Data Structures and Algorithms Design

A.Baskar

BITS Pilani, K. K. Birla Goa Campus

04 October 2015

Recap

- Sorting problem
- Insertion sort, Selection sort
- Priority Queue ADT

Outline

- Heaps
- Heap-sort
- Dictionary ADT

Sorting Problem

- Input : A sequence of n numbers $< a_1, a_2, \ldots, a_n >$
- Output : A permutation (reordering) $< a'_1, a'_2, \ldots, a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Sorting Problem

- Input : A sequence of n numbers $< a_1, a_2, \dots, a_n >$
- Output : A permutation (reordering) $< a'_1, a'_2, \dots, a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- Solutions : Many!
- First Solution : Selection Sort

Priority Queue ADT

- In selection sort and insertion sort we are using only a few methods
- removeMin, removeFirst, insert

Priority Queue ADT

- In selection sort and insertion sort we are using only a few methods
- removeMin, removeFirst, insert
- In selection sort we use removeMin and insert
- In insertion sort we use removeFirst and insert

Priority Queue ADT

- In selection sort and insertion sort we are using only a few methods
- removeMin, removeFirst, insert
- In selection sort we use removeMin and insert
- In insertion sort we use removeFirst and insert
- Priority queue supports these methods

Sorting using Priority Queue ADT

- We have collection C of n items and we want to sort them.
- First insert each into the priority queue Q using the insert method
- Now remove the minimum element from P and add into C until there are no elements left in Q.

Sorting using Priority Queue ADT

- We have collection C of n items and we want to sort them.
- First insert each into the priority queue Q using the insert method
- Now remove the minimum element from P and add into C until there are no elements left in Q.
- n insert operations and n removeMin operations are used.

Priority Queue Implementation: Unsorted sequence

- Using unsorted sequence to implement priority queue
- insert method takes constant time
- removeFirst and removeMin take O(n) time

Priority Queue Implementation: Unsorted sequence

- Using unsorted sequence to implement priority queue
- insert method takes constant time
- removeFirst and removeMin take O(n) time
- Selection sort can be seen as Priority Queue sort as follows
- From a given collection C insert element into the Priority Queue Q
- Use removeMin on Q and store it in C

Priority Queue Implementation: Sorted sequence

- Using sorted sequence to implement priority queue
- removeMin method takes constant time
- insert method takes O(n) time

Priority Queue Implementation: Sorted sequence

- Using sorted sequence to implement priority queue
- removeMin method takes constant time
- insert method takes O(n) time
- Insertion sort can be seen as Priority Queue sort as follows
- From a given collection C insert element into the Priority Queue Q
- Use removeMin on Q and store it in C

Priority Queue Implementation: Heap

- Both the above implementations use linear data structure
- Can we use some nonlinear data structure to implement the priority queue in an efficient way?

Priority Queue Implementation: Heap

- Both the above implementations use linear data structure
- Can we use some nonlinear data structure to implement the priority queue in an efficient way?
- In heap implementation insert and removeMin methods take O(logn) time
- From a given collection C insert element into the Priority Queue Q
- Use removeMin on Q and store it in C

Priority Queue Implementation: Heap

- Both the above implementations use linear data structure
- Can we use some nonlinear data structure to implement the priority queue in an efficient way?
- In heap implementation insert and removeMin methods take O(logn) time
- From a given collection C insert element into the Priority Queue Q
- Use removeMin on Q and store it in C
- This is known as heap sort and it takes O(nlogn) time

Heaps

- Storing elements and keys in internal nodes of binary tree
- External nodes will not have any element
- Heap-order property and complete binary tree property

Heap-order property

- Every node v other than the root, the key stored at v is greater than or equal to the key stored at v's parent.

Heap-order property

- Every node v other than the root, the key stored at v is greater than or equal to the key stored at v's parent.
- Keys on path from root node to an external node are in nondecreasing order
- The root node will have the minimum key

Complete binary tree

- A binary tree is a complete binary tree if in every level, except possibly the deepest, is completely filled. At depth n, the height of the tree, all nodes must be as far left as possible.

Complete binary tree

- A binary tree is a complete binary tree if in every level, except possibly the deepest, is completely filled. At depth n, the height of the tree, all nodes must be as far left as possible.
- There is a special node called last node
- A heap storing n keys has height $\lceil log(n+1) \rceil$

Array representation of a heap

- We can use array to represent heap and index of last node is equal to n
- If there are n keys to be stored, there will be $2n+1$ nodes in the tree
- But it is not necessary to store all of them in the array representation

Insertion in a heap

- If we want to insert a key k in the heap, first we have to identify the correct external node z.
- Then we perform an expandExternal(z) operation: replaces z with an internal node (which has two external nodes)
- Then insert e at the newly created internal node.

Insertion in a heap

- If we want to insert a key k in the heap, first we have to identify the correct external node z.
- Then we perform an expandExternal(z) operation: replaces z with an internal node (which has two external nodes)
- Then insert e at the newly created internal node.
- It might violate the heap-order property
- Up-heap bubbling to resolve this issue
- Insert method takes O(logn) time

removeMin in a heap

- First copy the key in the last node to root node
- Now change the last node as external node
- Reassign the last node

removeMin in a heap

- First copy the key in the last node to root node
- Now change the last node as external node
- Reassign the last node
- It might violate the heap-order property
- Down-heap bubbling to resolve this issue
- removeMin method takes O(logn) time

Heap-sort

- First we have to insert n items and it will take $O(nlogn)$ time
- Then we have to removeMin n times and it will take $O(nlogn)$ time
- So overall running time for heap-sort is $O(nlogn)$

Dictionary ADT

- Store items (k, e)
- search(k), insert(k,e), remove(k)
- size(), isEmpty()

A naive implementation: Log file

- Store items in an array,lists
- insert takes O(1)
- search and delete take O(n)

A naive implementation: Log file

- Store items in an array,lists
- insert takes O(1)
- search and delete take O(n)
- Space requirement is $\Theta(n)$

Direct Address Table

- Uses an array to store items
- item (k,e) will be stored in the cell A[k]
- Search takes O(1) time
- insert and delete take O(1) time

Direct Address Table

- Uses an array to store items
- item (k,e) will be stored in the cell A[k]
- Search takes O(1) time
- insert and delete take O(1) time
- Drawback: Space requirement

Ordered Dictionary ADT

- Store items (k, e)
- search(k), insert(k,e), remove(k)
- size(), isEmpty()

Ordered Dictionary ADT

- Store items (k, e)
- search(k), insert(k,e), remove(k)
- size(), isEmpty()
- successor(k), predecessor(k)
- maximum(k), minimum(k)

Look-up table

- Store items in an array but with ordering
- Size of look-up table is $\Theta(n)$

Look-up table

- Store items in an array but with ordering
- Size of look-up table is $\Theta(n)$
- Use the order to find an element efficiently
- Search takes O(logn) time
- insert and delete take O(n) time

Binary Search Tree

A Binary Search Tree (BST) is a binary tree with the following properties:

- The key of a node is always greater than the keys of the nodes in its left subtree
- The key of a node is always smaller than the keys of the nodes in its right subtree