Summary
Algorithms & Data Structures

Marie Simon s1023848

January 17, 2020

1 Notations

Big-oh O provides an asymptotic upper bound on a function

Big-omega Ω provides an asymptotic lower bound on a function

Big-theta Θ denotes asymptotic tight bounds. $\Theta(g) = O(g) \cap \Omega(g)$

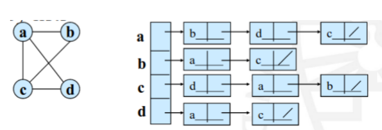
2 Graphs

- A Graph $G = (V, E)$ consists of the nonempty set V of vertices/nodes and a set E of edges
- $E \subseteq V \times V$
- $(u, v) \in E$ is an edge from u to v also denoted $u \to v$
- Self-loops are allowed
- **Directed** edge has direction
- **Undirected** edge goes both ways. Self-loops usually not allowed
- **Weighted** each edge as an associated weight given by function $w : E \to \mathbf{R}$
- **Dense**$|E| \simeq |V|^2$
- **Sparse**$|E| << |V|^2$
- A graph is connected if there is a path between every pair of vertices $|E| \geq |V| - 1$
- A graph is a tree if it is undirected, connected and $|E| = |V| - 1$ (equivalently, each pair of nodes is connected via a unique path)

2.1 Representation

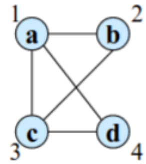
Adjacency List: Consists of an array Adj of $|V|$ lists (one list per vertex)

Efficient for sparse graphs



Adjacency Matrix: $V \times V$ matrix A

Symmetric for undirected graphs



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

3 Breadth first search

Expands frontier first. We use colors to keep track of process
White: undiscovered
Gray: Discovered but not finished
Black: Finished (all adjacent vertices are discovered)

```
for each vertex v in G
  color[v] = white
  dist[v] = infinity     //distance from s to v
  p[v] = nil            //predecessor of v
color[s] = grey
dist[s] = 0
p[s] = nil
Q = empty
Q = successors of s
while Q not empty
   u = next in queue
      for each v adjacent to u
         if(color[v] = white)
             color[v] = grey
             d[v] = d[u] + 1
             p[v] = u
             put all successors of v in Queue
      color[u] = black
```

3.1 Analysis

- Initialization takes $O(V)$
- Traversal en-queues and de-queues each vertex at most once, so $O(V)$
- Adjacency list of each vertex is scanned at most once. Sum of lengths of al adjacency lists is $O(E)$
- **Total running time** $O(V + E)$

4 Depth first search

Searches as deep as possible first. If any undiscovered vertices remain, one of them is chosen a new source and search is repeated from there.
`d[v]` discovery time `f[v]` finishing time

```
DFS(G)
  for each vertex v in G
     colro[v] = white
     p[v] = nil
  time = 0
  for each vertex v in G
     if color[v] = white
        DFS-Visit(v)
```

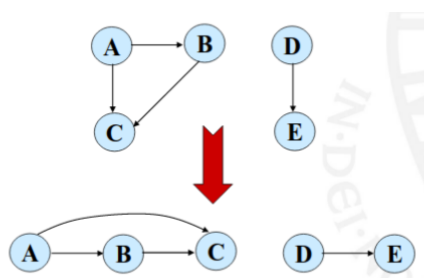
```
DFS-Visit(v)
  color[v] = grey
  time++
  d[v] = time
  for each u adjacent to v
     if(color[u] = white)
        p[u] = v
        DFS-Visit(u)
  color[v] = black
  time++
  finished[u] = time
```

4.1 Analysis

- Initializing loop and loop to start DFS take $O(V)$ time
- DFS-Visit is called once for each vertex $v \in V$
- Loop in DFS-Visit executes as often adjacent vertices are there, so `Adj[v]` times.
- Total cost of DFS-Visit is $\sum_{v \in V} |Adj[v]| = O(E)$
- **Total running time** $O(V + E)$

5 Topological sort

- Sorting a directed acyclic graph
- Original graph is partial order, want to extend it to total order




```
1. call DFS(G) to computer f[v] for all v
2. as each vertex is finished, insert onto the front of a linked list
3. return linked list of vertices
```

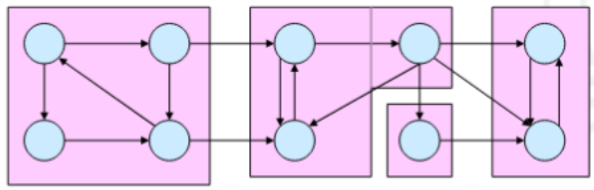
Running time $O(V + E)$

6 Strongly Connected Components

- G is strongly connected if every pair (u, v) of vertices in G is reachable from one another
- A strongly connected component (SCC) of G is a maximal set of vertices $C \subseteq V$ such that for all u and $v \in C$ there is $u \rightsquigarrow v$ and $v \rightsquigarrow u$
- To determine you need transpose of directed graphs

  G^T = transpose of directed Graph

  G^T is G with all edges reversed



```
1. call DFS(G) to determine f[u] for all u
2. compute G^T
3. call DFS(G^T) but consider vertices in 4. order of decreasing f[u]
4. output vertices in each tree of the depth-first forest formed in second DFS as a seperate SCC
```

Running time $O(V + E)$

7 Heaps

- Array views as a nearly complete binary tree. Array elements map to tree nodes:
  - Root: `A[0]`
  - Left[i]: `A[2i]`
  - Right[i]: `A[2i+1]`
  - Parent[i]: `A[i/2]`
- A heap has the largest element stored at root
- In any subtree, no values are larger than value stored at subroot
- For Min-heap the same but for minimal element
- Height of a heap is $\lfloor \lg n \rfloor$, thus basic operations on a heap run in $O(\lg n)$ time
- Number of leaves: $\lceil \lg n \rceil$
- Number of nodes of height $h \leq \frac{n}{2^{h+1}}$

8 Heap Sorting

- Sorts in place
- creates a data structure (heap) to manage information

Steps in sorting:

1. Convert given array of size n to a heap
2. Swap first and last element of array, so that element is in position where it belongs (`BuildHeap`)
3. That leaves $n - 1$ elements to be placed in right locations
4. But, first $n - 1$ elements are not a heap anymore
5. Float the element at the root down one if its subtrees so that the array remains a heap (`Heapify`)
6. Back to step 2 until sorted

Code for sorting from small to big, so max heaps:

```
Heapify(A, i)
  left = left(i)
  right = right(i)
  if(l <= heapSize(A) and A[l] > A[i])
    largest = l
  else
    largest = i
  if(r <= heapSize(A) and A[r] > A[largest])
    largest = r
  if(largest = i)
    switch (A[i], A[largest])
    Heapify(A, largest)

BuildHeap(A)
  heapSize[A] = length(A)
  for int i = length[A]/2; i = 1; i--
    Heapify(A, i)

 HeapSort(A)
   BuildHeap(A)
   for int i = length[A]; i = 1; i--
     switch(A[0], A[i])
     heapSize--;
     Heapify(A, i)
```

8.1 Analysis

- Time for Heapify: $O(\lg n)$
- Time for building heap: $O(n)$
- **Total running time** Building heap is linear time and then each of the $n-1$ calls to heapify. So: $O(n \log n)$

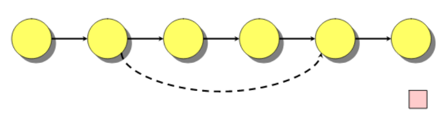
9 Paths

Shortest path $\delta(u, v)$ from u to v is a path of minimum weight u to v

$\delta(u, v) = \infty$ if no path from u to v exists

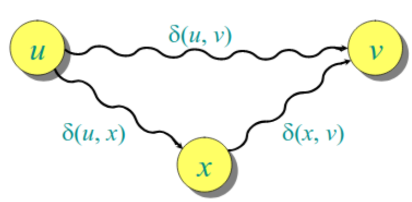
Optimal substructure A subpath of a shortest path is a shortest path

Proof: Cut and paste:



Triangle inequality For all $u, v, x \in V$ we have that $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$

Proof:



Well-definedness of shortest path If a graph contains a negative weight cycle, then some shortest paths may not exist

10 Dijkstra

From a given source vertex $s \in V$, find the shortest-paths weights $\delta(s, v)$ for all $v \in V$.
If all edge weights $w(u, v) \geq 0$, all shortest-path weights must exist.
Idea: greedy algorithm

1. Maintain set S with vertices where we know the shortest path from source s
2. At each step add the vertex that is estimated as the minimal distance from s and not yet in S
3. Update the distance to estimates of vertices adjacent to v

- For unweighted graphs simply use FIFO queue instead of priority queue

```
d[s] = 0
for each vertex v except s
  d[v] = infinity
S = empty
Q = all vertices in priority queue
while Q not empty
  u = extractMin(Q)
  S.add(u)
  for each v that is adjacent to u
    if(d[v] > d[u] + w(u, v)
      d[v] = d[u] + w(u, v)     //implicit decrease key
```

10.1 Analysis

- The for-loop executes as often as $degree(u)$
- The while loop runs $|V|$ times
- In those loops are the functions `extractMin` and the implicit `decrease Key`
- **Total running time:** $O(V) \cdot T_{\text{ExtractMin}} + O(E) \cdot T_{\text{DecreaseKey}}$

11 Bellman-Ford

Solution for graphs with negative cycles. Finds all shortest-path lengths from a source $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

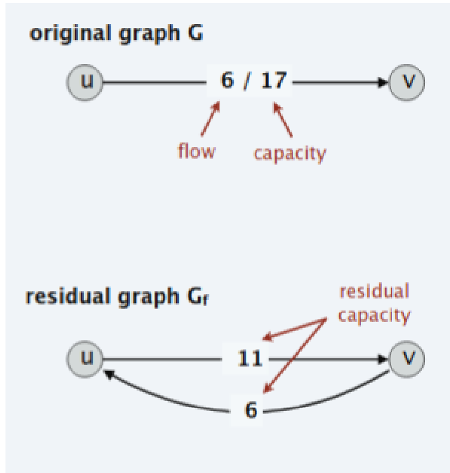
```
d[s] = 0
for each vertex V except s
  d[v] = infinity
for i = 1 to i <= |V|-1
  for each edge (u,v)
    if(d[v] > d[u] + w(u, v))
      d[v] = d[u] + w(u, v)
for each edge (u, v)
  if(d[v] > d[u] + w(u, v))
     report negtive weight cycle
```

Running time $O(VE)$

12 Network flow

- Digraph $G = (V, E)$ with source $S \in V$ and sink $t \in V$
- Nonnegative integer capacity $c(e)$ for each $e \in E$

- **Minimum cut problem** A st-cut is a partition (A, B) of the vertices with $s \in A$ and $t \in B$. Capacity is the sun of the capacities of the edges that go from A into B
- **Bottleneck capacity** of an augmenting P is the minimum residual capacity of any edge in P
- **Residual graphs**



13 Ford-Fulkerson

- Finds the maximum flow
- Start with $f(e) = 0$ for each edge $e \in E$
- Augment flow along path P
- Repeat until you get stuck

```
AUGMENT(f, c, P)
  b = bottleneck Capacity of path P
  for each edge e in P
    if(e in E)
       f(e) = f(e) + b
    else
      f(e^R) = f(e^R) - b
   return f

FORD-FULKERSON(G, s, t, c)
  for each edge e
    f(e) = 0
     G = residual graph
  while there is an augmenting path P in G
    f = AUGMENT(f, c, P)
    update G
  return f
```

Total running time $O(mnC)$ where C is the maximum capacity on edges, n is the number of nodes

14 Capacity scaling algorithm (Dinic)

- With a runtime like Ford-Fulkerson, if C is very large, the running time gets very large
- Need to make wiser decisions on augmenting paths

- Idea: No need to find the exact highest bottleneck path. Maintain a scaling parameter Δ (which should be a lot smaller than most edges)
- Let $G_f(\Delta)$ be the subgraph of the residual graph, only consisting of arcs with a capacity $\geq \Delta$

```
for each edge e in Edges
  f(e) = 0
Delta = largest power of 2 <= C

while(Delta >= 1)
  G_f(Delta) = Delta-residual graph
  while(there is an augmenting path P in G_f(Delta))
    f = AUGMENT(f, c, P)
    update(G_f(Delta))
  Delta = Delta/2

return f
```

14.1 Analysis

- Outer while loop repeats $1 + \log_2 C$ times
- There are at most $2m$ augmentations per scaling phase
- The scaling max-flow alorithm finds a max flow in $O(m \log C)$ augmentations
- **Total running time** $O(m \log C)$

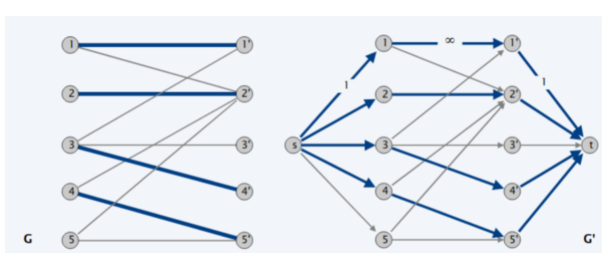
15 Shortest augmenting path

```
for each e in Edges
  f(e) = 0
G_f = residual graph
while(there is an augmenting path in G_f)
  P = BFS(G_f, s, t)
  f = AUGMENT(f, c, P)
  update(G_f)
return f
```

Total running time $O(m^2 n)$

16 Bipartite Matching

- Can be formulated as a max-flow problem
- Add a source and a sink node
- Connect source to each node in L with unit capacity and each node in R with unit capacity to sink
- Connect L to R with unit/infinite capacity
- Solves bipartite matching in $O(mn)$ time



- Hopcroft-Karp can solve it in $O(mn^{1/2})$ time

17 Binary Search Trees

- Each node has at most 2 children, thus
  - Storage is small
  - Operations are simple
  - Expected depth is small
- All keys in left subtree are smaller/equal than root's key
- All keys in right subtree are smaller/equal than root's key
  - easy to find any given key
  - Insert/delete by changing links
- **Complete binary search tree** links are completely filled except possibly bottom-level which is filled left to right

17.1 In-Order-Tree-Walk

```
InOrderTreeWalk(x)
  if(x != NIL)
    InOrderTreeWalk(x.left)
    print x.key
    InOrderTreeWalk(x.right)
```

Running time: $O(n)$

17.2 Recursive find

```
TreeSearch(x, find)

if(x == NIL or find == x.key)
  return x
if(find < x.key)
  return TreeSearch(x.left, find)
else
  return TreeSearch(x.right, find)
```

Running time $O(h)$

17.3 Insertion

```
TreeInsert(T, z)
y = NIL        //trailing pointer
x = T.root
while x != NIL
  y = x
  if(z.key < x.key)
    x = x.left
  else
    x = x.right
z.p = y
if y == NIL
```

```
  T.root = z      //tree T was empty
else if (z.key < y.key)
  y.left = z
else
  y.right = z
```

17.4 Tree-Successor

```
Tree-Successor(x)
  if(x.right != NIL)
    return Tree-Minimum(x.right)
  y = x.p
  while y != NIL and x == y.right
    x = y
    y = y.p
  return y
```

17.5 Deletion

Leaf case: Easy, just delete

One child case: Delete and reconnect the tree

Two child case: Replace node with descendant whose value is guaranteed to be between left and right subtrees: the successor

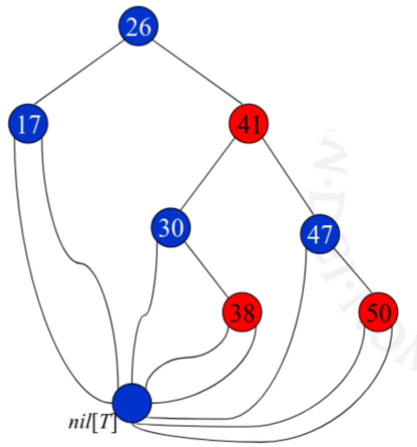
18 AVL trees

- Special type of binary search trees
- For every node require heights of left and right children differ by at most ± 1
- Each node stores its height
- This way the worst case gets reduced since

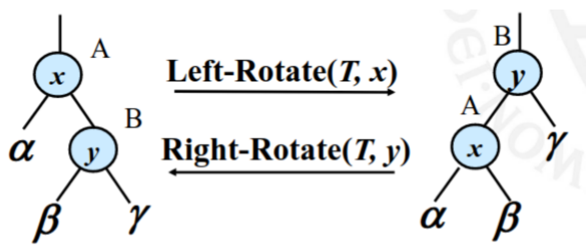
$$
\begin{aligned}
N_h &= N_{h-1} + N_{h-2} + 1 > F_h \qquad h\text{-th Fibonacci Number} \\
F_h &= \frac{\phi^h - \psi^h}{\sqrt{5}} \qquad \text{Binet's formula} \\
h &= O(\lg N_h)
\end{aligned}
$$

19 Red-Black Trees

- Just like binary search trees but with one extra bit per node: attribute whether it is red or black
- We use a single sentinel, *nil*, for all leaves of red-black tree
    1. Every node is either red or black
    2. All empty trees/leaves are colored black
    3. The root is black
    4. If a node is red, then both its children are black
    5. For each node, all paths from node to descendant leaves contain same number of black nodes



- **Height of a node** $h(x)$ = number of edges in a longest path to a leaf
- **Black-height** $bh(x)$ = number of black nodes (including *nil[T]*) on the path from x to leaf, not counting x
- Relation: $bh(x) \leq h(x) \leq 2bh(x)$
- All operations can be performed in $O(\lg n)$ time



19.1 Left-Rotate

```
Left-rotate(T, x)
  y = right[x]      //we assume y is not nil[T]
  right[x] = left[y]     //turn left subtree into x's right subtree
  if(left[y] != nil[T])
    p[left[y]] = x
  p[y] = p[x]     //link x parent to y
  if(p[x] = nil[T])
    root[T] = y
  else if(x == left[p[x]])
    left[p[[x]] = y
  else
    right[p[x]] = y
  left[y] = x     //put x on y's left
      p[x] = y
```

Constant time

19.2 Insertion

- Use tree-insert from BST (slightly modified)
- Color the node red
- Fix modified tree by re-coloring and rotating to preserve RB tree properties

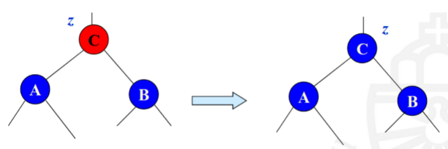
```
TreeInsert(T, z)
  y = nil[T]        //trailing pointer
  x = T.root
  while x != NIL
    y = x
    if(z.key < x.key)
      x = x.left
    else
      x = x.right
  z.p = y
  if y == NIL
    T.root = z      //tree T was empty
  else if (z.key < y.key)
    y.left = z
  else
    y.right = z
  z.left = nil[T]
  z.right = nil[T]
  color[z] = red
  RB-Insert-FixUp(T, z)
```

Only difference

- Needs to connect inserted node to $nil[T]$ node
- Need to color that node red

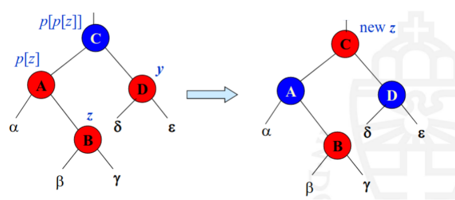
19.3 FixUp

Case 0: z is a root



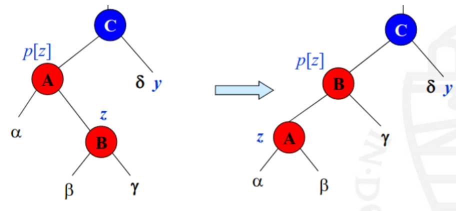
- Make z black $\rightarrow$ restores property 2

Case 1: uncle y is red



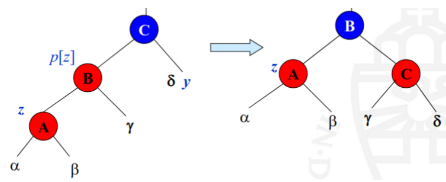
- zs grandparent($p[p[z]]$) must be black, since z and $p[z]$ are both red and there are not other violations of property 4
- Make $p[z]$ and y black $\rightarrow$ now z and $p[z]$ are both not red. Now property 5 might be violated
- Make $p[p[z]]$ red $\rightarrow$ restores property 5
- Next iteration has $p[p[z]]$ as new z

Case 2: uncle y is black, z is a right child



- Left rotate around $p[z]$ so that it switches role with z → now z is a left
- Takes us to case 3

Case 3: uncle y is black, z is a left child

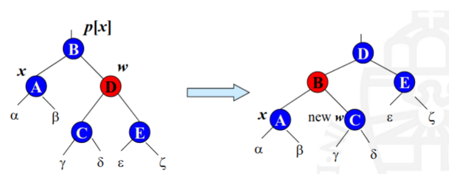


- Make $p[z]$ black and $p[p[z]]$ red
- Then right rotate on $p[p[z]]$. Ensures property 4 is maintained
- No longer have 2 red in a row
- $p[z]$ is now black → no more iterations

19.4 Deletion

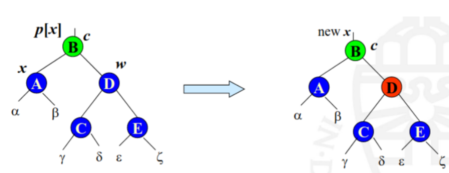
- Almost same deletion as BST
- Need to have fixup strategies again

Case 1: w is red



- w must have black children
- Make w black and $p[x]$ red (because w is red $p[x]$ could not be red before)
- Then left rotate on $p[x]$
- New sibling of x was a child of w before rotation → must be black
- Go immediately to next matching case

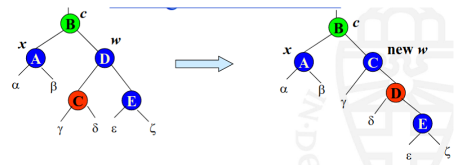
Case 2: w is black, both w's children are black



- Take 1 black off x (→ singly black) and off w (→ red)
- Move that black to $p[x]$

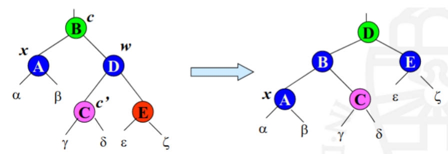
- Do the next iteration with $p[x]$ as the new x
- If entered this case from case 1, then $p[x]$ was red $\rightarrow$ new x is red&black $\rightarrow$ color attribute of new x is RED $\rightarrow$ loop terminates. Then new x is made black in the last line

Case 3: w is black, w's left child is red, w's right child is black



- Make w red and w's left child black
- Then right rotate on w
- New sibling w of x is black with a red right child $\rightarrow$ case 4

Case 4: w is black, w's right child is red



- Make w be $p[x]$'s color c
- Make $p[x]$ black and w's right child black
- Then left rotate on $p[x]$
- Remove extra black on x ($\rightarrow$ is not singly black) without violating any red-black properties
- All done. Setting x to root causes loop to terminate

20 Divide and conquer with Merge Sort as example

- *Divide* a problem into *independent* sub-problems
- *Conquer* the sub-problems
- *Combine* the solutions of a sub-problem into a solution to the original problem
- Typically recursive
- Examples:
  - Euclid's algorithm for computing greatest common divisior
  - Merge sort and quick sort

20.1 Merge Sort

- *Divide* vector in two vectors of similar size
- *Conquer:* recursivley sort the two vectors (trivial for size 1)
- *Combine* two ordered vectors into a new ordered vector $\rightarrow$ auxiliary `merge` function
- For `merge` think of two sorted piles of cards, both placed face up. Merge those by choosing the smaller one at all times

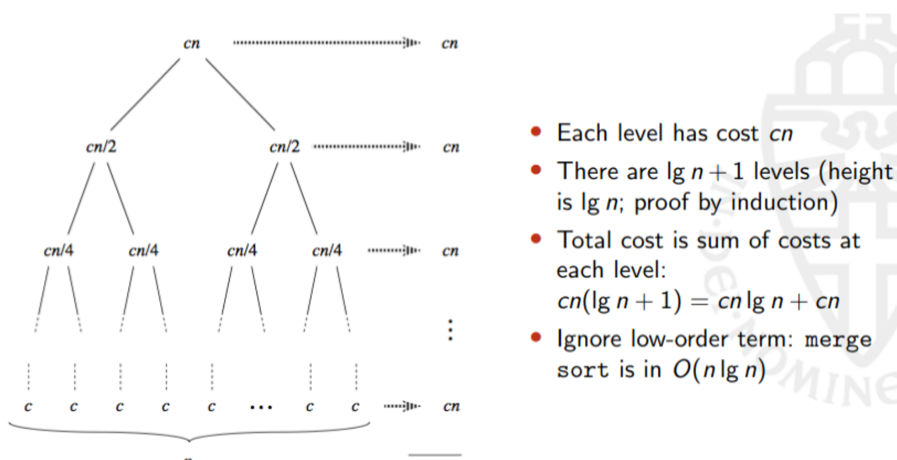
```
void mergeSort(int A[], int p, int r)
  if(p < q)       //base case..?
    q = (p+r)/2;      //Divide
    mergeSort(A, p, q)    //Conquer
    mergeSort(A, q+1, r)     //Conquer
    merge(A, p, q, r)     //Combine

 void merge(int A[], int p, int q, int r)
   int L[MAX], R[MAX]
   int n1 = q-p+1     //length of A[p..q]
   int n2 = r-q     //length of array A[q+1..r]
   for(i = 1; i <= n1; i++)
     L[i] = A[p+i-1]
   for(j = 1; j<= n2; j++)
     R[j] = A[q+j]
   L[n1+1] = MAXINT    //sentinel
   R[n2+1] = MAXINT    //sentinel
   i = 1; j = 1;
   for(k = p; k<= r; k++)
     if(L[i] <= R[i])
       A[k] = L[i]
       i++
     else
       A[k] = R[j]
       j++
```

20.2 Analysis

- `Merge` executes in $O(n)$ time
- *Conquer* Two problems of size $n/2$ are solved: $2T(n/2)$
- Thus (left is assuming array size is a multiple of 2)

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(1) + 2T(n/2) + O(n) & \text{if } n > 1 \end{cases} \quad T(n) \leq \begin{cases} O(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n > 1 \end{cases}$$

- We can guess $O(n \lg n)$ by looking at recurrence tree and then *verify* it using the substitution method



20.3 Substitution method

1. Guess solution (from recurrence tree)

   This case: $T(n) \leq n \lg n + n$

2. Use induction to find constants and show that solution works

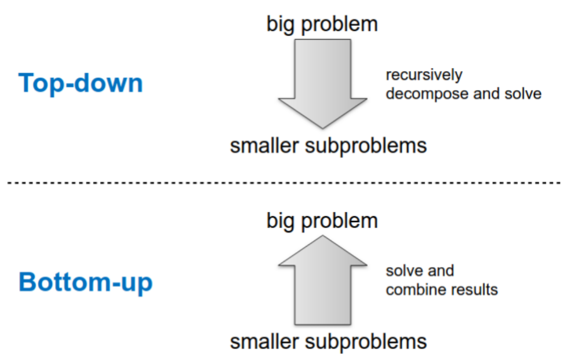
   Base case: $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$

   Inductive Step: Inductive hypothesis is that $T(k) \leq k \lg k + k$ for all $k < n$. We will use this inductive hypothesis for $T(n/2)$

$$\begin{aligned} T(n) &\leq 2T(n/2) + 2 \\ &\leq 2(n/2 \lg n/2 + n/2) + n \quad \text{IH} \\ &= n \lg n/2 + n + n \\ &= n(\lg n - \lg 2) + n + n \\ &= n \lg n - n + n + n \\ &= n \lg n + n \end{aligned}$$

21 Dynamic programming with example of Floyd-Warshall algorithm

- Method for solving a complex problem by breaking it down into a *overlapping* subproblems, solving each of those subproblems just once
- storing their solutions using a memory-based data structure (array, map,etc). Indexed in some way to make lookup easy
- Called memoization
- Needs optimal substructure and overlapping subproblems
- *There are like a million examples in the slides but they are huge and I do not feel like putting them in here*



21.1 Floyd-Warshall algorithm

- Shortest path between all nodes
- No negative cycles are allowed
- Stores the best values found so far

```
Floyd-Warshall(n X n matrix D_0)
  for(k= 1 to n)
    D_k = new n X n matrix
    for(i = 1 to n)
      for(j = 1 to n)
        //whether it is faster to go through k or not
        D_k[i][j]=min(D_k-1[i][j], D_k-1[i][k]+D_k-1[k][j])

   return D_n
```

22 Hashing

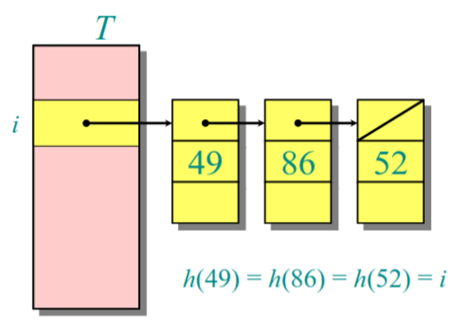
- A symbol table S should hold n records and we will need instructions `insert, delete, search`
- **Direct access table:** Suppose the keys are drawn from a set $U \subseteq \{0, 1, \ldots, m-1\}$ and keys are distinct. Then set up an array $T[0 \ldots m-1]$

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } key[x] = k \\ NIL & \text{otherwise} \end{cases}$$

- Operations take $O(1)$ time then
- But range of keys can be very large
- **Solution:**A *hash function* h to map the universe U of all keys into $\{0, 1, \ldots, m-1\}$
- When a record to be inserted is already mapped to an already occupied slot in T a *collision* occurs

22.1 Chaining

- Link records in the same slot into a list.
- worst case would be $O(n)$ again, if all keys hash to the same slot



- we assume *simple uniform hashing*, so each key $k \in S$ is equally likely to be hashed to any slot of table T, independent of where other keys are hashed
- we define the *load factor* of T to be $\alpha = n/m$ where n is the number of keys in the table and m is the number of slots
- Expected time for a search with a given key is then $O(1 + \alpha)$. It is $O(1)$ for applying the hash function and accessing the slot and $O(\alpha)$ to search the list

22.2 Open addressing

- No storage outside of hash table is used
- Insertion systematically probes the table until an empty slot is found
- Hash function depends on key and probe number
- Table may fill up and deletion is difficult
- Search uses same probe sequence

  Terminates successfully if it finds the key

  Terminates unsuccessful if it encounters an empty slot
- **Linear probing** Given an ordinary hash function $h'(k)$, linear probing uses the hash function

$$h(k, i) = (h'(k) + i) \mod m$$

- Method suffers from primary clustering (long runs of occupied slots build up)
- **Double hashing** Given two ordinary hash function $h_1(k)$ and $h_2(k)$ double hashing uses the function

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

- $h_2(k)$ must be relatively prime to m

22.3 Choosing a hash function

Division method

- Assume all keys are integers adn define $h(k) = k \mod m$
- Problem: If m has a small divisor d, there will be a lot more keys that are congruent modulo d
- Even bigger problem: If $m = 2^r$ then the hash does not depend on all bits of k, thus more collisions
- Often: m is a prime not too close to a power of 2 or 10 (and not otherwise prominently used in computing environment)

Multiplication method

- Assume all keys are integers and $m = 2^r$ and computer has w-bit words. Then define $h(k) = (A \cdot k \mod 2^w) rsh(w - r)$
- rsh is bitwise-right-shift
- A is an odd integer in range $2^{w-1} < A < 2^w$ (but not too close to the borders)
- Multiplication modulo 2^w is faster than division
- rsh operator is fast

23 Prim's algorithm

- computes the minimum spanning tree
- Initialize S with any node
- Repeat $n - 1$ times
  - Add to the tree the min weight edge with one endpoint in S
  - Add new node to S

24 Kruskal's algorithm

- Computes minimum spanning tree
- Consider edges in ascending order of weight
- Add to tree unless it would create a cycle

25 Reverse-delete algorithm

- Computes minimum spanning tree
- Remove edge unless it would disconnect the graph