NOI.PH Training: Persistence

Persistent Data Structures

Kevin Charles Atienza

Contents

1 Introduction

In this lesson, we will discuss the idea of *persistence* and also delve a little bit into *declarative programming*.

What even is persistence? Roughly speaking:

Definition 1.1. A **persistent data structure** is a data structure that preserves all its previous versions.

For example, say you have a *set* of numbers. Normally, there's only one version of the set kept in memory, and once you insert some numbers, this version gets updated, and the old version—the state before the insertion—is lost. But with a persistent data structure, you may access all its old versions even after having inserted numbers.

For example, here's a valid sequence of operations on a fully persistent set (pseudocode):

```
s = new empty persistent set()

t = s.insert(5) # .insert returns the new version of the set, but the old one is still valid
u = t.insert(3)
v = u.insert(8)
# now, all versions are available. note that:
# s = {}
# t = {5}
# u = {3, 5}
# v = {3, 5, 8}
print s.contains(5) # prints false
print t.contains(5) # prints true
print u.contains(8) # prints false
print v.contains(8) # prints true
print v.contains(5) # prints true
print u.size() # prints 2

w = t.insert(7) # one can modify any version without affecting other versions
# w = {5, 7}
print w.contains(7) # prints true
print w.contains(8) # prints false
print v.contains(7) # prints false
print v.contains(8) # prints true

x = v.remove(5) # one can also remove elements.
# x = {3, 8}
print x.contains(5) # prints false
print x.contains(8) # prints true
print t.contains(5) # prints true
print t.contains(7) # prints false
print t.contains(8) # prints false
```

Obviously, it's easy to implement such a structure naïvely: you can simply *copy* the whole set every time a modification is made. And that's a valid approach! It's just inefficient. It takes a lot of time *and* memory. After all, every insertion or deletion requires a whole copy of the set. And if your set is a balanced binary tree (presumably for $\mathcal{O}(\log n)$ access times), then making it persistent this way cancels out any efficiencies gained from using a binary tree in the first place.

The main challenge is to come up with persistent data structures that are efficient—say, almost as good as their nonpersistent counterparts.

We motivate persistent data structures with two problems:

Problem 1.1. Implement a segment tree but with an additional operation, **revert**(x), which means revert the state of the segment tree to the version right after the xth **update** operation. (We define an **update** operation to be any operation that changes the state of the structure.) For example, **revert**(0) returns the segment tree to its initial state. You can consider this as some sort of a batch **undo** operation. (Note that **revert** is considered to be an **update** operation as well, so one can also revert a revert!)

Problem 1.2. Solve the NOI.PH 2016 eliminations round problem *A Little Bit Frightening*[a], but this time, you're required to answer the queries *online*, that is, you don't know what the queries will be beforehand, and you can only get the next query after you answer the current query.

[a] https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/in-fact-it-was

2 Imperative programming

Let's start by describing how a human programs in the first place. The kind of programming you're familiar with is probably **imperative programming**, which consists of *telling the computer what to do*. For example,

1. Writing `x = y + 1;` means telling the computer to *set the value of the variable x to the value of variable y plus* 1.
2. Writing `x = x + 1;` means telling the computer to *set the value of the variable x to the value of variable x plus* 1. In other words, *increment x by* 1.
3. Writing `break;` means telling the computer to *break out of the current (innermost) loop.*
4. Writing `return 5;` means telling the computer to *exit the current function and return the value* 5 *to the caller.*
5. Writing `for (<A>; <B>; <C>) {<D>}` means telling the computer the following things:
   - Run `<A>`.
   - While `<B>` is true, run `<D>`, and then run `<C>`.

Unfortunately, this style of programming is partly the cause of why learning programming (and crucially, *debugging*) is hard. For a simple example, consider the second item, "`x = x + 1;`". It looks like a glaring contradiction![1] "How can x equal $x + 1$? That's impossible, right?" Introducing someone to programming requires explaining what lines of code like this mean, and this entails explaining that *variables* have *state* that can change as the program executes. Note that this also means that variables don't retain their initial values, so to understand what a program does, one needs to keep track of what's happening to each variable as lines of code are run. To someone who isn't accustomed to thinking about the whole state of the program and memory all at once, this could be quite difficult.

Now, you're probably wondering, so what? Isn't it just the nature of programming? Well, you might be surprised to know that *this is not the only way to program*!

Declarative programming is a different style of programming that gets rid of state changes altogether. In other words, *you're not allowed to change the values of variables.*[2] This might sound bizarre, so we will explain this in a bit, but to motivate why we'd want to use such a paradigm, let's first explain the *evils of imperative programming.*

2.1 The evils of imperative programming

2.1.1 Global variables

Consider the following lines of code:

1. `cout << f(5,n) + f(5,n) << endl;`
2. `cout << f(5,n) * 2 << endl;`

[1] at least to the eyes of someone who hasn't learned programming yet. Obviously, this criticism is somewhat tongue-in-cheek.

[2] Actually, the word *variable* then becomes a misnomer; in declarative programming, the word *identifier* is usually used.

Assuming `f` returns an integer type, are these lines equivalent? Disregard efficiency issues here and consider only the output.

While the intuitive answer is *yes*, unfortunately the answer is *no*. Consider when `f` is defined this way:[3]

```
int z = 0;
int f(int x, int y) {
    return x + 3*y + ++z;
}
```

Now, assuming `n == 1` for example, `f(5,n) + f(5,n)` will be `19`, but `f(5,n) * 2` will be `18`. Thus, they're not equivalent!

The problem here is that we are calling `f` a "function", but in reality, `f` is not a *mathematical function*: every time `f` is called, some global state is changed (namely `z`). Thus, `f` can return different outputs even when called on the same arguments. However, mathematical functions are not like that. In math, $f(5,n) + f(5,n)$ is the same as $f(5,n) \cdot 2$. And in math, there's no such thing as a global state in the first place.

In fact, not only do the lines above output different things, but they can also affect succeeding lines. Consider for example doing `cout << z << endl;` after either.

Another consequence of this is that `f(a,b) * f(b,c)` is not equivalent to `f(b,c) * f(a,b)`, even though `f` returns an `int` (so multiplication is commutative). Convince yourself of this.

The takeaway here is that **using global variables is evil, so from now on we promise not to use global variables anymore.**[4] Let's make this official:

> **Rule 2.1** (Globals are Evil)**.** Using global variables is evil, so from now on we promise not to use global variables anymore.

By disallowing global variables, we can't define `f` that way anymore, and we avoid this problem.

2.1.2 Side effects

Now that we don't use globals anymore, consider the following two lines:

1. `cout << f(a,b) * f(b,c) << endl;`
2. `cout << f(b,c) * f(a,b) << endl;`

Now, are they equivalent? Unfortunately, the answer is still no. Consider for example when `f` is defined this way:

```
int f(int& x, int y) {
    return ++x + y;
}
```

Note, crucially, how `x` is declared with a `&`. It means *pass by reference*, so when `x` is incremented, `a` or `b` gets incremented as well (depending on the call).

[3] Recall that `++z` is shorthand for `z += 1`, i.e., "increment `z` before using it."

[4] This also means that we also promise not to use the `static` keyword inside functions, for those who know it.

Now, try running the lines above with the variables `a`, `b` and `c` initialized to, say, `a = 1`, `b = 2` and `c = 3`. Then the first line prints `24`, but the second prints `30`. Therefore, they're not equivalent!

The problem here is that even though global variables are not allowed, `f` is still allowed to modify things passed to it.[5] In other words, `f` has *side effects*. Thus, **side effects in functions are evil, so from now on we promise not to have side effects in functions**. By doing so, we disallow the use of the `++` operator (along with other assignment operators like `+=`, `*=` or even just `=` inside functions) and we can't define `f` as above anymore.

But this is not enough! Even without using functions, the evils of side effects still manifest. For example, consider this snippet of code:

```cpp
// version 1
int main() {
    int x = 5;
    int y = ++x;
    int z = ++x;
    cout << x << " " << y << " " << z << endl;
}
```

Notice that the expression "`++x`" appears twice, so naturally I would expect both occurrences to be completely equivalent, just like in math.[6] But since I assigned `y` to the expression `++x`, I expect to be able to substitute `y` whenever I see `++x` without changing the result, as long as it's after the declaration of `y`,[7] like so:

```cpp
// version 2
int main() {
    int x = 5;
    int y = ++x;
    int z = y;
    cout << x << " " << y << " " << z << endl;
}
```

It's now easy to see that these two snippets of code are not equivalent. Therefore, the mathematical paradigm of *substituting equivalent expressions when you see them* doesn't apply, and "=" doesn't represent equivalence.

The lesson here is that **side effects in general are evil, so we promise not to use side effects anywhere at all**.

Unfortunately, this is too restrictive; if we disallow side effects, we can't even initialize variables! So we will allow an exception: We will allow "=" only when initializing a variable.

> **Rule 2.2** (Side Effects are Evil)**.** Side effects are evil, so we promise not to use side effects anywhere at all.

[5] In this case, we only passed an integer so this can be resolved by not allowing *pass by reference*, but consider for example when we're passing data structures as arguments to a function. Disallowing passing by reference would be impractical. Thus, we shall continue allowing pass by reference.

[6] Assuming I have a math background and I don't know what `++` means yet.

[7] Note however that this is only a restriction in C++; in real declarative languages we can use an identifier even *before* it's initialized. It sounds weird, but it works!

> Except when initializing a variable.

Note by the way that taking input and printing output are considered to be side effects as well, but please ignore taking input and printing output for now. Note that the IOI format will be useful here: in the IOI, the grader is responsible for input and output, and the code you write simply takes in the input as arguments and returns the result. So let's just assume that the grader isn't required to follow our restrictions and is able to take the input and print output. If you like, you may assume that the grader simply magically obtains the input.

3 Declarative programming

3.1 Paradigm shift

Now, we should feel happy since we got rid of everything that is evil in programming. But now the question is, how the hack do we program anything now? It looks like by getting rid of all the evil, we also got rid of everything useful in programming.

So, how much can we even program with these limitations? The surprising answer is: *a whole lot*. And in fact, the restrictions that we currently have form the basis of *declarative programming*.

In fact, in some theoretically precise sense, everything computable using imperative programming can also be computed using declarative programming.[8] But more importantly, we gain some properties that are not present in imperative programming:

1. "Variables" are not variables. That is, once declared, their values stay the same (in the same scope[9]). For example, writing `int x = y+5;` means `x` will always have the value `y+5`. (So it may be better to called them *identifiers*.)

2. The *substitution property*. That is, declaring, say, `int x = <y>` allows us to replace each instance of the expression `<y>` with the identifier `x` (in the same scope) and vice versa.[10]

3. Since globals and side effects are disallowed, *function results are completely determined by their arguments*. In other words, functions in our new paradigm behave more like mathematical functions.

   For example, if `x = y`, then `f(x,3) = f(y,3)`. This is useful when coupled with the substitution property above. Note that this also means that *functions with zero arguments* are not very useful; one can simply substitute the call to such a function with its return value, by the substitution property. (Note that side effects are not allowed, so getting rid of the function call doesn't affect our program in any way.)

Another consequence of our restrictions is that there's no point in writing loops anymore. For example, consider the following snippet:

```
while (<X>) {
    <Y>
}
```

Assume without loss of generality that `<Y>` doesn't contain a `break`, `continue` or `return` statement; after all, any loop with those statements can be converted into a loop without them. (at least in principle, but don't do that in practice!)

In imperative programming, this makes sense, and it means telling the computer that while `<X>` is true, run `<Y>`, and `<Y>` might have side effects that do something useful. But in declarative programming, this doesn't make sense. For example, due to the properties above, if `<X>` is true, then it is true forever, and if it is false, then it is false forever. Thus, we can replace the code above with the following equivalent code:

[8] Furthermore, speaking in terms of complexity, we can also always do so with only a small penalty in running time and memory; think just a single log factor. This is challenging to state (and prove) precisely, so think of it as just an approximation/heuristic for now.

[9] The *scope* of a variable-value assignment is just the part of code where that assignment is still valid. For example, a variable defined in a function has its scope inside a call to that function.

[10] In the literature, this is called **referential transparency**.

```
if (<X>) {
    while (true) {
        <Y>
    }
} else {
    while (false) {
        <Y>
    }
}
```

But the second condition is just a *do nothing* operation. Also, remember that there are no side effects, so running a piece of code multiple times is the same as running it once, or even not at all. Hence, the above is equivalent to:

```
if (<X>) {
    while (true) {}
}
```

We can now see why this is a useless piece of code, assuming we don't want to get the TLE verdict.

Thus, there's no point in doing **while** loops, or loops of any kind whatsoever, because all loops can be converted to **while** loops.

Another consequence is that there's no point in declaring *arrays* anymore, since we will not allow modifying any of their elements, which is considered a side effect! We shall see later how to get around this.[11]

We shall see very soon that even with the limitations of not having side effects, we can still program pretty much anything.

3.2 Examples

3.2.1 Factorial

Let's begin programming!

Here's your first exercise: Given n, compute factorial(n), defined mathematically as:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{otherwise} \end{cases}$$

Note that this is a recursive definition, which is common in math.

If this were the IOI, you would probably be given a function signature and be asked to implement it, like this:

[11] The main thing we will use is **struct**s and **class**es, so to proceed, you need to learn **struct**s and **class**es. You can use http://training.noi.ph/topics/cpp_structs, http://training.noi.ph/topics/cpp_methods as reference. Actually, just try reading the C++ stuff in http://training.noi.ph/topics. Finally, ask away if you don't understand a particular piece of code, construct, or keyword.

```cpp
using ll = long long;
ll factorial(int n) {
    ... // write your solution here
}
```

(Disregard overflow issues for now. Also, we alias `long long` to `ll`.)

Surprisingly, we can program this declaratively very easily, by essentially copying the mathematical definition:

```cpp
ll factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

This is just a straightforward translation of the mathematical definition. Furthermore, it's not a coincidence that this is possible—many well-defined mathematical definitions can be translated directly into declarative definitions.

But more importantly, notice that our constraints aren't violated: there are no side effects, variable modifications, and global variables. Now, you might say that the variable `n` is being updated into `n - 1`, but this doesn't violate our constraints, since the new `n` is inside a *different* function call (and hence in a different scope), and that new `n` can't interact in any way with the `n` in the current call.

Crucially, this also means that we still allow recursion. As we shall see later on, we can rely on recursion to replace procedures where we traditionally use loops, and we will do so heavily.

3.2.2 Fibonacci numbers

Now that's over, here's your second exercise: Given n compute $\text{fib}(n)$, defined recursively as:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

Very easy!

```cpp
ll fib(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Again, this is a very straightforward translation of the mathematical definition.

However, here we run into our first issue with our paradigm: efficiency. It's not hard to see that the function above is horribly inefficient: try calling `fib(60)`.

Unfortunately, we can't use the traditional ways of optimizing this—memoization and dynamic programming—since those involves global states or arrays (and side effects), which we don't allow.

Luckily, we don't have to use those techniques in this particular case. We can define a new function `genfib(n, a, b)` which returns the nth Fibonacci number *assuming the first two terms are a and b*. To solve the function, we simply return `genfib(n, 0, 1)`.

```
ll genfib(int n, ll a, ll b) {
    if (n == 0)
        return a;
    else if (n == 1)
        return b;
    else
        return genfib(n - 1, b, a + b);
}
ll fib(int n) {
    return genfib(n, 0, 1);
}
```

Notice that `genfib` only requires a single recursive call to itself, making it run in $\mathcal{O}(n)$ time.[12]

Notice that by not reassigning variables, our variables and functions are basically the same as *mathematical* variables and functions. Also, in this case, programming doesn't consist of *telling the computer what to do* (imperative), rather, it consists of *telling the computer what is true* (declarative). In fact, to drive home this point, let's write the following line on top of our whole program:

```
#define is return
```

(In C++, `#define x <Y>` tells the compiler to replace any instance of `x` with `<Y>` before actually compiling the code.)

And from now on, we will use the new "keyword" `is` instead of `return`. Thus, we can rewrite our implementations above as:

```
ll factorial(int n) {
    if (n == 0)
        is 1;
    else
        is n * factorial(n - 1);
}

ll genfib(int n, ll a, ll b) {
    if (n == 0)
        is a;
    else if (n == 1)
        is b;
    else
```

[12] Whereas the original implementation runs in $\Theta(\phi^n)$ time where $\phi \approx 1.618$ is the golden ratio.

```
        is genfib(n - 1, b, a + b);
}
ll fib(int n) {
    is genfib(n, 0, 1);
}
```

Using `is` in place of `return` doesn't do anything special. We're just replacing a keyword, but it's still `return`. But it helps us—the programmers—in the *paradigm shift* that we're doing. Whereas `return` encourages you to read the `factorial` implementation imperatively, as the following:

- *To compute `factorial(n)`, do the following:*
  - *If `n == 0`, return the value `1`,*
  - *otherwise multiply `n` by `factorial(n - 1)` and return that value.*

with `is`, you should begin reading it declaratively, as the following:

- *`factorial(n)` is defined as:*
  - *`1` if `n == 0`,*
  - *`n` multiplied by `factorial(n - 1)` otherwise.*

In other words, we're declaring what `factorial(n)` is, not telling the computer how to compute `factorial(n)`. We're shifting from an imperative mindset to a declarative mindset.

We can also read our code more directly, though it results in slightly awkward English:

- *The `factorial` of `n`,*
  - *if `n == 0`, is `1`;*
  - *else, is `n` times `factorial(n - 1)`.*

3.2.3 Sorting

Now, let's move on to another nice example. Suppose you are given a list, and you're asked to compute the sorted version of that list. How do we solve it *declaratively*?

The first issue that we have is that we can't use arrays. Fortunately, we can use an alternative way of representing lists: *linked lists*.

We can define a list mathematically as: A *list* is either *null* (an empty list) or *a pair consisting of a value and a list* (the first value, and the list containing the rest of the elements). We can translate this straightforwardly using C++ structs this way:

```
struct List {
    int value;
    List* rest;  // a pointer to the rest of the list
    List(int value, List* rest): value(value), rest(rest) {}
};
```

And now, we can write `List*` whenever we need a list-like structure. For the empty list, we can use `NULL`.[13]

Note also that if we represent lists this way, we lose *random access*, i.e., getting the kth element doesn't take $\mathcal{O}(1)$ time anymore. But we shall accept this limitation for now. (We'll get around it later.)

We can now write the function signature for sorting as:

```
List* sorted(List* a) {
    // write your solution here
}
```

Note that we use the name "`sorted`" instead of "`sort`" because "`sort(x)`" is a command that says "You sort `x`!", while "`sorted(x)`" is a declaration that it is just the sorted version of `x`. It also signals that we don't expect `x` to be changed/updated at all.

Now, even disregarding efficiency, it seems like even implementing selection sort or insertion sort is hard. But let's think declaratively. How can we define sorting mathematically using insertion sort? Well, here goes:

- The sorted version of the empty list is the empty list.
- The sorted version of the list $(value, rest)$ is the *insertion* of *value* to the *sorted version* of *rest*.

Translating this into declarative code, we get:

```
List* insertion(int x, List* l) {
    ... // TODO
}
List* sorted(List* a) {
    if (a == NULL)
        is NULL;
    else
        is insertion(a->value, sorted(a->rest));
}
```

Now, we need to define `insertion` as well. Mathematically, we can do it as:

- The insertion of x into the empty list is the list containing only x.
- The insertion of x into the list $a = (value, rest)$, if $x \leq value$, is the list (x, a).
- The insertion of x into the list $a = (value, rest)$, if $x > value$, is the list $(value, newRest)$ where $newRest$ is the insertion of x into the list $rest$.

So the full implementation is now:

[13] Note that the constructor modified the variables `value` and `rest`, but this is another exception where we can use assignment: within the constructor, and only using this special syntax. Also, note that it's better to use `nullptr` instead of `NULL`, but we'll be using `NULL` because it's the word that appears in the mathematical definition.

```
List* insertion(int x, List* a) {
    if (a == NULL)
        is new List(x, NULL);
    else if (x <= a->value)
        is new List(x, a);
    else
        is new List(a->value, insertion(x, a->rest));
}
List* sorted(List* a) {
    if (a == NULL)
        is NULL;
    else
        is insertion(a->value, sorted(a->rest));
}
```

Now, we have a declarative sorting program! You can test that it works with the following imperative program.

```
List* take_list(int n) {
    if (n == 0)
        return NULL;
    else {
        int value;
        cin >> value;
        return new List(value, take_list(n - 1));
    }
}
void print_list(List* a) {
    if (a == NULL)
        cout << endl;
    else {
        cout << a->value << " ";
        print_list(a->rest);
    }
}
int main() {
    int n;
    cin >> n;
    List* a = take_list(n);
    List* b = sorted(a);

    // print result
    cout << "The sorted version of:\n";
    print_list(a);
    cout << "is:\n";
    print_list(b);
    cout << endl;
}
```

Note that we use the keyword **return** here to signify the fact that we're in an imperative paradigm. When I/O is involved, this is somewhat inevitable. But whenever we're doing declarative, we'll use **is**.

Now, insertion sort runs in $\mathcal{O}(n^2)$, which is slow. But there's nothing stopping us from implementing a declarative version of, say, merge sort.

To implement merge sort, we must define two auxiliary functions **split** and **merged**. Roughly

speaking, `split(x)` is a splitting of the list `x` into two lists of (approximately) equal length, and `merged(x, y)` is the combination of the two sorted lists `x` and `y` into a single sorted list.

One can easily define these mathematically (using recursion), and then implement them as follows:

```
pair<List*,List*> split(List* a) {
    if (a == NULL || a->rest == NULL)  // "|a| has at most one element"
        is {a, NULL};
    else {
        auto [list1, list2] = split(a->rest->rest);
        is {new List(a->value, list1), new List(a->rest->value, list2)};
    }
}
List* merged(List* a, List* b) {
    if (a == NULL)
        is b;
    else if (b == NULL)
        is a;
    else if (a->value <= b->value)
        is new List(a->value, merged(a->rest, b));
    else
        is new List(b->value, merged(a, b->rest));
}
List* sorted(List* a) {
    if (a == NULL || a->rest == NULL)  // "|a| has at most one element"
        is a;
    else {
        auto [list1, list2] = split(a);
        is merged(sorted(list1), sorted(list2));
    }
}
```

Here, we use C++'s `std::pair` class. Note that this is still coded declaratively, and there's a straightforward translation into a mathematical definition! (I encourage you to do so and define `split`, `merged` and `sorted` mathematically using the code above.)

Finally, we end this example with an implementation of (non-randomized) quicksort:

```
// concat(a, b) is the concatenation of lists a and b
List* concat(List* a, List* b) {
    if (a == NULL)
        is b;
    else
        is new List(a->value, concat(a->rest, b));
}
pair<List*,List*> partition(int value, List* a) {
    if (a == NULL)
        is {NULL, NULL};
    else {
        auto [smalls, bigs] = partition(value, a->rest);
        if (a->value <= value)
            is {new List(a->value, smalls), bigs};
        else
            is {smalls, new List(a->value, bigs)};
    }
}
List* sorted(List* a) {
    if (a == NULL || a->rest == NULL)
        is a;
```

```
    else {
        auto [smalls, bigs] = partition(a->value, a->rest);
        is concat(sorted(smalls), new List(a->value, sorted(bigs)));
    }
}
```

3.2.4 Useful list functions

Note that the expression `a == NULL || a->rest == NULL` tests if the length of the list is ≤ 1.

Since using `List*` is the primary way we will represent lists under the declarative paradigm, it would be nice if we can implement some useful functions in a list. This will also serve as good practice in using the declarative paradigm. We have already defined a useful function above, namely the `concat` function. What else can we define?

The first one is straightforward: given a list, what's its length? Well, how do you define the length of the list? We can do it recursively:

- The length of the empty list is 0.
- The length of the list $(value, rest)$ is 1 more than the length of $rest$.

Converting this to (declarative) code is straightforward:

```
int length(List* a) {
    if (a == NULL)
        is 0;
    else
        is 1 + length(a->rest);
}
```

We can also define a function that gives the kth element of a list. (Let's say zero-indexed)

```
#define error (throw "error")
int kth(int k, List* a) {
    if (a == NULL)
        error; // error: out of bounds!
    else if (k == 0)
        is a->value;
    else
        is kth(k - 1, a->rest);
}
```

Note that we raise an error whenever the index is too large.[14] Depending on your application, you may want return some other value here to signify out-of-bounds, say -1.

Here's an interesting exercise: given a list, compute its *reversal*. Using a straightforward mathematical definition, this can be done like this:

[14] Note that we can also type in `throw "error"` directly, but instead we aliased it as `error` so we don't see the word "throw," which is quite imperative.

```
List* reverse(List* a) {
    if (a == NULL)
        is NULL;
    else
        is concat(reverse(a->rest), new List(a->value, NULL));
}
```

Unfortunately, this takes $\mathcal{O}(\text{length}(a)^2)$ time, which is inefficient. Can you improve this?

Thankfully, yes! We can use an auxiliary function called `_reverse`:

```
List* _reverse(List* a, List* r) {
    if (a == NULL)
        is r;
    else
        is _reverse(a->rest, new List(a->value, r));
}
List* reverse(List* a) {
    is _reverse(a, NULL);
}
```

Now this runs in $\mathcal{O}(\text{length}(a))$ time![15]

> **Exercise 3.1.** How does `reverse(a)` work? What does `_reverse(a, r)` mean?

3.3 Automatic persistence

So far, we've been trying to implement some basic programming routines using the declarative paradigm, and so far, we see that we are still able to implement them even with the restriction of not having side effects. But by using this new paradigm, we gain two important properties:

- Because there are no state changes, it's much easier to prove the correctness of programs.
- More importantly, because we disallow modifying any state, no data structure ever gets destroyed or modified whatsoever, so **all our data structures are automatically persistent**!

For example, consider the following sequence of operations on a list:

```
List* x = new List(5, NULL);       // x is [5]
List* y = new List(7, x);          // y is [7, 5]
List* z = new List(3, y);          // z is [3, 7, 5]

List* a = new List(2, NULL);       // a is [2]
List* b = new List(10, a);         // b is [10, 2]
```

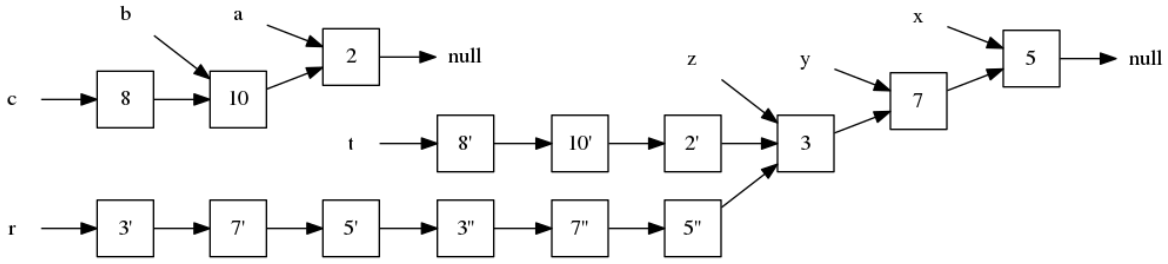
[15] At this point, it would be instructive to verify that all functions defined so far are declarative, i.e., there are no globals and no side effects. This includes the sorting algorithms in the previous part.

```
List* c = new List(8, b);          // c is [8, 10, 2]

List* t = concat(c, z);            // t is [8, 10, 2, 3, 7, 5]
List* r = concat(z, concat(z, z)); // r is [3, 7, 5, 3, 7, 5, 3, 7, 5]
```

Note that we have defined each list in terms of other lists. But crucially, even after doing so, the original lists are still accessible, and they can even still be operated upon! For example, we can add the line `List* d = new List(6, b);` anywhere after defining `b`, and this will define a new list `d` as `[6, 10, 2]` *without affecting all the other lists*! To be specific, all the lists defined above will still be valid and contain the same elements.[16] And this will remain true regardless of what we do with these lists afterwards, *as long as we stick to the declarative paradigm.* Also, since we don't allow reassignments of variables, it follows that these variables are *bound* to those lists forever.

Behind the scenes, these lists avoid complete copying for every operation by using *shared representation* in memory. For example, the lists above are represented in memory as:



Under the declarative paradigm, shared representation is fine, since the shared parts of the structures will never get modified.

3.3.1 A persistent stack

Using the declarative paradigm, we can now implement a *persistent* **stack**. Suppose we are given a problem of implementing a stack s, initially empty, with four operations:

- s**.push(**x**)**. Push x onto the stack.
- s**.top()**. Return the top of the stack (without removing it).
- s**.pop()**. Remove the top of the stack.
- s**.undo()**. Undo the last push or pop operation.

The traditional way to solve this problem efficiently, without **undo**s, is to use either a dynamic array or a linked list. Each operation can then be done in $\mathcal{O}(1)$ time. However, the **undo** operation complicates things a bit.

Now, there are ways to solve this problem efficiently using the imperative paradigm, but we will show how straightforward it is to solve in the declarative paradigm.

The first thing we need to do is to define a stack (no arrays allowed!). We can define it recursively as with a list. Alternatively, we can simply reuse our definition of list above.

[16] You can verify this by printing those lists at the end.

```
struct List {
    int value;
    List* rest;
    List(int value, List* rest): value(value), rest(rest) {}
};

class Stack {
    List* values;
public:
    Stack(): values(NULL) {}  // initially empty

    ... // the operations will be implemented here later
};
```

However, this only implements a stack of **int**s. What if we want to implement a stack of **double**s? Then we will have to copy-paste our code but replace the **int**s with **double**s. But this is not a good idea!

A better idea is to generalize. We want our list (and stack) to contain *any type*. We can implement it in C++ using *templates*:

```
template<class T>
struct List {
    T value;
    List<T>* rest;
    List(T value, List<T>* rest): value(value), rest(rest) {}
};

template<class T>
class Stack {
    List<T>* values;
public:
    Stack(): values(NULL) {}

    ... // the operations will be implemented here later
};
```

This is basically the same as before, except we're using a generalized class name `T` instead of **int**. For example, if we want a stack of **int**s, we write `Stack<int>`; if we want a stack of **double**s, we write `Stack<double>`, etc. Everything else stays the same.

Using this definition, we can already implement `push`, `top` and `pop` straightforwardly:

```
template<class T>
class Stack {
    List<T>* values;

    Stack(List<T>* values): values(values) {}

```

```
public:
    Stack(): values(NULL) {}

    bool is_empty() {
        is values == NULL;
    }

    Stack<T>* push(T value) {
        is new Stack<T>(new List<T>(value, values));
    }

    T top() {
        is values->value;
    }

    Stack<T>* pop() {
        is new Stack<T>(values->rest);
    }
};
```

Note that this stack implementation is *persistent*; the `push` and `pop` functions return the updated version of the stack but keep the original intact. Additionally, note that each operation still runs in $\mathcal{O}(1)$ each. Notice also that we have defined two constructors, one for an empty stack, and another for a stack containing a list of values as elements, but only the former constructor is made **`public`**. We have also defined a bonus function `is_empty`.

Take some time to read the implementation above using a declarative mindset; for example, in the `pop()` implementation, one could read it declaratively as: *the pop of a stack is the stack containing all the values except the first.*

Here's an example usage of the stack:

```
void print_stack(Stack<int>* stack) {  // this function is not declarative
    if (stack->is_empty()) {
        cout << endl;
    } else {
        cout << stack->top() << " ";
        print_stack(stack->pop());
    }
}

int main() {
    Stack<int>* stack = new Stack<int>();  // []
    Stack<int>* stack2 = stack->push(5);   // [5]
    Stack<int>* stack3 = stack2->push(7);  // [7, 5]
    Stack<int>* stack4 = stack3->push(3);  // [3, 7, 5]
    Stack<int>* stack5 = stack4->pop();    // [7, 5]
    Stack<int>* stack6 = stack5->pop();    // [5]
    Stack<int>* stack7 = stack6->pop();    // []
    Stack<int>* stack8 = stack5->push(8);  // [8, 7, 5]
    print_stack(stack);
```

```
    print_stack(stack2);
    print_stack(stack3);
    print_stack(stack4);
    print_stack(stack5);
    print_stack(stack6);
    print_stack(stack7);
    print_stack(stack8);
}
```

Notice that all versions of the stack are still intact, and can still be operated upon!

We can now exploit persistence to implement the `undo` operation. The idea is to keep a *stack of all versions of our stack*!

First, let's define a new class called `VersionedStack`, which is just a stack but keeps all its previous versions. We can define it as a pair consisting of a stack s and another stack containing *all versions of* s:

```
template<class T>
class VersionedStack {
    Stack<T>* stack;
    Stack<Stack<T>*>* versions;  // a stack of stacks

    ... //
};
```

Now, we can implement all four operations, including `undo`!

```
template<class T>
class VersionedStack {
    Stack<T>* current;
    Stack<Stack<T>*>* versions;

    VersionedStack(Stack<T>* current, Stack<Stack<T>*>* versions):
            current(current), versions(versions) {}

public:
    VersionedStack(): current(new Stack<T>()), versions(new Stack<Stack<T>*>()) {}

    bool is_empty() {
        is current->is_empty();
    }

    VersionedStack<T>* push(T value) {
        is new VersionedStack<T>(current->push(value), versions->push(current));
    }

    T top() {
        is current->top();
    }

    VersionedStack<T>* pop() {
        is new VersionedStack<T>(current->pop(), versions->push(current));
    }

    VersionedStack<T>* undo() {
```

```
        is new VersionedStack<T>(versions->top(), versions->pop());
    }
};
```

Again, we create two constructors, but only make one public.

We can now test this new stack class by replacing `Stack<T>` with `VersionedStack<T>`.

```
void print_stack(VersionedStack<int>* stack) {  // this function is not declarative
    if (stack->is_empty())
        cout << endl;
    else {
        cout << stack->top() << " ";
        print_stack(stack->pop());
    }
}

int main() {
    VersionedStack<int>* stack = new VersionedStack<int>(); // []
    VersionedStack<int>* stack2 = stack->push(5);           // [5]
    VersionedStack<int>* stack3 = stack2->push(7);          // [7, 5]
    VersionedStack<int>* stack4 = stack3->push(3);          // [3, 7, 5]
    VersionedStack<int>* stack5 = stack4->pop();            // [7, 5]
    VersionedStack<int>* stack6 = stack5->pop();            // [5]
    VersionedStack<int>* stack7 = stack6->pop();            // []
    VersionedStack<int>* stack8 = stack5->push(8);          // [8, 7, 5]
    VersionedStack<int>* stack9 = stack6->undo();           // [7, 5]
    VersionedStack<int>* stack10 = stack9->undo();          // [3, 7, 5]
    VersionedStack<int>* stack11 = stack9->push(11);        // [11, 7, 5]
    print_stack(stack);
    print_stack(stack2);
    print_stack(stack3);
    print_stack(stack4);
    print_stack(stack5);
    print_stack(stack6);
    print_stack(stack7);
    print_stack(stack8);
    print_stack(stack9);
    print_stack(stack10);
    print_stack(stack11);
}
```

As you can see, all operations work properly, all versions are still available, and each operation still runs in $\mathcal{O}(1)$, all by sticking to the declarative paradigm!

3.3.2 A persistent binary tree

We used the simple *stack* problem above to illustrate how declarative programming is used to create a persistent version of a stack, but there's no reason to stop there.

For example, we can also define a binary tree declaratively. Mathematically, we can define a **binary tree** as either *null* (an empty tree) or *a triple consisting of a value and two binary trees* (the value at the node and the left and right subtrees, respectively). This can be translated straightforwardly as:

```
struct Tree {
    int value;
    Tree* left;
    Tree* right;
    Tree(int value, Tree* left, Tree* right): value(value), left(left),
    ↪  right(right) {}
};
```

One can also define a generalized version using templates like this:

```
template<class T>
struct Tree {
    T value;
    Tree<T>* left;
    Tree<T>* right;
    Tree(T value, Tree<T>* left, Tree<T>* right): value(value), left(left),
    ↪  right(right) {}
};
```

But in this section, we'll stick to a version with just `int`s for simplicity.

Now, if we want, we can turn this into a binary *search* tree by implementing the *insert* operation. Mathematically, we can define the insertion of a value *value* into a tree *tree* recursively as:

- The insertion of x into the empty tree is the tree containing only x, with a null left and right subtree, i.e., the tree $(x, null, null)$.
- The insertion of x into the tree $(value, left, right)$, if $x < value$, is the tree obtained by inserting x into the left subtree, i.e., $(value, \text{insert}(left, x), right)$.
- The insertion of x into the tree $(value, left, right)$, if $x \geq value$, is the tree obtained by inserting x into the right subtree, i.e., $(value, left, \text{insert}(right, x))$.

Converting (straightforwardly) into declarative code, we get:

```
Tree* insert(Tree* tree, int x) {
    if (tree == NULL)
        is new Tree(x, NULL, NULL);
    else if (x < tree->value)
        is new Tree(tree->value, insert(tree->left, x), tree->right);
    else
        is new Tree(tree->value, tree->left, insert(tree->right, x));
};
```

We can also define a function which checks if a tree contains a given value:

```
bool contains(Tree* tree, int x) {
    if (tree == NULL)
        is false;
    else if (x == tree->value)
```

```
        is true;
    else if (x < tree->value)
        is contains(tree->left, x);
    else
        is contains(tree->right, x);
}
```

And now, we get a persistent binary search tree! Here's a sample usage which illustrates that it's indeed persistent:

```
int main() {
    Tree* a = NULL;
    Tree* b = insert(a, 4);  // {4}
    Tree* c = insert(b, 6);  // {4, 6}
    Tree* d = insert(c, 2);  // {2, 4, 6}
    Tree* e = insert(d, 9);  // {2, 4, 6, 9}
    Tree* f = insert(c, 8);  // {4, 6, 8}

    cout << contains(d, 8) << endl;  // false
    cout << contains(e, 8) << endl;  // false
    cout << contains(f, 8) << endl;  // true
    cout << contains(d, 9) << endl;  // false
    cout << contains(e, 9) << endl;  // true
    cout << contains(f, 9) << endl;  // false
}
```

Of course, this implementation takes $\mathcal{O}(n)$ per operation in the worst case, since it's possible for our binary tree to degenerate into a line on specifically chosen inputs, but this is no problem: there's nothing stopping us from using our favorite balancing rule for binary search trees!

For example, we can use the AVL tree rules of balancing.[17] With, AVL trees, we want to store the height of each subtree,[18] so we redefine our `Tree` struct:

```
struct AVLTree {
    int value;
    int height;
    AVLTree* left;
    AVLTree* right;
    AVLTree(int value, int height, AVLTree* left, AVLTree* right):
    ↪  value(value), height(height), left(left), right(right) {}
};
```

The functions will change as well. For example, will `insert` look very similar, but now, instead of calling the tree constructor directly, we will use an auxiliary `balanced_combine` function which does the rebalancing (according to the AVL tree rules). The others only change slightly.

[17] Read more about AVL trees here: https://en.wikipedia.org/wiki/AVL_tree .
[18] at least in the simplest implementations

```
AVLTree* balanced_combine(int value, AVLTree* left, AVLTree* right) {
    ... // combines trees with rebalancing
}
AVLTree* insert(AVLTree* tree, int x) {
    if (tree == NULL)
        is new AVLTree(x, 1, NULL, NULL);
    else if (x < tree->value)
        is balanced_combine(tree->value, insert(tree->left, x), tree->right);
    else
        is balanced_combine(tree->value, tree->left, insert(tree->right, x));
};

bool contains(AVLTree* tree, int x) {
    if (tree == NULL)
        is false;
    else if (x == tree->value)
        is true;
    else if (x < tree->value)
        is contains(tree->left, x);
    else
        is contains(tree->right, x);
}
```

As always, here's some testing code:

```
// testing code. (not declarative)
int main() {
    AVLTree* a = NULL;
    AVLTree* b = insert(a, 4);  // {4}
    AVLTree* c = insert(b, 6);  // {4, 6}
    AVLTree* d = insert(c, 2);  // {2, 4, 6}
    AVLTree* e = insert(d, 9);  // {2, 4, 6, 9}
    AVLTree* f = insert(c, 8);  // {4, 6, 8}

    cout << contains(d, 8) << endl;  // false
    cout << contains(e, 8) << endl;  // false
    cout << contains(f, 8) << endl;  // true
    cout << contains(d, 9) << endl;  // false
    cout << contains(e, 9) << endl;  // true
    cout << contains(f, 9) << endl;  // false
}
```

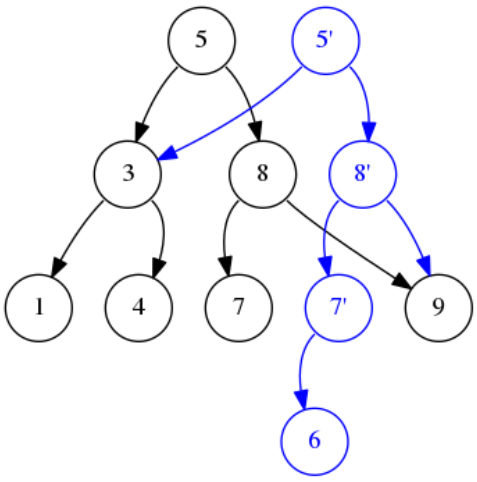
But how do we implement `balanced_combine`? Well, we can do so by simply translating the AVL tree rules and handling all cases! The implementation is quite long, so we will not show it here. But to be honest, there's nothing really surprising about it. If you really want to see it, check out Appendix A.

By using the AVL tree rules to balance our tree, we now have a fully persistent self-balancing binary search tree that runs in $\mathcal{O}(\log n)$ per operation in the worst case![19]

Behind the scenes, self-balancing binary search trees are able to become persistent while retaining their $\mathcal{O}(\log n)$ per-operation running time by using *shared representation*, like with lists. The key here is noticing that only $\mathcal{O}(\log n)$ nodes need to be updated in the first place; everything else can stay the same (and since we're coding declaratively, they *will* stay the same).

[19] One can also use the red-black tree rules of rebalancing; it's up to you what you want to use.

For example, here's how a new tree gets created from an existing tree. Assume that 6 is inserted. The blue nodes are the newly-created nodes.



As you can clearly see, the old version of the tree still remains! Insertion simply reuses huge chunks of the previous tree to create the new tree. And since we're coding declaratively, we're guaranteed that these structures will stay this way. This is how efficiency is maintained even with persistence.

3.3.3 A persistent array

At the beginning of this lesson, we have decided that we are not allowed to use arrays since they need side effects for them to be useful. Well, it turns out that it's possible to implement an efficient, *fully persistent* array! More precisely, speaking in terms of data structures, we have only disallowed the use of the **array data structure**, but not the implementation of the **array abstract data type**, which we're about to do here.

The key is to represent the array ADT as perfect binary trees. With this idea, we can try to define an *array on the indices* $[i, j]$ mathematically as follows:

- An array on the indices $[i, j]$ is *null* (an empty array) if $i > j$.
- An array on the indices $[i, j]$ is a 4-tuple of values: an index k, the value at index k, an array on the indices $[i, k-1]$ and an array on the indices $[k+1, j]$, where $k = \lfloor \frac{i+j}{2} \rfloor$.

In code, it looks like:

```
template<class T>
class Array {
    int k;
    T value;
    Array<T>* left;
    Array<T>* right;
    Array(int k, T value, Array<T>* left, Array<T>* right): k(k),
    ↪  value(value), left(left), right(right) {}

```

```
    ... //
};
```

We can now define a (declarative) function that returns a persistent array on a certain range of indices. It's also simple to define the modification and access functions declaratively:

```
template<class T>
class Array {
    int k;
    T value;
    Array<T>* left;
    Array<T>* right;
    Array(int k, T value, Array<T>* left, Array<T>* right): k(k), value(value), left(left),
    ↪  right(right) {}

public:
    static Array<T>* create(int i, int j, T initial) {
        if (i > j)
            is NULL;
        else {
            int k = (i + j) / 2;
            is new Array<T>(k, initial,
                    create(i, k-1, initial),
                    create(k+1, j, initial));
        }
    }

    T get(int i) {
        if (i == k)
            is value;
        else if (i < k)
            is left->get(i);
        else
            is right->get(i);
    }

    Array<T>* set(int i, T new_value) {
        if (i == k)
            is new Array<T>(k, new_value, left, right);
        else if (i < k)
            is new Array<T>(k, value, left->set(i, new_value), right);
        else
            is new Array<T>(k, value, left, right->set(i, new_value));
    }
};
```

(The **static** keyword means that the function is not bound to a class instance. So in the example above, **create** is basically just a normal function, not a member function of **Array**s, except that we wanted to put it inside the class definition anyway because it *logically* should be there.)

This implementation assumes that the indices don't go out of bounds; otherwise the behavior is undefined.

One can now check that this is works and is fully persistent:

```
void print_array(Array<int>* arr, int i, int j) {  // not declarative
```

```
    for (int k = i; k <= j; k++) {
        cout << arr->get(k) << " ";
    }
    cout << endl;
}

int main() {
    Array<int>* a = Array<int>::create(1, 20, 0);

    Array<int>* b = a->set(5, 700);
    Array<int>* c = b->set(11, 200);
    Array<int>* d = c->set(5, 300);
    Array<int>* e = b->set(11, 900);
    Array<int>* f = c->set(7, 800);

    print_array(a, 1, 20);
    print_array(b, 1, 20);
    print_array(c, 1, 20);
    print_array(d, 1, 20);
    print_array(e, 1, 20);
    print_array(f, 1, 20);
}
```

Unfortunately, our array requires $\mathcal{O}(\log n)$ time for access and modification, which is worse than $\mathcal{O}(1)$; and we will have to live with this limitation. But don't be sad; the gains we get (having automatic persistence, etc.) are very much worth that log factor, aren't they?

As a side note, one can also define a dynamic array, i.e., an array that can resize, by using a self-balancing binary tree instead of a fixed tree.

3.3.4 A persistent segment tree

We are now ready to implement a fully persistent segment tree. Actually, it's not much different from what we've been doing before.

Let's begin with a simple variant: consider an array of n elements indexed from 0 to $n-1$, initially all 0s. We need to implement the following operations:

- **.set**$(I,\ v)$. Set the value at index I to v.
- **.rangemax**$(I,\ J)$. Return the largest value among all elements whose index is in the range $[I, J]$.
- **.revert**(x). Revert the state of the segment tree to the version right after the xth **set** or **revert** operation. Note that this also means that *one can revert a revert operation*!

For simplicity, assume that all operations will be valid.

The first step is to define a simple (declarative) segment tree which allows for *range max* queries. We can define it very similarly to our persistent array, except that we must keep track of the maximum at each range.

We will represent a node of a segment tree as a 5-tuple $(i, j, \mathit{value}, \mathit{left}, \mathit{right})$, where $[i, j]$ represent the range of keys, *value* is the maximum on that range, and *left* and *right* are the left and right subtrees, respectively.

Just like before, we can mathematically define *a segment tree on the keys* $[i, j]$ as follows:

- A segment tree on the keys $[i, j]$, if $i = j$, is the 5-tuple $(i, j, value, null, null)$.
- A segment tree on the keys $[i, j]$, if $i < j$, is the 5-tuple $(i, j, value, left, right)$ where *left* and *right* are segment trees on the keys $[i, k]$ and $[k + 1, j]$, respectively, where $k = \lfloor \frac{i+j}{2} \rfloor$.

We can write this declaratively as:

```
class Segtree {
    int i, j;
    int value;
    Segtree* left;
    Segtree* right;
    Segtree(int i, int j, int value, Segtree* left, Segtree* right): i(i), j(j), value(value),
    ↪  left(left), right(right) {}

    ... //
};
```

Let's now write a function that constructs a segment tree on a given range of keys, with initial value 0, based on the definition. The operations can also be implemented straightforwardly:

```
const int INF = 1<<30;  // some really large number

class Segtree {
    int i, j;
    int value;
    Segtree* left;
    Segtree* right;
    Segtree(int i, int j, int value, Segtree* left, Segtree* right): i(i), j(j), value(value),
    ↪  left(left), right(right) {}

    static Segtree* combine(Segtree* left, Segtree* right) {
        is new Segtree(left->i, right->j, max(left->value, right->value), left, right);
    }

public:
    static Segtree* create(int i, int j) {
        if (i == j)
            is new Segtree(i, j, 0, NULL, NULL);
        else {
            int k = (i + j) / 2;
            is combine(create(i, k), create(k+1, j));
        }
    }

    Segtree* set(int I, int v) {
        if (i <= I && I <= j) {  // needs modification
            if (left == NULL)
                is new Segtree(i, j, v, NULL, NULL);
            else
                is combine(left->set(I, v), right->set(I, v));
        } else
            is this;  // no modification
    }

    int rangemax(int I, int J) {
```

```
        if (I <= i && j <= J)  // completely contained
            is value;
        else if (J < i || j < I)  // disjoint
            is -INF;
        else  // partially overlaps
            is max(left->rangemax(I, J), right->rangemax(I, J));
    }
};
```

Now, we have a fully persistent segment tree, with $\mathcal{O}(\log n)$ per operation! To check its persistence, we can use the following code:

```
int main() {
    Segtree* a = Segtree::create(0, 99);
    Segtree* b = a->set(3, 100);
    Segtree* c = b->set(7, 200);
    Segtree* d = b->set(5, 150);

    cout << a->rangemax(3, 7) << endl;  // 0
    cout << b->rangemax(3, 7) << endl;  // 100
    cout << c->rangemax(3, 7) << endl;  // 200
    cout << d->rangemax(3, 7) << endl;  // 150

    cout << a->rangemax(4, 7) << endl;  // 0
    cout << b->rangemax(4, 7) << endl;  // 0
    cout << c->rangemax(4, 7) << endl;  // 200
    cout << d->rangemax(4, 7) << endl;  // 150

    cout << a->rangemax(3, 6) << endl;  // 0
    cout << b->rangemax(3, 6) << endl;  // 100
    cout << c->rangemax(3, 6) << endl;  // 100
    cout << d->rangemax(3, 6) << endl;  // 150
}
```

Now, all we have to do is to implement **revert**(x) efficiently.

The first thought is to use the same technique as with the *undoable* stack—keep a stack of all versions—but there is a problem with this: it's possible for the required version to be very old, which means we might have to pop our version stack a lot of times. Additionally, **revert**(x) is considered to be an operation in its own right, so we need to append the retrieved version again in our list of versions. This makes the running time of **revert**(x) slow![20]

The solution is to use a (persistent) array! Instead of using a stack to keep our history, we will use an array. We will also keep track of the number of versions of our segment tree so far.

Here's the implementation, including `revert`:

```
class VersionedSegtree {
    Segtree* curr;
    Array<Segtree*>* versions;
    int currv;

```

[20] If revert is not considered to be an operation, then actually just using a stack and repeatedly popping is okay and takes $\mathcal{O}(1)$ amortized time per revert. (Why?)

```
    VersionedSegtree(Segtree* curr, Array<Segtree*>* versions, int currv) :
            curr(curr), versions(versions), currv(currv) {}

    VersionedSegtree* push_version(Segtree* newcurr) {
        is new VersionedSegtree(
                newcurr,
                versions->set(currv + 1, newcurr),
                currv + 1
            );
    }

public:
    static VersionedSegtree* on_segtree(Segtree* curr, int opmax) {
        Array<Segtree*>* versions = Array<Segtree*>::create(0, opmax, NULL);
        is new VersionedSegtree(curr, versions->set(0, curr), 0);
    }

    VersionedSegtree* set(int I, int v) {
        is push_version(curr->set(I, v));
    }

    int rangemax(int I, int J) {
        is curr->rangemax(I, J);
    }

    VersionedSegtree* revert(int x) {
        is push_version(versions->get(x));
    }
};
```

In this implementation, `opmax` represents the maximum number of operations to be done on the segment tree, and is used to determine the size of the version list.

We can now test this using the following (imperative) code:

```
int main() {
    VersionedSegtree* a = VersionedSegtree::on_segtree(Segtree::create(0, 99), 10000);

    cout << a->rangemax(2, 6) << endl;  // 0
    cout << a->rangemax(3, 7) << endl;  // 0
    cout << a->rangemax(4, 8) << endl;  // 0

    a = a->set(2, 1000);
    cout << a->rangemax(2, 6) << endl;  // 1000
    cout << a->rangemax(3, 7) << endl;  // 0
    cout << a->rangemax(4, 8) << endl;  // 0

    a = a->set(5, 100);
    cout << a->rangemax(2, 6) << endl;  // 1000
    cout << a->rangemax(3, 7) << endl;  // 100
    cout << a->rangemax(4, 8) << endl;  // 100

    a = a->set(8, 800);
    cout << a->rangemax(2, 6) << endl;  // 1000
    cout << a->rangemax(3, 7) << endl;  // 100
    cout << a->rangemax(4, 8) << endl;  // 800

    a = a->set(6, 1600);
    cout << a->rangemax(2, 6) << endl;  // 1600
    cout << a->rangemax(3, 7) << endl;  // 1600
    cout << a->rangemax(4, 8) << endl;  // 1600
```

```
27
28      a = a->revert(2);
29      cout << a->rangemax(2, 6) << endl;  // 1000
30      cout << a->rangemax(3, 7) << endl;  // 100
31      cout << a->rangemax(4, 8) << endl;  // 100
32
33      a = a->set(3, 700);
34      cout << a->rangemax(2, 6) << endl;  // 1000
35      cout << a->rangemax(3, 7) << endl;  // 700
36      cout << a->rangemax(4, 8) << endl;  // 100
37
38      a = a->revert(4);
39      cout << a->rangemax(2, 6) << endl;  // 1600
40      cout << a->rangemax(3, 7) << endl;  // 1600
41      cout << a->rangemax(4, 8) << endl;  // 1600
42
43      a = a->revert(5);
44      cout << a->rangemax(2, 6) << endl;  // 1000
45      cout << a->rangemax(3, 7) << endl;  // 100
46      cout << a->rangemax(4, 8) << endl;  // 100
47  }
```

We can see that it's working properly! Now we have a segment tree with all the operations, including `revert`, running in $\mathcal{O}(\log n)$ time each!

One limitation of this implementation is that we have to know `opmax` (the maximum number of operations) beforehand. However, this is a minor issue, since in most problems, this will be true anyway. Also, as we shall see later, we can get around that limitation with the use of *infinite arrays*. Yes, you read it right: infinite.

By the way, it's also possible to implement a persistent segment tree *with lazy propagation*. I will assume you know how to implement a normal segment tree with lazy propagation, and I will leave it to you to implement a persistent version. Just think declaratively!

3.3.5 Persistent union-find

Now, what about a persistent union-find?

Actually, there's a pretty simple and straightforward way of doing this that requires only a little thought. Remember that normally, one would implement union-find on the values $\{0, \ldots, n-1\}$ using 2 arrays of length n called `parent` and `rank`.[21] Well, to make it persistent, we can simply replace the arrays with *persistent* arrays!

```
1   class UnionFind {
2       Array<int>* parent;
3       Array<int>* rank;
4       UnionFind(Array<int>* parent, Array<int>* rank): parent(parent), rank(rank) {}
5
6   public:
7       static UnionFind* create(int n) {
8           is new UnionFind(Array<int>::create(0, n-1, -1), Array<int>::create(0, n-1, 0));
9       }
10
11      int find(int i) {
12          int j = parent->get(i);
13          is j == -1 ? i : find(j);
14      }
```

[21] Or maybe just one array if you're treating negative values as negative ranks.

```
15
16      UnionFind* onion(int i, int j) {
17          int fi = find(i);
18          int fj = find(j);
19          if (fi == fj)
20              is this;  // no change needed
21          else {
22              int ri = rank->get(fi);
23              int rj = rank->get(fj);
24              if (ri < rj)
25                  is new UnionFind(parent->set(fi, fj), rank);
26              else if (ri > rj)
27                  is new UnionFind(parent->set(fj, fi), rank);
28              else
29                  is new UnionFind(parent->set(fi, fj), rank->set(fj, rj + 1));
30          }
31      }
32  };
```

That's it! We can now test that this works and this is persistent using the following (imperative) code:

```
1   int main() {
2       UnionFind* x = UnionFind::create(11);
3       UnionFind* y = x->onion(3, 4);
4       UnionFind* z = y->onion(2, 4);
5       UnionFind* w = z->onion(3, 7);
6       UnionFind* v = y->onion(5, 6);
7
8       cout << (x->find(2) == x->find(7)) << endl;  // false
9       cout << (y->find(2) == y->find(7)) << endl;  // false
10      cout << (z->find(2) == z->find(7)) << endl;  // false
11      cout << (w->find(2) == w->find(7)) << endl;  // true
12      cout << (v->find(2) == v->find(7)) << endl;  // false
13      cout << (w->find(5) == w->find(6)) << endl;  // false
14      cout << (v->find(5) == v->find(6)) << endl;  // true
15  }
```

In fact, many traditionally imperative structures can be turned persistent by simply replacing the arrays with persistent arrays. (At the cost of a log factor.)

Unfortunately, we couldn't perform *path compression* in this implementation since it involves modifying the structure on access, which we don't want to do in a declarative setting. Therefore, the implementation above runs in $\mathcal{O}(\log^2 n)$ time per operation. Keep this in mind.[22]

3.4 Lazy data structures

Another thing about declarative programming we haven't discussed yet is the concept of *lazy evaluation*. You probably have seen glimpses of this idea when implementing segment trees with *lazy* propagation. However, in many *purely declarative* programming languages, **everything is lazily evaluated**. As we shall see later, this property allows us to define infinite data structures on such programming languages.

[22] There's actually a different approach to implementing a persistent union-find that achieves a better $\mathcal{O}(\log n)$ time per operation, but we won't discuss it here.

Roughly speaking, lazy evaluation means that an expression is only evaluated when its value is needed. For example, consider the following "function" which, given n, attempts to construct the infinite list $[n, n+1, n+2, \ldots]$:

```
List* count(int n) {
    is new List(n, count(n + 1));
}
```

Obviously, calling `count` on any argument results in an infinite loop. However, the equivalent code in a purely declarative language will actually work, since anything is only really evaluated when it is needed! For instance, in the above definition, the expression `count(n + 1)` will only be evaluated if the `List`'s `rest` member is accessed by the program. In other words, the evaluation is suspended until the result is actually required.

For example, in a lazily-evaluated programming language, the following sequence of operations will be valid:

```
int main() {
    List* natural = count(1);
    cout << natural->value << endl;  // 1
    cout << natural->rest->value << endl;  // 2
    cout << natural->rest->rest->value << endl;  // 3
    cout << natural->rest->rest->rest->value << endl;  // 4
}
```

And this will not result in an infinite loop. In fact, the function call `count(5)` will not be evaluated at all!

The takeaway here is that in a purely declarative language, it's possible to create *infinite* lists!

In fact, using an infinite list like this, we can write the following "solution" to the NOI.PH 2017 eliminations round problem "Pak Ganern":[23]

```
template<class T>
List<T>* concat(List<T>* a, List<T>* b) {
    if (a == NULL)
        is b;
    else
        is new List<T>(a->value, concat(a->rest, b));
}

// takes the first n elements of list a
template<class T>
List<T>* take(int n, List<T>* a) {
    if (n == 0)
        is NULL;
    else
        is new List<T>(a->value, take(n - 1, a->rest));
}

// returns the list [value, ..., value] where value is repeated k times
template<class T>
```

[23] https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/pak-ganern

```
List<T>* replicate(int k, T value) {
    if (k == 0)
        is NULL;
    else
        is new List<T>(value, replicate(k - 1, value));
}

string pak("PAK"), ganern("GANERN");
List<string>* pak_ganern_sequence(int k) {
    is  concat(replicate(k, pak),
        concat(replicate(k, ganern),
        pak_ganern_sequence(k + 1)));
}

// non-declarative part
void print_all(List<string>* s) {
    if (s != NULL) {
        cout << s->value << endl;
        print_all(s->rest);
    }
}
int main() {
    int n;
    cin >> n;
    print_all(take(n, pak_ganern_sequence(1)));
}
```

The idea here is that the function `pak_ganern_sequence` generates the infinite pak ganern sequence, but the program only prints the first n elements of it.

Obviously, this code doesn't work in C++ since C++'s evaluation strategy is not lazy evaluation, but *strict evaluation*. Thus, the above will enter an infinite loop. Which of course sucks. Fortunately, we can still *simulate* lazy evaluation if we code things right. In this section, we will discuss ways on doing so.

Just to illustrate that it's really working for lazily-evaluated languages, here's a working solution to the Pak Ganern problem in Haskell (a language with lazy evaluation):

```
pakGanernSequence k =
    replicate k "PAK" ++
    replicate k "GANERN" ++
    pakGanernSequence (k + 1)

solve n = take n (pakGanernSequence 1)

main = do
    s <- getLine
    let n = read s
    mapM_ putStrLn (solve n)
```

Note that `pakGanernSequence` is indeed an infinite list. And it's really working![24]

3.4.1 Lazy evaluation in Python

(You may skip this small subsection if you don't know Python.)

As a side note, Python in fact has some support for lazy evaluation in the form of *generators*

[24] Obviously, you can't submit it since Haskell is not allowed in the contest. But you can check that it works and that it doesn't enter an infinite loop by running it offline. Note that **`take`** and **`replicate`** are Haskell built-ins, and **`concat`** is also a built-in and is written with the operator `++`.

and *iterators*. For example, the following is a valid solution to Pak Ganern:

```python
from itertools import count, islice
def pak_ganern_sequence():
    for c in count(1):
        for i in range(c): yield 'PAK'
        for i in range(c): yield 'GANERN'

n = int(input())
for word in islice(pak_ganern_sequence(), n):
    print(word)
```

In this code, `count` (from Python's `itertools` package) does precisely what the `count` function above intends to do: generate the infinite list $[n, n+1, n+2, \ldots]$. However, Python genuinely does this *lazily*, so this doesn't go into an infinite loop. This also illustrates how to write code with lazy evaluation in Python through the use of the `yield` keyword: here, `pak_ganern_sequence()` genuinely generates the *infinite pak ganern sequence*, but the imperative part of the code only prints the first n of them. Python only evaluates a function until it encounters the `yield` keyword; thereafter, it stops and returns until the next value is being requested again. Python's `islice` function is the equivalent of our `take` function above.

3.4.2 An infinite list

Let's start with a simple example. Suppose we want to implement `count(n)` properly in C++. Well, we can simulate lazy evaluation by simply initializing the `rest` member to *null* and only constructing it when it's accessed. In other words, we will enforce the suspension of evaluation.

We can do it like this, for example:

```cpp
class Count {
    Count* _rest;
public:
    int value;
    Count(int value): value(value), _rest(NULL) {}

    Count* rest() {
        if (_rest == NULL) _rest = new Count(value + 1);  // evaluate lazily
        is _rest;
    }
};
```

We can check that this works with our (slightly modified) imperative code:

```cpp
int main() {
    Count* natural = new Count(1);
    cout << natural->value << endl;  // 1
    cout << natural->rest()->value << endl;  // 2
    cout << natural->rest()->rest()->value << endl;  // 3
```

```
    cout << natural->rest()->rest()->rest()->value << endl;  // 4
}
```

It works, i.e., it doesn't go into an infinite loop. Therefore, we have created our first infinite list in C++!

Unfortunately, implementing it this way, while pretty simple, has some drawbacks:

- We have to write `->rest()` instead of just `->rest`, which is longer and uglier.
- The code above is not purely declarative since we're reassigning the `_rest` member under the `rest()` function. Unfortunately, there seems no way to get around this, so we have to slightly break our rules to be able to simulate lazy evaluation.
- We can't pass a `Count*` object in existing functions that take `List*`s as input since it is a different class.

These make the possible applications of this list somewhat limited in C++. Nevertheless, it's a good start to implementing infinite data structures in C++! Also, this would probably still be helpful in some cases.

Exercise 3.2. Convert the *pak ganern* solution above into a working C++ code that uses an infinite lazy list.

3.4.3 An infinite array

Now, let's tackle how we can implement an infinite array. By this, we mean that we want to implement an array data structure on the range of indices $[0, \infty)$.

It's possible to do this using self-balancing binary search trees so that each operation (access and modification) takes $\mathcal{O}(\log n)$ time. However, there's in fact a simpler way using lazy evaluation. The idea is to use an *infinite binary tree* to represent the array. The root will represent the index 0, and the left and right subtree will represent the positive odd and even indices, respectively!

Now, if C++ had lazy evaluation, we can easily implement this with the following:

```
template<class T>
class InfArray {
    T zero;
    InfArray<T>* odd;
    InfArray<T>* even;
    InfArray(T zero, InfArray<T>* odd, InfArray<T>* even): zero(zero), odd(odd), even(even) {}

public:
    static InfArray<T>* create() {
        is new InfArray<T>(T(), create(), create());
    }

    T get(int i) {
        if (i == 0)
            is zero;
        else if (i % 2)
            is odd->get((i - 1) / 2);
        else
            is even->get((i - 1) / 2);
```

```
    }

    InfArray<T>* set(int i, T v) {
        if (i == 0)
            is new InfArray<T>(v, odd, even);
        else if (i % 2)
            is new InfArray<T>(zero, odd->set((i - 1) / 2, v), even);
        else
            is new InfArray<T>(zero, odd, even->set((i - 1) / 2, v));
    }
};
```

The trick here is in the definition of `create()`, which recursively calls `create()` twice to construct the left and right subtrees. Unfortunately, without lazy evaluation, this will enter an infinite loop (or the program runs out of memory). To make this work in C++, we need to simulate lazy evaluation, like before:

```
template<class T>
class InfArray {
    T zero;
    InfArray<T>* _odd;
    InfArray<T>* _even;
    InfArray(T zero, InfArray<T>* _odd, InfArray<T>* _even): zero(zero), _odd(_odd),
    ↪  _even(_even) {}

    InfArray<T>* odd() {
        if (_odd == NULL) _odd = create();
        is _odd;
    }

    InfArray<T>* even() {
        if (_even == NULL) _even = create();
        is _even;
    }

public:
    static InfArray<T>* create() {
        is new InfArray<T>(T(), NULL, NULL);
    }

    T get(int i) {
        if (i == 0)
            is zero;
        else if (i % 2)
            is odd()->get((i - 1) / 2);
        else
            is even()->get((i - 1) / 2);
    }

    InfArray<T>* set(int i, T v) {
        if (i == 0)
            is new InfArray<T>(v, odd(), even());
        else if (i % 2)
            is new InfArray<T>(zero, odd()->set((i - 1) / 2, v), even());
        else
            is new InfArray<T>(zero, odd(), even()->set((i - 1) / 2, v));
    }
};
```

Now, by moving the creation of the subtrees into the `odd()` and `even()` getter functions, we are now only creating them when they are actually needed. This means we now have a persistent

infinite array! Here's some testing code:

```
void print_array(InfArray<int>* arr, int i, int j) {  // not declarative
    for (int k = i; k <= j; k++) {
        cout << arr->get(k) << " ";
    }
    cout << endl;
}

int main() {
    InfArray<int>* a = InfArray<int>::create();

    InfArray<int>* b = a->set(5, 700);
    InfArray<int>* c = b->set(11, 200);
    InfArray<int>* d = c->set(5, 300);
    InfArray<int>* e = b->set(11, 900);
    InfArray<int>* f = c->set(7, 800);

    print_array(a, 1, 20);
    print_array(b, 1, 20);
    print_array(c, 1, 20);
    print_array(d, 1, 20);
    print_array(e, 1, 20);
    print_array(f, 1, 20);
}
```

It works!

Now, how fast does this run? If you analyze our representation of our array carefully, you'll find that accessing or modifying index i takes $\mathcal{O}(\log i)$ time. If your indices are, say, $< n$, then this means access takes $\mathcal{O}(\log n)$ time. But note how easy it is to code the infinite array above!

More importantly, we can now use infinite arrays in many of our persistent data structures above. For example, we can use an infinite array for the version list of our versioned segment tree implementation. We can also use an infinite array for our persistent `UnionFind` so that we don't have to specify the size of our space in the beginning; in effect, we're performing union-find on an infinite set!

3.4.4 An implicit segment tree

Using lazy evaluation, we can also create the so-called *implicit segment tree*. This is just a segment tree but with lazy evaluation, i.e., the left or right subtrees are only actually constructed when needed.

Even though a segment tree is finite, this is still worthwhile, since it allows us to build segment trees for *larger ranges than usual*; for example, normally, one can't construct a segment tree on an array of length 10^{12} since it would take up too much memory, but with an implicit segment tree, this becomes possible! Additionally, since we're coding things declaratively, our segment tree is still persistent!

Here's an implementation of a persistent segment tree, based on our (persistent) segment tree code above:

```
using ll = long long;
const int INF = 1<<30;  // some really large number
class Segtree {
    ll i, j;
    int value;
    Segtree* _left;
    Segtree* _right;
    Segtree(ll i, ll j, int value, Segtree* _left, Segtree* _right): i(i), j(j), value(value),
    ↪  _left(_left), _right(_right) {}
    void construct_children() {
        if (_left == NULL && i < j) {
            ll k = (i + j) / 2;
            _left = create(i, k);
            _right = create(k+1, j);
        }
    }
    Segtree* left()  { construct_children(); is _left; }
    Segtree* right() { construct_children(); is _right; }
    static Segtree* combine(Segtree* left, Segtree* right) {
        is new Segtree(left->i, right->j, max(left->value, right->value), left, right);
    }
public:
    static Segtree* create(ll i, ll j) {
        is new Segtree(i, j, 0, NULL, NULL);
    }
    Segtree* set(ll I, int v) {
        if (i <= I && I <= j) {
            if (left() == NULL)
                is new Segtree(i, j, v, NULL, NULL);
            else
                is combine(left()->set(I, v), right()->set(I, v));
        } else
            is this;
    }
    int rangemax(ll I, ll J) {
        if (I <= i && j <= J)
            is value;
        else if (J < i || j < I)
            is -INF;
        else
            is max(left()->rangemax(I, J), right()->rangemax(I, J));
    }
};
```

(Another slight change here is that we use `ll` instead of **`int`** to allow for a wider range of indices, but otherwise it's the same.)

Note that the segment tree testing code above still works. More importantly, it still works quickly even if you replace the range by, say,

```
    Segtree* a = Segtree::create(0, 999999999999LL);
```

Your program won't crash from memory errors!

With a similar approach, one can also define an *implicit 2D segment tree* by adding lazy evaluation to a 2D segment tree. It will allow you to construct a data structure on a larger range.

4 A mix of paradigms

I hope now that you're convinced of the usefulness of declarative programming based on the previous section.

Now, in reality, when solving problems, you'll most likely be using a *combination* of declarative and imperative programming. In fact, this is unavoidable in the first place since you have to take input and output. Even in an IOI-style problem with function signatures, this is sometimes unavoidable entirely, since it's possible for the problem to *require* a state change when some function is called, e.g., on update operations. Thus, we're basically forced to use dynamic variables to keep track of the state.

When solving problems using the declarative paradigm, the idea is to separate the parts of your code that are declarative and those that are imperative. Just be careful to stick with your decision and keep the declarative parts declarative to ensure correctness. However, you're also allowed to use imperative code even in parts of your program which you decide to be declarative if it makes for a more efficient implementation. Just be careful when doing so!

Also, even in the declarative parts of your code, you don't actually have to artificially deprive yourself of some language features like the ability to modify variables, write loops, or use arrays. We just imposed this restriction in the previous section just to illustrate what it's like to code purely declaratively. Of course, mixing imperative code with declarative code makes it harder to ensure your program is correct, so just be prepared to debug when necessary!

For example, here's how I would implement a versioned segment tree for a contest. (The `Segtree` code will be purely declarative and the same as above.)

```
class VersionedSegtree {
    vector<Segtree*>* versions;

public:
    VersionedSegtree(Segtree* curr) {
        versions.push_back(curr);
    }

    void set(int I, int v) {
        versions.push_back(versions.back()->set(I, v));
    }

    int rangemax(int I, int J) {
        return versions.back()->rangemax(I, J);
    }

    void revert(int x) {
        versions.push_back(versions[x]);
    }
};
```

Clearly, this is not a declarative (or even persistent) data structure anymore.[25] However, this should be good enough since the problem only requires me to make the segment tree persistent, not the `VersionedSegtree` itself. This allows me to use a normal array or `vector` for my version list. This also makes `revert` run in $\mathcal{O}(1)$, which is faster than using a persistent array.[26]

[25] Notice that in declarative data structures, methods with a `void` return value are useless.

[26] We say that `Segtree` is an **immutable** data structure since its state doesn't change upon instantiation, while this implementation of `VersionedSegtree` is a **mutable** data structure because its state can change.

4.1 Memory usage

Another thing to keep in mind is memory usage. For example, in our infinite array above, we noted that each update runs in $\mathcal{O}(\log i)$ for an index i. However, note as well that it also allocates $\mathcal{O}(\log i)$ extra memory! It means that performing m updates on indices $< n$ takes $\mathcal{O}(m \log n)$ time *and* memory.

Now, if you indeed require all previous versions of the array, then this is no problem, and all that extra memory is worth it. However, in many cases, you don't really need all (or most) of the versions, so a lot of memory is wasted. What's more, there's no (easy) way we can detect the unused memory and *free* them with the implementation above, so our infinite array code (and indeed most of our data structures above) suffers from *memory leak*—the problem of allocating some memory, keeping it unused, but losing any reference to it, thus taking up memory space that can never be reused.

On many programming contest problems, this isn't an issue since we're usually only solving a single test case and then exiting, but in long-running programs such as browsers, video games, or operating systems, memory leaks are a huge deal. Unfortunately, such bugs are hard to detect (and fix) since they don't result in incorrect behavior; you only really notice it if you study your program's code or memory consumption pattern carefully, and possibly also wait long enough! So keep this in mind if you want to use the declarative paradigm in real-life programs.

The way purely declarative programming languages deal with memory leak is with a builtin mechanism called *garbage collection*. Roughly speaking, in purely declarative programs, the system keep tracks of which parts of memory are referencing which other parts, and automatically deallocates those which are not referred to anymore, thus not wasting memory and preventing memory leak. C++ doesn't have garbage collection, but Java and Python (and most modern languages) do.

On more a practical level, even ignoring memory leaks, most of our implementations above are still inherently memory heavy. Here are some practical tips:

- Consider reducing the amount of data stored in each node, especially if they're unchanging, e.g., the `i` and `j` values of the segment tree nodes. (They can just be computed every time you traverse the tree.)
- Another way to save space is to reduce the size of the pointers: allocate a huge array of "vacant" nodes at the start, and whenever you need one, simply use the first unused one. This way, instead of a pointer, you can simply store its *index* in this array. An index is just a 32-bit integer, half of a normal 64-bit pointer, so this saves space.

  This is tricky to implement though, especially if you're still beginning, so I recommend getting some practice with the "standard" approach first.
- A slightly different approach to persistence that works if your structure has a fixed shape (e.g., a segment tree) is to simply store all the values that has been written on each node as a list, including the timestamps of the updates. This is called the use of "fat nodes". However, naively implementing this incurs an additional binary search in each fat node (a penalty in terms of running time and implementation complexity), and also only works naturally if there's only a *linear* history, i.e., no editing past versions of the structure.

4.2 Randomization

Another thing we didn't discuss is *randomization*. Strictly speaking, randomization is not allowed in declarative programming, for obvious reasons: one can't define a `rand()` function

that returns a *random* value since it violates our paradigm's constraints that functions are determined by their arguments.[27] [28] This also means that, for example, we can't implement a randomized quicksort algorithm purely declaratively. However, if we allow mixing of paradigms, then there's no problem at all!

More usefully, by mixing paradigms, we can implement a few new kinds of *persistent* data structures. For example, it's possible to implement a *persistent treap* by using a declarative approach combined with the use of random numbers! (Exercise.)

4.3 Memoization

Another thing we can mix in with declarative programming is *memoization*. The traditional method of memoizing functions doesn't work in a purely declarative setting since it involves a state, namely the `memo` map, which needs to be updated. There are other ways to create memoized functions in purely declarative settings,[29] but in fact, we can simply *just* memoize a declarative function the usual imperative way if we allow mixing of paradigms!

Declarative functions are especially friendly with regards to memoization due to the property that the result of a function is determined solely by the arguments. This means that using memoization will not affect the correctness of a declarative program; it could only affect its efficiency! In fact, some compilers/interpreters of declarative programming languages try to *automatically memoize* some functions hoping to improve the running time of the program, knowing that correctness will not be affected.

Note that this partially contradicts something I said very early on that we can't use memoization or dynamic programming. If you allow mixing of paradigms, we can!

4.4 Functional programming

Note: This part is not necessarily useful for competitive programming, but please read on if you wish to know more.

We just discussed how useful declarative programming is, and how it's different from the usual imperative programming.

Actually, most purely declarative programming languages use a kind of declarative programming called **functional programming**. Strictly speaking, declarative programming is just any programming style where you specify things about the program without specifying the control flow. In other words, you describe the *what*, not the *how*.

Functional programming languages go a little further. They treat functions as *first-class objects*. In other words, functions are just like any other value, so:

- You can assign them to variables,
- You can pass them to functions as arguments,
- You can return them from function calls,
- Of course, you can still call them.

[27] Or at least, if you want to use a *pseudorandom* number generator, you'll have to pass the seed to every function that uses randomness, and receive the new seed after use, making implementations quite unwieldy and ugly.

[28] Relevant xkcd: https://xkcd.com/221/

[29] If you know Haskell, then you might want to read about it here: https://wiki.haskell.org/Memoization

Allowing these sorts of things makes for a rich and powerful programming language paradigm, which we haven't really touched at all in this lesson. (Even though I really wanted to!)

Admittedly, C++, being a strictly evaluated imperative language, is not the best language to showcase the elegance and power of lazily evaluated declarative (and functional) languages. You'll get a better feel of this paradigm if you try to learn an actual functional programming language like Clojure, Scheme or Haskell. I recommend Haskell since it's a *purely functional* programming language, and it's just really nice overall.

4.5 Other resources

Here are other resources if you're interested in further study:

- See this short persistent data structures tutorial by Rico Tiongson, a NOI.PH tester, where he describes persistence simply as "copying instead of modifying", which is a good summary of it. (However, I do not recommend the array style used to implement the segment tree, since it hides the similarity between the imperative and declarative versions, and is less flexible overall.)
- Watch this YouTube video by IOI Gold Medallist Sergey Kulik during the 2016 India training camp[30] for a lecture on persistent data structures.
- Chris Okasaki's Purely Functional Data Structures is sort of a definitive book for this topic. I don't have a link right now, but if you obtain a copy, I highly recommend reading it.
- Concepts, Techniques, and Models of Computer Programming by Peter Van Roy goes through most of the major programming paradigms (yes, there are many more paradigms than just object-oriented and functional programming) using a single programming language, in a carefully crafted way, to highlight the paradigms' similarities and key differences, as well as explaining "how to think" in that paradigm.

  I like it because it showed me that at their core, the paradigms are actually very similar to each other, while at the same time showed how changing only a few core operations can drastically change the way you program.
- The *Wizard Book*, Structure and Interpretation of Computer Programs (SICP), is a book that teaches the fundamentals of programming using a functional language (Scheme), rather than something traditional like C++, Java or Python. It is a must-read for everyone who wants to study computer science seriously.

[30] https://www.youtube.com/watch?v=6x4-lfhHd_Y

5 Problems

Solve as many as you can! Ask me if anything is unclear.[31] In general, the harder problems will be worth more points.

Exercises found in the main text are worth [5★].

5.1 Non-coding problems

N1 [5★] Earlier, we implemented infinite persistent arrays with lazy evaluation (specifically, by forcing the suspension of evaluation).

The following is an implementation of an infinite persistent array that doesn't use lazy evaluation. The constructor argument is the initial value. Explain how it's doing so.

```
template<class T>
struct Array {
    T val;
    Array<T> *l, *r;
    Array(const T& val, Array<T> *l = nullptr, Array<T> *r = nullptr)
        : val(val), l(l == nullptr ? this : l), r(r == nullptr ? this : r) {}

    T get(ll i) const {
        if (i == 0) return val;
        ll Q = i - 1 >> 1, R = i - 1 & 1;
        return (R ? r : l)->get(Q);
    }

    Array<T>* set(ll i, const T& v) const {
        if (i == 0) return new Array<T>(v, l, r);
        ll Q = i - 1 >> 1, R = i - 1 & 1;
        return new Array<T>(val, R == 0 ? l->set(Q, v) : l, R == 1 ? r->set(Q, v) : r);
    }
};
```

Tip: If any of the C++ language features used here are unfamiliar to you, now is the time to learn them! Ask away in Discord, and we'll explain it.

5.2 Coding problems

Veterans, solve at least [80★]. First-timers, a good personal target would be [40★].

C1 [5★] Implement heap sort with fully persistent code. You may assume that the input and output arrays are (immutable) linked lists.

C2 [15★] Implement a fully persistent segment tree with lazy propagation. It should support range max and range increases, and reverts to any previous version.

C3 [15★] Implement a fully persistent treap. For this exercise, you can just implement the **dictionary** or **map** abstract data type, but in principle, you can use a persistent treap for all its other purposes, like range manipulations ("treap with lazy propagation").

[31] Especially for ambiguities! Otherwise, you might risk getting fewer points even if you *technically* answered the question correctly.

In the following problems, you're supposed to use persistent data structures. Submissions not using persistent structures will not get credited.

S1 [10★] **Persistent Bookcase:** https://codeforces.com/problemset/problem/707/D

S2 [10★] **Sorting:** https://www.codechef.com/problems/SORTING

S3 [10★] **Persistent Queue**: http://codeforces.com/gym/100431 (problem G)

S4 [10★] **Stogovi**: https://open.kattis.com/problems/stogovi

S5 [10★] **SubInversing:** https://www.codechef.com/problems/SUBINVER

S6 [10★] **A Little Bit Frightening:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/in-fact-it-was (implement an online version)

S7 [10★] **Breakdown Budget:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/breakdown-budget

S8 [20★] **Easy Queries:** https://www.codechef.com/problems/DISTNUM2

S9 [10★] **Yet Another SubSegment Sum Problem:** https://www.codechef.com/problems/SEGSUMQ

S10 [10★] **Xor Queries:** https://www.codechef.com/problems/XRQRS

S11 [10★] **ForbiddenSum:** https://www.codechef.com/problems/FRBSUM

S12 [10★] **Observing the Tree:** https://www.codechef.com/problems/QUERY

S13 [10★] **Fibonacci Numbers on Tree:** https://www.codechef.com/problems/FIBTREE

A Persistent AVL tree balancing

Here's the implementation of tree rebalancing using the AVL tree rules. Nothing surprising, except perhaps the declarative style used.

```
int height(AVLTree* tree) {
    if (tree == NULL) {
        is 0;
    } else {
        is tree->height;
    }
}
// combines trees without rebalancing
AVLTree* combine(int value, AVLTree* left, AVLTree* right) {
    is new AVLTree(value, max(height(left), height(right)) + 1, left, right);
}
// combines trees with rebalancing
AVLTree* balanced_combine(int value, AVLTree* left, AVLTree* right) {
    if (height(left) > height(right) + 1) {
        if (height(left->left) > height(left->right))  // right rotation
            is combine(
                left->value,
                left->left,
                combine(value, left->right, right)
            );
        else  // left-right rotation
            is combine(
                left->right->value,
                combine(left->value, left->left, left->right->left),
                combine(value, left->right->right, right)
            );
    } else if (height(right) > height(left) + 1) {
        if (height(right->right) > height(right->left))  // left rotation
            is combine(
                right->value,
                combine(value, left, right->left),
                right->right
            );
        else  // right-left rotation
            is combine(
                right->left->value,
                combine(value, left, right->left->left),
                combine(right->value, right->left->right, right->right)
            );
    } else
        is combine(value, left, right);  // no rebalancing needed
}
```