CPSC 221 NOTES

Resource came from: www.students.cs.ubc.ca/~cs-221/2018W2/lectures/

Analysis of Algorithm Efficiency

Why we care about analysis of algorithm efficiency?

- If there are different solutions/approaches to a single problem, we want to be able to decide which one to use!!
- A good algorithm should not only provide the correct solution but also finishes in a "reasonable" amount of time and a "reasonable" amount to space
- We care about *Time Complexity* AND *Space Complexity*

In CPSC 221, We use three different notations to analysis the Time and Space Complexities.

• Definition

Let f and g be real-valued functions defined on the same set of nonnegative real numbers. Then

1. f **is of order at least** g, written $f(x)$ **is** $\Omega(g(x))$, if, and only if, there exist a positive real number A and a nonnegative real number a such that

$$A|g(x)| \leq |f(x)| \quad \text{for all real numbers } x > a.$$

2. f **is of order at most** g, written $f(x)$ **is** $O(g(x))$, if, and only if, there exist a positive real number B and a nonnegative real number b such that

$$|f(x)| \leq B|g(x)| \quad \text{for all real numbers } x > b.$$

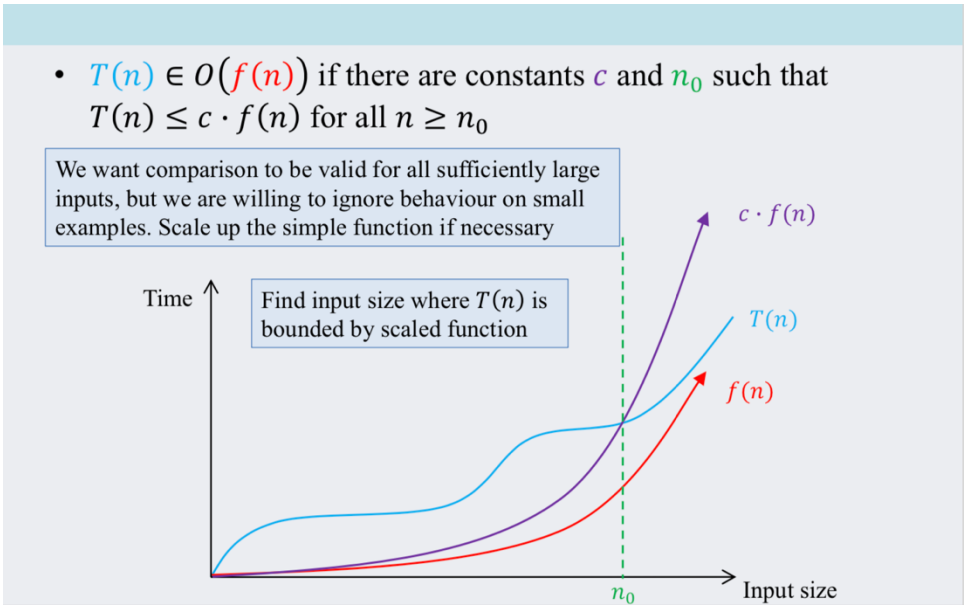
3. f **is of order** g, written $f(x)$ **is** $\Theta(g(x))$, if, and only if, there exist a positive real number A, B, and a nonnegative real number k such that

$$A|g(x)| \leq |f(x)| \leq B|g(x)| \quad \text{for all real numbers } x > k.$$

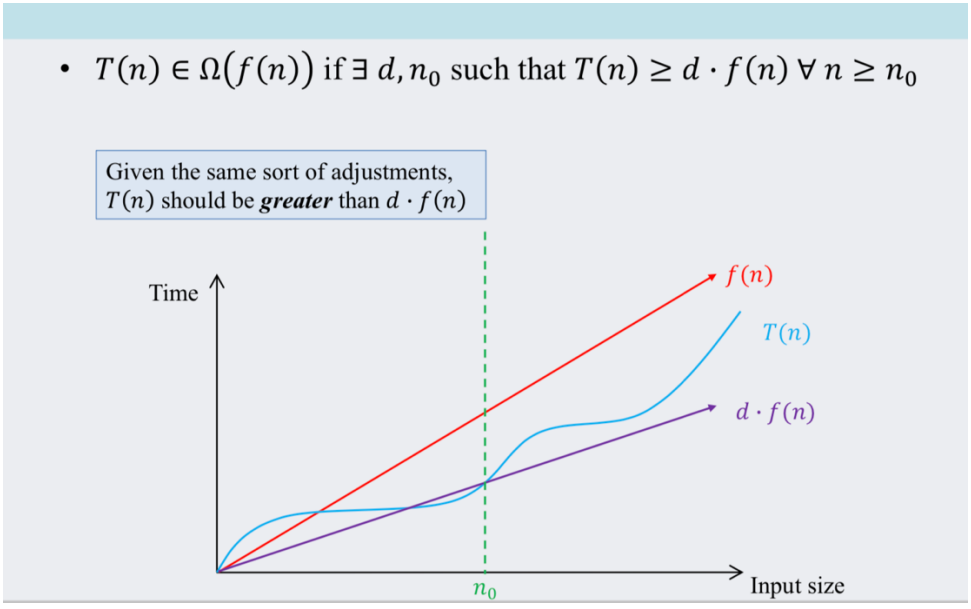
All three notation, big Omega, big O, big theta is used to analyze *Asymptotic bounds*

- When looking for Upper Bound – **Big O**
- Lower Bound – **Big Omega Ω**
- Tightest Bound – **Big Theta Θ [**if applicable!!!!]

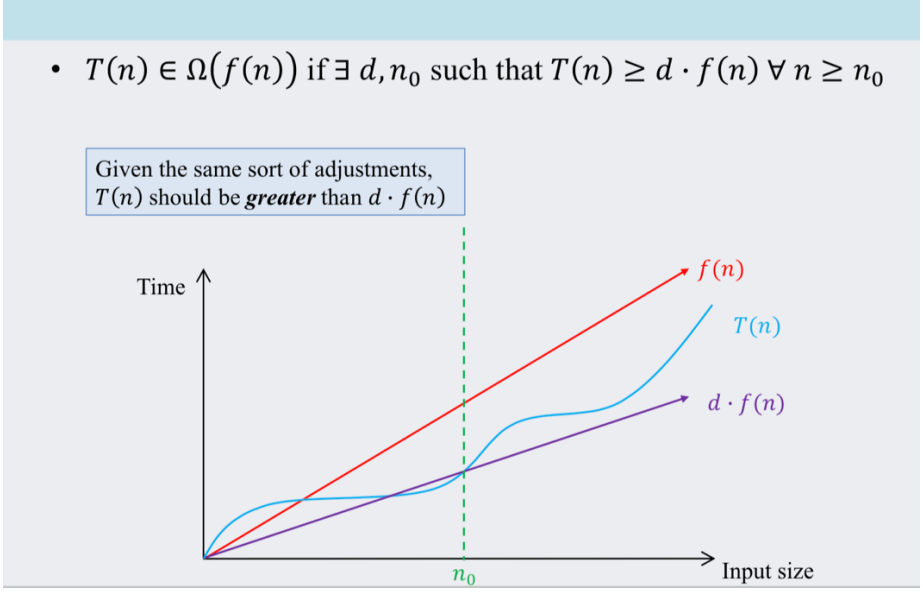
Big O – T(n) is bounded from above by c*f(n)



Big Omega Ω - T(n) is bounded from above by d*f(n)



Big Theta Θ – T(n) is bounded from above and below by f(n)



When writing asymptotic analysis, we

1. Eliminate low order terms

   4n + 5 - > 4n

   0.5nlogn-2n+7 - > 0.5nlogn

2. Eliminate constant coefficients

   4n - > n

   0.5nlogn - > nlogn

Examples:

$10{,}000n^2+25n \in \Theta(n^2)$

$10^{-10}n^2 \in \Theta(n^2)$

$nlogn \in O(n^2)$ => nlogn is at most n^2 in terms of growth rate

nlogn $\in \Omega$ (n) => nlogn is at least n^2 in terms of growth rate

nlogn $\in \Theta$(nlogn)

$n^3+4 \in O(n^4)$, but not $\in \Theta(n^4)$

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"

****Need to remember - IMPORTANT: constant, log, nlogn, n, n^k

Exercise1. Find the asymptotic upper bound for the following functions

```
1   int mystery(vector<int>& arr, int q){
2       for (int i = 0; i < arr.size(); i++){
3           if (arr[i] == q)
4               return i;
5           return -1;
6       }
7   }
8
9
10  bool hasDuplicate(int arr[], int size){
11      for (int i = 0; i < size-1; i++){
12          for (int j = i+1; j < size; j++){
13              if (arr[i] == arr[j])
14                  return true;
15          }
16      }
17  }
```

Ans: Θ (n), $\Theta(n^2)$

Key Points ©

◊ Determine the range of your loop variable

◊ Determine how many elements within that range will be "hit"

Write down examples to check your solution

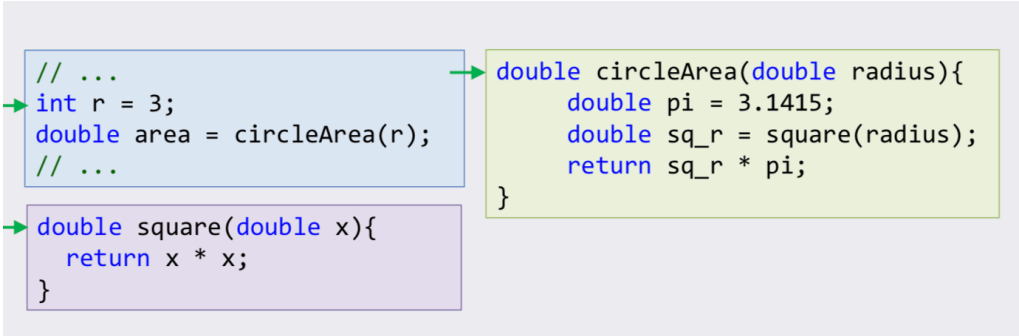
◊ Complexities of nested loops are (usually) multiplied

◊ Complexities of separate loops are (usually) added

Memories and Pointers

I. Function Parameters

- Unless written otherwise, parameters in C++ are passed by value, the *value* of the actual parameter is copied to the formal parameter
- The actual parameters and formal parameters are different variables in memory, even if they are named the same
- If you change the value of the formal parameter, this does **NOT** affect the values of the actual parameter back in the caller's memory
- Example:



- Call-by-value problems: If you want to modify the actual parameter you put in, there is no way for you to do it
- This can be fixed using call-by-reference, add a & symbol after the parameter's data type in the signature.

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

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

II. Dynamic Memory

- Variables declared in a function only exist within the scope of that function (or code block enclosed by { })
- The data structures we will learn about in this course require objects and variables to persist beyond a function's lifetime.
    - Cannot be allocated to call stack, need to put somewhere else – heap memory / dynamic memory
- We still need local variables that refer or point to the dynamically allocated memory
    - in C++ such variables are *pointers*
- A pointer is a special type of variable that stores an address rather than a value, the address is used to find a value elsewhere in memory

- Declaring pointers:

Declaring pointers

- Pointer variables are declared as follows:

```
datatype* identifier
```

  - e.g. `int* ptr;` or `int * ptr;` or `int *ptr;`
- Note that the type of a pointer is not the same as the type it points to
  - e.g. `ptr` is a pointer to an `int`, but is itself not an `int`
- Warning! The declaration

```
int* var1, var2;
```

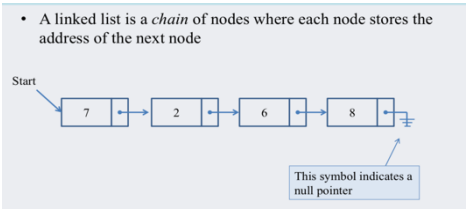
  - declares `var1` as a pointer, but `var2` as an integer!
- To declare both as pointers, either declare individually, or:

```
int *var1, *var2;
```

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

Basic Structure

I. Linked Lists



- A linked list is a dynamic data structure that consists of nodes linked together
- A node is a data structure that contains
    i. Data
    ii. The location of the next node

```
class Node {
  public:
    int data;
    Node* next;

    Node(int d, Node* nd) {
      data = d; next = nd;
    }
};
```

```
class LinkedList {
  private:
    Node* head;
    int length;

  public:
    LinkedList();
  ...
};
```

```
class LinkedList {
  private:
    Node* head;
    Node* tail;
    int length;

  public:
    LinkedList();
  ...
};
```

- Suppose we have a basic singly-linked list with a head pointer as defined above
    - Operations at the back of the list have relatively poor complexity requiring a traversal
    - But we can give ourselves a tail pointer for very little overhead, maintained during operations at the back of the list
- There are different kind of linked-lists: doubly linked-list, circular linked-list, circular doubly-linked-lists

II. Stacks ADT – a stack only allows items to be inserted and removed at one end (top)

- Access to other items in the stack is not allowed
- A stack can be used to naturally store data for postfix notation

Stack behaviour

- A stack ADT should support at least the first two of these operations:
  - **push** – insert an item at the top of the stack
  - **pop** – remove and return the top item
  - **peek** – return the top item
  - **is_empty** – does the stack contain any items

Must also have constructor(s) and destructor

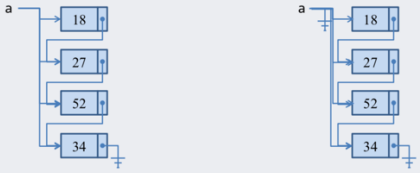
- ADT operations should be performed efficiently
  - The definition of efficiency varies from ADT to ADT
  - The order of the items in a stack is based solely on the order in which they arrive

- In this course, the stack ADT can be implemented using a linked list or an array
- Both implementations must implement all the stack operations in *constant time*

Stack implementation

Using a linked list

- Recall linked list construction from previous lessons
  - New nodes added at the "null" end of the list
  - Or inserted anywhere in the list
- Implement a linked-list stack by adding/removing from the front of the list



Stack implementation

Using arrays

- We need to keep track of the index that represents the top of the stack
  - When we insert an item increment this index
  - When we delete an item decrement this index
- Insertion or deletion time is independent of the number of items in the stack

Running time considerations

- Linked list implementation
  - push and pop simply call insert or remove from front of list
  - All operations $O(1)$
- Array implementation
  - push to full array requires $O(n)$ resize
  - resize by a constant **factor** (e.g. `capacity = 2 * capacity;`) leads to $O(1)$ average cost per operation
  - resize by a constant **amount** (e.g. capacity = capacity + 500;) leads to $O(n)$ average cost per operation

III. Queue ADT – In queue items are inserted at the back (end/tail) and removed from the front (head)

- Queues are FIFO data structures – *fair data structure*
- A queue implementation need to support: enqueue, dequeue, peek, isEmpty
- Linked List Implementation:

Queue implementation

Using a Linked List

- Removing items from the front of the queue is straightforward
- Items should be inserted at the back of the queue in constant time
  - So we must avoid traversing through the list
  - Use a second node pointer to keep track of the node at the back of the queue
    - Requires a little extra administration

Member attributes:

```
Node* front;
Node* back;
int num;
```

- Array Implementation

Circular arrays

- **Trick**: use a *circular array* to insert and remove items from a queue in constant time
- The idea of a circular array is that the end of the array "wraps around" to the start of the array

The modulo operator

- The `mod` operator (%) calculates remainders:
  - `1%5 = 1, 2%5 = 2, 5%5 = 0, 8%5 = 3`
- The `mod` operator can be used to calculate the front and back positions in a circular array
  - Thereby avoiding comparisons to the array size
  - The back of the queue is:
    - `(front + num) % arrlength`
    - where `num` is the number of items in the queue
  - After removing an item, the front of the queue is:
    - `(front + 1) % arrlength`

Member attributes:

```
int front;
int arrlength;
int* arr;
int num;
```

Sorting Algorithms

I. Selection Sort – a simple sorting algorithm that repeatedly finds the smallest item – we *select* the smallest item to put into place – the array is divided into sorted part and an unsorted part

```
21  void SelectionSort(vector<int>& arr){
22      for (int i = 0; i < arr.size() - 1; i++){
23          int smallest = i;
24          // Find the index of the smallest element
25          for (int j = i + 1; j < arr.size(); j++){
26              if (arr[j] < arr[smallest]){
27                  smallest = j;
28              }
29          }
30          //Swap the smallest with current item
31          temp = arr[i];
32          arr[i] = arr[smallest];
33          arr[smallest] = temp;
34      }
35  }
```

- Repeatedly swap the first unsorted item with the smallest unsorted item, start with the element with index 0, and ending with last but one element (index n - 1)
- Inner loop invariant: Before iteration of the inner loop, smallest contains the index of the smallest element of arr[i…j-1]. Aka, smallest contains the index of the smallest element seen so far
    - Base case: j = i+1, before the first iteration of the inner loop, the invariant states that smallest contains the index of the smallest element of arr[i…i], => only one must be the smallest seen so far.
    - Induction Hypothesis: at the end of loop k-1, loop invariance is maintained
    - Maintenance: j = k, i ≤ k < arr.size() ……
    - Termination: j = arr.size() ….

- Outer loop invariant: Before iteration of the outer loop, arr[0..i-1] contains the smallest element of the arr in ascending order. Also important to note [i…arr.size()-1] contains the arr.size() – i largest element of arr, unordered.
    - Base Case: i = 0. Before executing the iteration for i = 0, the invariant states that arr[0…-1] contains 0 smallest element of arr in ascending order. Since no element is in [0…-1], the base case invariant maintained.
    - Induction Hypothesis: at the end of loop k-1, loop invariance is maintained. Assume the invariant holds before processing index k, ie. arr[0…k-1] contains the k smallest element of arr in ascending order
    - Maintenance: i = k, $0 \leq k <$ arr.size() ….
    - Termination: i = arr.size() – 1 …..

II. Insertion Sort – find the correct place in the sorted part the replace 1st element of the unsorted part

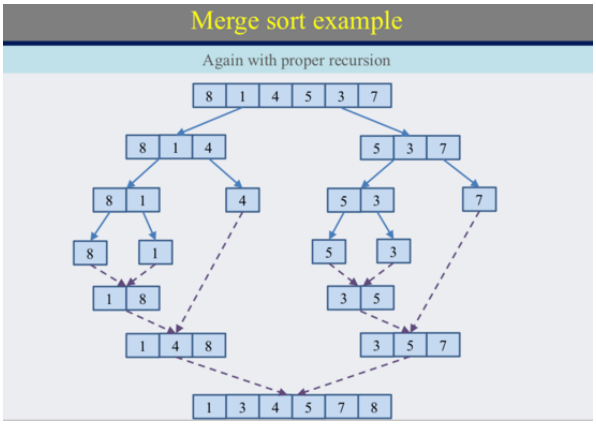
```
37  void InsertionSort(vector<int>& arr){
38      for (int i = 1; i < arr.size(); i++){
39          temp = arr[i];
40          int pos = i;
41          // Shuffle uo all sorted items > arr[i]
42          while(pos > 0 && arr[pos - 1] > temp){
43              arr[pos] = arr[pos - 1];
44              pos--;
45          }
46          // Insert the current item
47          arr[pos] = temp;
48      }
49  }
```

- Insertion sort is a good choice when data are nearly sorted, or when problem size is small

III. Merge Sort – an efficient sorting algorithm

```
53    void mergeSort(vector<T>& A, int L, int R){
54        if (R > L){
55            int M = (R + L) / 2;
56            mergeSort(A,L,M);
57            mergeSort(A, M+1, R);
58            merge(A,L,M+1,R-M);
59            }
60    }
```

- Divides an array in half, merges the two sorted halves
- Repeatedly divides array in half until each subarray contains a single element
- Note: the merge step copies the subarray halves into a temporary array and merged elements are copied from the temporary array back to the original array



Now with more math!

- Merge sort algorithm
  - Split list in half, sort first half, sort second half, merge together

$T(1) \leq b$

$T(n) \leq 2 \cdot T(n/2) + c \cdot n$

- Analysis

$$
\begin{aligned}
T(n) &\leq 2 \cdot T(n/2) + c \cdot n \\
&\leq 2\big(2 \cdot T(n/4) + c(n/2)\big) + cn \\
&= 4 \cdot T(n/4) + c \cdot n + c \cdot n \\
&\leq 4\big(2 \cdot T(n/8) + c(n/4)\big) + c \cdot n + c \cdot n \\
&= 8 \cdot T(n/8) + c \cdot n + c \cdot n + c \cdot n \\
&\leq 2^k \cdot T\big(n/2^k\big) + k \cdot c \cdot n \qquad \text{extrapolating, } 1 < k \leq n \\
&\leq n \cdot T(1) + c \cdot n \log n \qquad \text{for } 2^k = n, \text{ or } k = \log_2 n
\end{aligned}
$$

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

IV. Quick Sort

```
62  void quickSort(int arr[], int left, int right) {
63      int i = left, j = right;
64      int tmp;
65      int pivot = arr[(left + right) / 2];
66
67      /* partition */
68      while (i <= j) {
69          while (arr[i] < pivot)
70              i++;
71          while (arr[j] > pivot)
72              j--;
73          if (i <= j) {
74              tmp = arr[i];
75              arr[i] = arr[j];
76              arr[j] = tmp;
77              i++;
78              j--;
79          }
80      };
81      /* recursion */
82      if (left < j)
83          quickSort(arr, left, j);
84      if (i < right)
85          quickSort(arr, i, right);
86  }
```

- The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:
    - **Choose a pivot value.** We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
    - **Partition.** Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
    - **Sort both parts.** Apply quicksort algorithm recursively to the left and the right parts.

V. Running Time – We will explore heap later on in the course, ignore it for now

Name	*Best*	*Average*	*Worst*	*Memory*
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	$O(n^2)$	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)

NOTE: Mergesort and Quicksort are recursive and require additional space on the call stack. Heapsort can be implemented iteratively and requires no additional stack space except for local variables.

Proving Correctness of Code

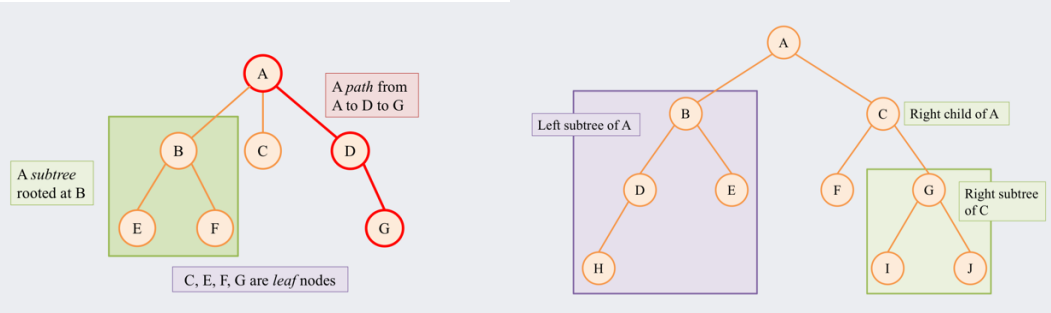
Induction variable:	Assume the invariant holds just before beginning some (unspecied) iteration.
Base case:	Prove the invariant holds at the end of that iteration for the next iteration.
Induction hypothesis:	Make sure the invariant implies correctness when the loop ends.
Inductive step:	Prove the invariant true before the loop starts.
Termination:	Number of times through the loop.

Proving correctness is only one benefit of loop invariants, they are also a natural way to think about your program!

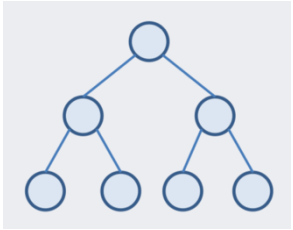
Trees

I. General Properties

- Trees are constructed from nodes – just like a linked-list structure
    i. Nodes may now have pointers to one or more other nodes
- A set of nodes with a single starting point
    i. Called the root of the tree (*root node*)
- Each node is connected by an edge to another node
- A tree is an *acyclic connected graph*, there is a path to every node in the tree and a tree has one less edge that the number of nodes
- A *leaf* is a node with no children
- A *path* is a sequence of nodes
- 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
- *Perfect Binary Tree* – Each node has either zero or two children, and all leaves are on the same level. A perfect binary tree of height h has $2^{h+1} - 1$ nodes, of which 2^h are leaves.
    - Perfect trees are also complete



- *Complete Binary Tree* – The leaves are on at most two different levels, the second to bottom level is completely filled in and leaves on the bottom level are push to the left

II. Tree-Traversals – algorithms for binary tress visits each node in the tree, typically do something while visiting each node!

- There are four different type of traversals
    i. Pre-Order Traversal – can be used in copy constructors / assignment operator
    ii. In-Order Traversal

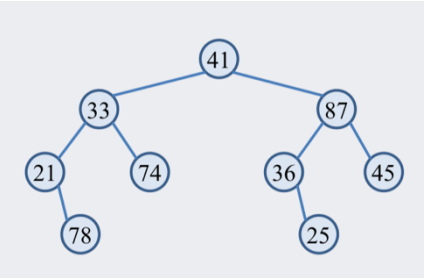
```
void inOrder(Node* nd)
{
  if (nd != nullptr)
  {
    inOrder(nd->leftchild);
    visit(nd);
    inOrder(nd->rightchild);
  }
}
```

```
88  void tree<T>::levelOrder(Node * croot){
89      Queue<Node*> q;
90      if(croot != NULL){
91          q.enqueue(croot);
92          while(!q.isEmpty()){
93              Node* t = q.dequeue;
94              //level order operations
95              cout<< t->data;
96              if(t->left != NULL)
97                  q.enqueue(t->left);
98              if (t->right != NULL);
99                  q.enqueue(t->right);
100         }
101     }
102 }
```

    iii. Post-Order Traversal – can be used to calculate height of each node in the tree, also can be use in the clear function
    iv. Level-Order Traversal – can be implemented using a queue structure, add child of each node when you remove the node from the queue

- Exercise2. Construct the Original Tree from the given traversals, also note down the post-order and level-order traversals after the tree has been reconstructed:
    i. Pre-Order: 41 33 21 78 74 87 36 25 45
    ii. In-Order: 21 78 33 74 41 36 25 87 45

  Ans:



  Post (78 21 74 33 25 36 45 87 41), level (41 33 87 21 74 36 45 78 25)

- **Key Point © - ALL tree traversals have running time O(n)**

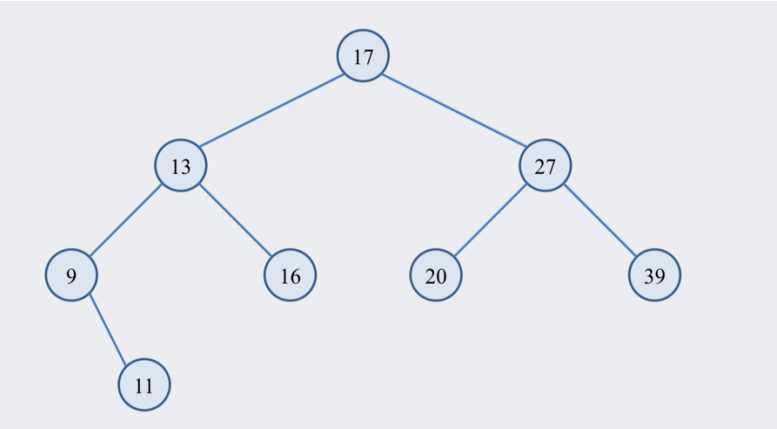
Dictionary ADT

I. General Properties

- A dictionary ADT stores *values* associated with user-specified keys
- Dictionary operations: create, destroy, insert, find, remove

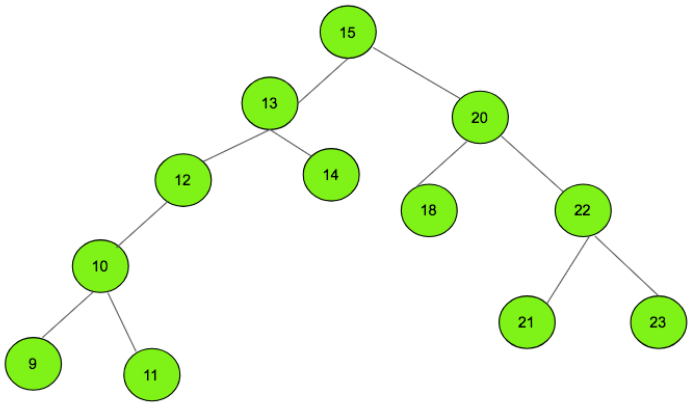
II. Binary Search Tree

- A *binary search tree* is a binary tree with a special property: For all nodes in the tree, 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



- In-Order traversal on a BST retrieves data in sorted order
- From the root, keep following left child links until no more left child exist, then you can find the minimum node
- From the root, follow right child links until no more right child exists, then you find the maximum node
- To find a value in a BST from the root node:
    - If the target is less than the value in the node, search its left subtree
    - If the target is greater than the value in the node, search its right subtree
    - Otherwise return true, ….or….

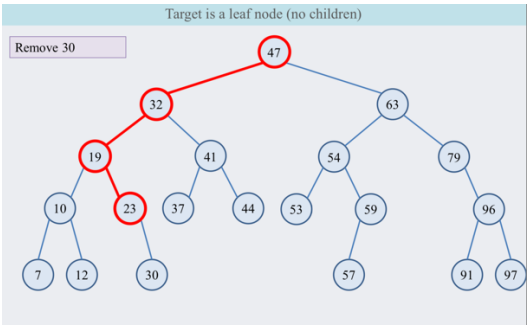
- Worst case running time for binary search is height of the tree+1 => O(h)
- Exercise: Trace down the paths performed by a binary search to find 11 in the above diagram
- BST Insertions:
    - BST property must hold after insertion, therefore the new node must be inserted in correct position
        - This position is found by performing a search
        - If the search end at the (null) left child of a node, make its left child refer to the new node
        - If the search ends as the right of a node make its right child refer to the new node
    - The cost is about the same as the cost of the search algorithm O(height)
    - Try this:



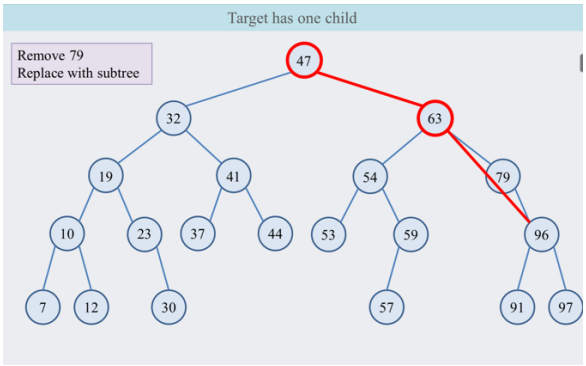
Where is the correct position to insert 16, 24, and 12?

- BST Removal:
    - Terminology:
        - Predecessor – right most node of its left subtree
        - Successor – left most node of it right subtree

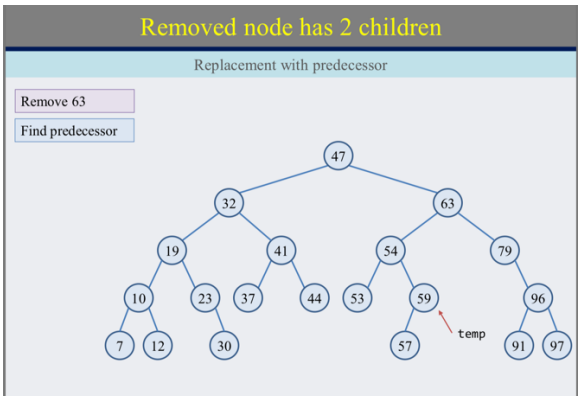
- When removes a node with no children, we delete it right away and set its' parents' reference to null



- When removes a node with one child, we replace the node with its subtree



- When removes a node has two children, we replace its value by its' predecessor's value, and then delete its's predecessor – it is okay to choose successor, just make sure to be consistent!



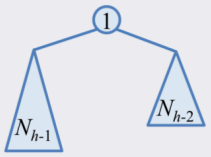
- The efficiency of BST operations depends on the *height* of the tree – all three operations (search, insert, and delete) are O (height)
- CODE for Binary Search Tree Operations:

```
Node* BST::Find(Node* &croot, const K & key){
   if (croot == NULL || croot->key == key)
      return croot;
   else if (key<croot->key)
      return find(croot->left, key);
   else
      return find(croot->right, key);
}

void BST::insert(Node* &croot, const K & key){
   if(croot == NULL)
      croot = new Node(key);
   else if (key < croot.key)
      insert(croot->left);
   else if (key >= croot.key)
      insert(croot->right);
}

void BST::remove(Node*& subtree, const K& key){
   if (!subtree)
      return;
   if (key < subtree->key) {
      remove(subtree->left, key);
   } else if (key > subtree->key) {
      remove(subtree->right, key);
   } else {
      /* Reached the node that we need to delete */
      if (subtree->left == NULL && subtree->right == NULL) {
         /* Case 1: Node to remove has no children */
         delete subtree;
         subtree = NULL;
         return;
      } else if (subtree->left != NULL && subtree->right != NULL) {
         /* Case 2: Node to remove has two children */
         // find the predecessor
         Node* pred = subtree->left;
         while (pred->right)
            pred = pred->right;
         swap(subtree, pred);
         remove(subtree->left, key);
      } else {
         /* Case 3: Node to remove has one child */
         Node* curr = subtree;
         subtree = max(subtree->left, subtree->right);
         delete curr;
      }
   }
}
```

III. AVL Tree – Balanced Binary Search Tree

- A binary tree is balanced if leaves are all about same distance from the root
    i. Height balance of each tree T is b = height(T_R) – height(T_L)
    ii. A tree T is balanced if T is an empty tree, or | b | ≤ 1
- Why do we want AVL Tree?
    - Key Idea: lease possible height (h) for a tree of n nodes is $\log_2 n$, if we keep the height ~ = $\log_2 n$, then the operations on BST will have an upper bound of O(logn)
    - !!! AVL operations (find, insert, remove) have running time O(logn)
    - But, how do we prove AVL trees have height O(logn)

- Proof:
  - Let N_h represent the minimum number of nodes in an AVL tree of height h
  - Since the AVL property must be satisfied at every node, the children of such a tree must also be minimal, and the height difference between the children must be 1
  - Thus $N_h = 1 + N_{h-1} + N_{h-2}$



$N_h = 1 + N_{h-1} + N_{h-2}$ $\quad N_{h-1} = 1 + N_{h-2} + N_{h-3}$

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

$= 2 + 2N_{h-2} + N_{h-3} > 2N_{h-2}$ $\quad N_{h-2} = 1 + N_{h-3} + N_{h-4}$

$= 2 + 2N_{h-4} + N_{h-5} > 2N_{h-4}$

$N_h > 2 \cdot 2N_{h-4}$

$N_h > 2 \cdot 2 \cdot 2N_{h-6}$

$N_h > 2 \cdot 2 \cdot 2 \cdot 2N_{h-8}$

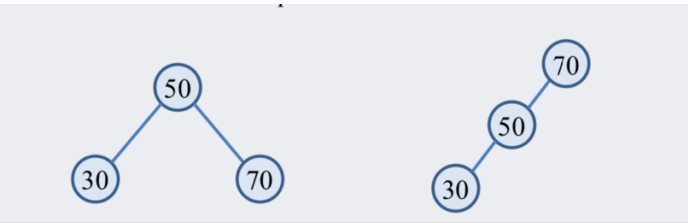
...

$N_h > 2^{h/2}$

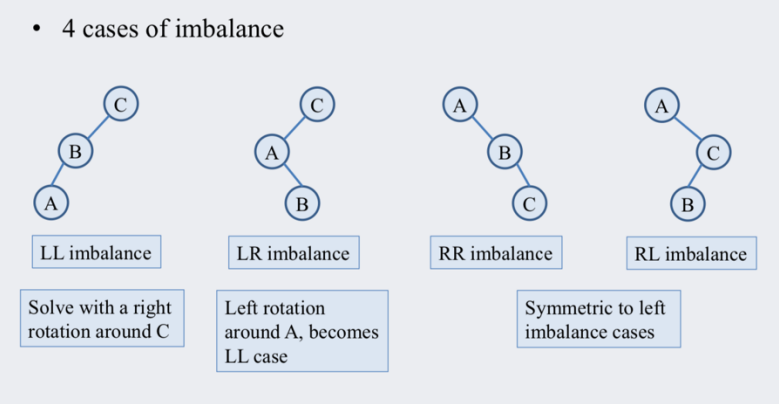
$h < \log N_h$ $\quad h \in O(\log n)$

How many times can we subtract 2 from h before we reach 0?
$\frac{h}{2}$ times.

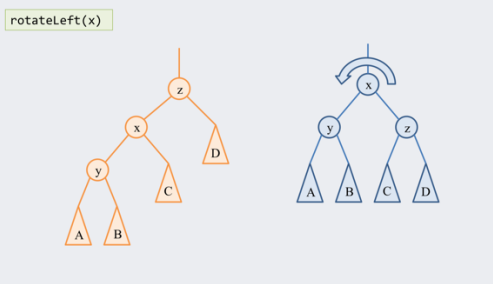
- Question: How do we maintain AVL property during those operations?
- A tree's shape can be altered by rotation while still preserving BST property

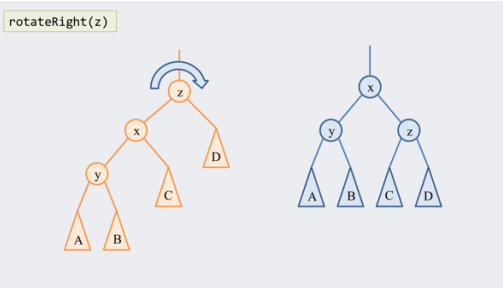


- We use rotations to maintain AVL Tree property during operations – AVL operations begins with ordinary BST insertions followed by rotations to maintain balance.

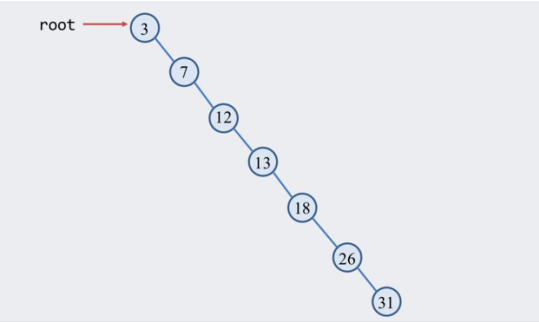


There are four types of Rotations: Right rotation, Left-right rotation, Left rotation, Right-Left rotations to solve those 4 cases of imbalance respectively.



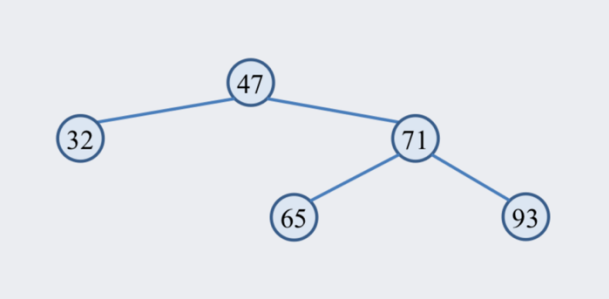


- Exercise: construct an AVL tree using rotations from the given tree:



- Exercise: perform following operation on the given AVL Tree

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

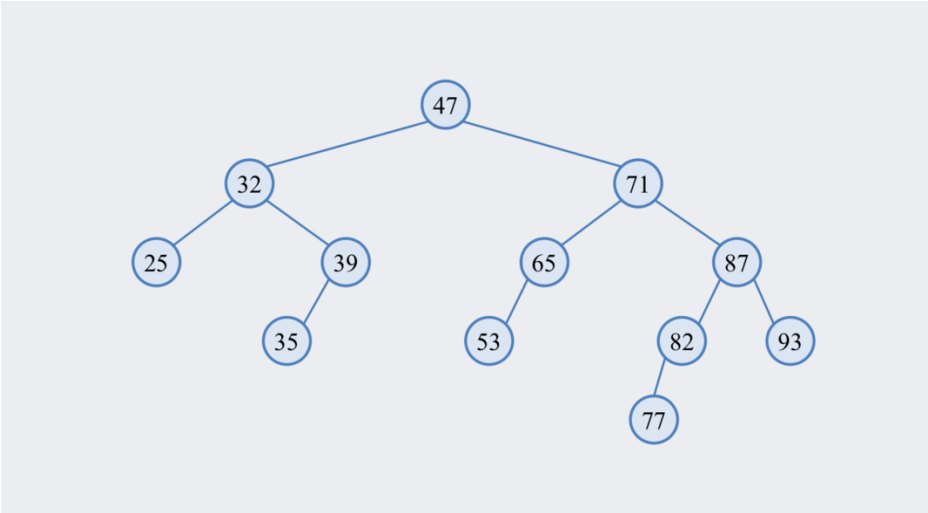


- Exercise: Starting with an empty AVL Tree, insert the keys in the following order:

  1,3,5,7,9,8,6,4,2

- Removal: Same with BST but just remember to rebalance it at the end!

  What happens if we remove 65, 47, 25 in order?



IV. Hash Tables

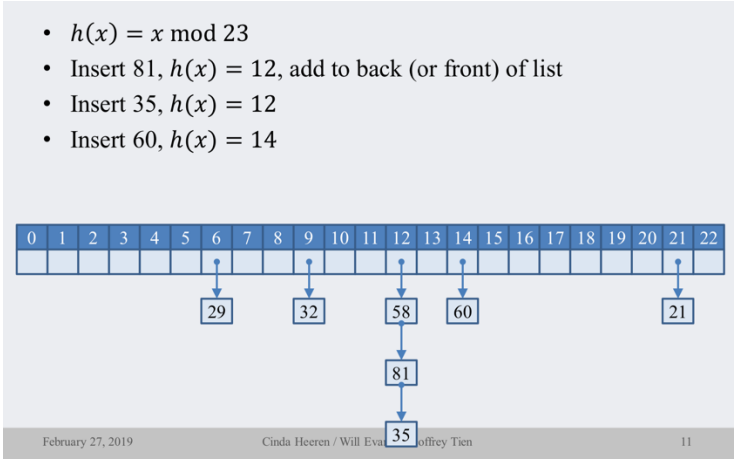
- A Hash Table consists of an array to store data, data often consists of complex types, or pointers to such objects. One attribute of the object is designated as the table's key
- KEY -> VALUE
- A *hash function* maps a key to an array index in 2 steps
    - The key should be converted to an integer
    - And that integer mapped to an index using some function.

Usually: (int mod *table-size*)

- What properties does a good hash function need to have? – IMPORTANT
    i. Fast - Integer value can be from h(k) can be computed in O(1) time
    ii. Deterministic – For all k1,k2, if k1=k2, then h(k1) = h(k2)
    iii. SUHA (Simple Uniform Hashing Assumption) – for all k, P(h(k)) ≤ 1/m

    Where m implies the table size.

- A typical hash function usually results in some collisions, where two different search keys map to the same index, the goal is to *reduce* the number and effect of collisions
- Collision handling – Separate Chaining



- Each entry in 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 as the appropriate location
- Running Time:

Running Time	**Insert**	**Remove/Find**
	O(1)	O(list length)

- Collision handling – Linear Probing
  - Idea: 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. For linear probing, we look for the next index and check whether if it is empty
  - Problem: Linear probing leads to *primary clustering* – the table contains groups of consecutively occupied locations, these clusters tend to get larger as time goes on, which reduces the efficiency of the hash table
  - Eg: h(x) = x mod 23

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 81, 35, 60, 12

  - Searching for an item is similar to insertion. Use linear probing to find a number or an empty space. If find, return index, otherwise conclude that the number is not in the table.

For the example above, find 32, 59

- Collision handling – Double Hashing
  - Key dependent probe sequence
  - The second hash function must follow these guidelines:
    1) $h_2(key) != 0$
    2) $h_2 != h_1$
    3) A typical h_2 is p – (key mod p) where p is a prime number
  - Eg. $h_2 = 5 – (key \bmod 5)$

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 81, 35, 60, 83

- Removals and open addressing: mark table locations as either empty, occupied, or removed
- Hash Table efficiency, when analyzing the efficiency of hashing it is necessary to consider *load factor* λ. λ = *number of items / table size*
- As table fills, load factor increase, and the chance of a collision occurring increases, it is important to base the table size on the largest possible number of items, table size should be selected so that λ does not exceed ½.
- KEY POINT!! If load factor is less than 1/2 , probing and separate chaining give the similary performance. As λ increase, separate chaining performs better than open addressing

V. B-Tree – ☺

Properties:

- All keys within a node are ordered.
- All leaf nodes contain no more than m-1 nodes.
- All internal nodes between $\lceil \boldsymbol{m}/\mathbf{2} \rceil$ and m children.
- Keys and values are only stored in the leaf nodes. Search keys in internal nodes only direct traffic.
- Root nodes can be a leaf or have between 2, and m children.
- All leaves are on the same level.

B Tree Insertion

1) Insert (key, data) pair in the target leaf page

2) If the leaf overflows (>= m):

a. Split the leaf into two leaves:

- i. Original (left) holds first half (rounded up)
- ii. New node holds second half (rounded down)

b. Copy the largest key in original leaf up to parent

Maximum # of Node in an order-m B-tree

$$\frac{m^h - 1}{m - 1}$$

Running Time: For each node, we must perform a linear search through m-1 keys, we need to search O(height) nodes. B-Tree Operations (Search, Insert): O(m*height) = O(mlogn)

Priority Queues ADT

Operations: Create, Destroy, Insert, RemoveMin, isEmpty

Property:

- For two element x and y in the queue, if x has a higher priory value than y, x will be removed before y
- A single PQ will only support RemoveMin OR RemoveMax but NOT both
- Two or more distinct item may have the same priority

I. Data structures for priority queues:

STRUCTURE	INSERT	REMOVEMIN/MAX
UNORDERED ARRAY	O(1)	O(n)
ORDERED ARRAY	O(n)	O(n)
UNORDERED LIST	O(1)	O(n)
ORDERED LIST	O(n)	O(1)
BST	O(n)	O(n)
AVL TREE	O(logn)	O(logn)
BINARY HEAP	O(logn)	O(logn)

II. Binary Heap

A *heap* is binary tree with two 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 as large as its children's values

Heap Implementations:

- 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
    - Because heaps are *complete* binary trees there can be no gaps in the array
- NOTE: There are two common implementations
    - Insertion starting at index 0 in the array
    - Insertion starting at index 1 in the array
- Reference children/parent: (index 1 method)
    - For node i, children indexed at $2i$ and $2i + 1$
    - For node i, parent indexed at $i/2$

Heap Insertion

- On insertion the heap properties have to be maintained
- The insertion algorithm first ensures that the tree is complete
    - $O(1)$ access using array index
- Fix the partial ordering
    - Repeated heapify-up operations

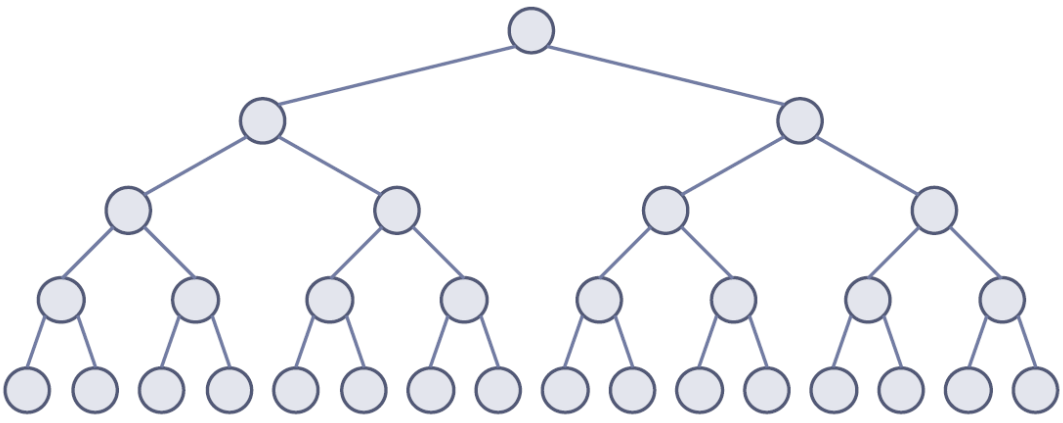
- Heap insertion has worst-case performance of O($\log n$)

Heap Removal

- First ensure that the heap remains complete
    - Replace root with last element O(1)
- Swap the new root with its *largest valued child* until the partially ordered property holds
    - Repeated heapify-down operations starting from root level
- Removing the priority item from a heap is also O($\log n$) in the worst case

Building a Heap

- First put all of the elements into an array – don't need to sort it or order it in any way
- To create a heap from an unordered array repeatedly call heapify-down
    - Call heapify-down on elements in the upper $\frac{1}{2}$ of the array



- Heapify-down is called on half the array
- It would appear that buildHeap cost is O(n log n) exactly $n - 1$ edges, thus the worst case number of swaps is $O(n)$

Heap Sort

- Heapify the array (max heap)
- Repeatedly remove the root (largest item)
- At the end of the sort the array will be sorted in ascending order
- Complexity: $O(n)$ + O(n log n) = O(n log n)

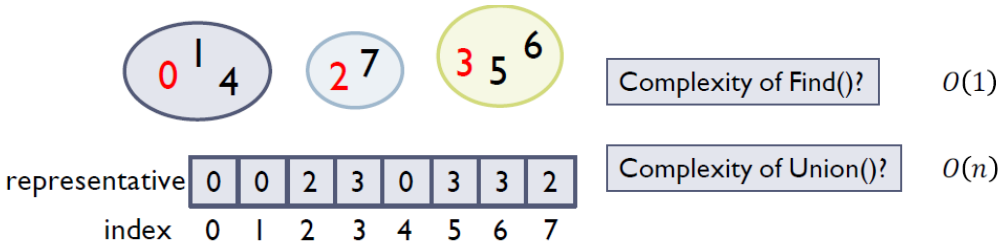
Algorithm	Best	Average	Worst	Space
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

Disjoint Set

Disjoint Set is an ADT that supports the following functions

- int Find(int x)
    - given person x, returns the leader of x's party
- void Union(int x, int y)
    - given people x and y, unites their parties under a single supreme leader

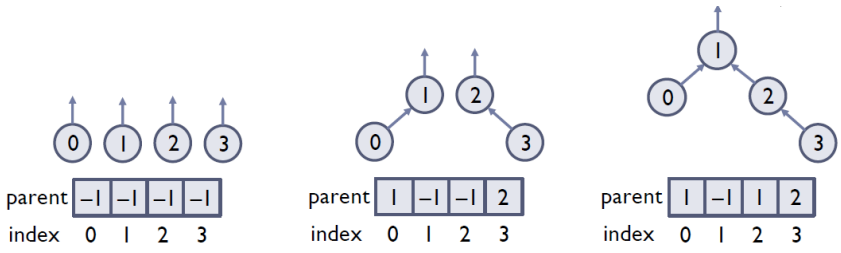
An array-based structure



A better structure for disjoint sets - “Uptree”

- A tree where a node points to its parent
- still array-based, but representative is the root of the tree

x and y are in the same tree $\Leftrightarrow x$ and y are in the same set

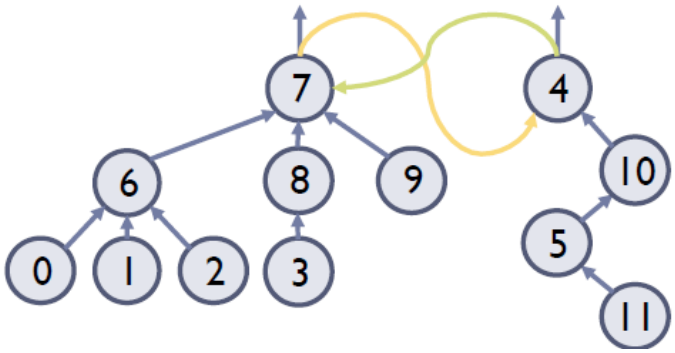


Tree-based disjoint set depends on the height of the trees in the disjoint sets

- average: $O(\log n)$, worst: $O(n)$, best: $O(1)$

"Smart" union

- both schemes guarantee that the height of the tree is $O(\log n)$



Union by height												
	parent	6	6	6	8		10	7	4	7	7	4	5
	index	0	1	2	3	4	5	6	7	8	9	10	11

Union by size												
parent	6	6	6	8	7	10	7		7	7	4	5
index	0	1	2	3	4	5	6	7	8	9	10	11

Path Compression :

- During a Find operation, we follow a path up the tree through a sequence of nodes
- Set the parent of each node along the path, to the root found at the end of the path
- Nearly O(1) Running time for find – if both path compression and smart union are used!

Graph

A graph consists of two sets, V and E

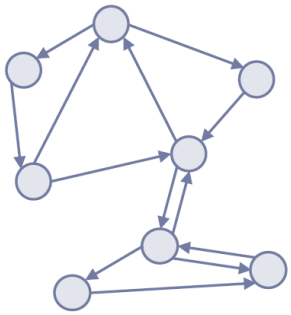
Two vertices may be connected by a *path*

If a graph has v vertices, how many edges does it have?

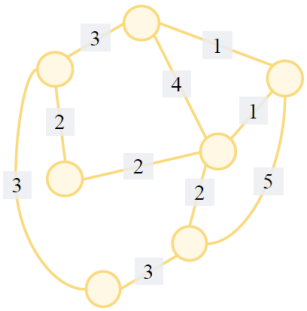
- Max $v^2 - v$ (connected)
- Min $v - 1$(connected)

Terminology:

1. Directed Graphs: In a directed graph (or *digraph*) each edge has a direction and is called a directed edge, directed edge can only be traveled in one direction.



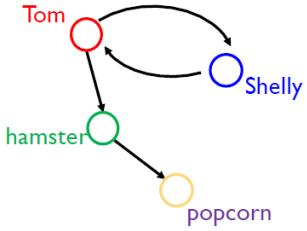
2. Weighted Graphs: In a weighted graph each edge is assigned a weight, Each edge's weight represents the cost to travel along that edge.



Graph Implementations:

1. Adjacency matrix

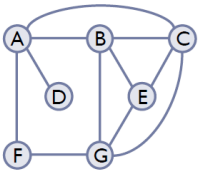
   - A $V \times V$ array in which an element u, v is true if and only if there is an edge from u to v



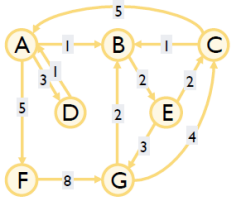
	Tom	Shelly	hamster	popcorn
Tom				
Shelly				
hamster				
popcorn				

2. Adjacency List

   - A V-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices



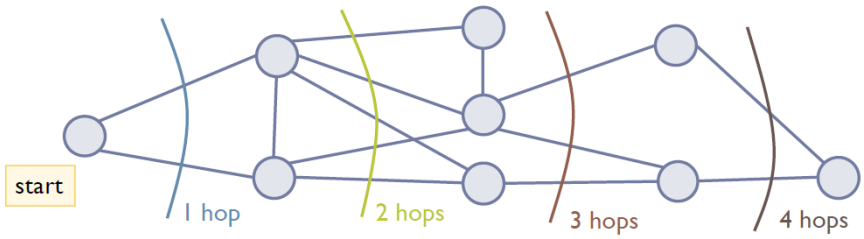
A	B → C → D → F
B	A → C → E → G
C	A → B → E → G
D	A
E	B → C → G
F	A → G
G	B → C → E → F



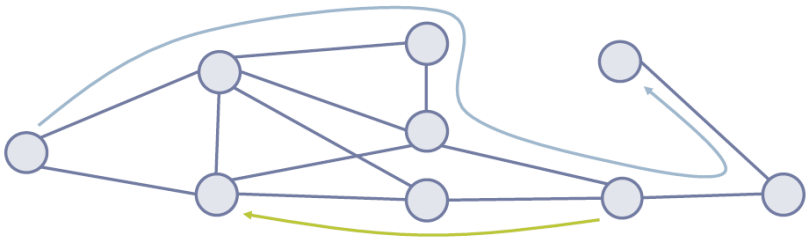
A	B 1 → D 3 → F 5
B	E 2
C	A 5 → B 1
D	A 1
E	C 2 → G 3
F	G 8
G	B 2 → C 4

Graph Traversal

- Like tree traversals, graph traversals systematically visit every (reachable) vertex in a graph
- Unlike trees, graphs may contain cycles, and the degree is not consistent
- Breadth-first search
    - Visits all vertices within d "hops" away from starting vertex, before visiting vertices within $d + 1$ hops, where d begins at 0
    - Radiate outwards from starting vertex



- 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



Spanning Trees

- A spanning tree $G' = V, E'$ must:
    - contain all the vertices of G
    - be connected
    - not contain cycles
- DFS spanning tree
- BFS spanning tree

Minimum Spanning Tree

- 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 (i.e. a tree)
    - The sum of the edge weights of E' is minimal among all such spanning trees

Kruskal's Algorithm

- Greedy algorithm, that 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
    - Use Priority queue to find the minimum weight edge

    - Use Disjoint sets to check for cycles and perform union

- Running Time: $O(E \log V)$

Prim's Algorithm

- Greedy algorithm
- 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
- Running Time: $O(E \log V)$

Single Source Shortest Path – Purpose of Minimum Spanning Tree

- Given a graph $G = V,E$ and a vertex $s \in V$, find the shortest path from s to every vertex in V
- Graph assumptions:
    - Undirected
    - weighted graphs
    - no negative cycles

Dijkstra's Algorithm

- A greedy algorithm
- Algorithm for solving shortest path in weighted graphs without negative weights
- Steps:

- Initialize the cost of reaching each vertex to ∞
- Initialize the cost of the source to 0
- While there are unvisited vertices left in the graph
    - Select the unvisited vertex with the lowest cost: u
    - Mark u as visited, and note the vertex v which was used to reach u
    - For each vertex w which is adjacent to u, update lowest cost

Running Time

Sorting

Name	*Best*	*Average*	*Worst*	*Memory*
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	$O(n^2)$	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)

NOTE: Mergesort and Quicksort are recursive and require additional space on the call stack. Heapsort can be implemented iteratively and requires no additional stack space except for local variables.

Binary Search Tree – *Run time depends on height of the tree*

Operations	*Average Case*	*Worst Case*	*Sorted Array*	*Sorted List*
Find	O(logn)	O(n)	O(logn)	O(n)
Insert	O(logn)	O(n)	O(n) – to shift data	O(n)
Delete	O(logn)	O(n)	O(n) – to shift data	O(n)
Traverse	O(n)	O(n)	O(n)	O(n)

Dictionary ADT Worst Case Complexities

	Insert	*Remove*	*Find*
Unordered Array	O(1)	O(n)	O(n)
Ordered Array	O(n)	O(n)	O(logn) – binary search
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(logn)	O(logn)	O(logn)

Graph Performance

n vertices *m edges* *No self-edges*	*Adjacency list*	*Adjacency matrix*
Space	O(n+m)	$O(n^2)$
IncidentEdges(v)	O(deg(v))	O(n)
areAdjacent(v,w)	O(deg(v))	O(1)
insertVertex(x)	O(1)	$O(n^2)$
removeVertex(x)	O(deg(v))	$O(n^2)$
insertEdge(v,w)	O(1)	O(1)
removeEdge(v,w)	O(deg(v))	O(1)

Kruskal's Algorithm

Priority Queue	*Heap*	*Sorted Array*
To build	O(m) - # of edges	O(mlogn)* mlogm <= $mlogn^2$ = 2mlogn Therefore O(mlogn)
Each removeMin	O(logn)	O(1)

Priority Queue	*Total Running Time of Kruskal's Algorithm*
Heap	O(n+m+mlogn) = O(mlogn) – since n and m is smaller than mlogn for each vertex make it a one item disjoint set (n), insert edges into Q (to build - m), and worst case we need to remove all edges (m edges) and each remove approx. cost logn. (mlogn)
Sorted Array	O(n+mlogn+m) = O(mlogn) – since n and m is smaller than mlogn

Prim's Algorithm (undirected graph with unconstrained edge weights)

Priority Queue \ Graph	*Adjacency matrix*	*Adjacency list*
Heap	$O(n^2 + mlogn)$	O(nlogn + mlogn)
Unsorted array	$O(n^2)$	$O(n^2+m)$

Which one to use depends on the density of graph.
Sparse graph $m \in O(n)$, Dense graph $m \in \Theta(n^2)$

Analysis of Algorithm Questions

1.

What is time complexity of fun()?

```
int fun(int n)
{
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count;
}
```

2.

What is the time complexity of fun()?

```
int fun(int n)
{
  int count = 0;
  for (int i = 0; i < n; i++)
     for (int j = i; j > 0; j--)
        count = count + 1;
  return count;
}
```

3.

What is the worst case time complexity of insertion sort where position of the data to be inserted is calculated using binary search?

4.

In a modified merge sort, the input array is splitted at a position one-third of the length(N) of the array. Which of the following is the tightest upper bound on time complexity of this modified Merge Sort.

5.

In the following C function, let n >= m.

```
int gcd(n,m)
{
  if (n%m ==0) return m;
  n = n%m;
  return gcd(m, n);
}
```

How many recursive calls are made by this function?

(A) \theta(logn)

(B) \Omega(n)

(C) \theta(loglogn)

(D) \theta(sqrt(n))

6.

Find Close Form for T(n)

T(1) = 1

T(n) = 2T(n − 1) + n, n ≥ 2

7.

What is the time complexity of following function fun()? Assume that log(x) returns log value in base 2.

```
void fun()
{
   int i, j;
   for (i=1; i<=n; i++)
      for (j=1; j<=log(i); j++)
         printf("CPSC221");
}
```

Linked List Coding Question

1.

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Example 1:

```
Input: head = [4,5,1,9], node = 5
Output: [4,1,9]
Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.
```

Example 2:

```
Input: head = [4,5,1,9], node = 1
Output: [4,5,9]
Explanation: You are given the third node with value 1, the linked list should become 4 -> 5 -> 9 after calling your function.
```

2.

Remove Nth Node From End of List

Example:

```
Given linked list: 1->2->3->4->5, and n = 2.


After removing the second node from the end, the linked list becomes 1->2->3->5.
```

Note:

Given ***n*** will always be valid.

3.

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

Example:

```
Input: 1->2->4, 1->3->4
Output: 1->1->2->3->4->4
```

Analysis of Algorithm Questions!

1.

O(n)

For a input integer n, the innermost statement of fun() is executed following times. n + n/2 + n/4 + ... 1 So time complexity T(n) can be written as T(n) = O(n + n/2 + n/4 + ... 1) = O(n) The value of count is also n + n/2 + n/4 + .. + 1

2.

$\Theta(n^2)$

The time complexity can be calculated by counting number of times the expression "count = count + 1;" is executed. The expression is executed 0 + 1 + 2 + 3 + 4 + ... + (n-1) times. Time complexity = Theta(0 + 1 + 2 + 3 + .. + n-1) = Theta (n*(n-1)/2) = Theta(n2)

3.

$O(n^2)$

Applying binary search to calculate the position of the data to be inserted doesnt reduce the time complexity of insertion sort. This is because insertion of a data at an appropriate position involves two steps: 1. Calculate the position. 2. Shift the data from the position calculated in step #1 one step right to create a gap where the data will be inserted. Using binary search reduces the time complexity in step #1 from O(N) to O(logN). But, the time complexity in step #2 still remains O(N). So, overall complexity remains O(N^2).

4.

The time complexity is given by: T(N) = T(N/3) + T(2N/3) + N Solving the above recurrence relation gives, T(N) = N(logN base 3/2)

5.

$\Theta(n)$

6.

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

7.

O(nlogn)

The time complexity of above function can be written as: θ(log 1) + θ(log 2) + θ(log 3) + ... + θ(log n) which is θ (log n!) Order of growth of 'log n!' and 'n log n' is same for large values of n, i.e., θ (log n!) = θ(n log n). So time complexity of fun() is θ(n log n).

Linked List Coding Question

1.

```
void deleteNode(ListNode* node) {
        while(node->next->next != NULL){
            node->val = node->next->val;
            node = node->next;
        }
        node->val = node->next->val;
        delete node->next;
        node->next = NULL;
   }
```

2.

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
        if(head->next == NULL){
            delete head;
            return NULL;
        }
        int count = 0;
        ListNode* temp = head;
        while(temp->next != NULL){
            count++;
            temp = temp->next;
        }
        count++;
        if(count == 2 && n==1){
            temp = head;
            delete temp->next;
            temp->next = NULL;
            return head;
        }
```

```
        if(count >= 3 && n==1){
            temp = head;
            while(temp->next->next != NULL){
            temp = temp->next;
            }
            delete temp->next;
            temp->next = NULL;
            return head;
        }
        int index = count-n;
        if(index == 0){
            ListNode* d = head->next;
            delete head;
            head = d;
            return head;
        }

        temp = head;
        for(int i = 0; i < index-1; i++){
            temp = temp->next;
        }
        ListNode* d = temp->next;
        temp->next = d->next;
        delete d;

        return head;
    }
```

3.

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(l1 == NULL) return l2;
        if(l2 == NULL) return l1;
        ListNode* head;
        if (l1->val < l2->val){
            head = l1;
            l1 = l1->next;
        }else{
            head = l2;
            l2 = l2->next;
        }
        ListNode* curr = head;
        while(l1 != NULL && l2 !=NULL){
            if (l1->val < l2->val){
                curr->next = l1;
                l1 = l1->next;
                curr = curr->next;
            }else{
                curr->next = l2;
                l2 = l2->next;
                curr = curr->next;
            }
        }
        if(l1 == NULL && l2 != NULL){
            curr->next = l2;
        }
        if(l2 == NULL && l1 != NULL){
            curr->next = l1;
        }
        return head;
    }
```