**CS5234: Combinatorial and Graph Algorithms** Lecture 7

# Caching

*Lecturer: Seth Gilbert* *September 29, 2016*

**Abstract**

Today, we will focus on caching. Imagine you have a 10TB dataset that you want to analyze. How do you go about it? The dataset is so large that it likely has to be stored on disk (i.e., it is too big for main memory). We will be considering data structures that can operate efficiently on these types of very large data sets stored in external memory. A key aspect to such data structures is that they use cache efficiently, minimizing the expensive transfers of data between disk and main memory.

## 1 Overview

One of the biggest factors in the practical performance of your algorithm is how efficiently it uses the cache. Imagine the following simple experiment, where you compare the performance of the following two procedures:

1. Store a large collection of data consisting of $n$ items in a linked list. Scan the entire list from beginning to end. Performance: $O(n)$.
2. Store a large collection of data consisting of $n$ items in an array. Scan the array from beginning to end. Performance: $O(n)$.

Both procedures are implementing the same "algorithm," scanning a large collection of data sequentially. However, using an array will be significantly faster than using a linked list. This performance difference is due to caching: the array is stored sequentially in memory, and hence when you can through it, you get good cache locality. The linked list is not stored sequentially in memory, since each node in the list is allocated separately. As a result, you get poor caching performance.

Consider a second experiment. Imagine comparing the performance of the following two procedures:

1. Store a large collection of data consisting of $n$ items in a balanced binary search tree (e.g., a red-black tree). Perform a search for an item in the tree. Performance: $O(\log n)$.
2. Store a large collection of data consisting of $n$ items in sorted order in an array. Performance a binary search for an item in the array. Performance: $O(\log n)$.

Again, both procedures appear to have the same performance. Each takes time $O(\log n)$. As before, the array is stored sequentially in memory and hence we expect good cachine performance. And the tree is not stored sequentially in memory, and hence we expect poor caching performance. And yet in this case, we see a very small difference in performance: the binary tree works almost as well as the array. In this case, caching does not make much of a difference.

Our goal in the next few weeks is to be able to understand these types of performance differences. We will explore how to analyze the performance of algorithms with respect to caching behavior. And we will look at algorithms specially designed to achieve good caching performance.

## 2 Real Machines

Caches on real machines today are remarkably complicated. As an example, consider the Intel Haswell architecture. It can have from 2 to 18 cores, and it has the following caches:

- An L1 cache of 64KB per core, with 32KB allocated for instructions and 32KB allocated for data. The cache is 8-way set associative. Each cache line has 64 bytes. Each cache access takes approximately four clock cycles.
- An L2 cache of 256KB per core. Each cache line has 64 bytes. Each cache access takes approximatley 10 clock cycles.
- An L3 cache of between 2 and 40MB, shared among the cores. Each cache line has 64 bytes. Each cache access takes between 40 and 74 clock cycles (depending on the sharing status).
- Optionally, an L4 cache of 128MB.
- A translation lookaside buffer (TLB).
- A decoded micro-operation cache (approximately 6KB).
- Main memory has a page size of 16KB, and takes 200-350 clock cycles.
- Disk, which takes approximately 20,000,000 clock cycles (or more). (For an SSD, that might only be 20,000 clock cycles.)

(All the latency numbers above are highly approximate, but hopefully give a sense of an order-of-magntitude.)

One other issue has to do with how cores and processes coordinate. Notice tha the L1 and L2 caches are per core: each gets its own memory. The L3 cache, by contrast, is shared by all the cores on the same CPU, i.e., in the same socket. And if cores on different sockets wants to coordinate, that requires accessing main memory.

To illustrate the importance of caching, let's work out a simple example. Imagine that 90% of memory requests are serviced by the L1 cache, at a cost of 4 clock cycle each. (In reality, the hit rate is often higher than that.) And imagine that another 8% of the requests are serviced by the L2 cache in 10 clock cycles. Finally, imagine that the remaining 2% of of requests go to main memory which takes 300 clock cycles. (For our example, we are rounding a bit, and ignoring the L3 and L4 caches.) In that case, during the actual execution of your system, you will spend approximately:

- 35% of your time waiting for the L1 cache to respond.
- 8% of your time waiting for the L2 cache to respond.
- 57% of your time waiting for the main memory to respond.

Notice that despite a seemingly very good cache hit rate (i.e., 98% of the time the data is in one of the caches), we still spend more than half of our time waiting for data from main memory.

In reality, cache hits rates are even higher than this. For many applications, over 99% of the memory accesses will be in cache. However, these types of numbers clearly demonstrate the importance of using a cache efficiently.

When data sets are sufficiently large, however, they cannot be stored in main memory. (Try, for example, to write a Java program to access a 10GB file.) Imagine you are trying to process a data file that consists of many terabytes of information. There is no hope to store all this information in memory.

Instead, most of the data will reside on disk, only loaded into memory as needed. In essence, the main memory is acting as a cache for the disk. And a disk access can be 10,000 times slower than main memory! Remember, a disk consists of physical moving parts: a head that must be moved to the correct location, and a physical platter that needs to be spun to the right location. Disks are very slow. (SSDs are better, of course, though still significantly slower than memory. Unfortunately for large enough data sets, it is hard to find SSDs that are big enough.)

So depending on your application, you may see different memory bottlenecks. For a sufficiently small application, all the working information will remain in cache. For a somewhat larger application, the bottleneck will be the memory transfers between the main memory and L1 and L2 caches. For a very large application, the bottleneck will be the cost of transfering data from the disk to the main memory.

# 3 Modeling a Cache

If we want to develop and analyze algorithms in a manner that takes caching behaivor into account, we need a way to model the performance of caches. (For example, if we try to analyze algorithms in the normal manner, there is no easy way to explain why a linear scan is so strongly impacted by caching performance while a binary search is not.)

Unfortunately, if we try to model the real complexity of existing caching hierarchies, we will quickly get bogged down in messy details. If, in order to analyze an algorithm, we need to work through a complicated multi-level caching model, each with different latences (and different sharing policies), each (possibly) with different cache replacement policies, etc., then we will never be able to use the model to design real algorithms.

Our goal, then, is to find a model for caching behavior that is:

- as simple as possible: we want a tractable model that we can work with;
- sufficiently realistic: we want a model that gives real insight into how our algorithms will work on real machines.

These two goals are always in conflict with each other, and lead to many vigorous and loud debates!

The classic—and perhaps simplest—model is the *External Memory Model* introduced by Aggarwal and Viter in 1988. This model focuses on the problem of accessing a disk, focusing on the cost of moving data between main memory and disk. This model simplifies the situation to its basic components:

- There is a memory (i.e., your cache) of size $M$.
- There is a much larger disk (whose size is often unspecified).
- Both the memory and the disk are divided into blocks of size $B$. Data is moved in blocks.
- There is a cost of $1$ for moving a block of data from disk to main memory.

Whenever you access a memory location, first we look for the location in the cache. If it is there, then we assume the cost of accessing it is zero, i.e., we can access it freely. On the other hand, if it is not there, then it must be retrieved from disk. To retrieve it from disk, we locate the block containing that location on disk, and copy that block into the main memory at a cost of $1$. If the memory is already full, then we first need to evict something from the memory.

Notice this model ignores many of the real-world aspects. It ignores the L1 and L2 caches altogether. It ignores the cost of accessing something in main memory, as in practice it is not *free* to access something in main memory. Etc.

In practice, however, recall that the cost of a disk access is at least 10,000 times as expensive as accessing main memory. Hence the cost of accessing disk will far swamp the costs that have been ignored.

What happens if you want to use the External Memory Model to examine the cost of transferring data from main memory to the L2 cache, where the L2 cache plays the role of the memory in the model, and the main memory plays the role of the disk? First, you might find it to be less accurate: the cost of accessing memory versus the cost of accessing your cache is only a factor of 30. Second, the value of $B$ is smaller. While a disk page may be 16KB, a cache line might consist of only 64 bytes. Despite these limitations, the external memory model does turn out to be useful for understanding this type of caching as well.

# 4 The Effect of Caching

Before looking at specific algorithms more closely, let's go over some of the known results in the External Memory Model.

**Scanning data.** As we have already seen, scanning an array is much faster than scanning a linked list. This can be made precise using the External Memory Model. To scan a linked list of size $n$ has cost $\Theta(n)$: each node in the linked list may be stored in a separate block, and hence each access incurs a cache miss resulting in a memory transfer.

By contrast, scanning an array of size $n$ has cost $\Theta(n/B)$. Since the data in the array is stored consecutively in memory, all the data in the array is stored on at most $n/B + 1$ blocks. Hence the total cost of scanning the entire array will be at most $n/B + 1$ memory transfers.

**Searching.** Recall our example of searching a binary tree for a value. Much like the linked list, each node of the binary tree may be stored in a separate block. Thus, the cost of searching the binary search tree is going to be $\Theta(\log n)$.

Now consider, instead, doing a binary search in an array. Notice that most of the times when the array is accessed, the queries are not near to each other. For example, during a binary search we begin by querying location $A[n/2]$ in the array, after which we might query $A[n/4]$ or $A[3n/4]$. These queries cannot take advantage of cache locality: each query is going to load a separate block.

Eventually, however, the binary search gets close to the target, and hence the queries are near to the same point in the array. At this point, we can take advantage of cache locality. Specifically, consider what happens once the range being search by the binary search is of size at most $B$. At this point, the range contains at most two blocks (depending on alignment), and so the rest of the search will take at most 1 memory transfer.

So the total cost of a binary search in an array of size $n$ in the external memory model is $O(\log(n/B))$. You can imagine that the binary search is really running on an array of size $n/B$, where each element is of size $B$, and so the search will take at most time $O(\log(n/B))$.

We conclude that there is a difference between a binary tree and a binary search: the binary search will be approximately $\Theta(\log B)$ steps faster. For a disk with large page size, this may well be significant. For an array stored in memory (where the cost is transferring data to the L2 cache), the block size $B$ is much smaller and this performance improvement will be harder to see.

In fact, we can do better: we can store data so that we can perform searches in $O(\log_B(n))$ times. Notice in this case we are seeing an improvement by a multiplicative factor of $\log B$, which can be very significant.

This performance can be achieved by a B-tree. A B-tree is a search tree in which every operation has cost $O(\log_B n)$.

**Sorting.** For sorting data in memory, we would typically use QuickSort or MergeSort (or HeapSort) which run in $O(n \log n)$ time. These sorting algorithms have reasonable cache-performance. For example, QuickSort is an in-place, divide-and-conquer algorithm and as such it does take some advantage of locality. (I will leave it as an exercise for later to think about the performance of QuickSort in the external memory model.)

Another way of sorting would be to build a binary search tree, inserting items one at a time. Assuming we use a B-tree, each insert operation has cost $O(\log_B n)$, and scanning the leaves of the tree has cost $O(n/B)$. So the total cost of sorting using a B-tree is $O(n \log_B n)$.

In fact, we can do a lot better: we can sort an array of $n$ items in $O(\frac{n}{B} \log_{(M/B)}(\frac{n}{B}))$ time. And this bound is tight, for comparison-based algorithms: we can show a lower-bound using the same decision-tree style technique that we would traditionally use to show that we need $\Omega(n \log n)$ time in the internal memory model.

There are three well-known algorithms that achieve this bound: External MergeSort, External QuickSort, and Buffer-tree Sort.

Recall, first, how MergeSort works. Imagine you are implementing a bottom-up version of MergeSort where you perform $\log n$ passes over the array: in the first pass, you merge neighbors, in the second pass you merge groups of 4, in the third pass you merge groups of eight, and so on. In each pass, you repeatedly run a 2-way merging protocol to merge two groups of size $k$ in $O(k)$ time. Since each pass takes $O(n)$ time and there are $\log n$ passes, the total running time is $O(n \log n)$.

External MergeSort is similar, except that in each pass, its performs an $(M/B)$-way merge on the $(n/B)$ blocks instead of a 2-way merge on the elements. That is, in the first pass, it merges $(M/B)$ groups of size $B$. In the second pass, it merges $(M/B)$ groups of size $(M/B)B$. In the third pass, it merges groups of size $(M/B)^2B$. Thus, it completes in $\log_{(M/B)}(n/B)$ passes. Each pass takes $n/B$ time. Thus we can a total running time of $O((n/B)\log_{(M/B)}(n/B))$.

**Priority Queues.** You can, of course, implement a priority queue using a B-tree, in which case each operation on the priority queue will have cost $O(\log_B(n))$. It turns out that you can do better. Using a "Buffer Heap," you can achieve a running time of $(1/B)\log(n/M)$, and using a Buffer Tree you can achieve $(1/B)\log_{(M/B)}(n/B)$. The basic idea is to add a large buffer of size $\sqrt{M}$ at each node in the tree. Instead of modifying the tree in its entirety, operations are simply added to the buffer at the root. Whenever a buffer is full, it is emptied, pushing the operations down to the children. Since the buffers contain so many items, we can amortize the cost: each time an item is moved down a level in the tree, it pays an amortized cost of $1/B$. By setting the branching degree of the tree to be $\sqrt{M}/B$, we get the desired performance. At the same time, we keep a cache of $\sqrt{M}$ elements with the minimum priority.

**Shortest Paths.** First, let us consider unweighted graphs. If your graph is stored as an adjacency list, then a naive BFS may be relatively slow: each edge may lead to a new location and hence involve loading a new block. Hence the running time may be $O(n+m)$. With some care, we can perform a BFS on a graph with $n$ nodes and $m$ edges in time $O(n + (m/B)\log_{M/B}(m/B))$ time. You will notice that the running time here is similar to the cost of sorting, and that is not an accident: here we using sorting to ensure that we visit nodes in a productive order. For dense graphs, this can be improved to a running time of $O(\sqrt{nm/B} + (m/B)\log_{M/B}(m/B))$.

For weighted graphs, we can use a variant of Dijkstra's algorithm, replacing the priority queue with the external memory version. This yields a running time of $O(n + (m/B)\log_2(m/M))$. (This should be compared against the regular version of Dijkstra's Algorithm which runs in time $O(m \log n)$ or $O(m + n \log n)$, depending on the implementation of the priority queue.)

For all-pairs-shortest-path, if the graph is unweighted, we can solve the problem in time $O((nm/B)\log_{M/B}(m/B))$. If the graph is weighted, the results are more complicated (but of course can be solved simply by running the single-source-shortest-path algorithm once for every node).

# 5 B-trees

Let's begin by reviewing perhaps the most important external memory data structure in use today: the B-tree. The B-tree is designed to store a set of keys so as to support insert, delete, and search operations in $O(\log_B(n))$ time.

A B-tree is really just a special case of an $(a, b)$-tree, where $a = B$ and $b = 2B$. An $(a, b)$-tree is a tree that satisfies the following rules:

- Every node (except the root) has at least $a$ children. The root has at least 2 children.
- Every node has at most $b$ children.
- Every leaf of the tree is at the same depth. (That is, every root-to-leaf path is the same length.)

Generally, when considering trees, there are two ways to store keys in the tree: the keys may be stored in the nodes themselves, or they may be stored at the leaves. For today, we will assume that all the keys are stored at the leaves.

The nodes in the tree simply contain pivots that guide the search to the correct leaf.

Occasionally, we might consider special variants of an $(a, b)$-tree where the leaves are special, i.e., the leaves consist of blocks of $B$ keys, even where $a$ and $b$ are must larger or smaller. (For example, you might consider a $(2, 4)$ tree with leaves of size $B$.) In general, if otherwise unspecified, we will assume that each leaf contains at least $a - 1$ and at most $b - 1$ keys. We will assume throughout that $b \geq 2a$.

Each node stores a collection of pivots $p_1, p_2, \ldots, p_k$. A node with $k$ pivots has $k + 1$ children, and so we know that $k$ will be at least $a - 1$ and at most $b - 1$. For each pivot $p_i$, there is a subtree (or leaf) $c_i$ that precedes it and a subtree $c_{i+1}$ that follows it. (That is, it is a complete tree.) The requirement for the tree is that:

- Every key in the subtree (or leaf) $c_i$ should be $\leq p_i$, and every key in subtree (or leaf) $c_{i+1}$ should be $> p_i$.

To search an $(a, b)$-tree, then, we simply walk down the tree using the pivots as guides:

- Begin a search for key $k$ at the root. Let node $v =$ the root.
- Repeat until we get to a leaf:
  - At node $v$, if $k \leq p_1$, then set $v =$ the root of subtree $c_1$.
  - Otherwise, let $\ell$ be the largest value such that $k > p_\ell$. (This implies that $k \leq p_{\ell+1}$, if $\ell + 1$ exists.) Then set $v =$ the root of subtree $c_{\ell+1}$.
- Search leaf $v$ for the element $k$.

The properties of the pivots ensure that this search will find the key $k$, if it is in the tree. (You might prove this more carefully by induction.)

Just based on these requirements, we can already bound the height of an $(a, b)$-tree:

**Claim 1** *An $(a, b)$-tree with $N$ keys has height at most $\log_a(N/a) + 1$.*

**Proof** Since each leaf has at most $a$ keys, there are at most $N/a$ leaves. Consider a child of the root $u$ and assume for the sake of contradiction that it has a path of length $> \log_a(N/a)$ to a leaf. Since every node has a branching factor of at least $a$ (except the root), all the children of $u$ have branching factor at least $a$, and hence the sub-tree rooted at $u$ has more than $a^{\log_a(N/a)} = N/a$ children. Since this is impossible, we assume that every path from $u$ to a leaf is of length at most $\log_a(N/a)$, and the claim follows. □

Now, let us consider how to insert a key $k$.

- Search for the appropriate leaf in the tree where key $k$ belongs.
- Insert $k$ into the leaf.
- If the leaf has more than $b - 1$ keys, then split the leaf, creating two new leaves $x$ and $y$: each should contain about half the keys, and all the keys in $x$ should be less than all the keys in $y$. Insert the largest element of the leaf $x$ into the parent as a new pivot $p_i$ and set $y$ to be the subtree $c_{i+1}$ of the parent.
- If the parent now has more than $b$ children, then split the parent and continue recursing up the tree. If you need to split the root, then create a new root node with the two nodes created from the split of the old root as the children.

When deleting a key from the tree, proceed similarly, merging nodes to maintain the invariants:

- Search for the appropriate leaf $u$ in the tree where key $k$ belongs.
- Delete $k$ from the leaf $u$.
- If the leaf $u$ has fewer than $a-1$ keys, then let $v$ be a sibling of $u$. There are now two cases:
  - Case 1: Node $u$ and node $v$, together, have $> b-1$ keys. In this case, divide the keys evenly between nodes $u$ and $v$. Each of $a$ and $b$ end up with at least $(b-1)/2 \geq a-1$ keys. At this point, we are done.
  - Case 2: Node $u$ and node $v$, together, have $\leq b-1$ keys. In this case, we merge nodes $u$ and $v$ into one new node $u'$. This new node clearly has at least $a-1$ keys (since $v$ had at least $a-1$ keys) and at most $b-1$ keys. We then delete the pivot separating the subtrees for $u$ and $v$ from the parent of $u$ and $v$. If this causes the parent to have fewer than $a-1$ pivots, then we continue recursing up the tree. If we reach the root and after the deletion there are no pivots left, then we delete the root node.

Finally, we consider the cost of these operations. For this purpose, we set $a = B$ and $b = 2B$. Notice that all the pivots for a node are stored in one or two blocks, and all the data for a node is stored in at most 4 blocks (including the pointers to the child subtrees). So the cost of accessing the data at a node is $O(1)$. The tree has height $O(\log_B(N/B))$, as previously discussed, and so every time we search the tree from the root to the leaf, it costs $O(\log_B(N/B)$. Similarly, the splitting and merging of the nodes during an insert and delete involves accessing at most one or two nodes at every level of the tree per operation and hence also has cost $O(\log_B(N/B))$.

In fact, if you set $a = B$ and $b = 4B$, you can decrease the amortized cost of the splitting and merging. You can ensure that after each split or merge, a node has at least $2B$ children and at most $3B$ children. Thus, before the next split or merge operation, there will be at least $B$ more operations. You can amortize the cost of that splitting and merge against these operations.

One case where this may be important is if you want to maintain parent pointers. Notice that if you have parent pointers, then when you split or merge a node, you have to update $\Theta(B)$ children with a new parent pointer. Thus, each split or join, instead of costing $O(1)$, instead costs $O(B)$. However, using the amortization strategy above, we can conclude that the *amortized* cost of each split or merge is still $O(1)$.

**Discussion.** On the one hand, for external memory (e.g., disk), B-trees work shockingly well. In this case, $B$ tends to be very large and so the tree ends up quite shallow. Even more importantly, if you access the tree a lot, the root of the tree and the top couple levels are likely to remain in the cache at all times. In total, then, even for very large data sets, you often only suffer $< 3$ cache misses. (With a page size of 16KB, that gets you more than 1000TB of data if the root is cached.)

On the other hand, the key challenge with using a B-tree is choosing the right value for $B$. Different hardware has different sized caches, and choosing the optimal value may be difficult.

Also, recall that real hardware has an entire hierarchy of caches. So far, you have just optimized for one level of that hierarchy, and reduced that cost to about 3 cache misses. At that point, the cost of accessing data in main memory may become your bottleneck—remember, each node you access has 16KB of data, which you are assuming you can search for free!

The classic solution here is to nest another B-tree inside the first B-tree, choosing a different value of $B$, i.e., one that matches the block size of the next level of the hierarchy. And you might continue again, at each step optimizing for the next cache.

An alternative, which we will talk about later, is a cache-oblivious algorithm, i.e., one that does not rely on precise knowledge of $B$ and $M$—and yet still yields good performance.

# 6 Buffer Trees

A B-tree guarantees that each operation has cost $O(\log_B n)$, i.e., you save a factor of $\log B$ over the cost of a regular binary search tree. I want to talk about another variant today, the Buffer Tree. There are three different motivations:

- A Buffer Tree prioritizes insertions over searches. This is often call a "write-optimized" data structure. In many applications, we believe that search operations are more common, and many data structures are optimized for searches. Here we see the opposite. There are actually an interesting class of problems where updates are more common than reads, and here we are optimizing for that case. (For example, think of a data structure designed to maintain a log. A log is constantly being updated with new data, and is only analyzed rarely.)
- Even if searches are more common, we may want to trade-off the cost of searches and inserts. What if I can make inserts a lot faster, while only slowing down searches a little bit? Depending on the ratio of searches and inserts, this may still yield better performance.
- For some uses of a tree, you do no searches. For example, imagine using a tree to perform a sort. You insert all the items, and then traverse the entire tree. If we can do fast inserts, then we can do faster sorting. Hence a Buffer Tree is another way of designing a fast sorting algorithm. We will also see that the same holds for Dijkstra's Algorithm, e.g., for finding shortest paths: we do not need to search to implement a priority queue.

Also, a buffer tree is a great example of being lazy. By deferring work until later, we get a more efficient algorithm. This general principle is often useful!

## Basic Description

A Buffer Tree is a $(2, 3)$-tree where every non-leaf node has a buffer of size $2B$. All the keys live at the leaves, and every node has degree 2 or 3. Each leaf has a block of between $B/2$ and $2B$ keys. We will maintain the invariant that each buffer has at most $B$ items in it at the end of each operation.

The main idea is that instead of completely performing an operation that modifies the tree, instead we simply add that operation to the buffer at the root. As operations on the tree progress, the operation will slowly work its way down the tree, hopping from buffer to buffer, until it reaches the proper leaf. Only then will the operation actually be performed.

**Inserting and deleting.** To insert or delete a key $x$, we simply add the operation $(insert, x)$ or $(delete, x)$ to the buffer at the root of the tree. (We do not modify the tree itself in any way at this point.) Check whether there are any other operations associated with $x$ in the buffer at the root, and combine/cancel the operations accordingly: a new insert cancels an existing delete, or a new delete cancels an existing insert. In the end, there should only be one operation associated with $x$ in a buffer. (This is not entirely necessary, and could be dealt with later at the leaves; however it is a little easier here.)

As described above, we will maintain an invariant that at the end of each operation there are $< B$ items in a buffer. Thus there is always room in the buffer for the insert/delete operation. However, adding the operation may cause the buffer to have $\geq B$ items. In this case, we say that the buffer has *overflowed* and we will need to empty the buffer. See below for a description of how to empty a buffer.

**Searching.** To search for a key $x$, walk down the tree in the usual manner, searching for $x$. At each node, first check the buffer for any operations that modify key $x$, e.g., insert it or delete it. Operations higher in the tree are more recent than operations lower in the tree and hence have precedence. (For example, an insert operation in a buffer at the root takes precedence over a delete operation in a buffer at one of the root's children.) If you you discover an insert or delete operation in a buffer, then, you can safely return from the search. (If you find an operation modifying the value of a key, then you may need to continue to the leaf to discover whether the key is really in the tree, etc.)

If you do not find an operation in the buffer, or if the operation in the buffer is not sufficient to resolve the search operation, then examine the "pivots" stored in the node itself. Recall that each node has either one or two pivots (i.e., either two or three children). After examining the pivots, the search can continue at the appropriate child.

When you get to the leaf, you now know the status of $x$ (i.e., whether it is in the tree or not). Apply the operations discovered in the buffers on the root-to-leaf path in reverse order (i.e., starting at the leaf and working back toward the root) to determine the final status of key $x$.

**Emptying a buffer.** When the buffer at a node overflows, i.e., has more than $B$ items in it, we empty the buffer by distributing the operations in the buffer being emptied to the buffers of the children.

First, sort the operations in the buffer (using any reasonable sorting algorithm). Each operation is associated with a single key (e.g., being inserted, deleted, or updated), and we can order the operations by their key. Then, scanning the operations in the buffer, we can compare the key for each operation with the pivots in the node. In this way, we can partition the operations into groups, one for each child. (That is, there will be either two or three groups, depending on the number of children.)

Each group of operations is then moved from the buffer of the node being emptied to the buffer of the proper child. If the child is not a leaf, then there are two cases to consider.

- There is enough room in the child buffer for all the operations being moved. In this case, we simply move the operations from the parent buffer to the child buffer. If the child buffer has $\geq B$ items, then we empty it recursively (maintaining the invariant that a buffer has $< B$ items).
- This is not enough room in the child buffer for all the operations being moved. Notice that there are at most $2B$ operations being moved to the child buffer (since the parent buffer is of size $2B$), and when the operation begins, the child buffer has at least $B$ free space. In this case, we first copy enough operations to completely fill the child buffer. This will be at least $B + 1$ operations. Then, we empty the child buffer (recursively). Finally, we finish copying the remaining $< B$ items to the child (leaving the child buffer with enough room that it does not need to be emptied again).

Notice that in either case, the child buffer is emptied recursively at most once.

At a leaf, there is no buffer. Hence if the buffer at a parent of a leaf needs to be emptied, we cannot perform the same buffer emptying above. Instead, the operations need to be applied directly to the children.

- For delete operations, delete the item from the leaf (if it exists). If the leaf becomes too empty (i.e., $\leq B/2$ keys), then add it to a list to be dealt with later.
- For update operations, update the value associated with the key directly (if it exists).
- For insert operations, insert the key into the leaf. If the child has $2B$ keys after the insertion, split the leaf and propagate the insertion up the tree (splitting buffers as you go).

When all the buffer emptying is complete, then we go and finish the delete operations, merging nodes and propagating up the tree as needed. As we merge nodes, we may need to empty more buffers (e.g., when we merge two nodes that have buffers of size $> B/2$), and so the process may continue. The reason we wait until the buffer emptying is complete is to implement the deletes is because during operations, some buffers may have $> B$ elements and so merging two buffers may result in $> 2B$ elements in a buffer.

## Analysis

Overall, we find that inserts and deletes have amortized cost $O((1/B)\log(n)$ and searches have cost $O(\log(n)$. Analyzing searches is easy: the tree has height $\log n$, and that bounds the cost of a search.

To analyze inserts, we are going to use amortized analysis. Imagine that each insert or delete operation has a bank account, and when we begin such an operation, we deposit $\Theta((1/B)\log(n))$ dollars into the account. Whenever we move an operation down from a parent buffer to a child buffer, we withdraw $O(1/B)$ dollars from its account to pay for the cost of emptying the buffer.

Each time we empty a buffer, we move at least $B$ items, and hence we can afford $\Theta(B/B) = \Theta(1)$ cost. And notice that emptying a buffer has cost $O(1)$, as it accesses $O(1)$ blocks in the parent buffer, and $O(1)$ blocks in each of the child buffers. (Recursive emptying of child buffers is paid before by the items that move during those emptying phases.)

Finally, an insert or delete may cause a split or join operation that cascades up the tree at a cost of $\Theta(\log(n))$. Notice that after a split or a merge operation, a leaf has at least $3B/4$ items and at most $3B/2$:

- A leaf is split when it has at least $2B$ keys, resulting in two nodes with $B$ keys.
- If a leaf is merged because it has $< B/2$ keys, and if together with its sibling it has $> 3B/2$ keys, then it shares the keys with its sibling. Each gets at least $3B/4$—since by assumption they have $> 3B/2$ keys joints. And each gets at most $B$ keys—since by assumption they each have at most $2B$ keys. (Recall in this case, the merge does not propagate up the tree.)
- If a leaf is merged because it has $< B/2$ keys, and if together with its sibling it has $\leq 3B/2$ keys, then it merges with its sibling. In this case, the new node has at most $3B/2$ keys (by assumption). It also has at least $B - 1 > 3B/4$ keys, since the sibling has at least $B/2$ keys and the node itself has at least $B/2 - 1$.

Thus, there will be at least $B/4$ more operations on the leaf before the next split/merge operation. Each of these $B/4$ operations can pay $\Theta((1/B)\log(n))$, yielding sufficient money to pay for the splitting and joining.

## Extensions

One variation is to increase the degree of each node to $\sqrt{B}$, i.e., make the tree a $(\sqrt{B}, 2\sqrt{B})$-tree. Now the height of the tree is $O(\log_B n)$, and hence the search time is $O(\log_B n)$. The amortized cost of an insert goes up, however. Each time we empty a buffer, we move $B$ items at a cost of $\Theta(\sqrt{B})$—since we may have to do one cache block access per child. Thus, each of the items being moved pays a cost of $O(1/\sqrt{B})$. Hence, the amortized cost of an insert is $O(\log_B(n)/\sqrt{B})$. Notice we get roughly the same performance on searches as a B-tree (within a factor of 2), while significantly reducing the cost of insertions.

This can be further generalized to the case where each node has degree between $B^\epsilon$ and $2B^\epsilon$ for some $0 < \epsilon < 1$ (where $B^\epsilon \geq 2$). In this case, the height of the tree is $(1/\epsilon)\log_B n$, and so searches run $O(1/\epsilon)$ times slower than a B-tree. At the same time, amortized cost of emptying buffers during an insert is $O(B^\epsilon/B)$, and so the amortized cost of an insert is $O((1/B^{1-\epsilon})(1/\epsilon)\log_B n)$.

What is the largest we can make the degree of a node? You can increase the degree to $M/B$, while making the size of the buffer at each node $M$. At this point, you might as well also make the size of each leaf $M$ keys. Now, searching becomes very slow, as it requires $M/B$ block loads at each level of the tree. However, the tree itself becomes very shallow, only depth $\log_{M/B}(n/B)$. Imagine you insert $n$ items into the tree at cost $(n/B)\log_{M/B}(n/B)$. Next, empty all the buffers: there are $O(n/M)$ nodes in the tree (since there are $n/M$ leaves), and emptying each one costs $O(M/B)$, hence the total cost of emptying all the nodes in $O(n/B)$. (The traversal can be simplified by keeping sibling pointers, but that is not necessary.) Now, simply traverse all the leaves of the tree in $O(n/B)$ time to produce a sorted list. This strategy yields the BufferSort algorithm which sorts everything in time $O((n/B)\log_{M/B}(n/B)$.

# 7 Priority Queue

Now let's consider a priority queue. (This will be important later for implementing Dijkstra's Algorithm.) Assume that the Priority Queue has a set of $(key, priority)$ pairs, and supports the following operations:

- Insert: adds a new $(key, priority)$ pair. If there is already an element with the same key in the tree, then it updates the priority if the new priority is smaller. Otherwise, it leaves the priority unchanged.
- Delete: removes the $(key, priority)$ priority for that key, if it exists.
- deleteMin: removes and returns the key with the smallest priority.
- decreaseKey: updates a $(key, priority)$ pair to a new, smaller priority. If the new priority is larger than the old one, then it does nothing. If the key is not in the tree, it is added.

We will show in this section how to build a priority queue that supports all operations in time $O((1/B)\log^2 n)$. There exist cache-efficient priority queues that have cost $O((1/B)\log n)$, but we will not see those today.[1]

The simplest way to implement a priority queue is to use a B-tree, where the nodes are sorted by priority. In this case, each operation has cost $O(\log_B(n))$.

To do better, we might try to use a Buffer Tree. Again, assume the keys are sorted by priority. Now we can support insert and delete in $O((1/B)\log n$ time. Unfortunately, there are two problems:

- Finding the minimum priority item is hard. It may be anywhere along the left edge of the tree. The deleteMin may cost $O(\log n)$.
- To perform decreaseKey, you need to find an element by key. There is no efficient way to do that in the buffer tree. We would need to maintain an auxiliary search structure, which would be expensive. Similarly, it is hard to find elements by key when inserting and deleting.

We will solve both of these problems the same way: by keeping the tree sorted by key (instead of priority), and keeping a local store of the smallest priority elements in each subtree. Imagine, for example, that this was a B-tree (instead of a Buffer Tree): then it would be easy to keep the tree sorted by key, while at the same time storing in each node in the tree a set containing the smallest $2B - 1$ items in the subtree:

- Each operation that inserted a key or decreased the priority of a key would update all the small element stores for all the nodes on the root-to-leaf path encountered.
- Each operation that deleted a key would delete that key from all the small element stores for all the nodes on the root-to-leaf path encountered. It would then re-fill the small-element-stores, as needed, by walking from the leaf where the delete occurred to the root and rebuilding the small-element-store in every node that had fewer than $B$ elements in its store. It would do this by copying values from the nodes childrens' stores.

The only difference in the Buffer Tree is that we cannot afford to go all the way to the leaves in every operation. Instead, all the updates are buffered and pushed down the tree only as needed. The updates to the small-element-stores must take this into account.

[1] See, for example, the Buffer Heap in the paper, "Cache-Oblivious Shortest Paths in Graphs Using Buffer Heap" by Chowdhury and Ramachandran. By contrast, the Tournament Tree in the paper, "Improved algorithms and data structures for solving graph problems in external memory" by Kumar and Schwabe is the basis of the algorithm described here.

## Data structure design

To build our cache-efficient priority queue, we will start instead with a Buffer Tree sorted by key, not by priority. We can now efficiently insert and delete items from our Priority Queue. It is also easy to execute a decreaseKey: we simply insert a $(decreaseKey, key, priority)$ instruction into the root buffer (just like an insert or a delete) and let it propagate down the tree just like other operations. When we get to a leaf, we modify the element with that key if the new priority is older than the previous priority.

In order to support deleteMin, we add a *small element store* $S$ to each non-leaf node in the tree. The small element store has the following properties:

- It is of size at least $B$ and at most $2B - 1$ at all times.
- At node $u$, the elements in the small element store $S$ represent copies of the smallest elements in the leaves of the subtree rooted at $u$ (including all operations in buffers of nodes in the subtree of $u$).

Now, deleteMin is easy to implement: simply take the smallest element from the small element store $S$ at the root, and delete it from the tree. (Notice that the small element store just contains copies of the elements. They remain at the leaves in their proper place.)

When there are $< B$ items in the small element store $S$ at the root (whether because of a delete or a deleteMin), we need to refill the store. Before we refill the store, we first empty the operation buffer, i.e., propagate all the $insert$, $delete$, and $decreaseKey$ operations in the buffer at the root to its children. (Notice that we do this even though the buffer may not be full.) Then, we refill the store as described below. In general, whenever any store falls below size $B$, we refill it in the same manner.

To refill a store at node $u$, we examine the children of node $u$. Our goal is to find the smallest $2B$ elements. We know that each child node has at least $B$ of the smallest elements in its subtree. Therefore, we proceed as follows:

- Flush the buffer at $u$. (At this point, the small element store at $u$ contains $< B$ keys. It may at this point hold zero keys, if more were deleted when the buffer was flushed.)
- Let $K$ be the smallest $2B$ items from the small element stores in the children.
- If $K$ contains all the elements for any child store, then refill the store for that child. (Notice there can be at most one, since each child store has at least $B$ items.)
- Let $K'$ be the smallest $2B$ items from the small element stores in the children.
- Add the elements in $K'$ to the small element store at $u$ (skipping duplicates).

This procedure ensures that when you refill the small element store at node $u$, it contains the smallest $2B - 1$ items in its subtree:

**Claim 2** *Assume you refill the small element store at node $u$. Then after the refill completes, the small element store at $u$ contains $2B - 1$ elements with the smallest priorities in the subtree rooted at $u$.*

**Proof** The proof is by induction on the height of the tree, starting at the leaves. For the leaves, this is clearly tree, since the small element store at a leaf contains exactly the items stored at that leaf, always. Assume that this holds for nodes at height $h - 1$. Consider a node $u$ at height $h$.

There are two cases. First, the set $K$ does not include all the elements from any child store. In this case, $K' = K$ must contain the smallest $2B - 1$ elements in the subtree rooted at $u$: if a child $v$ contains an element with smaller priority than those in $K$, then it was not included in the small element store at $v$; that means that it must have been larger than the largest element in the small element store at $v$ (by induction), and hence the largest item in the small element store at $v$ should have also been included in $K$—but it was not.

Second, the set $K$ includes all the elements from some child store $v$. Since this must include at least $B$ elements from child store $v$, there can only be at most one such child. After the store at $v$ is refilled, the new set $K'$ contains either elements that were in $K$, or new elements that were added to $v$. As a result, after the refill, set $K'$ does not include all the elements from any child store—unless it includes all $2B - 1$ elements from the refilled child store. In either case, by the identical argument as above (substituting $K'$ for $K$), we conclude that $K'$ contains the smallest keys in the subtree rooted at $u$. □

There is one final important component of the priority queue. Each node $u$ now has a buffer, storing operations, and a small element store containing elements. Each operation in the buffer targets some key that is in the subtree of node $u$. Each element in the store $S$ is an element in the subtree of node $u$. When a new operation is inserted into the buffer at $u$, we should also apply it to any of the elements in $S$ that have the same key. Specifically:

- For an $(insert, key, priority)$ operation in the buffer: If the key is in the store $S$, then we should update the priority, if the new priority is smaller than the existing priority. If the key is not in the store, but the priority of the key is smaller than the largest element in the small element cache, then we should add it to the small element cache. If the small element cache has more than $2B$ items, then we kick out the largest priority element in $S$.
- For a $(delete, key)$ operation in the buffer: If the key is in the store $S$, then we should delete it from the store.
- For a $(decreaseKey, key, priority)$ operation in the buffer: If the key is in the store $S$, then we should update the priority, if the new priority is smaller than the existing priority. If the key is not in the store, but the priority of the key is smaller than the largest element in the small element cache, then we should add it to the small element cache. If the small element cache has more than $2B$ items, then we kick out the largest priority element in $S$.

When two nodes are merged (during a Buffer Tree operation), we simply merge their stores, sort them, and discard all but the smallest $2B$ items in the store. When two sibling nodes share keys, the two stores are combined, sorted, and then partitioned properly between the two nodes. When a node is split, we sort the store and partition it between the two new nodes, refilling their stores as needed.

Notice that a decreaseKey and an insert are, in this case, identical operations. At a leaf they should also be implemented identically. This is important because we cannot tell whether a key is in the tree or not until the operation reaches a leaf. Hence we choose to implement both insert and decreaseKey in a manner that does not require knowing in advance whether the key is in the tree.

These operations collectively ensure that the invariants for the small element store continue to hold. (Exercise: prove this by induction.)

## Analysis

The only new cost that we have added (compared to a Buffer Tree) is refilling the stores. Every so often, we need to fill a store, examining the children and copying the $2B$ smallest elements (while potentially recursively refilling stores). Each such refill operation costs $O(1)$ plus the cost of any recursive refilling, as it accesses a constant number of blocks.

To analyze this, we will use the accounting method of amortized analysis (again). Assume that each node has an account to pay for the cost of refill operations. Assume that the cost of accessing the small element store at a node and all of its children is $c = O(1)$. We will maintain the following invariant: whenever a refill operation occurs at node $u$ at height $h$, the account at $u$ contains at least $ch$ dollars.

Whenever a refill operation takes place at node $u$ of height $h$, we update the accounts as follows: we spend $c$ dollars to access the small element stores at $u$ and its children; if we need to refill a child $v$ of $u$ recursively, then we move $c(h-1)$ dollars from the account of $u$ to the account of $v$ (to pay for the refill).

We will charge operations as follows. Whenever a delete or insert operations traverses a node $u$ (moving into its buffer), the it adds $(c/B)\log n$ dollars to the account at $u$. This increases the cost of an insert or delete by $\Theta((1/B)\log^2 n)$. Similarly, whenever a deleteMin occurs at the root, it deposits $(c/B)\log n$ dollars into the account at the root and every node on the path to the leaf of the item that is removed. This increases the cost of a deleteMin by $\Theta((1/B)\log^2 n)$.

The only remaining issue is to prove that each account has sufficient money to pay for the refill operation:

**Claim 3** *Whenever a refill operation on a store at node $u$ takes place, the account at $u$ contains at least $ch$ dollars.*

**Proof** First, consider the root. A refill occurs at the root only after there have been at least $B$ delete or deleteMin operations, and so the root account contains at least $B(c/B)\log n$ dollars. Next, consider some node $u$ at height $h$. A refill occurs at node $u$ for one of two reasons: either $B$ items have been deleted or modified in the subtree rooted at $u$, or $u$ is being refilled in order to complete the refill of a parent. In the former case, there must have been at least $B$ operations that depoisted $(c/B)\log n$ dollars. In the latter case, the parent refill deposits $ch$ dollars. Hence in all cases the account at $u$ contains at least $ch$ dollars. □

In total, then, we conclude that each of the operations above can be supported in $O((1/B)\log^2(n))$ time.

# 8 Dijkstra's Algorithm

We now consider the problem of finding shortest paths in a graph from a single source $s$. Assume the graph $G = (V, E)$ with $n$ nodes and $m$ edges is stored as an adjacency list. Each edge $(u, v)$ has a weight $w(u, v) > 0$.

For today's purposes, we are going to assume that each node has a unique shortest distance from the source, i.e., for every pair of nodes $u$ an $v$, the distance $d(s, u) \neq d(s, v)$. This simplifies the analysis, as we do not need to worry about breaking ties. To implement the algorithm in this section in the general case, you need to specify a proper rule for breaking ties.

## 8.1 Review

Recall the basic idea behind Dijkstra's Algorithm. Each node $u$ stores an estimate $est[u]$ of its distance from the root. It uses a priority queue to find the node with smallest current estimate that has not yet been examined. It then extracts that node $u$ from the priority queue and relaxes all of its outgoing edges, updating the estimates in the priority queue for all of $u$'s neighbors. At the same time, it finalizes its estimate for $u$, declaring the current estimate for $u$ to be its distance from $s$. Eventually, once all the nodes have been extracted from the priority queue (and all the edges relaxed), we have calculated the shortest distance to every node in the graph.

To state the algorithm a bit more precisely, let $P$ be the priority queue, and assume that initially, the source $s$ is added with an estimate of 0. The algorithm proceeds as follows:

The analysis proceeds by induction, showing that when an item is extracted from the priority queue, its estimate is correct (i.e., it is equal to its distance on a shortest path from the source). Basically, nodes are divided into three categories: resolved nodes $R$ that have already had their distances finalized (by induction), fringe nodes $F$ that are neighbors of nodes in $R$, and other nodes $O$. Each iteration of the main loop extracts the node $w$ with the smallest estimate in $F$. If there is a shorter path to $w$ than indicated by the estimate, then there must be a path $P$ starting at $s$ and ending at $w$. Moreover, the distance on path $P$ has to be less than the estimate at $u$. But this path $P$ must include some other fringe node $z$ (as it must at some point cross from $R$ to $F$), and the estimate at $z$ must be at least as large as the estimate at $u$—otherwise it would have been selected from the priority queue instead of $u$. And the path $P$ must have a distance that is at least as large as the estimate at $z$. Hence the length of path $P$ cannot be smaller than the estimate at $u$.

**Algorithm 1:** Dijkstra($G = (V, E), w, n, m$)

```
1 Initialize priority queue: P.insert(s, 0).
2 repeat n times:
3     ⟨v, p⟩ = P.deleteMin()
4     finished[v] = true
5     dist[v] = p
6     for each neighbor u of v:
7         if finished[u] ≠ true and (dist[v] + e(v, u)) < est[u]) then
8             Relax edge (v, u): P.decreaseKey(u, dist[v] + e(v, u))
```

It is relatively easy to see that during the execution of Dijkstra's algorithm, there are the following queue operations:

- $n$ deleteMin operations (i.e., one for each iteration of the main loop)
- $m$ decreaseKey operations (i.e., one for each edge relaxed).

The remaining costs of the algorithm are $O(n + m)$: the main loop is executed $n$ times, and within that loop each edge is examined once. Hence if we implement the priority queue using a standard red-black tree, the total cost of Dijkstra's algorithm will be $O(m \log n)$, i.e., dominated by the decreaseKey operations. If instead we use a Fibonacci heap, the total cost will be $O(m + n \log n)$.

## 8.2 Using a better priority queue

We can run this same algorithm, using our cache-efficient priority queue. In this case, the total cost of all the priority queue operations will be $O((1/B)(m + n) \log(n))$. This is a significant improvement!

However, the remaining cost of the external loop remains $O(m + n)$, since each node and edge are accessed in an unstructured fashion. Thus the total cost ends up $O(n + m + (1/B)m \log(n))$. We would like to do better than this, since $m$ may be large. (In particular, if $m > n \log n$, we are better off using a Fibonacci heap.)

We are going to focus on improving lines 7 and 8:

> if ($finished[u] \neq true$ and $(dist[v] + e(v, u)) < est[u]$) then:
>
> Relax edge $(v, u)$: $P.decreaseKey(u, dist[v] + e(v, u))$

There are two things we want to avoid: we want to avoid checking $finished[u]$ and we want to avoid checking $est[u]$. These two accesses do not use the cache efficiently. One of these is easy: we do not need to check whether $(dist[v] + e(v, u)) < est[u]$. Recall that the priority queue automatically only decreases a key if the new priority is smaller than the old one. Otherwise, it is left unchanged. Hence we can replace that line of pseudocode with the following:

> if ($finished[w] \neq true$) then relax edge $(v, w)$: $P.decreaseKey(w, dist[v] + e(v, w))$

Since we are using our cache-efficient priority queue, this will perform the same check as part of the decreaseKey operation.

On the other hand, the second change is more difficulty: we want to remove the check whether $u$ is finished. That is, we want to replace the line of pseudocode with:

> Relax edge $(v, u)$: $P.decreaseKey(u, dist[v] + e(v, u))$

Recall, however, that once a node is finished, it is removed from the priority queue. And the decreaseKey operation will incorrectly add it back to the priority queue with a higher priority. This will cause all sorts of trouble, as we will later revisit node $w$ (and relax its edges again) even though it was completed. Ideally, we would like to do something like the following:

> Relax edge $(v, w)$: $P.decreaseKey(w, dist[v] + e(v, w))$
>
> if $finished[w]$ then $P.delete(w)$

Unfortunately, we cannot afford to check whether $w$ is finished.

Instead, we use the following trick. We maintain a second priority queue $SP$. The priority queue $SP$ contains only nodes that have already been finished, in particular, each key in $SP$ consists of an edge $(x, y)$ where $x$ is a node that is already finished. Whenever we complete a node $v$ and relax its edges, we add it to $SP$ with priorities as follows:

> For every neighbor $u$ of $v$:
>
> - Relax edge $(v, u)$: $P.decreaseKey(w, p + e(v, u))$
> - Add $((v, u), dist[v] + e(v, u))$ to $SP$.

Now, in each iteration of the main loop of Dijkstra's, we compare the smallest item in $P$ to the smallest item in $SP$. If $P$ contains the smallest item, then we proceed as before, extracting it and relaxing its outgoing edges. If $SP$ contains the smallest item, then we delete the already complete node from $P$, ensuring that we do not revisit it later.

## 8.3 Cache-efficient Dijkstra's

This leads to our final version of Dijkstra's: see Algorithm 2. Assume that the priority queue supports a $Peek$ function that returns the smallest item in the priority queue. (Clearly we can implement $Peek$ at the same cost as $deleteMin$, at worst.)

**Algorithm 2:** ExternalDijkstra$(G = (V, E), w, n, m)$

```
1  Initialize priority queue: P.insert(s, 0).
2  Initialize secondary priority queue SP. repeat until P is empty:
3      ⟨v, p⟩ = P.Peek()
4      ⟨(x, y), p'⟩ = SP.Peek()
5      if p ≤ p' then
6          ⟨v, p⟩ = P.deleteMin()
7          dist[v] = p
8          for each neighbor u of v:
9              Relax edge (v, u): P.decreaseKey(u, dist[v] + e(v, u))
10             Update SP: SP.insert((v, u), dist[v] + e(v, u))
11     else
12         ⟨(x, y), p'⟩ = SP.deleteMin()
13         P.delete(x)
```

## 8.4 Analysis

We will now analyze this modified version of Dijkstra's algorithm. Throughout, we will make a slightly funny assumption: we will assume that each node in the graph has a unique shortest distance to the source. This is equivalent

to saying that when a node is extracted from the priority queue $P$, it has a unique priority. This assumption makes the algorithm easier to state and analyze. (Otherwise, we need to state carefully how to break ties when two items have the same priority in the two different priority queues.)

Assume, temporarily, that the secondary priority queue has the desired effect, i.e., each node is only extracted from $P$ once. In that case, we get exactly the desired performance:

- Each node is extracted from $P$ once, i.e., cost $O((1/B)N \log^2 N/B)$.
- Each edge is decreased in $P$ once, i.e., cost $O((1/B)M \log^2 N/B)$.
- Each edge is added to $SP$ once and deleted from $SP$ once, i.e., cost $O((1/B)M \log^2 N/B)$.
- Each edge is added to $SP$ once and deleted from $SP$ once, i.e., cost $O((1/B)M \log^2 N/B)$.
- The only remaining unaccounted for cost is enumerating all the neighbors of each node, which has cost $O(N + M/B)$. (We have to access each node's adjacency list, so there is a minimum cost of $N$.)

This yields a total cost of $O(N + M/B + (1/B)m \log N/B) = O(N + (M/B) \log^2 N/B$.

The remaining piece we have to show is that each node is extracted from $P$ at most once. The first important fact is that the priority of the item being removed from the priority queue $P$ never decreases. This is a basic fact of Dijkstra's algorithm (and follows from the proof that Dijkstra's algorithm works):

**Claim 4** *Assume that two consecutive deleteMin operations on $P$ return $(u, p)$ and $(v, p')$. Then $p \leq p'$.*

**Proof** When $u$ is extracted from $P$, every other item in the priority queue has priority at least $p$. Hence the only way that we can extract $(v, p')$ afterward where $p' < p$ is if there is an intervening $insert(v, p')$ or $decreaseKey(v, p')$. After $u$ is extracted, we relax all the edges to its neighbors. So the only way that $(v, p')$ can be updated in $P$ is if $v$ is a neighbor of $u$. In that case, we do perform $decreaseKey(v, p + w(u, v)$. However, since $w(u, v) > 0$, we conclude that $p + w(u, v) > p$. Hence it cannot possibly result in a priority $p' < p$. Thus there are no operations on the priority queue $P$ between when $u$ and $v$ are extracted that update a priority $< p$. When $(v, p')$ is extracted, it must be the case that $p' \geq p$. □

By induction, then, we conclude that the priority of the item extracted from $P$ is always non-decreasing.

We will now show that the same node is not extracted from the priority queue twice. Assume, for the sake of contradiction, that $u$ is extracted from $P$ twice. Consider the very first time this happens:

```
P.deleteMin() ==> u
...
P.deleteMin() ==> u
```

When $u$ is extracted from $P$, we add all the outgoing edges from $u$ to $SP$. If $u$ is extracted twice, then sometime in between, $u$ must have been added back to $P$ when a neighbor $v$ of $u$ relaxed its outgoing edges:

```
P.deleteMin() ==> u
SP.insert((u,v), dist[u] + w(u,v))
...
relax (v,u) ==> P.decreaseKey(u, dist[v] + w(v,u))
...
P.deleteMin ==> u
```

We need to choose carefully which $v$ to look at: Let $t$ be the last time that $u$ is removed from priority queue $P$ before $u$ is extracted from $P$ for a second time—that is, $u$ is removed either during the first $deleteMin$ or during some later $delete$ operation before the second $deleteMin$. Choose $v$ to be the neighbor of $u$ extracted from $P$ after time $t$ with the minimum value of $dist[v] + w(u, v)$.

Notice that when $u$ is extracted from $P$ for the first time, we add $(u, v)$ to $SP$ with priority $dist[u] + w(u, v)$. Since nodes are extracted from $P$ in non-decreasing order, so we know that $dist[u] < dist[v]$ because $u$ was extracted (for the first time) from $P$ before $v$ (and we are assuming all the shortest paths are unique).

We can now examine what happens in the secondary priority queue $SP$:

**Claim 5** *When we extract $v$ from $P$ and relax $(v, u)$, we have not yet deleted $(u, v)$ from $SP$.*

This follows because after relaxing the edges of $u$, we know that the estimate of $v$ (in $P$) is $\leq dist[u] + w(u, v)$ because we performed a $P.decreaseKey(v, dist[u] + w(u, v))$ when $u$ was extracted from $P$. From this point on, the estimate at $v$ can only decrease. Since $(u, v)$ in $SP$ has priority $dist[u] + w(u, v)$, we conclude that $v$ will be extracted from $P$ and processed before $(u, v)$ is extracted from $SP$ and processed.

**Claim 6** *Before we extract $u$ for a second time, we delete $u$ from $P$.*

When $v$ is extracted from $P$ and relaxes its outgoing edge to $u$, the estimate for $u$ in $P$ is set to $dist[v] + w(u, v)$. Since $v$ is the neighbor of $u$ with the smallest value of $dist[v'] + w(u, v')$ (that is extracted after $u$ is last deleted from $P$ and before $u$ is extracted a second time), the estimate at $u$ will not decrease any further. Hence we can assume that after $v$ is extracted and up until $u$ is extracted, the estimate of $u$ in $P$ is $dist[v] + w(u, v)$.

We also know that the priority for $(u, v)$ in $SP$ is $dist[u]+w(u, v)$. Recall that $dist[u] < dist[v]$, since $u$ was extracted from $P$ before $v$ (and since every distance from $s$ is unique). Therefore, $dist[u] + w(u, v) < dist[v] + w(u, v)$ and hence $(u, v)$ will be extracted from $SP$ before we extract $u$ from $P$. When we extract $(u, v)$ from $SP$, we delete $u$ from $P$, as claimed, before $u$ is extracted from $P$ for a second time.

These two claims, however, create a contradiction: we fixed $t$ to be the last time that $u$ was removed from $P$ before its second extraction, and yet we have identified a time after $t$ where $u$ is deleted from $P$ before its second extraction. From this we conclude that $u$ is not extracted twice from $P$.[2]

Putting everything together, we get the following:

**Theorem 7** *Cache-efficient Dijkstra's algorithm correctly finds the shortest paths from a source $s$ to every node in the graph in time $O(N + (M/B) \log^2 N/B$. Using a more efficient priority queue (e.g., a Buffer Heap), this would run in time $O(N + (M/B) \log N/B$.*

[2]There are several ways to deal with the assumption that nodes have unique distances from $s$. If all the edge weights are integers, one solution is to add small random noise (e.g., in the range $(0, 1/n^2)$) to the edge weights. This ensures that, with high probability, the distances are unique, and yet the final distances will be correct when added to the nearest integer.

Alternatively, you can modify the way in which the algorithm breaks ties. Assume that for each entry in either priority queue, we also store a *tie-breaking* value which will consist of a distance and a node identifier. We will ensure that every entry in a priority queue has a unique tie-breaking value, and the priority queue should break ties using this value when needed. We will assign the tie-breaking values as follows: (1) whenever $u$ relaxes edge $(u, v)$, we will assign tie-breaking value $(dist[u], u)$; (2) when we add $(u, w)$ to $SP$, we assign tie-breaking value $(dist[u], u)$.

We also need to modify the algorithm in one way further: when we relax edge $(u, v)$, we annotate the entry with $u$. That means that if $u$ is the last node to decrease the priority on $v$, then when we extract $v$, we will learn about $u$. As always, if the relax operation does not decrease the key of $v$, then this annotation has no impact. Now, when we extract $v$, we will not relax the edge to $u$.

Together with the tie-breaking mechanisms, this ensures that the secondary priority queue operates correctly. I will leave it as an exercise to verify this, and/or to correct any bugs.

# 9 Cache-oblivious algorithms

Throughout, we have assumed that we know the value of $M$ and $B$, and they play a critical role in the algorithms we have considered. We say that an algorithm is *cache-oblivious* if it does not know the values of $M$ and $B$. In that case, it should provide the desired performance on any machine, and on any caching architecture, even if the values of $M$ and $B$ differ.

For a large number of the problems we are interested in, there are cache-oblivious versions that work as well (or almost as well) as the cache-aware versions we have seen so far. For example, FunnelSort is a cache-oblivious sorting algorithm that achieves optimal sorting performance. A BufferHeap is a cache-oblivious priority queue that performs almost as well as the cache-aware priority queue discussed here. By replacing the cache-aware priority queue with the cache-oblivious version, you get a cache-oblivious version of Dijkstra's algorithm.

We do not have time here to cover all of these algorithms and data structures. Instead, we present one simple example: a cache-oblivious search tree for storing static data. That is, assume we are given a set of (key, value) pairs $V$. The goal is to store them in a tree supporting the following operations:

- $search(key)$: searches for the specified key in $V$.
- $rangeQuery(k_1, k_2)$: returns all the values in $V$ that are in the range $[k_1, k_2]$.

Notably, we will not support inserting and deleting data from the set.

## 9.1 The van Emde Boas Tree Layout

The key idea is to arrange the data so as to allow efficient searching. First, assume that the keys are sorted and stored in an array. Let $n = |V|$, i.e., the number of key/value pairs we are storing. We will assume, for simplicity, that the size of the array is a power of 2, e.g., that $\log n$ is an integer. As we saw earlier, simply performing a binary search on this array has cost $O(\log(n/B)$; we would like to do better than that.

Instead, we build a tree to facilitate searching in the array. The root node $r$ of the tree represents the entire array, e.g., $A[1, \ldots n]$. The left child of the root represents the left half of the array and the right child of the root represents the right half of the array. We can then recursively build the rest of the tree:

- Given a node $v$ that represents a portion of the array $[b, e]$.
- Let $m = (b + e)/2$. Since $n$ is a power of 2, we know that $m$ will be non-integral. (For example, if $n = 32$, $b = 9$ and $e = 16$, then $m = 12.5$.)
- If $b = \lfloor m \rfloor$, set $v.left$ to point to $A[b]$. Otherwise, create a left node $v.left$ that represents $[b, \lfloor m \rfloor]$.
- If $e = \lceil m \rceil$, set $v.right$ to point to $A[e]$. Otherwise, create a right node $v.right$ that represents $[\lceil m \rceil, e]$.
- Node $v$ stores as its pivot the largest value in the array represented by $v.left$, i.e., it stores the value $A[\lfloor m \rfloor]$.

The leaves of the tree represent a single node; instead of creating a separate leaf, the parent points directly to the slot in the array represented by that node. Notice that we can search this tree as usual using the pivot value stored in each node. The tree has height $\log n$.

If we store each node of the tree separately, then the cost of a search will be $O(\log n)$. Instead, we are going to store all the nodes of the tree in a single array that is big enough to store every node in the tree. We are going to store the tree using the following divide-and-conquer approach, known as the van Emde Boas layout. Specifically, we are going to cut the tree at depth $\log n/2$, leaving a root subtree containing approximately $\sqrt{n}$ nodes and approximately $\sqrt{n}$ leaf subtrees, each containing approximately $\sqrt{n}$ nodes.

- The root of the tree has depth 0, and the leaves have depth $\log n$.
- If the tree only has one node, then the layout for the tree simply consists of that node.
- Otherwise, let $T_r$ be the subtree consisting of all the nodes of depth $\leq \lfloor \log n/2 \rfloor$.
- Let $T_1, T_2, \ldots, T_k$ be the subtrees consisting of all the nodes of depth $> \lfloor \log n/2 \rfloor$, each subtree rooted at a unique node of depth $\lfloor \log n/2 \rfloor + 1$.
- Let $L(T_i)$ be the recursive van Emde Boas layout for subtree $T_i$.
- Then arrange the array as follows: $L(T_r), L(T_1), L(T_2), \ldots, L(T_k)$.
- For each node, update the pointer to its right and left children to indicate the index in the array layout at which they are stored.

There are a couple things to notice about this layout. First, since the depth of the root subtree $T_r$ is $\lfloor \log n/2 \rfloor$, we know that it contains at least $\sqrt{n}/2$ leaves and at most $\sqrt{n}$ leaves. This implies that the root subtree has at least $\sqrt{n}-1$ nodes and at most $2\sqrt{n}-1$ nodes.

Next, since the root subtree has between $\sqrt{n}/2$ and $\sqrt{n}$ leaves, it also has between $\sqrt{n}$ and $2\sqrt{n}$ children, hence $\sqrt{n} \leq k \leq 2\sqrt{n}$. The depth of each of these child subtrees is $\lceil \log n/2 \rceil - 1$. (We subtract 1 because the root has depth 0.) This means that each child subtree has at least $\sqrt{n}/2$ leaves and at most $\sqrt{n}$ leaves. This implies that the leaf subtrees also have at least $\sqrt{n}-1$ nodes and at most $2\sqrt{n}-1$ nodes.

## 9.2 Analysis

To understand the performance, notice that we can break the final array up into chunks, each of which contains a recursively laid out subtree. Divide the array into chunks where each subtree has size $\leq B$ nodes, but $\geq \sqrt{B}/2$ nodes. Notice that if we have a subtree of size $> B$, we can always look at the recursive layout of the subtrees of that node; those subtrees will always have size $\geq \sqrt{B}-1$. Thus we can repeat the procedure until we find subtrees of the proper size.

Notice that each of these subtrees is stored in at most 2 consecutive blocks in memory, i.e., if a search of the tree traverses this subtree, then it will cost $O(1)$ memory transfers.

It remains to calculate the number of subtrees accessed. Notice that the entire original tree $T$ has $n$ leaves and hence height at most $\log 2n = \log(n) + 1$. Each of the subtrees has height at least $\log(\sqrt{n}/2) = (1/2)\log(n) - 1$. Thus any root-to-leaf traversal has cost at most $O(\log n/\log B) = O(\log_B(n)$.

**Claim 8** *Searching for a value in the van Emde Boas tree has cost $O(\log_B n)$.*

To search for a range of values $[k_1, k_2]$, it is sufficient to search for value $k_1$, and then scan the array:

**Claim 9** *Searching for a range $[k_1, k_2]$ in the van Emde Boas tree that contains $k$ values in the range has cost $O(\log_B n + k/B + 1)$.*