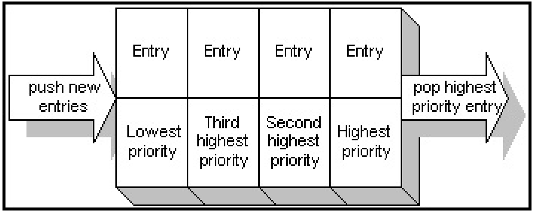
Computer Science 3A - CSC3A10

Lecture 7: Priority Queues, Heaps and Adaptable Priority Queues

Academy of Computer Science and Software Engineering
University of Johannesburg

Priority Queue ADT



Priority Queue Properties

- Priority Queue is a ADT for storing a collection of prioritized elements
- Supports arbitrary inserts
- Support removal of elements in order of priority
- Stores elements according to their priority and exposes no notion of position to the user
- Key:
  - Object assigned to an element which can be used to identify or weigh that element
  - Keys are not necessarily unique
  - Can be of any type (usually a type appropriate to a specific application)

Entries and Total Orders

Entries:

- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - **key()**: returns the key for this entry
  - **value()**: returns the value associated with this entry

Entries and Total Orders II

Total Order Relations:

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key
- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Priority Queue Methods

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - **insert(k, x)** - inserts an entry with key k and value x
  - **removeMin()** - removes and returns the entry with smallest key
- Additional methods:
  - min() returns, but does not remove, an entry with smallest key
  - size()
  - isEmpty()
- Applications include:
  - Standby flyers
  - Auctions
  - Stock market

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator
- The primary method of the Comparator ADT:
  - compare(x, y): Returns an integer i such that
  - $i < 0$ if $a < b$, $i = 0$ if $a = b$, and $i > 0$ if $a > b$;
  - an error occurs if a and b cannot be compared.

Priority Queue Sorting

We can use a priority queue to sort a set of comparable elements

- Insert the elements one by one with a series of insert operations
- Remove the elements in sorted order with a series of removeMin operations

The running time of this sorting method depends on the priority queue implementation

```
Algorithm PQ-Sort(S, P)
Input: sequence S,
priority queue P with
    comparator C for the
    elements of S
Output: sequence S sorted
      in increasing order
    according to C

while !S.isEmpty ()
  e = S.removeFirst ()
  P.insert (e, ∅)
while !P.isEmpty()
  e = P.removeMin().
      getkey()
  S.addLast(e)
```

Linked List-Based Priority Queue

Implementation with an unsorted list.

Performance

- insert takes O(1) time since we can insert the item at the beginning or end of the sequence
- removeMin and min take O(n) time since we have to traverse the entire sequence to find the smallest key

Implementation with a sorted list.

Performance

- insert takes O(n) time since we have to find the place where to insert the item
- removeMin and min take O(1) time, since the smallest key is at the beginning

Selection Sort

Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted list.

Running time of Selection-sort:

- Phase 1: Inserting the elements into the priority queue with n insert operations takes O(n) time
- Phase 2: Bottleneck, Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to
- $n + (n-1) + ... + 2 + 1$ $O(n + (n-1) + ... + 2 + 1) = n(n+1)/2$

Selection-sort runs in $O(n^2)$ time as phase 2 runs in O(n^2)

Selection-Sort II

		Sequence S	*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)	()

Insert-Sort

Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted list.

Running time of Insertion-sort:

- Phase 1: Insert into priority queue by scanning the list and inserting in the correct position. Inserting the elements into the priority queue with n insert operations takes time proportional to
- $1 + 2 + ... + n$
- $O(1 + 2 + ... + (n - 1) + n) = n(n + 1)/2$
- Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time

Insertion-sort runs in $O(n^2)$ time as phase 1 runs in $O(n^2)$

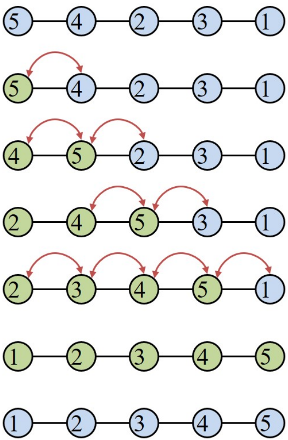
Insert-Sort II

	Sequence S	*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)
..	..	..
.	.	.
(g)	(2,3,4,5,7,8,9)	()

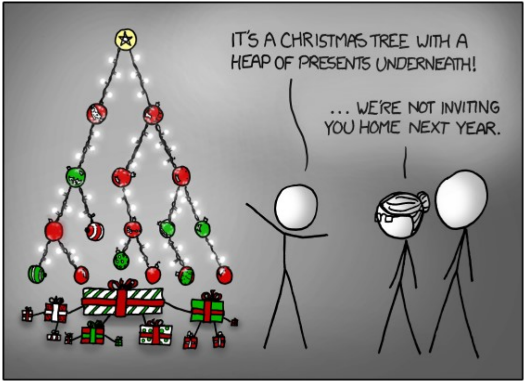
In-place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use **swaps** instead of modifying the sequence

In-place Insertion-Sort II



Heap ADT


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

Heap ADT Properties

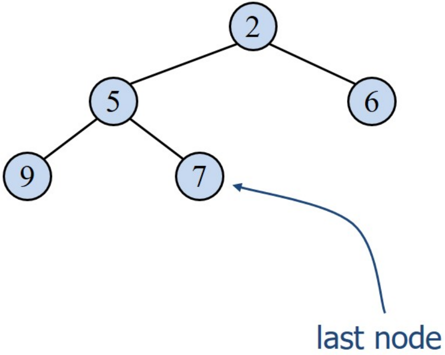
A heap is a binary tree storing keys at its nodes and satisfying the following properties:

- Heap-Order: for every internal node v other than the root
- $key(v) \geq key(parent(v))$
- Complete Binary Tree:
  - let h be the height of the heap
  - for $i = 0, ..., h - 1$, there are 2^i nodes of depth i

  at depth $h - 1$, the internal nodes are to the left of the external nodes

Heap ADT Properties II

The last node of a heap is the rightmost node of depth h

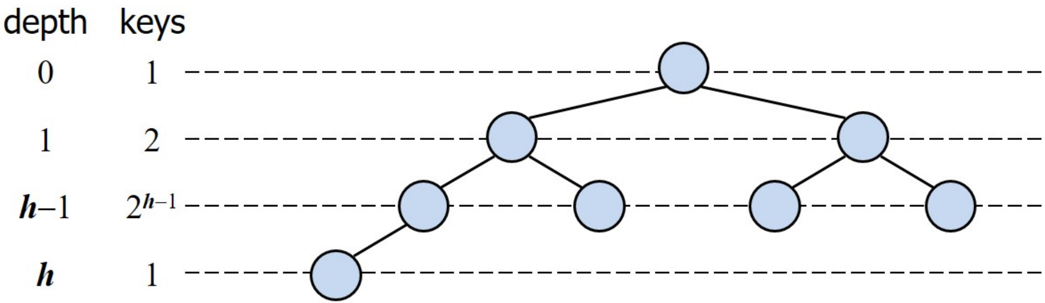


Height of a Heap

Theorem: A heap storing n keys has height $O(logn)$
Proof: (we apply the complete binary tree property)

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, ..., h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Thus, $n \geq 2^h, i.e., h \leq logn$

Height of a Heap II

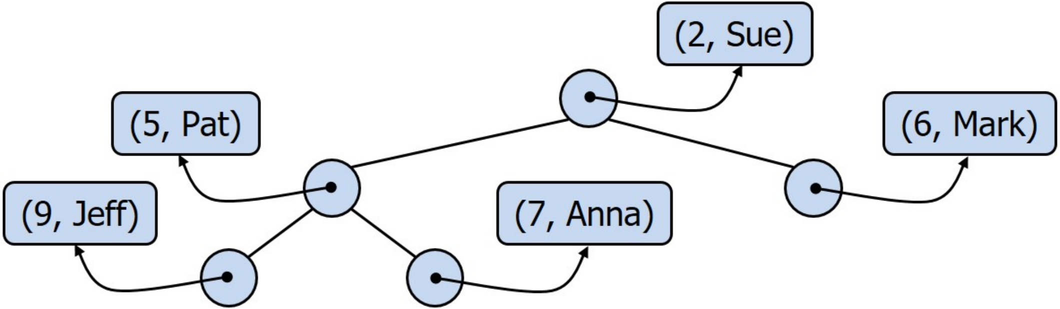


Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each node
- We keep track of the position of the last node

Heaps and Priority Queues II

For simplicity, we show only the keys in the images:



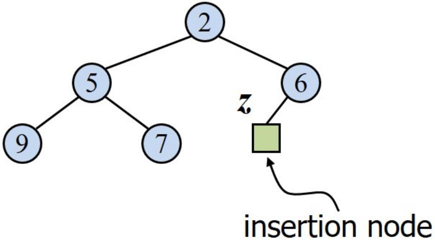
Insertion into a Heap

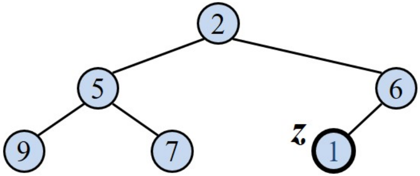
Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap.

The insertion algorithm consists of three steps:

1. Find the insertion node z (the new last node)
2. Store k at z
3. Restore the heap-order property (discussed next)

Insertion into a Heap II

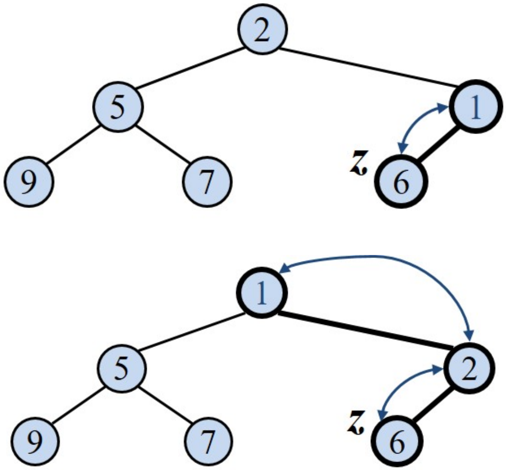




Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(logn)$, upheap runs in $O(logn)$ time

Upheap II



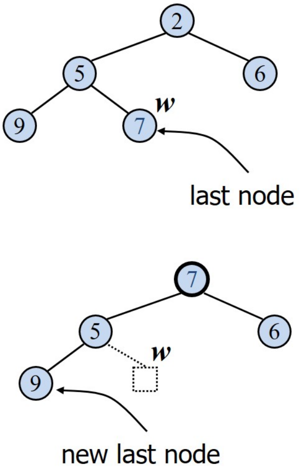
Removal from a Heap

Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap.

The removal algorithm consists of three steps

1. Replace the root key with the key of the last node w
2. Remove *w*
3. Restore the heap-order property (discussed next)

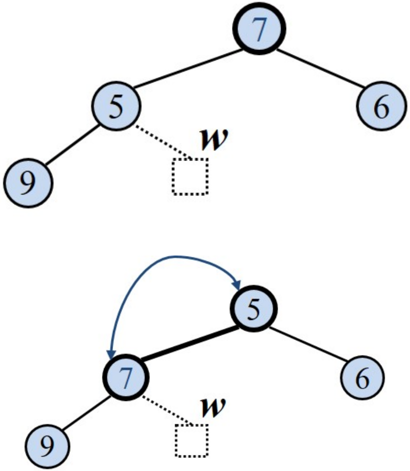
Removal from a Heap II



Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(logn)$, downheap runs in $O(logn)$ time

Downheap II

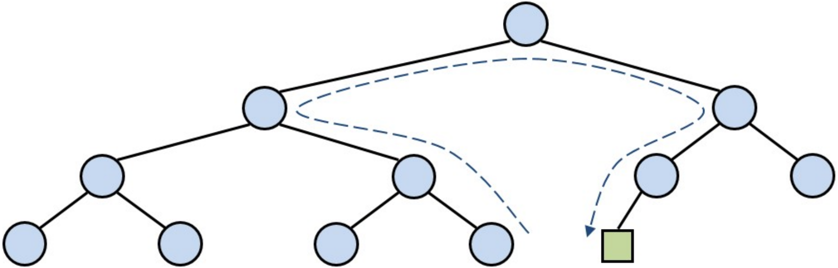


Updating the Last Node

The insertion node can be found by traversing a path of $O(logn)$ nodes

1. Go up until a left child or the root is reached
2. If a left child is reached, go to its right sibling
3. Go down left until a leaf is reached

Updating the Last Node II



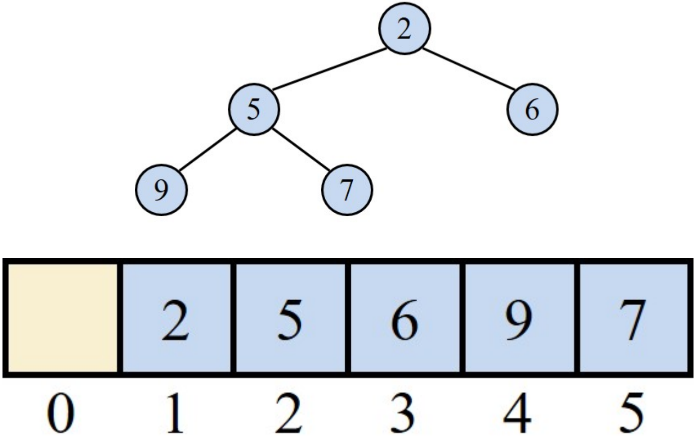
Heap-Sort

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods *insert* and *removeMin* take $O(logn)$ time
  - methods *size*, *isEmpty* and *min* take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(nlogn)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

ArrayList-Based Heap Implementation

- We can represent a heap with n keys by means of a vector of length n + 1
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at *rank* n
- Yields in-place heap-sort

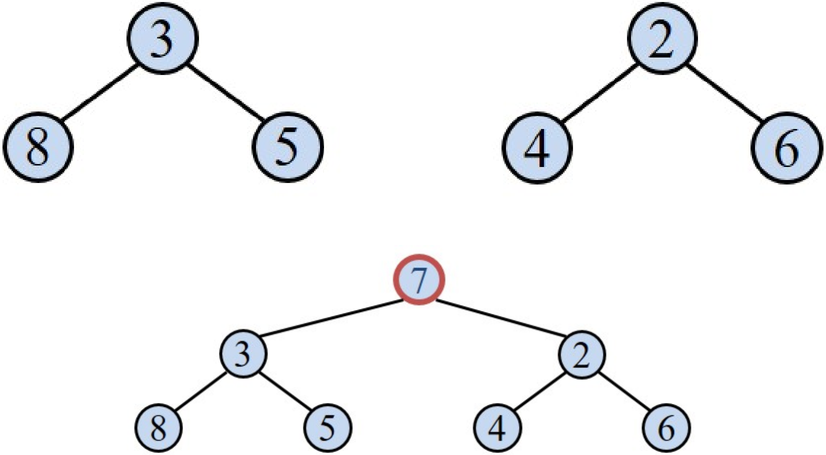
ArrayList-Based Heap Implementation



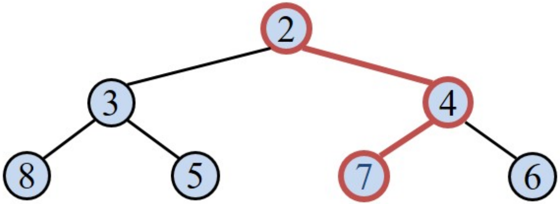
Merging Two Heaps

- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property

Merging Two Heaps II



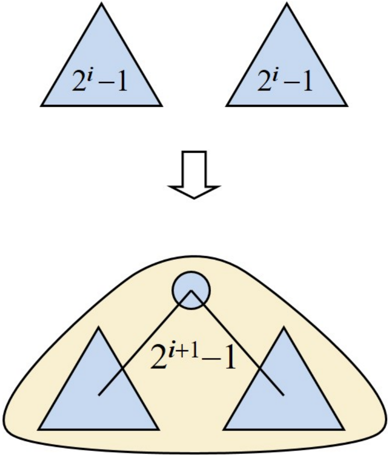
Merging Two Heaps III



Bottom-up Heap Contruction

- We can construct a heap storing n given keys in using a bottom-up construction with $log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

Bottom-up Heap Contruction II

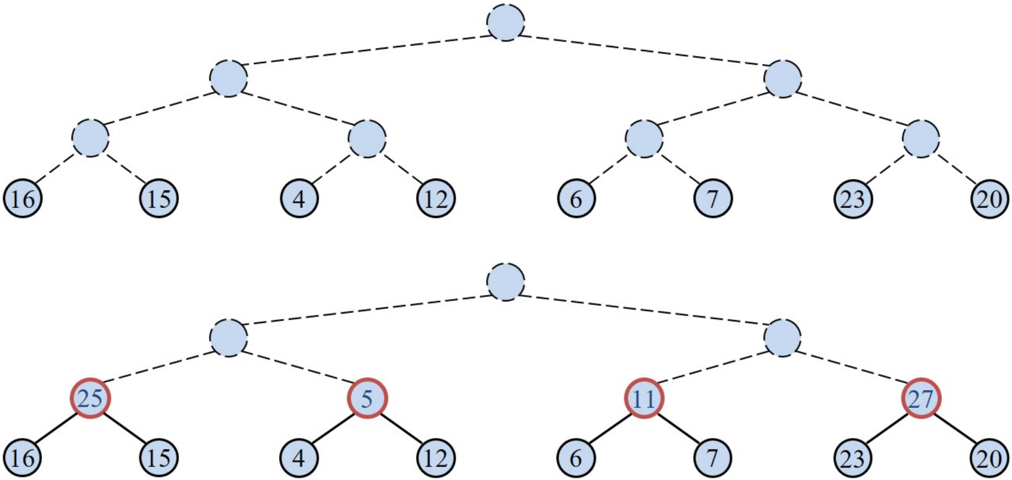


Bottom-up Heap Construction III

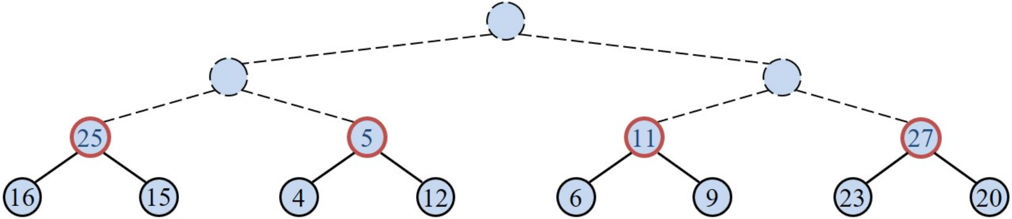
```
Algorithm BottomUpHeap(S)
Input A List L storing pow(2,i+1) −1 entries
Output A heap T storing the entries in L

If S.isEmpty () then
  return an empty heap
e = L.remove(L.first())
Split L into two lists , L1 and L2 , each size (n−1)/2
T1 = BottomUpHeap(L1)
T2 = BottomUpHeap(L2)
Create Binary Tree T with root r storing e, left subtree T1 and right
     subtree T2
Perform a down−heap from the root r of T, if necessary
return T
```

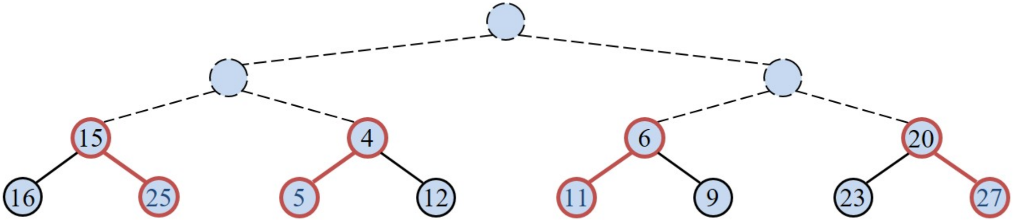
Bottom-up Algorithm

Bottom-up Heap Construction Example

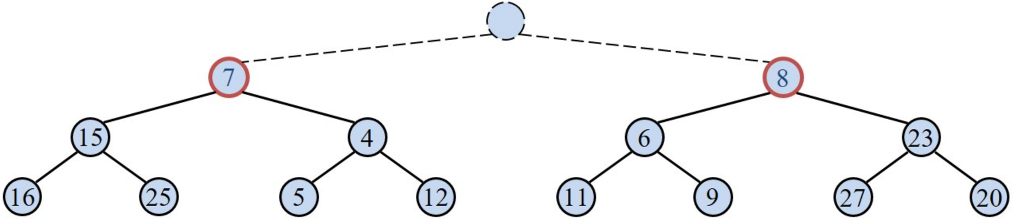


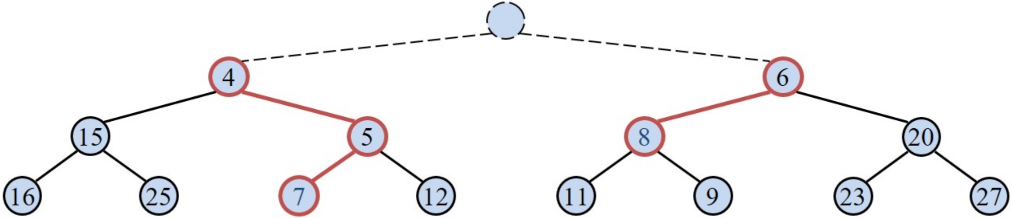
Bottom-up Heap Construction Example II



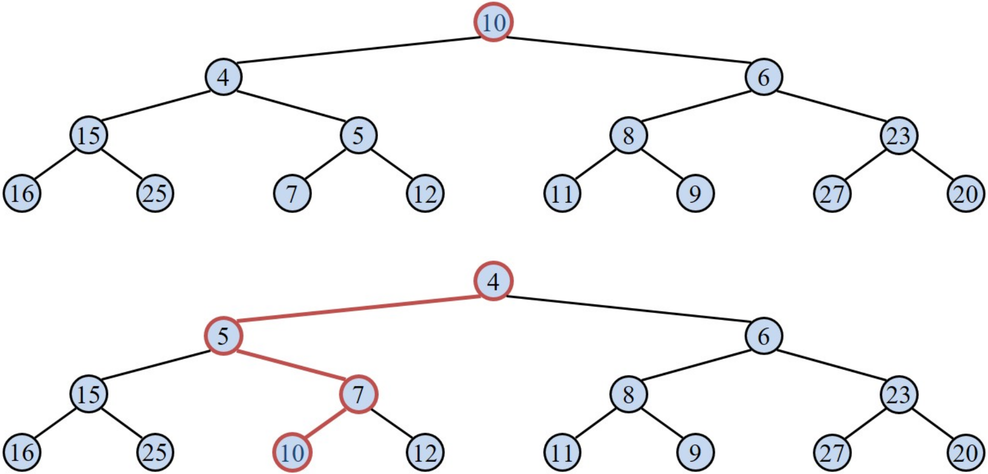


Bottom-up Heap Construction Example III





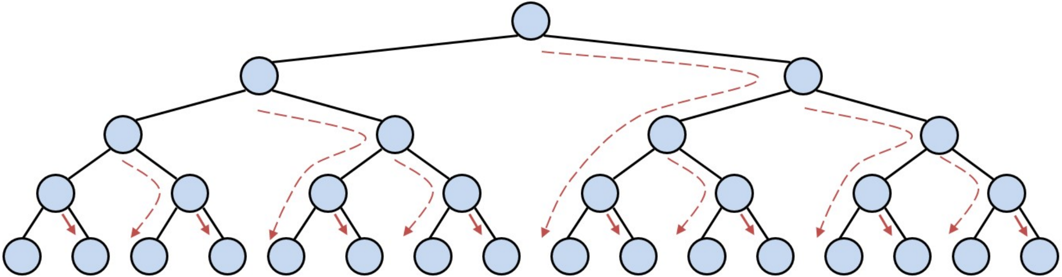
Bottom-up Heap Construction Example IV



Downheap Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort

Downheap Analysis II



Adaptable Priority Queue ADT

Adaptable Priority Queue Methods

- *remove(e)*: Remove from P and return entry e.
- *replaceKey(e,k)*: Replace with k and return the key of entry e of P; an error condition occurs if k is invalid (that is, k cannot be compared with other keys).
- *replaceValue(e,x)*: Replace with x and return the value of entry e of P.

Adaptable Priority Queue Methods II

Operation	*Output*	*P*
insert(5,*A*)	*e*1	(5,*A*)
insert(3,*B*)	*e*2	(3,*B*),(5,*A*)
insert(7,*C*)	*e*3	(3,*B*),(5,*A*),(7,*C*)
min()	*e*2	(3,*B*),(5,*A*),(7,*C*)
key(*e*2)	3	(3,*B*),(5,*A*),(7,*C*)
remove(*e*1)	*e*1	(3,*B*),(7,*C*)
replaceKey(*e*2,9)	3	(7,*C*),(9,*B*)
replaceValue(*e*3,*D*)	C	(7,*D*),(9,*B*)
remove(*e*2)	*e*2	(7,*D*)

Locating Entries and Location-Aware Entries

- In order to implement the operations *remove(k)*, *replaceKey(e)*, and *replaceValue(k)*, we need fast ways of locating an entry e in a priority queue.
- We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.
- A locator-aware entry identifies and tracks the location of its (key, value) object within a data structure

Main idea: Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier

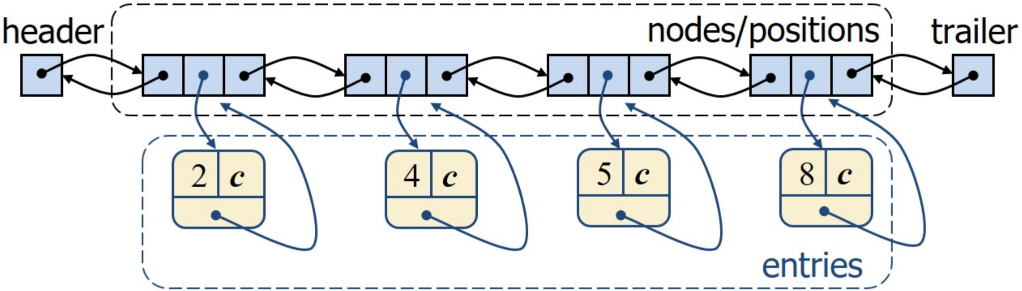
Location-Aware list Implementation

A location-aware list entry is an object storing:

- key
- value
- position (or rank) of the item in the list

In turn, the position (or array cell) stores the entry. Back pointers (or ranks) are updated during swaps

Location-Aware list Implementation II



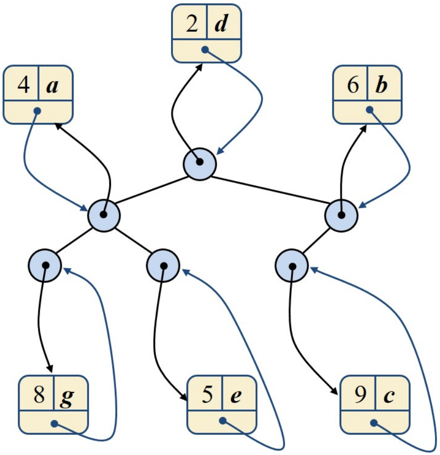
Location-Aware Heap Implementation

A location-aware heap entry is an object storing:

- key
- value
- position of the entry in the underlying heap

In turn, each heap position stores an entry. Back pointers are updated during entry swaps

Location-Aware Heap Implementation II



Performance

Using location-aware entries we can achieve the following running times (times better than those achievable without location-aware entries are in bold):

Method	Unsorted List	Sorted List	Heap
size, isEmpty	O(1)	O(1)	O(1)
insert	O(1)	O(n)	O(log n)
min	O(n)	O(1)	O(1)
removeMin	O(n)	O(1)	O(log n)
remove	**O(1)**	**O(1)**	**O(log n)**
replaceKey	**O(1)**	O(n)	**O(log n)**
replaceValue	**O(1)**	**O(1)**	**O(1)**

Exercises

Reinforcement exercises:

- R8.2
- R8.3
- R8.5 - R8.8
- R8.11 - R8.13
- R8.15 - R8.17
- R8.23

Creativity exercises:

- C8.3 - C8.5
- C8.12 - C8.14
- C8.16 - C8.17