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 ℓ 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 $3B/4$ and at most B keys.
- If a leaf is merged because it has $< B/2$ keys, and if together with its sibiling it has $\leq 3B/2$ keys, then it merges with its sibiling.

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.