53 Space-efficient data structures

In this chapter, we will explore space-efficient data structures. This may sound like a boring topic at first – after all, many of the commonly-used data sctructures have linear space complexity, which is asymptotically optimal. However, in this chapter, we shall use a much more fine-grained notion of space efficiency and measure space requirements in bits.

Imagine we have a data structure whose size is parametrized by some parameter n (e.g. number of elements). Let us define $X(n)$ as the universe of all possible values that a size-n data structure (as a whole) can hold. For example if we have a data structure for storing strings from a fixed alphabet, $X(n)$ may be the universe of all length-n strings from this alphabet.

Let us denote $s(n)$ the number of bits needed to store a size-n data structure. The information-theoretical optimum is $OPT(n) := \lceil \log |X(n)| \rceil$ (which is essentially the entropy of a uniform distribution over $X(n)$).

Note: We will always ignore constant additive factors, so sometimes we will use the definition $OPT(n) := \log |X(n)|$ (without rounding, differs by at most one from the original definition) interchangably.

Definition: *Redundancy* of a space-efficient data structure is $r(n) := s(n) - OPT(n)$.

Now we can define three classes of data structures based on their fine-grained space efficiency:

Definition: A data structure is

- *implicit* when $s(n) \leq OPT(n) + \mathcal{O}(1)$, i.e., $r(n) = \mathcal{O}(1)$,
- *succinct* when $s(n) \leq OPT(n) + \mathrm{o}(OPT(n))$, i.e., $r(n) = \mathrm{o}(OPT(n))$,
- *compact* when $s(n) \leq \mathcal{O}(OPT(n))$.

A typical implicit data structure contains just its elements in some order and nothing more. Examples include sorted arrays and heaps.

Note that some linear-space data structures are not even compact – because we are counting bits now, not words. For example, a linked list representing a length-n sequence of numbers from range $[m]$ needs $\mathcal{O}(n(\log n + \log m))$ bits ($\log n$ bits are used to represent a next-pointer), whereas OPT is $n \log m$. For $n \gg m$, this does not satisfy the requirements for a compact data structure.

And of course, as with any data structure, we want to be able to perform reasonably fast operations on these space-efficient data structures.

53.1 Representation of strings over arbitrary alphabet

Let us consider the problem of representing a length-n string over alphabet $[m]$, for example a string of base-10 digits. The following two naive approaches immediately come to mind:

(a) Consider the whole string as one base-10 number and convert that number into binary. This achieves the information-theoretically optimum size of $OPT(n) = \lceil n \log 10 \rceil \approx 3.32n = \Theta(n+1)$. However, this representation does not support local decoding and modification – you must always decode and re-encode the whole string.

(b) Store the string digit-by-digit. This uses space $n\lceil \log 10 \rceil = 4n = OPT(n) + \Theta(n)$. For a fixed alphabet size, this is not succinct because $\Theta(n) > o(OPT(n)) = o(n+1)$⟨1⟩. However, we get constant-time local decoding and modification for free.

We would like to get the best of both worlds – achieve close-to-optimum space requirements while also supporting constant-time local decoding and modification.

A simple solution that may work in practice is to encode the digits in groups (e.g. encode each 2 subsequent digits into one number from the range [100] and convert that number to binary).

With groups of size k, we get

$$s(n) = \lceil n/k \rceil \lceil k \log 10 \rceil \le (n/k + 1)(k \log 10 + 1) = \underbrace{n \log 10}_{OPT(n)} + n/k + \underbrace{k \log 10 + 1}_{\mathcal{O}(1)}.$$

Thus we see that with increasing k, redundancy goes down, approaching the optimum but never quite reaching it. For a fixed k it is still linear and thus our scheme is not succinct. Also, with increasing k, local access time goes up. In practice, however, one could chose a good-compromise value for k and happily use such a scheme.

We will develop a succinct encoding scheme later in this chapter.

53.2 Intermezzo: Prefix-free encoding of bit strings

Let us forget about arbitrary alphabets for a moment and consider a different problem. We want to encode a binary string of arbitrary length in a way that allows the decoder to determine when the string ends (it can be followed by arbitrary other data). Furthermore,

⟨1⟩ More formally, if we consider $\mathcal{O}$ and o to be sets of functions, $\Theta(n) \cap o(n+1) = \emptyset$.

we want this to be a streaming encoding – i.e., encode the string piece by piece while it is being read from the input. The length of the string is not known in advance – it will only be determined when the input reaches its end⟨2⟩

A trivial solution might be to split the string into b-bit blocks and encode each of them into a $(b+1)$-bit block with a simple padding scheme:

- For a complete block, output its b data bits followed by a one.
- For an incomplete final block, output its data bits, followed by a one and then as many zeros as needed to reach $b+1$ bits.
- If the final block is complete (input length is divisible by b), we must add an extra padding-only block (one followed by b zeros) to signal the end of the string.

The redundancy of such encoding is at most $n/b+b+1$ (one bit per block, $b+1$ for extra padding block). For a fixed b, this is $\Theta(n)$, so the scheme is not succinct.

SOLE (Short-Odd-Long-Even) Encoding

In this section we will present a more advanced prefix-free string encoding that will be succinct.

First, we split the input into b-bit blocks. We will add a padding in the form of $10\cdots0$ at the end of the last block to make it b bits long. If the last block was complete, we must add an extra padding-only block to make the padding scheme reversible.

Now we will consider each block as a single character from the alphabet $[B]$, where $B := 2^b$. Then we shall extend this alphabet by adding a special EOF character. We will add this character at the end of encoding. This gives us a new string from the alphabet $[B+1]$ that has length at most $n/b+2$ (+1 for padding, +1 for added EOF character).

However, as $B+1$ is not a power of two, now we have a question of how to encode this string. Note that this is a special case of the problem stated above, i.e. encoding a string from an arbitrary alphabet. We will try to solve this special case as a warm-up and then move on to a fully general solution.

First, we need to introduce a new concept: re-encoding character pairs into different alphabets. Let's assume for example, that we have two characters from alphabets [11] and [8], respectivelly. We can turn them into one character from the alphabet [88] (by the simple transformation of $8x+y$). We can then split that character again into two in a different way. For example into two characters from alphabets [9] and [10]. This can be

⟨2⟩ If the length were known in advance, we could simply store the length using any simple variable-size number encoding, followed by the string data itself. This would give us $\mathcal{O}(\log n)$ redundancy almost for free.

accomplished by simple division with remainder: if the original character is $z \in [88]$, we transform in into $\lfloor z/10 \rfloor$ and $(z \bmod 10)$. For example, if we start with the characters 6 and 5, they first get combined to form $6 \cdot 8 + 5 = 53$ and then split into 5 and 3.

We can think of these two steps as a single transformation that takes two characters from alphabets [11] and [8] and transforms them into two characters from alphabets [9] and [10]. More generally, we can always transform a pair of characters from alphabets $[A]$ and $[B]$ into a pair from alphabets $[C]$ and $[D]$ as long as $C \cdot D \geq A \cdot B$ (we need an output universe large enough to hold all possible input combinations).

We will use this kind of alphabet re-encoding by pairs heavily in the SOLE encoding. The best way to explain the exact scheme is with a diagram (fig. 53.1).

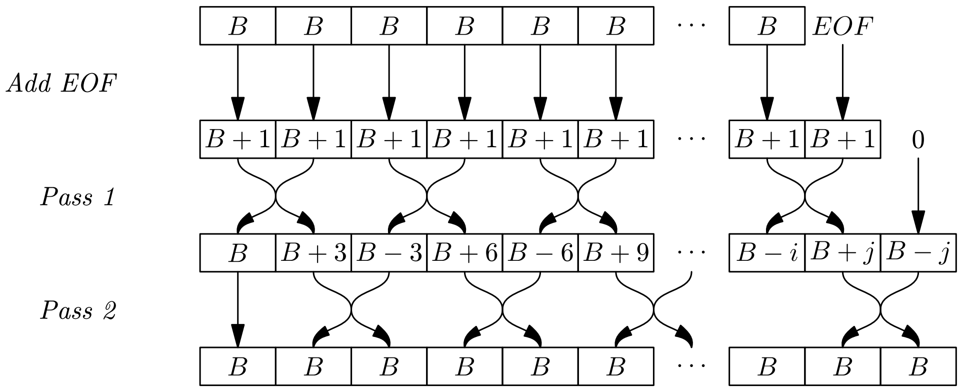

Figure 53.1: SOLE alphabet re-encoding scheme

There are two re-encoding phases. The first transforms blocks with alphabet $[B + 1]$ into blocks with variable alphabet sizes (of the form of alternating $[B + 3k]$, $[B - 3k]$). This is the origin of the name: after this pass, odd-numbered blocks have smaller alphabets than even-numbered ones. The second pass runs phase shifted by one block and converts the variable-alphabet blocks into blocks with alphabet $[B]$.

What is the redundancy of this scheme? Let us count how the number of blocks increases throughout the encoding passes:

- If the original length was a multiple of b, we must add one block to complete padding.
- We always add one block with EOF character.
- Before the first pass, we may need to add an extra padding block to make number of blocks even (not shown in fig. 53.1).

- Before the second pass, we always add an extra padding block to make number of blocks odd.

In total, we add at most 4 blocks. Thus $r(n) \le 4b$.

For the scheme to work, we need to set $b \ge 2\log n + 2$ (so $B \ge 4n^2$). This gives us redundancy $r(n) = \mathcal{O}(\log n)$. Thus we have a succinct scheme. Also, one block fits into $\mathcal{O}(1)$ words on a RAM, so we can do constant-time arithmetic on the blocks.

Note that this representation is locally decodable and modifiable – each input block affects at most 4 output blocks.

Now we must check that all the alphabet transformations are valid, i.e., the output universe of each transformation is always at least as big as the input universe.

For the first pass, we want:

$$\begin{gathered}(B+1)^2 \le (B-3i)(B+3i+3)\\ B^2+2B+1 \le B^2+3B-9i^2-9i\\ B \ge 9i^2+9i+1\end{gathered}$$

We know $B \ge 4n^2$ and $i \le \frac{n+1}{2}$. By plugging $i = \frac{n+1}{2}$ and $B = 4n^2$ and doing some algebraic manipulation, we can verify that the inequality holds. For smaller i the right-hand side decreases so it holds for those too.

For the second pass, this is trivial, as $(B+i)(B-i) = B^2 - i^2 \le B^2$.

53.3 Mixers as a building block for succinct structures

A reinterpretation of the SOLE encoding

There is another way of looking at the SOLE encoding from the previous section. We can group the alphabet translations into "encoding boxes" that take input from the alphabet $(B+1)^2$, output the alphabet B^2 and the part of the information that did not fit into the output is passed as a "carry"[3] to the next encoding box (similarly to how carrying works when doing addition). See fig. 53.2. We will also call these boxes *mixers*.

The start and end of the encoding are irregular, but we will ignore that for now. An important property of these boxes is that outgoing carry does not depend on incoming carry (unlike in addition). This allows for local decoding and modification. Otherwise a

[3] Sometimes the alternative term *spill* is used instead.

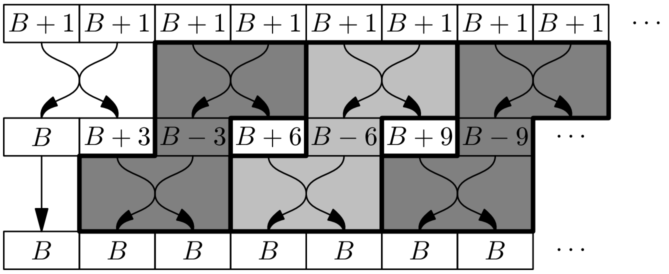

Figure 53.2: SOLE interpreted as a chain of mixers

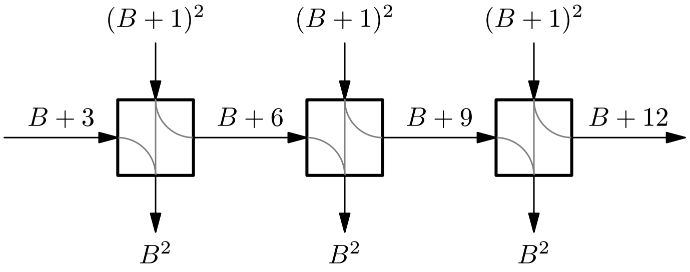

Figure 53.3: SOLE high-level mixer diagram

single input change could affect the whole output. Now we can describe this scheme in a more abstract, high-level way (fig. 53.3).

In our case, the input alphabet size is always $(B+1)^2$, the output alphabet size is B^2 and the carry alphabet sizes form the sequence $B+3i$. Given that the output alphabet is smaller than the input alphabet, it makes sense that the carry alphabet has to increase in size to accomodate the accumulating information that did not fit into the output. The final carry is then used to output some extra blocks at the end.

Generalizing the mixer concept

At a high level, a mixer can be thought of as a mapping $f : [X] \times [Y] \to [2^M] \times [S]$ with the property that when $(m,s) = f(x,y)$, s depends only on x. This is the key property that allows local decoding and modification because carry does not cascade.

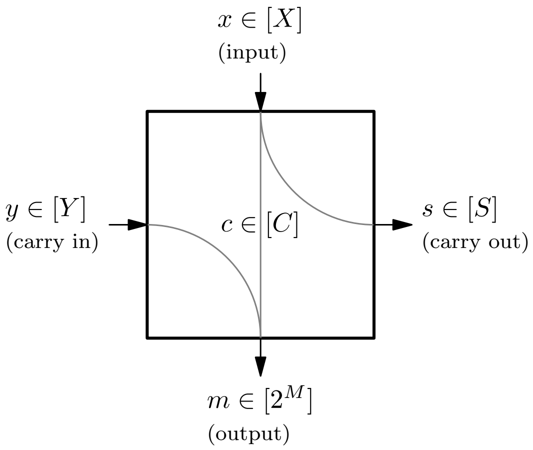

Figure 53.4: General structure of a mixer

Internally, the a mixer is always implemented as a composition of two mappings, f_1 that transforms $x \to (c, s)$ and f_2 that transforms $(y, c) \to m$. See fig. 53.4. Both f_1 and f_2 must be injective so that the encoding is reversible.

The mappings f_1 and f_2 themselves are trivial alphabet translations similar to what we used in the SOLE encoding. You can for example use $f_1(x) = (\lceil x/S \rceil, x \bmod S)$ and $f_2(y, c) = c \cdot Y + y$.

Thus implementing the mixer is simple as long as the parameters allow its existence. A mixer with parameters X, Y, S, M can exist if and only if there exists C such that $S \cdot C \geq X$ and $C \cdot Y \leq 2^M$ (once again, the alphabet translations need their range to be as large as their domain in order to work).

Lemma: A mixer f has the following properties (as long as all inputs and outputs fit into a constant number of words):

- f can be computed on a RAM in constant time
- s depends only on x, not y
- x can be decoded given m, s in constant time
- y can be decoded given m in constant time

All these properties should be evident from the construction.

Definition: The redundancy of a mixer is

$$r(f) := \underbrace{M + \log S}_{\text{output entropy}} - \underbrace{(\log X + \log Y)}_{\text{input entropy}}.$$

In general, the redundancy of a mapping (with possibly multiple inputs and multiple outputs) is the sum of the logs of the output alphabet size, minus the sum of the logs of the input alphabet sizes. Note that there is no rounding (because the inputs and outputs can be from arbitrary alphabets, not necessarily binary) and the redundancy can be non-integer. Compare this to the concept of redundancy for space-efficient datastructures defined above.

On the existence of certain kinds of mixers

Now we would like to show that mixers with certain parameters do exist.

Lemma: For X, Y there exists a mixer $f : [X] \times [Y] \to [2^M] \times [S]$ such that:

- $S = \mathcal{O}(\sqrt{X})$, $2^M = \mathcal{O}(Y \cdot \sqrt{X})$
- $r(f) = \mathcal{O}(1/\sqrt{X})$

Proof: First, let's assume we have chosen an M (which we shall do later). Then we want to set C so that it satisfies the inequality $C \cdot Y \leq 2^M$. Basically we are asking the question how much information can we fit in m in addition to the whole of y. Clearly we want C to be as high as possibly, thus we set $C := \lfloor 2^M / Y \rfloor$.

Now let us calculate the redundancy. First we shall note that we can compute redundancy for f_1 and f_2 separately and add them up:

$$\begin{aligned} r(f) &= M + \log S - \log X - \log Y \\ &= (M - \log C - \log Y) + (\log C + \log S - \log X) \\ &= r(f_2) + r(f_1) \end{aligned}$$

This is just a telescopic sum. It works similarly for more complex mapping compositions: as long as each intermediate result is used only once as an input to another mapping, you can just sum the redundancies of all the mappings involved.

For example, if you have a mapping composition as in fig. 53.5, you can easily see $r(g) = r(g_1) + r(g_2) + r(g_3)$. For every edge fully inside the composition, the same number is added once and subtracted once.

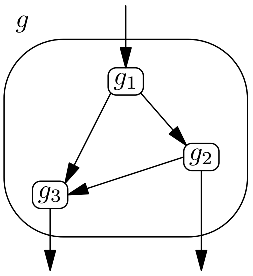

Figure 53.5: Mapping composition

First, we shall estimate $r(f_2)$:

$$r(f_2) = M - \log(Y \cdot C) = M - \log(\overbrace{Y \cdot \lfloor 2^M / Y \rfloor}^{\geq 2^M - Y})$$
$$r(f_2) \leq M - \log(2^M - Y) = \log \frac{2^M}{2^M - Y} = \log \frac{1}{1 - \frac{Y}{2^M}}$$

Now we shall use a well-known inequality form analysis:

$$e^x \geq 1 + x$$
$$x \geq \log(1 + x)$$
$$-x \leq \log \frac{1}{1 + x}$$

By substituting $x \to -x$ we get:

$$x \geq \log \frac{1}{1 - x}$$

Thus

$$r(f_2) \leq \frac{Y}{2^M} = \mathcal{O}\left(\frac{1}{C}\right)$$

Now to $r(f_1)$:

$$r(f_1) = \log C + \log S - \log X = \log C + \log \left\lceil \frac{X}{C} \right\rceil - \log X = \log \left(\frac{C \left\lceil \frac{X}{C} \right\rceil}{X} \right)$$

$$r(f_1) \leq \log \left(\frac{X + C}{X} \right) = \log \left(1 + \frac{C}{X} \right) \leq \frac{C}{X} \qquad \text{(because } \log(x) \leq x - 1\text{)}$$

Putting this together:

$$r(f) = r(f_1) + r(f_2) \leq \mathcal{O}\left(\frac{1}{C} + \frac{C}{X} \right)$$

In order to minimize this sum, we should set $C = \Theta\left(\sqrt{X}\right)$. Then $r(f) = \mathcal{O}\left(1/\sqrt{X}\right)$ and $S = \left\lceil \frac{X}{\Theta(\sqrt{X})} \right\rceil = \Theta\left(\sqrt{X}\right)$ as promised. Note that this holds for any value of Y.

However, we cannot freely set C, as we have already decided that $C := \lfloor 2^M/Y \rfloor$. Instead, we need to set a value for M that gives us the right C.

The whole mixer parameter selection process could be as follows (it may be useful to refer back to fig. 53.4):

1. We are given X, Y as parameters.
2. Set $M := \left\lceil \log\left(Y\sqrt{X}\right) \right\rceil$.
3. Set $C := \lfloor 2^M/Y \rfloor$. This ensures that $2^M \geq C \cdot Y$ and gives us $C = \Theta\left(\sqrt{X}\right)$.
4. Set $S := \lceil X/C \rceil$. This ensures that $C \cdot S \geq X$ and gives us $S = \Theta\left(\sqrt{X}\right)$.

All the inequalities required for mixer existence are satisfied and based on the analysis above the parameters satisfy what our lemma promised. □

53.4 Succinct representation of arbitrary-alphabet strings

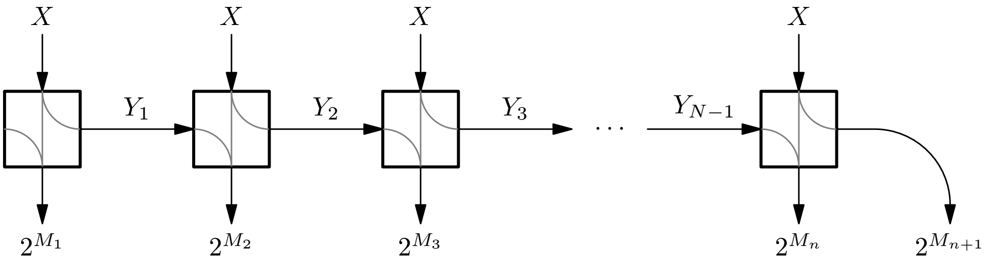

Figure 53.6: Mixer chain for string encoding

A naive first try

We would like to use mixers to encode string from an arbitrary alphabet into the binary alphabet. Let's assume we have a string $A \in [\Sigma]^n$. We shall split it into some blocks of size k, which gives us a block alphabet $[X] = [\Sigma^k]$. Then we could use a mixer chain as in fig. 53.6, similar to what we did in the SOLE encoding.

The intuition behind this is simple: whatever part of X did not fit into a whole number of bits is sent on as carry and whenever a whole extra bit of information has accumulated in the chain, it can be output. The final carry is output at the end using the neccessary number of bits. Here we don't mind rounding because it is an additive constant.

Everything is also locally decodable and modifiable – to decode i-th input block, you only need i-th and $(i+1)$-st output blocks. And vice versa, you only need modify these two output blocks after changing the i-th input block.

Now we just need to set k and calculate redundancy. It will be useful to set $k \approx 2\log_\Sigma n$. Then $X \approx n^2$ and by previous lemmas, $Y_i \in \mathcal{O}(n)$ and redundancy of the mixers is $\mathcal{O}(1/n)$. As there is less than n mixers, the total redundancy is $\mathcal{O}(1)$.

That all sounds wonderful. However, there is one serious problem. Each of the mixers will have differrent parameters (Y_i, M_i, $S_i = Y_{i+1}$). In order to compute the parameters for i-th mixer, we need to know the parameters for the $(i-1)$-st, namely the $Y_i = S_{i-1}$. For that, we need the $(i-2)$-nd and so on...

If we did encoding / decoding in a streaming fashion, this would not matter – we could compute the mixer parameters one by one as we go. But if we wish for random access in constant time, we would need to store a table of all the mixer parameters – i.e., a table with $\Theta(n/\log_\Sigma n)$ rows. That is impractical.

Note that this was not an issue for sole as there the Y_i's formed an arithmetic sequence. They weren't even the optimal Y_i's that would be created by the generic mixer construction but a close enough approximation that still yielded good results, up to an additive constant. That was a special case – in general, we do now know how to approximate the mixer parameters by something easier to compute locally.

A tree encoding to the rescue

To remedy the situation, instead of a chain, we will organize mixers into a binary tree. Each vertex will contain one mixer whose carry output goes to its parent (thus most vertices receive two carry inputs but it is trivial to combine them into one). This is depicted in fig. 53.7. Now we need $Y \cdot Z \cdot C \leq 2^M$.

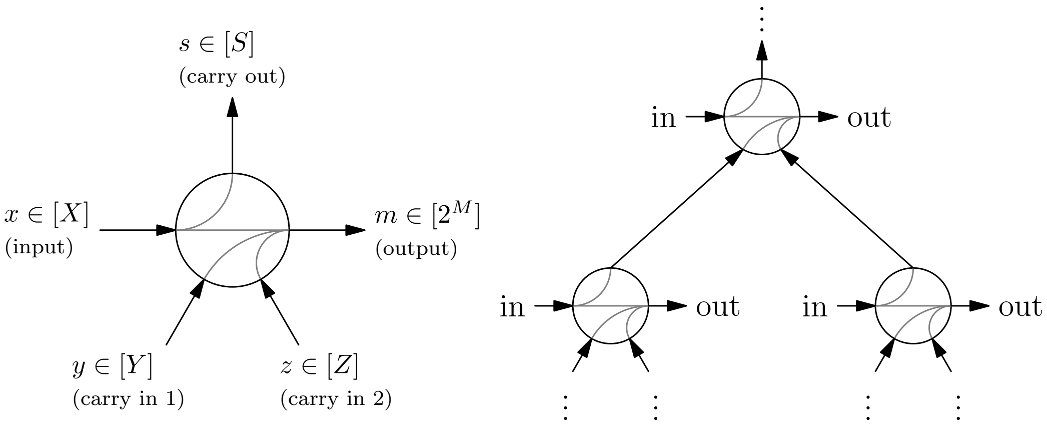

Figure 53.7: A single mixer vertex and the organization of those into a tree

Then you can create a linear order on the vertices (e.g. by layers bottom-to-top), split the input string into blocks and feed the blocks through the mixer vertices in this order and save the corresponding outputs in the same order.

Note that this scheme still has all the nice properties, for example it is locally decodable. To decode a vertex's input, you only need the output of that vertex and its parent.

But how does a tree help us determine individual mixer parameters more easily? The parameters of a mixer in a vertex are uniquely determined by the shape of the subtree under that vertex. This is easily seen by induction: all leaves have the same parameters (as they have dummy carry-in alphabets of size 1) and the parameters of any vertex are determined by the parameters of its children.

We will use the same tree shape as for binary heaps: all the levels are full, except for possibly the last and in the last level all the vertices in one contiguous segement starting at the very left.

Now let us consider a level at height h (from the bottom). There are at most three three vertex types by subtree shape and they appear on the level in a specific order:

1. a contiguous segment of vertices with full subtrees of height h (type A)
2. one vertex with an irregular subtree (type B)
3. a contiguous segment of vertices with full subtrees of height $h-1$ (type C)

See fig. 53.8. If the last level happens to be full, there are only type-A vertices.

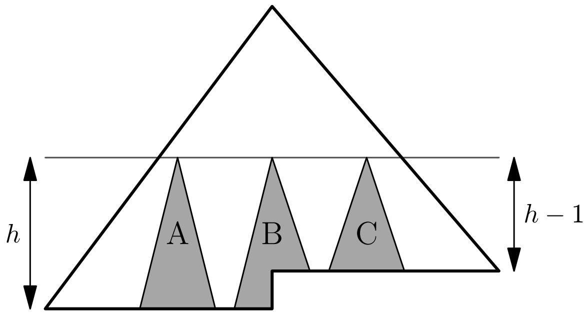

Figure 53.8: Vertex types by subtree shape

Thus, for each level and each vertex type, it is sufficient to remember:

- Number of vertices of this type on this level. From this, we can easily determine vertex type from its index by simple comparison.
- Mixer parameters.
- Starting address of the output of first vertex of this type in the output stream. From this, we can easily compute starting address of any vertex by simple addition and multiplication as all vertices of a given type on a given level have the same number of output bits (parameter M). This will be useful for local decoding.

This a precomputed table of $\mathcal{O}(\log n)$ words.

Block size and redundancy computation is exactly the same as in the chain case and we still get $\mathcal{O}(1)$ redundancy. The chain can be thought of as a degenerate case of the tree construction where the tree has the shape of a path (and thus all subtrees have distinct shapes and distinct mixer parameters).

Local decoding of i-th input block could be done as follows:

- Convert block index into a position in the tree (level + index on level)
- Determine the vertex type and mixer parameters, compute position in output stream and extract the corresponding output $m \in 2^M$
- Do the same for the parent vertex
- Using the parent mixer, decode the carry going up from our vertex
- Using our mixer, decode the original input block from our output and carry

Local modification can be done in a similar fashion the other way around. Both take $\mathcal{O}(1)$ time on RAM.

Theorem: On a Word-RAM, we can represent a string $A \in [\Sigma]^n$ in space $\lceil n \log \Sigma \rceil + \mathcal{O}(1)$ bits, with random-access element read and write operations in $\mathcal{O}(1)$ time, using a precomputed table of $\mathcal{O}(\log n)$ constants dependent on n and Σ.