In this lecture we will consider funnels — a cache-oblivious data structure that we will use for building funnel heap, a cache-oblivious priority queue.

1 Funnel data structure

Funnels are cache-oblivious data structures used for merging sorted sequences. We can view them as binary trees that have an input buffer at each leaf, and an output buffer at the root. A funnel with $k$ leaves is called a $k$-funnel. The size of the output buffer of a $k$-funnel is $k^3$. A $k$-funnel has $2k - 1$ nodes. We can view a $k$-funnel as consisting of $\sqrt{k} + 1 \sqrt{k}$-funnels, one at the top and one for each leaf of the top $\sqrt{k}$-funnel, with output buffers of the bottom funnels acting as the input buffer for the top one. In the example above, $T_0$ is the top $\sqrt{k}$-funnel, $T_1, T_2, \ldots, T_{\sqrt{k}}$. The size of a buffer connecting two $\sqrt{k}$-funnels is $\sqrt{k}^3 = k^{3/2}$. Funnels are using the van Emde Boas
memory layout, the top $\sqrt{k}$-funnel is stored first, followed by the leaf $\sqrt{k}$-funnels. For our example the memory layout would be: $T_0T_1T_2 \cdots T_{\sqrt{k}}$. The $\sqrt{k}$-funnels are also using the van Emde Boas memory layout. The smallest funnel is the 2-funnel and its output buffer size is $2^3 = 8$.

2 Filling the output buffer

![Diagram of funnel and buffers]

```plaintext
Procedure Fill(u)
while $B_u$ not full do
    if $B_v$ empty then
        Fill(v);
    end
    if $B_w$ empty then
        Fill(w);
    end
    perform one merge step;
end
```

When filling the buffer $B_u$ we check before each merge step if the buffers $B_v$ and $B_w$ of the two children still contain any elements. If that is not the case, we fill the buffer that is empty.

**Theorem 1.** The space complexity of a $k$-funnel, excluding the input and output buffers, is $O(k^2)$.

**Proof.** The space complexity of the $k$-funnel is defined by the recurrence:

$$S(k) = S(\sqrt{k}) + \sqrt{k}S(\sqrt{k}) + \sqrt{k}k^{3/2}$$

$$= (1 + \sqrt{k}) \cdot S(\sqrt{k}) + O(k^2)$$

$$= O(k^2)$$

(1)

**Theorem 2.** Assuming $M = \Omega(B^2)$, a merging step using $k$-funnel that output $k^3$ elements takes $O(\frac{k^3}{\log M} \frac{k^4}{\log M})$ I/Os.

**Proof.** Let a $k$-funnel be the largest funnel that fits in main memory. It will take $O(\tilde{k}^2)$ space, with $\tilde{k}^2 \leq M$. We shall call it a base funnel (see Figure 1).

Consider the path beginning at the root and ending at a leaf of the $k$-funnel. There will be $\frac{\log k}{\log \tilde{k}} = \log_{\tilde{k}} k$ base funnels on this path (see Figure 2).
Figure 1: A base funnel – the largest $k$-funnel that fits in internal memory.

Figure 2: A path within a $k$-funnel consisting of base funnels.
The I/O complexity of merging $\bar{k}^3$ in a $\bar{k}$-funnel is: $O\left(\bar{k} + \frac{\bar{k}^3}{B}\right)$ I/Os. Since the next biggest funnel is the $\bar{k}^2$-funnel with size $O(\bar{k}^4) > M > B^2$, $\frac{\bar{k}^2}{B} > 1 \Rightarrow \frac{\bar{k}^3}{B} > \bar{k}$. And, therefore, $O\left(\bar{k} + \frac{\bar{k}^3}{B}\right) = O\left(\frac{\bar{k}^3}{B}\right)$

Therefore, we spend $O\left(\frac{1}{B}\right)$ I/O’s per element when operating on base funnels. Each element passes through $\frac{\log k}{\log k}$ base funnels. Then the total of I/O’s that an element needs to go from an input buffer to an output buffer of a $k$-funnel is:

$$O\left(\frac{1}{B} \cdot \frac{\log k}{\log k}\right) = O\left(\frac{1}{B} \cdot \frac{\log k}{\log \log M}\right) = O\left(\frac{1}{B} \cdot \log M k\right)$$

Because we process $k^3$ elements, the total I/O will be:

$$O\left(\frac{k^3}{B} \cdot \log M k\right) = O\left(\frac{k^3}{B} \cdot \log M_B k^3\right) = O\left(\frac{k^3}{B} \left(\log M_B k^3 + \log M_B / B\right)\right)$$

Assuming $M = \Omega(B^2)$, $\log M_B / B \leq \log B = 1$ and the total I/O complexity of merging using a $k$-funnel outputting $k^3$ items from $k$ sorted streams is

$$O\left(\frac{k^3}{B} \cdot \log \frac{M}{B} \cdot \frac{k^3}{B}\right).$$

\[\square\]

3 Funnel sort

Now we show how we can implement a cache-oblivious merge sort using funnels. We cannot simply use an $N$-funnel because it will take too much space. Instead we will use the following recursive procedure.

1. Split A into $N^\frac{1}{2}$ sub-arrays;
2. Recursively sort each sub-array;
3. Merge the $N^\frac{1}{2}$ streams using a $N^\frac{1}{2}$-funnel;

Procedure $\text{SORT}(A)$
The space complexity of funnel sort is defined by the following recursion

\[ S(k) = O(k) + O(N^{\frac{2}{3}}) = O(N) \]

The I/O complexity of funnel sort is:

\[
Q(N) = N^{\frac{1}{3}} Q(N^{\frac{2}{3}}) + O\left( \frac{N}{B} \log \frac{M}{W} \frac{N}{B} \right)
\]

\[ = O\left( \frac{N}{B} \log \frac{M}{W} \frac{N}{B} \right) \]

which is optimal I/O complexity of sorting \( N \) items.

4 Funnel heap

In this section we describe a cache-oblivious priority queue based on the funnel heap data structure.
The funnel heap consists of multiple chained links. The image above shows one link $L_i$. It consists of a $k_i$-funnel with an output buffer $B_i$. A 2-merger then merges the elements of the buffer $B_i$ with the elements of the next link and outputs them in the $A_i$ buffer. The size of the $A_i$ and $B_i$ buffers is $k_i^3$. The size of the $k_i$ input buffers to the $k_i$-merger is $s_i$.

The sizes $s_i$ and $k_i$ for each link $L_i$ are defined recursively as follows.

$$(k_1, s_1) = (2, 8)$$
$$\left(k_i, s_i\right) = \left(\lceil\lceil \sqrt[3]{s_{i-1}} \rceil \right), s_{i-1} \cdot (k_{i-1} + 1)\right),$$

where $\lceil x \rceil$ represent the smallest number that is a power of 2 and is greater than $x$.

The links $L_1, L_2, \ldots$ are connected to each other as follows:

Funnel heap maintains the items in heap order, meaning the items on the path from the first element of buffer $A_1$ to every leaf $s_{ij}$ are in in non-decreasing order. Thus, the smallest element in the heap is always the element $A_1[0]$. This leads to the following simple procedure for the priority queue `DELETEMIN()` operation. That is we fill the buffer $A_1$ if it’s empty and return the smallest element in it.

Obviously, the hardest part is maintaining the heap order during the insertion.
**DELETEMIN()**

if $A_1$ empty then
  Fill($A_1$);
end

Return and delete $A_1[0]$;

**INSERT(x)**

1. Let $S_{ij}$ be the first empty leaf buffer;
2. Empty all $L_r(r < i)$ by marking $A_i$ as empty and repeatedly calling DELETEMIN();
3. Empty the path from $S_{ij}$ to $A_i$;
4. Merge the two sorted sequences producing a single sorted sequence of all the removed items;
5. Place the two sorted sequences on the path from $A_1$ to $S_{ij}$ in such a way that the buffers $A_r(r \leq i)$ and $B_i$ contain the same number of items as before they were removed from it;

Note that the removed elements will fit in the buffers on the path because the total items remaining after all buffers $A_r(r \leq i)$ and $B_i$ have been filled is at most $\sum_{r=1}^{i-1} s_r \cdot (k_r + 1)$, which is at most $|s_{ij}| = s_i$ by definition of $s_i$.

In order to analyse the I/O complexity observe that an item might participate in the removal (and merging) multiple times. However, each time it is removed, it will never be placed in a funnel of lower links than where it was removed from. And once it reaches the largest link $L_i$ (with largest $i$) in its lifetime, it will move in the future only upward and to the left. The number of I/Os it'll spend going up the link $L_i$ is at most $O\left(\frac{1}{B} \cdot s_i\right)$.

Thus, the total I/Os that we spend on moving the item up within the funnels is

$$\sum_{r=1}^{i} O\left(\frac{1}{B} \log_M s_r \right) = O\left(\sum_{r=1}^{i} \frac{1}{B} \log_M 2^{(4/3)^r}\right) = O\left(\sum_{r=1}^{i} \frac{1}{B} (4/3)^r \cdot \log_M 2\right)$$

$$= O\left(\frac{1}{B} (4/3)^{Max} \cdot \log_M 2\right) = O\left(\frac{1}{B} \cdot \log_M 2^{(4/3)^{Max}}\right)$$

$$= O\left(\frac{1}{B} \cdot \log_M s_{Max}\right) = O\left(\frac{1}{B} \cdot \log_M N\right)$$

$$= O\left(\frac{1}{B} \cdot \log_M/B \frac{N}{B}\right)$$

The last equality is under our usual assumption that $M = \Omega(B^2)$.

Thus, the amortized I/O complexity of DELETEMIN() and INSERT(x) operations on the cache-oblivious priority queue is $O\left(\frac{1}{B} \cdot \log_M/B \frac{N}{B}\right)$, just like in the EM model.
5 Applications

Using the cache-oblivious priority queue we can implement all the graph algorithms we have considered in the EM model.