne.** You should read the University code on Academic integrity and details on plagiarism. Make sure you are not plagiarising, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) in the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Late policy

The late penalty is 20% of the available marks for that project for each day (or part thereof) overdue. Requests for extensions on medical grounds will need to be supported by a medical certificate. Any request received less than 48 hours before the assessment date (or after the date!) will generally not be accepted except in the most extreme circumstances. In general, extensions will not be granted if the interruption covers less than 10% of the project duration. Remember that faculty servers are often heavily loaded near project deadlines, and unexpected outages can occur; these will not be considered as grounds for an extension.

Students who experience difficulties due to personal circumstances are encouraged to make use of the appropriate University student support services, and to contact the lecturer, at the earliest opportunity.

Finally, we are here to help! There is information about getting help in this subject on the LMS. Frequently asked questions about the project will be answered on Ed.