CSCI 6057

Advanced Data Structures

He

Alex Safatli

Monday, September 8, 2014

Contents

Introduction

Relevance

The course website is located at `http://www.cs.dal.ca/~mhe/csci6057`. Check announcements on this website regularly. Why take this course? Advanced data structures is typically used in algorithm and theory research, a major research area in algorithms. Furthermore, performance issues in systems and applications may often require their use (databases, information retrieval, geographic information, bioinformatics, etc.).

Topics in this course are found on the syllabus. Three assignments will be released. A project will include a proposal, report, and presentation. A research topic related to this course should be chosen. This could be a problem from your research requiring advanced data structure techniques.

Note-taking is required and will be enough for assignments. No textbook is required. Research papers will be given online for each topic where you may read if interested. These are difficult to read; do not use them as an alternative to lectures.

Amortized Analysis

This course is about designing data structures to store content and support certain operations. We also want to prove they are correct and have a certain complexity and performance. Normally, we can talk about worst-case, average-case, and best-case analysis. However, we usually assume the worst- or, sometimes, the average-case.

Knowing these complexities, the actual time will not be able to be determined. All that is known is a uniform behaviour and distribution of input. Considering these cases give either *guarantees* (worst case): it could do better but *no worse*. Or it could provide a probablistic sense of the performance (average case).

Any black box data structure will (1) store data and (2) have operations. When using a data structure, usually do not care about worst cost case of data structure implementations. How many applications ask a *single* query (which is where one would care about a single implementation and its cost). Otherwise, data structures are used to accomplish non-trivial operations inside a larger scale algorithm — a series of operations are performed on that data structure. All that matters is that the algorithm, as a whole, performs well on average.

Goal. UNDERSTAND THE COST OF SEQUENCES OF OPERATIONS. Want to give worst-case guarantees on their cost. This does not mean we want worst-case costs for *individual* operations.

Why not aim for a data structure with all individually efficient operations? The point is that building good data structures with well-performing sequences of operations takes *less effort* and is easier to do than those with efficient individual operations. Even if this was achievable, the implementation is most likely more complicated and have larger constant factors.

Given some data structure D, we can consider a set of comprising operations $o_1, o_2, ..., o_k$ with corresponding worst costs c_i and amortized costs $\hat{c}_i$. The **amortized cost** of an operation has little to do with the cost of operation, but only that the following relations are true. T is the total cost of all of these operations and $n_1, n_2, ..., n_k$ is the number of operations you would see in some input sequence.

$$T \leq \sum_{i=1}^{k} c_i n_i \tag{1}$$

$$T \leq \sum_{i=1}^{k} \hat{c}_i n_i \tag{2}$$

Amortized Analysis continued on next page...

In order to guarantee the latter equation, we bound all amortized costs to some average. A given operation on this data structure has a certain amortized cost that only needs to be upper bounded.

There are three methods of amortized analysis used in order to prove the order of complexity for a sequence of operations in a data structure.

1. **Aggregate Analysis**,
2. **Accounting Method**, and
3. **Potential Method**.

The first two are not used as frequently as the **Potential Method**.

Example I. A STACK has a `push` ($O(1)$), `pop` ($O(1)$), and `multipop(k)` operation. The `multipop(k)` operation removes `min(k,`size of the stack`)` ($O(k)$). Given some sequence of n operations, the largest `multipop` has parameter $k_{\max}$. Therefore, the cost of the sequence of operations is $O(k_{\max} n)$. We want to prove this is actually $O(n)$ for the worst case. *What we are proving here is that the amortized cost for a push operation is* $O(1)$ *and that the pop and multipop operations have a zero amortized cost.*

Solution. We want to prove that n operations take $O(n)$ time in the worst case. This, in particular, implies that the *amortized cost* per operation is $\frac{O(n)}{n} = O(1)$. A multipop *could* be translated into a number of pop operations based on how many items it removes. The cost of every pop or multipop operation is equal to $O(1)$ to check that the stack is empty plus how many items are removed. However, items can only be removed if it has been pushed. So, this is at most $O(n)$ because this would be equal to $O(n) + O(n)$.

Aggregate Analysis. More formally, we first consider that there are n operations, $\leq n$ push operations valued at $O(n)$. Therefore, the cost per pop and multipop operation is $= O(1 + \text{num. removed elements})$. It is important to note that there is $\leq n$ pop and multipop operations. *Every element that is removed must be added first.* This means $\leq n$ push operations implies an addition of $\leq n$ elements and removal of $\leq n$ elements. Therefore, the total cost of pop and multipop operations is $O(n)$.

In order to conclude from this analysis that the amortized cost per operation is constant, we have to assume the stack is initially empty. Otherwise, we can immediately start with a large amount of data. This is an aggregate analysis because we looked at the total number of operations and then divided by the number of operations.

Accounting Method. Again, consider the three operations. Notice that the push operation adds an x to the stack, which we can assign a unit of credit to. This is charging an extra cost of credit in order to charge for it to carry the item along with it. The amortized cost can still be defined to be $O(1)$ = the cost of the operation and credit assigned which are constant.

The pop operation will remove the credit of the element it removes to pay for its own cost. Therefore, the amortized cost of this operation is zero. A multipop operation, on the other hand, removes k elements, and removes an amount of credit associated with each of them. Again, this is an amortized cost of zero. Therefore, the only thing we need to argue now is that there is never an element with negative credit. When performing an operation, we must either add a certain amount of credit or remove credit that exists. There is no operation that takes credit away other than one that removes it. Therefore, this guarantees there will never be negative credit.

Potential Method. To illustrate this technique, we revisit the three operations. Recall each operation has an actual cost c_i. In many ways, this is probably the clearest method but most likely the hardest to apply. We define the *potential* of the data structure as Φ, $\Phi \geq 0$. Given the sequence of operations $o_1, o_2, ..., o_n$, Φ_i = potential of data structure after the first i operations. Then, $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$.

Notice that we satisfy:

Amortized Analysis continued on next page...

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0 \mid \Phi_n - \Phi_0 \geq 0 \tag{3}$$

In this problem, Φ = the number of elements in the stack. The push operation has an actual cost $\leq c$, some constant, the difference $\Delta\Phi = +c \implies$ amortized cost of the push operation $\leq 2c = O(1)$. In the pop operation, there is a difference of $-c$, which is still an amortized cost = 0. For the multipop function, the actual cost is $\leq c \times \min(k, \text{number of elements on stack})$, and $\Delta\Phi = -c \times \min(k, \text{number of elements on stack}) \implies$ an amortized cost = 0.

Example II. A BINARY COUNTER can be used to track a value. For example, 010111 is 47. Incremented, this becomes 0110000 is 48 with k bits. How can this be implemented? This is a carry: 0101111 + 0000001, moving the carry along. Walk until you hit the first zero and flip the other bits. The worst case per increment would be $O(k)$ (the number of bits). With n increments, this would be $O(nk)$. Again, starting with a counter value of 0, n increment operations can also be proven to be $O(n)$ in cost. *What we are proving here is that the amortized cost for an increment operation is* $O(1)$.

Solution. We again have to assume an initially zeroed counter (a counter starting at zero). In this counter, we will have a sequence of increment operations that occurs similarly to the below.

0000, 0001, 0010, 0011, ...

Aggregate Analysis. The first bit (on the right) will flip n times. The second will flip $n/2$ times. The third will flip $n/4$ times, and so on. Therefore, the total number of bit flips is $\leq n\sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$. The number of bit flips on average can be considered to be 2 bit flips. In this regard, the amortized cost of a given operation is related to its number of bit flips, which is constant.

Accounting Method. Let us give our bit flips amortized costs so that the cost of operation is the sum of the number of bit flips it performs. Then, we can argue that there is only one bit flip that has a non-zero cost which is constant. Let us define a bit flip operation as being either a transition of type (a) $0 \rightarrow 1$ or (b) $1 \rightarrow 0$. Type (a) is assigned a credit ($O(1)$) but (b) takes one of these credits (cost zero).

Therefore, the amortized cost of an increment operation = $\sum$ amortized costs of its bit flips = 2 = $O(1)$ because there is only ever a single type (a) bit flip during an increment.

Potential Method. Let us define the potential method Φ = the number of 1s in the binary counter. Realize that the counter is always non-negative. Moving from $0 \rightarrow 1$ is free with this postulation. The only thing the potential cannot tend to is the transition from $1 \rightarrow 0$. Notice that the amortized cost per increment is therefore $O(1)$.

If we define the cost of an operation as the number of flipped bits, $\Delta\Phi = -(\text{number of flipped bits} - 2)$. The amortized cost is = cost + $\Delta\Phi$ = 2 = $O(1)$.

Example III. BINARY SEARCH. Given a sorted array, we search for an element by repeatedly halving the scope of the search. After a logarithmic number of steps, can discern if an element is being found is in the array or not. This is $O(\log_2 n)$ in the worst case.

If we want to insert a new element in a sorted array, either all elements are shifted on the left or on the right by one position, which is a linear insertion cost. This is why binary trees form a reasonable marriage between linked lists and sorted arrays.

A red-black tree would have an $O(\log_2^2 n)$ search complexity in the worst case and a slow insertion operation. We want to amortize this insertion as $O(\log_2 n)$.

Amortized Analysis continued on next page...

Let us imagine a data structure where the number of elements n be represented as a binary integer where its corresponding binary elements are represented as a sorted set of a number of elements corresponding to that element. No requirement is necessary indicating these are larger or smaller sets of values relative to each other. A search would go through all of these sets of elements, of which are consecutively easier to search. If the largest set takes $\log_2 n$ time to search, the next smallest set would take $\log_2 \frac{n}{2}$ time.

Incrementing the size of the structure increases a "binary counter". Every time we visit a level, we merge if an existing list is located at that level and construct a new level, corresponding to the new binary integer. For example, $n = 10111011 \rightarrow 10111100$. A merging of sorted lists can be done in linear time. The worst case cost for an insertion is $O(n)$ because the largest merging dominates. The total cost is $\leq$ cost of the last level $\times \sum_{i=0}^{\infty} \frac{1}{2^i} = 2$.

Solution. The potential of an element x, $\Phi_x = \log_2 n -$ level that x is stored at ≥ 0. The total potential $\Phi = \sum_x \Phi_x$. The initial insertion of an element provides the amortized cost $O(\log_2 n)$ since everything following would have a zero amortized cost, each successively paying for merge costs.

Notice this can easily be transformed to an Accounting Method solution if we consider the addition of a credit $\log_2 n$ with an operation of inserting an element. To solve this using Aggregate Analysis, we consider **Example II**. Realize the association with a binary counter. Each position flips some amount of times related to n, as specified in the solution above. There are $\log_2 n$ levels, each with an $O(n)$ cost $\implies O(n \log_2 n)$ cost operations divided by $n \implies O(\log_2 n)$ amortized cost of the operation. One could argue by proving each element is only involved in one operation per level.

Note. If what we want is a priority queue, a data structure like this could be used where a heap property is maintained between levels. Using this data structure, one could prove that without knowing cache sizes, an algorithm can achieve almost the optimal number of cache misses for priority queues (limit disk accesses).

Resizable Arrays

Arrays are fundamental data structures used for random access, given an index – elements can be acquired in $O(1)$ time. Furthermore, they are compact and require little space overhead when elements of the same type are stored. A **resizable array** A represents n fixed-size elements, each assigned an index from $[0..n-1]$. This type of data structure supports the following operations:

- *Locate*(i): Determine the location of $A[i]$. Returns a memory location and allows reading or writing.
- *Insert*(x): Store x in $A[n]$ (appending an element), incrementing n by 1.
- *Delete*(): Discard $A[n-1]$, decrementing n by 1.

Solution I. Simple solutions are found in the C++ standard library as `vector` and in Java as `ArrayList`. An even simpler strategy or solution will not implement the *Delete* operation. The strategy for such a solution would be to allocate a set block of memory, say an array. Upon inserting an element into a full array, we allocate a new array with twice the size as the old one, deallocating the old block.

Let n be the number of elements in A. Denote s as the maximum number of elements that can be stored in A. One could also say that A has s slots. We can also describe the **load factor** of A as α.

$$\alpha = \begin{cases} n/s, & \text{when } A \text{ is nonempty} \\ 1, & \text{otherwise} \end{cases} \tag{4}$$

The following are pseudocode implementations of the *Insert* operation.

```
Insert(x)
  if s = 0
    allocate A with 1 slot
    s <-- 1
  if n = s
    allocate A' with 2s slots
    copy elements of A into A'
    deallocate A
    A <-- A'
    s <-- 2s
  A[n] <-- x
  n <-- n + 1
```

Notice that α is always $\geq \frac{1}{2}$. It is equal to $\frac{1}{2}$ as insertion is being done on a full array block. We can now follow with an analysis of this solution.

The real cost of the ith *Insert* operation (linear in number of elements copied) has two cases (Equation 5).

$$c_i = \begin{cases} 1, & \text{if } A \text{ is not full before the } i\text{th insert} \\ i, & \text{otherwise} \end{cases} \tag{5}$$

An **amortized analysis** (potential analysis) is as follows. Let A_i denote A after the ith *Insert* operation. We can suppose the potential function $\Phi(A_i) = n$, the number of elements. Therefore, $\Phi(A_i) - \Phi(A_{i-1}) = 1$. However, this means it is still linear time; the potential is never really ever being released. We can also try $\Phi(A_i) = n - s$. This would not work because the potential function can be negative.

Instead, let $\Phi(A_i) = 2n - s$. This will never be negative because s is only at most $2n$. At the beginning, $\Phi(A_0) = 0$ and $\Phi(A_i) \geq 0$ for all i. To analyze the amortized cost, $a_i = \hat{c}$, of the ith insertion, let n_i denote the number of elements stored in A_i and s_i denote the total size of A_i.

There are two cases.

CASE 1. The ith insert does not require a new array to be allocated. Then, $s_i = s_{i-1}$. Thus, $a_i = c_i + \Phi(A_i) - \Phi(A_{i-1}) = 1 + (2n_i - s_i) - (2n_{i-1} - s_{i-1}) = 1 + (2(n_{i-1} + 1) - s_{i-1} - (2_{n-1} - s_{i-1}) = 3$.

CASE 2. The ith insert requires a new array to be allocated. Therefore, $s_i = 2 \cdot s_{i-1}$ and $s_{i-1} = n_{i-1}$. Notice $c_i = n_i$ here. Therefore $a_i = c_i + \Phi(A_i) - \Phi(A_{i-1}) = (n_{i-1} + 1) + (2(n_{i-1} + 1) - 2n_{i-1}) - (2n_{i-1} - n_{i-1}) = 3$.

Given that these are both constant, the *Insert* operation requires $O(1)$ amortized time.

Solution II. To support both *Insert* and *Delete* operations, realize that we need to reduce the memory waste of the latter operation. When α is too small, we allocate a new smaller array, and copy the elements from the old array into the new one.

A strategy that *would* work but *not achieve constant amortized time* is to double array size when inserting into a full array *and* halve the array size when a deletion would cause an array to be less than half full. Why?

Proof. By *Counterexample*. Want to show that for any n, construct a sequence of operations that would require $\omega(n)$ time in total. For simplicity, assume n is a power of 2. The first $n/2$ operations are all insertions, which means that the array is full. The next $n/2$ operations are *Insert*, *Delete*, *Delete*, *Insert*, *Insert*, *Delete*, *Delete*, ... The total running cost for this sequence of operations would be $\Theta(n^2) = \omega(n)$. This is a standard generalization where we

split into two classes such that $n' = n^{\lfloor \log_2 n \rfloor}$ so that $n/2 < n' \leq n$. □

The actual solution will involve still doubling the array size when inserting into a full array. However, we halve the array size when a deletion would cause the array to be less than $\frac{1}{4}$ full. To analyze this, let α_i be the load factor after the ith operation. Consider the following potential function.

$$\Phi(A_i) = \begin{cases} 2n_i - s_i, & \text{if } \alpha_i \geq 1/2 \\ s_i/2 - n_i, & \text{if } \alpha_i < 1/2 \end{cases} \tag{6}$$

Notice two cases.

1. The ith operation is *Insert*.
   (a) $\alpha_{i-1} \geq 1/2$ and a new array is allocated,
   (b) $\alpha_{i-1} \geq 1/2$ and no new array is allocated,
   (c) $\alpha_{i-1} < 1/2, \alpha_{i-1} < 1/2$, no new array is allocated,
   (d) $\alpha_{i-1} < 1/2, \alpha_i \geq 1/2$, no new array is allocated.
2. The ith operation is *Delete*.
   (a) $\alpha_{i-1} < 1/2$ and a new array is allocated,
   (b) $\alpha_{i-1} < 1/2$ and no new array is allocated,
   (c) $\alpha_{i-1} \geq 1/2, \alpha_i \geq 1/2$, no new array is allocated,
   (d) $\alpha_{i-1} \geq 1/2, \alpha_i < 1/2$, no new array is allocated.

We will more carefully prove 2(a). At this point, $n_i = n_{i-1} - 1$, $s_{i-1} = 4n_{i-1}$, and $s_i = \frac{s_{i-1}}{2} = 2n_{i-1}$. The load factor $\alpha_i = n_i/s_i < 1/2$, as from the definition. The amortized cost $a_i = c_i + (\Phi(A_i) - \Phi(A_{i-1})) = n_i + (s_i/2 - n_i - (s_{i-1}/2 - n_{i-1}))$. If we represent everything by n_{i-1}, then we get: $a_i = 0$.

Those that implement this in standard libraries for languages, it is implemented in a similar fashion to this. This is from the *Introduction to Algorithms* textbook. This is a constant amortized cost and linear-time space cost solution. Can we improve this?

To go beyond these solutions, we must formally define the commonly-used term **overhead**. The overhead of a data structure is the amount of memory usage *beyond* the minimum necessary to store its n elements.

The resizable array, which we can also refer to as a VECTOR, we can specify its overhead to be $\Theta(n)$. If we have n elements, in the worst-case, the *total space usage* is $6n$ (anytime during execution) plus any constant terms for bookkeeping. To get the overhead, we subtract n. Recognize the importance of the use of caches or GPU memory for high-performance computing.

There is an important **lower bound** to identify here. At some point in time, $\Omega(\sqrt{n})$ overhead is necessary for any data structure that supports insertion of elements or the location of any of these elements, where n is the number of elements currently stored in the data structure. This is an incredibly strong claim that applies to resizable arrays. It also applies to many other data structures and is a very general result.

Proof. Consider the following sequence of operations for any n: *Insert*(a_1), *Insert*(a_2), ..., *Insert*(a_n) where a_i refers to element i in the array A. After *Insert*(a_n), let $k(n)$ be the number of memory blocks (contiguous space of memory) occupied by the data structure, and $S(n)$ be the size of the largest of these blocks. Since any element can be located, all elements must be stored in memory. Hence, $S(n) \cdot k(n) \geq n$.

At this time, the overhead is at least $k(n)$. Memory addresses (bookkeeping information of some sort) have to be stored somewhere for the blocks that have been allocated (e.g., a linked list node). Furthermore, immediately after allocation of the block of size $S(n)$, the largest block, the overhead is at least $S(n)$ – this space has been allocated but nothing has been placed into it yet.

Therefore, the worst case overhead is at least $\max(\{S(n), k(n)\})$. Because $S(n) \cdot k(n) \geq n$, the overhead is at least, at some point, $\sqrt{n}$ $\square$.

NOTE For a resizable array, $k(n) = \Theta(1)$. After execution, all elements are in a single block. $S(n) = \Theta(n)$, a single large block that is at least a quarter full. A linked list, $k(n) = \Theta(n)$ and $S(n) = \Theta(1)$ (the size of the data).

How do we reduce the overhead for the vector as described in **Solution II**?

Solution III. Given the RAM model, let us define an operation *Allocate(S)* that allocates a block of size S ($O(1)$). Let us also define an operation *Deallocate(B)* that deallocates a block B ($(O(1)$), and *Reallocate(B,S)* that attempts to resize block B to size S. If not, it will allocate a block of size S and copy the content of B to the new block, deallocating B in the process ($O(S)$). The operation returns the resulting block (e.g., `realloc` from C).

Two solutions that *do not work*:

- Trying to store the elements in $\Theta(\sqrt{n})$ blocks where the size of the largest block is $\Theta(\sqrt{n})$. Imagine blocks $B_1, B_2, ..., B_k$ where block B_i's size is i. The total number of elements in the first k blocks is $\frac{k(k+1)}{2}$. The number of blocks required to store n elements is $\leq k(k+1)/2$ which is $\lceil(\sqrt{1+8n}-1)/2\rceil = \Theta(\sqrt{n})$. ISSUE. For *Locate(i)*, the element i is in the $\lceil(\sqrt{1+8n}-1)/2\rceil$th block. The square root, in the RAM model, cannot be done in $O(1)$ time (usually done via Newton's Method).
- Have a set of blocks B_0, B_1, B_2 where the size of B_i is equal to 2^i. For *Locate(i)*, notice that $i+1$ in binary (1,10,11,100,101,...) possesses a number of digits (without leading zeroes) equal to one more than the block number that item is located. There is a machine operation that allows one to find the most significant bit this in constant time. ISSUE. Size of the largest block is $\Theta(n)$ – the largest block is sized more than half of n.

Imagine the following data structures. An **index block** is a resizable array (using the vector **Solution II**) which contains memory addresses of *data blocks* (with a space overhead that is linear in the number of blocks). **Data blocks** (DB_0, DB_1, ..., DB_{d-1}) contain data (contain elements of the array) and are not contiguous.

Another levelling of grouping exists at a **Superblock** level (SB_0, SB_1, ..., SB_{s-1}). Data blocks are grouped into superblocks conceptually (not physically assigned). Furthermore, data blocks in the same superblock are of the same size. When superblock SB_k is fully allocated, it consists of $2^{\lfloor k/2\rfloor}$ data blocks, each of size $2^{\lceil k/2\rceil}$. The size of SB_k is therefore $2^{\lfloor k/2\rfloor+\lceil k/2\rceil} = 2^k$. Only SB_{s-1} might not be fully allocated.

LOCATE. Considering this, we can go about computing *Locate(i)* as follows. Let r denote the binary representation of $i+1$ without leading zeroes. The ith element is element e of data block b of superblock k. The value $k = |r| - 1$ where $|r|$ is the number of bits in r (see the second solution that does not work above). If the leading 1 is ignored in r, b is the $\lfloor k/2\rfloor$ bits of r immediately after the leading 1-bit. The remaining $\lceil k/2\rceil$ bits of r is e.

This does not immediately give the location of the block; must still refer to the index block and do additional work to find the index of the data block amongst all other data blocks. If we know that the number of data blocks in superblocks before SB_k (Equation 7), we can merely add b to this value and acquire everything that we need.

$$p = \sum_{j=0}^{k-1} 2^{\lfloor j/2\rfloor} = \begin{cases} 2\cdot(2^{k/2}-1), & \text{if } k \text{ is even} \\ 3\cdot 2^{(k-1)/2}-2, & \text{otherwise} \end{cases} \tag{7}$$

Notice that this can be done with a single machine operation (the *shift* operation). The location of the element e is therefore in DB_{p+b}. This is $O(1)$ time.

INSERT. Some easy-to-maintain bookkeeping information are: n, s, d (number of non-empty data blocks), the number of empty data blocks (0 or 1) – the data structure is allowed to possess this, the size and occupancy of the last nonempty data block, the last super block, and the index block. These are maintained in some fashion. The index block, again, is particularly maintained as a vector. When performing an *Insert(x)*, (a) if the last superblock SB_{s-1} is full, increment s (these are merely concepts), (b) if the last data block DB_{d-1} is full, and there are no empty data blocks, increment d and allocate a new data block as DB_{d-1}, (c) if DB_{d-1} is full and there is an empty data block, increment d create a new empty data block DB_{d-1}, and (d) store x in DB_{d-1}. Notice that insertion never *creates an empty data block*; it will always fill a block.

DELETE. The last element is removed. Several situations may arise. (a) If DB_{d-1} is empty, decrement d. If there is another empty data block, deallocate it. (b) If SB_{s-1} is empty, decrement s.

Space Bound. Notice that $s = \lceil \log_2(n+1) \rceil$. *Proof.* The number of elements in the first s superblocks is $\sum_{i=0}^{s-1} 2^i = 2^s - 1$ (geometric series). If we let this equal n, then $s = \log_2(1+n)$. For slightly smaller than n, the same number of superblocks are required and hence the ceiling. □

The number of data blocks = $O(\sqrt{n})$. *Proof.* As there is ≤ 1 empty data blocks, it suffices to prove that $d = O(\sqrt{n})$ (Equation 8).

$$d \leq \sum_{i=0}^{s-1} 2^{\lfloor i/2 \rfloor} \leq \sum_{i=0}^{s-1} 2^{i/2} = \frac{2^{s/2}-1}{\sqrt{2}-1} = O(\sqrt{n}) \tag{8}$$

The above is true because $2^s = O(n)$. We can also recognize that the last data block has size $O(\sqrt{n})$. *Proof.* Size of DB_{d-1} is $2^{\lceil (s-1)/2 \rceil} = O(\sqrt{n})$. If there is an empty block, it has either the same size or twice the size.

From these analyses, the total overhead is $O(\sqrt{n})$ □.

Update Time. Given the preceding pseudocode, each piece of pseudocode had a constant number of operations. We will make the claim that update time for this data structure is $O(1)$ amortized time. If the *Allocate* or *Deallocate* operations are called when $n = n_0$, the next call to them will occur after $\Omega(\sqrt{n_0})$ operations. This guarantees $O(1)$ amortized time even if either operation requires time linear in the block allocated or deallocated. *Proof.* Immediately after allocating or deallocating a data block, there is exactly one empty data block We only deallocate when there is an extra empty data block. So, we must call *Delete* as many times as the size of DB_{d-1}, which is $\Omega(\sqrt{n_0})$. As we only allocate a data block when this empty block is full, this requires $\Omega(\sqrt{n_0})$ insertions □.

Data Structures in a Bounded Universe

Bounded Universes

PROBLEM. We want to maintain an ordered set of n elements that will support the following operations.

- Membership (Search),
- Successors & Predecessors to Elements (Strict – An element cannot be its own successor),
- Insert & Delete

Bounded Universes continued on next page...

Imagine the following sequence: $2,7,11,20,25$. The strict successor of element 20 is 25, and its predecessor is 11. To maintain such a set, there are a large number of possible solutions for data structures. The *most well-known solution* is to use **balanced search trees** (e.g., red/black, AVL). The search and other operations of these trees can always be done in $O(\log_2 n)$ time and require $O(n)$ space.

BINARY DECISION TREE MODEL. The *lower bound* for membership (search) in an array under **the binary decision tree** model is $\Omega(\log_2 n)$. In this model, we represent the possible execution paths of the algorithm as a binary tree. Each node is labelled by a comparison $x < y$ and the leaf contains the result of this computation. Program execution corresponds to a root-leaf-path. Decision trees correspond to algorithms that only use comparison to gain knowledge about input. Recognize that one decision tree can be constructed for any possible algorithm that only uses comparison.

Proof. Any binary decision tree must have $n+1$ leaves, one for each element in the array, and one for the answer "not found". As a binary tree on $n+1$ leaves must have height $\Omega(\log_2 n)$ (when the tree is full), the lower bound applies $\square$.

Dynamic Perfect Hashing. A hashing approach called **dynamic perfect hashing** supports membership in $O(1)$ worst case time, insert/delete in $O(1)$ expected time, and has a linear $O(n)$ space cost. Is this a counterexample? The previous was proven under a particular model – only comparisons are allowed. With hashing, certain arithmetic operations need to be done. This is a different machine model where RAM is required with arithmetic and input values being used as memory addresses.

To perform better, we require a new model that allows more than just comparisons.

WORD RANDOM ACCESS MACHINE (RAM) MODEL. Try to model a realistic computer where there is not infinite precision. If everything is finite precision, can treat all values as integers. A word size in the computer is a number of bits $w \geq \log_2 n$ where n is the number of elements in the input. The "C-style" operations are those that are $O(1)$. The universe size u refers to the notion that elements have values in $\{0,1,...,u-1\}$. Elements must also fit in a machine word — $w \geq \log_2 u$.

Preliminary approaches include the following.

Direct Addressing. Consider a bit vector where $u = 16, n = 4, S = \{1,9,10,15\}$ where S is the set being maintained. The bit vector that encodes the set is therefore:

$$0\,1\,0\,0\,0\,0\,0\,0\,0\,1\,1\,0\,0\,0\,0\,1$$

For membership/insert/delete, the complexity is $O(1)$. For successor/predecessor operations, this is $O(n)$ (e.g., to find there is no successor or predecessor). In terms of running time, this is not so good for the latter operation(s). How can this solution be improved?

Widgets. Carve the structure into widgets of size $\sqrt{u}$ where widget w_i denotes the elements $i\sqrt{u}+j$, $j = 0,1,...,\sqrt{u}-1$. Each widget is a corresponding bit vector for those elements. Recognize that there can also be a summary widget where $\text{Summary}[i] = 1 \iff w_i$ contains a 1. Recognize that the summary widget has $\sqrt{u}$ bits, one for each widget, if we assume $u = 2^{2k}$.

We can define the following functions, both being able to be done in $O(1)$, $\text{high}(x) = \lfloor x/\sqrt{u} \rfloor$ and $\text{low}(x) = x \bmod \sqrt{u}$. From this, $x = \text{high}(x)\sqrt{u} + \text{low}(x)$. When x is represented as a $\log_2 u$-bit binary number, $\text{high}(x) =$ highest $(\log_2 u)/2$ bits. and $\text{low}(x)$ is the lowest. Recognize that $2^{(\log_2 u)/2} = \sqrt{u}$.

From this, we can immediately locate a value in terms of which widget and which element in that widget it is. For example, $x = 9$ is represented as 1001 where $\text{high}(x) = 2$ and $\text{low}(x) = 1$. Membership computes a location at $w_{\text{high}(x)}[\text{low}(x)]$.

To find the successor, search to the right with x's widget. If a 1 is found, the result is found (its position gives us the result). This is $O(\sqrt{u})$. If not, we go to the summary widget. Search to the right within the summary widget —

$O(\sqrt{u})$. If a 1 is found, that is the widget where the successor is contained – we then find the left-most one in that widget. This is $O(\sqrt{u})$. The total running time of this is $O(\sqrt{u})$.

To support insert/delete, we update the corresponding position in the widget that it should or does belong in, and update the summary widget if necessary. This, overall, requires $O(1)$ time for insertion and $O(\sqrt{u})$ time for deletion (need to know if should set the summary position to zero).

This solution holds an improvement over the previous. Notice, though, that there remains some $O(\sqrt{u})$ computational complexity for operations. This is still not better than binary search trees where logarithmic time still remains an improvement.

A RECURSIVE STRUCTURE. Imagine a solution can be proposed that has a running time for some operation $T(u) = T(\sqrt{u}) + O(1)$. The Master Theorem did not typically possess square roots. Let $m = \log_2 u$ so that $u = 2^m$. What this produces is $T(2^m) = T(2^{m/2}) + O(1)$. Rename $S(m) = T(2^m)$ so that $S(m) = S(m/2) + O(1) = O(\log_2 m)$ (the same recurrence for the binary search tree) by the Master Theorem. Change back from $S(m)$ to $T(u)$ (Equation 9).

$$T(u) = T(2^m) = S(m) = O(\log_2 m) = O(\log_2 \log_2 u) \tag{9}$$

Realize that this function grows incredibly slowly. We want to design something that achieves this sort of performance. In the *Widgets* solution, we divided the problem into portions of $\sqrt{u}$. In that structure, there was not very much recursion. If we *could*, we could express the running time with this manner of formula. Recall the binary search tree where a node can point to two different structures built using the same strategy as the one just built (e.g., the node is chosen as a median value).

Each substructure of our proposed structure is a recursive call. In the original solution, we partitioned the universe once. In this case, we do so recursively. Given a widget w, divide it into $\sqrt{|w|}$ sub-widgets. Therefore, a widget has a universe of such a size. A widget will also have a summary widget corresponding to it. This continues to be built recursively for both widgets and its summary widget. The base case is that a widget contains two bits (is a universe of size 2).

Implementation. An internal node corresponds to a widget w, has a corresponding pointer to a summary widget for that widget, and has a corresponding number of pointers to its sub-widgets. If laid out in memory, a multifurcating tree will be formed.

How do we support operations for such a structure?

Membership. We would have the following pseudocode. Can label the root widget w.

```
Member(w,x)
  if |w| = 2
    return w[x]
  else
    return Member(sub[w][high(x)],low(x)) // x / sqrt(w)
```

Recognize that this has $O(\log_2 \log_2 u)$ running time since it follows the preceding recursive formula.

Insertion. We would have the following pseudocode.

```
Insert(w,x)
  if |w| = 2
    w[x] = 1
  else
    Insert(sub[w][high(x)],low(x))
    Insert(summary[w],high(x))
```

This is different from the membership query in the sense that there are two recursive calls being made for each node involved. The same function cannot be used to express the running time. We must instead use $T(u) = 2T(\sqrt{u}) + \Theta(1)$. This recursive formula can be solved using the Masters Theorem again. Let $m = \log_2 u$. Therefore, $T(2^m) = 2T(2^{m/2}) + \Theta(1)$. Giving this another functional name, $S(m) = T(2^m)$. In other words, $S(m) = 2S(m/2) + \Theta(1)$. By the Masters Theorem, we have $S(m) = \Theta(m)$ which means $T(u) = T(2^m) = S(m) = \Theta(m) = \Theta(\log_2 u)$. This is not as good as the previous running time.

Successor. To find the successor for a value, we can use the following pseudocode. This function returns the smallest value that is greater than x.

```
Succesor(w,x)
  j <-- Successor(sub[w][high(x)],low(x))
  if j < infinity
    return high(x) * sqrt(|w|) + j // there are high(x) subwidgets before
  else // Need to look in another subwidget
    i <-- Successor(summary[x],high(x))
    j <-- Successor(sub[w][i],-infinity) // look for the minimum value
    return i * sqrt(|w|) + j
```

Recognize that there are three recursive calls here. If we follow the same logic as before, $T(u) = 3T(\sqrt{u}) + \Theta(1) = \Theta((\log_2 u)^{\log_2 3})$. To make this faster, realize that there is a special-case Successor call which looks for the minimum value. Imagine storing this minimum value into a structure associated with w. If, at the same time, we also store the maximum, we can also benefit. Notice that the first recursive call may fail to find a successor. A possible solution with this strategy in mind is the structure known as the *van Emde Boas Tree*.

van Emde Boas Trees

van Emde Boas Trees have a number of improvements over the **Widget** solution proposed in the previous section.

Improvement 1. For each widget w, store `min[w]` and `max[w]`. Both of these are set to -1 if w is empty. Membership does not change. It is still $O(\log_2 \log_2 u)$. Realize what now happens for determination of a successor.

```
Successor(w,x)
  if x < min[w]
    return min[w]
  else if x >= max[w]
    return infinity
  w' <-- sub[w][high(x)] // Subwidget whose universe contains x
  if low(x) < max[w']
    return high(x) * sqrt(|w|) + Successor(w',low(x))
  else
    i <-- Successor(summary[w],high(x))
    return i * sqrt(|w|) + min[sub[w][i]]
```

Since this now only requires a single recursive call in either branch, this requires only $O(\log_2 \log_2 u)$ time as well. To use these new augmented fields, we must change the algorithm for insertion. This is still $O(\log_2 u)$.

```
Insert(w,x)
  ...
  // Recursive case.
  if sub[w][high(x)] is empty
    Insert(summary[w],high(x))
```

van Emde Boas Trees continued on next page...

```
  Insert(sub[w][high(x)],low(x))
  if x < min[w]
    min[w] <-- x
  if x > max[w]
    max[w] <-- x
```

Improvement 2. Do not store any of these values in any subwidget of w. Maintain them in the root node of subwidgets in their parent widget. With this change, membership now can check if a value is between extremes (it is no longer present in subwidgets). This requires trivial changes and does not affect the running time. For the successor function, this is also a trivial change – merely a few special cases need to be taken care of and the running time does not change. How does this help with insertion?

```
Insert(w,x)
  if min[w] = max[w] = -1
    min[w] <-- max[w] <-- x
  else if min[w] = max[w]
    (min[w],max[w]) <-- (min(x,min[w]),max(x,max[w]))
  else
    if x < min[w]
      swap(x,min[w])    // Not stored in subwidget.
    else if x > max[w]
      swap(x,max[w])    // Not stored in subwidget.
    w' <-- sub[w][high(x)]
    Insert(w',low(x))    // (1)
    if max[w'] = max[w'] // (2)
      Insert(summary[x],high(x))
```

If (2) is true, then (1) takes $O(1)$ time. Therefore, we can now definitively express this as $T(u) = T(\sqrt{u}) + O(1) = O(\log_2 \log_2 u)$.

DELETION. The supporting of deletion requires a number of special cases.

```
Delete(w,x)
  // Special Case 1
  if min[w] = -1
    return // Do not need to do anything

  // Special Case 2
  if min[w] = max[w]
    if min[w] = x // Only element in widget is equal to x.
      min[w] <-- max[w] <-- -1 // Not stored in subwidgets; done.
    return

  // Special Case 3
  if min[summary[w]] = -1 // Two elements only (min and max).
    if min[w] = x
      min[w] <-- max[w]
    else if max[w] = x
      max[w] <-- min[w]
    return

  // ** More than two elements; some are stored in subwidgets.
```

```
// Special Case 4
if x = min[w]
  j <-- min[summary[w]] // Cannot be an empty set.
  min[w] <-- min[sub[w][j]] + j * sqrt(|w|)
  x <-- min[w]
else if x = max[w]
  j <-- max[summary[w]]
  max[w] <-- max[sub[w][j]] + j * sqrt(|w|)
  x <-- max[w]

w' <-- sub[w][high(x)]
Delete(w',low(x))          // (1)
if min[w'] = max[w'] = -1  // (2)
  Delete(summary[w],high(x))
```

Recognize that if `(2)` is true, `(1)` takes $O(1)$ time since it only had a single element. Therefore, the same recursive formula applies: $T(u) = T(\sqrt{u}) + O(1)$. Deletion, then, only requires $O(\log_2 \log_2 u)$ time.

SPACE. $S(u) = (\sqrt{u}+1)S(\sqrt{u}) + \Theta(\sqrt{u})$ where there are $(\sqrt{u}+1)$ number of subwidgets and a summary widget and $\Theta(\sqrt{u})$ pointers and min/max fields. This is $\Theta(u)$. What if $\sqrt{u}$ is not an integer? Each widget will have a size $\lceil\sqrt{u}\rceil$. Similarly, $T(u) = T(\lceil\sqrt{u}\rceil) + O(1) \leq T(u^{2/3}) + O(1)$ which is still $O(\log_2 \log_2 u)$.

Can we make this occupy less space? There are two other related data structures we can discuss.

x-fast trie

Another solution to this problem is the data structure known as **x-fast trie**. This solution will use less space, but sacrifices the efficiency of some of the operations.

PRELIMINARY APPROACH. Suppose that $u = 16, n = 4, S = \{1, 9, 10, 15\}$. The set of possible values range from 0 to 15. A bit vector can be formed that has a 1 at the position of a value if it exists in the set.

0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 1

We can then use the OR logical operator to condense this bit array by half by considering every successive pair of values.

1 0 0 0 1 1 0 1

And we can then do this again to halve this array once more.

1 0 1 1

This will happen until only a single node exists. We therefore construct a binary tree across our universe. The elements of S are stored as a doubly-linked list where pointers are kept between them (this allows us to find the successor and predecessor). We store three sets of structures: the bit vector, the tree, and the doubly-linked list. Pointers are maintained between positions in the bit vector and linked list.

Performance Analysis. For membership, we have $O(1)$ time by accessing the bit vector. Finding the successor can be done using the following pseudocode which takes $O(\log_2 u)$ time if the item is not in the list.

```
Successor(x)
  if x in S
    use the linked list // O(1)
```

x-fast trie continued on next page...

```
else
  walk up the tree until we find a 1 node // O(log u)
  // this node must have a 1 child
  if this is the right child
    find its min value and this is the successor
  else if this is the left child
    find its max value and this is the predecessor
    use the linked list to find the successor
```

For insertion and deletion, we have $O(\log_2 u)$ time. The space cost is ultimately $O(u)$. This does not seem to be a much better solution. However, this can be improved tremendously.

ACTUAL SOLUTION. We do not keep this bit vector, but we do keep the doubly-linked list. Also, we keep the 1-nodes only. The 1-node leaf elements are chained as a doubly linked list and connected to a tree which is no longer strictly bifurcating but resembles more of a sparse binary tree.

Operation Support. Will this structure still support all of the operations? Imagine starting from a position that is not present in the structure. This may become a problem. A number of challenges are present here.

To find the predecessor of x, how do we locate the lowest 1-node on the leaf-to-root path? For any 1-leaf, we can just refer to its own structure. For a 0-leaf, consider the following solution.

- Each ancestor of leaf x corresponds to a prefix of the binary representation of x. For example, $x = 13$ is expressed in binary as $(1101)_2$. which has ancestors 1101, 110, 11, 1, and an empty prefix. The root corresponds to the empty prefix. The presence of a 0 corresponds to a left branch and 1 corresponds to a right branch.
- Using such a prefix as a key for any one node, we store in a dynamic perfect hashing table. There will be dynamic perfect hashing on m keys which will possess $O(m)$ space, support membership in $O(1)$ time, and therefore require $O(1)$ amortized expected cost.
- The lowest 1-node that is an ancestor of x corresponds to the longest prefix of x that is in the hash table. Can scan ancestors by doing a binary search on $O(\log_2 u)$ bits – a binary search in the prefixes of x would take $O(\log_2 \log_2 u)$ time.
  - If this node has a 1-child as its only right child, precompute and store a *descendent pointer* to the smallest leaf of its right subtree.
  - If this node has a 1-child as its only left child, store a *descendent pointer* to the largest leaf of its left subtree.

Finding the successor should then only take $O(\log_2 \log_2 u)$ time. Membership would only be $O(1)$ because of the hash table. Insertion and deletion, however, is $O(\log_2 u)$ amortized expected cost. Space for this structure requires $O(n)$ for the list, $O(n \log_2 u)$ ($\log_2 u$ prefixes for each item in n) for the hash tablex, and $O(n \log_2 u)$ space for the tree – recognize that there are ≤ 3 pointers per node.

How do we make the space $O(n)$? We improve on insertion and deletion.

y-fast trie

A very useful principle is the use of **indirection** for improving on space costs in data structures. Indirection is a general approach for decreasing space cost. In the **y-fast trie**, we clusters elements of S into consecutive groups of size $\frac{1}{4}\log_2 u \sim 2\log_2 u$. Each of these groups have elements stored in a balanced binary search tree (BST). A set of representative elements, one per group, is maintained and these are stored in an **x-fast trie** structure.

y-fast trie continued on next page...

A group's *representative element* does not have to be in S. However, it must be between the maximum in the preceding and the minimum of the succeeding group in value, exclusively.

The space this structure therefore entails is $O(n)$ as it requires $O(\frac{n}{\log_2 u}\log_2 u) = O(n)$ space for the x-fast trie structure (recall the structure requires $O(n\log_2 u)$ for n elements but this has $\frac{n}{\log_2 u}$ elements as leaves), $O(n)$ for each of the respective BSTs, and therefore $O(n)$ total.

OPERATIONS. Determining if an item is in the structure can be done with the following pseudocode which takes $O(\log_2\log_2 u)$ time.

```
Membership(x)
  if x is stored in x-fast trie
    check the corresponding BST for x                 // O(log log u)
  else
    find x's predecessor and successor in x-fast trie // O(log log u)
    check two balanced BSTs to check for x in S       // O(log log u)
```

Finding the `Successor` of an item is similar and takes $O(\log_2\log_2 u)$ time as well. To perform `Insert` or `Delete` of x, we do the following:

- Use the *x-fast trie* to locate the group that x is in. $O(\log_2\log_2 u)$
- Insert x into the BST of that group (or delete). $O(\log_2\log_2 u)$
- Ensure the group size is within the required bounds $\frac{1}{4}\log_2 u \sim 2\log_2 u$.
- If a group is of size $> 2\log_2 u$, we *split* that group. Else if a group if of size $< 1/4\log_2 u$, we merge it with an adjacent group which may cause another split. $O(\log_2 u)$ amortized expected. This should only take place every $\Theta(\log_2 u)$ insertions/deletions and therefore the sequence of insertions or deletions is $O(1)$ amortized expected $= \frac{O(\log_2 u)}{\Theta(\log_2 u)}$.

The total cost of insertion or deletion is $O(\log_2\log_2 u)$ amortized expected.

Self-Adjusting Data Structures

Self-Adjusting Lists

Problem. The problem here is, assuming a linked list, we want a linear search of n elements. n worst case, there are n comparisons that are done. If all elements have an equal possibility of search (being the goal), the expected time is $\frac{n+1}{2}$ (in a successful case) or n (in an unsuccessful case).

$$p = \sum_{i=1}^{n} \frac{1}{n} i = \frac{n+1}{2} \tag{10}$$

Real Questions for Real Applications. Consider the following.

1. What if the probabilities of access differ (where p_i is the probability of accessing element i)?
2. What if these (independent) probabilities are not given?
3. How are solutions compared with the best we could do?

Issues. Expected behavior (given independent probabilities), if probabilities change, amortized behavior, and considering structures that will adapt to input sequences.

Self-Adjusting Lists continued on next page...

Model is a key here.

Self-Adjusting Data Structures adapt to changing probabilities. If all probabilities are the same and independent, it does not matter how you change your structure. If both probabilities and this list are fixed, we want to minimize:

$$\sum_{i=1}^{n} i p_i \tag{11}$$

In practice, MTF is used most frequently because it works well in operating systems and other applications. Let us compare this strategy with the starting optimal solution (S_{opt}). The *model* that is used here will start with an empty list.

```
start with an empty list
scan for element requested
if not present
  insert (and charge) as if found at the end of list (then apply heuristics)
```

The claim that will be made here is that the cost of MTF is $\leq 2S_{\text{opt}}$ for any sequence of requests.

Proof. Consider an arbitrary list, but focus only on searches for b and c, and "unsuccessful" comparisons where we compare query value b or c against c or b respectively. Assume request sequence has k bs and m cs. Without loss of generality, $k \leq m$. The order for S_{opt} is cb and there will be k unsuccessful comparisons.

What order of requests will maximize the number of unsuccessful operations under MTF? Clearly $c^{m-k}(bc)^k$. So there are $2k$ "unsuccessful" comparisons. Another way of seeing this: under the MTF rule, an unsuccessful comparison will be made whenever the request sequence changes b to c or from c to b. Therefore, the latter order of requests is not a unique solution. Since each change involves a b, and each request for b can be involved in at most 2 changes (before and after), the total number of such changes $\leq 2k$.

Note that this holds for any pair of elements, sum over all total number of unsucessful comparisons of MTF ≤ 2 number of unsuccesful comparison in S_{opt}. We also observe that the total number of successful comparisons of MTF is equal to the total number of successful comparisons for S_{opt}. If we add these two together, the cost of MTF is $\leq 2S_{\text{opt}}$. □

This bound is tight. Given $a_1, a_2, a_3, ..., a_n$, repeatedly asking for the last value in the list (so all requested equally often). The cost of MTF is n comparisons per search. In S_{opt} (where nothing is done), the cost is $\frac{n+1}{2}$ per search.

For some sequences, MTF performs a lot better than S_{opt}. Given $a_1^n a_2^n a_3^n ... a_n^n$, S_{opt} has $\frac{n+1}{2}$ comparisons per search. For MTF, the upper bound of comparisons is

$$\frac{(1+(n-1))+(2+(n-1))+(3+(n-1))+...+(n+(n-1))}{n^2} = \frac{(n(n+1))/2+n(n+1)}{n^2} = 3/2 - O(1/n) \tag{12}$$

Dynamic Optimality

In algorithms, **online** means that an algorithm must respond as requests come (the full sequence is not known beforehand). **Offline** means that the opportunity is had to see the entire schedule (request sequence) beforehand and this can be made use of to determine how to move values. The **competitive ratio** of algorithms is defined as is found in Equation 13.

Dynamic Optimality continued on next page...

$$\text{competitive ratio} = \text{worst case of } \frac{\text{online time of algorithm}}{\text{optimal offline time}} \tag{13}$$

A method is **competitive** if this ratio is constant, which means the algorithm, when online, is within a constant factor of the optimal offline algorithm.

We will show that the MTF algorithm is competitive.

MODEL. Developing a model, we can search for or change an element in position i by scanning to location i at cost i. Let a **free exchange** be the number of exchanges or swaps required to move element i closer to the front of the list, keeping all other elements in the same order. This effectively has no cost. A **paid exchange** refers to any other type of exchange that is not free (of cost). This would cost a single unit of time. Given this, consider the following operations (before the cost of self-adjusting):

- *Access*: costs i if element is in position i.
- *Delete*: costs i if element is in position i.
- *Insert*: costs $n+1$ if n elements are already present.

Furthermore, the request sequence can be said to possess a total of M queries and there is a maximum of n (different) elements. We start with the empty list. The cost model for a search that ends by finding an element in position i is i + number of paid exchanges. Define $C_A(S)$ as the total cost of all the operations for an algorithm A not counting paid exchanges and $X_A(S)$ as the number of paid exchanges. Also define $F_A(S)$ as the number of free exchanges.

We thus can claim that $X_{\text{MTF}}(S) = X_{\text{TR}}(S) = X_{\text{FC}}(S) = 0$. We can also observe that for any algorithm A, $F_A(S) \leq C_A(S) - M$. After accessing or inserting or deleting the ith element, there are $\leq i-1$ free exchanges (we are subtracting 1 M times in this observation).

Therefore, $C_{\text{MTF}}(S) \leq 2C_A(S) + X_A(S) - F_A(S) - M$ for any algorithm A that starts with an empty list.

Proof. By *Amortized Analysis* with a *Potential Method*. Suppose we run algorithm A and MTF in parallel on some request sequence S. The potential function Φ is defined as the number of inversions between MTF's list and A's list. An inversion is defined as when an element i occurs before j in one list, but i occurs after j in the other.

Notice first that the initial potential is 0 and that $\Phi \geq 0$. We now bound the amortized cost of access.

Consider access by A to position i, and assume we go to position k in MTF's list (for the same item). No assumption is made about their position relative to each other. Define x_i as the number of items preceding this item in MTF's list but not in A's list. Therefore, the number of items preceding the item in both lists is $k-1-x_i$. After access, the item is moved to the front in MTF's list; this creates $k-1-x_i$ inversions and destroys x_i others. Following this analysis, the amortized time is $k+(k-1-x_i)-x_i = 2(k-x_i)-1$. Since $k-1-x_i \leq i-1$, $k-x_i \leq i$ holds. Therefore, the amortized time of access in MTF's list $\leq 2i-1 = 2C_A - 1$. The same argument can be made for insertion and deletion.

However, a change in potential can *also* occur by A's list being self-adjusted. An exchange by A has 0 cost to MTF (it does not affect that list). The change in potential is all that is incurred and must be considered. The amortized time of an exchange by A is just the increase in the number of inversions caused by exchange, i.e., some number of free (-1) or paid exchanges (1).

The total amortized cost for the request sequence is therefore $a_i = 2C_A(S) - M + X_A(S) - F_A(S)$. □

MTF is competitive. Define $T_A(S) = C_A(S) + F_A(S) + X_A(S)$. We know that $F_{\text{MTF}}(S) \leq C_A(S) - M$ and that $X_{\text{MTF}}(S) = 0$. Therefore, $T_{\text{MTF}}(S) = C_{\text{MTF}}(S) + F_{\text{MTF}}(S) + X_{\text{MTF}}(S) \leq 2C_{\text{MTF}}(S) - M \leq 4C_A(S) + 2X_A(S) - 2F_A(S) - 3M \leq 4T_A(S)$.

Self-Adjusting Binary Trees

PROBLEM. Given a sequence of access queries, what is the best way of organizing the binary search tree (BST)?

Model. Searches must start at the root, and the cost is equal to the number of nodes inspected.

Worst Case. In the worst-case, there is an $\Theta(\log_2 n)$ upper and lower bound.

STATIC OPTIMAL BINARY SEARCH TREES. Over the distribution of the queries, we have elements $A_1 < A_2 < ... < A_n$. We are given p_i, the probability of access of element A_i such that $i = 1, 2, ..., n$. Define q_i = the probability of access for the values between and excluding A_i and A_{i+1} where $i = 0, 1, ..., n$. We define $A_0 = -\infty$ and $A_{n+1} = \infty$ where $p_0 = p_{n+1} = 0$.

Also define $T[i, j]$ as the root of the optimal tree on ranges q_{i-1} to q_j and $C[i, j]$ as the cost of the tree rooted at $T[i, j]$ defined as a sum $\sum$ of the probabilities of looking for one of the values (or gaps) in the range.

Example. Picture the following tree where A_2 has children A_1 and A_5. A_5 has child A_4 which has child A_3. $p_1 = 0.15$, $p_2 = 0.1$, $p_3 = 0.05$, $p_4 = 0.1$ and $p_5 = 0.2$. $q_0 = 0.05$, $q_1 = 0.1$, $q_2 = 0.05$, $q_3 = 0.05$, $q_4 = 0.05$ and $q_5 = 0.1$. Therefore, $C[1,5] = 0.15(2) + 0.1(1) + 0.05(4) + 0.1(3) + 0.2(2) + 0.05(2) + 0.1(2) + 0.05(4) + 0.05(3) + 0.1(2) = 2.35$ is an expected cost where costs of access follow as coefficients.

Dynamic Programming. We can make an observation that some root r has a subtree L and R as left and right children. Define W[tree] as the probability of being in the tree = the probability of visiting the root. Therefore, C[tree rooted at r] $= 1 \times W$[tree] $+ C[L] + C[R]$. It will also be useful to define $W[i, j]$, the probability of accessing any node $p_i, ..., p_j$ or gap $q_{i-1}, ..., q_j$ in $T[i, j]$.

$$W[i, j] = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k \tag{14}$$

Notice that $W[i, j+1] = W[i, j] + p_{j+1} + q_{j+1}$. This will be $O(n^2)$ to compute all of them.

$$C[i, j] = \begin{cases} \min_{r=i}^{j}(W[i, j] + C[i, r-1] + C[r+1, j]), & \text{if } i \leq j \\ 0, & \text{if } j = i - 1 \end{cases} \tag{15}$$

Consider the following pseudocode.

```
OPTIMAL-BST(p,q,n)
  for i <-- 1 to n+1
    C[i,i-1] <-- 0
    W[i,i-1] <-- q_{i-1}
  for l <-- 1 to n
    for i <-- 1 to n - l + 1
      j <-- i + l - 1
      C[i,j] <-- infinity
      W[i,j] <-- W[i,j-1] + p_j + q_j
      for r <-- i to j
        t <-- C[i,r-1] + C[r+1,j] + W[i,j]
        if t < C[i,j]
          C[i,j] <-- t
          T[i,j] <-- r
  return C[1,n+1]
```

The running time of finding the optimal BST is $O(n^3)$. However, $O(n^2)$ is possible.

SPLAY TREES. After accessing an item, we *rotate* it to the top (the root) of the tree. Formally, on accessing a node, we move (splay) it to the root by a sequence of local moves.

Splaying. Let x be the node being moved towards the root, y is the parent of x, and z is the parent of y. The **zig step** is as follows. If y is the root, rotate x so that it becomes the root (a single rotation). The **zig-zig step** is where y is not the root, and x and y are both left or right children. Here, we first rotate y and then rotate x. The **zig-zag step** is where y is not the root, and x is a left child and y a right child, or vice versa. We double rotate x.

Splaying a node x of depth d takes $O(d)$ time. If we *only* perform single rotations (with no splaying where the parent can be rotated), we will create a single path subtree (where its length is unchanged) and accessing will not remedy a bad tree shape. Splaying will roughly halve the length of the access path.

Analysis. Let $w(i)$ be the weight of item i (a positive value). This is not given as data. Let $S(x)$ be the size of a node x in the tree (i.e., the sum of the individual weights of all items in the subtree that is rooted at x). Let $r(x)$ be the rank of node x where $r(x) = \log_2 S(x)$.

We define the potential Φ of the tree to be the sum of the ranks of all of its nodes. Notice that the initial potential function is not necessarily zero and that Φ_i may be negative because of the log function. The amortized cost could therefore not serve as an upper bound for the actual cost. Realize, though, that $\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m$ still holds. The cost is equal to the number of rotations that are performed.

Access Lemma. The amortized time to splay a tree with root t is at most $3(r(t) - r(x)) + 1 = O(\log_2(S(t)/S(x)))$.

Proof. If there are no rotations, the bound is immediate. Thus, suppose there is at least one rotation. Consider the first *splaying step*. Let $r_0(i), r_1(i)$ and $s_0(i), s_1(i)$ represent the ranks and sizes of node i before and after the first step respectively.

CASE 1. Zig Step. The actual cost of this step is 1 because only a single rotation is performed. The amortized cost would be equal to $1 + \Delta\Phi = 1 + r_1(x) + r_1(y) - r_0(x) - r_0(y)$. Only x and y can change ranks in this step. This is $\leq 1 + r_1(x) - r_0(y)$ because $r_0(y) \geq r_1(y)$. Similarly, this is $\leq 1 + 3(r_1(x) - r_0(y))$ because $r_1(x) \geq r_0(x)$.

CASE 2. Zig-Zig Step. The actual cost of this step is 2 because there are two rotations. The amortized cost would be equal to $2+\Delta\Phi = 2+r_1(x)+r_1(y)+r_1(z)-r_0(x)-r_0(y)-r_0(z)$. Notice that $r_1(x) = r_0(z)$. This means that the amortized cost is $\leq 2 + r_1(y) + r_1(z) - r_0(x) - r_0(y)$. This is further upper bounded where the amortized cost is $\leq 2 + r_1(x) + r_1(z) - 2r_0(x)$ because $r_1(x) \geq r_1(y)$ and $r_0(x) \leq r_0(y)$. We now show that $2 + r_1(x) + r_1(z) - 2r_0(x) \leq 3(r_1(x) - r_0(x))$. This is equivalent to $r_1(z) + r_0(x) - 2r_1(x) \leq -2$. The left hand side (LHS) $r_1(z) + r_0(x) - 2r_1(x) = (r_1(z) - r_1(x)) + (r_0(x) - r_1(x)) = \log_2(S_1(z)/S_1(x)) + \log_2(S_0(x)/S_1(x)) = \log_2((S_1(z)/S_1(x))(S_0(x)/S_1(x)))$. Since $S_1(x) > S_1(z) + S_0(x)$, $(S_1(z)/S_1(x)) + (S_0(x)/S_1(x)) \leq 1$, their product is $\leq 1/4$, since the geometric mean of two positive values $\sqrt{ab} \leq \frac{a+b}{2}$ where $a, b > 0$. Therefore, $r_1(z) + r_0(x) - 2r_1(x) \leq \log_2(1/4) = -2$.

CASE 3. Zig-Zag Step. The actual cost of this step is 2. The amortized cost is equal to $2 + r_1(x) + r_1(y) + r_1(z) - r_0(x) - r_0(y) - r_0(z)$. A similar observation can be made as in Case 2 where $r_1(x) = r_0(z)$ and therefore the amortized cost is $= 2 + r_1(y) + r_1(z) - r_0(x) - r_0(y)$. Because $r_0(x) \leq r_0(y)$, the cost is $\leq 2 + r_1(y) + r_1(z) - 2r_0(x)$. We now show that $2 + r_1(y) + r_1(z) - 2r_0(x) \leq 3(r_1(x) - r_0(x))$. This is equivalent to $r_1(y) + r_1(z) + r_0(x) - 3r_1(x) \leq -2$. As $r_1(y)+r_1(z)+r_0(x)-3r_1(x) = (r_0(x)-r_1(x))+\log_2((S_1(y)/S_1(x))(S_1(z)/S_1(x)))$ and $(r_0(x)-r_1(x)) \leq 0$. Because $S_1(x) \geq S_1(y) + S_1(z)$, this expression is $\leq \log_2(1/4) = -2$.

Now, suppose a splay operation requires $k > 1$ steps. Only the kth step could be a zig step. The amortized cost of the whole of the whole splay operation is $\leq 3r_k(x) - 3r_{k-1}(x) + 1 + \sum_{j=1}^{k-1}(3r_j(x) - 3r_{j-1}(x)) = 3r_k(x) + 1 - 3r_0(x)$.

□

NOTE To view solutions online, use user name `ads` and password `movetofront`.

Balance Theorem. Given a sequence of m accesses into a n-node splay tree T, the total access time is $O((m+n)\log_2(n)+m)$.

Proof. Assign a weight of $\frac{1}{n}$ to each item. Then, $\frac{1}{n} \leq S(x) \leq 1$ where the lower bound is associated with a single node and the upper bound is a sum of all nodes' weights. Therefore, the amortized cost of *any access* is at most $3(\log_2(1) - \log_2(\frac{1}{n})) + 1 = 3\log_2(n) + 1$. The maximum potential of T is $\sum_{i=1}^{n} \log_2(S(i)) \leq 0$. The minimum potential of T is $\sum_{i=1}^{n} \log_2(S(i)) \geq \sum_{i=1}^{n} \log_2(\frac{1}{n}) = -n\log_2(n)$. Therefore, $\Phi_0 - \Phi_m \leq 0 - (-n\log_2(n)) = n\log_2(n)$. The total cost is $\leq m(3\log_2(n)+1) + n\log_2(n) = O((m+n)\log_2(n)+m)$ □

If we access $m \geq n$ times, the average cost per access is $O(\log_2(n))$.

Static Optimality Theorem. Given a sequence of m accesses into a n-node splay tree, where each item is accessed at least once, the total access time is $O(m + \sum_{i=1}^{n} q(i)\log_2 \frac{m}{q(i)})$ where $q(i)$ is the access frequency of item i.

Proof. Assign a weight of $\frac{q(i)}{m}$ to each item. Therefore, $\frac{q(x)}{m} \leq S(x) \leq 1$. Now we apply the Access Lemma. The amortized cost of each access is $\leq 3(\log_2 1 - \log_2 q(x)/m) + 1 = 3\log_2 m/q(x) + 1$. The maximum potential of T is $\sum_{i=1}^{n} \log_2 S(i) \leq \sum_{i=1}^{n} \log_2(1) = 0$. The minimum potential of T is $\sum_{i=1}^{n} \log_2 S(i) \geq \sum_{i=1}^{n} \log_2 \frac{q(i)}{m}$. Therefore, $\Phi_0 - \Phi_m \leq -\sum_{i=1}^{n} \log_2 \frac{q(i)}{m} = \sum_{i=1}^{n} \log_2 \frac{m}{q(i)}$. The total cost is $\leq 3\sum_{i=1}^{n} q(i)\log_2 \frac{m}{q(i)} + m + \sum_{i=1}^{n} \log_2 \frac{m}{q(i)} = O(m + \sum_{i=1}^{n} q(i)\log_2 \frac{m}{q(i)})$. □

We can actually define $\sum_{i=1}^{n} q(i)\log_2 \frac{m}{q(i)}$ as the entropy of access $\times m$.

Update. **Inserting** an element involves (a) inserting a value as a new leaf and (b) splaying at the new node. **Deletion** is move complicated. We (a) delete the node – creating three disconnected trees, (b) splay the largest item in its *left* subtree, (c) make the *right* subtree a subtree of the new root of the left subtree, (d) replace x by this new root (1 tree), and (e) splay at x's parent.

Entropy

Entropy

Let $S = \{k_1, ..., k_n\}$ be a set of objects. Let $D = \{p_1, ..., p_n\}$ be a probability distribution where p_i is the probability associated with object k_i. We can now define the **entropy** of D as $H(D)$.

$$H(D) = -\sum_{i=1}^{n} p_i \log_2 p_i = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \tag{16}$$

Shannon's Theorem. There is a sender and a receiver. The sender wants to send a sequence $S_1, ..., S_m$ where each S_j is chosen randomly and independently according to D so that $S_j = k_i$ with probability p_i. For any protocol the sender and receiver might use, the expected number of bits required to transmit $S_1, ..., S_m$ using that protocol is at least (greater than or equal to) $mH(D)$.

Example I. Imagine you have a uniform distribution where $p_i = \frac{1}{n}$. Then, $H(D) = \log_2(n)$. This is the worst possible case of entropy. No compression can be achieved.

Entropy continued on next page...

Example II. Imagine you have a geometric distribution where $p_i = \frac{1}{2^i}$ for all $1 \leq i < n$. To have these all sum to 1, $p_n = \frac{1}{2^{n-1}}$. Therefore, $H(D) = \sum_{i=1}^{n-1} \frac{i}{2^i} + \frac{n-1}{2^{n-1}} \leq 2$. Notice that 2 is much less than $\log_2(n)$. In this regard, we can find an encoding scheme to send 2 bits per character (i.e., Huffman Encoding). To acquire this upper bound, consider that $S = \sum_{i=1}^{\infty} \frac{i}{2^i}$ and $\frac{1}{2}S = \sum_{i=1}^{\infty} \frac{i}{2^{i+1}} = \sum_{i=1}^{\infty} \frac{i+1}{2^{i+1}} - \sum_{i=1}^{\infty} \frac{1}{2^{i+1}} = S - \frac{1}{2^1} - \frac{1}{2} = S - 1$.

ENTROPY AND DATA STRUCTURE LOWER BOUNDS. We will now discuss the following theorem.

Theorem 1. For any comparison-based data structure storing $k_1, k_2, ..., k_n$, each associated with some probability $p_1, p_2, ..., p_n$, the expected number of comparisons while searching for S_i is $\Omega(H(D))$, the entropy for the distribution.

Proof. The sender and receiver both store the elements $k_1, k_2, ..., k_n$ in some comparison-based data structure (that is agreed upon beforehand). When the sender wants to transmit S_1, he performs a search for S_1 in the data structure. This results in a sequence of comparisons. The results of comparisons form a sequence of 1s (`True`) and 0s (`False`), which the sender sends to the receiver.

On the receiving end, this sequence is sufficient to find this value (he stores the same data structure). The receiver runs the search algorithm without knowing the value of S_1. This can be done since the receiver is doing exactly the same comparisons and knows the results of these comparisons. This way, the sender can transmit a sequence to the receiver. By applying Shannon's Theorem, the expected number of bits required to transmit $S_1, ..., S_m$ using this approach is at least $mH(D)$.

Notice that the expected number of bits of transmitting this sequence = the expected number of comparisons of handling this request sequence. □

Example I. *Static Optimal Binary Search Trees.* All of the elements in the request sequence are in the tree. p_i corresponds to the probability of access for A_i such that $\sum_{i=1}^{n} p_i = 1$. The entropy $H = \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$. The expected cost c of the static optimal binary search tree $\geq H$ by this theorem.

Example II. *Static Optimal Binary Search Trees.* Not all requested elements are in the tree. We have p_i as before, and q_i corresponds to the probability of access for values between A_i and A_{i+1}. We can claim that $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$. The entropy $H = \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right) + \sum_{i=0}^{n} q_i \lg\left(\frac{1}{q_i}\right)$.

Add a leaf for each gap. To cope with gaps, we add a leaf for each of them in the tree. If A_i is a leaf, we replace it with a structure such that it has children q_{i-1} and q_i. To make it easier to present these results, we assume that accessing these leaves is the number of nodes examined (1 in addition to its parent).

Therefore, $C \geq H$. In fact, we can prove that $H \leq C \leq H + 3$.

Nearly-Optimal Binary Search Trees

We can generalize the case for gaps by making a generalization of their structure as found in **Example II** above. For now, consider only parameters p_i and H for a binary tree. The goal here is to acquire a query time of $O(H+1)$ – the addition of 1 is for the case that entropy is equal to zero. We also want fast construction time (recall that constructing an optimal binary search tree is $O(n^3)$ or $O(n^2)$).

The idea here is known as **probability splitting**, developed at least 30 years ago but still present in recent publications. First, consider keys $k_1, k_2, ..., k_n$. We find a key k_i such that the following two inequalities hold.

$$\sum_{j=1}^{i-1} p_j \leq \frac{1}{2} \sum_{j=1}^{n} p_j \tag{17}$$

$$\sum_{j=i+1}^{n} p_j \leq \frac{1}{2} \sum_{j=1}^{n} p_j \tag{18}$$

We can always find such a key.

The key k_i will become the root of a binary search tree. We then recurse. That is, if k_i is the root, then the left sub-tree will contain all keys from k_1 to k_{i-1} — it can be constructed recursively from $(k_1, ..., k_{i-1})$. Similarly, the right-subtree will contain all keys from $k_{i+1}, ..., k_n$. Use T to represent the resulting tree.

ANALYSIS. We want to show that this will construct a tree that has the query time and fast construction time we requested. First make observation that in T, if the node stored at k_i (possibly different than the root) has depth $\text{depth}_T(k_i)$, then

$$\sum_{k_j \in T(k_i)} p_j \leq \frac{1}{2^{\text{depth}_T(k_i)}} \tag{19}$$

where $T(k_i)$ is the set of all nodes in the subtree rooted at k_i. Recognize that this means the sum of all probabilities to the left and right of the root of T is $\leq \frac{1}{2}$. This is a result of the probability splitting that was done during construction. A proof by induction can be done to prove this claim.

Theorem. T can answer queries in $O(H+1)$ time.

Proof. As $p_i \leq \sum_{k_j \in T(k_i)} p_j \leq \frac{1}{2^{\text{depth}_T(k_i)}}$ by Equation 19, $\text{depth}_T(k_i) \leq \lg\left(\frac{1}{p_i}\right)$. Thus, the expected depth of a key chosen is $\sum_{i=1}^{n} p_i \times \text{depth}_T(k_i) \leq \sum_{i=1}^{n} p_i \lg(1/p_i) = H$. The time for a query is therefore $O(H+1)$. □

Construction. When finding k_i, we can employ the **prefix sum**. By precomputing all prefix sums, which takes $\Theta(n)$ time, we can then investigate subtractions of prefix sums to get the sum of any sub-array in constant time. Because this is done n times for all keys, this is $\Theta(n^2)$. This is no better than finding the optimal tree.

A better algorithm to find key k_i would be to employ a **binary search** (after precomputing prefix sum arrays). This is $\Theta(n \lg n)$. This can still be improved, however.

The best algorithm to find this key is to perform a **doubling (exponential) search**. This is another method for finding an element in a sorted array similar to a binary search. We still employ a binary search to refine it. Ultimate, what is done is:

1. Check $k_1, k_2, k_4, k_8, ...$ until we overshoot, or find the first k_{2^j} with $2^j \geq i$.
2. Binary search on $k_{2^{j-1}}, ..., k_{2^j}$. The run time of this is $\Theta(\lg n)$. In fact, this is $\Theta(\lg i)$. Notice that $2^j \leq 2i$.

While looking for the key, search simultaneously, starting from k_1, and walk forward (two steps above) and starting from k_n and walk backwards. No parallelism is needed here; these can be interleaved together. The key can be found in the forward or backward search and in a time $O(\lg(\min(i, n-i+1)))$. Using a recurrence, the total running time $T(n) = O(\lg(\min(i, n-i+1))) + T(i-1) + T(n-i)$. We can prove that $T(n) = O(n)$ using induction.

Move-To-Front Compression

Elias Gamma Coding is commonly used to code positive integers whose maximum value is not known. It is easy to design a coding scheme that uses $\log u$ bits if u is known because you know the maximum value.

Steps of Encoding some integer i. (1) Write down the binary representation of i with a number of bits equal to $\lfloor \lg i \rfloor + 1$. (2) Prepend it by $\lfloor \lg i \rfloor$ 0s. The number of bits to encode i is equal to $1 + 2\lfloor \lg i \rfloor$.

Example I. *Elias Gamma Coding.* For example, 9 would be 0001001.

Another Coding Scheme. A coding scheme that uses fewer bits recognizes that the leading 0s of Elias Gamma Coding always encodes the Gamma Coding number $(\lfloor \lg i \rfloor + 1)$. This number can be encoded using $1 + 2\lfloor \lg(\lfloor \lg i \rfloor + 1) \rfloor \leq 1 + 2\lfloor \lg(\lg i + 1) \rfloor$ bits using Elias Gamma Coding. For larger integers, this would save space. The total number of bits here is $\lfloor \lg i \rfloor + 1 + 1 + 2\lfloor \lg(\lg i + 1) \rfloor = \lg i + O(\lg \lg i)$.

Example II. For example, 9 has a binary expression that is 1001. Instead of prepending 3 0s in front, we could encode the Gamma Coding (4) again getting 001001001.

Recurse. If we recurse and finally encode a 1, what we get is a scheme is known as **Elias Omega** or **Recursive Elias**.

Move-To-Front (MTF) Compression of encoding uses m integers from $1, 2, ..., n$. Using concepts borrowed from the Shannon Theorem, we can discuss the sender and receiver here. The sender and receiver each maintain identical lists (using MTF) containing integers $1, 2, ..., n$.

Sender. The send the symbol j, the sender looks for j in his list and finds it at position i. The sender then encodes i using $\lg i + O(\lg \lg i)$ bits (using the coding scheme discussed in *Another Coding Scheme* above) and sends that to the receiver. Subsequently, this item (j) is moved to the front of the list.

Receiver. The receiver decodes i and looks at the ith element in his list to find item j. Subsequently, this item (j) is moved to the front of the list.

After both the sender and receiver move element j to the front of their lists, they continue with this scheme.

Jensen's Inequality. Let $f : R \to R$ be a strictly increasing concave function. Then, the sum

$$\sum_{i=1}^{n} f(t_i) \tag{20}$$

subject to the constraint that the sum $\sum_{i=1}^{n} t_i \leq m$ is maximized when $t_1 = t_2 = ... = t_n = \frac{m}{n}$. A concave function is where f' is decreasing.

The MTF compression algorithm compresses the sequence $S_1, S_2, ..., S_m$ into $n \lg n + mH + O(n \lg \lg n + m \lg H)$ bits.

Proof. If the number of distinct symbols between two consecutive occurrences of the integer j is $t - 1$, then the cost of encoding the second j in bits is $\lg t + O(\lg \lg t)$ because it would be at position t. Therefore, the total cost of encoding all occurrences (m_j) of j is $C_j \leq \lg n + O(\lg \lg n) + \sum_{i=1}^{m_j} \lg t_i + O(\lg \lg t_i)$ where $t_i - 1$ is the number of symbols (we are interested in an upper bound so this will still hold) between the $(i-1)$st and ith occurrences of j. Recognize that $\sum_{j=1}^{m_j} t_j \leq m$. Notice also that $\lg n$ is a concave function. Therefore, $C_j \leq \lg n + O(\lg \lg n) + m_j(\lg(\frac{m}{m_j}) + O(\lg \lg(\frac{m}{m_j})))$. Summing them over all j, the total number of bits is

Move-To-Front Compression continued on next page...

$$\sum_{j=1}^{n} C_j \leq n\lg n + O(n\lg\lg n) + mH + \sum_{j=1}^{m} m_j \lg\lg \frac{m}{m_j} = n\lg n + mH + O(n\lg\lg n + m\lg H) \tag{21}$$

We could translate the third term to mH by considering $p_j = \frac{m_j}{m}$. Notice that $n\lg n$ (the first term) is a startup cost for the compression.

Note. For natural language text strings, you are looking at around 30 to 70 percent of the original string ($128\lg 128 \approx$ 14 words). *bzip2* implements the compression algorithm based on the BWT (B-W Transform) and applies MTF compression to this transform (with Huffman Encoding). Typically much faster than more efficient statistical approaches and still among the best.

□

Text Indexing Structures

Text Search Problem

In this problem, define Σ as the alphabet with size σ. Define T as a text string of length n over alphabet Σ. Define P to be a pattern string of length p over alphabet Σ. The problem of text search involves finding occurrences of P in T.

Example. Have a text string T that is "to be or not to be". The pattern string could be "be". Finding "be" in T is an example of text search.

Two different approaches were used for this problem. One could treat this, firstly, as an algorithm problem.

STRING MATCHING ALGORITHMS. No preprocessing is allowed. Merely given T and P as strings of characters — they cannot be placed into any sort of other data structure beforehand. Constructing a data structure would be part of the algorithm.

Obvious Method. Search the string for occurrences character-by-character. There are n different positions where you must compare p characters. Therefore, this is $O(pn)$ time.

$O(n+p)$-time Solutions include the Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp algorithms. These are included in the undergraduate textbook but will not be focused on due to the nature of this course.

TEXT INDEXING. Preprocessing is allowed; preprocess T by constructing a data structure called a **text index** for it. Text index speeds up text search each time a new pattern is given. Notice this is a strategy of trading space for (query) time.

Tries

As a warmup structure, a **trie** represents strings $T_1, ..., T_k$ as a rooted tree in which the child branches are labelled with letters of the alphabet Σ. Strings are represented as root-to-leaf paths; to do this, we terminate each string with a special symbol \$ which is less than any other symbol in Σ. If this is not done, we cannot distinguish prefixes as absent or present (see below example).

Tries continued on next page...

Example I. Given strings {ama,ann,anna,anne}. Starting at the root, only one child is present representing the letter "a". This is repeated (where the next child is n and the child after that is a) in order to arrive at what we can distinguish as a full string (ana) and a leaf node corresponding to this string. Notice this tree is not strictly bifurcating. If the $ symbol was not present, we would not be able to ascertain if the prefix was present in the set or not.

Properties. An in-order traversal of the leaves would provide the strings in a sorted lexicographic order. Furthermore, this structure allows us to determine if a pattern search string P is one of the strings in the set $T_1, ..., T_k$ (an exact match) by performing a top-down traversal.

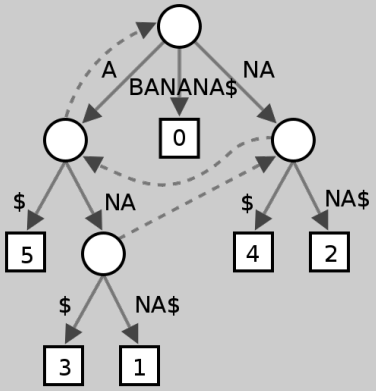
Analysis. Define n as the number of nodes in the trie. We claim that $n \leq \sum_{i=1}^{k} |T_i| + 1$. Notice that n is equal to this expression $\iff$ each word starts with a different letter in the set. This does not occur commonly in practice. If each node stored its pointers to its children a(n increasing) sorted order, the query time is $O(p \lg \sigma) = O(|P| \lg |\Sigma|)$ which is independent of n. Furthermore, the space is $O(n)$ + the space for $T_1, ..., T_k$.

COMPRESSED TRIE. Contract nonbranching paths in a trie to simple edges (e.g., an for ana in the above example). If these labels are still stored, the search can still be performed. Sometimes, people prefer to have, for each node, there are members of fixed length (e.g., rather than variable length) – in this case, they store the first letter as a label. In this case, though, searching is not possible any longer (because you do not know what you skipped).

Suffix Trees

A **suffix tree** is a **text index** and a **compressed trie** that is defined over all of the suffixes of the text (T\$).

Example II. Imagine you have text `banana$` with array indices 0123456. The first step of constructing a suffix tree is write down all suffixes, defined as $T[0..6], T[1..6], ...$ In this case, we have banana$, anana$, nana$, ana$, na$, a$, and $. From this, a compressed trie is built. Leaves contain starting position in the text (from there, the suffix can be recovered).



Notice there are $n+1$ leaves where n is the length of the text. An edge label is a substring $T[i..j]$; these two indices can therefore be stored as (i, j) instead.

Analysis. The space here is $O(n)$ because there are a constant amount of entries stored in each edge with at least two children. A search for P gives a subtree whose leaves correspond to all occurrences of P in the text. If each node stores pointers to children (in sorted order), the query time is $O(p \lg \sigma)$ because you must perform a binary search at each node.

Notice that we can use perfect hashing to make this $O(p)$. Reporting all occurrences would be $O(p + \text{occ})$.

Suffix Trees continued on next page...

Applications. Can be used to find the longest repeated substring in T by considering a suffix tree for T and finding the branching node of maximum "letter depth". In **Example II**, this is "ana". This is $O(n)$ by using a pre-order traversal. Also used in many areas of bioinformatics and for data mining. Notice there is still a price here, despite being the fastest method of text indexing – in practice this is around 15 to 20 times more than the character string itself.

Suffix Arrays

To save more space, consider sorting the suffixes of T and just store the indices. This array of indices is the **suffix array** and has $n+1$ words.

Example III. Refer to *Example II*. The suffix array for $T =$ `banana$` is 6531042 where the suffixes are sorted.

ANALYSIS. Text search involves a binary search of time $O(p \lg n)$ to see if P occurs in T by considering this array. A *simple accelerant* would be to consider L, R, the left and right boundaries of the current search interval. Have that the algorithm keep track of l, the length of the longest prefix of $T[SA[L]..n]$ that matches P and r for $T[SA[R]..n]$. Let $h = \min(l, r)$.

Example IV. Refer to *Example III*. L, R would transition from $0,6 \rightarrow 4,6 \rightarrow 4,4$. l, r would transition from $0,0 \rightarrow 4,0$.

Observation. The first h characters of P, $T[SA[L]..n]$, $T[SA[L+1]..n]$,...,$T[SA[R]..n]$ (all of the suffixes corresponding to a current search interval) agree; they are equal to $P[0..h-1]$. Therefore, for comparison in the binary search, start from $P[h]$. The initial values of l and r are acquired from direct comparison.

Super Accelerant. We must establish a number of definitions. The **longest common prefix** of S_1 and S_2 is the longest string that is a prefix of both S_1 and S_2. Let $\text{Lcp}(i, j)$ be the length of the longest common prefix of the suffixes specified in $SA[i]$ and $SA[j]$. Call some examination of a character in P *redundant* if that character has been examined before.

The goal of this accelerant is to reduce the number of redundant character examinations to at most one per iteration (of binary search) which is $O(\lg n)$ in total. Therefore, the total time would be $O(p + \lg n)$ since there would still be p redundant comparisons. This is done by building more data structures.

More preprocessing also needs to be done. For each triple (L, M, R) that can possible arise during binary search (all possible binary searches using any pattern P), precompute $\text{Lcp}(L, M)$ and $\text{Lcp}(R, M)$. The number of such triples is $(n+1) - 2 = n - 1$. The total amount of extra space necessary here is therefore $2n - 2$ (two arrays indexed by M).

Let $\text{Pos}(i)$ be the suffix $T[SA[i]..n]$.

Case 0. In the *simplest case*, in any iteration of the binary search, if $l = r$, then compare P to suffix $\text{Pos}(M)$, starting from $P[h]$, as before (simple accelerant).

In the *general case*, if $l \neq r$, assume without loss of generality that $l > r$:

- **Case 1**. $\text{Lcp}(L, M) > l$. Then the common prefix of suffix $\text{Pos}(L)$ and $\text{Pos}(M)$ is longer than the common prefix of P and $\text{Pos}(L)$. Therefore, $P > \text{Pos}[M]$ (due to the structure of the sorted array). No examinations of P are needed. Make L equal to M, leaving l and r unchanged.
- **Case 2**. $\text{Lcp}(L, M) < l$. Then $P < \text{Pos}(M)$. No examinations of P are needed. Make R equal to M. r becomes $\text{Lcp}(L, M)$. l remains unchanged.

Suffix Arrays continued on next page...

- **Case 3**. $\text{Lcp}(L, M) = l$. Then P agrees with $\text{Pos}(M)$ up to positive $l-1$. Start comparison from position l (as simple accelerant).

For the Super Accelerant, cases **1** and **2** require no examinations with P. For the two cases in which the algorithm examines a character, the comparison starts with $P[\max(l,r)]$. Suppose that k characters of P are examined in the iteration. Then, there are $k-1$ matches, and $\max(l,r)$ increases by $k-1$.

Hence, at the start of any iteration, $P[\max(l,r)]$ may have already been examined, but the next character in P has not been. Therefore, there are ≤ 1 redundant character examinations per iteration.

CONSTRUCTION. Suffix trees or arrays for strings over constant-sized alphabets can be constructed in $O(n)$ time. Similarly, if $\sigma \leq n$, this is also true.

Succinct Data Structures

Trees

Tries and suffix trees use a great amount of space, and while suffix arrays are better, the accelerant makes them use considerable more space (but still not more than suffix trees). Inverted lists and file provide an implementation with a smaller amount of space for similar functionality, but are slower.

The goal of **succinct data structures** is to use far less space than the previously discussed methods but still support fast search.

Number of trees on n nodes. For a binary tree, this is the n^{th} **Catalan number** $C_n = \frac{\binom{2n}{n}}{n+1}$. On the other hand, we can discuss **ordinal trees** (ordered trees). An ordinal tree is one where each node can have larger node degree (which may or may not be two). No notion of left or right children, just their index.

NOTE Bring own laptop for presentations or send presentation to the professor. Outline a general idea of the proof; no need to show results. Organize logically in whatever way that fits. Seven graduate students are in the class; can only schedule two presentations per class (last three classes). Can request to present as early as next week on a Wednesday.

A mapping between ordinal trees and binary trees can be done as follows. Every left child in the binary tree would be the first child in the ordinal tree. On the other hand, the right child would be the immediate right sibling in the ordinal tree. We can also prove that there is a unique mapping for any given ordinal tree. Also notice the root can never have a right child. Given this fact, the number of ordinal trees for n nodes is actually C_{n-1}.

Information-Theoretic Lower Bound. Consider a *combinatorial object* of size n (has n elements) where u is the number of different such objects. The information-theoretic lower bound of expressing this object is $\lg u$ bits.

Information-Theoretic Lower Bound of Representing a Binary Tree. For a binary tree, notice that we have $\lg C_n = \lg\binom{2n}{n} - \lg(n+1) = \lg\frac{(2n)!}{(n!)^2} - \lg(n+1) = \lg(2n)! - 2\lg(n!) - \lg(n+1)$. For further analysis, we use Stirling's Approximation and acquire $\lg C_n = 2n\lg(2n) - 2n\lg n + o(n)$. This is simplified to $2n + o(n)$. Even more exactly, $\lg C_n = 2n - \frac{3}{2}\lg n + O(1)$. Effectively, trees can be represented in $2n$ bits. Does this mean we can still navigate them? For this, we need succinct data structures.

REPRESENTING BINARY TREES. A number of these representations will be shown below.

Level-Order Representation of Binary Trees. Append an external node $\square$ for each missing child. For each node in *level order*, write 0 if external and 1 if internal. Notice that the space this requires is $2n+1$ bits (per tree).

Navigation. The left and right children of the ith internal node are at positions $2i$ and at $2i+1$ in the bit vector.

Trees continued on next page...

Proof. Induction on i. When i is 1, this is the root. This statement is clearly true since its left and right children must be the two successive positions in the vector. For $i > 1$, by the induction hypothesis, the children of the $(i-1)$st internal node are at position $2i-2$ and $2i-1$.

We have two cases: whether or not $(i-1)$st and ith internal nodes are at the same level, **(1)** and **(2)** respectively. Therefore, Case (2) would have the $i-1$st node be above the ith node. We can make the observation that the level ordering is preserved in children. That is to say, some node A's children would precede B's children $\iff A$ precedes B in level order. Furthermore, all nodes between the $(i-1)$st and ith internal nodes must be external nodes (with no children). So, the children of the ith internal node must immediately follow the $(i-1)$st internal node's children at positions $2i$ and $2i+1$. □

Rank and Select. A fundamental data structure in succinct data structures is some data structure that is capable of supporting two operations **Rank** and **Select** on bit vectors. We call $\text{rank}_1(i)$ as the number of 1s up to position i and $\text{select}_1(j)$ as the position of the jth 1.

If we identify each node by its position in the bit vector, then for this representation, $\text{left-child}(i) = 2\text{rank}_1(i)$ and $\text{right-child}(i) = 2\text{rank}_1(i)+1$. Similarly, $\text{parent}(i) = \text{select}_1(\lfloor i/2 \rfloor)$.

To support the rank operation, the most naive method would be to store the counts; this may be constant time but requires one word per position. Instead, split into $\lg^2 n$-bit **superblocks**. For each successive superblock, we store the cumulative rank (number of 1s) which requires $\lg n$ bits before the next superblock. Therefore, the amount of space that is necessary here is:

$$O\left(\lg n \frac{n}{\lg^2 n}\right) = O\left(\frac{n}{\lg n}\right) \tag{22}$$

We then split each superblock into $\frac{1}{2}\lg n$-bit **blocks**. For each block, again also store cumulative ranks *within its superblock*. Here, the amount of space that is required (in bits) is $O(\lg\lg n)$ for each of these. Overall, the amount of space that is necessary here (in bits) is:

$$O\left(\lg\lg n \frac{n}{\lg n}\right) = o(n) \tag{23}$$

We use a lookup table for a bit vector of length $\frac{1}{2}\lg n$. For example, for *any* possible bit vector of length $\frac{1}{2}\lg n$ and each of its position, store the answer. The amount of extra space necessary for this lookup table is:

$$2^{\frac{\lg n}{2}} \frac{1}{2} \lg n O(\lg\lg n) = O(\sqrt{n}\lg n \lg\lg n) = o(n) \tag{24}$$

Space necessary is therefore $n + o(n)$. Ultimately, we consider the $\text{rank}_1(i)$ as the number of 1s before the superblock containing i + the number of 1s before the block containing i in the superblock + the number of 1s in the block up to i. This requires $O(1)$ time by using the lookup table.

With this data structure, for a binary tree, we can store them in $2n + o(n)$ bits and allow $O(1)$ time to navigate them.

NOTE A2 sample solutions posted online. SRI this Wednesday. Please bring laptops.

Generalizing Rank and Select to Strings

Notation. We define the alphabet of our string(s) as $\Sigma = \{1, 2, ..., \sigma\}$. Let the string be $S[1..n]$.

Operations. We would like to support the following operations:

- access(x) retrieves $S[x]$.
- $\text{rank}_\alpha(i)$ retrieves the number of occurrences of α in $S[1..i]$.
- $\text{select}_\alpha(i)$ retrieves the position of the ith occurrence of α in S.

Example. Given string $S = aabacccdaddabbbc$, we would have access(8) = d, $\text{rank}_\text{a}(8) = 3$, and $\text{select}_\text{b}(3) = 14$.

An Easy Solution. Copy the string, and a bitvector B_α for each of the alphabet symbols α. Use 1s to map all of the positions where each character occurs. If all of these are stored, the running time for the respective operations, access, rank, and select, would be $O(1)$. However, for this, the space cost would be $n \lg \sigma + \sigma(n + o(n))$ in bits. Realize there are σ^n different strings of length n for an alphabet of size σ and that $\lg(\sigma^n) = n \lg \sigma$ — this solution is far from being succinct or ideal.

WAVELET TREES. Before we go further, we need to discuss a structure known as a **wavelet tree**. The structure is such that the *root* of this tree is some bit vector $B_\text{r}[1..n]$ where

$$B_\text{r}[i] = \begin{cases} 0, & \text{if } S[i] \text{ is a character in the smaller half of the alphabet} \\ 1, & \text{otherwise} \end{cases} \tag{25}$$

Example. Have string $S = aabececdahdabfgh$ where the bit vector B_r would be 0001010001000111 where 0 implies a character in $\{a, b, c, d\}$.

We call the subsequence of $S[1..n]$ formed by symbols in the first half of the alphabet $S_0[1..n_0]$ where n_0 is the number of zeroes. Similarly, we can describe $S_1[1..n_1]$. From this, the left child of the root r is S_0, and the right child of the root is S_1.

Recursively, we construct wavelet trees for these children respectively. At each level, we concatenate the bitvectors into a vector of length n. For each bitvector of length n, we store using rank/select structure. Subsequences and characters are not stored; only the vectors.

Space. The space occupied by this structure is $(n + o(n)) \lg \sigma = n \lg \sigma + o(n) \lg \sigma$ bits. This is much closer to the information-theoretic lower bound. But how do we support operations?

Operations. We can implement the following operations as so.

- access(i) would involve a top-down traversal of the tree. At the root, if access(i) = 1, then the position of the corresponding bit in the right child is $\text{rank}_1(i)$ over B_r. This also applies recursively. This is $O(\lg \sigma)$ running time.
- $\text{rank}_\alpha(i)$ would also involve a top-down traversal of the tree. You would take the rank of the bit corresponding to that half of the alphabet. Using what we learned above, a sequence of mappings can be done to understand how far needs to be travelled in each vector.
- $\text{select}_\alpha(i)$ requires more steps. First, a top-down traversal is done in order to locate the bitvector leaf that contains α. From there, a bottom-up traversal is done in order to find the answer in the original string (see below example). This is also $O(\lg \sigma)$.

Generalizing Rank and Select to Strings continued on next page...

Example. *Top-Down.* To find $\text{select}_g(1)$, we first find out how many times a,b,c,d,e,f occurs in S to determine the position of the leaf bitvector node. This is first done by determining how many times a,b,c,d occurs — $\text{rank}_0(16)$ where 16 is n. Then, we find out how many times e,f occurs — $\text{rank}_0(6)$. The first is equal to 10, the second is equal to 3 — therefore, the leaf is located at position 14 since $10+3=13$. *Bottom-Up.* We determine $\text{select}_0(1)$ and find 2. Notice that the 2nd position of 1 in the parent bitvector would be the position of the 2nd character in the parent node.

Succinct Text Indexes

BURROWS-WHEELER TRANSFORM . Recall the compression algorithm by the same name referred to earlier. More information is found here. Consider the following example.

$$T\# = \text{mississippi\#}$$

Given a string $T\#$, a string T terminated by an end-of-string character less than any other character, we determine all of its **cyclic shifts**. For example,

mississippi# ississippi#m ssissippi#mi ... #mississippi

We then lexicographically sort these into a sequence M.

#mississippi i#mississipp ippi#mississ ... ssissippi#mi

Compression. The last character in each row (string) of this sequence forms a string T^{BWT} = ipssm#pissii. In *BWT Compression*, what was done was starting from the original string and constructing this transform string in order to use MTF. A number of positions are generated for this list. From here and finally, Huffman Encoding was used. Even without the transform string, we would still have a compression algorithm but still not as good a one as with this transform. Characters in this transform string are grouped together by phrases appearing after them which is highly compressible.

Decompression requires a few steps but is not overtly complicated and works well in practice (reverse transformation can be done quickly). As was said before, the open-source software `bzip2` uses some variant of this algorithm to compress text.

Observation. Notice that $T^{\text{BWT}}[i]$ is the character before $T[SA[i]]$. This shows that the suffix array and the BWT strings are related. We can now make the following claims using our previous example.

- Since T = mississippi#, we have that Σ = {i,m,p,s}.
- We have that T^{BWT} = ipssm#pissii that can support rank and select using a wavelet tree.
- Notice that the SA positions have the following starting characters: 1 has "#", 2, 3, 4, and 5 have i, 6 has "m", 7 and 8 have "p", and the remaining suffixes start with "s".

Let $N[1..\sigma]$ be an array such that $N[i]$ is the total number of occurrences of character #, 1, 2, ..., $i-1$, cumulatively, in T. In this example, the elements in this array would be $1,5,6,8$.

Backward Search. Starting from the last character in some pattern P, we can count the number of occurrences of P in T using `Count`(T,P). For $i=p,p-1,...,1$, compute the interval $[s..e]$ of the suffix array SA whose corresponding suffixes are prefixed with $P[i..p]$.

```
Count(T,P)

  // Entire range of suffix array at first.
```

Succinct Text Indexes continued on next page...

```
s <-- 1
e <-- n

// Start at p.
i <-- p
while (i > 0 and s <= e)
  s <-- N[P[i]] + rank_P[i](T^BWT,s-1) + 1
  e <-- N[P[i]] + rank_P[i](T^BWT,e)

return max(e-s+1,0) // Valid range or no occurrences.
```

Example. How many times does "ssi" occur in "mississippi"? In the beginning of this algorithm, $[s..e] = [1..12]$. Starting at $i = 3$ (where $P[i] =$ "i"), $[s..e] = (1+0+1, 1+4) = [2..5]$ (notice these are all the suffixes that start with "i"). When $i = 2$ (where $P[i] =$ "s"), $[s..e] = (8+0+1, 8+2) = [9..10]$. Finally, when $i = 1$ (where $P[i] =$ "s"), we have that $[s..e] = (8+2+1, 8+4) = [11..12]$. We know that "ssi" therefore occurs twice.

Correctness Analysis. We update the values of s and e correctly.

Proof. Suppose that $[s..e]$ corresponds to suffixes that are prefixed (that start) with $P[i+1..p]$. Also suppose that $P[i] = \alpha$. The entries of SA corresponding to the suffix array that starts with α occupy the interval $[N[\alpha]+1..N[\alpha+1]]$. Because all such suffixes start with α, they are sorted according to the suffixes whose starting positions are one character after their starting positions. Therefore, the smallest suffix prefixed by $P[i..p]$, and the smallest suffix prefixed by $P[i+1..p]$ that follows α are one character apart in T by their starting positions. Recognize that $\text{rank}_\alpha(T^{\text{BWT}}, s-1)$ computes how many suffixes smaller than $P[i+1..p]$ follow character α in T. So, the expression used to update s points to the smallest suffix that starts with $P[i..p]$. □