8.2-3 Argue that insertion sort is better than Quicksort for sorting checks

---

In the best case, Quicksort takes $\Theta(n \lg n)$. Although using median-of-three turns the sorted permutation into the best case, we lose if insertion sort is better on the given data.

$$1\ 2\ 3\ 4\ 6\ 7\ 9\ 11 \text{ --- } 5$$

In insertion sort, the cost of each insertion is the number of items which we have to jump over. In the check example, the expected number of moves per items is small, say c. We win if $c << \lg n$.

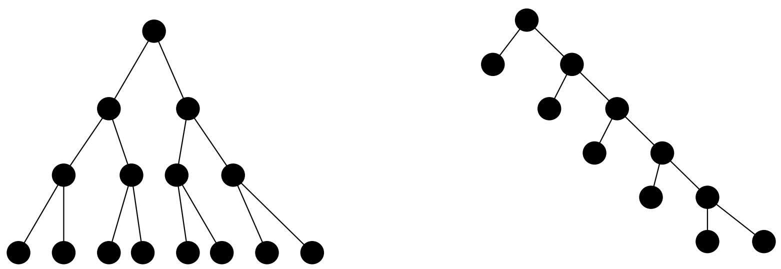
8.3-1 Why do we analyze the average-case performance of a randomized algorithm, instead of the worst-case?

In a randomized algorithm, the worst case is not a matter of the input but only of luck. Thus we want to know what kind of luck to expect. Every input we see is drawn from the uniform distribution.

8.3-2 How many calls are made to Random in randomized quicksort in the best and worst cases?

Each call to random occurs once in each call to partition.

The number of partitions is $\Theta(n)$ in any run of quicksort!!



There is some potential variation depending upon what you do with intervals of size 1 – do you call partition on intervals of size one? However, there is no asymptotic difference between best and worst case.

The reason – any binary tree with n leaves has $n - 1$ internal nodes, each of which corresponds to a call to partition in the quicksort recursion tree.

Elementary Data Structures

"Mankind's progress is measured by the number of things we can do without thinking."

Elementary data structures such as stacks, queues, lists, and heaps will be the "of-the-shelf" components we build our algorithm from. There are two aspects to any data structure:

- The abstract operations which it supports.
- The implementation of these operations.

The fact that we can describe the behavior of our data structures in terms of abstract operations explains why we can use them without thinking, while the fact that we have different implementation of the same abstract operations enables us to optimize performance.

Stacks and Queues

Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.

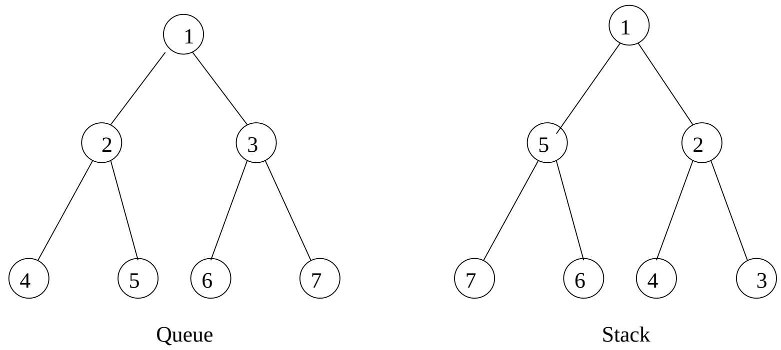
A *stack* supports last-in, first-out operations: push and pop.

A *queue* supports first-in, first-out operations: enqueue and dequeue.

A *deque* is a double ended queue and supports all four operations: push, pop, enqueue, dequeue.

Lines in banks are based on queues, while food in my refrigerator is treated as a stack.

Both can be used to traverse a tree, but the order is completely different.



Which order is better for WWW crawler robots?

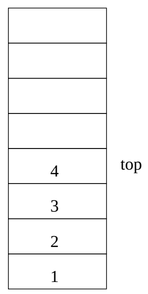
Stack Implementation

Although this implementation uses an array, a linked list would eliminate the need to declare the array size in advance.

STACK-EMPTY(S)
    if top[S] $= 0$
        then return TRUE
        else return FALSE

PUSH(S, x)
    $top[S] \leftarrow top[S] + 1$
    $S[top[S] \leftarrow x$

POP(S)
    if STACK-EMPTY(S)
        then error "underflow"
        else $top[S] \leftarrow top[S] - 1$
            return $S[top[S] + 1]$

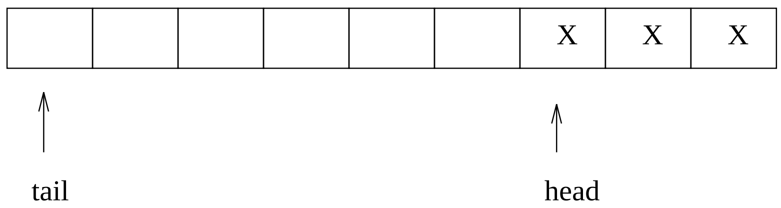


All are $O(1)$ time operations.

Queue Implementation

A circular queue implementation requires pointers to the head and tail elements, and wraps around to reuse array elements.

ENQUEUE(Q, x)
    Q[tail[Q]] $\leftarrow$ x
    if tail[Q] $=$ length[Q]
        then tail[Q] $\leftarrow$ 1
        else tail[Q] $\leftarrow$ tail[Q] $+$ 1



DEQUEUE(Q)
    x $=$ Q[head[Q]]
    if head[Q] $=$ length[Q]
        then head[Q] $=$ 1
        else head[Q] $=$ head[Q] $+$ 1
    return x

A list-based implementation would eliminate the possibility of overflow.

All are $O(1)$ time operations.

Dynamic Set Operations

Perhaps the most important class of data structures maintain a set of items, indexed by keys.

There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.

- *Search(S,k)* – A query that, given a set S and a key value k, returns a pointer x to an element in S such that $key[x] = k$, or nil if no such element belongs to S.

- *Insert(S,x)* – A modifying operation that augments the set S with the element x.

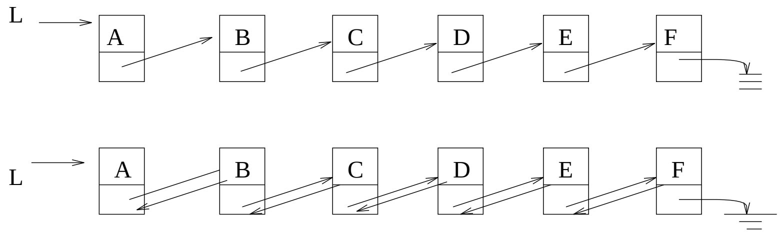
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key value.

- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.

- *Next(S,x), Previous(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) element in S, or NIL if x is the maximum (minimum) element.

Pointer Based Implementation

We can maintain a dictionary in either a singly or doubly linked list.



We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.

Since the extra big-Oh costs of doubly-linkly lists is zero, we will usually assume they are, although it might not be necessary.

Singly linked to doubly-linked list is as a Conga line is to a Can-Can line.

Array Based Sets

Unsorted Arrays

- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot, $O(1)$
- Delete(S,x) - copy nth item to the xth spot, $O(1)$
- Min(S,x), Max(S,x) - sequential search, $O(n)$
- Successor(S,x), Predecessor(S,x) - sequential search, $O(n)$

Sorted Arrays

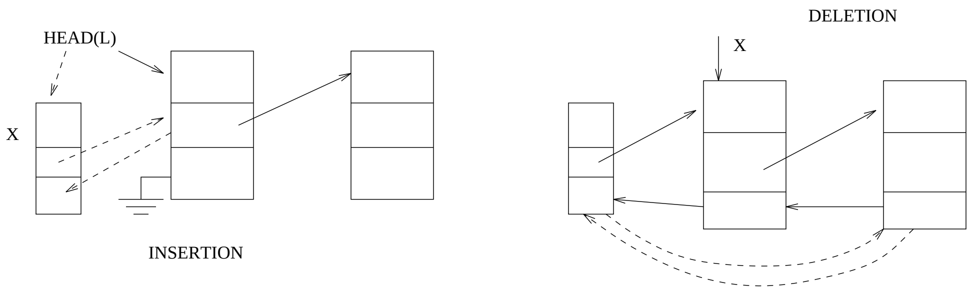
- Search(S,k) - binary search, $O(\lg n)$
- Insert(S,x) - search, then move to make space, $O(n)$
- Delete(S,x) - move to fill up the hole, $O(n)$
- Min(S,x), Max(S,x) - first or last element, $O(1)$
- Successor(S,x), Predecessor(S,x) - Add or subtract 1 from pointer, $O(1)$

What are the costs for a heap?

Unsorted List Implementation

LIST-SEARCH(L, k)
    x = head[L]
    while $x <> NIL$ and $key[x] <> k$
        do x = next[x]
    return x

Note: the while loop might require two lines in some programming languages.

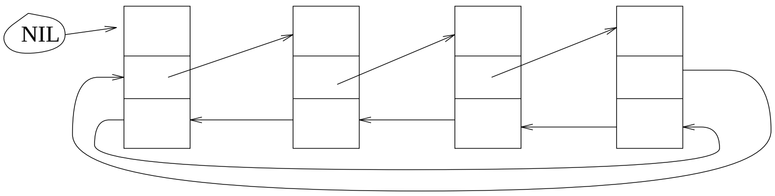


LIST-INSERT(L, x)
    next[x] = head[L]
    if head[L] <> NIL
        then prev[head[L]] = x
    head[L] = x
    prev[x] = NIL

LIST-DELETE(L, x)
    if $prev[x] <> NIL$
        then next[prev[x]] = next[x]
        else head[L] = next[x]
    if $next[x] <> NIL$
        then prev[next[x]] = prev[x]

Sentinels

Boundary conditions can be eliminated using a sentinel element which doesn't go away.



LIST-SEARCH'(L, k)
    x = next[nil[L]]
    while $x <> NIL[L]$ and $key[x] <> k$
        do x = next[x]
    return x

LIST-INSERT'(L, x)
    next[x] = next[nil[L]]
    prev[next[nil[L]]] = x
    next[nil[L]] = x
    prev[x] = NIL[L]

LIST-DELETE'(L, x)
    next[prev[x]] <> next[x]
    next[prev[x]] = prev[x]

Hash Tables

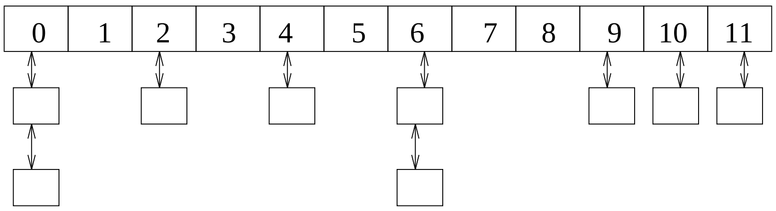
Hash tables are a *very practical* way to maintain a dictionary. As with bucket sort, it assumes we know that the distribution of keys is fairly well-behaved.

The idea is simply that looking an item up in an array is $\Theta(1)$ once you have its index. A hash function is a mathematical function which maps keys to integers.

In bucket sort, our hash function mapped the key to a bucket based on the first letters of the key. "Collisions" were the set of keys mapped to the same bucket.

If the keys were uniformly distributed, then each bucket contains very few keys!

The resulting short lists were easily sorted, and could just as easily be searched!



Hash Functions

It is the job of the hash function to map keys to integers. A good hash function:

1. Is cheap to evaluate

2. Tends to use all positions from $0 \ldots M$ with uniform frequency.

3. Tends to put similar keys in different parts of the tables (Remember the Shifletts!!)

The first step is usually to map the key to a big integer, for example

$$h = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

This large number must be reduced to an integer whose size is between 1 and the size of our hash table.

One way is by $h(k) = k \bmod M$, where M is best a large prime not too close to $2^i - 1$, which would just mask off the high bits.

This works on the same principle as a roulette wheel!

Good and Bad Hash functions

The first three digits of the Social Security Number

0	1	2	3	4	5	6	7	8	9

The last three digits of the Social Security Number

0	1	2	3	4	5	6	7	8	9

The Birthday Paradox

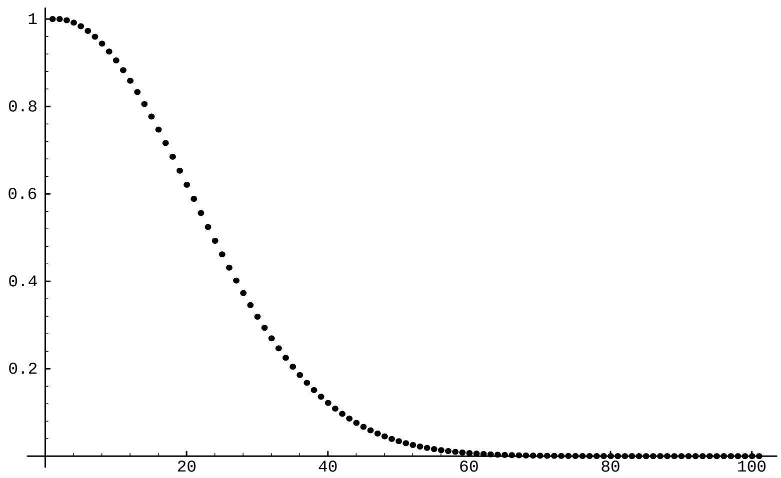
No matter how good our hash function is, we had better be prepared for collisions, because of the birthday paradox.

J	F	M	A	M	J	J1	A	S	O	N	D

The probability of there being *no* collisions after n insertions into an m-element table is

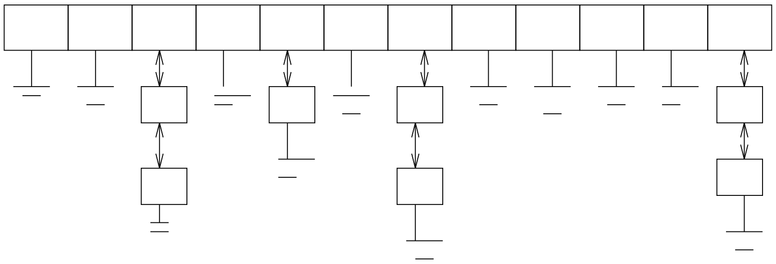
$$(m/m)\times((m-1)/m)\times\ldots\times((m-n+1)/m) = \Pi_{i=0}^{n-1}(m-i)/m$$

When $m = 366$, this probability sinks below 1/2 when $N = 23$ and to almost 0 when $N \geq 50$.



Collision Resolution by Chaining

The easiest approach is to let each element in the hash table be a pointer to a list of keys.



Insertion, deletion, and query reduce to the problem in linked lists. If the n keys are distributed uniformly in a table of size m/n, each operation takes $O(m/n)$ time.

Chaining is easy, but devotes a considerable amount of memory to pointers, which could be used to make the table larger. Still, it is my preferred method.

Open Addressing

We can dispense with all these pointers by using an implicit reference derived from a simple function:

1	2	3	4	5	6	7	8	9	10	11	
		X		X	X		X	X			

If the space we want to use is filled, we can examine the remaining locations:

1. Sequentially $h, h+1, h+2, \ldots$

2. Quadratically $h, h+1^2, h+2^2, h+3^2 \ldots$

3. Linearly $h, h+k, h+2k, h+3k, \ldots$

The reason for using a more complicated science is to avoid long runs from similarly hashed keys.

Deletion in an open addressing scheme is ugly, since removing one element can break a chain of insertions, making some elements inaccessible.

Performance on Set Operations

With either chaining or open addressing:

- Search - $O(1)$ expected, $O(n)$ worst case
- Insert - $O(1)$ expected, $O(n)$ worst case
- Delete - $O(1)$ expected, $O(n)$ worst case
- Min, Max and Predecessor, Successor $\Theta(n + m)$ expected and worst case

Pragmatically, a hash table is often the best data structure to maintain a dictionary. However, we will not use it much in proving the efficiency of our algorithms, since the worst-case time is unpredictable.

The best worst-case bounds come from balanced binary trees, such as red-black trees.