Nonlinear Data Structure

WF

http://smartlead.hallym.ac.kr

Instructor: Jin Kim

010-6267-8189(033-248-2318)

jinkim@hallym.ac.kr

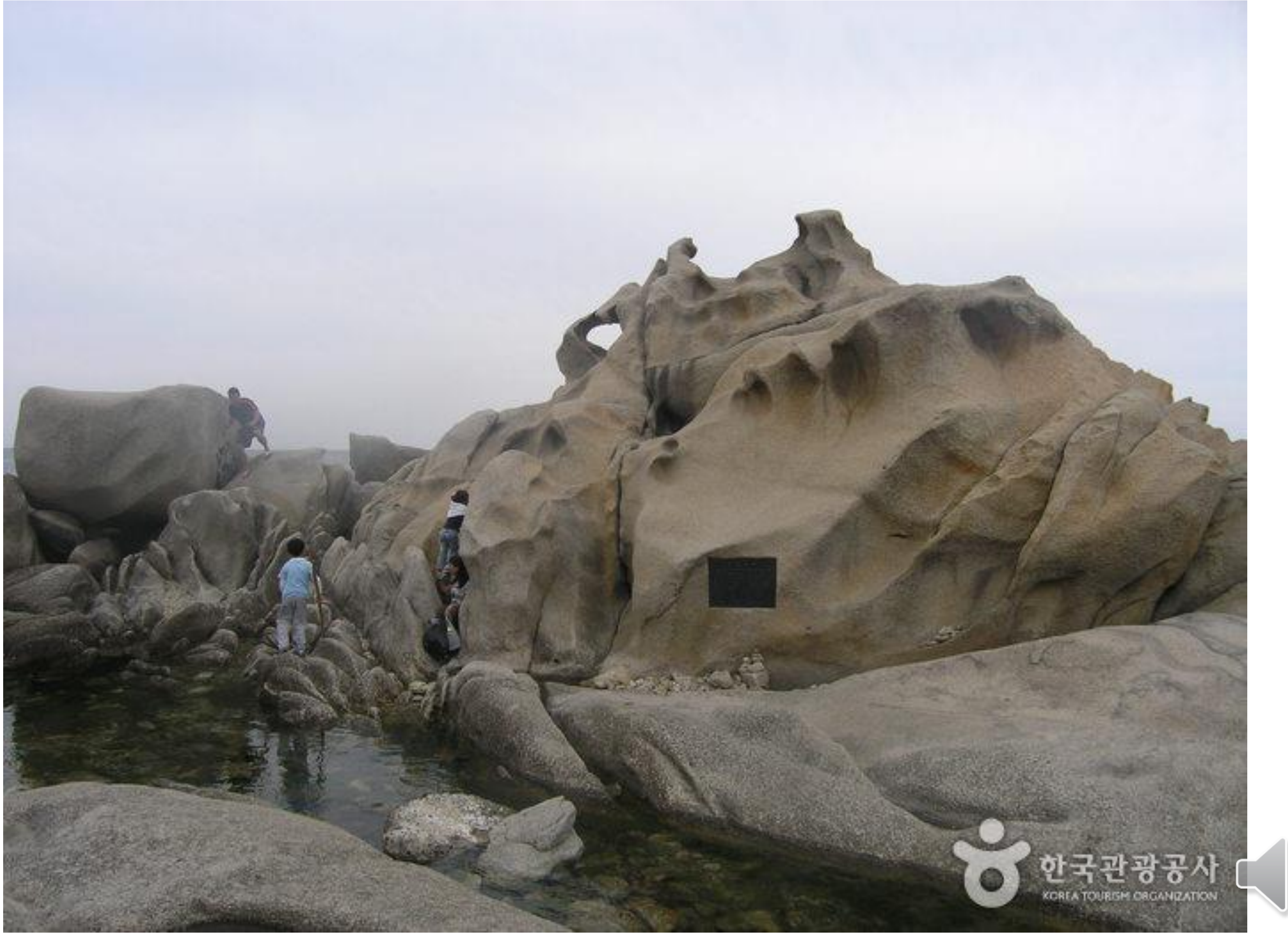
Office Hours:

Non-linear Data Structure

- Data structure we will consider this semister:
    - Tree
    - Binary Search Tree
    - Graph
    - Weighted Graph
    - ➡ Sorting
    - Balanced Search Tree

주문진

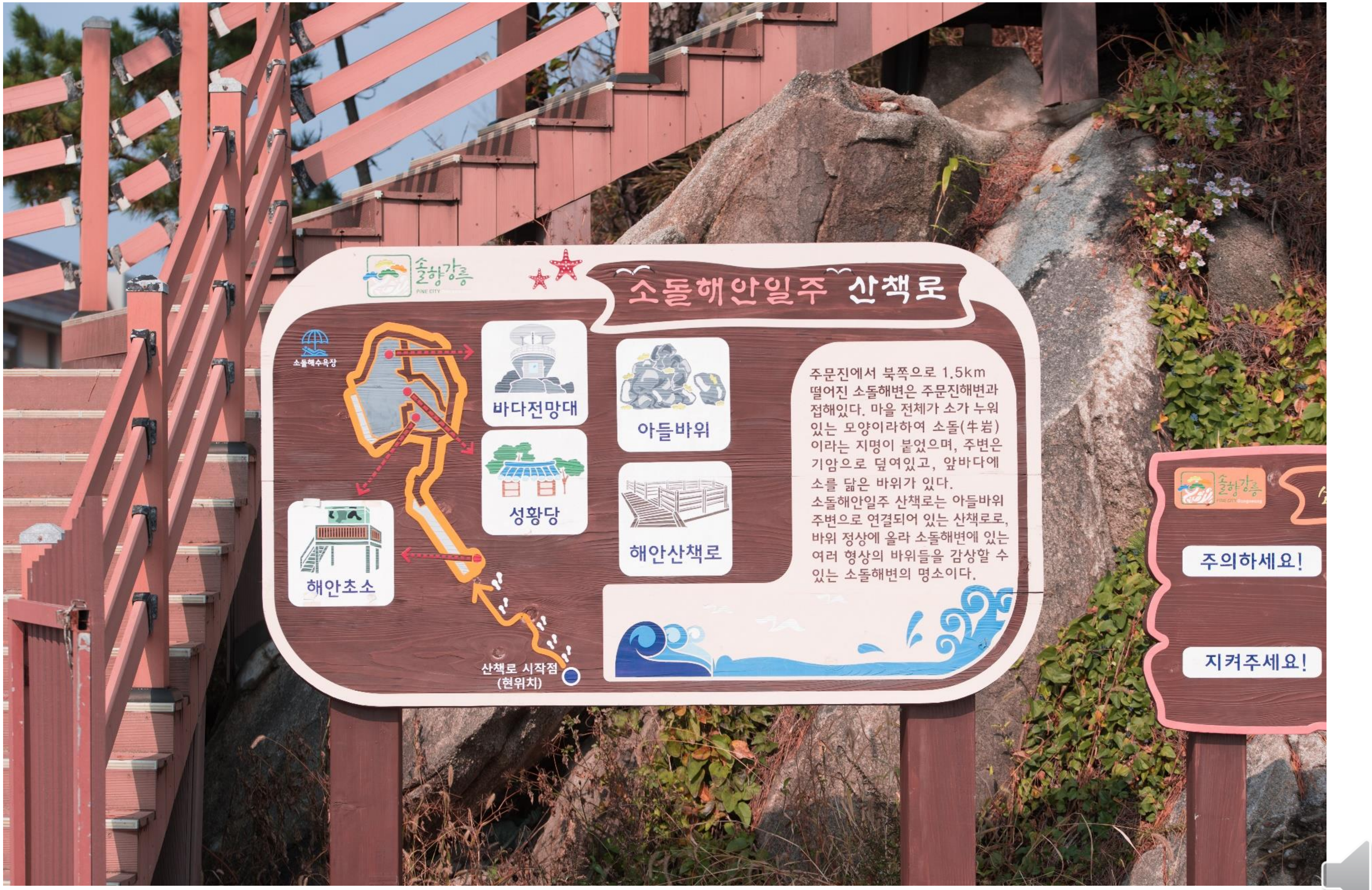
주문진 아들바위



주문진 아들바위

- 강릉시 북쪽의 해안도로를 따라 올라가다 언덕 끝자락의 주문진의 하얀 등대 너머 위치한 소돌아들바위공원에서는 바람과 파도에 깍인 절묘하고 기괴한 모습의 기암 괴석들을 볼 수 있다. 도로쪽에서 보면 거무튀튀하고 날카롭게 각진 바위가 마치 힘센 수소를 연상케하며, 코끼리바위 등 자연 형성된 신기하고 기괴한 바위들로 가득하다.
- 소돌바위는 일억 오천만년 쥬라기시대에 바다 속에 있다가 지각변동으로 인하여 지상에 솟은 바위로 먼 옛날 노부부가 백일 기도 후 아들은 얻은 후로 자식이 없는 부부들이 기도를 하면 소원을 성취한다는 전설이 있어 신혼 부부들이 즐겨찾는 곳으로 유명하다.

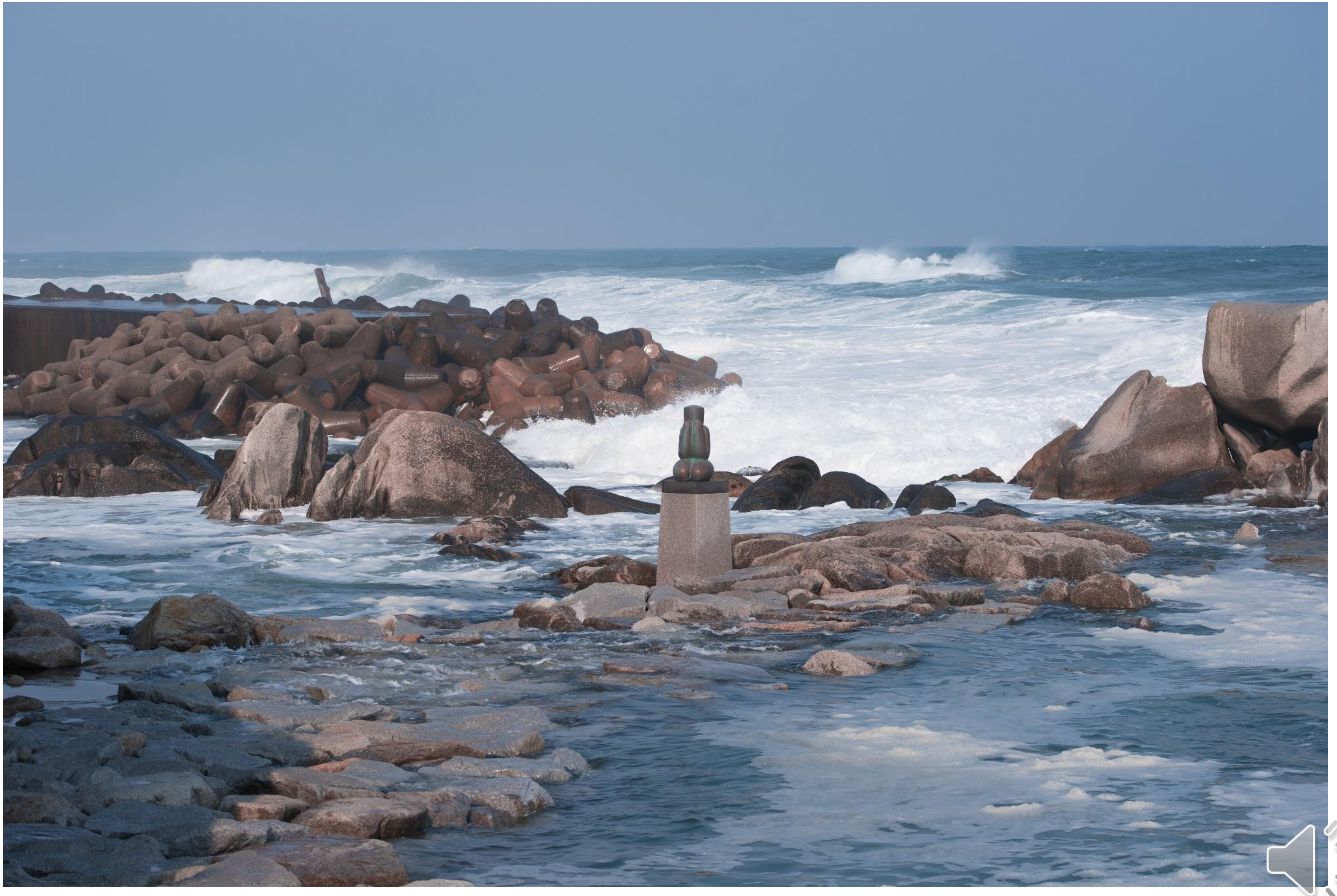
주문진 아들바위



주문진에서 북쪽으로 1.5km 떨어진 소돌해변은 주문진해변과 접해있다. 마을 전체가 소가 누워 있는 모양이라하여 소돌(牛岩)이라는 지명이 붙었으며, 주변은 기암으로 덮여있고, 앞바다에 소를 닮은 바위가 있다.

소돌해안일주 산책로는 아들바위 주변으로 연결되어 있는 산책로로, 바위 정상에 올라 소돌해변에 있는 여러 형상의 바위들을 감상할 수 있는 소돌해변의 명소이다.

주문진 아들바위



주문진 아들바위 기도

$\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$정렬 알고리즘

Quick sort(퀵정렬)

Merge sort(합병정렬)

Heap sort(히프정렬)

Quick Sort(퀵 정 렬)

Partition(파티션)

사무실 또는 전시장 등의 특정 공간과 다른 공간의 사이를 나누는 벽

Example 1

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.

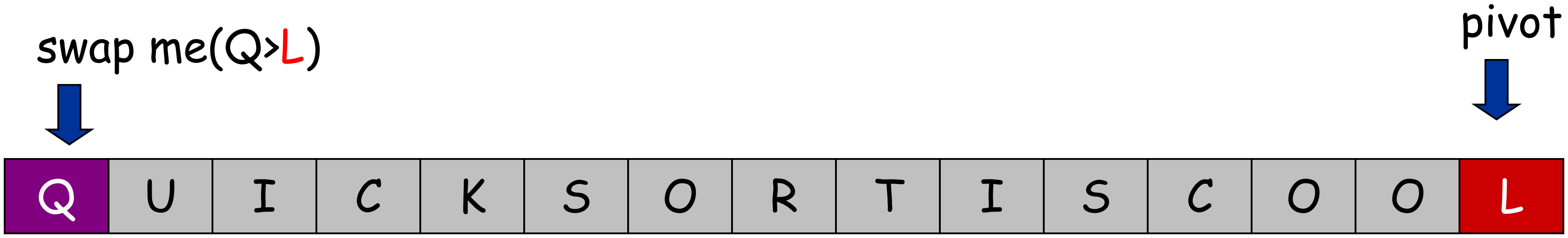
Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



swap me(Q>L)

pivot

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

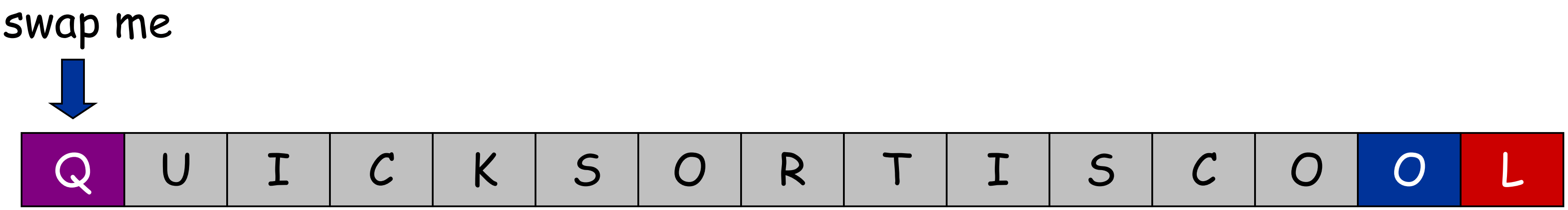
🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



swap me

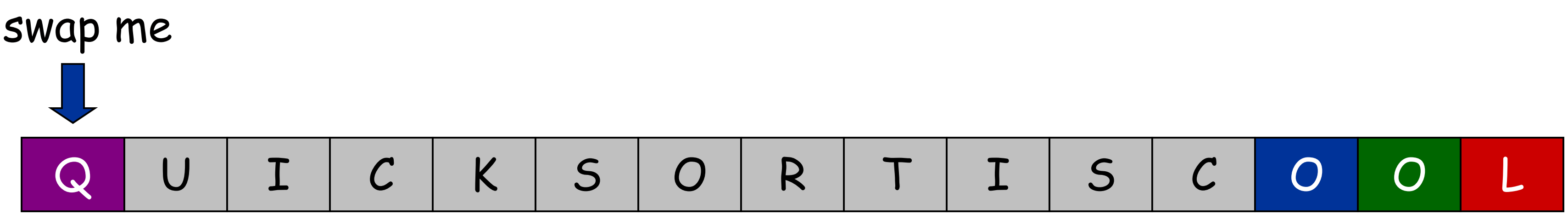
Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.


swap me

Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

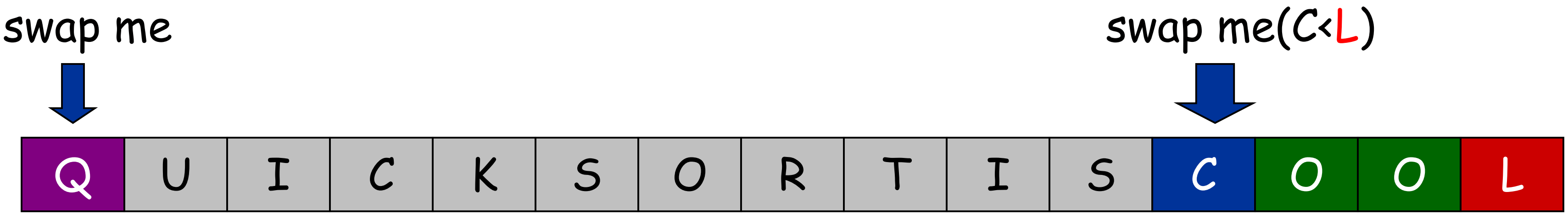
🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



Q	U	I	C	K	S	O	R	T	I	S	C	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.

🟩 C	⬜ U	⬜ I	⬜ C	⬜ K	⬜ S	⬜ O	⬜ R	⬜ T	⬜ I	⬜ S	🟩 Q	🟩 O	🟩 O	🟥 L

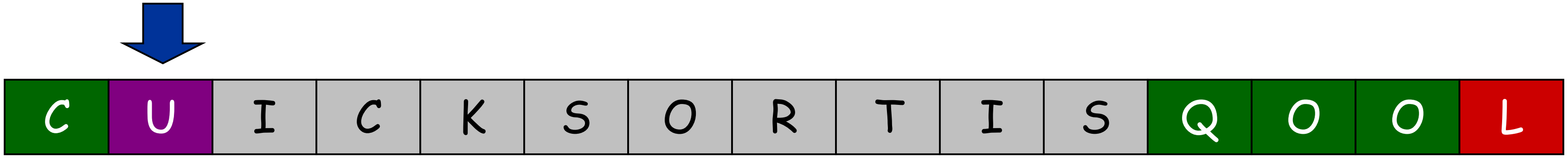
🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.

swap meU>L



C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

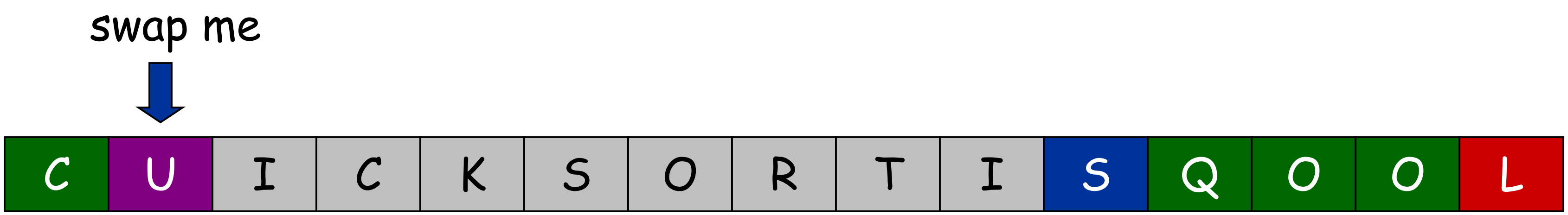
🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



swap me

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

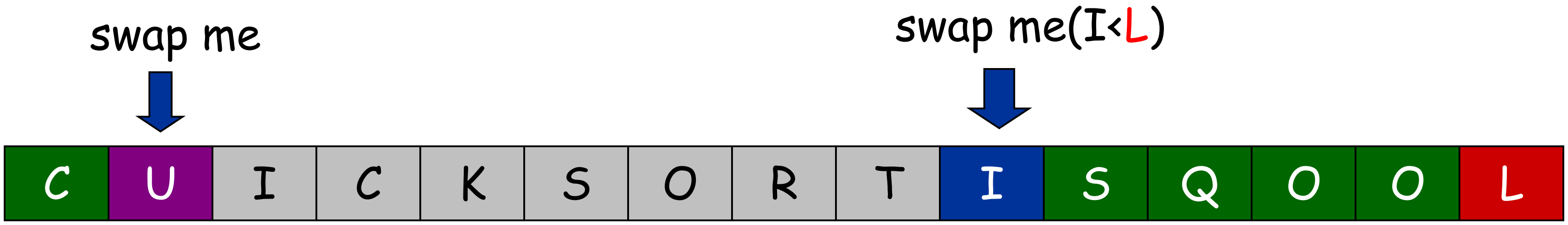
🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



swap me

swap me(I<L)

C	U	I	C	K	S	O	R	T	I	S	Q	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

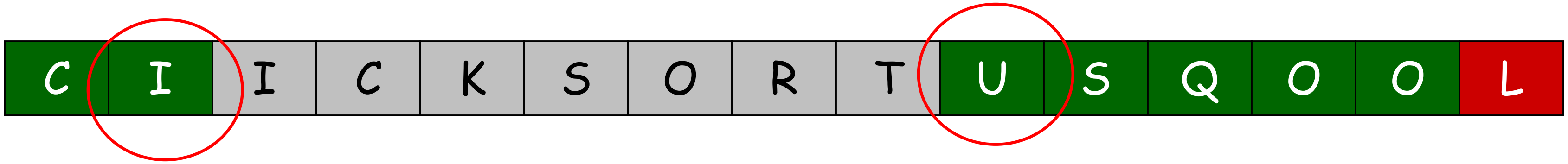
🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

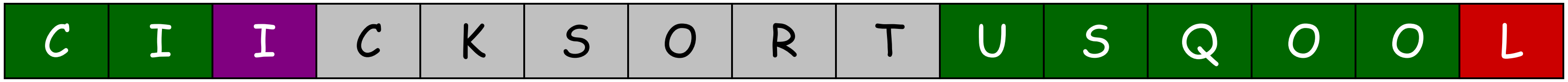
🟥 partition element ⬜ unpartitioned 🟪 left

🟩 partitioned 🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



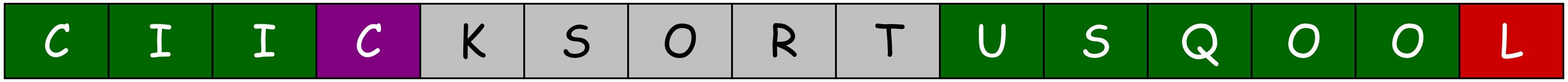
C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

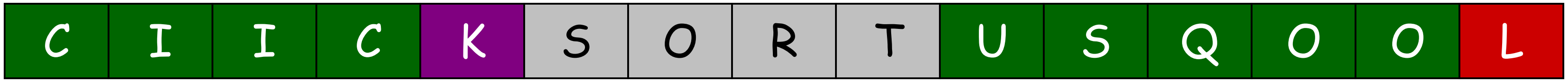
🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



🟥 partition element

⬜ unpartitioned

🟩 partitioned

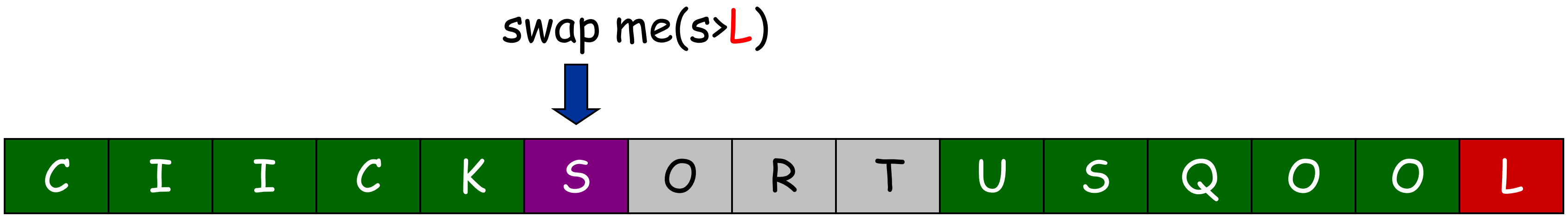
🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



swap me(s>L)

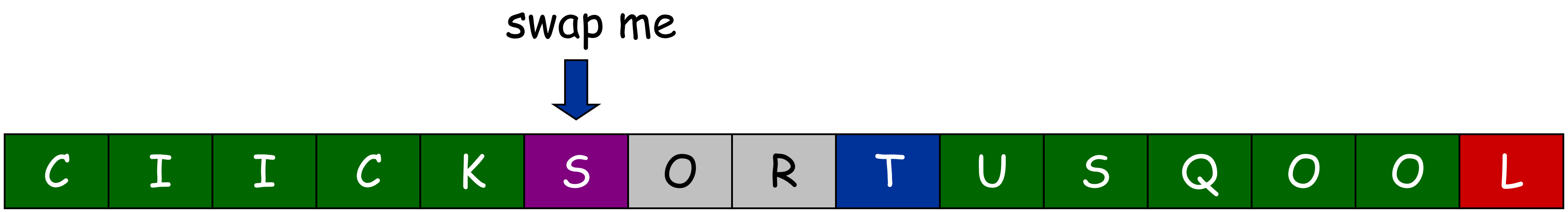
C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



swap me

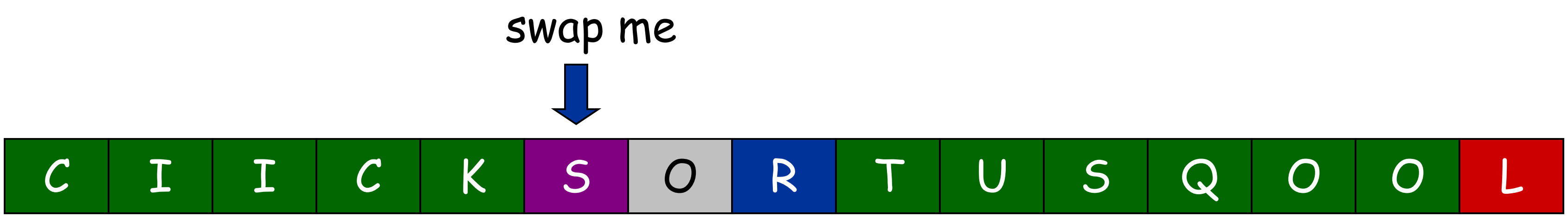
C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



swap me

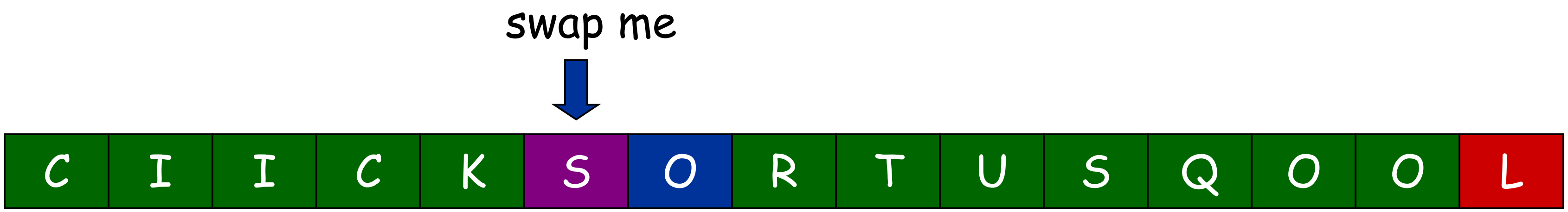
C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

🟪 left

🟦 right

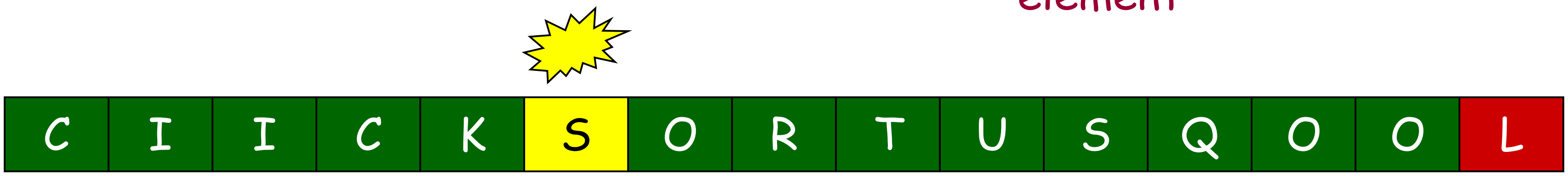
Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.

pointers cross

swap with partitioning element



C	I	I	C	K	S	O	R	T	U	S	Q	O	O	L

🟥 partition element

⬜ unpartitioned

🟩 partitioned

🟪 left

🟦 right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.

partition is complete

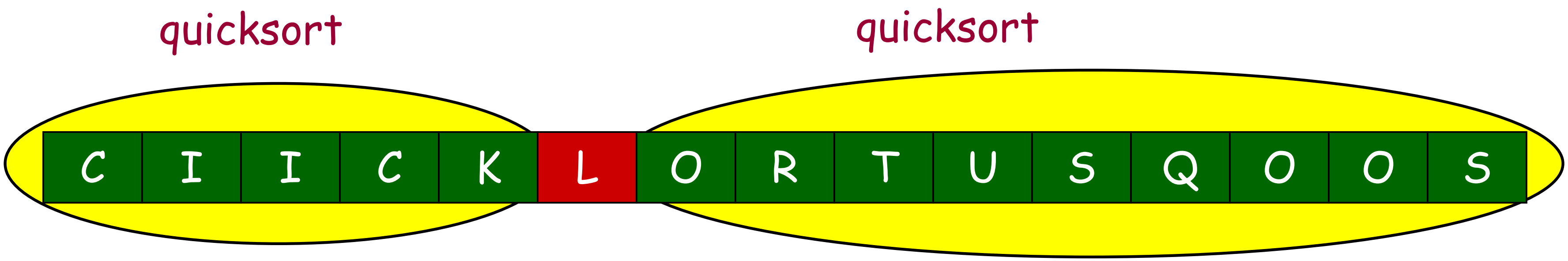
C	I	I	C	K	L	O	R	T	U	S	Q	O	O	S

🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Partitioning in Quicksort

How do we partition in-place efficiently?

- Partition element = rightmost element.
- Scan from left for larger element.
- Scan from right for smaller element.
- Exchange.
- Repeat until pointers cross.



C	I	I	C	K	L	O	R	T	U	S	Q	O	O	S

🟥	partition element	⬜	unpartitioned	🟪	left
		🟩	partitioned	🟦	right

Quick Sort

- **Divide**:
  - Pick any element **p** as the **pivot**, e.g, the first element
  - Partition the remaining elements into

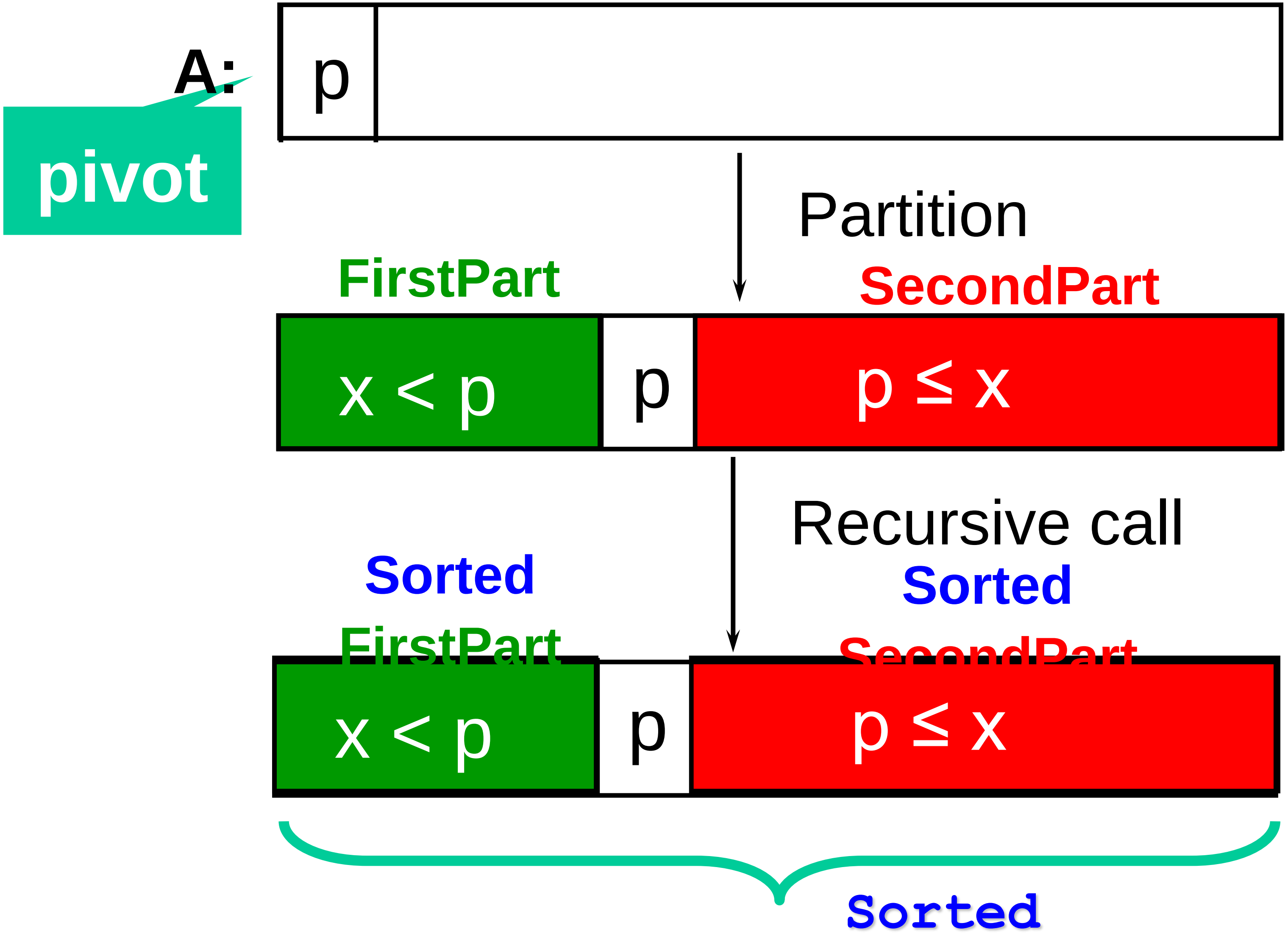
    FirstPart, which contains all elements $< p$

    SecondPart, which contains all elements $\geq p$

- **Recursively sort** the FirstPart and SecondPart

- **Combine**: no work is necessary since sorting is done in place

Quick Sort



A:

pivot

p

Partition

FirstPart

SecondPart

x < p

p

p ≤ x

Recursive call

Sorted

FirstPart

Sorted

SecondPart

x < p

p

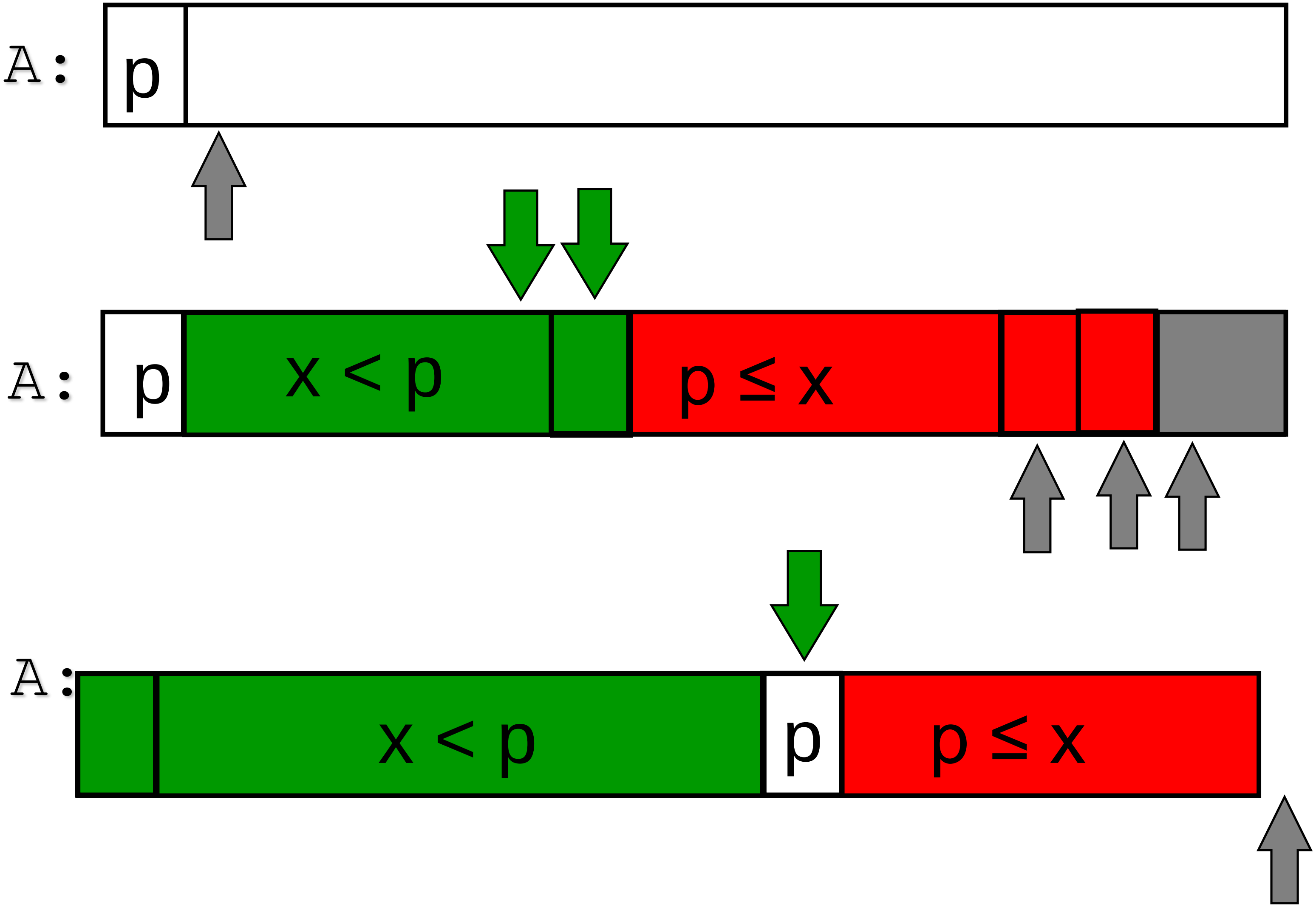
p ≤ x

Sorted

Quick Sort

```
Quick-Sort(A, left, right)
  if    left ≥ right  return
  else
        middle ← Partition(A, left, right)
        Quick-Sort(A, left, middle−1 )
        Quick-Sort(A, middle+1, right)
  end if
```

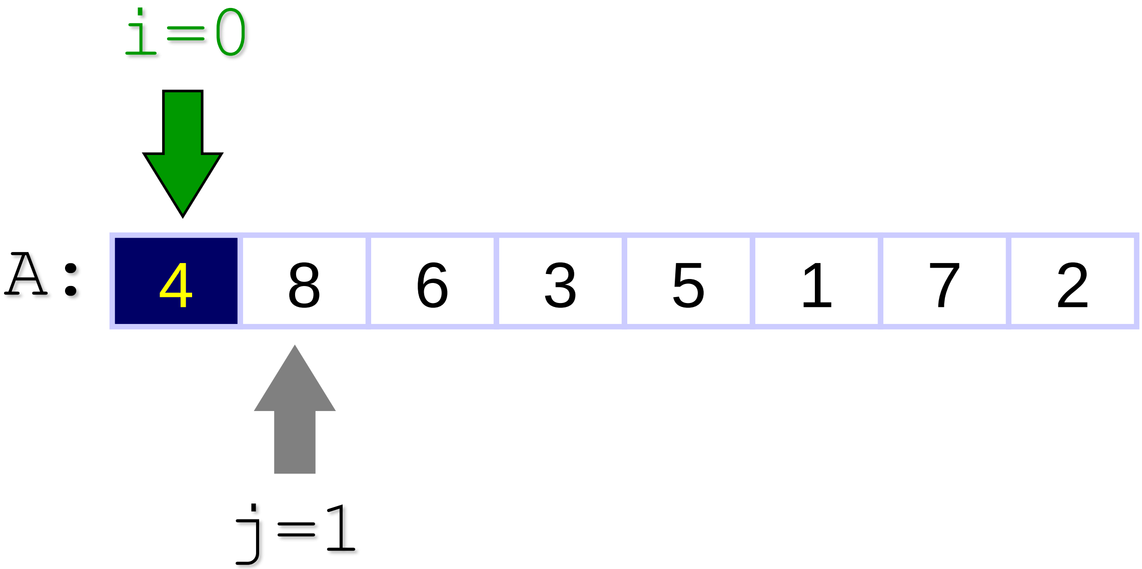
Partition



Partition Example (교과서의 예)

A:	4	8	6	3	5	1	7	2

Partition Example



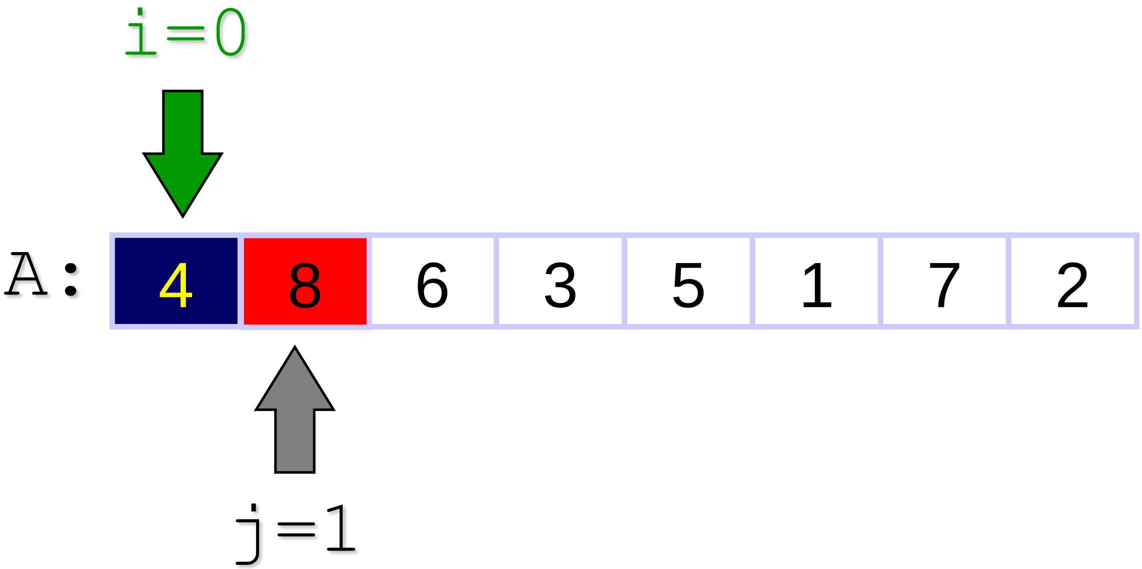
i=0

A:

4	8	6	3	5	1	7	2

j=1

Partition Example

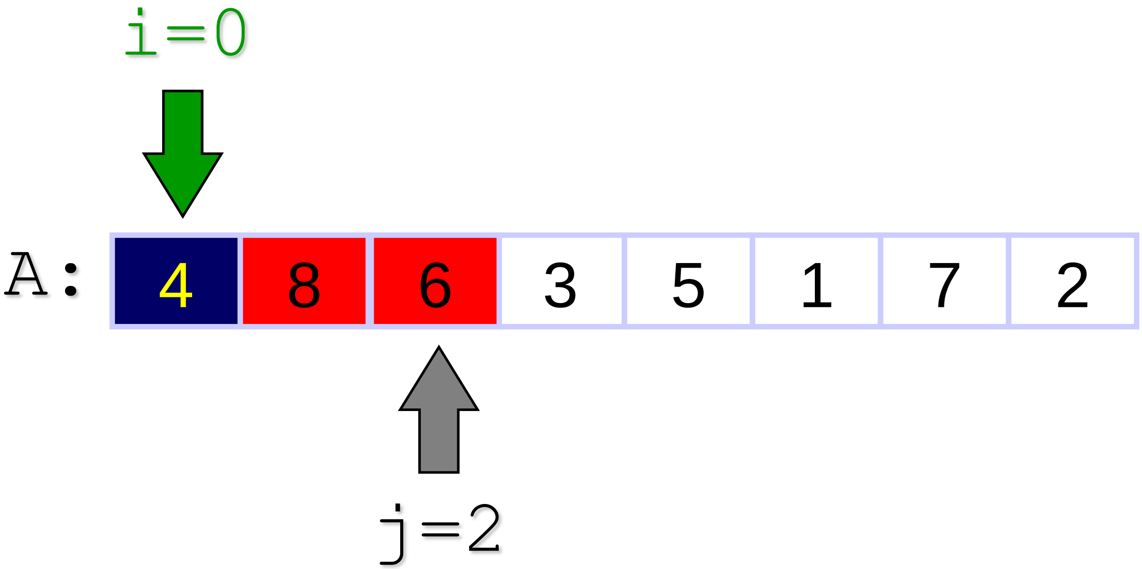


i=0

A:	4	8	6	3	5	1	7	2

j=1

Partition Example

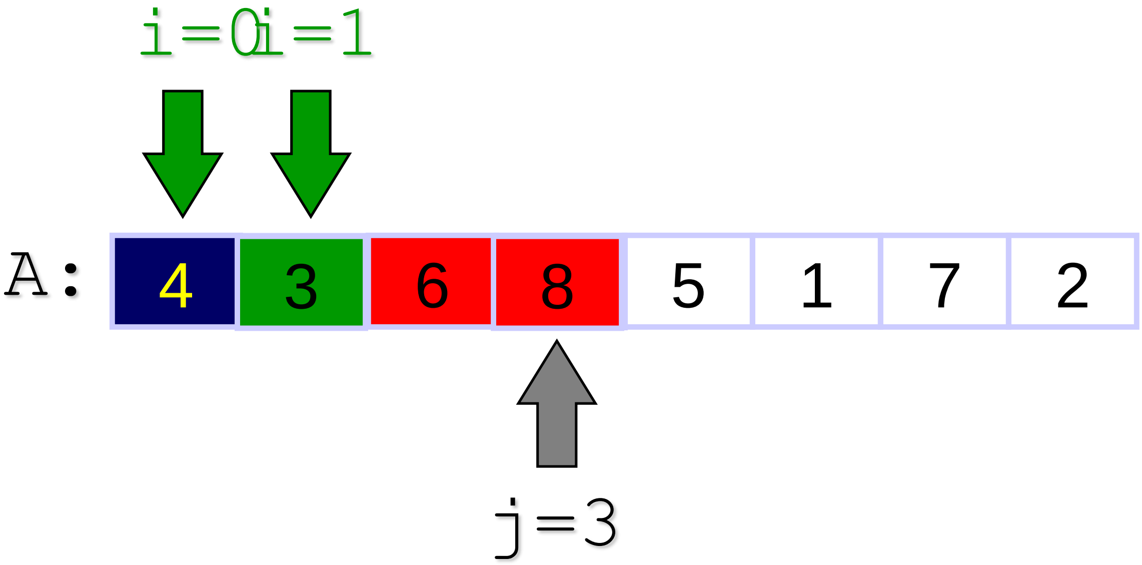


i=0

A:							
4	8	6	3	5	1	7	2

j=2

Partition Example



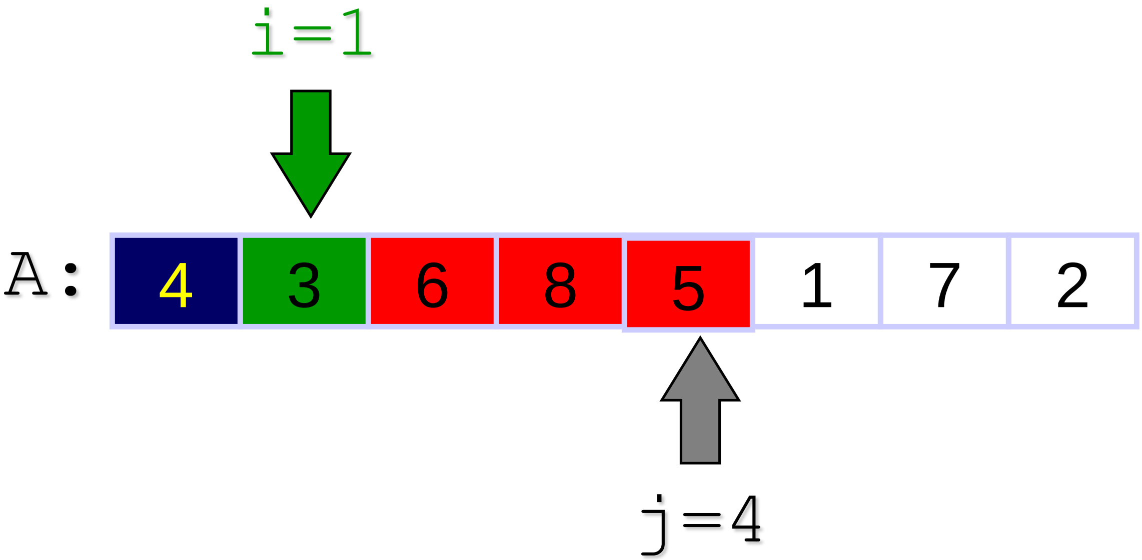
i=0 i=1

A:

4	3	6	8	5	1	7	2

j=3

Partition Example

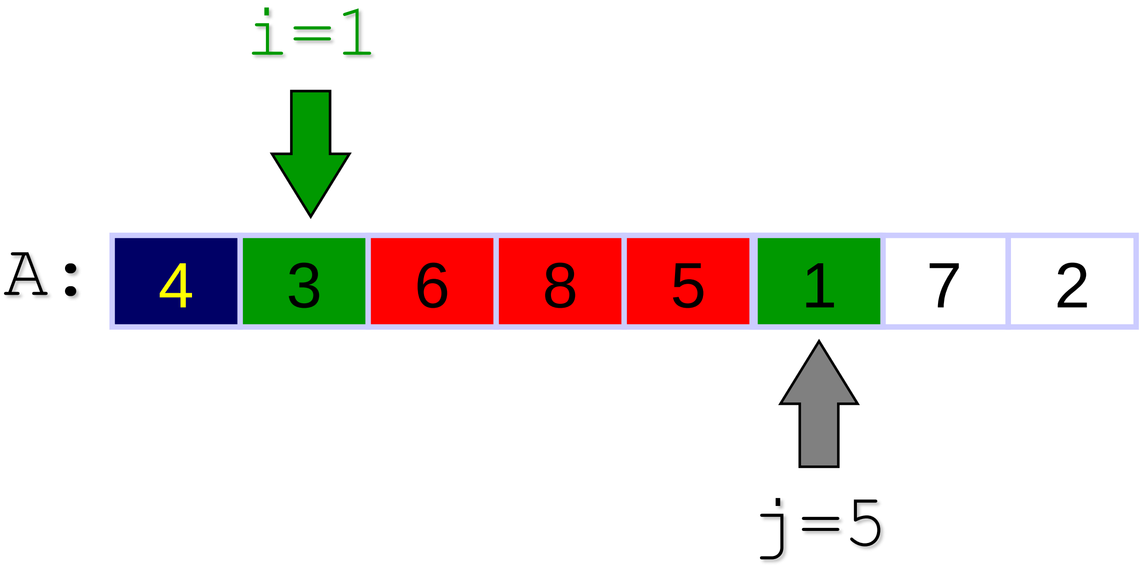


i=1

A: 4 3 6 8 5 1 7 2

j=4

Partition Example



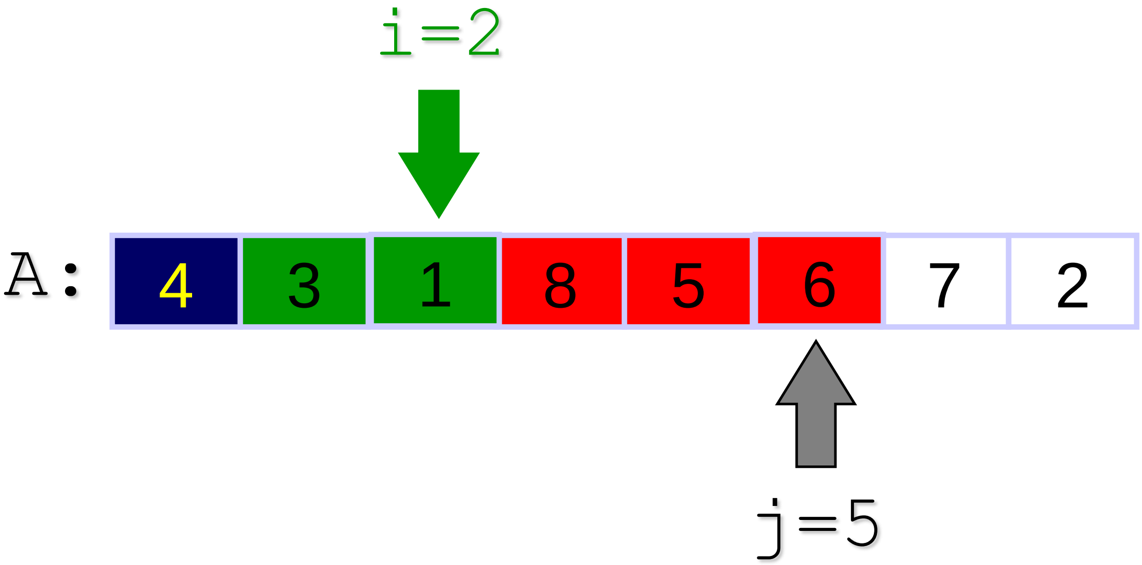
i=1

A:

4	3	6	8	5	1	7	2

j=5

Partition Example

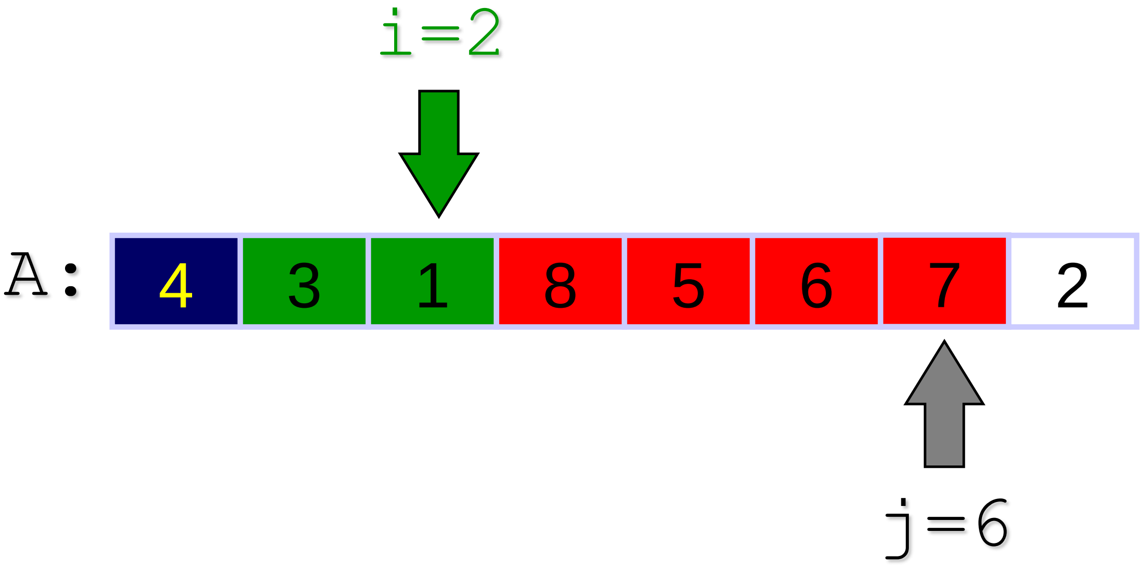


i=2

A:							
4	3	1	8	5	6	7	2

j=5

Partition Example



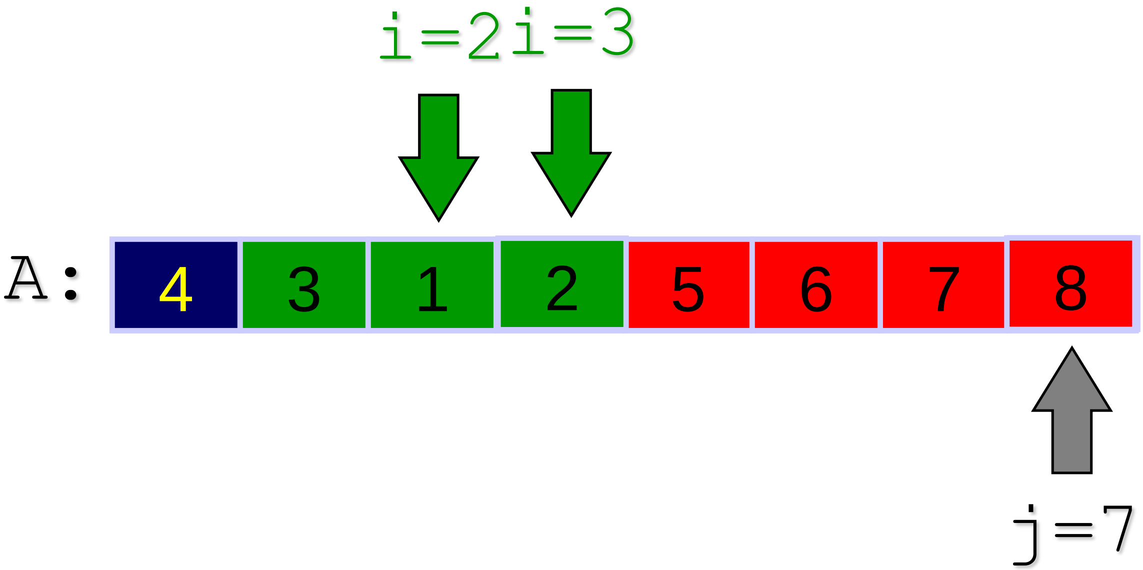
i=2

A:

4	3	1	8	5	6	7	2

j=6

Partition Example

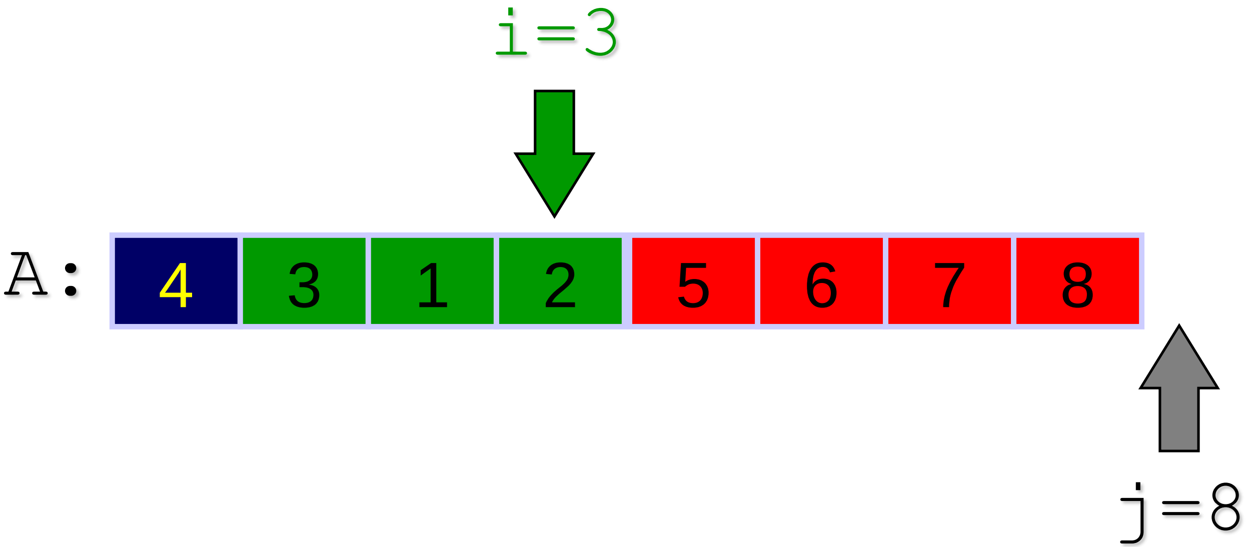


i=2 i=3

A: 4 3 1 2 5 6 7 8

j=7

Partition Example



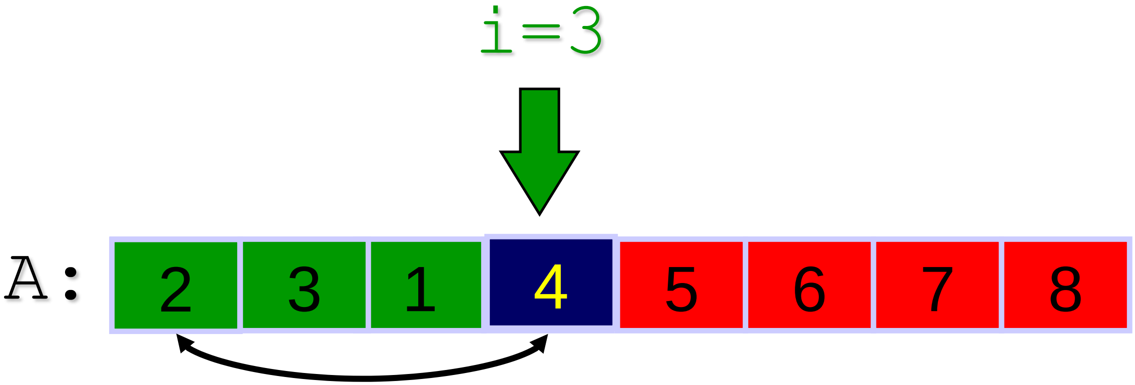
i=3

A:

4	3	1	2	5	6	7	8

j=8

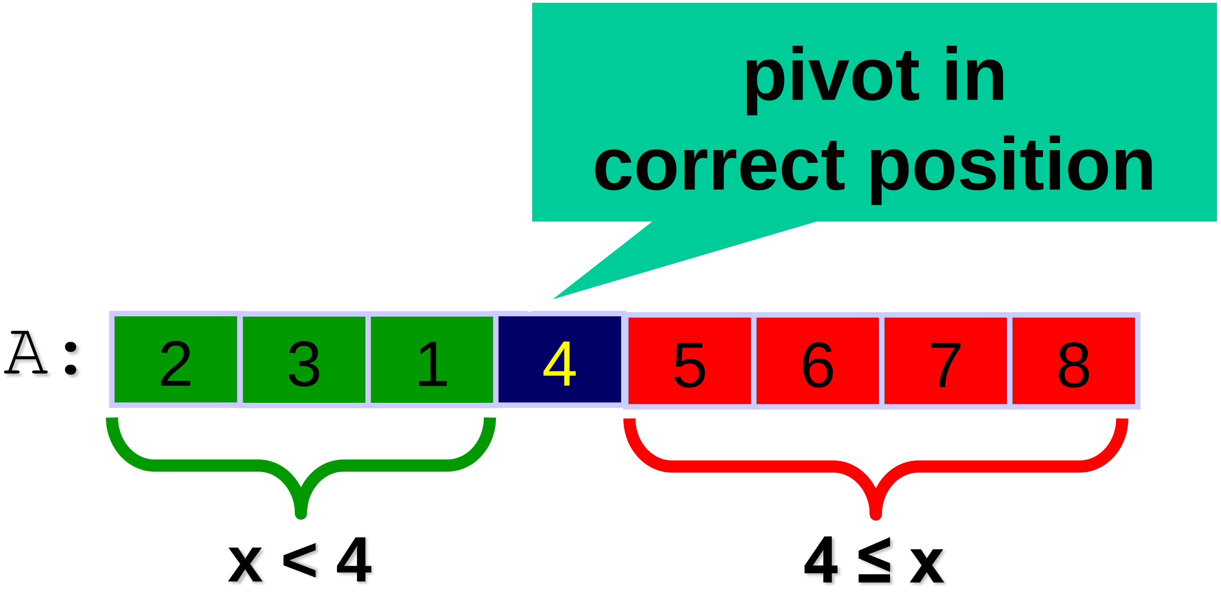
Partition Example



i=3

A:	2	3	1	4	5	6	7	8

Partition Example



pivot in correct position

A:

2	3	1	4	5	6	7	8

$x < 4$

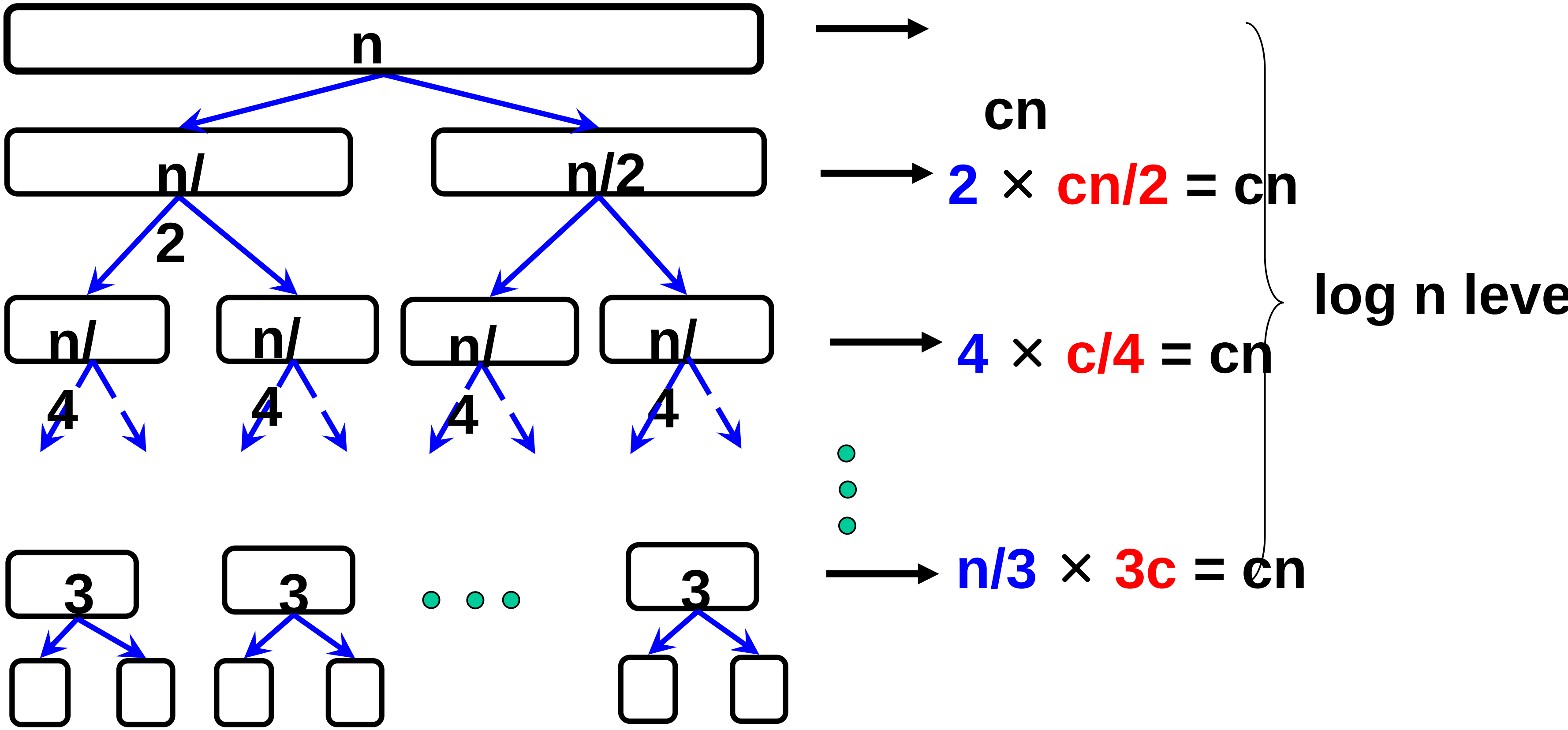
$4 \leq x$

```
Partition(A, left, right)
    x ← A[left]
    i ← left
    for j ← left+1 to right
        if A[j] < x then
            i ← i + 1
            swap(A[i], A[j])
        end if
    end for j
    swap(A[i], A[left])
    return i
```

n = right – left +1
Time: cn for some constant c
Space: constant

Quick-Sort: Best Case

- Even Partition



cn

2 × cn/2 = cn

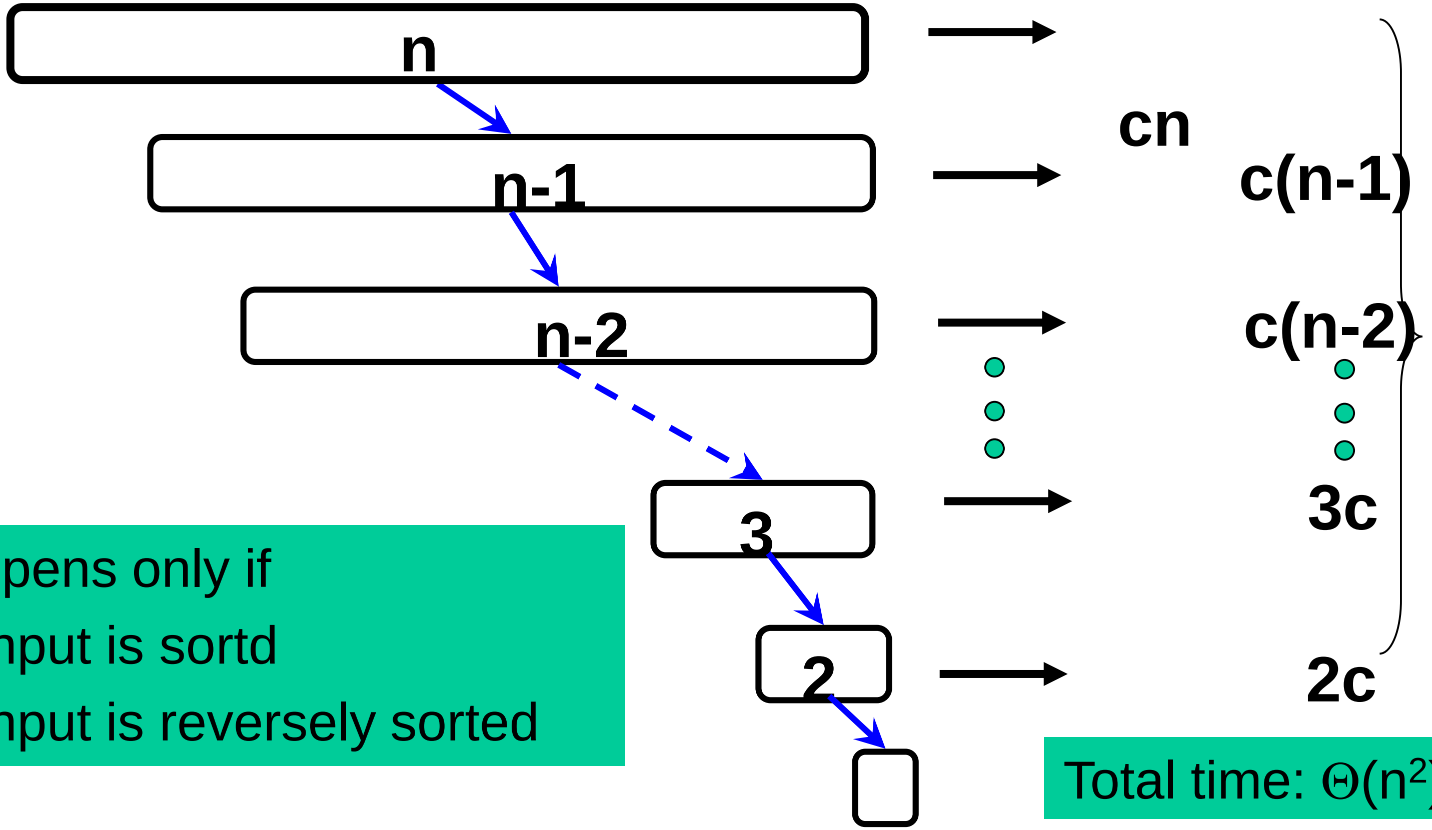
4 × c/4 = cn

n/3 × 3c = cn

Total time: Θ(nlogn)

Quick-Sort: Worst Case

- Unbalanced Partition



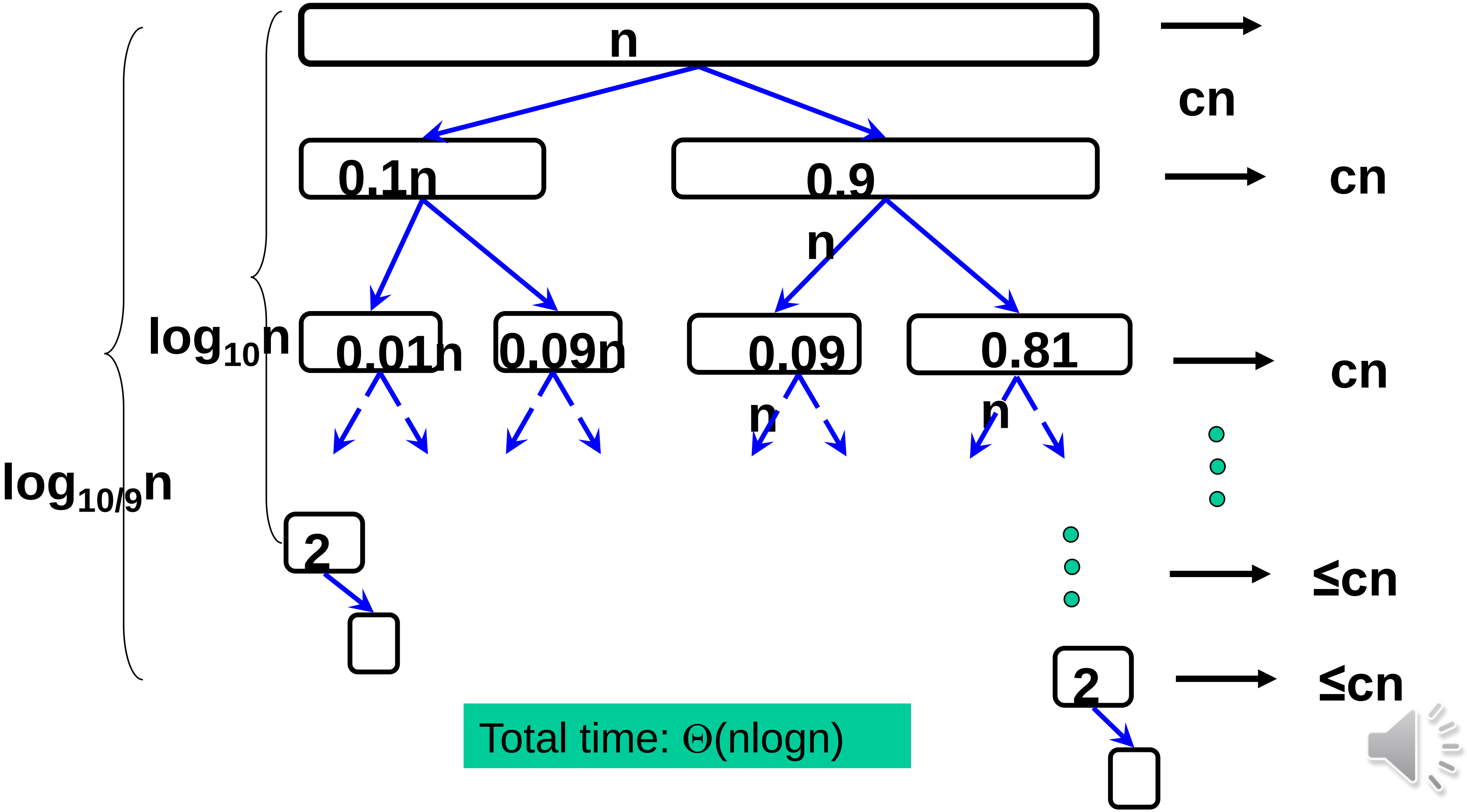
Happens only if

- input is sortd
- input is reversely sorted

Total time: $\Theta(n^2)$

Quick-Sort: an Average Case

- Suppose the split is 1/10 : 9/10



Total time: Θ(nlogn)

Quick-Sort Summary

- Time
  - – Most of the work done in partitioning.
  - – Average case takes **Θ(n log(n))** time.
  - – Worst case takes **$\Theta(n^2)$** time

Space

- – Sorts in-place, i.e., does not require additional space

Summary

- Divide and Conquer
- Merge-Sort
  - Most of the work done in Merging
  - **$\Theta(n \log(n))$** time
  - **$\Theta(n)$ space**
- Quick-Sort
  - Most of the work done in partitioning
  - Average case takes **$\Theta(n \log(n))$** time
  - Worst case takes **$\Theta(n^2)$** time
  - **$\Theta(1)$ space**

퀵 정렬 (5)

- partition 알고리즘

```
partiton(a[], i, j) //=> public static int partition(int[] a, int i, int j){
   //middle ← (i + j) / 2;     // middle은 a[]의 중앙 인덱스 값
  // pivot ← a[middle];     // a[]의 중앙 원소값을 pivot으로 설정
   // a[middle] ← a[i];     // a[i]와 a[middle]을 서로 교환
   // a[i] ← pivot;     //a[i]는 pivot 값을 보관  제일왼쪽
   pivot=a[i];  //ppt의 알고리즘
   p ← i;     // p는 두 파티션의 경계를 지시하는 인덱스
   for (k ←i+1;  k ≤ j;  k ← k+1) do  {
                    // a[i]를 제외한 a[i+1 : j]에 있는 모든 원소 a[k]들을 검사하여
      if (a[k] < pivot) then  {     // a[k]가 pivot보다 작으면
         p ← p+1;                    // p를 1 증가시켜 a[k]를 p 인덱스 범위
         temp ← a[p];               // 안으로 포함되게 함
         a[p] ← a[k];
         a[k] ← temp;
      }
   }
   temp ← a[i];     // a[i]와 a[p]를 교환
   a[i] ← a[p];
   a[p] ← temp;
   return p;
end partition()
```

퀵 정렬 (9)

- Sorting 클래스의 메소드 멤버 구현

```
public static void quickSort(int[] a)  {
// 퀵 정렬의 메인 메소드
    internalQuickSort(a, 0, a.length-1);
}

private static void internalQuickSort(int[] a, int m, int n)  {
// quickSort()의 보조 메소드
    int p;
    if (m > n) then return;
    p = partition(a, m, n);  //한 피봇이 제 위치를 찾음
    internalQuickSort(a, m, p-1);
    internalQuickSort(a, p+1, n);
}

private static int partition(int[] a, int m, int n)  {
// internalQuicksort()의 보조 메소드
    • • • • •      // partition 알고리즘의 Java 코드
    return p;
}
```

퀵 정렬 (4)

- SortMain 클래스 (샘플 프로그램)

```
public class SortMain  {
    public static void main(String[] args)  {
        int[] a = { 5, 2, 8, 3, 1};
        System.out.println(“정렬전 배열 원소 : ”);
        int i;
        for (i = 0; i < a.length; i++)
             System.out.print(a[i] + "  ");
        System.out.println();
        Sorting. quickSort(a);     // 원하는 정렬 메소드를 선택
       // Sorting a1 = new Sorting();   //   ?
        //a1.selectionSort(a);    //?
        System.out.println(“정렬된 배열 원소 : ”);
        for (i = 0; i < a.length; i++)
             System.out.print(a[i] + " ");
        System.out.println();
    }
}
```

<실행결과>

```
정렬전 배열 원소 :
5   2   8   3   1
정렬된 배열 원소 :
1   2   3   5   8
```

Mergesort(합병정렬)

Divide and Conquer

- **Divide and Conquer cuts the problem in half each time, but uses the result of both halves:**
  - **cut the problem in half until the problem is trivial**
  - **solve for both halves**
  - **combine the solutions**

Mergesort

- A divide-and-conquer algorithm:
- Divide the unsorted array into 2 halves until the sub-arrays only contain one element
- Merge the sub-problem solutions together:
    - Compare the sub-array's first elements
    - Remove the smallest element and put it into the result array
    - Continue the process until all elements have been put into the result array

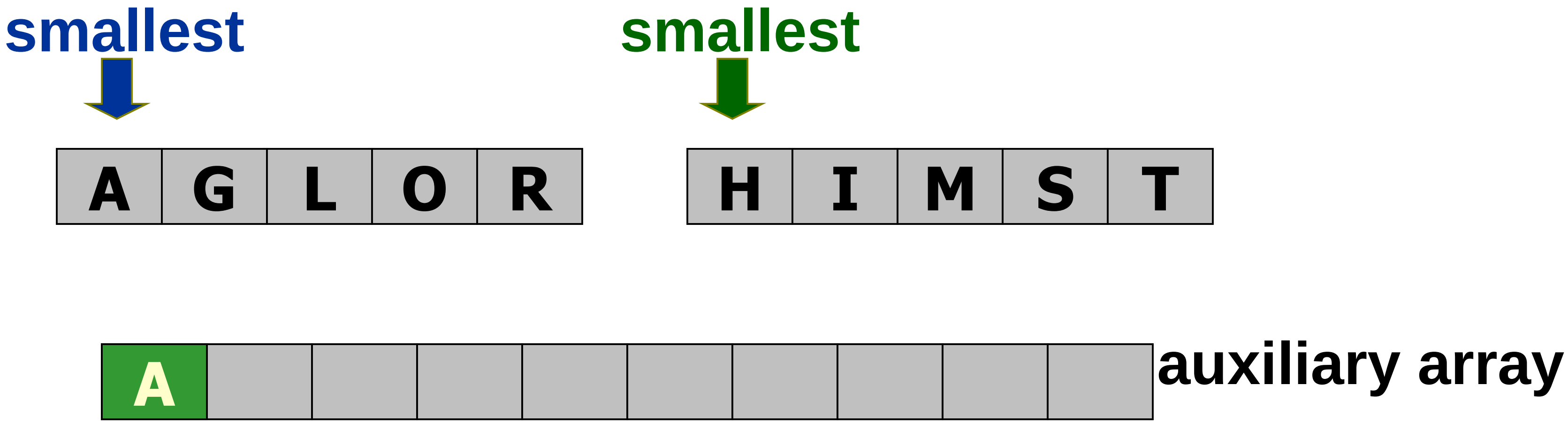
37	23	6	89	15	12	2	19

Algorithm

```
Mergesort(Passed an array)
   if array size > 1
      Divide array in half
      Call Mergesort on first half.
      Call Mergesort on second half.
      Merge two halves.

Merge(Passed two arrays)
   Compare leading element in each array
   Select lower and place in new array.
      (If one input array is empty then place
       remainder of other array in output array)
```

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



smallest

smallest

A	G	L	O	R

H	I	M	S	T

A									

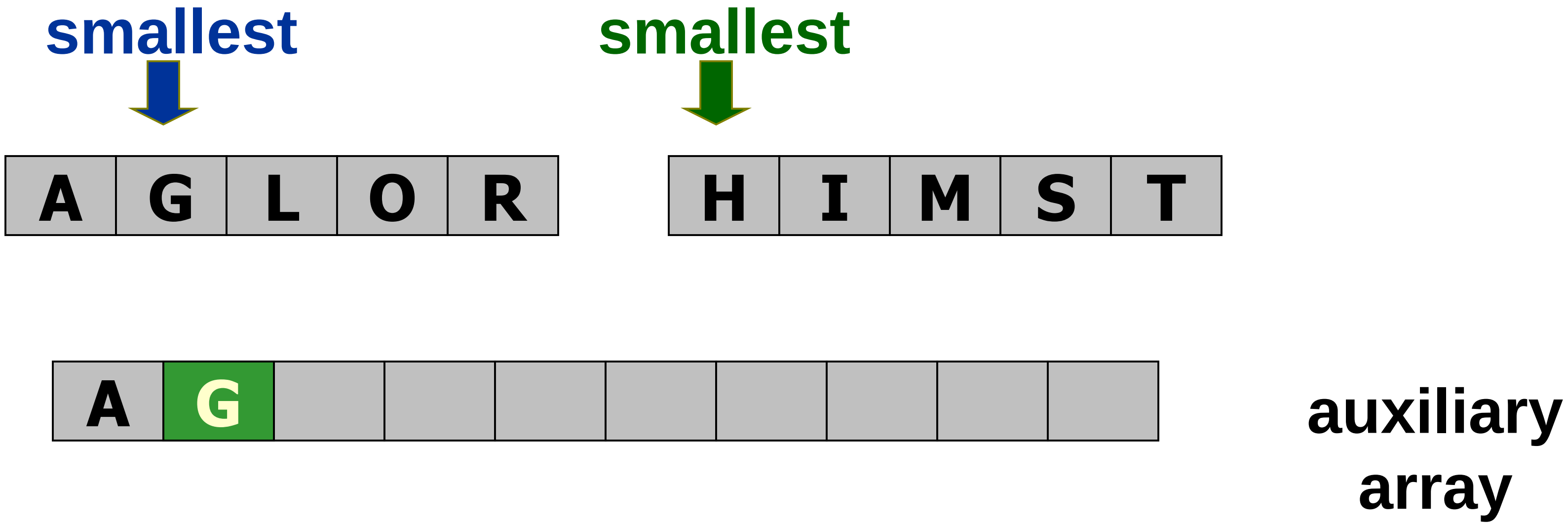
auxiliary array

Not in-place sorting algorithm
제자리정렬 알고리즘아님

원소의 개수에
비례하는추가공간필요

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



smallest smallest

A	G	L	O	R

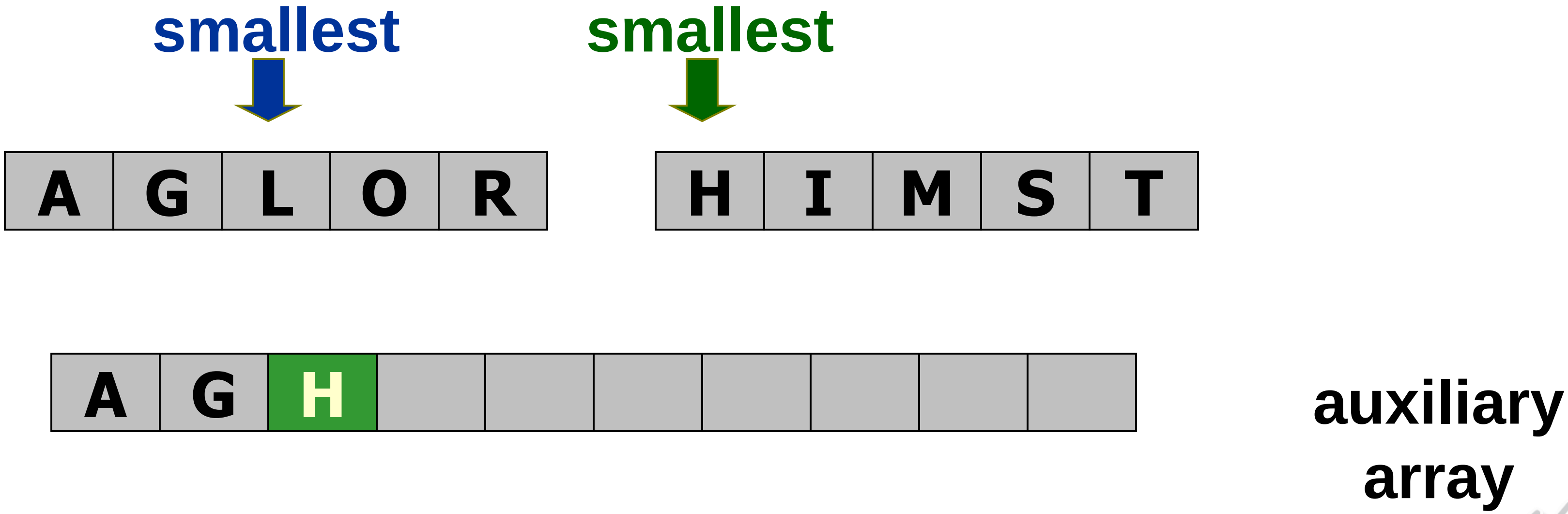
H	I	M	S	T

A	G								

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



smallest

smallest

A	G	L	O	R

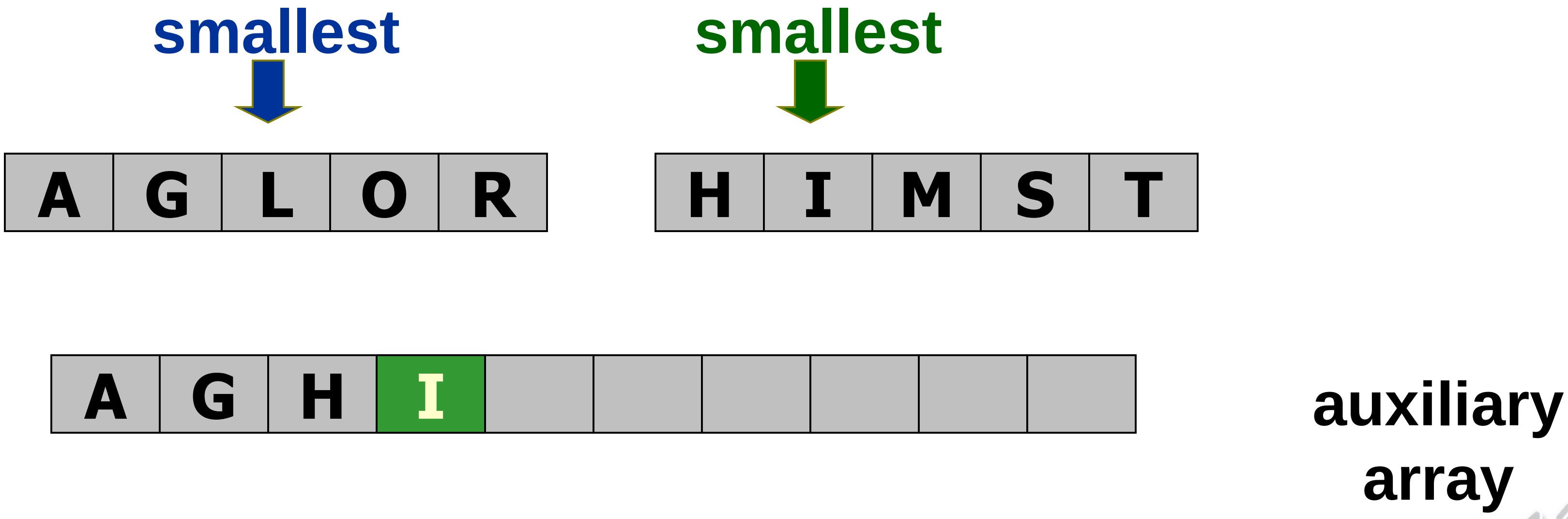
H	I	M	S	T

A	G	H							

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



smallest

smallest

A	G	L	O	R

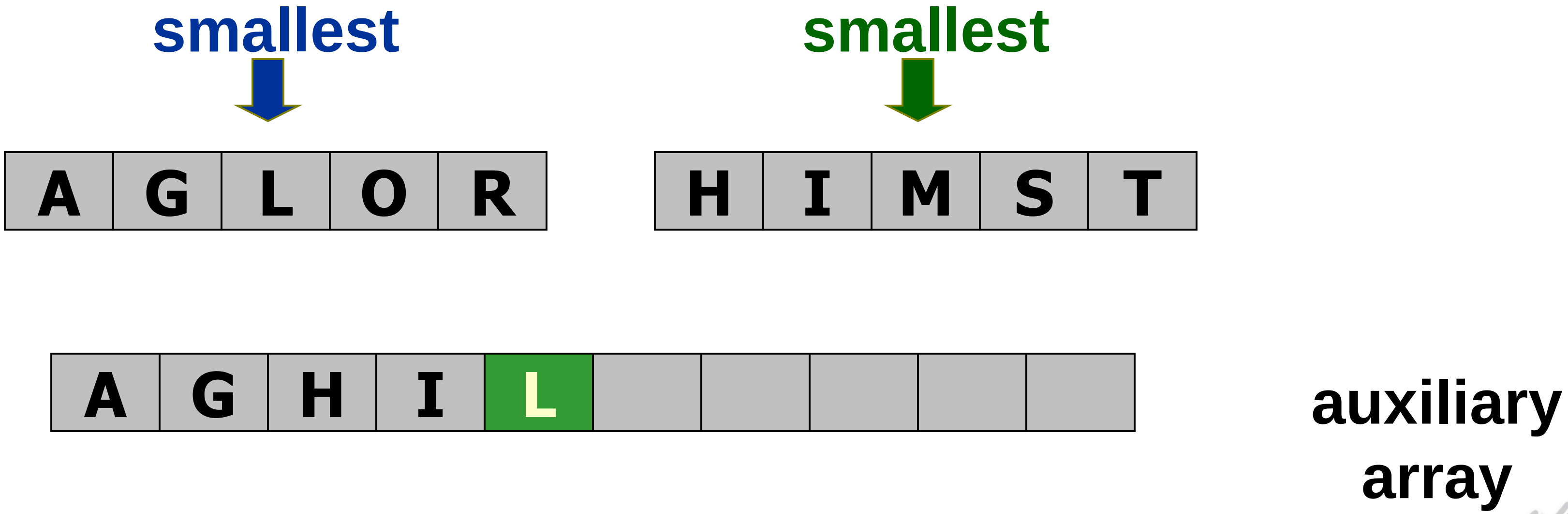
H	I	M	S	T

A	G	H	I						

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



smallest

smallest

A	G	L	O	R

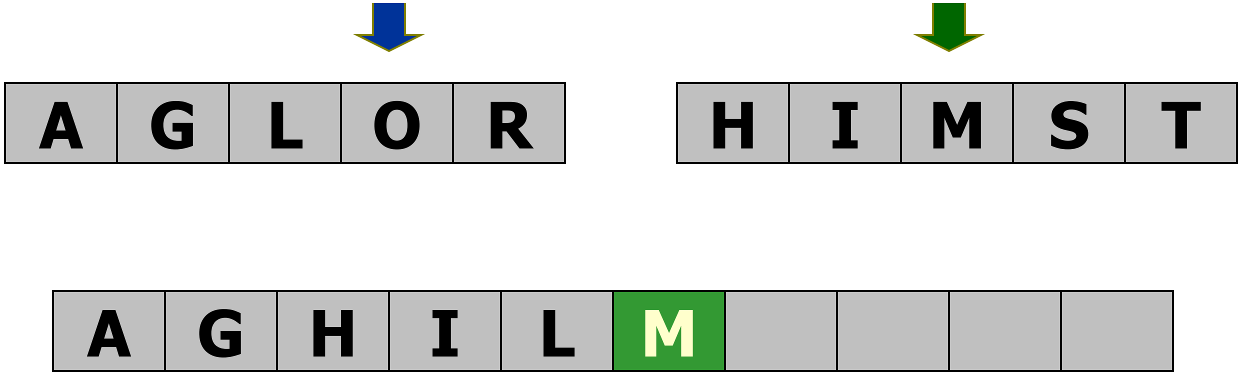
H	I	M	S	T

A	G	H	I	L					

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



A G L O R

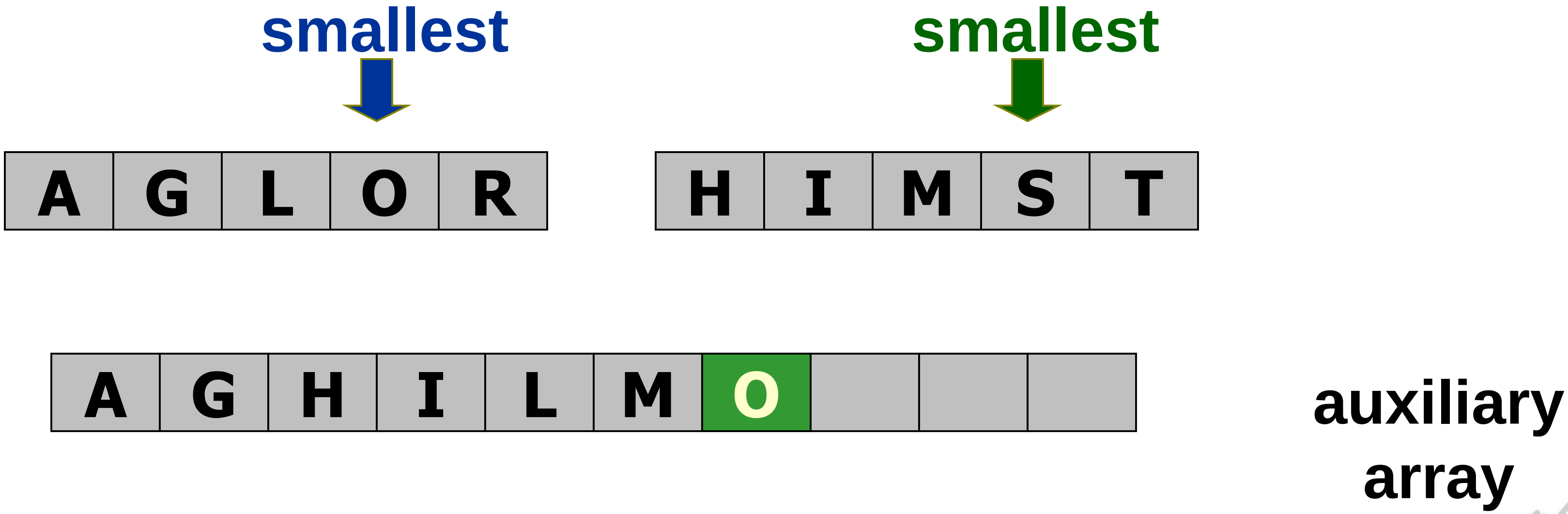
H I M S T

A G H I L M

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



smallest

smallest

A	G	L	O	R

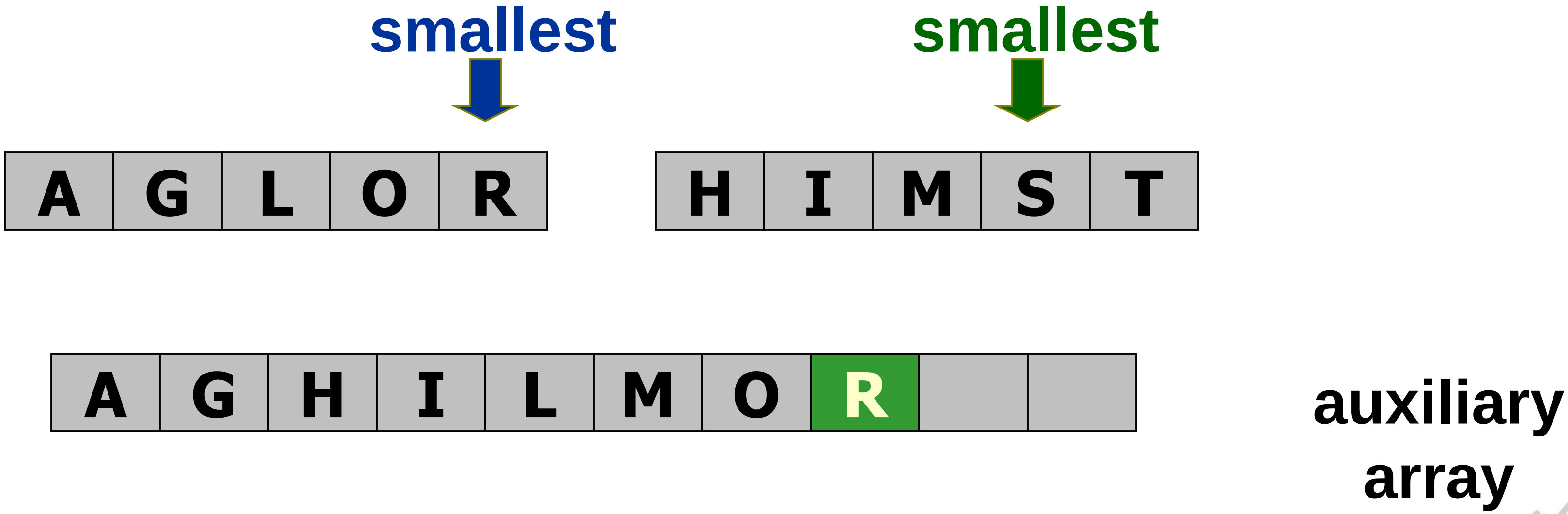
H	I	M	S	T

A	G	H	I	L	M	O			

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



A	G	L	O	R

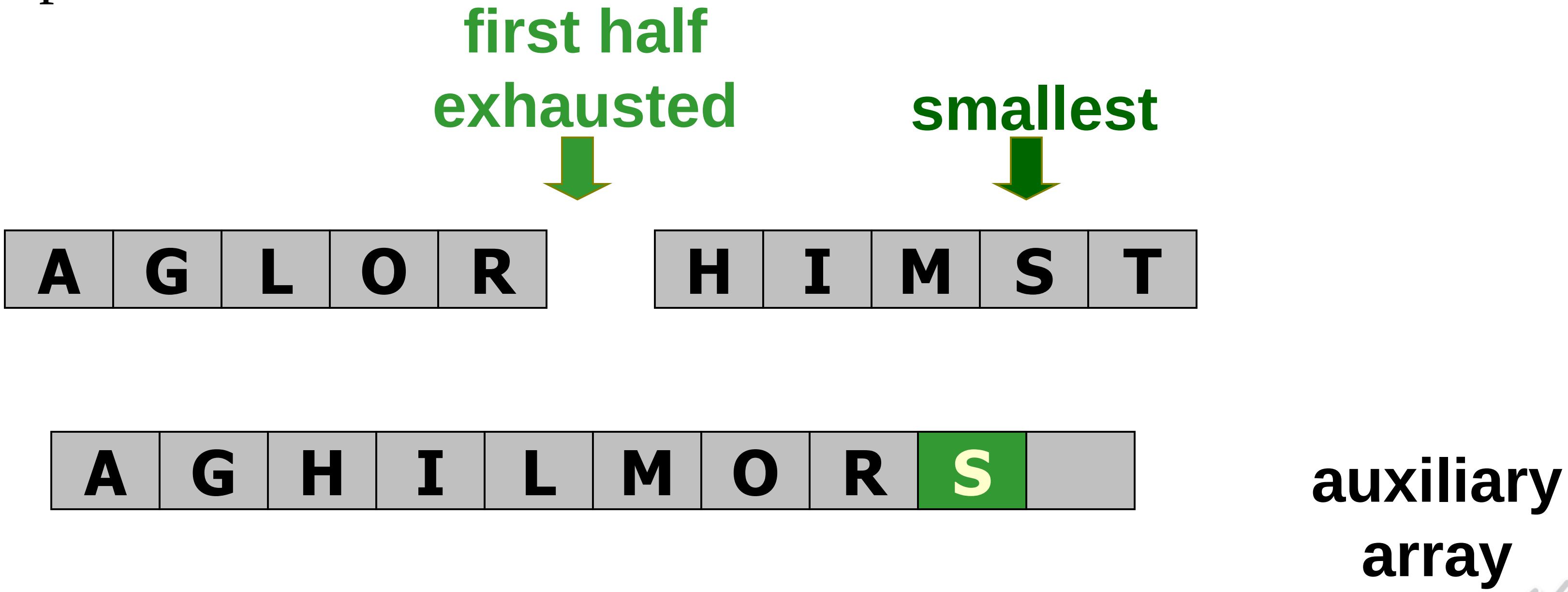
H	I	M	S	T

A	G	H	I	L	M	O	R		

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



first half exhausted

smallest

A	G	L	O	R

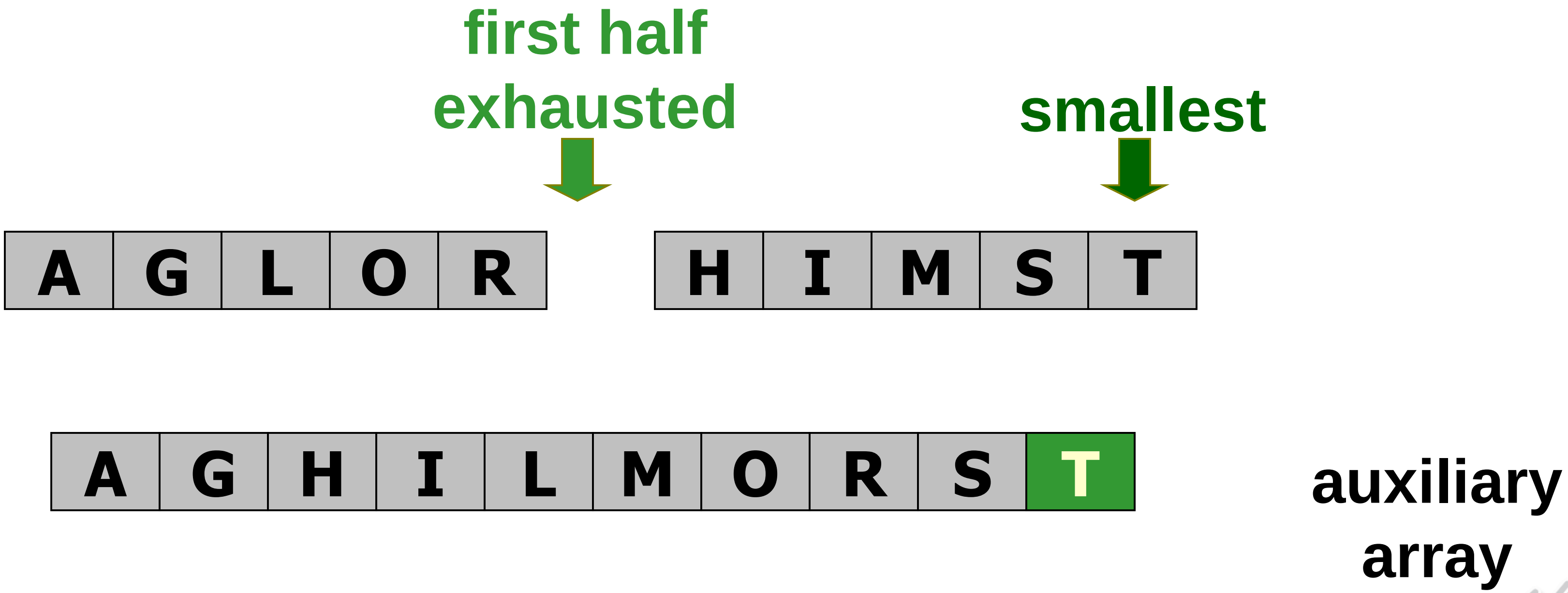
H	I	M	S	T

A	G	H	I	L	M	O	R	S	

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



first half exhausted

smallest

A	G	L	O	R

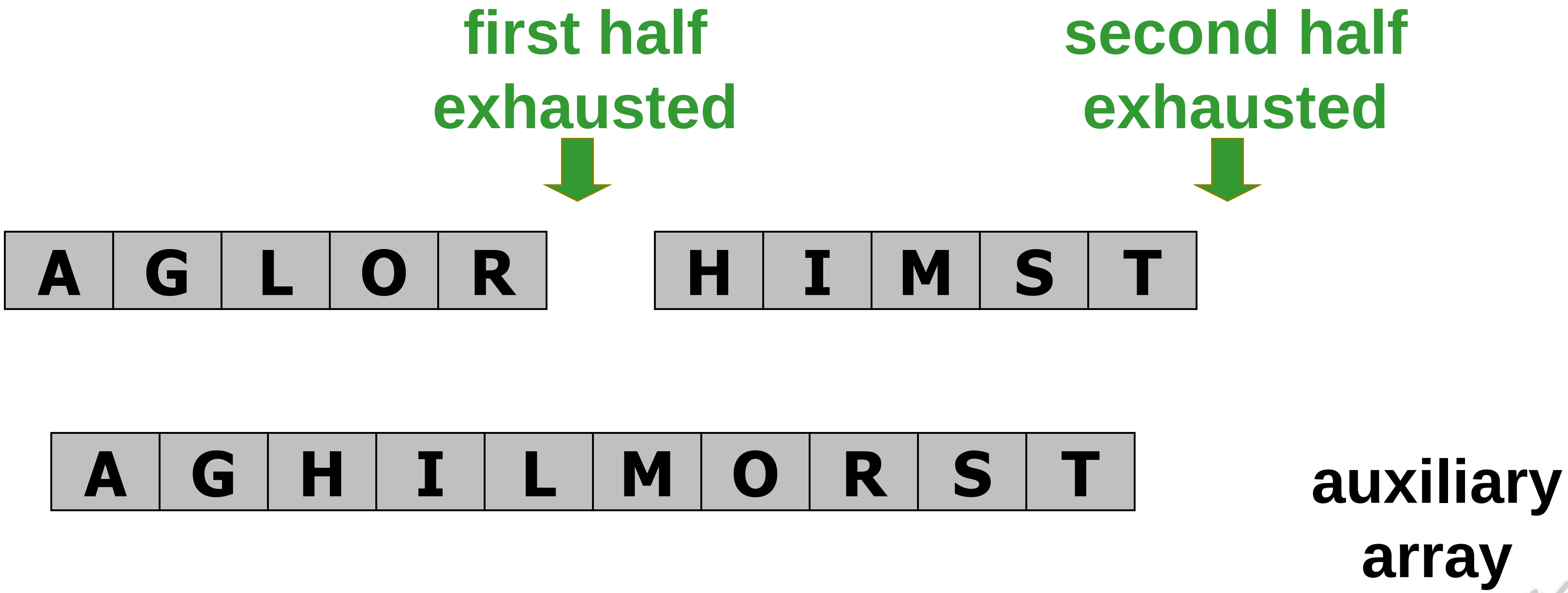
H	I	M	S	T

A	G	H	I	L	M	O	R	S	T

auxiliary array

Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



first half exhausted

second half exhausted

A	G	L	O	R

H	I	M	S	T

A	G	H	I	L	M	O	R	S	T

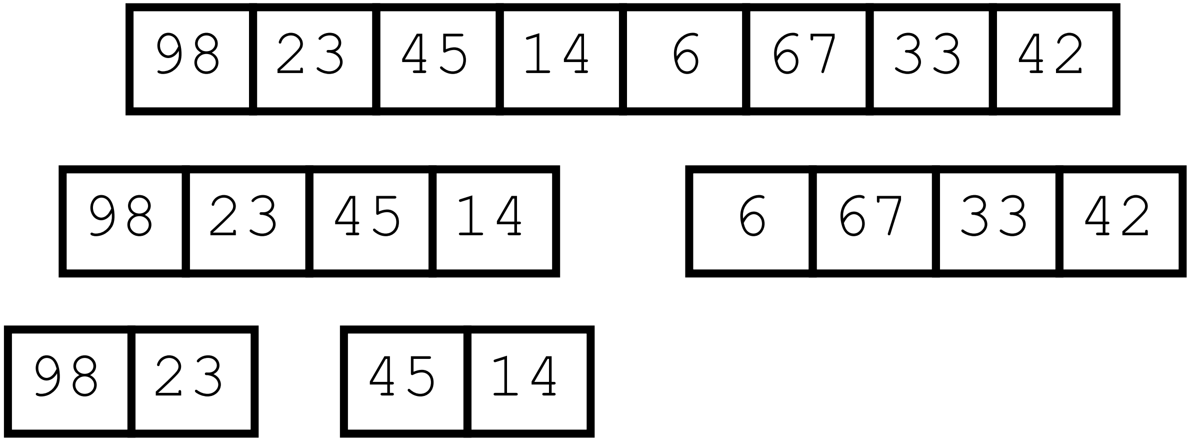
auxiliary array

98	23	45	14	6	67	33	42

98	23	45	14	6	67	33	42

98	23	45	14

6	67	33	42



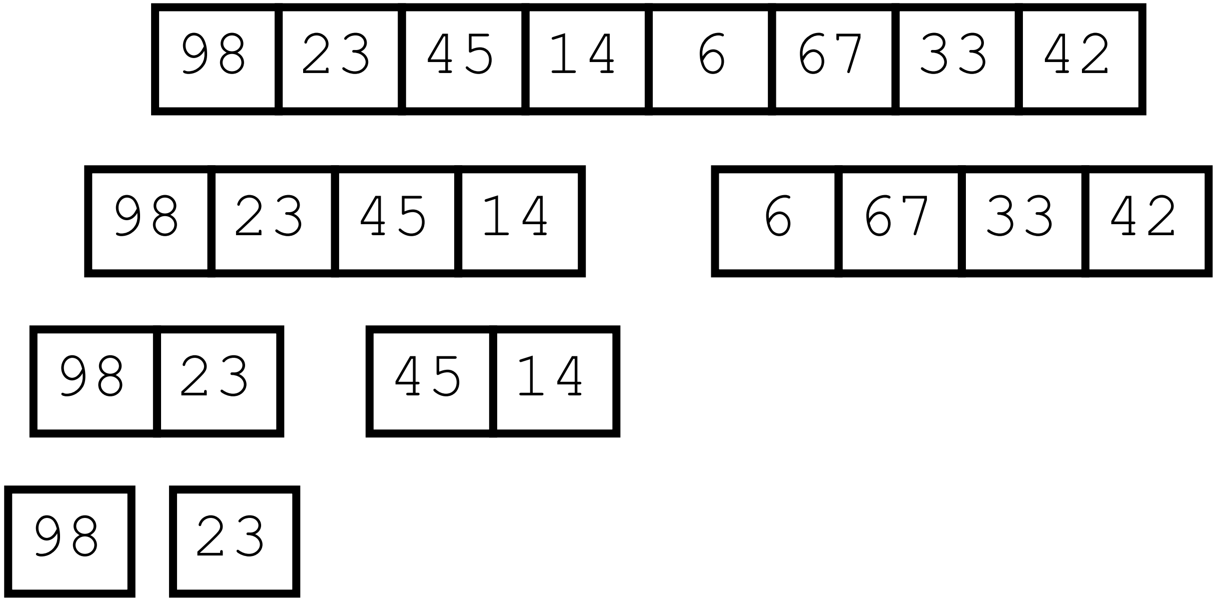
98	23	45	14	6	67	33	42

98	23	45	14

6	67	33	42

98	23

45	14

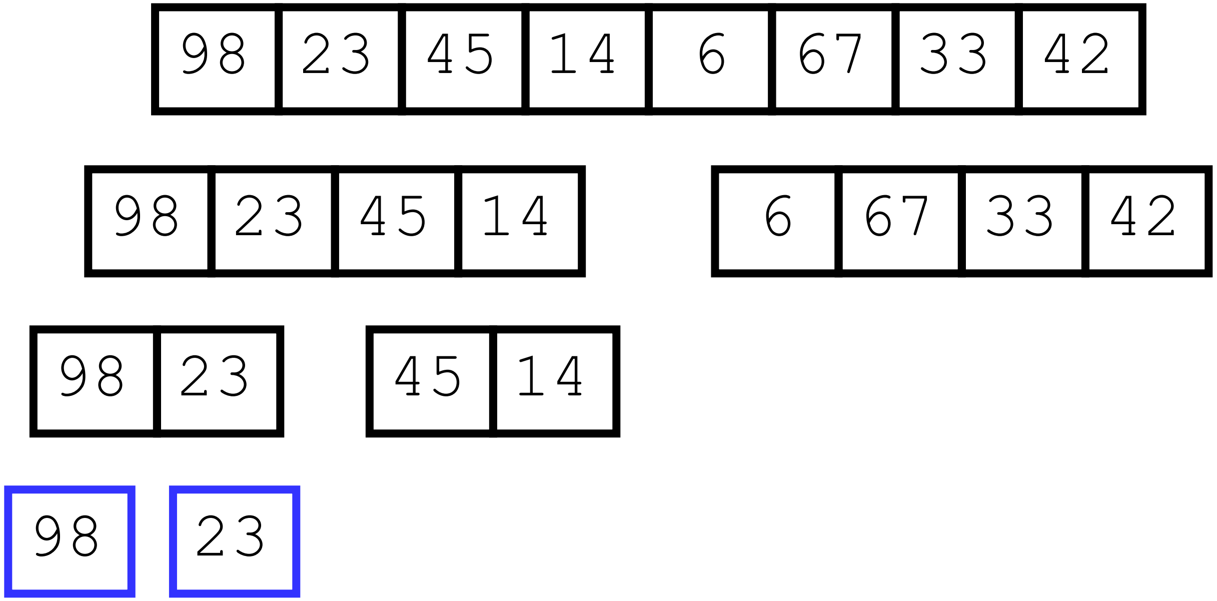


98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14

98   23



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

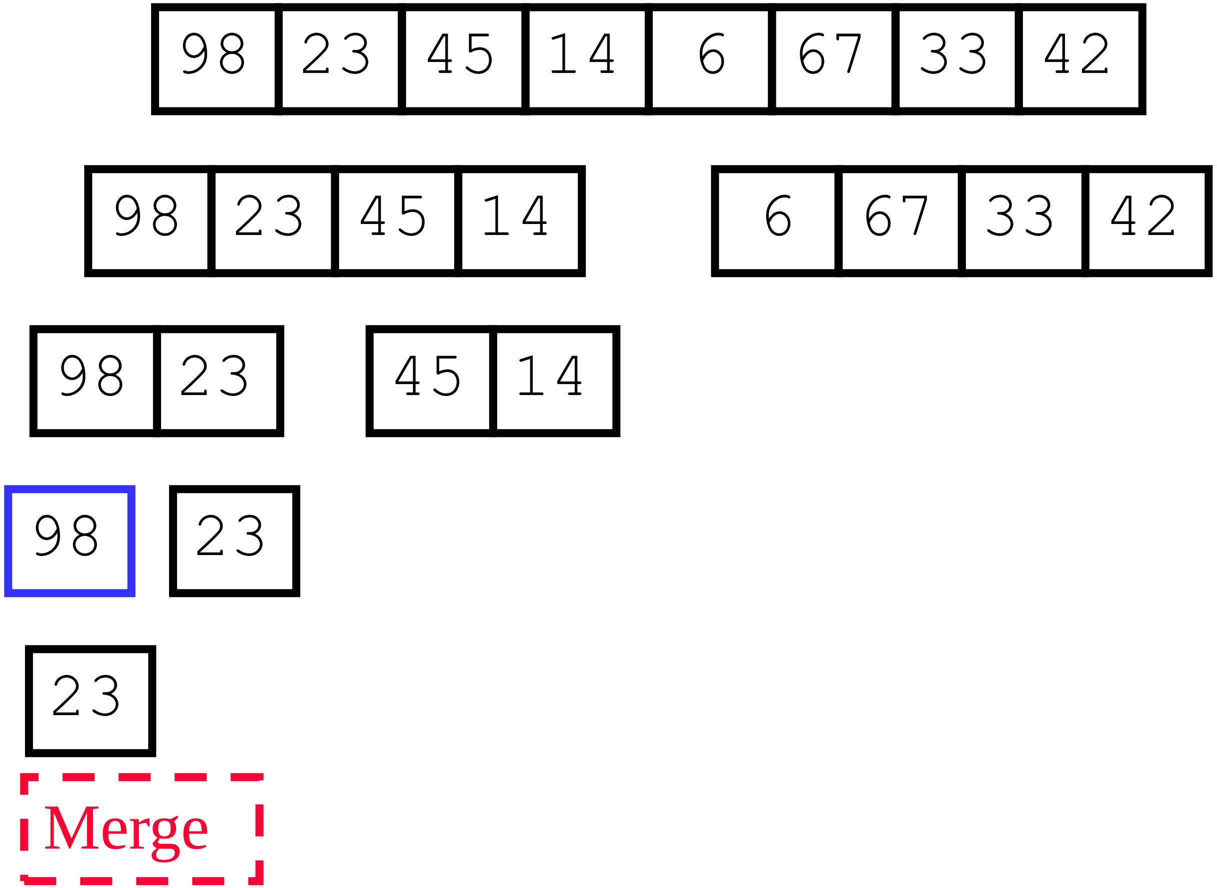
98 23

45 14

98

23

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

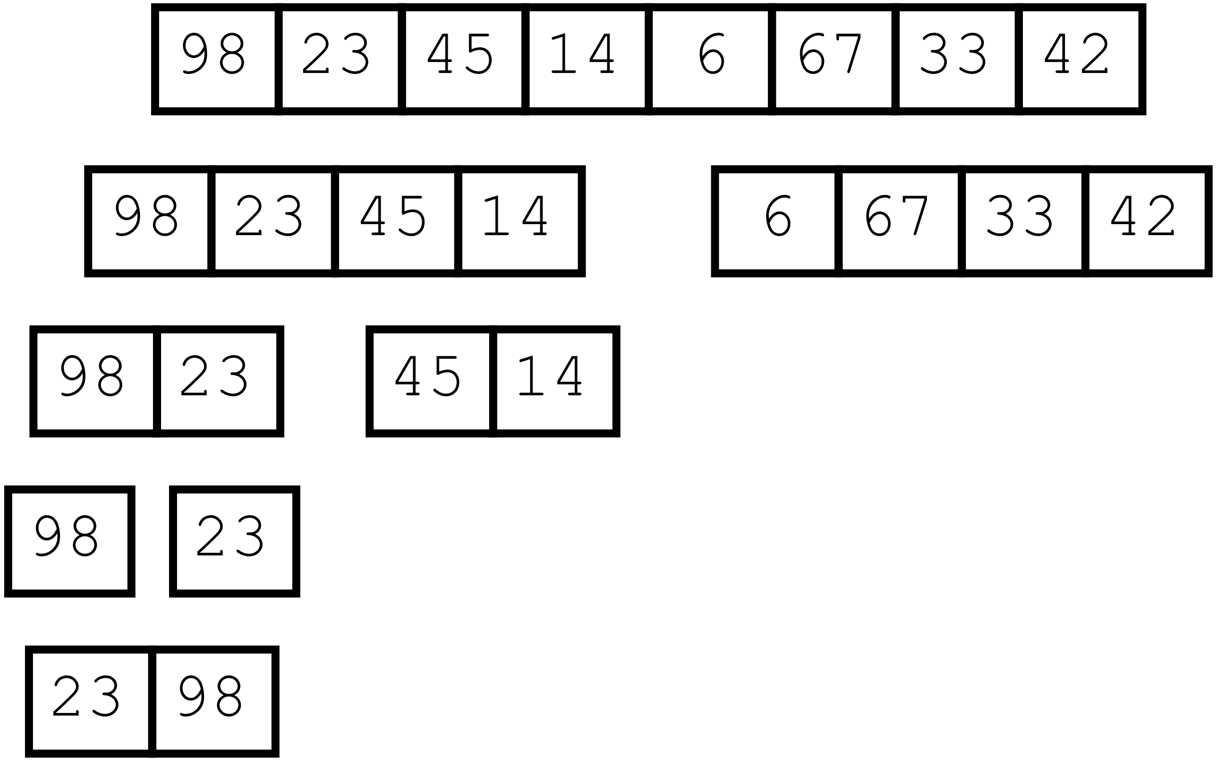
45 14

98

23

23

Merge



98 23 45 14 6 67 33 42

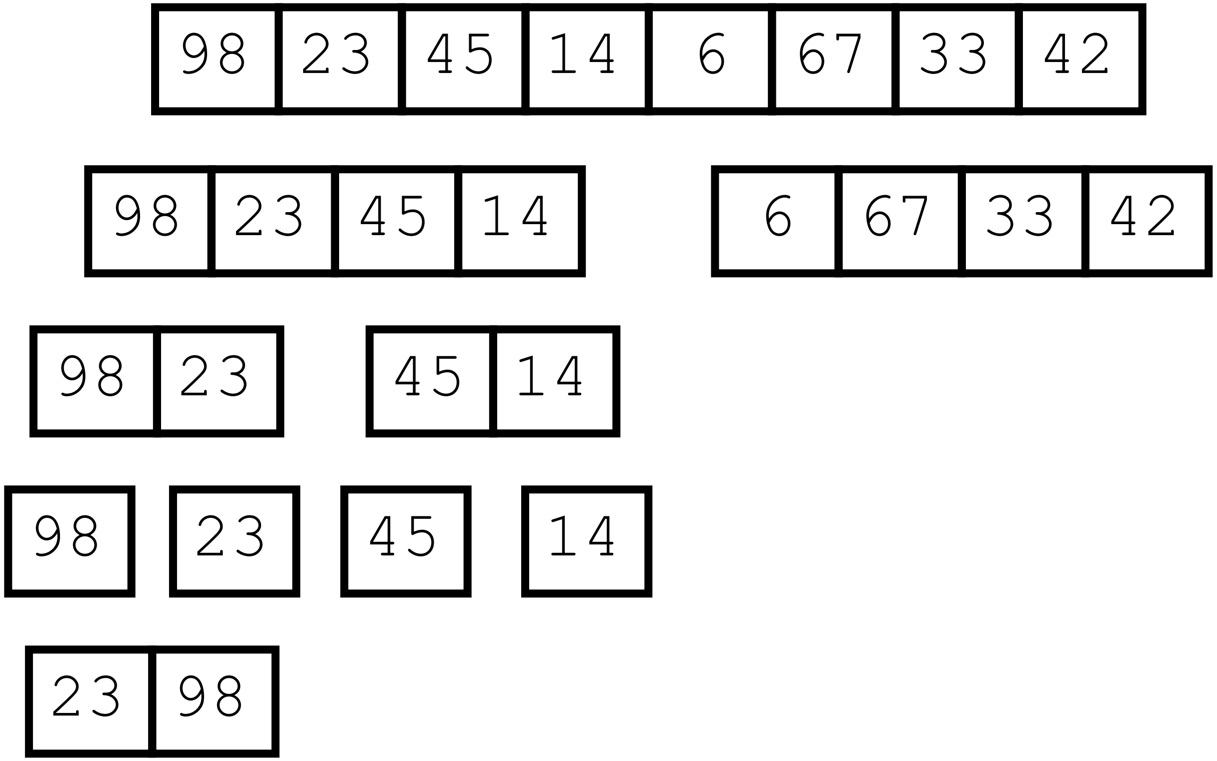
98 23 45 14 6 67 33 42

98 23 45 14

98 23

23 98

Merge



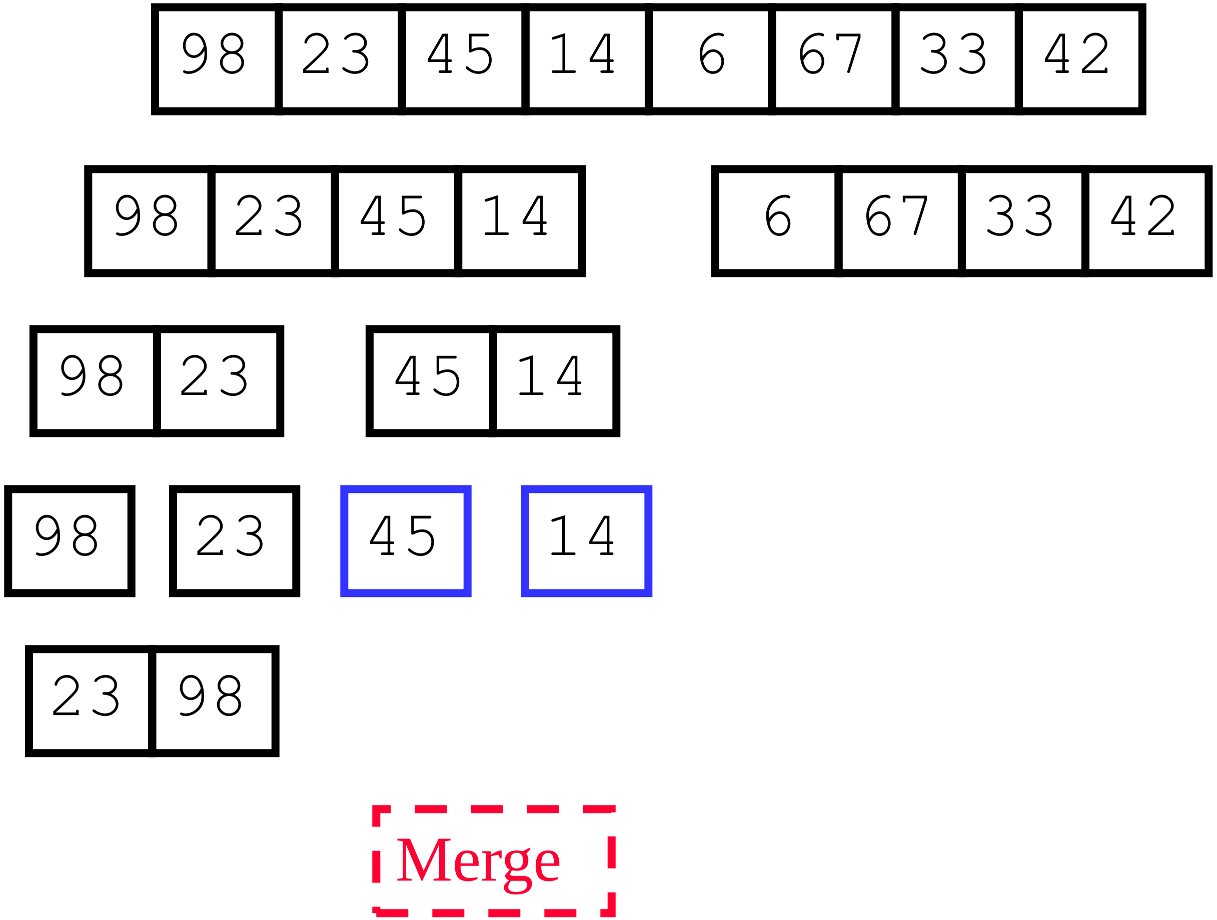
98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

98 23 45 14

98 23 45 14

23 98



98 23 45 14 6 67 33 42

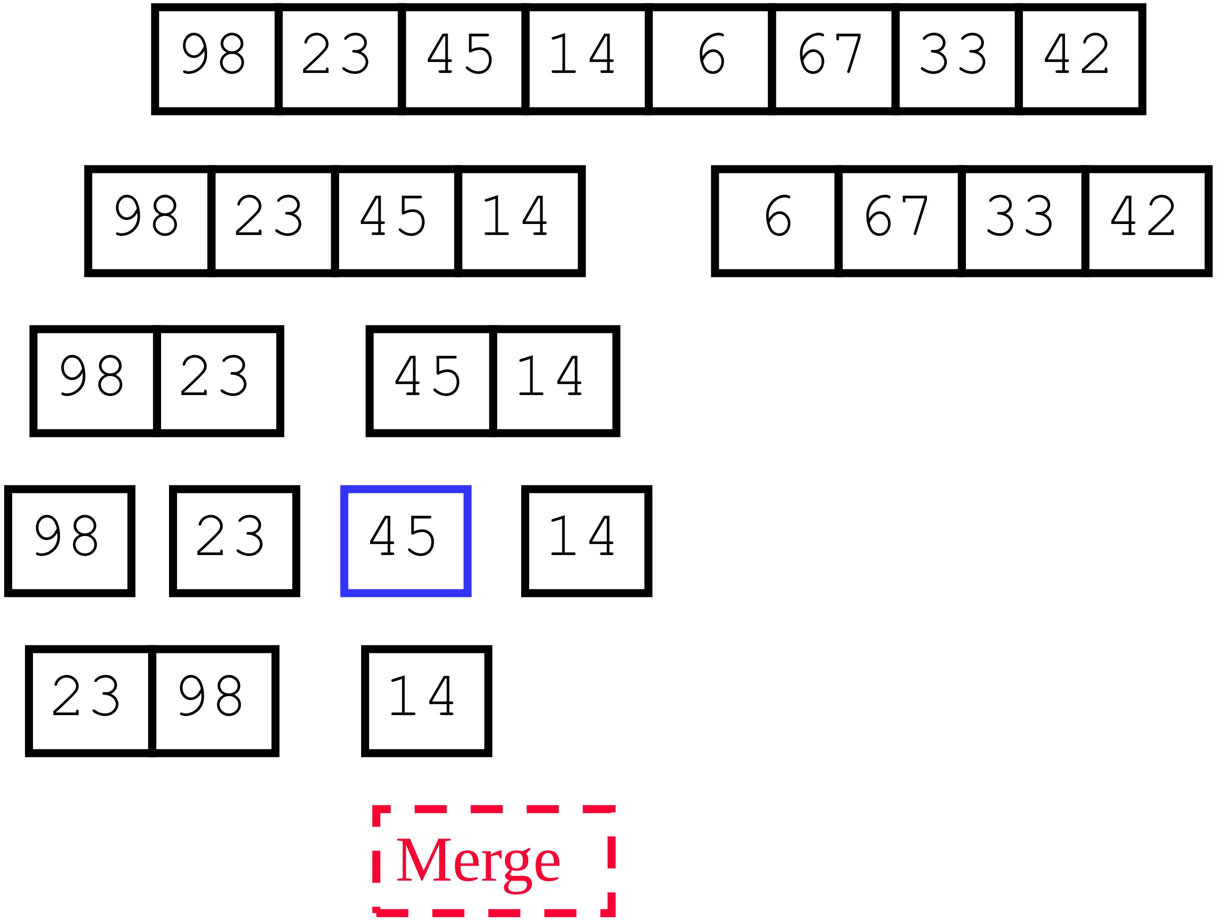
98 23 45 14   6 67 33 42

98 23   45 14

98   23   45   14

23 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

23

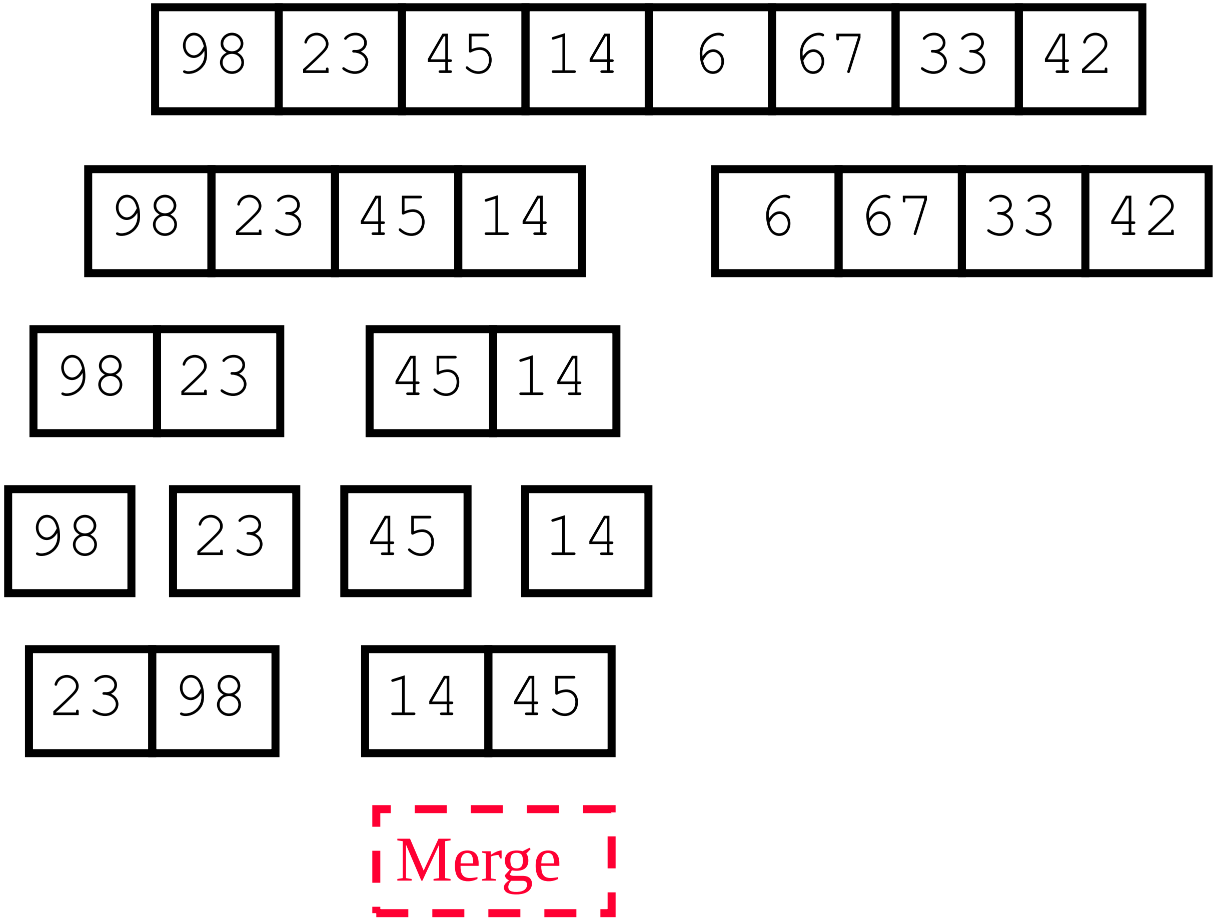
45

14

23 98

14

Merge



98 23 45 14 6 67 33 42

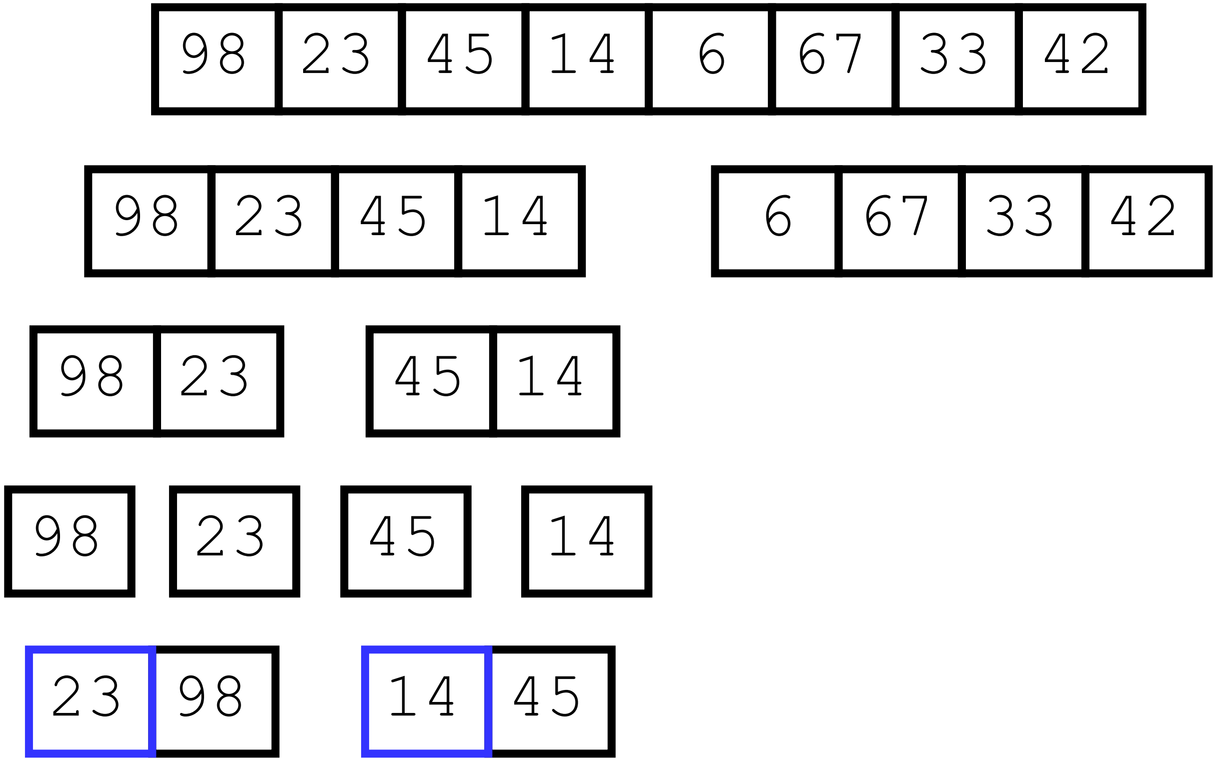
98 23 45 14   6 67 33 42

98 23   45 14

98   23   45   14

23 98   14 45

Merge



98 23 45 14 6 67 33 42

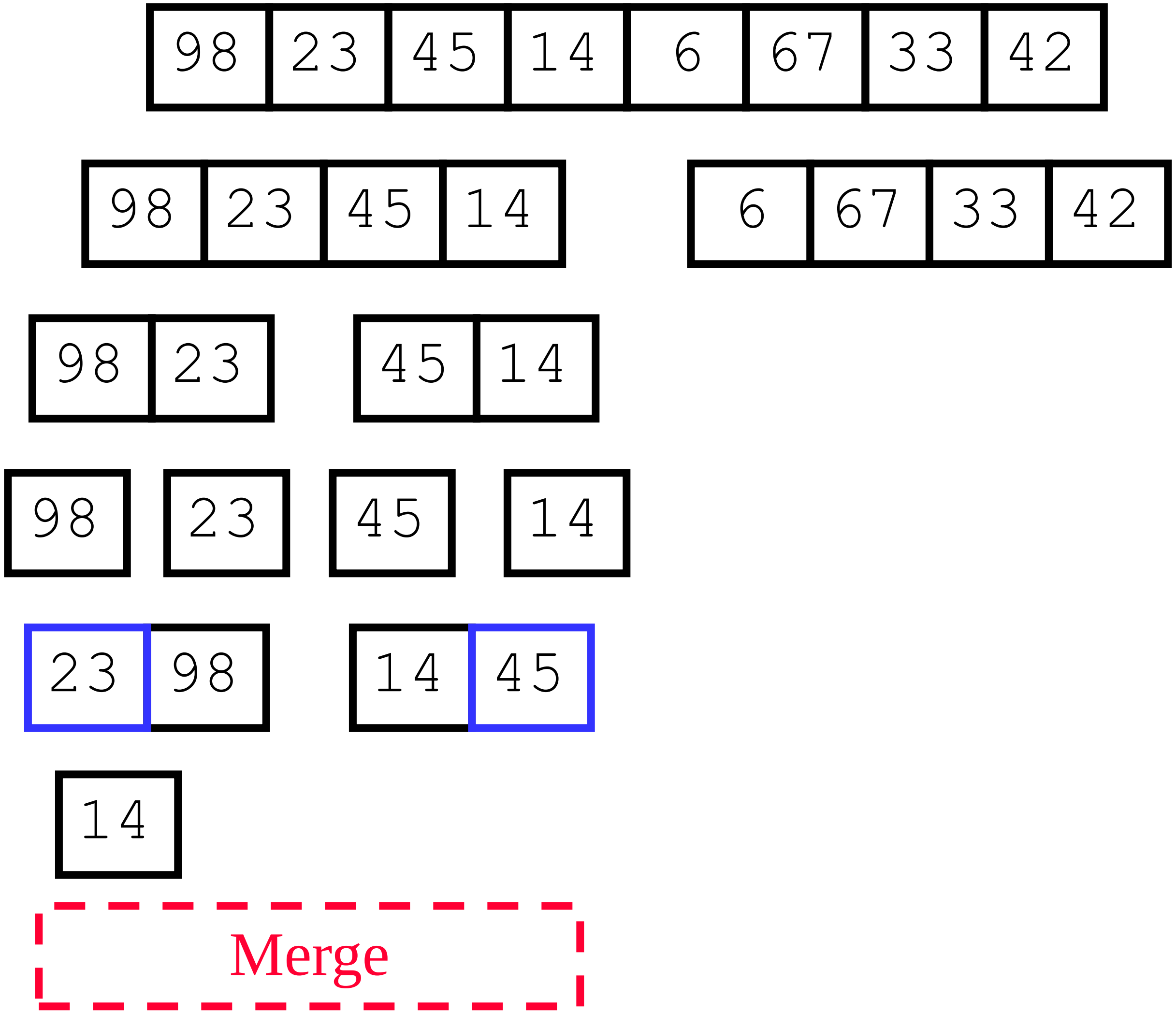
98 23 45 14 6 67 33 42

98 23 45 14

98 23 45 14

23 98 14 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

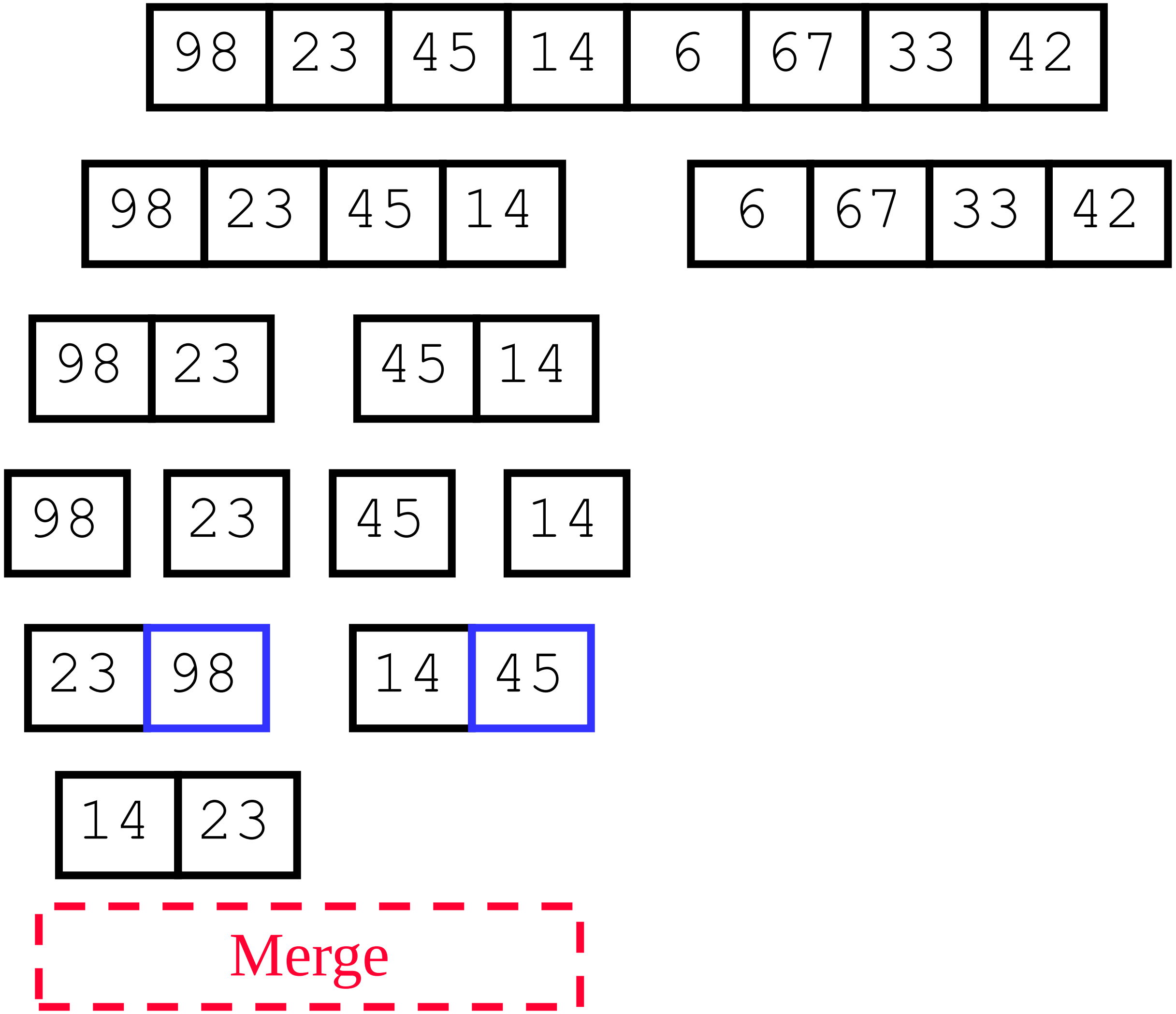
98 23 45 14

98 23 45 14

23 98 14 45

14

Merge



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

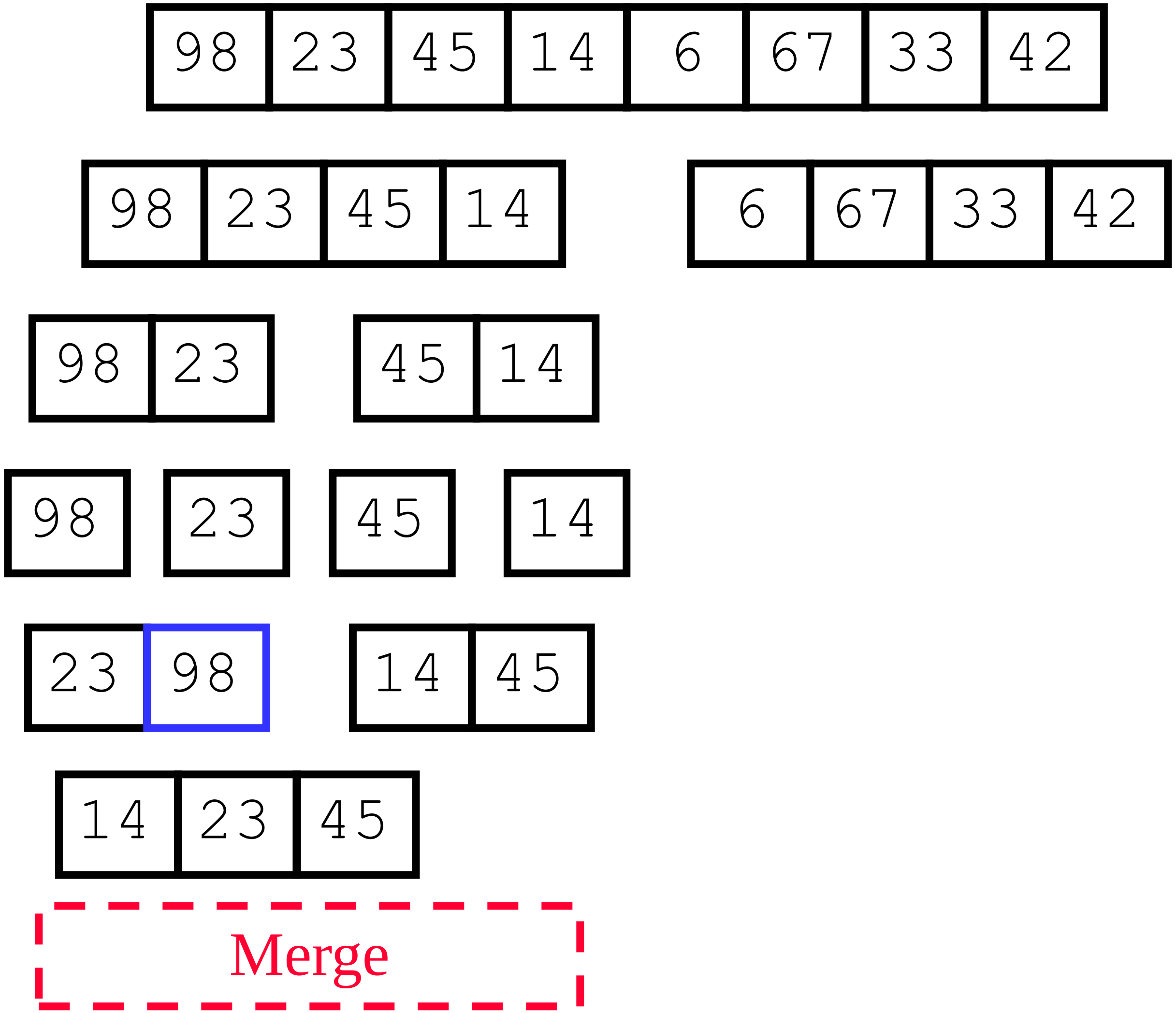
98 23 45 14

98 23 45 14

23 98 14 45

14 23

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

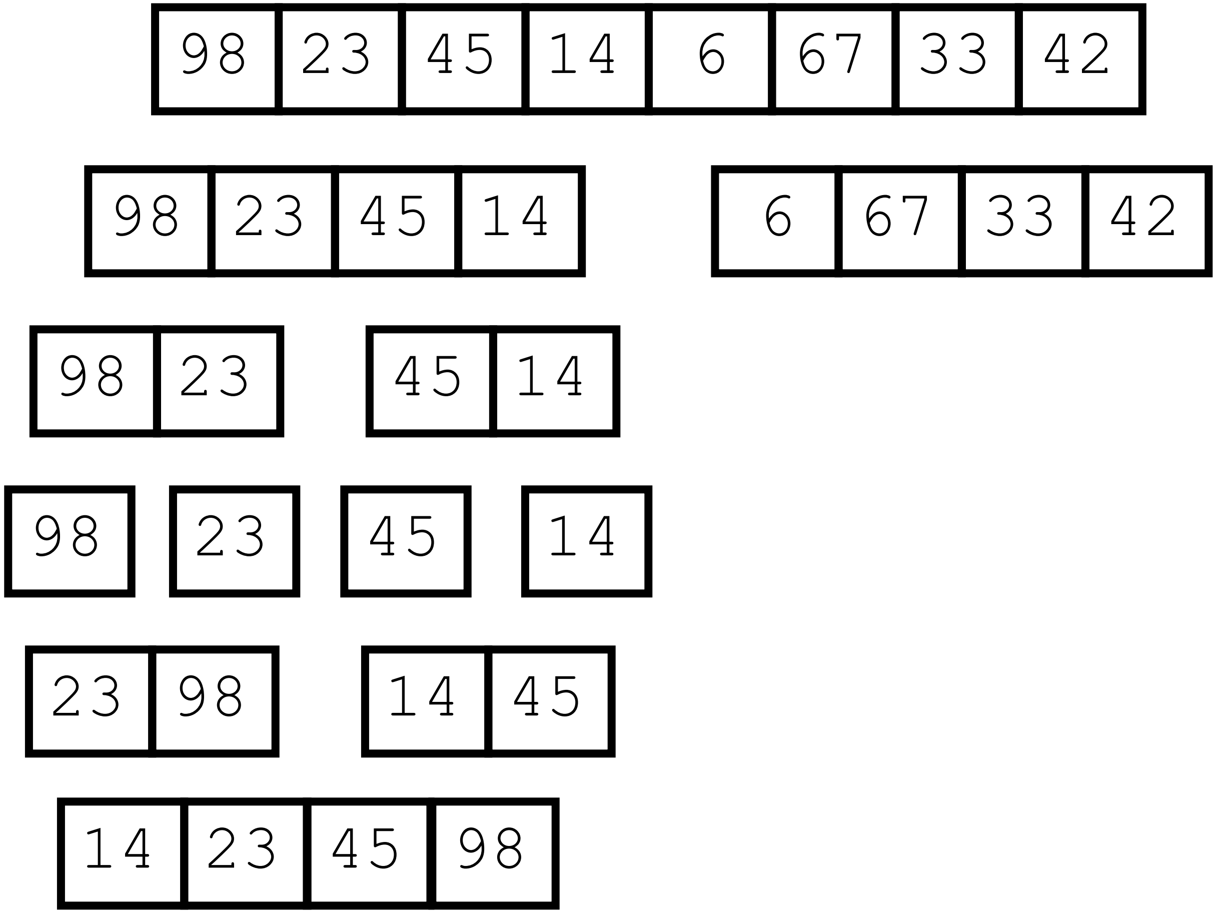
98 23   45 14

98   23   45   14

23 98   14 45

14 23 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

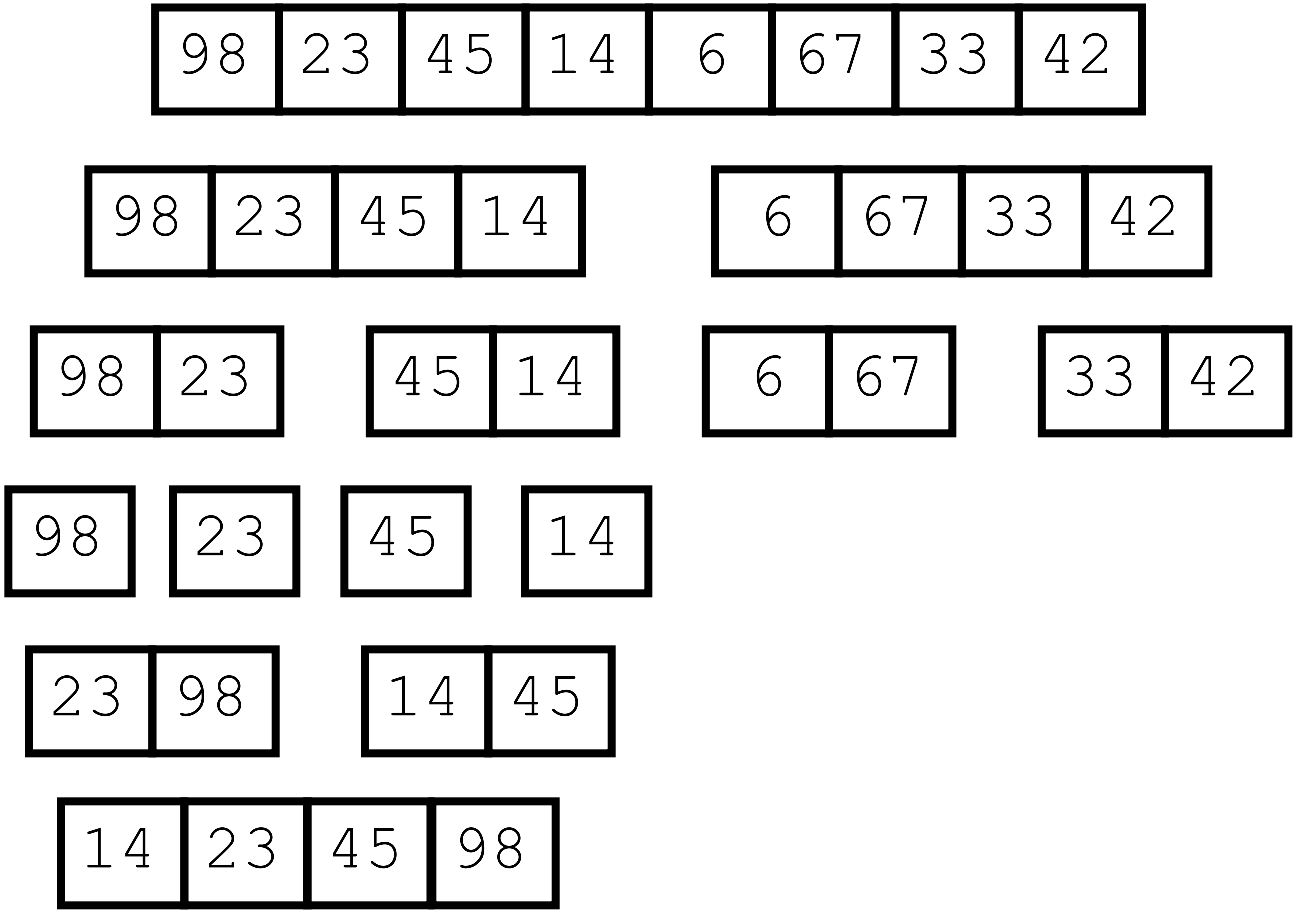
98 23   45 14

98   23   45   14

23 98   14 45

14 23 45 98

Merge



98 23 45 14 6 67 33 42

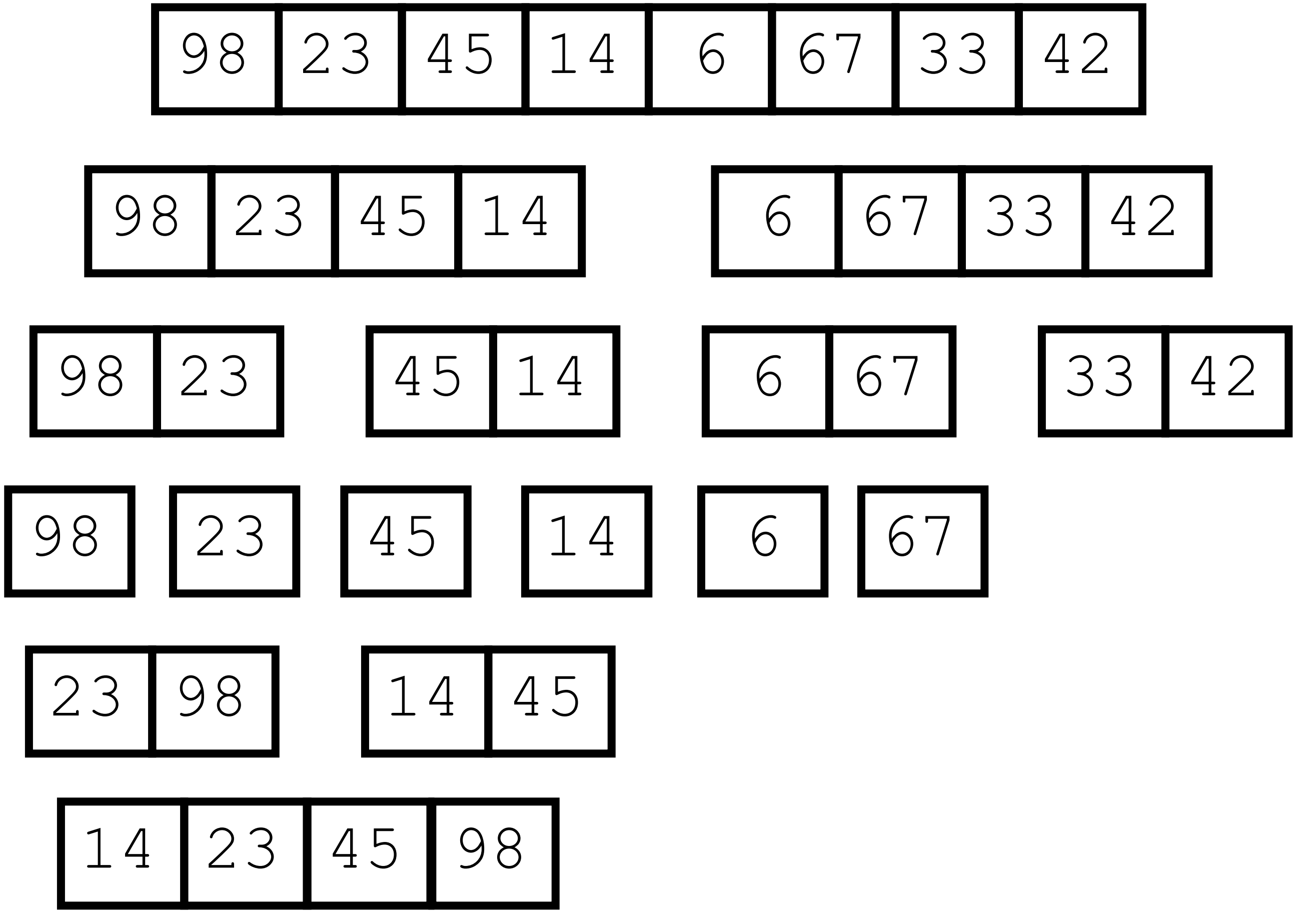
98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14

23 98 | 14 45

14 23 45 98



98 23 45 14 6 67 33 42

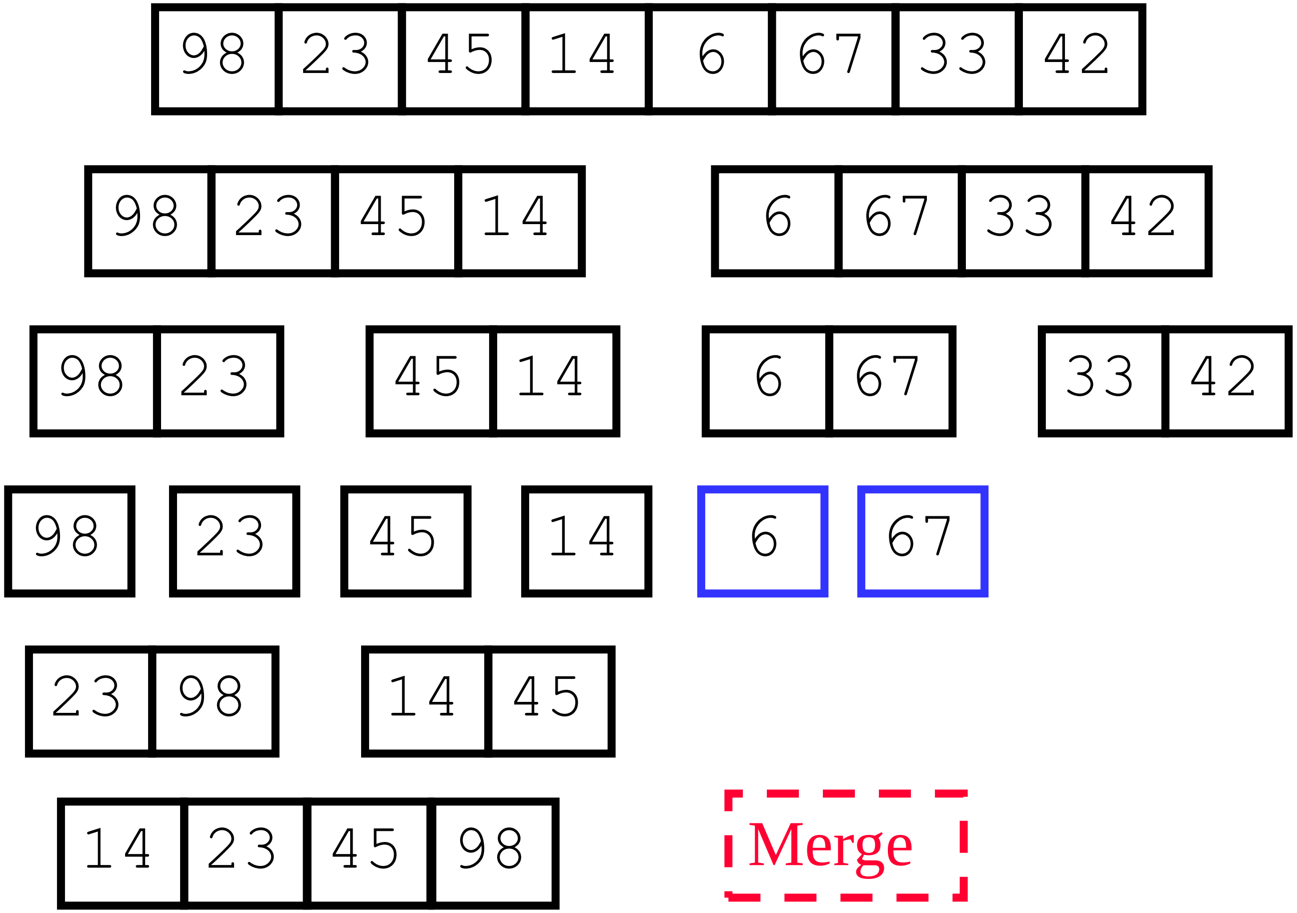
98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67

23 98 | 14 45

14 23 45 98



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

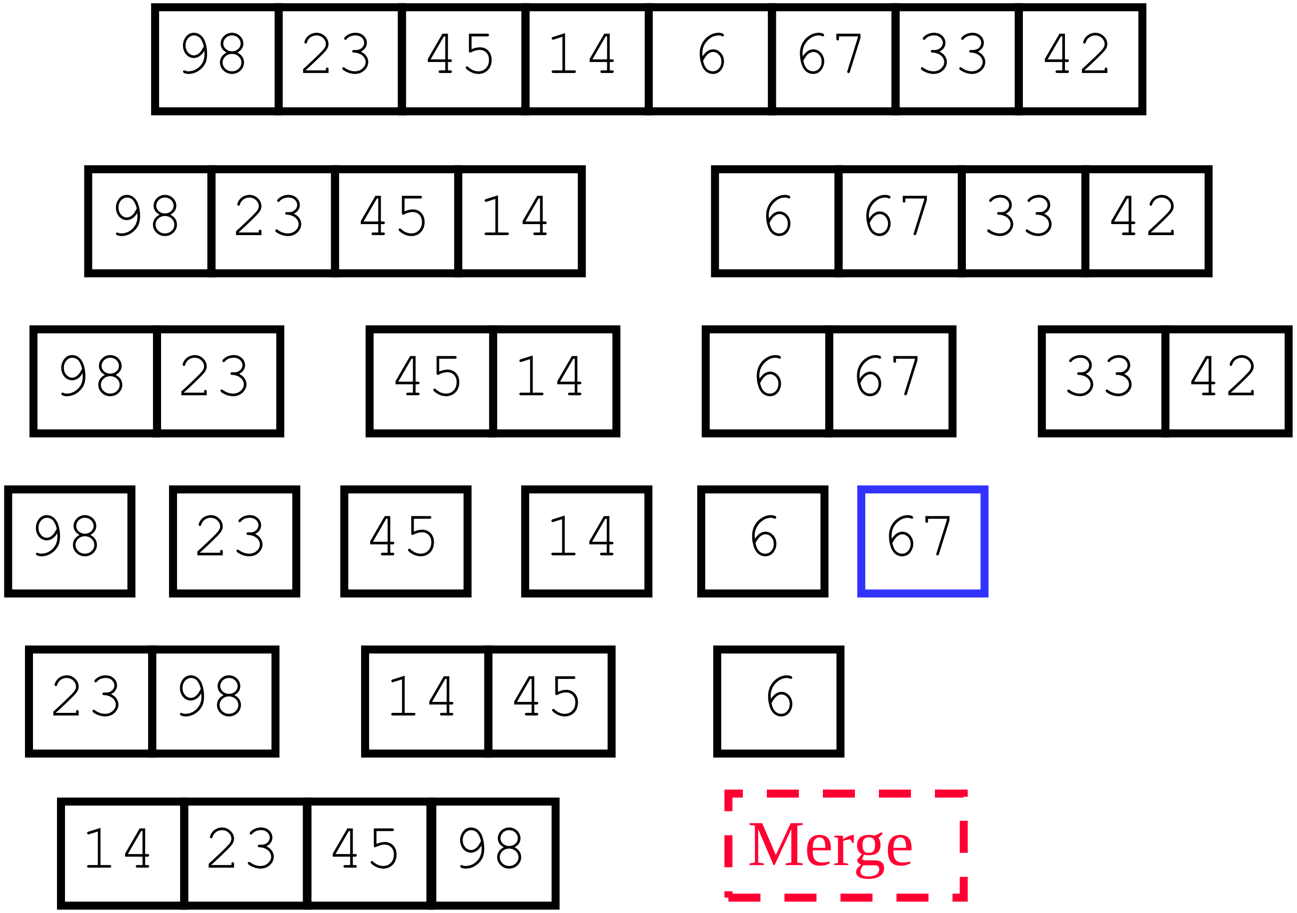
98 23   45 14   6 67   33 42

98   23   45   14   6   67

23 98   14 45

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

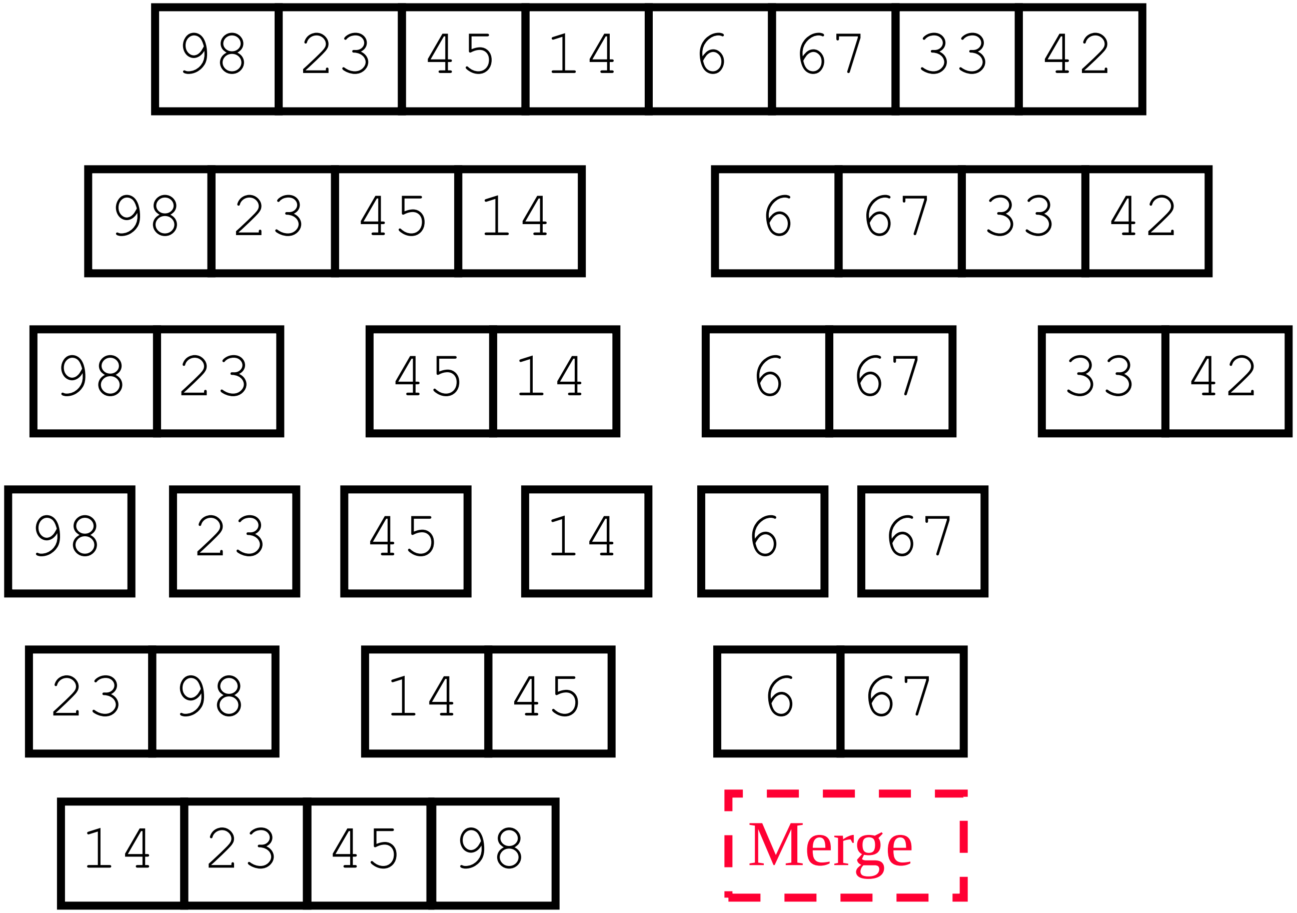
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67

23 98 | 14 45 | 6

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

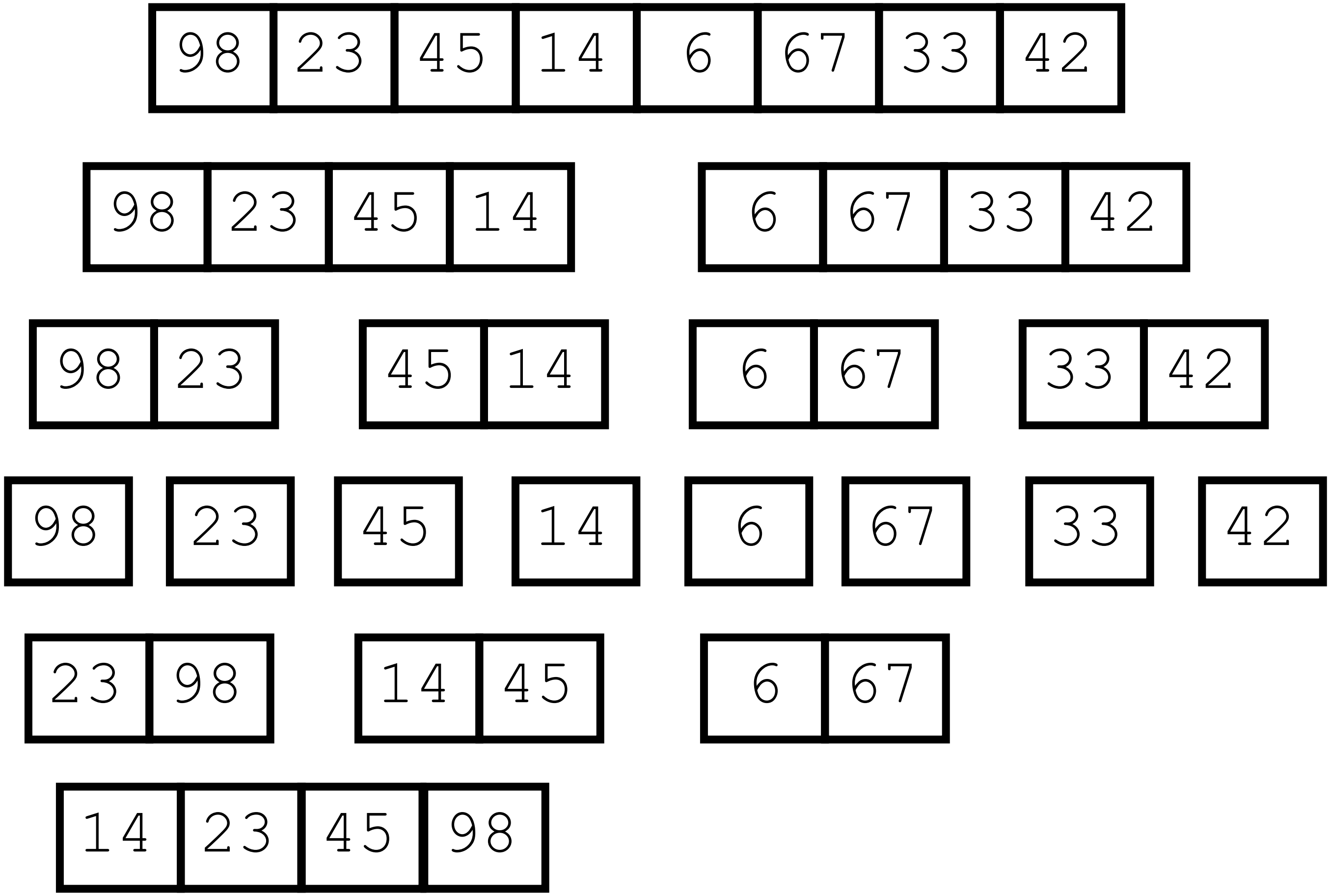
98 23   45 14   6 67   33 42

98   23   45   14   6   67

23 98   14 45   6 67

14 23 45 98

Merge



98 23 45 14 6 67 33 42

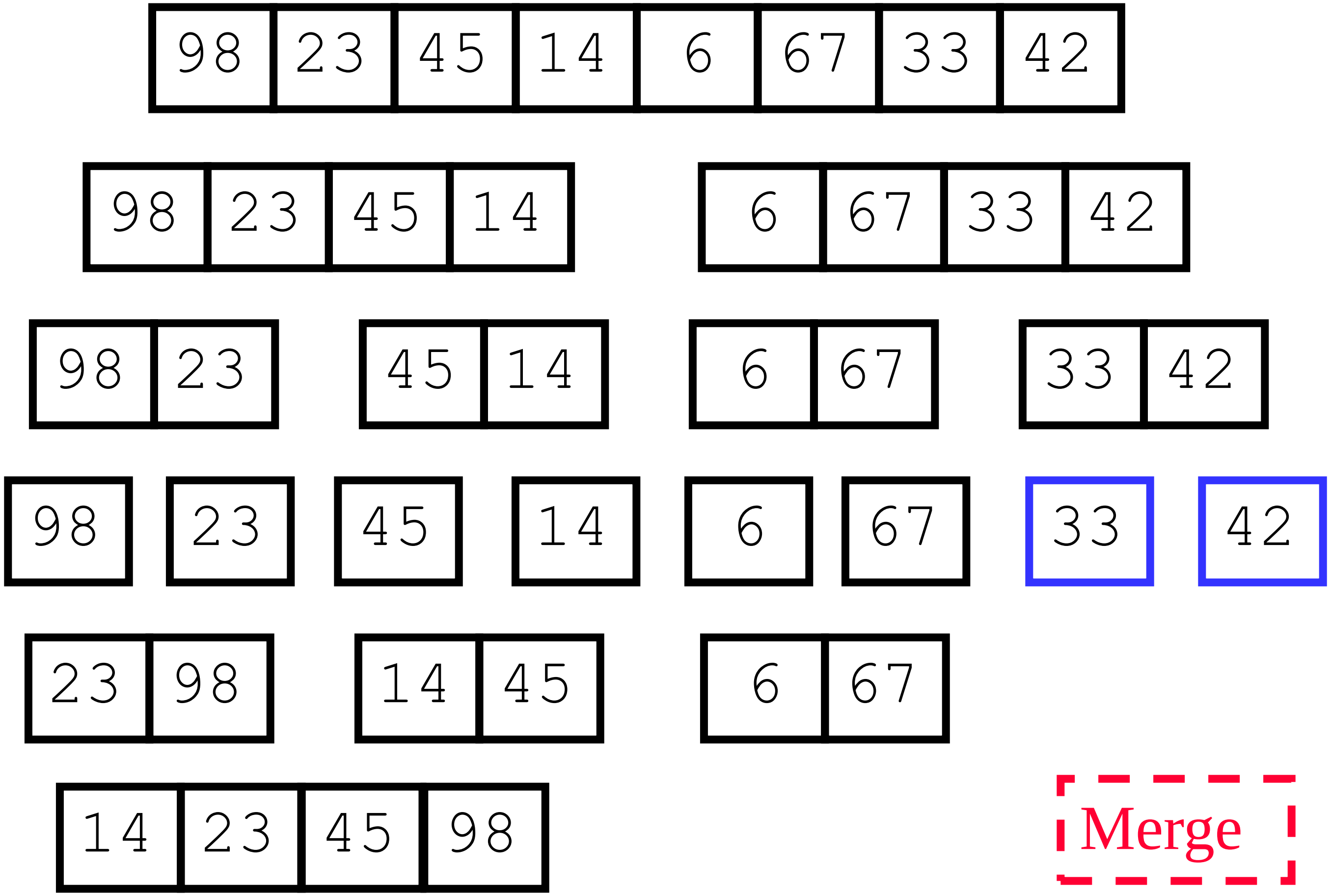
98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67

14 23 45 98



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

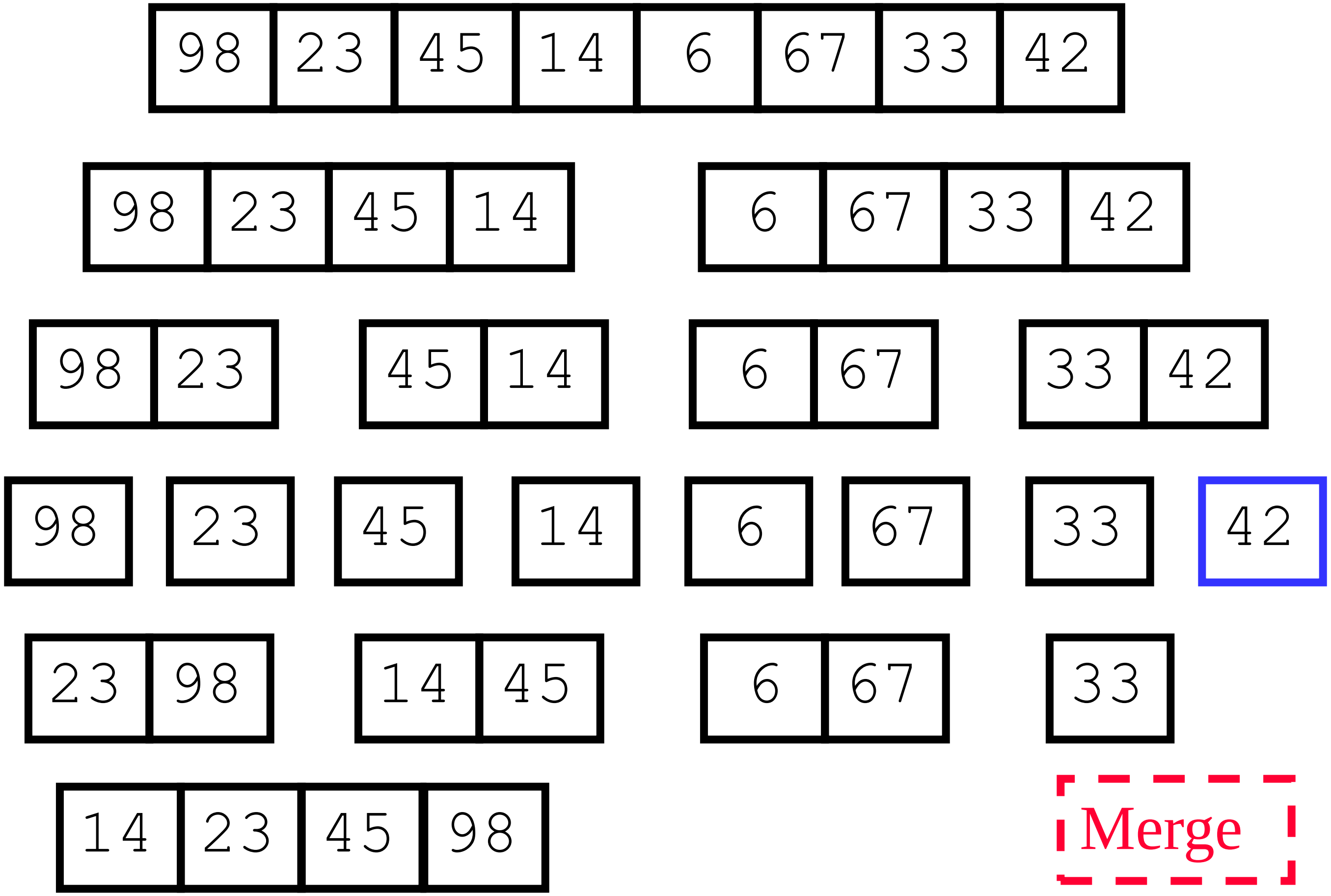
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

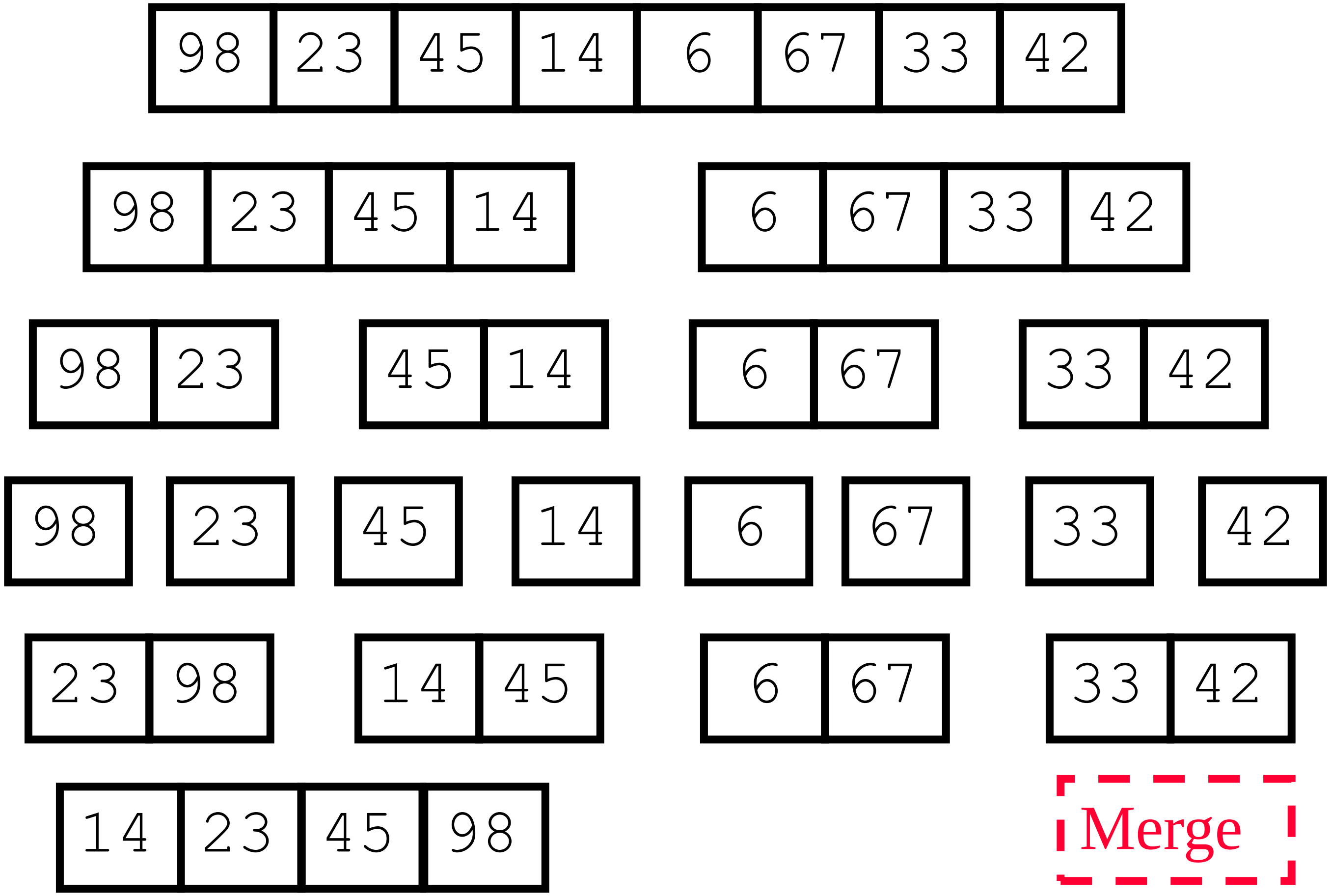
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

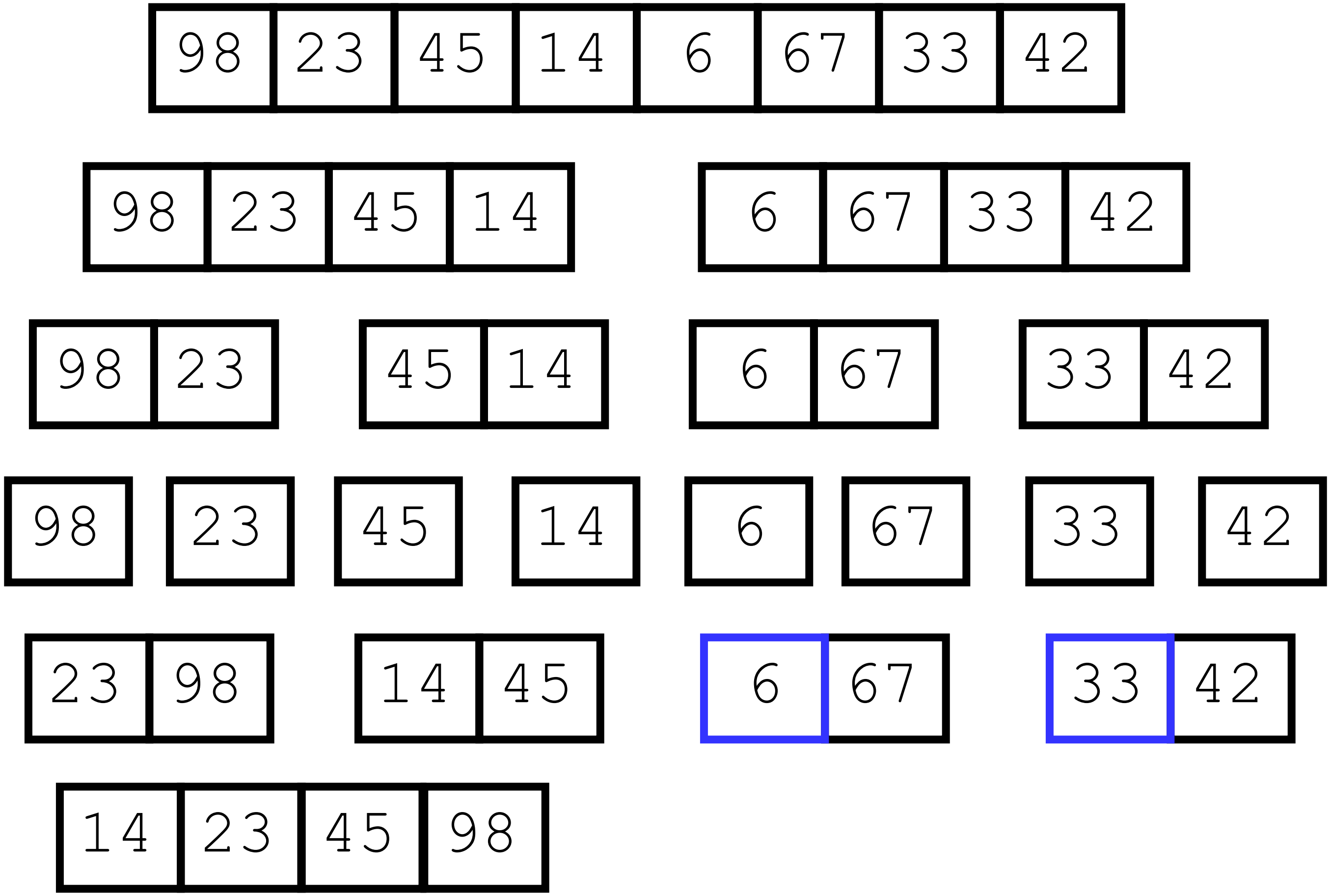
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

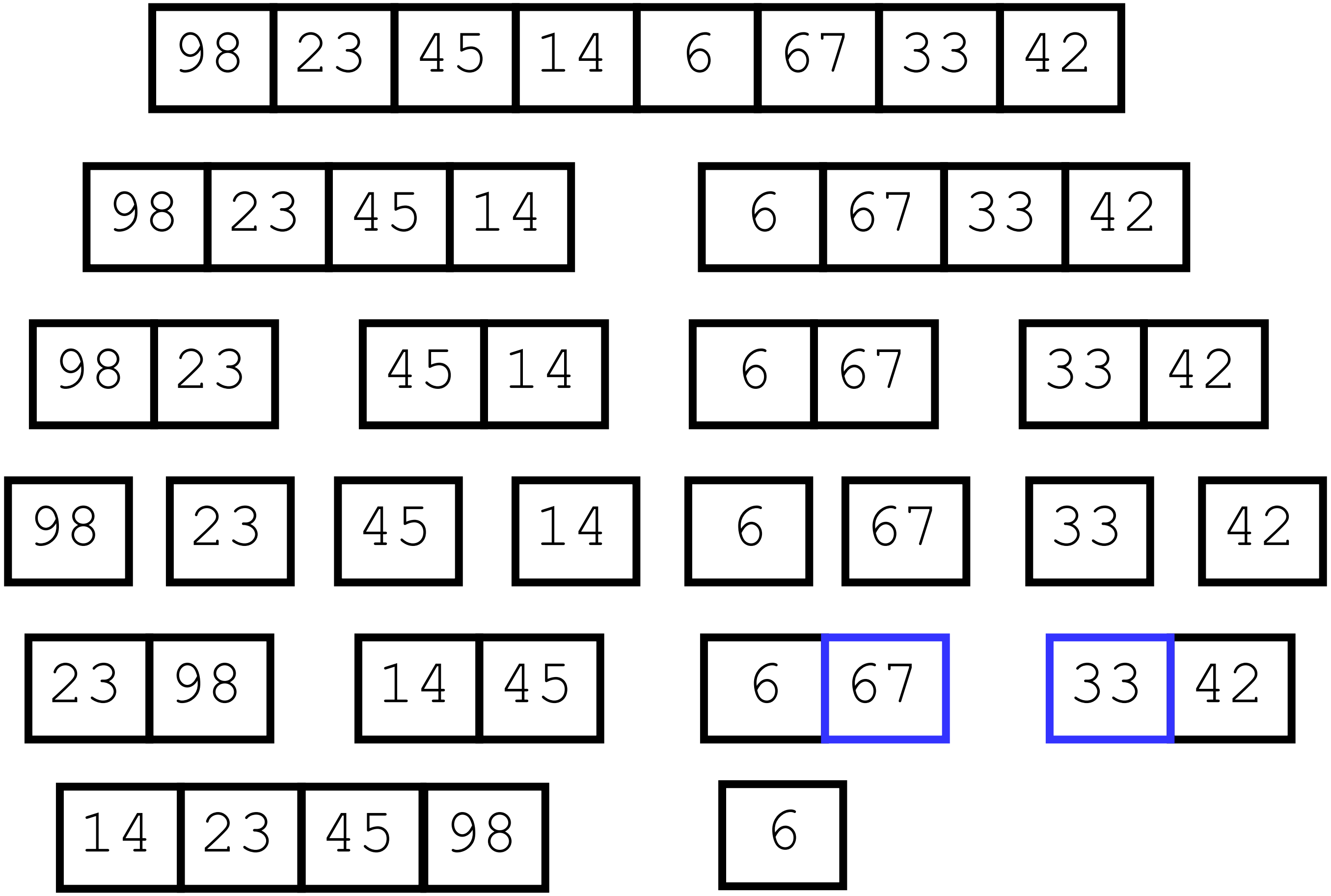
98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

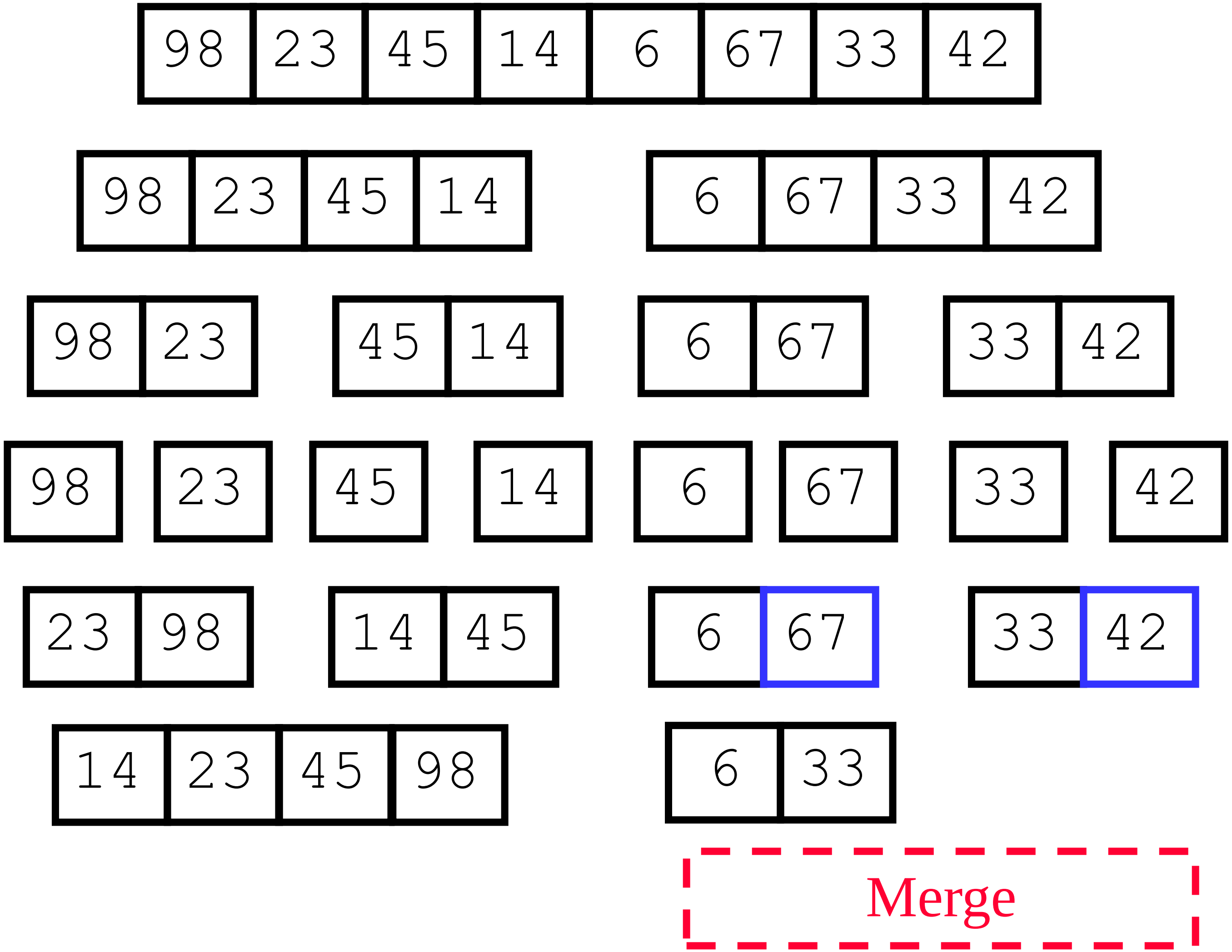
98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

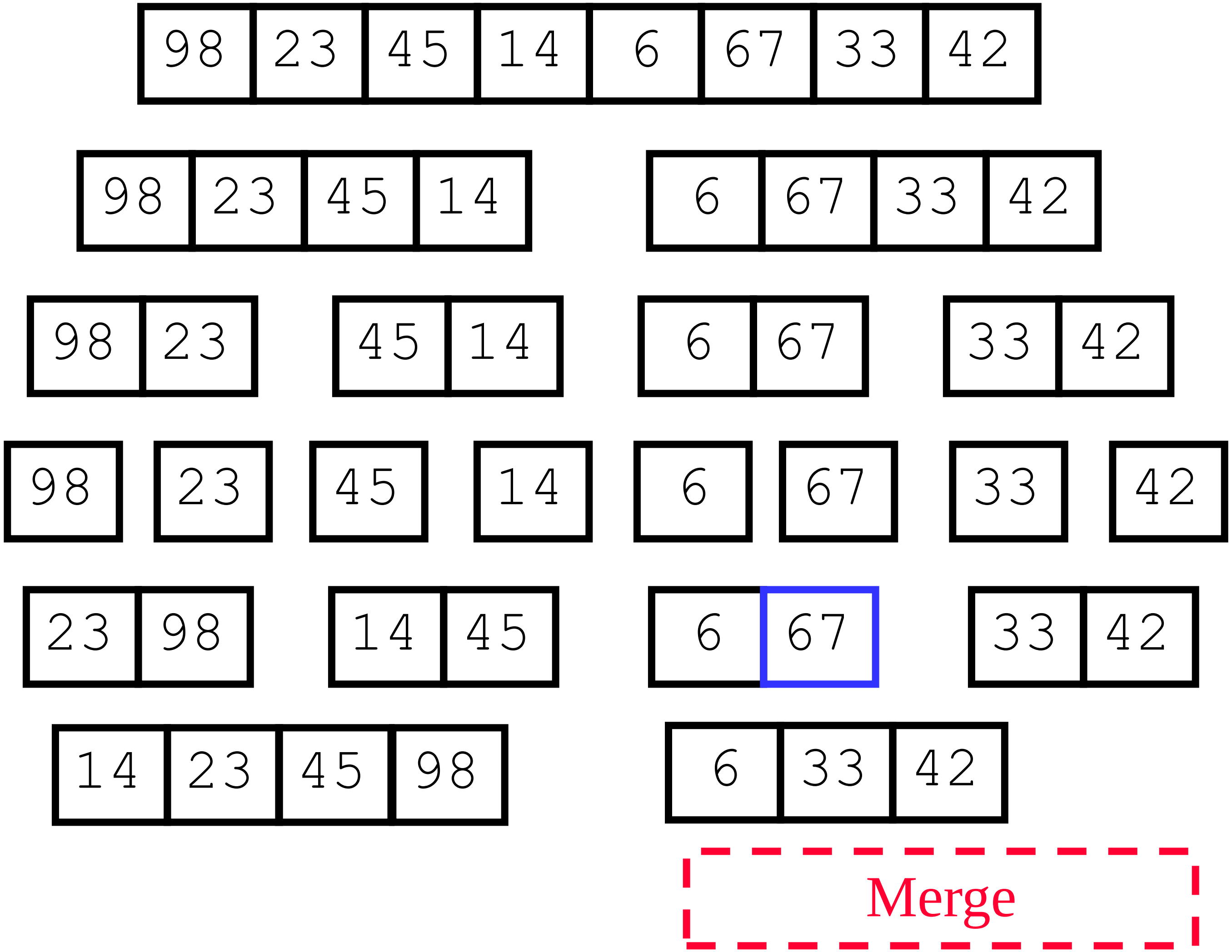
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

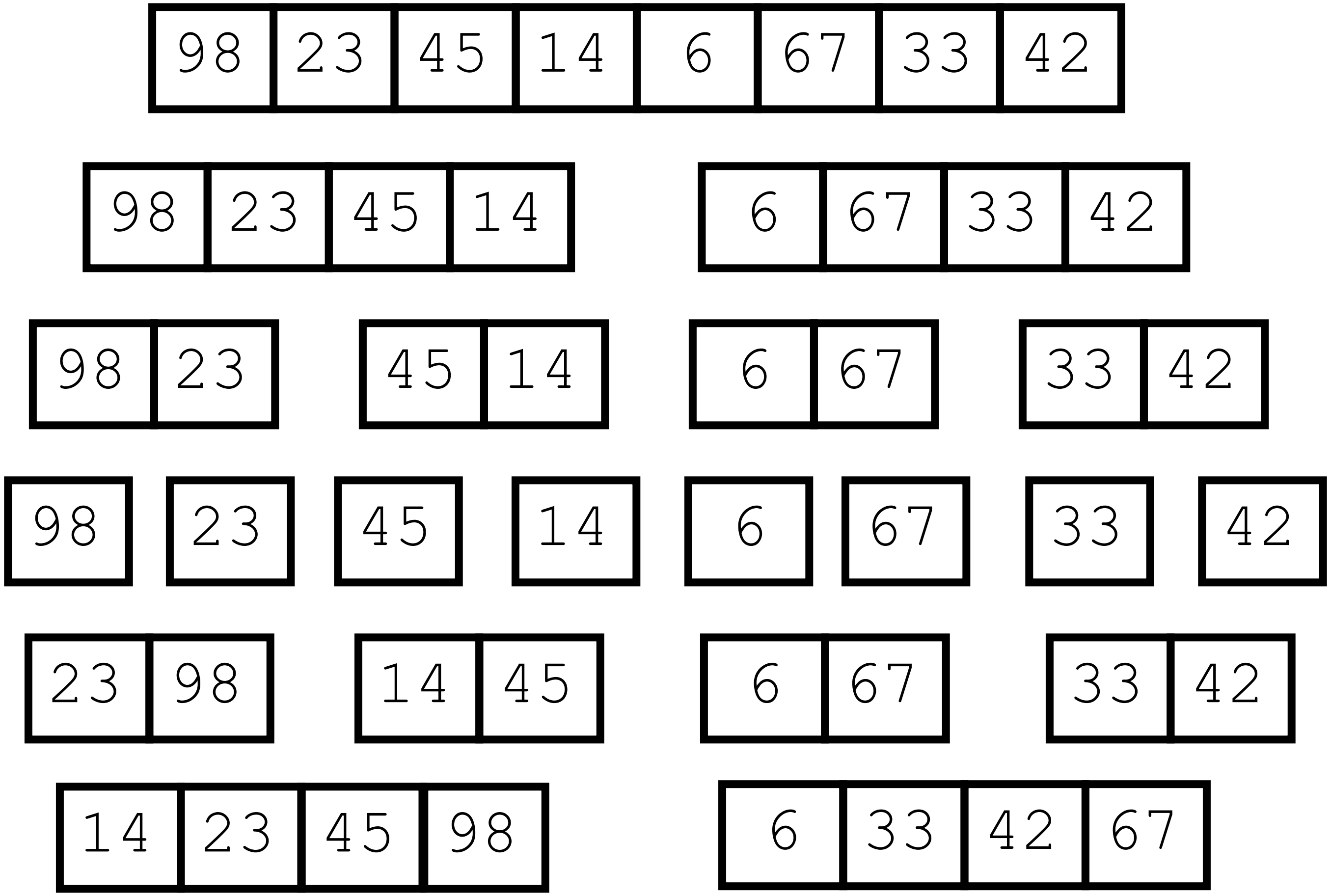
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

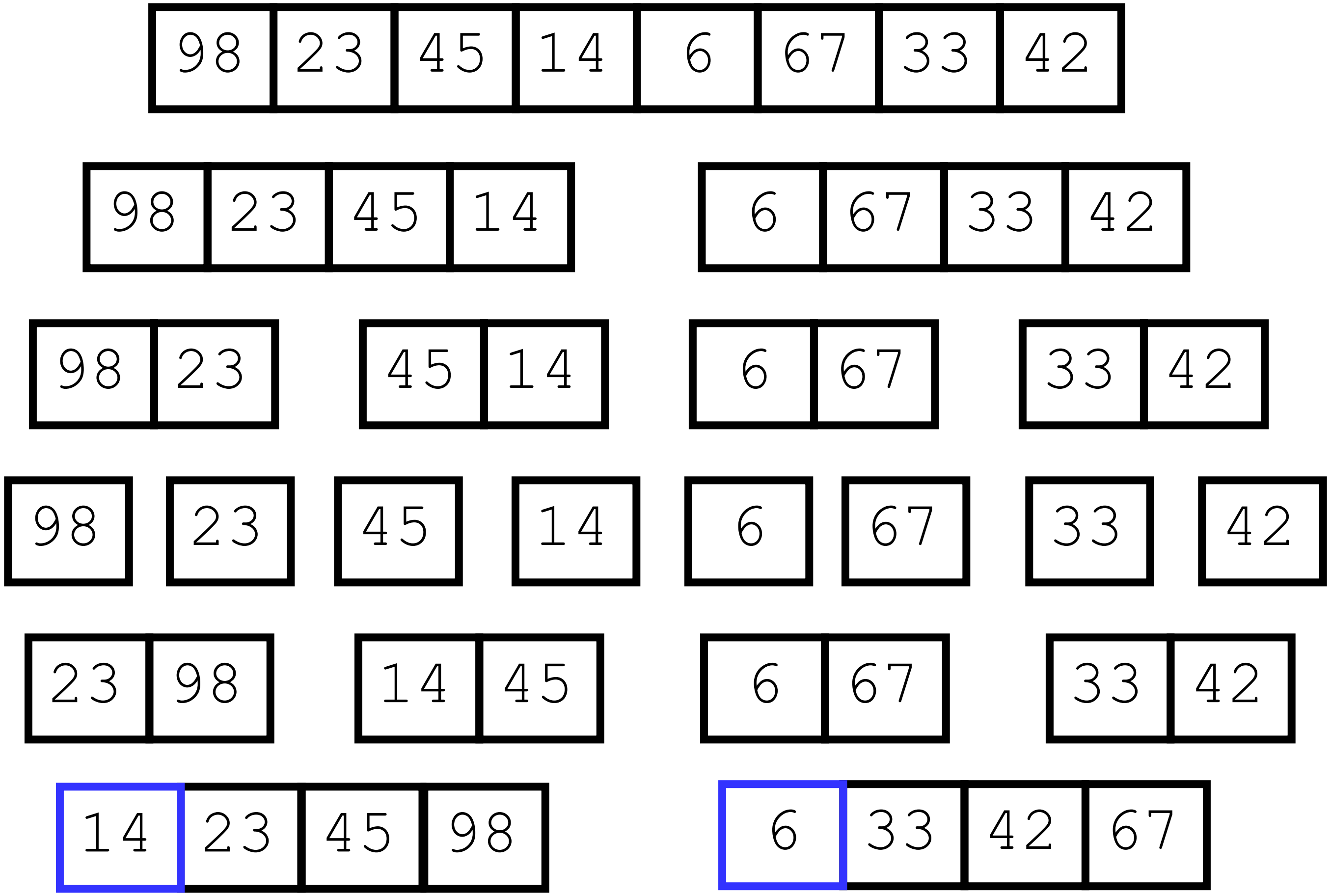
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

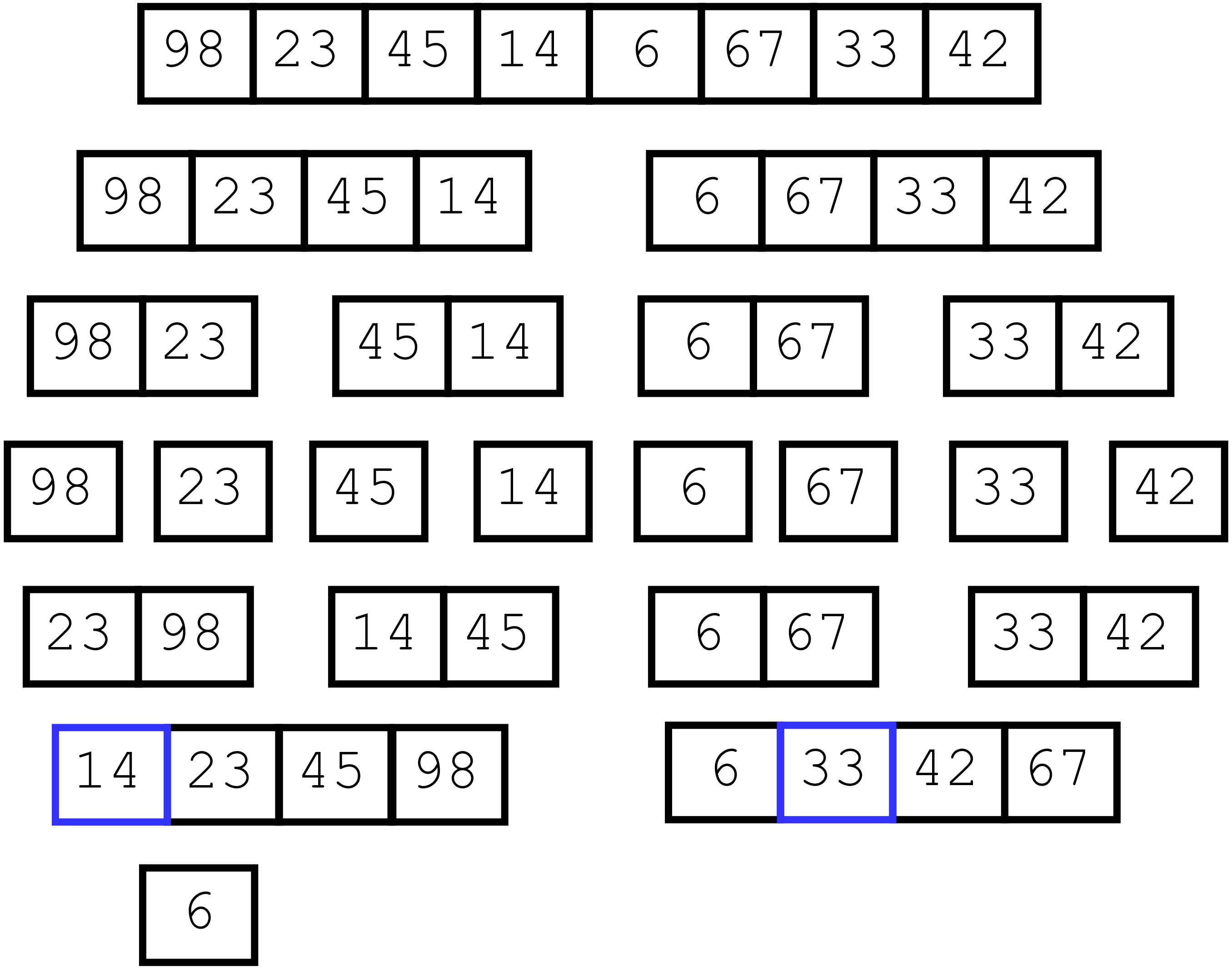
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

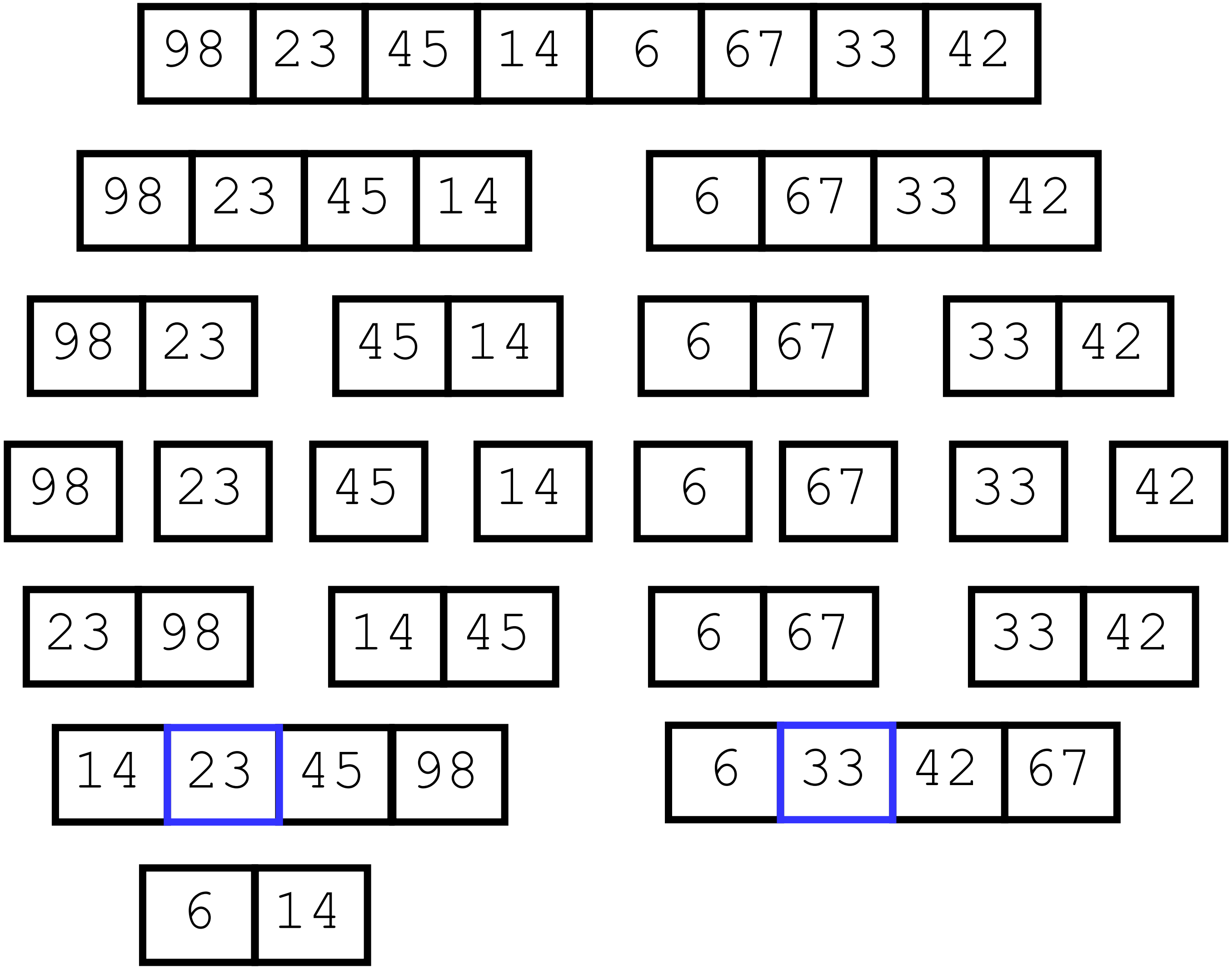
98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

6

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

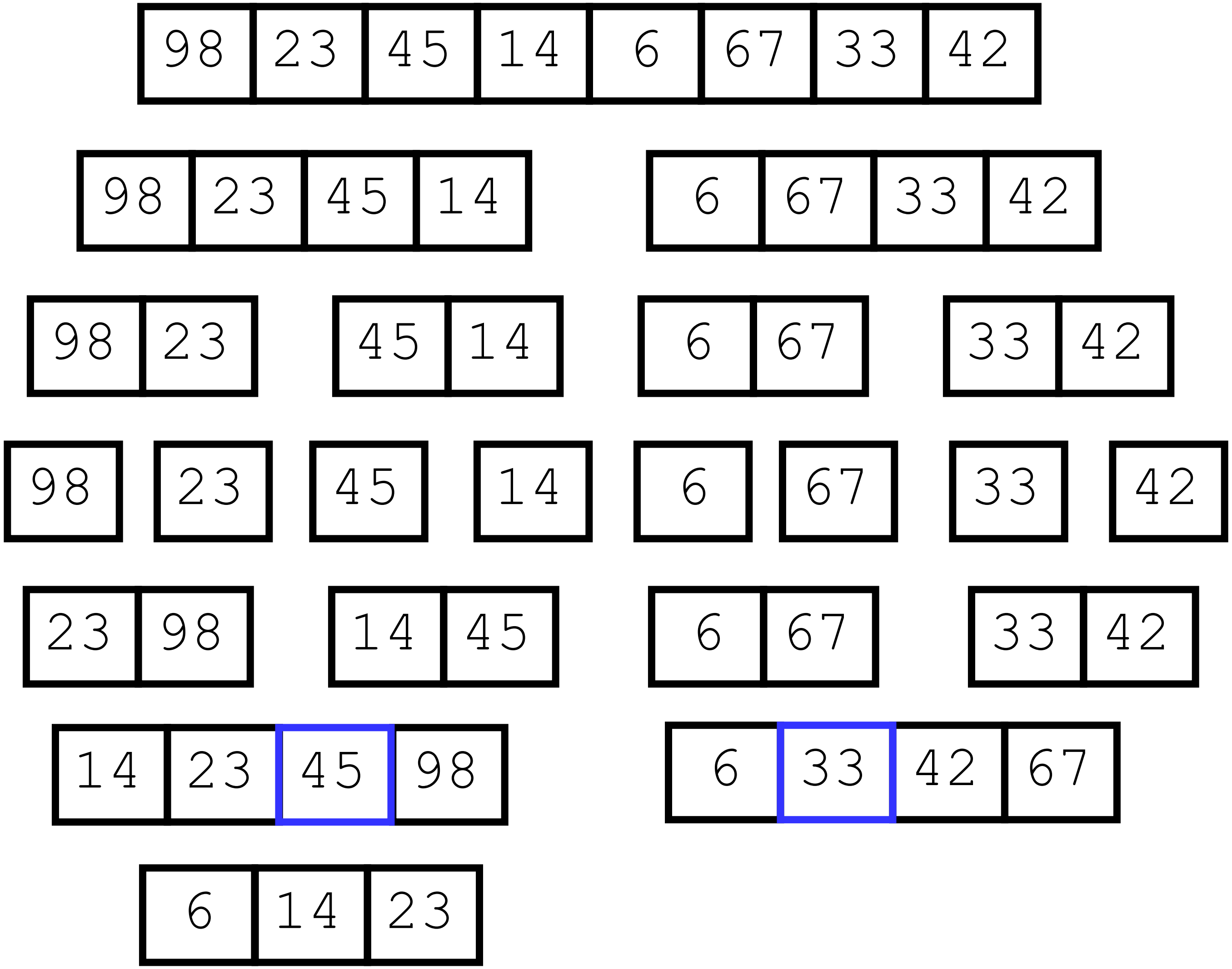
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

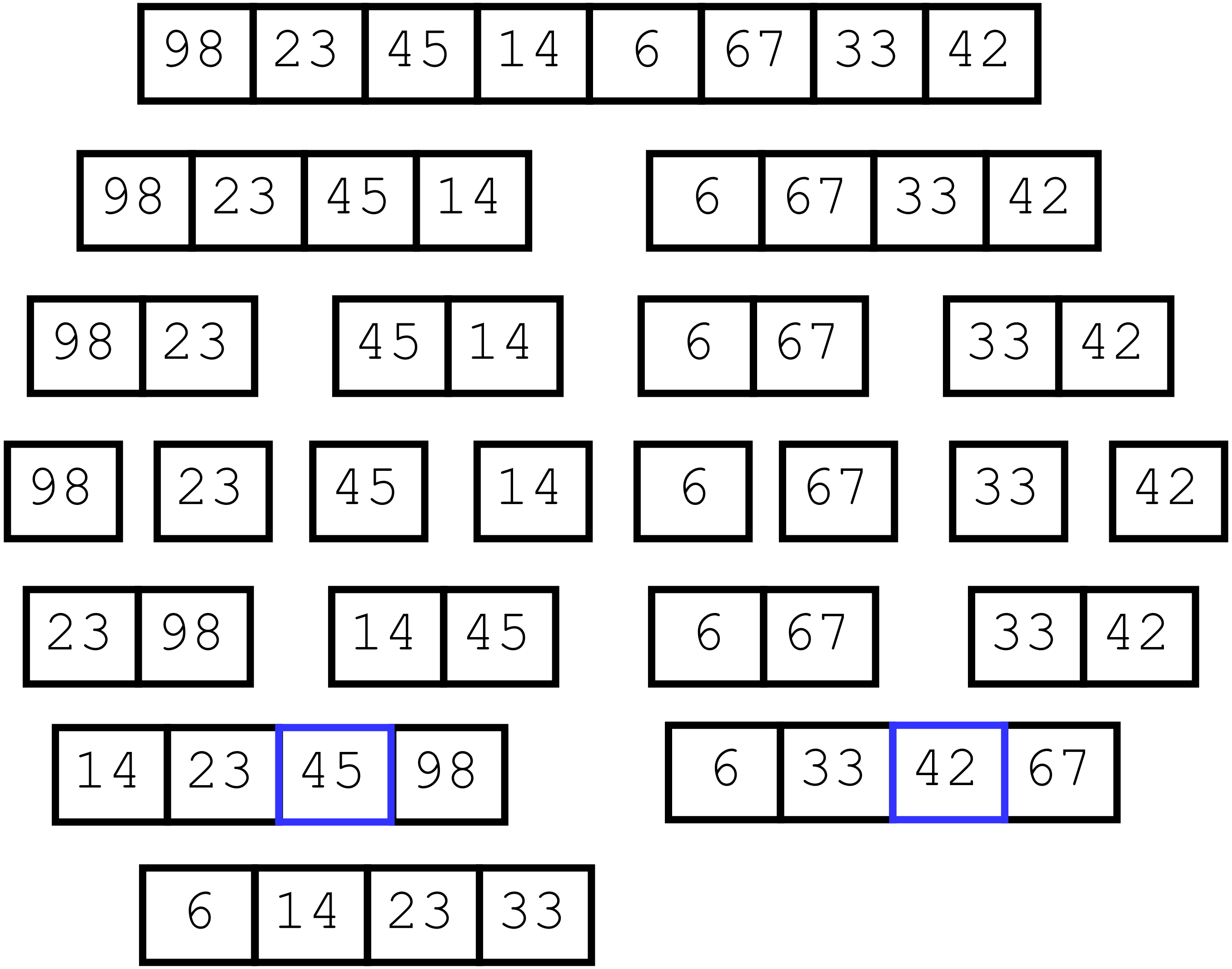
98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

6 14 23

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

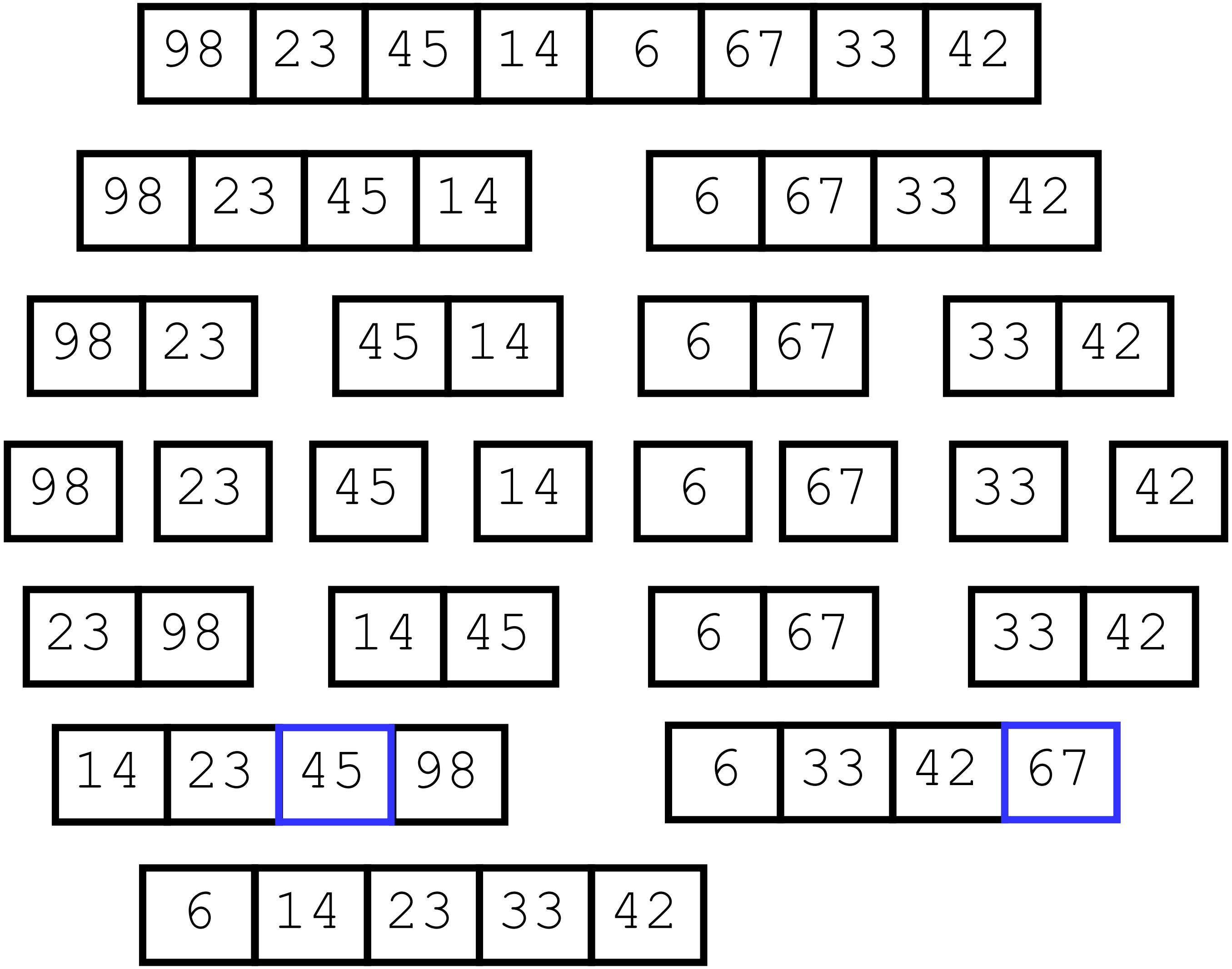
98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

6 14 23 33

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

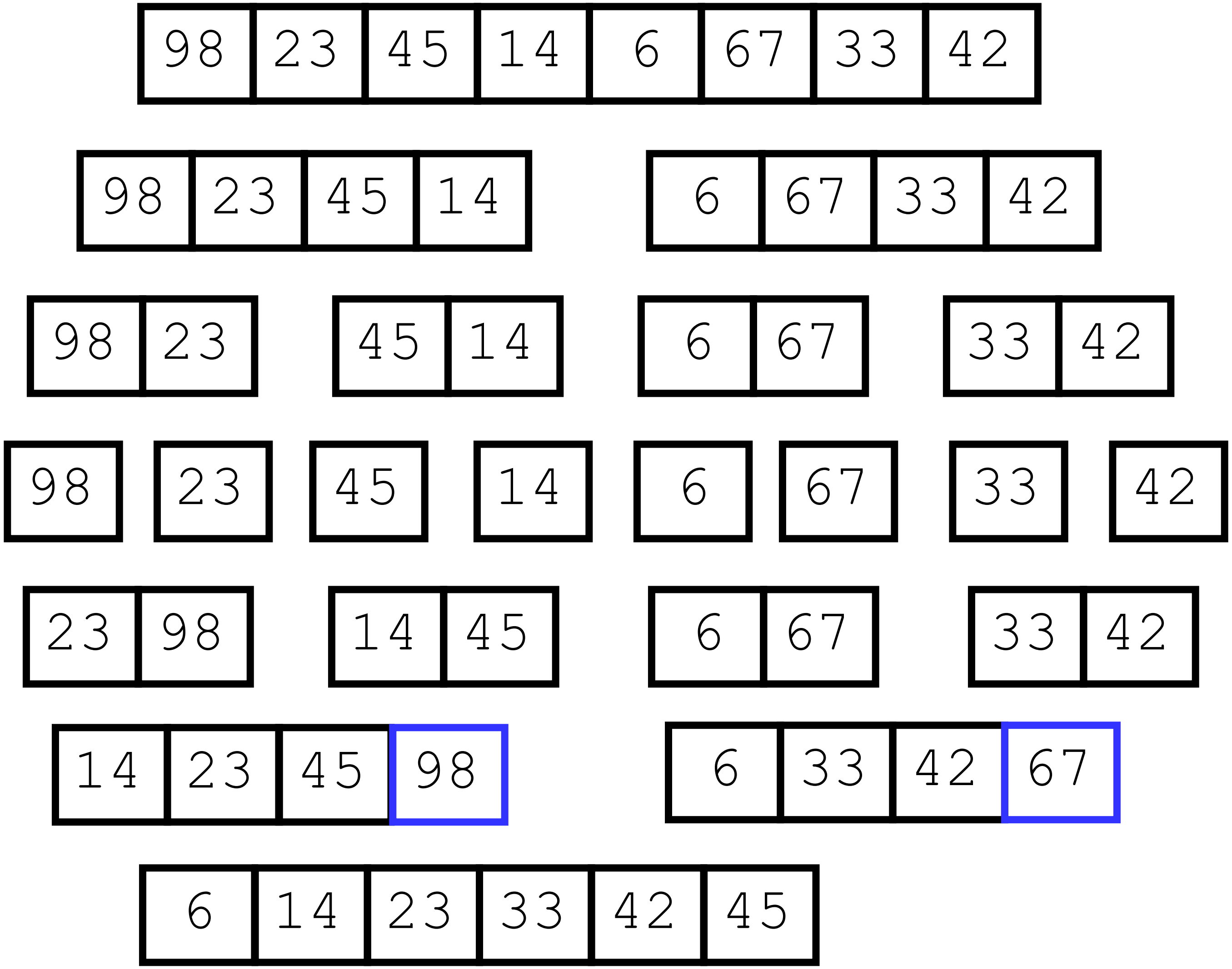
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

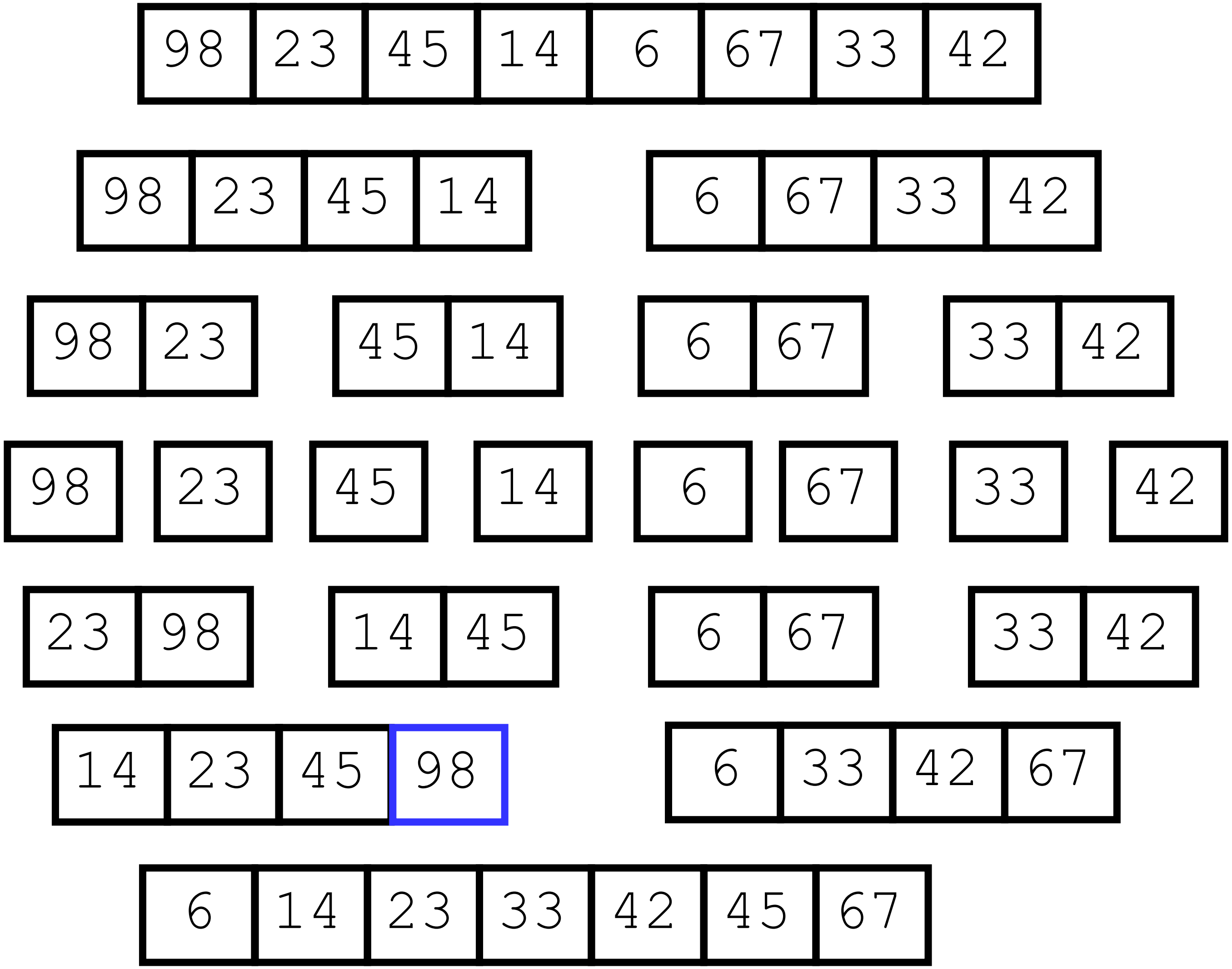
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

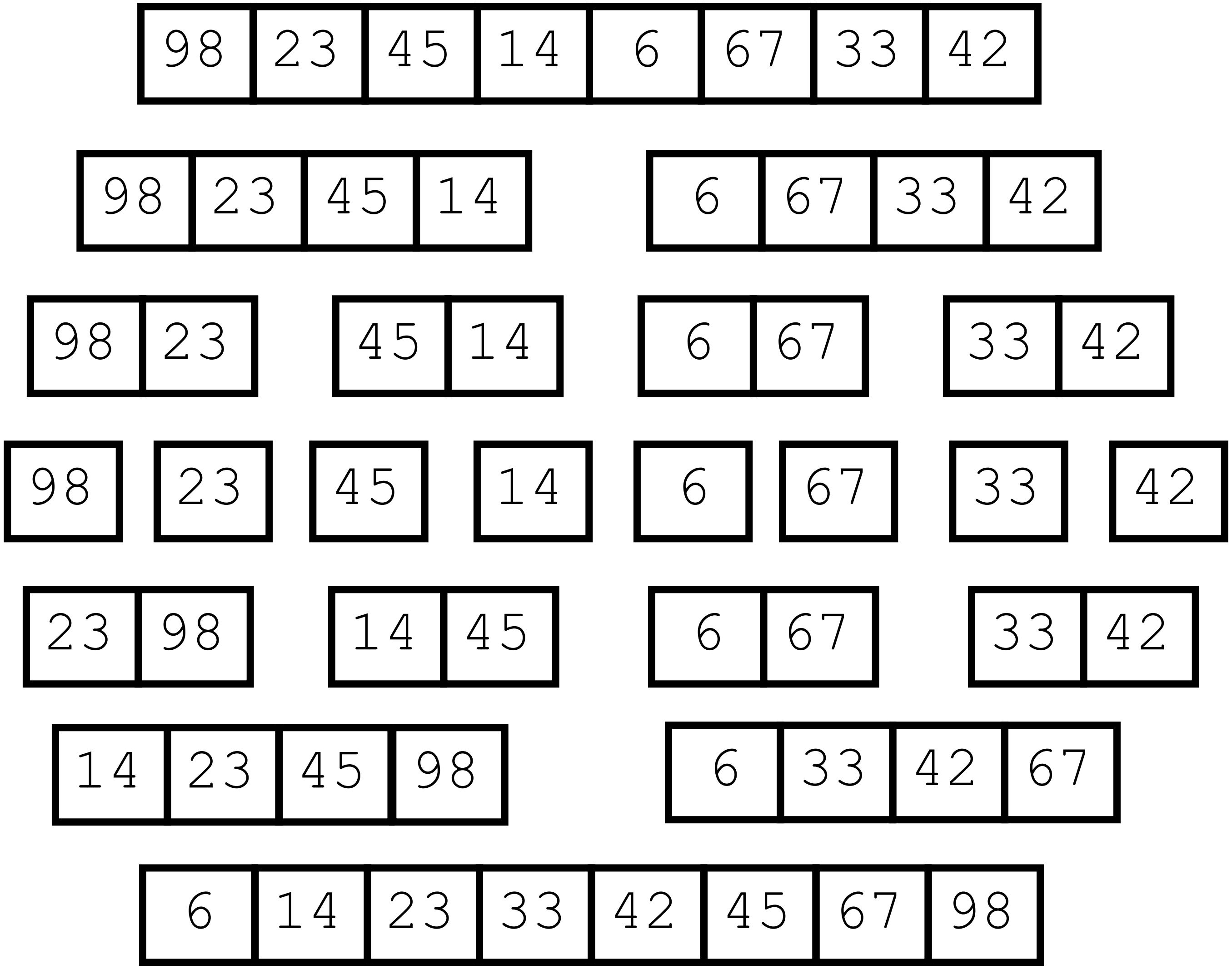
98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

6 14 23 33 42 45 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

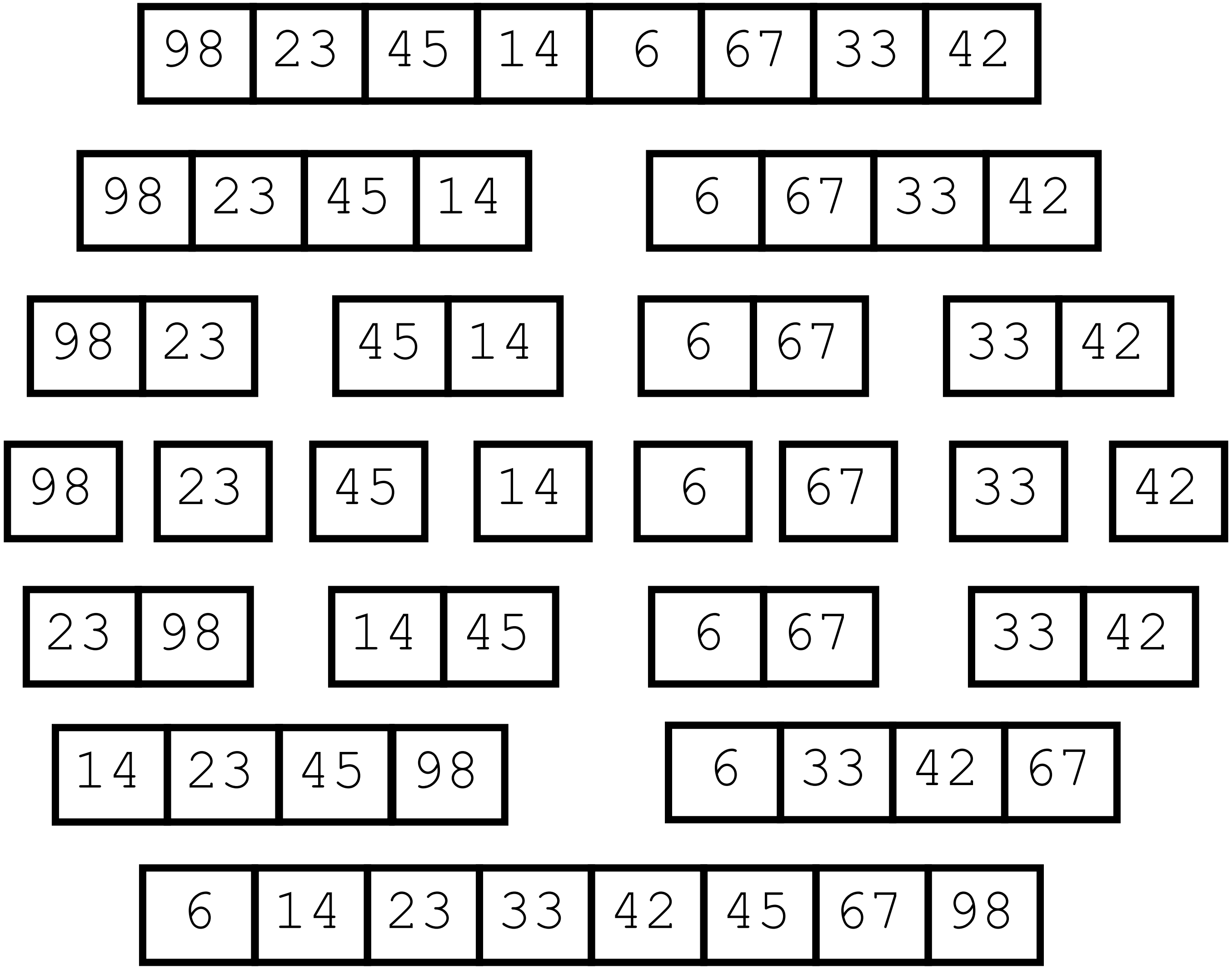
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

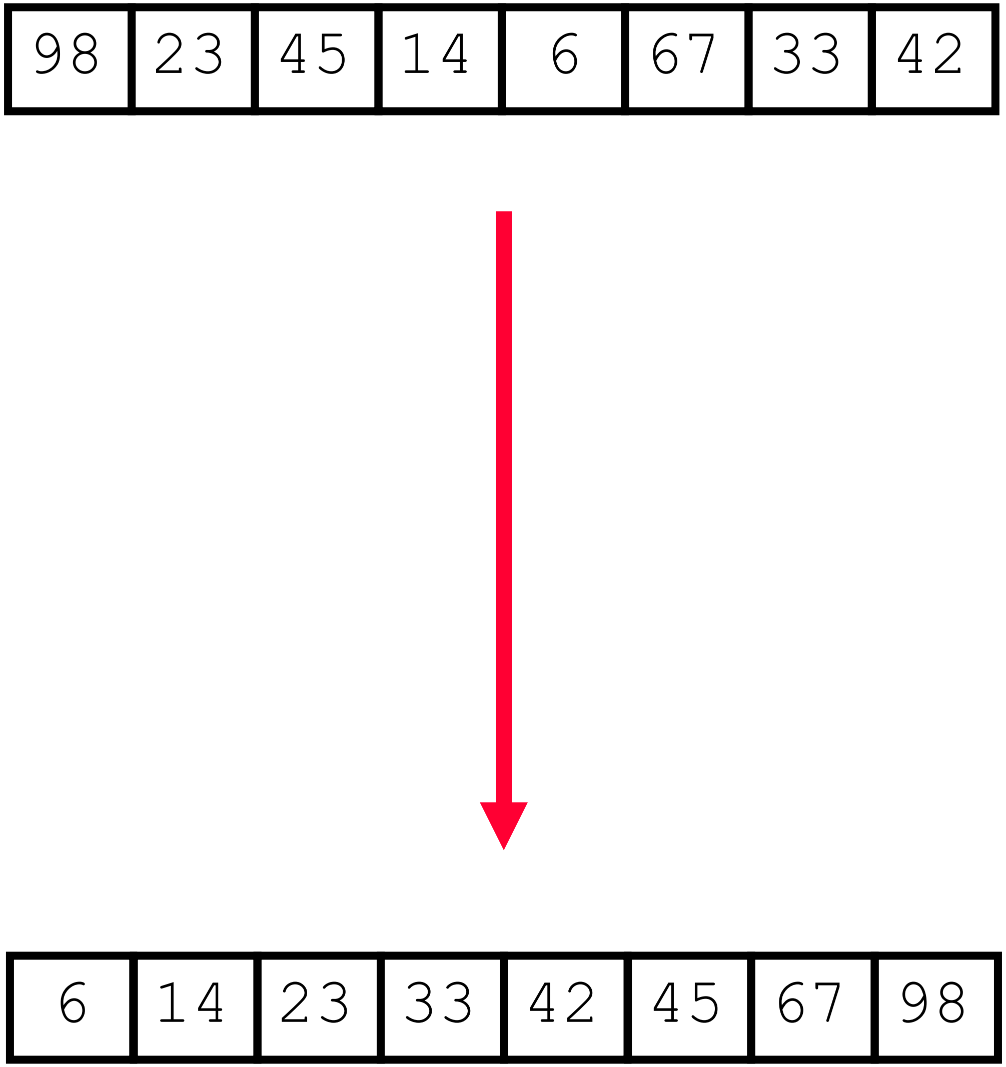
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67 98



98	23	45	14	6	67	33	42

6	14	23	33	42	45	67	98

Summary

- **Divide the unsorted collection into two**

- **Until the sub-arrays only contain one element**

- **Then merge the sub-problem solutions together**

합병 정렬 (4)

- mergeSort 프로그램(1)

```
public static void mergeSort(int[] a)  {
          // 합병 정렬의 메인 메소드
   int[] temp[a.length];  //원소의 개수에 비례하는 추가적인 공간
   internalMergeSort(a, temp, 0, a.length-1);
}

private static void internalMergeSort(int[] a, int[] temp, int m, int n)  {
          // 순환 호출을 하는 mergeSort()의 보조 메소드
   if (m < n)  {     // 정렬할 원소가 2개 이상인 경우
      int middle = (m+n) / 2;
      internalMergeSort(a, temp, m, middle);
      internalMergeSort(a, temp, middle+1, n);
      merge(a, temp, m, middle, middle+1, n);
   }
}
```

합병 정렬 (5)

- mergeSort 프로그램(2)

```
private static void merge(int[] a, int[] temp, int m, int p, int q, int n)  {
            // internalMergeSort()의 보조 메소드
    int t = m;
    int numElements = n – m + 1;
    while (m <= p && q <= n)  {
              if (a[m] < a[q])
                   temp[t++] = a[m++];
              else temp[t++] = a[q++];
    }
    while (m <= p)     // 왼쪽 부분 배열에 원소가 남아 있는 경우
              temp[t++] = a[m++];
    while (q <= n)     // 오른쪽 부분 배열에 원소가 남아 있는 경우
              temp[t++] = a[q++];
    for (int i = 0,  i < numElements; i++, n--)     // 배열 temp[]를 a[]로 복사
           a[n] = temp[n];
}
```

- 단점 : 주어진 배열과 동일한 크기의 임시배열 **temp[]**가 필요

Heap sort(히프정렬)
heap: 쌓아 올린 더미

- We already studied Heap.

Heaps (1/6)

- The heap abstract data type
  - **Definition:** A *max*(*min*) *tree* is a tree in which the key value in each node is no smaller (larger) than the key values in its children. A *max* (*min*) *heap* is a complete binary tree (완전이진트리)that is also a *max* (*min*) *tree*
  - **Basic Operations:**
    - creation of an empty heap
    - insertion of a new element into a heap
    - deletion of the largest element from the heap

Heaps (2/6)

- The examples of max heaps and min heaps
  - Property: The root of max heap (min heap) contains the largest (smallest) element

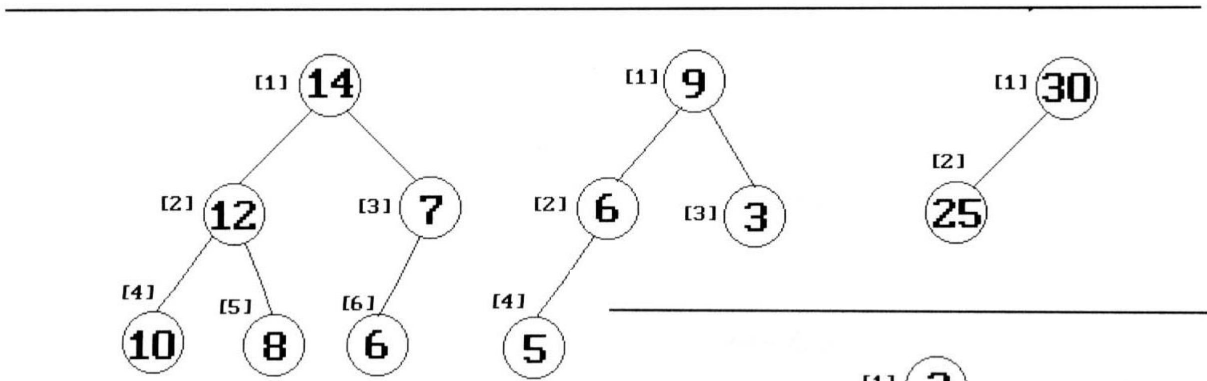


Figure 5.25: Sample max heaps

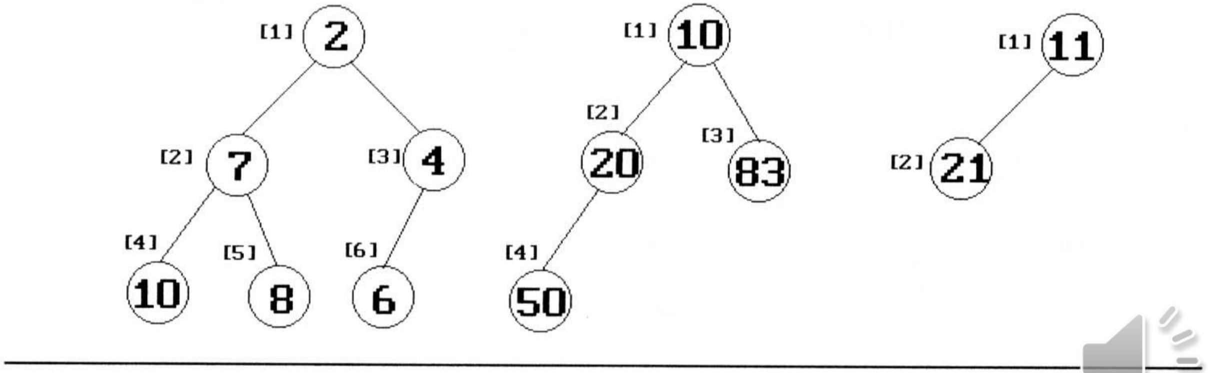


Figure 5.26: Sample min heaps

Heaps (3/6)

- Abstract data type of Max Heap

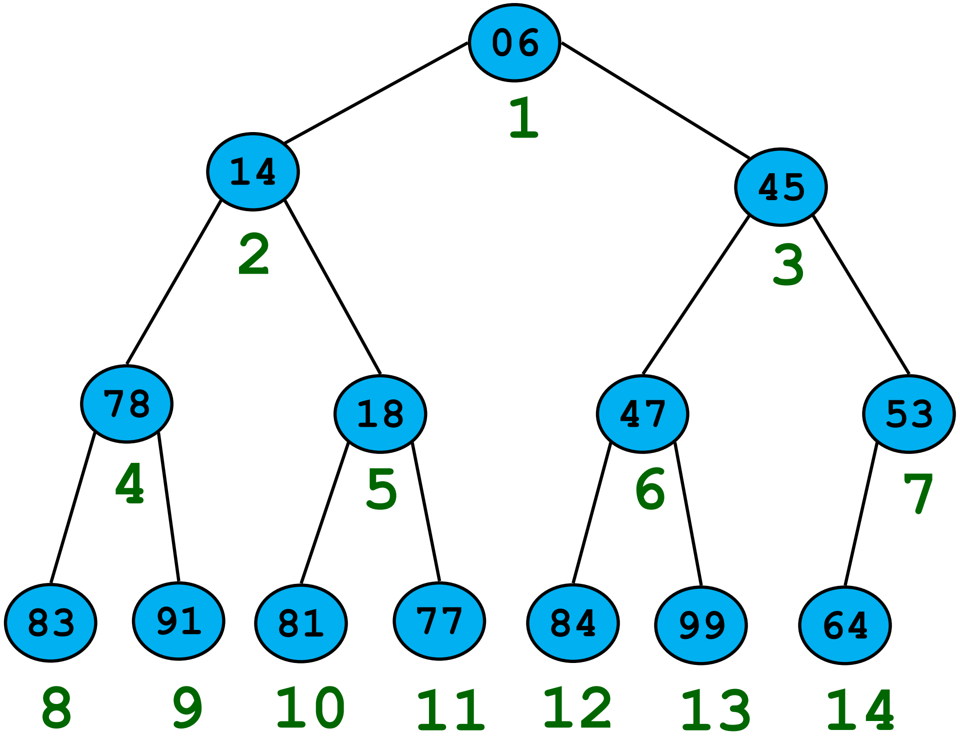
```
ADT Heap

Data : n>0 원소로 구성된 완전 이진 트리로 각 노드의 키값은 그의
               자식 노드의 키값보다 작지 않다.

Operators :
     H∈Heap; e ∈Element;
     createHeap() := create an empty heap;
     insertHeap(H,e) := insert a new item e into H
     isEmpty(H) := if the heap H is empty
                        then return true
                        else return false
     deleteHeap(H) := if isEmpty(H) then null
                         else {
                                e ← the largest element in H;
                                remove the largest element in H;
                                return e;
                         }

End Heap
```

Heaps (4/6)

- Queue : FIFO
- Priority queues
  - Heaps are frequently used to implement *priority queues (히프는 우선순위큐 구현에 최적)*
  - delete the element with highest (lowest) priority
  - insert the element with arbitrary priority
  - Heaps is the only way to implement priority queue(우선순위큐)

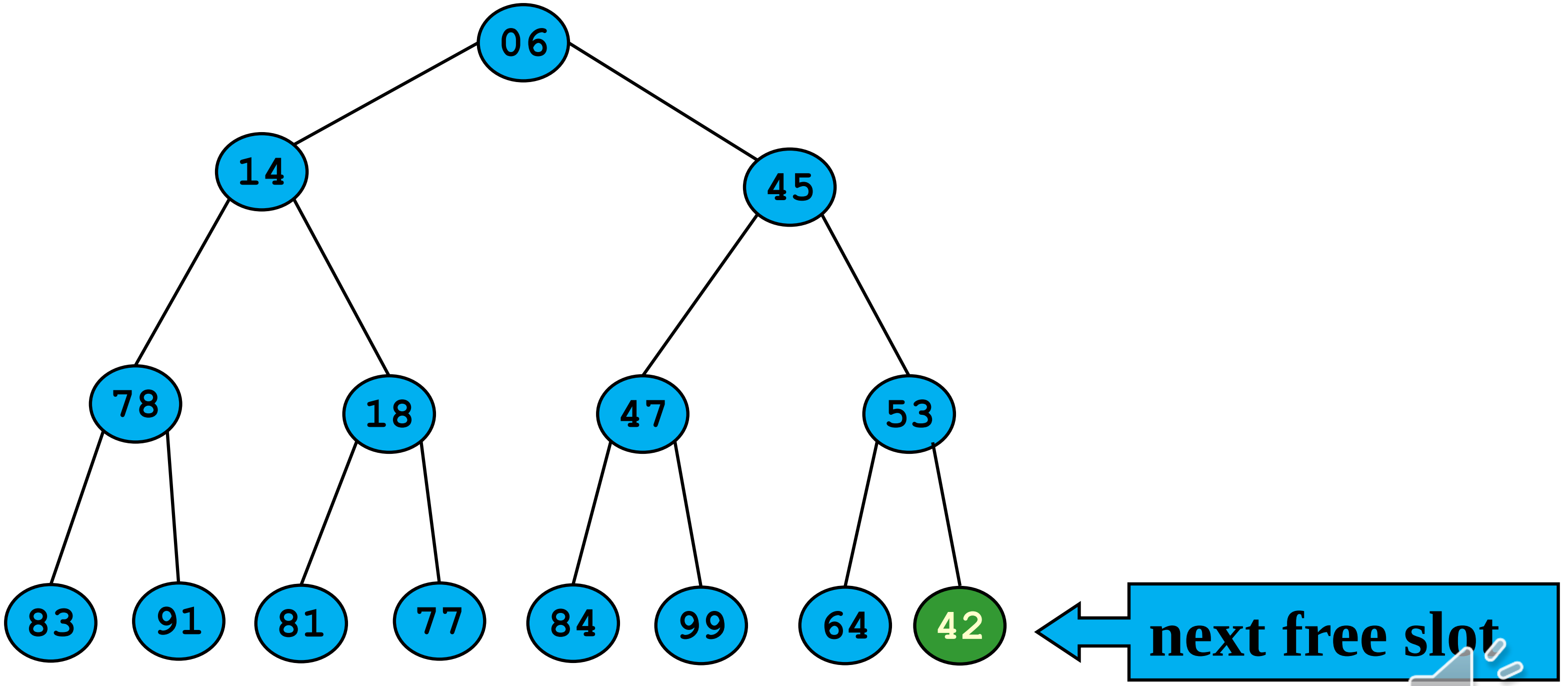
Binary Heaps: Array Implementation

- Implementing binary heaps.
  - Use an array: no need for explicit parent or child pointers.
    - `Parent(i)  = ⌊i/2⌋`
    - `Left(i)    = 2i`
    - `Right(i)   = 2i + 1`



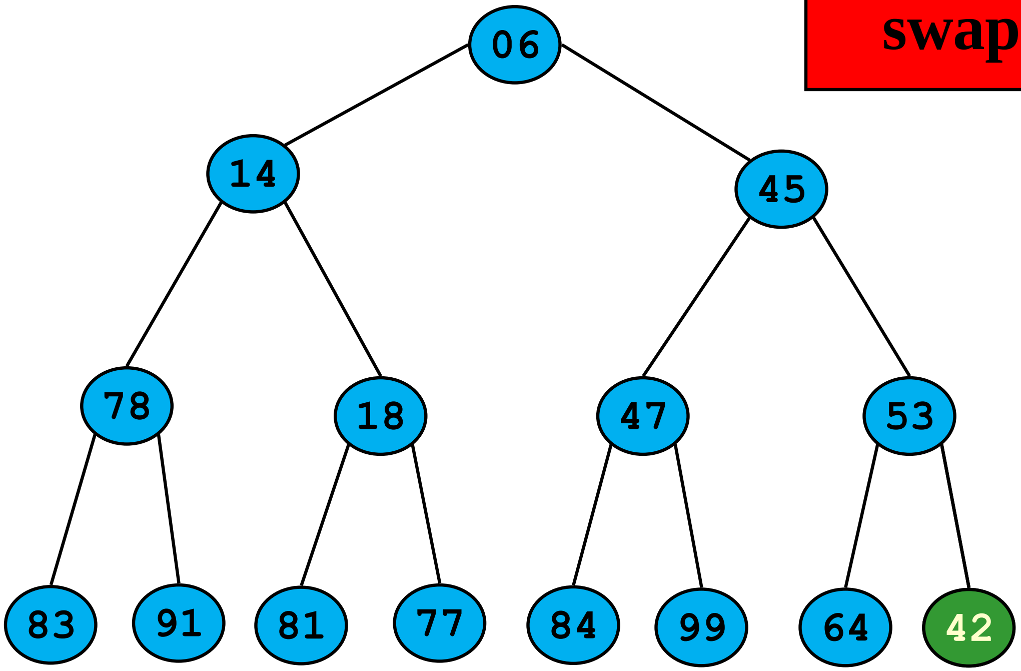
Binary Heap: Insertion(노드삽입)

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence



Binary Heap: Insertion

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence

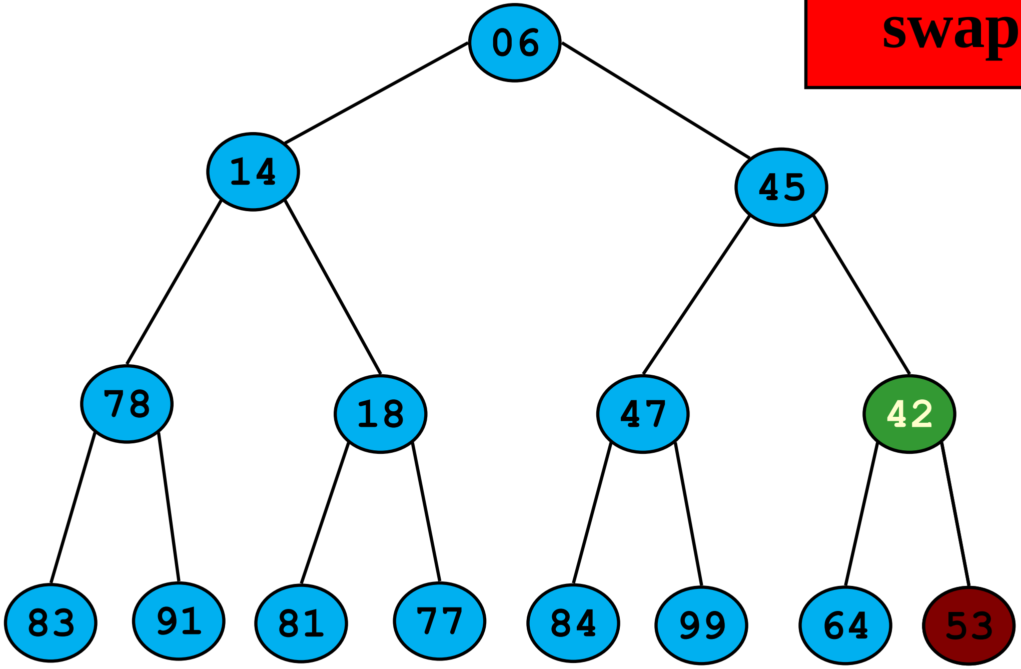


swap with parent

Binary Heap: Insertion

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence

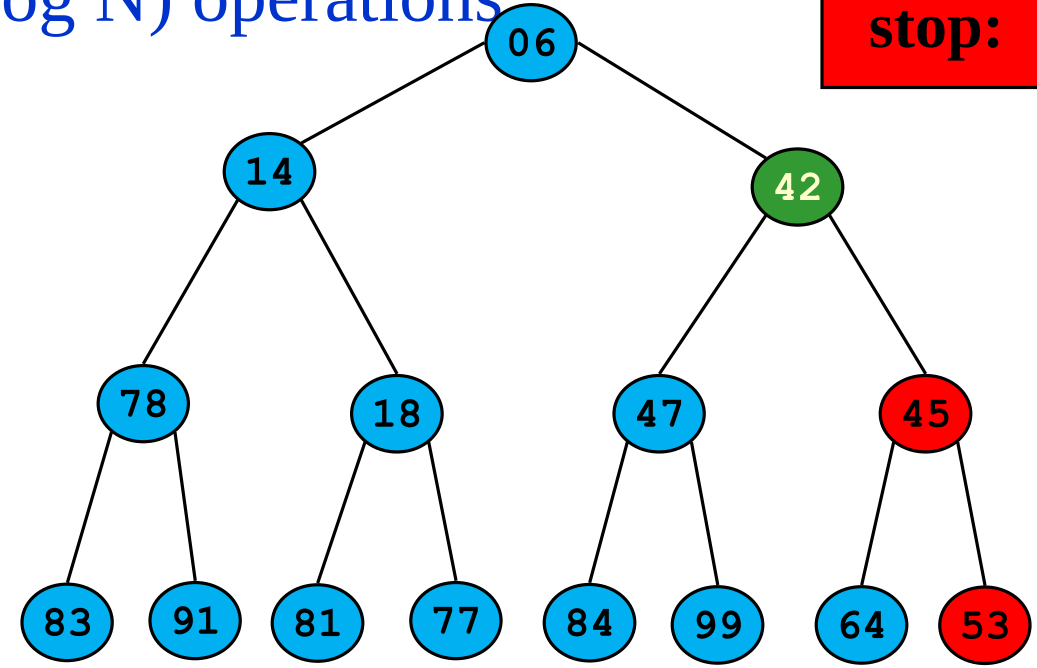
swap with parent



Binary Heap: Insertion

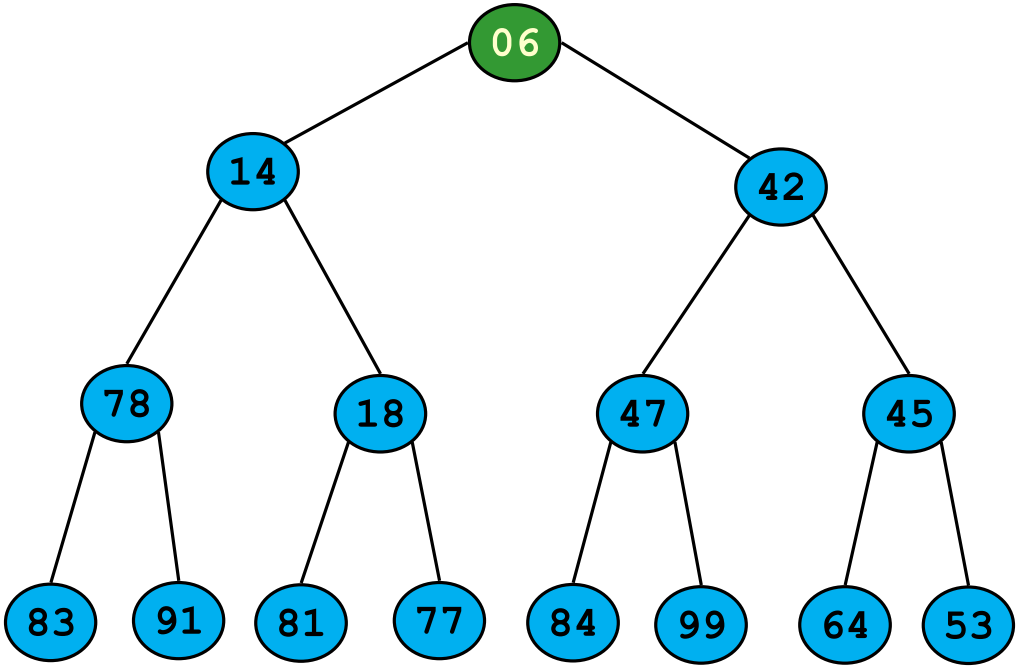
- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
    - Peter principle: nodes rise to level of incompetence
  - O(log N) operations

stop: heap ordered



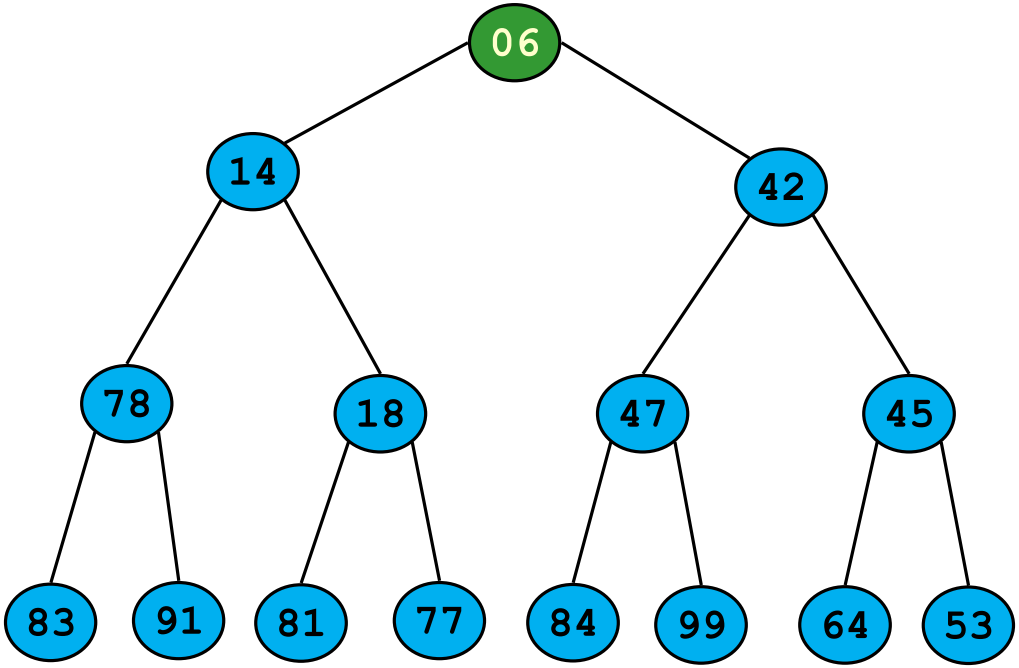
Binary Heap: Delete Min

- 히프에서 삭제는 루트만을 삭제함.



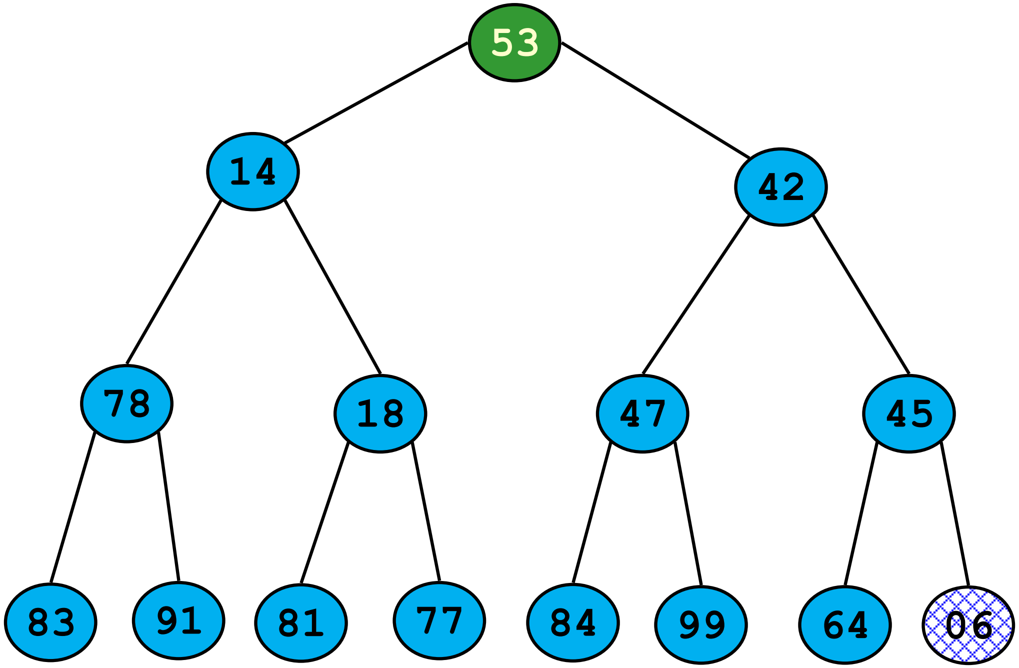
Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

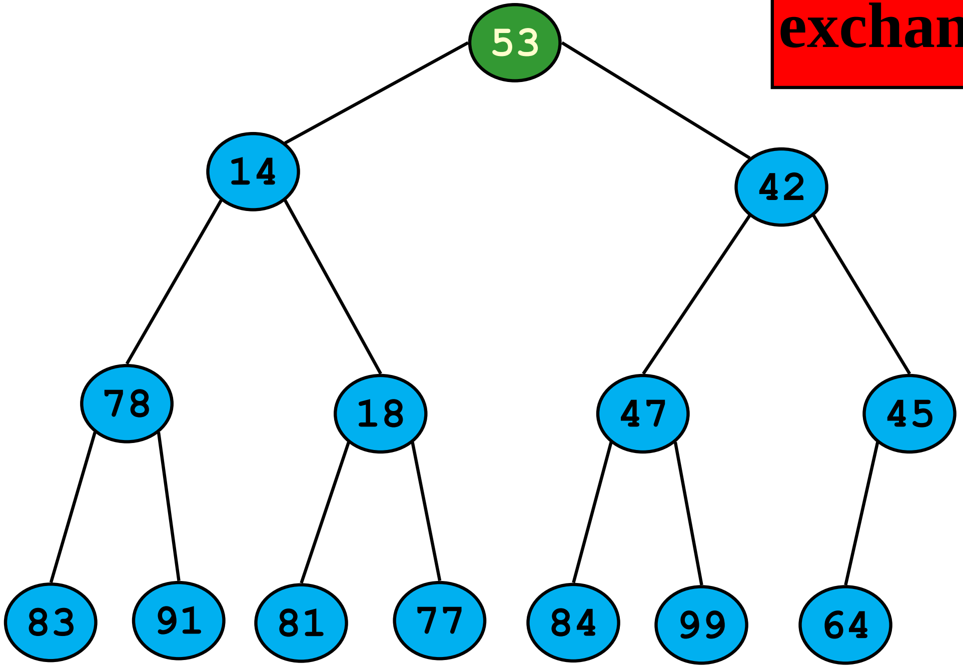
- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted

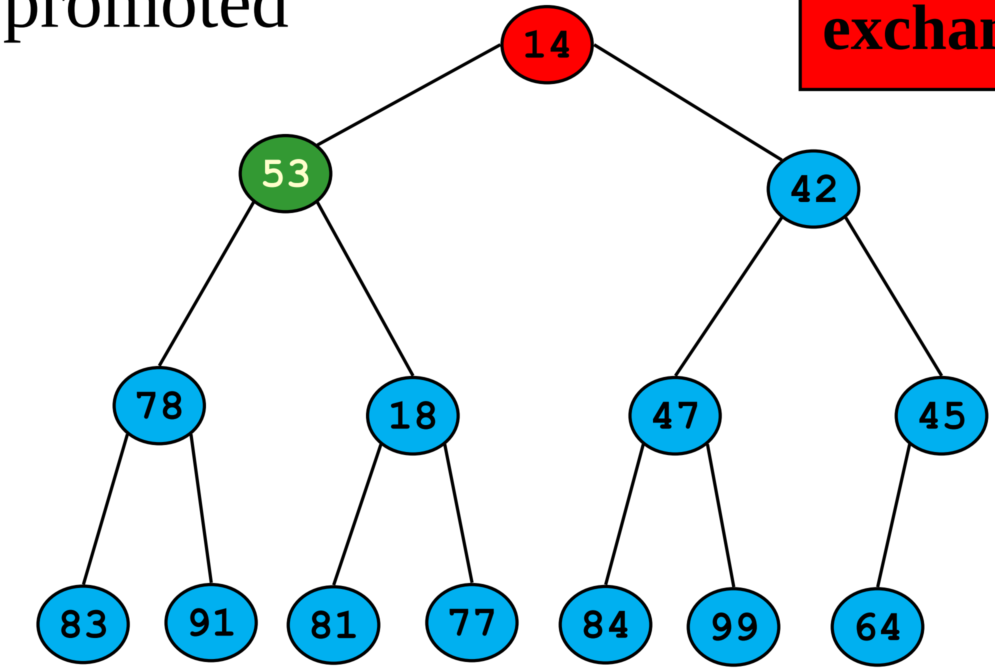
exchange with left child



Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted

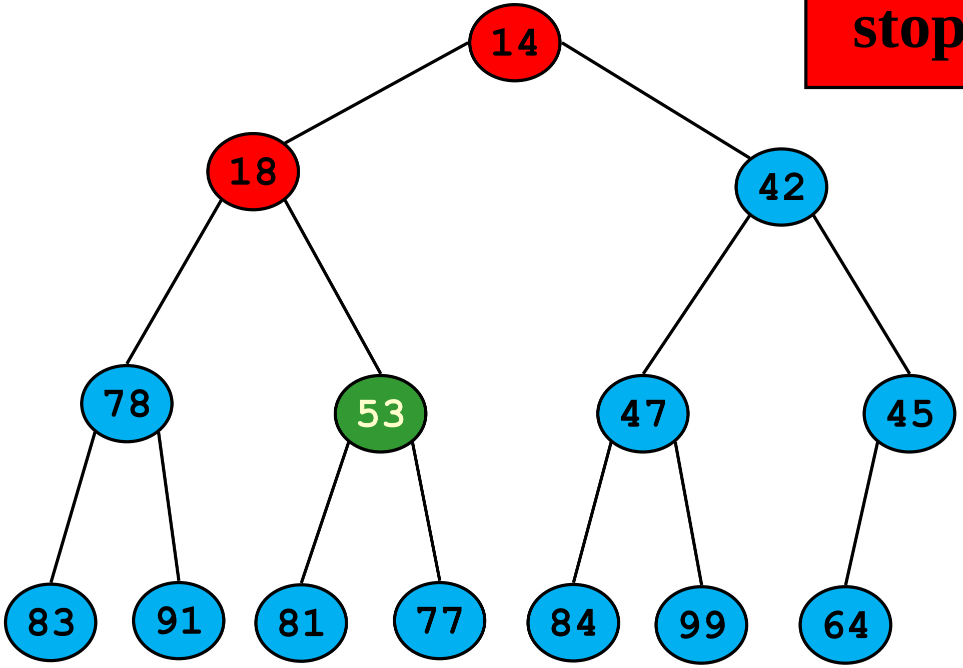
exchange with right child



Binary Heap: Delete Min

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
    - power struggle principle: better subordinate is promoted
  - O(log N) operations.

stop: heap ordered



Insertion algorithm

```
insertHeap(Heap,e)
       // 순차 표현으로 구현된 최대 히프
       // 원소 e를 히프 Heap에 삽입, n은 현재 히프의 크기(원소 수)
   if (n = maxSize) then heapFull;   // 히프가 만원이면 히프 크기를 확장
   n←n+1;   // 새로 첨가될 노드 위치
   for (i←n; ; ) do {
      if (i = 1) then exit;   // 루트에 삽입
      if(e.key ≤ Heap[  i/2  ].key) then exit;   // 삽입할 노드의 키값과
                                  // 부모 노드 키값을 비교
      Heap[i] ← Heap[  i/2  ];   // 부모 노드 키값을 자식노드로 이동
      i ←   i/2  ;
   }
   Heap[i] ← e;
end insertHeap()
```

Deletion algorithm

```
deleteHeap(heap)
            // 히프로부터 원소 삭제, n은 현재의 히프 크기(원소 수)
    if (n=0) then return error;   // 공백 히프
    item ← heap[1];    // 삭제할 원소
    temp ← heap[n];    // 이동시킬 원소
    n ← n-1;        // 히프 크기(원소 수)를 하나 감소
    i ← 1;
    j ← 2;   // j는 i의 왼쪽 자식 노드
    while (j ≤ n) do {
        if (j < n) then if (heap[j] < heap[j+1])
            then j ← j+1;   // j는 값이 큰 자식을 가리킨다.
        if (temp ≥ heap[j]) then exit;
        heap[i] ← heap[j];   // 자식을 한 레벨 위로 이동
        i ← j;
        j ← j*2; // i와 j를 한 레벨 아래로 이동
    }
    heap[i] ← temp;
    return item;
end deleteHeap()
```

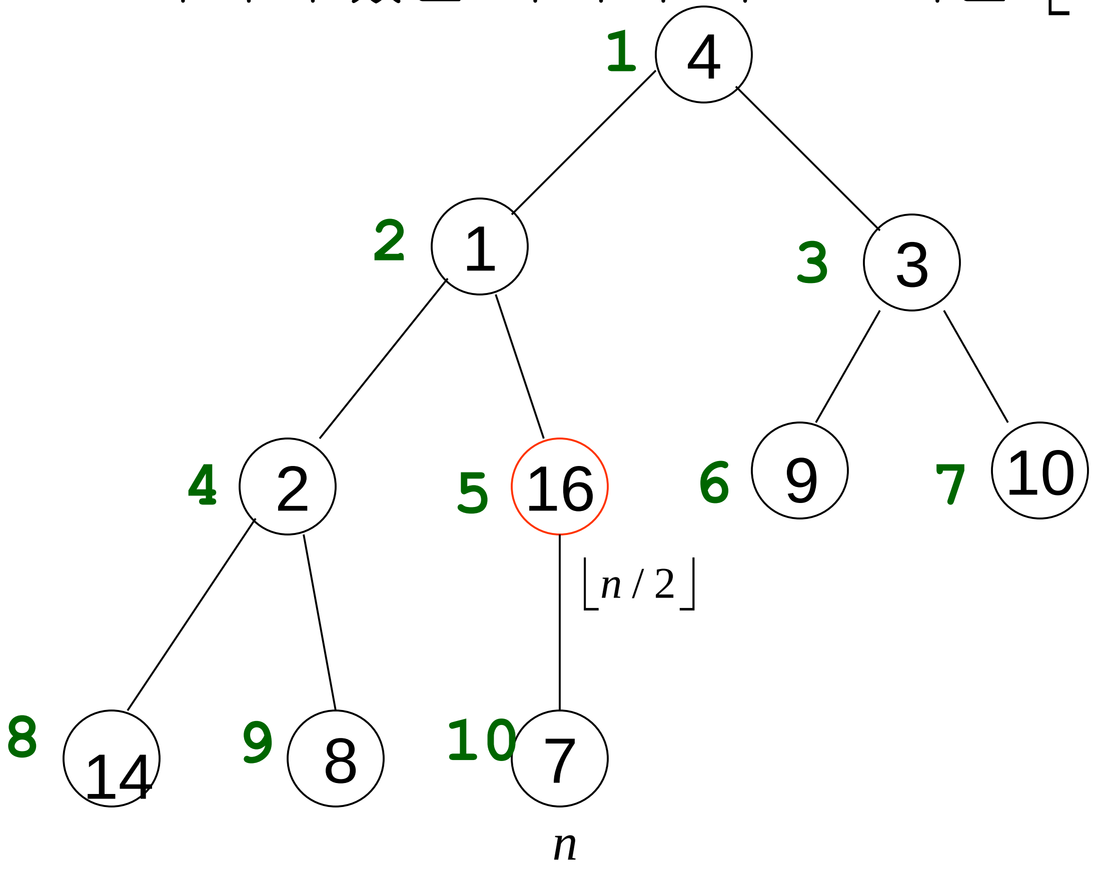
Building a Heap(히프만들기)

```
Build-Max-Heap(A)
heap-size[A] ← length[A];
for i ← ⌊length[A]/2⌋ downto 1
    do Max-Heapify(A, i);
```

e.g., 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

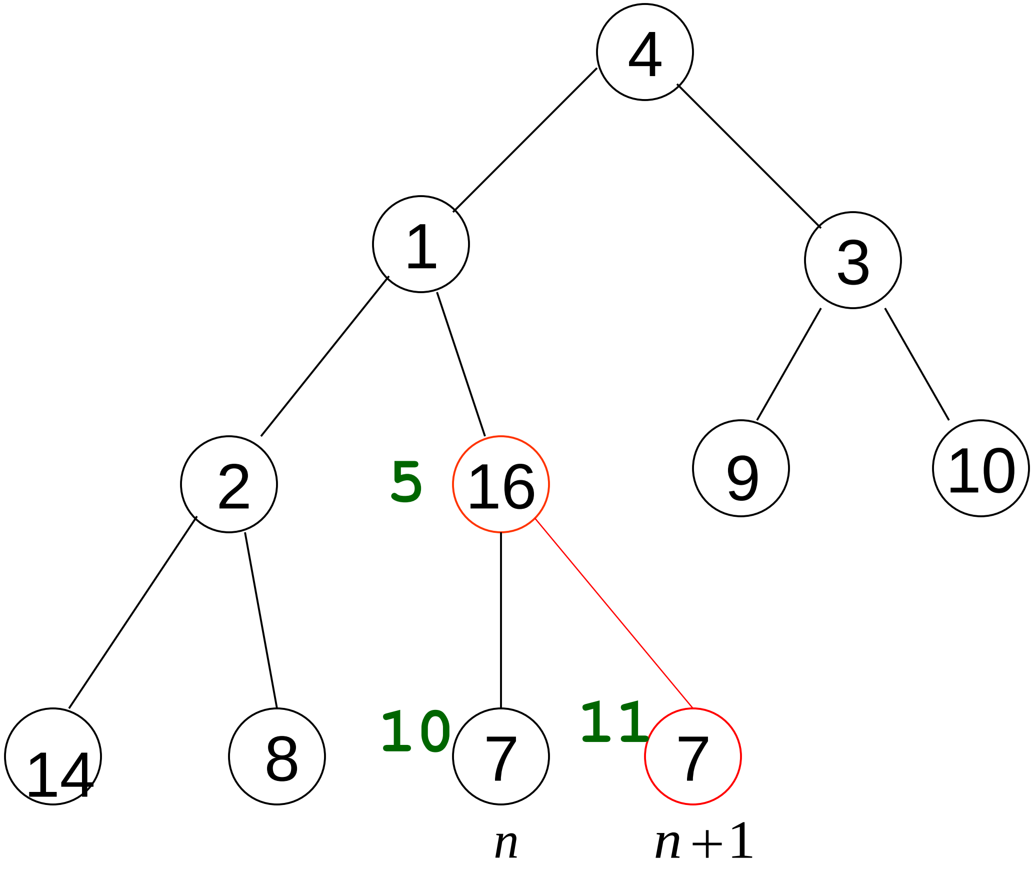
The last location who has a child is $\lfloor n/2 \rfloor$.

자식이 있는 마지막 서브트리는 $\lfloor n/2 \rfloor$에 위치

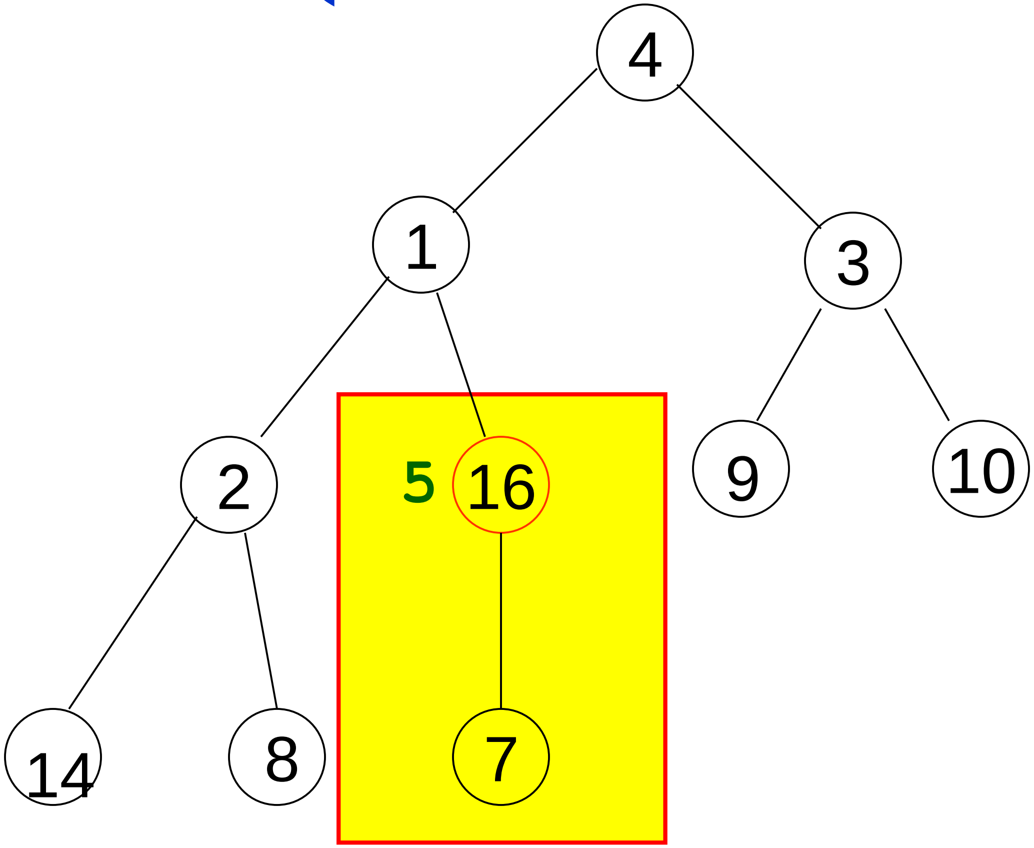


Case 1. n is even. $\lfloor (\mathrm{n}+1)/2 \rfloor = n/2$.

Case 2. n is odd. $\lfloor (\mathrm{n}+1)/2 \rfloor = \lfloor n/2 \rfloor + 1$.

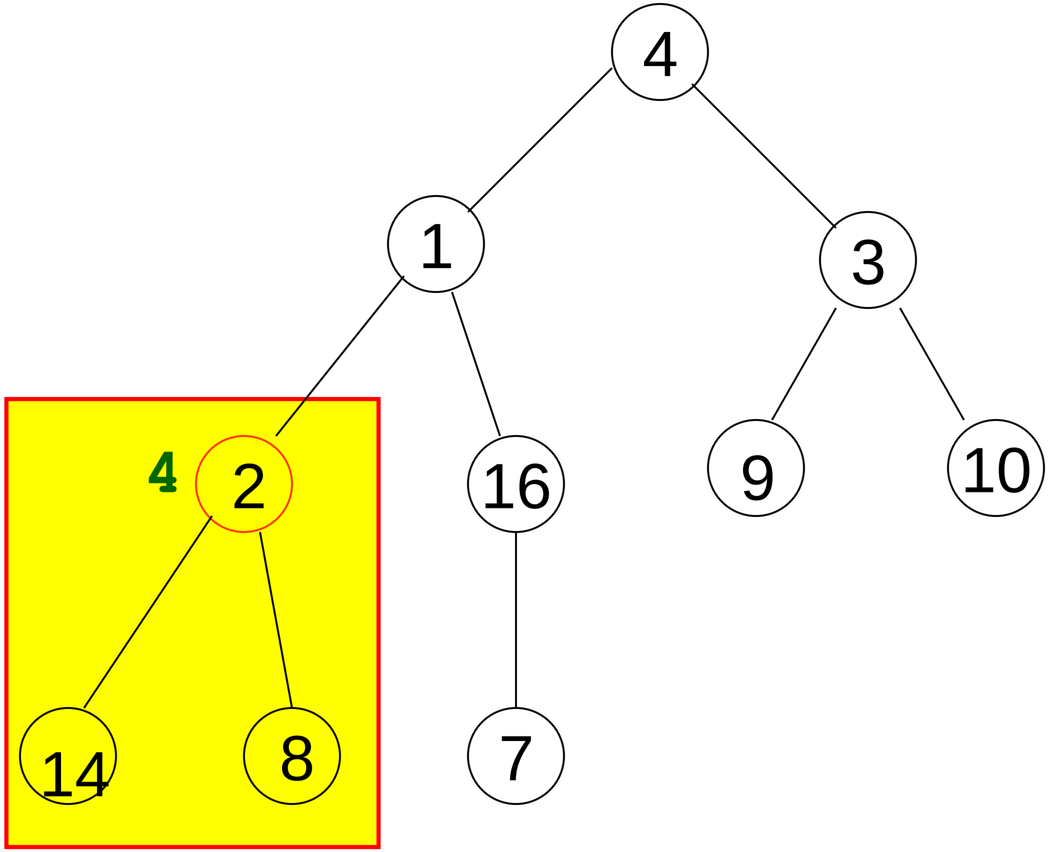


Max-Heapify(node5)
(최대히프만들기)



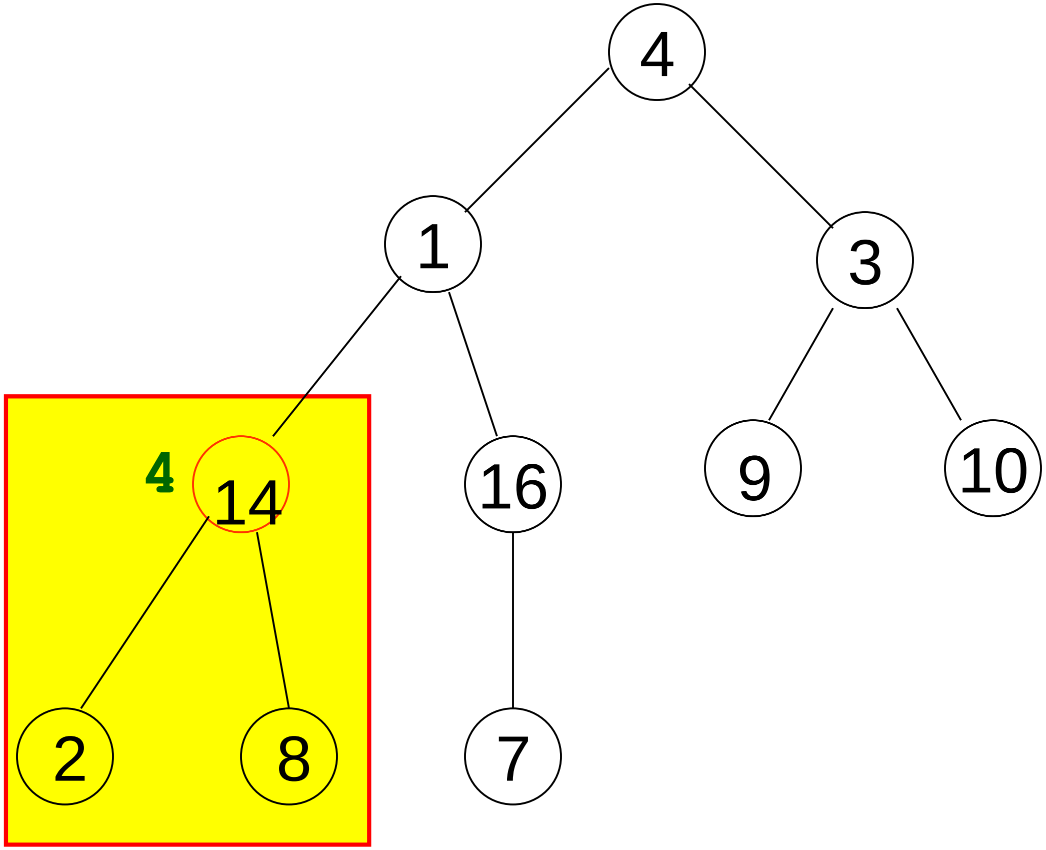
5를 루트로 하는 서브트리를 히프로 만든다

Max-Heapify(node 4)

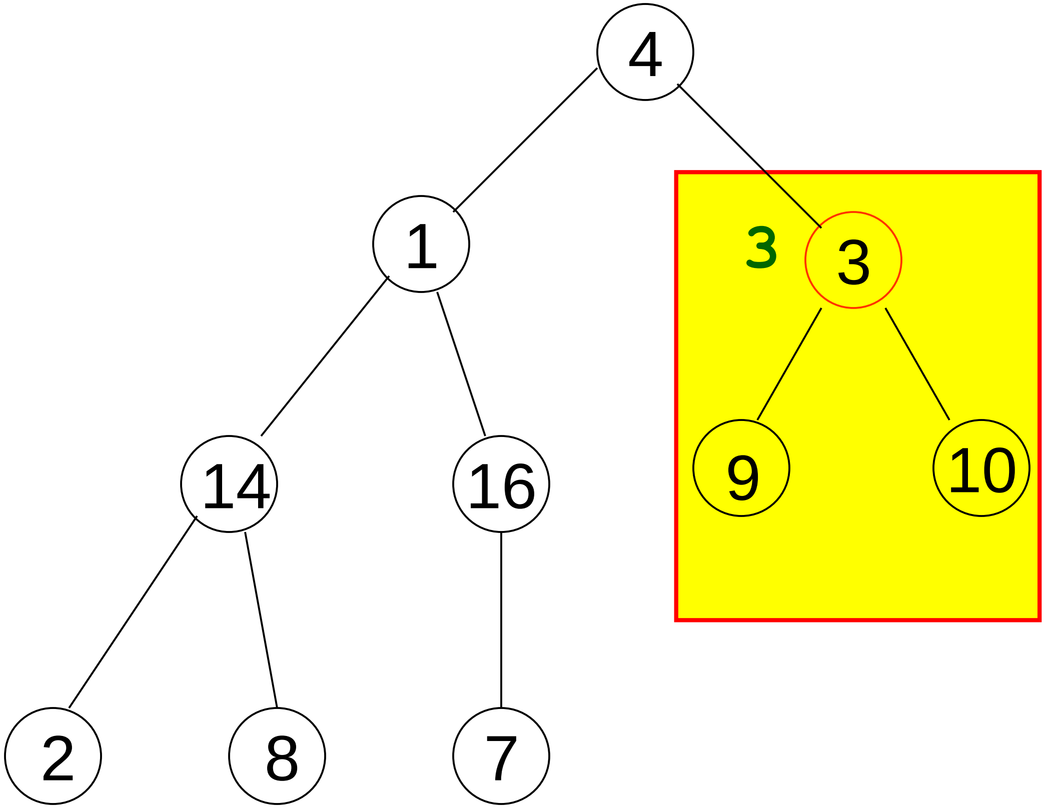


4를 루트로 하는 서브트리를 히프로 만든다

Max-Heapify(node4)

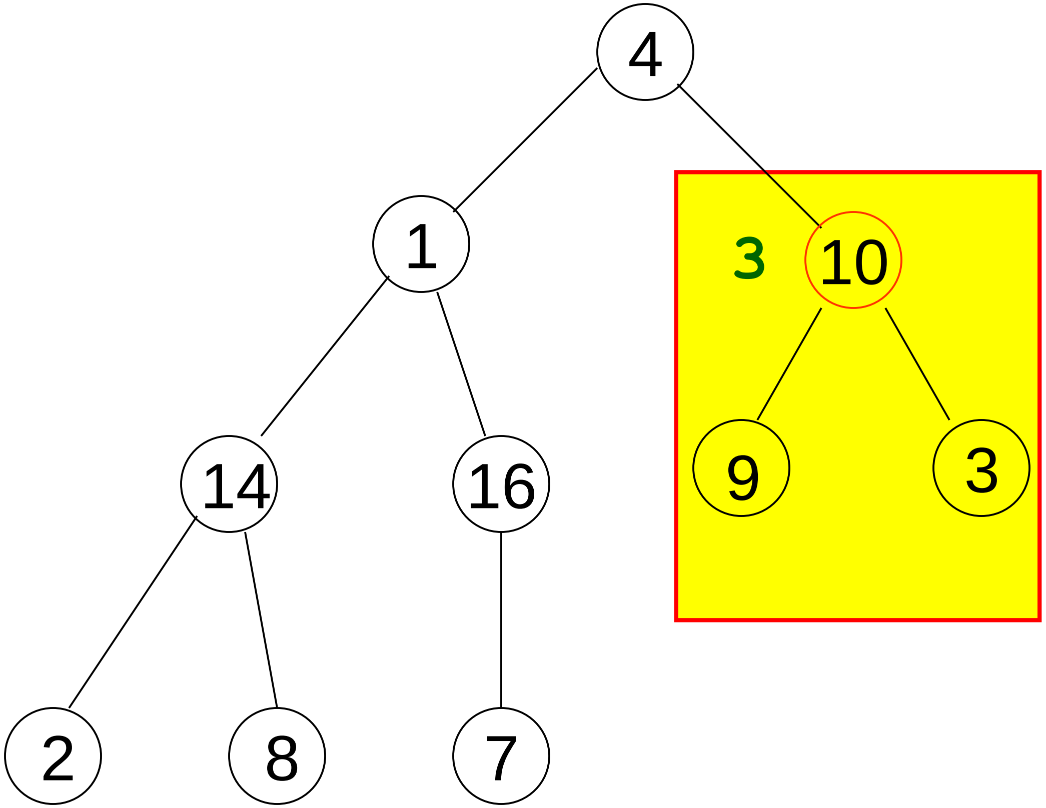


Max-Heapify(node3)

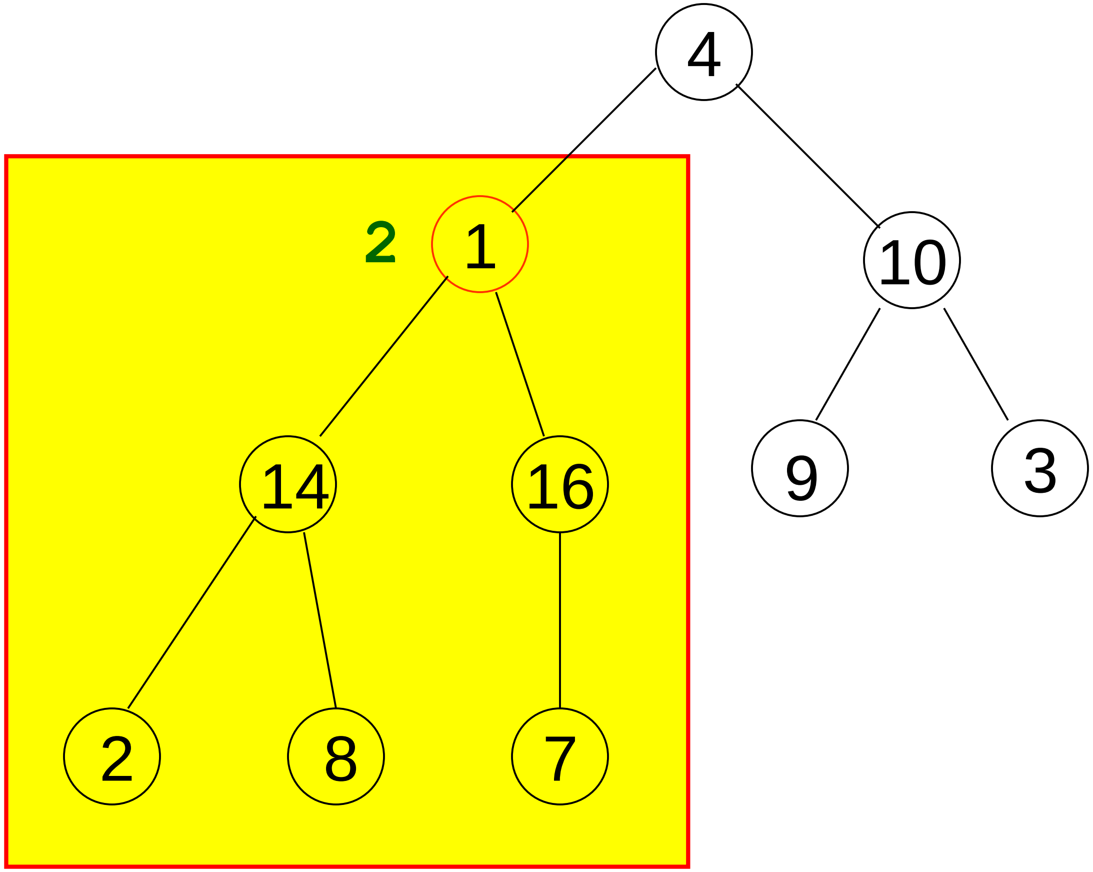


3을 루트로 하는 서브트리를 히프로 만든다

Max-Heapify(node3)

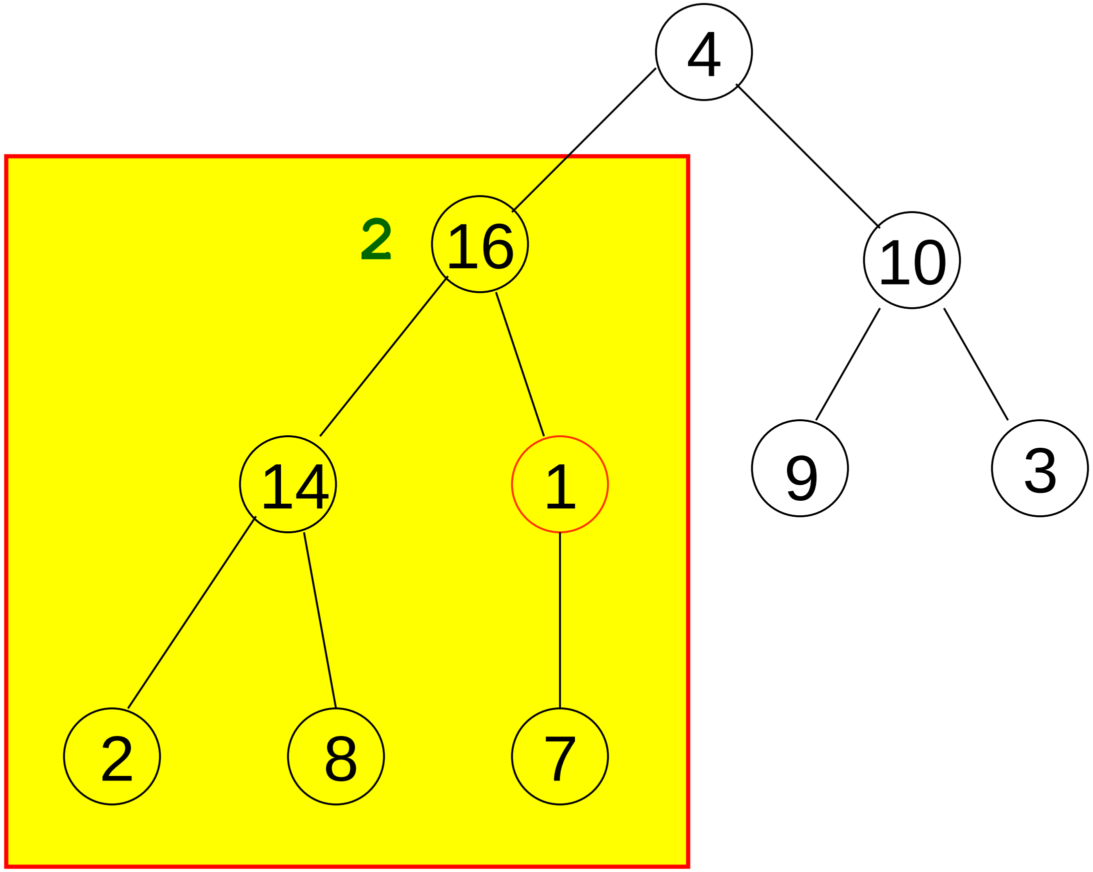


Max-Heapify(node2)

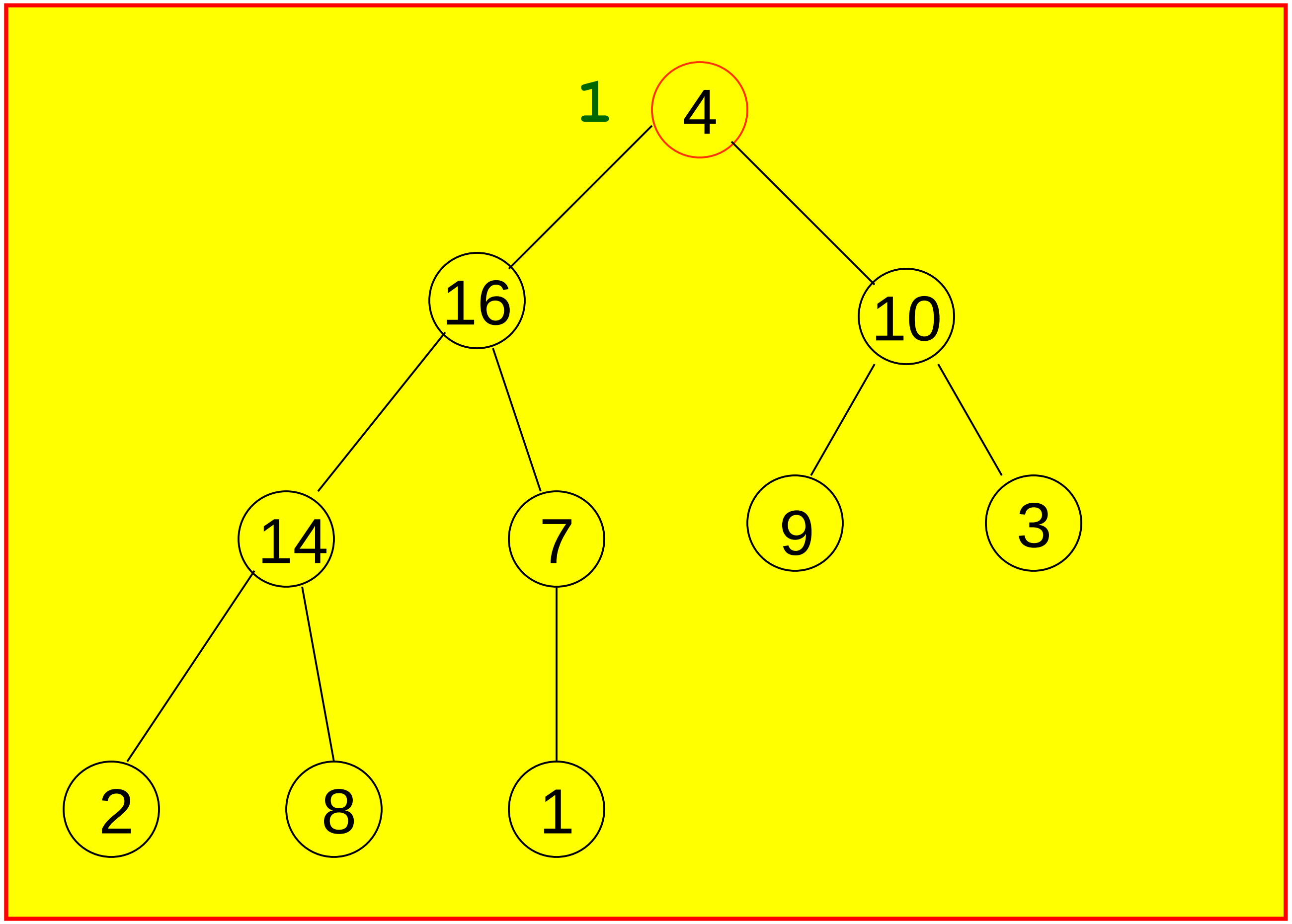


2를 루트로 하는 서브트리를 히프로 만든다

Max-Heapify(node2)

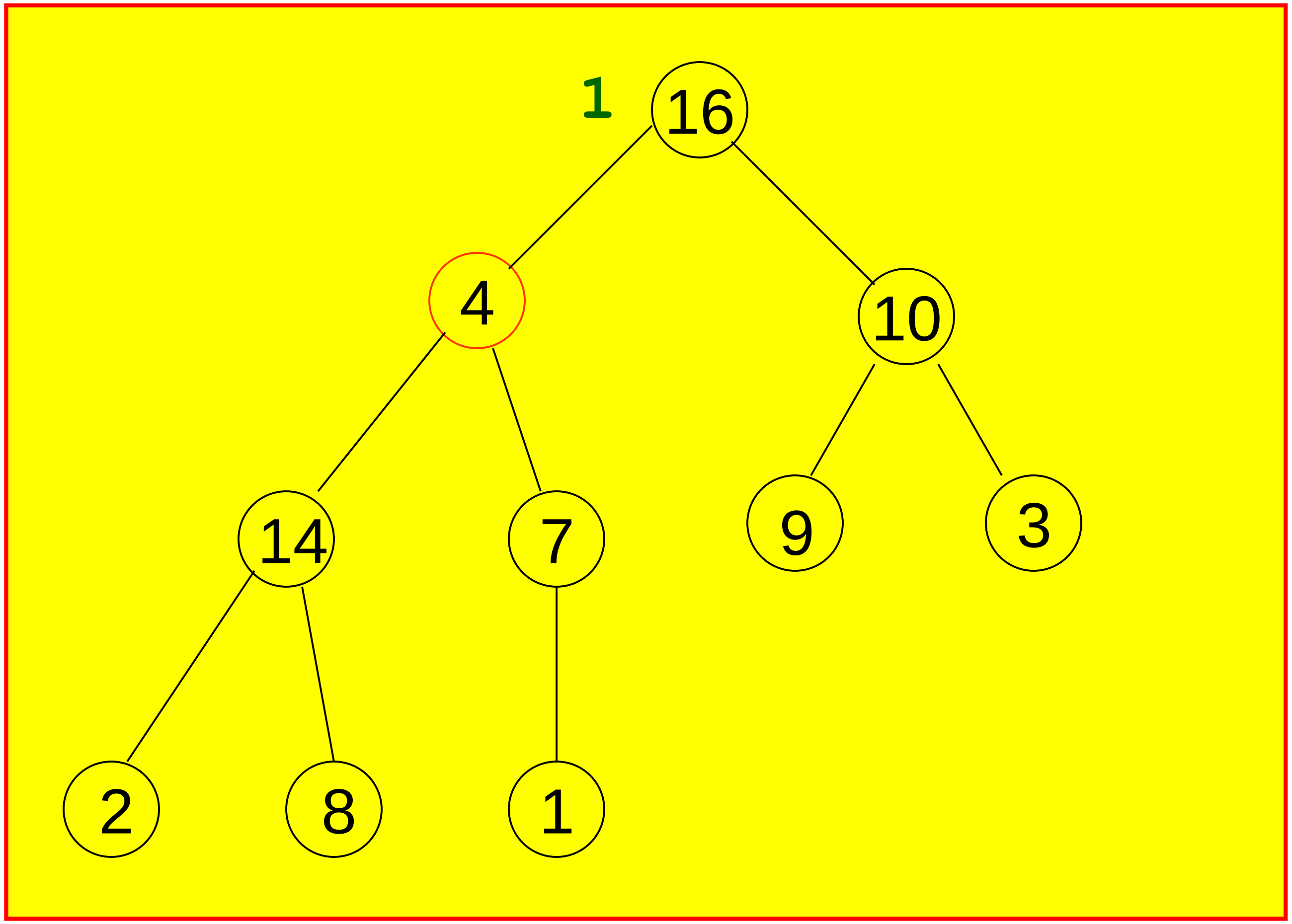


Max-Heapify(node1)

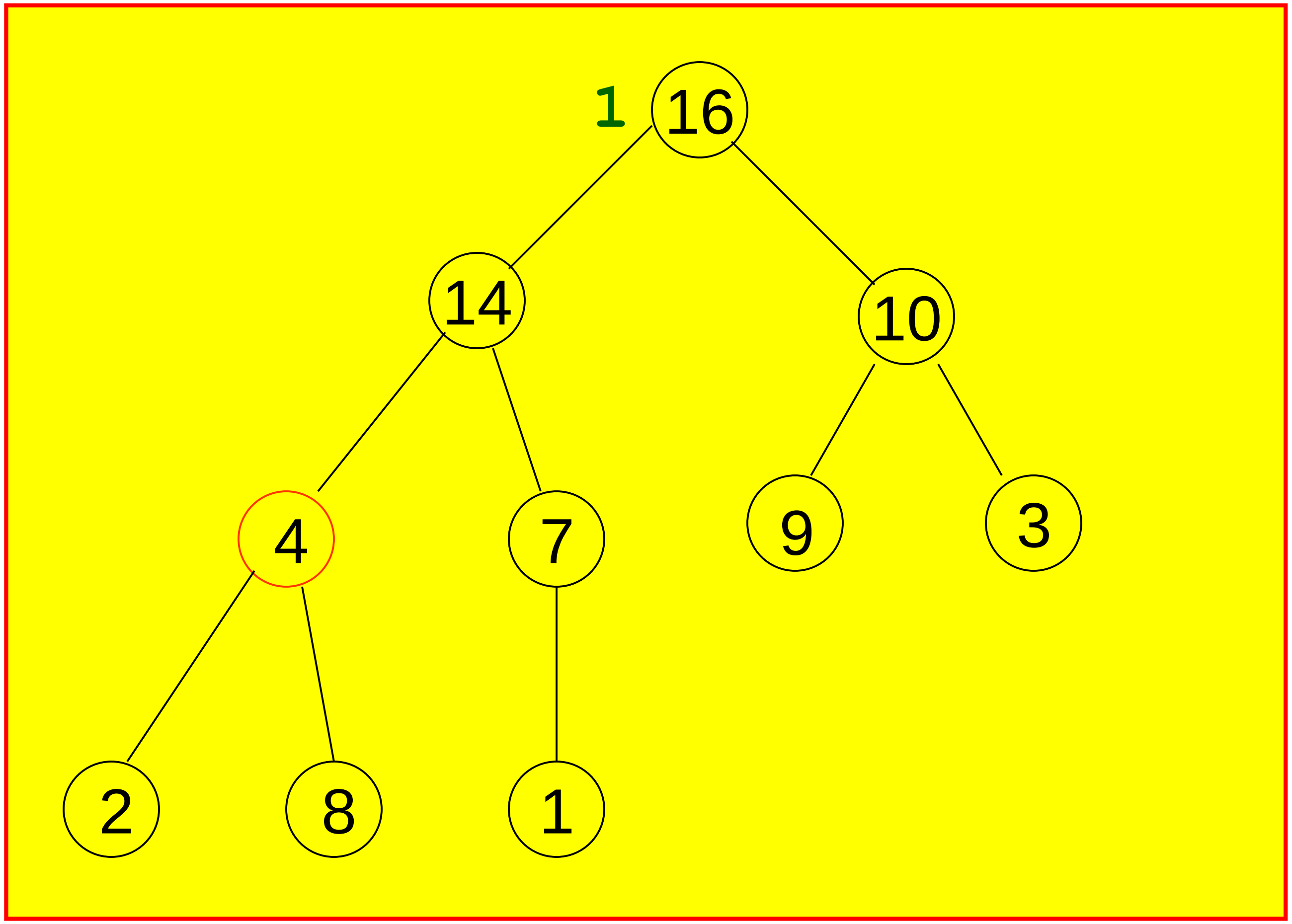


1을 루트로 하는 서브트리를 히프로 만든다

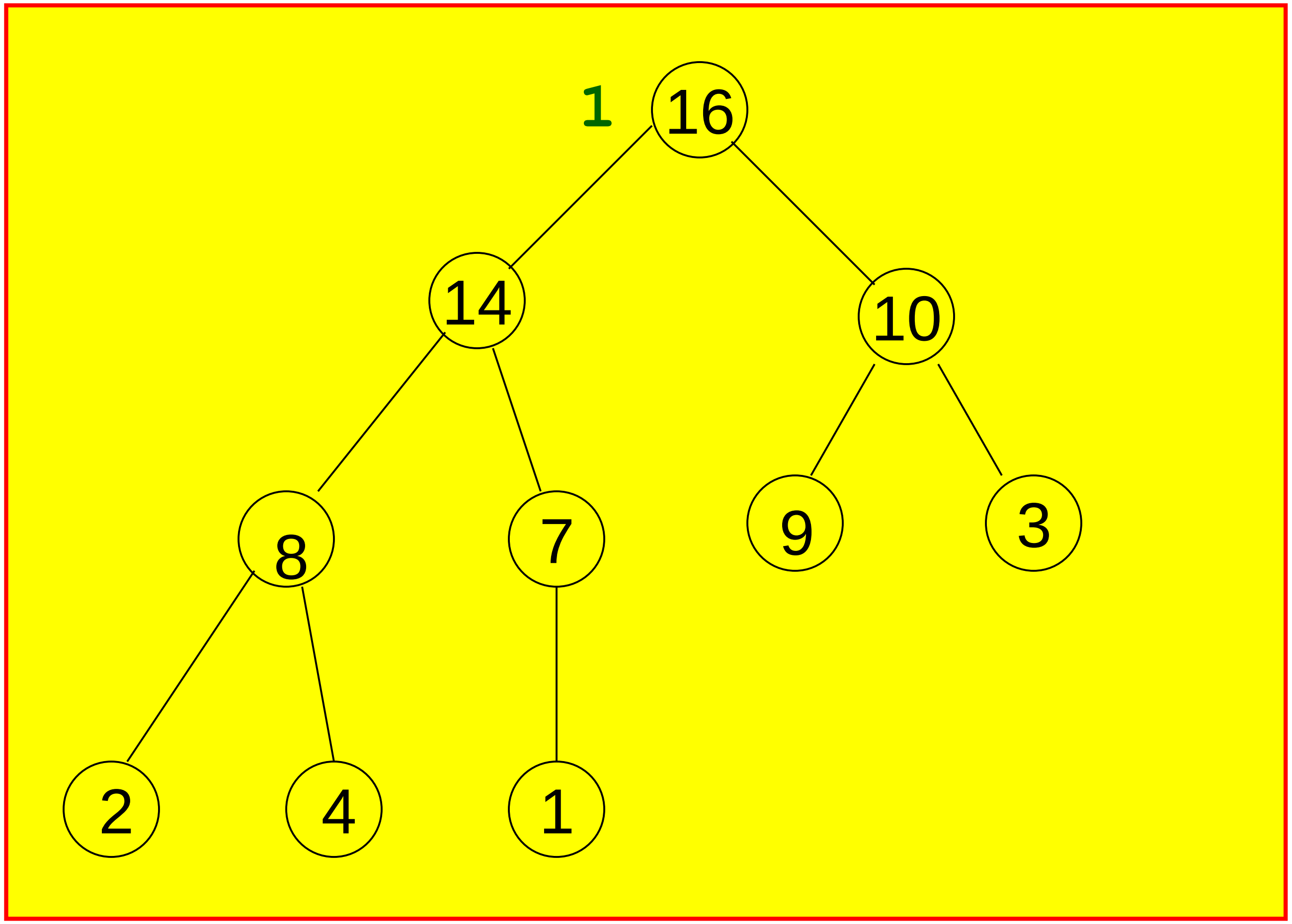
Max-Heapify(node1)



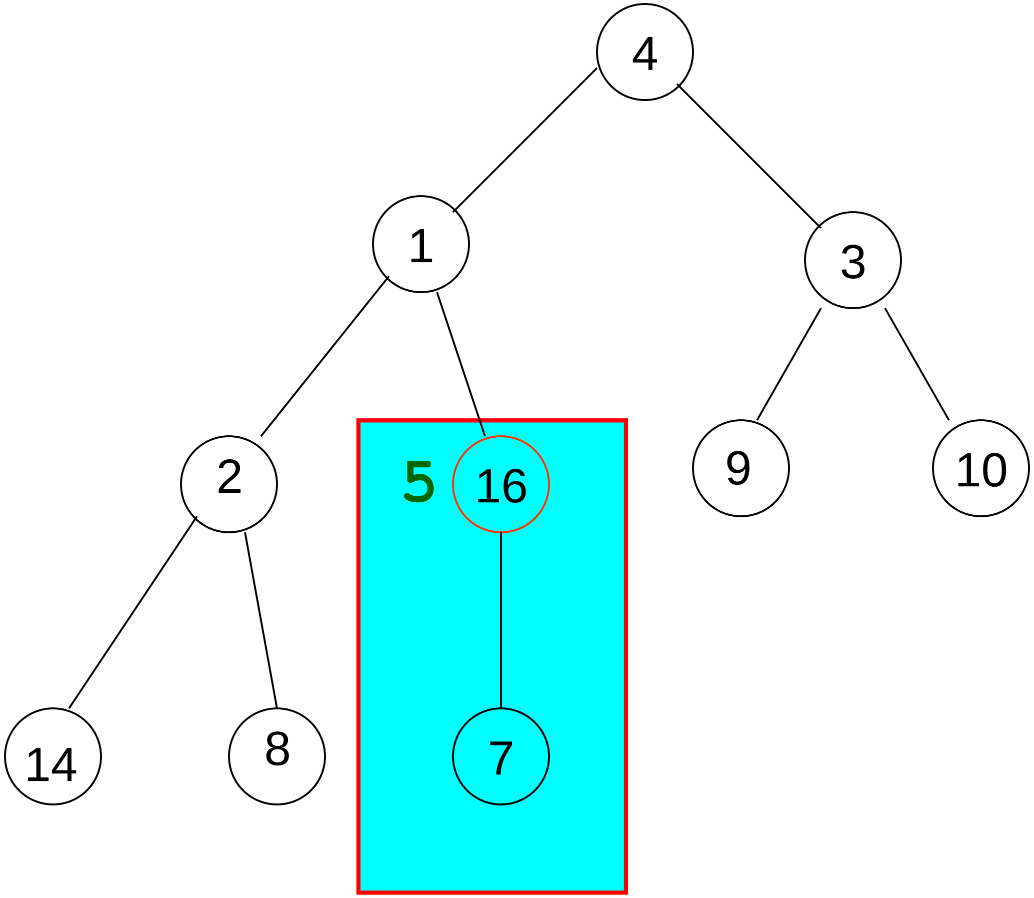
Max-Heapify(node1)



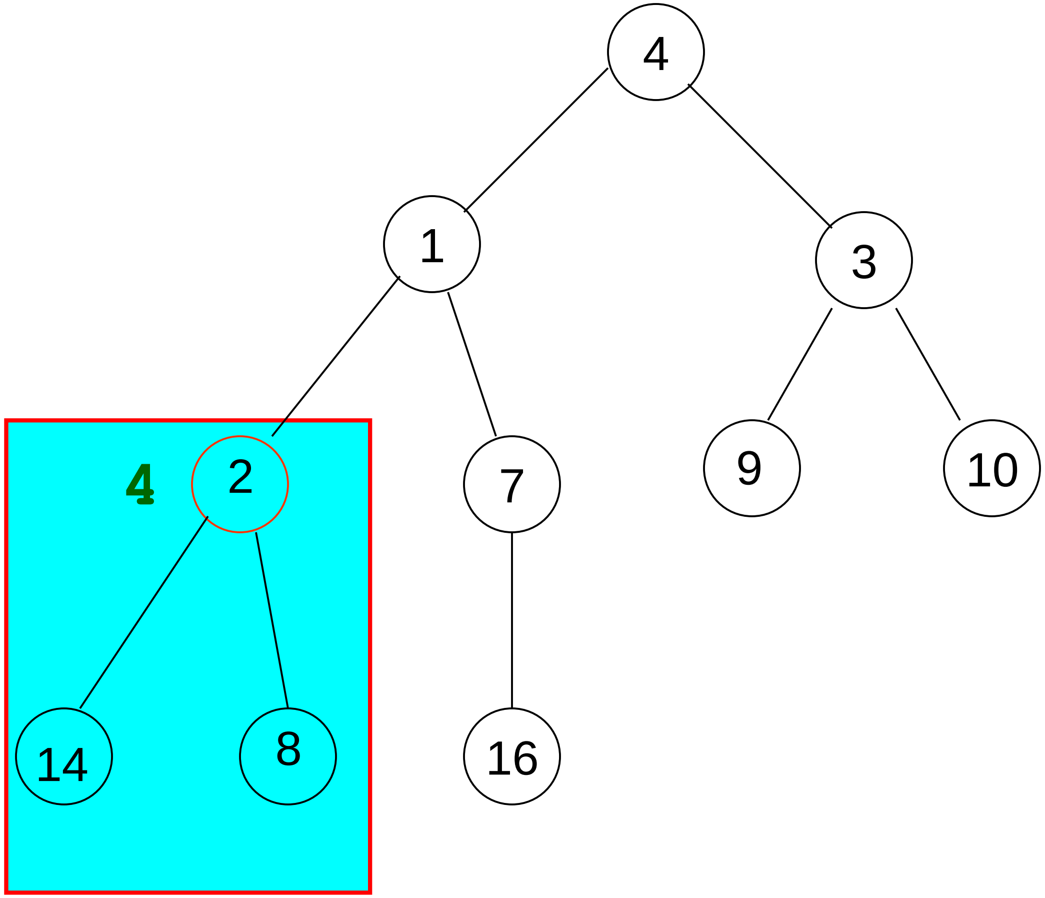
Max-Heapify(node1)



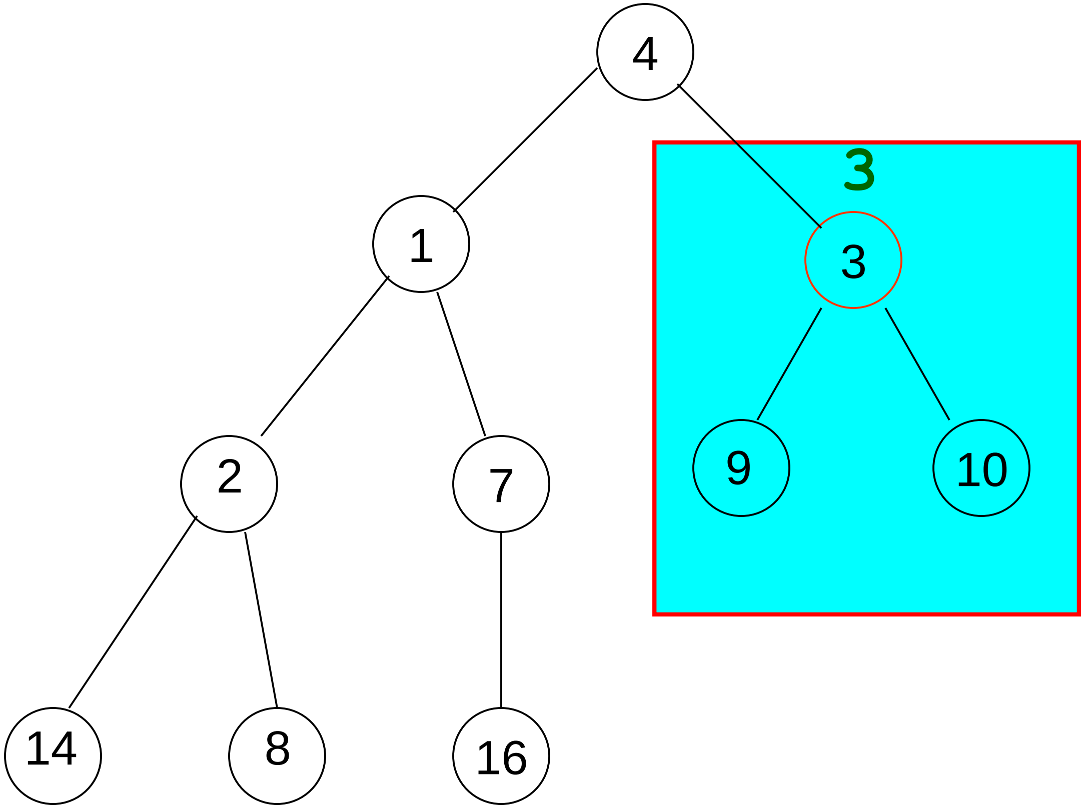
Min-Heapify(노드5)(최소히프만들기)



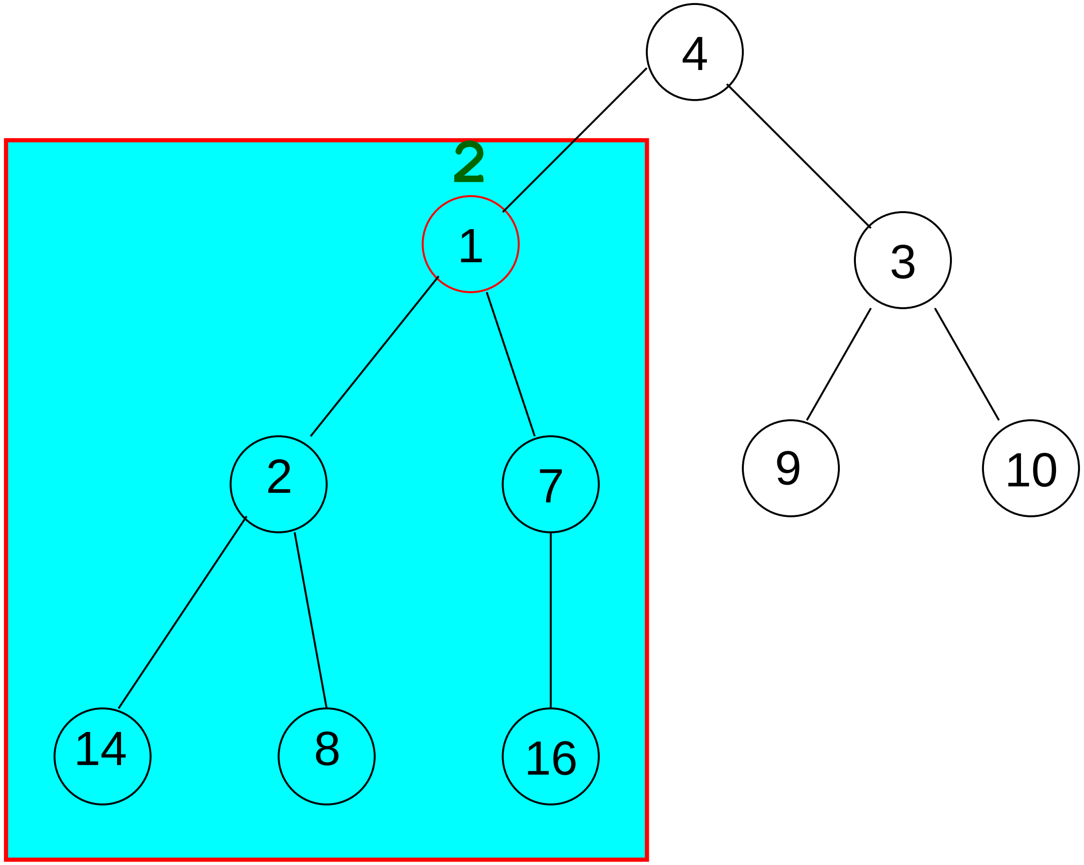
Min-Heapify(노드4)



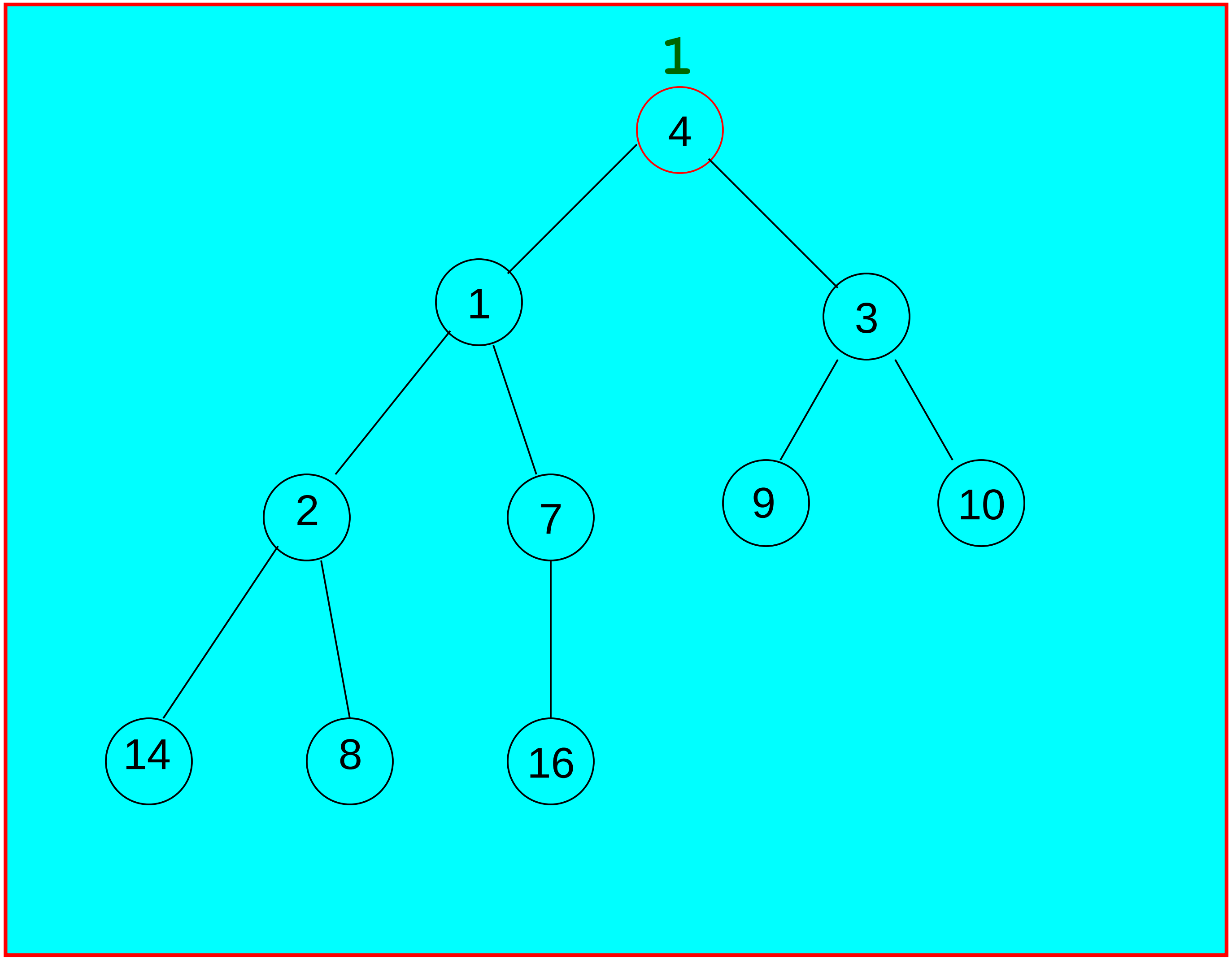
Min-Heapify(노드3)



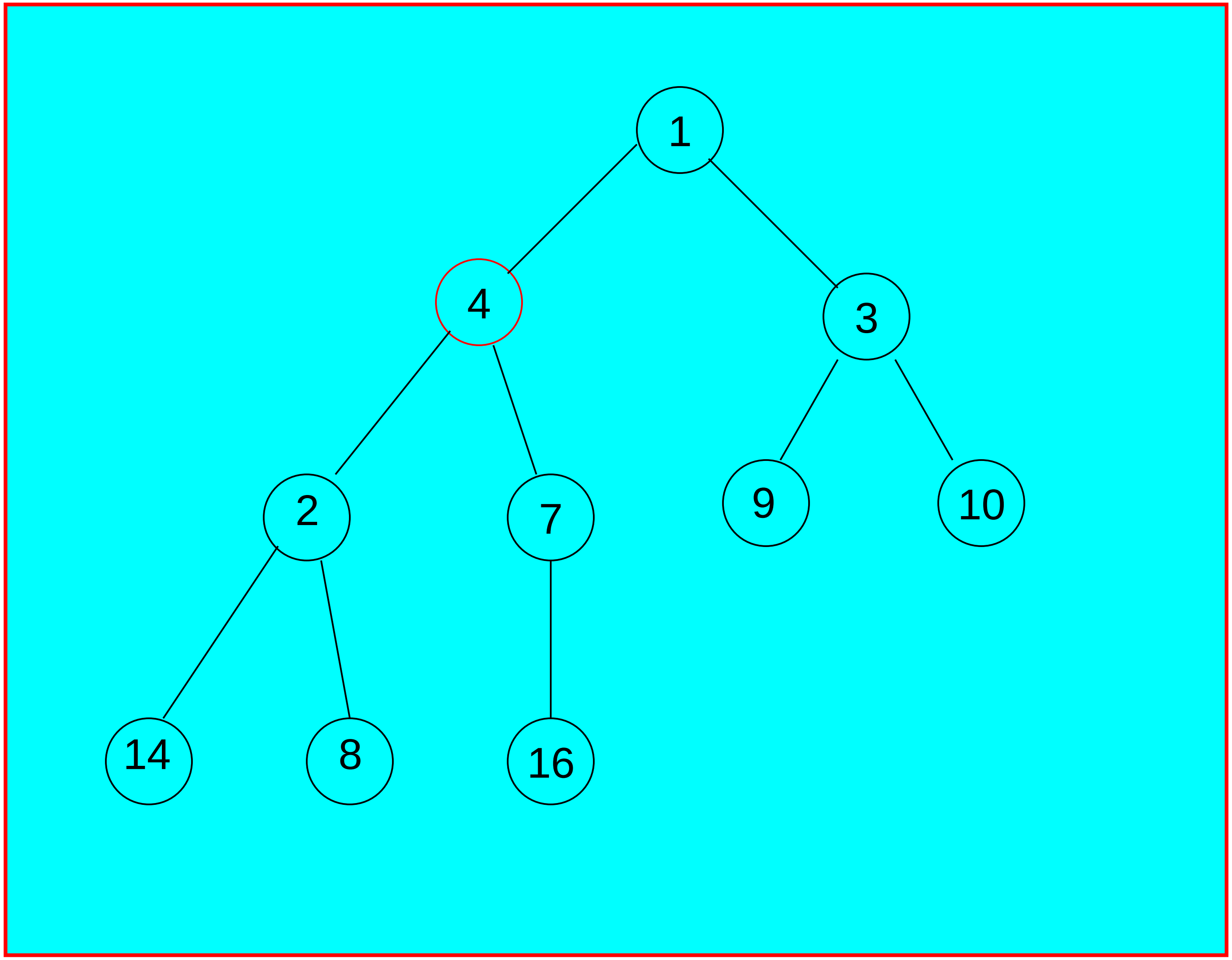
Min-Heapify(노드2)



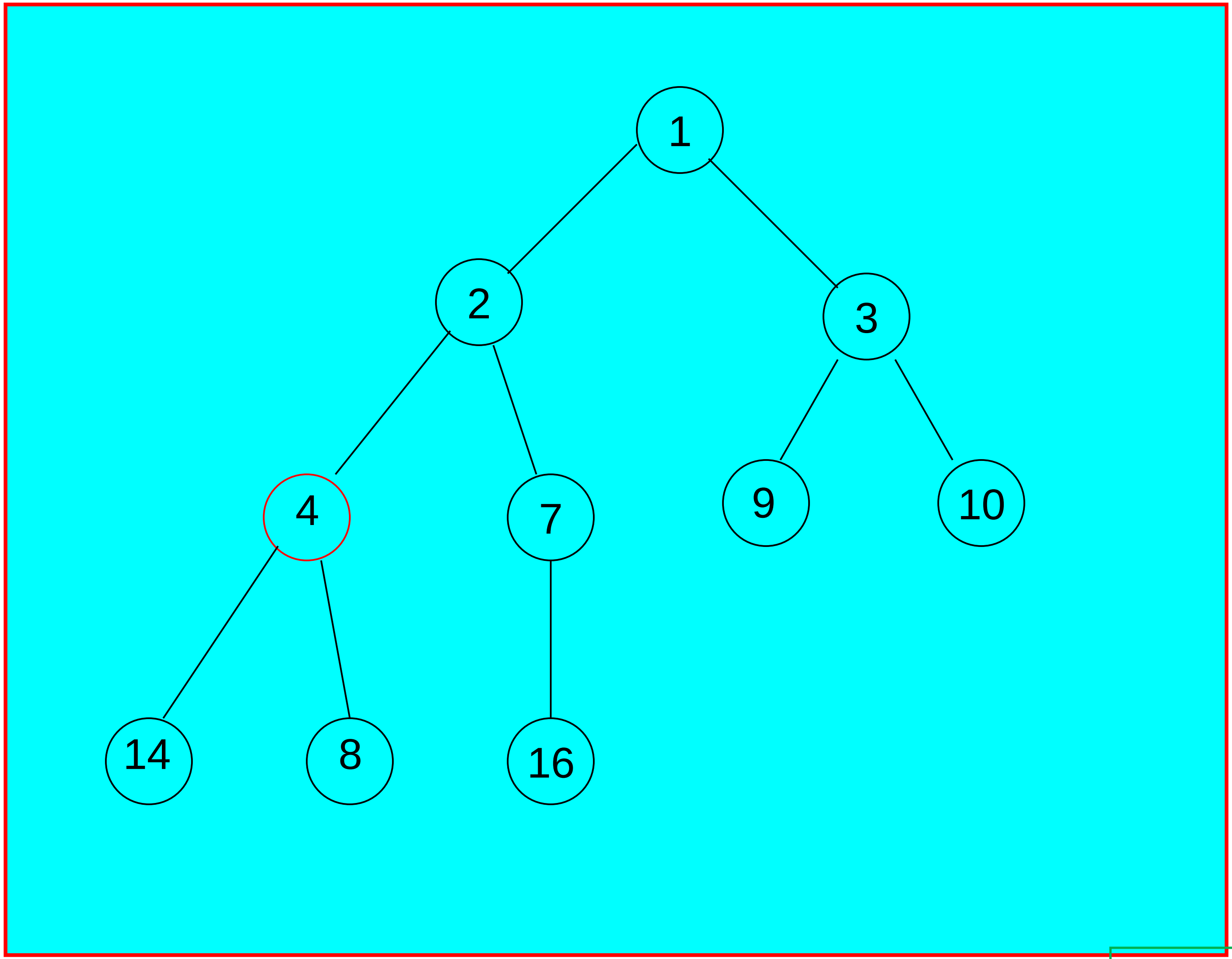
Min-Heapify(노드1)



Min-Heapify(노드1)



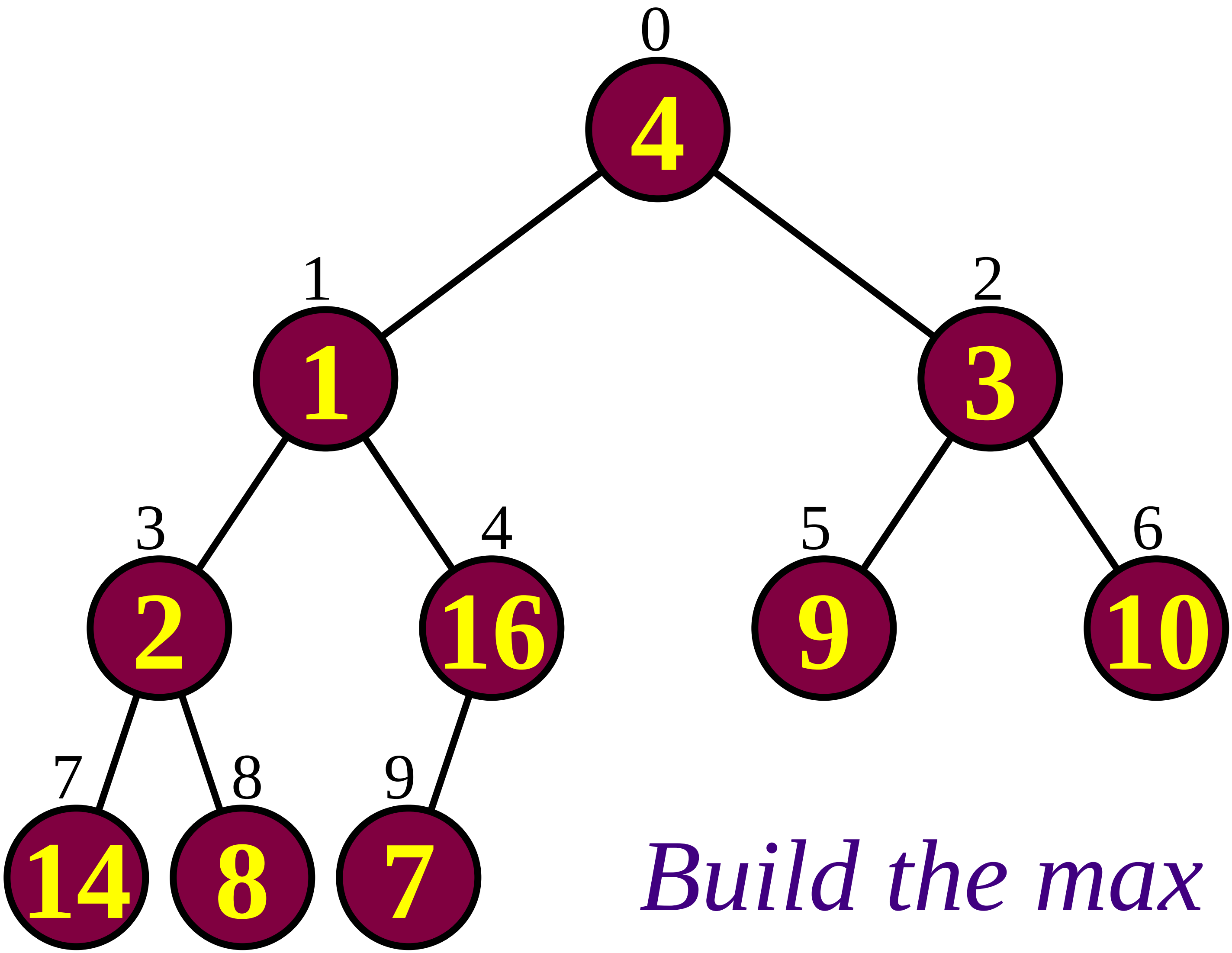
Min-Heapify(노드1)



$O(n)$ time

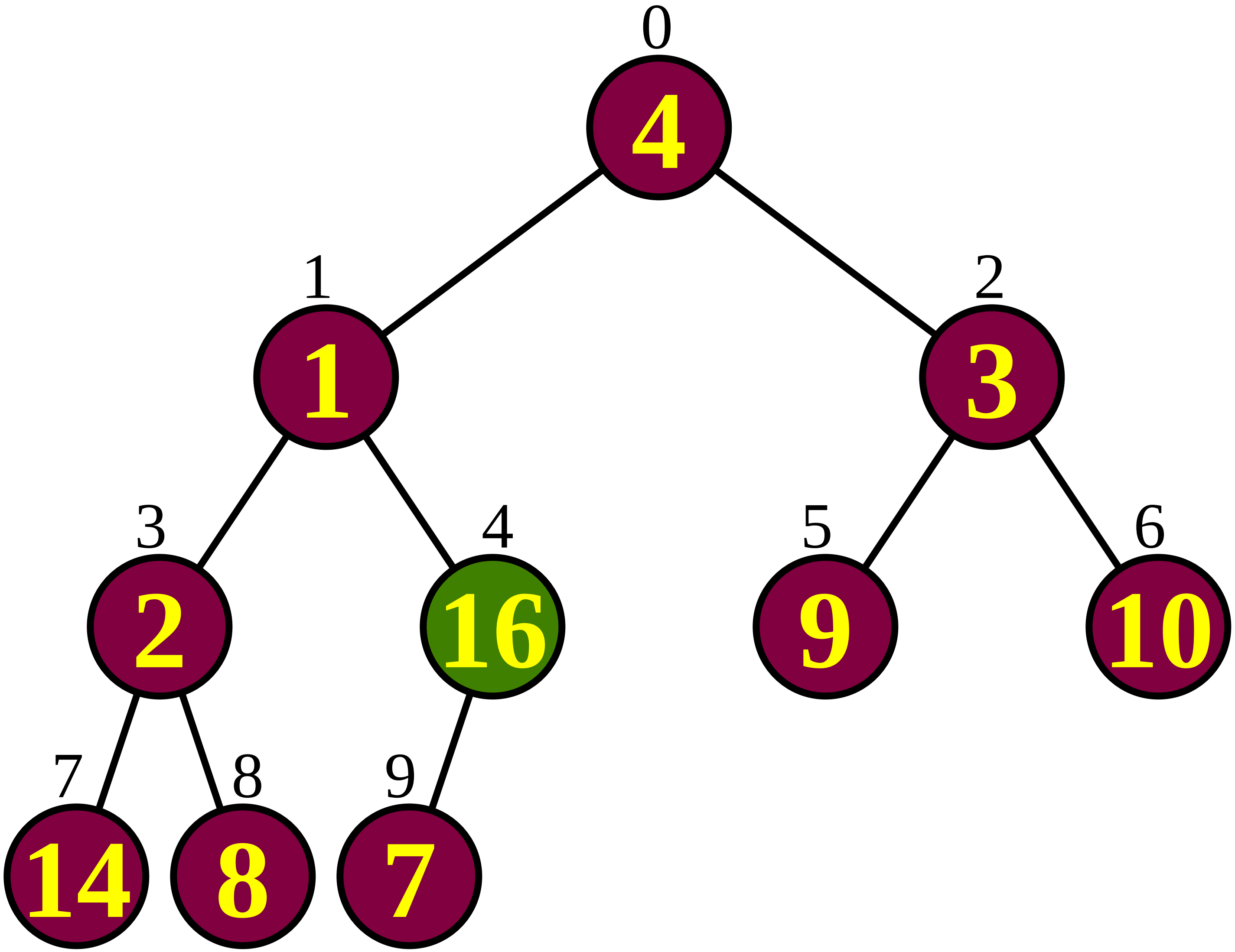
Now Heapsort

Heapsort

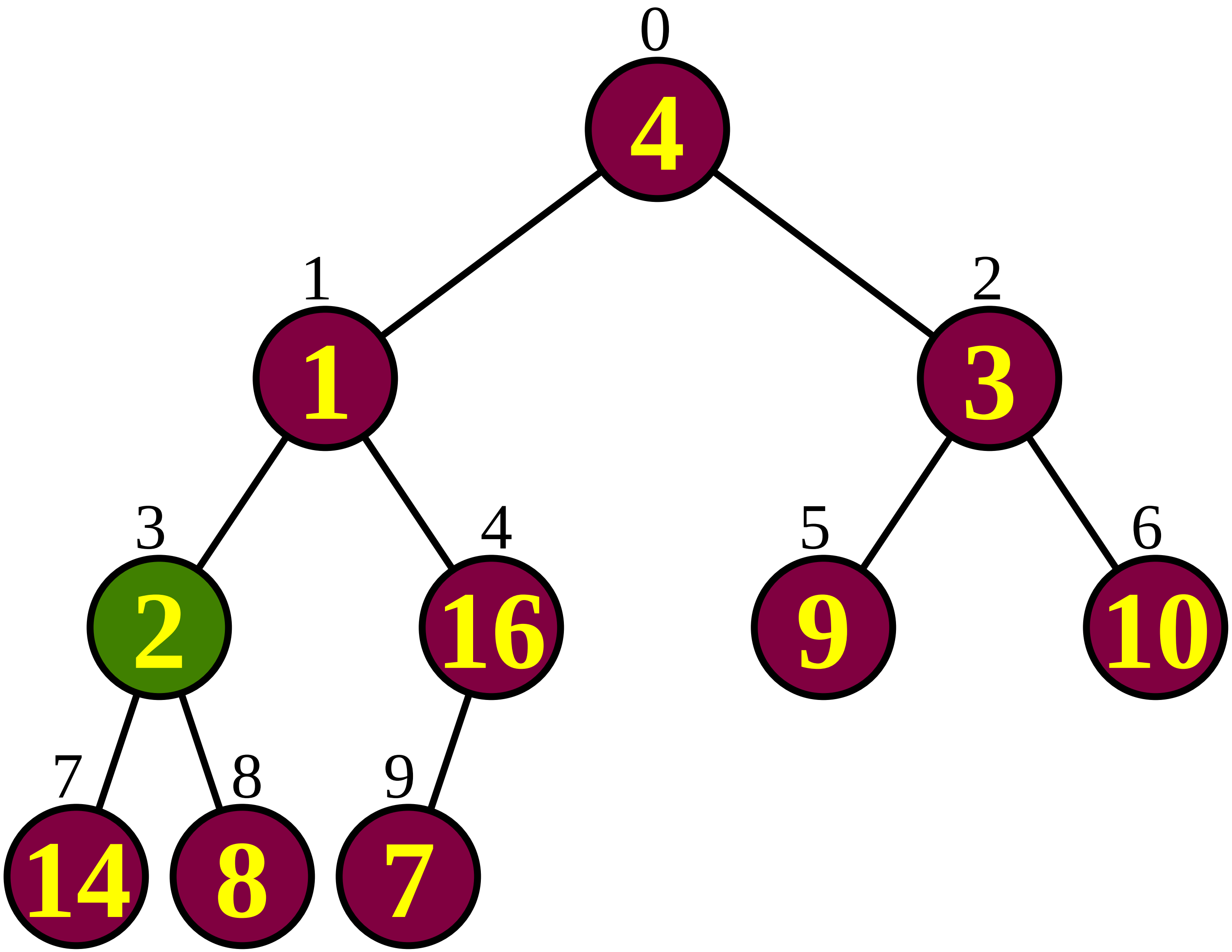


Build the max heap

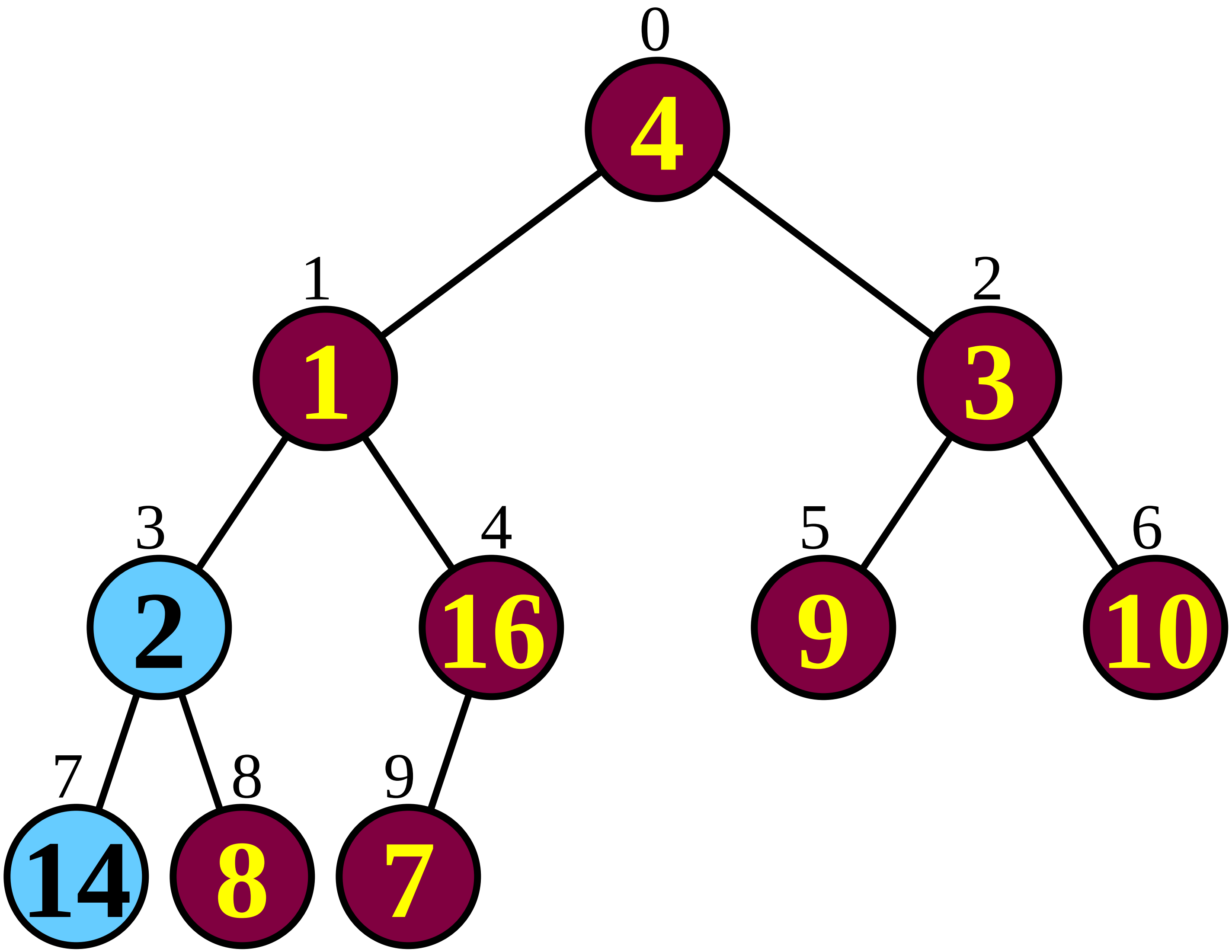
Heapsort



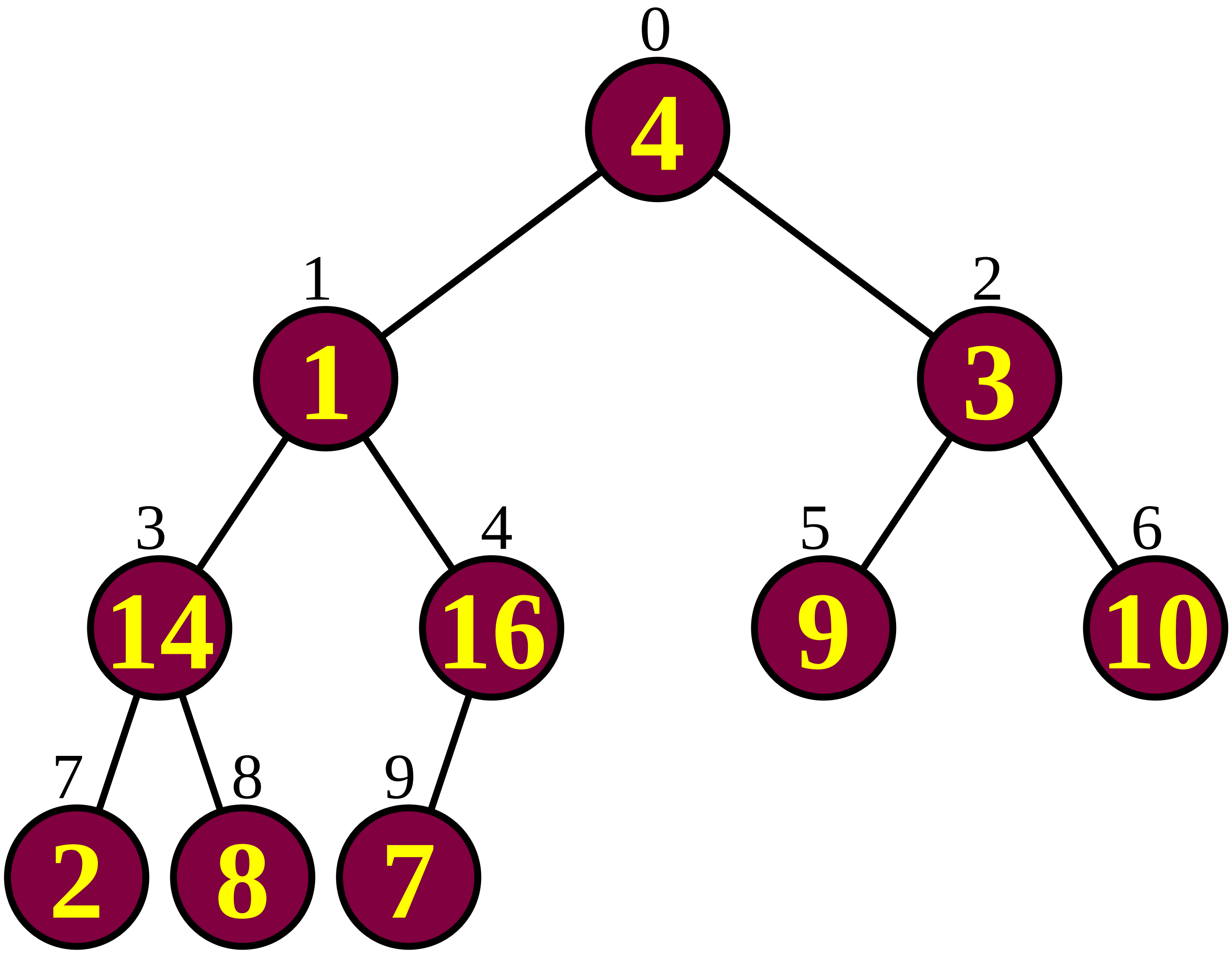
Heapsort



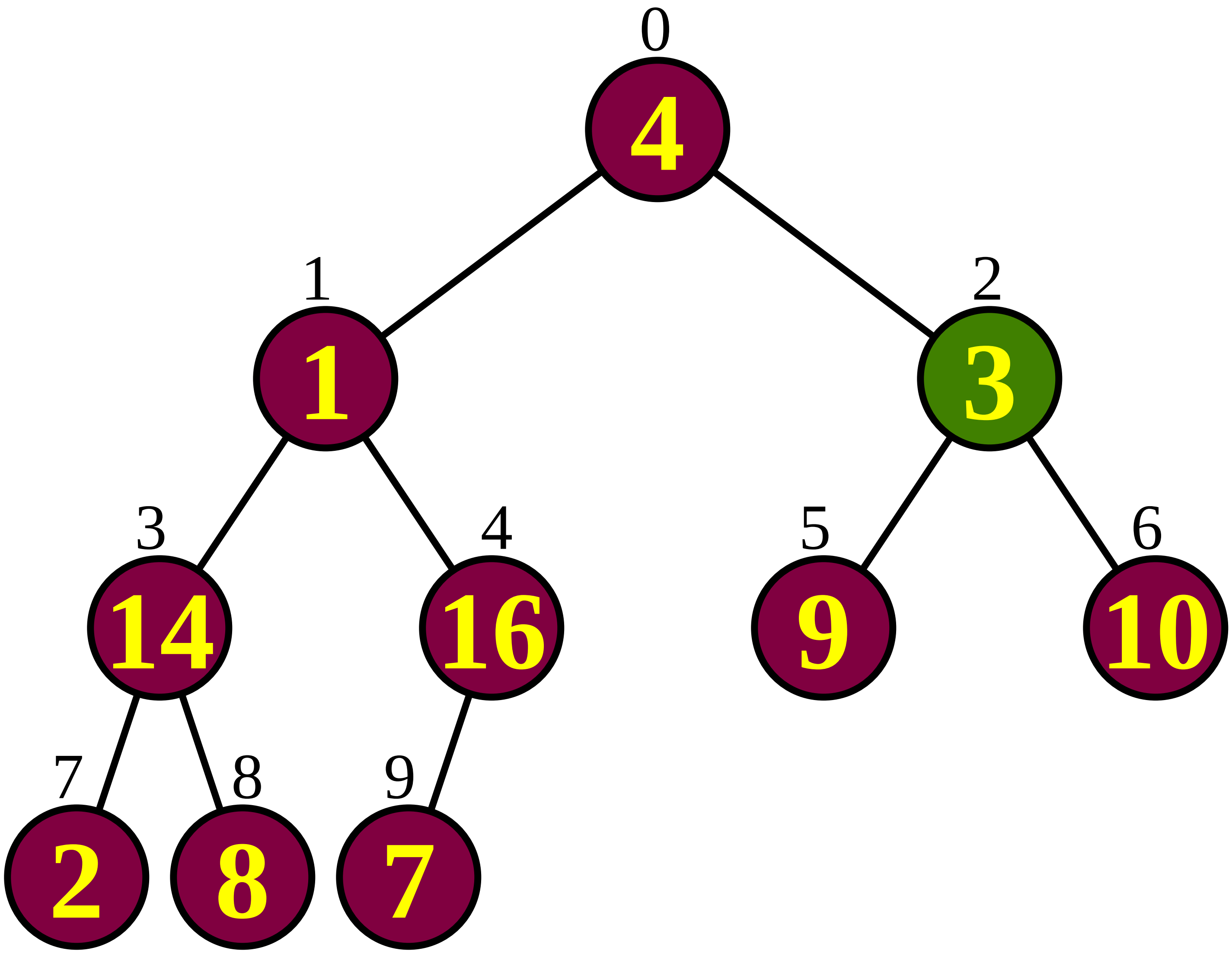
Heapsort



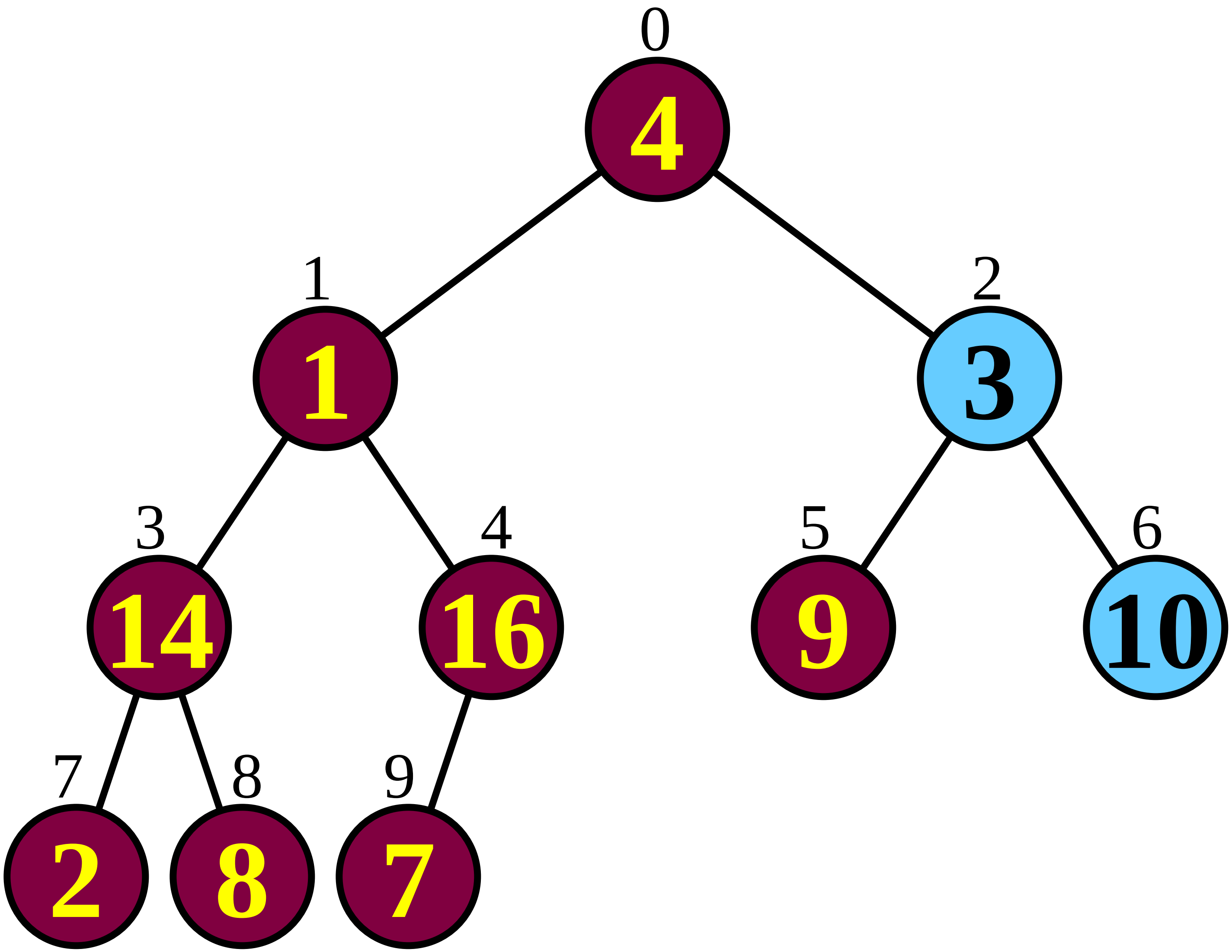
Heapsort



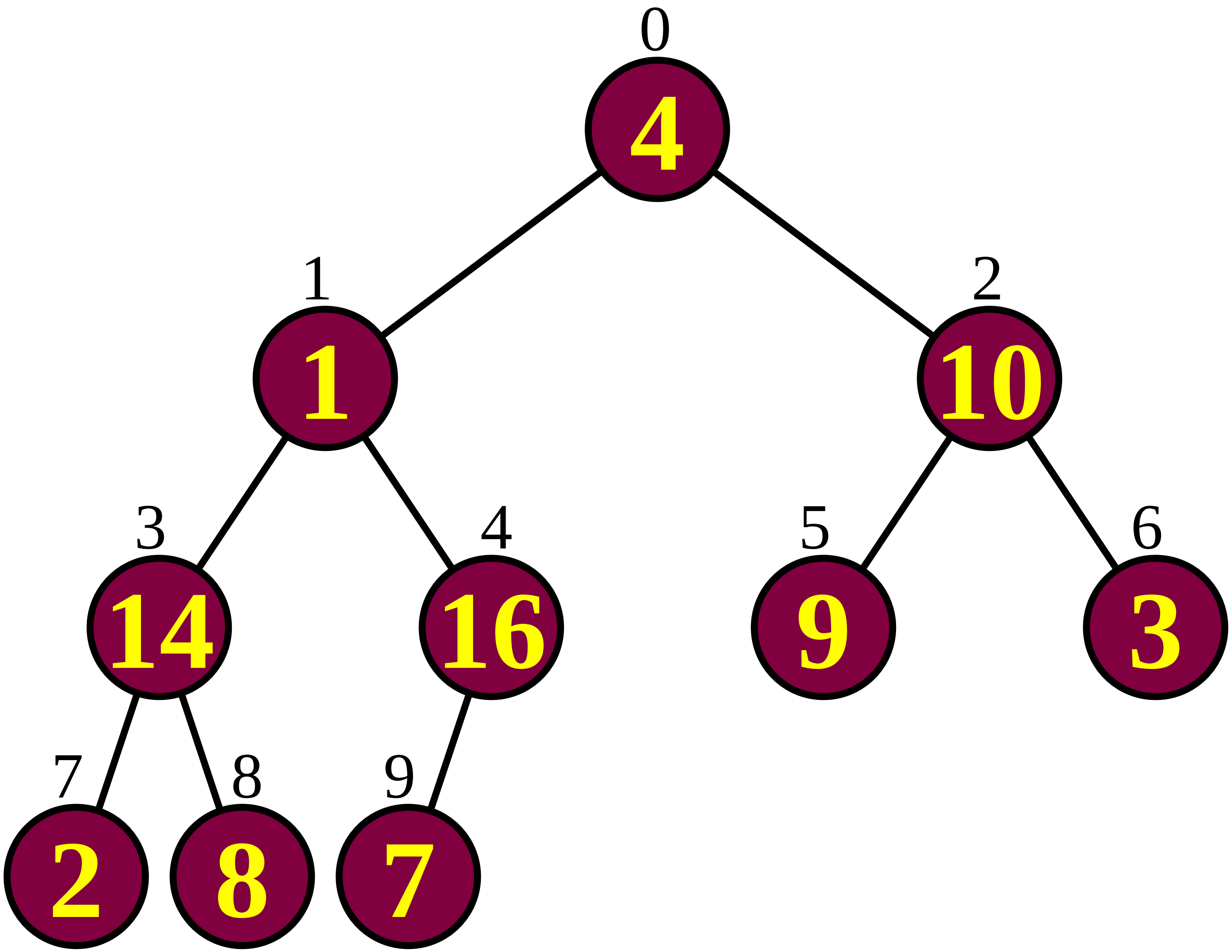
Heapsort



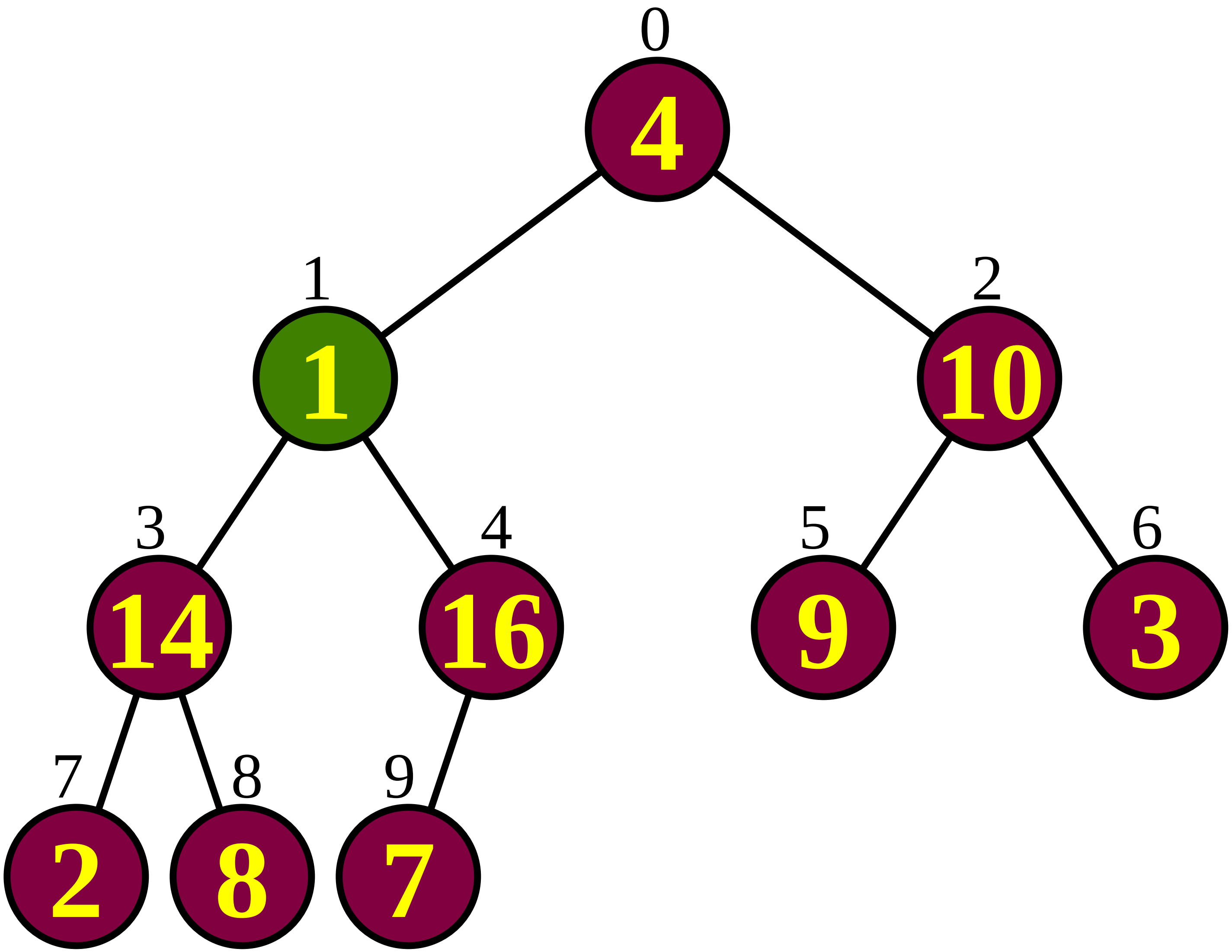
Heapsort



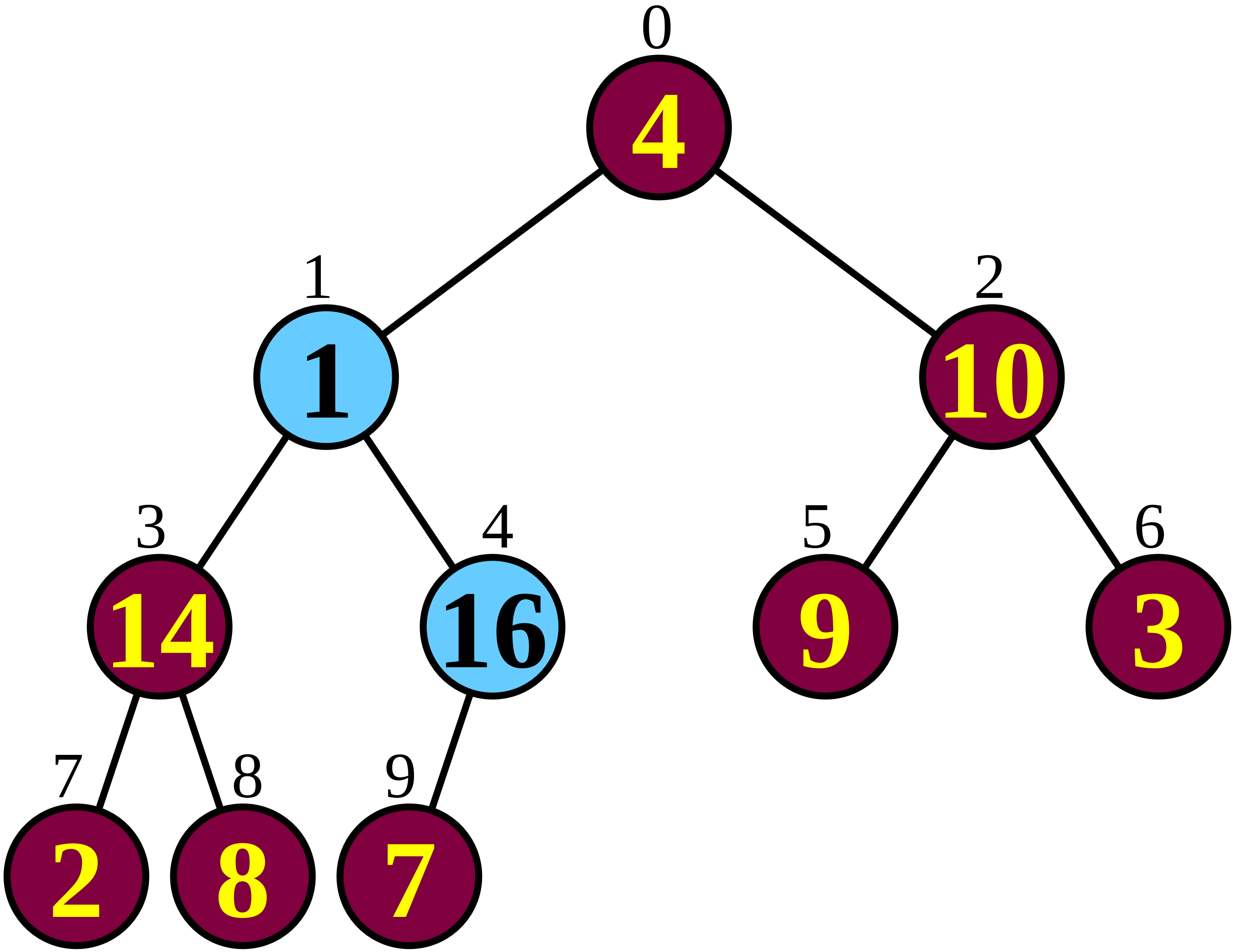
Heapsort



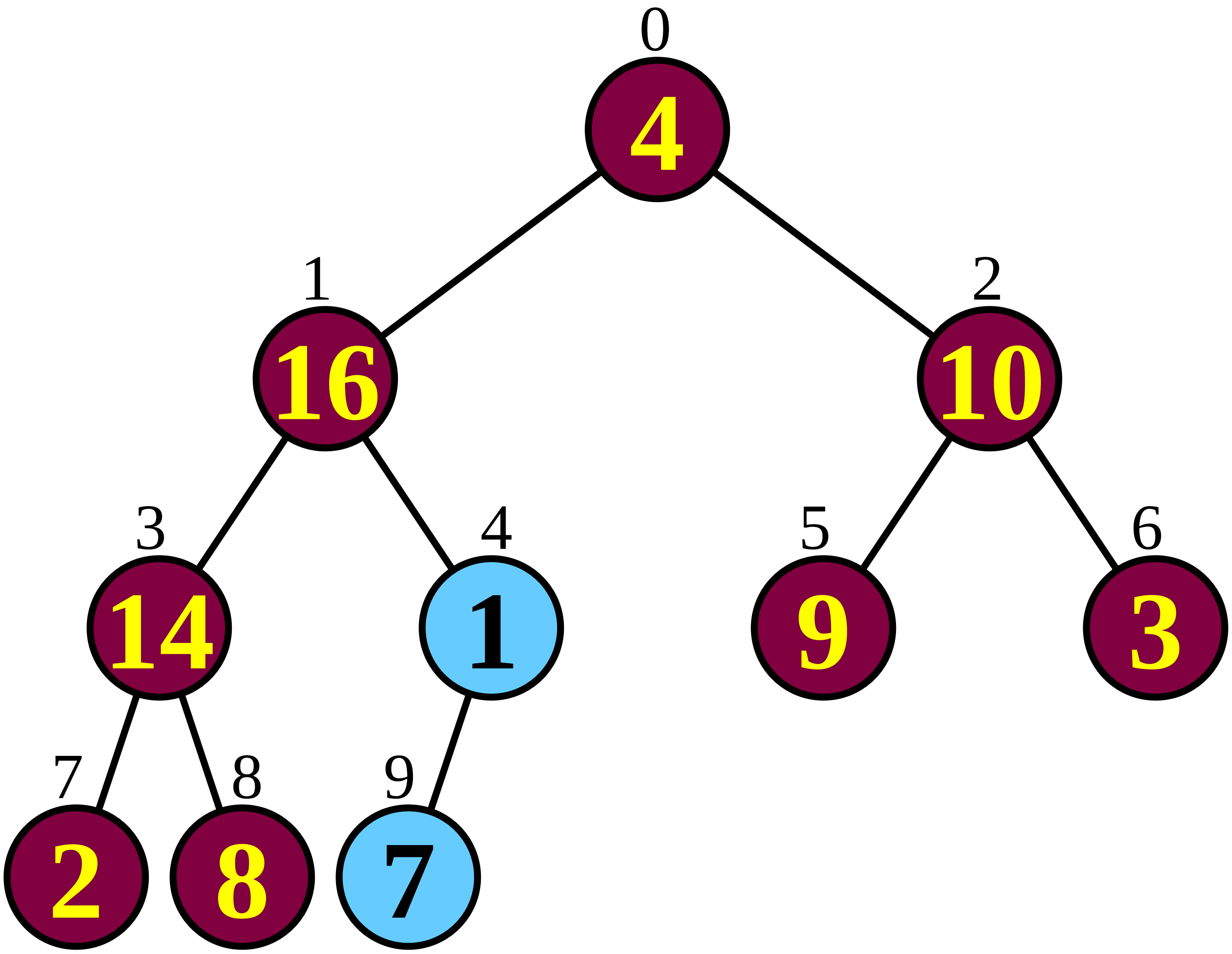
Heapsort



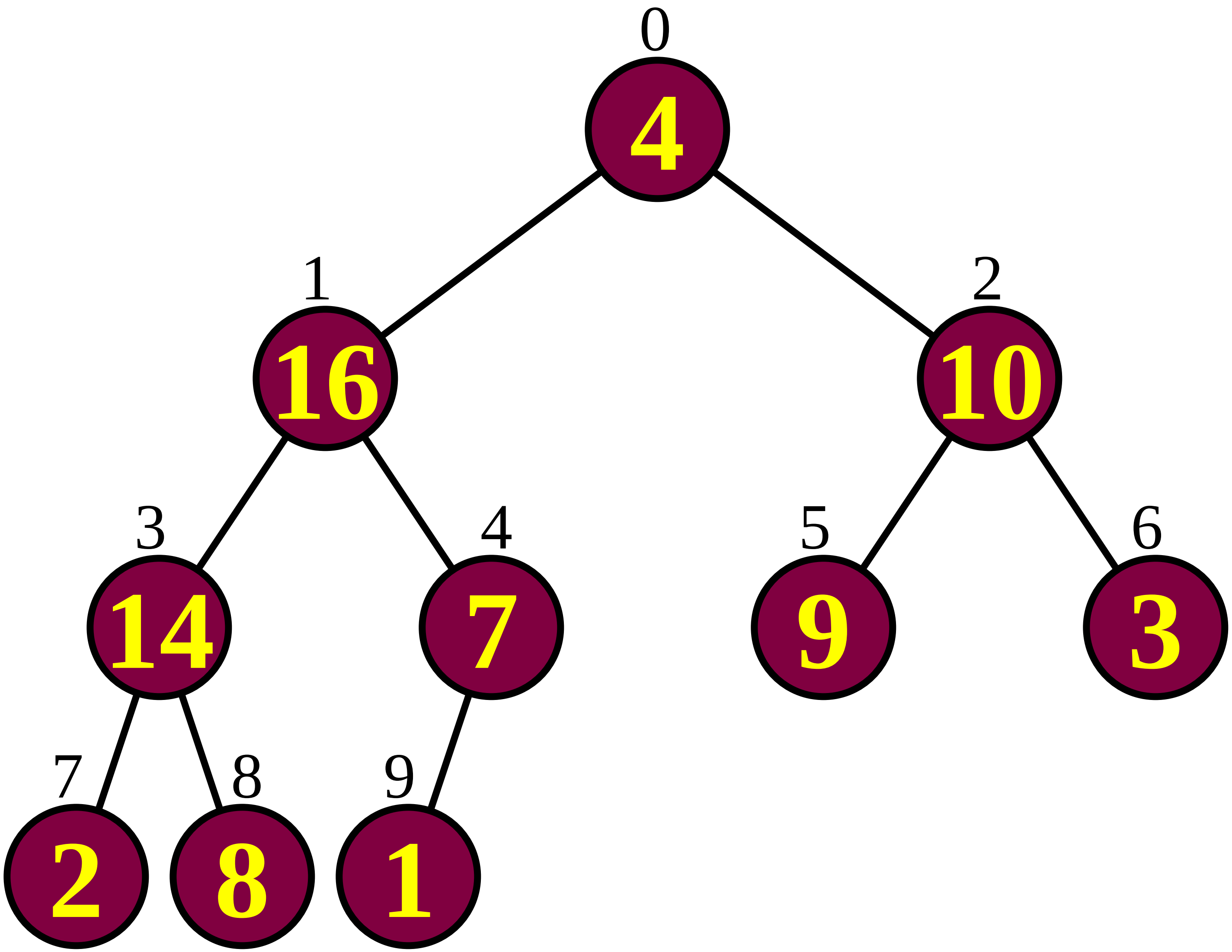
Heapsort



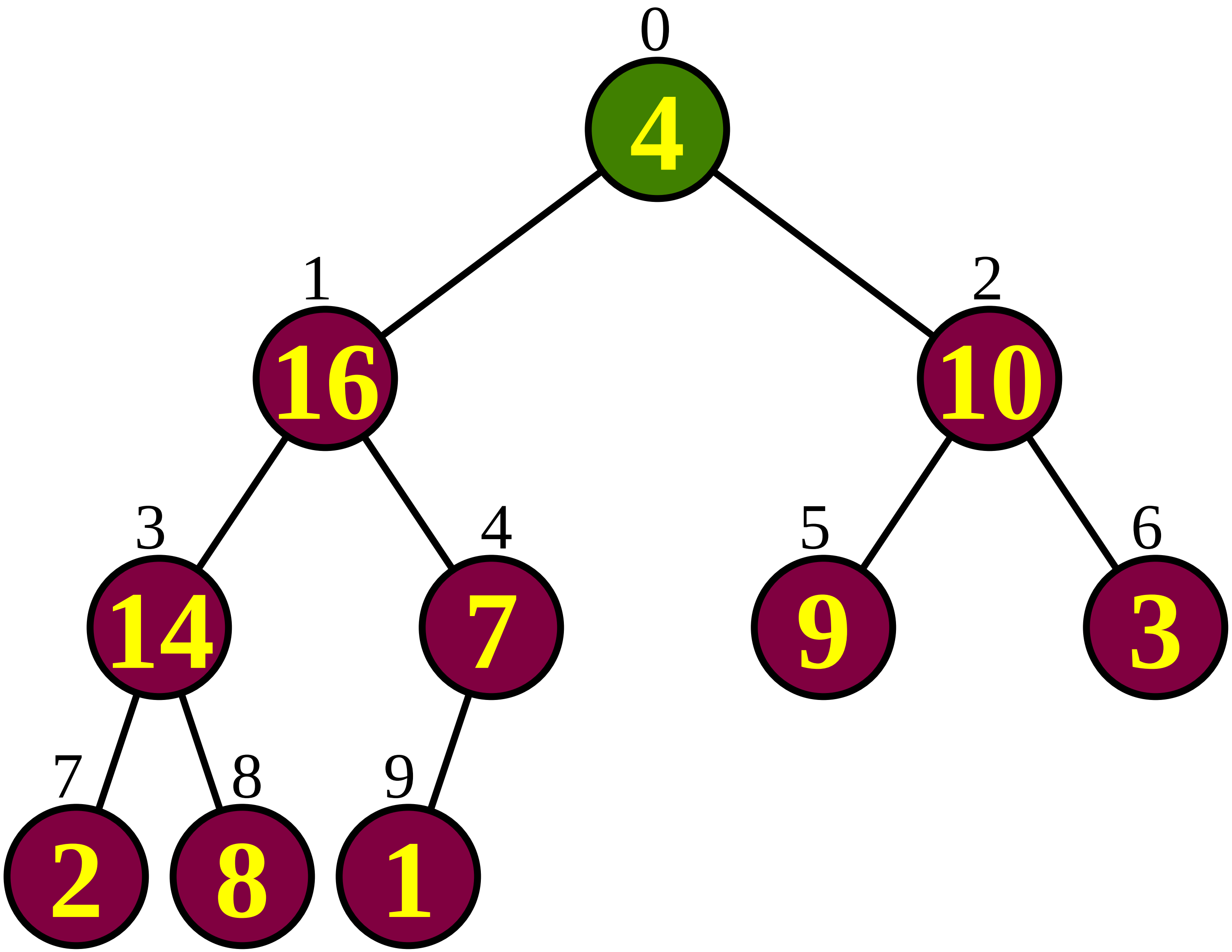
Heapsort



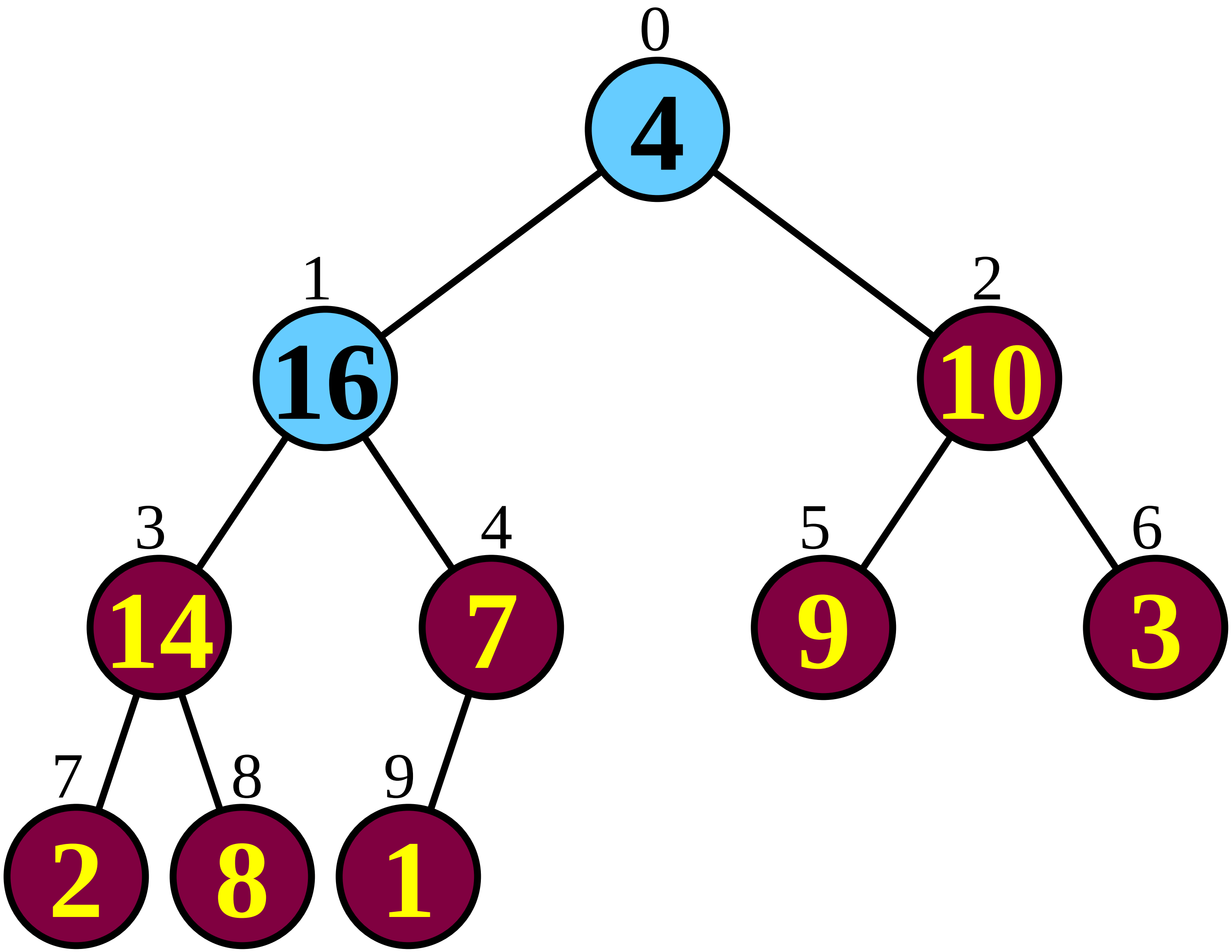
Heapsort



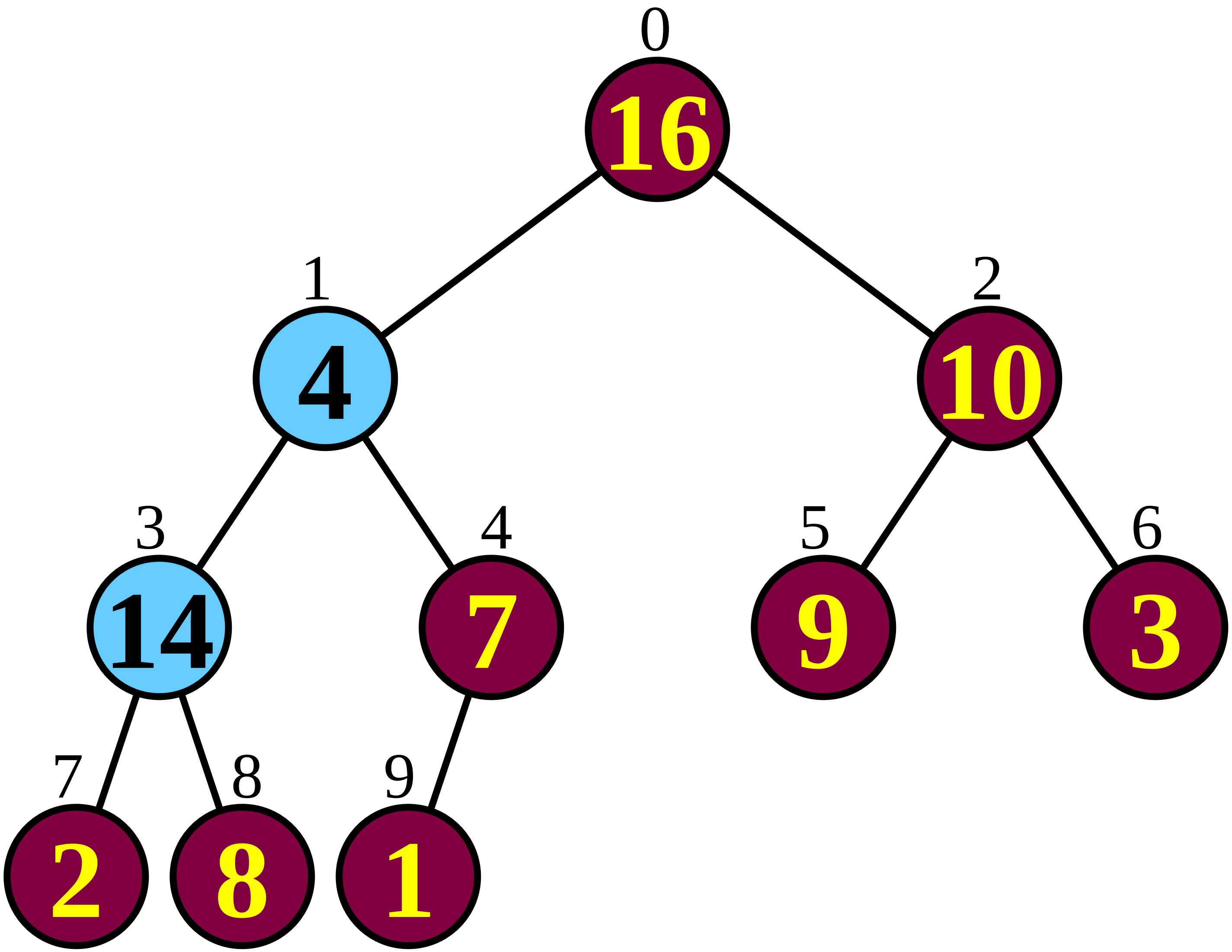
Heapsort



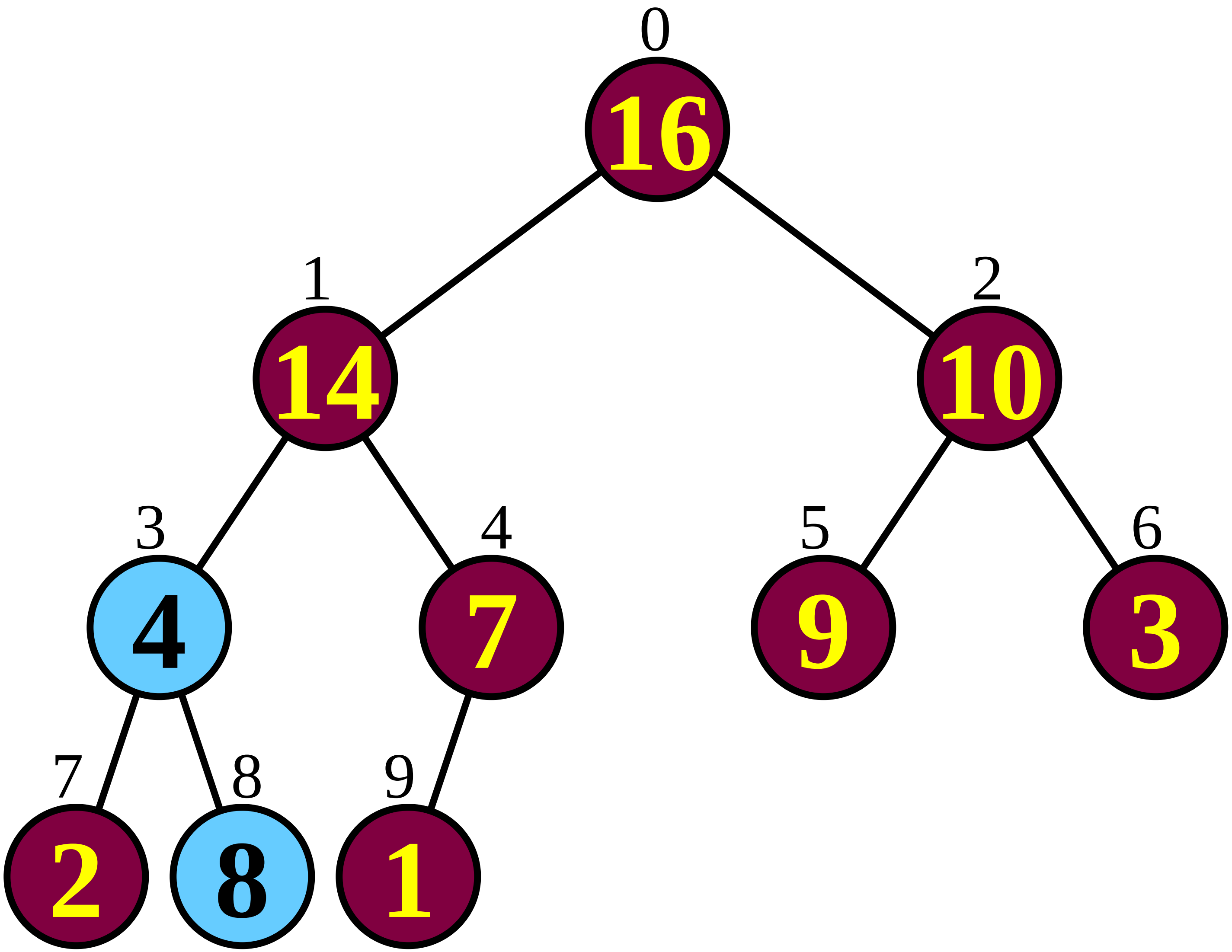
Heapsort



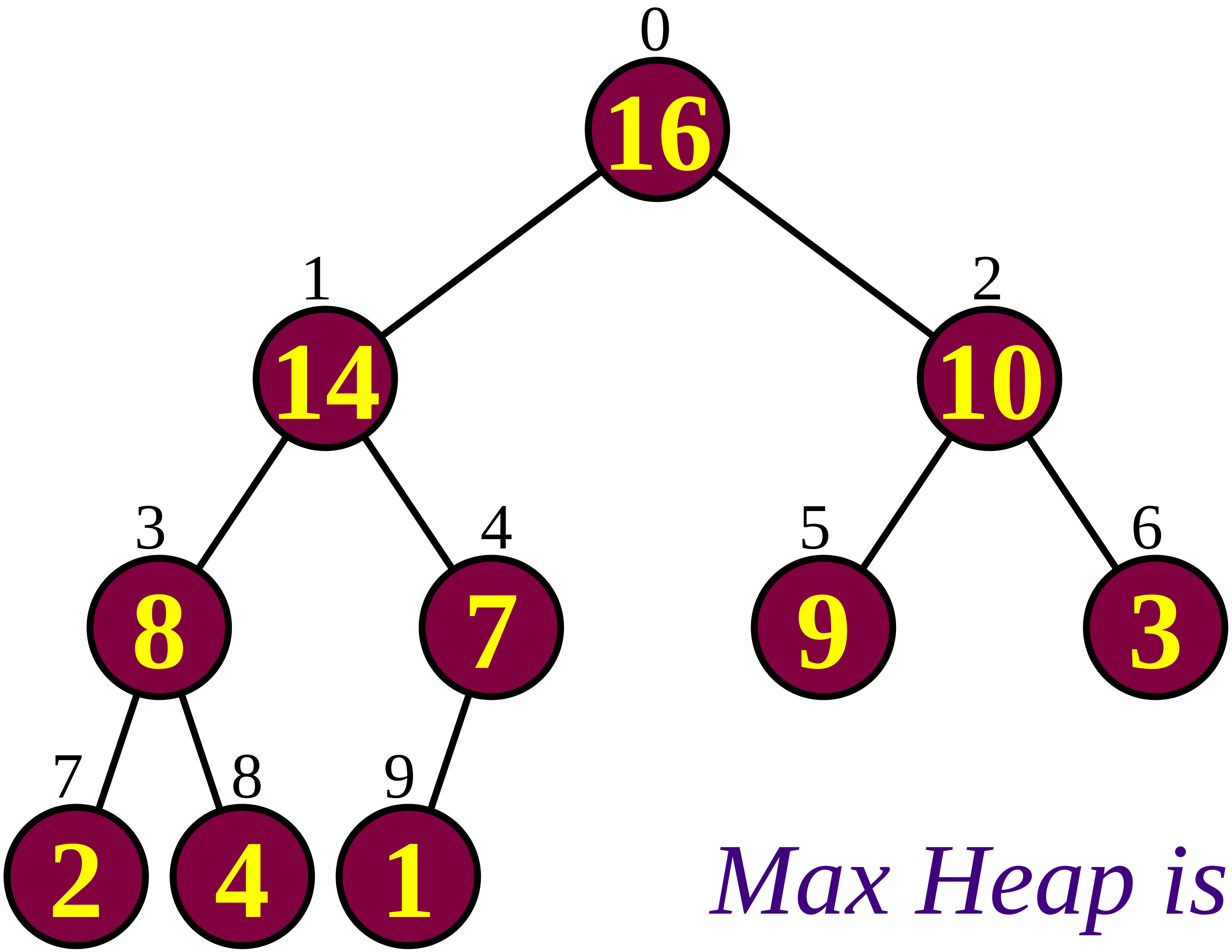
Heapsort



Heapsort



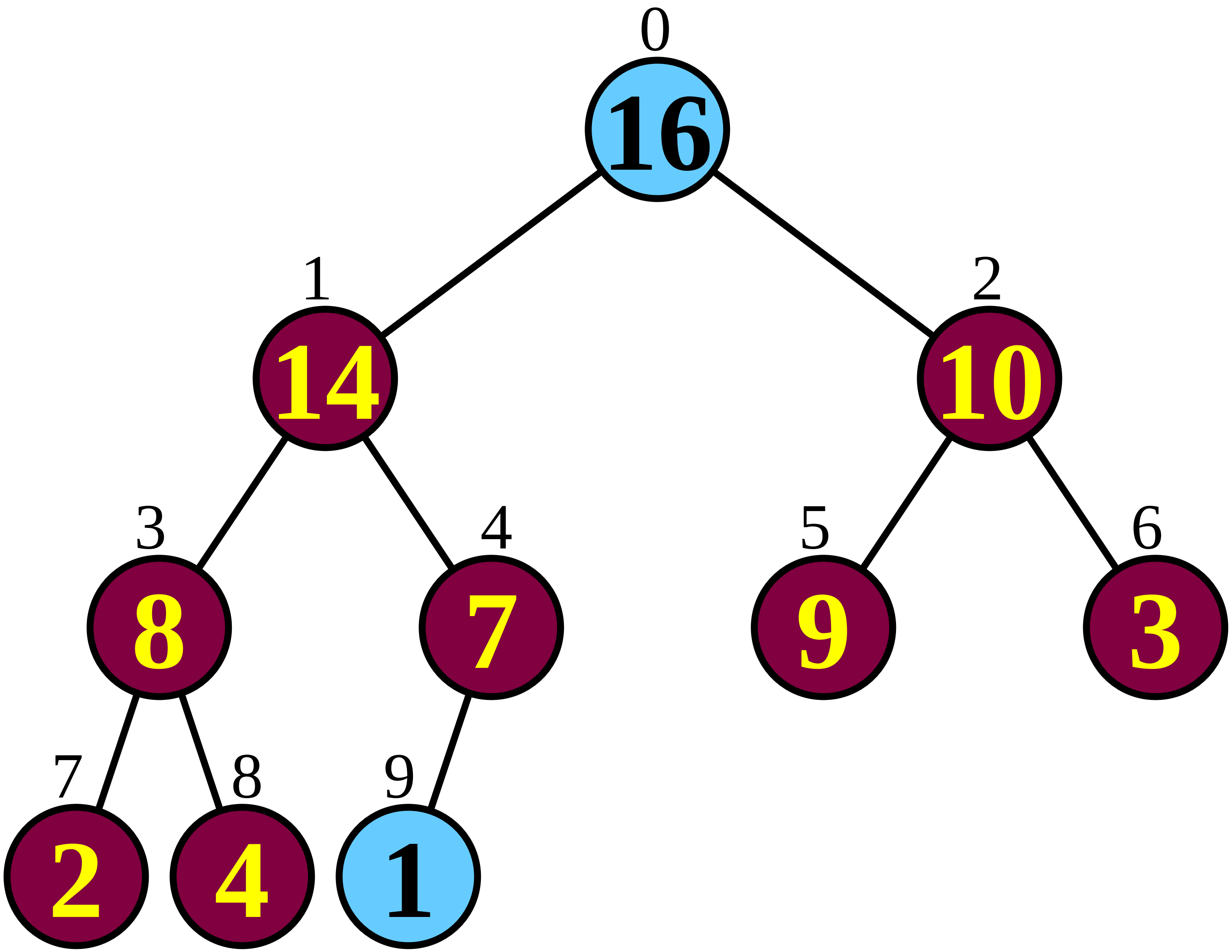
Heapsort



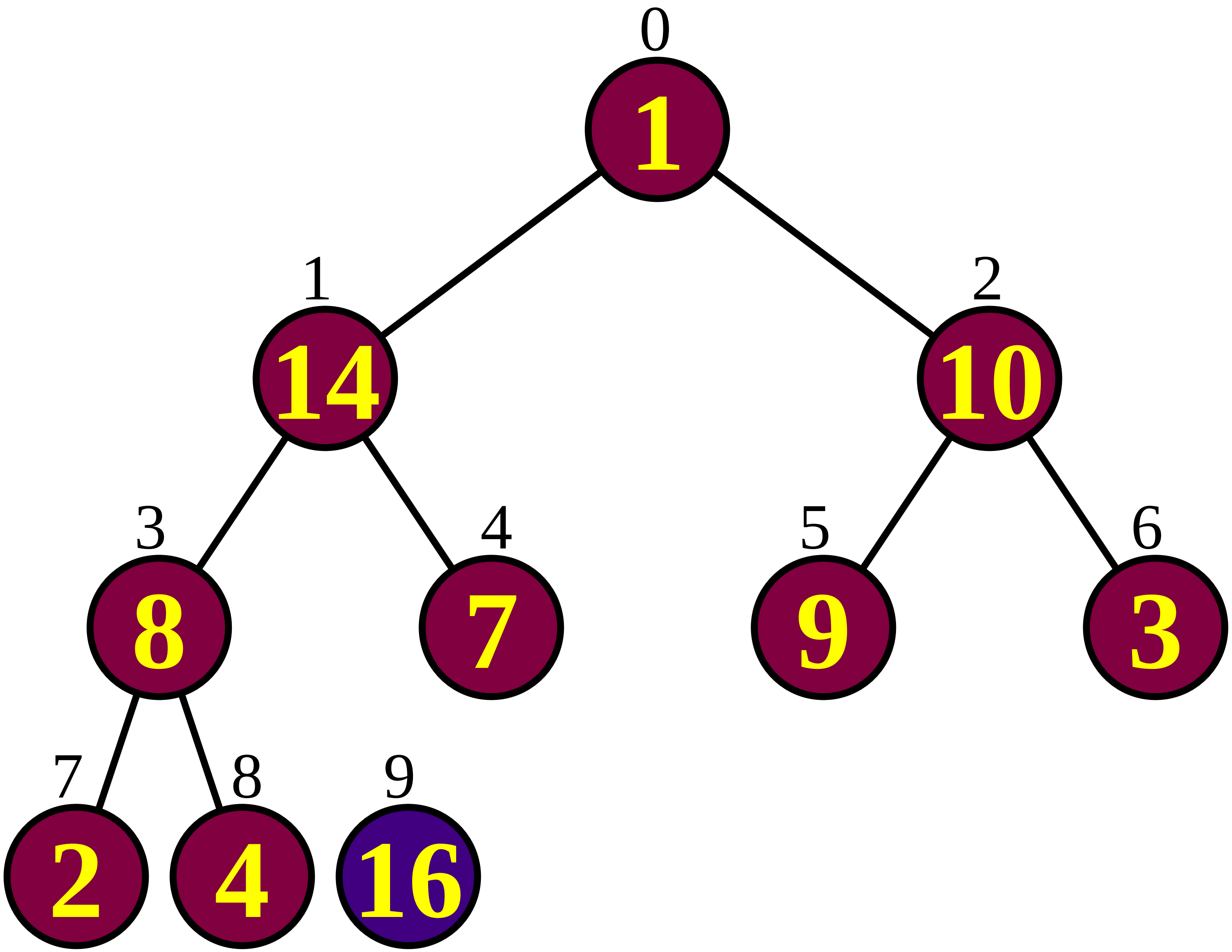
Max Heap is built

- Now we delete operation *n* times. N is the number of elements in a heap.

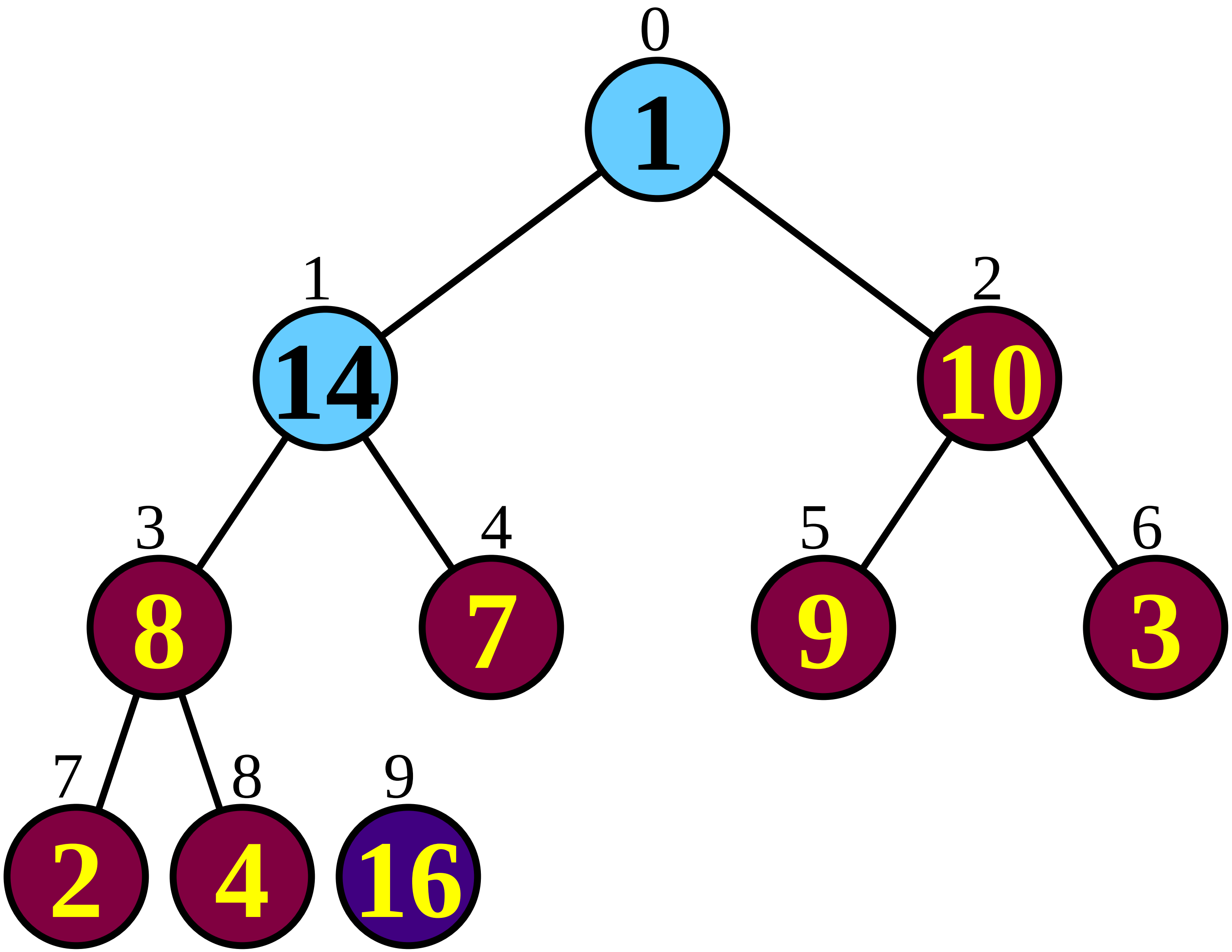
Heapsort



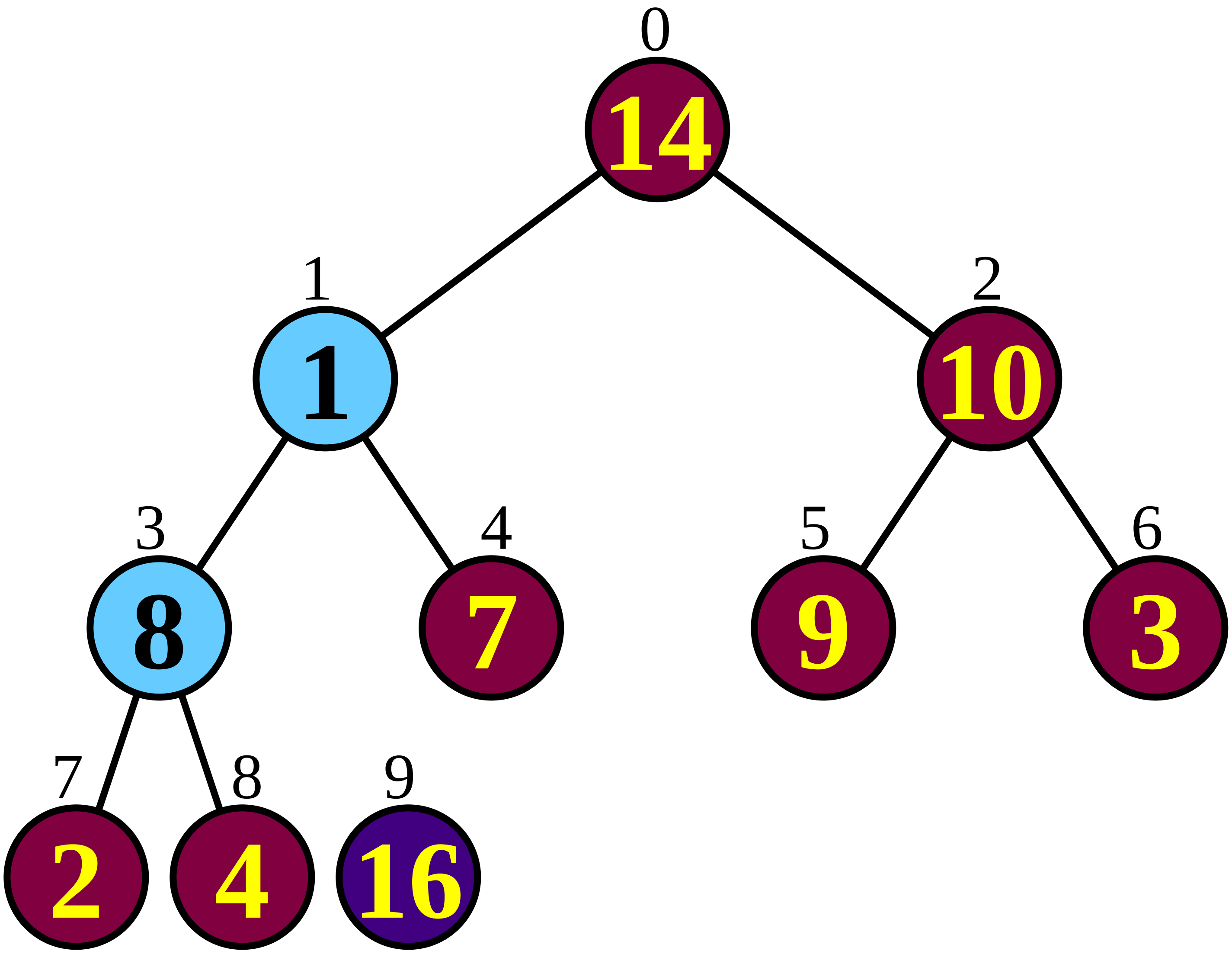
Heapsort



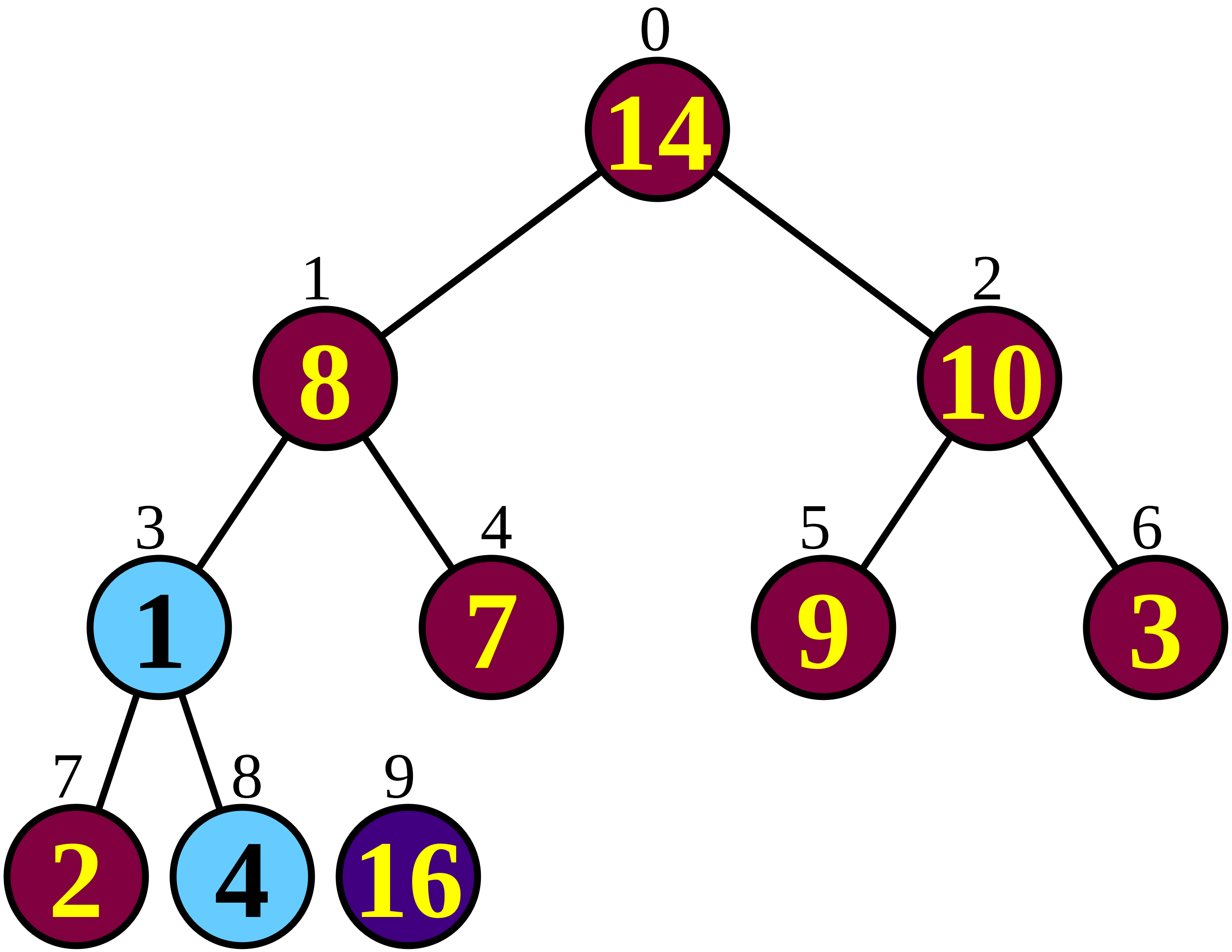
Heapsort



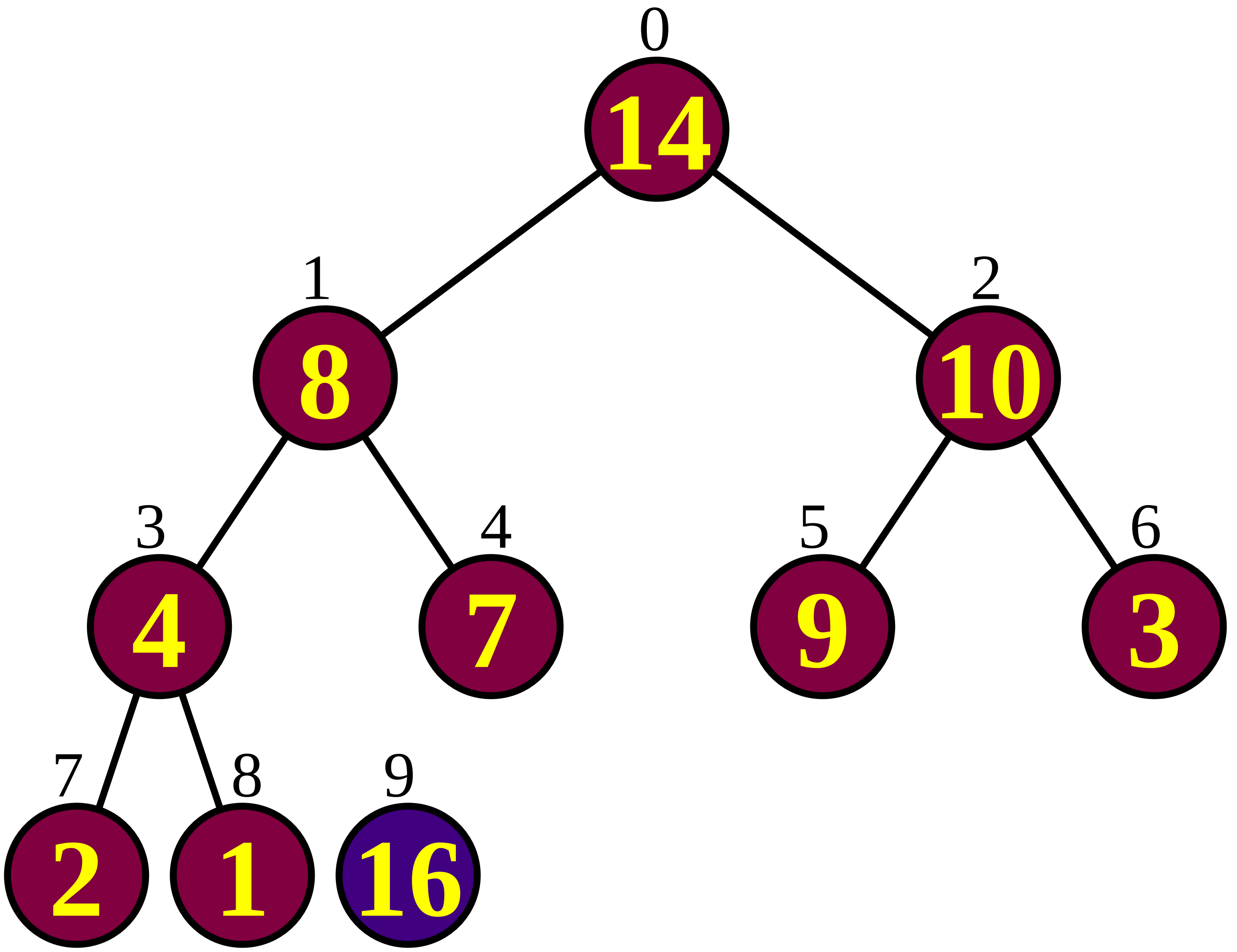
Heapsort



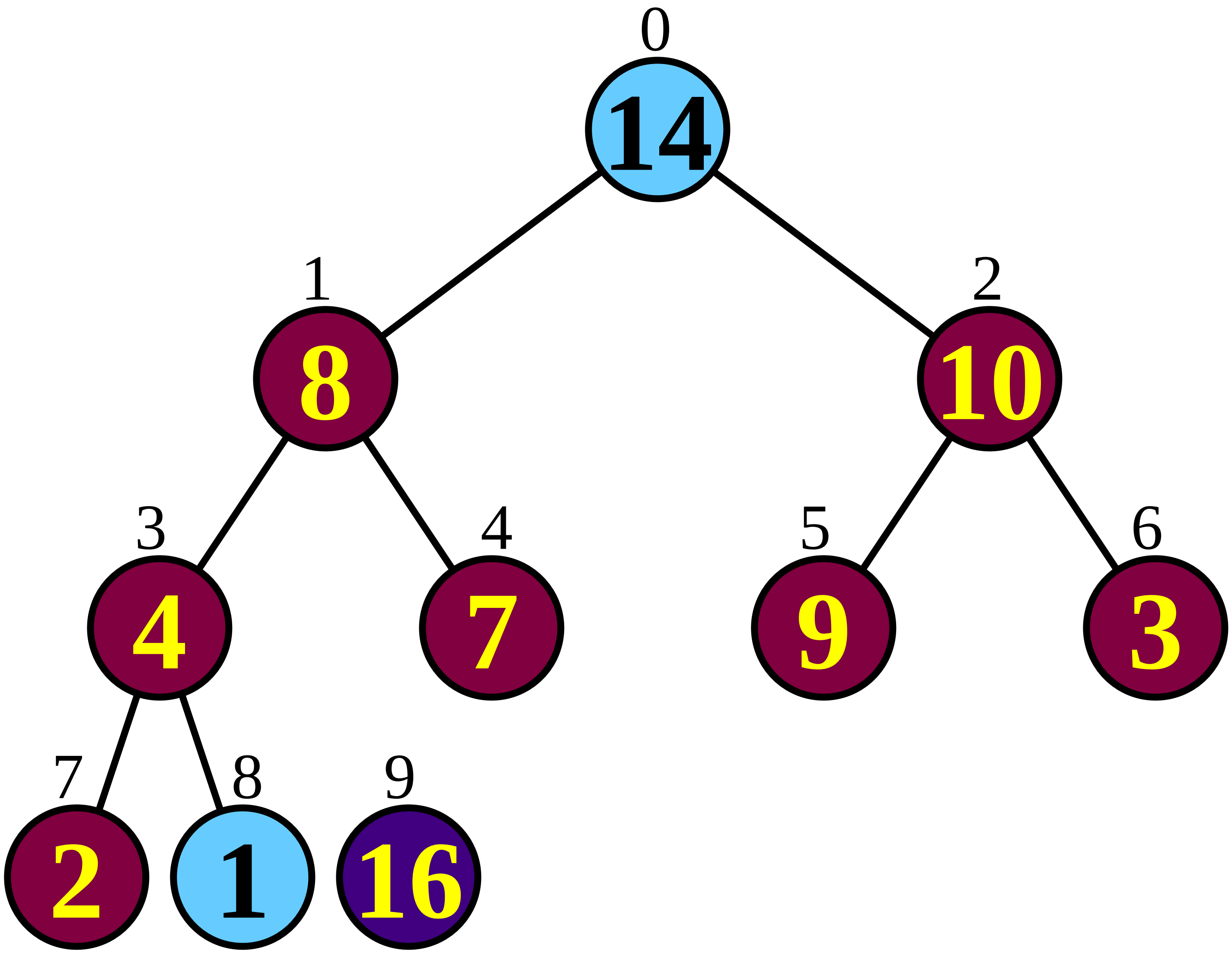
Heapsort



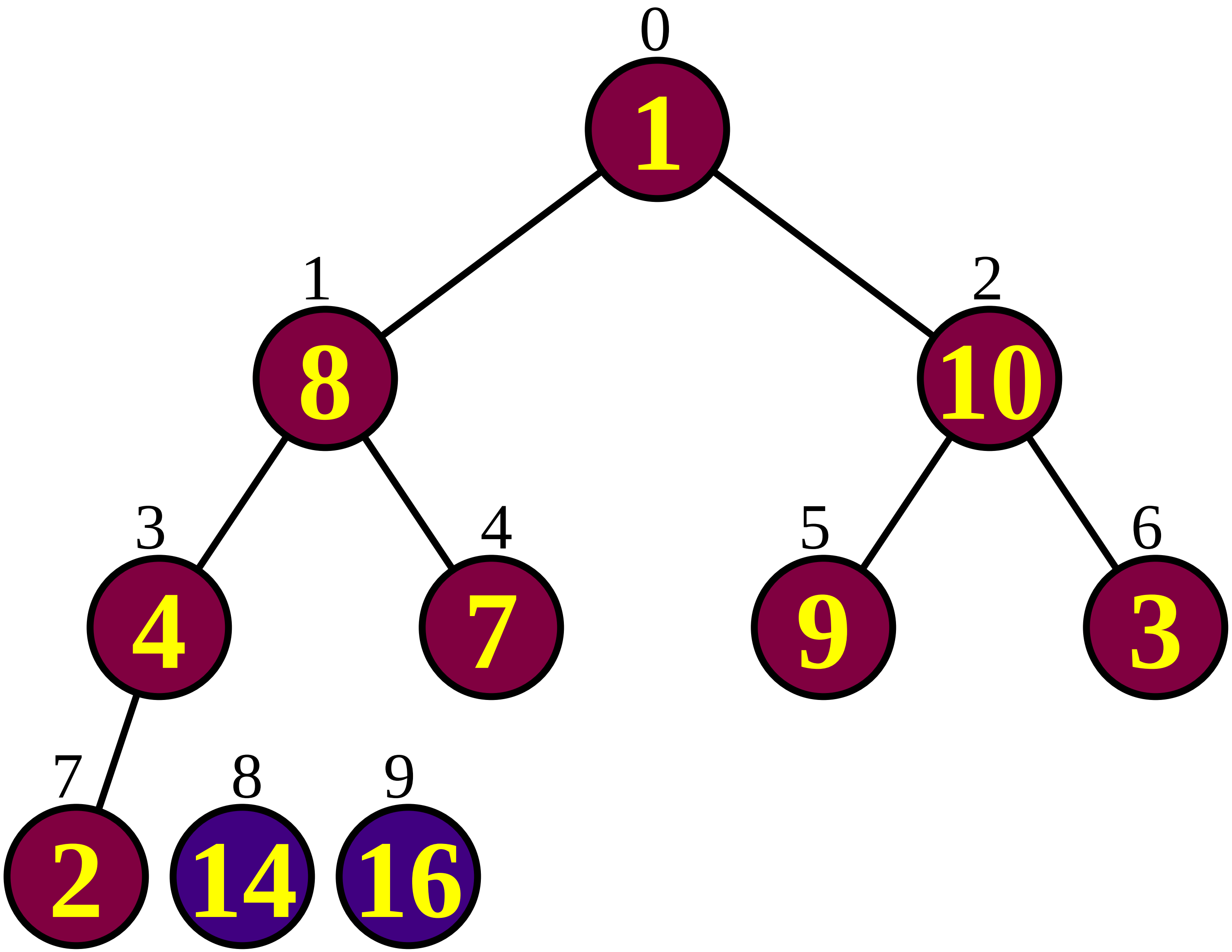
Heapsort



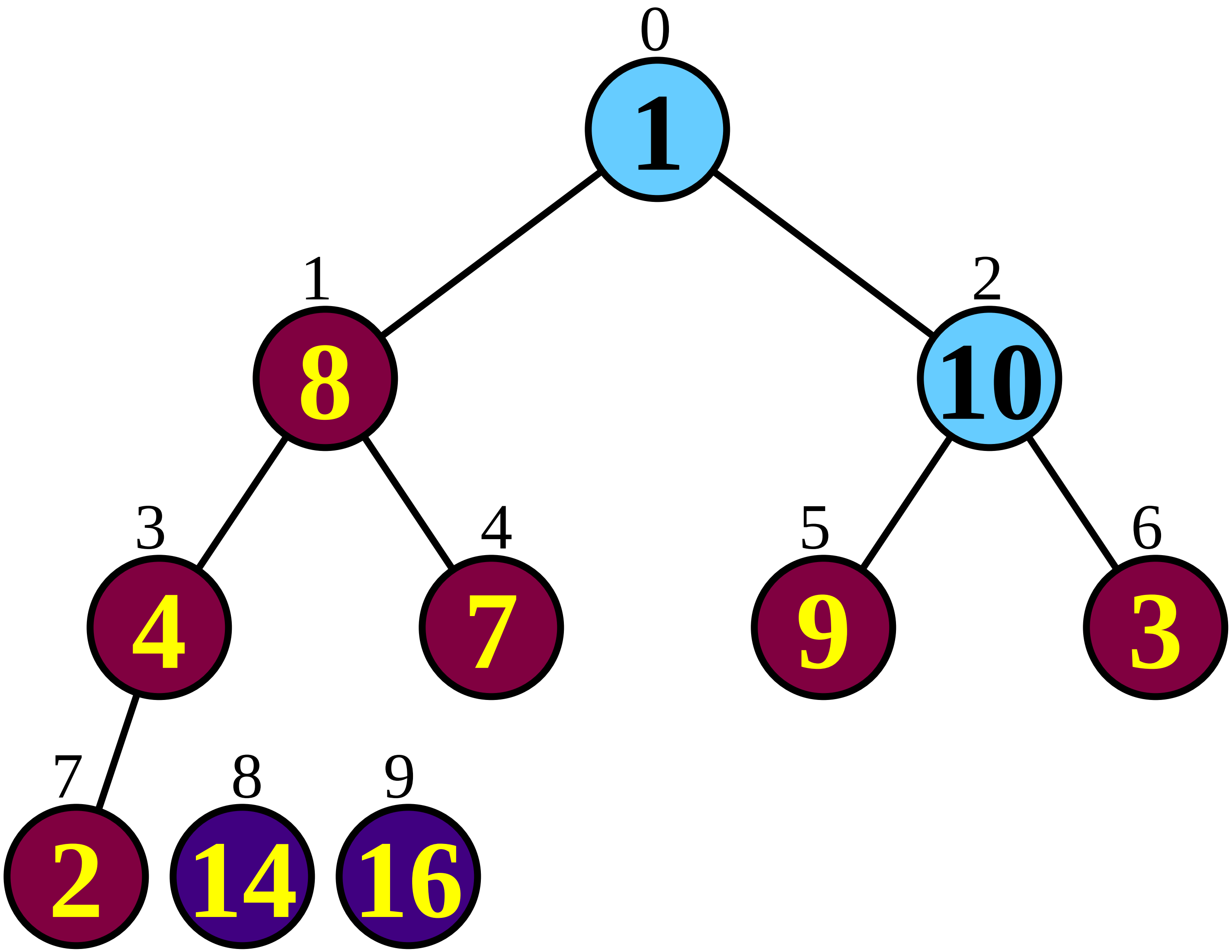
Heapsort



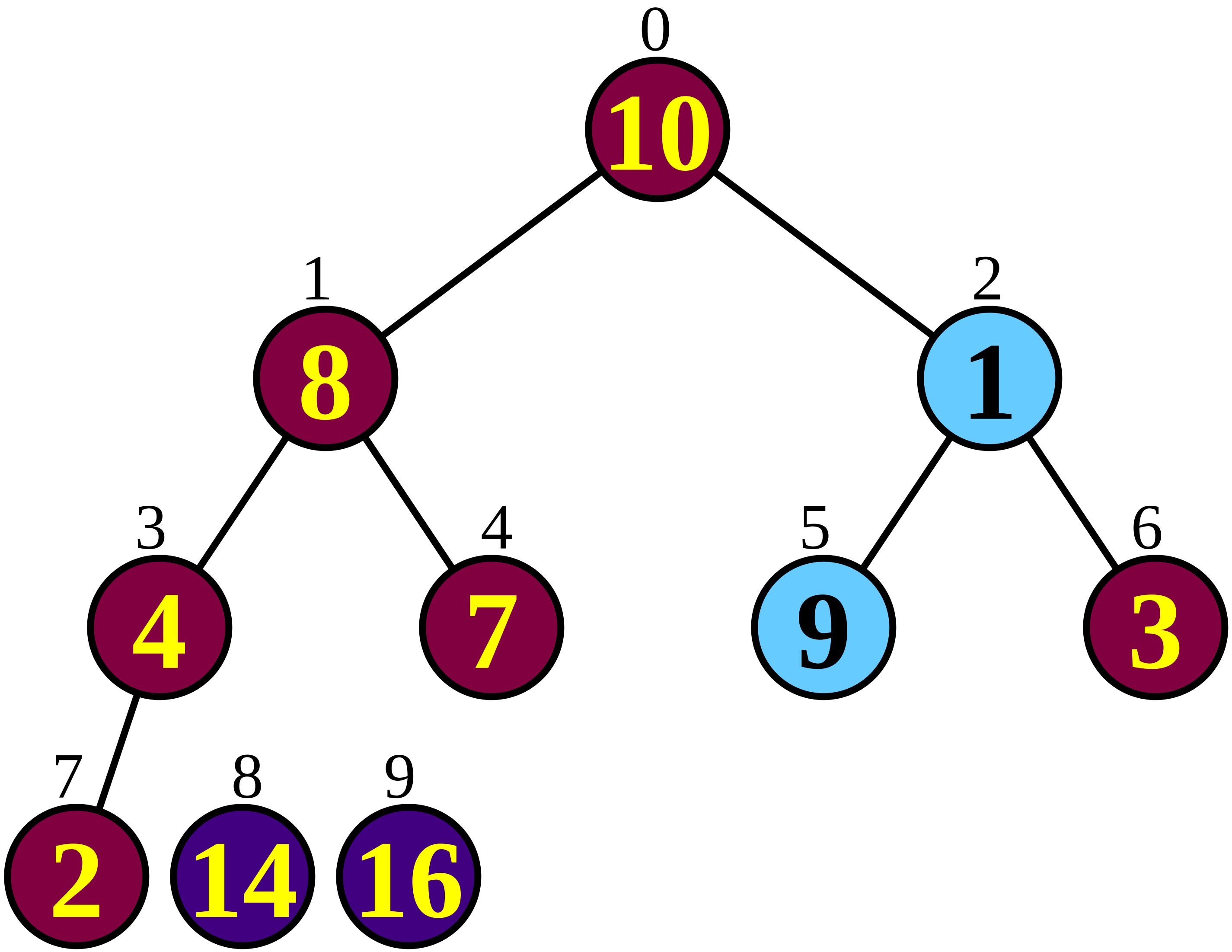
Heapsort



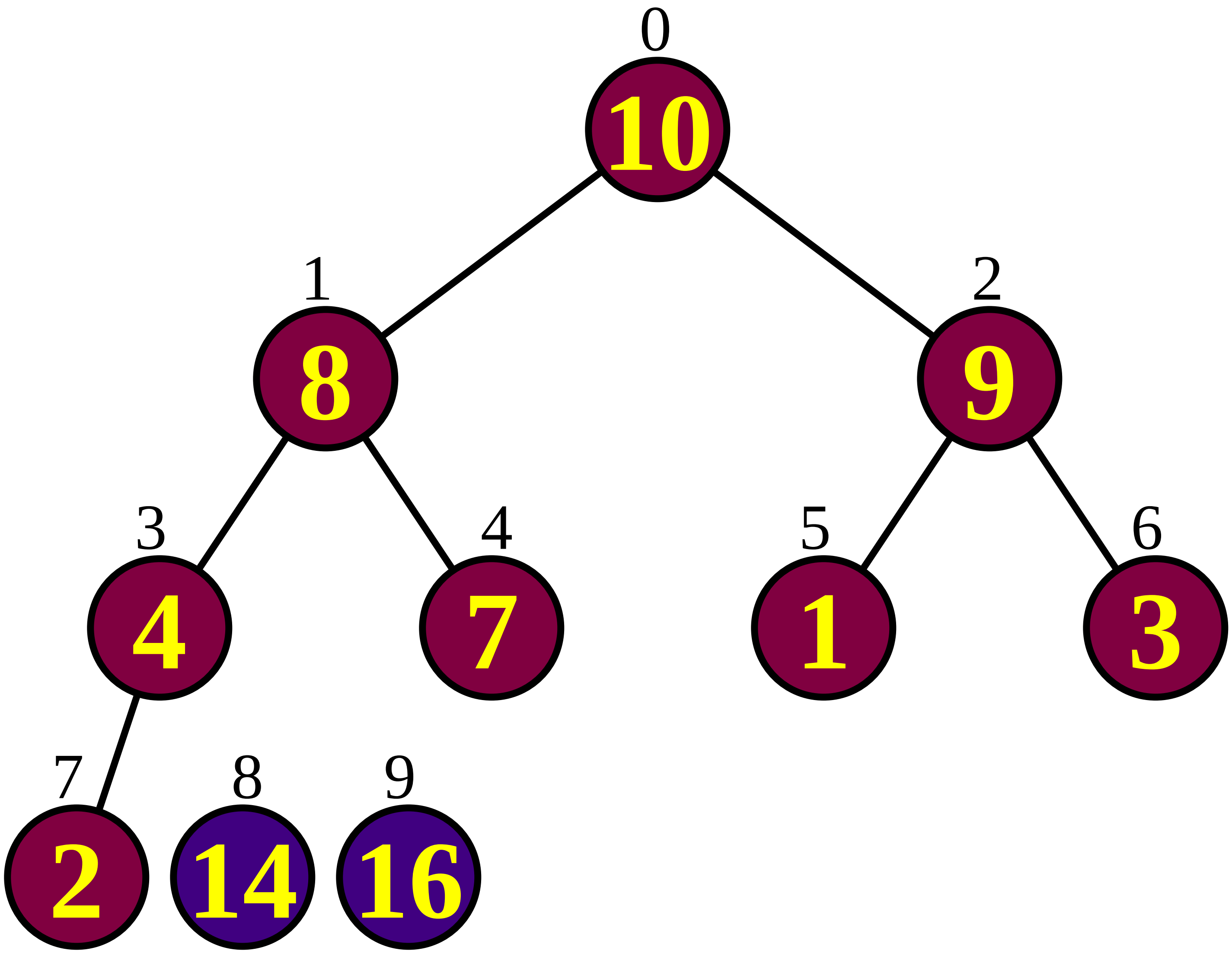
Heapsort



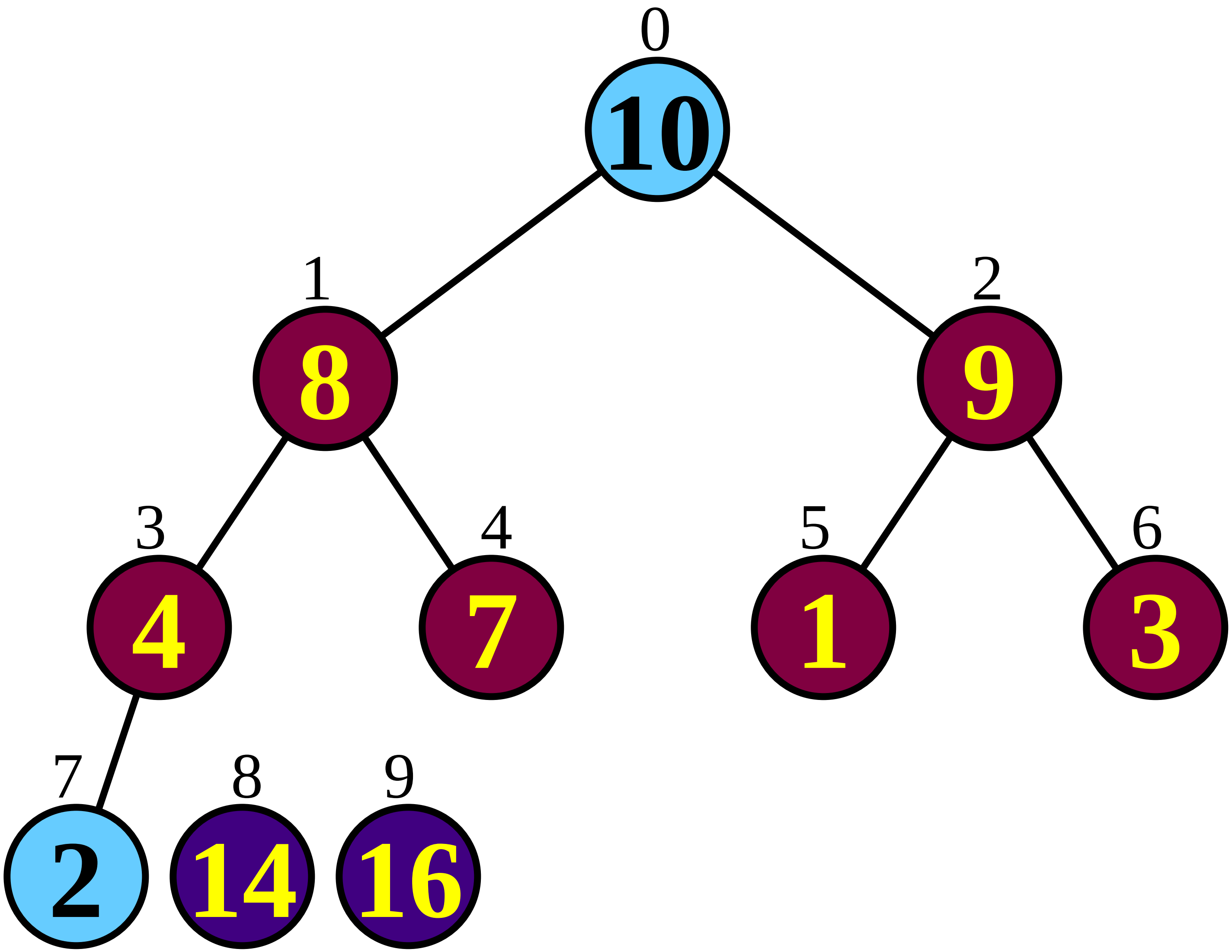
Heapsort



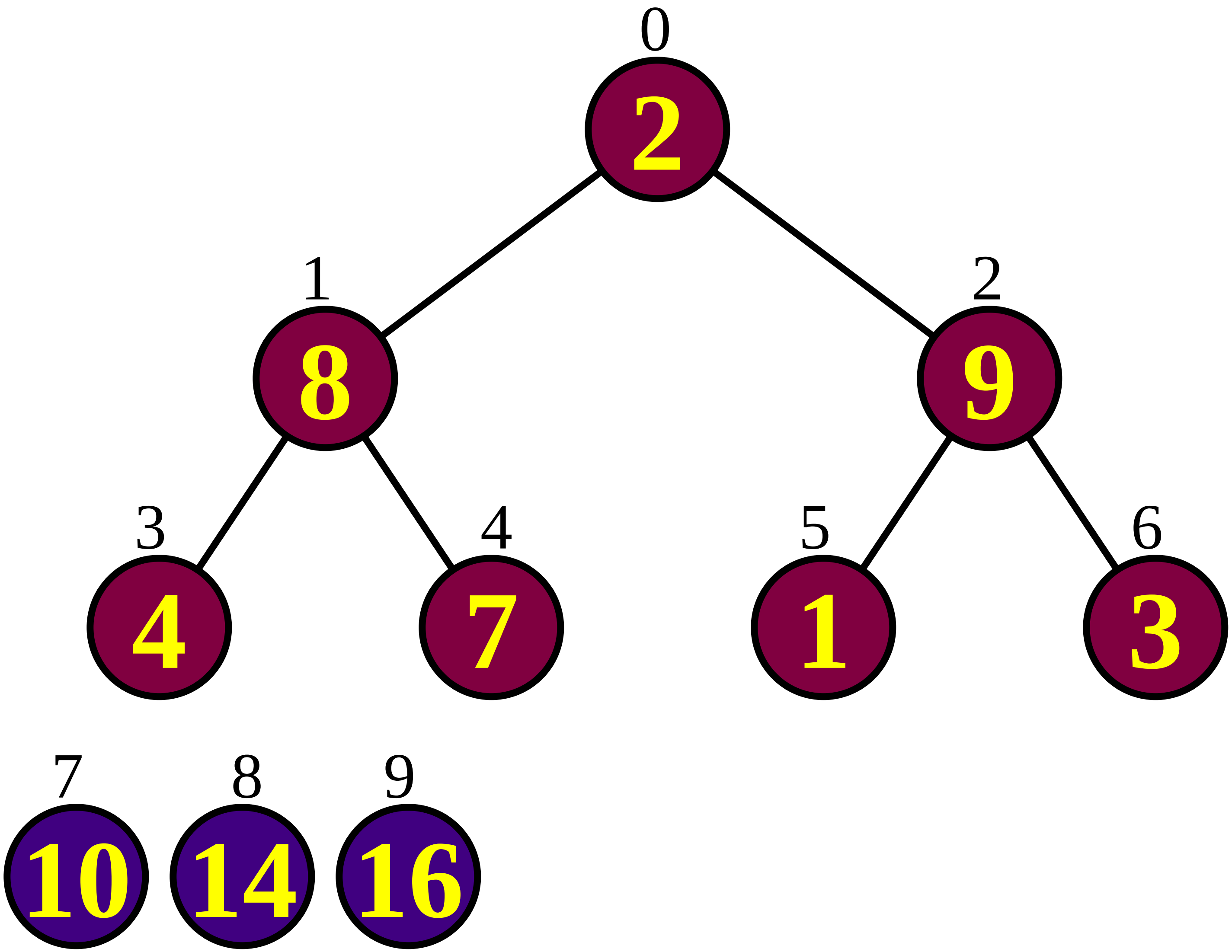
Heapsort



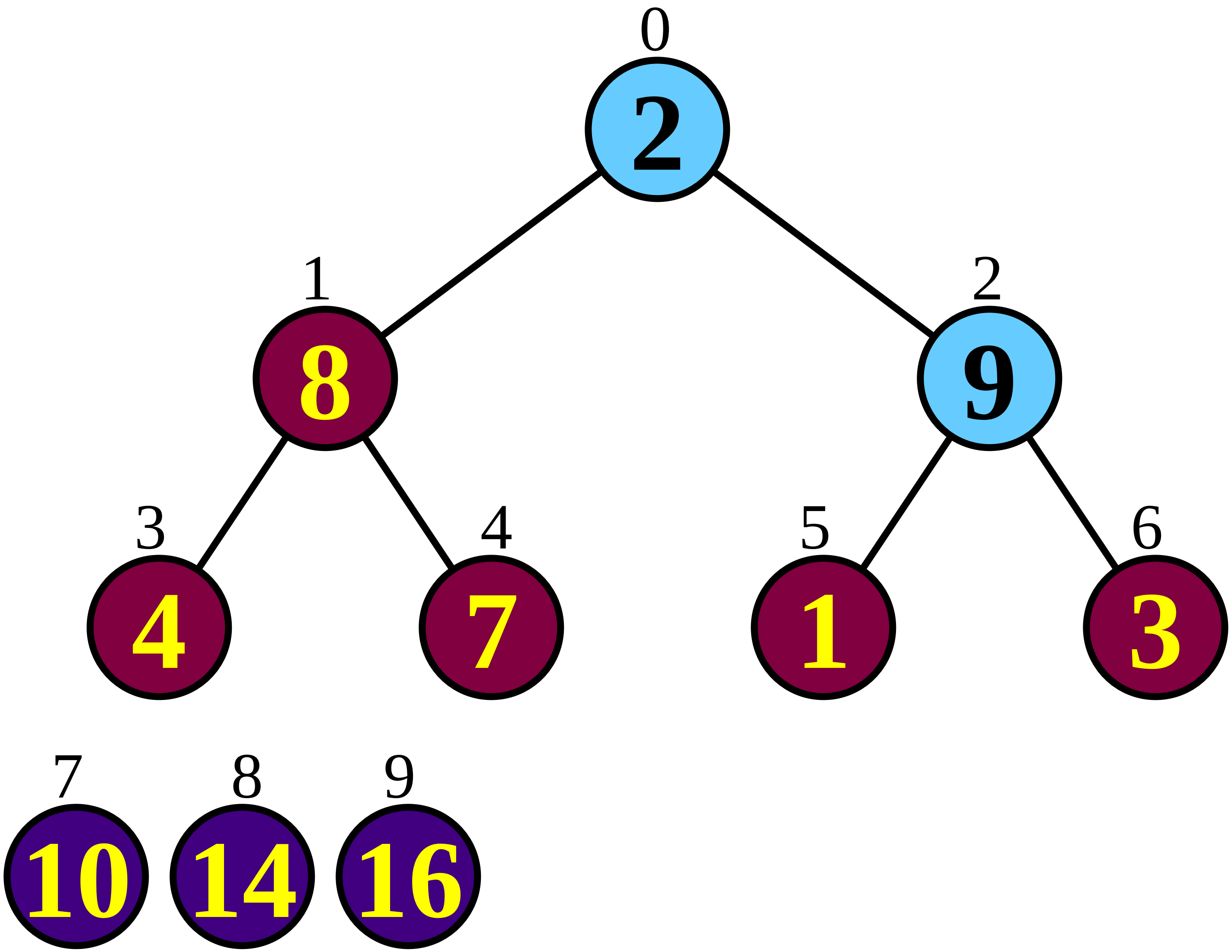
Heapsort



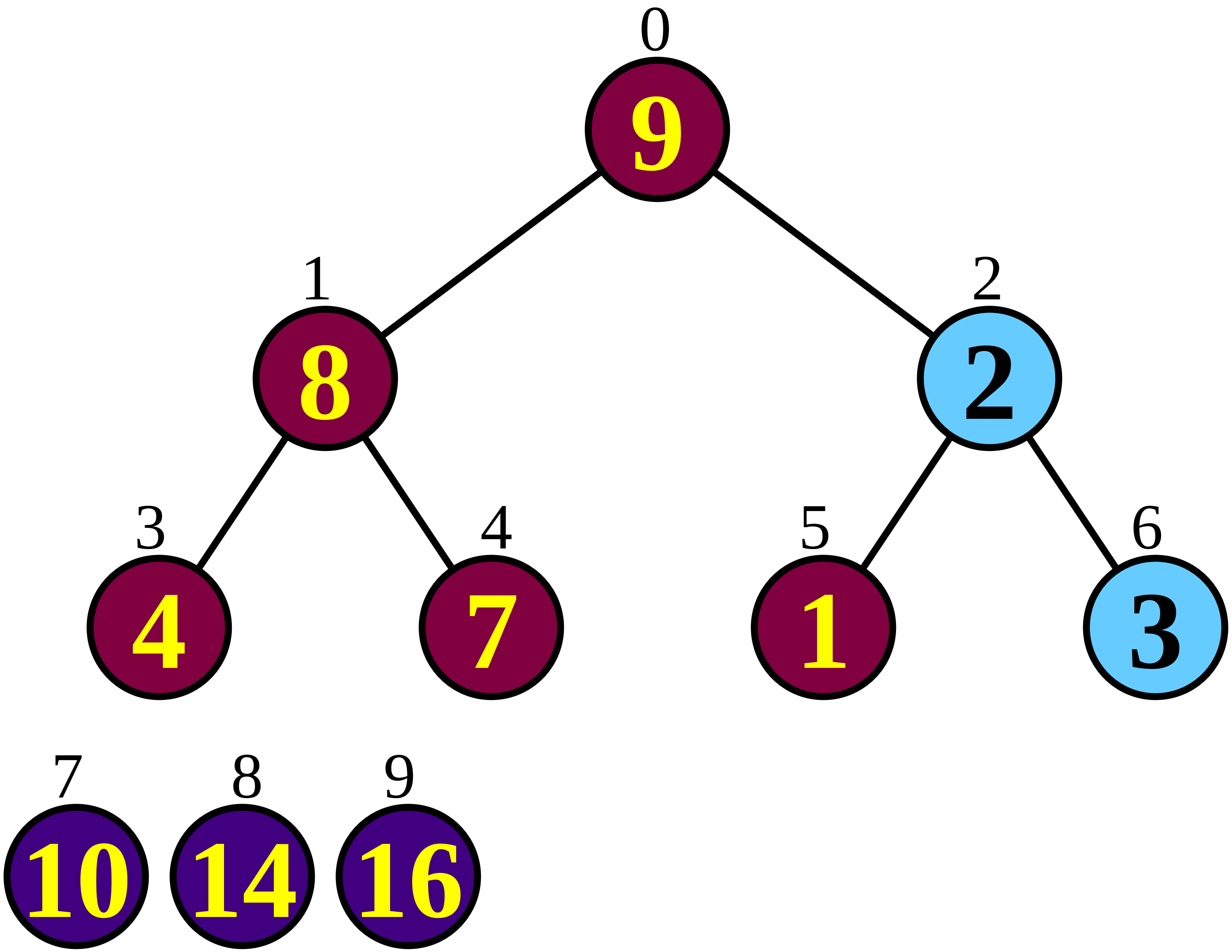
Heapsort



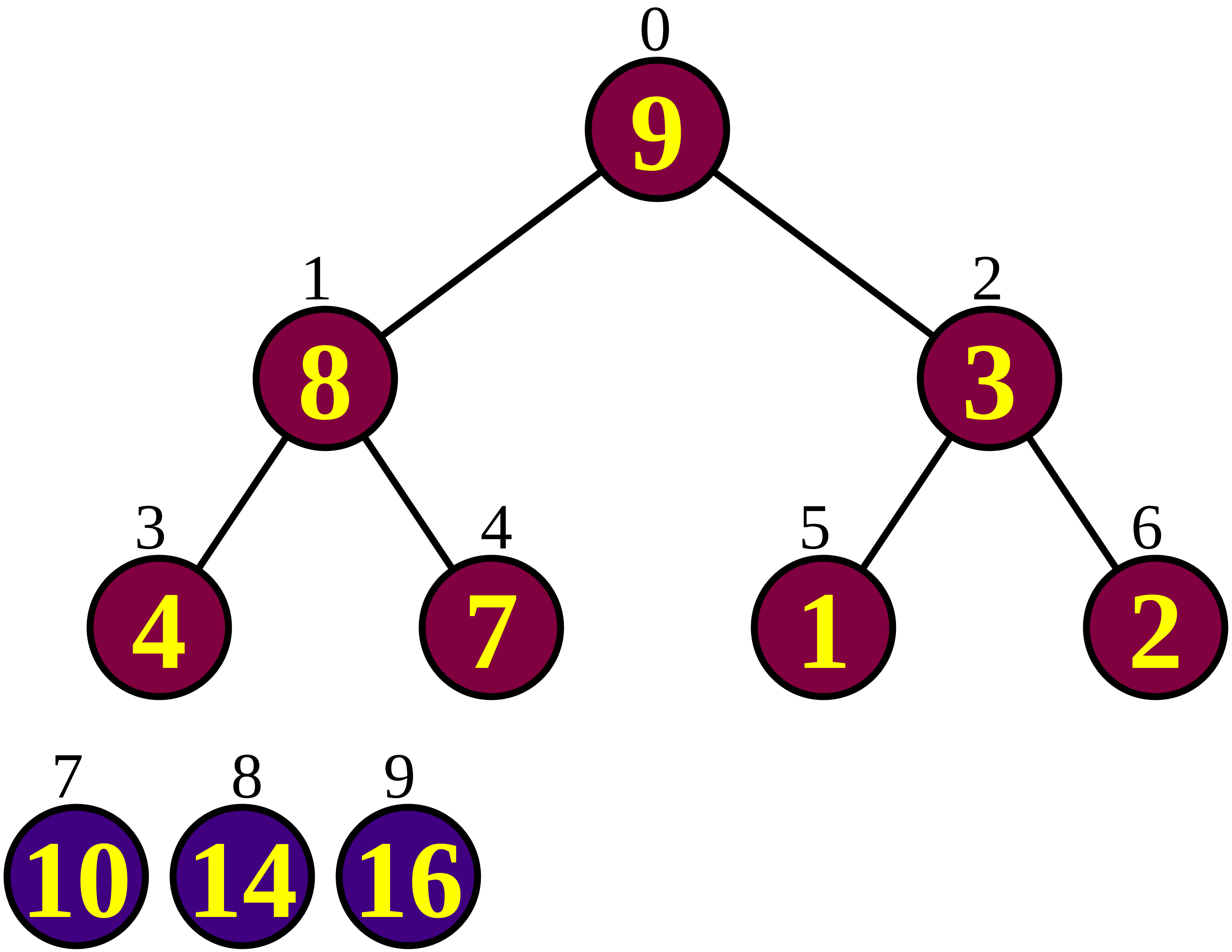
Heapsort



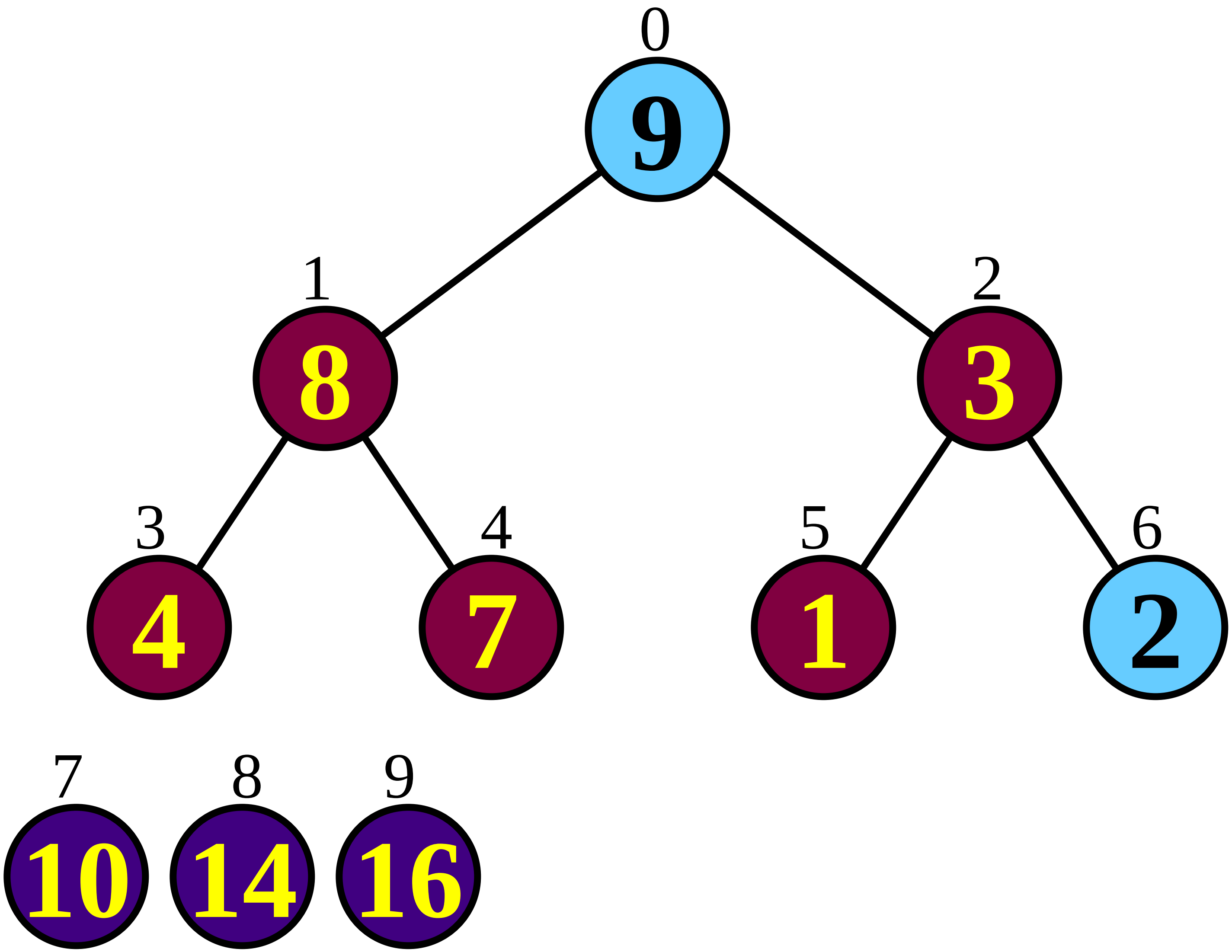
Heapsort



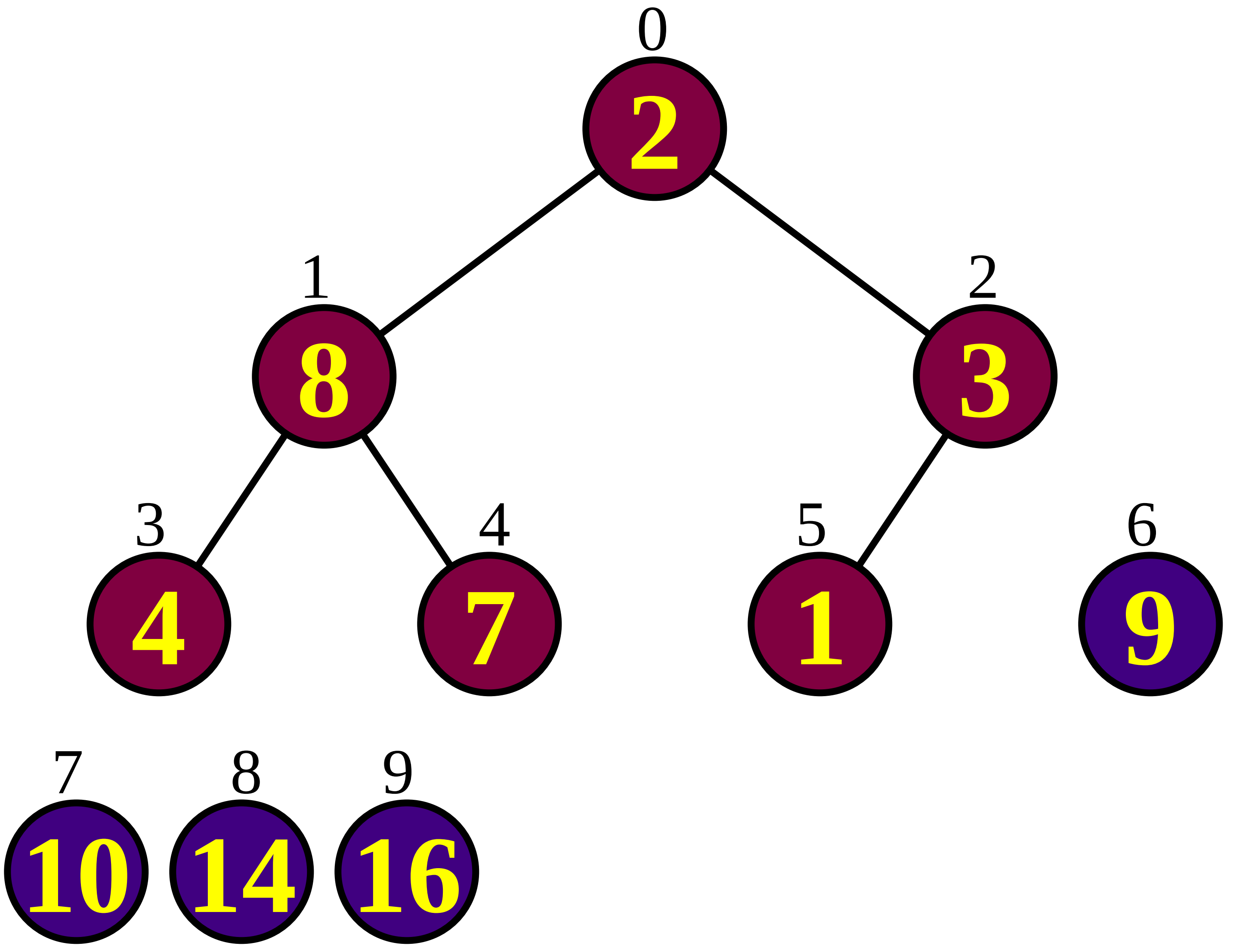
Heapsort



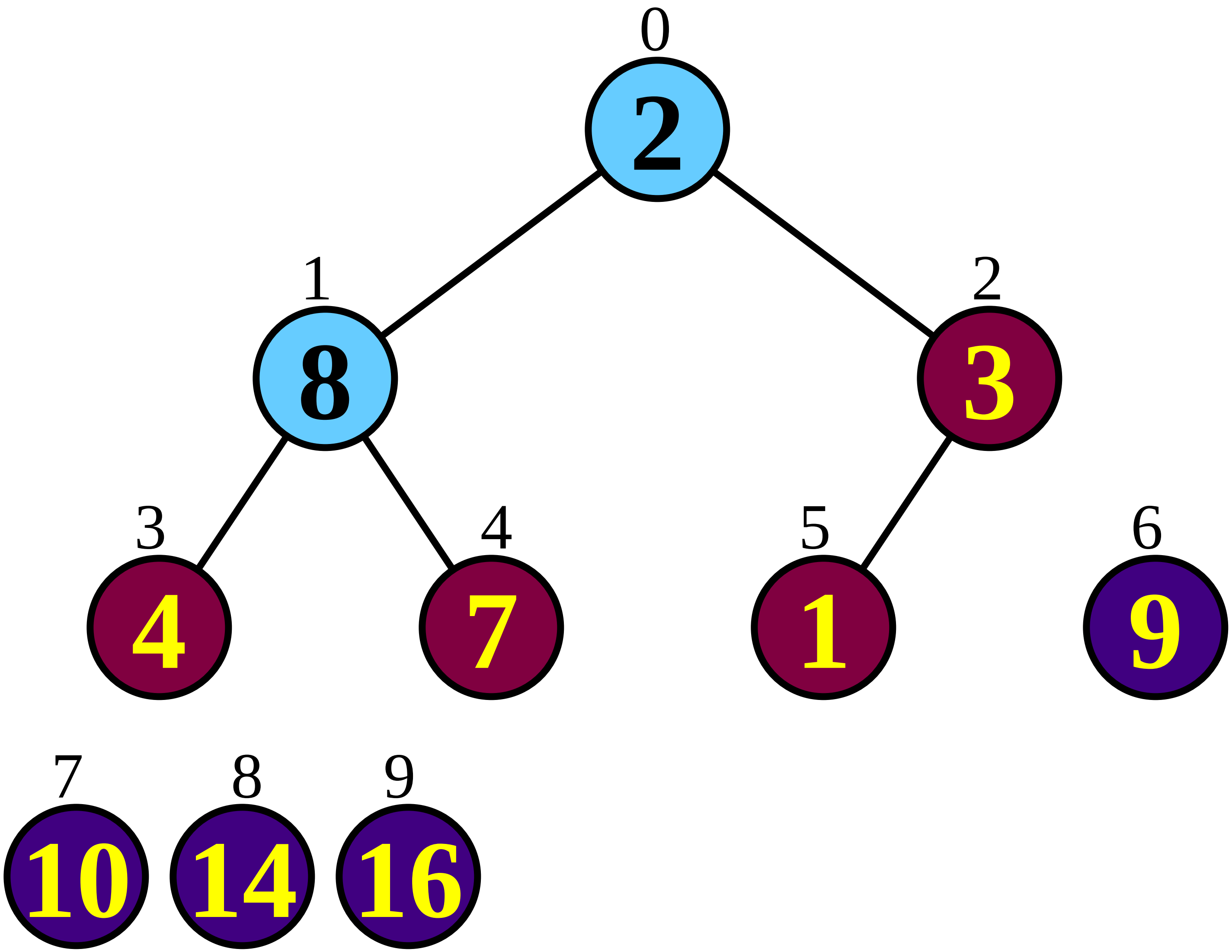
Heapsort



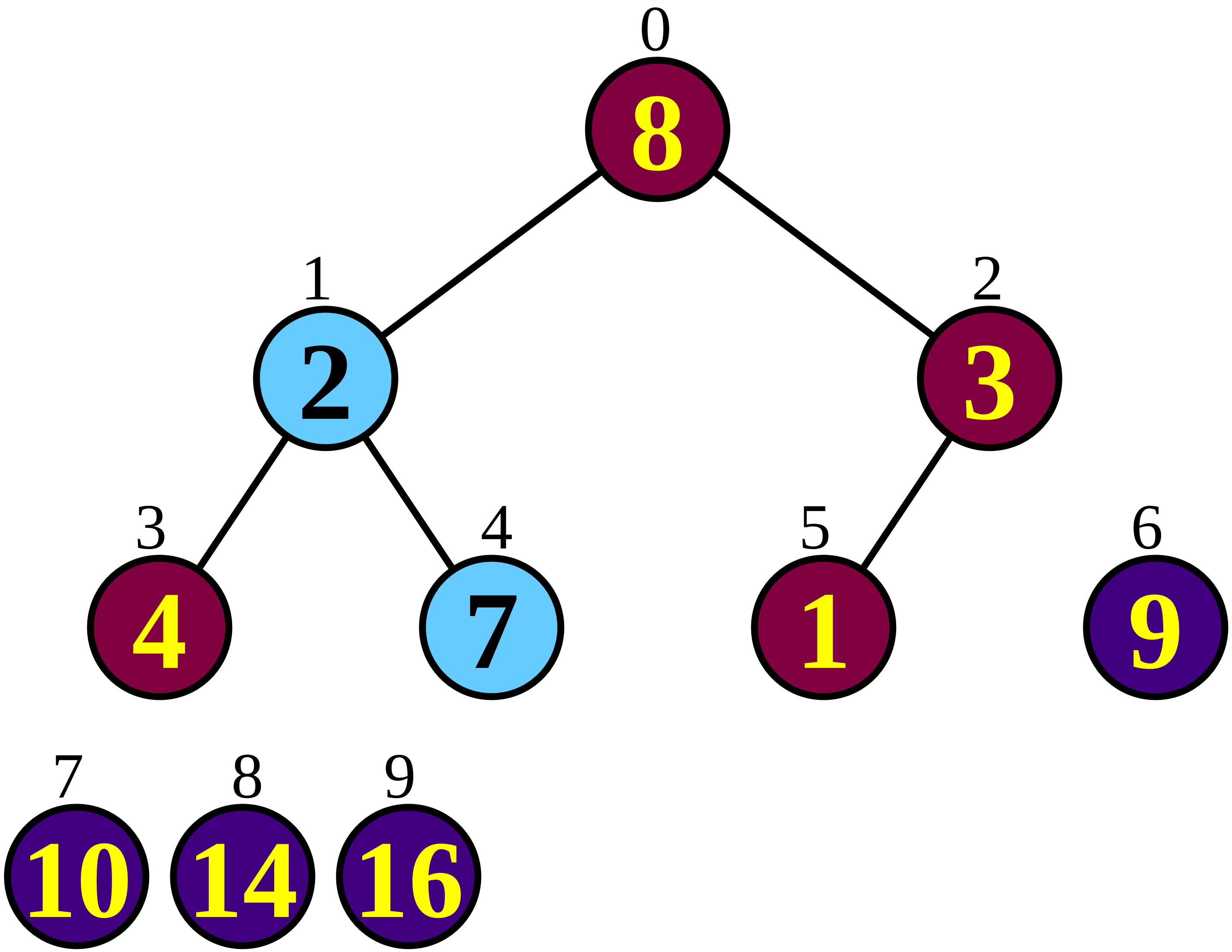
Heapsort



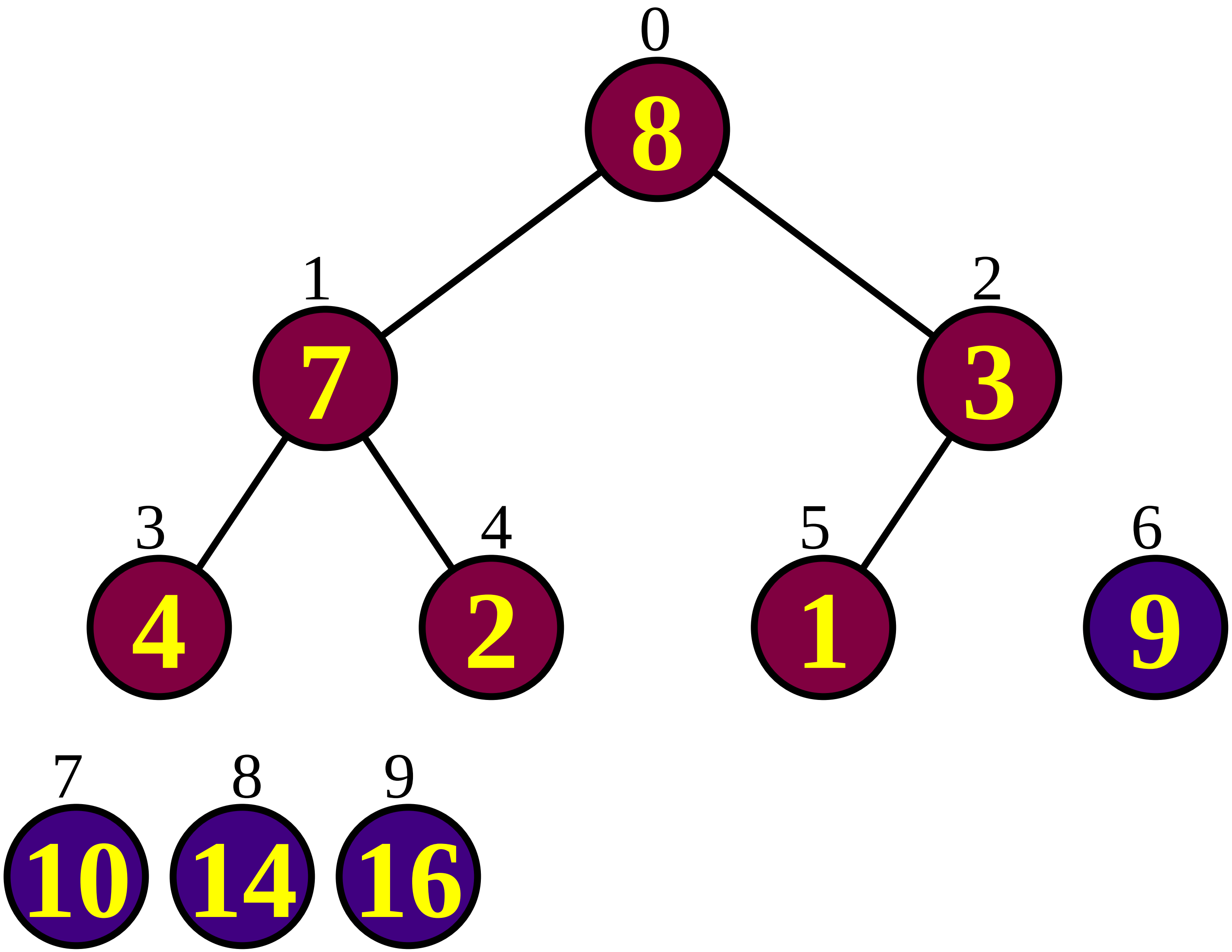
Heapsort



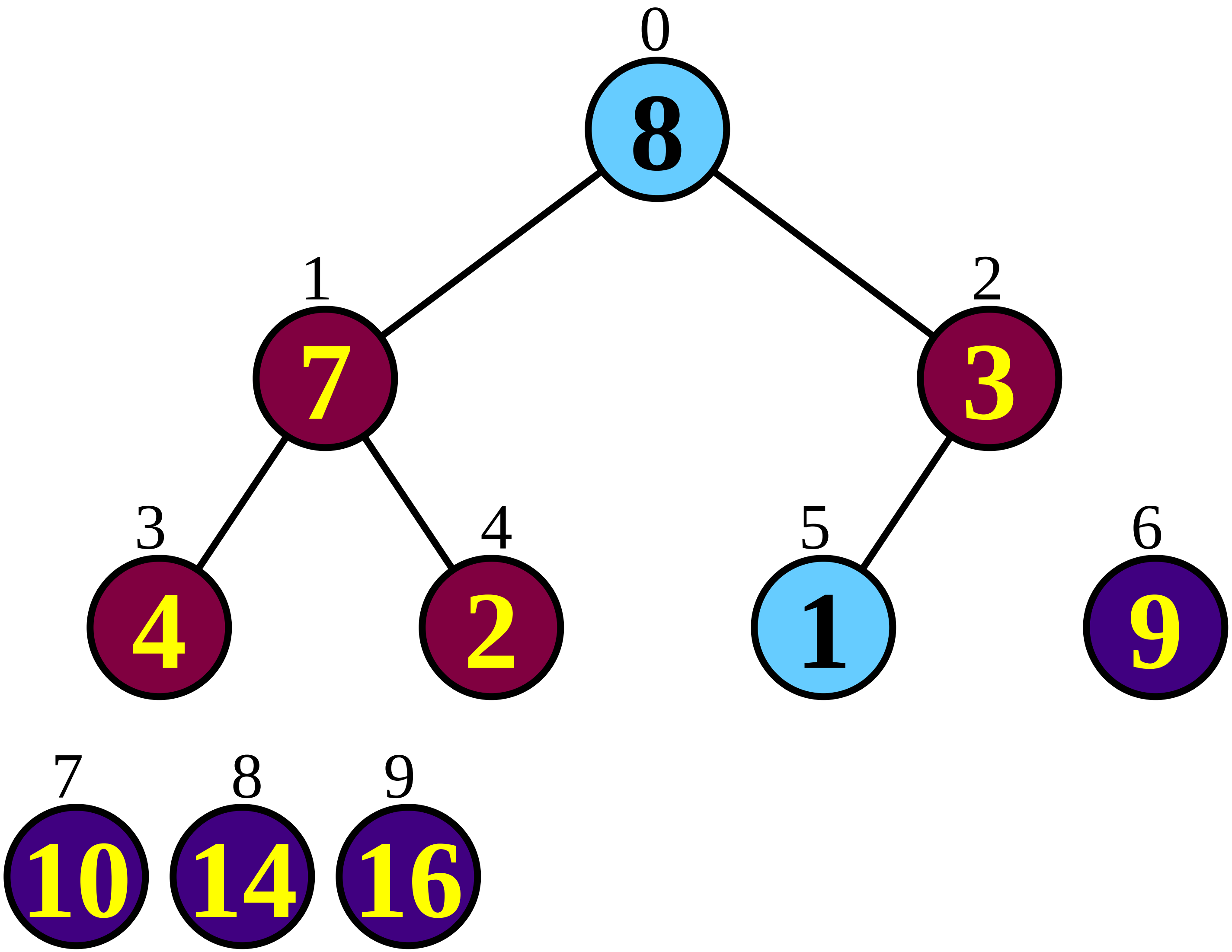
Heapsort



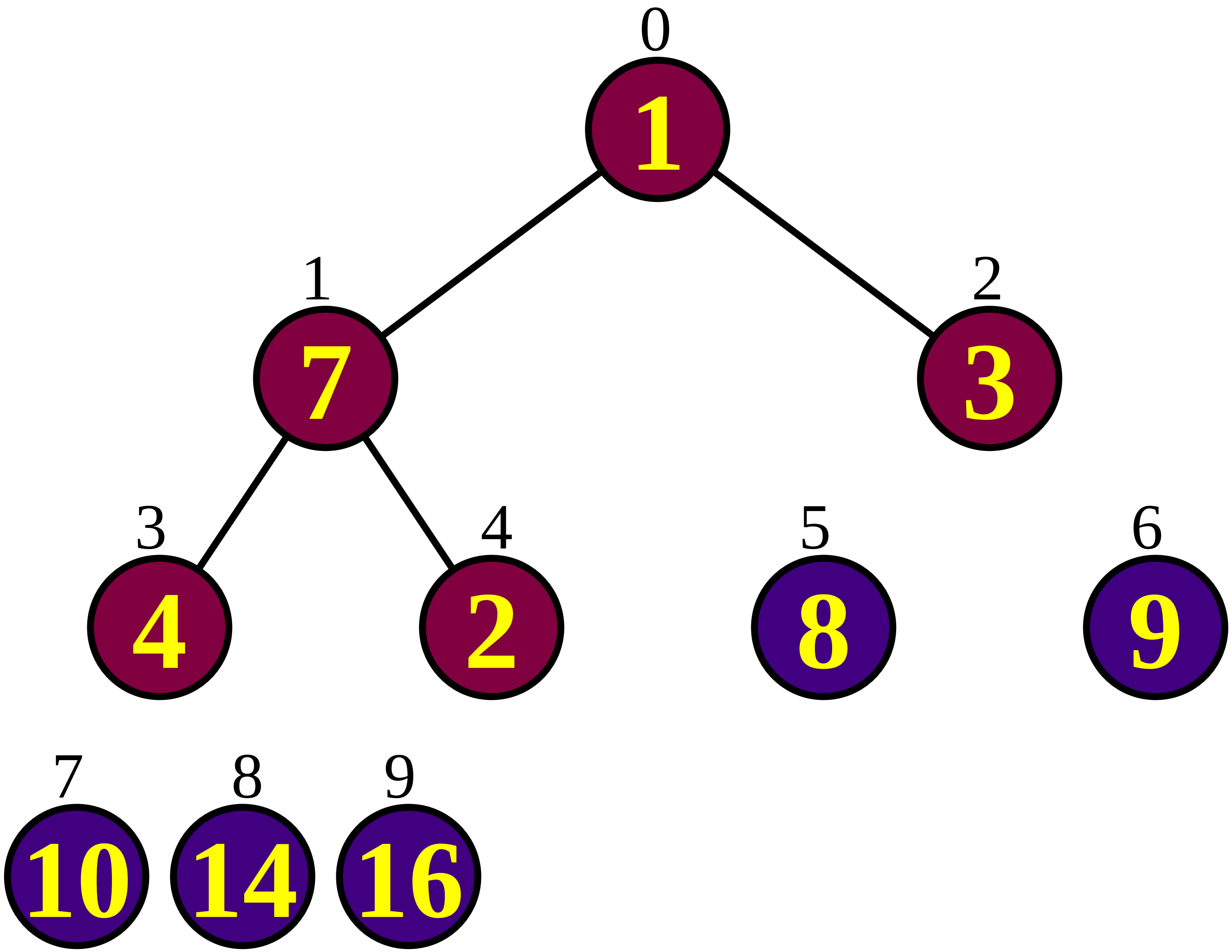
Heapsort



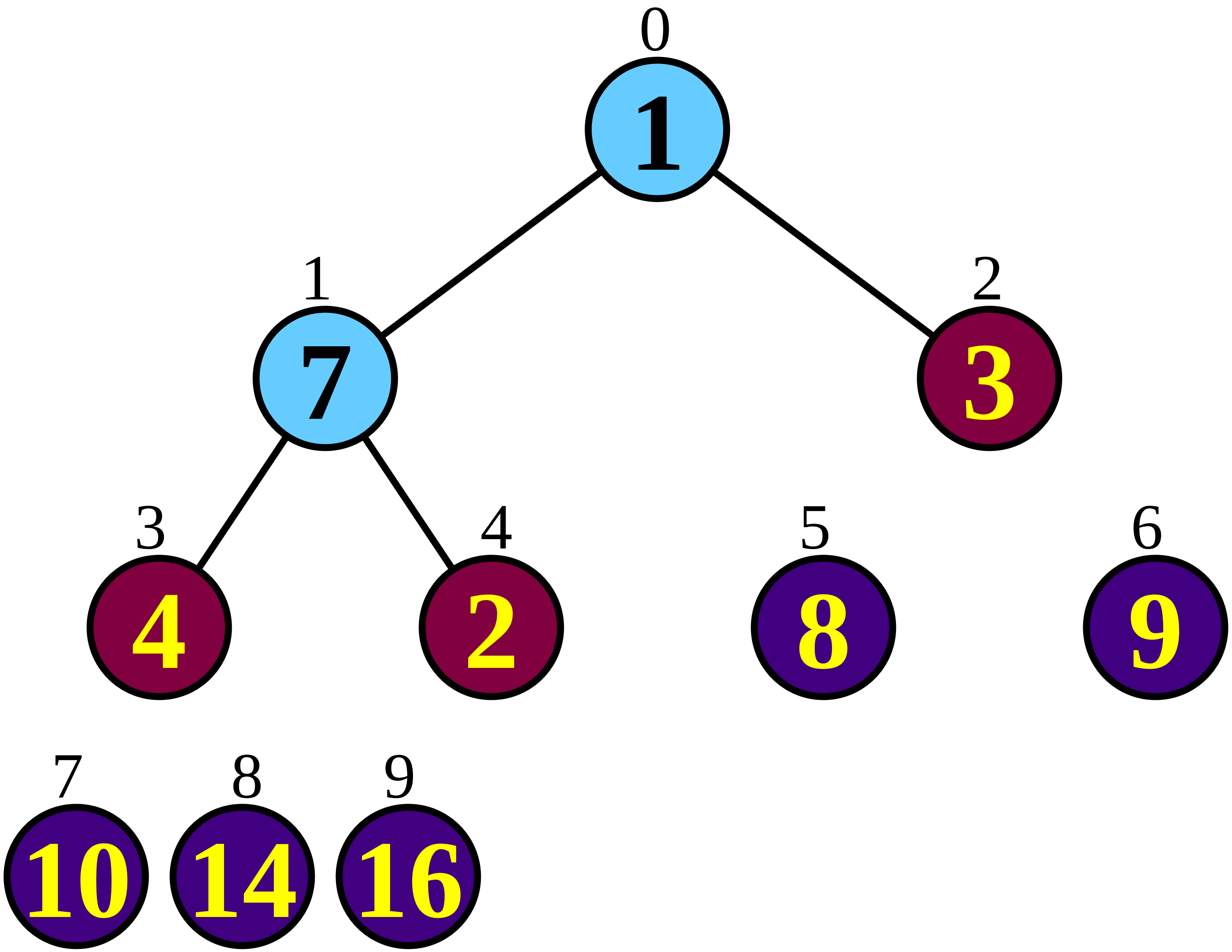
Heapsort



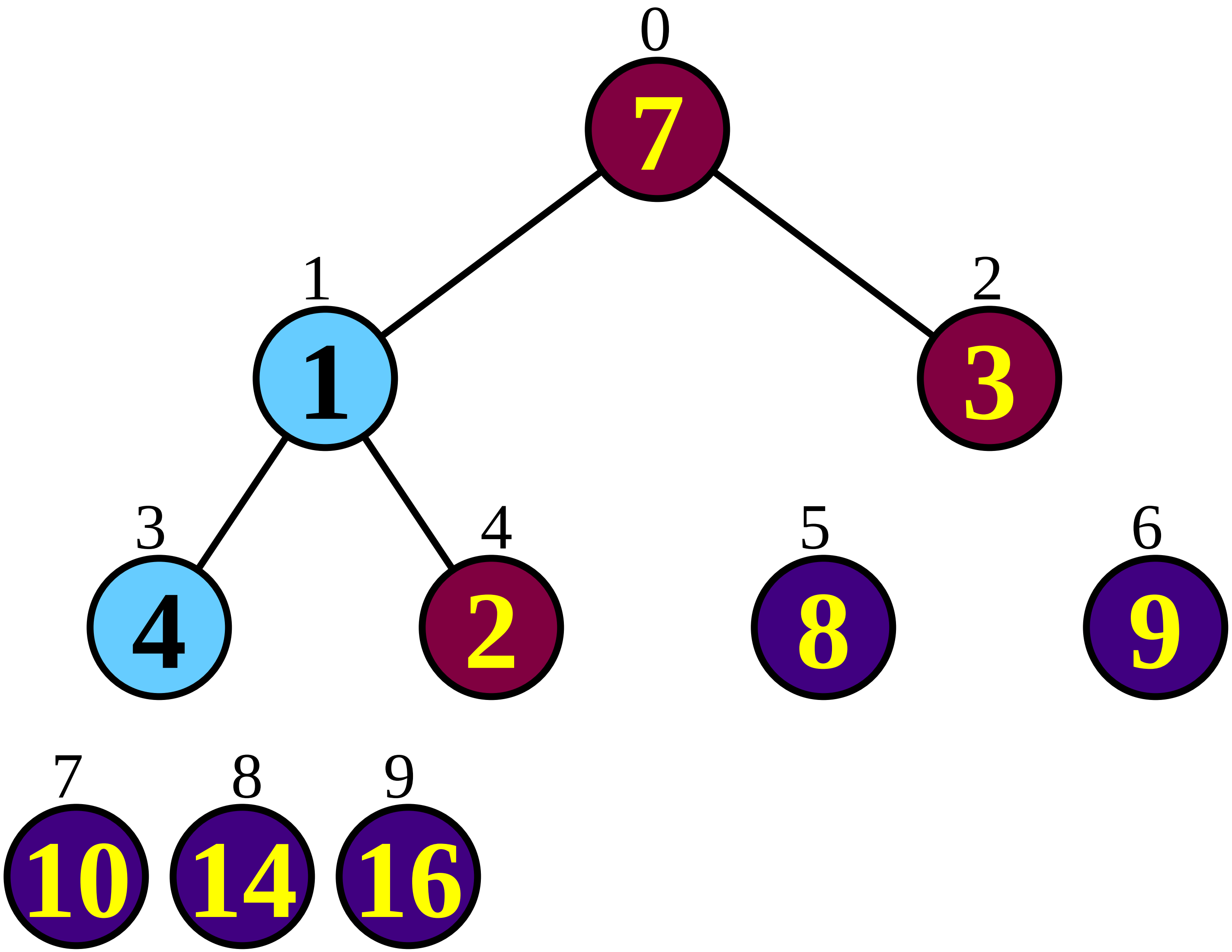
Heapsort



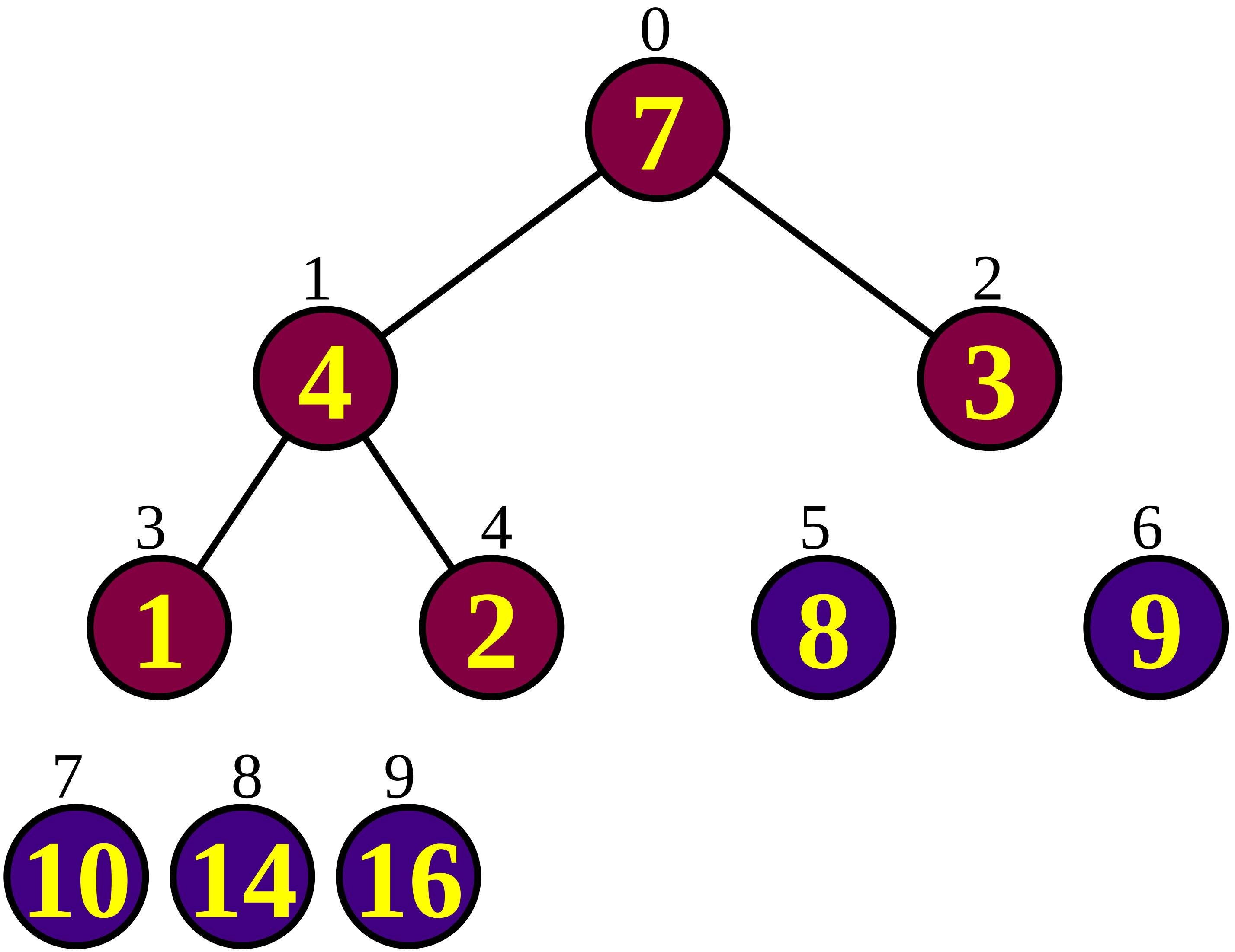
Heapsort



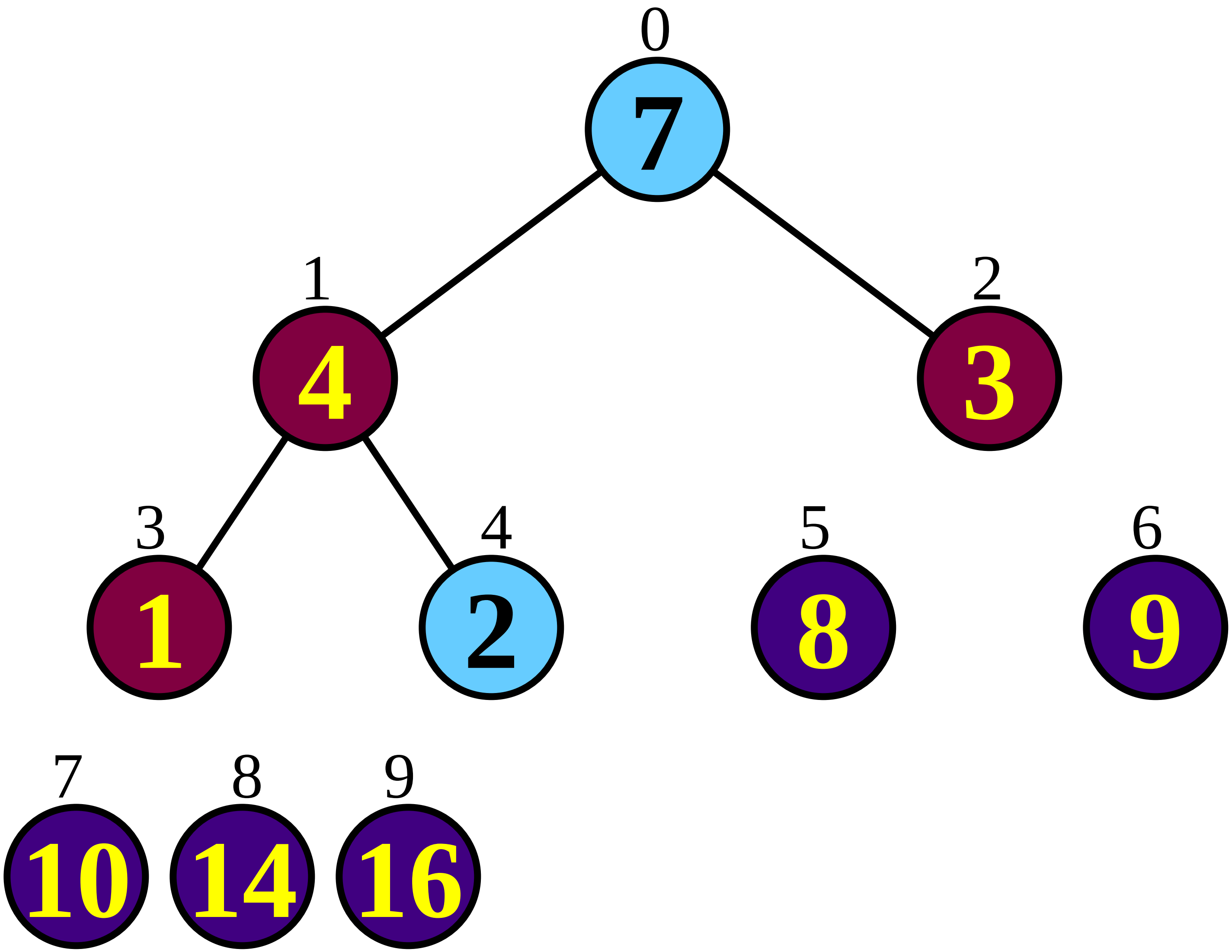
Heapsort



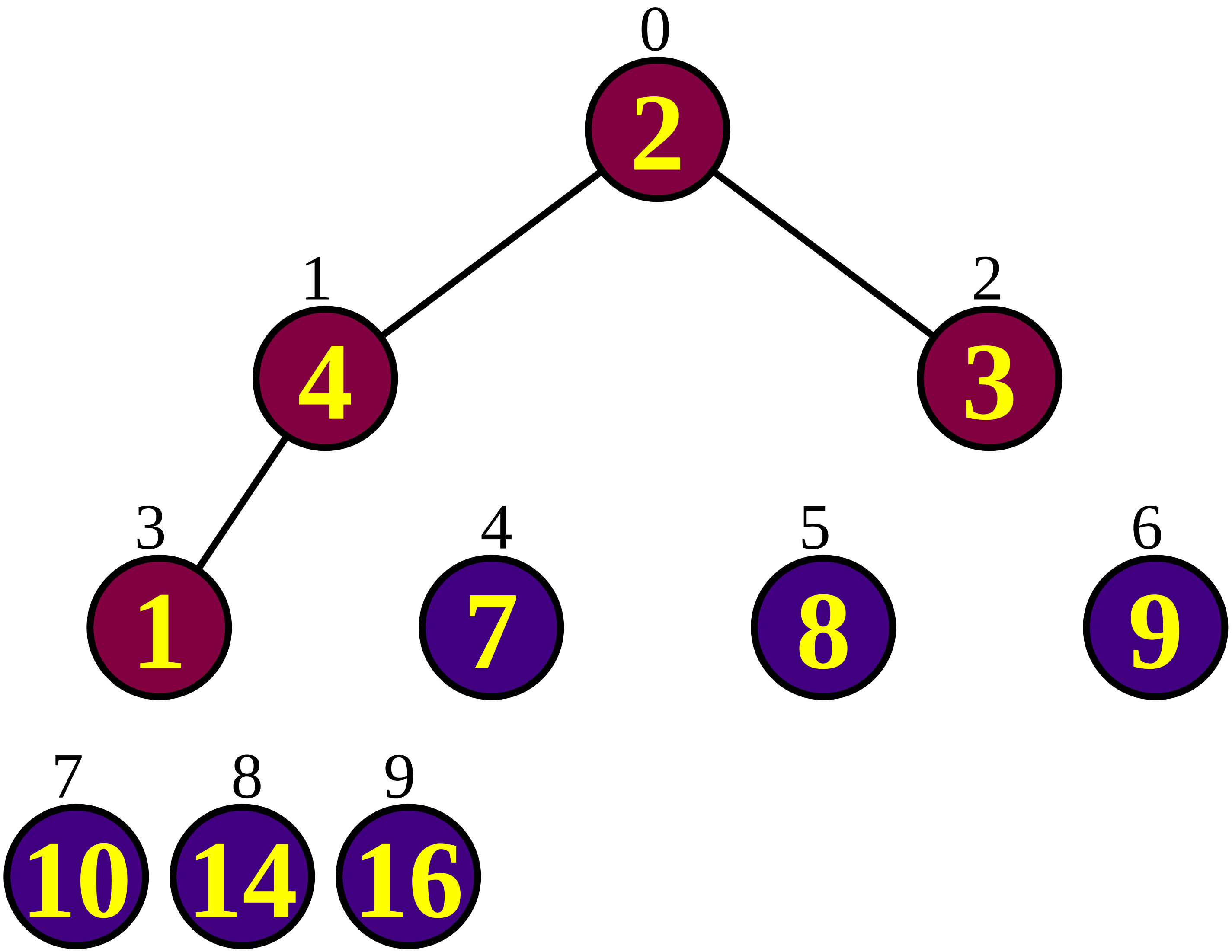
Heapsort



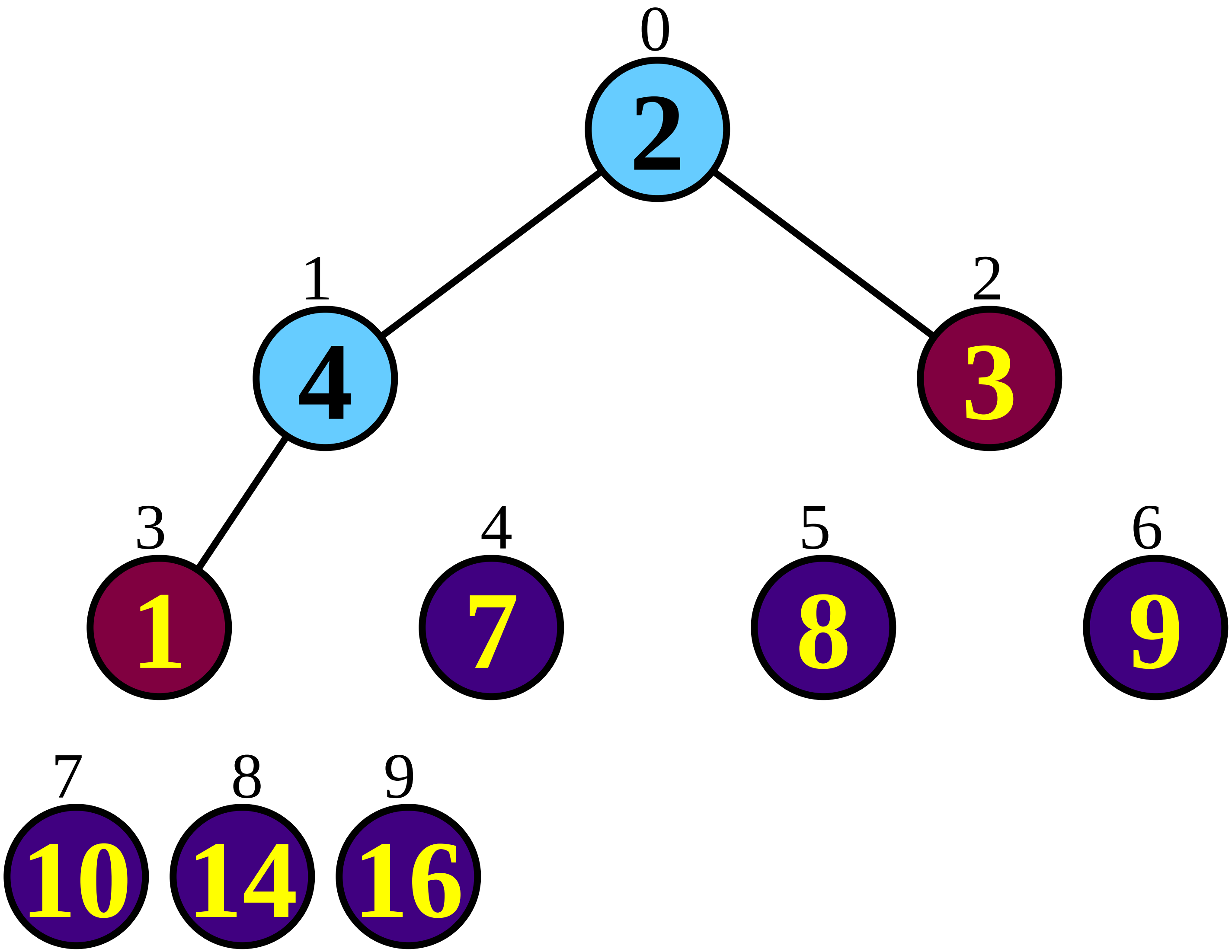
Heapsort



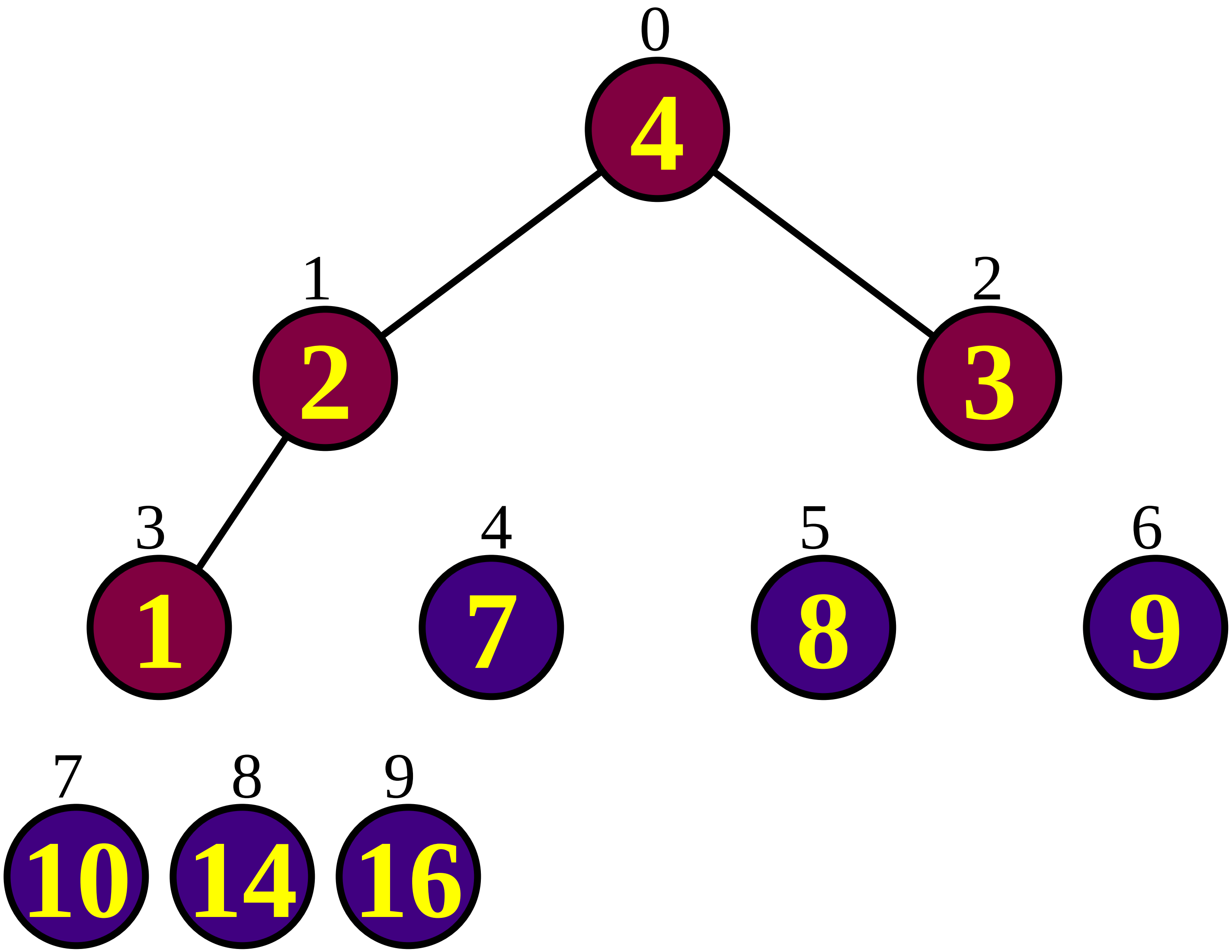
Heapsort



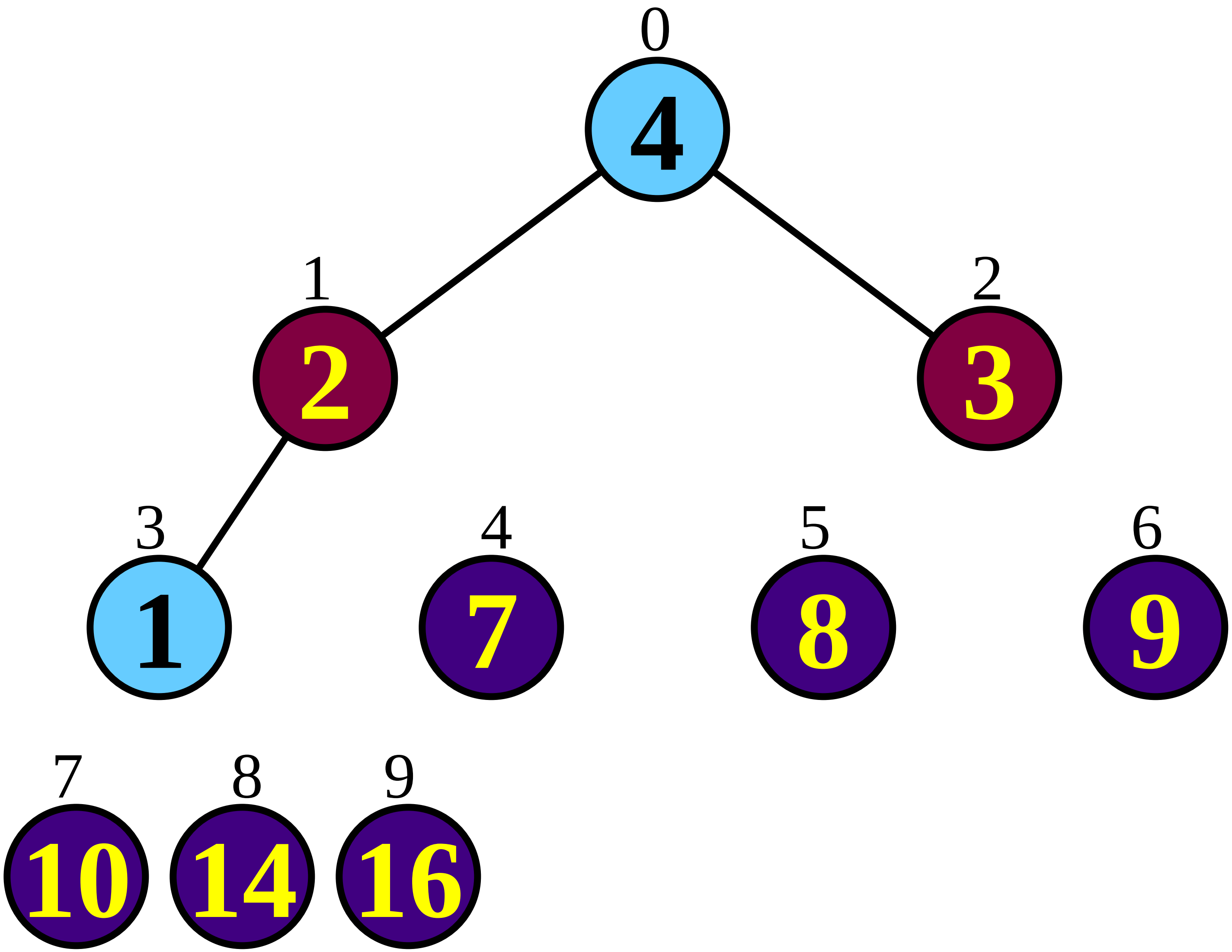
Heapsort



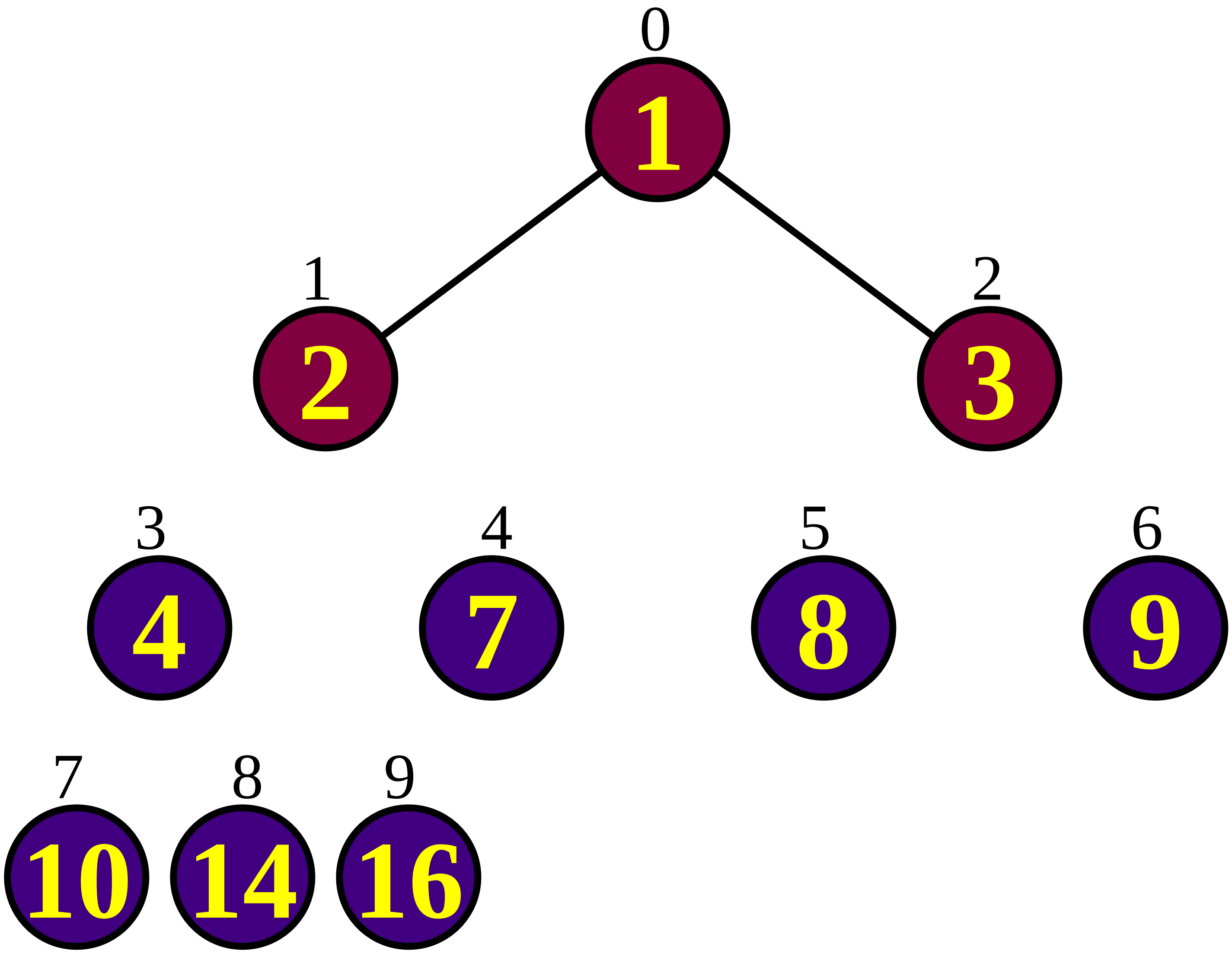
Heapsort



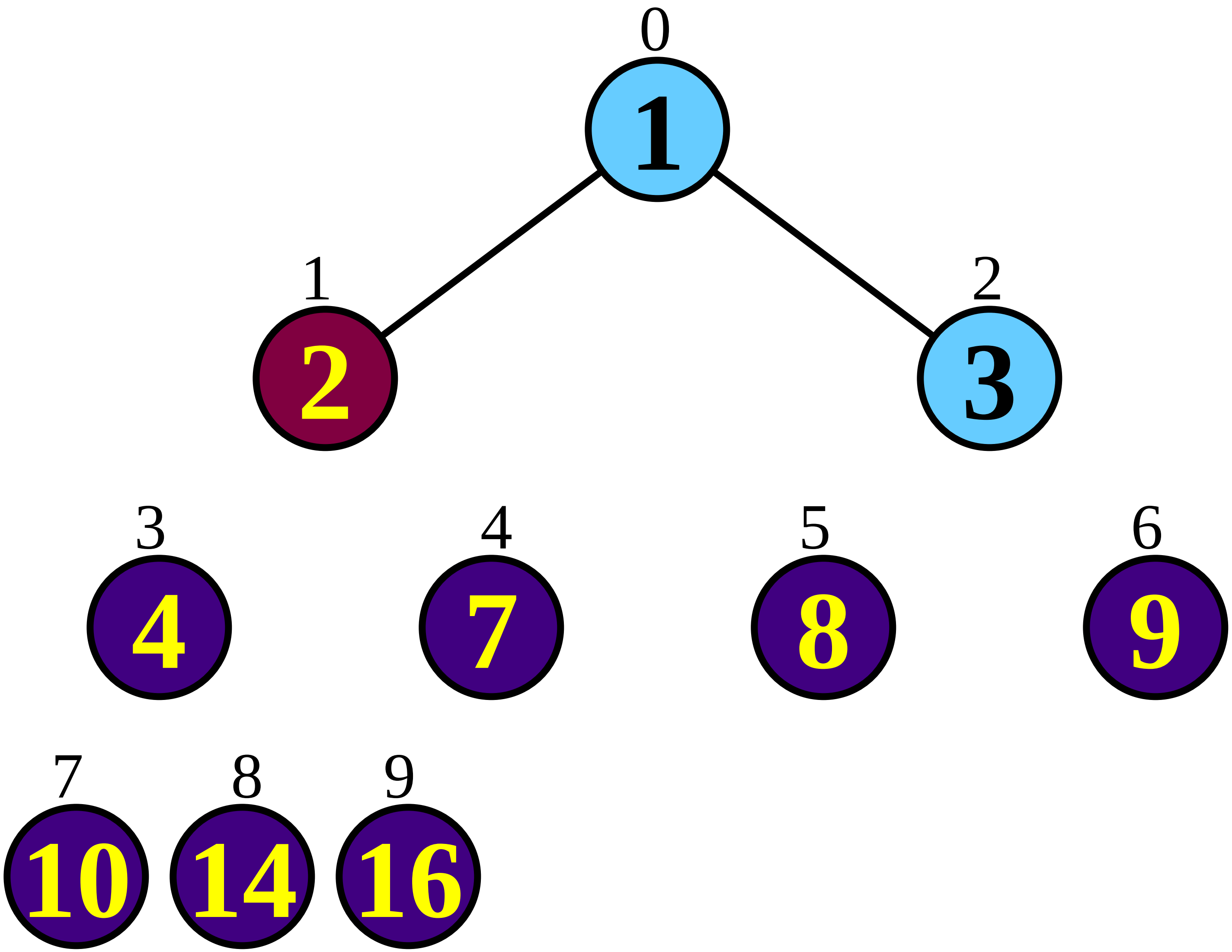
Heapsort



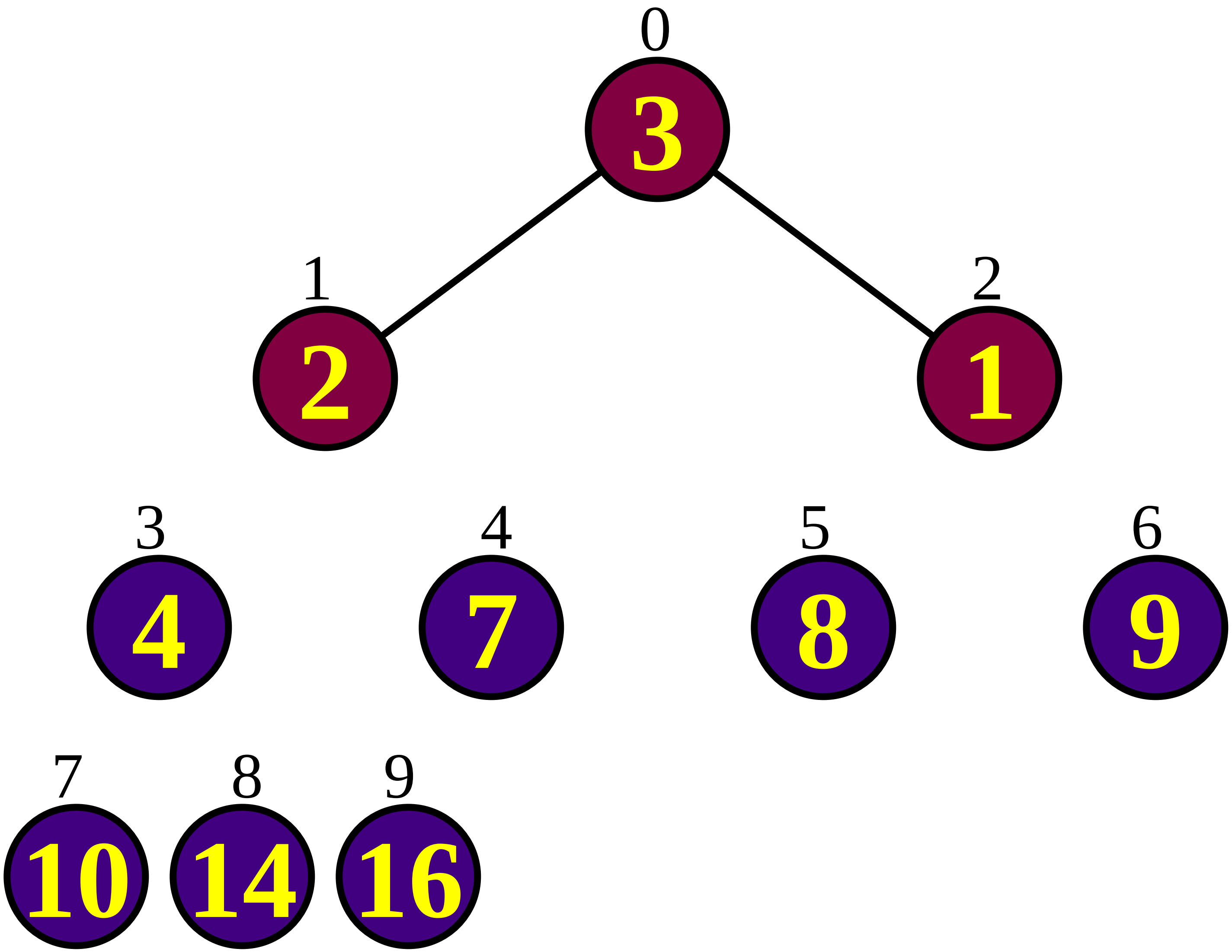
Heapsort



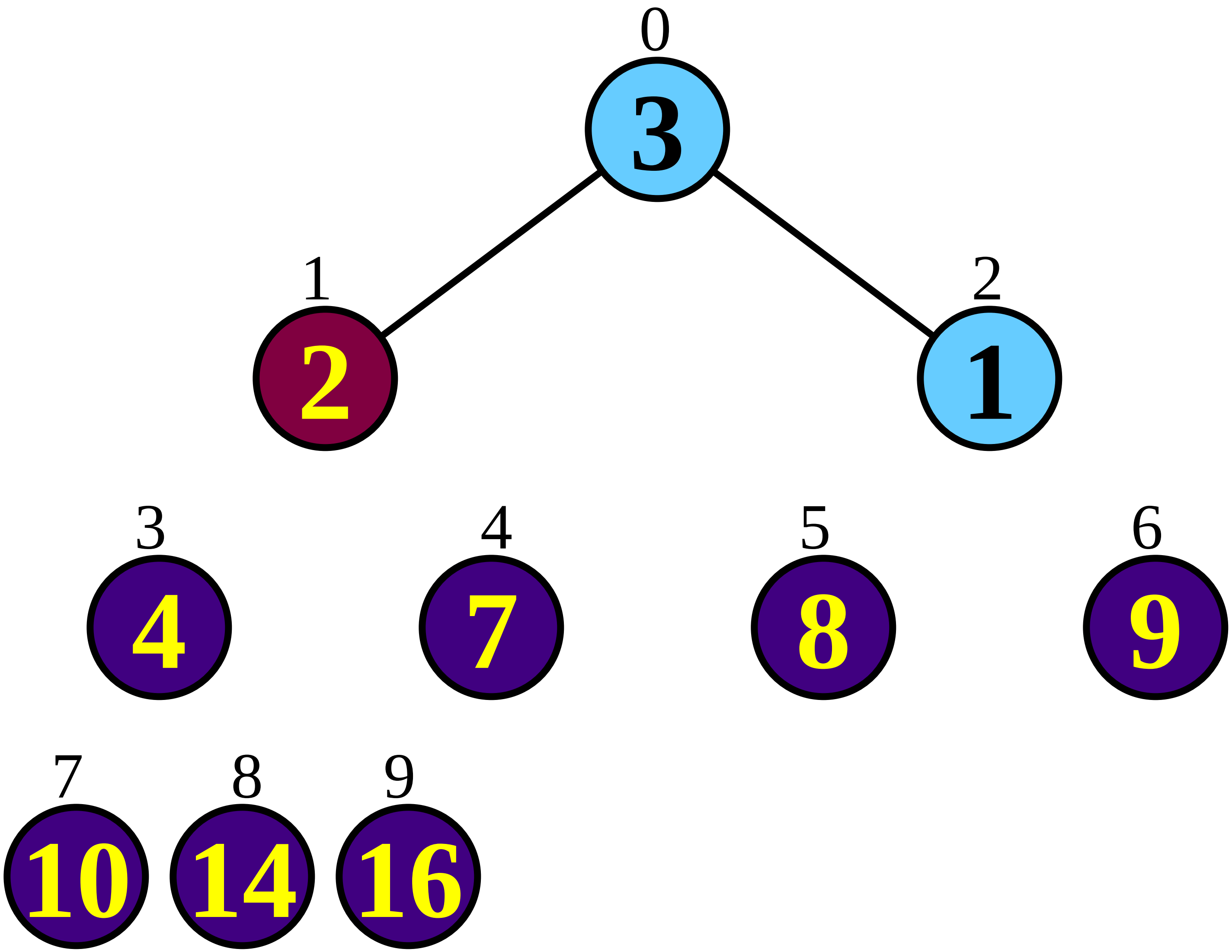
Heapsort



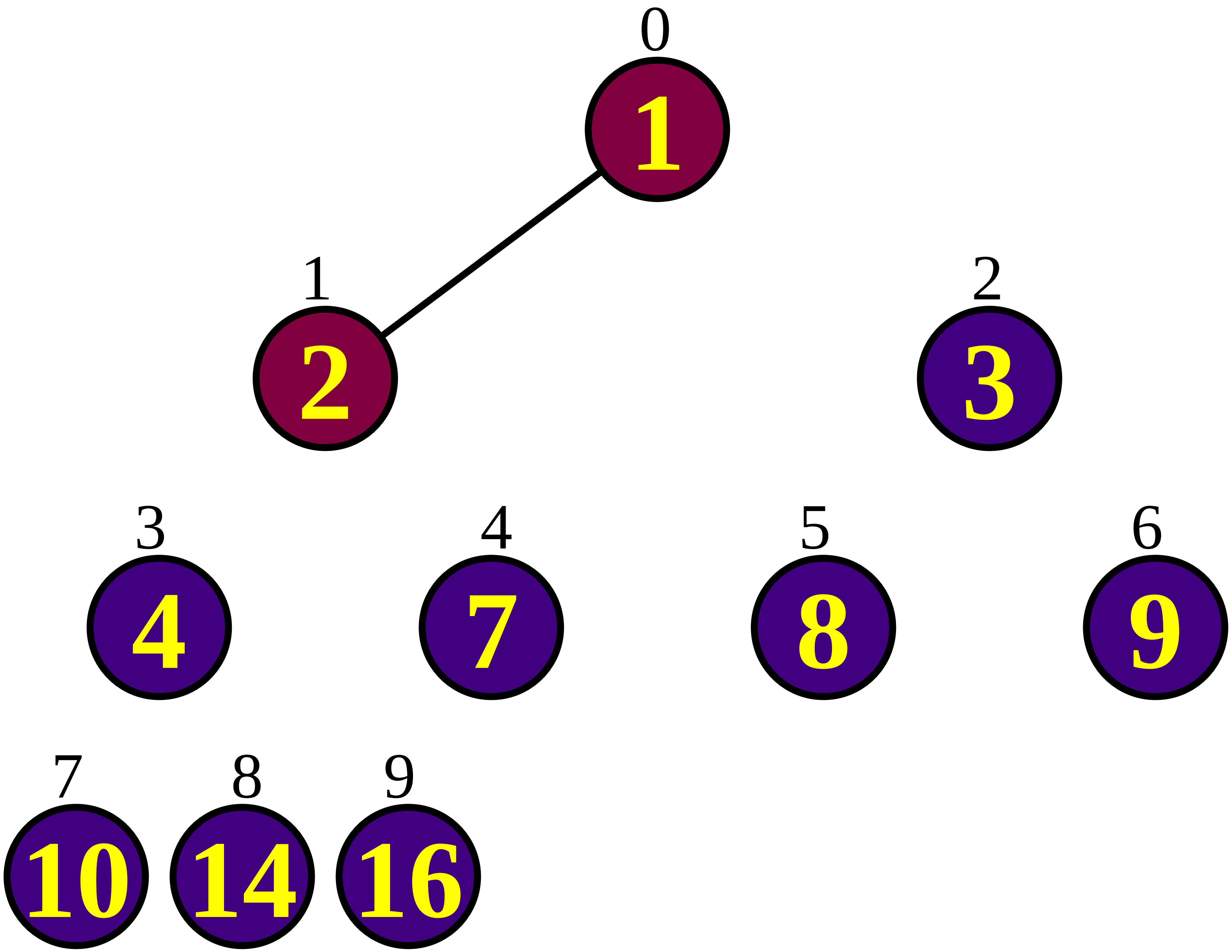
Heapsort



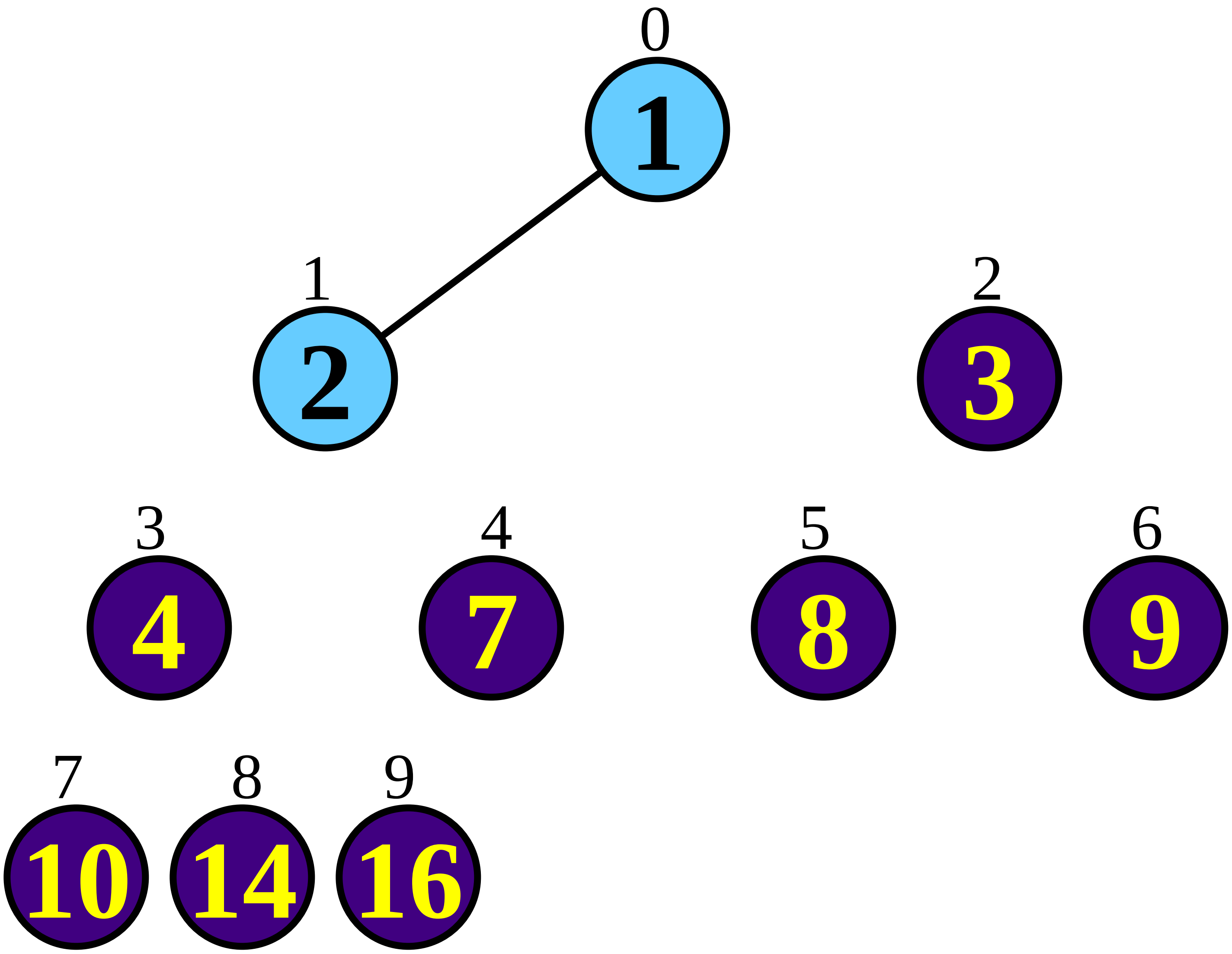
Heapsort



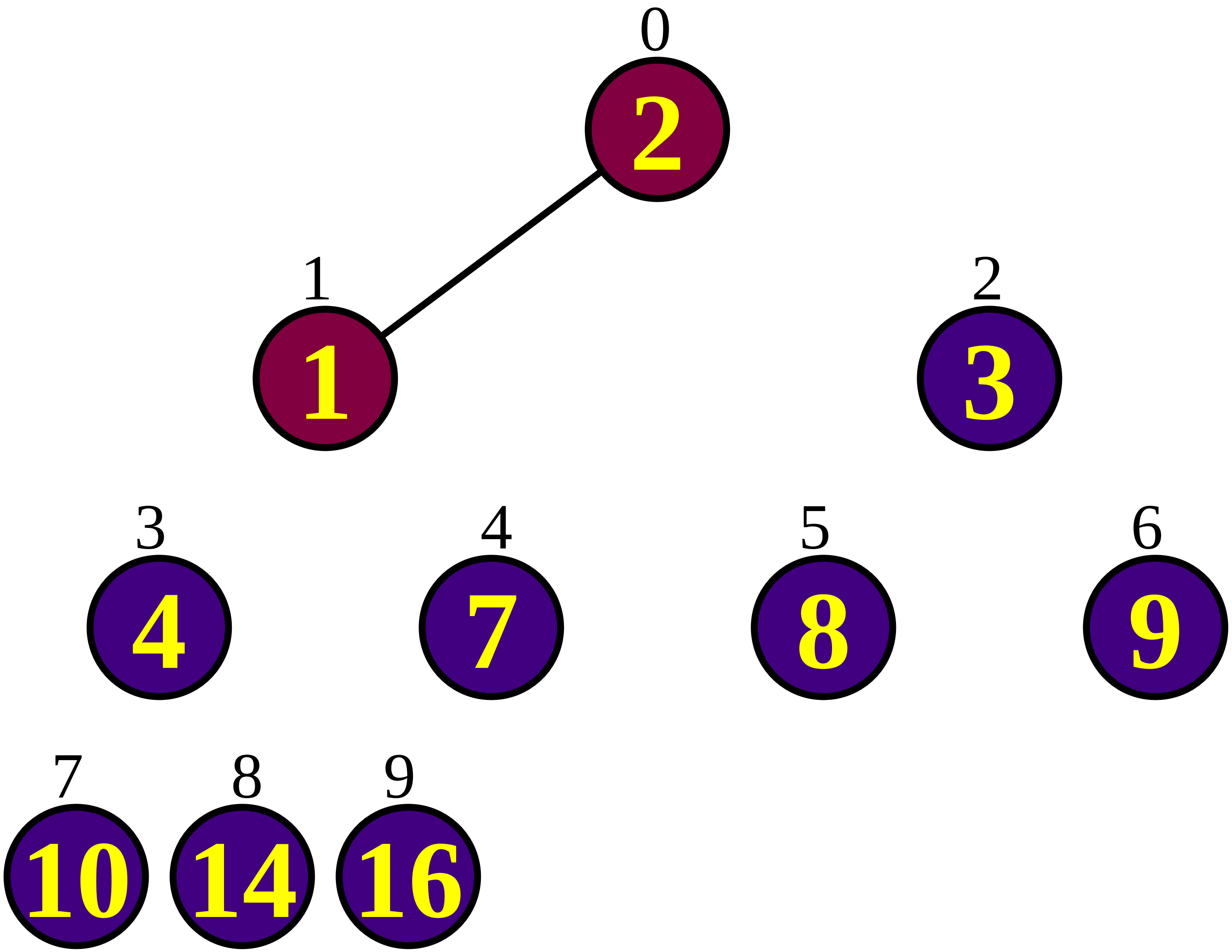
Heapsort



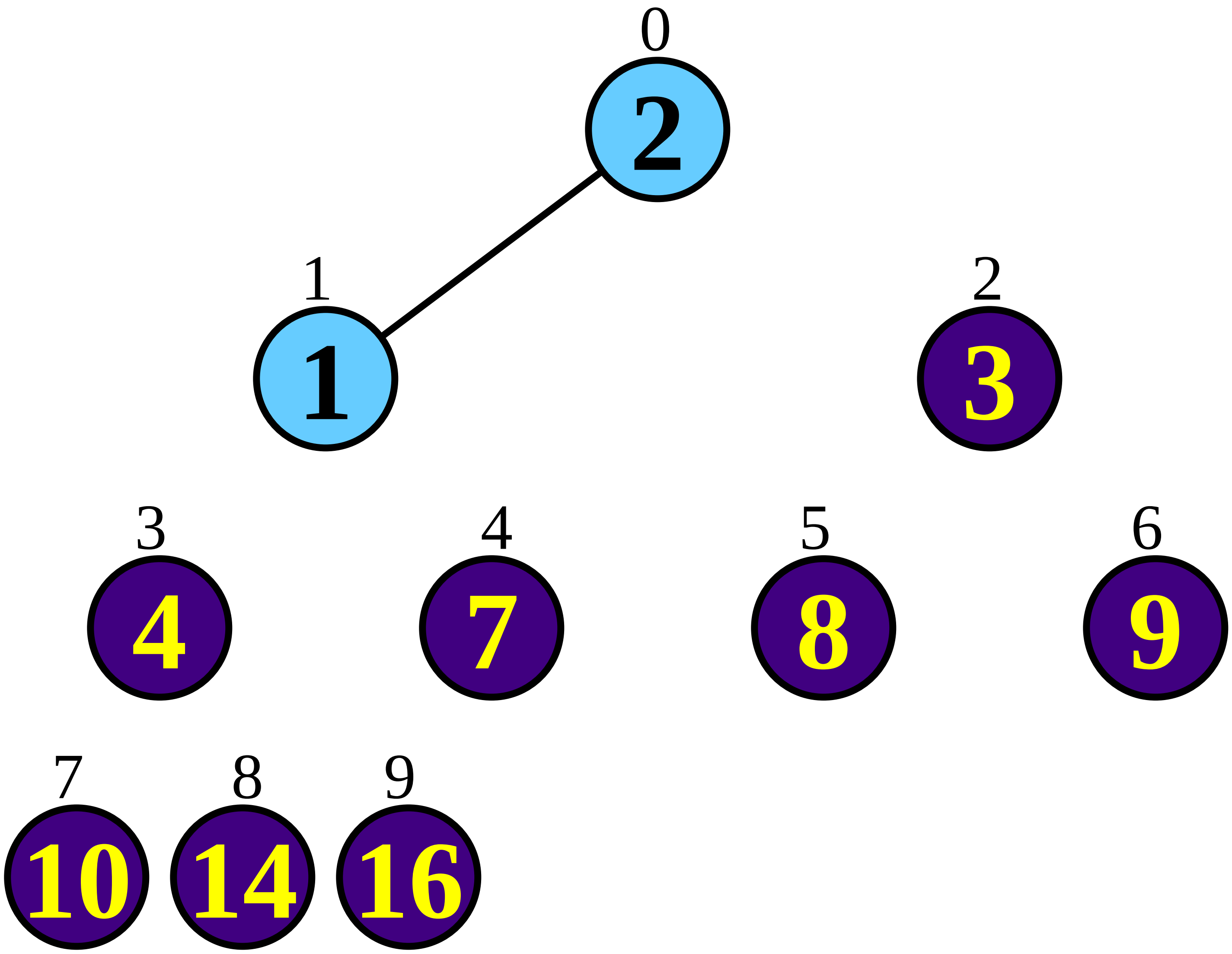
Heapsort



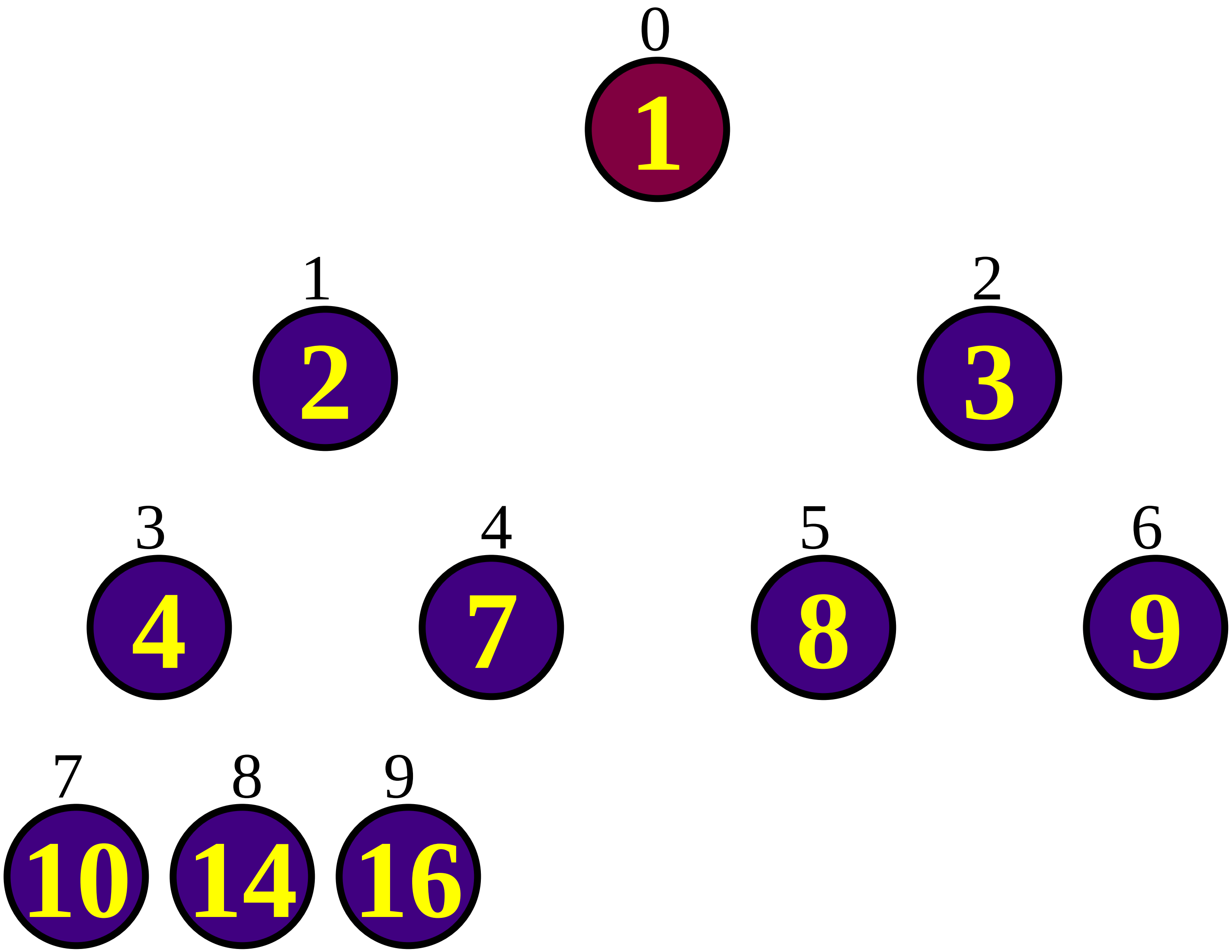
Heapsort



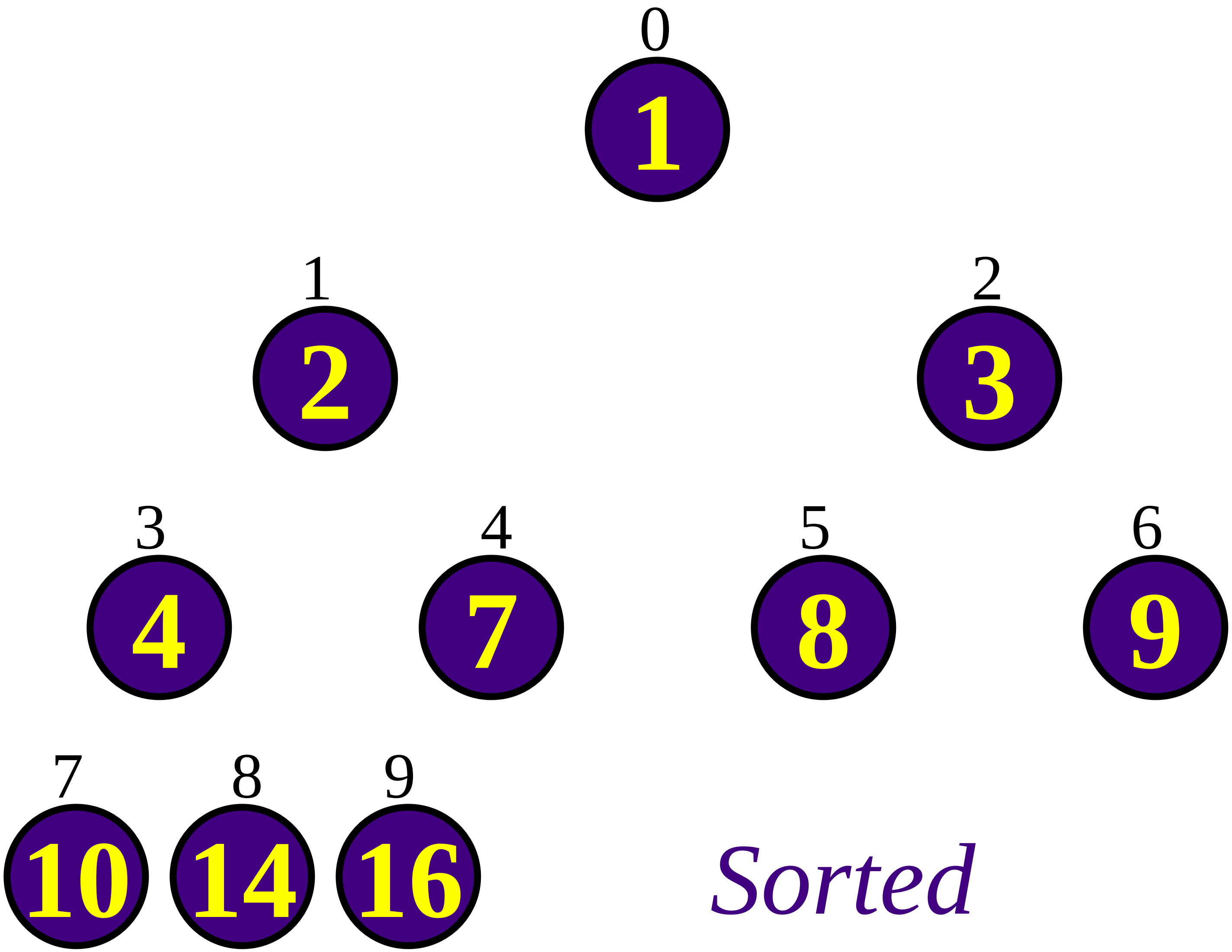
Heapsort



Heapsort



Heapsort



Sorted

히프 정렬 (1)

- HeapSort 알고리즘

```
heapSort(a[])
   n ← a.length-1;   // n은 히프 크기(원소의 수)
                     // a[0]은 사용하지 않고 a[1 : n]의 원소를 오름차순으로 정렬
   for (i ← n/2;  i ≥ 1;  i ← i-1) do  {     // 배열 a를 히프로 변환
       heapify(a, i, n);                     // i는 내부 노드
   for (i ← n-1;  i ≥ 1;  i ← i-1) do  {     // 배열 a[]를 오름차순으로 정렬
       temp ← a[1];      // a[1]은 제일 큰 원소
       a[1] ← a[i+1];    // a[1]과 a[i+1]을 교환
       a[i+1] ← temp;
       heapify(a, 1, i);
   }
end HeapSort()
```

- Heapify()를 호출하여 배열 a[1 : n]을 히프 구조로 변환
- 원소를 교환하여 최대 원소 저장
- Heapify()를 호출하여 나머지 원소를 히프로 재구성

히프 정렬 (2)

- Heapify 알고리즘

```
heapify(a[], h, m)
        // 루트 h를 제외한 h의 왼쪽 서브트리와 오른쪽 서브트리는 히프
        // 현재 시점으로 노드의 최대 레벨 순서 번호는 m
  for ($j \leftarrow 2*h$;  $j \le m$;  $j \leftarrow 2*j$) do  {
    if ($j < m$) then
      if ($a[j] < a[j+1]$) then $j \leftarrow j+1$;    // j는 값이 큰 왼쪽 또는 오른쪽 자식 노드
    if ($a[h] \ge a[j]$) then exit
    else $a[j/2] \leftarrow a[j]$;    // a[j]를 부모 노드로 이동
  }
  $a[j/2] \leftarrow a[h]$;
end heapify()
```

- 완전 2진 트리를 히프로 변환

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	in-place slow (good for small inputs)
insertion-sort	$O(n^2)$	in-place slow (good for small inputs)
bubble-sort	$O(n^2)$	in-place slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	in-place, randomized fastest (good for large inputs)
heap-sort	$O(n \log n)$	in-place fast (good for large inputs)
merge-sort	$O(n \log n)$	sequential data access Not in-place(추가공간필요) fast (good for huge inputs)

Some Remarks

- Insertion-sort is a good choice for small input size (say, less than 50) and for sequences that are already "almost" sorted. (삽입정렬은 50개 이하정렬에좋음)
- Merge-sort is difficult to run in-place, is an excellent algorithm for situations where the input can not fit into main memory, but must be stored in blocks on an external memory device, e.g., disks.(합병정렬은 메모리크기보다 큰 자료정렬에 좋음)
- Quick sort is an excellent choice for general-purpose, in-memory sorting. In spite of its slow worst-case running time. The constant factors hidden in O(nlgn) for average case are quite small.

Quicksort for Small Arrays

- For very small arrays (N<= 20), quicksort does not perform as well as insertion sort(20개이하에서 퀵보다 삽입정렬이 좋음)
- A good cutoff range is N=10
- Switching to insertion sort for small arrays can save about 15% in the running time

Comparisons of Mergesort and Quicksort

- Both run in O(nlogn)
- Compared with Quicksort, Mergesort has less number of comparisons but larger number of moving elements
- In Java, an element comparison is expensive but moving elements is cheap. Therefore, Mergesort is used in the standard Java library for generic sorting

Comparison-based Sorting(비교기반정렬)

- **Comparison sort(비교정렬)**
  - Only comparison of pairs of elements may be used to gain order information about a sequence.
  - Hence, a lower bound on the number of comparisons will be a lower bound on the complexity of any comparison-based sorting algorithm.
- All our sorts have been comparison sorts
- The best worst-case complexity so far is $\Theta(n \lg n)$ (merge sort and heapsort).
- 비교기반정렬은 아무리 빨라도 $\Theta(n \lg n)$

감사합니다.