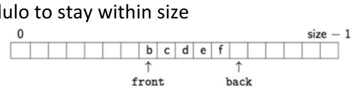
Abstract Data Type – mathematical description and set of operations on the object / interface of a data structure
Data Structure – set of algorithms which implement an ADT / way of storing and organizing data for an ADT

Queue ADT

- create|destroy|enqueue|dequeue|is_empty
- set order of data, breadth first search
- FIFO

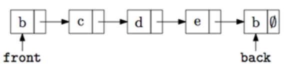
Circular Array Queue

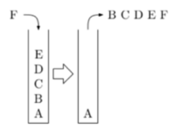
- enqueue|dequeue|is_empty|is_full
- use modulo to stay within size



Linked List Queue

- enqueue|dequeue|is_empty|
- Node *temp = front;
  front = front->next;
  delete temp;



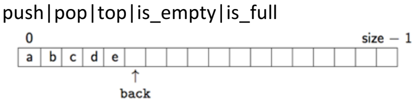


Stack ADT

- create|destroy|push|pop|top|is_empty
- LIFO
- can reverse order if needed, or randomize
- call stack, depth first search, balancing symbols

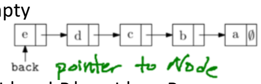
Array Stack

- push|pop|top|is_empty|is_full



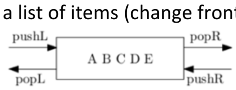
Linked List Stack

- push|pop|top|is_empty



Deque ADT

- create|destroy|pushL|pushR|popL|popR
- maintains a list of items (change front/back)



Analysis of Algorithms

- measure as a function of input size n, and ignore constant factors
- **Big O:** $T(n) \in O(f(n))$ if there are positive constants c and n_0 such that $T(n) \leq cf(n)$ for all $n \geq n_0$.
- **Big Omega:** $T(n) \in \Omega(f(n))$ if there are positive constants c and n_0 such that $T(n) \geq cf(n)$ for all $n \geq n_0$.
- **Big Theta:** $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$.
- **Proof by Big O:**
  - Proof: by definition of Big-O we must find positive constants c and n0 for all n>=n0, T(n) <= c(f(n)).
  - Consider example c = ____, n0 = _____.
  - Change an existing equality to T(n) <= c(f(n)), and thus proven Big-O
- **Example Big O Proof:**
  - Prove 3n^2 + 2n lg(n) is Big O of (n^2)
  - By definition of Big-O, we must find positive constants c and n0 such that for all n>=n0, 3n^2+2n lg(n) <= cn^2
  - Consider c = 5, n0 = 1.
  - For all n >= 1, lg(n) <= n, and therefore 2 lg(n) <= 2n
  - Since c = 5, then c – 3 = 2, and 2 lg(n) <= (c – 3)n
  - 2 lg(n) <= cn – 3n and 2 lg(n) + 3n <= cn
  - 2n lg(n) + 3n^2 <= cn^2, and thus we have proven the Big O. QED.

Typical Asymptotic Relations

Tractable

- constant: $\Theta(1)$
- logarithmic: $\Theta(\log n)$ $(\log_b n,\ \log n^2 \in \Theta(\log n))$
- poly-log: $\Theta(log^k n)$ $(\log^k n \equiv (\log n)^k)$
- linear: $\Theta(n)$
- log-linear: $\Theta(n \log n)$
- superlinear: $\Theta(n^{1+c})$ (c is a constant > 0)
- quadratic: $\Theta(n^2)$
- cubic: $\Theta(n^3)$
- polynomial: $\Theta(n^k)$ (k is a constant)

Intractable

- exponential: $\Theta(c^n)$ (c is a constant > 1)

Eliminate lower order terms and coefficients
Big O– upper bound
Big Omega – lower bound

Analyzing Code

- single operations – CONSTANT TIME
- consecutive operations – SUM TIMES
- conditionals – CONDITIONAL + MAX BRANCH
- loops – SUM OF LOOP BODY TIMES
- functions – FUNCTION TIMES
- loop in a loop – MULTIPLICATION OF LOOP TIMES

Tight Bound

- Big Theta asymptotically tight
- *No better reasonable bound which is different*
- Run time of algorithm matches provable lower bound on any algorithm

Memoization – keep past recursive call values in a table, accessible by future calls – O(n)

Types of Proof

Counterexample

- show an example which does not fit with the theorem, thus, theorem is false (show why)

Contradiction

- assume the opposite of a theorem
- derive a contradiction, thus theorem is true

Induction

- prove for the base case (smallest n)
- assume for all n <= k and prove for the next value (n = k+1).

eg. Show that for n>= 5, 4n <= 2^n
Base case: n = 5, 4*5 = 20 <= 32 = 2^5
Inductive step, rule holds for k < n
4n = 4 (n - 1) + 4, and 2^n = 2^(n-1) + 2^(n-1)
By induction hypothesis, we know that
4(n – 1) <= 2^(n-1), and since n>=5, 4 <= 2^(n-1)
Add inequalities together
4(n – 1) <= 2^(n – 1)
4 <= 2^(n – 1)
4(n – 1) + 4 <= 2^n, and thus we've proven it

Tree Terminology

root – no incoming edges/arcs, or parent (top)
leaf – node with no children
child – node pointed to by "me" node
parent – node that points to "me" node (at most one parent)
sibling – nodes with the same parent
ancestor – parent, or ancestor of a parent
descendent – child, or descendent of a child
subtree – node and its descendants
depth - # of edges on path from root to node
height - # of edges on longest path from node to desendent (whole tree – root to leaf)
degree - # of children of a node
branching factor – max degree of any nodes
complete – max nodes possible for given height
nearly complete – complete, plus some nodes on the left at the bottom
binary – each node has degree at most 2
d-ary – degree at most d

Tree Rules

If a tree has *height* "h", the # of nodes in a complete binary tree is *n = 2^(h+1) – 1*
If a nearly-complete tree has *nodes* "n", the height of the tree is *h = floor(lg(n))*
The longest path in a tree:

- if can contain the root, path = 2 + height of 2 tallest children
- if cannot contain root, path = longest path in any child's subtree

Some Queue Thoughts

Applications

- ordering CPU jobs and simulating events | picking next search site
- short jobs should go first
- earliest events should go first
- most promising sites should be searched first

Log Aside

$\log_b x$ is the exponent b must be raised to to equal x.

- $\lg x \equiv \log_2 x$ (base 2 is common in CS)
- $\log x \equiv \log_{10} x$ (base 10 is common for 10 fingered mammals)
- $\ln x \equiv \log_e x$ (the natural log)

Note: $\Theta(\lg n) = \Theta(\log n) = \Theta(\ln n)$ because

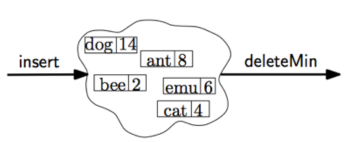
$$\log_b n = \frac{\log_c n}{\log_c b}$$

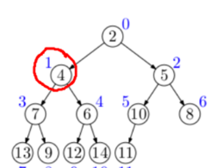
for constants $b, c > 1$.

Priority Queue

ADT

- create|destroy|insert|deleteMin|is_empty
- higher priority elements dequeued first
- good for anything "greedy"
- *Unsorted list* - insert O(1), deleteMin ø(n)
- *Sorted list* – insert ø(n), deleteMin O(1)





Binary Heap Priority Q Data Structure

- deleteMin|swapDown/Up|insert|heapify
- O(log n)   Heapify: O(n) at most h steps
- **heap-order property:** parent's key <= children's keys (min at top)
- **structure property:** "nearly complete tree" – depth O(log n) – height above
- Store nodes in array
- left_child(i) = 2i + 1|right_child(i) = 2i + 2
- parent(i) = floor((i-1)/2) | root = 0
- next free position = n | n = size

0	1	2	3	4	5	6	7	8	9	10	11	12
2	4	5	7	6	10	8	13	9	12	14	11	

Recursion (***trust the natural recursion***! *Break it down to see it simpler)*

Proof:
- base case
- inductive hypothesis – works for smaller-sized inputs
- inductive step – breaks problem down
- *proving loops using loop invariants*

Loop Invariant Proof:
- base case (prove true before loop)
- inductive hypothesis – true before part
- inductive step – true at end of part
- prove loop eventually ends

recurrence relation – formula that relates each term to its predecessors

Recursion VS Iteration
- elegance
- iteration avoids fn calls on the stack
- similar efficiency

tail recursion – no overflow

to iteration:
- change parameters passed in values (inc or dec) to simulate
- start executing at start of function (may use a while loop)
- clean up code
- check control flow

Recurrence Relations

Algorithm	Recurrence	Big O	
Binary Search	T(n) = T(n/2) + O(1)	O (log n)	1. find recurrence relation
Sequential Search	T(n) = T(n-1) + O(1)	O(n)	2. extrapolate
Tree Traversal	2 T(n/2) + O(1)	O(n)	3. figure out pattern
Selection Sort	T(n-1) + O(n)	O(n^2)	4. simplify
Mergesort	2 T(n/2) + O(n)	O(n log n)	5. find Big O

```
int max(A, n)
  if( n == 1 ) return A[0]
  return larger of A[n-1] and max(A, n-1)
```
b C is

Recursion almost always yields a recurrence relation:

$T(1) \le b$

$T(n) \le c + T(n-1)$ if $n > 1$

Solving recurrence:

$T(n) \le c + c + T(n-2)$ (substitution)

$\le c + c + c + T(n-3)$ (substitution)

$\le kc + T(n-k)$ (extrapolating $k > 0$)

$= (n-1)c + T(1)$ (for $k = n-1$)

$\le (n-1)c + b$

$T(n) \in O(n)$

Mergesort algorithm:

Split list in half, sort first half, sort second half, merge together

Recurrence relation:

$T(1) \le b$

$T(n) \le 2T(n/2) + cn$ if $n > 1$

Solving recurrence:

$T(n) \le 2T(n/2) + cn$

$\le 2(2T(n/4) + cn/2) + cn$ (substitution)

$= 4T(n/4) + 2cn$

$\le 4(2T(n/8) + cn/4) + 2cn$ (substitution)

$= 8T(n/8) + 3cn$

$\le 2^k T(n/2^k) + kcn$ (extrapolating $k > 0$)

$= nT(1) + cn \lg n$ (for $2^k = n$) $k = \lg$

$T(n) \in O(n \log n)$

Sorting (stable = identical keys same order / potential memory usage)

Name	Best Case	Worst Case	Stability	Memory
Insertion Sort	Θ(n)	Θ(n^2)	can be stable	in-place - 1
Heap Sort	Θ(n lg n)	Θ(n lg n)	stable w/ index	in-place - 1
Merge Sort	Θ(n lg n)	Θ(n lg n)	can be stable	Ω(n) extra
Quick Sort	Θ(n lg n)	Θ(n^2)	can be stable	in-place – log n

Hashing

Basic Hashing Concepts
- choose a fast, low-collision hash function: hash(x) = x mod m, where m is a *prime number*
- **universal hash function**– if probability of a collision is at most 1/(sizeofarray)
- **General Form**

1. map key to a sequence of bytes
2. map bytes to an integer x
3. map x to a table index using x mod m

- **pigeonhole principle** – if more than m pigeons fly into m pigeonholes then some pigeonhole contains at least 2 pigeons
  if n pigeons fly into m holes, at least one hole contains at least
  k = celing(n/m) pigeons
- **corollary** – if we hash n>m keys into m slots, keys will collide
- **birthday paradox** – if we randomly hash sqrt(2m) keys into m slots, we get a collision with probability > ½
- **tombstone** – after deletion

insert treats as empty, find treats as full

Dealing with Collisions

separate chaining – each entry is a linked-list-style implementation (store multiple items in each entry)
- can hash more than m items into a size m table
- performance depends on length of chains
- **load factor** = (# of hashed items) / (table size)
- new memory on each add. insert

open addressing – only allows one item in each slot – keep trying
- cannot hash more than size
- memory allocated once
- **probe sequence** – sequence we examine when insert/find
- *h(k,i)* maps k and i to result of h
  - k = key and i = # of tries

rehashing – when LF gets too large takes Θ(n) time, amortized O(1) if doubling table size (flexibility) spreads keys out, clears tombstone

<u>**linear probing**</u>

h(k,i) = (hash(k) + i) mod m)

suffers from primary clustering

(long chain consecutive filled slots)

performance degrades for LF > ½

<u>**quadratic probing**</u>

h(k,i) = (hash(k) + i^2) mod m)

first celing(m/2) probes distinct

suffers from secondary clustering

(initial slot follows same probe sequence)

<u>**double hashing**</u> *h(k,i) = (hash(k) + i*hash2(k)) mod m)*

hash2(k) should be quick, differ from hash(k), and never be 0 (mod m)

no clustering – one extra hash calculation

Trees

AVL Trees

binary tree – max 2 children/node

search tree – left < key, right > key

balanced – |bal(node)|<=1 for all

height = Θ(log n)

If |bal|<=1, height <= c lg n (c<2)

O(log n) for find, insert, delete

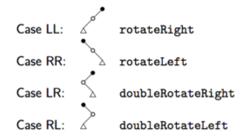
rotateLeft(a) ***Node* b = a->right***
a->right = b->left
b->left = a
updateHeight(both)
b=a

rotateRIght(a) reverse left/right in rotateLeft(a), and a = b instead

doubleRotateLeft(a)
rotateRight(a->right)
rotateLeft(a)

doubleRotateLeft(a) reverse left/right in doubleRotateRight(a)

Insert Algorithm
- find place for new key & add
- search for imbalance



Delete Algorithm
- BST delete and fix imbalances

Bal(x) = height(left)-height(right)

B+ Trees

m-ary tree – max M children/node

search tree – max M-1 search keys

nodes have ceil(M/2) < # children < L

leaves hold ceil(L/2) < # KV pairs < L

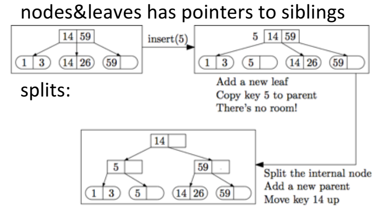
height, find,insert,delete = Θ(logM n)

all leaves are at the same depth

values stored in leaves and internal nodes are search keys (root can both)

nodes&leaves has pointers to siblings

splits:



Insert Algorithm
- Insert key,value pair in its leaf
- If overflow, split (copy[leaf] or move[node] smallest key in new (middle)up to parent)

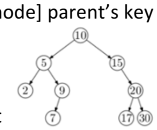
Delete Algorithm
- Remove key value pair from leaf
- If underflow, take from sibling if possible and update parent key OR merge with sibling and delete[leaf] or pull down[node] parent's key

Traversal

In-order:2 5 6 9 10 15 17 20 3|in(L), root, in(R)

Pre-order:10 5 2 9 7 15 20 17 30|root, pre(L), pre(R)

Post-order:2 7 9 5 17 30 20 15 10|post(L), post(R), root



Parallelism(improve performance by using many processors) fork|join|reduce

divide n-sized array into k pieces (threads)

solve pieces in parallel, time is Θ (n/k)

combine and sum results, time is Θ (k)

*total time is **Θ(n/k) + Θ(k)**, best k is sqrt(n)*

Switch to sequential if tasks is more work OR (**divide and conquer**)

recursively divide array into CUTOFF-sized pieces, time is Θ (log n)

solve these pieces in parallel, time, Θ(CUTOFF)

combine by summing results, time, Θ (log n)

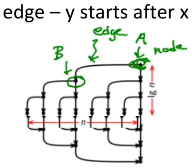
*total time is **Θ (log n)***

FORALL x in A - do every iteration in parallel

DAG – models parallelism

node – constant seq. work

edge – y starts after x done



Let $T_P(n)$ be the run time of a parallel program with P processors on input size n.

$T_1(n)$ = **work**, the number of nodes|$T_{infinity}(n)$ = **span**, # nodes on the longest path

$T_P(n)$ => $T_1(n)$, otherwise we didn't do all the work, AND $T_p(n)$ >= $T_{inf}(n)$ as P < inf

$$T_p(n) \in \Theta(T_1(n)/P + T_\infty(n)). \qquad T_P(n) \in \Theta(n/P + \log n)$$

Amdahl's Law – the overall speedup with P processors is $\frac{T_1(n)}{T_P(n)} \le \frac{1}{s + (1-s)/P}$

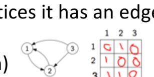
The overall speedup with inf. Processors is $\frac{T_1(n)}{T_\infty(n)} \le \frac{1}{s}$.

<u>**Graphs**</u>: create|insert|iterate|edge? **n=#nodes, m=#edges**

Topological Sort – total order of vertices in a graph G = (V,E) such that if (u,v) is an edge of G, then u appears before v in that order (time is Θ(#nodes+#edges))

1. Find each vertex's in-degree(# of in-edges) and make queue for 0 in-degrees
2. If there are vertices in queue, dequeue and output. Then reduce the in-degrees of all vertices it has an edge to, and enqueue all vertices with in-d 0

Adjacency Matrix



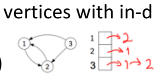
Iterate vertices Θ(n)

Iterate edges $\Theta(n^2)$, exist? Θ(1)

Iterate vertices adj. to vertex Θ(n)

Memory $\Theta(n^2)$

Adjacency List



Iterate vertices Θ(n)

Iterate edges Θ(m), exist? (u,v) Θ(OutD u)

Iterate vertices adj. to u Θ(OutD u)

Memory Θ(n+m)

handshaking theorem - undirected graph $\sum_{v \in V} \deg(v) = 2|E|$

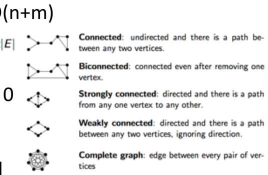
Dijkstra's Algorithm – find shortest path to vertex

Set dist. to each vertex to inf., and dist(source) to 0

while unmarked vertices, mark v with lowest dist

for edge (v,w),*dist(w)=min (dist(w), dist(v) + weight (v,w)*

proven Cloud Thrm – no shorter path out of cloud



Runtime: O((|V|+|E|) log (|V|)) – V times find/deleteMin, E times changeKey

Spanning tree – tree subset from a connected graph that touches all vertices

Kruskal's Algorithm – find spanning tree with lowest total edge dist.

Start empty tree, add min weight edge to T unless cycle forms: O(|E|log|E|)