Part IA Algorithms
Further problems on lists, stacks and queues

In this document you will find several problems using lists, stacks and queues. Most of these problems are quite well-known and you will find solutions for them online. There are also several short projects in case you want to go more in depth into some interesting algorithms (most of these are not in the tripos and will probably not come up in an interview - so I am happy to discuss during the breaks after the course is over).

1 Linked Lists

Core tasks

Exercise 1 [Singly linked list] Describe how you would implement the following operations for a singly linked list.

(a) `insertAfter(n, x)`: insert node x after node n. How efficiently can you insert at the beginning or insert at the end? Why is it important to handle these cases?

(b) `deleteAfter(n)`: delete an element after node n. How efficiently can you delete at the beginning or delete at the end? Why is it important to handle these cases?

(c) `kth(n, k)`: determine the k-th element after n.

(d) `merge(l1, l2)`: merge two lists ℓ_1 and ℓ_2. How can you do this in $\mathcal{O}(1)$ time?

(e) `insertListAfter(n, l)`: insert the linked list ℓ after n

Exercise 2 [k-th node from the end] Design an algorithm such that given a singly linked list finds the k-th element from the end using only $\mathcal{O}(1)$ memory.

Exercise 3 [Middle node] Design an algorithm such that given a singly linked list finds the middle element using only $\mathcal{O}(1)$ memory.

Exercise 4 [Intermediate operations]

(a) Give a linear time algorithm with $\mathcal{O}(1)$ extra space for merging two sorted lists.

(b) Rotate a linked list by k positions (using $\mathcal{O}(1)$ space).

(c) Remove duplicates from a sorted list.

(d) Reverse a linked list.

(e) Split a given list into two lists one containing the elements at odd indices and one containing the elements at even indices.

Exercise 5 [Integer operations] Explain how to represent big integers using linked lists. Design an algorithm to add, subtract and multiply these integers.

Exercise 6 [Doubly linked list] Describe how you would implement the following operations for a doubly linked list.

(a) Insert an element after node n.

(b) Insert an element before node n.

(c) Remove node n.

Exercise 7 [Maintaining the k-th entry] Given a fixed k, explain how you would maintain a linked list (with insert, remove, etc) and also the operation of accessing the k-th element in $\mathcal{O}(1)$ time. (You may assume that this element will always exist).

Exercise 8 [Circular linked list] Describe how you would implement the following operations for a circular linked list.

(a) Check if a linked list is circular.

(b) Find the length of the cycle.

Exercise 9 [Array-based implementation] Explain how you can implement the three list variants (singly-linked, doubly-linked and circular) using an array.

Exercise 10 [Josephus Problem] In the Josephus problem there are N people numbered from $0, \ldots, N-1$ starting in a circle. A count starts from person 0 and the person to be counted as M is removed from the circle. In the next round the count continues from the person after the one that was removed last. For example, if $N = 5$ and $M = 2$, then $L_0 = [0; 1; 2; 3; 4]$, $L_0 = [0; 1; 3; 4]$, $L_2 = [1; 3; 4]$, $L_3 = [1; 3]$, $L_2 = [1]$.

(a) Design an algorithm that simulates this process and finds the last remaining person in the circle.

(b) How does your implementation compare with the array implementation?

(c) (+) How can you use BSTs to make your implementation more efficient?

Exercise 11 [Common nodes]

(a) Given two linked lists (which are not circular), determine if they have any common nodes using $\mathcal{O}(1)$ space.

(b) Determine the earliest common node in the list using $\mathcal{O}(1)$ space.

More challenging tasks

Exercise 12 [XOR-List] How can you store a doubly linked list using one pointer and still support the common operations?

Exercise 13 Attempt **UVA 11988**.

Exercise 14 [Linked list with a loop] Describe how you would implement the following operations for a circular linked list. *Hint:* You may want to look at this algorithm.

(a) How does a circular linked list differ from a linked list with a loop?

(b) Design an algorithm to determine if a linked list has a loop.

(c) (+) Ensure that your algorithm has $\mathcal{O}(N)$ time and $\mathcal{O}(1)$ space complexity.

(d) (+) Compute the length of the shortest cycle.

Exercise 15 [Multi-level list]

(a) (+) In this exercise, we have a fixed list and we will be adding some extra arrows in order to access the k-th element more efficiently.

(b) Assume that we keep at the i-node two pointers: one to the next element and one to the element that is T_1 places away. Show that the time to find any element given its index is $\mathcal{O}(T_1 + N/T_1)$.

(c) Find the value of T_1 that minimises the function above.

(d) Design an algorithm to efficiently compute the T_1 pointers. What is the time and space complexity of your implementation?

(e) How you change your implementation if you had three pointers per node?

Further Reading 1 [List unrolling] Linked Lists require one pointer per element and also are not very cache friendly because elements may not be stored in nearby positions. If you do not expect the linked list to be constantly modified, a technique known as *list unrolling* can be applied, where each node of the linked list keeps more than one elements in a array. Read more about this in the "Unrolling Lists" paper by Z. Shao. (You may also want to check about CDR-coding)

Project 1 [Dancing links] Dancing links is an interesting technique that can be used to speed up exhaustive search computations (similar to the problems you look in the Foundations of Computer Science course). You can learn more about this in this lecture by D. Knuth or through this and this articles.
Choose your favourite complete enumeration problem from the Foundations of Computer Science course (n-queens, sudoku, knight tours) and implement it using dancing links. Do you notice an improvement in speed?

Further Reading 2 [Streaming algorithms] Assume that we have a very large collection of items that does not fit in main memory and we want to determine process them. One possible way of abstracting this problem is as having a stream of elements on which we can call `getNext()` plus some limited RAM memory that we can use for processing. The goal in these kinds of tasks is to solve the problem using only a few (usually constant number of) iterations over the input stream. If this is not possible, then we sometimes search for an approximation to the solution.

(a) One such task which we saw during supervisions is the Necklace task. Can you think how you would solve this given access to two streams of A?

(b) You can read more about streaming algorithms in these lecture notes.

Project 2 [Skip Lists] (+) Skip Lists is a data structure that extends the idea of keeping more than one links per node in the list. They are very simple to code and they can be used as powerful BSTs. You can read more about them in this chapter. In order to prove their guarantees you need to know randomised analysis techniques which you have not covered in the tripos, so the following questions mostly guide you through the implementation of the operations:

(a) Explain how the height of an element is chosen.

(b) (optional) What is the expected height of an element? What can you concluded about the expected number of nodes in the tree?

(c) Explain how the insert operation works.

(d) Explain how the remove operation works.

(e) Gather together the previous operations to implement a BST.

(f) Explain how you can modify your BST, so that you maintain a dynamic array which supports `merge` (merge two dynamic arrays) and `splitAt(i)` (split a dynamic array at index i).

(g) Explain how you would use `merge` and `splitAt` to implement `insertAfter(i)` and `remove(i)`.

(h) (+ optional) How can you modify your dynamic array to allow adding a value to all entries in the interval $[i, j]$. Your implementation should have the same asymptotic running time as `merge` and `split`.

2 Stacks

Core tasks

Exercise 16 [Implementation using a linked list]

(a) Explain how you would use a linked list to implement a stack with the standard `push`, `pop` and `is_empty` operations.

(b) What is the time complexity of these operations?

(c) Explain how you could `merge` two stacks.

Exercise 17 [Stack with min]

(a) Implement a stack that supports a `getMin()` operation in $\mathcal{O}(1)$ time.

(b) Why is implementation of the stack in this article, not using $\mathcal{O}(1)$ memory?

Exercise 18 [Implementation using an array]

(a) Explain how you would use a fixed size array to implement a stack (of bounded size).

(b) What if you are not given that the stack size is bounded?

(c) How could you store two stacks in a single array? Where is this used in practice?

Exercise 19 [Converting recursive code to iterative] One of the main application of stacks is in making code iterative (and also implementing the call stack of recursive programs).

(a) Explain how you can convert any recursive function into an iterative using a stack.

(b) What is the connection between your conversion and making code tail recursive?

(c) Implement a DFS algorithm using a stack. For example, you can try solving sudoku, a search problem on a graph, etc.

More challenging tasks

Exercise 20 [Implementation using other data structures]

(a) Explain how you would implement the core operations of a stack using several queues? State the time complexity for each operation.

(b) Explain how you would implement the core operations of a stack using a single queue? State the time complexity for each operation.

(c) Explain how you would implement the core operations of a stack using a heap. State the time complexity for each operation.

Exercise 21 [Balanced parentheses] A string s of parentheses is called *balanced* if $s = \epsilon$, $s = s_1 s_2$ or $s = (s_1)$ where s_1, s_2 are balanced strings.

(a) Design a linear time algorithm that given a string determines if it is balanced or not. State the time complexity of your algorithm.

(b) Modify your algorithm to compute for each opening parenthesis the index of the closing parenthesis.

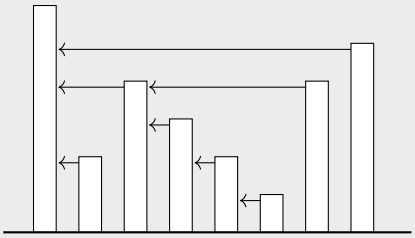
(c) Determine the maximum depth of nested parentheses in a balanced string. For example, in $()((()(())))$, the depth is 4, since $()(_1(_2()(_3(_4)_4)_3)_2)_1$.

Exercise 22 [More parentheses]

(a) Find the longest balanced substring of a given substring. (You can also do this in $\mathcal{O}(1)$ space, see this article)

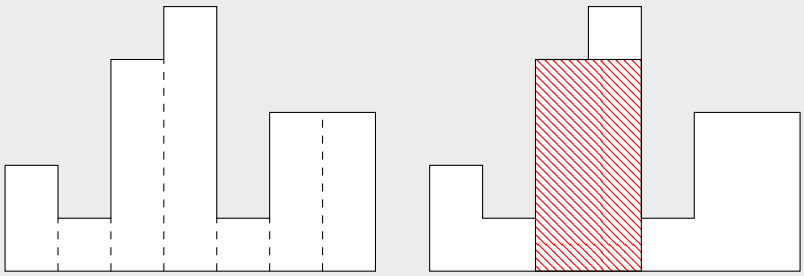
(b) Find the minimum number of parentheses that need to be flipped in order to make a string balanced or indicate that this is not possible. (see this article)

Exercise 23 [All nearest smaller values] The *all nearest smaller values* problem is given an array A, finding for each i, the largest j such that $j < i$ and $A[j] < A[i]$. The variant for larger values is easier to visualise (see figure). Write an amortised linear time algorithm for solving the task.



Exercise 24 [All nearest smaller values Problems]

(a) (**SPOJ HISTOGRA**) Given a histogram where the width of each bar is 1, find the area of the largest rectangle inside it. For example, in the figure below the largest rectangle is shaded in red.

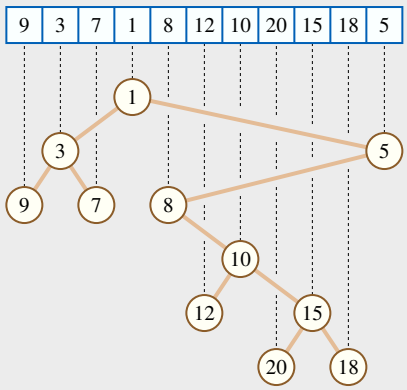


(b) (**SPOJ CTGAME**, **UVA 11871**) You are given a binary array (i.e. the entries are 0 and 1, find the area of the largest rectangle consisting of only zero entries. *Hint:* Aim for an $\mathcal{O}(nm)$ solution. There are several ways to approach it.

(c) Given an array A of distinct elements, find for each item i in how many subarrays of A is $A[i]$ the smallest element. For example, for $A = [8; 6; 10; 12; 7; 9; 2; 3]$ and element $A[4] = 7$ the answer is 6, since it is the minimum in the following subarrays $[10; 12; 7]$, $[12; 7]$, $[7]$, $[10; 12; 7; 9]$, $[12; 7; 9]$, $[7; 9]$.

(d) (+) Given an array A, find for each size w the maximum of the minimums of subarrays of length w. (See this article)
(Possibly hard) extra question: Can you design an $\mathcal{O}(n)$ comparison-based algorithm or prove that none exists? (I don't think have answer for this one)

Exercise 25 (+) The *Cartesian Tree* of an array A is the binary min-heap whose inorder traversal is that sequence. For example, for the array $A = [9; 3; 7; 1; 8; 12; 10; 20; 15; 18; 5]$, the Cartesian tree is given below



(a) Show that the Cartesian tree for an array always exists.

(b) (+) Use the *all-nearest-smaller-values* algorithm (twice) to compute the Cartesian tree in $\mathcal{O}(n)$ time.

(c) (+) Show that the Cartesian tree for a given array is unique.

Project 3 [Persistent stacks] A *persistent data structure* is one data structure where it is possible to maintain the previous versions of the data structure.

(a) Argue why all immutable data structures are persistent.

(b) Explain why lists in OCaml implement a persistent stack.

(c) You can read more about persistent data structures here.

Project 4 [Stack-sortable permutations] Use the Wikipedia article and this article to answer the following questions:

(a) Define what it means for a permutation to be *stack-sortable*.

(b) Design a linear-time algorithm to determine if a permutation is stack-sortable.

(c) Describe (and prove) the bijection between stack-sortable permutations.

Further Reading 3 [Pancake sorting] Read about the pancake sorting problem from these notes and from wikipedia (where you will also find a few related open problems).

Parsing

The stack is a core data structure to various computational models. For example, in Part IA Discrete Mathematics you learn about DFAs and NFAs. In Part IB Compilers, you will learn about DFAs with a stack. This is more powerful and can recognise the languages of a *context-free grammar*. These languages can express most of the core programming languages and hence, they are used to design parsing algorithms for these.

Exercise 26 [Parsing arithmetic expressions] An example of such a language is the language of arithmetic expressions. In this exercise, you will investigate how to parse and evaluate arithmetic expressions.

(a) Read about and define what it means for an expression to be in *infix*, *postfix* and *prefix* notation. Two examples are shown below:

Infix	Prefix	Postfix
`10/7 + 2 * 3`	`+ / 10 7 * 2 3`	`10 7 / 2 3 * +`
`5 + 4 * (3/ 2 + 7) - 3`	`+5 -* 4 +/3 2 7 3`	`5 4 3 2 / 7 + * + 3 -`

(b) Design an algorithm which given an expression in postfix notation it evaluates it. What is the time

complexity of your algorithm?

(c) Repeat the previous question with the expression given in prefix notation.

(d) (optional +) Design an algorithm to convert an expression from infix to postfix notation. *Hint:* You may want to read about the Shunting yard algorithm (or this article).

Exercise 27 (++ optional) This is a preview of what you will be dealing with in Computation Theory. Argue that if you have a DFA with two stacks, then this is equivalent to a Turing Machine (or the RAM model).

3 Queues

Core tasks

Exercise 28 [Implementation using linked lists] Describe how linked lists are used to implement the following core operations of queues: `push_back`, `pop_front` and `front`.

Exercise 29 [Implementation using an array] Describe how to implement a queue using an array.

Exercise 30 [Additional operations] Describe how to implement the following operations:

(a) `reverse`: reverse the order of the elements in the queue

(b) `sort`: sort a queue without extra space

More challenging tasks

Exercise 31 [BFS] Explain how to implement BFS using queues.

Exercise 32 Given an array A of integers and $k \in \mathbb{N}$, find the leftmost negative element in each subarray of length k. For example, if $A = [1; -2; 3; 5; 10; -1; -3; -4; 7]$ and $k = 3$, the answers are: -2 for $[1; -2; 3]$, -2 for $[-2; 3; 5]$, none for $[3; 5; 10]$, -1 for $[5; 10; -1]$, -1 for $[10; -1; -3]$, -1 for $[-1; -3; -4]$ and 7 $[-3; -4; 7]$.
Hint: Try to store the index of each element together with the element in the queue.

Project 5 [Functional queues with $\mathcal{O}(1)$ worst-case operations]

(a) Read Okasaki's "Simple and efficient purely functional queues and deques" (e.g. here) and answer the following questions:

1. What is the problem with the implementation given in the FoCS lecture notes and that this paper attempts to solve? (*Hint:* Consider using this implementation in a real-time application)
2. How does this new queue work? How important is the assumption that the languge is lazy (meaning that if you call the same function with the same arguments it gives the same result?)
3. Why is this laziness assumption problematic?

(b) Read this 4-page report "Real-time queue operations in Pure Lisp". You may also find this implementation useful.

i. Describe how the queue is implemented in this case.

ii. (optional) Write an implementation in OCaml.

(c) According to the discussion here it is an open problem to find the smallest number of stacks needed to implement a queue with $\mathcal{O}(1)$ worst-case access time. Read the discussion.

Priority Queue with Attrition (or monotonous queue)

Exercise 33 [**PQA**] Explain how to implement `min()` operation for a queue. Your algorithm should take amortised $\mathcal{O}(1)$ time and $\mathcal{O}(1)$ space.

Exercise 34 [**Problems solved using PQA**] Use PQA to solve the following problems:

(a) Given an array A of integers and $k \in \mathbb{N}$, find the minimum for each subarray of length k.

(b) Given an array A of integers and $k \in \mathbb{N}$, find i and j such that $|A[i] - A[j]|$ is maximised and $|i - j| \leq k$.

(c) Given an array A of integers and $k \in \mathbb{N}$, find the subarray $A[i : j]$ of maximum length so that $\max A[i : j] - \min A[i : j] \leq k$.

(d) Given an array A of integers, find the largest subarray with average greater than or equal to k.

(e) Given an array A of integers and $\ell, r \in \mathbb{N}$, find the subarray with length between ℓ and r and with maximum sum. What if instead of a sum we are looking for product?

Project 6 [**Real-time PQA**] The amortised $\mathcal{O}(1)$ guarantee might not be ideal for real-time applications. In this project you can investigate a few approaches for making this an $\mathcal{O}(1)$ worst-case guarantee, as described in "Worst-case data structures for the priority queue with attrition".

Deques

Exercise 35 [**Implementation using linked lists**]

(a) Describe how linked lists are used to implement the following core operations of deques: `push_front`, `push_back`, `pop_front`, `pop_front`.

(b) What is the time complexity for each operation?

(c) Argue that deques can implement both lists and queues.

Exercise 36 [**Implementation using an array**] Explain how you would implement a deque using an array.

Exercise 37 [**Functional implementation**] (+) Explain how you would implement deques using stacks (or in OCaml) so that all operations have $\mathcal{O}(1)$ access time. (See also [**2018P1Q2 (b)**])

Project 7 [**Real-time functional deque**] (++) Read the paper "Purely Functional, Real-Time Deques with Catenation" by Kaplan and Tarjan, and write an exposition on the *real-time* functional deque.