CH8. HEAPS AND PRIORITY QUEUES

CSED233 Data Structure

Prof. Hwanjo Yu
POSTECH

The Priority Queue ADT

- Keys, Priorities, and Total Order Relations
  - *Key: something assigned to an element to identify, rank, or weigh it.*
  - *For every pair of keys,*
    - **Reflexive property**: $k \le k$
    - **Antisymmetric property**: if $k_1 \le k_2$ and $k_2 \le k_1$, then $k_1 = k_2$
    - **Transitive property**: if $k_1 \le k_2$ and $k_2 \le k_3$, then $k_1 \le k_3$
  - *Thus, keys can be ordered linearly.*
  - *Priority Queue: a container of elements, each associated with a key, and the key determines the "priority" of the element.*
- Define a priority queue for each key type? => no
  - *Comparator approach*
    - Overload comparison operators
    - Define and use comparator objects
  - *Composition method: separate key from element*

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

```
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(); }
};
```

The Priority Queue ADT

- Interface functions

```
template <typename E, typename C>
class PriorityQueue {
public:
  int size() const;
  bool isEmpty() const;
  void insert(const E& e);
  const E& min() const throw(QueueEmpty);
  void removeMin() throw(QueueEmpty);
};
```

Operation	*Output*	*Priority Queue*
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	"error"	{}

The STL priority_queue Class

- Interface functions

size():
empty():

push(e):
top():

pop():

```
#include <queue>
using namespace std;                    // make std accessible
priority_queue<int> p1;                 // a priority queue of integers
                     // a priority queue of points with left-to-right order
priority_queue<Point2D, vector<Point2D>, LeftRight> p2;
```

```
priority_queue<Point2D, vector<Point2D>, LeftRight> p2;
p2.push( Point2D(8.5, 4.6) );                // add three points to p2
p2.push( Point2D(1.3, 5.7) );
p2.push( Point2D(2.5, 0.6) );
cout << p2.top() << endl; p2.pop();          // output: (8.5, 4.6)
cout << p2.top() << endl; p2.pop();          // output: (2.5, 0.6)
cout << p2.top() << endl; p2.pop();          // output: (1.3, 5.7)
```

Implementing a Priority Queue with a List

- Implementation with a list

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

- Implementing it with a sorted list

```
template <typename E, typename C>
class ListPriorityQueue {
public:
  int size() const;
  bool empty() const;
  void insert(const E& e);
  const E& min() const;
  void removeMin();
private:
  std::list<E> L;
  C isLess;
};
```

```
template <typename E, typename C>
void ListPriorityQueue<E,C>::insert(const E& e) {
  typename std::list<E>::iterator p;
  p = L.begin();
  while (p != L.end() && !isLess(e, *p)) ++p;
  L.insert(p, e);
}
```

```
template <typename E, typename C>
const E& ListPriorityQueue<E,C>::min() const
  { return L.front(); }

template <typename E, typename C>
void ListPriorityQueue<E,C>::removeMin()
  { L.pop_front(); }
```

Selection-Sort using Priority Queue

		List 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)	()

Figure 8.1: Execution of selection-sort on list $L = (7,4,8,2,5,3,9)$.

Insertion-Sort using Priority Queue

		List L	Priority Queue P
Input		$(7,4,8,2,5,3,9)$	$()$
Phase 1	(a)	$(4,8,2,5,3,9)$	(7)
	(b)	$(8,2,5,3,9)$	$(4,7)$
	(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)	$(2,3)$	$(4,5,7,8,9)$
	$\vdots$	$\vdots$	$\vdots$
	(g)	$(2,3,4,5,7,8,9)$	$()$

Figure 8.2: Execution of insertion-sort on list $L = (7,4,8,2,5,3,9)$. In Phase 1, we repeatedly remove the first element of L and insert it into P, by scanning the list implementing P until we find the correct place for this element. In Phase 2, we repeatedly perform removeMin operations on P, each of which returns the first element of the list implementing P, and we add the element at the end of L.

The Heap Data Structure (not the “heap” memory)

- Heap
  - *A binary tree that stores a collection of (key, value)*
  - ***Heap-Order Property**: key of $v \leq$ key of v’s parent (the minimum key)*
    - The maximum key simply changes isLess(x,y) returning true when x $\geq$ y
  - ***Complete Binary Tree Property**: to have a small height*
    - Every level except level h has max # of nodes 2^{i-1}, for $0 \leq i \leq h-1$
    - The nodes at level h fill from left to right
    - The heap storing n entries has height $h = \lfloor \log n \rfloor$

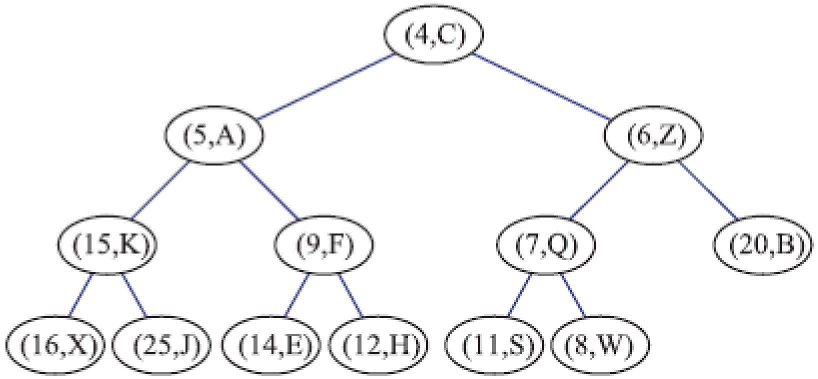

Figure 8.3: Example of a heap storing 13 elements. Each element is a key-value pair of the form (k, v). The heap is ordered based on the key value, k, of each element.

Complete Binary Trees

- Complete Binary Tree ADT

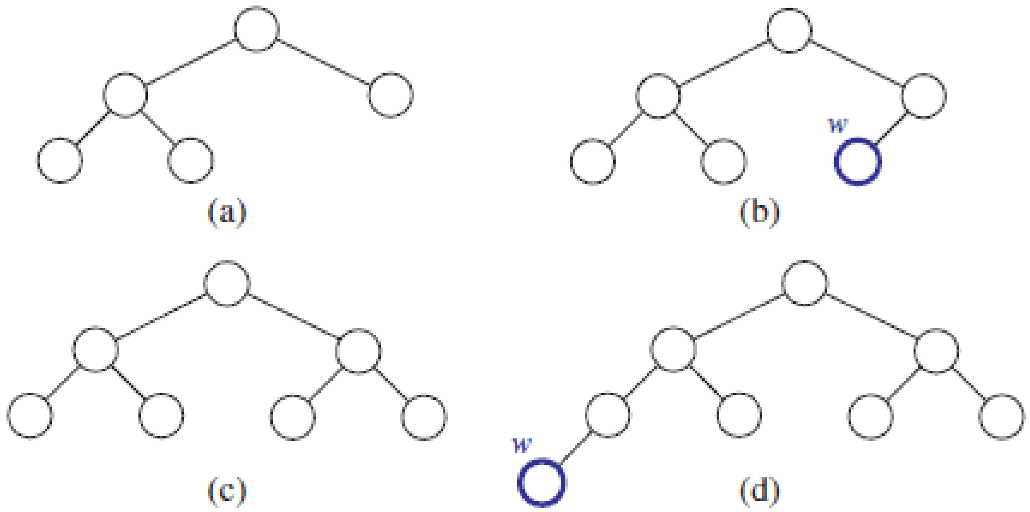

Figure 8.4: Examples of operations add and remove on a complete binary tree, where w denotes the node inserted by add or deleted by remove. The trees shown in (b) and (d) are the results of performing add operations on the trees in (a) and (c), respectively. Likewise, the trees shown in (a) and (c) are the results of performing remove operations on the trees in (b) and (d), respectively.

Complete Binary Trees

- Vector Representation
  - *If v is root, $f(v) = 1$*
  - *If v is left child of u, $f(v) = 2f(u)$*
  - *If v is right child of u, $f(v) = 2f(u) + 1$*

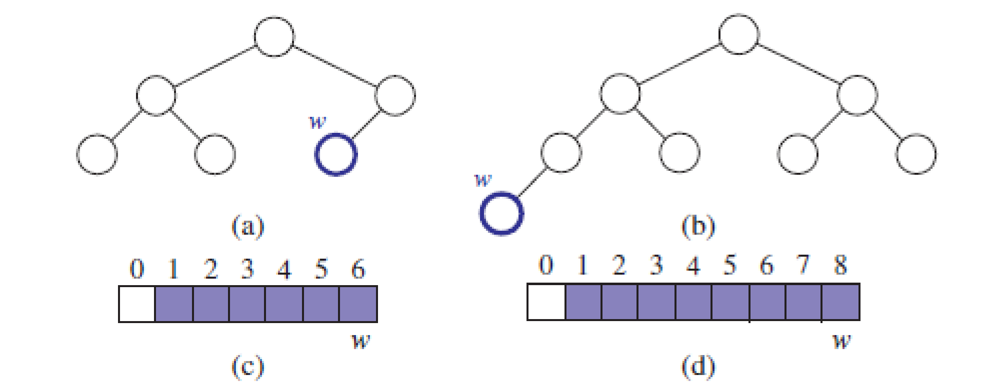

Figure 8.5: Two examples showing that the last node w of a heap with n nodes has level number n: (a) heap T_1 with more than one node on the bottom level; (b) heap T_2 with one node on the bottom level; (c) vector-based representation of T_1; (d) vector-based representation of T_2.

  - *Function add() and remove() take O(1)*

A C++ Implementation of a Complete Binary Trees

```
template <typename E>
class VectorCompleteTree {
  //... insert private member data and protected utilities here
public:
  VectorCompleteTree() : V(1) {}              // constructor
  int size() const                            { return V.size() − 1; }
  Position left(const Position& p)            { return pos(2*idx(p)); }
  Position right(const Position& p)           { return pos(2*idx(p) + 1); }
  Position parent(const Position& p)          { return pos(idx(p)/2); }
  bool hasLeft(const Position& p) const       { return 2*idx(p) <= size(); }
  bool hasRight(const Position& p) const      { return 2*idx(p) + 1 <= size(); }
  bool isRoot(const Position& p) const        { return idx(p) == 1; }
  Position root()                             { return pos(1); }
  Position last()                             { return pos(size()); }
  void addLast(const E& e)                    { V.push_back(e); }
  void removeLast()                           { V.pop_back(); }
  void swap(const Position& p, const Position& q)
                                              { E e = *q; *q = *p; *p = e; }
};
```

Code Fragment 8.13: A vector-based implementation of the complete tree ADT.

```
private:                                       // member data
  std::vector<E> V;                            // tree contents
public:                                        // publicly accessible types
  typedef typename std::vector<E>::iterator Position; // a position in the tree
protected:                                     // protected utility functions
  Position pos(int i)                          // map an index to a position
    { return V.begin() + i; }
  int idx(const Position& p) const             // map a position to an index
    { return p − V.begin(); }
```

Code Fragment 8.12: Member data and private utilities for a complete tree class.

Implementing a Priority Queue with a Heap

- Heap
  - *A complete binary tree satisfying the heap-order property*
- Insertion
  - *Insert e using add(e) which adds e to the last position of T => satisfying complete binary tree*
  - *Do something to satisfy the heap-order property => keeping swapping with parent until no violation => up-heap bubbling*
- Removal
  - *Remove the top element => violation of complete binary tree*
  - *Move the last node to the root => satisfying complete binary tree but violation of heap-order property*
  - *Do something to satisfy the heap-order property => keeping swapping with smaller child until no violation => down-heap bubbling*
- Analysis
  - *Insert(e): O(log n)*
  - *removeMin(): O(log n)*

Implementing a Priority Queue with a Heap

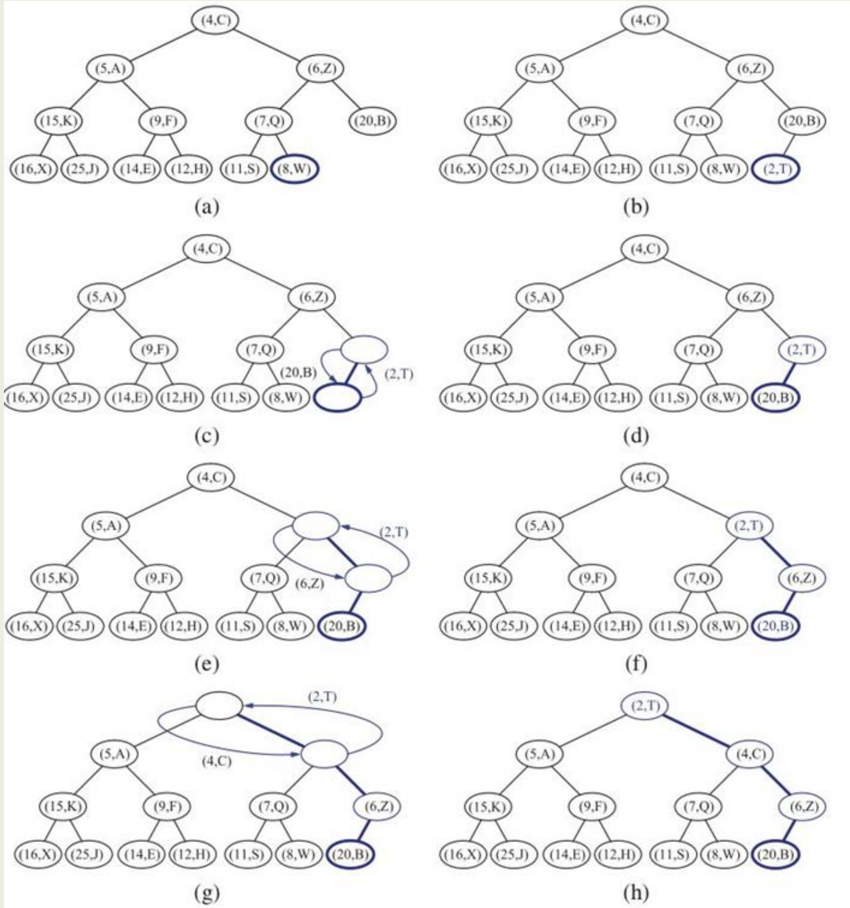

Figure 8.7: Insertion of a new entry with key 2 into the heap of Figure8.6: (a) initial heap; (b) after performing operation add; (c) and (d) swap to locally restore the partial order property; (e) and (f) another swap; (g) and (h)final swap.

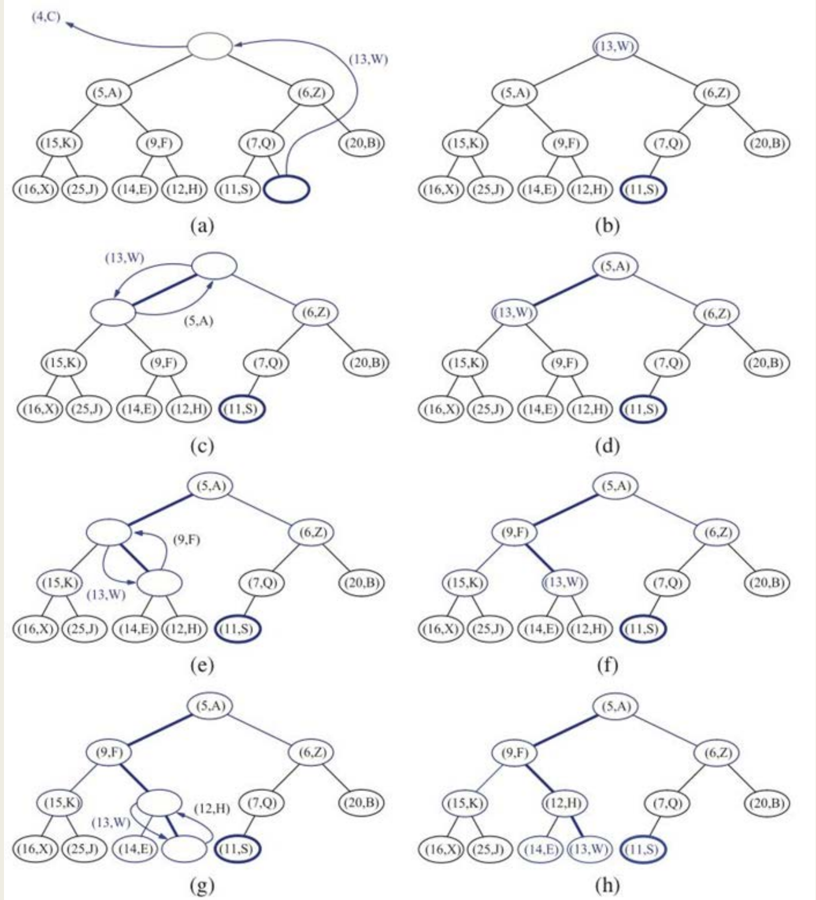

Figure 8.8: Removing the element with the smallest key from a heap: (a) and (b) deletion of the last node, whose element is moved to the root; (c) and (d) swap to locally restore the heap-order property; (e) and (f) another swap; (g) and (h) final swap.

Implementing a Priority Queue with a Heap

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
  int size() const;                          // number of elements
  bool empty() const;                        // is the queue empty?
  void insert(const E& e);                   // insert element
  const E& min();                            // minimum element
  void removeMin();                          // remove minimum
private:
  VectorCompleteTree<E> T;                   // priority queue contents
  C isLess;                                  // less-than comparator
                                             // shortcut for tree position
  typedef typename VectorCompleteTree<E>::Position Position;
};
```

```
template <typename E, typename C>            // insert element
void HeapPriorityQueue<E,C>::insert(const E& e) {
  T.addLast(e);                              // add e to heap
  Position v = T.last();                     // e's position
  while (!T.isRoot(v)) {                     // up-heap bubbling
    Position u = T.parent(v);
    if (!isLess(*v, *u)) break;              // if v in order, we're done
    T.swap(v, u);                            // ...else swap with parent
    v = u;
  }
}
```

```
template <typename E, typename C>            // remove minimum
void HeapPriorityQueue<E,C>::removeMin() {
  if (size() == 1)                           // only one node?
    T.removeLast();                          // ...remove it
  else {
    Position u = T.root();                   // root position
    T.swap(u, T.last());                     // swap last with root
    T.removeLast();                          // ...and remove last
    while (T.hasLeft(u)) {                   // down-heap bubbling
      Position v = T.left(u);
      if (T.hasRight(u) && isLess(*(T.right(u)), *v))
        v = T.right(u);                      // v is u's smaller child
      if (isLess(*v, *u)) {                  // is u out of order?
        T.swap(u, v);                        // ...then swap
        u = v;
      }
      else break;                            // else we're done
    }
  }
}
```

In-Place Heap Sort

1. Insert to make max-heap
2. Remove from top and fill the array from the right side

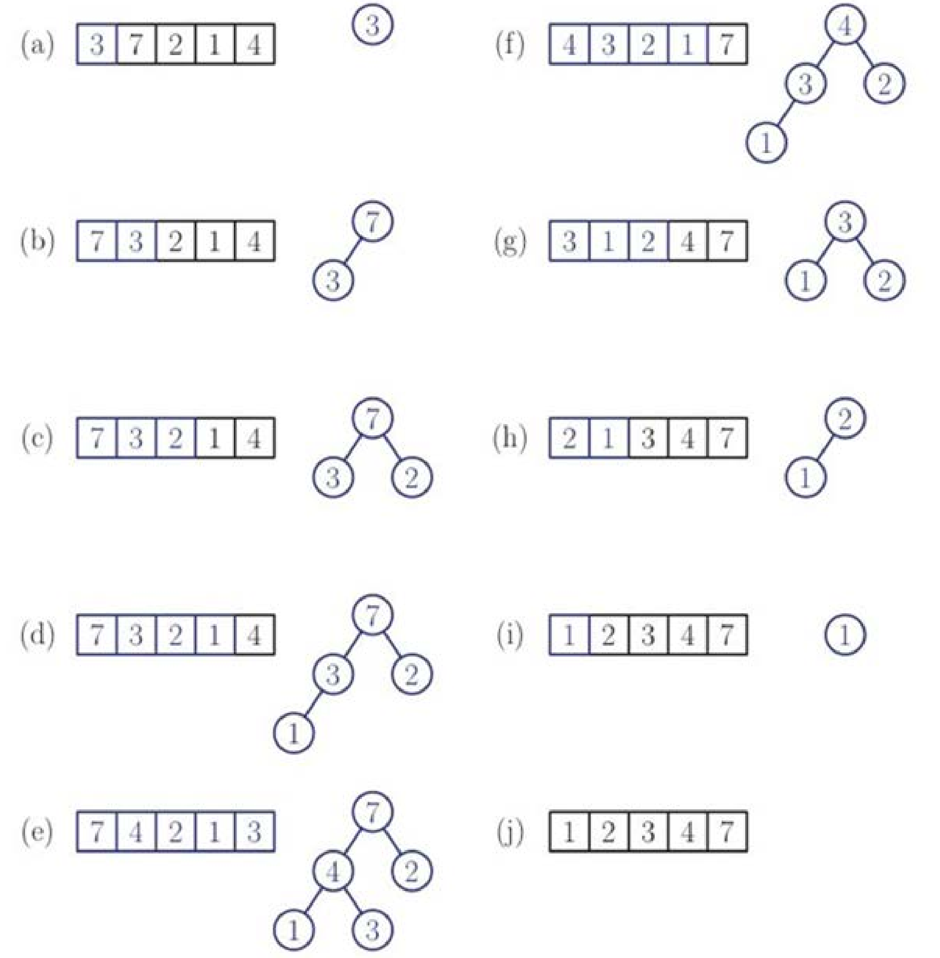

Figure 8.9: In-place heap-sort. Parts (a) through (e) show the addition of elements to the heap; (f) through (j) show the removal of successive elements. The portions of the array that are used for the heap structure are shown in blue.