Data Structures & Programming

More on Heap and Priority Queue

Golnar Sheikhshab

Heap Sort

PriorityQueueSort on a Heap-based Priotity queue

Algorithm PriorityQueueSort(L, P):
Input: An STL list L of n elements and a priority queue, P, that compares elements using a total order relation
Output: The sorted list L

```
while !L.empty() do
  e ← L.front
  L.pop_front()          {remove an element e from the list}
  P.insert(e)            {...and it to the priority queue}
while !P.empty() do
  e ← P.min()
  P.removeMin()          {remove the smallest element e from the queue}
  L.push_back(e)         {...and append it to the back of L}
```

Code Fragment 8.5: Algorithm PriorityQueueSort, which sorts an STL list L with the aid of a priority queue P.

Heap Sort (in-place)

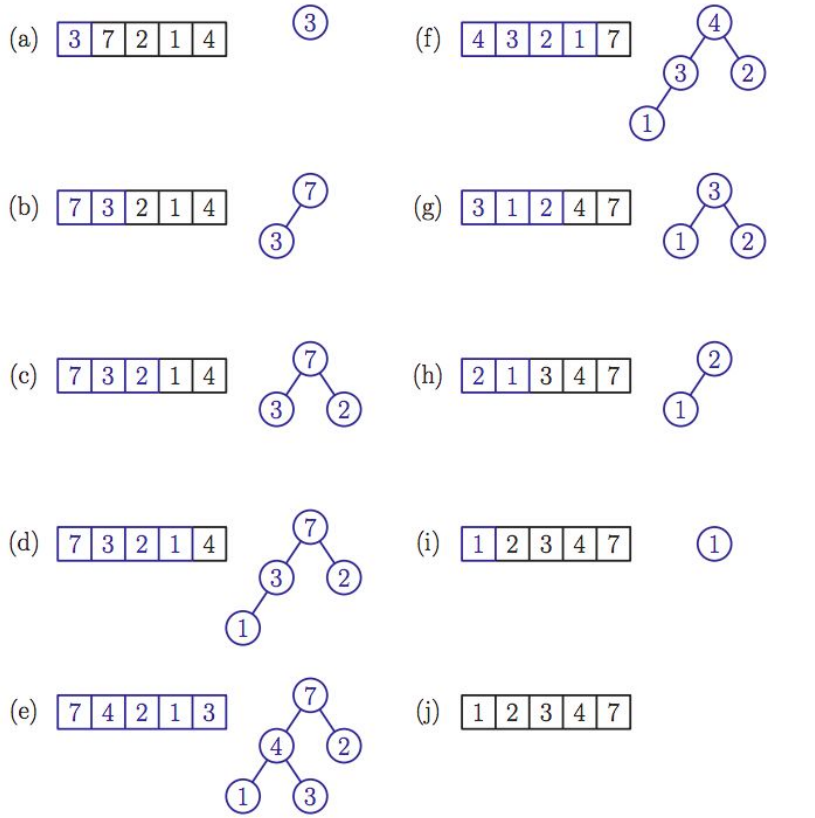
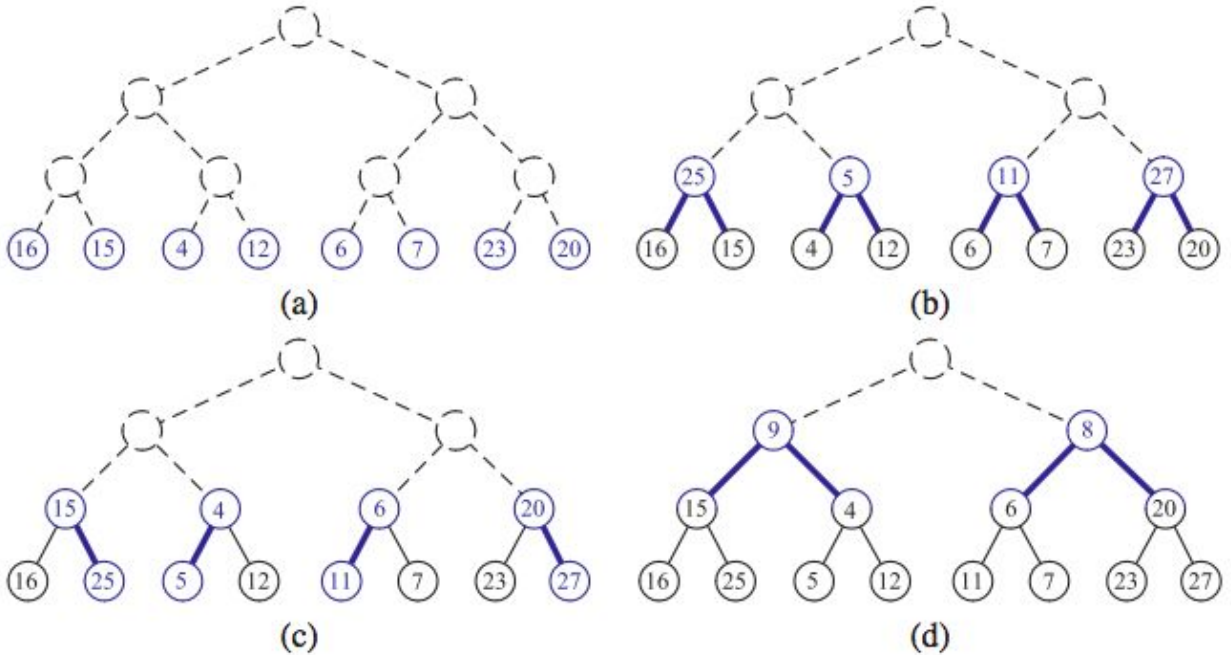
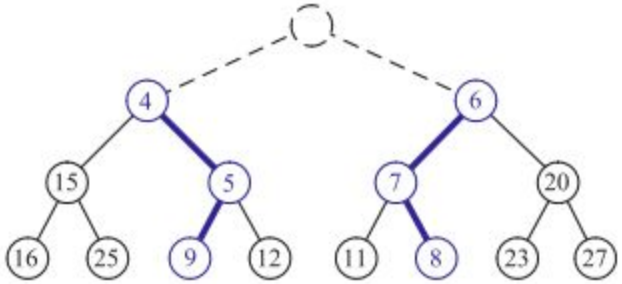


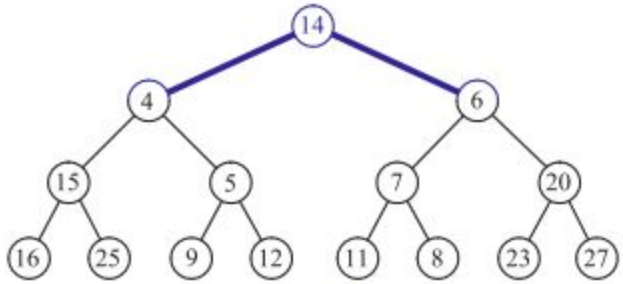
Figure 8.9: In-place heap-sort. Parts (a) through (e) show the addition of elements to the heap; (f) through (j) show the removal of successive elements. The portions of the array that are used for the heap structure are shown in blue.

Bottom-Up Heap Construction

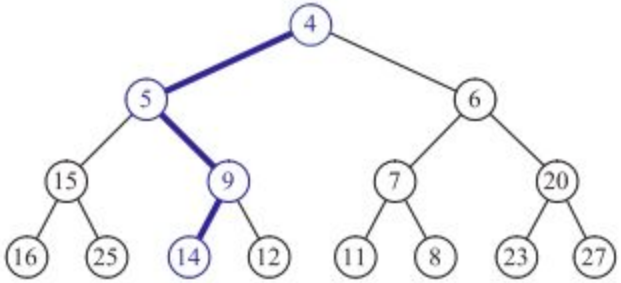




(e)



(f)



(g)

Figure 8.10: Bottom-up construction of a heap with 15 entries: (a) we begin by constructing one-entry heaps on the bottom level; (b) and (c) we combine these heaps into three-entry heaps; (d) and (e) seven-entry heaps; (f) and (g) we create the final heap. The paths of the down-heap bubblings are highlighted in blue. For simplicity, we only show the key within each node instead of the entire entry.

Bottom-Up Heap Construction (Time Complexity?)

Algorithm BottomUpHeap(L):
Input: An STL list L storing $n = 2^{h+1} - 1$ entries
Output: A heap T storing the entries of L.
if L.empty() **then**
  return an empty heap
$e \leftarrow L$.front()
L.pop_front()
Split L into two lists, L_1 and L_2, each of size $(n-1)/2$
$T_1 \leftarrow$ BottomUpHeap(L_1)
$T_2 \leftarrow$ BottomUpHeap(L_2)
Create binary tree T with root r storing e, left subtree T_1, and right subtree T_2
Perform a down-heap bubbling from the root r of T, if necessary
return T

Code Fragment 8.18: Recursive bottom-up heap construction.

Adaptable Priority Queues

• **A standby passenger** with a pessimistic attitude may become tired of waiting and decide to **leave ahead of the boarding time**, requesting to be removed from the waiting list. Thus, we would like to remove the entry associated with this passenger from the priority queue. Operation removeMin is not suitable for this purpose, since it only removes the entry with the lowest priority. Instead, we want **a new operation that removes an arbitrary entry**.

Another standby passenger finds her gold frequent-flyer card and shows it to the agent. Thus, her priority has to be **modified accordingly**. To achieve this change of priority, we would like to have a new operation that changes the information associated with a given entry. This might affect the entry's key value (such as frequent-flyer status) or not (such as correcting a misspelled name).

More Functions in Adaptable Priority Queues

insert(e):	Insert the element e into P and return a position referring to its entry.
remove(p):	Remove the entry referenced by p from P.
replace(p, e):	Replace with e the element associated with the entry referenced by p and return the position of the altered entry.

A list based implementation

```
template <typename E, typename C>
class AdaptPriorityQueue {                    // adaptable priority queue
protected:
  typedef std::list<E> ElementList;           // list of elements
public:
  // ...insert Position class definition here
public:
  int size() const;                           // number of elements
  bool empty() const;                         // is the queue empty?
  const E& min() const;                       // minimum element
  Position insert(const E& e);                // insert element
  void removeMin();                           // remove minimum
  void remove(const Position& p);             // remove at position p
  Position replace(const Position& p, const E& e); // replace at position p
private:
  ElementList L;                              // priority queue contents
  C isLess;                                   // less-than comparator
};
```

Code Fragment 8.19: The class definition for an adaptable priority queue.

```
class Position {                          // a position in the queue
private:
  typename ElementList::iterator q;       // a position in the list
public:
  const E& operator*() { return *q; }     // the element at this position
  friend class AdaptPriorityQueue;        // grant access
};
```

Code Fragment 8.20: The class representing a position in AdaptPriorityQueue.

```
template <typename E, typename C>                    // insert element
typename AdaptPriorityQueue<E,C>::Position
AdaptPriorityQueue<E,C>::insert(const E& e) {
  typename ElementList::iterator p = L.begin();
  while (p != L.end() && !isLess(e, *p)) ++p;        // find larger element
  L.insert(p, e);                                    // insert before p
  Position pos;  pos.q = --p;
  return pos;                                        // inserted position
}
```

Code Fragment 8.21: The function insert for class AdaptPriorityQueue.

```
template <typename E, typename C>                    // remove at position p
void AdaptPriorityQueue<E,C>::remove(const Position& p)
  { L.erase(p.q); }

template <typename E, typename C>                    // replace at position p
typename AdaptPriorityQueue<E,C>::Position
AdaptPriorityQueue<E,C>::replace(const Position& p, const E& e) {
  L.erase(p.q);                                      // remove the old entry
  return insert(e);                                  // insert replacement
}
```

Code Fragment 8.22: The functions remove and replace for AdaptPriorityQueue.

Reading Material

Sections 8.3.5 -- 8.4.1 of the textbook