Backtracking

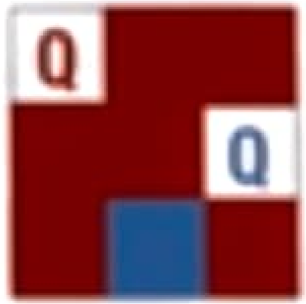
- Systematically search for a solution
- Build the solution one step at a time
- If we hit a dead-end
    - Undo the last step
    - Try the next option

Eight queens

- Place 8 queens on a chess board so that none of them attack each other
- In chess, a queen can move any number of squares along a row column or diagonal

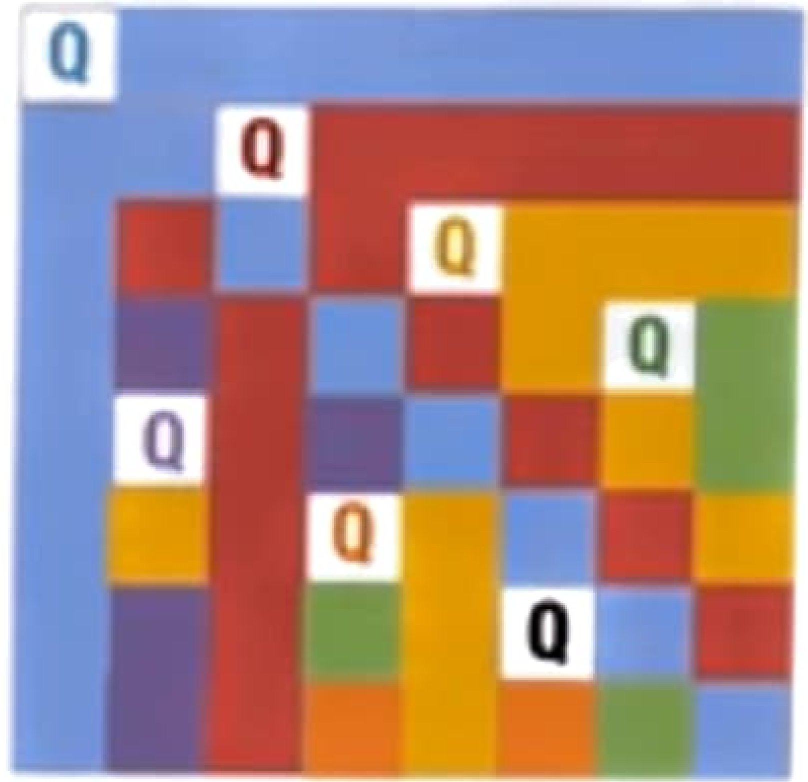
N queens

- Place N queens on an N x N chess board so that none attack each other
- N = 2, 3 impossible
- N = 4 is possible
- And all bigger N as well



8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column
- Can't place a queen in the 8th row!



Backtracking

- Keep trying to extend the next solution
- If we cannot, undo previous move and try again
- Exhaustively search through all possibilities
- … but systematically!

Coding the solution

- How do we represent the board?
- n x n grid, number rows and columns from `0` to `n-1`
    - `board[i][j] == 1` indicates queen at `(i,j)`
    - `board[i][j] == 0` indicates no queen
- We know there is only one queen per row
- Single list `board` of length `n` with entries `0` to `n-1`
    - `board[i] == j` : queen in row `i`, column `j`, i.e. `(i,j)`

Overall structure

```
def placequeen(i,board): # Trying row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      return(True) # Last queen has been placed
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True) # This solution extends fully
    else:
      undo this move and update board
  else:
    return(False) # Row i failed
```

Backtracking

- Systematically search for a solution
- Build the solution one step at a time
- If we hit a dead-end
  - Undo the last step
  - Try the next option

Coding the solution

- How do we represent the board?
- n x n grid, number rows and columns from `0` to `n-1`
  - `board[i][j] == 1` indicates queen at `(i,j)`
  - `board[i][j] == 0` indicates no queen
- We know there is only one queen per row
- Single list `board` of length `n` with entries `0` to `n-1`
  - `board[i] == j` : queen in row `i`, column `j`, i.e. `(i,j)`

Overall structure

```
def placequeen(i,board): # Trying row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      return(True) # Last queen has been placed
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True) # This solution extends fully
    else:
      undo this move and update board
  else:
    return(False) # Row i failed
```

Updating the board

- Our 1-D and 2-D representations keep track of the queens
- Need an efficient way to compute which squares are free to place the next queen
- n x n `attack` grid
  - `attack[i][j] == 1` if `(i,j)` is attacked by a queen
  - `attack[i][j] == 0` if `(i,j)` is currently available
- How do we undo the effect of placing a queen?

Updating the board

- Our 1-D and 2-D representations keep track of the queens
- Need an efficient way to compute which squares are free to place the next queen
- n x n `attack` grid
  - `attack[i][j] == 1` if `(i,j)` is attacked by a queen
  - `attack[i][j] == 0` if `(i,j)` is currently available
- How do we undo the effect of placing a queen?
  - Which `attack[i][j]` should be reset to `0`?

Updating the board

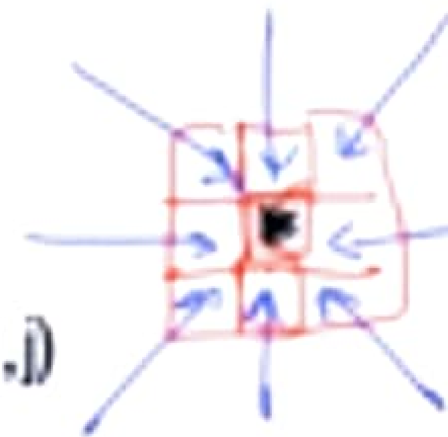
- Queens are added row by row
- Number the queens 0 to n-1
- Record earliest queen that attacks each square
    - `attack[i][j] == k` if `(i,j)` was first attacked by queen `k`
    - `attack[i][j] == -1` if `(i,j)` is free
- Remove queen `k` — reset `attack[i][j] == k` to `-1`
    - All other squares still attacked by earlier queens

Updating the board

- Queens are added row by row
- Number the queens 0 to n-1
- Record earliest queen that attacks each square
  - `attack[i][j] == k` if `(i,j)` was first attacked by queen `k`
  - `attack[i][j] == -1` if `(i,j)` is free
- Remove queen `k` — reset `attack[i][j] == k` to -1
  - All other squares still attacked by earlier queens

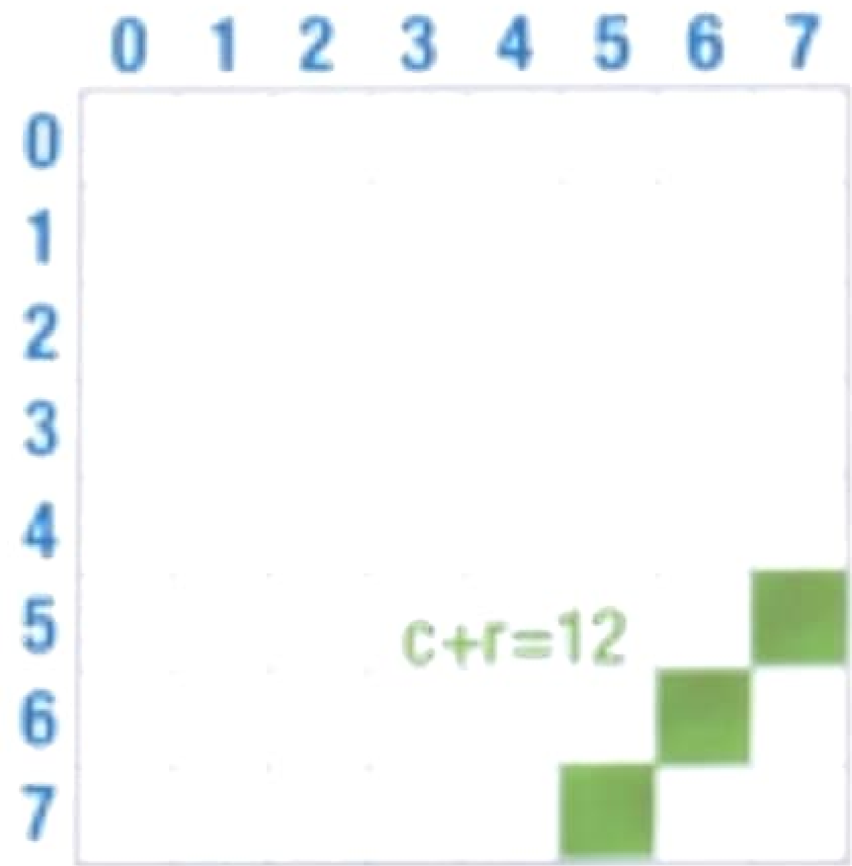
A better representation

- How many queens attack row i?
- How many queens attack row j?
- An individual square (i,j) is attacked by upto 4 queens
    - Queen on row i and on column j
    - One queen on each diagonal through (i,j)



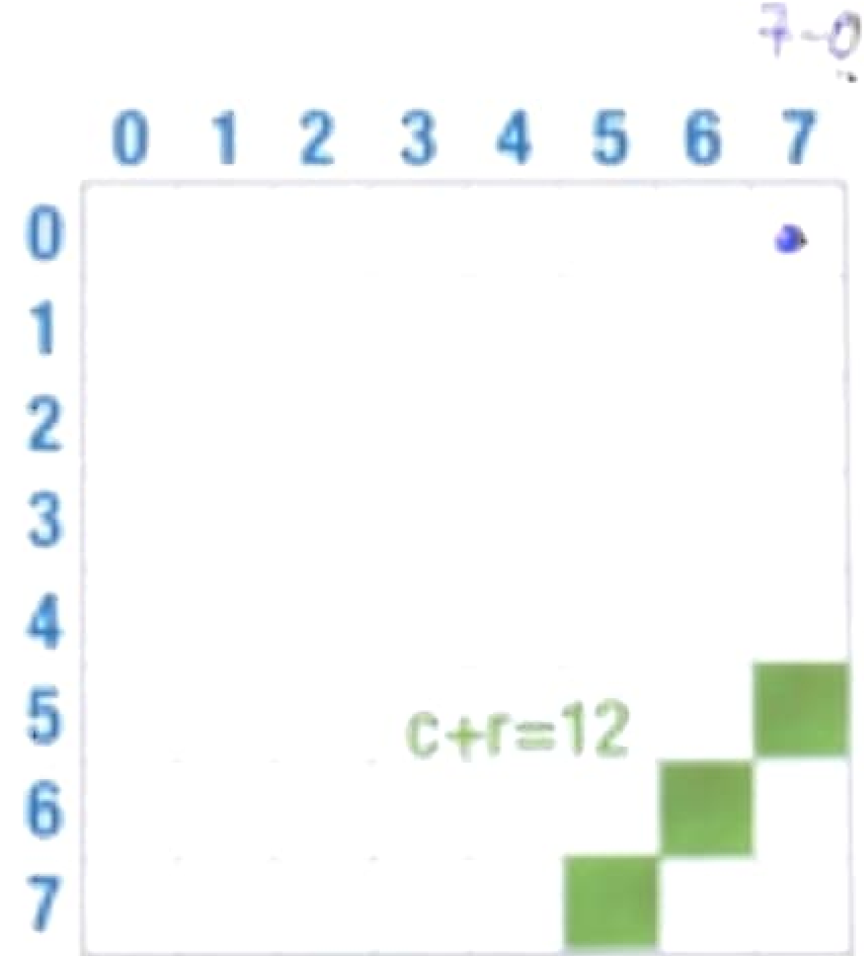
Numbering diagonals

- Decreasing diagonal: column - row is invariant
- Increasing diagonal: column + row is invariant
- (i,j) is attacked if
  - row i is attacked
  - column j is attacked
  - diagonal j-i is attacked
  - diagonal j+i is attacked



Numbering diagonals

- Decreasing diagonal: column - row is invariant
- Increasing diagonal: column + row is invariant
- (i,j) is attacked if
  - row i is attacked
  - column j is attacked
  - diagonal j-i is attacked
  - diagonal j+i is attacked



O(n) representation

- `row[i] == 1` if row i is attacked, `0..N-1`
- `col[i] == 1` if column i is attacked, `0..N-1`
- `NWtoSE[i] == 1` if NW to SE diagonal `i` is attacked, `-(N-1) to (N-1)`

Updating the board

- `(i,j)` is free if

  `row[i]==col[j]==NWtoSE[j-i]==SWtoNE[j+i]==0`

- Add queen at `(i,j)`

```
board[i] = j
(row[i],col[j],NWtoSE[j-i],SWtoNE[j+i]) =
                                   (1,1,1,1)
```

- Remove queen at `(i,j)`

```
board[i] = -1
(row[i],col[j],NWtoSE[j-i],SWtoNE[j+i]) =
                                   (0,0,0,0)
```

Overall structure

```
def placequeen(i,board: # Trying row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      return(True) # Last queen has been placed
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True) # This solution extends fully
    else:
      undo this move and update board
  else:
    return(False) # Row i failed
```

Implementation details

- Maintain `board` as nested dictionary
  - `board['queen'][i] = j` : Queen located at `(i,j)`
  - `board['row'][i] = 1` : Row `i` attacked
  - `board['col'][i] = 1` : Column `i` attacked
  - `board['nwtose'][i] = 1` : NWtoSW diagonal `i` attacked
  - `board['swtone'][i] = 1` : SWtoNE diagonal `i` attacked

Updating the board

- `(i,j)` is free if

  `row[i]==col[j]==NWtoSE[j-i]==SWtoNE[j+i]==0`

- Add queen at `(i,j)`

```
board[i] = j
(row[i],col[j],NWtoSE[j-i],SWtoNE[j+i]) =
                                    (1,1,1,1)
```

- Remove queen at `(i,j)`

```
board[i] = -1
(row[i],col[j],NWtoSE[j-i],SWtoNE[j+i]) =
                                    (0,0,0,0)
```

```
board['col'][j] = 0
board['nwtose'][j-i] = 0
board['swtone'][j+i] = 0

f placequeen(i,board):
n = len(board['queen'].keys())
for j in range(n):
  if free(i,j,board):
    addqueen(i,j,board)
    if i == n-1:
      return(True)
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True)
    else:
      undoqueen(i,j,board)
else:
  return(False)

ard = {}
= int(input("How many queens? "))
[illegible]
 placequeen(0,board):
printboard(board)
```

U:---- F1 Nqueens.py   Bot L56   (Python) ----------------------------

8:43 Vo)) LTE1 61%

```
def initialize(board,n):
  for key in ['queen','row','col','nwtose','swtone']:
    board[key] = {}
  for i in range(n):
    board['queen'][i] = -1
    board['row'][i] = 0
    board['col'][i] = 0
  for i in range(-(n-1),n):
    board['nwtose'][i] = 0
  for i in range(2*n-1):
    board['swtone'][i] = 0

def printboard(board):
  for row in sorted(board['queen'].keys()):
    print((row,board['queen'][row]))

def free(i,j,board):
  return(board['row'][i] == 0 and board['col'][j] == 0 and
          board['nwtose'][j-i] == 0 and board['swtone'][j+i] == 0)

def addqueen(i,j,board):
  board['queen'][i] = j
  board['row'][i] = 1
  board['col'][j] = 1
  board['nwtose'][j-i] = 1
  board['swtone'][j+i] = 1
```

-UU-:----F1 8queens.py   Top L1   (Python) --

8:45 61%

```python
  for row in sorted(board['queen'].keys()):
    print((row,board['queen'][row]))

def free(i,j,board):
  return(board['row'][i] == 0 and board['col'][j] == 0 and
         board['nwtose'][j-i] == 0 and board['swtone'][j+i] == 0)

def addqueen(i,j,board):
  board['queen'][i] = j
  board['row'][i] = 1
  board['col'][j] = 1
  board['nwtose'][j-i] = 1
  board['swtone'][j+i] = 1

def undoqueen(i,j,board):
  board['queen'][i] = -1
  board['row'][i] = 0
  board['col'][j] = 0
  board['nwtose'][j-i] = 0
  board['swtone'][j+i] = 0

def placequeen(i,board):
  n = len(board['queen'].keys())
  for j in range(n):
    if free(i,j,board):
      addqueen(i,j,board)
```

-UU-:----F1 8queens.py 26% L28 (Python) ----

All solutions?

```
def placequeen(i,board): # Try row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      record solution # Last queen placed
    else:
      extendsoln = placequeen(i+1,board)
    undo this move and update board
```

```
ef placequeen(i,board):
 n = len(board['queen'].keys())
 for j in range(n):
   if free(i,j,board):
     [illegible]
     if i == n-1:
       printboard(board)
     else:
       extendsoln = placequeen(i+1,board)
    undoqueen(i,j,board)

ard = {}
 int(input("How many queens? "))
tialize(board,n)
 placequeen(0,board):
rintboard(board)
```

10:07 81%

```
    board['col'][i] = 0
  for i in range(-(n-1),n):
    board['nwtose'][i] = 0
  for i in range(2*n-1):
    board['swtone'][i] = 0

def printboard(board):
  for row in sorted(board['queen'].keys()):
    print((row,board['queen'][row]),
  print("")

def free(i,j,board):
  return(board['row'][i] == 0 and board['col'][j] == 0 and
         board['nwtose'][j-i] == 0 and board['swtone'][j+i] == 0)

def addqueen(i,j,board):
  board['queen'][i] = j
  board['row'][i] = 1
  board['col'][j] = 1
  board['nwtose'][j-i] = 1
  board['swtone'][j+i] = 1

def undoqueen(i,j,board):
  board['queen'][i] = -1
  board['row'][i] = 0
  board['col'][j] = 0
```

UU F1 8queens_allsolns.py 14% L8 (Python)

Recall 8 queens

```
def placequeen(i,board): # Trying row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      return(True) # Last queen has been placed
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True) # This solution extends fully
    else:
      undo this move and update board
  else:
    return(False) # Row i failed
```

Global variables

- Can we avoid passing board explicitly to each function?
- Can we have a single global copy of board that all functions can update?

Scope of name

- Scope of name is the portion of code where it is available to read and update
- By default, in Python, scope is local to functions
    - But actually, only if we update the name inside the function

Two examples

```
def f():
    y = x
    print(y)

x = 7
f()
```

Global variables

- Actually, this applies only to immutable values

Global variables

- Actually, this applies only to immutable values

```
def f():
  y = x[0]
  print(y)
  x[0] = 22

x = [7]
f()
```

Recall 8 queens

```
def placequeen(i,board): # Trying row i
  for each c such that (i,c) is available:
    place queen at (i,c) and update board
    if i == n-1:
      return(True) # Last queen has been placed
    else:
      extendsoln = placequeen(i+1,board)
    if extendsoln:
      return(True) # This solution extends fully
    else:
      undo this move and update board
  else:
    return(False) # Row i failed
```

Global variables

- Can we avoid passing `board` explicitly to each function?
- Can we have a single global copy of `board` that all functions can update?

Two examples

```
def f():
  y = x
  print(y)

x = 7
f()
```

4:16 77%

```
madhavan@dolphinair:...016-jul/week6/python/scope$ more f1.py
def f():
  y = x
  print(y)

x = 7
f()
madhavan@dolphinair:...016-jul/week6/python/scope$ python3.5 f1.py
7
madhavan@dolphinair:...016-jul/week6/python/scope$
```

Two examples

```python
def f():
    y = x
    print(y)

x = 7
f()
```

Fine!

```python
def f():
    y = x
    print(y)
    x = 22

x = 7
f()
```

Two examples

```python
def f():
    y = x
    print(y)

x = 7
f()
```

Fine!

```python
def f():
    y = x
    print(y)
    x = 22

x = 7
f()
```

Error!

- If x is not found in f(), Python looks at enclosing function for global x

Two examples

```
def f():
    y = x
    print(y)

x = 7
f()
```

Fine!

```
def f():
    y = x
    print(y)
    x = 22

x = 7
f()
```

Error!

- If `x` is not found in `f()`, Python looks at enclosing function for global `x`
- If `x` is updated in `f()`, it becomes a local name!

Global variables

- Actually, this applies only to immutable values

```
def f():
  y = x[0]
  print(y)
  x[0] = 22

x = [7]
f()
```

Fine!

```
def initialize(n):
  for key in ['queen','row','col','nwtose','swtone']:
    board[key] = {}
  for i in range(n):
    board['queen'][i] = -1
    board['row'][i] = 0
    board['col'][i] = 0
  for i in range(-(n-1),n):
    board['nwtose'][i] = 0
  for i in range(2*n-1):
    board['swtone'][i] = 0

def printboard():
  for row in sorted(board['queen'].keys()):
    print((row,board['queen'][row]))

def free(i,j):
  return(board['row'][i] == 0 and board['col'][j] == 0 and
         board['nwtose'][j-i] == 0 and board['swtone'][j+i] == 0)

def addqueen(i,j):
  board['queen'][i] = j
  board['row'][i] = 1
  board['col'][j] = 1
  board['nwtose'][j-i] = 1
  board['swtone'][j+i] = 1
```

Global immutable values

- What if we want a global integer
  - Count the number of times a function is called
- Declare a name to be `global`

```
def f():
  global x
  y = x
  print(y)
  x = 22

x = 7
f()
print(x)
```

Nest function definitions

- If we look up x, y inside g() or h() it will first look in f(), then outside
- Can also declare names global inside g(), h()
- Intermediate scope declaration: nonlocal

```
def f():
    def g(a):
        return(a+1)

    def h(b):
        return(2*b)

    global x
    y = g(x) + h(x)
    print(y)
    x = 22

x = 7
f()
```

Backtracking

- Systematically search for a solution
- Build the solution one step at a time
- If we hit a dead-end
  - Undo the last step
  - Try the next option

Generating permutations

- Often useful when we need to try out all possibilities
  - Each potential columnwise placement of N queens is a permutation of {0,1,…,N-1}
- Given a permutation, generate the next one
- For instance, what is the next sequence formed from `{a,b,…,m}`, in dictionary order after

`d c h b a e g l k o n m j i`

Next permutation

- Longest suffix that cannot be incremented
  - Already in descending order

$$d\ c\ h\ b\ a\ e\ g\ l\ \boxed{k\ o\ n\ m\ j\ i}$$

- The suffix starting one position earlier can be incremented
  - Replace k by next largest letter to its right, m
  - Rearrage k o n j i in ascending order

$$d\ c\ h\ b\ a\ e\ g\ l\ m\ i\ j\ k\ n\ o$$

Implementation

- From the right, identify first decreasing position

d c h b a e g l k o n m j i

- Swap that value with its next larger letter to its right

d c h b a e g l m o n k j i

  - Finding next larger letter is similar to insert

- Reverse the increasing suffix

d c h b a e g l m i j k n o

Data structures

- *Algorithms + Data Structures = Programs*
  Niklaus Wirth
- Arrays/lists — sequences of values
- Dictionaries — key-value pairs

Data structures

- *Algorithms + Data Structures = Programs* Niklaus Wirth
- Arrays/lists — sequences of values
- Dictionaries — key-value pairs
- Python also has sets as a built in datatype

Sets in Python

- List with braces, duplicates automatically removed

```
colours = {'red','black','red','green'}
```

```
>>> print(colours)
{'black', 'red', 'green'}
```

- Create an empty set

```
colours = set()
```

  - Note, not `colours = {}` — empty dictionary!

Sets in Python

- Set membership

```
>>> 'black' in colours
True
```

- Convert a list into a set

```
>>> numbers = set([0,1,3,2,1,4])
>>> print(numbers)
{0, 1, 2, 3, 4}
```

Sets in Python

- Set membership

```
>>> 'black' in colours
True
```

- Convert a list into a set

```
>>> numbers = set([0,1,3,2,1,4])
>>> print(numbers)
{0, 1, 2, 3, 4}

>>> letters = set('banana')
>>> print(letters)
{'a', 'n', 'b'}
```

Set operations

```
odd = set([1,3,5,7,9,11])
prime = set([2,3,5,7,11])
```

- Union
  `odd | prime → {1, 2, 3, 5, 7, 9, 11}`
- Intersection
  `odd & prime → {3, 11, 5, 7}`

Set operations

```
odd = set([1,3,5,7,9,11])
prime = set([2,3,5,7,11])
```

- Union
  `odd | prime → {1, 2, 3, 5, 7, 9, 11}`

- Intersection
  `odd & prime → {3, 11, 5, 7}`

Set operations

```
odd = set([1,3,5,7,9,11])
prime = set([2,3,5,7,11])
```

- Union
  `odd | prime → {1, 2, 3, 5, 7, 9, 11}`
- Intersection
  `odd & prime → {3, 11, 5, 7}`
- Set difference
  `odd - prime → {1, 9}`
- Exclusive or
  `odd ^ prime → {1, 2, 9}`

Stacks

- Stacks are natural to keep track of recursive function calls
- In 8 queens, use a stack to keep track of queens added
  - Push the latest queen onto the stack
  - To backtrack, pop the last queen added

Queues

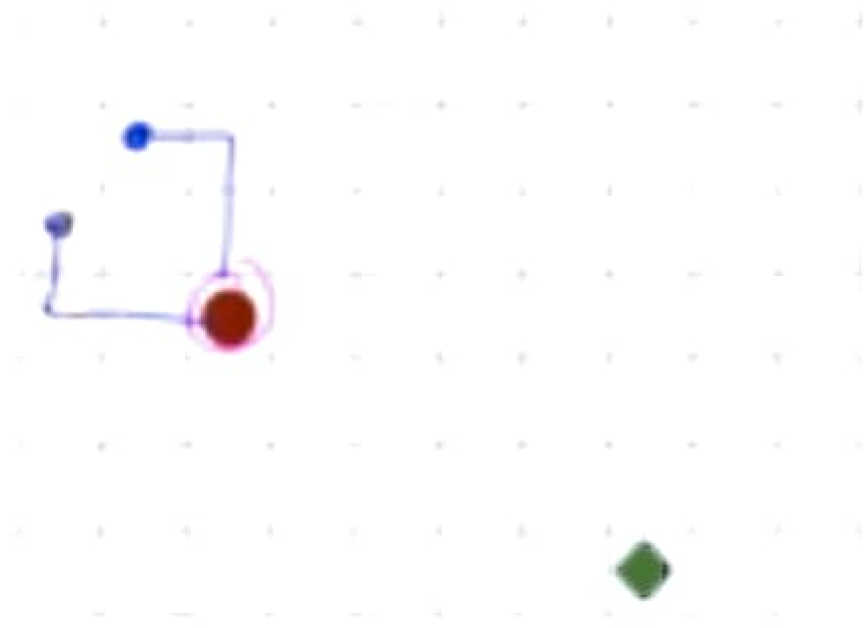
- First-in, first-out sequences
  - `addq(q,x)` — adds x to rear of queue q
  - `removeq(q)` — removes element at head of q
- Using Python lists, left is rear, right is front

Queues

- First-in, first-out sequences
  - `addq(q,x)` — adds x to rear of queue q
  - `removeq(q)` — removes element at head of q
- Using Python lists, left is rear, right is front
  - `addq(q,x)` is `q.insert(0,x)`
    - `l.insert(j,x)`, insert x before position j

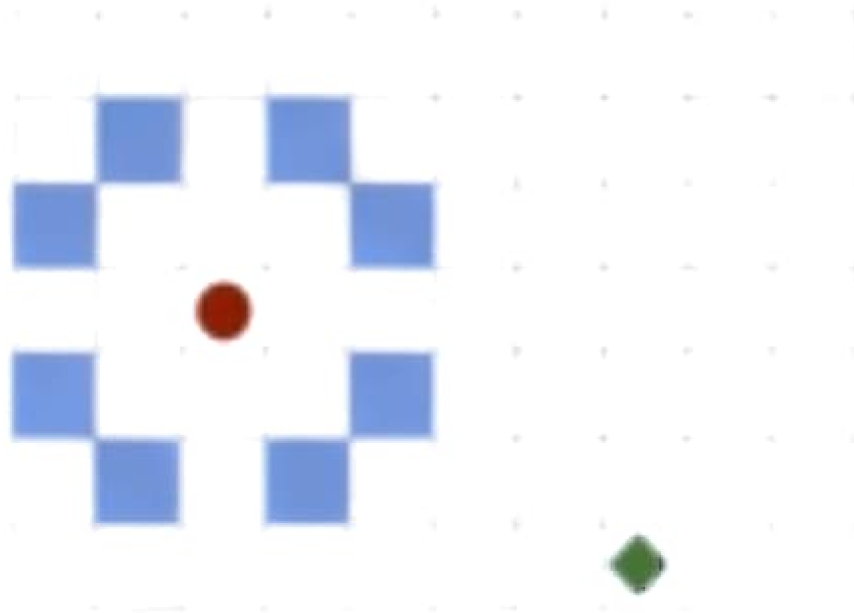
Systematic exploration

- Rectangular m x n grid
- Chess knight starts at (sx,sy)
  - Usual knight moves
- Can it reach a target square (tx,ty)? 



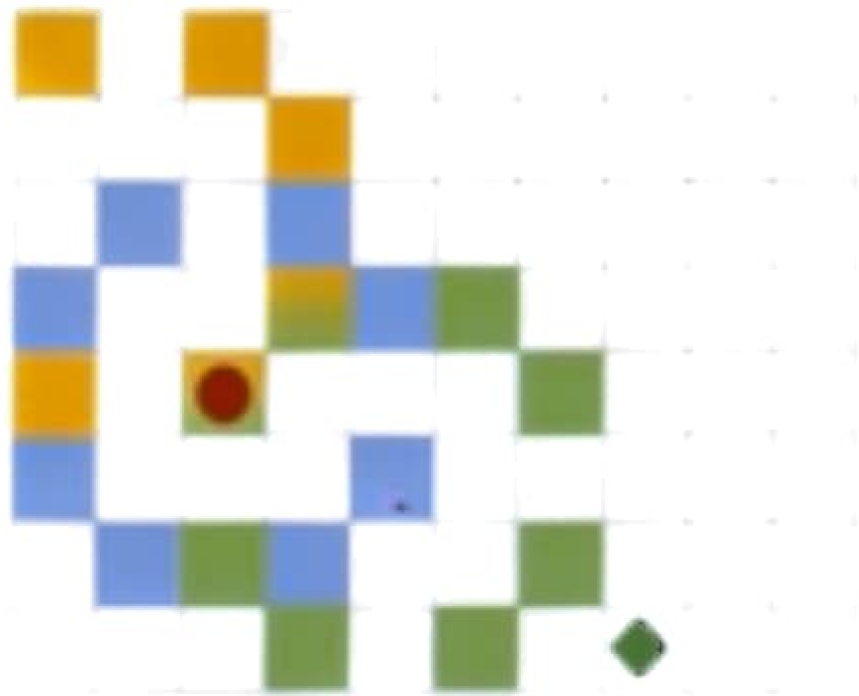
Systematic exploration

- Rectangular m x n grid
- Chess knight starts at (sx,sy)
  - Usual knight moves
- Can it reach a target square (tx,ty)? 



Systematic exploration

- Rectangular m x n grid
- Chess knight starts at (sx,sy)
  - Usual knight moves
- Can it reach a target square (tx,ty)? 



Systematic exploration

- X1 — all squares reachable in one move from (sx,sy)
- X2 — all squares reachable from X1 in one move
- . . .
- Don't explore an already marked square
- When do we stop?

Systematic exploration

- Maintain a queue Q of cells to be explored
- Initially Q contains only start node (sx,sy)
    - Remove (ax,ay) from head of queue
    - Mark all squares reachable in one step from (ax,ay)

Systematic exploration

- Maintain a queue Q of cells to be explored
- Initially Q contains only start node (sx,sy)
  - Remove (ax,ay) from head of queue
  - Mark all squares reachable in one step from (ax,ay)
  - Add all newly marked squares to the queue

Systematic exploration

```
def explore((sx,sy),(tx,ty)):
  marked = [[0 for i in range(n)]
                       for j in range(m)]
  marked[sx][sy] = 1
  queue = [(sx,sy)]
  while queue != []:
    (ax,ay) = queue.pop()
    for (nx,ny) in neighbours((ax,ay)):
      if !marked[nx][ny]:
        marked[nx][ny] = 1
        queue.insert(0,(nx,ny))
  return(marked[tx][ty])
```

Systematic exploration

- X1 — all squares reachable in one move from (sx,sy)
- X2 — all squares reachable from X1 in one move
- . . .
- Don't explore an already marked square
- When do we stop?
    - If we reach target square
    - What if target is not reachable?

Systematic exploration

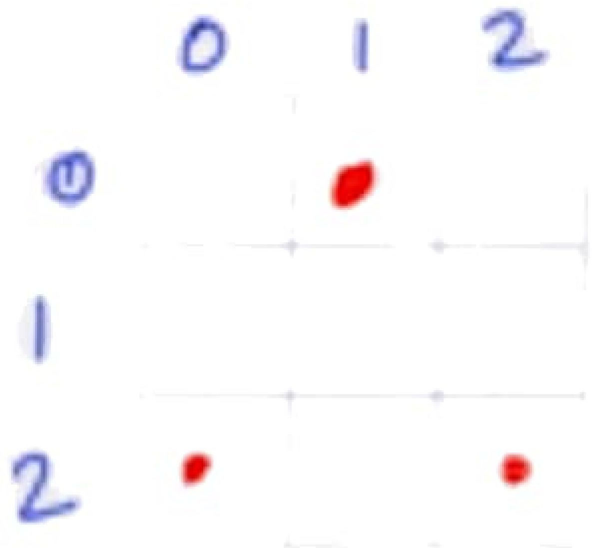
- Maintain a queue Q of cells to be explored
- Initially Q contains only start node (sx,sy)
  - Remove (ax,ay) from head of queue
  - Mark all squares reachable in one step from (ax,ay)
  - Add all newly marked squares to the queue
- When the queue is empty, we have finished

Systematic exploration

```
def explore((sx,sy),(tx,ty)):
  marked = [[0 for i in range(n)]
                        for j in range(m)]
  marked[sx][sy] = 1
  queue = [(sx,sy)]
  while queue != []:
    (ax,ay) = queue.pop()
    for (nx,ny) in neighbours((ax,ay)):
      if !marked[nx][ny]:
        marked[nx][ny] = 1
        queue.insert(0,(nx,ny))
  return(marked[tx][ty])
```

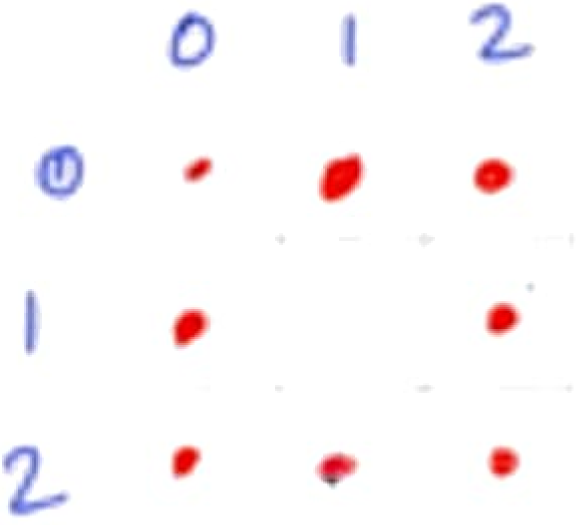
Example

src = (0,1) **tgt =(1,1)**



Example

src = (0,1) **tgt =(1,1)**



~~(2,0)~~ ~~(2,2)~~ ~~(1,2)~~ ~~(1,0)~~ ~~(0,0)~~

(0,2)

Example

src = (0,1)   tgt =(1,1)

- This is an example of breadth first search

Summary

- Data structures are ways of organising information that allow efficient processing in certain contexts
- Python has a built-in implementation of sets
- Stacks are useful to keep track of recursive computations
- Queues are useful for breadth-first exploration

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities.
- When the processo [illegible] e scheduler picks out the job with max [illegible] iority in the list and schedules it.
- New jobs may join the list at any time.
- How should the scheduler maintain the list of pending jobs and their priorities?

Priority queue

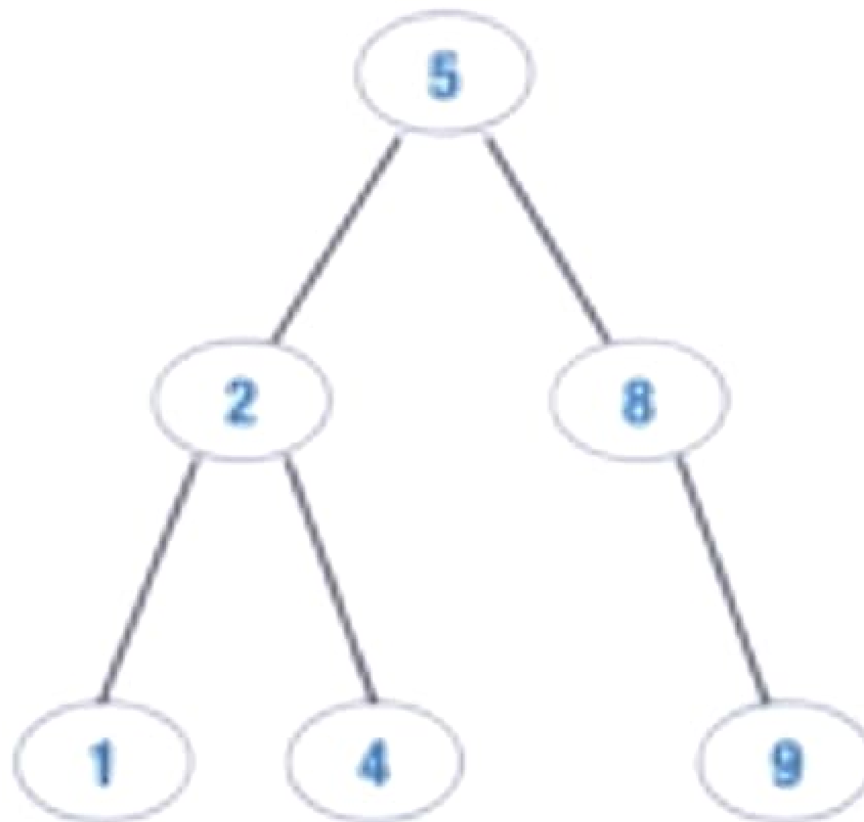
- Need to maintain a list of jobs with priorities to optimise the following operations
  - delete_max( )
    - Identify and remove job with highest priority
    - Need not be unique
  - insert( )
    - Add a new job to the list

Linear structures

- Unsorted list
    - insert( ) takes O(1) time
    - delete_max( ) takes O(n) time
- Sorted list
    - delete_max( ) takes O(1) time
    - insert( ) takes O(n) time
- Processing a sequence of n jobs requires $O(n^2)$ time

Binary tree

- Two dimensional structure
- At each node
  - Value
  - Link to parent, left child, right child

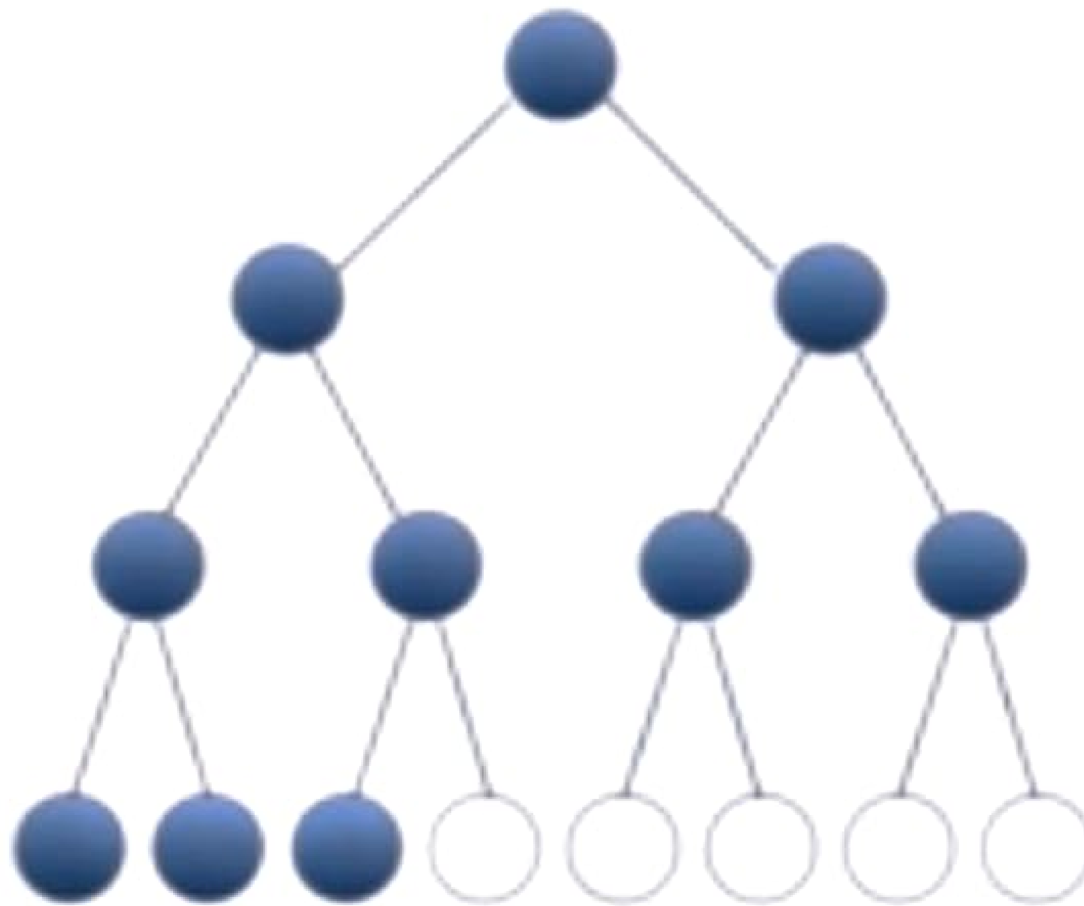


Priority queues as trees

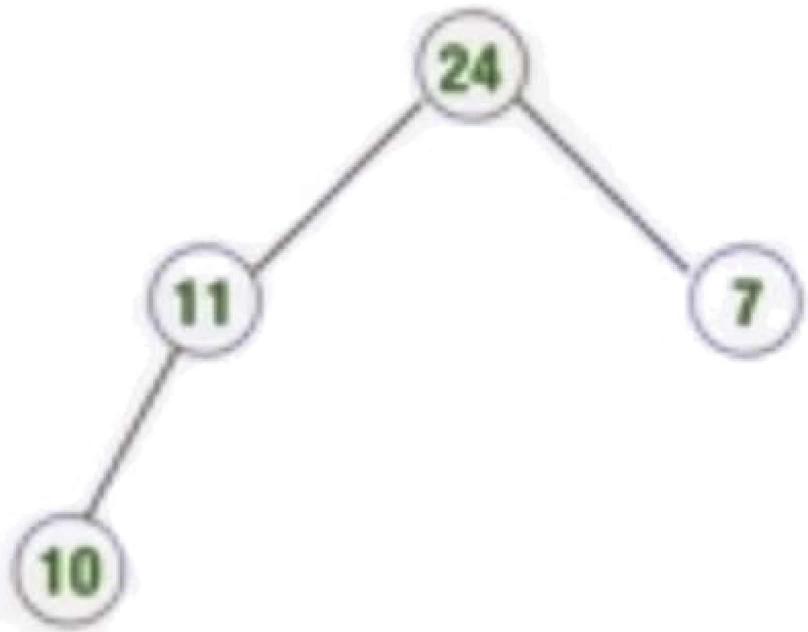
- Maintain a special kind of binary tree called a heap
    - Balanced: N node tree has height log N
- Both insert( ) and delete_max( ) take O(log N)
    - Processing N jobs takes time O(N log N)
- Truly flexible, need not fix upper bound for N in advance

Heaps

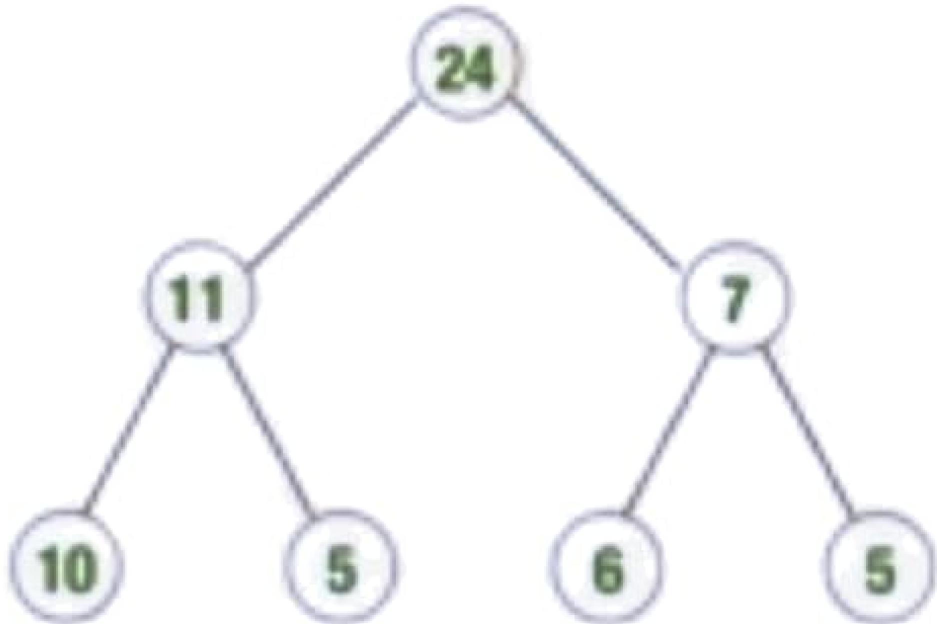
- Binary tree filled level by level, left to right
- At each node, value stored is bigger than both children
    - (Max) Heap PropertyBinary tree filled level by level, left to right



Examples

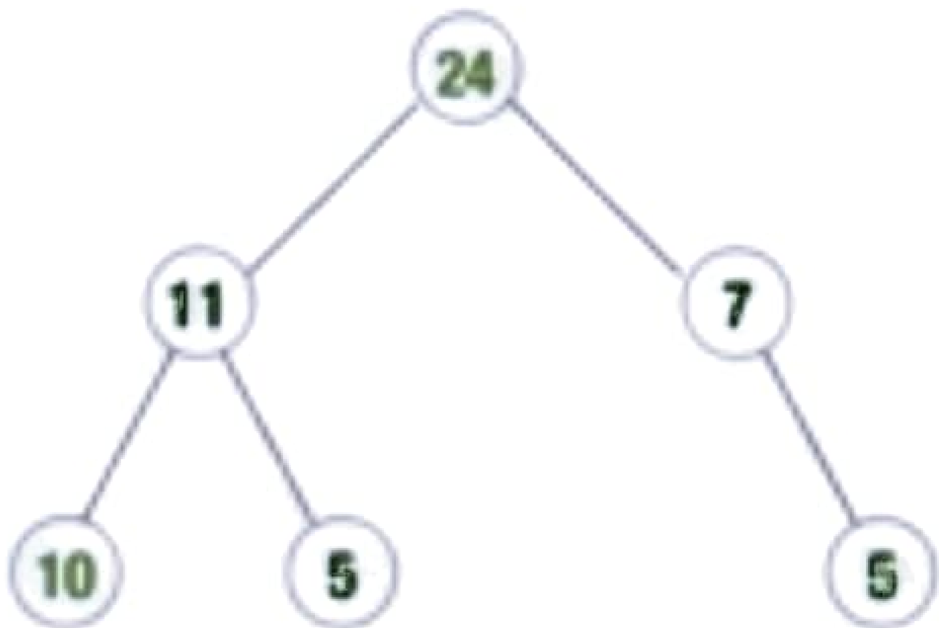


Examples



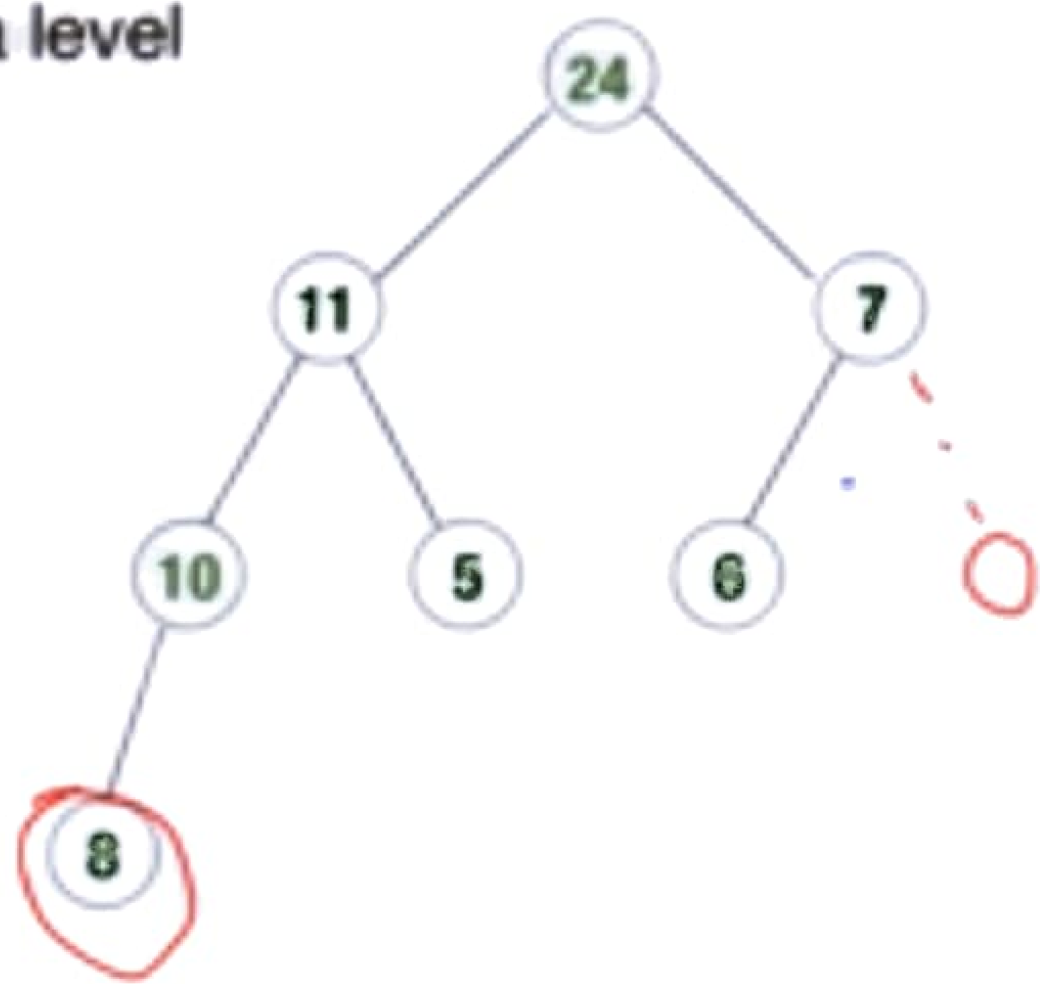
Non-examples

- No “holes” allowed



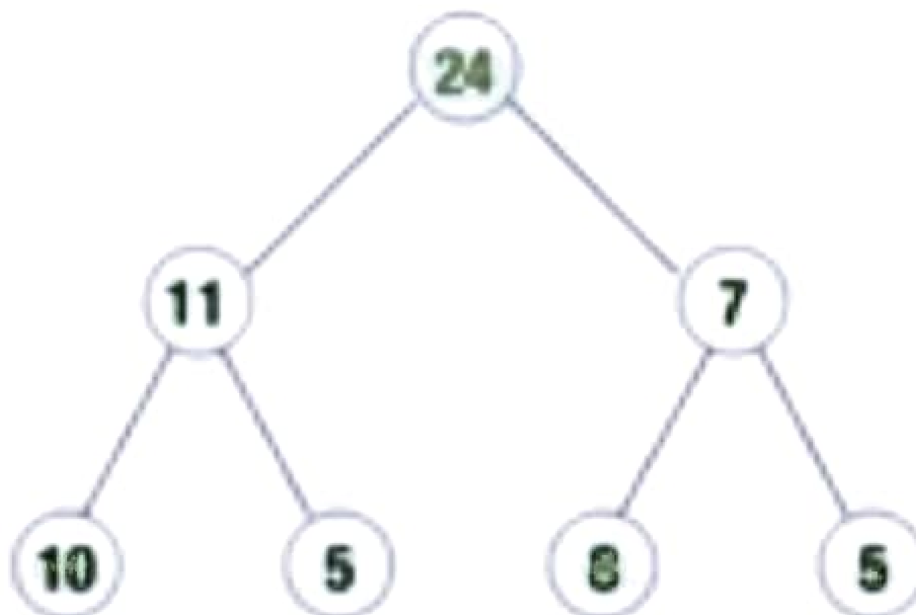
Non-examples

- Can't leave a level incomplete



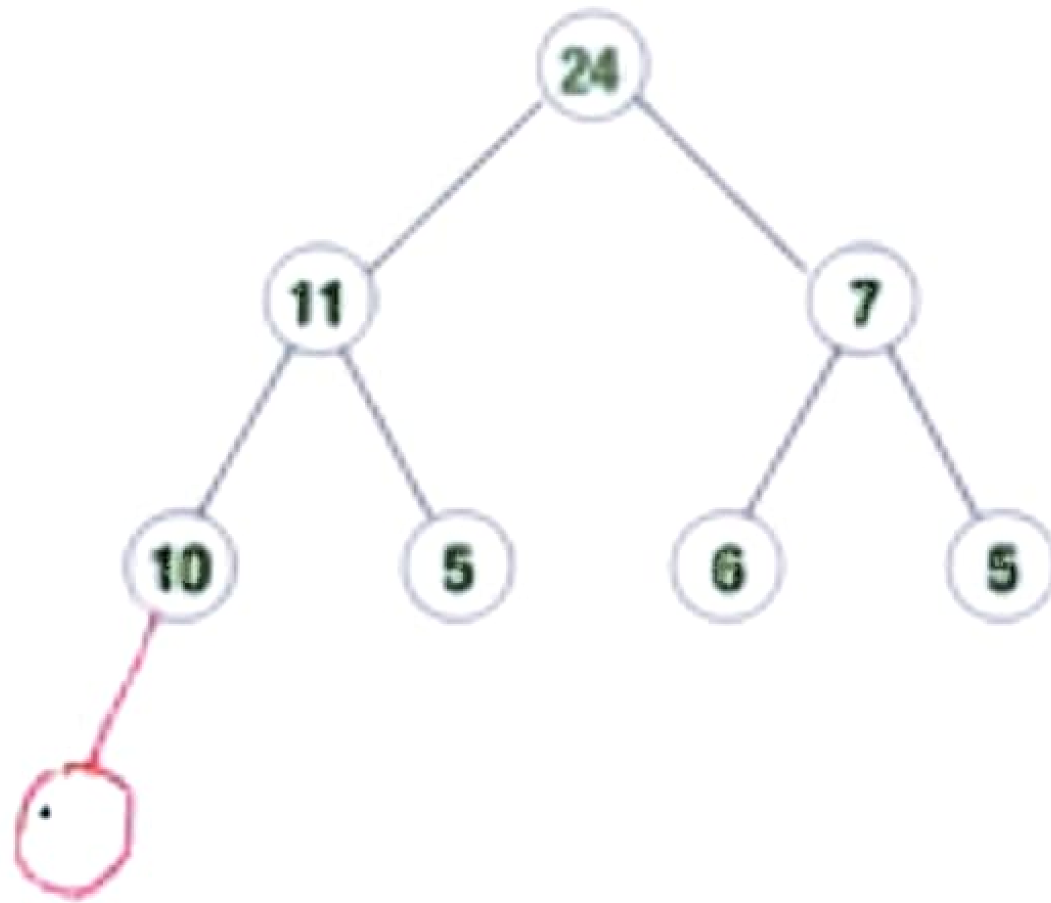
Non-examples

- Violates heap property



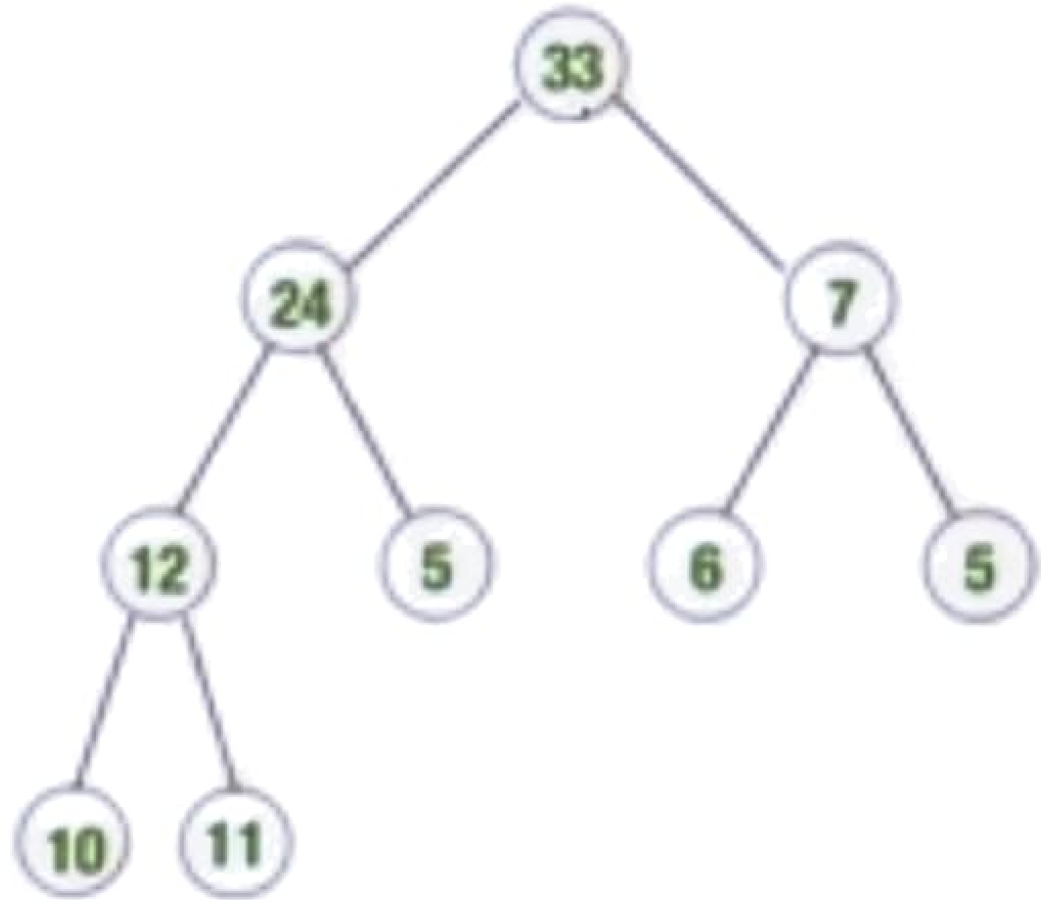
insert( )

- insert 12
- Position of new node is fixed by structure
- Restore heap property along the path to the root



insert( )

- insert 33

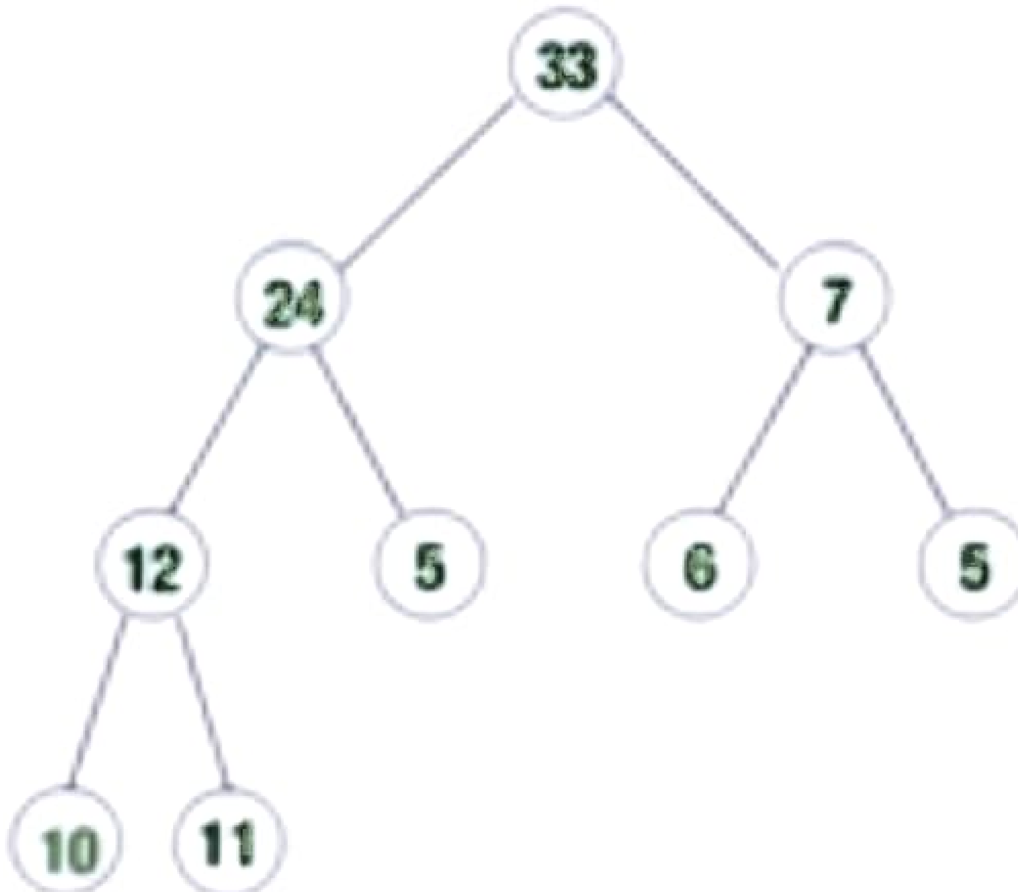


Complexity of insert( )

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level 0,1,…,i is $2^0, 2^1, \ldots, 2^i$
- K levels filled : $2^0+2^1+\ldots+2^{k-1} = 2^k - 1$ nodes
- N nodes : number of levels at most log N + 1
- insert( ) takes time O(log N)

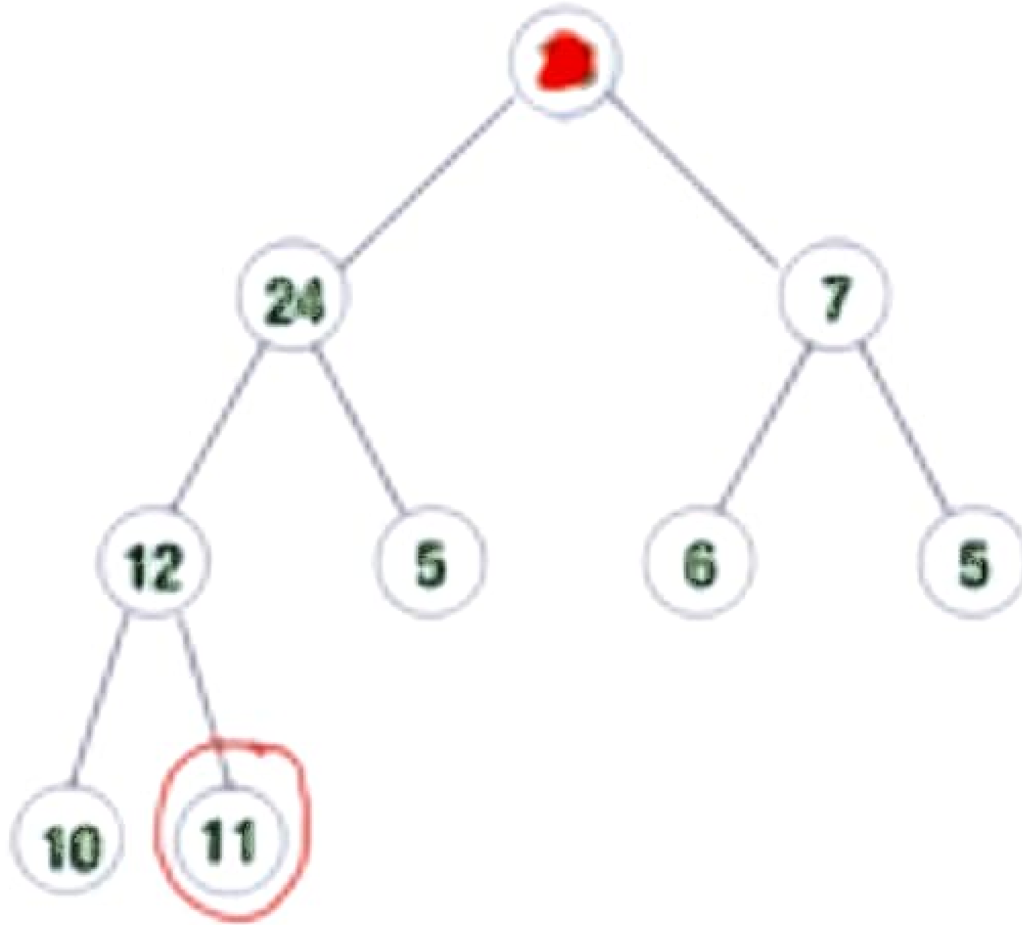
delete_max( )

- Maximum value is always at the root
  - From heap property, by induction
- How do we remove this value efficiently?



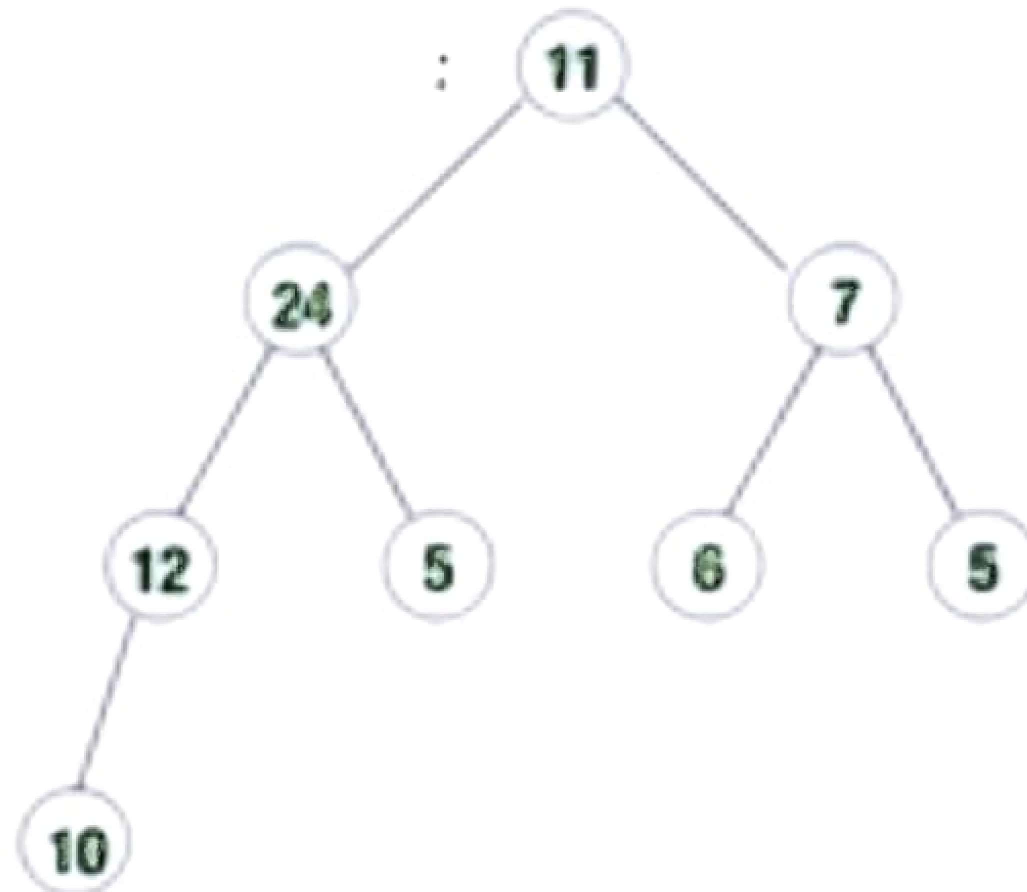
delete_max( )

- Removing maximum value creates a "hole" at the root
- Reducing one value requires deleting last node
- Move "homeless" value to root



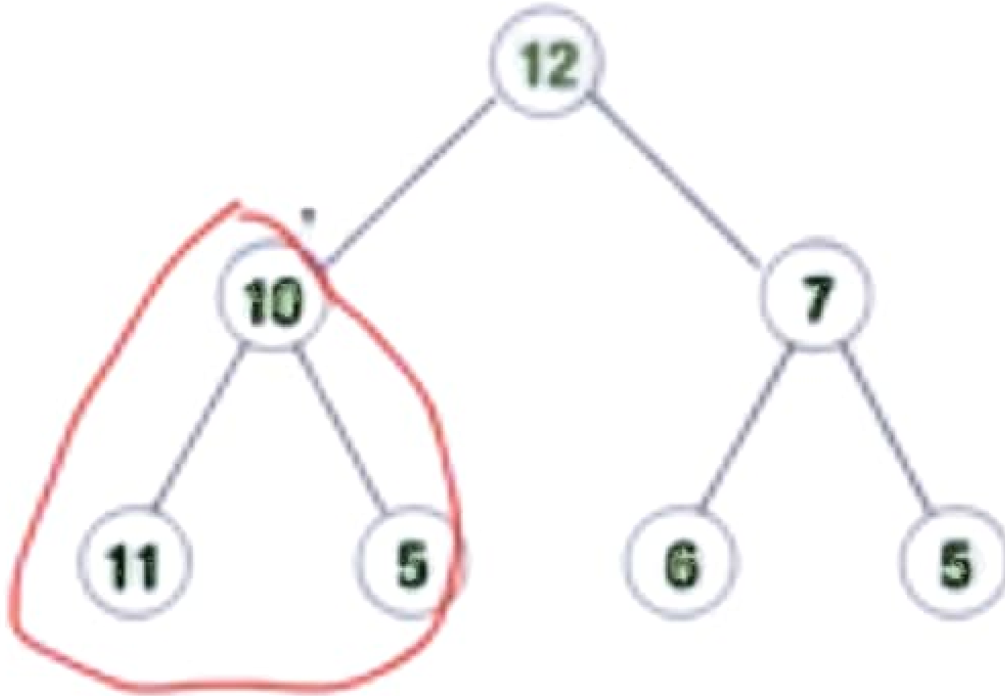
delete_max( )

- Now restore the heap property from root downwards
  - Swap with largest child
- Will follow a single path from root to leaf



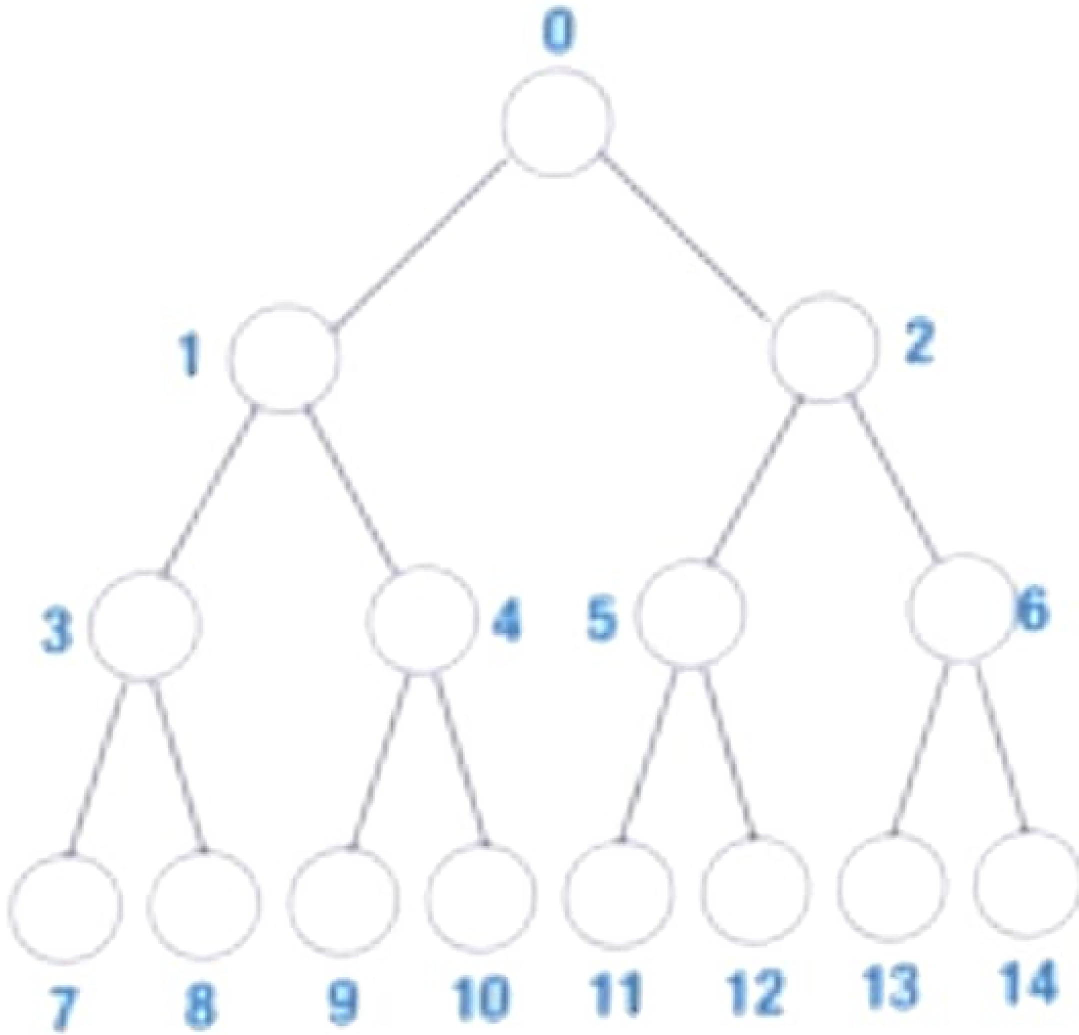
delete_max( )

- Will follow a single path from root to leaf
- Cost proportional to height of tree
- O(log N)



Impementing using arrays

- Number the nodes left to right, level by level
- Represent as an array H[0..N-1]
- Children of H[i] are at H[2i+1], H[2i+2]
- Parent of H[j] is at H[floor((j-1)/2)] for j > 0



Building a heap, heapify( )

- Given a list of values $[x_1, x_2, \ldots, x_N]$, build a heap
- Naive strategy
  - Start with an empty heap
  - Insert each x_i
  - Overall $O(N \log N)$

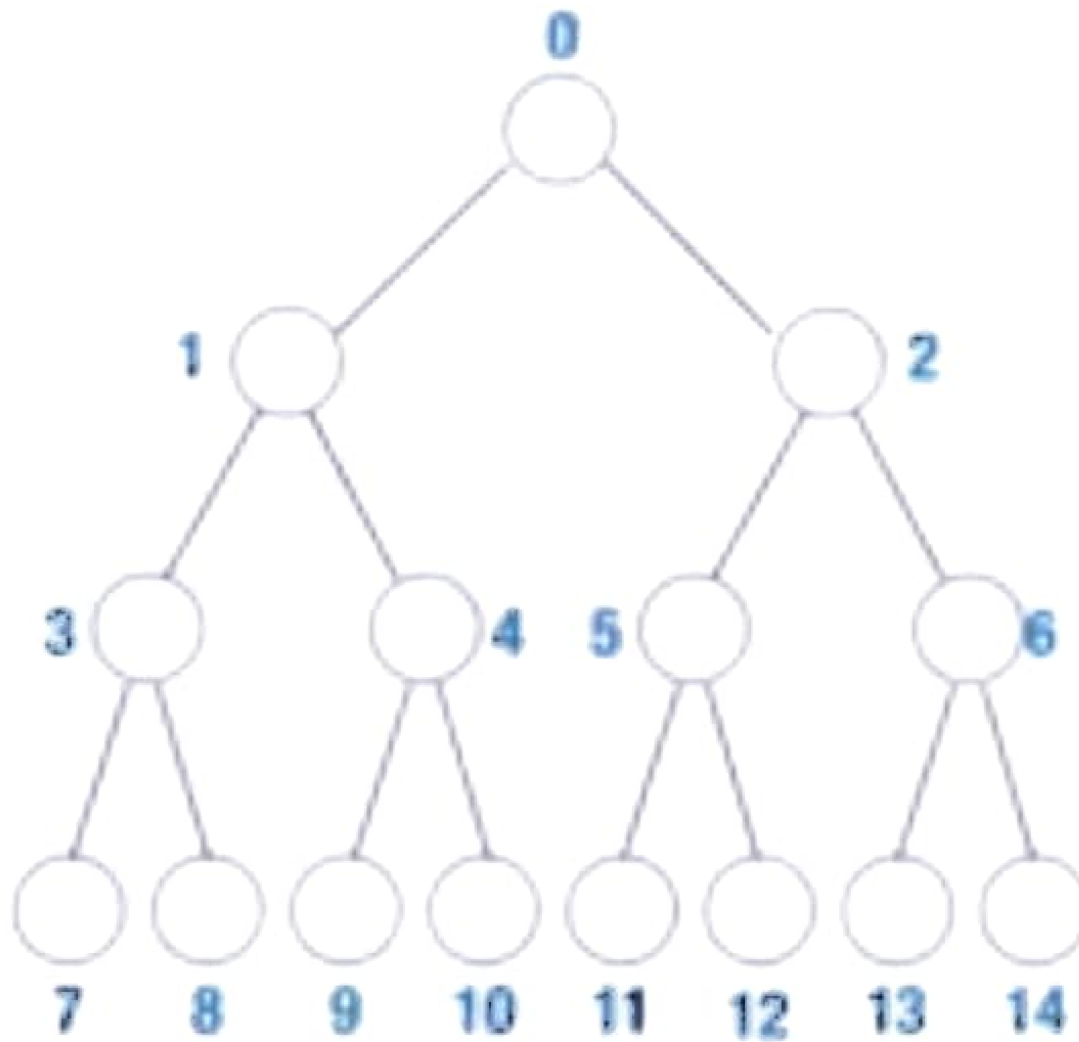
Building a heap, heapify( )

- Given a list of values $[x_1, x_2, \ldots, x_N]$, build a heap
- Naive strategy
  - Start with an empty heap
  - Insert each x_i
  - Overall O(N log N)

Better heapify( )

- Set up the array as $[x_1, x_2, \ldots, x_N]$
  - Leaf nodes trivially satisfy heap property
  - Second half of array is already a valid heap
- Assume leaf nodes are at level k
  - For each node at level k-1, k-2, … , 0, fix heap property
  - As we go up, the number of steps per node goes up by 1, but the number of nodes per level is halved
  - Cost turns out to be O(N) overall

Better heapify( )

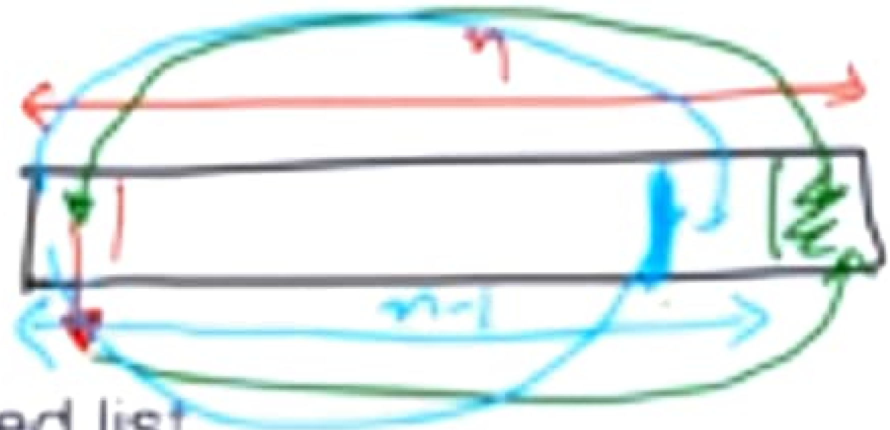


N/2 nodes already satisfy heap property

Heap sort

- Start with an unordered list
- Build a heap — O(n)
- Call delete_max( ) n times to extract elements in descending order — O(n log n)
- After each delete_max( ), heap shrinks by 1
  - Store maximum value at the end of current heap
  - In place O(n log n) sort

Heap sort



- Start with an unordered list
- Build a heap — O(n)
- Call delete_max( ) n times to extract elements in descending order — O(n log n)
- After each delete_max( ), heap shrinks by 1
    - Store maximum value at the end of current heap
    - In place O(n log n) sort

Summary

- Heaps are a tree implementation of priority queues
  - insert( ) and delete_max( ) are both O(log N)
  - heapify( ) builds a heap in O(N)
  - Tree can be manipulated easily using an array
- Can invert the heap condition
  - Each node is smaller than its children
  - Min-heap, for insert( ), delete_min( )