Berner Fachhochschule - Technik und Informatik

Algorithms and Data Structures

Priority Queues and Heaps

Dr. Rolf Haenni

Fall 2008

Outline

Outline

Priority Queues

Heaps

Heap-Based Priority Queues

Bottom-Up Heap Construction

Priority Queue ADT

- A priority queue stores a collection of items
- An item is a pair (*key*, *element*)
- Characteristic operations:
  - → `insertItem(k,e)`: inserts an item with key k and element e
  - → `removeMin()`: removes the item with the smallest key and returns its element
  - → `minKey()`: returns the smallest key of an item (no removal)
  - → `minElement()`: returns the element of an item with smallest key (no removal)
- General operations:
  - → `size()`: returns the number of items
  - → `isEmpty()`: indicates whether the priority queue is empty
- Two distinct items in a priority queue can have the same key

Total Order

- Keys in a priority queue can be arbitrary objects on which a total order is defined
- Mathematically, a total order is a binary relation $\preceq$ defined on a set K which satisfies three properties:
  - → Totality: $x \preceq y$ or $y \preceq x$, for all $x, y \in K$
  - → Antisymmetry: $x \preceq y$ and $y \preceq x$ implies $x = y$, for all $x, y \in K$
  - → Transitivity: $x \preceq y$ and $y \preceq z$ implies $x \preceq z$, for all $x, y, z \in K$
- Totality implies Reflexivity: $x \preceq x$, for all $x \in K$
- Examples: $\leq$ or $\geq$ for $\mathbb{R}$, alphabetical or reverse alphabetical order for $\{A, \ldots, Z\}$, lexicographical order for $\{A, \ldots, Z\}^*$, etc.

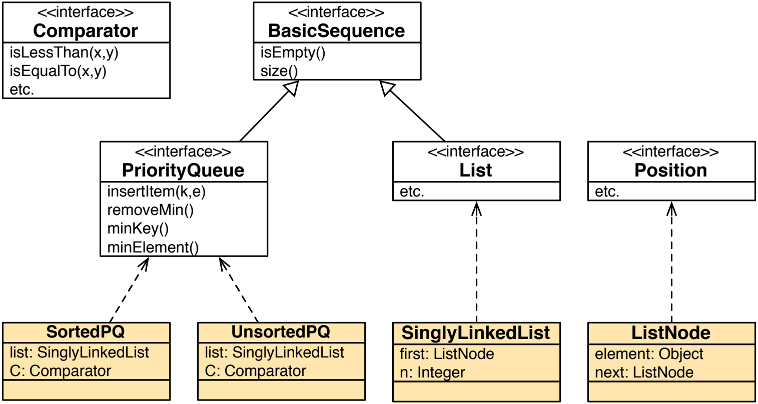
Comparator ADT

- A comparator encapsulates the action of comparing two keys according to a given total order relation
- A generic priority queue uses an auxiliary comparator (passed as a parameter to the constructor)
- When the priority queue needs to compare two keys, it uses its comparator
- Operations (all with Boolean return type):
  - → `isLessThan(x,y)`
  - → `isLessThanOrEqualTo(x,y)`
  - → `isEqualTo(x,y)`
  - → `isGreaterThan(x,y)`
  - → `isGreaterThanOrEqualTo(x,y)`
  - → `isComparable(x)`

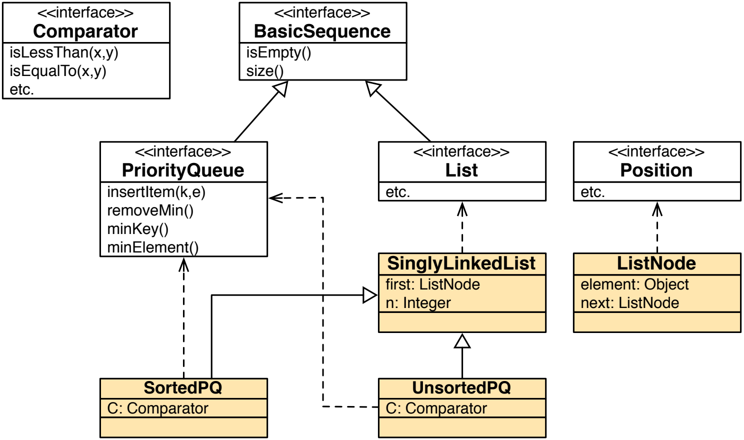
Sequence-Based Priority Queue

- There are two ways to implement a priority queue with a sequence (list or vector)
- Using an unsorted sequence
  - → `insertItem(k,e)` runs in $O(1)$ time, since we can insert the item at the beginning of the sequence
  - → `removeMin()`, `minKey()`, `minElement()` run in $O(n)$ time since we have to traverse the entire sequence to find the smallest key
- Using a sorted sequence
  - → `insertItem(k,e)` runs in $O(n)$ time, since we have to find the place where to insert the item
  - → `removeMin()`, `minKey()`, `minElement()` run in $O(1)$ time since the smallest key is at the beginning or end of the sequence

UML Diagram: Composition



UML Diagram: Inheritance



Sorting with a Priority Queue

- We can use a priority queue to sort a collection of comparable elements
  - → Insert the elements one by one with a series of `insertItem(e,e)` operations
  - → Remove the elements in sorted order with a series of `removeMin()` operations
- Depending on how the priority queue is implemented (sorted or unsorted), this leads to two well-known sorting algorithms
  - → Selection-Sort
  - → Insertion-Sort

PQ-Sort Algorithm

Solution in pseudo-code:

```
Algorithm PQ−Sort(S,C)
  P ← new priority queue with comparator C
  while not S.isEmpty() do
    e ← S.removeElement(S.first())
    P.insertItem(e,e)
  while not P.isEmpty() do
    e ← P.removeMin()
    S.insertLast(e)
  return S
```

Selection-Sort

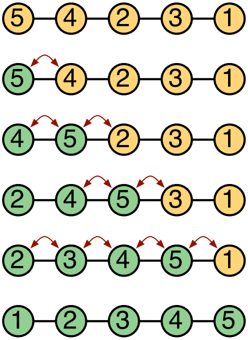
- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- The main task is to select the elements from the unsorted sequence in the right order
- Running time analysis:
  - → Inserting the elements into the priority queue with n `insertItem(e,e)` operations runs in $O(n)$ time
  - → Removing the elements in sorted order from the priority queue with n `removeMin()` operations takes time proportional to $n + \ldots + 2 + 1$, and thus runs in $O(n^2)$ time
- Selection-sort runs in $O(n^2)$ time

Insertion-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an sorted sequence
- The main task is to insert the elements at the right place
- Running time analysis:
  - → Inserting the elements into the priority queue with n `insertItem(e,e)` operations takes time proportional to $1 + 2 + \ldots + n$, and thus runs in $O(n^2)$ time
  - → Removing the elements in sorted order from the priority queue with n `removeMin()` operations runs in $O(n)$ time
- In general (worst case), insertion-sort runs in $O(n^2)$ time
- If the input sequence is in reverse order (best case), it runs in $O(n)$ time

In-Place Sorting

- Instead of using external data structures, we can implement insertion- and selection-sort in-place
- A portion of the input sequence itself serves as priority queue
- For in-place insertion-sort we
  - → keep the initial portion of the sequence sorted
  - → use `swapElements(p,q)` instead of modifying the sequence

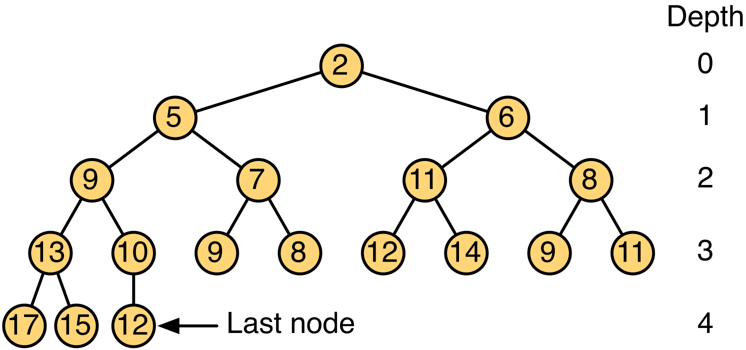


Outline

The Heap ADT

- A heap is a specialized binary tree storing keys at its nodes (positions) and satisfying the following properties:
  - → Heap-Order: $key(p) \succeq key(parent(p))$, for every node p other than the root
  - → Completeness: let h be the height of the tree
    1) there are 2^i nodes of depth i for $i = 0, \ldots, h-1$
    2) at depth h, nodes are filled up from the left
- The last node of a heap is the rightmost node of depth h
- Heap operations:
  - → `insertKey(k)`: inserts a key k into the heap
  - → `removeMin()`: removes smallest key and returns it
  - → `minKey()`: returns the smallest key (no removal)

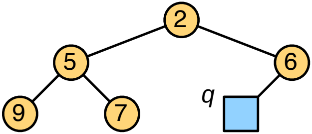
Heap Example

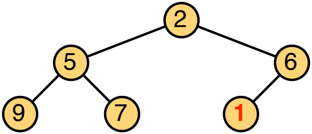


- A heap storing n keys has height $O(\log n)$

Insertion to a Heap

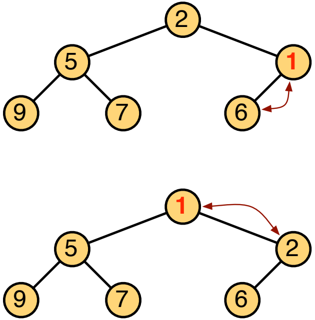
- The insertion of a key k consists of 3 steps:
  - → Add a new node q (the new last node)
  - → Store k at q
  - → Restore the heap order property (discussed next)





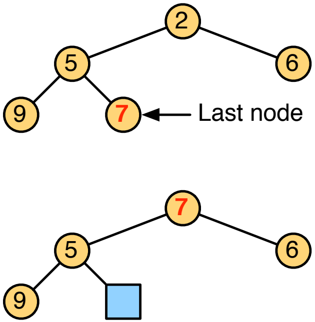
Upheap

- Algorithm `upheap` restores the heap-order property
  - → Swap k along an upward path from the insertion node
  - → Stop when k reaches the root or a node whose parent has a key smaller than or equal to k
  - → Since the height of the heap is $O(\log n)$, `upheap` runs in $O(\log n)$ time
- `insertKey(k)` runs in $O(\log n)$ time



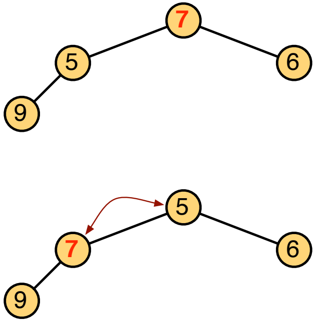
Removal from a Heap

- The removal algorithm consists of 4 steps:
  - → Return the key of the root
  - → Replace the root key with the key k of the last node
  - → Delete the last node
  - → Restore the heap order property (discussed next)



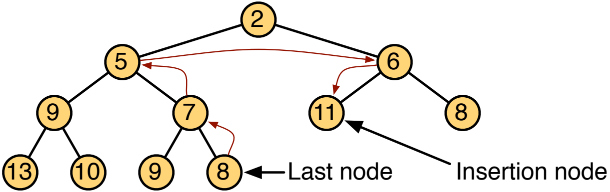
Downheap

- Algorithm `downheap` restores the heap-order property
  - → Swap k along an downward path from the root (always with the smaller key of its children)
  - → Stop when k reaches a leaf or a node whose children have keys greater than or equal to k
  - → Since the height of the heap is $O(\log n)$, downheap runs in $O(\log n)$ time
- `removeMin()` runs in $O(\log n)$ time



Finding the Insertion Node

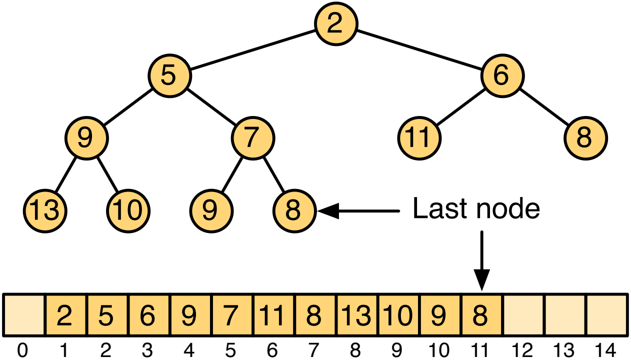
- Starting from the last node, the insertion node can be found by traversing a path of $O(\log n)$ nodes
  - → If the last node is a left child, return the parent node
  - → While the current node is a right child, go to the parent node
  - → If the current node is a left child, go to the right child
  - → While the current node is internal, go to the left child



Array-Based Heap Implementation

- We can represent a heap with n keys directly by means of an array of length $N > n$
  - → By-pass the binary tree ADT
  - → No explicit position ADT needed
- Idea similar to array-based implementation of binary trees
  - → The array cell at index 0 is unused
  - → The root of the heap is at index 1
  - → The left child of the node at index i is at index $2i$
  - → The right child of the node at index i is at index $2i + 1$
- The insertion operations corresponds to inserting at index $n + 1$ and thus runs in $O(1)$ time

Array-Based Heap: Example



Outline

Priority Queues

Heaps

Heap-Based Priority Queues

Bottom-Up Heap Construction

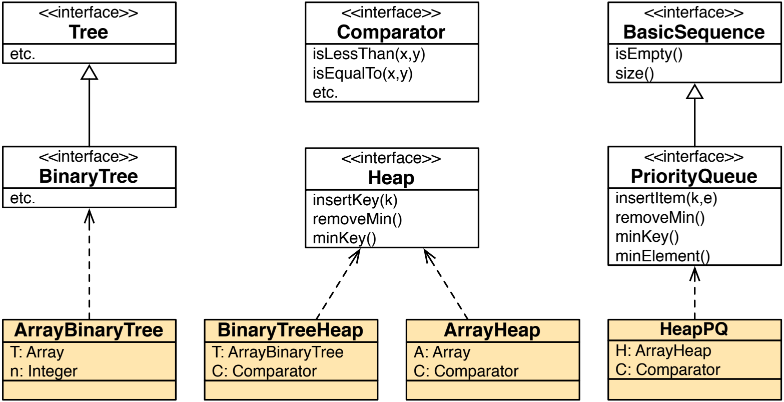
Heap-Based Priority Queues

- We can use a heap to implement a priority queue by storing a pair $item = (key, element)$ at each node of the heap
- Running times for different implementations

Operation	Unsorted Sequence	Sorted Sequence	Heap
size, isEmpty	1	1	1
minElement, minKey	n	1	1
insertItem	1	n	$\log n$
removeMin	n	1	$\log n$
PQ-Sort	n^2	n^2	$n \cdot \log n$

- In the long run, the heap-based implementation beats any sequence-based implementation

UML Diagram: Composition



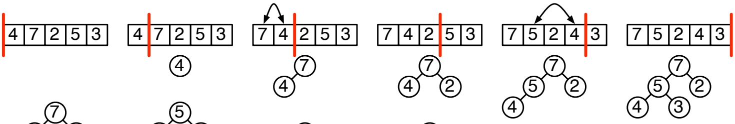
Heap-Sort

- Heap-sort is the variation of PQ-sort where the priority queue is implemented with a heap
- Running time analysis:
  - → Inserting the elements into the priority queue with n `insertItem(e,e)` operations runs in $O(n \log n)$ time
  - → Removing the elements in sorted order from the priority queue with n `removeMin()` operations runs in $O(n \log n)$ time
- In general (worst case), heap-sort runs in $O(n \log n)$ time
- For large n, heap-sort is much faster than quadratic sorting algorithms such as insertion-sort or selection-sort

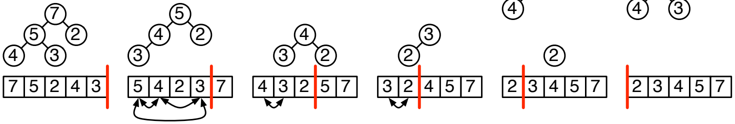
In-Place Heap-Sort

- To implement Heap-Sort *in-place*, use the front of the original array to implement the heap
  - → Step 1: Use the reverse comparator (e.g. $\succeq$ instead of $\preceq$) to build up the heap (by swapping elements)
  - → Step 2: Iteratively remove the maximum element from the heap and insert it in front of the sequence

Phase 1:



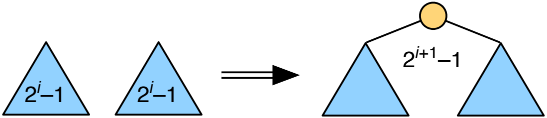
Phase 2:



Outline

Bottom-up Heap Construction

- We can construct a heap storing $n = 2^h - 1$ keys using a recursive bottom-up construction with $h - 1$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys
- Merging two heaps (and a new key k):
  - → Create a new heap with the root node storing k and with the two heaps as subtrees
  - → Perform downheap to restore the heap-order property

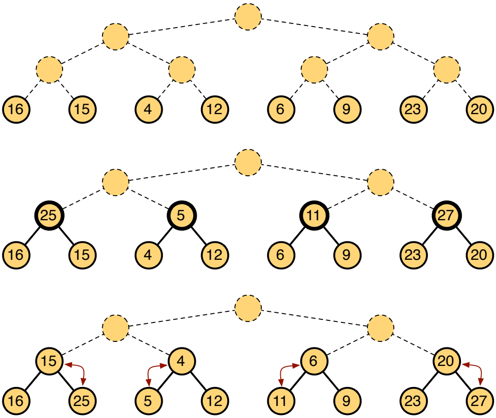


Recursive Algorithm in Pseudo-Code

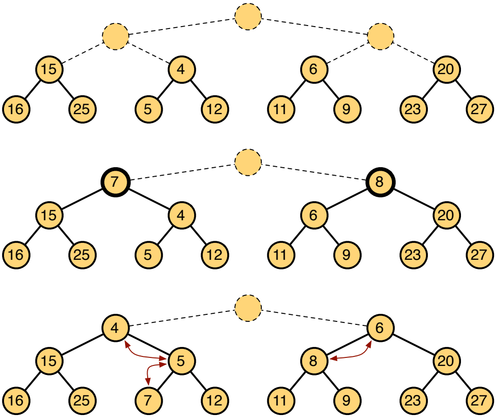
```
Algorithm bottomUpHeap(S) // sequence of keys of length 2^h−1
  if S.isEmpty() then
    return new heap
  else
    k ← S.elemAtRank(0)
    S.removeAtRank(0)
    H1 ←bottomUpHeap(firstHalf(S)) // 1st recursive call
    H2 ←bottomUpHeap(secondHalf(S)) // 2nd recursive call
    return mergeHeaps(k, H1, H2) // merge the results
```

In the general case, i.e. if the sequence is not of size $n = 2^h - 1$, we can adapt the splitting of the sequence accordingly

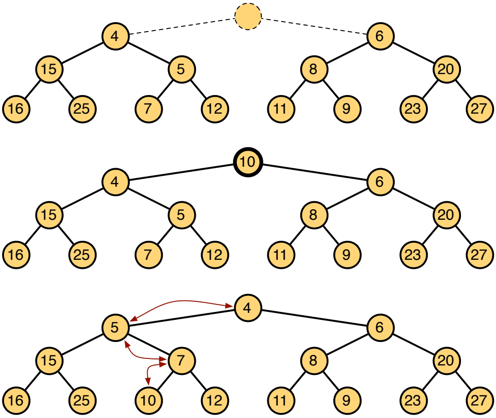
Example: Phase 1



Example: Phase 2



Example: Phase 3



Complexity Analysis

- The recursive `BottomUpHeap` algorithm decomposes a problem of size n into two problems of size $\frac{n-1}{2}$
  - → Rule D2 with $q = r = 2$ (see Topic 2, Page 31)
- The running time $f(n)$ of each recursive step is $O(\log n)$
  - → with an array-based sequence implementation, `firstHalf()` and `secondHalf()` run in $O(1)$ time
  - → the necessary downheap in `mergeHeaps` runs in $O(\log n)$ time
- Apply the second column of D2, i.e. bottom-up heap construction runs in $O(n)$
- Speeds up the heap construction (Phase 1) of the Heap-Sort algorithm, but not Phase 2