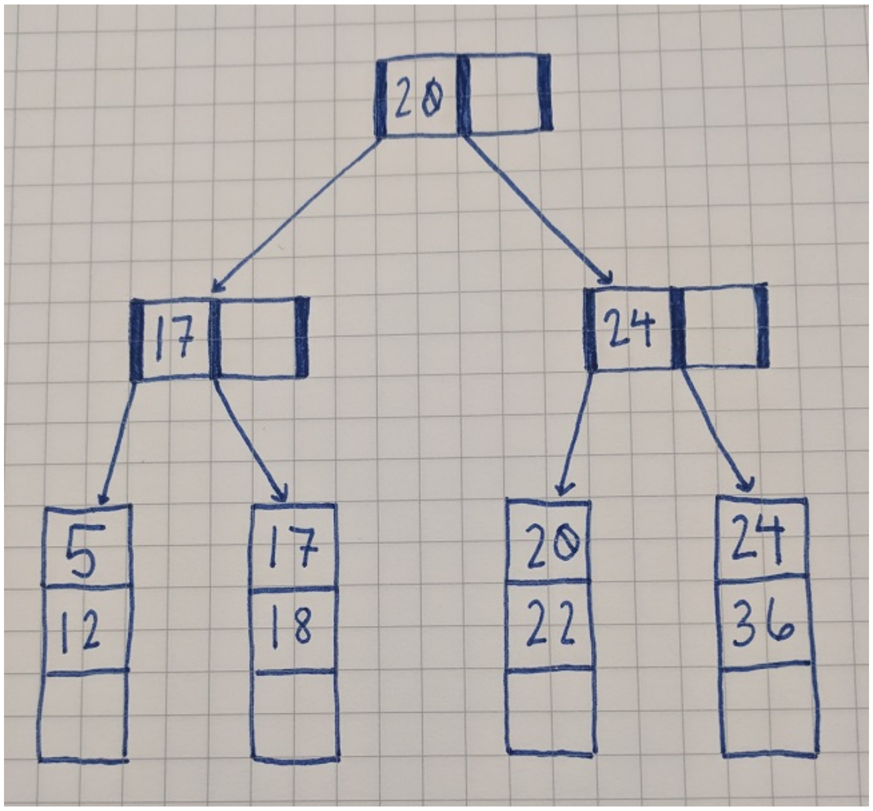
Section 06: Solutions

1. B-Trees

(a) Insert the following into an empty B-Tree with M = 3 and L = 3: 12, 24, 36, 17, 18, 5, 22, 20.

Solution:



(b) Given the following parameters for a B-Tree with $M = 11$ and $L = 8$:

- Key Size = 10 bytes
- Pointer Size = 2 bytes
- Data Size = 16 bytes per record (includes the key)

Assuming that M and L were chosen appropriately, what is the likely page size on the machine where this implementation will be deployed? Give a numeric answer based on two equations using the parameter values above. **Hint**: Some equations you might need to use are:

$$M = \lfloor \frac{p+k}{t+k} \rfloor$$
$$L = \lfloor \frac{p-t}{k+v} \rfloor$$

where p is the page size in bytes, k is key size in bytes, t is pointer size in bytes, and v is value size in bytes. **Hint**: Think about where these values come from.

Solution:

$$p \geq Mt + (M-1)k$$
$$\geq (11)(2) + (11-1)(10)$$
$$\geq 122p \qquad \geq t + L(k+v) \geq 2 + 8(16) \qquad \geq 2 + 128 \geq 130$$

Page size must be at least 130 bytes.

(c) Give an example of a situation that would be a good job for a B-tree. Furthermore, are there any constraints on the data that B-trees can store?

Solution:

B-trees are most appropriate for very, very large data stores, like databases, where the majority of the data lives on disk and cannot possibly fit into RAM all at once. B-trees require orderable keys. B-trees are typically not implemented in Java because because what makes them worthwhile is their precise management of memory.

(d) Find a tight upper bound on the worst case runtime of these operations on a B-tree. Your answers should be in terms of L, M, and n.

(i) Insert a key-value pair

Solution:

The steps for insert and delete are similar and have the same worst case runtime.

(i) Find the leaf: $\mathcal{O}\left(lg(M)log_M(n)\right)$.

(ii) Insert/remove in the leaf – there are L elements, essentially stored in an array: $\mathcal{O}\left(L\right)$

(iii) Split a leaf/merge neighbors: $\mathcal{O}\left(L\right)$

(iv) Split/merge parents, in the worst case going up to the root: $\mathcal{O}\left(Mlog_M(n)\right)$

The total cost is then $lg(M)log_M(n) + 2L + Mlog_M(n)$.

We can simplify this to a worst-case runtime $\mathcal{O}\left(L + Mlog_M(n)\right)$ by combining constants and observing that $Mlog_M(n)$ dominates $lg(M)log_M(n)$. Note that in the average case, splits for any reasonably-sized B-tree are rare, so we can amortize the work of splitting over many operations.

However, if we're using a B-tree, it's because what concerns us the most is the penalty of disk accesses. In that case, we might find it more useful to look at the worst-case number of disk lookup operations in the B-tree, which is $\mathcal{O}\left(log_M(n)\right)$.

(ii) Look up the value of a key

Solution:

(i) We must do a binary search on a node containing M pointers, which takes $\mathcal{O}(lg(M))$ time, once at each level of the tree.

(ii) There are $\mathcal{O}(logM(n))$ levels.

(iii) We must do a binary search on a leaf of L elements, which takes $\mathcal{O}(lg(L))$ time.

(iv) Putting it all together, a tight bound on the runtime is $\mathcal{O}(lg(M)log_M(n) + lg(L))$.

(iii) Delete a key-value pair

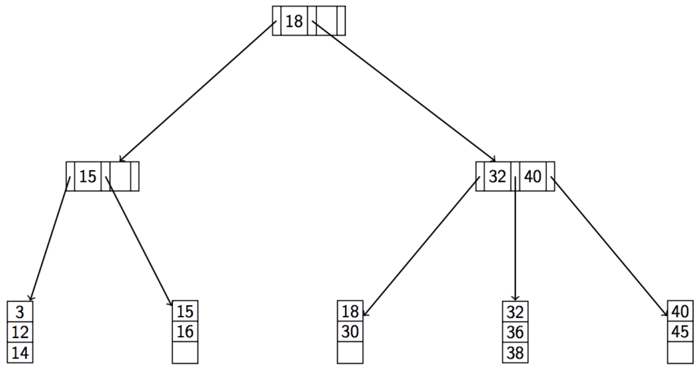
Solution:

See solution for inserting a key-value pair.

(e) Insert and then delete the following from a B-tree:

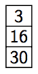
(i) Insert the following into an empty B-Tree with $M = 3$ and $L = 3$: 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45, 38.

Solution:



(ii) Delete 45, 14, 15, 36, 32, 18, 38, 40, 12 from the tree in the previous part.

Solution:



2. Memory

(a) What are the two types of memory locality?

Solution:

Spatial locality is memory that is physically close together in addresses. Temporal locality is the assumption that pages recently accessed will be accessed again.

(b) Does this more benefit arrays or linked lists?

Solution:

This typically benefits arrays. In Java, array elements are forced to be stored together, enforcing spatial locality. Because the elements are stored together, arrays also benefit from temporal locality when iterating over them.

(c) What about Java makes it a poor choice for implementing B-trees?

Solution:

Java can't page-align its memory allocation.

3. Memory and B-Trees

(a) Based on your understanding of how computers access and store memory, why might it be faster to access all the elements of an array-based queue than to access all the elements of a linked-list-based queue?

Solution:

The internal array within the array-based queue is more likely to be contiguous in memory compared to the linked list implementation of an array. This means that when we access each element in the array, the surrounding parts of the array are going to be loaded into cache, speeding up future accesses.

One thing to note is that the array-based queue won't necessarily automatically be faster then the linked-list-based one, depending on how exactly it's implemented.

A standard queue implementation doesn't support the `iterator()` operation, and a standard array-list based queue implements either $\mathcal{O}(n)$ enqueue or dequeue.

In that case, if we're forced to access every element by progressively dequeueing and re-enqueuing each element, iterating over a standard array-based queue would take $\mathcal{O}(n^2)$ time as opposed to the linked-list-based queue's $\mathcal{O}(n)$ time. In that case, the linked-list version is going to be far faster then the array-list version for even relatively smaller values of n.

The only way we could have the array-based queue be consistently faster is if it supported $\mathcal{O}(1)$ enqueues and dequeues. (Doing this is actually possible, albeit slightly non-trivial.)

(b) Why might `f2` be faster than `f1`?

```
public void f1(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim();        // omits trailing/leading whitespace
    }
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].toUpperCase();
    }
}

public void f2(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim(); // omits trailing/leading whitespace
        strings[i] = strings[i].toUppercase();
    }
}
```

Solution:

Temporal Locality. At each iteration, the specific string from the array is already loaded into the cache. When performing the next process `toUppercase()`, the content can just be loaded from cache, instead of disk or RAM.

(c) Let k be the size of a key, t be the size of a pointer, and v be the size of a value.

Write an expression (using these variables as well as M and L) representing the size of an internal node and the size of a leaf node.

Solution:

An internal node has M children, and therefore $M - 1$ keys inside. We need a pointer to each child, so we get $Mt + (M - 1)k$ bytes.

A leaf node is defined as having L key-value pairs, so the total size is $L(k + v)$ bytes.

Note: This is all assuming the node objects themselves contain no overhead. Java objects *do* have a certain amount of overhead associated with them, but many programming languages let you construct objects (or object-like structures) where their size in bytes is exactly the sum of the size of the fields, with no extra overhead.

For the sake of simplicity, we will be assuming we are using those kinds of programming languages, and that our B-tree nodes have no extra memory overhead.

(d) Suppose you are trying to implement a B-tree on a computer where the page size (aka the block size) is $p = 130$ bytes.

You know the following facts:

- Key size $k = 10$ bytes
- Value size $v = 6$ bytes
- Pointer size $t = 2$ bytes

What values of M and L should you pick to make sure that your internal and external nodes (a) fit within a single page and (b) uses as much of that page as possible.

Be sure to show your work. The equations you derived in the previous part will come in handy here.

Solution:

We want to pick the largest M and L that satisfy $Mt + (M-1)k \leq p$ and $L(k+v) \leq p$.

We can start by computing L, since that's easier. We have $L(10+6)+2 \leq 130$, which simplifies into $L \leq \frac{128}{16}$.

If we divide 128 by 16, we get about 8.0. L must be a whole number, but this works out anyway, so we know $L = 8$.

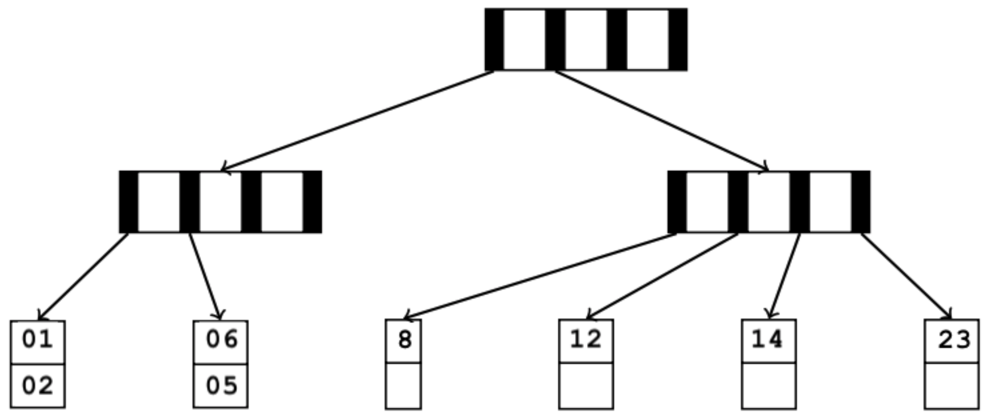
We can do something similar for M. We can rearrange the inequality:

$$\begin{aligned} Mt + (M-1)k &\leq p \\ Mt + Mk - k &\leq p \\ M(t+k) &\leq p + k \\ M &\leq \frac{p+k}{t+k} \end{aligned}$$

We plug in numbers, and get $M \leq \dfrac{130+10}{2+10}$. If we divide 140 by 12, we get about 11.66. So, we know $M = 11$.

Our final answer: $M = 11$ and $L = 8$.

(e) Consider the following "B-Tree":



(i) What are M and L?

Solution:

$M = 4$; $L = 2$.

(ii) Is there anything wrong with the above B-Tree? If so, what is wrong?

Solution:

(i) 6 and 5 should be swapped in the second leaf

(ii) The leaves containing 8 and 12 should be consolidated; the leaves containing 14 and 23 should be consolidated.

(iii) The inner nodes are missing the values: left-most inner node should have 5 in the first entry, right-most inner node should have 14 in the first entry and the root node should have 8 in the first entry.

(f) Consider the following code:

```
public static int sum(IList<Integer> list) {
    int output = 0;
    for (int i = 0; i < 128; i++) {
        // Reminder: foreach loops in Java use the iterator behind-the-scenes
        for (int item : list) {
            output += item;
        }
    }
    return output;
}
```

You try running this method twice: the first time, you pass in an array list, and the second time you pass in a linked list. Both lists are of the same length and contain the exact same values.

You discover that calling `sum` on the array list is consistently 4 to 5 times faster then calling it on the linked list. Why do you suppose that is?

Solution:

This is most likely due to spatial locality. When we iterate through a linked list, accessing the value at one particular index will load the next few elements into the cache, speeding up the overall time needed to access each element.

In contrast, each node in the linked list is likely loaded in a random part of memory – this means we likely must load each node into the cache, which slows down the overall runtime by some constant factor.

(g) Suppose you are trying to use a B-Tree somebody else wrote for your system. You know the following facts:

- $M = 10$ and $L = 12$
- The size of each pointer is 16 bytes
- The size of each key is 14 bytes
- The size of each value is 11 bytes

Assuming M and L were chosen wisely, what is most likely the page size on this system?

Solution:

If L is 12, and each key-value pair occupies $14 + 11 = 25$ bytes, we know each leaf node occupies at least $12 \times 25 = 300$ bytes.

If M is 10, we know each each branch node occupies at least $M \times 16 + (M - 1) \times 14 = 160 + 126 = 286$ bytes.

This leads us to conclude that the page size is most likely 300 bytes on this system.

4. Memory and B-Trees: A Sequel

(a) Suppose you are writing a program that iterates over an `AvlTreeDictionary` – a dictionary based on an AVL tree. Out of curiosity, you try replacing it with a `SortedArrayDictionary`. You expect this to make no difference since iterating over either dictionary using their iterator takes worst-case $\Theta(n)$ time.

To your surprise, iterating over `SortedArrayDictionary` is consistently almost 10 times faster!

Based on your understanding of how computers organize and access memory, why do you suppose that is? Be sure to be descriptive.

Solution:

This is almost absolutely because the `SortedArrayDictionary` is implemented with an array, which has much better spatial locality, than how the `AvlTreeDictionary` is most likely implemented, with a series of linked AVL tree nodes. Since we know that iterating over an array is faster than iterating over a linked list, the reasoning behind a faster tree is similar and reasonable.

(b) Excited by your success, you next try comparing the performance of the `get(...)` method. You expected to see the same speedup, but to your surprise, both dictionaries' `get(...)` methods seem to consistently perform about the same.

Based on your understanding of how computers organize and access memory, why do you suppose that is?

(Note: assume that the `SortedArrayDictionary`'s `get(...)` method is implemented using binary search.)

Solution:

Spatial locality can only be taken advantage of when iterating sequentially. With that, it's not surprising that because we have to jump from `i=100` to `i=50`, etc. until we find what we're looking for. If the array is so big that it spans over multiple pages, the locality that regular iteration takes advantage of is not available to jumping around with `get()`.

(c) You want to implement a B-Tree for a computer that has a page or block size of $p = 256$ bytes. Your pointers are $t = 4$ bytes long, your keys are $k = 2$ byte long, and your values are $v = 8$ bytes long. What should you select for M and L in order to maximize the performance of your B-tree? Please show your work.

Reminder: M and L must selected such that the following two inequalities remain true:

$$Mt + (M-1)k \le p \qquad \text{and} \qquad L(k+v) \le p$$

Solution:

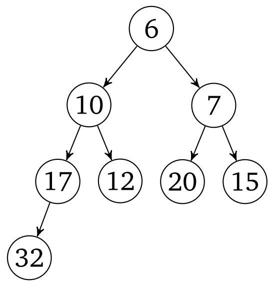
$M = 43$ and $L = 25$

5. Heaps

(a) Insert the following sequence of numbers into a *min heap*:

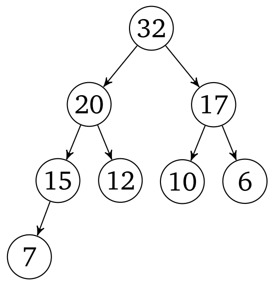
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



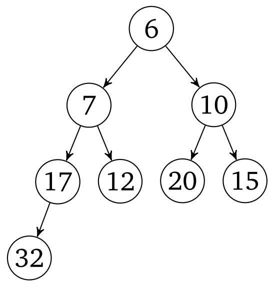
(b) Now, insert the same values into a *max heap*.

Solution:



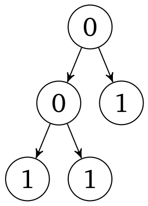
(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

Solution:



(d) Insert 1, 0, 1, 1, 0 into a *min heap*.

Solution:

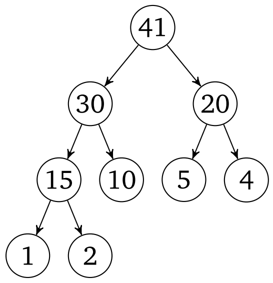


6. Heaps: A Sequel

(a) Insert the following numbers into a max heap in the order provided: 10, 15, 5, 2, 20, 30, 4, 1, 41

DO NOT use `buildHeap`; just insert them one by one. Showing your work might be prudent on a real exam (for partial credit).

Solution:



(b) Re-order the numbers so that if you were to insert them into a heap again, (1) the result would be an identical heap, and (2) the number of comparisons is minimal.

Solution:

41, 30, 20, 15, 10, 5, 4, 1, 2

(c) Draw out the array version of the heap from part (a).

Solution:

41	30	20	15	10	5	4	1	2