Trees 🌲

Why We Need To Know

We need to know which data structure to utilize within certain use cases so that we may optimize our program's functionality as well as usability. We need to know when to use these structures and how to implement them.

Time Complexity Of Data Structures

Operation	Vector	Linked List	Deque	Tree (Unordered)	Hashtable (Unordered)
Insert front	O(n)	O(1)	O(1)	O(log(n))	NA
Insert Back	O(1)	O(1)	O(1)	O(log(n))	NA
Insert Middle	O(n)	O(1)	O(N)	O(log(n))	O(1)*
Remove Front	O(n)	O(1)	O(1)	O(log(n))	NA
Remove back	O(1)	O(1)	O(1)	O(log(n))	NA
Remove Middle	O(n)	O(1)	O(n)	O(log(n))	O(1)*
Random Access	O(1)	O(n)	O(1)	O(log(n))	NA
Search	O(n)	O(n)	O(n)	O(log(n))	O(1)*

- star is the Average Complexity
- C++ STL ordered map and set: balanced tree
- C++ STL unordered map and set: Hash table

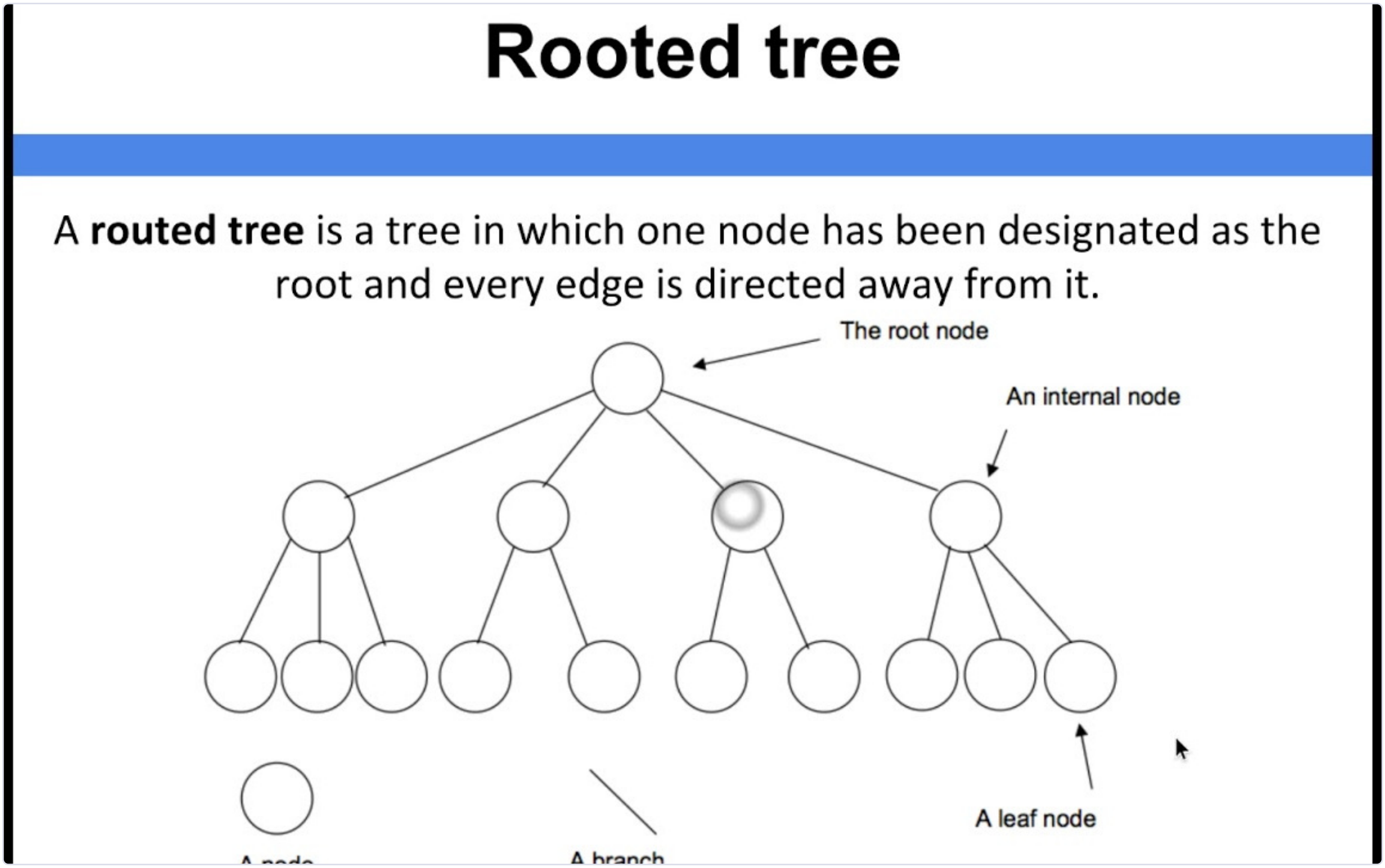
Theory and Terminology

Tree

- The tree is a connected graph with no cycles (no circles) - Consequences: - Between any two vertices, there is exactly one unique path

Rooted Tree

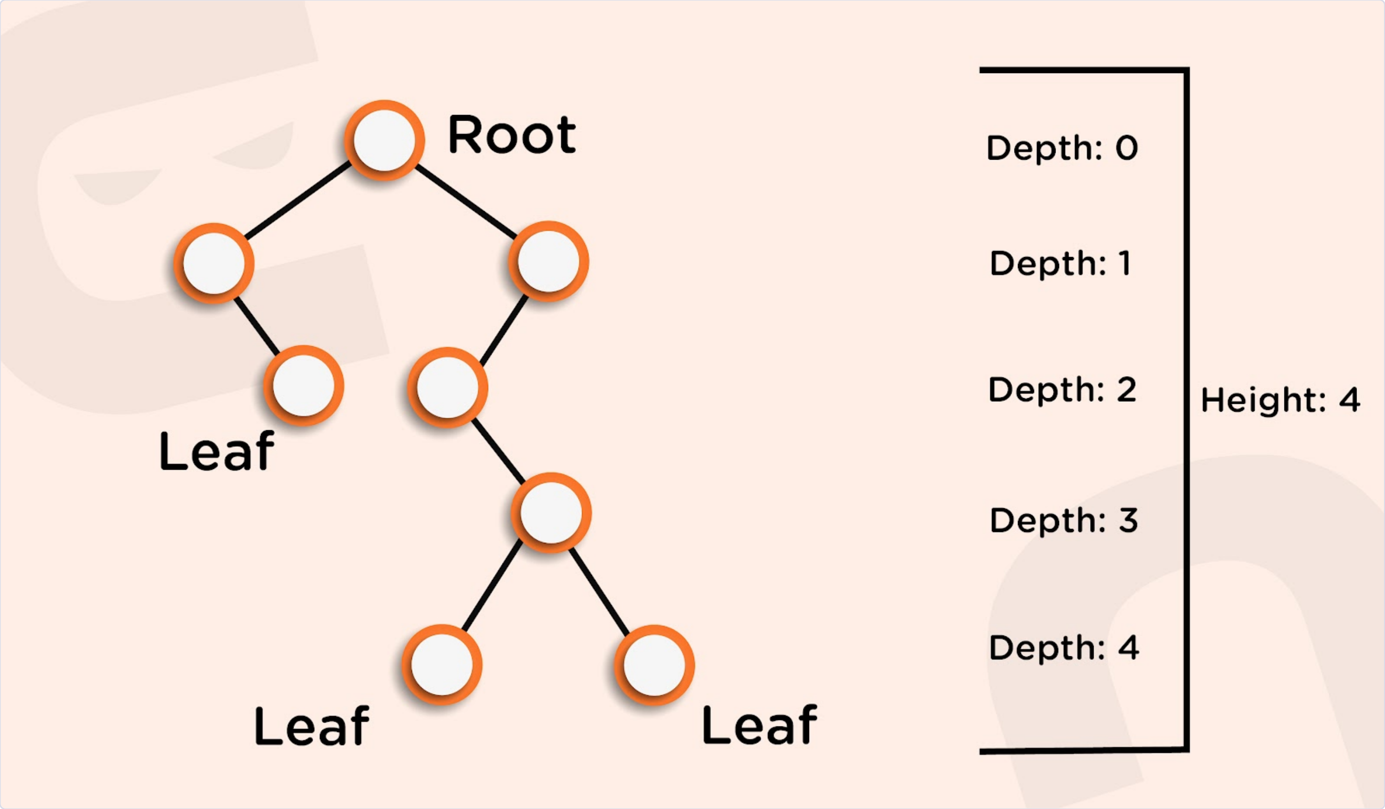
- A rooted tree - is connected - has no cycles - has exactly one vertex called the root of the tree - Consequences - Can be arranged so that the root is at the top - parent vs. child nodes and edges - sibling nodes - Nodes of the same parent nodes - leaf nodes - Nodes without children nodes



Rooted tree

A **routed tree** is a tree in which one node has been designated as the root and every edge is directed away from it.

- A unique path from the root to any vertex is a descending path
- Depth of vertex
  - Length of the unique descending path from the root to v
  - the root is at a depth 0
- The height of a vertex v is the length of the longest path from v to one of its leaves
- The height of a tree is the height of the root
  - Equal to the max depth



Rooted Tree: Recursive Definition

- A graph with N nodes and N-1 edges
- Graph has...
  - one root r
  - Zero or more non-empty subtrees

```C++
// Tree Node
struct TreeNode{
        Object element;
        TreeNode *firstChild;
        TreeNode *nextSibling;
}
```

Tree Traversal

- Often defined recursively
- Each kind corresponds to an iterator type
- Iterators are implemented non-recursively

Step	Description
1	Go to the root
2	Visit child subtrees

- Depth First Search
    - Begin at root
    - Visit vertex on arrival
- An implementation may be a recursive, stack-based, or nested loop
- For more information on depth-first search visit Stacks notes

Postorder Transversal

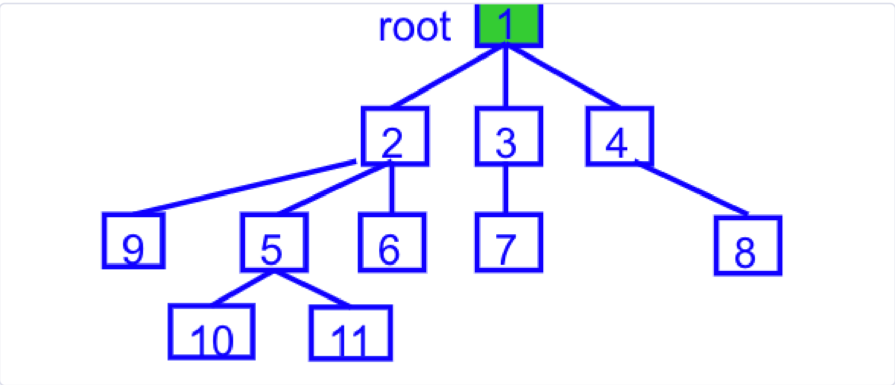
- The left subtree is traversed first
- Then the right subtree is traversed
- Finally, the root node of the subtree is traversed

Inorder Transversal

- The left subtree is traversed first
- Then the root node for that subtree is traversed
- Finally, the right subtree is traversed

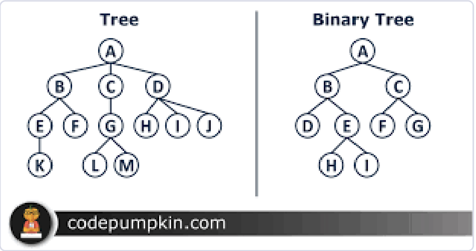
Preorder Transversal

- The root node of the subtree is visited first.
- Then the left subtree is traversed.
- At last, the right subtree is traversed.



Binary Tree

- Each node has at most two children, referred to as the left child and the right child.
- Every layer except maybe the bottom is fully populated with vertices.
- All nodes at the bottom level must occupy the leftmost spots consecutively



A complete binary tree with n vertices and h height satisfies

- $2^H \leq n < 2^{H+1}$
- $H \leq log(n) < H + 1$
- $H = floor(log(n))$

Trees 2 🌲

Binary Trees

- A binary tree is a rooted tree with no vertex
- has more than two children
  - left and right child nodes

```C++
// Syntax
struct BinaryNode{
        Object element;     // The Data in the node
        BinaryNode *left;   // & of Left Child
        BinaryNode *right;  // & of Right Child
}
```

- A binary tree is complete iff every layer except possibly the bottom is fully populated with vertices. All nodes at the bottom level must occupy the leftmost spots consecutively.
- A complete binary tree with n vertices and h height satisfies
  - $2^H \leq n < 2^{H+1}$
  - $2^2 \leq 7 < 2^{2+1}, 2^2 \leq 4 < 2^{2+1}$
  - $2^H \leq n < 2^{H+1}$
  - $H \leq log(n) < H + 1$
  - $H = floor(log(n))$
- **Proof:**
  - At level $k \leq H - 1$, there are 2^K vertices
  - At level $k = H$, there is at least 1 node, and at most 2^H vertices
  - Total number of vertices when all levels are fully populated (maximum level k)
    - $n = 2^0 + 2^1 + ... + 2^k$
    - $n = 1 + 2^1 + 2^2 + ... 2^k$ (Geometric Progression)
    - $n = \frac{1(2^{k+1} - 1)}{2-1}$
    - $n = 2^{k+1} - 1$

Case 1:

A tree has the maximum number of nodes when all levels are fully populated

- Let $k = H$
  - $n = 2^{H+1} - 1$
  - $n < 2^{H+1}$

Case 2:

The tree has a minimum number of nodes when there is only one node in the bottom level

- Let $k = H - 1$
  - $n' = 2^{H} - 1$
  - $n \geq n' + 1 = 2^{H}$

Combining two conditions we have

- $2^{H} \leq n \leq 2^{H+1}$

Representation of Complete Binary Tree

- All trees can be represented by the generic representation shown in the code above
- Due to the structure of a complete binary tree, it cannot be represented by a vector
  - As long as one can figure out the parent/child relationship
  - Parent/child relationship embedded in the index of parent and child
  - Vector elements carry data
- **Tree Structure : Vector**
  - Vector indices carry tree structure
  - Index order = levelorder
  - The tree structure is implicit
  - Uses integer arithmetic for tree navigation
  - No need to explicitly store the tree node pointers
- **Tree Navigation : Vector**
  - The root at `v[0]`
  - Parent of `v[k] = k[(k-1)/2]`
  - Left child of `v[k] = v[2*k + 1]`

- Right child of `v[k] = v[2*k]+1`

Binary Tree Traversal

Inorder traversal

- Definition
  - Left child
  - Vertex
  - Right Child (recursive)
- Algorithm
  - Depth-first search (visit between children)

Other Traversals that apply to binary case

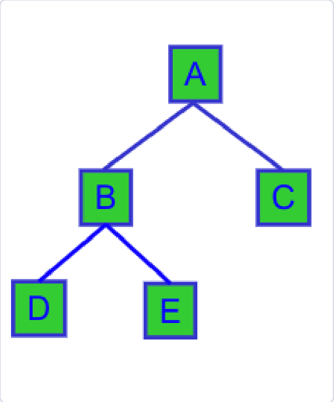
- Preorder
  - Trees 🌲
- Postorder
  - Trees 🌲
- Level order traversal

> A tree can be rebuilt from its inorder and preorder (or postorder) traversal results

---

Rebuild Tree from Traversal

- Let each node be associated with a letter, traversals print the letters when visiting a node. The results are:
  - Preorder: "ABDEC"
  - Postorder: "DEBCA"
  - Inorder: "DBEAC"



Rebuild tree from preorder + inorder traversal

- Find the root from the preorder result: A
- Decide left and right subtrees
  - Find the letter for the root in the inorder string and decide the inorder string for the two subtrees
- Decide the preorder string for left and right subtrees
  - Inorder for the traversal string length should be equal to another string length, extract preorder strings from the whole preorder string
- Recursively do this to the sub-trees

Build Expression Tree from Postfix Expression

```C++
stack<T> s;
while(s != /*end of postfix expression*/){
        // Get the next token
        if(token == /* operand */){
                // Create a new node with the operand
                s.push(/* New Node */);
        }
        if(token == /* operator */){
                s.pop(); // corresponding operands from S
                // Create a new node with the operator (and
corresponding operands as left/right children)
                s.push(/* New Node */);
        }
} // s.top is the final binary expression
```

Rebuilding with a tree

- Postorder string: FECAHJIGB
- Inorder string: CFEABHGJI

Root: B

- Last in the post-order string

  Left Subtree: CFEA
- Before root (B) in the inorder string

  Right Subtree: HGJI
- After root (B) in the inorder string

Trees 3 🌴

Everything on the left should be smaller than the right within a binary search tree. In other words the smallest element must be on the left.

- The complexity must be $O(n)$, due to all nodes needing to be traversed
- Assumes nodes are organized in a totally ordered binary tree

Consequences

- The smallest element in a binary search tree is the "leftmost" node.
- The largest element is the rightmost node
- Inorder traversal of the BST encounters nodes in increasing order

Search In BST

- Compare the search value to the current node, and decide whether to go left or right (depending if less (left) or more (right).
- Runtime ≤ descending path length ≤ depth of tree or height of tree

C++

```cpp
/* How It Works */

// Note: This is my personal code idea not concrete implementation, check documentation for better code example

void TreeAddNode(Node* currNode, Node* nodeAdd){
        if(currNode != nullptr){
                if(currNode->val > nodeAdd->val){
                        TreeAddNode(currNode->left, nodeAdd);
                }
                else if(currNode->val < nodeAdd->val){
                        TreeAddNode(currNode->right, nodeAdd);
                }
        } else {
                if(currNode->val > nodeAdd->val){
                        currNode->left = nodeAdd;
                }
                else if(currNode->val < nodeAdd->val){
                        currNode->right = nodeAdd;
                }
        } return;
}
```

Delete Node From Tree

- Must restructure the tree
- Find the similar branches to restructure
- Pick largest node in the subtree to be a new root

Finding the Minimum

- We can do this recursively, by going all the way to the left, aka following all of the left nodes
- If we have no left notes then we are done with following the tree (we have found the minimum value)

Tail Recursion

- Recursion is in the last line of the program
- Can be replaced with a loop (some compilers do this by default), very efficient

Finding the Maximum

- Non recursive

Implementation :

```C++
Node* findMax(){
        if(t!= nullptr){
                while(t->right != nullptr){
                        t=t->right;
                }
                return t;
        }
}
```

Deletion Example

```C++
#include <iostream>
using namespace std;

```

```
void deleteNode(const Comparable& x, BinaryNode* &t){
    if(t == nullptr){
        return;
    }
    if(x <t>element){
        remove (x, t->left);
    }
    else if(t->left != nullptr && t->right != nullptr){
        t->element = findMin(t->right)->element;
        remove(t->element, t->right);
    } else{
        BinaryNode * oldNode = t;
        t = (t->left != nullptr) ? t->left: t->right;
        delete oldNode;
    }
```

Destructor

- Left tree first
- Right tree second
- loop, Post order traversal, can use any traversal

Course Code Examples

IO Examples :

```C++
// IO Examples

Template <typename Comparable>

Class BinarySearchTree {

   public:

BinarySearchTree();

BinarySearchTree(const BinarySearchTree & rhs); // copy

BinarySearchTree(BinarySearchTree &&rhs); // move

~BinarySearchTree();
```

```
const Comparable & findMin() const;
const Comparable & findMax() const;
bool contains(const Comparable &x) const;
bool isEmpty() const;
void printTree(ostream & out = std::cout) const;


void makeEmpty();
void insert(const Comparable &x);
void insert(Comparable &&x); // move
void remove(const Comparable &x);


BinarySearchTree & operator=(const BinarySearchTree &rhs);
BinarySearchTree & operator=(BinarySearchTree && rhs); // move
```

Finding Smallest Element :

```C++
BinaryNode * findMin( BinaryNode *t ) const
    {
        if( t == nullptr )
            return nullptr;
        if( t->left == nullptr )
            return t;
        return findMin( t->left );

    }
```

Finding the Largest Element :

```C++
BinaryNode * findMax( BinaryNode *t ) const
    {
        if( t != nullptr )
            while( t->right != nullptr )
                t = t->right;
        return t;
}
```

Deletion :

```C++
void remove( const Comparable & x, BinaryNode * & t ) {

        if( t == nullptr )

            return;   // Item not found; do nothing

        if( x < t->element )
            remove( x, t->left );
        else if( t->element < x )
            remove( x, t->right );
        else if( t->left != nullptr && t->right != nullptr )  { // two children
            t->element = findMin( t->right )->element;
            remove( t->element, t->right );
        }
        else {
            BinaryNode *oldNode = t;
            t = ( t->left != nullptr ) ? t->left : t->right;
            delete oldNode;
        }
    }
```

Destructor :

```C++
    ~BinarySearchTree( )
    {
        makeEmpty( );
    }

    /**
     * Internal method to make subtree empty.
     */
    void makeEmpty( BinaryNode * & t )
    {
        if( t != nullptr )
        {
            makeEmpty( t->left );
            makeEmpty( t->right );
            delete t;
        }
        t = nullptr;
    }
```

Inorder Traversal :

C++

```
    // Print the tree contents in sorted order.
    void printTree( ostrem & out ) const
    {
        if( isEmpty( ) )
            cout << "Empty tree" << endl;
        else
            printTree( root, out);
    }

    /**
     * Internal method to print a subtree rooted at t in sorted
order.
     */
    void printTree( BinaryNode *t, ostream & out ) const
    {
        if( t != nullptr )
        {
            printTree( t->left );
            out << t->element << endl;
            printTree( t->right );
        }
    }
```

B-Trees 🌳

Balanced Binary Search Tree

- H = O(Log(N)). Insert, remove, and search all have complexity of O(log(n)) = O(Log_2(N))
- Each Node has a maximum of 2 children

C-ary Tree in Tree

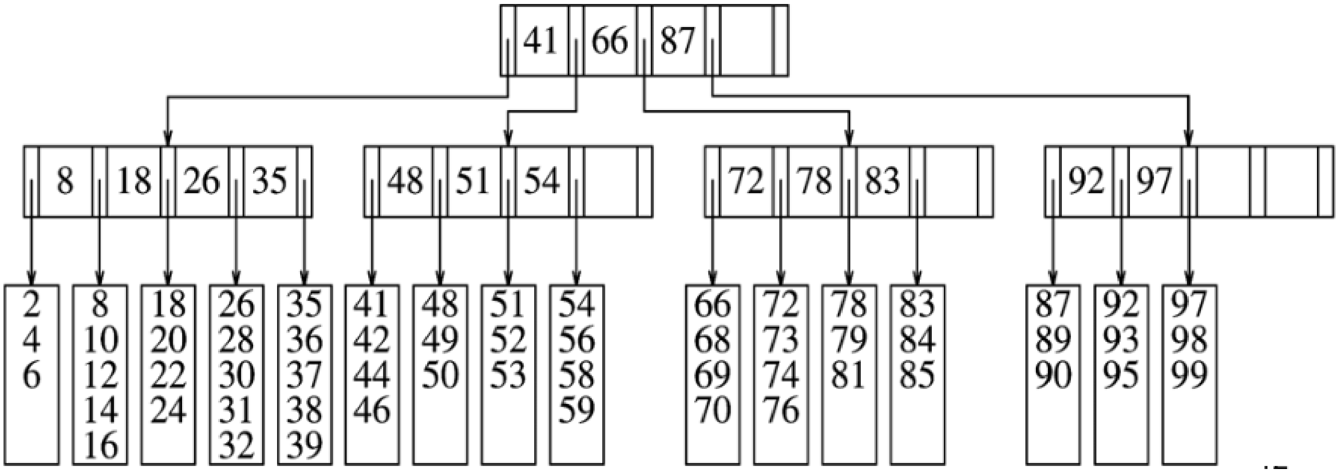
- $N = C^0 + C^1 + ... + C^H$
- $N = \frac{C^{H+1} - 1}{C-1} \approx C^H$
- So $H \approx log_{\in}(H) = \frac{Log_2(N)}{Log_2(C)}$
- If we increase the max number of children from 2 to C and maintain a balanced tree, the height is reduced by $(Log_2(C))$

M-ary Trees

- Allow up to M children for each node
  - Instead of 2 max for binary trees
- A complete M-ary tree of N nodes has a depth of $log_M N$
- Each node has (M-1) keys to decide which branch follows
- Larger M, smaller tree depth
- **Balancing M-ary**
  1. Restrict the tree shape like in AVL
  2. Restrict the number of children each node can have
- B-Tree takes the second approach, easier to implement

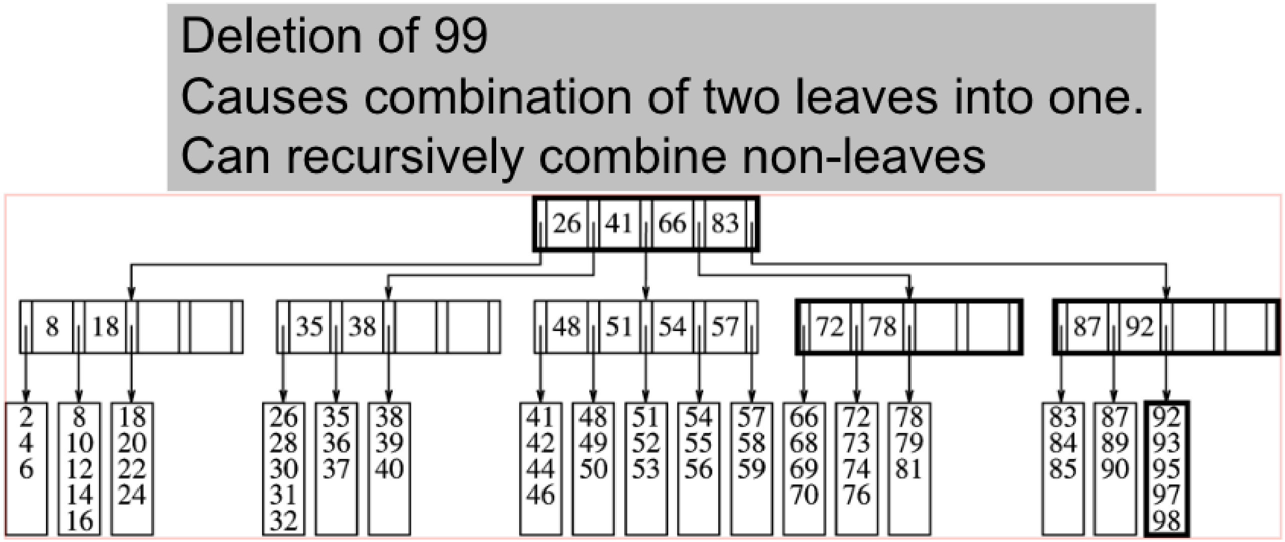
Balanced Trees (B-Tree)

- B-Tree is an M-ary search tree with restrictions
  1. Data items stored at the leafs
  2. Non-lead nodes store up to M-1 Keys to guide search
  3. The root can be a leaf or have between 2 to M children
  4. Non lead Nodes except the root have between ceil M/2 and M children
  5. All leaves are at the same depth, have between ceil(L/2) and L data items
- Keys in each node are sorted. The i'th key in a node is the smallest data in the i+1 subtree



Deletion

- Do a search to find the leaf node to delete
- If the lead still has at least L/2 data entries, done
- Else, merge the data with neighboring leaf to ensure L/2 data entries



Deletion of 99
Causes combination of two leaves into one.
Can recursively combine non-leaves

AVL Trees 🌴

- Binary search tree has simple search, insert, and remove
    - O(H) - H is the height of tree
    - Binary search does not guarantee small H, in worst case O(n)
- Ideal to maintain a binary search tree who's height H is O(log(N))
    - Search, insert, min, max all O(log(N))
    - Balanced Tree

Adelson- Velski and Landis

- A balanced binary search tree
- Every node in tree, height of left and right subtree only differ by 1 (at most)
- Guarantees O(Log(N))

Height of AVL Tree

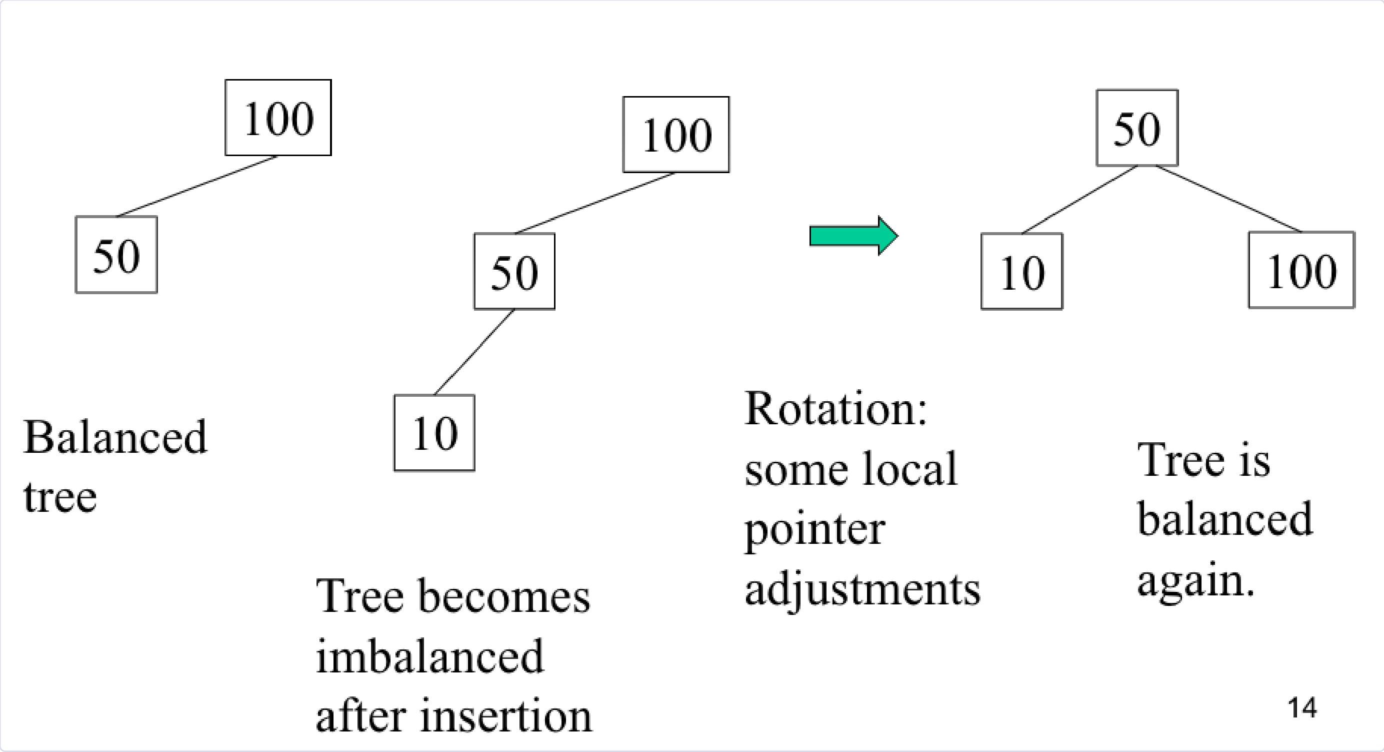
- For **every node** in tree, height of left and right subtree can differ by at most 1
- Let the number of nodes in the smallest AVL tree with height of H be N_H
    - The height of left and rightmost subtrees must have at least N_{H-2}
- Tree size at least doubles when H increases by 2, so the tree only needs to be twice the height as the full complete binary tree to have the same number of tree nodes
- **What is a balance condition**
    - The absolute difference of heights of left and right subtrees at any node is less than 1
- If we can maintain the balance condition in the insert and remove all operations to the AVL tree (with O(H) for each insert and remove we much have a data structure that archives O(log(N))) for search, insert and remove

Overhead

- Extra space needed for maintaining height information at each node, which is used to maintain balance of tree

Advantage

- Insert, Remove, and Search are all O(Log(N))



Balanced tree

Tree becomes imbalanced after insertion

Rotation: some local pointer adjustments

Tree is balanced again.

14

> **Important**
>
> Must Maintain Balance!
> This can be done through shifting the formation of the tree as shown above

Summary

- Find the deepest node whose AVL property is violated
  - Consider only the nodes from the root to the new node
- Preform the rotation

Delete

- Single rotation or double rotation, both are O(1)
- Depends on the shape of tree for single or double
- Delete can be done by deleting as in BST delete, and then fix all nodes along the path from the root to the deleted node, this would take at most O(Log(N))

Implementation

```C++
struct AvlNode
{
        Comparable      element;
        AvlNode         *left;
        AvlNode         *right;
        int                     height;
        AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int
h = 0 )
          : element{ ele }, left{ lt }, right{ rt }, height{ h } { }

        AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0
)
          : element{ std::move( ele ) }, left{ lt }, right{ rt },
height{ h } { }
};

 /**
  * Return the height of node t or -1 if nullptr.
  */
 int height( AvlNode *t ) const
 {
        return t == nullptr ? -1 : t->height;
 }
```

Insertion

```cpp
 /* Internal method to insert into a subtree.
     * x is the item to insert.
     * t is the node that roots the subtree.
     * Set the new root of the subtree.
*/
    void insert( const Comparable & x, AvlNode * & t )
    {
        if( t == nullptr )
            t = new AvlNode{ x, nullptr, nullptr };
        else if( x < t->element )
            insert( x, t->left );
        else if( t->element < x )
            insert( x, t->right );

        balance( t );
    }
```

```cpp
// Assume t is balanced or within one of being balanced
    void balance( AvlNode * & t )
    {
        if( t == nullptr )
            return;

        if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE
)
            if( height( t->left->left ) >= height( t->left->right ) )
```

```
                rotateWithLeftChild( t );
            else
                doubleWithLeftChild( t );
        else
        if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
            if( height( t->right->right ) >= height( t->right->left ) )
                rotateWithRightChild( t );
            else
                doubleWithRightChild( t );

        t->height = max( height( t->left ), height( t->right ) ) + 1;
    }
```

Single Rotation

```C++
void rotateWithLeftChild(AVLNode * &k2){
        AVLNode * k1 = k2->left;
        k2->left = k1->right;
        k1->right = k2;
        k2->height = max( height(k2->left), height( k2->right)) +1;
        k1->height = max( k1->left ), k2->height) + 1;
        k2 = k1;
}
```

Double Rotation

```C++
void doubleWithLeftChild(AVLNode * & k3){
        rotateWithRightChild(k3->left);
        rotateWithLeftChild(k3);
}
```

Remove

- Just do the binary search tree remove, and then call balance (t) at the end.

Example

```Markdown
T1, T2, T3 and T4 are subtrees.


         z                              y
        / \                           /   \
       y   T4    Right Rotate        x      z
      / \        - - - - - >       /  \    /  \
     x   T3                       T1  T2  T3  T4

```

```
   / \
 T1   T2
```

Stacks

Stack

- Last in First Out

```
#include <stack>
// Stack operations
stack<T> stackExample;
stackExample.push();
stackExample.pop();
stackExample.top();
stackExample.empty();
stackExample.size();
// with constructor & destructor
```

Stack Model - LIFO

- The top allows access to the top of the "Stack"
- Any list implementation could be used to make a stack
  - Operating on one end
- Vector/List ADTs
  - push_front()/pop_front()
  - push_back()/pop_back()

Stack Uses

- Depth-first search/backtracking
- Evaluating postfix expressions
- Converting infix to postfix
- Function calls (runtime stack)
- Recursion

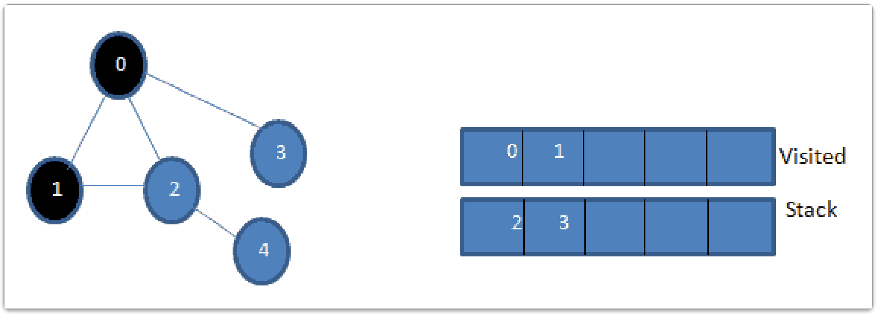
Runtime Stack

- Static
  - Executable code
  - Global variables
- Stack
  - Push for each function call
  - Pop for each function return

    - Local variables
- Heap
    - Dynamically allocated memories
    - new and delete

Depth First Expanded

- If there is an unlisted neighbor go there
- Retreat along the path to find unlisted neighbor, it cannot go deeper
- If there is a path from start to goal, DFS finds one such path
- Discover a path from **start** to the goal
    - Start from Node start stop if Node reaches goal



> *Keep Going Deeper*

```
// Depth First Search
DFS() {
stack<location> S;
//Mark the start location as visited
        S.push(start);
        while (!S.empty()) {
                t = S.top();
                if (t == goal) Success(S);
                if (// t has unvisited neighbors) {
                        //Choose an unvisited neighbor n
                        // mark n visited;
                        S.push(n);
                } else {
                        BackTrack(S);
                }
        }
        Failure(S);
}

/*
```

```
                        Another Implementation Of DFS

  ────────────────────────────────────────────────────
*/

BackTrack(S) {
        while (!S.empty() && S.top() has no unvisited neighbors) {
                S.pop();
        }
}

Success(S) {
        // print success
        while (!S.empty()) {
                output(S.top());
                S.pop();
        }
}

Failure(S) {
        // print failure
        while (!S.empty()) {
                S.pop();
        }
}
```

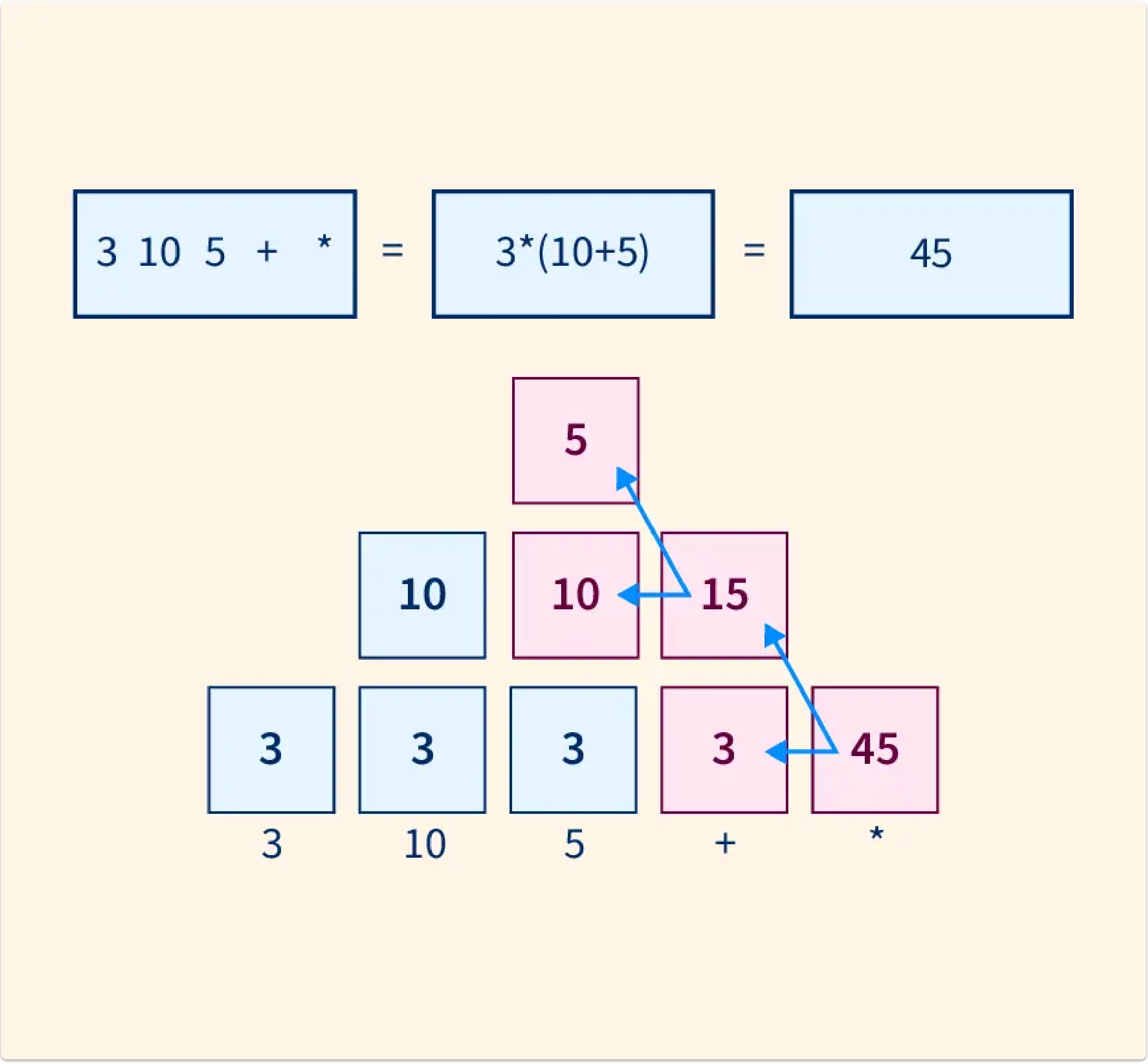
Postfix Expressions

- Use a stack of tokens
- Repeat
    - If operand, push onto the stack
    - If operator
    - pop operands off the stack
    - evaluate operator on operands
    - push the result onto the stack
    - Until expression is read
    - Return top of the stack

```
Evaluate(postfix expression) {

        // use stack of tokens;
        while(// expression is not empty) {
                t = next token;
                if (t is operand) {
                        // push onto the stack
                } else {
                        // pop operands for t off stack
                        // evaluate t on these operands
                        // push the result onto the stack
                }
        }
        // return top of stack
}
```

Postfix Visualized



Queue

Queue ADT - FIFO

- Elements of some proper type of T
- Operations
    - Feature: First In, First Out
    - void push(T t)
    - void pop()
    - T front()
    - bool empty()
    - unsigned int size()
    - Constructors and destructors

Operation Number	Command	Stack
1	Q.push(Ant)	Ant
2	Q.push(Bee)	Bee
3	Q.push(Cat)	Cat
4	Q.push(Dog)	Dog

> *What if we were to do Q.pop()?*

Operation Number	Command	Stack
1	Q.push(Bee)	Bee
2	Q.push(Cat)	Cat
3	Q.push(Dog)	Dog

Stack Uses

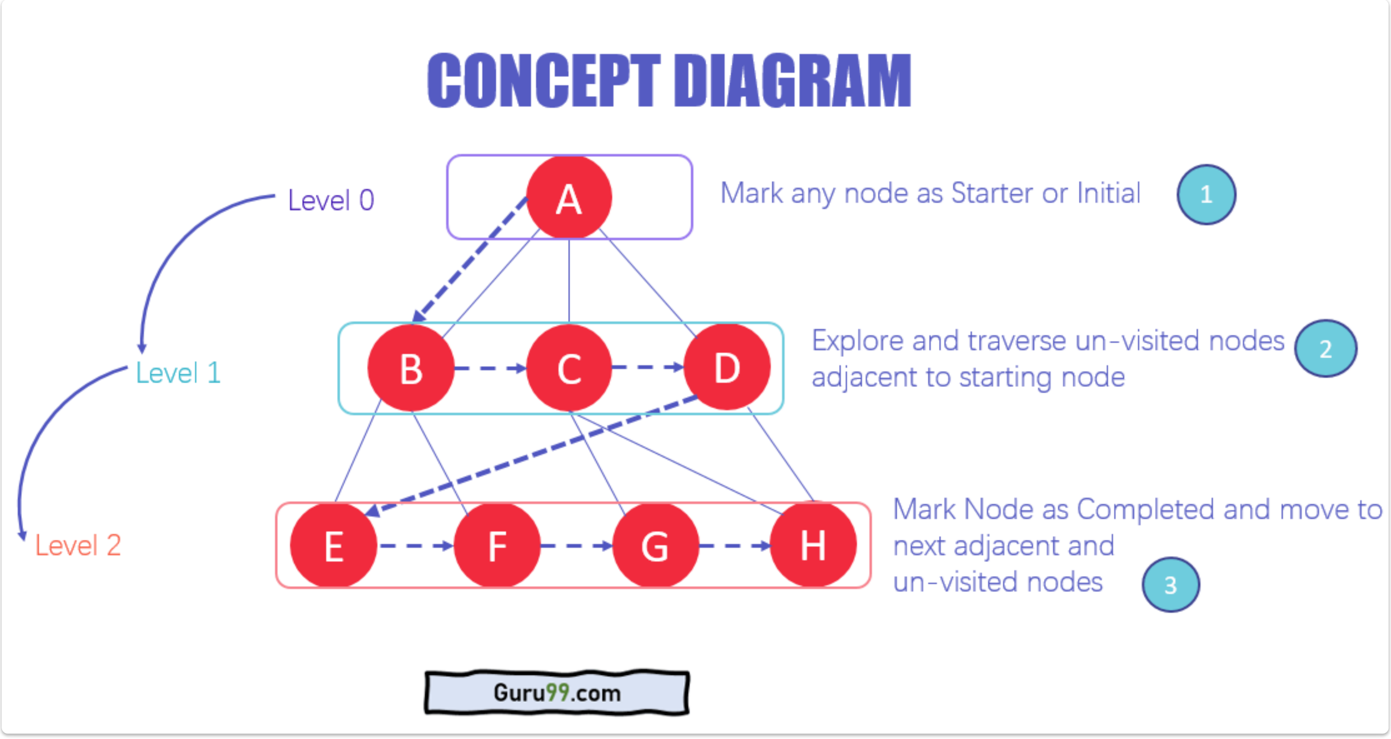
- Buffers
- Breadth first search
- Simulations
- Producer-Consumer Problems

Breadth first search Expanded

- Used to find the shortest path to the start to the goal
- Starting from Node start
- Visit all neighbors of the node

- Stop
    - if the neighbor is the goal
- Otherwise
    - Visit neighbors two hops away
- Repeat until visiting all neighbors

Breadth First Visualized


CONCEPT DIAGRAM
Level 0
Level 1
Level 2
A
B
C
D
E
F
G
H
Mark any node as Starter or Initial
1
Explore and traverse un-visited nodes adjacent to starting node
2
Mark Node as Completed and move to next adjacent and un-visited nodes
3
Guru99.com

Linked Lists

Singly Linked List

- push_front- Make the new node the head pointer, and make it point to the previous head node
- push_back- Make the tail node point to the new back node, and the back node point to null
- pop_front- Make the next node, following the head node, the new head node
- pop_back- Make the node before the tail node point to null, rather than the tail node

> *Efficiently = O(1) time complexity*

Doubly Linked List

- Contains a reference to the next element, as well as a reference to the previous element

```
// Insertion
auto I = Cities.begin();

for (; I != Cities.end(); ++I) {

        if (“Miami” == *I) {
        break;
        }
}

//Insert the new string

Cities.insert(I, “Orlando”);

// “Jacksonville”, “Tallahassee”, “Gainesville”, “Orlando”, “Miami”

// Remove Orlando
List<string>::iterator I = Cities.begin();

// auto I = Cities.begin();  // c++11
while( I != Cities.end()) {
if (“Orlando” == *I) {
        I = Cities.erase(I);
} else {
        I++;
}}
```

Node -

- Data Value
- Pointers to the previous and next element
- Defined within the List class, with limited scope

Creating A List

```
template <typename Object>

class List

{
  private:
    struct Node
    {
        Object  data;
        Node   *prev; // Points to previous Node
        Node   *next; // Points to next Node

        Node( const Object & d = Object{ }, Node * p = nullptr, Node * n = 
nullptr )
          : data{ d }, prev{ p }, next{ n } { }

        Node( Object && d, Node * p = nullptr, Node * n = nullptr )
          : data{ std::move( d ) }, prev{ p }, next{ n } { }
    };
```

Insertion within List

```


iterator insert( iterator itr, const Object & x )

    {

        Node *p = itr.current;

        ++theSize;
        return iterator( p->prev = p->prev->next = new Node{ x, p->prev, p
} );

    }



  iterator insert( iterator itr, Object && x )

    {

        Node *p = itr.current;

        ++theSize;

        return iterator( p->prev = p->prev->next = new Node{ std::move( x
), p->prev, p } );

    }
```

Empty List

```
  private:
    int   theSize;
    Node *head;
    Node *tail;
    void init( )
        // Doubly Linked List Init
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail; // Head -> &Tail
        tail->prev = head; // Tail (previos) -> &Head
    }

};
```

Erase Node

```

iterator erase( iterator itr )
```

```
   {

       Node *p = itr.current;

       iterator retVal( p->next );

       p->prev->next = p->next;

       p->next->prev = p->prev;

       delete p;

       --theSize;

       return retVal;
   }

iterator erase( iterator from, iterator to )

   {
       for( iterator itr = from; itr != to; )
           itr = erase( itr );
       return to;
   }
```

Priority Queues (Heaps)

Each job within a computer takes turns using the cpu. If a job comes earlier than another job it needs to be executed earlier. If a job has been scheduled, it should not be scheduled a second time if there exists a job that hasn't been scheduled once. The queue is the answer.

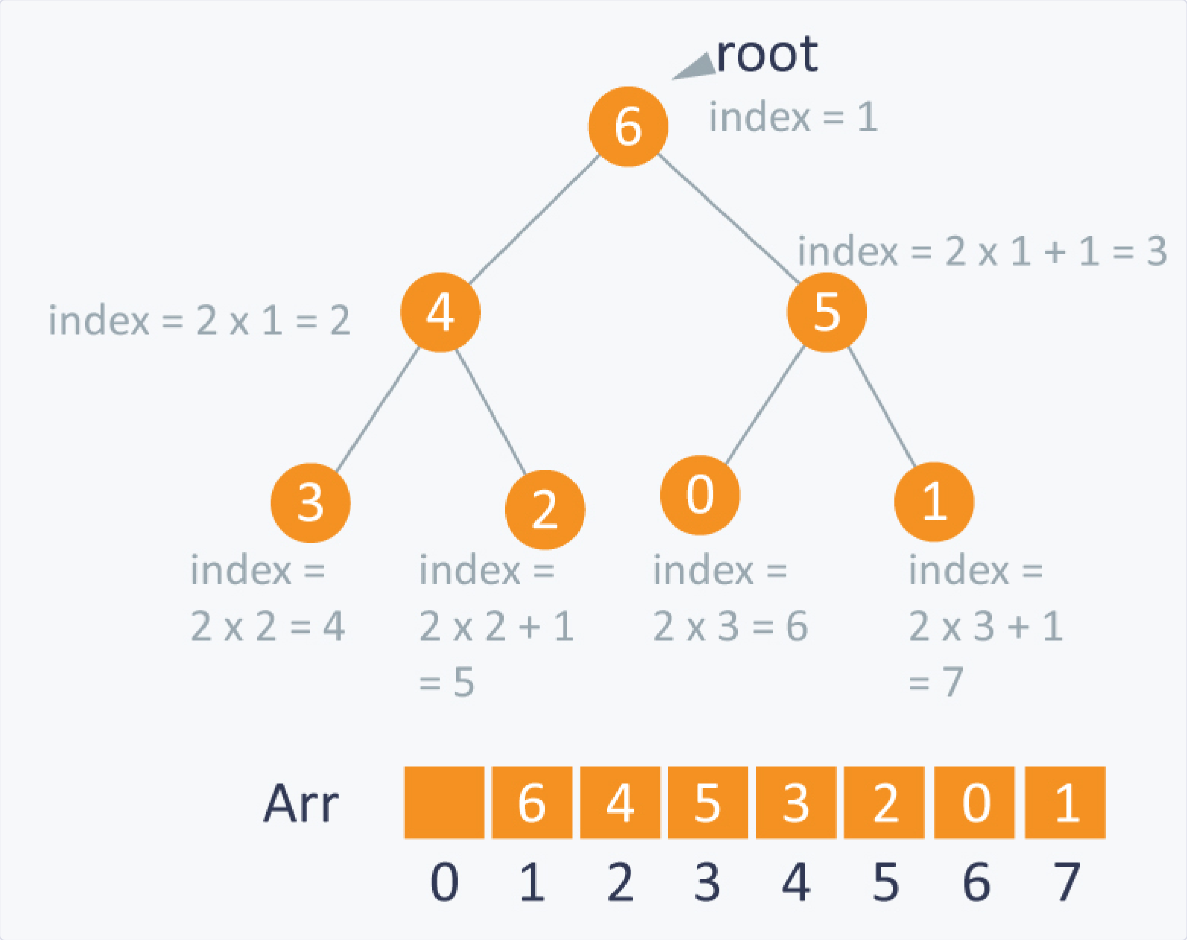
- first in first out
- when a new job comes in it is put at the end of the queue

Regular Queues

- Enqueue adds a new element
- Dequeue removes the eldest element
- Insert adds a new element
- deleteMin removes the minimum element in a priority queue

Application of Priority Queue

- Find the shortest job
- Scheduling next event based on time
- find greedy algorithms



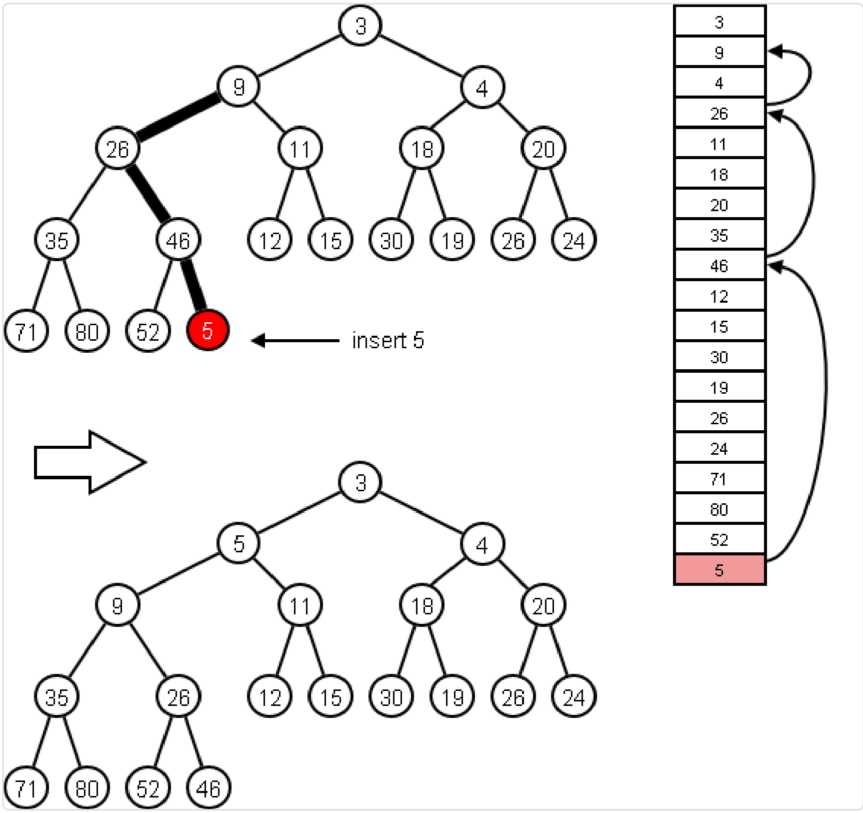
Applications of Priority Queue

Implemented as adaptor class around

- Linked list
  - O(N) worst case in insert or deleteMin
- AVL Tree
  - O(log(N)) worst case on insert and delete
  - Can be overkill when compared to heap, due to being sorted
- Heaps
  - O(log(N)) worst case for both insert and delete

Partially Ordered Trees (POT)

- There is an order relation <= defined for the vertices of T
- For any vertex p and any child c of p, p <= c
- smallest element is the root
- no conclusion can be drawn about the children



3
9
4
26
11
18
20
35
46
12
15
30
19
26
24
71
80
52
5

Binary Heaps

- A **binary heap** is a partially ordered complete binary tree
  - The tree is completely filled on all levels, with the exception of possibly the lowest
  - d-Heap, a parent node can have d children
  - just refer to as heaps

A vector representation of a complete binary tree is as follows...
Storing element in a vector in level order

- Parent of `v[k] = v[k/2]`
- Left Child of `v[k] = v[2*k]`
- Right Child of `v[k] = v[2*k + 1]`

Basic Heap Operations

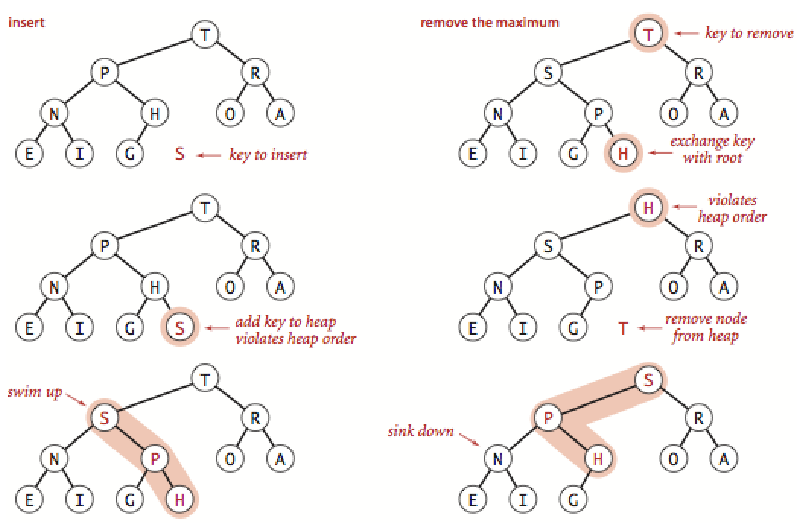
Insert

1. Create a hole at the next leaf
2. // Repair upward
3. Repeat
4. Locate Parent
5. if POT not satisfied (should x inserted in the hole)
    1. sliding parent element to hole
6. else
    1. stop

Delete Min

- Move the last element to the root and then fix the heap
    - Find the smaller child. If the new element is larger than the smaller child, swap with the small child (POT property violated), swap with the small child
    - Repeat in the new subtree

1. Delete the root
2. // root becomes a hole
3. Must move last element (leftmost node) to somewhere
4. let y be the last element (rightmost node)
5. Repeat
    1. find the smaller child of the node
    2. if POT not satisfied should y inserted in hole, sliding smaller child in whole
6. else
    1. stop
7. insert y into hole



Heap operations

Constructor

- Insert each element
- Worst Case O(N(Log(N))
- First insert all elements into a tree without worrying about POT, then satisfy the POT

Hash Table

Hash Tables support insert, remove and search in O(1) time.

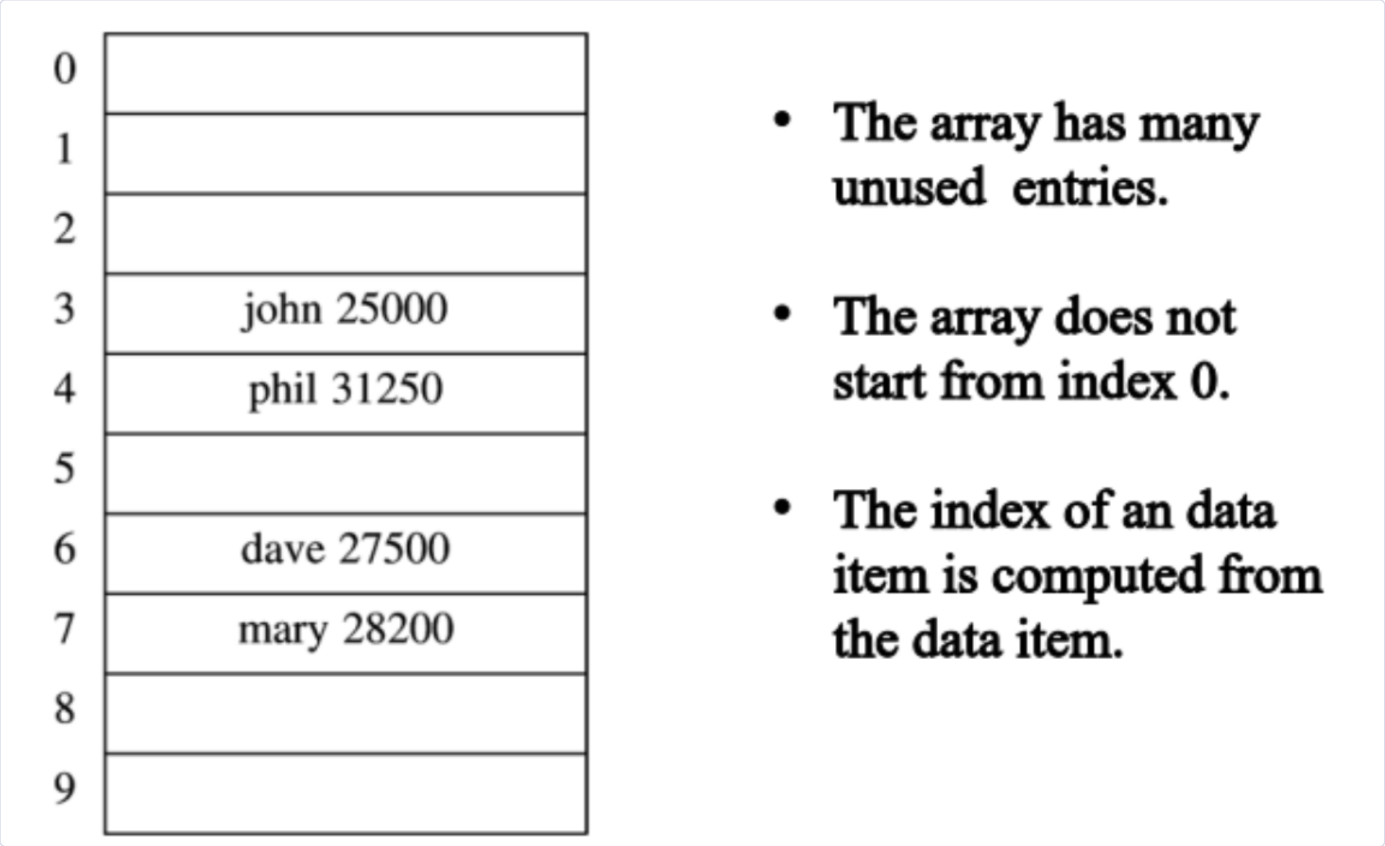
> **Important**
>
> **Idea Behind Hash Table:** O(1) is the worst case for insert, remove, search

Issues that need to be solved

- Key may or may not be an integer
  - Need to use a function to map the key into an integer. This part of the functionality of a hash function. To hash the key into an index.
- Key space can be infinite
  - **Examples**: The key is a string of any length
  - Need to restrict the hash function to return a small value in order to index into the hash table
- Hash typically map a fairly large number
- **Use a hash function to map key**: integer or non integer to a relatively small integer as the index to the hash table
- To get good performance we must also use the hash function to evenly map keys into different indices of the has table

Hashing

- Data items stored in an **array** of some fixed size
  - Hash table
- Search performed using some part of the data item
  - key
- Used for performing insertions, deletions, and finds in constant average time O(1)
- Operations requiring ordering information not supported efficiently



0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

- The array has many unused entries.
- The array does not start from index 0.
- The index of an data item is computed from the data item.

Applications of Hash Tables

- Comparing search efficiency of different data structures:
  - Vector, list: O(N)
  - Binary search tree: O(Log(N))
  - Hash table: O(1)
- C++ STL: `std::unordered_map`, `std::unordered_set`
- Compilers to keep track of declared variables
  - Symbol tables
- Game programs to keep track of positions visited
  - Transportation table
- On-line spelling checkers

Hashing Functions

- Map keys to integers
- `Hash(key) = Integer`
- Evenly distributed index values
  - Even if the data is not evenly distributed
- **Assumptions**:
  - K: an unsigned 32 bit int
  - M: the number of buckets (the number of entries in a hash table)

- **Goal**:
    - If a bit is changed in K all bits are equally likely to change for Hash(K)
    - So that items evenly distributed in hash table

Simple Function

- `Hash(K) = K % M`
- Where M is of any integer value
- Values of K may not be **evenly distributed**, however `Hash(k)` must be **evenly distributed**.
- **If** M = 10, K = 10, 20, 30, 40
- **Then** `K % M = 0, 0, 0, 0, 0 ...`

Another Example

- `Hash(K) = K % P`
- Where P is Prime
- **Suppose** then P = 11, K = 10, 20, 30, 40
- `Then K % P = 10, 9, 8, 7`
- **A well designed hash table always has a prime number of entries**

Hashing a Sequence of Keys

- K = $\{K_1, K_2, \ldots, K_n\}$
- `Hash("test") = 98157`
- Design Principles
    - Use the entire key
    - Use the ordering information

Use the Entire Key

```C++
unsigned int Hash(const string& Key){
        unsigned int hash = 0;
        for(string::size_type k =0; j !- K.size();++j)
        {
                hash = hash ^ Key[j]; // Xor
        }
        return hash;
}
// Problem: Hash("ab") == Hash("ba")
```

Use the Ordering Information

```C++
unsigned int Hash(const string &Key){
        unsigned int hash = 0;
        for(size_type j =0; j != Key.size(); ++j)
        {
                hash = hash ^ Key[j];
                hash = hash * (j % 32);
        }
        return hash;
}
```

Better Hash Function

```C++
unsigned int Hash(const string& S){
        size_type i;
        long unsigned int bigval = S[0];
        for(i = 1; i < S.size(); ++i){
                // low16 * magicNumber
                bigval = ((bigval & 65535) * 18000)
                + (bigcal >> 16) // high16
                + S[i];
        }
        bigval = ((bigval & 65535) * 18000) + (bigval >> 16);
        // bigval = low16 * magicNumber + high16

        // return low16
        return bigval & 65535;

}
```

Even if the function just returned 0 it would still be a legit hash function. Replacing the hash function in any hash table with this would still work but the O(1) complexity may not be maintained.

Hash Table 2

Recap of Hash Table 1 Notes

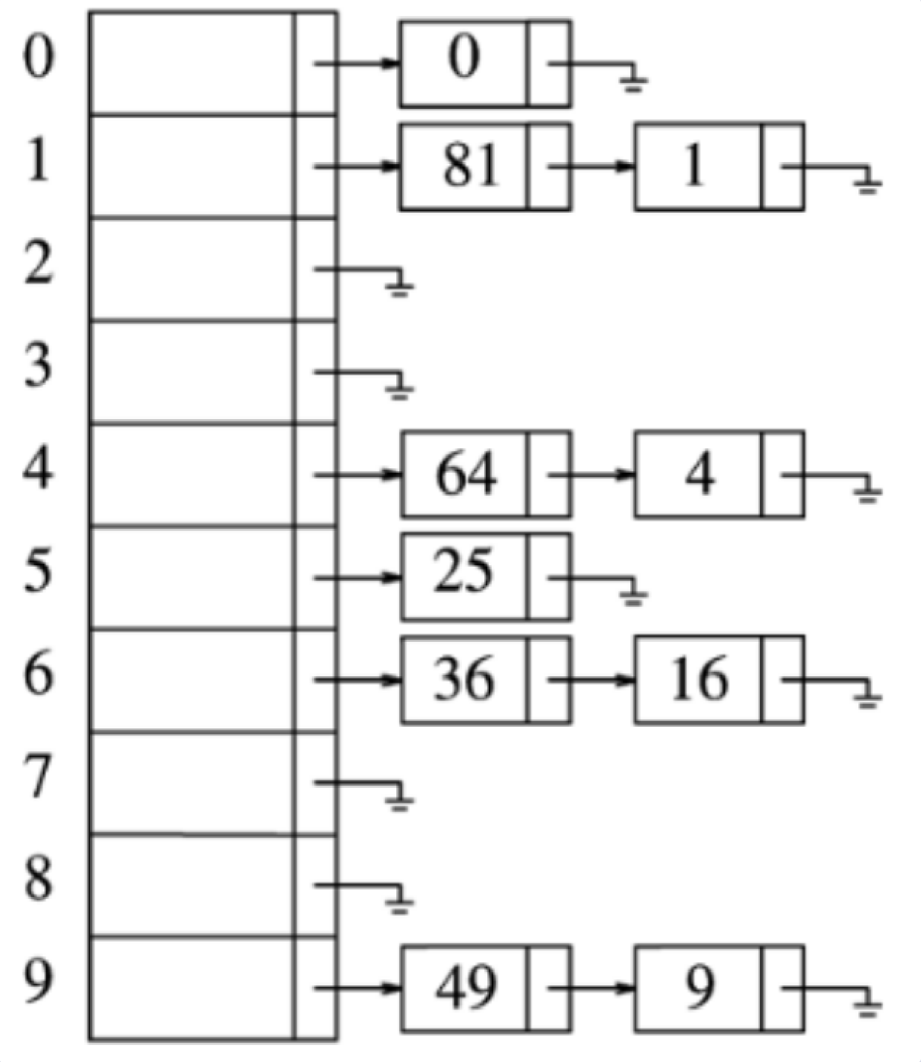
1. The basic idea of hash table is to approximate a giant array that is indexed by the key
2. A hash table is an array where the index of the data is computed (by the hash function) based on the key of the data

```
Index = hash(key) % table_size;
```

3. The situation when two keys are hashed into the same index is called a conflict or collision
4. A good hash function doesn't remove all conflicts. It statistically minimized the probability of collision across the key space

Separate Chaining

- Each table entry is a list - the hash table is physically an array of lists
- Multiple keys mapped to the same entry will be stored in the list



- **Assumptions**: `hash(k) = k % 10`
- Each `array[i]`, 0 ≤ i < size, is a list

Search with Separate Chaining

- To search for a key X we must do the following
  - `index = hash(X)`
  - Check list array `array[index]` to see if X is in it.
- Check the above graphic, using the assumptions for a value.

Inserting with Separate Chaining

- To insert key X we must do the following
  - `index = hash(X)`
  - Insert X into list `array[index]`
- Check the above graphic using the assumptions to insert a value

Delete with Separate Chaining

- To delete key X we must do the following
  - `index = hash(X)`
  - Insert X into list `array[index]`
- Check the above graphic using the assumptions to remove a value

Separate Chaining

- With separate chaining, the hash table is an array of containers
- Insertion, removal and deletion can be so quick because it only needs to do one calculation to find the location rather than comparing values
- The number of lists in the hash table needs to be roughly the same as the number of data items in the hash table
- The load factor (λ) of a hash table with separate chaining is the ratio of the number of elements in the table to the table size
- With separate chaining, the average list size is equal to λ!
- Typically, we want λ ≈ 1
- λ decides when to perform rehash (expanding the table)

Implementation

C++

```cpp
template <typename HashedObj>
class HashTable
{
public:
        explicit HashTable(int size=101);
        bool contains(HashedObj& x) const;

        void makeEmpty();
        bool insert(const HashedObj& x);
        bool insert(HashedObj&& x);
        bool remove(const HashedObj& x);

private:
        vector<list<HashedObj>> theLists;
        int currSize;

        void rehash();
        size_t myhash(const HashObj& x) const;
}
```

Hashed Object

```C++
template <typename key>
class Hash {
public:
        size_t operator()(const key& k) const;

};
template <>
class Hash<string>{ // Implementation
public:
        size_t operator()(const string& key){
                // ...
        }
};
```

Class Example

```C++
class Employee{
public:
        const string& getName() const{
                return name;
        }
        bool operator==(const Employee& rhs) const
        {
                return getName() == rhs.getName();
        }
        bool operator!=(const Employee& rhs)const {
                return !(*this == rhs);
        }
private:
        string name;
        double salary;
        int seniority;
        // Aditional private members
};
template<>
class hash<Employee>{
public:
        size_t operator()(const Employee& item){
                static hash<string> hf;
                return hf(item.getName());
        }
};
```

Separate Chaining

```C++
// Separate Chaining
size_t myhash(const HashedObj& x){
        static hash<HashedObj> hf;
        return hf(x) % theList.size();
}

// Separate Chaining Cont'd
// More Function Definitions
void makeEmpty(){
        for(auto& theList: theList){
                theList.clear();
        }

}

bool contains(const HashedObj& x) const{
        auto & whichList = theList[myhash(x)];
        return find(begin(whichList), end(whichList) != end(whichList));
}

bool remove(const HashObj& x){
        auto& whichList = theList[myhash(x)];
        auto itr = find(begin(whichList), end(whichList), x);

        if(itr == end(whichList)){
                return false;
        }
        whichList.erase(itr);
        --currentSize;
        return true;
}

bool insert(const HashedObj& x){
        auto& whichList = theList[myhash(x)];
        if(find(begin(whichList), end(whichList), x) != end(whichList)){
                return false;
        }
        whichList.push_back(x);

        // rehash...
        if(++currentsize > theList.size()){
                rehash();
        }
        return true;
}
```

Hash Tables without Chaining

- Try to avoid buckets with separate list - no list, just an array of elements
- Still need to result conflicts - use Probing Hash Tables
  - If collision occurs, try another cell in the hash table.
  - More formally,, try cells $h_0(x), h_1(x), h_2(x), h_3(x), \ldots$ in succession until a free cell is found.
  - $h_i(x) = hash(x) + f(i)$
  - AND $f(0) = 0$

Linear Probing

- `f(i) = i`
  - Try `hash(x), hash(x) + 1, hash(x) + 2, ...`

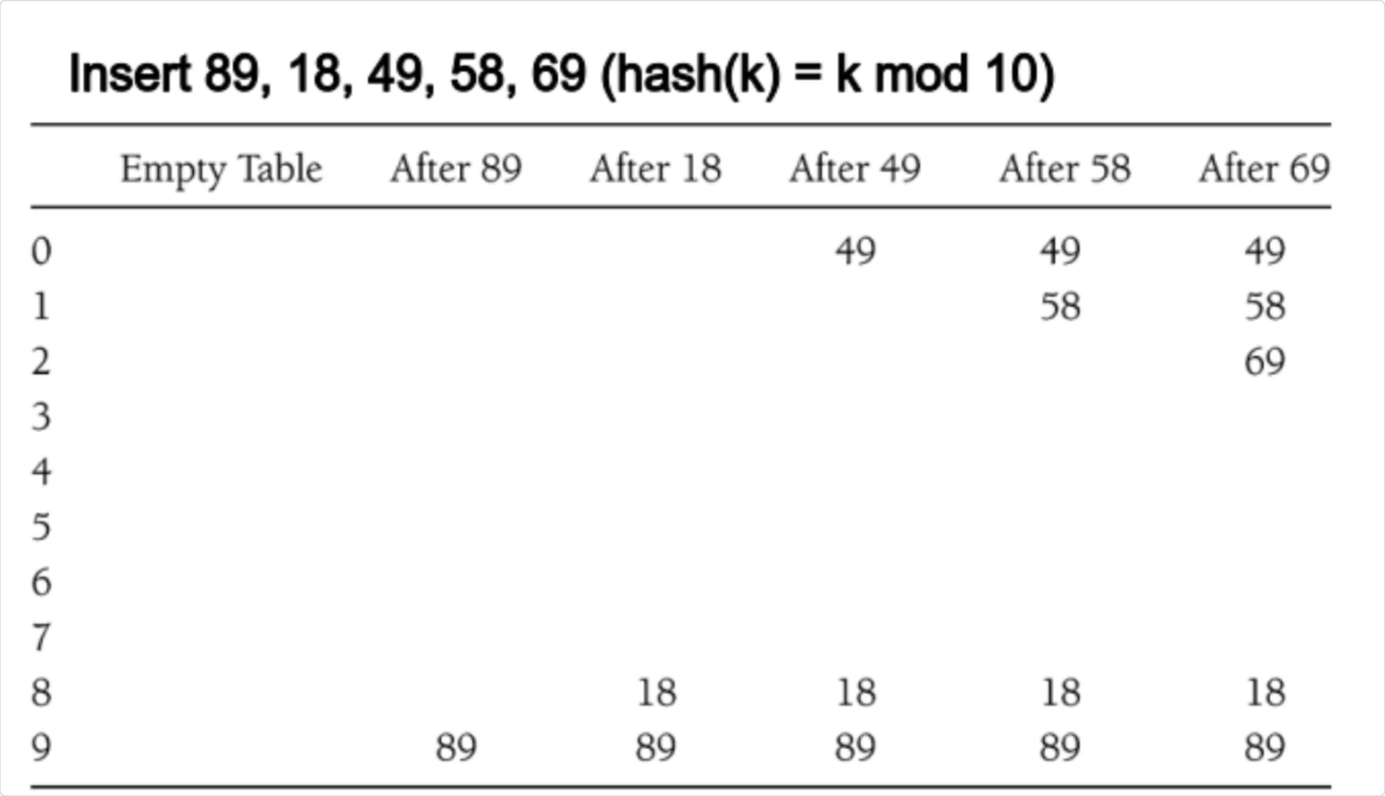
Insert (assume no duplicate keys)

1. `Index = hash(key) % table_size;`
2. If `table[index]` is empty, put informations (key and others) in entry `table[index]`

3. If `table[index]` is not empty then, `index++; index = index % table_size; goto 2`

Search (key)

1. `Index = hash(key) % table_size;`
2. If (`table[index]` is empty) return -1 (not found)
3. `Else if (table[index].key == key) return index;`
4. `index++; index = index % table_size; goto 2;`



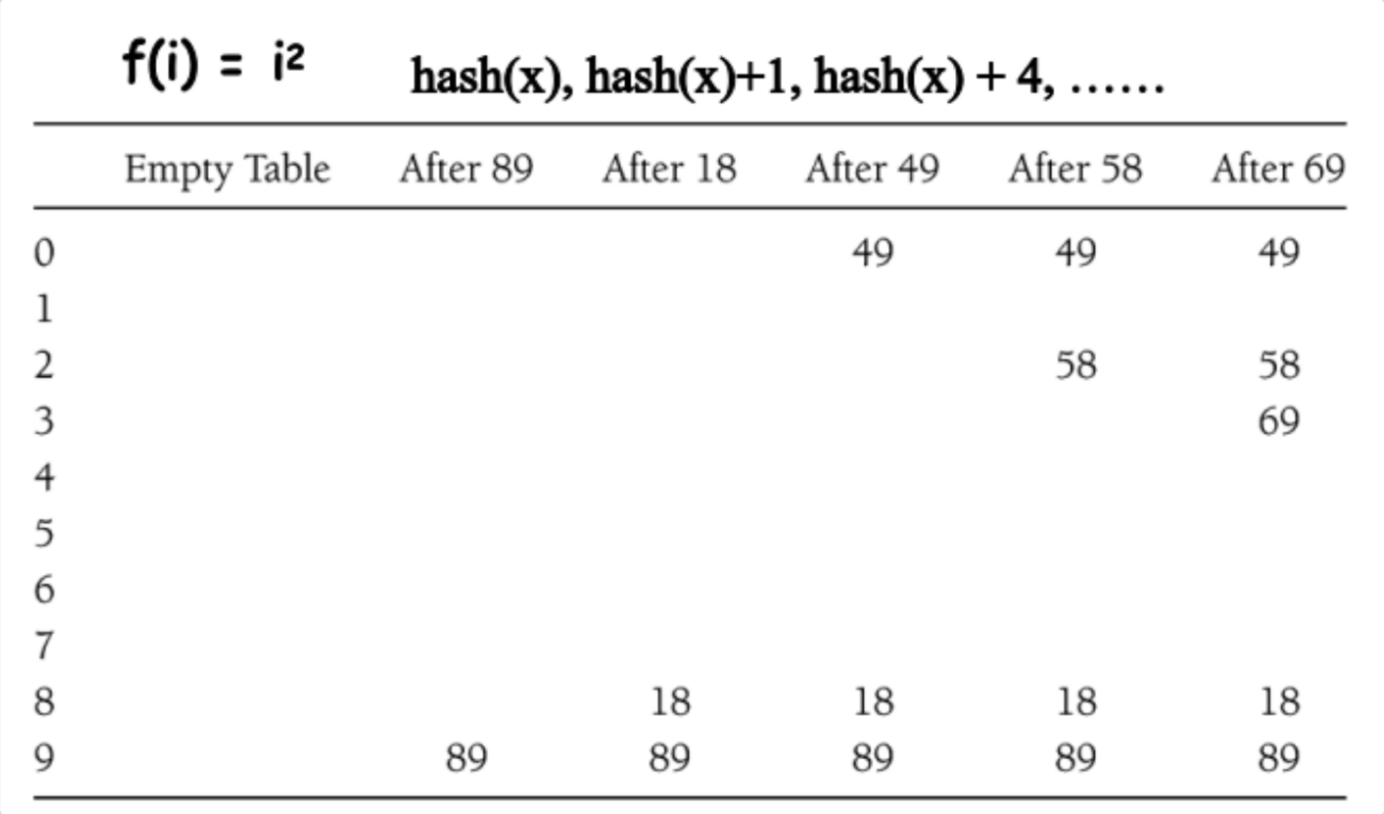
Insert 89, 18, 49, 58, 69 (hash(k) = k mod 10)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Delete

- Can be tricky, must maintain the consistency of the hash table, consider the number 89 in the table above.
- What is the simplest deletion strategy you can think of??

Quadratic Probing



f(i) = i^2 hash(x), hash(x)+1, hash(x) + 4, ……

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

```C++
// Nested within Hash Table class
enum EntryType{
        ACTIVE,
        EMPTY,
        DELETED
};

struct HashEntry{
        HashedObj element;
        EntryType info;
        HashedEntry(const HashedObj& e = HashedObj{}, EntryType ≠ EMPTY)
                : element{e}, info{|}{}
        HashEntry(HashedObj&& e, EntryType ≠ EMPTY)
                : element{std::move(e)}, info{|}{}
};
```

Double Hashing

Sorting

Comparison Based

Comparison based sorting: sorting based on the comparison of two items
In place sorting

- Sorting of data structure does not require any external data structure for sorting the intermediate steps
  External sorting
- Sorting of records not present in memory
  Stable sorting
- If the same element is present multiple times, then they retain the original positions

Stable
input- 2, 3, *1*, 15, 11, 23, **1**
output- *1*, **1**, 2, 3, 11, 15, 23

Not Stable
input- 2, 3, *1*, 15, 11, 23, **1**
output- **1**, *1*, 2, 3, 11, 15, 23

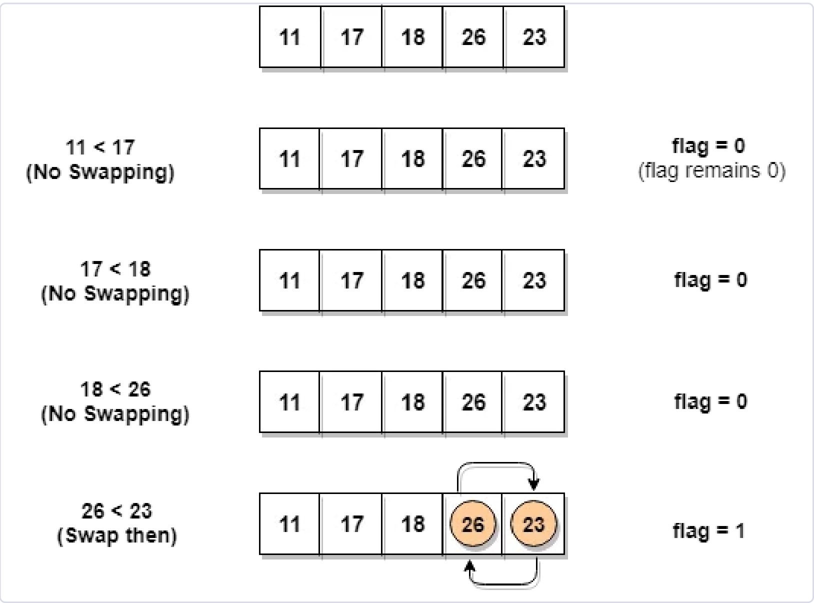
Some Sorting Algorithms

Simple sorting algorithms, performing only adjacent exchanges. Bubble sort and insertion sort are examples of this.

Bubble Sort

- Simple and uncomplicated
- compare to neighbor, swap if x is greater than y

Step	View
Step 1	2 3 1 15
Step 2	2 1 3 15
Step 3	1 2 3 15
Step 4	1 2 3 5



enhanced-bubble-sort.webp

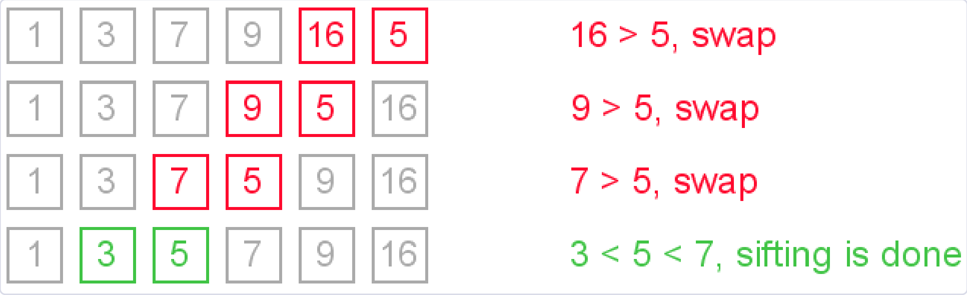
```C++
// bubble sort
int i, j;
for (i = 0; i < n - 1; i++)

        // Last i elements are already
        // in place
        for (j = 0; j < n - i - 1; j++)
                if (arr[j] > arr[j + 1])
                        swap(arr[j], arr[j + 1]);
```

Insertion Sort

- insert one element at a time
- If reversely sorted you will need to swap every item
- $O(N^2)$ Time Complexity
- $O(N)$ Best Time Complexity
- Good for if data is almost sorted

Step	View
Step 1	8 34 64 51 32 21
Step 2	8 34 64 51 32 21
Step 3	8 32 34 51 64 21
Step 4	8 21 32 34 51 64



```C++
// insertion sort
int i, key, j;
```

```
for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of their
        // current position
        while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
        }
        arr[j + 1] = key;
}
```

Shell Sort

- A sorting algorithm that allows for comparison of not adjacent items
- `h-sort` all elements spaced `h` apart are sorted
- Performing h-sort using insertion sort, the items compared are not longer adjacent - potential for improvement

C++

```
for (int gap = n/2; gap > 0; gap /= 2)
{
        // Do a gapped insertion sort for this gap size.
        // The first gap elements a[0..gap-1] are already in gapped order
        // keep adding one more element until the entire array is
        // gap sorted
        for (int i = gap; i < n; i += 1)
        {
                // add a[i] to the elements that have been gap sorted
                // save a[i] in temp and make a hole at position i
                int temp = arr[i];

                // shift earlier gap-sorted elements up until the correct
                // location for a[i] is found
                int j;
                for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                        arr[j] = arr[j - gap];

                //  put temp (the original a[i]) in its correct location
                arr[j] = temp;
        }
}
return 0;
```

Sorting 2

- Any comparison based sorting requires Ω(NlogN) comparisons

A General Lower Bound For Sorting

- The root represents the set of all possible orderings: when the sorting algorithm is given a array to sort, any possible ordering is possible!
- Each node performs one comparison, which patricians its set of orderings into two sets based on the comparison results. The two children are in each of its set, respectively.
- The algorithm needs to perform enough comparison to get a set that contains one ordering (the sorting result).
- **Each comparison based sorting algorithm can be represented by a binary decision tree.**
- The worst case is the largest set of comparisons possible needed.
  - This is the height of the decision tree
- Different algorithms differ in items selected for comparison at each node.
- The most efficient algorithm is the one with the smallest height.
- We know the number of leaves in the tree
  - The number of possible sorted orders of N items, let us denote it as X
- To get the minimum tree height of the decision tree, let us decide the number of leaves on the tree
  - Number of leaves = number of orderings N numbers
  - $N! = N * (N-1) * (N-2) * ... 2 * 1$
- *For a binary tree to have N! leaves, the tree is at least...*
  - $log(N!) = Log(N) + log(N-1) + ... + 1$
  - ≥ $log(N) + log(N-1) + ... + Log(\frac{N}{2})$
  - ≥ $log(\frac{N}{2}) + log(\frac{N}{2}) + ... + Log(\frac{N}{2})$
  - ≥ $\frac{N}{2} * Log(\frac{N}{2}) = \Omega(NlogN)$
- *No Comparison Based Sorting can be better than N(Log(N))*
- Heapsort, Mergesort, and Quicksort - *N(Log(N))*

Heap Sort

- Build binary minheap of N elements
  - O(N)
- The perform N deletemin operations
  - Log(N) time per deletemin
- *N(Log(N))*
- Requires another array to store results
- To eliminate this requirement
  - using heap to store sorted elements
  - using maxheap instead

```C++
void heapify(int arr[], int N, int i)
{

    // Initialize largest as root
    int largest = i;

    // left = 2*i + 1
    int l = 2 * i + 1;

    // right = 2*i + 2
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest
    // so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int N)
{

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element
    // from heap
    for (int i = N - 1; i > 0; i--) {

        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// A utility function to print array of size n
void printArray(int arr[], int N)
{
    for (int i = 0; i < N; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}
```

Merge Sort

- Divide N values to be sorted into two halves

- Recursively sort each half using merge sort
  - Base case N = 1, no sorting required
- Merge two halves into one list
- Keep a counter for each list starting at the start of each list
- Compare the two values indexed by the counters, output the smaller value and increment the counter
- when one list is processed output all items in the other list
  - 4, 1, 9, 4
  - 2, 5, 6, 8
  - Compare 4 and 2 and output 2

Complexity

- T(N) complexity when size N
- Merge O(N)
- Complexity O(NLogN)

Quick Sort

- Fastest sorting algorithm in practice
  - *Caveat*: not always stable
  - Can do it as a stable sort
- Average Complexity: $O(NLog(N))$
- Worst Complexity: $O(N^2)$
  - Rare
- Can vary in space complexity

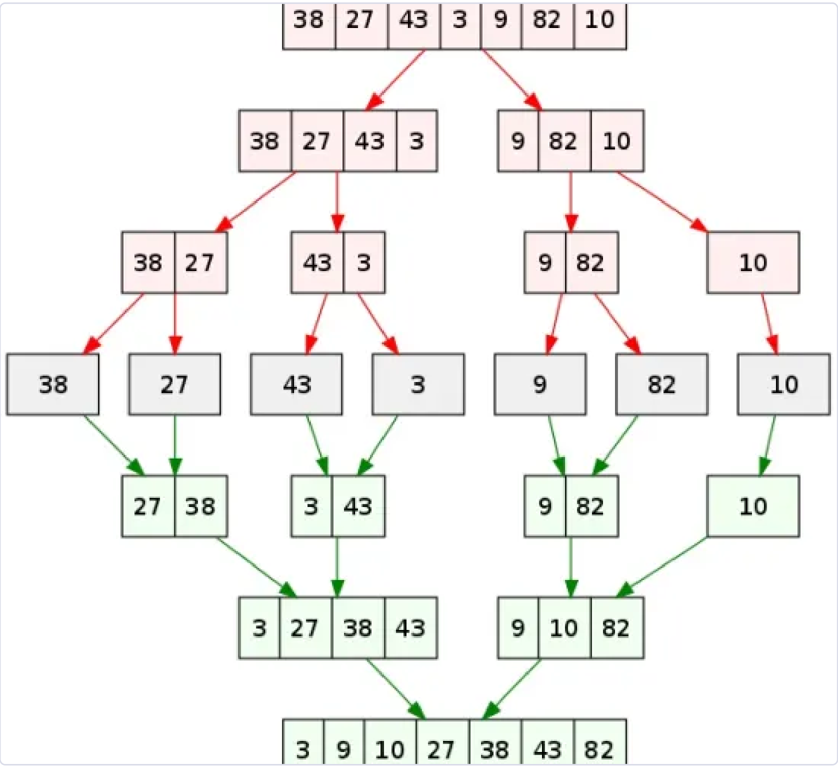
Sorting 3 For More....

Sorting 3

Quick Sort

Given array S to be sorted

- If size of S < 1 then done;
- Pick any element v in S as the pivot
- Partition S-{v} (remaining elements in S) into two groups
- S1 = {all elements in S-{v} that are smaller than v}
- S2 = {all elements in S-{v} that are larger than v}
- Return {quicksort(S1) followed by v followed by quicksort(S2) }
- **Trick lies in handling the partitioning (step 3).**
- Picking a good pivot
- Efficiently partitioning in-place



Difference-Between-Quicksort-and-Merge-Sort_Figure-1.webp

Picking the Pivot

- Partition takes O(N) time.
- **Middle**
  - T(N) = 2 T(N/2) + N, T(N) = O(N log N) – same as the merge sort
- **Min/Max**
  - T(N) = T(N-1) + N, same as insertion sort, T(N) = O(??)
- **Strategy 1: Pick the first element in S**
  - works *only* if input is random
  - Quicksort is recursive, so sub-problems could be sorted
- **Strategy 2: Pick the pivot randomly**
  - Expensive operation
  - Usually works well

    - works well with mostly sorted
- **Strategy 3: Median-of-three Partitioning**
    - Ideally, the pivot should be the *median* of input array S
    - divide the input into two almost equal partitions
    - Pivot = median of the left-most, right-most and center element
    - Solves the problem of sorted input

Example

• Example: Median-of-three Partitioning
– Let input S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}
– left=0 and S[left] = 6
– right=9 and S[right] = 8
– center = (left+right)/2 = 4 and S[center] = 0
– Pivot
• = Median of S[left], S[right], and S[center]
• = median of 6, 8, and 0
• = S[left] = 6

Partitioning Algorithm

```C++
While (i < j)
        Move i to the right till we find a number greater than pivot
        Move j to the left till we find a number smaller than pivot
        If (i < j) swap(S[i], S[j])
//(The effect is to push larger elements to the right and smaller elements to the left)
Swap the pivot with S[i]
```

When Dealing With Small Arrays

- Insertion sort in the best under ten items
- Quick sort is recursive so it may take a long time sorting small arrays
- Hybrid (Switch between insertion and quick sort depending on array size)

Code Examples

```C++
template <typename Comparable>
void quicksort( vector<Comparable> & a, int left, int right )
{
        if( left + 10 <= right )
        {
                const Comparable & pivot = median3( a, left, right );
                // Begin partitioning
                int i = left, j = right - 1;
                for( ; ; ) {
                        while( a[ ++i ] < pivot ) { }
                        while( pivot < a[ --j ] ) { }
                        if( i < j )
                                std::swap( a[ i ], a[ j ] );
                        else
                                break;
        }
        std::swap( a[ i ], a[ right - 1 ] ); // Restore pivot
        quicksort( a, left, i - 1 ); // Sort small elements
        quicksort( a, i + 1, right ); // Sort large elements
        }
        else // Do an insertion sort on the subarray
                insertionSort( a, left, right )
}
```

Runtime

- Worst Case: $O(n2)$
- Best Case: $O(n * logn)$
- Average Case: $O(n * logn)$

Linear Time Sorts

- Comparison based sorting requires Ω(NlogN) time in the worst case
- Linear sorting applies to special cases

Bucket Sort

- Input: $A_1, A_2, , \dots A_N$ of positive integers smaller than M
- Output: Sorted list of integers
- Algorithm:
  - Keep an array with counts of each occurrence of the data
  - Set each count to 0
  - Need to know your data range
- Complexity? O(N+M)
- What if the data has other fields than the key?
  - Modify the count array to be an array of buckets
  - Is the sorting stable in this case?
- good if M is the same order as N
- limitation: needs an O(M) space so M cannot be very large

Radix Sort

- What if we want to use the idea of Bucket sort to sort based on social security number
- We can use a count array of size 1000 and perform bucket sort 3 times to sort based on the social security number
  - Use bucket sort to sort from the last sig bits to the most sig bits
  - Sort based on the last three digits in the first pass
  - sort based on the middle three digits in the second pass
  - sort based on the first three digits in the third pass
- Because bucket sort is stable
-

Graphs Algorithms

- A GRAPH G = (V, E)
  - V : set of vertices (nodes)
  - E : set of edges (links)
- Complete graph
  - There is an edge between every pair of vertices
  - Tow kinds of graph
    - undirected
    - directed (digraph)
- Undirected graph
  - E consists of sets of two elements each: Edge (u,v) is the same as {v,u}

Terminology

- Adjacency
  - Vertex w is adjacent to v if and only if (v, w) is in E
- Weight
  - A const parameter associated with each edge
- Path
  - Sequence of vertices where there is an edge for each pair of consecutive vertices
- Length of path
  - Number of edges along path
  - Length of a path of n vertices is n-1

Cycles

- A path is simple if all its vertices are distinct (first and last may be equal)
- A cycle path is a path $w_1, w_2, ... w_n = w_1$
  - A cycle is simple if the path is simple
  - It has a loop if a node repeats
- An undirected graph is connected if
  - Each pair of vertices u,v there is a path that starts at u and ends at v
- A digraph H that satisfies the above condition is strongly connected
- Otherwise if H is not strongly connected, but the undirected graph G with the same set of vertices and edges is connected, H is said to be weakly connected
- BFS Finds all nodes

Representation of Graphs

- To store graph information, we need to store the connectivity (link) information.
- Two popular representations
  - Adjacency matrix
    - Use a 2d array to store the connectivity: `A[u][v]` is true if there is an edge from u to v, false otherwise
  - \

Adjacency matrix

- `A[u][v]` is true if there is an edge from u to v
- False otherwise

- For a weighted graph, assign weight instead of t or f
- O(1) time to decide whether (u,v) is an edge
- `A[N][N]` - O(|V|^2) space
  - Wasteful if the graph is sparse, not too many edges
- *Adjacency list*
  - Each node maintains a list of neighbors
  - Need to go through the list to decide if u, v is an edge

Topological Sorting

- Let G be a directed acyclic graph (DAG)
- an ordering the vertices of G such that if there is an edge from v_i to v_j appears after v_i
- In a DAG, there must be a vertex with no incoming edges
- Have each vertex maintain its indegree, indegree of v = number of edges (u, v)
- Repeat
  - Find a vertex of current indegree 0
  - assign it a rank
  - reduce the indegrees of the vertices in its adjacency list

```C++
void Graph()::topsort{
        for(int counter = 0; counter < NUM_VERT; counter++){
                Vertex v - findNewVertexOfIndegreeZero();
                if(v == NOT_A_VERTEX){
                        throw CycleFoundException()
                }
                v.topNum = counter;
                for each Vertex w adjacent to v
                        w.indegree--;
        }
}

void Graph::topsort(){
        Queue<Vertex> q;
        int counter = 0;
        q.makeEmpty();
        for each Vertex v
                if( v.indegree == 0){
                        q.dequeue(v);
                }
        while(!q.isEmpty){
                Vertex v = q.dequeue();
                v.topNum ++counter;
        for each Vertex w adjacent to v
                if(--w.indegree == 0)
                        // Etc
        }
}
```

Single Source Shortest Path Problem

- Unweighted shortest paths
  - BFS
- Weighted
  - Dijkstras algorithm
  1. Pick one node with the shortest distance

2. Update the distance for all nodes that are adjacent to the picket node
3. repeat till all nodes are picked

Chapter 4 to 7 Review

Trees

- A tree is a connected graph with no cycle
- **Rooted tree**: a connected graph with no cycle and a particular node called a root
  - **Parent node, child node, sibling node, ancestors, decendants, leaf nodes**
  - Depth of vertex - number of edges from the node to the tree's root node. A root node will have a depth of 0.
  - Height of vertex - number of edges on the *longest path* from the node to a leaf. A leaf node will have a height of 0.

Insert (Recursive)

```C++
if (root == nullptr) {
        return new TreeNode(key);
    }

    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }

    return root;
```

Delete (Recursive)

```C++
if (root == nullptr) {
        return root;
    }

    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    } else {
        if (root->left == nullptr) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }

        TreeNode* temp = minValueNode(root->right);

        root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);
    }
```

Find Min (Recursive)

```C++
TreeNode* current = node;

    while (current->left != nullptr) {
        current = current->left;
    }

    return current;
```

A **binary tree** is a data structure in which each node has at most two children, referred to as the left child and the right child. Every node, except for the leaves, has exactly one parent. The left and right children of a node are distinct, and the difference in the number of nodes between the left and right subtrees of any node is at most one.

A **complete binary tree** is a special type of binary tree in which all levels are completely filled, except possibly for the last level, which is filled from left to right. In other words, all nodes are as left as possible in each level.

Pre-order Traversal:

1. Visit the root node.
2. Traverse the left subtree in pre-order.
3. Traverse the right subtree in pre-order.

In-order Traversal:

1. Traverse the left subtree in in-order.
2. Visit the root node.
3. Traverse the right subtree in in-order.

Post-order Traversal:

1. Traverse the left subtree in post-order.
2. Traverse the right subtree in post-order.
3. Visit the root node.

Level-order Traversal (Breadth-First Traversal):

1. Start at the root node.
2. Traverse each level of the tree from left to right.

Binary Search Tree

- There is an order relation ≤ defined for the vertices of B
- For any vertex v, and and descendant u of v.left u ≤ v
- For any vertex w, and and descendant u of w.right u ≤ w
- also known as a *totally ordered tree*

Operation	Description
Insert	Start at the root. Compare the key to be inserted with the key of the current node. If the key is less, move to the left subtree; if greater, move to the right subtree. Repeat this process until reaching a null position, then insert the new node with the given key at this position.
Delete	Search for the node to be deleted starting at the root. Identify different cases: if the node has no children (a leaf node), remove it; if the node has one child, replace the node with its child; if the node has two children, find the in-order successor (or predecessor), replace the node's key with the successor's (or predecessor's) key, and then recursively delete the successor (or predecessor).

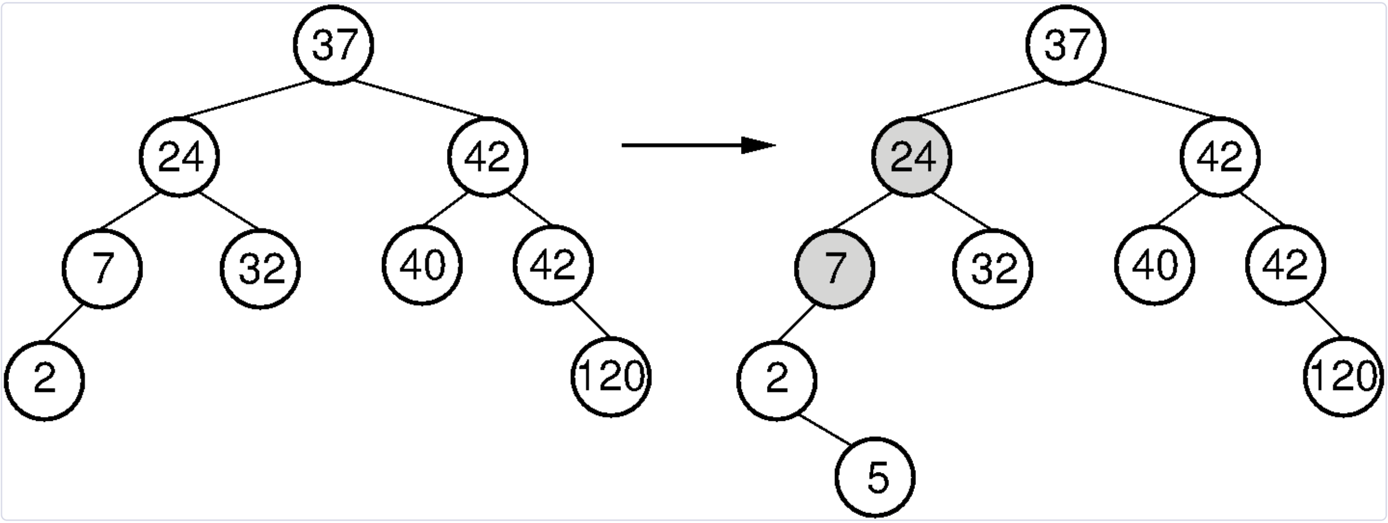
Operation	Description
Search	Start at the root. Compare the key to be searched with the key of the current node. If the key is equal to the current node's key, you have found the node. If the key is less, move to the left subtree; if greater, move to the right subtree. Repeat this process until you find the node with the matching key or reach a null position, indicating the key is not in the tree.

More info on binary search trees can be found in Trees 2 🌲

AVL Tree

An **AVL tree** is a binary search tree with the balance condition, for every node in the tree, height of left and right subtree can differ at most by 1. An avl tree is maintained by fixing the nodes that violate this condition after each insertion or deletion. Fixing is done through either single or double rotation.

Check from the parent of the newly inserted node upwards to the root to ensure that the AVL tree is not being violated by a newly inserted node.



B-tree

A **b-tree** is a m-ary search tree with the following balance restrictions

- Data items are stored at the leaves
- Non leaf nodes store up to `M-1` keys to guide the searching, keys are sorted within a node, Key i represents the smallest key in the subtree (i+1)
- The root can either be a lead, or have between 2 to M children
- All non-leaf nodes except the root have between `ceil(M/2)` and `M` children
- All leaves are stored at the same and have between `ceil(l/2)` and `l` data items for some `l`
- In other words, each node except for the root is at least half full

---

Hash Table

Hash table is a vector of lists, conflicts are resolved with the list. Multiple items can be stored in one list.

- If collision occurs try another cell to look
  - Try cells $h_0(x), h_1(x), \ldots$ in succession until a free cell is found
  - This is called **linear probing**
- A partially ordered tree (POT) is a tree T such that

  - There is an order relation ≤ defined for the vertices of T
  - For any vertex p and any child c of p, p ≤ c
- Partially ordered complete binary tree
  - Allowing vector representation of the tree

For more on hash tables view... Hash Table and Hash Table 2

Priority Queue

Each job within a computer takes turns using the cpu. If a job comes earlier than another job it needs to be executed earlier. If a job has been scheduled, it should not be scheduled a second time if there exists a job that hasn't been scheduled once. The queue is the answer.

Vector Representation

- sorting the tree in level order continuously
- may start from index 0 or index 1
  - Different starting affects the trees navigation

Heap operations

- DeleteMIN decreased the heap size by one
  - Move the last element to the root and percolate down from the root

Percolate Up (Heapify Up):

This operation is typically used during the insertion of a new element into the priority queue.

1. **Insert Element at the Bottom:**
   - Insert the new element at the bottom of the heap (last position).
2. **Compare with Parent and Swap:**
   - Compare the value of the new element with its parent.
   - If the value of the new element is greater (for a max-heap) or smaller (for a min-heap) than its parent, swap the element with its parent.
   - Repeat this process until the heap order property is restored.

Percolate Down (Heapify Down):

This operation is typically used during extraction or removal of the top element from the priority queue.

1. **Replace Top with Bottom:**
   - Replace the top element (the root) with the last element at the bottom of the heap.
2. **Compare with Children and Swap:**
   - Compare the value of the new root with its children.
   - If the value is smaller (for a max-heap) or greater (for a min-heap) than at least one of its children, swap the element with the smaller (for a max-heap) or larger (for a min-heap) child.
   - Repeat this process until the heap order property is restored.

Building the heap

- Inserting one by one resulting in O(N log N) heap building algorithms

- A better O(N) heap building algorithm builds head from bottom up by percolate elements backward (from N/2 to 1).