CS 225 – Data Structures

Final Exam A

Final A – Model Answers

EXERCISE 1. Suppose you have to sort sequences of real numbers x in the range $a \leq x < b$. For a sequence of length n split $[a, b)$ into n equi-distant buckets B_i ($0 \leq i \leq n-1$), i.e. intervals $B_i = [a_i, a_{i+1})$ with $a_i = a + i \cdot \frac{b-a}{n}$. Use insertion sort to insert each element x of the input sequence into a list ℓ_i, provided $x \in B_i$ holds, then concatenate the lists $\ell_0, \dots, \ell_{n-1}$ to obtain the sorted output list.

Show that the average case time complexity of this bucket sort algorithm is in $\Theta(n)$ assuming that the elements in the input sequence are equally distributed over $[a, b)$.

SOLUTION. Finding i with $x \in B_i$ requires constant time, so the time required to sort an input sequence of size n is

$$T(n) \;\in\; \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \;,$$

where $n_i = \sum_{j=1}^{n} X_{ij}$ is the size of the i'th bucket. Here we use the random variables

$$X_{ij} \;=\; \begin{cases} 1 & \text{if the } j\text{'th input } x_j \text{ is placed into the } i\text{'th bucket } B_i \\ 0 & \text{else} \end{cases}$$

The quadratic complexity bounds comes from the fact that insertion sort requires worst case quadratic complexity.

As n_i is random, so is $T(n)$, and we get the expectation

$$E(T(n)) \;\in\; \Theta(n) + \sum_{i=0}^{n-1} O(E(n_i^2)) \;,$$

as the expectation is a linear function. Then we get

$$\begin{aligned} E(n_i^2) &= E\Bigg(\Big(\sum_{j=1}^{n} X_{ij}\Big)^2\Bigg) \\ &= E\Big(\sum_{j=1}^{n} X_{ij} \cdot \sum_{k=1}^{n} X_{ik}\Big) \\ &= E\Big(\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k\neq j}}^{n} X_{ij} X_{ik}\Big) \\ &= \sum_{j=1}^{n} E(X_{ij}^2) + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k\neq j}}^{n} E(X_{ij} X_{ik}) \end{aligned}$$

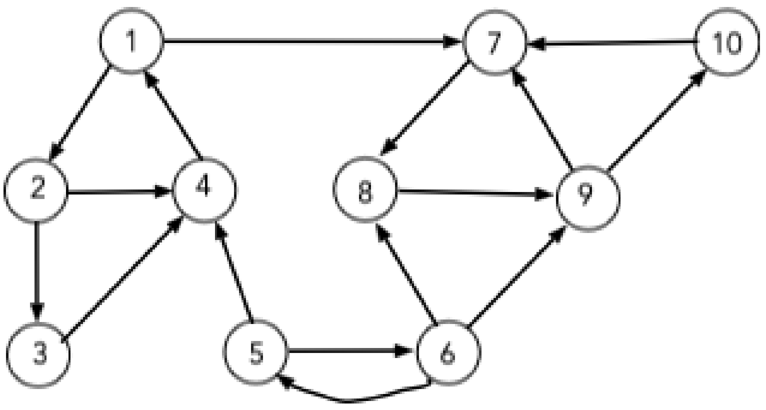
As $P(X_{ij} = 1) = \frac{1}{n}$ holds, we get $E(X_{ij}^2) = 1^2 \cdot \frac{1}{n} + 0^2(1 - \frac{1}{n}) = \frac{1}{n}$.

As X_{ij} and X_{ik} are independent for $j \neq k$ we get $E(X_{ij}X_{ik}) = E(X_{ij})E(X_{ik}) = \frac{1}{n^2}$. This gives

$$E(n_i^2) \ = \ \frac{n}{n} + \frac{n(n-1)}{n^2} \ = \ 2 - \frac{1}{n}$$

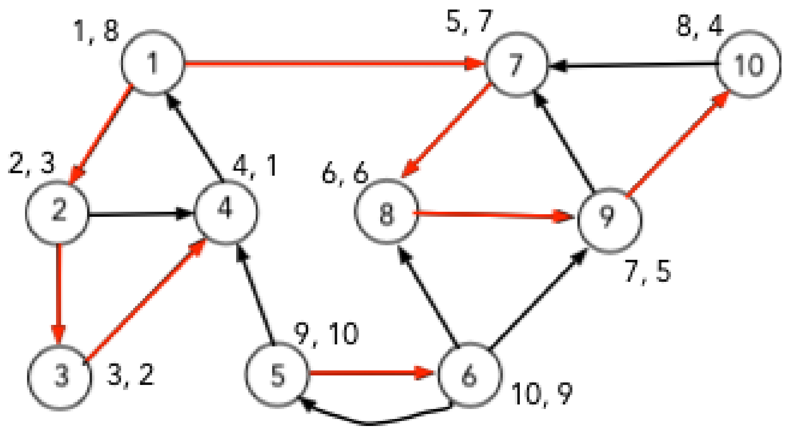
and consequently $E(T(n)) \ \in \ \Theta(n) + O(n(2 - \frac{1}{n})) \ = \ \Theta(n)$.

EXERCISE 2. Let $G = (V, E)$ be the following simple directed graph:

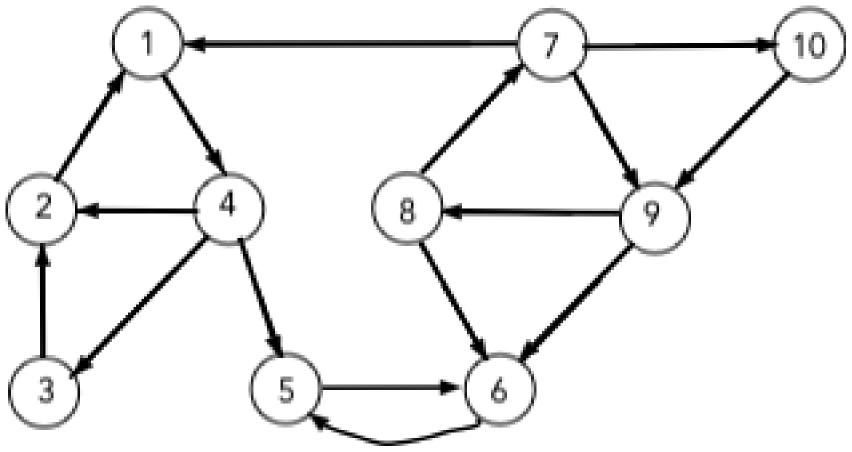


Apply the algorithm for the determination of *strongly connected components* to G. Show explicitly the representation of the input, the progression of the algorithm, and the resulting output.

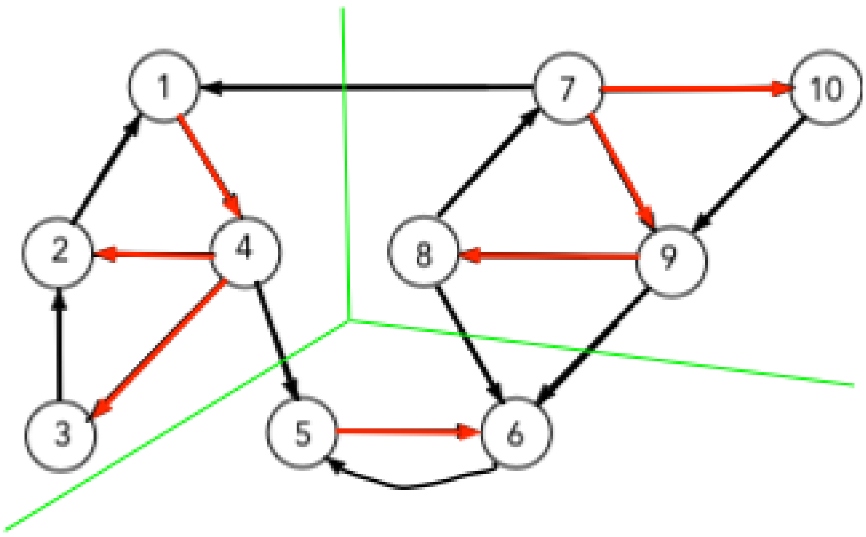
SOLUTION. First execute dfs starting with node 1. So vertices are processed in the order $1, 2, 3, 4, 7, 8, 9, 10$ with finishing times $8, 3, 2, 1, 7, 6, 5, 4$. Then continue with $5, 6$ with finishing times $10, 9$. This creates the following forest in the graph with roots 1 and 5:



Next use the inverse graph $G' = (V, E')$:



Start dfs on G' with node 5, which has highest finishing time and process nodes $5, 6$. Then continue with node 1, which has the highest finishing time among the remaining nodes, and process nodes in the order $1, 4, 2, 3$. Finally, continue with 7, which has the highest finishing time among the remaining nodes, processing $7, 9, 8, 10$. This results in the following forest with roots 5, 1 and 7:



Thus, the strongly connected components are $\{5, 6\}$, $\{1, 2, 3, 4\}$ and $\{7, 8, 9, 10\}$.

EXERCISE 3. Assume a single server that processes one job at a time. Each job i requires one time unit, has a deadline d_i and produces the profit g_i. Consider the following jobs:

job	1	2	3	4	5	6	7	8	9
profit	41	37	31	25	17	13	6	5	4
deadline	3	1	4	3	2	6	4	5	5

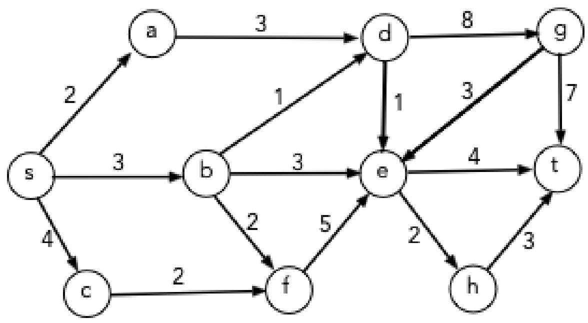
Determine in almost linear time an optimal scheduling of these jobs, i.e. a sequence of jobs that can all be completed within their deadline such that the profit is maximised. Show explicitly the progression of the algorithm including the union-find data structure and the resulting output.

Solution. As $n = 9$ and $\max_{j=1}^{n} d_j = 6$ we need to consider sequences of length 6. Initially all positions are undefined. Furthermore, we use a union-find data structure for candidate sets K with minimum $F(K)$ with an additional fictious position 0. Initially we have $F(\{i\}) = i$ for all $0 \le i \le 6$. Then we proceed creating the scheduling sequence as follows:

job considered	sequence σ	candidate sets K with $F(K)$
	$[\cdot,\cdot,\cdot,\cdot,\cdot,\cdot]$	$[0,1,2,3,4,5,6]$
1	$[\cdot,\cdot,1,\cdot,\cdot,\cdot]$	$[0,1,2,2,4,5,6]$
Find $K = \{3\}$ with $F(K) = 3$, then merge with $L = \{2\}$, so $F(K \cup L) = 2$		
2	$[2,\cdot,1,\cdot,\cdot,\cdot]$	$[0,0,2,2,4,5,6]$
Find $K = \{1\}$ with $F(K) = 1$, then merge with $L = \{0\}$, so $F(K \cup L) = 0$		
3	$[2,\cdot,1,3,\cdot,\cdot]$	$[0,0,2,2,2,5,6]$
Find $K = \{4\}$ with $F(K) = 4$, then merge with $L = \{3,2\}$, so $F(K \cup L) = 2$		
4	$[2,4,1,3,\cdot,\cdot]$	$[0,0,0,2,2,5,6]$
Find $K = \{2,3,4\}$ with $F(K) = 2$, then merge with $L = \{0,1\}$, so $F(K \cup L) = 0$		
5	$[2,4,1,3,\cdot,\cdot]$	
Find $K = \{0,1,2,3,4\}$ with $F(K) = 0$, so the job 5 is rejected		
6	$[2,4,1,3,\cdot,6]$	$[0,0,0,2,2,5,5]$
Find $K = \{6\}$ with $F(K) = 6$, then merge with $L = \{5\}$, so $F(K \cup L) = 5$		
7	$[2,4,1,3,\cdot,\cdot]$	$[0,0,0,2,0,5,5]$
Find $K = \{0,1,2,3,4\}$ with $F(K) = 0$, so the job 7 is rejected, but K is compressed		
8	$[2,4,1,3,8,6]$	$[0,0,0,2,0,0,5]$
Find $K = \{5\}$ with $F(K) = 5$, then merge with $L = \{0,1,2,3,4\}$, so $F(K \cup L) = 0$		
9	$[2,4,1,3,8,6]$	
Find $K = \{0,1,2,3,4,5\}$ with $F(K) = 0$, so the job 9 is rejected		

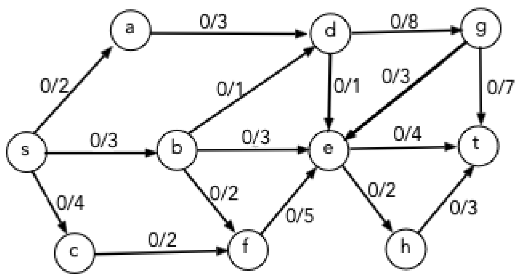
So the optimal scheduling sequence is $2,4,1,3,5,6$.

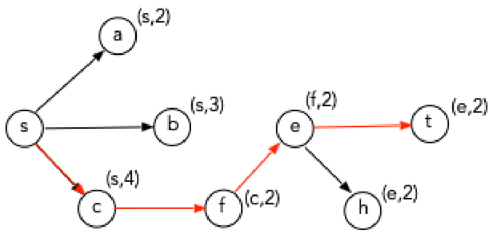
Exercise 4. Consider the following network (G, c) with source s and sink t, where each edge e is labelled by its capacity $c(e)$:



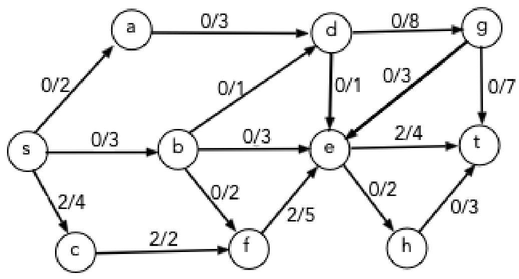
Use the Ford-Fulkerson, Edmonds-Karp or Dinic algorithm to determine a maximum flow in this network. Show explicitly the progression of the algorithm and the resulting output.

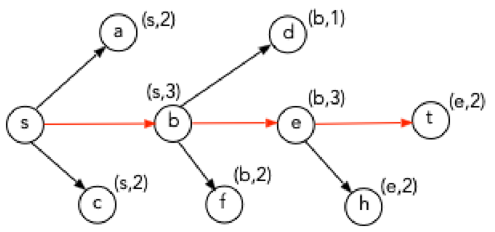
Solution. We initialise the flow to be 0 for all edges. This gives the network and flow on the left. Then the Ford-Fulkerson algorithm yields the augmenting flow in the residual network on the right.



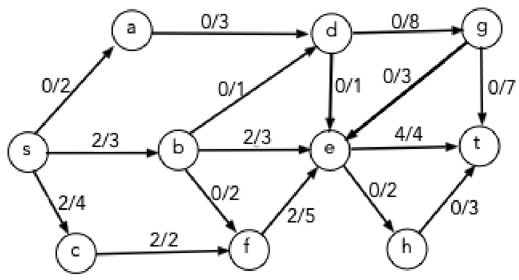


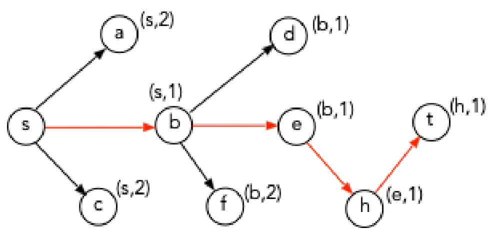
Augmenting the flow gives a new flow as shown on the left. Then the Ford-Fulkerson algorithm yields the augmenting flow in the residual network on the right.



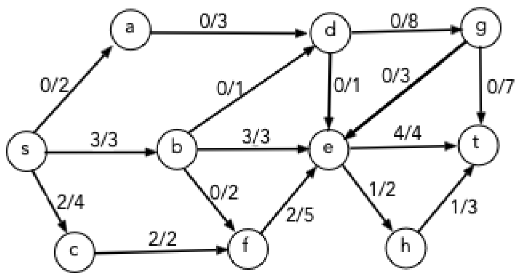


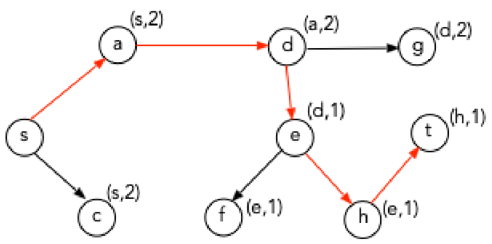
Augmenting the flow gives a new flow as shown on the left. Then the Ford-Fulkerson algorithm yields the augmenting flow in the residual network on the right.



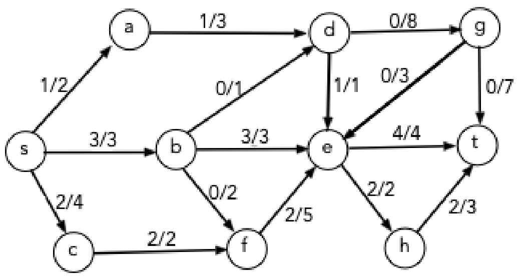


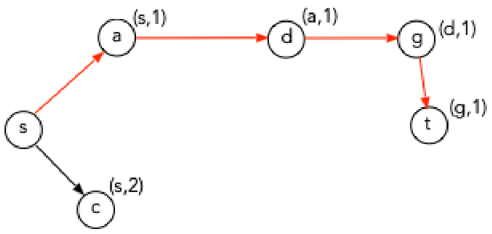
Augmenting the flow gives a new flow as shown on the left. Then the Ford-Fulkerson algorithm yields the augmenting flow in the residual network on the right.



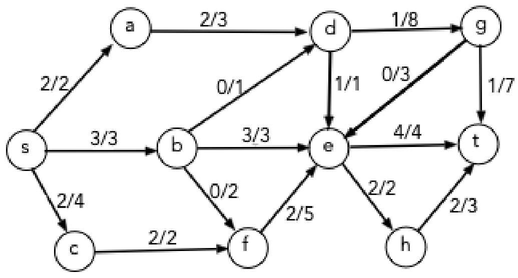


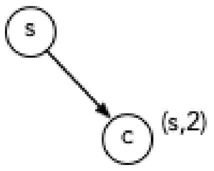
Augmenting the flow gives a new flow as shown on the left. Then the Ford-Fulkerson algorithm yields the augmenting flow in the residual network on the right.





Augmenting the flow gives a new flow as shown on the left. Then the Ford-Fulkerson algorithm fails to find another augmenting flow in the residual network as indicated on the right. So the obtained flow f is maximal with flow value $|f| = 7$.





EXERCISE 5. Show that an arbitrary binary search tree with n nodes can be transformed into any other binary search tree with n nodes by a sequence of $r(n)$ rotations with $r(n) \in O(n)$.

SOLUTION. Use the following way to represent a BST: $v_1, \dots, v_k$ are vertices such that

- v_1 is the root, and v_{i+1} is the right successor of v_i for $1 \leq i \leq k-1$;
- t_i denotes the left successor tree of v_i;
- t_i contains n_i elements (n_i may be 0);

– the right successor tree of v_k is a right-going chain (it may be empty).

Assume that k s maximal with these properties. We show that we need at most $\sum_{i=1}^{k} n_i \leq n-1$ right rotations to transform this BST into a right-going chain.

Obviously, if we can show this for $k = 1$, then it also holds for arbitrary k: We can use $\sum_{i=1}^{k}$ right rotations to turn the subtree rooted at v_2 into a right-going chain, then use n_1 right rotations to turn the whole BST into a right-going chain. So assume now $k = 1$.

Now let v_0 denote the root, and v_1 be its left successor. Let the left and right successor trees of v_1 be t_1 and t_2 with n_1 and n_2 nodes, respectively. Assume that the right successor tree of v_0 is already a right-going chain. Then apply a single right rotation on v_0 and v_1. Using induction ($n_2 < n_1 + n_2 + 1$) we need at most n_2 right rotations to transform the subtree rooted at v_0 into a right-going chain. Again by induction ($n_1 < n_1 + n_2 + 1$) we then need at most n_1 right rotations to transform the the whole BST into a right-going chain.

It remains to have an induction base for $n_i \leq 1$. The four cases $n_1 = n_2 = 0$, $n_1 = 1/n_2 = 0$, $n_1 = 0/n_2 = 1$ and $n_1 = n_2 = 1$ require $1, 2, 2$ and 3 rotations, respectively, i.e. always $\leq n$ rotations.

Finally, if we have two arbitrary BSTs B_1 and B_2 with the same n elements, then there are sequences of r_1 and r_2 right rotations transforming B_1 and B_2 into right-going chains. These chains must be the same BST B. Let these sequences be

$$B_1 \overset{\rho_1}{\to} B_{11} \overset{\rho_2}{\to} B_{12} \to \cdots \overset{\rho_{r_1}}{\to} B \overset{\rho'_{r_2}}{\leftarrow} \cdots \leftarrow B_{22} \overset{\rho'_2}{\leftarrow} B_{21} \overset{\rho'_1}{\leftarrow} B_2 \ .$$

Then for each ρ'_i take the inverse left rotation λ_i, which defines a sequence of rotations

$$B_1 \overset{\rho_1}{\to} B_{11} \overset{\rho_2}{\to} B_{12} \to \cdots \overset{\rho_{r_1}}{\to} B \overset{\lambda_{r_2}}{\to} \cdots \to B_{22} \overset{\lambda_2}{\to} B_{21} \overset{\lambda_1}{\to} B_2$$

transforming B_1 into B_2. The number of rotations is $r_1 + r_2 \leq 2(n-1) \in O(n)$.

EXERCISE 6. Outline an algorithm to delete a key from a hash table, when linear hashing is used for inserting keys.

SOLUTION. Linear hashing uses a sequence of hash functions satisfying $h_0(k) \in \{0, \dots, n-1\}$ and either $h_{j+1}(k) = h_j(k)$ or $h_{j+1}(k) = h_j(k) + 2^j n$ (e.g. $h_j(k) = k \bmod 2^j n$). At each stage we have a linear sequence of B buckets with $2^L n \leq B < 2^{L+1} n$ numbered $0, \dots, B-1$, and each bucket can store up to b keys. Furthermore, $p = B - 2^L n$ denotes the number of the bucket that will be split next, and splitting occurs, when $\frac{N}{B \cdot b} > \varTheta$ becomes true, where $0 < \varTheta < 1$ is some threshold value.

For `delete` use another threshold value $0 < \vartheta < \varTheta$ and apply a *merge condition* $\frac{N}{B \cdot b} < \vartheta$. If the merge condition becomes true, the buckets with numbers $p-1$ and $B-1$ will be merged and p will be decremented by 1. If for these buckets the hash function h_{j+1} was used, the new hash function for the merged bucket will be h_j.

Thus, the `delete` algorithm works as follows:

(i) Use the appropriate hash functions (first h_j, then if necessary h_{j+1} to locate the bucket of the key to be deleted.

(ii) Delete the key from the bucket.

(iii) If after the `delete` the merge condition is satisfied, merge the two buckets $B-1$ and $p-1$, and decrement p.

Exercise 7. Discuss what happens with Fibonacci heaps, if the marking of nodes is dropped.

(i) Describe which operations need to be changed. Which alternatives do you have?

(ii) Describe how the changes affect the amortised complexity of the affected operations. Is it possible to obtain the same complexity bounds?

(iii) Sketch an example of a sequence of operations on a Fibonacci heap, which would have worse complexity in case there are no marked nodes.

Solution. **(i)** Marked nodes in Fibonacci heaps are used to indicate, whether cascading cuts are triggered or not. If there are no marked nodes, we have two alternatives:

(a) Do not perform any cascading cuts, i.e. in the `decrease` operation simply cut out the identified subtree and insert its root into the root list.

(b) Always perform cascading cuts until a root is reached.

(ii) For `decrease` the complexity is in $O(c)$, where c is the number of cascading cuts. For alternative (a) above this becomes $O(1)$, which is equal to the amortised complexity of `decrease` with marked nodes. For alternative (b) in case of cascading cuts the change in the potential is bounded by $4-c$, so the amortised complexity remains in $O(1)$.

However, in both cases it is not possible to maintain a logarithmic bound on the maximum degree of nodes in the Fibonacci heap. For alternative (a) the lack of cut operations does nor decrease degrees. For alternative (b) the abundance of cut operations leads to large root lists. The amortised complexity of `delete_min` including the `consolidate` remains in $O(D(n))$, but as $D(n)$ is not bounded by $\log_\phi n$, this may be as bad as $O(n)$. That is, the amortised complexity bound for `delete_min` becomes worse.

(iii) As the analysis in (ii) shows, we obtain worse complexity, if `delete_min` is executed when $D(n)$ is large (alternative (a)) or the root list is large (alternative (b)). In both cases sufficiently long sequences of `decrease` operations lead to such a state.

Exercise 8. Assume that in order to avoid reorganisation within nodes you use singly linked lists to represent the nodes of a B+-tree. Discuss the effects:

(i) For a B+-tree of order m with fixed size for the keys how much more space is required, when singly linked lists are used?

(ii) Compare the reduction of the effort for reorganisation with the increased effort for insert and delete operations.

SOLUTION. **(i)** In a B+-tree node we store the number s of keys, the parent pointer, $s \leq m-1$ keys (where m is the order of the B+-tree, and $s+1 \leq m$ pointers to children nodes. In a leaf node we have pointers to main data storage instead of pointers to children nodes.

If we use singly linked lists we need to store in addition $s-1$ pointers to successor keys. If he space needed for a key is the same as for storing a pointer, the increase in space is approximately given by a factor $3/2$. Then the height of the tree increases by $\log_2 3 - 1 \approx 0.585$. If the space needed for a key is c-times the space for a pointer, the increase in space reduces to a factor $\frac{c+2}{c+1}$.

(ii) In a B+-tree `Insert` and `delete` require (1) finding the key, and (2) shifting other keys before `insert` or after `delete`. Using just arrays the `find` operation can exploit binary search with complexity in $O(\log m)$. The shifting of keys then requires time in $O(m)$. Using linked lists the `find` operation requires time in $O(m)$, but there is no need to shift keys, as the actual `insert` or `delete` can be done in constant time. In both cases the complexity for `Insert` and `delete` remains in $O(\log n \cdot m$ for a B+-tree with n keys. Differences in the hidden constant depend on the size of the keys.

In summary, using singly linked lists for B+-tree nodes does not offer advantages.

Programming Questions

The model answers for the programming questions comprising header, program and test files are provided by separate zip-files.

EXERCISE 9. Implement a data structure for stacks using singly linked lists of arrays of fixed size. There is no need for a dummy node. Implement the functions `pop`, `push`, `top` and `isempty` for this data structure.

Programming instructions. Use the class template STACK, but modify it using the representation from class template DLIST. However, only sinly linked lists are needed here. Then re-implement the four stack operations.

The header file `dstack.h` and the makefile are provided. A template for the cpp-file `dstack.cpp` is also available—you have to rename `dstack_template.cpp`.

Testing instructions. Test your program on a stack containing the integers $1, \dots, 1000$. Let the size of the arrays be 50. Use `push` to create the stack. Then perform a sequence of 100 **pop** operations, each together with a check of the `top` element before and after the `pop`. Finally, push 35 random elements onto the stack, and return all stack elements by emptying the stack. Show that the stack is finally empty.

The test-file `dstacktest.cpp` is also provided.

EXERCISE 10. Let $G = (V, E)$ be a directed acyclic graph. Implement the algorithm to determine a topological order on the set V of vertices.

Programming instructions. Use the class template GRAPHTRAVERSAL. First modify the base class GRAPH$\langle T \rangle$ such that it can only contain directed graphs. Then modify the depth-first search function `_dfs` such that also the acyclicity of the graph is checked when traversing non-tree edges. Finally, implement a member function `topol_order` that returns the list of vertices in topological order. There is no need for the counters *dfspos* and *finishingtime*; only a list of vertices needs to be returned.

The header file `graphtraversal.h` and the makefile are provided. A template for the cpp-file `graphtraversal.cpp` is also available. You need to rename `graphtraversal_template.cpp`.

Testing instructions. Test your program on the directed acyclic graph with the edges

$$(1,2),(1,3),(2,4),(3,4),(3,5),(5,6),(5,7),(5,8),(6,7),(8,9),(8,10) .$$

The test-file `test.cpp` is provided.

EXERCISE 11. Let $G = (V, E)$ be a connected graph. An *articulation point* is a vertex $v \in V$ such that the subgraph obtained from deleting v is no longer connected. Implement the following algorithm to determine an articulation point in a connected graph:

(i) Perform a depth-first search in G starting from an arbitrary vertex v. Mark all vertices by $dfs(v)$, the order in which the vertices have been processed.

(ii) Traverse the DFS search tree T in *postorder*, i.e. first explore all children from left to right, then a vertex itself. For each vertex v calculate $lowest(v)$ as the minimum of

- $dfs(v)$,
- $dfs(w)$ for all vertices $w \in V$ with an edge $\{v, w\} \in E$ that does not occur in T, and
- $lowest(v')$ for all children v' of v in T.

(iii) Then the root of T is an articulation point iff it has more than one child.

(iv) Any other vertex v is an articulation point iff it has a child w in T with $dfs(v) \leq lowest(w)$.

Programming instructions. Use the class template GRAPHTRAVERSAL. First modify the depth-first search function `_dfs` such that not only *dfspos* and *finishingtime* are computed for each vertex, but also the value of *lowest* and the set of *children*. For this the classes DEEPSEARCHTREE and DEEPTREENODE need to be modified. Finally, implement a member function `articulation` that returns the list of vertices that are articulation points according to criteria (iii) and (iv).

The header file `graphtraversal.h` and the makefile are provided. A template for the cpp-file `graphtraversal.cpp` is also available. You have to rename `graphtraversal_template.cpp`.

Testing instructions. Test your program on the undirected graph with the edges

$$(1,2),(1,3),(2,4),(3,4),(3,5),(5,6),(5,7),(5,8),(6,7),(8,9),(8,10) .$$

The test-file `test.cpp` is provided.