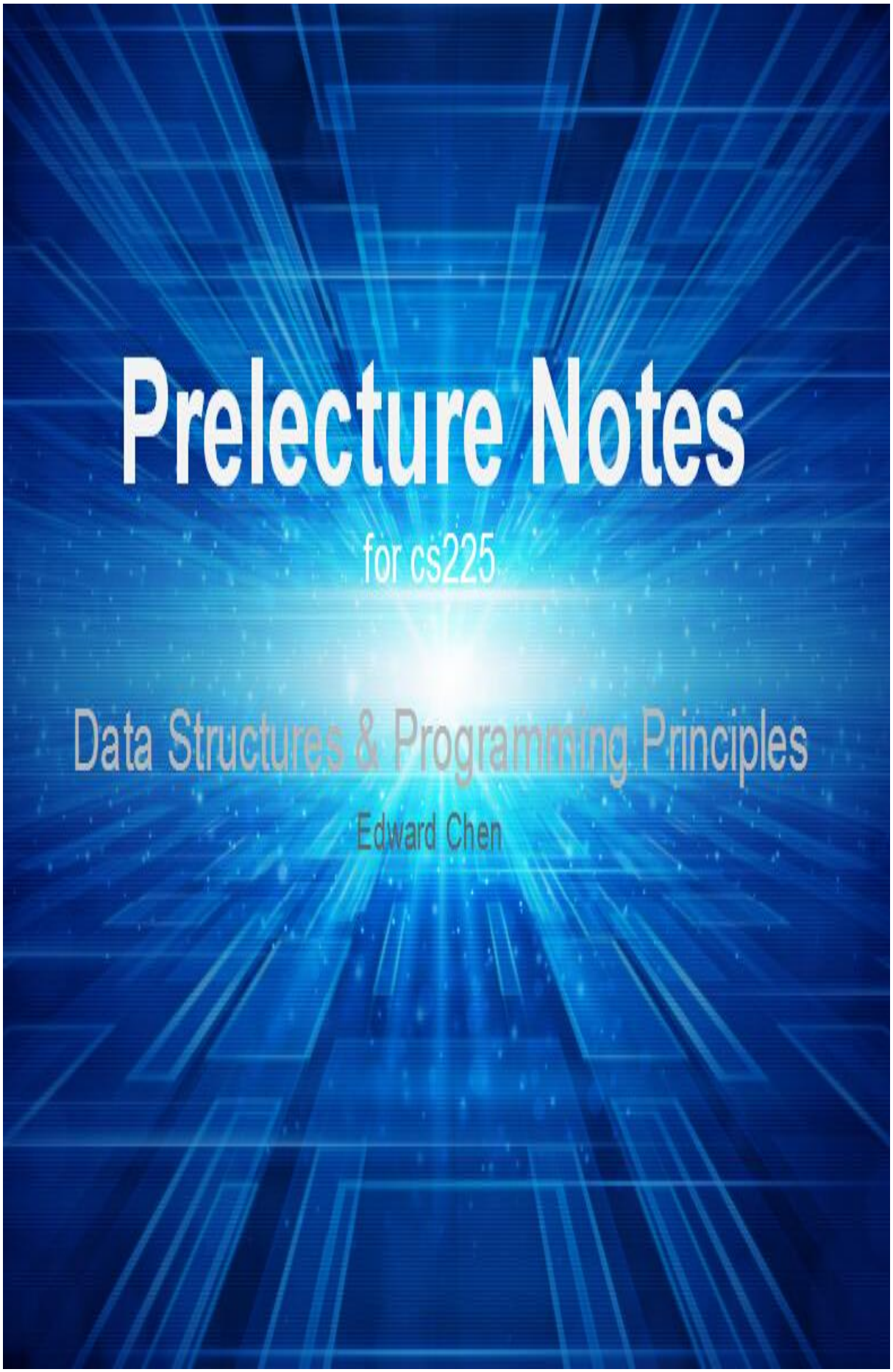

Prelecture Notes
for cs225
Data Structures & Programming Principles
Edward Chen

Prelecture Notes

Data Structures and Algorithm Principles in C++

By Edward Yawen Chen

If you come from Java, the first thing you need to know is that C++ separates declaration in head file (.h) and implementation in implementation file(.cpp), aka encapsulation.

```
#ifndef PERSON_H
#define PERSON_H
#include <string>
using namespace std;

class Person{
    public:
        Person(string PName,
        int PAge, bool MarrageStatus);
        string getName();
        int getAge();
        bool getMarried();
    private:
        string name;
        int age;
        bool married;
};
#endif
```

```
#include "person.h"

Person:: Person(string PName,
    int PAge, bool MarriageStatus)
{
    name = PName;
    age = PAge;
    married = MarriageStatus;
}

string Person:: getName(){
    return name;
}

int Person:: getAge(){
    return age;
}

bool Person:: getMarried(){
    return married;
}
```

In the head file, "#ifndef" and "#define" are called preprocessors. "#ifndef" checks if the identifier, PERSON_H in this case, has been defined. If not, define it now. The purpose of preprocessors in head file is to prevent double declaration of identifier. The "#include <string>" and "using namespace std;" are imported because string type is being used in the code. You don't need to worry about it now. One last note before getting into the class, at the end of the head file, you need to include "#endif" which tells the compiler that the class declaration ends here.

In the class of person, you see all the public functions are gathered into a key word "public" and all the private values/variables/functions should be inside the "private" scope. The most common syntax error in head file is missing the semi-colon after the ending curly brackets.

In the implementation file, since this file is used to implement the functions that declare in the person.h (head file), you need to include the name of the head file in the beginning of the file. The format to implement a function is as follow:

<return_type> <class_name>:: <finction_name>(<parameter/s>)

The double colon is called scope resolution operator.

Basically, these are all the syntax you need to know now in .h and .ccp files. You will explore more when you learn template implementation and pure virtual function.

Keyword *const*

In many codes, you will see soon, there is a keyword [const] which guarantees "something" from changing when you call functions or use variables. The "something" might refer to different meanings due to the use of [const].

1. *const* variables: when you have a fixed variable that you don't want its value being changed for the whole class, you can use *const* to force the variable from changing.

Example: int const x = 15
same as const int x = 15.

2. *const pointers*: (you will learn about pointer in next page)

A. prevent data from changing

const int *ptr

```
//A. prevent data from changing
int x = 5; int y = 10;
const int *p = &x;
cout << "pointer address before: " << p << endl; // addredd 0xffffccdb
cout << "pointed value before: " << *p << endl; // value 5
//*p becomes read only, you cant change the data.
// but once you change the address, the data will change by the address.
*p = 12; // this is not allowed
p = &y; // this is allowed
cout << "pointer address after: " << p<< endl; // addredd 0xffffeeab
cout << "pointed value after: " << *p << endl; // value 10
```

B. prevent address from changing

int * const ptr

```
//B. prevent address from changing
int x = 5; int y = 10;
int * const ptr = &x; //ptr becomes read only.
cout << "pointer address before: " << ptr << endl; // addredd 0xffffccde
cout << "pointer value before: " << *ptr << endl; // value 5
*ptr = 12; // ptr = &y is not allowed
cout << "pointer address after: " << ptr << endl; // addredd 0xffffccde
cout << "pointer value after: " << *ptr << endl; // value 12
```

3. *const functions*:

In implementation files, you will see some functions with this format:

<return_type><class>:: <name_of_the_function> (<parameter/s>) const

The const here implies that nothing will change the object by function call.

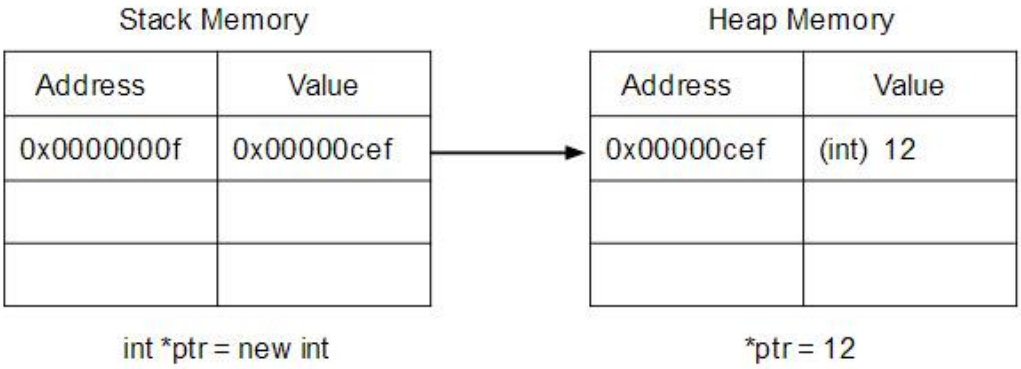
There are two points you need to keep in mind:

1. const functions only can call const functions.
2. const functions can always be called

Keyword ***new*** & ***pointer***

Variables and data are allocated to one of the two memories: stack memory and heap memory. Stack memory is a chunk of sequential memory addresses, like an array, which the system will erase memory blocks and set to available list whenever function calls end, or data out of scope. Compare to stack memory, heap memory is much larger and it is used for dynamic allocation by using operator *new*. And dynamic memory must be accessed through a pointer. Whenever we talk about dynamic allocation, we mean allocating memory in heap using operator *new*. After use of heap memory, you need to free the memory manually (destructor).

Now, let's explore a key concept in C++: *pointer*. Pointer would be unfamiliar to those who come from Java. Basically, pointer is just a variable that stores an address rather than a value. It contains an address of other memory block.



In the figure above, pointer variable *ptr* is declared in stack memory. And the operator *new* allocates an address in heap memory and assigns the address to *ptr*.

Two more operators you need to know in order to manipulate pointers

1. * : dereference pointer to its value, also use in pointer declaration
2. & : reference operator which gives an address of a variable

Example:

```
char * ptr;              // declare a pointer variable called ptr
char *ptr, ptr2;         // ptr is a pointer but ptr2 is just a char variable
ptr2 = 'x';              // assign a char 'x' to ptr2
ptr = &ptr2;             //&ptr2 will give the memory address storing the char 'x'
// now, ptr is pointing to ptr2 which means ptr stores the address of ptr2
cout << ptr2;            // print out character x
cout << *ptr;            // also print out x. here ptr is dereferenced by the operator *.
cout << &ptr2;           // print out an address
cout << ptr;             // print out the same address as &ptr2 since ptr is pointing to
                            &ptr2
int *pointer = new int;    // using operator new to allocate a memory block in heap
*pointer = 100-50;         // dereference pointer and assign 50 to it
cout << "pointer value: " << *pointer <<endl;    // print out "pointer value: 50"
```

We have mentioned that memory in heap has to be erased manually. In C++, we use a keyword "*delete*" to free the memory in heap and put the address into the list of available memory.

```
int * ptr = new int;
*ptr = 33;
delete ptr;
ptr = NULL;            //set the pointer to NULL. Not necessary.
```

Do not delete a pointer twice. You will get undefined behavior in the second time you delete a deleted pointer; it might lead to a system crash.

Suppose you have int x = 10 in your stack memory, when you assign ptr = &x, the address that is stored in ptr now has changed to stack memory address. Remember, delete can **only** use to free memory in heap.

Parameter Passing

1. *Pass-by-value*:
   - Cannot change the value of the parameter (read-only)
   - When you working on some large objects like pictures which contains thousand pixels, copy is very expensive (on running time)

   Example:

   Assume the addTen function has defined to add ten to the parameter integer and return the new value

```
int n = 5;
addTen ( n) ;
cout << n;    // print out 5.
```

2. *Pass-by-pointer*:
   - Copying only change the pointer to point to other object —> very fast
   - However, you have to manipulate pointers and must check for null pointers

   Example:

```
int x = 5;
int * ptr = &x ;
int addTen( ptr);
cout << *ptr;          // print out 15;
```

3. *Pass-by-reference*:

   - The first choice in C++ because it is fast for copying and no needs to deal with pointers

   Example:

```
int n = 5;
addTen( n );
cout << n;    // print out 15;
```

Big Three

1. copy constructor
2. destructor
3. operator =

If you have any reason to implement one of these three, implement them all

1. *Constructor*:
   - Can have more than one constructor
   - System will provided a default constructor by assigning all the variables to default values, if no constructor is implemented in the class
   - A copy constructor accepts a same type of object as a parameter and copies the data over

2. *Destructor*:
   - Wherever dynamic memory is allocated in your class, you need a destructor to free the memories after the object goes out of scope.
   - Destructor is never being called but the system

3. Operator = :
   - Use to copy variables from object to object (same type of object)

* Example code will not be provided here. You will see a lot in labs and mps

Inheritance

You should have learned inheritance before. So, I will just quickly walk through it.

1. Inheritance involves base class and derived class.

2. The derived class inherits the public and protected members from the base class

3. The private part and big three will not be inherited

4. Syntax:

```
class <derived_class_name>: public<base_class_name>{
<statement>
    ……
}
```

Virtual Members

A virtual member is a member function that can be redefined in a derived class.

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

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

Both of the derived classes inherit the protected variables and public functions from base class. The virtual keyword in the base class indicates that the derived class can override the function < area >. On the right hand above, we have three base class pointers point to the derived class objects. When the area function is called, it will call the function corresponding to the derived class. Without virtual, zero will be printed out in all three cases. Specifically, since ppoly1 points to the rectangle object, when it calls the area function, the one in Rectangle class is executed and the result is 20. And the areas of ppoly2 and ppoly3 are 10 and 0 respectively. For more information, check here.

Note:

Suppose we have two objects

Case 1: base x;	case 2: base * x;
derived y;	derived * y;
x = y; // allowed	x = y; //allowed
y = x; // not allowed	y = x; // not allowed

In both cases, assigning a derived object/object pointer to a base object/object pointer will result in compiler error because base classes usually have private section that does not inherit to the derived classes. Think this way, base classes have everything derived classes have but not the other way around.

Remarks:

1. A derived class is not required to override an existing implementation of an inherited virtual method
2. Constructors cannot be virtual
3. Destructor should be virtual
4. Virtual method return type cannot be overwritten
5. Virtual functions need to implement in base class. Otherwise, the class becomes an abstract base class which contains at least one pure virtual function
6. Pure virtual function: virtual int area ( ) = 0;
7. Pure virtual functions have to be override in derived classes
8. You cannot declare an object of the class but a pointer if a class contains all pure virtual functions.

Template

Many times, you may have to write separate classes with same operations for different types of data. In order to improve efficiency in this case, function templates is the best choice.

```
template<class T>
class pair{
    public:
        pair(T x, T y);
        T max();
    private:
    T i, j;

}

template<class T>
pair<T>::pair(T x, T y){
    i = x;
    j = y;
}

template<class T>
T pair<T>::max(){
    return (i > j ? i: j);
}
```

```
int main()
{
   int a = 3; int b =6;
   double i = 1.6;
   double j = 6.6;
   int sum = add<int>(a,b);
   double Dsum = add<double>(i,j);
}


template<class T>
T add(T x, T y){
    T sum;
    sum = x+y;
    return sum;
}
```

On the right side, a function template is built for any type that is capable of addition, such as primitive type: int, double, and float etc. T just a variable name for a type, you can name it whatever you prefer to.

On the left figure, we have a class template, same logic and syntax as function template. However, when you implement your functions, you need the “template < class T >” sentence for every single function. Following is a sample of declaring an object in the template class pair:

//declare an object and use constructor to initialize the private variable;

pair < char > character (‘y’, ‘k’);

//dynamic allocation of an array

pair < int > * p = new pair < int > [6];

At this point, we have done enough C++ syntax for this course. For more information, click here. The following content in this note is all about data structure which is one of the most important parts in computer science. Like I said, I will not go deep here. This is just a note for the class.

Linked List

Linked List is a list of nodes that each node contains a data and a pointer (can be more than one data and pointers). Each Linked List might have a head pointer (no data) pointing the first node. A head pointer in an empty Linked List points to NULL and a last node in a non-empty Linked List also points to NULL.(some Linked List implementations include one or more sentinel nodes at the beginning and ending to make some algorithm simpler. These nodes do not hold any data. They act like indicators indicating the first/last node in the linked list. With no doubt, head pointer is a sentinel node)

Let me introduce you a new memory model, using "sturct" to create a new block in memory. For example:

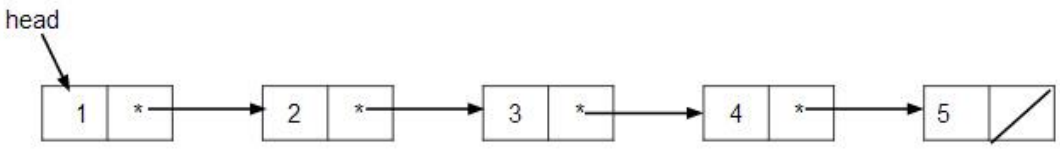
```
template < class LIT>
struct ListNode{
    LIT value;
    ListNode * next;
    ListNode(LIT data): value(data), next(NULL){}

};
```

ListNode can have more than one constructors based on your implementation. Pointer next contains the value of a memory address. You can imagine the ListNode appears in memory as following:

InsertInFront: create a new ListNode with the given data and make its next point to the node as head is pointing to and reset the head pointer

```
//assume the pointer head has implemented
void insertInFront(LIT newData){
    ListNode *temp = new ListNode<LIT>(newData);
    temp->next = head;
    head = temp;
}
```

If you call *insertInFront* for type int: 5, 4,3,2,1 into an empty linked list sequentially, you will get as below:



We are going to implement the function *insertInkth* which insert a data in a specific location, such that if the location is 1, insert in the front. The idea is to:

- Loop through the list and stop on the previous node of the given location.
- Create a new node with the data and insert in the list.

This algorithm takes O (n) in worst case if the location is at the end of the list. The following code is a basic implementation of the function. You may use recursion or create a helper function to find the location and then do the insertion.

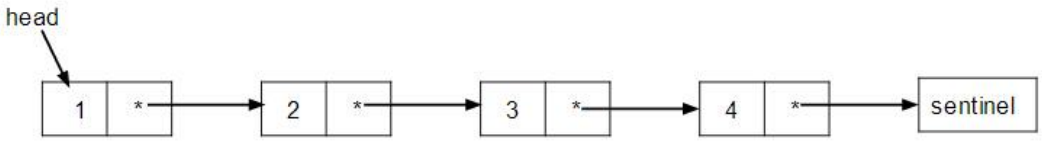
```
//insert new node in kth position
void insertInkth(int loc, LIT data){
    //size is the length of the list
    if(loc < 1 || loc > size+1) return;
    if(loc == 1)
        insertInFront(data);
    else{
        ListNode* curr = head;
        while(loc > 2){ // stop at the previous position
            curr = curr->next;
            loc--;
        }
        ListNode * temp = new LinkNode < LIT >(data);
        temp->next = curr->next;
        curr->next = temp;
        }
}
```

*If you do not fully understand what the code does, I recommend you to draw out an example and follow the code line by line

To remove a given node, if you loop from beginning to get the previous node of the given one and then do the removal, this takes O(n) if the give node is the last node. There is a better way to remove a given node in constant time:

- Copy the data from the next node to the given node
- Remove the next node

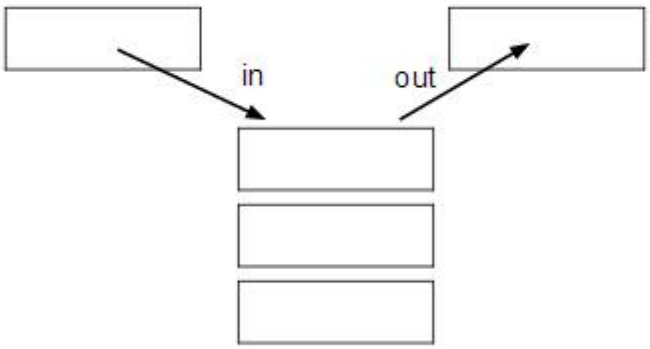
in this case, putting a sentinel node at the end would make this implementation easier if the given node is the last node in the list.




```
void removeCurrentNode(ListNode * curr){
    ListNode * temp = curr->next;
    curr->value = temp->value;
    curr->next = temp->next;
    delete temp;
}
```

Stack

Stack is another abstract data type that *push* (insert) and *pop* (delete) can only be performed in one position or only the top data is accessible. *Push* inserts data at the top of the stack and *pop* deletes the most recent inserted data from the stack. Stack can be implementing by a linked list or an array. It is known as a Last In First Out data structure.



Linked List Implementation:

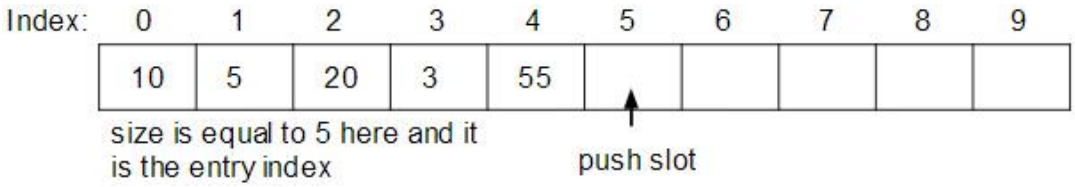
```
template < class SIT >
class Stack{
    public:
        Stack();    //constructor
        ~Stack();   //destructor
        //copy constructor
        Stack(const Stack< SIT > & other);
        //operator equal
        void operator= (const Stack< SIT > & source)
        void push(const SIT & x);
        SIT pop();
        bool isEmpty() const;
        int getSize();
    private:
        struct stackNode{
            SIT data;
            stackNode * next;
        };
        stackNode *top;
        int size;
};
```


```
template<class SIT>
void Stack<SIT>::push(const SIT & x){
    stackNode * temp = new stackNode(x);
    temp->next = top;
    top = temp;
    size++;
}
```

```
template<class SIT>
SIT Stack<SIT>::pop(){
    //assuming the stack is not empty
    StackNode * temp = top->next;
    SIT val = top->data;
    delete top;
    top = temp;
    size--;
    return val;
}
```

Array Based Implementation:

```
template <class SIT>
class Stack{
    public:
        Stack();    //constructor
        ~Stack();   //destructor
        //copy constructor
        Stack(const Stack< SIT > & other);
        //operator equal
        void operator= (const Stack< SIT > & source);
        void push(const SIT & x);
        SIT pop();
        bool isEmpty() const;
        int getSize();
    private:
        int capacity;   //capacity of the array
        int size;   //number of items in the array;
        SIT * array;
};
```



```
template<class SIT>
Stack<SIT>::Stack(){
    capacity = 4; //any reasonable number
    size = 0;
    array = new SIT[capacity];
}

template<class SIT>
SIT Stack<SIT>::pop(){
    SIT result = array[size-1];
    size--;
    return result;
}
```

```
template<class SIT>
void Stack<SIT>::push(const SIT & x ){
    if(size >= capacity){
    //grow array by double the array size
    SIT *temp = SIT[capacity*2];
    for(int i = 0; i < capacity; i++)
        temp[i] = array[i];
    array = item;
    capacity*=2;
    }

    item[size] = x;
    size++;
}
```

Linked list implementation takes constant time on push ( ) and pop ( ) which is perfect. In the array implementation stack, we are facing a problem of growing the array when the array fills. When we grow the array, it is needed to create a new one with bigger size and copy the data over; too frequently updating is expensive (on running time). The best way to grow the array is double the size, which makes the push operation in constant time on average. (I am not going to prove it here; you will learn it in lecture).

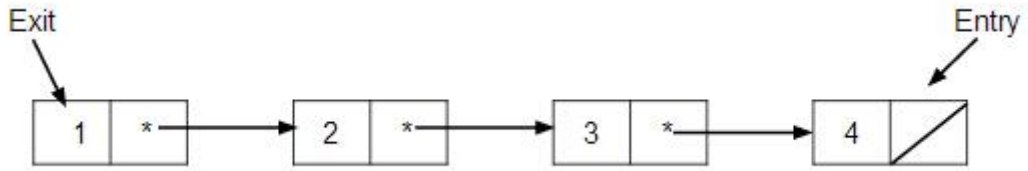
In conclusion, both implementations have constant time pushing and popping data from stacks. Which one you prefer? Even though the running time is constant on both, accessing an array is actually faster than accessing a linked list. Also, you don't need to implement a linked list when the array is already ready for you to use.

Queue

Queue is another constrained access linear structure like stack. However, queue is known as a fair one by First In First Out. There are also two ways to implement queue: linked List and Array. Both implementations give O (1) running times for every operation.

Linked List Implementation:

Linked list implementation is very straightforward. The only problem to concern in order to have constant time on both insert (enqueue) and delete (dequeue) is to choose the right entry and exit. 1) Entry points to the last node. 2) Exit points to the first node.

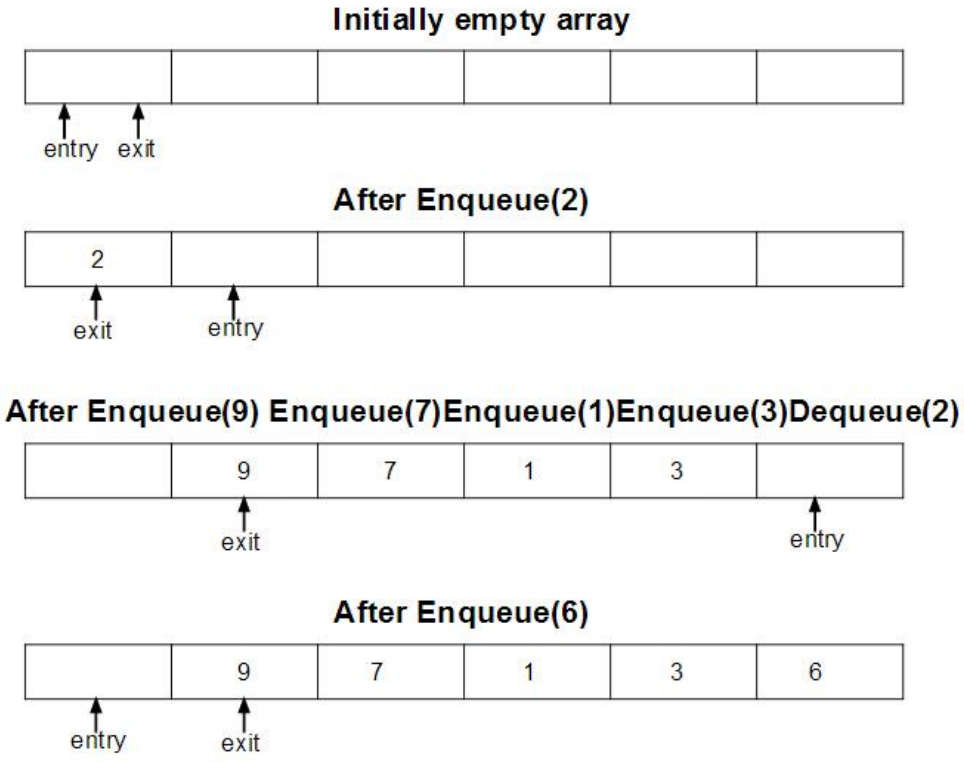


The figure above contains four nodes with data 1, 2, 3, and 4 respectively. To enqueue a data into the queue, you just need to dynamically allocate a new list node and the next pointer points to the entry node and update the entry pointer. To dequeue a list node, all you have to do is delete the exit node and update the exit pointer. Based on the First In First Out rule here, enqueue( ) and dequeue( ) take O (1) .

```
template<class SIT>
class Queue{
    public:
        Queue();
        ~Queue();
        Queue(const Queue<SIT> & other);
        void operator=(const Queue<SIT> & source);
        bool isEmpty() const;
        SIT dequeue();
        void enqueue(const SIT & x);
        int getSize();
    private:
        struct queueNode{
            SIT data;
            queueNode * next;
            queueNode(SIT newData){
                data = newData;
                next = NULL;
                }
            };
            queueNode * entry;
            queueNode * exit;
            int size;
    }
```

```
template<class SIT>
SIT Queue<SIT>::dequeue(){
    SIT result;
    if(exit != NULL){
        queueNode *temp = exit;
        exit = exit->next;
        result = temp->data;
        delete temp;
        size--;
        }
    return result;
    }

template<class SIT>
void Queue<SIT>:: enqueue(const SIT & x){
    queueNode* newNode = new queueNode(x);
    entry->next = newNode;
    entry = newNode;
    size++;
    }
```

Circular Array Implementation



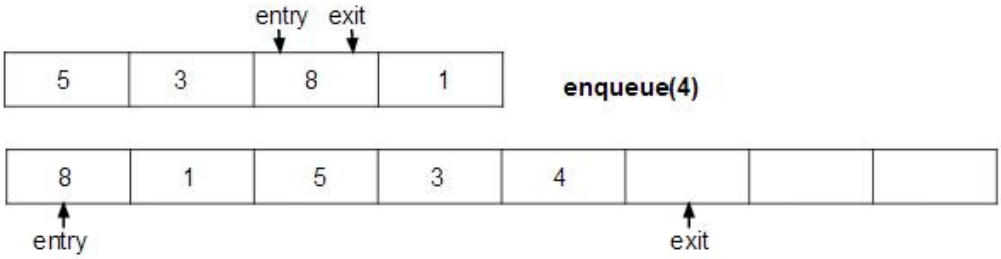
When the entry at the last slot of the array is filled, it does not have to be full because some of the slots in the front might dequeue so that the slots are now available to use. With circular array implementation, we never waste space in the array.

There are two more concerns about the circular array implementation:

1. Do not dequeue from an empty queue
2. Grow the array when it is full

To check whether the queue is empty, you should have a private variable such as "size" to keep track of the number of elements in the queue. So, you should check the number of elements in the queue before deleting an element in the dequeue function. Otherwise, when dequeueing from an empty queue, it will return an undefined value

Secondly, as long as the entry is not equal to the exit, the queue is not full. However, if the queue is full, create a new array with double size and copy the data.



Tree

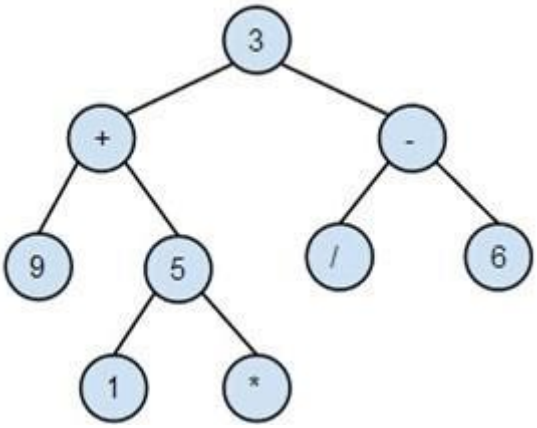
Tree is one of the most important nonlinear structures in computer science. There are some tree-terminologies you need to know before going further: *root, children, edge, parent, sibling, ancestor, descendant, leaves, subtree and height.* You should have learned them before. If you are not familiar with any of them, you can google it.

Height:

- The length of the longest path from the root to a leaf
- Empty tree has height –1 and a tree with only one node has height 0

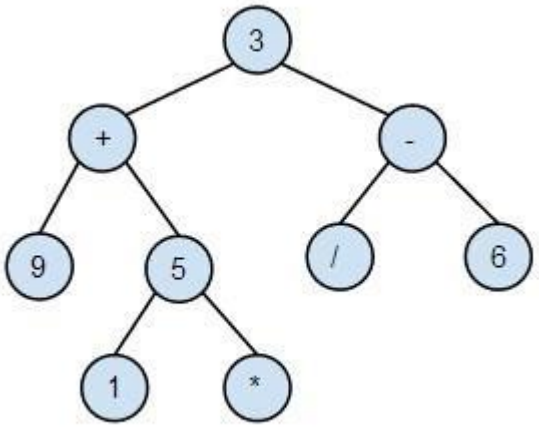
Binary Tree:

- rooted, directed, ordered
- each node has at most 2 children



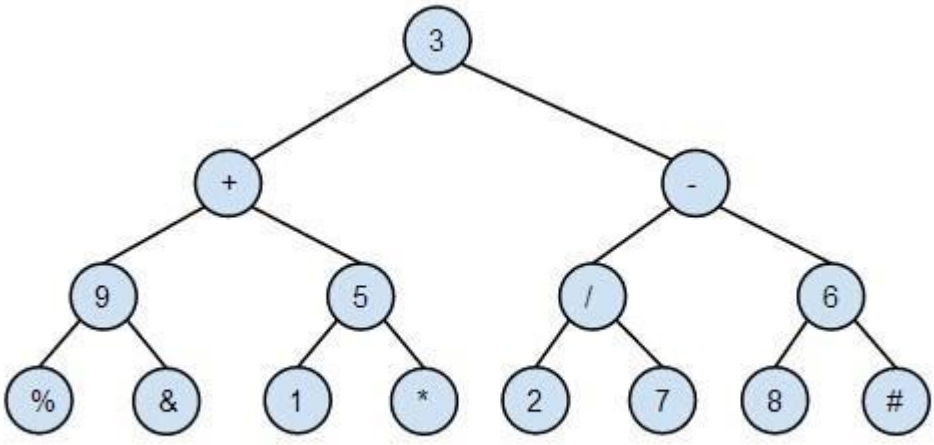
Full Binary Tree:

- A tree which every node has 2 or 0 children
- Each subtree are full



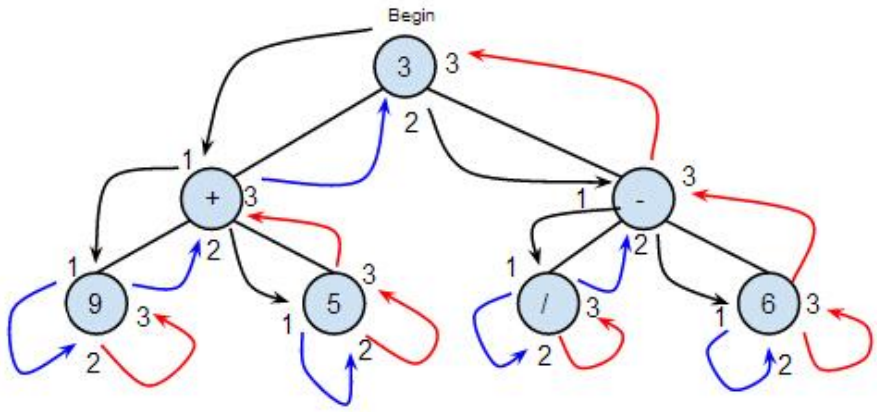
Perfect Binary Tree:

- All the leaves are at the same height
- Except the leaves nodes, all internal nodes have 2 children
- The numbers of nodes in a perfect tree is relative to its height
- # of nodes = 2^(h+1) -1



Binary Tree Traversal

- Visit every node in a tree (each node has its own data)
- Two choices of each node (left, right)
- After both left and right subtrees of a node are complete, move back up
- Each node is visited 3 times in a traversal
- O(n) to traverse a tree
- Each of these visits corresponding to a particular kind of traversal:
    1) Preorder traversal (first visit of a node)
    2) Inorder traversal (second visit of a node)
    3) Postorder traversal (third visit of a node)



Traversal often starts at the root of a tree. However, you might start at any of the node if you just want to traversal a specific subtree of a tree and the node is the root of the subtree.

Traversal the tree above and print out the data:
Preorder traversal: 3 + 9 5 - / 6
Inorder traversal: 9 + 5 3 / - 6
Postorder traversal: 9 5 + / 6 - 3
*levelOrder traversal: 3 + - 9 5 / 6 (from left to right at each level)

Traversal is very important while dealing with tree data structure. Functions, such as insert, re-move, copy, and clear, etc., are implemented by traversal. The following shows the four different traversal methods and some functions that a binary tree class needs.

```
template<class T>
void binaryTree<T>::preorder(TreeNode * node){
    if( node != NULL){
        dosomething(node->data);
        preorder(node->left);
        preorder(node->right);
    }
}
```

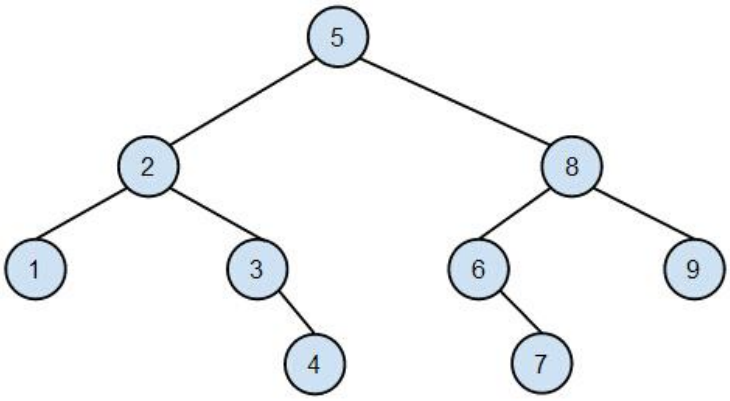
```
template<class T>
void binaryTree<T>::inorder(TreeNode * node){
    if( node != NULL){
        inorder(node->left);
        dosomething(node->data);
        inorder(node->right);
    }
}

template<class T>
void binaryTree<T>::postorder(TreeNode * node){
    if( node != NULL){
        postorder(node->left);
        postorder(node->right);
        dosomething(node->data);
    }
}
```

```
template<class T>
void binaryTree<T>::levelOrder(treeNode* node){
    queue<TreeNode*> q;
    q.enqueue(node);
    while(!q.isEmpety()){
        TreeNode * temp = q.dequeue();
        if(temp !=NULL){
            dosomething(temp->data);
            q.enqueue(temp->left);
            q.enqueue(temp->right);
        }
    }
}
```

```
template<class T>
TreeNode * binaryTree<T>::copy(TreeNode* other){
    if(other != NULL){
        TreeNode * node = new TreeNode(other->data);
        node->left = copy(other->left);
        node->right = copy(other->right);
    }
    return node;
}

//to clear a tree, use postorder traversal
//to delete each node from the leaves to root
template<class T>
void binaryTree<T>::clear(TreeNode * & node){
    if(node != NULL){
        clear(node->left);
        clear(node->right);
        delete node;
        node = NULL;
    }
}
```

Binary Search Tree

- A binary search tree is either empty or a root with left subtree and right subtree
- Left child of a node is always less than the node and right child is always larger than the node
- Usually, a node contains a key, data, left child pointer, and right child pointer
- Template dictionary implementation
- The height of a tree is log n where n is the total nodes in the tree
- Worst case running time is O(n) for insert, remove, and find



Find function:

- Given a root and a key as parameters, return a node with the given key in a given tree
- Dealing with pointer, check whether the root is null
- Compare the root key to the given key if the root isn't null; otherwise, return the node
- If the root key is greater the given key, recursive call the find function with the left child as parameter
- If the root key is less than the given key, recursive call the find function with the right child as parameter

Insert function:

- Insert function inserts a node as a leaf in a tree
- Given a root node, key, and data. Return nothing (void function).
- Check the root node; if it's null, dynamically allocate a new node with the given data and key and use root pointer points to it.
- Compare the key to determine the direction and then recursive call the function

Remove function:

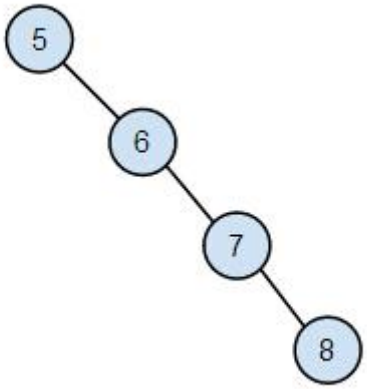
- Given a key and remove the node containing that key (void function)
- Again, check the root node first. If it's null, you are done
- Otherwise, using traversal to find the node containing the key
- Identify whether the node has not child, one child, or two children
- With no child, just delete the target node
- With one child, replace the target node with its child and delete the target node
- With two children, find the right most node of the left child of the target node by traversal, then replace the target node with it, and delete the target node

```
template<class D, class K>
struct treeNode{
    D data;  K key;
    treeNode * left;  treeNode * right;
    treeNode(K KEY, D DATA ){
        data = DATA; key = KEY;
        left = NULL; right = NULL;
        }
}

template<class D, class K>
treeNode * BST::find(treeNode* & tRoot, const k & key){
    if(tRoot == NULL) return NULL;
    if(tRoot->key == key)
        return tRoot;
    if(tRoot->key > key  )
        return find(tRoot->left, key);
    return find(tRoot->right, key);
}
```

```
template<class D, class K>
void BST::insert(treeNode* & tRoot, const k & key, const D & data){
    if(tRoot == NULL)
        tRoot = new treeNode(key, data);
    else if(key < tRoot->key)
        insert(tRoot->left, key, data);
    else if(key > tRoot->key)
        insert(tRoot->right, key, data);
}

template<class D, class K>
void BST::remove(treeNode* tRoot, const k & key){
    if(tRoot != NULL){
        if(tRoot->key == key)
            doRemoval(tRoot);
        else if (key < tRoot->key)
            remove(tRoot->left, key);
        else
            remove(tRoot->right, key);
    }
}
```

AVL Tree

As I mentioned previously, the running time of insert, find, and remove functions are O(n) in worst case where n is the total number of nodes in a binary search tree. If we insert 5,6,7,8 sequentially into a tree, what do you get? You will get a one way binary search tree below. If you want to find a node contains a key 8, you need to travel the entire tree to get it. If you want to insert any number greater than 8, you also have to go all the way to the leaf. In both case, you just travel all the nodes in the tree so that the running time is big O of n. In order to avoid this case, I am introducing you the AVL tree.

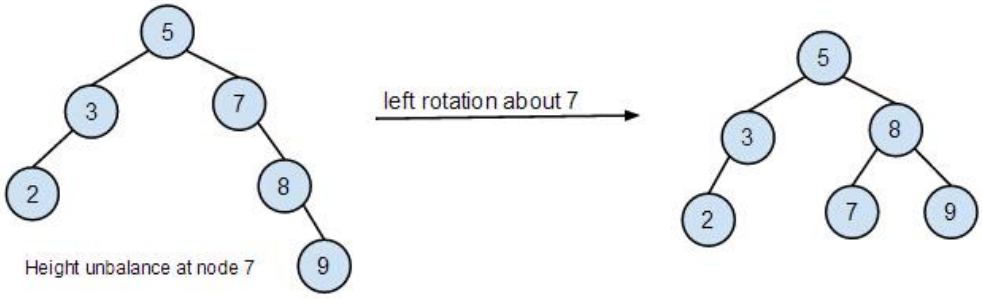


This is a one way BST

AVL Tree

- Binary search tree with height balanced in each node.
- Height balance of a tree is:

  abs(height of a left subtree - height of a right subtree) < 2
- Use rotation to maintain balance of each node in BST
- Insert, find, and remove functions are guaranteed with O(log n) in worst cases
- Rotation:
  - left, right, left-right, right-left
  - constant time operation
  - maintaining height



The subtree with root at node 7 has height balance = 2. We need a left rotation about node 7 to restore the AVL tree property.

Right rotation is similar to the left rotation on the above example. In order to check if a tree is height balanced, traverse the tree and find the maximum heights of the left subtree and right subtree of each node and calculate the height balance.

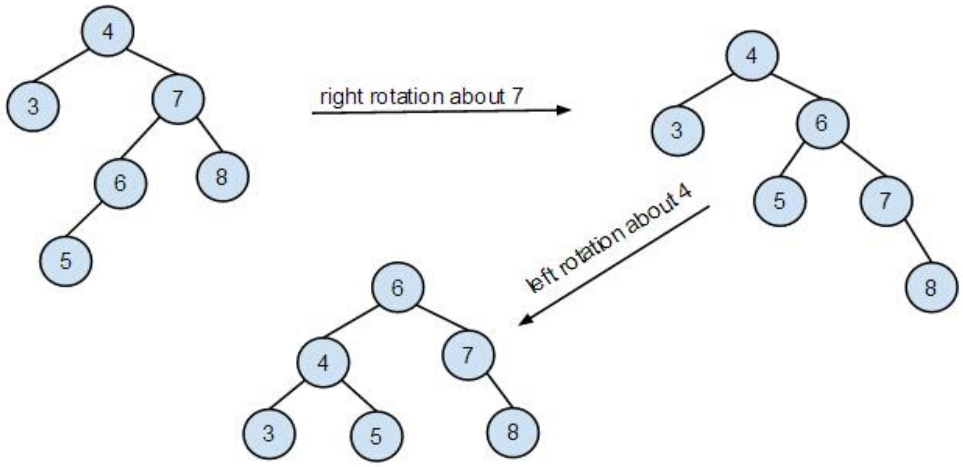
The algorithms to rotate left and right are simple. Here's the algorithm of left rotation

- Given the imbalanced node pointer(x)
- Create a stack node pointer(y) points to the its right child (y = x -> right)
- Right child of imbalanced node points to the left child of the stack node pointer

  (x ->right = y -> left)
- Left child of stack node points to the imbalanced node (y -> left = x)
- Imbalanced node points to stack node (x = y)

The figure below has height imbalance at node 4. This is an example of right-left rotation.

- Given a imbalanced node which is 4 in this case
- Call right rotation function on the right child of the imbalanced node
- Call left rotation function on the imbalanced node



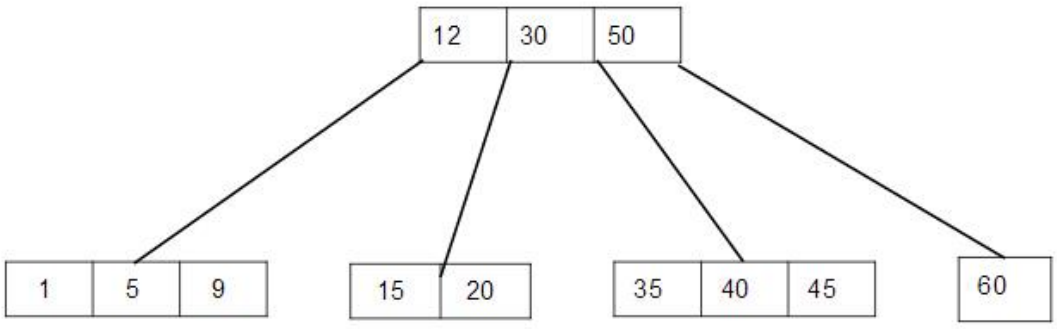
The left-right rotation function is the same logic as right-left rotation except you call the left rotation function on the left child of the imbalanced node and right rotation on the imbalanced node.

The insert and remove functions are a bit harder to code. After insertion or removal, you have to go up the tree to check for height balance. If the height balance property is not broken, you are done. Otherwise, you need to do rotation/s to restore it.

B-Tree

- Try to minimize the number of reads from disk (less time) because reading from disk is very slow. Searching key 35/40/45 in the figure below will require two disk reads
- For a m-way tree, internal nodes #key = #children –1
- The root is either a leaf or has between 2 and m children
- In a 32-way tree, maximum children is 32 and maximum key is 31 for each node
- All leaves are at the same level; hold no more than m-1key
- Non root internal nodes have between ceiling (m/2) and m children
- Keys in nodes are in order
- Search, insert, find are O(log n) in worst case
- Proofs of minimum number of nodes and maximum number of keys in a B-Tree might be introduced in class

B-Tree of order 4



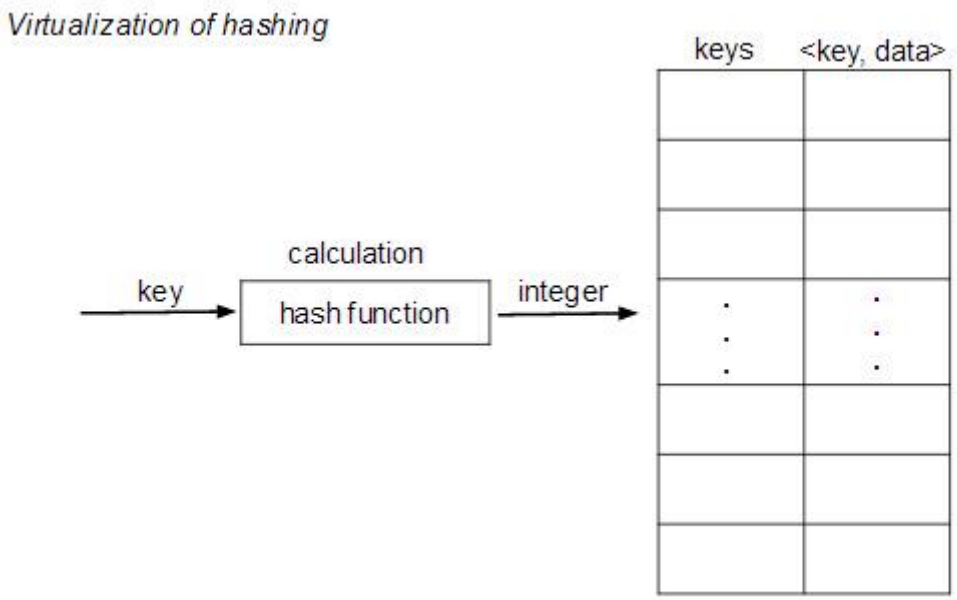
Hashing

- Two parts: a hashing function and an associative array
- Use a function to map a key space into a small set of integers
- Key space is a set of all the possible keys for data
- Hashing function → f (key): key -> int
- Accessing data: find the integer by a key and use the integer to get data from an array

  Example: hashing object h: h [key] = data

  Assume the [ ] operator has been overwritten for insert and find
- Perfect hash: bijection (both one-to-one and onto)
- An ideal hash function has three properties:
  1. Hashing function computes in constant time (O (1))
  2. Deterministic: if two keys are the same, hash function computes a same integer
  3. Satisfy the SUHA (Simple Uniform Hashing Assumption):

     Probability that two non-equal keys will hash to the same slots = 1/N Where N is the number of slots in an array
- Collision: when two or more different keys map to a same integer
- Two ways to handle hashing collision: separate chaining and probe based hashing
- Cofactor (α) = numbers of keys / numbers of cells

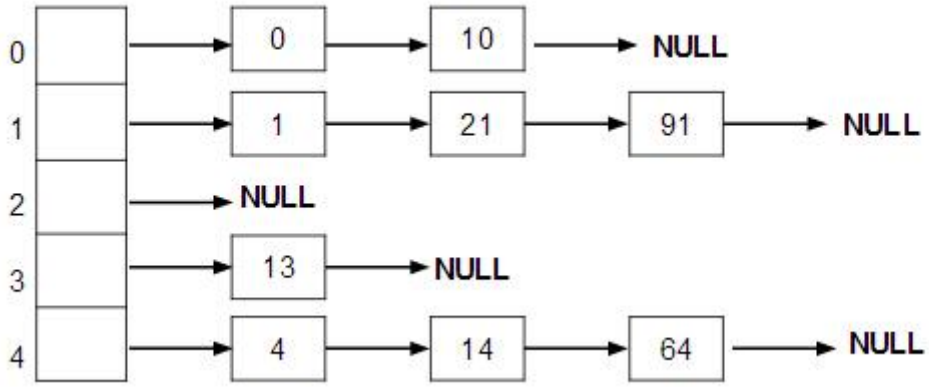


Collision Handling

Handling collision is the most important part on hashing. If more than one key hash into a same slot in its hash table (array) without collision handling, the insert functions will be crashed. And perfect hashing is barely happened in real problems.

A. Separate Chaining (open hashing):

- Each slot in a hash table contains a linked list
- Always insert in the front in linked list to guarantee the constant inserting time
- Also, try to create the best hashing function as you can



Insert Function:

- Since data always insert in the front of a linked list in each slot, insertion will only take constant time
- Hash function will return a head pointer of a certain list in the array of pointer
- Create a new list node and insert in the front

Remove Function:

- Using hash function to find the head of the linked list containing the key
- Go through the list and find the node with the given key and remove it

The remove and find functions might take O (n) in the worst case that all data hashing into the same slot. However, under SUHA, the probability of collision is the cofactor (α), their running time becomes O (α)

	Worst case	Under SUHA
insert	O (1)	O (1)
remove/find	O (n)	O (α)

B. Open Addressing (close hashing):

- Several kinds of close hashing, and the basic idea is if the slot has occupied, find one that has not.
- For example: linear Probing, quadric Probing, double hashing
- You may create your own way to find the available slot based on you key space

*close hashing might cost a big running time. Imagine when most of the slots are filled and only one left. Now you are going to insert 64 which will be hashing into index 4. In this case, the next available one is index 3 and you need to go around to check the empty slot which cost O(n-1). If n is very big, it will be expensive on running time. To avoid this, you can set when the array has filled at so point, such as 65% full, rehashing. This will improve the close hashing performance

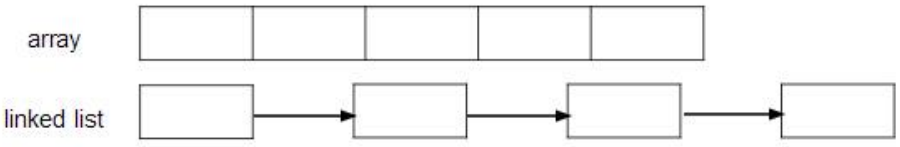
Rehashing:

When the array in a hashing object fills
- Double the size of the array to the next prime number size
- Compute a new hashing function and copy over the data
- Handle collision

Priority Queue

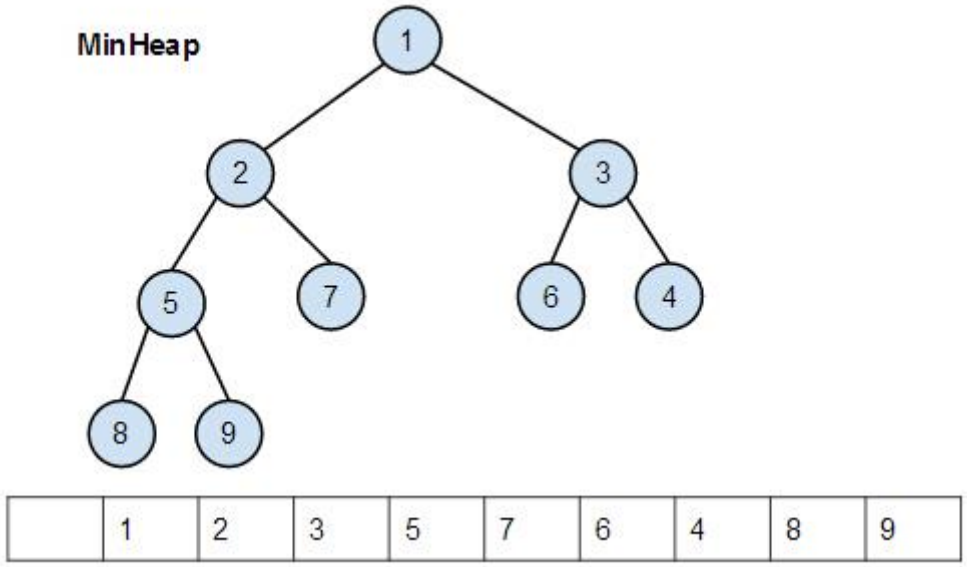
- Abstract data type which takes comparable types or objects as key such as, int, double, and string etc.
- Can be implemented with an array or a linked list
- Remove function, removeMin, always remove the minimum key
- As the array implementation and the linked list implementation take the same running time on both sorted and unsorted cases, you can choose the one that you prefer to

	insert	removeMin	implementation
unsorted array	O(1)	O(n)	- insert in the back only take constant time - removeMin needs to go through the array to find the minimum
unsorted linked list	O(1)	O(n)	same as unsorted array
sorted array	O(n)	O(1)	- binary search for the position and shift for insertion - minimum is always in the front, take constant time to remove
sorted linked list	O(n)	O(1)	- need to loop through the list for position - remove in the front



MinHeap

- A complete binary tree (leaves are filled from left to right)
- Children are always greater than parents
- To avoid dealing with pointer, we can represent the tree in an array form



- Binary trees and arrays are representations for heap, given either one, you will be able build the other.
- Keys increase as going down a path in the binary trees
- Keys start putting in an array in index 1 position for convenience
- Left child of node i is 2i in the corresponding array
- Right child of node i is 2i+1
- Parent of node i is the floor of (i / 2)
- Since it is a complete binary tree, height = log n where n is the total number of nodes
- RemoveMin always removes the minimum value which is the root in the tree or the first element in the array
- * maxHeap has the same structure as minHeap but its keys decrease as going down a path and remove function always remove the maximum

Insert:

- The most convenient way is to insert at the end of the array or leaf at the binary tree
- After insertion, heapifyUp the tree which means going up the tree to make sure the key is smaller than its child, if not, swap them.
- If the array fills, double the size
- Takes O (log n) by going through a path

RemoveMin:

- Remove the first element in the array and replace with the last element
- Heapify Down (compare with the smaller child)
- Takes O (log n) by going through a path

BuildHeap:

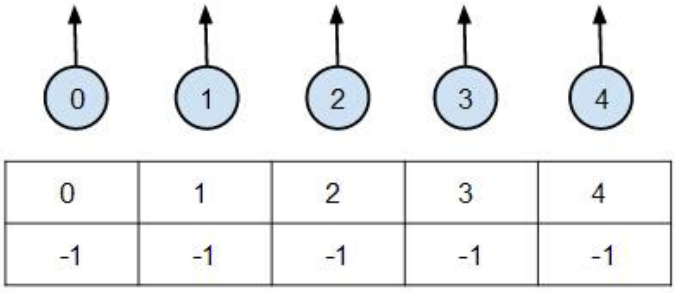
- Copy the given array to the heap's storage array
- Heapifydown from the end of the array to the beginning
- O (n)

```
template<class k>
void Heap<k>::insert(const & key){
    if(size == capacity) //size = # of keys; capacity = size of array
        growArray(); // double the array size and copy data over
    size++;
    elems[size] = key;
    heapifyUp(size);
}
template<class k>
void Heap<k>::heapifyUp(int index){
    if( index > 1){  //if index is 1, we are done
        if(elems[index] < elems[parent(index)]){
            swap(index, parent(index));
            heapifyUp(parent(index));
        }
    }
}
```

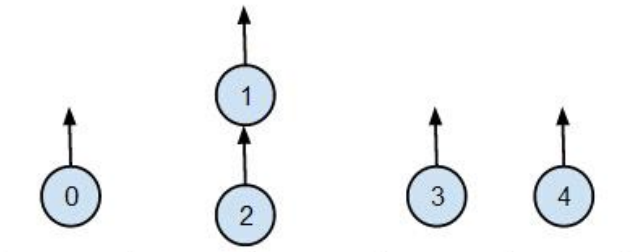
```
template<class k>
k Heap<k>::removeMin(){
    T min = elems[1];
    elems[1] = item[size];
    size--;
    heapifyDown(1);
    return min;
}

tempalt<class k>
void Heap<k>::heapifyDown(int index){
    //first, check if the node has any child
    if(hasAChild(index)){
        // compare with the smaller childe
        // minChild returns a smaller child if it has two children
        // and returns one if it has only one
        int minChildIndex = minChild(index);
        if(elems[index] > elems[minChildIndex]){
            swap(index, minChildIndex);
            heapifyDown(minChildIndex);
        }
    }
}
```

Disjoint Sets

- Usually implemented by uptree
- Uptree is easy to implement and all it needs just a simple array
- Array[i] in the array represents the parent of element i
- Each set has a representative or root which is the minimum index among the set
- Imagine that each index in the array acts like a key in a node with an arrow the picture below is an initial state of a disjoint set.
- When array value is –1, then we have reached a root
- Find function returns the index of a root (minimum)

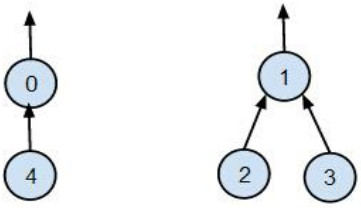


0	1	2	3	4
-1	-1	-1	-1	-1



0	1	2	3	4
-1	-1	1	-1	-1

- Pass the roots of two sets as parameters to union them
- To union two sets: union (find (1), find (2))
- Simply set the value of index find (2) to be find (1)
- In this case, the value of index 2 in the array change to 1 since now 1 is its parent
- After a few unions, you will get something like below

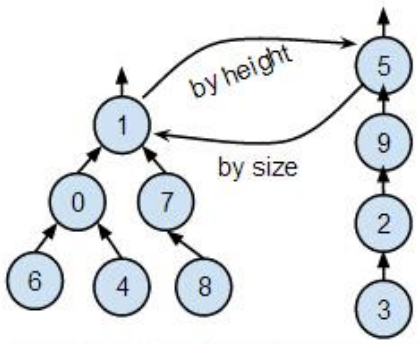


0	1	2	3	4
-1	-1	1	1	0

```
int DisjointSet:: find(int index){
    if(array[index] < 0)
        return index;
    return find(array[index]);
}

void DisjointSet:: Union(int root1, int root2){
    //randomly select root2 to be the new root
    s[root1] = root2;
}
```

Smart Union

- Union by height:
  Keep the height of the uptree as small as possible
  Negative value in the root represents the height of the tree
- Union by size:
  Would increases distance to the root
  Negative value in the root represents the numbers of nodes in the tree
- Both methods guarantee the height of the uptree to be O (log n)



Union by height

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

5

after the union of two uptrees, the height of the new uptree is still 3

Union by size

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

-10 1

the new uptree has 10 nodes in total. The root value changes to -10

```
void DisjointSet::unionBySize(int root1, int root2){
    int newSize = array[root1]+array[root2];
    if(array[root1] <= array[root2]){
        array[root2] = root1;
        array[root1] = newSize;
    }
    else{
        array[root1] = root2;
        array[root2] = newSize;
    }

}
```

Path Compression

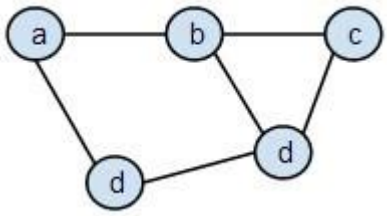
- suppose find(x), every node on the path from x to the root has its parent changed to the root. Next time you call find function, it will be much faster especially for larger uptree

```
int DisjointSet:: find(int index){
    if(array[index] < 0)
        return index;
    return array[index] = find[array[index];
}
```

* Recommend to draw out an example and follow the code step by step

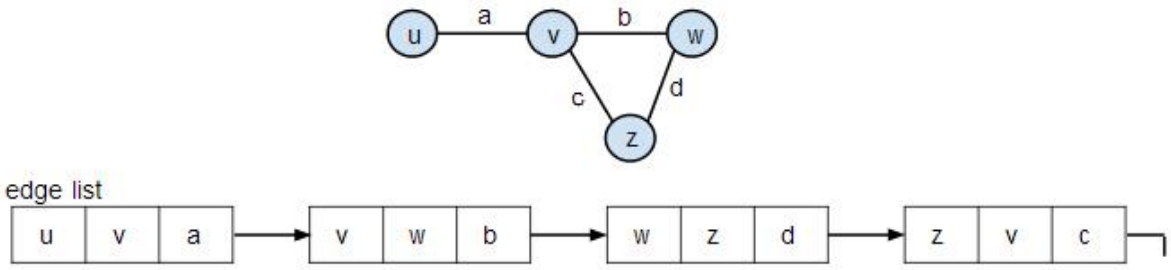
Graph

- A graph G = (V, E) consists of a set of vertices V and a set of edge E
- A pair of two vertices represents an edge: pair (v, w) where v and w are vertices
- Incident Edges of vertex v: all the edges that connect v and other nodes
- Degree of v (deg(v)): the numbers of edges that connect to v
- Path: sequence of vertices connected by edges
- Cycle of a graph: has a common begin and end vertex
- Simple graph: graph with no self-loop and no multi-edges
- A subgraph S of a graph G is a graph whose all the vertices and edges are subsets of G
- Spanning tree of a graph is a connected and undirected graph which includes all of the vertices and some or all of the edges of the graph
- Normally, **n** represents the numbers of nodes and **m** represents the numbers of edges
- Important functions in graphs:
    1. insertVectex (vertex v)
    2. removeVertex (vertex v)
    3. areAdjacent (vertex v, vertex u)
    4. incidentEdge (vertex v)



Graph Sample

Edge List Implementation



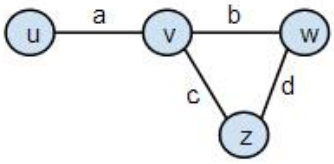
insertVertex (vertex v) : O(1) insert at the front of the list

removeVertex (vertex v): O(m) linearly search to remove all the edges including the given node

areAdjacent (vertex v, vertex u): O(m) linearly search on the edge list

incidentEdge (vertex V): O(m) linearly search on the edge list

AdjacencyMatrixImplementation



	u	v	w	z
u	0	1	0	0
v	1	0	1	1
w	0	1	0	1
z	0	1	1	0

- 1 indicates the existence of the edge; 0 otherwise
- Good for high density graphs

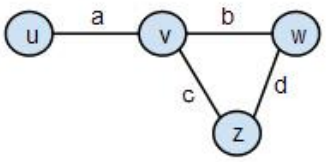
insertVertex (vertex v) : O(n) search the vertices that connect to the given one and mark 1 or 0

removeVertex (vertex v): O(n) search for the adjacency vertices and change to 0

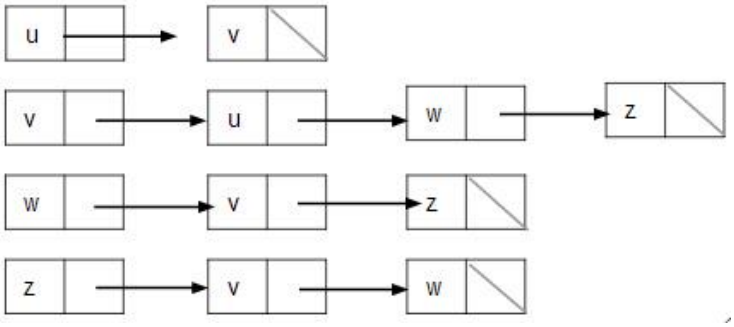
areAdjacent (vertex v, vertex u): O(1) access the intersection of two given vertices

incidentEdge (vertex V): O(n) linearly search on the row or column of the vertex

AdjacencyListImplementation



- good for sparse graph (short linked list)
- each vertex has its own linked list with adjacent nodes



InsertVertex (vertex v): O (n) searches all the vertices. If there exists an edge between the given vertex and any of them, insert a new vertex node

removeVertex (vertex v): O(m) delete the linked list that begin with the given vertex
check the rest of the linked list to see if they also contain the given vertex, if yes, delete the node

areAdjacent (vertex v, vertex u): O(min (deg (v), deg (u)) checking one of these two linked list is sufficient to find out

incidentEdge (vertex V): O(deg(v)) each list node in a vertex represents a incident edge

Performance

	Edge List	Adjacency Matrix	Adjacency List
insertVertex(v)	1	n	n
removeVertex(v)	m	n	m
areAdjancent(v, w)	m	1	min(deg(v), deg(w))
incidentEdge(v)	m	n	deg(v)

Traversal on Graphs:

- Two ways to traversal: BFS and DFS
- Both have running time O (m+n) with adjacency list implementation
- The basic logic of BFS is similar to lever order algorithm. If you don't remember what level order algorithm is, go back to the lecture notes now.

BFS

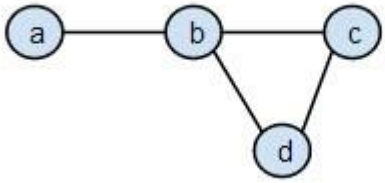
This is a BFS template:

- Label all the vertices and edges with “unvisited”
- Create a vertex queue
- Find a starting vertex, push it into the queue and change its label to visited
- While the queue is not empty, get the front vertex and then dequeue it
- Get all the adjacency vertices of the vertex from last step
- For all the adjacency vertices, if label of a vertex is “unvisited”, push it into the queue and do something with edges or vertices or data (depends on your function description)

Example:

- Traversal a graph and classify each edge as either “discovery” or “cross”
- For a vertex, if an adjacency vertex is unvisited, the edge between them is classified as “discovery”. Otherwise, the edge is “cross”.

```
For all u in graph.vertices( )
    setLabel (u, unvisited)
For all e in graph.edges( )
    setLabel (e, unvisited)
queue <vertex> q
Graph.getStartVertexV( )
setLabel (v, visited)
q.enqueue (v)
While (! q.empty( ) )
    q.dequeue( )
    // v is the front in the queue
    For all w in graph.adjacentVertices( v )
        If getLabel (w) == unvisited
            setLabel ( (v, w) , discovery)
            setLabel (w, visited) q.enqueue(w)
        else if getLabel (v, w) == unvisited
            setLabel ( (v, w), cross)
```



The above steps are the basic BFS implementation; you will need to add some more steps depending on your task.

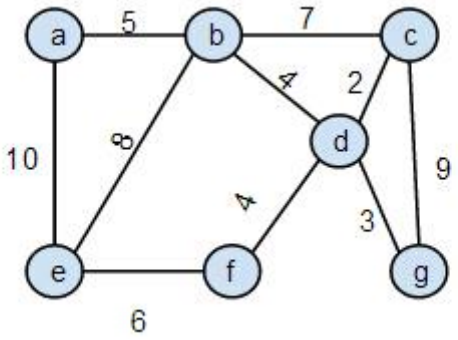
DFS

- Use recursive method to traversal all vertices in a graph
- Same running time as BFS, O (m+n) with adjacency list implementation
- Below is a template of DFS for the general style

```
For all u in graph.vertices ( )
    setLabel (u, unvisited)
For all e in graph.edges ( )
    setLabel (e, unvisited)
For all v in graph.vertices ( )
    If getLabel (v) == unvisited
        DFS (G, v)

DFS (graph G, vertex v) {
    Visited[v] = visited;
    For each w adjacent to v
    If (visited[w] == unvisited)
        DFS (w)
    }
```

Minimum Spinning Tree

- Minimum spinning tree or MST of a weighted graph G is a tree with edges that connects all the vertices in G at the lowest total weight
- Number of edges in MST is always n—1 where n is the total number ofvertices
- MST exists iff the graph is connected
- Two basic algorithms to find a MST from a graph:
    1. Kruskal's Algorithm
    2. Prim's Algorithm

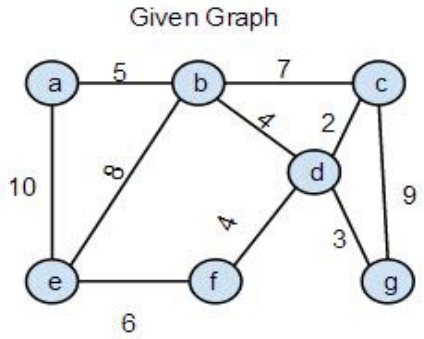


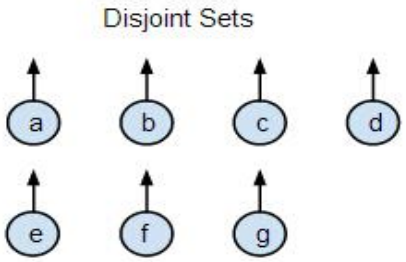
Example of a weighted graph

Kruskal's Algorithm

Given a connected and undirected graph with weighted edges

1) Initialize a graph T with all the vertices and no edge, T will be the output
2) Initialize a disjoint sets structure where each vertex is represented by a set
3) Initialize a priority queue that is sorted by the weights of the edges on the graph
4) Remove the minimum from the priority queue. If that edge connects two vertices from different sets, set the edge to T and union the two vertex set. Otherwise, do nothing. Repeat until n-1 edges are set to T

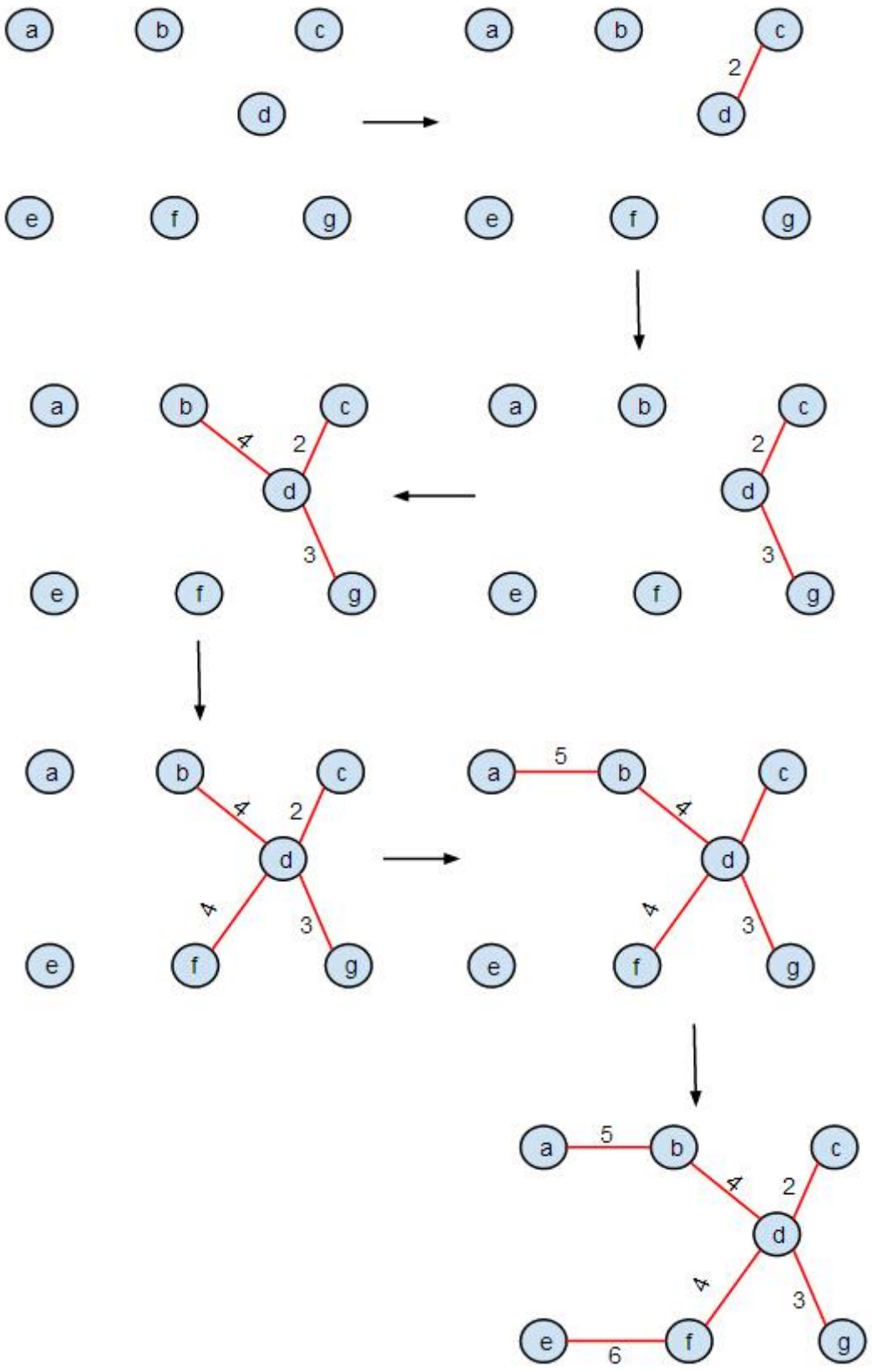




Priority Queue

(c,d)
(d,g)
(b,d)
(d,f)
(a,b)
(e,f)
(b,c)
(b,e)
(c,g)
(a,e)

See the step-by-step graph on the next page



removing the minimum edge from priority queue one by one until you have set n-1 edges in the graph T. If two vertices are not in the same set, they are not connected by an edge. So, you need to set an edge to connect them and union them.

Kruskal's Algorithm Pseudocode:

```
Graph T
contains all vertices
and no edge

DisjointSets DS
For every vertex v in V
Set v as a set

Priority queue Q
Insert all edges into Q
and sort by weights

While T has less than n-1 edges
    edge e = Q.removeMin( )
    let u and v are endpoints of e
    If DS.find(u) != DS.find(v)
        set edge between u and v
        DS.smartUnion( DS.find(u) , DS.find(v) )

Return T
```

Priority queue can be implemented by sorted array or minHeap.

m = number of edges
n = number of nodes

Priority Queue	Heap	Sorted Array
To Build	O (m)	O (m log n)
To removeMin	O (log n)	O (1)

As we have learned before, building a minHeap takes big O of m time and removeMin takes big O of log m because you need to restore the heap property.

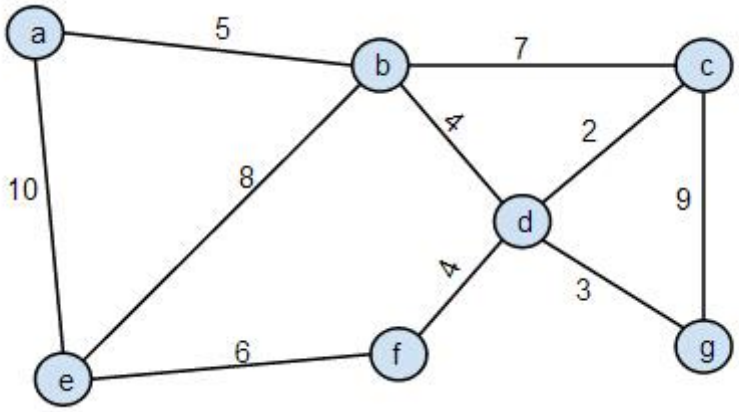
Building a sorted array needs big O of (m log m) and remove only takes constant time.

NOTE:

m = n*n in worst case that all vertices are connected to each other. So, removeMin in heap takes O (log n) because log m = log n*n = 2log n = log n; and building a sorted array takes O(m log n) because m log m = m log n*n = 2m log n = m log n.

Prim's Algorithm

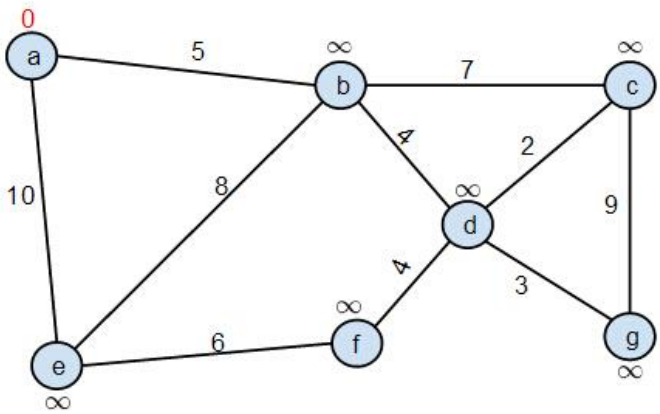
- Another method to find the MST of a weighted graph
- Based on the Partition Property



Given a connected and undirected graph with weight edges

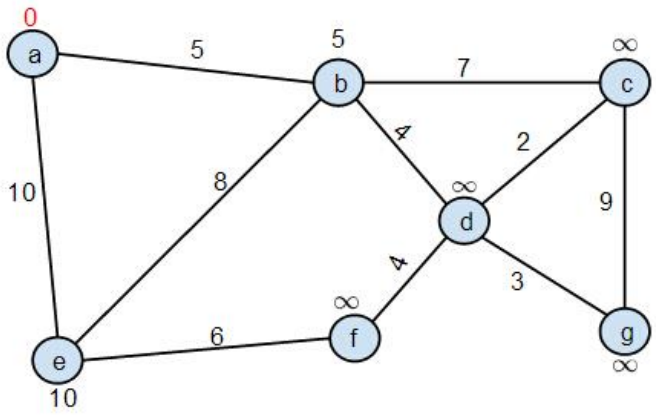
1. For all vertices v, distance[v] = "infinity" (use INT_MAX if weight type is int)
2. Set pre[v] = NULL // previous vertex of v is NULL
3. Initialize a starting vertex s, distance[s] = 0
4. Initialize priority queue
5. Initialize a set of labeled vertices to unvisited
6. Repeat the following step n times (spanning tree contains all the vertices in the graph)
   Find and remove the minimum distance [ ] v labelled with unvisited from priority queue
   Label vertex v as visited
   For all the labelled with unvisited neighbors w of v
      If weight (v, w) < distance[w]
         Distance [w] = weight (v, w)
         pre [w] = v

	label	distance	previous
a	unvisited	0	NULL
b	unvisited	infinity	NULL
c	unvisited	infinity	NULL
d	unvisited	infinity	NULL
e	unvisited	infinity	NULL
f	unvisited	infinity	NULL
g	unvisited	infinity	NULL



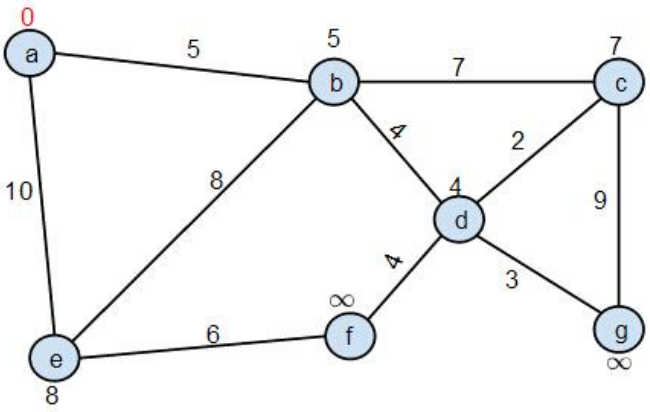
Before entering the loop

	label	distance	previous
a	visited	0	NULL
b	unvisited	5	a
c	unvisited	infinity	NULL
d	unvisited	infinity	NULL
e	unvisited	10	a
f	unvisited	infinity	NULL
g	unvisited	infinity	NULL



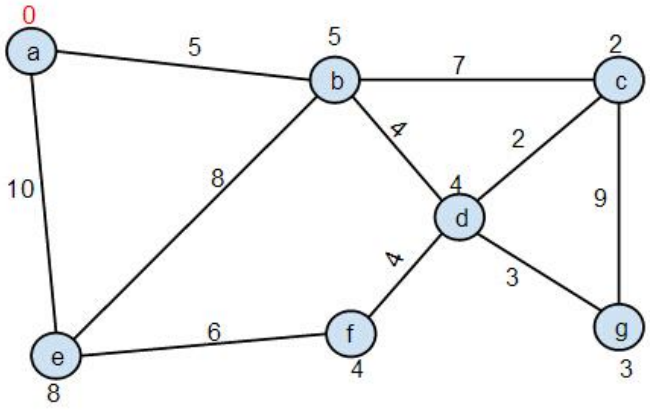
After vertex ‘a’ was declared as visited

	label	distance	previous
a	visited	0	NULL
b	visited	5	a
c	unvisited	7	b
d	unvisited	4	b
e	unvisited	8	b
f	unvisited	infinity	NULL
g	unvisited	infinity	NULL



After vertex ‘b’ was declared as visited

	label	distance	previous
a	visited	0	NULL
b	visited	5	a
c	unvisited	2	d
d	visited	4	b
e	unvisited	8	b
f	unvisited	4	d
g	unvisited	3	d

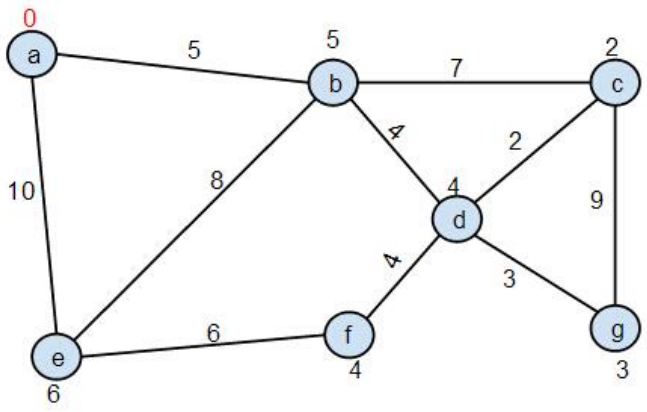


After vertex ‘d’ was declared as visited

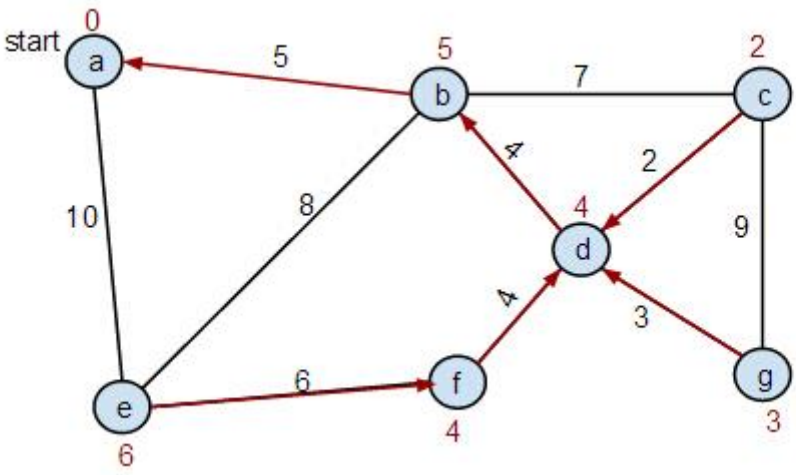
After vertex *c* was labelled as visited, nothing changes to the MST. The adjacency vertices of *c* are *b*, *d*, and *g*; *b* and *d* has marked as visited and the weight of the edge between *c* and *g* is greater than the current weight of *g*.

After vertex g was labelled as visited, all of its adjacency vertices has labelled as visited. Therefore, nothing has to be done.

	label	distance	previous
a	visited	0	NULL
b	visited	5	a
c	visited	2	d
d	visited	4	b
e	visited	6	f
f	visited	4	d
g	visited	3	d



After vertices 'f' and 'e' were declared ad visited



MST on the original graph

*The red arrows indicate the previous vertex of the current one

Table below shows the running time of different implementation of graphs
For more information, go to class!

	adjacency Matrix	adjacency List
minHeap	O (n^2 +m log n)	O (n log n + m log n) sparse graph
unsorted array	O (n^2) dense graph	O (n^2) dense graph

Dijkstra's Algorithm

- Find the single source shortest path from a directed and weighted graph
- Does not work on negative weight but shortest path might exists
- Negative weight in cycle implies no shortest path exists
- Mostly the same algorithm as prim's

Input: directed graph with non-negative edge weights and a start vertex s
Output: a subgraph consisting of the shortest path from s to every other vertex in the graph

1. For all vertices v, distance[v] = "infinity" (use INT_MAX if weight type is int)
2. Set pre[v] = NULL // previous vertex of v is NULL
3. Initialize a starting vertex s, distance[s] = 0
4. Initialize priority queue
5. Initialize a set of labeled vertices to unvisited
6. Repeat the following step n times
   Find and remove the unlabeled vertex v with minimum distance [ ] from priority queue
   Label vertex v as visited
   For all the label with unvisited neighbors w of v
      If (distance[v] + weight (v, w)) < distance[w])
         Distance [w] = (distance[v] + weight (v, w))
         Pre [w] = v

7. At the end, each vertex will mark with the minimum weight from the start vertex to it.

Running Time:
   Same as Prim's Algorithm

Reference

Data Structures and Programming Principles Course, Professor Cinda Heeren, computer science department, University of Illinois at Urbana-Champaign

Data Structures and Algorithms in C++, Mark Allen Weiss

http://www.cplusplus.com/