CS 61b: Final Review

Data Structures

Amir Kamil and Jack Sampson

DISCLAIMER

We have NOT seen the exam.
We do NOT know the format of the exam

What we are presenting is what we
"think is important" for the exam

Review Topics

- Inheritance, Method Calls
- Asymptotic Analysis
- Data Structures
  - Binary Search Trees
  - B-Trees
  - Heaps
  - Hash Tables
  - AVL Trees
- Graphs
  - DFS, BFS
  - Topological Sort
  - Strongly Connected Components
  - Dijkstra
  - Kruskal
- Sorting
- Skip Lists
- Threading, Synchronization
- Scheduling
- Minimax
- B+ Trees
- Threaded Trees

Inheritance/Method Calls

- Given the class definitions on the next slide, which lines in class foobarbaz are illegal?

Inheritance

```
package foo;
public class foo {
   static void f1() {…}
   protected boolean f2(int x) {…}
   private String f3(String s) {…}
}
```

```
package foo;
public class baz extends foo {
   private String f3(String s) {…}
}
```

```
package bar;
import foo.foo;
public class bar extends foo {
   protected boolean f3(int x) {…}
}
```

```
package foo;
import bar.bar;
public class foobarbaz {
   static void main(String[] args) {
      foo f = new foo();
      bar r = new bar();
      baz z;
      r.f3(3);
      f.f2(3);
      z = (baz) f;
      f = new baz();
      f.f2(3);
      z = (baz) f;
      z.f1();
      r.f1();
      ((foo) r).f1();
   }
}
```

Inheritance/Method Calls

- Access table:

	world	package	child	definer
public	X	X	X	X
private				X
protected		X	X	X
<default>		X		X

- Static methods called according to static type
- Child type can be assigned to parent variable without a cast, but the reverse requires one, and the dynamic types must match

Inheritance

```
package foo;
public class foo {
    static void f1() {…}
    protected boolean f2(int x) {…}
    private String f3(String s) {…}
}
```

```
package foo;
public class baz extends foo {
    private String f3(String s) {…}
}
```

```
package bar;
import foo.foo;
public class bar extends foo {
    protected boolean f3(int x) {…}
}
```

```
package foo;
import bar.bar;
public class foobarbaz {
    static void main(String[] args) {
        foo f = new foo();
        bar r = new bar();
        baz z;
        r.f3(3);               // ILLEGAL
        f.f2(3);
        z = (baz) f;           // ILLEGAL
        f = new baz();
        f.f2(3);
        z = (baz) f;
        z.f1();
        r.f1();               // ILLEGAL
        ((foo) r).f1();
    }
}
```

Asymptotic Analysis

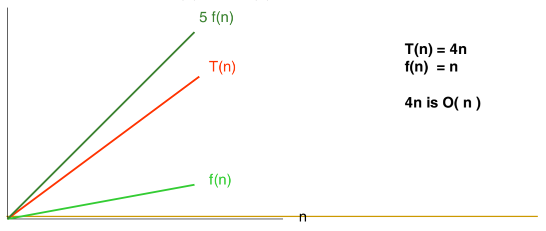
- O – Upper bound/Worst case
- Ω – Lower bound
- Θ – both
- o – strictly Upper bound

More detail…

Asymptotic Analysis

T(n) is O(f(n)) if and only if there exists positive constants C and N such that

<u>T(n) <= C f(n)</u> for all n >= N



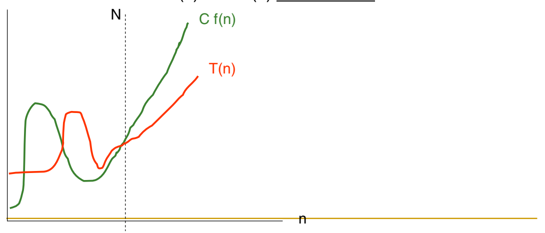
T(n) = 4n
f(n) = n

4n is O(n)

Asymptotic Analysis

T(n) is O(f(n)) if and only if there exists positive constants C and N such that

T(n) <= C f(n) <u>for all n >= N</u>



Asymptotic Analysis

T(n) is O(f(n)) if and only if there exists positive constants C and N such that

T(n) <= C f(n) for all n >= N

T(n) is Ω(f(n)) if and only if there exists positive constants C and N such that

T(n) >= C f(n) for all n >= N

Asymptotic Analysis

T(n) is Θ(f(n)) if and only if

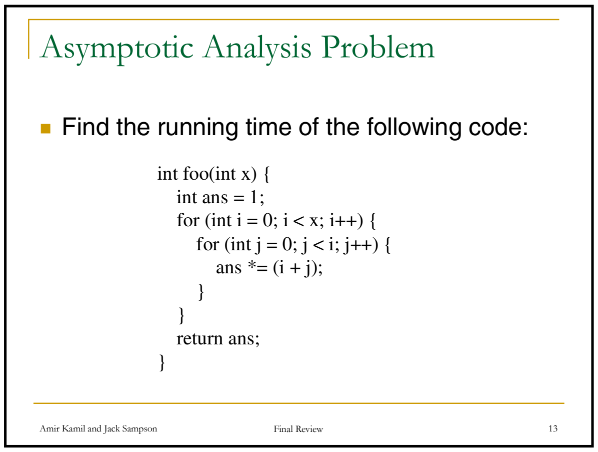
- T(n) is O(f(n))

  and
- T(n) is Ω(f(n))

Examples

$5n^2+1$ is $\Theta(n^2)$

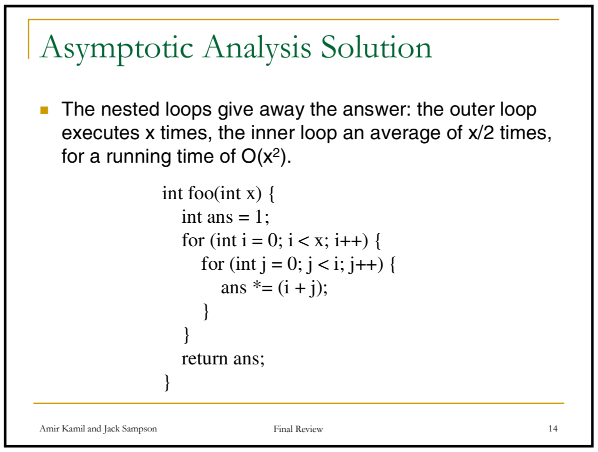
3n is $O(n^2)$, but 3n is NOT $\Theta(n^2)$ because 3n is not $\Omega(n^2)$



Asymptotic Analysis Problem

- Find the running time of the following code:

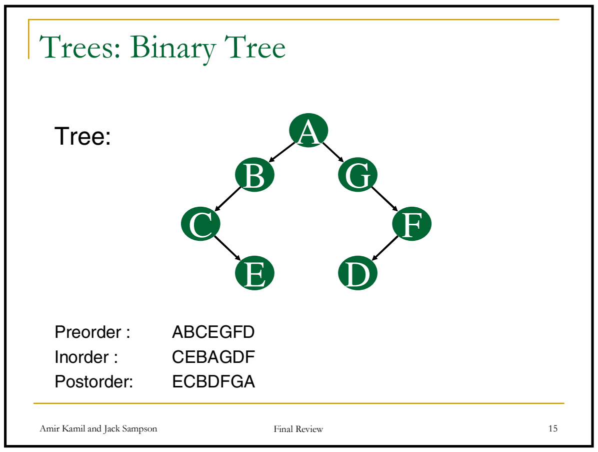
```
int foo(int x) {
    int ans = 1;
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < i; j++) {
            ans *= (i + j);
        }
    }
    return ans;
}
```



Asymptotic Analysis Solution

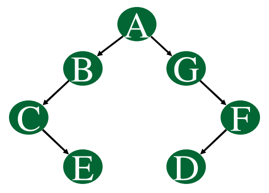
- The nested loops give away the answer: the outer loop executes x times, the inner loop an average of x/2 times, for a running time of O(x^2).

```
int foo(int x) {
    int ans = 1;
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < i; j++) {
            ans *= (i + j);
        }
    }
    return ans;
}
```

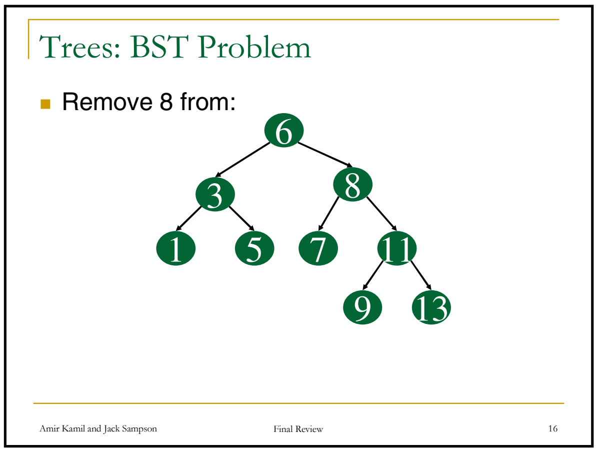


Trees: Binary Tree

Tree:

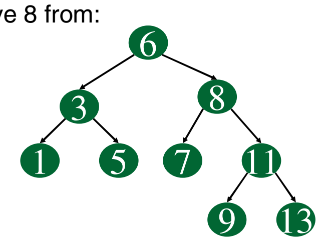


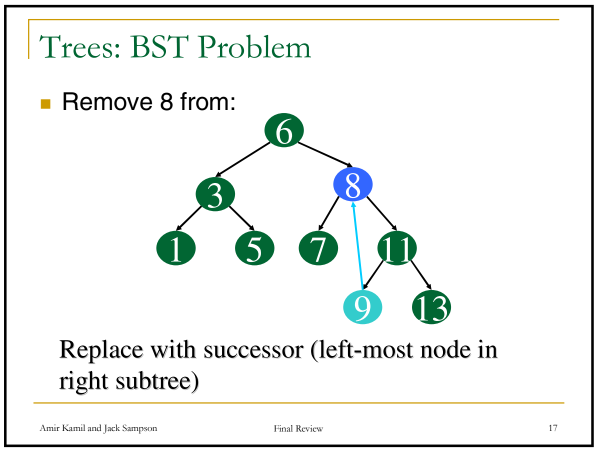
Preorder :	ABCEGFD
Inorder :	CEBAGDF
Postorder:	ECBDFGA



Trees: BST Problem

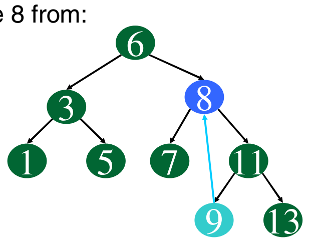
- Remove 8 from:



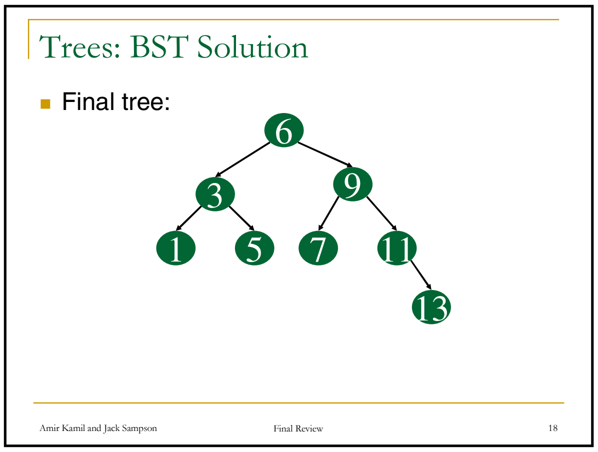


Trees: BST Problem

- Remove 8 from:

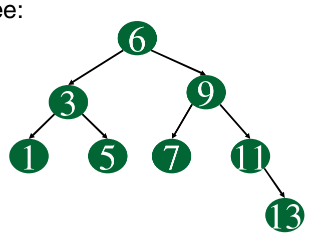


Replace with successor (left-most node in right subtree)



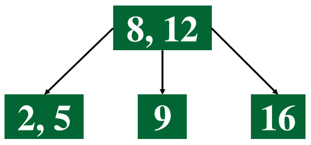
Trees: BST Solution

- Final tree:



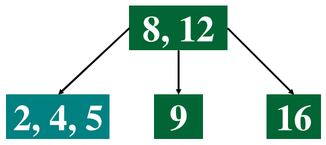
Trees: B-Tree of Order 4 / 2-3-4 Tree

- Insert 4 and 6 into the following 2-3-4 tree



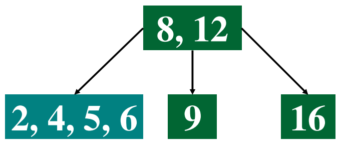
Trees: B-Tree of Order 4 / 2-3-4 Tree

- Insert 4



Trees: B-Tree of Order 4 / 2-3-4 Tree

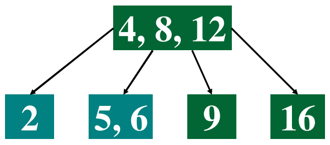
- Insert 6



Overflow, so split node and promote middle element

Trees: B-Tree of Order 4 / 2-3-4 Tree

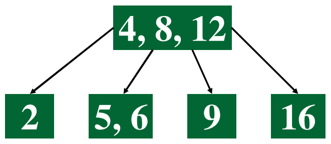
- Insert 6



Overflow, so split node and promote middle element

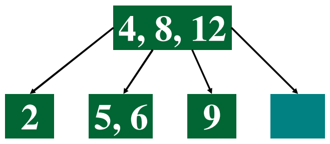
Trees: B-Tree of Order 4 / 2-3-4 Tree

- Remove 16 from the following 2-3-4 tree



Trees: B-Tree of Order 4 / 2-3-4 Tree

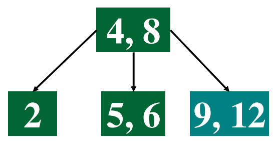
- Remove 16



Underflow, so merge with sibling and demote parent element

Trees: B-Tree of Order 4 / 2-3-4 Tree

- Remove 16



Underflow, so merge with sibling
and demote parent element

Priority Queues – Problem

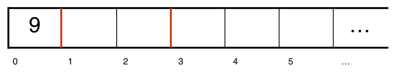
- Add 9, 76, 54, 3, 33, 21 to a max heap, using only the array based representation

Priority Queues – Insertion

- Insert at the last position in the heap
- Reheapify up: if the element is greater than its parent, swap them and repeat
- For an element at position n, its children are at 2n+1 and 2n+2
- For an element at position n, its parent is at floor[(n-1)/2]

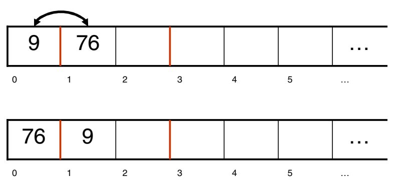
Priority Queues – Solution

- Add 9, 76, 54, 3, 33, 21 to a max heap, using only the array based representation



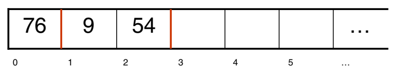
Priority Queues – Solution

- Add 9, 76, 54, 3, 33, 21 to a max heap, using only the array based representation



Priority Queues – Solution

- Add 9, 76, 54, 3, 33, 21 to a max heap, using only the array based representation



Priority Queues – Solution

- Add 9, 76, 54, 3, 33, 21 to a max heap, using only the array based representation

76	9	54	3			…
0	1	2	3	4	5	…

Priority Queues – Solution

- Add 9, 76, 54, 3, 33, 21 to a max heap, using only the array based representation

76	9	54	3	33		…
0	1	2	3	4	5	…

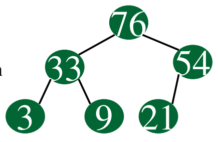
76	33	54	3	9		…
0	1	2	3	4	5	…

Priority Queues – Solution

- Add 9, 76, 54, 3, 33, 21 to a max heap, using only the array based representation

76	33	54	3	9	21	…
0	1	2	3	4	5	…

Tree Representation



Priority Queues – Problem

- Remove the max from the heap

76	33	54	3	9	21	…
0	1	2	3	4	5	…

Priority Queues – Removal

- Replace the max element with the last element in the heap
- Reheapify down: if one or both of its children is larger than it, swap with the larger of the children and repeat
- For an element at position n, its children are at 2n+1 and 2n+2
- For an element at position n, its parent is at floor[(n-1)/2]

Priority Queues – Solution

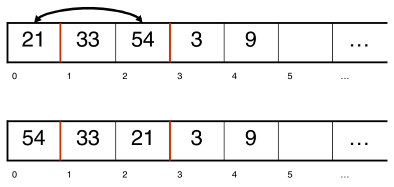
- Remove the max from the heap

76	33	54	3	9	21	…
0	1	2	3	4	5	…

21	33	54	3	9		…
0	1	2	3	4	5	…

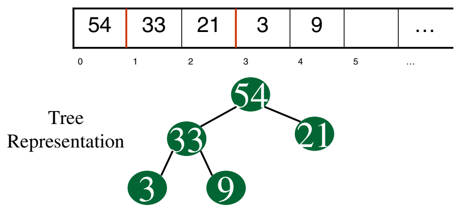
Priority Queues – Solution

- Remove the max from the heap



Priority Queues – Solution

- Remove the max from the heap



Hash Table Problem

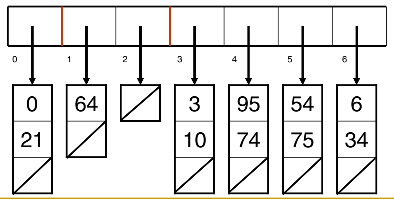
- Draw the structure of a size 7 hash table after insertion of keys with the following hash codes: 0, 95, 21, 6, 64, 74, 3, 54, 34, 75, 10.

Hash Tables

- High-level idea – 2 components
  1. Big array called *hash table* of size M
  2. Function *h* which maps keys to integer values
- For (key, item), use h(key) % M to find location of item in table
- Linked list in each entry that stores all items that map to that location (chaining)

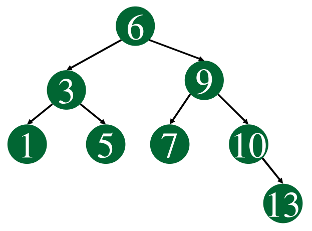
Hash Table Solution

- Draw the structure of a size 7 hash table after insertion of keys with the following hash codes: 0, 95, 21, 6, 64, 74, 3, 54, 34, 75, 10.



AVL Tree Problem

- Given the following AVL Tree, performs these consecutive operations and draw out the tree in each step:
  - Remove(7)
  - Insert (11)
  - Insert(12)



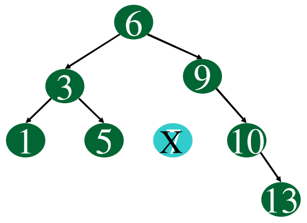
AVL Trees

- AVL Trees are just Binary Search Trees that can rotate their nodes to try to maintain balance.
  - Two kinds of rotations – single and double
  - Can decide which to do based on structure of tree

Insertions/Removals

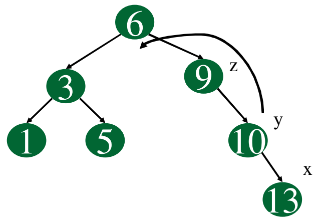
- You have 3 nodes of importance, which we will call x, y, and z (z is the parent of y which is the parent of x)
  - If x is the right child of y, and y is the right child of z, you do a single rotation (same goes for left child of left child)
  - If x is the right child of y, and y is the left child of z, you do a double rotation (same goes for left child of right child)

Remove(7)



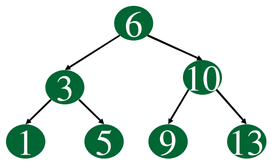
Remove 7 as in BST

Remove(7)



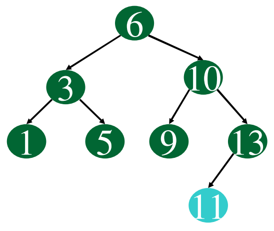
Single rotate

Remove(7)



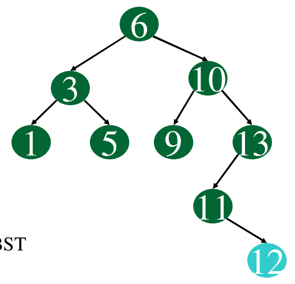
Final tree

Insert(11)



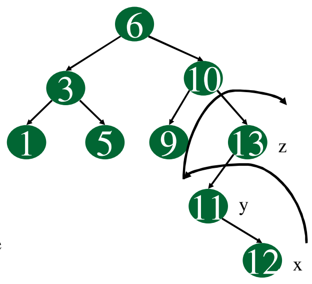
Insert as in BST

Insert(12)



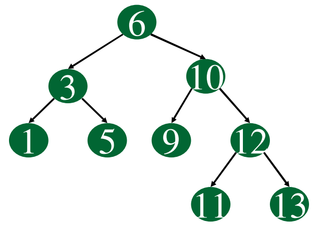
Insert as in BST

Insert(12)



Double rotate

Insert(12)



Final tree

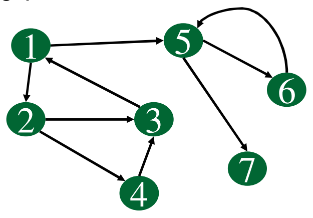
Searches (BFS and DFS)

- BFS uses a queue, DFS uses a stack

```
public void BFS/DFS(Node start) {
    Queue/Stack s = new Queue/Stack();
    s.enqueue/push(start);
    while (!s.empty()) {
        Node n = s.dequeue/pop();
        mark(n);
        for (all children that are not yet marked) {
            s.enqueue/push(child);
        }
    }
}
```

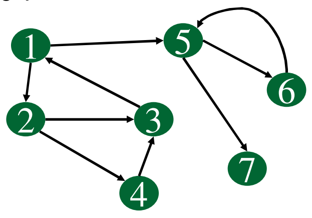
Searches (BFS and DFS) Problem

- Perform BFS and DFS on the graph, starting at node 1



Searches (BFS and DFS) Solution

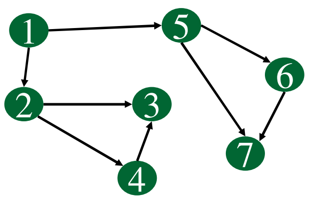
- Perform BFS and DFS on the graph, starting at node 1



BFS	DFS
1	1
2	2
5	3
3	4
4	5
6	6
7	7

Topological Sort Problem

- Perform a topological sort on the graph

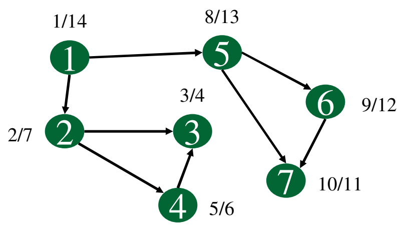


Topological Sort

- Perform DFS, computing start/finish times
- Order nodes by decreasing finish times

Topological Sort Solution

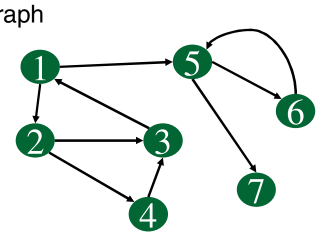
- Perform a topological sort on the graph



1. 1
2. 5
3. 6
4. 7
5. 2
6. 4
7. 3

SCC Problem

- Find the strongly connected components of the graph

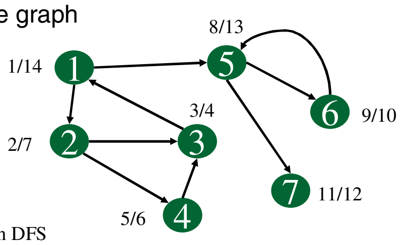


SCC Algorithm

- Perform DFS, computing start/finish times
- Invert graph
- Repeatedly run DFS on the remaining node with the highest finishing time
- The nodes marked in each DFS run compose a strongly connected component

SCC Solution

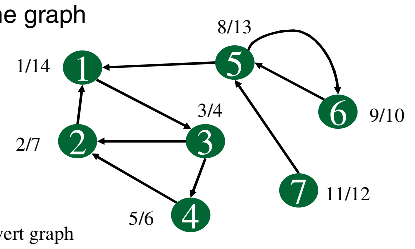
- Find the strongly connected components of the graph



Run DFS

SCC Solution

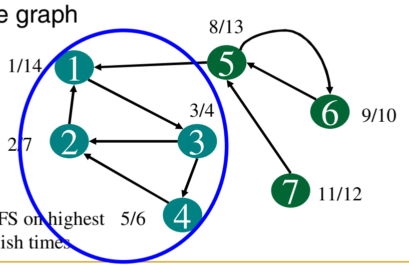
- Find the strongly connected components of the graph



Invert graph

SCC Solution

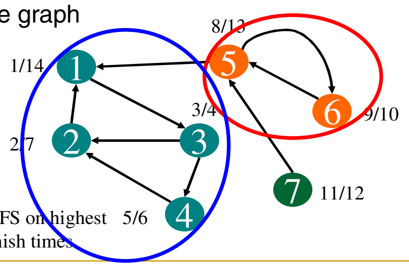
- Find the strongly connected components of the graph



Run DFS on highest finish times

SCC Solution

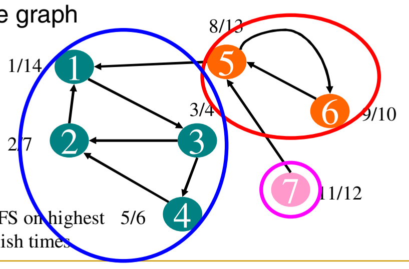
- Find the strongly connected components of the graph



Run DFS on highest finish times

SCC Solution

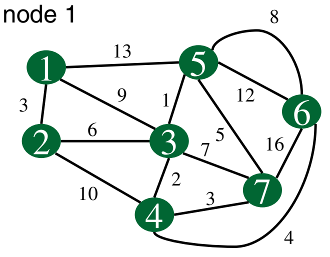
- Find the strongly connected components of the graph



Run DFS on highest finish times

Dijkstra's Algorithm Problem

- Find the shortest distances to each node from node 1

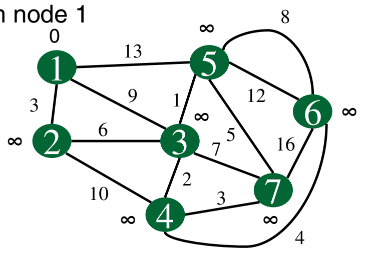


Dijkstra's Algorithm

- Set all distances initially to ∞, except the start node, which should be set to 0
- Construct a min priority queue of the nodes, with their distances as keys
- Repeatedly remove the minimum element, updating each of its adjacent node's distances if they are still in the queue and if the updated distance is less than the current distance

Dijkstra's Algorithm Solution

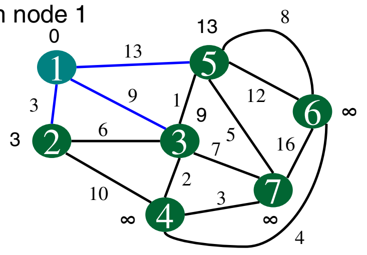
- Find the shortest distances to each node from node 1



P.Q.
1 (0)
2 (∞)
3 (∞)
4 (∞)
5 (∞)
6 (∞)
7 (∞)

Dijkstra's Algorithm Solution

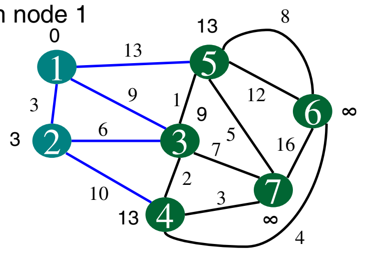
- Find the shortest distances to each node from node 1



P.Q.
x1 (0)
2 (3)
3 (9)
5 (13)
4 (∞)
6 (∞)
7 (∞)

Dijkstra's Algorithm Solution

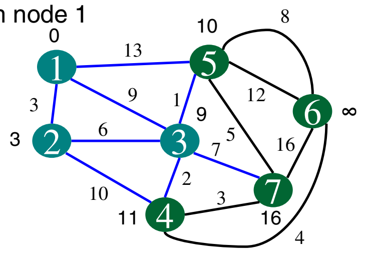
- Find the shortest distances to each node from node 1



P.Q.
x1 (0)
x2 (3)
3 (9)
5 (13)
4 (13)
6 (∞)
7 (∞)

Dijkstra's Algorithm Solution

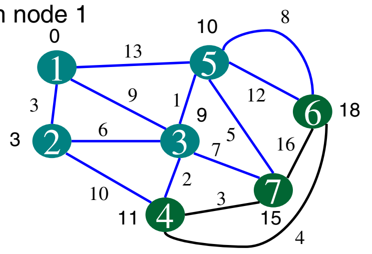
- Find the shortest distances to each node from node 1



P.Q.
x1 (0)
x2 (3)
x3 (9)
5 (10)
4 (11)
7 (16)
6 (∞)

Dijkstra's Algorithm Solution

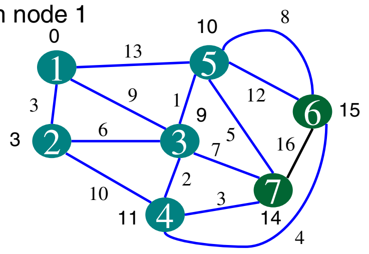
- Find the shortest distances to each node from node 1



P.Q.
x1 (0)
x2 (3)
x3 (9)
x5 (10)
4 (11)
7 (15)
6 (18)

Dijkstra's Algorithm Solution

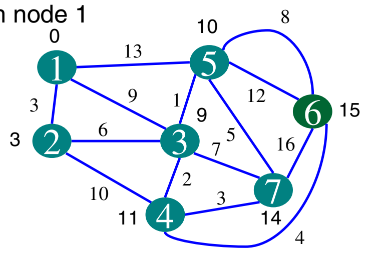
- Find the shortest distances to each node from node 1



P.Q.
x1 (0)
x2 (3)
x3 (9)
x5 (10)
x4 (11)
7 (14)
6 (15)

Dijkstra's Algorithm Solution

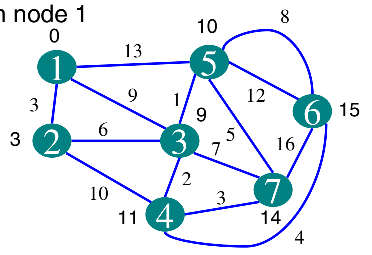
- Find the shortest distances to each node from node 1



P.Q.
x1 (0)
x2 (3)
x3 (9)
x5 (10)
x4 (11)
x7 (14)
6 (15)

Dijkstra's Algorithm Solution

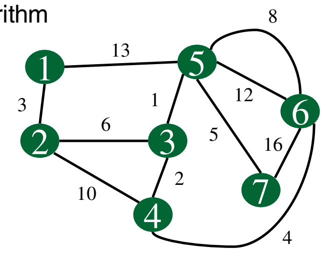
- Find the shortest distances to each node from node 1



P.Q.
x1 (0)
x2 (3)
x3 (9)
x5 (10)
x4 (11)
x7 (14)
x6 (15)

Kruskal's Algorithm Problem

- Find the MST of the graph, using Kruskal's Algorithm

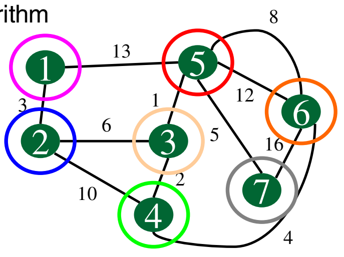


Kruskal's Algorithm

- Put each node into a set by itself
- Sort all the edges in ascending order by their weights
- Pick the least-weight edge, if the edge connects two nodes in different sets, add the edge to the MST and merge the two sets

Kruskal's Algorithm Solution

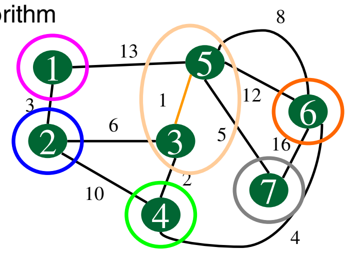
- Find the MST of the graph, using Kruskal's Algorithm



Edges
3-5 (1)
3-4 (2)
1-2 (3)
4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)

Kruskal's Algorithm Solution

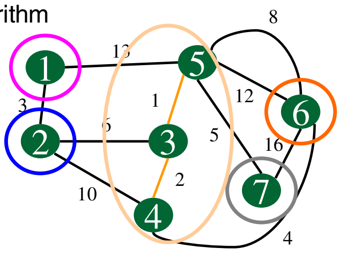
- Find the MST of the graph, using Kruskal's Algorithm



Edges
x3-5 (1)
3-4 (2)
1-2 (3)
4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)

Kruskal's Algorithm Solution

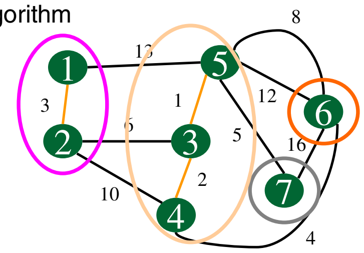
- Find the MST of the graph, using Kruskal's Algorithm



Edges
x3-5 (1)
x3-4 (2)
1-2 (3)
4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)

Kruskal's Algorithm Solution

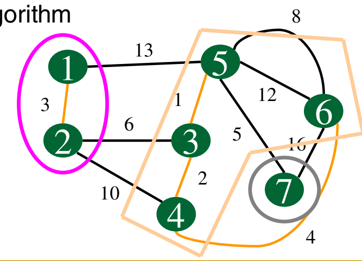
- Find the MST of the graph, using Kruskal's Algorithm



Edges
x3-5 (1)
x3-4 (2)
x1-2 (3)
4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)

Kruskal's Algorithm Solution

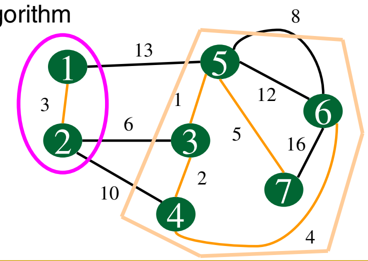
- Find the MST of the graph, using Kruskal's Algorithm



Edges
x3-5 (1)
x3-4 (2)
x1-2 (3)
x4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)

Kruskal's Algorithm Solution

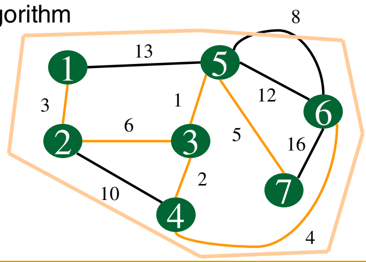
- Find the MST of the graph, using Kruskal's Algorithm



Edges
x3-5 (1)
x3-4 (2)
x1-2 (3)
x4-6 (4)
x5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)

Kruskal's Algorithm Solution

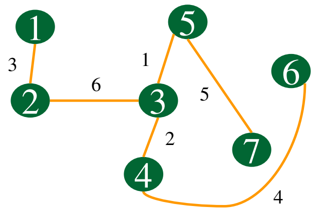
- Find the MST of the graph, using Kruskal's Algorithm



Edges
x3-5 (1)
x3-4 (2)
x1-2 (3)
x4-6 (4)
x5-7 (5)
x2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)

Kruskal's Algorithm Solution

- Find the MST of the graph, using Kruskal's Algorithm



Sorting

- Given the following steps, which sorting algorithms were used in each case?

```
13 27 89 26 9 37 5 1 38        13 27 89 26 9 37 5 1 38
1 27 89 26 9 37 5 13 38        13 27 26 9 37 5 1 38 89
1 5 89 26 9 37 27 13 38        1 13 27 26 9 37 5 38 89
1 5 9 26 89 37 27 13 38        1 5 13 27 26 9 37 38 89
1 5 9 13 89 37 27 26 38        1 5 13 27 26 9 37 38 89
1 5 9 13 26 37 27 89 38        1 5 9 13 27 26 37 38 89
1 5 9 13 26 27 37 89 38        1 5 9 13 26 27 37 38 89
1 5 9 13 26 27 37 89 38
1 5 9 13 26 27 37 38 89
```

Sorting

```
Selection Sort                 Quick Sort

13 27 89 26 9 37 5 1 38        13 27 89 26 9 37 5 1 38
1 27 89 26 9 37 5 13 38        13 27 26 9 37 5 1 38 89
1 5 89 26 9 37 27 13 38        1 13 27 26 9 37 5 38 89
1 5 9 26 89 37 27 13 38        1 5 13 27 26 9 37 38 89
1 5 9 13 89 37 27 26 38        1 5 13 27 26 9 37 38 89
1 5 9 13 26 37 27 89 38        1 5 9 13 27 26 37 38 89
1 5 9 13 26 27 37 89 38        1 5 9 13 26 27 37 38 89
1 5 9 13 26 27 37 89 38
1 5 9 13 26 27 37 38 89
```

Sorting

- Do a radix sort on the following sequence, showing each step

(1087 643 2532 954 8174 65 340 1752)

Sorting

- Step 1: sort by ones place

(1087 643 2532 954 8174 65 340 1752)

$\downarrow$

(340 2532 1752 643 954 8174 65 1087)

Sorting

- Step 2: sort by tens place

(340 2532 1752 643 954 8174 65 1087)

$\downarrow$

(2532 340 643 1752 954 65 8174 1087)

Sorting

- Step 3: sort by hundreds place

(2532 340 643 1752 954 65 8174 1087)

$\downarrow$

(65 1087 8174 340 2532 643 1752 954)

Sorting

- Step 4: sort by thousands place

(65 1087 8174 340 2532 643 1752 954)

$\downarrow$

(65 340 643 954 1087 1752 2532 8174)

Skip List Problem

- Write code for searching a skip list for a key. Assume a skip list node is defined as

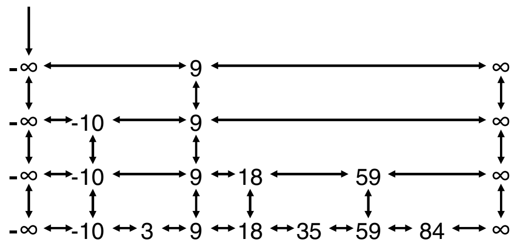
```
class Node {
   Comparable key;
   Node left, right, up, down;
}
```

and that the skip list pointer references the top left node.

Skip Lists

- 2D linked lists
- Bottom level contains all keys, and each subsequent level contains probabilistically half the keys of the previous level
- Each level starts at -∞ and ends at +∞
- The keys in each level are in ascending order

Skip List Example



Skip List Searching

- Start at top left node
- If the current key is equal to the search key, return the node
- If the next key is greater than the search key, go down and repeat search
- Otherwise go right and repeat search

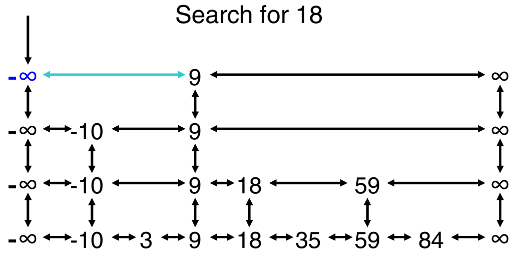
Skip List Solution

- Write code for searching a skip list for a key

```
Node search(Node n, Comparable key) {
   if (n.key.equals(key)) {
      return n;
   } else if (n.next.key.compareTo(key) > 0) {
      return search(n.down, key);
   } else {
      return search(n.next, key);
   }
}
```

Skip List Searching

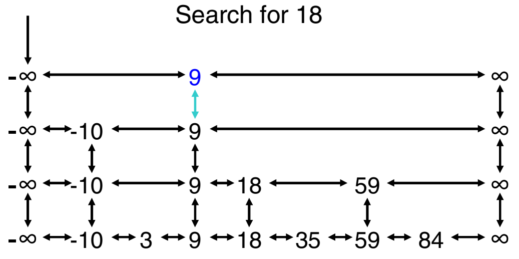
Search for 18



9 is not greater than 18, so move right

Skip List Searching

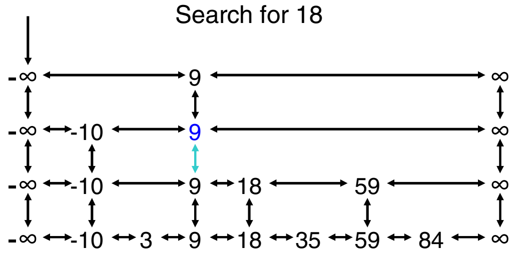
Search for 18



∞ is greater than 18, so move down

Skip List Searching

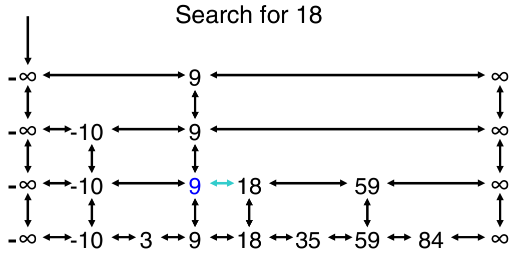
Search for 18



∞ is greater than 18, so move down

Skip List Searching

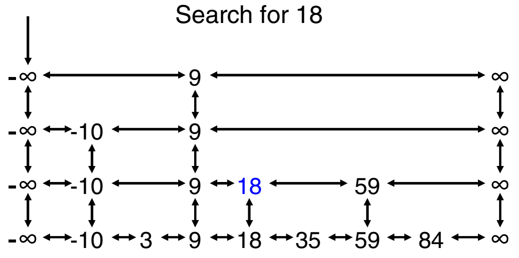
Search for 18



18 is not greater than 18, so move right

Skip List Searching

Search for 18



18 is equal to 18, so return node

Threading

- Motivations:
  - Modeling of simultaneous actions
  - Counteract I/O Latency
- Mechanism: Multiple threads of control
  - Shared memory space, multiple program counters
- Dangers:
  - Shared access to memory can result in conflicts
  - Multiple threads per processor can result in unequal time sharing (see scheduling)
- Conflict types:
  - WAR (write after read)
  - WAW (write after write)
  - RAW (read after write)
- How to avoid shared data conflicts? Locking
- Dangers of locking? Deadlock

Scheduling

- **Throughput** – Average number of tasks completed per unit time
- **CPU Utilization** – Average usage of the processor
- **Wait time** – time spent waiting for processor
- **Turnaround time** – time from task assignment to task completion
- **Response time** – time between assignment of task and first work on task
- **Large values => GOOD:**
  - throughput
  - cpu utilization
- **Large values => BAD (maybe):**
  - wait time
  - turnaround time
  - response time
- **I/O ?**

The Min-Max Algorithm

- An algorithm for making the best possible move in a **ZERO-SUM-GAME** (not applicable to other types of games)

```
MinMax( State, maxtype)
  if gameover(State) return [null move, score(State)]
  if (maxtype)
    return pair with max score from
      for each valid move from State MinMax(NewState, !
  maxtype)
  else
    return pair with min score from
      for each valid move from State MinMax(NewState, !
  maxtype)
```

- Justification:
  - In a zero-sum-game, the best move for an opponent is to minimize your score, just as your best move is to maximize your score. This will therefore return the best possible move under the assumption that one's opponent plays perfectly.

The Min-Max Algorithm

- The following is an implementation of Min-Max in Common Lisp:

```
;;; The minimax decision procedure returns the optimal move in the game
;;; using exhaustive generation of the entire game tree.  Implementation
;;; uses the fact that the evaluation and utility functions return a list of
;;; values from the point of view of each player, with the "current" player
;;; first. Hence, rather than using #'min, we always use #'max for the
;;; current player.  A successor value is passed up the tree using
;;; right-rotation.  This works for any number of players.
;;; The notation "a+s" means an (action . state) pair.

(defun minimax-decision (state game)
  (car (the-biggest
        #'(lambda (a+s) (first (right-rotate (minimax-value (cdr a+s) game))))
        (game-successors state game))))

(defun minimax-value (state game)
  (if (game-over? game state)
      (terminal-values state)
      (right-rotate
       (the-biggest
        #'(lambda (values) (first (right-rotate values)))
        (mapcar #'(lambda (a+s) (minimax-value (cdr a+s) game))
                (game-successors state game))))))
```

Min-Max with cutoff

```
(defun minimax-cutoff-decision (state game eval-fn limit)
  "Return the best action, according to backed-up evaluation down to LIMIT.
  After we search LIMIT levels seep, we use EVAL-FN to provide an estimate
  of the true value of a state; thus the action may not actually be best."
  (car (the-biggest
        #'(lambda (a+s)
            (first (right-rotate
                    (minimax-cutoff-value (cdr a+s) game eval-fn (- limit 1)))))
        (game-successors state game))))

(defun minimax-cutoff-value (state game eval-fn limit)
  (cond ((game-over? game state) (terminal-values state))
        ((<= limit 0) (funcall eval-fn state))
        (t (right-rotate
            (the-biggest
             #'(lambda (values) (first (right-rotate values)))
             (mapcar #'(lambda (a+s)
                                   (minimax-cutoff-value (cdr a+s) game eval-fn
                                                         (- limit 1)))
                      (game-successors state game)))))))
```

Min-Max with cutoff

```
(defun game-successors (state game)
  "Return a list of (move . state) pairs that can be reached from this state."
  (mapcar #'(lambda (move) (cons move (make-move game state move)))
          (legal-moves game state)))

(defun terminal-values (state)
  "Return the values of the state for each player."
  (mapcar #'(lambda (player) (getf (game-state-scores state) player))
          (game-state-players state)))
```

alpha-beta pruning

```
(defun alpha-beta-decision (state game eval-fn &optional (limit 4))
  "Return the estimated best action, searching up to LIMIT and then
  applying the EVAL-FN."
  (car (the-biggest
        #'(lambda (a+s)
            (first (right-rotate
                    (alpha-value (cdr a+s) game
                                                  (game-worst game) (game-worst game)
                                 eval-fn (- limit 1)))))
        (game-successors state game))))

(defun alpha-value (state game alpha beta eval-fn limit)
  (cond ((game-over? game state) (terminal-values state))
        ((= 0 limit) (funcall eval-fn state))
        (t (dolist (a+s (game-successors state game)
                              (list alpha (- alpha)))
             (setq alpha (max alpha
                                  (first (right-rotate
                                          (beta-value (cdr a+s) game alpha beta
                                                      eval-fn (- limit 1))))))
             (when (>= alpha (- beta))
               (return (list (- beta) beta)))))))
```

alpha-beta pruning

```
(defun beta-value (state game alpha beta eval-fn limit)
 (cond ((game-over? game state) (terminal-values state))
     ((= 0 limit) (funcall eval-fn state))
     (t (dolist (a+s (game-successors state game)
                                  (list beta (- beta)))
        (setq beta (max beta
                                  (first (right-rotate
                                              (alpha-value (cdr a+s) game alpha beta
                                                                   eval-fn (- limit 1))))))
        (when (>= beta (- alpha))
          (return (list (- alpha) alpha)))))))
```

Cool tree variants

- The threaded tree:
  - Motivations:
    - Inorder traversals are common
    - Naive BST implementation can waste space (~half of all child pointers are null)
  - Mechanism:
    - Add boolean flag to pointers (or do fun polymorphism) so as to have leaf nodes point to the next node in an inorder traversal
  - Results:
    - For a minimal change in the space requirements and structure of a tree, inorder traversals can now be computed using a straightforward iterative algorithm

Cool tree variants continued

- The B+ tree:
  - Motivations:
    - Range queries are common
    - size of Data >> size of Key, so treat differently
  - Mechanism:
    - Start with B-tree
    - Differentiate between Leaf and index nodes. Index nodes hold keys, leaf nodes hold data. Key values for all data are in leaf nodes.
    - Insert and delete as before, except keys are copied up on split, not moved, and keys may remain on delete for data that no longer exists
    - Add next and previous fields to all leaf nodes, forming a doubly linked list
  - Results:
    - Range query now straightforward to return result for - tree now optimized for contiguous storage on physical media

Credits

- Thanks to CS 61b staff of
  - Fall 2001
  - Spring 2002
- Thanks to Steve Sinha and Winston Liaw
- Thanks to
  - CMU – MIT
  - Cornell – Johns Hopkins U

for slide and example ideas

GOOD LUCK!

(and may you not need it)