# Ex8: Hash Functions & Data Structures

## Submission deadline - Thursday, 26.12.2013, 20:55

## Objectives

This purpose of this exercise is to practice the use and implementation of dynamic data structures - simulating hash functions and playing with linked lists.

## Exercise Description

### Part 1 - hash functions

In this part you will run simulations of insertion of data to a hash table using different hash functions. Here is a description of the stages in the simulations (see **simulations.py**):

1. Construct a list with N elements:
    - We supply you with the function **id_generator** that generates a list of random string sequences.
    - You are also expected to generate a random list of integers.
2. Insert the elements to a hash table one by one (see the function **simulation** and **hash_table.py**).
    - Define the hash table size (M). Try using a non-prime M and a prime M (using the next prime function). Do you see any difference?
    - Use a user defined hash function (F) to determine the index of the hash table where an item is inserted.
    - Repeat the simulation at least ten times to estimate the average case behavior of the hash function.
3. **simulation** returns a pair of values:
    - A tuple with the following values:
        - The expected maximum number of collisions in the hash table (the number of items that were mapped to the most loaded index).
        - The expected average number of collisions in the hash table.
        - The expected standard deviation of collisions in the hash table.
        - The expected normalized variance of collisions in the hash table (when the total weight of the table load is 1, allows comparison between different simulations).
    - The expected time in seconds it takes to insert the whole list of items to the hash table.

We supply you with eight different hash functions, some were learned in class, some were not (see **hash_functions.py**). Some are very simple and naive, some are a bit more complicated.

Your task is to run simulations with the **simulation** function code, with various hash functions and values of M, N. Try different sizes of N and different ratios between M and N (M>>N, M<<N, M=N), each with one of the eight hash functions.

To assist you with your simulations the **simulations.py** file can be executed with the following parameters:

```
usage: simulations.py [-h] [--repeats REPEATS] [--minhashsize MINHASHSIZE]
                      [--idlistsize IDLISTSIZE] [--toprime]
                      [--function FUNCTION] [--sourcefile SOURCEFILE]
                      [--csvoutput]
```

-r: number of repeats
-m: minimal hash table size (M)
-n: list size (N)
-p: if assigned M will be the smallest prime number that is bigger than the given M
-f: number of hash function to use
-s: use a source file for the parameters
-c: prints the results in a more convenient form for analysis.

**Report your results and conclusions:**

- How does each function map items to integers?
- Which function is the best? for what type of data?
- What is the impact of different ratios of M and N on the effectiveness of the functions?
- Why should the hash table size be a prime number? Can you see difference when using a prime number?
- Explain the advantages and drawbacks of each of the functions.

## Part 2 - algorithms on linked lists

In this part you will implement a few operations on singly linked lists.

We provide you with a class **List** (source code, see **sllist.py**) that implements a singly linked list (each node only has a reference to the next node in the list) as defined by its API and the **Node** API. The list can contain any comparable data type.

Your task is to implement the functions in the **sllist_util.py** file according to its API and the description below.

You are not allowed to use ANY python built-in containers, iterators or related functions, or import any module except for the classes List and Node. The goal of this assignment is for you to practice the internal implementations of the built-in functions and containers. The only header to your file should be:

```
from list import List, Node
```

Note that when we refer in this exercise to a list in the functions description we mean an instance of our class **List** and not the python container with that name.

Implement the following functions in the file **sllist_util.py**:

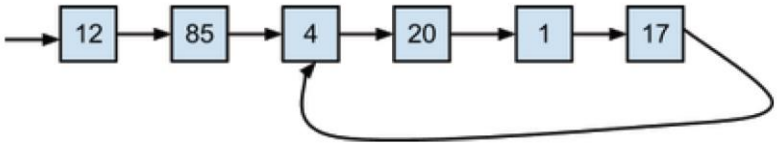
1. **is_palindrome(sll)** - detects if a list is a palindrome. A list is a palindrome if for j=1…n/2 (where n is the number of elements in the list) the item in location j equals

to the item in location n-j-1. Note that you should compare the values stored in the nodes and not the node objects themselves.

2. **contains_cycle(sll)** - takes a list and returns true if the list contains a cycle. A list contains a cycle if at some point a Node in the list points to a Node that previously appeared in the list.

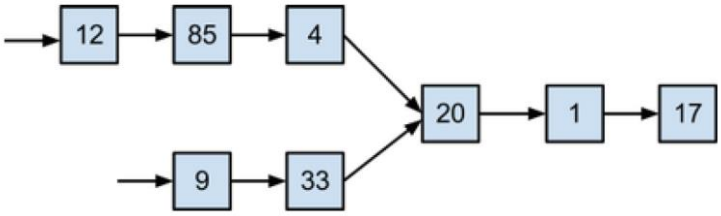
Hint: This problem is very popular in job interviews. You may find an idea for a solution by searching the internet.

For example, this method should return true on the following list:




Note that the cycle does not necessarily contain all the nodes in the list.

3. **have_intersection(first_list, second_list)** - checks if two lists intersect. Two lists intersect if at some point they start to share nodes. Once two lists intersect, they become a single list from that point on and can no longer split apart. For example, the two lists in the next example intersect:




Note that two lists might intersect even if their lengths are not equal, just as in the example.

4. **reverse(sll)** – reverses the given list (so the head becomes the last item, and an item *j* points to an item *i* if in the original list *i* was pointing to *j*). The function returns None; it modifies the original list to become the reversed list.
5. **get_item(sll,k)** - returns the k'th item of the list. If k is negative returns the k'th item from the end of the list (same behavior as using list[k] in python lists).
6. **slice(sll, start, stop, step)** – returns a new list after slicing out items according to the given arguments (same as using list[start:step:stop] in python lists). For example:
   sll = {10,11,12,13,14,15,16,17,18,19,20}
   sliced_sll = slice(sll, 4, 8, 2)
   The new list = {14,16}
7. **merge_lists(first_list, second_list)** - takes two sorted lists (in ascending order) and returns a new sorted list that merges the two input lists.
8. **merge_sort(sll)** - sorts the list into ascending order using the merge-sort algorithm learnt in class. Make sure that you use the merge-sort algorithm and not any other sorting algorithm.

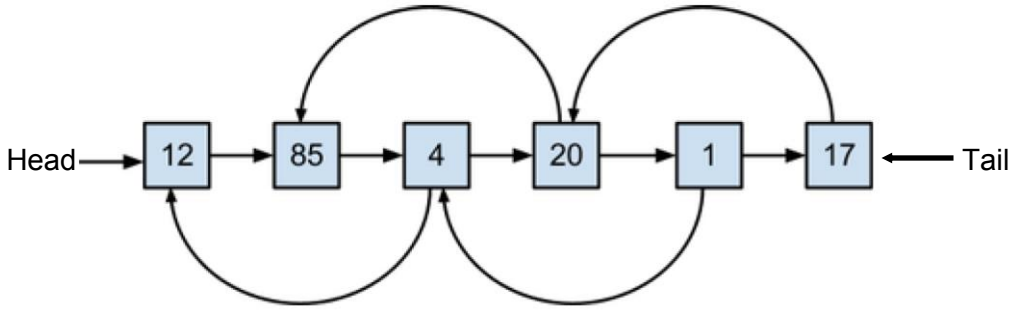
**Important notes:**

- Your implementations should be efficient. Non-efficient implementations will not grant full grade.

- You may assume the list contains no cycles in all functions except the **contains_cycle()** function.
- In functions 1, 2, 3 and 5 you should not change the values stored in any of the nodes or replace any node in the list.
- In the functions **reverse()** and **merge_sort ()** you should modify the original list.
- In the functions **slice()** and **merge_lists ()** you should create a new list.
- In all functions you should use extra O(1) space (meaning that you are not allowed to create more than a fixed number of new nodes), excluding new nodes which will be part of a new list returned by the function.
- You should not submit the file **sllist.py**, and thus you may not make any changes to the file.
- You must explain the complexity of your implementation for each of the above functions in a **README** file. A non-efficient implementation will not get a full grade.

### Part 3 - Implementing a special linked list

In this part, you will implement a special linked list, called a **SkipiList**. **SkipiList** is a special variant of a doubly-linked list, where each Node has a reference to the next Node in the list, and another reference to the Node two positions before (prev-prev) in the list, which is called the **skip_back** reference (hence the prefix "skipi"). In addition, a **SkipiList** has references both to the head of the list and to the tail of the list. Initially, the list is created empty with both of these references set to None. The tail node's next pointer is always None (in the figure below, note that there is no right arrow from the node whose data is 17). The **skip_back** references of the first two nodes in the list are also always None (in the figure below, note that there are no left curved arrows from the nodes whose data are 12 and 85).




The file **skipi_list.py** contains starter code and docstrings for each of the functions you are required to implement.

Again, you are not allowed to use ANY python built-in containers, iterators or related functions, or import any module except for the class SkipiNode

```
from sllist import SkipiNode as Node
```

**Answer in the README file:**

How will your complexity analysis from **part 2** change if instead of using a singly linked list you will use a Skipi List?

## Submission and Further Guidelines

All the files can be downloaded from here.

**README**

Explain in the usage part how to run the project and about the different options.

**Creating a tar file**

You should have edited the files:

- **sllist_utils.py**
- **skipi_list.py**
- **README** (as explained in the course guidelines)**.** In addition to regular explanations, your file should contain:
    - Report for part 1.
    - Complexity analysis for part 2.
    - Complexity analysis for part 2 using SkipiList (from part 3).

Create a TAR file named ex8.tar containing only the above files by invoking the shell command:

```
tar cvf ex8.tar slist_utils.py skipi_list.py README
```

The TAR file should contain only these files!

**Submitting the tar file**

- You should submit the file ex8.tar via the link on the course home page, under the ex8 link.
- Note that submission requires you to be registered as a student and logged in.
- Few minutes after you upload the file you should receive an email to your university mailbox. That email contains the test results of your submission and the PDF file that was passed to the grader.
- Read the submission file again and make sure that your files appear and fully readable.

**Running the tests yourself**

There are two ways you can run these testers and see the results without submitting your exercise and getting the PDF:

The first – place your tar file in an empty directory, go to that directory using the shell and type: ~intro2cs/bin/testers/ex8 ex8.tar

The second – download a file - ex8testing.tar.bz2. Extract the contents (using tar –xjv ex8testing.tar.bz2) of the file and follow the instructions on the file named TESTING.

***Congratulations!*** **One more Intro2cs project completed.**