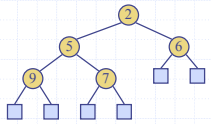
More Data Structures

Priority Queues, Comparators, Locators, Dictionaries

Priority Queues and Heaps



Priority Queue ADT (§ 2.4.1)

- A priority queue stores a collection of items
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - insertItem(k, o) inserts an item with key k and element o
  - removeMin() removes the item with smallest key and returns its element
- Additional methods
  - minKey(k, o) returns, but does not remove, the smallest key of an item
  - minElement() returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Total Order Relation

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct items 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: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Comparator ADT (§ 2.4.1)

- 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
- Methods of the Comparator ADT, all with Boolean return type
  - isLessThan(x, y)
  - isLessThanOrEqualTo(x,y)
  - isEqualTo(x,y)
  - isGreaterThan(x, y)
  - isGreaterThanOrEqualTo(x,y)
  - isComparable(x)

Sorting with a Priority Queue (§ 2.4.1)

- We can use a priority queue to sort a set 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
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
Input sequence *S*, comparator *C* for the elements of *S*
Output sequence *S* sorted in increasing order according to *C*
P ← priority queue with comparator *C*
while ¬*S.isEmpty* ()
e ← *S.remove* (*S. first* ())
P.insertItem(*e*, *e*)
while ¬*P.isEmpty*()
e ← *P.removeMin*()
S.insertLast(*e*)

List-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

Selection-Sort

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

- Running time of Selection-sort:
  - Inserting the elements into the priority queue with n insertItem operations takes $O(n)$ time
  - Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

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

Insertion-Sort

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

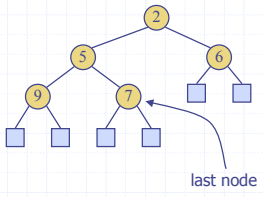
- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with n insertItem operations takes time proportional to

$$1 + 2 + \ldots + n$$

  - 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

What is a heap (§2.4.3)

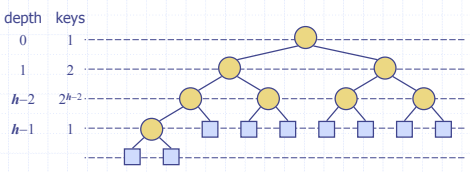
- A heap is a (proper) binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
    - at depth $h-1$, the internal nodes are to the left of the external nodes
- The last node of a heap is the rightmost internal node of depth $h-1$



Height of a Heap (§2.4.3)

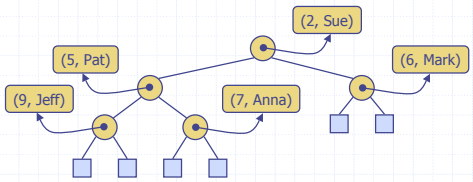
- Theorem: A heap storing n keys has height $O(\log n)$

  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 $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



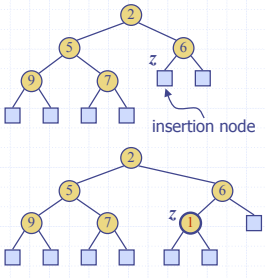
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures



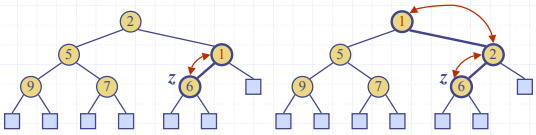
Insertion into a Heap (§2.4.3)

- 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
  - Find the insertion node z (the new last node)
  - Store k at z and expand z into an internal node
  - Restore the heap-order property (discussed next)



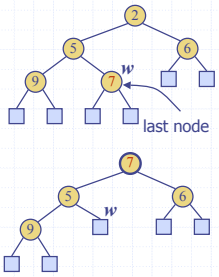
Upheap

- 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 $O(\log n)$, upheap runs in $O(\log n)$ time



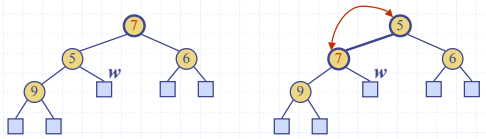
Removal from a Heap (§2.4.3)

- 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
  - Replace the root key with the key of the last node w
  - Compress w and its children into a leaf
  - Restore the heap-order property (discussed next)



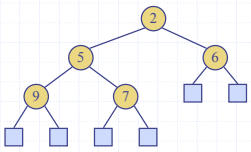
Downheap

- 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
- Upheap 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 $O(\log n)$, downheap runs in $O(\log n)$ time



Vector-based Heap Implementation (§2.4.3)

- 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 leaves are not represented
- The cell of at rank 0 is not used
- Last node at rank n
- Operation insertItem corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at rank n
- Yields in-place heap-sort



	2	5	6	9	7
0	1	2	3	4	5

PQ using Vector-based Heap

- InsertItem(k,o)
  - Insert pair (k,o) at rank n+1, followed by Upheap(n+1) call to restore heap-order property
- UpHeap(k)
  - // swaps item at rank k upward into correct position
  - If k=1 then return; // already bubbled up to root.
  - If rank k item is smaller than its parents
    - Swap rank k item with its parent (rank k/2)
    - Upheap(k/2).
- O(log n) time for insertItem.

PQ using Vector-based Heap

- removeMin()
  - Swap rank 1 item with rank n item;
  - remove rank n item; store it in *temp*.
  - call Downheap(1) to restore heap-order property.
  - return *temp*.
- Downheap(k)
  - // Swaps item at rank k downard into correct position
  - If 2k> n then return; // item has no children
  - If rank k item is not smaller than its children
    - Let j be rank of child of rank k item with smaller key (j= 2k or 2k+1).
    - Swap rank k item with rank j item
    - Downheap(j)
- O(log n) time to removeMin.

Heap-Sort (§2.4.4)

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

PQ Implementations

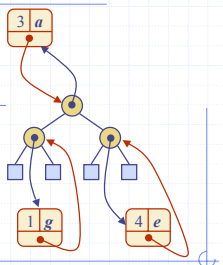
Implementation:	insertItem	removeMin	PQ-Sort cost
sorted list	O(n)	O(1)	$O(n^2)$
unsorted list	O(1)	O(n)	$O(n^2)$
(Vector-based) heap	O(log n)	O(log n)	O(n log n)

- Space is O(n) for all implementations.

In-place sorting

- Sort in-place – only use O(1) extra space.
- For array-based implementations, PQ-Sort takes 2n space.
- Implement in-place sorting by having priority queue use input array to store values.
- For heap-sort:
  - Use reverse comparator (so we remove maximum)
  - Phase I: As items are inserted, heap expands from left to right.
  - Phase II: As items removed, they are placed from right to left; heap contracts from right to left.

Locators



Outline and Reading

- Locators (§2.4.4)
- Locator-based methods (§2.4.4)
- Implementation
- Positions vs. Locators

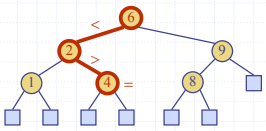
Locators

- A locators identifies and tracks a (key, element) item within a data structure
- A locator sticks with a specific item, even if that element changes its position in the data structure
- Intuitive notion:
  - claim check
  - reservation number
- Methods of the locator ADT:
  - key(): returns the key of the item associated with the locator
  - element(): returns the element of the item associated with the locator
- Application example:
  - Orders to purchase and sell a given stock are stored in two priority queues (sell orders and buy orders)
    - the key of an order is the price
    - the element is the number of shares
  - When an order is placed, a locator to it is returned
  - Given a locator, an order can be canceled or modified

Locator-based Methods

- Locator-based priority queue methods:
  - insert(k, o): inserts the item (k, o) and returns a locator for it
  - min(): returns the locator of an item with smallest key
  - remove(l): remove the item with locator l
  - replaceKey(l, k): replaces the key of the item with locator l
  - replaceElement(l, o): replaces with o the element of the item with locator l
  - locators(): returns an iterator over the locators of the items in the priority queue

Dictionaries



Outline and Reading

- Dictionary ADT (§2.5.1)
- Log file (§2.5.1)
- Binary search (§3.1.1)
- Lookup table (§3.1.1)
- Binary search tree (§3.1.2)
  - Search (§3.1.3)
  - Insertion (§3.1.4)
  - Deletion (§3.1.5)
  - Performance (§3.1.6)

Dictionary ADT

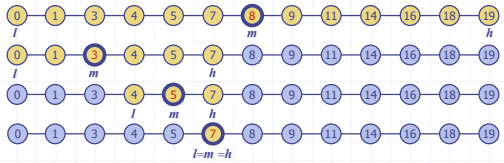
- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)
- Dictionary ADT methods:
  - findElement(k): if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
  - size(), isEmpty()
  - keys(), Elements()

Log File

- A log file is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked lists or a circular array), in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - findElement and removeElement take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Binary Search

- Binary search performs operation findElement(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - At each recursive call, ask if number is higher, lower, or equal to midpoint.
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
- Example: findElement(7)

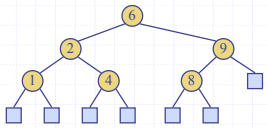


Lookup Table

- A lookup table is a dictionary implemented by means of a sorted sequence
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - findElement takes $O(\log n)$ time, using binary search
  - insertItem takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - removeElement take $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- The lookup table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search Tree

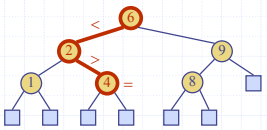
- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items
- An inorder traversal of a binary search trees visits the keys in increasing order



Search

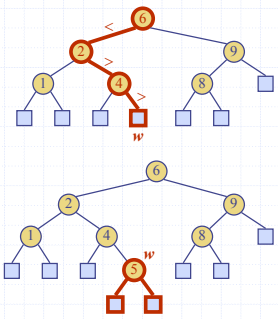
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY
- Example: findElement(4)

```
Algorithm findElement(k, v)
  if T.isExternal (v)
    return NO_SUCH_KEY
  if k < key(v)
    return findElement(k, T.leftChild(v))
  else if k = key(v)
    return element(v)
  else { k > key(v) }
    return findElement(k, T.rightChild(v))
```



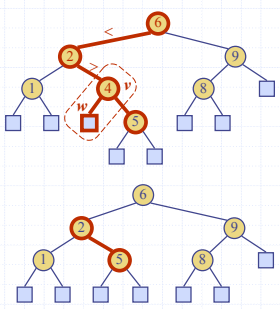
Insertion

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



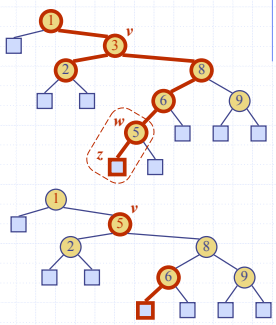
Deletion

- To perform operation removeElement(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeAboveExternal(w)
- Example: remove 4



Deletion (cont.)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeAboveExternal(z)
- Example: remove 3



Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods findElement , insertItem and removeElement take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case