Lecture 12
Priority Queue

OOP

Linked Lists

Arrays

Queues

Stacks

Trees

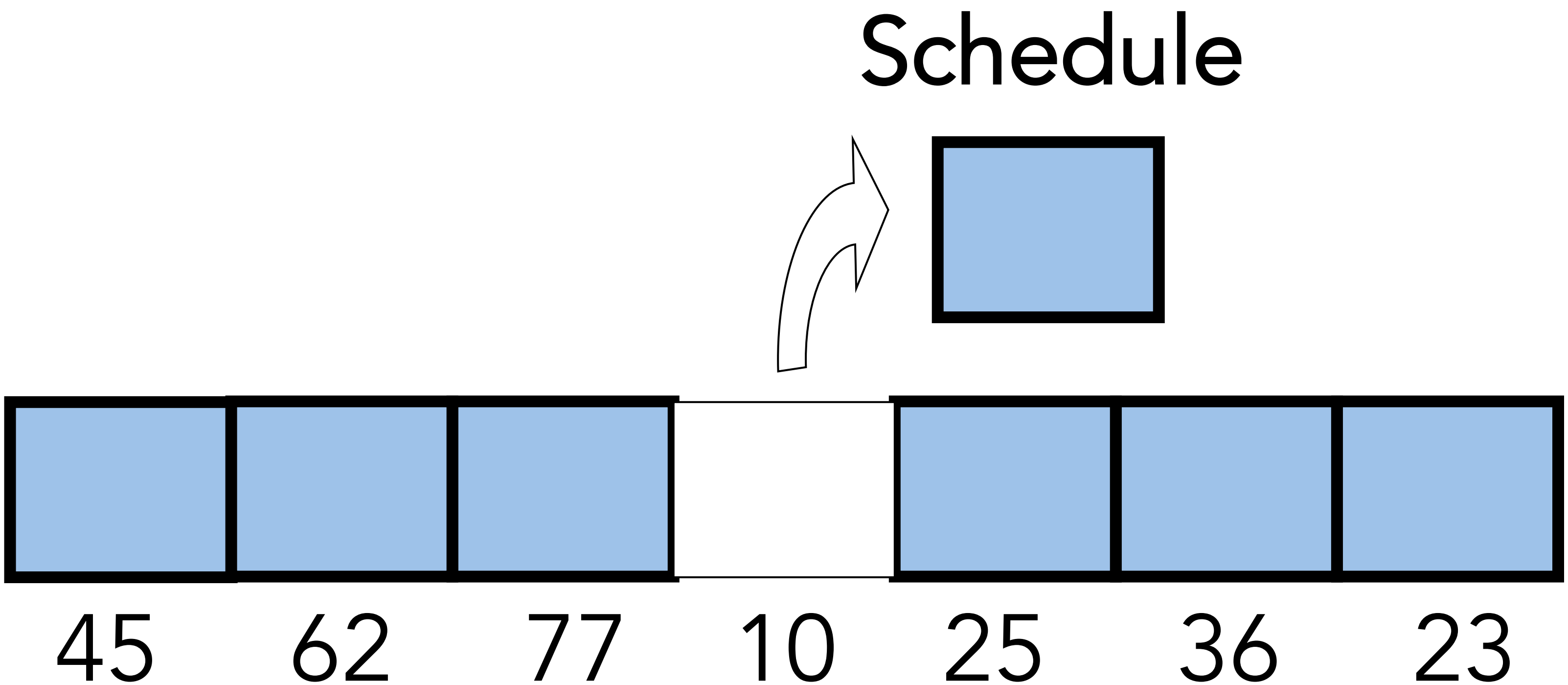
Algorithm Analysis

Scheduler

– ready programs are added to the scheduler

– it decides which program to execute next

SRTF Scheduling Policy



Main Operations

insert()

removeMin()

Priority Queue ADT

- a collection of entries
- entry = (key, value)
- main methods
  - insert(k, v)
  - removeMin()
- additional methods
  - min()
  - size(),
  - isEmpty()

Example

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Keys

- specific attribute of the element
- many times assigned to the element by user of application
- key may change for the same element, e.g. popularity

Total Order Relations

1. Comparability property:
   either $x \leq y$ or $y \leq x$
2. Antisymmetric property:
   $x \leq y$ and $y \leq x \Rightarrow x = y$
3. Transitive property:
   $x \leq y$ and $y \leq z \Rightarrow x \leq z$

- keys with total order
  - weight
- keys not having total order
  - 2D point

Comparator ADT

- implements isLess(p,q)
- can derive other relations from this:
  – (p == q)?
- for STL, in C++ overload "()"

Comparator Examples

```
class LeftRight {
public:
    bool operator()(Point2D& p, Point2D& q)
    { return p.getX() < q.getX(); }
};
```

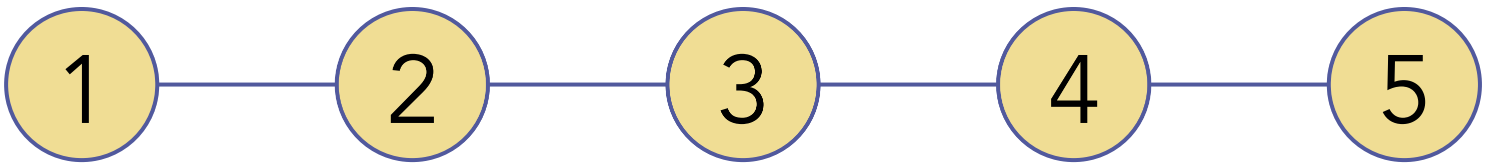
```
class BottomTop {
public:
    bool operator()( Point2D& p, Point2D& q)
    { return p.getY() < q.getY(); }
};
```

Sort Sequence L with Priority Queue P

```
while !L.empty ()
        e ← L.front();
        L.eraseFront()
        P.insert (e)
while !P.empty()
        e ← P.removeMin()
        L.insertBack(e)
```

What is the time complexity of this sorting?

Implementation with sorted sequence



- insert()?
- removeMin()?

Insertion-Sort

1. insert at right place to keep the list sorted
2. remove head repeatedly

Insertion-Sort Example

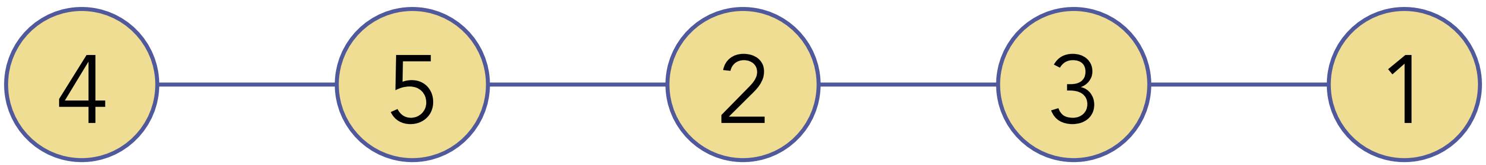
	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Time complexity

Best case?

Worst case?

Implementation with unsorted sequence



- insert()?
- removeMin()?

Selection-Sort

1. insert elements in unsorted list
2. remove min repeatedly

Selection-Sort Example

	Sequence L	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

What is time complexity of selection sort?

Worst case?

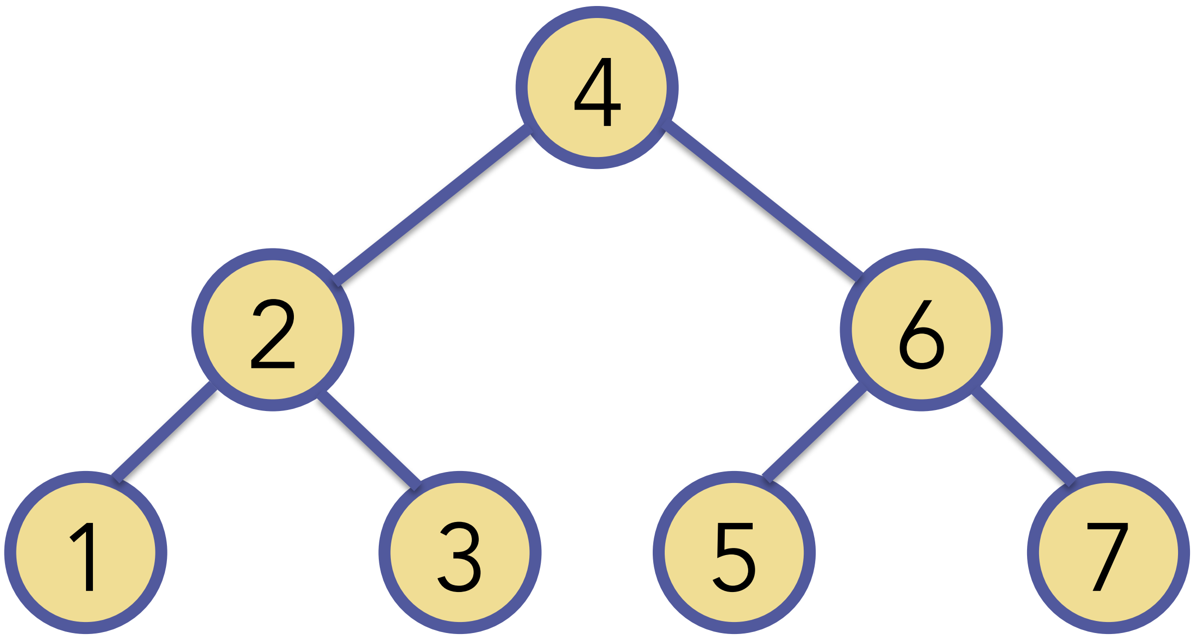
Best case?

Give two advantages of selection sort over others?

Summary: either insert is O(n) or removeMin() is O(n)!

Can we do better?

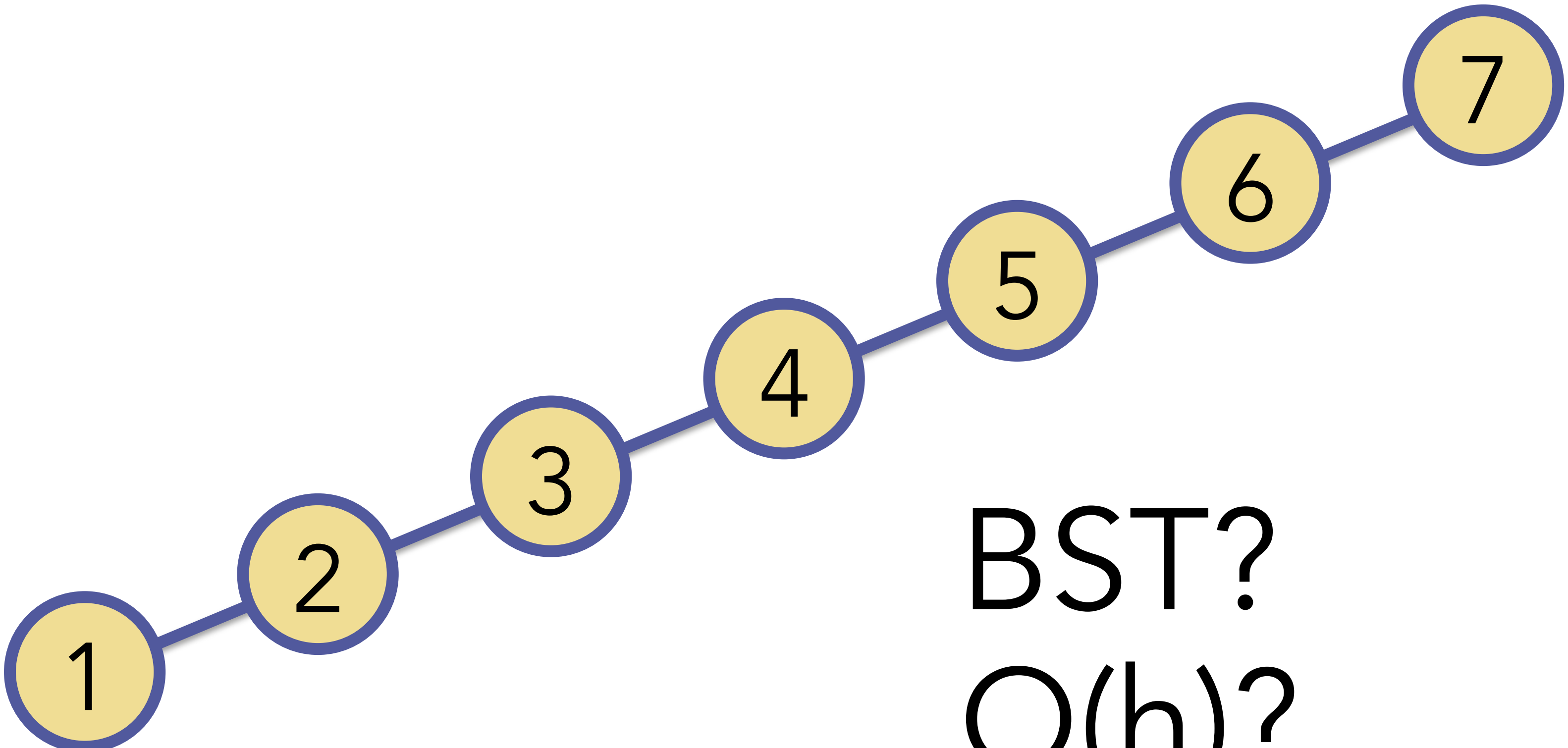
Lets Try BST



Time complexity
removeMin()?
insert()?

Is $O(h)$ good enough?

Look at this tree…



BST?
O(h)?

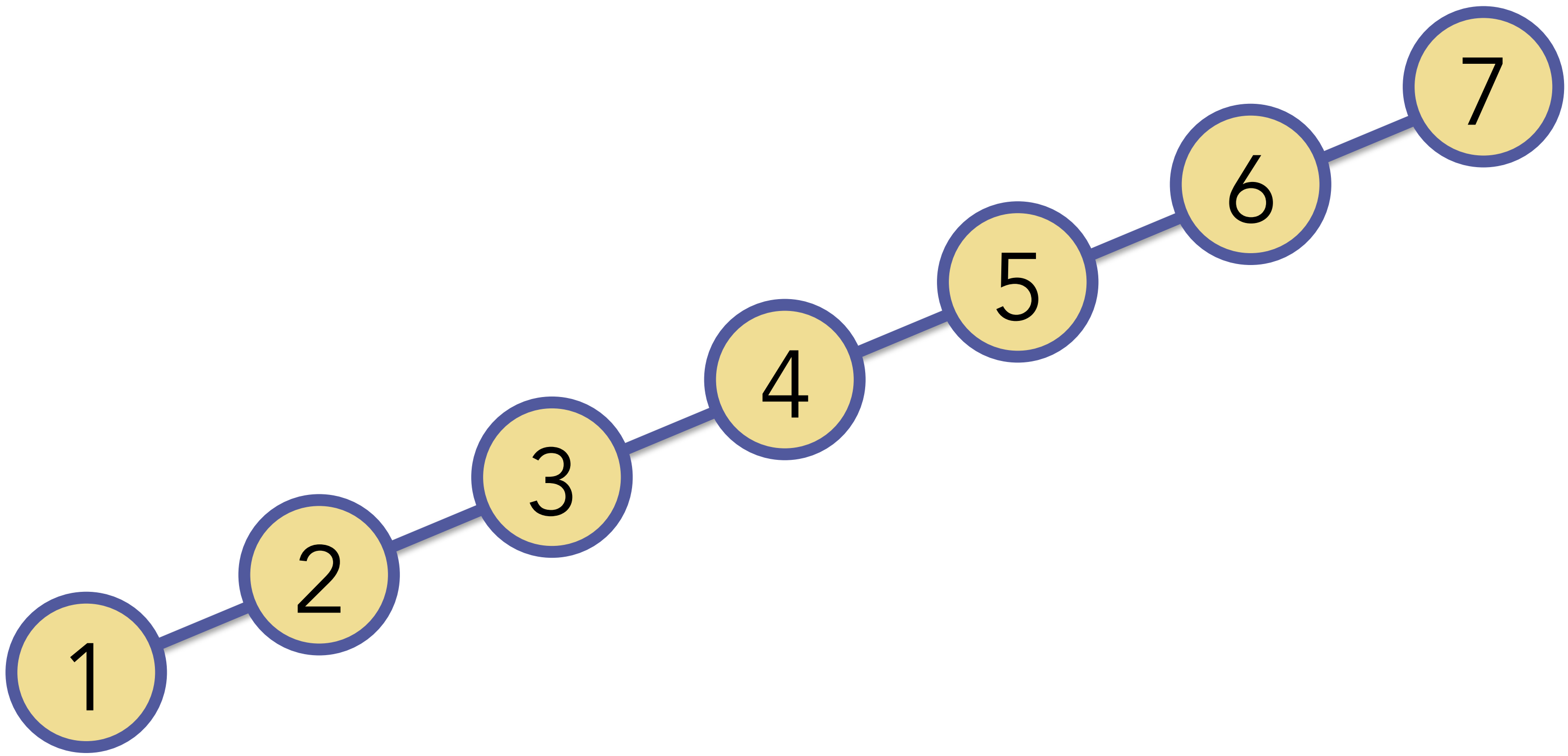
BST Order Restrictions

1. left subtree ≤ node
2. right subtree ≥ node

A simpler order restriction…

1. parent ≤ child

Remake this tree with new order restriction…



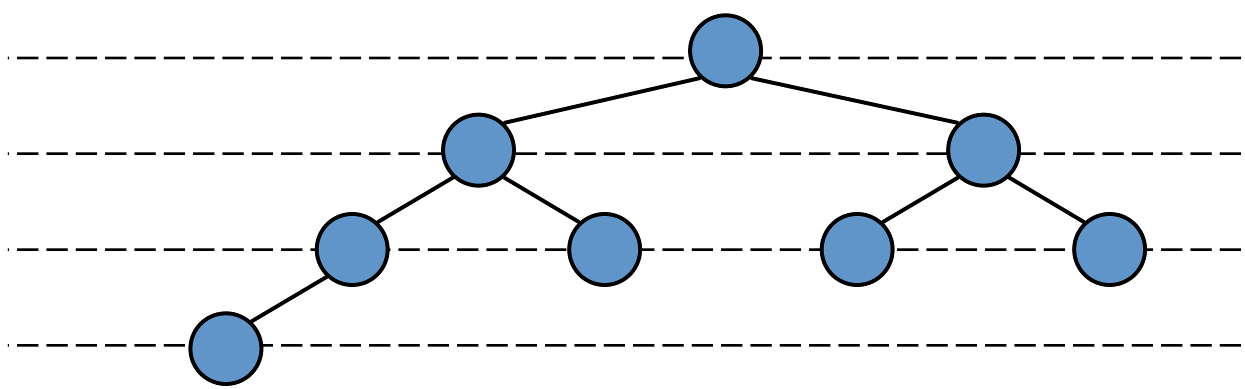
Heap

1. Order restriction:
   K(parent) ≤ K(child)
2. Structural restriction:
   complete binary

Height of Heap

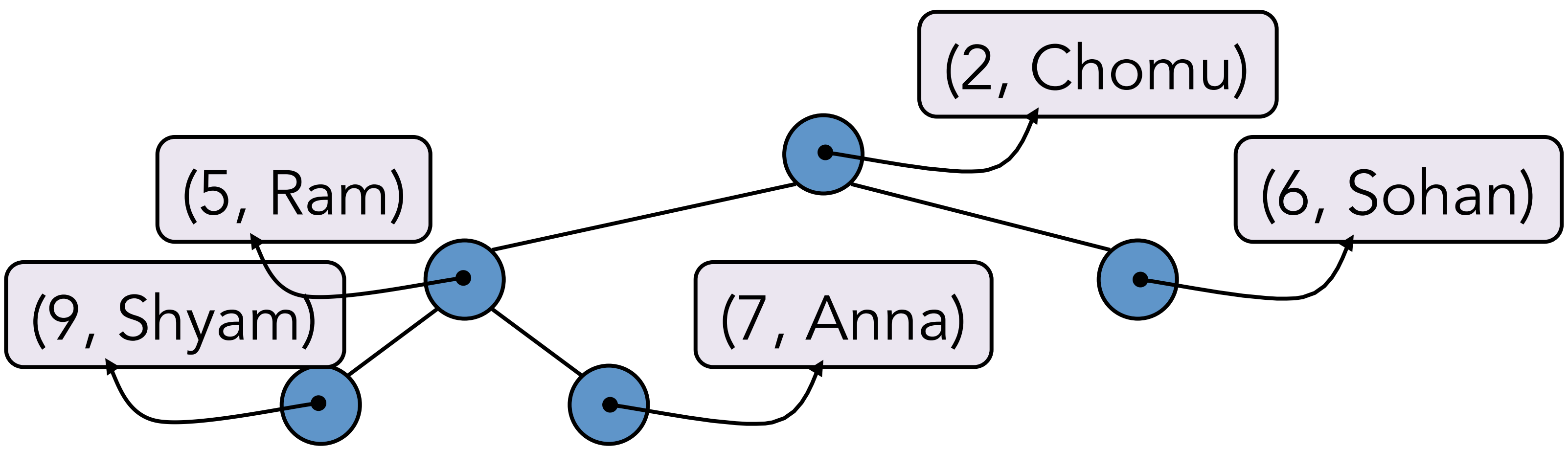
$h \leq \log n$, so, h is $O(\log n)$

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

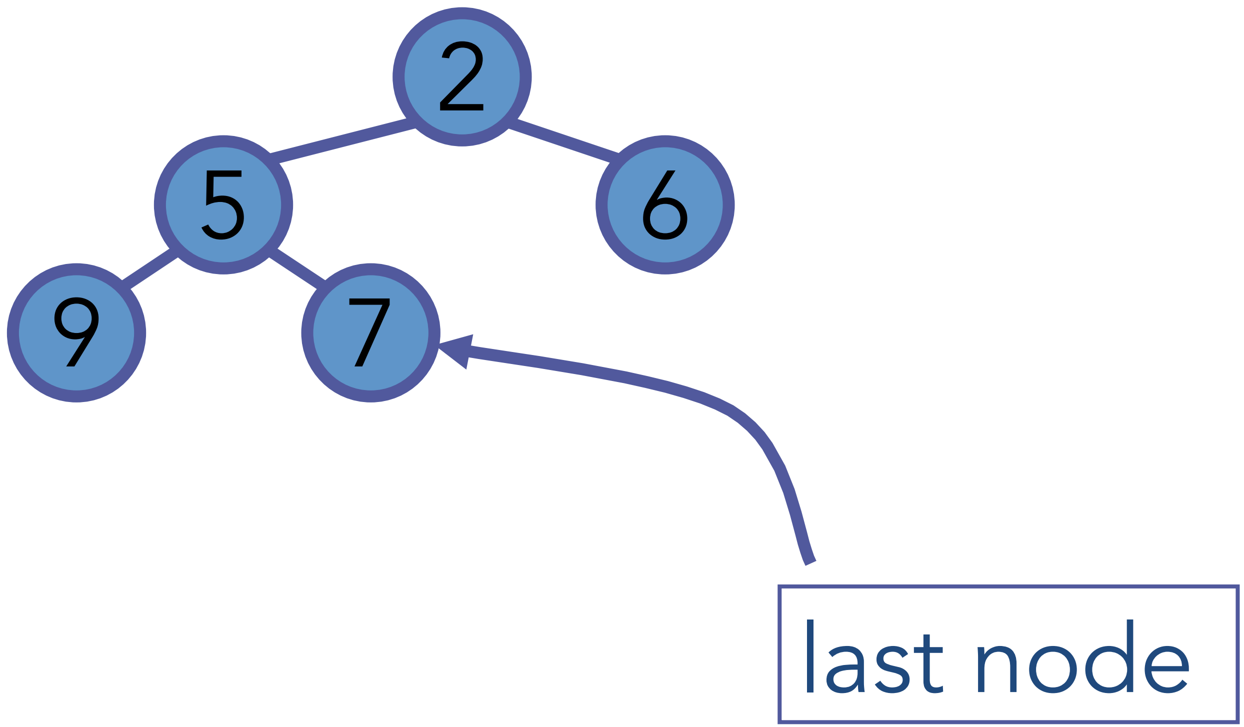


Heaps and Priority Queues

- Heap can be used to implement a priority queue
- store a (key, element) item at each internal node
- keep track of the last node



The last node of a heap is the rightmost node of maximum depth!

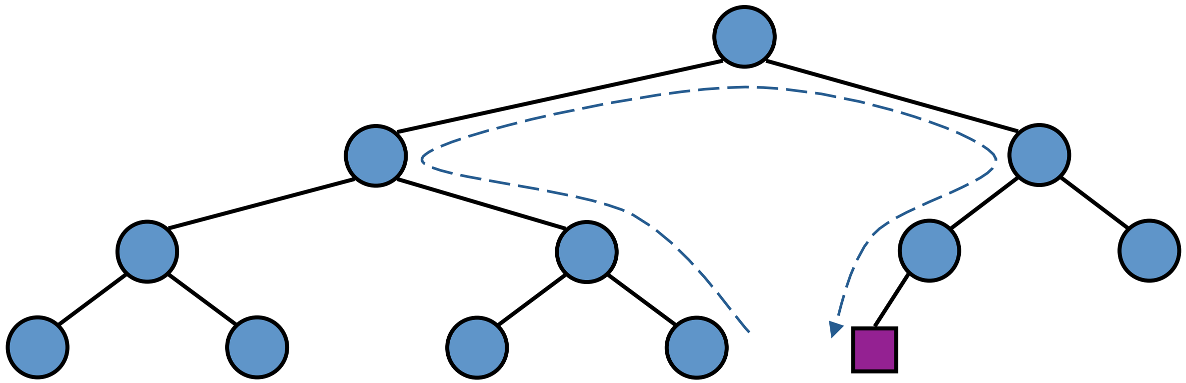


insert()

1. find insertion node
2. add new element (k) at this node
3. restore heap order

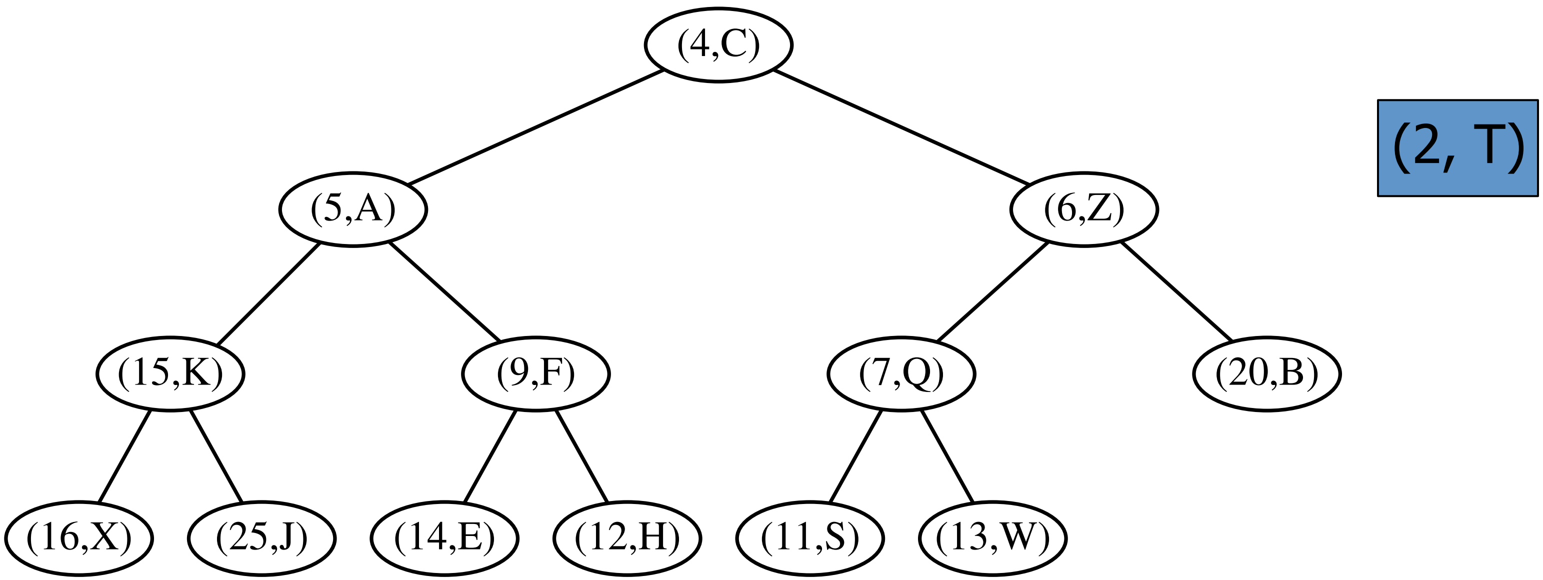
Finding the Insertion Node

- go up until node becomes a left child or the root is reached
- if root is reached, go left until leaf node is reached
- if node becomes left child, go to the right sibling and go down left until a leaf is reached

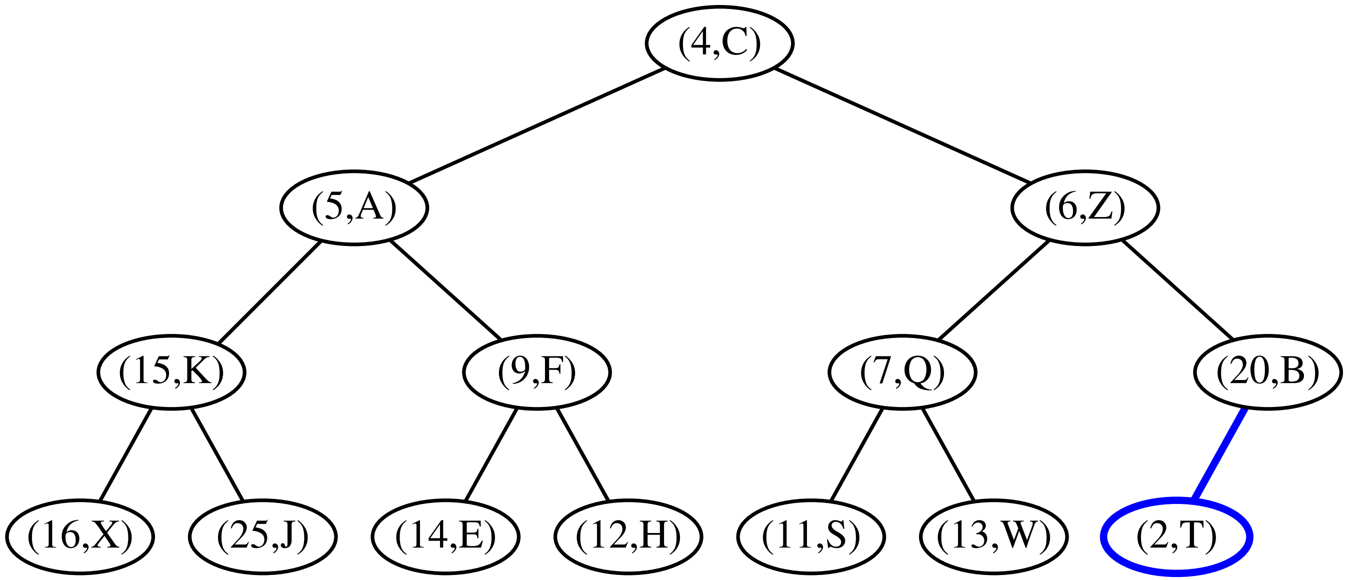


time complexity of finding the insertion node

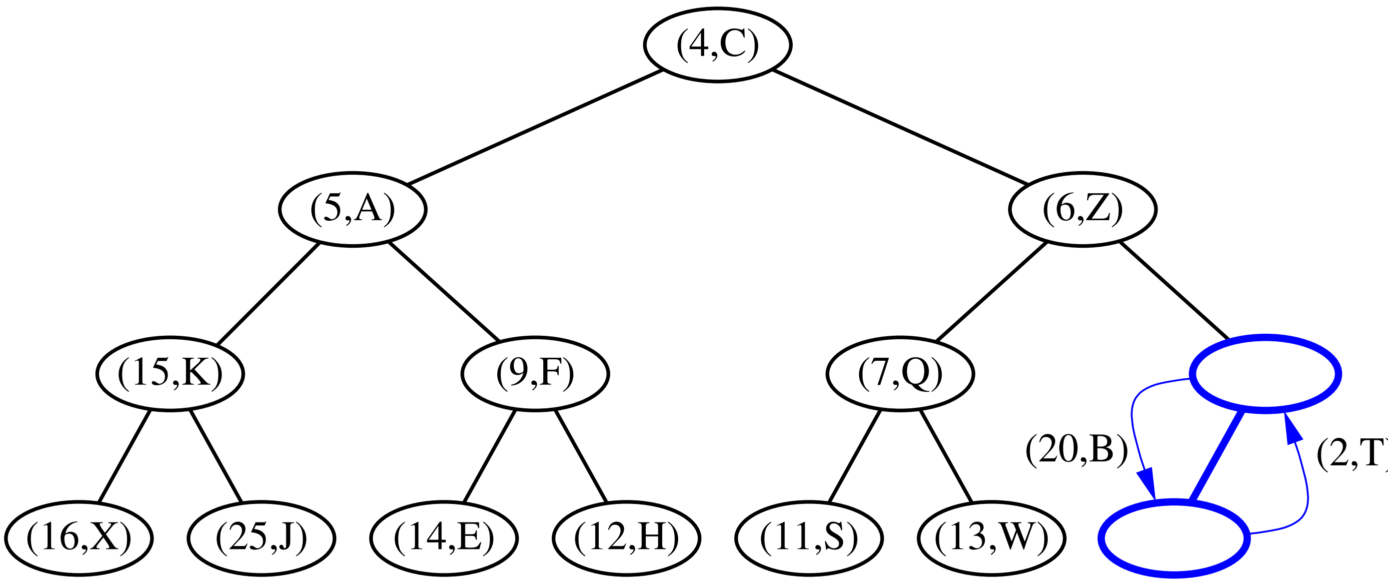
Insertion into a Heap



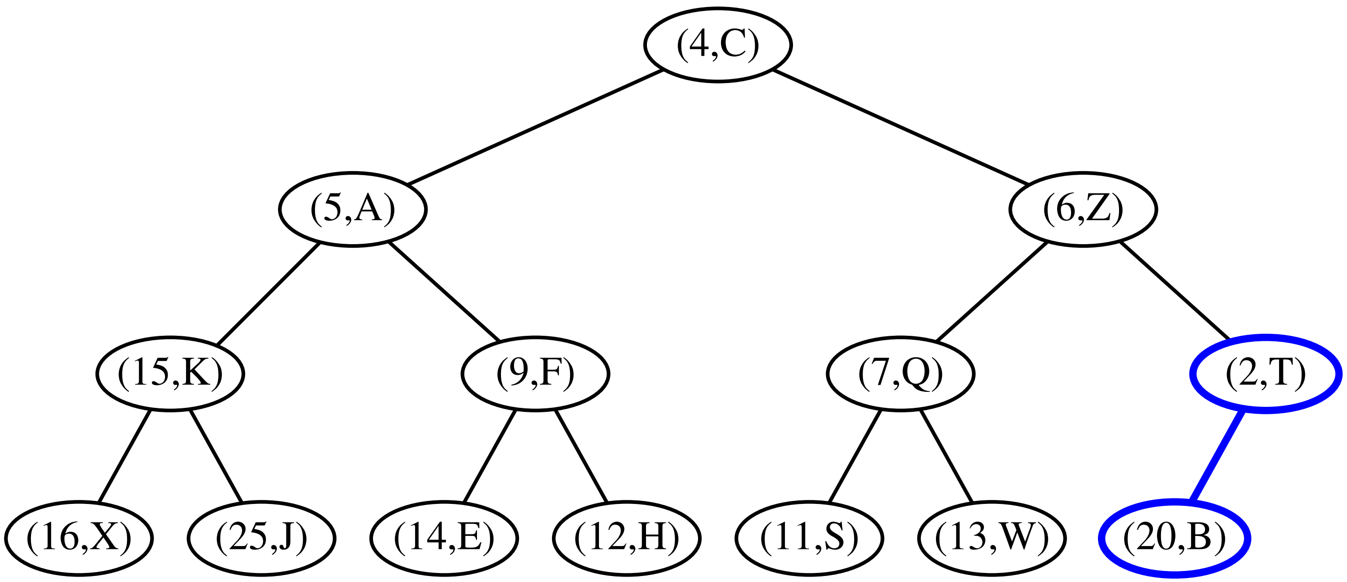
up-heap bubbling



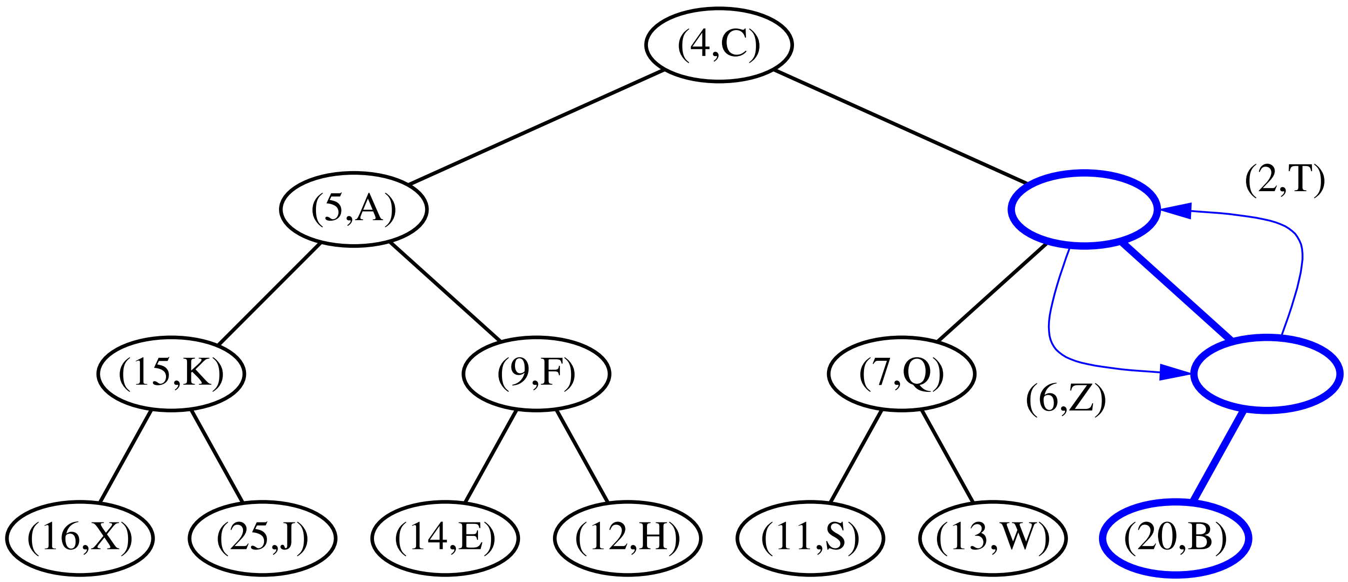
up-heap bubbling



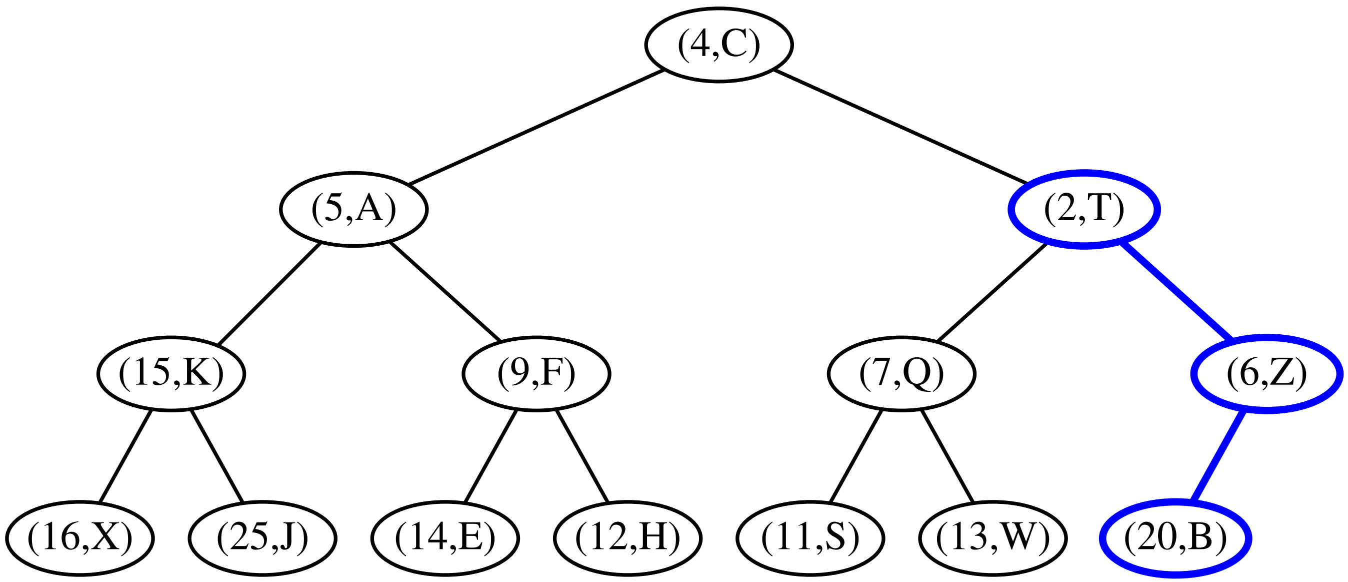
up-heap bubbling



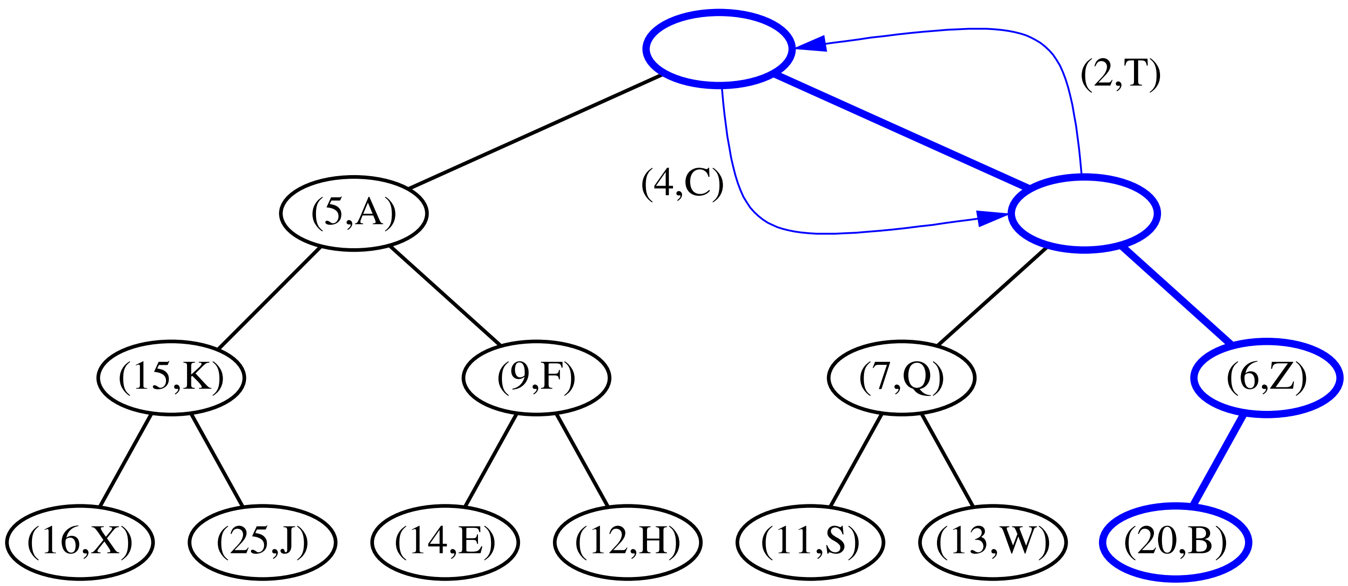
up-heap bubbling



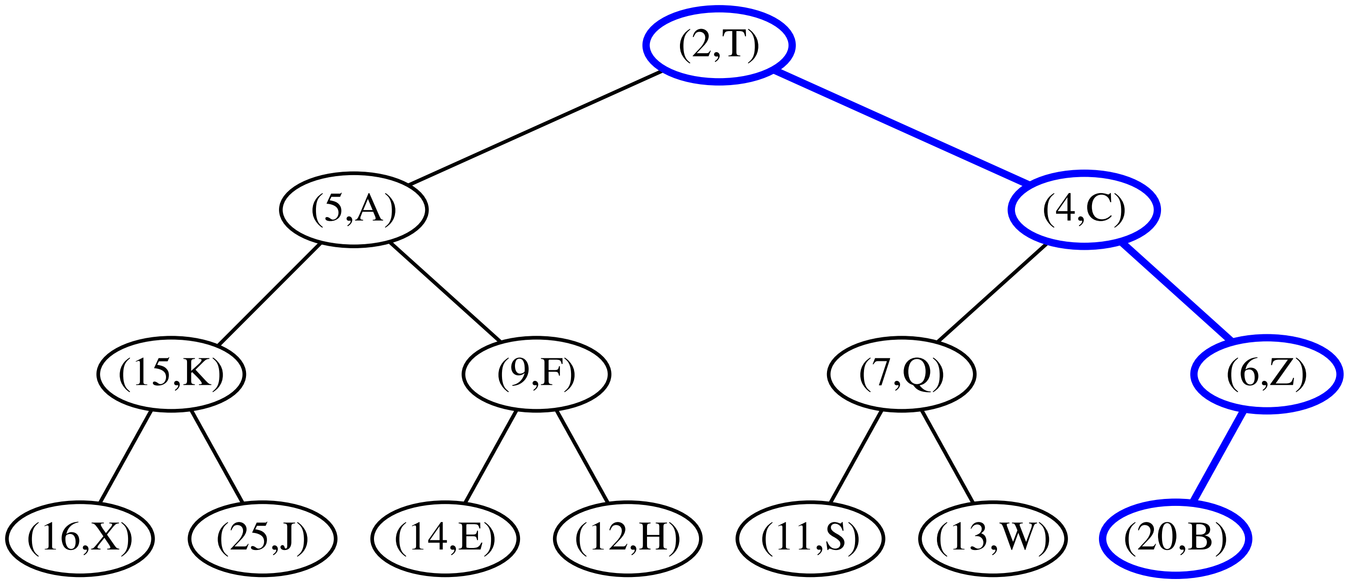
up-heap bubbling



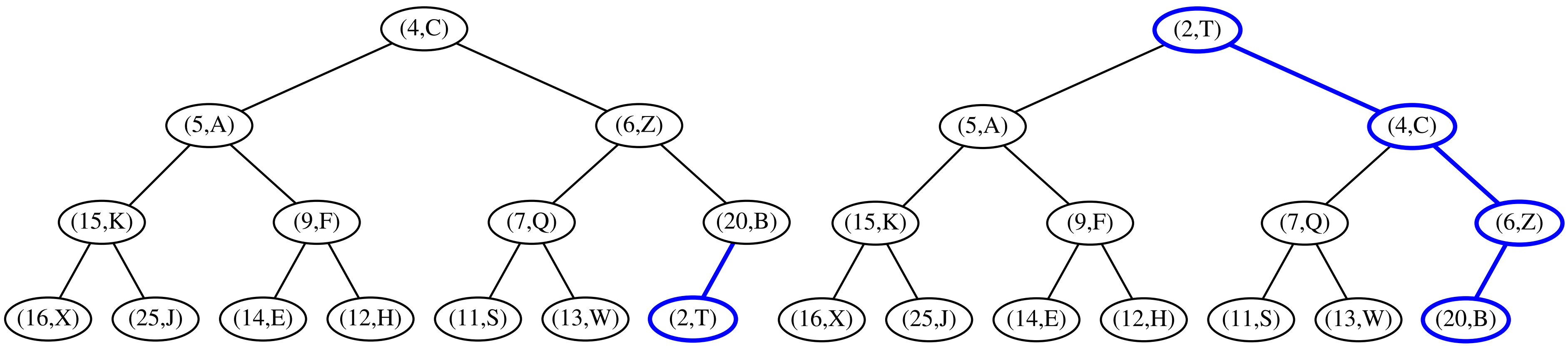
up-heap bubbling



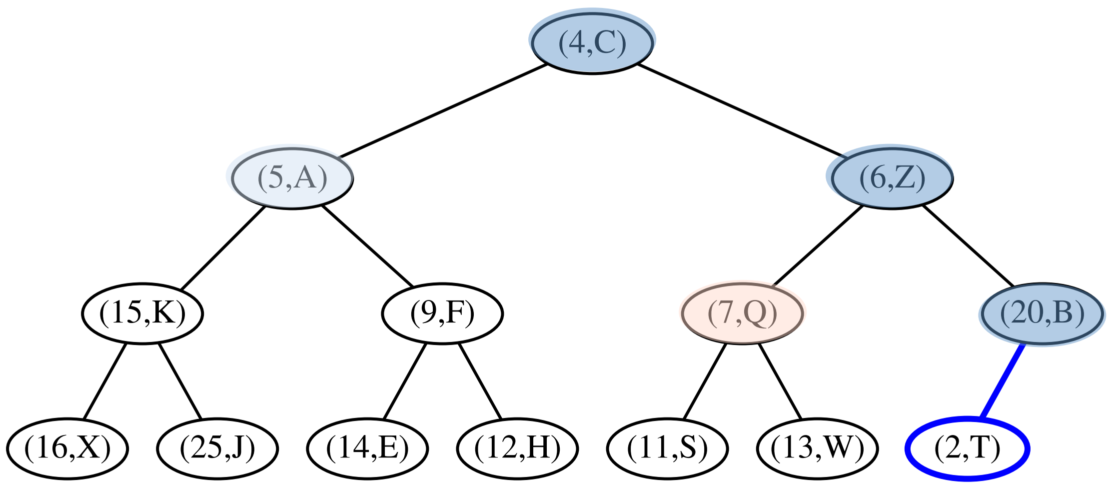
up-heap bubbling



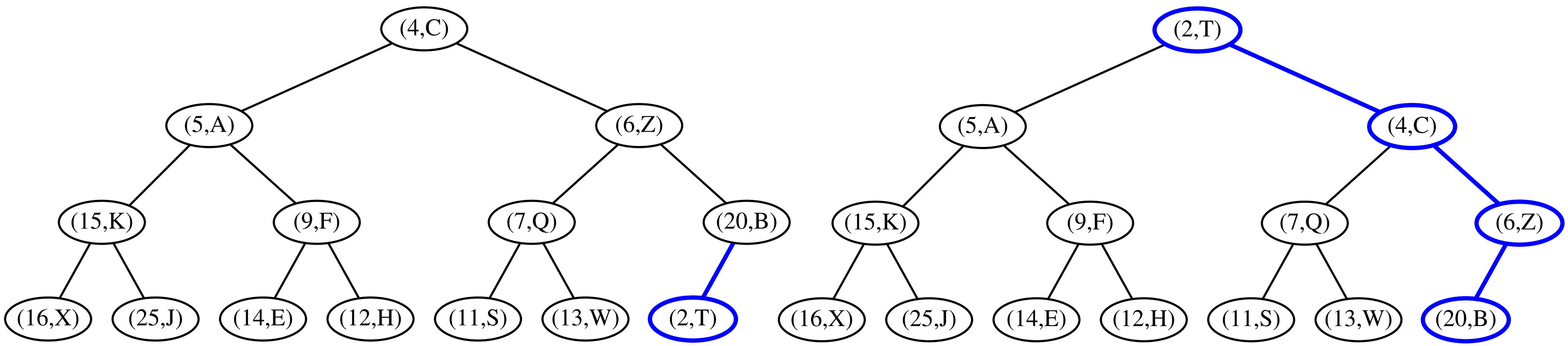
Another View of Insertion



Correctness of Upheap



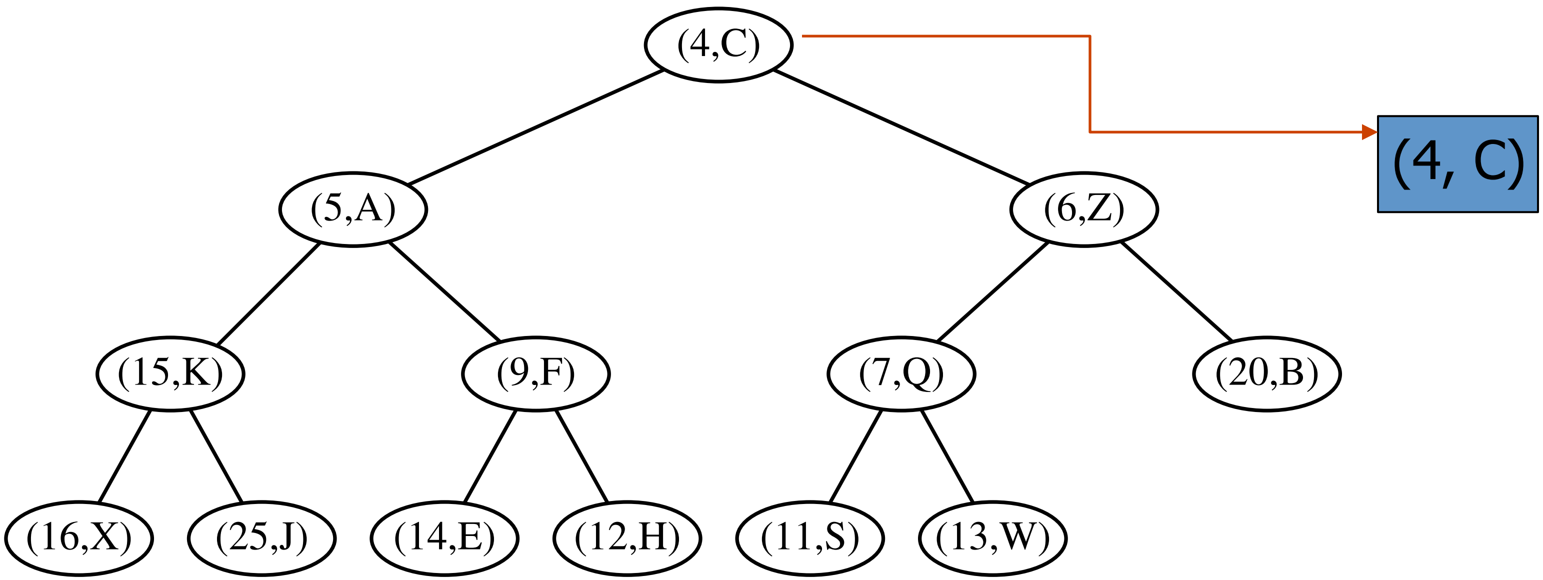
Always replaced with smaller key!



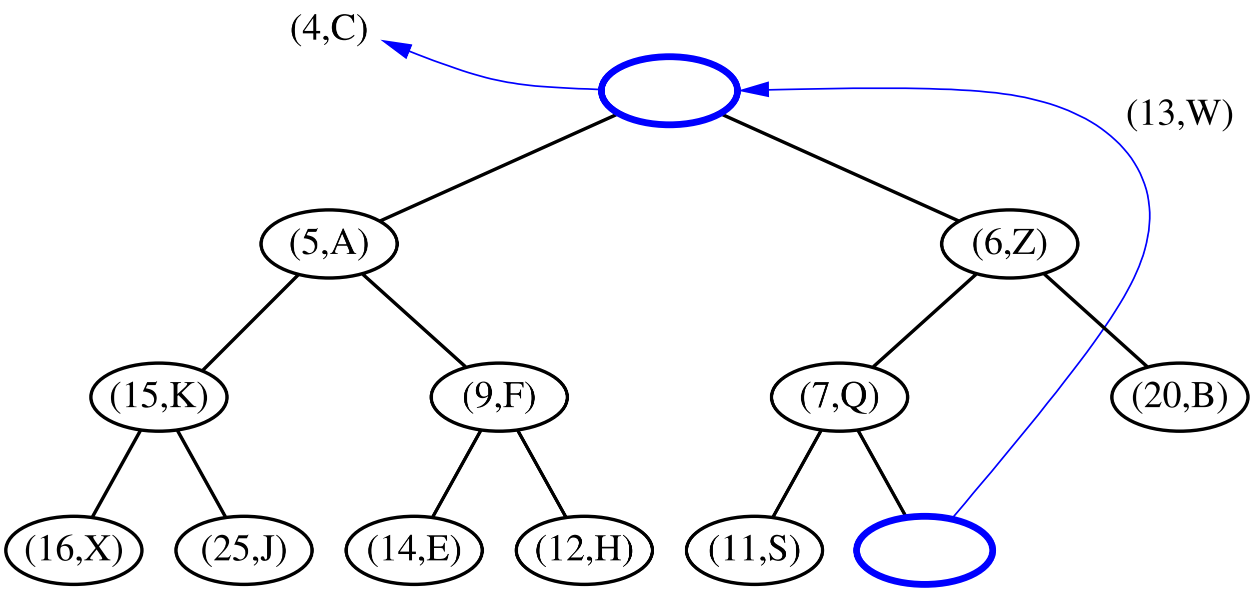
Since a heap has height $O(\log n)$, up-heap runs in $O(\log n)$ time

removeMin()

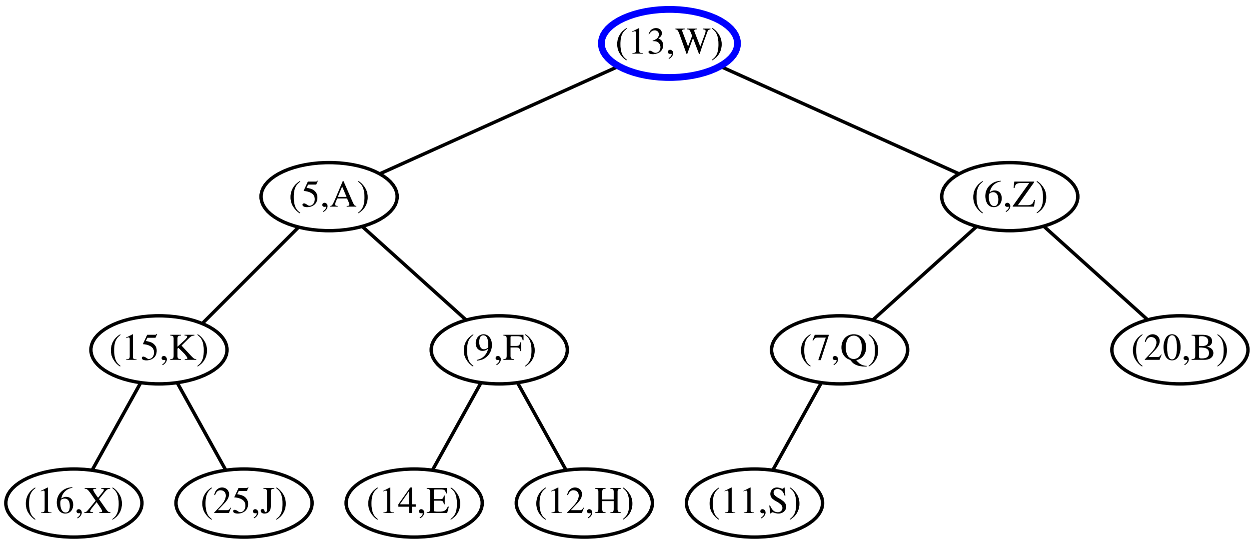
Removal from a Heap



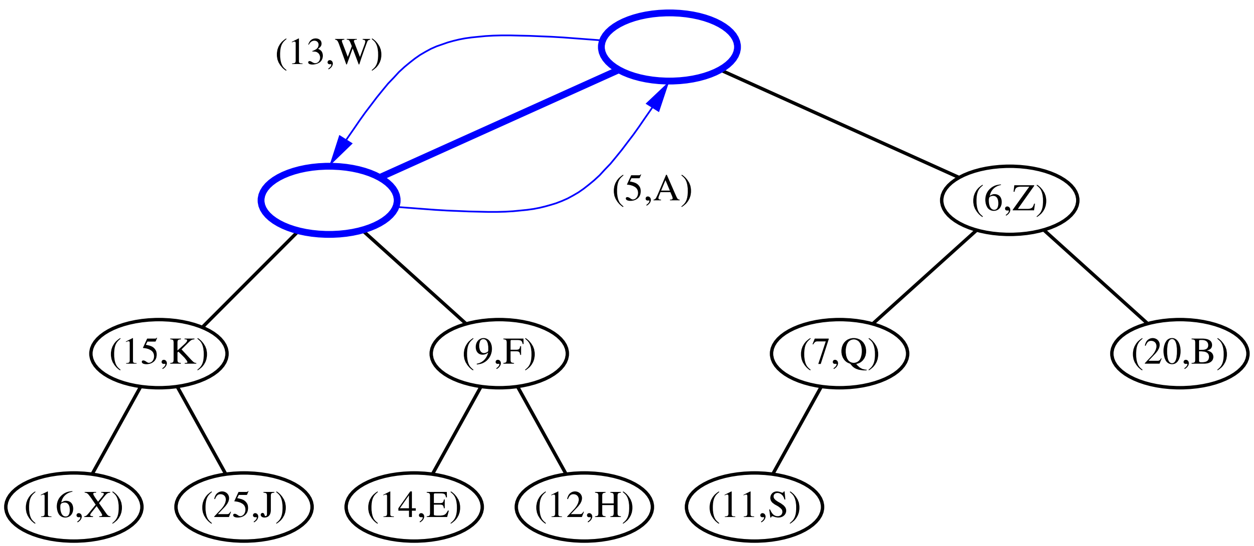
Removal from a Heap



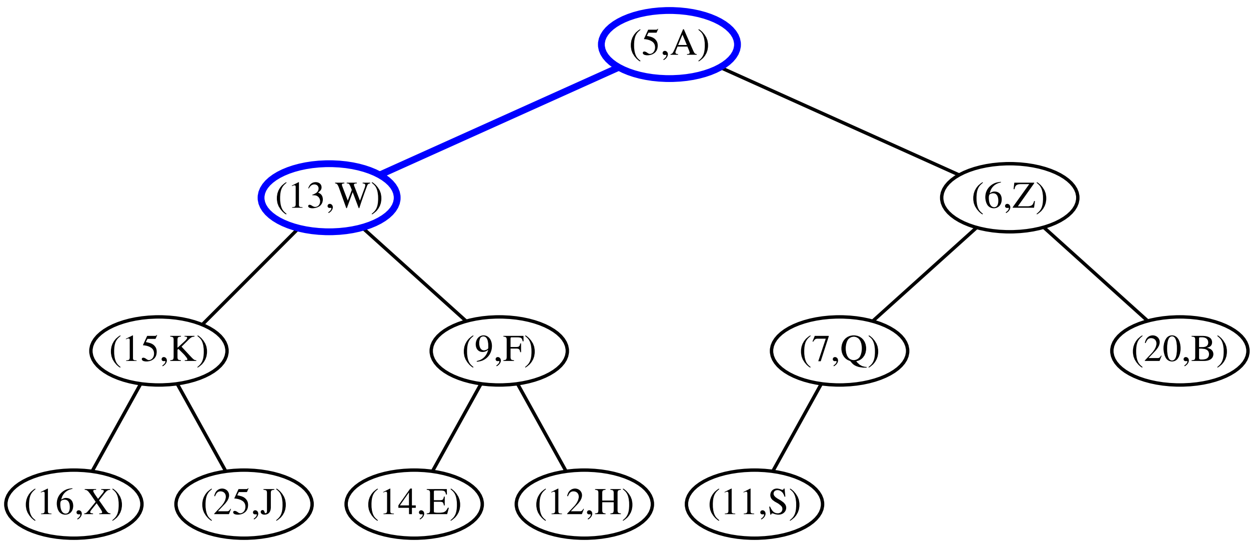
down-heap bubbling



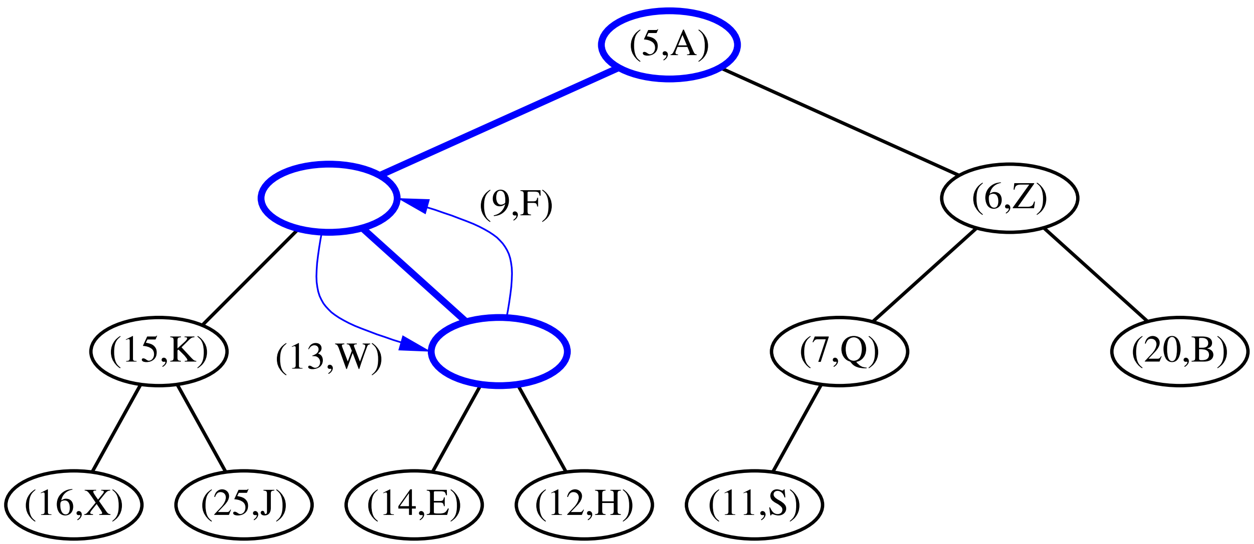
down-heap bubbling



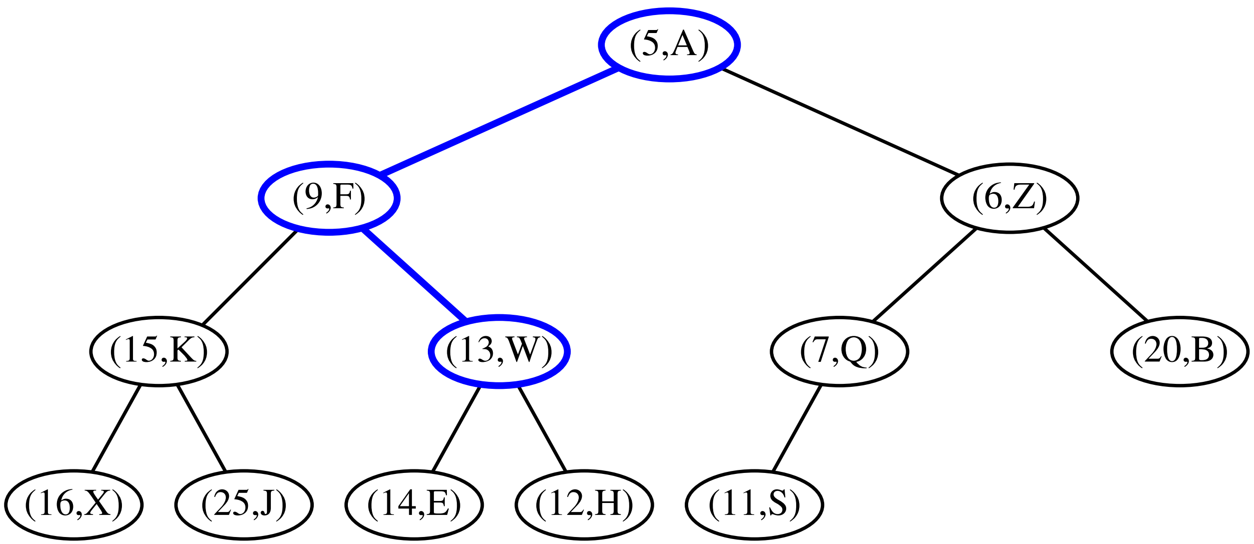
down-heap bubbling



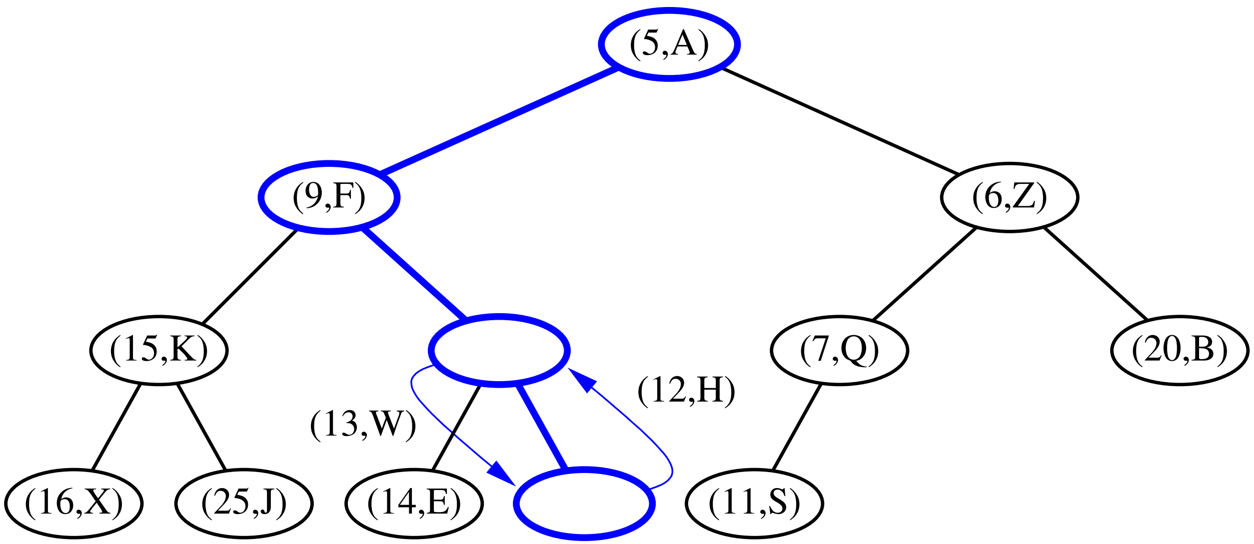
down-heap bubbling



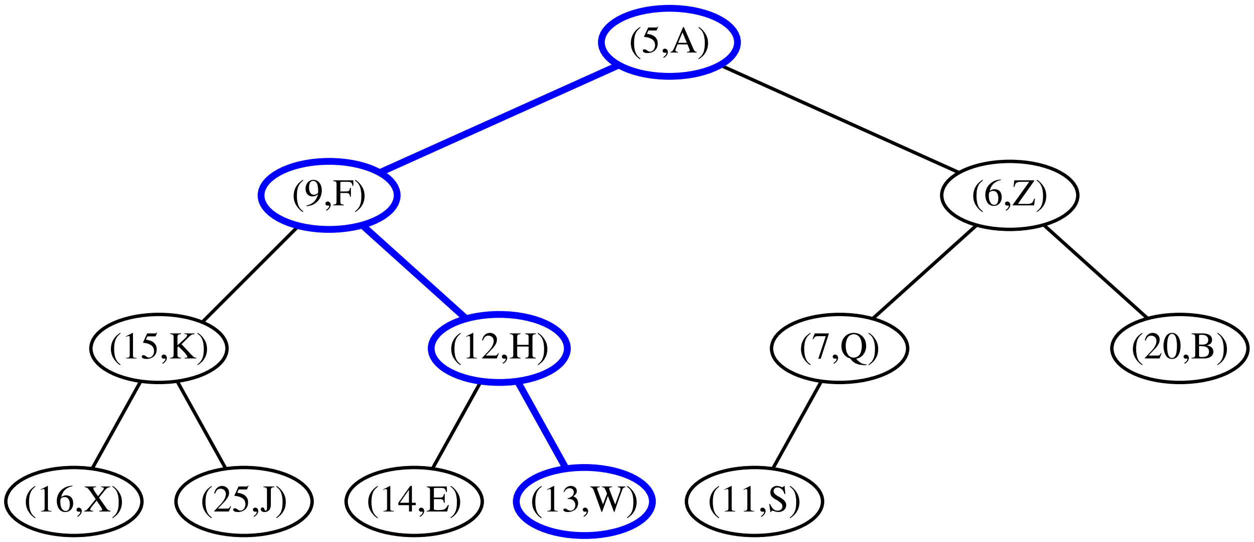
down-heap bubbling



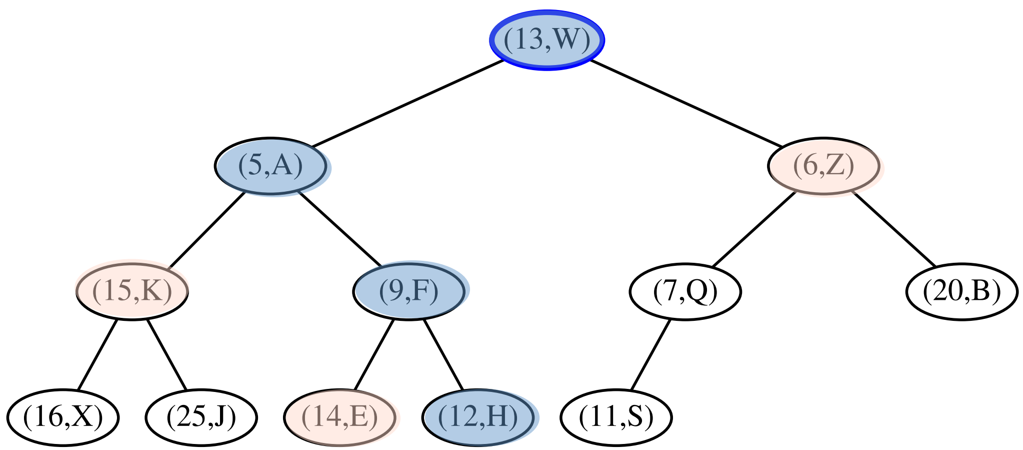
down-heap bubbling



down-heap bubbling



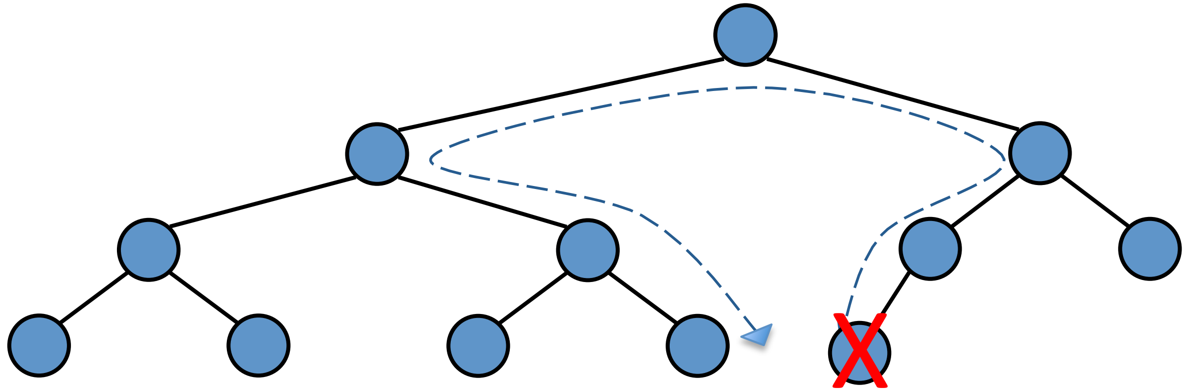
Correctness of Downheap



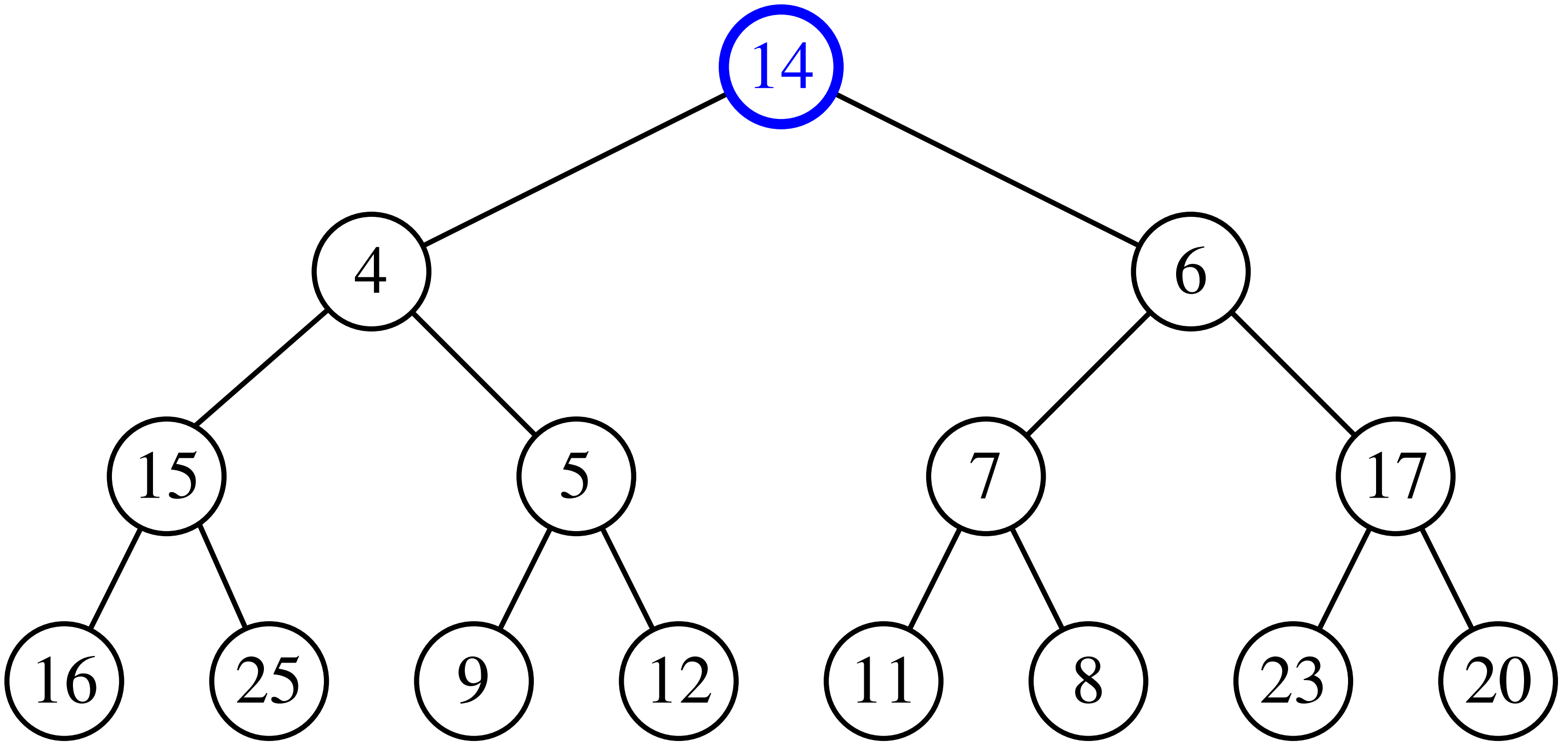
Since a heap has height $O(\log n)$, down-heap runs in $O(\log n)$ time

Keeping track of last node after removeMin()

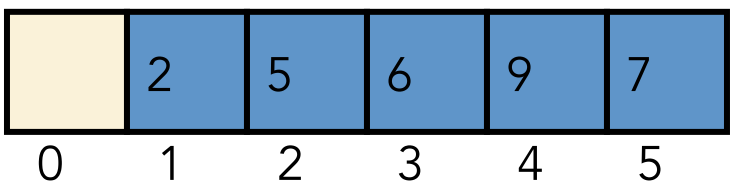
Similar to finding node for insertion

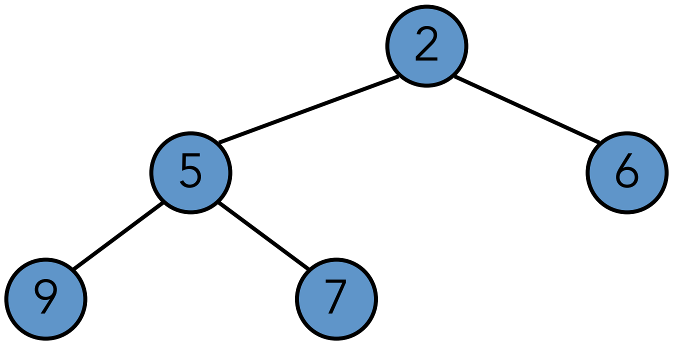


Vector Implementation of Complete Binary Tree


14
4
6
15
5
7
17
16
25
9
12
11
8
23
20

- start at rank 1
- for the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$





insert() and removeMin() on vector heap!

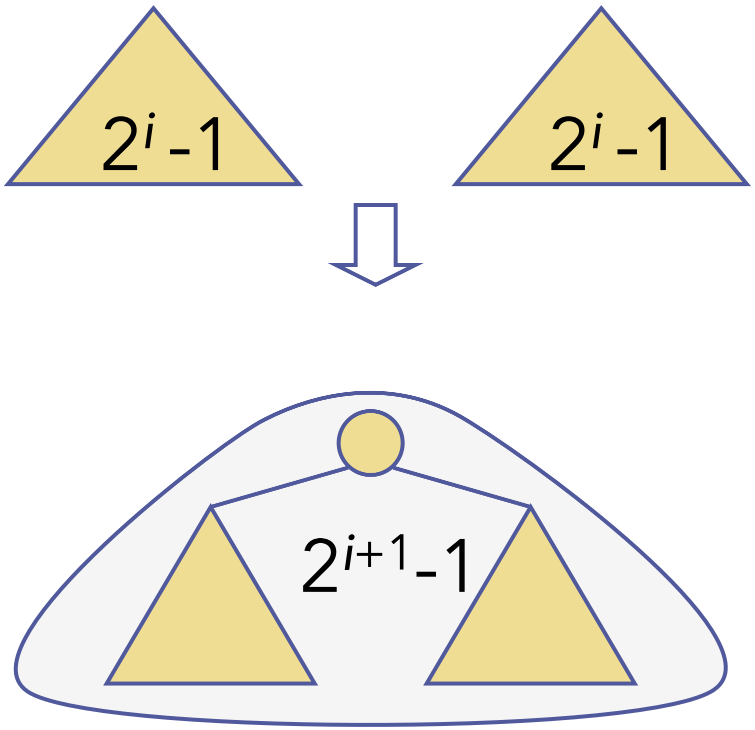
How to build a heap?

1. insert repeatedly
$O(n \log n)$

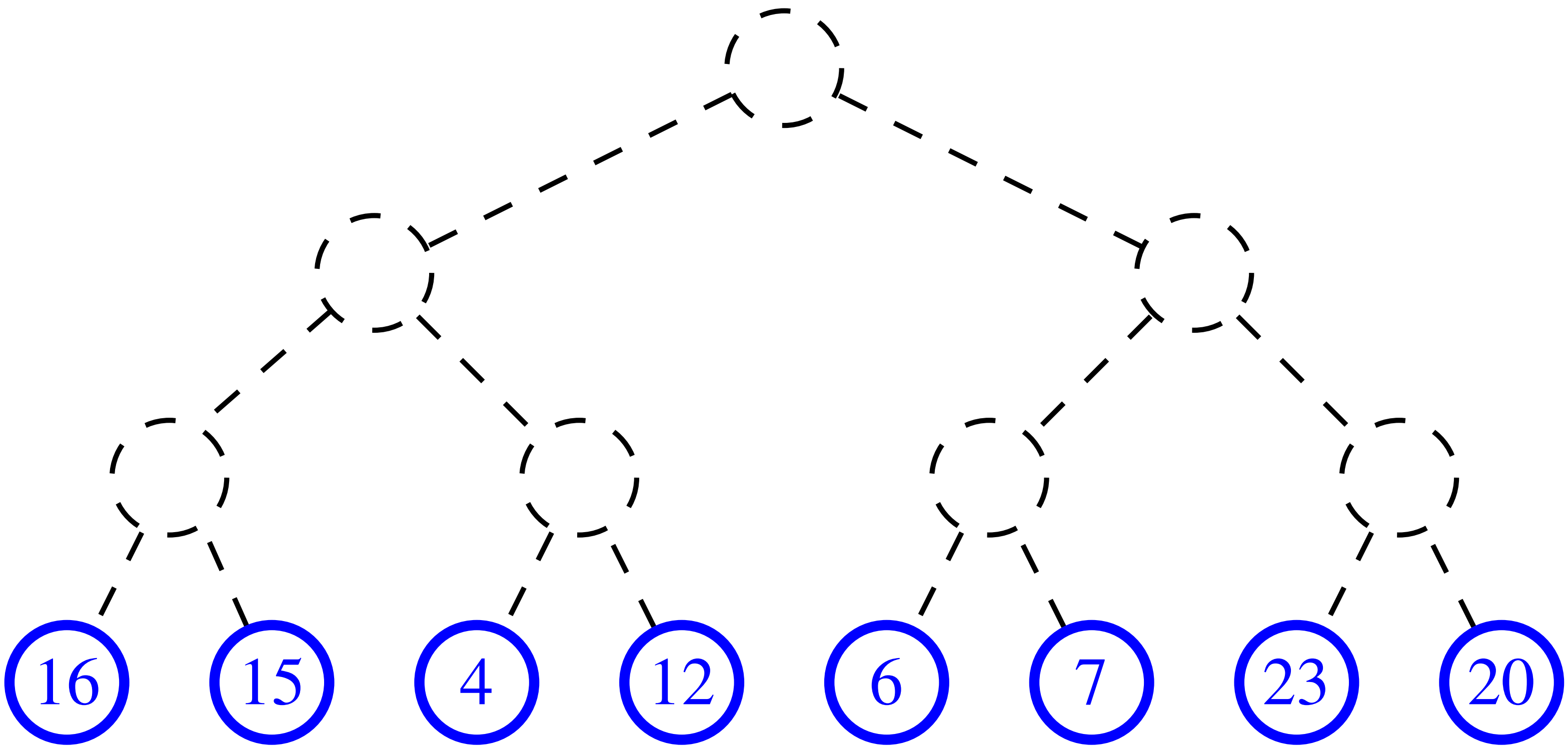
Build a heap with the below keys!

(16, 15, 4 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14)

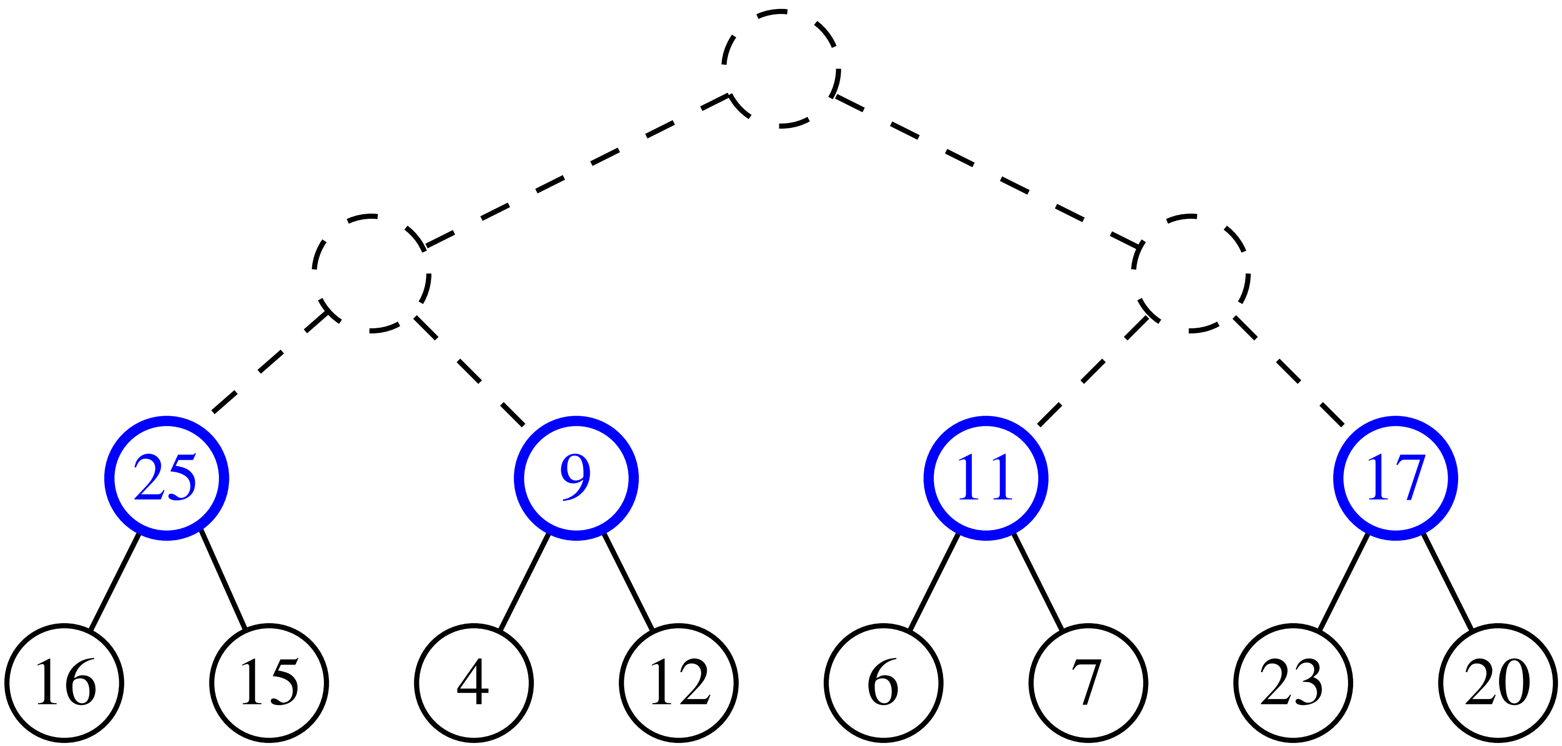
2. Bottom-up Heap Construction



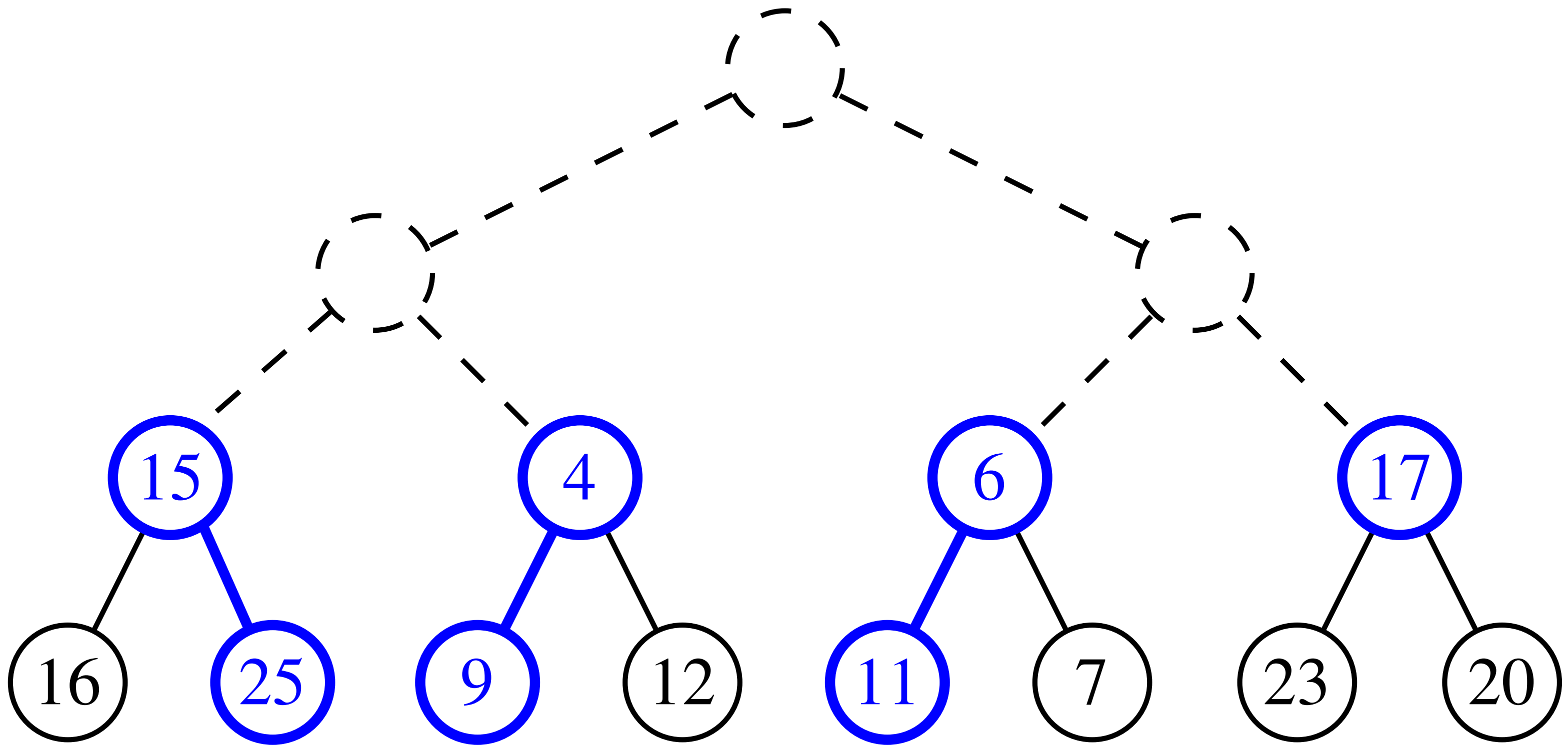
Building a Heap – Bottomup



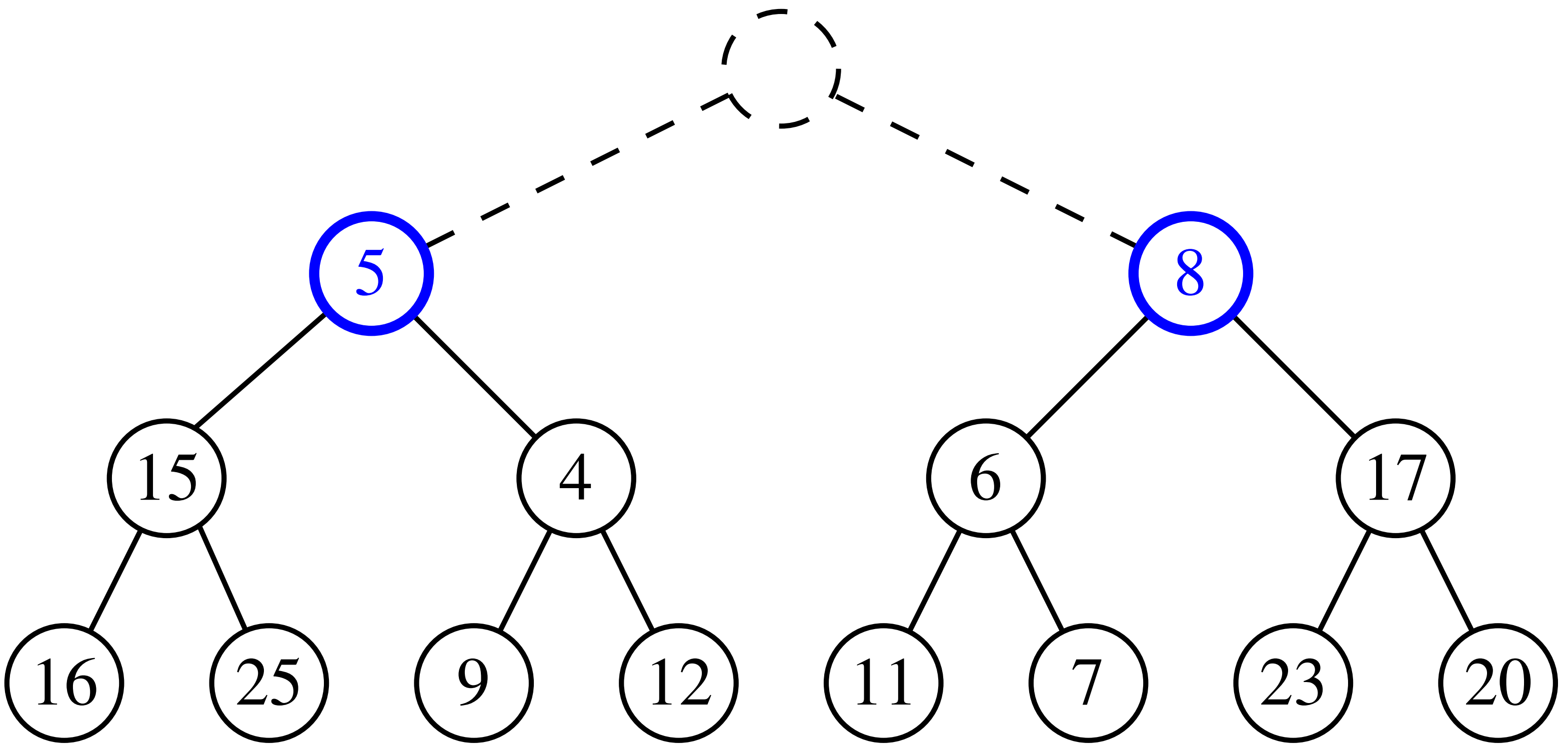
Building a Heap – Bottomup



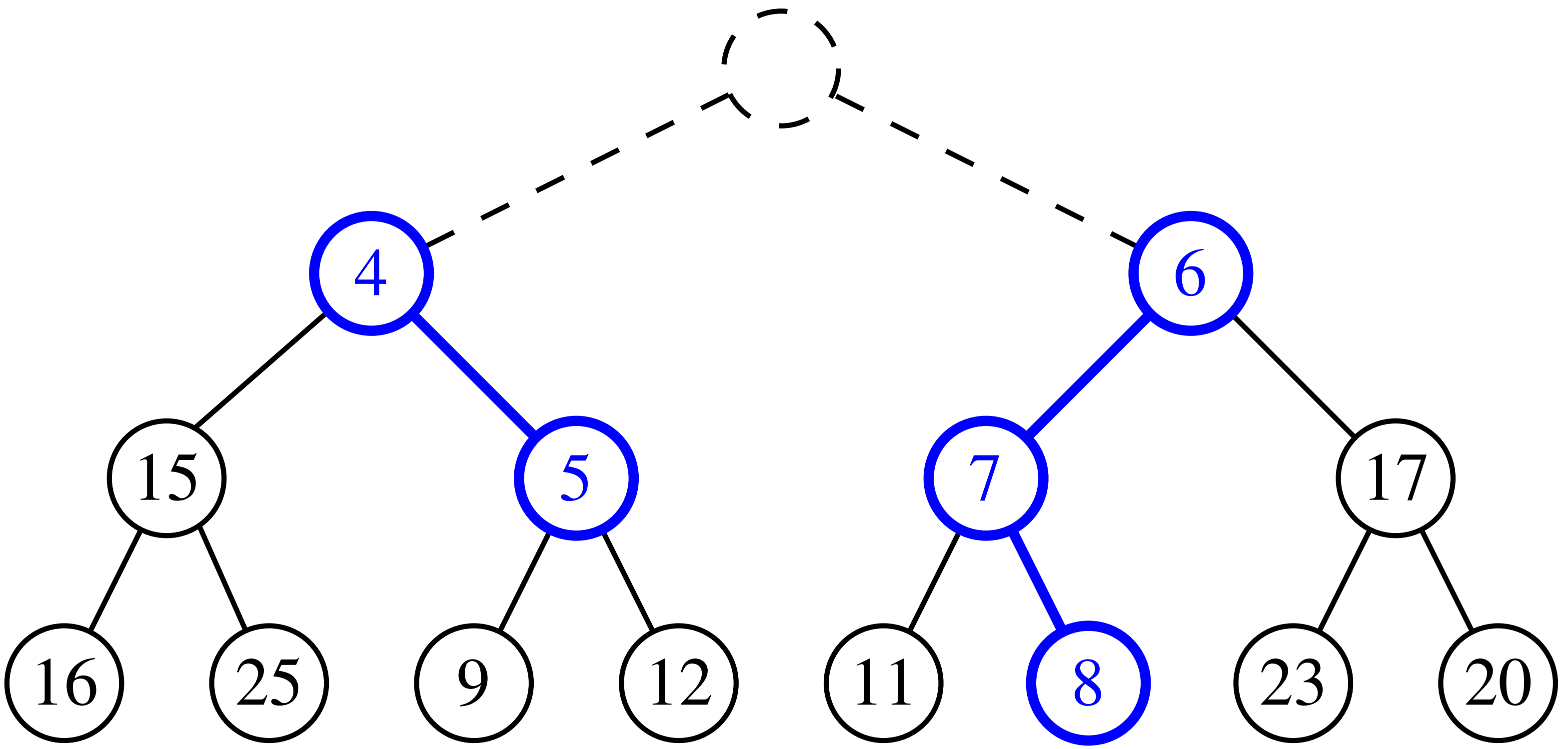
Building a Heap – Bottomup



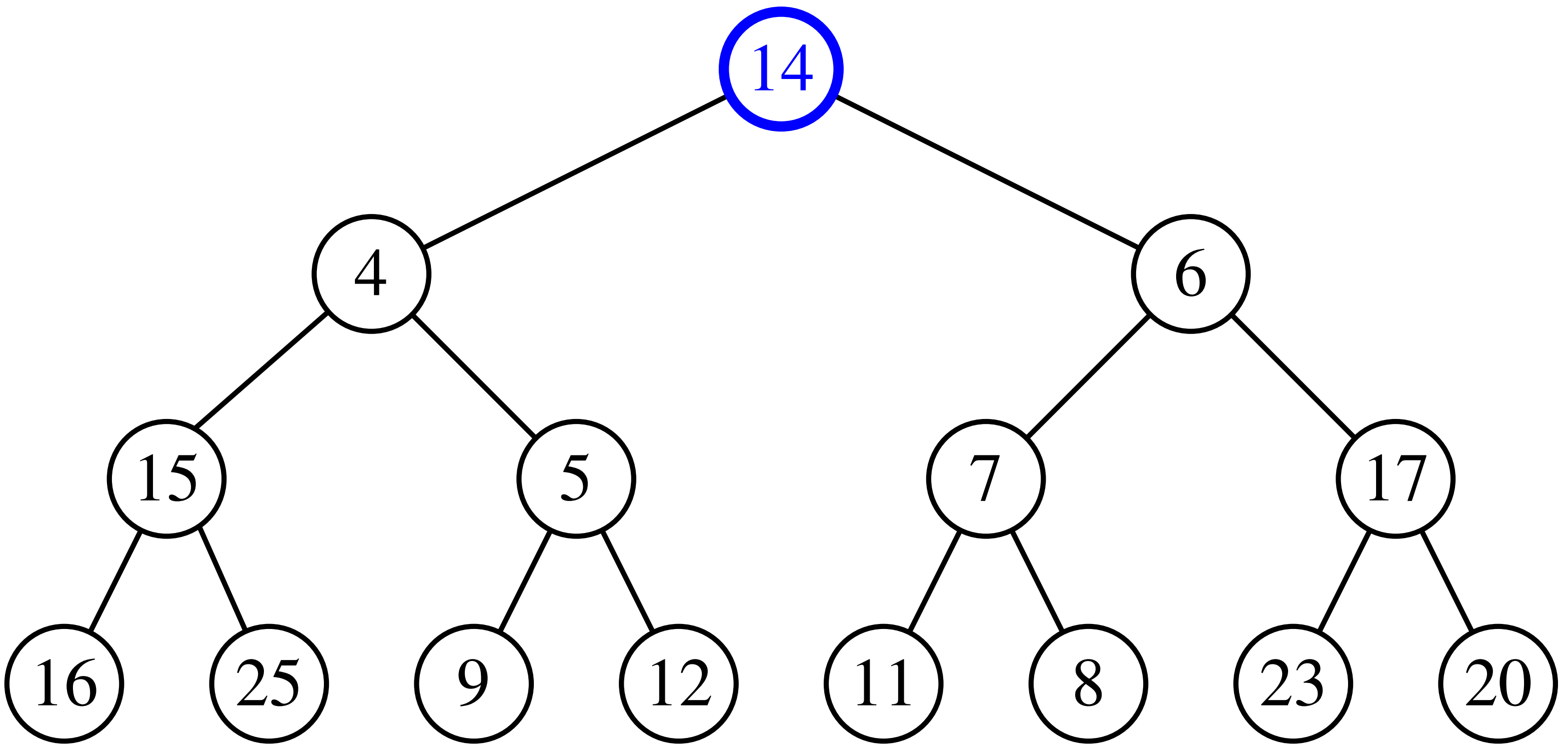
Building a Heap – Bottomup



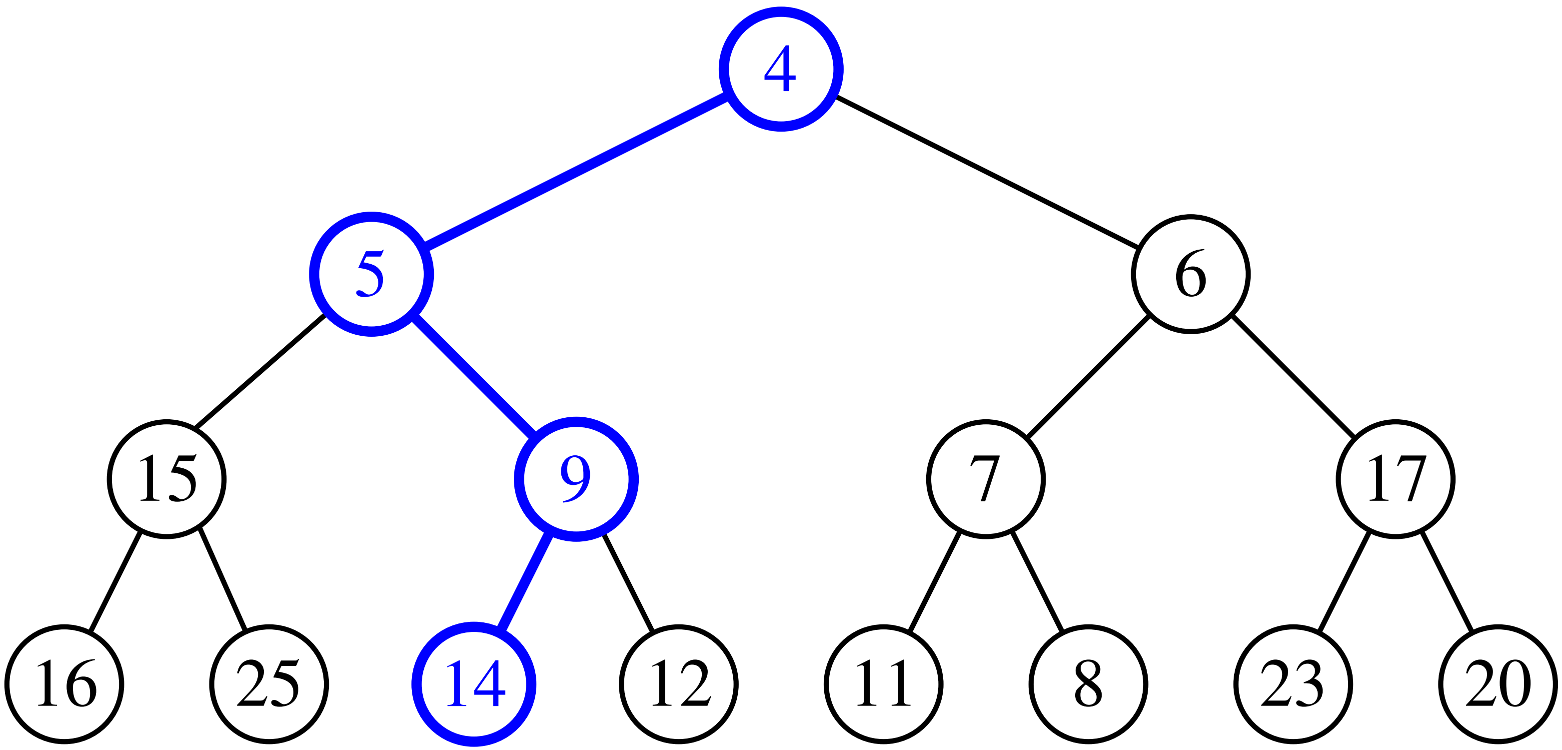
Building a Heap – Bottomup



Building a Heap – Bottomup

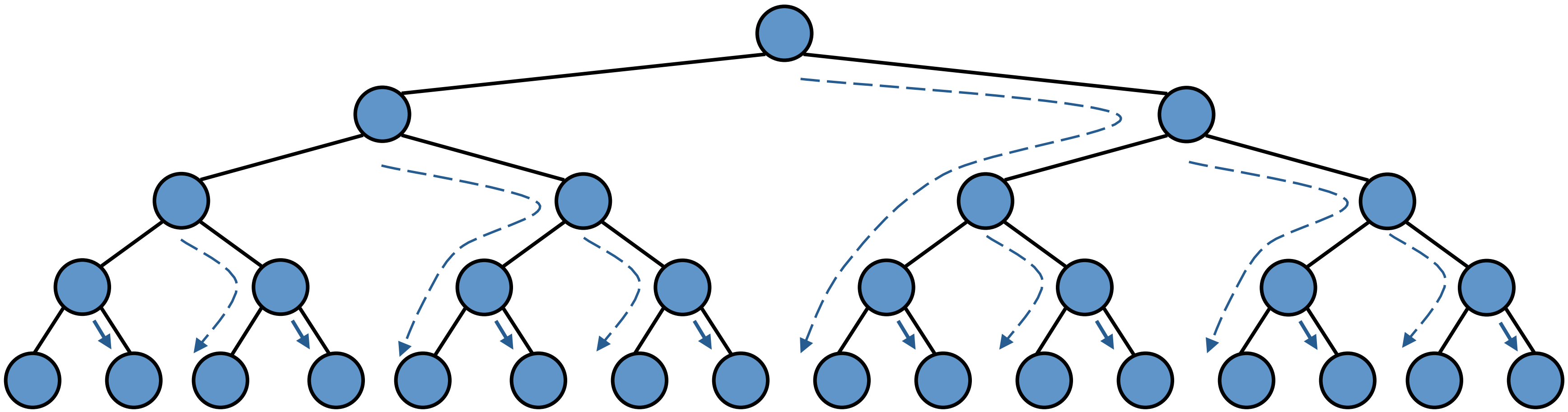


Building a Heap – Bottomup

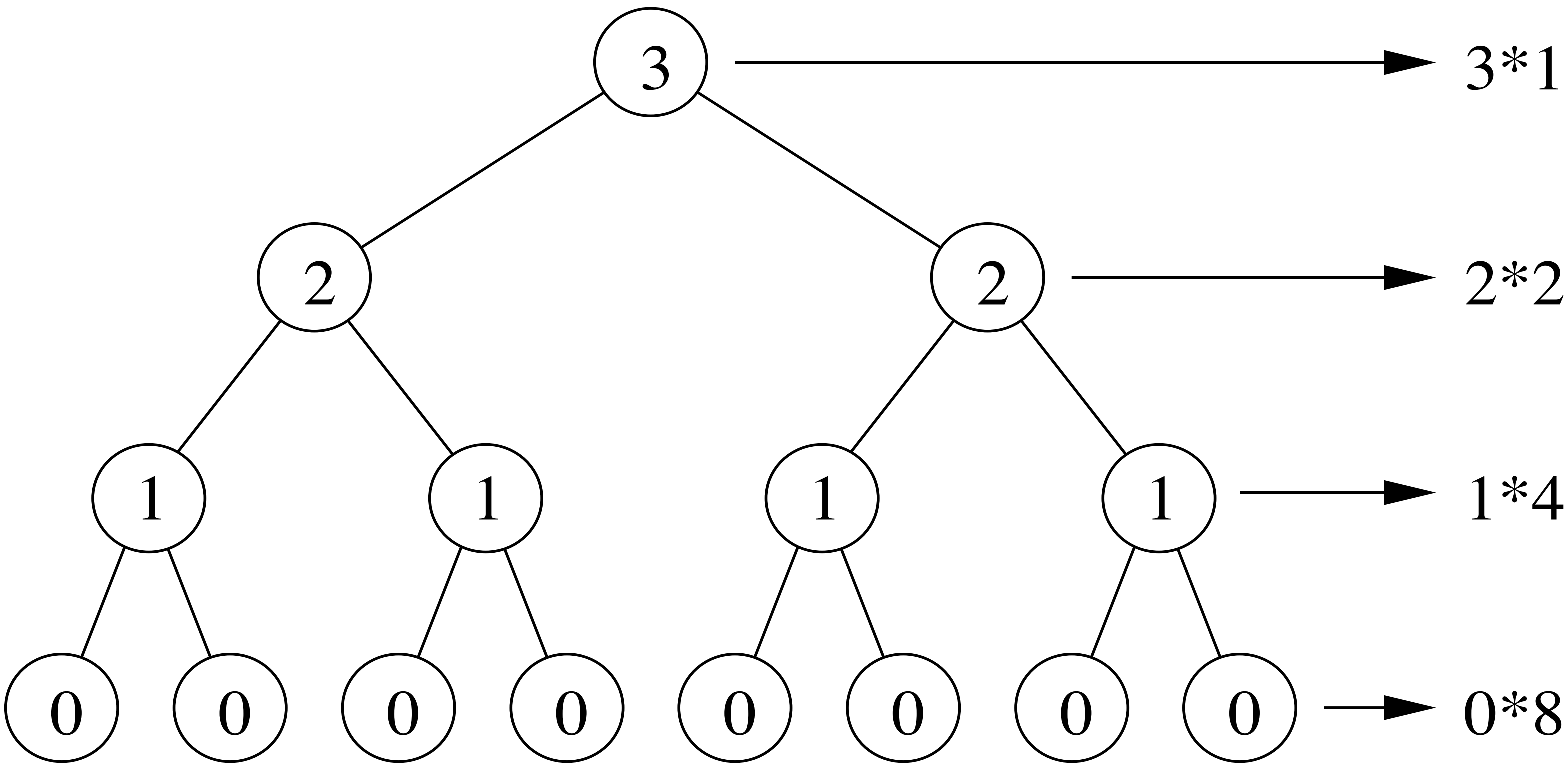


in phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

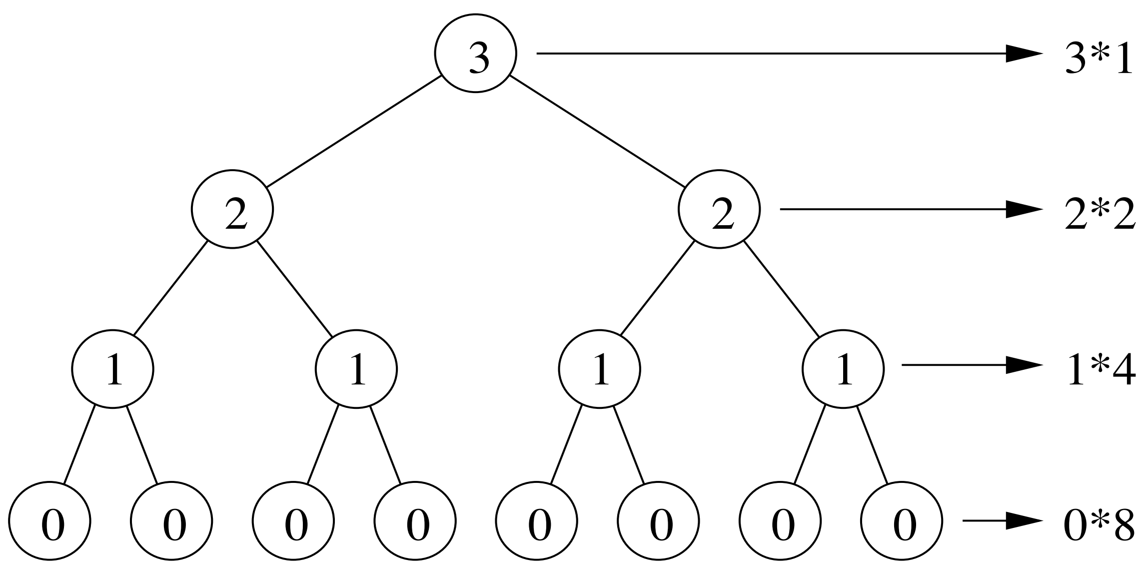
Analysis 1
-first right, then left until leaf



Analysis 2
-sum of heights of all nodes



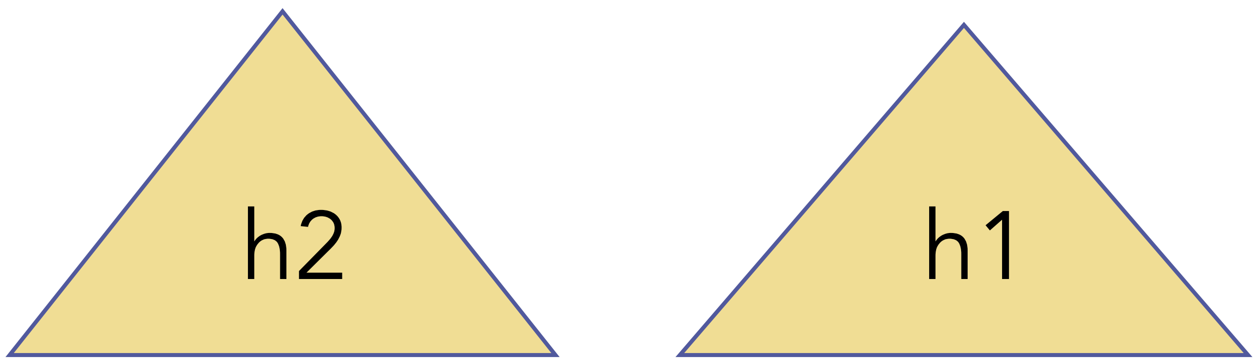
$$\sum_{j=0}^{h} j2^{h-j}$$

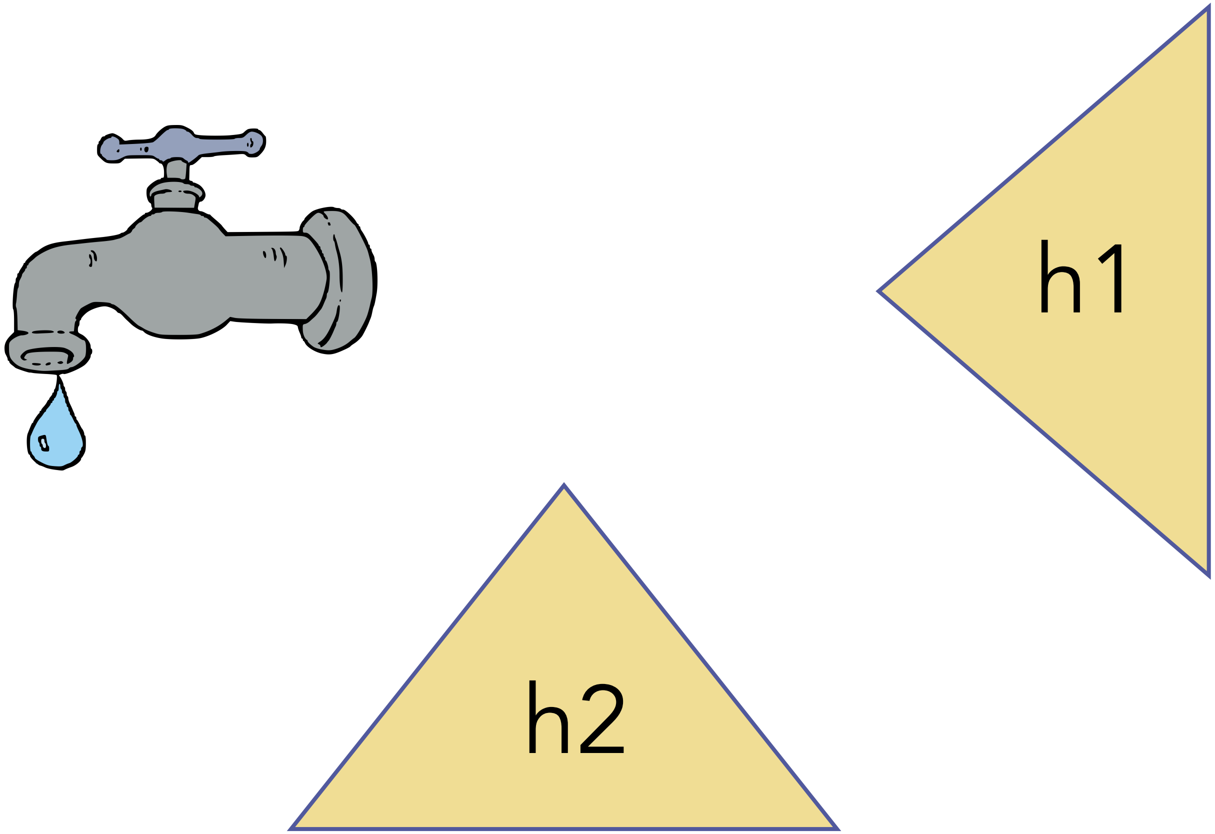


sum of heights

$O(n)$

How to merge two heaps?

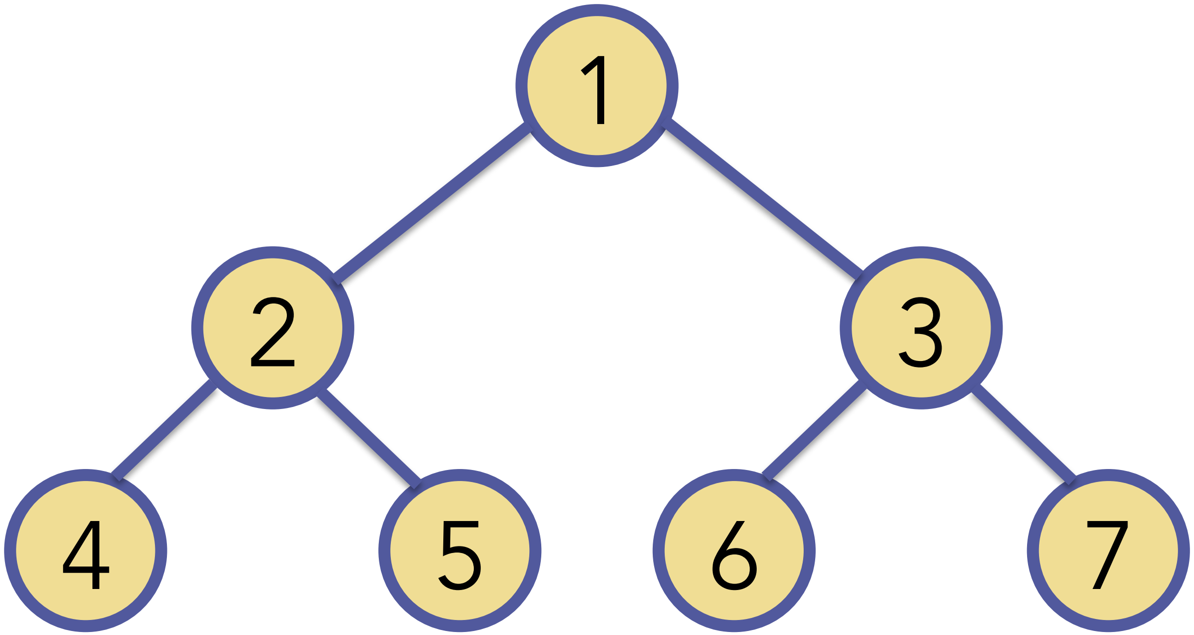

h2
h1



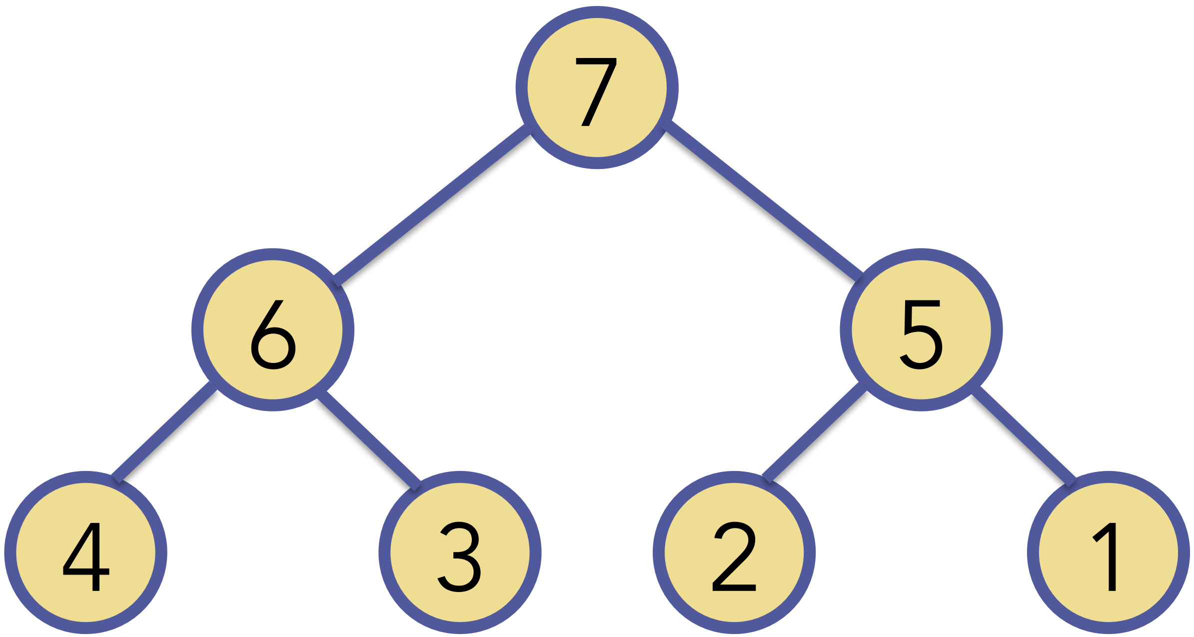
Time Complexity??

Merge two heaps in O(n)

Min Heap



Max Heap



Recall Priority Queue ADT

- a collection of entries
- entry = (key, value)
- main methods
  - insert(k, v)
  - removeMin()
- additional methods
  - min()
  - size(),
  - isEmpty()

Heap-Sort

```
while !L.empty ()
        e ← L.front();
        L.eraseFront()
        P.insert (e)
while !P.empty()
        e ← P.removeMin()
        L.insertBack(e)
```

Check if a given Binary Tree is Heap

Converting min-heap into max-heap

Implementing Heap-Sort In-Place

- Use left side array for heap and right side array for list
- move from left to right, and then right to left

| 4 7 2 1 3 6 5

4 | 7 2 1 3 6 5

4 7 | 2 1 3 6 5

7 4 | 2 1 3 6 5

⋮

7 4 6 1 3 2 5 |

What should be the next step?

What is the time complexity?

$n \log n + n \log n$

Can we make it faster?

$n + n \log n$

Bottom-up Heap Construction for a Linked List implementation

- BottomUpHeap(L):
- if L.empty() then return an empty heap
- $e \leftarrow L$.front()
- L.pop front()
- Split L into two lists, $L1$ and $L2$, each of size $(n - 1)/2$
- $T1 \leftarrow$ BottomUpHeap($L1$)
- $T2 \leftarrow$ BottomUpHeap($L2$)
- Create binary tree T with root r storing e, left subtree $T1$, and right subtree $T2$
- Perform a down-heap bubbling from the root r of T, if necessary
- return T

How to find top k students who will be allowed to go for 6 month internship?

E.g. A = [8.1, 7.2, 7.5, 9, 9.8, 10, 5.4], k = 3

Output = [10, 9.8, 9]

- Solution 1: Sort the numbers => $O(n \log n)$
- Solution 2: Select min k times => $O(nk)$
- Solution 3: Build min-heap=> $O(n+k \log n)$
- Solution 4: Build min-heap of first k elements, for rest:
  - if < root then ignore else replace root with the number and heapify
  - $O(k + (n-k) \log k)$
- More?

A company has n items ($w_{1..}w_n$) and wants to make packages of minimum weight W. Give a fast algorithms to do this in minimum number of steps.

- Sort, add first two, sort again, stop when first element in greater than W
  - O(xnlog n)
- Create min heap, compare W with the root, if smaller, remove two elements, add them, and insert into the heap, repeat the procedure
  - O(n+x log n)