CSE221 Data Structures
Lecture 15: Heaps and Priority Queues

Antoine Vigneron

antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

November 8, 2021

Introduction

- I updated attendance records. They can be found in the portal under E-attendance.
- Assignment 2 was graded by Hyeyun Yang (`gm1225@unist.ac.kr`).
- I graded the midterm. You should be able to see your answers on BB and your score for each question.
- I will post Assignment 3 this week.
- Reference for this lecture: Textbook Chapter 8.

Midterm

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

class myClass {
public:
  myClass(int m=0) : n(m){}
  ~myClass(){ cout << "calling destructor\n";}
  int n;
};
int main(){
  myClass A;
  myClass B(10);
  A=B;
  cout << "end of main function\n";
  return EXIT_SUCCESS;
}
```

Midterm

- Does this program call the destructor of `myClass`?
- Answer: Yes. The program outputs

```
end of main function
calling destructor
calling destructor
```

- Reason: The destructor is called when an object goes out of scope.
- Average score: 81/110
- You should be able to see your answers and detailed score in BB.
- Questions were randomized: Each of you had the same number of questions of each type, but they were taken from a random pool. So the question numbering on your exam and on the solutions are different.

Priority Queues

element	ICN	GMP	PUS	TAE	USN
key	1	2	2	3	3

Operation	Output	Priority queue
insert(GMP)		{GMP}
insert(ICN)		{GMP, ICN}
insert(USN)		{GMP, ICN, USN}
min()	ICN	{GMP, ICN, USN}
removeMin()		{GMP, USN}
insert(PUS)		{GMP, USN,PUS}
min()	GMP	{GMP, USN, PUS}
removeMin()		{USN, PUS}
removeMin()		{USN}
insert(TAE)		{USN, TAE}
min()	TAE	{USN, TAE}
removeMin()		{USN}

Priority Queues

- A *priority queue* is a container.
- Each element e is associated with a key k.
- The key could be an integer or a floating point number, for instance.
- A priority queue P supports the following operations:
- **insert**(e): Insert the element e (with an implicit associated key value) into P.
- **min**(): Return an element of P with the smallest associated key value, that is, an element whose key is less than or equal to that of every other element in P.
- **removeMin**(): Remove from P the element min().

Total Orders

- In the previous example, keys were integers. More generally, they could come from any set with a *total order* (also called *linear order*):

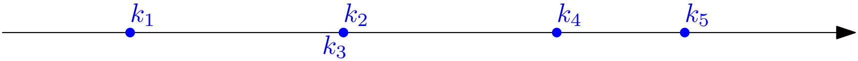
Definition

A set X associated with a relation $\leqslant$ is a total order iff for all $a, b, c \in X$:

- $a \leqslant a$ (reflexive)
- $a \leqslant b$ and $b \leqslant c$ implies $a \leqslant c$ (transitive)
- $a \leqslant b$ and $b \leqslant a$ implies $a = b$ (antisymmetric)
- $a \leqslant b$ or $b \leqslant a$ (strongly connected)

- Often,the keys will be numbers and these properties are obvious.

Total Orders



- Intuitively, having a total order means that you can place the keys along a line.
- In the picture above, we have

$$k_1 \leqslant k_2 \leqslant k_3 \leqslant k_4 \leqslant k_5$$
$$k_1 \leqslant k_3 \leqslant k_2 \leqslant k_4 \leqslant k_5$$

- We can also write it

$$k_1 < k_2 = k_3 < k_4 < k5.$$

where $a < b$ means $a \leqslant b$ and $a \neq b$.

Total Orders

- Another example: *Lexicographic order*.
- This is the order in which words are listed in a dictionary.
- It applies to strings:

Example

Alice $\leqslant$ Bob $\leqslant$ Boris $\leqslant$ Carol

- We can also apply it to points in 2D:

Example

$(0,0) \leqslant (0,1) \leqslant (1,0) \leqslant (1,1)$

Comparators

- For 2D points, we can implement lexicographic order by overloading the $<$ operator:

```
bool operator<(const Point2D& p, const Point2D& q) {
  if (p.getX() == q.getX())
    return p.getY() < q.getY();
  else
    return p.getX() < q.getX();
}
```

Comparators

- In the previous slide, we implemented this order relation:

$$(x, y) < (x', y') \text{ iff } \begin{cases} x < x' & \text{or} \\ x = x' \text{ and } y < y' \end{cases}$$

- Within the same program, we may want to use the order:

$$(x, y) < (x', y') \text{ iff } \begin{cases} y < y' & \text{or} \\ y = y' \text{ and } x < x' \end{cases}$$

  where we first compare the y-coordinate and break ties with the x-coordinates.

- Problem: We cannot overload the $<$ operator *twice*.

Comparators

- Solution: We can use *comparator classes* and overload the () operator.

```
class LeftRight {                    // a left-right comparator
public:
  bool operator()(const Point2D& p, const Point2D& q) const
    { return p.getX() < q.getX(); }
};
```

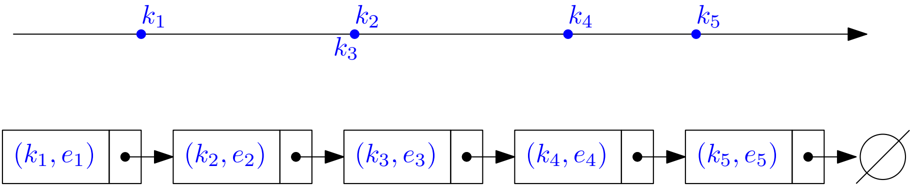
```
class BottomTop {                    // a bottom-top comparator
public:
  bool operator()(const Point2D& p, const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

Comparators

```
                              // element and comparator class
template <typename E, typename C>
void printSmaller(const E& p, const E& q, const C& isLess)
{
  cout << (isLess(p, q) ? p : q) << endl;
                              // print the smaller of p and q
}
```

```
Point2D p(1.3, 5.7), q(2.5, 0.6);                // two points
LeftRight leftRight;                   // a left-right comparator
BottomTop bottomTop;                   // a bottom-top comparator
printSmaller(p, q, leftRight);             // outputs: (1.3, 5.7)
printSmaller(p, q, bottomTop);             // outputs: (2.5, 0.6)
```

Implementing a Priority Queue using a List

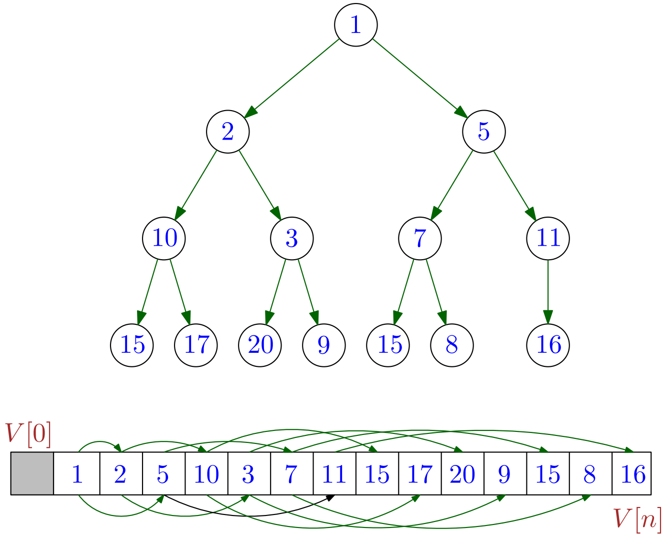


- Problem:

Operation	Unsorted list	Sorted list
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

- We will see a better implementation using *heaps*.

Heaps



Heaps

- A *heap* is a binary tree T that stores a collection of elements with their associated keys at its nodes

Property

A heap T with height h is a complete binary tree: each level i, $0 \leqslant i \leqslant n-1$, has the maximum number of nodes 2^i, and the nodes at level h fill this level from left to right.

Corollary

A heap storing n entries has height $h = \lfloor \log n \rfloor$.

Heaps

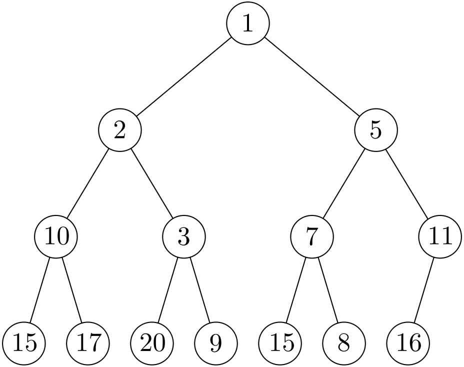
- The nodes of a heap have the *heap property*:

Property

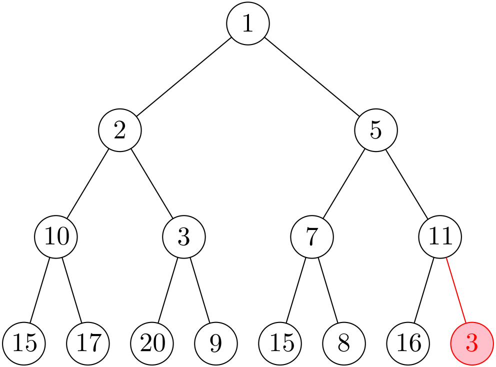
If v is the parent of w, then $\text{key}(v) \leqslant \text{key}(w)$.

- The heap is recorded in a vector $V[0, 1, \dots, n]$.
- $V[0]$ is not used.
- The root is at $V[1]$.
- The two children of $V[i]$ are $V[2i]$ and $V[2i+1]$.
- So the parent of $V[i]$ is $V[\lfloor i/2 \rfloor]$.

Insertion

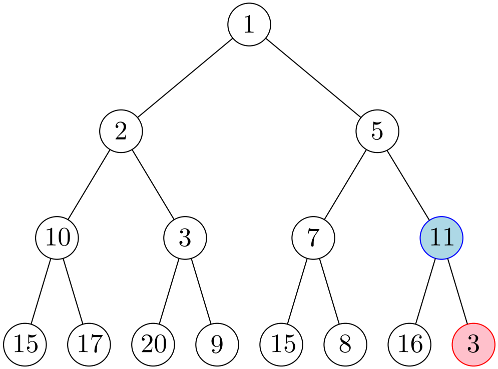


Insertion



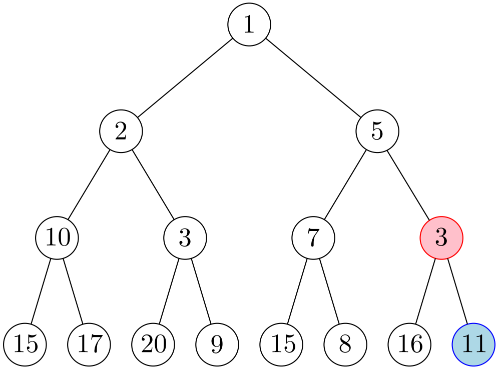
The new node is inserted at the last position

Insertion



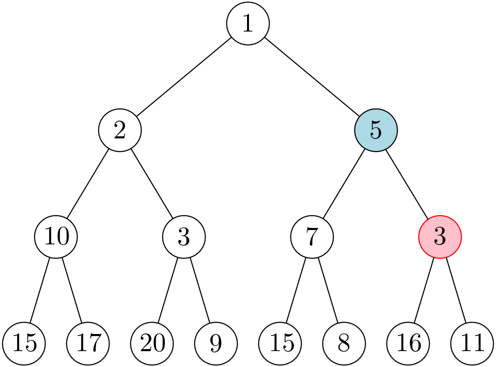
The heap property does not hold for the new node

Insertion



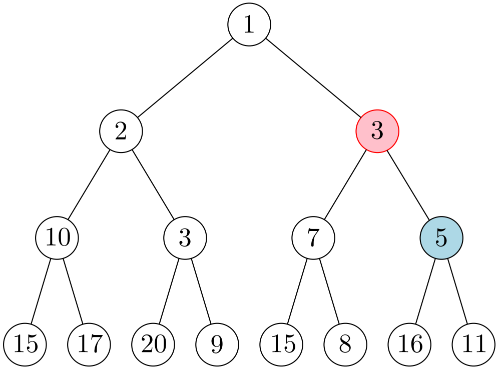
Fixing the heap

Insertion



The heap property does not hold

Insertion



Now the heap is fixed

Insertion

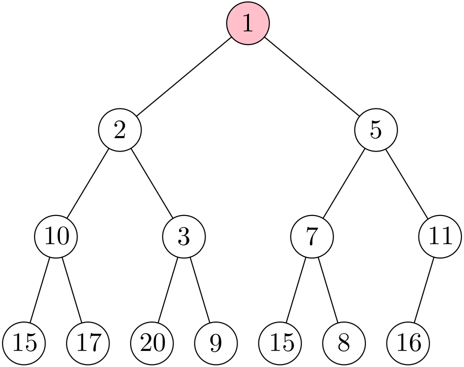
- If the heap contains n nodes, the new node is inserted at $V[n+1]$.
- Then we fix the heap by calling $\textsc{Heapify-up}(H, n+1)$

Pseudocode

procedure $\textsc{Heapify-up}(V, i)$
    if $i > 1$ **then**
        $p \leftarrow \lfloor i/2 \rfloor$ $\triangleright$ p is the parent of i
        if key$(V[p]) >$ key$(V[i])$ **then**
            exchange $V[i]$ with $V[p]$
            $\textsc{Heapify-up}(V, p)$

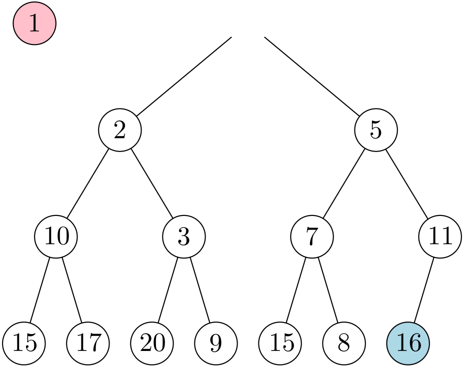
- It takes time $O(\log n)$ because i gets halved at each recursive call.

Removing the Minimum



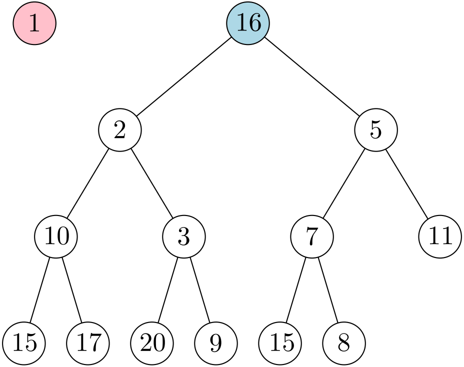
The minimum is at the root.

Removing the Minimum



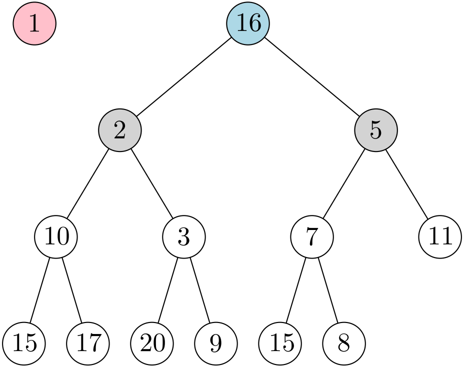
After we remove the minimum, a hole is left at the root.

Removing the Minimum



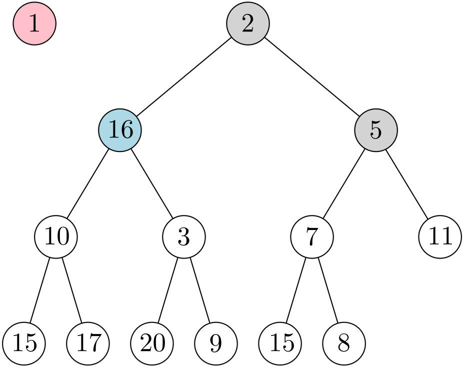
We move the last element to the root.

Removing the Minimum



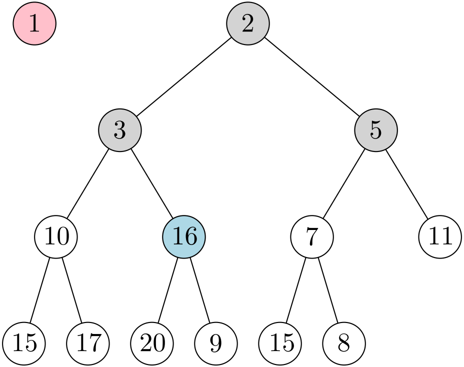
The heap property is violated.

Removing the Minimum



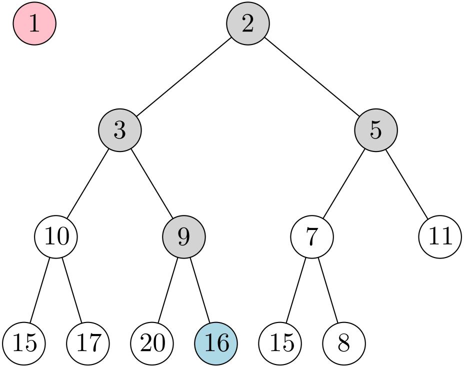
Fixing the heap.

Removing the Minimum



Fixing the heap.

Removing the Minimum



Now the heap is fixed.

Removing the Minimum

- The minimum is at the root node.
- So we first remove the root node.
- We replace it with the last node.
- We fix the heap property by calling HEAPIFY-DOWN(V).
  (See next slide.)

Removing the Minimum

Pseudocode

```
procedure HEAPIFY-DOWN(V)
    n ← V.size()
    i ← 1
    while 2i ⩽ n do
        j ← the index of the child of i with smallest key.
        if key(V[i]) > key(V[j]) then
            exchange V[i] with V[j]
            i ← j
        else
            return
```

- This procedure runs in time $O(\log n)$ because i becomes $2i$ or $2i+1$ at the end of each iteration of the WHILE loop.

Heap Operations

- As the height is $O(\log n)$, it follows that:

Proposition

A heap records a set of n elements using $O(n)$ space. We can insert a new element in $O(\log n)$ time, and remove the element with minimum key in $O(\log n)$ time.

- We can also delete any element $V[i]$ in $O(\log n)$ time:
  - First $V[i] \leftarrow V[n]$.
  - Then, if the key of $V[i]$ is smaller than its parent, call $\textsc{Heapify-up}(V, i)$
  - Otherwise, if the key of $V[i]$ is larger than one of its child, call a modified version of $\textsc{Heapify-down}$ that starts at $V[i]$.

Remarks

- What we described above is a *min heap*.
- In a *max heap*, the order is reversed: The key of a node is not larger than its parent, so the *largest* key is stored at the root of any subtree.
- A *max heap* allows us to remove the *maximum*, to insert and to delete an element in $O(\log n)$ time.
- We can sort a set of n numbers by inserting them all into a heap, and then removing the minimum repeatedly.
- It takes $O(n \log n)$ time.
- There is a slightly better algorithm for sorting using a heap, called HEAPSORT.
  - Use a *max heap*. (Why?)
  - All the elements can be inserted in $O(n)$ time, but we still need $\Theta(\log n)$ time for each removal. (Not covered in this lecture.)

Heap Implementation of a Priority Queue

Operation	Time
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

- So a sequence of n operations on an empty queue runs in $O(n \log n)$ time.
- This is a large improvement over a linked list implementation, which takes $\Theta(n^2)$ time in the worst case.