Chapter 8 Sorting

信息科学与技术学院　　　黄方军

data_structures@163.com

东校区实验中心B502

8.1 Introduction and Notation

We shall study only a few methods in detail, chosen because:

- They are good—each one can be the best choice under some circumstances.
- They illustrate much of the variety appearing in the full range of methods.
- They are relatively easy to write and understand, without too many details to complicate their presentation.

8.1.1 Sortable Lists

```
template <class Record>
class Sortable_list: public List<Record> {
public:                         // Add prototypes for sorting methods here.
private:                        // Add prototypes for auxiliary functions here.
};
```

Every Record *has an associated key of type* Key. *A* Record *can be implicitly converted to the corresponding* Key. *Moreover, the keys (hence also the records) can be compared under the operations* ' < ,' ' > ,' ' >= ,' ' <= ,' ' == ,' *and* ' != .'

8.2.1 Ordered Insertion

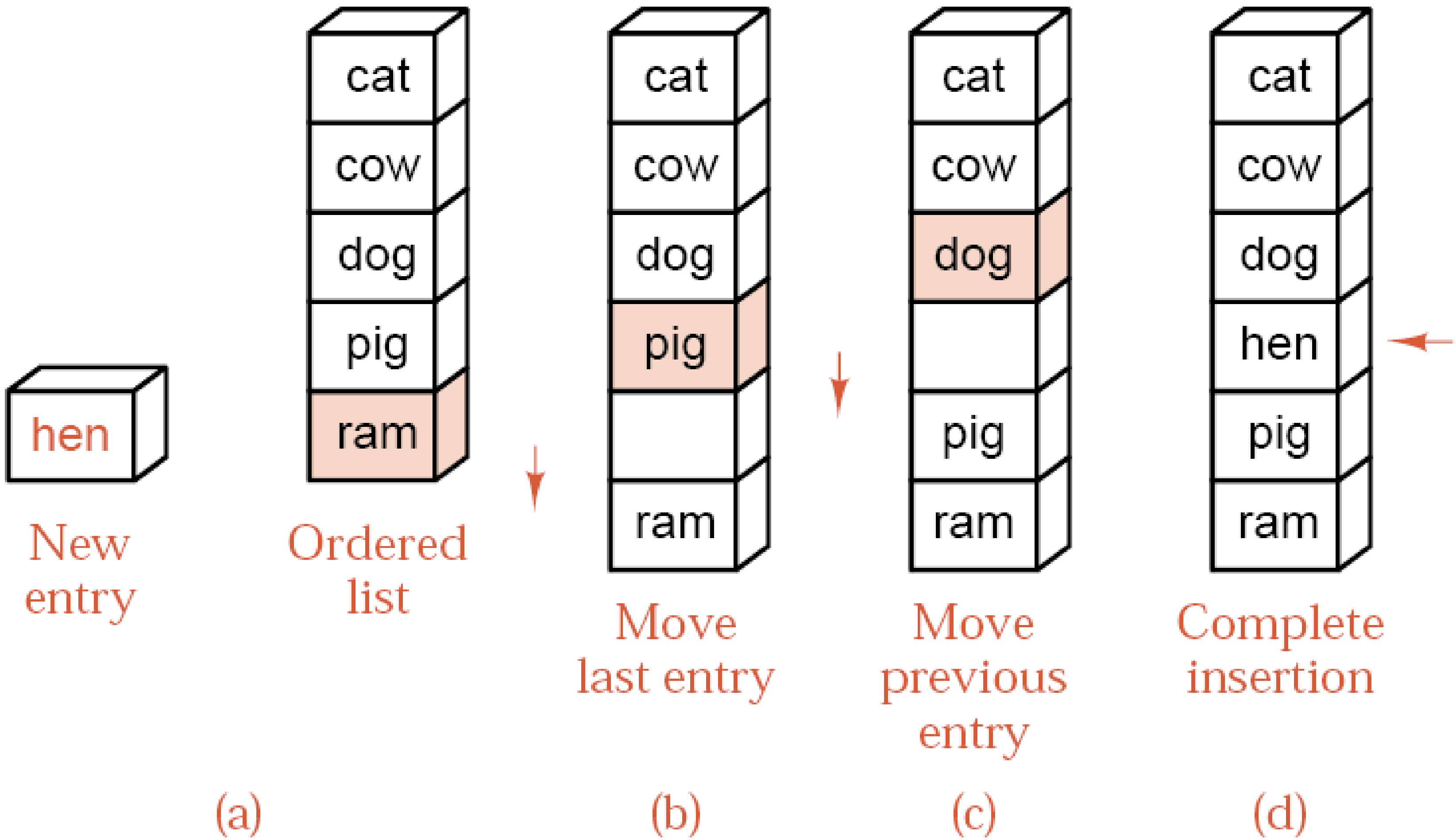


Figure 8.1. Ordered insertion

8.2.2 Sorting by Insertion

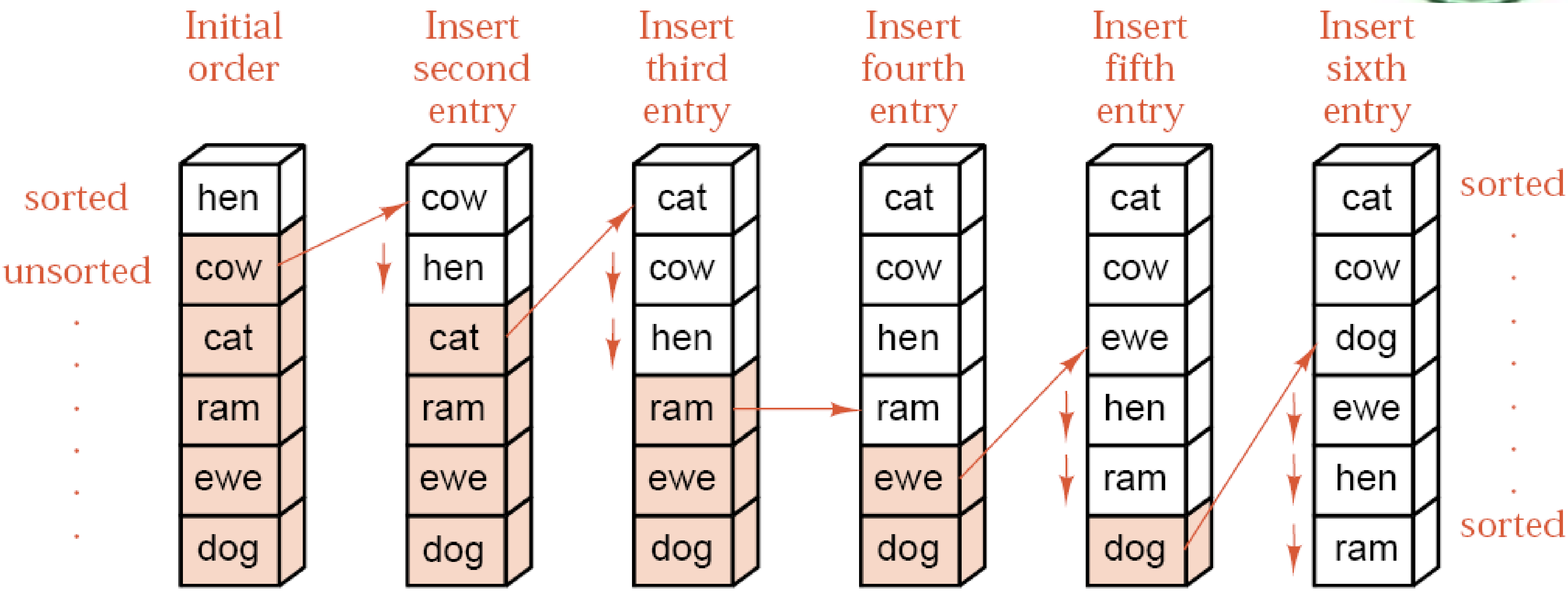


Figure 8.2. Example of insertion sort

8.2.2 Sorting by Insertion

```
template <class Record>
void Sortable_list<Record>::insertion_sort()
{
   int first_unsorted;          //  position of first unsorted entry
   int position;                //  searches sorted part of list
   Record current;              //  holds the entry temporarily removed from list
   for (first_unsorted = 1; first_unsorted < count; first_unsorted++)
      if (entry[first_unsorted] < entry[first_unsorted - 1]) {
         position = first_unsorted;
         current = entry[first_unsorted];   //  Pull unsorted entry out of the list.
         do {                     //  Shift all entries until the proper position is found.
            entry[position] = entry[position - 1];
            position--;                     //  position is empty.
         } while (position > 0 && entry[position - 1] > current);
         entry[position] = current;
      }
}
```

do 循环内一次比较一次数组赋值操作；
do 循环外一次比较两次数组赋值操作。

8.2.2 Sorting by Insertion

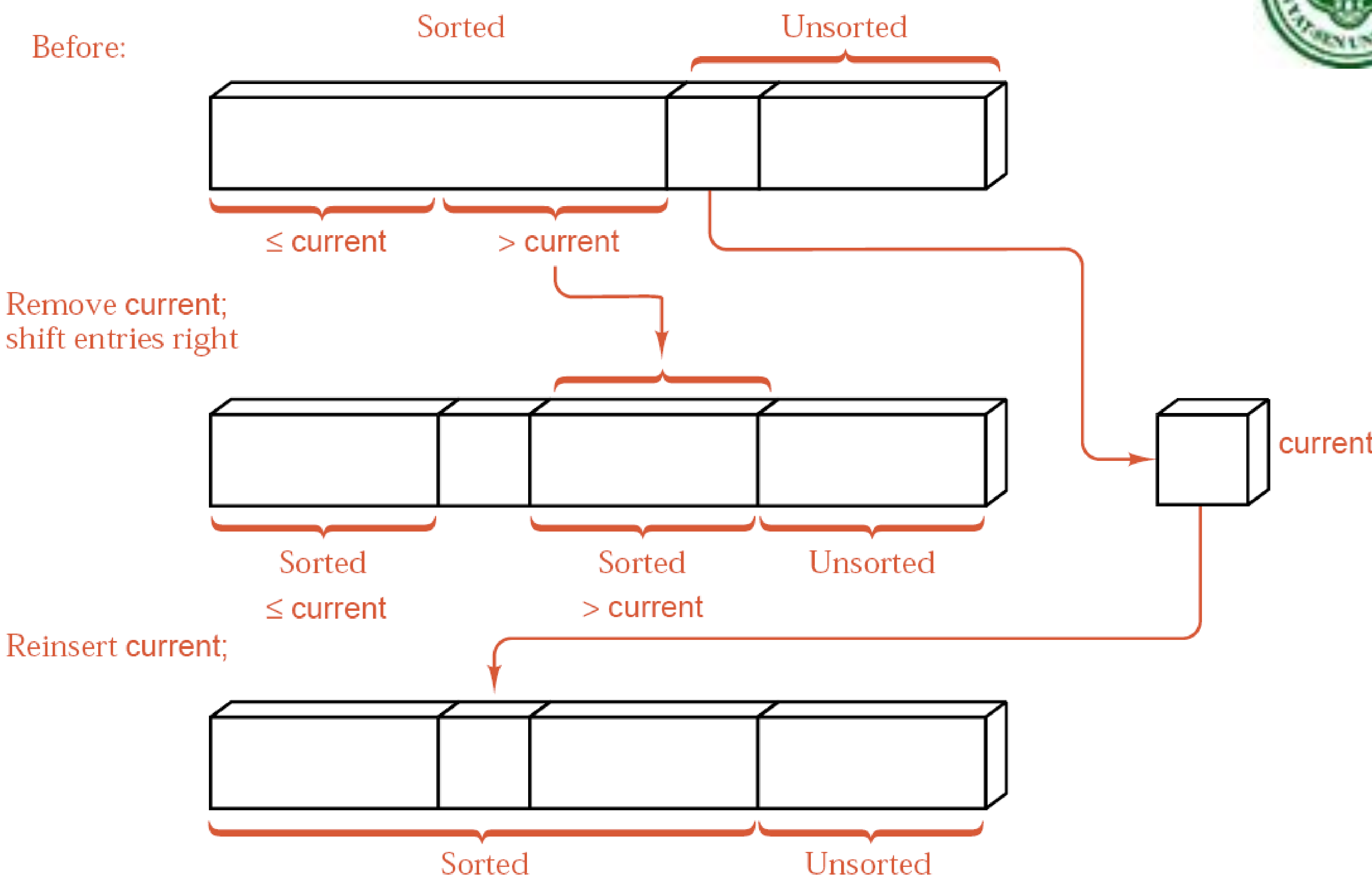


Figure 8.3. The main step of contiguous insertion sort

8.2.3 Linked Version

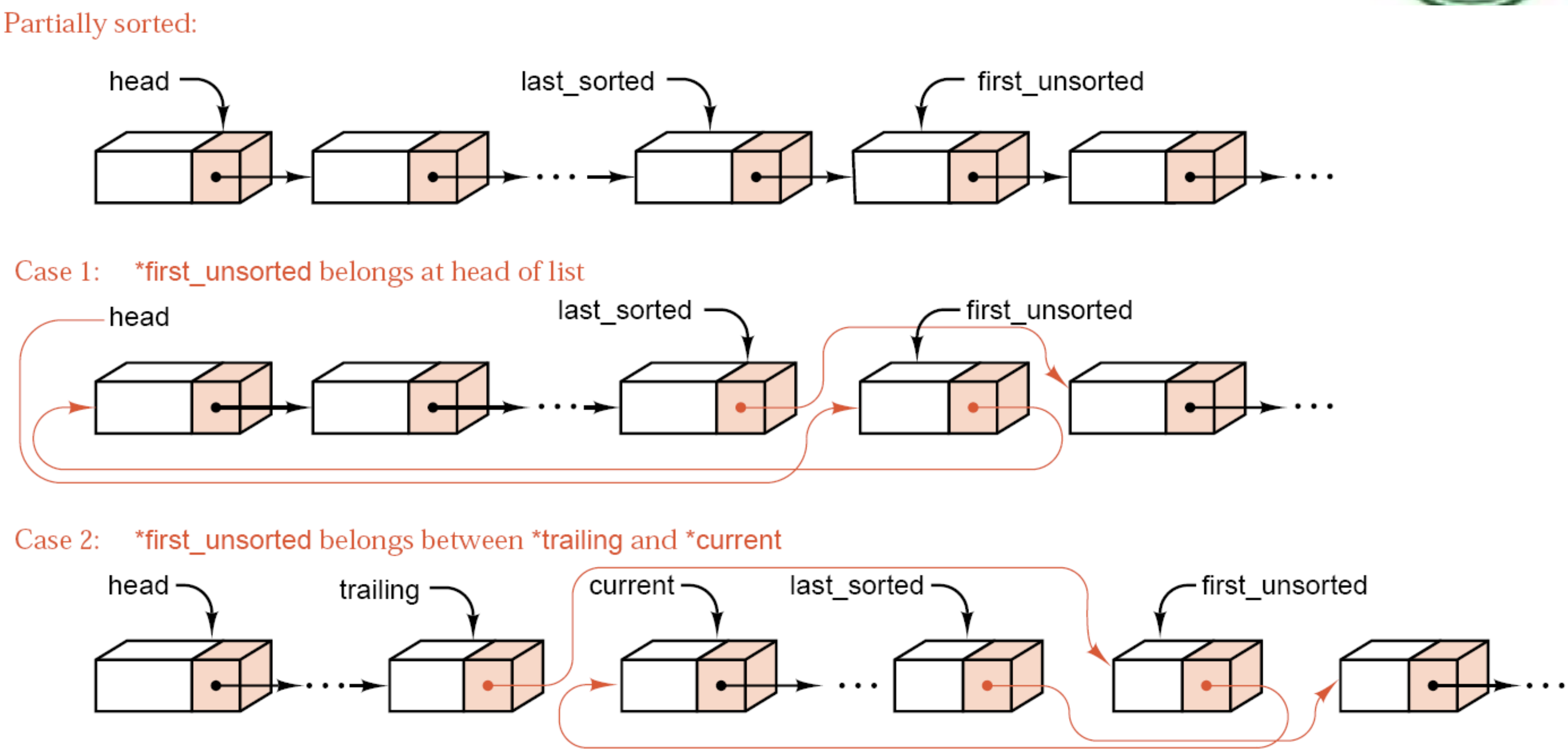


Figure 8.4. Trace of linked insertion sort

8.2.4 Analysis

When we deal with entry i, there are i possible ways to move it: not moving it at all, moving it one position up to moving it i-1 positions to the front of the list.

1 2 3 4 5

∧ ∧ ∧ ∧ i

- The probability that it need not be moved is thus $1/i$, in which case one comparison of keys is done.
- The remaining case, in which entry i must be moved, occurs with probability $(i-1)/i$.

8.2.4 Analysis

Since all of the $i-1$ possible positions are equally likely, the average number of iterations is

$$\frac{1+2+\cdots+(i-1)}{i-1} = \frac{(i-1)i}{2(i-1)} = \frac{i}{2}.$$

One key comparison and one assignment are done for each of these iterations, with one more key comparison done outside the loop, along with two assignments of entries. Hence, in this second case, entry i requires, on average, $\frac{1}{2}i+1$ comparisons and $\frac{1}{2}i+2$ assignments.

参见课本322页程序

8.2.4 Analysis

When we combine the two cases with their respective probabilities, we have

$$\frac{1}{i} \times 1 + \frac{i-1}{i} \times \left(\frac{i}{2} + 1\right) = \frac{i+1}{2}$$

comparisons and

$$\frac{1}{i} \times 0 + \frac{i-1}{i} \times \left(\frac{i}{2} + 2\right) = \frac{i+3}{2} - \frac{2}{i}$$

assignments.

8.2.4 Analysis

To add $\frac{1}{2}i + O(1)$ from $i = 2$ to $i = n$, we apply Theorem A.1 on page 647 (the sum of the integers from 1 to n). We also note that adding n terms, each of which is $O(1)$, produces as result that is $O(n)$. We thus obtain

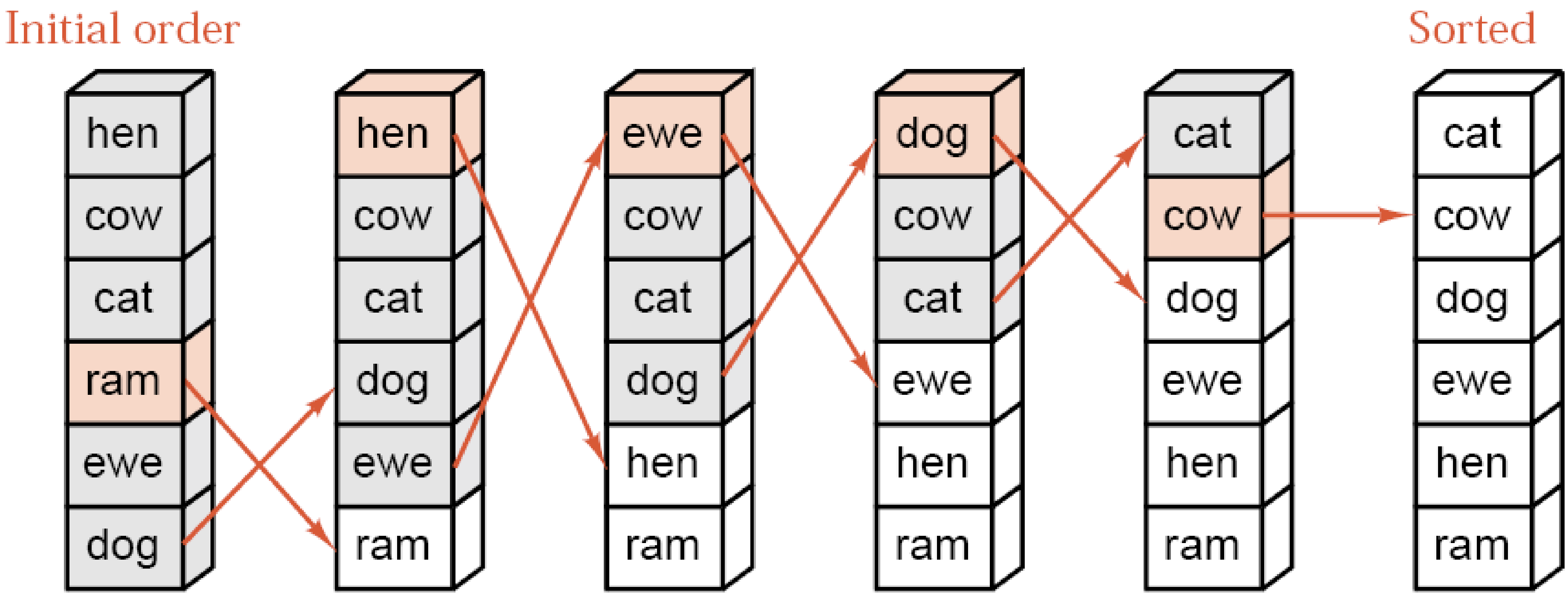
$$\sum_{i=2}^{n}\left(\frac{1}{2}i + O(1)\right) = \frac{1}{2}\sum_{i=2}^{n} i + O(n) = \frac{1}{4}n^2 + O(n)$$

for both the number of comparisons of keys and the number of assignments of entries.

8.3 Selection Sort

- Insertion sort has one major disadvantage. Even after most entries have been sorted properly into the first part of the list, the insertion of a later entry may require that many of them be moved.
- All the moves by insertion sort are moves of only one position at a time.

8.3.1 The Algorithm



Colored box denotes largest unsorted key.
Gray boxes denote other unsorted keys.

Figure 8.5. Example of selection sort

8.3.1 The Algorithm

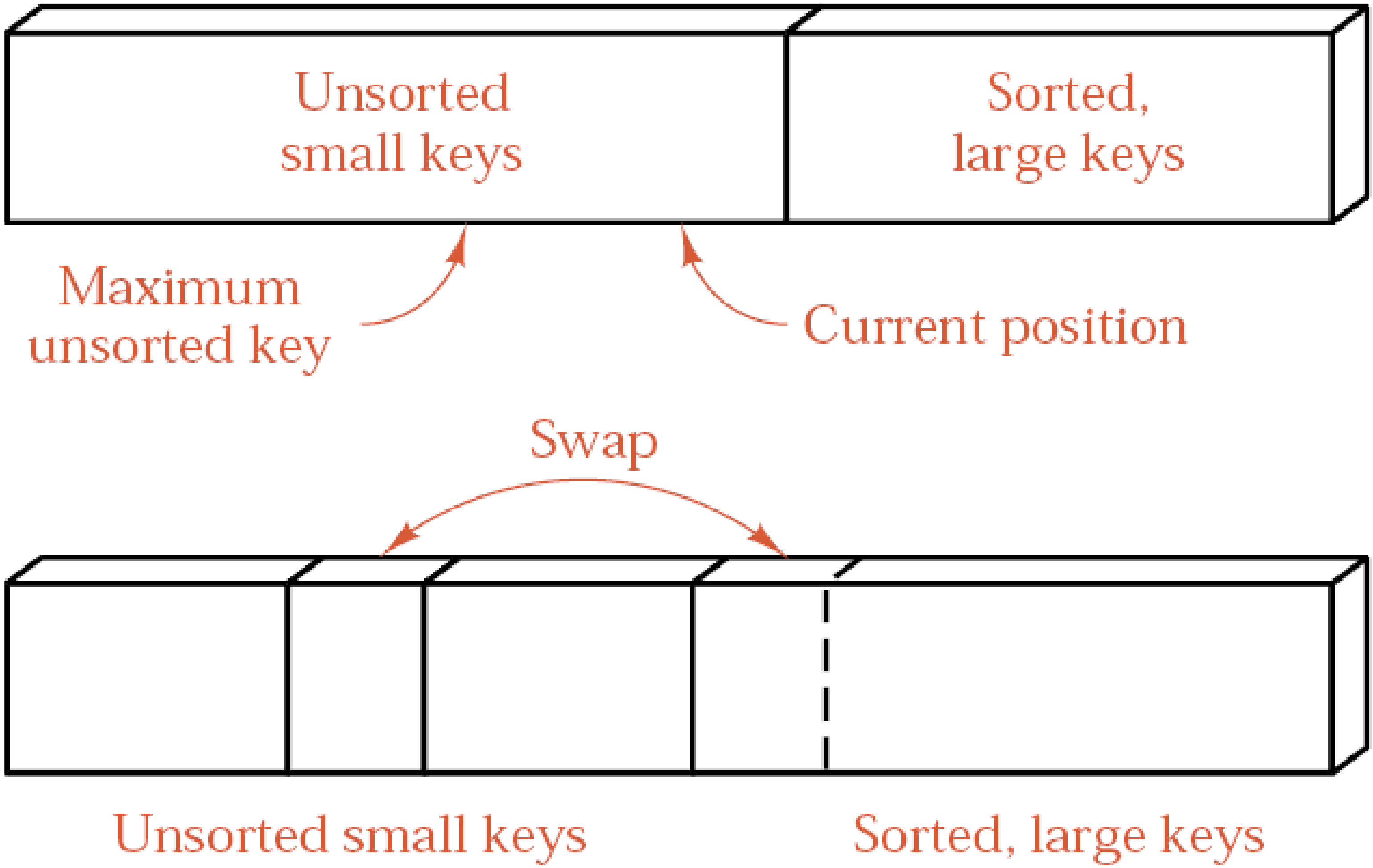


Figure 8.6. The general step in selection sort

8.3.2 Contiguous Implementation

```
template <class Record>
void Sortable_list<Record>::selection_sort()
{
   for (int position = count − 1; position > 0; position−−) {
      int max = max_key(0, position);
      swap(max, position);
   }
}
```

8.3.2 Contiguous Implementation

```
template <class Record>
int Sortable_list<Record>::max_key(int low, int high)
{
   int largest, current;
   largest = low;
   for (current = low + 1; current <= high; current++)
      if (entry[largest] < entry[current])
         largest = current;
   return largest;
}
```

8.3.2 Contiguous Implementation

```
template <class Record>
void Sortable_list<Record>::swap(int low, int high)
{
    Record temp;
    temp = entry[low];
    entry[low] = entry[high];
    entry[high] = temp;
}
```

8.3.3 Analysis

- Selection sort does have the advantage of predictability: Its worst-case time will differ little from its best.
- The function swap is called n-1 times, and each call does 3 assignments of entries, for a total count of 3(n-1).
- There are (n-1)+(n-2)+…+1 = (1/2) n(n-1) comparisons of keys, which we approximate to (1/2) n^2+O(n).

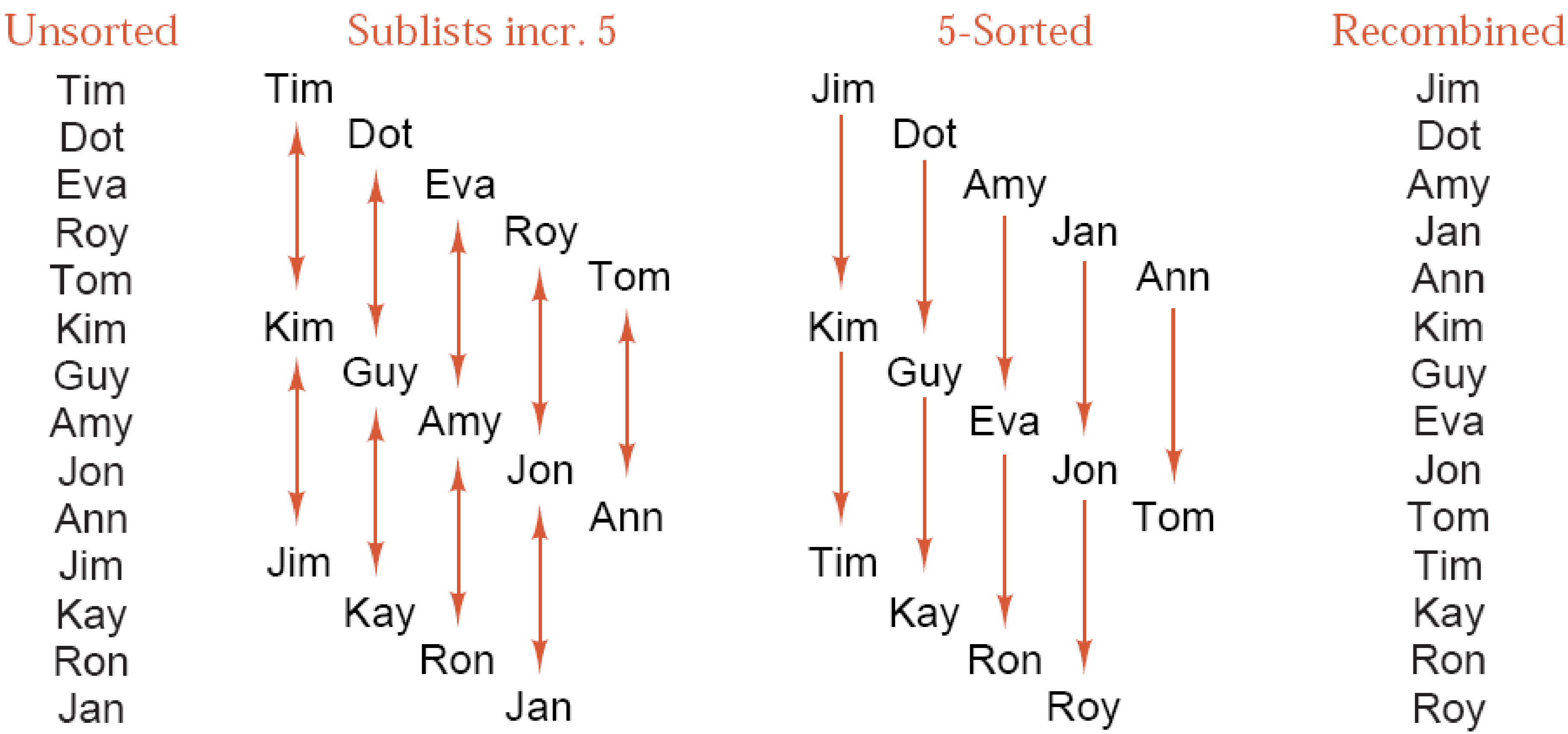
8.3.4 Comparisons

	Selection	*Insertion (average)*
Assignments of entries	$3.0n + O(1)$	$0.25n^2 + O(n)$
Comparisons of keys	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

8.5 Shell Sort

- Selection sort moves the entries very efficiently but does many redundant comparisons.
- In its best case, insertion sort does the minimum number of comparisons, but it is inefficient in moving entries only one position at a time.

8.5 Shell Sort



8.4 Shell Sort

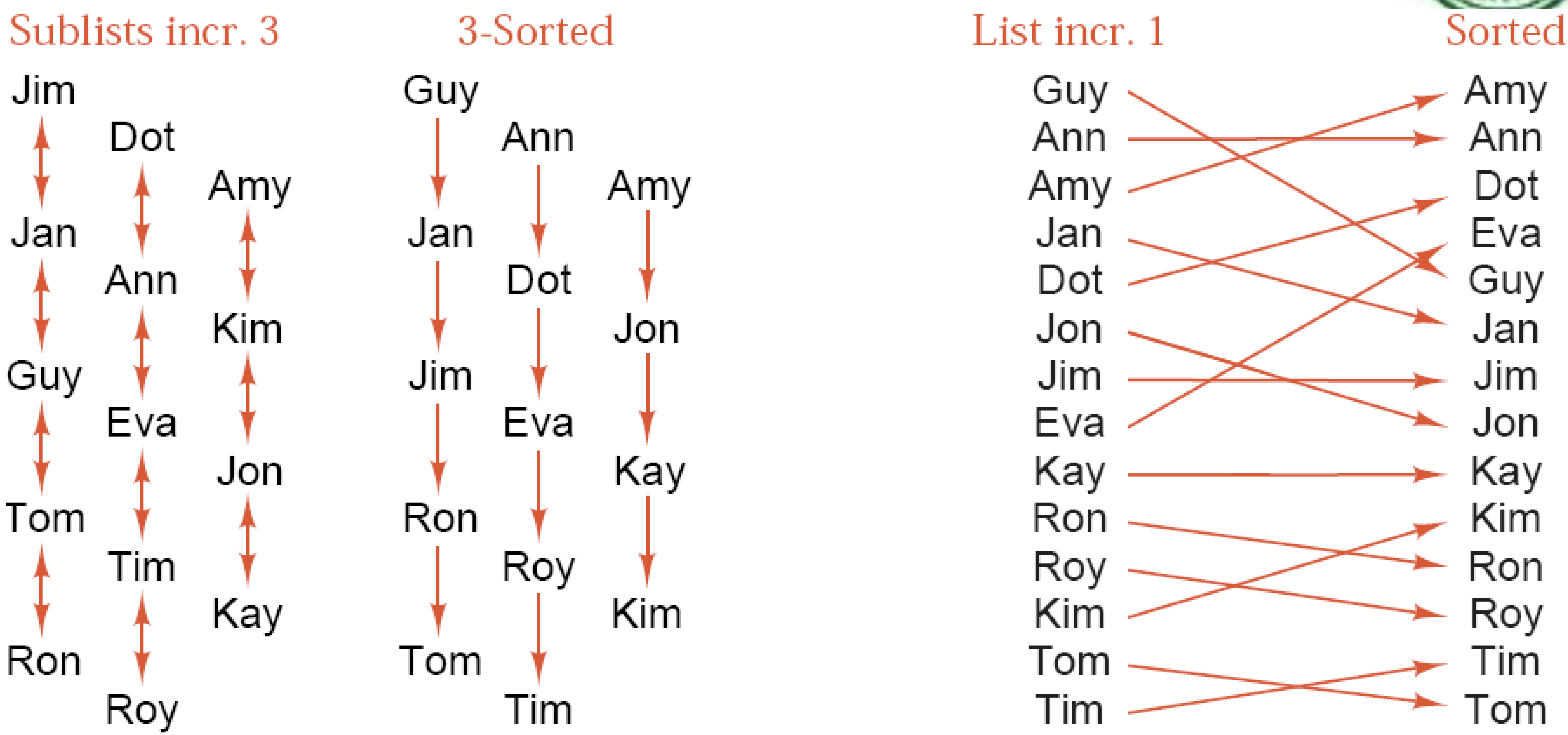
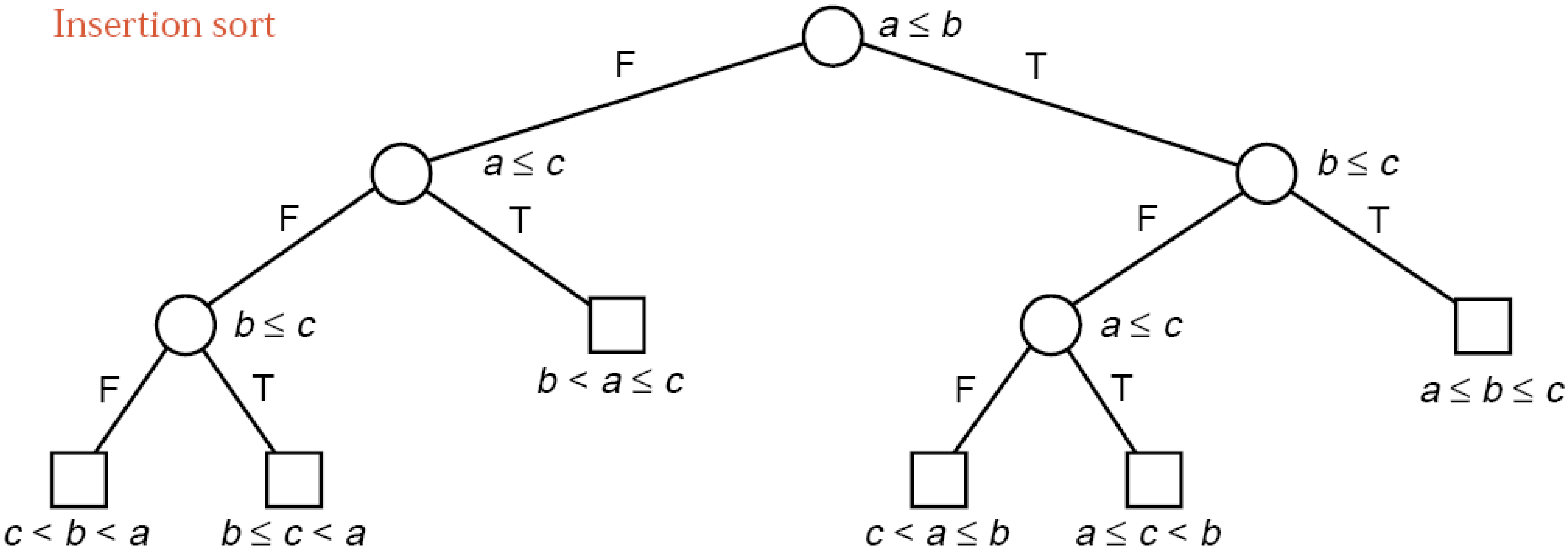


Figure 8.7. Example of Shell sort

8.5 Lower Bounds

Insertion sort



8.5 Lower Bounds

Selection sort

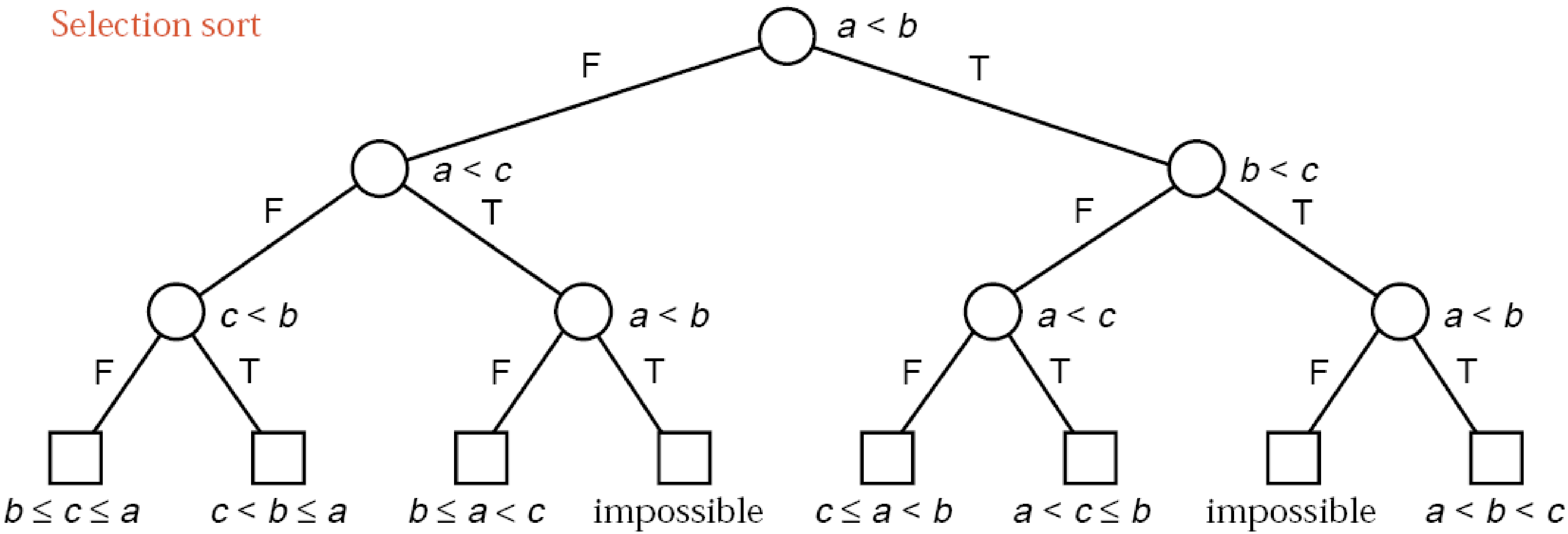


Figure 8.8. Comparison trees, insertion and selection sort, $n = 3$

8.5 Lower Bounds

Any algorithm that sorts a list of n entries by use of key comparisons must, in its worst case, perform at least $\lceil \lg n! \rceil$ comparisons of keys, and, in the average case, it must perform at least $\lg n!$ comparisons of keys.

$$\lg n! \approx (n + \tfrac{1}{2}) \lg n - (\lg e) n + \lg \sqrt{2\pi} + \frac{\lg e}{12n}.$$

$$\lg n! \approx (n + \tfrac{1}{2})(\lg n - 1\tfrac{1}{2}) + 2$$

$$\lg n! = n \lg n + O(n)$$

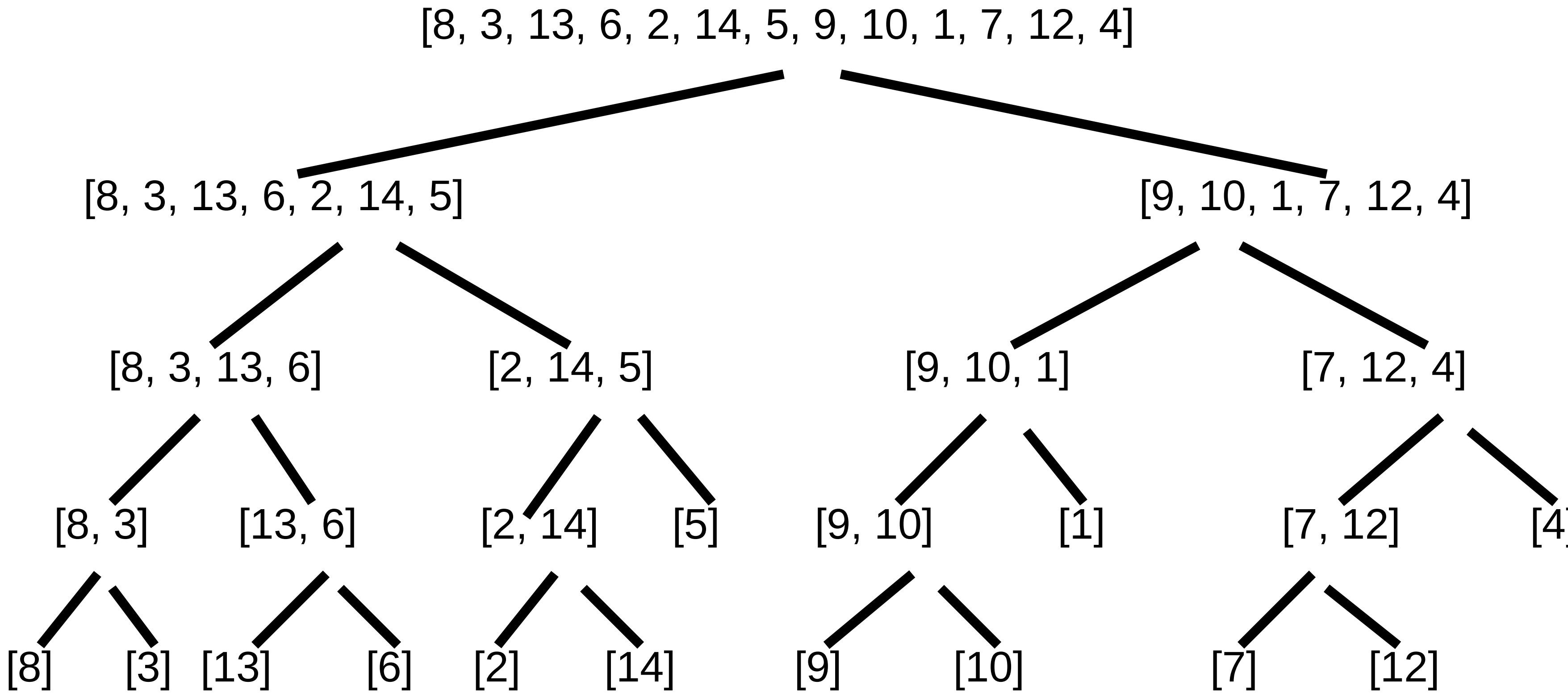
8.6 Divide-and-Conquer Sorting

```
void Sortable_list :: sort( )
{
   if the list has length greater than 1 {
      partition the list into lowlist, highlist;
      lowlist.sort( );
      highlist.sort( );
      combine(lowlist, highlist);
   }
}
```

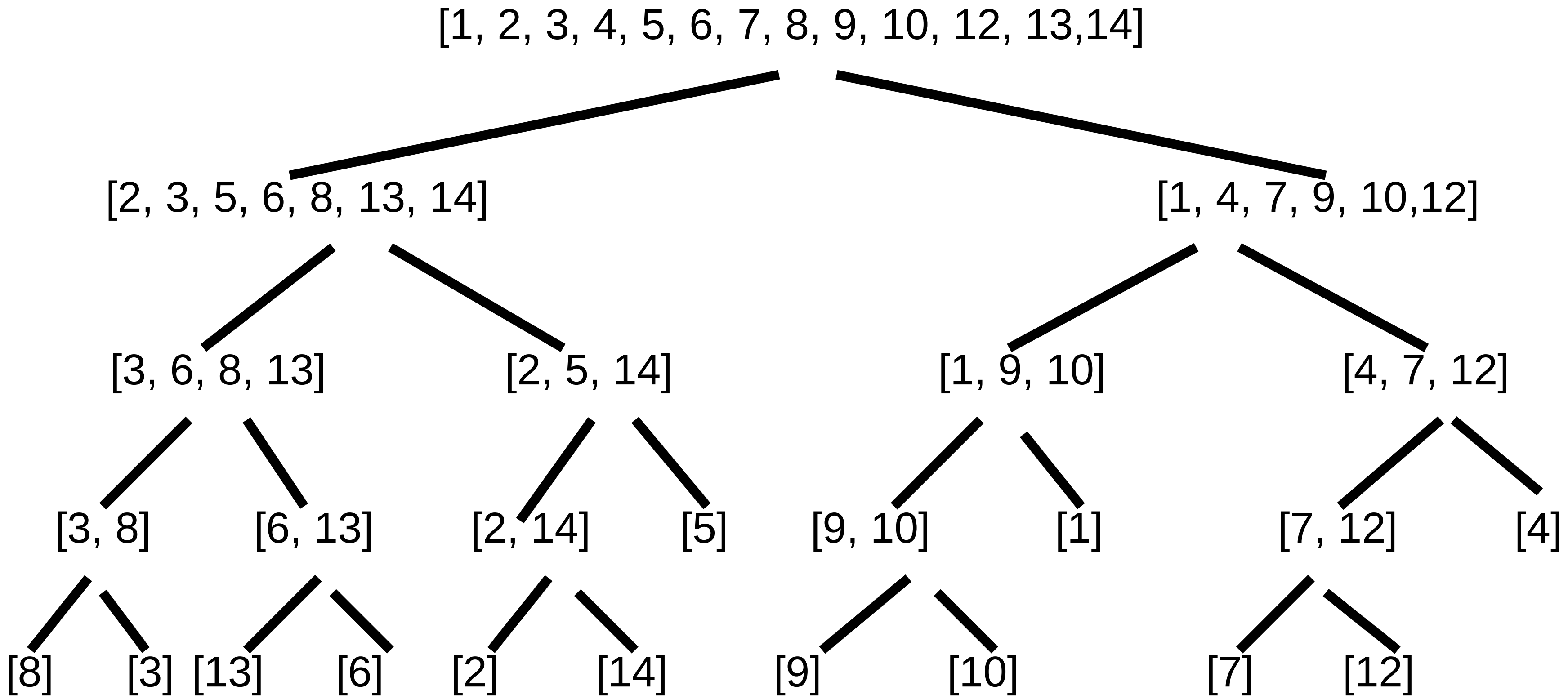
Two methods:

- Mergesort
- Quicksort

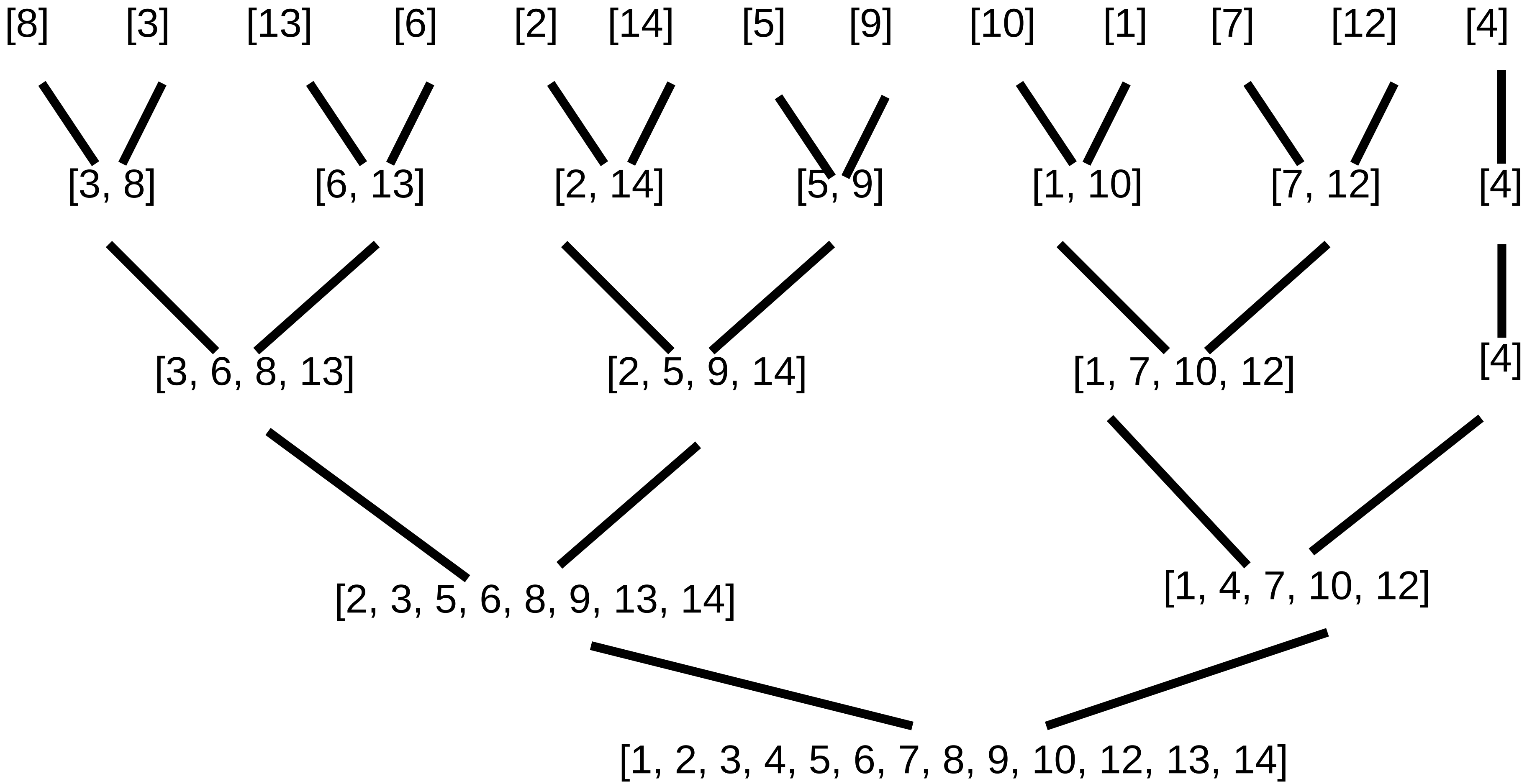
Merge Sort


[8, 3, 13, 6, 2, 14, 5, 9, 10, 1, 7, 12, 4]
[8, 3, 13, 6, 2, 14, 5]
[9, 10, 1, 7, 12, 4]
[8, 3, 13, 6]
[2, 14, 5]
[9, 10, 1]
[7, 12, 4]
[8, 3]
[13, 6]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

Merge Sort


[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13,14]
[2, 3, 5, 6, 8, 13, 14]
[1, 4, 7, 9, 10,12]
[3, 6, 8, 13]
[2, 5, 14]
[1, 9, 10]
[4, 7, 12]
[3, 8]
[6, 13]
[2, 14]
[5]
[9, 10]
[1]
[7, 12]
[4]
[8]
[3]
[13]
[6]
[2]
[14]
[9]
[10]
[7]
[12]

Nonrecursive Merge Sort



Quick Sort

- Small instance has n <= 1. Every small instance is a sorted instance.
- To sort a large instance, select a pivot element from out of the *n* elements.
- Partition the *n* elements into 3 groups left, middle and right.
- The middle group contains only the pivot element.
- All elements in the left group are <= pivot.
- All elements in the right group are > pivot.
- Sort left and right groups recursively.
- Answer is sorted left group, followed by middle group followed by sorted right group.

Example

6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot.

2	5	4	1	3	6	7	9	10	11	8

Sort left and right groups recursively.

1	2	5	4	3	6	7	9	10	11	8

8.7 Mergesort for Linked Lists

In the case of mergesort, we shall write a version for linked lists and leave the case of contiguous lists as an exercise.

For quicksort, we shall do the reverse, writing the code only for contiguous lists.

Both of these methods, however, work well for both contiguous and linked lists.

8.7.1 The Functions

```
template <class Record>
void Sortable_list<Record>::recursive_merge_sort(Node<Record> * &sub_list)
/* Post: The nodes referenced by sub_list have been rearranged so that their keys
         are sorted into nondecreasing order. The pointer parameter sub_list is
         reset to point at the node containing the smallest key.
   Uses: The linked List implementation of Chapter 6

{
   if (sub_list != NULL && sub_list->next != NULL) {
      Node<Record> *second_half = divide_from(sub_list);
      recursive_merge_sort(sub_list);
      recursive_merge_sort(second_half);
      sub_list = merge(sub_list, second_half);
   }
}
```

8.7.1 The Functions

```
template <class Record>
Node<Record> *Sortable_list<Record>::divide_from(Node<Record> *sub_list)
{
   Node<Record> *position,   //   traverses the entire list
                *midpoint,   //   moves at half speed of position to midpoint
                *second_half;
   if ((midpoint = sub_list) == NULL) return NULL;   //   List is empty.
   position = midpoint->next;
   while (position != NULL) {   //   Move position twice for midpoint's one move.
      position = position->next;
      if (position != NULL) {
         midpoint = midpoint->next;
         position = position->next;
      }
   }
   second_half = midpoint->next;
   midpoint->next = NULL;
   return second_half;
}
```

8.7.1 The Functions

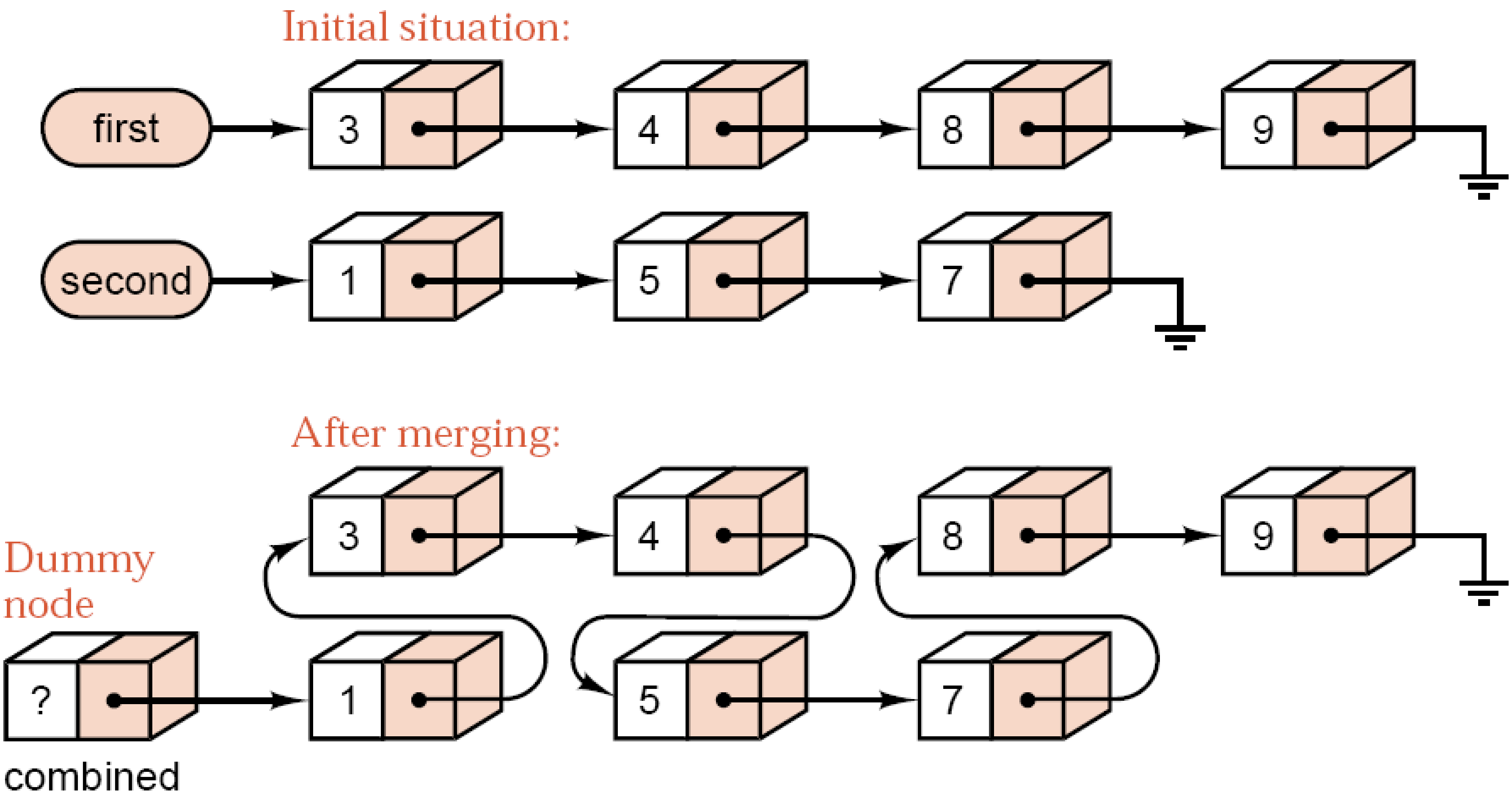


Figure 8.13. Merging two sorted linked lists

8.7.1 The Functions

```
template <class Record>
Node<Record> *Sortable_list<Record>::merge(Node<Record> *first,
                                           Node<Record> *second)
{
  Node<Record> *last_sorted;  //  points to the last node of sorted list
  Node<Record> combined;      //  dummy first node, points to merged list
  last_sorted = &combined;
  while (first != NULL && second != NULL) { //  Attach node with smaller key
    if (first->entry <= second->entry) {
      last_sorted->next = first;
      last_sorted = first;
      first = first->next;    //  Advance to the next unmerged node.
    }
    else {
      last_sorted->next = second;
      last_sorted = second;
      second = second->next;
    }
//  After one list ends, attach the remainder of the other.
  if (first == NULL)
    last_sorted->next = second;
  else
    last_sorted->next = first;
  return combined.next;
}
```

8.7.2 Analysis of Mergesort

1. Counting Comparisons

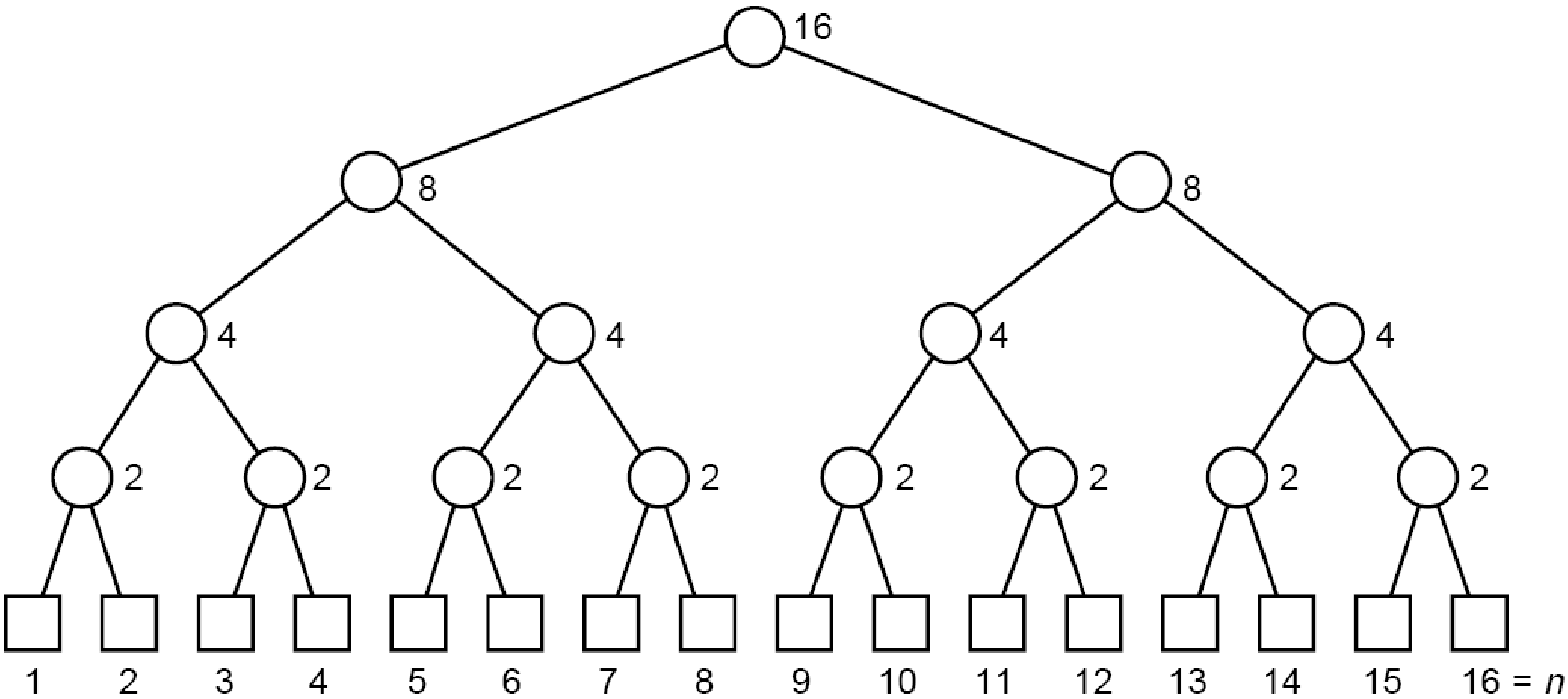


Figure 8.14. Lengths of sublist merges

比较次数不会超过$\lceil nlgn \rceil$

8.7.2 Analysis of Mergesort

2. Contrast with Insertion Sort

	Selection	*Insertion (average)*
Assignments of entries	$3.0n + O(1)$	$0.25n^2 + O(n)$
Comparisons of keys	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

The appearance of the expression $n \lg n$ in the preceding calculation is by no means accidental, but relates closely to the lower bounds established in Section 8.5, where it was proved that any sorting method that uses comparisons of keys must do at least

$$\lg n! \approx n \lg n - 1.44n + O(\log n)$$

comparisons of keys. When n is large, the first term of this expression becomes more important than what remains. We have now found, in mergesort, an algorithm that comes within reach of this lower bound.

8.7.2 Analysis of Mergesort

3. Improving the Count

- 针对图8.14的每一层，比较次数共减少

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1 = n - 1.$$

- 固总比较次数小于

$$n \lg n - n + 1.$$

- 另外还考虑到尾部追加的情况（即从某一个元素开始，序列2的所有元素比序列1的最大元素还要大）

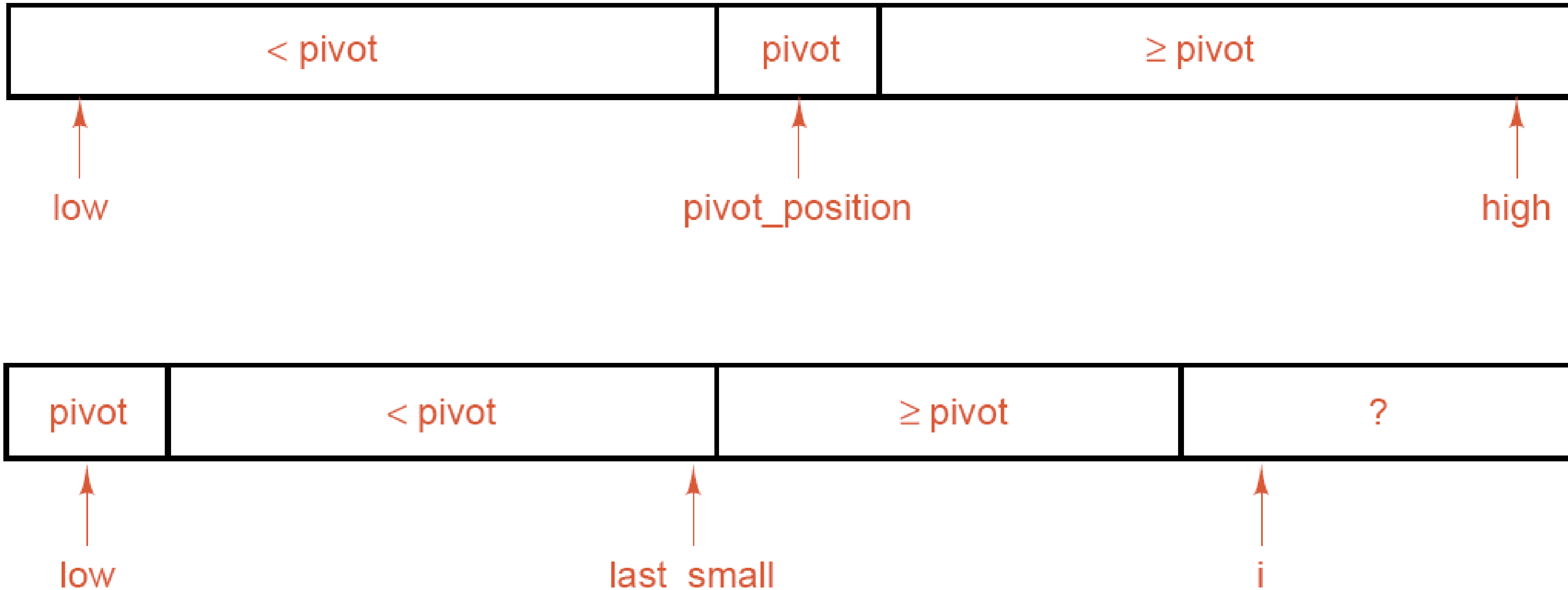
8.8 Quicksort for Contiguous Lists

- The most important applications of quicksort are to contiguous lists, where it can prove to be very fast
- and where is has the advantage over contiguous mergesort of not requiring a choice between using substantial extra space for an auxiliary array or investing great programming effort in implementing a complicated and difficult merge algorithm.

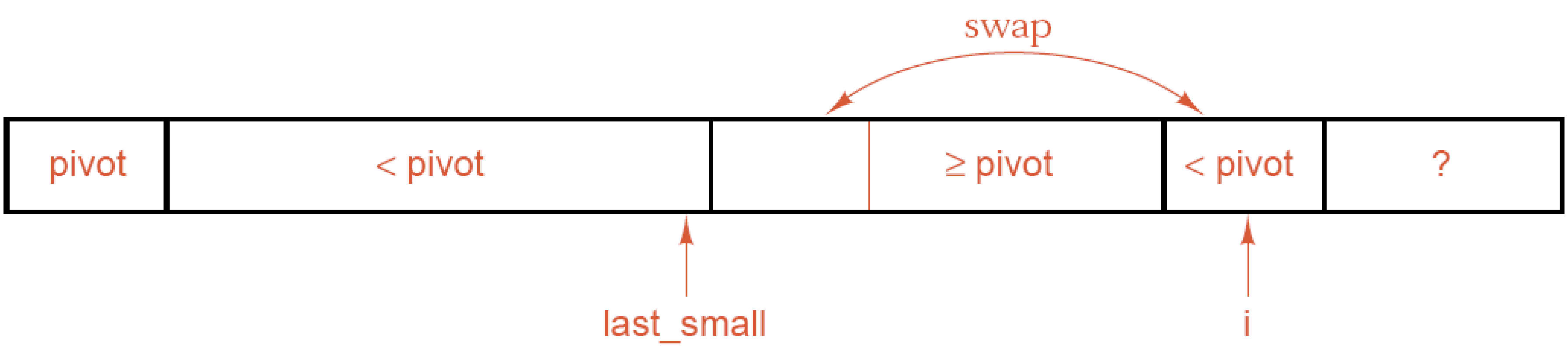
8.8.1 The Main Function

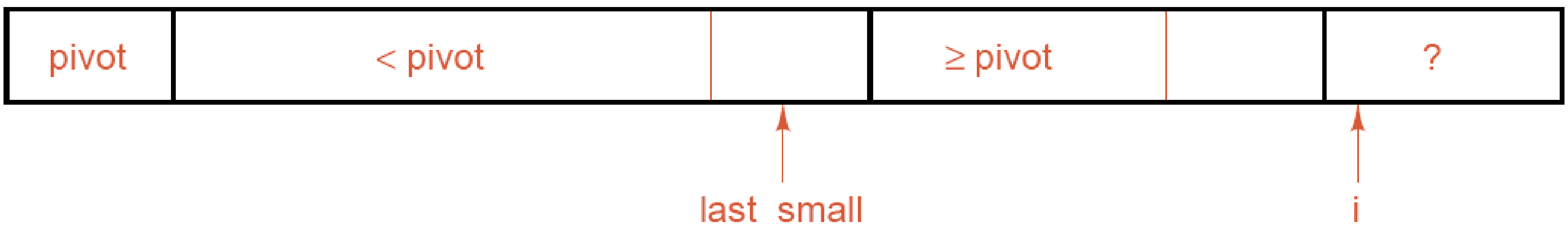
```
template <class Record>
void Sortable_list<Record>::quick_sort( )
{
   recursive_quick_sort(0, count − 1);
}

template <class Record>
void Sortable_list<Record>::recursive_quick_sort(int low, int high)
{
   int pivot_position;
   if (low < high) {               //   Otherwise, no sorting is needed.
      pivot_position = partition(low, high);
      recursive_quick_sort(low, pivot_position − 1);
      recursive_quick_sort(pivot_position + 1, high);
   }
}
```

8.8.2 Partitioning the List

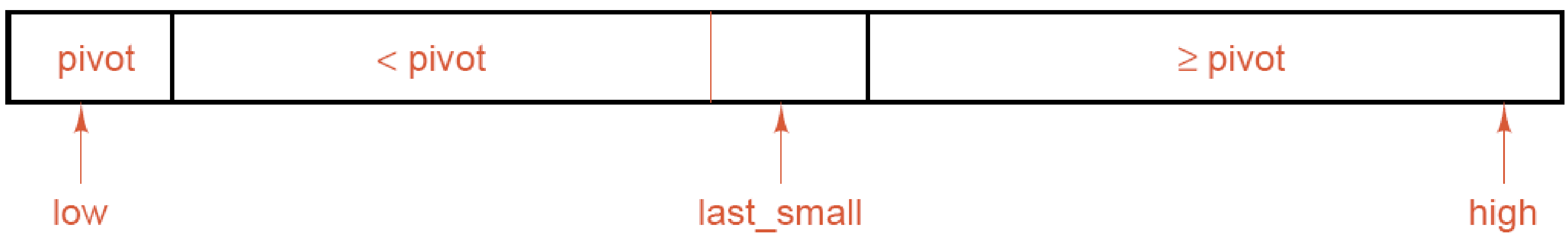


8.8.2 Partitioning the List





When the loop terminates, we have the situation:



8.8.2 Partitioning the List

```
template <class Record>
int Sortable_list<Record>::partition(int low, int high)
{
   Record pivot;
   int i,                         //   used to scan through the list
       last_small;                //   position of the last key less than pivot
   swap(low, (low + high)/2);
   pivot = entry[low];            //   First entry is now pivot.
   last_small = low;
   for (i = low + 1; i <= high; i++)
       if (entry[i] < pivot) {
          last_small = last_small + 1;
          swap(last_small, i);    //   Move large entry to right and small to left.
       }
    swap(low, last_small);        //   Put the pivot into its proper position.
    return last_small;
 }
```

8.8.3 Analysis of Quicksort

1. Choice of Pivot

---选择中间元素

2. Count of Comparisons

The partition function compares the pivot with every other key in the list exactly once, and thus the function partition accounts for exactly $n - 1$ key comparisons. If one of the two sublists it creates has length r, then the other sublist will have length exactly $n - r - 1$. The number of comparisons done in the two recursive calls will then be $C(r)$ and $C(n - r - 1)$. Thus we have

$$C(n) = n - 1 + C(r) + C(n - r - 1).$$

8.8.3 Analysis of Quicksort

3. Comparison Count, Worst

$$
\begin{aligned}
C(1) &= 0. \\
C(2) &= 1 + C(1) &&= 1. \\
C(3) &= 2 + C(2) &&= 2 + 1. \\
C(4) &= 3 + C(3) &&= 3 + 2 + 1. \\
&\ \vdots &&\ \vdots \\
C(n) &= n - 1 + C(n-1) &&= (n-1) + (n-2) + \cdots + 2 + 1 \\
& &&= \tfrac{1}{2}(n-1)n = \tfrac{1}{2}n^2 - \tfrac{1}{2}n.
\end{aligned}
$$

8.8.3 Analysis of Quicksort

4. Swap Count, Worst Case

The partition function does one swap inside its loop for each key less than the pivot and two swaps outside its loop (如果pivot是List中最大值，需*n*-1次swap).

$$S(n)= n + 1 + S(n - 1)$$

$$S(n)= (n + 1)+n + \cdots + 3 = \frac{1}{2}(n + 1)(n + 2)-3 = 0.5n^2 + 1.5n - 1$$

8.8.4 Average-Case Analysis of Quicksort

1. Counting Swaps

n 2 n-1 3 … *p*-1, *p*, *p*+1,…1

the pivot for the first partition is p

$$S(n,p) = (p+1) + S(p-1) + S(n-p).$$

by adding them from $p = 1$ to $p = n$ and dividing by n

$$S(n) = \frac{n}{2} + \frac{3}{2} + \frac{2}{n}\Big(S(0) + S(1) + \cdots + S(n-1)\Big)$$

$$S(n) \approx 0.69(n \lg n) + O(n).$$

8.8.3 Analysis of Quicksort

3. Counting Comparisons

Since a call to the partition function for a list of length n makes exactly $n-1$ comparisons,

$$C(n,p) = n - 1 + C(p-1) + C(n-p).$$

When we average these expressions for $p = 1$ to $p = n$, we obtain

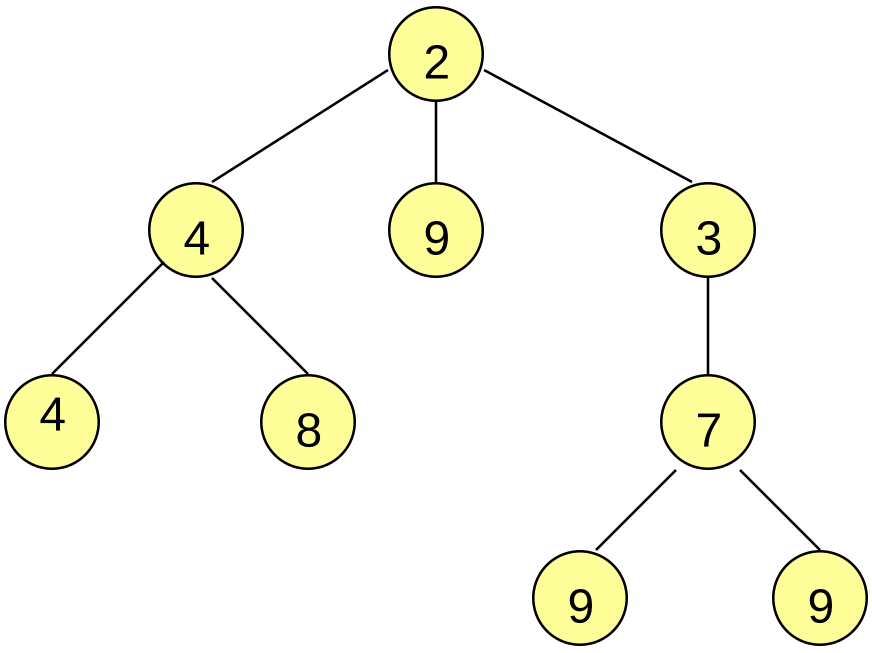
$$C(n) = n + \frac{2}{n}\Big(C(0) + C(1) + \cdots + C(n-1)\Big).$$

$$C(n) \approx 2n \ln n + O(n) \approx 1.39 n \lg n + O(n).$$

8.9 Heaps and Heapsort

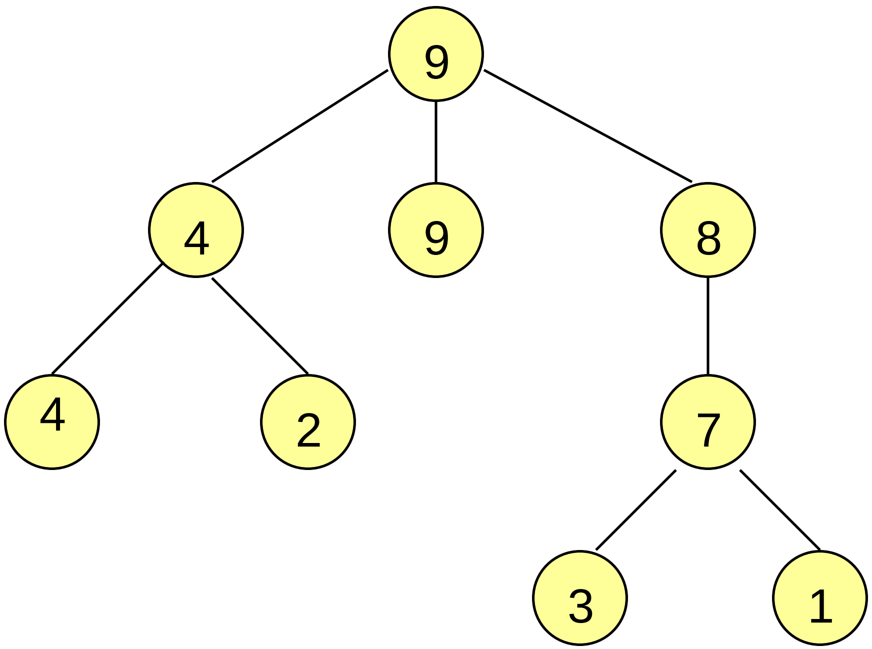
- Max tree
- Min tree
- Max heap
- Min Heap

Min Tree Example



Root has minimum element.

Max Tree Example

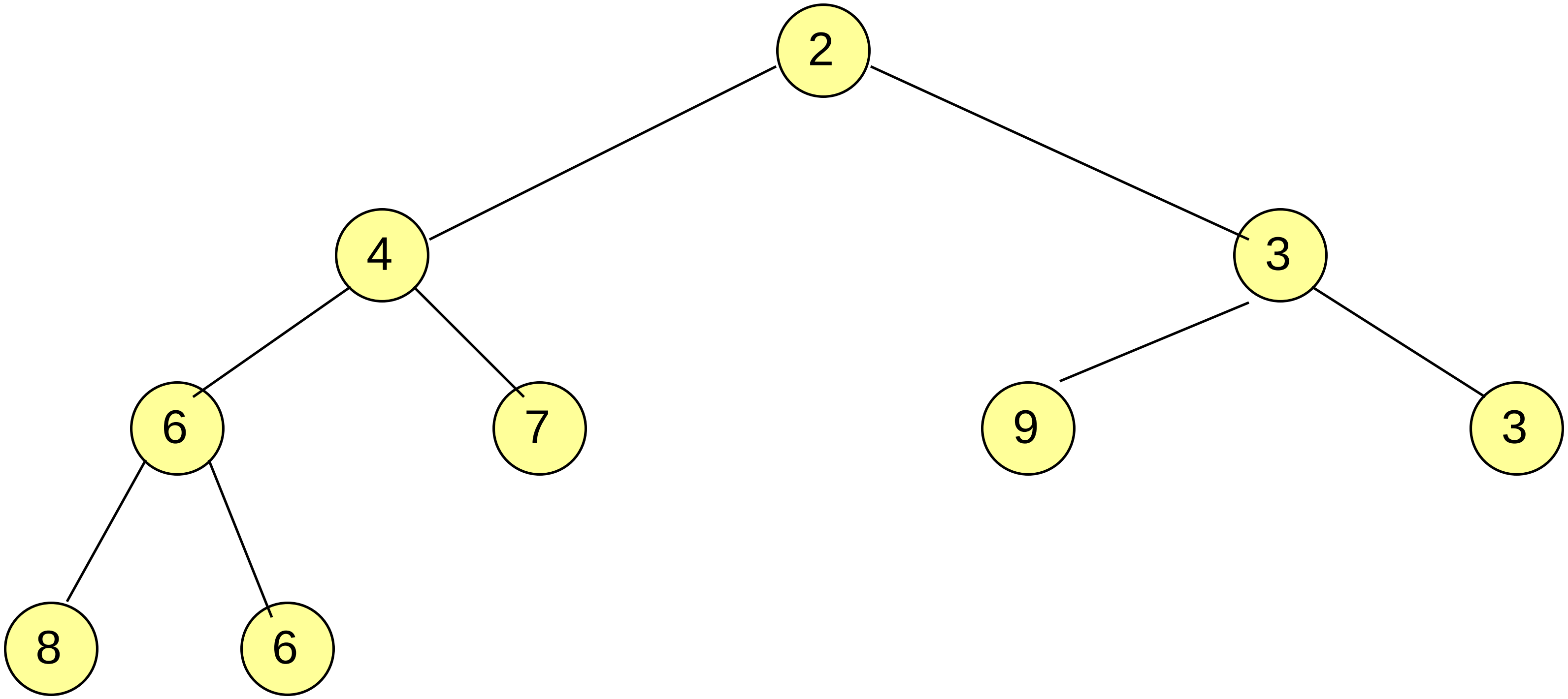


Root has maximum element.

Min Heap Definition

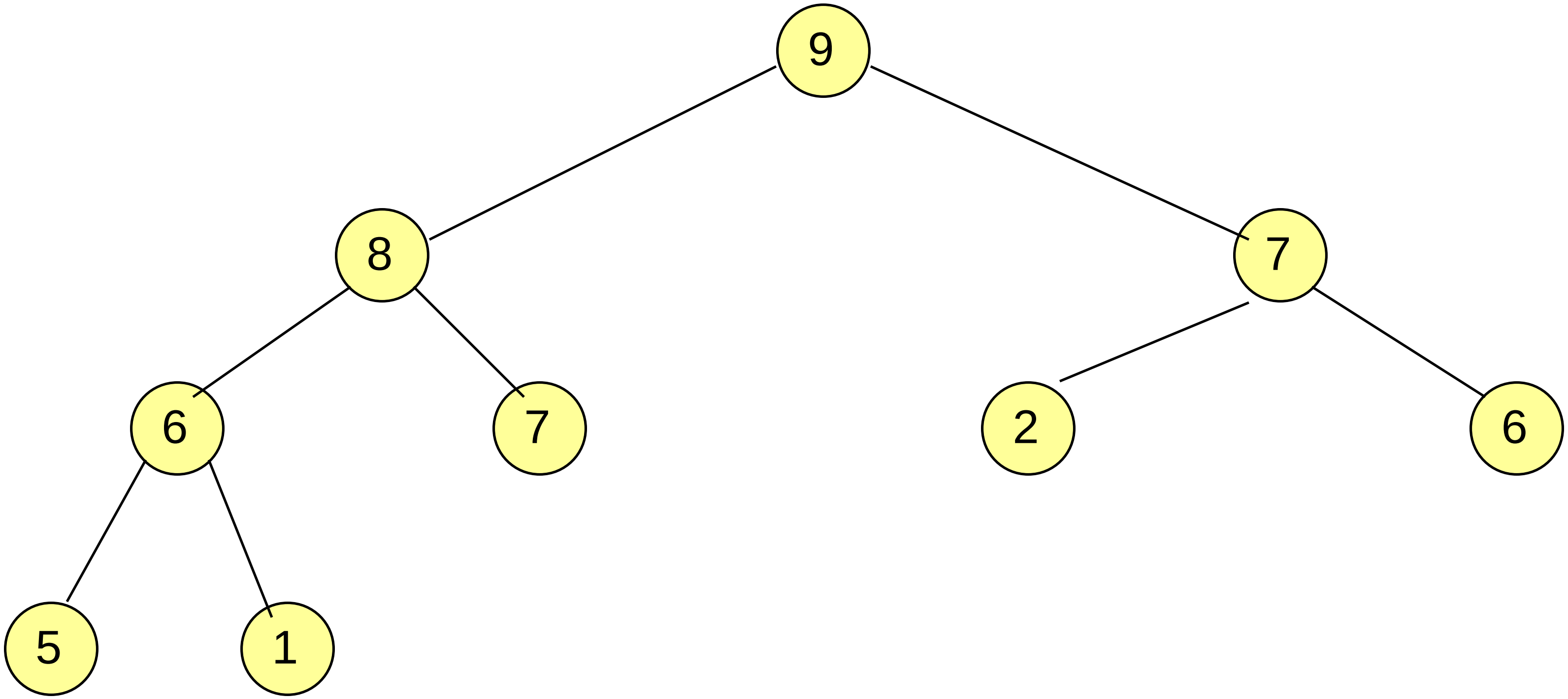
- complete binary tree
- min tree

Min Heap With 9 Nodes



Complete binary tree with 9 nodes that is also a min tree.

Max Heap With 9 Nodes



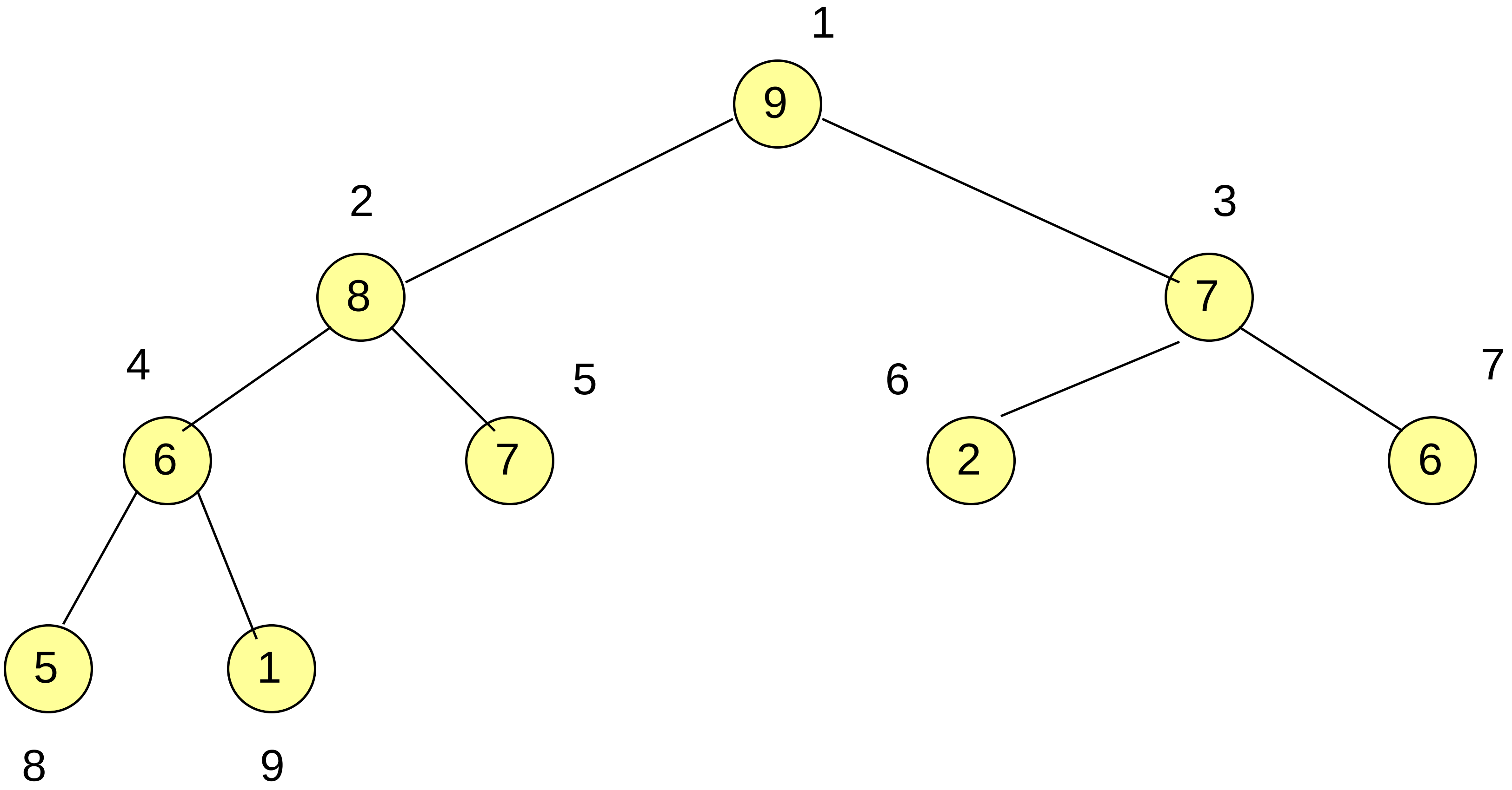
Complete binary tree with 9 nodes that is also a max tree.

Heap Height

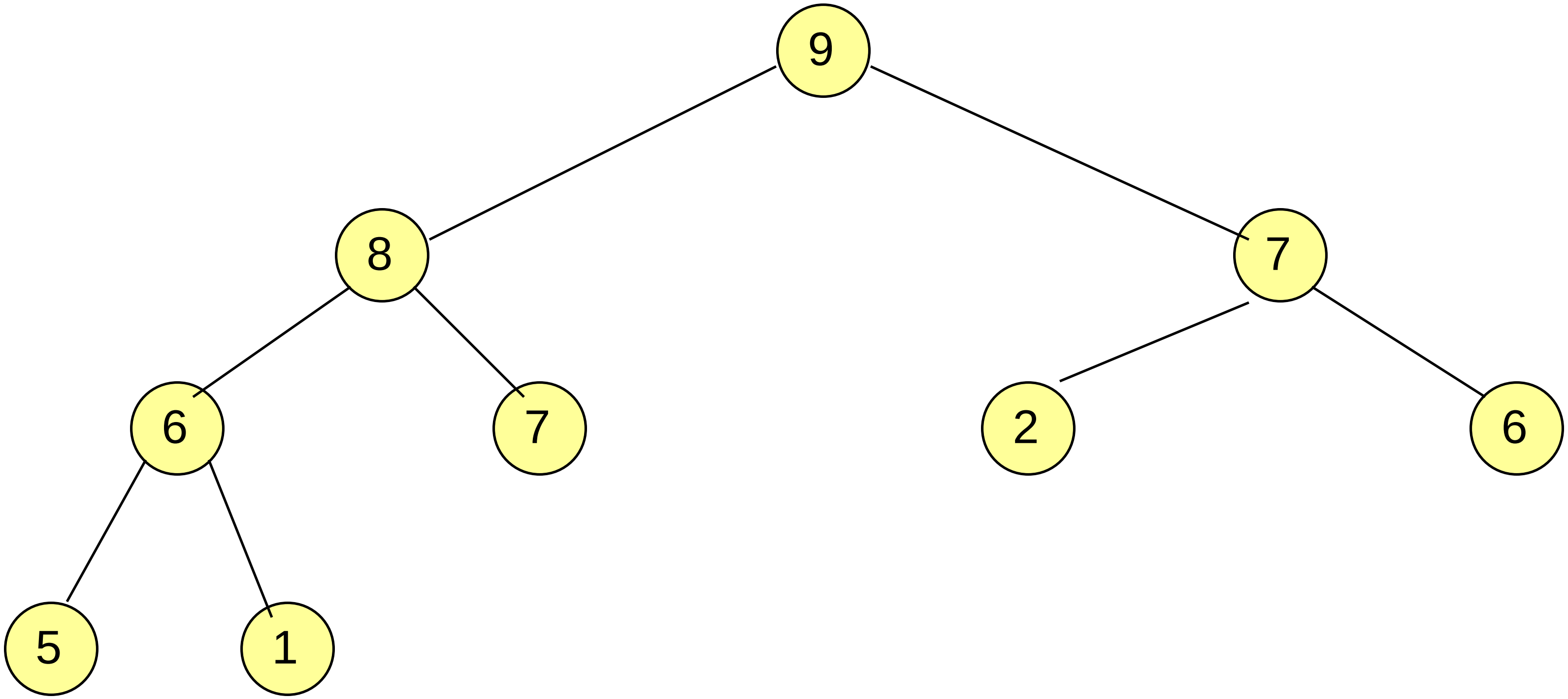
Since a heap is a complete binary tree, the height of an *n* node heap is

$$\lceil \log_2(n+1) \rceil$$

Moving Up And Down A Heap

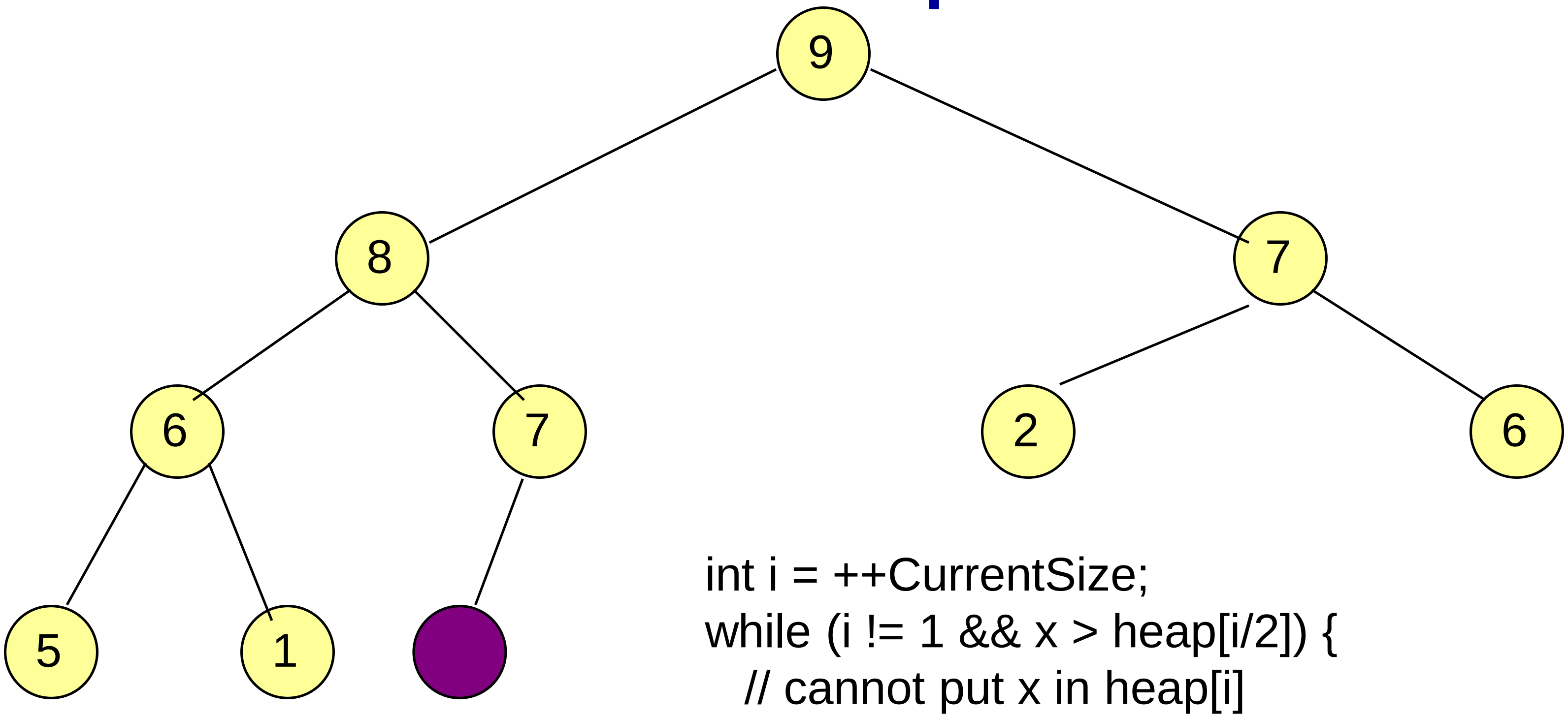


A Heap Is Efficiently Represented As An Array



	9	8	7	6	7	2	6	5	1							
0	1	2	3	4	5	6	7	8	9	10						

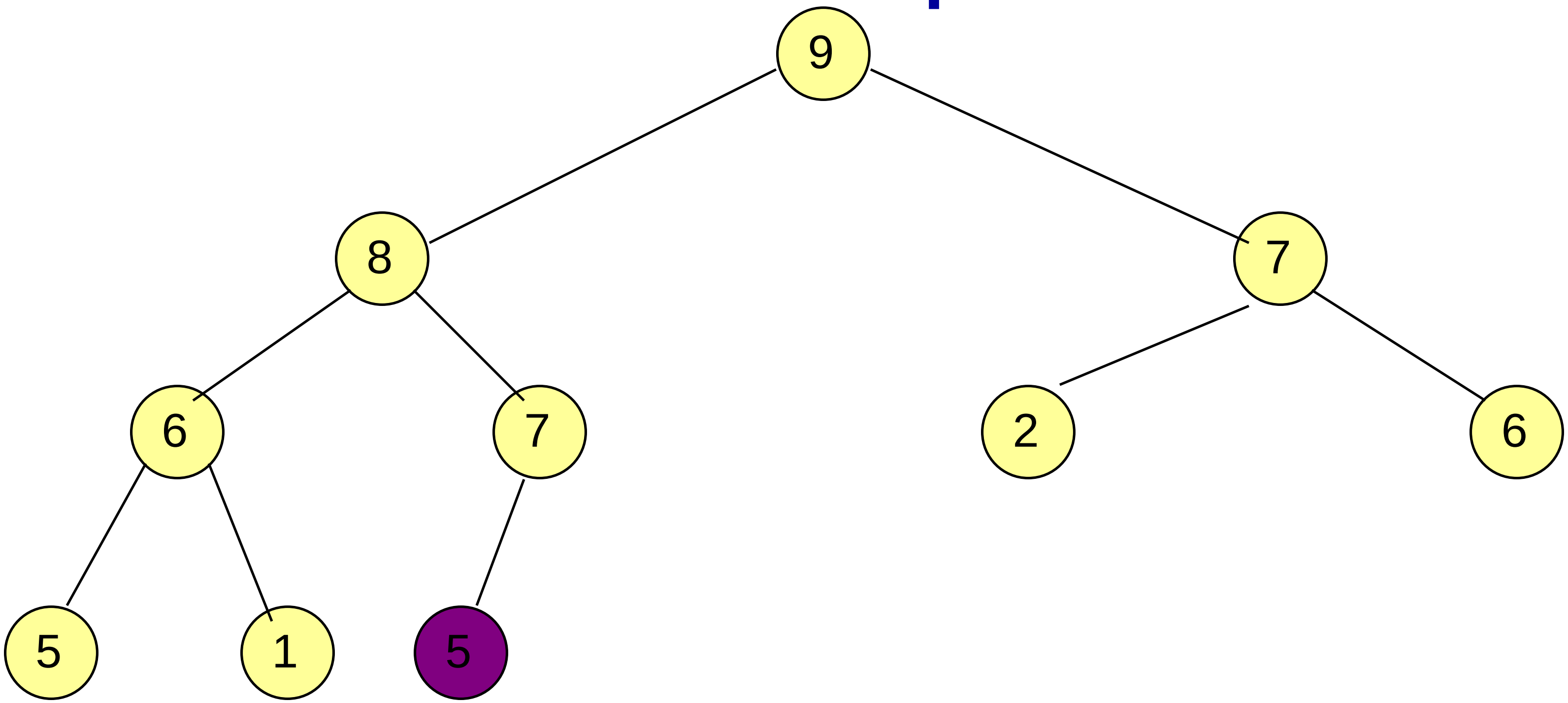
Putting An Element Into A Max Heap




```
int i = ++CurrentSize;
while (i != 1 && x > heap[i/2]) {
   // cannot put x in heap[i]
   heap[i] = heap[i/2]; // move element down
   i /= 2;              // move to parent
   }
 heap[i] = x;
```

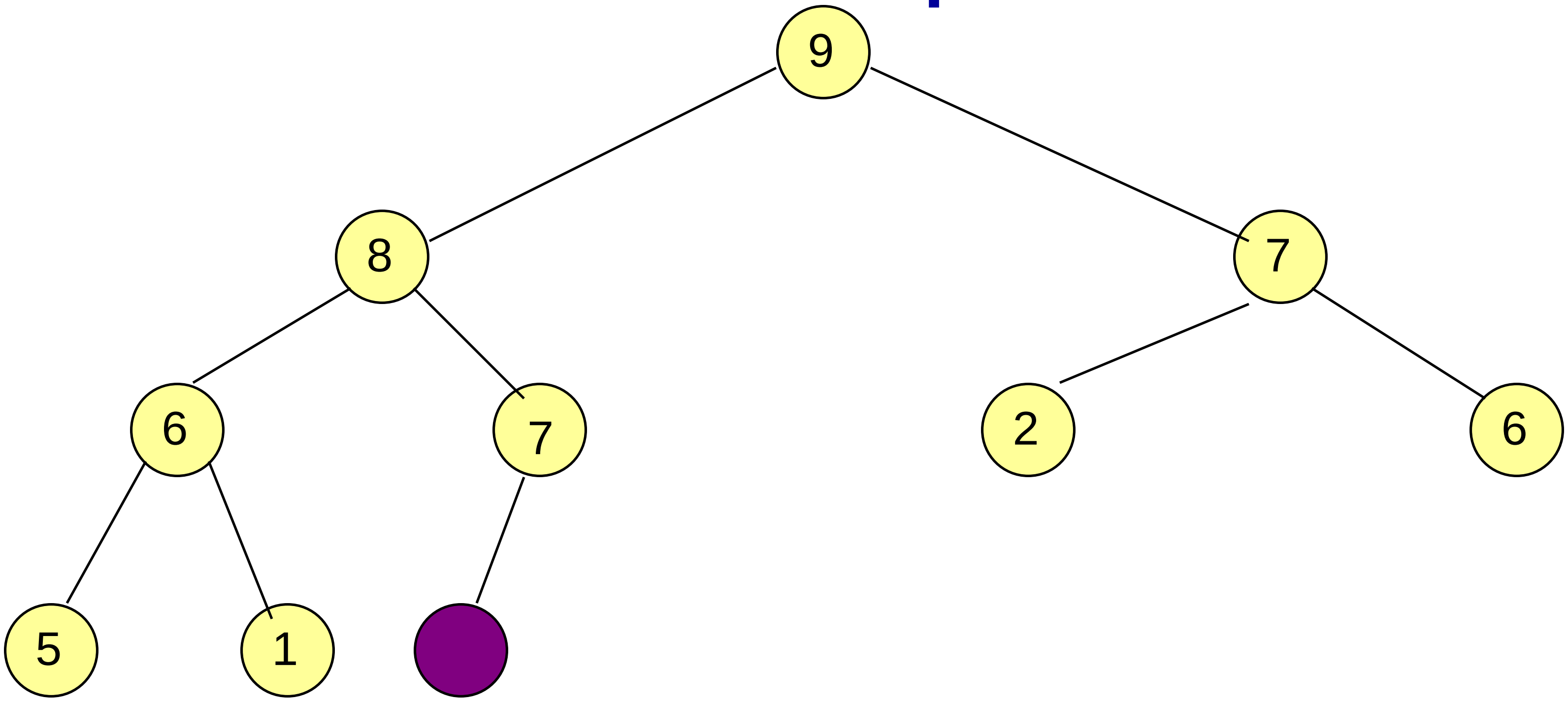
Complete binary tree with 10 nodes.

Putting An Element Into A Max Heap



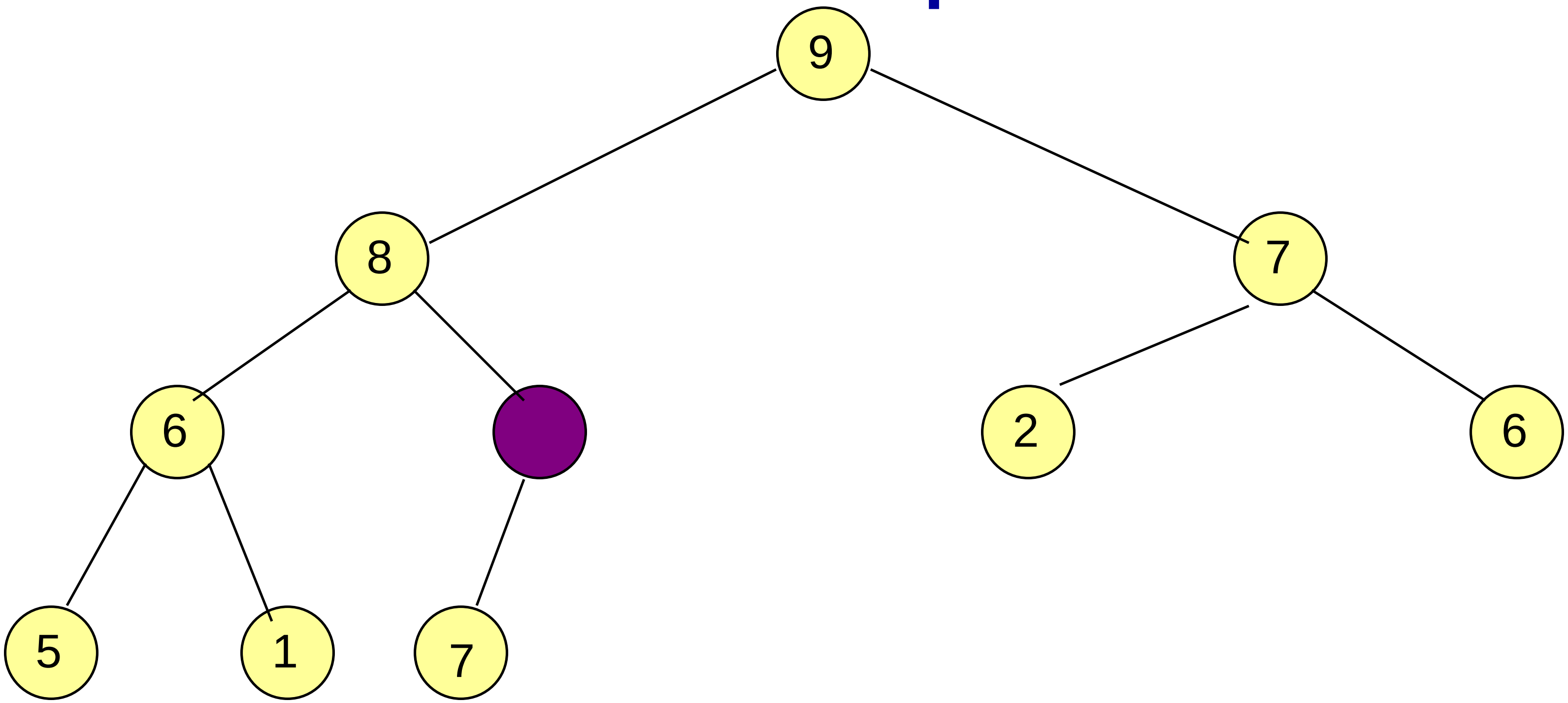
New element is 5.

Putting An Element Into A Max Heap



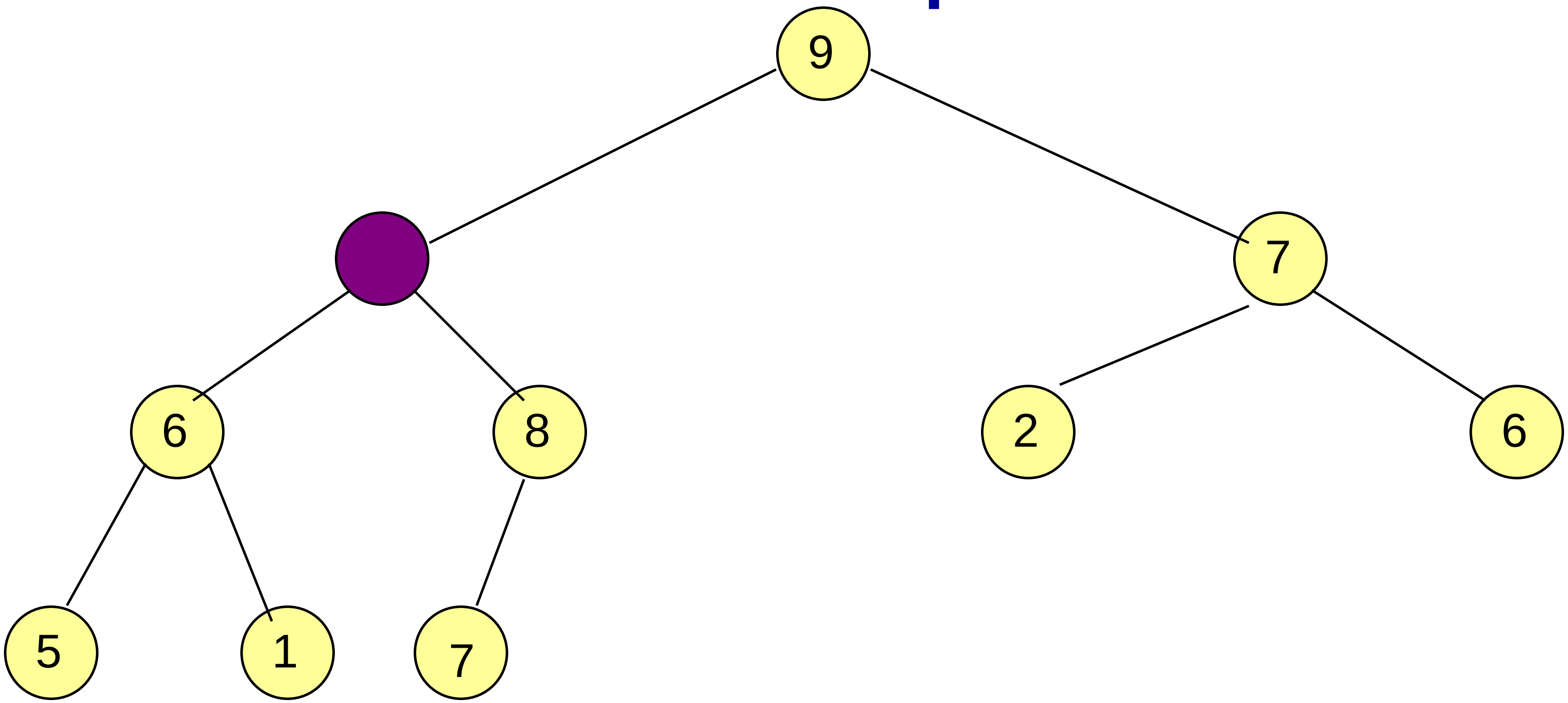
New element is 20.

Putting An Element Into A Max Heap



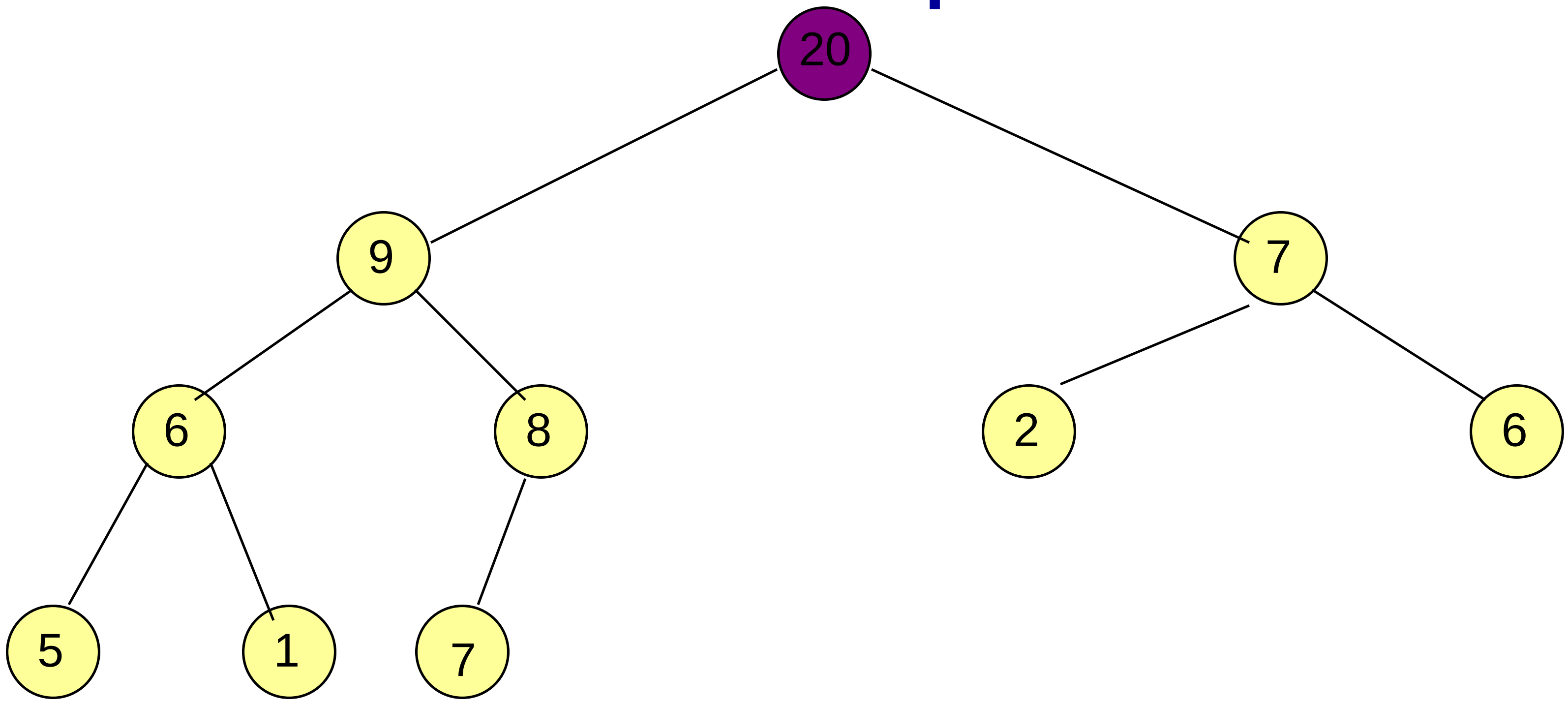
New element is 20.

Putting An Element Into A Max Heap



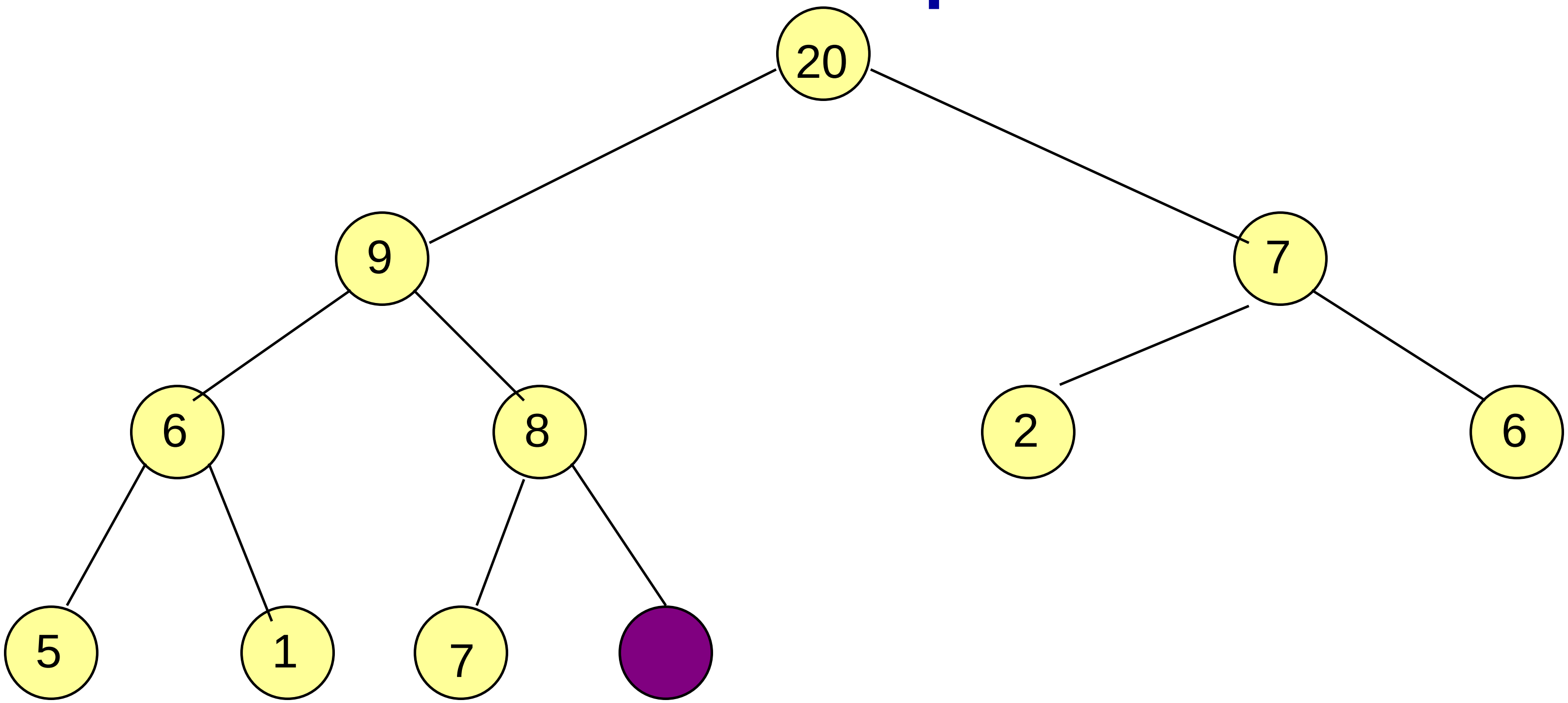
New element is 20.

Putting An Element Into A Max Heap



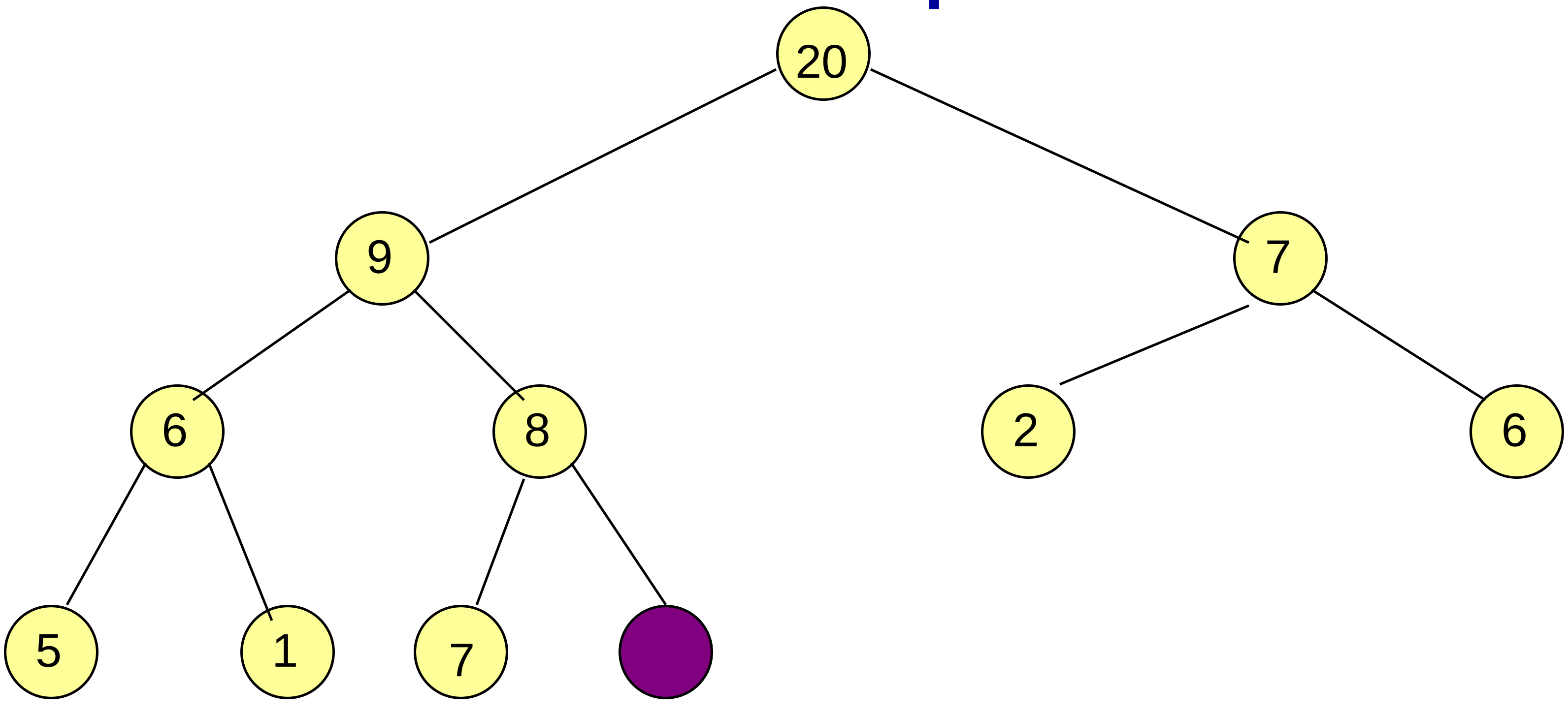
New element is 20.

Putting An Element Into A Max Heap



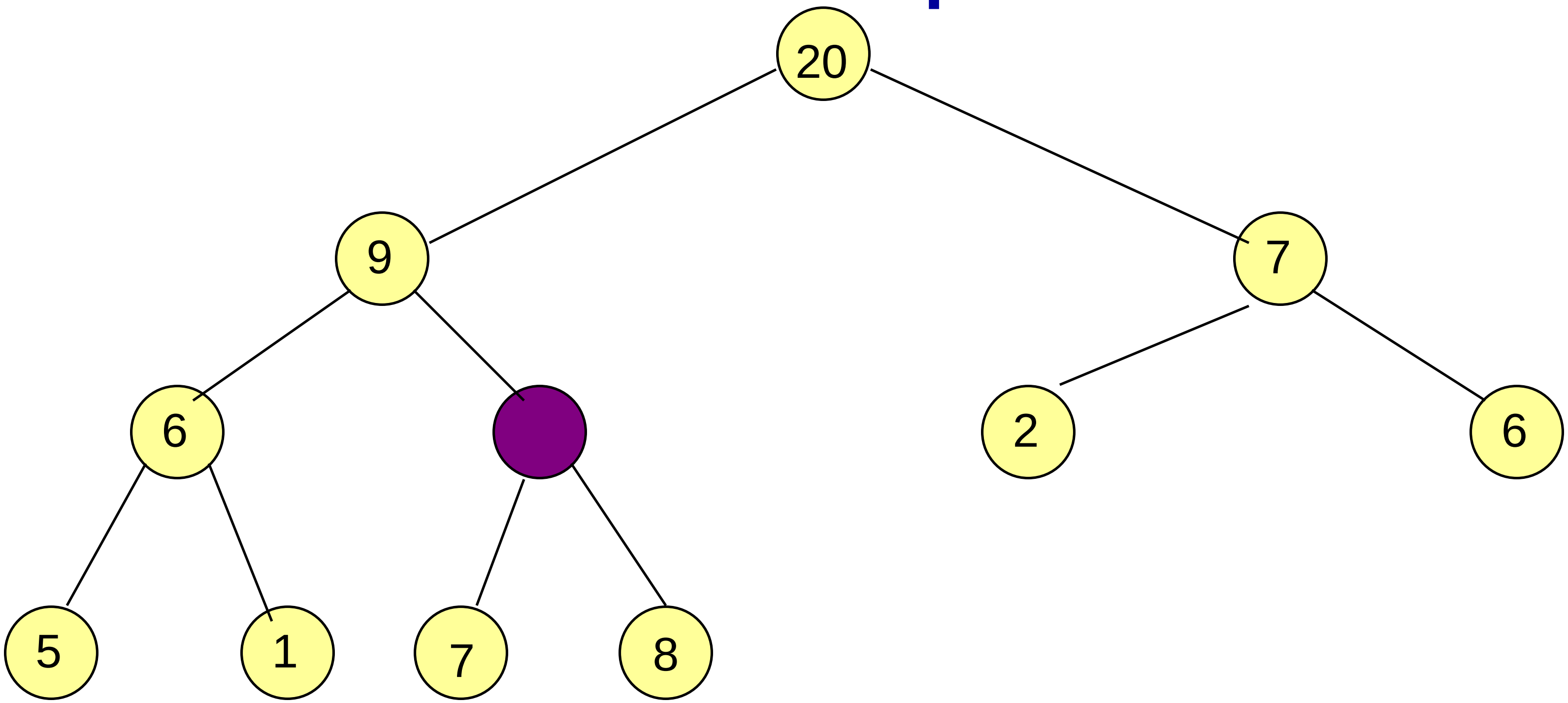
Complete binary tree with 11 nodes.

Putting An Element Into A Max Heap



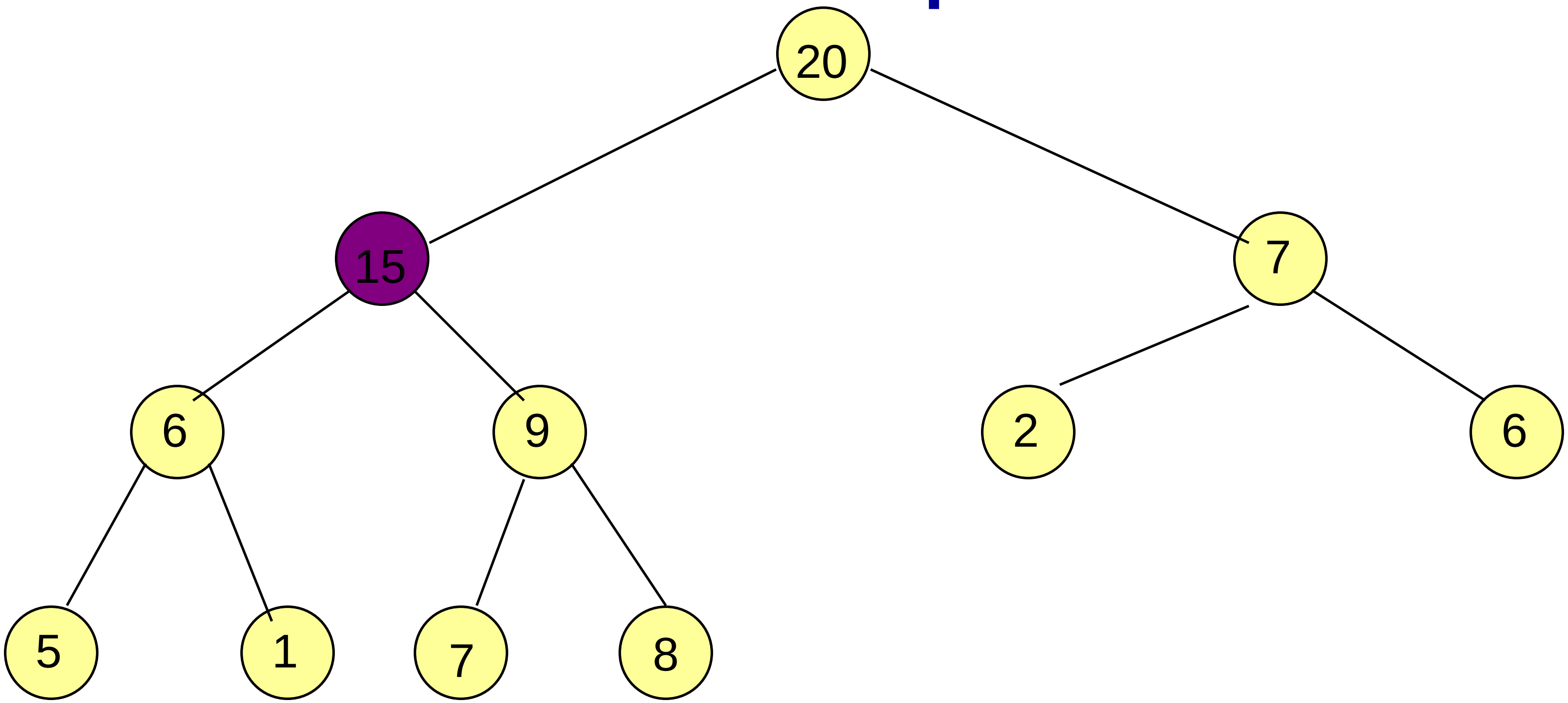
New element is 15.

Putting An Element Into A Max Heap



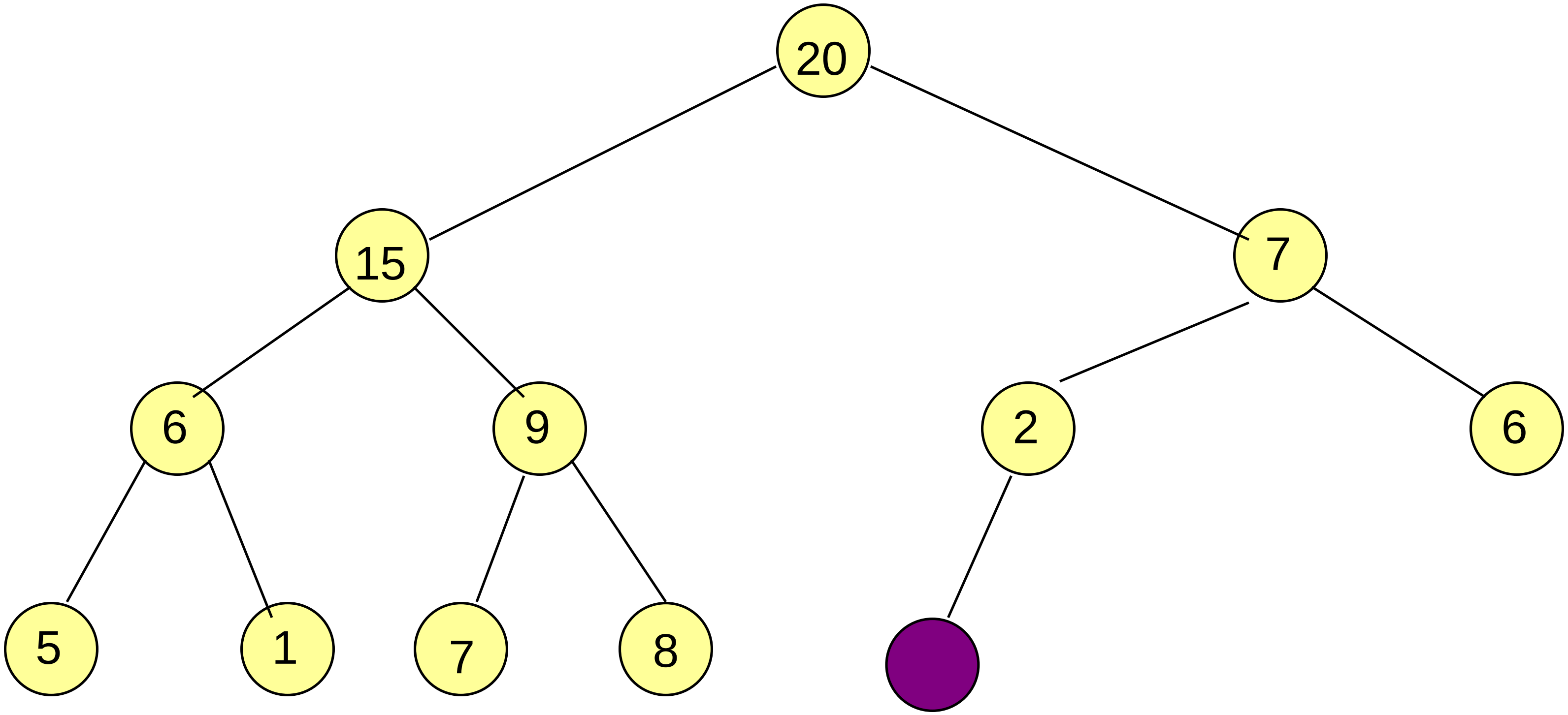
New element is 15.

Putting An Element Into A Max Heap



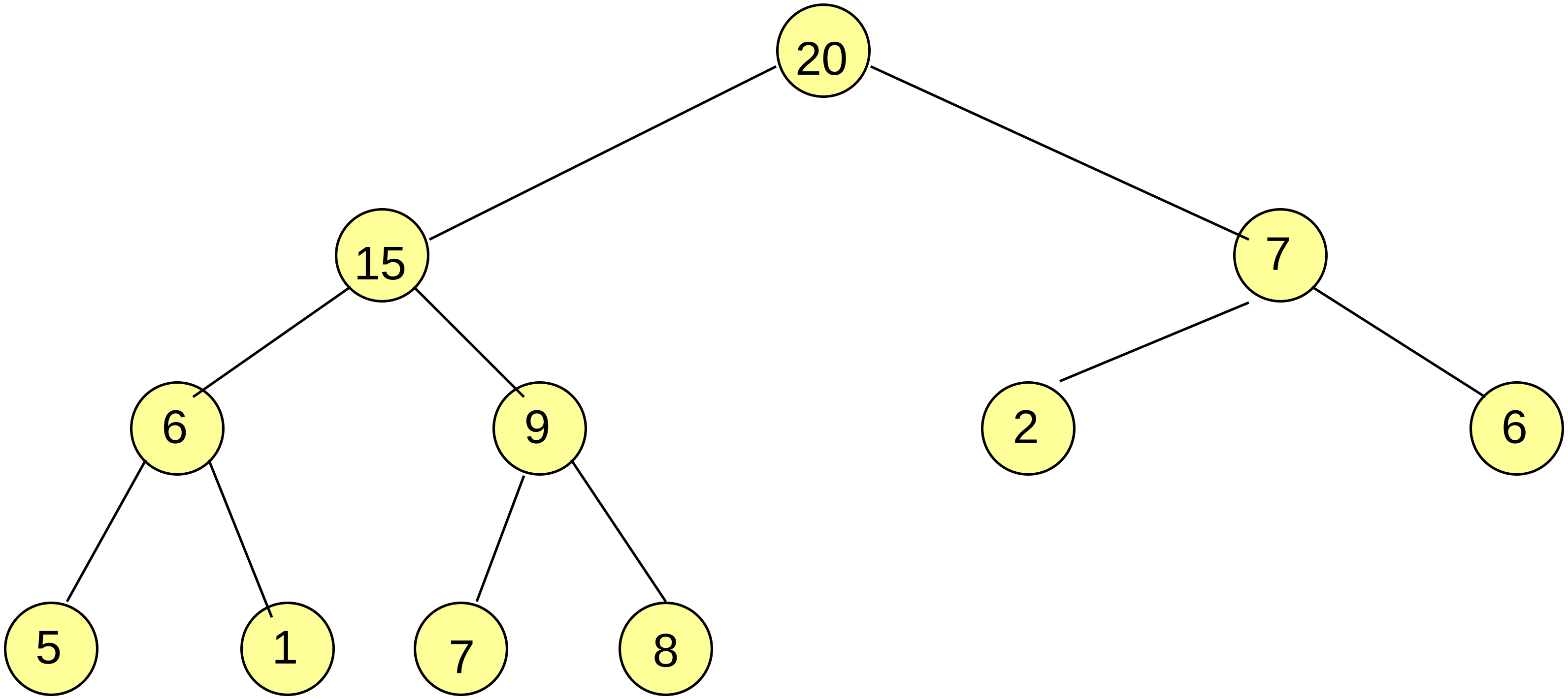
New element is 15.

Complexity Of Put



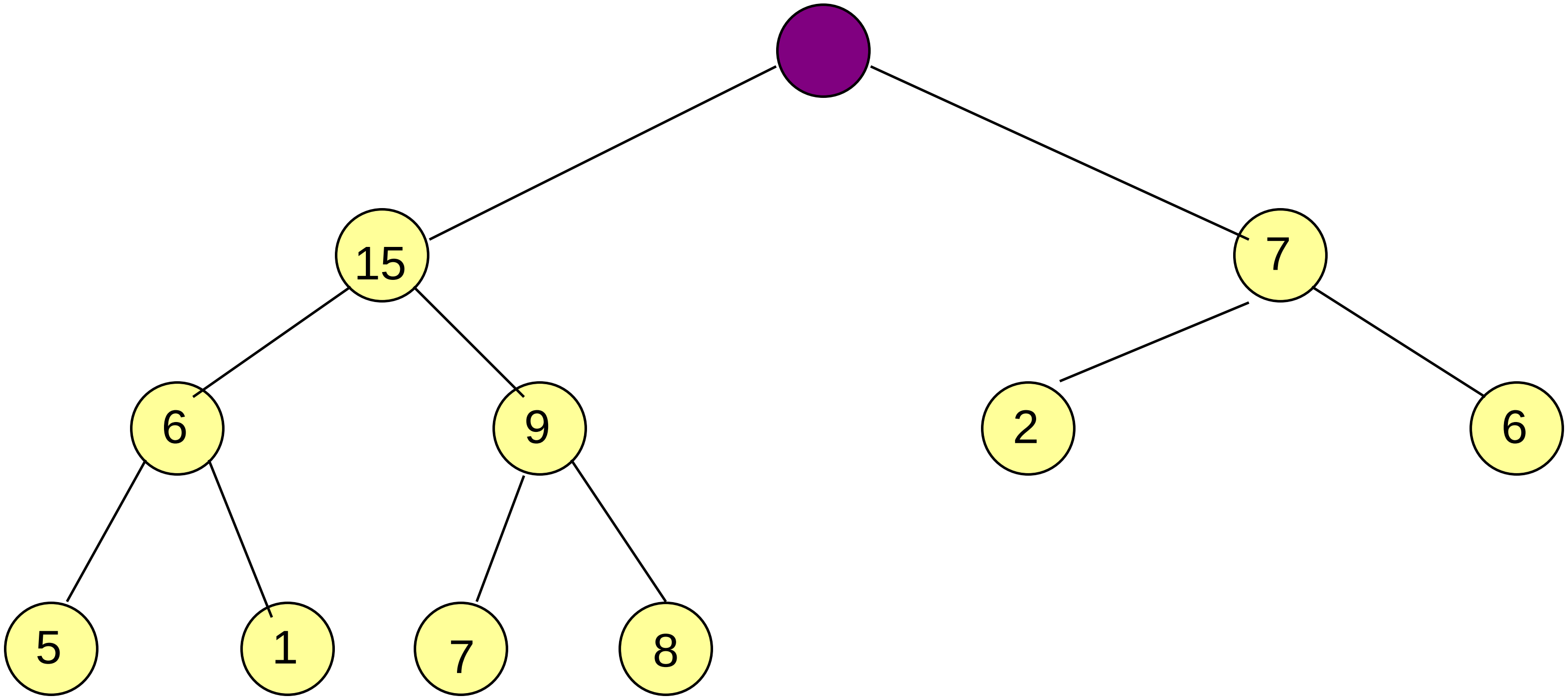
Complexity is O(log n), where n is heap size.

Removing The Max Element



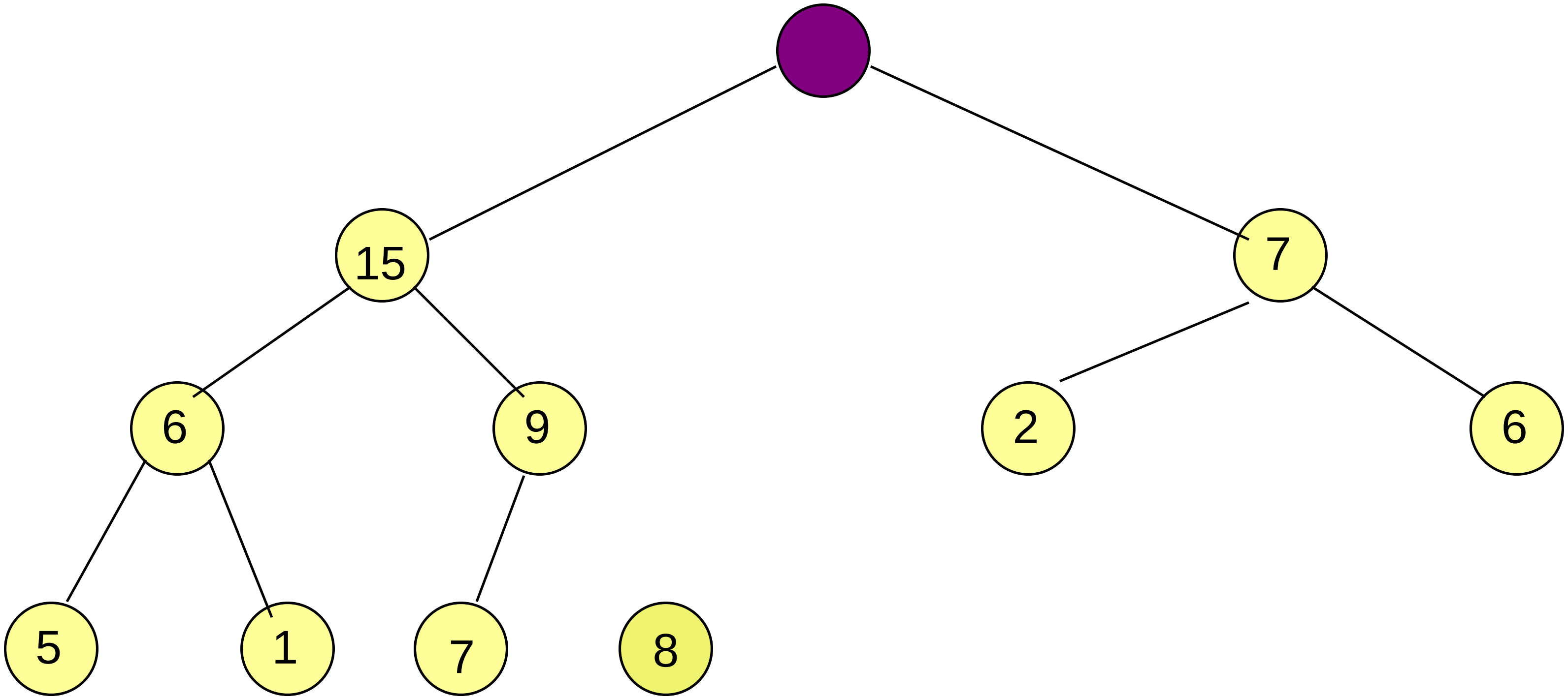
Max element is in the root.

Removing The Max Element



After max element is removed.

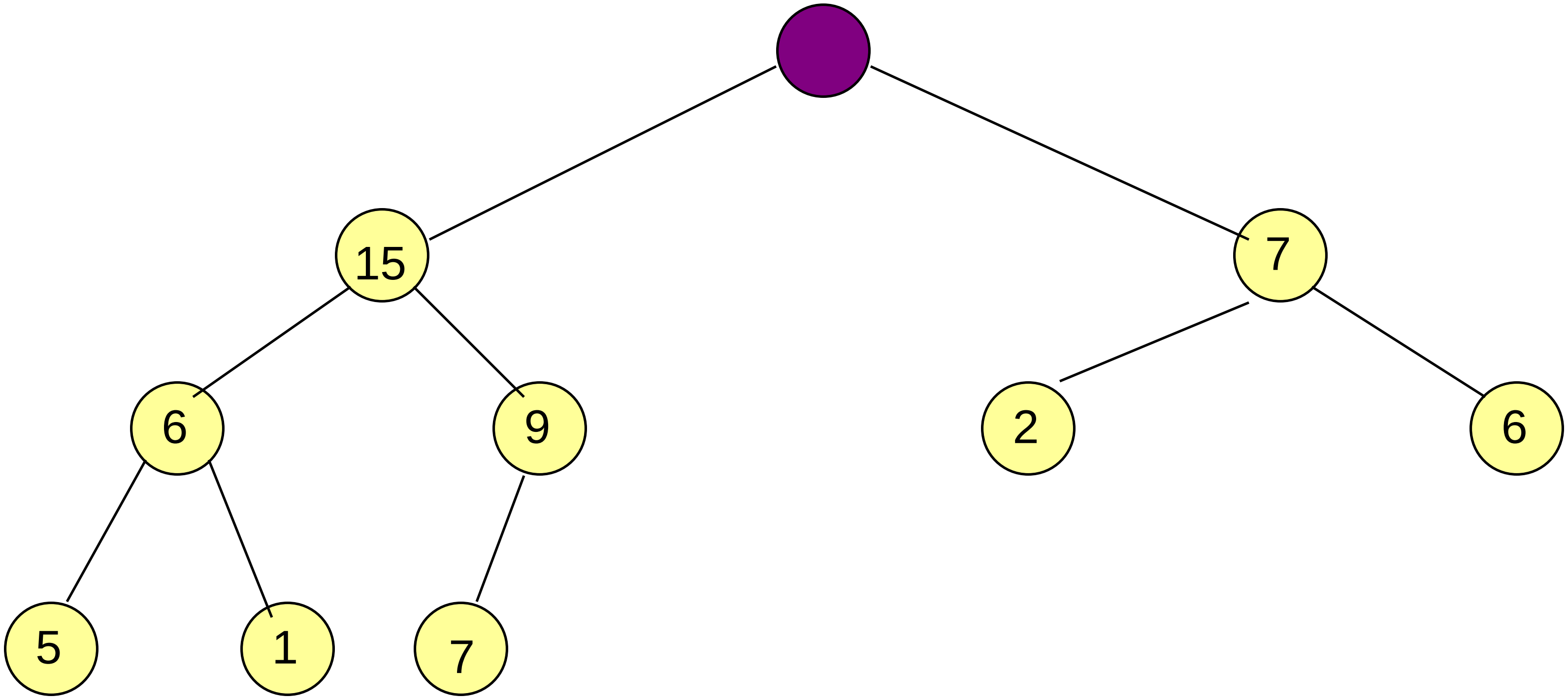
Removing The Max Element



Heap with 10 nodes.

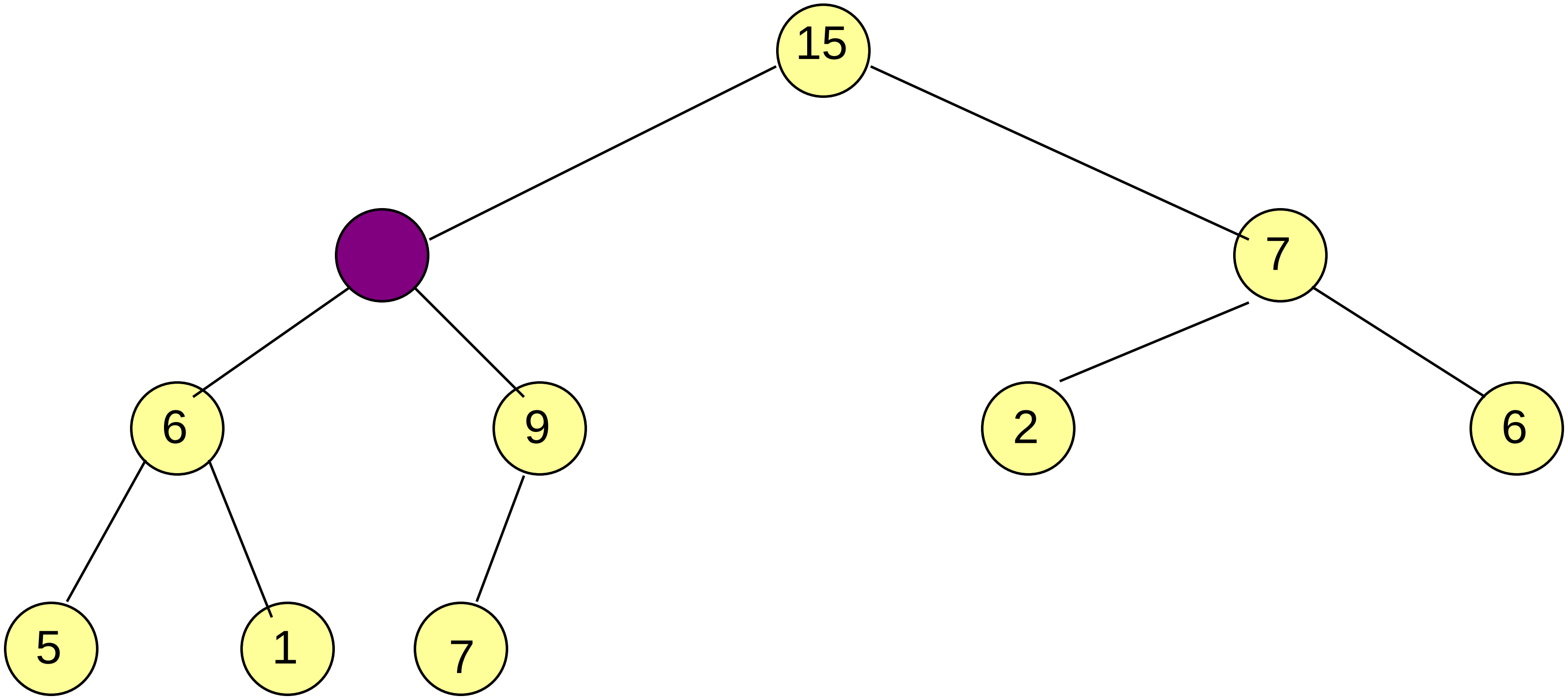
Reinsert 8 into the heap.

Removing The Max Element



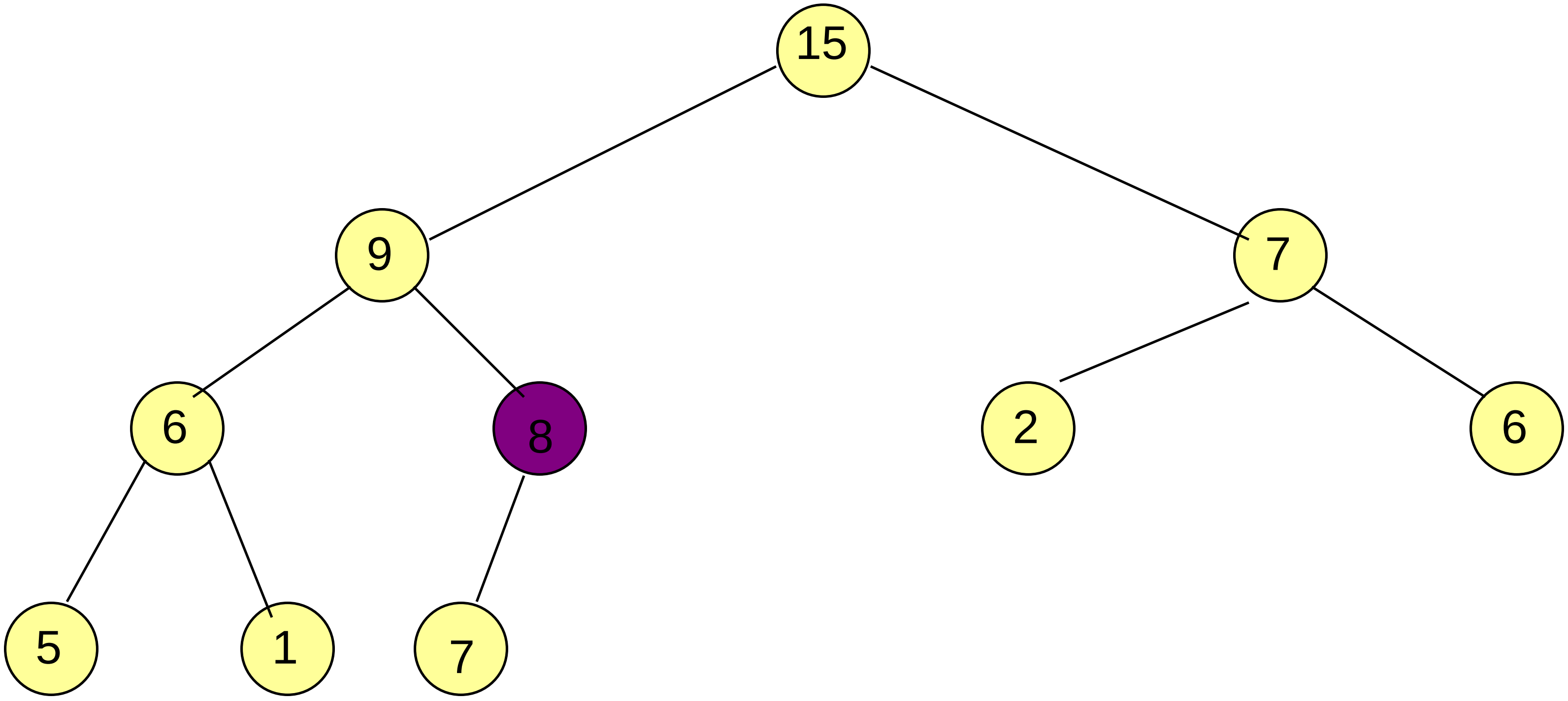
Reinsert 8 into the heap.

Removing The Max Element



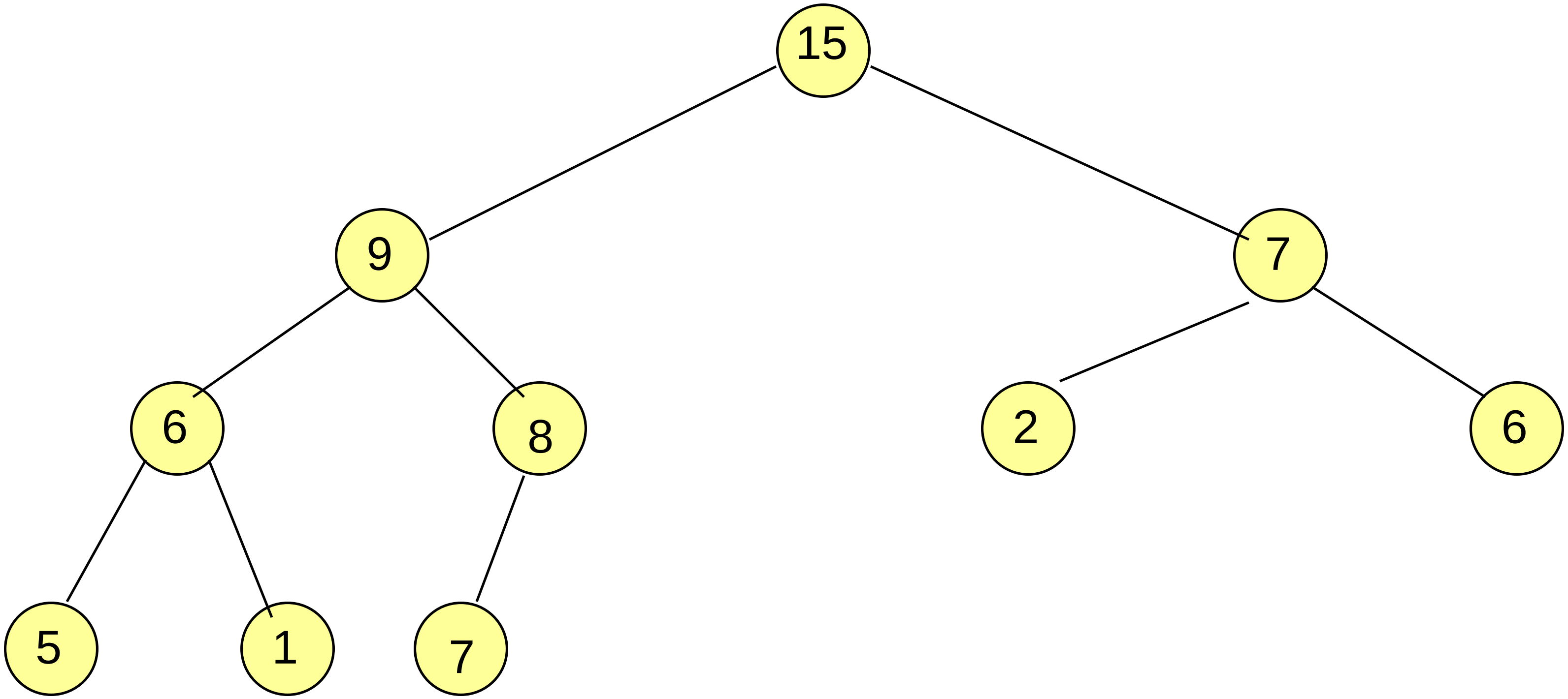
Reinsert 8 into the heap.

Removing The Max Element



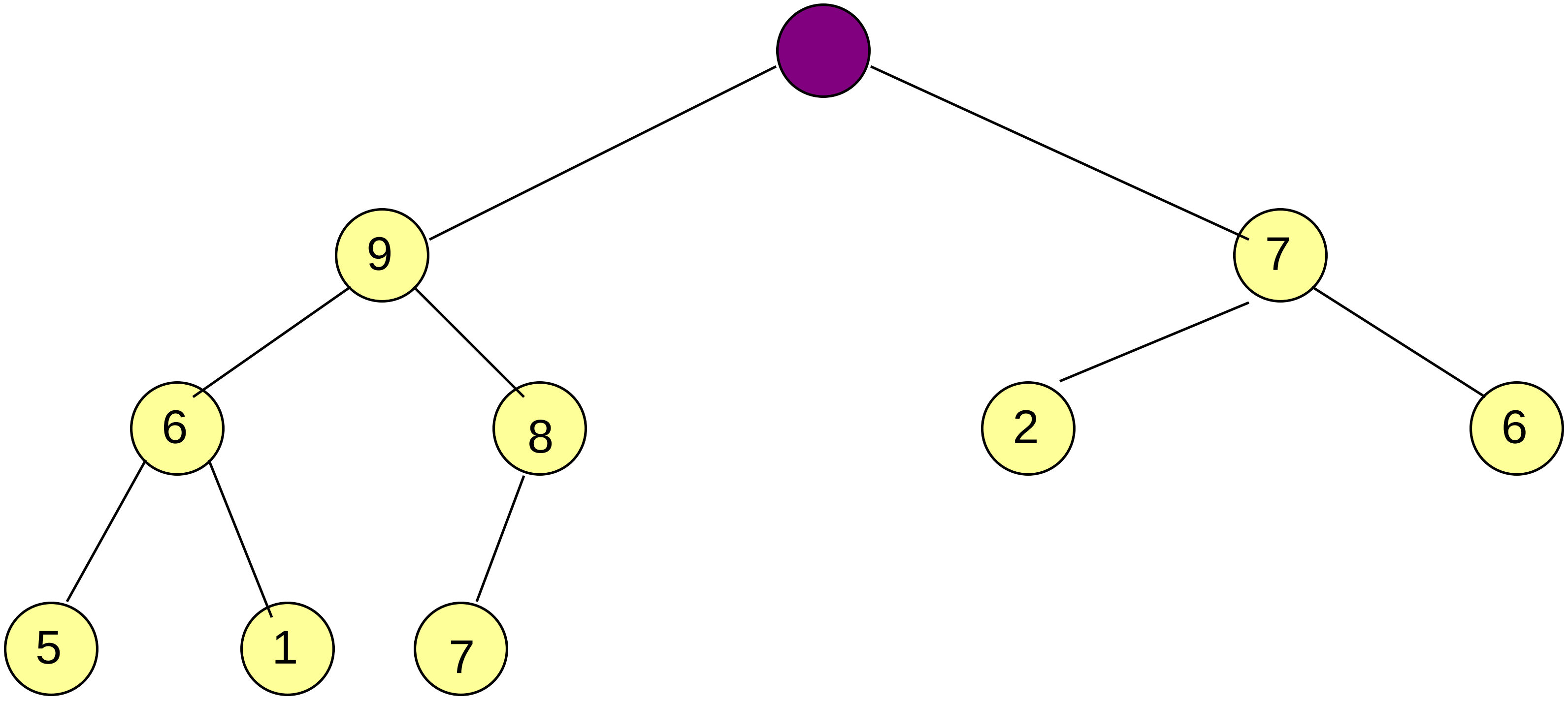
Reinsert 8 into the heap.

Removing The Max Element



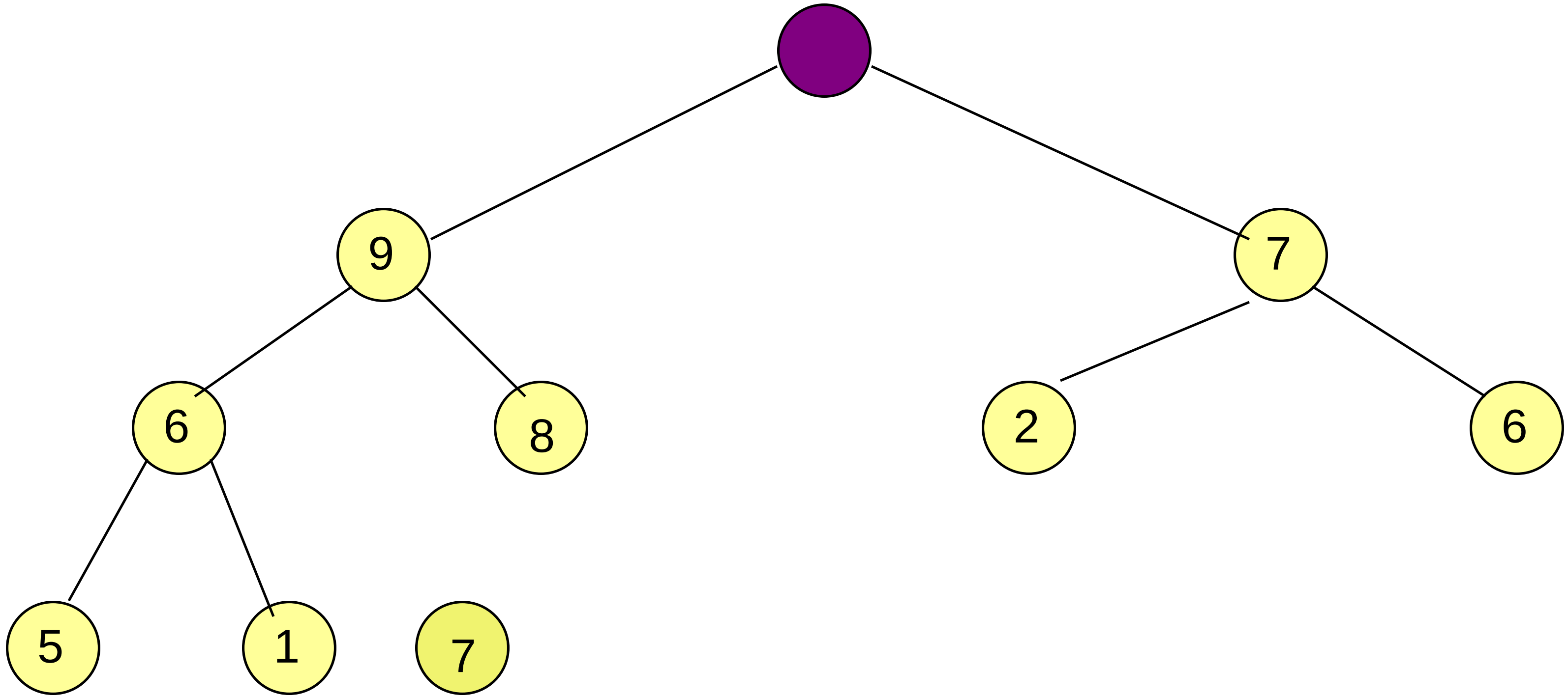
Max element is 15.

Removing The Max Element



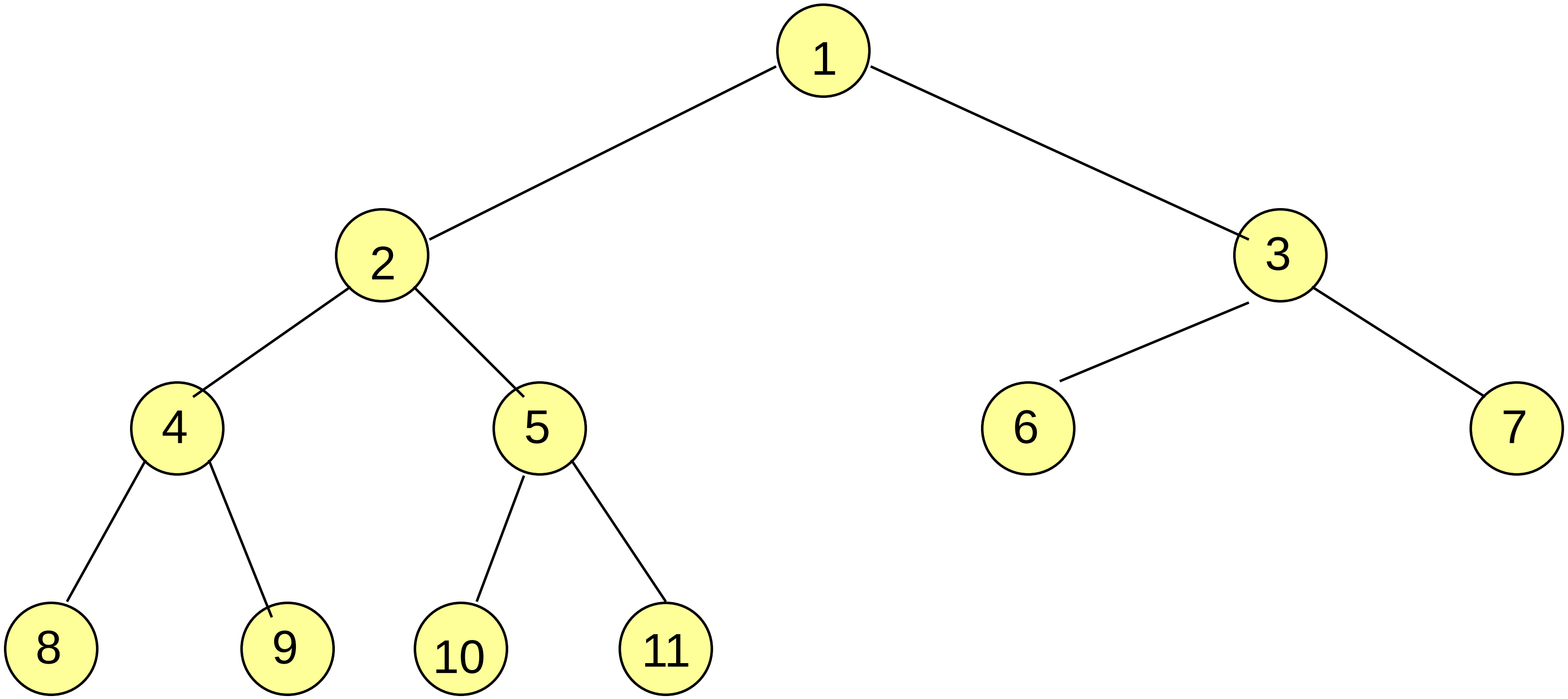
After max element is removed.

Removing The Max Element



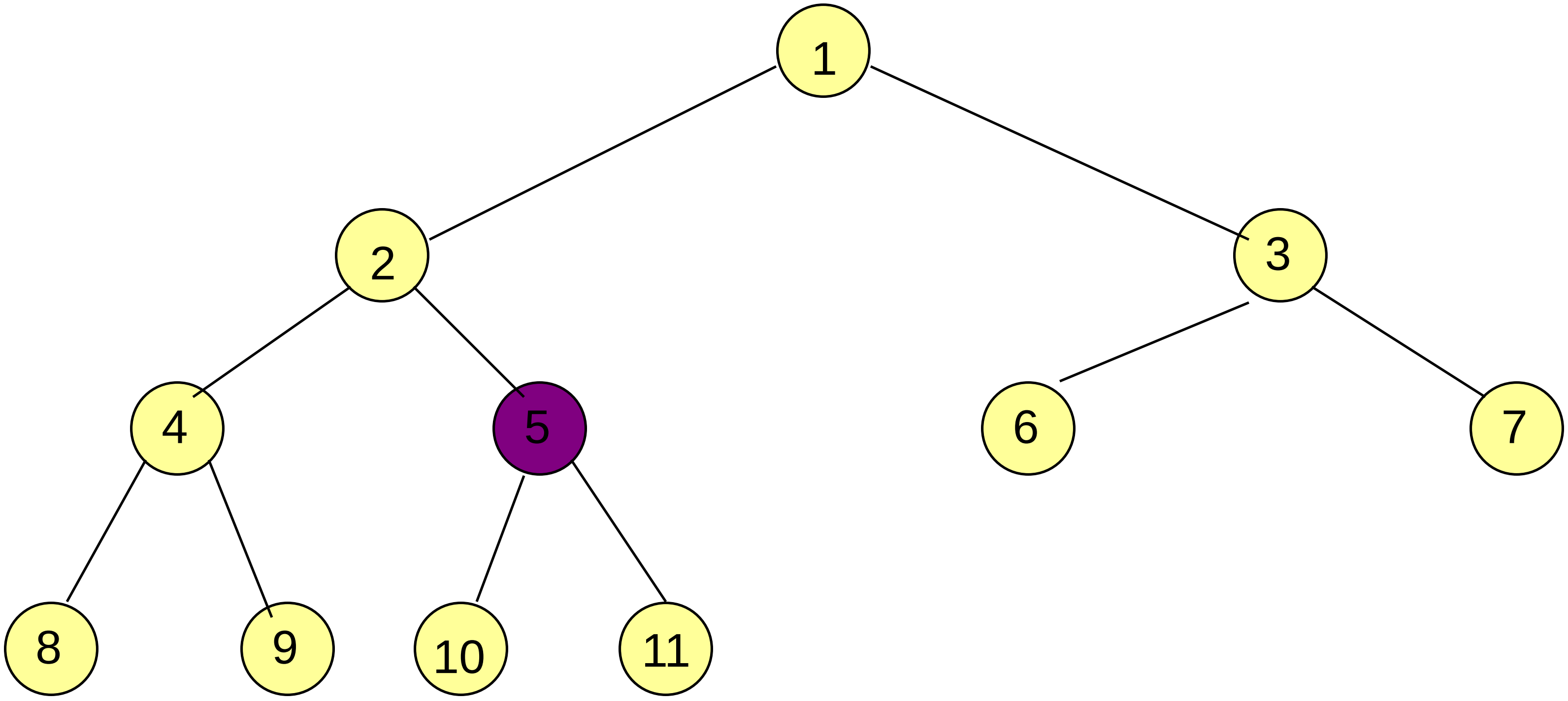
Heap with 9 nodes.

Initializing A Max Heap



input array = [-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

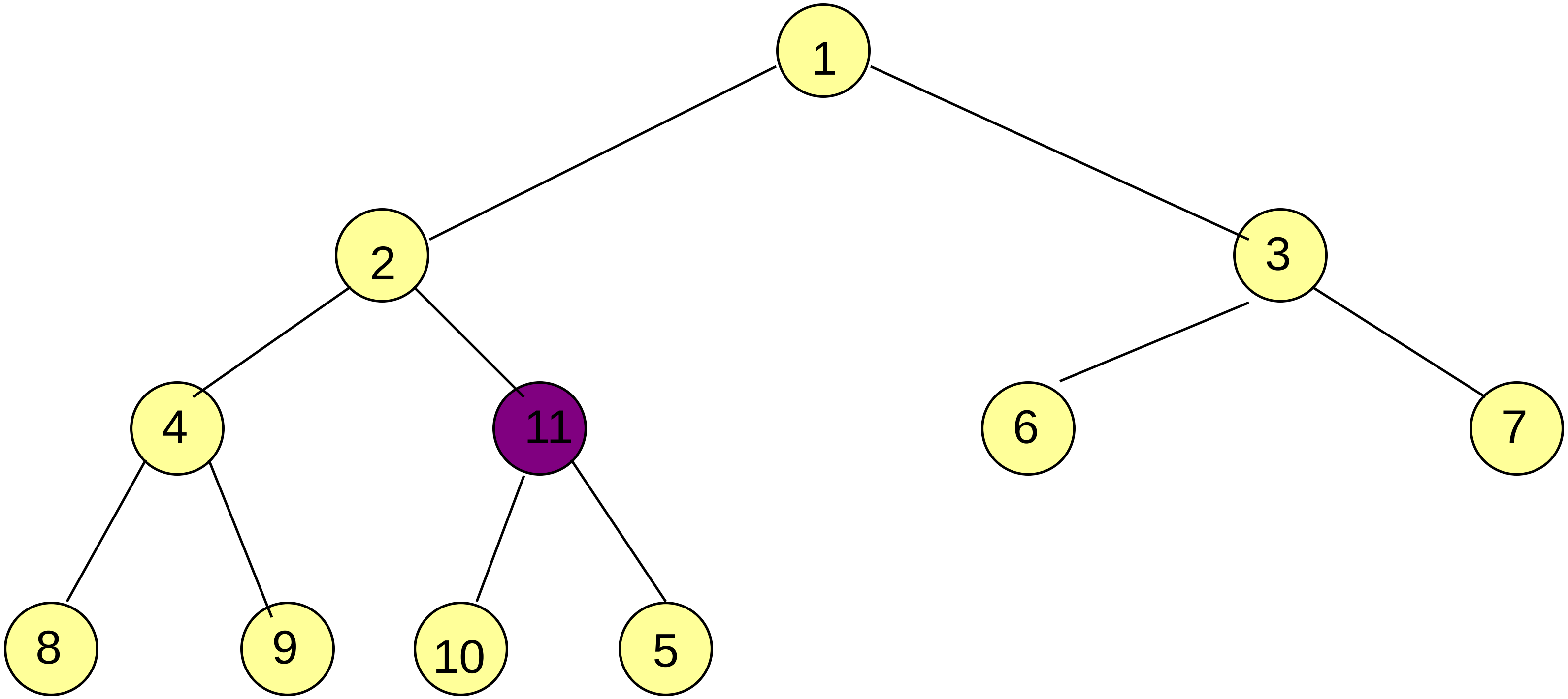
Initializing A Max Heap



Start at rightmost array position that has a child.

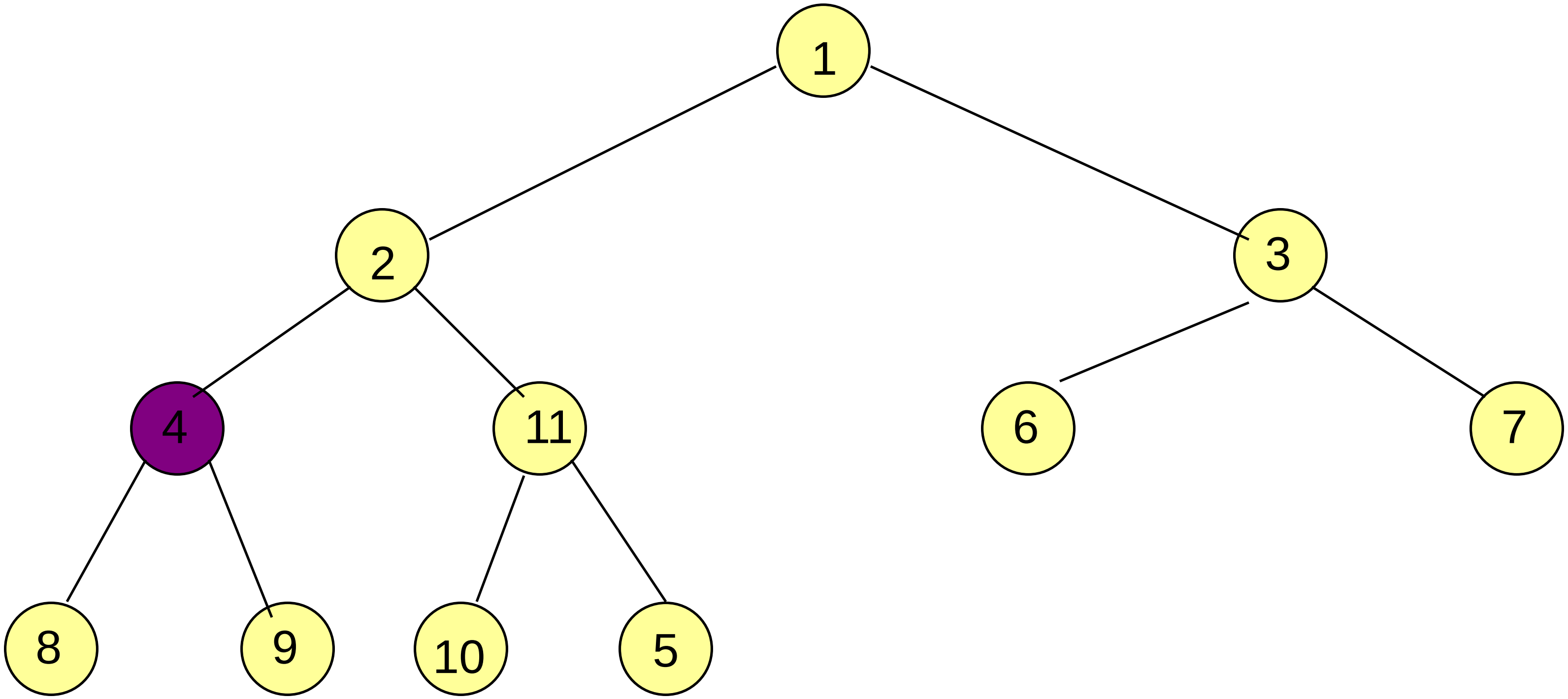
Index is n/2.

Initializing A Max Heap

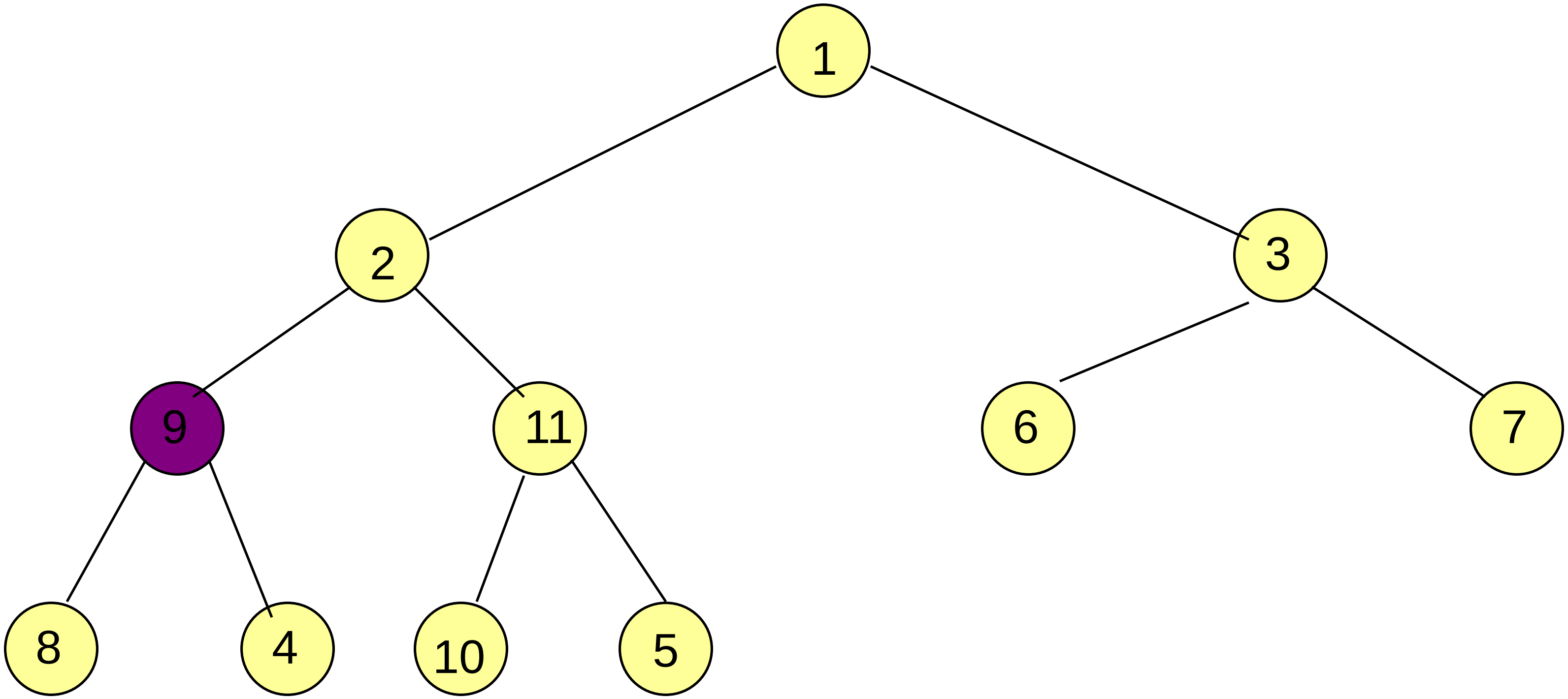


Move to next lower array position.

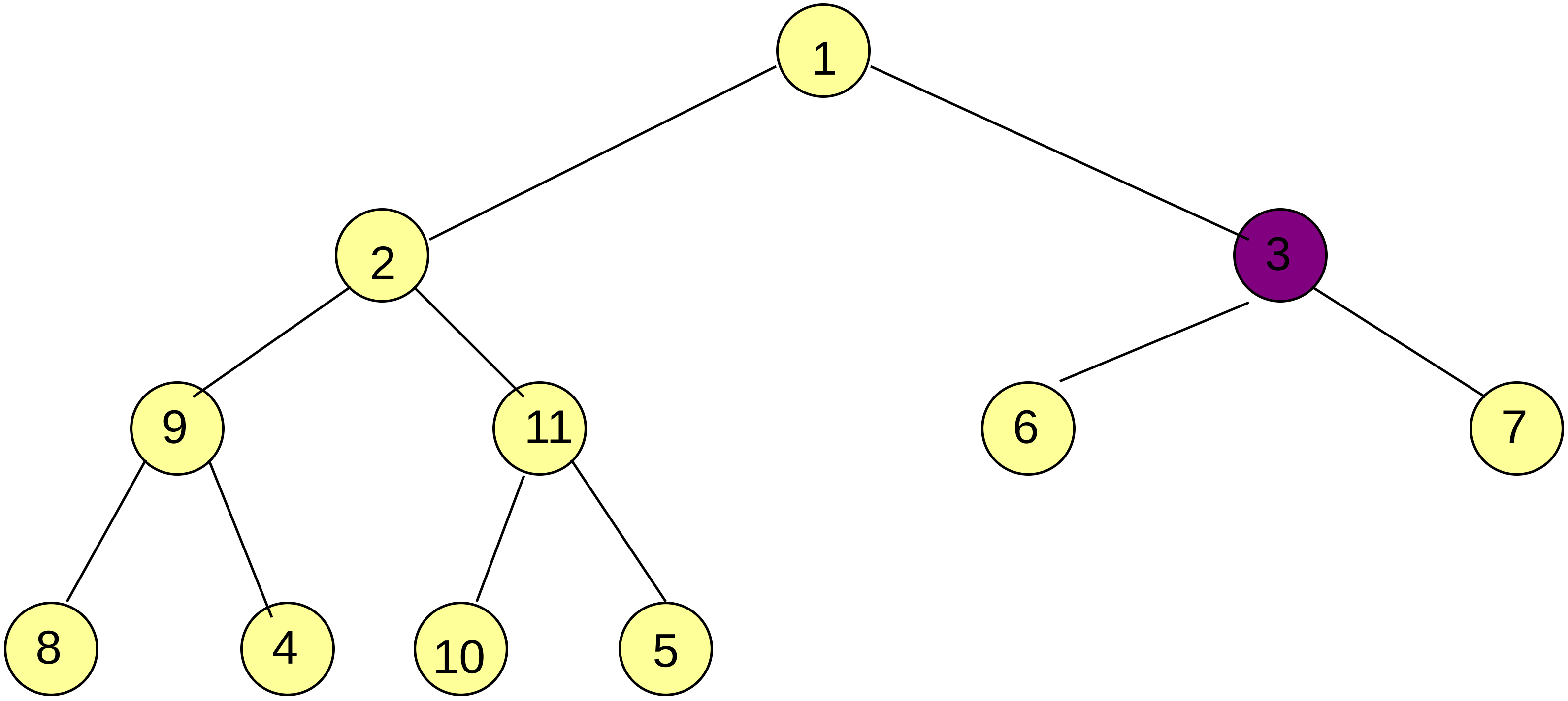
Initializing A Max Heap



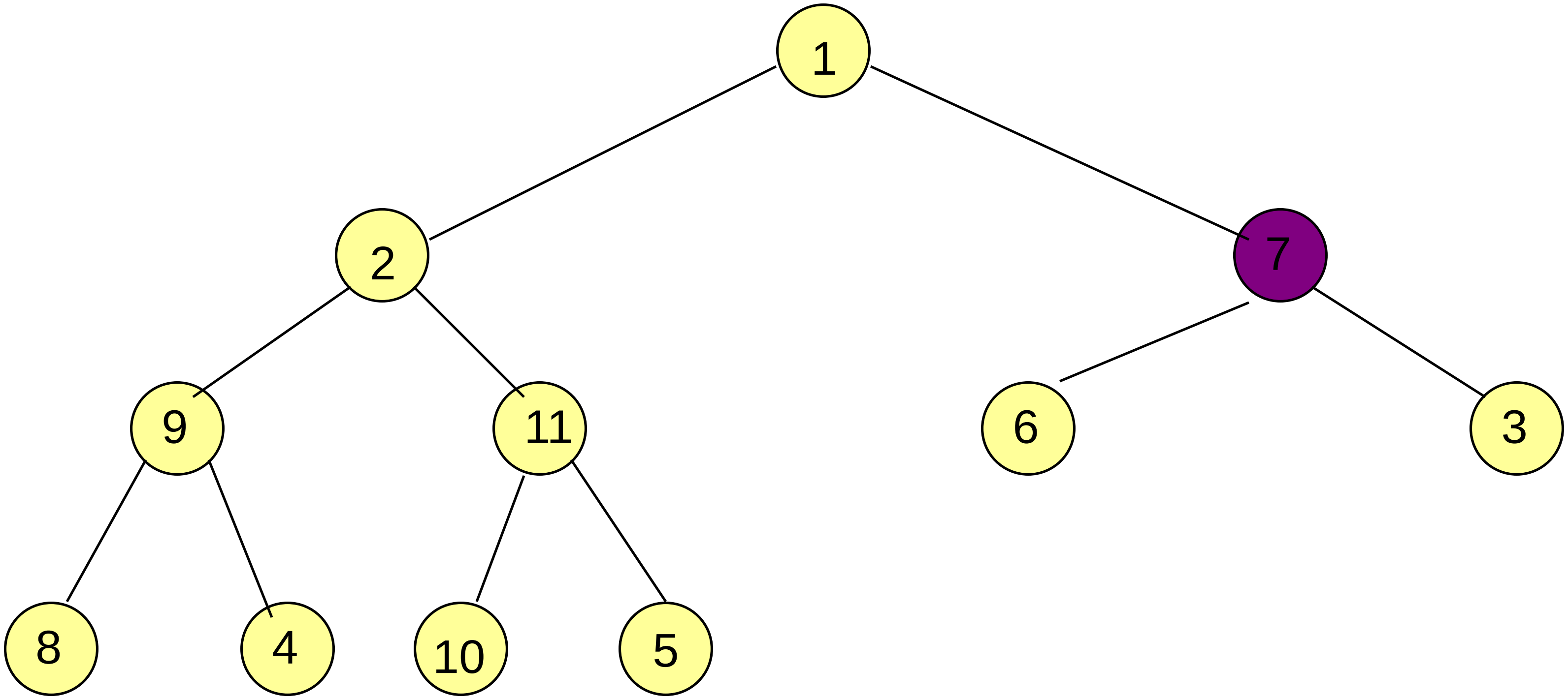
Initializing A Max Heap



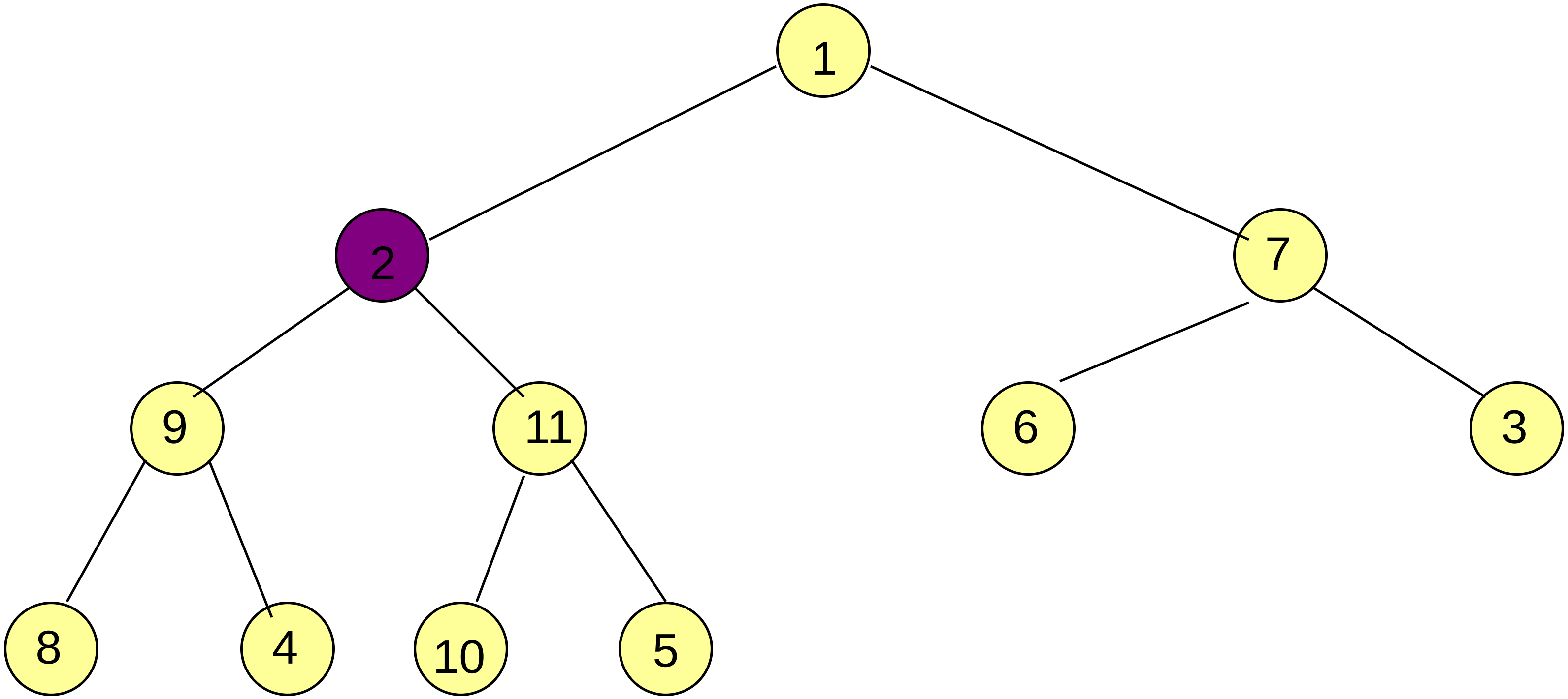
Initializing A Max Heap



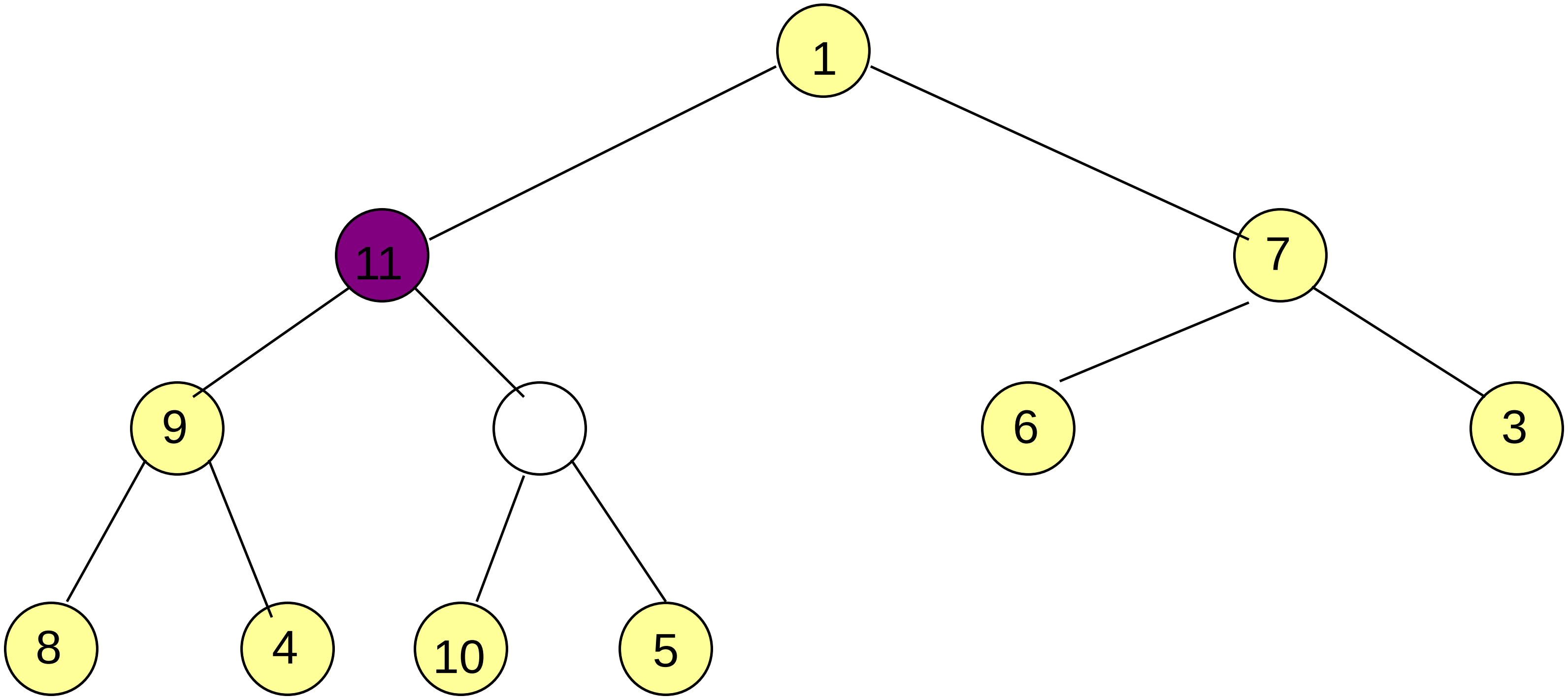
Initializing A Max Heap



Initializing A Max Heap

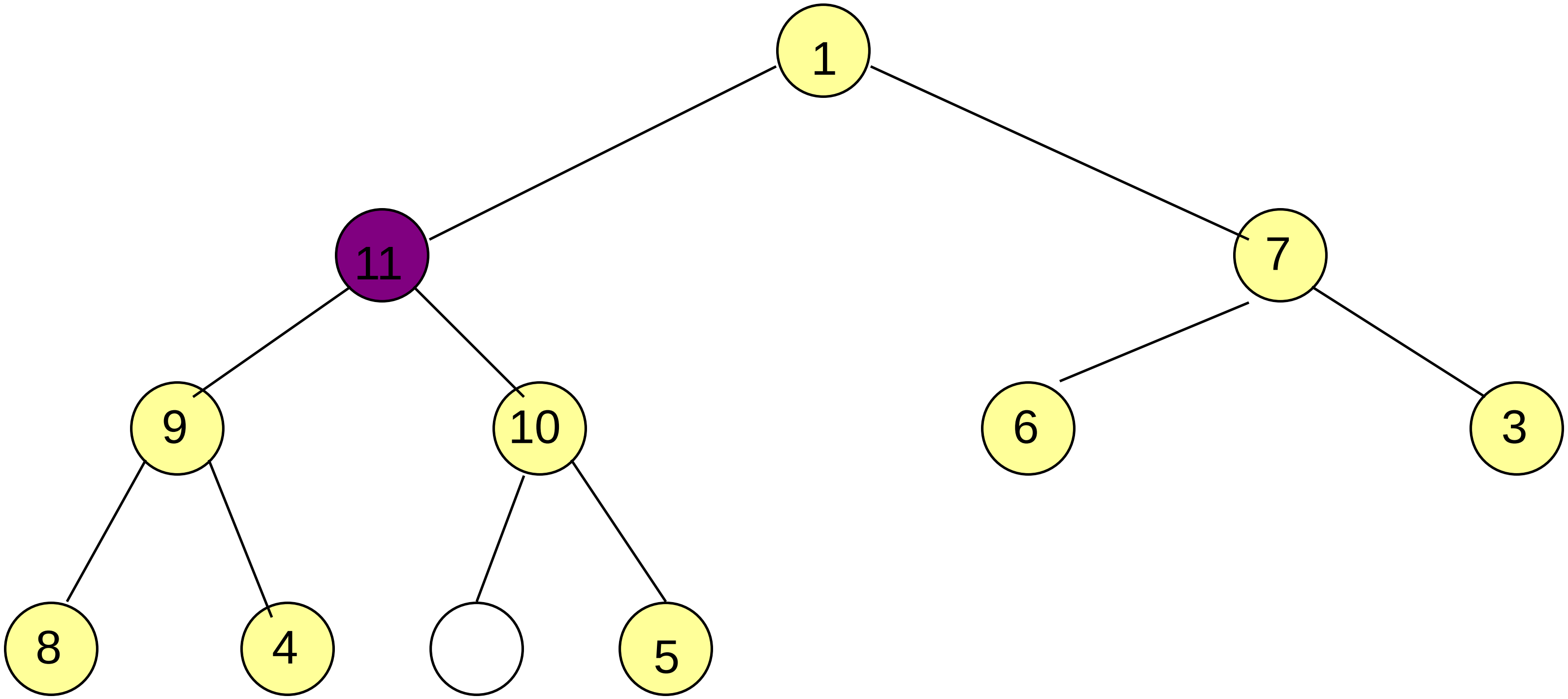


Initializing A Max Heap



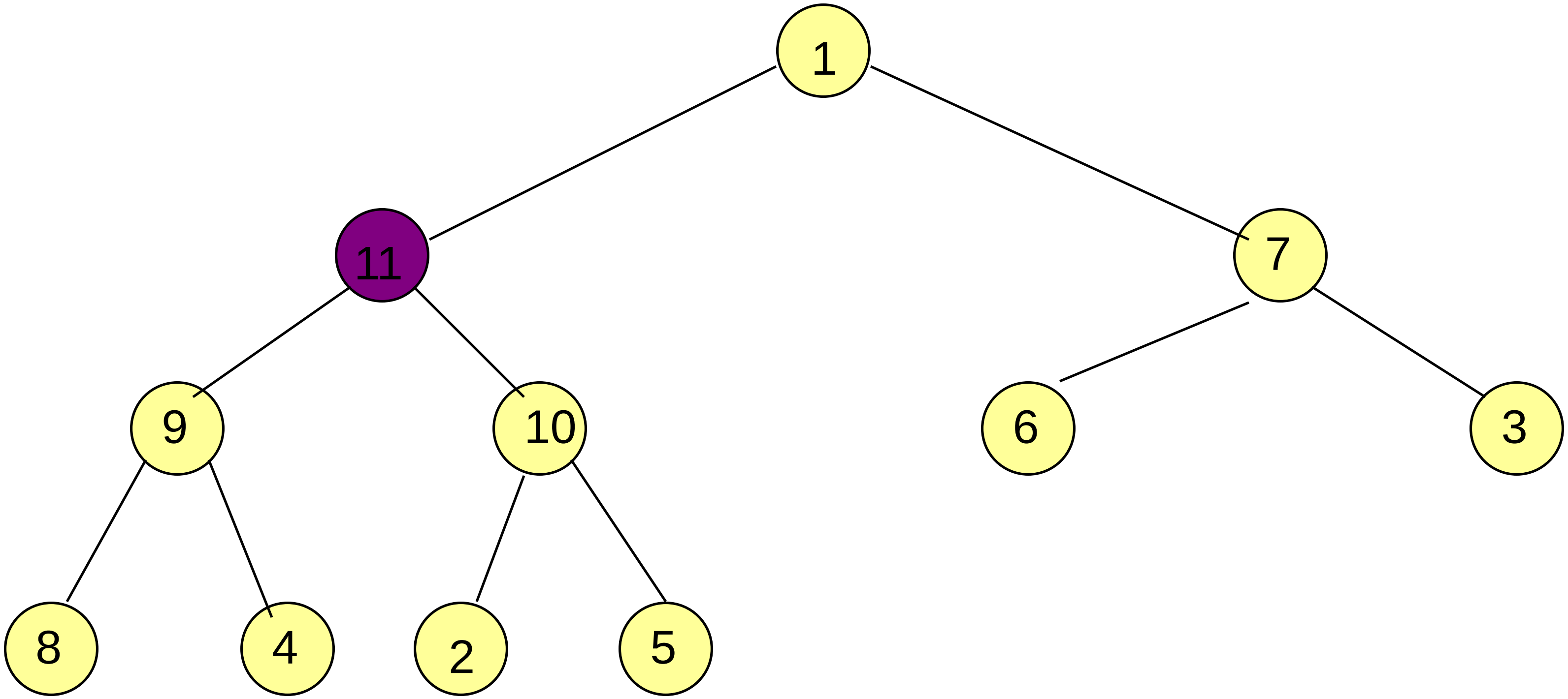
Find a home for 2.

Initializing A Max Heap



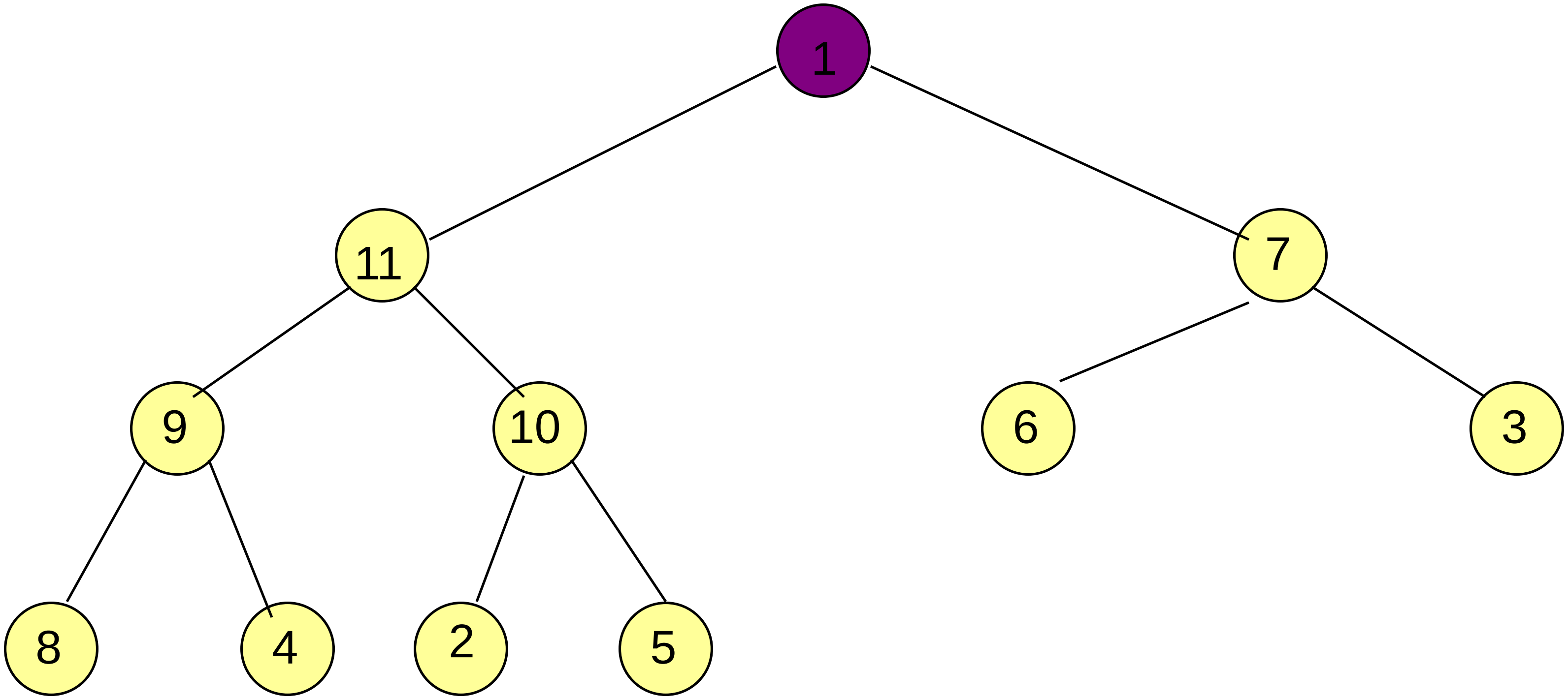
Find a home for 2.

Initializing A Max Heap



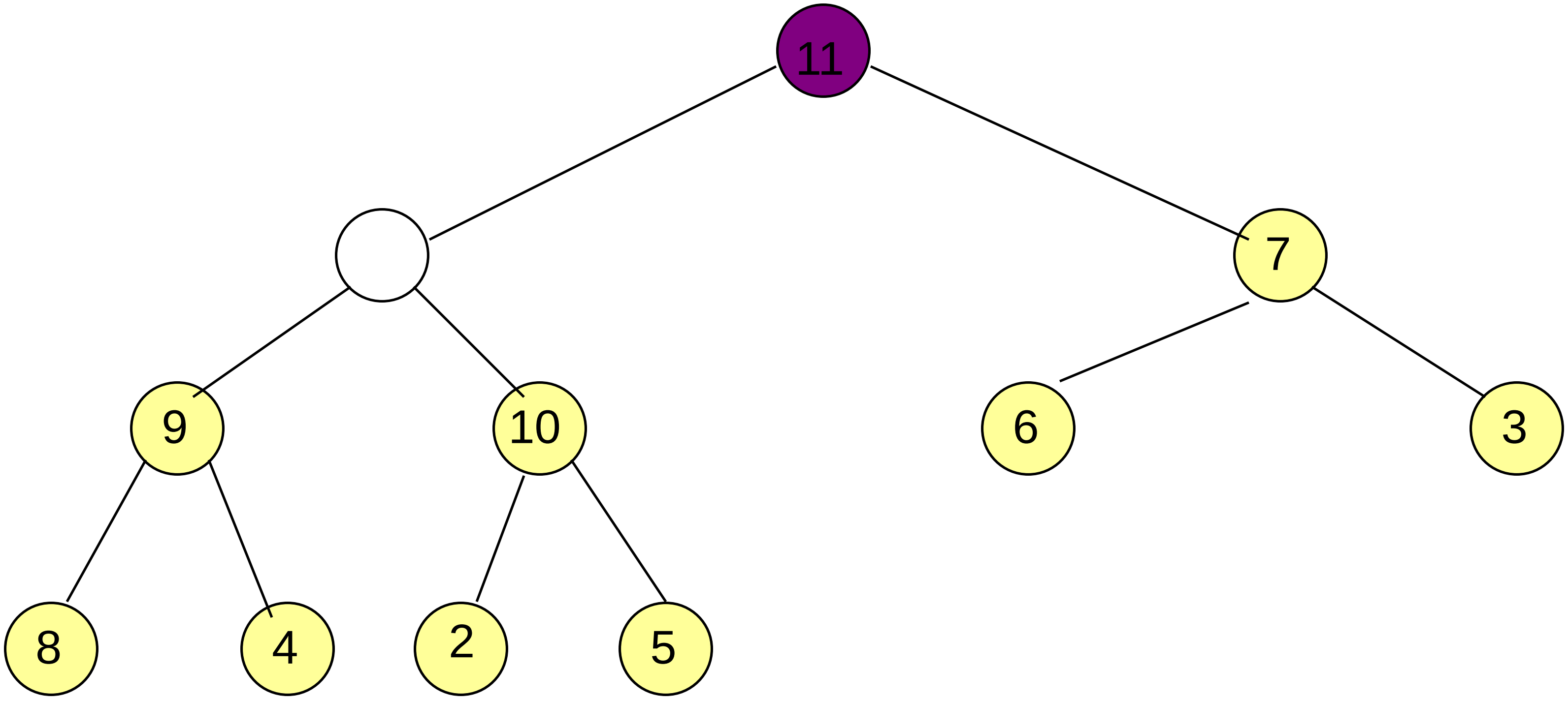
Done, move to next lower array position.

Initializing A Max Heap



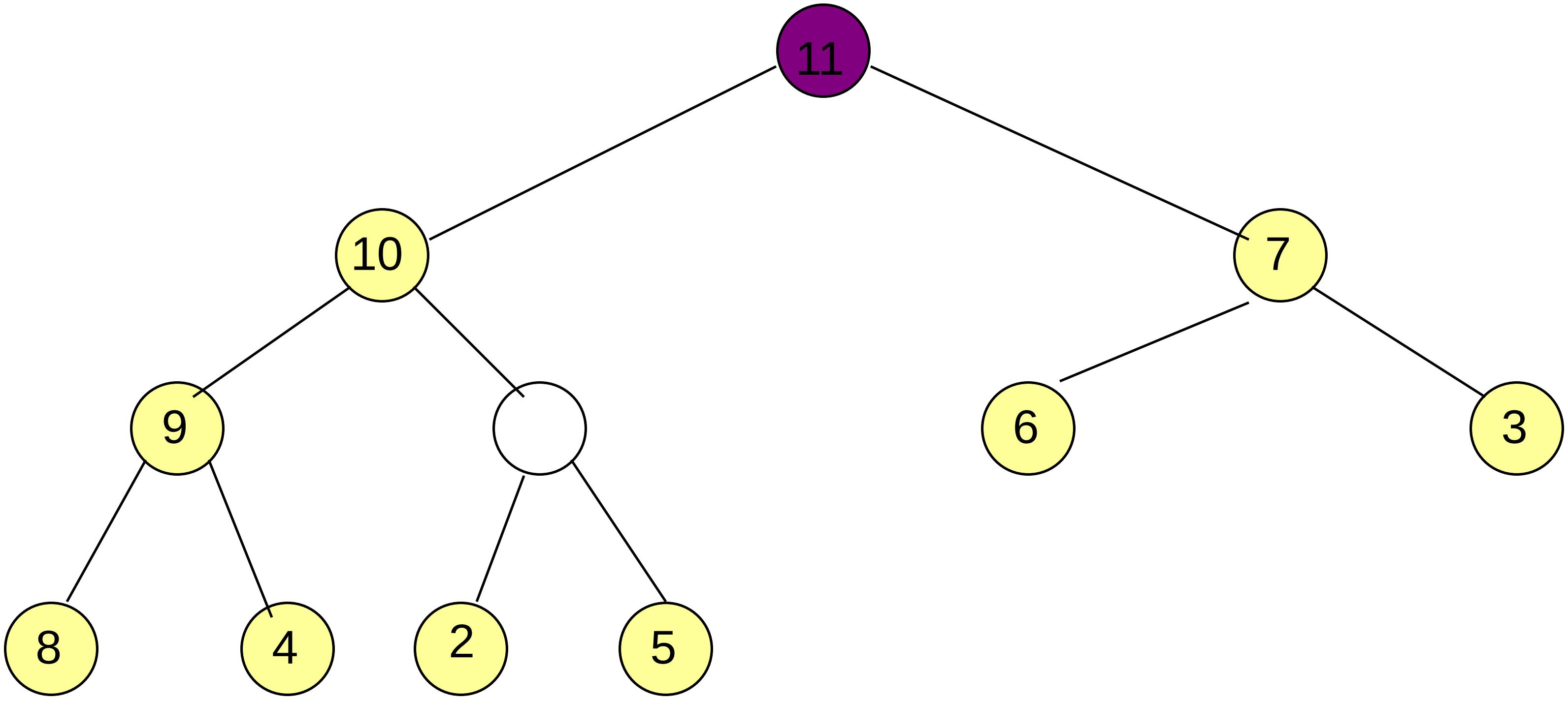
Find home for 1.

Initializing A Max Heap



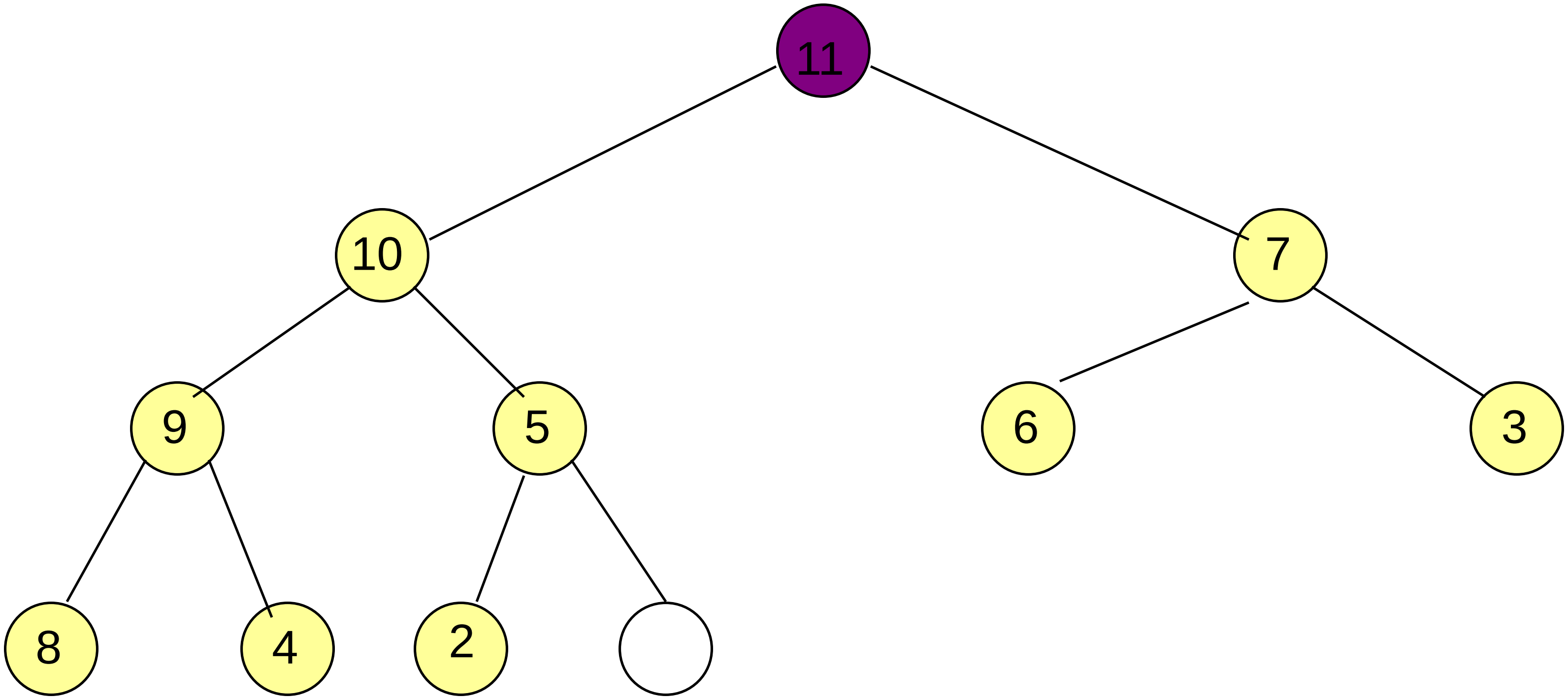
Find home for 1.

Initializing A Max Heap



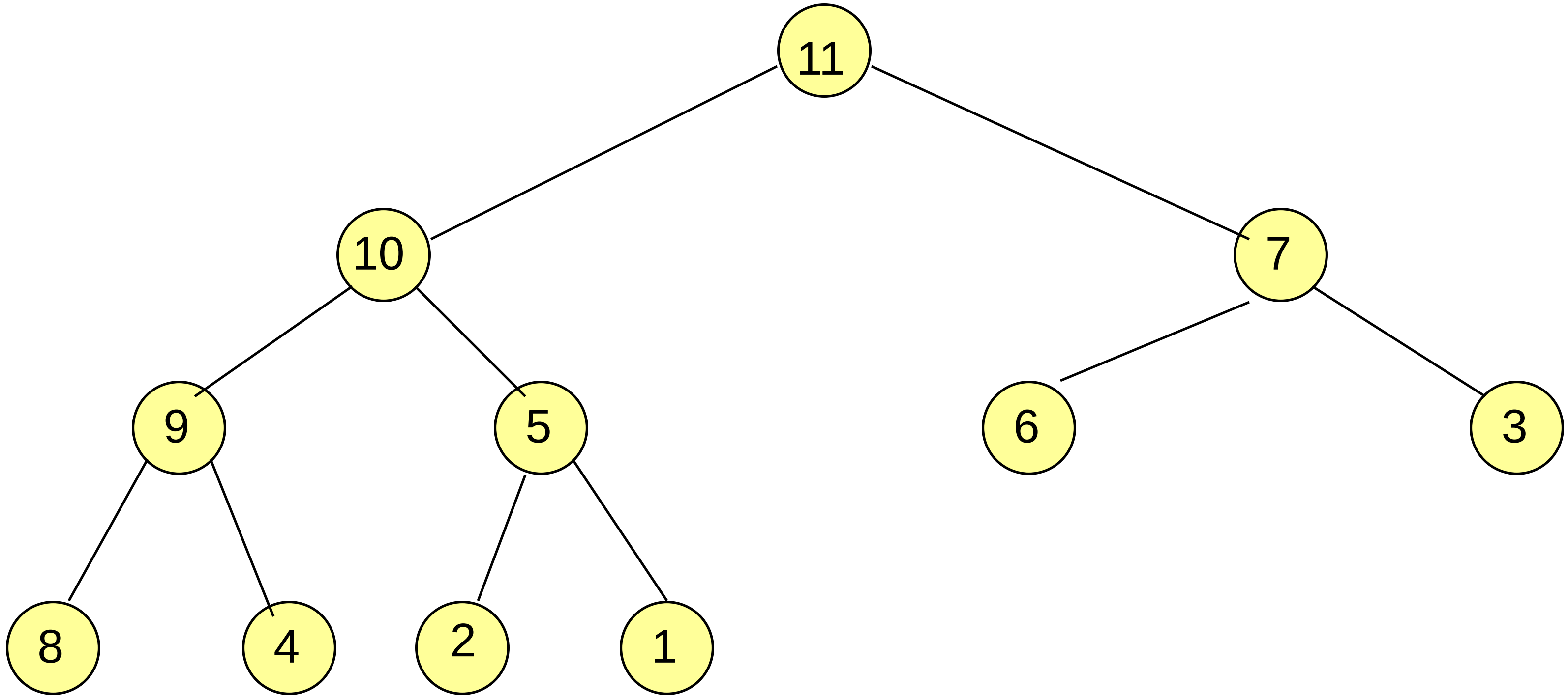
Find home for 1.

Initializing A Max Heap



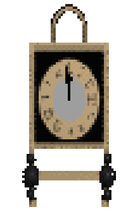
Find home for 1.

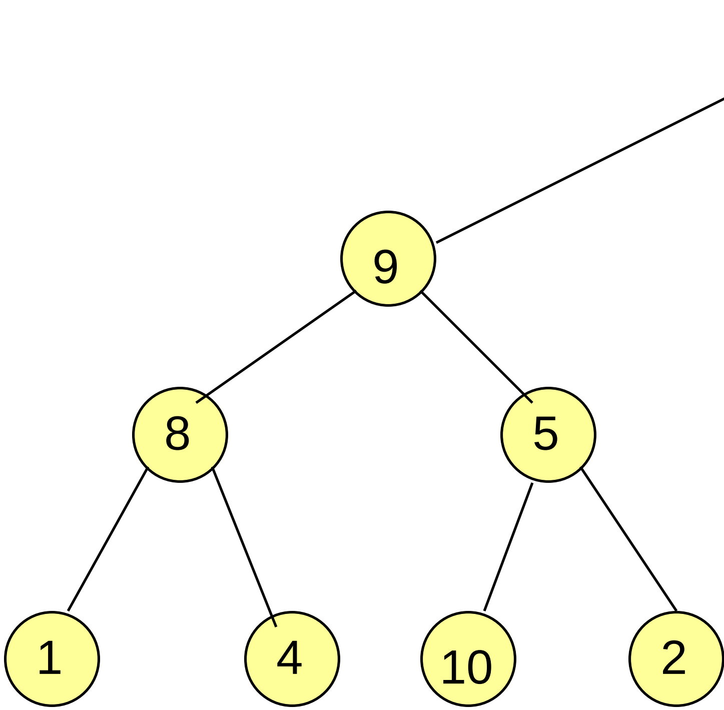
Initializing A Max Heap



Done.

Time Complexity





$$O\left(\sum_{j=1}^{h-1} 2^{j-1}(h-j+1)\right) = O(2^h) = O(n)$$

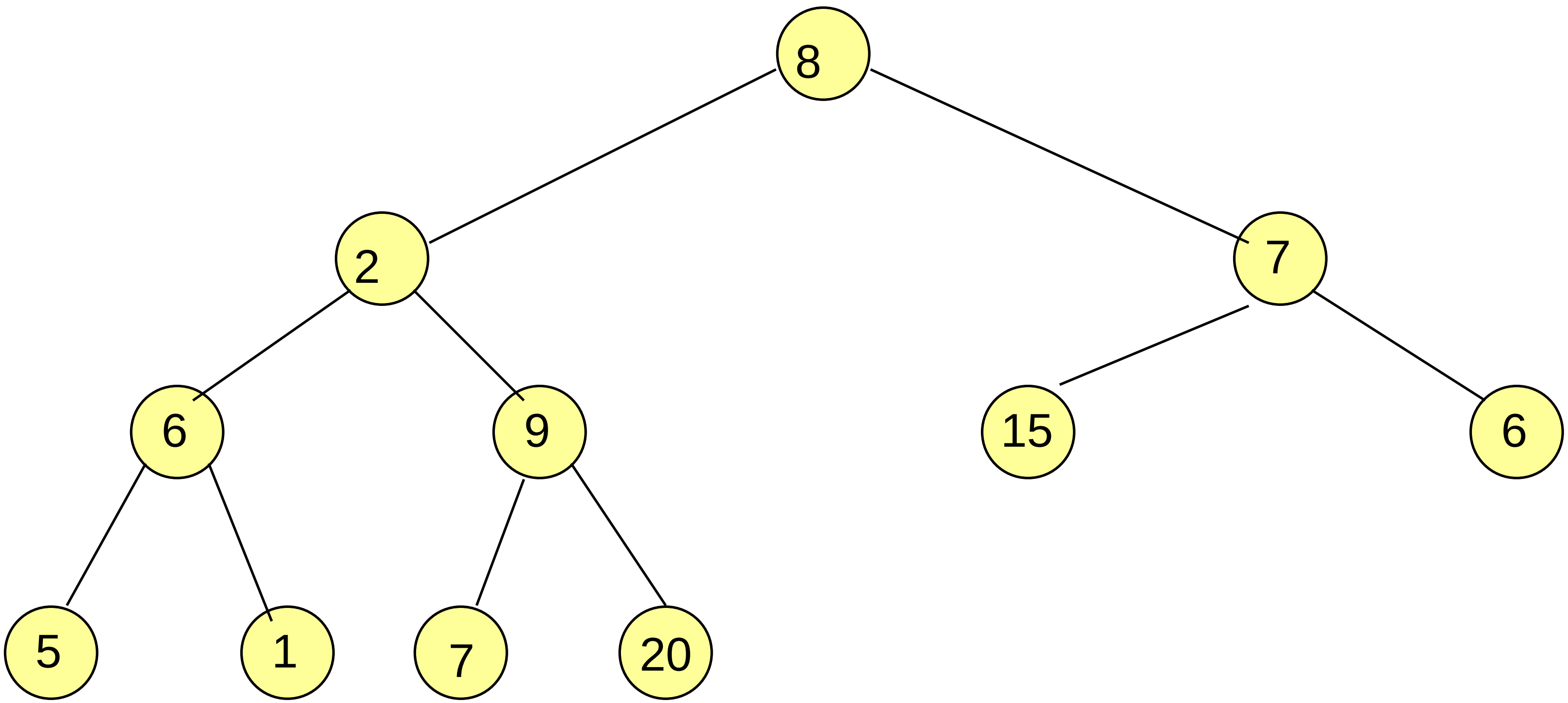
1. For each subtree, it takes O(h_i) time where h_i is the height of subtree with root i.

2. It has at most 2^{j-1} nodes at level j.

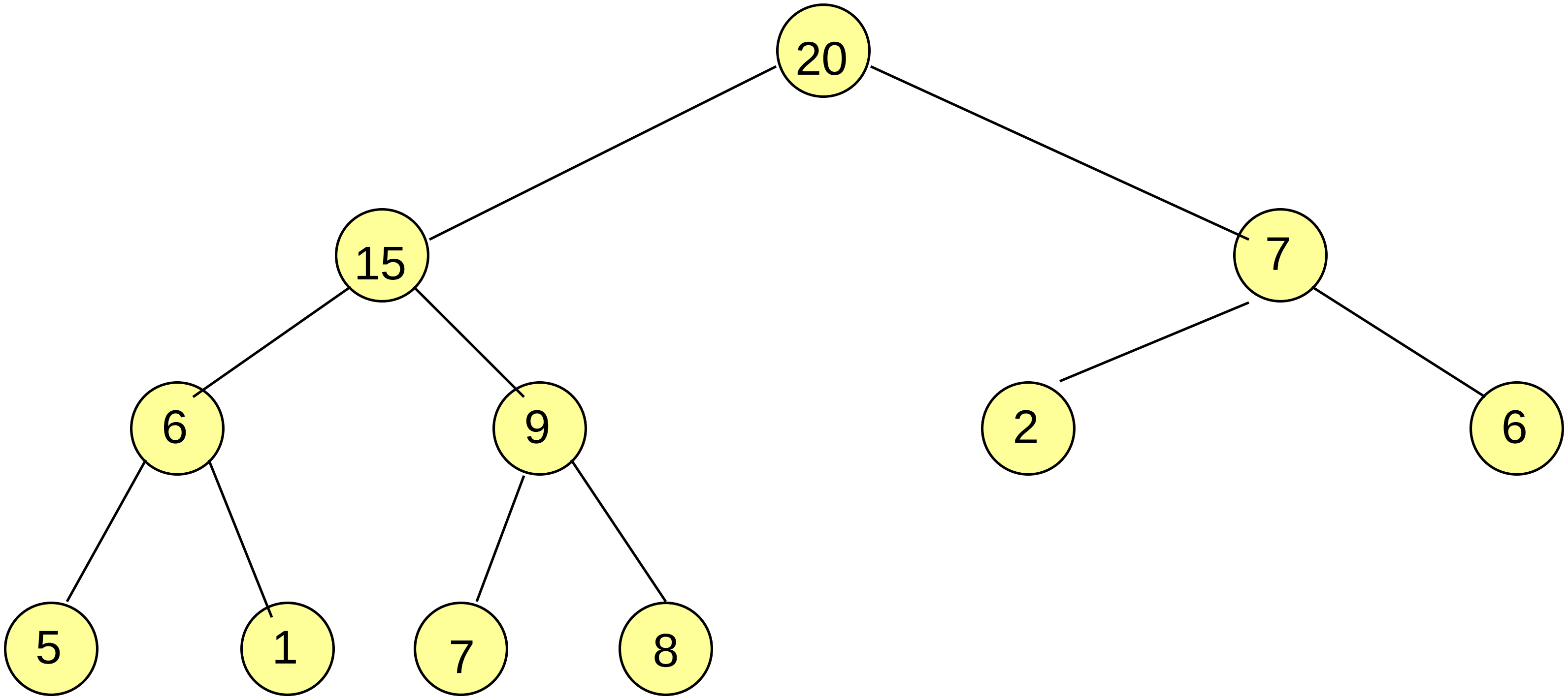
3. Hence at most 2^{j-1} of subtrees have $h_i = h\text{-}j\text{+}1$.

8.10 Applications of Heaps

Heap Sort

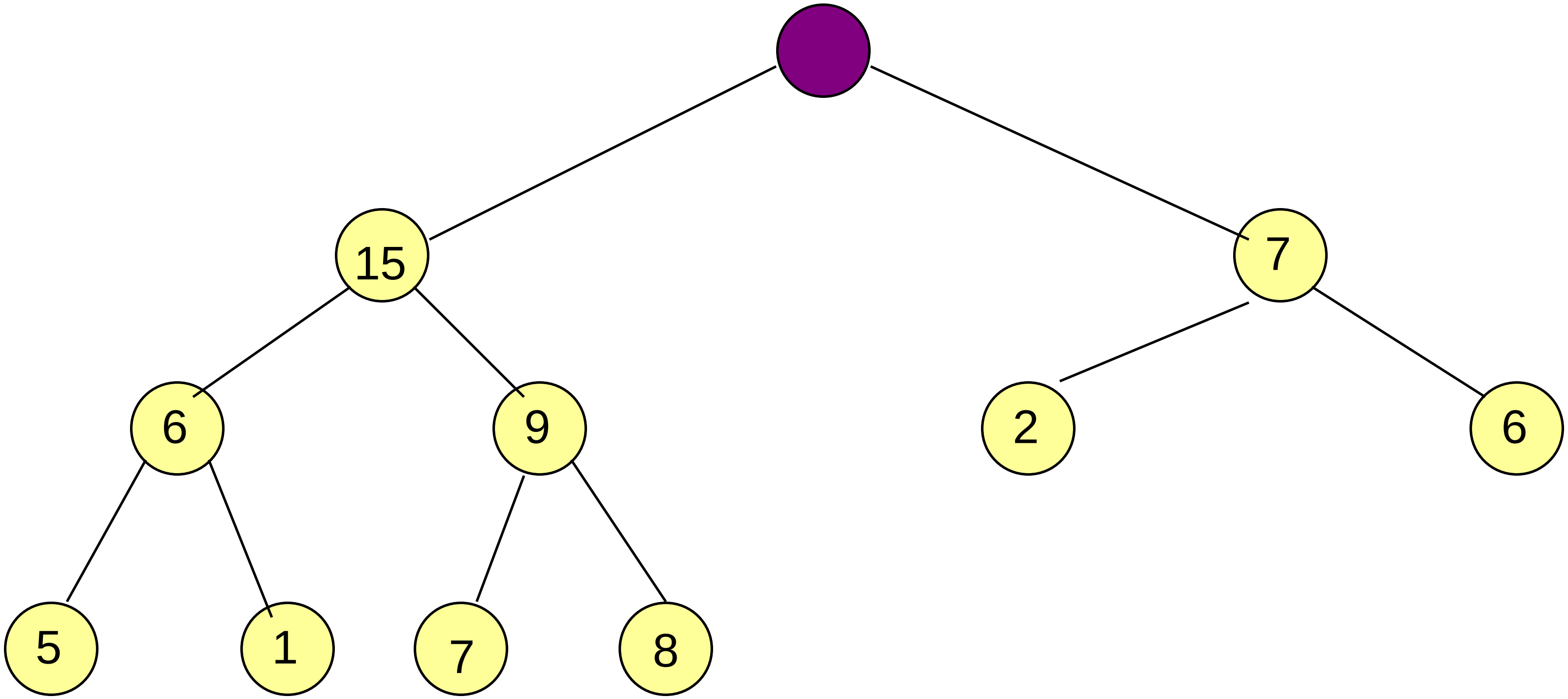


Removing The Max Element



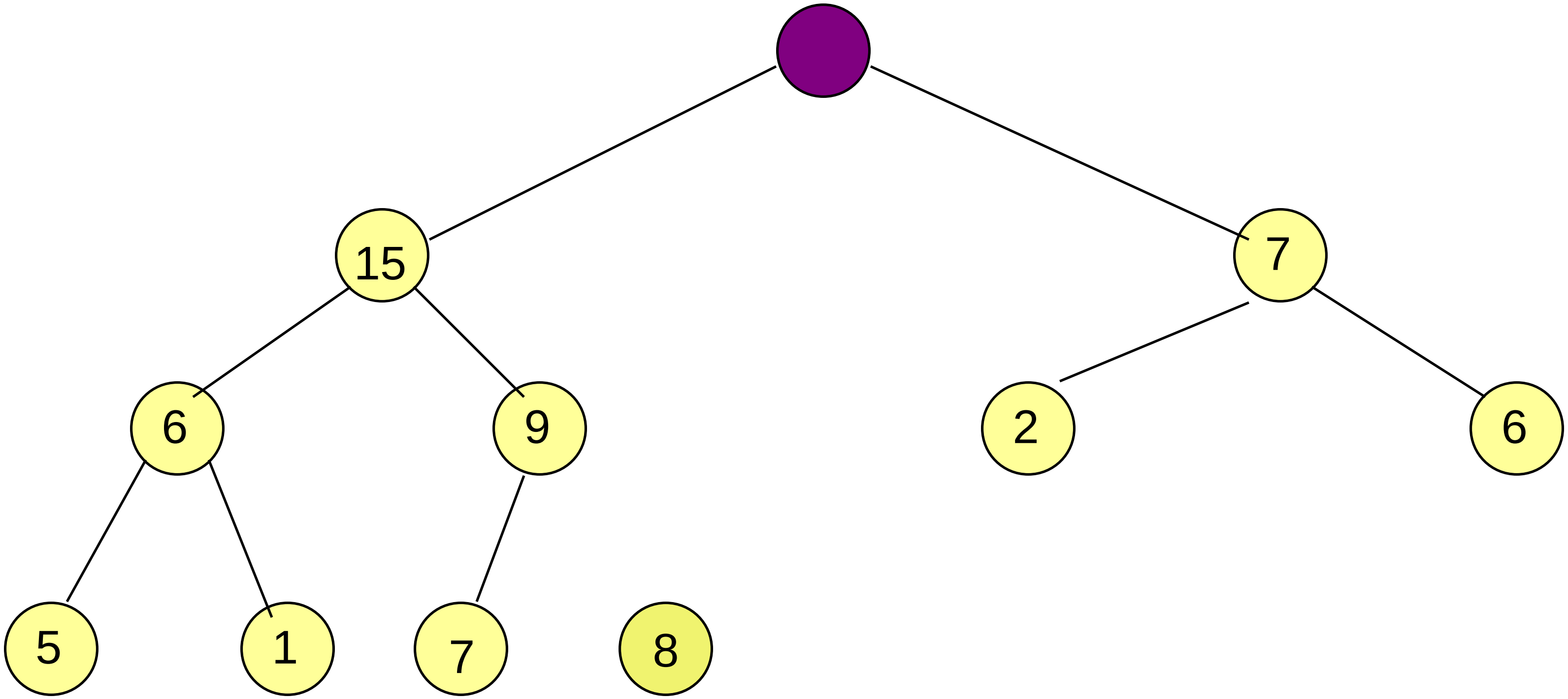
The initialization takes take Θ(n) time, and each deletion takes O(log*n*) time (there are *n* deletions). So the total time is O(*n*log(*n*)).

Removing The Max Element



After max element is removed.

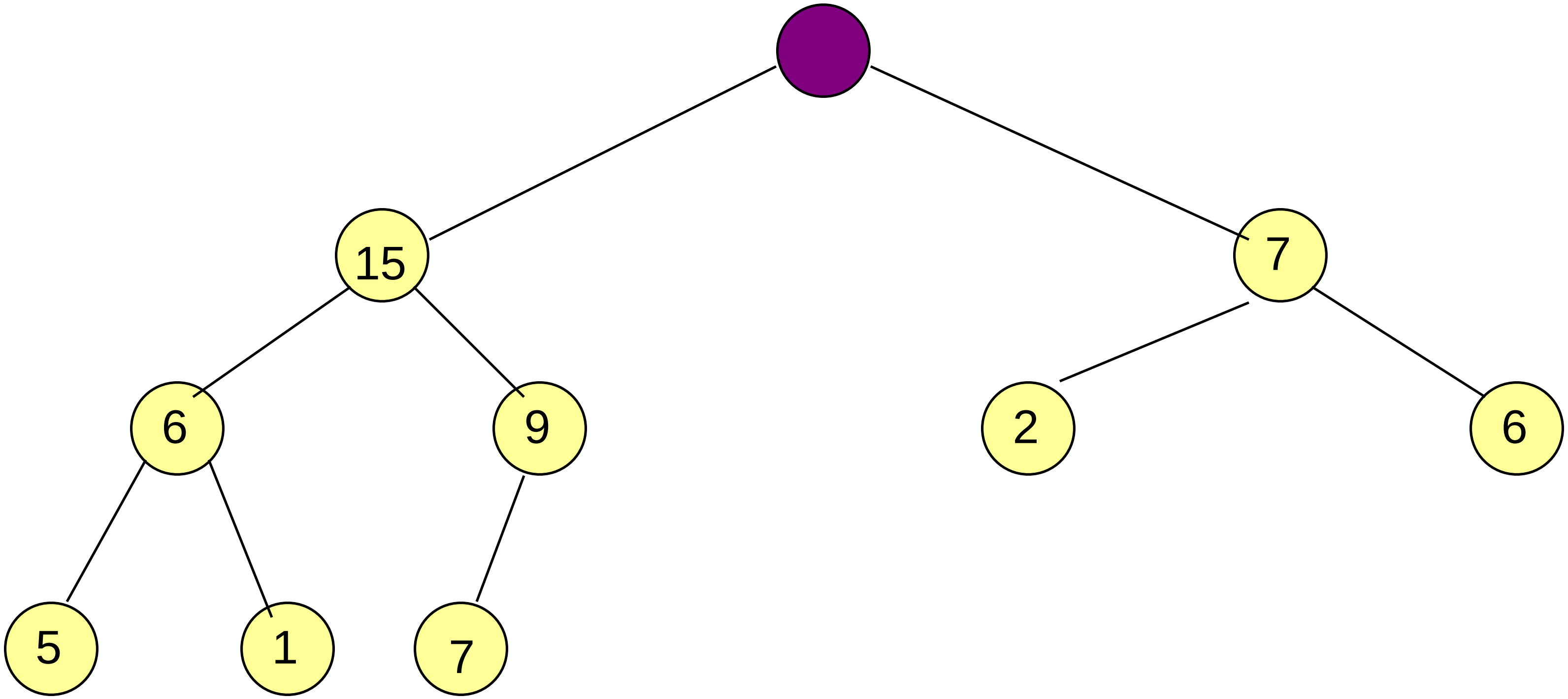
Removing The Max Element



Heap with 10 nodes.

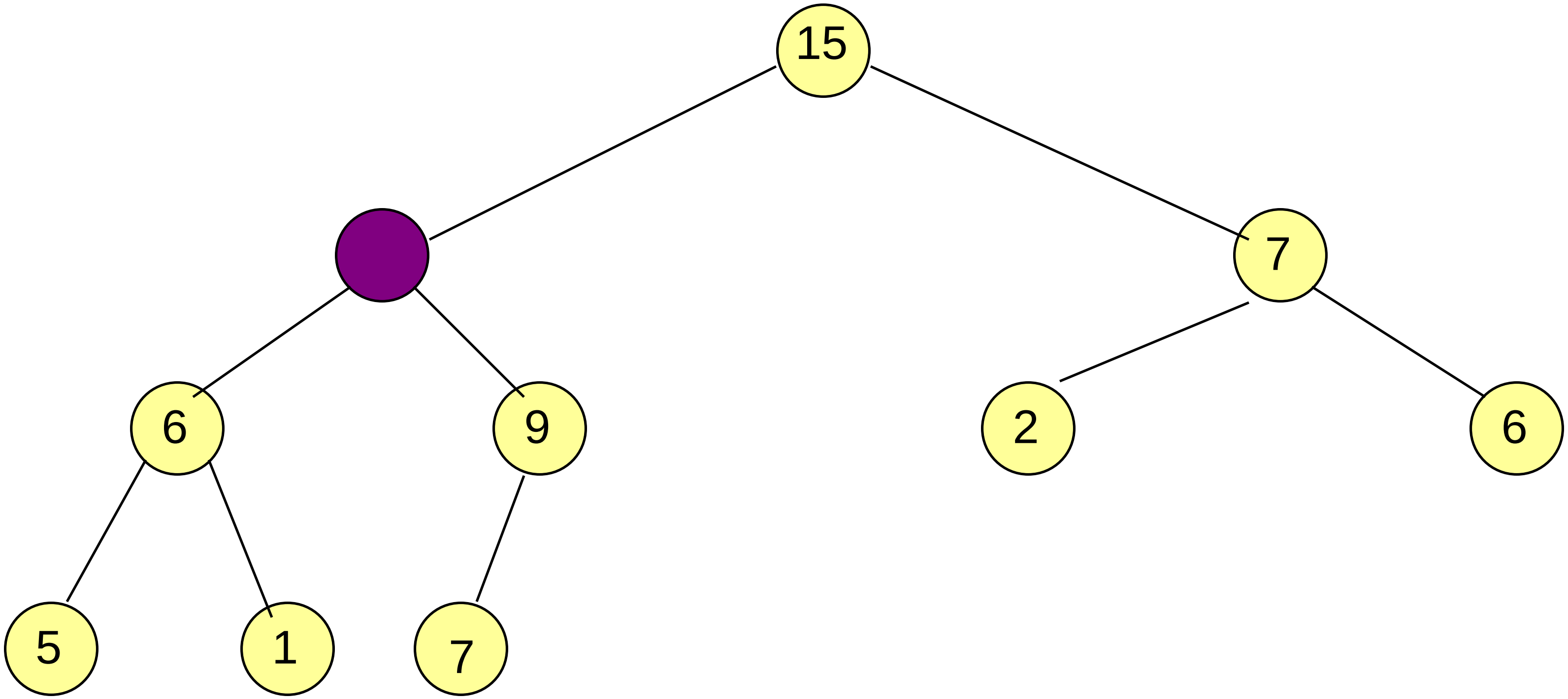
Reinsert 8 into the heap.

Removing The Max Element



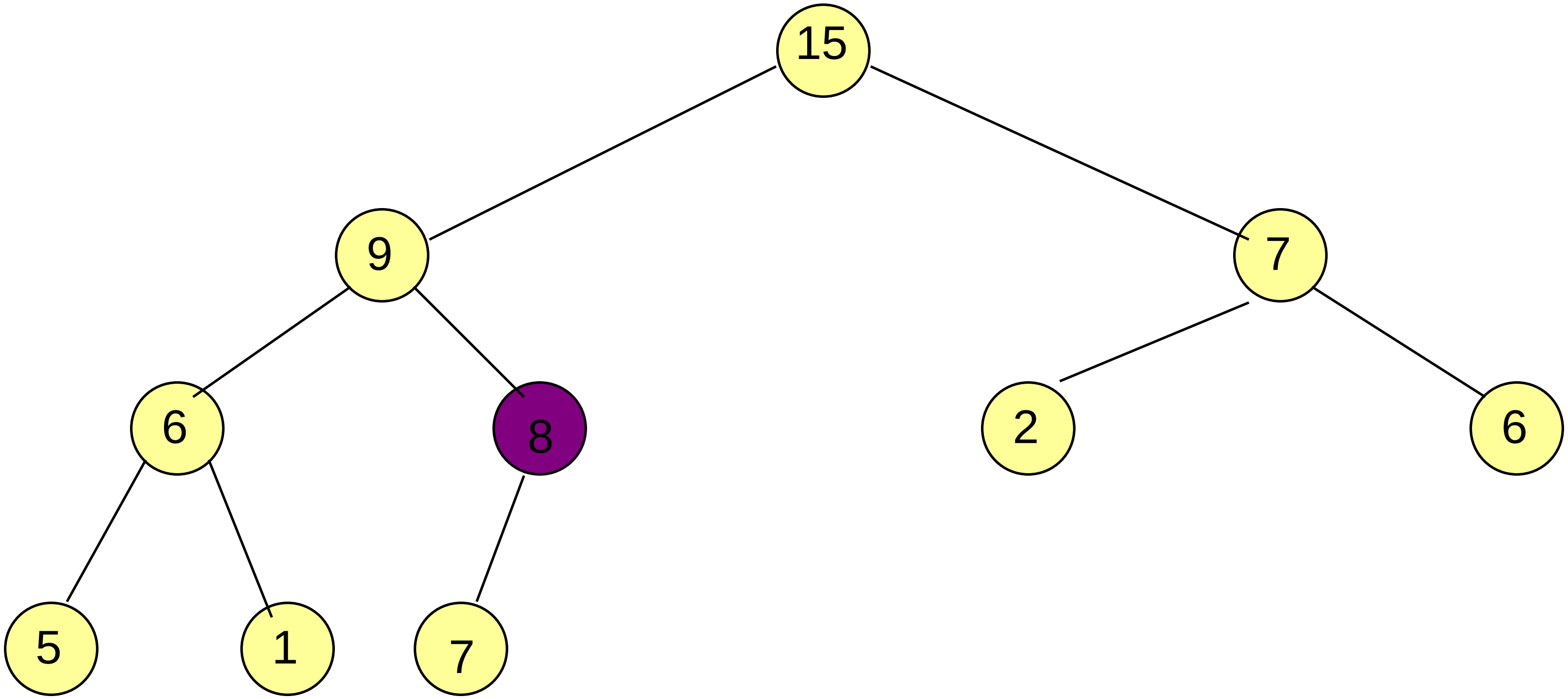
Reinsert 8 into the heap.

Removing The Max Element



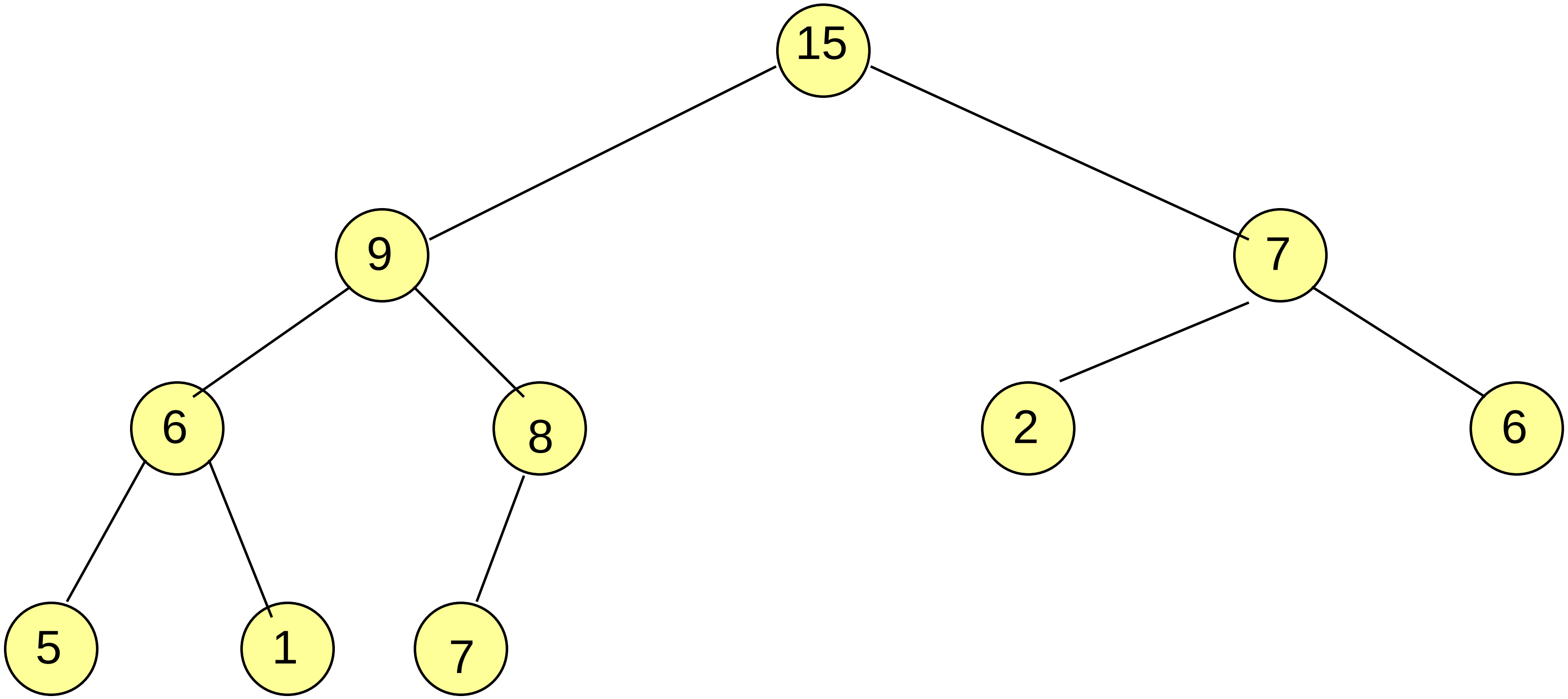
Reinsert 8 into the heap.

Removing The Max Element



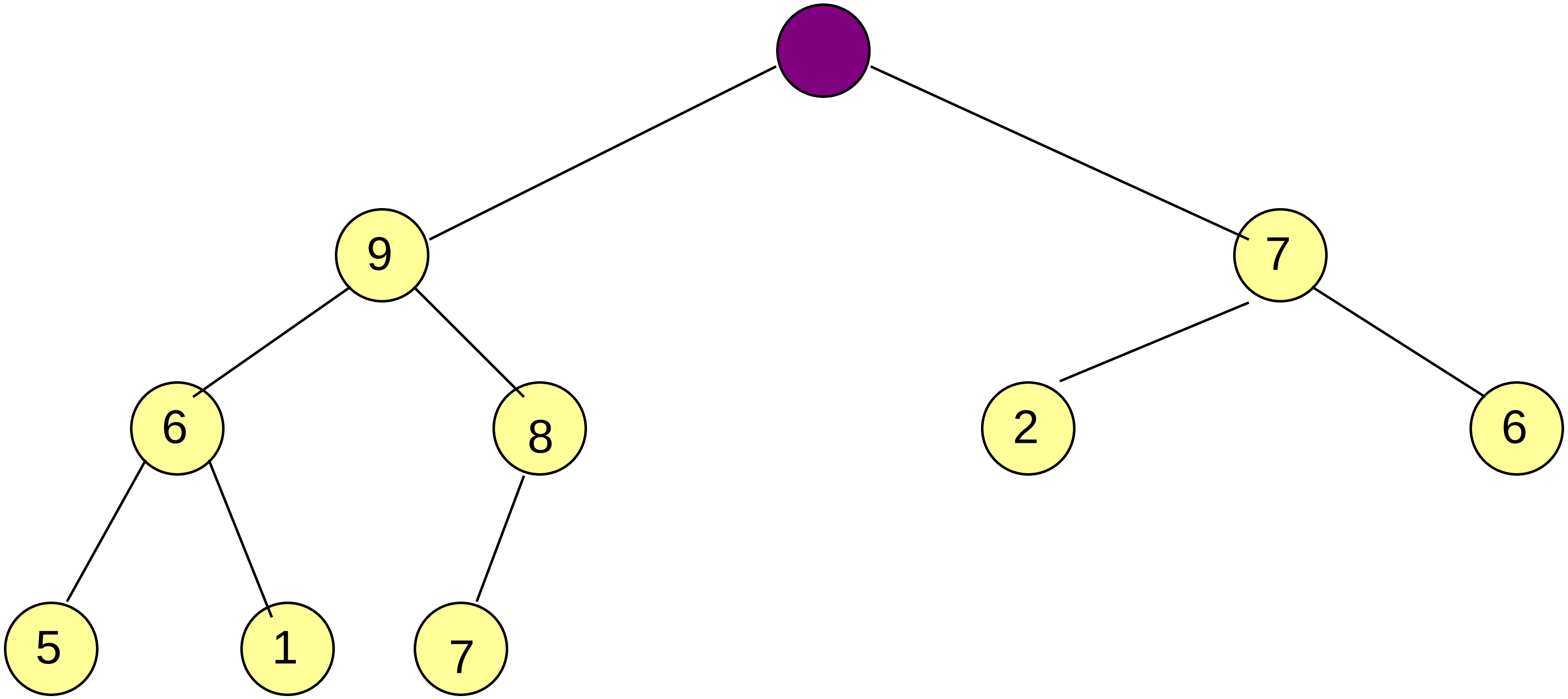
Reinsert 8 into the heap.

Removing The Max Element



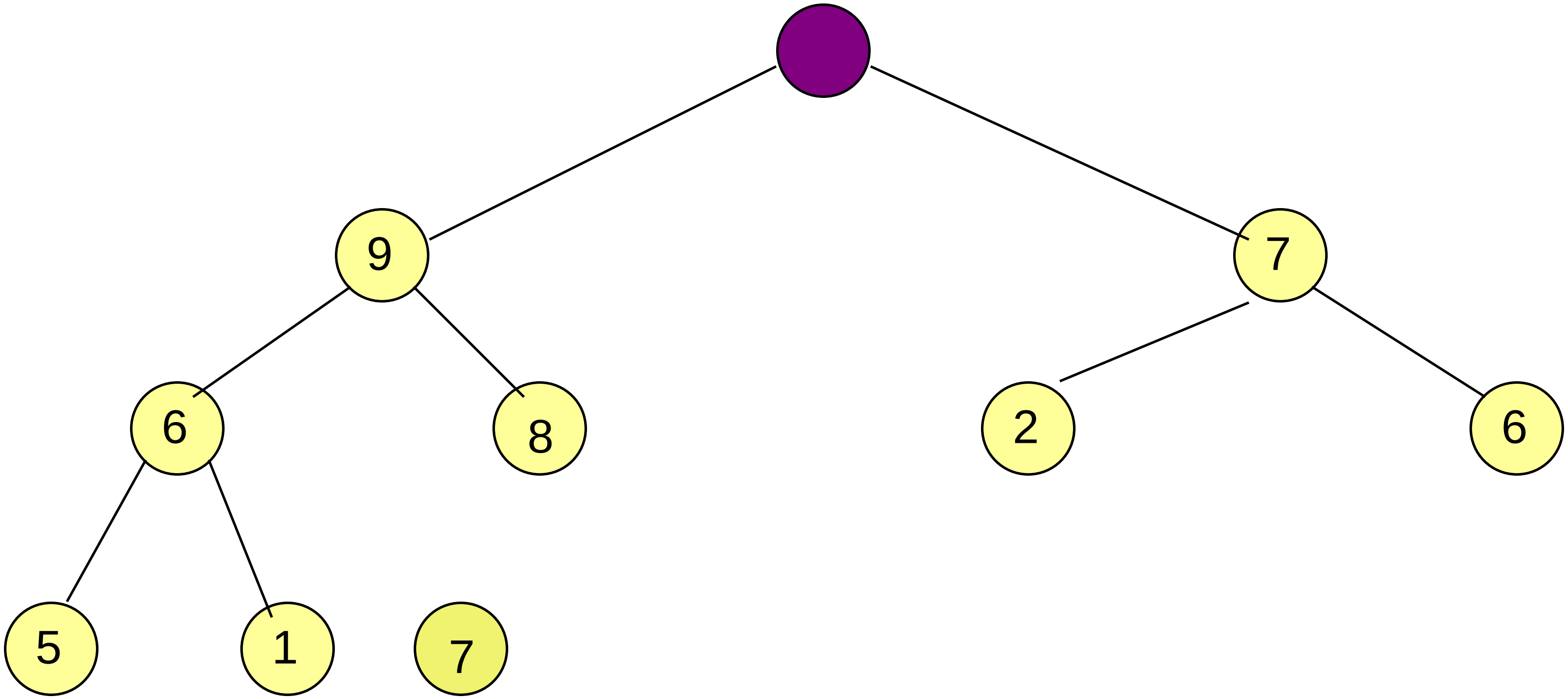
Max element is 15.

Removing The Max Element



After max element is removed.

Removing The Max Element



Heap with 9 nodes.