Data Structure

Spring 2019

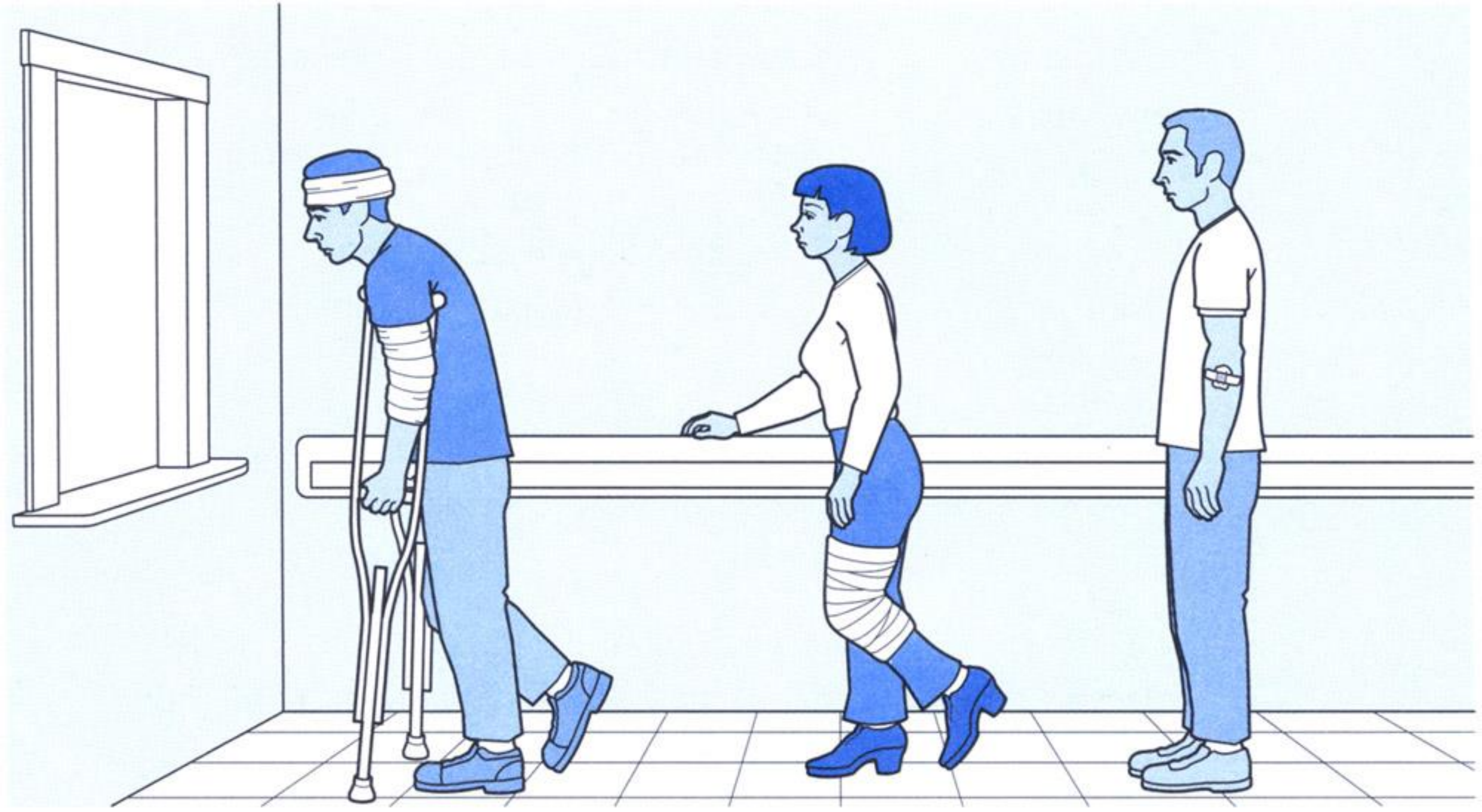
MW 11:00-12:15, SN 014

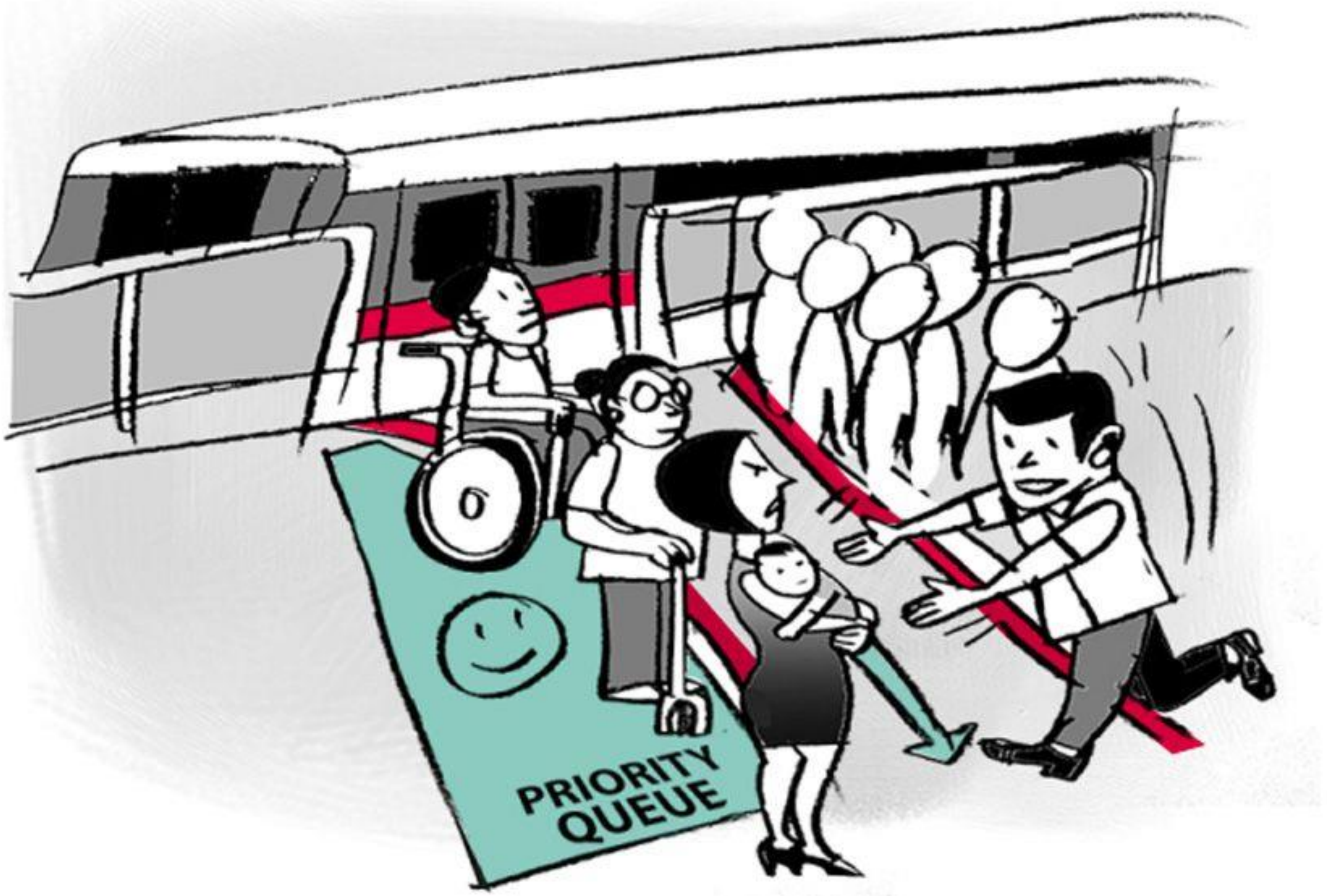
http://smart.hallym.ac.kr

Instructor: Jin Kim
010-6267-8189(033-248-2318)
jinkim@hallym.ac.kr

Office Hours: M 12:15-1, Tu 3:30-5pm M 2-3pm, W 2-3:30

Priority Queue(우선순위 큐)





'Can I borrow your baby?...'

Priority Queues

- Items in a priority queue have a priority
  - ✓The priority is usually numerical value
  - ✓Could be lowest first or highest first
- The highest priority item is removed first가장 높은 우선순위 아이템이 가장 먼저 제거됨
- Priority queue operations 작업
  - ✓Insert 삽입
  - ✓Remove삭제 in priority queue order (not a FIFO!)

Implementing a Priority Queue

- Items have to be removed in priority order
- This can only be done efficiently if the items are ordered in some way
  - ✓ A balanced binary search (e.g., red-black) tree is an efficient and ordered data structure but
    - Some operations (e.g. removal) are complex to code
    - Although operations are O(log *n*) they require quite a lot of structural overhead
- There is another binary tree solution – *heaps*
  - ✓ **Note:** We will see that search/removal of the maximum element is efficient, but it's not true for other elements

Priority Queue – Linked List 구현

- Ordered Linked List(정렬되어 있음)
  - ✓findMin and deleteMin in constant time.
  - ✓insert in O(n).
- Unordered List(정렬되지 않음)
  - ✓insert in constant time.
  - ✓findMin and deleteMin in O(n).

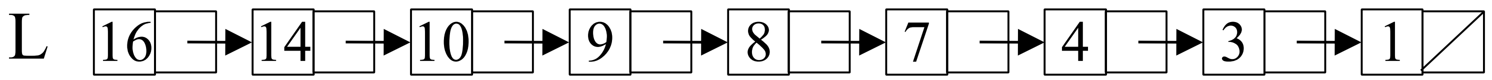
Priority Queue Trees

- Binary Search Tree
  - ✓Find can be more than O(lg n) if out of balance.
  - ✓Insert and delete can be more than O(lg n).
- AVL Tree
  - ✓Find is O(lg n) time.
  - ✓Insert and delete are O(lg n).

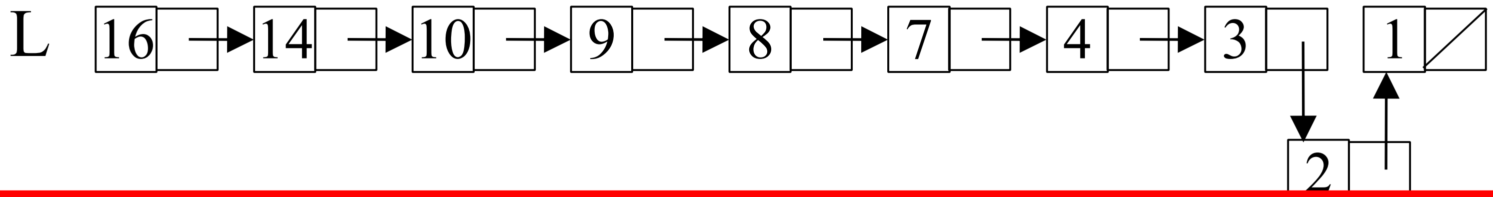
Priority Queue: implementations

- Unordered linked list
  - ✓insert is O(1), deleteMin is O(n)
- Ordered linked list
  - ✓deleteMin is O(1), insert is O(n)
- Balanced binary search tree
  - ✓insert, deleteMin are O(log n)
  - ✓increaseKey, decreaseKey, remove are O(log n)
  - ✓union is O(n)
- Most implementations are based on heaps . . .

Linked List Implementation of Priority Queues



Suppose an item with priority 2 is to be added:

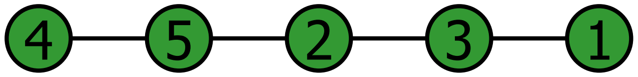


Only *O(1)* (constant) pointer changes required, but it takes *O(N)* pointer traversals to find the location for insertion.
연결리스트구현: 원소가 정렬되어 있음을 전제
Dequeue시 상수시간소요O(1), enqueue시 원소의 개수 n에 비례O(n)

Wanted: a data structure for PQs that can be both **searched** and **updated** in better than *O(N)* time.

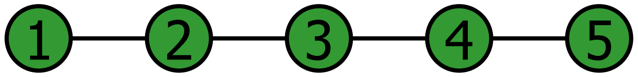
Priority Queue Implementation Sequence-based Priority Queue

- Implementation with an unsorted list(비정렬)



- Performance:
  - insert takes ***O***(1) time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take ***O***(***n***) time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list(정렬)



- Performance:
  - insert takes ***O***(***n***) time since we have to find the place where to insert the item
  - removeMin and min take ***O***(1) time, since the smallest key is at the beginning

Implementations

Linked list:

- **Insert at the beginning - O(1)**
- **Find the minimum - O(N)**

Binary search tree (BST): 이진탐색트리

- **Insert O(logN)**
- **Find minimum - O(logN)**

Implementations

Binary heap

Better than BST because it does not support links

- **Insert O(logN)**
- **Find minimum O(logN)**

Deleting the minimal element takes a constant time, however after that the heap structure has to be adjusted, and this requires **O(logN)** time.

Priority Queue Implementation

- 배열과 연결리스트로 구현해보자
- 아주 다양한 구현 방법이 있다
- 트리를 이용하는것은 2학기 비선형자료구조를 공부할 때 해보자.
- 우리는 원소를 정렬할 것이다.
- 우선순위큐에서 큐에 큐가 텅빌때까지 원소삭제하여 늘어놓으면 정렬이 됨.

Priority Queue Implementation

- 배열을 사용하여 구현(원소가 정렬되어 있을때를 전제)
  - ✓삽입 : 원소의 개수에 비례
  - ✓삭제 : 원소의 개수와 무관. 상수시간

Priority Queues: Array(배열을 사용)

remove() : 마지막 방의 원소(키)를 제거한다. 마지막 키는 가장 작은 키값.

So, the first item has priority and can be retrieved (removed) **quickly** and returned to calling environment.

Hence, 'remove()' is easy (and will take O(1) time)

insert() : 키값에 맞는 위치에 저장한다.

But, we want to **insert** quickly. Must go into proper position.

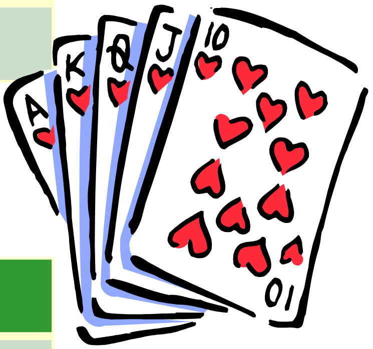
For our purposes here: Implementing data structure: **array**;

- **slow to insert**(), **but** for
  - ❿small number of items in the pqueue, **and**
  - ❿where insertion speed is not critical,
- this is the simplest and best approach.

실제 구현 (배열 사용)

초기상태(작은 수가 가장 큰 우선순위를 가진다 가정)

0	1	2	3	4	5	6	7	8	9
68	33	25	21	13	8				



삽입 (29) : O(n)
원소개수에 비례

29

29

0	1	2	3	4	5	6	7	8	9
68	33	29	25	21	13	8			

삭제 O(1) : 상수시간이 소요

0	1	2	3	4	5	6	7	8	9
68	33	29	25	21	13				

8

큐 full이면 크기 늘려야 한다.

```
//priorityQ.java
class PriorityQ   {    // array in sorted order, from max at 0 to min at nitems-1
  private int maxSize;
  private long[] queArray;
  private int nItems;
 public PriorityQ(int s)   {     // constructor            //  Samo samo!
     maxSize = s;
     queArray = new long[maxSize];
     nItems = 0;
     }
 public void insert(long item)  {     // insert item             //  ============================
     int j;                                                                   // insert:  30, 50, 10, 40, 20
     if (nItems==0)                         // if no items,                // look carefully at the 'disposition' of the index….
        queArray[nItems++] = item;  // insert at 0                //  Note: algorithm inserts and then increments
     else                                  // if queue has items,       // So index always points to the last item inserted!
     {                                                                      //   So, if pqueue is empty, this is simple, but:
       for (j=nItems-1; j>=0; j--)        // start at end,              //      if pqueue is NOT empty, must find correct spot!!!!

          if (item > queArray[j] )                             // if new item larger,
             queArray[j+1] = queArray[j];                      // shift upward;  Keep shifting until 'if' is false.
          else                                                          // if smaller,  done shifting.
             break;                                                     // We've moved everything forward to open a slot for 'item.'
          // end for
       queArray[j+1] = item;
       nitems++;                                            // insert new item whether we've had to shift or not.
                                                                      // increment number of items in the pqueue.
     }  // end else (nItems > 0)              // Note value of j when we leave the for loop!  queArray[j+ 1 is correct.
=============================================
     }  // end insert()
 public long remove()                                   // remove minimum item;  remove from front of queue.  Easy
     { return queArray[--nItems]; }                    //      can we infer we always point to next open slot?
 public long peekMin()          // peek at minimum item
     { return queArray[nItems-1]; }
 public boolean isEmpty()       // true if queue is empty                              Very straightforward.
     { return (nItems==0); }
 public boolean isFull()        // true if queue is full
     { return (nItems == maxSize); }
 }  // end class PriorityQ`
```

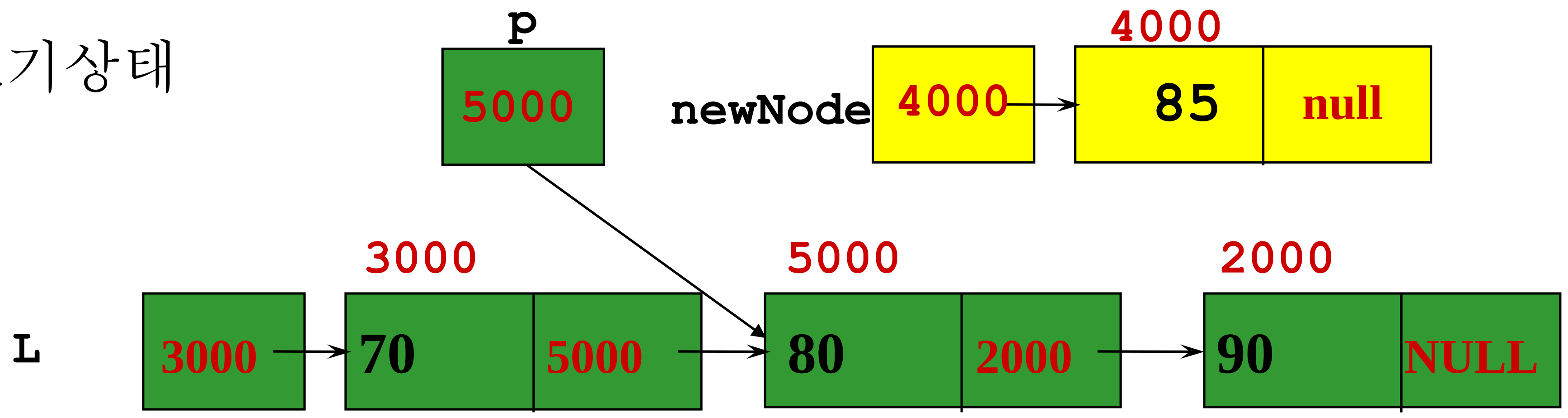
```
class PriorityQApp
  {
  public static void main(String[] args)
    {
    PriorityQ thePQ = new PriorityQ(5);
          // what do you know about the priority queue at this point?
    thePQ.insert(30);
    thePQ.insert(50);
    thePQ.insert(10);
    thePQ.insert(40);
    thePQ.insert(20);

    while( !thePQ.isEmpty() )
      {
      long item = thePQ.remove();
      System.out.print(item + " ");  // 10, 20, 30, 40, 50 Note: ORDERED!
      }  // end while
    System.out.println("");
    }  // end main()
//-------------------------------------------------------------
  }  // end class PriorityQApp
```

Priority Queue Implementation

- 연결 리스트를 사용하여 구현
  - ✓삽입 : 원소의 개수에 비례
  - ✓삭제 : 원소의 개수와 무관. 상수시간. Header가 첫번째 원소에서 두 번째 원소를 포인트하면 됨

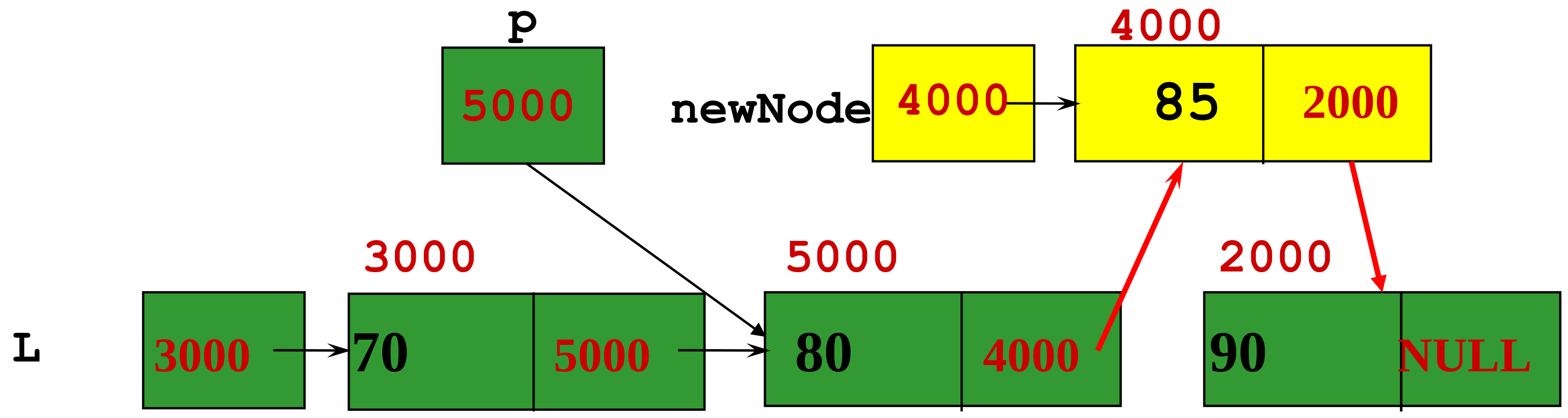
실제 구현 (연결리스트 사용)

초기상태



L=(70, 80, 90)

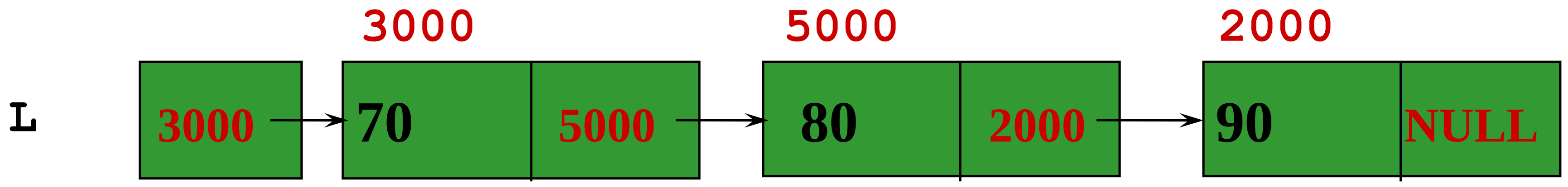
삽입 (75) O(n)



L=(70, 80, 85, 90)

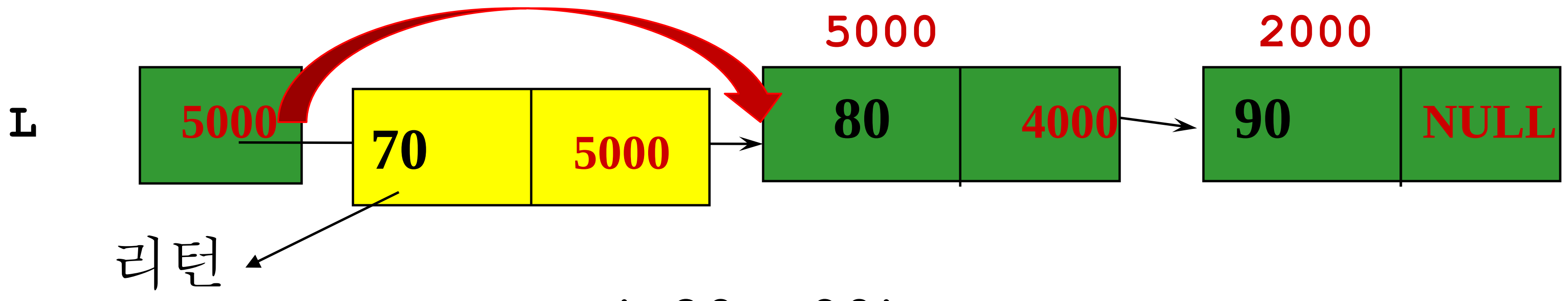
실제 구현 (연결리스트 사용)

초기상태



L=(70, 80, 90)

삭제 O(1)



L=(80, 90)

Priority queue(연결리스트)

```
public class QlistNode {
        String name;
        int priority;
        QlistNode link;
}
```

```
public class PQmain {
        public static void main(String[] args){
                PQ aQ= new PQ();
                aQ.enqueue("Kim",30);
                aQ.enqueue("Lee",20);
                aQ.enqueue("Pak",10);
                aQ.enqueue("Cho",70);
                //aQ.numberofitem();
                //aQ.peek();
                System.out.println(aQ.dequeue());
                System.out.println(aQ.dequeue());
                System.out.println(aQ.dequeue());
                System.out.println(aQ.dequeue());
                System.out.println(aQ.dequeue());
        }
}
```

```
public class PQ {
        QlistNode front;
        public PQ(){
        }
        void enqueue(String name, int priority){ //키를 맞는 위치에 삽입
                QlistNode p;
                QlistNode newNode=new QlistNode();
                newNode.name=name;
                newNode.priority=priority;
                newNode.link=null;
                if((front==null)||(newNode.priority<front.priority)){
                        newNode.link=front; front=newNode;
                }
                else{
                        p=front;
                        while(p.link!=null&&p.link.priority<=newNode.priority){
                                p=p.link;
                        }
                        newNode.link=p.link;
                        p.link=newNode;
                }
        }
        String dequeue(){ //선두원소 제거
            String name="";
                if(front==null){ System.out.println("Queue underflow");}
                else {name=front.name; front=front.link;}
                return name;
        }
        int numberofitem(){
                int x=0;
                return x;
        }
        String peek(){
                String name="";
                return name;
        }
}
```

Efficiency of Priority Queues

remove() runs in O(1) time – super

Recall, front is at location nitems-1

insert() runs in O(n) time (makes sense!)

If the priority queue is implemented with a sorted sequence, the sort is called **insertion sort**. The table here shows the performance of insertion sort on the same sequence
S = (7, 4, 8, 2, 5, 3, 9).

	Sequence *S*	Priority Queue *P*
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1	(4, 8, 2, 5, 3, 9)	(7)
	(8, 2, 5, 3, 9)	(4, 7)
	(2, 5, 3, 9)	(4, 7, 8)
	(5, 3, 9)	(2, 4, 7, 8)
	(3, 9)	(2, 4, 5, 7, 8)
	(9)	(2, 3, 4, 5, 7, 8)
	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2	(2)	(7, 4, 8, 5, 3, 9)
	(2, 3)	(7, 4, 8, 5, 9)
	…	…
	(2, 3, 4, 5, 7, 8, 9)	()

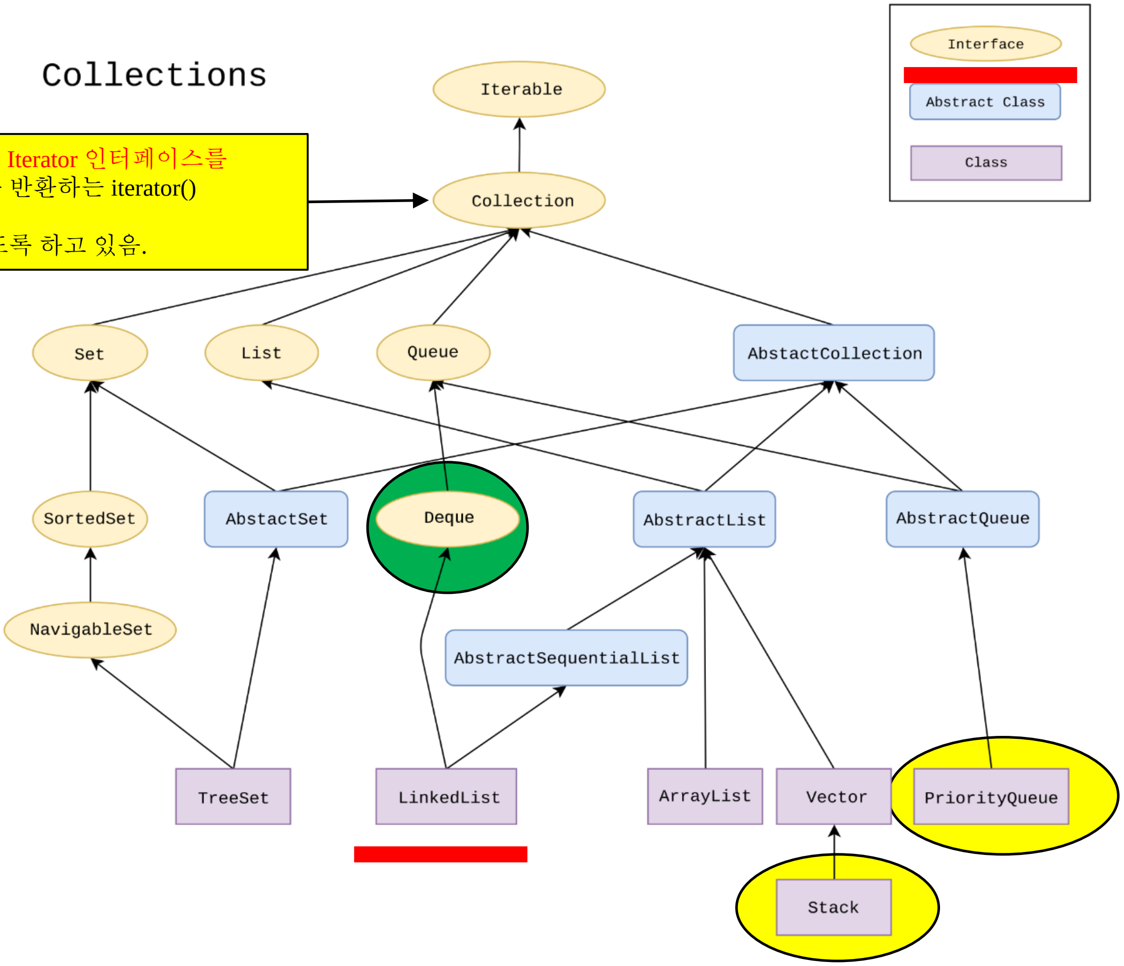
Priority queue

- A stack is first in, last out
- A queue is first in, first out
- A priority queue is *least-first-out(작은 것 우선)*
  - ✓ The “smallest” element is the first one removed
    - (You could also define a *largest-first-out* priority queue)
  - ✓ The definition of “smallest” is up to the programmer (for example, you might define it by implementing Comparator or Comparable)
  - ✓ If there are several “smallest” elements, the implementer must decide which to remove first
    - Remove any “smallest” element (don’t care which)
    - Remove the first one added

Java Priority Queue

- https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html
- java.util
  - ✓ **Class PriorityQueue<E> // PriorityQueue는 class. 따라서 직접 빵을 찍을 수 있다.**

Java collections frameworks



Collection 인터페이스에서는 Iterator 인터페이스를
구현한 클래스의 인스턴스를 반환하는 iterator()
메소드를
정의하여 각 요소에 접근하도록 하고 있음.

Priority Queue in Java

Constructors of Priority Queue:

PriorityQueue(): Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
PriorityQueue(Collection<E> c): Creates a PriorityQueue containing the elements in the specified collection.
PriorityQueue(int initialCapacity): Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
PriorityQueue(int initialCapacity, Comparator<E> comparator): Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
PriorityQueue(PriorityQueue<E> c): Creates a PriorityQueue containing the elements in the specified priority queue.
PriorityQueue(SortedSet<E> c): Creates a PriorityQueue containing the elements in the specified sorted set.

Methods in Priority Queue Class

1. boolean add(E element): This method inserts the specified element into this priority queue.
2. public remove(): This method removes a single instance of the specified element from this queue, if it is present
3. public poll(): This method retrieves and removes the head of this queue, or returns null if this queue is empty.
4. public peek(): This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
5. Iterator iterator(): Returns an iterator over the elements in this queue.
6. boolean contains(Object o): This method returns true if this queue contains the specified element
7. void clear(): This method is used to remove all of the contents of the priority queue.
8. boolean offer(E e): This method is used to insert a specific element into the priority queue.
9. int size(): The method is used to return the number of elements present in the set.
10. toArray(): This method is used to return an array containing all of the elements in this queue.
11. Comparator comparator(): The method is used to return the comparator that can be used to order the elements of the queue.

java.util.PriorityQueue 사용하기

Generic : 클래스 내부에서 사용할 데이터 타입을
나중에 인스턴스를 생성할 때 확정하는 것을 제네릭이라 한다

객체 사용

```
import java.util.PriorityQueue;

Public class MyPQ {
                …
        PriorityQueue<String> pQueue =
                                  new PriorityQueue<String>();
        pQueue.add("C");
        pQueue.add("C++");
        pQueue.add("Java");
        pQueue.add("Python");
              …
        pQueue.poll();
```

PQ에서 큰 수 먼저 출력하는 방법(PQascending.java)

PQ는 정수를 우선 순위 큐에 넣고 출력할 경우, 작은 수부터 출력한다.
이를 반대로 출력하도록 하는 몇가지
방법이 있다. Pqascending.java를 보고 그 방법을 이해하라.

Comparable 사용하여 비교

- 정렬가능한 클래스들의 기본 정렬 기준과 다르게 정렬하고 싶을때 사용하는 인터페이스
- Comparable interface를 implements한 후 compareTo() 메소드를 오버라이드함
  - ✓첫번째 파라메터 < 두번째 파라메터 : 음수리턴
  - ✓첫번째 파라메터 > 두번째 파라메터 : 양수리턴
  - ✓첫번째 파라메터 = 두번째 파라메터 : 0리 턴

```
import java.lang.Comparable;
import java.util.PriorityQueue;

class PQComparable {
	public static void main(String[] args) {
		PriorityQueue<MyClass> queue = new PriorityQueue<MyClass>();
		queue.add(new MyClass(2, "short"));
		queue.add(new MyClass(2, "very long indeed"));
		queue.add(new MyClass(1, "medium"));
		queue.add(new MyClass(1, "very long indeed"));
		queue.add(new MyClass(2, "medium"));
		queue.add(new MyClass(1, "short"));
		while (queue.size() != 0)
			System.out.println(queue.remove());
	}
}
class MyClass implements Comparable<MyClass> {
	int sortFirst;
	String sortByLength;

	public MyClass(int sortFirst, String sortByLength) {
		this.sortFirst = sortFirst;
		this.sortByLength = sortByLength;
	}

	@Override
	public int compareTo(MyClass other) {
		if (sortFirst != other.sortFirst)
			return Integer.compare(sortFirst, other.sortFirst);
		else
			return Integer.compare(sortByLength.length(), other.sortByLength.length());
	}

	public String toString() {
		return sortFirst + ", " + sortByLength;
	}
}
```

PQ에서 객체를 비교하기(PQCompObj.java)

PQ에 객체를 입력하면, 객체들을 비교하기 위한 방법이 있어야 한다.
우선순위를 제공하는 방법을
. PQCompObj.java를 보고 이해하라.

Comparator 사용하여 비교

- 정렬가능한 클래스들의 기본 정렬 기준과 다르게 정렬하고 싶을때 사용하는 인터페이스
- Comparator interfac를 implements한 후 compare() 메소드를 오버라이드한 myComparator class를 작성
  - ✓첫번째 파라메터 < 두번째 파라메터 : 음수리턴
  - ✓첫번째 파라메터 > 두번째 파라메터 : 양수리턴
  - ✓첫번째 파라메터 = 두번째 파라메터 : 0리 턴

```
import java.util.PriorityQueue;

class MyClass {
    int id;
   String name;

   public MyClass(int id, String name) {
        this.id = id;
        this.name = name;
   }
}

public class PQCompObj {
  public static void main(String[] args) {
    Comparator<MyClass> comparator = new MyClassComparator();
    PriorityQueue<MyClass> queue = new PriorityQueue<MyClass>(10, comparator);  //comparator를 사용하여 정렬
    MyClass n1 = new MyClass(300, "Kim");
    MyClass n2 = new MyClass(100, "Lee");
    MyClass n3 = new MyClass(200, "Park");
    MyClass n4 = new MyClass(400, "Choi");
    queue.add(n1);
    queue.add(n2);
    queue.add(n3);
    queue.add(n4);

    while (queue.size() != 0) {
      System.out.println(queue.remove());
   }
                  }
}
```

```
class MyClassComparator implements Comparator<MyClass> {
    @Override
    public int compare(MyClass x, MyClass y) {
      // Assume neither string is null. Real code should
      // probably be more robust
      // You could also just return x.length() - y.length(),
      // which would be more efficient.
      if (x.id < y.id) { //id를 기준으로 정렬
         return -1;
      }
      if (x.id > y.id) {
          return 1;
      }
      return 0;
   }
}
```

참고 Iterator (PriorityQueue1.java)

Iterator는 자바의 컬렉션 프레임워크에서 컬렉션에 저장되어 있는 요소들을 읽어오는 방법을 표준화 하였는데 그 중 하나가 Iterator. 인터페이스이다. 모든 컬렉션클래스에서 데이터를
읽을 때 사용됨.

```
public interface Iterator {
        boolean hasNext();   // 원소가 남아있는냐?
        Object next();   //다음 원소를 가져와라.
        void remove(); // next 메소드가 호출한 원소를 제거하라.
}
```

사용법

Iterator 타입의 변수를 Iterator로 변환한다.

변수

```
Iterator it1 = pq.iterator();
While(it1.hasNext()){
     ….
}
```

Heap

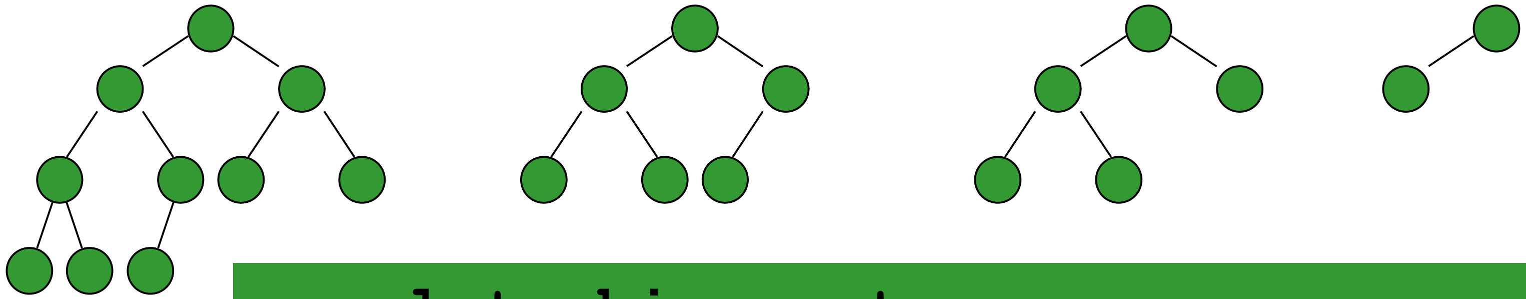
비선형 자료구조

이므로 다음 학기에 공부하도록 하자

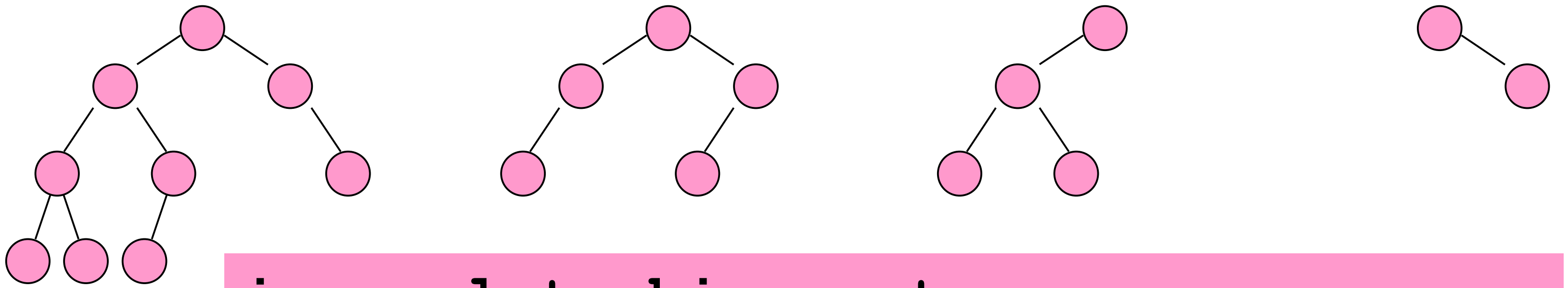
Heaps

- A heap is binary tree with two properties
- Heaps are complete
  - All levels, except the bottom, must be completely filled in
  - The leaves on the bottom level are as far to the left as possible.
- Heaps are partially ordered ("heap property"):
  - **The value of a node is at least as large as its children's values, for a *max heap*** or
  - The value of a node is no greater than its children's values, for a *min heap*

Complete Binary Trees

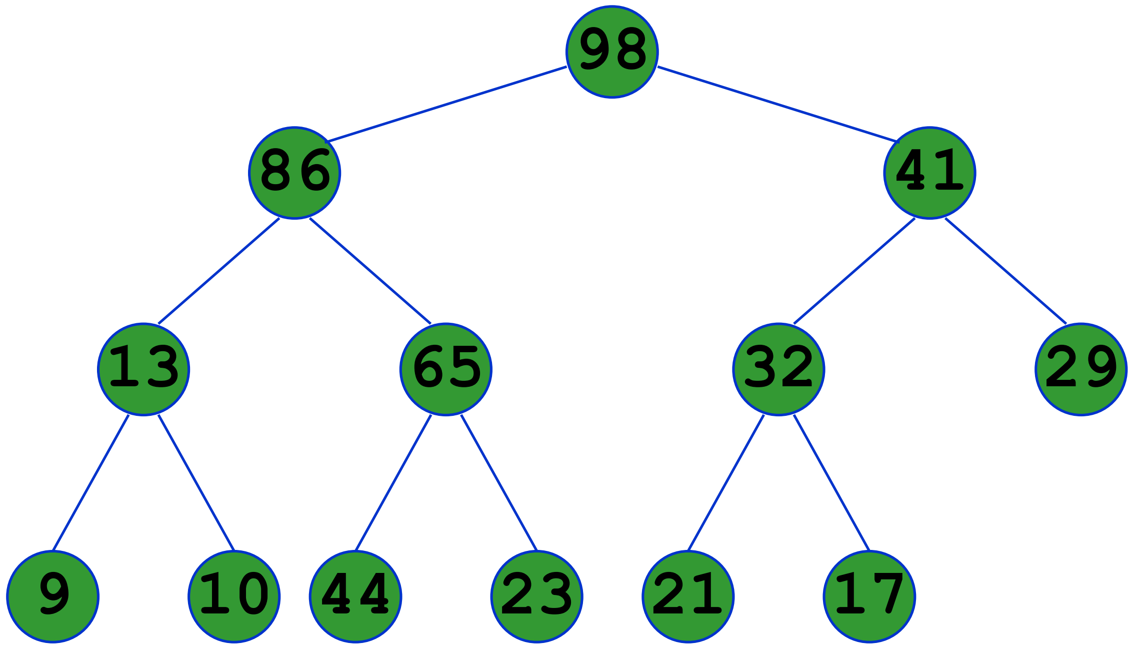


complete binary trees



incomplete binary trees

Partially Ordered Tree – max heap



Note: an inorder traversal would result in:

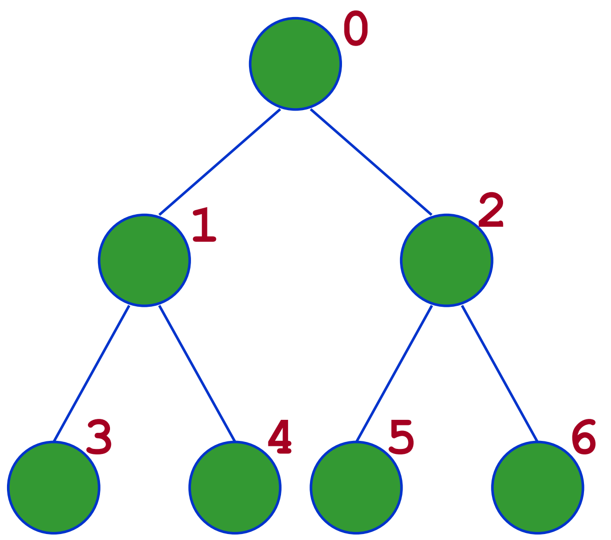
9, 13, 10, 86, 44, 65, 23, 98, 21, 32, 17, 41, 29

Priority Queues and Heaps

- A heap can be used to implement a priority queue
- Because of the partial ordering property the item at the top of the heap must always the largest value
- Implement priority queue operations:
  - ✓Insertions – insert an item into a heap
  - ✓Removal – remove and return the heap’s root
  - ✓For both operations preserve the heap property

Heap Implementation

- Heaps can be implemented using arrays
- There is a natural method of indexing tree nodes
  - Index nodes from top to bottom and left to right as shown on the right (by levels)
  - Because heaps are complete binary trees there can be no gaps in the array



Array implementations of heap

```
public class Heap<T extends KeyedItem> {
  private int HEAPSIZE=200;
  // max. number of elements in the heap
  private T items[];          // array of heap items
  private int num_items;      // number of items

  public Heap() {
    items = new T[HEAPSIZE];
    num_items=0;
  }  // end default constructor
```

- We could also use a **dynamic array** implementation to get rid of the limit on the size of heap.
- We will assume that priority of an element is equal to its key. So the elements are partially sorted by their keys. They element with the biggest key has the highest priority.

Referencing Nodes

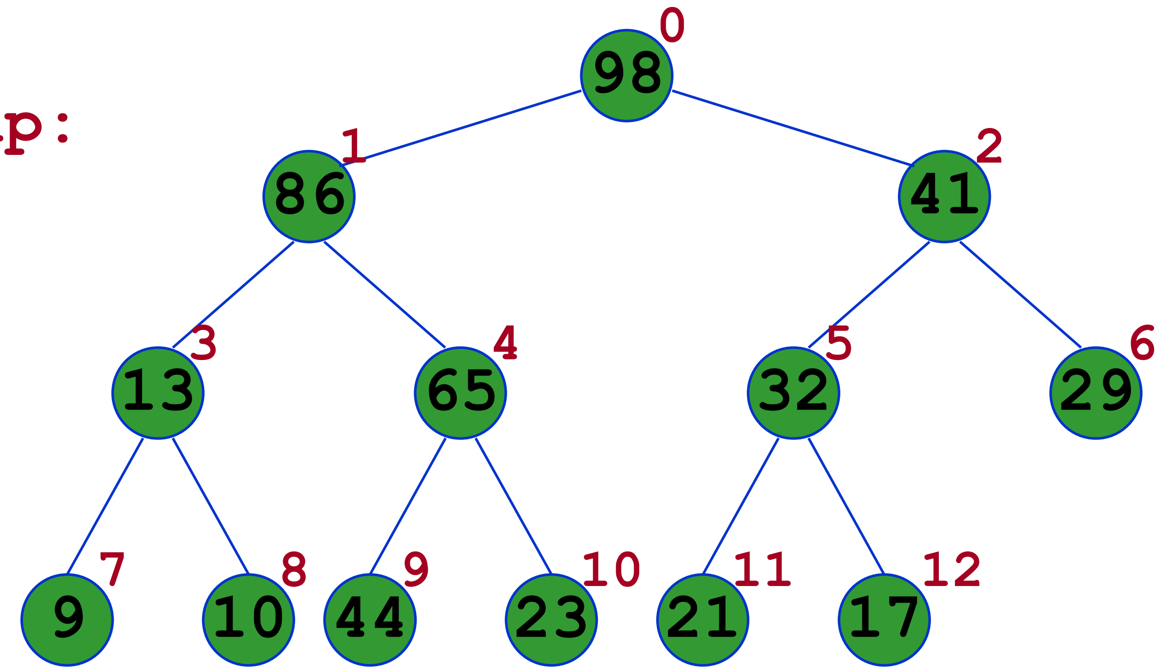
- It will be necessary to find the indices of the parents and children of nodes in a heap's underlying array
- The children of a node `i`, are the array elements indexed at `2i+1` and `2i+2`
- The parent of a node *i*, is the array element indexed at `floor[(i-1)/2]`

Helping methods

```
private int parent(int i)
{ return (i-1)/2; }

private int left(int i)
{ return 2*i+1; }

private int right(int i)
{ return 2*i+2; }
```

Heap Array Example

Heap:



Underlying Array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

Heap Insertion

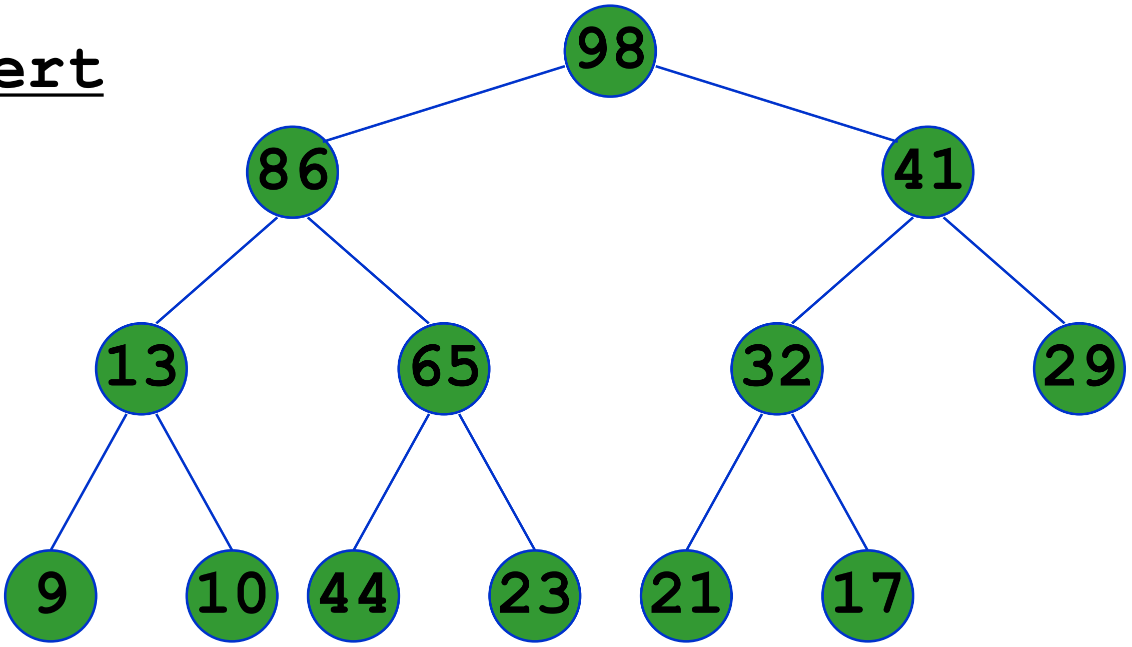
- On insertion the heap properties have to be maintained; remember that
  - ✓ A heap is a complete binary tree and
  - ✓ A partially ordered binary tree
- There are two general strategies that could be used to maintain the heap properties
  - ✓ Make sure that the tree is complete and then fix the ordering, or
  - ✓ Make sure the ordering is correct first
  - ✓ Which is better?

Heap Insertion algorithm

- The insertion algorithm first ensures that the tree is complete
  - ✓ Make the new item the first available (left-most) leaf on the bottom level
  - ✓ i.e. the first free element in the underlying array
- Fix the partial ordering
  - ✓ Repeatedly compare the new value with its parent, swapping them if the new value is greater than the parent (for a max heap)
  - ✓ Often called “bubbling up”, or “trickling up”

Heap Insertion Example

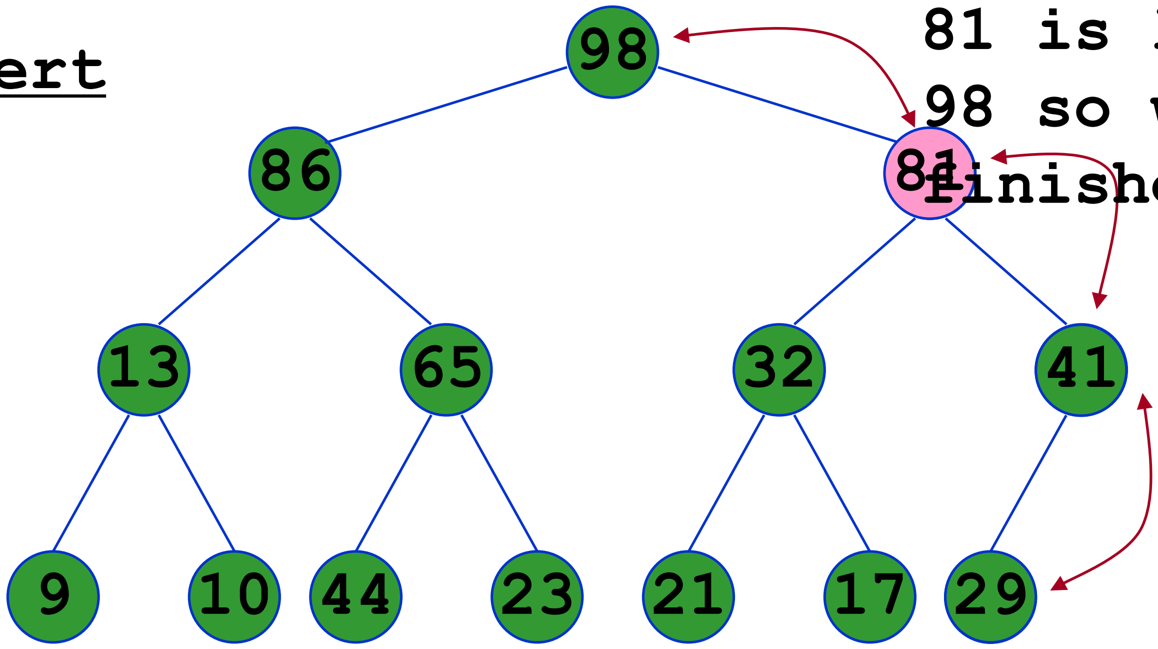
Insert 81



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	98	86	41	13	65	32	29	9	10	44	23	21	17	

Heap Insertion Example

Insert 81



81 is less than 98 so we are finished

(13-1)/2 = 6

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	98	86	81	13	65	32	41	9	10	44	23	21	17	29

Heap Insertion algorithm

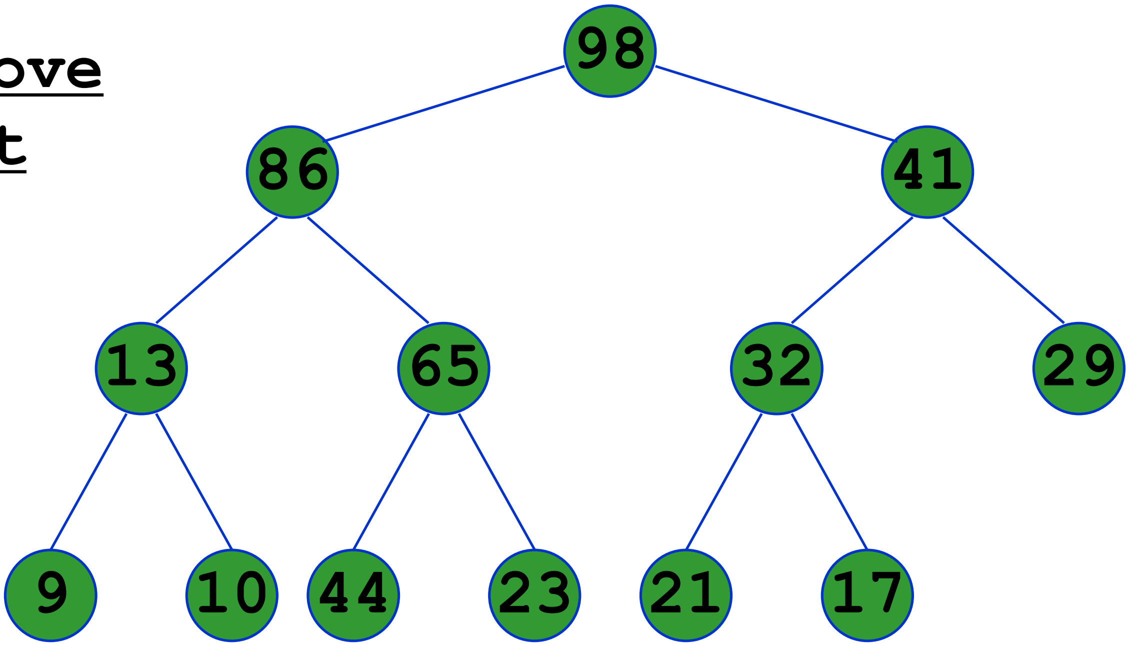
```
public void insert(T newItem) {
  // TODO: should check for the space first
  num_items++;
  int child = num_items-1;
  while (child > 0 &&
         item[parent(child)].getKey() <
         newItem.getKey()) {
    items[child] = items[parent(child)];
    child = parent(child);
  }
  items[child] = newItem;
}
```

Heap Removal algorithm

- Make a temporary copy of the root’s data
- Similar to the insertion algorithm, ensure that the heap remains complete
  - ✓ Replace the root node with the right-most leaf on the last level
  - ✓ i.e. the highest (occupied) index in the array
- Repeatedly swap the new root with its largest valued child until the partially ordered property holds
- Return the root’s data

Heap Removal Example

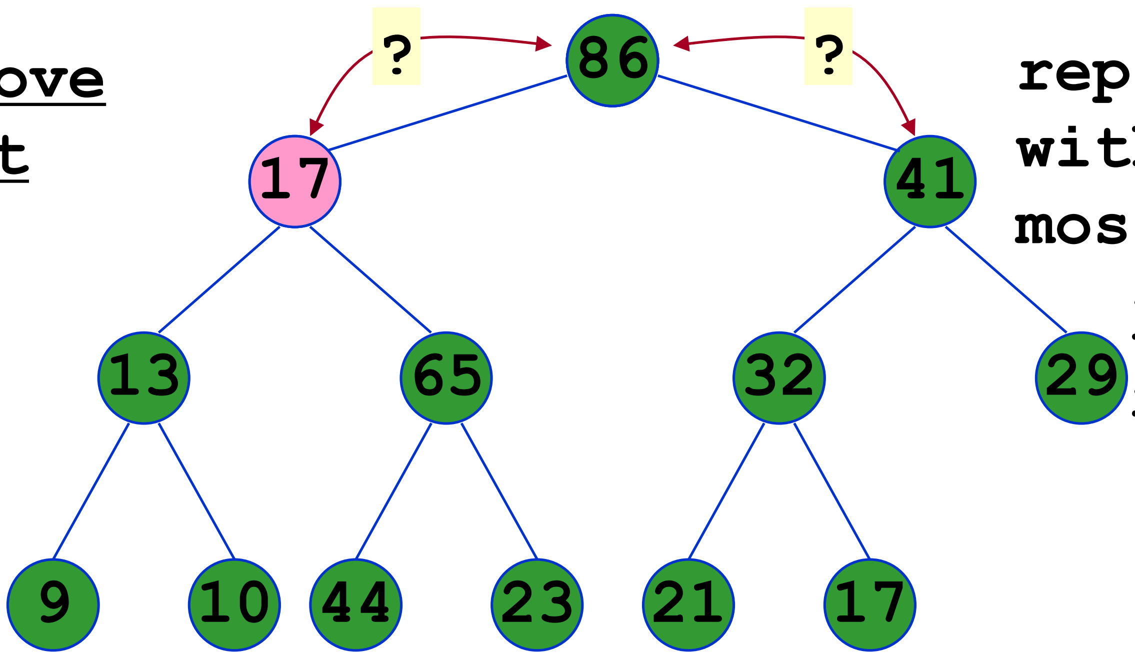
Remove root



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	98	86	41	13	65	32	29	9	10	44	23	21	17

Heap Removal Example

Remove root



replace root with right-most leaf

left child is greater

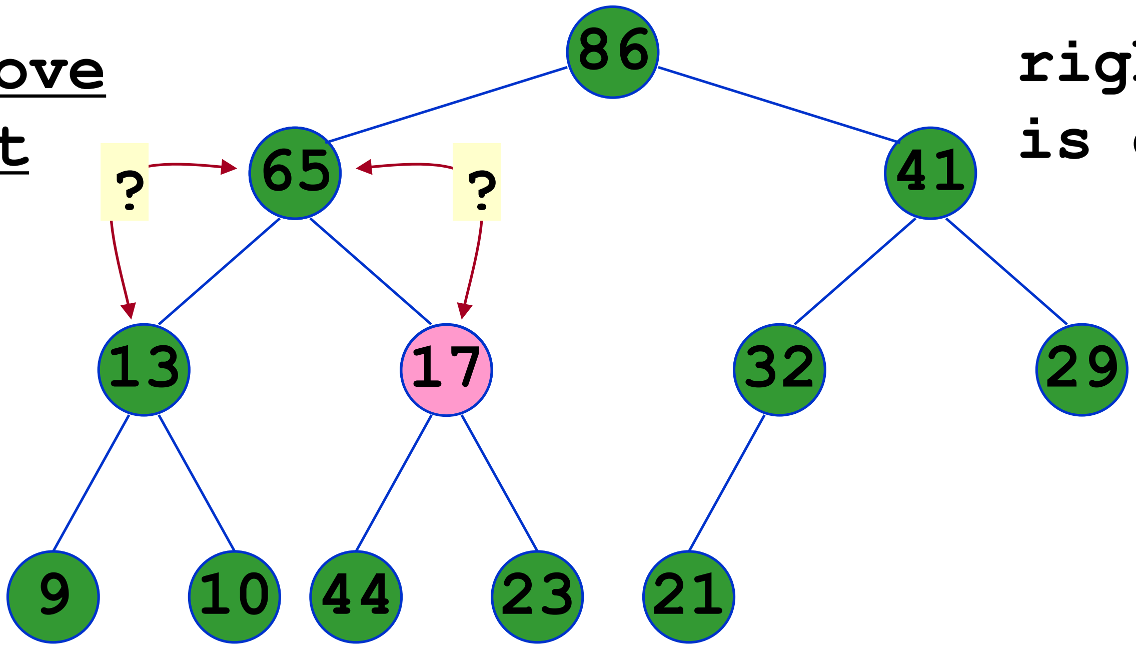
children of root: 2*0+1, 2*0+2 = 1, 2

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	8	1	41	13	65	32	29	9	10	44	23	21	

76 7

Heap Removal Example

Remove root



right child is greater

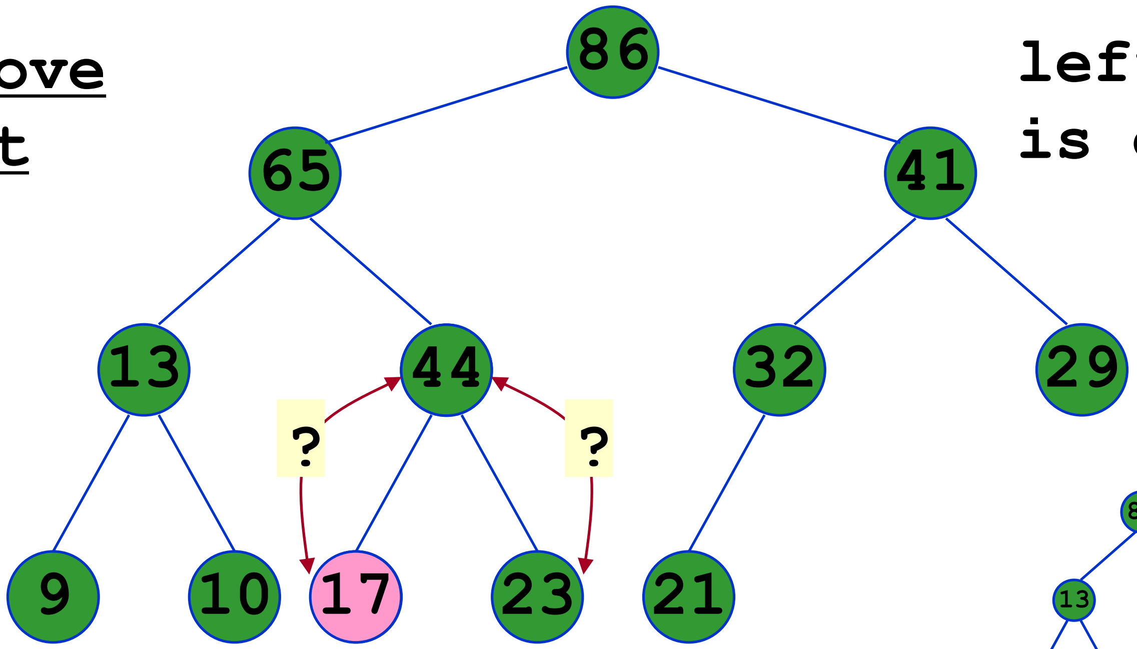
children: 2*1+1, 2*1+2 = 3, 4

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	6	41	13	1	32	29	9	10	44	23	21	

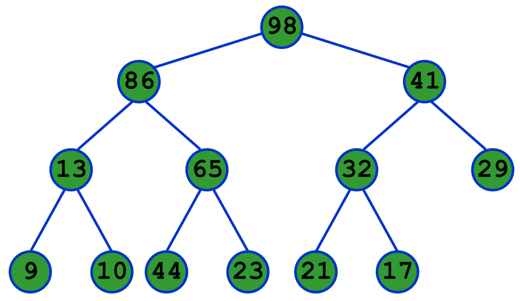
5 7 7

Heap Removal Example

Remove root



left child is greater



children: 2*4+1, 2*4+2 = 9, 10

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	86	65	41	13	4	32	29	9	10	1	23	21	

4 7

Heap Removal algorithm

```java
public T remove()
  // remove the highest priority item
  {
    // TODO: should check for empty heap
    T result = items[0]; // remember the item
    T item = items[num_items-1];
    num_items--;
    int current = 0; // start at root
    while (left(current) < num_items) { // not a leaf
      // find a bigger child
      int child = left(current);
      if (right(current) < num_items &&
          items[child].getKey() < items[right(current)].getKey()) {
        child = right(current);
      }
      if (item.getKey() < items[child].getKey()) {
        items[current] = items[child];
        current = child;
      } else
        break;
    }
    items[current] = item;
    return result;
  }
```

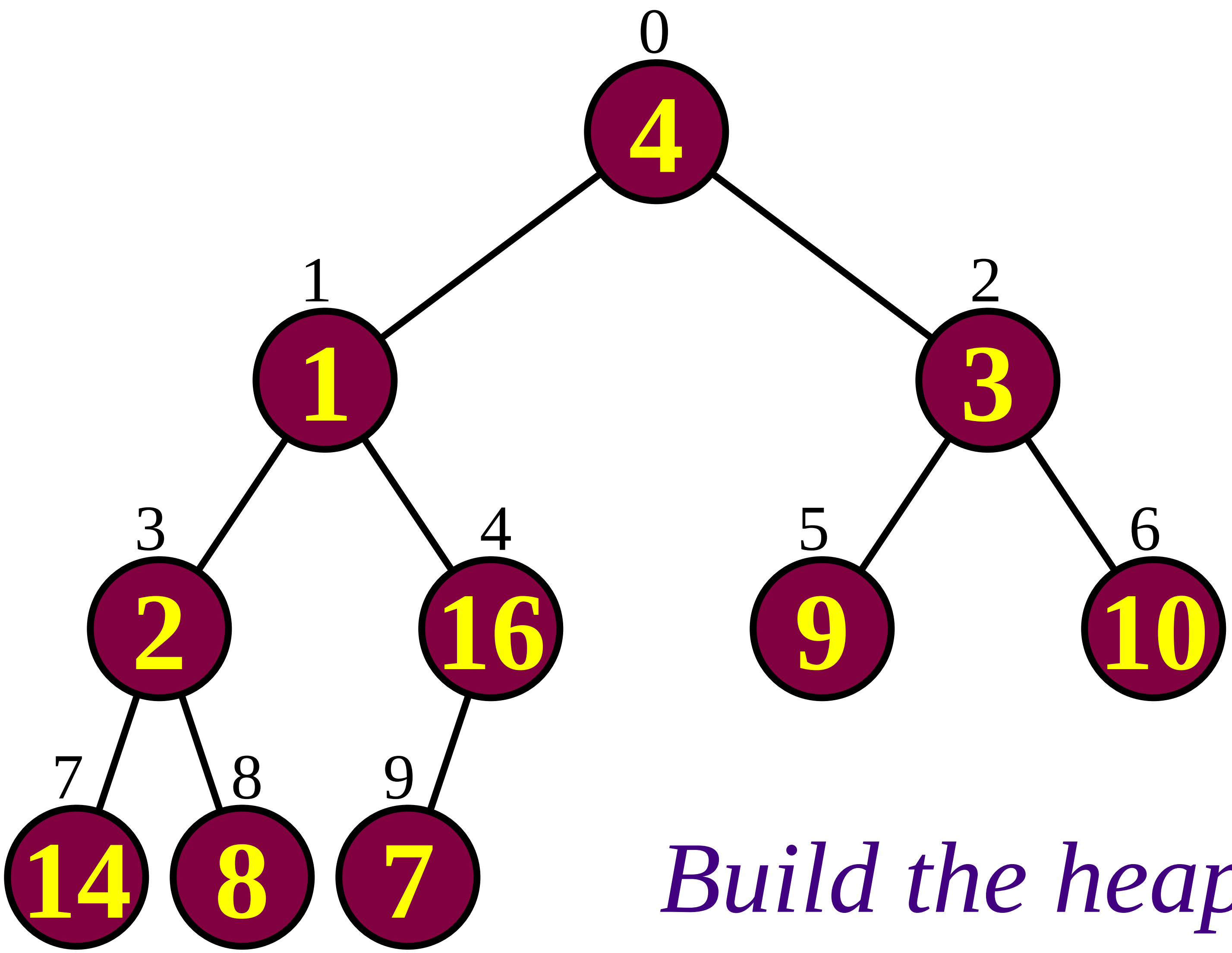
Heap Efficiency

- For both insertion and removal the heap performs at most *height* swaps
  - ✓ For insertion at most *height* comparisons
  - ✓ For removal at most *height**2 comparisons
- The height of a complete binary tree is given by $\lfloor \log_2(n) \rfloor + 1$
  - ✓ Both insertion and removal are O(log*n*)
  - ✓ 즉 삽입과 삭제시간이 O(log*n*)

Remark: but removal is only implemented for the element with the highest key!

Heapsort

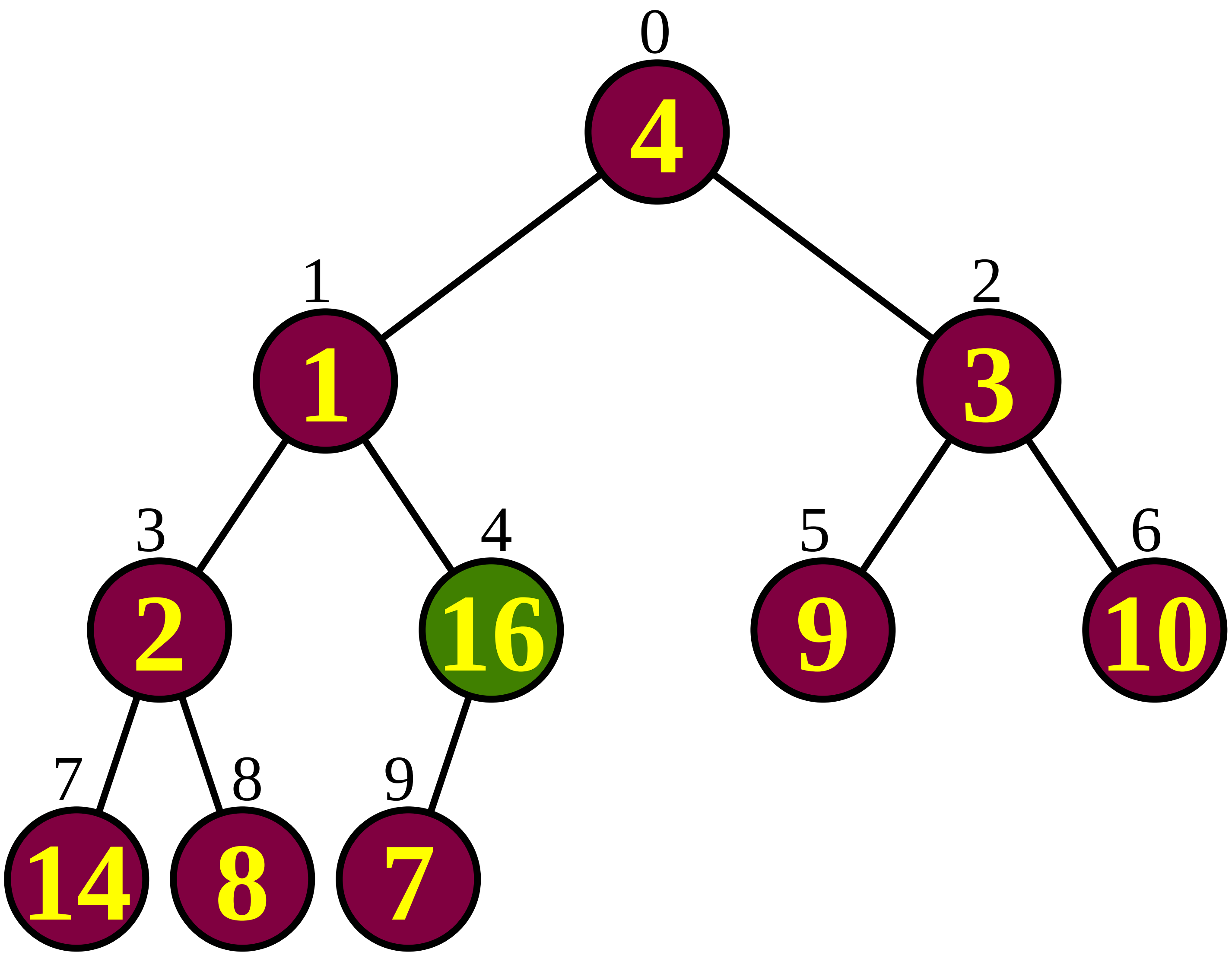
(초기 : 배열에 원소가 무작위 저장되어있음)



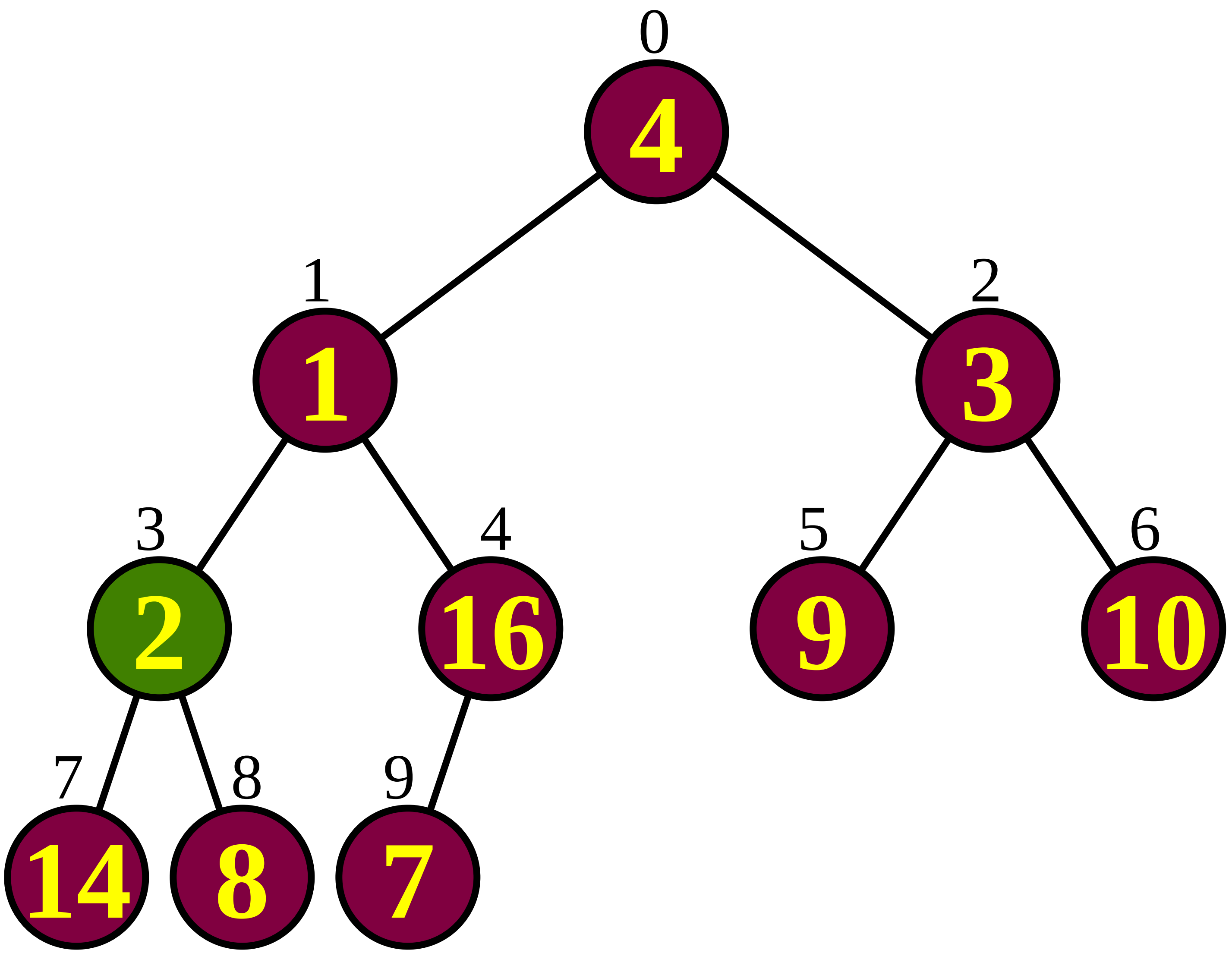
Build the heap

일단 히프만들기

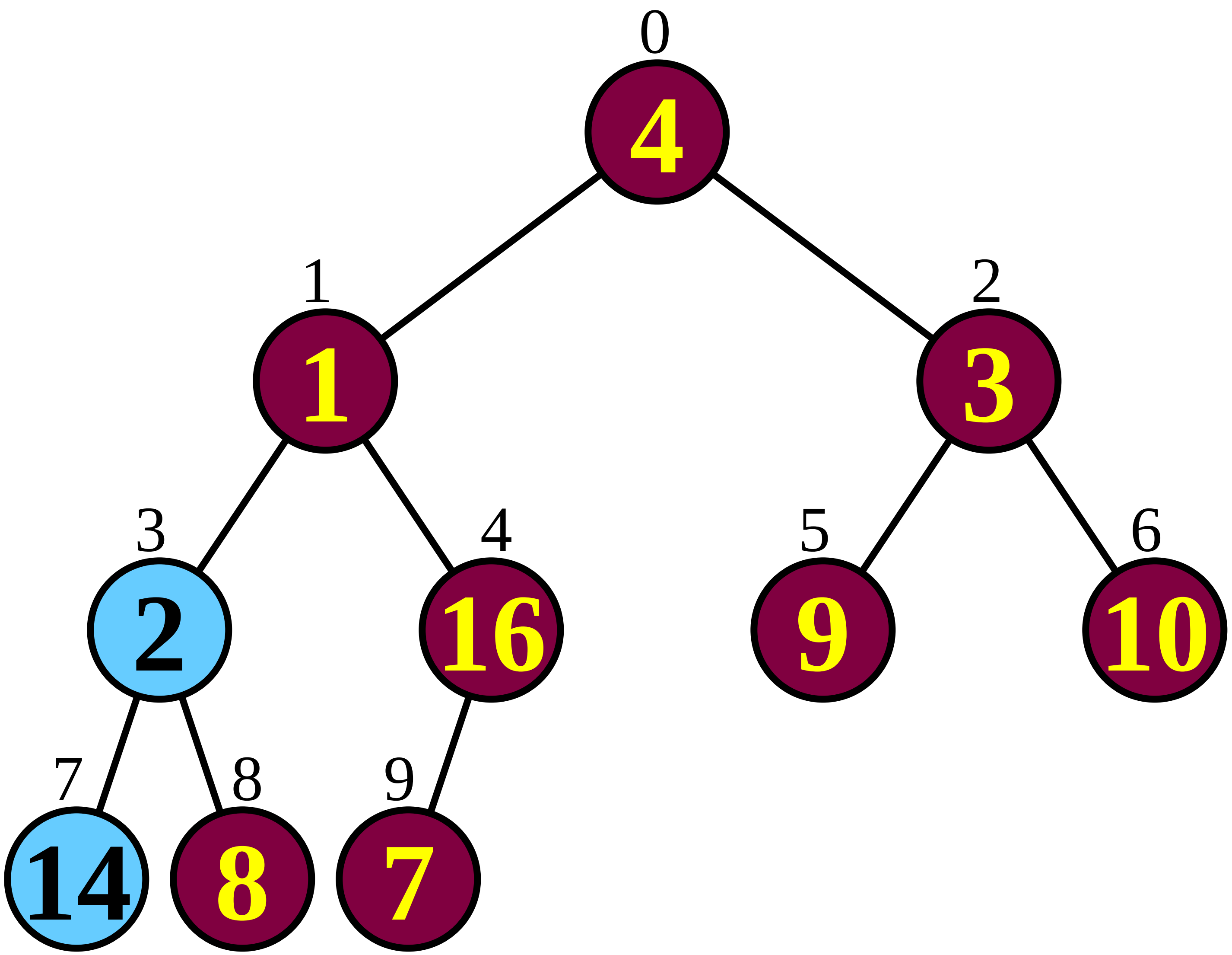
Heapsort



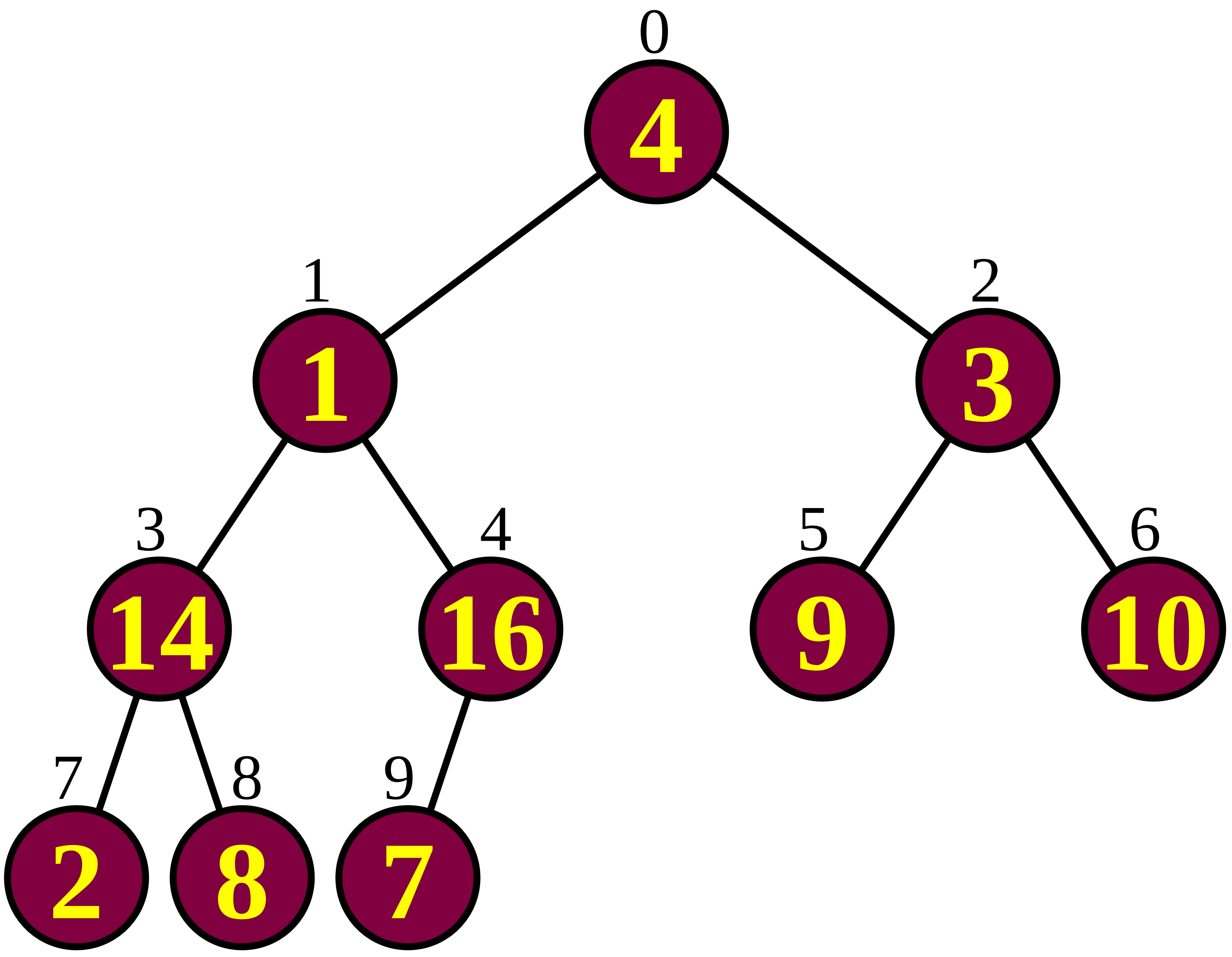
Heapsort



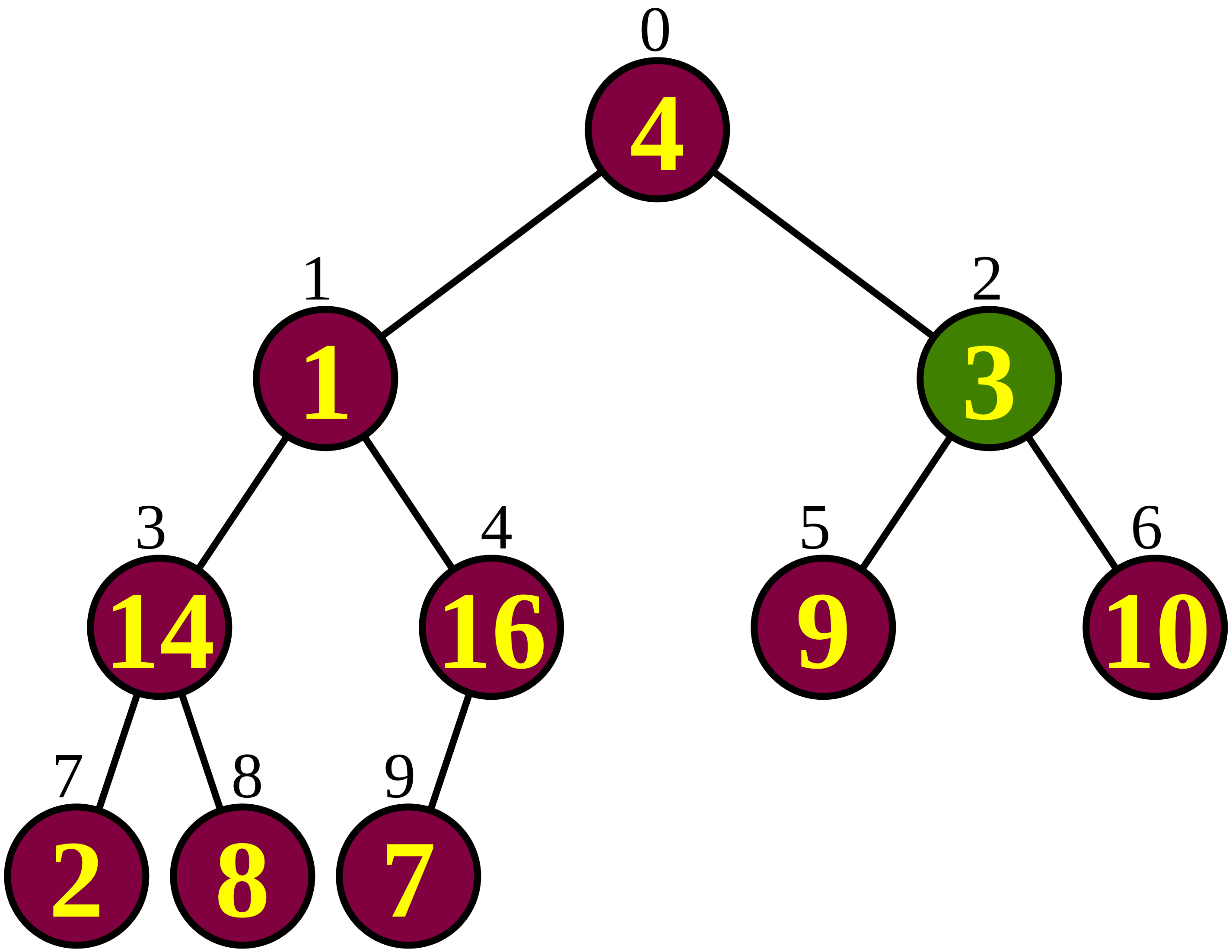
Heapsort



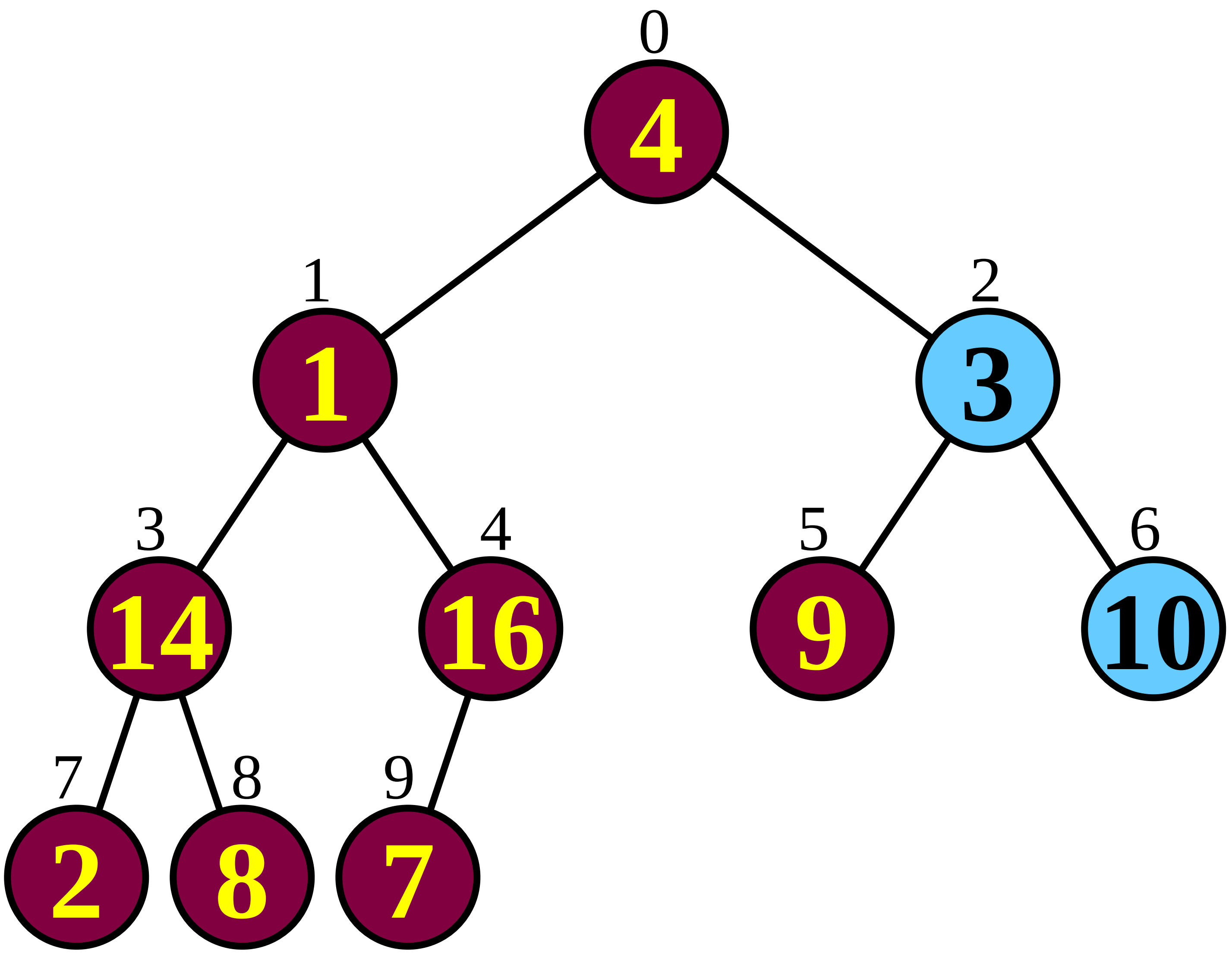
Heapsort



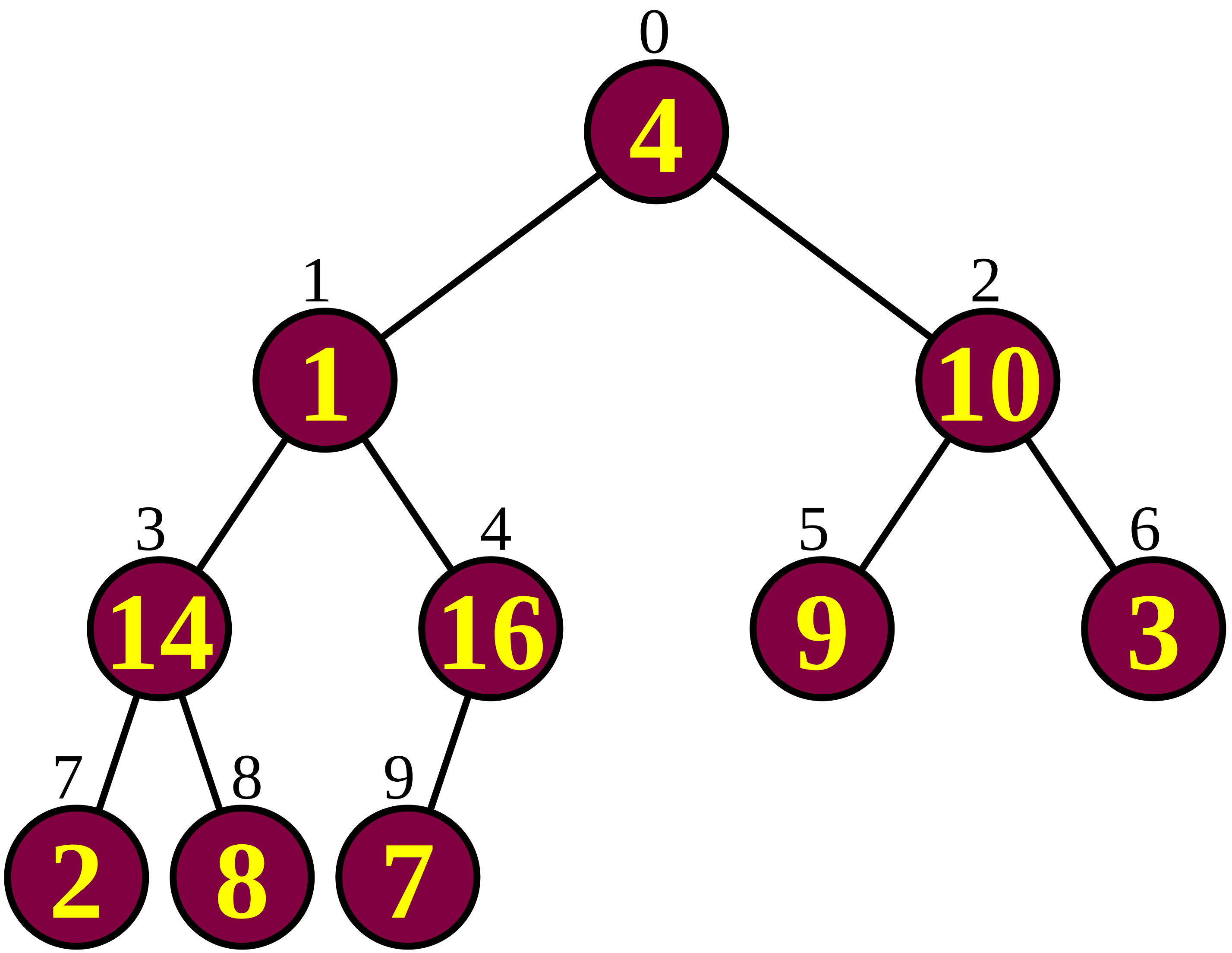
Heapsort



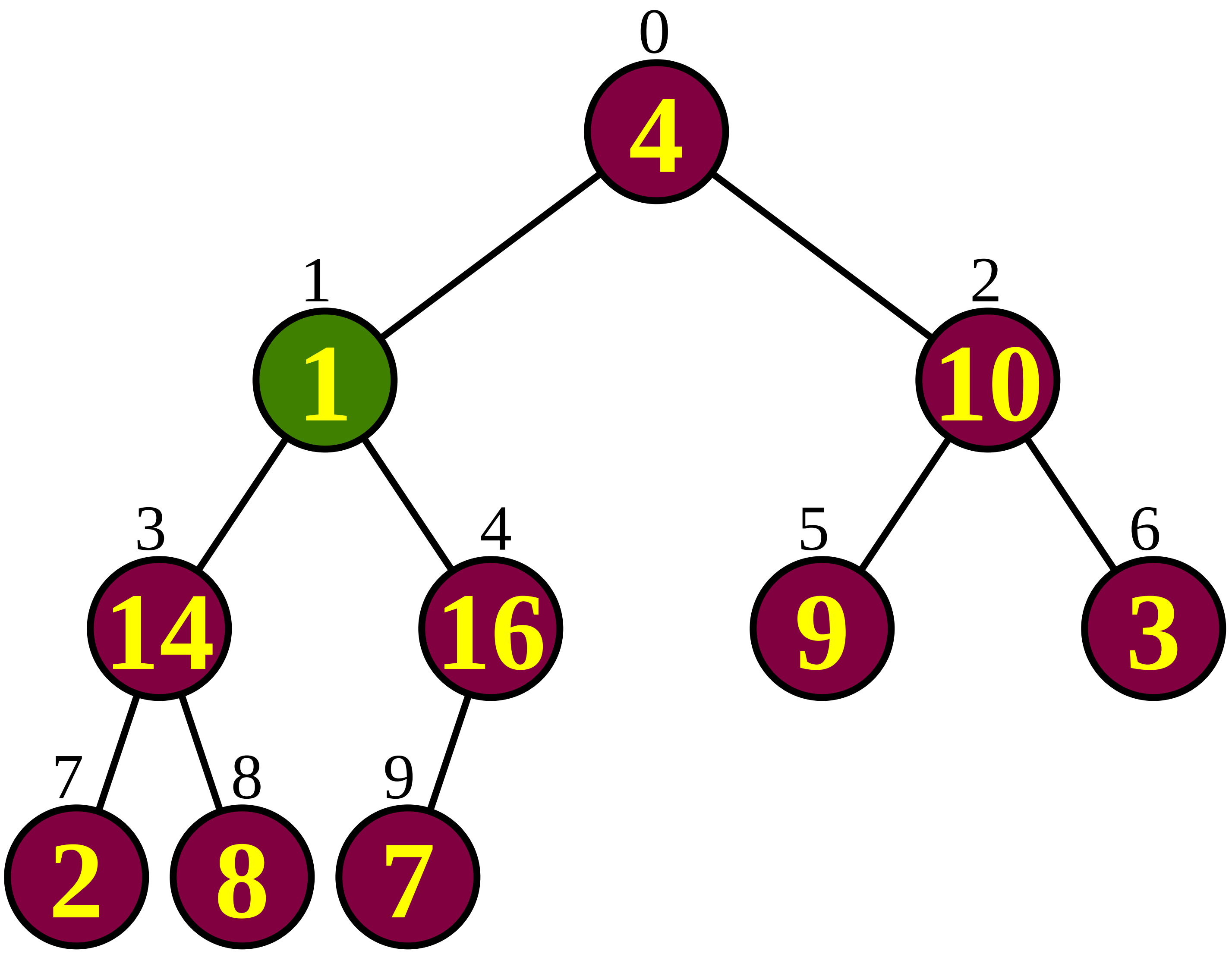
Heapsort



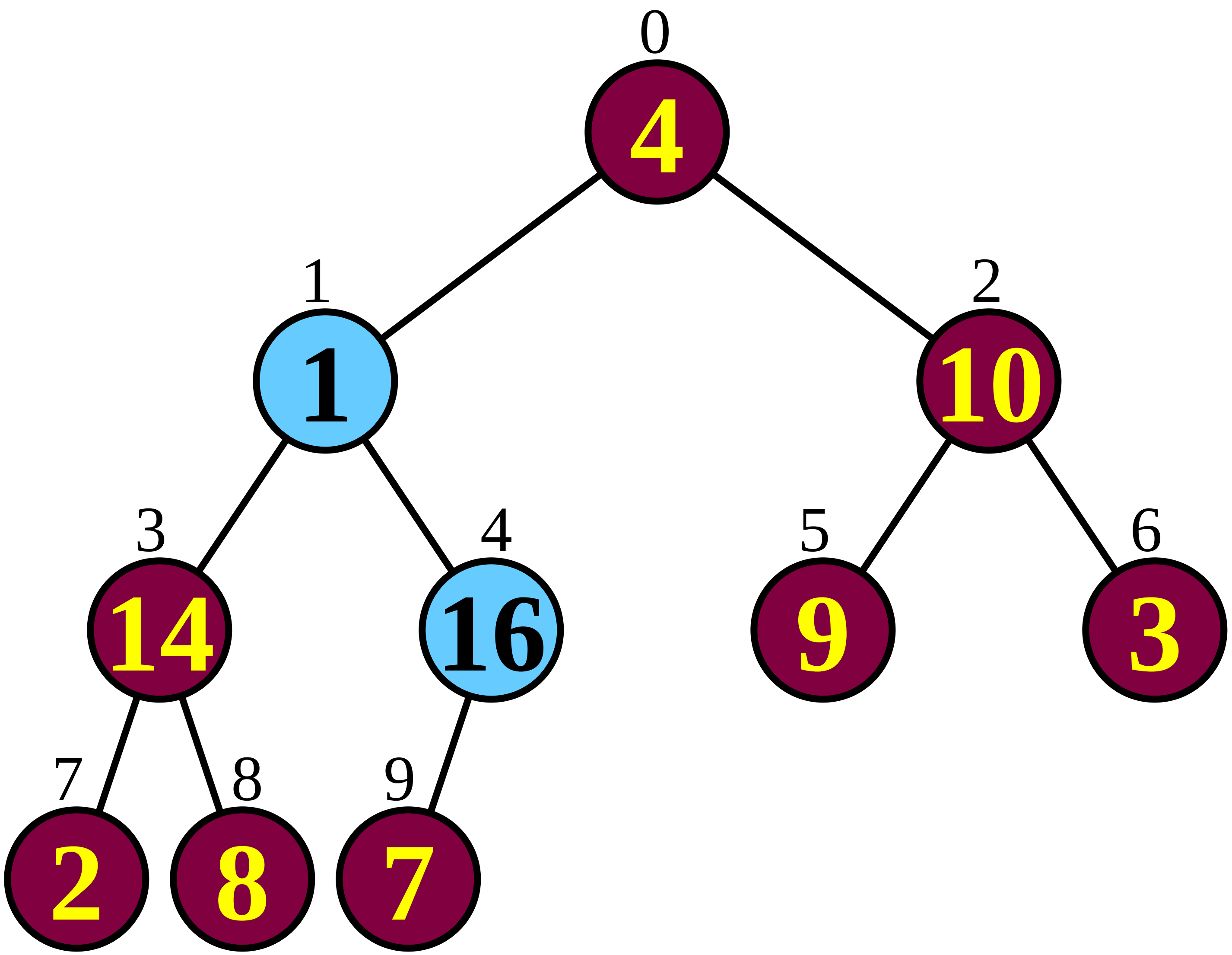
Heapsort



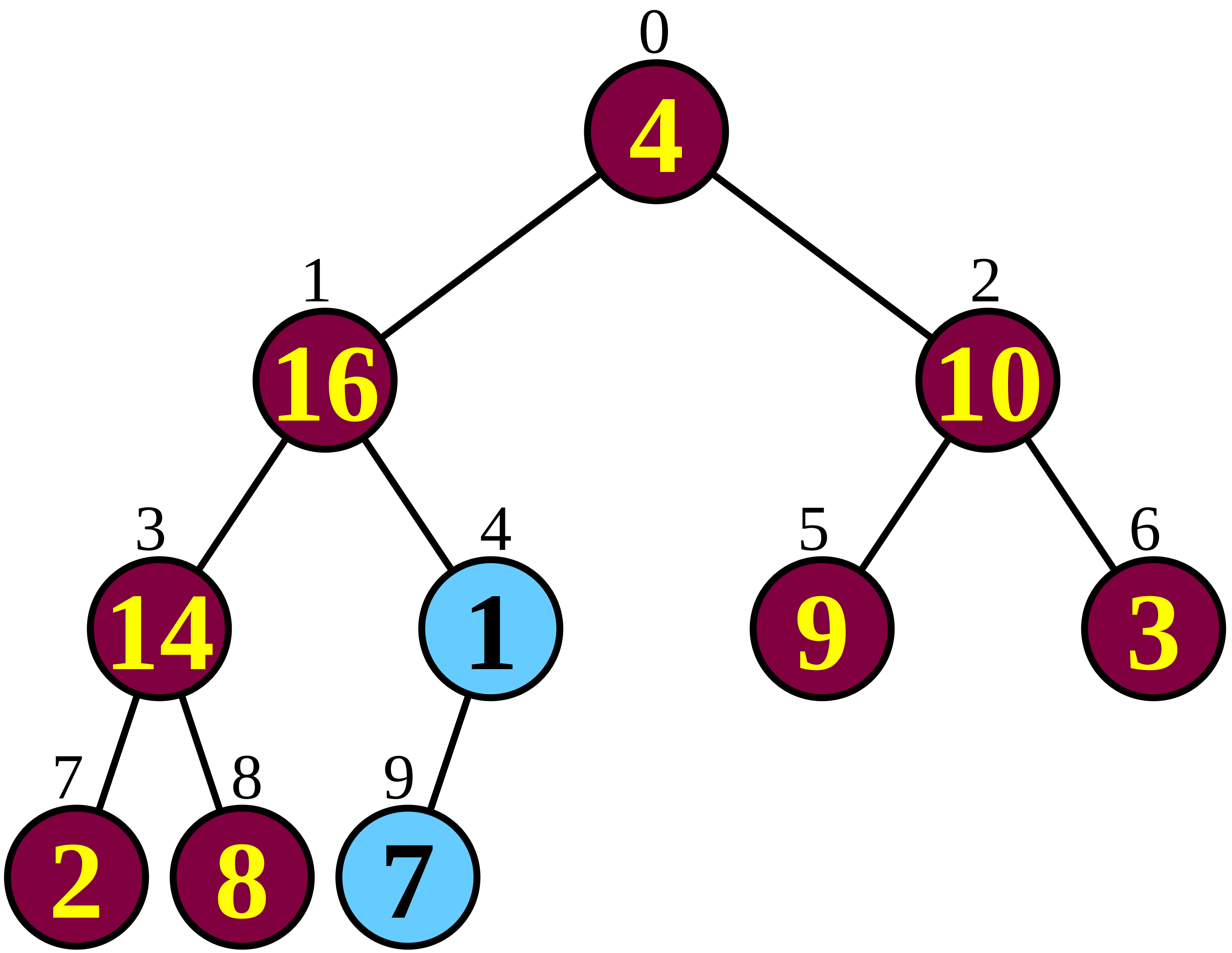
Heapsort



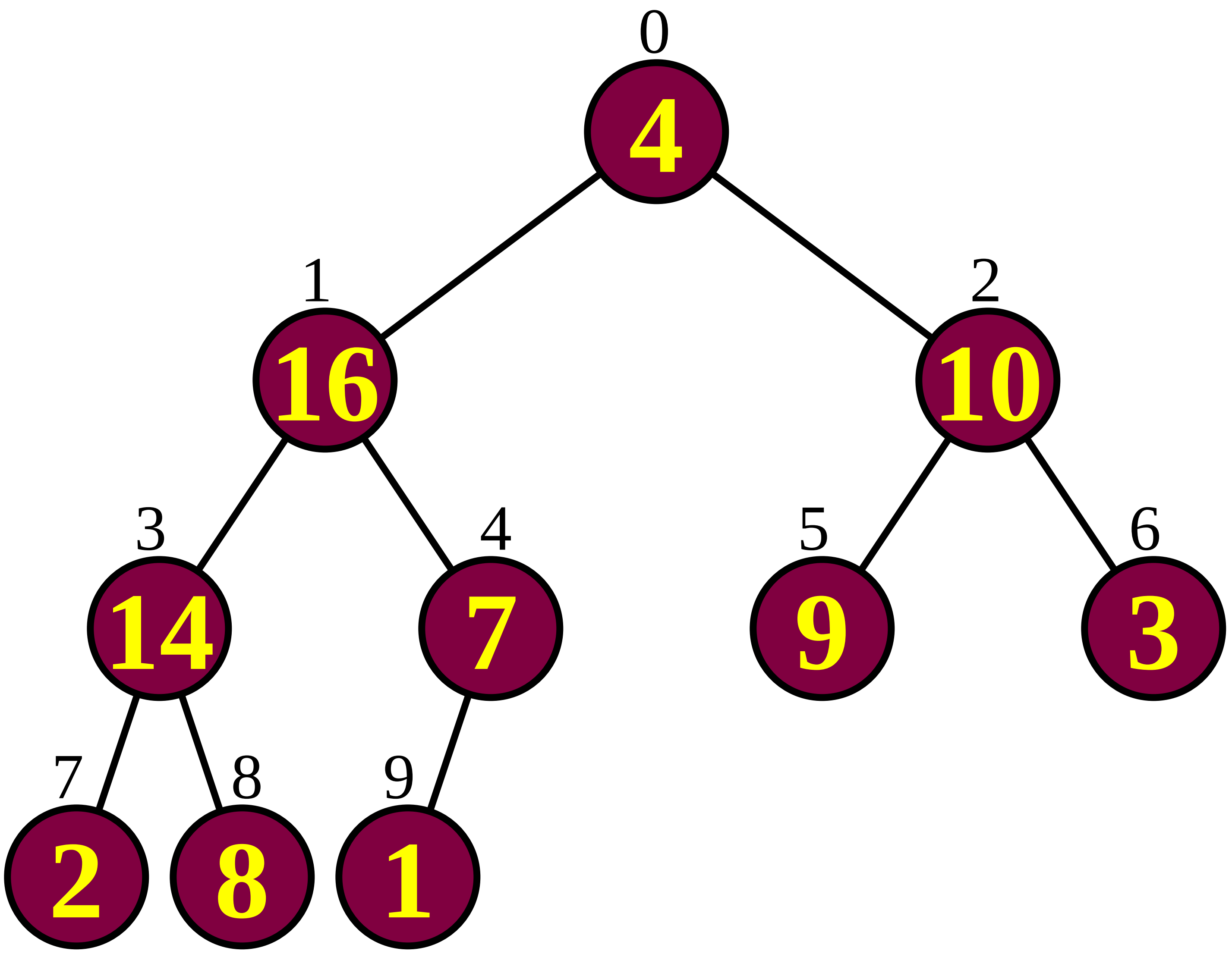
Heapsort



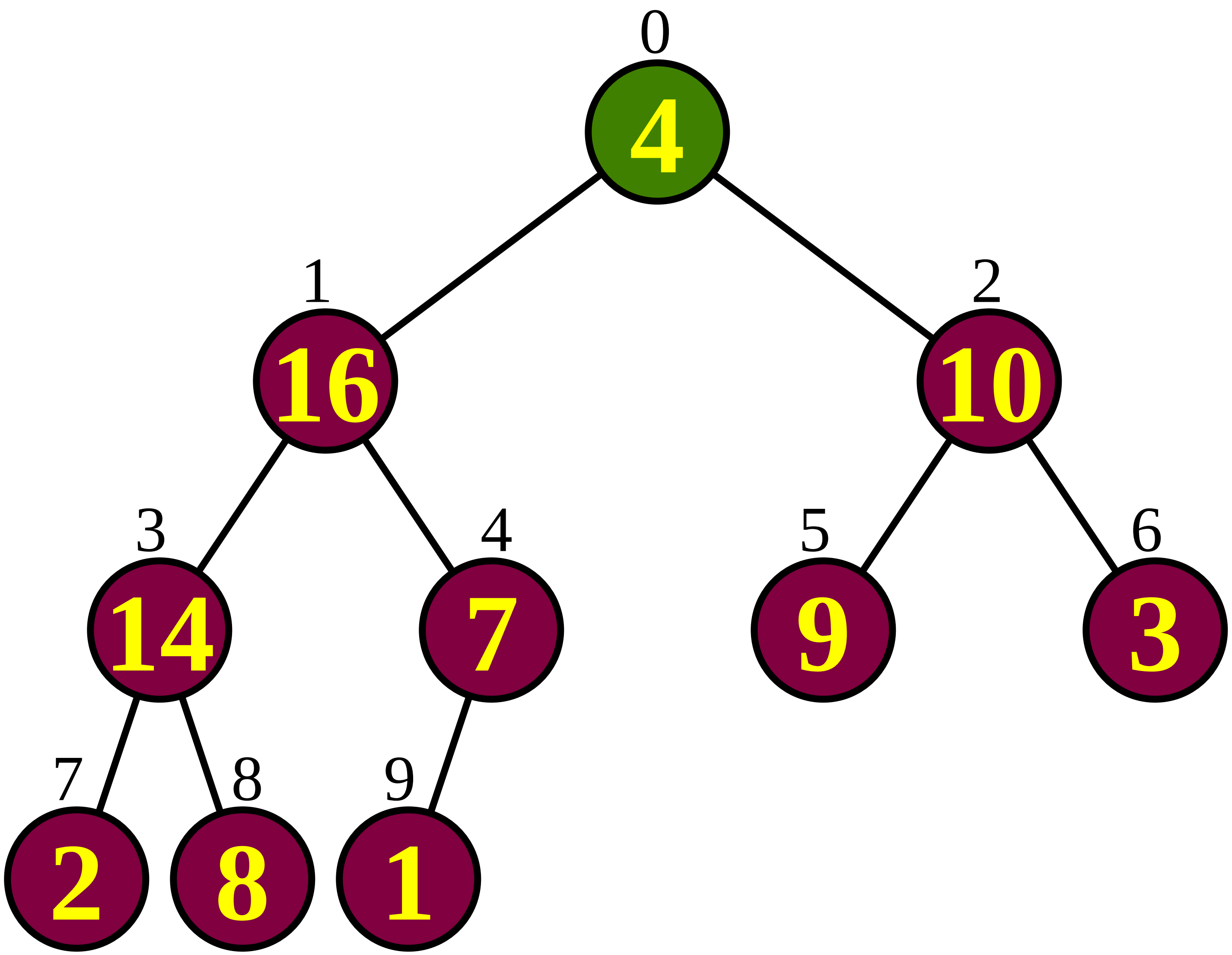
Heapsort



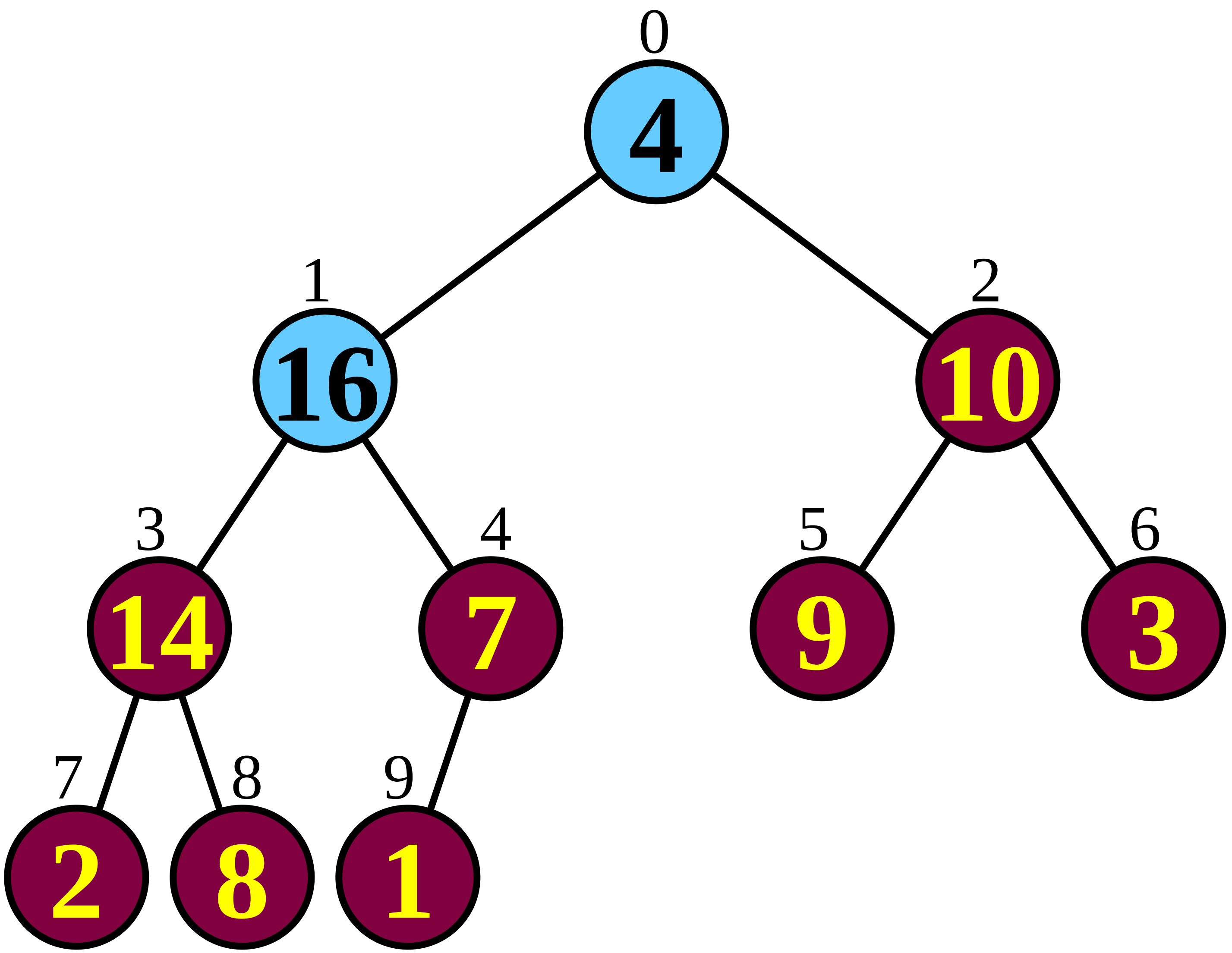
Heapsort



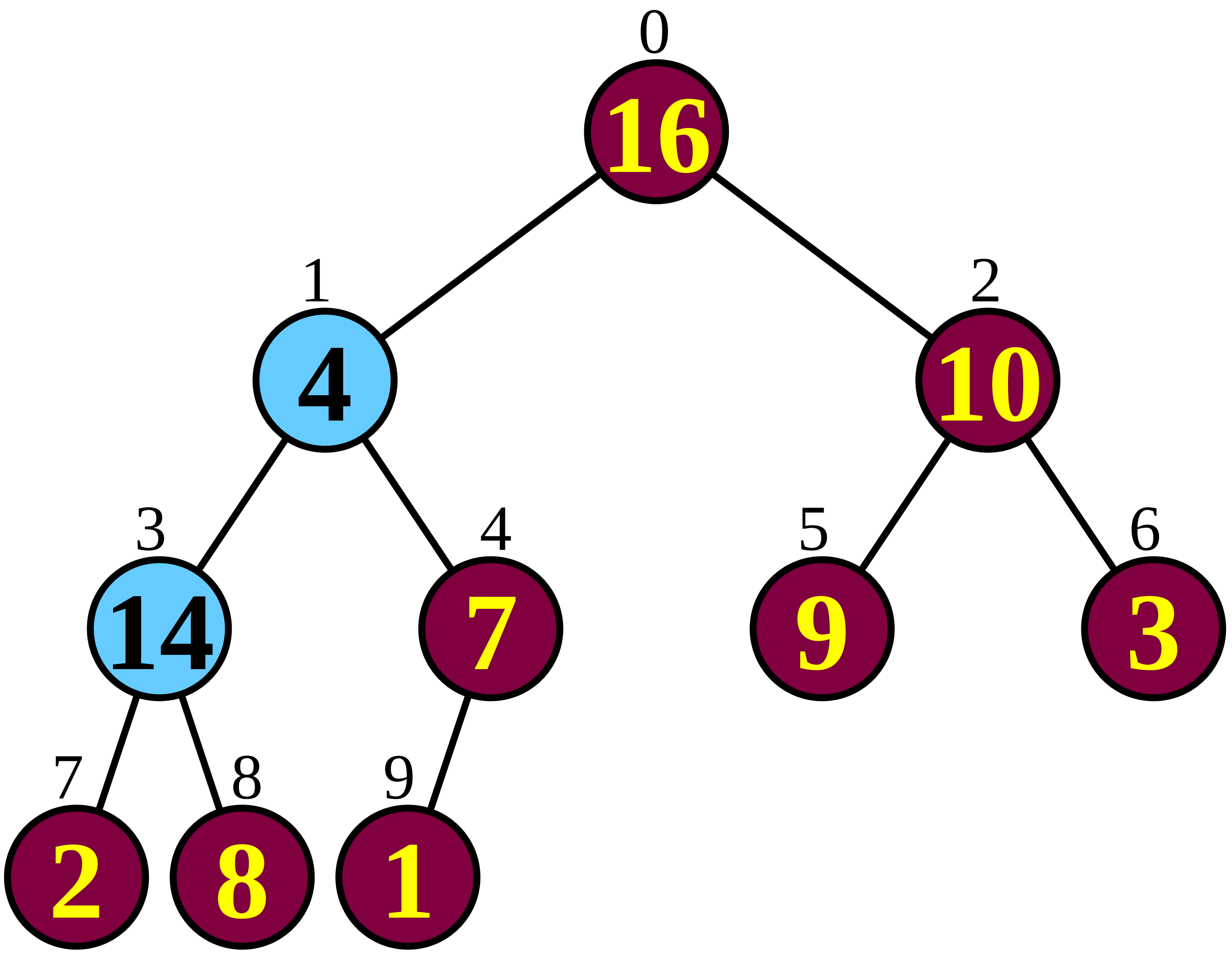
Heapsort



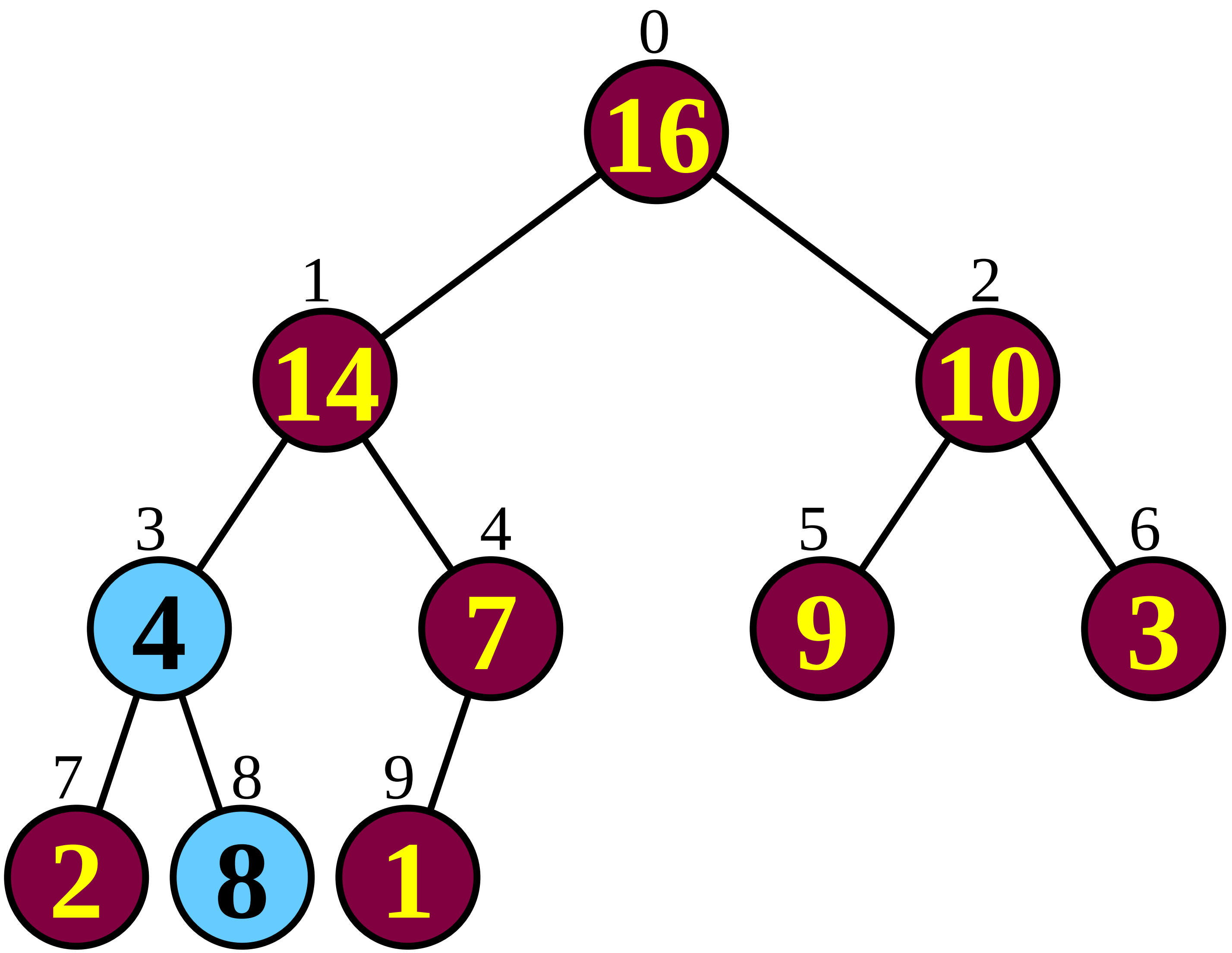
Heapsort



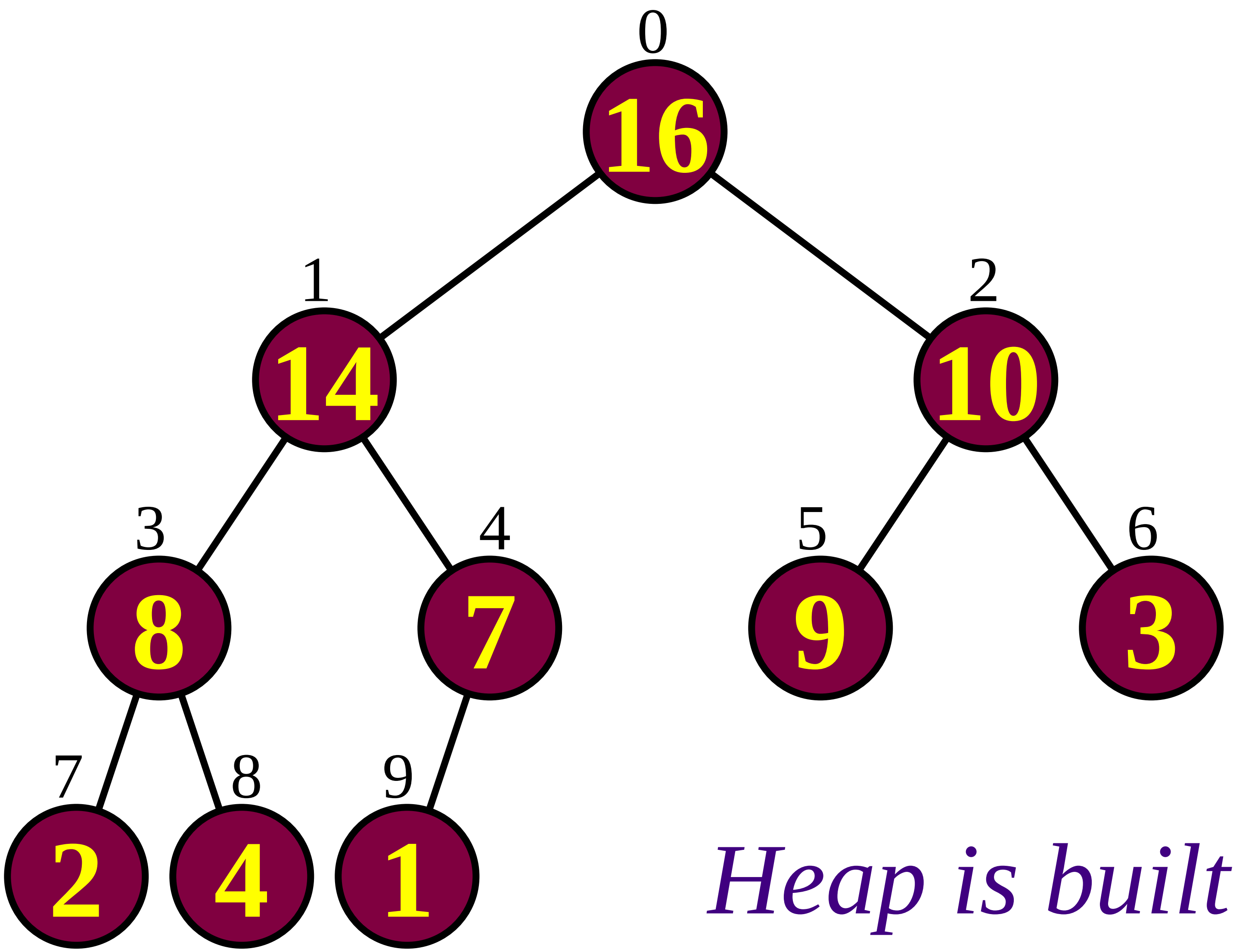
Heapsort



Heapsort

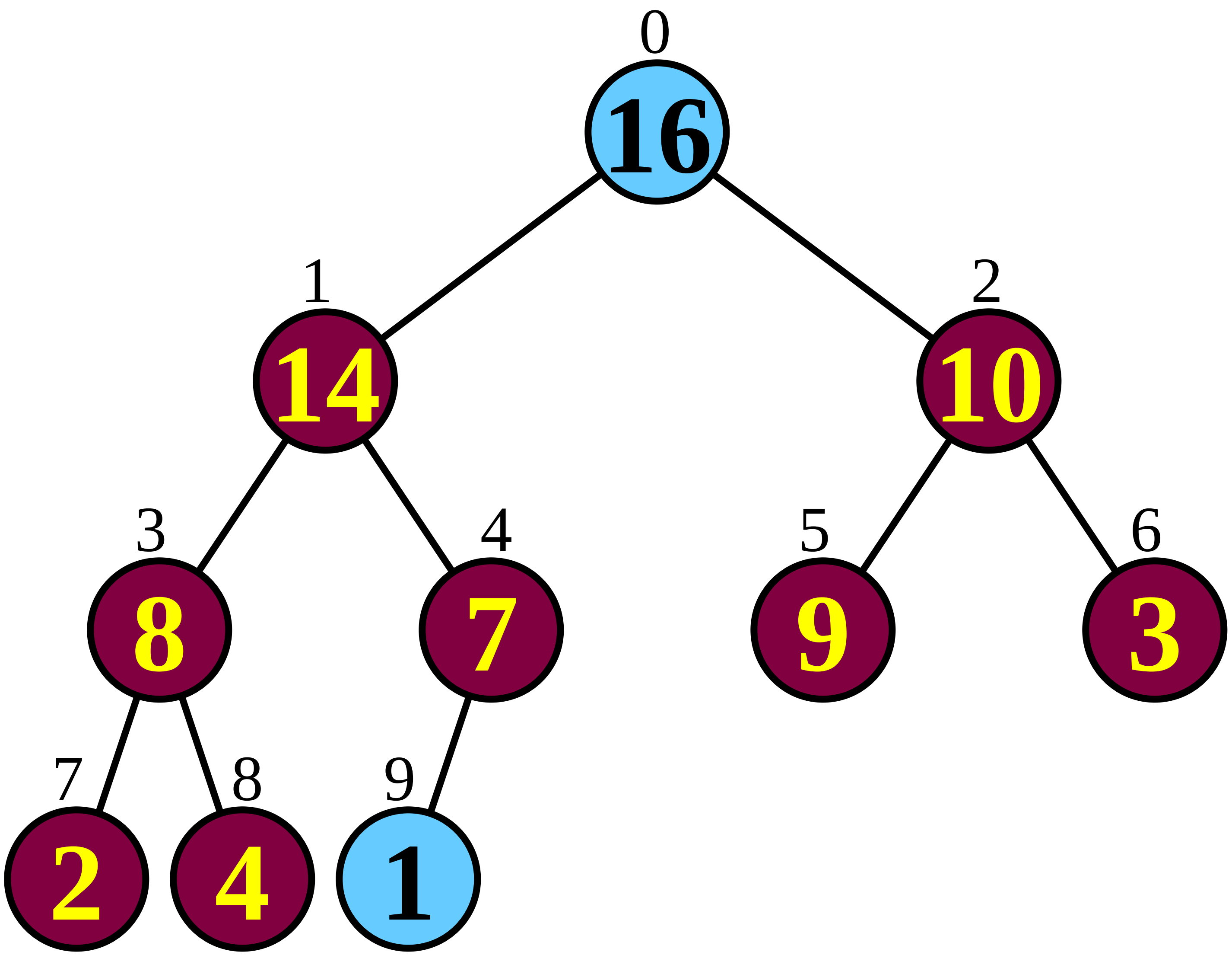


Heapsort

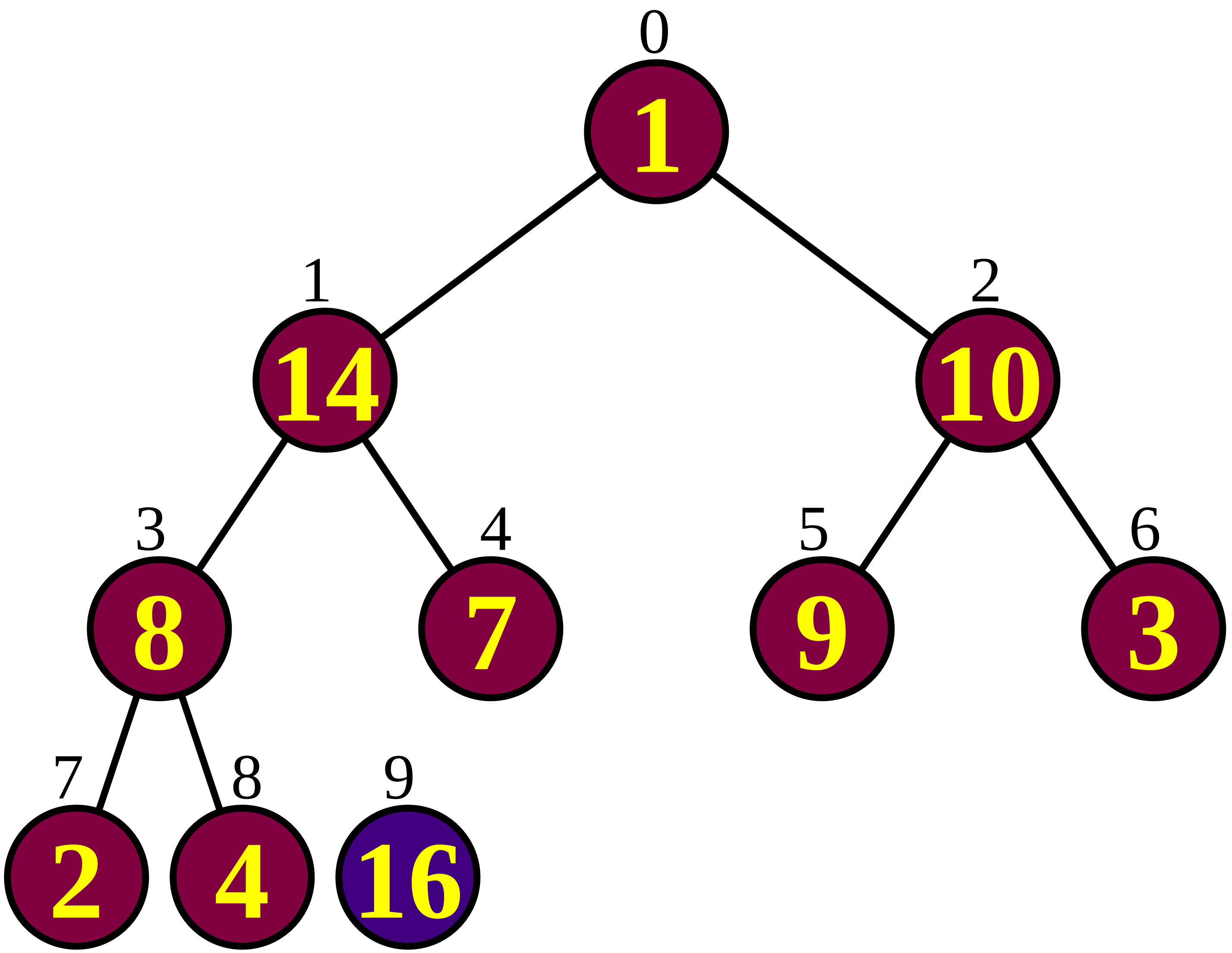


Heap is built
히프가 완성

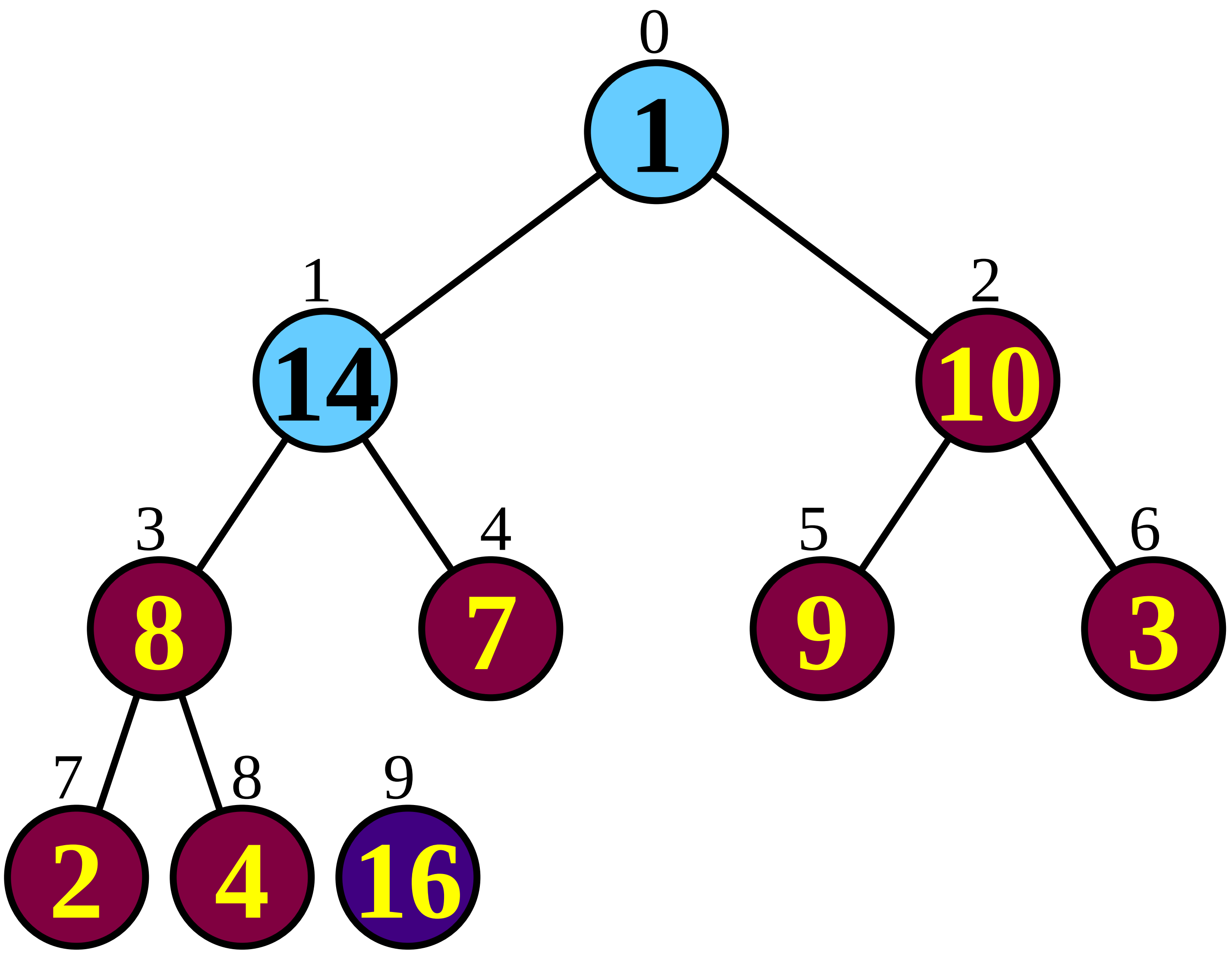
Heapsort



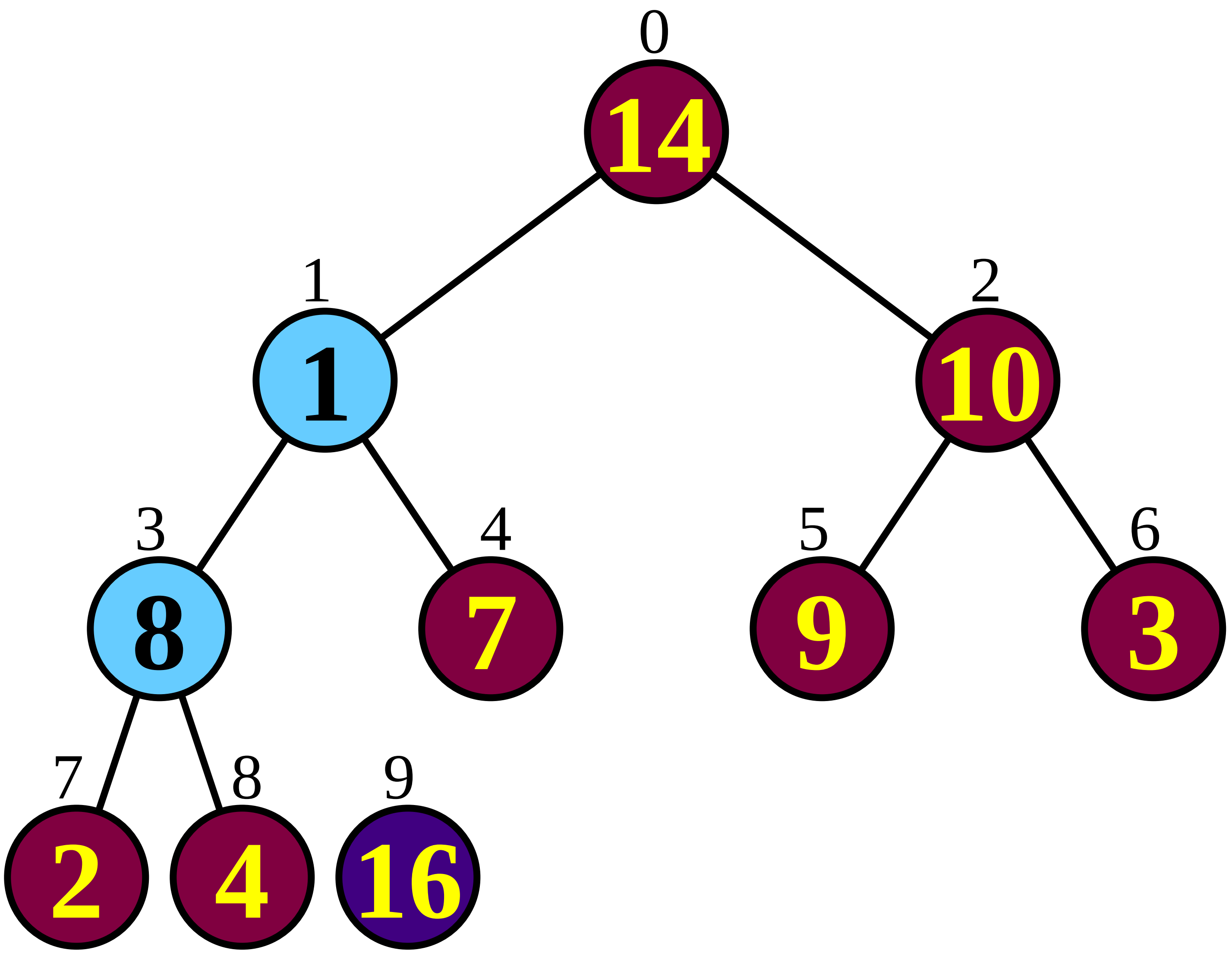
Heapsort



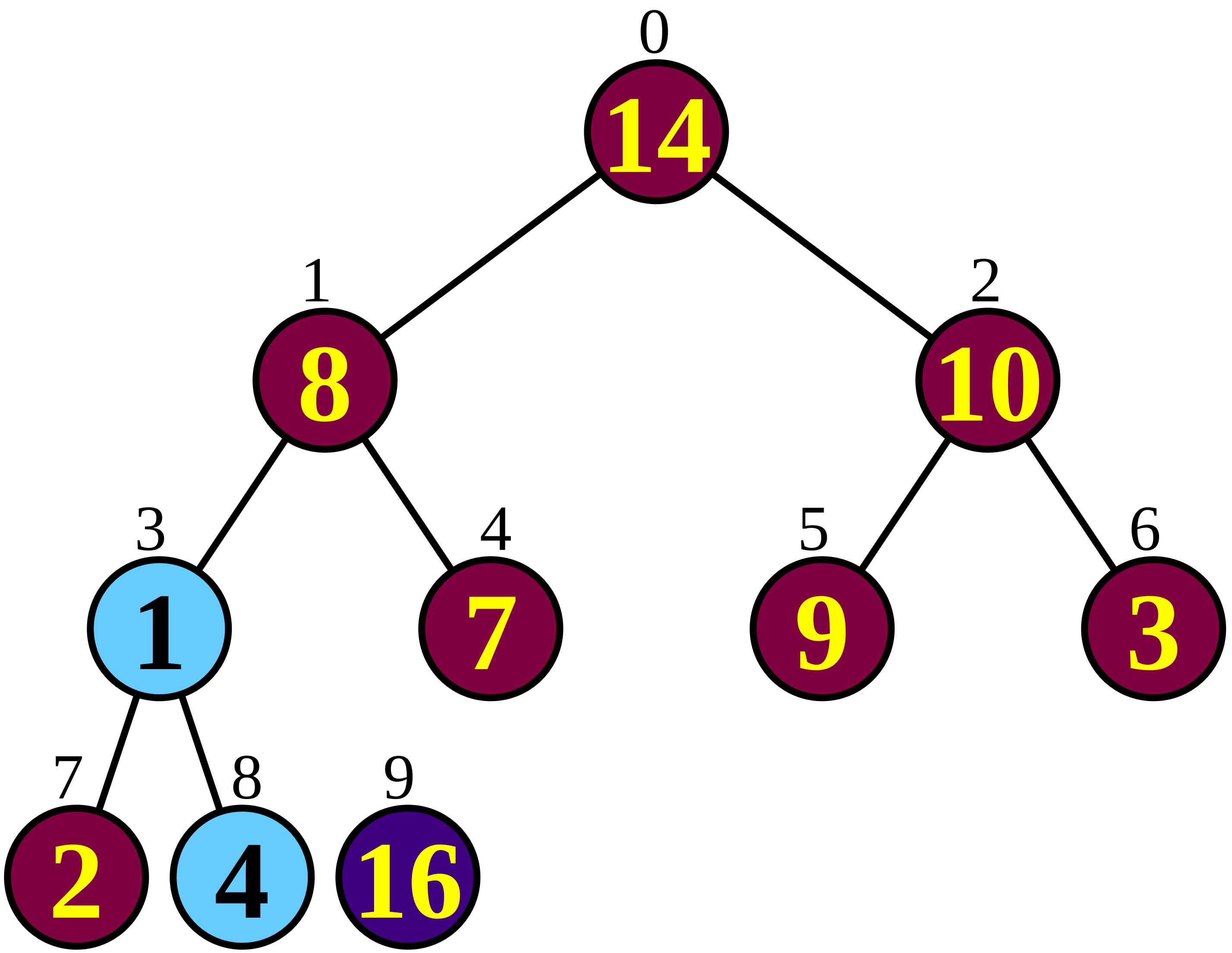
Heapsort



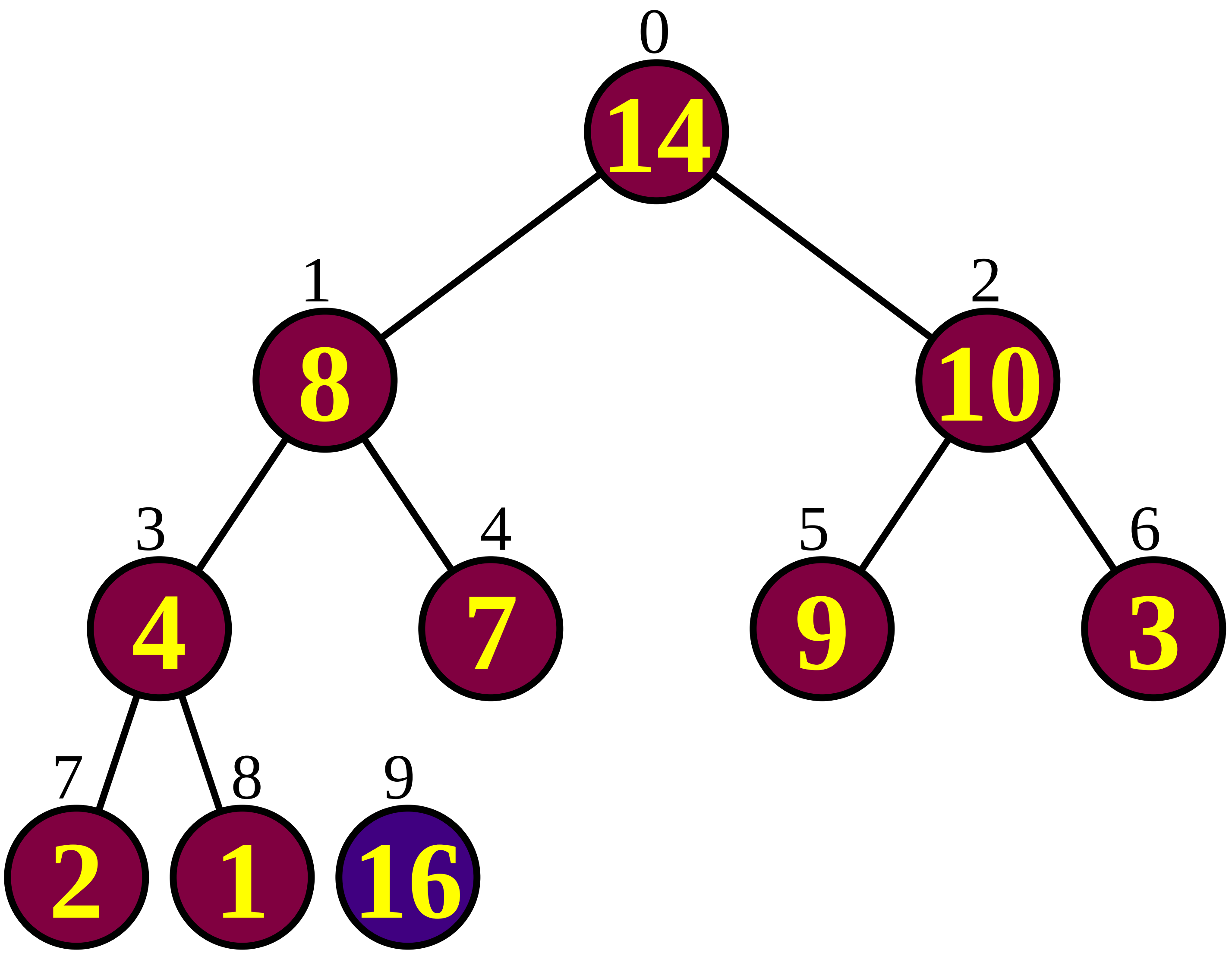
Heapsort



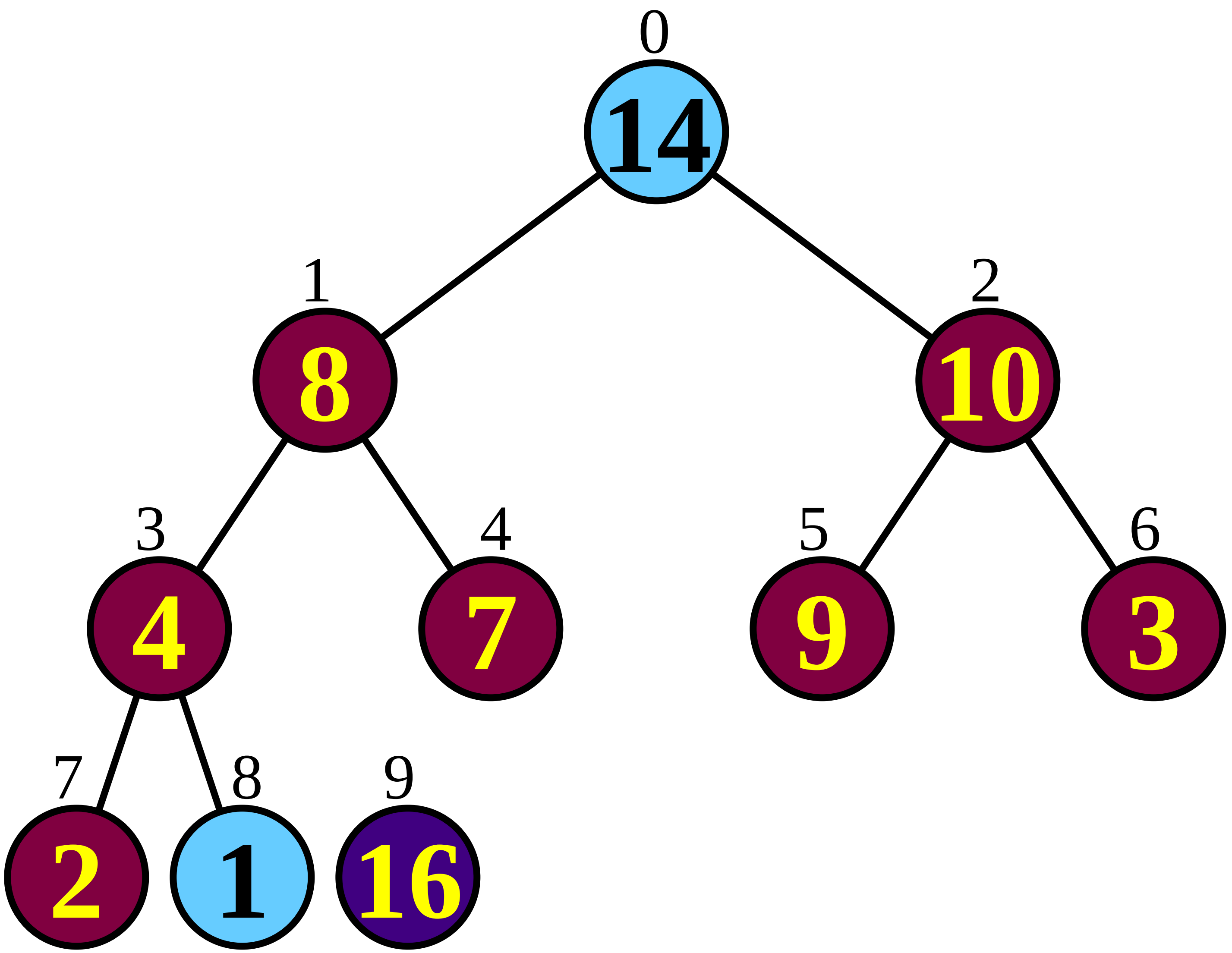
Heapsort



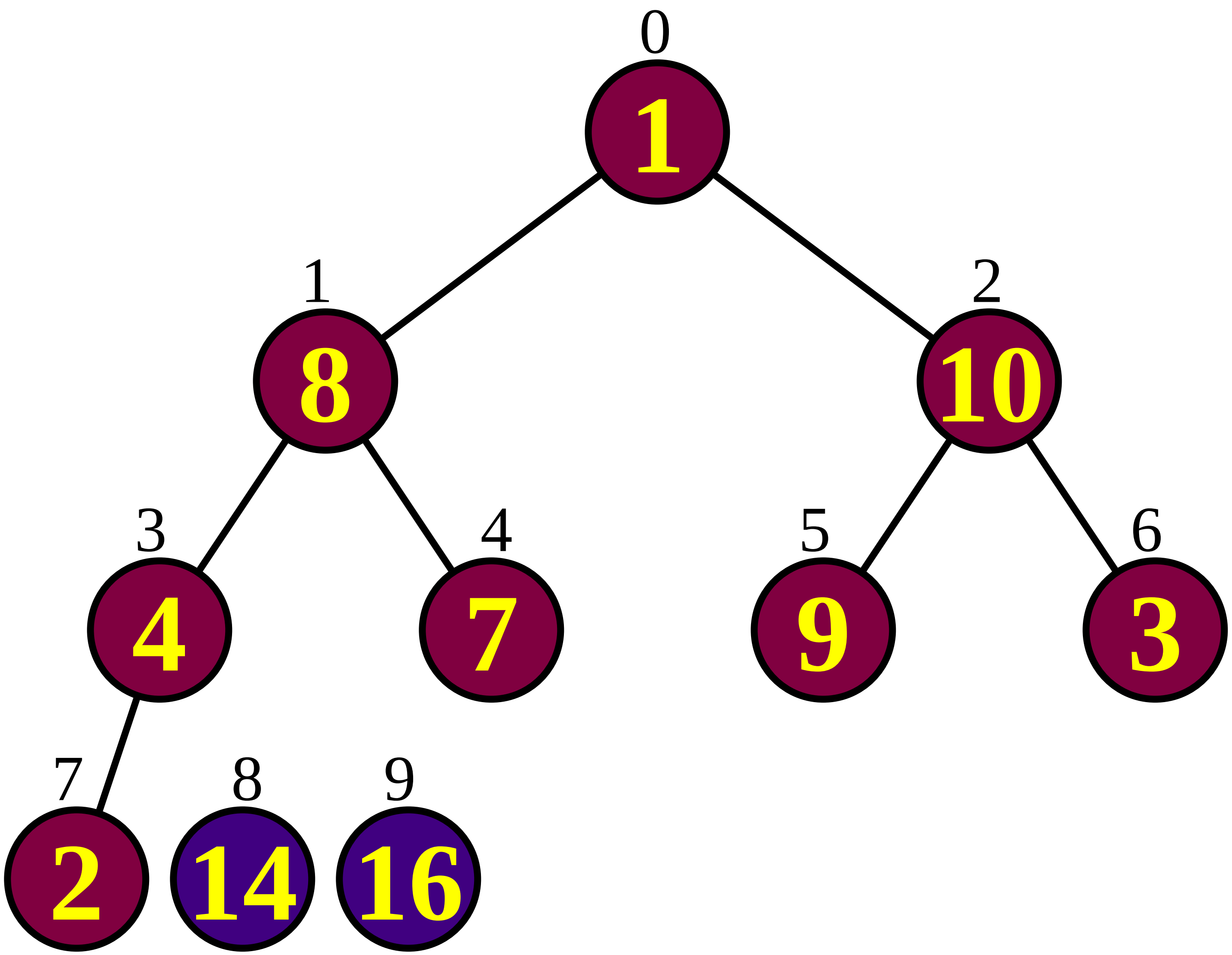
Heapsort



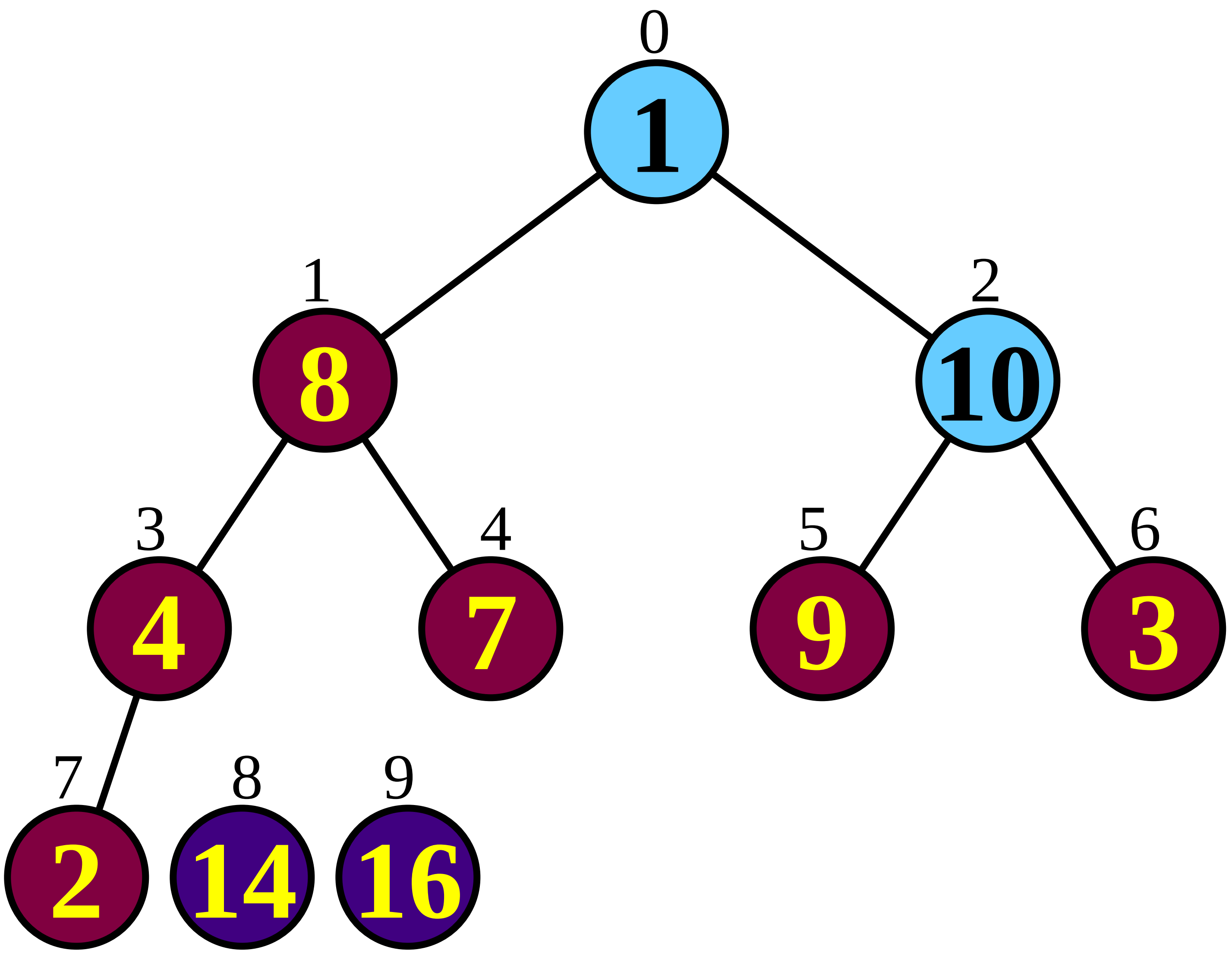
Heapsort



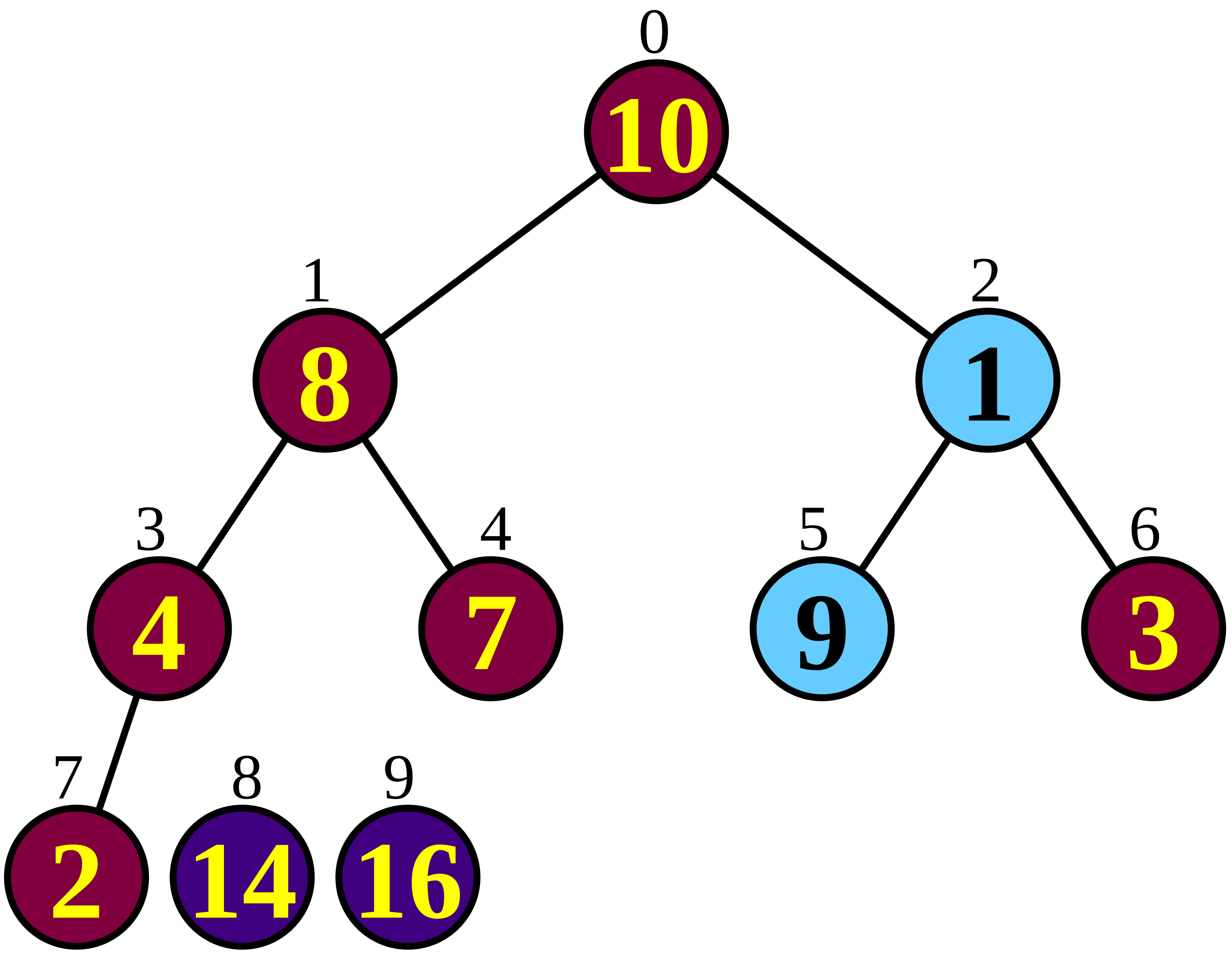
Heapsort



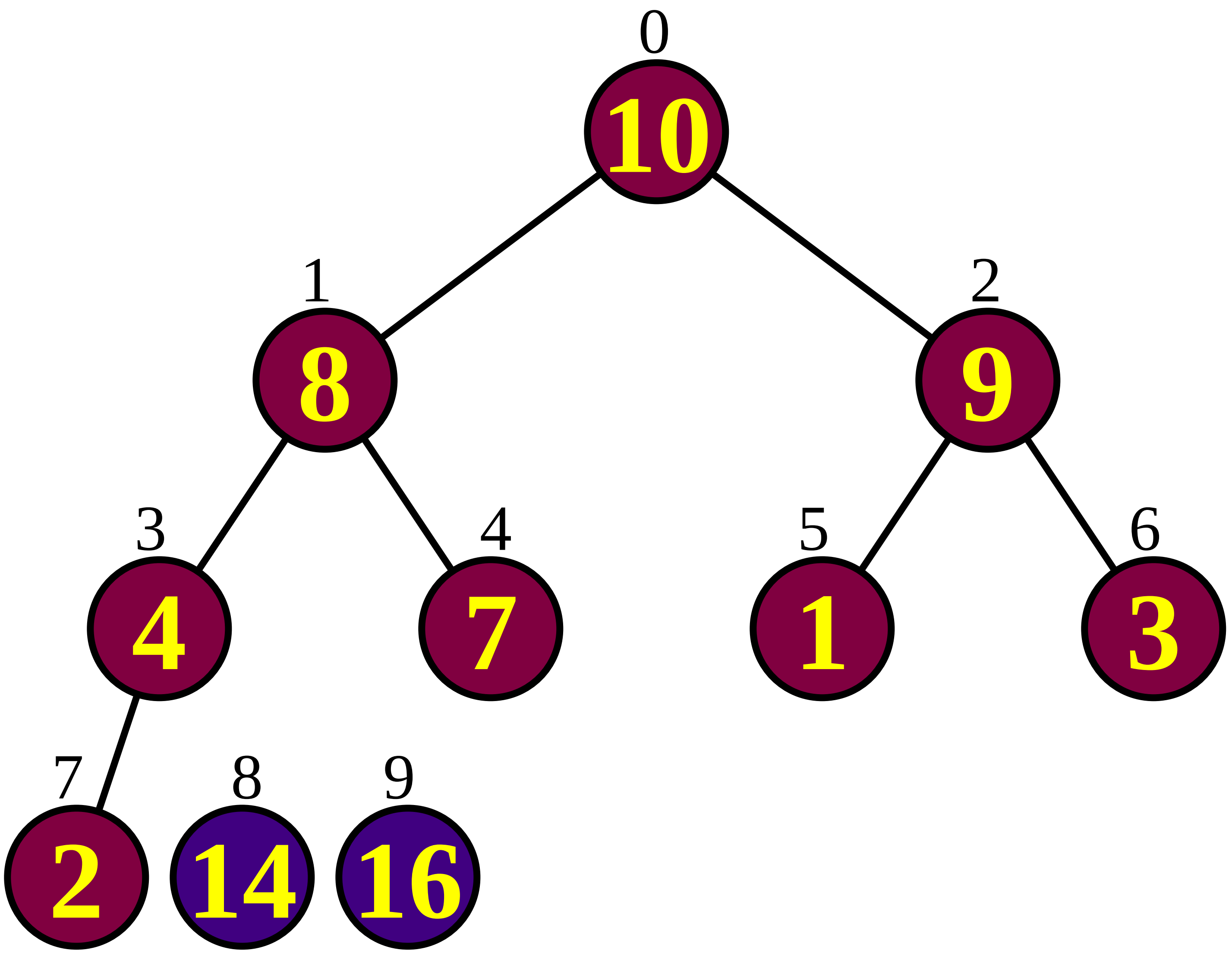
Heapsort



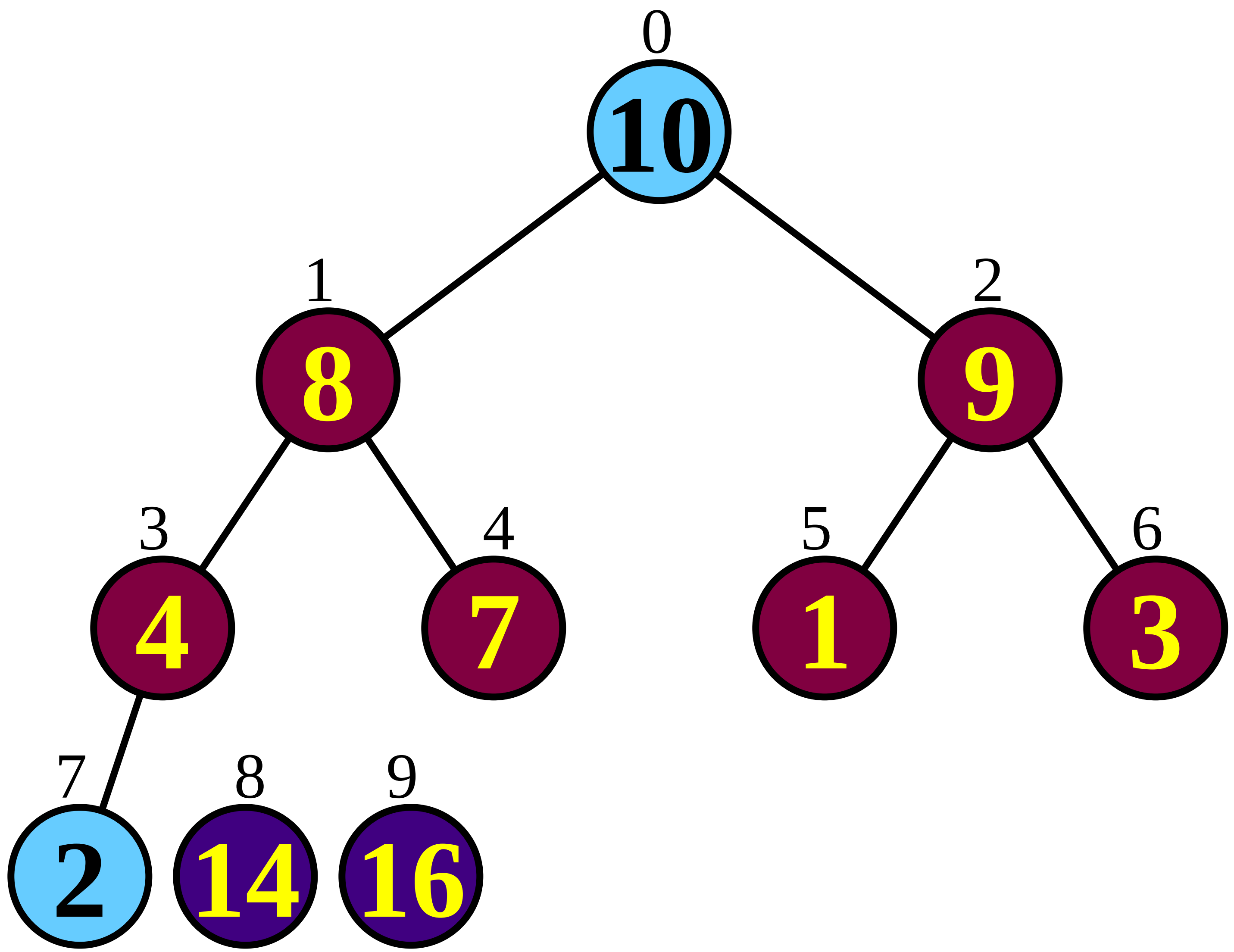
Heapsort



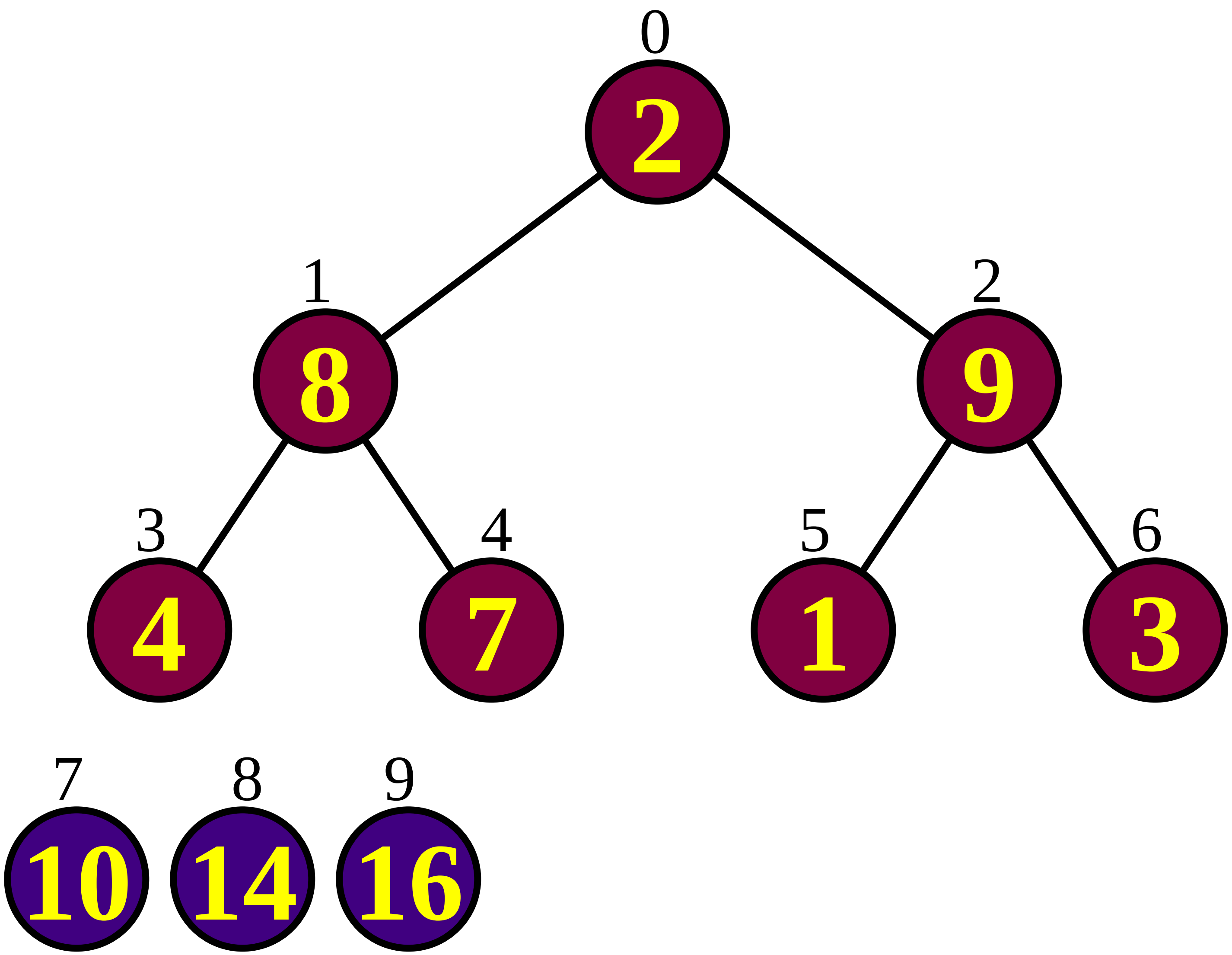
Heapsort



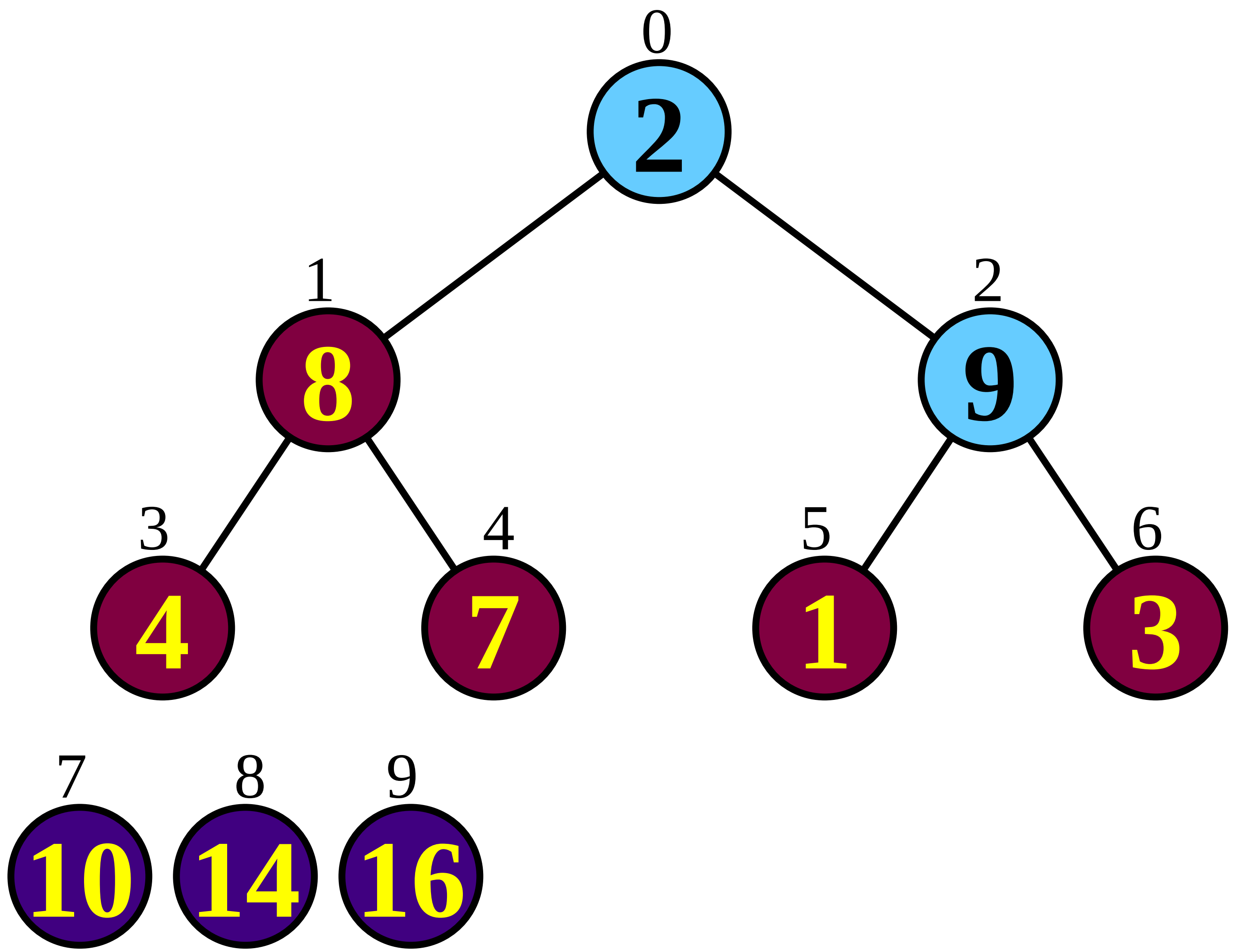
Heapsort



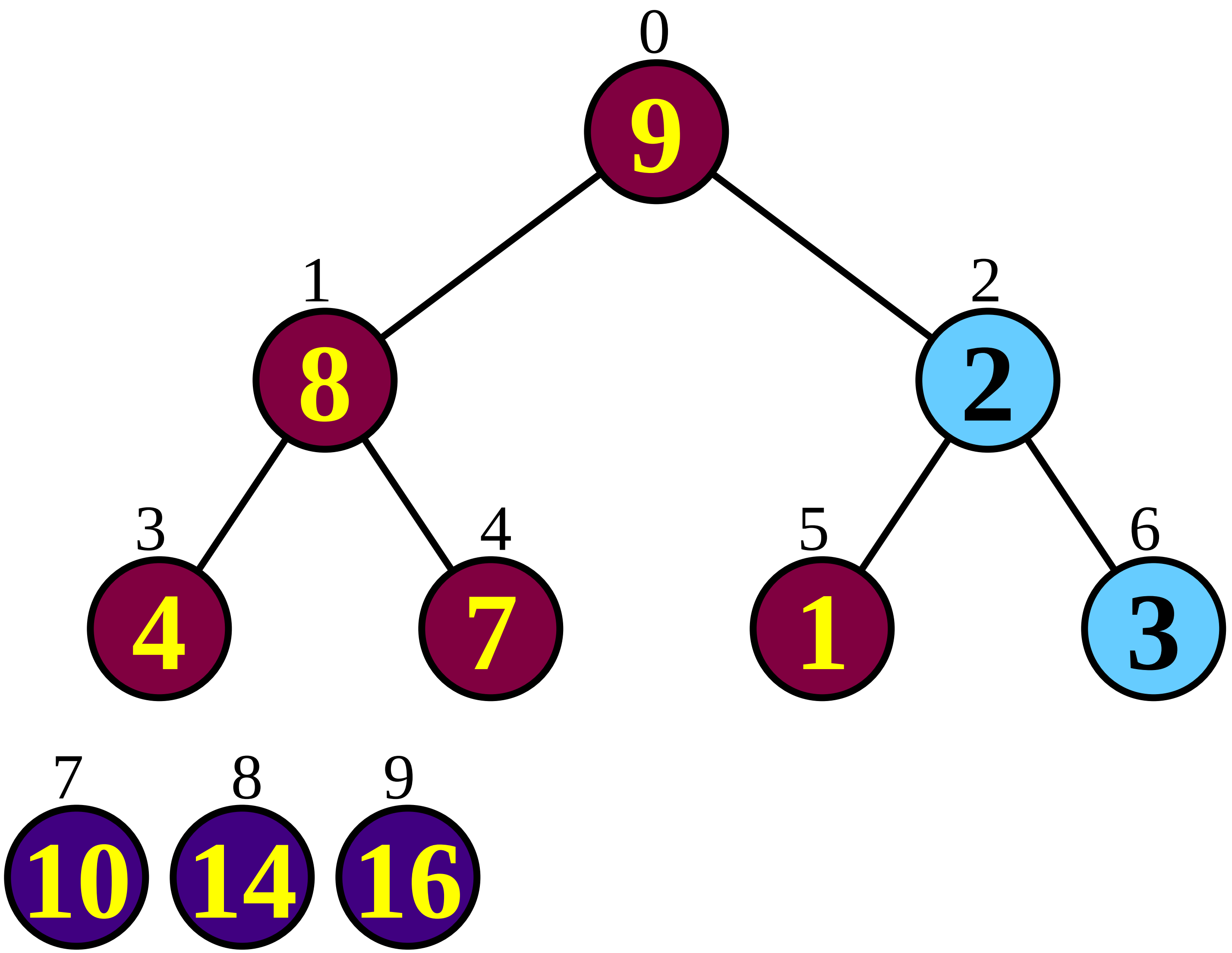
Heapsort



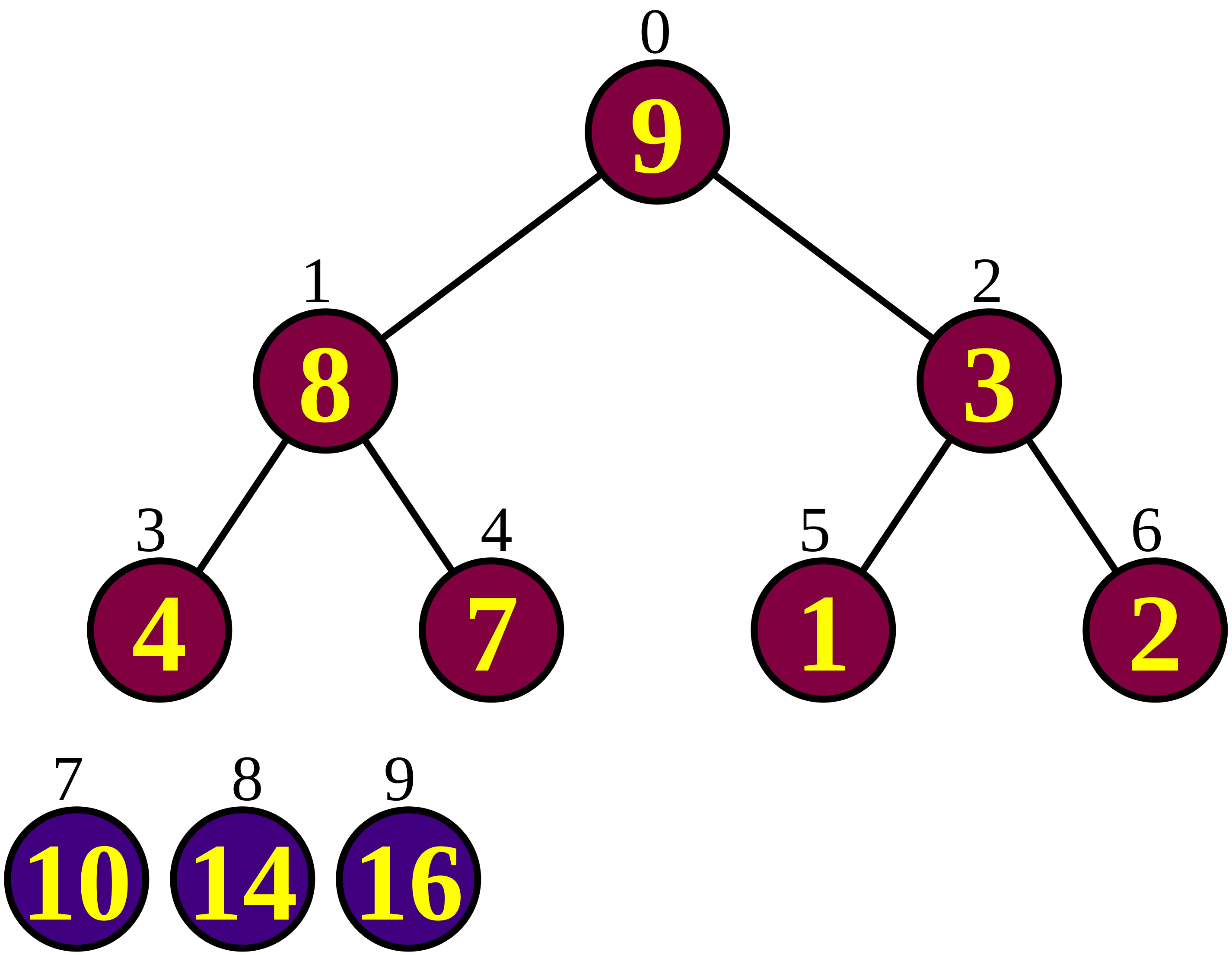
Heapsort



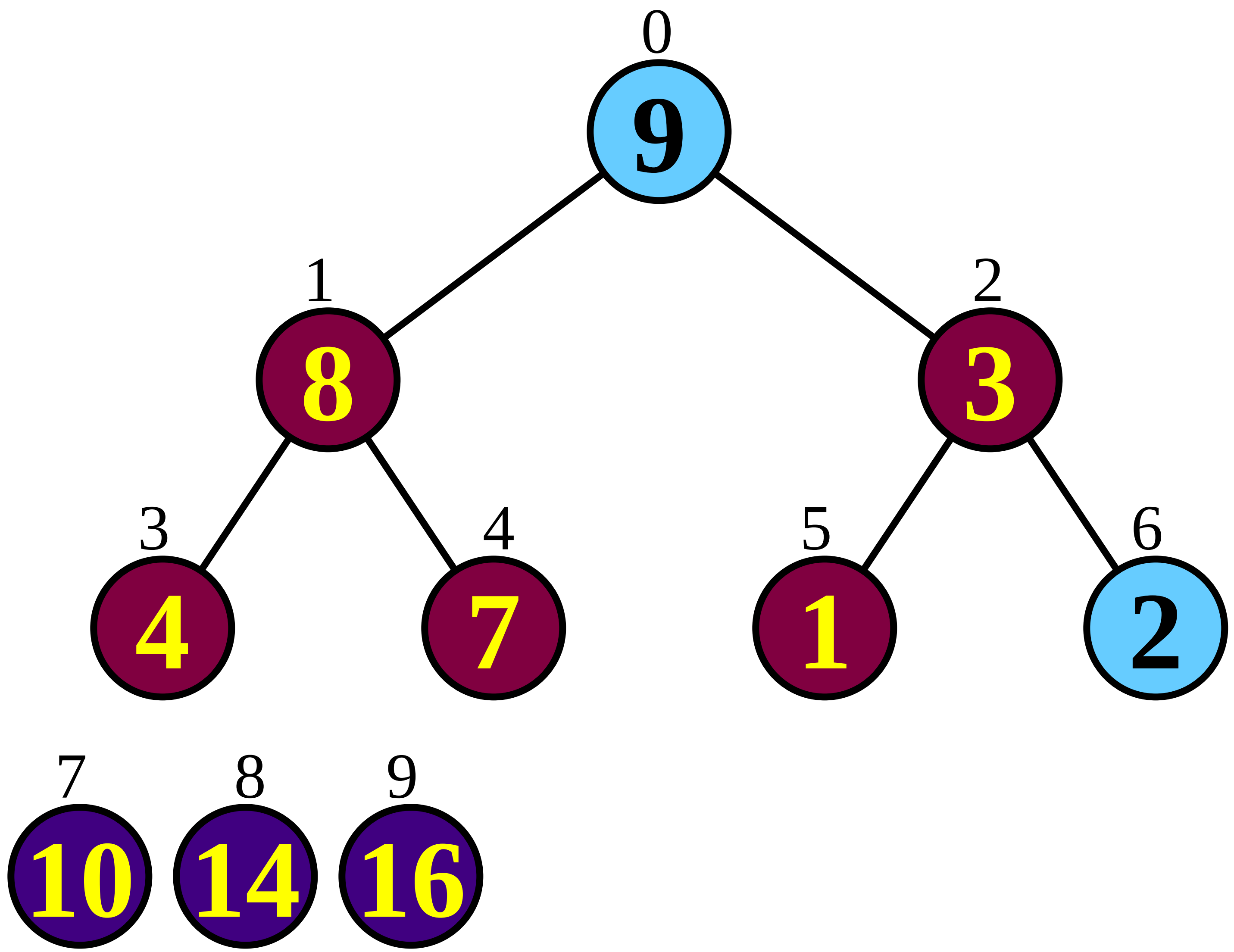
Heapsort



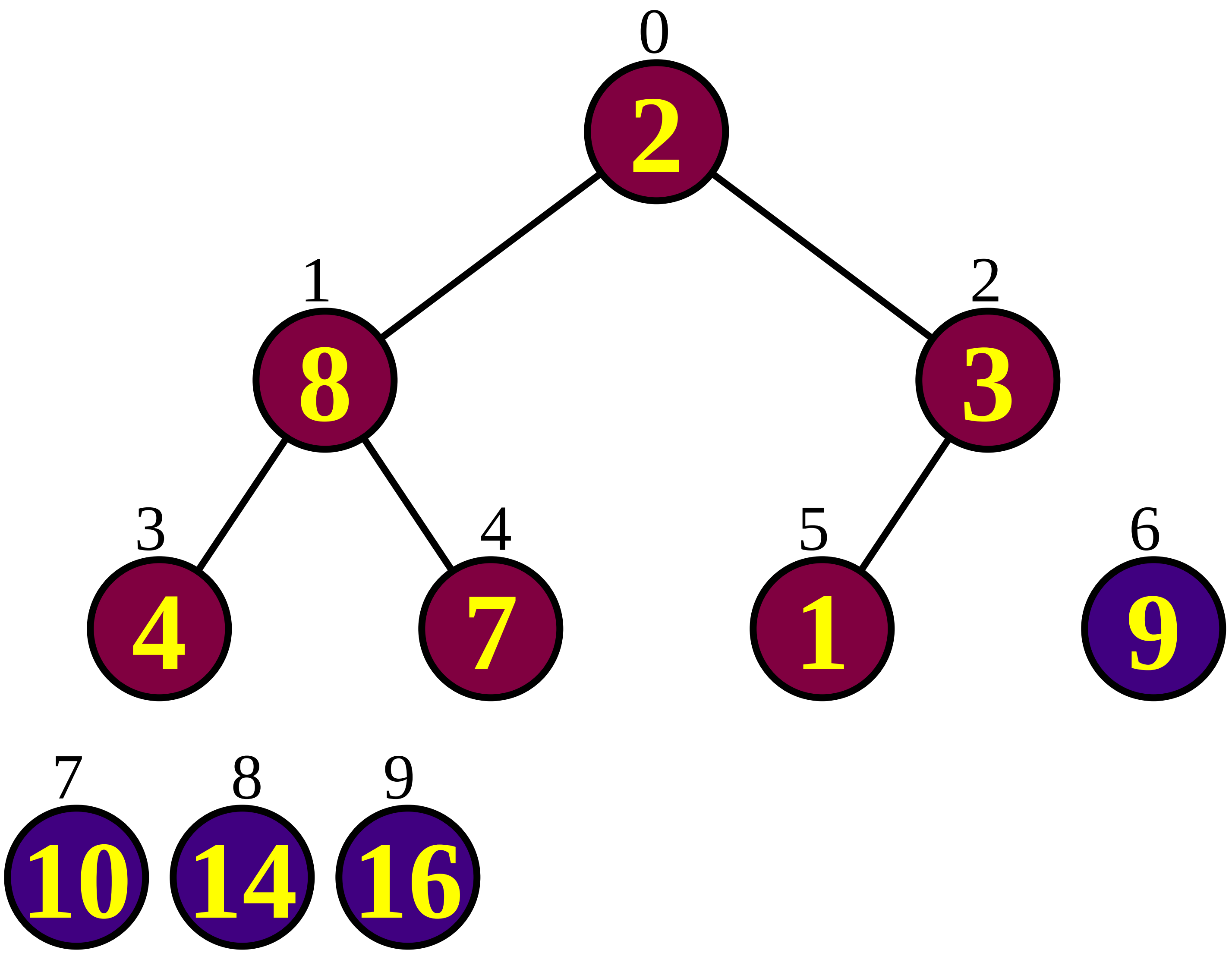
Heapsort



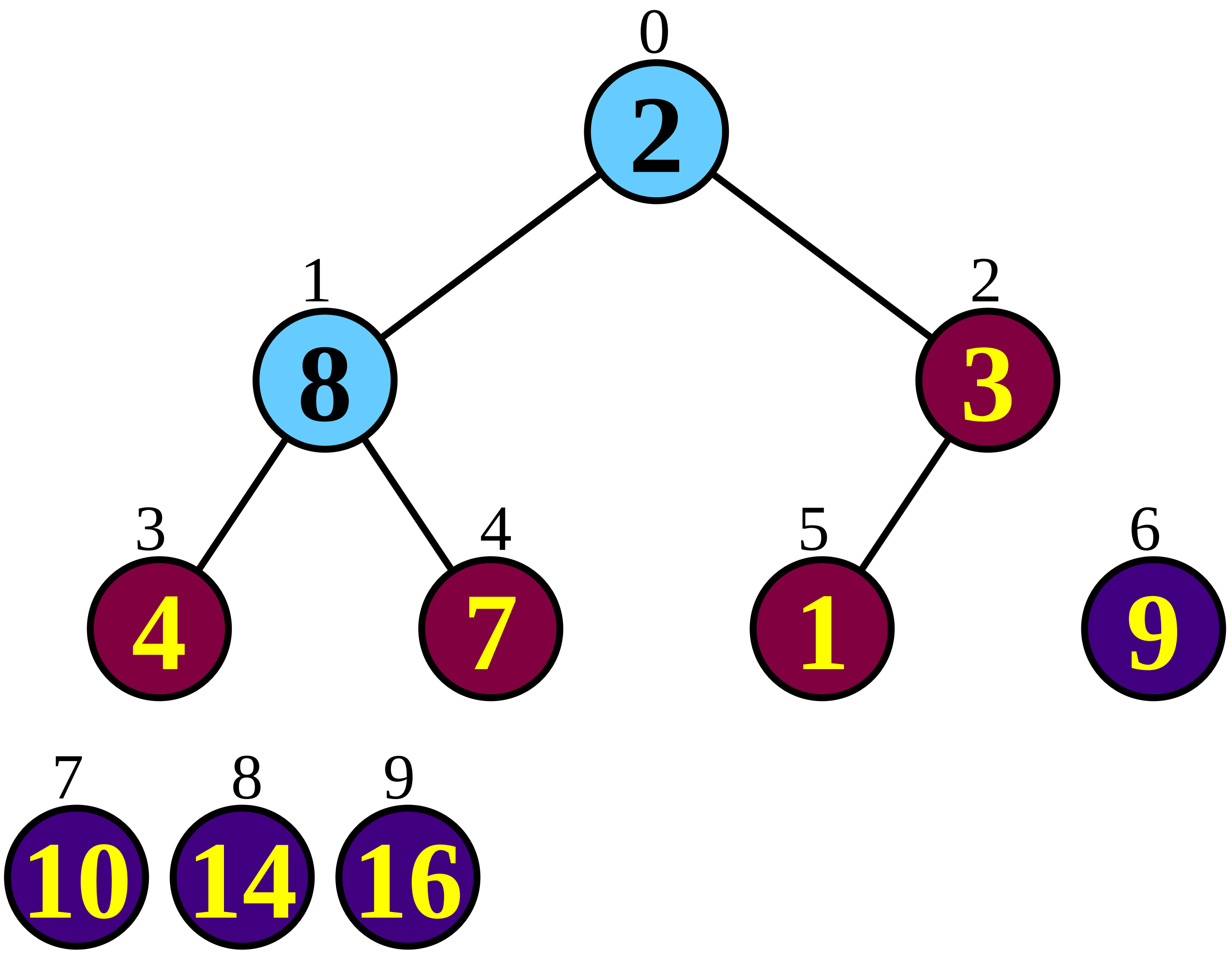
Heapsort



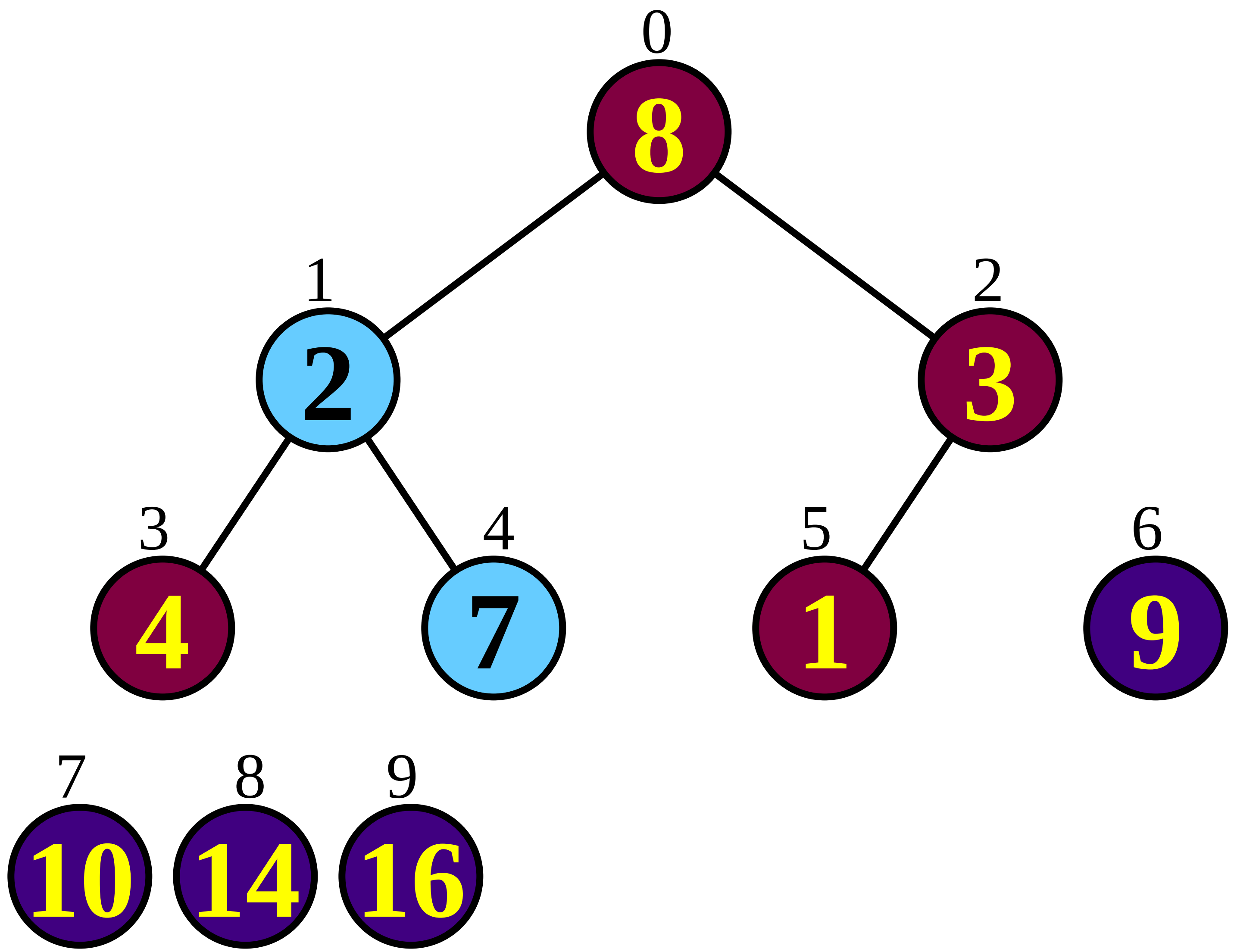
Heapsort



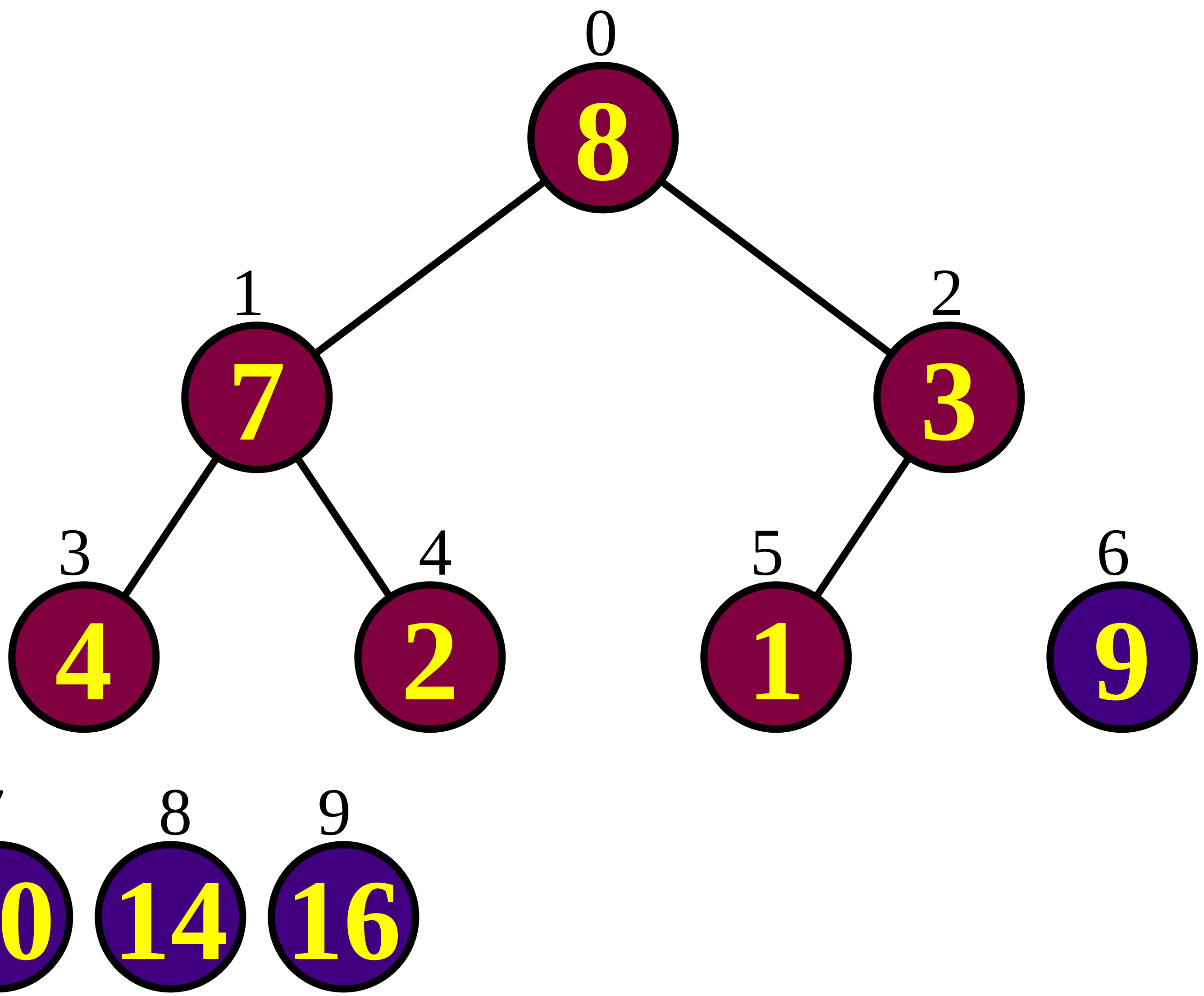
Heapsort



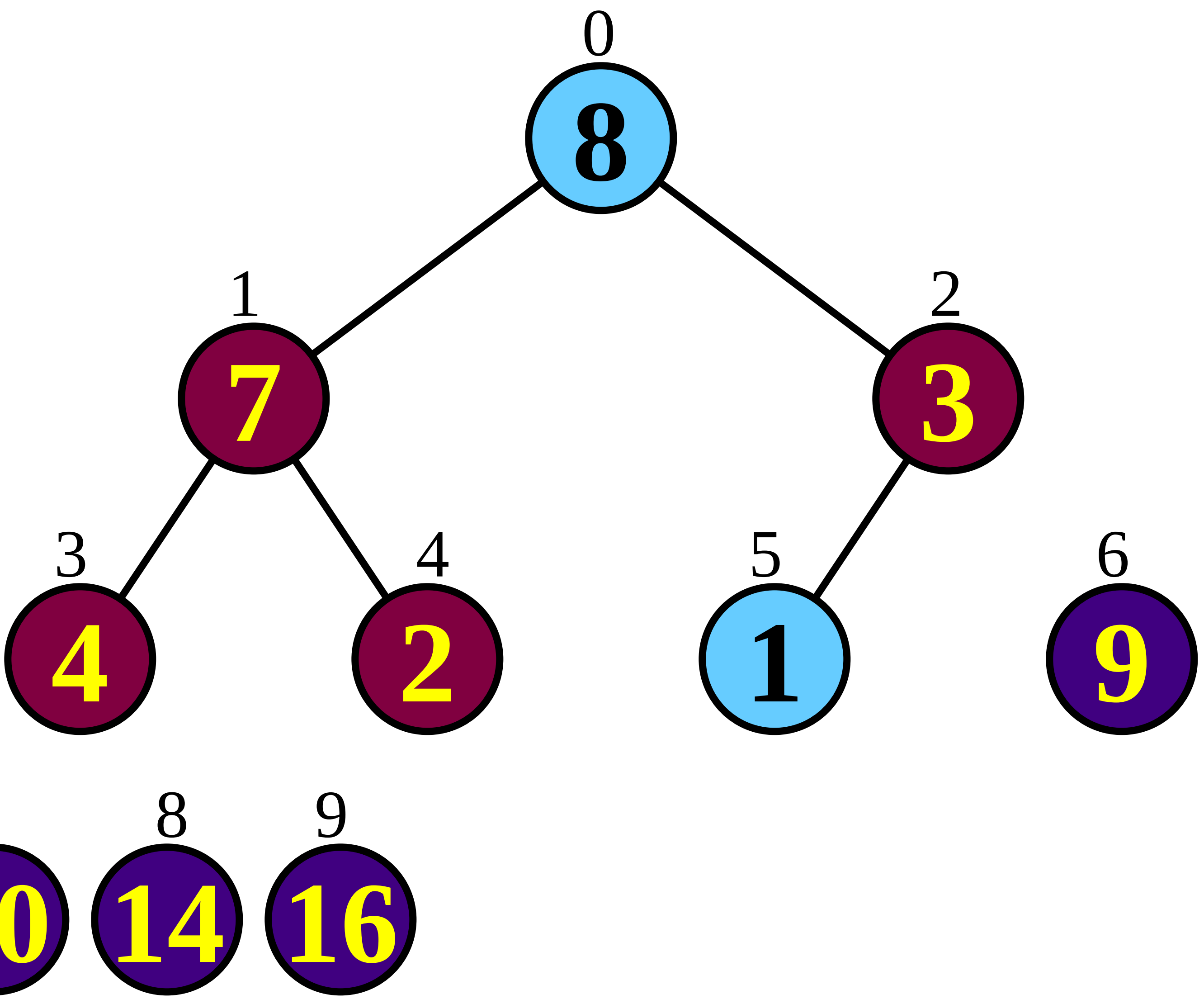
Heapsort



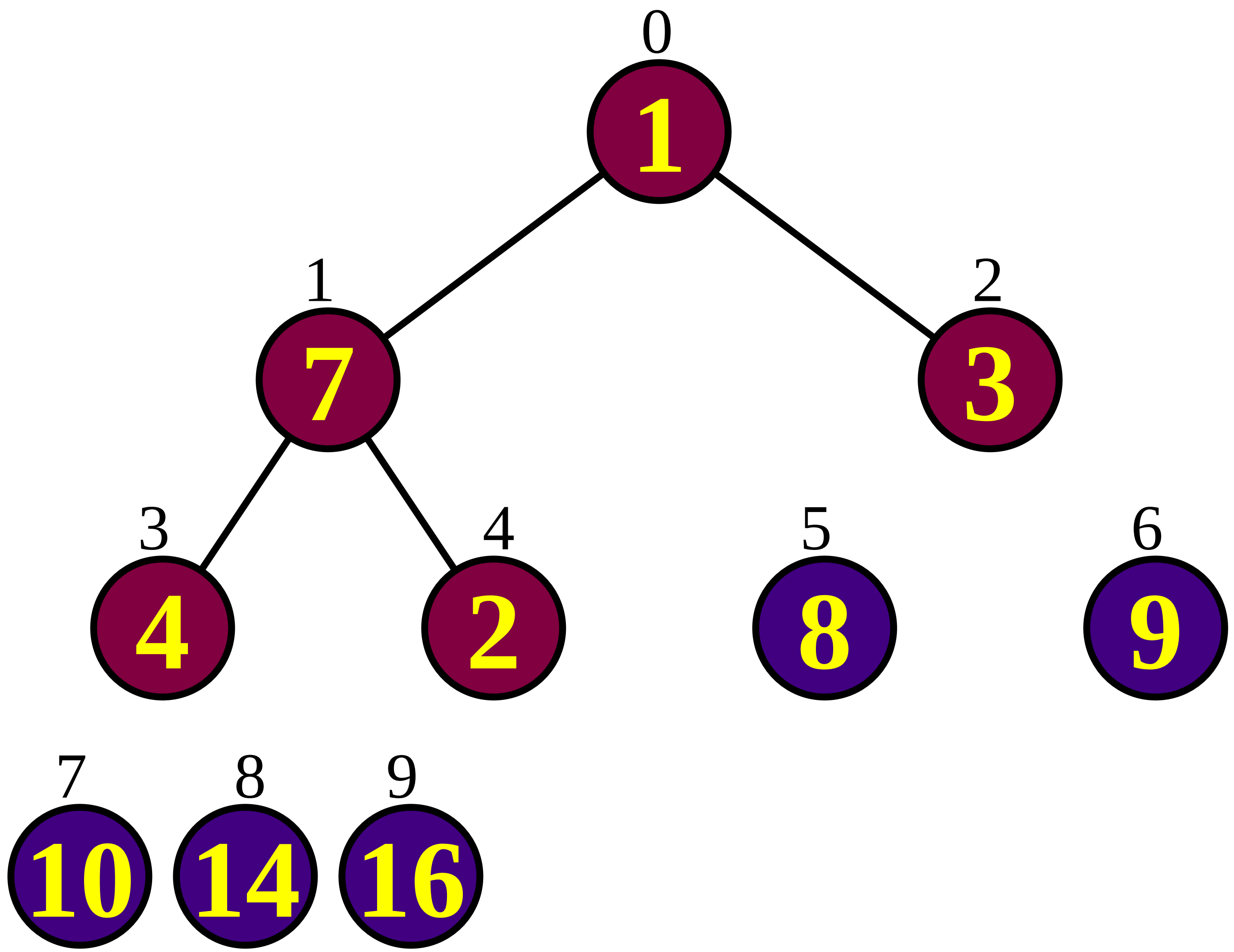
Heapsort



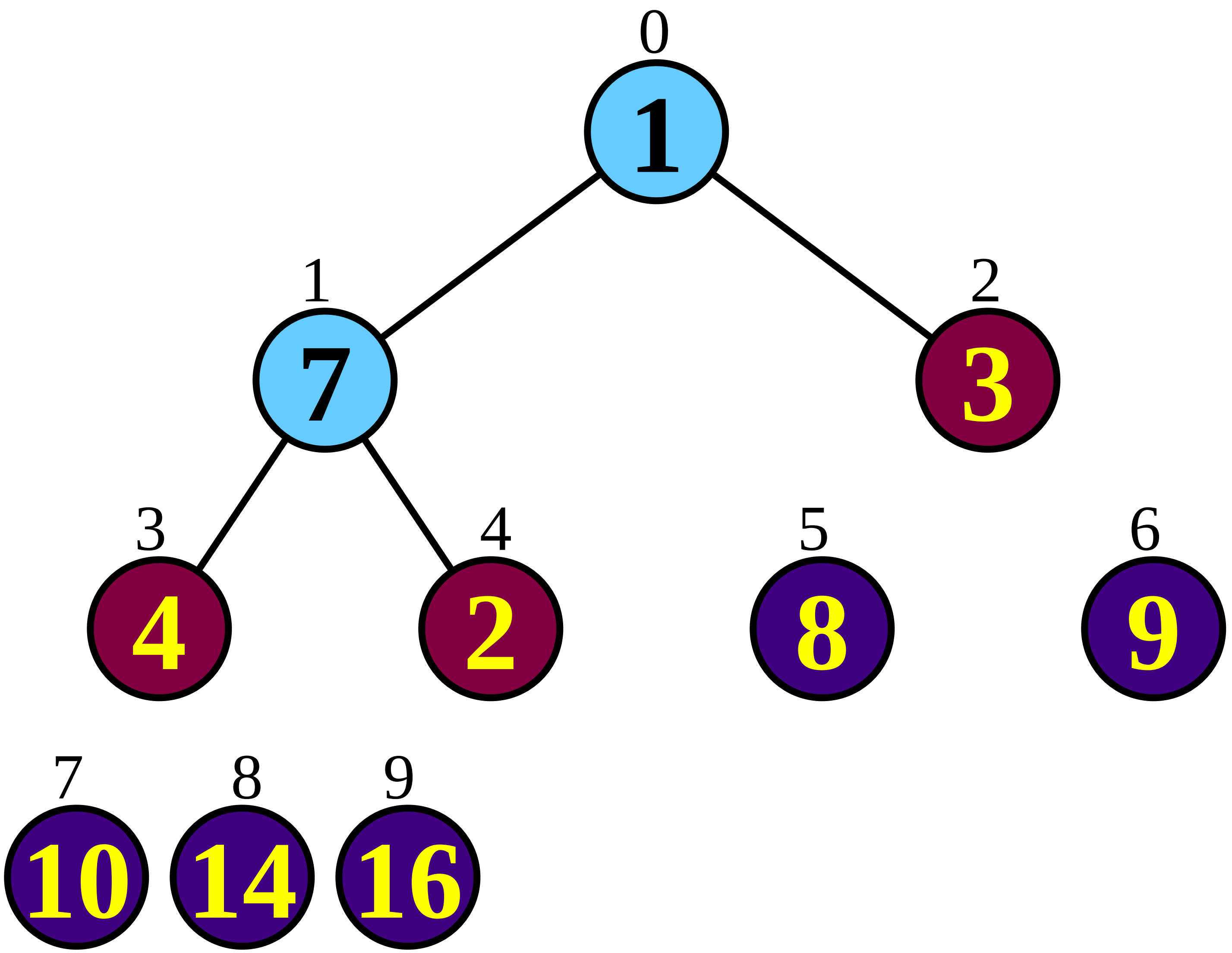
Heapsort



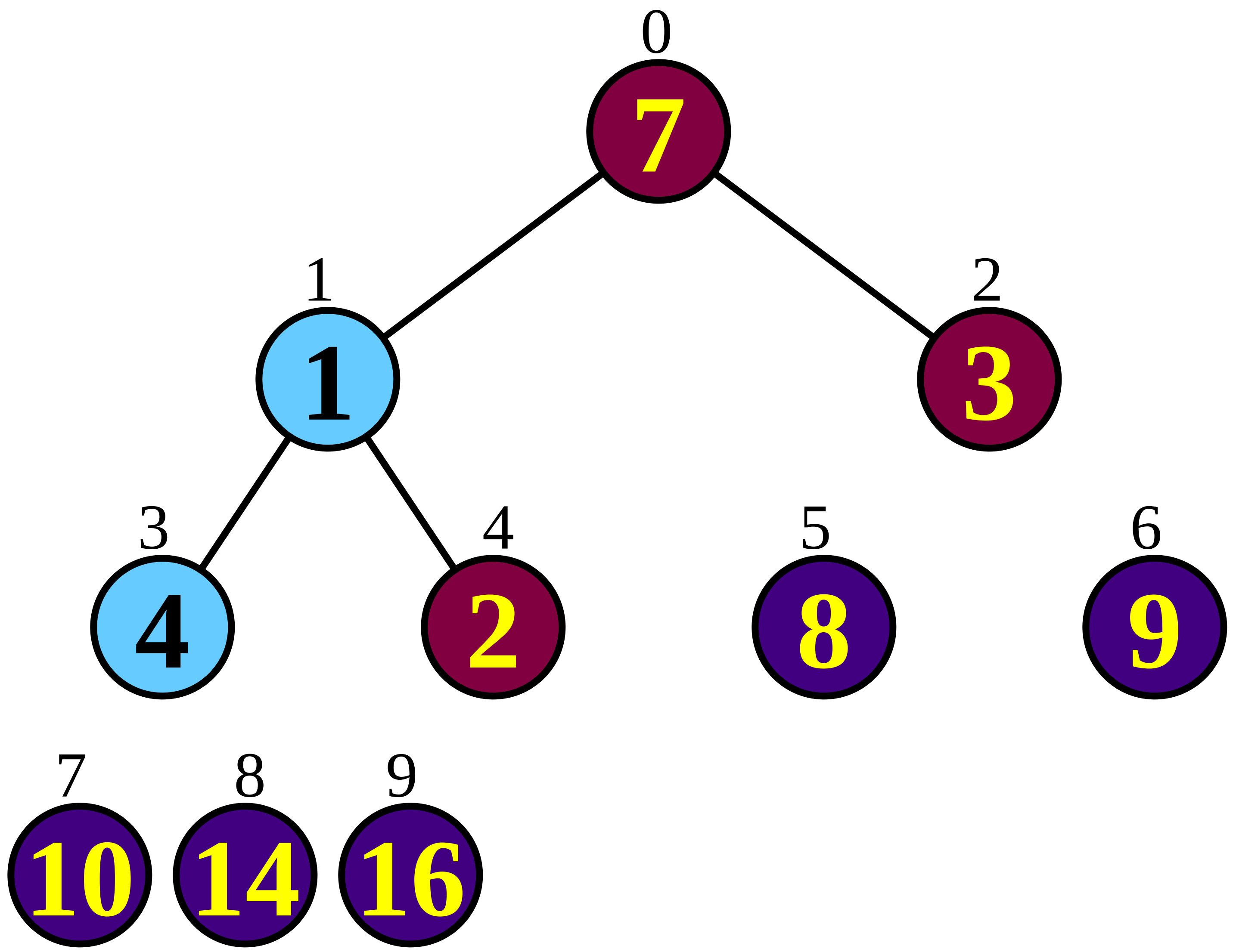
Heapsort



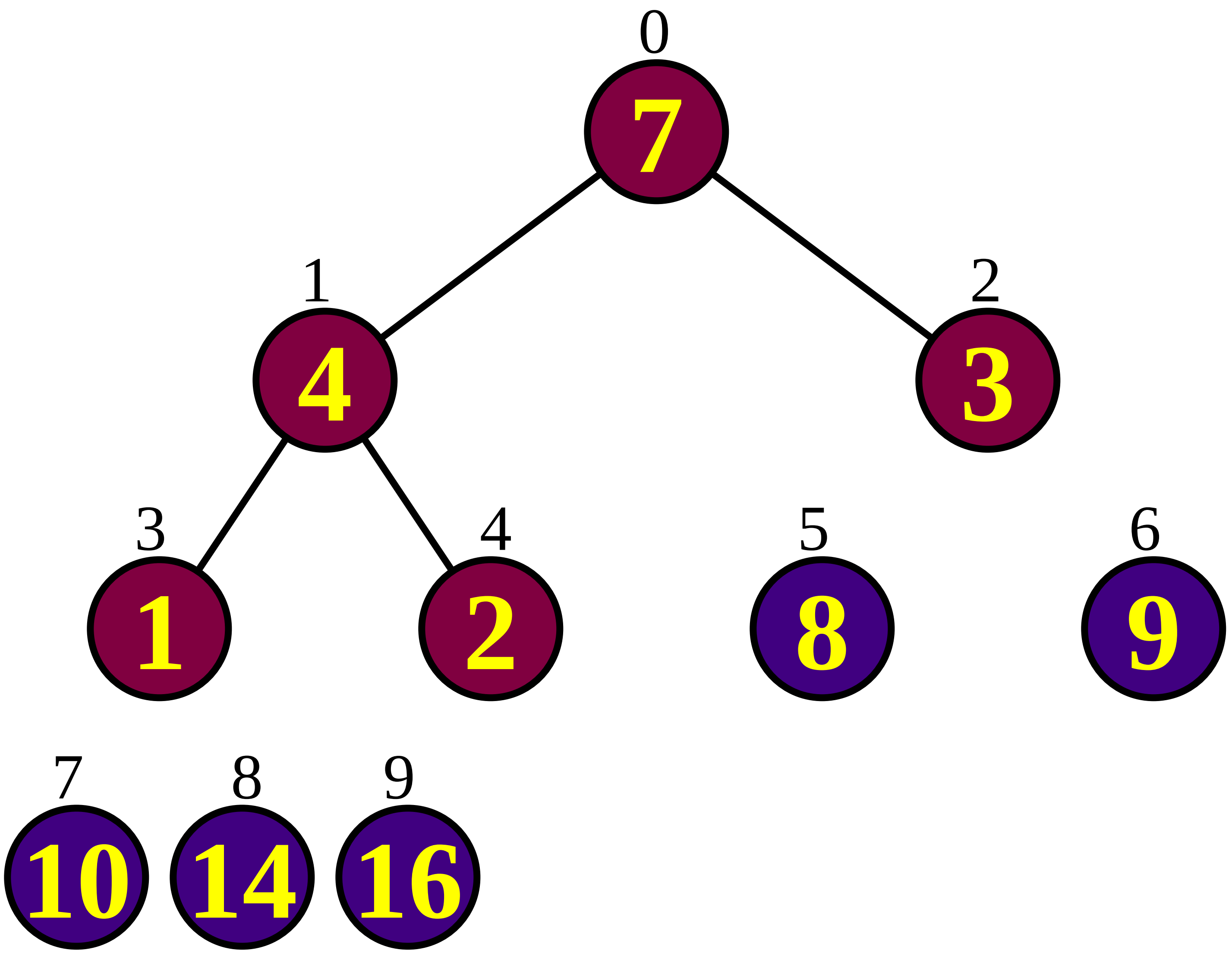
Heapsort



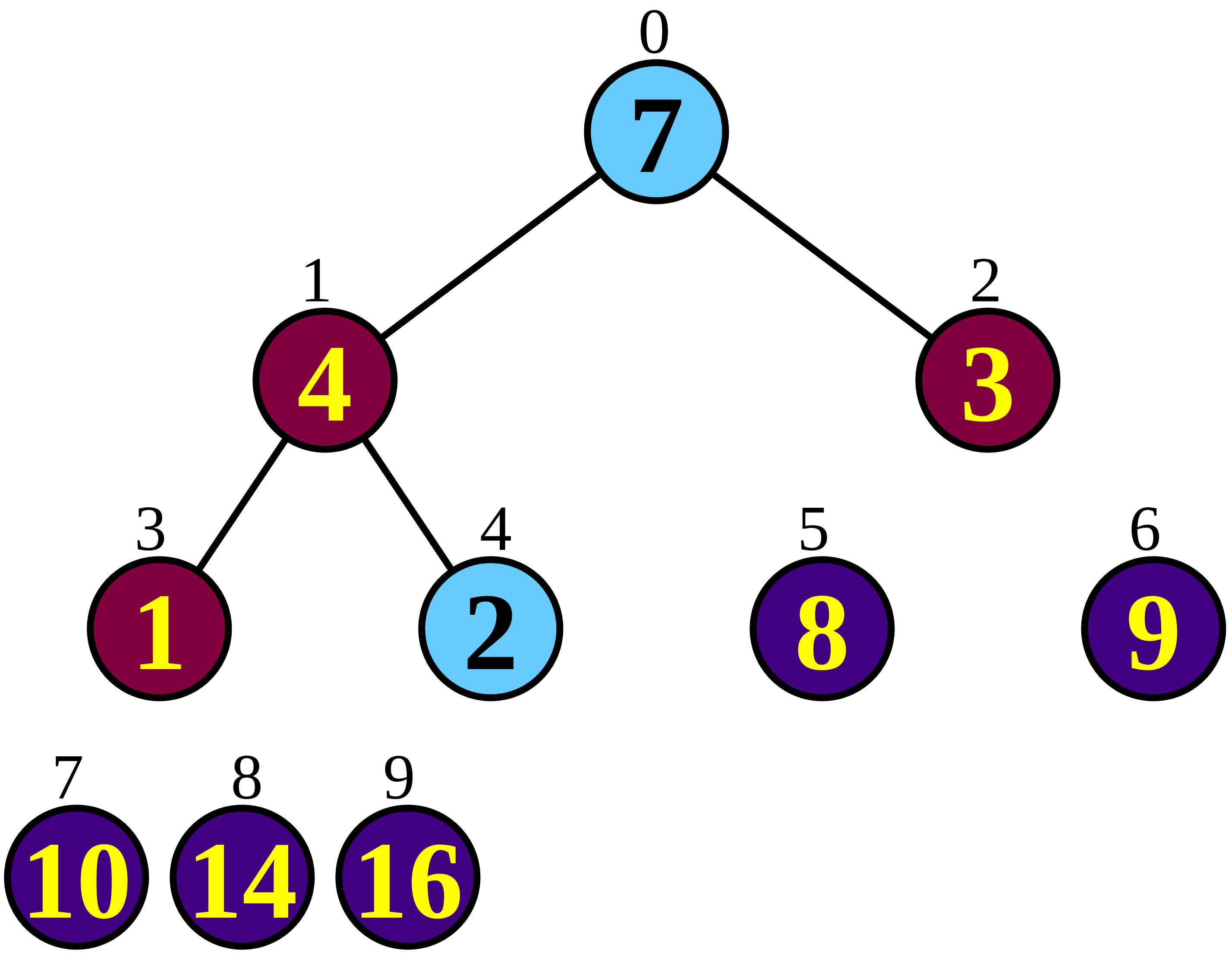
Heapsort



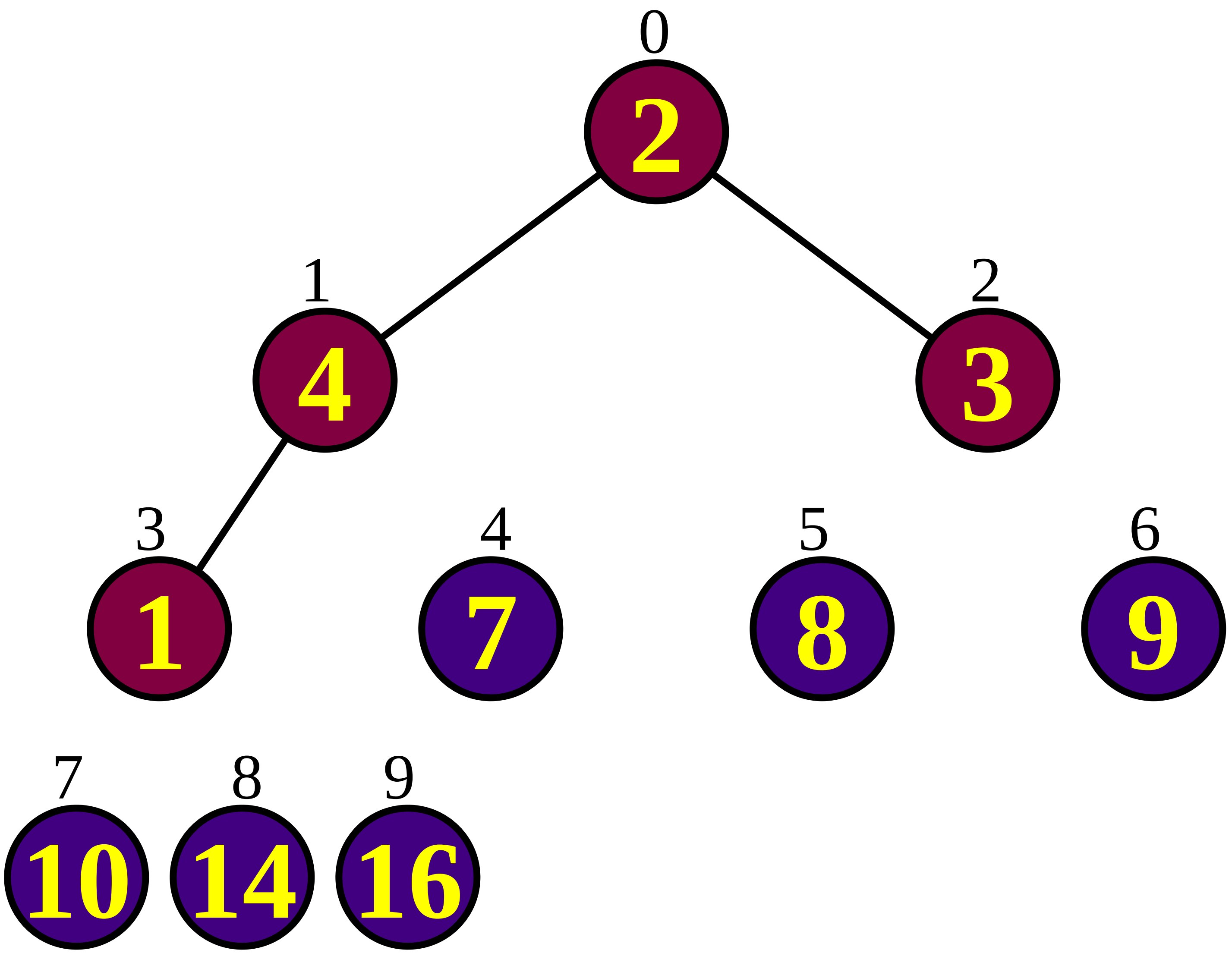
Heapsort



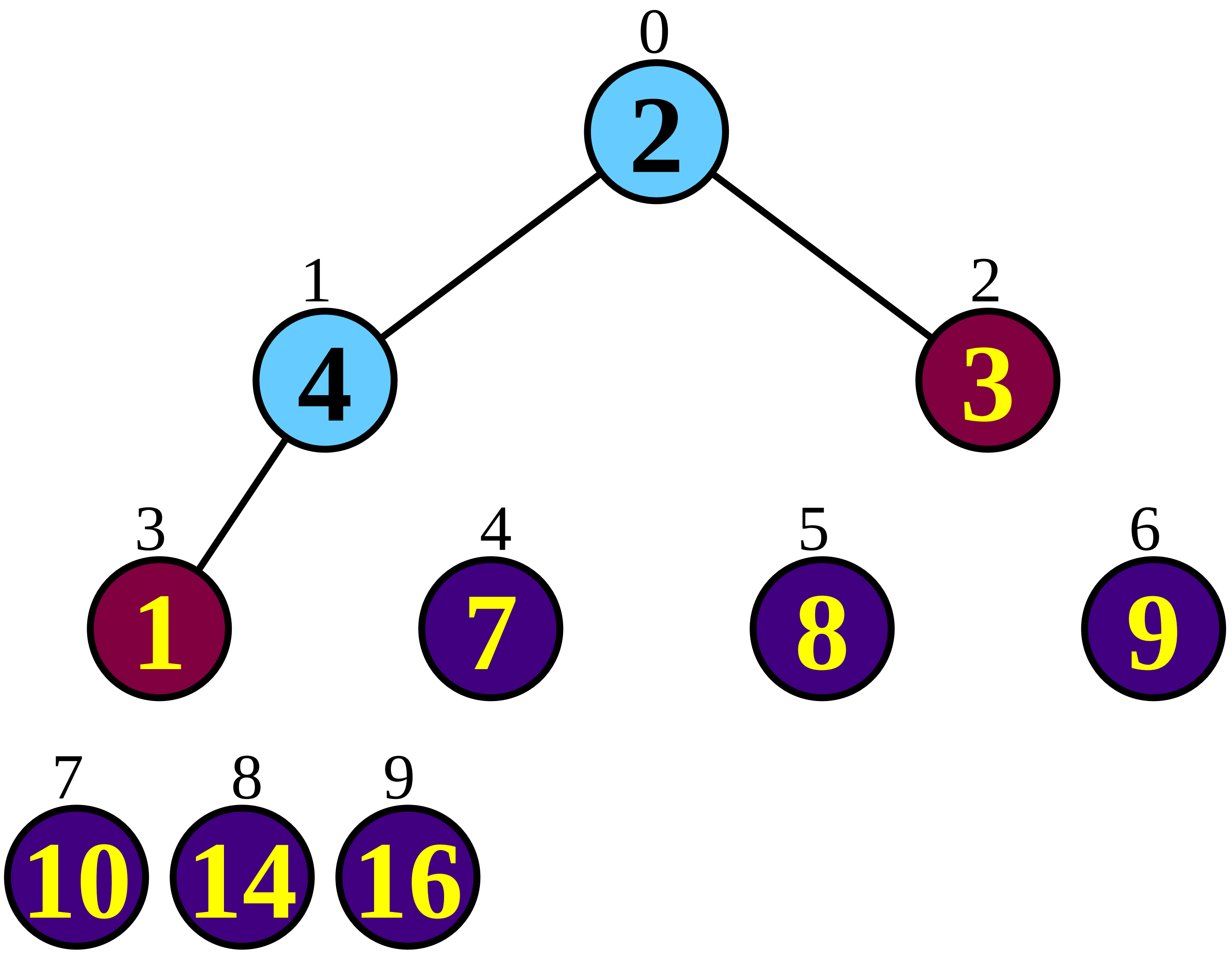
Heapsort



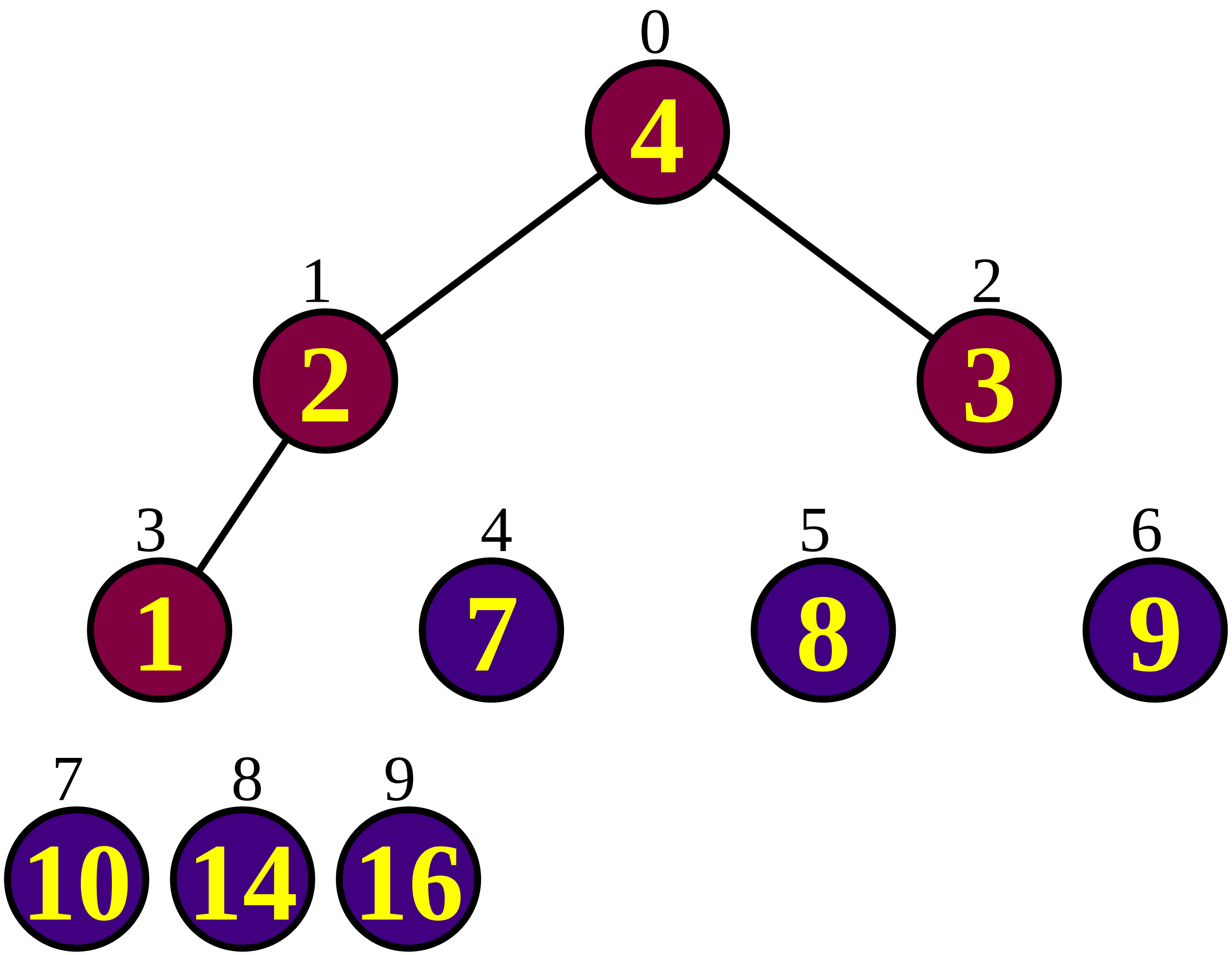
Heapsort



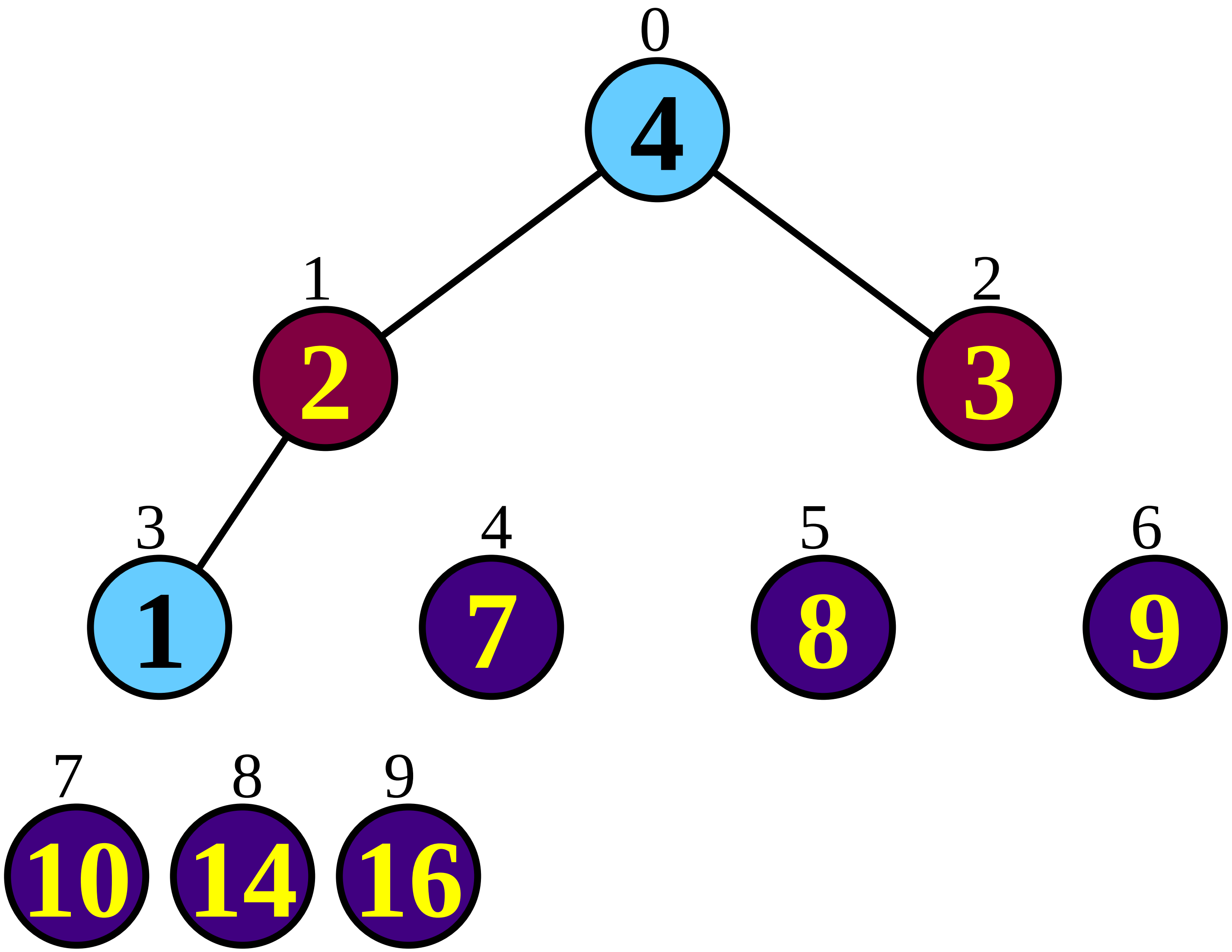
Heapsort



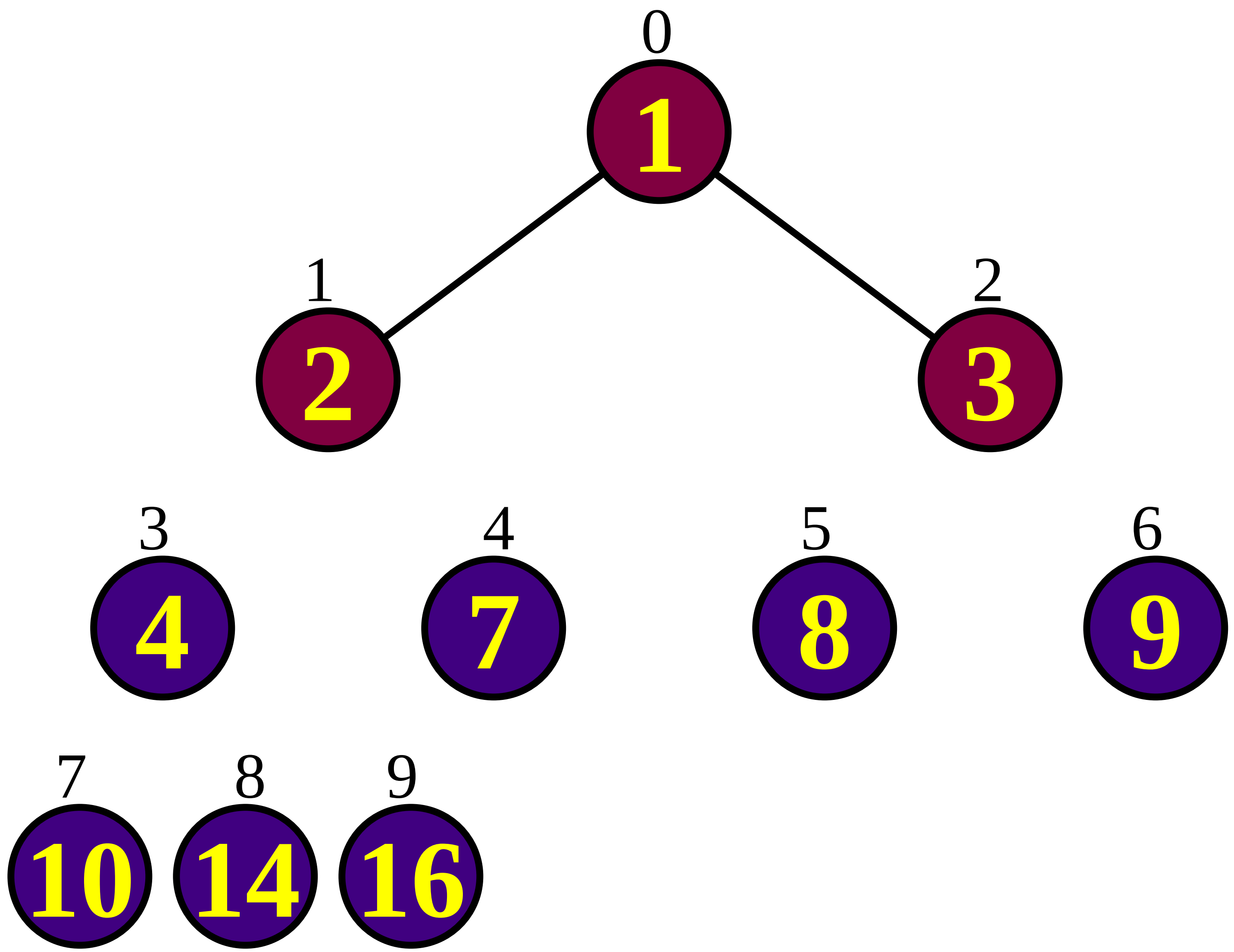
Heapsort



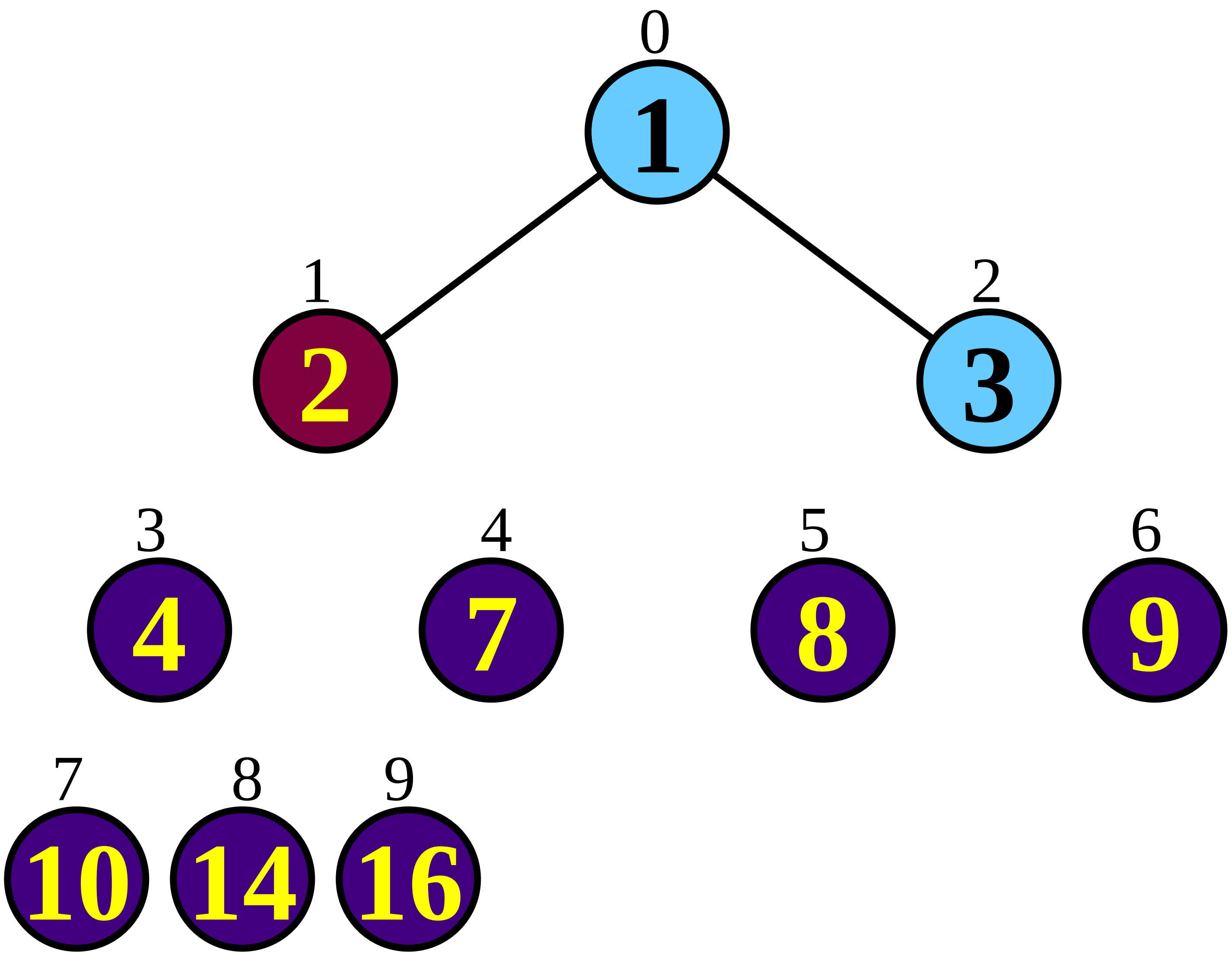
Heapsort



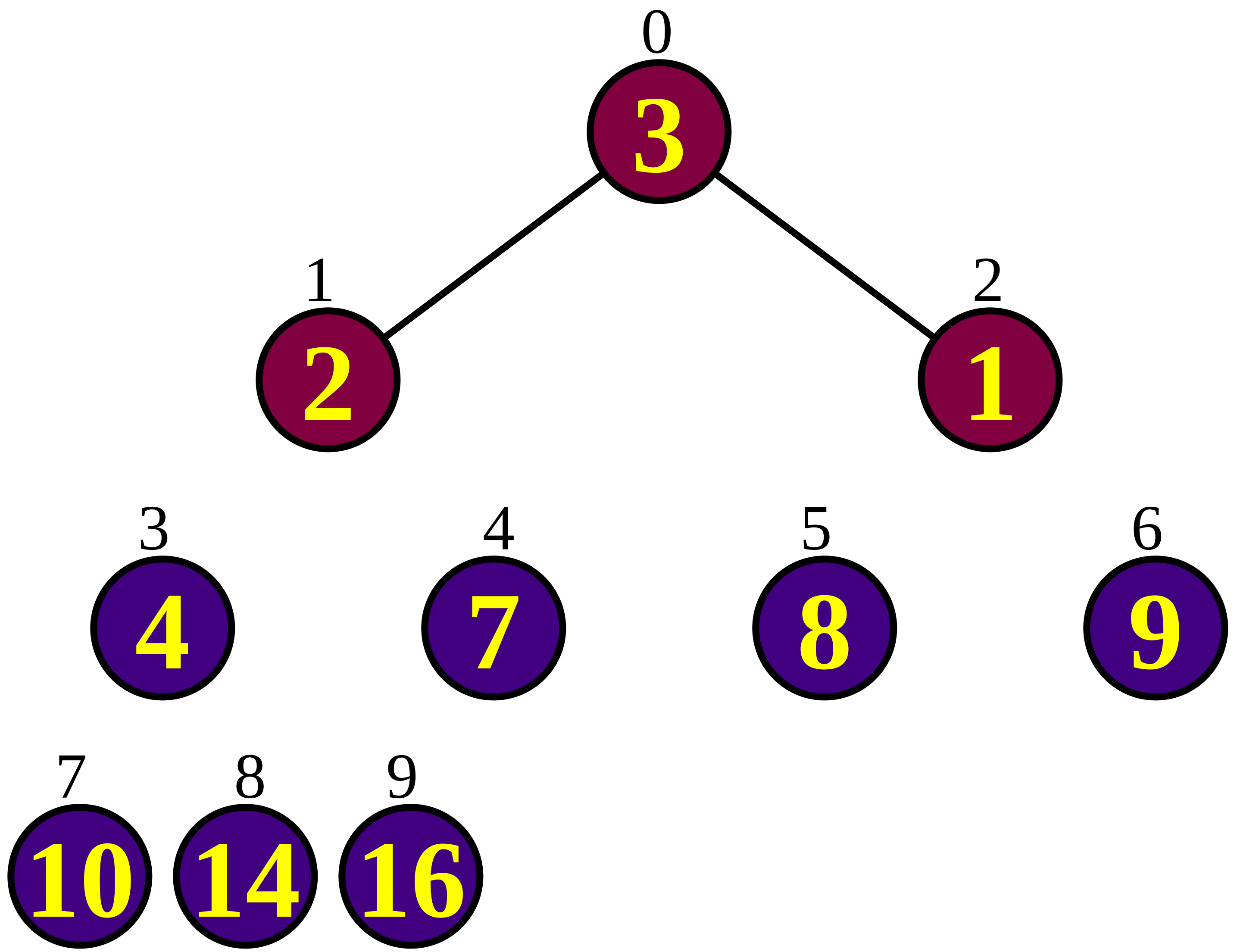
Heapsort



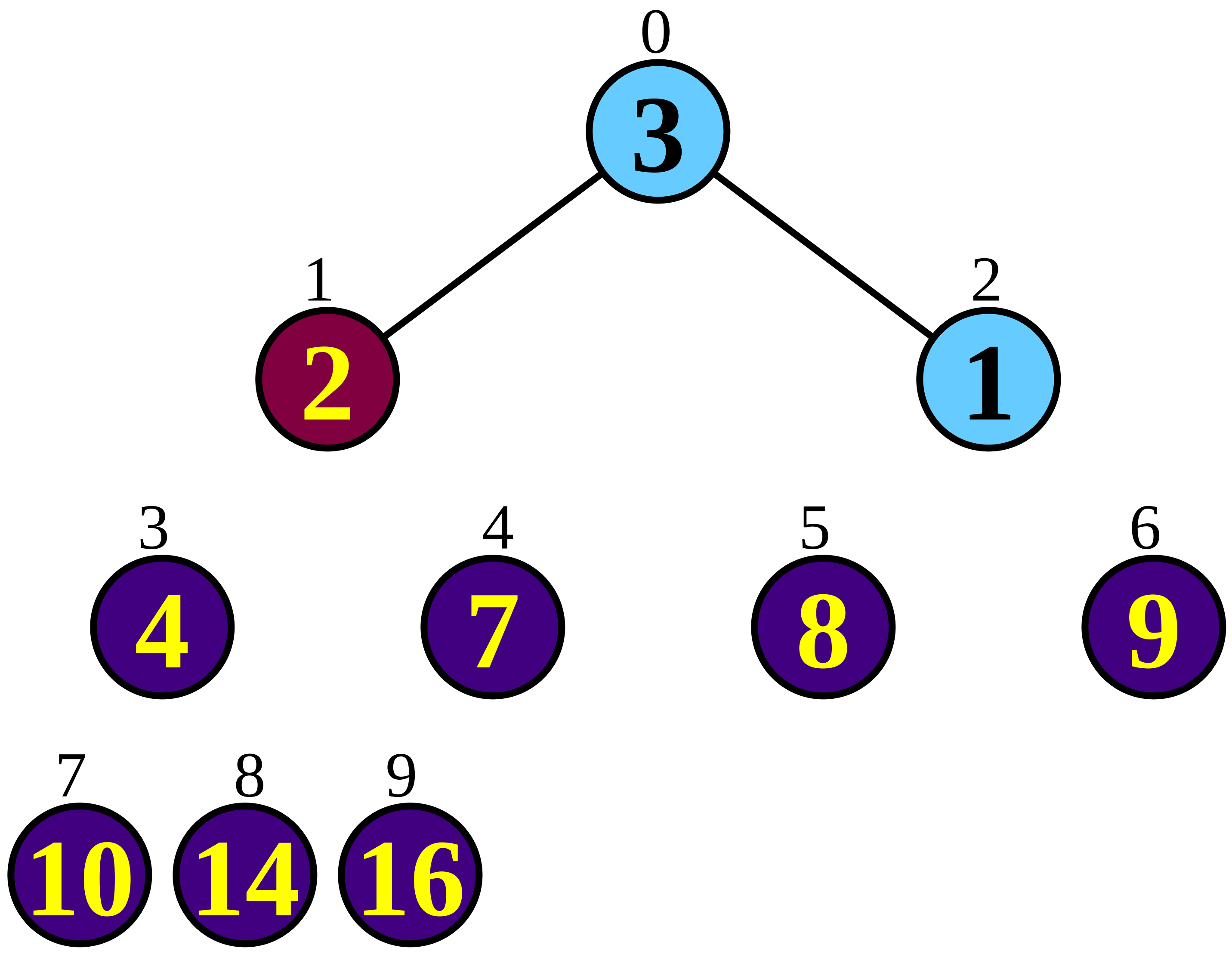
Heapsort



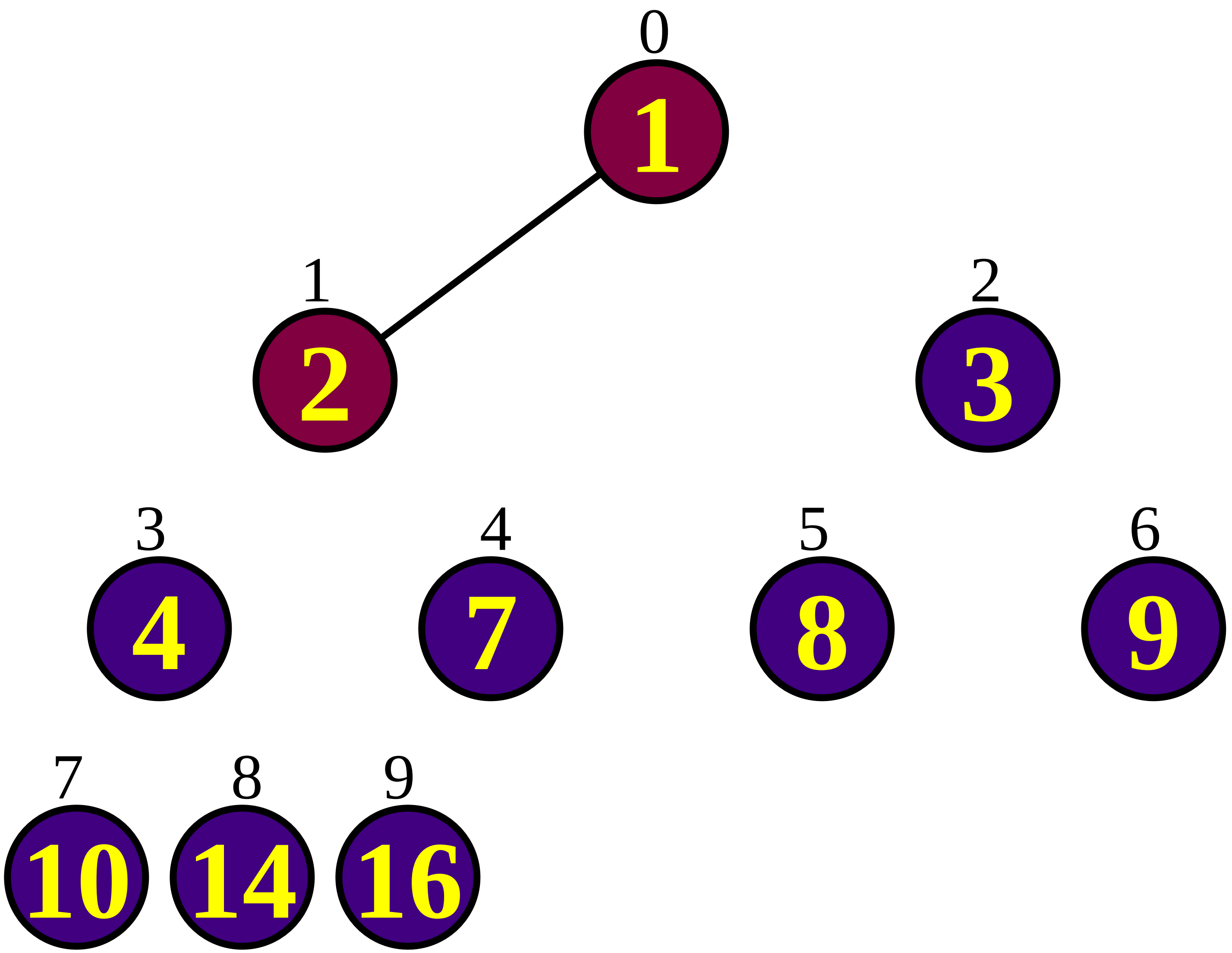
Heapsort



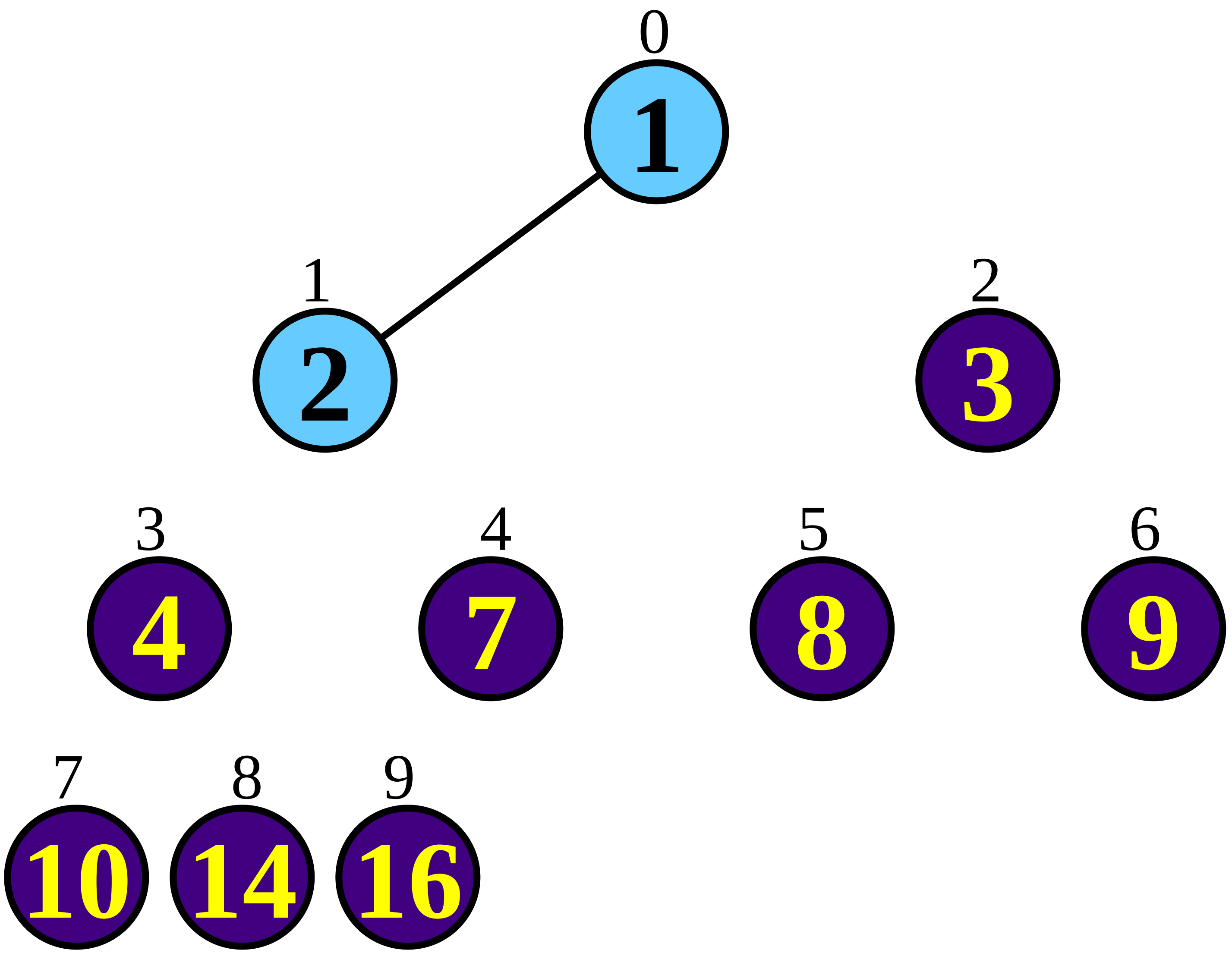
Heapsort



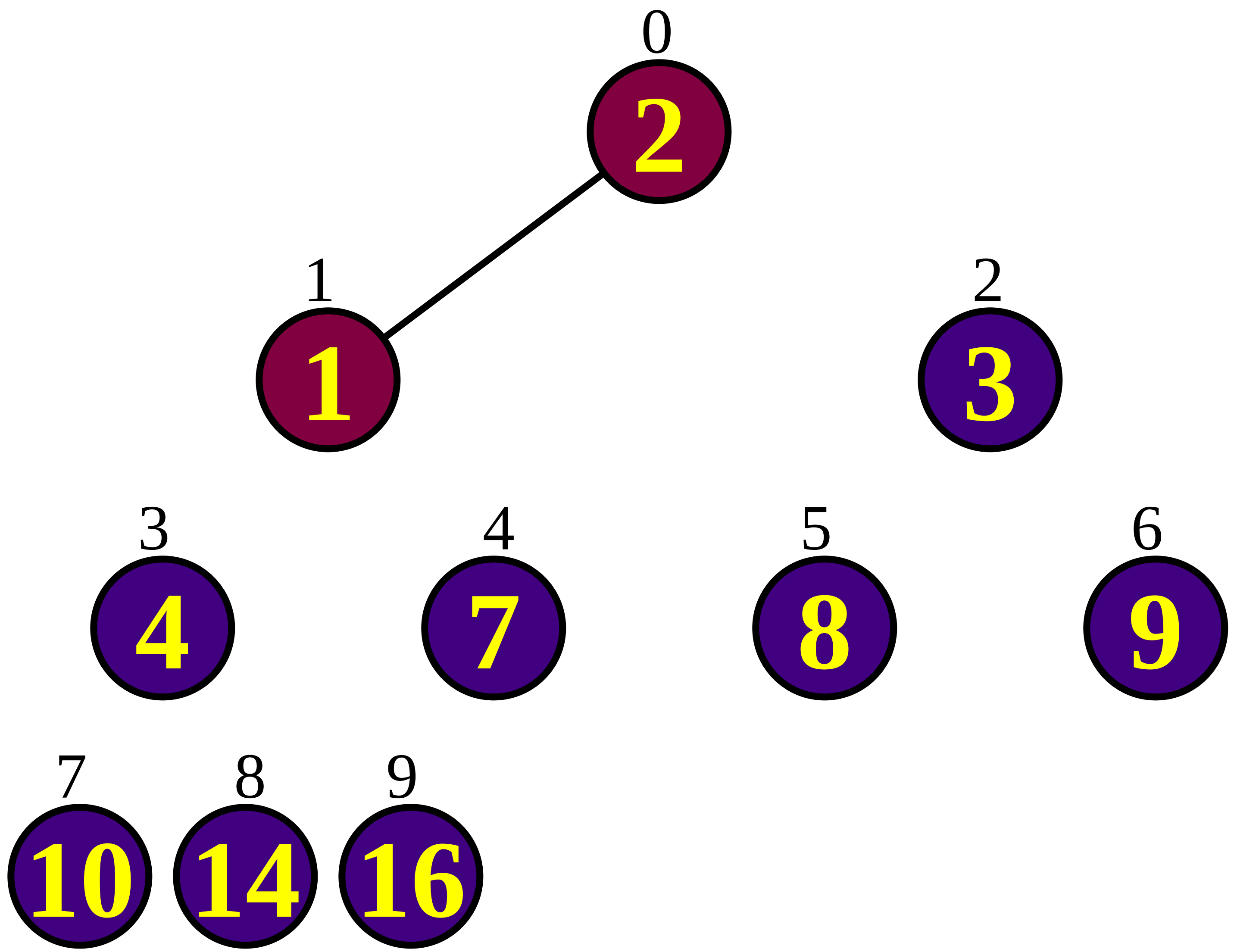
Heapsort



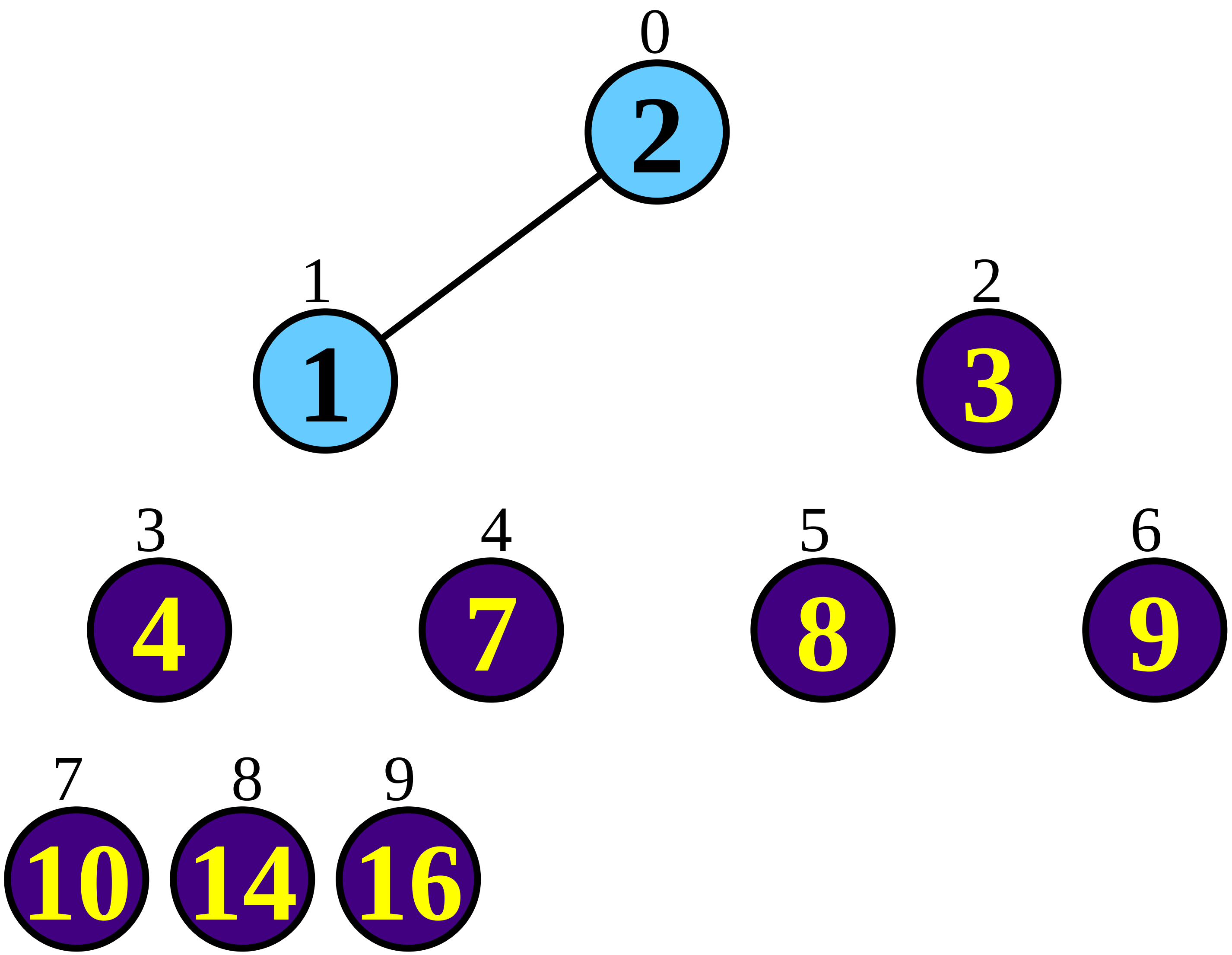
Heapsort



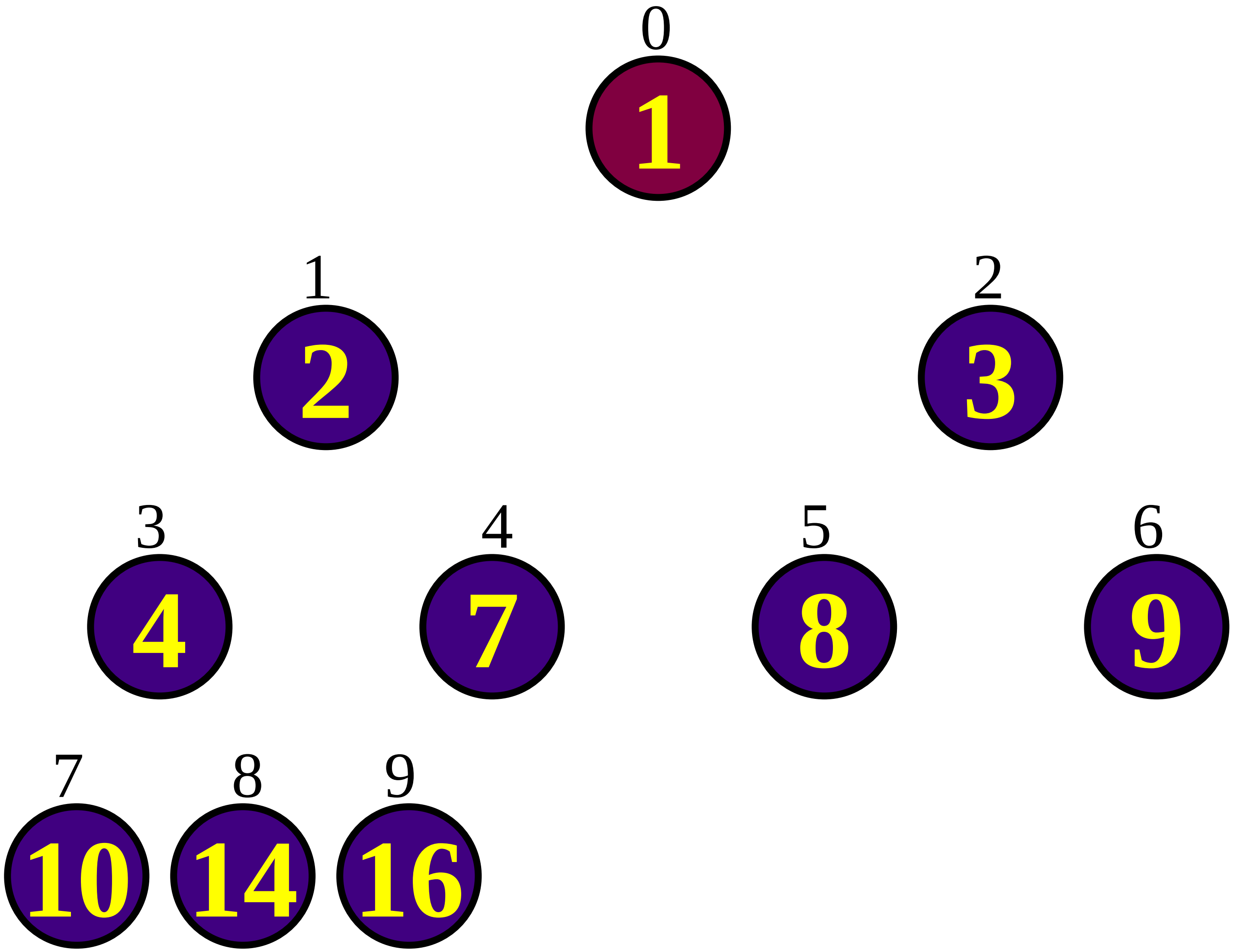
Heapsort



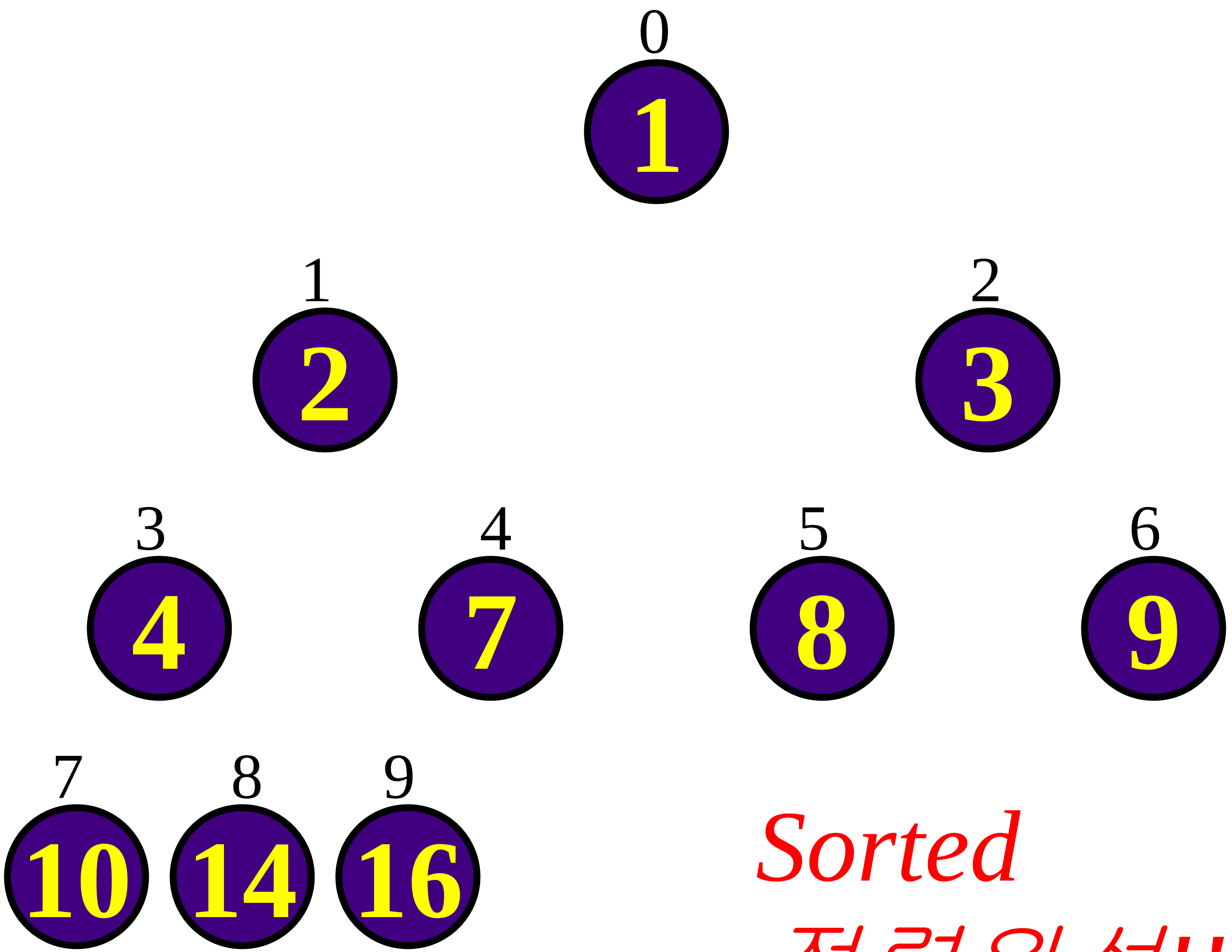
Heapsort



Heapsort



Heapsort



Sorted

정렬완성!!

히프 정렬 (2)

- 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 (3)

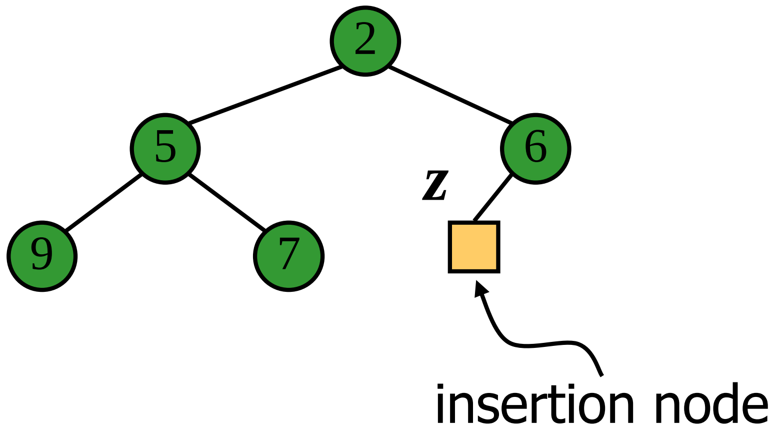
- Heapify 알고리즘

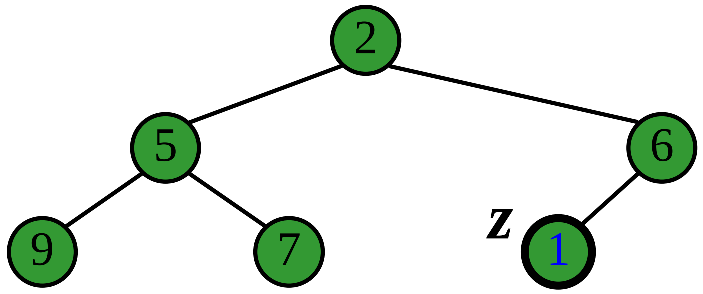
```
heapify(a[], h, m)
            // 루트 h를 제외한 h의 왼쪽 서브트리와 오른쪽 서브트리는 히프
            // 현재 시점으로 노드의 최대 레벨 순서 번호는 m
    for (j ← 2*h;  j ≤ m;  j ← 2*j) do  {
        if (j < m) then
            if (a[j] < a[j+1]) then j ← j+1;    // j는 값이 큰 왼쪽 또는 오른쪽 자식 노드
        if (a[h] ≥ a[j]) then exit
        else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

- 완전 2진 트리를 히프로 변환

Insertion into a Heap

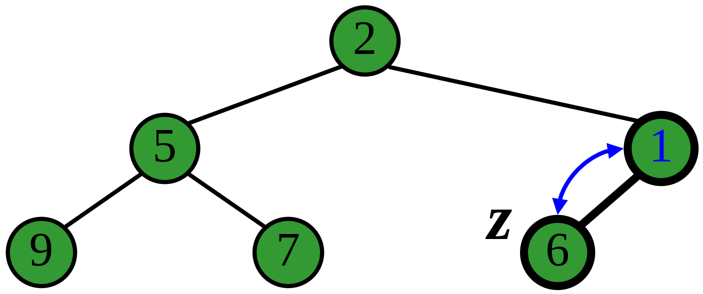
- Method insertItem of the priority queue ADT corresponds to the insertion of a key ***k*** to the heap.
- The insertion algorithm consists of three steps:
  - Find the insertion node ***z*** (the new last node)
  - Store ***k*** at ***z***
  - Restore the heap-order property (discussed next)

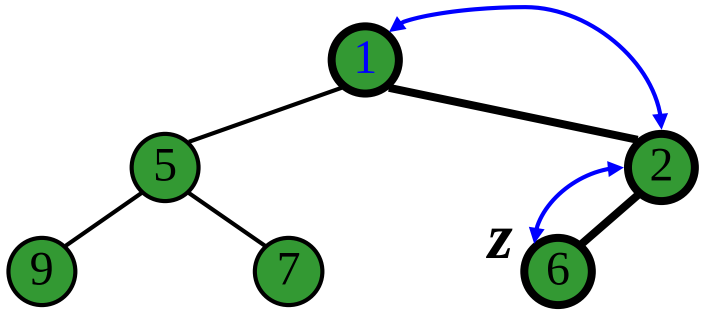




Upheap

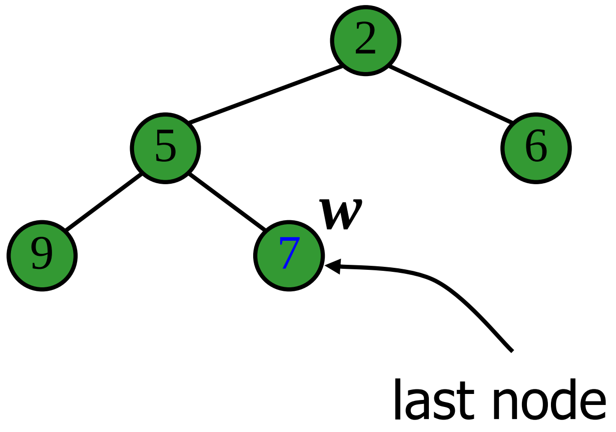
- After the insertion of a new key ***k***, the heap-order property may be violated.
- Algorithm upheap restores the heap-order property by swapping ***k*** along an upward path from the insertion node.
- Upheap terminates when the key ***k*** reaches the root or a node whose parent has a key smaller than or equal to ***k***.
- Since a heap has height ***O***(log ***n***), upheap runs in ***O***(log ***n***) time.

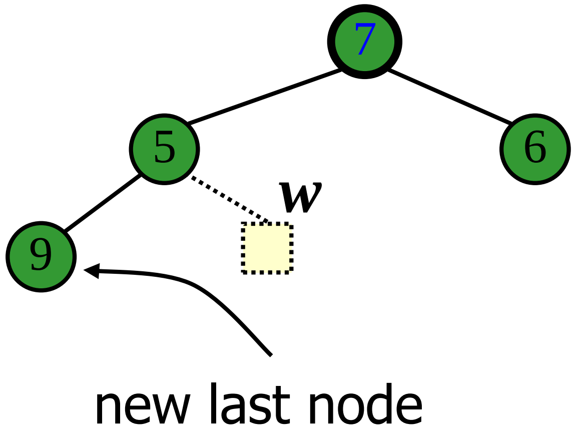




Removal from a Heap

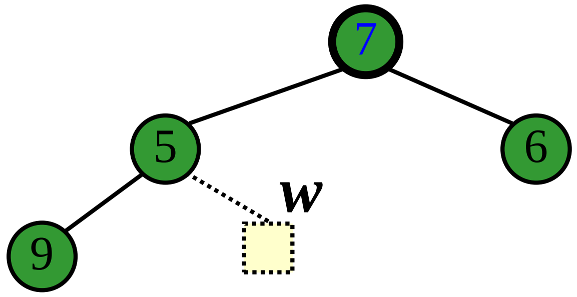
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap.
- The removal algorithm consists of three steps
  - ✓ Replace the root key with the key of the last node ***w***
  - ✓ Remove ***w***
  - ✓ Restore the heap-order property (discussed next)

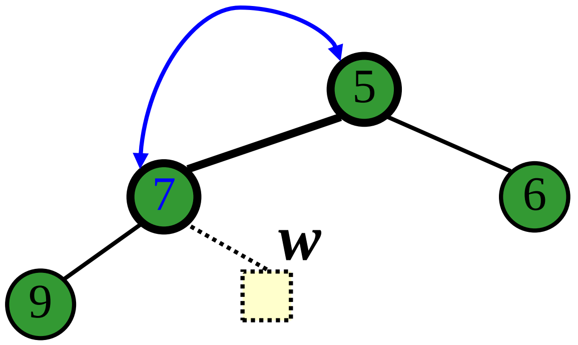




Downheap

- After replacing the root key with the key ***k*** of the last node, the heap-order property may be violated.
- Algorithm downheap restores the heap-order property by swapping key ***k*** along a downward path from the root.
- Downheap terminates when key ***k*** reaches a leaf or a node whose children have keys greater than or equal to ***k***.
- Since a heap has height ***O***(log ***n***), downheap runs in ***O***(log ***n***) time.





Updating the Last Node

- The insertion node can be found by traversing a path of ***O***(log ***n***) nodes:
  - ✓ Go up until a left child or the root is reached
  - ✓ If a left child is reached, go to the right child
  - ✓ Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal.