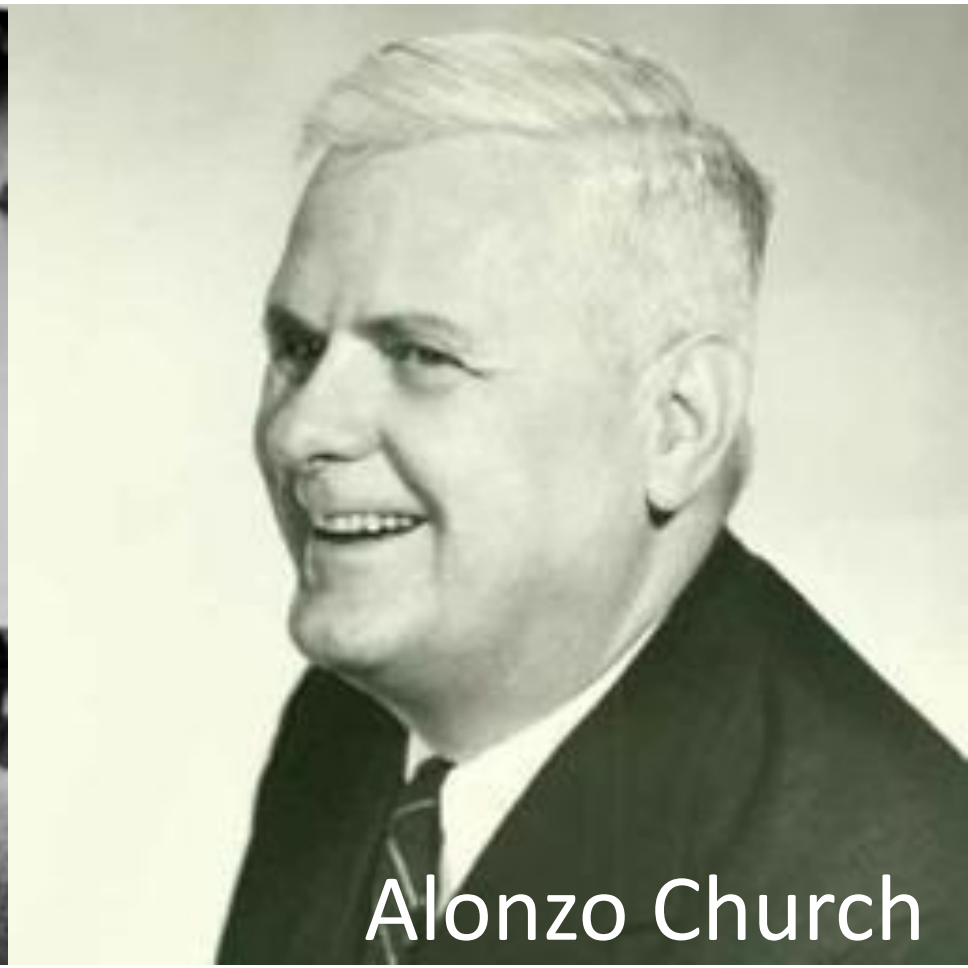
CSCI 1102 Computer Science 2

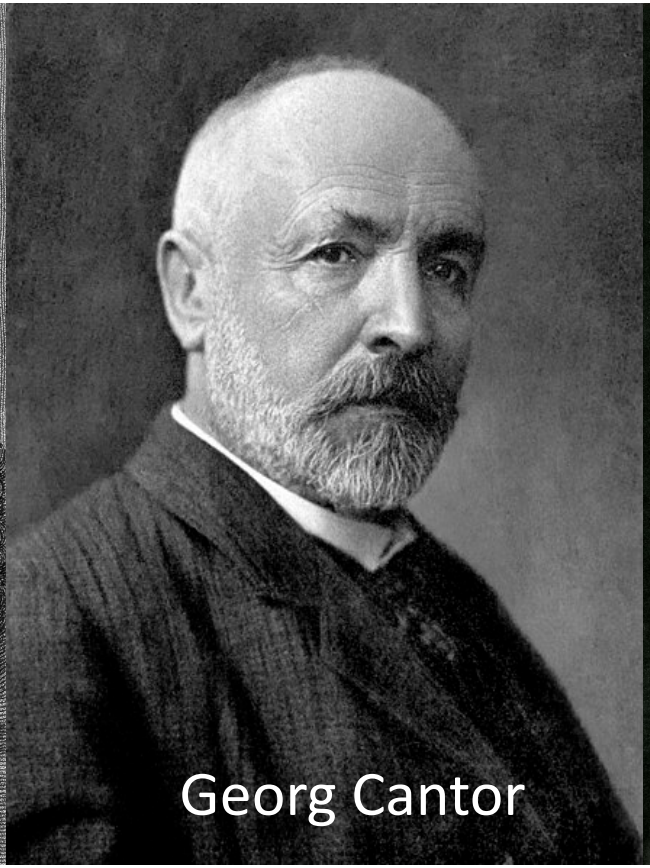
Meeting 13: Tuesday 3/16/2021

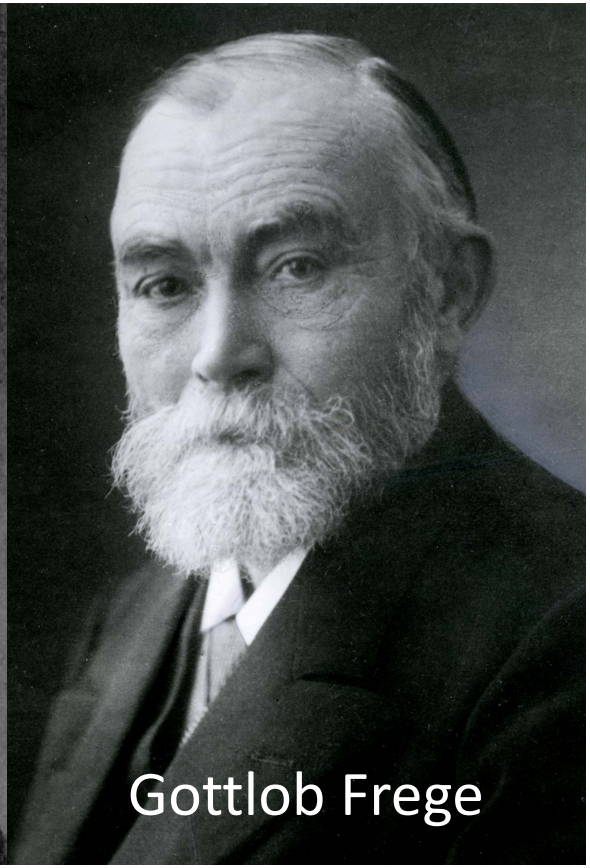
Sorting Algorithms

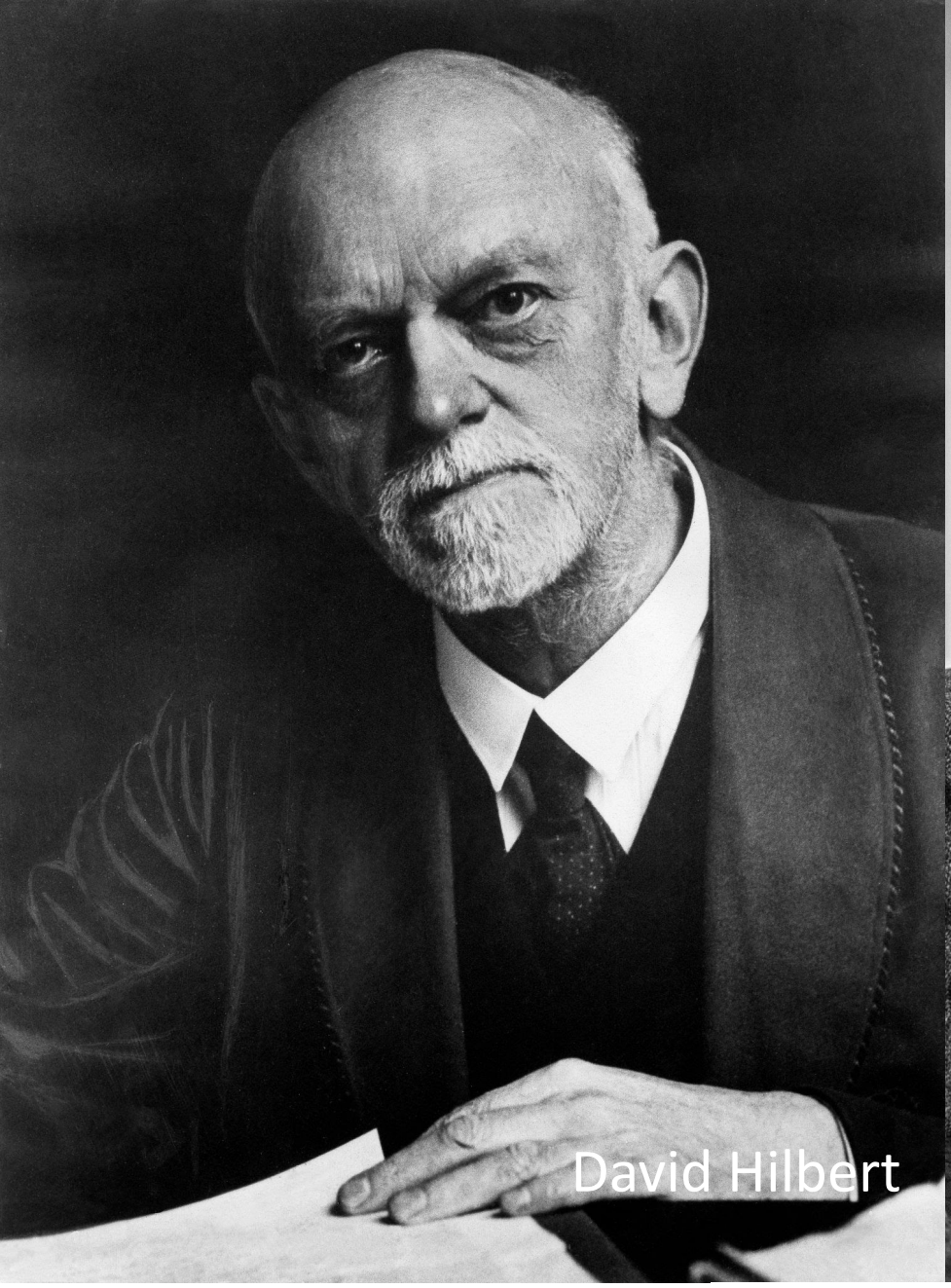


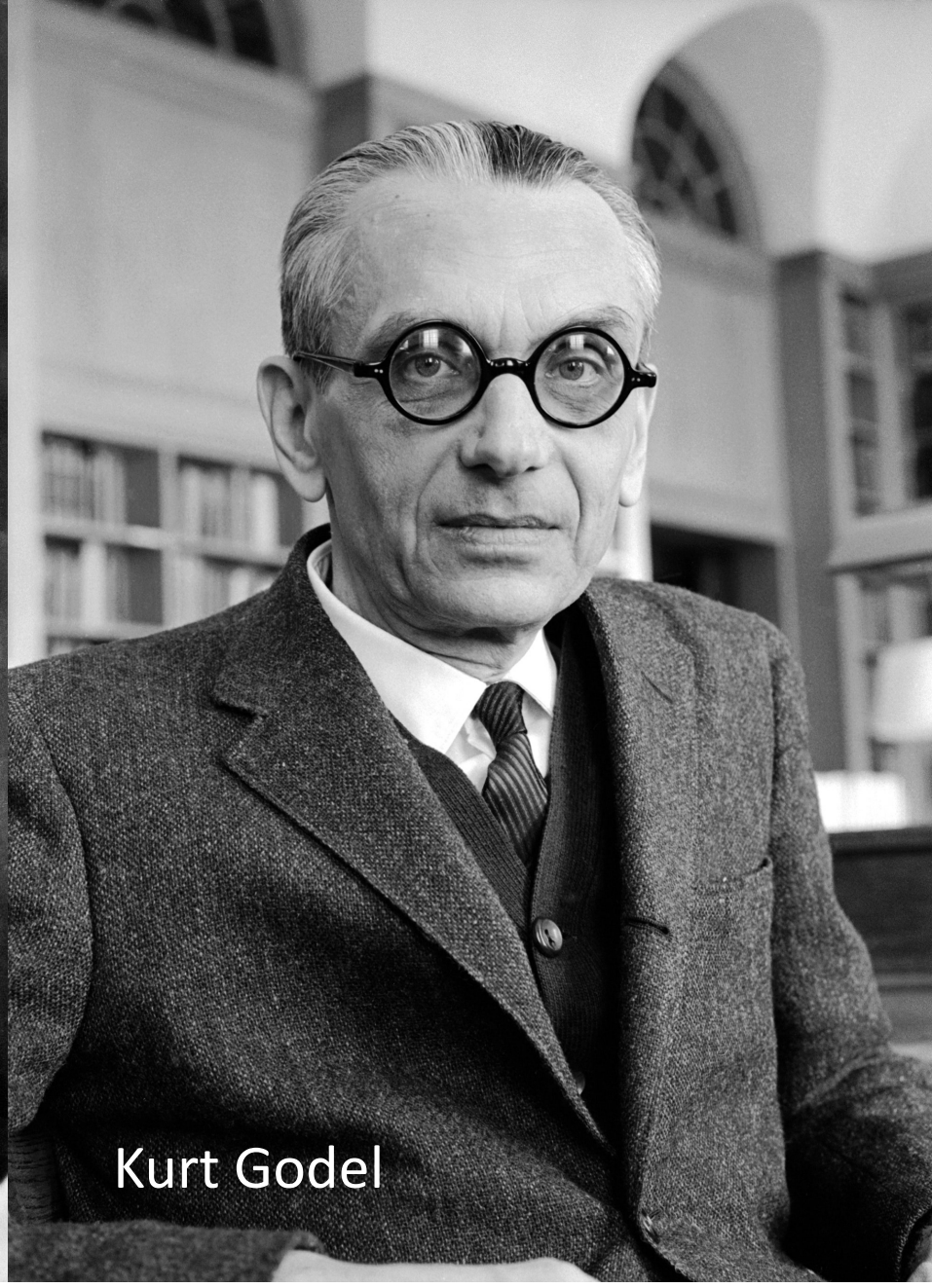
Alonzo Church


Gottfried Leibniz


Georg Cantor


Gottlob Frege


David Hilbert


Kurt Godel

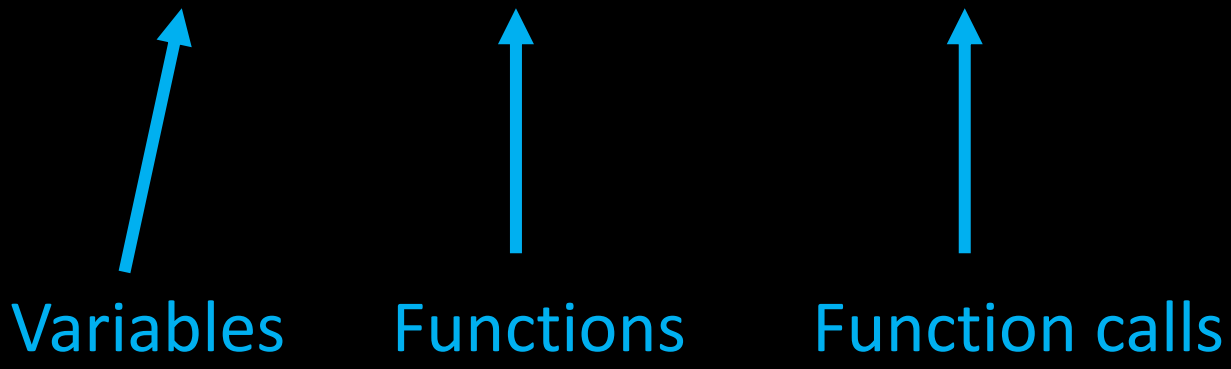
$$f(x) = 2x$$

$$\{(0, 0), (1, 2), (2, 4), (3, 6), \ldots\}$$

$$M ::= x \mid (\lambda x.M) \mid (M\ M)$$

Lambda Calculus

$$M ::= x \mid (\lambda x.M) \mid (M\ M)$$

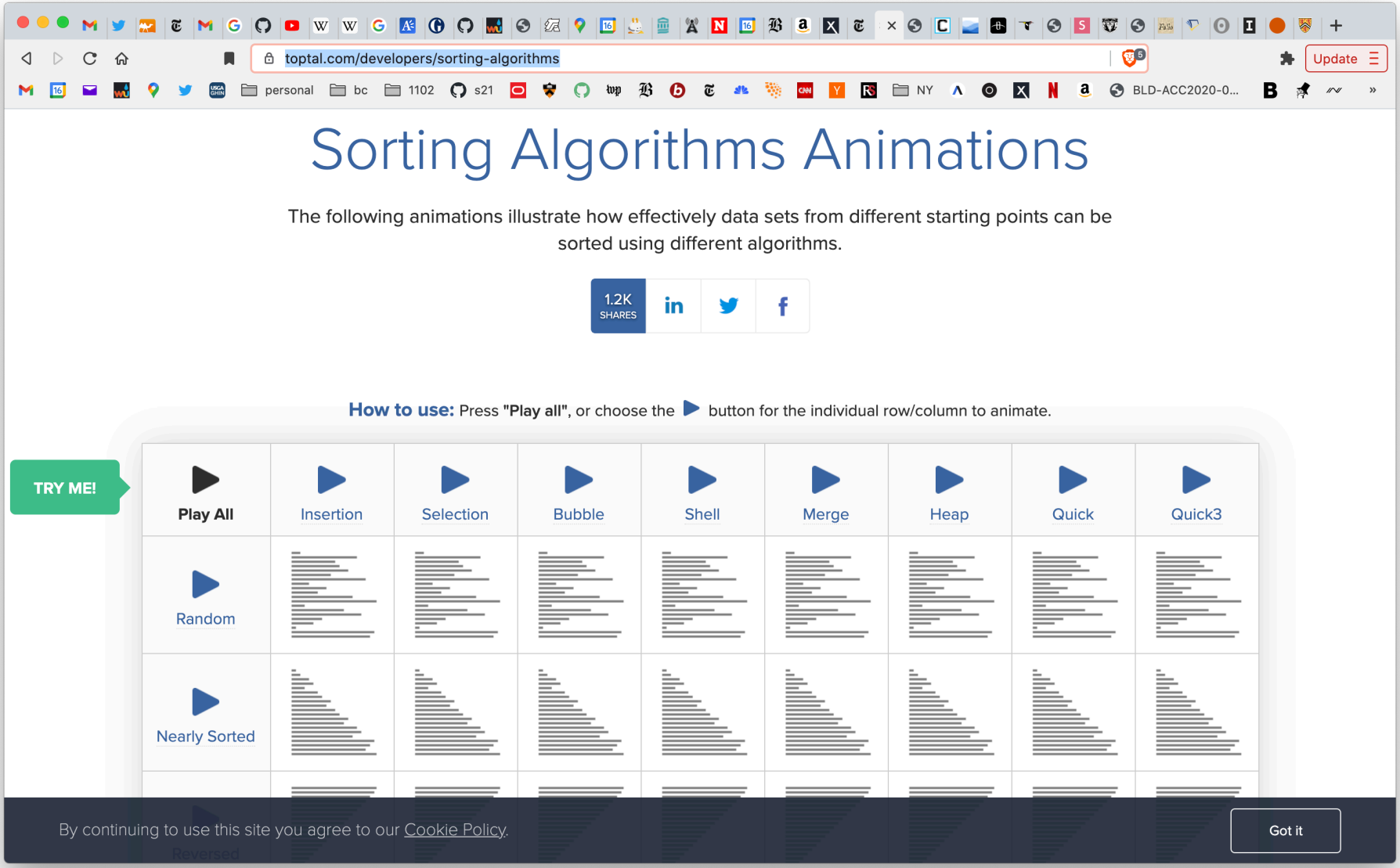


Lambda Calculus

- A mathematically well-founded theory of *functions as computation rules*.
- A mathematically well-founded characterization of effectively computability.
- The essence of a programming language.

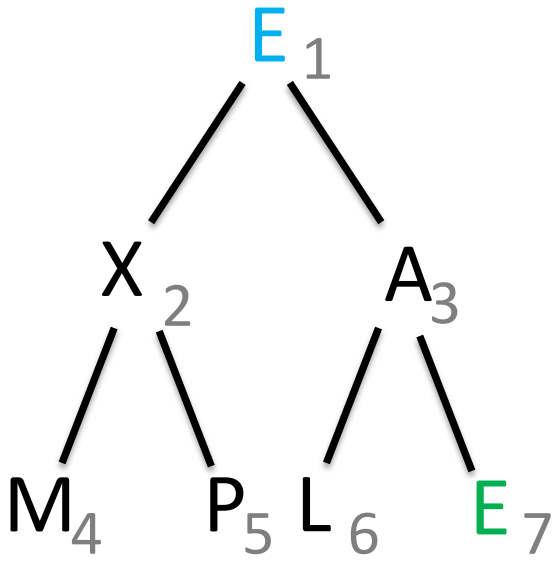
Today

Sorting Algorithms


toptal.com/developers/sorting-algorithms
Update
Sorting Algorithms Animations
The following animations illustrate how effectively data sets from different starting points can be sorted using different algorithms.
1.2K SHARES
How to use: Press "Play all", or choose the button for the individual row/column to animate.
TRY ME!
Play All
Insertion
Selection
Bubble
Shell
Merge
Heap
Quick
Quick3
Random
Nearly Sorted
Reversed
By continuing to use this site you agree to our Cookie Policy.
Got it

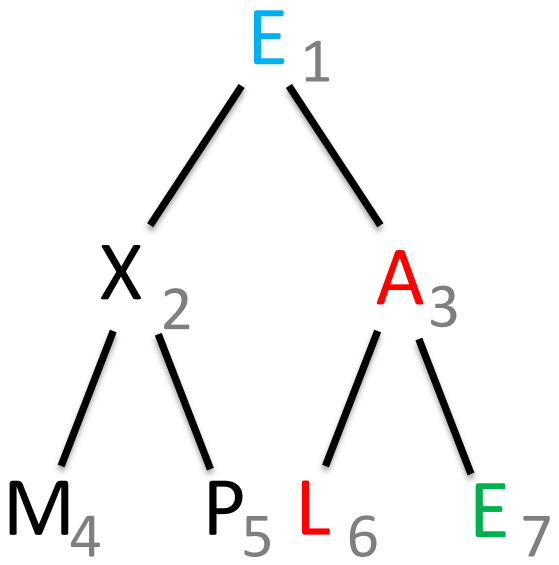
Heapsort

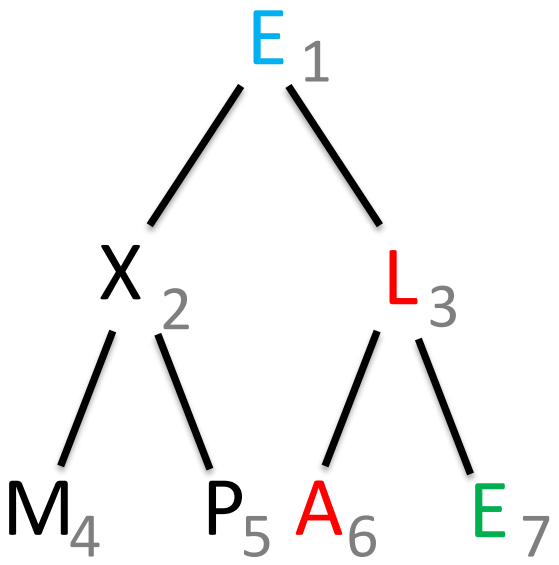
Bill Williams, 1964



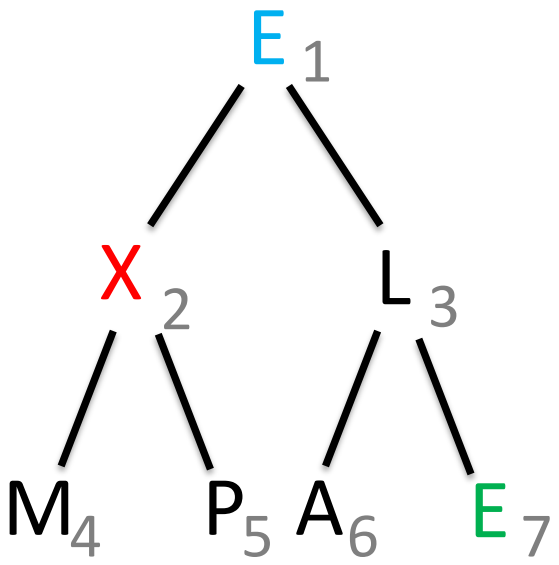
E X A M P L E

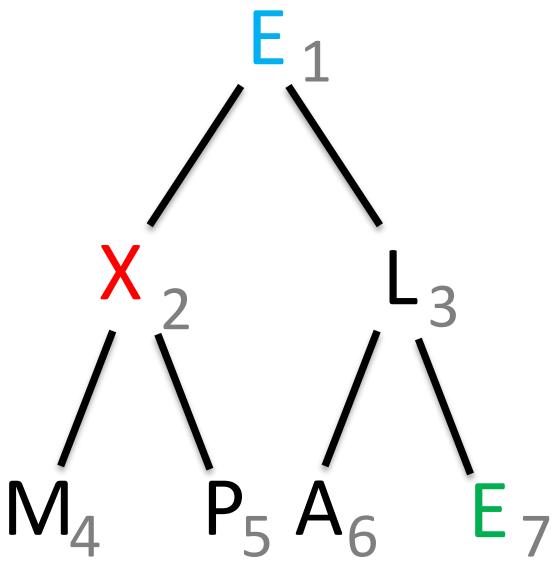
Build heap: sink(3)



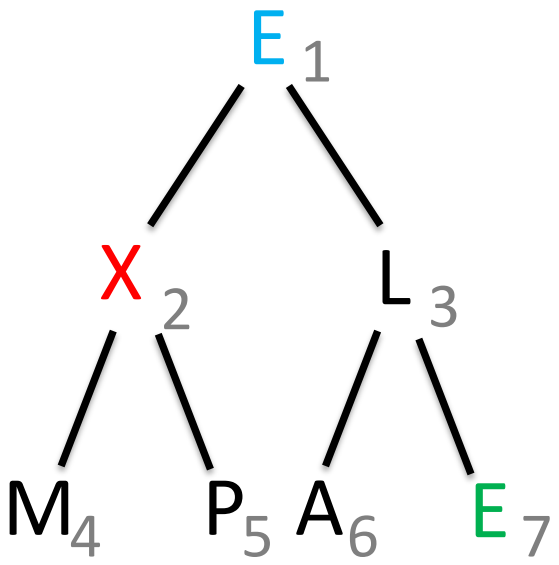


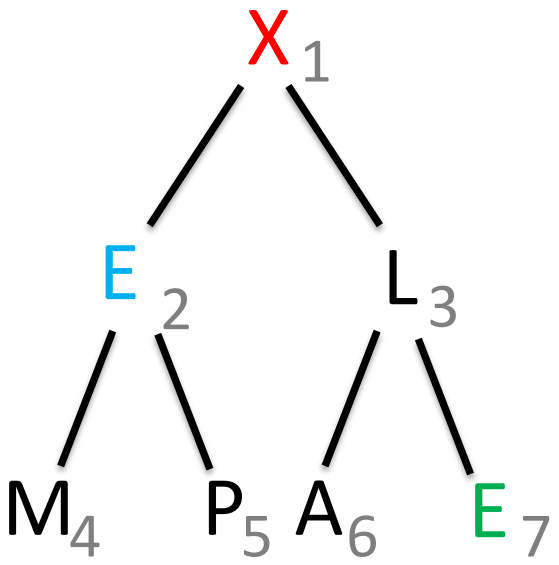
Build heap: sink(2)



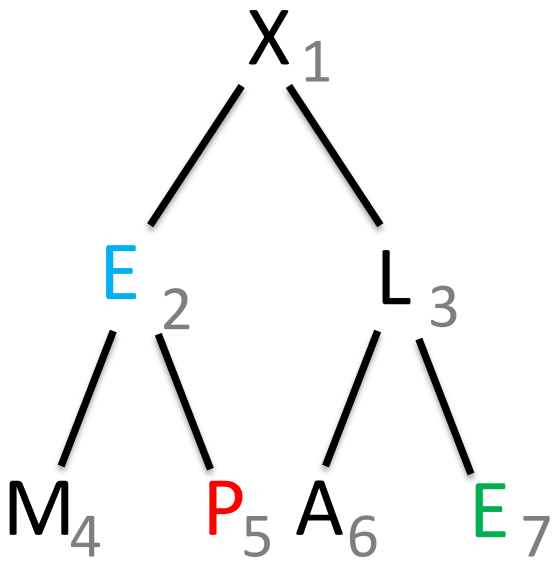


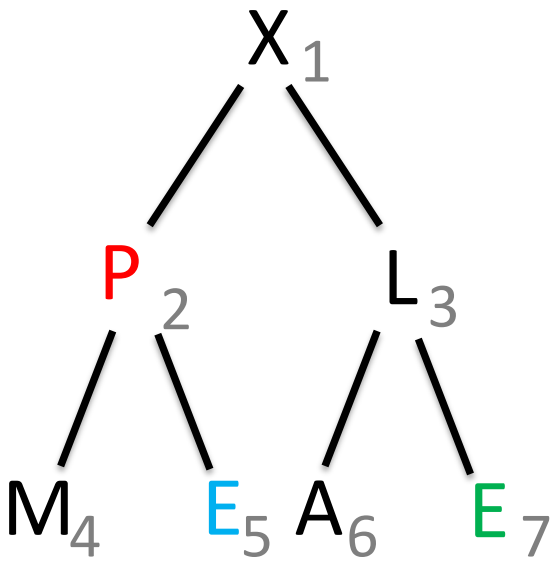
Build heap: sink(1).



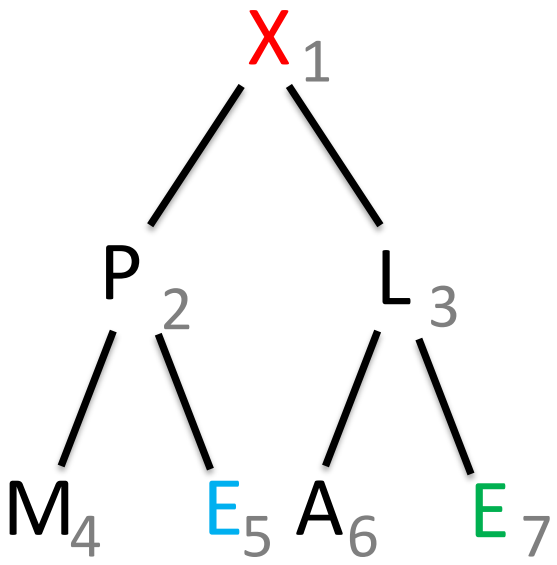


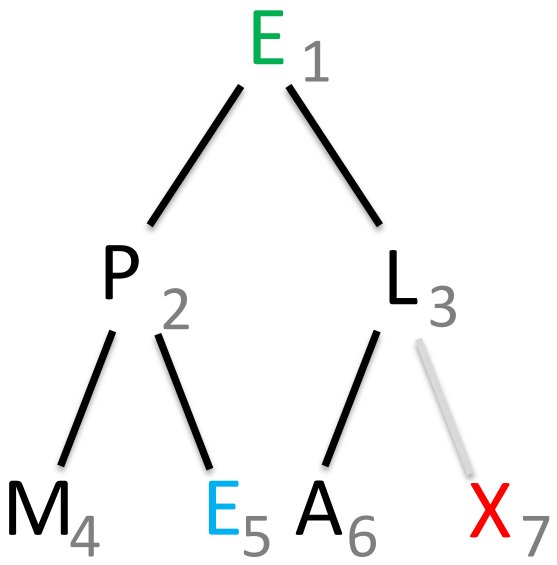
Build heap: after sink(2), heap is well-formed.



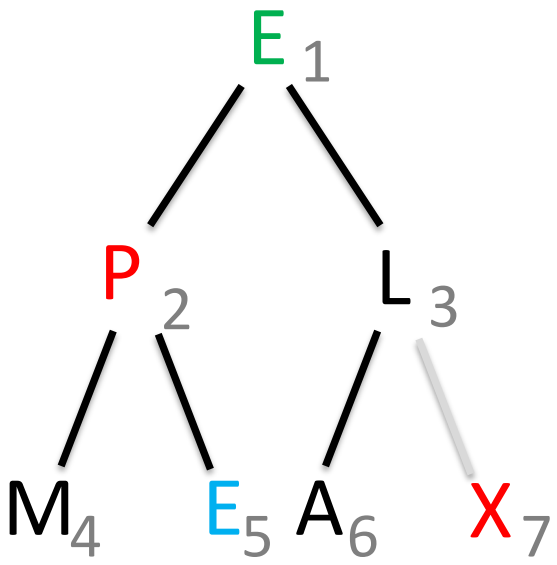


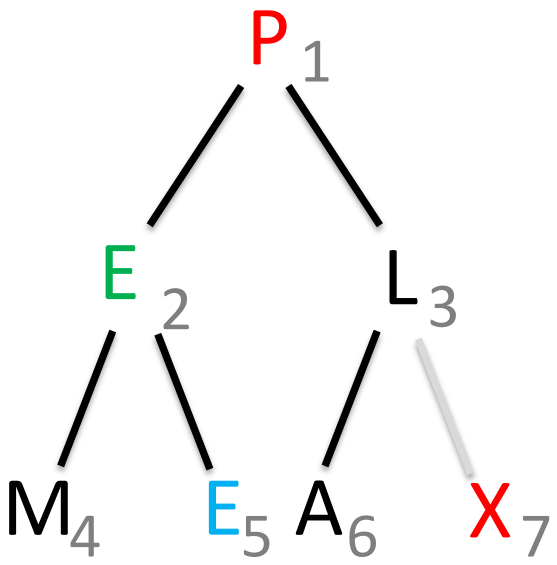
Sort: evict X, place it in used slot.



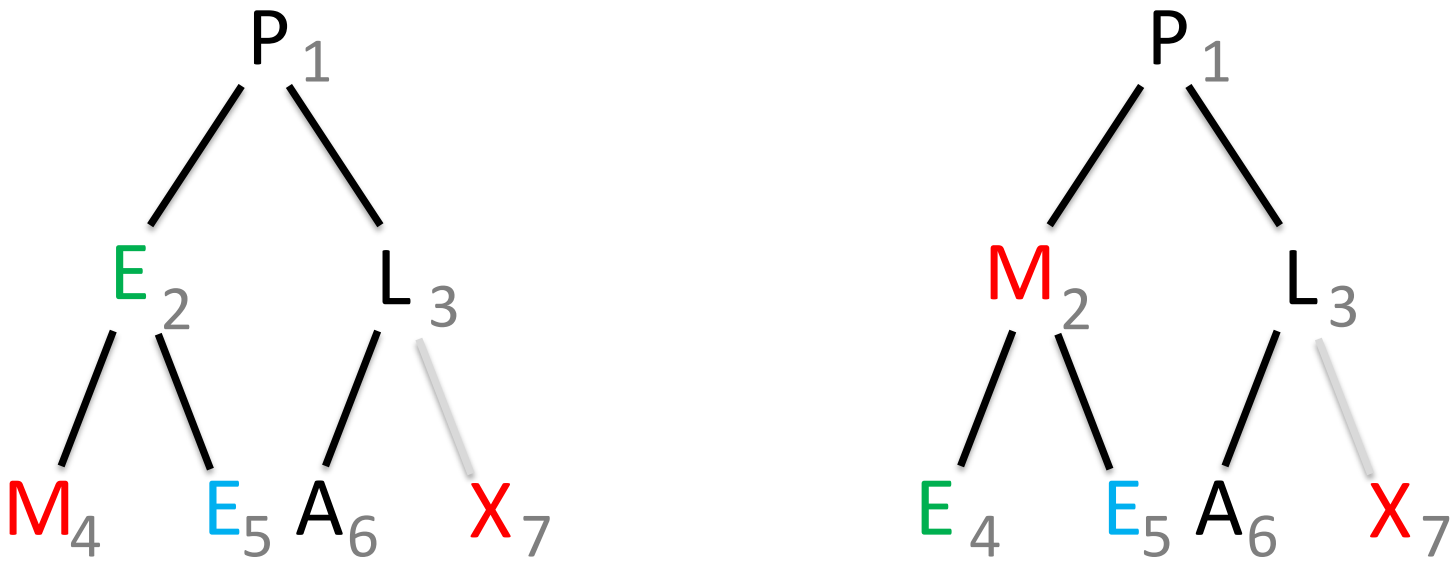


Sort: sink(1)

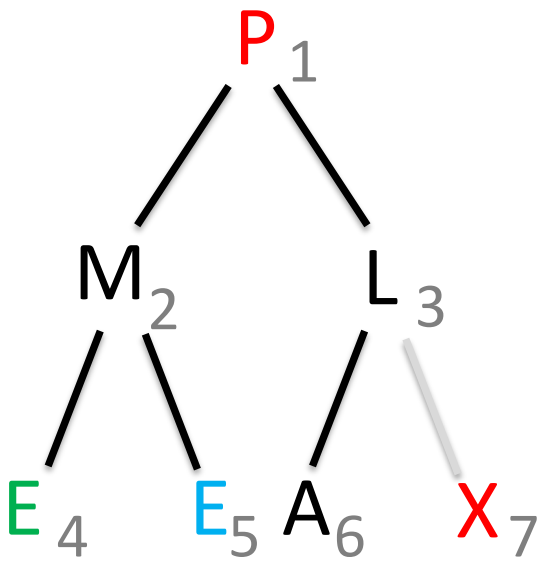


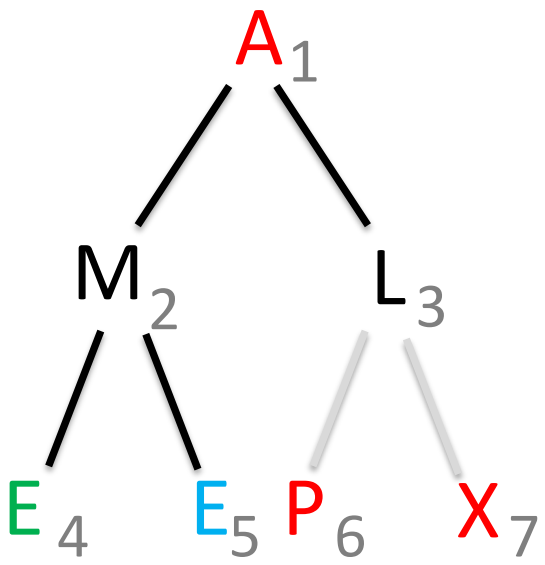


Sort: after sink(2), heap is restored.

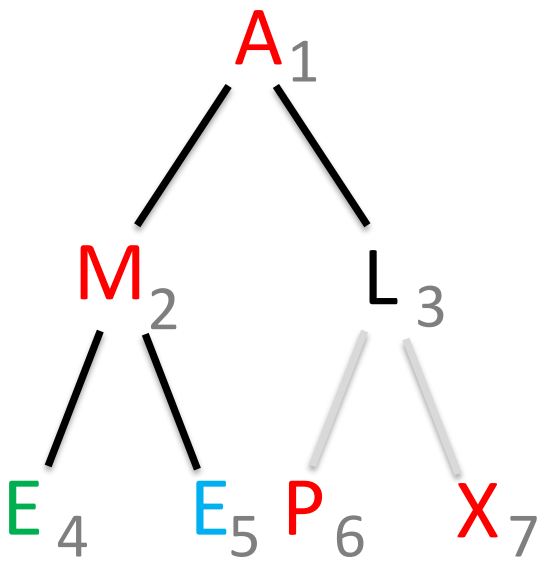


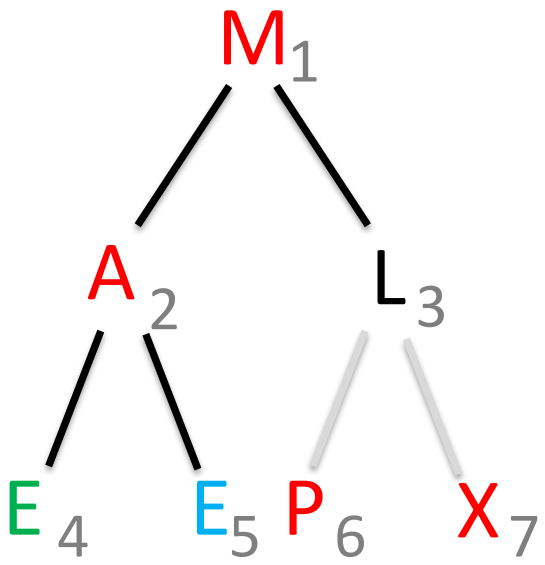
Sort: evict P, place it in used slot.



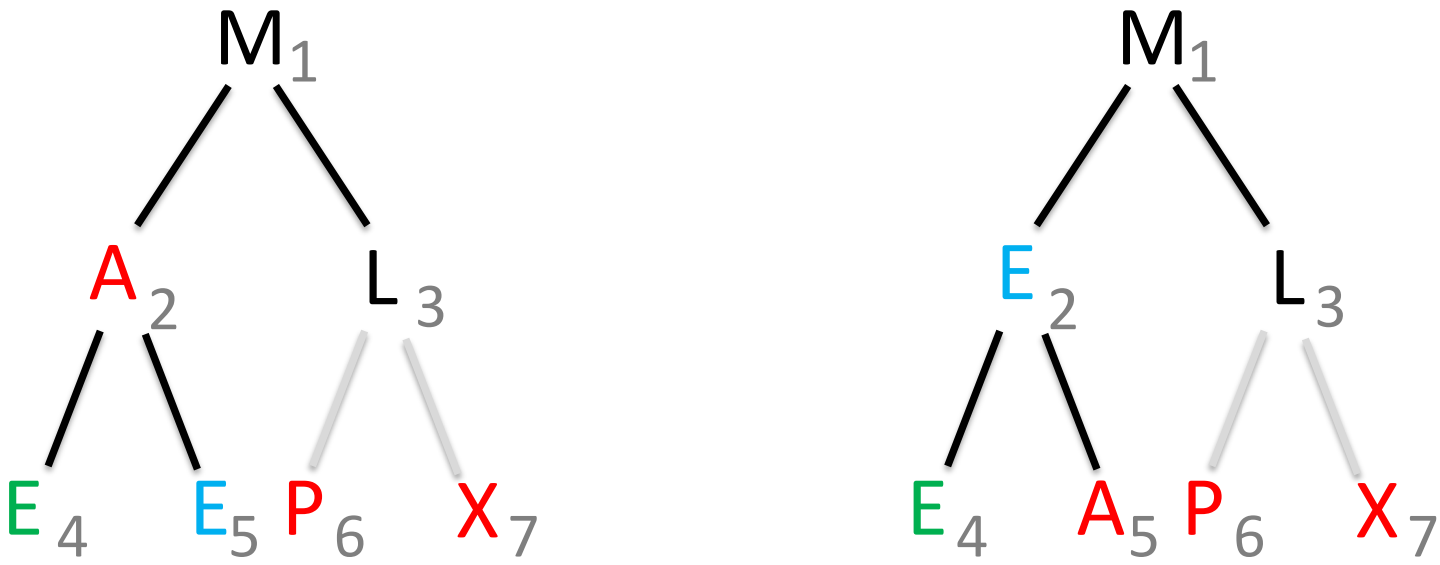


Sort: sink(1).

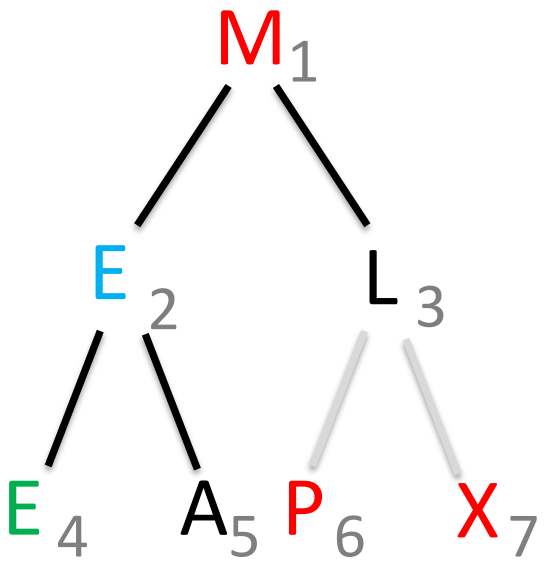


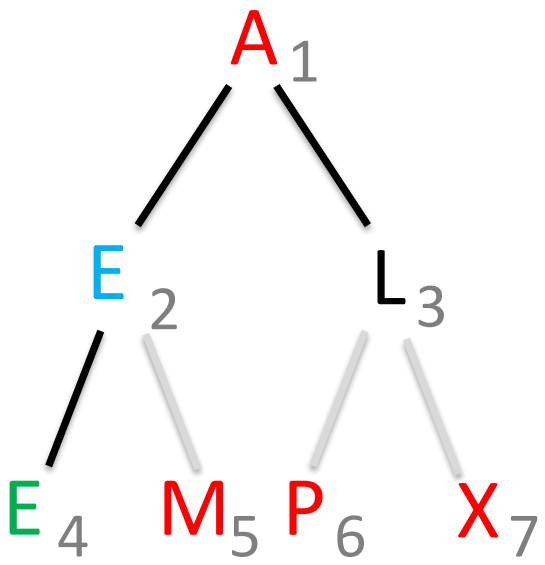


Sort: after sink(2), heap is restored.

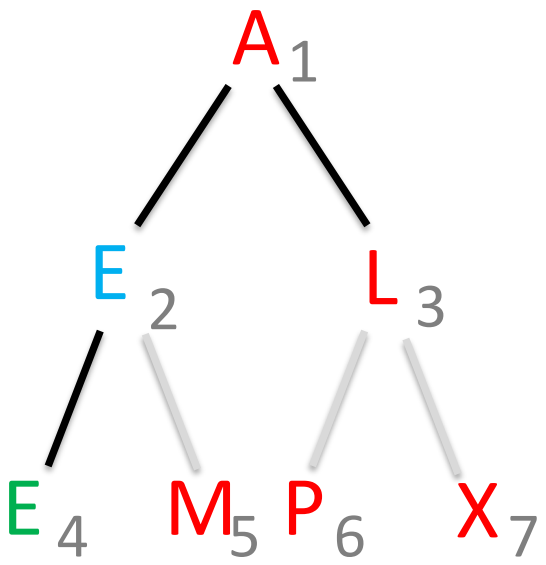


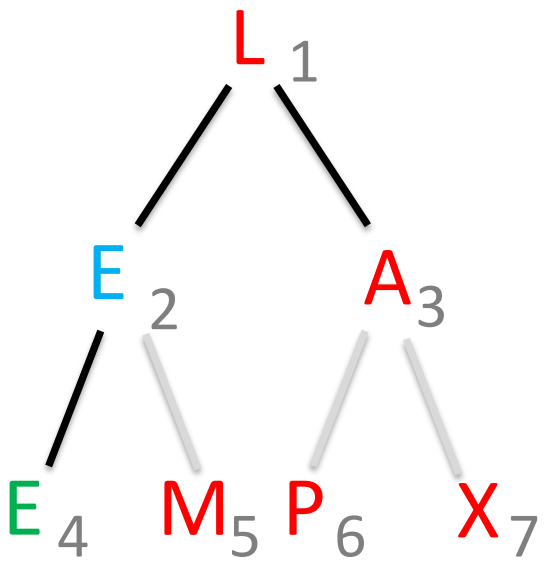
Sort: evict M from heap, place in unused slot.



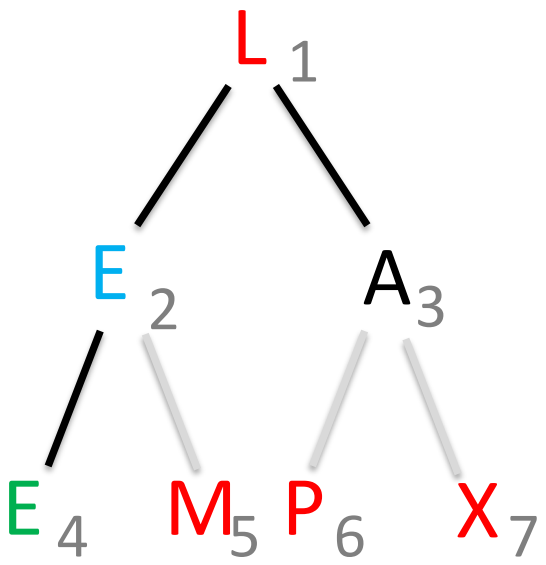


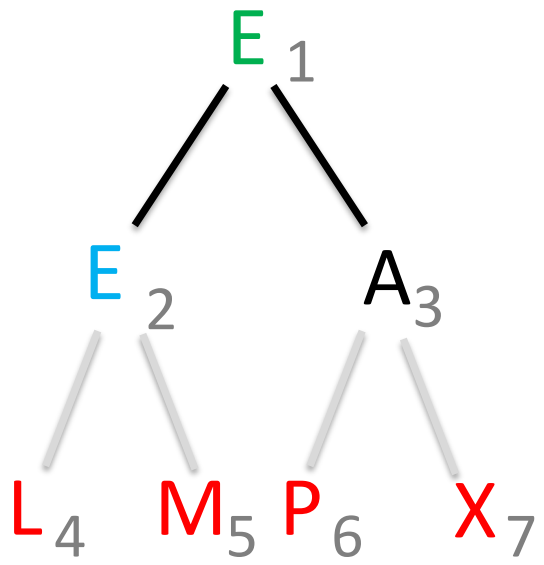
Sort: after sink(1), heap is restored.



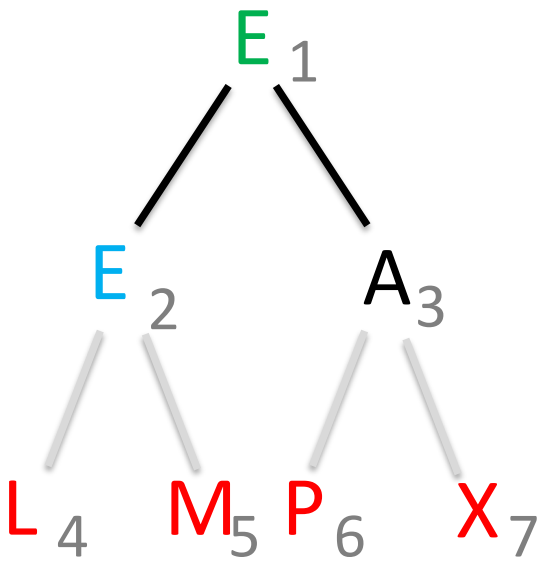


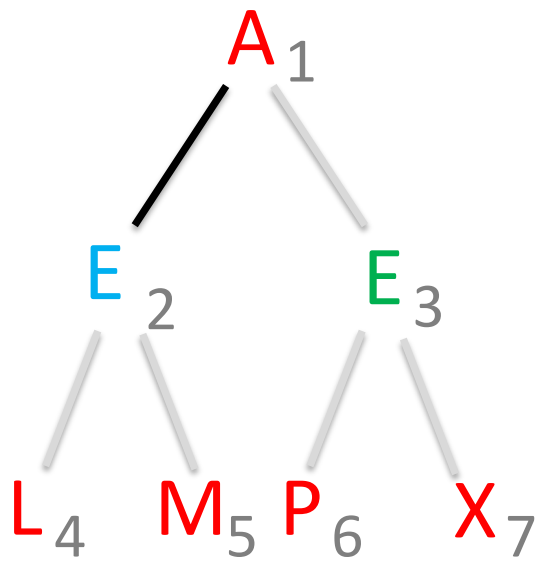
Sort: evict L from the heap, place in unused slot.



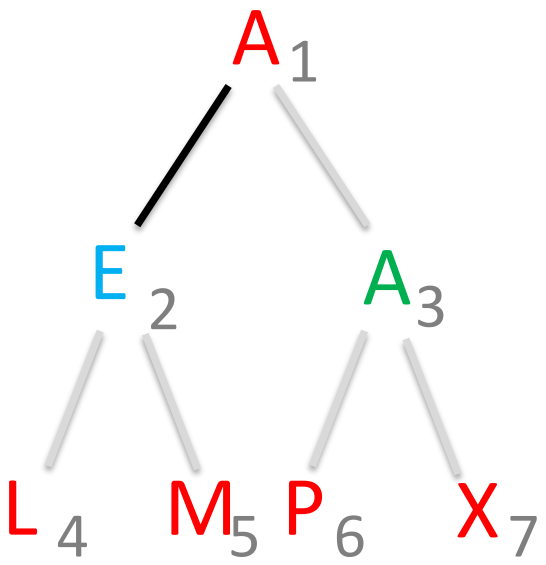


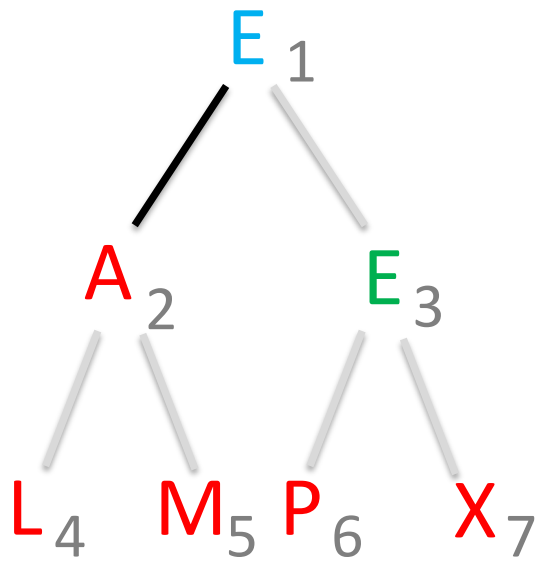
Sort: sink(1), no change, good heap, evict E.



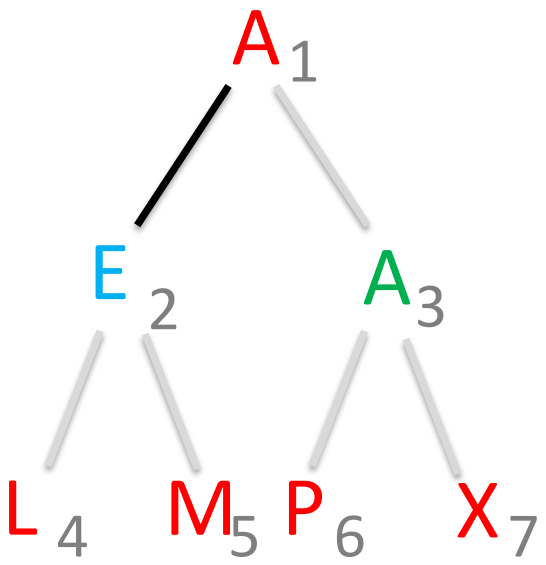


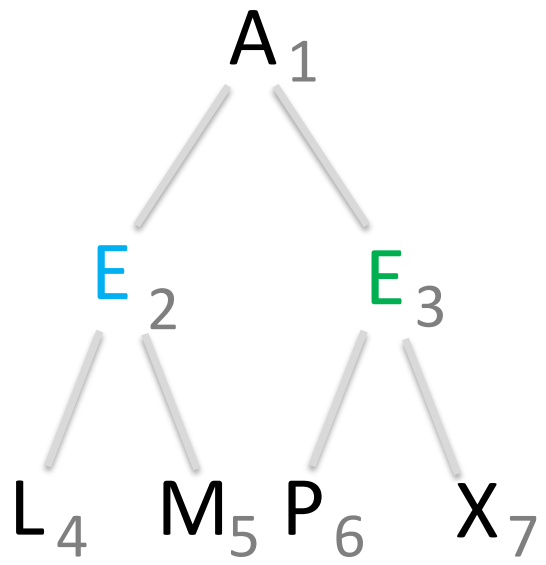
Sort: after sink(1), heap restored.





Sort: after evict E, a one-element heap.





A E E L M P X

Stability

- In this particular example, the order of E and E was preserved

E X A M P L E

A E E L M P X

- When a sorting algorithm preserves the order of equal keys the algorithm is said to be *stable*.

Heapsort

- Requires O(N log(N)) steps on average and in the worst case;
- Sorts *in place*, no additional storage required.
- In general, heapsort is unstable.

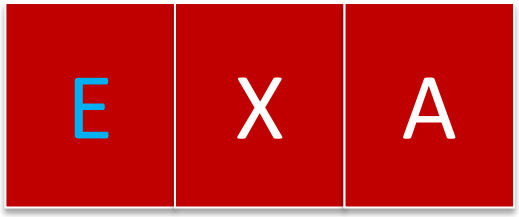
Mergesort

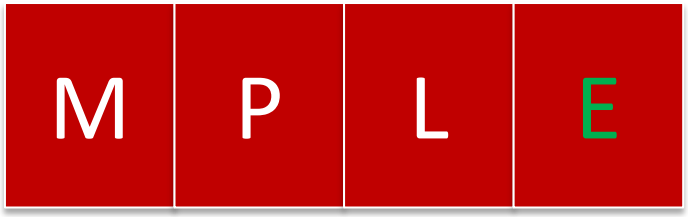
John von Neumann, 1945

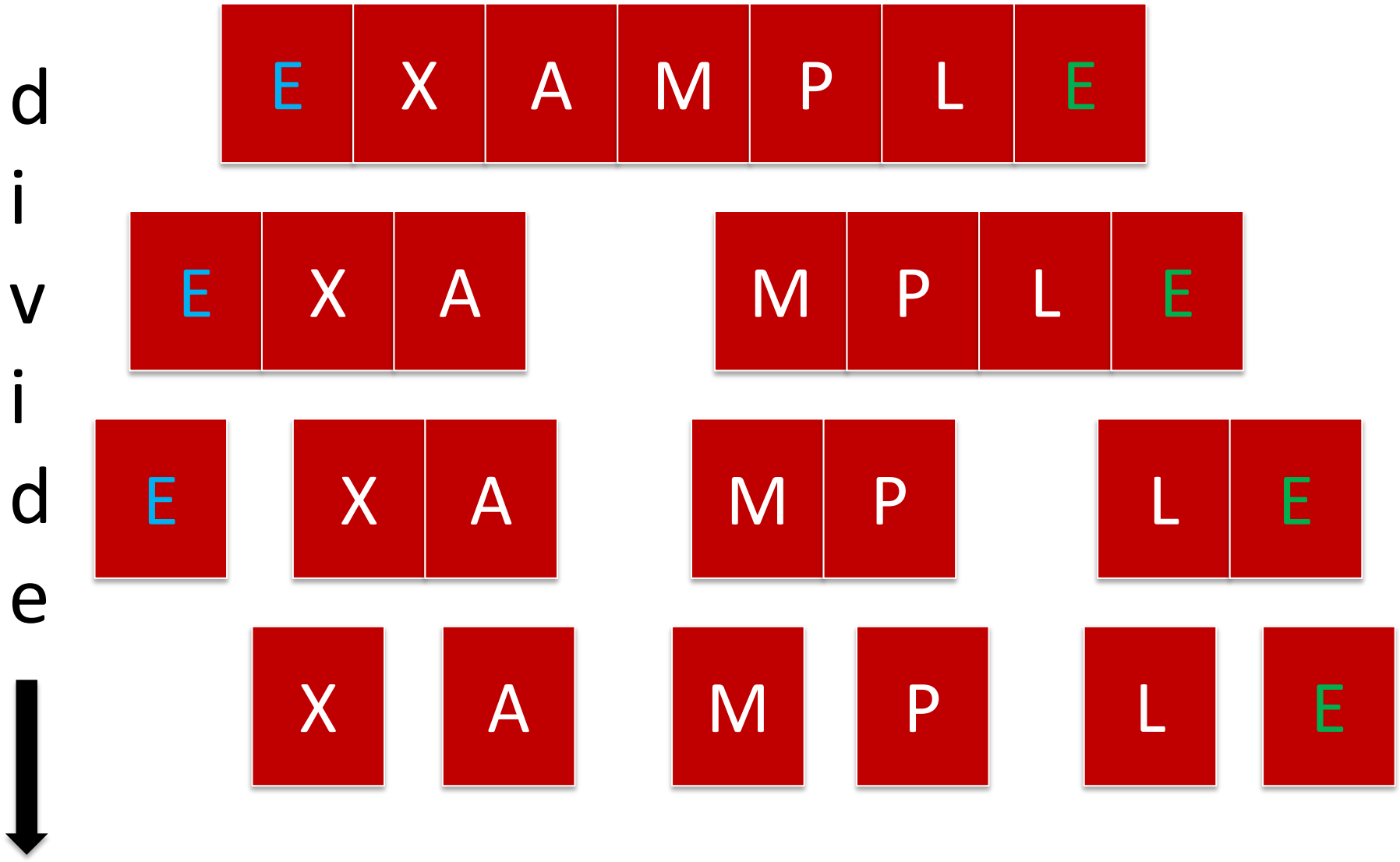
```
2#
3# If xs and ys are sorted lists, merge(xs, ys)
4# will be combined sorted list.
5def merge(xs, ys):
6  result = []
7  i = j = 0
8  while (i < len(xs) and j < len(ys)):
9    if xs[i] < ys[j]:
10     result.append(xs[i])
11     i += 1
12   else:
13     result.append(ys[j])
14     j += 1
15 result.extend(xs[i:] if i < len(xs) else ys[j:])
16 return result
17
```

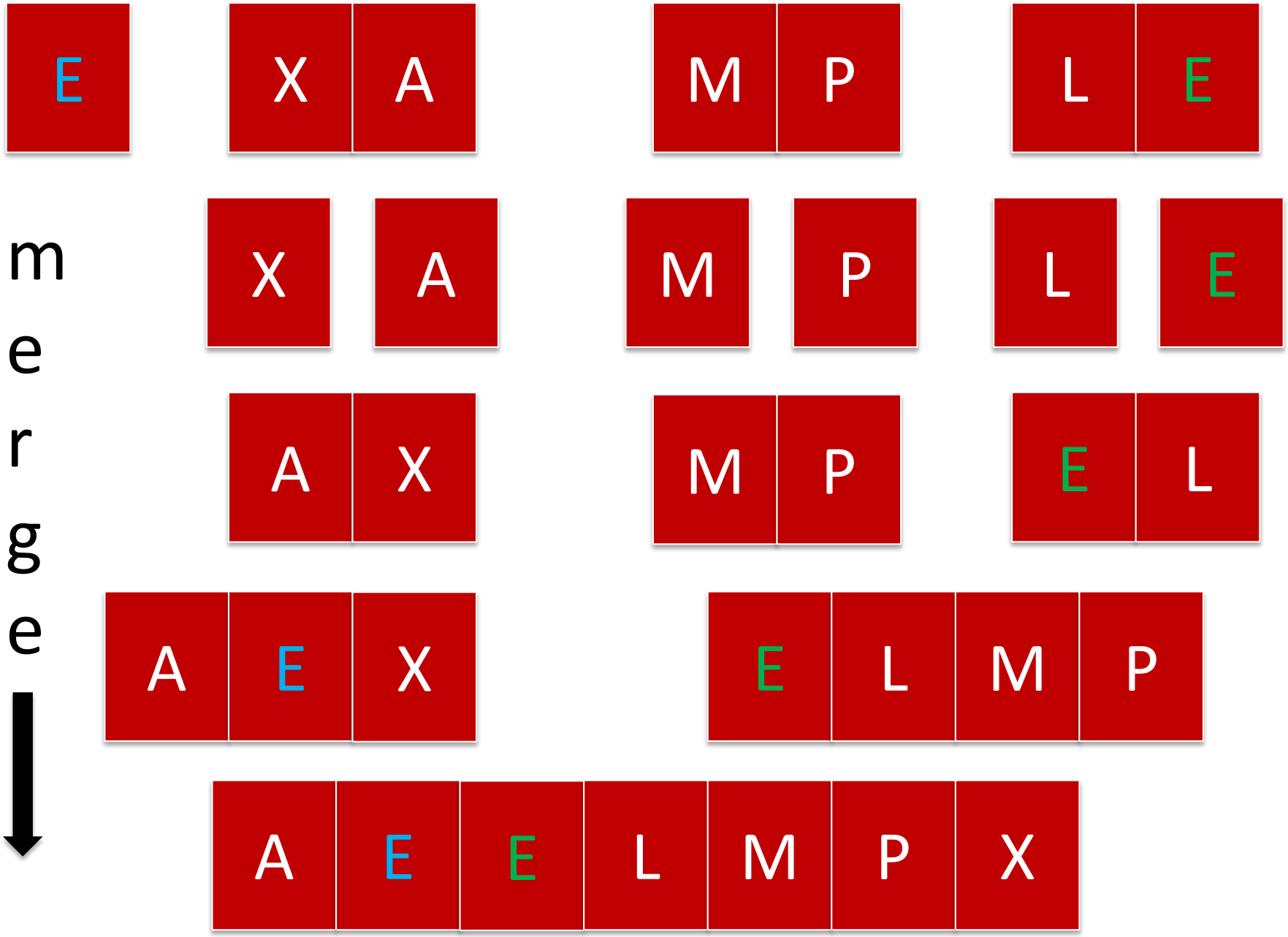
```
17
18 # split the list xs in half.
19 def split(xs, n):
20   return (xs[:n], xs[n:])
21
22 # Divide xs in half until the halves are trivially
23 # This version is inefficient with memory usage.
24 def msort(xs):
25   n = len(xs)
26   if n < 2:
27     return xs
28   else:
29     (left, right) = split(xs, int(n / 2))
30     return merge(msort(left), msort(right))
31
```


E X A M P L E


E X A

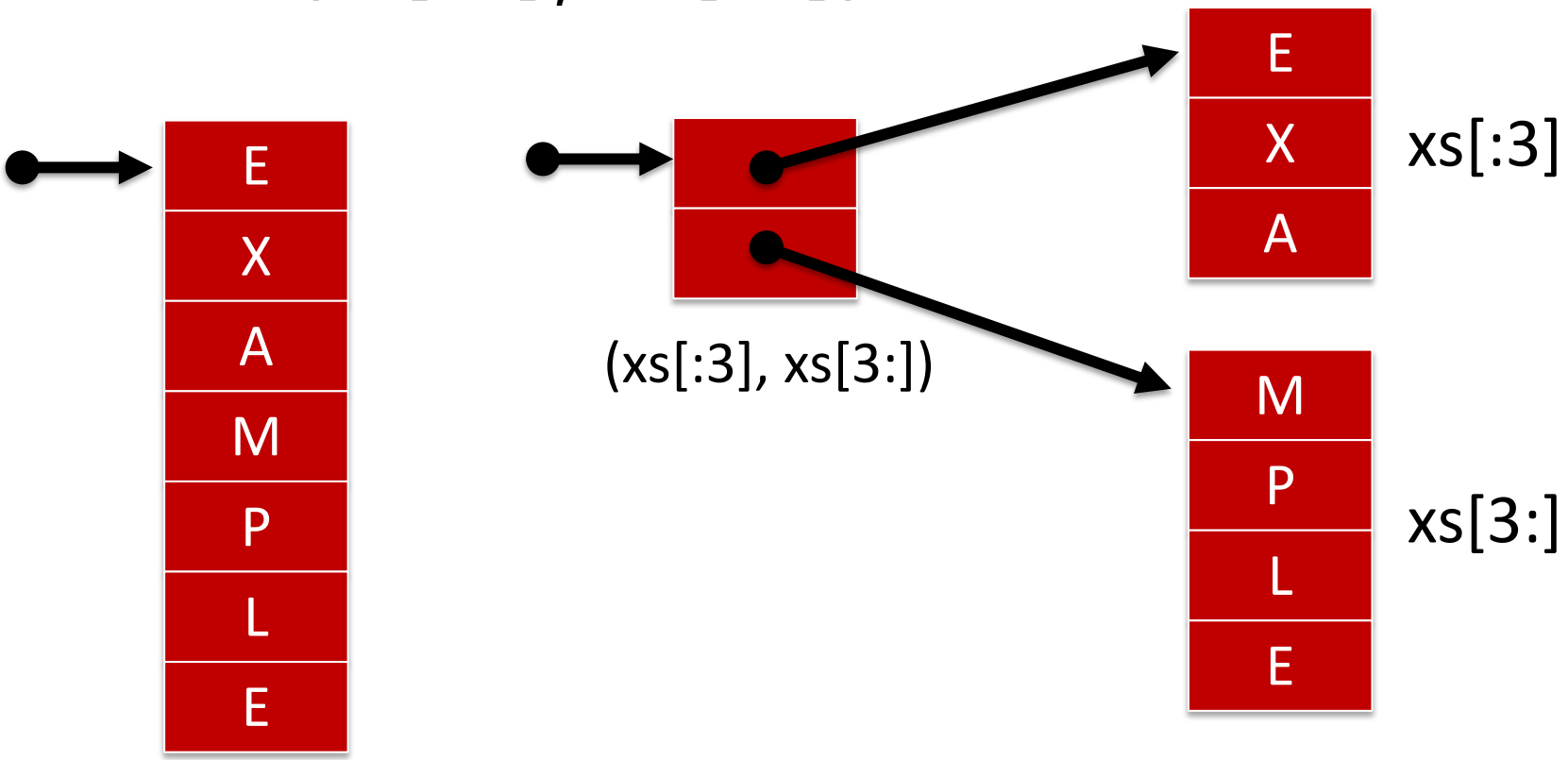

M P L E


divide
E X A M P L E
E X A
M P L E
E
X A
M P
L E
X
A
M
P
L
E

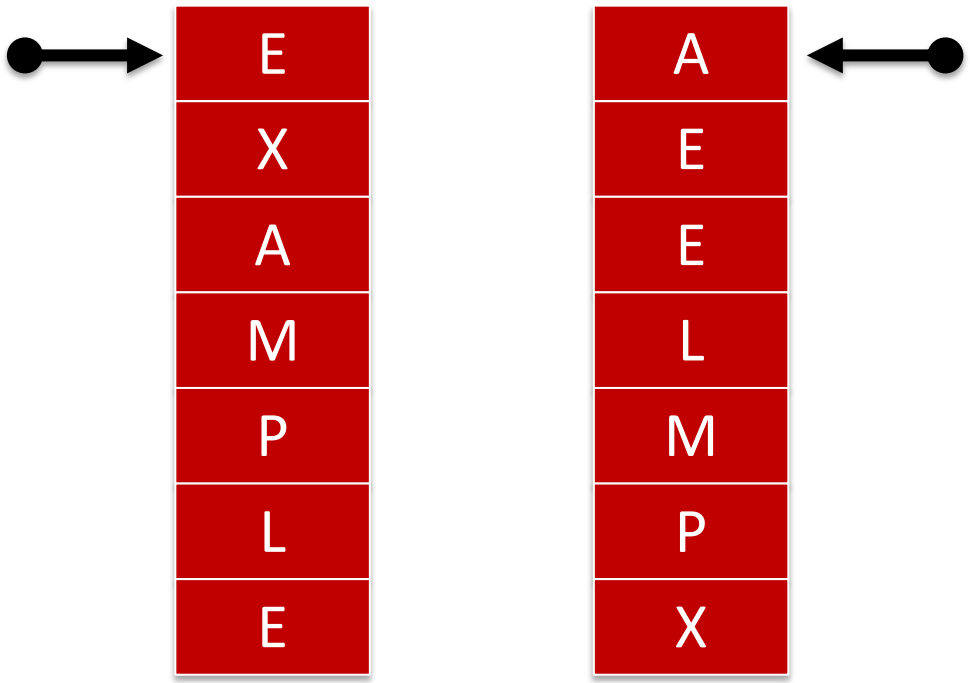

E
X A
M P
L E
merge
X
A
M
P
L
E
A X
M P
E L
A E X
E L M P
A E E L M P X

Memory Usage in Python

```
17
18 # split the list xs in half.
19 def split(xs, n):
20   return (xs[:n], xs[n:])
21
```



Memory Usage



Use another array as a work area.

```java
// stably merge a[lo .. mid] with a[mid+1 ..hi] using aux[lo .. hi]
static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {

  // copy to aux[]
  for (int k = lo; k <= hi; k++)
    aux[k] = a[k];

  // merge back to a[]
  int i = lo, j = mid+1;
  for (int k = lo; k <= hi; k++) {
    if      (i > mid)              a[k] = aux[j++];
    else if (j > hi)               a[k] = aux[i++];
    else if (less(aux[j], aux[i])) a[k] = aux[j++];
    else                           a[k] = aux[i++];
  }
}

// mergesort a[lo..hi] using auxiliary array aux[lo..hi]
static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
  if (hi <= lo) return;
  int mid = lo + (hi - lo) / 2;
  sort(a, aux, lo, mid);
  sort(a, aux, mid + 1, hi);
  merge(a, aux, lo, mid, hi);
}
```

Mergesort

- Requires O(N log(N)) steps on average and in the worst case;
- Sorts using O(N) storage.
- Mergesort is stable.