CSE 638: Advanced Algorithms

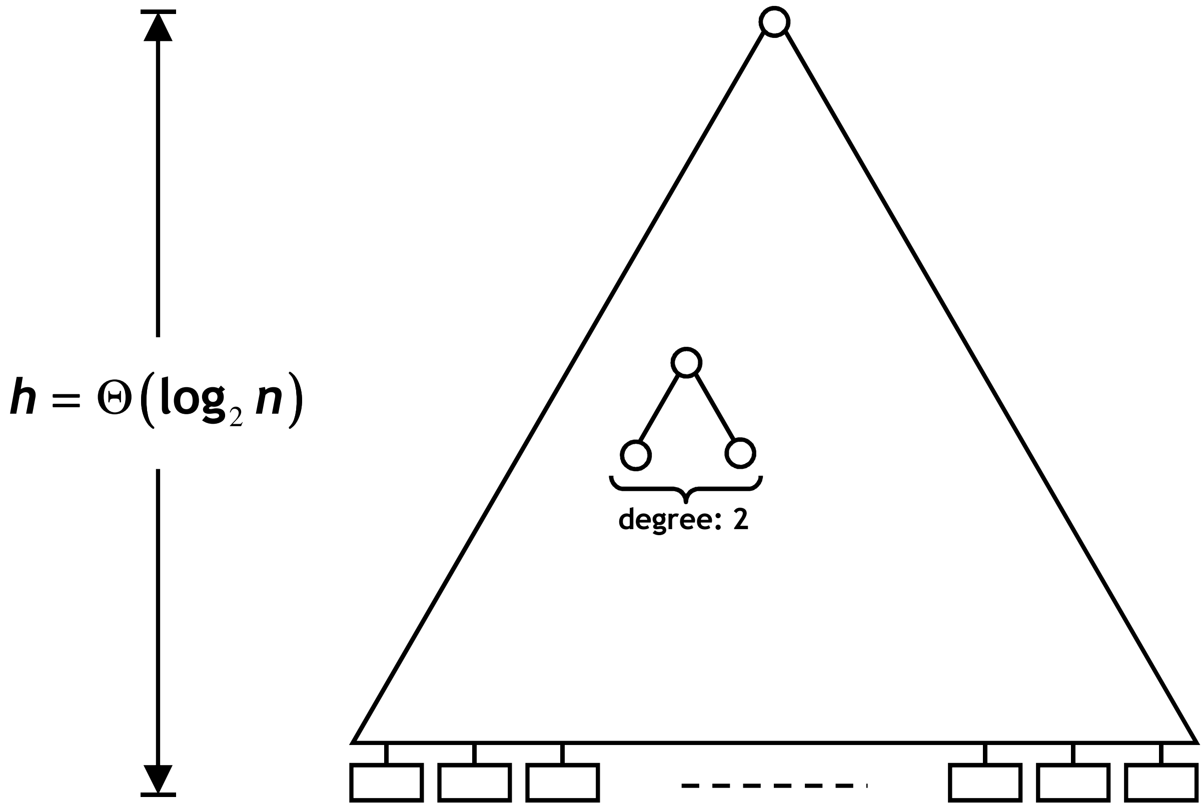
Lectures 18 & 19
(Cache-efficient Searching and Sorting)

Rezaul A. Chowdhury

Department of Computer Science
SUNY Stony Brook
Spring 2013

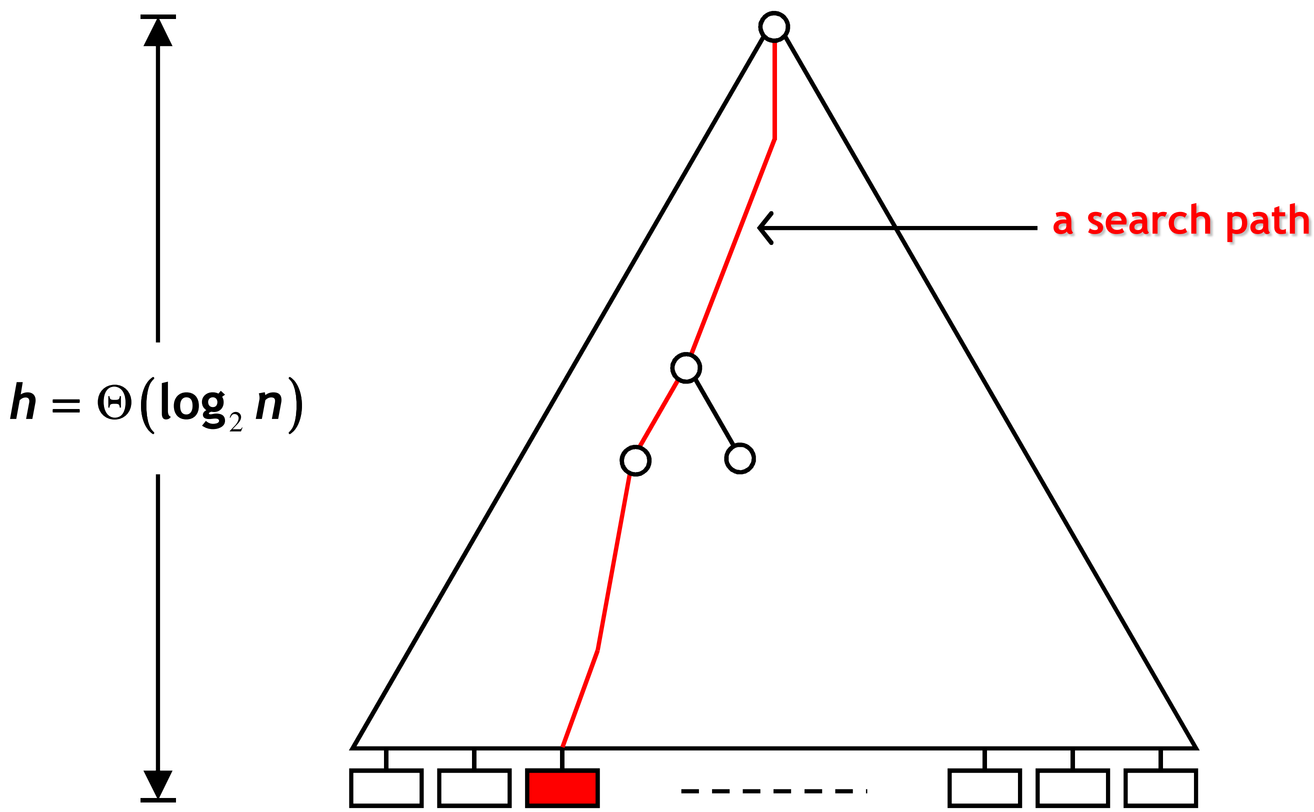
Searching
(Static B-Trees)

A Static Search Tree



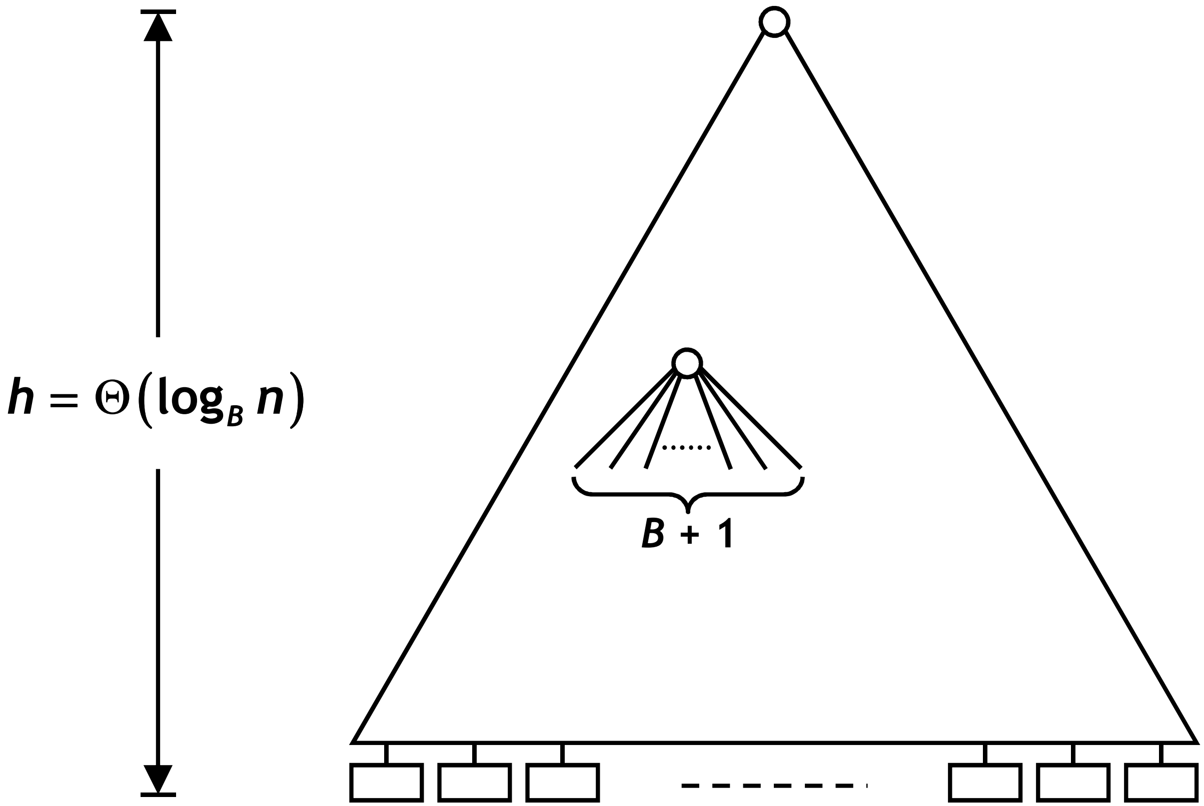
- A perfectly balanced binary search tree
- Static: no insertions or deletions
- Height of the tree, $h = \Theta(\log_2 n)$

A Static Search Tree



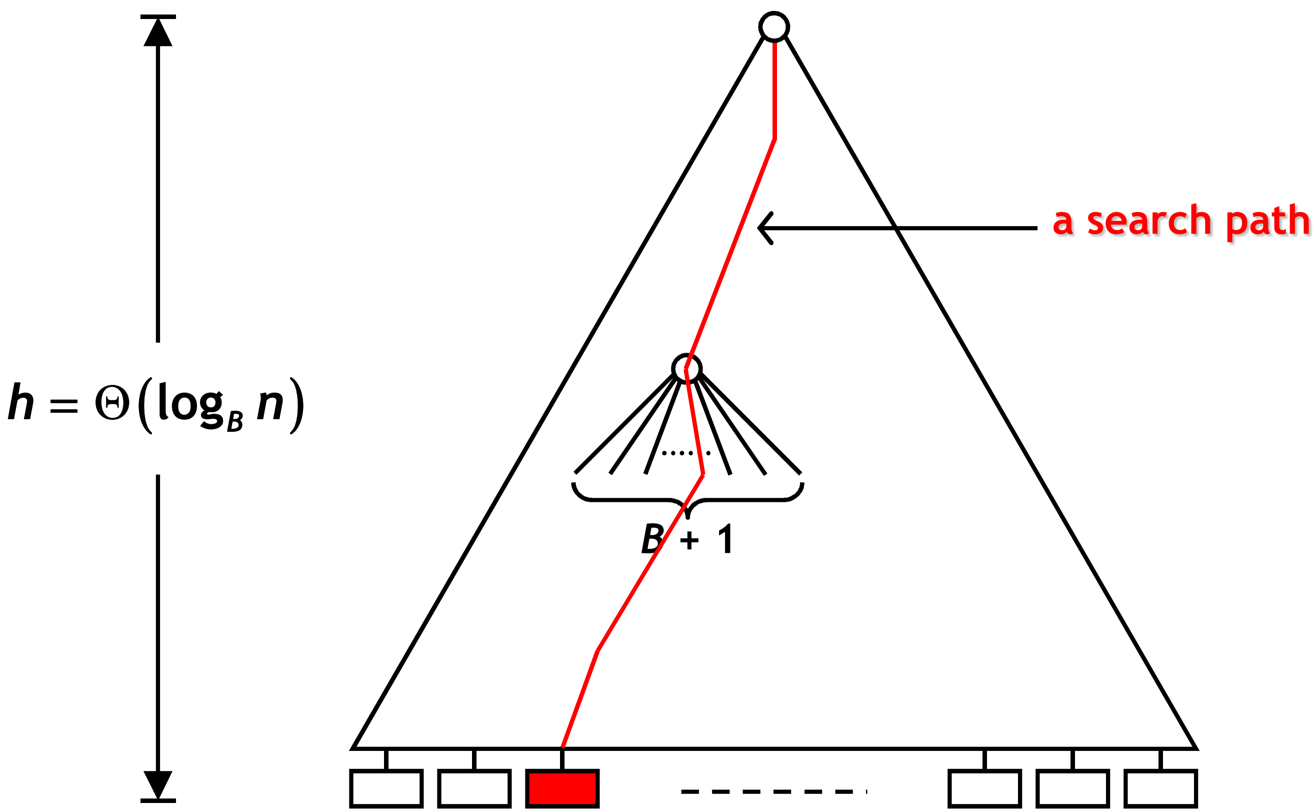
- A perfectly balanced binary search tree
- Static: no insertions or deletions
- Height of the tree, $h = \Theta(\log_2 n)$
- A search path visits $O(h)$ nodes, and incurs $O(h) = O(\log_2 n)$ I/Os

I/O-Efficient Static B-Trees



- Each node stores B keys, and has degree $B + 1$
- Height of the tree, $h = \Theta(\log_B n)$

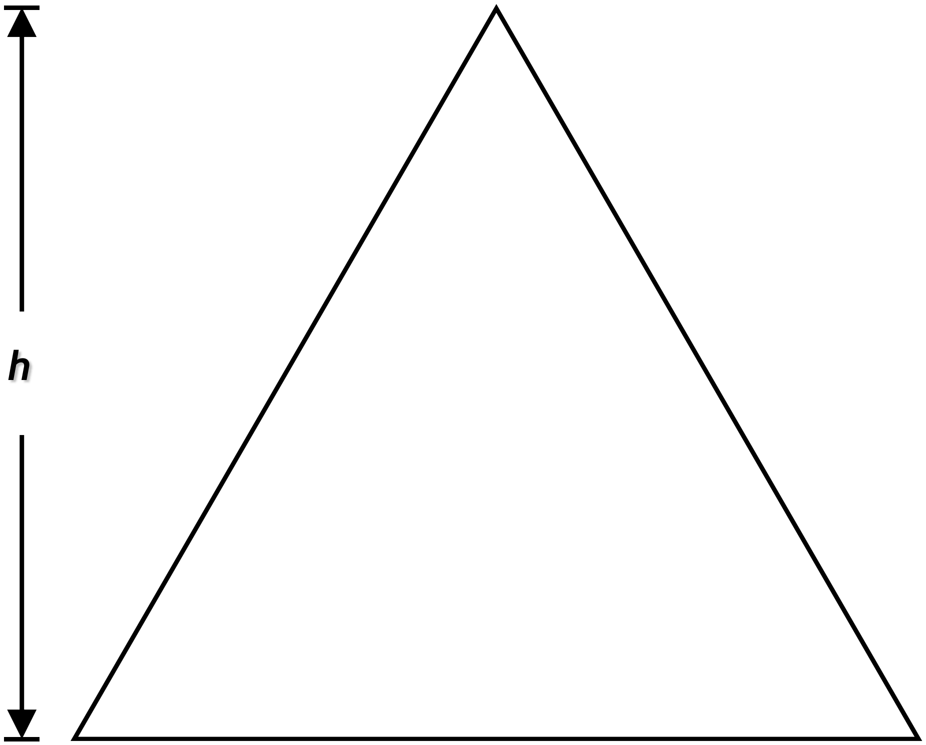
I/O-Efficient Static B-Trees



- Each node stores B keys, and has degree $B + 1$
- Height of the tree, $h = \Theta(\log_B n)$
- A search path visits $O(h)$ nodes, and incurs $O(h) = O(\log_B n)$ I/Os

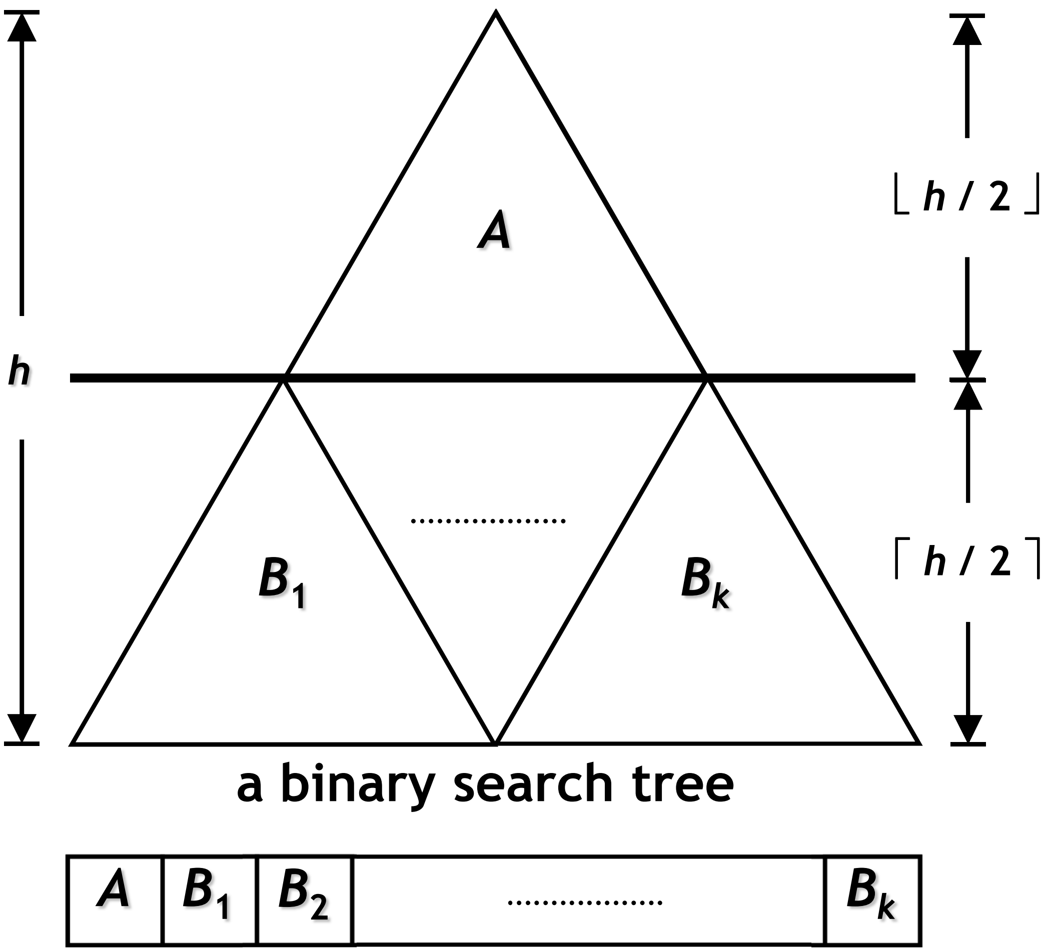
Cache-Oblivious Static B-Trees?

van Emde Boas Layout



a binary search tree

van Emde Boas Layout

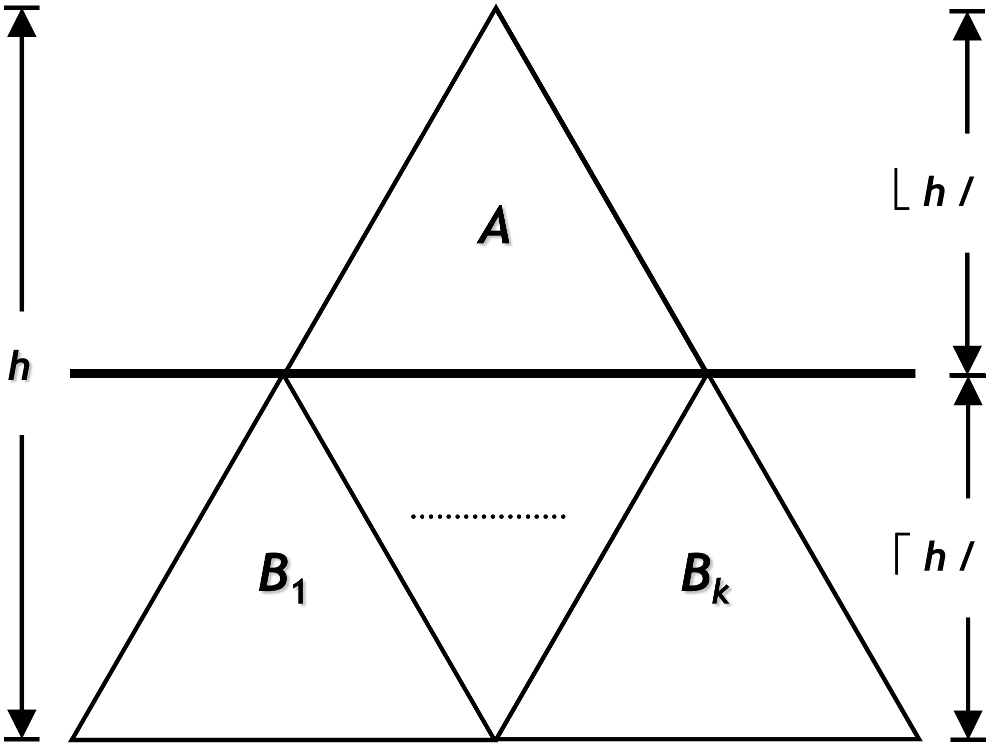


If the tree contains n nodes,

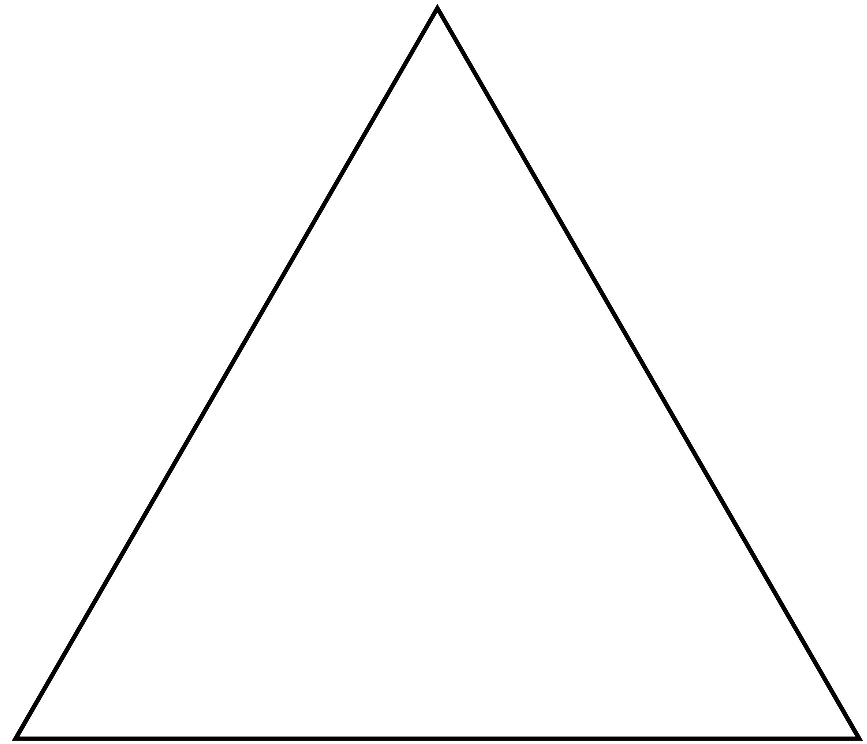
each subtree contains $\Theta\left(2^{h/2}\right) = \Theta(\sqrt{n})$ nodes,

and $k = \Theta(\sqrt{n})$.

van Emde Boas Layout



a binary search tree



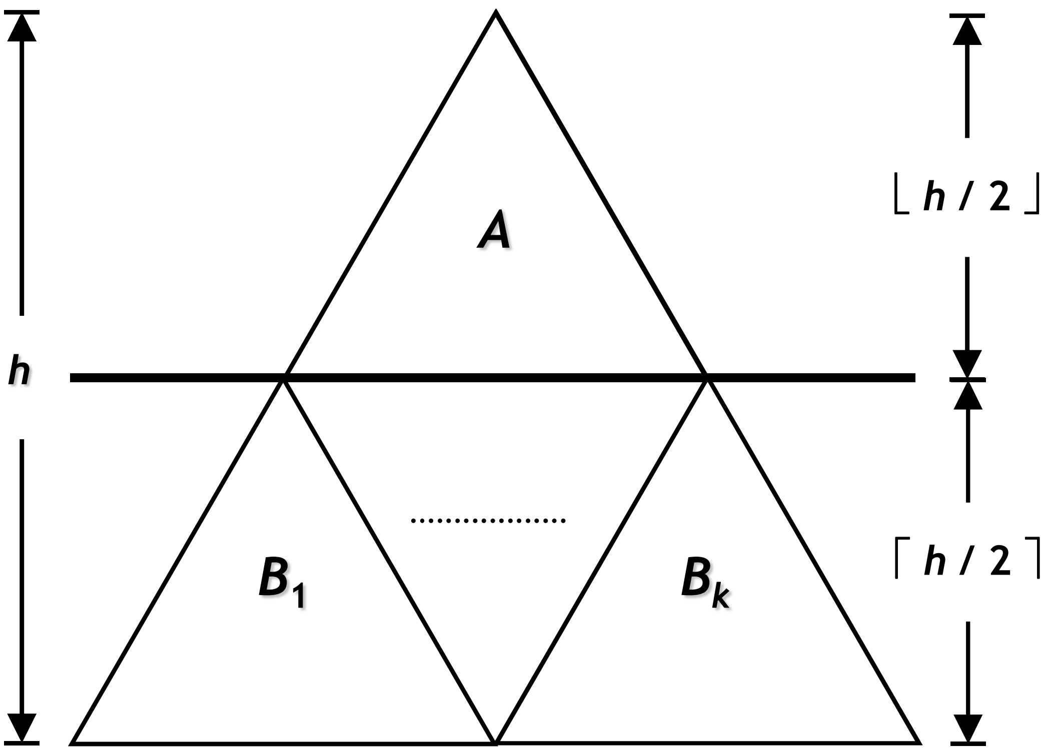
Recursive Subdivision

If the tree contains n nodes,

each subtree contains $\Theta\left(2^{h/2}\right) = \Theta(\sqrt{n})$ nodes,

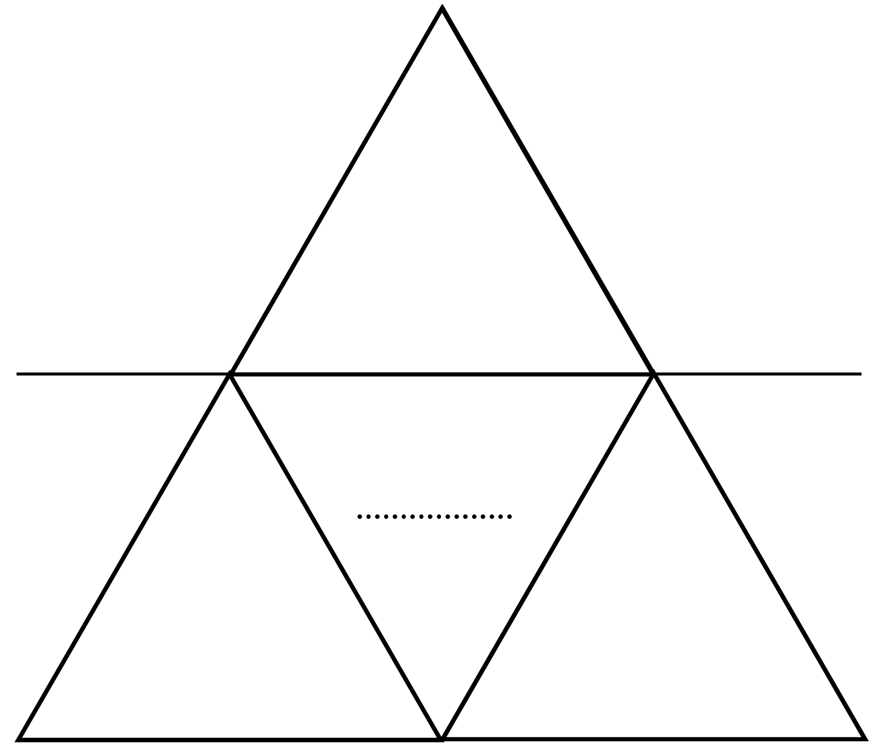
and $k = \Theta(\sqrt{n})$.

van Emde Boas Layout



a binary search tree

A	B_1	B_2	....	B_k



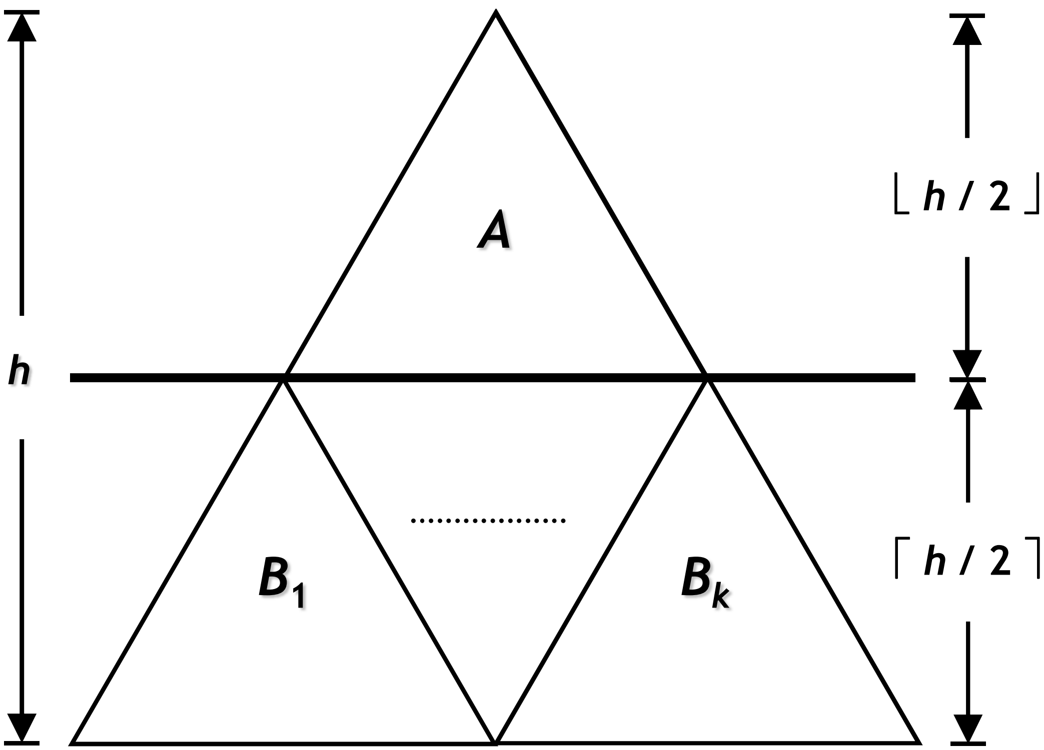
Recursive Subdivision

If the tree contains n nodes,

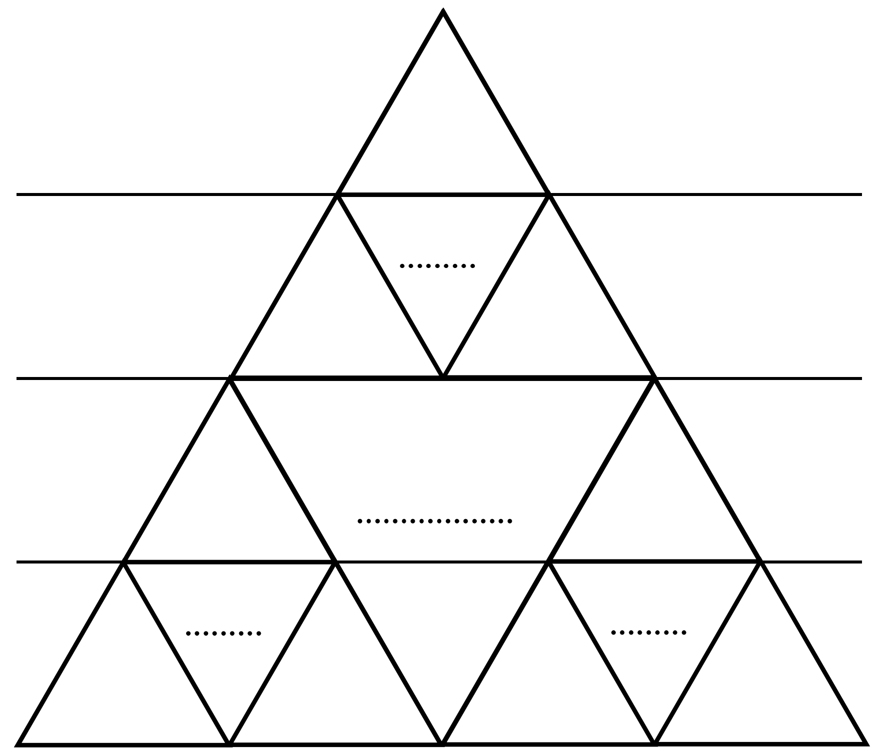
each subtree contains $\Theta\left(2^{h/2}\right) = \Theta(\sqrt{n})$ nodes,

and $k = \Theta(\sqrt{n})$.

van Emde Boas Layout



a binary search tree



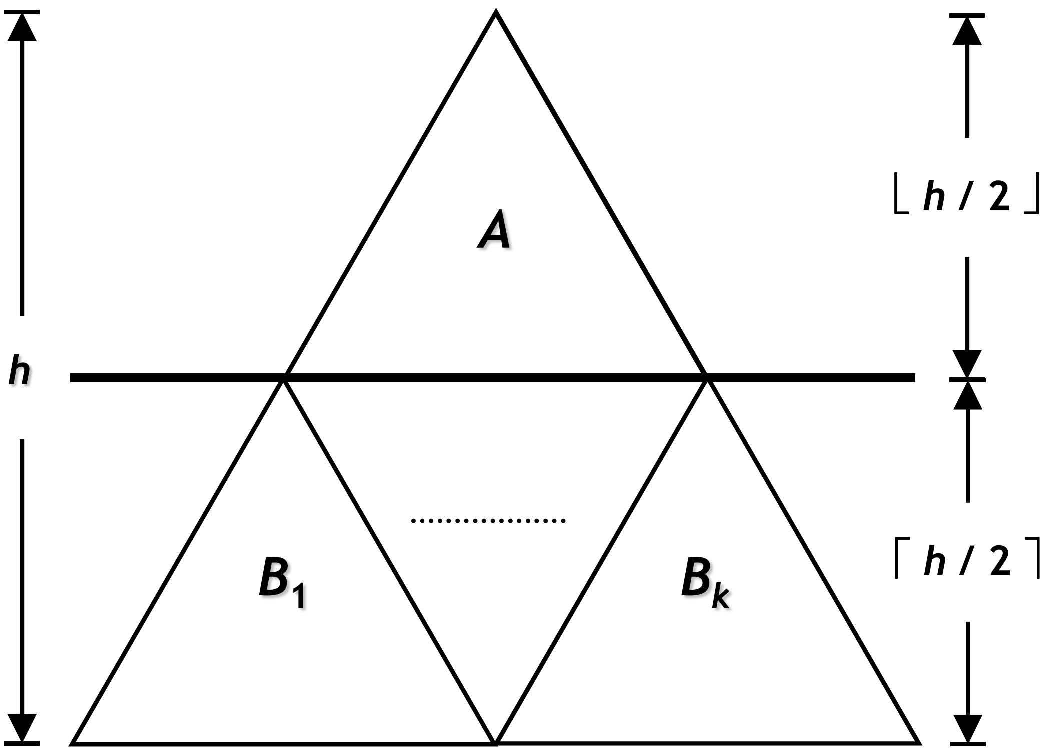
Recursive Subdivision

If the tree contains n nodes,

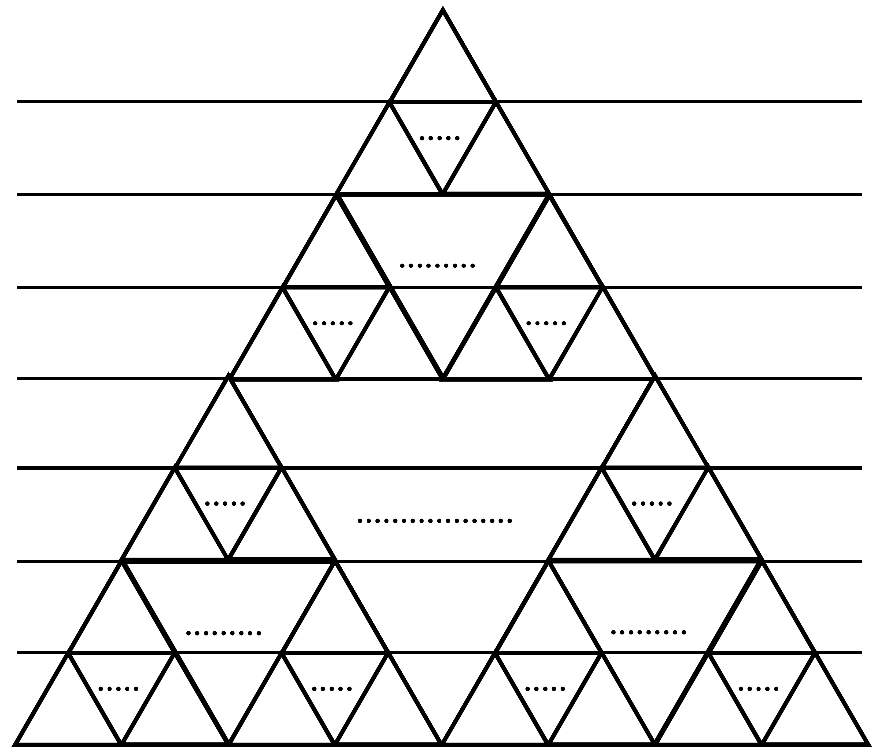
each subtree contains $\Theta\left(2^{h/2}\right) = \Theta(\sqrt{n})$ nodes,

and $k = \Theta(\sqrt{n})$.

van Emde Boas Layout



a binary search tree



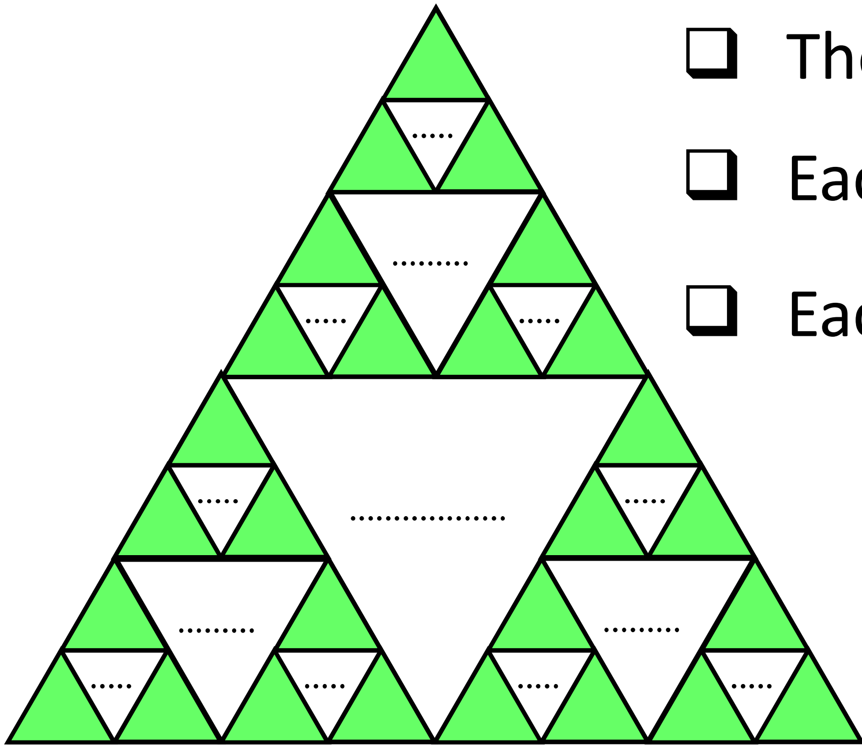
Recursive Subdivision

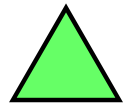
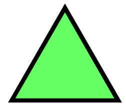
If the tree contains n nodes,

each subtree contains $\Theta\left(2^{h/2}\right) = \Theta(\sqrt{n})$ nodes,

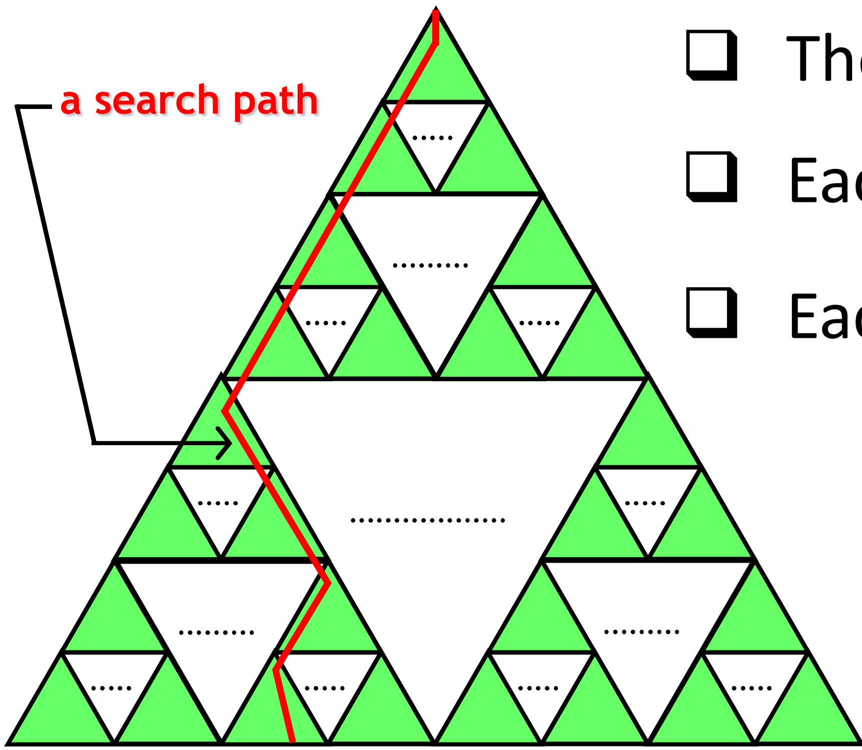
and $k = \Theta(\sqrt{n})$.

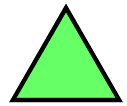
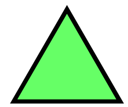
I/O-Complexity of a Search

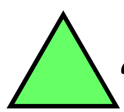


- The height of the tree is $\log n$
- Each  has height between $\frac{1}{2}\log B$ & $\log B$.
- Each  spans at most 2 blocks of size B.

I/O-Complexity of a Search



- The height of the tree is $\log n$
- Each  has height between $\frac{1}{2}\log B$ & $\log B$.
- Each  spans at most 2 blocks of size B.

- p = number of 's visited by a search path
- Then $p \geq \frac{\log n}{\log B} = \log_B n$, and $p \leq \frac{\log n}{\frac{1}{2}\log B} = 2\log_B n$
- The number of blocks transferred is $\leq 2 \times 2\log_B n = 4\log_B n$

Sorting
(Mergesort)

Merge Sort

Merge-Sort (A, p, r)   { *sort the elements in* $A[\,p \ldots r\,]$ }

1. *if* $p < r$ *then*
2. $\quad q \leftarrow \lfloor (p + r) / 2 \rfloor$
3. $\quad$ *Merge-Sort* (A, p, q)
4. $\quad$ *Merge-Sort* (A, $q + 1$, r)
5. $\quad$ *Merge* (A, p, q, r)

Merging *k* Sorted Sequences

- $k \geq 2$ sorted sequences $S_1, S_2, \ldots, S_k$ stored in external memory
- $|S_i| = n_i$ for $1 \leq i \leq k$
- $n = n_1 + n_2 + \cdots + n_k$ is the length of the merged sequence S
- S (initially empty) will be stored in external memory
- Cache must be large enough to store
    - one block from each S_i
    - one block from S

Thus $M \geq (k + 1)B$

Merging *k* Sorted Sequences

- Let $\mathcal{B}_i$ be the cache block associated with S_i, and let $\mathcal{B}$ be the block associated with S (initially all empty)
- Whenever a $\mathcal{B}_i$ is empty fill it up with the next block from S_i
- Keep transferring the next smallest element among all $\mathcal{B}_i$s to $\mathcal{B}$
- Whenever $\mathcal{B}$ becomes full, empty it by appending it to S
- In the *Ideal Cache Model* the block emptying and replacements will happen automatically $\Rightarrow$ cache-oblivious merging

I/O Complexity

- Reading S_i: #block transfers $\leq 2 + \frac{n_i}{B}$
- Writing S: #block transfers $\leq 1 + \frac{n}{B}$
- Total #block transfers $\leq 1 + \frac{n}{B} + \sum_{1 \leq i \leq k} \left(2 + \frac{n_i}{B}\right) = O\left(k + \frac{n}{B}\right)$

Cache-Oblivious 2-Way Merge Sort

```
Merge-Sort ( A, p, r )        { sort the elements in A[ p … r ] }

if p < r then
    q ← ⌊ ( p + r ) / 2 ⌋
    Merge-Sort ( A, p, q )
    Merge-Sort ( A, q + 1, r )
    Merge ( A, p, q, r )
```

I/O Complexity: $$Q(n) = \begin{cases} O\left(1 + \frac{n}{B}\right), & if\ n \leq M, \\ 2Q\left(\frac{n}{2}\right) + O\left(1 + \frac{n}{B}\right), & otherwise. \end{cases}$$

$$= O\left(\frac{n}{B} \log \frac{n}{M}\right)$$

How to improve this bound?

Cache-Oblivious k-Way Merge Sort

I/O Complexity: $Q(n) = \begin{cases} O\left(1 + \frac{n}{B}\right), & if\ n \leq M, \\ k \cdot Q\left(\frac{n}{k}\right) + O\left(k + \frac{n}{B}\right), & otherwise. \end{cases}$

$$= O\left(k \cdot \frac{n}{M} + \frac{n}{B} \log_k \frac{n}{M}\right)$$

How large can k be?

Recall that for k-way merging, we must ensure

$$M \geq (k+1)B \Rightarrow k \leq \frac{M}{B} - 1$$

Cache-Aware $\left(\frac{M}{B}-1\right)$-Way Merge Sort

I/O Complexity: $Q(n) = \begin{cases} \mathrm{O}\left(1+\frac{n}{B}\right), & if\ n \le M, \\ k \cdot Q\left(\frac{n}{k}\right) + \mathrm{O}\left(k+\frac{n}{B}\right), & otherwise. \end{cases}$

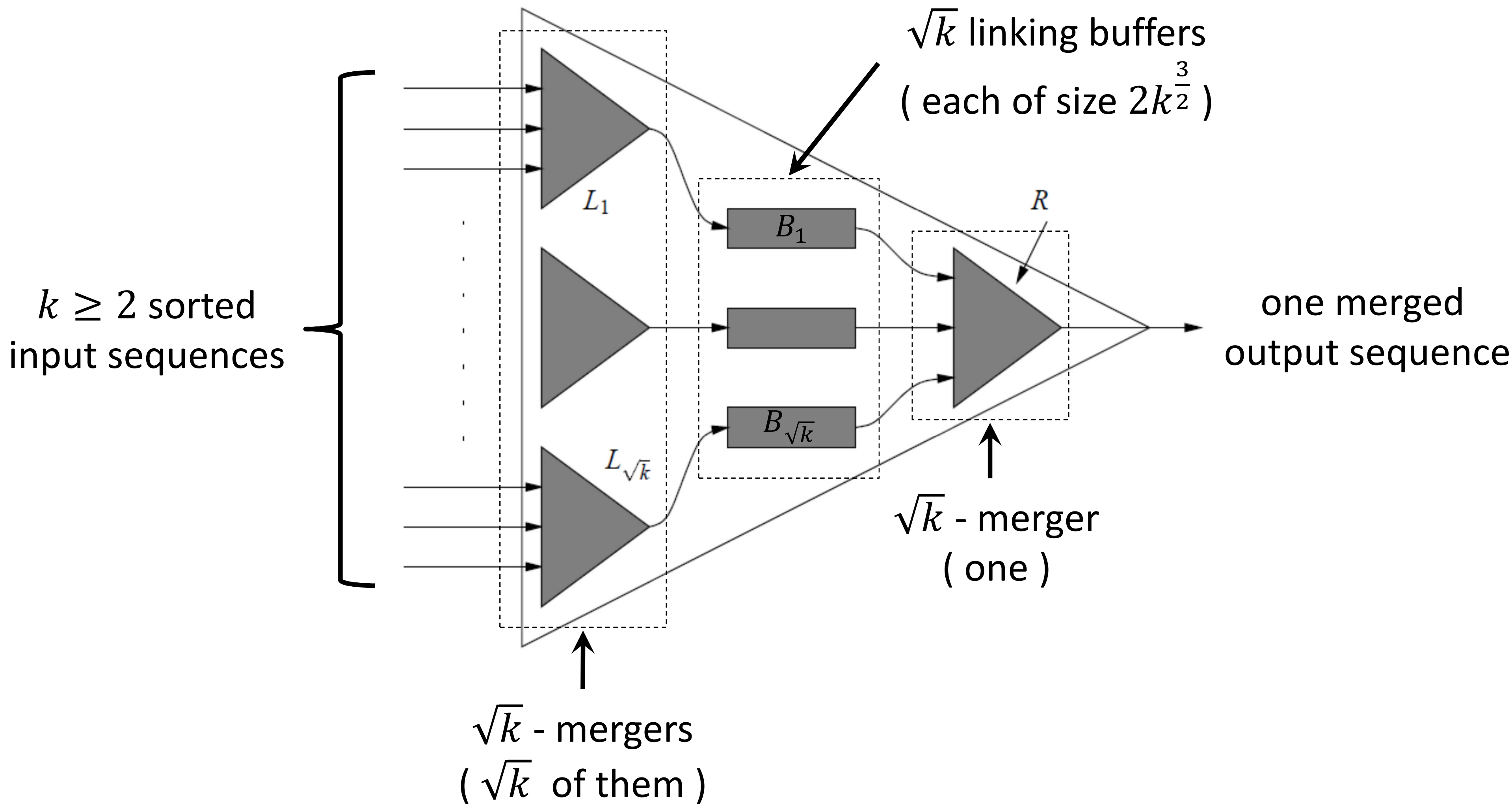
$$= \mathrm{O}\left(k \cdot \frac{n}{M} + \frac{n}{B}\log_k \frac{n}{M}\right)$$

Using $k = \frac{M}{B} - 1$, we get:

$$Q(n) = \mathrm{O}\left(\left(\frac{M}{B}-1\right)\frac{n}{M} + \frac{n}{B}\log_{\frac{M}{B}}\left(\frac{n}{M}\right)\right) = \mathrm{O}\left(\frac{n}{B}\log_{\frac{M}{B}}\left(\frac{n}{M}\right)\right)$$

Sorting
(Funnelsort)

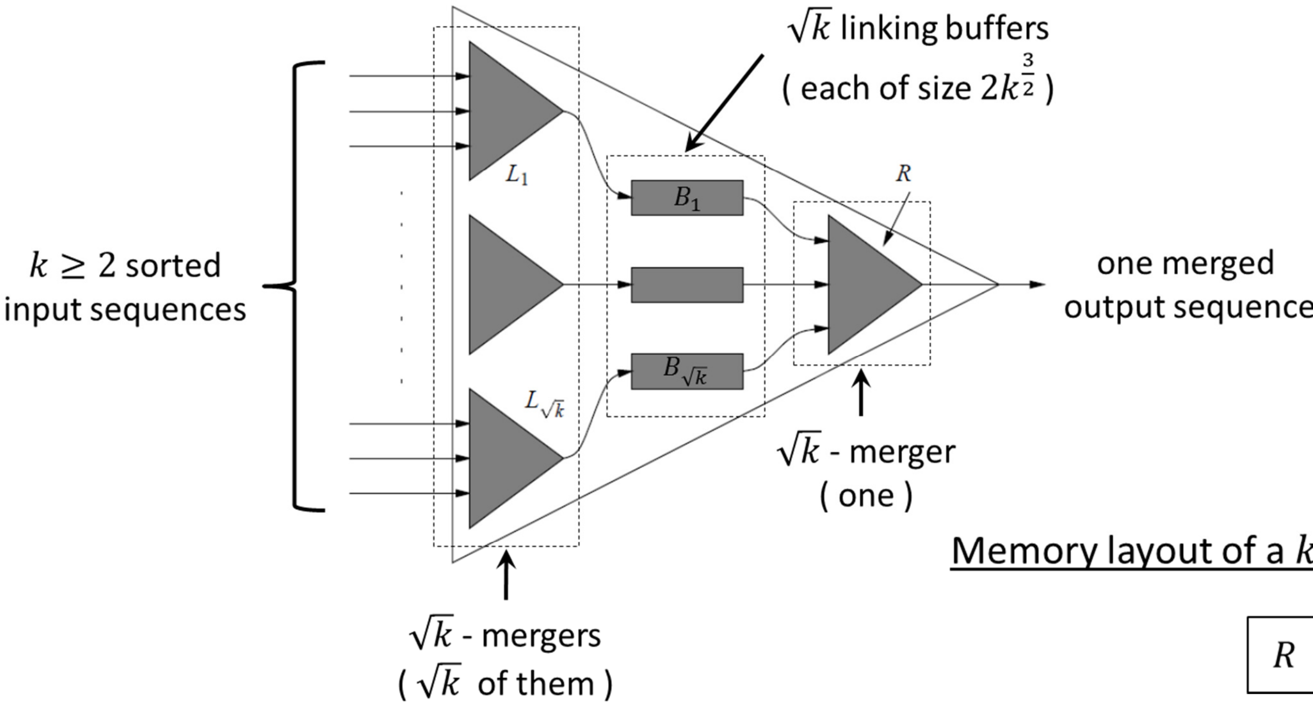
k-Merger (*k*-Funnel)



Memory layout of a k-merger:

R	L_1	B_1	L_2	B_2		$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

k-Merger (k-Funnel)



Memory layout of a k-merger:

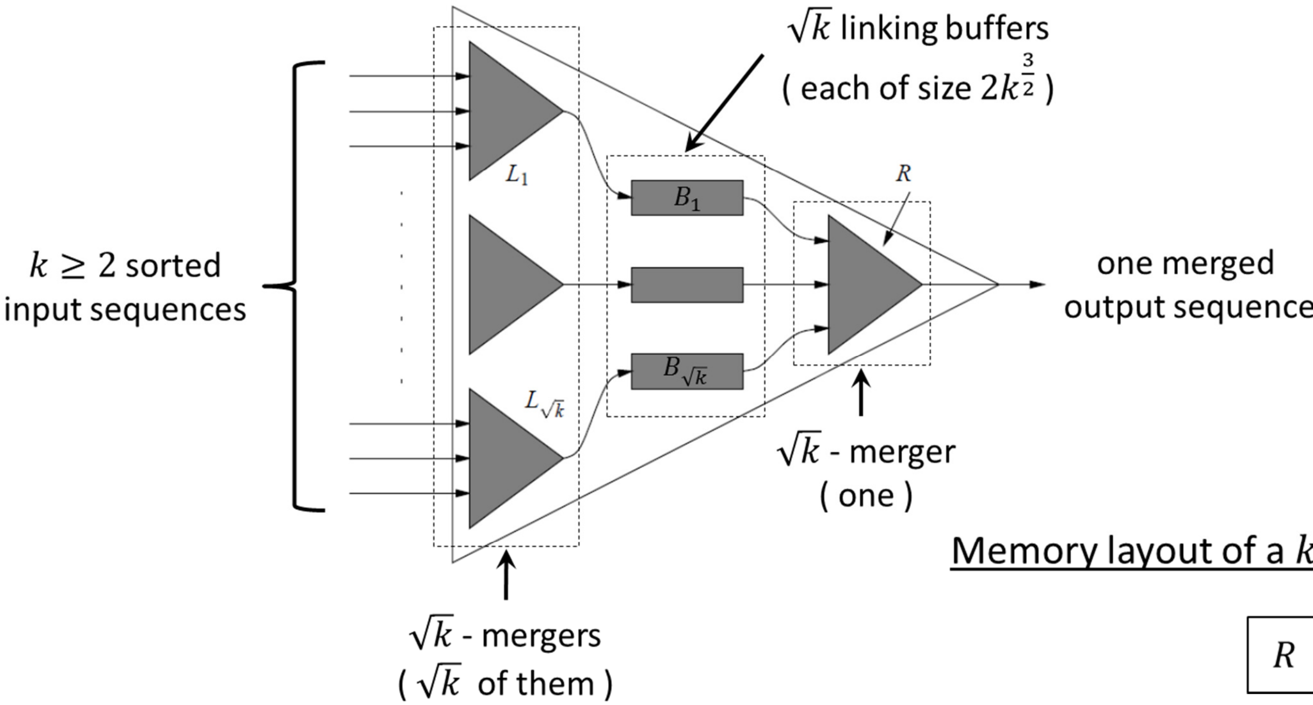
R	L_1	B_1	L_2	B_2		$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Space usage of a k-merger: $S(k) = \begin{cases} \Theta(1), & if\ k \leq 2, \\ (\sqrt{k}+1)S(\sqrt{k}) + \Theta(k^2), & otherwise. \end{cases}$

$$= \Theta(k^2)$$

A k-merger occupies $\Theta(k^2)$ contiguous locations.

k-Merger (*k*-Funnel)



Memory layout of a k-merger:

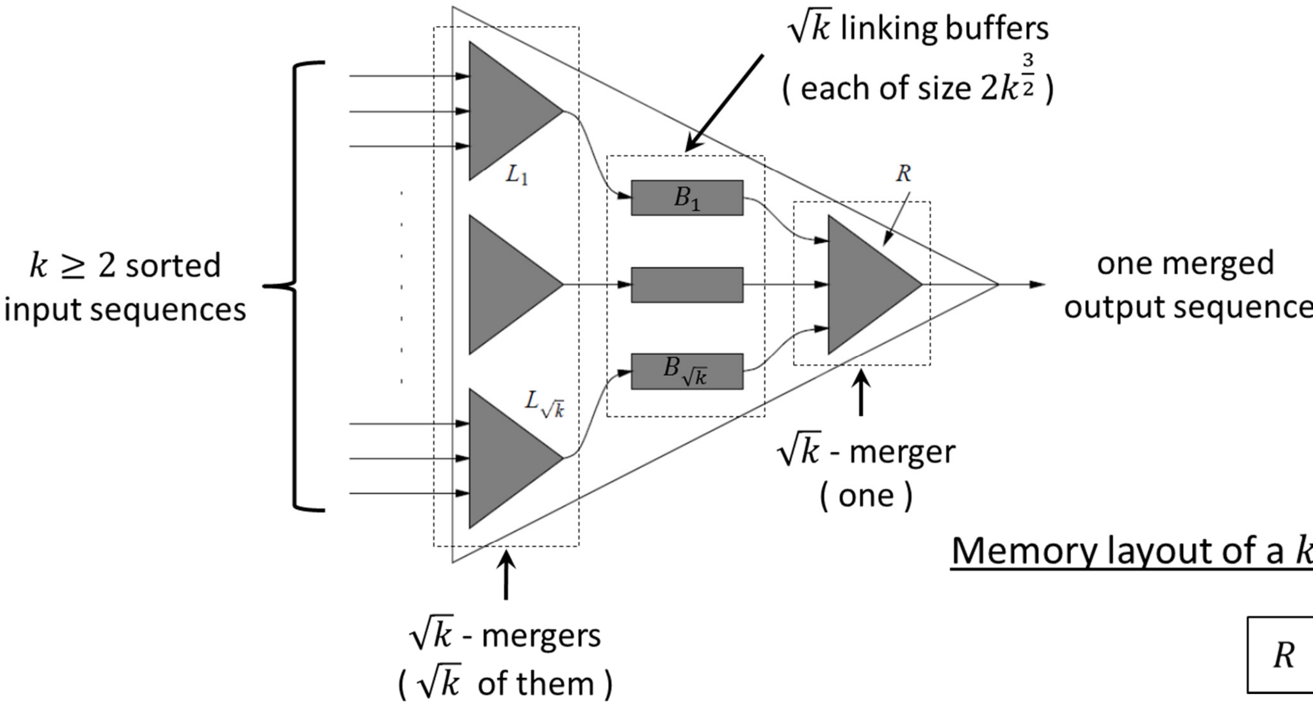
R	L_1	B_1	L_2	B_2		$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Each invocation of a k-merger

– produces a sorted sequence of length k^3

– incurs $\mathrm{O}\left(1 + k + \frac{k^3}{B} + \frac{k^3}{B}\log_M\left(\frac{k}{B}\right)\right)$ cache misses provided $M = \Omega(B^2)$

k-Merger (k-Funnel)



Memory layout of a k-merger:

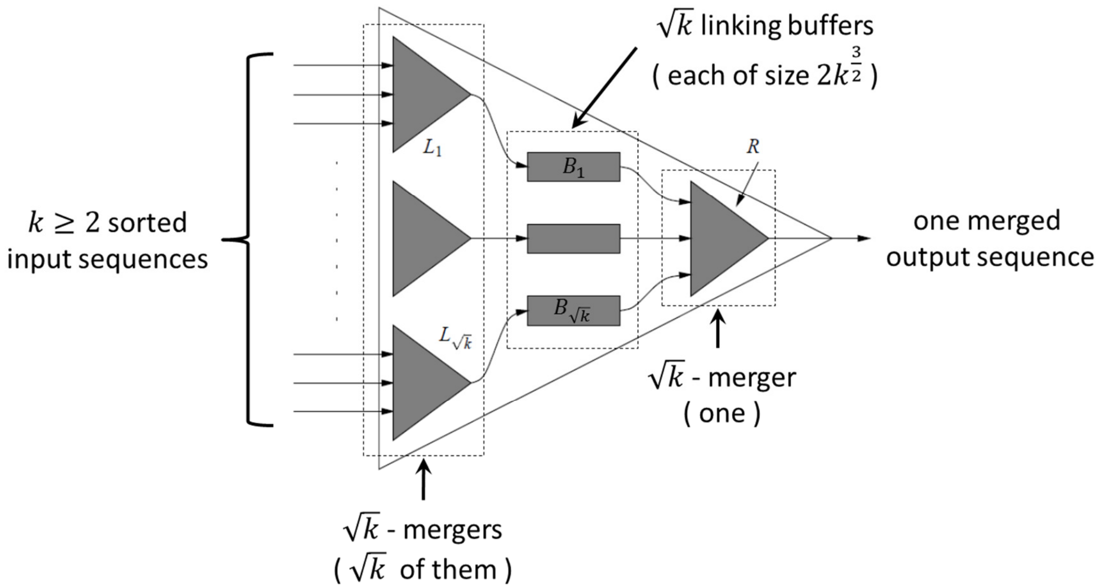
R	L_1	B_1	L_2	B_2		$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Cache-complexity:

$$Q'(k) = \begin{cases} \mathrm{O}\left(1 + k + \frac{k^3}{B}\right), & if\ k < \alpha\sqrt{M}, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right) Q'\left(\sqrt{k}\right) + \Theta(k^2), & otherwise. \end{cases}$$

$$= \mathrm{O}\left(\frac{k^3}{B} \log_M\left(\frac{k}{B}\right)\right), \qquad \text{provided } M = \Omega(B^2)$$

k-Merger (k-Funnel)



Memory layout of a k-merger:

R	L_1	B_1	L_2	B_2	...	$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Cache-complexity:

$$Q'(k) = \begin{cases} O\left(1 + k + \frac{k^3}{B}\right), & if\ k < \alpha\sqrt{M}, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right)Q'\left(\sqrt{k}\right) + \Theta(k^2), & otherwise. \end{cases}$$

$$= O\left(\frac{k^3}{B}\log_M\left(\frac{k}{B}\right)\right), \qquad \text{provided } M = \Omega(B^2)$$

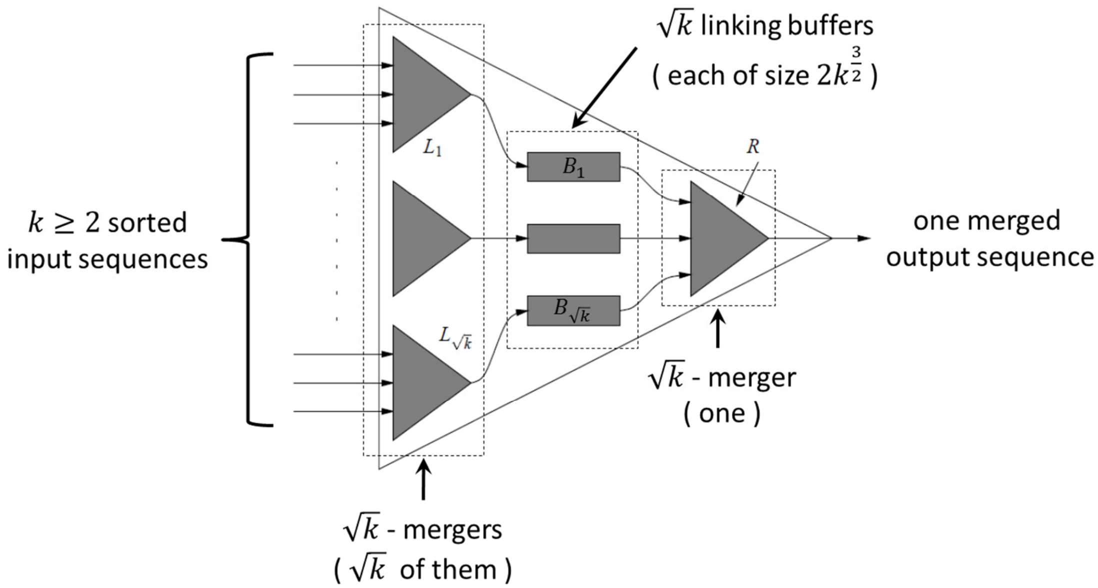
$$k < \alpha\sqrt{M}\text{: } Q'(k) = O\left(1 + k + \frac{k^3}{B}\right)$$

– Let r_i be #items extracted the i-th input queue. Then $\sum_{i=1}^{k} r_i = O(k^3)$.

– Since $k < \alpha\sqrt{M}$ and $M = \Omega(B^2)$, at least $\frac{M}{B} = \Omega(k)$ cache blocks are available for the input buffers.

– Hence, #cache-misses for accessing the input queues (assuming circular buffers) $= \sum_{i=1}^{k} O\left(1 + \frac{r_i}{B}\right) = O\left(k + \frac{k^3}{B}\right)$

k-Merger (k-Funnel)



Memory layout of a k-merger:

R	L_1	B_1	L_2	B_2		$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Cache-complexity:

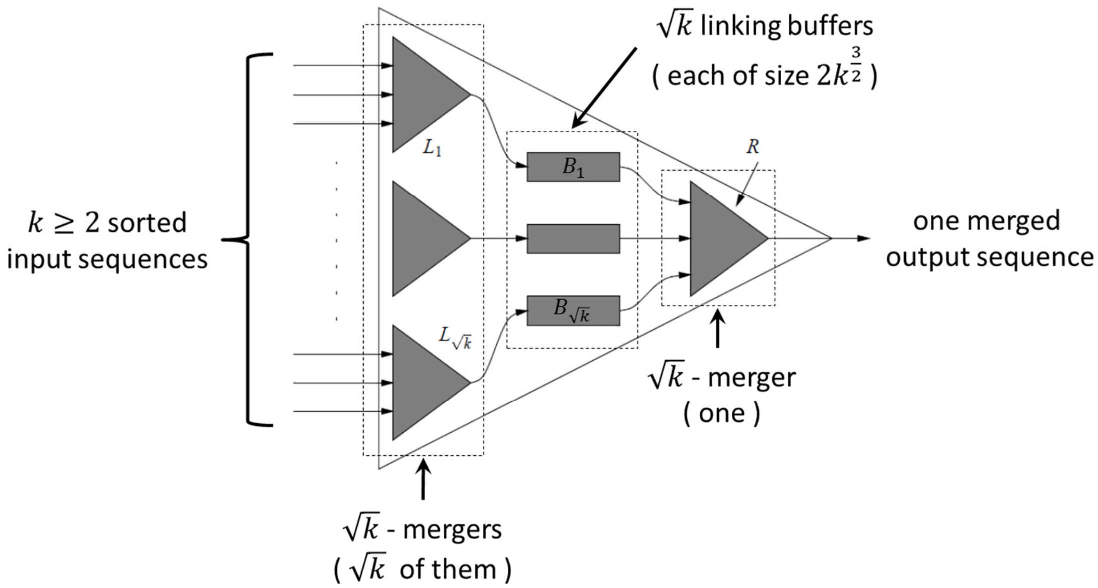
$$Q'(k) = \begin{cases} O\left(1 + k + \frac{k^3}{B}\right), & if\ k < \alpha\sqrt{M}, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right)Q'\left(\sqrt{k}\right) + \Theta\left(k^2\right), & otherwise. \end{cases}$$

$$= O\left(\frac{k^3}{B}\log_M\left(\frac{k}{B}\right)\right), \qquad \text{provided } M = \Omega\left(B^2\right)$$

$$k < \alpha\sqrt{M}\text{: } Q'(k) = O\left(1 + k + \frac{k^3}{B}\right)$$

- #cache-misses for accessing the input queues $= O\left(k + \frac{k^3}{B}\right)$
- #cache-misses for writing the output queue $= O\left(1 + \frac{k^3}{B}\right)$
- #cache-misses for touching the internal data structures $= O\left(1 + \frac{k^2}{B}\right)$
- Hence, total #cache-misses $= O\left(1 + k + \frac{k^3}{B}\right)$

k-Merger (k-Funnel)



Memory layout of a k-merger:

R	L_1	B_1	L_2	B_2		$L_{\sqrt{k}}$	$B_{\sqrt{k}}$

Cache-complexity:

$$Q'(k) = \begin{cases} \mathrm{O}\left(1 + k + \frac{k^3}{B}\right), & if\ k < \alpha\sqrt{M}, \\ \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right) Q'(\sqrt{k}) + \Theta(k^2), & otherwise. \end{cases}$$

$$= \mathrm{O}\left(\frac{k^3}{B} \log_M \left(\frac{k}{B}\right)\right), \qquad \text{provided } M = \Omega(B^2)$$

$$k \geq \alpha\sqrt{M}:\ Q'^{(k)} = \left(2k^{\frac{3}{2}} + 2\sqrt{k}\right) Q'(\sqrt{k}) + \Theta(k^2)$$

- Each call to R outputs $k^{\frac{3}{2}}$ items. So, #times merger R is called $= \frac{k^3}{k^{\frac{3}{2}}} = k^{\frac{3}{2}}$
- Each call to an L_i puts $k^{\frac{3}{2}}$ items into B_i. Since k^3 items are output, and the buffer space is $\sqrt{k} \times 2k^{\frac{3}{2}} = 2k^2$, #times the L_i's are called $\leq k^{\frac{3}{2}} + 2\sqrt{k}$
- Before each call to R, the merger must check each L_i for emptiness, and thus incurring $\mathrm{O}(\sqrt{k})$ cache-misses. So, #such cache-misses $= k^{\frac{3}{2}} \times \mathrm{O}(\sqrt{k}) = \mathrm{O}(k^2)$

Funnelsort

- Split the input sequence A of length n into $n^{\frac{1}{3}}$ contiguous subsequences $A_1, A_2, \ldots, A_{n^{\frac{1}{3}}}$ of length $n^{\frac{2}{3}}$ each
- Recursively sort each subsequence
- Merge the $n^{\frac{1}{3}}$ sorted subsequences using a $n^{\frac{1}{3}}$-merger

Cache-complexity:

$$Q(n) = \begin{cases} \mathrm{O}\left(1 + \frac{n}{B}\right), & if\ n \le M, \\ n^{\frac{1}{3}} Q\left(n^{\frac{2}{3}}\right) + Q'\left(n^{\frac{1}{3}}\right), & otherwise. \end{cases}$$

$$= \begin{cases} \mathrm{O}\left(1 + \frac{n}{B}\right), & if\ n \le M, \\ n^{\frac{1}{3}} Q\left(n^{\frac{2}{3}}\right) + \mathrm{O}\left(\frac{n}{B} \log_M \left(\frac{n}{B}\right)\right), & otherwise. \end{cases}$$

$$= \mathrm{O}\left(1 + \frac{n}{B} \log_M n\right)$$

Sorting
(Distribution Sort)

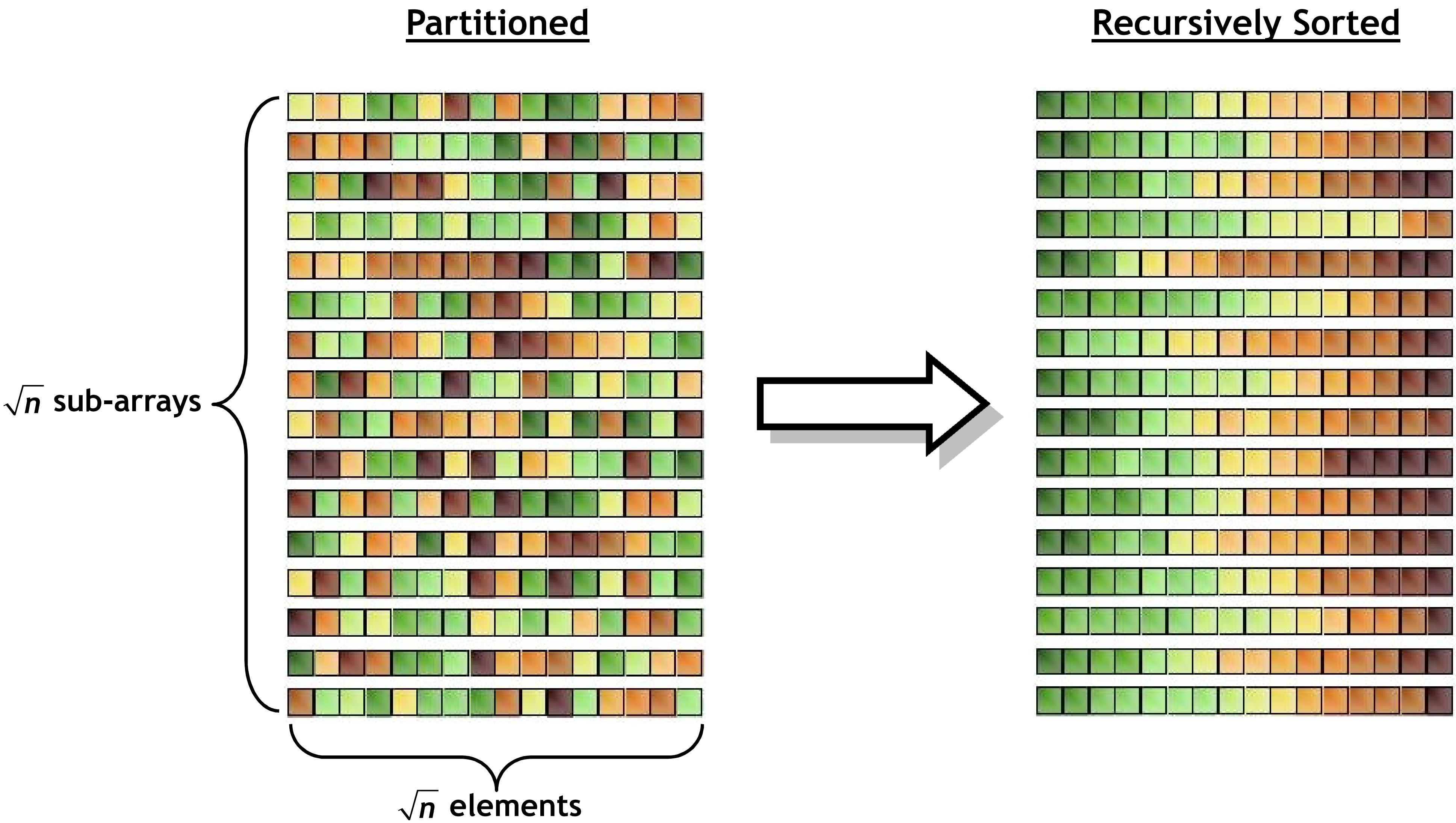
Cache-Oblivious Distribution Sort

Step 1: Partition, and recursively sort partitions.

Step 2: Distribute partitions into buckets.

Step 3: Recursively sort buckets.

Step 1: Partition & Recursively Sort Partitions



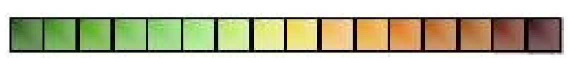
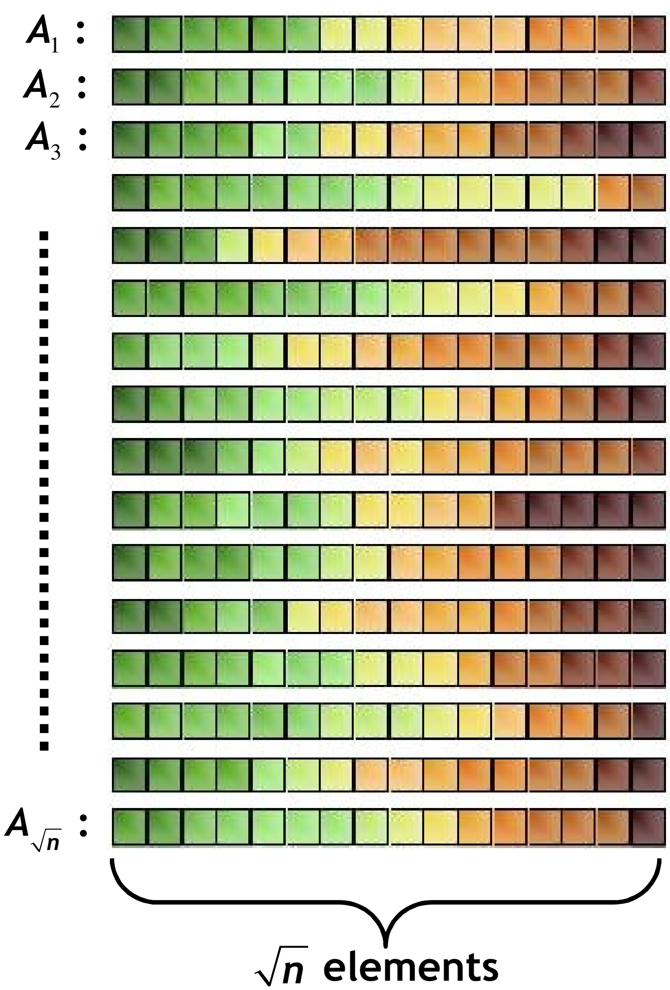
Order: 

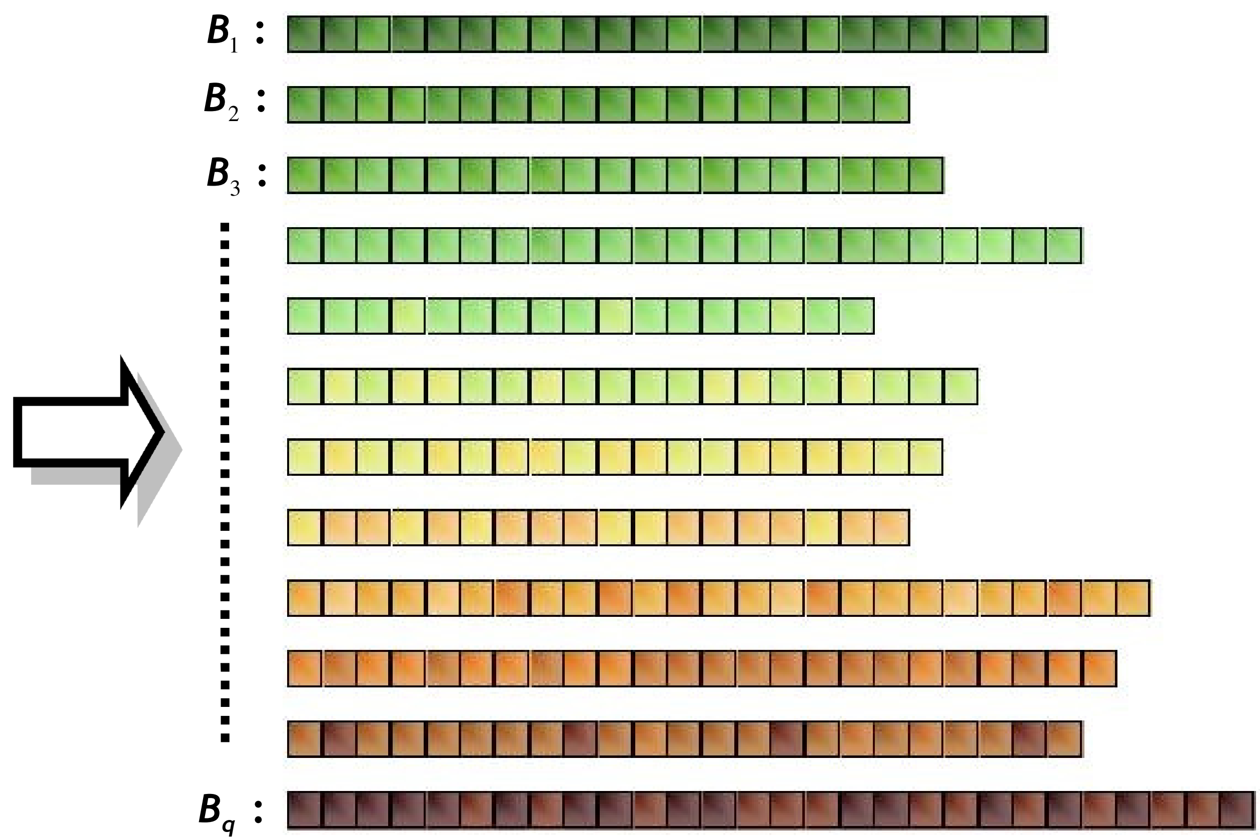
Figure Source: Adapted from figures drawn by Piyush Kumar (2003), FSU

Step 2: Distribute to Buckets

Recursively Sorted



Distributed to Buckets

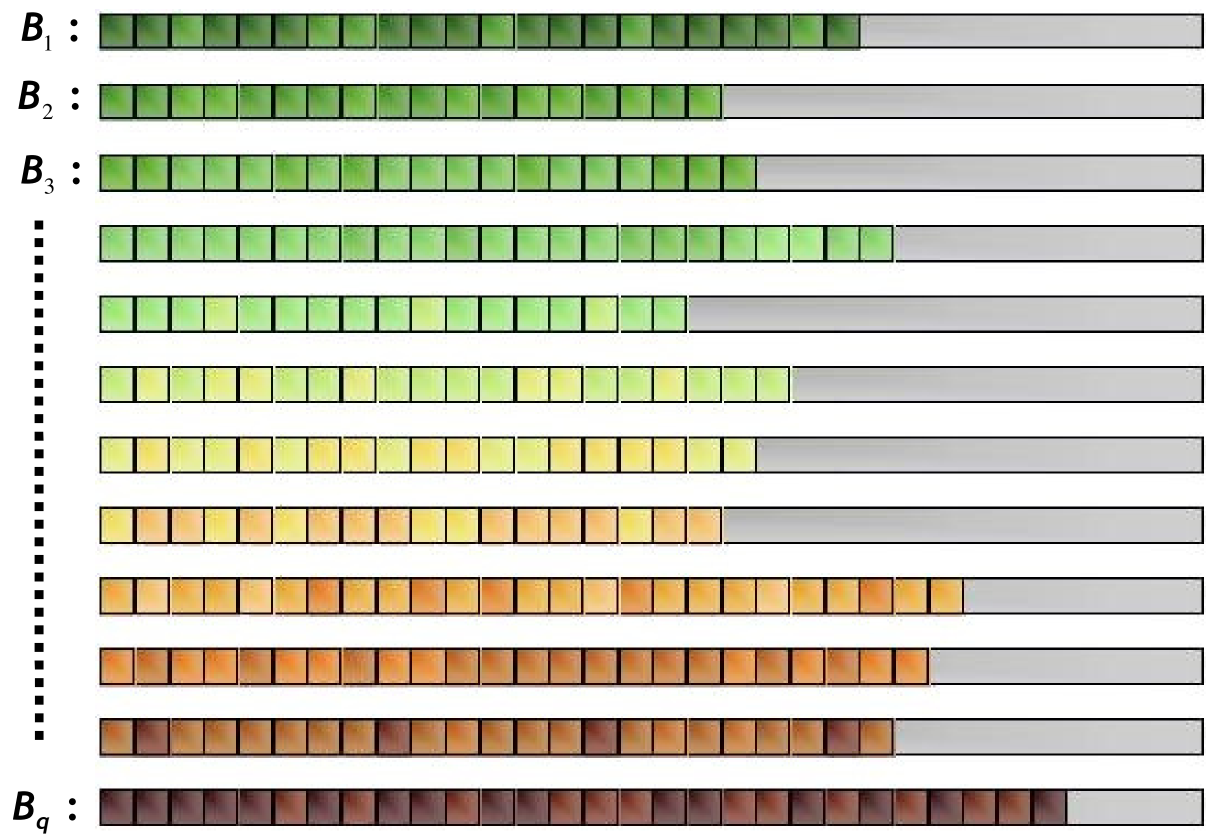


- ❑ Number of buckets, $q \leq \sqrt{n}$
- ❑ Number of elements in $B_i = n_i \leq 2\sqrt{n}$
- ❑ $\max\{x | x \in B_i\} \leq \min\{x | x \in B_{i+1}\}$

Figure Source: Adapted from figures drawn by Piyush Kumar (2003), FSU

Step 3: Recursively Sort Buckets

Recursively Sort Each Bucket



Done!

Figure Source: Adapted from figures drawn by Piyush Kumar (2003), FSU

Distribution Sort

Step 1: Partition, and recursively sort partitions.

$\boxed{\text{Step 2: Distribute partitions into buckets.}}$

Step 3: Recursively sort buckets.

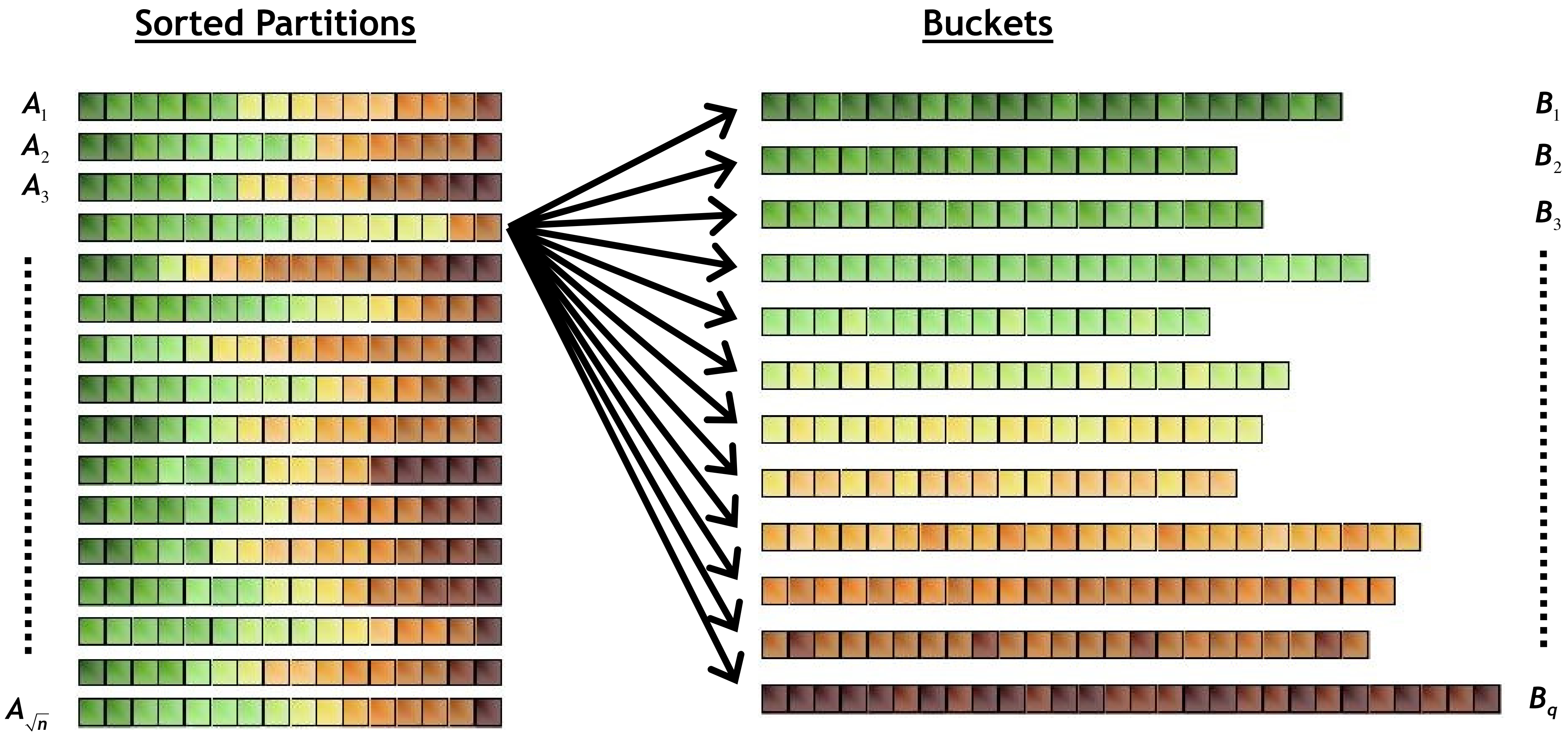
Distribution Sort

Step 1: Partition, and recursively sort partitions.

Step 2: Distribute partitions into buckets.

Step 3: Recursively sort buckets.

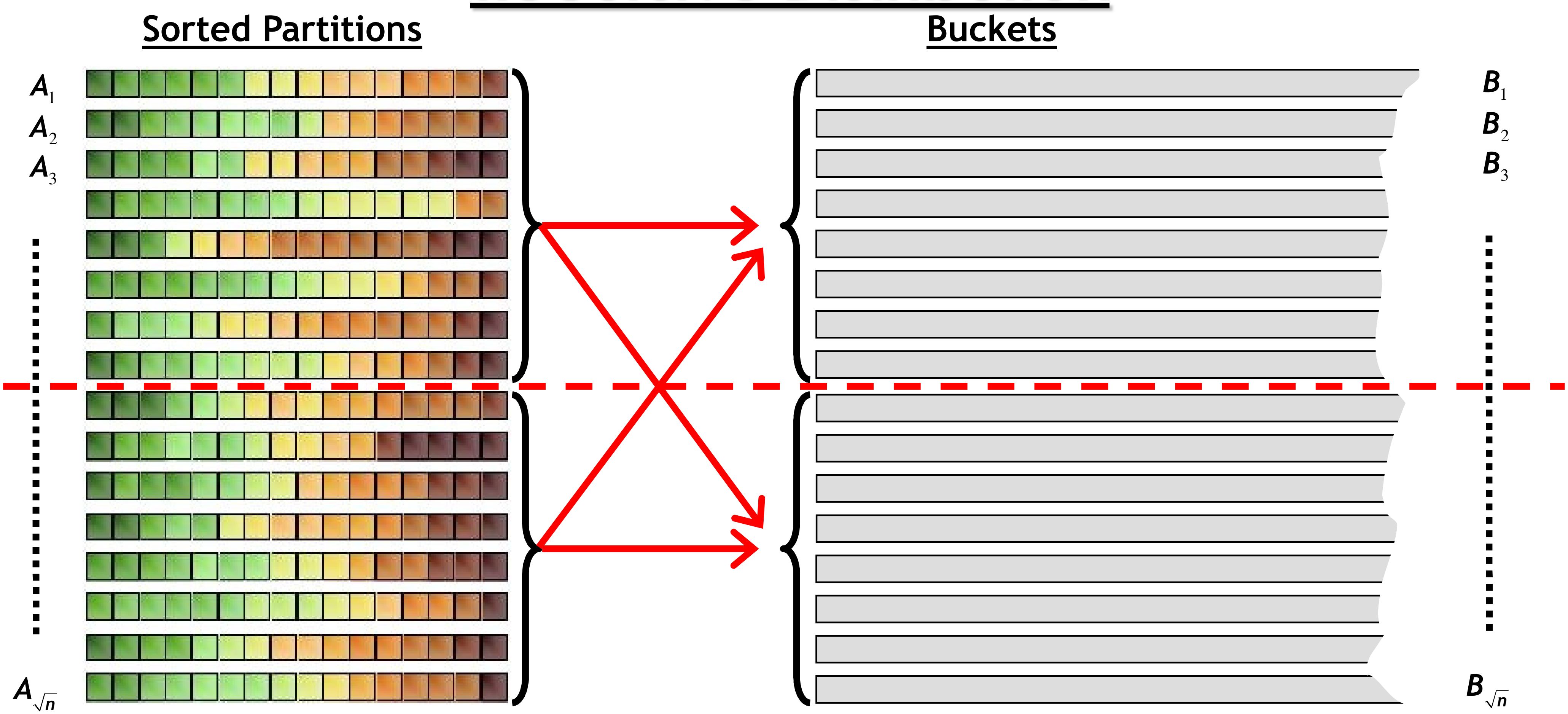
The Distribution Step



- We can take the partitions one by one, and distribute all elements of current partition to buckets
- Has very poor cache performance: upto $\Theta(\sqrt{n} \times \sqrt{n}) = \Theta(n)$ cache-misses

Figure Source: Adapted from figures drawn by Piyush Kumar (2003), FSU

Recursive Distribution



$[\, A_i, \ldots, A_{i+m-1} \,]$

⇓

$[\, B_j, \ldots, B_{j+m-1} \,]$

```
Distribute ( i, j, m )
    if  m = 1  then copy elements from A_i to B_j
    else
        Distribute (         i,         j,   m / 2 )
        Distribute ( i + m / 2,         j,   m / 2 )
        Distribute (         i, j + m / 2,   m / 2 )
        Distribute ( i + m / 2, j + m / 2,   m / 2 )
```

may need to split B_i to maintain $B_i \leq 2\sqrt{n}$

Recursive Distribution

```
Distribute ( i, j, m )
  1. if  m = 1  then copy elements from A_i to B_j
  2. else
  3.      Distribute (          i,            j,   m / 2 )
  4.      Distribute ( i + m / 2,             j,   m / 2 )
  5.      Distribute (          i,   j + m / 2,    m / 2 )
  6.      Distribute ( i + m / 2,    j + m / 2,    m / 2 )
```

ignore
the cost of splits
for the time being

Let $R(m, d)$ denote the cache misses incurred by *Distribute* (*i*, *j*, *m*) that copies d elements from *m* partitions to m buckets. Then

$$R(m,d) = \begin{cases} \mathrm{O}\left(B + \dfrac{d}{B}\right), & if\ n \le \alpha B, \\ \displaystyle\sum_{i=1}^{4} R\left(\frac{m}{2}, d_i\right), & otherwise, where\ d = \displaystyle\sum_{i=1}^{4} d_i\,. \end{cases}$$

$$= \mathrm{O}\left(B + \frac{m^2}{B} + \frac{d}{B}\right)$$

$$R(\sqrt{n}, n) = \mathrm{O}\left(\frac{n}{B}\right)$$

Recursive Distribution

Distribute (i, j, m)

1. *if* $m = 1$ *then* $\boxed{\textit{copy elements from } A_i \textit{ to } B_j}$
2. *else*
3. *Distribute* (i, j, $m / 2$)
4. *Distribute* ($i + m / 2$, j, $m / 2$)
5. *Distribute* (i, $j + m / 2$, $m / 2$)
6. *Distribute* ($i + m / 2$, $j + m / 2$, $m / 2$)

ignore

the cost of splits

for the time being

Recursive Distribution

```
Distribute ( i, j, m )
  1. if  m = 1  then copy elements from A_i to B_j
  2. else
  3.     Distribute (         i,          j,   m / 2 )
  4.     Distribute ( i + m / 2,          j,   m / 2 )
  5.     Distribute (         i,  j + m / 2,   m / 2 )
  6.     Distribute ( i + m / 2,  j + m / 2,   m / 2 )
```

#cache-misses incurred by all splits

$$= \sqrt{n} \times \mathrm{O}\left(\frac{\sqrt{n}}{B}\right)$$

$$= \mathrm{O}\left(\frac{n}{B}\right)$$

Cache-complexity of *Distribute*($1,1,\sqrt{n}$) is $= R(\sqrt{n}, n) + \mathrm{O}\left(\frac{n}{B}\right) = \mathrm{O}\left(\frac{n}{B}\right)$

Cache-Complexity of Distribution Sort

Step 1: Partition into $\sqrt{n}$ sub-arrays containing $\sqrt{n}$ elements each and sort the sub-arrays recursively.

Step 2: Distribute sub-arrays into buckets $B_1, B_2, \ldots, B_q$.

Step 3: Recursively sort the buckets.

Cache-complexity of Distribution Sort:

$$Q(n) = \begin{cases} O\left(1 + \frac{n}{B}\right), & if\ n \le \alpha' M, \\ \sqrt{n}Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i) + O\left(1 + \frac{n}{B}\right), & otherwise. \end{cases}$$

$$= O\left(1 + \frac{n}{B}\log_M n\right), \qquad when\ M = \Omega(B^2)$$