Data Structures & Programming

Heap

Golnar Sheikhshab

Heaps

A binary Tree with two extra properties:

Heap-Order Property: In a heap T, for every node v other than the root, the key associated with v is greater than or equal to the key associated with v's parent.

Complete Binary Tree Property: A heap T with height h is a complete binary tree, that is, levels 0,1,2,...,h−1 of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \le i \le h - 1$) and the nodes at level h fill this level from left to right.

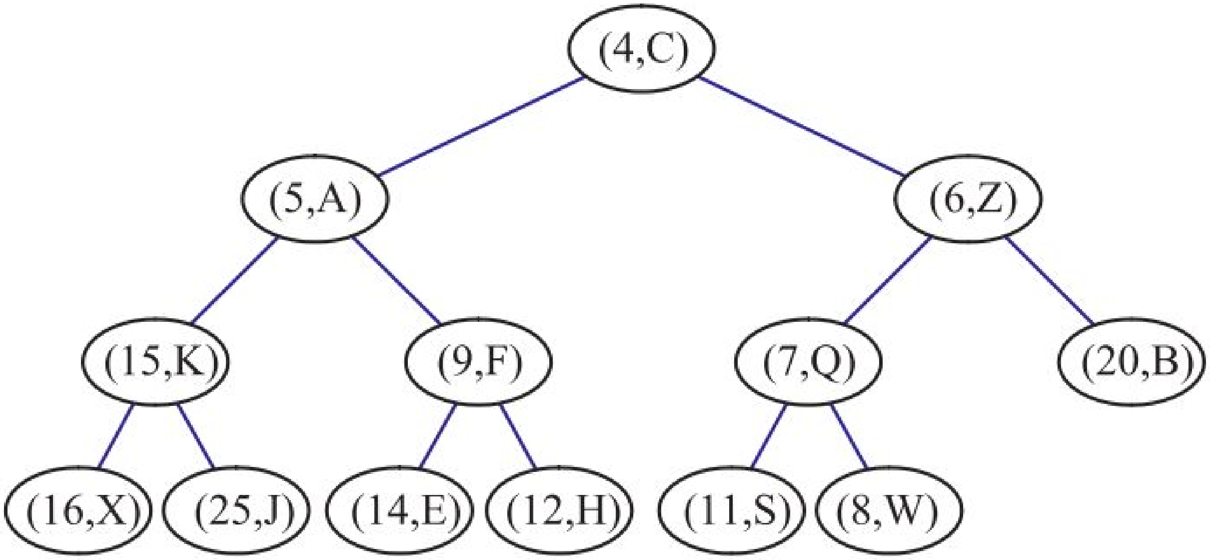


Figure 8.3: Example of a heap storing 13 elements. Each element is a key-value pair of the form (k, v). The heap is ordered based on the key value, k, of each element.

Note

1. We defined a min-heap. One can define a max-heap similarly (Like in STL priority Queue).
2. The element with minimum key is on top of the heap (at the root) and in each path from root to an external node, keys are ordered non-decreasingly.
3. Do not confuse the data structure heap with the freestore memory heap (Section 14.1.1) used in the run-time environment supporting programming languages like C++.

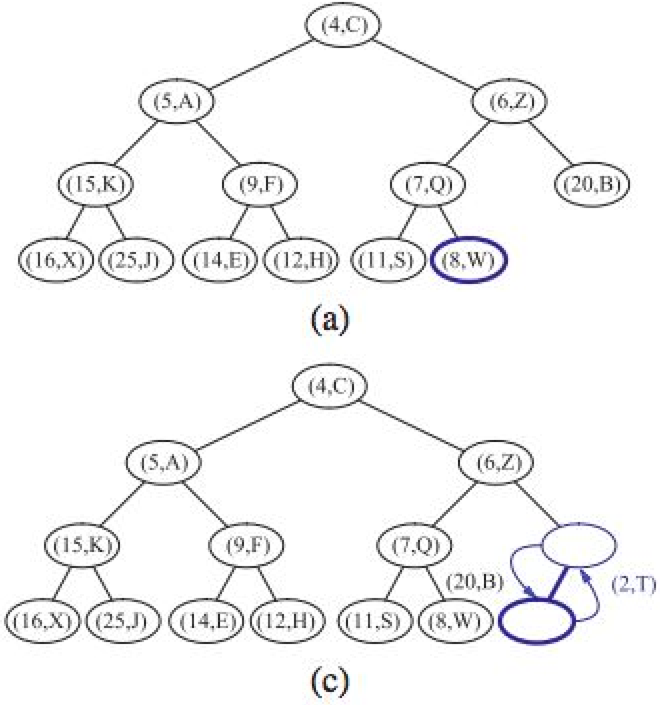
The Height of a Heap

Proposition 8.5: *A heap T storing n entries has height*

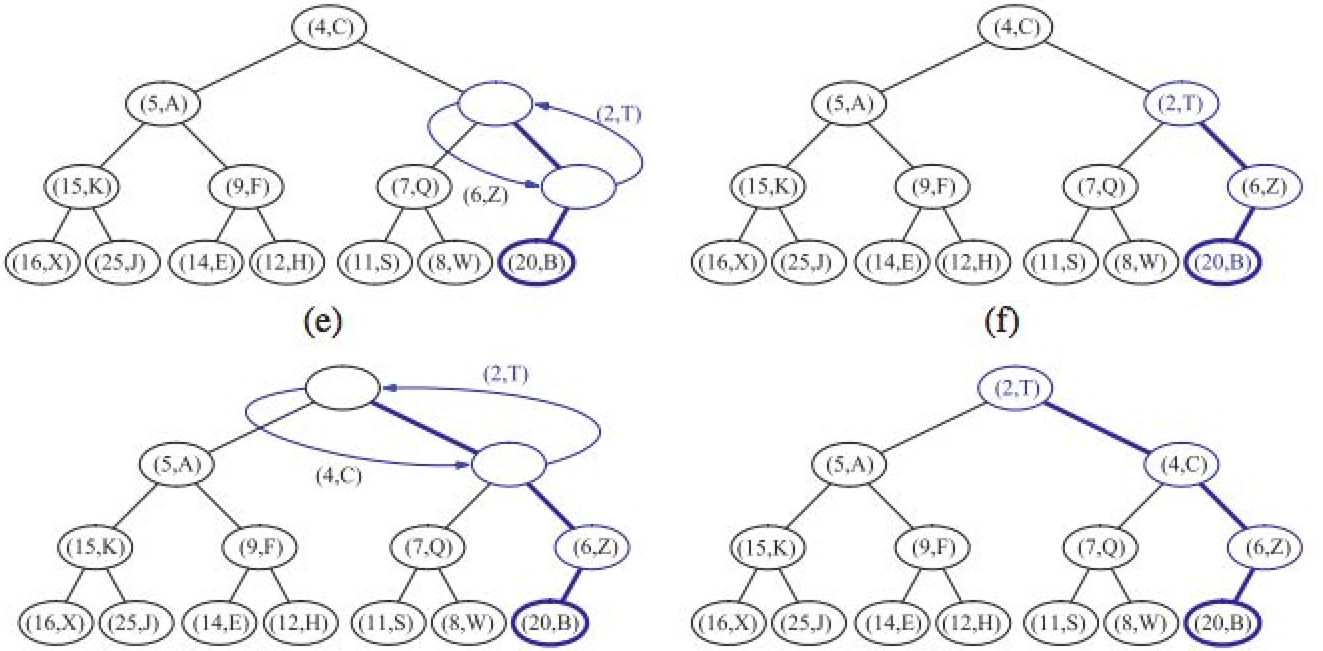
$$h = \lfloor \log n \rfloor .$$

We like this property because it helps us do both add(e) and remove_min() in O(log n)

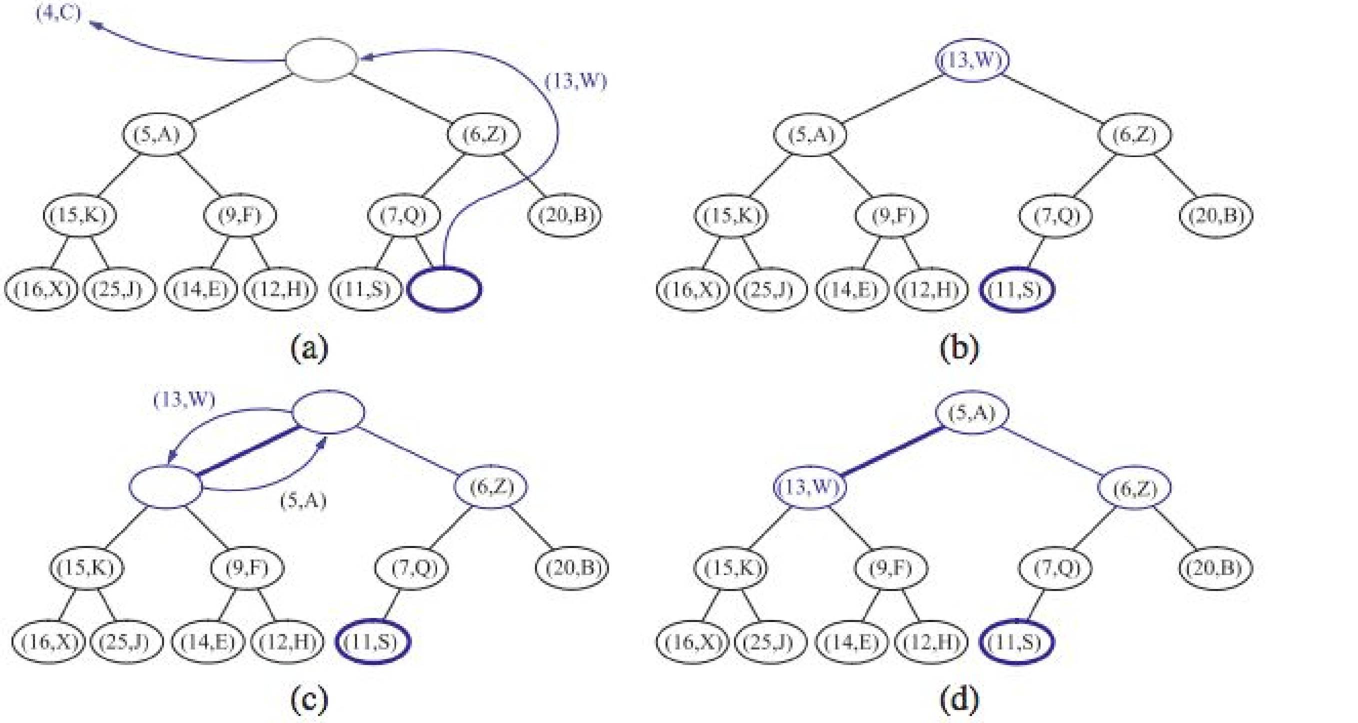
Insertion to a Heap (1)



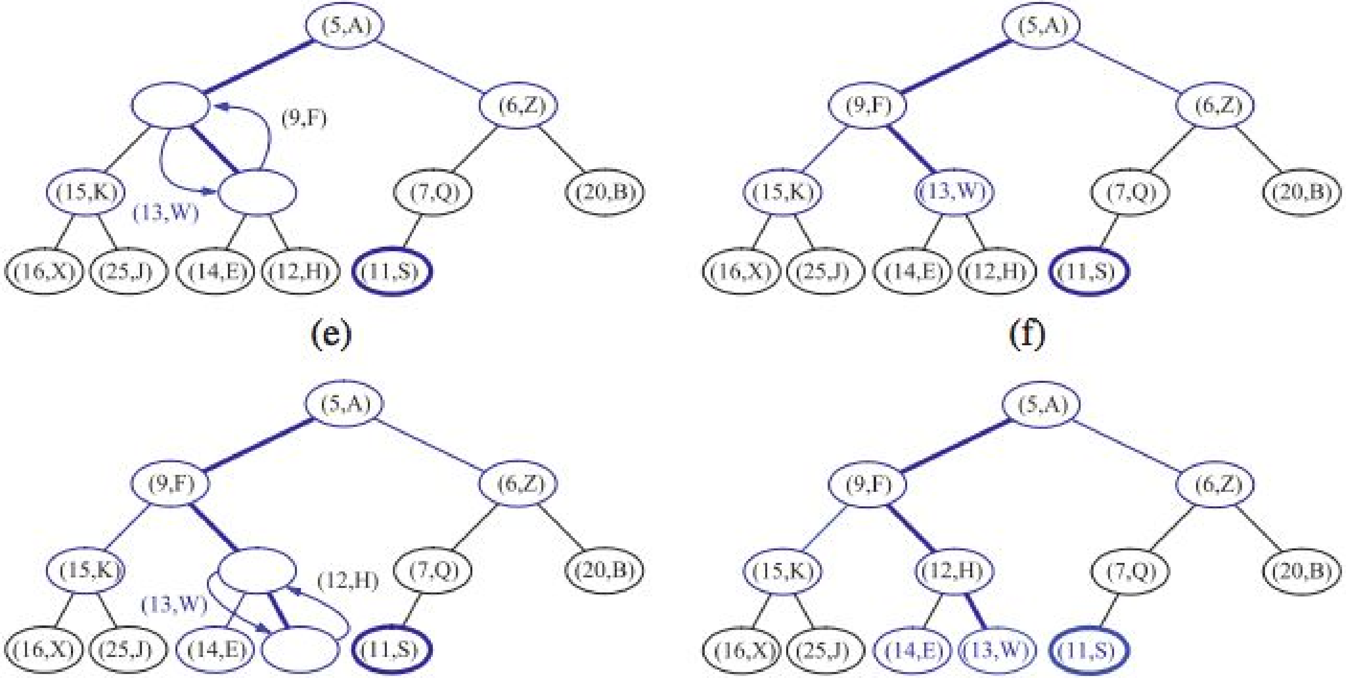
Insertion to a Heap (2)



Removal (1)


(4,C)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(a)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(b)
(13,W)
(5,A)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(c)
(5,A)
(13,W)
(6,Z)
(15,K)
(9,F)
(7,Q)
(20,B)
(16,X)
(25,J)
(14,E)
(12,H)
(11,S)
(d)

Removal (2)



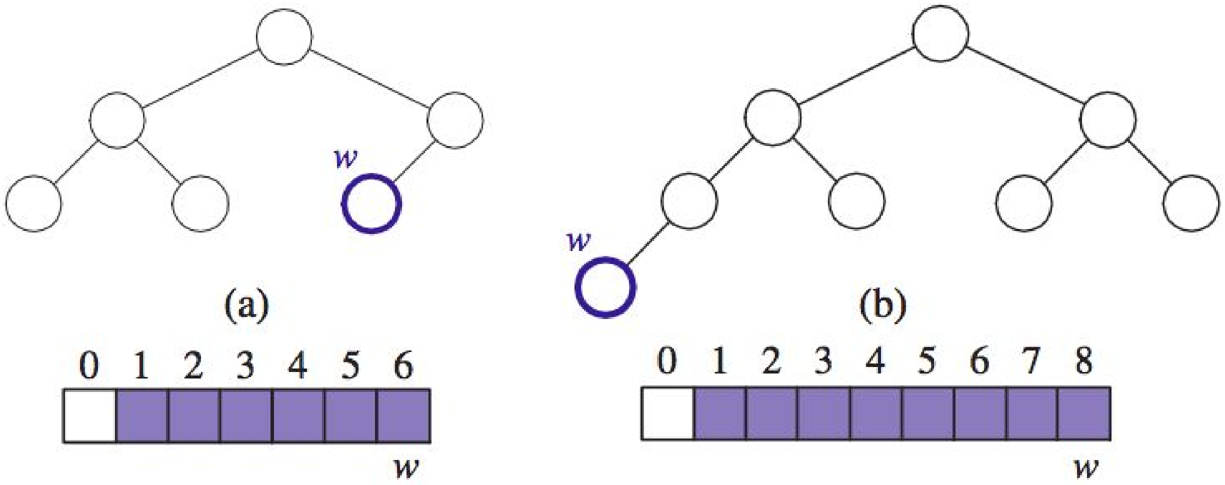
Complexity Aanalysis

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

Table 8.2: Performance of a priority queue realized by means of a heap, which is in turn implemented with a vector or linked structure. We denote with n the number of entries in the priority queue at the time a method is executed. The space requirement is $O(n)$. The running time of operations insert and removeMin is worst case for the array-list implementation of the heap and amortized for the linked representation.

Heap Implementation

Usually array/vector based



C++ Implementation

```
template <typename E>
class CompleteTree {                          // left-complete tree interface
public:                                       // publicly accessible types
  class Position;                             // node position type
  int size() const;                           // number of elements
  Position left(const Position& p);           // get left child
  Position right(const Position& p);          // get right child
  Position parent(const Position& p);         // get parent
  bool hasLeft(const Position& p) const;      // does node have left child?
  bool hasRight(const Position& p) const;     // does node have right child?
  bool isRoot(const Position& p) const;       // is this the root?
  Position root();                            // get root position
  Position last();                            // get last node
  void addLast(const E& e);                   // add a new last node
  void removeLast();                          // remove the last node
  void swap(const Position& p, const Position& q); // swap node contents
};
```

Code Fragment 8.11: Interface CompleteBinaryTree for a complete binary tree.

C++ Implementation

```
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.

```
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.

```
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;
};
```

Code Fragment 8.14: A heap-based implementation of a priority queue.

```
template <typename E, typename C>     // number of elements
int HeapPriorityQueue<E,C>::size() const
  { return T.size(); }

template <typename E, typename C>     // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
  { return size() == 0; }

template <typename E, typename C>     // minimum element
const E& HeapPriorityQueue<E,C>::min()
  { return *(T.root()); }             // return reference to root element
```

Code Fragment 8.15: The member functions size, empty, and min.

```
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;
  }
}
```

Code Fragment 8.16: An implementation of the function insert.

```
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
    }
  }
}
```

Code Fragment 8.17: A heap-based implementation of a priority queue.

Reading Material

8.3.1 - 8.3.4