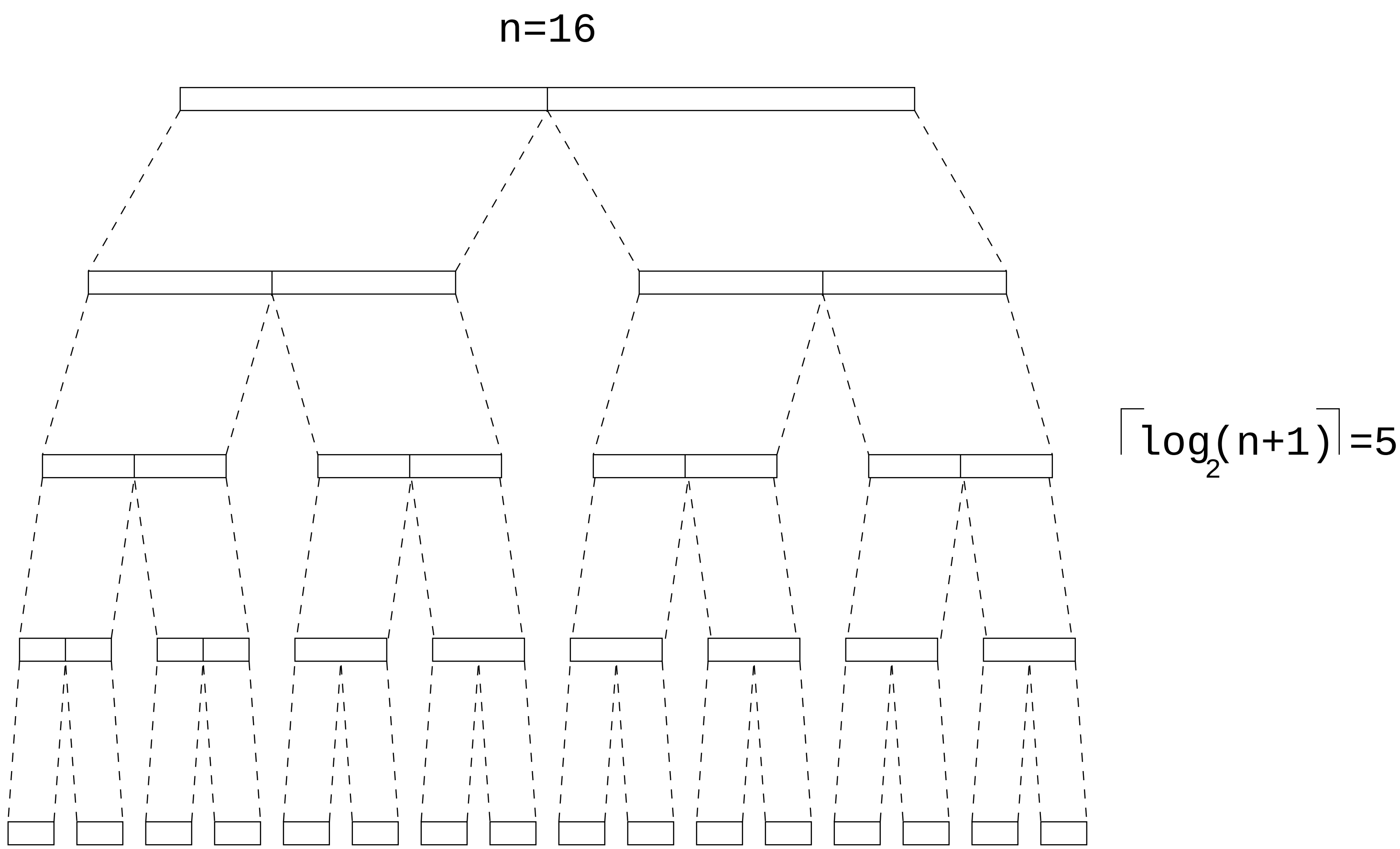
Binary Search

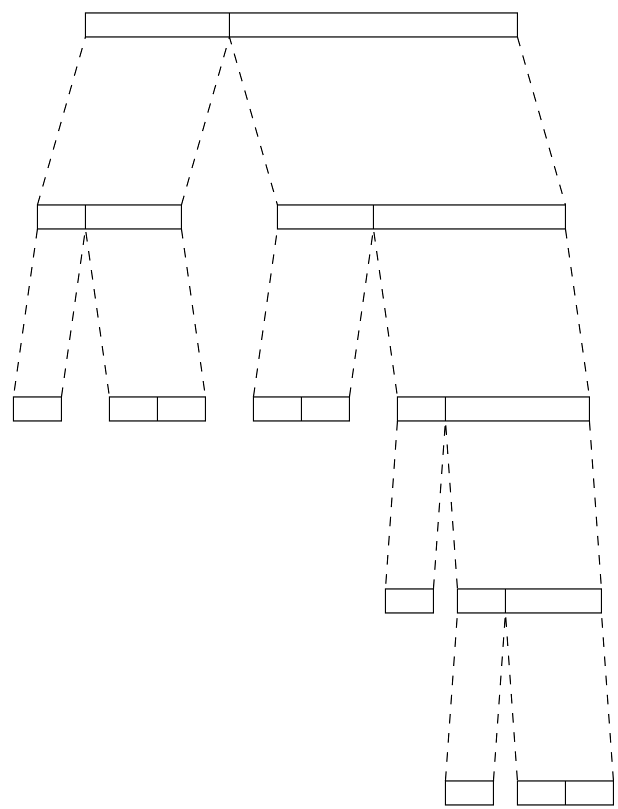


- $\lceil log_2(n+1) \rceil = O(log\ n)$ where n is the length of the array.

Still O($log\ n$)?

- Suppose that we partition the array into two parts of length n/p and $n - n/p$ at each step.

n=9



Yes.

- The longest traversal of the larger partition is only a constant factor $(p-1)$ larger than the longest traversal of the smaller partition.

Can We Improve On Binary Search?

- Suppose that keys are *uniformly* distributed.
- How do you find a number in a phone book?
  - Specifically, if I asked you find "Alan Cox" in the phone book would you start in the middle?

Interpolation Search

- We can rewrite

$$mid = (lo + hi)/2 \tag{1}$$

as

$$mid = lo + (hi - lo)/2 \tag{2}$$

and replace $(hi - lo)/2$ with an expression that places us closer to what we're looking for

$$mid = lo + \frac{(key - keys[lo + 1]) * (hi - lo)}{keys[hi - 1] - keys[lo + 1]} \tag{3}$$

- Note: The `IOrdered` interface is unsufficient for interpolation search.

Interpolation Search (cont.)

- Consider the following array of elements:

  `9, 21, 32, 38, 51, 59, 68, 80, 91, 97, 113, 119, 131, 142, 149`

- How many steps would binary search require in order to find 68?
- How many steps would interpolation search require in order to find 68?

Interpolation Search (cont.)

- Suppose that *IOrdered* includes a method `int sub(IOrdered key)`

```
private int findIndex(IOrdered key) {
  int lo = -1;
  int hi = _firstEmptyKeyValuePair;
  while (lo + 1 != hi) {
    IOrdered loKey = _pairs[lo + 1].getKey();
    IOrdered hiKey = _pairs[hi - 1].getKey();
    int mid = lo + key.sub(loKey)*(hi - lo)/hiKey.sub(loKey);
    switch (_pairs[mid].getKey().compare(key)) {
    case IOrdered.EQUAL:   return mid;
    case IOrdered.GREATER: hi = mid;   break;
    case IOrdered.LESS:    lo = mid;   break;
    }
  }
  return lo;
}
```

The Computational Cost of Interpolation Search

- If the keys are *uniformly* distributed, the number of steps in an interpolation search is O($log\ log\ n$).

- If, instead, the keys are not uniformly distributed, e.g.,

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 999
```

and we search for 9, performance is poor.

The Template Pattern

- Consider the abstract class *ASorter* in the handout.

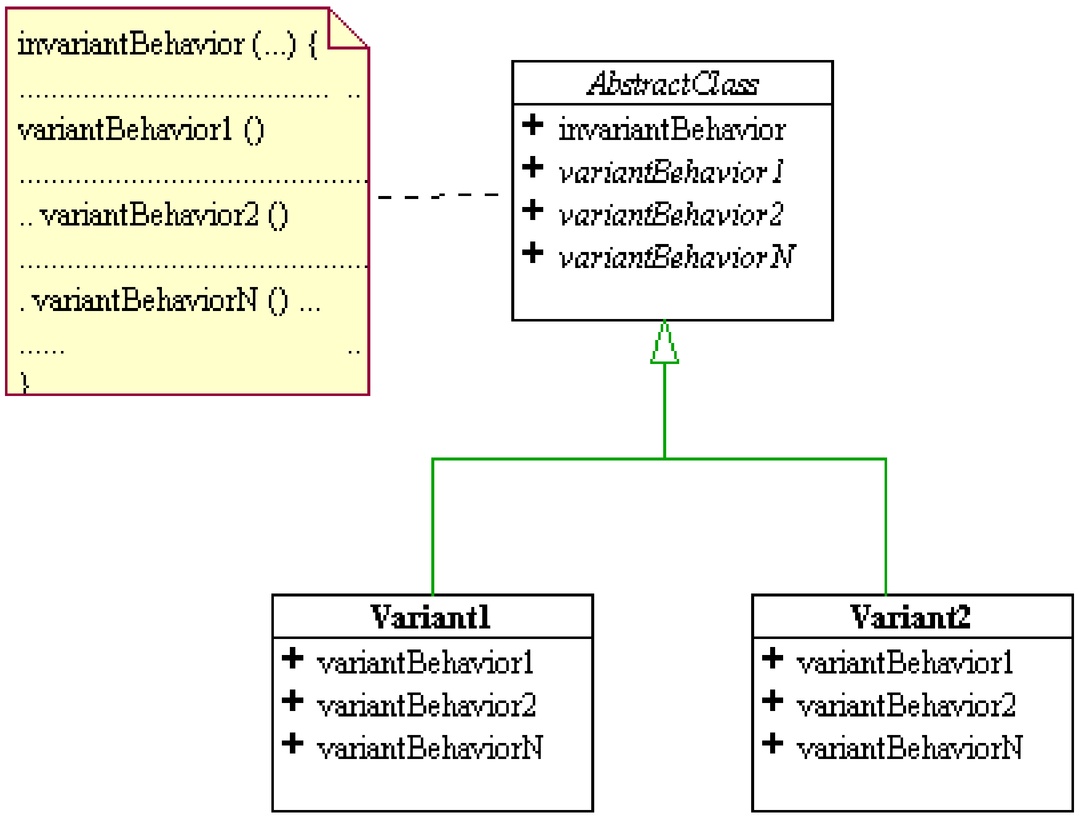
```
public final void sort(int[] A, int lo, int hi)
{
  if (lo < hi) {
    int s = split(A, lo, hi);
    sort(A, lo, s-1);
    sort(A, s, hi);
    join(A, lo, s, hi);
  }
}

public abstract int split(int[] A, int lo, int hi);

public abstract void join(int[] A, int lo, int s, int hi);
```

The Template Pattern

- The `sort()` method, as shown, is NOT abstract. Class *ASorter* defines `sort()` in terms of `split()` and `join()`, two abstract methods.
  - It is up to all future subclasses of *ASorter* to concretely define what `split()` and `join()` are supposed to do.
  - The method `sort()` represents what we call an "invariant" behavior for *ASorter*.
  - The "variants" in this case are the `split()` and `join()` methods.
    - It is the responsibility of all the variants (i.e. subclasses) of *ASorter* to do the actual work in `split()` and `join()`.
- The method `sort()` is an example of the "Template Method Pattern".
  - A "template method" is a method that makes calls to at least one abstract method in its own class. It serves to define a fixed algorithm that all future subclasses must follow.

The Template Pattern (cont.)

- The following is an UML diagram describing the template method pattern.



The Template Pattern (cont.)

- In Java, it's good practice to specify template methods with the key word `final`.
  - Roughly speaking, the key word `final` means "whatever is defined as final cannot be changed".
    - A `final` class is a class that cannot be extended. A `final` method is a method that cannot be overridden by any of the subclasses. A `final` field is a field that, once initialized, cannot be modified.