Algorithms and Analysis

Lesson 14: *Use Heaps!*

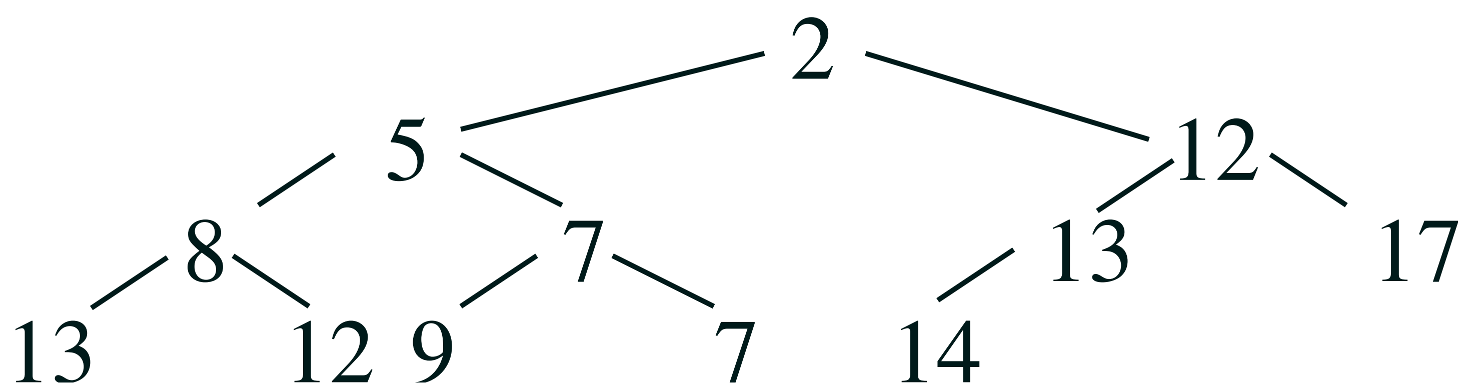
Heaps, Priority queues, Heap Sort, Other heaps

Outline

1. **Heaps**
2. Priority Queues
   - Array Implementation
3. Heap Sort
4. Other Heaps

Heaps

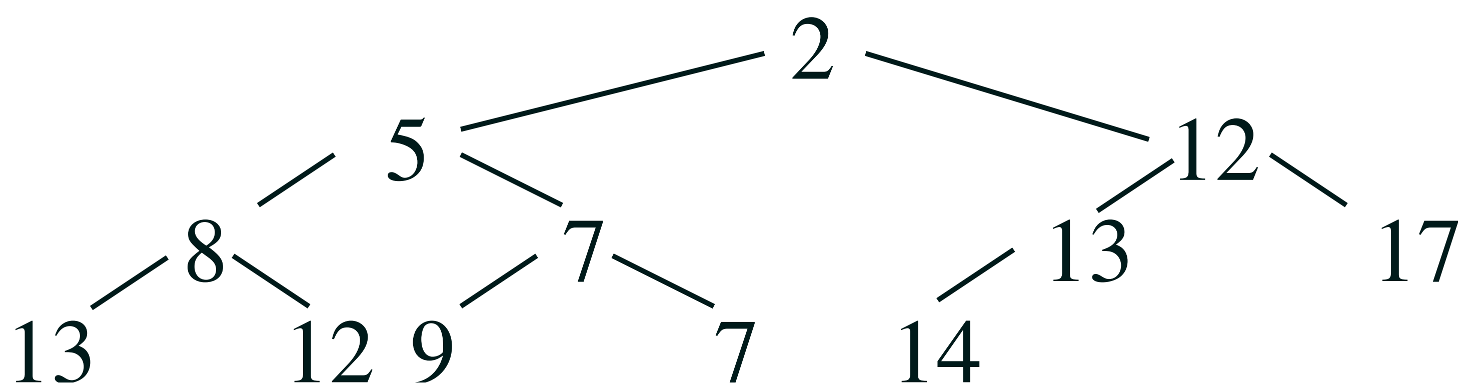
- A (min-)heap is (from one perspective) a binary tree
- It is a binary tree satisfying two constraints
  - ⋆ It is a **complete** tree
  - ⋆ Each child has a value ‘greater than or equal to’ its parent



- **complete** means that every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left

Heaps

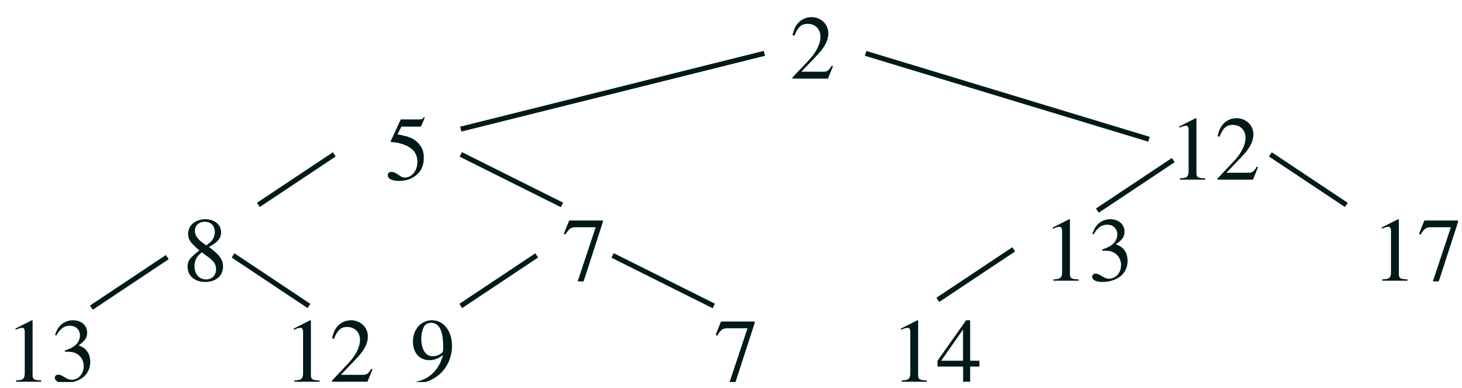
- A (min-)heap is (from one perspective) a binary tree
- It is a binary tree satisfying two constraints
  - ⋆ It is a **complete** tree
  - ⋆ Each child has a value ‘greater than or equal to’ its parent



- **complete** means that every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left

Heaps

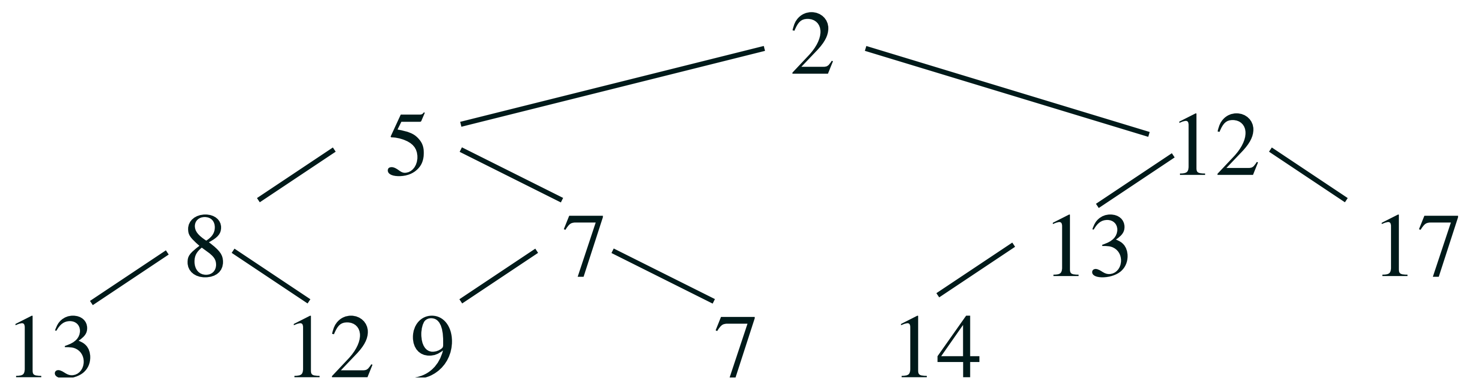
- A (min-)heap is (from one perspective) a binary tree
- It is a binary tree satisfying two constraints
  - ⋆ It is a **complete** tree
  - ⋆ Each child has a value ‘greater than or equal to’ its parent



- **complete** means that every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left

Heaps

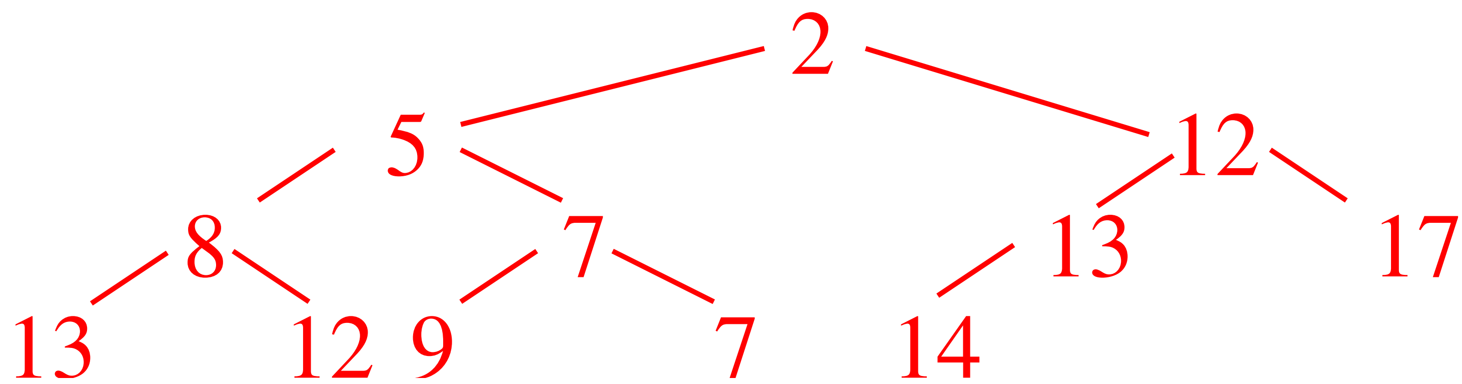
- A (min-)heap is (from one perspective) a binary tree
- It is a binary tree satisfying two constraints
  - ⋆ It is a **complete** tree
  - ⋆ Each child has a value 'greater than or equal to' its parent



- **complete** means that every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left

Heaps

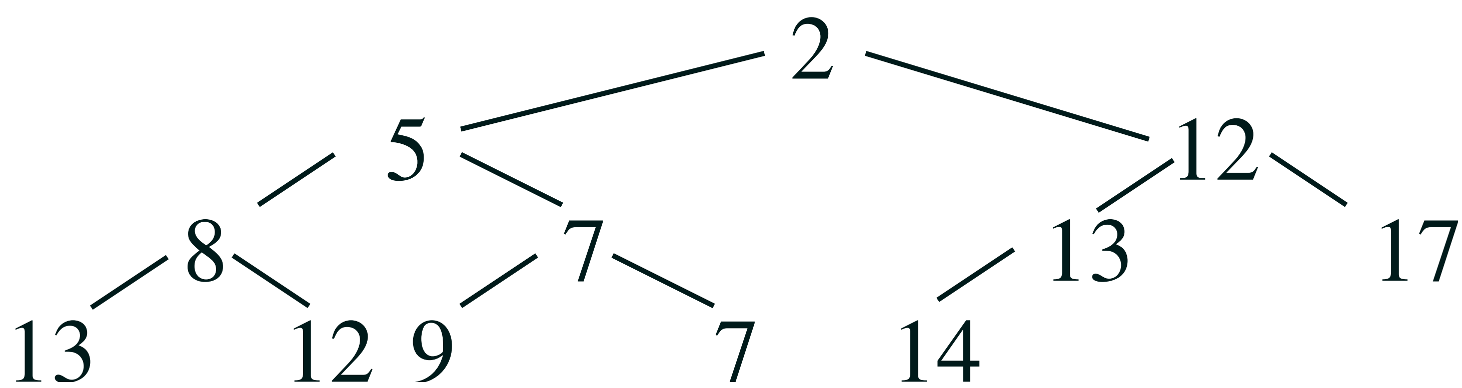
- A (min-)heap is (from one perspective) a binary tree
- It is a binary tree satisfying two constraints
  - ⋆ It is a **complete** tree
  - ⋆ Each child has a value 'greater than or equal to' its parent



- **complete** means that every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left

Heaps

- A (min-)heap is (from one perspective) a binary tree
- It is a binary tree satisfying two constraints
  - ⋆ It is a **complete** tree
  - ⋆ Each child has a value ‘greater than or equal to’ its parent



- **complete** means that every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left

Outline

1. Heaps
2. **Priority Queues**
   - Array Implementation
3. Heap Sort
4. Other Heaps

Priority Queues

- One of the prime uses of heaps is to implement a priority queue
- A Priority Queue is a queue with priorities
- That is, we assign a priority to each element we add
- The head of the queue is the element with highest priority (smallest number)
- Used, for example, in simulating real time events
- Used to implement "greedy algorithms"

Priority Queues

- One of the prime uses of heaps is to implement a priority queue
- A Priority Queue is a queue with priorities
- That is, we assign a priority to each element we add
- The head of the queue is the element with highest priority (smallest number)
- Used, for example, in simulating real time events
- Used to implement "greedy algorithms"

Priority Queues

- One of the prime uses of heaps is to implement a priority queue
- A Priority Queue is a queue with priorities
- That is, we assign a priority to each element we add
- The head of the queue is the element with highest priority (smallest number)
- Used, for example, in simulating real time events
- Used to implement "greedy algorithms"

Priority Queues

- One of the prime uses of heaps is to implement a priority queue
- A Priority Queue is a queue with priorities
- That is, we assign a priority to each element we add
- The head of the queue is the element with highest priority (smallest number)
- Used, for example, in simulating real time events
- Used to implement “greedy algorithms”

Priority Queues

- One of the prime uses of heaps is to implement a priority queue
- A Priority Queue is a queue with priorities
- That is, we assign a priority to each element we add
- The head of the queue is the element with highest priority (smallest number)
- Used, for example, in simulating real time events
- Used to implement "greedy algorithms"

Priority Queues

- One of the prime uses of heaps is to implement a priority queue
- A Priority Queue is a queue with priorities
- That is, we assign a priority to each element we add
- The head of the queue is the element with highest priority (smallest number)
- Used, for example, in simulating real time events
- Used to implement "greedy algorithms"

Priority Queue

- A simple Priority Queue might include
  - `unsigned size()` returning the the number of elements
  - `bool empty()` returns true if empty
  - `void push(T element, int priority)` adds an element
  - `T top()` returns head of queue
  - `void pop()` dequeues head of queue

Priority Queue

- A simple Priority Queue might include
  - ⋆ `unsigned size()` returning the the number of elements
  - ⋆ `bool empty()` returns true if empty
  - ⋆ `void push(T element, int priority)` adds an element
  - ⋆ `T top()` returns head of queue
  - ⋆ `void pop()` dequeues head of queue

Priority Queue

- A simple Priority Queue might include
  - ⋆ `unsigned size()` returning the the number of elements
  - ⋆ `bool empty()` returns true if empty
  - ⋆ `void push(T element, int priority)` adds an element
  - ⋆ `T top()` returns head of queue
  - ⋆ `void pop()` dequeues head of queue

Priority Queue

- A simple Priority Queue might include
  - `unsigned size()` returning the the number of elements
  - `bool empty()` returns true if empty
  - `void push(T element, int priority)` adds an element
  - `T top()` returns head of queue
  - `void pop()` dequeues head of queue

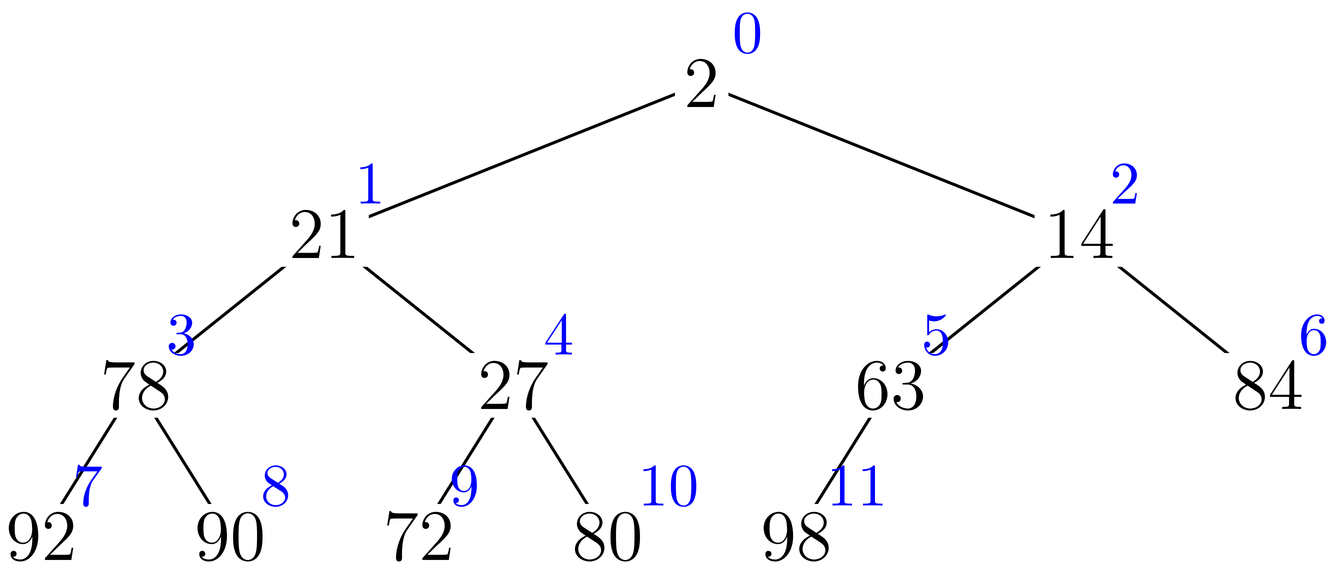
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array

Array Implementation of Heaps

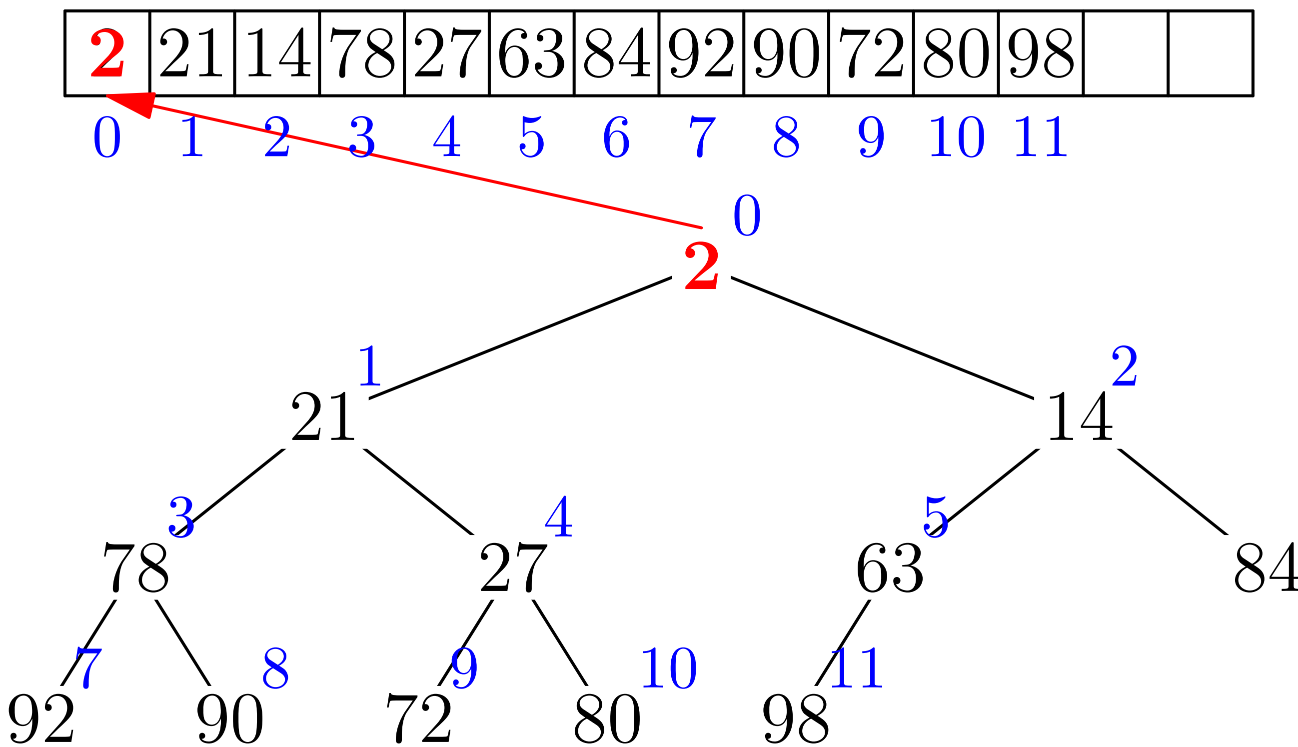
- Because the tree is complete we can implemented the heap efficiently using an array

2	21	14	78	27	63	84	92	90	72	80	98		
0	1	2	3	4	5	6	7	8	9	10	11		



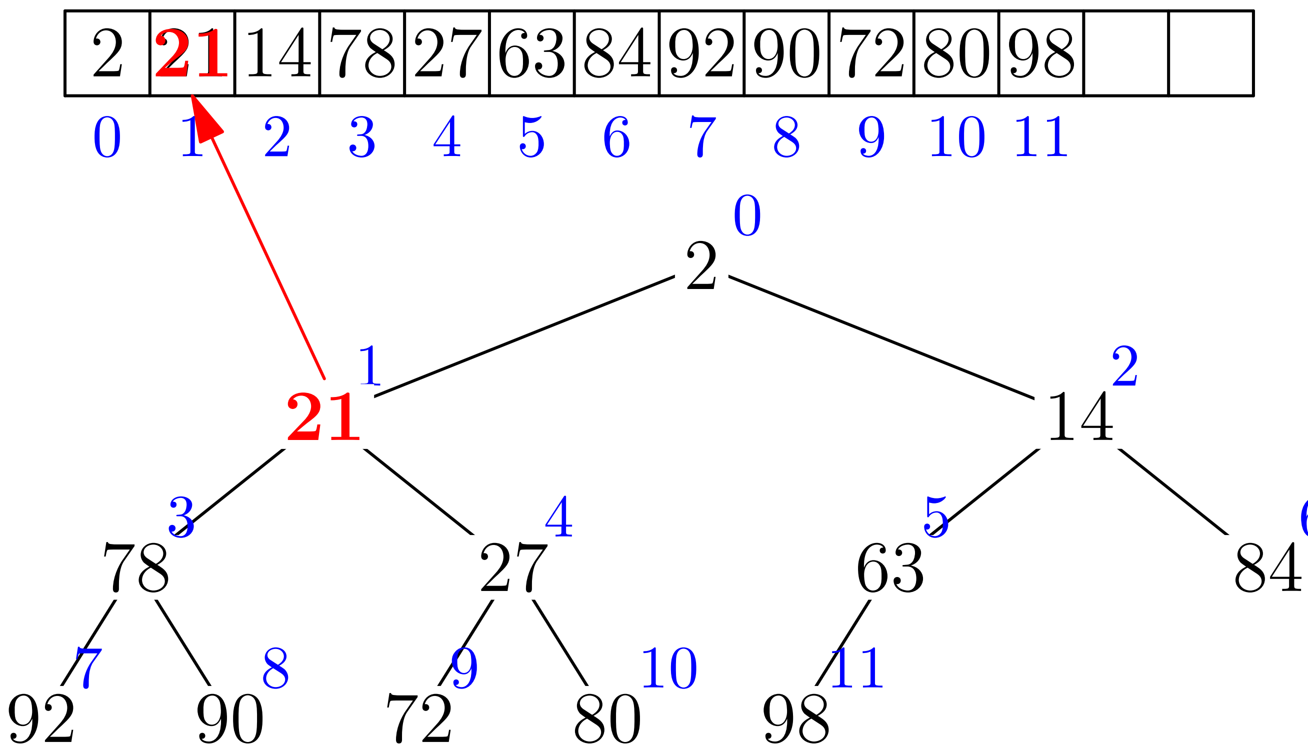
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



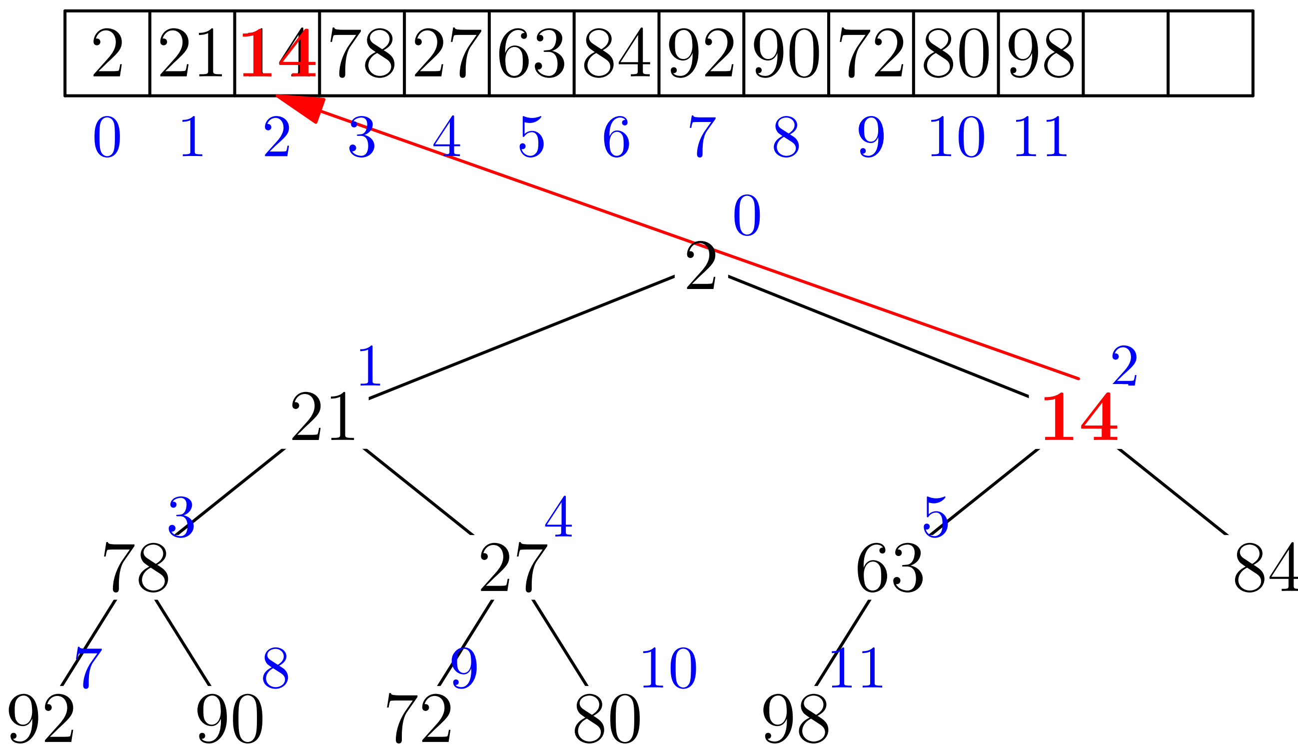
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



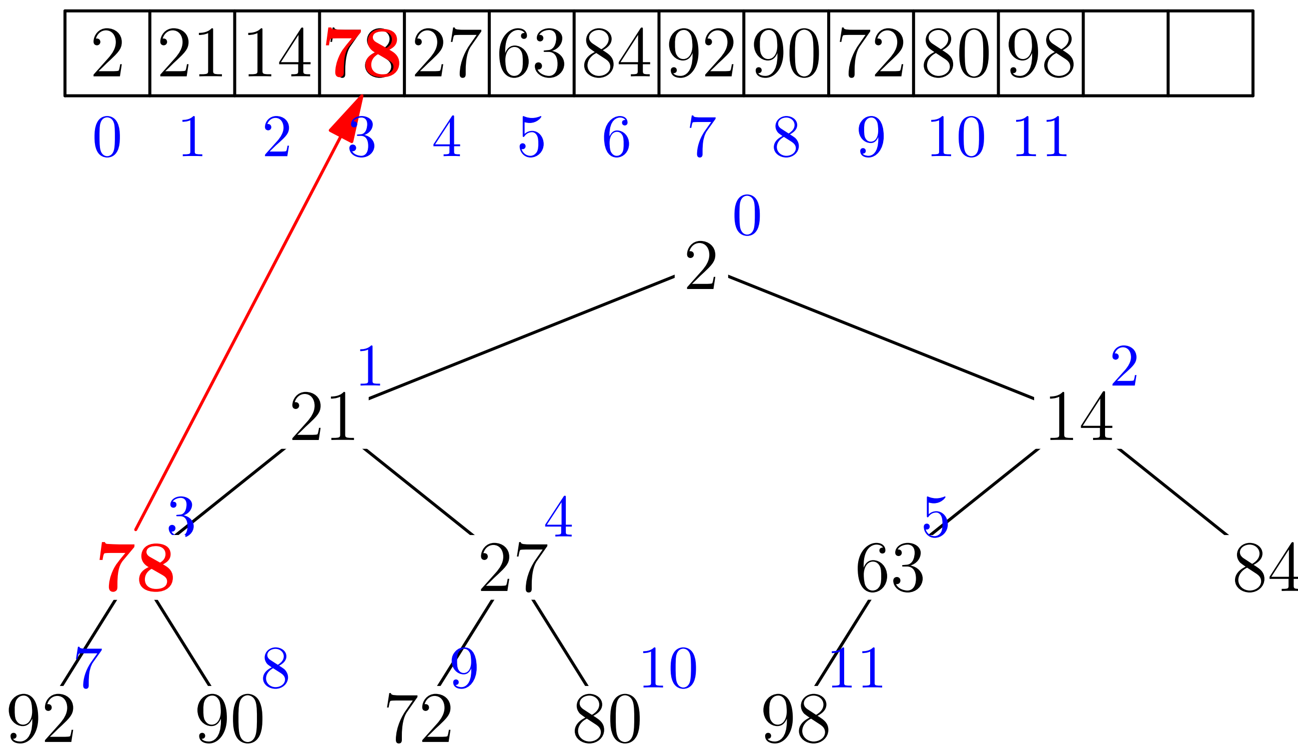
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



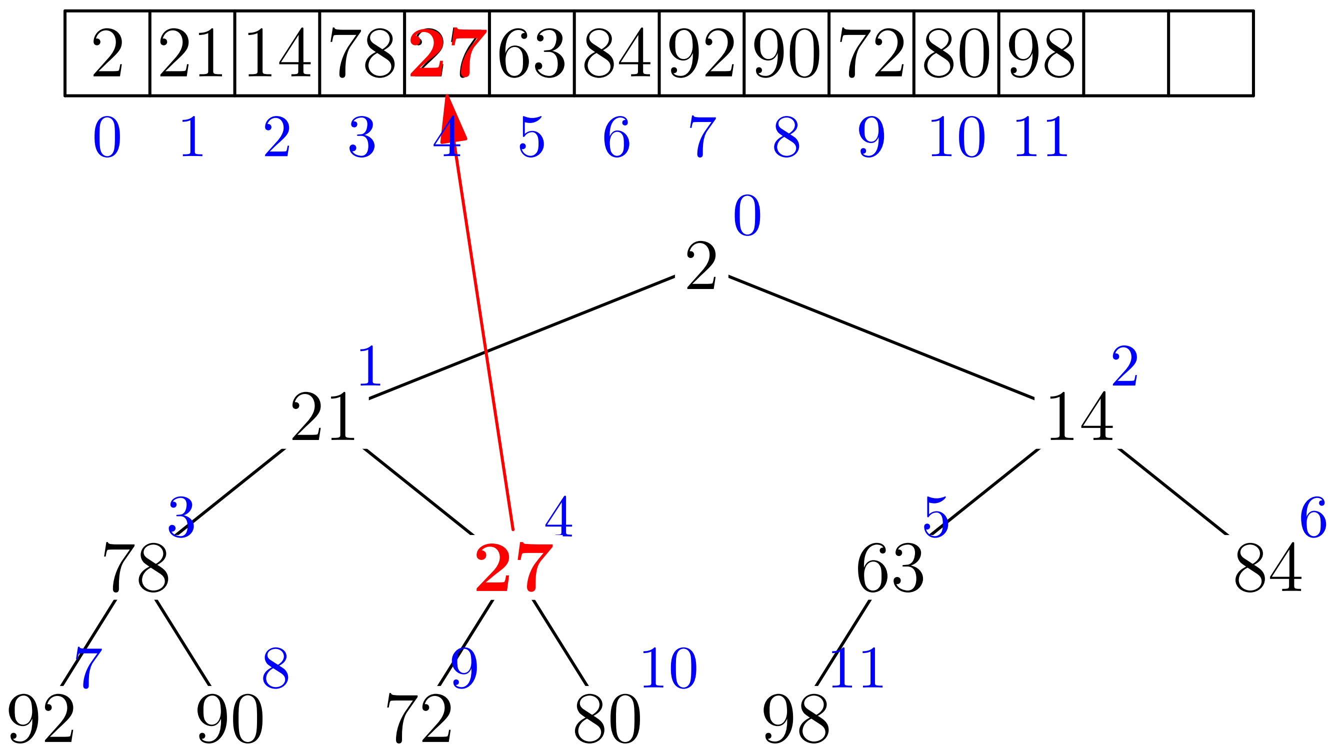
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



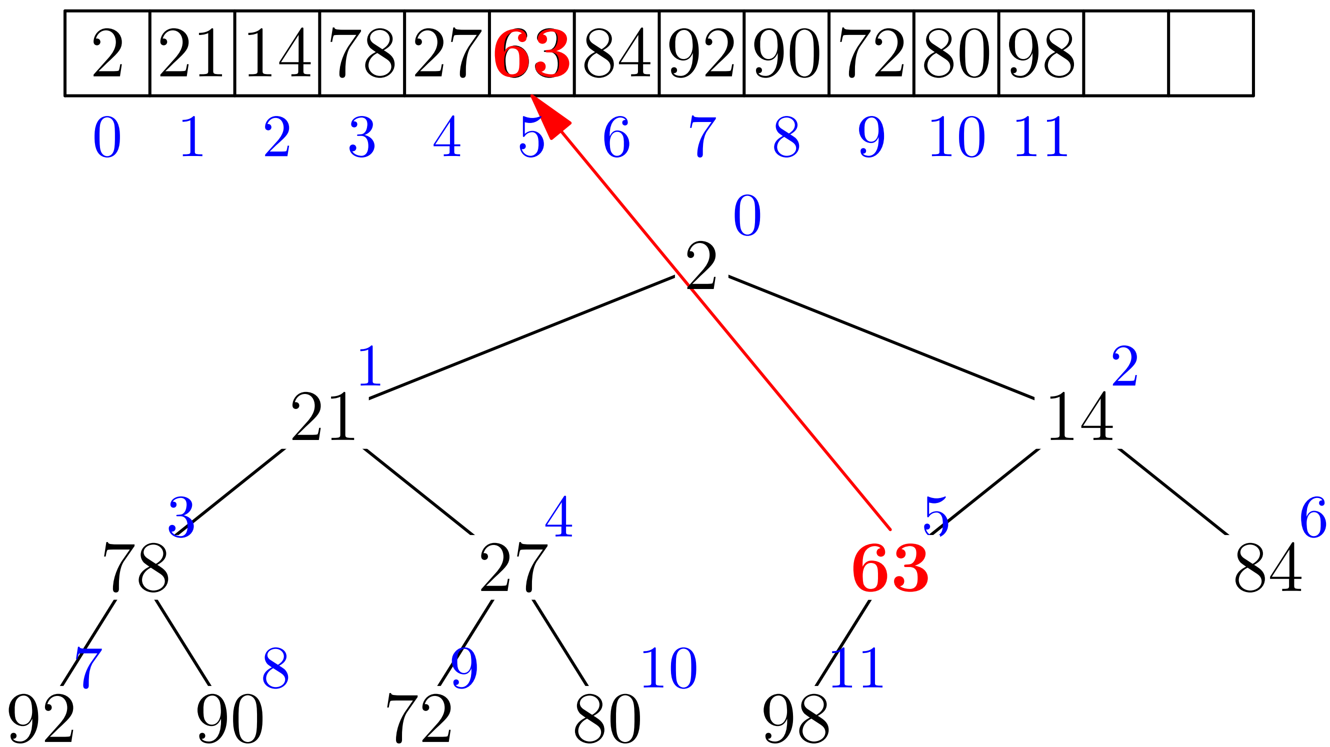
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



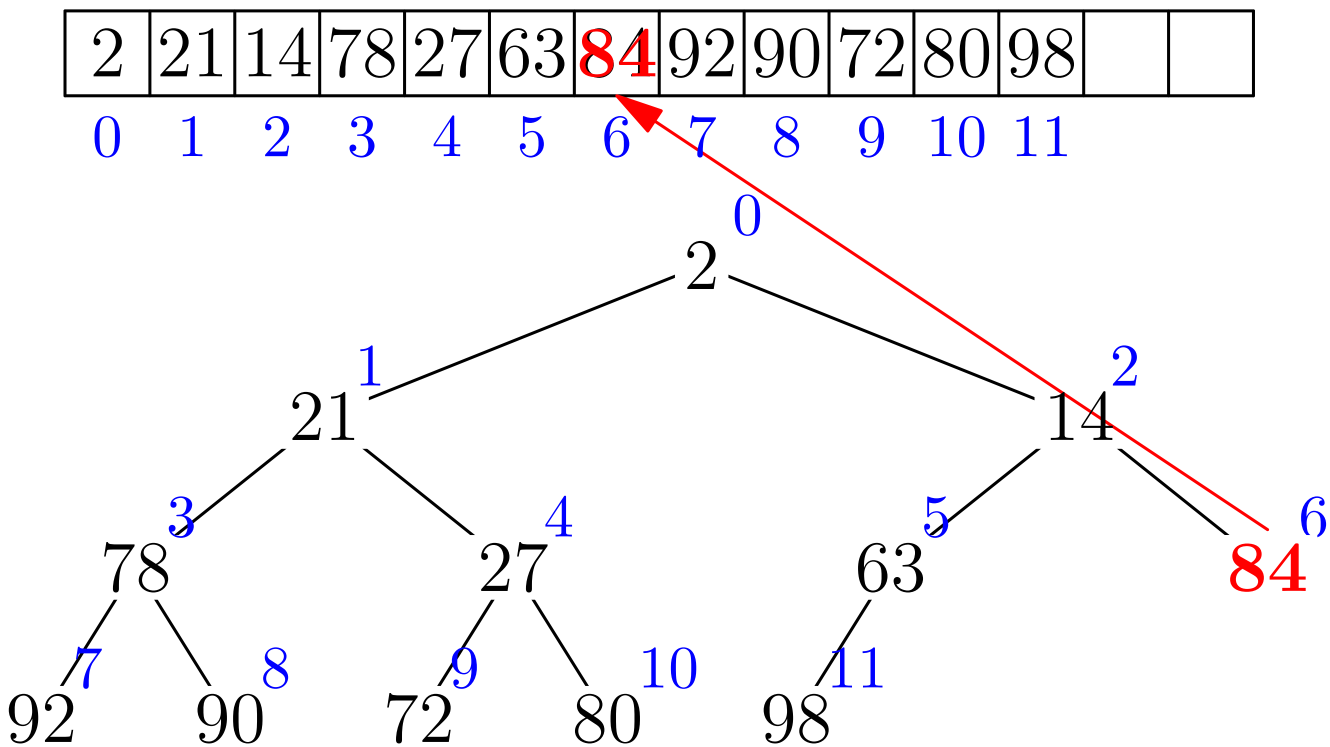
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



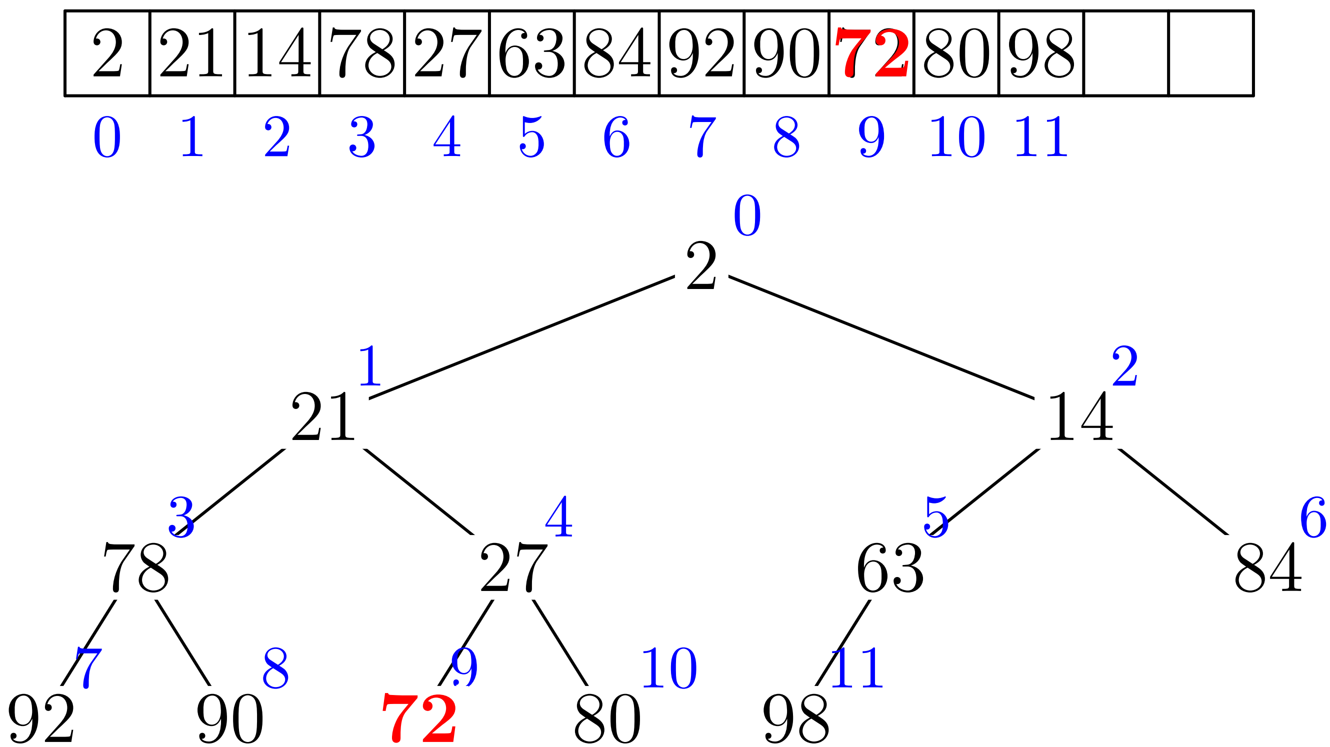
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



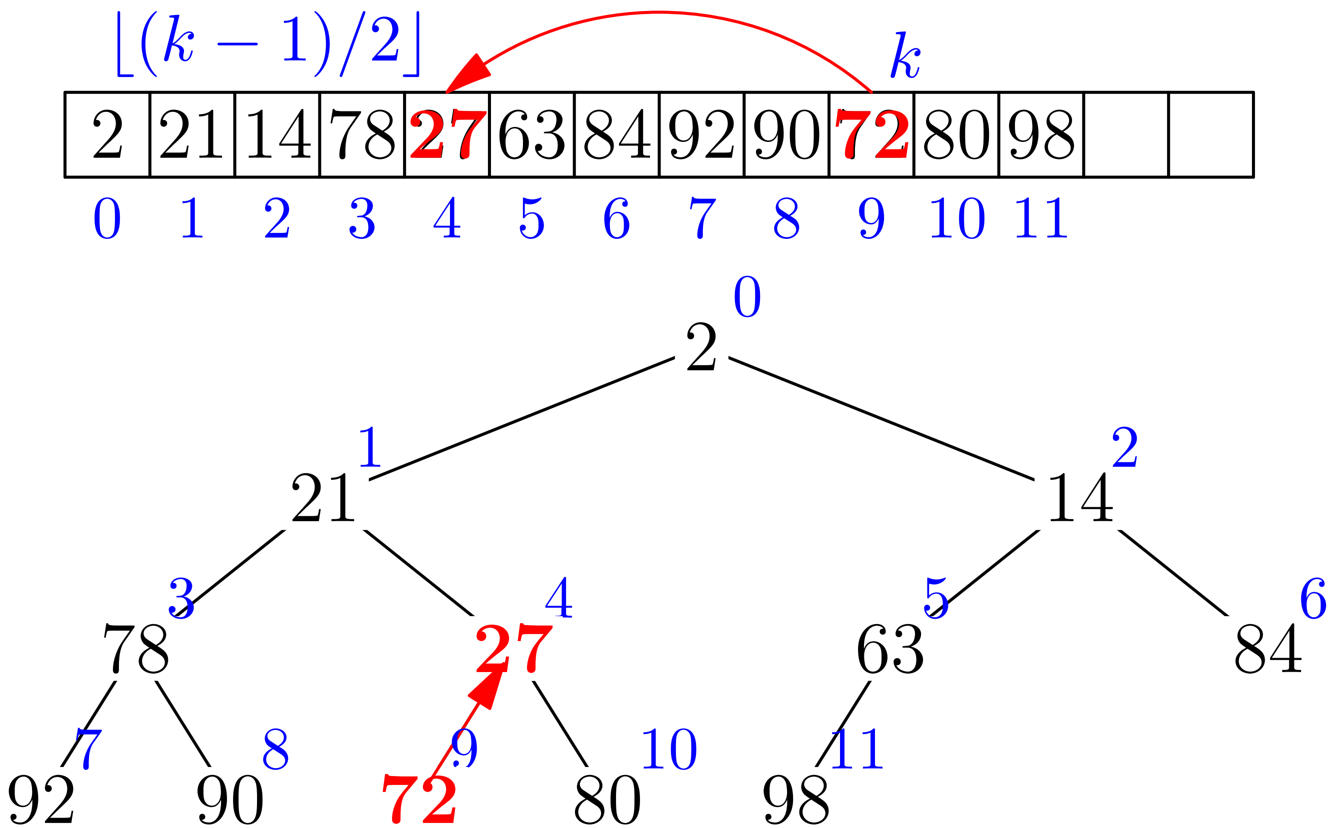
Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



Array Implementation of Heaps

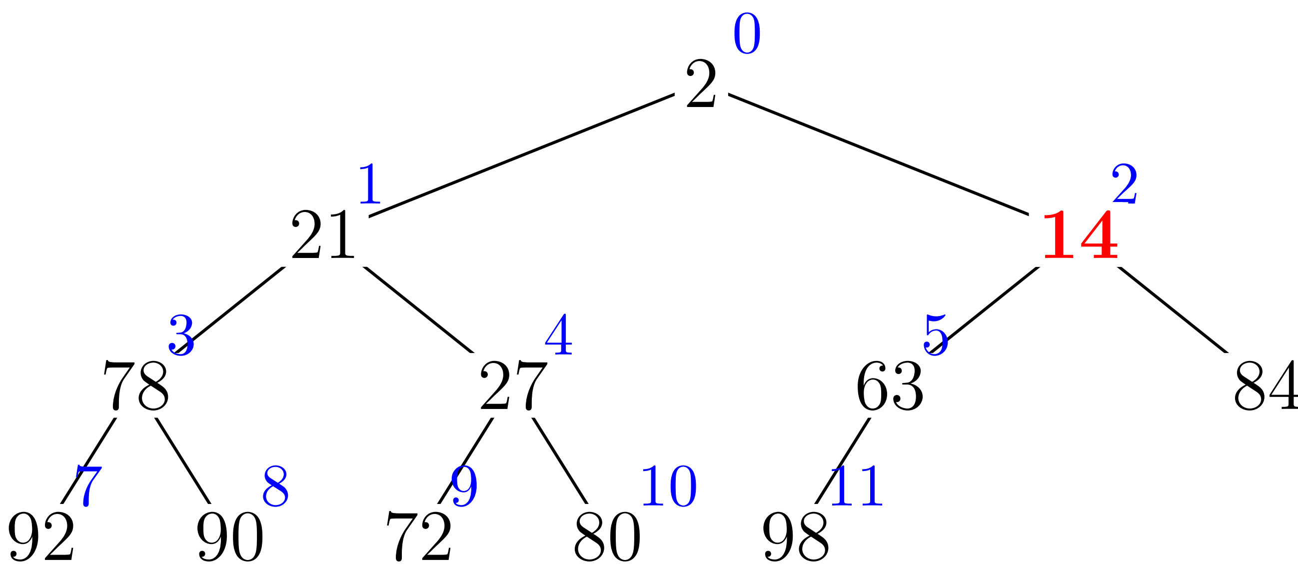
- Because the tree is complete we can implemented the heap efficiently using an array



Array Implementation of Heaps

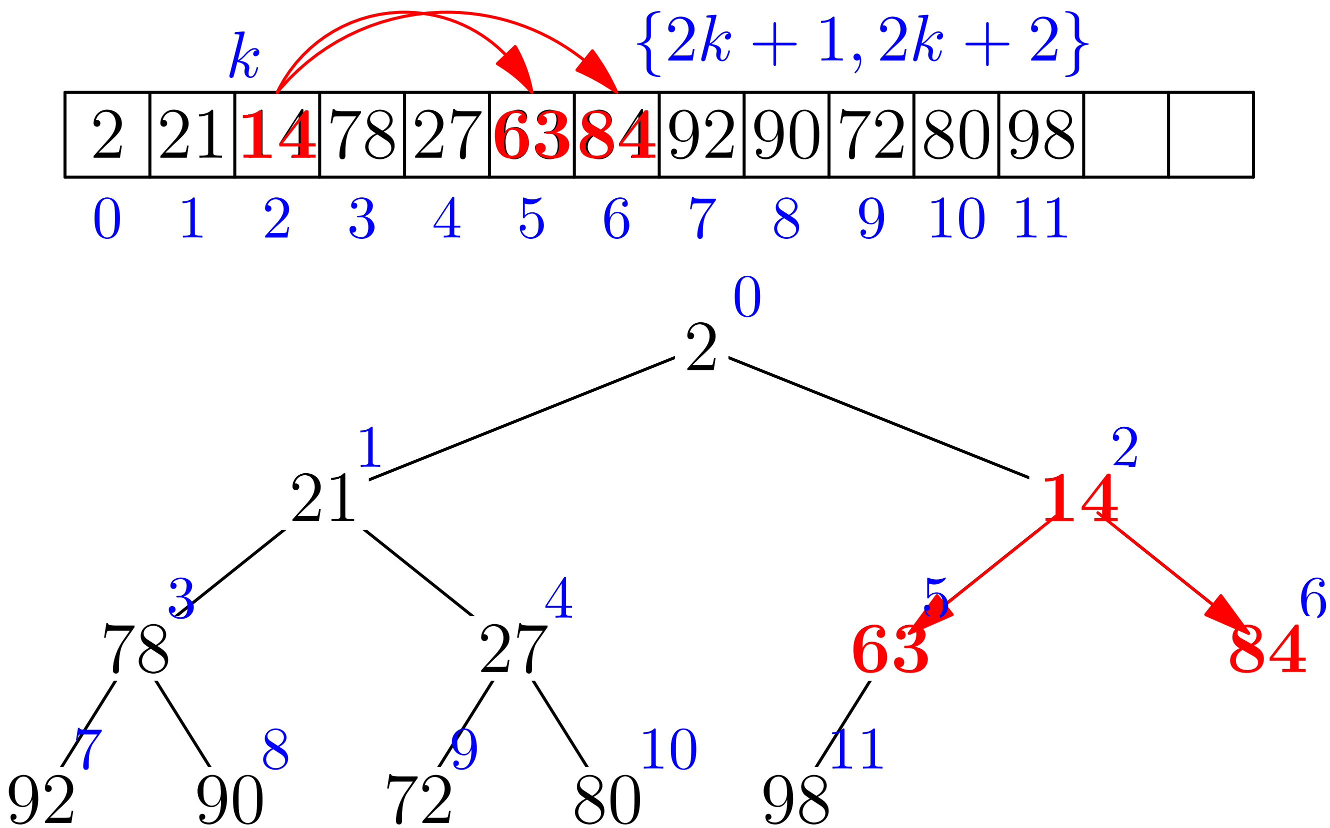
- Because the tree is complete we can implemented the heap efficiently using an array

2	21	**14**	78	27	63	84	92	90	72	80	98		
0	1	2	3	4	5	6	7	8	9	10	11		



Array Implementation of Heaps

- Because the tree is complete we can implemented the heap efficiently using an array



Code for a Priority Queue

```
#include <vector>
using namespace std;

template <typename T, typename P>
class heapPQ {
private:
  vector<pair<T, P> > array;
```

Code for a Priority Queue

```
#include <vector>
using namespace std;

template <typename T, typename P>
class heapPQ {
private:
  vector<pair<T, P> > array;

public:

  heapPQ(unsigned capacity=11) {
    array.reserve(capacity);
  }
```

Code for a Priority Queue

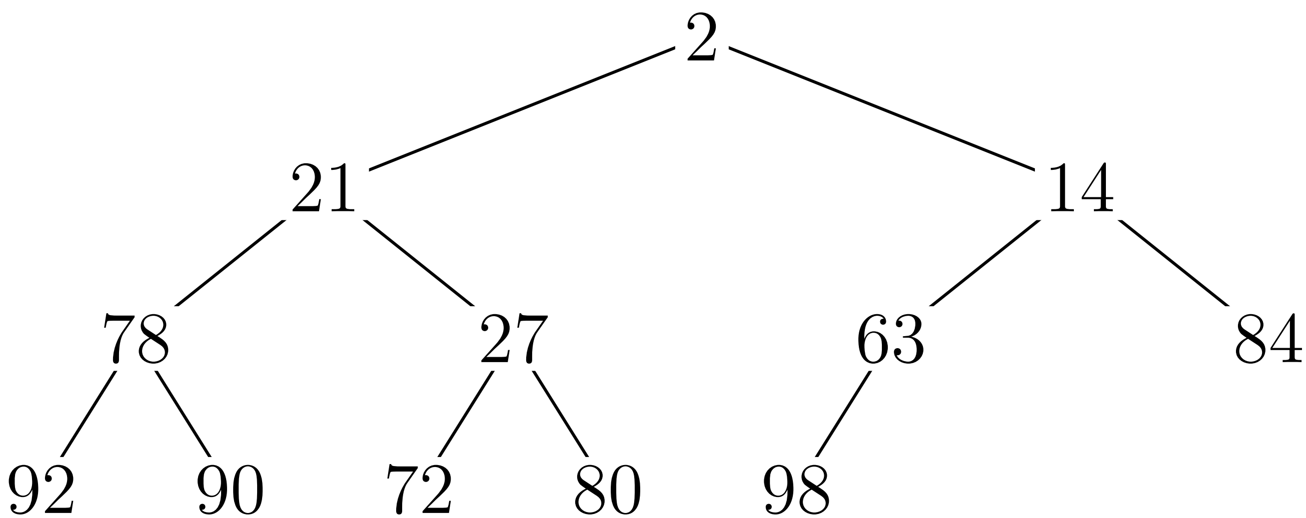
```
#include <vector>
using namespace std;

template <typename T, typename P>
class heapPQ {
private:
  vector<pair<T, P> > array;

public:

  heapPQ(unsigned capacity=11) {
    array.reserve(capacity);
  }

  unsigned size() {return array.size();}

  bool empty() {return array.empty();}
```

Code for a Priority Queue

```
#include <vector>
using namespace std;

template <typename T, typename P>
class heapPQ {
private:
  vector<pair<T, P> > array;

public:

  heapPQ(unsigned capacity=11) {
    array.reserve(capacity);
  }

  unsigned size() {return array.size();}

  bool empty() {return array.empty();}

  const T& top() {return array[0].first;}
```

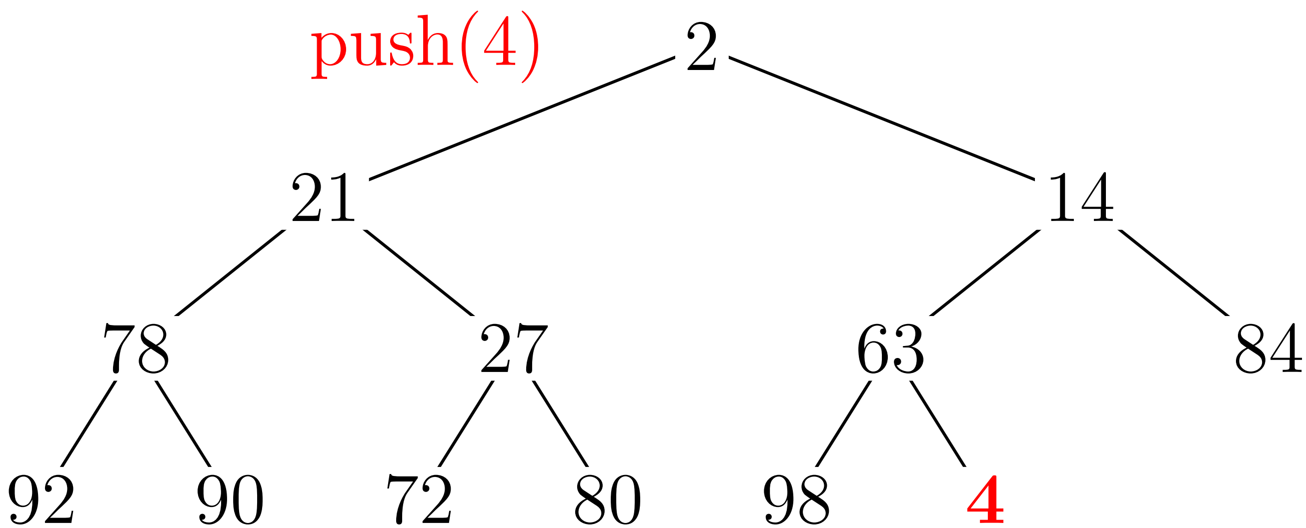
Adding an Element

2	21	14	78	27	63	84	92	90	72	80	98		



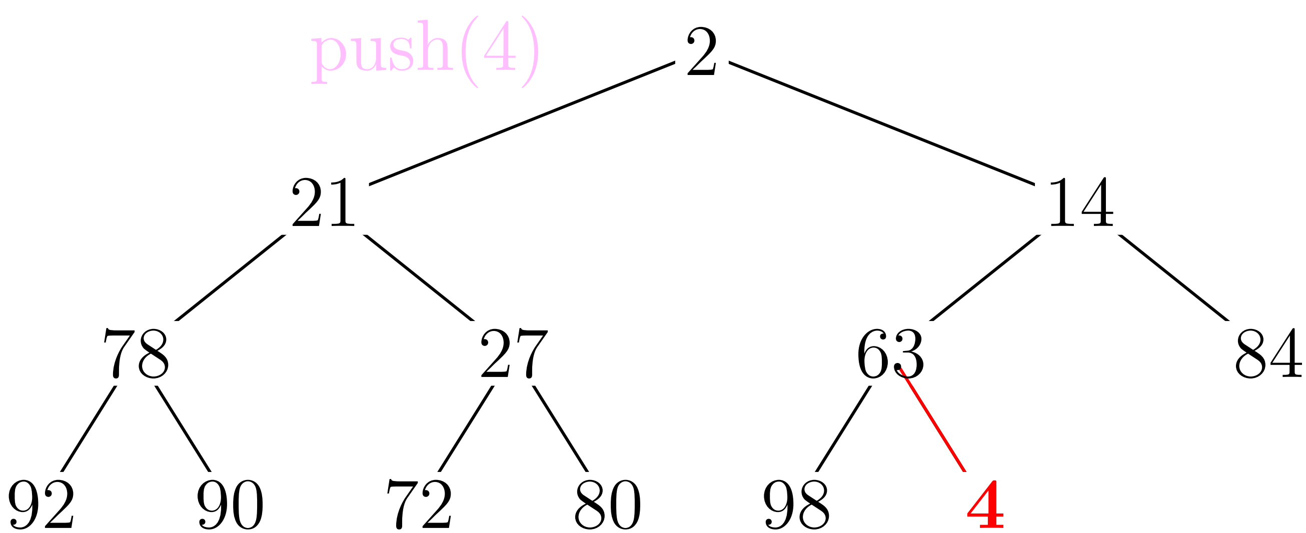
Adding an Element

2	21	14	78	27	63	84	92	90	72	80	98	**4**	



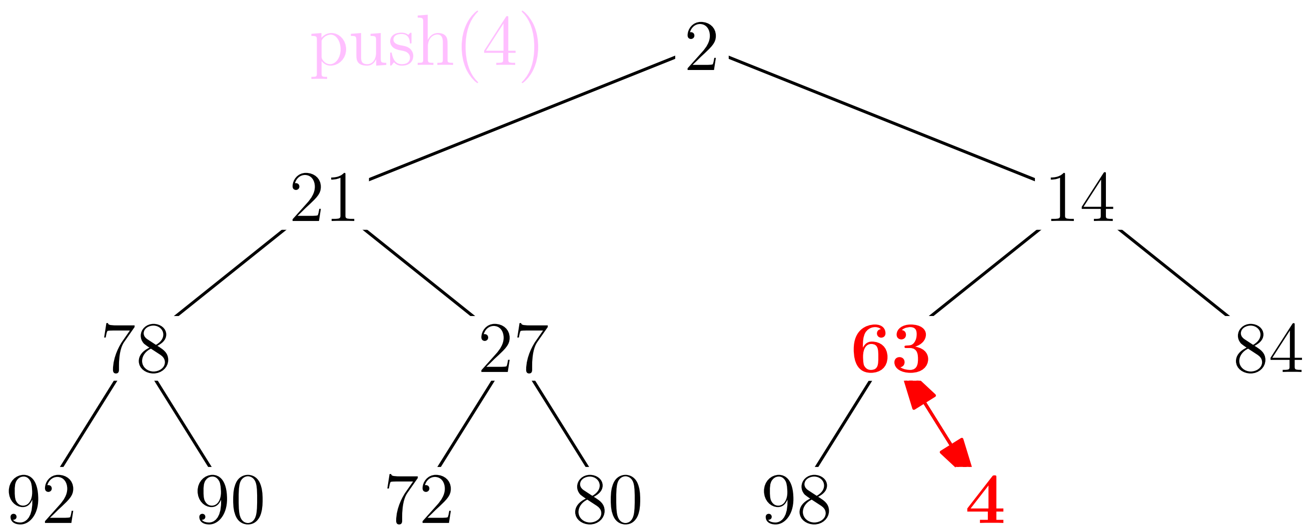
Adding an Element

2	21	14	78	27	63	84	92	90	72	80	98	**4**	

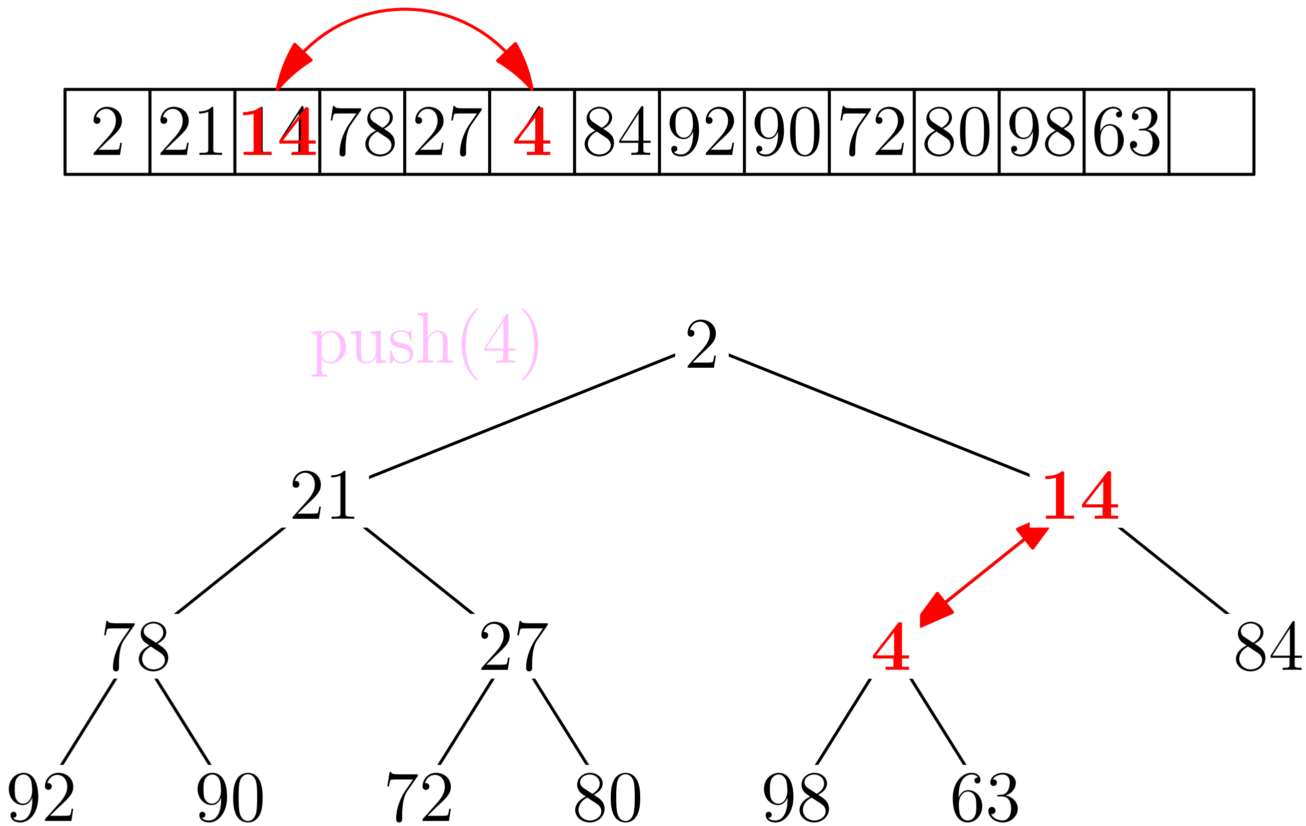


Adding an Element

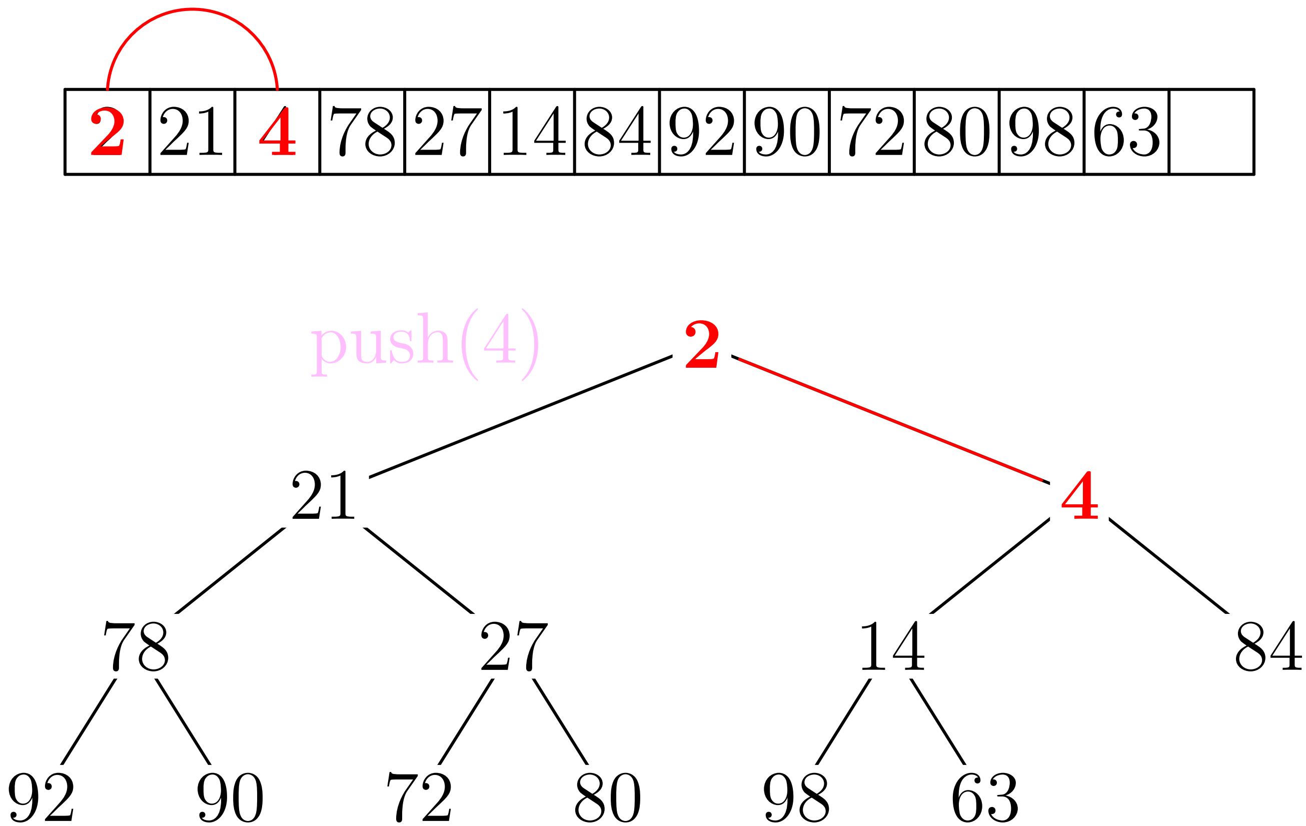
2	21	14	78	27	**63**	84	92	90	72	80	98	**4**	



Adding an Element



Adding an Element



Adding an Element

```
void push(T value, P priority) {
  pair<T,P> tmp(value, priority);
  array.push_back(tmp);
  unsigned child = size() - 1;

  /* Percolate Up */
  while(child!=0) {
    unsigned parent = (child-1)>>1;  // floor((child-1)/2)
    if (array[parent].second < array[child].second)
      return;
    array[child] = array[parent];
    array[parent] = tmp;
    child = parent;
  }
}
```

Adding an Element

```
void push(T value, P priority) {
  pair<T,P> tmp(value, priority);
  array.push_back(tmp);
  unsigned child = size() - 1;

  /* Percolate Up */
  while(child!=0) {
    unsigned parent = (child-1)>>1;  // floor((child-1)/2)
    if (array[parent].second < array[child].second)
      return;
    array[child] = array[parent];
    array[parent] = tmp;
    child = parent;
  }
}
```

Adding an Element

```
void push(T value, P priority) {
  pair<T,P> tmp(value, priority);
  array.push_back(tmp);
  unsigned child = size() - 1;

  /* Percolate Up */
  while(child!=0) {
    unsigned parent = (child-1)>>1;  // floor((child-1)/2)
    if (array[parent].second < array[child].second)
      return;
    array[child] = array[parent];
    array[parent] = tmp;
    child = parent;
  }
}
```

Adding an Element

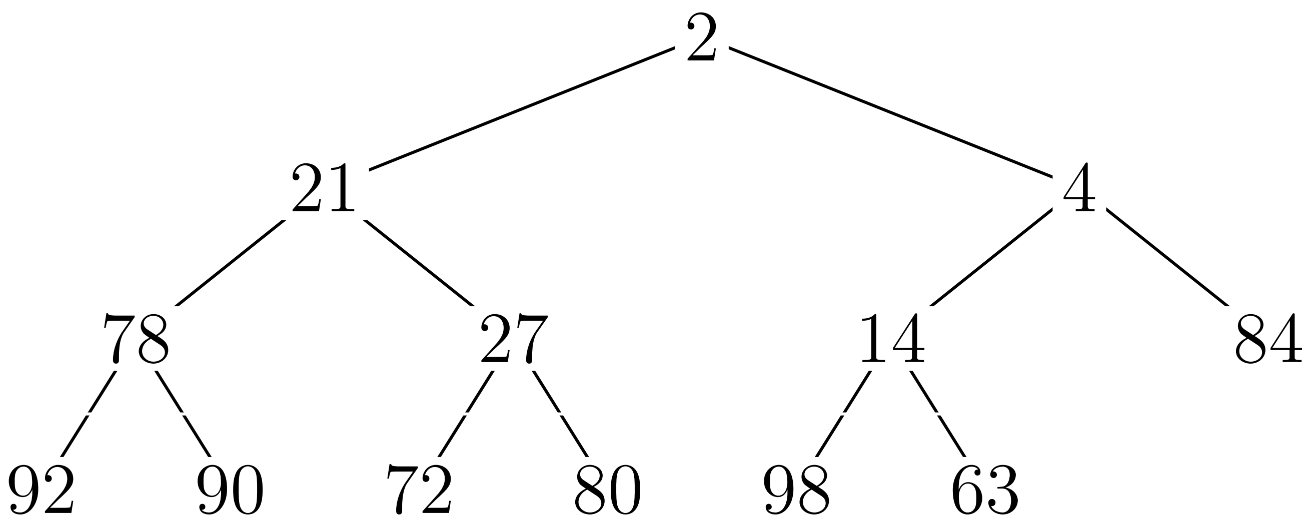
```
void push(T value, P priority) {
  pair<T,P> tmp(value, priority);
  array.push_back(tmp);
  unsigned child = size() - 1;

  /* Percolate Up */
  while(child!=0) {
    unsigned parent = (child-1)>>1;  // floor((child-1)/2)
    if (array[parent].second < array[child].second)
      return;
    array[child] = array[parent];
    array[parent] = tmp;
    child = parent;
  }
}
```

Adding an Element

```
void push(T value, P priority) {
  pair<T,P> tmp(value, priority);
  array.push_back(tmp);
  unsigned child = size() - 1;

  /* Percolate Up */
  while(child!=0) {
    unsigned parent = (child-1)>>1;  // floor((child-1)/2)
    if (array[parent].second < array[child].second)
      return;
    array[child] = array[parent];
    array[parent] = tmp;
    child = parent;
  }
}
```

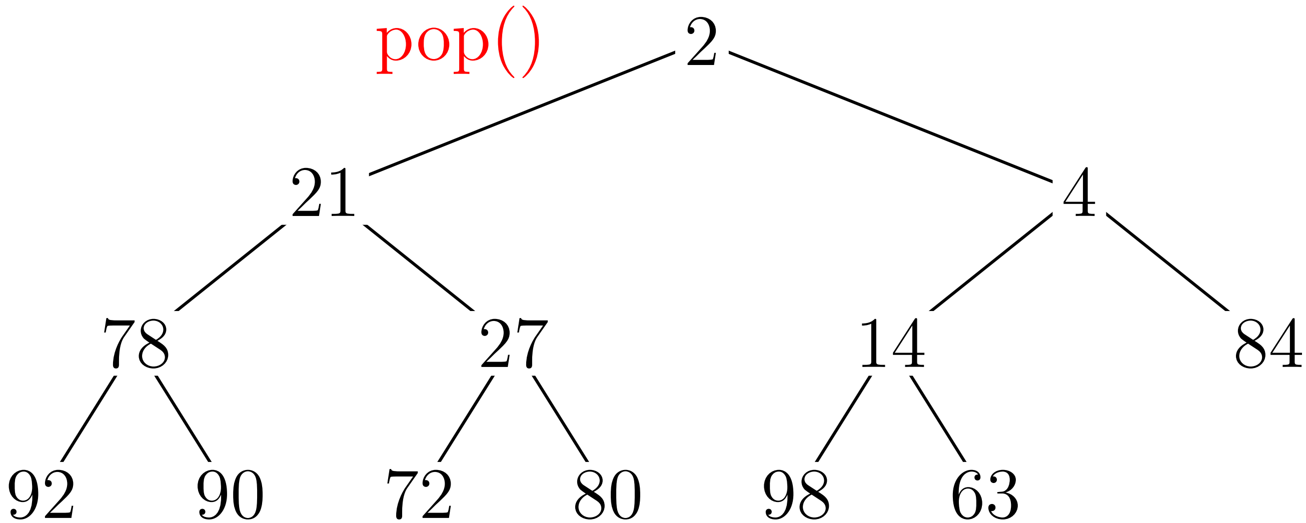
Popping the Top

2	21	4	78	27	14	84	92	90	72	80	98	63	



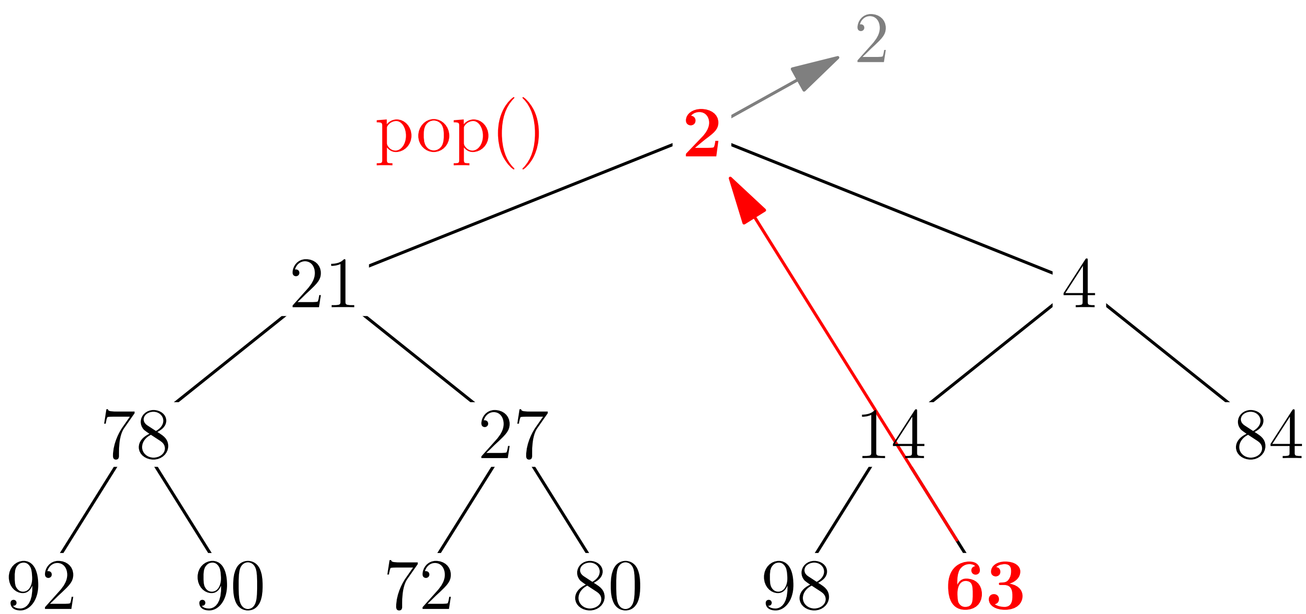
Popping the Top

2	21	4	78	27	14	84	92	90	72	80	98	63	



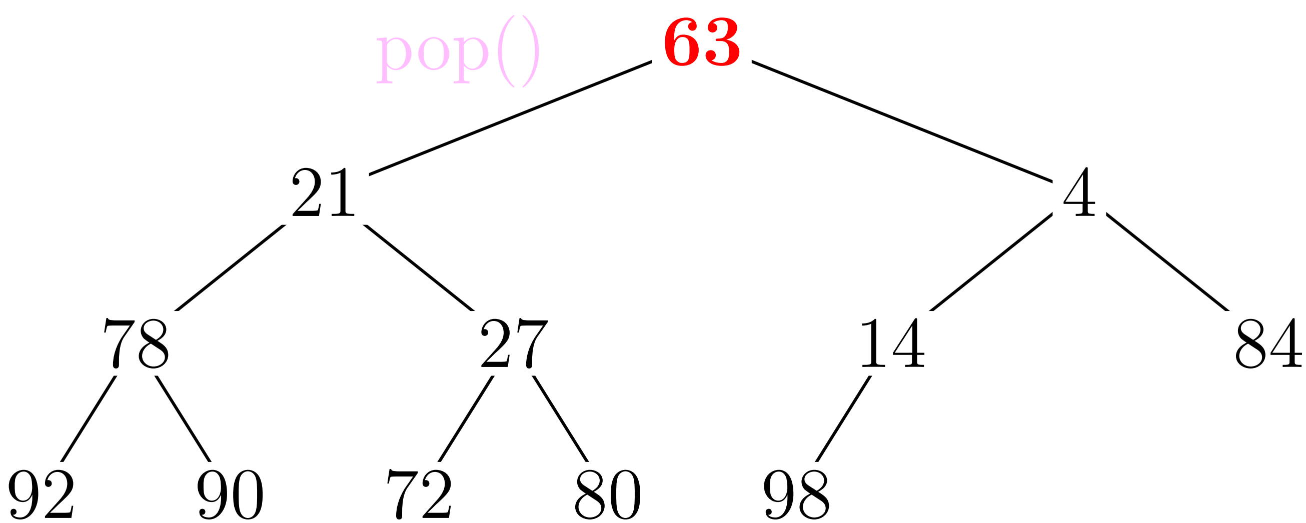
Popping the Top

2	21	4	78	27	14	84	92	90	72	80	98	**63**	



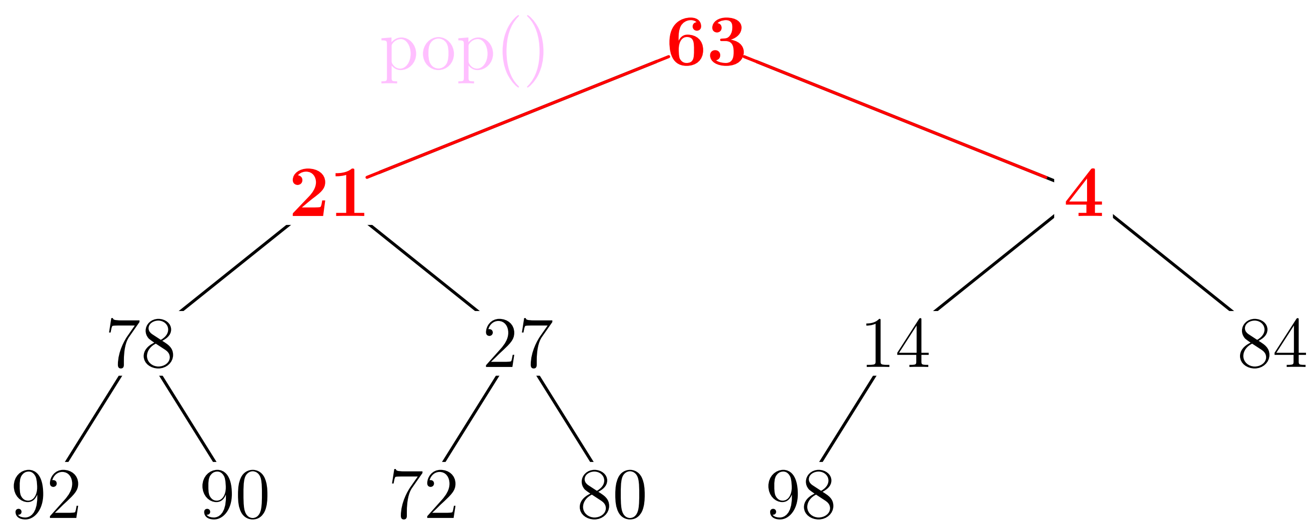
Popping the Top

63	21	4	78	27	14	84	92	90	72	80	98		

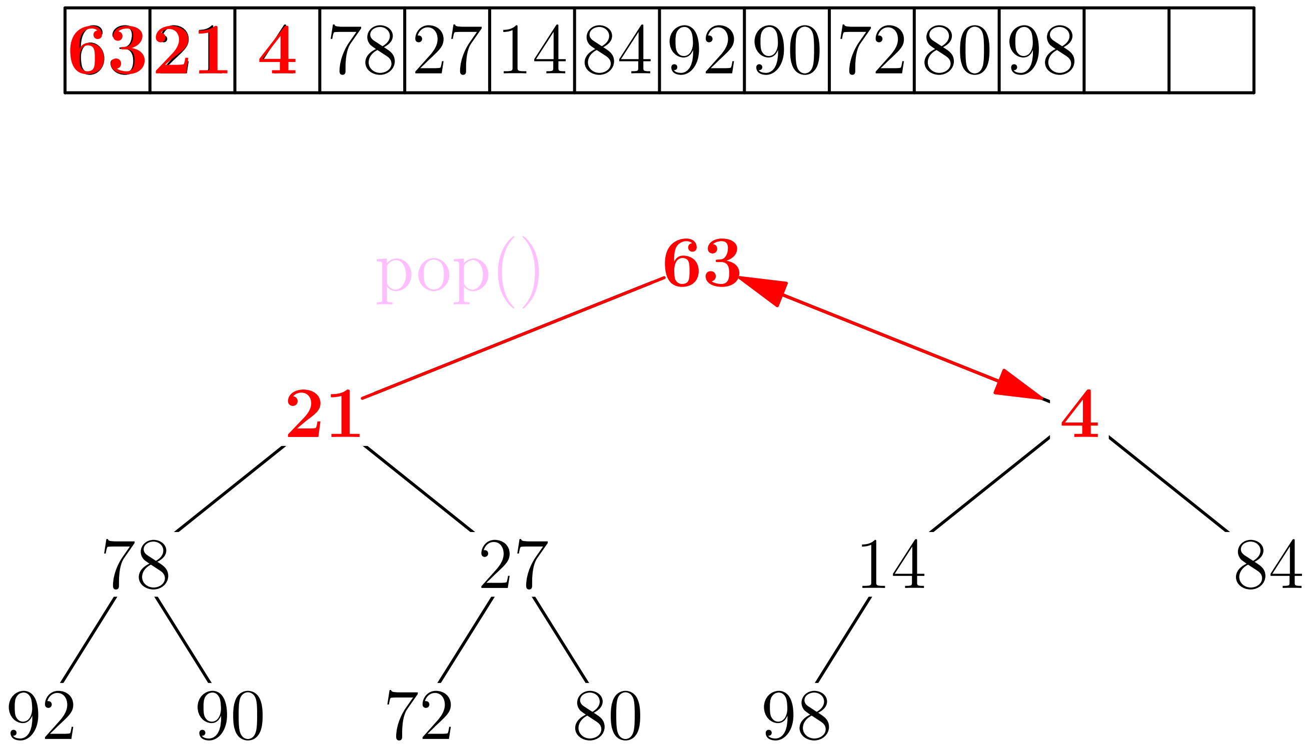


Popping the Top

63	**21**	**4**	78	27	14	84	92	90	72	80	98		

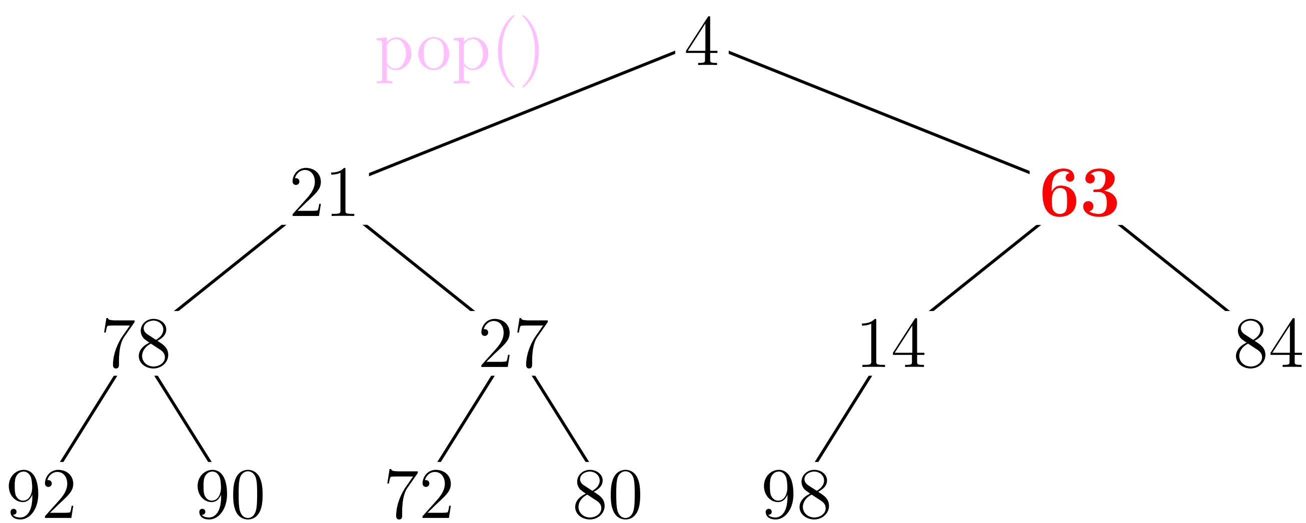


Popping the Top



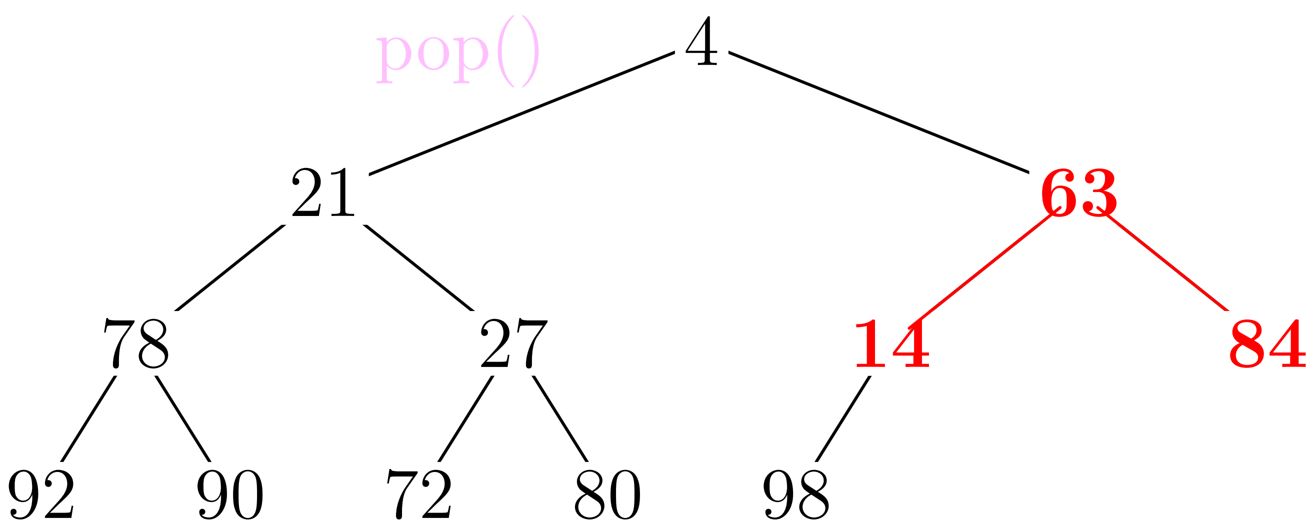
Popping the Top

4	21	**63**	78	27	14	84	92	90	72	80	98		



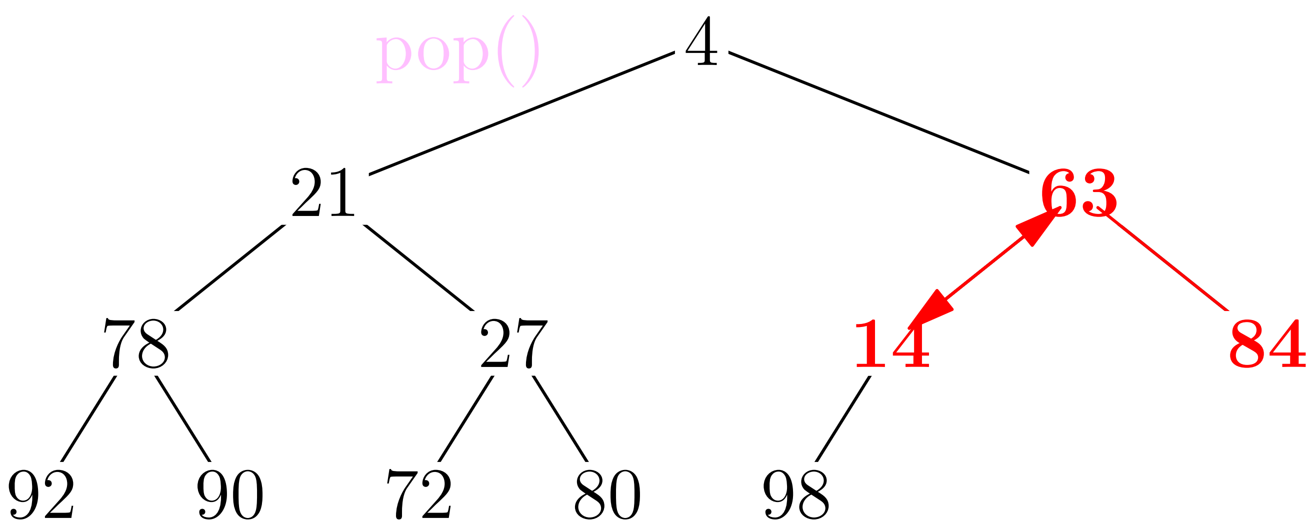
Popping the Top

4	21	**63**	78	27	**14**	**84**	92	90	72	80	98		

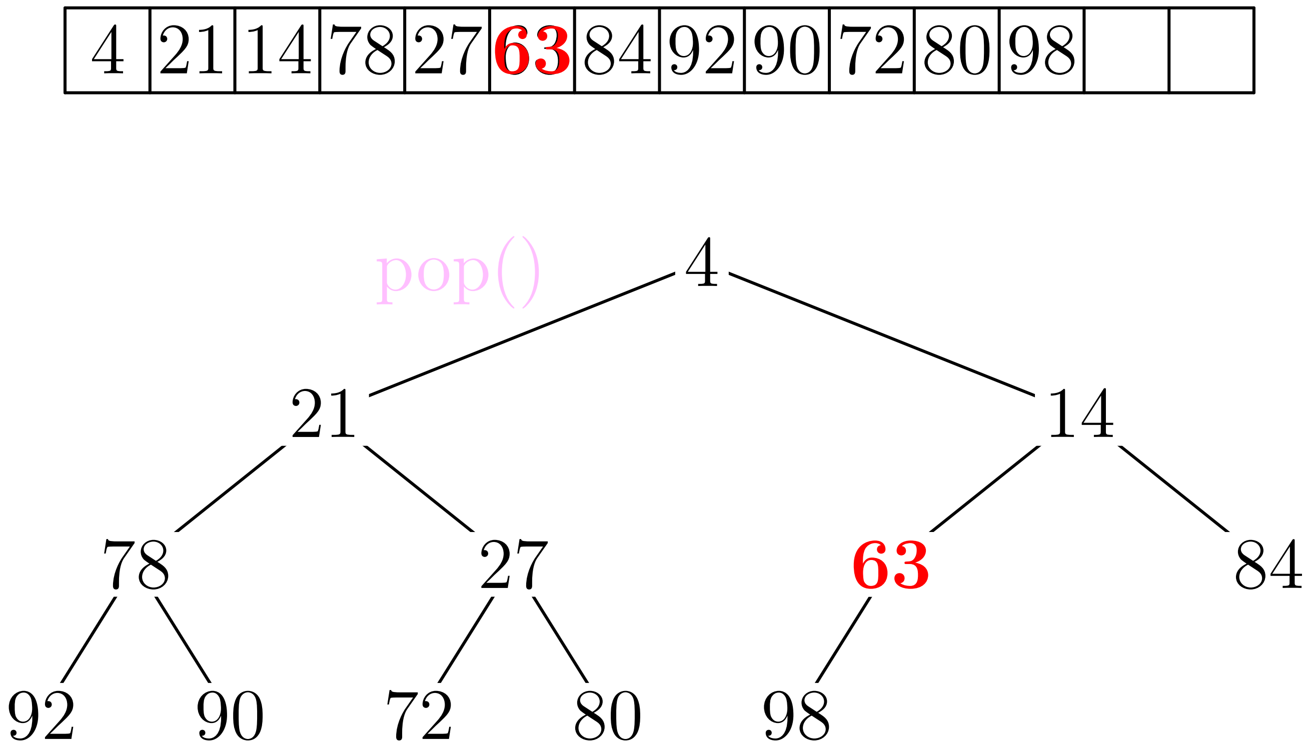


Popping the Top

4	21	**63**	78	27	**14**	**84**	92	90	72	80	98		

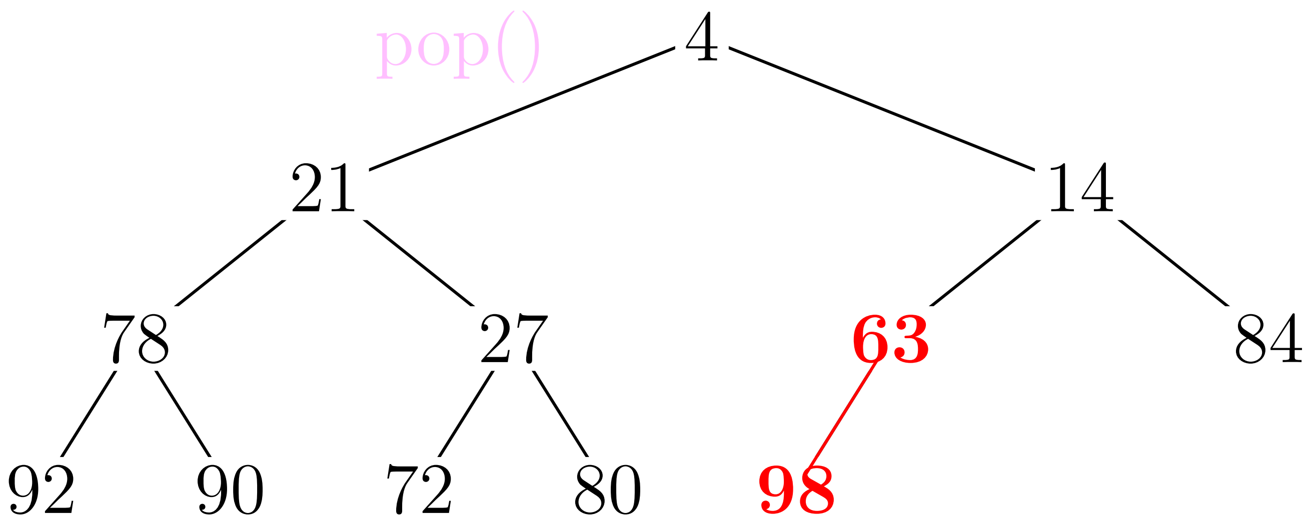


Popping the Top



Popping the Top

4	21	14	78	27	**63**	84	92	90	72	80	**98**		



Popping the Top

```
void pop() {
unsigned parent = 0;
pair<T, P> tmp = array.back();
array[0] = tmp;
array.pop_back();
```

Popping the Top

```
void pop() {
unsigned parent = 0;
pair<T, P> tmp = array.back();
array[0] = tmp;
array.pop_back();
unsigned child = 1;

/* Percolate down */
while(child<size()) {
```

Popping the Top

```
void pop() {
unsigned parent = 0;
pair<T, P> tmp = array.back();
array[0] = tmp;
array.pop_back();
unsigned child = 1;

/* Percolate down */
while(child<size()) {
  if (child+1<=size() && array[child+1].second < array[child].second)
    ++child;
```

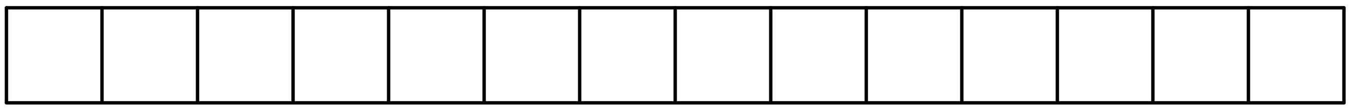
Popping the Top

```
void pop() {
unsigned parent = 0;
pair<T, P> tmp = array.back();
array[0] = tmp;
array.pop_back();
unsigned child = 1;

/* Percolate down */
while(child<size()) {
  if (child+1<=size() && array[child+1].second < array[child].second)
    ++child;
  if (array[child].second > array[parent].second)
    return;
```

Popping the Top

```
void pop() {
unsigned parent = 0;
pair<T, P> tmp = array.back();
array[0] = tmp;
array.pop_back();
unsigned child = 1;

/* Percolate down */
while(child<size()) {
  if (child+1<=size() && array[child+1].second < array[child].second)
    ++child;
  if (array[child].second > array[parent].second)
    return;
  array[parent] = array[child];
  array[child] = tmp;
  parent = child;
  child = 2*parent + 1;
 }
}
```

Heaps in Action

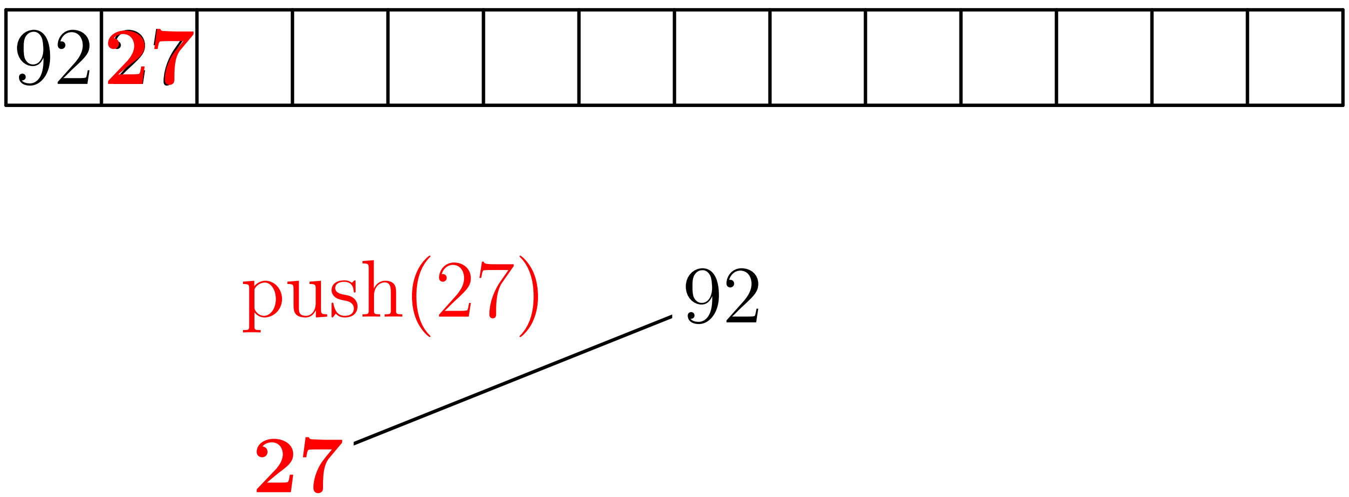


Heaps in Action

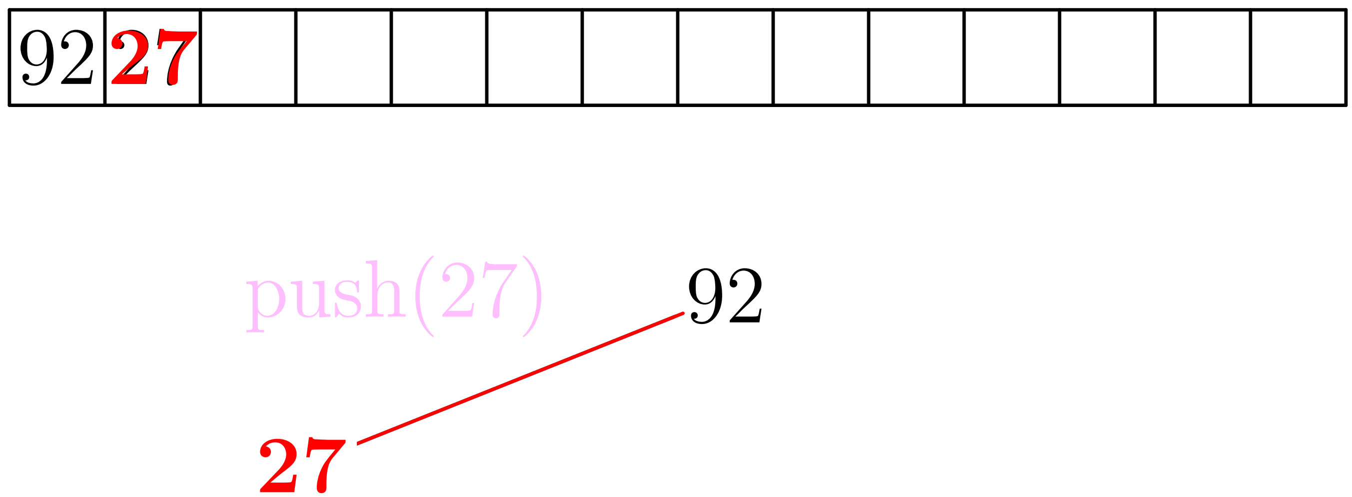
92													

push(92) **92**

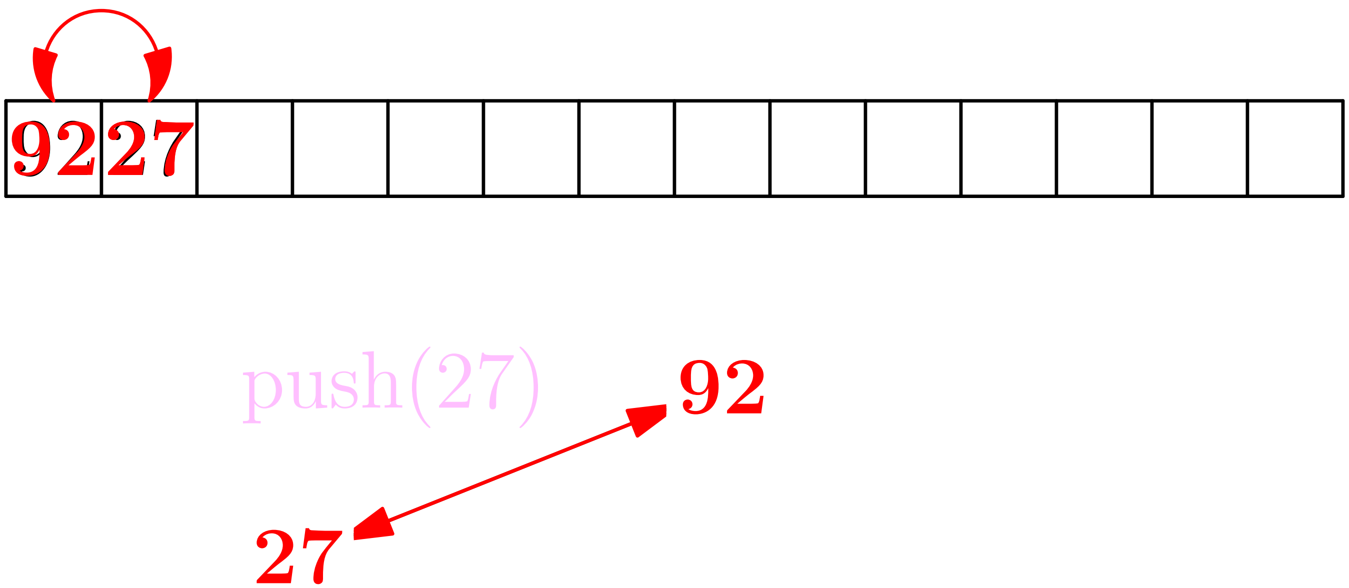
Heaps in Action



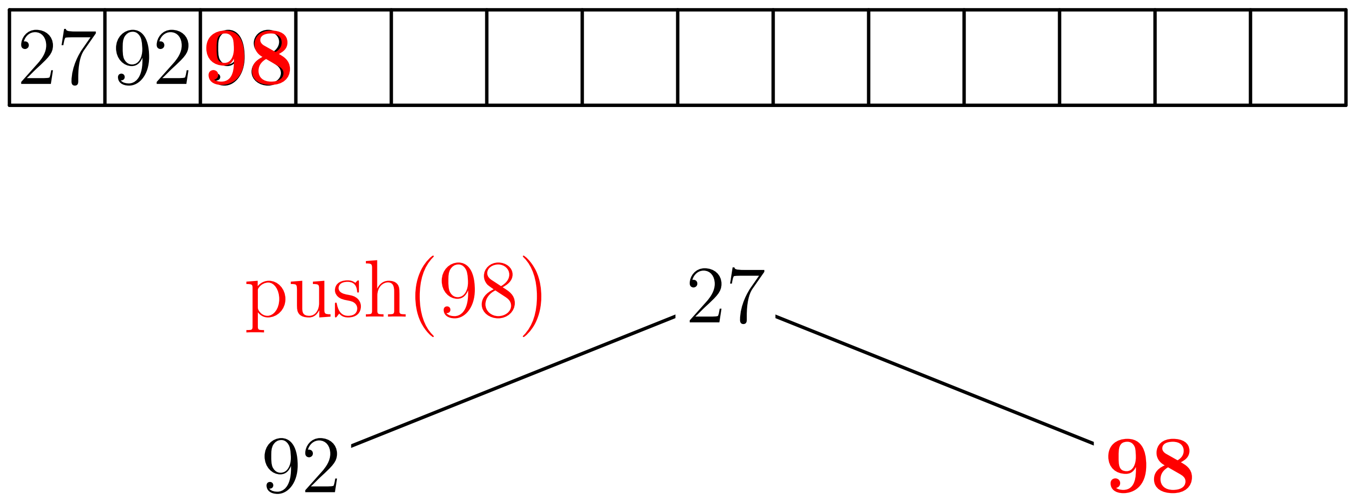
Heaps in Action



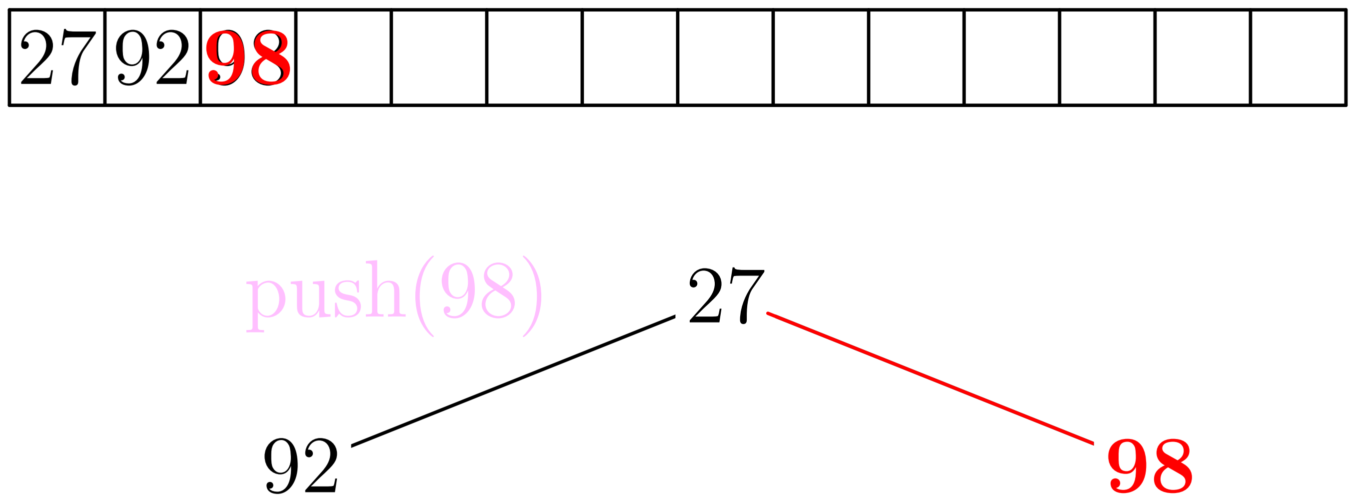
Heaps in Action



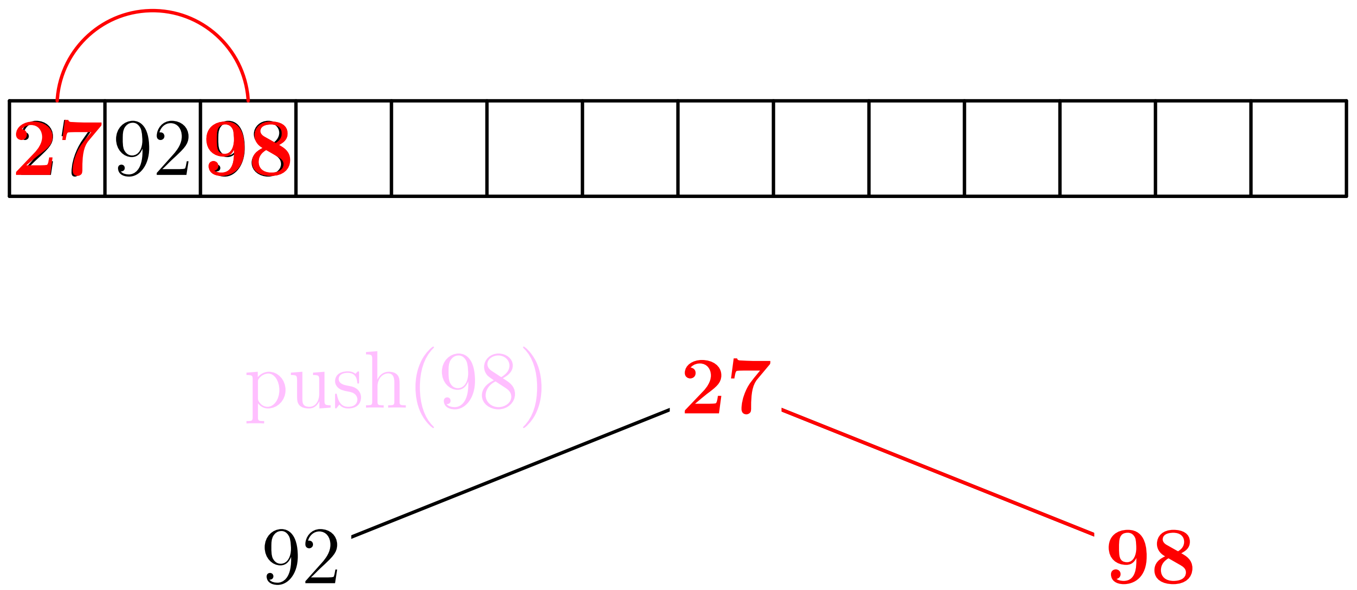
Heaps in Action



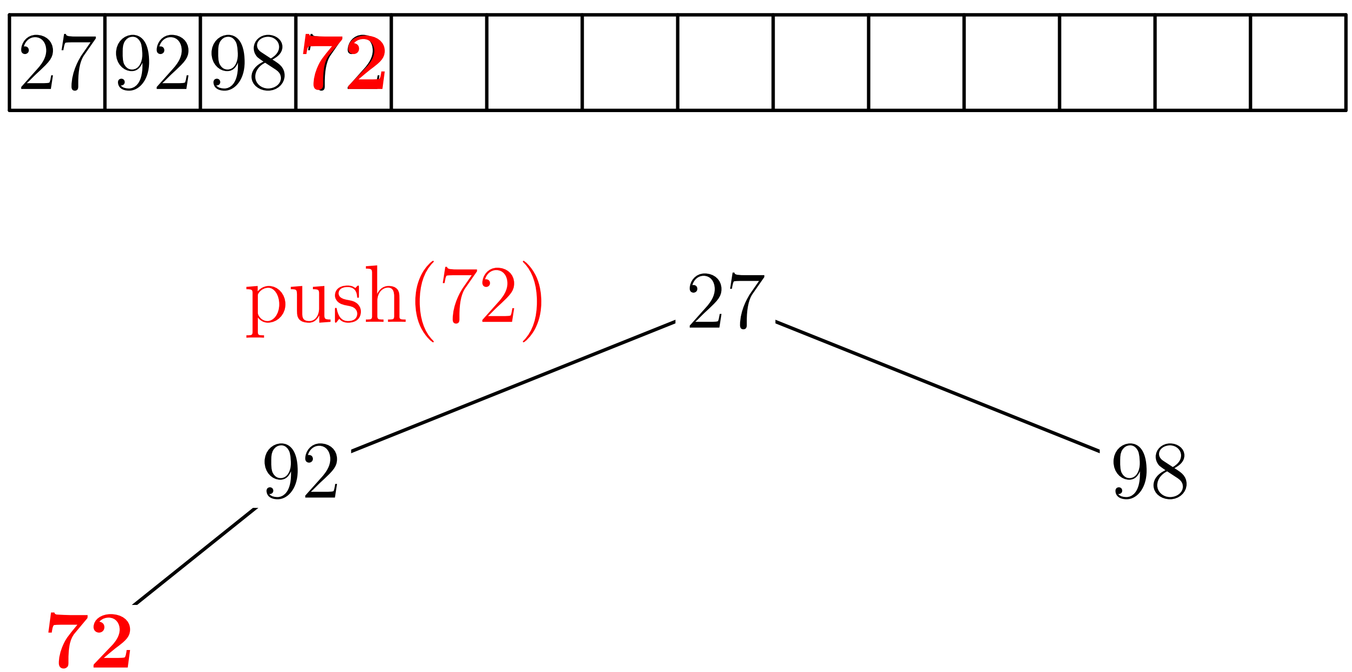
Heaps in Action



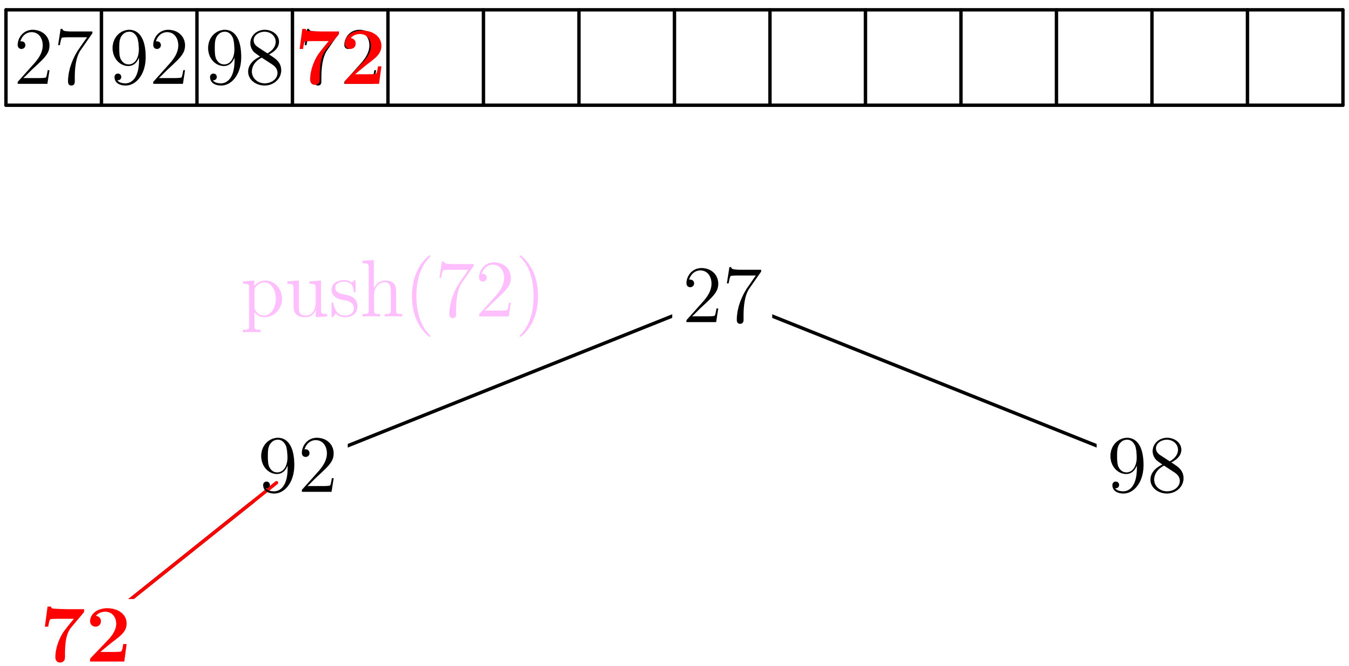
Heaps in Action



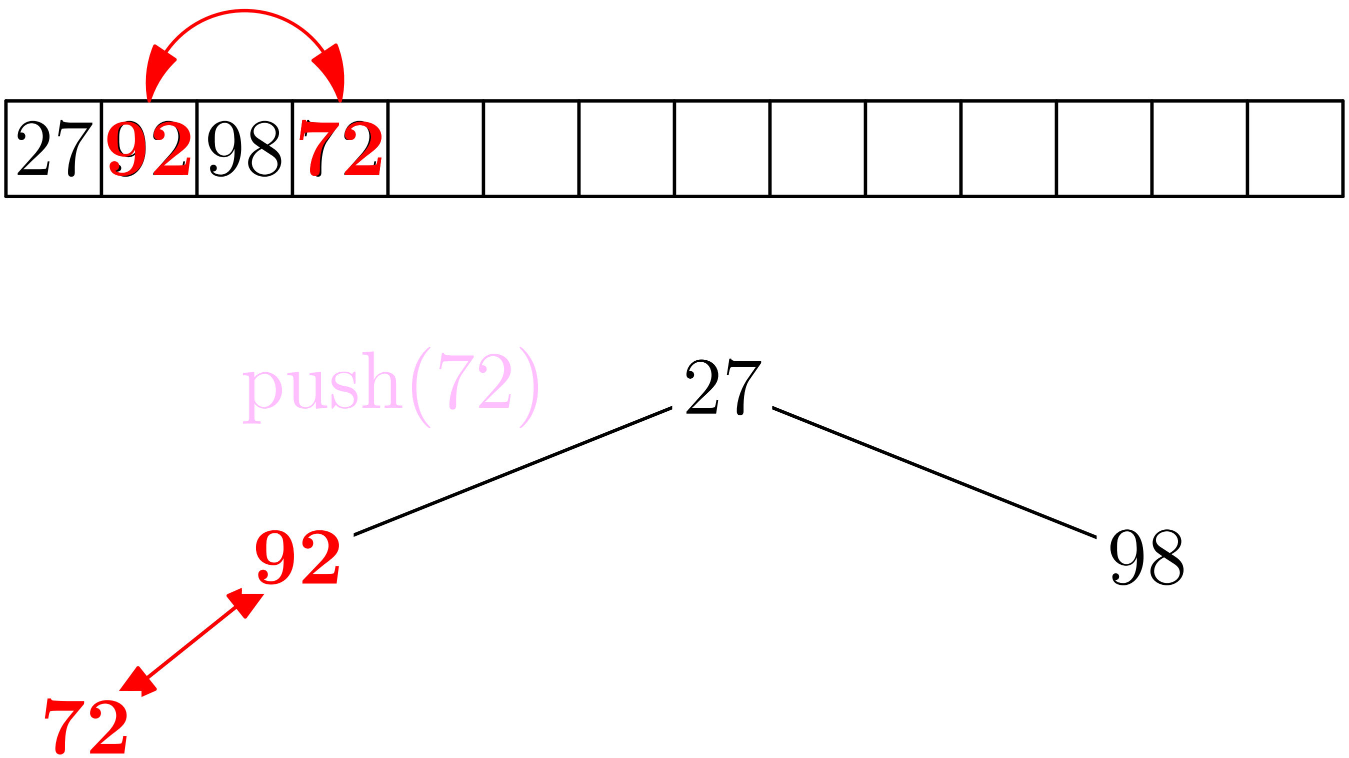
Heaps in Action



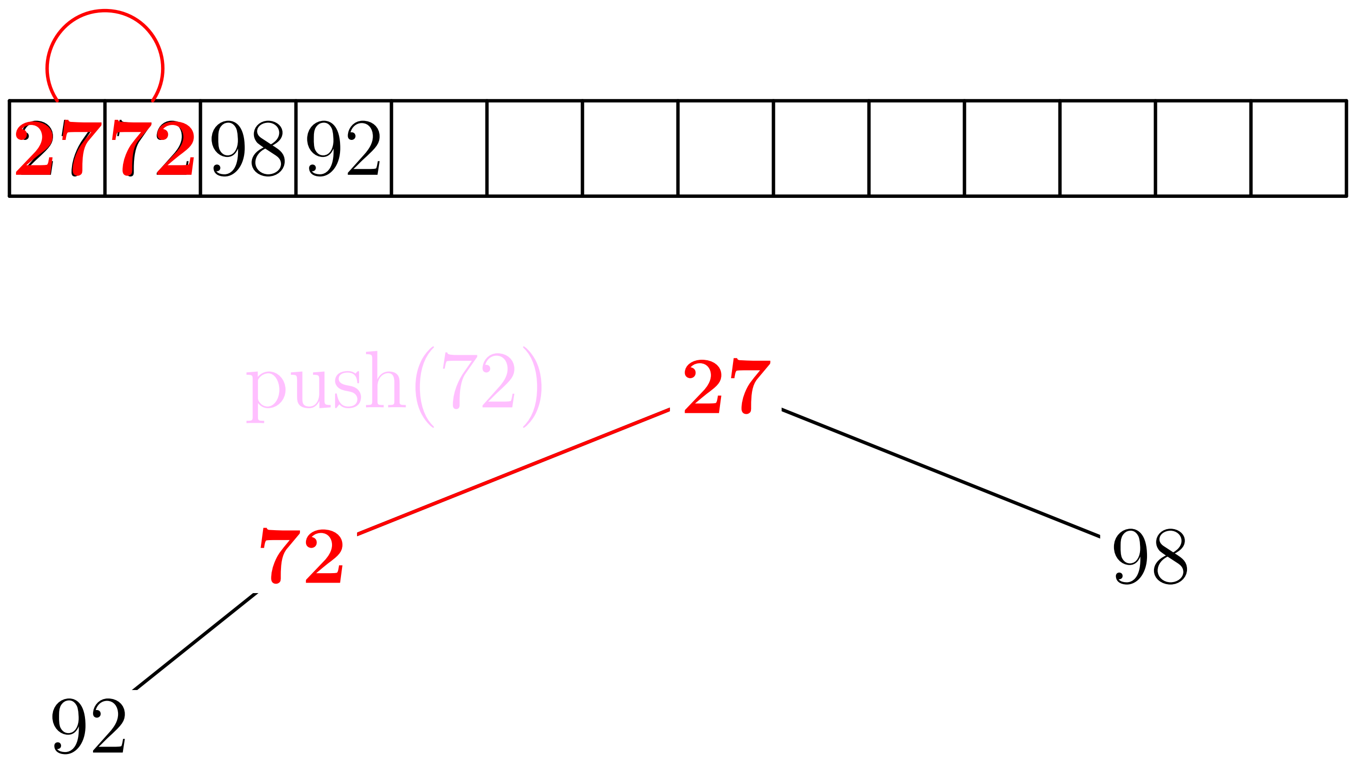
Heaps in Action



Heaps in Action

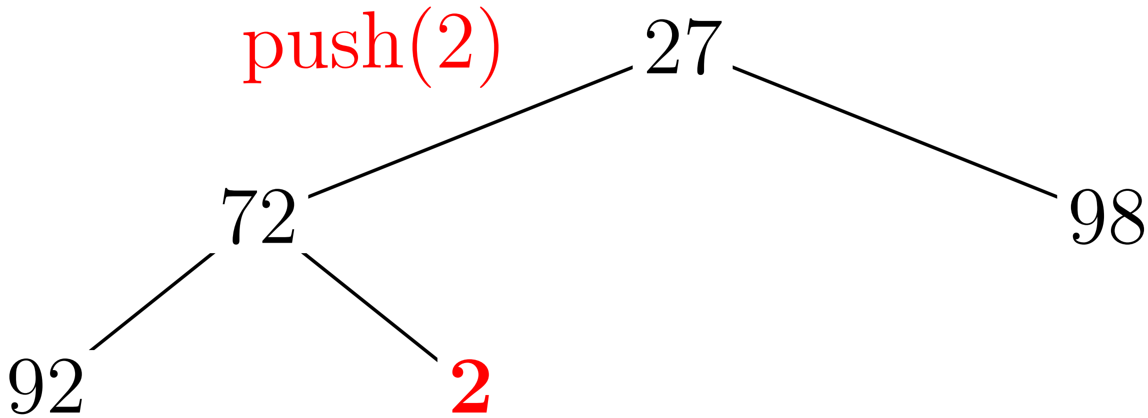


Heaps in Action



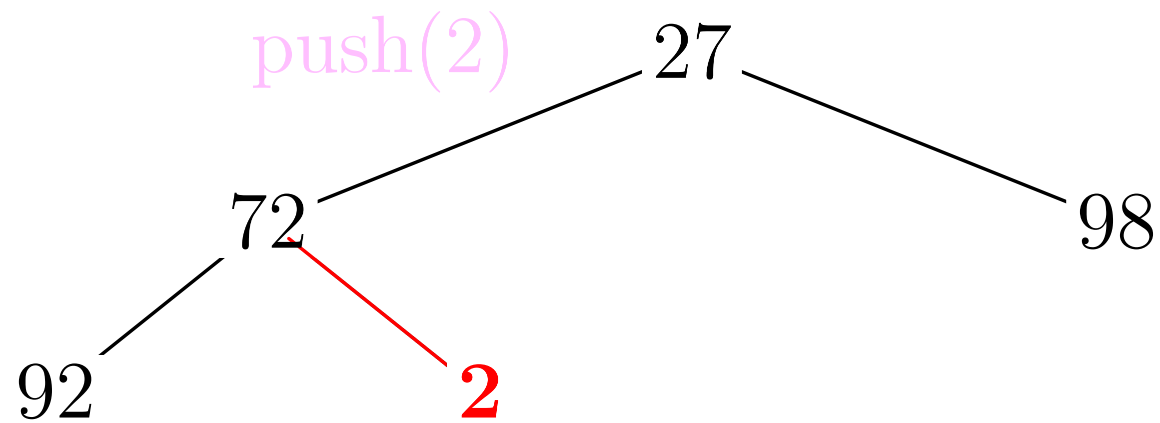
Heaps in Action

27	72	98	92	**2**									

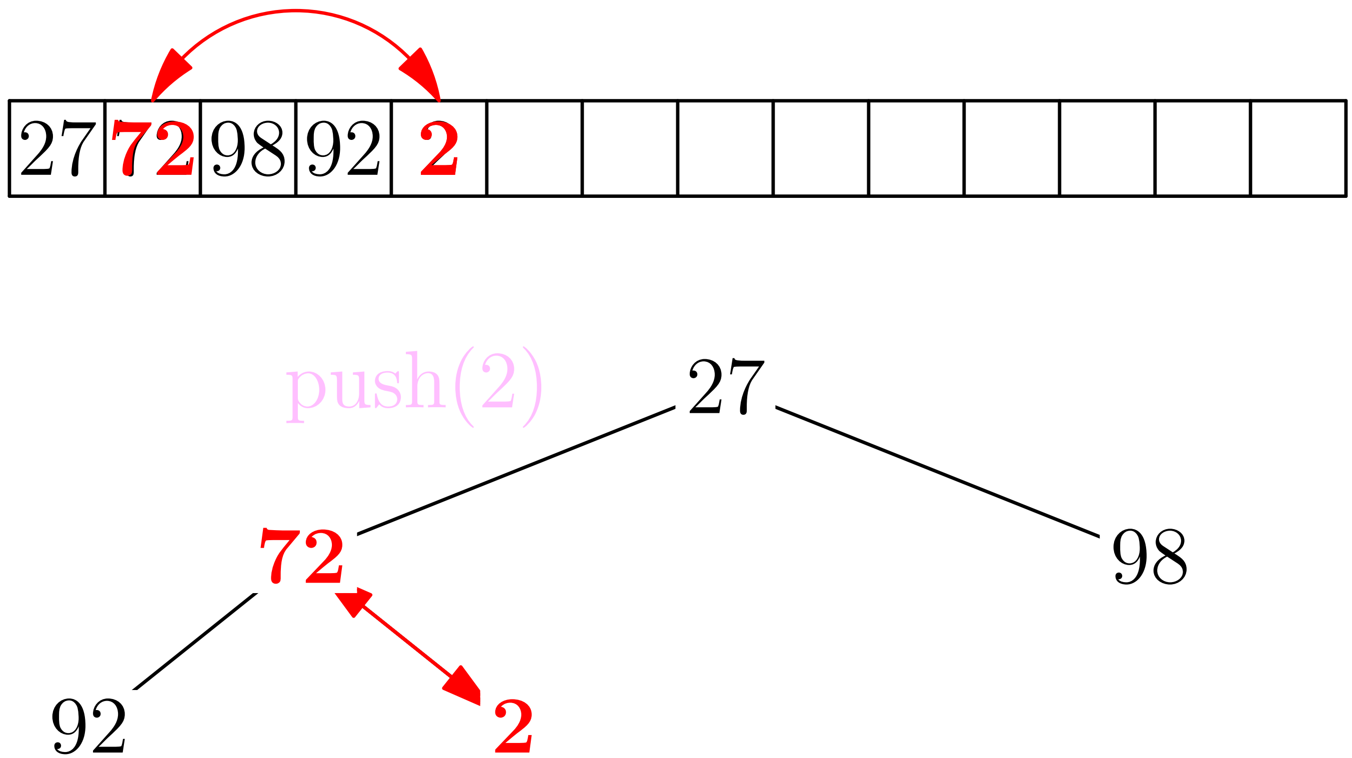


Heaps in Action

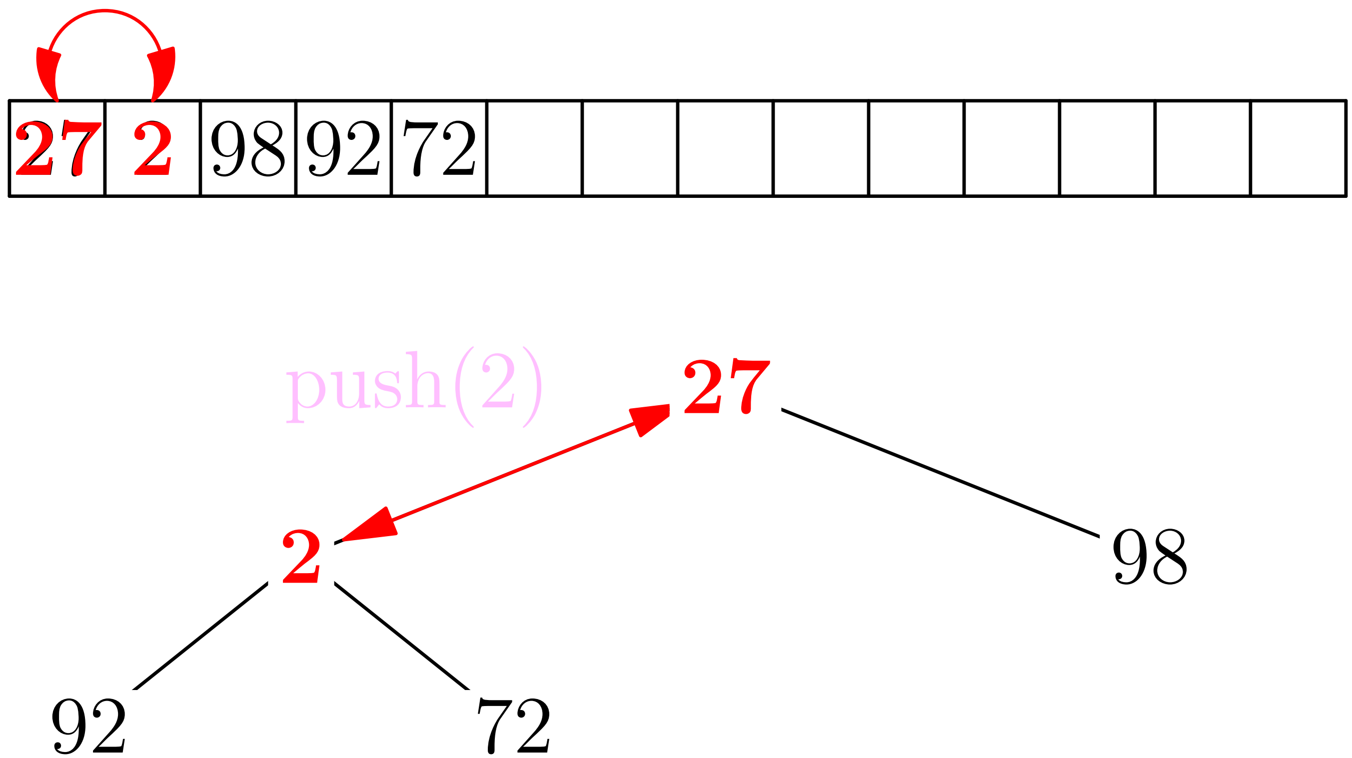
27	72	98	92	**2**									



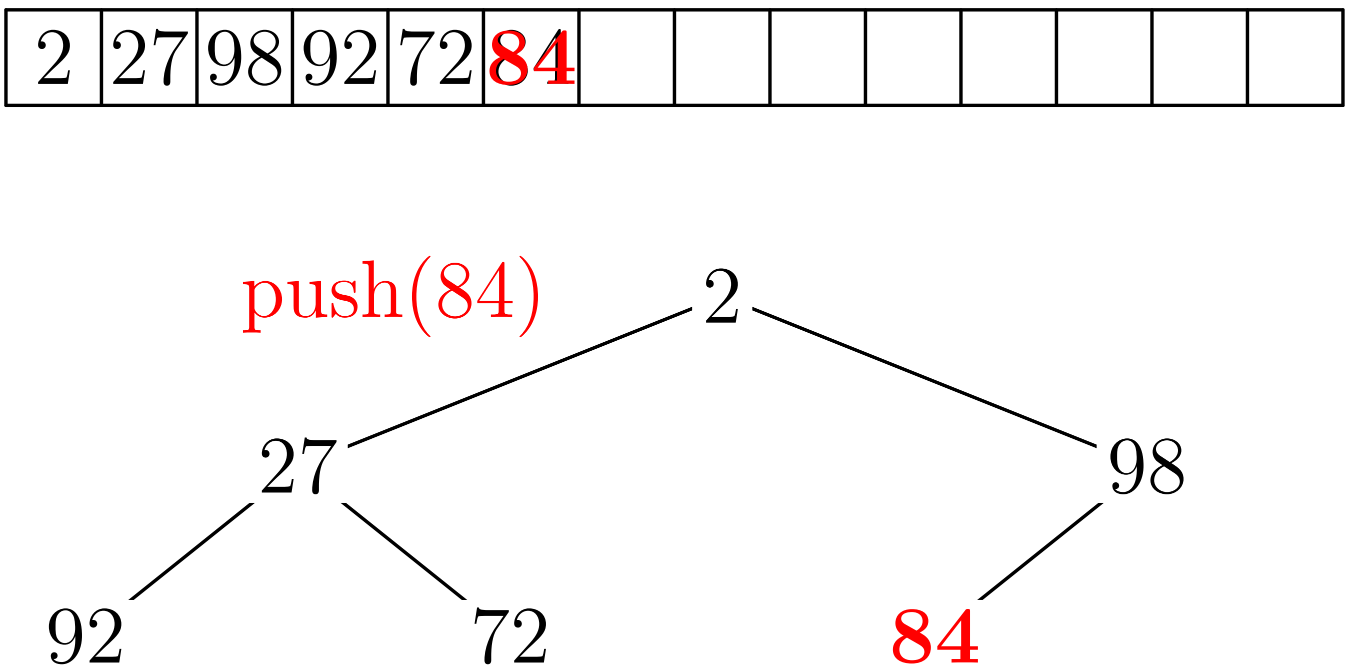
Heaps in Action



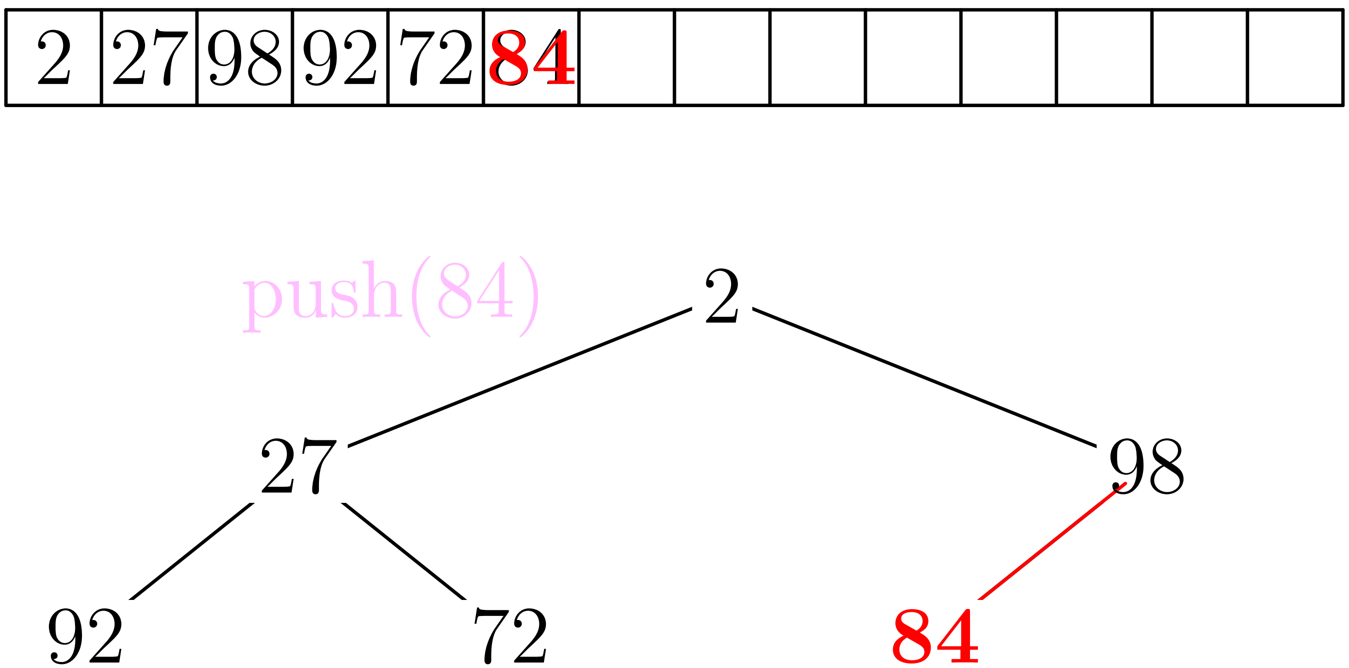
Heaps in Action



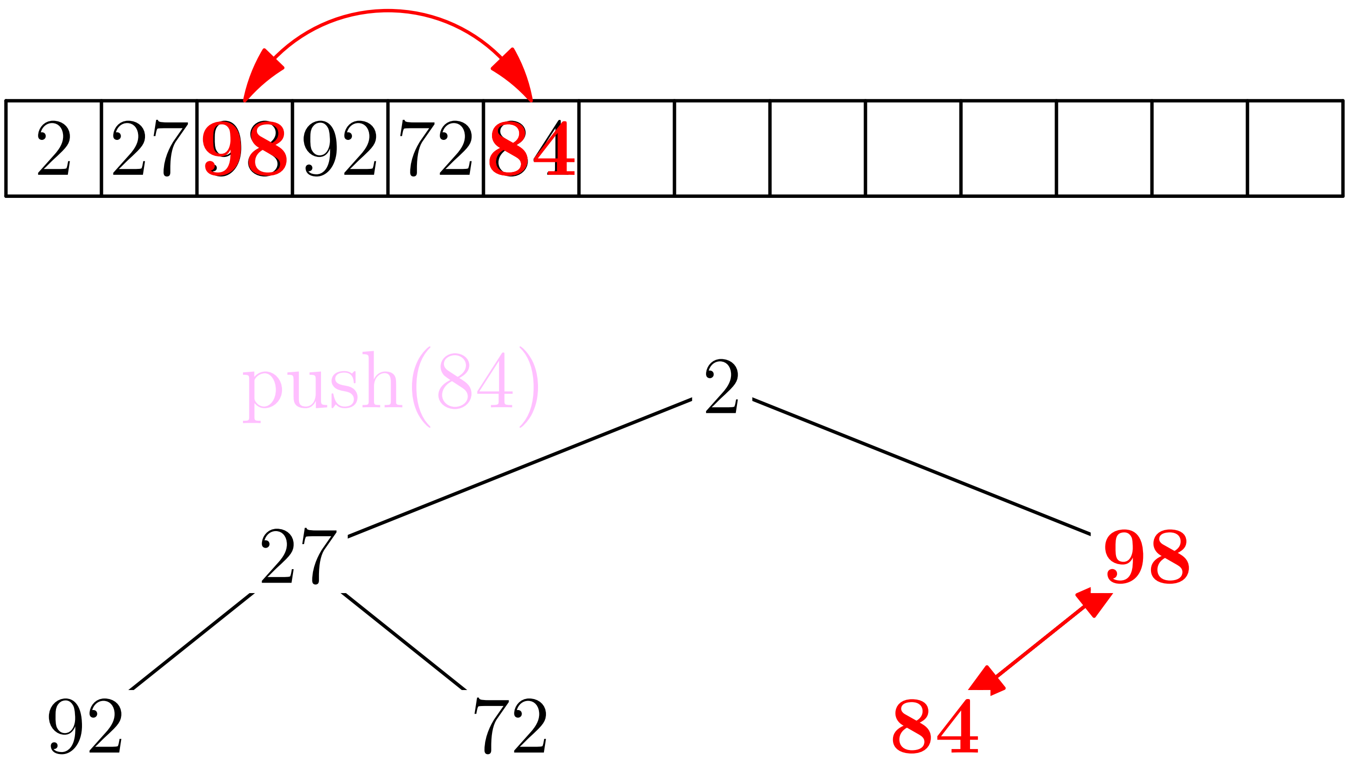
Heaps in Action



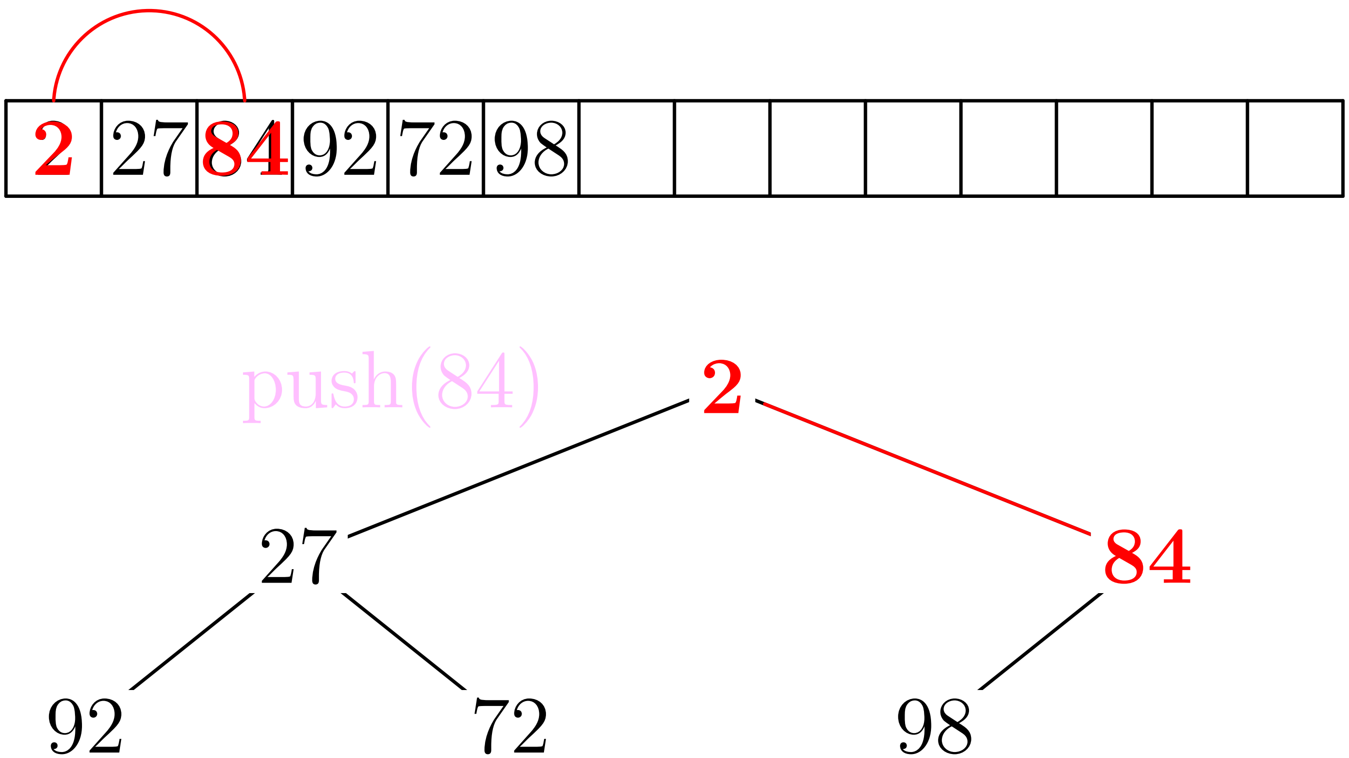
Heaps in Action



Heaps in Action

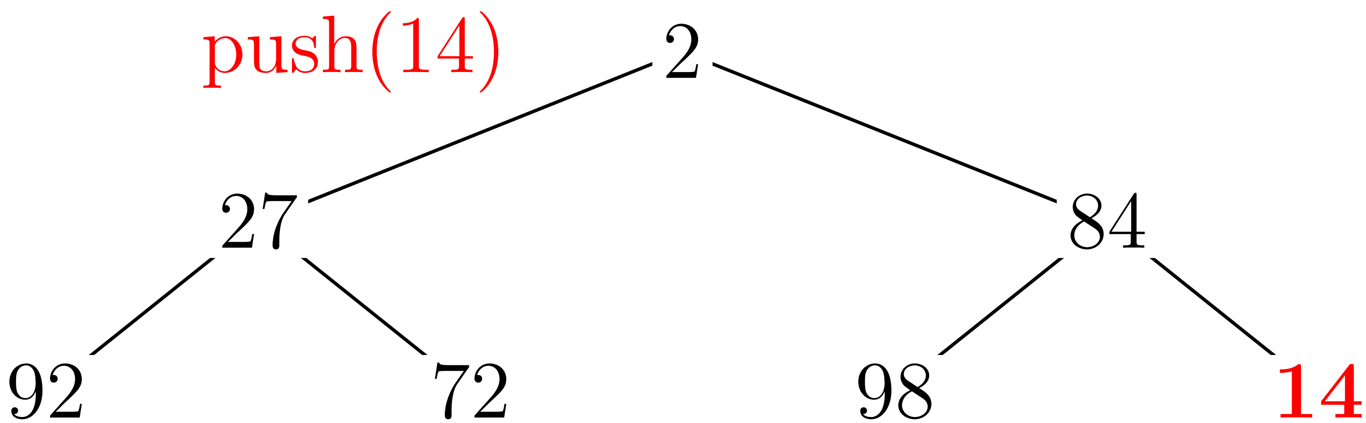


Heaps in Action



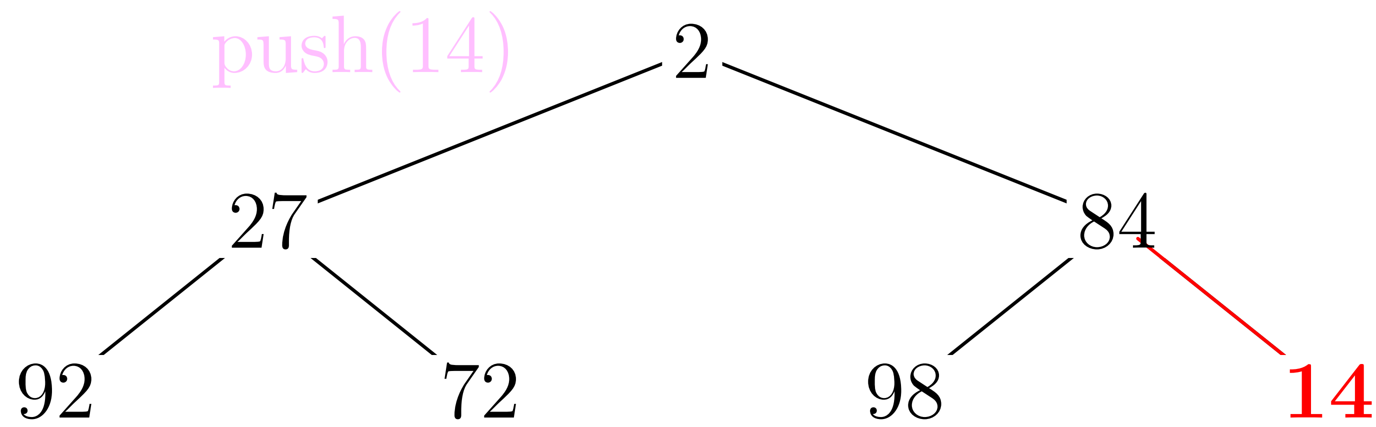
Heaps in Action

2	27	84	92	72	98	**14**							

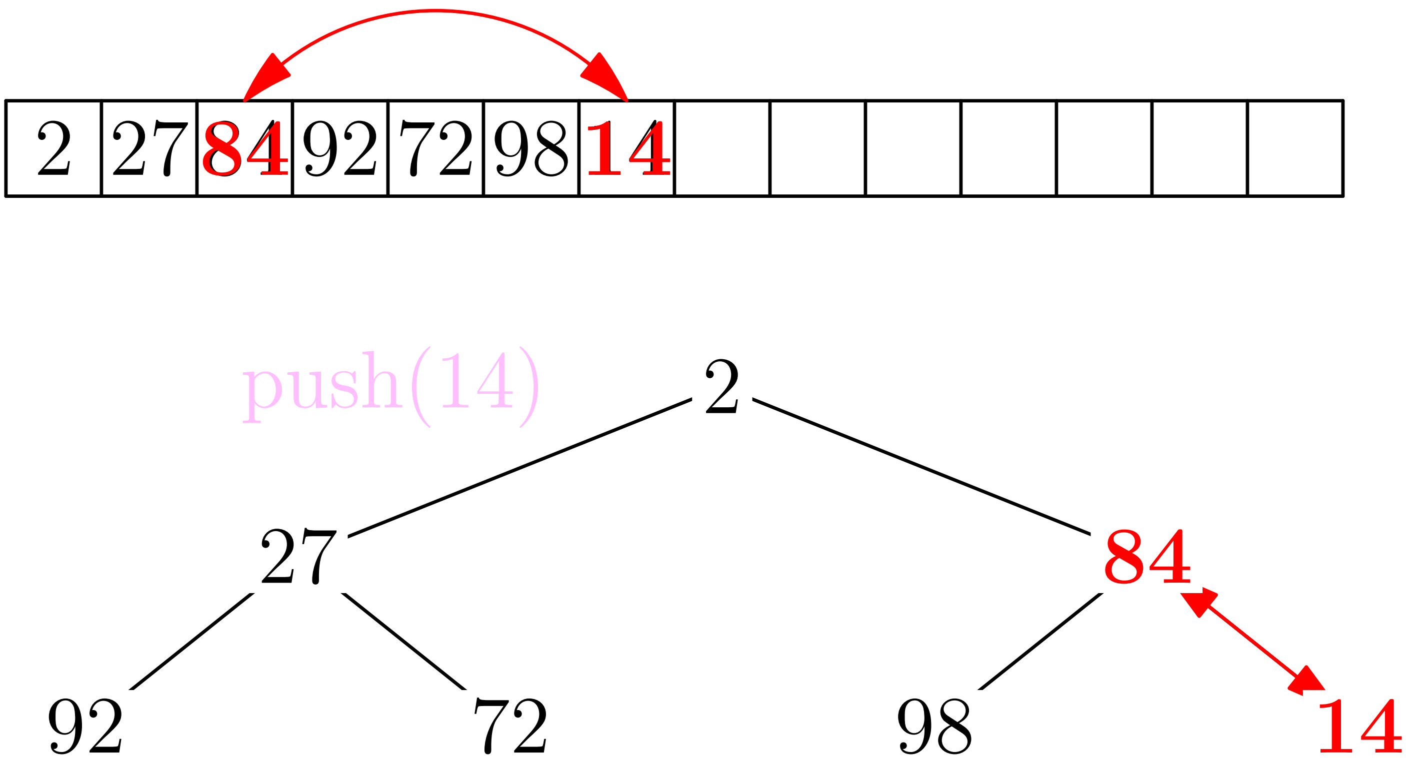


Heaps in Action

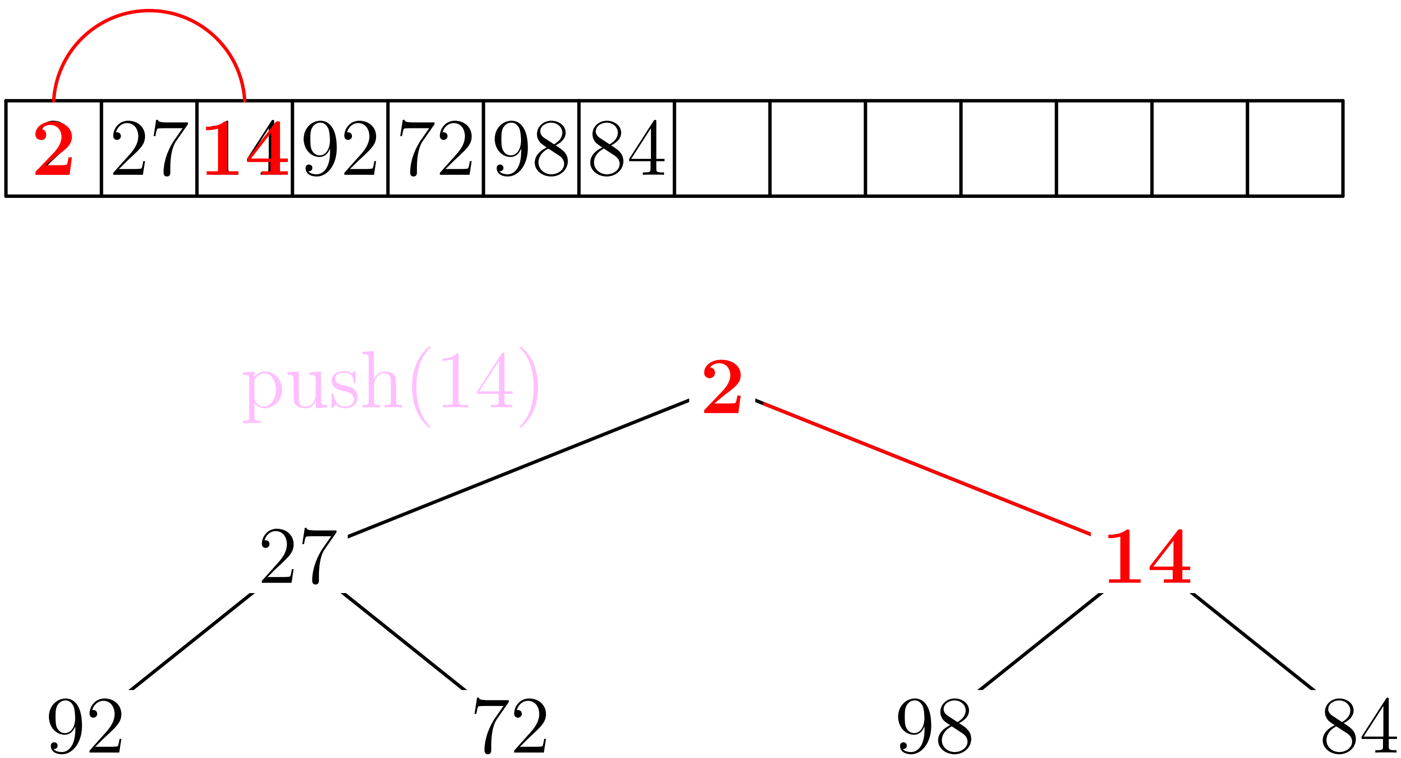
2	27	84	92	72	98	**14**							



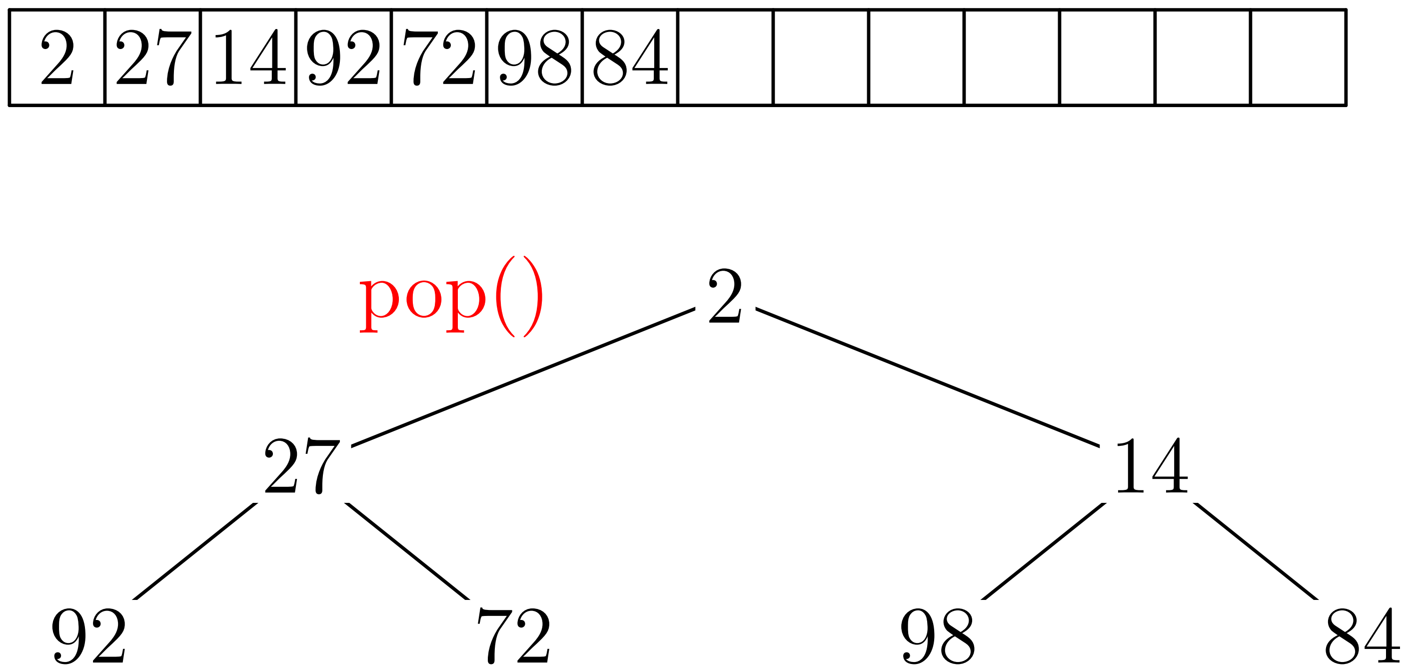
Heaps in Action



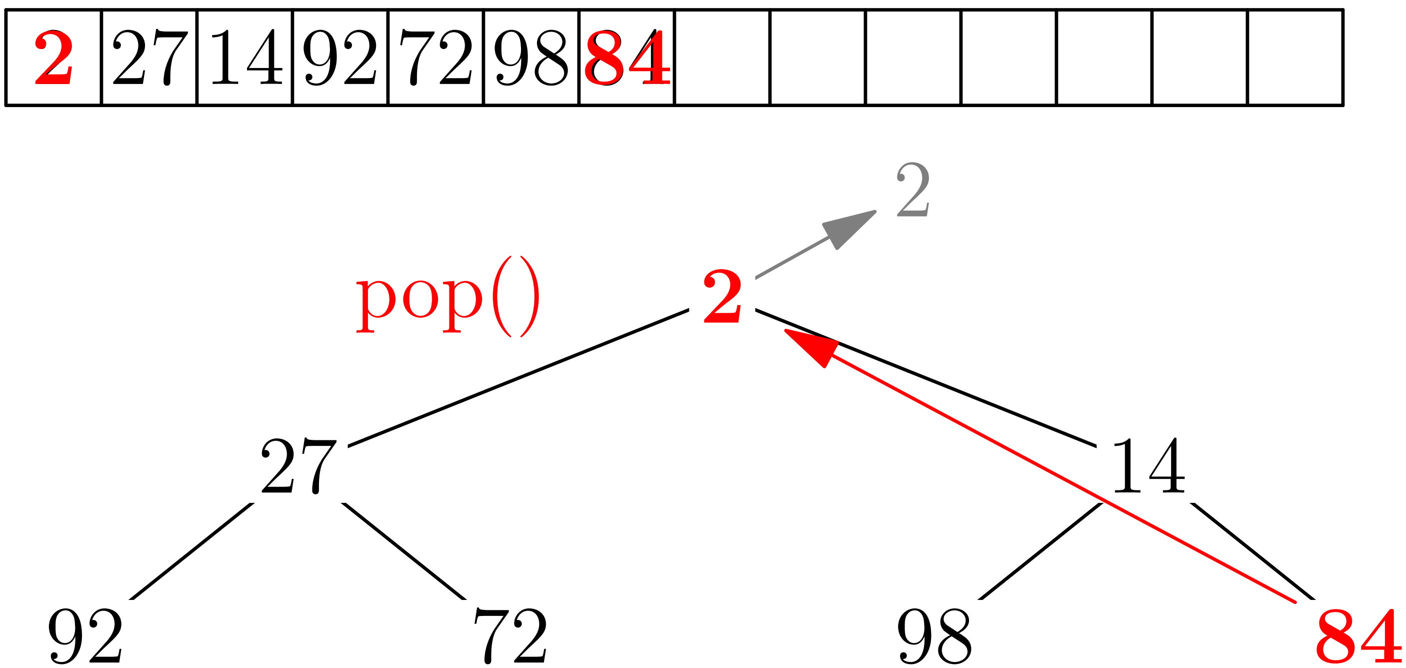
Heaps in Action



Heaps in Action

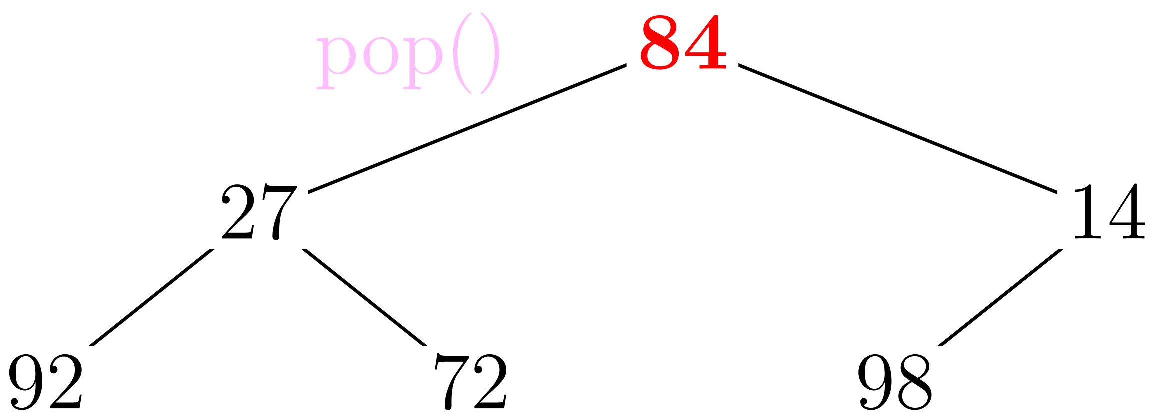


Heaps in Action

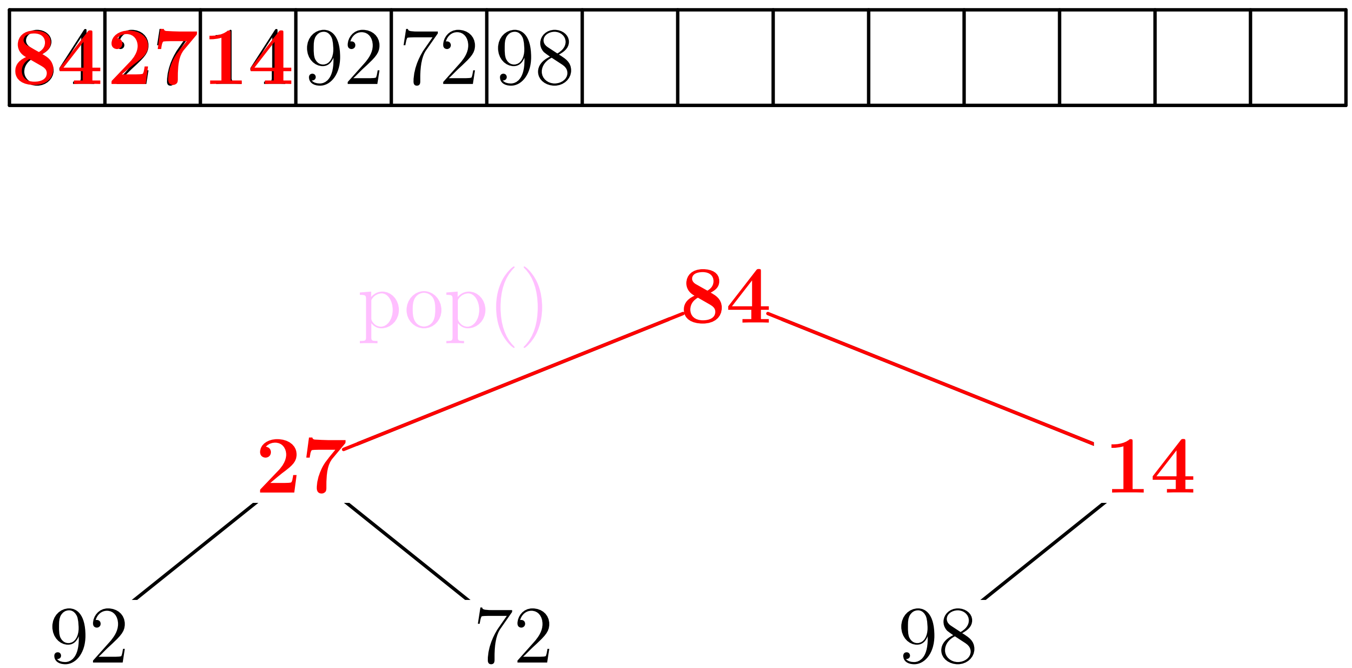


Heaps in Action

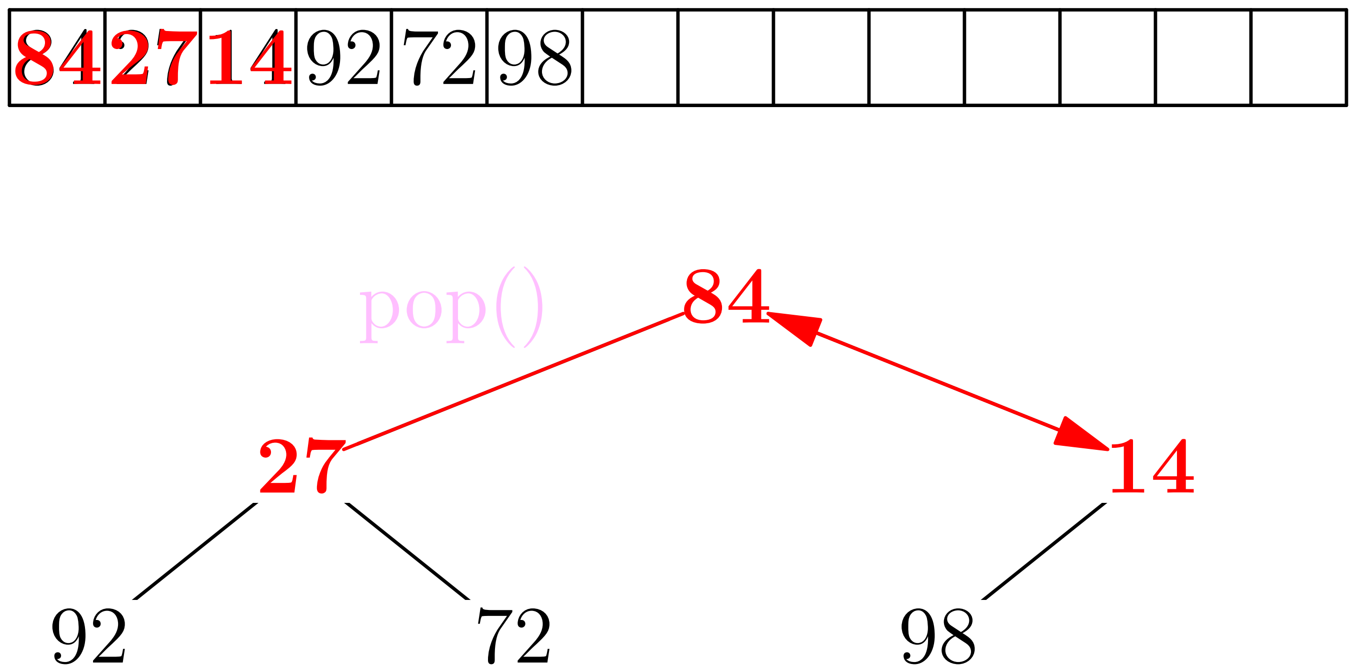
84	27	14	92	72	98								



Heaps in Action

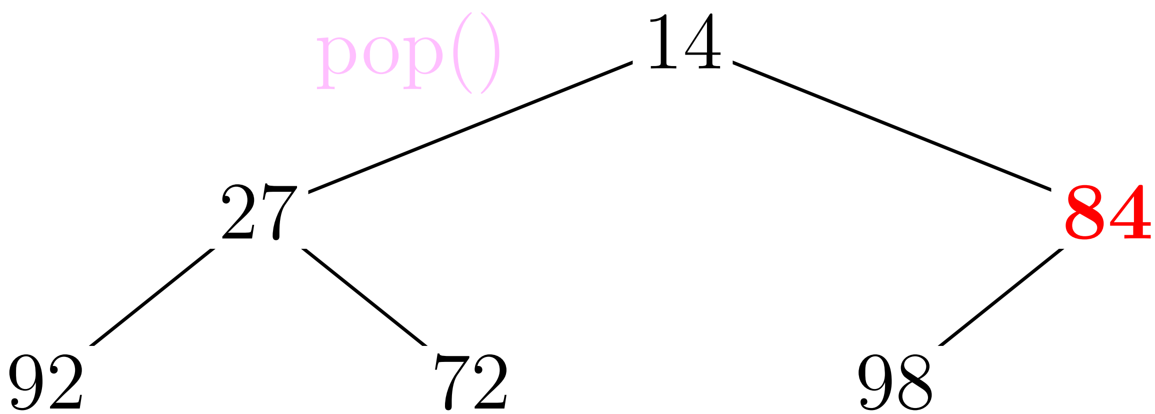


Heaps in Action



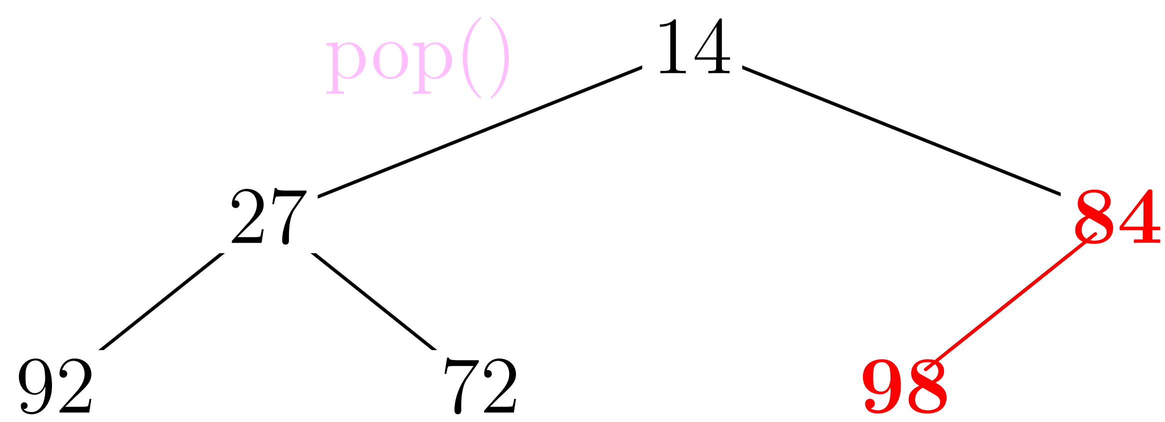
Heaps in Action

14	27	**84**	92	72	98								



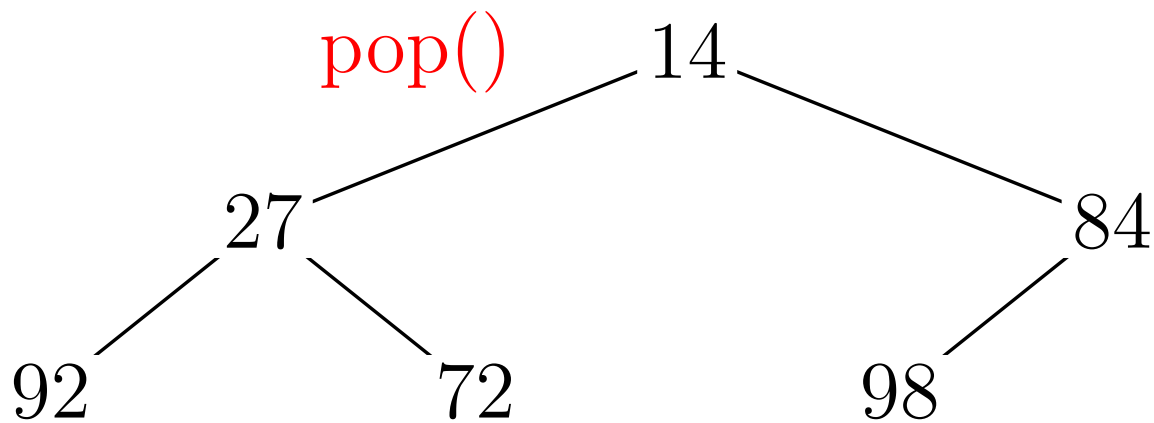
Heaps in Action

14	27	**84**	92	72	**98**								

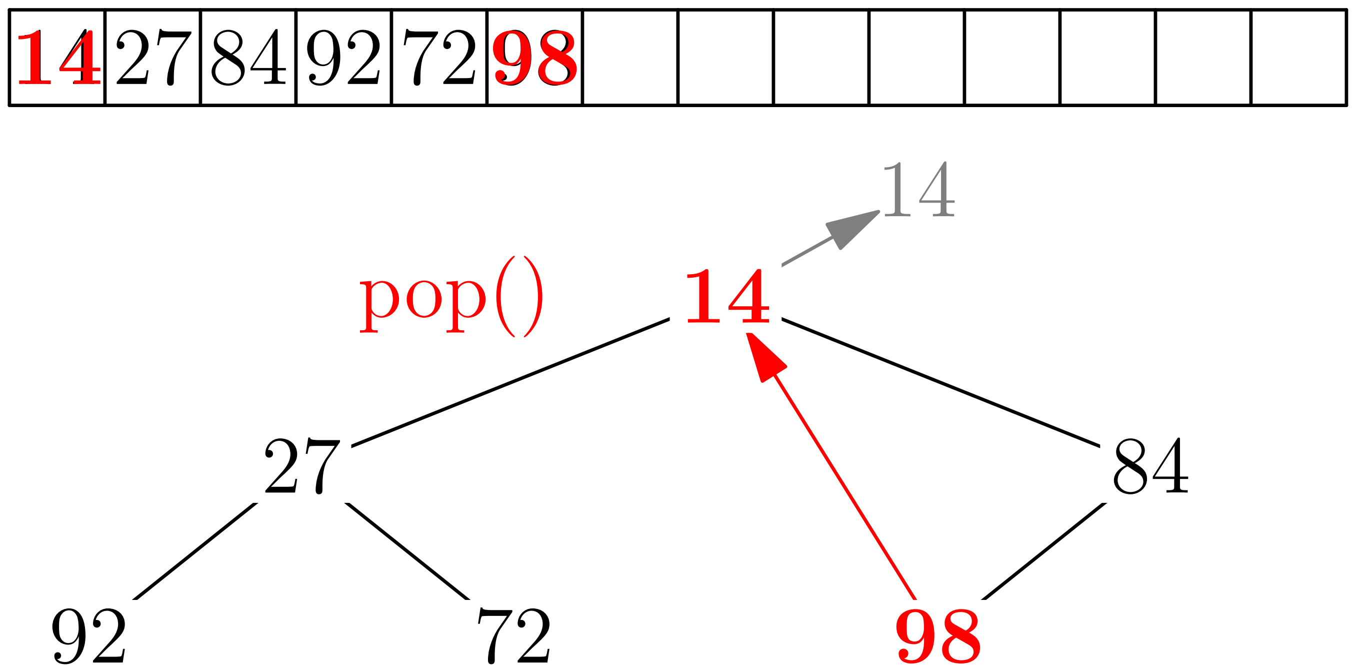


Heaps in Action

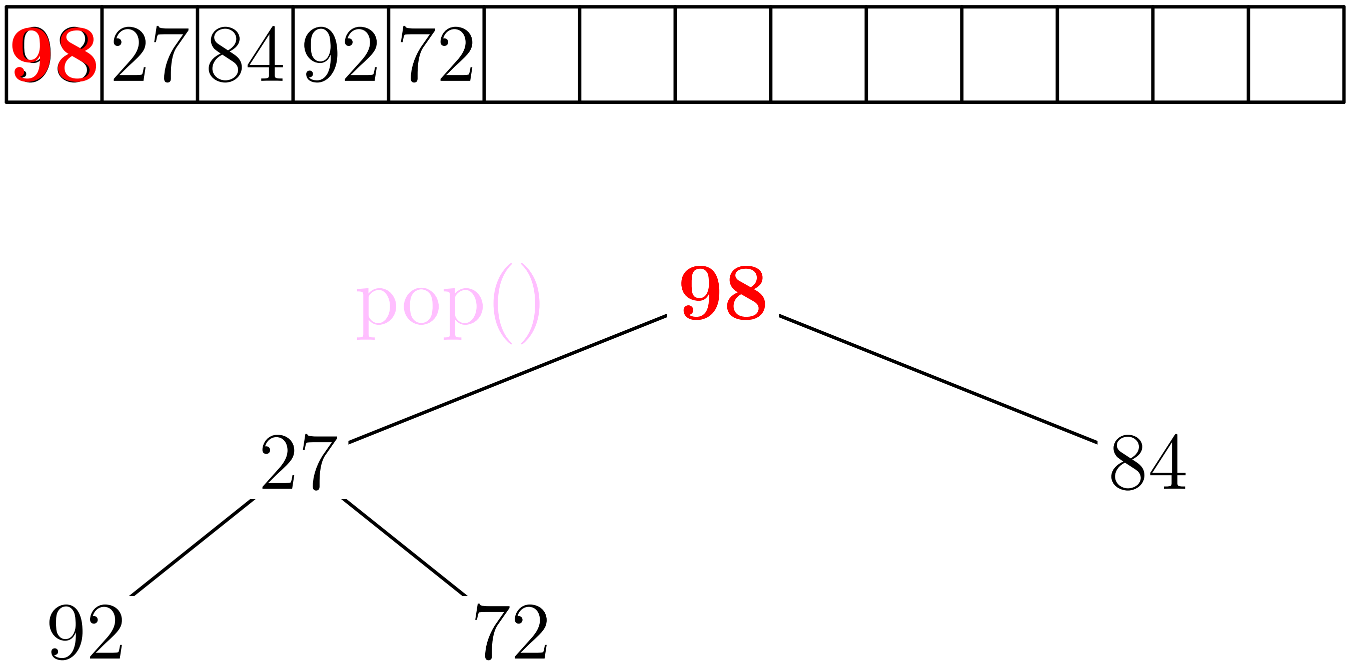
14	27	84	92	72	98								



Heaps in Action

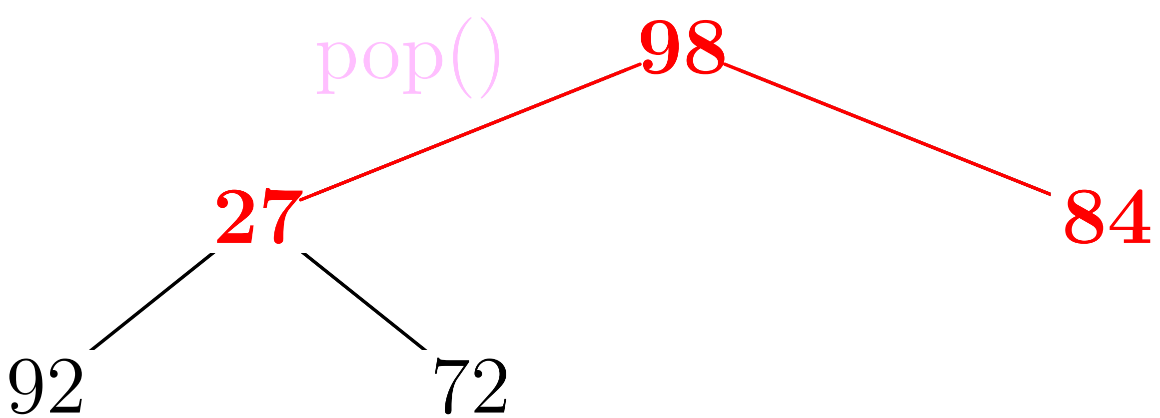


Heaps in Action

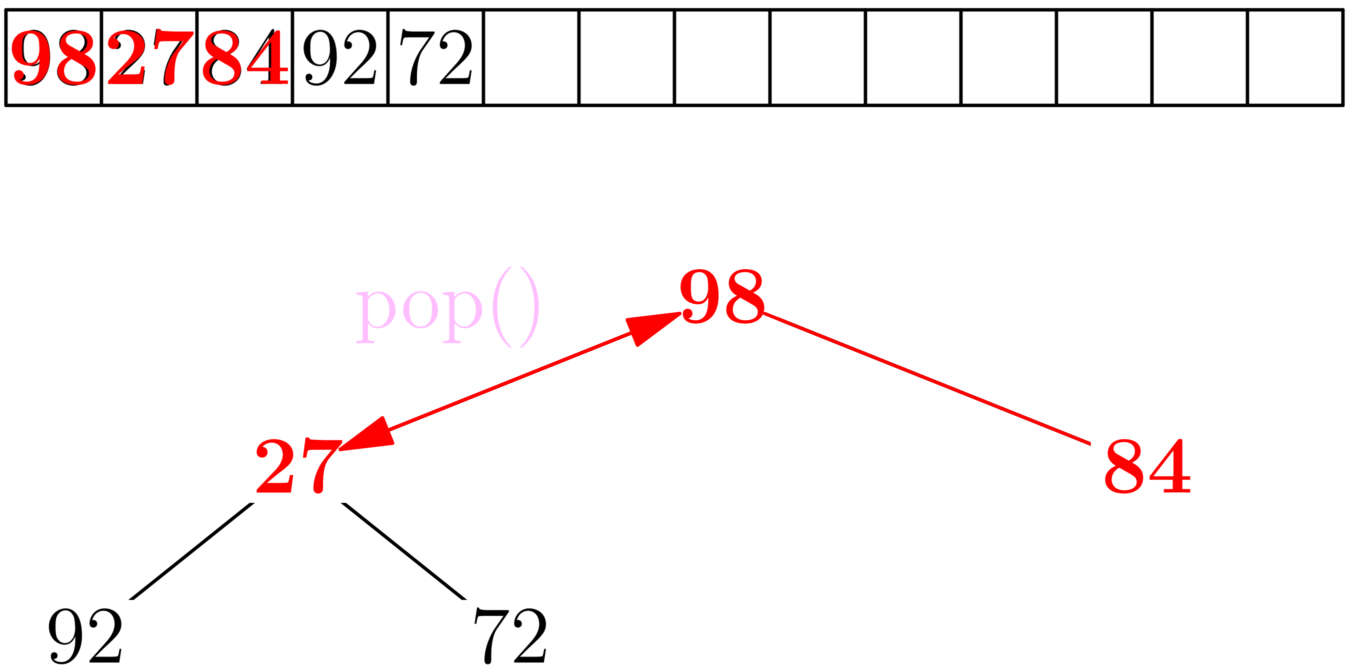


Heaps in Action

98	27	84	92	72									

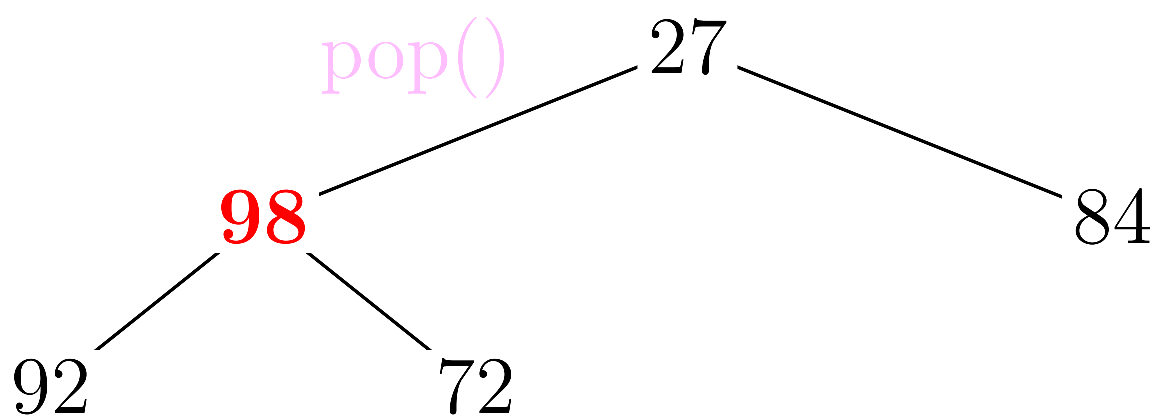


Heads in Action



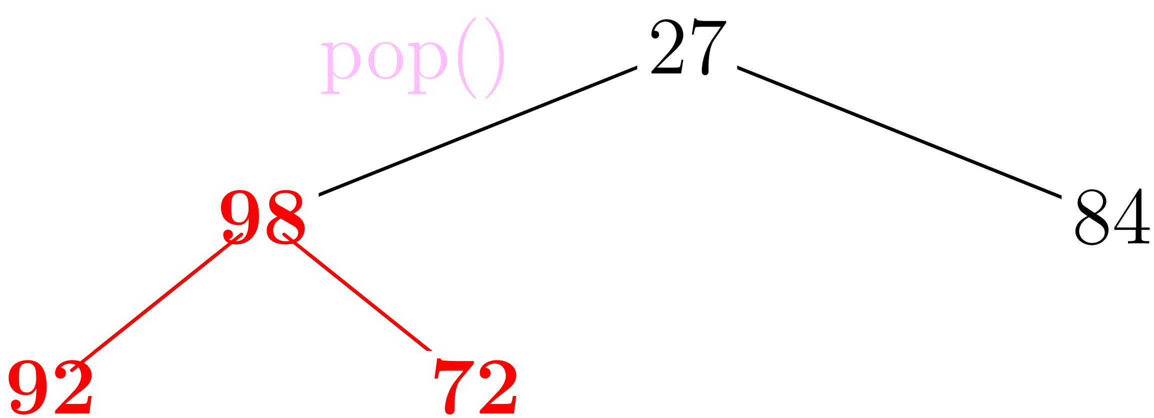
Heads in Action

27	**98**	84	92	72									

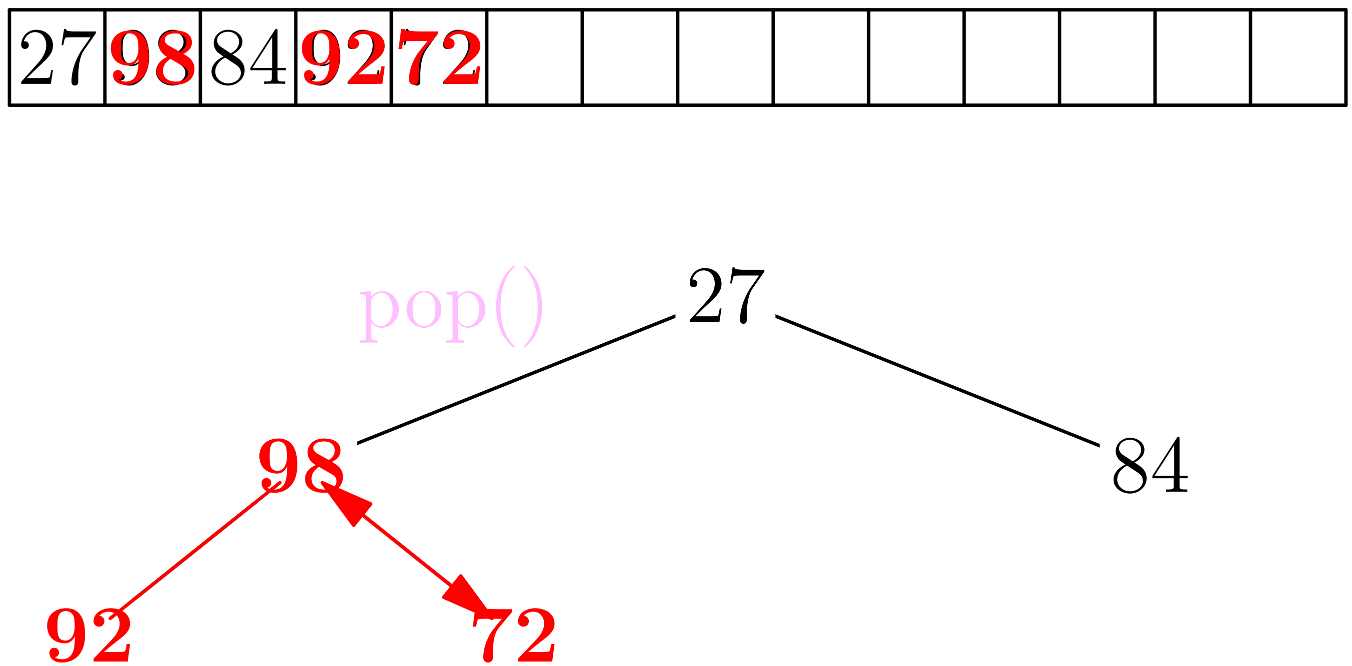


Heaps in Action

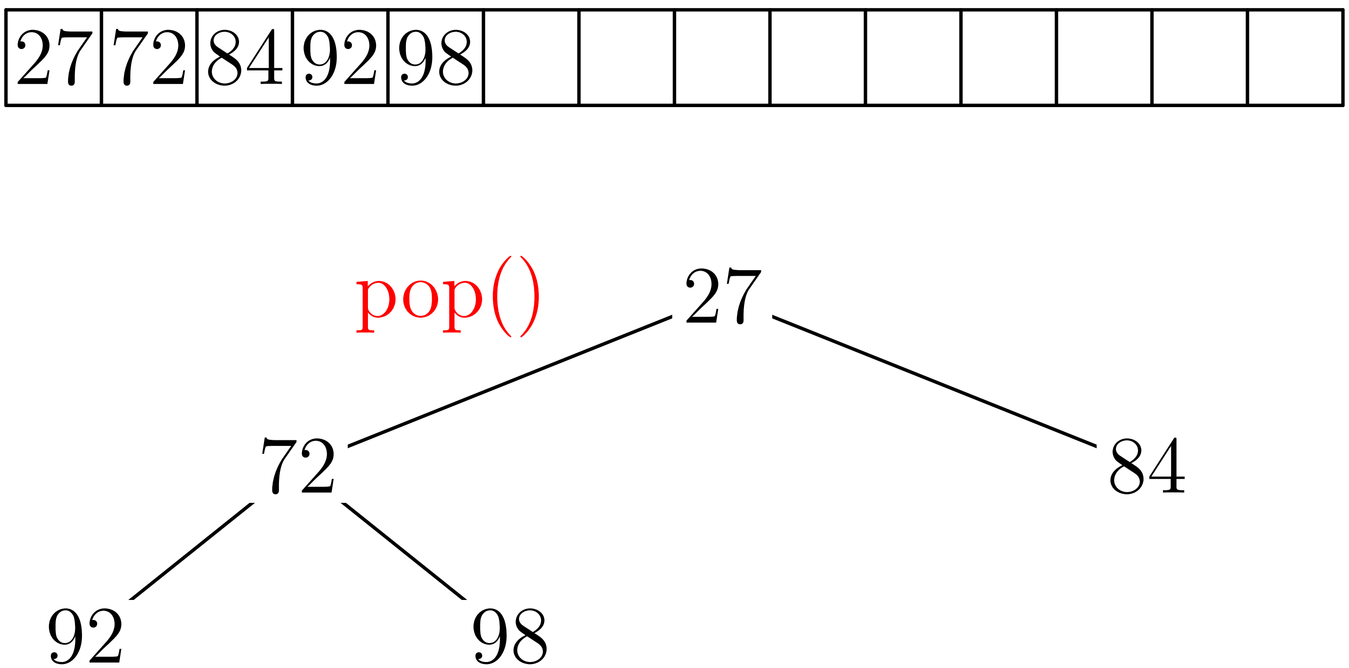
27	**98**	84	**92**	**72**									



Heaps in Action

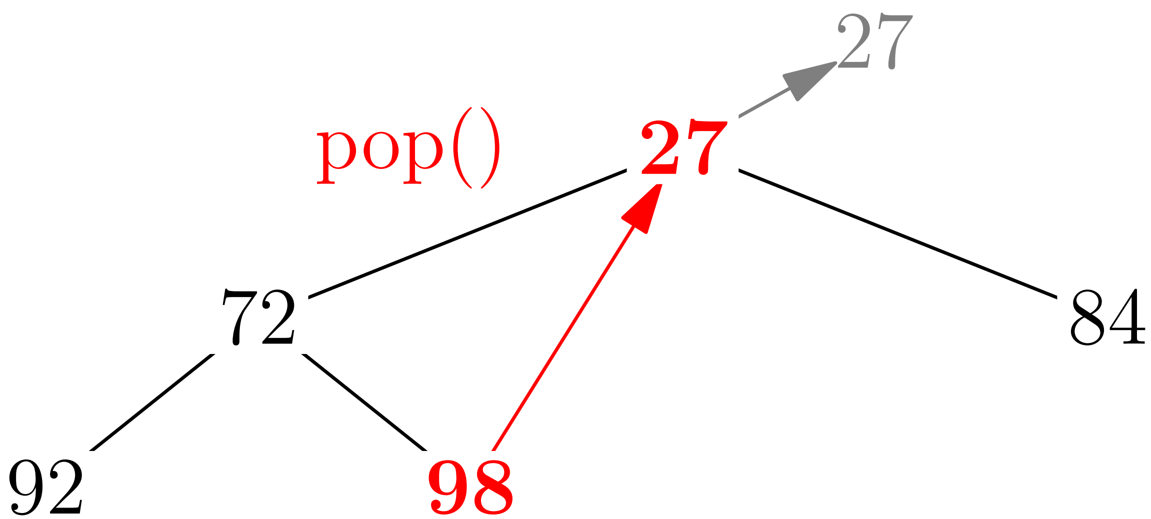


Heaps in Action



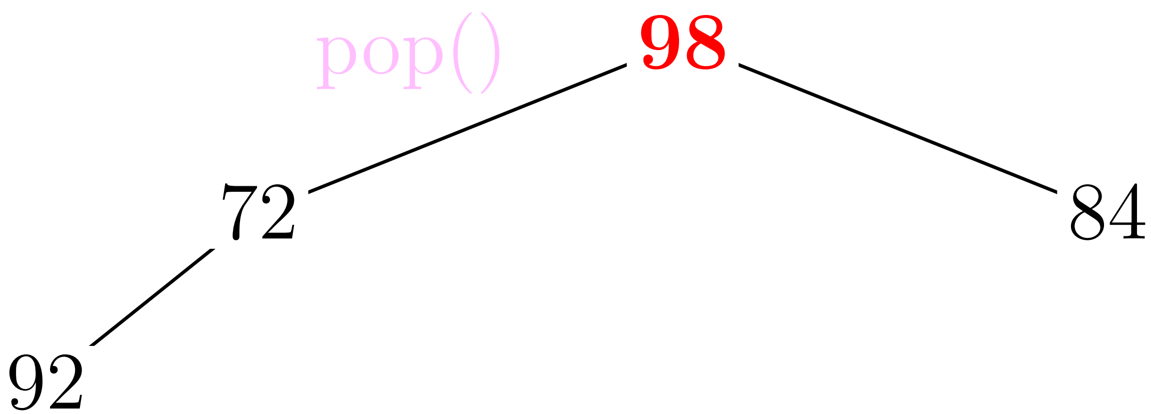
Heaps in Action

27	72	84	92	98									

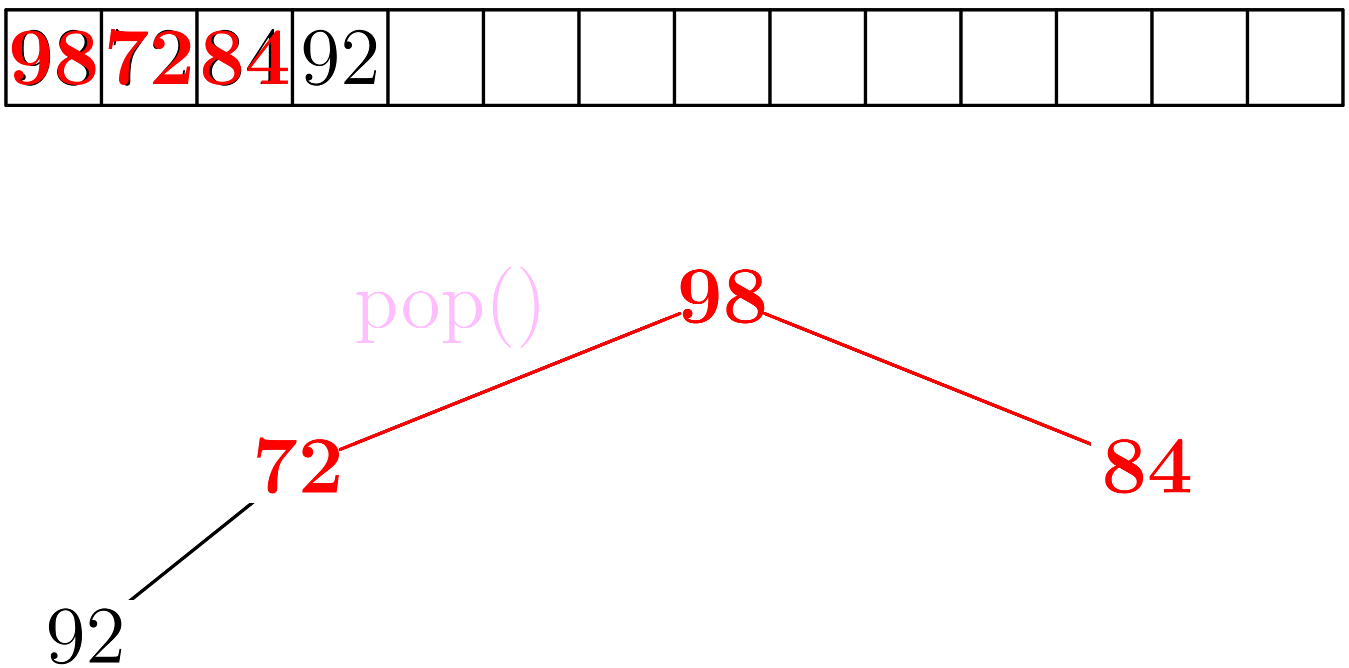


Heaps in Action

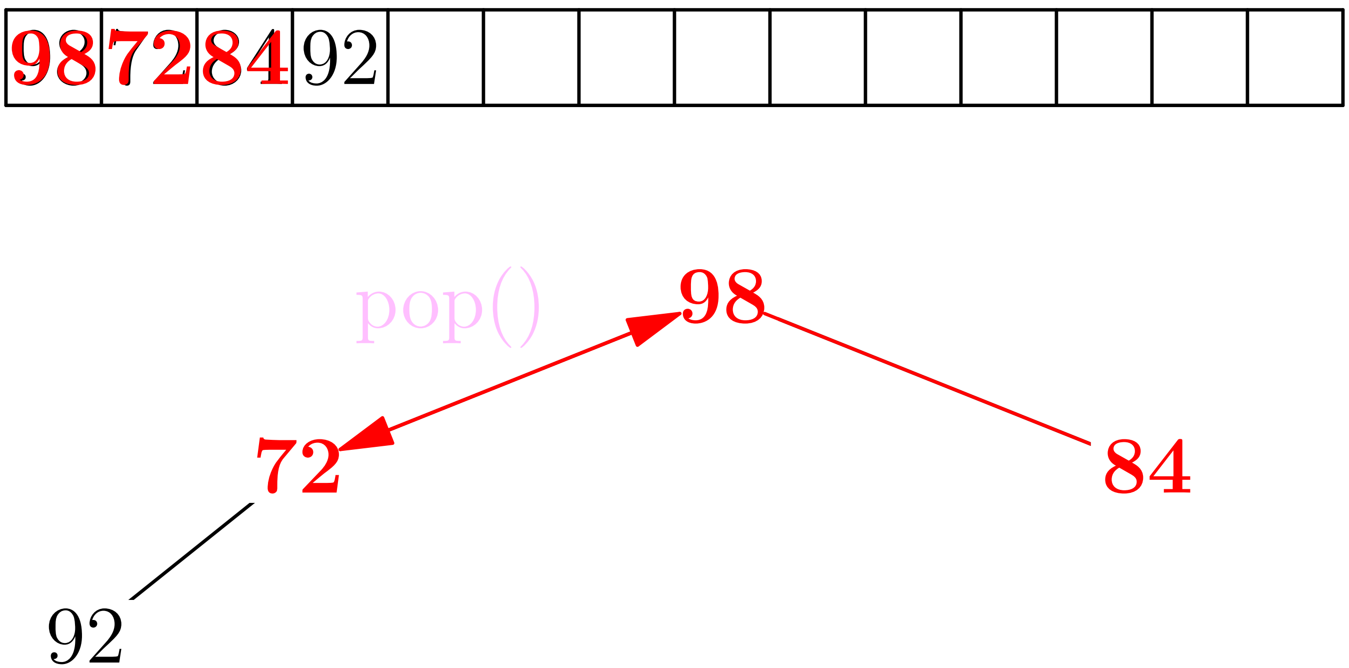
98	72	84	92										



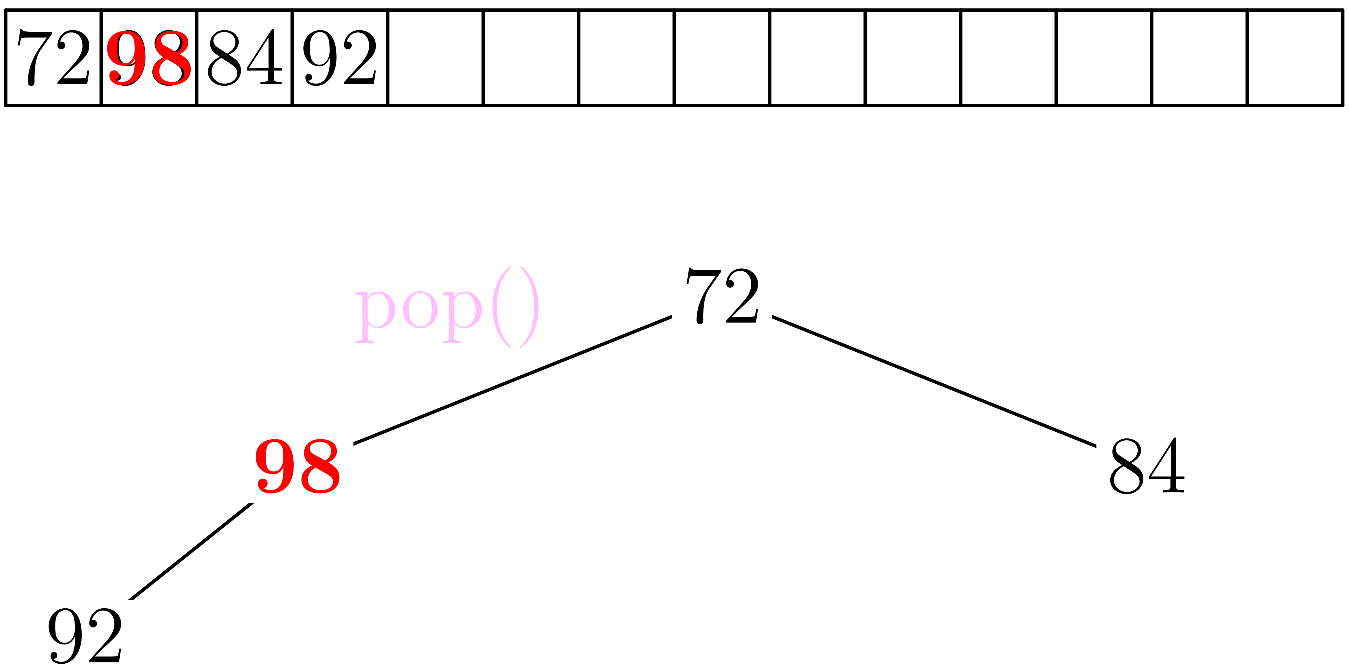
Heaps in Action



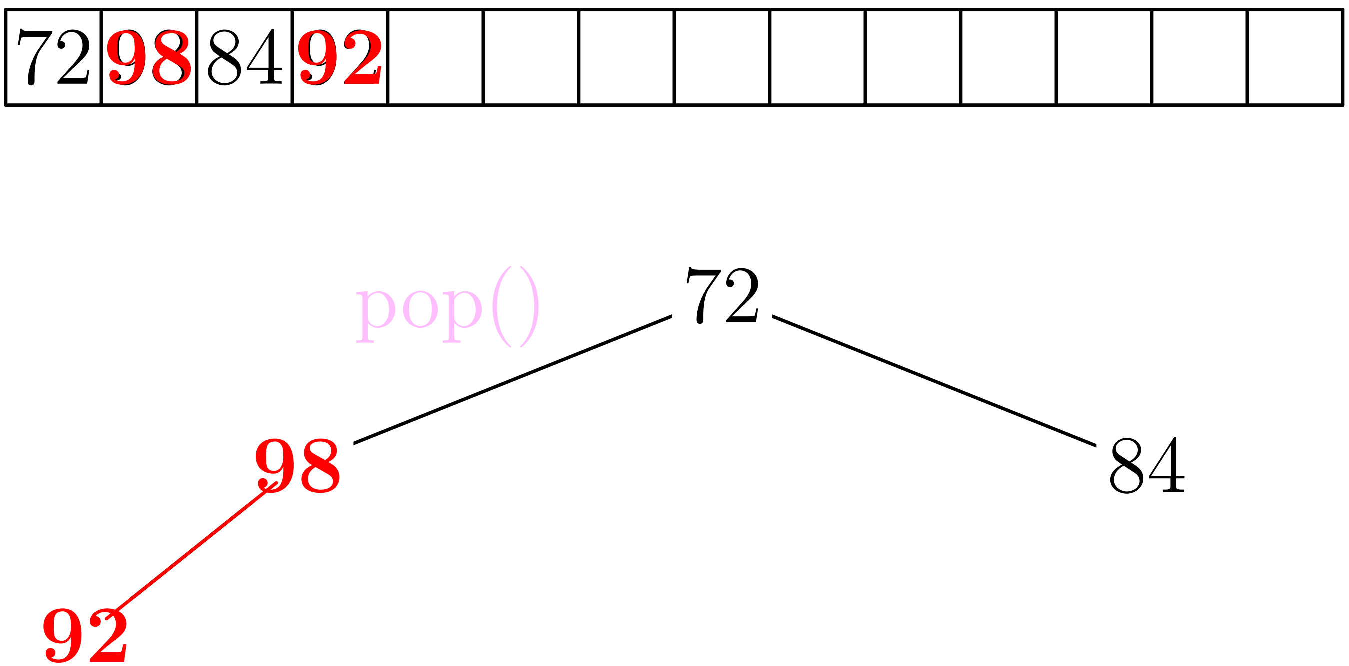
Heaps in Action



Heaps in Action

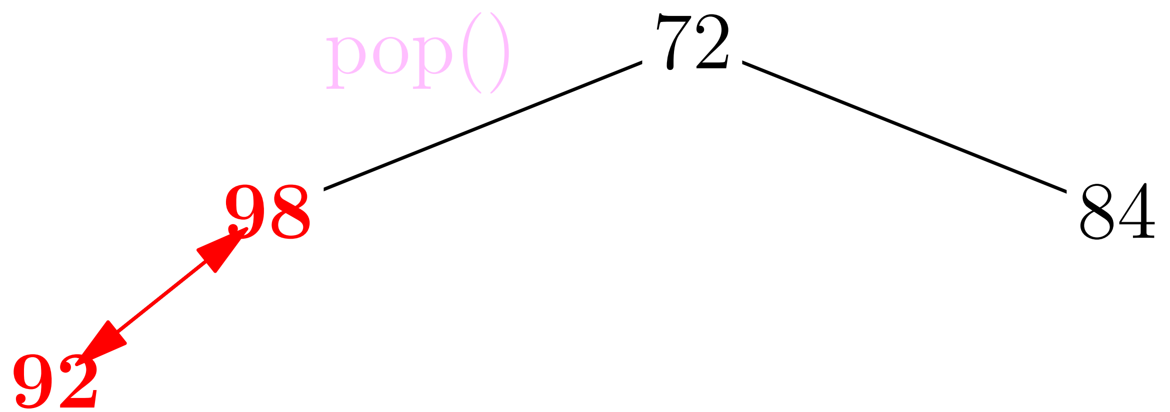


Heaps in Action



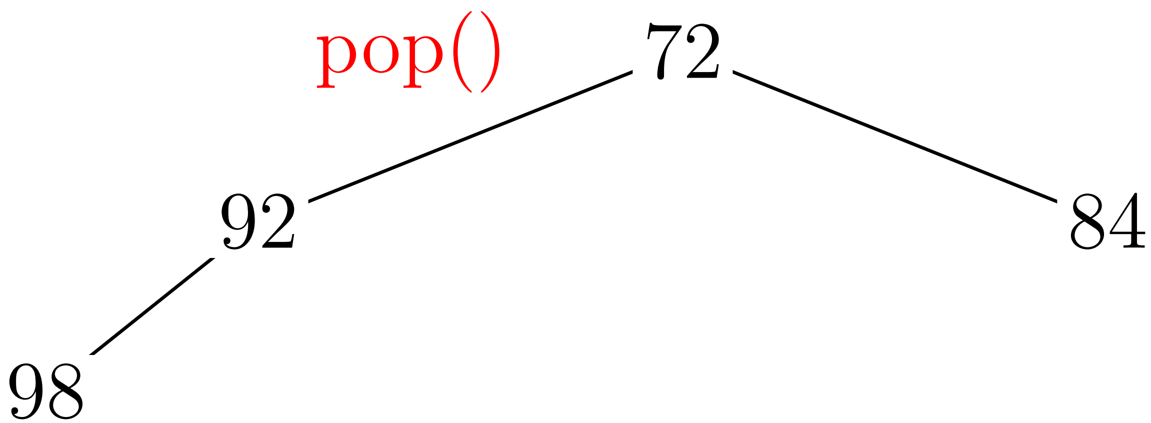
Heaps in Action

72	**98**	84	**92**										

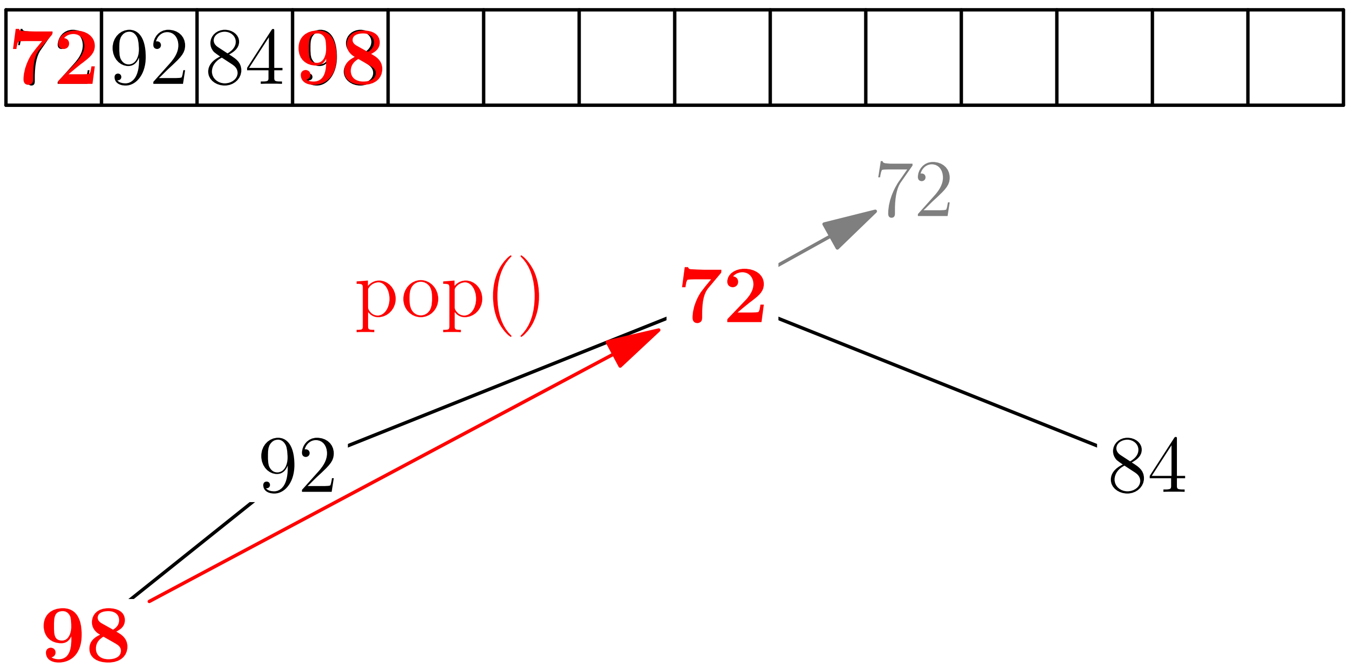


Heads in Action

72	92	84	98										

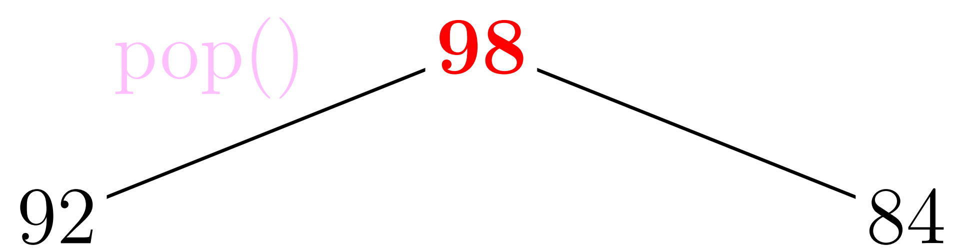


Heaps in Action

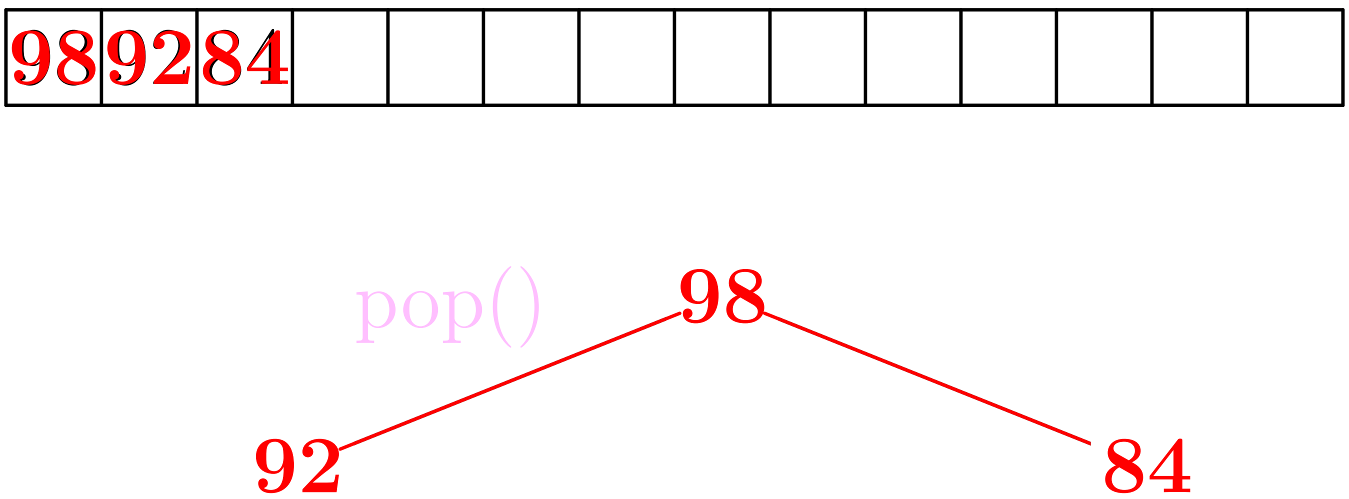


Heaps in Action

98	92	84											

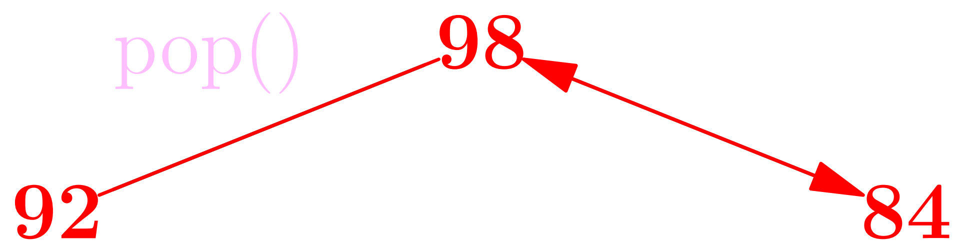


Heaps in Action

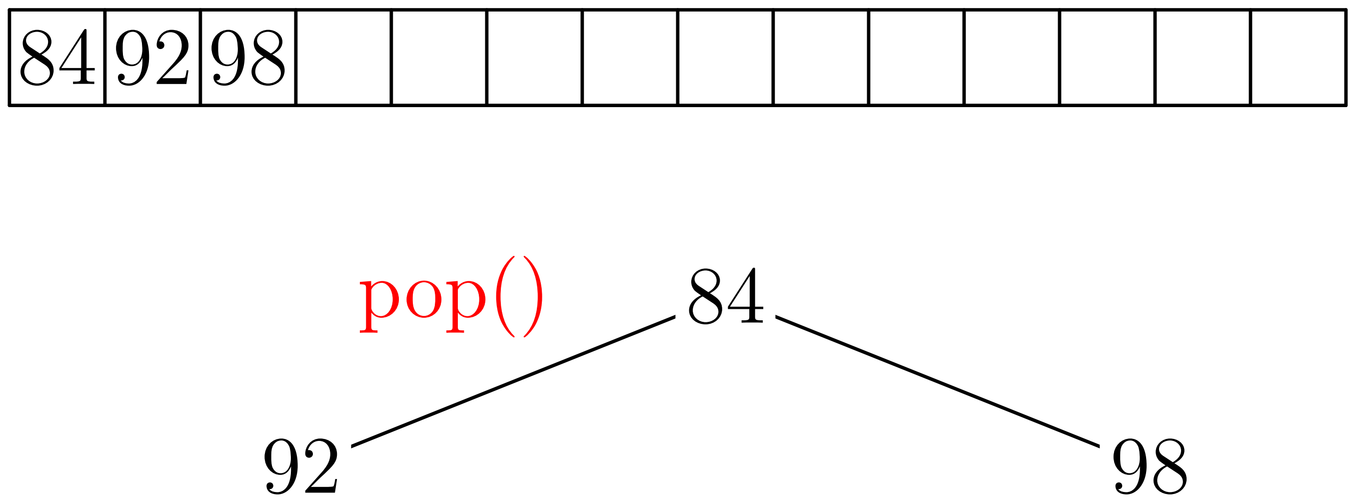


Heaps in Action

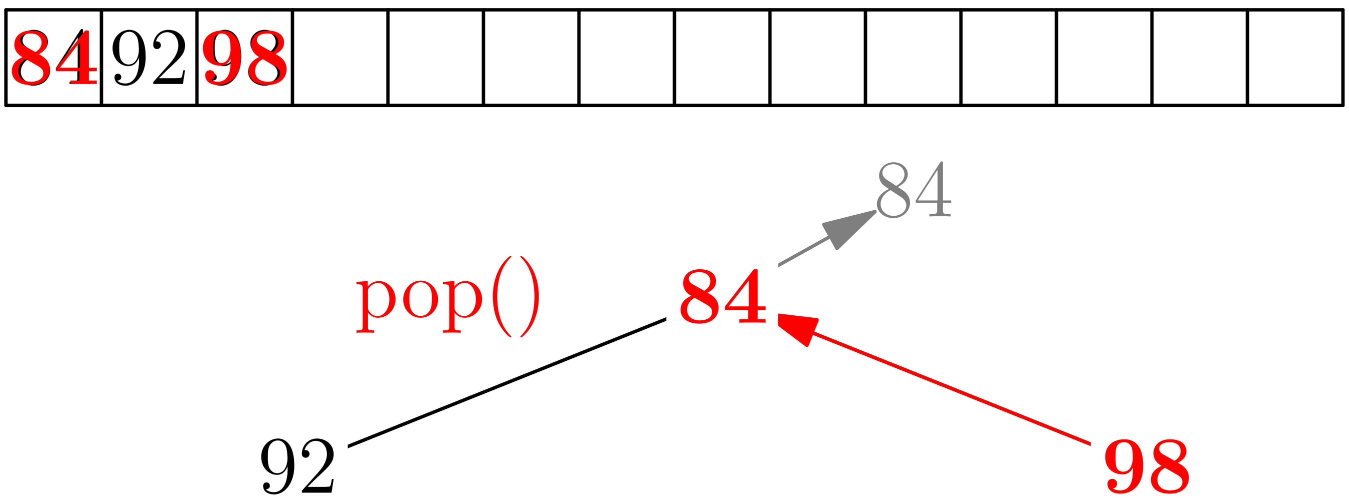
98	92	84											



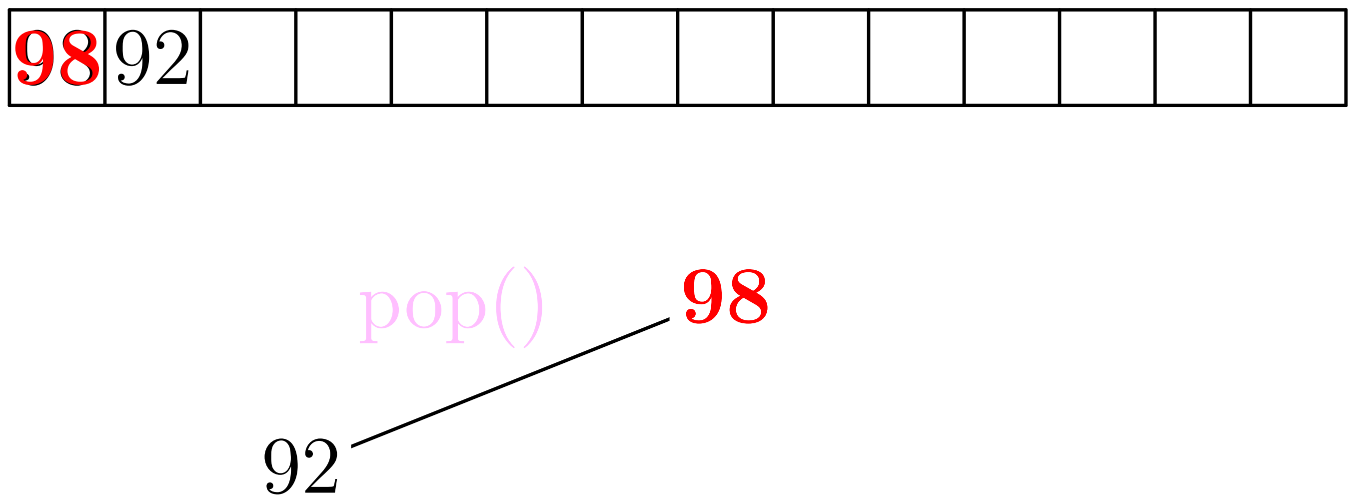
Heaps in Action



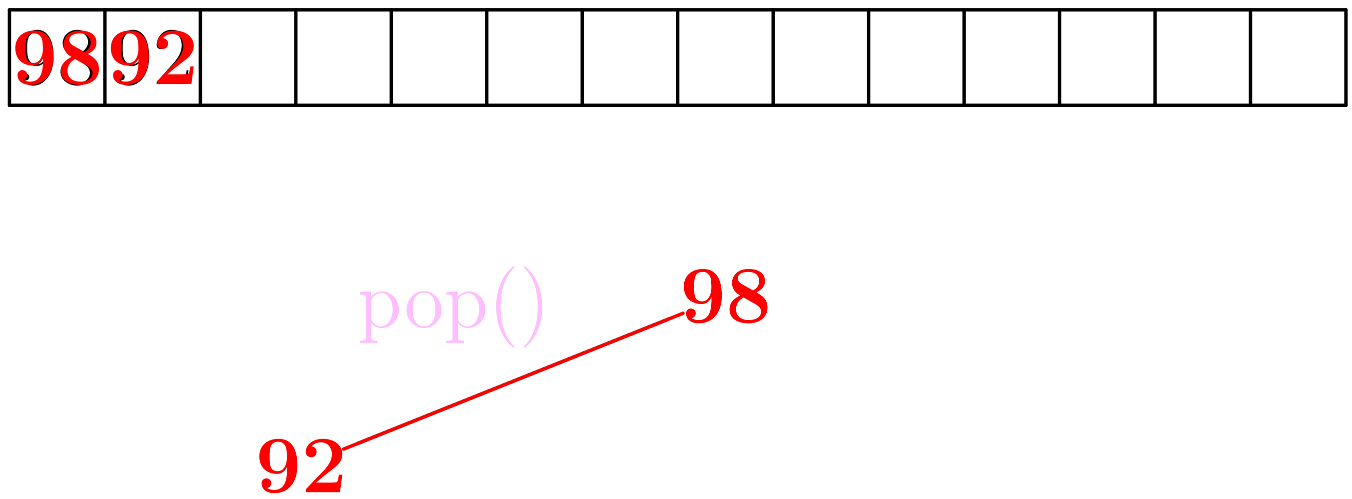
Heaps in Action



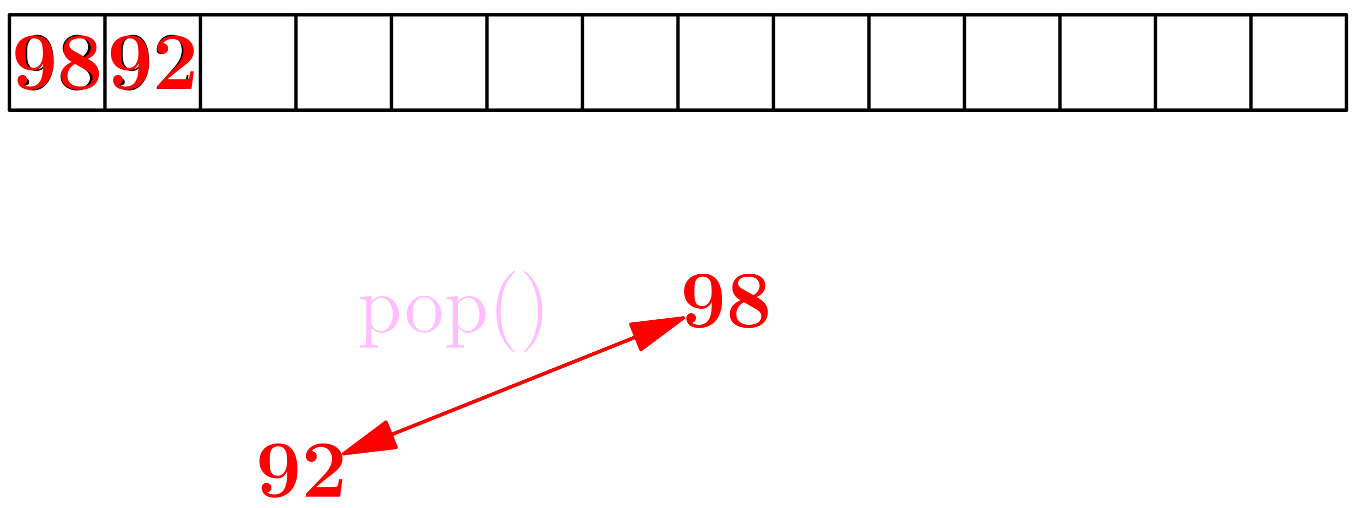
Heaps in Action



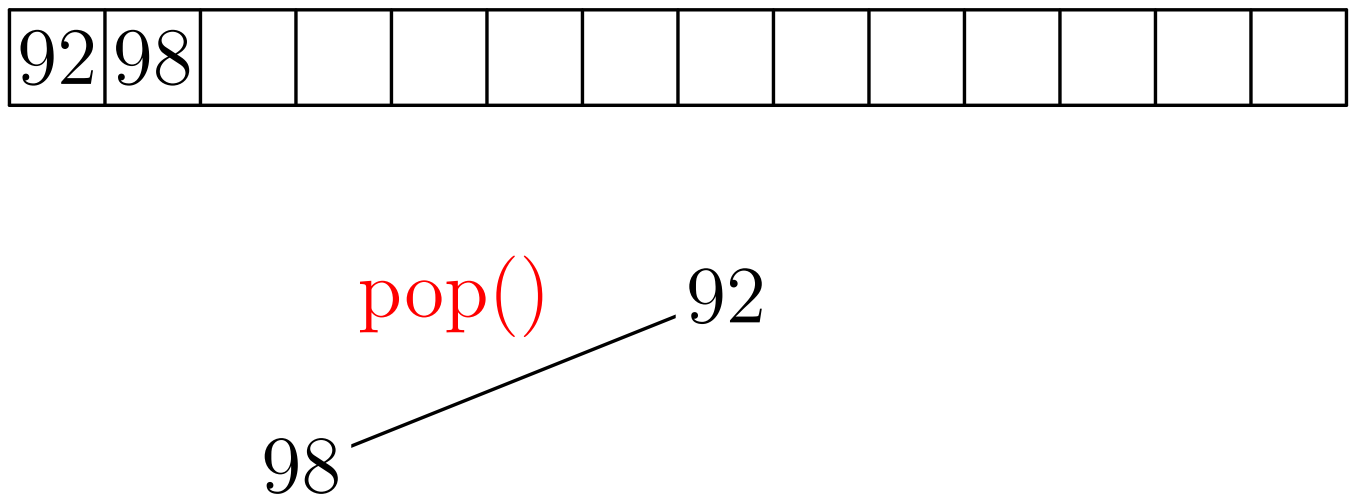
Heaps in Action



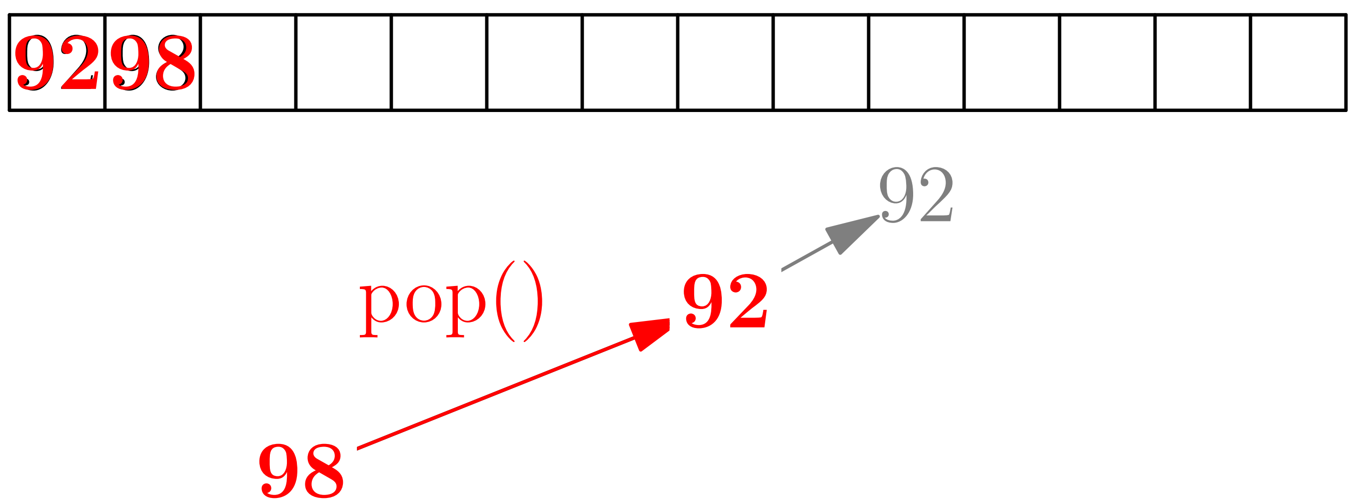
Heaps in Action



Heaps in Action



Heaps in Action

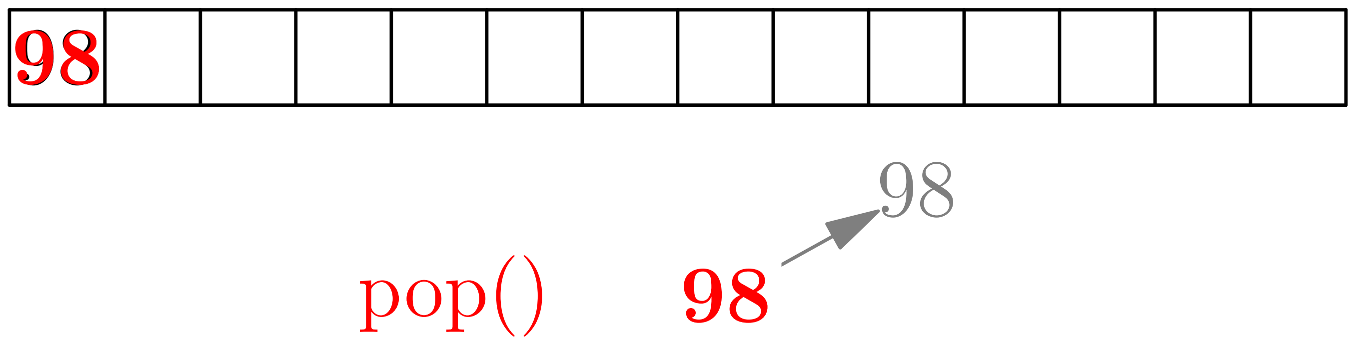


Heaps in Action

98													

pop() 98

Heaps in Action



Time Complexity of Heaps

- The two important operation are `add` and `removeMin`
- These both either percolating an element up the tree or percolating an element down the tree
- The number of operations depends on the depth of the tree which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $O(\log(n))$
- Except `add` could also require resizing the array, but the amortised cost of this is low

Time Complexity of Heaps

- The two important operation are `add` and `removeMin`
- These both either percolating an element up the tree or percolating an element down the tree
- The number of operations depends on the depth of the tree which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $O(\log(n))$
- Except `add` could also require resizing the array, but the amortised cost of this is low

Time Complexity of Heaps

- The two important operation are `add` and `removeMin`
- These both either percolating an element up the tree or percolating an element down the tree
- The number of operations depends on the depth of the tree which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $O(\log(n))$
- Except `add` could also require resizing the array, but the amortised cost of this is low

Time Complexity of Heaps

- The two important operation are `add` and `removeMin`
- These both either percolating an element up the tree or percolating an element down the tree
- The number of operations depends on the depth of the tree which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $O(\log(n))$
- Except `add` could also require resizing the array, but the amortised cost of this is low

Time Complexity of Heaps

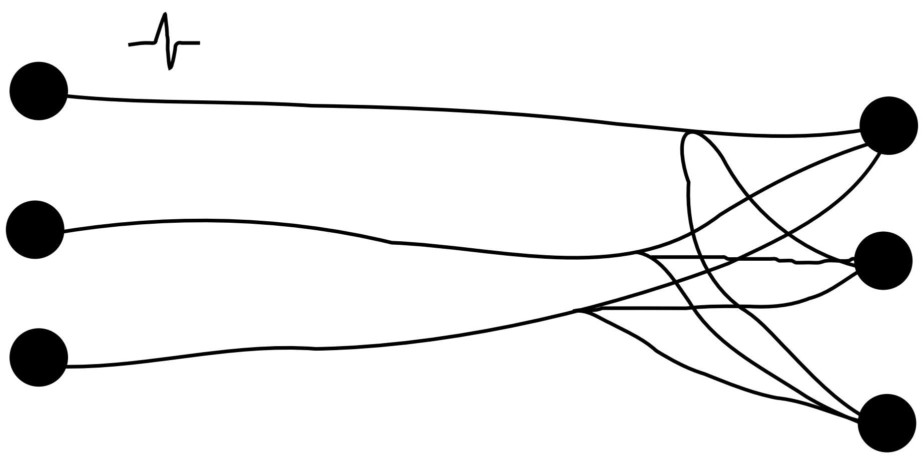
- The two important operation are `add` and `removeMin`
- These both either percolating an element up the tree or percolating an element down the tree
- The number of operations depends on the depth of the tree which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $O(\log(n))$
- Except `add` could also require resizing the array, but the amortised cost of this is low

Time Complexity of Heaps

- The two important operation are `add` and `removeMin`
- These both either percolating an element up the tree or percolating an element down the tree
- The number of operations depends on the depth of the tree which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $O(\log(n))$
- Except `add` could also require resizing the array, but the amortised cost of this is low

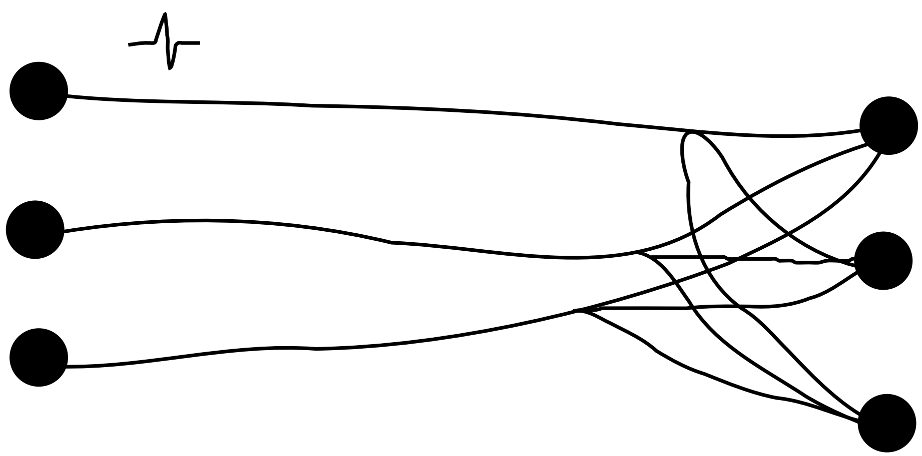
Real Time Simulation

- A nice application of priority queues is to perform real time simulations
- I was once modelling a neural network where neuron fired an impulse which would then be received by other neurons



Real Time Simulation

- A nice application of priority queues is to perform real time simulations
- I was once modelling a neural network where neuron fired an impulse which would then be received by other neurons



Synchronised Firing

- We wanted to show that if a group of neurons fired together they could make another group of neurons fire in synchrony despite the fact that it would take different times for the receiving neurons to feel the pulse (due to the different lengths of the axons)

- A famous Israeli group had "proved" this couldn't happen

- Using a priority queue we modelled the neurons
  - ⋆ When a neuron fired the receiving neurons would be put on a priority queue according to when they received the pulse
  - ⋆ If the receiving neurons received enough pulse in a short enough time they would then fire

Synchronised Firing

- We wanted to show that if a group of neurons fired together they could make another group of neurons fire in synchrony despite the fact that it would take different times for the receiving neurons to feel the pulse (due to the different lengths of the axons)
- A famous Israeli group had "proved" this couldn't happen
- Using a priority queue we modelled the neurons
  - ⋆ When a neuron fired the receiving neurons would be put on a priority queue according to when they received the pulse
  - ⋆ If the receiving neurons received enough pulse in a short enough time they would then fire

Synchronised Firing

- We wanted to show that if a group of neurons fired together they could make another group of neurons fire in synchrony despite the fact that it would take different times for the receiving neurons to feel the pulse (due to the different lengths of the axons)
- A famous Israeli group had "proved" this couldn't happen
- Using a priority queue we modelled the neurons
  - ⋆ When a neuron fired the receiving neurons would be put on a priority queue according to when they received the pulse
  - ⋆ If the receiving neurons received enough pulse in a short enough time they would then fire

Synchronised Firing

- We wanted to show that if a group of neurons fired together they could make another group of neurons fire in synchrony despite the fact that it would take different times for the receiving neurons to feel the pulse (due to the different lengths of the axons)
- A famous Israeli group had “proved” this couldn’t happen
- Using a priority queue we modelled the neurons
  - ⋆ When a neuron fired the receiving neurons would be put on a priority queue according to when they received the pulse
  - ⋆ If the receiving neurons received enough pulse in a short enough time they would then fire

Efficient Modelling

- Using a priority queue meant we knew when the next event would happen
- We did not have to run a clock where most of the time nothing happened
- This allowed us to perform a very large simulation efficiently
- The simulation showed that the pulse of neurons synchronised despite the “proof” that this wouldn’t happen

Efficient Modelling

- Using a priority queue meant we knew when the next event would happen
- We did not have to run a clock where most of the time nothing happened
- This allowed us to perform a very large simulation efficiently
- The simulation showed that the pulse of neurons synchronised despite the “proof” that this wouldn’t happen

Efficient Modelling

- Using a priority queue meant we knew when the next event would happen

- We did not have to run a clock where most of the time nothing happened

- This allowed us to perform a very large simulation efficiently

- The simulation showed that the pulse of neurons synchronised despite the “proof” that this wouldn’t happen

Efficient Modelling

- Using a priority queue meant we knew when the next event would happen
- We did not have to run a clock where most of the time nothing happened
- This allowed us to perform a very large simulation efficiently
- The simulation showed that the pulse of neurons synchronised despite the “proof” that this wouldn’t happen

Outline

1. Heaps
2. Priority Queues
   - Array Implementation
3. **Heap Sort**
4. Other Heaps

Heap Sort

- A priority queue suggests a very simple way of performing sort

- We simply add elements to a heap and then take them off again

```
template <typename T>
void sort(vector<T> aList)
{
    HeapPQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
      aHeap.push(element, element);

    aList.clear();
    while(aHeap.size() > 0) {
      aList.push_back(aHeap.top());
      aHeap.pop();
    }
}
```

- Note that this is not an in-place sort algorithm (i.e. it uses lots of memory)

Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

```
template <typename T>
void sort(vector<T> aList)
{
    HeapPQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
      aHeap.push(element, element);

    aList.clear();
    while(aHeap.size() > 0) {
      aList.push_back(aHeap.top());
      aHeap.pop();
    }
}
```

- Note that this is not an in-place sort algorithm (i.e. it uses lots of memory)

Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

```
template <typename T>
void sort(vector<T> aList)
{
    HeapPQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
      aHeap.push(element, element);

    aList.clear();
    while(aHeap.size() > 0) {
      aList.push_back(aHeap.top());
      aHeap.pop();
    }
}
```

- Note that this is not an in-place sort algorithm (i.e. it uses lots of memory)

Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

```
template <typename T>
void sort(vector<T> aList)
{
    HeapPQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
      aHeap.push(element, element);

    aList.clear();
    while(aHeap.size() > 0) {
      aList.push_back(aHeap.top());
      aHeap.pop();
    }
}
```

- Note that this is not an in-place sort algorithm (i.e. it uses lots of memory)

Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

```
template <typename T>
void sort(vector<T> aList)
{
    HeapPQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
      aHeap.push(element, element);

    aList.clear();
    while(aHeap.size() > 0) {
      aList.push_back(aHeap.top());
      aHeap.pop();
    }
}
```

- Note that this is not an in-place sort algorithm (i.e. it uses lots of memory)

Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

```
template <typename T>
void sort(vector<T> aList)
{
    HeapPQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
      aHeap.push(element, element);

    aList.clear();
    while(aHeap.size() > 0) {
      aList.push_back(aHeap.top());
      aHeap.pop();
    }
}
```

- Note that this is not an in-place sort algorithm (i.e. it uses lots of memory)

Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63

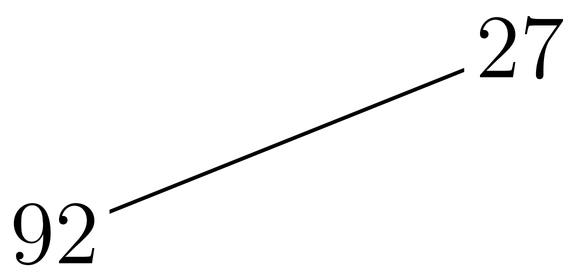
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63

92

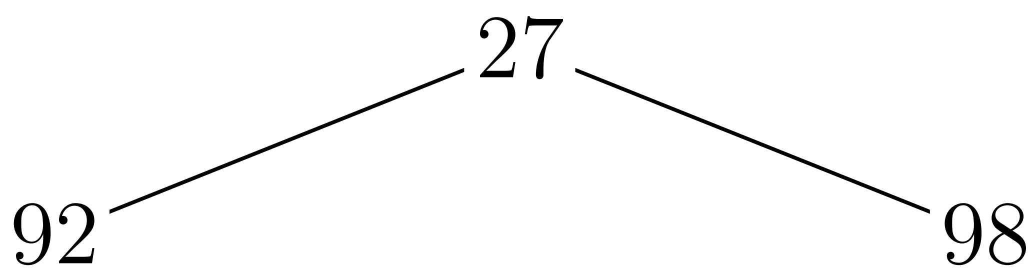
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



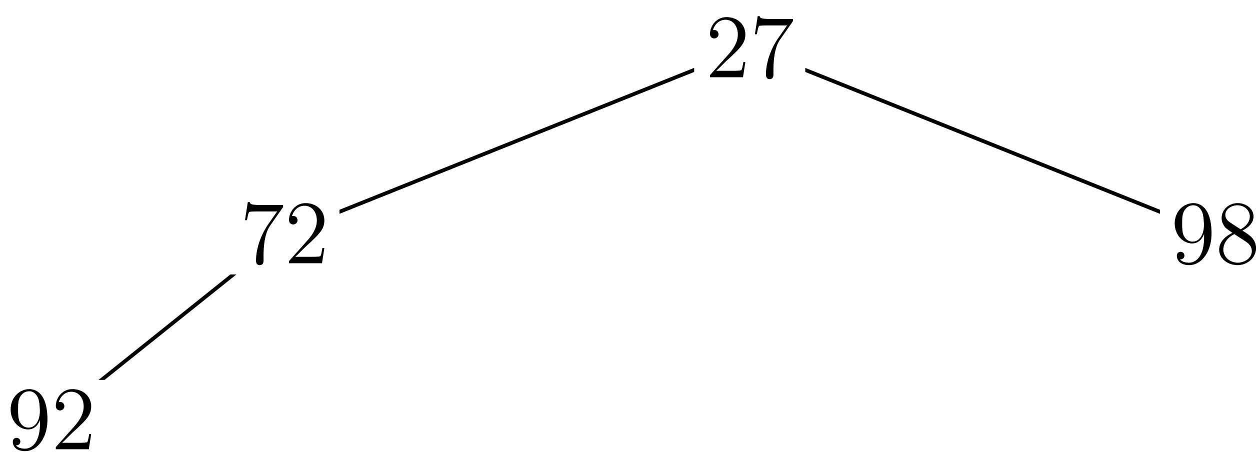
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



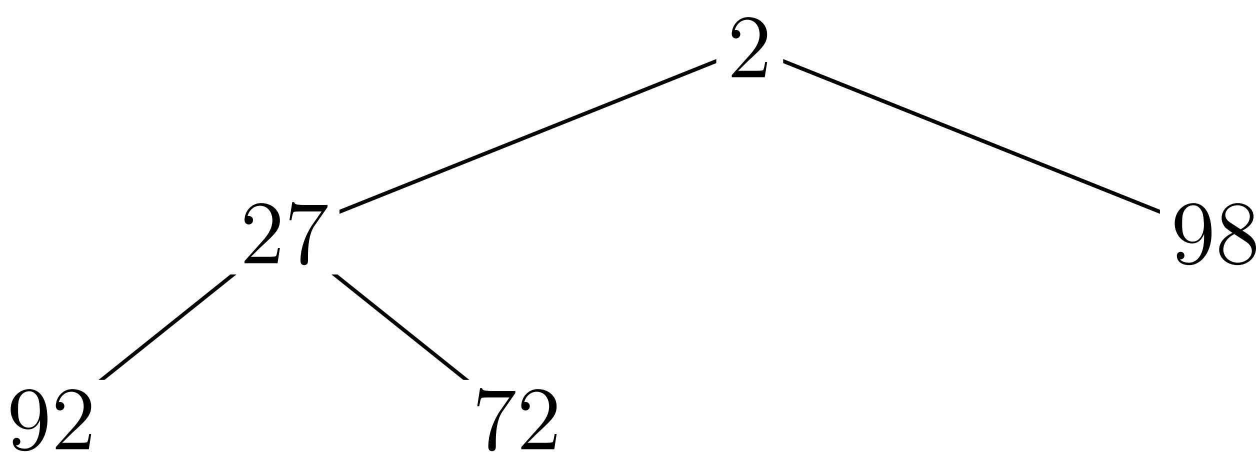
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



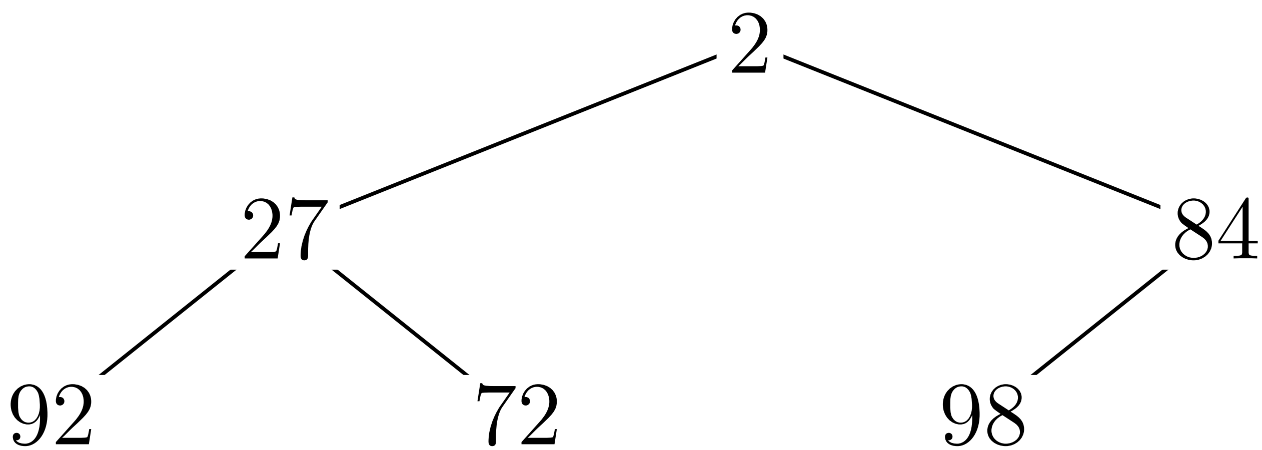
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



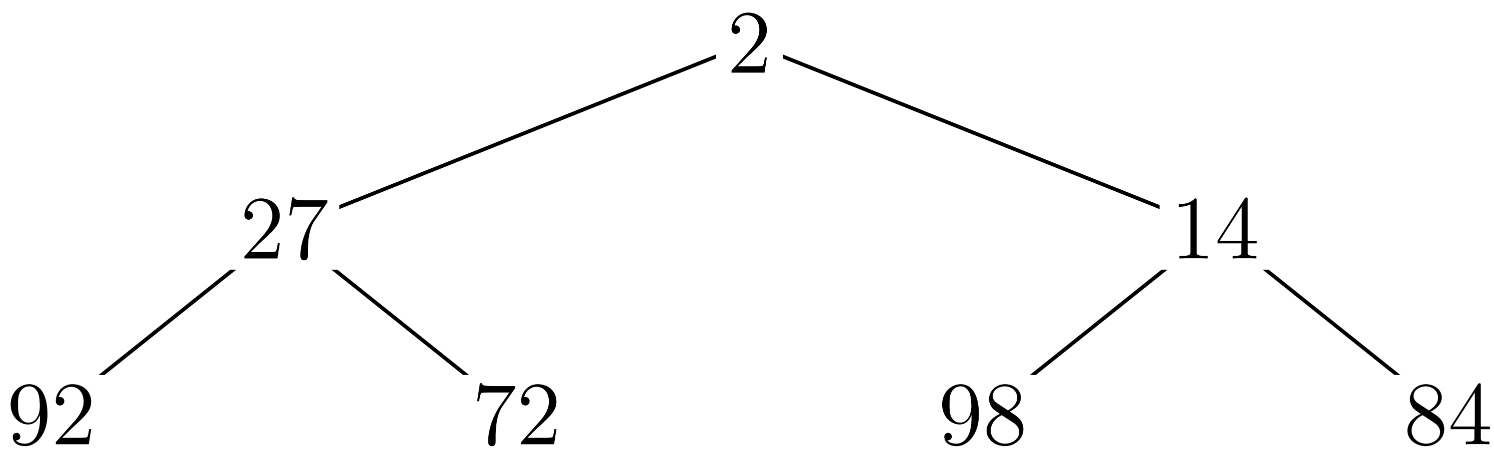
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



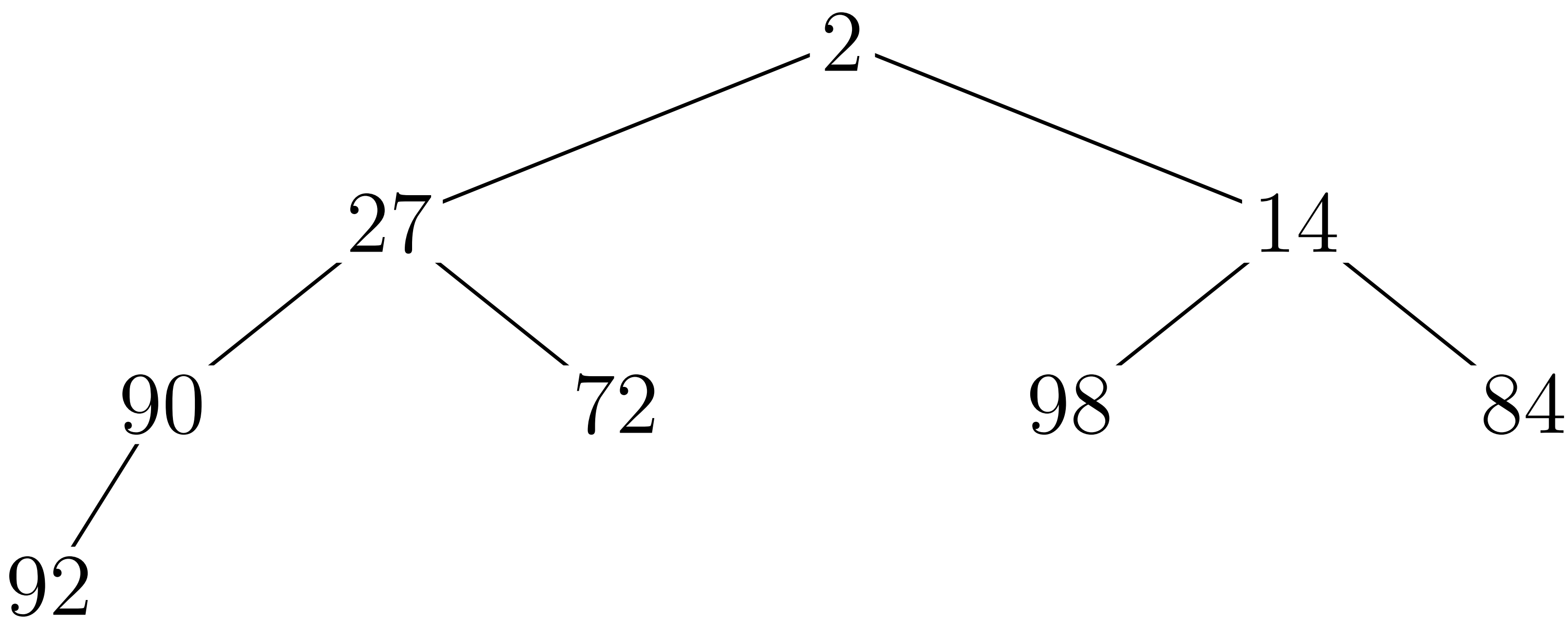
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



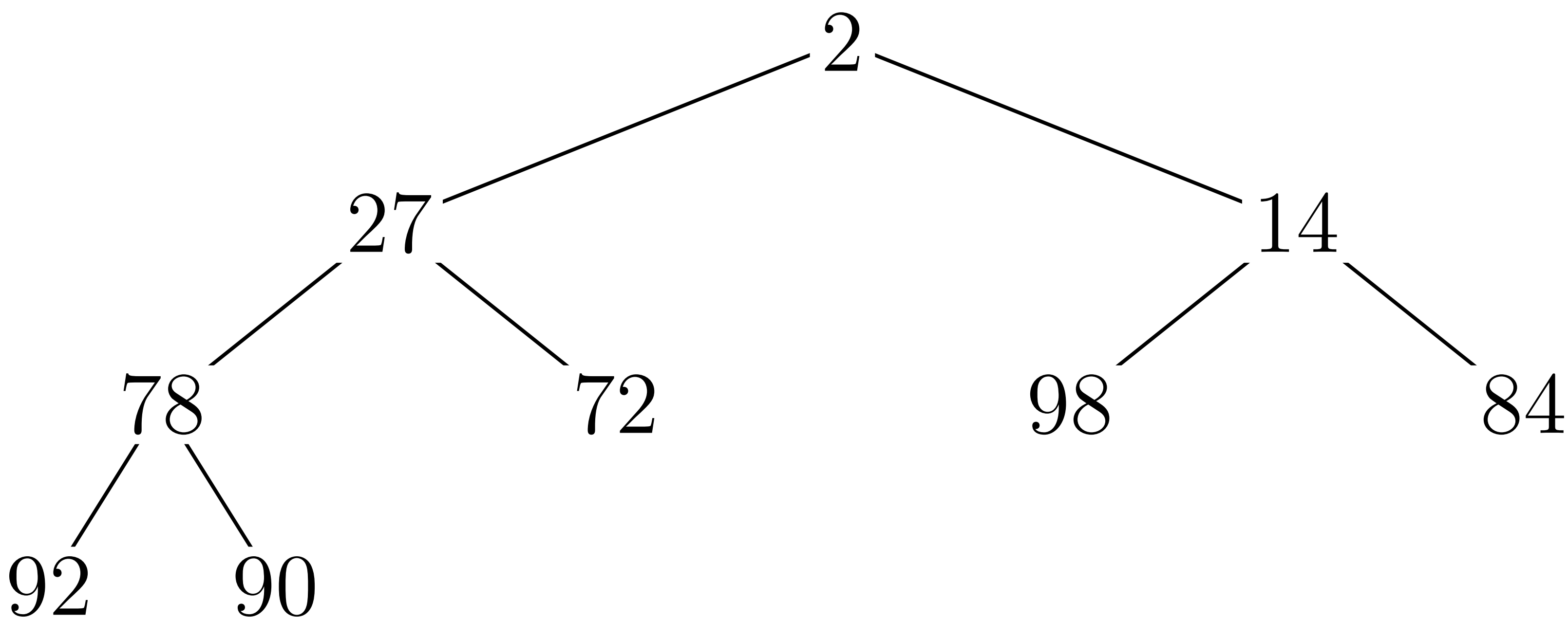
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



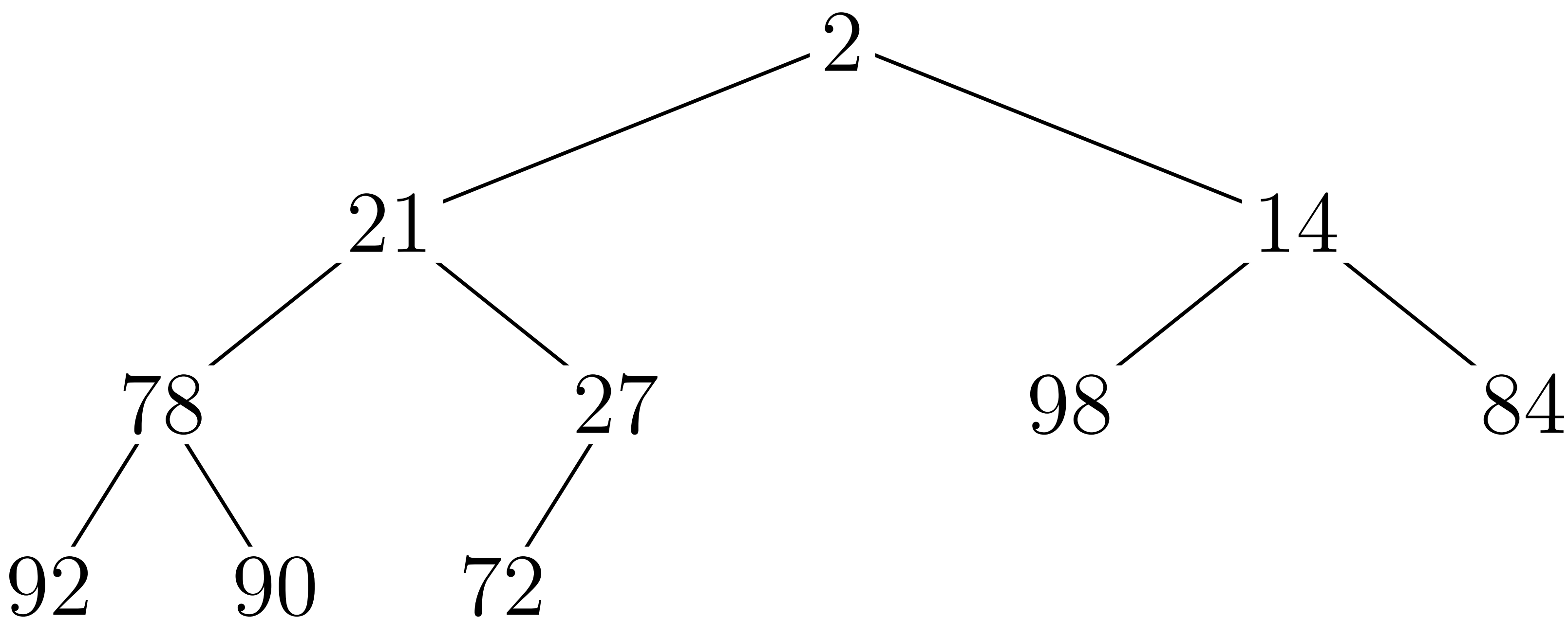
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



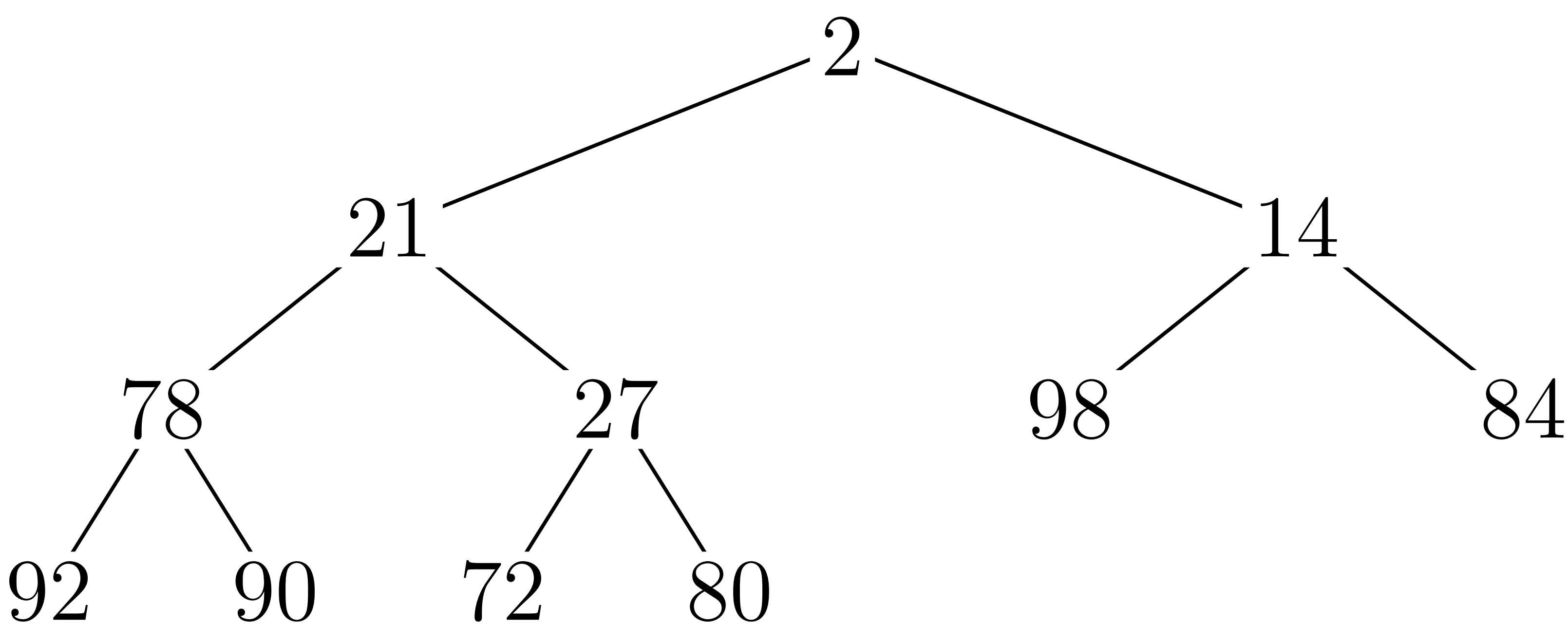
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63



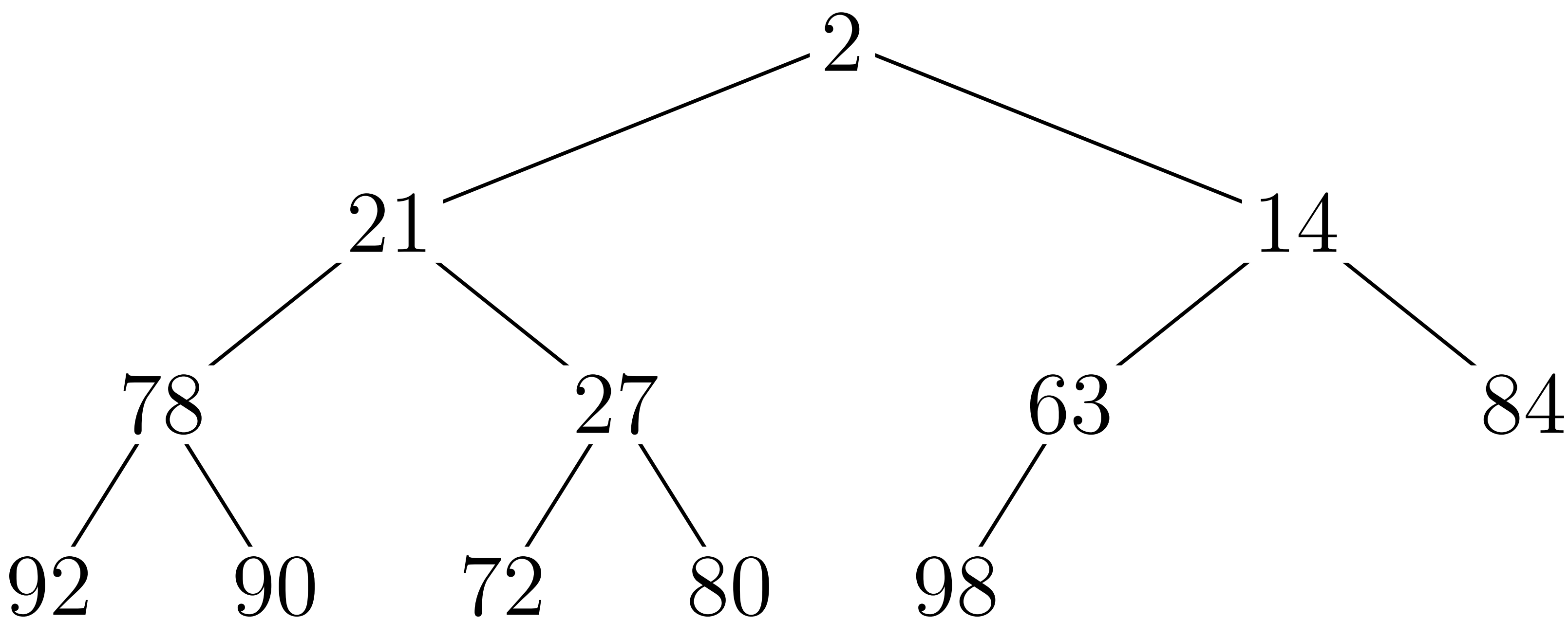
Example of Heap Sort

92	27	98	72	2	84	14	90	78	21	80	63

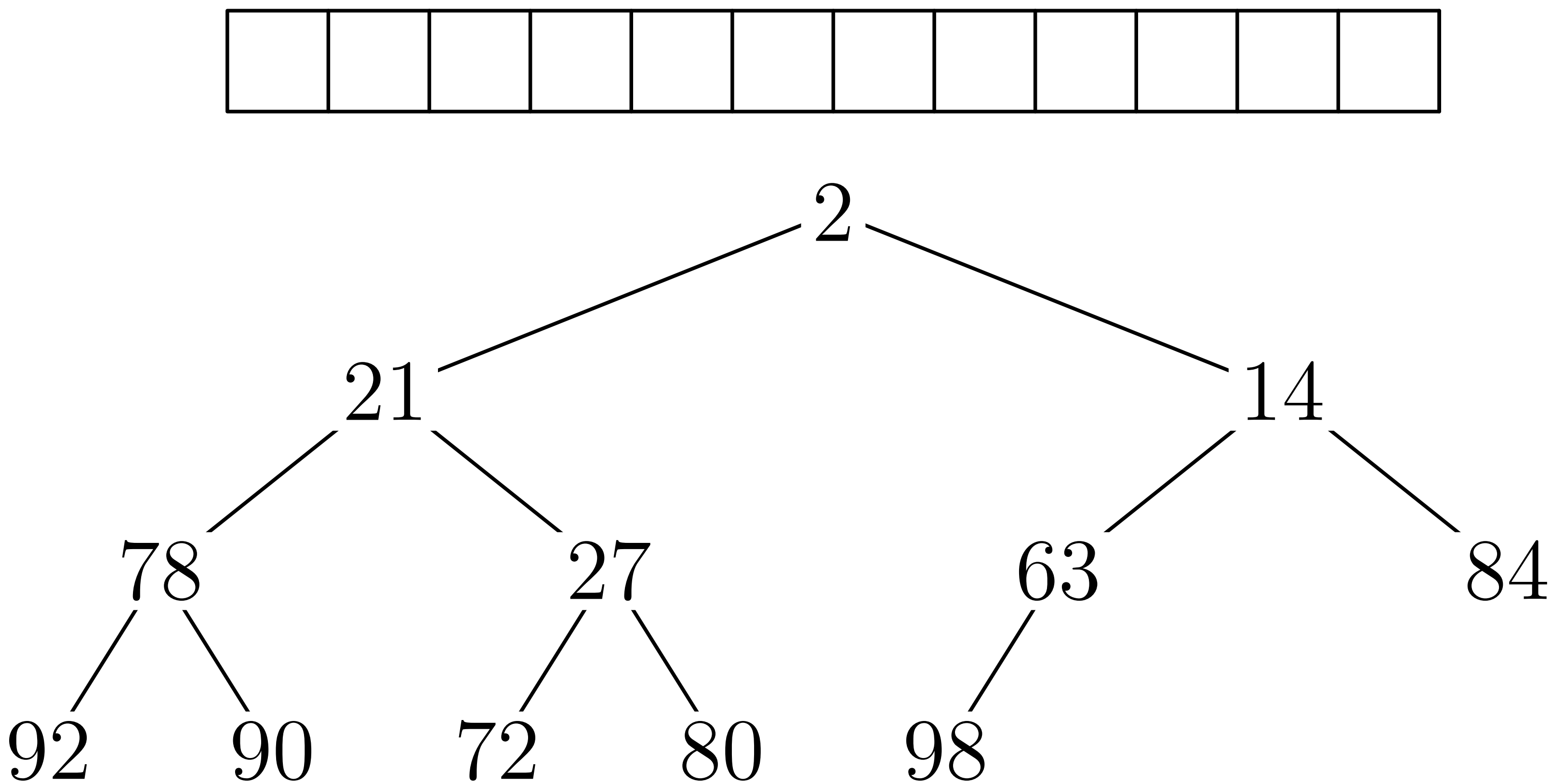


Example of Heap Sort

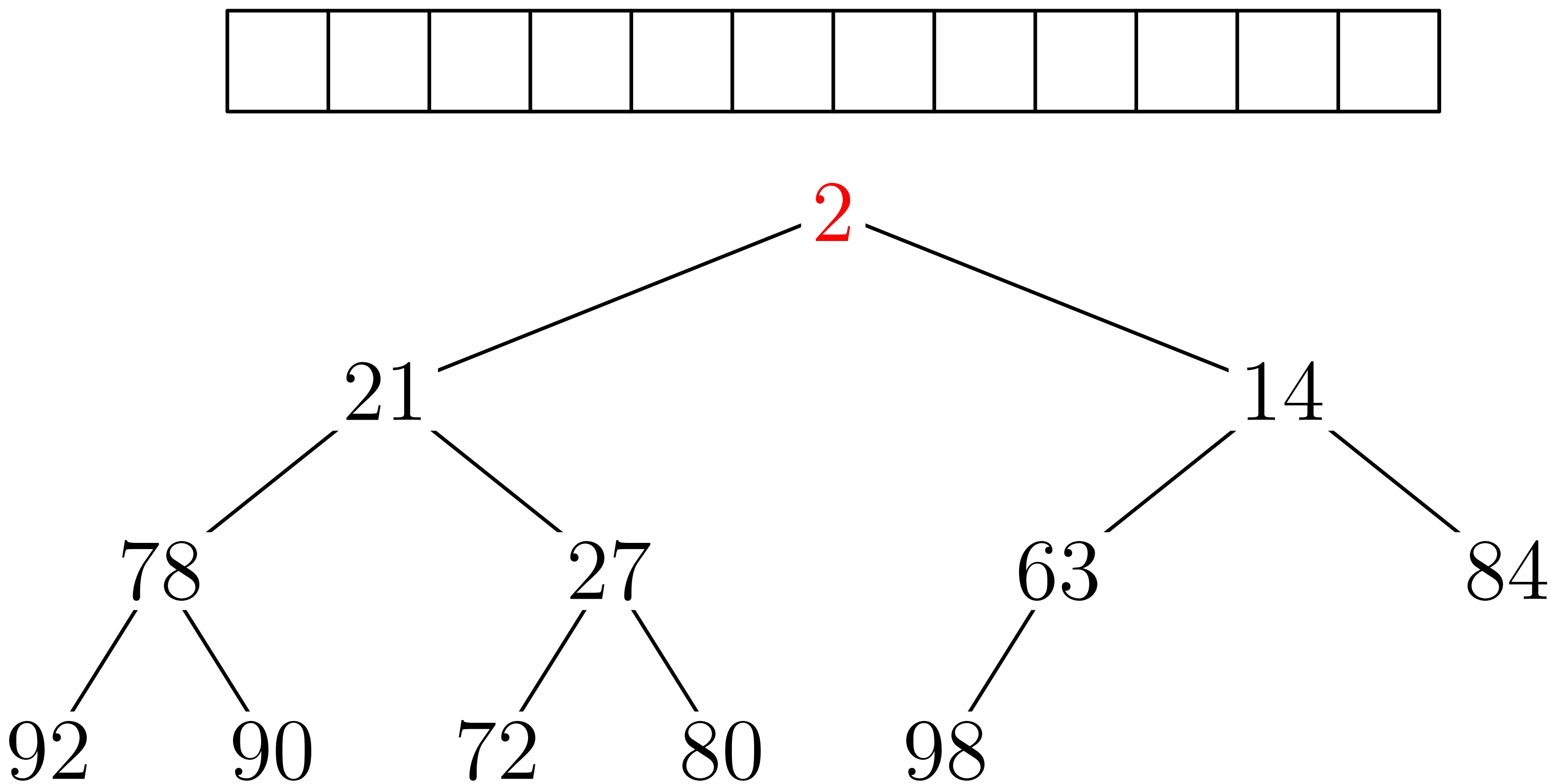
92	27	98	72	2	84	14	90	78	21	80	63



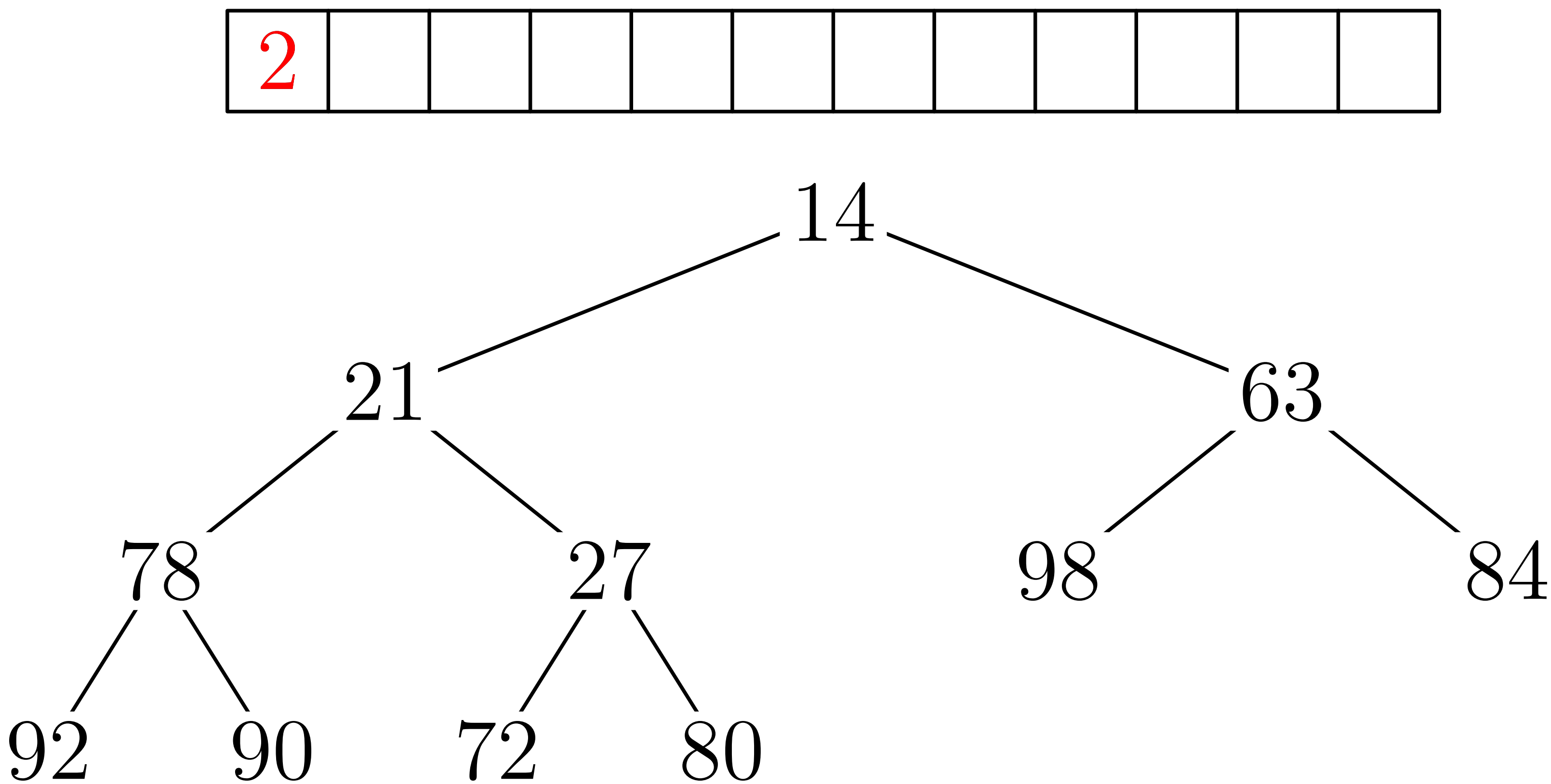
Example of Heap Sort



Example of Heap Sort

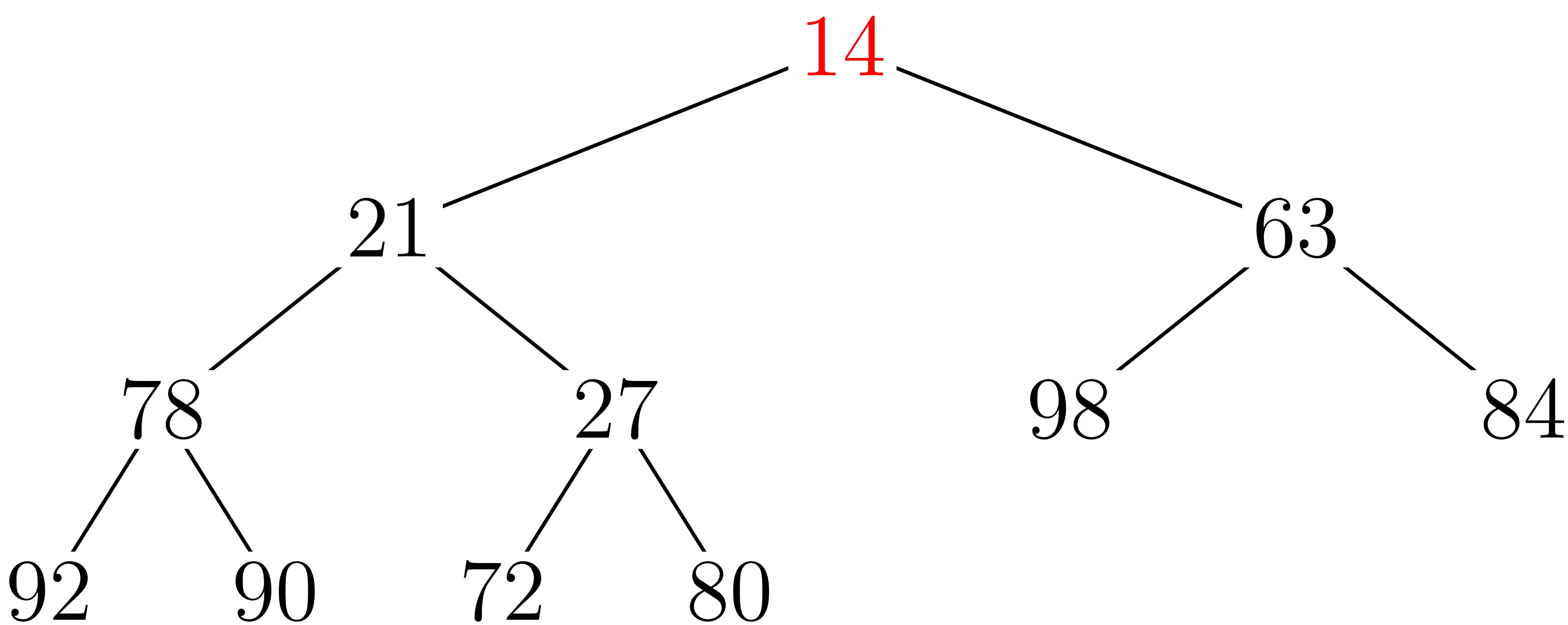


Example of Heap Sort



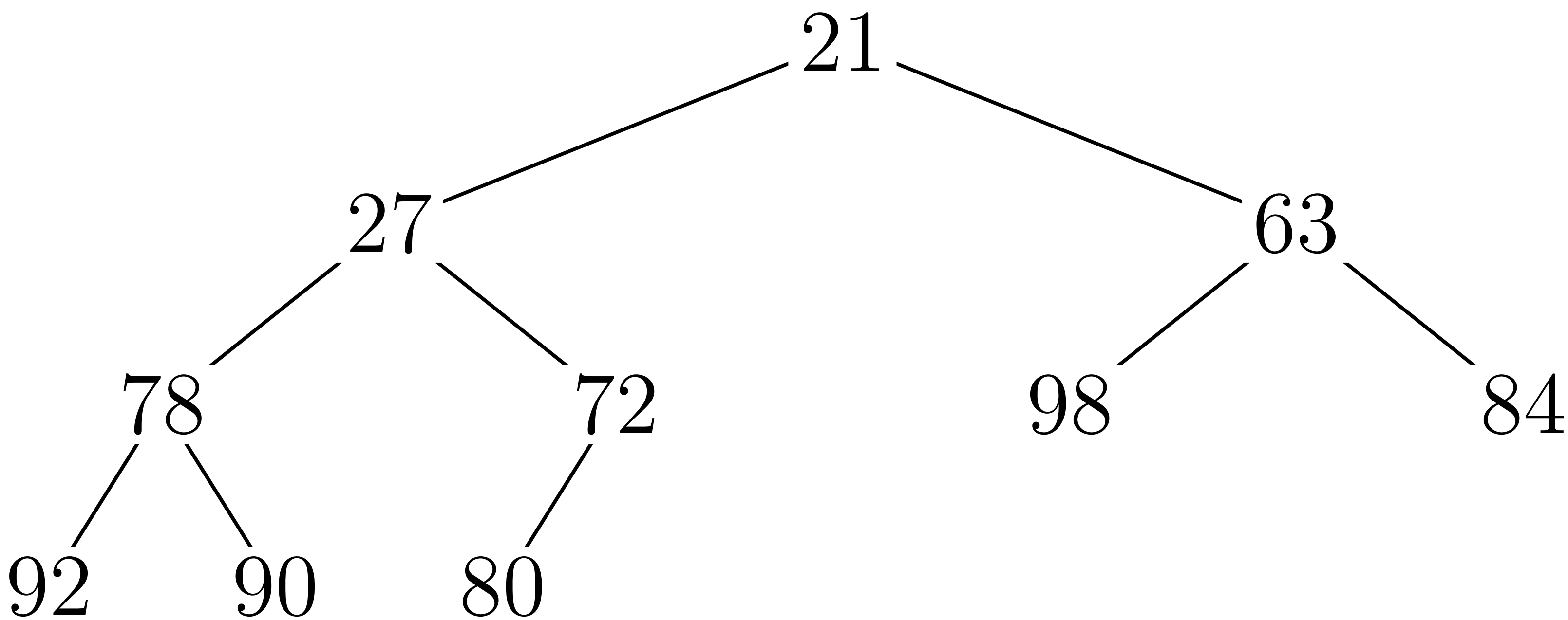
Example of Heap Sort

2											

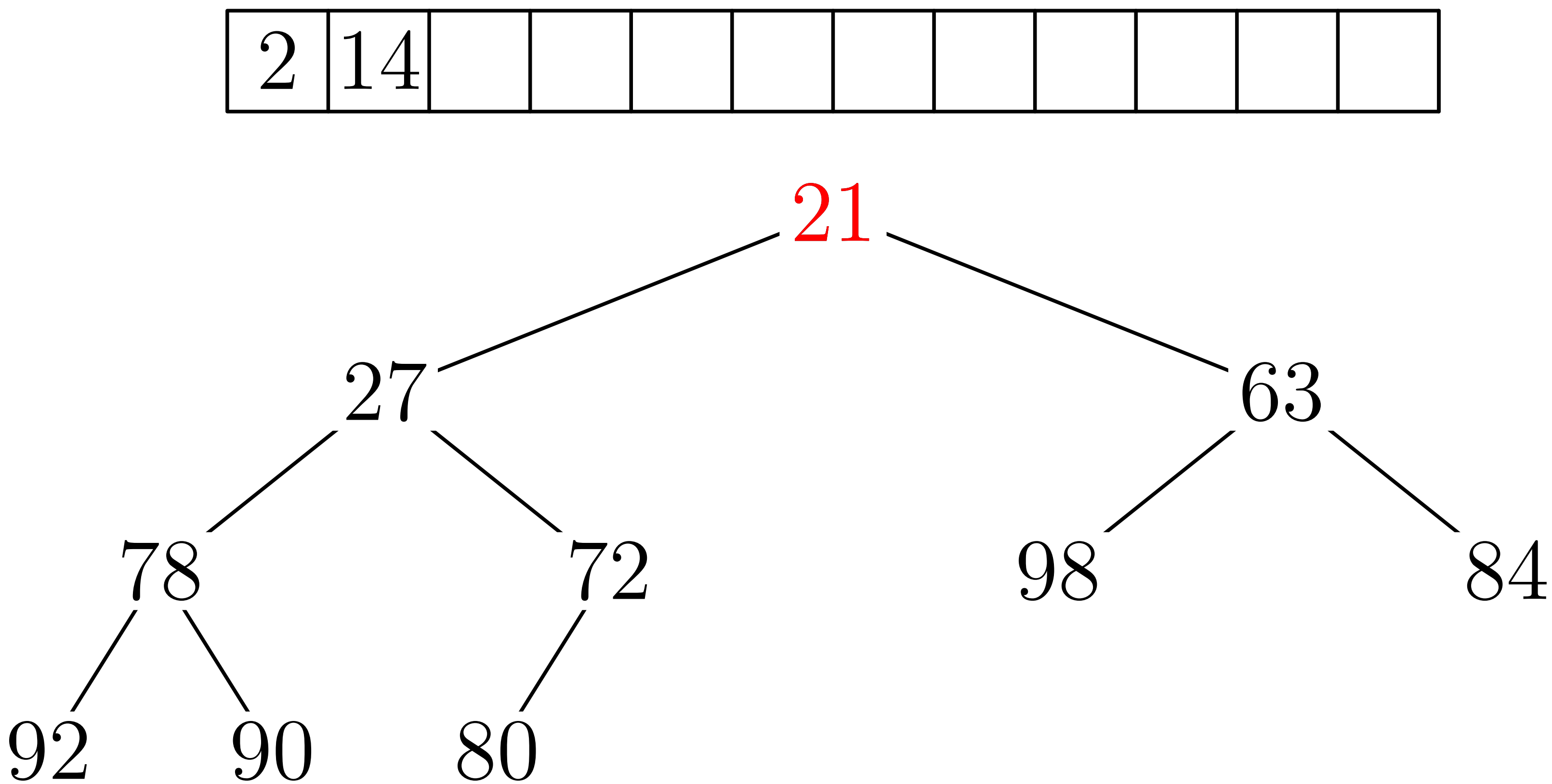


Example of Heap Sort

2	14										

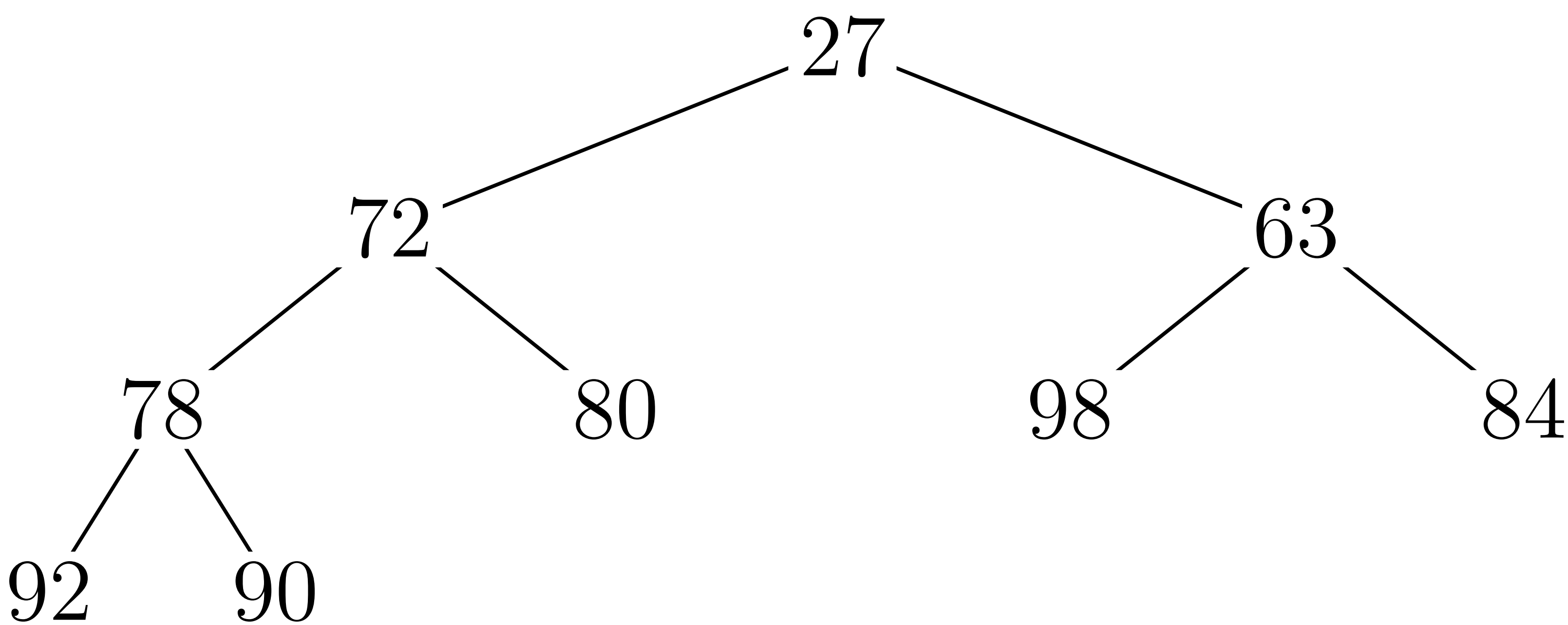


Example of Heap Sort



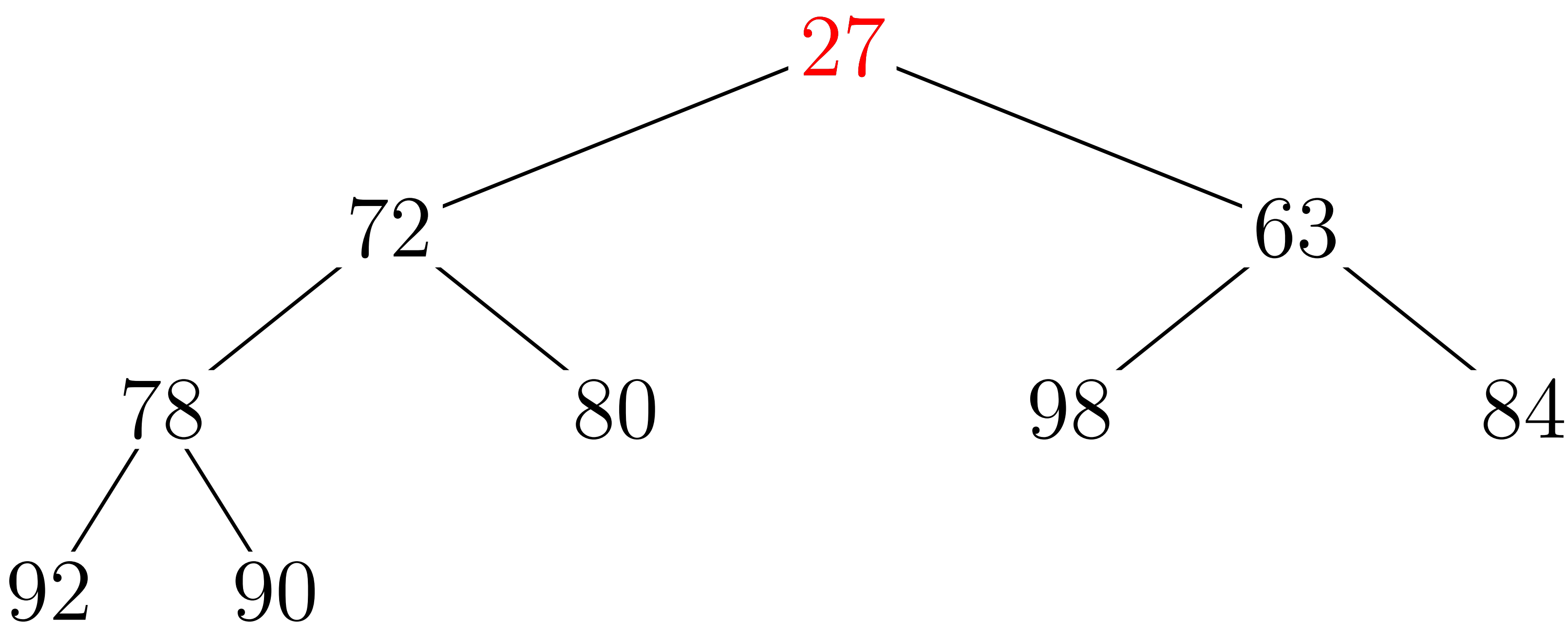
Example of Heap Sort

2	14	21									



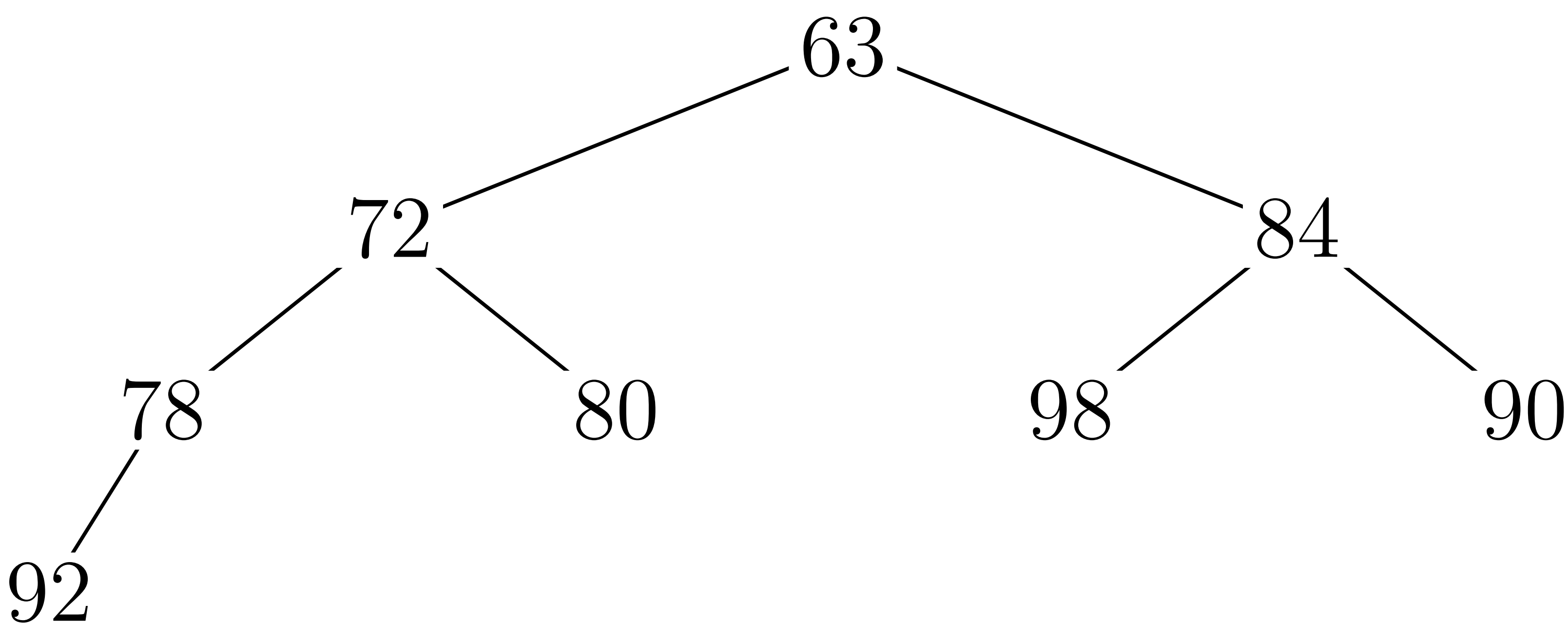
Example of Heap Sort

2	14	21									



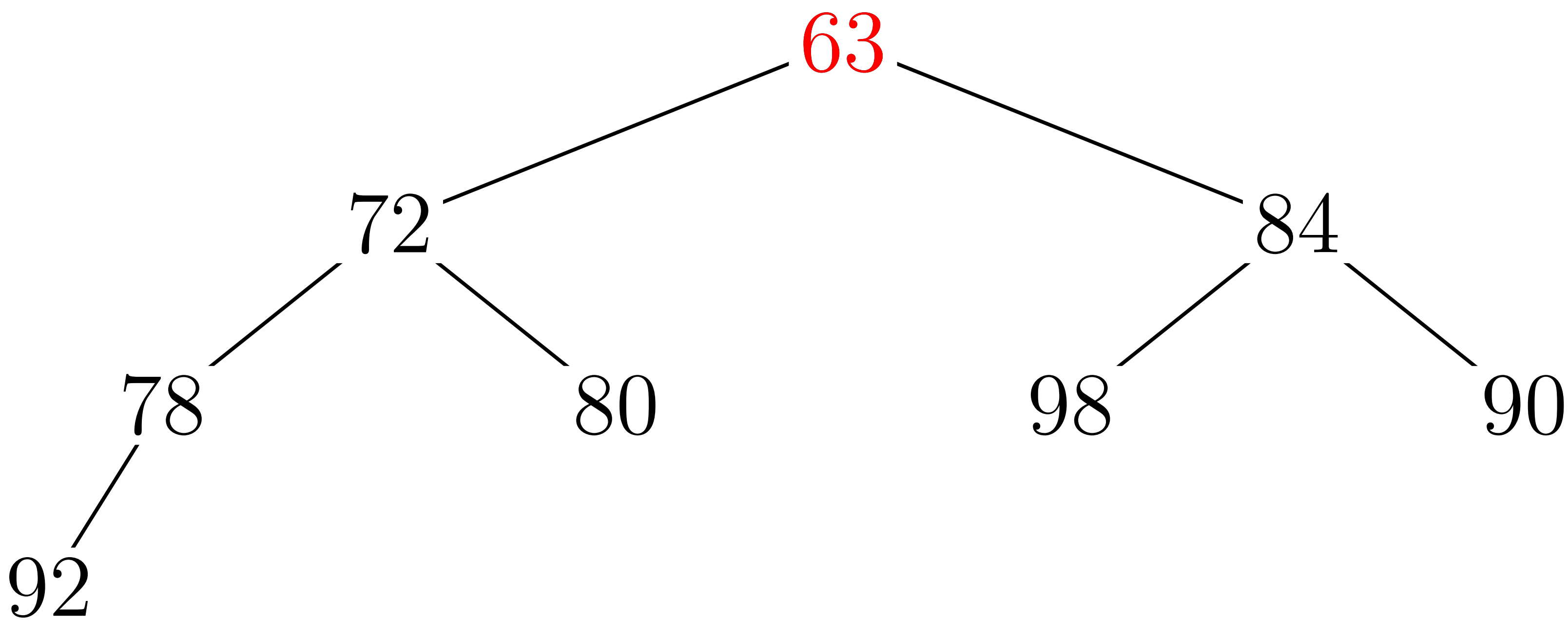
Example of Heap Sort

2	14	21	27								

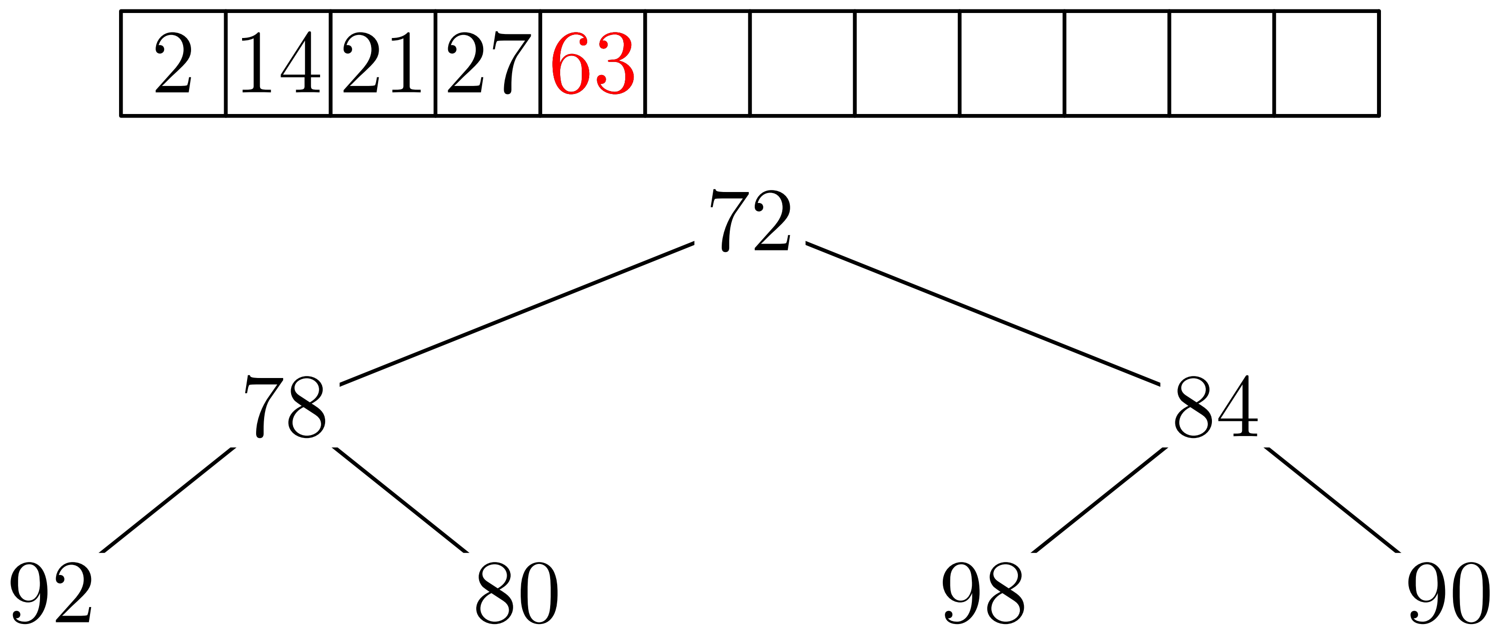


Example of Heap Sort

2	14	21	27								

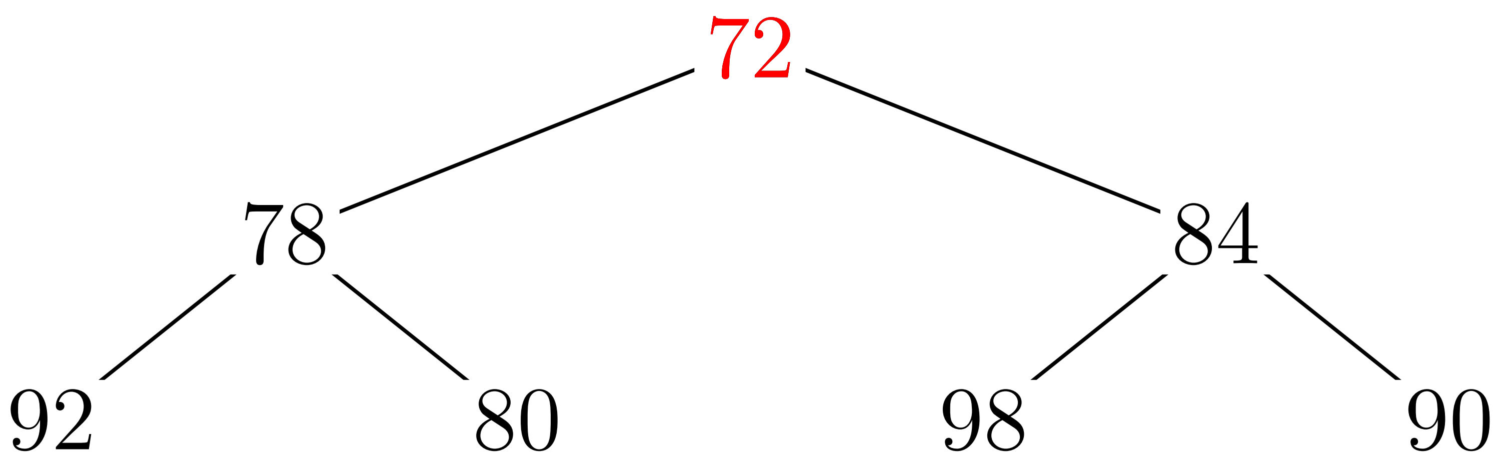


Example of Heap Sort



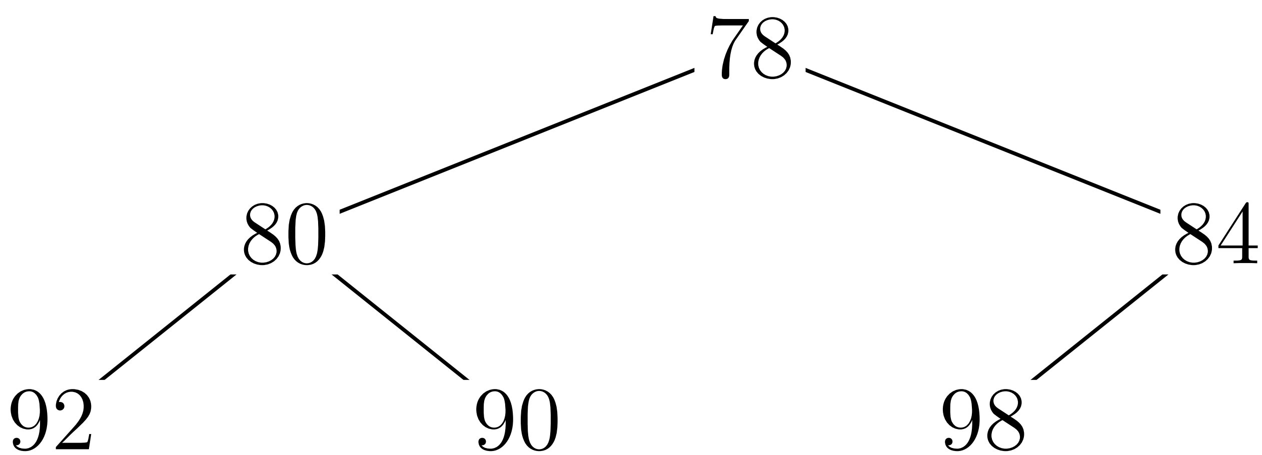
Example of Heap Sort

2	14	21	27	63							



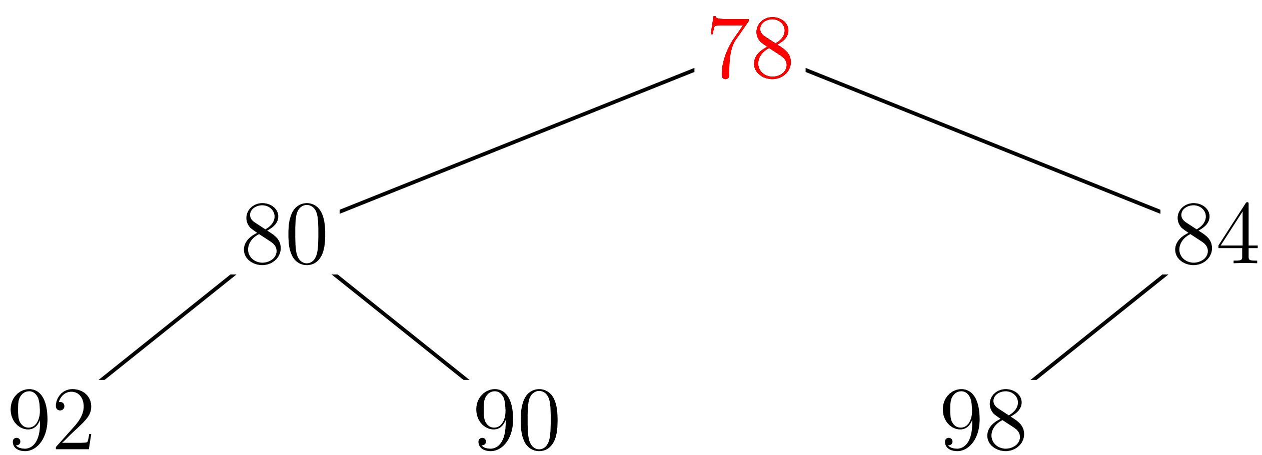
Example of Heap Sort

2	14	21	27	63	72						



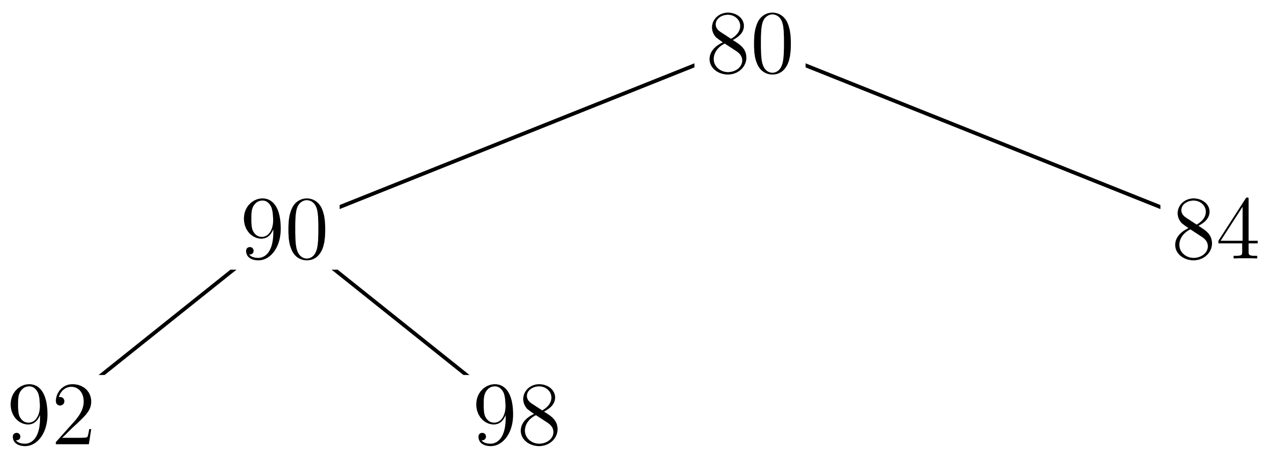
Example of Heap Sort

2	14	21	27	63	72						



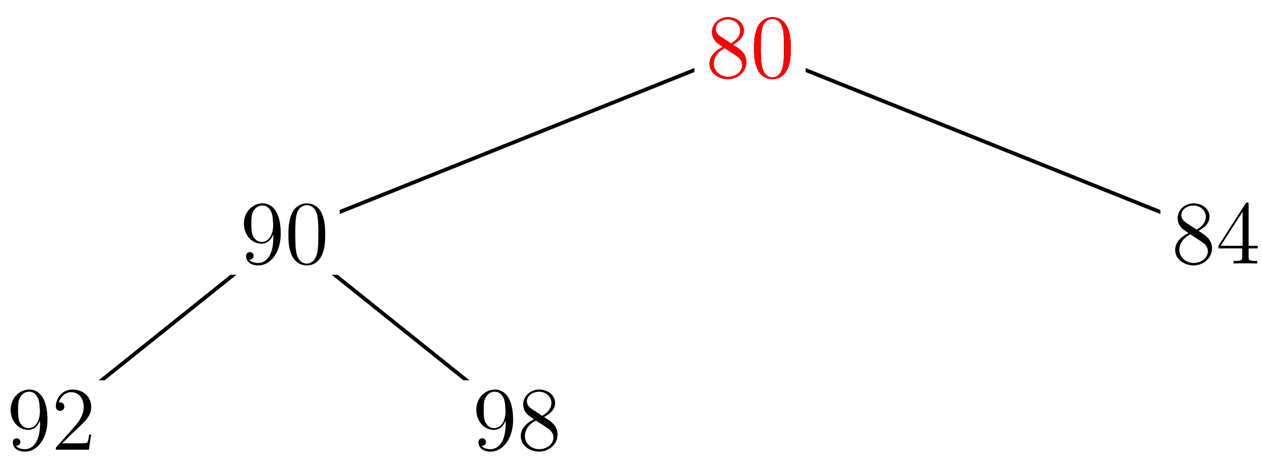
Example of Heap Sort

2	14	21	27	63	72	78					



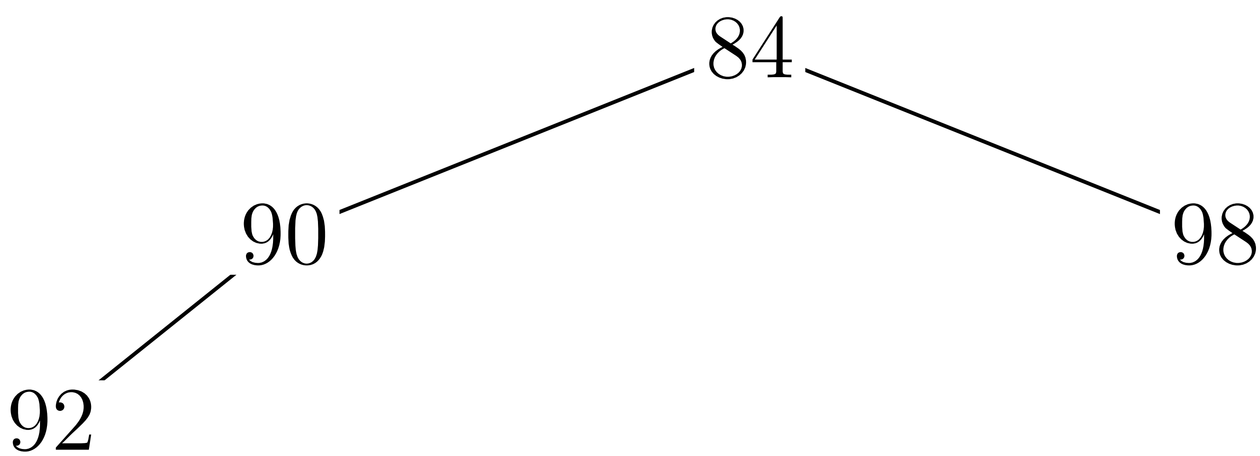
Example of Heap Sort

2	14	21	27	63	72	78					



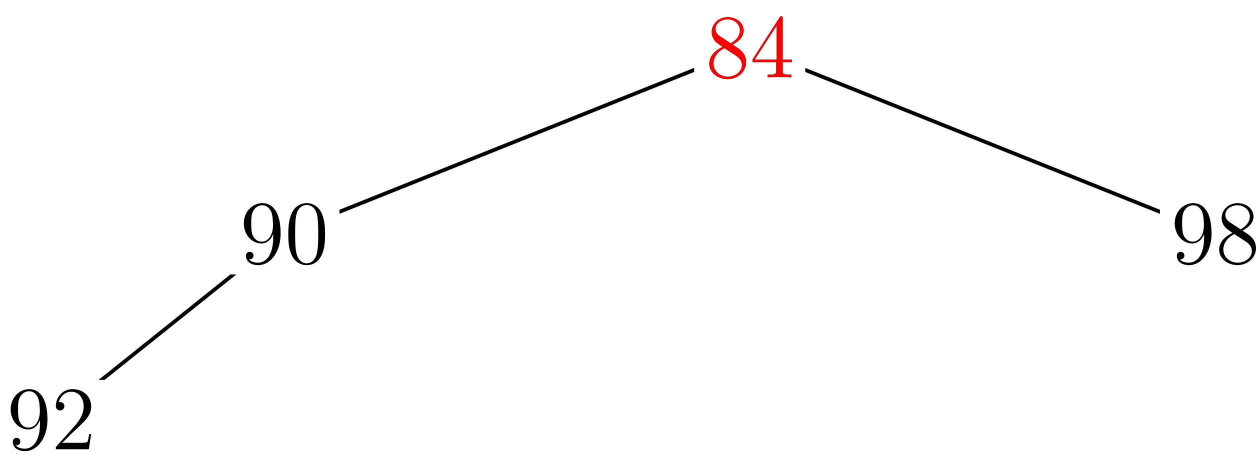
Example of Heap Sort

2	14	21	27	63	72	78	80				



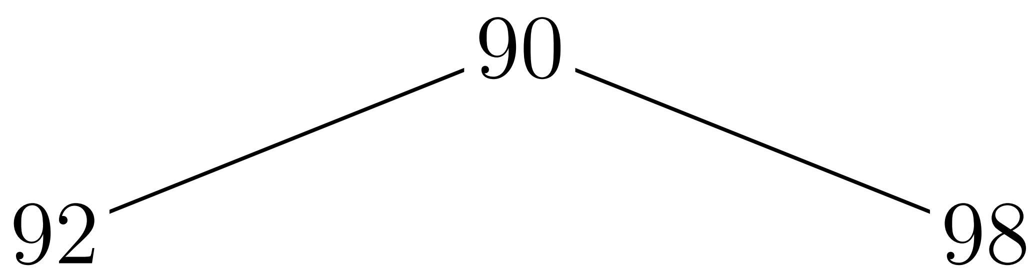
Example of Heap Sort

2	14	21	27	63	72	78	80				



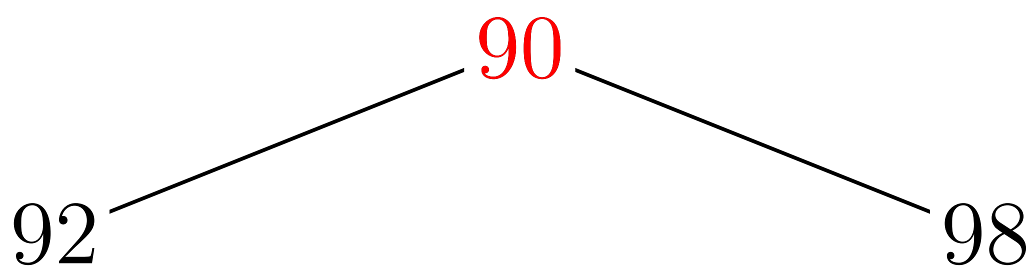
Example of Heap Sort

2	14	21	27	63	72	78	80	84			



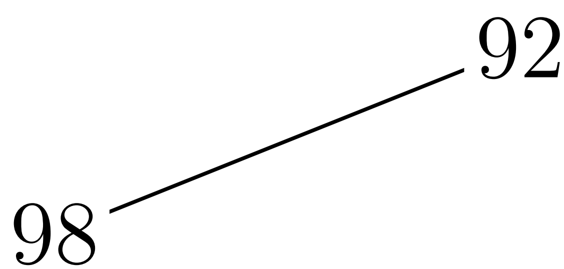
Example of Heap Sort

2	14	21	27	63	72	78	80	84			



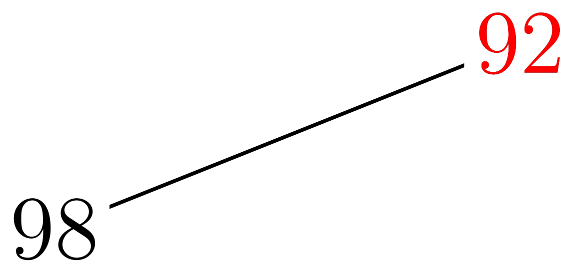
Example of Heap Sort

2	14	21	27	63	72	78	80	84	90		



Example of Heap Sort

2	14	21	27	63	72	78	80	84	90		



Example of Heap Sort

2	14	21	27	63	72	78	80	84	90	92	

98

Example of Heap Sort

2	14	21	27	63	72	78	80	84	90	92	

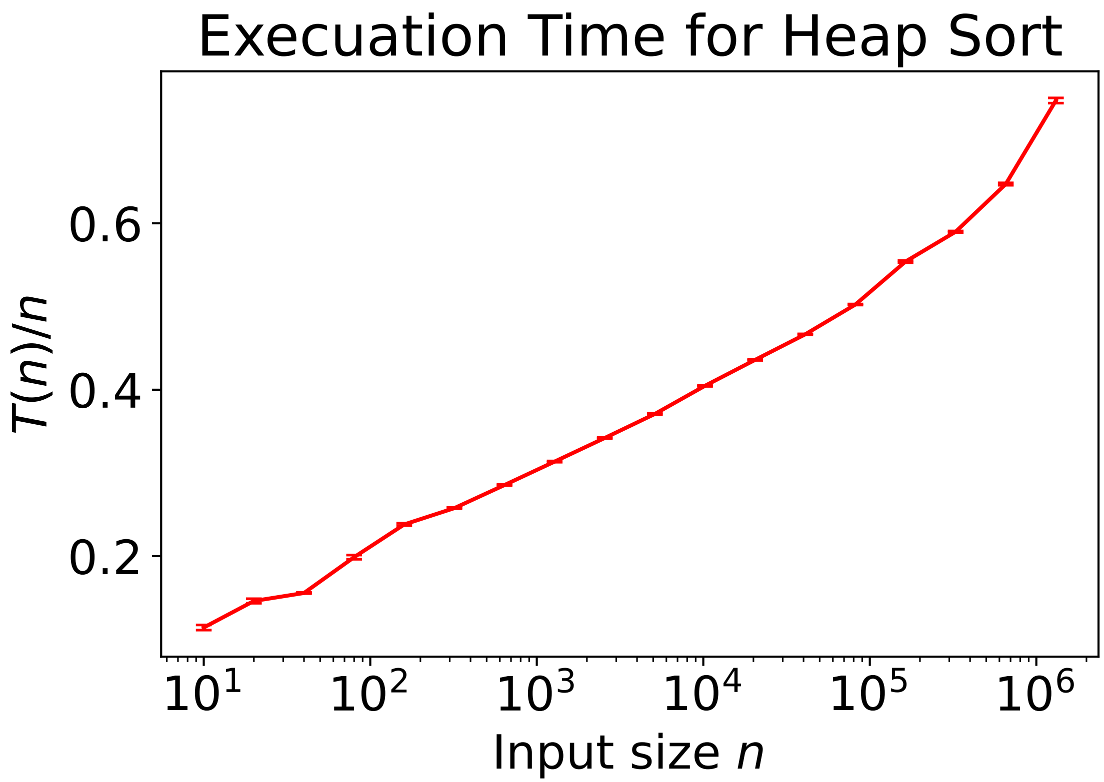
98

Example of Heap Sort

2	14	21	27	63	72	78	80	84	90	92	98

Complexity of Heap Sort

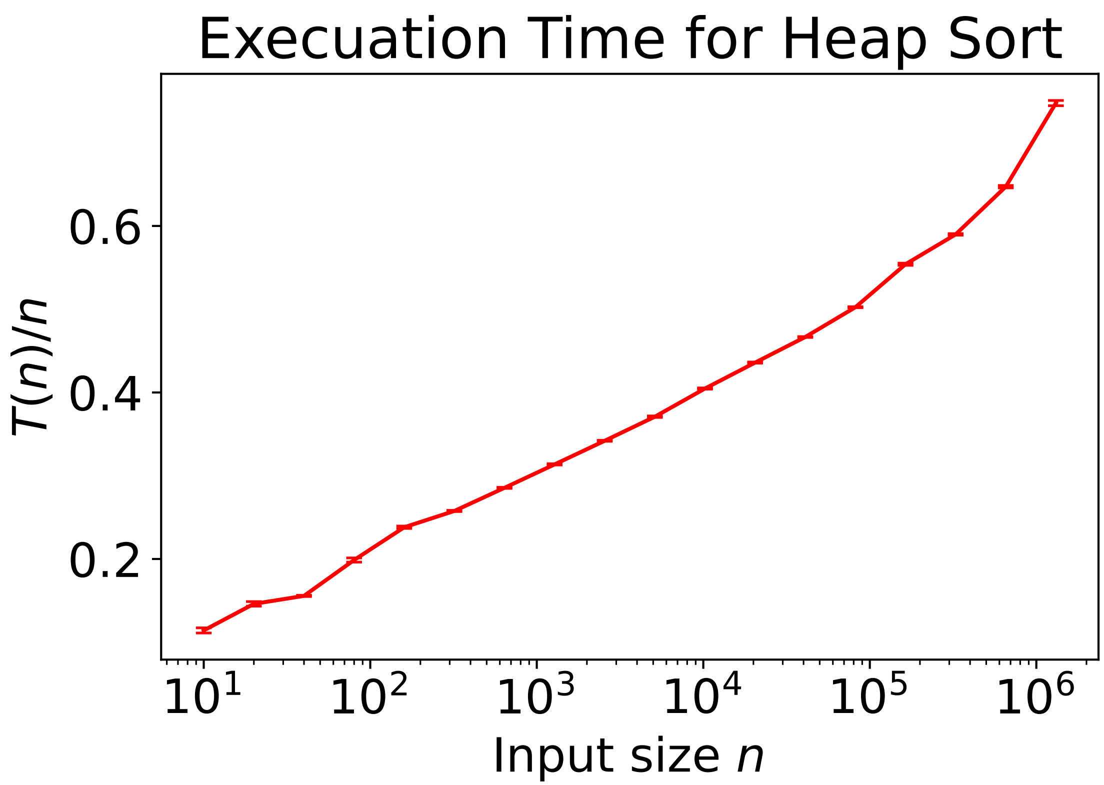
- As we have to add n elements and then remove n elements the time complexity is log-linear, i.e. $O(n\log(n))$



- This is actually a very efficient algorithm

Complexity of Heap Sort

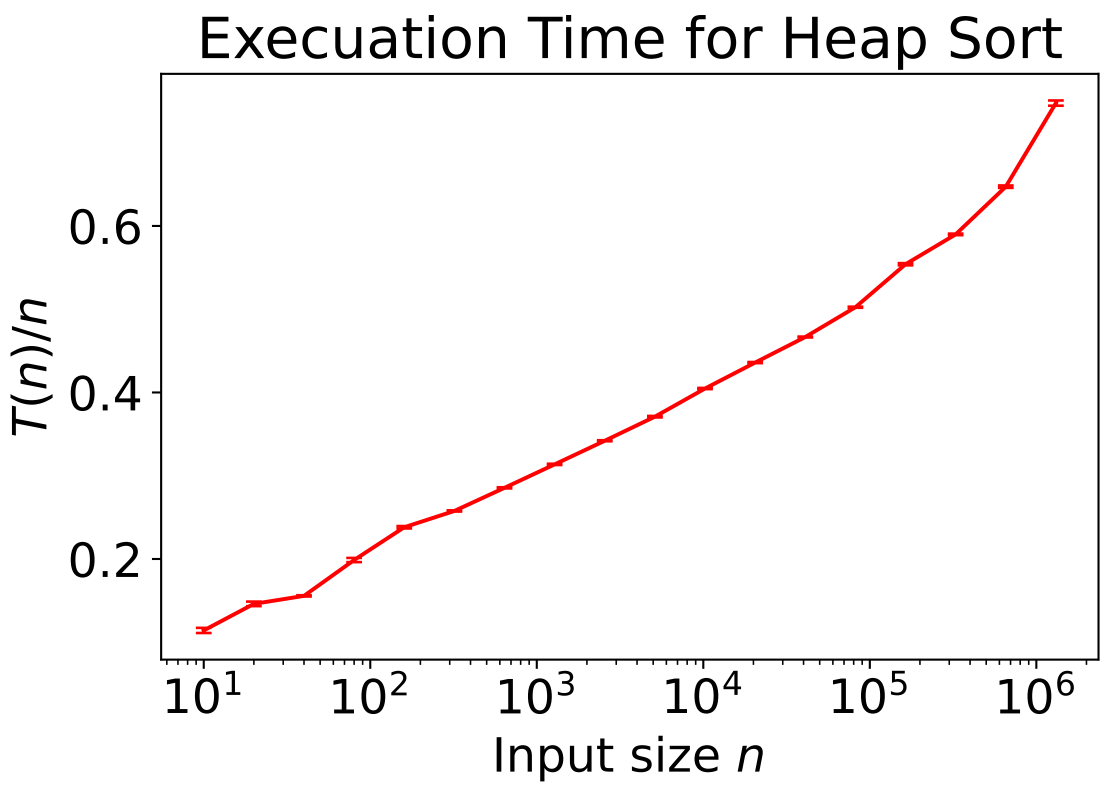
- As we have to add n elements and then remove n elements the time complexity is log-linear, i.e. $O(n\,\log(n))$



- This is actually a very efficient algorithm

Complexity of Heap Sort

- As we have to add n elements and then remove n elements the time complexity is log-linear, i.e. $O(n\,\log(n))$



- This is actually a very efficient algorithm

Outline

1. Heaps
2. Priority Queues
   - Array Implementation
3. Heap Sort
4. **Other Heaps**

Other Heaps

- Binary Heaps are so useful that other types of heaps have been developed
- The simplest enhancement is to combine a binary heap with a map which maintains a pointer to each element
- The map has to be updated every time elements are moved in the heap (fortunately only $O(\log(n))$ elements are move each time the heap is updated)
- The advantage of this heap is that the priorities of elements can be changed (involving percolating elements up or down the tree)

Other Heaps

- Binary Heaps are so useful that other types of heaps have been developed
- The simplest enhancement is to combine a binary heap with a map which maintains a pointer to each element
- The map has to be updated every time elements are moved in the heap (fortunately only $O(\log(n))$ elements are move each time the heap is updated)
- The advantage of this heap is that the priorities of elements can be changed (involving percolating elements up or down the tree)

Other Heaps

- Binary Heaps are so useful that other types of heaps have been developed
- The simplest enhancement is to combine a binary heap with a map which maintains a pointer to each element
- The map has to be updated every time elements are moved in the heap (fortunately only $O(\log(n))$ elements are move each time the heap is updated)
- The advantage of this heap is that the priorities of elements can be changed (involving percolating elements up or down the tree)

Other Heaps

- Binary Heaps are so useful that other types of heaps have been developed
- The simplest enhancement is to combine a binary heap with a map which maintains a pointer to each element
- The map has to be updated every time elements are moved in the heap (fortunately only $O(\log(n))$ elements are move each time the heap is updated)
- The advantage of this heap is that the priorities of elements can be changed (involving percolating elements up or down the tree)

Merging Heaps

- One common demand on a heaps is to merge two heaps
- Unfortunately binary heaps are not efficiently merged
- There are a variety of different heaps (leftist heaps, skew heaps, binomial queues, . . .) designed to be merged
- All these heaps are real binary trees (i.e. represented by pointers rather than put in an array)
- They are slower than a binary heap because indexing is slightly slower as is creating node objects
- However, they allow merging

Merging Heaps

- One common demand on a heaps is to merge two heaps
- Unfortunately binary heaps are not efficiently merged
- There are a variety of different heaps (leftist heaps, skew heaps, binomial queues, . . .) designed to be merged
- All these heaps are real binary trees (i.e. represented by pointers rather than put in an array)
- They are slower than a binary heap because indexing is slightly slower as is creating node objects
- However, they allow merging

Merging Heaps

- One common demand on a heaps is to merge two heaps
- Unfortunately binary heaps are not efficiently merged
- There are a variety of different heaps (leftist heaps, skew heaps, binomial queues, . . .) designed to be merged
- All these heaps are real binary trees (i.e. represented by pointers rather than put in an array)
- They are slower than a binary heap because indexing is slightly slower as is creating node objects
- However, they allow merging

Merging Heaps

- One common demand on a heaps is to merge two heaps
- Unfortunately binary heaps are not efficiently merged
- There are a variety of different heaps (leftist heaps, skew heaps, binomial queues, . . .) designed to be merged
- All these heaps are real binary trees (i.e. represented by pointers rather than put in an array)
- They are slower than a binary heap because indexing is slightly slower as is creating node objects
- However, they allow merging

Merging Heaps

- One common demand on a heaps is to merge two heaps
- Unfortunately binary heaps are not efficiently merged
- There are a variety of different heaps (leftist heaps, skew heaps, binomial queues, . . .) designed to be merged
- All these heaps are real binary trees (i.e. represented by pointers rather than put in an array)
- They are slower than a binary heap because indexing is slightly slower as is creating node objects
- However, they allow merging

Merging Heaps

- One common demand on a heaps is to merge two heaps
- Unfortunately binary heaps are not efficiently merged
- There are a variety of different heaps (leftist heaps, skew heaps, binomial queues, . . .) designed to be merged
- All these heaps are real binary trees (i.e. represented by pointers rather than put in an array)
- They are slower than a binary heap because indexing is slightly slower as is creating node objects
- However, they allow merging

Other Operations

- All other operations are achieved by merging
  - ⋆ Adding an element is achieved by merging the current heap with a heap of one element
  - ⋆ Removing the minimum element is achieved by removing the root and merging the left and right tree
- For details see the course text (you won't be examined on the details of these heaps)

Other Operations

- All other operations are achieved by merging
  - ⋆ Adding an element is achieved by merging the current heap with a heap of one element
  - ⋆ Removing the minimum element is achieved by removing the root and merging the left and right tree
- For details see the course text (you won't be examined on the details of these heaps)

Other Operations

- All other operations are achieved by merging
  - ⋆ Adding an element is achieved by merging the current heap with a heap of one element
  - ⋆ Removing the minimum element is achieved by removing the root and merging the left and right tree
- For details see the course text (you won't be examined on the details of these heaps)

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - ⋆ They are used when you need a queue with priorities, e.g. in operating systems
  - ⋆ They can be used to perform pretty efficient sort
  - ⋆ They are often used for implementing greedy type algorithms
  - ⋆ One important application is in real time simulations
- There exists many extensions of heaps

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - ⋆ They are used when you need a queue with priorities, e.g. in operating systems
  - ⋆ They can be used to perform pretty efficient sort
  - ⋆ They are often used for implementing greedy type algorithms
  - ⋆ One important application is in real time simulations
- There exists many extensions of heaps

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - ⋆ They are used when you need a queue with priorities, e.g. in operating systems
  - ⋆ They can be used to perform pretty efficient sort
  - ⋆ They are often used for implementing greedy type algorithms
  - ⋆ One important application is in real time simulations
- There exists many extensions of heaps

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - ⋆ They are used when you need a queue with priorities, e.g. in operating systems
  - ⋆ They can be used to perform pretty efficient sort
  - ⋆ They are often used for implementing greedy type algorithms
  - ⋆ One important application is in real time simulations
- There exists many extensions of heaps

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - ⋆ They are used when you need a queue with priorities, e.g. in operating systems
  - ⋆ They can be used to perform pretty efficient sort
  - ⋆ They are often used for implementing greedy type algorithms
  - ⋆ One important application is in real time simulations
- There exists many extensions of heaps

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - ⋆ They are used when you need a queue with priorities, e.g. in operating systems
  - ⋆ They can be used to perform pretty efficient sort
  - ⋆ They are often used for implementing greedy type algorithms
  - ⋆ One important application is in real time simulations
- There exists many extensions of heaps

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - ⋆ They are used when you need a queue with priorities, e.g. in operating systems
  - ⋆ They can be used to perform pretty efficient sort
  - ⋆ They are often used for implementing greedy type algorithms
  - ⋆ One important application is in real time simulations
- There exists many extensions of heaps

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - ⋆ They are used when you need a queue with priorities, e.g. in operating systems
  - ⋆ They can be used to perform pretty efficient sort
  - ⋆ They are often used for implementing greedy type algorithms
  - ⋆ One important application is in real time simulations
- There exists many extensions of heaps