

COMP 572: Combinatorial Optimization – 2007 Fall Semester
Written Assignment # 1
Distributed: Sept 25, 2007
Due: October 11, 2007 at 1:30PM
Sketch Solution Key
Updated October 20, 2007.

Rules of the Game:

1. All solution text must be **typeset**, using e.g., L^AT_EX or MS-Word. Handwritten submissions will not be accepted. If your solution includes figures, though, the figures may be hand drawn.
2. Solutions can either be submitted in hardcopy at the beginning of class on October 11, 2007 or by email to the TA at *qinzhang@cse.ust.hk*. Email submissions should have the subject heading *COMP572 Assignment 1: Your name* and contain an attached file in either PDF, PS or MS-Word format.
3. Please keep a *copy*, either online or a photocopy, of your solutions available after you submit them (in case we misplace your assignment and need to ask you for a resubmit; quite possible, especially with soft copy submissions).
4. Although you may discuss the problems with your fellow students, all solutions must be *your own work*. All contributions from outside sources must be explicitly acknowledged in your solution. For example, if you got an idea from a friend, that friend must be named. If you found a solution in a book or on the web, you should provide a citation.
5. When asked to prove a statement, the only facts that you are allowed to assume are the ones taught in class. For example, you are allowed to prove something by quoting the *max-flow min-cut* theorem as taught in class but not by using a different version of the theorem that you found in a different reference.

Problem 1 Prove the following lemma on flow functions f stated on page 7 of the Max-Flow slides (this is Exercise 26.1-4 in the CLRS book):

1. $\forall X \subseteq V, \quad f(X, X) = 0.$
2. $\forall X, Y \subseteq V, \quad f(X, Y) = -f(Y, X).$
3. $\forall X, Y, Z \subseteq V$ with $X \cap Y = \emptyset,$

$$\begin{aligned} f(X \cup Y, Z) &= f(X, Z) + f(Y, Z), \quad \text{and} \\ f(Z, X \cup Y) &= f(Z, X) + f(Z, Y). \end{aligned}$$

Proof (due to Zhou Zhen)

Let's first prove the second lemma as it will be useful for the others.

2.

$$\begin{aligned} f(X, Y) &= \sum_{x \in X} \sum_{y \in Y} f(x, y) \\ &= \sum_{x \in X} \sum_{y \in Y} -f(y, x) \quad (\text{skew symmetry}) \\ &= \sum_{y \in Y} \sum_{x \in X} -f(y, x) \\ &= -f(Y, X) \end{aligned}$$

1.

$$\begin{aligned} f(X, X) &= -f(X, X) \quad (\text{the second lemma}) \\ \Rightarrow f(X, X) &= 0 \end{aligned}$$

3.

$$\begin{aligned} f(X \cup Y, Z) &= \sum_{v \in X \cup Y} \sum_{z \in Z} f(v, z) \\ &= \sum_{v \in X} \left(\sum_{z \in Z} f(v, z) \right) + \sum_{v \in Y} \left(\sum_{z \in Z} f(v, z) \right) \quad (X \cap Y = \emptyset) \\ &= f(X, Z) + f(Y, Z) \end{aligned}$$

The other part is symmetric.

If you proved 1 first you have to show 1-to-1 correspondence of $f(u, v)$ and $f(v, u), \forall u, v$ in the expanded summation of $f(X, X)$.

You should at least mention $X \cap Y = \emptyset$ to justify the second equality

Problem 2 Let $G = (V, E)$ be an undirected bipartite graph and M a matching of G . Describe an $O(|E| + |V|)$ time algorithm that, given G and M , either (i) confirms that M is a maximum-cardinality bipartite matching of G or (ii) provides an *augmenting path* in G relative to M that permits incrementing the size of M by one.

Note: The definition of an *augmenting path* is given on page 5 of the Matching slides. Also note that using the internal loop of the Hungarian algorithm with all edge weights equal 1 will not work since the Hungarian algorithm assumes that G is a *complete* bipartite graph with $E = L \times R$ which we are not requiring here.

Sketch Proof

As in the class notes on Max-Flow (page 17), construct the flow network G' that would be used to find a max-matching in G . Now let f be the 0/1 flow corresponding to matching M . This can be constructed in $O(|V| + |E|)$ time.

It is straightforward (but necessary) to prove that an augmenting path in the residual network G_f is equivalent to an augmenting path in G for matching M ,

So, finding an augmenting path in G relative to M is the same as finding an augmenting path in G_f , which we know can be done in $O(|V| + |E|)$ time.

By the max-flow min-cut theorem we know that an augmenting path in G_f **doesn't** exist if and only if flow f is maximal. Since we already saw in class that f is maximal if and only if the matching M is maximal, we are done.

Note: To get full credit for this problem you **must** have proven correctness.

1. If you used the proof technique above, you had to have shown that augmenting paths in G_f are in 1-to-1 correspondence with augmenting paths in the matching.
2. If you attacked this problem from first principles using a DFS or BFS based approach it was necessary to show that
 - The paths produced by your algorithm would always be *alternating* paths.
 - If an augmenting path exists your algorithm would produce one.

Problem 3 The Edmonds-Karp algorithm that we learnt in class was the Ford-Fulkerson technique which specified, that in the step

Finds an augmenting path, an s - t path p in G_f along which flow can be pushed (page 12 in the Maximum Flow slides),

the path p had to be a shortest path (where length is number of edges) in G_f . In that case, we proved that only $O(|V||E|)$ augmenting steps were needed. Since finding a shortest path in G_f requires only $O(|E|)$ time, this led to an $O(|V||E|^2)$ time algorithm.

The original Edmonds-Karp paper actually suggested two different ways of choosing p . One was the shortest path approach just mentioned. The other was to always choose a *max-bottleneck* path in G_f .

Let G be a graph with edge capacities $c(e)$ and p an s - t path. The *bottleneck cost* of p is

$$c(p) = \min_{\text{edge } e \text{ on path } p} c(e).$$

(Note that, by definition, the bottleneck cost of p in G_f is exactly what we previously defined as $c_f(p)$.) A path p is a *max-bottleneck* path on G if

$$c(p) = \max\{c(p') : p' \text{ is an } s\text{-}t \text{ path in } G\}.$$

The second Edmonds-Karp technique was to always choose a *max-bottleneck* path in G_f to augment. We will call this the **MP** heuristic.

(a) Prove that if f is any flow in a original graph G with capacities $c(e)$ and p is a max-bottleneck path in G then

$$c(p) \geq \frac{|f|}{|E|}.$$

Hint: Let e' be the bottleneck *edge* on bottleneck path p . Construct an (S, T) cut in G_f such that $C(S, T) \leq c(e') \cdot |E|$.

(b) Let f^* be an optimal flow in the original graph. Let f_t be the flow constructed after t steps of the **MP** heuristic. Prove that

$$|f^*| - |f_t| \leq \left(1 - \frac{1}{|E|}\right)^t |f^*|.$$

Hint: Let f_t^* be the optimal flow available in G_{f_t} and \bar{c}_t be the value of the max-bottleneck flow that can be pushed through G_{f_t} . What does part (a) tell you about the relationship between \bar{c}_t and f_t^* ? What does that tell you about $|f^*| - |f_t|$?

(c) Assume that all capacities are integral. Prove as good a bound D as possible on the number of augmenting steps that the **MP** heuristic will require. Your bound can be a function of $|E|$, $|V|$ and $|f^*|$.

Note that since a bottleneck path can be found in $O(|E| \log |V|)$ time (how?), you have just proven that the running time of the **MP** heuristic is $O(D|E| \log |V|)$.

Proof

(a) Let the set S consist of s together with all nodes which can be reached from s by a directed path in G consisting of edges (u, v) such that $c(u, v) > c(p)$. Let T denote the remaining nodes. It is easy to see that $t \in T$ so (S, T) is a *cut*.

Note that every edge (u, v) crossing the cut (S, T) satisfies $e(u, v) \leq c(p)$ since otherwise $v \in S$.

G has $|E|$ edges, so at most $|E|$ edges cross the (S, T) cut. We therefore have $C(S, T) \leq c(p) \cdot |E|$. Since, for *any* flow f , we have $|f| \leq C(S, T)$, therefore,

$$\frac{|f|}{|E|} \leq c(p).$$

Note: To get this part correct you had to have defined the cut. You couldn't just say that there is some cut with $C(S, T) \leq c(p) \cdot |E|$.

(b) We can further prove that,

If f is any flow in an original graph G with capacities $c(e)$, and p is a max-bottleneck path in the corresponding residual graph G_f , then

$$|f^*| - |f| \leq c(p) \cdot |E|,$$

where f^* is an optimal flow in G .

Let f be some current flow in G . The residual capacities in G_f are given by $c_f(e) = c(e) - f(e)$. Let $c(p)$ be the max bottleneck cost of an augmenting path in G_f . Using the same technique as in (a) we can construct an (S, T) cut in G_f such that $C_f(S, T) \leq c(p) \cdot |E|$. Now notice that

$$c(p) \cdot |E| \geq C_f(S, T) = C(S, T) - f(S, T) \geq |f^*| - |f|.$$

By the given definition of f_t we have $c(p) = |f_{t+1}| - |f_t|$. Therefore,

$$|f^*| - |f_t| \leq (|f_{t+1}| - |f_t|) \cdot |E|,$$

or, equivalently,

$$|f^*| - |f_{t+1}| \leq (|f^*| - |f_t|) \left(1 - \frac{1}{|E|}\right).$$

Letting $|f_0| = 0$ (i.e., starting with a zero flow) we get, by induction,

$$|f^*| - |f_t| \leq |f^*| \left(1 - \frac{1}{|E|}\right)^t.$$

(c) Since all capacities are integral, each flow is integral. Thus, if f_t is not a maximum flow, then

$$|f^*| - |f_t| \geq 1,$$

so from part (b)

$$|f^*| \left(1 - \frac{1}{|E|}\right)^t \geq 1.$$

Now note that $(1 - 1/x)^x \leq e^{-1}$ so,

$$\left(1 - \frac{1}{|E|}\right)^{|E| \ln |f^*|} \leq \frac{1}{|f^*|}$$

implying that

$$t \leq |E| \ln |f^*|.$$

Note: To get full credit you needed to have an answer in a form similar to $O(|E| \ln |f^*|)$.

Problem 4 (20 points) Suppose that we have m workers and n tasks with the following constraints.

- Worker i can work on t_i jobs at one time. Thus $\forall i, t_i \leq n$.
- Job j requires s_j workers to complete it. Thus $\forall j, s_j \leq m$.
- $\sum_{i=1}^m t_i = \sum_{j=1}^n s_j$, so we have exactly enough workers to complete all of the jobs.
- A *valid job assignment* is an assignment of workers to jobs so that worker i performs exactly t_i jobs, and job j has exactly s_j workers participating in it.
- Each worker i also has a list of preferred jobs P_i on which he'd like to work. For a given valid job assignment, let G_i be the number of jobs on his preferred list to which worker i gets assigned.

(a) Give an algorithm that maximizes *average worker satisfaction*.

That is, your algorithm should provide a valid job assignment that maximizes

$$\frac{1}{m} \sum_{i=1}^m G_i$$

over all valid job assignments.

Give the most efficient algorithm you can find. Your algorithm's running time should be stated in $O()$ notation as a function of n and m .

(b) Now give an algorithm that maximizes *max-min worker satisfaction*. That is, your algorithm should provide a valid job assignment that maximizes

$$\min_{1 \leq i \leq m} G_i$$

over all valid job assignments.

As above, give the most efficient algorithm you can find. Your algorithm's running time should be stated in $O()$ notation as a function of n and m .

Hint: Model the problem as a network-flow one.

Note: After the due date it was noted that this problem was actually *ill-defined* since it didn't clearly discuss the situation in which one worker did two units of work on the same job. To see that we must allow this, consider for example the case with 3 workers and 3 jobs.

$$t_1 = 1; t_2 = 2; t_3 = 3; \quad s_1 = s_2 = 3; s_3 = 0.$$

This satisfies the conditions of the problem, yet it is easy to see that there is no valid job assignment in which job 1 and job 2 both have three different workers working on them.

Since we will allow a worker to do multiple work-units on one task the question then becomes, “how do we score this?” For example, suppose job 1 is on P_1 and worker 1 does 2 units of work on this job. Should this be counted as one unit of good work or two? The solution that we will look at assumes that this counts as two units of good work. *Proof*

We will say that an assignment is a partial assignment if worker i is assigned no more than t_i units of work and job j has no more than s_j units assigned to it.

Before starting we note that, given the setup of the problem as described above, i.e., that a worker can do more than one unit of work on one job, then every partial assignment of workers to jobs can be easily completed to a full assignment (how?) in time linear in $m + n$. (Note that if we did not make this assumption then it might be possible that a partial assignment could *not* be completed to a full assignment.)

(a) We first model the problem as a bipartite graph with workers on one side (that is, the workers form the vertex set $L = \{u_1, u_2, \dots, u_m\}$, u_i denotes worker i) and jobs on the other (that is, the jobs form the vertex set $R = \{v_1, v_2, \dots, v_n\}$, v_j denotes job j). If job v_j is in worker u_i 's preference list, we add an edge from u_i to v_j . For every edge (u_i, v_j) , we assign $c(u_i, v_j) = t_i$.

As in the method given in class to model the Max-Cardinality Bipartite-Matching problem as a Max-Flow problem, we add a source vertex S and a sink T to the graph. We connect S to each u_i ($1 \leq i \leq m$), and assign edge (S, u_i) capacity t_i . We also connect each v_j ($1 \leq j \leq n$) to T with capacity s_j . Note that all the edges added are directed.

We denote the graph obtained by the construction above as $G = (V, E)$, where V is the vertex set and E is the edge set. Clearly, $V = O(m + n)$ and $E = O(mn)$. We run the Ford Fulkerson algorithm to solve the max-flow problem on the new graph, and we denote by f^* the value of the maximum flow. Recall that when FF is run on a graph with integral capacities such as this one, the resulting flow also has integral values.

Note that any integral flow on this graph must satisfy

1. $\forall i, j, f(u_i, v_j)$ is an integer
2. $\forall i, \sum_j f(u_i, v_j) = f(S, u_i) \leq t_i$
3. $\forall j, \sum_i f(u_i, v_j) = f(v_j, T) \leq s_j$.

Take any integral flow. We will call the (u_i, v_j) edges with $f(u_i, v_j) > 0$, *good*.

Now assign worker i to do $f(u_i, v_j)$ units of work on job j . By properties (2) and (3) above this is a legal partial assignment so, as discussed at the very beginning, we can complete this into a full assignment.

G_i for this assignment will be exactly the flow over (S, u_i) so $\sum_i G_i$ will be the value of the flow.

Alternatively, given any assignment, one can use the *good jobs* to define a flow in the graph and the flow will have value equal to $\sum_i G_i$.

Thus, finding a max-flow in the graph is equivalent to finding an integral assignment that maximizes $\sum_i G_i$, i.e., one that maximizes $\frac{1}{m} \sum_i G_i$. We can do this using the Ford-Fulkerson algorithm.

Now we show that the total running time is $O(m^2n^2)$.

Clearly, the time spent on the construction of G is $O(mn)$. Now we try to bound the running time of the Ford Fulkerson algorithm. From class we know that, for integral assignments, this takes at most

$$O((|E| + |V|)f^*)$$

time. Since $|V| = O(m + n)$, $|E| = O(mn)$ and $f^* \leq mn$, we get a final bound of $O(m^2n^2)$ time.

(b) The general idea to find $G^* = \max_{\text{all assignments}} \min_{1 \leq i \leq m} G_i$ is to perform a binary search for G^* using a modified version of (a).

We maintain an upper bound R and a lower bound L for the value G^* ; initially, we set $L = 1$ and $R = \min\{t_i\}$ ($i = 1, 2, \dots, m$).

Formally our algorithm is **Algorithm. Find_Max_Satisfaction**(L, R)

1. If $L \geq R$, return L .
2. Let $M = \lceil (L + R)/2 \rceil$.
 - (a) If **Check_Flow**(M) = true, return **Find_Max_Satisfaction**(M, R).
 - (b) return **Find_Max_Satisfaction**($L, M - 1$).

The subroutine **Check_Flow**(M) works as follows. We build the same graph G that we had in part (a) with the one exception that $\forall i$, we set $c(S, u_i) = M$. We run the Ford Fulkerson algorithm on the graph with these edge capacities. If the resulting max flow value is equal to $(M \cdot n)$, the subroutine returns true, otherwise, returns false.

Note that if the routine returns true then, since the flow value is $M \cdot n$, we must have $\forall u_i, f(S, u_i) = M$. Furthermore, as in part (a), we can create an assignment that contains all of the good edges in the matching. Thus, in this assignment $G_i \geq f(S, u_i) = M$. So, if **Check_Flow**(M) returns true, there is an assignment with $\min_i G_i \geq M$.

In the other direction, if there is an assignment with $\min_i G_i \geq M$, create that assignment. For each worker, chose M jobs on his list that he will do in that assignment. We can then create a flow of value $M \cdot n$ on the associated graph (by setting the flow on those chosen edges equal to 1) so **Check_Flow**(M) will return true.

Therefore, **Check_Flow**(M) will return true if and only if there is an assignment with $\min_i G_i \geq M$ so our binary search above works properly.

The running time is $O(m^2 n^2 \log n)$, since the binary search makes at most $O(\log n)$ calls and, similar to part (a), each call to the **Check_Flow**(M) subroutine costs time $O(m^2 n^2)$.

Problem 5 A dynamic dictionary data structure. Do Parts A,B,C.

A *Dynamic Dictionary* data structure supports operations:

- Find(x):** Returns a pointer to the record with key x .
If it does not exist, return a null pointer.
- Succ(x):** Returns a pointer to the record containing the smallest key larger than x . If it does not exist, return a null pointer.
- Pred(x):** Returns a pointer to the record containing the largest key smaller than x . If it does not exist, return a null pointer.
- Insert(x):** Insert a record with key x into data structure.
- Delete(x):** Delete record with key x from data structure (assumes that a pointer to the record is given)

A balanced binary search tree, e.g., an AVL or 2-3 tree, can be used to implement these operations in $O(\log n)$ worst case time per operation, where n is the number of nodes currently in the tree. In this problem, you will see a simpler data structure that implements these operations, and be asked to perform an amortized analysis of it.

A *Leaf Binary Search Tree (LBST)* is a BST in which all data is kept in the leaves; internal node u stores the maximum value of all of the keys in the subtree rooted at the left child of u . Internal node information is therefore only used for navigating in the tree. Each non-leaf node in a LBST will have pointers to its left and right children and each non-root node in a LBST will also contain a pointer to its parent.

A LBST will be *full* if it contains 2^i leaves for some integer $i \geq 0$; we will call such trees full-LBSTs. Note that the data in two LBSTs of size 2^i can be merged into a LBST of size 2^{i+1} in $O(2^i)$ time (a merge operation destroys the two original trees).

Our Dynamic-Dictionary (DD) structure will consist of a collection of full-LBSTs, such that no trees in the DD data structure have the same size.

Part A: (warmup 1)

Prove that **Find(x)**, **Succ(x)** and **Pred(x)** can be implemented in worst case $O(\log^2 n)$ time, where n is the number of keys stored in the DD.

Proof

To perform these operations one performs **Find(x)**, **Succ(x)** and **Pred(x)** on each of the individual LBSTs and then returns a combined result. The amount of time each of these operations requires on a LBST of height i is $O(i)$.

Suppose there are n keys in the DD. Let k be the size of the largest LBST. Then $n \geq 2^k$ so $k \leq \log_2 n$.

Since there is at most one LBST of each size there are at most $O(\log n)$ LBSTs in the data structure and each one has height at most $O(\log n)$. Thus, the total amount of time needed for these operations is $O(\log^2 n)$.

Insert(x) is implemented by

1. Creating a size 1 full-LBST containing x as its only data.
2. While there are two full-LBSTs of the same size (2^i) in the DD, merge them together to create a new full-LBST of size 2^{i+1}

Figures 1-6 illustrate this operation.

Part B: (warmup 2)
Prove that **Insert(x)** takes $O(\log n)$ amortized time where n is the number of elements in the DD at the time that x is inserted.

Proof

A tree of size 2^i is created (perhaps as an intermediate step towards the creation of a larger tree) by merging two trees of size 2^{i-1} exactly when n is a multiple of 2^i . So, a tree of size 2^i is created every 2^i time units. Creating a tree of 2^i by merging two trees of size 2^{i-1} requires $O(2^i)$ time. So, the total work needed for merging in a sequence of n insert operations will be

$$\sum_{i=1}^{\log n} \left\lfloor \frac{n}{2^i} \right\rfloor O(2^i) \leq \sum_{i=1}^{\log n} \frac{n}{2^i} O(2^i) = O(n \log n)$$

and the average amount of work required per step for merging is

$$\frac{1}{n} O(n \log n) = O(\log n).$$

Note that we haven't included the cost of actually creating a tree of size 0 at each step but this is $O(1)$ so the total amount of amortized work per step is $O(1 + \log n) = O(\log n)$.

Delete(x) is implemented using a *lazy* protocol.

That is, when **Delete(x)** is called:

1. node x is *marked* but not deleted.
2. The algorithm walks up from x , marking an internal node if both of the internal node's children are marked. It stops the walk when it reaches an unmarked internal node whose "other" child is not marked.
3. If, after marking x , half of the nodes in the 2^i full-LBST T containing x are now marked then, in $O(2^i)$ time, T is destroyed. If $i > 0$ then a new LBST T' is built containing the 2^{i-1} unmarked nodes from T . If the DD already contains a size 2^{i-1} full-LBST T'' , then T' and T'' will be merged in $O(2^i)$ time creating a new size 2^i full-LBST.

Note that the actual cost of the delete operation is the time needed to mark all appropriate nodes *plus* the time needed to destroy and create new trees.

Figures 7-13 illustrate this operation.

Part C:

Assume that a DD implements **Insert(x)** and **Delete(x)** as described above. Let n be the number of items in the DD at the time that an operation is performed.

1. Prove that **Find(x)**, **Succ(x)** and **Pred(x)** can be implemented in worst case $O(\log^2 n)$ time. (Take marked nodes into account).
2. Prove that **Insert(x)** takes $O(\log n)$ amortized time.
3. Prove that **Delete(x)** takes $O(1)$ amortized time.

Note: your proof from Part B that **Insert(x)** takes $O(\log n)$ amortized time might have to be modified in order to remain correct once **Delete(x)** is allowed.

Proof

1. The algorithms for **Find(x)**, **Succ(x)** and **Pred(x)** are very similar to the ones in Part A with the exception that they have to be modified to handle marked nodes. This is straightforward and won't be done here **but you needed to at least mention this in your solution.**

The key part of the analysis was that it was necessary to notice that the introduction of marked keys might increase the number of LBSTs.

But, since every LBST has more unmarked keys than marked keys, we see that the total number of keys in the LBST is $\leq 2n$. Therefore, by the same observations as in part A, we find that the largest LBST can have height at most $\log(2n) = 1 + \log n$, where n is the number of active keys. Again, as before, this

means that we can have at most $O(\log n)$ LBSTs in our DD, each of height at most $O(\log n)$ and we still have a $O(\log^2 n)$ time bound.

2. In order to get this part correct you had to have shown that your analysis of the amortized costs of *Insert* and *Delete* were valid when you could have *Deletes* occurring between *Inserts*, and *Inserts* occurring between *Deletes*. In particular, this meant that an analysis that examined the two operations separately, would not work.

We will use the accounting method. This proof is based on an approach suggested by Yiu Wai Pun.

Every LBST will have credits associated with it. Merging two LBSTs of size 2^{i-1} to become a LBST of size 2^i requires $O(2^i)$ work in total (including the actual destruction of the old trees, the creation of the new ones *and* the marking of any appropriate nodes).

Note that to be fully correct an analysis had to at least mention that when merging two LBSTs of size 2^{i-1} we might have to mark as many as $\Theta(2^{i-1})$ internal nodes in the new LBST, but that this can be done in $O(2^i)$ total time.

We scale the value of a credit so that 2^{i-1} credits hold enough value to pay for the merger of two LBSTs of size 2^{i-1} .

Let T be a LBST. Let N_T be the *total number of keys, both marked and unmarked*, in LBSTs *smaller* than T . Let C_T be the number of credits currently stored in T .

A LBST is *good* if $N_T \leq C_T$, i.e., if C_T holds at least as many credits as there are total nodes in all smaller LBSTs.

An **Insert(x)** operation is now defined as

- Add one credit to every LBST in the DD. (Since there are $O(\log n)$ LBSTs, this uses $O(\log n)$ credits.)
- Create a size-1 LBST T'_0 containing x . (Uses $O(1)$ time).
- Merge same size LBSTs together until there are no two LBSTs of the same size. When merging two LBSTs T_1 and T_2 of size 2^{i-1} into a new LBST T of size 2^i put all of the credits from the old trees into T and pay for the merge using 2^{i-1} credits from the new combined tree. So, $C_T = C_{T_1} + C_{T_2} - 2^{i-1}$. A merge is *legal* if $C_T \geq 0$, i.e., there were enough credits to pay for the merge.

Claim: Suppose all LBSTs are *good* before **Insert(x)**. Then all merges performed during **Insert(x)** are legal and all LBSTs in DD after the operation is finished are good.

Note that when we start with an empty DD, all existing LBSTs (none) are good so this claim implies that, for a sequence of **Insert**(\mathbf{x}) operations, the amortized cost of **Insert**(\mathbf{x}) is $O(\log n)$. This is because it says that besides the $O(1)$ initial work per step, prepaying $O(\log n)$ credits per step is enough to cover the cost of all later merges.

In the next section we will show that if all LBSTs in the DD are good before a **Delete**(\mathbf{x}) operation they will remain good after a **Delete**(\mathbf{x}) operation. Combined with the above claim, this will imply that even in the presence of **Delete**(\mathbf{x}) operations, the amortized cost of **Insert**(\mathbf{x}) is $O(\log n)$.

Proof of claim: Suppose that immediately before **Insert**(\mathbf{x}) is performed there are LBSTs of size 2^i , where $i = 0, 1, 2, \dots, m$, but no LBST of size 2^{m+1} . Let T_i be the LBST of size 2^i .

Then, after the merges are performed, all of the T_i ($i = 0, 1, 2, \dots, m$) will have been destroyed and there will be a new LBST T_{m+1} of size 2^{m+1} . One can think of this as merging T'_i and T_i to create T'_{i+1} for $i = 0, 1, \dots, m$, and then setting $T_{m+1} = T'_{m+1}$.

Note that the LBSTs of size greater than 2^{m+1} in the new DD are the same as in the old DD. Since all merges were done in smaller LBSTs, if these LBSTs were good before the merge they remain good after the merge. So, we only need to examine T_i , for $i = 0, 1, 2, \dots, m + 1$.

Before **Insert**(\mathbf{x}) starts, C_{T_i} , where $i \leq m$, is bigger than the sum of the sizes of all of the smaller LBSTs, i.e., $C_{T_i} \geq \sum_{j=0}^{i-1} 2^j = 2^i - 1$. After the addition of one more credit in the first step of **Insert**(\mathbf{x}), $C_{T_i} \geq 2^i$. Therefore, every merge of T'_i and T_i to create T'_{i+1} can be paid for using the 2^i credits already in T_i and the merge is legal. After the last merge we might have that $C_{T_{m+1}} = C_{T'_{m+1}} = 0$ since all of the credits were used up to pay for the last merge. Note, though, that because there are no keys in any smaller LBSTs, $N_{T_{m+1}} = 0$ and T_{m+1} is good.

3. We now show that **Delete**(\mathbf{x}) takes $O(1)$ amortized time. We will show this by placing 2 credits on each marked node, one red and one blue. The total amortized cost will be $O(1)$ plus the cost of the two credits which will be $O(1)$ in total.

As mentioned, you had to show that your analysis for Delete works in the presence of Inserts. Since a Delete operation might require walking up an arbitrarily long path in an LBST to mark internal nodes, you also needed to show how to pay for walking up this path.

Before starting, we note that there are two very different cases when deleting a node from a size 2^i LBST T :

- (i) if fewer than half the nodes are marked, then we must mark the new node and its appropriate ancestors. We will maintain the invariant that all marked nodes whose parents are not marked will contain a red credit.

- (ii) if half the nodes are marked then, in $O(2^i)$ time, we destroy T and, in $O(2^i)$ time, create a new size 2^{i-1} LBST T' . If the DD already contained a size 2^{i-1} LBST T'' then, in $O(2^i)$ time, we must merge T' and T'' to create a new size 2^i LBST T''' . Note that at most one such merge can occur.

For (i) we mark x . Note that all of the ancestors of x are unmarked (since if one was marked that would mean that x had been deleted from the tree earlier). Let $p(x)$ be the parent of x and $s(x)$ be the unique sibling of x , i.e., the other child of $p(x)$.

We now start the following procedure at $t = x$ to walk up the tree, marking appropriate ancestors.

- (a) Check if $s(t)$ is marked (this can be done in constant time by following two pointers). If it isn't, put a red credit on t and exit the walk.
- (b) If $s(t)$ is marked, then there is a red credit stored at $s(t)$ (since t was unmarked, $s(t)$ is a marked node whose parent is unmarked and therefore contains a credit). Use that red credit to pay for marking $p(t)$ and setting $t = p(t)$ (walking one step up the tree). Go back to (a).

Note that every step in this procedure is paid for by a previously placed red credit except for the last one. So, the total unpaid cost of this procedure is $O(1)$ and one red credit. Note too, that it maintains our invariant that all marked nodes whose parents are not marked will contain a red credit.

For (ii) note that when the tree is destroyed there will be 2^{i-1} marked nodes and therefore 2^{i-1} blue credits in the LBST. This is enough to pay the $O(2^i)$ cost of destroying the old LBST, creating the new smaller one and, possibly, merging the new smaller one with one other old one. Also note that we are paying for the destruction and merge using blue credits so we do not have to use the credits already stored in the tree by the **Insert(x)** procedure. We can therefore transfer all of those old credits to the newly created trees and save them for later. That is, $C_{T'} = C_T$, and if T''' is created, then $C_{T'''} = C_T + C_{T''}$.

It remains to prove one final statement:

Claim: Suppose all LBSTs are *good* before **Delete(x)**. Then all LBSTs are good after **Delete(x)**.

As mentioned earlier, this claim implies that **Insert(x)** still has an $O(\log n)$ amortized cost, even when it is interspersed with **Delete(x)** operations.

Proof of Claim: Let x be in some LBST T of size 2^i . If T is not destroyed, then the claim is obviously true since we have not changed the number of keys in any tree.

If T is destroyed, note that all LBSTs of size $< 2^{i-1}$, which were good before, remain good. This is because no keys in smaller LBSTs have been physically removed from or added to the DD.

Similarly all LBSTs of size $> 2^i$, which were good before, remain good. This is because the set of keys kept in smaller LBSTs can only have *gotten smaller* (by the removal of marked keys).

So, we only need to show that if T' or T''' is created, then it is good.

First suppose that no old T'' , a LBST of size 2^{i-1} , existed. Then $N_{T'} = N_T$. Since $C_{T'} = C_T$,

$$N_{T'} = N_T \leq C_T = C_{T'},$$

so T' is good.

Now suppose that an old T'' , a LBST of size 2^{i-1} , existed. Then by definition, $N_{T'''} = N_{T''}$ (since the trees smaller than $N_{T'''}$ are exactly the trees smaller than $N_{T''}$), so

$$N_{T'''} = N_{T''} \leq C_{T''} \leq C_{T''} + C_T = C_{T'''},$$

so T''' is good.

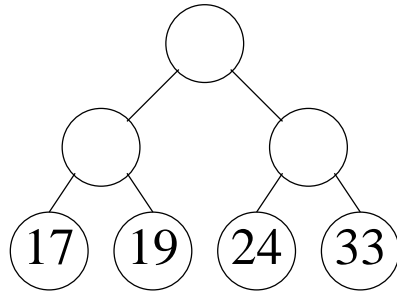


Figure 1: An original DD with 4 items.

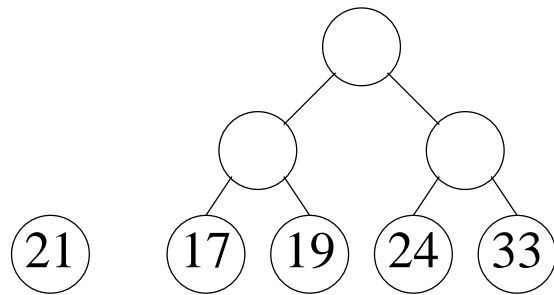


Figure 2: Key 21 added to the DD.

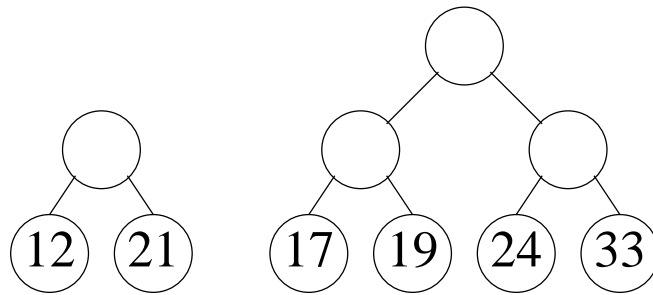


Figure 3: Key 12 added to the DD.

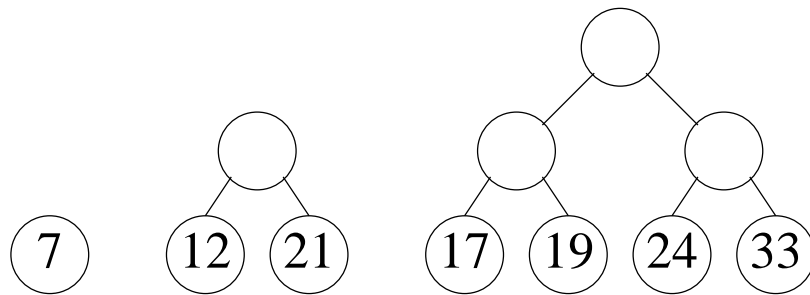


Figure 4: Key 7 added to the DD.

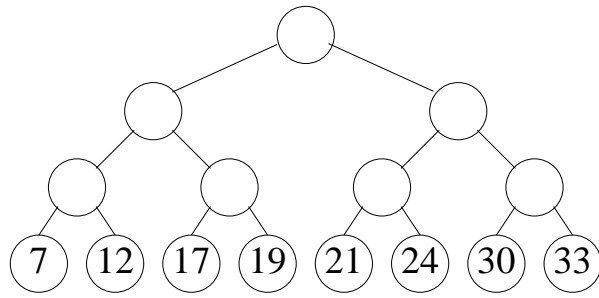


Figure 5: Key 30 added to the DD.

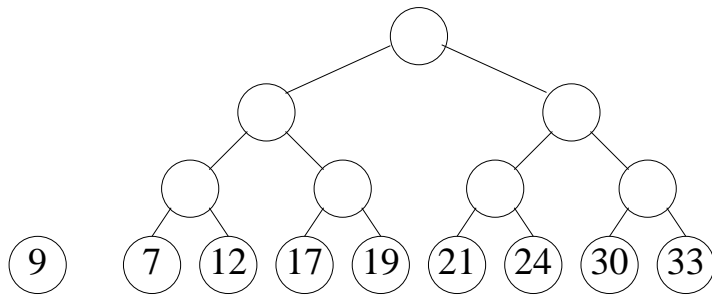


Figure 6: Key 9 added to the DD.

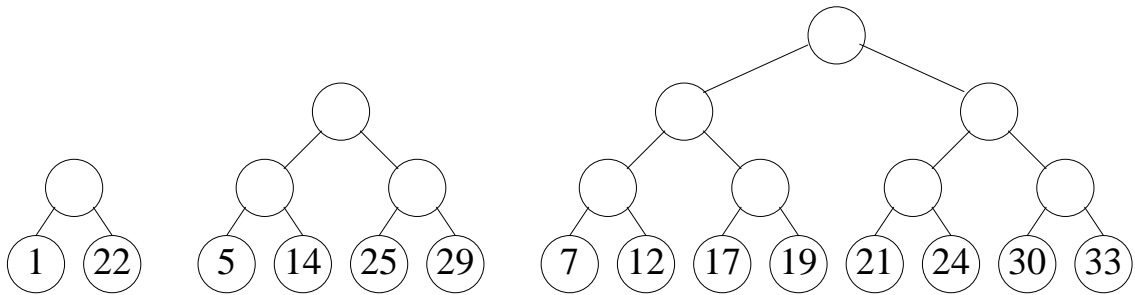


Figure 7: Another Original DD

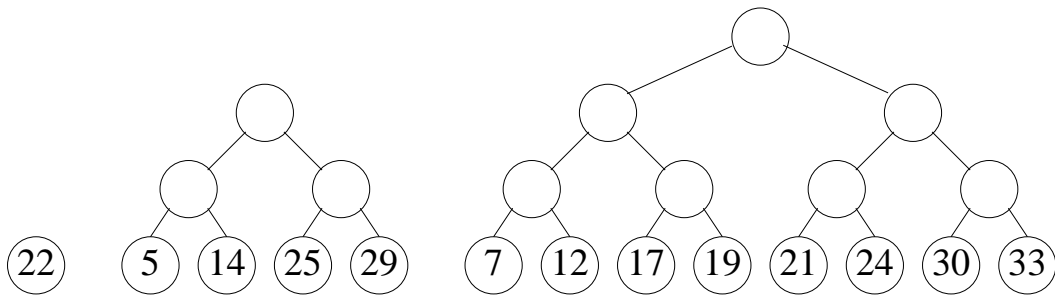


Figure 8: Key 1 deleted from the DD.

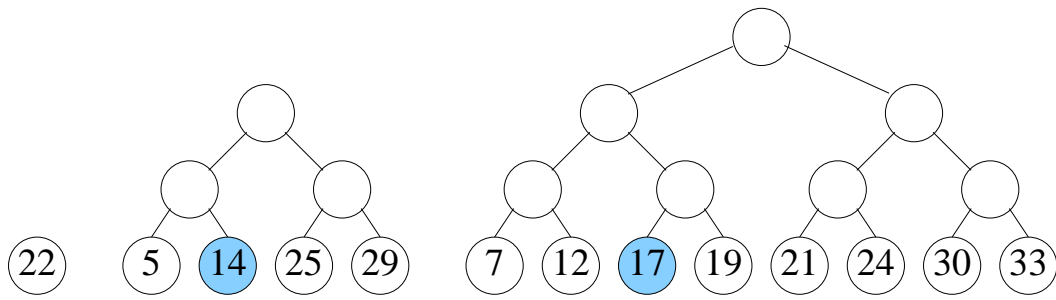


Figure 9: Keys 14 and 17 deleted from the DD.

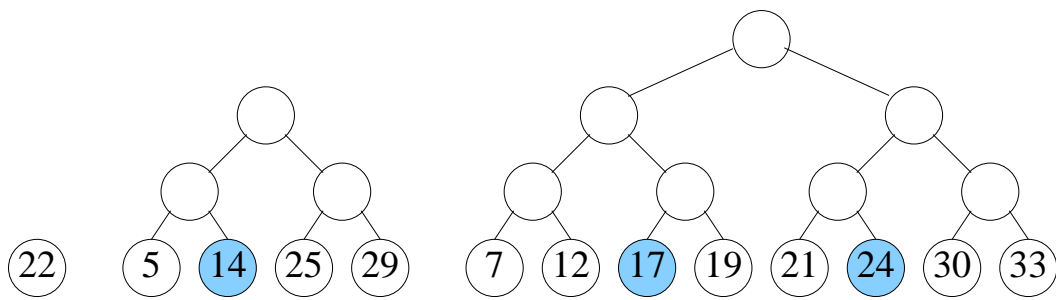


Figure 10: Key 24 deleted from the DD.

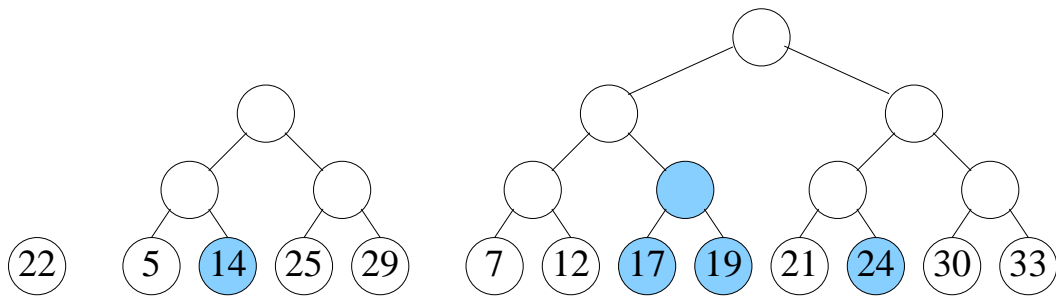


Figure 11: Key 19 deleted from the DD. Note that parent of 19 is also marked

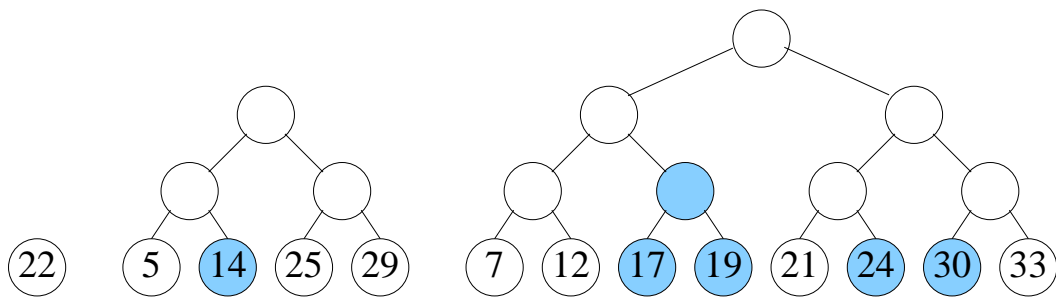


Figure 12: Key 30 deleted from the DD: step 1.

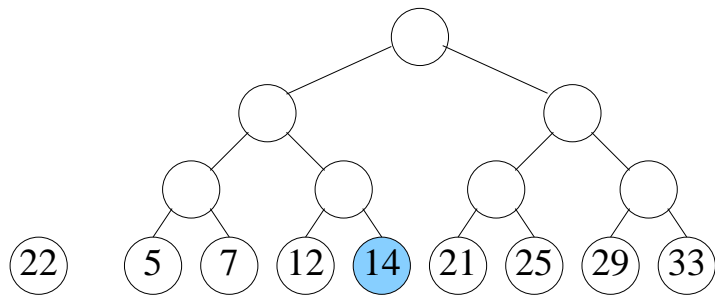


Figure 13: Key 30 deleted from the DD: step 2. Note that the original tree of size 8 was destroyed and its unmarked nodes merged with the nodes in the tree originally of size 4. Also note that marked node 14, from the tree of size 4, is kept in the new merged tree.