CSC230

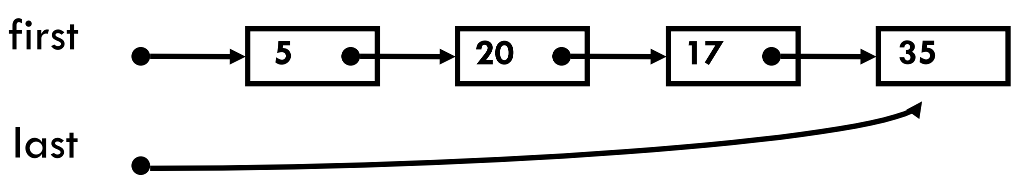
Intro to C++ Lecture 26

Outline

- Priority Queue
- Heap

Stacks and Queues as Lists

- Stack (LIFO) implemented as list
  - **push()**, **pop()** from front of list
- Queue (FIFO) implemented as list
  - **push()** on back of list, **pop()** from front of list
- All operations are?
  - O(1)



Priority Queue

- Data items are **Comparable**
- *Each* element has *its* priority
- **pop()** returns the element with the highest priority
- break ties arbitrarily

Priority Queue Examples

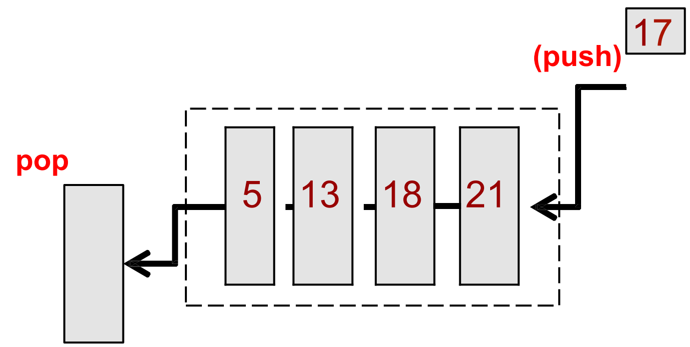
- Scheduling jobs to run on a computer
  - default priority = arrival time
  - priority can be changed by operator
- Scheduling events to be processed by an event handler
  - priority = time of occurrence
- Airline check-in
  - first class, business class, coach
  - FIFO within each class

Difference between Queue and Priority Queue

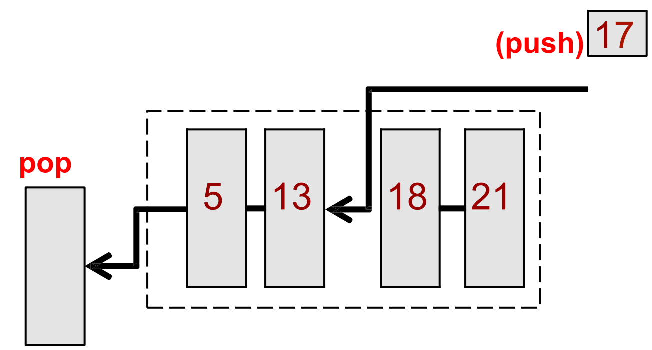
- **Traditional Queue:**
  - Push to the back
  - Pop from the front
  - Push/Pop do not care about the values

- **Priority Queue:**
  - Data are ordered
    - Increasing or decreasing
    - Hospital ER service

  - Items can arrive in arbitrary order
  - When remove an item, we always get the minimum or maximum depending on the implementation



Traditional Queue



Priority Queue

STL Priority Queue

- Decreasing-order by default
- Methods:
  – push(new_item)
  – pop(): removes
  – top() return top item
  – size()
  – empty()

```
// priority_queue::push/pop
#include <iostream>
#include <queue>

using namespace std;

int main ()
{
priority_queue<int> myPQ;
myPQ.push(10);
myPQ.push(80);
myPQ.push(15);
myPQ.push(30);
cout << "Popping out elements...";
while (!myPQ.empty()) {
cout << myPQ.top() << endl;
myPQ.pop();
}
return 0;
}
```

How about increasing order?


```
// priority_queue::push/pop
#include <iostream>
#include <queue>
using namespace std;
int main ()
{
  priority_queue<int,vector<int>,greater<int> > myPQ;
  myPQ.push(30);
  myPQ.push(100);
  myPQ.push(25);
  cout<< "Popping out elements..." << endl;
  while (!myPQ.empty()) {
    cout << myPQ.top() << endl;
    myPQ.pop();
  }
}
```

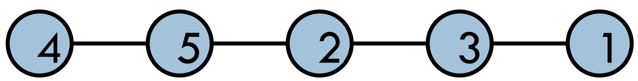
Data type of inserted data

Data type used by priority queue to store data

Increasing order (less denotes decreasing order)

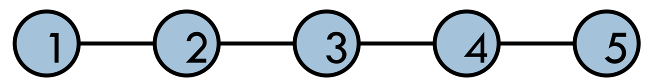
List-based Priority Queue

- Unsorted list implementation
  - Store the items of the priority queue in a list-based sequence, in arbitrary order



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

- sorted list implementation
  - Store the items of the priority queue in a sequence, sorted by key



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

Priority Queues as Lists

- Maintain as unordered list
  - **push()** puts new element at front ?
    - O(1)
  - **pop()** must **search** the list ?
    - O(n)

- Maintain as ordered list
  - **push()** must **search** the list ?
    - O(n)
  - **pop()** gets element at front ?
    - O(1)

- In either case, $O(n^2)$ to process n elements

Can we do better?

Important Special Case

- Fixed number of priority levels 0,...,p – 1
- FIFO within each level
- Example: airline check-in

- **push()**– insert in appropriate queue – O(1)
- **pop()**– must find a nonempty queue – O(p)

- How to implement Priority Queue ?

Heaps

- A ***heap*** is a concrete data structure that can be used to implement priority queues
- Gives better complexity than either ordered or unordered list implementation:
  - **push():** O(log n)
  - **pop():** O(log n)
- O(n log n) to process n elements
- Do not confuse with *heap memory*, where C++ dynamically allocates space – different usage of the word *heap*

Heaps

- Binary tree with data at each node
- Satisfies the *Heap Order Invariant*:

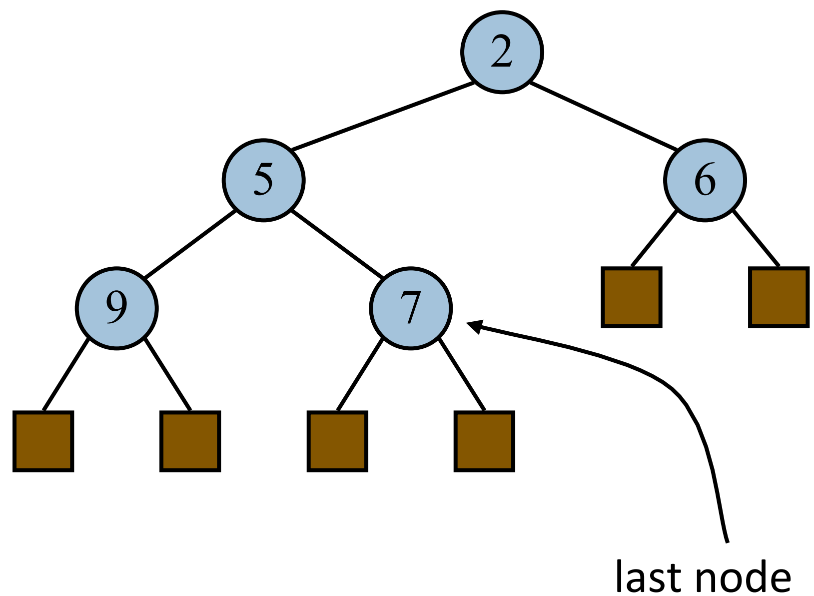
The least or highest priority element of any subtree is found at the root of that subtree

- Size of the heap is "fixed" at *n*. (But can usually double n if heap fills up)

What is a heap?

- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
    - at depth $h - 1$, the internal nodes are to the left of the leaf nodes

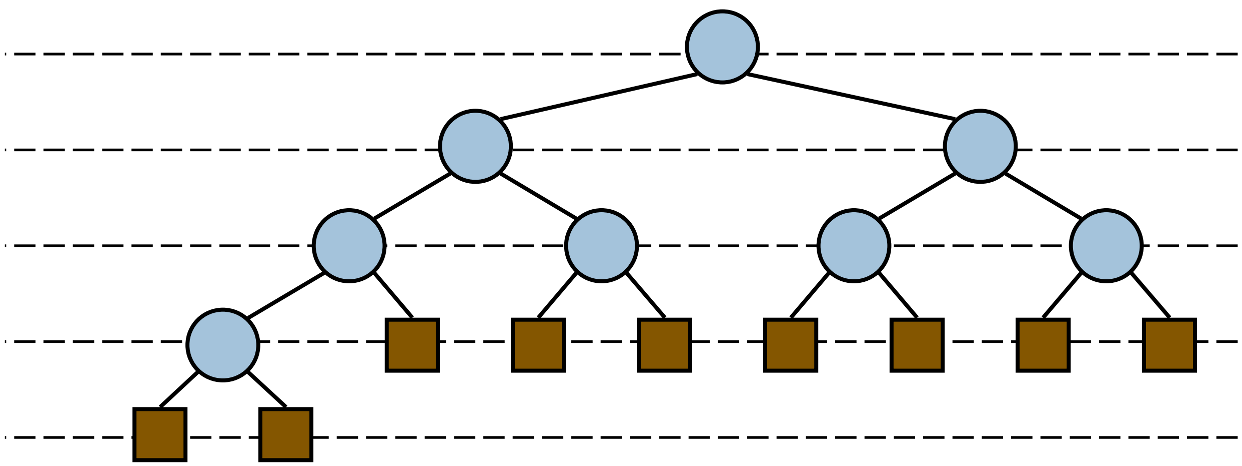
- The last node of a heap is the rightmost internal node of depth $h - 1$



Height of a Heap

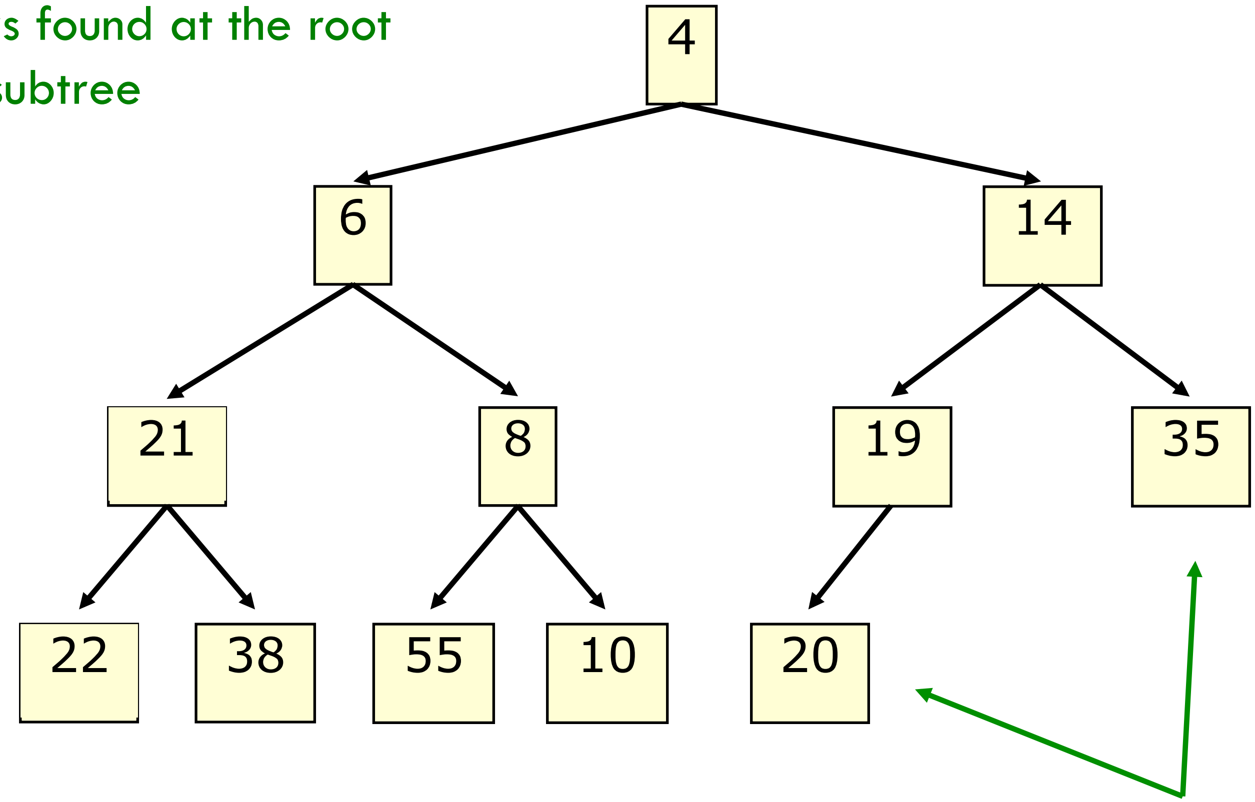
- Theorem: A heap storing n keys has height $O(\log n)$
- Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h - 2$ and at least one key at depth $h - 1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$

depth	keys
0	1
1	2
$h-2$	2^{h-2}
$h-1$	1



Heaps

Least element in any subtree
is always found at the root
of that subtree



Note: 19, 20 < 35: we can often find
smaller elements deeper in the tree!

Examples of Heaps

- Ages of people in family tree

– parent is always older than children, but you can have an uncle who is younger than you (larger number will be on the root)

- Salaries of employees of a company

– bosses generally make more than subordinates, but a VP in one subdivision may make less than a Project Supervisor in a different subdivision

Balanced Heaps

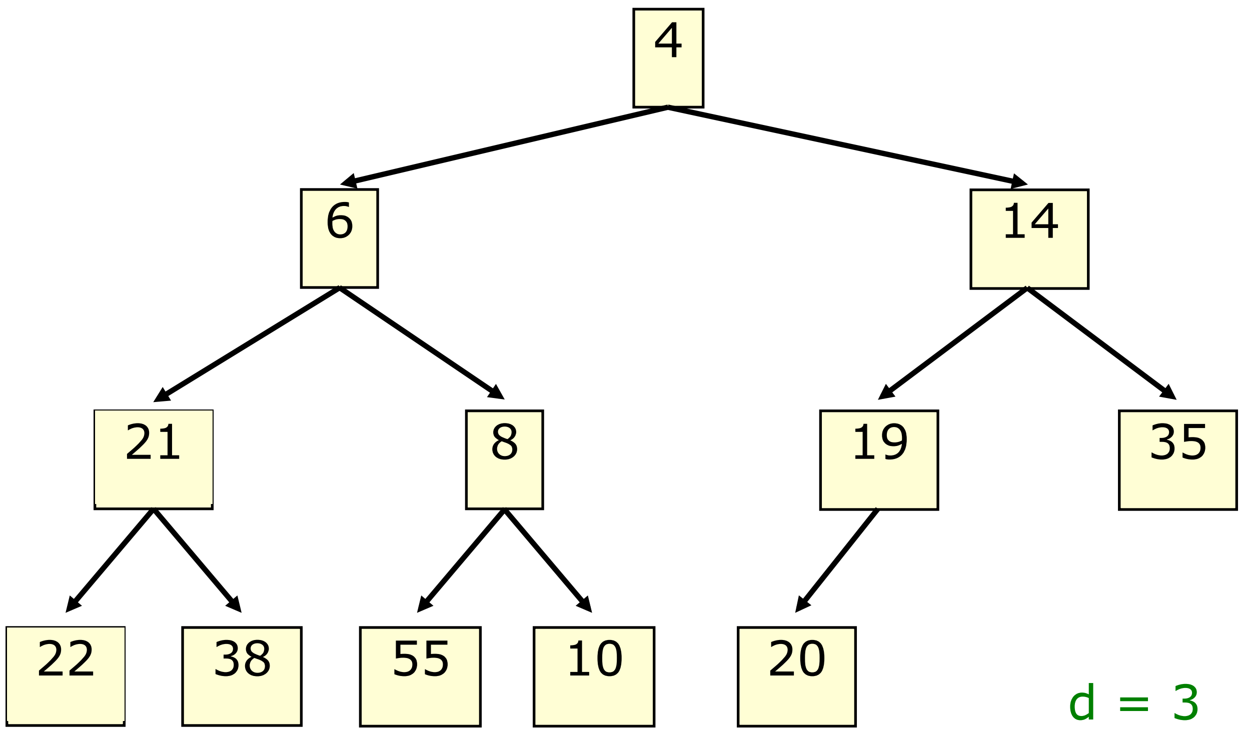
These add two restrictions:

1. Any node of depth $< d - 1$ has **exactly** 2 children, where d is the height of the tree

– implies that any two maximal paths (path from a root to a leaf) are of length d or $d - 1$, and the tree has at least 2^d nodes

- All **maximal** paths of length d are to the left of those of length $d - 1$

Example of a Balanced Heap



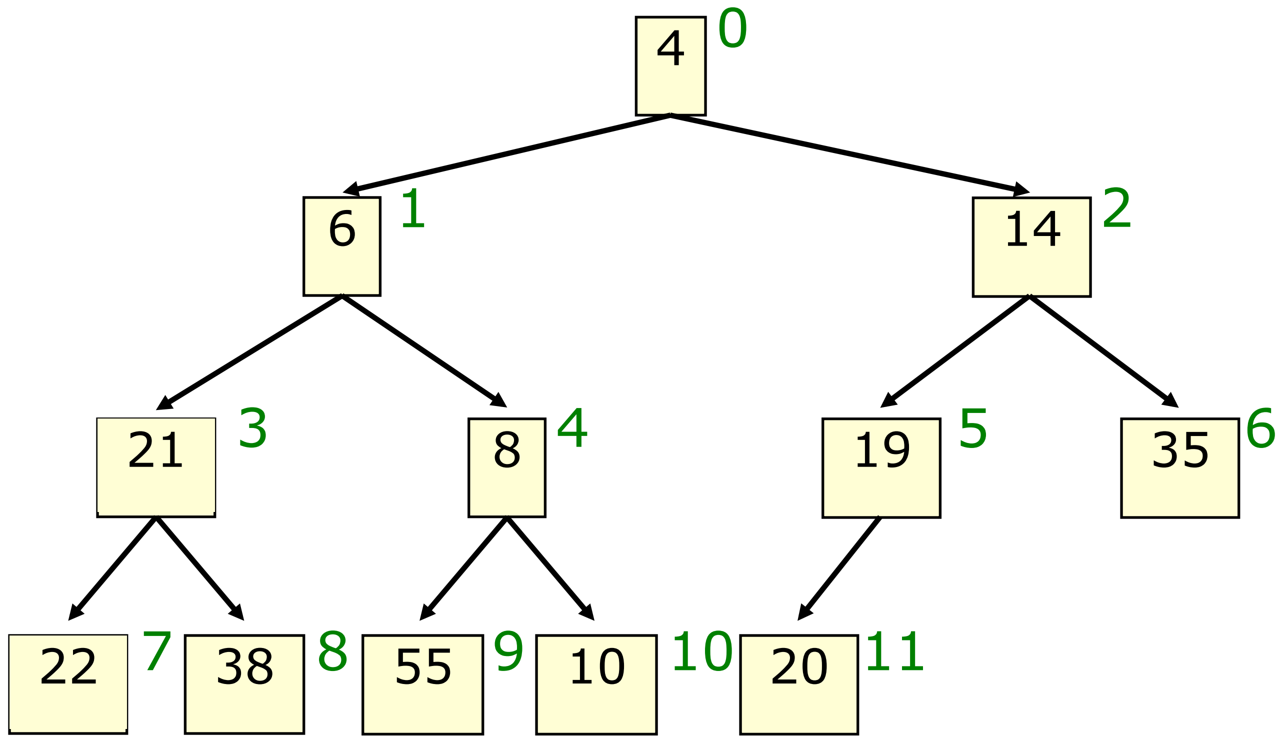
Store in an Array or Vector

- Elements of the heap are stored in the array in order, going across each level from left to right, top to bottom

- The children of the node at array index n are found at $2n + 1$ and $2n + 2$

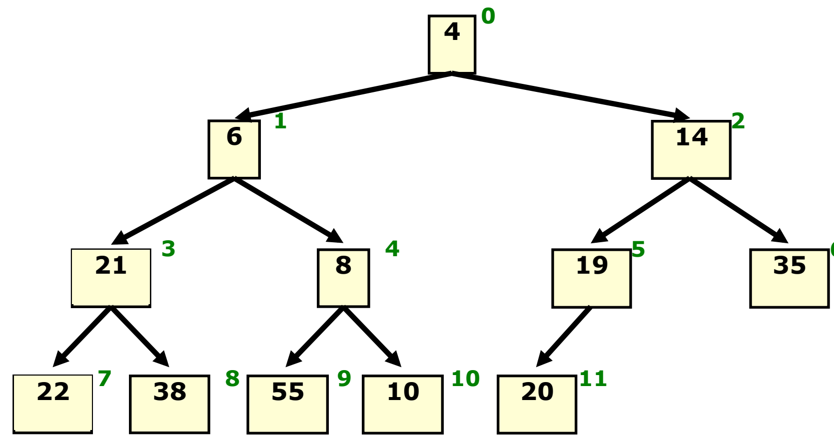
- The parent of node n is found at $(n - 1)/2$

Store in an Array or Vector



children of node n are found at 2n + 1 and 2n + 2

Store in an Array or Vector



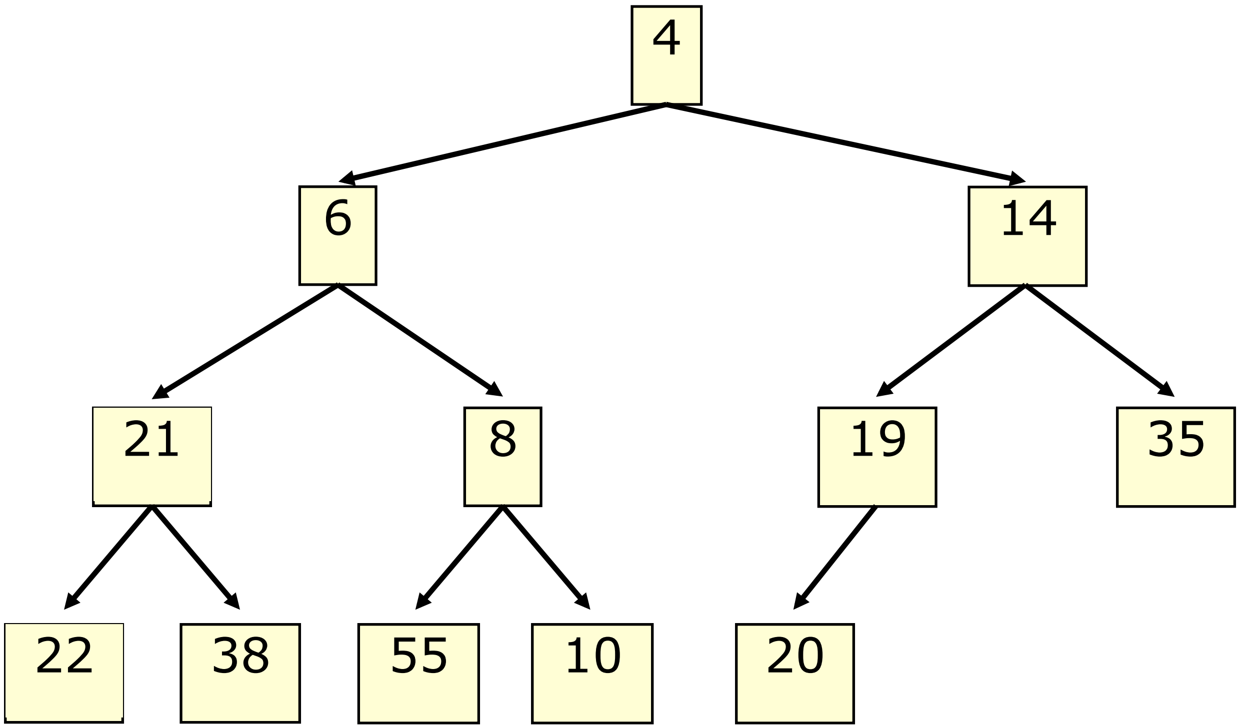
0	1	2	3	4	5	6	7	8	9	10	11
4	6	14	21	8	19	35	22	38	55	10	20

children of node n are found at 2n + 1 and 2n + 2

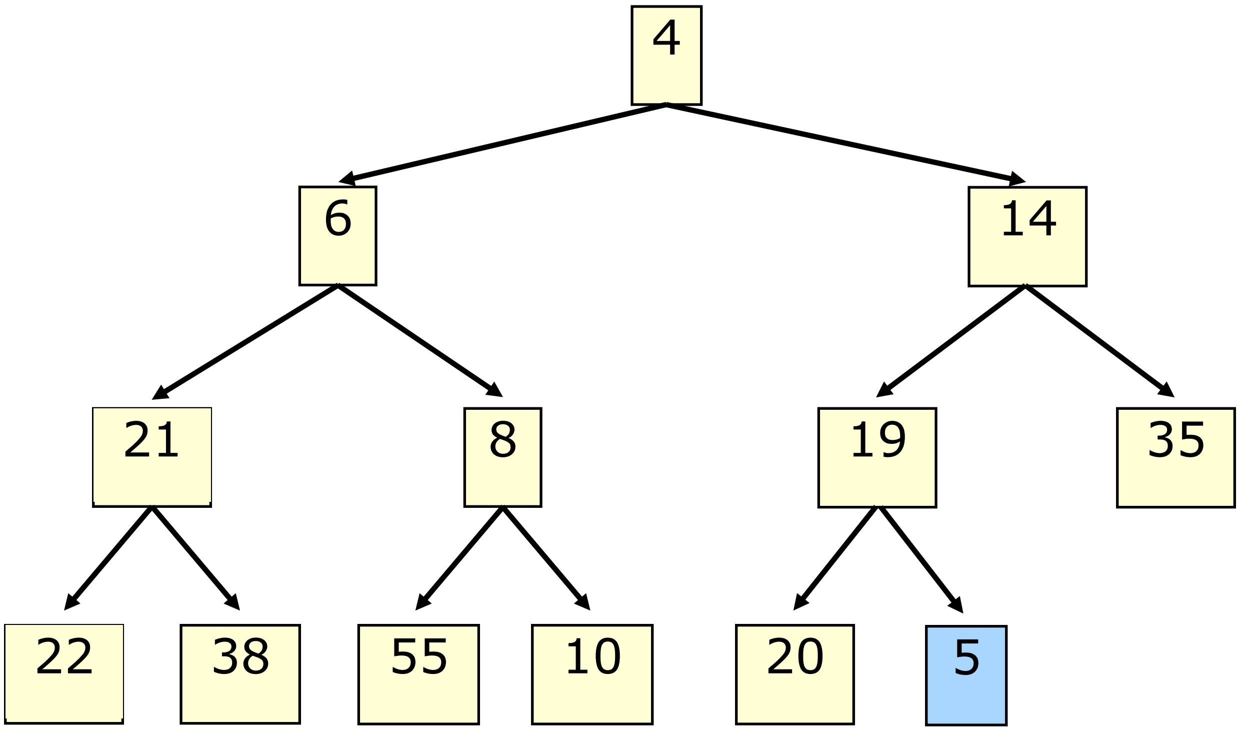
`push()`

- Put the new element at the **end** of the array
- If this violates heap order because it is smaller than its parent, swap it with its parent
- Continue swapping it up until it finds its rightful place
- The heap invariant is maintained!

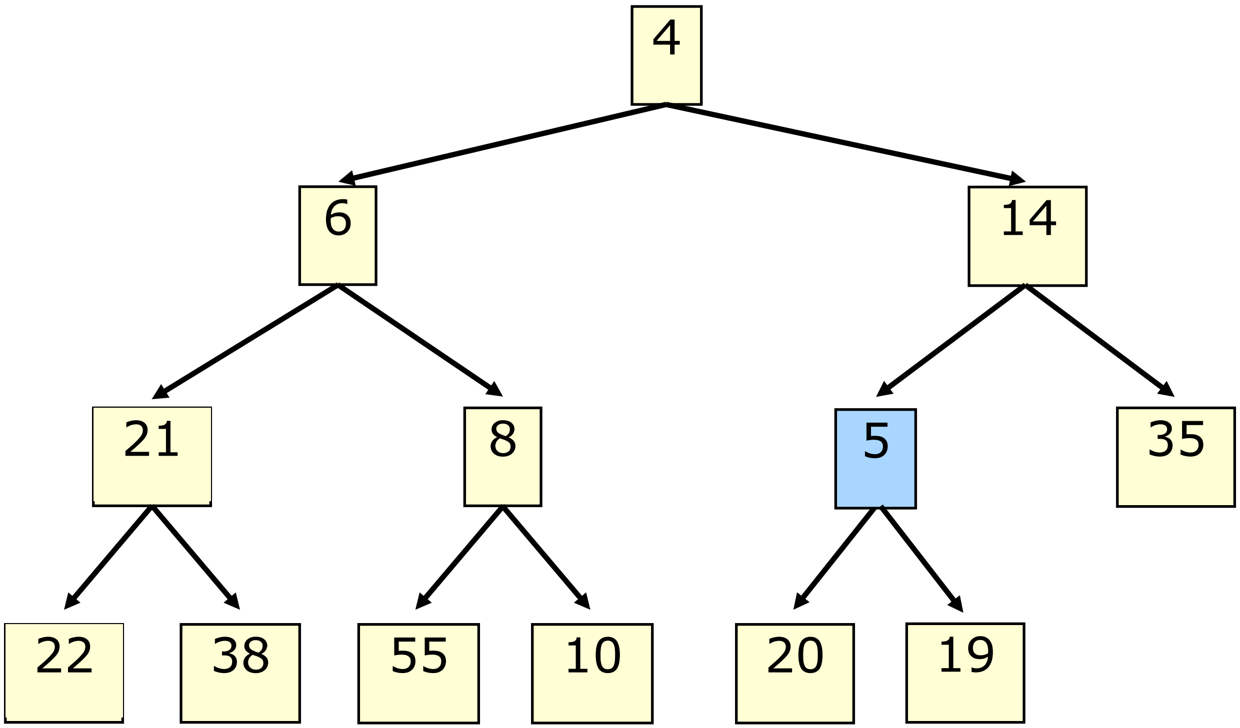
push()



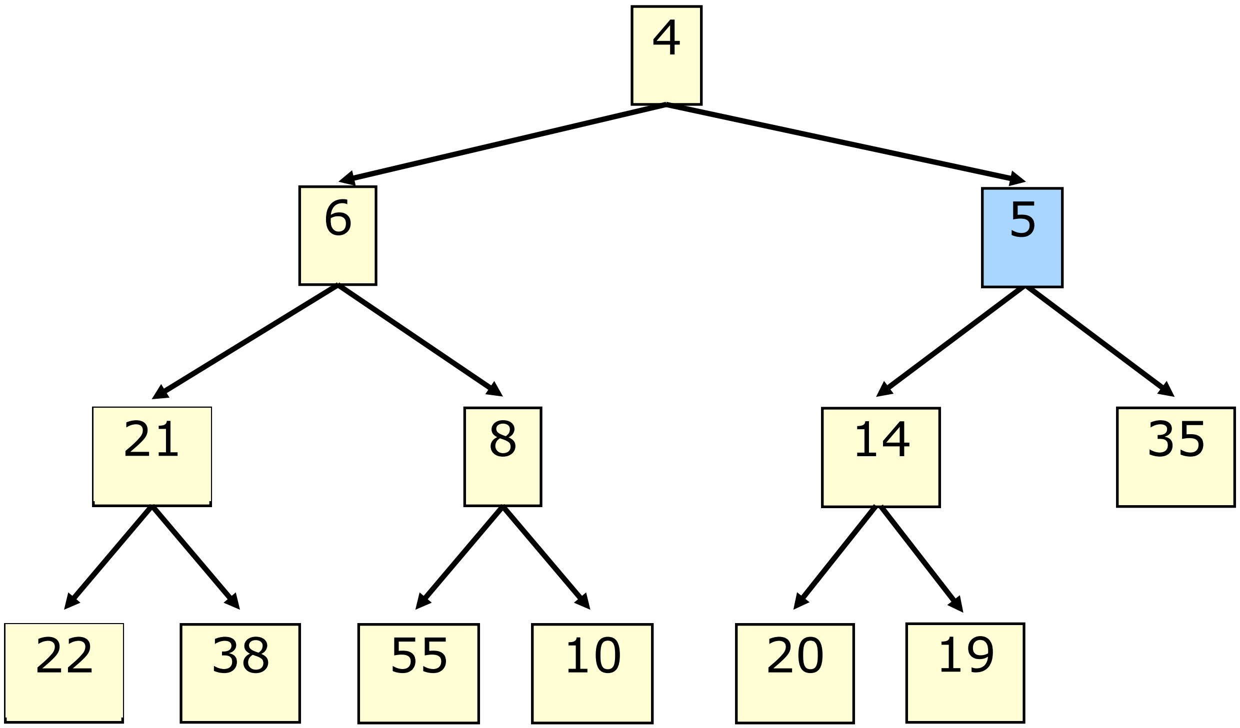
push()



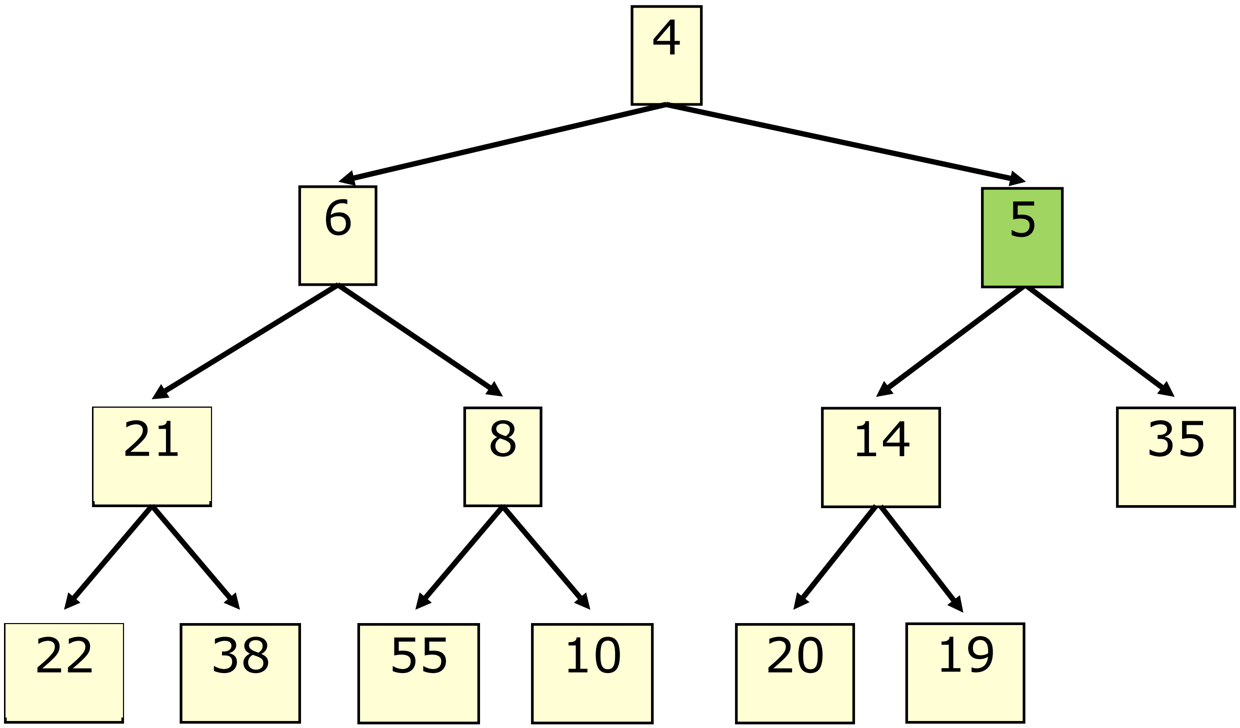
push()



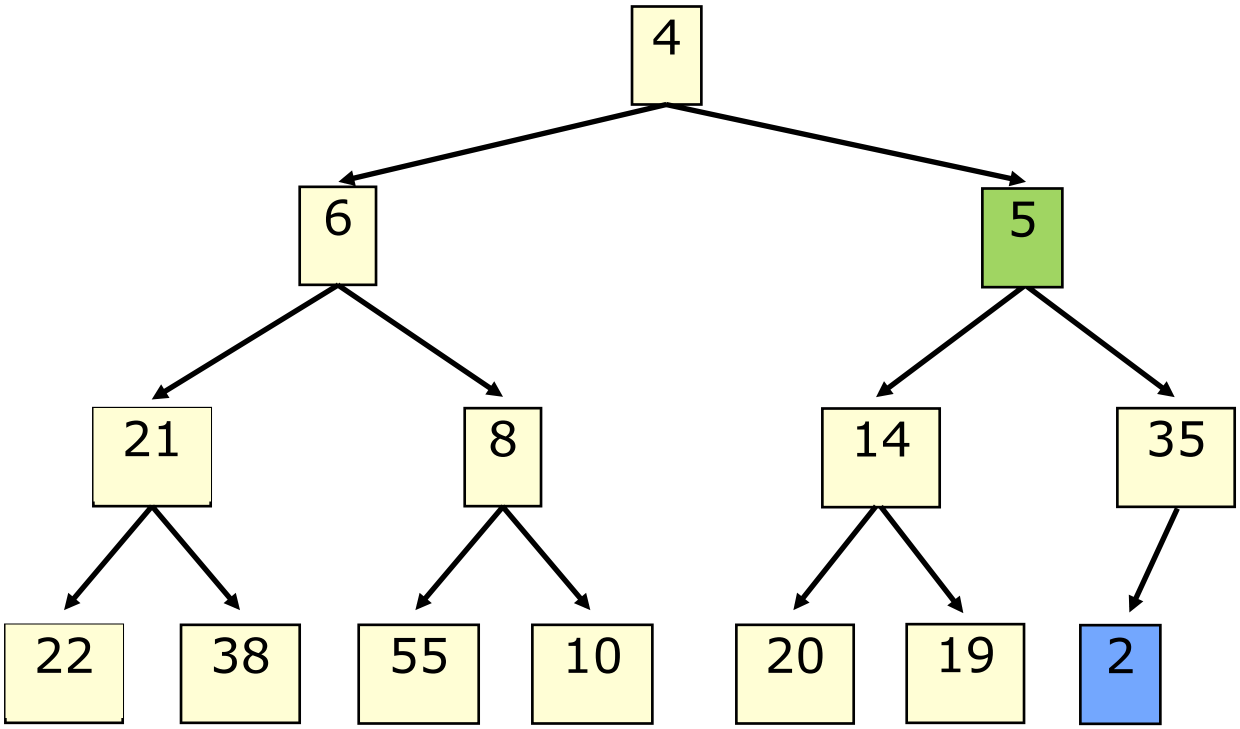
push()



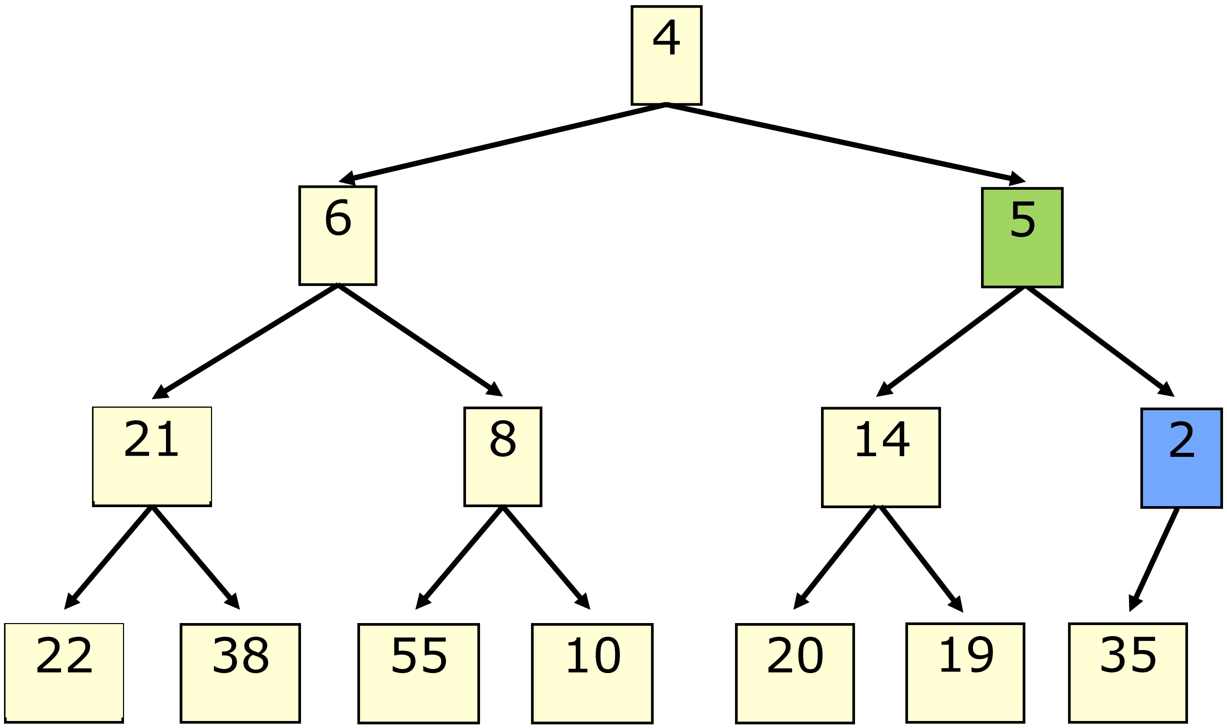
push() (upheap)



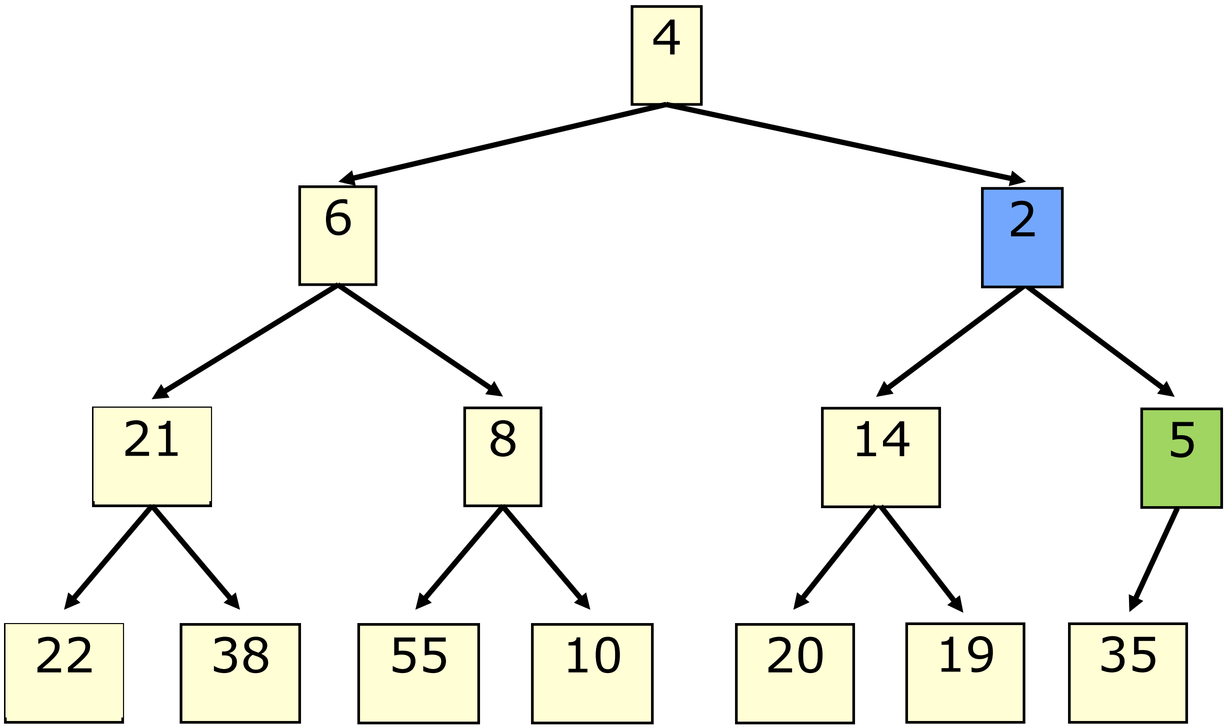
push() (upheap)



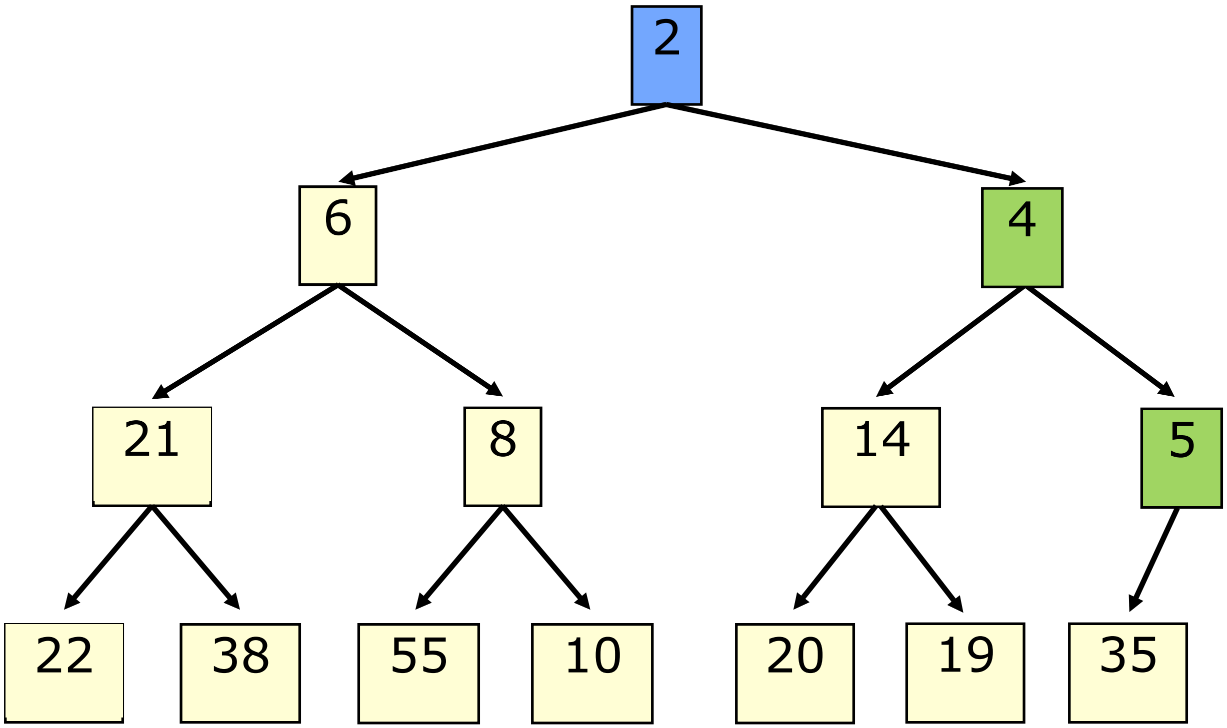
push() (upheap)



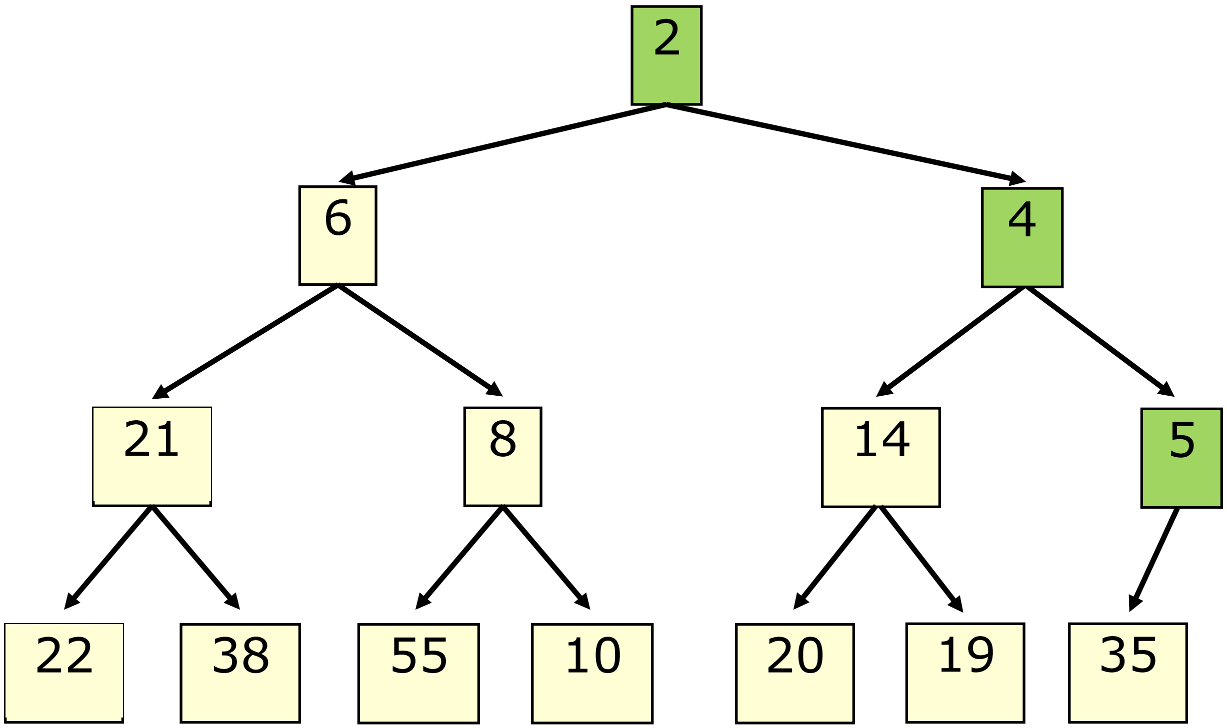
push() (upheap)



push() (upheap)



`push()` (upheap)



`push()` (upheap)

- Time is O(log n), since the tree is balanced

– size of tree is exponential as a function of depth

– depth of tree is logarithmic as a function of size

push()

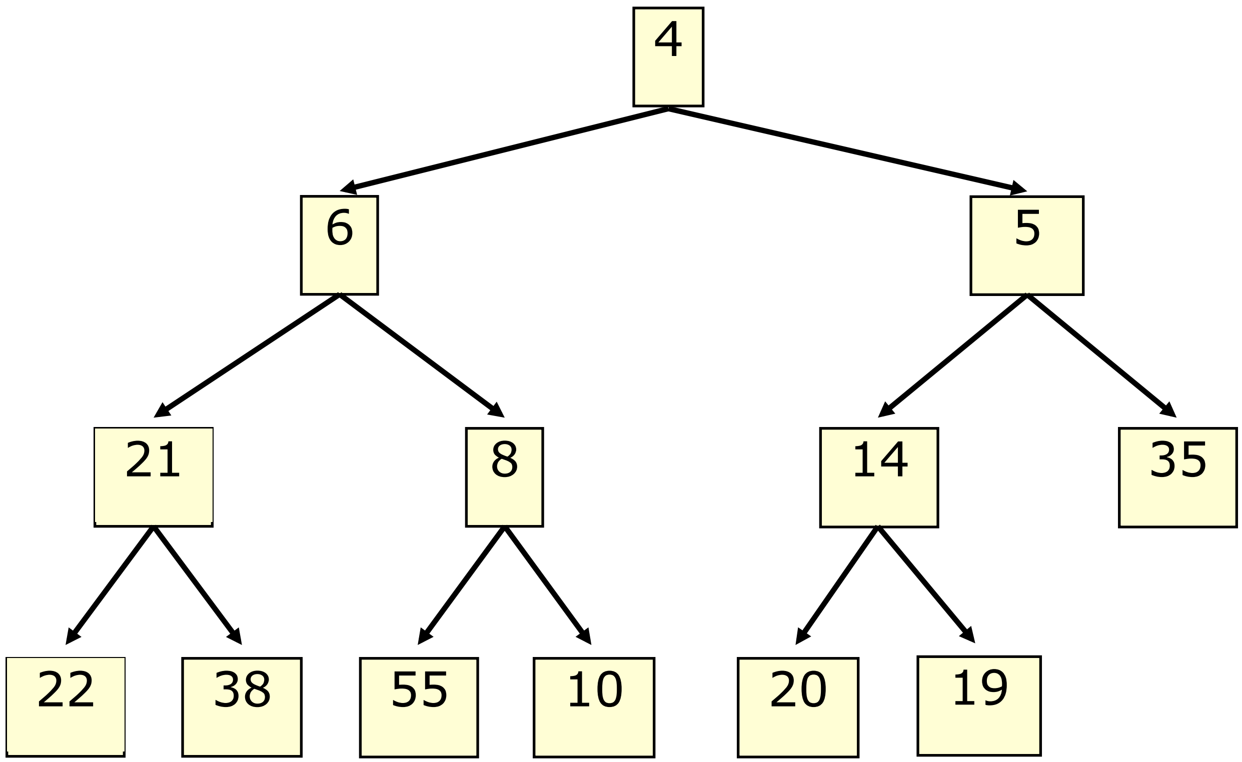

```
void ArrayMinHeap<T>::push(const T& item)
{
items_.push(item); rotateUp(items_.size()-1);
}

void rotateUp(int loc)
{
// could be implemented recursively
int parent = loc/2;
while(parent >= 1 &&
items_[loc] < items_[parent] )
{       swap(items_[parent], items_[loc]);
        loc = parent;
        parent = loc/2;
}
}
```

`pop()` (downheap)

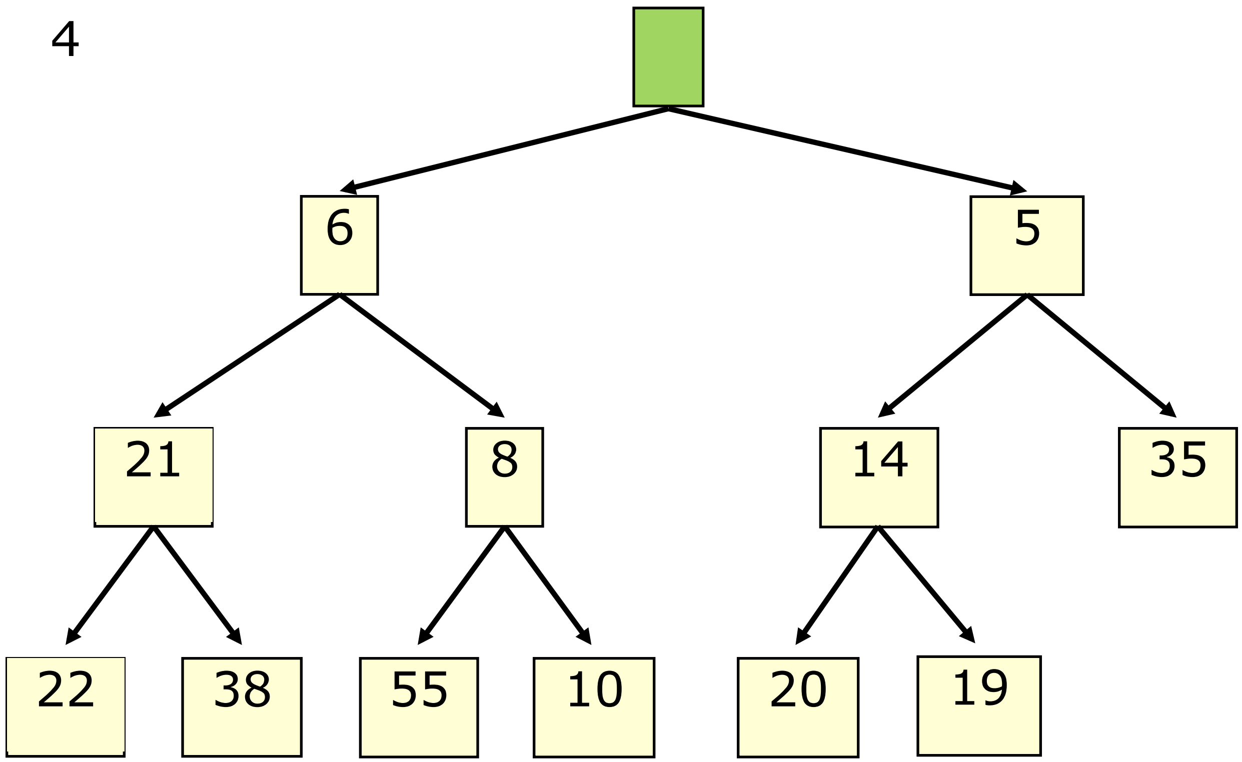
- Remove the least element – it is at the root
- This leaves a hole at the root – fill it in with the **last element** of the array
- If this violates heap order because the root element is too big, swap it down with the smaller of its children
- Continue swapping it down until it finds its rightful place
- The heap invariant is maintained!

pop() (downheap)



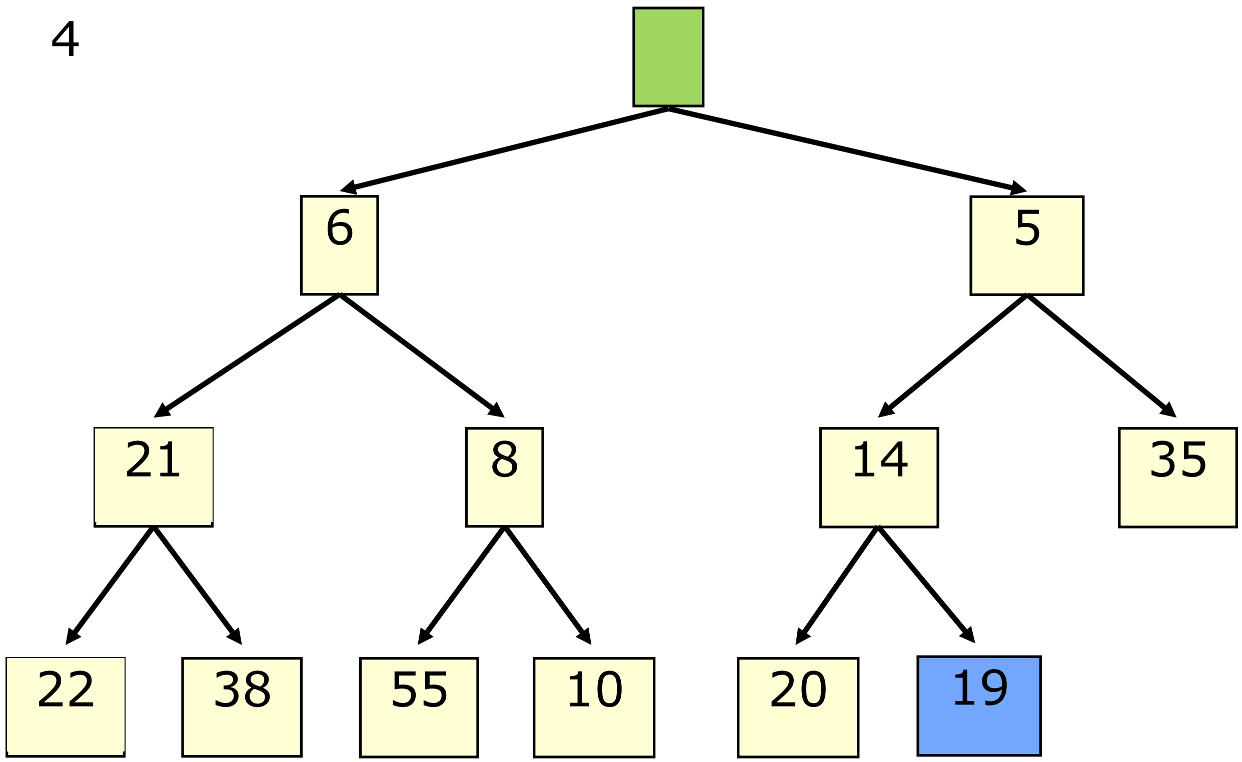
pop() (downheap)

4



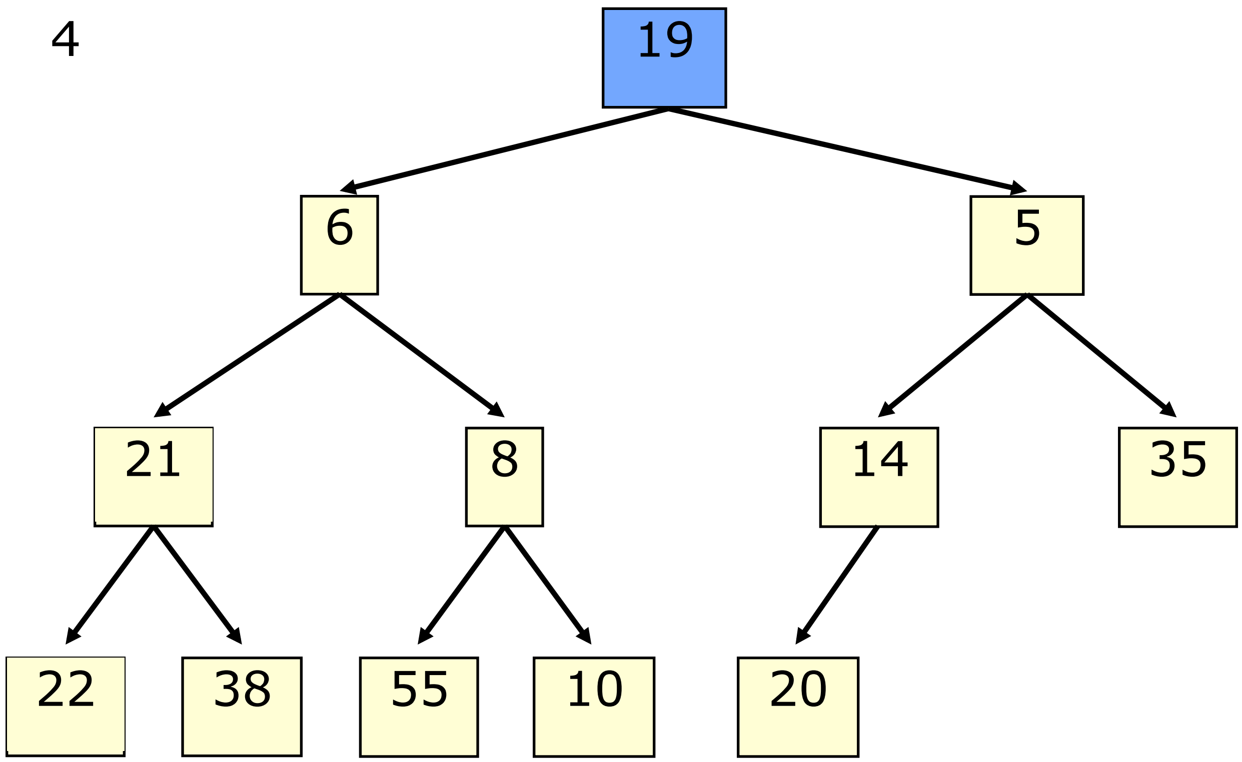
pop() (downheap)

4



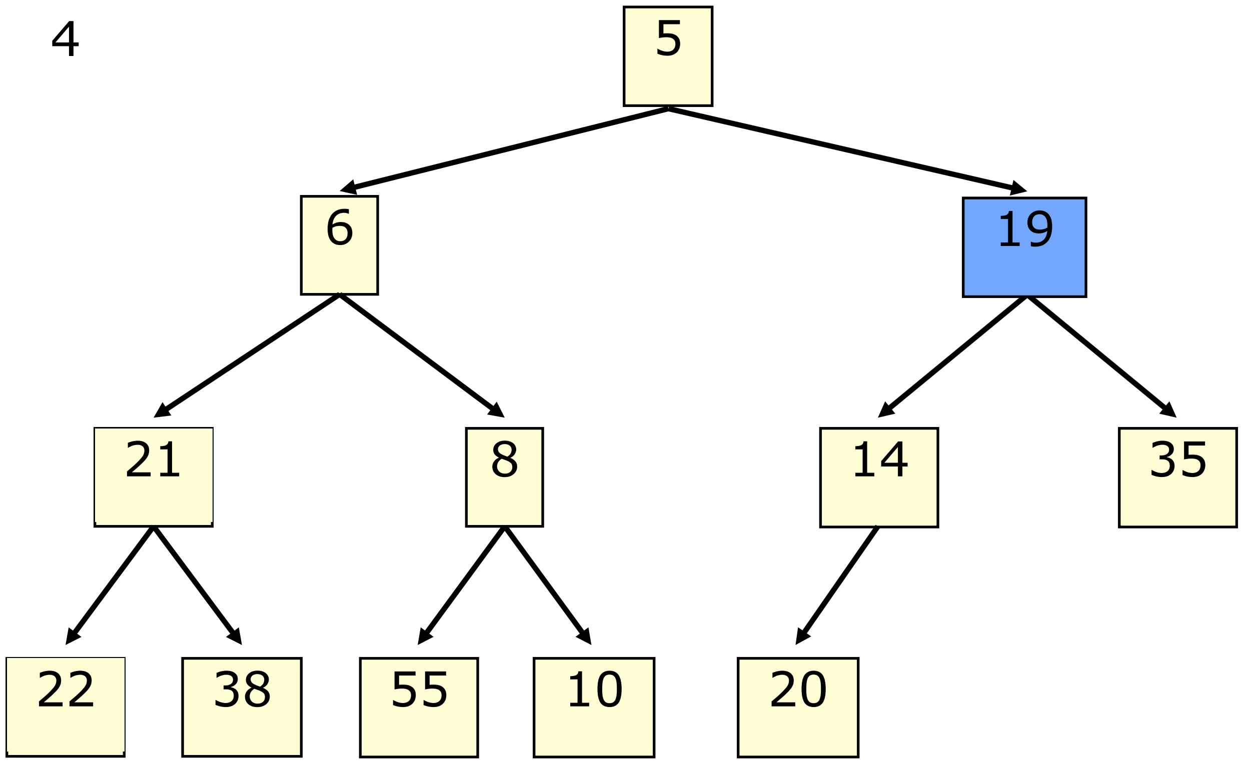
pop() (downheap)

4

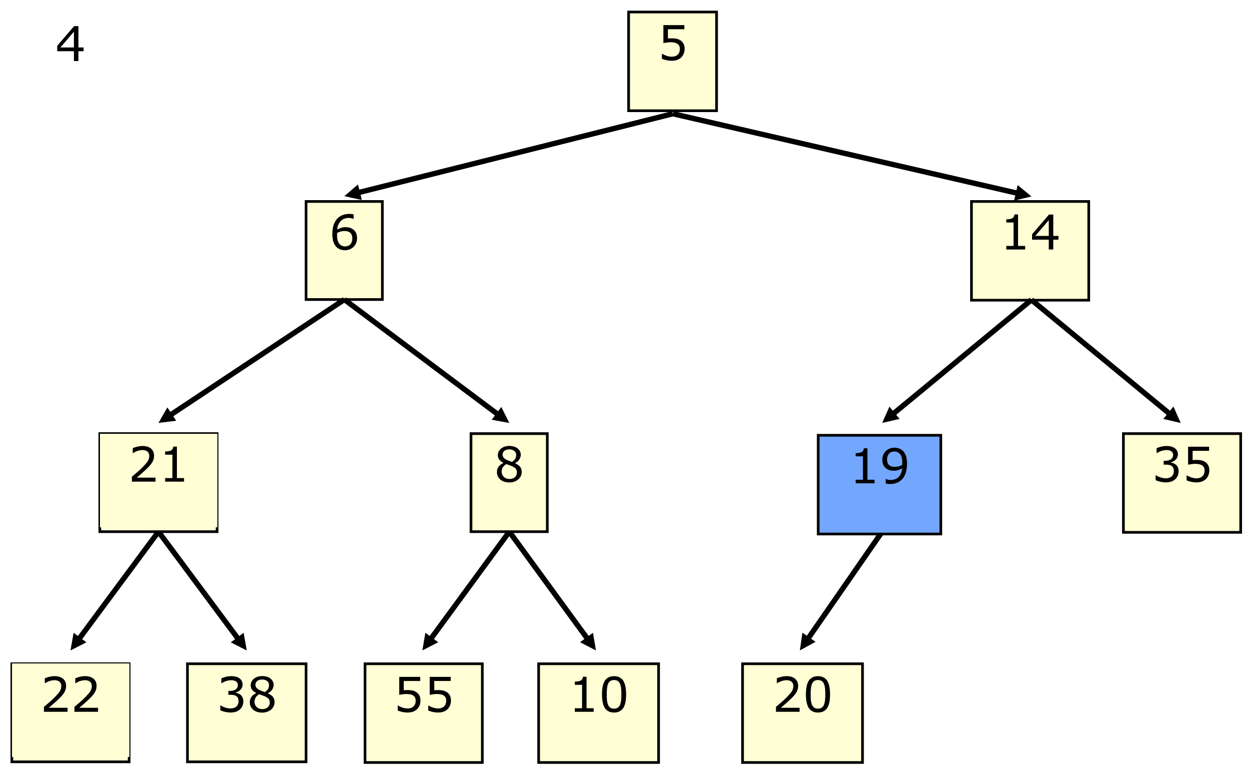


pop() (downheap)

4

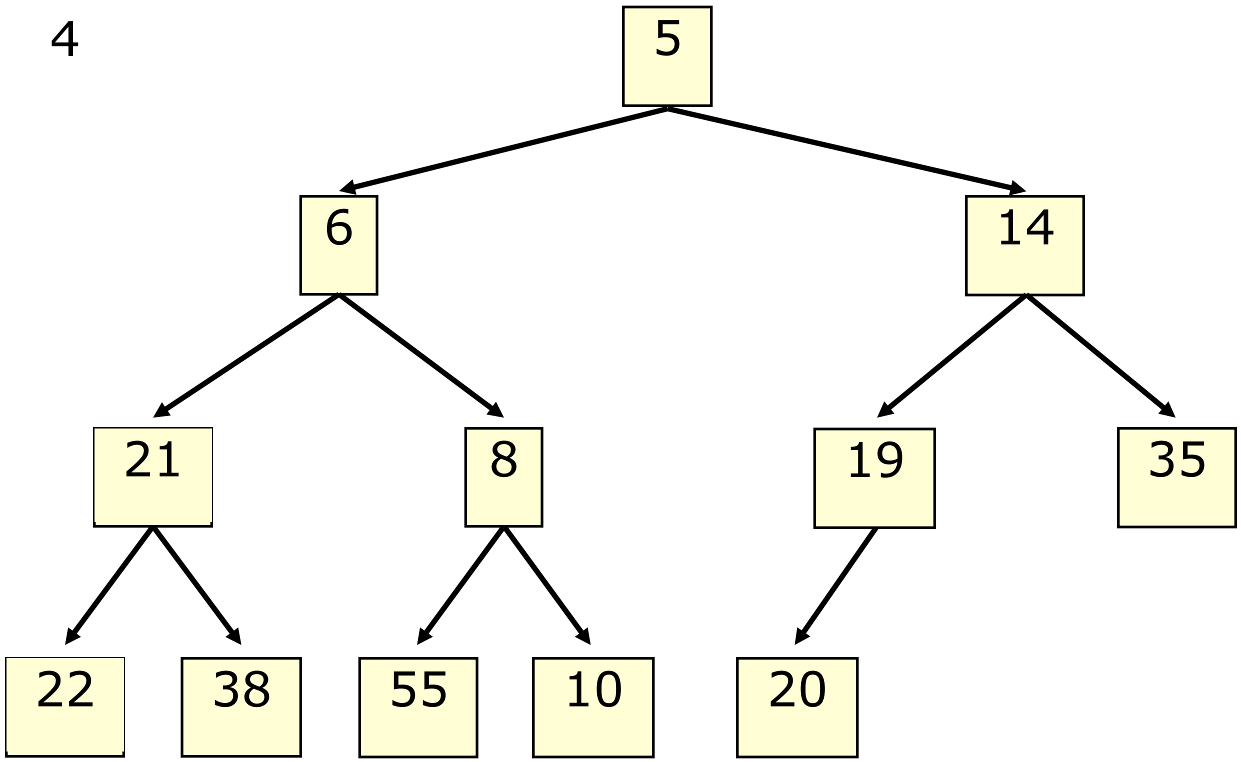


pop() (downheap)



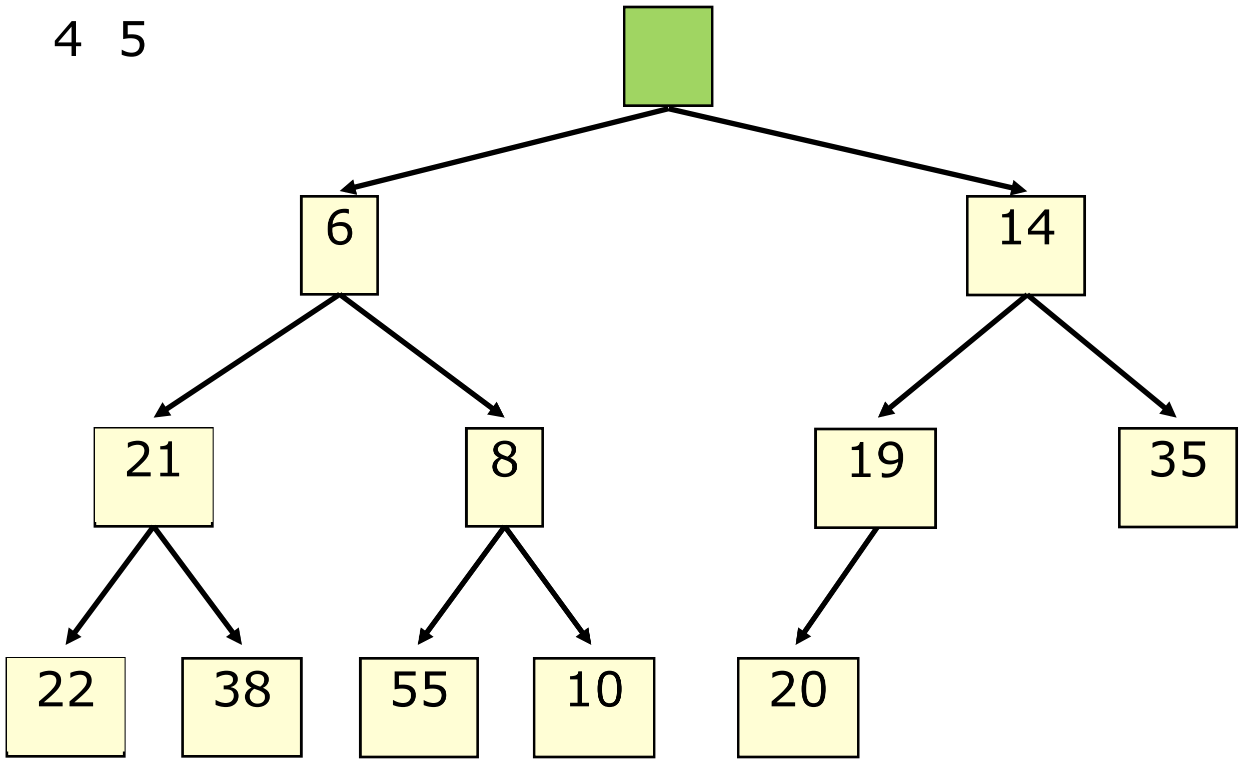
pop() (downheap)

4



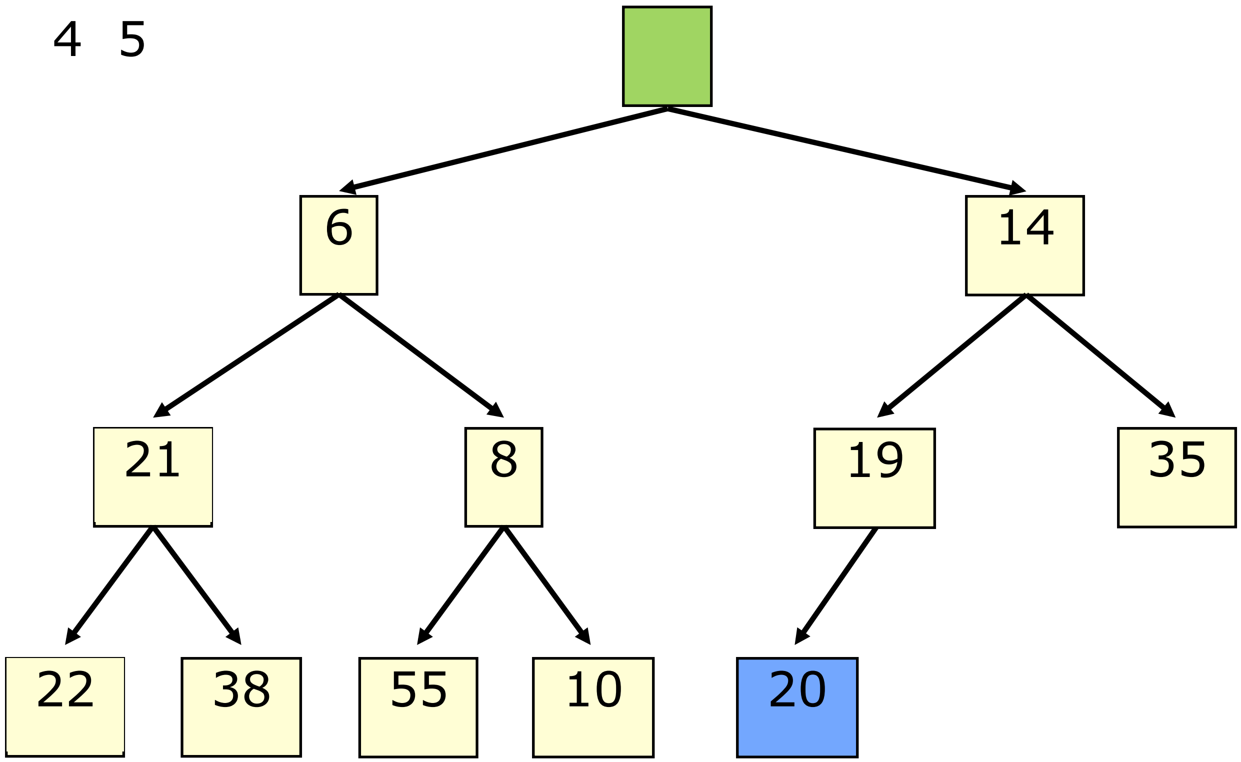
pop() (downheap)

4 5



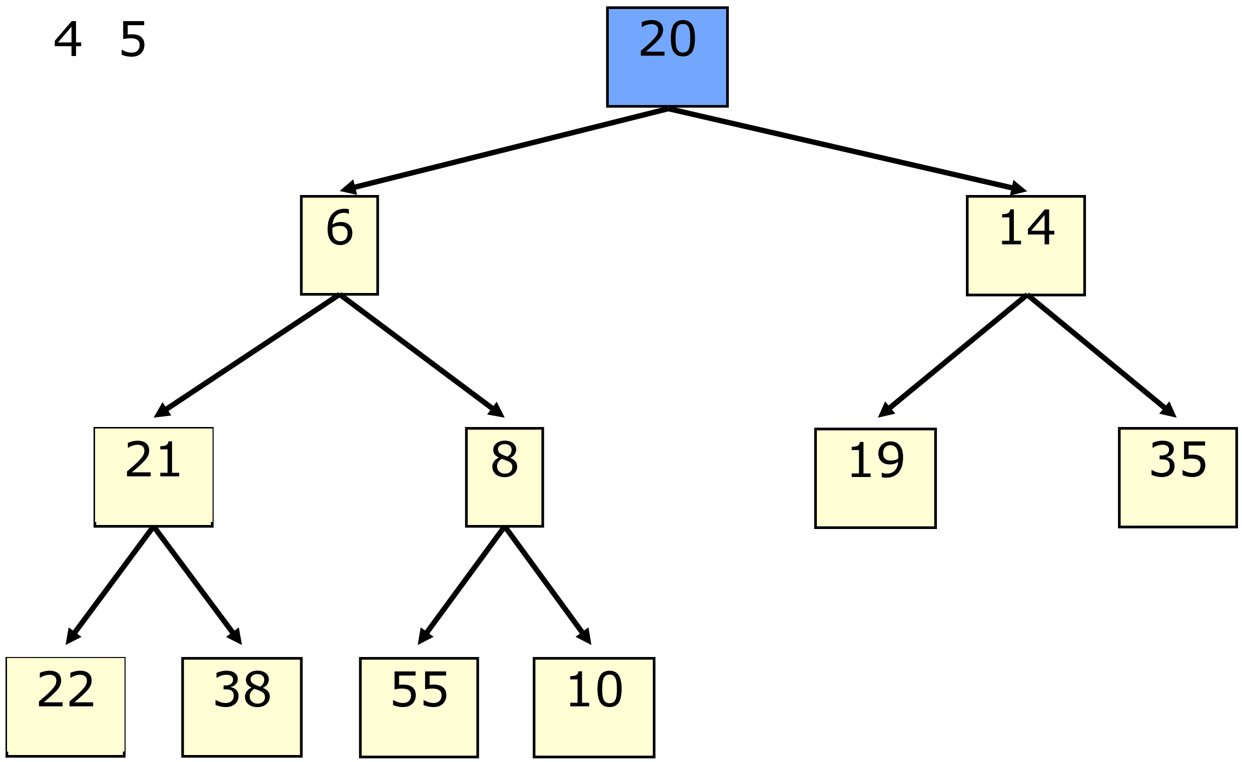
pop()   (downheap)

4 5

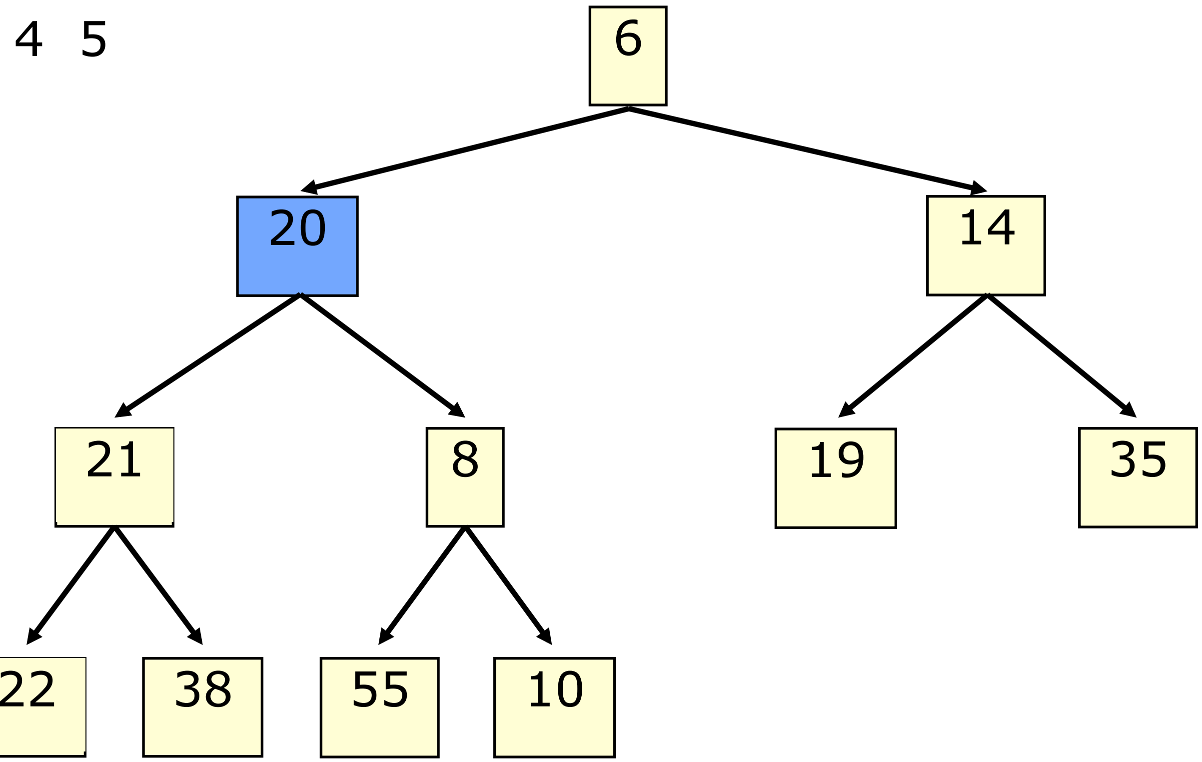


pop()   (downheap)

4   5

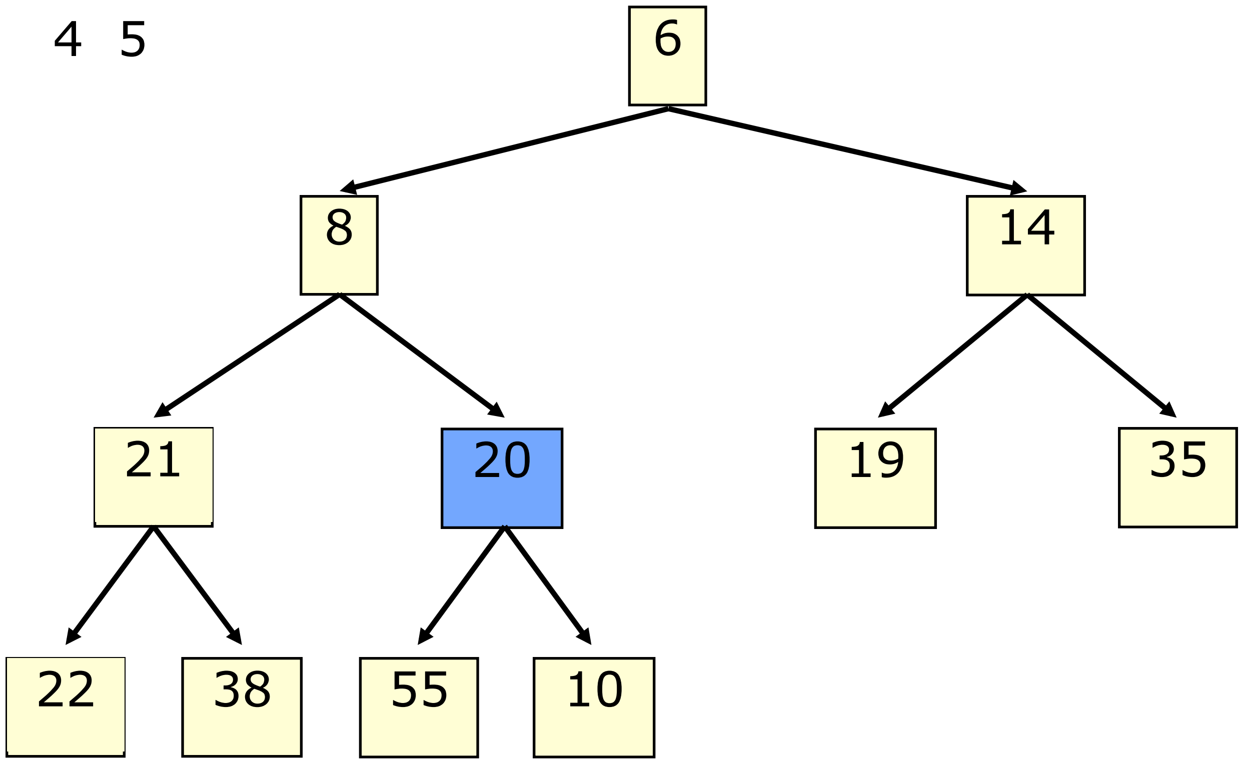


pop() (downheap)



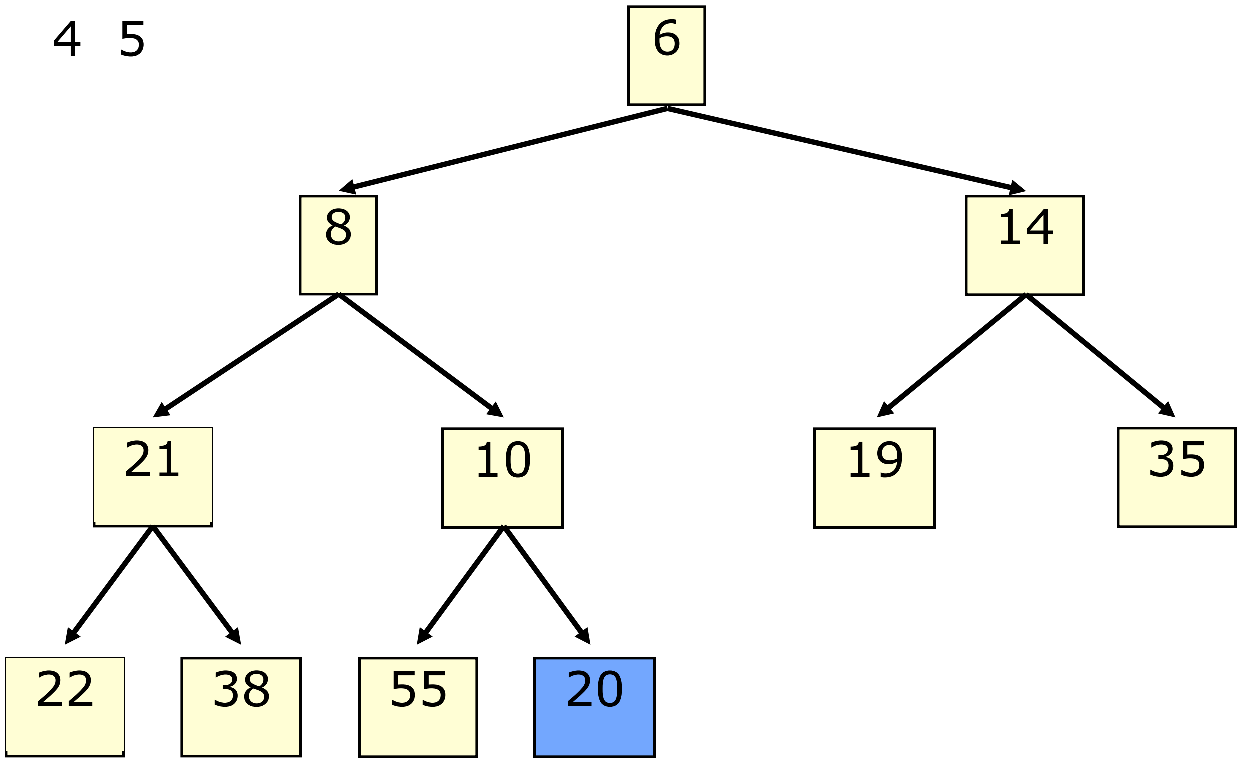
pop()

4 5



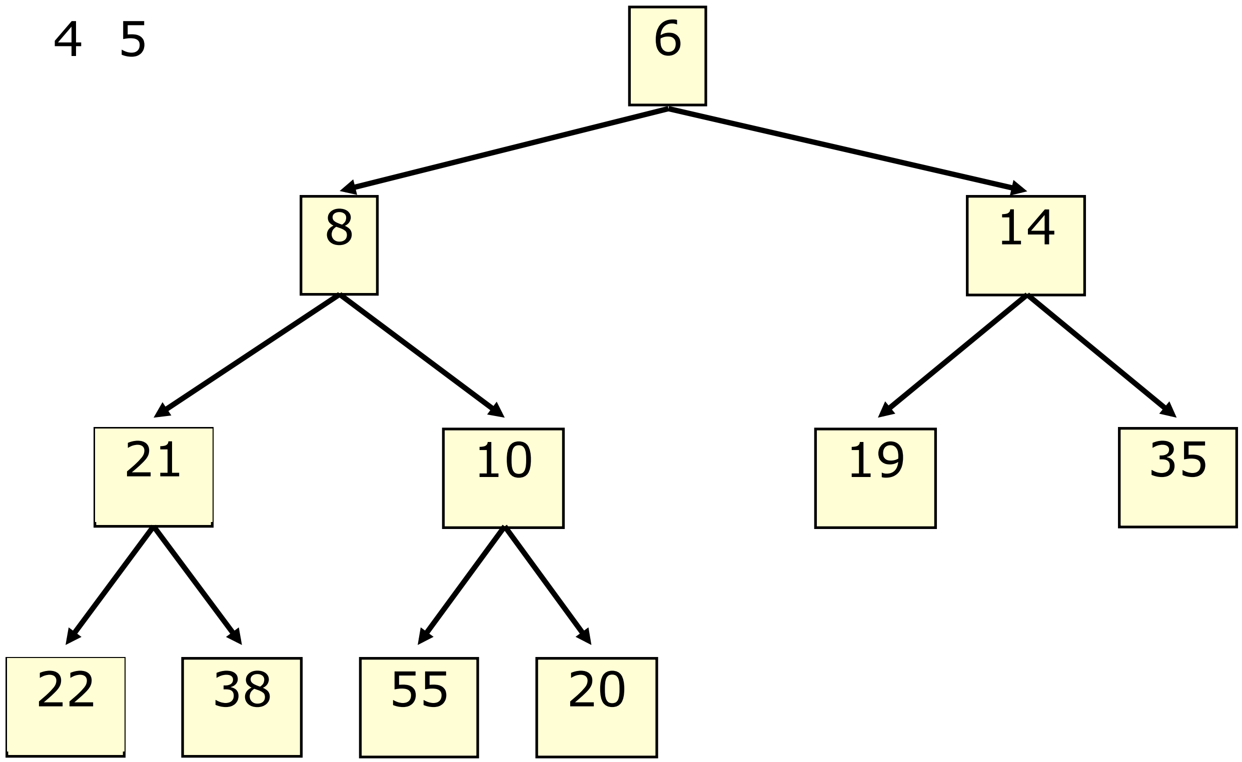
pop()

4 5



pop()

4 5



pop()

- Time is O(log n), since the tree is balanced

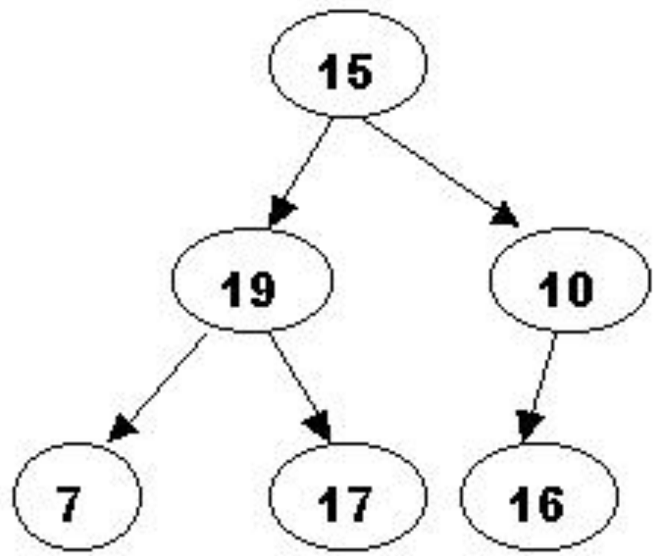
pop()


```
void ArrayMinHeap<T>::pop()
{ items_[1] = items_.back();
 rotateDown(1);
}
```

```
void ArrayMinHeap<T>::rotateDown(int idx)
{
if(idx == leaf node) return;
int smallerChild = 2*idx; // start w/ left
 if(right child exists) {
int rChild = smallerChild+1; if(items_[rChild]
< items_[smallerChild])
smallerChild = rChild;
} }
if(items_[idx] > items_[smallerChild]
){ swap(items_[idx], items_[smallerChild]);
 heapify(smallerChild);
} }
```

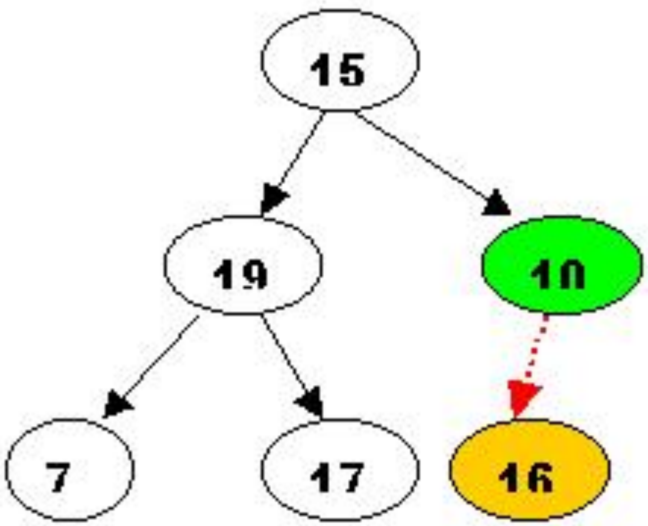
Build the Heap Tree

15	19	10	7	17	16



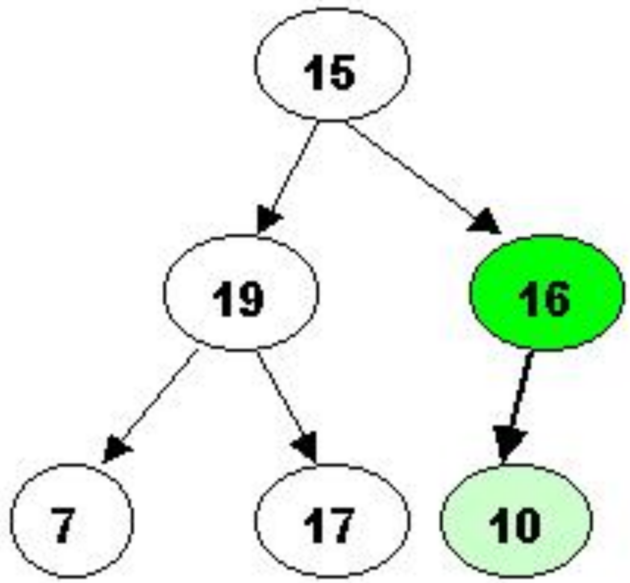
Build the Heap Tree

15	19	10	7	17	16

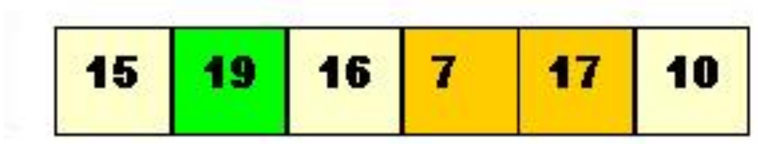


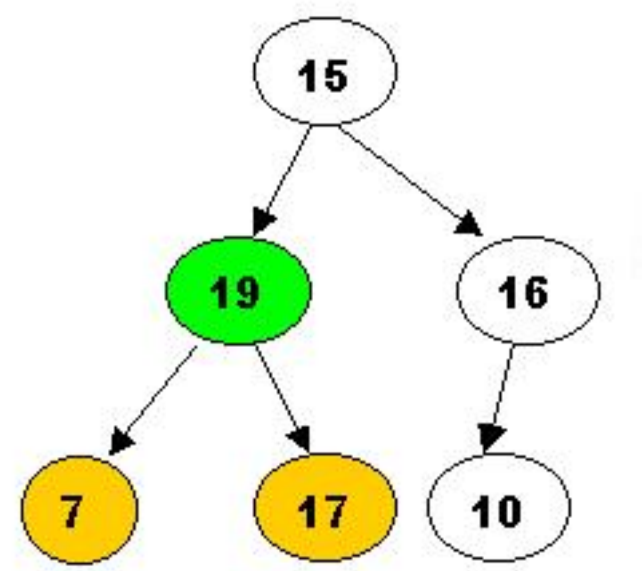
Build the Heap Tree

15	19	16	7	17	10



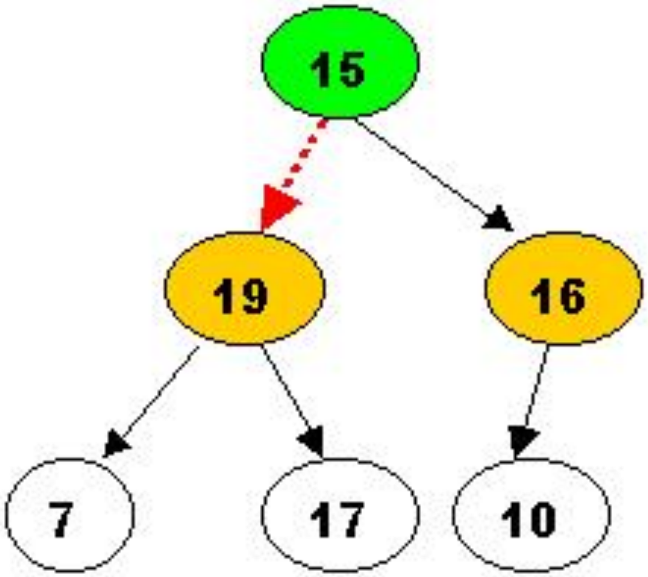
Build the Heap Tree



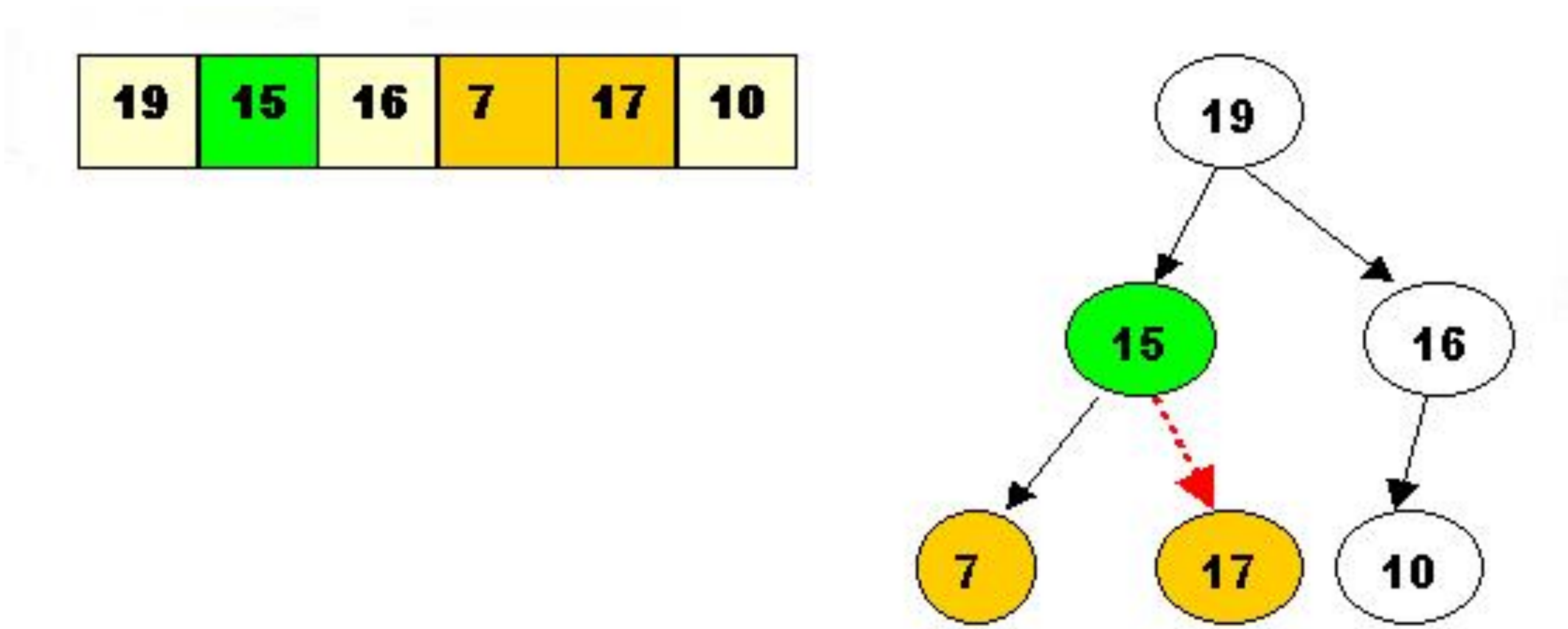


Build the Heap Tree

15	19	16	7	17	10

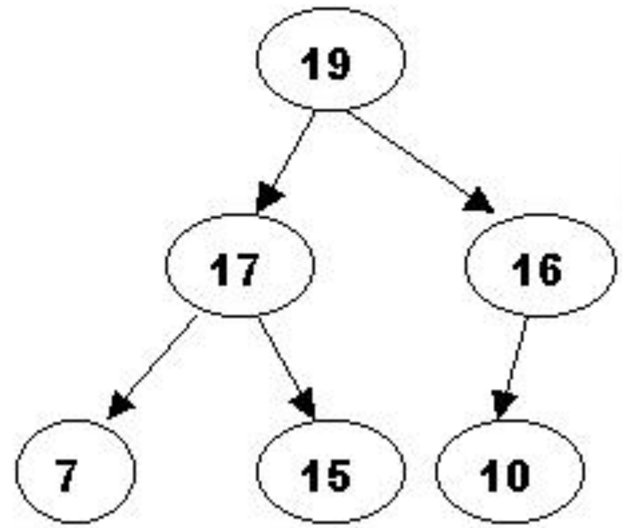


Build the Heap Tree



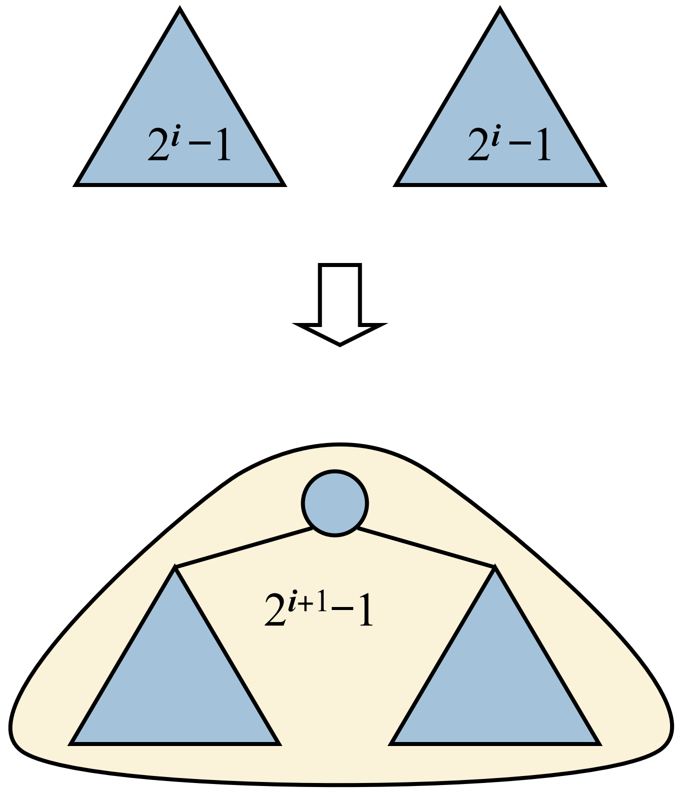
Build the Heap Tree

19	17	16	7	15	10



Bottom-up Heap Construction

- We can construct a heap storing ***n*** given keys in using a bottom-up construction with log ***n*** phases
- In phase ***i***, pairs of heaps with 2^i -1 keys are merged into heaps with 2^{i+1}-1 keys

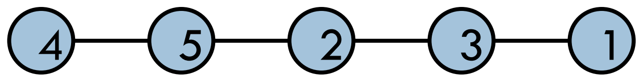


Heap-Sort

- Consider a priority queue with ***n*** items implemented by means of a heap
  - the space used is **O**(***n***)
  - methods push and pop take **O**(log ***n***) time
  - methods size, isEmpty, minKey, and minElement take time **O**(1) time
- Using a heap-based priority queue, we can sort a sequence of ***n*** elements in **O**(***n*** log ***n***) time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Exercise: Heap-Sort

- Heap-sort is the variation of PQ-sort where the priority queue is implemented with a heap



- Illustrate the performance of heap-sort on the following input sequence:
  - (22, 15, 36, 44, 10, 3, 9, 13, 29, 25)

Priority Queue Sort Summary

- PQ-Sort consists of n insertions followed by n removeMin ops

	Insert	**RemoveMin**	**PQ-Sort Total**
Insertion Sort (ordered sequence)	O(n)	O(1)	O(n^2)
Selection Sort (unordered sequence)	O(1)	O(n)	O(n^2)
Heap Sort (binary heap, vector-based implementation)	O(logn)	O(logn)	O(n logn)