CPSC 221

January 8th - 1.1 Introduction

- Logistics
    - Instructor: Geoffrey Tien
    - Course coordinator: Jesse Wong (cpsc221-admin@cs.ubc.ca)
    - Syllabus is on Canvas
    - No lectures on Friday, substituted by videos on Tuesdays (MUST WATCH before Wednesday!)
- Course description
    - All programs manipulate data
        - gather, process, store, display
        - numbers, texts, images, sound
    - Programmers must decide how to store and manipulate data
    - Choices influence programs on
        - Execution speed
        - Memory requirements
        - Maintenance
    - Problem $\leftrightarrow$ Solution
        - Each solution can be described in detailed sequence of steps - **algorithm**
        - Each solution depends on the existence of certain hardware or organization of the collection - **data structure**
        - Each solution should produce the same result
            - Requires different amounts of time (in general, computing resources)
            - Comparison between solutions
- Data organization
    - Have some effect on the speed of program execution
- Measuring and Comparing Algorithms
    - "Good" algorithms

        - Produce the correct solution
        - Finish in a "reasonable" amount of time - **Time complexity**
        - Use a "reasonable" amount of system memory - **Space complexity**
    - Running time: $T(n) : \mathbb{Z}^0 \to \mathbb{R}^0$
    - n is the collection size

January 9th - 1.2: Reasoning About Code Intro

- Algorithm Analysis
    - Reason
        - Code correctness
        - Running time
        - Memory usage
    - Improvement made for alternative or better algorithms
- Analyzing algorithms
    -
```
int mystery(vector<int>& arr, int q) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == q) {
            return i;
        }
    }
    return -1;
}
```

      $arr.size() = n = 16$
    - What does it do?
        - Find the index at which the element is equal to integer q
        - If the desired integer is not in this array (not an element of this array), then return -1.
    - How long does it take?
        - Best case scenario
            - 1-element array
            - Big array + q is one of the first values
        - Worst case scenario
            - q is not in the array, and $arr.size()$ is large.

        - Average case scenario
            - It depends on more assumptions: real-life applications (e.g. 30% in array, 70% not in array; in-array: sorted vs. randomly distributed)
    - Getting $T(n)$
        - How many / which lines are executed?
            - We initialize the array and q
            - Initialize i
            - Increment i, n times
            - Compare i and $arr.size()$, $n+1$ times
            - Compare $arr[i]$ and q, n times
            - Return once
            - $1+1+(n+1)+n+n+1=3n+5$
        - Roughly, $T(n) \approx n$
- Asymptotic notation (Worst case analysis)
    - $T(n) \in O(f(n))$, if $\exists c, n_0$ such that $T(n) \leq c \cdot f(n)$, for $n \geq n_0$
    - We want to compare the "overall" runtime of the program's function against a simpler function
    - We want comparisons to be valid for all sufficiently large inputs, we ignore relatively small sample sizes
    - We can demonstrate that $T(n) \in O(2^n)$, but it is not useful, as our algorithm cannot be worse than one of the worst algorithms ever.
    - The "smallest" reference function is the **tight upper bound**
    - $T(n) \in \Omega(f(n))$, if $\exists c, n_0$ such that $T(n) \geq c \cdot f(n)$, for $n \geq n_0$
    - $T(n) \in \Theta(f(n))$, if $T(n) \in O(f(n)) \land T(n) \in \Omega(f(n))$
    - $T(n) \in o(f(n))$, if $\forall c \in \mathbb{Z}^+, \exists n_0$ such that $T(n) < c \cdot f(n)$, for $n \geq n_0$
    - $T(n) \in \omega(f(n))$, if $\forall c \in \mathbb{Z}^+, \exists n_0$ such that $T(n) > c \cdot f(n)$, for $n \geq n_0$

January 10th - 1.3: Asymptotic Reprise

- Logistics
    - First examlet on Jan. 25/26 - self registration open next week
    - HW1 to release this week - due Jan. 26 at 22:00
- Asymptotic analysis
    - Hacks - Running time approximation
        - Eliminate low-order terms

        - $4n + 5 \to 4n$
        - $0.5n \log n - 2n + 7 \to 0.5n \log n$
        - $2^n + n^3 + 3n \to 2^n$
    - Eliminate constant coefficients
        - $4n \to n$
        - $0.5n \log n \to n \log n$
        - $n \log (n^2) = 2n \log n \to n \log n$
- Typical growth rates in order
    - Constant: $O(1)$
    - Logarithmic: $O(\log n)$ $\left(\log_k n, \log(n^2) \in O(\log n)\right)$
    - Poly-log: $O\left((\log n)^k\right)$
    - Sublinear: $O(n^c)$ (c is a constant, $0 < c < 1$)
    - Linear: $O(n)$
    - Log-linear: $O(n \log n)$
    - Superlinear: $O(n^{1+c})$ (c is a constant, $0 < c < 1$)
    - Quadratic: $O(n^2)$
    - Cubic: $O(n^3)$
    - Polynomial $O(n^k)$ (k is a constant) "tractable"
    - Exponential $O(c^n)$ (c is a constant > 0) "intractable"
-
    - Factorial: $O(n!)$
- Examples
    - $n^3 + 4 \in O(n^4), \notin \Theta(n^4)$
    - $n^3 + 4 \in \Omega(n^2), \notin \Theta(n^2)$
    - $n^4 + 3n^3 \in O(n^4), \notin o(n^4)$
- Dominance
    - $T(n) = 2n^2 + 600n + 60000$
        - Up to $n = 100$, the constant term dominates
        - Up to $n = 300$, the linear term dominates
        - For $n > 300$, the quadratic term dominates
    - Which is faster $n^3, n^3 \log n$?
        - Split up and use dominance relationship
- Analyzing nested loops

- ```
  bool hasDuplicate(int arr[], int size) {
      for (int i = 0; i < size - 1; i++) {
          for (int j = i + 1; j < size; j++) {
              if (arr[i] == arr[j]) {
                  return true;
              }
          }
      }
      return false;
  }
  ```

    - Outer loop: $n-1$ times, $O(n)$; Inner loop: $n-1$ times, $O(n)$?
I	# INNER LOOP OPRNS
0	n-1
1	n-2
2	n-3
...	...
n-2	1
    - $$\sum_{k=1}^{n-1} k = \frac{n^2 - n}{2}$$
    - Still $O(n^2)$.
- Typically
    - Nested loops: multiply complexities
        - $O(n) \times O(m) \in O(m \cdot n)$
    - Sequential loops: add complexities
        - $O(n) + O(m) \in O(\max(m, n))$
- How loop variable changes is **critical** as well
    - ```
      void candyapple(int n) {
          for (int i = 1; i < n; i *= 3) {
              cout << "iteration: " << i << endl;
          }
      }
      ```

        - $T(n) = \log_3(n) \in O(\log n)$

- ```
  void caramelcorn(int n) {
      for (int i = 0; i * i < 6 * n; i++) {
          cout << "iteration: " << i << endl;
      }
  }
  ```

    - $T(n) = \sqrt{6n} \in O(\sqrt{n})$
- For loop executions
    - Determine the range of loop variable
    - Determine how many elements will be "hit"
- Code analysis
    - Single operations: constant time
    - Consecutive operations: sum operation times
    - Conditionals: condition check time plus branch time

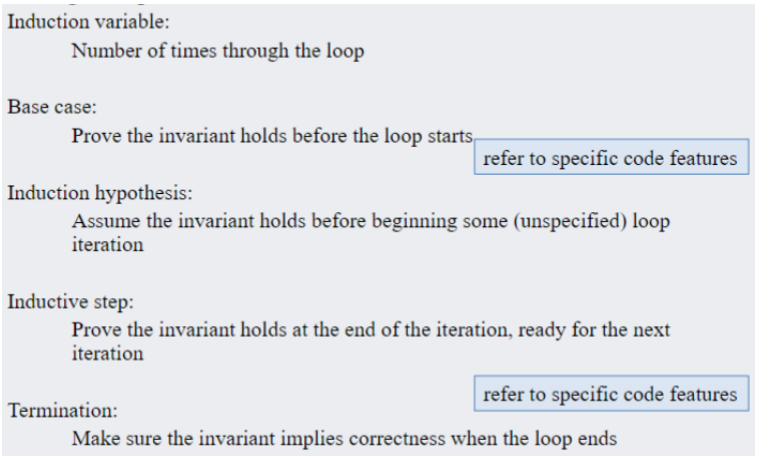
Jan. 15th - 2.1: Correctness

- Proving correctness: convincing ourselves that an algorithm does what it is supposed to do
    - ```
      int someMethod(vector<int>& arr, int a) {
          int m = a;
          for (int i = a + 1; i < arr.size(); i++) {
              if (arr[i] < arr[m]) {
                  m = i;
              }
          }
          return m;
      }
      ```

- Induction
    - Loop invariant
        - Properties of the algorithm or structure that are always true at particular points in the program
        - In a loop, it may be violated briefly, but the rest of the loop should fix the properties for the next iteration
        - Must use features of the code to demonstrate that any violations of the loop invariant are restored

- For the code segment above
    - Good name? (Function)
        - Index of minimum element
    - Running time: $O(n)$

arr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	32	11	73	21	29	86	81	92	57	61	64	15	79	44	7	45

    -
        - What are possible values of i, given $a == 5$?
            - From $a + 1$ to $arr.size() == 16$
        - Suppose we pause our program at the **start** of some iteration
            - We pause at $i == 9$, we processed 6, 7, 8
            - In this case, $m == 8$
            - If we complete execution, $m == 14$
            - m would then be the index of the smallest element **so far**.
    - Invariant property
        - For any iteration i in $\{a + 1, n\}$, m is: the index of the smallest element from a to $i - 1$ (at the beginning of iteration i)
    - Proof
        -


    - Specific proof
        - Invariant property: m is the index of the minimum element in $arr[a \ldots i-1]$ at the start of iteration i, for all $i \in [a+1, \ldots, n]$
        - Base case: $i = a + 1$
            - Before first iteration, m contains the index of the smallest element of $arr[a \ldots a]$, which is a
            -
```
int m = a;
```

        - Referred back to code
    - The loop invariant holds in the base case
- Induction step: $i = k$
    - Assume loop invariant still holds (inductive hypothesis)
    - Case 1: $arr[k] < arr[m]$
        - m is set to k, because $arr[k]$ is less than any element from $arr[a \dots k-1]$.
        - Loop invariant holds after processing k
    - Case 2: $arr[k] \geq arr[m]$
        - No action is taken, m is then the index of the smallest element from $arr[a \dots k]$, after processing k
        - Loop invariant holds
- Termination: $i == arr.size()$
    - By invariant, m is the index of the smallest element from $arr[a \dots arr.size() - 1]$, the subarray
    - How do we know that the loop ends?
        - `i < arr.size()`
        - i increases, therefore i will exceed $arr.size() - 1$ eventually.
- ```
  void someMethod(vector<int>& arr) {
      for (int i = 0; i < arr.size(); i++) {
          int min = indexOfMin(arr, i);
          swap(arr[i], arr[min]);
      }
  }
  ```

Jan. 16th - 2.2: Selection Sort

- Algorithm analysis - Simple sorting
    - Example: sort a random hand of cards (algorithmically)
    - Selection sort: Repeatedly finds the smallest item - we *select* the smallest item to put into place
  ```

        - Sorted part + Unsorted part
        - Repeatedly swap the first unsorted item with the smallest unsorted item
        - From index 0 to index $n-1$
    - Running time: $T(n) \in \Theta(n^2)$
    - ```
      void selectionSort(vector<int>& arr) {
          for (int i = 0; i < arr.size(); i++) {
              int min = indexOfMin(arr, i);
              swap(arr[i], arr[min]);
          }
      }
      ```
        - Main loop: executed n times, if $arr.size() == n$
        - indexOfMin: from previous lecture, executed about $n - i - 1$ times each iteration
        - In total, there are about $\frac{n(n-1)}{2}$ times, $T(n) \in O(n^2)$
        - Swap: in $O(n)$
    - Neither of the loop nor the swapping is affected by the organization of the input
    - Best case: no way to break loop early

      Worst case: no way to break loop early

      Average case: $\Theta(n^2)$
    - Reason to use selection sort
        - In extremely limited systems where reads are cheap but writes are expensive
        - Only at most $O(n)$ swaps (writes)
        - Small program $\approx$ Small memory when compiling
    - Space complexity (memory needed to run on top of inputs)
        - Other than input, we need a loop variable i, and min, with number of variables from indexOfMin
        - Amount of variable at most a constant: $O(1)$
- Loop-invariant proof
    - Loop's purpose: place the smallest item from the unsorted subarray at the end of the ordered subarray
    - Consider a partial iteration progress
        - Sorted + Unsorted = Original (different permutation)
      ```

        - Any sorted are smaller than any unsorted elements
    - Invariant property
        - Before iteration i, $arr[0 \dots i-1]$ contains the i smallest elements of arr in ascending order; $arr[i \dots arr.size()-1]$ contains unordered largest elements of arr
    - Base case: $i = 0$
        - Before this iteration, the invariant states that $arr[0 \cdots -1]$ contains 0 smallest elements of arr in ascending order
        - True
    - Inductive step: $i = k$
        - Assume $arr[0 \dots k-1]$ contains k smallest elements of arr in ascending order
        - After iteration, $arr[0 \dots k]$ contains the $k+1$ smallest elements of arr in ascending order, as the smallest of the unordered subarray is placed at the end of the already sorted subarray.
        - The invariant property still holds
    - Termination: $i == arr.size()$
        - i increases eventually reaching $arr.size()$
        - By loop invariant, $arr[0 \dots arr.size()0-1]$ contains the $arr.size()$ smallest elements of arr in ascending order
        - Correct!

Jan. 17th - 2.3: Insertion Sort

- ```
  void someMethod(vector<int>& arr, int p) {
      // assumes arr[0...p-1] are in sorted order
      int temp = arr[p];
      int j = p;
      while (j > 0 && arr[j-1] > temp) {
          arr[j] = arr[j-1];
          j--;
      }
      arr[j] = temp;
  }
  ```

    - "Slides" the element at p in order with the sorted order.
    - Running time: $O(n)$ max; $\Omega(1)$ min
    - Loop invariant?
  ```

- $arr[0 \dots j-1]$ is sorted (untouched elements), and $arr[j+1 \dots p]$ is sorted (shifted elements)
- $arr[0 \dots j-1] \cup arr[j+1 \dots p] \cup temp$ form the original elements of $arr[0 \dots p]$

- Insertion sort
  - Divides array into sorted and unsorted parts
  - Sorted array expands one element at a time
    - Find the correct placement index
    - Move the elements one position to make space
    - **Insert** element into correct place
  - ```
    void InsertionSort(vector<int>& arr) {
        for (int i = 1; i < arr.size(); i++) {
            slide(arr, i);
        }
    }
    ```

    - Main loop: $n-1$ times
    - **slide**: $\Omega(1)$ or $O(n)$
    - Worst case: $O(n^2)$

      Search placement index: $\frac{n(n-1)}{2}$

      Slide times: $\frac{n(n-1)}{2}$

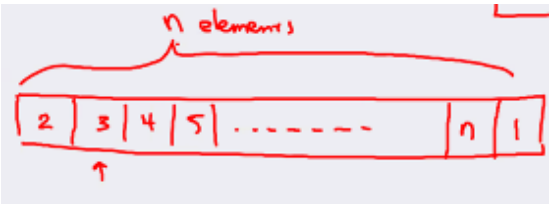
      The array is descending

    Best case: $\Omega(n)$

      The array is already sorted;

      Only $n-1$ comparisons, but no movement is needed.

      Another linear case is an array of such:

      From index 1 to n-2, each costs 1 action; The last element will be compared in $O(n)$, so $(n-2) \times O(1) + 1 \times O(n) \in O(n)$.

      Thus, if there are $O(1)$ elements out of place, then the best case occurs.

    - Average case
      - On average, at index k, we need to perform $\frac{k}{2}$ comparisons
    ```

        - So for every index, we need to perform on average $\frac{n(n-1)}{4}$ comparisons
        - So closer to worst case
    - In summary, it is good for data that is almost sorted, thus it is advantageous to use it for **maintenance**.
- Algorithm comparison
    -

NAME	BEST	WORST	AVERAGE	MEMORY
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

- Proof
    - $arr[0 \dots i-1]$ is sorted, and $arr[0 \dots size()-1]$ contains the original elements
    - Base case, $i == 1$
        - Index from 0 to 0 is sorted, ✓
    - Inductive step, $i == k$
        - By hypothesis, $arr[0 \dots k-1]$ are in sorted order
        - After iteration, $arr[k]$ is placed appropriately by slide function
        - So after iteration, $arr[0 \dots k]$ is sorted
    - Termination, $i == size()$
        - Then $arr[0 \dots size()-1]$ is sorted, and it contains all the original elements, ✓

Jan. 22nd - 3.1: Introduction to Memory

- C++ and memory
    - Memory: Long street with numbered mailboxes
        - Mailbox number: memory address
        - Letter: memory value
        - Variables in the stack can be assigned names
            - `double bank_balance = 21.03;`
    - 4 area of memory
        - Code: compiled machine code instructions
        - Stack: small area for local variables/function parameters (automatically managed by system), on the scale of several MB

        - Heap: large area for on-demand memory needs (explicitly managed by program code)
        - NULL: mailbox number 0
- Calling functions in C++
    -
```
int sum(int x, int y) {
    return x + y;
}

int main() {
    int a = 5;
    int b = 7;
    int result = sum(a, b);
    return 0;
}
```

        - Actual parameters
            - Value/variable specified by the function **caller**
        - Formal parameters
            - Variables found in the function **header/signature**
        - Formal parameters must match with actual parameters in **order, number, and data type**
        - Function parameters
            - Most parameters in C++ are **passed by value** (the value of the actual parameter is *copied* to the formal parameter)
            - Actual parameters and formal parameters are **different in memory**
            - If the value of the formal parameter is changed, the value of actual parameter is NOT changed.
            - As soon as the function call *returned*, the formal parameters will be released from memory.
    -
```
double square(double x) {
    return x * x;
}

double circleArea(double radius) {
    double pi = 3.1415;
    double sq_r = square(radius);
    return sq_r * pi;
}
```

```
int main() {
    int r = 3;
    double area = circleArea(r);

    return 0;
}
```

- Swap functions

```
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5;
    int b = 7;
    swap(a, b);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
}
```

    - Call stack progression
        - a = 5; b = 7
        - a = 5; b = 7; (x = 5; y = 7)
        - a = 5; b = 7; (x = 5; y = 7; temp = 5)
        - a = 5; b = 7; (x = 7; y = 7; temp = 5)
        - a = 5; b = 7; (x = 7; y = 5; temp = 5)
        - a = 5; b = 7
        - **a, b** are not swapped!
    - We can fix this problem using **call-by-reference**
        - Add & symbol after the parameter's data type in the signature
        -
        ```
        void swap(int& x, int& y) {
            // same as above
        }
        ```

- Dynamic memory

        - Variables declared in a function only exists within the scope of that function
        - We require objects and variables to persist beyond a function's lifetime
        - We need local variables that refer or point to the dynamically allocated memory
            - *Pointers*
- Addresses and pointers
    - Memory address: location in memory where a given variable/identifier stores its data
    - Pointer
        - Stores **address** rather than a value
        - The stored address is used to find a value elsewhere in memory
        - Declaration

```
datatype* identifier;

int* n_ptr;
int * n_ptr;
int *n_ptr;
```

            - n_ptr is a pointer to an **int**, but itself is not an **int**
    - Pointers can be assigned the address of an existing variable
        - Use &:

```
int a = 23;
int* n_ptr = &a;
```

        - The value which a pointer points to can be accessed by **dereferencing** the pointer, using the * operator

```
int b = *n_ptr
```

    - Passing pointer parameters allows the function to access and modify the actual parameter
    - "You can keep adding levels of pointers until your brain explodes or the compiler melts - whichever happens soonest."
    - Pointers and Dynamic memory

- **new** keyword allocates space in dynamic memory and returns the first address of the allocated space
- **delete** releases the memory at the address referenced by its pointer variable
    - delete[] is used to release memory allocated to array variables

Jan. 23rd - 3.2: Linked Lists

- Pointers and Dynamic Memory
    -
```
int a = 5;
int* b = new int;
int* c = &a;
*c = 4;
int** d = &b;
int* e = new int[a];
**d = 3;
int* f = new int[*b];
delete b;
delete e; // causes a memory leak
delete[] f;
```

- **delete**: releases the memory of the object in the heap at the reference (memory address)
- Dangling pointers
    -
```
void fun() {
    int* i = new int;
    *i = 5;
    delete i;

    // i = NULL

    cout << *i << endl;
}
```

- If the pointer continues to refer to the deallocated memory, it will behave unpredictably when dereferenced - resulting in a dangling pointer
- Setting the pointer to NULL can solve this issue

- Memory Leaks
    - Losing access to allocated memory space so that that space can no longer be referenced or freed
    - The remains will be marked as allocated for the lifetime of the program

- Linked Lists
    - Nodes
        - A dynamic data structure that consists of nodes linked together
        - A **node** is a data that contains: data & the location of the next code
        - ```
          template <class LIT>
          struct Node {
              LIT data;
              Node* next;
              Node(LIT ndata, Node* nx = NULL): data(ndata),
          next(nx) {
                  //
              }
          }
          ```
            - template: Allow our containers to store any data type (at time of declaration)
            - struct: Like a class definition where all members are public by default
    - Node pointer
        - A node contains the address of the next node
        - Nodes are created in dynamic memory, so their memory are not in sequence
        - Data attribute of a node depends on what the node intends to store
    - A linked list is a **chain** of nodes
    - Attributes of a particular node can be accessed
        - using . operator

          ```
          Node<int> nd;
          nd.data = 5;

          Node<int>* p = nd.next;
          (*p).data = 5;
          ```
          ```

- using -> arrow operator as shorthand for pointer types (equivalent to dereferencing and using dot operator)

```
Node<int>* q = *((*p.next)).next;
Node<int>* r = q->next->next;
```

- Constructing a linked list
  - ```
    Note<int>* a = new Node<int>(7, null);
    a -> next = new Node<int>(3, null);
    ```

- Traversing a linked list
  - ```
    // Start of the linked list
    Node<int>* p = a;

    // Referencing the next element in list
    p = p -> next;

    // Stop when p -> next is NULL
    ```

- Insertion
  - Singly-linked lists requires only updating the next node reference of the preceding position
  - ```
    Node<int>* b = new Node<int>(17, p -> next);
    p -> next = b;
    ```

- Deletion
  - Remove a node by updating the pointer of the preceeding node
  - ```
    p -> next = b -> next;
    delete b; // remove the node from the heap
    b = NULL; // prevent dangling pointer
    ```

Jan. 24th - 3.3: Linked List (Cont.d)

- Features of linked lists
  - Linkage
    - Singly-linked: every node has a link to the next node in sequence
    - Doubly-linked: every node has a link to the previous/next node in sequence

- Termination
    - NULL termination: the last node points to NULL
    - Circular termination: the last node points to the head pointer
    - Sentinels
        - e.g. Doubly-linked w/ sentinels, head + tail nodes
        - head -> sentinel node (no data) <-> first data node <-> ... <-> last data node <-> sentinel node <- tail
- Access/Entry
    - Front/head pointer:

```
Node<int>* a = new Node<int>(5, NULL)
```

    - Tail/back pointer
- Linked list variations
    -

```
template <class LIT>
class LinkedList {
    private:
        Node* head;
        int length;
    public:
        LinkedList();
};
```

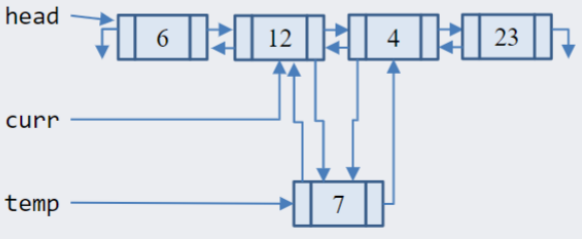
        - Suppose we need to access the last node of the linked list:

```
Node<int>* p = head;
if (p != NULL) {
    while (p -> next != NULL) {
        p = p -> next;
    }
    p -> next = new Node<int>(some_data,
NULL);
}
```

        - For a list containing n elements, traversing the list cost $O(n)$.
        - We can also give ourselves a tail pointer, maintained during operation at the back of the list
            - This now perform insertion at the end in $O(1)$ time

        - Removal from the back of the list is different
            - The last node can be released from memory, but we need to set the previous node to point to NULL, and handle the dangling pointer
            - This situation, we need to access from the head
- Doubly-linked list
    - Node definition with an additional pointer
        - One pointer points to the next; the other pointer points to the previous
        - Now front traversal and back traversal are now possible
    - ```
      template <class LIT>
      struct Node {
          LIT data;
          Node* prev;
          Node* next;
          // etc
      }
      ```
    - After some specified node

      ```
      Node<int>* curr, * temp;
      ... // use a loop to move curr into place
      temp = new Node<int>();
      temp->data = 7;
      temp->prev = curr;
      temp->next = curr->next;
      curr->next->prev = temp;
      curr->next = temp;
      ```

    - Removal
        - ```
          Node<int>* curr;
          ... // Move curr to the node to be removed
          curr -> next -> prev = curr -> prev;
          curr -> prev -> next = curr -> next;
          delete curr; // release the object from memory
          curr = NULL; // prevent dangling pointers
          ```
- Linked list are recursive

- Iteration is convenient for some list operations (traversal, insertion, removal)
- However, consider printing the contents of a singly-linked list in reverse
    - By iteration, the run time is in $O(n^2)$
    - By recursion

```
template <class LIT>
void PrintReverse(Node<LIT>* curr) {
    if (curr != NULL) {
        if (curr -> next != NULL) {
            PrintReverse(curr -> next);
        }
        cout << curr -> data << endl;
    }
}
```

    - Running time
        - $$T(n) = c + T(n-1)$$
        - This is in $O(n)$
- What about printing in reverse, but only the data at even indices (0, 2, ...)
    -

```
template <class LIT>
void PrintReverseOdds(Node<LIT>* curr) {
    // no items in the list
    if (curr == NULL) {
        // do nothing
    }

    // one item in the list
    if (curr != NULL && curr -> next == NULL)
{
        cout << curr -> data << " ";
        return ;
    } else { // > 1 item in list
        PrintReverseOdds(curr -> next ->
next);
        cout << curr -> data << " ";
        return ;
    }
}
```

Jan. 29th - Week 4.1: Stack Abstract Data Type (Description Implementations)

- Stacks in practical usage
    - Function call stack

```
main() {
    CircleArea() {
        Sq()
    }

    Distance() {
        Sq()
        Sq()
        sqrt()
    }
}
```

    - 4 5 + 7 2 - * 3 - (Polish notation)
    - ([( ) ({ })] [({ [ ] } {})])
- Stack ADT
    - ADT describes a data collection and the operations that can be performed on the collection
        - Effects of the operations are well-understood
        - **How** the operations are implemented are not described
    -

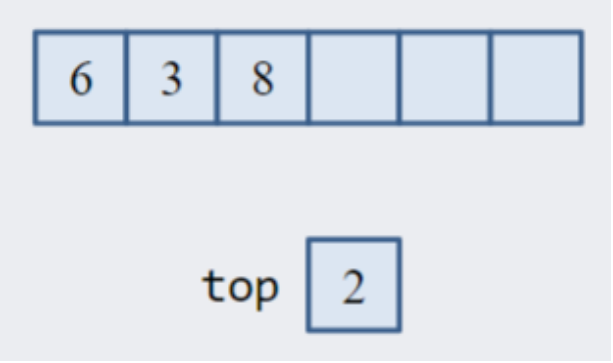
```
template <class LIT>
class Stack {
    public:
        Stack();                    // copy,
destructor, etc
        bool IsEmpty() const;
        void Push(const LIT & e);   // Insert at the
top of stack
        LIT Pop();                  // Remove and
return from top of stack
    private:
        // ???
};
```

    - **Last in, first out**

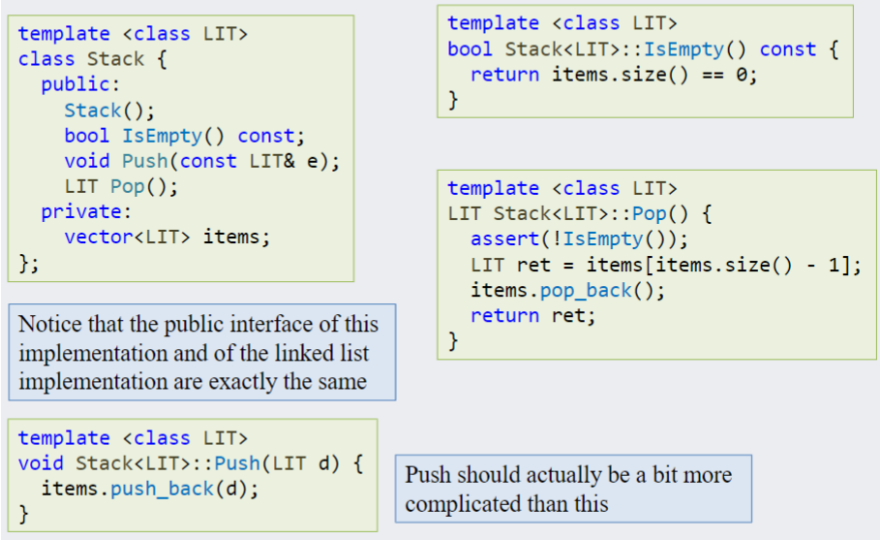
- Implementation
    - Singly-linked list (null terminated, head pointer)
        - Make the front of the list serve as top of the stack
        - All operations can be done in $O(1)$ time
        - ```
          template <class LIT>
          class Stack {
            public:
              Stack();
              bool IsEmpty() const;
              void Push(const LIT& e);
              LIT Pop();
            private:
              struct Node {
                LIT data;
                Node* next; };
              Node* top;
              int size;
          };
          ```

          ```
          template <class LIT>
          bool Stack<LIT>::IsEmpty() const {
            return top == nullptr;
          }
          ```

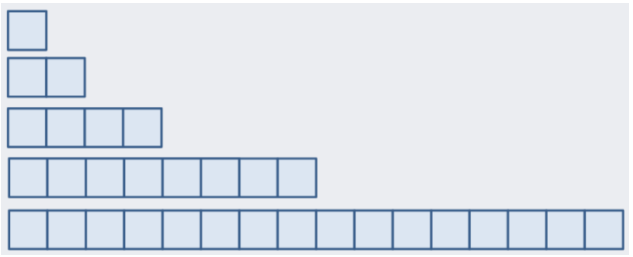
          ```
          template <class LIT>
          LIT Stack<LIT>::Pop() {
            assert(!IsEmpty());
            LIT ret = top->data;
            Node* temp = top;
            top = top->next;
            delete temp;
            return ret;
          }
          ```

          ```
          template <class LIT>
          void Stack<LIT>::Push(LIT d) {
            Node* newnode = new Node(d);
            newnode->next = top;
            top = newnode;
          }
          ```
        - If the back of the list is the top of the stack
            - The operations are done in $O(n)$ time
            - If we have a tail pointer, **Push()** is in $O(1)$, but **Pop()** is in $O(n)$.
            - If we have a doubly-linked list, both operations are again $O(1)$
    - Array
        - Make the most recently occupied index the top of stack
        - 

          ```




- 
    - push_back(d): is not in $O(1)$. It is more complicated than that
- Array resizing
    - If the array is filled up, the elements of the original array must be copied to a new location
    - Assume each resize adds 2 more empty spaces (Start with arr[2])
        - insert 2 elements
        - want to add 2 more
        - 2 existing elements to copy at $O(1)$ each, then 2 more space are added
        - wants to add 2 more
        - 4 existing elements to copy at $O(1)$ each, then 2 more space are added
        - Over a sequence of $n$ push operations, we have $n$ insertions at $O(1)$ + $(n-2)$ copied + $(n-4)$ copied + ... + 2 copied $\in O(n^2)$
        - Average cost per push operation is $O(n)$
    - Assume each resize is to double the size (Start with arr[1])
        - insert 1 element
        - want to add 1 more
        - 1 existing element to copy at $O(1)$, then 1 more space is added
        - insert 1 element
        - want to add 2 more
        - 2 existing ...

- 
- Over a sequence of $n$ push operations, we have $n$ insertions + $\frac{n}{2}$ copies + $\frac{n}{4}$ copies + $\ldots$ + 2 copies + 1 copy $\in O(n)$
- Average cost per push operation is $O(1)$

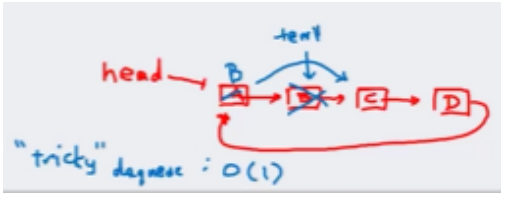
- Summary
    - Linked list: $O(1)$
    - Array-based implementation
        - Pop: $O(n)$, Push: $O(n)$
        - Cost over $O(n)$ is amortized for an *average* of $O(1)$ per push
        - Why array? Better cache performance

## Jan. 30th - 4.2: Queue ADT

- Queues
    - Items are inserted at the back and removed from the front
    - First in first out
    - Applications
        - Server requests
            - Instant messaging servers queue up incoming messages
            - Database requests
        - Print queues
        - OS use queues to schedule CPU jobs
        - Waiting list
        - Algorithmic implementations
- Operations
    - Implement at least first two
        - Enqueue: insert at the back
        - Dequeue: Remove from the front
        - Peek: return the item at the front of the queue without removing

        - Is_Empty: Check if the queue contains any items
    - Assume that these operations will be implemented efficiently
        - In $O(1)$
- List queue
    - Null-terminated, singly-linked list, head/tail pointers
    - Head pointer: front of the list; Tail pointer: back of the list
- Implementation
    -
```
template <class T>
class Queue {
    public:
        Queue();
        bool is_empty();
        void enqueue();
        T dequeue();
        T peek();
    private:
        struct Node {
            T data;
            Node* next;
        };
        Node* front, back;
};
```
    -
```
template <class T>
bool Queue<T>::is_empty() const {
    return front == nullptr;
}
```
    -
```
template <class T>
bool Queue<T>::enqueue(T d) {
  Node* newN = new Node(d);
  if (back == nullptr)
    front = back = newN;
  else {
    back->next = newN;
    back = newN;
  }
}
```

-
```
template <class T>
T Queue<T>::dequeue() {
  assert(!is_empty());
  T ret = front->data;
  Node* temp = front;
  front = front->next;
  delete temp;
  return ret;
}
```
    - Running time
        - Singly-linked list, circular, head pointer, front of list is front of queue
            - enqueue(): $O(n)$
            - "tricky" enqueue(): 
            - dequeue(): $O(n)$
            - "tricky" dequeue():

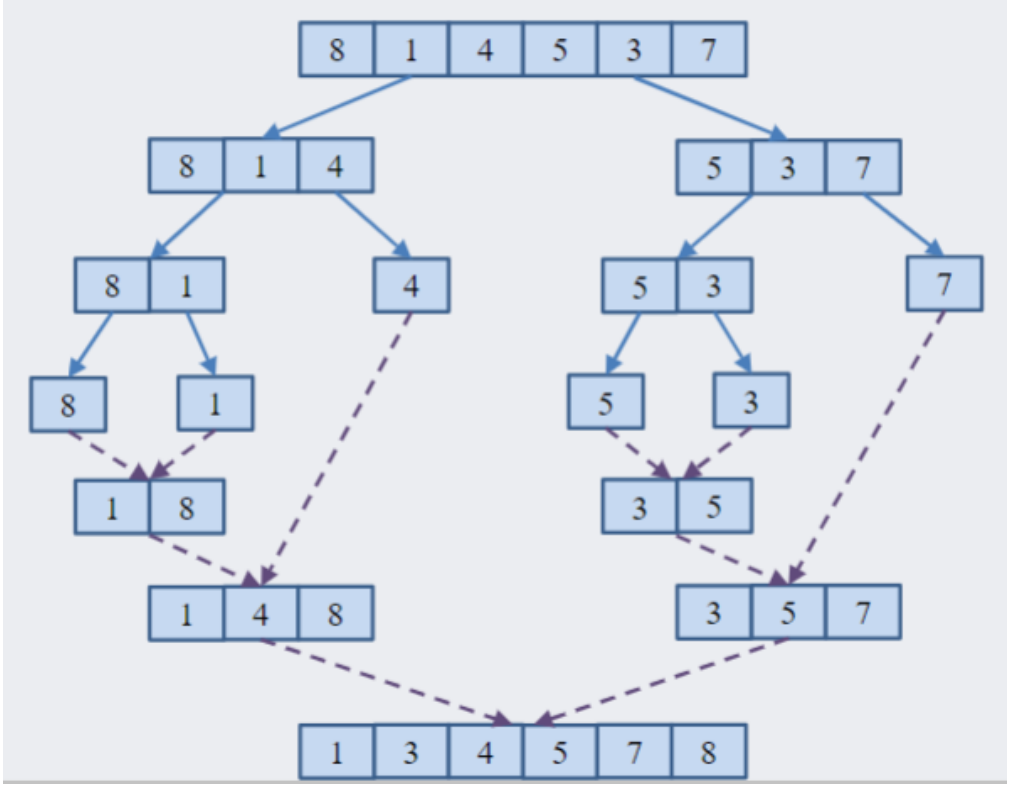


- Array implementation
    - Assume
        - Front of queue: index 0
        - Back of queue: index size() - 1
    - Then
        - enqueue(): $O(1)$
        - dequeue(): $O(n)$ if move each element to the front by 1
            - Increment the front index: $O(1)$, but waste space
    - Circular Array
        - The end of the array "wraps around" to the front of the array
        - Modulo operator: used to calculate the front and back positions in a circular array
        -
```
// Member attributes
int front;
int capacity;
T* arr;
int num;
```

        - Back of queue (new): (front + num) % capacity; num += 1
        - Front of queue (new): (front + 1) % capacity; num -= 1
    - Array queue resizing
        - Either reset front to 0 after resizing
        - Or maintain the front index, but copy in order of the queue into the new array
        - NOT: just copy index-to-index

Jan. 31st - 4.3: Merge Sort

- Subarray merging
    - Of two **sorted** subarrays
        - [1, 4, 8], [3, 5, 7]
        - Compare and increment smaller number's index by 1.
        - Should yield [1, 3, 4, 5, 7, 8]
    - Getting sorted subarrays
        - Recursively divide arrays in half until each subarray contains a single element
            - an element itself is sorted
            - merging two single-element arrays is simply a single comparison
        - Merge step copies the subarray halves into a temporary array
            - The merged elements are copied from the temporary array back to the original array
            - If we use the original array space directly, we would overwrite some values in an unexpected way.
- Merge sort example

- 

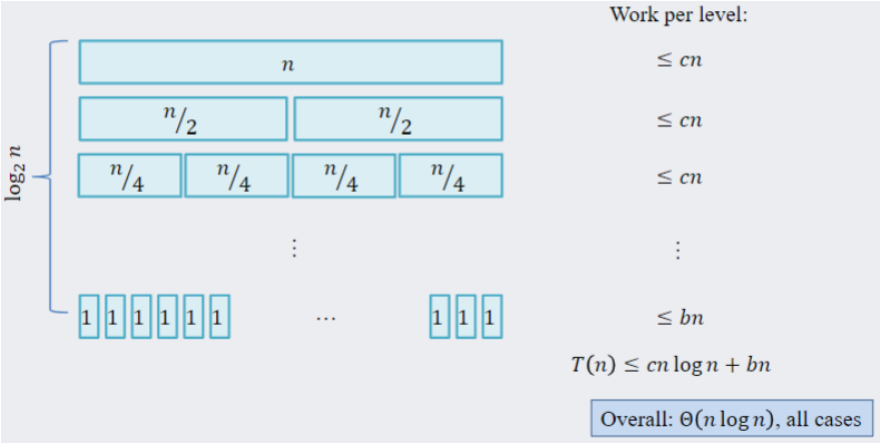
- ```
  void MergeSort(vector<T> & arr) {
      MSort(arr, 0, arr.size() - 1);
  }

  void MSort(vector<T> & arr, int low, int high) {
      if (low < high) {
          int mid = (low + high) / 2;

          MSort(arr, low, mid);       // sort the left
  half
          MSort(arr, mid + 1, high);  // sort the
  right half

          // Actual sorting work is done by merge
          Merge(arr, low, mid, high); // merge the
  sorted halves (of interest)
      }
  }
  ```

- Algorithm Analysis 1
  - Number of comparisons in the merge step
    - Worst case: $n-1$ comparisons (need to check every subarray index)
      - [1, 2, 3, 4, 99], [10, 11, 12, 13, 14] $\rightarrow$ 9 comparisons
    - Best case: $\frac{n}{2}$ comparisons (reach the end of one subarray, copy the rest of the second array)
  ```

        - [90, 91, 92, 93, 94], [1, 2, 3, 4, 5] → 5 comparisons
    - Still copying n subarray items in any case
    - Recursion Tree
        - Number of divisions: $\log_2 n$ divisions to reach 1−element subarrays
        - 

- Algorithm Analysis 2 (By recurrence)
    - Split in half → sort first half → sort second half → merge together
    - $T(1) \leq b$
    - $T(n) \leq 2 \times T(\frac{n}{2}) + c \times n$

      $\leq 2 \times (2 \times T(\frac{n}{4}) + c(\frac{n}{2})) + cn = 4 \times T(\frac{n}{4}) + cn + cn$

      $\leq 4 \times (2 \times T(\frac{n}{8}) + c(\frac{n}{8})) + cn + cn = 8 \times T(\frac{n}{8}) + cn + cn + cn$

      $\leq 2^k \times T(\frac{n}{2^k}) + 2^k \times cn$, extrapolating that $1 \leq k \leq n$

      choose k s.t. $T(\frac{n}{2^k}) = T(1)$, then $k = \log_2 n$

      thus,

      $\leq n \times T(1) + c \times n \log_2 n \leq c \times n \log_2 n + bn$

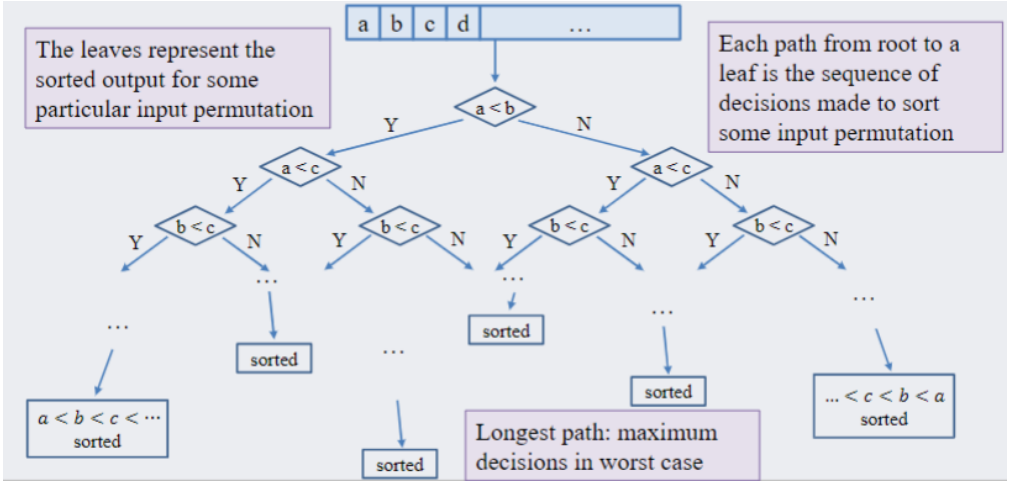
      $\in \Theta(n \log_2 n)$
    - Proof by strong induction
- Correctness
    - Claim: MSort(arr, low, high) sorts arr[low...high]
    - Base case: $n == 0, n == 1$; array is empty or size 1, already sorted
        - low == high + 1 or low == high; condition fails and function returns without modifying the array
    - Inductive step
        - Consider $1 \leq k \leq arr.size()$: assume that MSort(arr, low, high) sorts arr[low ... high] for all 0 ≤ high - low + 1 < n
        - Show that MSort(arr, low, high) sorts arr[low ... high] for all high - low + 1 = n
        - Needs to separately prove correctness of Merge

- Termination: finite number of recursive calls (halving is shrinking strictly in size), eventually reach base

Algorithm	Best	Average	Worst
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

-
  - Merge sort requires $O(n)$ space complexity

Feb. 5th - 5.1: Sort Complexity

- Comparison-based sorting
  - We have an unordered array of length n, which we ask our algorithm to sort
  - The algorithm operates input permutation to ordered permutation
  - Different input permutation, the sequence of operations will be different
  - A "correct" algorithm must be able to transform every/any input permutation to the ordered permutation
  - Sequence of operations depends on the successive comparisons between 2 elements (a and b)
    - $a < b$ or $a \geq b$
  - Thus, the possible sequences of operations can be described as a binary decision tree
- Lower bounds on sorting
  - Given an input permutation: $[a, b, c, d, \ldots]$
  - 

  - How many leaves are there in this tree?
    - How many starting permutations are there?

      $n!$

        - Thus, we would have $n!$ **different** paths to a leaf
    - What is the minimum height of a tree with $n!$ leaves?

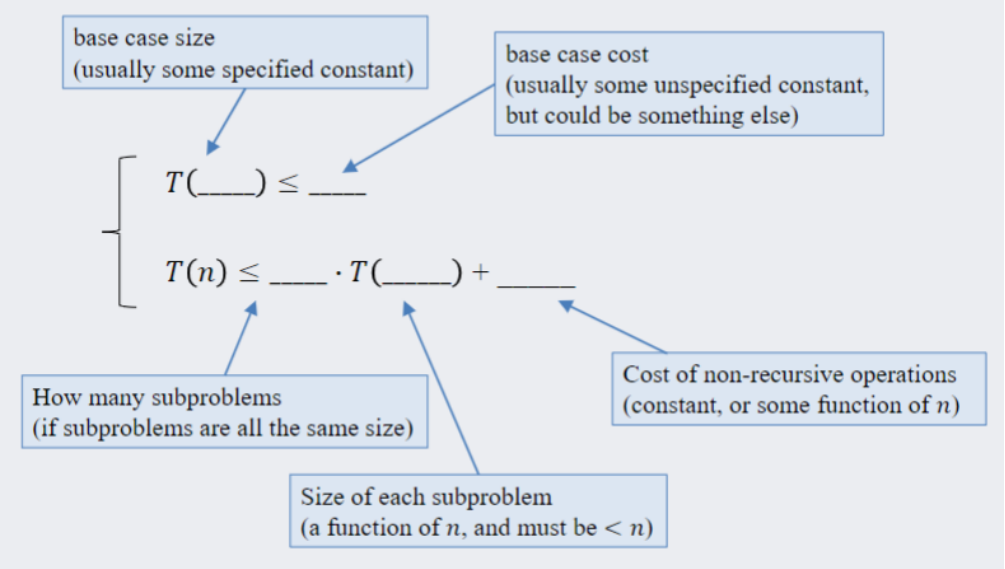
      Start with lvl. 0

      A perfect tree with h levels has $2^{h+1} - 1$ nodes; if the bottom level has $n!$ nodes, then the bottom level: $h = \texttt{ceil}(\log_2(n!))$

      $\geq \sum_{i=1}^{n} \log_2(i)$

      $\geq \sum_{i=1}^{\frac{n}{2}} \log_2(\frac{n}{2})$

      $\geq \frac{n}{2} \cdot \log_2(\frac{n}{2})$

      $\in \Omega(n \log_2(n))$
    - This means, we need to do at least $\Omega(n \log n)$ operations to complete a comparison-based sort
- Recurrences
    -
      

    - Binary search
        - Inspect midpoint, recursively search left or right half of array
        - Base case at a single element

          $T(1) \leq b$
        - $T(n) \leq T(\frac{n}{2}) + c$

          $\leq T(\frac{n}{4}) + c + c$

          $\leq T(\frac{n}{2^k}) + kc$

          $\leq T(1) + \log_2(n) \cdot c$

          $\in O(\log(n))$
    - Exact recurrences
        - $T(1) = 2$
        - $T(n) = 2 \times T(n-1) + 4$

          $= 2 \times (2 \times T(n-2) + 4) + 4 = 4 \times T(n-2) + 2 \times 4 + 4$

$$= 2 \times (4 \times T(n-3) + 2 \times 4 + 4) + 4 = 8 \times T(n-3) + 2^2 \times 4 + 2^1 \times 4 +$$

$$= 2^k \cdot T(n-k) + (2^k - 1) \cdot 4$$

$$= 2^{n-1} \times T(1) + (2^{n-1} - 1) \cdot 4$$

$$= 2^n + 2^{n+1} - 4$$

$$\in O(2^n)$$

Feb. 6th - 5.2: Trees

- Review: Linked Lists
    - Linked lists are constructed out of **nodes**, consisting of a data element and a pointer to another node
    - Lists are constructed as chains of such nodes
- Trees
    - Constructed from nodes (nodes may now have pointers to one or more other nodes)
    - A set of nodes with a single starting point (called the **root** of the tree)
    - Each node is connected by an **edge** to another node
    - A tree is a **connected** graph
        - There is a **path** from the **root** to every node in the tree
        - A tree has one less edge than the number of nodes
- Tree relationships
    - Node v is said to be a **child** of u, and u the **parent** of v if:
        - there is an edge between the nodes u and v
        - u is above v in the tree
    - This relationship can be generalized.
        - E and F are **descendants** (does not have be **parent-child** relationship) of A
        - D and A are **ancestors** of G
        - B, C, and D are **siblings**
    - Every node must have only 1 parent, except the root node.
- More tree terminology
    - A **leaf** is a node with no children
    - A **path** is a sequence of nodes $v_1, \ldots, v_n$, where v_i is the parent of v_{i+1}
    - A **subtree** is any node in the tree along with all of its descendants

    - A **binary tree** is a tree with at most two children per node
        - The children are referred to as **left** and **right**
        - We can also refer to left and right subtrees
- Measuring trees
    - The **height** of a node v is the length of the longest path from v to a leaf (the height of the tree is the height of the root)
    - The **depth** of a node v is the length of the path from v to the root (also referred to as the **level** of a node)
    - If the tree is empty?
- Perfect binary trees
    - A binary tree is **perfect** if:
        - No node has only one child
        - And all leaves have the same depth
    - A perfect binary tree of height h has $2^{h+1} - 1$ nodes, of which 2^h are leaves
    - Perfect trees are also **complete** and **full**
- Complete binary trees
    - The leaves are on at most two different levels
    - The second to bottom level is completely filled in and
    - The leaves on the bottom level are as far to the left as possible
- Full binary trees
    - Every node has exactly 0, or 2 children
- Implementation
    -
```
template <typename T>
class Tree {
    public:
        //  ..., insert, remove, travese
    private:
        struct Node {
            T data;
            Node * left;
            Node * right;
        };
        Node * root;
        // ... and other private functions
};
```

- Review
    - Precise upper bound (maximum) on the number of nodes in a binary tree of height h

      $2^{h+1} - 1$
    - Precise lower bound on the height of a binary tree containing n nodes

      $\lfloor \log_2 n \rfloor$
    -

Feb. 7th - 5.3: Tree Traversal

- Binary Tree
    -
```
template <typename T>
class BinaryTree {
    public:
        //  ..., insert, remove, traverse
    private:
        struct Node {
            T data;
            Node * left;
            Node * right;
        };
        Node * root;
        //  ... and other private functions
}
```

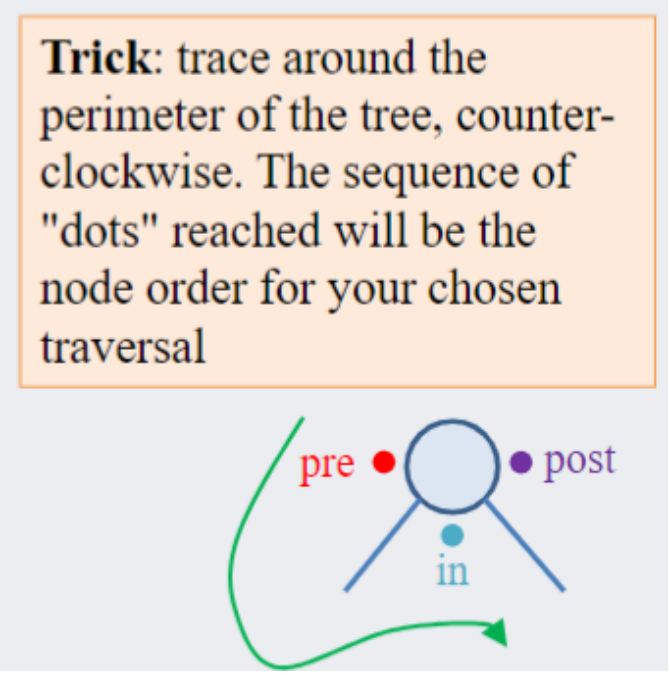
    - Recursive definition
    - Full binary tree: each node has exactly 0 or 2 children
        - Must contain odd numbers of nodes, therefore, impossible draw out 8-node full binary tree
    - Complete binary tree: almost perfect, bottom level is packed to the left
- Traversal
    - A traversal algorithm for a binary tree visits each node in the tree
        - Iterative traversal

```
Node * p = root;
while (p != NULL) {
    // do smth useful
    p = p -> left;
    // but how to backtrack?
}
```

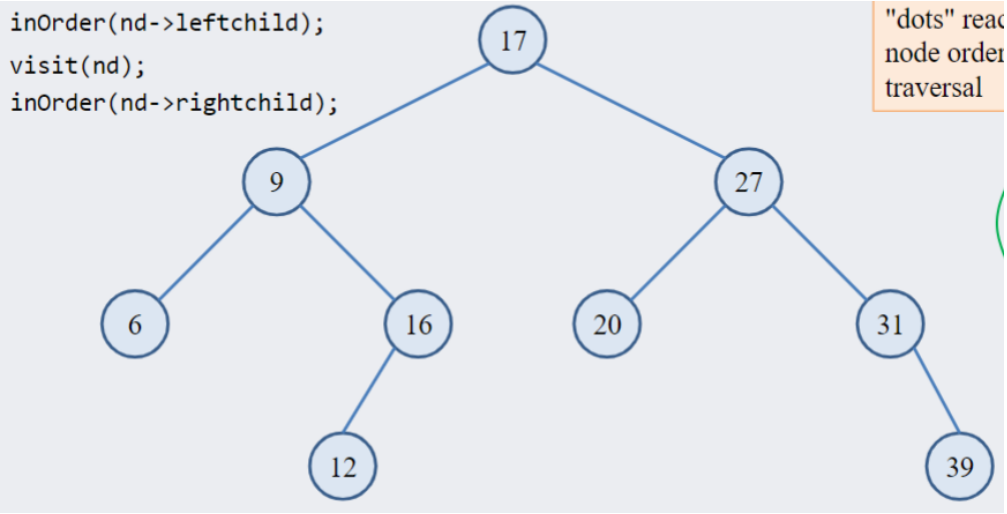
- Traversal algorithms are naturally recursive
    - inOrder
    - preOrder
    - postOrder
- inOrder
    - ```
      void inOrder(Node * nd) {
          if (nd != NULL) {
              inOrder(nd -> left);
              visit(nd);
              inOrder(nd -> right);
          }
      }
      ```

- preOrder
    - ```
      void preOrder(Node * nd) {
          if (nd != NULL) {
              visit(nd);
              preOrder(nd -> left);
              preOrder(nd -> right);
          }
      }
      ```

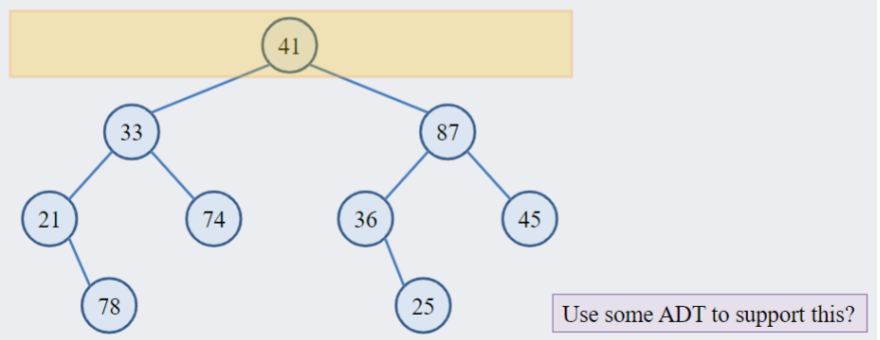
    - Root is always first to be output
    - Both left subtree contents and right subtree contents are grouped together
- postOrder
    - ```
      void postOrder(Node * nd) {
          if (nd != NULL) {
              postOrder(nd -> left);
              postOrder(nd -> right);
              visit(nd);
          }
      }
      ```
      ```



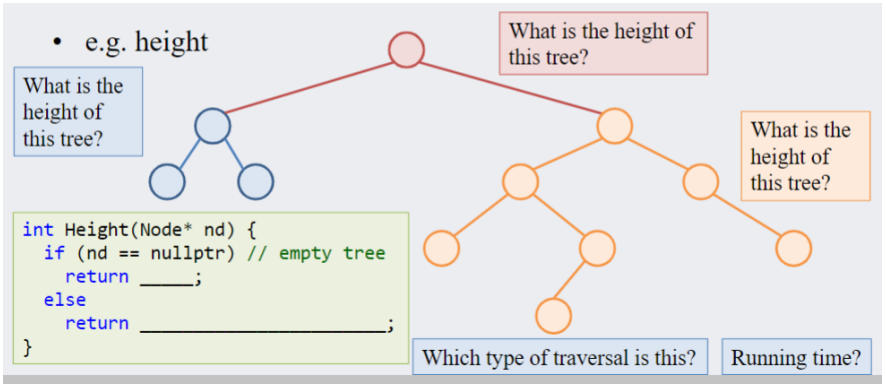
- - Left-bias version: trace counter-clockwise
  - Right-bias version: trace clockwise, and pre/post swapped



- - In-order traversal of this tree is increasing: 6, 9, 12, 16, 17, 20, 27, 31, 39

- Non-recursive traversal
  - Level-order traversal
    - 

- "Visiting"
  - Operations to be done at the current node (counting, arithmetic, creating a node, deleting a node)

- e.g. Height
  - 

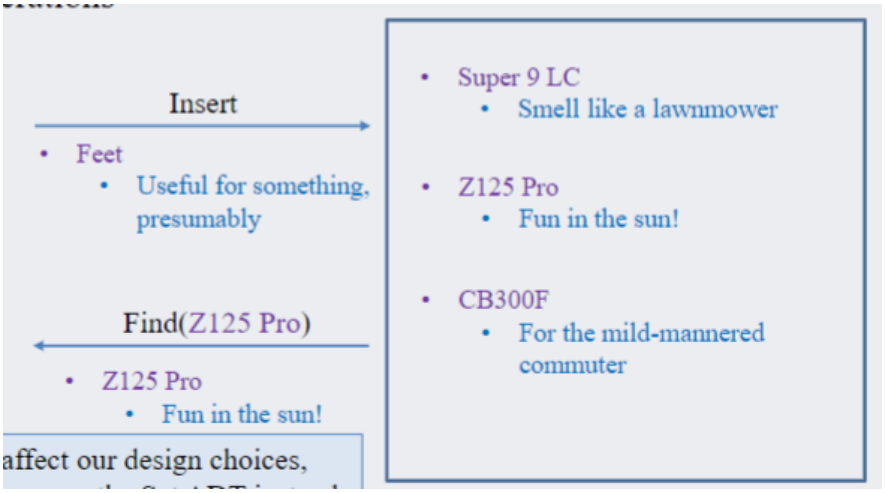
  - ```
    int Height(Node * nd) {
        if (nd == NULL) {
            return -1;      // empty tree
        } else {
            return 1 + max(Height(nd -> left),
    Height(nd -> right));
        }
    }

    int ExplicitHeight(Node * nd) {
        if (nd == NULL) {
            return -1;
        } else {
            // post-order traversal
            int height_left = Height(nd -> left);
            int height_right = Height(nd ->
    right);
            int curr_height = 1 +
    max(height_left, height_right);
            return curr_height;
        }
    }
    ```

Feb. 12th - 6.1: Level Order traversal + Dict/Map ADT

- Recursive traversal
  - ```
    void PrePrint(Node * nd) {
        if (nd != NULL) {
            cout << nd -> data << endl;
            PrePrint(nd -> left);
            PrePrint(nd - > right);
        }
    }
    ```

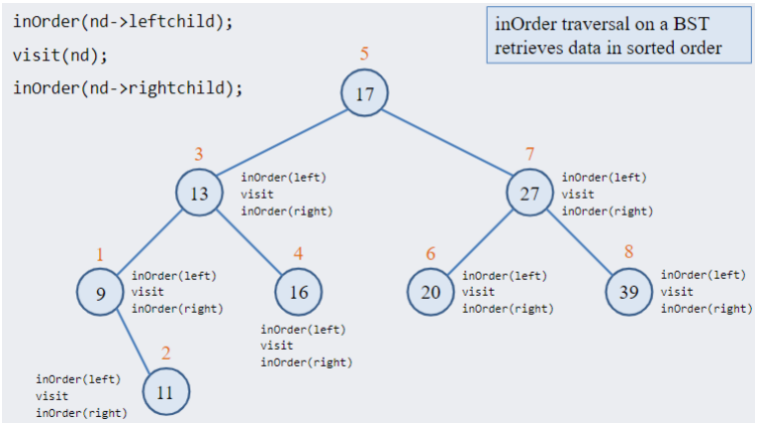
- Iterative traversal
    - We can store `Node *` in `st` for more complex tasks
    - But a call stack is much more convenient
        - ```
          void PrePrint(Node * nd) {
              Stack<Node *> st;
              if (nd != NULL) {
                  st.Push(nd);
              }
              while (!st.IsEmpty()) {
                  Node * curr = st.Pop();
                  cout << curr -> data << " ";
                  if (curr -> right != NULL) {
                      st.Push(curr -> right);
                  }
                  if (curr -> left != NULL) {
                      st.Push(curr -> left);
                  }
              }
          }
          ```
        - What controls the worst-case space complexity? **The height of the tree**, with siblings.
    - What if we change it to a call queue?
        - ```
          void Print(Node * nd) {
              Queue<Node *> q;
              if (nd != NULL) {
                  q.Enqueue(nd);
              }
              while (!q.IsEmpty()) {
                  Node * curr = q.Dequeue();
                  cout << curr -> data << " ";
                  if (curr -> left != NULL) {
                      q.Enqueue(curr -> left);
                  }
                  if (curr -> right != NULL) {
                      q.Enqueue(curr -> right);
                  }
              }
          }
          ```

        - Level-order traversal
        - What controls the worst-case space complexity here?
            - A nearly-perfect tree is wider
- Motivation for an efficient lookup (Dictionary / Map ADT)
    - Stores **values** associated with user-specified keys
        - Values may be any (homogenous) type
        - Keys may be any (homogenous) comparable type
    - Dictionary operations
        - Create, Destroy, Insert, Find, Remove
        - 

    - Set ADT (typically not storing duplicates)
        - Stores keys (quickly tests for membership)
        - Set Operations: create, destroy, insert, **find**, remove
    - Data structures for Dictionary ADT
        -

	SEARCH	INSERT	REMOVE
Linked list	$O(n)$	$O(1)$ at front	$O(n) + O(1)$ (search + remove)
Unsorted array	$O(n)$	$O(1)$ at back	$O(n) + O(1)$ (search + remove)
Sorted array	$O(\log n)$ (binary search)	$O(\log n) + O(n)$ (search + shifting to the right)	$O(\log n) + O(n)$ (search + shifting to the left)
Ordered linked list	$O(n)$	$O(n) + O(1)$ (search + insertion)	$O(n) + O(1)$ (search + removal)

    - Binary Search Tree property
        - For all nodes

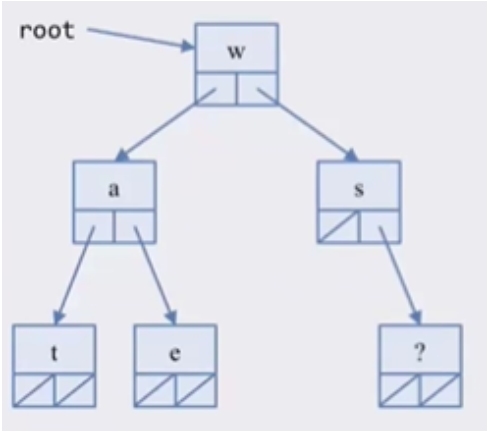
        - All nodes in a left subtree have labels **less** than the label of the subtree's root
        - All nodes in a right subtree have labels **greater** than or equal to the label of the subtree's root
    - It is **fully ordered**
    - BST inOrder traversal
        -
```
inOrder(nd -> leftChild);
cout << nd -> data << " ";
inOrder(nd -> rightChild);
```
        -


    - BST search
        - To find a value in a BST search from the root node, if the target is less than the value in the node search its left subtree; if the target is larger than the value in the node search its right subtree
        - Operations (One for each node on the path, the worst case: height of the tree + 1)

Feb. 13th - 6.2: How many nulls?

- Where's the waste?
    - How much space does a null-terminated, singly-linked list "waste" when storing n values?
        - What counts as waste? Focus on "anything that is not for content" (anything other than the actual data)
        - Each node "wastes" a pointer + head pointer + null-termination
        - In general, extra n pointers needed for n nodes ($\Theta(n)$ space wasted)
    - How much space is "wasted" just on null values?

        - For singly linked lists, every list will waste 1 null ($\Theta(1)$ space wasted), 1 null wasted even empty list
        - For doubly linked lists, every list will waste 2 nulls, 2 nulls wasted except empty list
        - In general, $\Theta(1)$ space wasted on null
    - How much space does an array "waste" when storing n values?
        - Indices are not stored by the array, but array needs to have a size when created.
        - The spaces of $capacity - size$ will be wasted
        - Best case scenario: we don't waste space $\Theta(1)$

          Worst case scenario: we double the array, we waste n spaces, $\Theta(n)$
    - How much space does a binary tree "waste" when storing n values?
        - 

        - For non-data waste: n nodes lead to $2n$ pointers + `root` pointer
        - In this specific example: there are 7 nulls
        - For an empty tree: there is 1 null for 0 nodes; for an `root` tree: there are 2 nulls for 1 node
        - Note that the leaf nodes contribute more nulls
        - It seems that with n nodes, there will be $n + 1$ nulls wasted.
- BST null waste (strong induction)
    - Hypothesis: for n nodes, the tree has $n + 1$ nulls
    - Definition: A finite binary tree is either an empty tree $\varnothing$ or a non-empty tree (v, T_L, T_R), where v is a value, and T_L and T_R are finite binary trees
    - Base case: empty tree has 0 nodes, 1 null wasted

      Inductive hypothesis: we have an arbitrary finite binary tree $T = (v, T_L, T_R)$, then T_L has $n_L + 1$ nulls (n_L nodes) and T_R has $n_R + 1$ nulls (n_R nodes)

      $(n_L < n, n_R < n)$, for any tree with nodes less than n, we form this hypothesis

Inductive step: we know $n = n_L + n_R + 1$, then for the number of nulls in T, there are $(n_L + 1) + (n_R + 1) = n + 1$ nulls

- BST null waste (intuition)
    - In a binary tree with n values, there are exactly n nodes
    - Exactly 1 of the those nodes has no parent, the `root` node
    - The other nodes each have exactly 1 parent
    - Every non-null pointer points from a **parent** to a **child**
    - There are exactly, $n - 1$ non-null pointers
    - The total number of pointers in the tree nodes is exactly $2n$
    - So, the number of nulls in a binary tree of n values is $2n - (n - 1) = n + 1$
- What about other structures?
    - Consider a 4-ary tree, then for each node, there are 4 pointers, in total there are $4n$ pointers. The non-null pointers are still $n - 1$, then the number of nulls is $3n + 1$.
    - In general, a k-ary tree, the number of nulls is $(k - 1)n + 1$
- Other questions to consider
    - Give a recurrence relation for the number of different shapes of binary trees with n nodes
        - e.g. For an empty tree, there is only 1 shape; for a `root` tree, there is only 1 shape; for 2 nodes, there are 2 shapes
        - Since a tree structure is recursive, we can still break it down to the left subtree and the right subtree. Recall we also know that $n = n_L + n_R + 1$

$$\begin{cases} S(0) = 1 \\ S(n) = \sum_{n_L=0}^{n-1} S(n_L) \cdot S(n - n_L - 1) \end{cases}$$

        - For this situation, we have

$$S(1) = \sum_{n_L=0}^{1-1} S(n_L) \cdot S(n - n_L - 1) = S(0) \times S(0) = 1,$$

$$S(2) = \sum_{n_L=0}^{2-1} S(n_L) \cdot S(n - n_L - 1) = S(1) \times S(0) + S(1) \times S(0) = 2$$

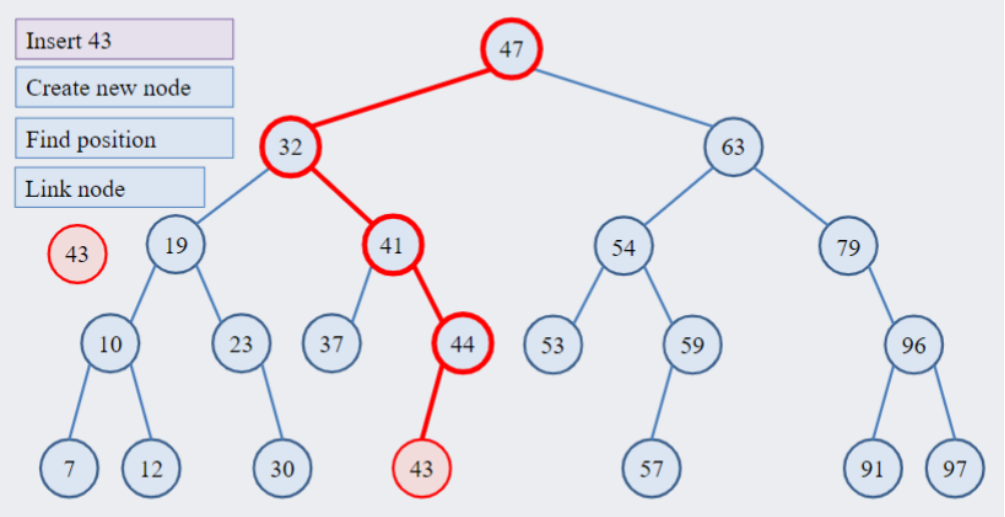
Feb. 14th - 6.3: BST Insert/Find

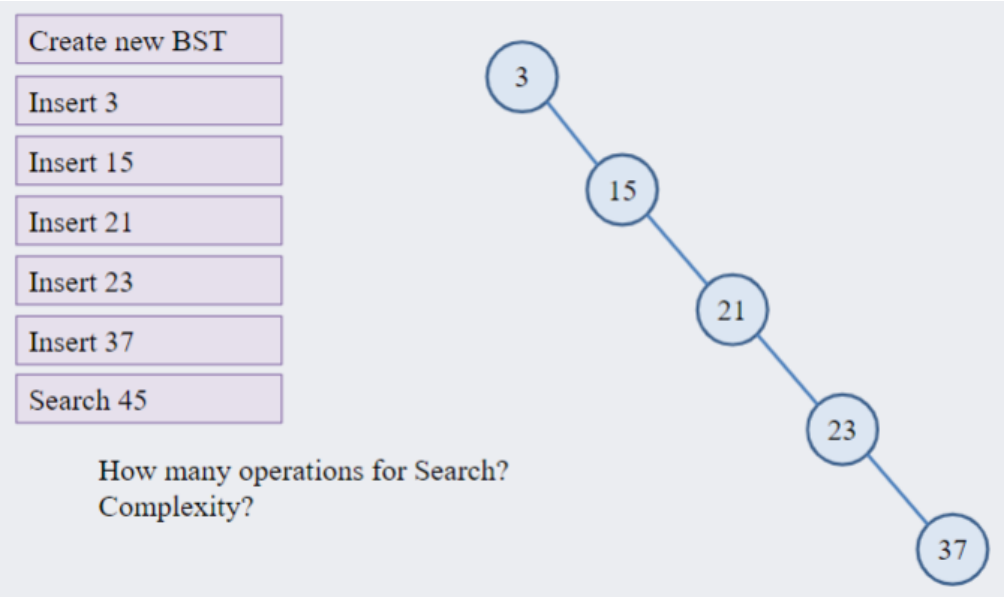
- Recall: Set & Dictionary ADT
    - Dictionary: Stores key/value pairs
    - Set: Stores keys

        - Organization: nebulous
        - Important operations: arbitrary insertion, removal, lookup
        - Concrete data structures for implementation: arrays, linked lists, ...
- Binary search tree property
        - Left: less than parent; Right: more than parent (fully ordered)
        - Tree sort: use BST to perform
- BST search
        - Worst case: length of longest path = height of the tree + 1
- BST insertion
        - BST property must hold after insertion, therefore, new node must be inserted in the correct position
                - This position is performed by a search
                - If the search ends at the (null) left child of a node make its left child refer to the new node
                - If the search ends at the (null) right child of a node make its right child refer to the new node
        - The cost is about the same as the cost for the search algorithm, $O(height)$
        - Search and insert can both be implemented iteratively or recursively
        -
```
void Insert(T key) {
    // start iterative code
}

void Insert(T key) {
    // initial recursive call
    root = InsertRec(root, key);
}

Node * InsertRec(Node * nd, T key) {
    // recursive insertion into subtree rooted at nd
    // returns root of subtree after insertion
}
```

- 

- 

- - How many operations for search? $O(height)$
  - Complexity? $O(n)$
  - If the tree is complete/perfect (or almost perfect), then $height \in O(\log n)$

- Find `min, max`
  - Find minimum: left-most child
  - Find maximum: right-most child
  - Worst case: $O(height)$, which depends on the structure of the tree
- Insertion
  - Recursive Implementation

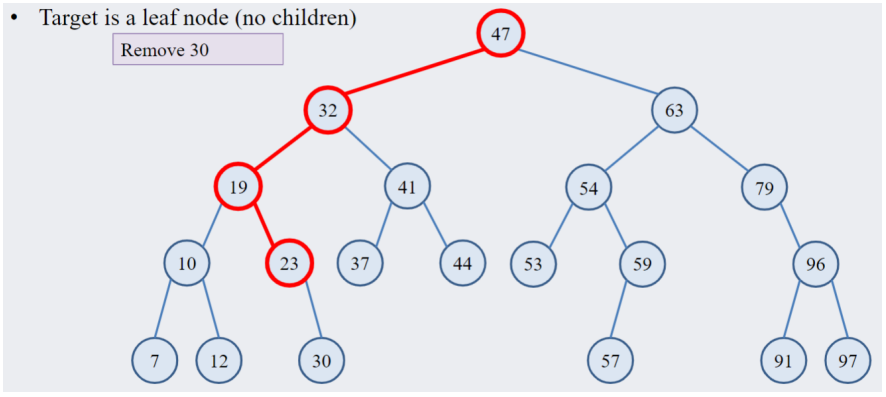
```
Node * InsertRec(Node * nd, T key) {
    if (nd == NULL) {       // tree is empty
        Node * newNode = new Node(key);     // left,
right pointers are null
        return newNode;
    } else {
        if (key < nd -> data) {
```

```
            nd -> left = InsertRec(nd -> left, key);
        } else {
            nd -> right = InsertRec(nd -> right,
key);
        }
        return nd;
    }
}

void Insert(T key) {
    root = InsertRec(root, key);
}
```

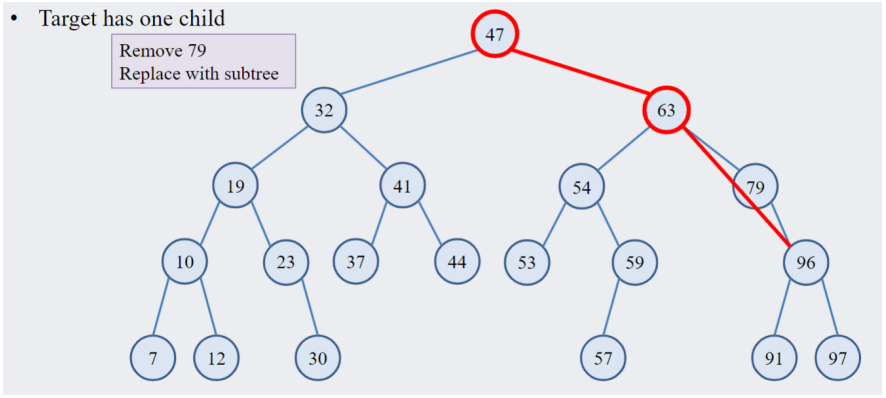
Feb. 26th - Week 8.1: BST Remove

- Review
    - Search: $O(height)$
    - Insertion: $O(height)$
- BST Removal (Preview)
    - After removal the BST property must hold
    - Removal is not as straightforward as search or insertion
        - Insertion: avoids changing the internal structure of the tree
        - Not possible with removal: the node's position is not chosen by algorithm
    - Different cases
        - The node to be removed has no children (remove it, assigning null to its parent's reference)
        - The node to be removed has one child (Replace the node with its subtree)
        - The node has 2 children (...?)
- BST Removal
    - No children node, leaf node
        - Need to search the node first

- 

  - call `delete` on `23`'s `right` pointer

```
delete p -> right;
p -> right = NULL;
```

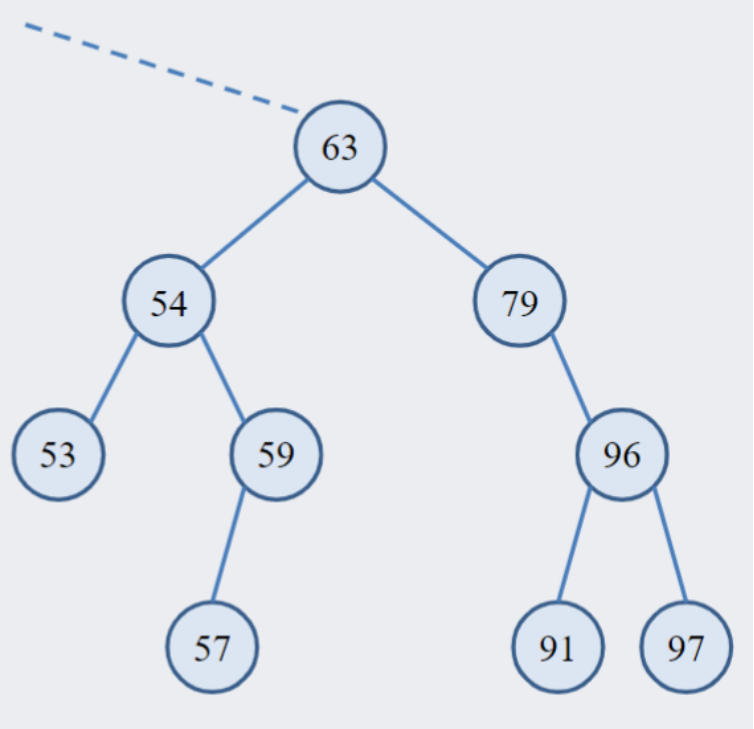
- Target has one child
  - 

  - ```
Node * target = p -> right;
p -> right = target -> right;
delete target;
target = NULL;
```

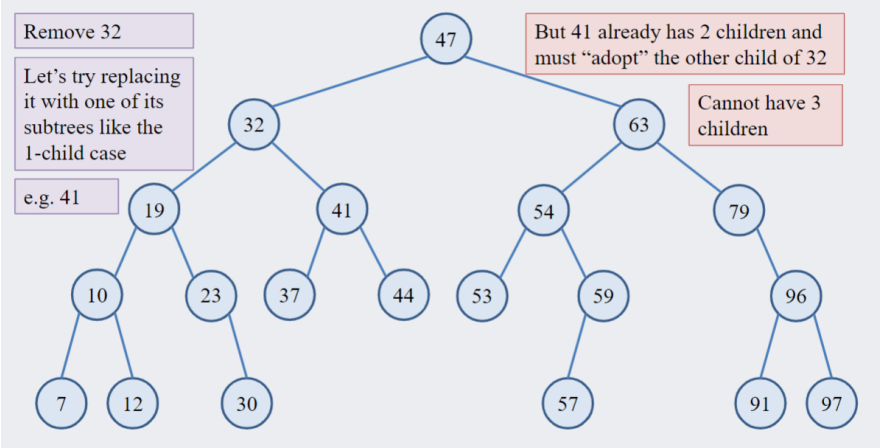
    - Regardless whether `79` has a left child or a right child, the subtrees will always have values larger than `63`
- Looking at the next node
  - One of the issues with implementing a BST is the necessity to look at both children
    - *look ahead* for insertion and removal
    - check that a node is null before accessing its member variables
  - Consider removing a node with one child in more detail

```
while (nd -> data != target) {
    if (nd == NULL) {
        return NULL;
    }
    if (target < nd -> data) {
```
```

```
            parent = nd;
            nd = nd -> left;
            isLeftChild = true;
        } else {
            parent = nd;
            nd = nd -> right;
            isLeftChild = false;
        }
    }
```

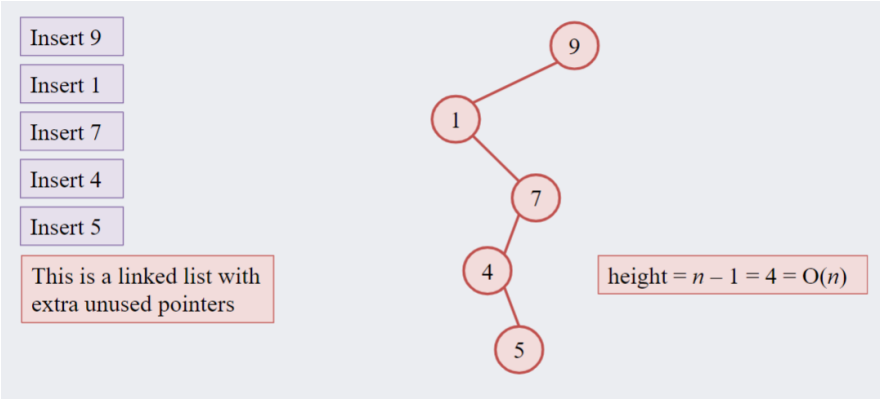
- 

    - Removing `59`
    - To make the correct link, we need to know if the node to be removed is a left or right child
- Removing a node with 2 children
    - The strategy when the removed node had one child was to replace with its child
    - Which child should we replace the node with? (Should we just pick one?)
    - What if the node we replace it with also has two children?

- 

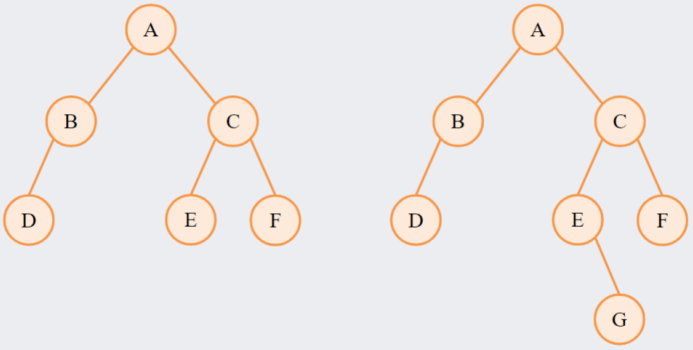
    - Find the predecessor
        - We find the node's predecessor (the **right-most** node of its **left subtree**) by in-order traversal
        - The predecessor cannot have a right child and therefore can have at most one child (the left child)

          Suppose the predecessor has a right-child, then itself will not be the **right-most** node of the node's **left subtree**, contradiction

          `p -> data` is the largest key $\leq$ `nd -> data`,
          `r -> data` $\geq$ `p -> data`
    - Predecessor's property
        - It must be the largest value less than its ancestor's value
        - It can have at most only one child
        - A good candidate to replace its ancestor
    - The successor works as well, but only pick one to be consistent.
- Efficiency
    - The efficiency of BST operations depends on the *height* of the tree
    - If the tree is complete (mostly complete), the height is $\lfloor \log(n) \rfloor$
    - Structure / Shape of a BST depends on the order of insertions
    - 

    - Back to Set/Dictionary implementation

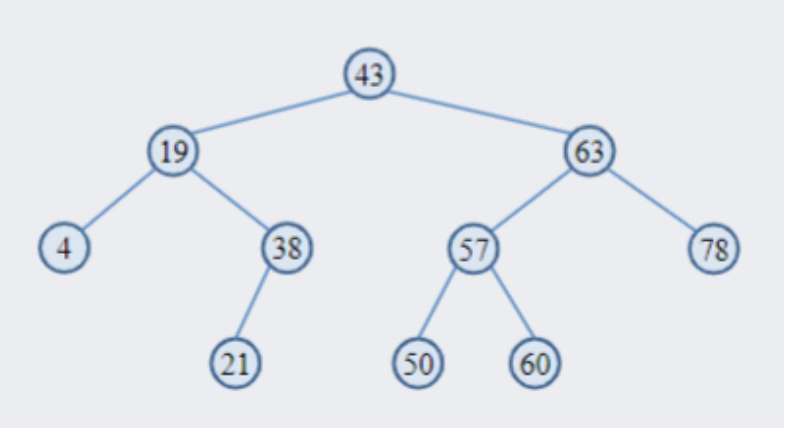
        - Worst-case complexity for the important set/dictionary operations
SEARCH	INSERT	REMOVE
$O(n)$	$O(n)$	$O(n)$
        - How can we guarantee it to do better? Balanced binary trees
- Balanced binary trees
    - A binary tree is **balanced** if
        - Leaves are all about the same distance from the root
        - The exact specification varies
    - Sometimes trees are balanced by comparing the height of nodes
        - Height of left subtree is at most one different from height of right subtree (AVL trees)
        - • By AVL property – L/R subtree heights differ by at most 1

    - Sometimes tree's height is compared to the number of nodes
        - Red-black trees

Feb. 27th - Week 8.2: AVL Analysis

- AVL tree
    - An AVL tree is a balanced BST
        - Each node's left and right subtrees differ in height by at most 1
        - Rebalancing via **rotations** occurs when an insertion or removal causes excessive height difference
    - AVL tree nodes contain extra information to support this height information

- 

- Height
    - The height of an AVL tree containing n key values is $O(\log(n))$
    - Intuition: for a fixed height h, a tree containing fewer nodes has a larger height-to-node ratio; we attempt to achieve the worst ratio by making an AVL tree of height h with the minimum number of nodes
    - Proof:

      Let N_h represent the minimum number of nodes in an AVL tree of height h

      The children of such a tree must also be minimal, then, for the root node, if one subtree has N_{h-1} nodes, then the other subtree can have N_{h-2} nodes (allow height difference of 1)

      This is minimizing the size by maximizing the height difference between the children

      Thus, $N_h = N_{h-1} + N_{h-2} + 1$

      $= (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1$

      $> 2N_{h-2}$

      $> 2(2N_{h-4})$

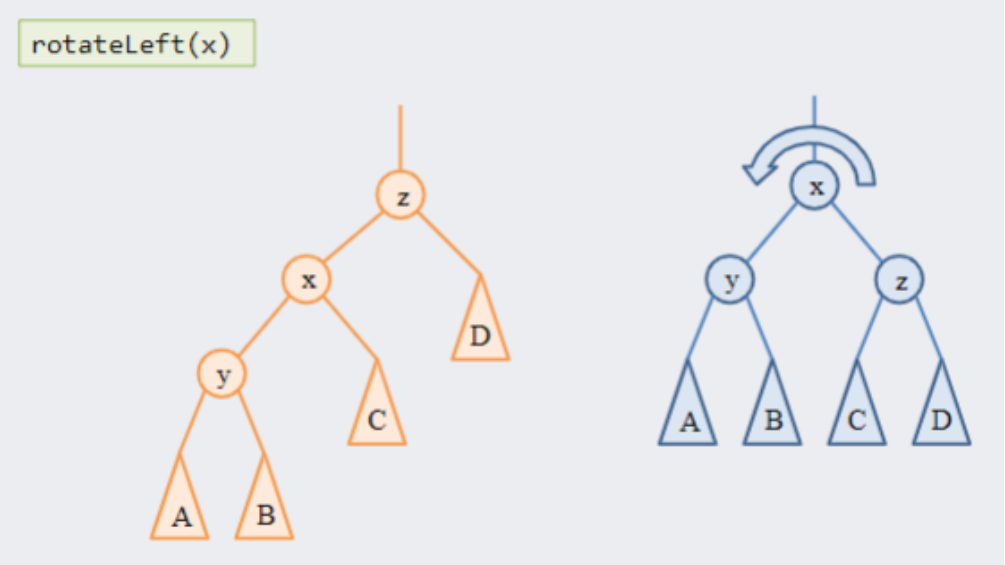
      $> 2^k N_{h-2k}$

      when $k = \frac{h}{2}$

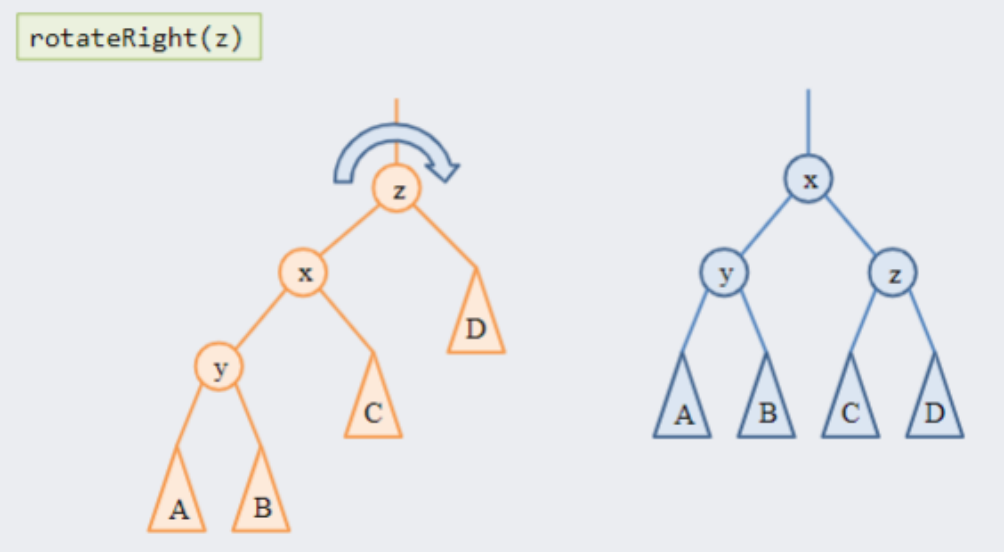
      $> 2^{\frac{h}{2}} N_0 = 2^{\frac{h}{2}}$

      With $N_h > 2^{\frac{h}{2}}$, we would then have $\frac{h}{2} < \log_2(N_h)$, equivalent to $h < 2\log_2(N_h) \in O(\log n)$
    - This guarantees that search, insertion and remove would be in $O(\log n)$ time
- Rotations
    - An item must be inserted into a BST at the correct position
    - The shape of a tree is determined by
        - the values of the items inserted into the tree

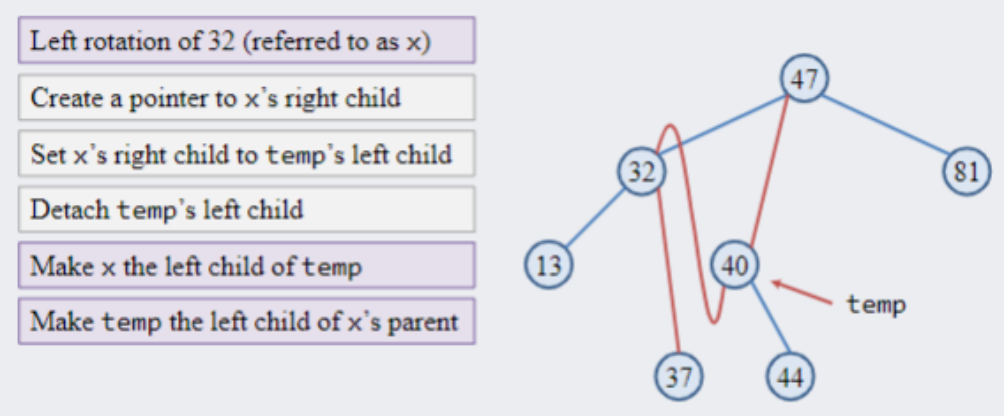
    - the order in which those values are inserted
- This suggests that there is more than one tree that can contain the same values
- A tree's shape can be altered by **rotation** while still preserving the BST property (in-order traversal is also preserved)
- `rotateLeft(x)`



- `rotateRight(z)`

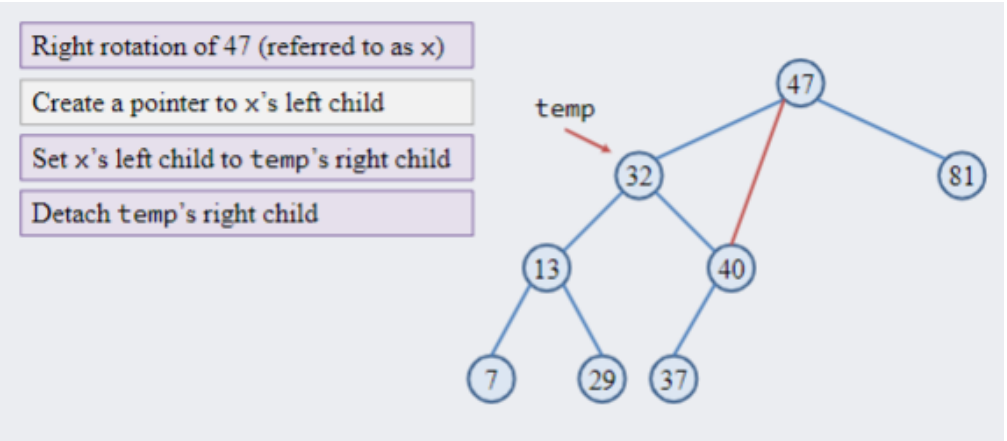


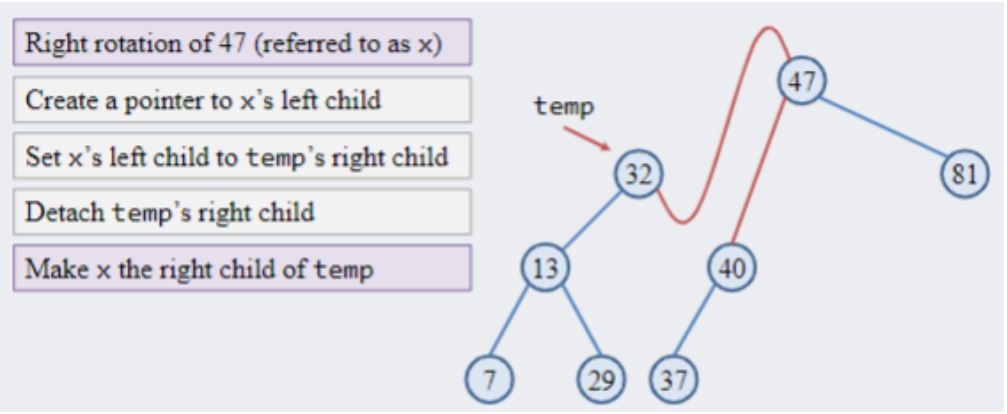
- Use rotations to transfer "weight" from a taller subtree to a shorter subtree
- Left rotation example

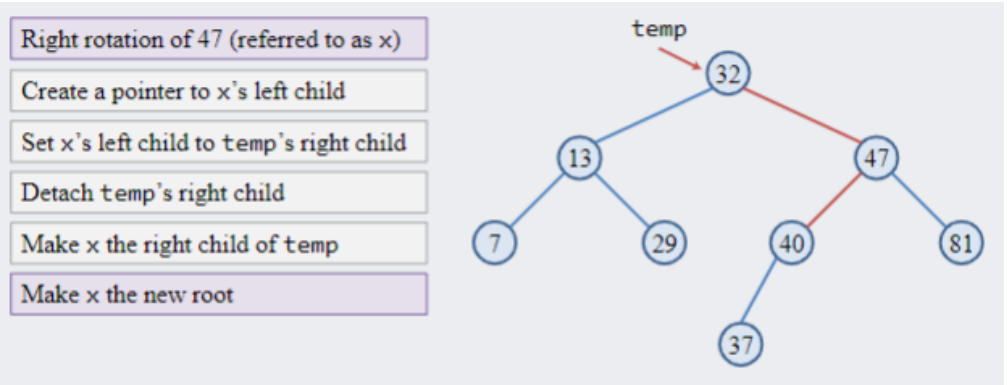


- 3 pointer reassignments, the cost would be in $O(1)$ time

- Right rotation example (around the root)







- 3 pointer reassignments

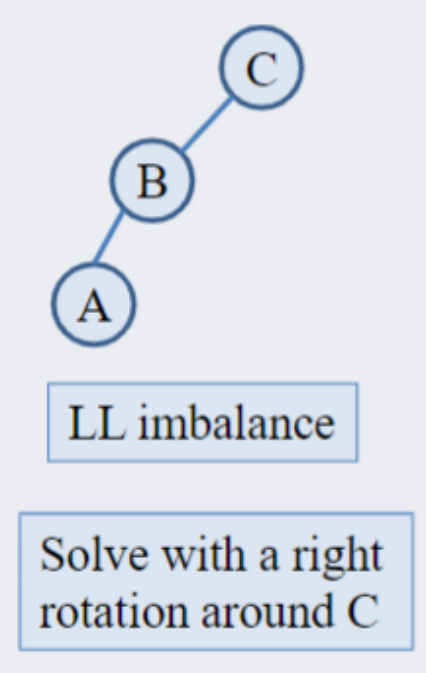
Feb. 28th - Week 8.3: AVL Insert/Remove

- AVL nodes
  - ```
    enum balance_type {LEFT_HEAVY = -1, BALANCED = 0,
    RIGHT_HEAVY = 1};

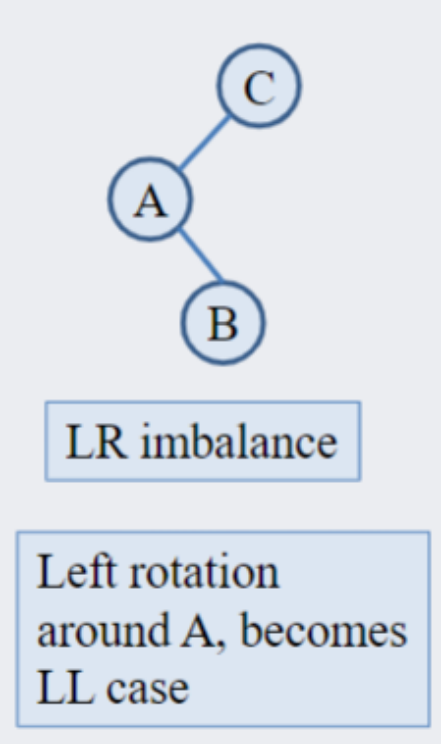
    class AVLNode {
        public:
            int data;       // or template type
            AVLNode * left;
            AVLNode * right;
            balance_type balance;

            AVLNode(int value) {...}
            AVLNode(int value, AVLNode * left1, AVLNode
    * right1) {...}
    }
    ```

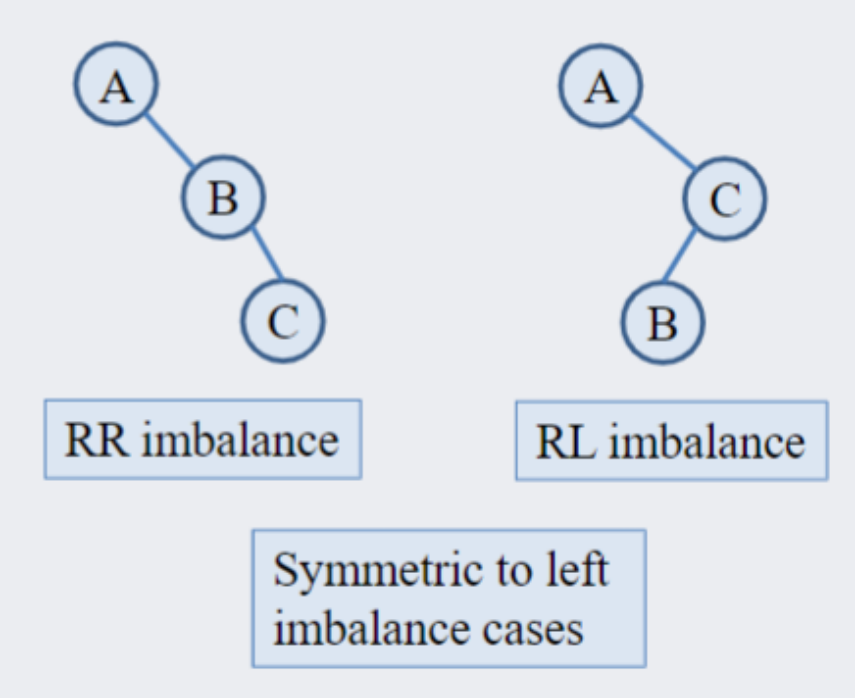
- AVL imbalance
  - 4 cases of imbalances
    - Critical imbalance: height difference > 1
    - LL imbalance:


    ```

- LR imbalance:



- R imbalances:



- AVL insertion
    - Maintaining balance
        - The best way to keep a tree balanced, is to never let it become imbalanced
        - AVL insertion starts with ordinary BST insertion followed by rotations to maintain balance
            - if the balance attribute of a subtree's root node becomes critical $(-2/+2)$ as a result of insertion, rebalance

-
```
if root is NULL
  Create new node containing item, assign root to it, and return true
else if item is equal to root->data
  item exists already, return false

else if item < root->data
  Recursively insert the item into the left subtree
  if height of left subtree has increased (increase variable is true)
    balance--;
    if balance == 0, reset increase variable to false
    if balance < -1
      reset increase variable to false
      perform rebalanceLeft

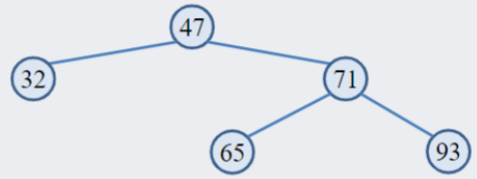
else if item > root->data
  (symmetric to left subtree case, incrementing balance)
```

Summary: BST insertion, then fix imbalances moving up towards root

rebalanceLeft and rebalanceRight are the rotations to correct the 4 imbalance cases

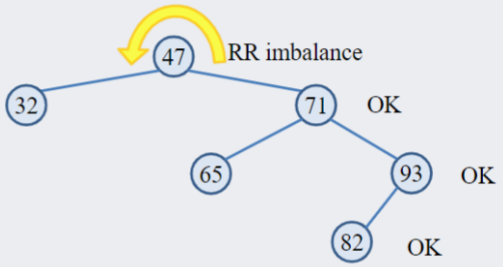
- Example

Insert(65)



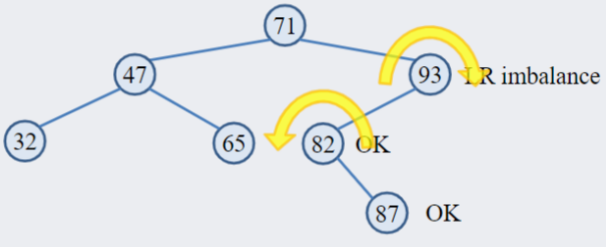
- So far the AVL tree is still balanced

Insert(65)
Insert(82)



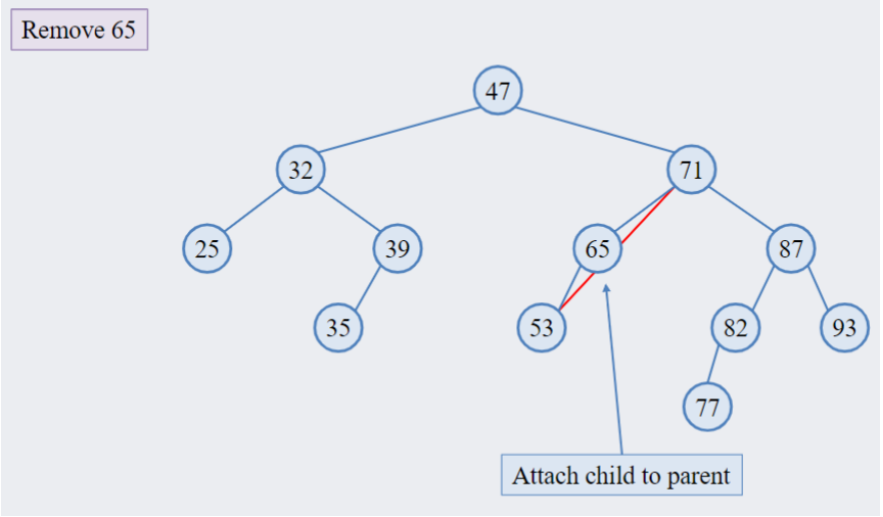
- Since 82 will make the node 47 to be RR imbalance, we need to `rotateLeft(42)`

Insert(65)
Insert(82)
Insert(87)

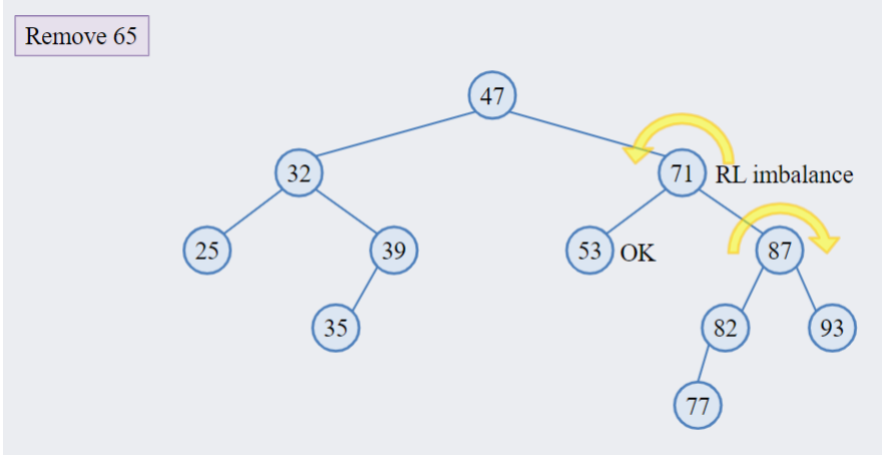


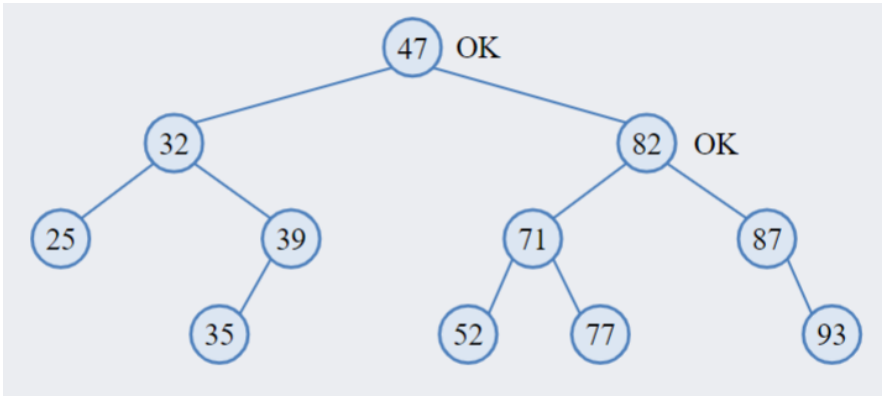
- $93, 82, 87$ is a LR imbalance, then `rotateLeft(82)`
- it becomes a LL imbalance, then `rotateRight(93)`

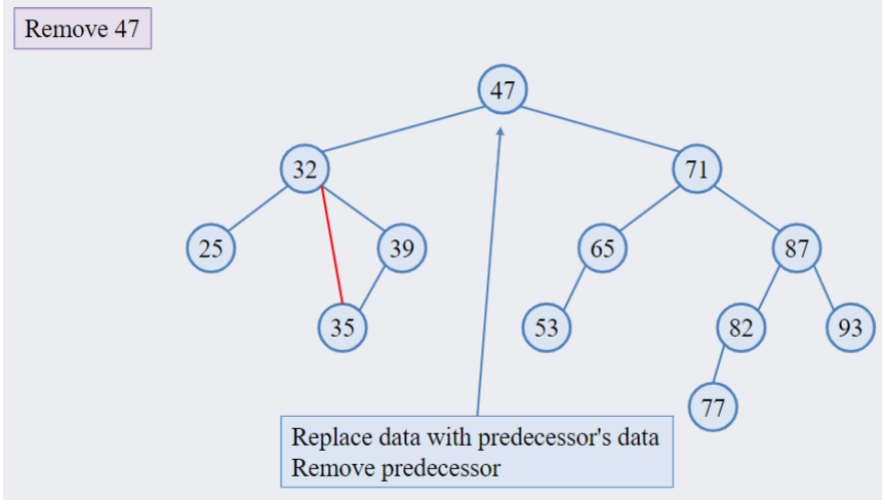
- Remember that rotations also changes `balance_type` as well!
- Complexity
  - BST insertion + fix

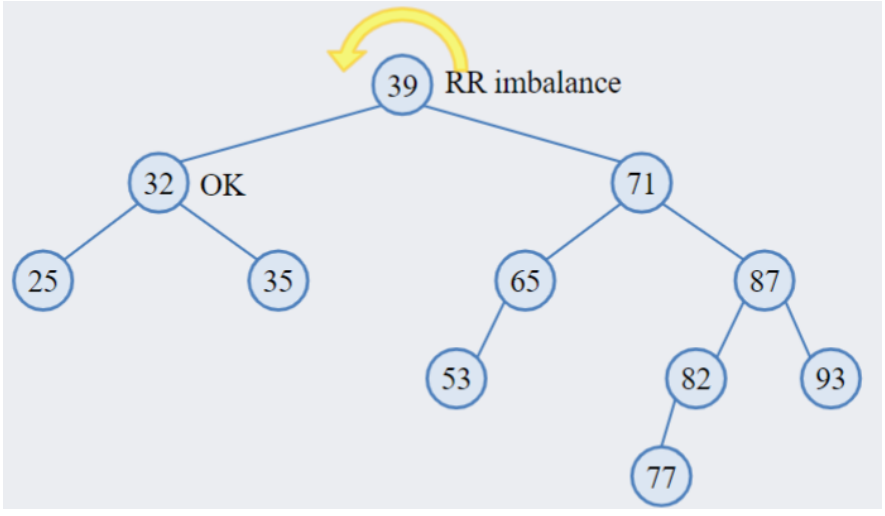
        - since BST insertion is guaranteed balanced, so this part is $O(\log(n))$
        - the fix involves: 1. move upwards looking for critically imbalanced nodes 2. at most 2 rotations
        - Each rotation is in $O(1)$, 3 pointer assignments
        - In general, the complexity will be $O(\log(n))$
        - actual, at most 1 critical imbalance
- AVL removal
    - AVL properties must be satisfied before and after removal
    - Begin with basic BST removal (using predecessor **data replacement**)
        - local root balance is adjusted for removing from left or right subtree
        - maintain a `decrease` variable (analogous to `increase`) in insertion to indicate whether the height of the tree decreased
        - use the `decrease` variable to increment or decrement local root balance
        - rebalance if critical
    - Example
        - 

        - This would lead to a RL imbalance
        - Perform `rotateRight(87)` to be RR imbalance
        - Perform `rotateLeft(71)` to be balanced

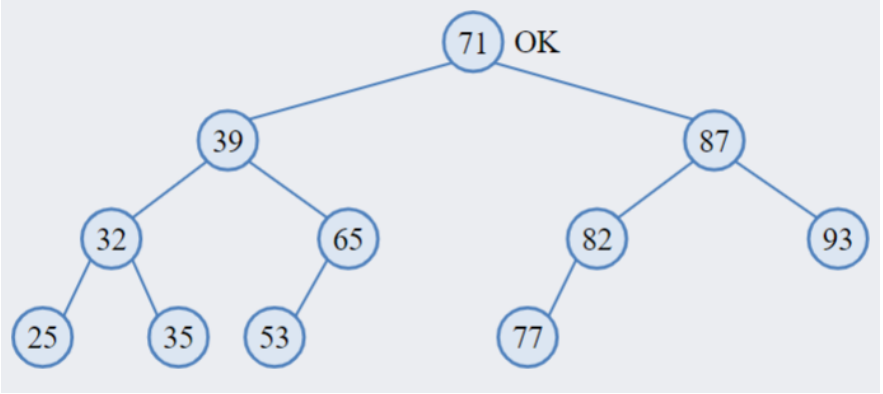
- 

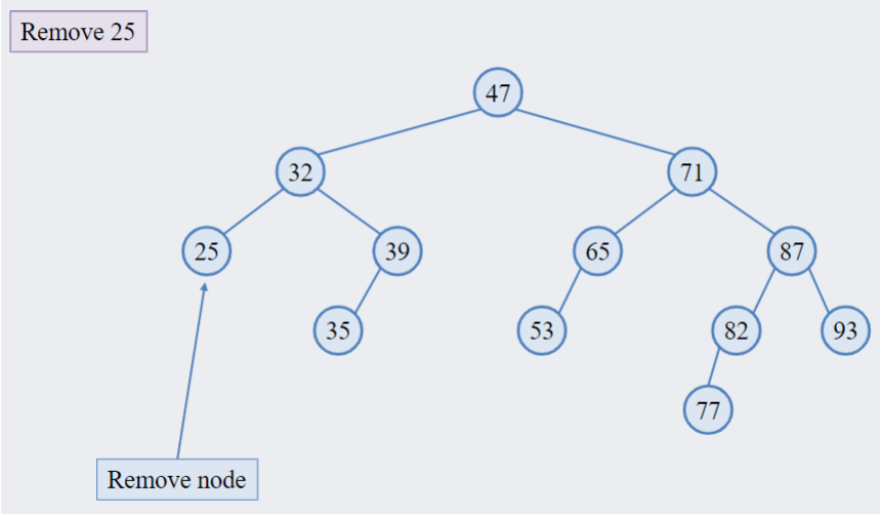
- 

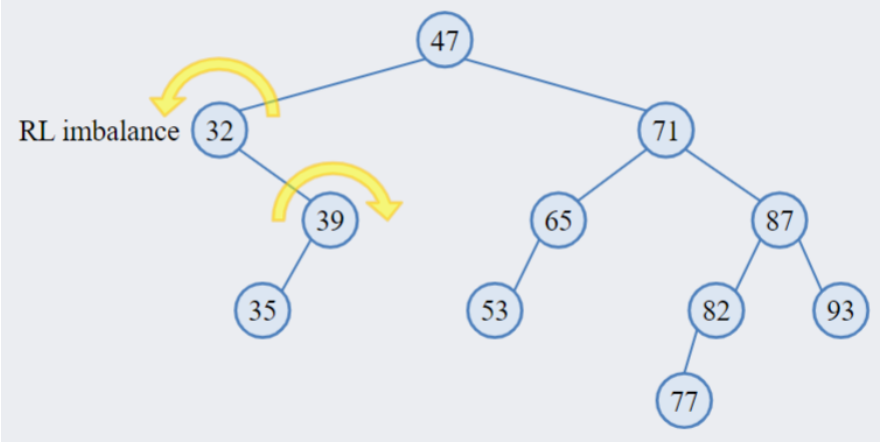
- Example
  - 

  - 

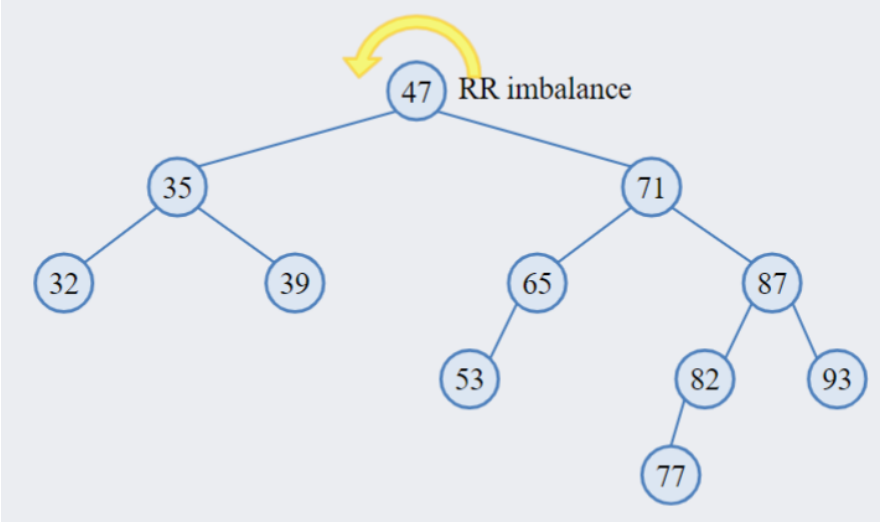
  - RR imbalance, perform `rotateLeft(39)`

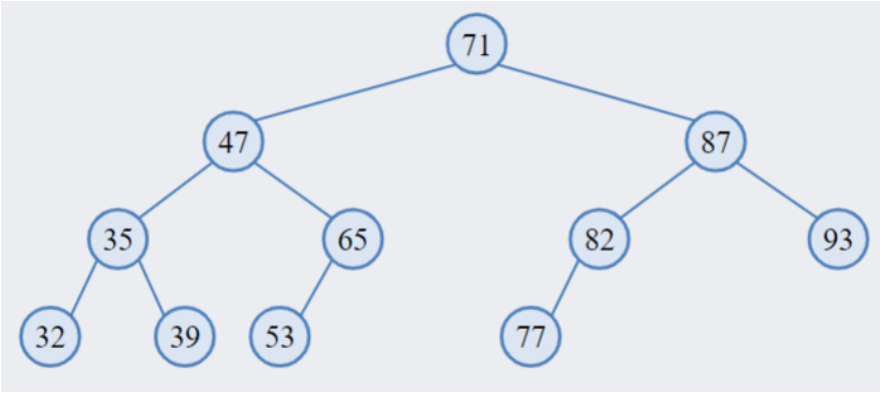
- 

- Example
  - 

  - Removing 25, would lead to the subtree to be RL imbalance. Fix by `rotateRight(35)`, then `rotateLeft(32)`
  - 

  - Then there is a RR imbalance

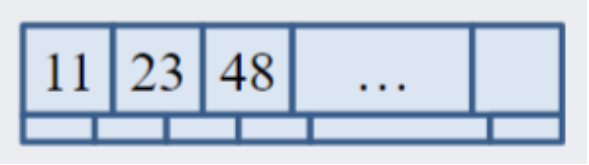
- 

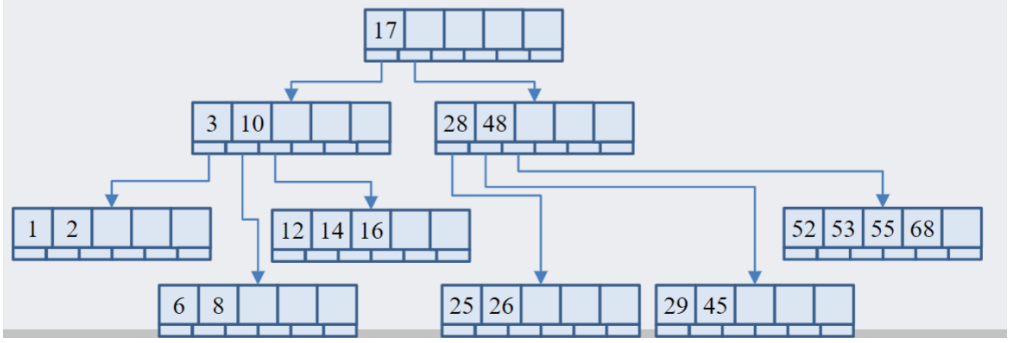
- Fix by `rotateLeft(47)`
- 

- Complexity: BST removal ($O(\log n)$) + Get predecessor ($O(\log n)$) + fixes (at most 2 rebalances), moving upwards ($O(\log n)$)

Mar. 4 - Week 9.1: BTrees

- AVL?
    - AVL balance invariant guarantees worst-case $O(\log n)$ time for find, insert, remove
    - Remember that nodes are allocated one at a time in dynamic memory
        - No guarantee that parents/children, siblings are near to each other in memory
        - No guarantee of spatial locality of reference
    - Realities
        - For large data sets, **disk accesses** dominate runtime
        - hierarchy of storage: registers, cache, RAM, disk
- A big BST
    - Suppose we have a **very** large data set stored in BST
        - so large that we cannot fit the entire tree inside of RAM

        - then the tree must reside on disk
    - Different levels of memory are accessed in blocks
        - For disk memory, the block unit is called a **page**
        - BST nodes may all reside in different blocks
        - a tree operation involving several nodes may require many expensive disk seeks (movement of the arm in the hard disk to the proper ring + data spinning to proper location)
    - Goal: put more relevant data together in a single block so they can be retrieved in a single disk access
- B-tree nodes
    - A binary search tree node has 1 key and up to 2 children - 2-ary tree
    - A B-tree node defines an m-ary tree: **branching factor** is m
        - Each node has up to $m-1$ keys and up to m children
        - nodes are still fully ordered
        - additional pointers: parent, left sibling, right sibling
        - 

    - Ideally, maximize of the size of a B-tree node to fill a disk block
        - in practice, branching factors > 1000 are commonly used
- B-tree properties
    - B-tree of order m is an ordered m-ary tree
        - internal node: $\#keys = \#children - 1$
        - all leaves are at the same depth
        - all leaves hold no more than $m-1$ keys
        - all non-root internal nodes have between $\lceil \frac{m}{2} \rceil$ and m children, minimum # of keys in non-root internal nodes is $\lceil \frac{m}{2} \rceil - 1$.
        - root can be a leaf or have between 2 and m children (root minimum of keys is 1)
        - 

        - 6-ary tree

- Non-root nodes must have at least 2 keys (3 valid child pointers)

- Search
  - ```
    bool btreeSearch(BtreeNode & nd, T key) {
        // Linear Search within node
        int i = 0;
        while ((i < nd.numkeys) && (nd.data[i] < key)) {
            i += 1;
        }
        if ((i < nd.numkeys) && (nd.data[i] == key)) {
            return true;
        }

        // Not found at leaf level
        if (nd.isLeaf()) {
            return false;
        } else {        // retrieve from disk and
    continue search
            BtreeNode b = DISK_READ(nd.child[i]);
            return btreeSearch(b, key);
        }
    }
    ```

  - Complexity
    - For each node, we must perform a linear search through $m - 1$ keys $\in O(m)$
      - Binary Search can improve to $O(\log m)$ per node
      - Typical implementations will hard-code the value of m to a constant, so $O(m)$ vs. $O(\log m)$ will be $O(1)$
    - How many nodes do we need to search? $O(height)$
    - Then: $O(m \cdot height)$
    - Worst height of a B-tree containing n keys?
      - Try to maximize height, and minimize the number of keys
      - Minimizing keys leads to minimizing the number of nodes, except for the root level
    ```

Mar. 5th - Week 9.2: BTree Analysis

- Recap
    - All leaves are at the same depth
    - All nodes except the root have between $\lceil \frac{m}{2} \rceil - 1$ and $m - 1$ keys (the root may have between 1 and $m - 1$ keys)
- Search Complexity
    - For each node, linear search $\in O(m)$
    - The number of nodes to search through $\in O(height)$
    - Search is $O(m \cdot height)$
    - Worst height of a B-tree containing n keys?
- Worst height analysis
    - Root node must have at least 1 key, which has at least 2 children
    - Each of both children has at least $\lceil \frac{m}{2} \rceil - 1$ keys, having $\lceil \frac{m}{2} \rceil$ children.
    - Let $t = \lceil \frac{m}{2} \rceil$
    - Thus, we have:

      Lvl. 0: 1 node, 1 key $= 1$

      Lvl. 1: 2 nodes, $t - 1$ keys $= 2(t - 1)$

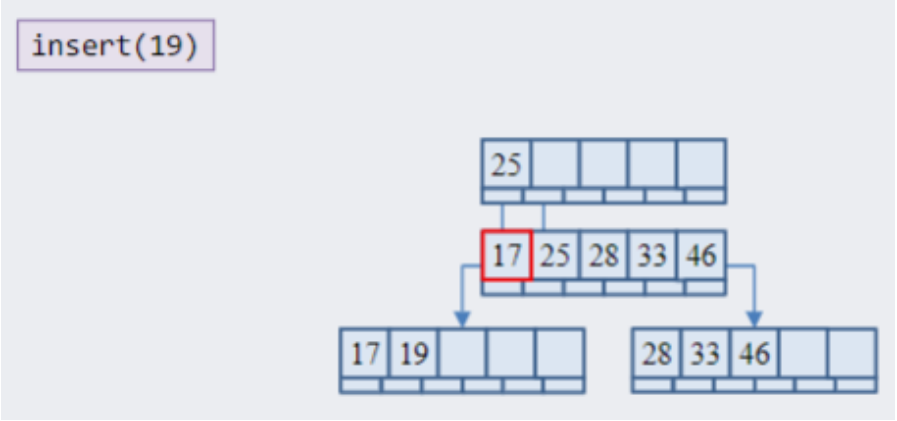
      Lvl. 2: $2t$ nodes, $t - 1$ keys $= 2t(t - 1)$

      ...

      Lvl. k: $2 \cdot t^{h-1}$ nodes, $t - 1$ keys $= 2t^{h-1}(t - 1)$
    - The other way is such: (x is number of nodes, n is number of keys)

      $$x = 1 + 2\sum_{i=0}^{h-1} t^i$$
      $$= 1 + 2(\tfrac{t^h - 1}{t - 1})$$
      $$n \geq 1 + 2(t - 1)(\tfrac{t^h - 1}{t - 1})$$
      $$\geq 1 + 2(t^h - 1)$$
      $$\geq 2t^h - 1$$
      $$h \leq \log_{\lceil \frac{m}{2} \rceil}(\tfrac{n+1}{2}) \in O(\log_m n)$$
- B-tree insertion
    - Like BST, we first search for the insertion location
    - Insertion always starts at a leaf node
        - if the node has space, insert the key into the node
        - otherwise, split the node and send the median value upwards
    - With space

        - Insert key at appropriate location in the node
    - Without space



- B-tree removal
    - Find node containing key to remove
    - If node is internal, swap key with predecessor (or successor)
    - Remove key which is now at leaf called X
    - While node X has $\max(0, \lceil \frac{m}{2} - 2 \rceil)$ keys, underflowing
        - If a sibling has a spare key: move it up, bring down parent's separator key; stop
        - Merge with a sibling and bring down parent's separator key
        - If X is root
            - If root has 0 keys and 1 child, remove root; make child the new root; stop
        - X = parent of X ? (continue fixing upwards)
- Thinking about B-trees
    - Remove is fast if node doesn't underflow or we can take from a sibling. Merging and propagation take more time.
    - Insert is fast if node doesn't overflow. (Could we give to a sibling?)

      Splitting and propagation take more time
    - Propagation is rare if m is large (Why?)
    - Repeated inserts and removes can cause thrashing (remember: push / pop, w/resizes?)
    - If $m = 128$, then a B-tree of height 4 will store at least $30,000,000$ items

Mar. 6th - Week 9.3: Hashing Introduction

	Insert	Remove	Find
Unordered array	$O(1)$	$O(n)$	$O(n)$
Ordered array	$O(n)$	$O(n)$	$O(\log n)$
Unordered list	$O(1)$	$O(n)$	$O(n)$
Ordered list	$O(n)$	$O(n)$	$O(n)$
BST	$O(n)$	$O(n)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

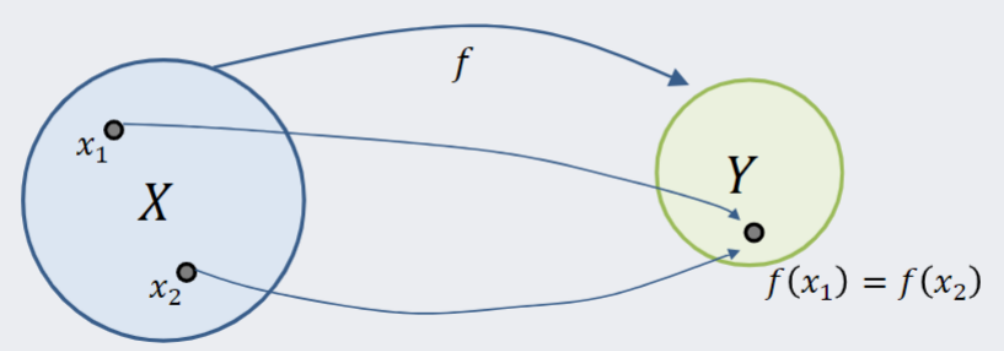
-
    - Return to arrays, what if we are allowed to leave gaps in our array, and we know the index of the element we want to access?
- A simple example
    - Suppose a company has 300 numbered lockers in its office building, and assigns a locker to every employee it hires
        - Suppose also, the company currently has 250 employees, and maintains information such as locker (employee) number, name, position, salary, etc
        - Every month when computing payroll, the company system must access an employee record, given a locker number
    - Why not simply using an array with 300 (plus one) elements, then we can do all operations in $O(1)$ time? What about scaling up?
- A larger example
    - Suppose we want to store some census data on Canadians with telephone numbers
        - telephone number in format (xxx)xxx-xxxx, can be conveniently converted to a number between 0 and 999,999,9999, let's use this as an array index
        - But considering that the entire population of Canada is approximately 40,528,396, so
            - over 99.6% of the array will be empty
            - And it probably will not fit in your computer's RAM
- Storing strings
    - What if we had to store data by name?
        - We would need to convert strings to integer indices
    - Here is one way to encode strings as integers

        - Assign a value between 1 and 26 to each letter
        - a = 1, z = 26 (regardless of case)
        - sum the letter values in the string
    - ```
      Insert("dog", description);
      Find("god");
      ```
- Finding unique string values
    - Ideally we would like to have a unique integer for each possible string
        - The "sum the letters" encoding scheme does not achieve this
    - There is a simple method to achieve this goal
        - Treat the string as a base 26 number
        - Multiply the letter's value by 26^i, where i is the position of the letter in the word
            - "dog" $= 4 \times 26^2 + 15 \times 26^1 + 7 \times 26^0 = 3101$
            - "god" $= 7 \times 26^2 + 15 \times 26^1 + 4 \times 26^0 = 5126$
        - But suppose we store strings of length 10, there are 26^{10} possible combinations, most of which are meaningless
    - A different approach
        - Do not determine the array size by the maximum possible number of keys
        - Fix the array size based on the amount of data to be stored
            - Map the key value to an array element
            - We still need to convert the key value to an integer index using a **hash function**
        - Basic idea behind hash tables
- Hash tables
    - A hash table consists of an **array** to store data
        - Data often consists of complex types, or pointers to such objects
        - One attribute of the object is designated as the table's key
    - A **hash function** maps a key to an array index in 2 steps
        - The key should be converted to an integer
        - And then that integer mapped to an array index using some function (often the modulo function)
- Collisions
    - A hash function may map two different keys to the same index, referred to as **collision**
      ```

        - Consider mapping phone numbers to an array size of 1000 where $h = \text{phone} \pmod{1000}$
    - A good hash function can significantly reduce the number of collisions
    - It is still necessary to have a policy to deal with any collisions that may occur
        - Collisions are actually unavoidable due to pigeonhole principle
    - Pigeonhole principle formally

      Let X and Y be finite sets where $|X| > |Y|$

      If $f : X \to Y$, then $f(x_1) = f(x_2)$ for some $x_1, x_2 \in X$, where $x_1 \neq x_2$

- Hash functions
    - Goal: Dictionary implementation with $O(1)$ access

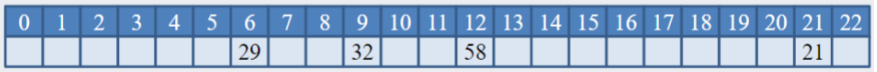
      key $\to$ index $\to$ array access
    - A hash function is a function that map key values to array indexes
    - Hash functions are performed in two steps
        - May the key value to an integer
        - Map the integer to a legal array index
    - Hash functions should have the following properties
        - Fast
        - Deterministic: non-random
        - Uniformity: spread out over the array, look like a uniform, random distribution
- Hash function speed
    - Hash functions should be fast and easy to calculate
        - Access to a hash table should be nearly instantaneous and in constant time
        - Most common hash functions require a single division or modulo on the representation of the key
        - Converting the key to a number should also be able to be performed quickly

- Deterministic
    - A hash function is **deterministic**
        - For a given input it must always return the same value
        - Hash functions should therefore not be determined by
            - System time
            - Memory location
            - Pseudo-random numbers
- Scattering data
    - A typical hash function usually results in some **collisions**
        - A **perfect** hash function avoids collisions entirely (each search key value maps to a different index)
    - The goal is to **reduce** the number and effect of collisions
    - To achieve this, the data should be distributed evenly over the table
- Uniformity
    - A good hash function generates each value in the output range with the same probability
        - Each legal hash table index has the same chance of being generated
    - This property should hold for the universe of possible values **and for the expected inputs**
        - The expected inputs should also be scattered evenly over the hash table
    - "Similar" keys should not map to "similar" indices
- A bad hash function
    - A hash table is to store 1000 numeric estimates that can range from 1 to 1,000,000
        - Hash function: $h(estimate) = estimate\%n$
        - when $n = 1000$
    - The distribution of values from the universe of all possible values is uniform. Estimates usually be round multiples of 10, 100, 1000.
- Another bad hash function
    - A hash table is to store 676 names
        - The hash function considers just the first two letters of a name
        - Function = (1st letter $\times 26$ + 2nd letter) % 676
    - The expected values' distribution is not uniform.

Mar. 11th - Week 10.1: Hashing Collisions

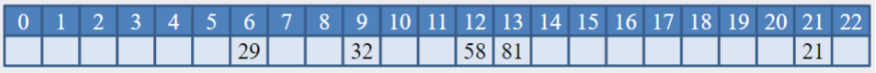
- Collision handling
    - A collision occurs when 2 different keys are mapped to the same index
        - Collisions may occur even when the hash function is good
        - Inevitable due to pigeonhole principle
    - Two main ways of dealing with collisions
        - Open addressing
        - Separate chaining
- Open addressing
    - When an insertion results in a collision, look for an empty array element
        - Start at the index to which the hash function mapped the inserted item
        - Look for a free space in the array following a particular search pattern, known as **probing**
    - 3 schemes
        - Linear probing
        - Quadratic probing
        - Double hashing
- Linear probing
    - The hash table is searched sequentially
        - Starting with the original hash location
        - For each time the table is probed (for a free location) add one to the index
            - Search $h(\textit{search key}) + 1$, then $h(\textit{search key}) + 2$, and so on until an available location is found
            - If the sequence of probes reaches the last element of the array, wrap around to `arr[0]`
    - Problem: **primary clustering**
        - The table contains groups of consecutively occupied locations
        - These clusters tend to get larger as time goes on
            - Reducing the efficiency of the hash table
    - Example
        - Hash table is size 23, the hash function is then $h(x) = x \pmod{23}$, x is the search key value

- - Insert 81, h = 81 mod 23 = 12
  - Which collides with 58 so use linear probing to find a free space
  - First look at 12 + 1, which is free so insert the item at index 13



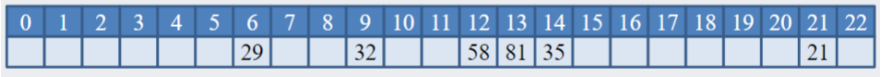
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

- - Insert 35, h = 35 mod 23 = 12
  - Which collides with 58 so use linear probing to find a free space
  - First look at 12 + 1, which is occupied so look at 12 + 2 and insert the item at index 14



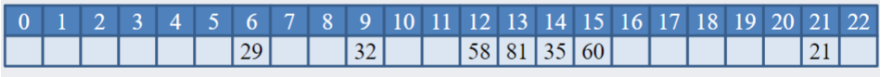
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81								21	

- - Insert 60, h = 60 mod 23 = 14
  - Note that even though the key doesn't hash to 12 it still collides with an item that did
  - First look at 14 + 1, which is free



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35							21	

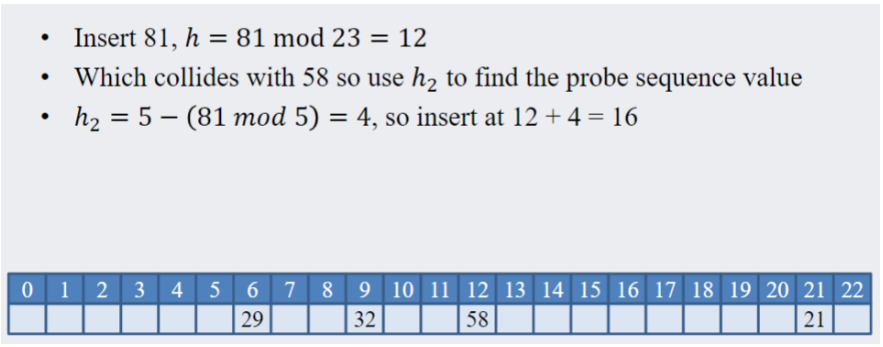
- - Insert 12, h = 12 mod 23 = 12
  - The item will be inserted at index 16
  - Notice that primary clustering is beginning to develop, making insertions less efficient

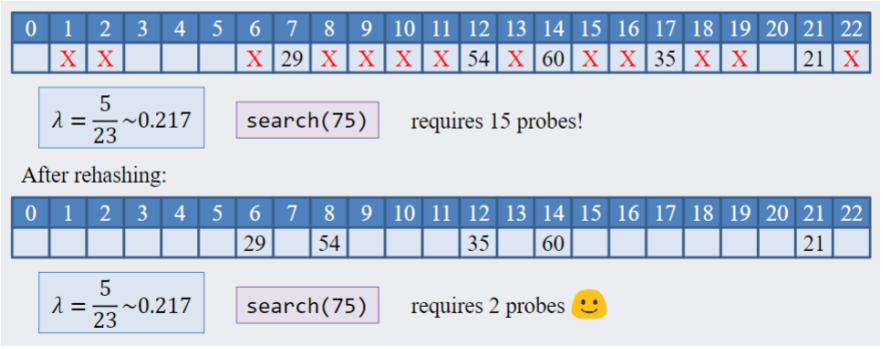


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	60						21	

  - The probability that a new insertion's hash value that lands in the primary cluster, is currently $\frac{5}{23}$ (12 to 16), and by linear probing, it has to move down the array, making the primary cluster even bigger. The probability of landing in the cluster thus become larger, forming a positive feedback loop.
- Search
  - Similar to insertion, wish to achieve $O(1)$ as best as we can
  - Whatever is used for probing, we need to use the same method for searching
  - Say we are looking for 59, its remainder is 13 when divided by 23, since it is occupied, so we move right to see if there is more elements.

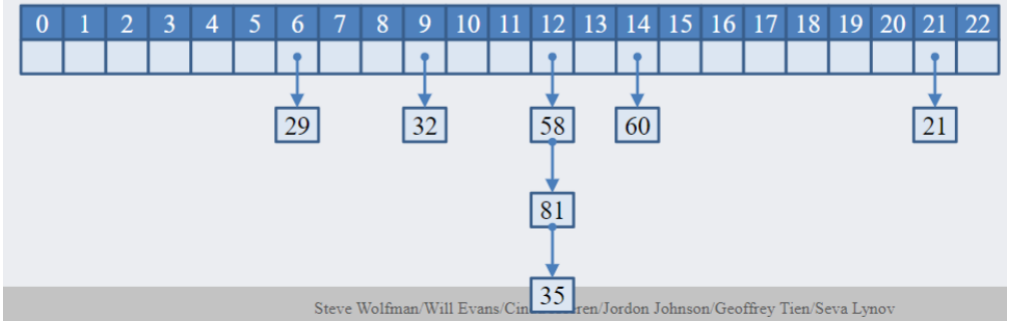
        - Terminates when item is found, empty space, or entire table is searched
- Hash Table Efficiency
    - When analyzing the efficiency of hashing, it is necessary to consider **load factor**, λ
        - λ = number of items / table size
        - As the table fills, λ increases, and the chance of a collision occurring also increases
            - Performance decreases as λ increases
            - Open addressing: λ has a maximum value of 1
        - Unsuccessful searches make more comparisons
            - An unsuccessful search only ends when a free element is found
    - It is important to base the table size on the largest possible number of items
        - The table size should be selected so that λ does not exceed $\frac{1}{2}$
        - Consider an array half full with a poor distribution (cluster altogether in the second half of the array), assume we use linear probing
            - Then, if we insert one element, this element landing in the first half has probability $\frac{1}{2}$, it would have $\frac{1}{2} \times 1$ operations
            - The element landing in the second half would have different probability for different number of operations, namely $\frac{1}{n} \times \sum_{i=1}^{\frac{n}{2}} i$ operations
            - This should be $\in O(n)$
        - Consider an array half full with a good distribution (elements alternate in the array), then assume we still use linear probing
            - We would have in total of $\frac{1}{n} \times (2+1) \times \frac{n}{2}$ operations
            - This would be $\in O(1)$
- Double hashing
    - In linear probing, the probe sequence is independent of the key
    - Double hashing produced **key dependent** probe sequences
        - In this scheme a second hash function, h_2, determines the probe sequence
    - Guidelines

        - $h_2(key) \neq 0$
        - $h_2 \neq h_1$
        - A typical h_2 is $p - (key \mod p)$ where p is a prime number
    - Example
        - 

- **Array capacity should be a prime number**
    - The insertion can become cyclical if the array capacity is not a prime number
- Removals and open addressing
    - Removals add complexity to hash tables
        - Easy to find and remove a particular item
        - What happens when you want to search for some other item
        - The recently empty space may make a probe sequence terminate prematurely
    - One solution is to mark a table location as either empty, occupied or removed (tombstone)
        - Locations in the removed state can be re-used as items are inserted (after confirming non-existence)
    - Tombstones and performance
        - After many removals, the table may become clogged with tombstones which must still be scanned as part of cluster (it may be beneficial to periodically re-hash all valid items)
        - 

- Separate chaining
    - Separate chaining takes a different approach to collisions

- Each entry in the hash table is a pointer to a linked list (or other dictionary-compatible data structure)
    - If a collision occurs the new item is added to the end of the list at the appropriate location
- Performance degrades less rapidly using separate chaining
    - with uniform random distribution, separate chaining maintains good performance even at high load factors $\lambda > 1$
- - $h(x) = x \bmod 23$
  - Insert 81, $h(x) = 12$, add to back (or front) of list
  - Insert 35, $h(x) = 12$
  - Insert 60, $h(x) = 14$



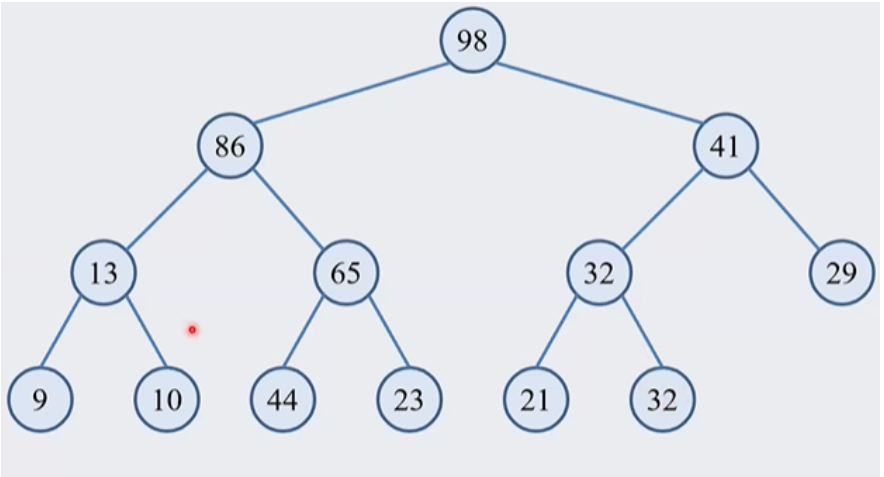
- Hash table discussion
    - If $\lambda < \frac{1}{2}$, open addressing and separate chaining give similar performance
        - As λ increases, separate chaining performs better than open addressing
        - However, separate chaining increases storage overhead for the linked list pointers
    - It is important to note that in the worst case hash table performance can be poor
        - That is, if the hash function does not evenly distribute data across the table
        - Almost always $O(1)$ with separate chaining

Mar. 12th - Week 10.2: Priority queues

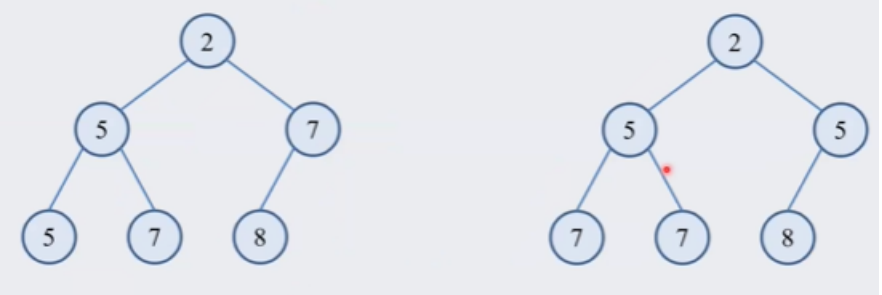
- Priority queues
    - Suppose a to-do lists (with priority values - higher values more important)
        - 6 - PA3, 2 - Vacuum home, 8 - Draft Examlet...

        - Interested in quickly finding the task with the highest priority
- Priority Queue ADT
    - A collection organized so as to follow fast access to and removal of the largest (smallest) element
        - **Prioritization** is a weaker condition than **ordering**
        - Order of insertion is irrelevant
        - Element with the highest priority is the first element to be removed
    - Operations
        - `create, destroy, insert, removeMin/removeMax, isEmpty`
    - Property
        - For two elements x and y in the queue, if x has a higher **priority** value than y, x will be removed before y regardless of order of insertion
        - Note that in most definitions, a single priority queue structure supports **only** `removeMin`, or **only** `removeMax`, and not both
        - A priority queue is an ADT that maintains a multiset of items (allows duplicate entries, duplicate in priority)
        - 2 or more distinct items in a priority queue may have the same priority
        - If all items have the same priority, not necessarily like FIFO like a queue (depends on implementation details)
    - Applications
        - Hold jobs for a printer in order of size
        - Manage limited resources such as bandwidth on a transmission line from a network router
        - Sort numbers
        - Anything **greedy**: an algorithm that makes the "locally best choice" at each step.
    - Data structures for priority queues

Structure	insert	removeMin
Unordered array	$O(1)$	$O(n)$
Ordered array	$O(n)$	$O(n)$
Unordered list	$O(1)$	$O(n)$
Ordered list	$O(n)$	$O(1)$
Binary search tree	$O(n)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$

  - What about HashTables? If we use priority value as keys, we cannot identify a minimum elements as hash function can put elements all across the table. $O(capacity)$
- Binary Heap (insert - $O(\log n)$, `removeMin` - $O(\log n)$)
  - Heap has asymptotically performance as AVL tree, but MUCH simpler to implement
  - A **heap** is binary tree with 2 properties
    - Heaps are **complete**
      - All levels, except the bottom, must be completely filled in
      - The leaves on the bottom level are as far to the left as possible
    - Heaps are **partially ordered**
      - For a **max** heap - the value of a node is at least large as its children's values
      - For a **min** heap - the value of a node is no greater than its children's values
  - Max heap example
    - 

  - Duplicate priority values

    - It is important to realize that 2 binary heaps can contain the same data, but items may appear in different positions in the structure
    - 

- Implementation
    - Heaps can be implemented using **arrays**
    - There is a natural method of indexing tree nodes
        - Index nodes from top to bottom and left to right (level-order indexing)
        - Because heaps are **complete** binary trees, so there are no gaps in the array
- Referencing nodes
    - To move around in the tree, it will be necessary to find the index of the parents of a node (or the children of a node)
    - The array is indexed from 0 to $n-1$
    - Each level's nodes are indexed from $2^{level}-1$ to $2^{level+1}-2$
        - The children of a node i, are the array elements indexed at $2i+1$ and $2i+2$
        - The parent of a node i, the array element is indexed at $\frac{i-1}{2}$
    - Right children are always even numbers, left children are always odd
- ```
  template <class LIT>
  class MinHeap {
      private:
          int size;       // num of stored elements
          int capacity;   // max capacity of array
          LIT * arr;      // array in dynamic memory
      public:
          ...
  }

  template <class LIT>
  MinHeap::MinHeap(int initcapacity) {
  ```
  ```

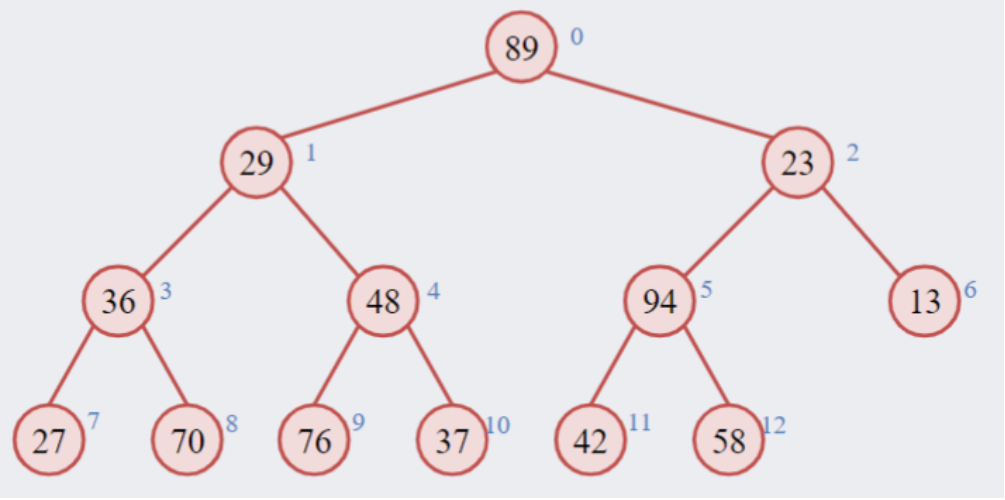
```
        size = 0;
        capacity = initcapacity;
        arr = new LIT[capacity];
    }
```

- Heap insertions
    - On insertion the heap properties have to be maintained
    - The algorithm first ensures that the tree is complete
        - new item is the first-available (left-most) leaf on the bottom level
    - Fix the partial ordering
        - Compare the new value to its parent
        - Swap them if the new value is greater than the parent
        - Repeat until this is not the case (heapify up, percolate up, bubble up, trickle up)
    - Complexity
        - Item is inserted at the bottom level in the first available space $\in O(1)$
        - Repeated heapify-up operations
            - Each heapify-up operation moves the inserted value up one level in the tree
            - Upper limit on the number of levels in a complete tree is $O(\log n)$
        - Heap insertion has worst-case performance of $O(\log n)$
- Building a heap?
    - A heap can be constructed by repeatedly inserting items into an empty
- Removing the priority item
    - Properties must also be satisfied after removal
    - Make a temporary copy of the root's data
    - Similarly to the insertion algorithm, we first ensure that the heap remains complete
        - Replace the root node with the right-most leaf
    - Swap the new root with its **largest valued child** (max heap) until the partially ordered property holds i.e. `heapifyDown`
    - Return the copied root's data
    - Complexity
        - Analysis is similar to insertion

        - Replace root with last element $\in O(1)$
        - Repeated heapify-down operations starting from root level, $\in O(\log n)$
- Array implementation and insertion
    - We have limited capacity at creation time
    - When array is full, we expand it
    - Since array indices correspond exactly to node positions in the tree, and nodes should remain in their original positions after expanding the array, we can simply copy into the same indices

Mar. 13th - Week 10.3: Heaps

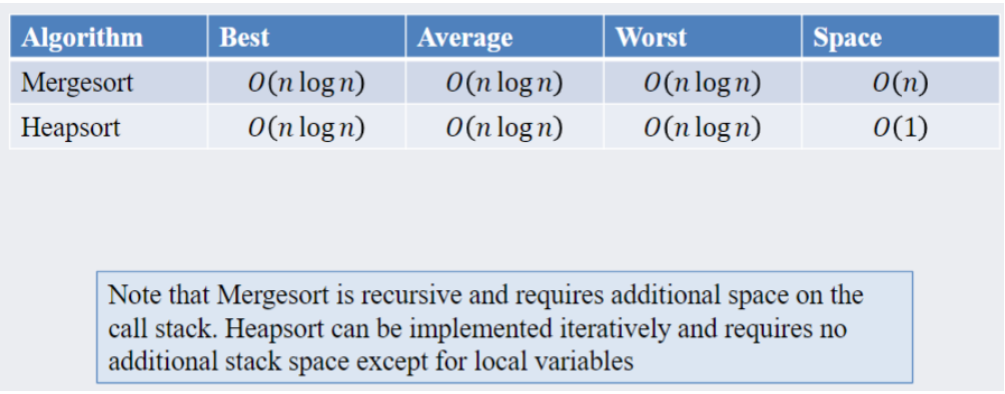
- Creating heaps
    - To create a heap given a list of items
        - Create an empty heap
        - For each item: insert into heap
        - e.g. create a (min) heap from the items, complexity?
            - $O(n \cdot \log n)$, about half of the elements will be at the bottom 2 levels, so they are consuming $\log n$ time
- Build Heap
    - Starting with an unordered array (Floyd's method)



        - Transform it to a (max) heap: key value in every node must be larger than key value of children
        - Given the tree representation of some unordered array, the leaf nodes always satisfy the condition for a heap
    - To create a heap from an unordered array, repeatedly call `heapifyDown`
        - Any subtree in a heap is itself a heap
        - Call `heapifyDown` on elements in the upper $\frac{1}{2}$ of the array

        - Lower half are leaf nodes and already heaps
    - Start with index $\frac{n}{2}$ and work up to index 0
        - From the last non-leaf node to the root
- `heapifyDown` does not need to be called on the lower half of the array (the leaves)
    - since it restores the partial ordering from any given node down to the leaves, the heap property remains satisfied at deeper levels of the tree
- Complexity
    - `heapifyDown` is called on half of the array $\in O(height)$
    - It would appear that `buildHeap` cost is $O(n \log n)$
    - In fact the cost is $O(n)$
    - Look at the total number of swaps that can occur in the worst case
        - the bottom level is full, and contains all the highest priority values (for a max heap)
    - `heapifyDown` must follow some path (along edges) to the bottom of the tree
        - For the level just above leaf nodes, each node will perform at most 1 swap (along 1 edge) = $\frac{n}{4}$ swaps
        - One level above (2 levels from bottom), each node will perform at most 2 swaps (along 2 edges) = $\frac{n}{8} \times 2 = \frac{n}{4}$ swaps
        - so, k levels from bottom, it should still perform at most $\frac{n}{4}$ swaps
        - The upper bound is $n - 1$ edges (many are not even swapped along), so the worst case number of swaps is $O(n)$
- Heapsort
    - Heapify the array (max heap)
    - Floyd's method to rearrange
    - Repeatedly remove the root (largest item)
        - After each removal swap the root with the last element in the tree
        - The array is divided into a heap part and a sorted part
    - At the end of the sort the array will be sorted in ascending order
    - Complexity: $O(n)$ (Floyd's method) + $O(n \log n) = O(n \log n)$

- Heapsort is also in-place, i.e. does not require allocating any temporary space
- 

- Algorithmic comparisons
- 

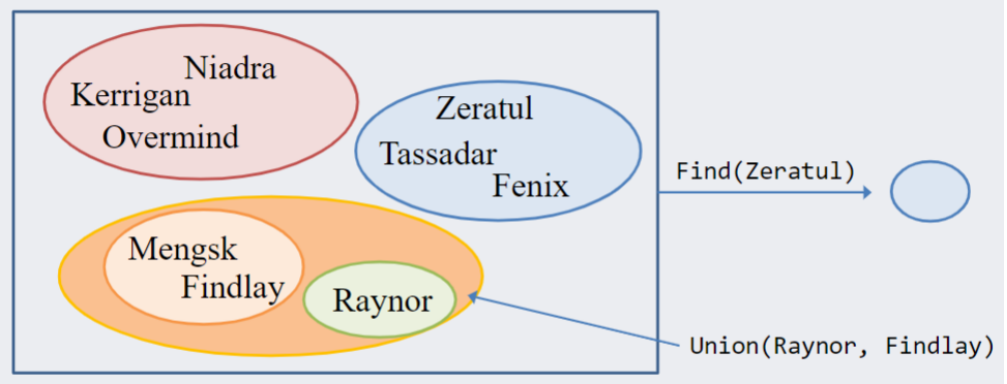
Algorithm	Best	Average	Worst	Space
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

Note that Mergesort is recursive and requires additional space on the call stack. Heapsort can be implemented iteratively and requires no additional stack space except for local variables

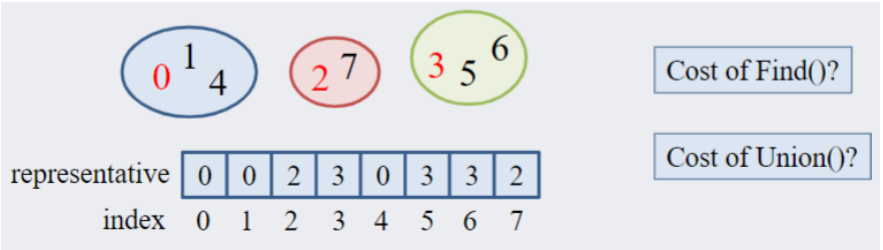
- Merge-sort space complexity: $O(n) + O(\log n)$ (temporary array + recursion stack)

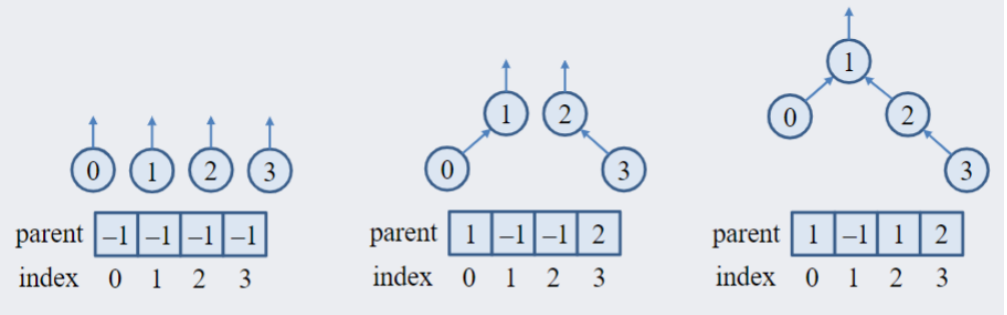
Mar. 18th - Week 11.1: Disjoint Sets

- New ADT about sets
    - Imagine a group of people, some of whom **trust** each other
        - A trusting pair shares secrets
        - Untrusting pairs do not
    - But, a person can learn your secrets from someone you both trust, or some more complex chain
    - How can we determine which groups or individuals a secret will spread to?

        - Applications: contagion analysis, maze building, laying cable infrastructure
- Set ADT operations
    - Initialize a network of people (none connected) - `MakeSet`
    - Each time we learn that a pair trusts each other, merge them into the same set - `Union`
    - Determine if people share a trust group by finding which set each is in (two `Find` operations)
- Disjoint Set ADT
    - No element is the member of two different sets
    - Disjoint Set operations
        - Create
        - Destroy
        - Union
        - Find
    - Property: `Find` on x produces the same result as `Find` on y if and only if x and y are in the same set
        - `Union`-ing x and y causes their sets to permanently merge
    - 

    - We don't care about the return type of `Find`, as long as it can be compared to see if two sets are the same
- A data structure for disjoint sets
    - Maintains a collection $S = \{s_0, s_1, \ldots, s_n\}$ of disjoint sets
    - Each set has a representative member
    - Required operations
        - `void MakeSet(int n)`
        - `Union(int x, int y)`
        - `void* Find(int x)`
    - Start with array-based structure

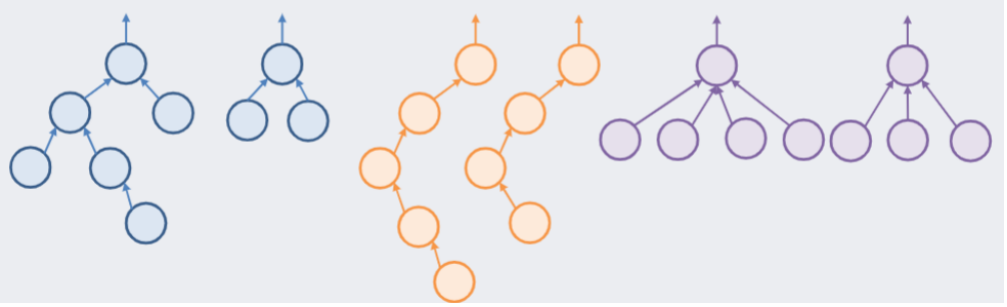
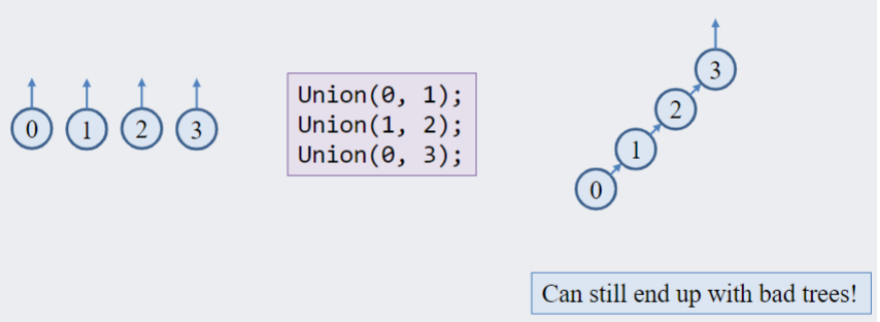
        - index as element ID
        - the content of element is the "leader" of the set
        - 

        - Cost of `Find()`: access the corresponding index and return $\in O(1)$
        - Cost of `Union()`: worst case, needs to search and change at most $O(n)$ elements, then $\in O(n)$
- A better structure for disjoint sets (Uptrees)
    - A tree where a node points to its parent
        - Still array-based, but representative is the root of the tree
            - If array value is -1, then the index is a root node
            - otherwise, the array value is the index's parent
        - No limit on the branching factor
    - x and y are in the same tree $\iff$ x and y are in the same set
    - 

    - Tree-based disjoint sets

```
int DisjointsSets::Find(int x) {
    if (parent[x] < 0) {
        return x;
    } else {
        return Find(parent[x]);
    }
}
```

    - Running time: worst case $\in O(n)$, best case $\in O(1)$, average case $\in O(\log n)$

- 
- Union: given arbitrary indices x and y, join their trees
    - naively: set root of x to y, or vice-versa
    - slightly better: set root of x to root of y, or vice-versa
    - 

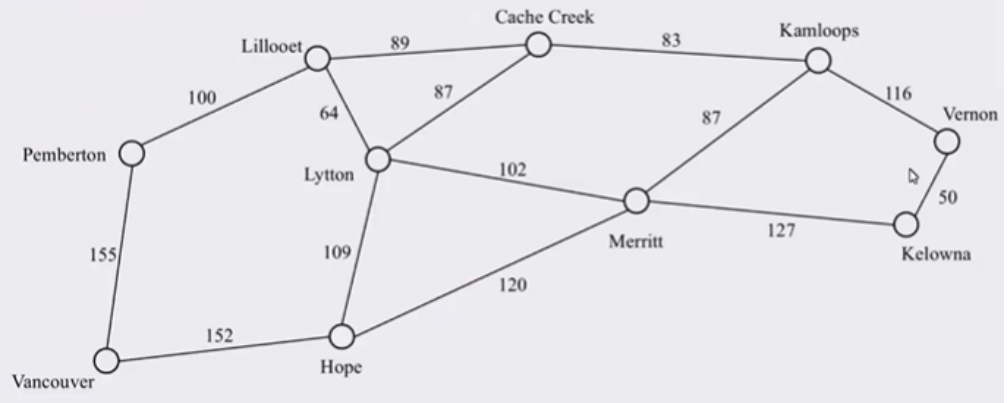
- "Smart" union / "Ranked" union
    - Union by height: keeps overall tree height as small as possible, attach the shorter tree to the taller tree
    - Union by size: Increases distance from root for as few nodes as possible
    - Both scheme guarantee that the height of the tree is $O(\log n)$
    - Analysis
        - After `MakeSet(n)`, we have n elements from 0 to $n-1$ in each of their own sets
        - Say `Union(0, 1)`, now 1 is 0's parent.
        - To make the tree as bad as possible (increasing the height), if we further `Union(0, 3)`, then the new element 3 will become 1's child
        - Thus, to make trees taller, we can only `Union` two trees with the same height; so instead of `Union(0, 3)`, we first perform `Union(2, 3)`, then we perform `Union(1, 2)`. This would increase the height to 2.
        - By this logic, we need to construct another tree with height 2 before union again.
        - We start n trees of height 0, then they form $\frac{n}{2}$ trees of height 1, ..., eventually they form $\frac{n}{2^k}$ of height k, where $k = \log n$ to be left with 1 tree in the end
- Path compression

- During a `Find` operation, we follow a path up the tree through a sequence of nodes
    - We look up a number entries in an array, where each lookup is $O(1)$
- The path of nodes can then be attached to the root directly reducing future `Find`
- We add an $O(1)$ operation for each entry we process: set the parent of each node along the path, to the root found at the end of the path
- The more `Find`, the better the performance gets **almost** $O(1)$, true complexity of smart union with path compression is $O(\log^*(n))$

Mar. 19th - Week 11.2 - Intro to Graph

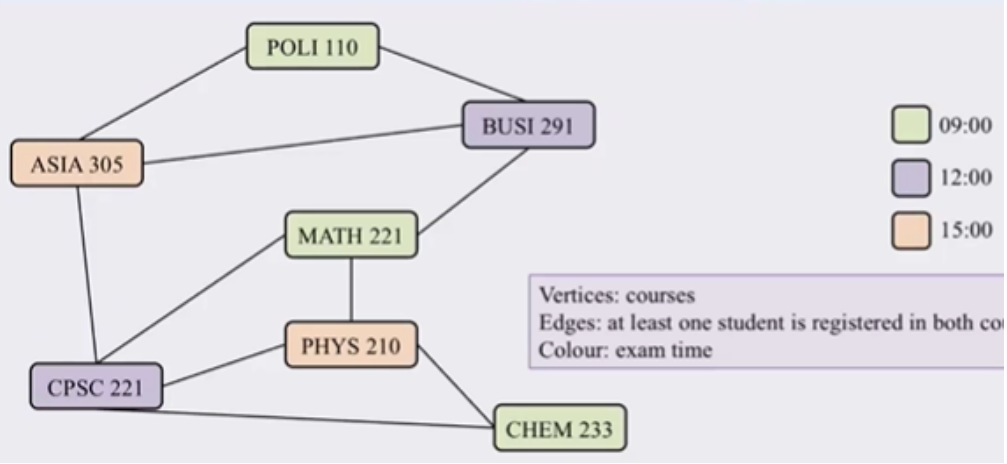
- What's a graph?
    - A graph G is a collection of **vertices** (or nodes) connected by **edges**
    - Formally, a graph is a pair of sets: $G = (V, E)$
        - V is a set of **vertices**: $\{v_1, v_2, \ldots, v_n\}$
        - E is a set of **edges**: $\{e_1, e_2, \ldots, e_m\}$ where each e_i is a pair of vertices: $e_i \in V \times V$
    - If each edge is an ordered pair, then the graph is **directed**, otherwise the graph is **undirected**
- Graph applications
    - Storing things that are "networks" by nature
        - Road networks, airline flights, relationships between people/things
    - Compilers
        - Call graph - which functions call which others
        - Dependency graphs - which variables depend on which others
    - Others
        - Circuits, class hierarchies, graphics meshes, computer networks, exam scheduling

- Example 1:



  - Efficient: Vancouver - Hope - Merritt - Kamloops
  - Tour: Vancouver - Pemberton - Lillooet - Cache Creek - Kamloops
  - This is a **weighted** graph
  - Vertices: cities; Edges: A highway connects the cities, with driving distance

- Example 2:



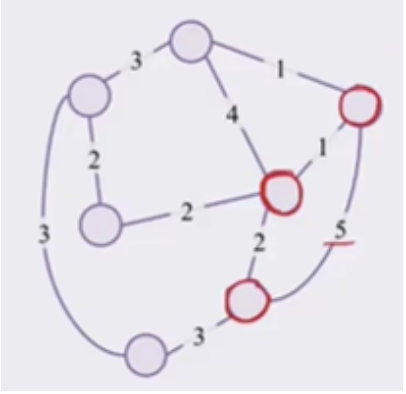
  - Vertices: courses; Edges: at least one student is registered in both courses at the endpoints; Color: exam time
  - Conflict-free schedule?

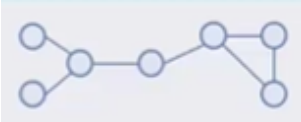
- Example 3:

  - Find a sequence of moves (path in graph) to the winning board arrangement

        - Vertices: arrangements of the game board; Edges: a valid move that transforms one arrangement to another
        - 

- Graph vocabulary
    - Vertices adjacent to v: $N(v) = \{u|(u, v) \in E\}$
    - Edges incident to v: $I(v) = \{(u, v)|u \in N(v)\}$
    - Degree of v: $\deg(v) = |I(v)|$
    - Path: sequence of vertices connected by edges
        - Simple path: a path with no repeated vertices
        - Cycle: Path with same start and end vertex
    - Simple graph: No self-loops or multi-edges
        - (v, v) is a self-loop, if v is a vertex
        - Multi-edge: more than one edge between two vertices
    - Subgraph of $G = (V, E) : (V' \subset V, E' \subset E)$ and if $(u, v) \in E'$ then $u, v \in V'$
        - Can leave vertices alone, but cannot leave edges alone
    - Complete graph: maximum number of edges
    - Connected graph: path exists between every pair of vertices
    - Connected component: maximal connected subgraph
    - Acyclic graph: no cycles
    - Spanning tree of $G = (V, E)$: Acyclic, connected graph with vertex set V
- Weighted graphs
    - In a **weighted graph**, each edge is assigned a weight (edges are labeled with their weights)
    - Each edge's weight represents the cost to travel along that edge
        - The cost could be distance, time, money, or some other measure
        - The cost depends on the underlying problem

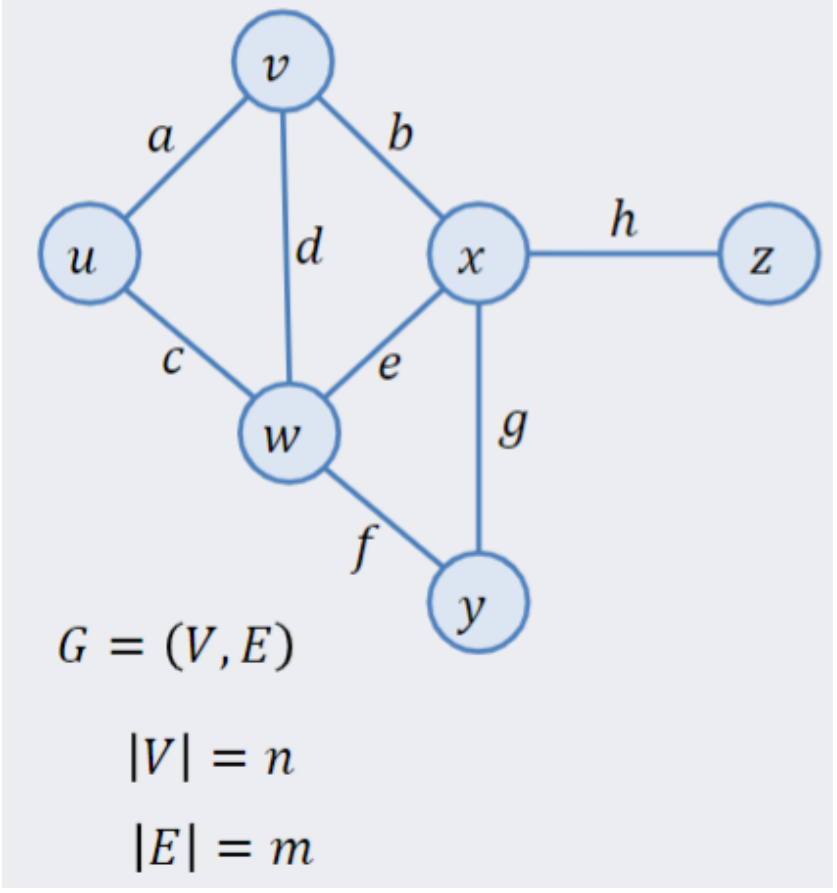
- 

- Connectivity
  - Undirected graphs are **connected** if there is a path between any two vertices 
  - Directed graphs are **strongly connected** if there is a directed path from any vertex to any other 
  - Directed graphs (digraphs) are **weakly connected** if there is a path between any two vertices, ignoring direction 
  - A complete graph has an edge between every pair of vertices 
- Isomorphism and subgraphs
  - We often care only about the structure of a graph, not the names of its vertices. Then, we can ask:
    - "Are two graphs **isomorphic**": do the graphs have identical structure / can you "line up" their vertices so that their edges match
    - "Is one graph a **subgraph** of another": is one graph isomorphic to a part of the other graph (a subset of vertices and a subset of edges connecting those vertices)
      - $G' = (V', E'), V' \subset V, E' \subset E$, if $(u, v) \in E'$ then $u, v \in V'$
  - Example applications
    - Identifying chemical compounds in a chemical database
    - Determining whether two different circuits are the same
- Degree

        - Handshaking theorem: if $G = (V, E)$ is an undirected graph, then $\sum_{v \in V} \deg(v) = 2|E|$
            - Each edge is counted twice
        - For directed graphs
            - The **in-degree** of a vertex is denoted as $\deg^-(v)$ and is the number of edges entering v
            - The **out-degree** of a vertex is denoted as $\deg^+(v)$ and is the number of edges leaving v
            - We let $\deg(v) = \deg^-(v) + \deg^+(v)$
            - Then: $\sum_{v \in V} \deg^-(v) + \sum_{v \in V} \deg^+(v) = \frac{1}{2} \sum_{v \in V} \deg(v) = |E|$
- Graph density
    - A **sparse** graph has $O(|V|)$ edges
    - A **dense** graph has $\Theta(|V^2|)$ edges
    - Anything in between is either on the sparse side or on the dense side, depending critically on context
    - Analysis of graph operations typically must be expressed in terms of both $|V|$ and $|E|$

Mar. 20th - Week 11.3: Graph ADT

- Theory

- 

    - Running times are often reported in terms of n, but they often depend on m
    - Edges (at least)
        - Connected: $n - 1$
        - Not connected: 0
    - Edges (at most)
        - Simple: $\frac{n(n-1)}{2}$
        - Not simple: ∞, unbounded
    - Relationship to degree sum
        - $\sum_{v \in V} \deg(v) = 2m \in O(m)$
        - useful for analysis of algorithms that iterate over the neighbors of each vertex
- Edges in a connected graph
    - Theorem: A minimal connected graph $G = (V, E)$ has $|V| - 1$ edges
    - Proof: Consider an arbitrary minimal connected graph $G = (V, E)$

      Lemma: Every connected subgraph of G is minimally connected

      IH: $\forall j < |V|$, any minimal connected graph of j vertices has $j - 1$ edges

      Suppose $|V| = 1$: a minimal connected graph of 1 vertex has no edges - $0 = 1 - 1$

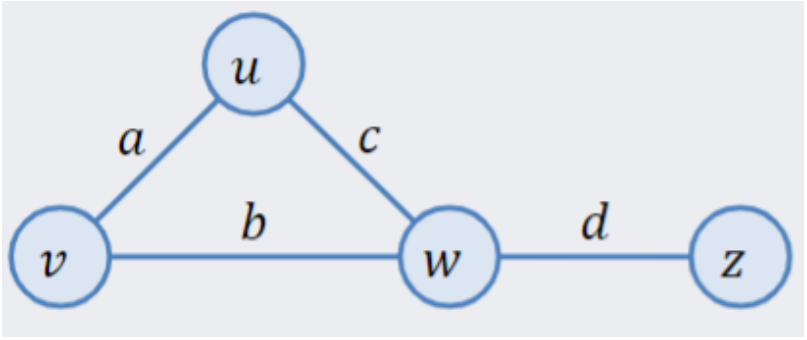
Suppose $|V| > 1$: Choose any vertex and let d denote its degree, set aside its incident edges, partitioning the graph into d components, $C_1 = (V_1, E_1), \ldots, C_d = (V_d, E_d)$, each of which is a minimal connected subgraph of G. This means that $|E_k| = |V_k| - 1$, by IH

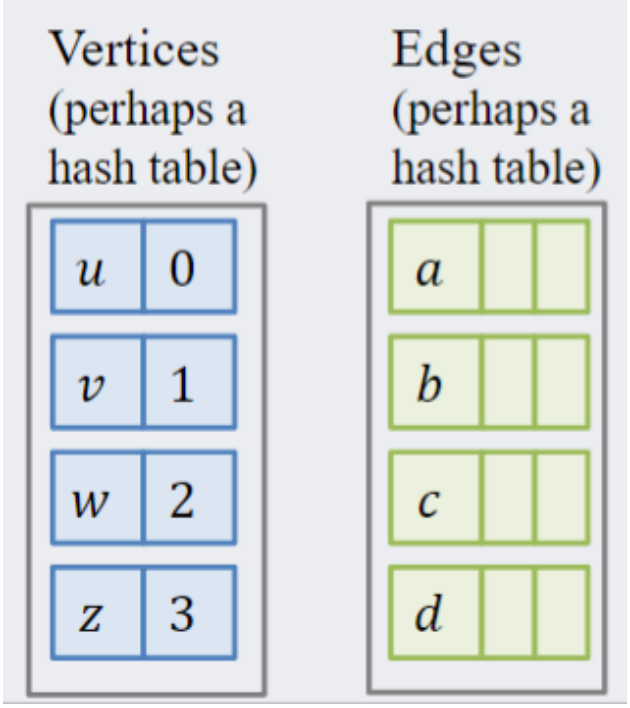
Now we add up the edges in the original graph:

$$d + \sum_{k=1}^{d} |E_k| = d + (|V_1| - 1) + \cdots + (|V_d| - 1) = \sum_{k=1}^{d} |V_k|$$

which is equivalent to $|V| - 1$, the number of vertices with the chosen vertex removed

- Graphs - toward implementation (ADT)
    - Data
        - Vertices + data
        - Edges + data
        - some structure that reflects the connectivity of the graph
    - Functions
        - `insertVertex(T1 vertexData)`, `insertEdge(vertex v1, vertex v2, T2 edgeData)`
        - `removeEdge, removeVertex`
        - `incidentEdges, areAdjacent`
    - Directed graph
        - `origin, destination`
- Adjacency matrix
    -





- Space: $\Theta(n), \Theta(m)$

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

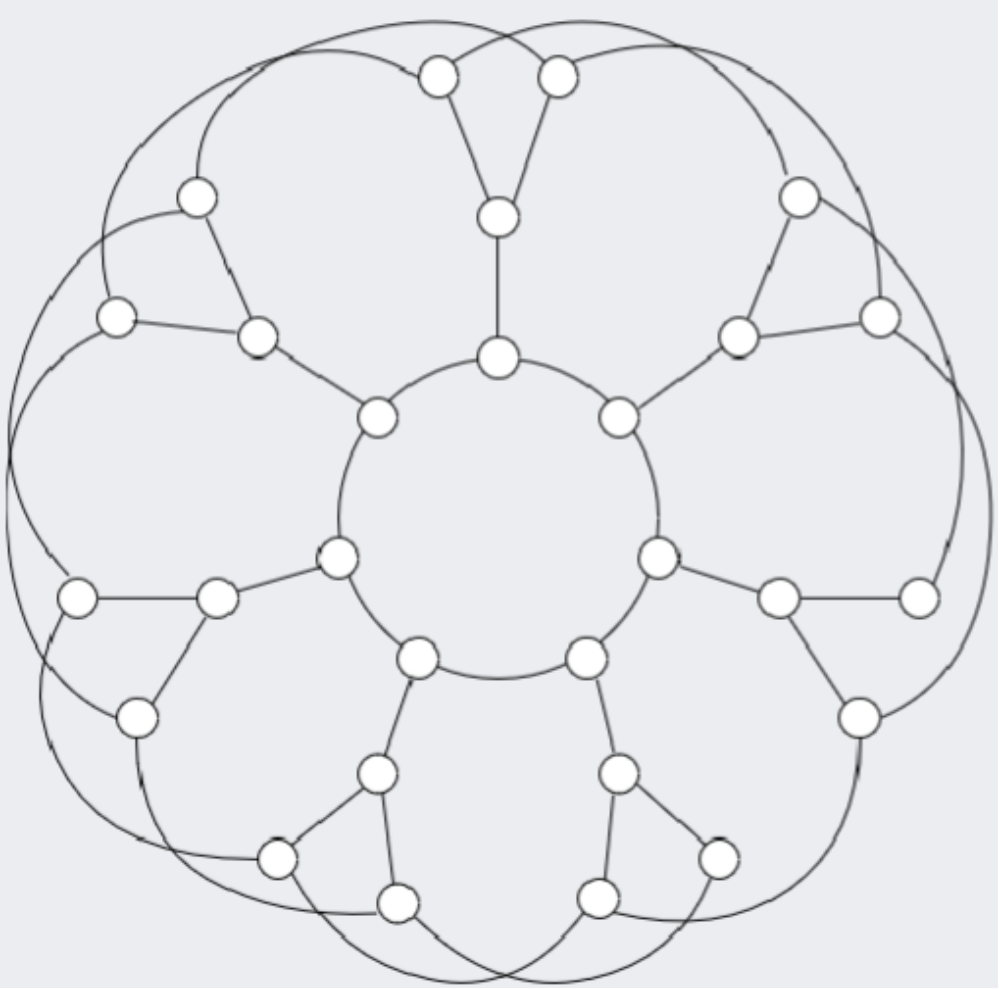
  - Symmetric for undirected graph
  - $\Theta(n^2)$
- Function analysis
  - `insertVertex`: $O(n)$ amortized
  - `removeVertex`: $O(1)$ or $O(n)$ amortized
  - `areAdjacent`: $O(1)$ amortized
  - `incidentEdges`: $O(n)$ amortized

- Adjacency list
  - For vertex table: each vertex has a doubly-linked list
  - Function analysis
    - `insertVertex`: $O(1)$ amortized, just add to the vertex table
    - `removeVertex`: need to remove neighbor's connection as well, $\in O(\deg(v))$
    - `areAdjacent`: $O(\max(\deg(v), \deg(u)))$, if size of neighbor is tracked $O(\min(\deg(v), \deg(u)))$
    - `incidentEdges`: $O(\deg(v))$

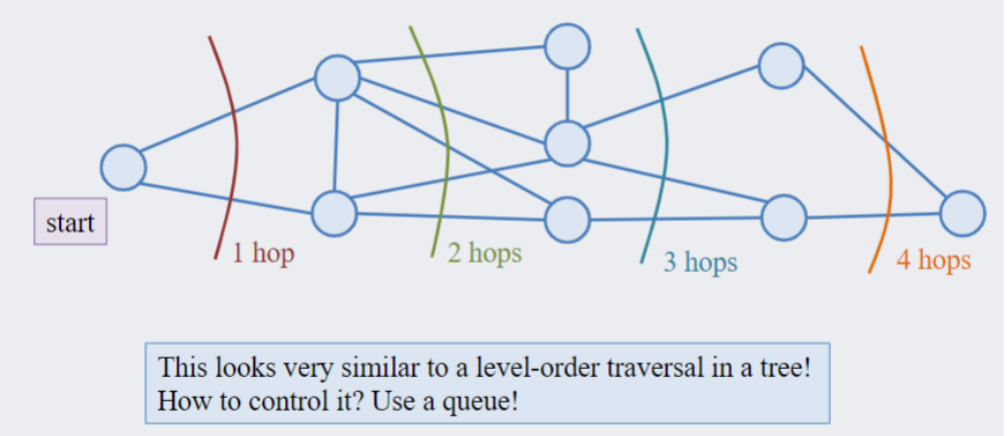
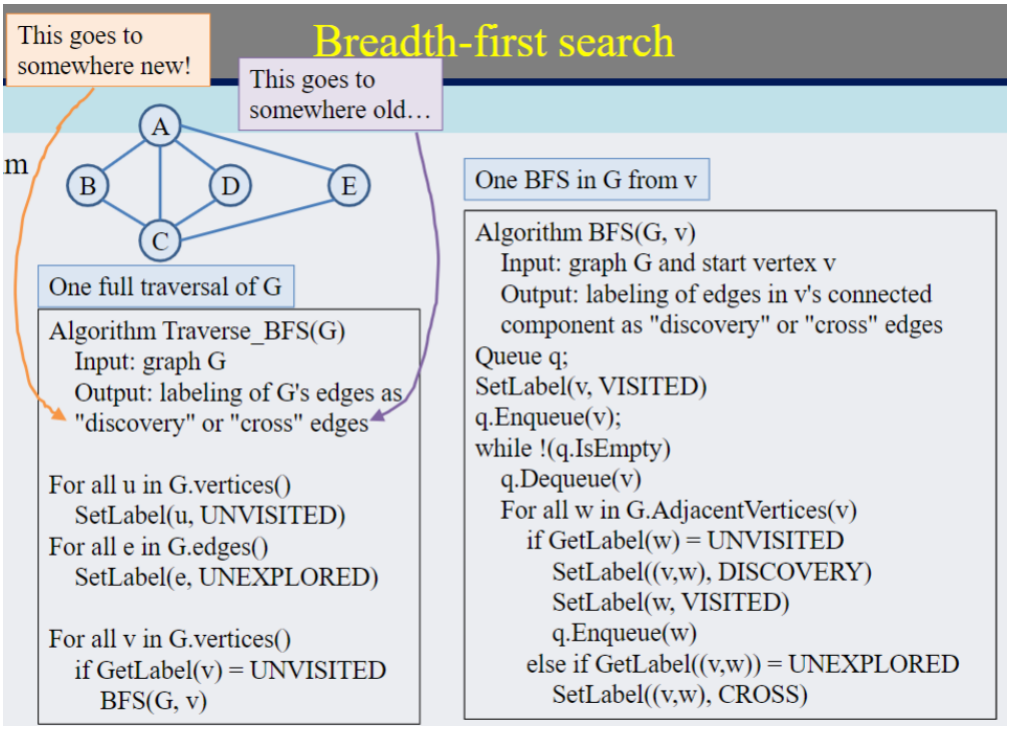
    - Space usage
        - vertex table: $\Theta(n)$
        - edge table: $\Theta(m)$
        - space for linked list of each vertex: $\Theta(m)$
        - in total: $\Theta(m+n)$
- Edge list
    - Just the edge table
    - Function analysis
        - `insertVertex`: might still need to store vertices explicitly
        - `removeVertex`:
        - `areAdjacent`:
        - `incidentEdges`: $\Theta(m)$
    - Costly in general
-

n vertices, m edges no parallel edges no self-loops bounds are big-Θ	Edge list	Adjacency list	Adjacency matrix
Space	$n+m$	$n+m$	n^2
`incidentEdges(v)`	m	$\deg(v)$	n
`areAdjacent(v, w)`	m	$\min(\deg(v), \deg(w))$	1
`insertVertex(o)`	1	1	n (amortized)
`insertEdge(v, w, o)`	1	1	1
`removeVertex(v)`	m	$\deg(v)$	n (amortized)
`removeEdge(e)`	1	1	1

Mar. 25th - Week 12.1: Breadth-first Search



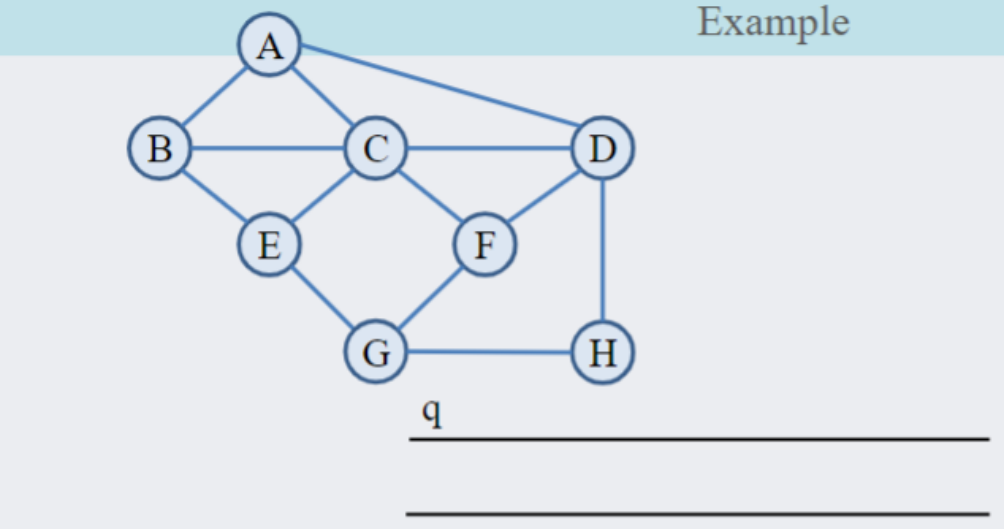
-
  - number of edges? Each vertex has degree 3
  - $\sum_{v \in V} \deg(v) = 28 \times 3 = 84$
  - $2 \times |E| = 84$, thus $|E| = 42$
- Graph traveral
  - Objective: visit **every** vertex and every edge in the graph, respecting the graph's structure
  - Purpose: we can search for interesting substructures in the graph, like solution paths in a game tree, shortest path to the enemy base, connected components
  - Contrast graph traversal to BST traversal, BST traversal is/has
    - Obvious start
    - meaningful L to R order
    - Directed (implicitly)
    - Acyclic
- BFS
  - Visits all vertices with d "hops" away from starting vertex, before visiting vertices within $d + 1$ hops, where d begins at 0

- 
start
1 hop
2 hops
3 hops
4 hops
This looks very similar to a level-order traversal in a tree!
How to control it? Use a queue!
- 
Breadth-first search
This goes to somewhere new!
This goes to somewhere old…
m
A
B
D
E
C
One full traversal of G
Algorithm Traverse_BFS(G)
Input: graph G
Output: labeling of G's edges as "discovery" or "cross" edges
For all u in G.vertices()
SetLabel(u, UNVISITED)
For all e in G.edges()
SetLabel(e, UNEXPLORED)
For all v in G.vertices()
if GetLabel(v) = UNVISITED
BFS(G, v)
One BFS in G from v
Algorithm BFS(G, v)
Input: graph G and start vertex v
Output: labeling of edges in v's connected component as "discovery" or "cross" edges
Queue q;
SetLabel(v, VISITED)
q.Enqueue(v);
while !(q.IsEmpty)
q.Dequeue(v)
For all w in G.AdjacentVertices(v)
if GetLabel(w) = UNVISITED
SetLabel((v,w), DISCOVERY)
SetLabel(w, VISITED)
q.Enqueue(w)
else if GetLabel((v,w)) = UNEXPLORED
SetLabel((v,w), CROSS)

- Example

"pred"	"dist"	v	?
		A	C B D
		B	A C E
		C	B A D E F
		D	A C F H
		E	B C G
		F	D C G
		G	H E F
		H	D G

- "pred": predecessor (where we come from); "dist": distance from start; "?": adjacency list



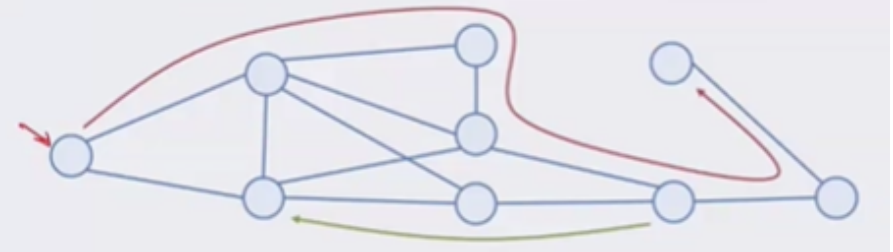
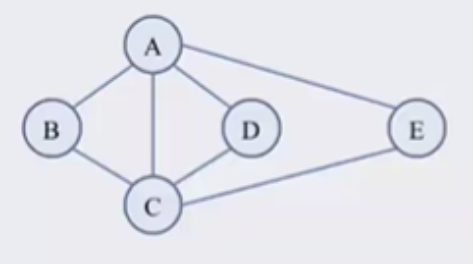
- "q": queue

- Process
  - Assume start with D, enqueue D into queue, mark D as "visited"
  - Queue is not empty, dequeue D, add all the unvisited neighbors: A C F H, and mark neighbors as visited; DA, DC, DF, DH are marked "discovery" edge

        - Queue is not empty, dequeue A, add all the unvisited neighbors: C F H B, mark B "visited"; AC marked "cross", AB marked "discovery"
        - Queue is not empty, dequeue C, add all the unvisited neighbors: F H B E, mark E "visited"; CB marked "cross", CF marked "cross", CE marked "discovery"
        - Queue is not empty, dequeue F, add all the unvisited neighbors: H B E G, mark G "visited"; FG marked "discovery".
        - Queue is not empty, dequeue H, add all the unvisited neighbors: B E G; HG marked "cross"
        - Queue is not empty, dequeue B, queue: E G; BE marked "cross"
        - Queue is not empty, dequeue E, queue: G; EG marked "cross"
        - Queue is not empty, dequeue G
        - Queue is empty
    - The discovery edges form: a spanning forest
    - The distances form: the number of steps away from the starting point (minimum number of steps)
    - One `BFS(G, v)`: visits one connected component
- Analysis
    - `while` loop (number of vertices)
        - $O(|V|)$
    - `for` loop (neighbors of v)
        - $O(\deg(v))$
    - Total running time
        - all connected components: $\sum_{v \in V} \deg(v) = 2|E| \in O(m)$
        - $\Theta(n)$ to initialize vertices + $\Theta(m)$ to initialize edges + all connected components (process edges) $\in \Theta(m + n)$

Mar. 26th - Week 12.2: Depth-First Search

- Depth-first search
    - Visits vertices along a single path as far as it can go, and then backtracks to the first junction and resumes down another path

- 
- Example
    - 

        - Start with D, mark as visited, add to stack: D.

          Check D's neighbors, mark DA discovery, mark A visited, add to stack: A D

          Mark AB discovery, mark B visited, add to stack: B A D

          Mark BC discovery, mark C visited, add to stack: C B A D

          Mark CA back, CD back, CE discovery, mark E visited, add to stack: E C B A D

          Mark EA back.

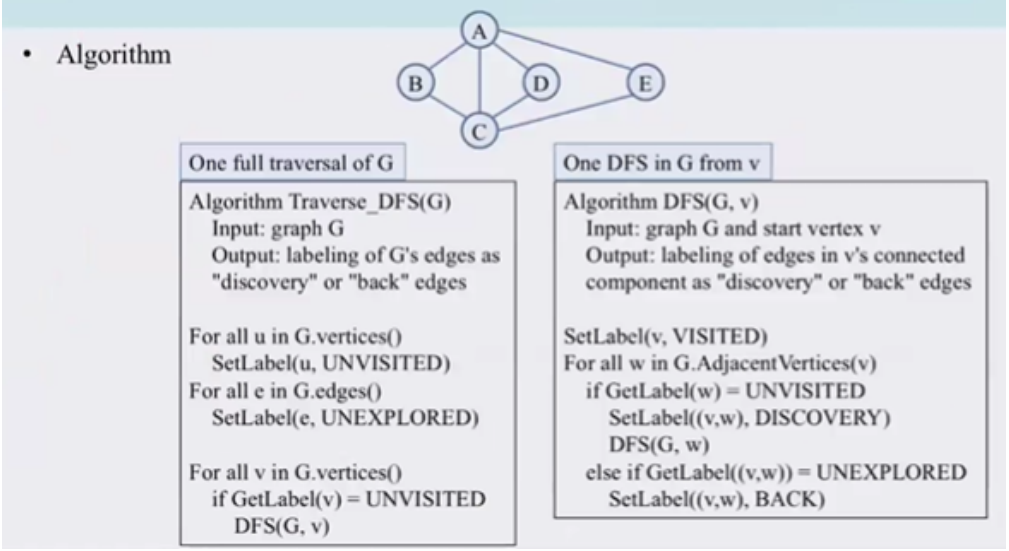
          For E, everything traversed, pop E

          For C everything traversed, pop C

          For B everything traversed, pop B

          For A, check AD, AE, everything traversed, pop A

          For D, check DC, everything traversed, pop C
        - The discovery edges form: tree
        - We can do pre- and post- order traversals
        - One `DFS(G, v)` visits one **connected component**
- Articulation point discovery, Bacon/Erdos numbers

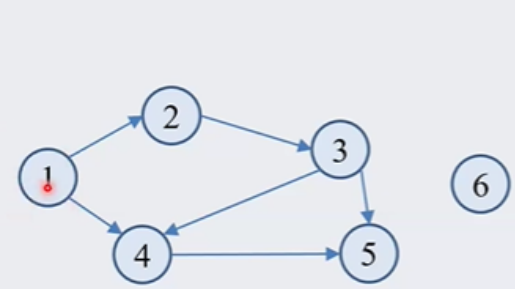
- 

- Algorithmic analysis
    - `for` loop (adjacent vertices)
        - Each neighbor is marked once for each vertex
        - Therefore, by handshaking theorem, the total number of iterations would be $2|E|$
        - $\in \Theta(m_c)$, c represents a component
    - `DFS(G, v)`
        - For the whole graph, it is adding every component's search, so it should still be $\Theta(m)$
        - The `for` loop for every vertex
        - Then we have $\Theta(m+n)$
    - The marking of each vertex and each edge $\in \Theta(m+n)$
    - The total traversal would be $\Theta(m+n)$
- Adjacency matrix: could be $\Theta(n^2)$

Apr 2nd - Week 13.1: Topological Sort, Spanning Trees

- Maze construction problem
    - A bunch of adjacent rooms
        - Each room is a vertex
        - Open wall between rooms form edge
        - Unpredictable, not easily solved
        - Highly branching, many dead ends, "feel random"
        - Just enough walls to get from any room to any other room (especially start and finish)
    - Given

        - Collection of rooms: $V, |V| \geq 1$
        - Connection between rooms (initially all "closed"), E
    - Construct a maze
        - Collection of rooms: $V' = V$
        - Designated rooms "in", $i \in V$, and "out", $o \in V$
        - Collection of connections to knock down: $E' \subset E$, such that **one unique path connects every pair of rooms**
    - So far, a number of walls have been knocked down, while others remain
    - Now we consider the wall between rooms A and B
        - If A and B are otherwise connected: Do not knock down
        - If A and B are otherwise not connected: Do knock down
    - Algorithm
        - While edges remain in E
            - Remove a random edge $e = (u, v)$ from E
            - If u and v have not been connected, add e to E', mark u and v as connected
        - How to check if two rooms are connected by some path?
- Spanning tree
    - A subset of edges from a connected graph that
        - touches all vertices in the graph
        - forms a tree (is connected and contains no cycles, **minimally connected**)
    - We already know two ways of constructing a spanning tree - BFS and DFS
        - Discovery edges from BFS/DFS spanning tree
        - produced from a traversal, so cost is the same as performing the traversal
    - In weighted graphs: minimum spanning tree
        - The spanning tree with the least total edge cost
- Topological Sort
    - Given a directed graph $G = (V, E)$, output all vertices in V such that no vertex is output before any other vertex with an edge to it (e.g. planning course schedule with prerequisites)
    - Label each vertex's **in-degree** (# of inbound edges)
    - Initialize a queue to contain all vertices with in-degree 0
    - While there are vertices remaining in the queue

    - Pick a vertex v from the queue and output it
    - Reduce the in-degree of all vertices adjacent to v
    - Put any of these with updated zero in-degree on the queue
    - Remove v from the queue
- Example
    - 

Vertex	In-degree
1	0
2	1
3	1
4	2
5	2
6	0

    - Queue: 1, 6

      Dequeue 1, go through neighbors 2, 4, decrement in-degree. Enqueue 2. Queue: 6, 2

      Dequeue 6, no neighbors. Queue: 2.

      Dequeue 2, go through neighbor 3, decrement in-degree. Enqueue 3. Queue: 3

      Dequeue 3, go through neighbors 4, 5, decrement in-degree. Enqueue 4. Queue: 4

      Dequeue 4, go through neighbor 5, decrement in-degree. Enqueue 5. Queue: 5

      Dequeue 5, no neighbors. Queue: [empty]

      Final sorting: 1, 6, 2, 3, 4, 5
- Is there always a topological sort?
    - Can only be performed on an acyclic directed graph
- How are algorithms affected (Adjacency list vs. Adjacency matrix)
    - Adjacency list

      Label each vertex v: $\deg^+(v)$; total graph: $\sum_{v \in V} \deg^+(v) \approx |E|$

      Initialize queue to contain all vertices with in-degree zero: $O(n)$

      While there are vertices remaining in the queue: $O(n)$

      Pick a vertex v from the queue and output it - $O(1)$

      Reduce the in-degree of all vertices adjacent to v - $\Theta(\deg(v))$

      Put any of these with updated 0 in-degree on the queue - $O(1)$

Remove v from the queue - $O(1)$

- Adjacency matrix

  Label each vertex v: $\Theta(n)$; total graph: $\Theta(n^2)$

  Initialize queue to contain all vertices with in-degree zero: $O(n)$

  While there are vertices remaining in the queue: $O(n)$

  Pick a vertex v from the queue and output it - $O(1)$

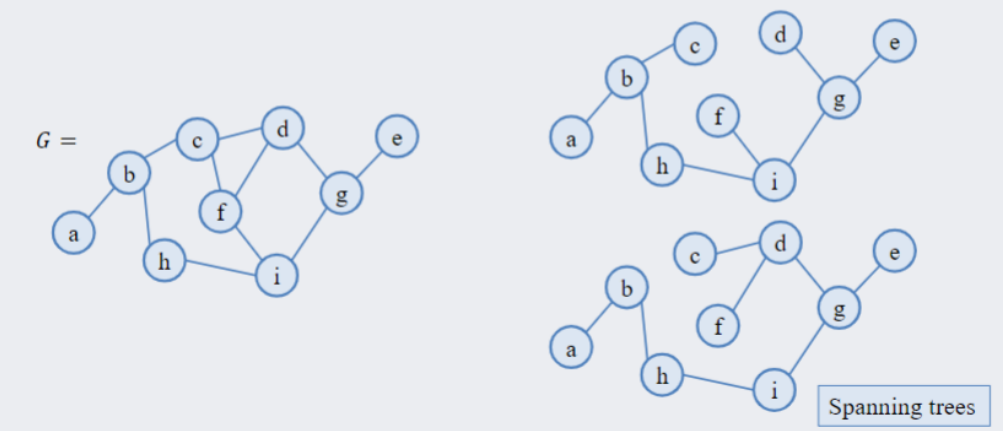
  Reduce the in-degree of all vertices adjacent to v - $\Theta(n)$

  Put any of these with updated 0 in-degree on the queue - $O(1)$

  Remove v from the queue - $O(1)$

- Therefore, Adjacency List - $O(m+n)$; Adjacency matrix - $O(n^2)$

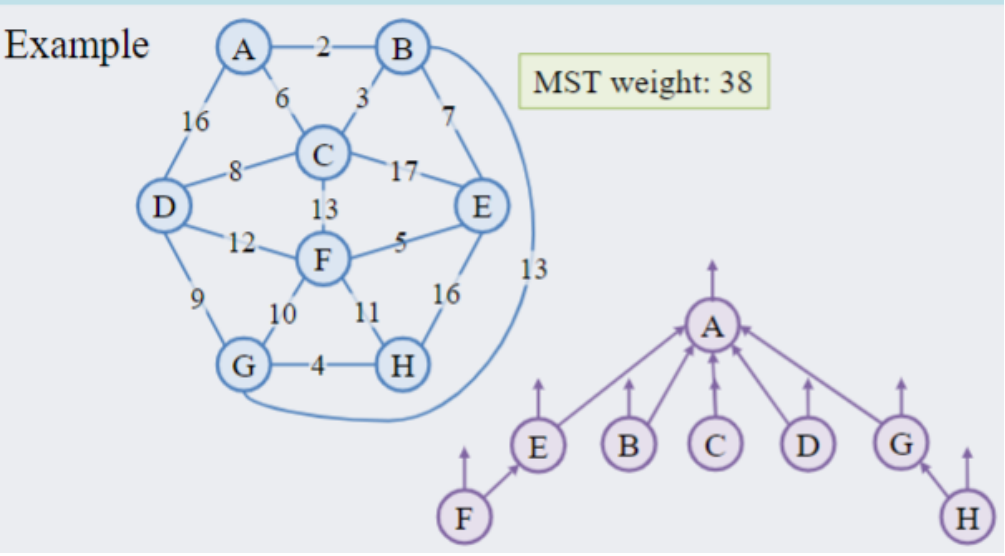
Apr 3rd - Week 13.2: Minimum Spanning Tree

- Spanning Trees
  - Given $G = (V, E)$, a spanning tree of G is a connected subgraph of G with exactly $|V| - 1$ edges
    - A minimal subset of edges that connects all the vertices of G
  - 

- Minimum spanning trees
  - Problem: installing power lines in Boolesville to supply all the residential/commercial/etc. districts, but it costs $5 per unit distance to construct the power line
  - Find a configuration of minimal cost that connects all the districts
  - This is a **minimal spanning tree**
  - Algorithmic requirements
    - Given a connected graph $G = (V, E)$ with unconstrained edge weights

        - Output a graph $G' = (V, E')$ with the following characteristics
            - G' is a spanning subgraph of G
            - G' is connected and acyclic
            - The sum of the edge weights of E' is minimal among all such spanning trees
- Kruskal's Algorithm (1956)
    - Builds a spanning tree from several **connected components**
    - Repeatedly chooses the minimum-weight joining two connected components, which does not form a cycle, until edge set has $|V| - 1$ edges
    -
```
KruskalsAlgorithm() {
    set E' = empty_set;
    while (|E'| != |V| - 1) {
        Find minimum weight edge e not in E' s.t. E'
union e does not contain cycles
        Add e to E'
    }
}
```

    - We need ADTs that support our required operations efficiently
        - How do we find the minimum weight edge? **Priority queue**
        - How do we prevent cycles and perform unions? **Disjoint set**
    - Example



prQ	
(A,B) 2	(F,G) 10
(B,C) 3	(F,H) 11
(G,H) 4	(D,F) 12
(E,F) 5	(B,G) 13
(A,C) 6	(C,F) 13
(B,E) 7	(A,D) 16
(C,D) 8	(E,H) 16
(D,G) 9	(C,E) 17

- Choose minimum weight edge (A,B), perform $A \cup B$

  Choose minimum weight edge (B,C), perform $B \cup C$, smart union

  Choose minimum weight edge (G,H), perform $G \cup H$

  Choose (E,F), perform $E \cup F$

  Choose (A,C), already union, remove (A,C) from PrQ

  Choose (B,E), perform $B \cup E$

  Choose (C,D), perform $C \cup D$

  Choose (D,G), perform $D \cup G$

  Choose (F,G), perform $F \cup G$

  All edges after this is already in the same set.

- Minimum spanning trees are not necessarily distinct as there can have different edges with same weight, and different trees with same total weight

- Running time analysis (Note that no insertions performed after build)

PRQ COST	HEAP	ORDERED ARRAY
Build	$O(m)$	$O(m \log m)$
RemoveMin	up to $m \times O(\log m)$	$m \times O(1)$

Overall Cost: MakeSet $\in O(n)$ + PrQ usage $\in O(m \log m)$ + Disjoint set usage $\in O(n \log^* n)$

- Maze under construction
    - Solve it using disjoint sets and random edge selection (like Kruskal's algorithm)
- Recall: BFS spanning tree
    -

B C D	A	(B,2), (C,6), (D,16)
E G	B	(A,2), (C,3), (E,7), (G,13)
F	C	(A,6), (B,3), (D,8), (E,17), (F,13)
	D	(A,16), (C,8), (F,12), (G,9)
H	E	(B,7), (C,17), (F,5), (H,16)
	F	(C,13), (D,12), (E,5), (G,10), (H,11)
	G	(D,9), (F,10), (H,4)
	H	(E,16), (F,11), (G,4)

    - What if we use a priority queue (with neighbors' edge weights) instead of an ordinary queue?
- Prim's algorithm (1957)
    - Based on Partition Property in graphs
    - Builds a spanning tree from initially one vertex
    - Repeatedly chooses the minimum-weight edge from a vertex in the tree, to a vertex outside the tree - adds that vertex to the tree
    -

```
PrimsAlgorithm(v)
{
  mark v as visited, add v to spanning tree
  while (graph has unvisited vertices)
  {
    Find least cost edge (w, u) from a visited vertex w to unvisited vertex u
    Mark u as visited
    Add vertex u and edge (w, u) to the minimum spanning tree
  }
}
```

    - Start from A

      Mark A visited, put $(A,B,2),(A,C,6),(A,D,16)$ into PrQ

      Use $(A,B,2)$, check if B visited. B unvisited, use (A,B), mark B visited, add (A,B) to E'. Put $(B,C,3),(B,E,7),(B,F,13)$ into PrQ

      Use $(B,C,3)$, check if C visited. C unvisited, use (B,C), mark C visited, add $(C,D,8),(C,E,17),(C,F,13)$ into PrQ

      Use $(A,C,6)$, check if C visited. C visited, take out of PrQ

      Use $(B,E,7)$, check if E visited. E unvisited, use (B,E), mark E visited, add $(E,F,5),(E,H,16)$ into PrQ

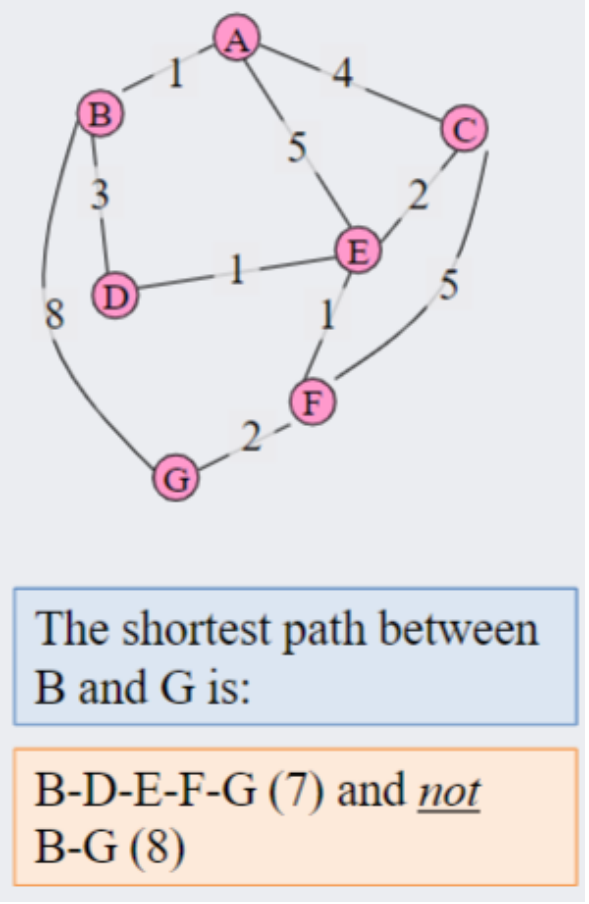
      Use $(E,F,5)$, check if F visited. F unvisited, use (E,F), mark F visited, add $(F,D,12),(F,G,10),(F,H,11)$ into PrQ

Use $(C, D, 8)$ ...

- Complexity
    - We will intersperse insertion and removal operations to the priority queue
    - Maximum number of insertions into the priority queue
        - Assuming heap implementation
        - $|E|$ in the worst case, then total cost of all insertions $\in O(m \log m)$
        - For dense graphs, $|E| \in O(|V|^2)$, then $\log |E| \in O(2 \log |V|) \in O(\log |V|)$
        - Thus the complexity of Prim's algorithm is $O(|E| \log |V|)$

Apr. 8th - Week 13.1: Shortest Path - Dijkstra's Algorithm

- Traversal variants
    - Traversal using stack / recursion: DFS
    - Traversal using queue: BFS
    - Traversal using priority queue (edge weights): Prim's algorithm for minimal spanning tree
    - Traversal using priority queue (path weights): ???
- Single-source shortest path
    - Given a graph $G = (V, E)$ and a vertex $s \in V$, find the shortest path from s to every vertex in V
    - Variations
        - weighted vs. unweighted (solve with BFS)
        - cyclic vs. acyclic
        - positive weights only vs. negative weights allowed
        - multiple weight types to optimize
    - Un/directed, weighted graphs, no negative cycles (guaranteed with no negative weight edges)
        - What is the least cost path from one vertex to another?
            - For weighted graphs, this is the path that has the smallest sum of its edge weights

- 

- Dijkstra's Algorithm
    - Classic algorithm for solving shortest path in weighted graphs **without negative weights** (if negative are present, the algorithm **might** produce the correct paths, but not guaranteed)
    - A greedy algorithm
        - Best local choice is made at each step, without considering future consequences
    - Intuition
        - Shortest pat from source vertex to itself is 0
        - Cost of going to adjacent nodes is at most edge weights
        - Cheapest of these must be shortest path to that node
        - Update paths for new node and continue picking shortest path

- Single-source shortest path

ource shortest path

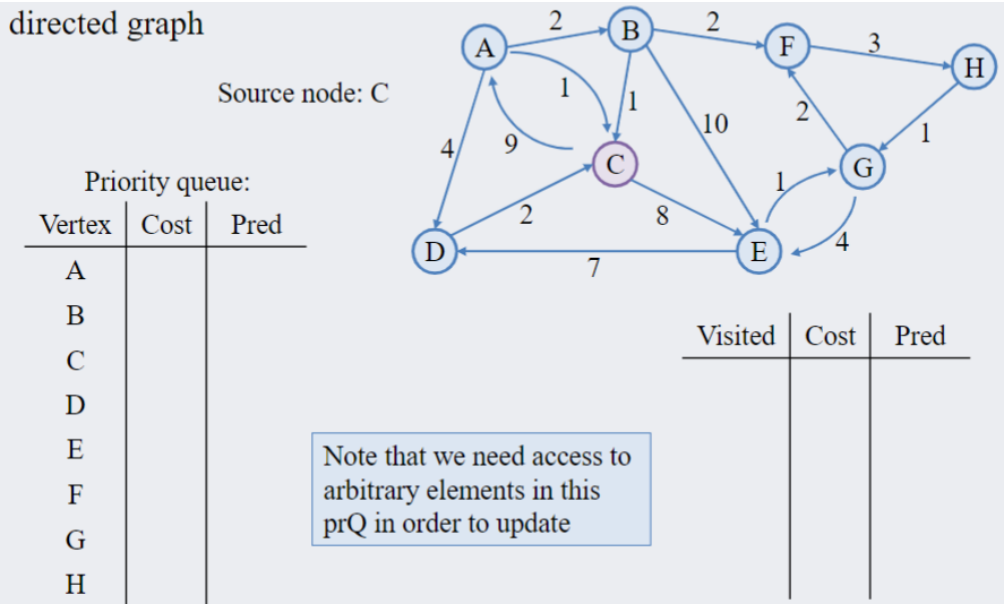
```
Dijkstra(G,s)
  d(s) = 0
  for all u ∈ V - {s}, d(u) = ∞
  R = ∅
  while R ≠ V
    pick u ∉ R with minimal d(u)
    R = R ∪ {u}
    for all vertices v adjacent to u
      if d(v) > d(u) + l(u,v)
        d(v) = d(u) + l(u,v)
```

Initialisation:
set distance to source = 0
set distance to other vertices to infinity
set of reached vertices = empty

While there are unreached vertices:
choose unvisited vertex u with minimal cost to reach
add u to collection of reached vertices

For all neighbours v of u
if path to v through u has lower cost than current known shortest path
update shortest cost path to v

- Example: Directed Graph



  - Source node: C

    Initialization: In PrQ, Cost(C) = 0, Cost(otherVertices) = ∞

    Dequeue `Vertex minCost()`, which is C, mark C visited, Pred(C) = $\varnothing$

    C can go to A, update Cost(A) = 9 + 0 = 9, Pred(A) = C

    C can go to E, update Cost(E) = 8 + 0 = 8, Pred(E) = C

    Dequeue E, mark E visited, Cost(E) = 8, Pred(E) = C

    E can go to D, update Cost(D) = 8 + 7 = 15, Pred(D) = E

    E can go to G, update Cost(G) = 8 + 1 = 9, Pred(G) = E

    Dequeue A, mark A visited, Cost(A) = 9, Pred(A) = C

    A can go to B, update Cost(B) = 9 + 2 = 11, Pred(B) = A

    A can go to C, C already visited

A can go to D, update Cost(D) = 9 + 4 = 13 < 15, Pred(D) = A

Dequeue G, mark G visited, Cost(G) = 9, Pred(G) = E

G can go to E, E already visited

G can go to F, update Cost(F) = 9 + 2 = 11, Pred(F) = G

Dequeue B, mark B visited, Cost(B) = 11, Pred(B) = A

B can go to C, C already visited

B can go to E, E already visited

B can go to F, Cost(F) = 11 + 2 = 13 > 11, do not update, Pred(F) = G (remains same)

Dequeue F, mark F visited, Cost(F) = 11, Pred(F) = G

F can go to H, update Cost(H) = 11 + 3 = 14, Pred(H) = F

Dequeue D, mark D visited, Cost(D) = 13, Pred(D) = A

D's neighbors all visited

Dequeue H, mark H visited, Cost(H) = 14, Pred(H) = F

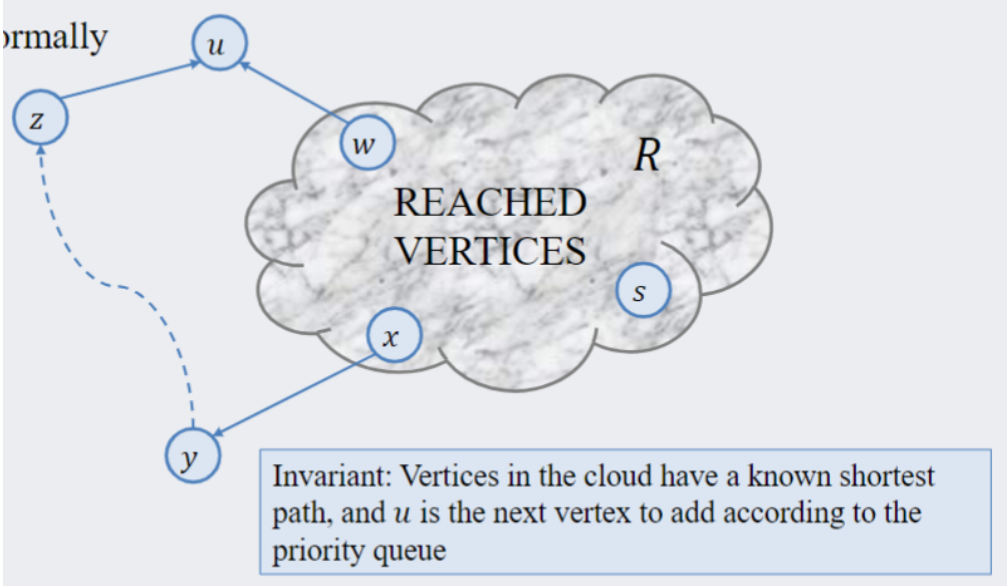
H's neighbors all visited

- The cost table is non-decreasing

- Data structure
  - Selecting unvisited node with the minimum cost
    - Priority queue / min-heap
    - Building initial heap is $O(|V|)$
    - `RemoveMin` executed $|V|$ times, total $O(|V| \log |V|)$
  - Each of $|E|$ edges has to be processed once
    - Looking up (and changing) the current cost of a vertex in a heap takes $O(|V|)$ for an unindexed heap ($O(1)$ if the heap is indexed) $\rightarrow$ use some kind of map
    - The heap property needs to be preserved after a change for an additional cost of $O(\log |V|) \rightarrow$ `heapifyUp()`
  - The total cost is

- BuildHeap: $|V|$ + RemoveMin: $|V| \log |V|$ + edges · (search + heapify): $|V| + \log |V|$
    - $O(|V| \log |V| + |E||V|)$
- If the heap is indexed the cost is $O((|V| + |E|) \log |V|)$

- Retrieving the shortest path
    - Once the results array is complete paths from the start vertex can be retrieved
    - Done by looking up the end vertex and backtracking through parent vertices to the start

- Correctness
    - Informally, cloud proof
    - 

    - Proof by induction, with **negative weights** wreck the proof
    - PrQ recommends u instead of y because $\text{Cost}(s \to w \to u) \leq \text{Cost}(s \to x \to y)$

      So, $\text{Cost}(s \to w \to u) < \text{Cost}(s \to x \to y) + \text{Cost}(y \to z) + \text{Cost}(z \to u)$