Question of the day

- How do we implement elementary data structures like stacks, queues, lists, and trees?

Stacks, queues, lists, and trees

William Hendrix

Lecture 3

Outline

- Lists
- Stacks
- Queues
- Sets
- Trees
  - Binary search trees

Review: iterative algorithm analysis

- Identify loops and function calls
  - Everything else is *Θ(1)*
- *For loops*:
  - Estimate number of iterations
    - Incrementing by *c*: divide range by *c* to get iterations
    - Multiplying by *c:* take $\log_c$ of the end/start ratio
  - Estimate loop body running time
    - Might depend on iteration #
  - If iterations don't depend on i: # iterations * time per iteration
  - Otherwise: sum up all iterations

$$\sum_{i=1}^{x} i = \Theta(x^2) \qquad \sum_{i=1}^{x} \frac{1}{i} = \Theta(\lg x) \qquad \sum_{i=1}^{x} 2^i = \Theta(2^x) \qquad \sum_{i=1}^{x} \frac{1}{2^i} = \Theta(1)$$

- *For functions:*
  - Analyze other functions separately
  - Recursive functions: set up a recurrence and solve
- Overall complexity: largest loop or function call complexity

Recursive analysis review

- Analyze pseudocode for recurrence
  - E.g., T(n) = T(n/2) + Θ(1)
    - # and size of all recursive calls
    - Time for all nonrecursive code
- **Recursion tree analysis**
  - Nodes represent recursive calls
  - Start with n, add children based on recursive calls
  - Add up nonrecursive complexity for all nodes
    - Complexity may depend on input size
- **Master Theorem**
  - Applies to $T(n) = aT(n/b) + f(n)$
  - Calculate $c = \log_b(a)$ and compare n^c vs. $f(n)$

$$T(n) = \begin{cases} \Theta(n^c), & \text{if } f(n) = O(n^{c-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^c \lg n), & \text{if } f(n) = \Theta(n^c) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{c+\epsilon}) \text{ for some } \epsilon > 0 \\ & \quad \underline{\text{and}}\ af(n/b) < f(n) \text{ for large } n \end{cases}$$

Exponent on n must be < or > c

Data structures in memory

- Contiguous
  - Allocate single chunk of memory
  - Retrieve elements by locating data's index in chunk
  - Very fast if elements are accessed consecutively (caching)
  - No memory wasted on storing pointers
  - Following *k* links takes *O(k)* time vs. *O(1)* for pointer arithmetic
- Link-based
  - Data are stored in small "islands" connected via pointers
  - Retrieve elements by starting at the head/tail/root and traversing links
  - Supports data with irregular structure (graphs, trees)
  - Easier for memory manager to allocate
  - Easy to modify structure by changing links (vs. copying data)
- Hybrid

List

- ADT that stores multiple data values
  - Access based on element *index*

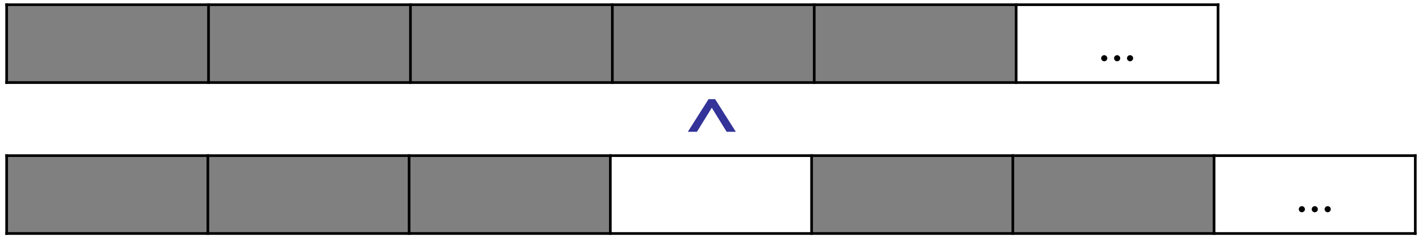
Main operations

- **Get(i):** returns element at position i
  - Sometimes denoted `list[i]`
- **Set(i, x):** changes element at position i to x
  - Sometimes denoted `list[i] = x`
- **Insert(i, x):** adds x as new element at position i
- **Delete(i):** deletes element at position i

- Not same as array: indexes change over time
  - No fixed size
- Two main implementations: array and linked list

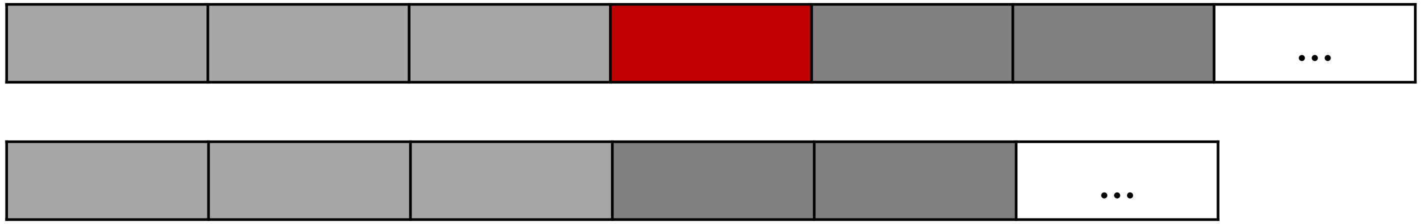
Array-based list

- Get and Set are trivial
  - Return or modify index i
- Insert and Delete are trickier
  - Insert shifts elements right to make room



    - Then insert

    - Needs to increase capacity first if full (next slide)
  - Delete shifts later elements left



Array enlargement policy

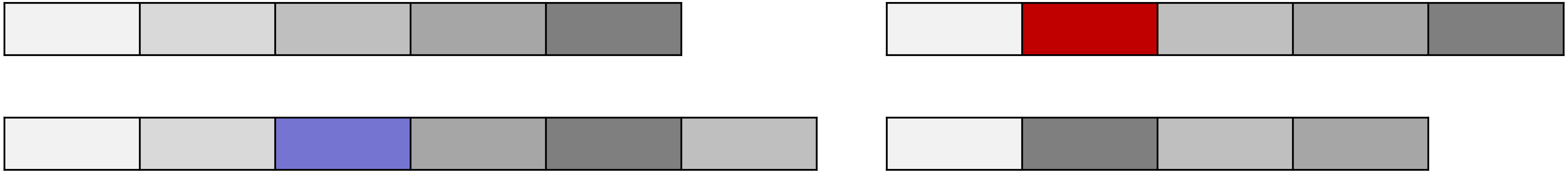
- **Bad policy:** increment by 1
  - Cost: $\Theta(n)$ every time
  - n pushes: $\Theta(1+2+\ldots+n)) = \Theta(n^2)$
  - *Average cost per push:* $O(n)$
- **Better policy:** increment by k (k=10, 100, etc.)
  - Cost: $\Theta(n)$ every k^{th} time
  - n pushes: $\Theta(1+(1+k)+\ldots+(1+k\lfloor\frac{n}{k}\rfloor)) = \Theta\left(\frac{n^2}{k}\right)$
  - *Average cost per push:* $\Theta(\frac{n}{k})$
  - *Caution*: space trade-off
    - Increment by 1B: small stacks will be mostly empty space
- **Best policy:** double array size
  - Cost: $\Theta(n)$ after powers of two
  - n pushes: $\Theta(1+2+4+\ldots+2^{\lfloor \lg n\rfloor}) = \Theta(n)$
  - *Average cost per push:* $\Theta(1)$
  - Array will be at least half-full if not deleting

Amortized analysis

Array-based list complexity

Operation	Worst-case complexity
Get(i)	$\Theta(1)$
Set(i, x)	$\Theta(1)$
Insert(i, x)	$O(n)$
Delete(i)	$O(n)$

- *Variant:* insert by swapping arr[i] to end, then inserting
  - Delete by swapping last element to index i and decrementing size

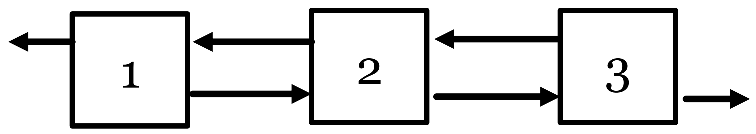


  - Insert and Delete become $\Theta(1)$
  - Doesn't maintain relative order of elements

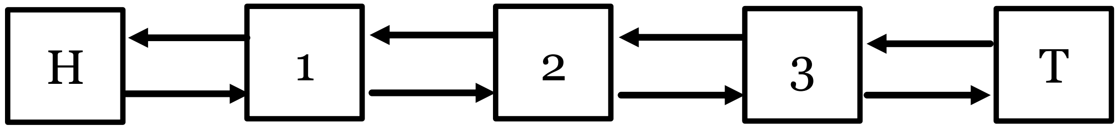
Linked lists

- Pointer-based data structure
  - Data values "chained" together by pointers
  - Alternative to array
  - Can be doubly- or singly-linked
- Two parts
  - List: represents list as a whole
    - Most list operations are here
    - Stores head pointer, maybe size and tail
  - Node: represents a single link in the chain
    - Data value and next pointer, maybe previous pointer
- May be implemented with or without *sentinel nodes*
  - Simplify code: no special cases for `nullptr/head/tail`
  - Extra nodes (constructor and destructor)
- **Sentinel value**
  - Design pattern where a special data value serves as a marker
    - E.g., a node containing -1 when all values are positive

No sentinel nodes:

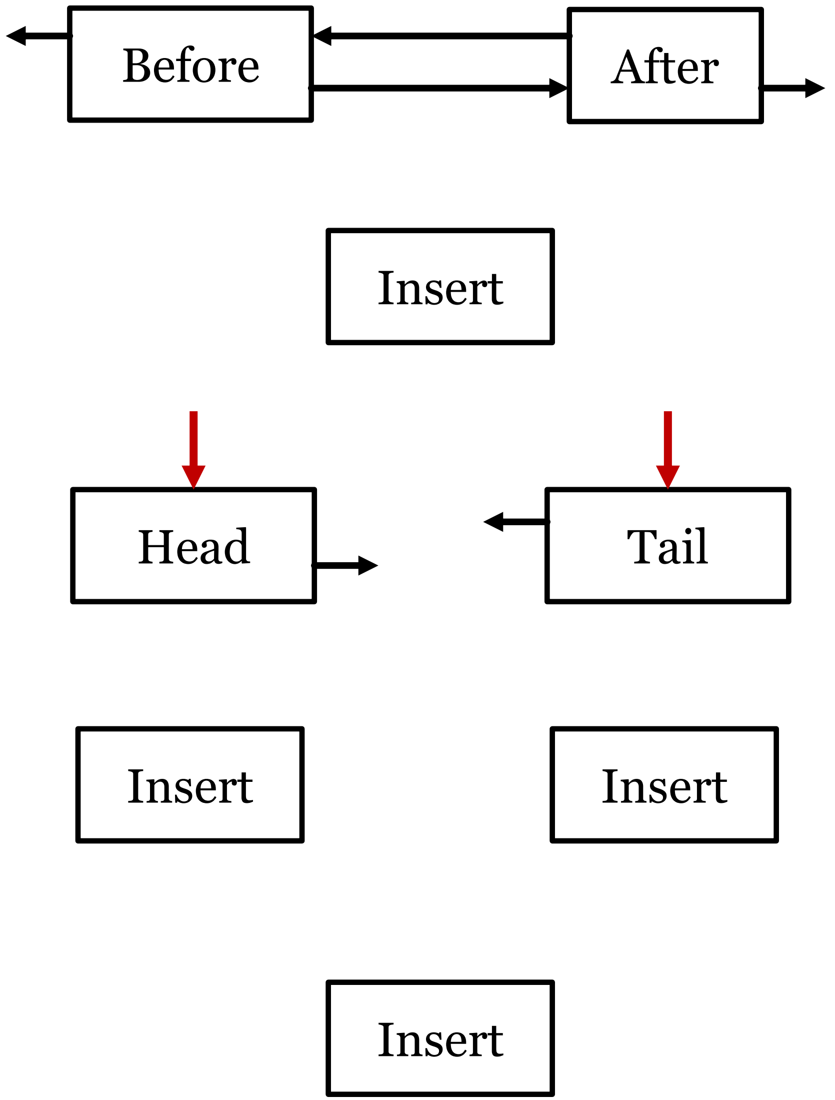


With sentinel nodes:



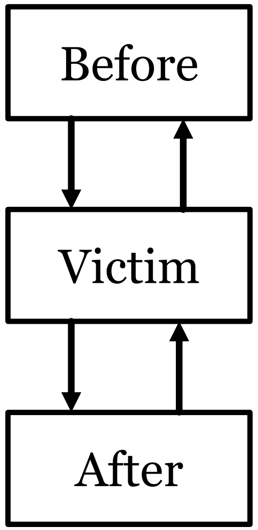
Linked list operations

- **Element access**
  - Start at head
  - Follow next pointers until at correct position
- **Insert (no sentinel nodes)**
  - Create new node, and increment size
  - Initialize node->next and node->prev
  - If not inserting after tail:
    - Set after->prev to node
  - Otherwise:
    - tail = node
  - If not inserting before head:
    - Set before->next to node
  - Otherwise:
    - head = node



Other linked list operations

- **Remove**
  - If not head, node->prev->next = node->next
    - Else, head = head->next
  - If not tail, node->next->prev = node->prev
    - Else, tail = tail->prev
- **Destructor**
  - Scan through list, deleting every node
  - Tricky: move to next node before deleting current
- **Copy constructor**
  - Scan through original
  - For each node:
    - Make copy
    - Connect to previous (if not head)
  - Set tail and size
- **Copy assignment**
  - If this != rhs, delete then copy



Linked list exercise

- Write pseudocode for a function that takes a doubly-linked list and returns a copy of the list in reverse order
  - LinkedList data members: *head*, *tail*, *size*
    - No sentinel values (nil/NULL pointers at head/tail)
  - LLNode data members: *data*, *next*, *prev*
    - Constructor that accepts *data* (*prev* and *next* set to nil)

```
reverse(list):
  reverseList = List()
  curr = list.tail
  while curr != nil:
    reverseList.append(curr)
    curr = curr.prev
```

Linked list exercise

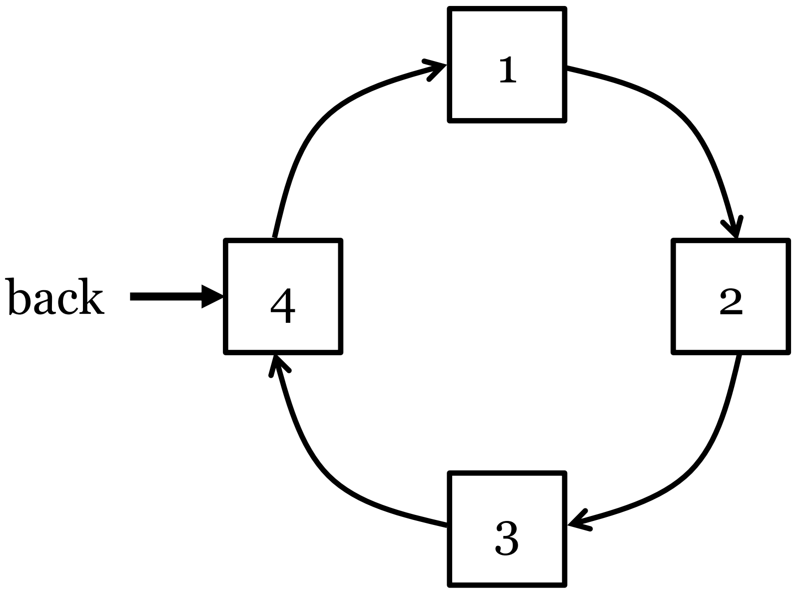
- Write pseudocode for a function that takes a doubly-linked list and returns a copy of the list <u>in reverse order</u>

```
reverse(list):
    ret = LinkedList()
    ret.size = list.size
    if list.size > 0:
        from = list.tail
        prev = ret.head = LLNode(from.data)
        for i = 2 to ret.size:
            from = from.prev
            prev.next = LLNode(from.data)
            prev.next.prev = prev
            prev = prev.next
        ret.tail = prev
 return ret
```

Circular linked lists

- Variant of linked list
- Last pointer points to first
- Used to represent cyclic data

- Traditionally singly-linked
- Store *back* pointer for easy appending
- May use *sentinel node* to indicate "end" of list



List comparison

Operation	Array	Linked list
Get(i)	$\Theta(1)$	$O(n)$
Set(i, x)	$\Theta(1)$	$O(n)$
Insert(x, i)	$O(n)$	$O(n)$
Delete(i)	$O(n)$	$O(n)$

- Both have advantages
  - LinkedList ops are $\Theta(1)$ if given pointers
  - Some LL ops (e.g., concatenation) are faster
  - Array faster if scanning
    - Especially if using swap variant

Stack implementations

- Can be implemented using linked list or array
- Linked list implementation
  - push(): append
    - O(1)
  - pop(): delete tail
    - O(1)
  - peek(): return tail
    - O(1)
- Array implementation ← Generally preferred
  - Similar to vector
  - push(): add and expand when full
    - O(1) "on average"
  - pop(): decrease size
    - O(1)
  - peek(): return last element
    - O(1)

Stack applications

- Stacks used for many algorithms
  - Reversing a string
    - Push everything, then pop everything
  - Matching parentheses
    - Push on (, [, {
    - Pop and match on),], }
    - Return true at end if everything matched
- Backtracking
  - Powerful technique for solving many problems
  - Start at initial state
  - Make a move and push on stack
  - Repeat until you find the goal or run out of moves
    - If out of moves, pop and undo last move, then try something else
  - Often implemented using recursion
    - Call stack

Queues

- Data structures related to stacks
- Three main operations
  - enqueue(x)
    - Add a new element to the back of the queue
  - dequeue()
    - Remove the element to the front of the queue
  - peek()
    - Return the front of queue without removing
  - Analogy: line
    - First-In, First-Out (FIFO) ordering
    - Complementary to stack

- Deque ("deck")
  - Double-Ended QUEue
  - 4 operations: push_front(), push_back(), pop_front(), pop_back()
  - Can simulate stack (push_back/pop_back) or queue (push_back/pop_front)

Queue applications

- Not as common as stacks overall

- Used for resource allocation
  - E.g., printer queues
  - Good because everyone waits roughly the same amount of time to get through the queue
- Also used in graph and tree traversals

- C++ STL provided templated stack and queue implementations

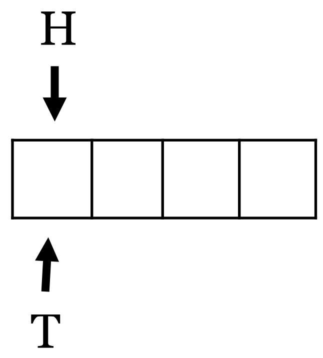
```
#include <stack>
#include <queue>
//...
stack<int> s; //s.push(0), s.pop(), s.top()
queue<double> q; //q.push(0); q.pop(), q.front()
//Both:  .size(), .empty()
```

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



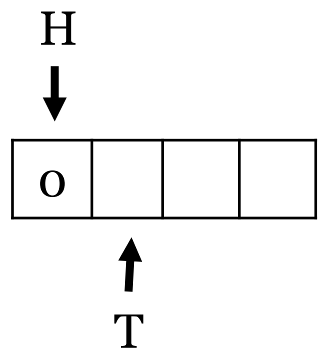
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



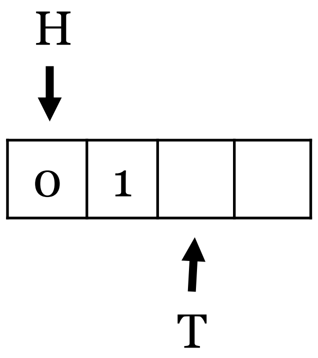
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



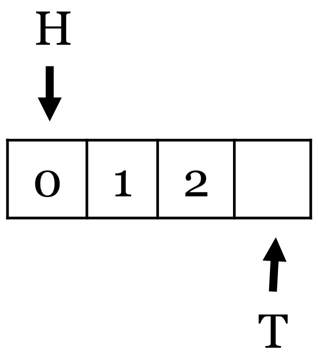
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



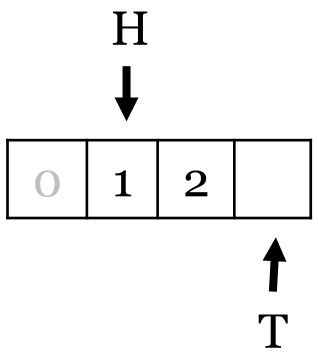
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



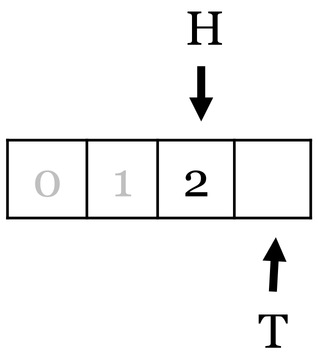
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



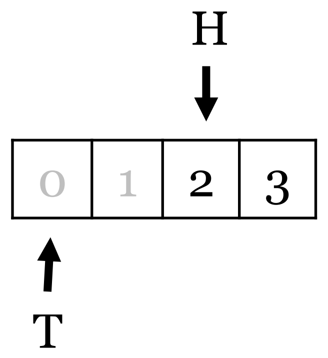
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



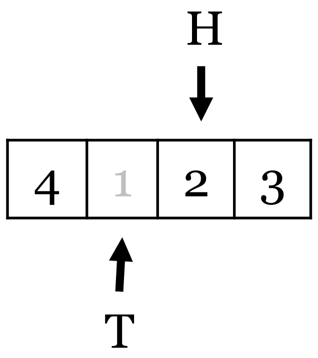
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



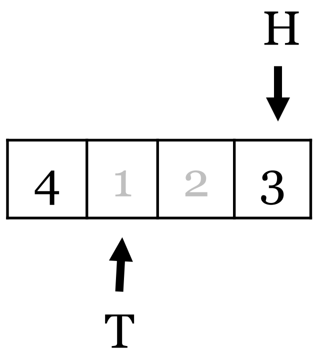
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)

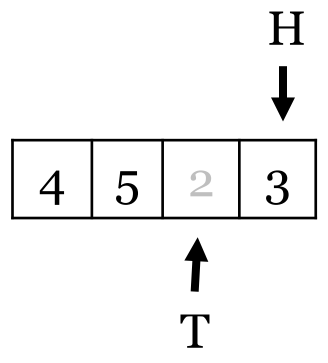


- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4
- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



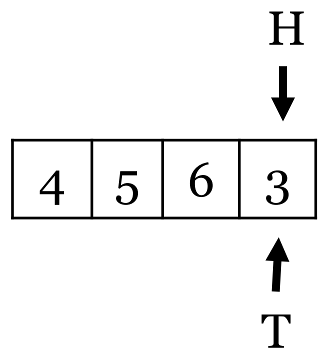
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



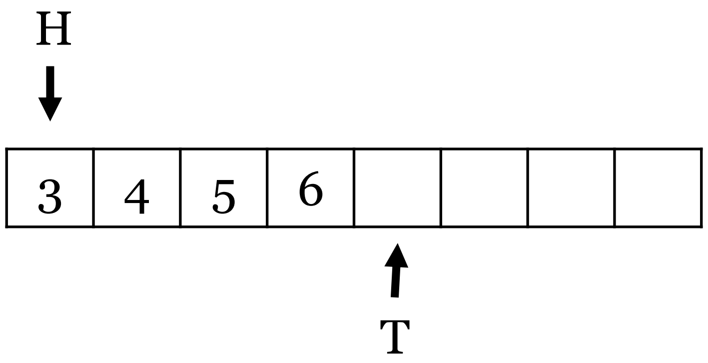
- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Queue implementations

- Linked list implementation
  - enqueue(): append
    - O(1)
  - dequeue(): delete *head*
    - O(1)
  - peek(): return *head*
    - O(1)
- Array implementation (usually preferred)
  - Two indexes: head and tail (plus capacity)
  - enqueue(): add after tail
    - O(1) "on average"
  - dequeue(): move head forward
    - O(1)
  - peek(): return `array[head]`
    - O(1)

Example: queue of size 4

- Enqueue(0, 1, 2)
- Dequeue() * 2
- Enqueue(3, 4)
- Dequeue()
- Enqueue(5, 6)



- Movement quirk
  - When head and tail reach capacity, start back over at 0
  - Queue is empty when head == tail
  - Queue is full when tail encounters head
    - Double capacity and copy

Stack/queue analysis

- Allow us to access elements in a specific order
- Used by many algorithms
- All operations O(1) worst case or amortized time complexity
- More restrictive than array/vector
  - Can only access one element at a time

Stack/queue exercise

- Use an STL stack to implement the `solve()` member function of the provided Maze class.
  - Use `push()`, `top()`, and `pop()`

- Relevant methods
  - `void getMoves(bool& up, bool& right, bool& left, bool& down);`
    - Returns whether you can move up, right, left, or down from current position
  - `bool move(direction_t dir);`
    - Moves UP, DOWN, LEFT, or RIGHT
    - Leaves a "breadcrumb trail"
      - `getMoves()` excludes previously visited directions
  - `bool move_back(direction_t dir)`
    - “Undoes” a move
    - Doesn't change visited status
  - `bool isSolved()`
    - Returns whether you’ve found the Maze exit

Set or dictionary

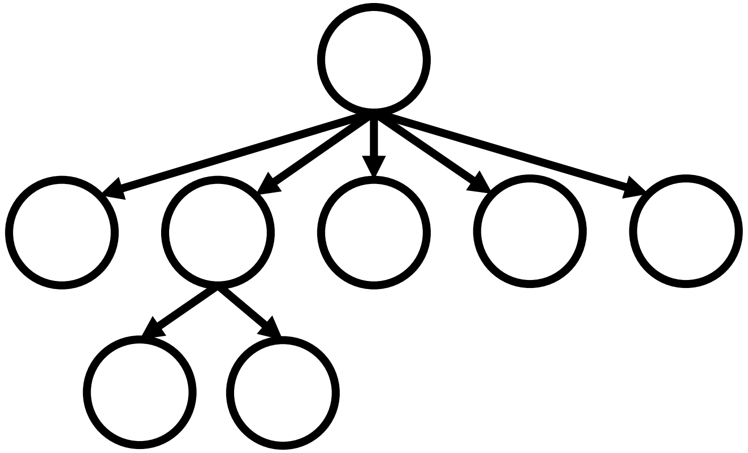
- Abstract data type for storing and retrieving values
  - *Main implementations:* balanced BSTs and hash tables

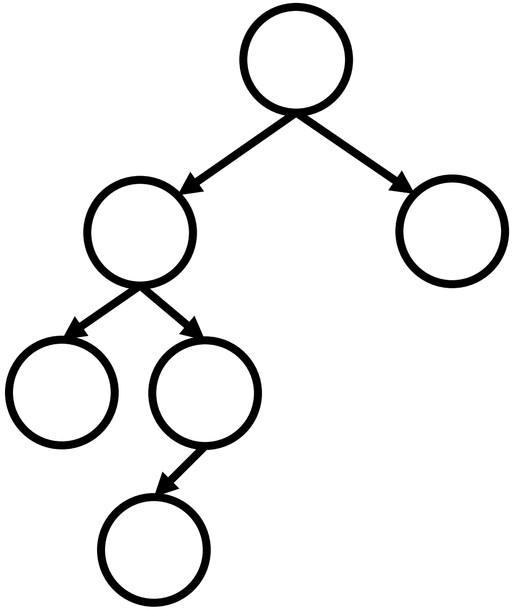
- **Primary operations**
  - *Search(x)*: returns the location of *x* in the set, or NIL if not contained
  - *Insert(x)*: adds *x* to the set
  - *Delete(x)*: removes *x* from the set

- **Additional operations**
  - *Build(data):* construct set from unsorted array
  - *Max(), Min()*: return the location of the largest/smallest element
  - *Successor(x), Predecessor(x)*: return the next largest/smallest element than *x*

Trees

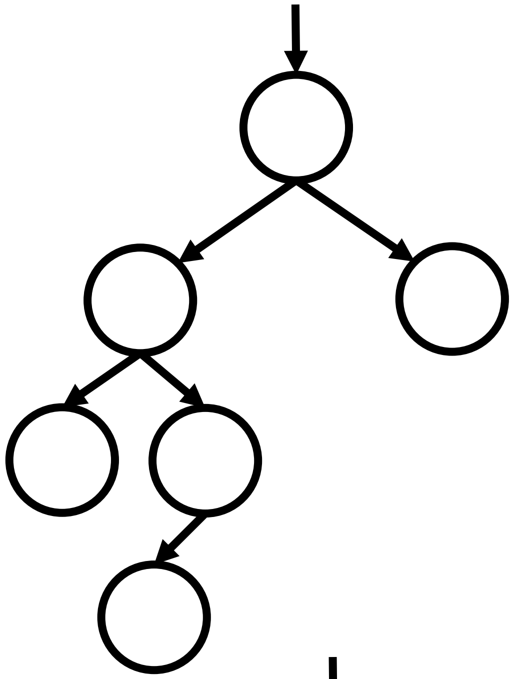
- Nonlinear linked data structure
- From graph theory: any connected acyclic graph
- Most trees in CS are *rooted*
  - Special vertex
  - Other vertices are defined in relation to root
  - Root has children, children have children, etc.
- *Leaf*: vertex with degree 1 (no children)
- *Level*: nodes at same distance from root
- *Height*: max level
- *Siblings*: vertices with same parent
- Binary tree
  - Tree where every node has at most 2 children
    - *left* and *right*
  - *m*-ary tree: every node has at most *m* children
  - *Full* binary tree: all non-leaf nodes have 2 children



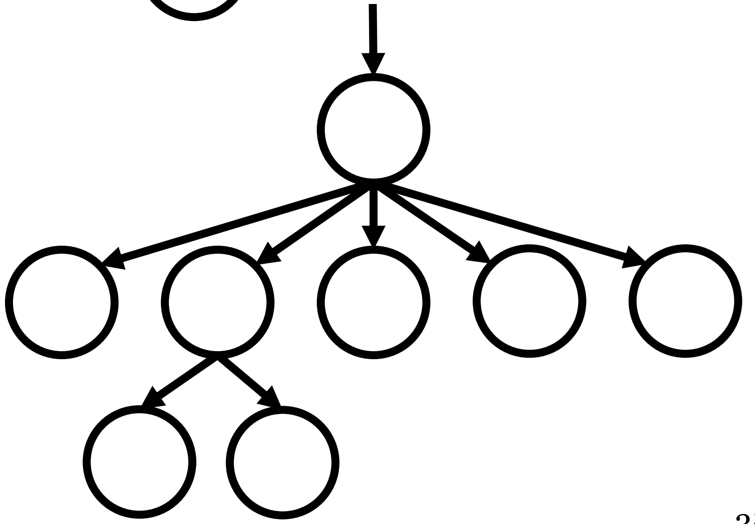


Tree implementation

- Often implemented similar to linked list
- BinaryTree
  - Data member: root
    - Optional: size, height
- BinaryTreeNode
  - Data members: data, left, right
    - Left and right null if not present
    - Optional: parent (null if root)

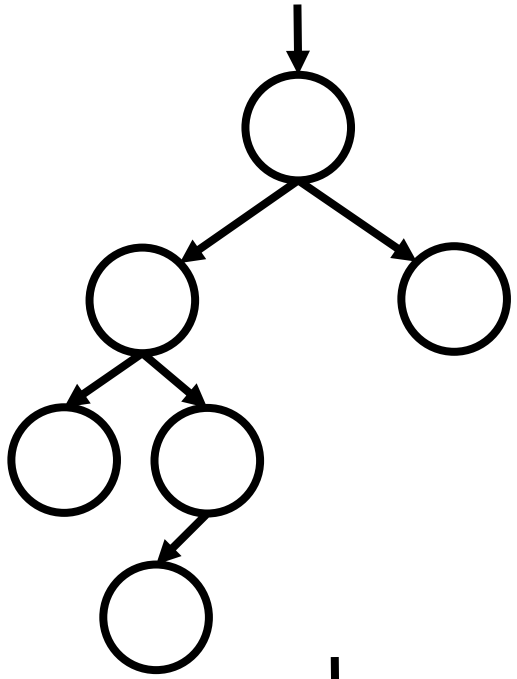


- *m*-ary tree: two main implementations
  - Main class the same
  - Implementation 1: node has array of children
    - Parent optional in both
  - Impl. 2: node has 2 pointers: firstChild and nextSibling

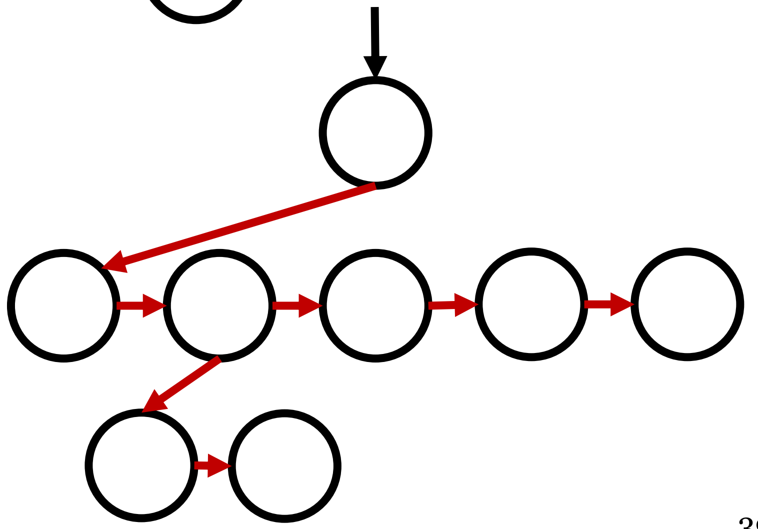


Tree implementation

- Often implemented similar to linked list
- BinaryTree
  - Data member: root
    - Optional: size, height
- BinaryTreeNode
  - Data members: data, left, right
    - Left and right null if not present
    - Optional: parent (null if root)

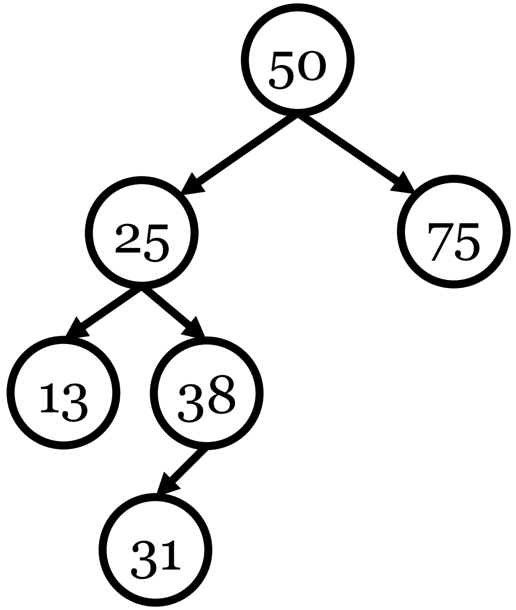


- *m*-ary tree: two main implementations
  - Main class the same
  - Implementation 1: node has array of children
    - Parent optional in both
  - Impl. 2: node has 2 pointers: firstChild and nextSibling
    - Effectively a binary tree



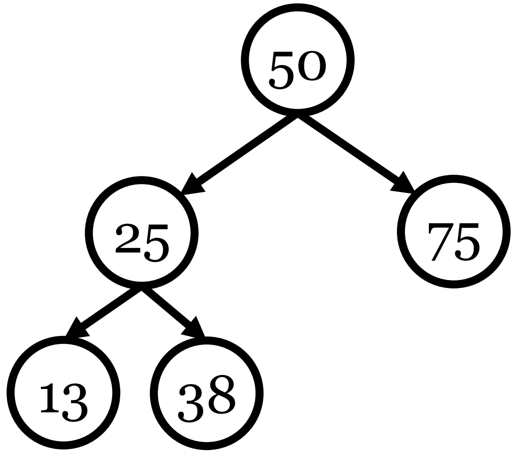
Binary search tree

- Important binary tree variant
- BST property:
  - All nodes to left are ≤
  - All nodes to right are ≥
  - Makes it easy to find values
- Search:
  - Start at root
  - If current node == target, return
  - Otherwise, if target < current, search left
  - Otherwise, search right
  - Repeat until you find the value or a null pointer
- Analysis
  - O(1) per recursive call
  - 1 node per level
  - O(h) time total (height h)



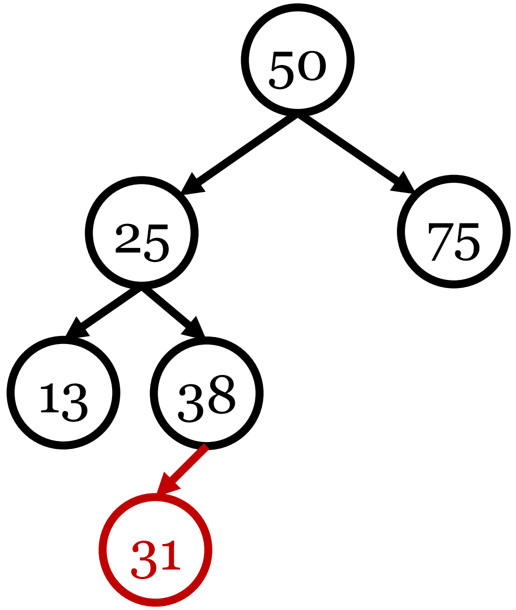
BST insert

- Also easy
  - Start at root
  - If ins < current node, move left
  - Otherwise, move right
  - Repeat until you find a null pointer
  - Insert new node at null pointer



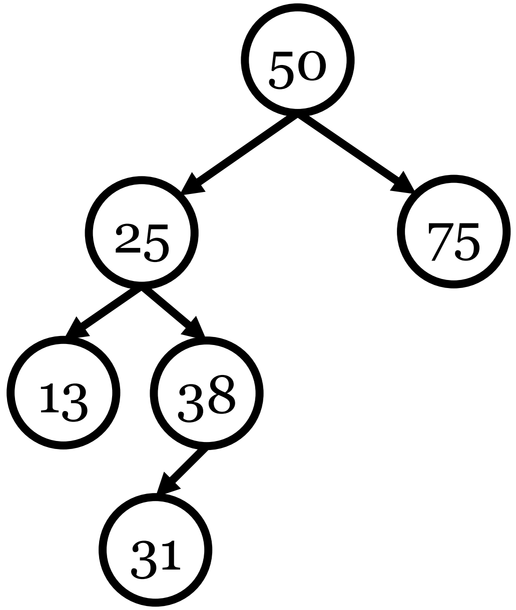
BST insert

- Also easy
  - Start at root
  - If ins < current node, move left
  - Otherwise, move right
  - Repeat until you find a null pointer
  - Insert new node at null pointer
- Analysis: same as search
  - O(1) / level -> O(*h*) total



BST remove

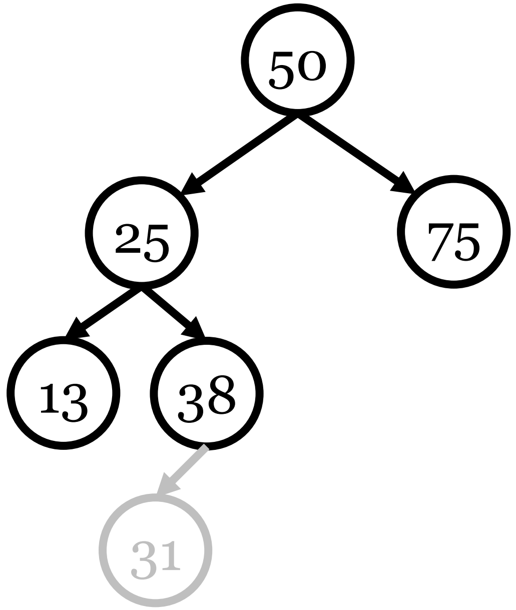
- More complicated!
- Step 1: search for node to delete
- Step 2: delete node
  - Case 1: no children (leaf)
    - Set parent's child to null
    - Delete node



- Step 3: decrement size
  - Update root if necessary

BST remove

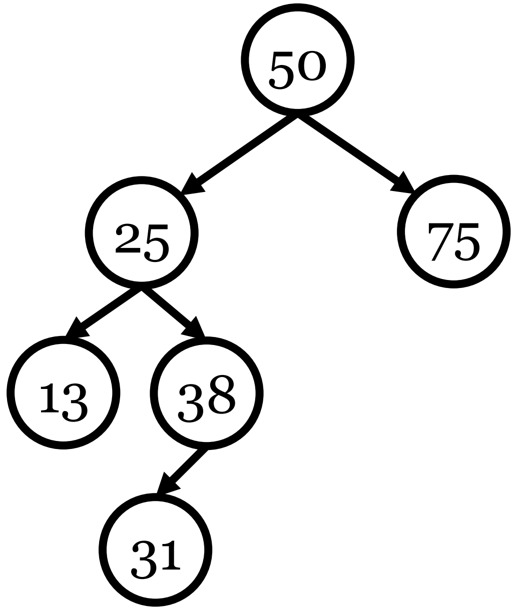
- More complicated!
- Step 1: search for node to delete
- Step 2: delete node
  - Case 1: no children (leaf)
    - Set parent's child to null
    - Delete node



- Step 3: decrement size
  - Update root if necessary

BST remove

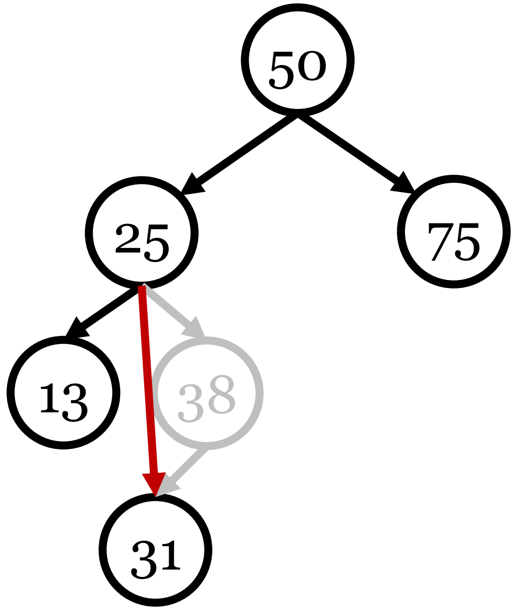
- More complicated!
- Step 1: search for node to delete
- Step 2: delete node
  - Case 1: no children (leaf)
    - Set parent's child to null
    - Delete node
  - Case 2: 1 child
    - Set parent's child to be child
    - Set child's parent to parent
    - Delete node



- Step 3: decrement size
  - Update root if necessary

BST remove

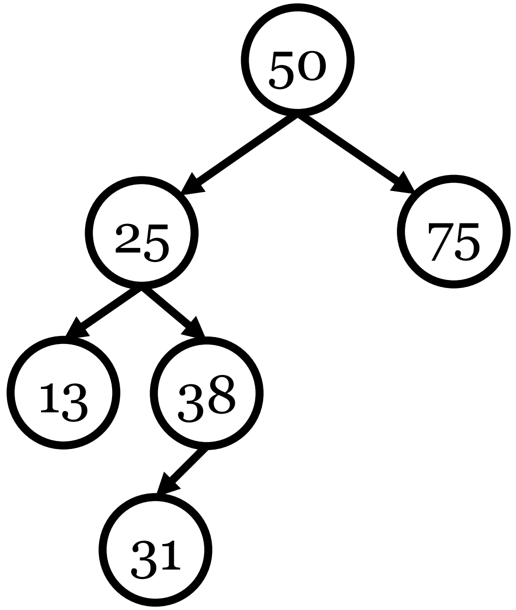
- More complicated!
- Step 1: search for node to delete
- Step 2: delete node
  - Case 1: no children (leaf)
    - Set parent's child to null
    - Delete node
  - Case 2: 1 child
    - Set parent's child to be child
    - Set child's parent to parent
    - Delete node



- Step 3: decrement size
  - Update root if necessary

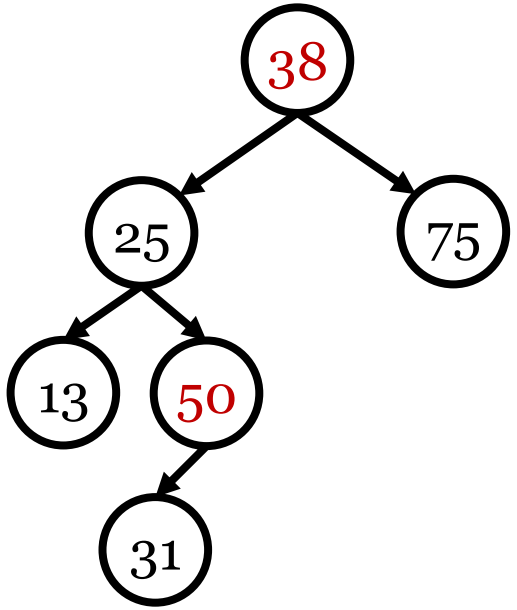
BST remove

- More complicated!
- Step 1: search for node to delete
- Step 2: delete node
  - Case 1: no children (leaf)
    - Set parent's child to null
    - Delete node
  - Case 2: 1 child
    - Set parent's child to be child
    - Set child's parent to parent
    - Delete node
  - Case 3: 2 children
    - Swap node with max(lhs)
    - Delete node at new location
- Step 3: decrement size
  - Update root if necessary



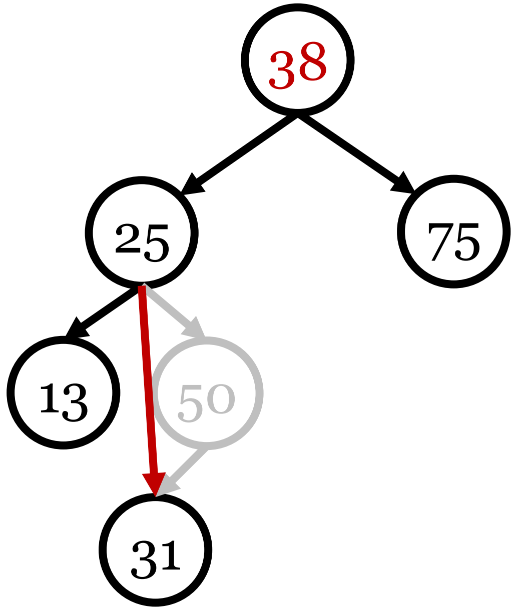
BST remove

- More complicated!
- Step 1: search for node to delete
- Step 2: delete node
  - Case 1: no children (leaf)
    - Set parent's child to null
    - Delete node
  - Case 2: 1 child
    - Set parent's child to be child
    - Set child's parent to parent
    - Delete node
  - Case 3: 2 children
    - Swap node with max(lhs)
    - Delete node at new location
- Step 3: decrement size
  - Update root if necessary



BST remove

- More complicated!
- Step 1: search for node to delete
- Step 2: delete node
  - Case 1: no children (leaf)
    - Set parent's child to null
    - Delete node
  - Case 2: 1 child
    - Set parent's child to be child
    - Set child's parent to parent
    - Delete node
  - Case 3: 2 children
    - Swap node with max(lhs)
    - Delete node at new location
- Step 3: decrement size
  - Update root if necessary
- Analysis: still O(h)
  - Case 3 has 0 or 1 children



Tree traversals

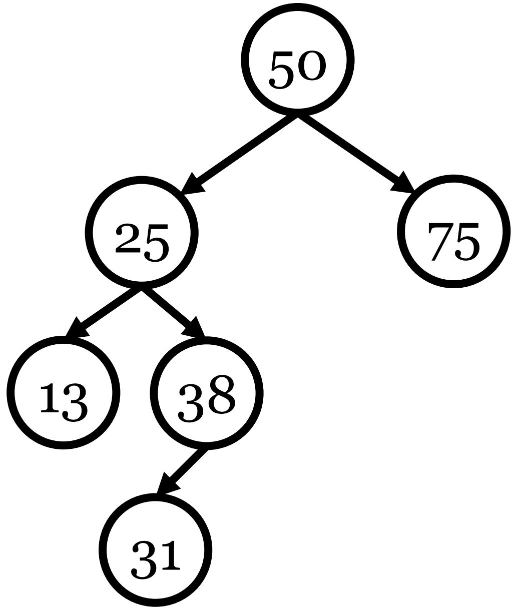
- Algorithms to visit every node in rooted tree
- 4 classic traversals
  - Pre-order
    - Process self
    - Process left tree
    - Process right tree
  - In-order
    - Process left, then self, then right
  - Post-order
    - Process left, then right, then self

  Recursive

  - Levelwise

  Queue-based

    - Start with root in queue
    - Dequeue: process, then enqueue children
    - Repeat until queue empty
- Analysis:
  - O(1) per node
  - O(n) total



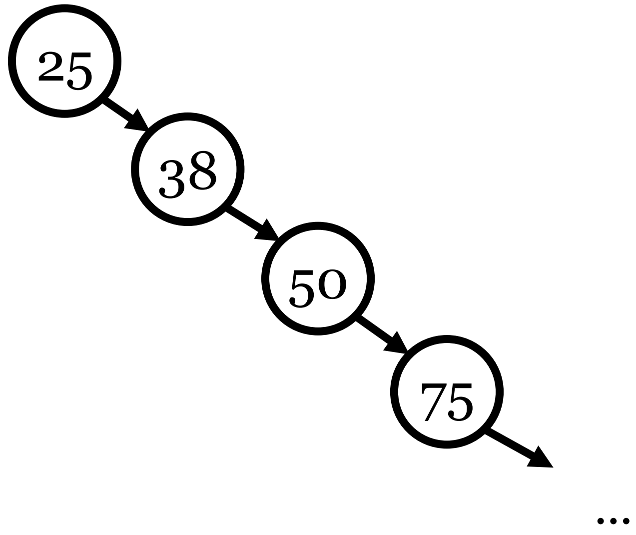

```
Pre-order:   50, 25, 13, 38, 31, 75
In-order:    13, 25, 31, 38, 50, 75
Post-order:  13, 31, 38, 25, 75, 50
Levelwise:   50, 25, 75, 13, 38, 31
```

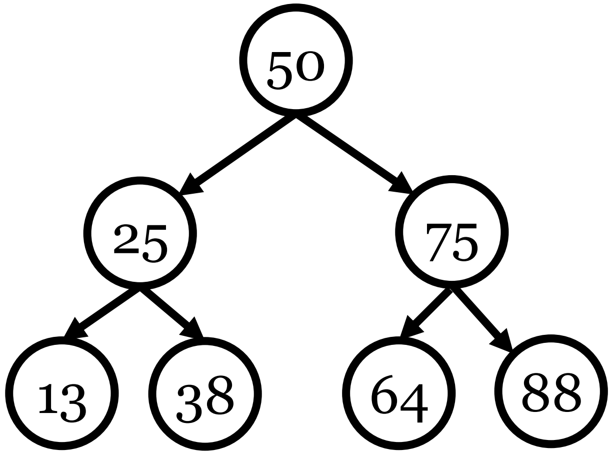
Tree height

- How tall can a BST be?

- **Worst case**
  - Degenerate tree
    - Only left or right children
    - Height = O(n)
  - Can happen if inserting in sorted order

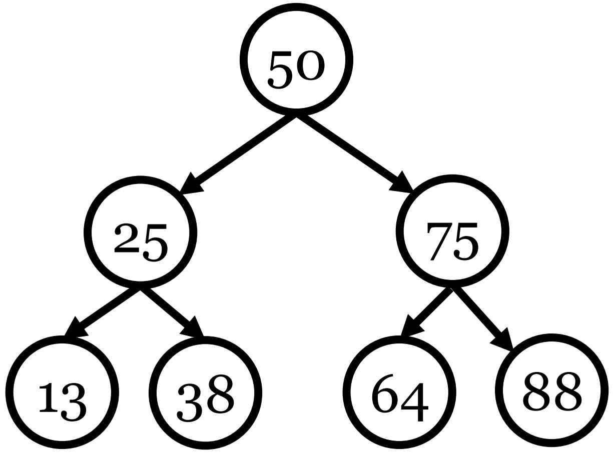
- **Best case**
  - Complete tree





Complete tree height

- **Theorem:** Complete trees have 2^i nodes on level i, for every level (except possibly bottom level)
  - Proof sketch:
    - $1 = 2^0$ on level 0
    - Every subsequent level has twice as many as previous



- # nodes in complete binary tree of height h: $\sum_{i=0}^{h} 2^i$
  - **Theorem:** $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
    - Proof by induction
      - *Exercise for the reader*
  - # nodes in complete tree w/ height h is $O(2^h)$
    - Height of tree with n nodes is $O(\lg n)$
- **Theorem:** half of all nodes in complete tree are leaves

BST complexity

	BST (best)	BST (worst)	Unsorted array	Sorted array
Search(x)	O(lg n)	O(n)	O(n)	O(lg n)
Insert(x)	O(lg n)	O(n)	O(1)	O(n)
Delete(x)	O(lg n)	O(n)	O(1)*	O(n)
Min()	O(lg n)	O(n)	O(n)	O(1)
Max()	O(lg n)	O(n)	O(n)	O(1)

* Swap to end and decrement size

- BST have great best-case complexity
- Worst case is worse than arrays!

Coming up

- Tree traversals
- Balanced BSTs
  - AVL trees

- **Recommended readings:** Chapter 2 (all except 2.2.1)
  - **Practice problems:** R-2.2, R-2.5, C-2.3, C-2.13, A-2.2