In fact, there is an infinite number of ranges we can use. The key is to make sure that the range is big enough and that all values are equally likely.

16.24 ***Pairs with Sum:*** Design an algorithm to find *all* pairs of integers within an array which sum to a specified value.

pg 185

SOLUTION

Let's start with a definition. If we're trying to find a pair of numbers that sums to z, the *complement* of x will be z - x (that is, the number that can be added to x to make z). For example, if we're trying to find a pair of numbers that sums to 12, the complement of –5 would be 17.

Brute Force

A brute force solution is to just iterate through all pairs and print the pair if its sum matches the target sum.

```
ArrayList<Pair> printPairSums(int[] array, int sum) {
   ArrayList<Pair> result = new ArrayList<Pair>();
   for (int i = 0 ; i < array.length; i++) {
      for (int j = i + 1; j < array.length; j++) {
         if (array[i] + array[j] == sum) {
            result.add(new Pair(array[i], array[j]));
         }
      }
   }
   return result;
}
```

If there are duplicates in the array (e.g., {5, 6, 5}), it might print the same sum twice. You should discuss this with your interviewer.

Optimized Solution

We can optimize this with a hash map, where the value in the hash map reflects the number of "unpaired" instances of a key. We walk through the array. At each element x, check how many unpaired instances of x's complement preceded it in the array. If the count is at least one, then there is an unpaired instance of x's complement. We add this pair and decrement x's complement to signify that this element has been paired. If the count is zero, then increment the value of x in the hash table to signify that x is unpaired.

```
ArrayList<Pair> printPairSums(int[] array, int sum) {
   ArrayList<Pair> result = new ArrayList<Pair>();
   HashMap<Integer, Integer> unpairedCount = new HashMap<Integer, Integer>();
   for (int x : array) {
      int complement = sum - x;
      if (unpairedCount.getOrDefault(complement, 0) > 0) {
         result.add(new Pair(x, complement));
         adjustCounterBy(unpairedCount, complement, -1); // decrement complement
      } else {
         adjustCounterBy(unpairedCount, x, 1); // increment count
      }
   }
   return result;
}

```

```
void adjustCounterBy(HashMap<Integer, Integer> counter, int key, int delta) {
  counter.put(key, counter.getOrDefault(key, 0) + delta);
}
```

This solution will print duplicate pairs, but will not reuse the same instance of an element. It will take O(N) time and O(N) space.

Alternate Solution

Alternatively, we can sort the array and then find the pairs in a single pass. Consider this array:

    {-2, -1, 0, 3, 5, 6, 7, 9, 13, 14}.

Let `first` point to the head of the array and `last` point to the end of the array. To find the complement of `first`, we just move `last` backwards until we find it. If `first + last < sum`, then there is no complement for `first`. We can therefore move `first` forward. We stop when `first` is greater than `last`.

Why must this find all complements for `first`? Because the array is sorted and we're trying progressively smaller numbers. When the sum of `first` and `last` is less than the sum, we know that trying even smaller numbers (as `last`) won't help us find a complement.

Why must this find all complements for `last`? Because all pairs must be made up of a `first` and a `last`. We've found all complements for `first`, therefore we've found all complements of `last`.

```
void printPairSums(int[] array, int sum) {
  Arrays.sort(array);
  int first = 0;
  int last = array.length - 1;
  while (first < last) {
    int s = array[first] + array[last];
    if (s == sum) {
      System.out.println(array[first] + " " + array[last]);
      first++;
      last--;
    } else {
      if (s < sum) first++;
      else last--;
    }
  }
}
```

This algorithm takes O(N log N) time to sort and O(N) time to find the pairs.

Note that since the array is presumably unsorted, it would be equally fast in terms of big O to just do a binary search at each element for its complement. This would give us a two-step algorithm, where each step is O(N log N).

16.25 LRU Cache: Design and build a "least recently used" cache, which evicts the least recently used item. The cache should map from keys to values (allowing you to insert and retrieve a value associated with a particular key) and be initialized with a max size. When it is full, it should evict the least recently used item. You can assume the keys are integers and the values are strings.

pg 185

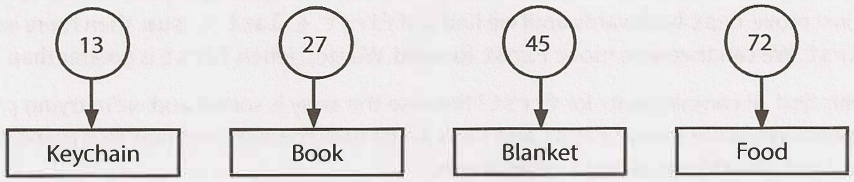
SOLUTION

We should start off by defining the scope of the problem. What exactly do we need to achieve?

- **Inserting Key, Value Pair:** We need to be able to insert a (key, value) pair.

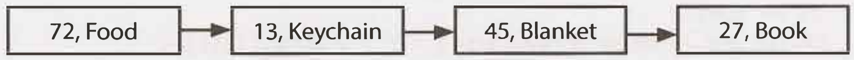
- **Retrieving Value by Key:** We need to be able to retrieve the value using the key.
- **Finding Least Recently Used:** We need to know the least recently used item (and, likely, the usage ordering of all items).
- **Updating Most Recently Used:** When we retrieve a value by key, we need to update the order to be the most recently used item.
- **Eviction:** The cache should have a max capacity and should remove the least recently used item when it hits capacity.

The (key, value) mapping suggests a hash table. This would make it easy to look up the value associated with a particular key.



Unfortunately, a hash table usually would not offer a quick way to remove the most recently used item. We could mark each item with a timestamp and iterate through the hash table to remove the item with the lowest timestamp, but that can get quite slow (O(N) for insertions).

Instead, we could use a linked list, ordered by the most recently used. This would make it easy to mark an item as the most recently used (just put it in the front of the list) or to remove the least recently used item (remove the end).

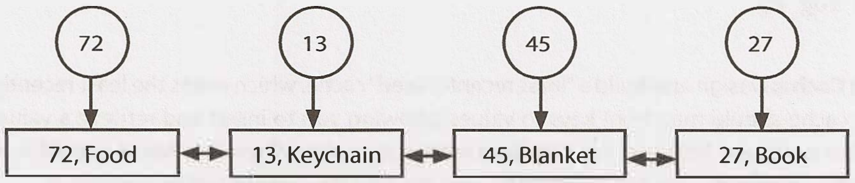


Unfortunately, this does not offer a quick way to look up an item by its key. We could iterate through the linked list and find the item by key. But this could get very slow (O(N) for retrieval).

Each approach does half of the problem (different halves) very well, but neither approach does both parts well.

Can we get the best parts of each? Yes. By using both!

The linked list looks as it did in the earlier example, but now it's a doubly linked list. This allows us to easily remove an element from the middle of the linked list. The hash table now maps to each linked list node rather than the value.



The algorithms now operate as follows:

- **Inserting Key, Value Pair:** Create a linked list node with key, value. Insert into head of linked list. Insert key -> node mapping into hash table.
- **Retrieving Value by Key:** Look up node in hash table and return value. Update most recently used item

(see below).

- **Finding Least Recently Used:** Least recently used item will be found at the end of the linked list.
- **Updating Most Recently Used:** Move node to front of linked list. Hash table does not need to be updated.
- **Eviction:** Remove tail of linked list. Get key from linked list node and remove key from hash table.

The code below implements these classes and algorithms.

```
public class Cache {
  private int maxCacheSize;
  private HashMap<Integer, LinkedListNode> map =
    new HashMap<Integer, LinkedListNode>();
  private LinkedListNode listHead = null;
  public LinkedListNode listTail = null;

  public Cache(int maxSize) {
    maxCacheSize = maxSize;
  }

  /* Get value for key and mark as most recently used. */
  public String getValue(int key) {
    LinkedListNode item = map.get(key);
    if (item == null) return null;

    /* Move to front of list to mark as most recently used. */
    if (item != listHead) {
      removeFromLinkedList(item);
      insertAtFrontOfLinkedList(item);
    }
    return item.value;
  }

  /* Remove node from linked list. */
  private void removeFromLinkedList(LinkedListNode node) {
    if (node == null)  return;

    if (node.prev != null) node.prev.next = node.next;
    if (node.next != null) node.next.prev = node.prev;
    if (node == listTail) listTail = node.prev;
    if (node == listHead) listHead = node.next;
  }

  /* Insert node at front of linked list. */
  private void insertAtFrontOfLinkedList(LinkedListNode node) {
    if (listHead == null) {
      listHead = node;
      listTail = node;
    } else {
      listHead.prev = node;
      node.next = listHead;
      listHead = node;
    }
  }

  /* Remove key/value pair from cache, deleting from hashtable and linked list. */
  public boolean removeKey(int key) {
      LinkedListNode node = map.get(key);
      removeFromLinkedList(node);
      map.remove(key);
      return true;
   }

   /* Put key, value pair in cache. Removes old value for key if necessary. Inserts
    * pair into linked list and hash table.*/
   public void setKeyValue(int key, String value) {
      /* Remove if already there. */
      removeKey(key);

      /* If full, remove least recently used item from cache. */
      if (map.size() >= maxCacheSize && listTail != null) {
         removeKey(listTail.key);
      }

      /* Insert new node. */
      LinkedListNode node = new LinkedListNode(key, value);
      insertAtFrontOfLinkedList(node);
      map.put(key, node);
   }

   private static class LinkedListNode {
      private LinkedListNode next, prev;
      public int key;
      public String value;
      public LinkedListNode(int k, String v) {
         key = k;
         value = v;
      }
   }
}
```

Note that we've chosen to make `LinkedListNode` an inner class of `Cache`, since no other classes should need access to this class and really should only exist within the scope of `Cache`.

16.26 Calculator: Given an arithmetic equation consisting of positive integers, +, -, * and / (no parentheses), compute the result.

EXAMPLE

Input: `2*3+5/6*3+15`

Output: `23.5`

pg 185

SOLUTION

The first thing we should realize is that the dumb thing—just applying each operator left to right—won't work. Multiplication and division are considered "higher priority" operations, which means that they have to happen before addition.

For example, if you have the simple expression 3+6*2, the multiplication must be performed first, and then the addition. If you just processed the equation left to right, you would end up with the incorrect result, 18, rather than the correct one, 15. You know all of this, of course, but it's worth really spelling out what it means.

Solution #1

We can still process the equation from left to right; we just have to be a little smarter about how we do it. Multiplication and division need to be grouped together such that whenever we see those operations, we perform them immediately on the surrounding terms.

For example, suppose we have this expression:

```
2 - 6 - 7*8/2 + 5
```

It's fine to compute 2 - 6 immediately and store it into a `result` variable. But, when we see `7*(something)`, we know we need to fully process that term before adding it to the result.

We can do this by reading left to right and maintaining two variables.

- The first is `processing`, which maintains the result of the current cluster of terms (both the operator and the value). In the case of addition and subtraction, the cluster will be just the current term. In the case of multiplication and division, it will be the full sequence (until you get to the next addition or subtraction).
- The second is the `result` variable. If the next term is an addition or subtraction (or there is no next term), then `processing` is applied to `result`.

On the above example, we would do the following:

1. Read +2. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
```
processing = {+, 2} --> null
result = 0          --> 2
```
2. Read -6. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
```
processing = {-, 6} --> null
result = 2          --> -4
```
3. Read -7. Apply it to `processing`. Observe next sign is a *. Continue.
```
processing = {-, 7}
result = -4
```
4. Read *8. Apply it to `processing`. Observe next sign is a /. Continue.
```
processing = {-, 56}
result = -4
```
5. Read /2. Apply it to `processing`. Observe next sign is a +, which terminates this multiplication and division cluster. Apply processing to `result`. Clear `processing`.
```
processing = {-, 28}    --> null
result = -4         --> -32
```
6. Read +5. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
```
processing = {+, 5} --> null
result = -32        --> -27
```

The code below implements this algorithm.

```
/* Compute the result of the arithmetic sequence. This works by reading left to
 * right and applying each term to a result. When we see a multiplication or
 * division, we instead apply this sequence to a temporary variable. */
double compute(String sequence) {
  ArrayList<Term> terms = Term.parseTermSequence(sequence);
  if (terms == null) return Integer.MIN_VALUE;

  double result = 0;
  Term processing = null;
  for (int i = 0; i < terms.size(); i++) {
    Term current = terms.get(i);
    Term next = i + 1 < terms.size() ? terms.get(i + 1) : null;

    /* Apply the current term to "processing". */
    processing = collapseTerm(processing, current);

    /* If next term is + or -, then this cluster is done and we should apply
     * "processing" to "result". */
    if (next == null || next.getOperator() == Operator.ADD
        || next.getOperator() == Operator.SUBTRACT) {
      result = applyOp(result, processing.getOperator(), processing.getNumber());
      processing = null;
    }
  }

  return result;
}

/* Collapse two terms together using the operator in secondary and the numbers
 * from each. */
Term collapseTerm(Term primary, Term secondary) {
  if (primary == null) return secondary;
  if (secondary == null) return primary;

  double value = applyOp(primary.getNumber(), secondary.getOperator(),
                         secondary.getNumber());
  primary.setNumber(value);
  return primary;
}

double applyOp(double left, Operator op, double right) {
  if (op == Operator.ADD) return left + right;
  else if (op == Operator.SUBTRACT) return left - right;
  else if (op == Operator.MULTIPLY) return left * right;
  else if (op == Operator.DIVIDE) return left / right;
  else return right;
}

public class Term {
  public enum Operator {
    ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
  }

  private double value;
  private Operator operator = Operator.BLANK;

  public Term(double v, Operator op) {
    value = v;
    operator = op;
  }

  public double getNumber() { return value; }
  public Operator getOperator() { return operator; }
  public void setNumber(double v) { value = v; }

  /* Parses arithmetic sequence into a list of Terms. For example, 3-5*6 becomes
```

```
     * something like: [{BLANK,3}, {SUBTRACT, 5}, {MULTIPLY, 6}].
    * If improperly formatted, returns null. */
   public static ArrayList<Term> parseTermSequence(String sequence) {
      /* Code can be found in downloadable solutions. */
   }
}
```

This takes `O(N)` time, where N is the length of the initial string.

Solution #2

Alternatively, we can solve this problem using two stacks: one for numbers and one for operators.

```
2 - 6 - 7 * 8 / 2 + 5
```

The processing works as follows:

- Each time we see a number, it gets pushed onto `numberStack`.
- Operators get pushed onto `operatorStack`—as long as the operator has higher priority than the current top of the stack. If `priority(currentOperator) <= priority(operatorStack.top())`, then we "collapse" the top of the stacks:
  - » Collapsing: pop two elements off `numberStack`, pop an operator off `operatorStack`, apply the operator, and push the result onto `numberStack`.
  - » Priority: addition and subtraction have equal priority, which is lower than the priority of multiplication and division (also equal priority).

  This collapsing continues until the above inequality is broken, at which point `currentOperator` is pushed onto `operatorStack`.
- At the very end, we collapse the stack.

Let's see this with an example: `2 - 6 - 7 * 8 / 2 + 5`

	action	numberStack	operatorStack
2	numberStack.push(2)	2	[empty]
-	operatorStack.push(-)	2	-
6	numberStack.push(6)	6, 2	-
-	collapseStacks [2 - 6] operatorStack.push(-)	-4 -4	[empty] -
7	numberStack.push(7)	7, -4	-
*	operatorStack.push(*)	7, -4	*, -
8	numberStack.push(8)	8, 7, -4	*, -
/	collapseStack [7 * 8] numberStack.push(/)	56, -4 56, -4	- /, -
2	numberStack.push(2)	2, 56, -4	/, -
+	collapseStack [56 / 2] collapseStack [-4 - 28] operatorStack.push(+)	28, -4 -32 -32	- [empty] +
5	numberStack.push(5)	5, -32	+
	collapseStack [-32 + 5]	-27	[empty]
	return -27		

The code below implements this algorithm.

```
public enum Operator {
  ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
}

double compute(String sequence) {
  Stack<Double> numberStack = new Stack<Double>();
  Stack<Operator> operatorStack = new Stack<Operator>();

  for (int i = 0; i < sequence.length(); i++) {
    try {
      /* Get number and push. */
      int value = parseNextNumber(sequence, i);
      numberStack.push((double) value);

      /* Move to the operator. */
      i += Integer.toString(value).length();
      if (i >= sequence.length()) {
        break;
      }

      /* Get operator, collapse top as needed, push operator. */
      Operator op = parseNextOperator(sequence, i);
      collapseTop(op, numberStack, operatorStack);
      operatorStack.push(op);
    } catch (NumberFormatException ex) {
      return Integer.MIN_VALUE;
    }
  }

  /* Do final collapse. */
  collapseTop(Operator.BLANK, numberStack, operatorStack);
  if (numberStack.size() == 1 && operatorStack.size() == 0) {
    return numberStack.pop();
  }
  return 0;
}

/* Collapse top until priority(futureTop) > priority(top). Collapsing means to pop
 * the top 2 numbers and apply the operator popped from the top of the operator
 * stack, and then push that onto the numbers stack.*/
void collapseTop(Operator futureTop, Stack<Double> numberStack,
                 Stack<Operator> operatorStack) {
  while (operatorStack.size() >= 1 && numberStack.size() >= 2) {
    if (priorityOfOperator(futureTop) <=
        priorityOfOperator(operatorStack.peek())) {
      double second = numberStack.pop();
      double first = numberStack.pop();
      Operator op = operatorStack.pop();
      double collapsed = applyOp(first, op, second);
      numberStack.push(collapsed);
    } else {
      break;
    }
  }
```

```java
}

/* Return priority of operator. Mapped so that:
 *     addition == subtraction < multiplication == division. */
int priorityOfOperator(Operator op) {
  switch (op) {
    case ADD: return 1;
    case SUBTRACT: return 1;
    case MULTIPLY: return 2;
    case DIVIDE: return 2;
    case BLANK: return 0;
  }
  return 0;
}

/* Apply operator: left [op] right. * /
double applyOp(double left, Operator op, double right) {
  if (op == Operator.ADD) return left + right;
  else if (op == Operator.SUBTRACT) return left - right;
  else if (op == Operator.MULTIPLY) return left * right;
  else if (op == Operator.DIVIDE) return left / right;
  else return right;
}

/* Return the number that starts at offset. * /
int parseNextNumber(String seq, int offset) {
  StringBuilder sb = new StringBuilder();
  while (offset < seq.length() && Character.isDigit(seq.charAt(offset))) {
    sb.append(seq.charAt(offset));
    offset++;
  }
  return Integer.parseInt(sb.toString());
}

/* Return the operator that occurs as offset. * /
Operator parseNextOperator(String sequence, int offset) {
  if (offset < sequence.length()) {
    char op = sequence.charAt(offset);
    switch(op) {
      case '+': return Operator.ADD;
      case '-': return Operator.SUBTRACT;
      case '*': return Operator.MULTIPLY;
      case '/': return Operator.DIVIDE;
    }
  }
  return Operator.BLANK;
}
```

This code also takes O(N) time, where N is the length of the string.

This solution involves a lot of annoying string parsing code. Remember that getting all these details out is not that important in an interview. In fact, your interviewer might even let you assume the expression is passed in pre-parsed into some sort of data structure.

Focus on modularizing your code from the beginning and "farming out" tedious or less interesting parts of the code to other functions. You want to focus on getting the core `compute` function working. The rest of the details can wait!