Section 05: Solutions

Section Problems

1. Selecting ADTs and data structures

For each of the following scenarios, choose:

(a) An ADT: `Stack` or `Queue`

(b) A data structure: `array list` or `linked list with front` or `linked list with front and back`

Justify your choice.

(a) You're designing a tool that checks code to verify all opening brackets, braces, parenthesis, etc... have closing counterparts.

Solution:

We'd use the `Stack` ADT, because we want to match the *most recent* bracket we've seen first.

Since `Stacks` push and pop on the same end, there is no reason to use an implementation with two pointers. (We don't need access to the "back" ever.)

Asymptotically (i.e. in big-$\mathcal{O}$ terms), there is no difference between the `LinkedList` with a `front` pointer and the `Array` implementation. In practice, the `Array` implementation is almost certain to be faster due to how computer caches work. Later in the quarter we will use the term *spatial locality* to explain this behavior.

(b) Disneyland has hired you to find a way to improve the processing efficiency of their long lines at attractions. There is no way to forecast how long the lines will be.

Solution:

We'd use the `Queue` ADT here, because we're dealing with…a line.

The important thing to note here is that if we try to use the implementation of a `LinkedList` with *only a* `front` *pointer*, either *add* or *next* will be very slow. That is clearly not a good choice.

Arguably, the `LinkedList` implementation with both pointers is better than the array implementation because we will never have to resize it.

(c) A sandwich shop wants to serve customers in the order that they arrived, but also wants to look ahead to know what people have ordered and how many times to maximize efficiency in the kitchen.

Solution:

This is still clearly the `Queue` ADT, but it's unclear that any of these implementations are a good choice!

One of the cool things about data structures is that if only one isn't good enough, you can use *two*. If we only care about the "normal queue features", then we would probably use the `LinkedList` implementation with one pointer. However, we can **ALSO** simultaneously use a *Map* to store the "number of times a food item appears in the queue".

2. Eyeballing Big-Θ bounds

For each of the following code blocks, what is the worst-case runtime? Give a big-Θ bound. You do not need to justify your answer.

(a)
```
void f1(int n) {
    int i = 1;
    int j;
    while(i < n*n*n*n) {
        j = n;
        while (j > 1) {
            j -= 1;
        }
        i += n;
    }
}
```

Solution:

$\Theta(n^4)$

One thing to note that the while loop has increments of $i+=n$. This causes the outer loop to repeat n^3 times, not n^4 times.

(b)
```
int f2(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println("j = " + j);
        }
        for (int k = 0; k < i; k++) {
            System.out.println("k = " + k);
            for (int m = 0; m < 100000; m++) {
                System.out.println("m = " + m);
            }
        }
    }
}
```

Solution:

$\Theta(n^2)$

Notice that the last inner loop repeats a small constant number of times – only 100000 times.

(c)
```
int f3(n) {
    count = 0;
    if (n < 1000) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < i; k++) {
                    count++;
                }
            }
        }
    } else {
        for (int i = 0; i < n; i++) {
            count++;
        }
    }
    return count;
}
```

Solution:

$\Theta(n)$

Notice that once n is large enough, we always execute the 'else' branch. In asymptotic analysis, we only care about behavior as the input grows large.

(d)
```
void f4(int n) {
    // NOTE: This is your data structure from the first project.
    LinkedDeque<Integer> deque = new LinkedDeque<>();
    for (int i = 0; i < n; i++) {
        if (deque.size() > 20) {
            deque.removeFirst();
        }
        deque.addLast(i);
    }
    for (int i = 0; i < deque.size(); i++) {
        System.out.println(deque.get(i));
    }
}
```

Solution:

$\Theta(n)$

Note that deque would have a constant size of 20 after the first loop. Since this is a `LinkedDeque`, `addLast` and `removeFirst` would both be $\Theta(1)$.

3. Best case and worst case runtimes

For the following code snippet give the big-Θ bound on the worst case runtime as well the big-Θ bound on the best case runtime, in terms of n the size of the input array.

```
1  void print(int[] input) {
2      int i = 0;
3      while (i < input.length - 1) {
4          if (input[i] > input[i + 1]) {
5              for (int j = 0; j < input.length; j++) {
```

```
                System.out.println("uh I don't think this is sorted plz help");
            }
        } else {
            System.out.println("input[i] <= input[i + 1] is true");
        }
        i++;
    }
}
```

Solution:

worst case: $\Theta(n^2)$ consider if the input is reverse sorted order – for each elemet we'd enter the inner for loop that loops over all n elements.

best case: $\Theta(n)$ consider if the input is already sorted and the check for if `(input[i] > input[i + 1])` is never true. Then the runtime's main contributor is just the outer while loop which will run n times.

4. Big-O, Big-Omega True/False Statements

For each of the statements determine if the statement is true or false. You do not need to justify your answer.

(a) $n^3 + 30n^2 + 300n$ is $\mathcal{O}(n^3)$ **Solution:**

T

(b) $nlog(n)$ is $\mathcal{O}(log(n))$ **Solution:**

F

(c) $n^3 - 3n + 3n^2$ is $\mathcal{O}(n^2)$ **Solution:**

F

(d) 1 is $\Omega(n)$ **Solution:**

F

(e) $.5n^3$ is $\Omega(n^3)$ **Solution:**

T

5. Tree Method

Find a summation for the total work of the following expressions using the Tree Method.

Hint: Just as a reminder, here are the steps you should go through for **any** Tree Method Problem:

i. Draw the recurrence tree.

ii. What is the size of the **input** to each node at level i? As in class, we call the root level $i = 0$. This means that at $i = 0$, your expression for the input should equal n.

iii. What is the amount of **work** done by each node at the i-th *recursive* level?

iv. What is the total number of nodes at level i? As in class, we call the root level $i = 0$. This means that at $i = 0$, your expression for the total number of nodes should equal 1.

v. What is the total work done across the i-th *recursive* level?

vi. What value of i does the last level of the tree occur at?

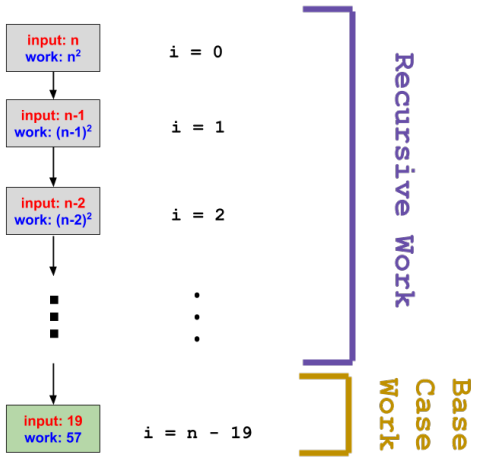
vii. What is the total work done across the base case level of the tree (i.e. the last level)?

viii. Combine your answers from previous parts to get an expression for the total work.

(a) $T(n) = \begin{cases} T(n-1) + n^2 & \text{if } n > 19 \\ 57 & \text{otherwise} \end{cases}$

Solution:

(i) Here's a drawing of the tree:



(ii) The input size at level i is $n - i$ (since we subtract the input by 1 at each level).

(iii) The previous answer makes the work in each recursive case node $(n - i)^2$, since each recursive node does the square of its input as work.

(iv) The number of nodes at any level i is 1 (since each node has a single child).

(v) Multiplying the work per recursive case node by the recursive nodes per level, we get: $1 \cdot (n - i)^2$.

(vi) The last level of the tree is when $n - i = 19$. Solving for i gives us $i = n - 19$.

(vii) The number of nodes in the base case level is 1, and each node does 57 work, so the total work is $1 \cdot 57$.

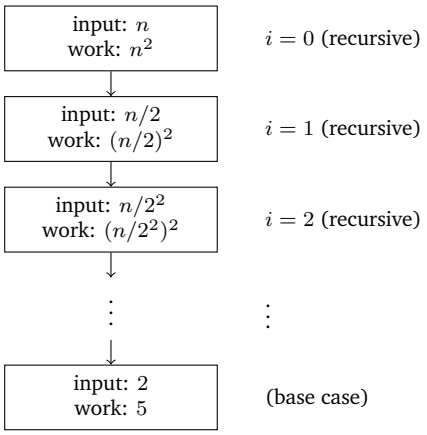
(viii) Summing up work across all recursive levels and then adding in the base case work, we get:

$$57 + \sum_{i=0}^{(n-19)-1} (n - i)^2$$

(b) $T(n) = \begin{cases} T(n/2) + n^2 & \text{if } n \geq 4 \\ 5 & \text{otherwise} \end{cases}$

Solution:

(i)



We assume the input size is a power of 2, so the base case input size is 2 instead of 3.

(ii) The input size is $n/2^i$ since we divide the input by 2 at each level.

(iii) The previous answer makes the work at each recursive node $(n/2^i)^2 = n^2/2^{2i}$, since each recursive node does the square of its input as work.

(iv) There is only 1 node at every level.

(v) The total work at each recursive level is $1 \cdot n^2/2^{2i}$.

(vi) The last level of the tree is when $n/2^i = 2$, so $i = \log_2 n - 1$.

(vii) Since there is only 1 node at the base case level, the total work is $1 \cdot 5$.

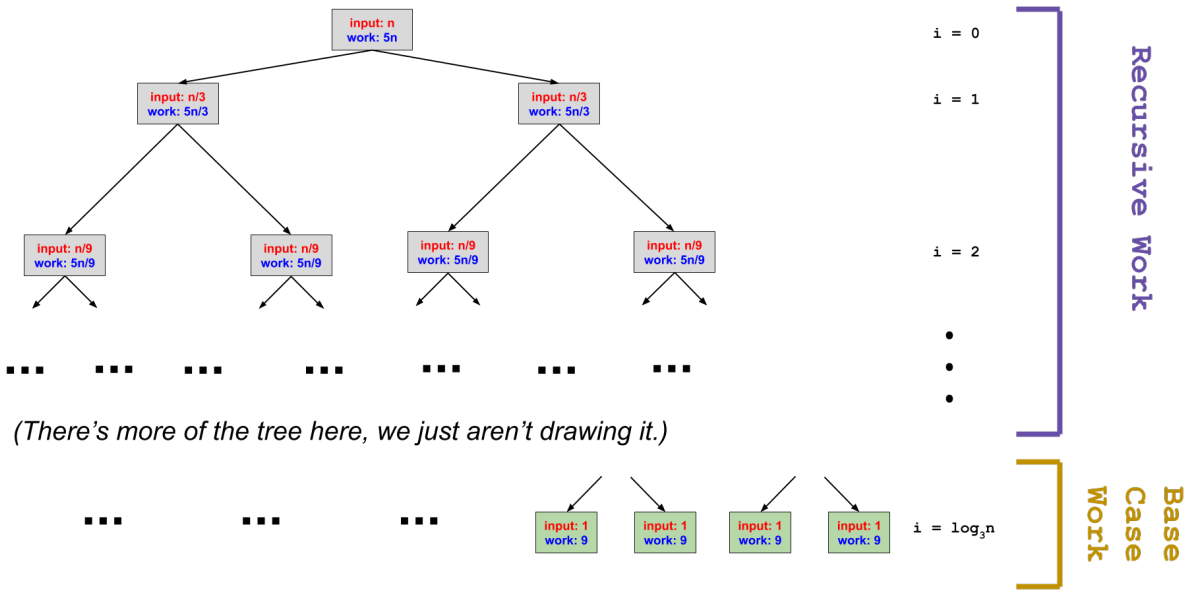
(viii)

$$5 + \sum_{i=0}^{\log_2 n - 2} \frac{n^2}{2^{2i}}$$

(c) $T(n) = \begin{cases} 2T(n/3) + 5n & \text{if } n > 1 \\ 9 & \text{otherwise} \end{cases}$

Solution:

(i) Here's a drawing of the tree:



(ii) The input size at level i is $n/3^i$ (since we divide the input by 3 at each level).

(iii) The previous answer makes the work in each recursive case node $5 \cdot (n/3^i)$, since each recursive node does five times its input as work.

(iv) The number of nodes at level i is 2^i (since each node has two children).

(v) Multiplying the work per recursive case node by the recursive nodes per level, we get: $5 \cdot \frac{n}{3^i} \cdot 2^i = 5n \cdot \left(\frac{2}{3}\right)^i$.

(vi) The last level of the tree is when $n/3^i = 1$. Solving for i gives us $i = \log_3(n)$.

(vii) Work across the base case level: The number of nodes (from previous parts) is $2^{\log_3(n)}$, and each node does 9 work, so the total work is $9 \cdot 2^{\log_3(n)}$.

(viii) Summing up work across all recursive levels and then adding in the base case work, we get:

$$\sum_{i=0}^{\log_3(n)-1} 5n \left(\frac{2}{3}\right)^i + 9 \cdot 2^{\log_3(n)}$$

6. Modeling

Consider the following method. Let n be the integer value of the `n` parameter, and let m be the size of the `LinkedDeque`.

```
public int mystery(int n, LinkedDeque<Integer> deque) {
    if (n < 7) {
        System.out.println("???");
        int out = 0;
```

```
            for (int i = 0; i < n; i++) {
                out += i;
            }
            return out;
        } else {
            System.out.println("???");
            System.out.println("???");
            out = 0;
            // NOTE: Assume LinkedDeque has working, efficient iterator.
            for (int i : deque) {
                out += 1;
                for (int j = 0; j < deque.size(); j++) {
                    System.out.println(deque.get(j));
                }
            }
            return out + 2 * mystery(n - 4, deque) + 3 * mystery(n / 2, deque);
        }
    }
```

Give a recurrence formula for the **worst-case** running time of this code. It's OK to provide a $\mathcal{O}$ for non-recursive terms (for example if the running time is $A(n) = 4A(n/3) + 25n$, you need to get the 4 and the 3 right but you don't have to worry about getting the 25 right). Just show us how you got there.

Solution:

$$T(n, m) = \begin{cases} 1 & \text{when } n < 7 \\ m^3 + T(n-4, m) + T(n/2, m) & \text{otherwise} \end{cases}$$

7. Challenge: Design

Imagine a database containing information about all trains leaving the Washington Union station on Monday. Each train is assigned a departure time, a destination, and a unique 8-digit train ID number.

What data structures you would use to solve each of the following scenarios. Depending on scenario, you may need to either (a) use multiple data structures or (b) modify the implementation of some data structure.

Justify your choice.

(a) Suppose the schedule contains 200 trains with 52 destinations. You want to easily list out the trains by destination.

Solution:

One solution would be to use a dictionary where the keys are the destination, and the value is a list of corresponding chains. Any dictionary implementation would work.

Alternatively, we could modify a separate chaining hash table so it has a capacity of exactly 52 and does not perform resizing. If we then make sure each destination maps to a unique integer from 0 to 51 and hash each train based on this number, we could fill the table and simply print out the contents of each bucket.

A third solution would be to use a BST or AVL tree and have each train be compared first based on destination then based on their other attributes. Once we insert the trains into a tree, we can print out the trains sorted by destination by doing an in-order traversal.

(b) In the question above, trains were listed by destination. Now, trains with the same destination should further be sorted by departure time.

Solution:

Regardless of which solution we modify, we would first need to ensure that the train objects are compared first by destination, and second by departure time.

We can modify our first solution by having the dictionary use a sorted set for the value, instead of a list. (The sorted set would be implemented using a BST or an AVL tree).

We can modify our second solution in a similar way by using specifically a BST or an AVL tree as the bucket type.

Our third solution actually requires no modification: if the trains are now compared first by destination and second by departure time, the AVL and BST tree's iterator will naturally print out the trains in the desired order.

(c) A train station wants to create a digital kiosk. The kiosk should be able to efficiently and frequently complete look-ups by train ID number so visitors can purchase tickets or track the location of a train. The kiosk should also be able to list out all the train IDs in ascending order, for visitors who do not know their train ID.

Note that the database of trains is not updated often, so the removal and additions of new trains happen infrequently (aside from when first populating your chosen DS with trains).

Solution:

Here, we would use a dictionary mapping the train ID to the train object.

We would want to use either an AVL tree or a BST, since we can list out the trains in sorted order based on the ID.

Note that while the AVL tree theoretically is the better solution according to asymptotic analysis, since it's guaranteed to have a worst-case runtime of $\mathcal{O}(\log(n))$, a BST would be a reasonable option to investigate as well.

big-O analysis only cares about very large values of n, since we only have 200 trains, big-O analysis might not be the right way to analyze this problem. Even if the binary search tree ends up being degenerate, searching through a linked list of only 200 element is realistically going to be a fast operation.

What's actually best will depend on the libraries you already have written, what hardware you actually run on, and how you want to balance code that will be sustainable if you get more trains vs. code that will be easy to understand and check for bugs right now.

8. Hash tables

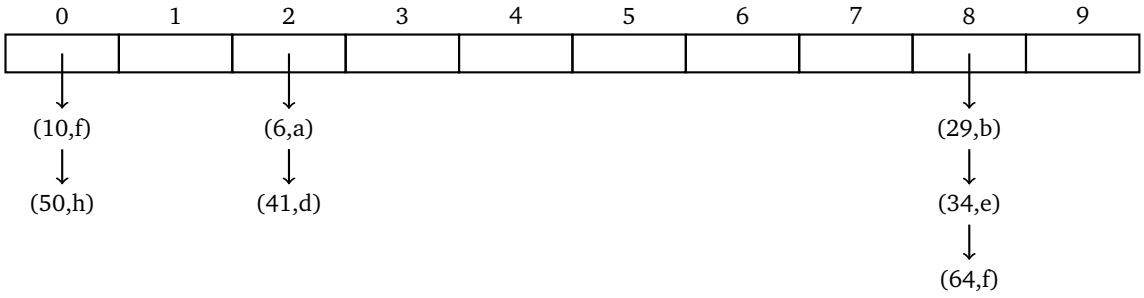
(a) Consider the following key-value pairs.

(6, a), (29, b), (41, d). (34, e), (10, f), (64, g), (50, h)

Suppose each key has a hash function $h(k) = 2k$. So, the key 6 would have a hash code of 12. Insert each key-value pair into the following hash tables and draw what their internal state looks like:

(i) A hash table that uses separate chaining. The table has an internal capacity of 10. Assume each bucket is a linked list, where new pairs are appended to the end. Do not worry about resizing.

Solution:



(ii) A hash table that uses linear probing, with internal capacity 10. Do not worry about resizing.

Solution:

0	1	2	3	4	5	6	7	8	9
(10, f)	(64, g)	(6, a)	(41, d)	(50, h)				(29, b)	(34, e)

(iii) A hash table that uses quadratic probing, with internal capacity 10. Do not worry about resizing.

Solution:

0	1	2	3	4	5	6	7	8	9
(10,f)	(50, h)	(6,a)	(41,d)				(64, g)	(29,b)	(34,e)

9. Analyzing dictionaries

(a) What are the constraints on the data types you can store in an AVL tree?

Solution:

The keys need to be orderable because AVL trees (and BSTs too) need to compare keys with each other to decide whether to go left or right at each node. (In Java, this means they need to implement `Comparable`). Unlike a hash table, the keys do *not* need to be hashable. (Note that in Java, every object is technically hashable, but it may not hash to something based on the object's value. The default hash function is based on reference equality.)

The values can be any type because AVL trees are only ordered by keys, not values.

(b) When is using an AVL tree preferred over a hash table?

Solution:

(i) You can iterate over an AVL tree in sorted order in $\mathcal{O}(n)$ time.

(ii) AVL trees never need to resize, so you don't have to worry about insertions occasionally being very slow when the hash table needs to resize.

(iii) In some cases, comparing keys may be faster than hashing them. (But note that AVL trees need to make $\mathcal{O}(\log n)$ comparisons while hash tables only need to hash each key once.)

(iv) AVL trees *may* be faster than hash tables in the worst case since they guarantee $\mathcal{O}(\log n)$, compared to a hash table's $\mathcal{O}(n)$ if every key is added to the same bucket. But remember that this only applies to pathological hash functions. In most cases, hash tables have better asymptotic runtime ($\mathcal{O}(1)$) than AVL trees, and in practice $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ have roughly the same performance.

(c) When is using a BST preferred over an AVL tree?

Solution:

One of AVL tree's advantages over BST is that it has an asymptotically efficient `find` even in the worst case.

However, if you know that `insert` will be called more often than `find`, or if you know the keys will be inserted in a random enough order that the BST will stay balanced, you may prefer a BST since it avoids the small runtime overhead of checking tree balance properties and performing rotations. (Note that this overhead is a constant factor, so it doesn't matter asymptotically, but may still affect performance in practice.)

BSTs are also easier to implement and debug than AVL trees.

(d) Consider an AVL tree with n nodes and a height of h. Now, consider a single call to `get(...)`. What's the maximum possible number of nodes `get(...)` ends up visiting? The minimum possible?

Solution:

The max number is $h + 1$ (remember that height is the number of edges, so we visit $h + 1$ nodes going from the root to the farthest away leaf); the min number is 1 (when the element we're looking for is just the root).

(e) **Challenge Problem:** Consider an AVL tree with n nodes and a height of h. Now, consider a single call to `insert(...)`. What's the maximum possible of nodes `insert(...)` ends up visiting? The minimum possible? Don't count the new node you create or the nodes visited during rotation(s).

Solution:

The max number is $h + 1$. Just like a `get`, we may have to traverse to a leaf to do an insertion.

To find the minimum number, we need to understand which elements of AVL trees we can do an insertion at, i.e. which ones have at least one `null` child.

In a tree of height 0, the root is such a node, so we need only visit the one node.
In an AVL tree of height 1, the root can still have a (single) `null` child, so again, we may be able to do an insertion visiting only one node.

On taller trees, we always start by visiting the root, then we continue the insertion process in either a tree of height $h - 1$ or a tree of height $h - 2$ (this must be the case since the the overall tree is height h and the root is balanced). Let $M(h)$ be the minimum number of nodes we need to visit on an insertion into an AVL tree of height h. The previous sentence lets us write the following recurrence

$$M(h) = 1 + \min\{M(h-1), M(h-2)\}$$

The 1 corresponds to the root, and since we want to describe the minimum needed to visit, we should take the minimum of the two subtrees.

We could simplify this recurrence and try to unroll it, but it's easier to see the pattern if we just look at the first few values:

$$M(0) = 1, M(1) = 1, M(2) = 1 + \min\{1, 1\} = 2, M(3) = 1 + \min\{1, 2\} = 2, M(4) = 1 + \min\{2, 2\} = 3$$

In general, $M()$ increases by one every other time h increases, thus we should guess the closed-form has an $h/2$ in it. Checking against small values, we can get an exactly correct closed-form of:

$$M(h) = \lfloor h/2 \rfloor + 1$$

which is our final answer.

Note that we need a very special (as empty as possible) AVL tree to have a possible insertion visiting only $\lfloor h/2 \rfloor + 1$ nodes. In general, an AVL of height h might not have an element we could insert that visits only $\lfloor h/2 \rfloor + 1$. For example, a tree where all the leaves are at depth h is still a valid AVL tree, but any insertion would need to visit $h + 1$ nodes.

10. Big-$\mathcal{O}$

Write down a tight big-$\mathcal{O}$ for each of the following. Unless otherwise noted, give a bound in the worst case.

(a) Insert and find in a BST.

Solution:

$\mathcal{O}(n)$ and $\mathcal{O}(n)$, respectively. This is unintuitive, since we commonly say that `find()` in a BST is "`log(n)`", but we're asking you to think about *worst-case* situations. The worst-case situation for a BST is that the tree is a linked list, which causes `find()` to reach $\mathcal{O}(n)$.

(b) Insert and find in an AVL tree.

Solution:

$\mathcal{O}(\log(n))$ and $\mathcal{O}(\log(n))$, respectively. The worst case is we need to insert or find a node at height 0. However, an AVL tree is always a balanced BST tree, which means we can do that in $\mathcal{O}(\log(n))$.

(c) Finding the minimum value in an AVL tree containing n elements.

Solution:

$\mathcal{O}(\log(n))$. We can find the minimum value by starting from the root and constantly traveling down the left-most branch.

(d) Finding the k-th largest item in an AVL tree containing n elements.

Solution:

With a standard AVL tree implementation, it would take $\mathcal{O}(n)$ time. If we're located at a node, we have no idea how many elements are located on the left vs right and need to do a traversal to find out. We end up potentially needing to visit every node.

If we modify the AVL tree implementation so every node stored the number of children it had at all times (and updated that field every time we insert or delete), we could do this in $\mathcal{O}(\log(n))$ time by performing a binary search style algorithm.

(e) Listing elements of an AVL tree in sorted order

Solution:

$\mathcal{O}(n)$. An AVL tree is always a balanced BST tree, which means we only need to traverse the tree in in-order once.

11. Memory, Caches, and B-Trees

(a) What is a cache and what is it used for?

Solution:

A cache is memory with a short access time that is used for the storage of frequently or recently used instructions or data. A cache is used to optimize the speed of data transfer between the CPU and RAM, but more generally, between any system elements with different characteristics (network interface cache, I/O cache, etc.)

(b) Define the two types of memory locality and give an example of when we might see each type of locality in code.

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. Temporal locality example is accessing a sum counter repeatedly or reading and writing to the same variable repeatedly. Spatial locality is accessing elements in an array.

(c) Does spatial locality benefit arrays or LinkedLists more when we are iterating through each data structure? Why?

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.

(d) Give an example of a situation that would be most appropriate for the use of a B-Tree as a data structure. Are there any constraints on the pieces of data that a B-Tree 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.