Programming Model - Hash Table!

Save items in a key-indexed table (index is a function of the key)

Hash function - Method for computing array index from key

Issues: - Computing the hash function
- Equality test, method for checking whether two keys are equal
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index

Classic Space-time tradeoff:

No space limitation: trivial hash function with key as index, hash(x) = x.

No time limitation: trivial collision resolution with sequential search (unordered array)

Space and time limitations: hashing (the real world)

Ideal(istic) Goal: Scramble the keys uniformly to produce a table index
- efficiently computable
- each table index equally likely for each key

All java classes inherit a method hashCode(), which returns a 32-bit int.

Requirement: If x.equals(y) then (x.hashCode() == y.hashCode()).

Highly desirable: If !x.equals(y), then (x.hashCode() != y.hashCode()).

5.

Programming Novel

	Search	Insert
BST:	N	N

Priority Queue

Remove (largest or smallest)

Getting (largest N items) out of M

Graphs

DFS

Search to bottom, not finish ⇒ come back / and continue

BFS

Search one level at a time

Programming Models - Priority Queues

Priority Queue: Remove the largest (or smallest) item

Challenge: Find the M largest items in a stream of N items
(N huge, M large)
if (pq.size > M) delete minimum

Order of growth of finding the largest M in a stream of N items:

implementation	time	space
sort	$N \log N$	N
elementary PQ	MN	M
binary heap	$N \log M$	M
best in theory	N	M

Use an array for unordered implementation.
for loop for finding max value and deleting

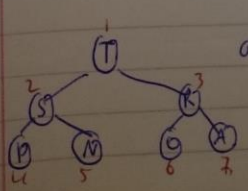
imp	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
Goal:	$\log N$	$\log N$	$\log N$

Binary heap

Binary heap: Array representation of a heap ordered complete binary tree

Heap ordered binary tree: -keys in nodes
-Parent's key no smaller than children's keys

Array representation: -Take nodes in level order.
-No explicit links needed



i:	0	1	2	3	4	5	6	7
a[i]	-	T	S	R	P	N	O	A

2.

Proposition - Largest key is a[1] which is root of binary tree
Proposition: Can use indices (start at 1) to move through tree
- Parent of node at k is at k/2
- Children of node are at k are at 2k and 2k+1

Scenario: Child's key becomes larger key than it parent's key.
To eliminate violation: Exchange key in child with key in parent.
Repeat until heap order restored

```
private void swim(int k) {
  while (k > 1 && less(k/2, k)) {
    exch(k, k/2);
    k = k/2;
  }
}
```

Peter principle: Node promoted to level of incompetence

Insertion:
Add node at end, then swim it up.
Cost: At most 1 + lg N compares.

Scenario: Parent's key became smaller than one (or both) of its children's keys.
To eliminate violation: Exchange key in parent with key of larger child
Repeat until heap order restored

```
private void sink(int k) {
  while (2*k <= N) {
    int j = 2*k;
    if (j < N && less(j, j+1)) j++;
    if (!less(k, j)) break;
    exch(k, j);
    k = j;
  }
}
```

Power struggle: Better sinking provided

Programming Notes - P2

Delete max - Exchange root with node at end, then sink down
Cost: At most 2 log N compares

```
Key max = pq[1]
exch (1, N--);
Sink (1);
pq[N+1] = null;
return max;
```

Impl	Insert	Del max	max
binary heap	log N	log N	1

Heapsort:

Create max-heap with all N keys
Repeatedly remove the maximum key

First pass: build heap using bottom up method

```
for (int k = N/2; k >= 1; k--) {
    sink (a, k, N); }
```

Second pass: Remove maximum one at a time
Leave in array instead of nulling out

```
while (N > 1) {  exch (a, 1, N--);
                 sink (a, 1, N); }
```

Proposition: Heap construction uses fewer than 2N compares and exchanges
Heapsort uses at most 2N log N compares and exchanges

1.

Programming 2 notes

Insert sort.

Input - sequence of n numbers $(a_1, \dots a_n)$

Output - A permutation (reordering) of the input such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Algorithm in English

1. Start from 1st element of the array (optimization: start from second)
2. Shift element back until its right position
3. Continue to next element
4. Repeat (2) and (3) until the end of the array

```
for (j=2 to A.length){
    //Shift A[j] into the sorted A[1, ..j-1]
      i=j-1;
      while (i>0 and A[i] > A[i+1]){
          Swap A[i], A[i+1];
          i = i-1
      }
}
Return A
```

Why is our algorithm correct?

We will make an argument of its correctness using a loop invariant

A loop invariant is a property which is true:

- At the beginning of the algorithm
- At the end of algorithm
- Before each iteration of the algorithm

Insertion Sort - Loop Invariant

At the end: $j = n+1$ Sorted $A[1..n]$
At the start: $j = 2$ Sorted $A[1..1]$ unsorted $A[2..n]$
At line 1: $\boxed{i < j \leq n}$ Sorted $A[1..j-1]$ unsorted $A[j..n]$
LOOP INVARIANT

Runtime Analysis of an algorithm

Each command takes 1 time unit t_i

Best case: $T(n) = n + n-1 + n-1 + 1 = 3n+1$

Worst case: $T(n) = n + n-1 + \text{Sum}_{2..n}(x) + 2\text{Sum}_{2..n}(x-1) + 1$
$= (3/2)n^2 + (3/2)n - 1$

Average we shift each $A[i]$ about $i/2$ positions to the left.
$T(n) = n + n-1 + \text{Sum}_{2..n} i/2 + \text{Sum}_{2..n-1}(x-1) + 1$
$= (3/4)n^2 + (7/4)n - 1/2$

- At large enough input size(s) only the rate of growth of an algorithm's running time matters
- we ignore everything except for the most significant growth function

Insertion Sort asymptotic worst case: $\Theta(n^2)$

Asymptotic Notation O, Ω, o, ω

Exact bounds: $T(n) = \Theta f(n)$
Upper bounds: $T(n) = O f(n)$ or $T(n) = \Theta f(n)$
Non-tight upper bounds: $T(n) = o f(n)$ or $T(n) < \Theta f(n)$
Lower bound: $T(n) = \Omega f(n)$ or $T(n) \geq \Theta f(n)$
Non tight lower bound: $T(n) = \omega(f(n))$ or $T(n) > \Theta f(n)$

Principle
Θ bound are the most precise
O/Ω may be generous
o/w are over generous

Programm 2 Notes

Binary search
Input: array A[0... n-1] integer key
Input property: a is sorted.
Output: integer position
output property: if key == a[i] then pos == i.

Invariant: if key in a[0.. n-1] then key in a[lo... hi]
T(n) = Θ log(n)

Insertion sort: worst case $\Theta(n^2)$

Definition of $T(n) = \Theta(g(n))$
$T(n) = \Theta\, g(n)$ if:
There exist) large enough input size n_0 such that for any $n \geq n_0$:
$c_{low} \cdot g(n) \leq T(n) \leq c_{up} \cdot g(n)$
→ for some constant) c_{low} and c_{up}

Example for $n \geq 10$: $n^2 < \frac{3}{2}n^2 + \frac{3}{2}n - 1 \leq 2n^2$

Definition of $T(n) = O(g(n))$
$T(n) = O\,g(n)$ if
There exist) large enough input size n_0, such that for any $n \geq n_0$:
$T(n) \leq c_{up} \cdot g(n)$
for some constant c_{up}
for $n \geq 10$ $T(n) = (\frac{3}{2})n^2 + (\frac{3}{2})n - 1 \leq 2n^2 \leq n^3 \leq 2^n$

Definition of $T(n) = \Omega(g(n))$
$T(n) = \Omega\, g(n)$ if
There exist) large enough input size n_0, such that for any $n \geq n_0$:
$c_{low} \cdot g(n) \leq T(n)$
for some constant c_{low}
For $n \geq 10$: $1 < n \leq n^2 \leq T(n) = (\frac{3}{2})n^2 + (\frac{3}{2})n - 1$

4

Definition of $T(n) = o(g(n))$ and $T(n) = w(g(n))$

$T(n) = o(g(n))$ if there exists large enough input size n_0 such that for any $n > n_0$:
$T(n) < c \cdot g(n)$ for certain c

$T(n) = w(g(n))$ if there exists large enough input size n_0 such that for any $n > n_0$:
$T(n) \rightarrow \infty$ $c \cdot g(n) < T(n)$ for some const c

Queues And Stacks

Stack: LIFO push(v), pop(), isEmpty()
Queue: FIFO enqueue(v), dequeue(), isEmpty()

STACK - Linked List implementation
isEmpty() { return head == null }
push(item) { newNode, newNode.item = item, newNode.next = head, head = newNode }
pop { item = head.item; head = head.next, return item }
push() $\Theta(1)$ pop() $\Theta(1)$ isEmpty() $\Theta(1)$ All worst case guarantee

- Linked list implementation ↑
- When stack holds N values, it uses N extra memory cells to store pointers
- Often sacrifice memory to gain speed → pay N memory to gain $\Theta(1)$ push/pop

Stack Array Implementation
- Stack overflow
push $\Theta(1)$ pop $\Theta(1)$ isEmpty $\Theta(1)$ all worst case guarantee

- Resizing array ⇒ grow and shrink to avoid overflow
push: when full create new array of double size and copy
Insert 1 item Best $\Theta(1)$ worst $\Theta(n)$
N items: $\Theta(n)$ worst $\Theta(n)$

Programming 2 Notes

Amortized cost- cost of N operations/N starting from an empty data structure

-push has amortized worst case cost of O(1).

Shrinking: shrink by half when the array is one quarter full

if (n>0 && n==s.length/4) resize(s.length/2);

Space? Best case is when stack is full: 1 extra space

Worst: when we are just before shrinking → 3N+1

LL or stack implementation

LL → push/pop take O(1) time in worst case

Always uses N extra spaces for N items

Resizing array → push/pop take O(1) amortized time

push/pop operation take O(n) in worst case

Uses between 1 and 3N+1 extra space for N items

QUEUE

void enqueue(String s) add s at the end (tail) of the Q

String dequeue() remove and returns the first element (head) of Q

Implement via LL.

```
enque(String s){
    QueNode newNode = new QueNode();
    newNode.item=s;
    newNode.next=null;
    if(head==tail){ tail=head=newNode;}
    else{ tail.next=newNode; tail=newNode;}
```

6

Sonny dequeue () {

if (head == null) return null

else if (head == tail) return head.item, head = tail = null;

else return head.item, head = head.next

Time: worst case enqueue: $\Theta(1)$ dequeue $\Theta(1)$

Space: for N elements N (+2 for head & tail pointers)

Programming 2 Notes   Doubly linked Lists

```
class QosNode {
  String item;
  QosNode next, prev;
}
```

insert first ()

- is empty   head = tail = newNode
- head.prev = newnode, newnode.next = head   head = newnode