Ch 1. Analysis of Algorithms

• Definition of O, Θ, Ω . I won’t ask you about little-oh, little-omega.

Big-Oh

Find the worst-case number of primitive operations executed as a function of the input size

Omega

Best-case running time

Theta

Average running time

• Show that f(n) is O(g(n))

O(g(n)) = { f(n) : ∃ positive constants c and n0 , such that ∀n ≥ n0 we have 0 ≤ f(n) ≤ cg(n) }

• Express and justify the runtime of an algorithm in Big-Oh notation

• Amortized analysis

Ch 2. Elementary Data Structures & Ch. 3 Search Trees How each data structure is implemented, any characteristic properties or definitions of the data structure, what operations can be performed on it, and understand the complexity of those operations. Data structures include:

• Stacks (including resizing and amortized analysis for stack operations)

Array based

Size can be incremented by a constant when full OR doubled

• Queues

Link-List based

• Linked List, Vector, Sequence

Operation Array List

size, isEmpty O(1) O(1)

atRank, rankOf, elemAtRank O(1) O(n)

first, last, before, after O(1) O(1)

replaceElement, swapElements O(1) O(1)

replaceAtRank O(1) O(n)

insertAtRank, removeAtRank O(n) O(n)

insertFirst, insertLast O(1) O(1)

insertAfter, insertBefore O(n) O(1)

remove (at given position) O(n) O(1)

• Trees

Properties of Trees:

n number of nodes

e number of external nodes

i number of internal nodes

h height (max depth)

e = i + 1

n = 2e - 1

h <= i

h <= (n - 1)/2

e <= 2h

h >= log2 e

h >= log2 (n + 1) – 1

Traversals:

(preorder) + ´ 2 – 5 1 ´ 3 2 == visit, left, right

(inorder) 2 ´ 5 – 1 + 3 ´ 2==left, visit, right

(postorder) 2 5 1 – ´ 3 2 ´ +==left, right, visit

• Binary Trees

Find the node with key k

Strategy

start at root r

if k = key(r), return r

if k < key(r), continue in left subtree

if k > key(r), continue in right subtree

Runtime is O(h), where h is the height of the tree

Algorithm BinarySearch(S, k, low, high):

if low > high then return NO\_SUCH\_KEY

mid ← ⌊(low + high) / 2⌋

if key(mid) = k then return elem(mid)

if key(mid) < k then return BinarySearch(S, k, mid + 1, high)

if key(mid) > k then return BinarySearch(S, k, low, mid -1)

Can be stored in an array:

Nodes are stored in an array

rank(root) = 1

If rank(node) = i,

then rank(leftChild) = 2\*i

rank(rightChild) = 2\*i + 1

• Priority Queues

Stores a collection of (key, element) pairs

Implementation with an unsorted sequence

Store the items of the priority queue in a list-based sequence, in arbitrary order

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 sequence

Store the items of the priority queue in a sequence, sorted by key

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

• Heaps (including definition, height)

A heap is a 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) >= key(parent(v))

Complete Binary Tree: let h be the height of the heap

for i = 0, … , h - 2, there are 2i internal nodes of depth i

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

A heap storing n keys has height O(log n)

• Dictionaries (hash tables, hash functions, universal hashing, how to avoid collisions, collision handling strategies, performance in relation to load factor)

Models a searchable collection of key-element items called entries

A hash table for a given key type consists of

Array (called table) of size N

Hash function h

A hash function h maps keys of a given type to integers in a fixed interval [0, N - 1]

Ex: h(x) = x mod N is a hash function for integer keys

The integer h(x) is called the hash value of key x

Recall that a good hash function guarantees the probability that two different keys have the same hash is 1/N.

A family of hash functions is universal if for any 0 ≤ j,k ≤ M-1, Pr( h(j)=h(k) ) ≤ 1/N

Collisions occur when different elements are mapped to the same cell

Chaining

each cell in the table points to a linked list of elements that map there

simple, but requires additional memory outside the table

Open Addressing

the colliding item is placed in a different cell of the table

no additional memory, but complicates searching/removing

common types: linear probing, quadratic probing, double hashing

In the worst case, searches, insertions and removals on a hash table take O(n) time

occurs when all inserted keys collide

The load factor a = n/N affects the performance of a hash table

Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is 1 / (1 - a)

The expected number of probes for an insertion with chaining is O(1 + a)

The expected running time of all the dictionary ADT operations in a hash table is O(1)

In practice, hashing is very fast provided the load factor is not close to 100%

• Arbitrary binary search trees (including definition, height)

Height depends on the order nodes were inserted

A binary search tree is a binary tree where each internal node stores a (key, element)-pair, and

each element in the left subtree is smaller than the root

each element in the right subtree is larger than the root

the left and right subtrees are binary search trees

An inorder traversal visits items in ascending order

Unordered Ordered

Log file Hash table Lookup table Binary Search Tree Balanced Trees

size, isEmpty O(1) O(1) O(1) O(1) O(1)

keys, elements O(n) O(n) O(n) O(n) O(n)

findElement O(n) O(n)\*\* O(logn) O(h) O(logn)

insertItem O(1) O(n)\*\* O(n) O(h) O(logn)

removeElement O(n) O(n)\*\* O(n) O(h) O(logn)

closestKey closestElem O(n) O(n) O(logn) O(h) O(logn)

\*\* Expected running time is O(1)

• Red-black trees (including definition, color properties, height). I won’t ask you to demonstrate the removal of an item from a red-black tree.

A binary search tree with nodes colored red and black in a way that satisfies the following color properties:

1. Root property: the root is black.

2. External property: every leaf is black.

3. Internal property: the children of a red node are black.

4. Depth property: all leaves have the same black depth.

A red-black tree storing n items has height O(log n)

Algorithms include:

• Execution of any data structure operation (e.g., insert into a red-black tree, remove an item from a binary search tree, find an item in a hash table using linear probing or double hashing, etc.)

insertion into Red/Black Tree:

Use insertion algorithm for binary search trees and color red the newly inserted node z, unless it’s the root.

we preserve the root, external, and depth properties

if the parent v of z is black, we also preserve the internal property and we are done

if the parent v of z is red, we have a double red (a violation of the internal property), which requires a reorganization of the tree

• Tree traversals (preorder, postorder, inorder, Euler tour)

Generic traversal of a binary tree

Includes preorder, postorder, and inorder traversals as special cases

Walk around the tree and visit each node three times:

– on the left (preorder) + ´ 2 – 5 1 ´ 3 2

– from below (inorder) 2 ´ 5 – 1 + 3 ´ 2

– on the right (postorder) 2 5 1 – ´ 3 2 ´ +

• Insert into a binary tree

start at root r

if k < key(r), continue in left subtree

if k > key(r), continue in right subtree

Remove from a binary tree

Case 1(a): n has two children which are external nodes

remove and replace with leaf

Case 1: n has a child which is an internal node

remove and replace with child

Case 2: n has two children which are internal nodes

Find the first internal node m that follows n in an inorder traversal

Replace n with m

• Binary search on an array

Can be stored in an array:

Nodes are stored in an array

rank(root) = 1

If rank(node) = i,

then rank(leftChild) = 2\*i

rank(rightChild) = 2\*i + 1

• Selection-sort, insertion-sort, heap-sort

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:

1. Inserting the elements into the priority queue with n insertItem operations takes O(n) time

2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to 1 + 2 + …+ n

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:

1. Inserting the elements into the priority queue with n insertItem operations takes time proportional to 1 + 2 + …+ n

2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes O(n) time

Runs in O(n^2) time

Heap-Sort

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

much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

• Bottom-up heap construction

If all keys are known in advance, we can build a heap recursively

For simplicity, assume number of keys n = 2h – 1 so the heap is a complete binary tree, so each depth i = 0, … , h - 2 contains 2i containing internal nodes

Given n keys, build heap using a bottom-up construction with log n phases

In phase i, pairs of heaps with 2i -1 keys are merged into heaps with 2i+1-1 keys

Ch. 4 Sorting and Selection Definitions of stable sort, in-place algorithm, divide and conquer paradigm, and lexicographic order. Lower bound on comparison-based sorting. A comparison of sorting algorithms based on time-complexity, if it is stable, and if it is in-place. Data structures include:

• Set (including operations using generic-merge). Algorithms include:

Generic Merging

Algorithm genericMerge(A, B)

S <- empty sequence

while ¬A.isEmpty() Ù ¬B.isEmpty()

a <- A.first().element(); b <- B.first().element()

if a < b

aIsLess(a, S); A.remove(A.first())

else if b < a

bIsLess(b, S); B.remove(B.first())

else { b = a }

bothAreEqual(a, b, S)

A.remove(A.first());

B.remove(B.first())

while ¬A.isEmpty()

aIsLess(a, S); A.remove(A.first())

while ¬B.isEmpty()

bIsLess(b, S); B.remove(B.first())

return S

• Merge-sort (including sub-routine of merging two sorted lists)

Merge-sort on an input sequence S with n elements consists of three steps:

Divide: partition S into two sequences S1 and S2 of about n/2 elements each

Recur: recursively sort S1 and S2

Conquer: merge S1 and S2 into a unique sorted sequence

Algorithm mergeSort(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if S.size() > 1

(S1, S2) <- partition(S, n/2)

mergeSort(S1, C)

mergeSort(S2, C)

S <- merge(S1, S2)

Algorithm merge(A, B)

Input sequences A and B with n/2 elements each

Output sorted sequence of A È B

S <- empty sequence

while ¬A.isEmpty() && ¬B.isEmpty()

if A.first().element() < B.first().element()

S.insertLast(A.remove(A.first()))

else

S.insertLast(B.remove(B.first()))

while ¬A.isEmpty()

S.insertLast(A.remove(A.first()))

while ¬B.isEmpty()

S.insertLast(B.remove(B.first()))

return S

• Quick-sort (including pivot selection, its affect on performance, and sub-routine of partitioning two lists)

A sorting algorithm based on the divide-and-conquer paradigm

Divide: pick a pivot element x and partition S into

L elements less than x

E elements equal to x

G elements greater than x

Recur: sort L and G

Conquer: join L, E and G

The choice of the pivot affects the algorithm’s performance.

Algorithm partition(S, x)

Input sequence S, pivot element x

Output subsequences L, E, G

L, E, G <- empty sequences

while ¬S.isEmpty()

y <- S.remove(S.first())

if y < x

L.insertLast(y)

else if y = x

E.insertLast(y)

else { y > x }

G.insertLast(y)

return L, E, G

• Bucket-sort and radix-sort

Bucket-Sort

Algorithm bucketSort(S, N)

Input sequence S of (key, element) items with keys in the range [0, N - 1]

Output sequence S sorted by increasing keys

B <- array of N empty sequences

while ¬S.isEmpty()

(k, o) <- S.remove(S.first())

B[k].insertLast((k, o))

for i <- 0 to N - 1

while ¬B[i].isEmpty()

(k, o) <- B[i].remove(B[i].first())

S.insertLast((k, o))

Radix-Sort

Algorithm radixSort(S, N)

Input sequence S of d-tuples such that (0, …, 0) £ (x1, …, xd) and (x1, …, xd) £ (N - 1, …, N – 1) for each tuple (x1, …, xd) in S

Output sequence S sorted in lexicographic order

for i <- d downto 1

bucketSort(S, N)

• Quick-select

A randomized selection algorithm based on the prune-and-search paradigm:

Prune: pick a random element x (called pivot) and partition S into

L elements less than x

E elements equal x

G elements greater than x

Search: depending on k, either answer is in E, or we need to recurse in either L or G

Algorithm Time Notes

selection-sort O(n^2) in-place, not stable

slow (good for small inputs)

insertion-sort O(n^2) in-place, stable

slow (good for small inputs)

quick-sort O(n log n) expected in-place, not stable

randomized

fast (good for large inputs)

heap-sort O(n log n) in-place, not stable

fast (good for large inputs)

merge-sort O(n log n) not in-place, stable

sequential data access

fast (good for huge inputs)