- Amortized Analysis

$$\frac{\text{total time for } n \text{ operations}}{n}$$

c	s

resize

r_1

c	s	2	2

resize

r_2

c	s	2	2	5	

$2r$ copies per round

$$\sum_{k=1}^{r} 2k = r + r^2$$

$$r = \frac{n}{2}$$

$$O\left(\frac{n^2+2n}{4}\right) = O(n^2)$$

What if we instead double our space each round?

c

c	s

c	s	2	2

c	s	2	2	5			

2^r copies per round

$\sum_{k=0}^{r} 2^r \Rightarrow$ total copies in r rounds

$$= 2^{(2^r - 1)} + 1$$

$2^r = n \Rightarrow r = \log_2 n$

$2n-1$

$$O(2n-1) = O(n)$$

total time for n inserts (x2 strategy):

$$O(n)$$

Amortized time for each insert

$$O\left(\frac{n}{n}\right) = O(1)$$

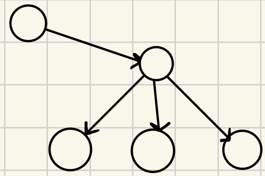
this does not mean each operation takes a constant amount of time.

- Queue ADT
  - [order]:
    FIFO (First In, First Out)
  - [Implementation]
    - enqueue (T data)
    - T dequeue ()
  - [Runtime]
    both functions O(1)
- Stack ADT
  - [order]
    - LIFO (Last In, First Out)
  - [Implementation]
    push(T)
    T pop()
  - [Runtime]
    both functions O(1)

Queue.h

```
template <typename T>
class Queue {
    public:
        void enqueue(T e);
        T dequeue();
        bool isEmpty();

    private:
        T *items_;                <- An array
        unsigned capacity_;
        unsigned size_;
        unsigned front_;
        unsigned where();

};

    enqueue(T e){
        items_[size++] = e;
    }

    dequeue
        size_--;
        return items_[front++];

    unsigned where(){
        return (size+front) % capacity
```

Trees:



- A tree is
  - an acyclic graph
  - In cs225, all trees have a root

- Binary Tree - Defined
  - A binary tree is either:
    1. $T = \emptyset$
    2. $T = \{r, T_L, T_R\}$

  - height: length of longest path from root to a leaf

    $\text{height}(T) = \max(\text{height}(T_L), \text{height}(T_r)) + 1$

  - A tree F is full if
    1. $F = \emptyset$
    2. $F = \{r, F_L, F_R\}$

       where either

       F_L and $F_R \neq \emptyset$

       or F_L and F_R are not empty

  - A perfect tree p is defined in terms of the tree's height
    1. $P_{-1} = \emptyset$
    2. $P_h = (r, P_{h-1}, P_{h-1})$

  - A complete tree
    1. A perfect tree for every level except last, where the last level is pushed to the left
    2. For all levels K in $[0, h-1]$, K has 2^{K-1} nodes

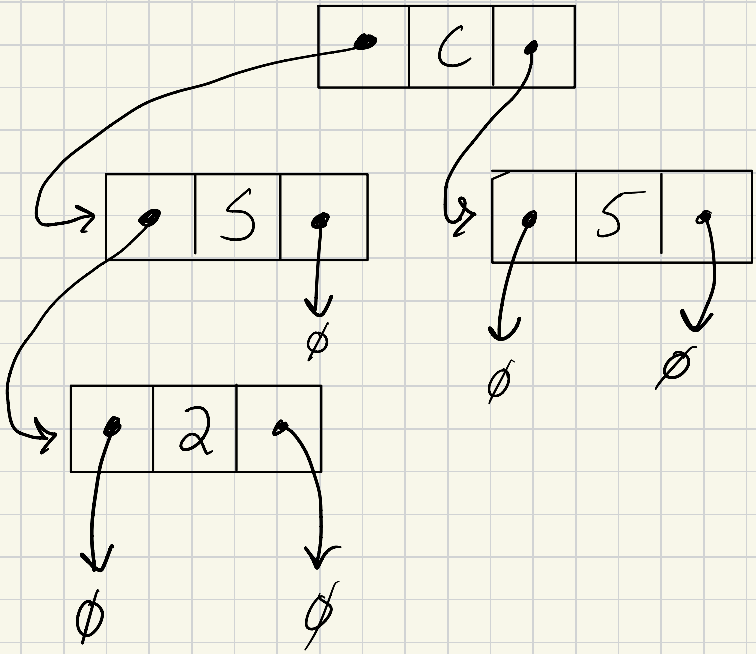
A complete tree C of height h, C_h:

1. $C_{-1} = \emptyset$
2. C_H (where $h>0$) $= \{r, T_L, T_R\}$ and either

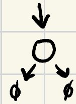
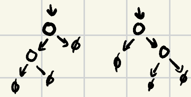
   T_L is C_{H-1} and T_R is P_{h-2}

   or

   T_L is P_{h-1} and T_R is C_{h-1}



How many Nulls in a binary tree with n data items

0 nodes: $\downarrow$ $\varnothing$ 1 node:  2 nodes: 

$n+1$ Null Pointers

<u>induction:</u>

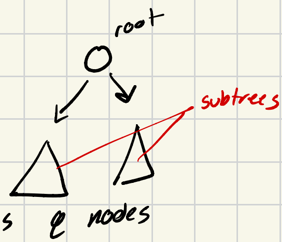
<u>Base:</u>

0 nodes $= 1$
1 node $= 2$

<u>Induction Hypothesis:</u>

suppose $\text{Nulls}(n) = n+1$ $\forall$ trees with $< K$ nodes

Consider an arbitrary tree T containing K data elements:
T has K nodes



WLOG: we choose an arbitrary subtree.

The first subtree contains $K - \underbrace{\ell}_{\text{other subtree}} - \underbrace{1}_{\text{root}}$ nodes

$\ell < K$

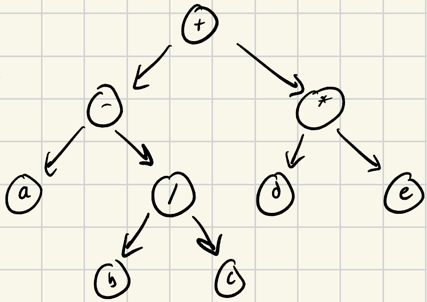
ℓ has $\ell + 1$ nulls by induction hypothesis.

The subtree has $K - \ell - 1 + 1$ nulls

T has $K \cancel{-\ell} \cancel{-1} \cancel{+1} \cancel{+\ell} + 1 = K+1$

Access all the nodes - traversals (how to visit all nodes in tree once)

one example:
(preorder traversal)



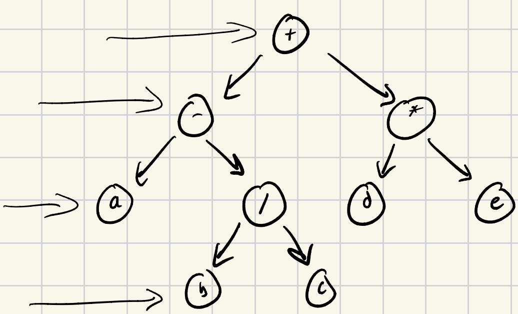

```
Traverse(root)
    if (root == NULL)
        return;
    visit(root)
    traverse(root->left)
    traverse(root->right)
```

Side note: "pointers" can be used in boolean:
zero if Null
one else

PostOrder: print after recursion
abc/-de*+

InOrder: print data in sequential order
a-b/c + d * e

A different type: go layer by layer




```
put root in a queue Q
while (Q not empty)
    node = Q.dequeue
    if(node)
        Visit(node)
        Q.enqueue(children)
```

+ - * a / d e b c

AVL Trees:

- Number of Nodes in an AVL Tree:

$$N(n) > 2^{h/2}$$

and inverting:

$$n >= N(h) > 2^{h/2}$$

$$n > 2^{h/2}$$

$$\log(n) > h/2$$

Other balanced BST

- Red-Black Tree
  - Max height: $2 * \lg(n)$
  - Constan number of rotations on insert, remove, and find.

- AVL Trees
  - Max height: $1.44 * \lg(n)$
  - rotations:

    insert: $O(1)$

    remove: $O(\log(n))$

Summery of Balanced BST

Cons:

- Runtime: $\log(n)$
- In-Memory Requirement

Every structure so far

	unsorted Array	Sorted array	unsorted list	Sorted list	BT	BST	AVL
Find	n	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(log(n))
Insert	O(1)*	O(n)	O(1)	O(n)	O(1)	O(n)	O(log(n))
Remove	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(log(n))
Traverse	O(n)	O(n)	O(n)	O(n)	O(n)	O(n) but also O(n)	O(n)

AVL is best so far

Iterators Types

- Forward
  - (in lecture) operator++, operator *, operator !=
- Bidirectional
  - operator--
- Random Access

  also in O(1) time

  if a and b are iterator

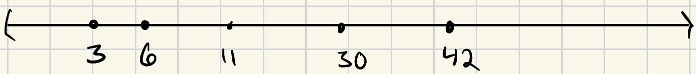
  (a+5) - five elements past a

  distance(a,b) - difference in # of elements from a to b.

- Random Access Iterators
  - Big steps
    - move by
      - + int
      - - int
      - += int
  - Distance
    - a<b - true if a is earlier in container than b
    - a-b - distance from a to b.

- Range-Based Searches
  - Balanced BSTs are useful structures for range-based and nearest-neighbor searches

What points fall in [11,42]?



R-B Trees in C++

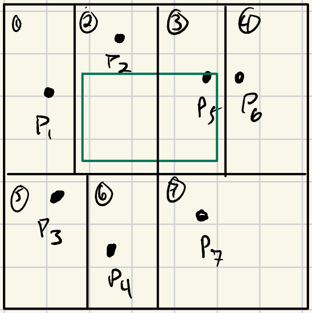
iterator std::map<K,V>::lowerbound(const K&); — an iterator at smallest not less than input

iterator std::map<K,V>::upperbound(const K&); — iterator smallest greater than input

or end

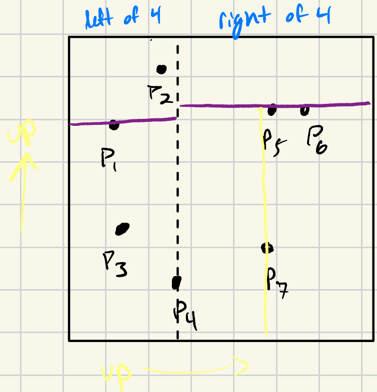
- Consider points in 2D: P{$P_1, P_2, P_3, \cdots P_n$}

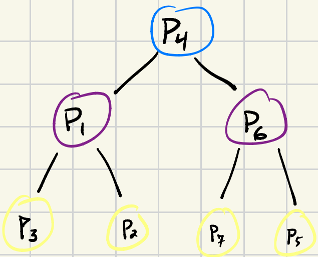
  What points are in the rectangle



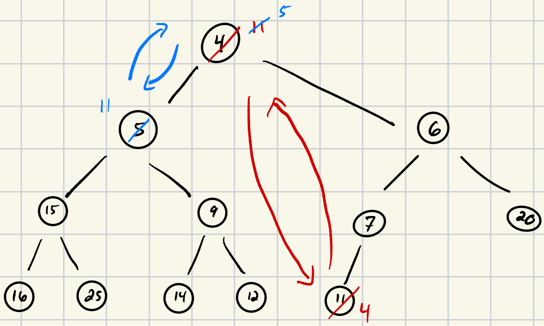
All we have to do is search box 2 + 3 because the square of interest is contained in 2+3. This is the best algorithm ever.

Tree construction:





March 4^{th} • Heap



5 ~~11~~	11										4	
~~4~~	~~5~~	6	15	9	7	20	16	25	14	12	~~11~~	

Want to remove min

1. Swap root with last
2. Swap root with minChild if larger

- Disjoint Sets

$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

2 5 9   7

0 1 4 8   3 6

$\{\{2,5,9\}, \{7\}, \{0,1,4,8\}, \{3,6\}\}$

Union two sets $\{\{2,5,7,9\}, \{0,1,4,8\}, \{3,6\}\}$

- Find Union
- Add to universe
- Union

- Operation find(n)

- Find(4) - using example above, returns 0

0	1	2	3	4	5	6	7	8	9	
0	0	2	3	0	2	3	7	0	2	

store conical element

Use Find to determine if two elements belong to the same space

ex. Find(4) == Find(8)

- Maintain a collection $S = \{S_0, S_1, \dots S_K\}$
- Each set has a representive element (conical element)

Implementation #1

{0,1,4}   {2,7}   {3 5 6}

element →	0	1	2	3	4	5	6	7
conical → (data)	0	0	2	5	0	5	5	2

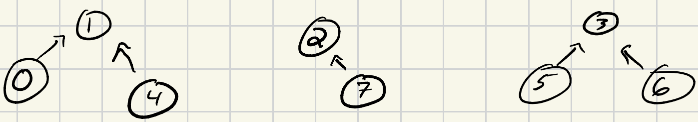
Find(K): return data[K]

Union(K_1, K_2):

```
for (i = 1:n){
    if (data(i) = find(K1))
        data(i) = K2
```

Implementation #2

{0 1 4}   {2 7}   {3 5 6}



0	1	2	3	4	5	6	7
1	-1	-1	-1	1	3	3	2

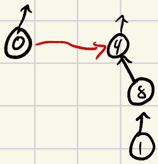
Find(K):

```
if(data[K] == -1)
return K
```

Union(K_1, K_2)

```
find(data[K])
data[K1] = K2
```

· While doing union:



union(0, 4)

we choose to add smaller height tree to the larger one

Getting started w/ graphs

Data Structures

Array
- sorted array
- unsorted array
  - stacks
  - queues
  - priority queues
    1. Heaps
  - Disjoint Sets
    1. Uptrees

Linked
- Doubly Linked List
- Trees
  - Btree
  - Binary Tree
    1. Kdtree
    2. AVL Tree
    3. BST
    4. BBST

Common Vocabulary:

$G = (V, E)$

$|V| = n$

$|E| = m$

Incident Edges:

$I(v) = \{\{x, v\} \text{ in } E\}$

Degree(v) : $|I|$

Adjacent Verticies:

$A(v) = \{x : \{x, v\} \text{ in } E\}$

Path : Sequence of vertices connected by edges

Cycle: Path w/ a common beginning and end vertex with at least 3 verticies

Subgraph:

$G' = (V', E')$:

$V' \in V$, $E' \in E$ and

$(u,v) \in E' \rightarrow u \in V', v \in V'$

Complete subgraph - all vertices are connected

Connected subgraph -

Connected component -

Acyclic subtree - a tree, a graph w/ no cycles

Spanning tree -

How many edges:

Minimum Edges:

Not connected: 0

Connected*: $n-1$

Maximum Edges:

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

Not Simple: no bound

Graph ADT

Data:

- Verticies
- Edges
- Some data structure maintaining the structure between verticies and edges

Functions:

- insertVertex(K key)
- insertEdge(Vertex v1, Vertex v2, K key)

- removeVertex(Vertex v)
- removeEdge(Vertex 1, Vertex 2)

- incidentEdges
- areAdjacent

- origin
- destination

Minimum Spanning Tree Algorithms

Input: Connected, undirected graph G with edge weights

Output: A graph G` with the following properties:

1. G` is a spanning graph of G
2. G` is a tree (connected, acyclic)
3. G` has a minimal total weight among all spanning trees