Comp 7720 - Online Algorithms

Notes 2: Self-Adjusting Data Structures

Shahin Kamalli
University of Manitoba - Fall 2017

October 7, 2017

1 Introduction

A data structure is *self-adjusting* if it adjusts itself based on the queries that it answers. For example, a classic AVL-tree is self-adjusting after insertion/deletion to keep itself 'balanced' so that searching for any item takes logarithmic time. In this section, we mainly consider self-adjusting structures when the input is a sequence of 'accesses' to items maintained by the data structure. In particular, we assume there is no insertion/deletions in the underlying structures.[1] Note that if there is no insertion/deletion, AVL tress (more generally any data structure that you have seen in your previous courses) does not adjust itself to the patterns observed in a sequence of accesses. However, in practice, it is desirable to adjust the data structure so that the total cost incurred for searching all requested items is minimized.

In a self-adjusting structure, the input is a sequence of *requests* to items stored in the data structure. In order to access each item, a number of *probes* is made. We call this the *access cost* for the request. A data structure can organize itself through some exchanges, rotations, etc. (depending on the nature of the data structure). Any such operation might involve a *re-organization cost*. The cost of an algorithm is the total cost that it pays for accessing and re-organization.

An offline algorithm knows the whole sequence of requests in advance. This knowledge, helps such algorithm to reorganize the list more effectively. For example, if there is a request to an item a, and an offline algorithm knows that there will be more requests to a in the near future, it will pay the reorganization cost to make a more accessible for the forthcoming requests. An online algorithm, however, does not have the future requests and should take its decisions with priory knowledge about request sequence.

[1] The structures that we study can be easily extended to support insertion/deletion. Nevertheless, these operations are not critical in terms of analysis of online algorithms.

2 List Update Problem

Consider you have a set of L items which are stored in a linked list of size L. Assume there is an online sequence of n accesses to these items. To access an item at position i in the list, we need to probe i items, and this results in an 'access cost' of i. After the access, the requested item can be moved closer to the front of the list at no additional cost (a *free exchange*). Moreover, at any time we can swap any two consecutive items at a cost of 1 (a *paid exchange*). Note that any number of paid exchanges can be made at any given time. The cost of an algorithm for a given sequence is the total access cost for all requests plus the total number of paid exchanges. The above model for list accessing yields to the classic *list update* problem. Although the assumptions behind this model are not necessarily valid for updating a linked list in practice, the list update problem is well accepted as the classic model for self-adjusting lists.

Move-To-Front (MTF) is a list update algorithm which, after accessing an item at position i, it moves the requested item to the front of the list using a free exchange. Move-To-Front (and most existing algorithms for list update) do not use paid exchanges. Timestamp is another online algorithm that uses a free exchange to move a request item x in front of the first item requested at most once since the last access to x (and does not make any move if such item does not exist). Transpose is another list update algorithm that, after accessing an item, moves it one unit closer to the front using a free exchange. We see more algorithms later in the section.

2.1 Offline algorithms

The offline version of list update is an NP-hard problem. In this section, we review the *static offline* algorithm, which works as follows. The algorithm counts the number of occurrences of each item in the request sequence, and sorts items in decreasing order of their occurrences (i.e., the item which is requested more frequently appear closer to the front of the list). More precisely, before starting to answer requests, the algorithm reorganizes the list so that items appear in the sorted order in the list. Such reorganization requires $\Theta(L^2)$ paid exchanges. To see that, assume we place items one by one in their correct position in the sorted list; this requires moving an item at index i in the sorted list using at most $L-i$ paid exchanges. This sums up to $\sum_{i=1} L(L-i) = L(L-1)/2$ paid exchanges. After sorting the list using paid-exchanges, the algorithm starts serving the sequence and in doing so, it never reorganizes the list.

Lemma 1. *The static offline algorithm has a cost of at most $nL/2 + O(L^2)$ to serve n requests on a list of length L.*

Proof. The cost for the initial reorganization (sorting) is $O(L^2)$ (the algorithm makes that many paid exchanges). Let n_i denote the number of requests to an item at index i in the sorted list; so we have $n_1 \leq n_2 \leq \ldots \leq n_L$. The access cost for the algorithm is

$$\begin{aligned} \mathit{Access\_Cost} &= n_1 + 2n_2 + \ldots + i \cdot n_i + \ldots + L \cdot n_L \\ &= (n_1 + L \cdot n_L) + (2n_2 + (L-1) \cdot n_{L-1}) + \ldots + (L/2 \cdot n_{L/2} + (L/2+1) \cdot n_{L/2+1}) \end{aligned}$$

If $n_1 = n_L$ then $n_1 + L \cdot n_L$ would be $(n_1 + n_L) \cdot L/2$. This is the maximum value for this formula because we know $n_1 \geq n_L$, i.e., among the $n_1 + n_L$ requests to the items at the front and back of the list, at least half of the requests (possibly more) are to the item located in the front. In conclusion we have $(n_1 + L \cdot n_L) \cdot L/2 \leq (n_1 + n_L) \cdot L/2$. We can make a similar argument for other pairs inside parentheses in the above equality to conclude:

$$\begin{aligned} \mathit{Access\_Cost} &\leq (n_1 + n_L) \cdot L/2 + (n_2 + n_{L-1}) \cdot L/2 + \ldots + (n_{L/2} + n_{L/2+1}) \cdot L/2 \\ &= (n_1 + \ldots + n_L) \cdot L/2 = n \cdot L/2 \end{aligned}$$

So, the cost for initial reorganization cost is $O(L^2)$ and the access cost is at most $n \cdot L/2$. Hence, the theorem. □

Intuitively, sorting items in decreasing order of their frequency results in an amortized access cost of $L/2$ per request. Note that the static offline algorithm is not necessarily an optimal algorithm. To see that consider the sequence

$$\langle \underbrace{a\ a\ \ldots a}_{x \text{ times}} \quad \underbrace{b\ b\ \ldots b}_{x+1 \text{ times}} \rangle$$

, where the initial list configuration is $a \to b$. For the above sequence, the static offline algorithm moves b in front of a for the first x requests, while an optimal algorithm keeps a in front and moves b to front *after* the first x requests.

Although finding an offline optimal algorithm is NP-hard for list update and many other problems, we often can associate properties to the optimal offline algorithm which are useful for analyzing the online algorithms. Here, we see one such property for list update:

Lemma 2. *There is an optimal offline algorithm for list update that uses only paid exchanges.*

Proof. Assume an optimal algorithm Opt uses a free exchange after accessing an item x at position i to move it closer to the front to position j ($j < i$). Note that the cost of Opt for accessing x is i.

We modify Opt in way that it does not use the free exchange and its cost remains the same. For that, just *before* accessing x, we make Opt to use $i - j$ paid exchanges to move x from position i to position j. This cost paid for these exchanges is $i - j$. After that, the algorithm accesses x at position j. The total cost paid for the request to x is $(i - j) + j = i$, which is the same cost the algorithm pays before the modification. Moreover, the list configuration remains the same before and after the modification, i.e., x is at position j after the serving the request (see Figure 1). So, we can repeatedly apply the above procedure to achieve an algorithm with the same cost as Opt in which all free exchanges are replaced by paid exchanges. □

(a) Before modification (b) After modification

Figure 1: Modifying an offline algorithm to avoid using free exchanges while maintaining the same cost.

2.2 Lower bound for deterministic algorithms

In this section, we show that no deterministic online algorithm for list update can have a competitive ratio better than 2. Our lower bound proof is based on a *cruel adversary* sequence, where the adversary looks at the actions of the algorithm, and based on the actions of the algorithm creates a worst-case sequence.

Theorem 1. *No deterministic list update algorithm can have a competitive ratio better than 2.*

Proof. Consider an input sequence in which the adversary always asks for the last item in the list maintained by any online algorithm A. So, the access cost of A is exactly $n \cdot L$ (there are n requests, each requires probing L items to access the requested item at the end of the list). The actual cost of the algorithm might be more than $n \cdot L$ (in case A uses paid exchanges). Regardless, the cost of A is at least $n \cdot L$. On the other hand, by Lemma 1, we know there is an offline algorithm with cost at most $n \cdot L/2$. In summary, for any online algorithm A, there is a sequence for which we can write $cost(A) \geq n \cdot L$ and $cost(\text{OPT}) \leq n \cdot L/2$. The ratio between the two costs is at least 2 which completes the proof. □

2.3 Competitive ratio of MTF

In this section, we use a general framework of *potential function* to prove an upper bound of 2 for the competitive ratio of Move-To-Front. This result, paired with Theorem 1, show that MTF is the optimal deterministic algorithm for list update (i.e., MTF has a competitive ratio of 2 and no deterministic algorithm can have a better competitive ratio).

2.3.1 Potential function

The potential function method is used to analyze many online algorithms. Assume that an online algorithm A and OPT are running at the same time on an input sequence σ. Before serving the t'th request, both A and OPT have a *state*. For example, in the case of list update, the state of A (resp. OPT) is the configuration of the list maintained by A (resp. OPT). Intuitively, it is desirable for A to have a state that is 'close' to that of OPT . Potential function is a way to quantify this 'closeness'. More formally, the potential function for an algorithm A, denoted by $\Phi(t)$, is a function of the states of both A and OPT when serving the t'th request. A high potential implies that the state of A is 'far' from the state of OPT while a small potential means the state of A is 'close' to that of OPT . So, it is desirable for A to have a small potential. Now, if A incurs a cost of c to serve the t'th request, and in the process of serving that request, decreases the potential by Δ, it has paid a cost of c and meanwhile has an advantage of Δ units in terms of potential. We define the *amortized cost* of A for serving the t'th request as

$$amortized\_cost(t) = actual\_cost(t) + \Phi(t+1) - \Phi(t)$$

.

In the above formula, $\Phi(t+1) - \Phi(t)$ can be positive (meaning that he potential has deteriorated) or negative (meaning that potential has improved). In summary, the amortized cost at time t sums the cost of the algorithm and the amount of increase in potential (which can be negative). Interestingly, in order to provide an upper bound for the competitive ratio of A, it suffices to bound the amortized cost. To see that, note that the cost of A is:

$$cost(A) = actual\_cost(1) + actual\_cost(2) + \ldots + actual\_cost(n)$$

Adding $\Phi(1) - \Phi(1) + \Phi(2) - \Phi(2) + \ldots + \Phi(n) - \Phi(n)$ and rewriting the above equation, we get:

$$\begin{aligned} cost(A) =& \Phi(1) + (actual\_cost(1) + \Phi(2) - \Phi(1)) + (actual\_cost(2) + \Phi(3) - \Phi(2)) \\ &+ (actual\_cost(3) + \Phi(4) - \Phi(3)) + \ldots + (actual\_cost(n) + \Phi(n+1) - \Phi(n)) - \Phi(n+1) \\ &= amortized\_cost(1) + amortized\_cost(2) + \ldots + amortized\_cost(n) + (\Phi(1) - \Phi(n+1)) \end{aligned}$$

Since the potential is a function of the state of the algorithm, it is independent of the length n of the input. So, for long inputs, the value of $\Phi(1) - \Phi(n+1)$ is a constant that can be ignored. So, if we can show that, for any sequence, the amortized cost of A at any time t is within a ratio c of the cost of OPT (i.e., we can show $\forall t \;\; amortized\_cost(t) \leq \text{OPT}(t)$, then we can write:

$$\begin{aligned} cost(A) &\approx amortized\_cost(1) + amortized\_cost(2) + \ldots + amortized\_cost(n) \\ &= c(\text{OPT}(1) + \text{OPT}(2) + \ldots + \text{OPT}(n)) = c \cdot cost(\text{OPT}) \end{aligned}$$

The above equation guarantees a competitive ratio of c for algorithm

In summary, in order to use the potential function method to derive an upper bound c for the competitive ratio of an algorithm A, we should take the following steps:

1 Define the potential as a function of the states of A and OPT at time t (before serving the t'th request). For most problems, the potential maps the states of A and OPT into a positive real value that is independent of the input length n. Finding the right potential function is often the tricky part in analysis of online algorithms. The potential should be defined in a way that we can prove the statement made in step 3.

2 Define the amortized cost at time t as the summation of the actual cost and the difference in potential before and after serving the t'th request, i.e., $amortized\_cost(t) = actual\_cost(t) + \Phi(t+1) - \Phi(t)$.

3 Assuming the potential is defined properly, we should be able to show $amortized\_cost(t) \leq c \cdot \text{OPT}(t)$, where $\text{OPT}(t)$ is the cost of OPT for serving the t'th request. If we can show this, we have proved a competitive ratio of c for A.

2.3.2 Competitiveness of MTF

We illustrate the potential function method by proving an upper bound of $c = 2$ for the competitive ratio of Move-To-Front. We follow the three steps mentioned in the previous section:

1 The first step is to define the potential function. Recall that the potential should be a function of the states of A and OPT and a higher potential should imply that these states are far from each other. Lets define the potential at time t as the number of *inversions* between the lists of MTF and OPT . An inversion is defined via two items x and y which appear in different orders in the lists maintained by MTF and OPT . Note that the number of inversions is maximized if all possible pair of items appear in different orders. The number of such pairs of a list of L items is $\binom{L}{2} = \frac{L(L-1)}{2}$ (which happens when the list of A is the inverse of the list of Opt). So, the potential is a number between 0 and $L(L-1)/2$. Note that initially MTF and OPT have the same state (list configuration) and hence $\Phi(0) = 0$.

2 In the second step, we should formulate the amortized cost of MTF. Assume at time t, an item x is requested. Assume x is at position i of the list maintained by MTF. So, for the actual cost of MTF we have $actual\_cost(t) = i$. For the amortized cost, we should count how much the potential is increased at time t as a result of actions of MTF and OPT . The increase in potential is the number of added inversions minus the number of removed inversions. So, for the difference in potential, we can write $\Phi(t+1) - \Phi(t) = no\_added\_inversions - no\_removed\_inversions$. In summary, the amortized cost of MTF at time t is $amortized\_cost(t) = i + no\_added\_inversions - no\_removed\_inversions$.

3 In this step we should show that the amortized cost of MTF is within a factor 2 of the cost of OPT at time t. Let's assume that OPT accesses the requested item x at position j of its list. After that access, OPT might use k paid exchanges to update its list. Note that by Lemma 2, we can assume OPT makes no free exchange and we do not need to worry about free exchanges. So, the cost of OPT will be $j + k$ (access cost of j and a total cost of k for k paid-exchanges). So, we can write $opt(t) = j + k$. Next, we have to see how potential is increased/decreased by the actions of MTF and OPT after serving the t'th request to x.

First, let's focus on action of MTF. Recall that MTF accesses x at position i and moves it to the front. We know that x is located at position j of the OPT 's list. So, among the $i-1$ items that appear in front of x in MTF's list, at most $j-1$ items also appear in front of x in OPT 's list. The other $(i-1)-(j-1) = i-j$ items that appear *before* x in MTF's list appear *after* x in OPT 's list. In other words, there are at least $i-j$ items which form an inversion with x when MTF accesses x. Moving x to the front removes all these inversions (because those $i-j$ items now appear after x in both lists). So we can write $no\_removed\_inversions \geq i-j$. Meanwhile, before moving x to front, there are at most $j-1$ items that appear before x in both lists of MTF and OPT (because x is at index j in OPT 's list). After moving x to front, these items appear after x in MTF's list and before x in OPT 's list. Hence, they form new inversions with x. In summary, the action of MTF (moving x to front) creates at most $j-1$ new inversions and removes at least $i-j$ inversions, i.e., we can write $no\_added\_inversions_{by-MTF} \leq j-1$ and $no\_removed\_inversions_{by-MTF} \geq i-j$.

For OPT , we know it makes $k \geq 0$ paid exchanges. Each paid exchange increases potential (number of inversions) by at most 1 unit. So, in the worst case, actions of OPT create k new inversions and remove no inversions. We can write $no\_added\_inversions_{by-OPT} \leq k$ and $no\_removed\_inversions_{by-OPT} \geq 0$.

In summary, after both MTF and OPT apply their actions, we have: $no\_added\_inversions \leq j+k-1$ and $no\_removed\_inversions \geq i-j$. So, for the amortized cost derived in step 2, we can write $amortized\_cost(t) = i + no\_added\_inversions - no\_removed\_inversions \leq i + (j+k-1) - (i-j) = 2j + k - 1$. Recall that the cost of OPT is $j+k$. So, the ratio between the amortized cost of MTF and cost of OPT is at most $\frac{2j+k-1}{j+k} \leq 2$. In other words, we have $amortized\_cost(t) \leq 2\,\text{OPT}(t)$. As mentioned before, it is sufficient for proving that MTF has a competitive ratio of 2.

From the above potential function argument, we can conclude the following theorem:

Theorem 2. *The competitive ratio of Move-To-Front is at most 2.*

2.4 Other deterministic algorithms

The results in the previous section show that MTF is the optimal deterministic algorithm for list update. However, there are other algorithms that we can consider. In this section, we review some

of them and show they are not as good as MTF.

Recall that Transpose is an algorithm that uses a free exchange to move an accessed item one unit closer to the front. So, if x is accessed at position i, it is moved to position $i-1$.

Theorem 3. *Transpose is not competitive.*

Proof. Consider a list of length m which is initially configured as $a_1 \to a_2 \to \ldots \to a_{m-1} \to a_m$. For a sequence $\sigma = \langle (a_m, a_{m-1})^k \rangle$, Transpose accesses each requested item at position m and incurs a cost of $k \cdot m$. The algorithm is clearly far from optimal because it keeps exchanging the last two items in the list. There is an offline algorithm that uses $m-1$ paid exchanges to move a_{m-1} to the front and $m-1$ exchanges to move a_m to the second position. After that, it does not rearrange the list. The cost of this algorithm will be $2m-2$ for paid exchanges and $3k$ for accesses (k for accesses a_{m-1} at the front and $2k$ for accesses to a_m at position 2). In summary, there is a sequence for which the cost of Transpose is $k \cdot m$ and the cost of OPT is at most $3k+2m-2$. As a result, the competitive ratio of Transpose is at least $\frac{km}{3k+2m-2} = \frac{km+(2m^2-2m)/3-(2m^2-2m)/3}{3k+2m-2} = m/3 - \frac{2m^2-2m}{9k+6m-3}$. The adversary can select k to be a large value so that the second term converges to 0. Hence, the competitive ratio of Transpose becomes at least $m/3$ for sufficiently long sequences. □

So, for a list of length L, Transpose has a competitive ratio of $L/3$. This is clearly not good for long lists. Next, consider MTF2, which is an algorithm that uses a free exchange to move a request item at position i half way to the front, i.e., right after position $\lceil i/2 \rceil$.

Theorem 4. *The competitive ratio of MTF2 is at least 4.*

Proof. Consider a list of length $2m$ which is initially configured as

$$a_1 \to a_2 \to \ldots \to a_m \to a_{m+1} \to a_{m+2} \to \ldots \to a_{2m-1} \to a_{2m}$$

Consider an input sequence $\sigma = \langle (a_{2m}, a_{2m-1}, \ldots, a_{m+1})^k \rangle$ which is formed by k 'rounds' of accesses to the items located in the second half of the list. To serve σ, MTF2 accesses each item at the end of the list, e.g., it accesses a_{2m} at position $2m$ and then moves it to right after position m. For the next request, a_{2m-1} is accessed at the end of the list and again moved half way to the front. Note that the length of σ is $n = km$. The cost of MTF2 for σ will be $n \cdot 2m = 2km^2$. An offline algorithm observes that $a_1, \ldots, a_m$ are never requested in σ and there is no need for them to 'block' other requested items. Hence, it uses m paid exchanges to move a_{m+1} to the front, m paid exchanges to move a_{m+2} to the second position, and more generally m paid exchanges to move a_{m+i} to position i. The cost of the offline algorithm for paid exchanges will be $m \cdot (m+1)$. After making these moves, the algorithm becomes a static offline algorithm (i.e., it never rearranges the list). The access cost of the algorithm for each of the k rounds will be $1+2+\ldots+m = m(m+1)/2$. In total the cost of the offline algorithm will be $m(m+1)$ for paid exchanges and $km(m+1)/2$ for accesses. In summary, we created a sequence σ for which the cost of MTF2 is $2km^2$ and the cost of OPT is at most $m(m+1) + km(m+1)/2$. The competitive ratio of MTF2 will be at least

$$\frac{2km^2}{km(m+1)/2 + m(m+1)} = \frac{4km^2}{km^2 + km + 2m^2 + 2m} = 4 - \frac{4km + 8m^2 + 8m}{km^2 + km + 2m^2 + 2m}$$

Adversary decides the list to be long $m \to \infty$ and lets k to be asymptotically larger than m, e.g., $k = \Theta(2^m)$. For this choice we have $4km + 8m^2 + 8m = \Theta(m \cdot 2^m)$ while $km^2 + km + 2m^2 + 2m \in \Theta(m^2 \cdot 2^m)$. As a result, for large values of m the denominator becomes asymptotically larger than the numerator, i.e., the second term converges to zero and the competitive ratio of MTF2 converges to 4. □

The above lower bound is tight and the competitive ratio of MTF2 is indeed 4 (we skip the proof here). So, MTF2 is competitive but it is not as good as MTF. Next, consider Move-To-Front-Every-Other-Access, where each item is moved to front on every-other-access to it. There are two variants of this algorithm, MTF-Even which moves an item x to front on even accesses to x (and does not make any move on odd accesses) and MTF-Odd which moves x to front on odd accesses to x (and again, does not make any move on even accesses). Unfortunately, these two algorithms are not a match for MTF. It is recently proved that the competitive ratio of these algorithms is 2.5.

Timestamp is an algorithm that indeed has the same competitive ratio as MTF. Recall that Timestamp moves a requested item x to the front of the first item y which appears before x and has been requested at most once since the last access to x. In case it is the first access to x or if such y does not exist, Timestamp does not make any move. In other words, the algorithm looks at the 'timestamp' between the current access to x and its previous x. Those items requested twice or more in this timestamp are in front of x in the second access to x because they will be moved in front of x on their second access in the timestamp. Let's call these items 'blue items'. Items that have been requested at most once in the timestamp might be locate before or after x in the list. Let's call these items 'red items'. At the time of the second request to x, all red items appear after all blue items; this is because on the second access to blue items in the timestamp, those are moved in front of red items, and in the potential one access to a red item, it is not moved in front of any blue item. After accessing x, Timestamp moves it to the front of all red items and just after all blue items. This ensures the following ordering for any two items x and y assuming x is more recently requested: if the last two accesses to y are 'sandwiched' between the two last accesses to x (i.e., we have $\langle x \dots y \dots y \dots x \rangle$, then y appears before x in the list of Timestamp; otherwise, x appears before y. Besides MTF and Timestamp, there are other online algorithms with competitive ratio of 2; however, the design of all these algorithms is somehow inspired by MTF or Timestamp or a combination of both.

2.5 Randomized algorithms for list update

We observed that no deterministic algorithm can achieve a competitive ratio better than 2. Meanwhile algorithms like MTF and Timestamp achieve this best-possible competitive ratio. So, the situation is somehow clear for deterministic algorithms (the magic number is 2). What about randomized algorithms? The lower bound argument that gives us a competitive ratio of at least 2 does not work for randomized algorithms. This is because the adversary does not know the outcome of the random bits used by the algorithm and hence it does not know the list configuration at each given time; hence, it cannot request the last item of the list as it did for deterministic algorithms. This suggests that a randomized algorithm for list update might indeed achieve a competitive ratio better than 2.

Let's start with a simple randomized algorithm that, upon accessing an item x, flips a coin and moves x to the front in case of a 'tail' (and does no make any move in case of a 'head'). One can show that this simple algorithm has a competitive ratio of 2. So, it does not give us any improvement over deterministic algorithms such as MTF.

BIT is another randomized algorithm which indeed results in better competitive ratios. The algorithms maintains a bit for each item in the list. The bit is initially set to '0' or '1' by flipping a coin for each item. Now, upon an access to an item x, the bit of x is flipped and in case it becomes '1', the item is move to the front (otherwise, it is not moved at all). It is known that BIT has a competitive ratio of 1.75, which is better than all deterministic algorithms. COMB is another randomized algorithm that uses a combination of BIT and Timestamp. At the beginning, it randomly chooses BIT with probability 0.8 and Timestamo with probability 0.20, and applies the

selected algorithm to serve the input sequence. COMB is the best existing randomized algorithm with a competitive ratio of 1.6. Can we do better than COMB? It is known that no randomized online algorithm has a competitive ratio better than 1.5. So, there is a gap between the competitive ratio of COMB and the best existing lower bound. This suggests that maybe there are better randomized algorithms; however, designing and proving such upper bounds seems to require better analysis techniques that the existing ones.

2.6 Projective property

Most existing list update algorithms have the so-called *projective property*. An algorithm is said to be projective if the relative order of any two items only depend of accesses to those items. In other words, you can describe under what condition x appears before y, and in your description, you only refer to requests to x and y (and no other item z). Let's see an example. In case of MTF, an item x appears before y if the last access to x is more recent than the last access to y. In this description, we did not discuss any other item y. As another example, recall from Section 2.4 that in the lists maintained by Timestamp, assuming item x is more recently requested than y, we can say y appears before x if the last two requests to y are sandwiched between the last two requests to x (otherwise, x appears before y). Again, in the above description, we did not discuss any other item except x and y. We conclude that both MTF and Timestamp have the projective property. As another example, Trasnpose is Not a projective algorithm. To show that, we provide an example where the relative order of x and y depend on items other than x and y. Consider the list $x \to y \to z$. Assume the requests are $\langle z\ y \rangle$. In this case, x remains before y. However, if we remove the request to z, then y will be moved to the front of x. So, the relative order of x and y depend on the request to z, and Transpose is not projective.

Most algorithms that are studied for the list update problem have the projective property. This is mainly because projective algorithm are much easier to analyze compared to non-projective algorithms. In particular, we can reduce their analysis to lists of length 2:

Theorem 5. *In order to show a projective list update algorithm A has a competitive ratio of c, it it suffices to show that A is has a competitive ratio of c for lists of length 2*

Unlike lists of arbitrary lengths, it is easy to devise an optimal offline algorithm for lists of length 2. As a consequence, finding upper bounds for competitive ratio of a projective algorithm is relatively easy. Almost all existing algorithms (MTF, Timestamp, MTF2, BIT, COMB, etc.) are analysed using the same technique. A survey on properties and the existing results for projective algorithms can be a potential topic for your research project.

2.7 Advice complexity of list update

The advice complexity of list update has been studied in one research paper. A follow up to that paper and improving its results is a potential research topic. Here, we just show how only 2 bits of advice can improve the competitive ratio of deterministic online algorithms. Consider the following deterministic algorithms: MTF, MTF-Even, and MTF-odd. We saw earlier that MTF has competitive ratio 2 while the other two algorithms have competitive ratio of 2.5. Although any of these algorithm has a cost twice or more than that of OPT for *some* worst-case sequences, it turns out that their worst-case sequences are different. For example, the worst-case sequence for MTF is not that bad for MTF-Even and vice-versa. Interestingly, for any sequence σ one of these algorithms handle σ pretty well:

Theorem 6. *For any sequence σ, the minimum cost that any of* MTF*, MTF-Even, and MTF-odd algorithms can achieve is within ratio* $1.6\bar{6}$ *the cost of* OPT *.*

In other words, for any sequence, we can select the best algorithm for that sequence, and this ensures that such algorithm has a competitive ratio of $1.6\bar{6}$. This is the best existing *approximation algorithm* for the list update problem, i.e., even if we know the whole input, no better algorithm is known. Interestingly, this result relates to the advice complexity of list update: for any sequence σ, we can indicate, using two bits of advice, which algorithm among MTF, MTF-even, and MTF-odd has a smaller cost for σ. Encoding that advice can help the online strategy to select the better algorithm and achieve a competitive ratio of $1.6\bar{6}$, which is better than the best competitive ratio that a purely-online algorithm can achieve.

3 List update & compression

One important application of the list update problem is in data compression. Assume you are given some data and you need to transform it into a compressed form so that the initial data can be decoded from the compressed data. We assume the input data is not random and has a low *entropy*. Informally speaking, information entropy is a way to measure the information content of data. Structured data (e.g., an English text) have lower entropies (are more 'predictable') and hence are more prone to compression. In summary, in data compression, we aim to encode an input data (often referred to as a 'text') in a compressed form.

One way to encode a text is to write down the ASCII or Unicode code for each character separately. This uncompressed encoding is quite wasteful because:

- Fixed-length problem: less frequent characters have the same code as more frequent ones (code for 'a' has the same length as 'q').
- Context problem: the 'context' of the text is ignored, i.e., the relationship between characters is overlooked ('qu' is encoded as a 'q' followed by a 'u', ignoring that a 'q' is often followed by a 'u').

To address the fixed-length problem, one can replace the ASCII code with Huffman code where more frequent characters are given shorter length. However, in order to fix the second problem a new approach is required.

Assume we have a list update algorithm A (for example, A can be MTF) and an input text T that we want to compress. We can start with a list that includes all characters that form T. Now, in order to encode T, we read the text character by character and write down the index of the character in the list (we assume indices start from 0). Meanwhile, we update the list using algorithm A. In order to decompress, given the indices that were written, we can start with the same initial list, access items at the encoded index, and update the list according to A. For example, assume the initial list is $a \to b \to c$ and $T = babac$. The encoded indices will be $1, 1, 1, 1, 2$. Similarly, given the same initial list and an encoding $0, 1, 1, 0$, we can decode text $T' = abaa$. The encoded indices have variable lengths: the larger numbers will need more bits while smaller numbers have smaller-length encoding; this can be achieved via a *self-delimited encoding*, where a character at index i is encoded using roughly $\log(i)$ bits. If we use MTF for maintaining the list, more frequent characters are generally closer to the front and hence have shorter codes. So, the fixed-length problem is addressed; however, we should still find a way to address the context problem.

Burrows-Wheeler Transform (BWT) of a text by permuting its characters so that they form 'runs' of repeated characters. For a structured text with repeated patterns (e.g., an English text),

the runs of repeated characters resulting from BWT are much easier to compress. BWT works as follows. Starting from a text T, we form all *rotations* of a text. The first rotation is made by moving the first character to the last position. Repeating this for the resulting rotation gives the second rotation, and this follows recursively. For a text of length m, there will be m rotations. Here, m includes a distinct character \$ which encodes the end of the text. Provided with all rotations, we sort them in lexicographic order (as words appear in a dictionary). BWT of the original text will be a new text formed by writing the last character of each rotation in the sorted order. Interestingly, provided by the BWT, it is possible to recover the original text. As an example, consider the text $\langle babada\$\rangle$. Forming the rotations and sorting them, we will have:

$$
\begin{array}{ll}
b\ a\ b\ a\ d\ a\ \$ & \\
a\ b\ a\ d\ a\ \$\ b & \\
b\ a\ d\ a\ \$\ b\ a & \\
a\ d\ a\ \$\ b\ a\ b & \xrightarrow{sort} \\
d\ a\ \$\ b\ a\ b\ a & \\
a\ \$\ b\ a\ b\ a\ d & \\
\$\ b\ a\ b\ a\ d\ a &
\end{array}
\qquad
\begin{array}{l}
\$\ b\ a\ b\ a\ d\ a \\
a\ \$\ b\ a\ b\ a\ d \\
a\ b\ a\ d\ a\ \$\ b \\
a\ d\ a\ \$\ b\ a\ b \\
b\ a\ b\ a\ d\ a\ \$ \\
b\ a\ d\ a\ \$\ b\ a \\
d\ a\ \$\ b\ a\ b\ a
\end{array}
$$

So, the BWT of the above text will be $\langle adbb\$aa\rangle$. Note that unlike the original text, the BWT has three runs of repeated characters. In general, if there is a pattern that is repeated in the original text (e.g., 'these' in English) when we sort the rotations, those starting with the same pattern but excluding the first character (e.g., rotations starting with 'hese') appear consecutively, and hence, in the last column, the first character (e.g., 't') makes a run of consecutive characters in the BWT.

The BWT creates an input with runs of consecutive characters. Consider we apply Mtf on such input. For any run of length m, all characters in the run except the first one are encoded as 0. This is because Mtf moves the requested character to the front and it remains there when encoding the subsequence $m-1$ characters. In summary, applying Mtf after BWT creates a compressions scheme in which the 'context' is translated into runs of consecutive characters (via BWT), and these runs of characters are encoded using short codes (via Mtf). As a result both fixed-length and context problems are addressed in this compressions scheme.

In fact, the popular BZip2 compression scheme works by: 1) applying BWT on an input text 2) applying Mtf on the output of BWT 3) encoding output of Mtf using the run-length encoding. The last step is added to further improve compression by encoding runs of repeated indices by encoding the index that forms the run as well as the length of the run. For example, if the indices recorder by Mtf form sequence $\langle 1\ 1\ 1\ 1\ 1\ 2\ 2\ 2\ 2\ 1\ 1\ 4\ 4\ 4\rangle$, the run-length encoding for these indices will be $\langle\ (1,5)\ (2,4)\ (1,2)\ (4,3)\ \rangle$.

BZip2 is one of the most commonly used compression schemes which gives promising results in many applications. As mentioned, the list-update algorithm behind BZip2 is Mtf. In principle, Mtf can be replaced by any *online* list update algorithm. Here, being online is critical since, in order to decompress a set of indices, the list update algorithms should update the list without any knowledge about the future requests (which are not decoded yet). Although other online algorithms can potentially replace Mtf, in practice, the high locality in the outputs of BWT give Mtf a huge advantage over other algorithms. Here, by 'locality', we mean 'temporal locality' which means items which are requested (appeared in the BWT output) are very likely to be requested again

(to form runs of characters)[2]. Regardless, it is still a great research topic to investigate whether other list update algorithms can achieve better compression schemes. One promising approach is to consider online algorithms with a small number of bits of advice. Here, the advice bits can be included in the compressed file and hence decompressions is possible. For example, assume we encode an input text with both MTF and Timestamp algorithm and realize that for this input MTF (resp. Timestamp) results in better compression. In this case we dismiss the encoding given by Timestamp (resp. MTF) and include 1 extra bit in the encoding of MTF (resp. Timestamp) to tell the decompression algorithm that indeed MTF (resp. Timestamp) is used to maintain the list for encoding the text. Provided with this extra bit, the decompresser can use the same algorithm to decompress the original. More generally, we can include k bits of advice in the compressed file to indicate which member of a give set of 2^k algorithms is used to encode the input text. Investigating this approach for compression is a promising topic for your final projects (in particular, if you are interested in coding).

4 Self-adjusting Binary Search Trees

List update formulates adjusting a linked-list to the patterns observed in an input sequence of accesses to items in the list. In this section, we consider the problem of adjusting a binary search tree (BST) to the patterns observed in the input. Recall that a binary search tree is a binary tree where each node has two pointers to its *left child* and *right child*. Any of these pointers can be NULL. In case both are null, we call the node a *leaf*; otherwise, it is an *internal* node. The *root* of the tree is the unique node which is not the child of any other node. You have previously seen examples of BSTs in form of AVL trees. These trees 'adjust' the list via 'rotations' to ensure the tree is 'balanced' in which the height (the maximum distance of any node from the root) is $\Theta(\log N)$ for a tree of N nodes. The logarithmic height ensures that accessing any item takes $\Theta(\log N)$ time per access in the worst case, which is the best possible. However, AVL trees (and similar trees such as red-black trees) ignore the patterns observed in the text. In particular, if there is a temporal locality (when the same node is repeatedly requested) probing a logarithmic number of nodes might be required *for each access*. However, if one adjust the tree so that the requested item is 'closer' to the root, then making k accesses to the same node can take $\Theta(\Theta(k))$ time instead of $\Theta(k \log N)$ of an AVL trees.

So, in a self-adjusting BST, we would like to 'adjust' the tree to patterns observed in an online sequence of requests to items in the tree. But, how can we adjust the tree? Recall that in the list update problem, we used free and paid exchanges to change the configuration of a list. In the case of BSTs, we use 'rotations' to adjust the list. A single rotation involves an internal node x and one of its children y. The idea is to change the structure of the tree so that y becomes the parent of x, while still maintaining the correct order required by the BST. A double-rotation is a sequence of two rotations which will put three vertices in a required order (a more precise definition of rotations will be followed). In a self-adjusting BST, when there is a request to an item x, an algorithm looks for x in the same way that it does in an ordinary BST. If there are i probes for accessing x (i.e., x is found at depth i), there will be an access cost of i for the request to x. Meanwhile, we can make any number of rotations on the path from the root to x at a cost of 1 per rotation.

It is not hard to think of different algorithms to update a BST in a same way that we update a list in the list-update problem. For example, we can use a algorithm similar to Transpose to update a BST, i.e., we use a single rotation to move x one node closer to the root. While this algorithm

[2]There is another type of locality called *spatial locality* which implies items that are 'close' to each other are likely to be requested together. We see examples of this type of locality later in the course.

greedily tries to reduce rotation-cost, one can show that not only its competitive ratio is not a constant, but also it is a function of the input-length n (as opposed to tree-size N which is much smaller). So, such algorithm does not work. A natural question is that, what is the 'equivalent' of MTF algorithm for BSTs. The answer to this question is an interesting type of BSTs known as *splay trees*.

4.1 Splay trees

The idea behind splay trees is to exploit the input locality by moving the recently requested items (and those which are spatially close to them) close to the root. The main ingredient is that the root should always be the most recently accessed item (in the same way that in the MTF, the first node is the most recently accessed item). So, when there is an access to an item x at depth d, the splay tree algorithm makes roughly d rotations to 'bubble-up' x all the way to the root of the tree. Interestingly, in this process, nodes which are close to x (e.g., its parent) are also moved closer to the root of the resulting tree. In what follows, we describe how an splay trees bubbles-up x to make it the root of the adjusted tree.

Assume there is an access to an item x. If x is already the root of the tree, there is nothing to do. Assume x is the child of the root. We need a single rotation, called a *zig* operation to update the tree. Assume x is the left (resp. right) child of the root p and let X and Y denote the left and right children of x while Z is the other child of p (they might be empty). In the adjusted tree, we let p be the right (resp. left) child of x. Meanwhile, we change the pointers of X, Y, Z to appear in the same order in the adjusted tree but with different parents (see Figure 2a). The resulting tree has x as its root.

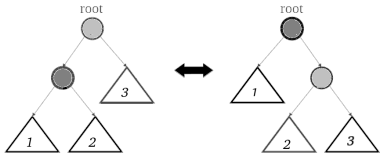
Next, assume x is not root neither a child of root. We apply a sequence of double-rotations so that after each rotation x becomes one unit closer to the root. After enough rotations, it will be a child of root and we can fix the tree with a zig operation. Let p denote the parent of x and g be the parent of p (both p and g exist because x is not root or a child of root). There are two cases to consider.

- First, assume x is smaller (resp. larger) than both p and g. In the adjusted tree, we make x replace g on 'top' and p and g both appear on its right (resp. left). Meanwhile, the four involved subtrees (two children of x, the other child of p and the other child of g) are placed accordingly so that the tree stays a BST. This operation is called a *zig-zig operation* (see Figure 2b).

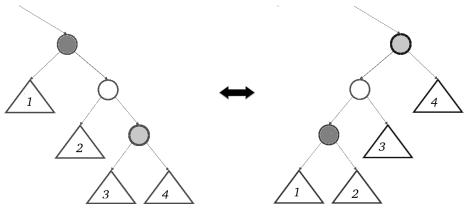
- Second, assume x is in between p and g (i.e., $p < x < g$ or $g < x < p$). In the adjusted tree, we make x replace g on 'top' and p and g each appear on one side of x (depending on their relative order). As before, the four involved subtrees (two children of x, the other child of p and the other child of g) are placed accordingly so that the tree stays a BST. This operation is called a *zig-zag operation* (see Figures 2c,2d).

In summary, after an access to x at depth d, the splay tree makes roughly d rotations to move x to the root. This operation is called an *splay*. So, the total cost for accessing an item at depth d is $2d$ (half of which is for accessing and the other half for splaying).

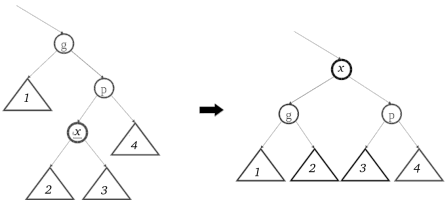
Splay trees were introduced by Sleator and Tarjan. It is not hard to use a potential function argument to show the competitive ratio of splay trees for the BST-update problem is $O(\log N)$ where N is the size of the tree. In other words, the competitive ratio of splay tree is *at most* logarithmic to the size of tree. But, can it be better? Sleator and Tarjan made a famous conjecture that splay trees have indeed a constant competitive ratio. This conjecture is still open after a few decades.



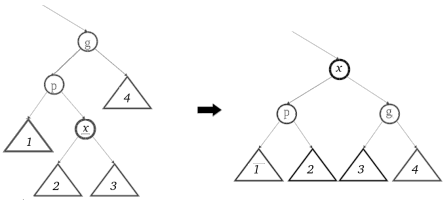
(a) zig operation; x is a either of the grey nodes which is moved above to become root.



(b) zig-zig operation; x is a either of grey nodes which is moved closer to the root



(c) zig-zag operation when $p < x < g$



(d) zig-zag operation when $g < x < p$

Figure 2: Splay rotations to move an accessed item x closer to the root.

Many people believe it is true; but we don't know for sure. The best existing trees (algorithm) for the BST-update problem is given by Tango trees[3]. Tango trees have a competitive ratio of $\Theta(\log \log N)$ for a tree of size N. It is still unknown whether there is a self-adjusting BST with constant competitive ratio.

5 Remarks

As a topic for your project, you can consider implementing existing self-adjusting BSTs such as splay trees, Tango trees, and other trees that are inspired by list-update algorithms (such as 'Transpose trees' that we discussed earlier). You can run real-world queries to compare the performance of these trees in the *average-case*. Recall that our discussions about competitive ratio mostly cover worst-case scenarios; it is likely that an algorithm with a bad competitive ratio works well in practice and vice versa.

As another topic for your research project, you can investigate how advice can help improving competitive ratio of self-adjusting BSTs. For example, how a splay tree can exploit some bits of advice to improve its competitive ratio? How many bits of advice are sufficient to achieve an optimal algorithm for the BST-update problem?

In case you like coding, you can investigate the application of self-adjusting BSTs in data compression. Can we replace MTF with splay tree when compressing a given text? Yes, it is indeed possible to develop compression schemes which update a tree instead of a list. How well they work in practice? What self-adjusting tree works better? It is a question that you can investigate.

In this section, we were focused on self-adjusting linked-lists and BSTs. What about other data structures? For example, can you update a linked-list to adjust it to patterns in the input sequence? This can be a topic for your research project.

[3]Authors of the paper that introduced Tango trees discussed the idea and came up with these trees in Buenos Aires. Hence, the name.