

## Meeting ID:

=====

1. Please join my meeting.

<https://global.gotomeeting.com/meeting/join/163634853>

2. Use your microphone and speakers (VoIP) - a headset is recommended. Or, call in using your telephone.

Dial +1 (571) 317-3112

Access Code: 173-379-397

Audio PIN: Shown after joining the meeting

Meeting ID: 173-379-397

=====

[Prerequisite - Windows Development tool installation \(NetBeans IDE\):](#)

[Class 1](#)

[Sign In Here Please](#)

[Questions? Type here](#)

[Content](#)

[C/C++ program memory layout](#)

[Pointer/reference, pass by reference vs. value](#)

[Array \(1D\) - the traditional way](#)

[c++ stl: "vector"](#)

[Homework practices \(class 1\)](#)

[Class 2](#)

[Sign in here please](#)

[Question - please type here](#)

[FAQ](#)

[Passing array vs. passing vector](#)

[++i vs. i++](#)

[Best practice of writing code & test - how to arrange files?](#)

[How to create graph d/s:](#)

[Content](#)

[Array vs. LinkList:](#)

[C++ stl "sort"](#)

[c++ stl "priority\\_queue"](#)

[Homework practices \(class 2\)](#)

[Binary Search \(recursion\)](#)

[Binary Search Variant 1 - find left boarder \(Iteration\)](#)

[Binary Search Variant 1 - find left boarder \(recursion\)](#)

[Binary Search Variant 2 - find right boarder \(iteration\)](#)

[Binary Search Variant 2 - find right boarder \(recursion\)](#)

[Find element in sorted array that has been rotated \(shifted\) unknown times.](#)

[Find element in unknown size \(sorted\) array.](#)

### [Class 3](#)

[Sign In Here Please](#)

[Questions?](#)

[Content](#)

[c++ stl "stack"](#)

[c++ stl "queue"](#)

[c++ stl "deque"](#)

[Heap and its implementation](#)

[Definition](#)

[Heapify a sequence of numbers](#)

[Implementation](#)

[Queue implementation supporting O\(1\) push & pop.](#)

[Stack implementation supporting getMin\(\) with smaller memory usage](#)

[Homework practices \(Class 3\)](#)

[Implementing a queue using two stacks.](#)

[Implement a stack with min\(\) function](#)

[How to reverse a linkedList in a recursive way](#)

[How to reverse a linkedList in an iterative way](#)

[Find linked list middle element](#)

[Determine if a linked list is cycled](#)

[Insert value into a sorted linked list](#)

[Merge two sorted linked list](#)

[Reorder list 1->2->3->4->5 to become 1->5->2->4->3](#)

[Partition a list given a target value such that all nodes with value less than target go to the left portion, the rest goes to the right.](#)

### [Class 4](#)

[Questions during the class?](#)

[FAQ](#)

[priority\\_queue usage.](#)

[Content](#)

[Fundamentals of BT & BST](#)

[Binary Tree](#)

[Binary Search Tree](#)

[Traverse Tree using recursion](#)

[Insert, find, delete an element in a binary search tree](#)

[Insert](#)

[Find](#)

[Delete](#)

[Tree Traversals - Pre/In/Post-Order](#)

[Definition](#)

[Iteration - the hard way](#)

## Class 5

Questions?

Content

c++ stl "set"

c++ stl "map"

Hash table

Summary: Binary search tree vs. Hash table

Implement a map

Using BST

Using Hash Table

Homework Practices

Determine whether an undirected graph is Bipartite

Find smallest k elements from an unsorted array

Find kth element in row+column sorted 2D N\*N matrix

Print the values of the nodes in a binary tree layer by layer

## Class 6

FAQ

About balanced binary search tree

A good hash function for string

Content

2D Array (with comparison to 1D)

Dynamically Allocating 1D/2D Array

Initializing 1D/2D Array

Graph Traversal Code

BFS

DFS

Iteration

Recursion

Homework Practices (BFS + DFS)

Determine whether an undirected graph is Bipartite

Print the values of the nodes in a binary tree layer by layer

Print all valid permutations of () () ()

Given infinite 25/10/5/1 cents, get number of ways to represent n cents

## Class 7

FAQ

Best practices when initializing vector

Using function argument (const int) to initialize array - invalid!

Return value using "pass by reference"

Content - All that you need to know about bit operations (for interview)

Basic logic operators:

Print an unsigned integer in binary format

Signed & Unsigned integers

Bit shifts

[Endianness \(Big Endian, Little Endian\)](#)

[Concept](#)

[Determine machine endianness in C++ program](#)

[Practices \(including homework exercises\)](#)

[Count one bit in int](#)

[Determine whether a number x is a power of 2](#)

[Reverse bits of unsigned int](#)

[Print all subsets of a set](#)

[Number of different bits between two positive integers](#)

[Bit tricky manipulations](#)

[Class 8](#)

[C++ stl "string"](#)

[C string, and common string.h APIs](#)

[memset, memcpy, memcmp, and struct memory layout](#)

[Homework Practice](#)

[Deduplicate same adjacent chars in a string](#)

[Update all space in a string to "%20"](#)

[Reverse words in a sentence: "hello world lol" -> "lol world hello"](#)

[Determine if a string is a substring of another](#)

[Class 9](#)

[FAQ](#)

[Pointer cast vs. Direct type cast](#)

[Some Frequent Keywords](#)

[const](#)

[static/extern](#)

[virtual](#)

[malloc](#)

[malloc vs new](#)

[Case study: allocating memory for 2D array with only one malloc call](#)

Prerequisite - Windows Development tool installation (NetBeans IDE):

**Note 1: Using NetBeans is absolutely just a suggestion. :) If you are already familiar with IDE such as eclipse, etc., or used to programming in the shell with vim/emacs, feel free to stick with your tradition.**












**Note 2: Installing NetBeans in Mac is very similar to the steps listed below, except that you don't have to install Cygwin, which is a win! Please download JDK and Netbeans for Mac OS accordingly.**

## 1. Download JDK

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Select appropriate option (32 bit/ 64 bit)

For example, my machine is 64 bit, I selected `jdk-8u20-windows-x64.exe`

Java SE Development Kit 8u20		
You must accept the <a href="#">Oracle Binary Code License Agreement for Java SE</a> to download this software.		
Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.		
Product / File Description	File Size	Download
Linux x86	135.24 MB	 <a href="#">jdk-8u20-linux-i586.rpm</a>
Linux x86	154.87 MB	 <a href="#">jdk-8u20-linux-i586.tar.gz</a>
Linux x64	135.6 MB	 <a href="#">jdk-8u20-linux-x64.rpm</a>
Linux x64	153.42 MB	 <a href="#">jdk-8u20-linux-x64.tar.gz</a>
Mac OS X x64	209.11 MB	 <a href="#">jdk-8u20-macosx-x64.dmg</a>
Solaris SPARC 64-bit (SVR4 package)	137.02 MB	 <a href="#">jdk-8u20-solaris-sparcv9.tar.Z</a>
Solaris SPARC 64-bit	97.09 MB	 <a href="#">jdk-8u20-solaris-sparcv9.tar.gz</a>
Solaris x64 (SVR4 package)	137.16 MB	 <a href="#">jdk-8u20-solaris-x64.tar.Z</a>
Solaris x64	94.22 MB	 <a href="#">jdk-8u20-solaris-x64.tar.gz</a>
Windows x86	161.08 MB	 <a href="#">jdk-8u20-windows-i586.exe</a>
Windows x64	173.08 MB	 <a href="#">jdk-8u20-windows-x64.exe</a>

## 2. Install JDK (All default options).

## 3. Download netbeans at:

<https://netbeans.org/downloads/>

NetBeans IDE Download Bundles					
Supported technologies *	Java SE	Java EE	C/C++	HTML5 & PHP	All
④ NetBeans Platform SDK	•	•			•
④ Java SE	•	•			•
④ Java FX	•	•			•
④ Java EE		•			•
④ Java ME					•
④ HTML5		•		•	•
④ Java Card™ 3 Connected					•
④ C/C++			•		•
④ Groovy					•
④ PHP				•	•
Bundled servers					
④ GlassFish Server Open Source Edition 4.1		•			•
④ Apache Tomcat 8.0.9		•			•
<div>Download</div> <div>Download</div> <div>Download</div> <div>Download</div> <div>Download</div>					

Choose “ALL” to download, or go to the below link:

<https://netbeans.org/downloads/start.html?platform=windows&lang=en&option=all>

#### 4. Install Netbeans. (All default options)

#### 5. Download Cygwin installation file:

<http://cygwin.com/install.html>

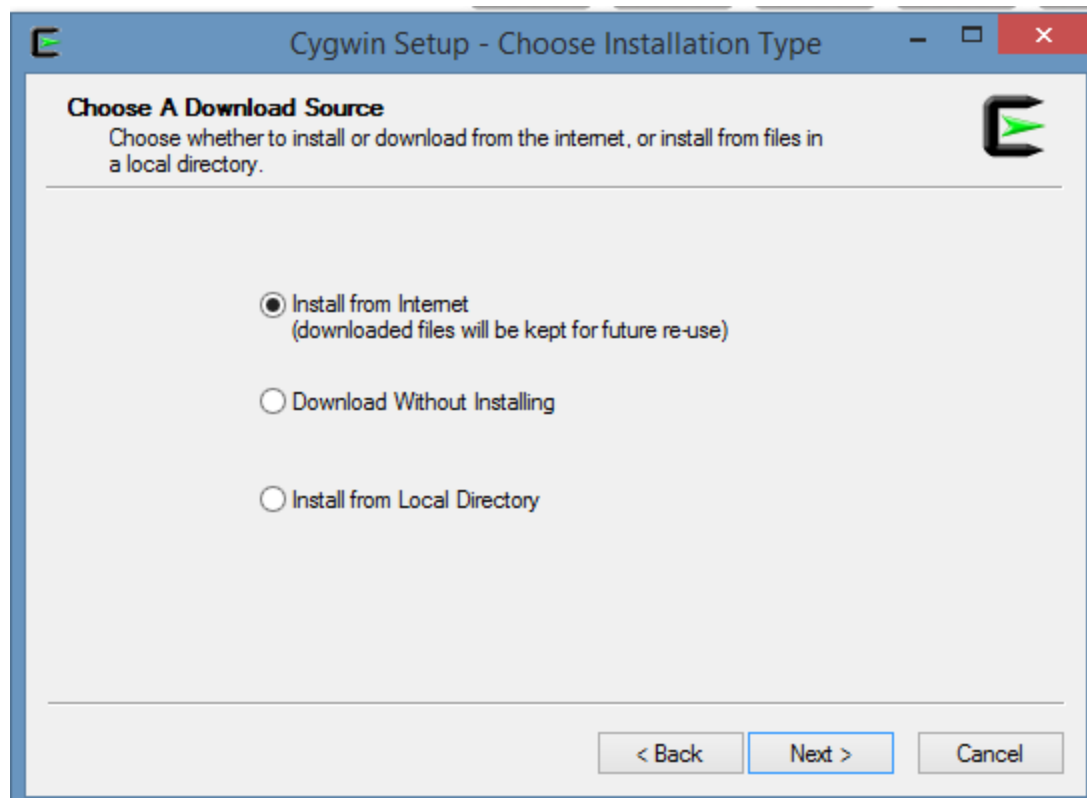
Choose your appropriate version (32/64 bit), e.g., in my case, I choose this:

##### Installing and Updating Cygwin for 64-bit versions of Windows

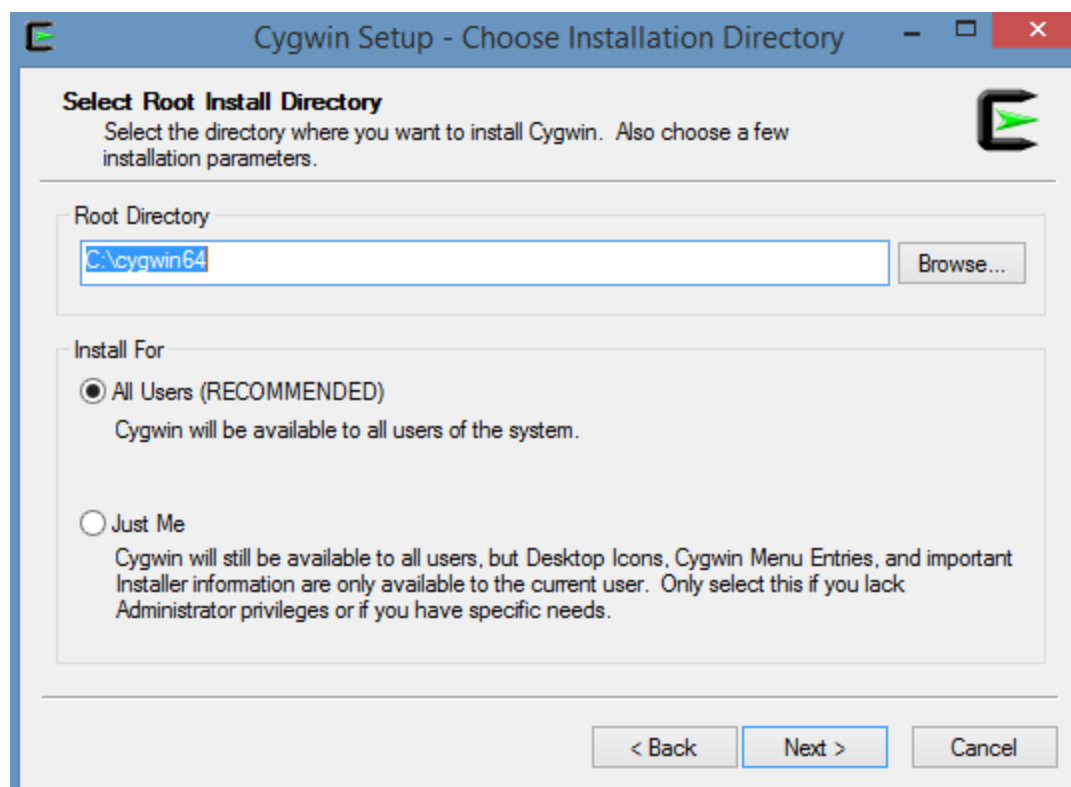
Run [setup-x86\\_64.exe](#) any time you want to update or install a Cygwin package for 64-bit windows. The [signature](#) for [setup-x86\\_64.exe](#) can be used to verify the validity of this binary using [this](#) public key.

#### 6. Install Cygwin:

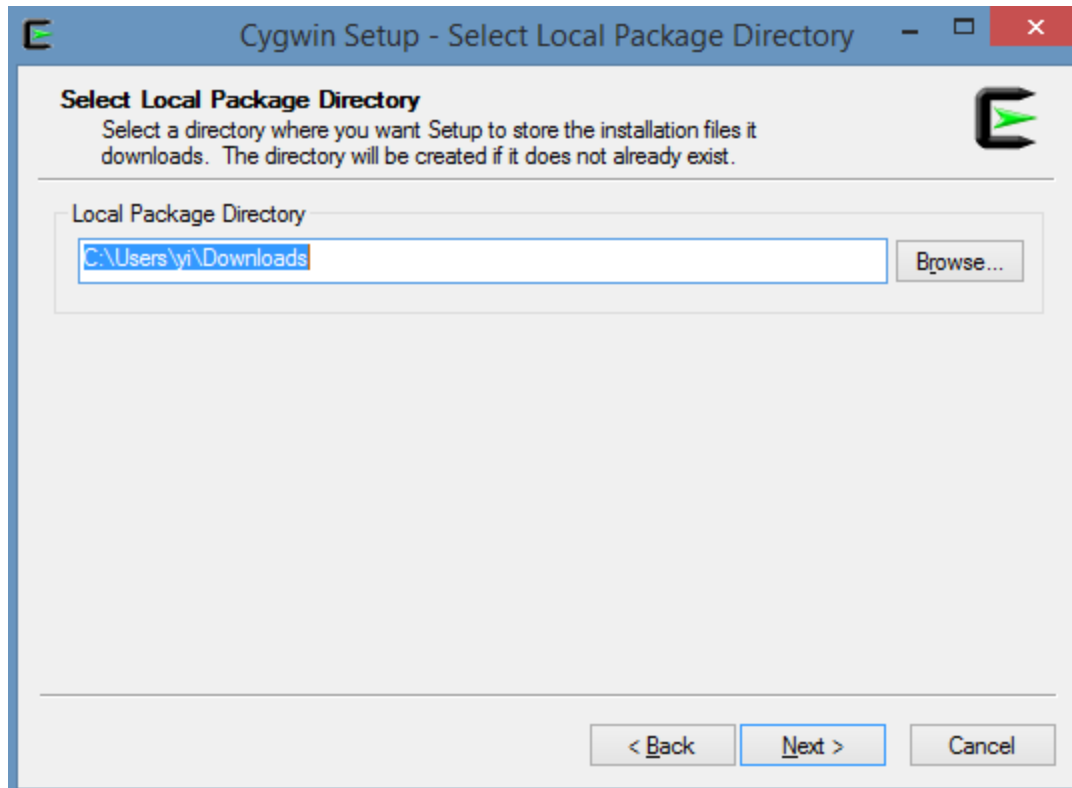
Run the downloaded file: (in my case, it is [setup-x86\\_64.exe](#))



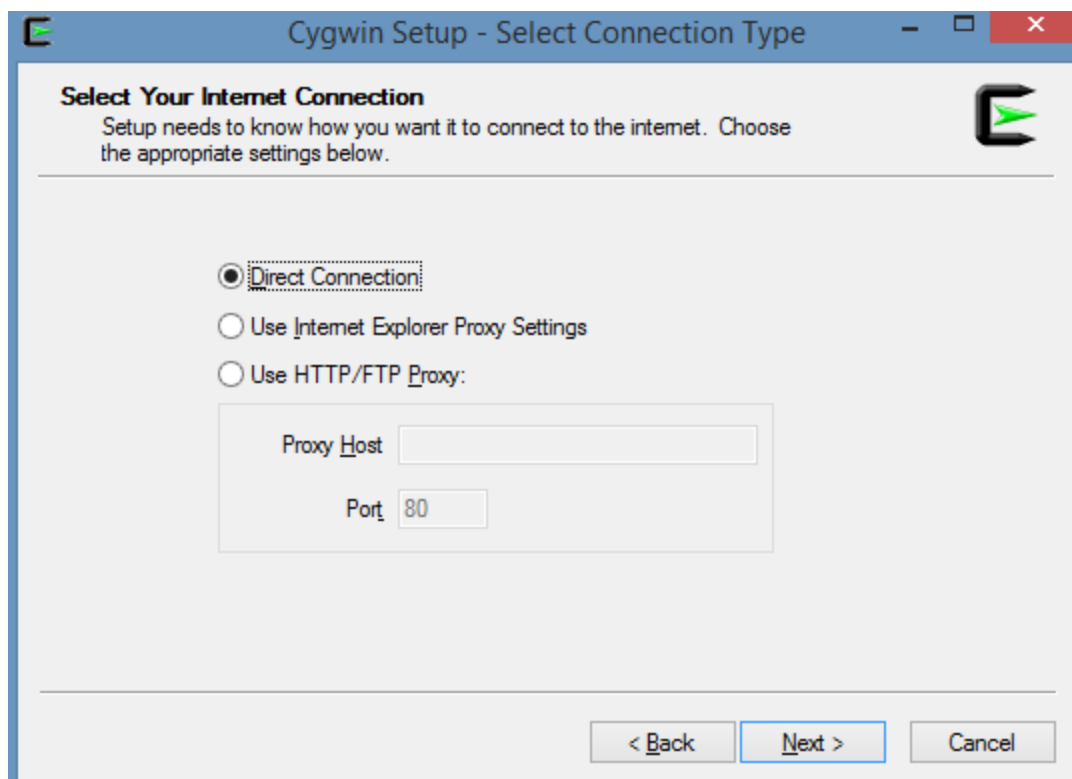
next



next



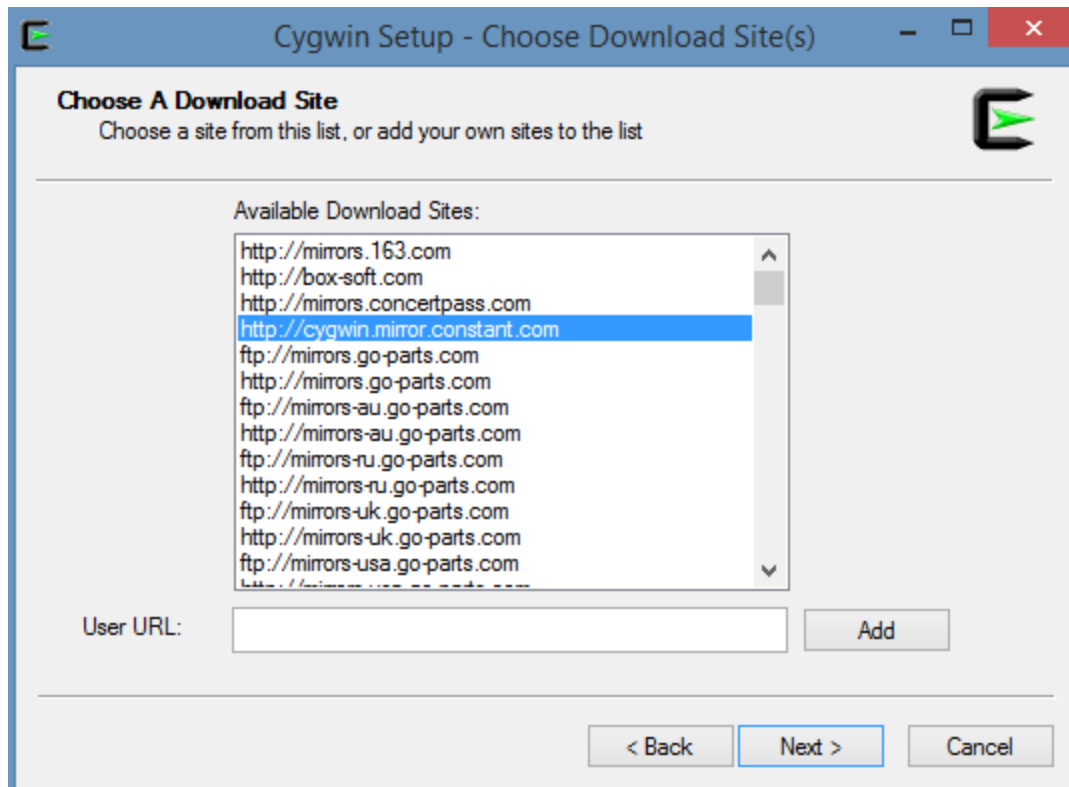
next



next.



Select a mirror site.



next

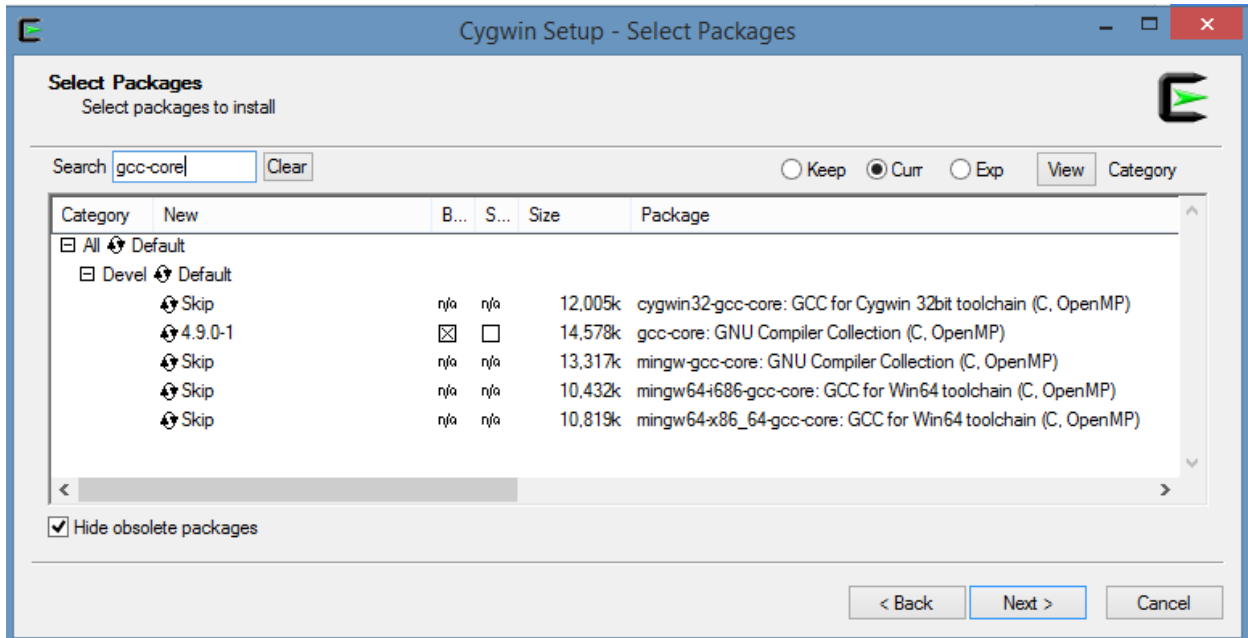
Now you should have reached the “select package” page.

Search the following items and download them :

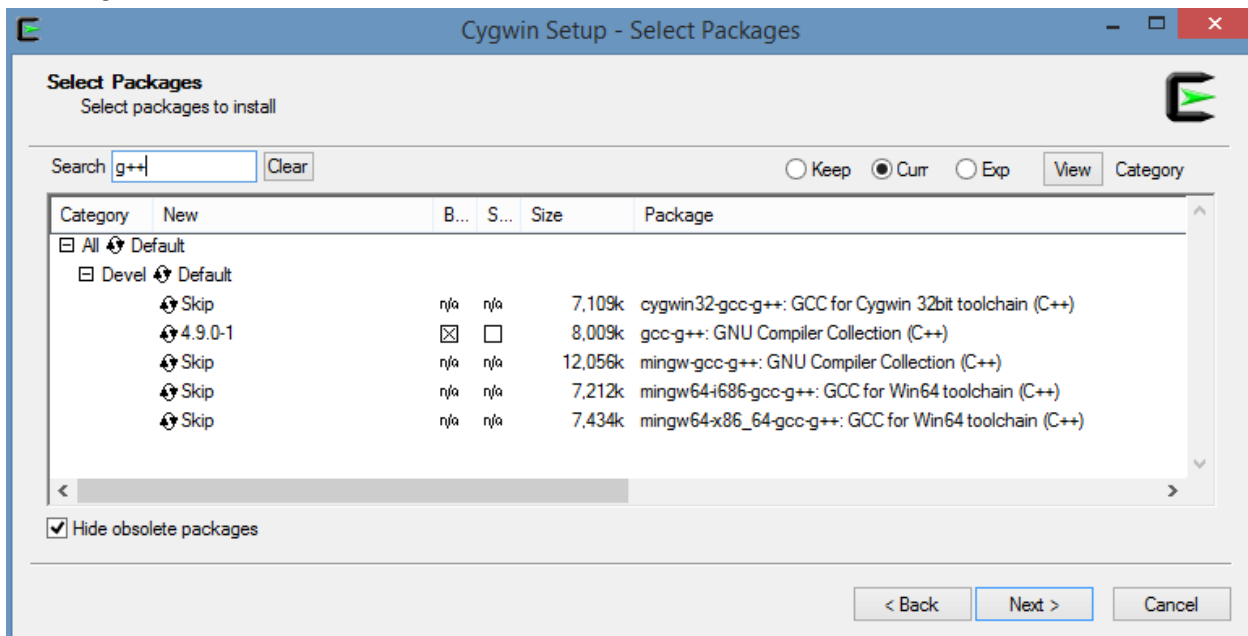
(Type these key words in the search box. Then go to Devel -> to select the corresponding items provided by GNU)

(Click the icon to the left of “skip” to download, instead of skip. When you click, the software version will show up. Feel free to choose the latest version.)

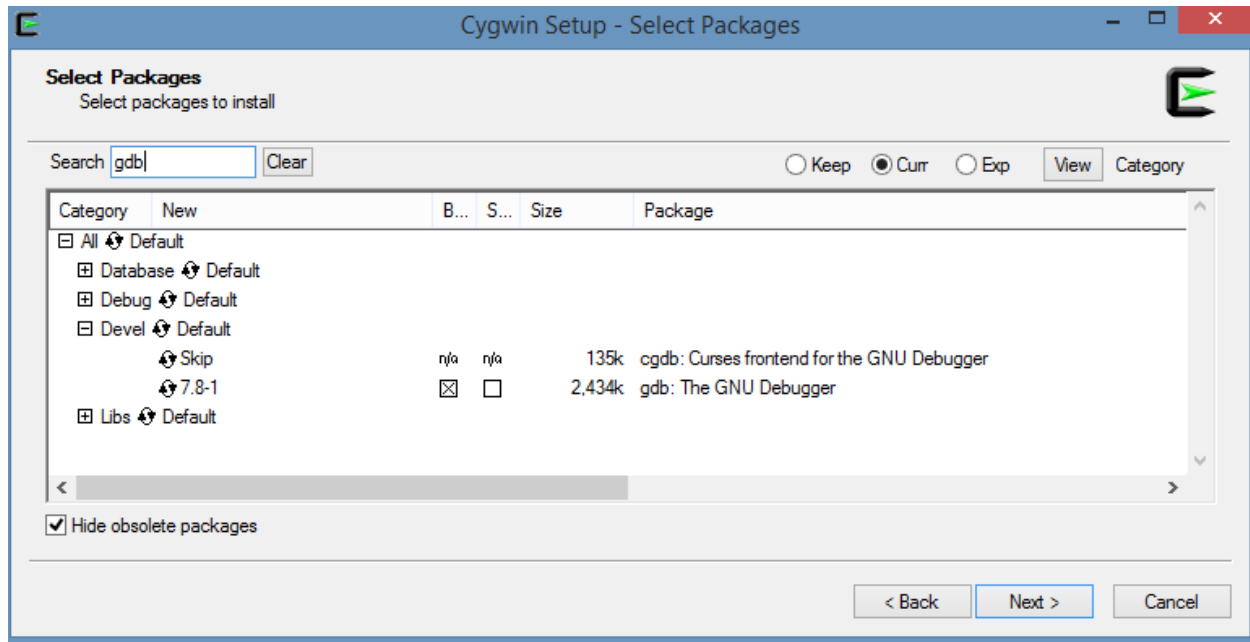
- gcc-core



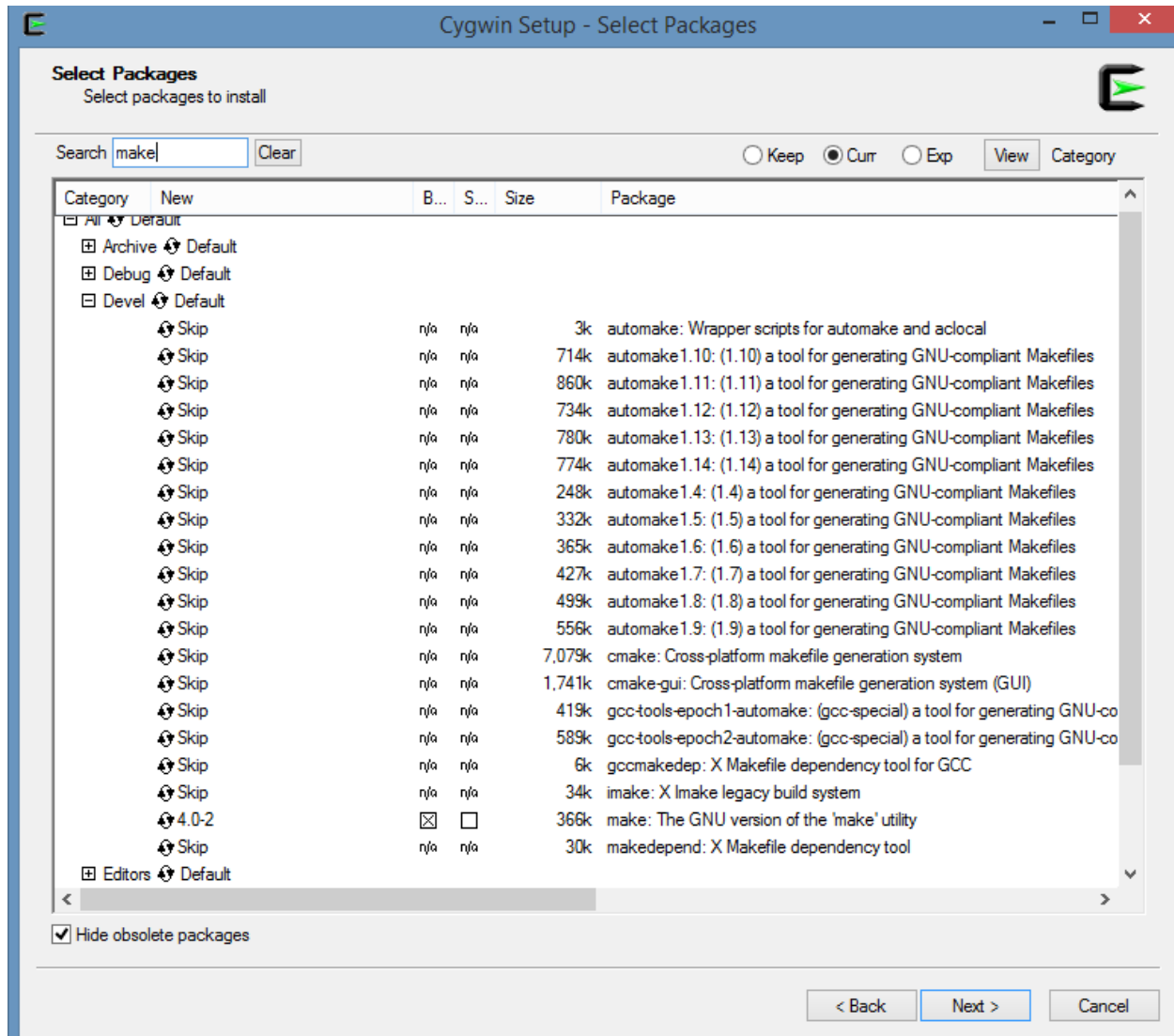
- g++



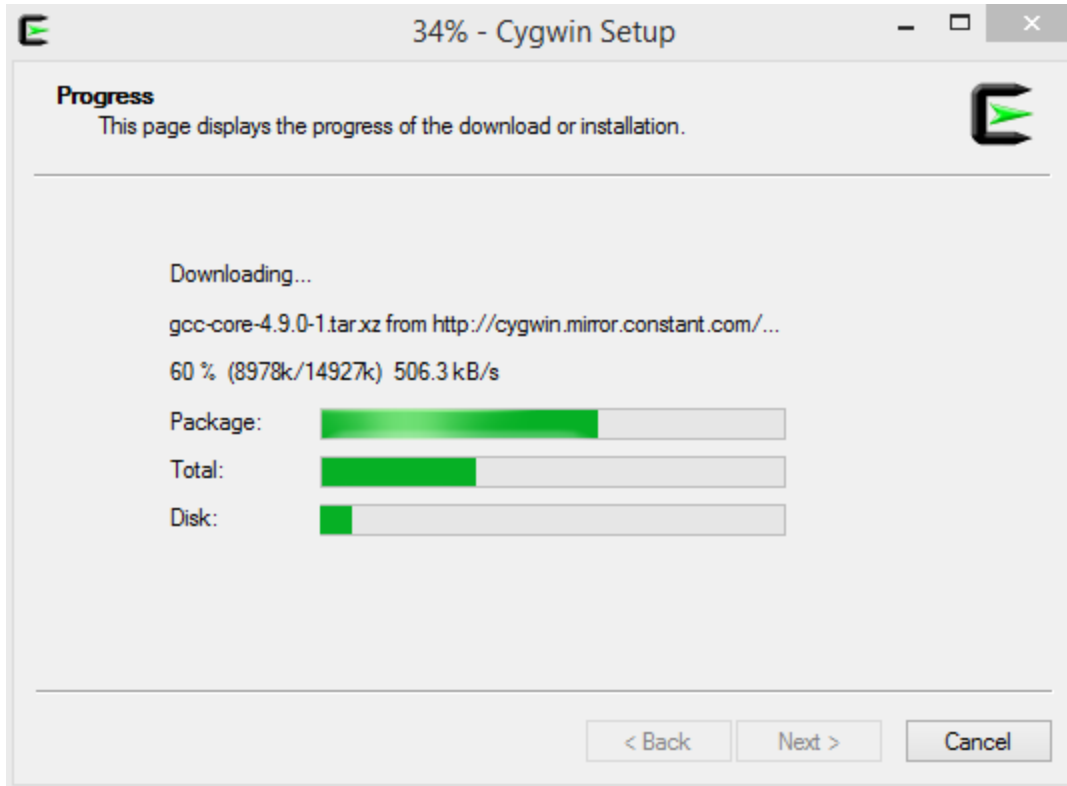
- gdb



- make



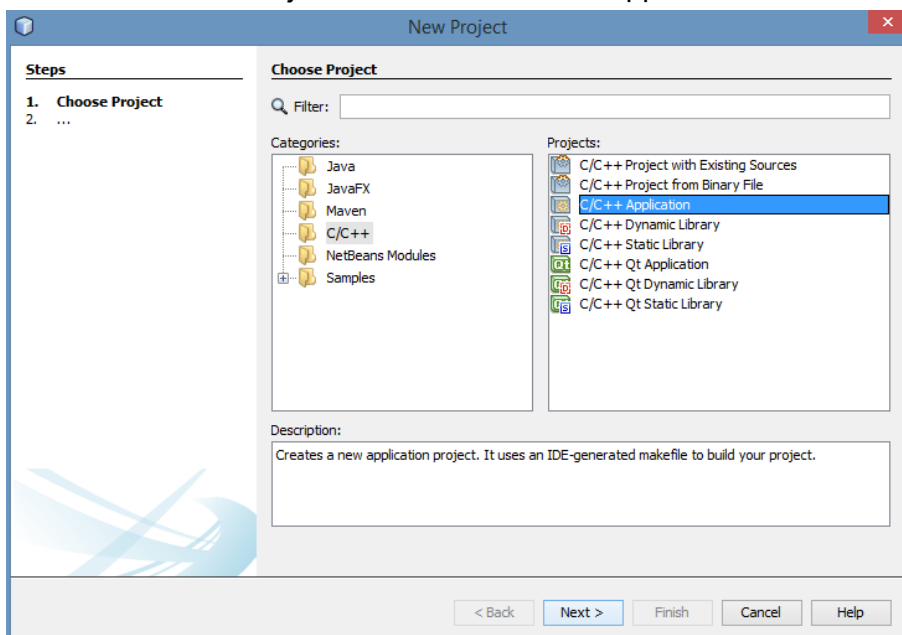
Then, click Next. Then click Next again. You should see it started downloading:



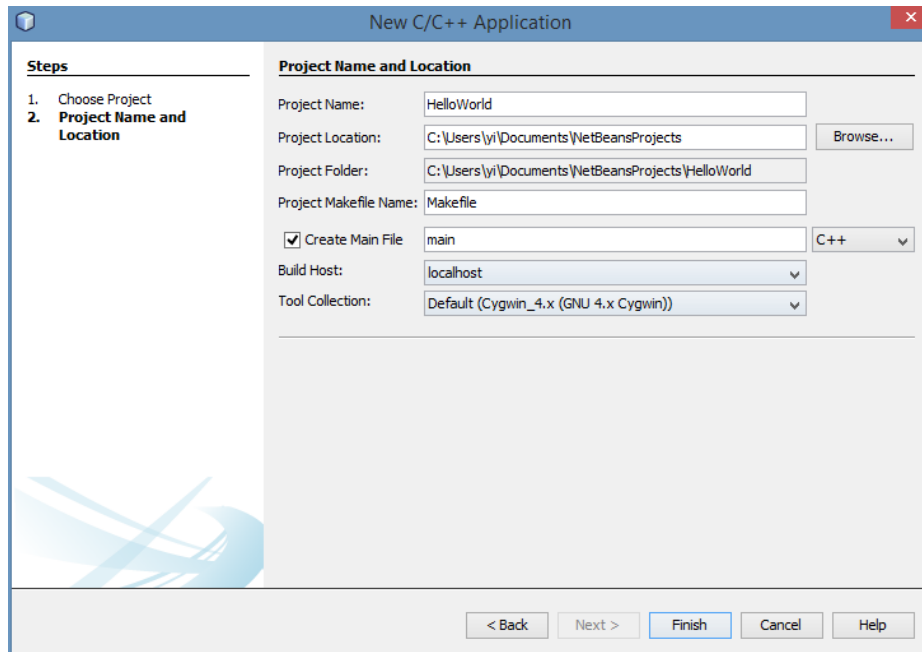
Then click finish when done.

## 8. Verify Netbeans installation

- Open Netbeans.
- File->New Project -> C/C++ -> C/C++ Application

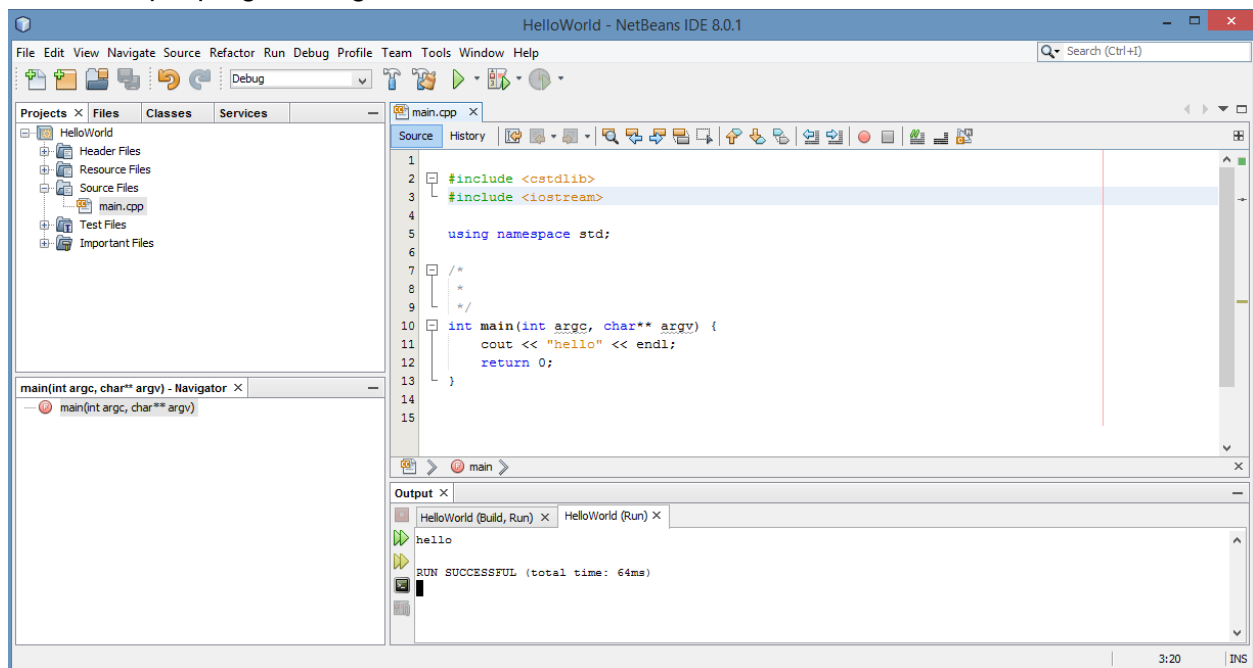


- Next



- Finish

Run a sample program, e.g.,



- Done.

# Class 1

## Sign In Here Please

Yi Wang  
Shi Zong  
Anna Yuan  
Chunyang  
Yi Sun  
Xin Jin  
Zaiqing Qu  
Yanni Wang  
Xudong Bai  
Liguo Yang  
Xudong Bai

## Questions? Type here

1. 老师，在13页example中这两个例子为什么上面一个的string 是data segment，下面那个的string是on stack呢？

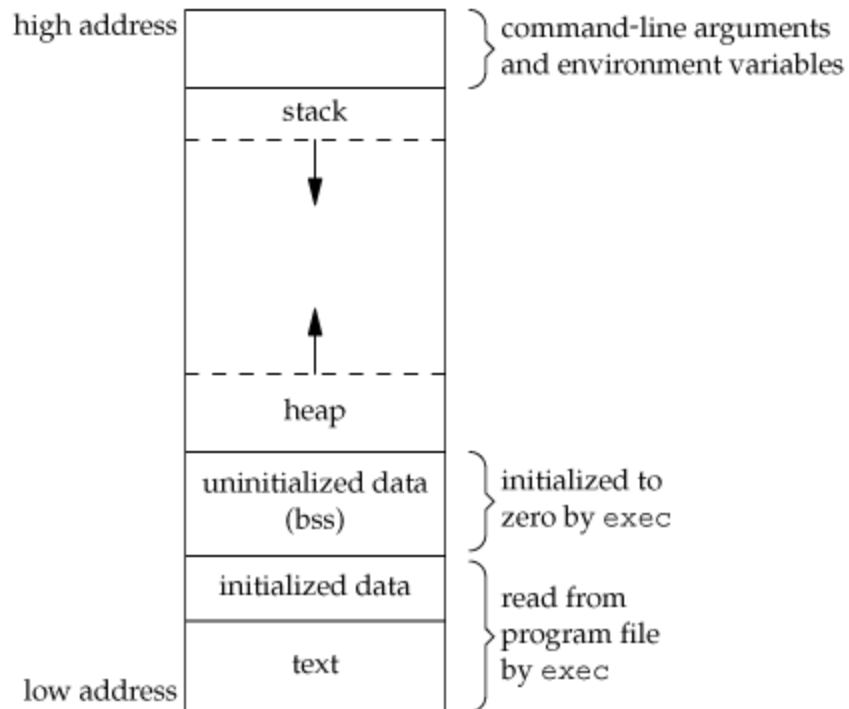
```
char* s = "abc";      // s (pointer) on stack, string: data segment.
```

```
char[] t = "xyz";     // both the pointer and the string are on stack. This is the better practice.
```

A: Compiler是这样定义的，这个记住就好，没什么太深的原理。

## Content

### C/C++ program memory layout



From high address to low address:

- stack
  - Stores local scope variables (i.e., within any function scope, including main)
  - Once function runs out of scope, memory will be deallocated.
  - If variable type belongs to a class, the class destructor will be called.

```
int main() {  
    int x = 5;  
    f();  
}
```

```
struct Node {  
    int value;  
    Node* next;  
};
```

```
void f() {  
    int a = 3;  
    Node s;           // stack  
    Node* t = new Node(); // heap
```



```

    int * b = new int;
    delete b;
}

```

- heap
  - Stores dynamically allocated variable through keyword “new”.
  - In order to deallocate the memory, a corresponding “delete” must be called.
- data segment
  - Stores constant variables
  - Further divided into
    - uninitialized data segment
    - initialized data segment
- text segment
  - stores code

### Examples:

```
#include <stdio>
```

```
int x = 5; // data segment (initialized)
```

```
const int x2 = 10; // data segment (initialized)
```

```
static int x3 = 20; // data segment (initialized)
```

```
int y; // data segment (uninitialized)
```

```
int main (int argc, char** argv) {
```

```
    int y2 = 5; // stack
```

```
    int* z = new z[10]; // heap
```

```
    char* s = “abc”; // s (pointer) on stack, string: data segment.
```

```
    char[] t = “xyz”; // both the pointer and the string are on stack. This is the better practice.
```

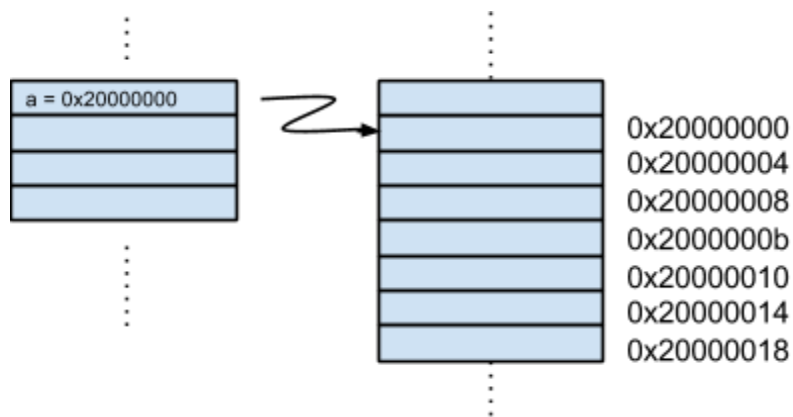
```
    delete z; // deallocating memory.
```

```
}
```

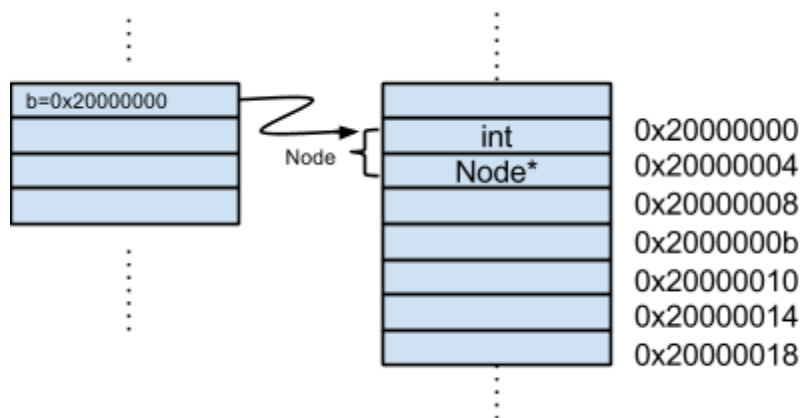
### Pointer/reference, pass by reference vs. value

#### Declare a pointer:

```
int* a; // Pointer to an int. The pointer will be pointing to some random address if
you don't initialize it.
```

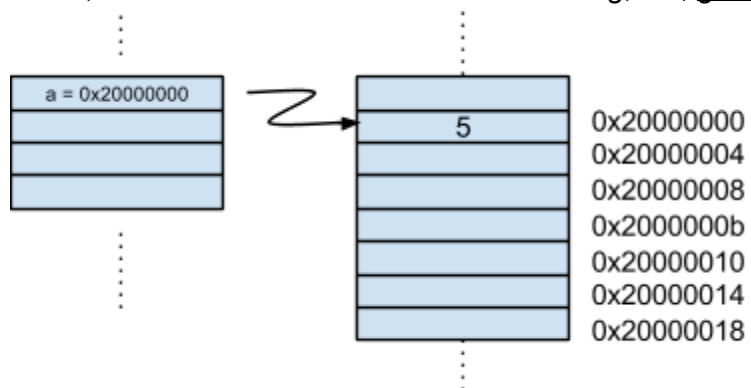


```
struct Node {
    int value;
    Node* next;
};
Node* b;           // Pointer to a complex data structure, i.e., Node
```

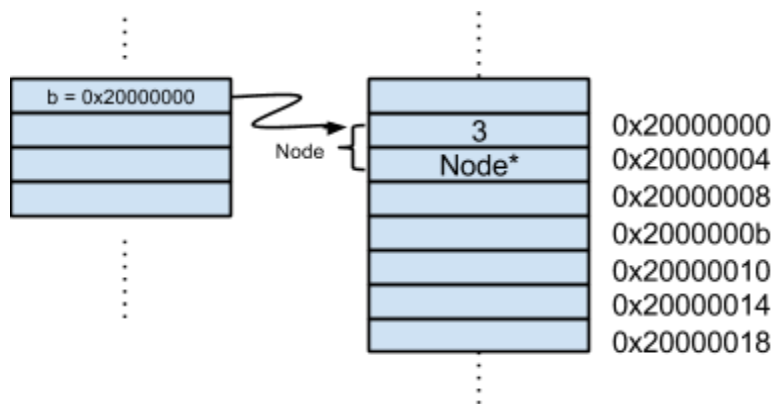


### Usage:

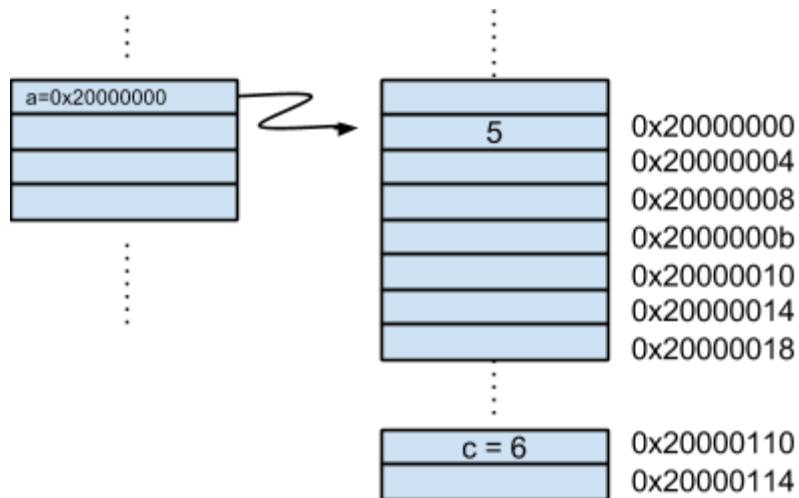
```
*a = 5;           // "*" means dereferencing, i.e., get the content of the pointer
```



```
(*b).value = 3;
b->value = 3;     // "->" is the way to access a field of a data structure based on its pointer
b->next = NULL;
```

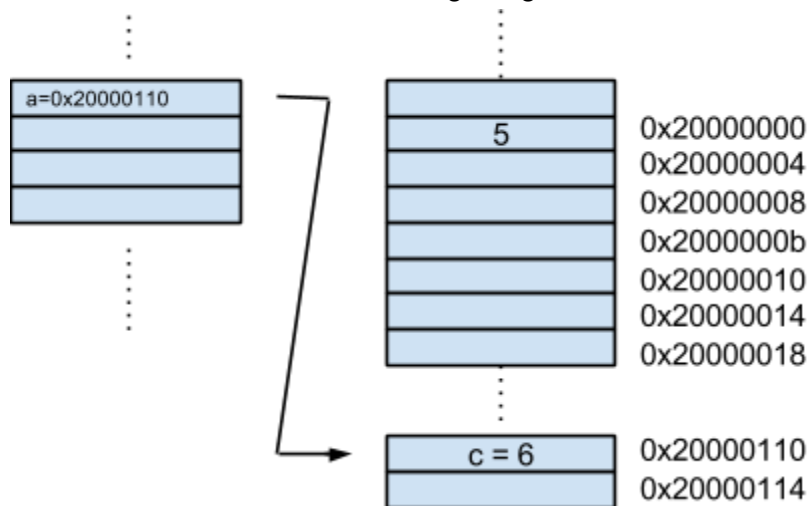


`int c = 6;`



`a = &c;`

// "&" means getting the address of some content.



**Best Practice:**

Do initialize your pointer!!

```

int* a = NULL;
if (a != NULL) {
    // do stuff
}
int* a = new int;           // Allocating heap to store content
Node* b = NULL;
Node* b = new Node;        // Allocating heap to store content

```

### Summary of pointer related operators

	Type definition	Function definition (in function argument)	regular statement
&	N/A	pass by reference	get address of variable
*	define as a pointer	indicating pointer type	dereferencing: get content pointed by that pointer
->	N/A	N/A	access field of pointer

### Pass by value (copy) vs. pass by reference

pass by reference: using key operator "&". "Any change you make to the reference will be reflected on the original value"

```

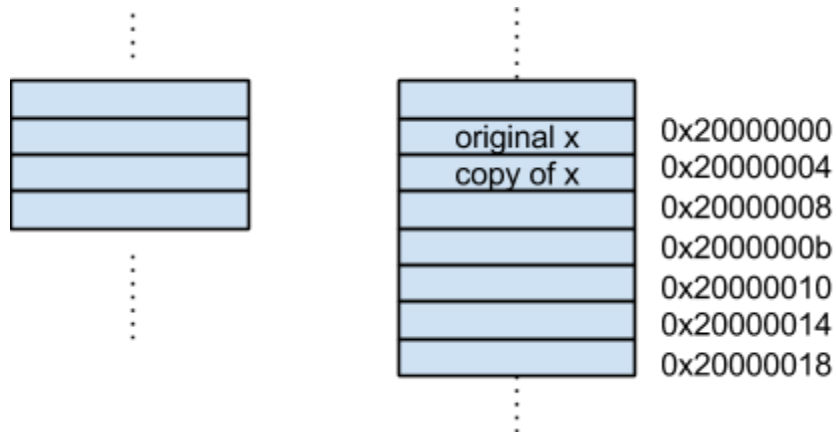
int main () {
    int x = 5;
    f1(x); f2(x); f3(&x); f4(&x);    // Run one of f1, f2, f3, f4
    cout << x;    // what will x be?
}
void f1(int x) { x = 10; }
void f2(int &x) {x = 10;}
void f3(int* x) {*x = 10;}
void f4(int* &x) {*x = 10;}

```

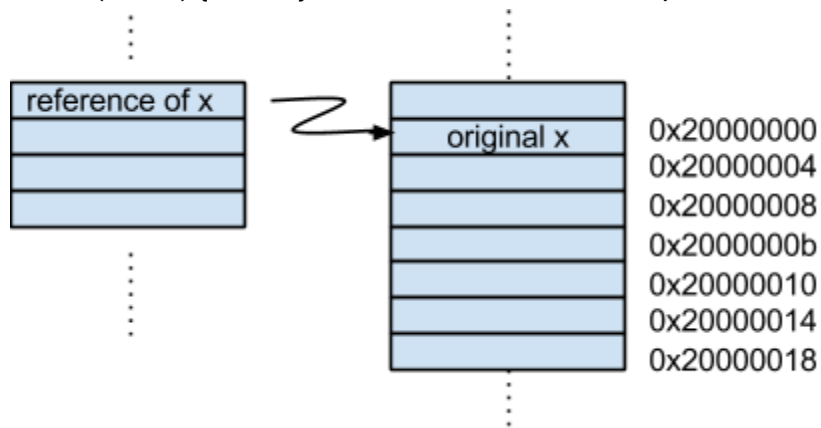
&: regular statement. Take address of variable.

&: Function definition (function argument): Pass this variable as reference.

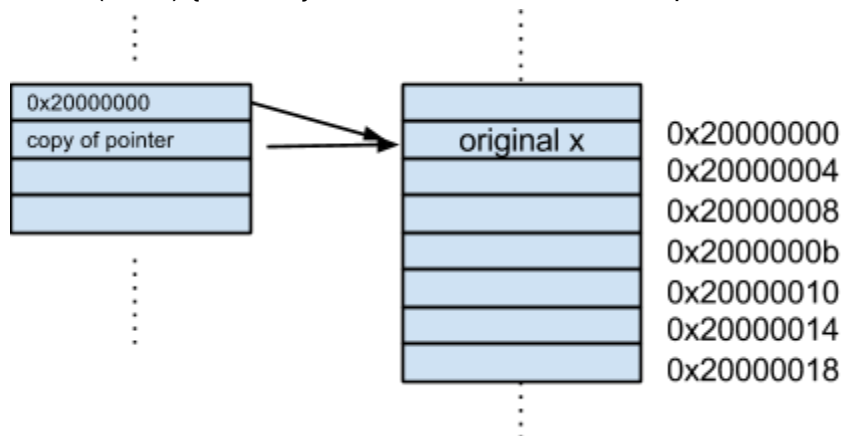
void f1(int x) { x = 10; } // the main function will print 5



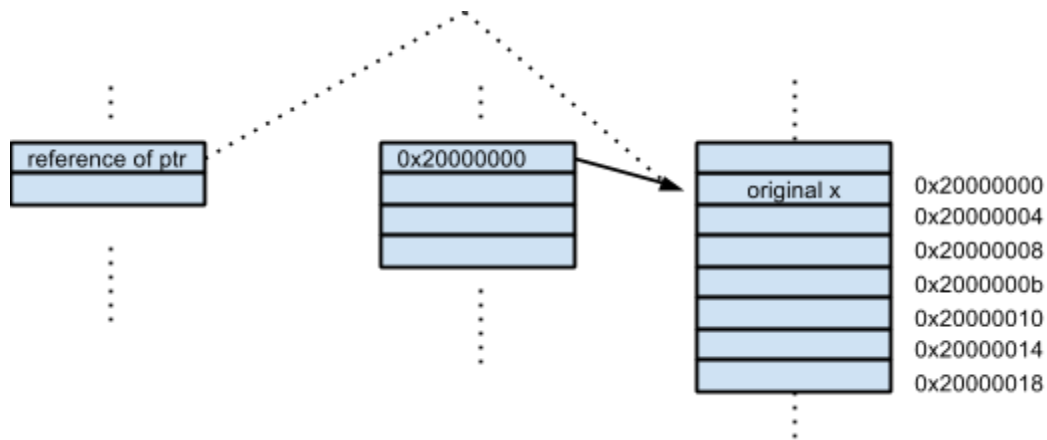
void f2(int &x) { x = 10; } // the main function will print 10



void f3(int\* x) {\*x = 10;} // the main function will print 10



void f4(int \*&x) {\*x = 10;} // the main function will print 10



## Array (1D) - the traditional way

Unknown size initialization

```
int a[] = {3, 4, 1, 7, 2, 9, 5};
```

How can you get the size of the array in your program?

```
int length = sizeof(a) / sizeof(int);
```

How about we know how many elements are there

```
int a[2] = {0, 0};
```

```
int a[3] = {0};           // simple way to initialize to zero.
```

```
int a[3] = {1};          // wrong
```

```
int a[3];
```

```
int a[];                  // wrong
```

```
void myPrintArray_stack(int a[], int length) {
    int len = sizeof(a) / sizeof(int);    // wrong
    for (int i = 0; i < length; ++i) {
        cout << a[i] << " ";
    }
}
```

```
void myPrintArray_stack(int* a, int length) {}    // this is correct, too.
```

```
int length = 3;
```

```
int* a = new int[length];
```

```
for (int i = 0; i < 3; ++i) {
```

```
    a[i] = i;
```

```
}
```

```
// {0, 1, 2}.
```

```
void myPrintArray_heap(int* a, int length) {}
```

```
void myPrintArray_heap(int a[], int length) {}           // wrong.
```

// again, don't forget to release memory!

**c++ stl: "vector"**

<http://www.cplusplus.com/reference/vector/vector/>

```
vector<int> a;           // declare a as an array of integers.
vector<Node*> b;         // array of pointers to Node structure
a.push_back(1);         // {1}
a.push_back(2);         // {1, 2}
a.push_back(3);         // {1, 2, 3}
a.pop_back();           // {1, 2}
cout << a[0];           // printed: 1
cout << a[1];           // printed: 2
```

**When passing a vector to a function, you don't need the "length".**

=> You can get it by calling a.size();

Some sample usages:

```
int a[] = {1, 2, 3, 4, 5};
vector<int> v_a(a, a + sizeof(a) / sizeof(int));
```

```
for (vector<int>::iterator it = v_a.begin(); it != v_a.end(); it++) {
    cout << *it << endl;
}
```

## Homework practices (class 1)

```
#include <cstdlib>
#include <iostream>
#include <vector>
```

```
using namespace std;
```

```
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```
void selectionSort(vector<int>& a){
```

```

if (a.size() == 0) return;
int len = a.size();
for (int i = 0; i < len - 1; ++i) {
    int min_idx = i;
    int min = a[i];
    for (int j = i + 1; j < len; ++j) {
        if (a[j] < min) {
            min = a[j];
            min_idx = j;
        }
    }
    swap(a[i], a[min_idx]);
}
}

```

```

void printVector(vector<int> a) {
    int length = a.size();
    for (int i = 0; i < length; i++) {
        cout << a[i] << " ";
    }
    cout << endl;
}

```

```

void test_selectionSort() {
    cout << "testing selection sort" << endl;
    int a[] = {};
    vector<int> v_a(a, a + sizeof(a) / sizeof(int));
    selectionSort(v_a);
    printVector(v_a);
    int b[] = {1};
    vector<int> v_b(b, b + sizeof(b) / sizeof(int));
    selectionSort(v_b);
    printVector(v_b);
    int c[] = {1, 2, 3, 4};
    vector<int> v_c(c, c + sizeof(c) / sizeof(int));
    selectionSort(v_c);
    printVector(v_c);
    int d[] = {4, 3, 2};
    vector<int> v_d(d, d + sizeof(d) / sizeof(int));
    selectionSort(v_d);
    printVector(v_d);
    int e[] = {5, 2, 3, 7, 1, 4, 6};
    vector<int> v_e(e, e + sizeof(e) / sizeof(int));
}

```



```

    selectionSort(v_e);
    printVector(v_e);
}

```

```

void merge(int* a, int left, int right, int mid) {
    int sorted[right-left+1];
    int i=left, j=mid+1;
    int idx = 0;
    while(i<=mid&& j<=right) {
        if(a[i]<a[j]) {
            sorted[idx]=a[i];
            idx=idx+1;
            i=i+1;
        } else {
            sorted[idx]=a[j];
            idx=idx+1;
            j=j+1;
        }
    }
    while(i<=mid) {
        sorted[idx]=a[i];
        idx=idx+1;
        i=i+1;
    }
    while(j<=right) {
        sorted[idx]=a[j];
        idx=idx+1;
        j=j+1;
    }
    for(int k=left, q=0; k<=right; k++, q++) {
        a[k]=sorted[q];
    }
}

```

```

void mergesort(int* a, int left, int right){
    if(left<right) {
        int mid=(left+right)/2;
        mergesort(a, left, mid);
        mergesort(a, mid+1, right);
        merge(a, left, right, mid );
    }
}

```

```

void printArray(int a[], int length) {
    for (int i = 0; i < length; i++) {
        cout << a[i] << " ";
    }
    cout << endl;
}

```

```

void test_mergeSort() {
    cout << "testing merge sort" << endl;
    int a[] = {};
    int length_a = sizeof(a) / sizeof(int);
    mergesort(a, 0, length_a - 1);
    printArray(a, length_a);
    int b[] = {1};
    int length_b = sizeof(b) / sizeof(int);
    mergesort(b, 0, length_b - 1);
    printArray(b, length_b);
    int c[] = {1, 2, 3, 4};
    int length_c = sizeof(c) / sizeof(int);
    mergesort(c, 0, length_c - 1);
    printArray(c, length_c);
    int d[] = {4, 3, 2};
    int length_d = sizeof(d) / sizeof(int);
    mergesort(d, 0, length_d - 1);
    printArray(d, length_d);
    int e[] = {5, 2, 3, 7, 1, 4, 6};
    int length_e = sizeof(e) / sizeof(int);
    mergesort(e, 0, length_e - 1);
    printArray(e, length_e);
}

```

```

void swap2(int *a, int *b) {
    int temp;
    temp = *b;
    *b = *a;
    *a = temp;
}

```

```

void quicksort(int *a, int left, int right){
    if (left >= right)
        return;
    int pivot = (left+right)/2;

```

```

        swap2(&a[pivot], &a[right]);
        int storeIdx = left;
        for (int i = left; i < right; i++) {
            if (a[i] < a[right]) {
                swap2(&a[storeIdx++], &a[i]);
            }
        }
        swap2(&a[storeIdx], &a[right]);
        quicksort(a, left, storeIdx-1);
        quicksort(a, storeIdx+1, right);
    }
}

```

```

void test_quickSort() {
    cout << "testing quick sort" << endl;
    int a[] = {};
    int length_a = sizeof(a) / sizeof(int);
    quicksort(a, 0, length_a - 1);
    printArray(a, length_a);
    int b[] = {1};
    int length_b = sizeof(b) / sizeof(int);
    quicksort(b, 0, length_b - 1);
    printArray(b, length_b);
    int c[] = {1, 2, 3, 4};
    int length_c = sizeof(c) / sizeof(int);
    quicksort(c, 0, length_c - 1);
    printArray(c, length_c);
    int d[] = {4, 3, 2};
    int length_d = sizeof(d) / sizeof(int);
    quicksort(d, 0, length_d - 1);
    printArray(d, length_d);
    int e[] = {5, 2, 3, 7, 1, 4, 6};
    int length_e = sizeof(e) / sizeof(int);
    quicksort(e, 0, length_e - 1);
    printArray(e, length_e);
}

```

```

int main(int argc, char** argv) {
    test_selectionSort();
    test_mergeSort();
    test_quickSort();
    return 0;
}

```

## Class 2

### Sign in here please

Yi Wang

Yi Sun

Yi Xuan

Anna Yuan

Chunyang Chi

Xudong Bai

Yuren

Liguo Yang

Yanni Wang

### Question - please type here

## FAQ

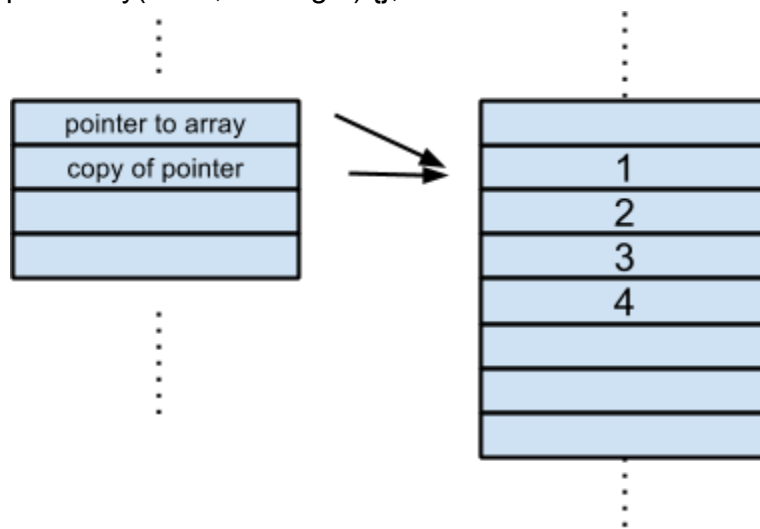
### Passing array vs. passing vector

- Array is always passed as a pointer.
- Vector, like others, is passed by value (by default).
  - Unless you use “&”, to make it pass by reference

```
int a[] = {1, 2, 3, 4};
```

```
printArray(int a[], int length) {} same as:
```

```
printArray(int *a, int length) {};
```



Question:

```
int a[] = {1, 2, 3, 4};
int length = sizeof(a) / sizeof(int);
void changeArray(int a[], int length) {
    a[1] = 3;
}
```

// Question: Array becomes?

```
int a[] = {1, 2, 3, 4};
vector<int> v_a(a, a + sizeof(a) / sizeof(int));
void changeVector(vector<int> v_a) {
    v_a[1] = 3;
}
```

// Question: Vector becomes? How about using "&"

### **++i vs. i++**

When not in a loop, YES, the difference is big. ++i does the "plus" first, then evaluate the value. "i++" does the evaluation first, then does the "plus one".

```
int a = 3;
cout << a++; // Question 3
cout << a + 4; // Question 8
```

```
int b = 3;
cout << ++b; // Question 4
cout << b + 4; // Question 8
```

In a loop, not much difference.

```
for (int i = 0; i < 10; ++i);
```

**Prefer to use ++i instead of i++ in a loop**, since the latter requires the compiler to generate code that creates an extra temporary variable. (both the previous and the new values need to be held since both may be evaluated somewhere).

### **Best practice of writing code & test - how to arrange files?**

Have your header files. Then include whatever you want to use in your main() file.

However, for practice speed purpose, feel free to put everything in one single file and just call the function you need in main().

See live screen IDE example.

### **How to create graph d/s:**

```
class GraphNode {
public:
```

```

GraphNode(int v) : value(v){};
int value;
vector<GraphNode*> neighbors;
}

```

```

1 - 2
| \
3  4

```

```

GraphNode n1(1);
GraphNode n2(2);
GraphNode n3(3);
GraphNode n4(4);

```

```

n1.neighbors.push_back(&n2);
n1.neighbors.push_back(&n3);
n1.neighbors.push_back(&n4);

```

```

n2.neighbors.push_back(&n1);

```

```

n3.neighbors.push_back(&n1);

```

```

n4.neighbors.push_back(&n1);
// Question: is the above graph directed, or undirected?

```

```

int a[4][4] = {{0, 1, 1, 1},
               {1, 0, 0, 0},
               {1, 0, 0, 0},
               {1, 0, 0, 0}};

```

Adjacency list vs. matrix. Question: why?

**Matrix:** fixed  $n^2$  space.

**Adjacency list:** could save space for sparse graph.

**Matrix:** support quick look up if one edge exists.

**Adjacency list:** doesn't support quick look up if one edge exists.

## Content

### Array vs. LinkedList:

- Memory Layout?
- Resizing (append at the end) - which one is better?
- Add/remove random element, which one is better?

- (Random) access time?
- Search time (non-sorted)?
- Search time (sorted) ? array:  $O(\lg N)$ , list:  $O(N)$

Practice question: create a linked list. Write a function to insert a value to the front. What's wrong with the below code?

```
struct Node {
    int value_;
    Node* next_;
    Node(int v) : value_(v), next_(NULL) {};
};

void insertInFront(int value, Node*& head) {
    Node* node = new Node(value);
    node->next_ = head;
    head = node;
}

Node* insertInFront(int value, Node* head) {
    Node* node = new Node(value);
    node->next_ = head;
    head = node;
    return head;
}
```

## C++ stl "sort"

```
#include <algorithm>
#include <string.h>
```

```
int a[] = {3, 6, 1, 2, 8, 9, 4};
vector<int> v(a, a + sizeof(a) / sizeof(int));
sort(v.begin(), v.end());
```

```
bool myComp(int left, int right) {
    return left > right;
}
```

```
template<typename T>
void printVector(vector<T> v) {
    for (typename vector<T>::iterator it = v.begin(); it != v.end(); ++it) {
```

```

        cout << *it << " ";
    }
    cout << endl;
}

```

Functor: A functor is a class that defines the operator(). That lets you create objects which "look like" a function.

```

struct myfunctor {
    x;
    myfunctor(int v) : x_(v) {};
    int operator() (int y) { return x + y; }
}

```

```

class myCompClass {
public:
    bool operator() (int left, int right) {
        return left > right;
    }
} myCompObject;

```

```

bool myCompStr(string left, string right) {
    if (strcmp(left.c_str(), right.c_str()) < 0) return true;
    else return false;
}

```

```

string s[] = {"c", "asdf", "uuu", "z", "bbn"};
vector<string> vs(s, s + sizeof(s) / sizeof(string));
sort(vs.begin(), vs.end(), myCompStr);
pntVector(vs);

```

### c++ stl "priority\_queue"

```

#include <queue>

```

```

int a[] = {4, 3, 7, 2, 9, 0, 6, 1, 8, 5};
priority_queue<int> pq;
for (int i = 0; i < sizeof(a) / sizeof(int); ++i) {
    pq.push(a[i]);
}
while (!pq.empty()) {
    cout << pq.top() << " ";
}

```



```

    pq.pop();
}

```

```

class compClass {
public:
    bool operator() (int left, int right) {
        return left > right;
    }
};

```

```

priority_queue<int, vector<int>, compClass> mypq;
for (int i = 0; i < sizeof(a) / sizeof(int); ++i) {
    mypq.push(a[i]);
}
while (!mypq.empty()) {
    cout << mypq.top() << " ";
    mypq.pop();
}

```

question: a school of student. Input: their name and age. find the 5 oldest people with their names.

```

vector<pair<int, string> > v;
v.push_back(make_pair(20, "a"));
v.push_back(make_pair(30, "b"));
v.push_back(make_pair(40, "c"));
v.push_back(make_pair(10, "d"));
v.push_back(make_pair(15, "e"));
v.push_back(make_pair(50, "f"));
v.push_back(make_pair(25, "g"));
v.push_back(make_pair(16, "g"));
v.push_back(make_pair(17, "g"));
v.push_back(make_pair(18, "g"));
v.push_back(make_pair(19, "g"));

```

```

priority_queue<pair<int, string>, vector<pair<int, string> >, studentComp> pq;
int limit = 5;
for (int i = 0; i < v.size(); ++i) {
    pq.push(v[i]);
    if (pq.size() > limit) pq.pop();
}

```

```

while (!pq.empty()) {
    cout << pq.top().first << " " << pq.top().second << endl;
    pq.pop();
}

class studentComp {
public:
    bool operator() (pair<int, string> left, pair<int, string> right) {
        return left.first > right.first;
    }
};

```

## Homework practices (class 2)

### Binary Search (recursion)

```

int binarySearch(int a[], int num, int left, int right) {
    if (left > right) return -1;
    int mid = left + (right - left) / 2;
    if (a[mid] == num) return mid;
    else if (a[mid] < num) return binarySearch(a, num, mid + 1, right);
    else return binarySearch(a, num, left, mid - 1);
}

```

```

void testBinarySearch() {
    int a[] = {};
    int length_a = sizeof(a) / sizeof(int);
    cout << binarySearch(a, 3, 0, length_a - 1) << endl;

    int b[] = {1};
    int length_b = sizeof(b) / sizeof(int);
    cout << binarySearch(b, 3, 0, length_b - 1) << endl;
    cout << binarySearch(b, 1, 0, length_b - 1) << endl;

    int c[] = {1, 2, 3, 4};
    int length_c = sizeof(c) / sizeof(int);
    cout << binarySearch(c, 3, 0, length_c - 1) << endl;

    int d[] = {1, 3, 6, 8, 9, 11, 15};
}

```

```

    int length_d = sizeof(d) / sizeof(int);
    cout << binarySearch(d, 3, 0, length_d - 1) << endl;
}

```

### Binary Search Variant 1 - find left boarder (Iteration)

```

int binarySearchV1(int a[], int target, int left, int right) {
    int mid;
    while (left < right - 1) {
        mid = left + (right - left) / 2;
        if (a[mid] == target) {
            right = mid; // do not stop here, keep checking to left
        } else if (a[mid] < target) {
            left = mid;
        } else {
            right = mid;
        }
    }
    if (a[left] == target) // check a[left] against target first
        return left;
    if (a[right] == target) // then check a[right] against target
        return right;
    return -1;
}

```

```

void testBinarySearchV1() {
    int a[] = {};
    int length_a = sizeof(a) / sizeof(int);
    cout << binarySearchV1(a, 3, 0, length_a - 1) << endl;

    int b[] = {1};
    int length_b = sizeof(b) / sizeof(int);
    cout << binarySearchV1(b, 3, 0, length_b - 1) << endl;
    cout << binarySearchV1(b, 1, 0, length_b - 1) << endl;

    int c[] = {1, 2, 3, 3, 3, 3, 3, 4};
    int length_c = sizeof(c) / sizeof(int);
    cout << binarySearchV1(c, 3, 0, length_c - 1) << endl;

    int d[] = {1, 3, 3, 3, 6, 8, 9, 11, 15};
    int length_d = sizeof(d) / sizeof(int);
    cout << binarySearchV1(d, 3, 0, length_d - 1) << endl;
}

```

```
{1, 2, 3, 3, 3, 3, 4};
```

1. error case: return -1.
2. only one element: {3}. {2} 0, 0. => 1, 0
3. two elements: {3, 3}, {2, 3}.

### Binary Search Variant 1 - find left boarder (recursion)

```
int binarySearchV1_recur(int a[], int num, int left, int right) {  
    if (left > right) return -1;    // done.  
    if (left == right && a[left] == num) return left;    // done  
    if (left == right - 1) {  
        if (a[left] == num) return left;  
        if (a[right] == num) return right;  
    }  
    int mid = left + (right - left) / 2;  
    if (a[mid] == num) return binarySearchV1_recur(a, num, left, mid);  
    else if (a[mid] < num) return binarySearchV1_recur(a, num, mid + 1, right);  
    else return binarySearchV1_recur(a, num, left, mid - 1);  
}
```

```
void testBinarySearchV1_recur() {  
    int a[] = {};  
    int length_a = sizeof(a) / sizeof(int);  
    cout << binarySearchV1_recur(a, 3, 0, length_a - 1) << endl;  
  
    int b[] = {1};  
    int length_b = sizeof(b) / sizeof(int);  
    cout << binarySearchV1_recur(b, 3, 0, length_b - 1) << endl;  
    cout << binarySearchV1_recur(b, 1, 0, length_b - 1) << endl;  
  
    int c[] = {1, 2, 3, 3, 3, 3, 3, 4};  
    int length_c = sizeof(c) / sizeof(int);  
    cout << binarySearchV1_recur(c, 3, 0, length_c - 1) << endl;  
  
    int d[] = {1, 3, 3, 3, 6, 8, 9, 11, 15};  
    int length_d = sizeof(d) / sizeof(int);  
    cout << binarySearchV1_recur(d, 3, 0, length_d - 1) << endl;  
}
```

### Binary Search Variant 2 - find right boarder (iteration)

```

int binarySearchV2(int a[], int target, int left, int right) {
    int mid;
    while (left < right - 1) { //if left neighbors right → terminate
        mid = left + (right - left) / 2;
        if (a[mid] == target) {
            left = mid; // do not stop here, keep checking to right
        } else if (a[mid] < target) {
            left = mid;
        } else {
            right = mid;
        }
    }
    if (a[right] == target)
        return right;
    if (a[left] == target)
        return left;
    return -1;
}

```

```

void testBinarySearchV2() {
    int a[] = {};
    int length_a = sizeof(a) / sizeof(int);
    cout << binarySearchV2(a, 3, 0, length_a - 1) << endl;

    int b[] = {1};
    int length_b = sizeof(b) / sizeof(int);
    cout << binarySearchV2(b, 3, 0, length_b - 1) << endl;
    cout << binarySearchV2(b, 1, 0, length_b - 1) << endl;

    int c[] = {1, 2, 3, 3, 3, 3, 3, 4};
    int length_c = sizeof(c) / sizeof(int);
    cout << binarySearchV2(c, 3, 0, length_c - 1) << endl;

    int d[] = {1, 3, 3, 3, 6, 8, 9, 11, 15};
    int length_d = sizeof(d) / sizeof(int);
    cout << binarySearchV2(d, 3, 0, length_d - 1) << endl;
}

```

### Binary Search Variant 2 - find right boarder (recursion)

```

int binarySearchV2_recur(int a[], int num, int left, int right)
{

```

```

        if (left > right) return -1;
        if (left == right && a[left] == num) return left;
        if (left == right - 1) {
            if (a[right] == num) return right;
            if (a[left] == num) return left;
        }
        int mid = left + (right - left) / 2;
        if (a[mid] == num) return binarySearchV2_recur(a, num, mid, right);
        else if (a[mid] < num) return binarySearchV2_recur(a, num, mid + 1, right);
        else return binarySearchV2_recur(a, num, left, mid - 1);
    }
}

```

```

void testBinarySearchV2_recur() {
    int a[] = {};
    int length_a = sizeof(a) / sizeof(int);
    cout << binarySearchV2_recur(a, 3, 0, length_a - 1) << endl;

    int b[] = {1};
    int length_b = sizeof(b) / sizeof(int);
    cout << binarySearchV2_recur(b, 3, 0, length_b - 1) << endl;
    cout << binarySearchV2_recur(b, 1, 0, length_b - 1) << endl;

    int c[] = {1, 2, 3, 3, 3, 3, 3, 4};
    int length_c = sizeof(c) / sizeof(int);
    cout << binarySearchV2_recur(c, 3, 0, length_c - 1) << endl;

    int d[] = {1, 3, 3, 3, 6, 8, 9, 11, 15};
    int length_d = sizeof(d) / sizeof(int);
    cout << binarySearchV2_recur(d, 3, 0, length_d - 1) << endl;
}

```

Find element in sorted array that has been rotated (shifted) unknown times.

123456789  
 456789123  
 789123456

```

int findElementInRotateSortedArr(int * a, int val, int left, int right) {
    if (left > right) {
        return -1;
    }
    int mid = left + (right - left) / 2;
}

```

```

    if (val == a[mid]) {
        return mid;
    } else if (val < a[mid] && val >= a[left] || val > a[mid] && val > a[right]) {
        findElementInRotateSortedArr(a, val, left, mid - 1);
    } else if (val > a[mid] && val <= a[right] || val < a[mid] && val < a[left]) {
        findElementInRotateSortedArr(a, val, mid + 1, right);
    }
}

void testFindElementInRotateSortedArr() {
    int a[] = {};
    int length_a = sizeof(a) / sizeof(int);
    cout << findElementInRotateSortedArr(a, 3, 0, length_a - 1) << endl;

    int b[] = {1};
    int length_b = sizeof(b) / sizeof(int);
    cout << findElementInRotateSortedArr(b, 3, 0, length_b - 1) << endl;

    int c[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int length_c = sizeof(c) / sizeof(int);
    cout << findElementInRotateSortedArr(c, 3, 0, length_c - 1) << endl;

    int d[] = {6, 7, 8, 1, 2, 3, 4, 5};
    int length_d = sizeof(d) / sizeof(int);
    cout << findElementInRotateSortedArr(d, 3, 0, length_d - 1) << endl;

    int e[] = {4, 5, 6, 7, 8, 1, 2, 3};
    int length_e = sizeof(e) / sizeof(int);
    cout << findElementInRotateSortedArr(e, 3, 0, length_e - 1) << endl;
}

```

**Find element in unknown size (sorted) array.**

```

int findElementInUnknownSizeSortedArray(int a[], int key) {
    int current = 0;
    int previous = 0;
    while (a[current] != -1) {
        if (a[current] == key) return current;
        else if (a[current] > key)
            return binarySearch(a, key, previous, current);
        else {
            previous = current;
        }
    }
}

```

```

        current = 2 * (current + 1) - 1;
    }
}
//otherwise, find the border
int right = binarySearchV1(a, -1, previous, current);
return binarySearch(a, key, previous, right - 1);
}

void testFindElementInUnknownSizeSortedArray() {
    int a[] = {};
    cout << findElementInUnknownSizeSortedArray(a, 3) << endl;
    int b[] = {1};
    cout << findElementInUnknownSizeSortedArray(b, 3) << endl;
    int c[] = {1, 2, 3, 4, 5};
    cout << findElementInUnknownSizeSortedArray(c, 3) << endl;
    int d[] = {1, 2, 3, 4, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1};
    cout << findElementInUnknownSizeSortedArray(d, 3) << endl;
    int e[] = {1, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
    cout << findElementInUnknownSizeSortedArray(e, 3) << endl;
}

```

## Class 3

### Sign In Here Please

### Questions?

const可以理解为read only?

### Content

#### c++ stl “stack”

Commonly used APIs:

empty()

push()

pop()

top()

size()

```
stack<int> myStack;
```

```
myStack.push(2);
```



```
cout << myStack.size() << endl;    // 1
cout << myStack.top() << endl;    // 2
myStack.pop();
assert(myStack.empty() == true);
```

### **c++ stl “queue”**

Commonly used APIs:

```
front()
back()
push()
pop()
size()
empty()
```

```
queue<int> myQueue;
myQueue.push(1);
myQueue.push(2);
myQueue.push(3);
cout << myQueue.front() << endl;    // 1
cout << myQueue.back() << endl;    // 3
```

### **c++ stl “deque”**

Double ended queue that also support random access. (Similar to vectors, but provide APIs to append & remove on both ends.)

Commonly used APIs:

```
size()
empty()
push_back()
push_front()
pop_back()
pop_front()
front()
back()
at()
begin() (supporting iterator)
```

```
deque<int> myDeque;
myDeque.push_front(1);
```

```

myDeque.push_front(2);
myDeque.push_front(3);
myDeque.push_back(1);
myDeque.push_back(2);
myDeque.push_back(3);
for (deque<int>::iterator it = myDeque.begin(); it != myDeque.end(); ++it) {
    cout << *it << " ";
}

cout << endl;
cout << myDeque[3] << endl;;

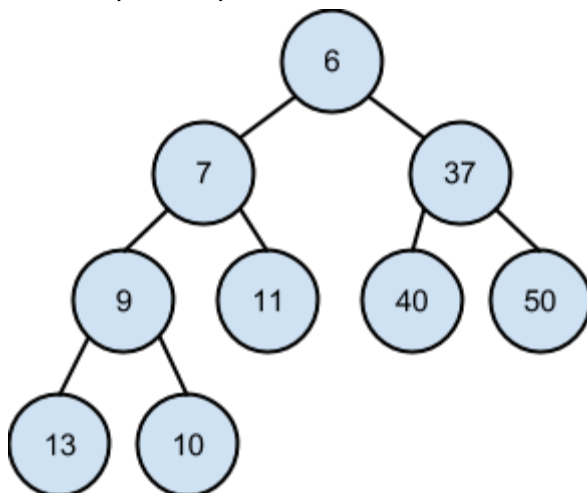
```

## Heap and its implementation

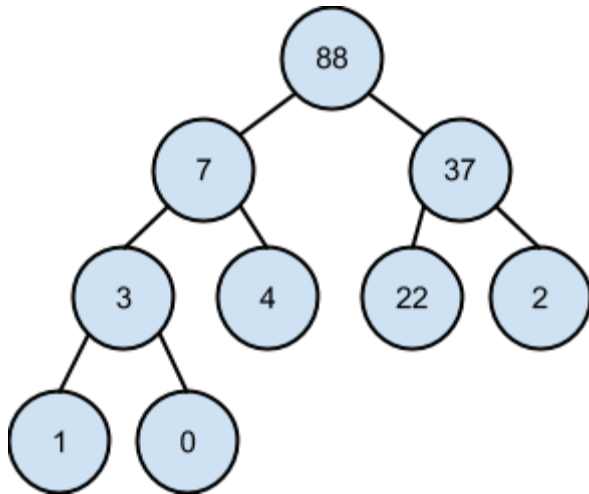
### Definition

- Heap is a tree based data structure
- Across the entire tree, the relation between a parent node and a child node stays consistent.
- Benefit:  $O(1)$  access to min/max item.
- The common implementation of a heap is using a complete binary tree.
  - A “complete binary tree” is a binary tree in which every level, except possibly the last level, is completely filled, and all nodes are as far left as possible.
- Property:
  - $O(1)$  access min (min heap), max(max heap).
  - Height (N nodes in total):  $\lg N$ .

Min Heap Example:



Max Heap Example:



Heapify a sequence of numbers

Input: 3, 22, 4, 2, 1, 0, 88, 7, 37

After heapifying,  $O(1)$  time we can get the maximum value: 88.

Heapify an array: we need to use an array to represent the binary tree. Start from the top of tree, then put the numbers from left to right on each level.

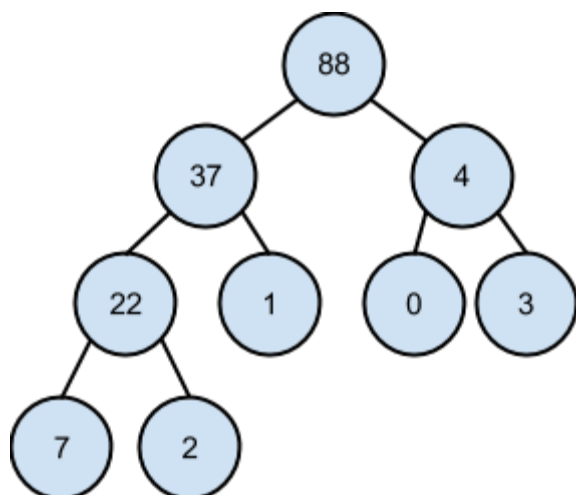
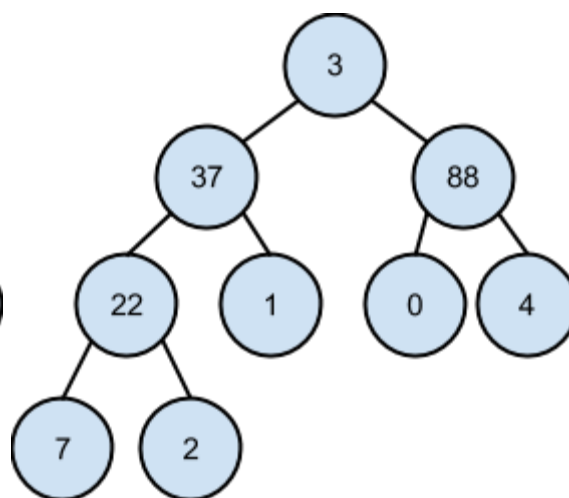
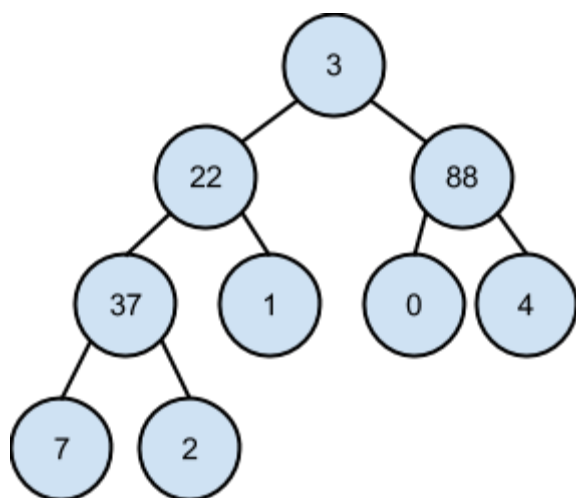
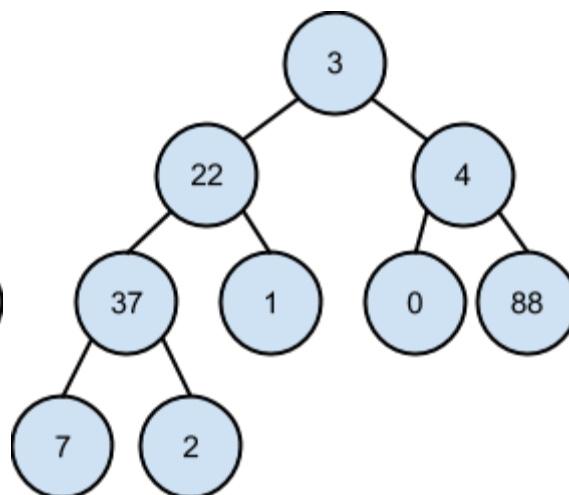
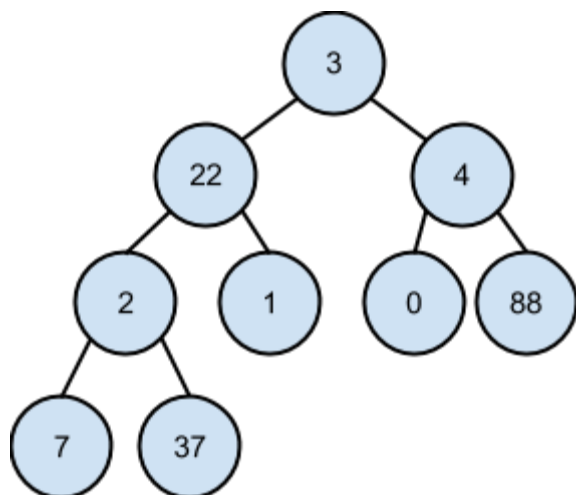
Output: {88, 37, 4, 22, 1, 0, 3, 7, 2}

What does this output array give us?  $O(1)$  access to max item  $\Rightarrow a[0]$ .

Heapify the input sequence would require  $O(N)$   $\Rightarrow$  for each node that has a child, we need to perform FilterDown() action. However, because the filter down distance for each level nodes is bounded  $\Rightarrow O(N)$  complexity.

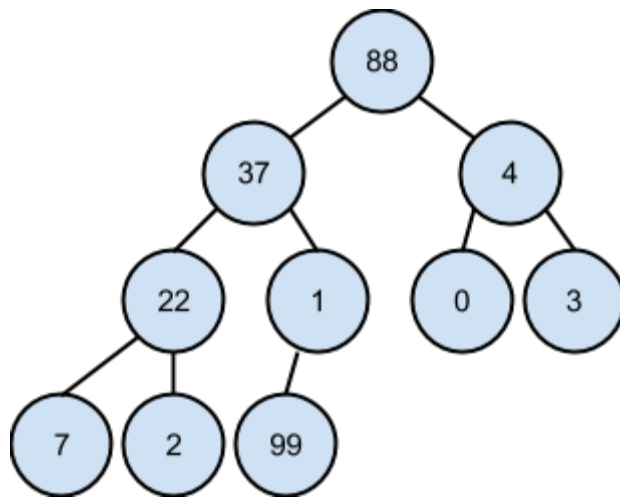
Illustration:

Heapify:

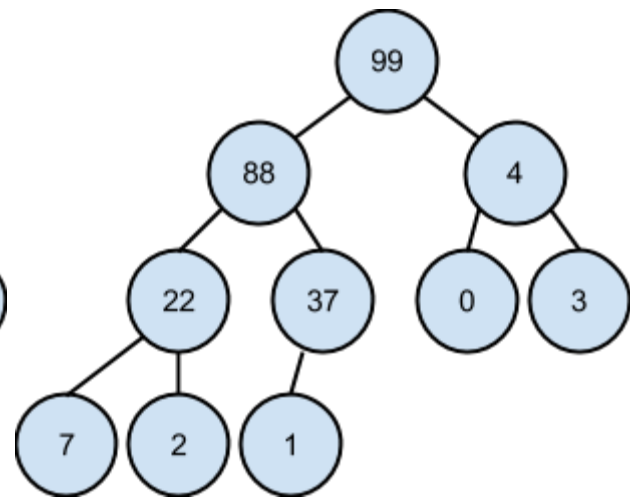


3, 22, 4, 2, 1, 0, 88, 7, 37

Insert:



99 88 4 22 37 0 3 7 2 1



### Implementation

```

/*
 * File: max_heap.h
 *
 */

#ifndef MAX_HEAP_H
#define MAX_HEAP_H

class MaxHeap {
public:
    MaxHeap(int* input, int size);
    int Max() const;
    void Insert(int value);
    void DeleteMax();
private:
    int* hList;
    int heapSize;
    int maxHeapSize;
    void FilterDown(int index);
    void FilterUp(int index);
};

#endif /* MAX_HEAP_H */
  
```

```

#include "max_heap.h"

#define MAXSIZE 100

MaxHeap::MaxHeap(int* input, int size) {
    hList = input;
    heapSize = size;
    maxHeapSize = MAXSIZE;
    int cut = (size - 2) / 2; // cut is the last node index that has a child
    for (int i = cut; i >= 0; --i) {
        FilterDown(i);
    }
}

void MaxHeap::FilterDown(int pos) {
    int curr = pos;
    int child = curr * 2 + 1;
    int temp = hList[curr];
    while (child < heapSize) {
        if (child + 1 < heapSize && hList[child + 1] > hList[child]) {
            child ++;
        }
        if (hList[child] > temp) {
            hList[curr] = hList[child];
            curr = child;
            child = child * 2 + 1;
        } else {
            break;
        }
    }
    hList[curr] = temp;
}

int MaxHeap::Max() const {
    if (heapSize == 0) {
        return -1;
    }
    return hList[0];
}

void MaxHeap::Insert(int value) {

```

```

    if (heapSize == maxHeapSize) {
        return;
    }
    hList[heapSize] = value;
    FilterUp(heapSize);
    heapSize++;
}

```

```

void MaxHeap::FilterUp(int pos) {
    int curr = pos;
    int parent = (pos - 1) / 2;
    int temp = hList[pos];
    while (curr != 0) {
        if (temp > hList[parent]) {
            hList[curr] = hList[parent];
            curr = parent;
            parent = (parent - 1) / 2;
        } else {
            break;
        }
    }
    hList[curr] = temp;
}

```

```

void MaxHeap::DeleteMax() {    // this can be easily extended to deleteIndex(int i).
    if (heapSize == 0) {
        return;
    }
    hList[0] = hList[heapSize - 1];
    FilterDown(0);
    heapSize--;
}

```

**Queue implementation supporting  $O(1)$  push & pop.**

```

struct Node {
    int value;
    Node* next;
}

```

```

    Node* prior;
    Node(int v) : value(v), next(NULL), prior(NULL) {};
};

struct Queue {
    Node* head;
    Node* tail;
    Queue() {
        head = tail = NULL;
    }
    void push(int s) {
        Node* n = new Node(s);
        n->next = head;
        n->prior = NULL;
        if (head != NULL) {
            head->prior = n;
        }
        head = n;
        if (tail == NULL) {
            tail = n;
        }
    }
    int front() {
        if (head == NULL) {
            return - 1;
        }
        return tail->value;
    }
    void pop() {
        if (head == NULL) {
            return;
        }
        Node* second_last = tail->prior;
        delete tail;
        tail = second_last;
        if (tail == NULL) {
            head = NULL;
        }
    }
};

```



## Stack implementation supporting getMin() with smaller memory usage

```
struct Node {
    Node* next;
    int value;
    Node(int v) {
        value = v;
        next = NULL;
    }
};

struct Stack {
    Node* head;
    stack<int> s;
    Stack() {
        head = NULL;
    }
    ~Stack() {
        while (head) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }
    void push(int v) {
        Node* n = new Node(v);
        n->next = head;
        head = n;
        if (s.empty() || s.top() >= v) s.push(v);
    }
    int top() {
        if (!head) return -1;
        return head->value;
    }
    int pop() {
        if (!head) return -1;
        Node* temp = head;
        head = head->next;
        if (temp->value == s.top()) s.pop();
        delete temp;
    }
    int max() {
```

```

        if (!head) return -1;
        return s.top();
    }
};

```

## Homework practices (Class 3)

Implementing a queue using two stacks.

```

class Q2S{
    stack<int> s1;
    stack<int> s2;

public:
    Q2S(){};
    ~Q2S() {
        while(!s1.empty()) {
            s1.pop();
        }
    }
    void push(int v) {           // O(N)
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }
        s2.push(v);
        while (!s2.empty()) {
            s1.push(s2.top());
            s2.pop();
        }
    }
    void pop() {                 // O(1)
        if (!s1.empty()) s1.pop();
    }
    int front() {
        if (s1.empty()) return -1;    // assuming error code is -1.
    }
}

```

```

        return s1.top();
    }
};

void testQ2S() {
    Q2S q;
    cout << q.front() << endl;
    q.push(1);
    cout << q.front() << endl;
    q.push(2);
    cout << q.front() << endl;
    q.push(3);
    cout << q.front() << endl;
    q.pop();
    cout << q.front() << endl;
    q.pop();
    cout << q.front() << endl;
    q.pop();
    cout << q.front() << endl;
    q.pop();
    cout << q.front() << endl;
}

```

**Implement a stack with min() function**

```

struct StackMinNode {
    int value;
    int min;
    StackMinNode* next;
    StackMinNode(int t){
        value = t;
        next = NULL;
    }
};

struct StackMin {
    StackMinNode* head;
    StackMin() {
        head = NULL;
    }
    ~StackMin() {
        while (head != NULL) {

```

```

        StackMinNode* temp = head;
        head = head->next;
        delete temp;
    }
}

void push(int d) {
    StackMinNode* n = new StackMinNode(d);
    n->next = head;
    if (head == NULL) {
        n->min = d;
    } else {
        n->min = d < head->min ? d : head->min;
    }
    head = n;
}

int getMin() {
    if (!head) return -1;
    return head->min;
}

int top() {
    if (head == NULL) {
        return -1;
    }
    return head->value;
}

void pop() {
    if (head == NULL) {
        return;
    }
    StackMinNode *temp = head;
    head = head->next;
    delete temp;
}
};

```

```

void testStackMin() {
    StackMin sm;
    cout << sm.top() << endl;
    cout << sm.getMin() << endl;
    sm.pop();
    cout << sm.top() << endl;
    cout << sm.getMin() << endl;
    sm.push(1);
}

```

```

    cout << sm.top() << endl;
    cout << sm.getMin() << endl;
    sm.push(2);
    cout << sm.top() << endl;
    cout << sm.getMin() << endl;
    sm.push(3);
    cout << sm.top() << endl;
    cout << sm.getMin() << endl;
    sm.pop();
    cout << sm.top() << endl;
    cout << sm.getMin() << endl;
    sm.pop();
    cout << sm.top() << endl;
    cout << sm.getMin() << endl;
    sm.pop();
    cout << sm.top() << endl;
    cout << sm.getMin() << endl;
}

```

### How to reverse a linkedList in a recursive way

```

struct Node {
    Node* next;
    int value;
    Node(int v) : value(v), next(NULL) {};
};

void reverseLL_recur(Node* head, Node*& head2, Node* prev) {
    if (!head) return;
    Node* next = head->next;
    head->next = prev;
    if (!next) {
        head2 = head;
        return;
    }
    reverseLL_recur(next, head2, head);
}

void printLL(Node* head) {
    while (head != NULL) {
        cout << head->value << " ";
        head = head->next;
    }
}

```

```

    }
    cout << endl;
}

void insertNodeInFront(Node*& head, Node* node) {
    node->next=head;
    head = node;
}

void testReverseLL_recur() {
    Node* head = NULL;
    Node* new_head = NULL;
    printLL(head);
    reverseLL_recur(head, new_head, NULL);
    printLL(head);

    Node m1(1);
    head = &m1;
    printLL(head);
    new_head = NULL;
    reverseLL_recur(head, new_head, NULL);
    printLL(new_head);

    Node n1(1);
    Node n2(2);
    Node n3(3);
    Node n4(4);
    head = &n4;
    insertNodeInFront(head, &n3);
    insertNodeInFront(head, &n2);
    insertNodeInFront(head, &n1);
    printLL(head);
    new_head = NULL;
    reverseLL_recur(head, new_head, NULL);
    printLL(new_head);
}

```

### How to reverse a linkedList in an iterative way

```

void reverseLL_iter (Node*& head) {
    if (head == NULL || head->next == NULL) return;
    Node * prev = NULL;

```

```

Node * next = NULL;;
Node * temp = head;
while (temp != NULL) {
    next = temp->next;
    temp->next = prev;
    prev = temp;
    temp = next;
}
head = prev;
}

```

```

void testReverseLL_iter() {
    Node* head = NULL;
    printLL(head);
    reverseLL_iter(head);
    printLL(head);

    Node m1(1);
    head = &m1;
    printLL(head);
    reverseLL_iter(head);
    printLL(head);

    Node n1(1);
    Node n2(2);
    Node n3(3);
    Node n4(4);
    head = &n4;
    insertNodeInFront(head, &n3);
    insertNodeInFront(head, &n2);
    insertNodeInFront(head, &n1);
    printLL(head);
    reverseLL_iter(head);
    printLL(head);
}

```

### Find linked list middle element

```

int findLinkedListMiddle(Node* head) {
    if (head == NULL) {
        return -1;
    }
    Node* slow = head;

```

```

Node* fast = head;
while (fast->next != NULL && fast->next->next != NULL) {
    slow = slow->next;
    fast = fast->next->next;
}
return slow->value;
}

```

```

void testFindLinkedListMiddle() {
    Node* head = NULL;
    assert(findLinkedListMiddle(head) == -1);

    Node m1(1);
    head = &m1;
    assert(findLinkedListMiddle(head) == 1);

    Node n1(1);
    Node n2(2);
    Node n3(3);
    Node n4(4);
    head = &n4;
    insertNodeInFront(head, &n3);
    insertNodeInFront(head, &n2);
    insertNodeInFront(head, &n1);
    assert(findLinkedListMiddle(head) == 2);
}

```

### **Determine if a linked list is cycled**

```

bool isLinkedListCycled(Node* head){
    Node* pt1 = head;
    Node* pt2 = head;
    while (pt1 && pt2) {
        pt1 = pt1->next;
        pt2 = pt2->next;
        if (!pt2) {
            return false;
        } else {
            pt2 = pt2->next;
        }
    }
    if (pt1 == pt2) {
        return true;
    }
}

```



```

    }
}
return false;
}

void testIsLinkedListCycled() {
    Node* n1 = new Node(1);
    Node* n2 = new Node(2);
    Node* n3 = new Node(3);
    Node* n4 = new Node(4);
    Node* head = n4;
    insertNodeInFront(head, n3);
    insertNodeInFront(head, n2);
    insertNodeInFront(head, n1);
    assert(!isLinkedListCycled(head));

    n4->next = n2;
    assert(isLinkedListCycled(head));
    delete n1;
    delete n2;
    delete n3;
    delete n4;

    Node* m1 = new Node(10);
    assert(!isLinkedListCycled(m1));
    delete m1;

    head = NULL;
    assert(!isLinkedListCycled(head));
}

```

### Insert value into a sorted linked list

```

void insertNodeIntoSortedList(int value, Node*& head) {
    Node* node = new Node(value);
    if (head == NULL || head->value >= value) {
        node->next = head;
        head = node;
        return;
    }
    Node* prev = head;
    while (prev->next != NULL && prev->next->value < value) {

```

```

        prev = prev->next;
    }
    node->next = prev->next;
    prev->next = node;
}

void testInsertNodeIntoSortedList() {
    Node* head = NULL;
    insertNodeIntoSortedList(1, head);
    printLL(head);
    insertNodeIntoSortedList(3, head);
    printLL(head);
    insertNodeIntoSortedList(7, head);
    printLL(head);
    insertNodeIntoSortedList(2, head);
    printLL(head);
    insertNodeIntoSortedList(5, head);
    printLL(head);
    insertNodeIntoSortedList(4, head);
    printLL(head);
}

```

### Merge two sorted linked list

```

Node* mergeTwoSortedList(Node* head1, Node* head2) {
    if (!head1 && !head2) return NULL;
    if (!head1) return head2;
    if (!head2) return head1;
    Node fakeHead(-1);
    Node* fakeHead = new Node(-1);
    Node* current = &fakeHead;
    Node * current = fakeHead;
    while (head1 != NULL && head2 != NULL) {
        if (head1->value <= head2->value) {
            current->next = head1;
            head1 = head1->next;
        } else {
            current->next = head2;
            head2 = head2->next;
        }
        current = current->next;
    }
    if (head1 != NULL) {

```

```

        current->next = head1;
    } else {
        current->next = head2;
    }
    Node* temp = fakeHead->next;
    delete fakeHead;
    return fakeHead.next;
    return temp;
}

```

```

void testMergeTwoSortedList() {
    Node* head1 = NULL;
    Node* head2 = NULL;
    printLL(mergeTwoSortedList(head1, head2));

    head1 = head2 = NULL;
    insertValueInFront(head1, 4);
    printLL(mergeTwoSortedList(head1, head2));

    head1 = head2 = NULL;
    insertValueInFront(head1, 4);
    insertValueInFront(head2, 3);
    printLL(mergeTwoSortedList(head1, head2));

    head1 = head2 = NULL;
    insertValueInFront(head1, 4);
    insertValueInFront(head1, 2);
    insertValueInFront(head2, 3);
    insertValueInFront(head2, 1);
    printLL(mergeTwoSortedList(head1, head2));
}

```

**Reorder list 1->2->3->4->5 to become 1->5->2->4->3**

```

Node* findLinkedListMiddleNode(Node* head) {
    if (head == NULL) return NULL;
    Node* slow = head;
    Node* fast = head;
    while (fast->next != NULL && fast->next->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
}

```

```
    return slow;
}
```

```
Node* mergeOneByOne(Node* head1, Node* head2) {
    Node fake(-1);
    Node* current = &fake;
    while (head1 != NULL && head2 != NULL) {
        current->next = head1;
        head1 = head1->next;
        current = current->next;
        current->next = head2;
        head2 = head2->next;
        current = current->next;
    }
    if (head1 != NULL) {
        current->next = head1;
    } else {
        current->next = head2;
    }
    return fake.next;
}
```

```
Node* reOrderLinkListHeadTail2ndHead2ndTailEtc(Node* head) {
    if (head == NULL) return NULL;
    Node* mid = findLinkListMiddleNode(head);
    Node* head2 = mid->next;
    Node* head1 = head;
    mid->next = NULL;    // so that the first half ends with NULL
    reverseLL_iter(head2);
    return mergeOneByOne(head1, head2);
}
```

```
void testReOrderLinkListHeadTail2ndHead2ndTailEtc() {
    Node* head = NULL;
    printLL(reOrderLinkListHeadTail2ndHead2ndTailEtc(head));
    head = NULL;
    insertValueInFront(head, 1);
    printLL(reOrderLinkListHeadTail2ndHead2ndTailEtc(head));
    head = NULL;
    insertValueInFront(head, 1);
    insertValueInFront(head, 2);
    printLL(reOrderLinkListHeadTail2ndHead2ndTailEtc(head));
    head = NULL;
}
```

```

insertValueInFront(head, 1);
insertValueInFront(head, 2);
insertValueInFront(head, 3);
insertValueInFront(head, 4);
printLL(reOrderLinkedListHeadTail2ndHead2ndTailEtc(head));
head = NULL;
insertValueInFront(head, 1);
insertValueInFront(head, 2);
insertValueInFront(head, 3);
insertValueInFront(head, 4);
insertValueInFront(head, 5);
printLL(reOrderLinkedListHeadTail2ndHead2ndTailEtc(head));
}

```

**Partition a list given a target value such that all nodes with value less than target go to the left portion, the rest goes to the right.**

```

Node* partitionListSuchThatValuesLessThanTargetGoLeft(Node* head, int target) {
    if (head == NULL) return NULL;
    Node fakeSmallHead(-1);
    Node fakeLargeHead(-1);
    Node* currSmall = &fakeSmallHead;
    Node* currLarge = &fakeLargeHead;
    while (head != NULL) {
        if (head->value < target) {
            currSmall->next = head;
            currSmall = currSmall->next;
        } else {
            currLarge->next = head;
            currLarge = currLarge->next;
        }
        head = head->next;
    }
    currSmall->next = fakeLargeHead.next;
    currLarge->next = NULL;
    return fakeSmallHead.next;
}

```

```

void testPartitionListSuchThatValuesLessThanTargetGoLeft() {
    int target = 5;
    Node* head = NULL;
    printLL(partitionListSuchThatValuesLessThanTargetGoLeft(head, target));
}

```

```

head = NULL;
insertValueInFront(head, 1);
printLL(partitionListSuchThatValuesLessThanTargetGoLeft(head, target));
head = NULL;
insertValueInFront(head, 1);
insertValueInFront(head, 10);
printLL(partitionListSuchThatValuesLessThanTargetGoLeft(head, target));
head = NULL;
insertValueInFront(head, 1);
insertValueInFront(head, 10);
insertValueInFront(head, 2);
insertValueInFront(head, 12);
insertValueInFront(head, 100);
printLL(partitionListSuchThatValuesLessThanTargetGoLeft(head, target));
head = NULL;
insertValueInFront(head, 1);
insertValueInFront(head, 2);
insertValueInFront(head, 3);
insertValueInFront(head, 4);
insertValueInFront(head, 2);
printLL(partitionListSuchThatValuesLessThanTargetGoLeft(head, target));
head = NULL;
insertValueInFront(head, 11);
insertValueInFront(head, 21);
insertValueInFront(head, 31);
insertValueInFront(head, 43);
insertValueInFront(head, 24);
printLL(partitionListSuchThatValuesLessThanTargetGoLeft(head, target));
}

```

## Class 4

### Questions during the class?

#### FAQ

##### priority\_queue usage.

Default/simple usage:

```
#include <queue>
```

```
priority_queue<int> pq;    // this is a max heap by default
```

To implement a minheap, do this:

```
priority_queue<int, vector<int>, MyComp> pq;
class MyComp {
public:
    bool operator() (int left, int right) {
        return left > right;
    }
};
```

In our homework, we use heap to store a complex type, i.e.:

`pair<int, pair<int, int> >`

Our Comp class will become:

```
class Comp {
public:
    bool operator() (pair<int, pair<int, int> > left, pair<int, pair<int, int> > right) {
        return left.first > right.first;
    }
};
```

Question: a data structure storing student's name and age. Write a Comp class to store students into min heap, sorted by their age.

`pair<string, int>`

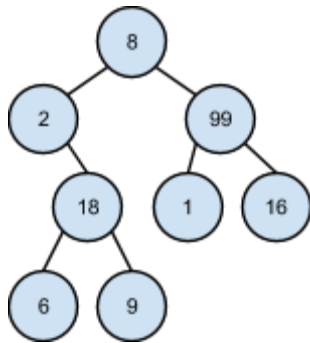
```
class Comp {
public:
    bool operator() (pair<string, int> left, pair<string, int> right) {
        return left.second > right.second;
    }
};
```

## Content

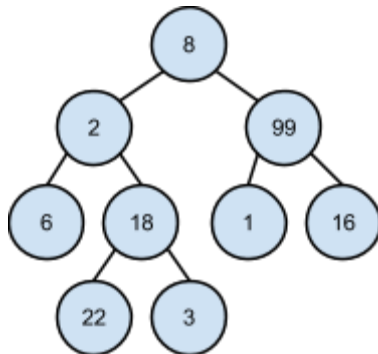
### Fundamentals of BT & BST

#### Binary Tree

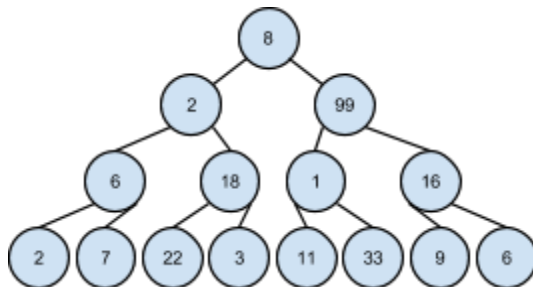
A **binary tree** is a tree data structure in which each node has at most two children.



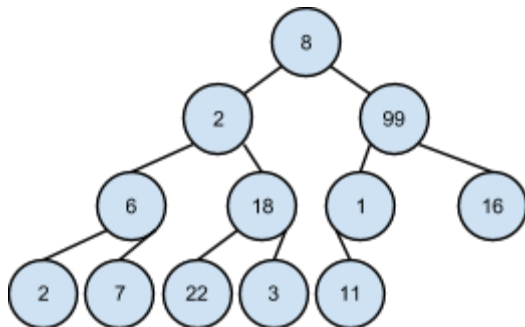
A **full binary tree** is a binary tree where every node other than the leaves has two children.



A **perfect binary tree** is a full binary tree where all leaves have the same depth.



A **complete binary tree** (see class 3) is a binary tree in which every level, except possibly the last level, is completely filled, and all nodes are as far left as possible.



```

struct TreeNode {           // binary tree node.
    int value;
    TreeNode* left;
    TreeNode* right;
  
```



```
};
```

In general, to access the entire tree, the root node is given as input.

Question: How can we represent a general tree, where each node can have multiple children?

```
struct TreeNode {  
    int value;  
    vector<TreeNode*> children;  
};
```

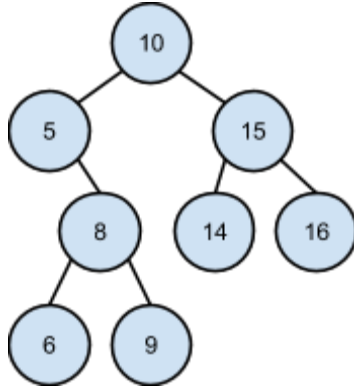
### Binary Search Tree

A **binary search tree** is a node-based binary tree data structure where each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right sub-tree.

- it is a binary tree.
- left child is less than parent, which is less than right child.

Root node: has nobody points to it.

Leaf node: has no child.

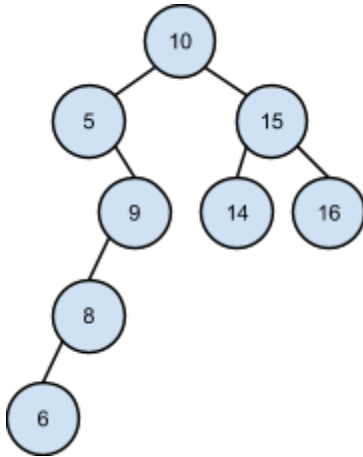


// Let's define a useful constructor for the tree node.

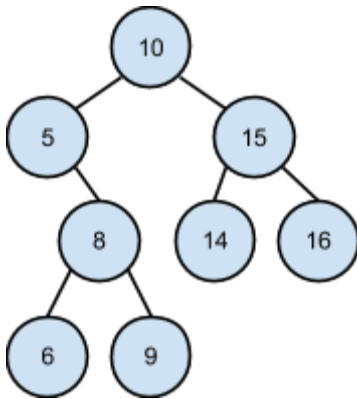
```
struct TreeNode {  
    int value;                // stores actual content  
    TreeNode* left;  
    TreeNode* right;  
    TreeNode(int n) {  
        left = NULL;  
        right = NULL;  
        value = n;  
    }  
};
```

A **balanced binary search tree** is a binary search tree that has the minimum possible maximum depth for the leaf nodes.

Unbalanced example:



Balanced example:



**Traverse Tree using recursion**

Pre/In/Post

```
void preOrder(TreeNode* root) {  
    if (root == NULL) return;  
    cout << root->value << " ";  
    preOrder(root->left);  
    preOrder(root->right);  
}  
  
void inOrder(TreeNode* root) {  
    if (root == NULL) return;  
    inOrder(root->left);  
    cout << root->value << " ";  
    inOrder(root->right);  
}
```

```

}
void postOrder(TreeNode* root) {
    if (root == NULL) return;
    postOrder(root->left);
    postOrder(root->right);
    cout << root->value << " ";
}

```

## Insert, find, delete an element in a binary search tree

### Insert

```

void insertBSTNode(TreeNode*& root, int val) {
    TreeNode * node = new TreeNode(val);
    if (root == NULL) {
        root = node;
        return;
    }
    TreeNode* temp = root;
    TreeNode* parent = NULL;
    while (temp) {
        if (temp->value > val) {
            parent = temp;
            temp = temp->left;
        } else {
            parent = temp;
            temp = temp->right;
        }
    }
    if (parent->value > val) {
        parent->left = node;
    } else {
        parent->right = node;
    }
}

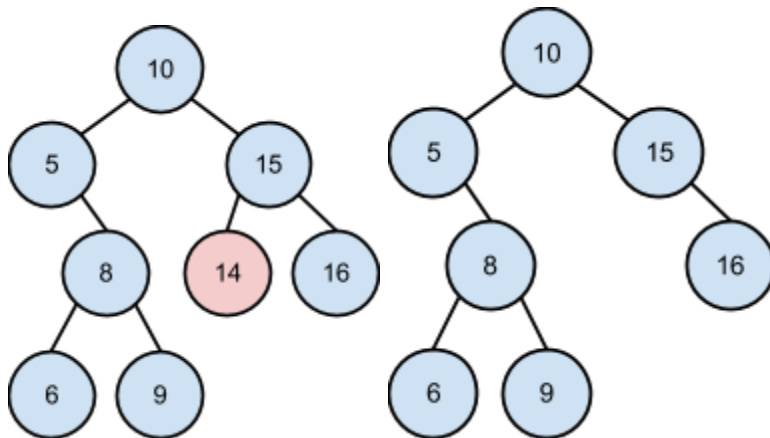
```

### Find

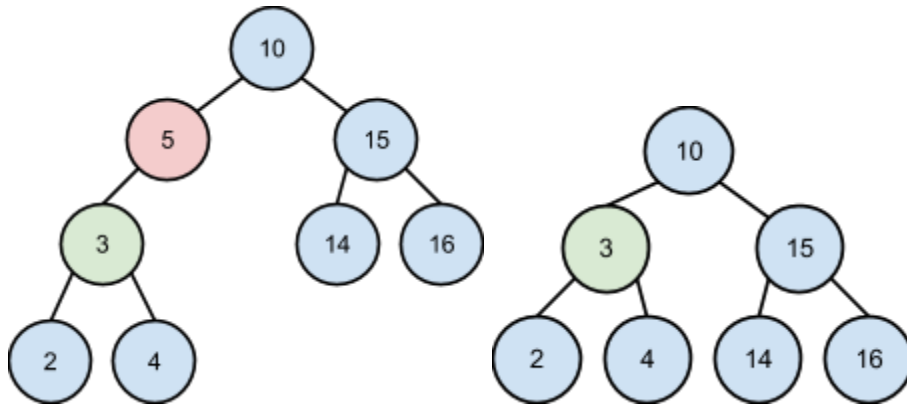
```
TreeNode* findBSTNode(int target, TreeNode* root, TreeNode*& parent) {
    TreeNode* temp = root;
    if (!root) return NULL;
    parent = NULL;
    while (temp) {
        if (target > temp->value) {
            parent = temp;
            temp = temp->right;
        } else if (target < temp->value) {
            parent = temp;
            temp = temp->left;
        } else {
            return temp;
        }
    }
    // Not found
    parent = NULL;
    return NULL;
}
```

### Delete

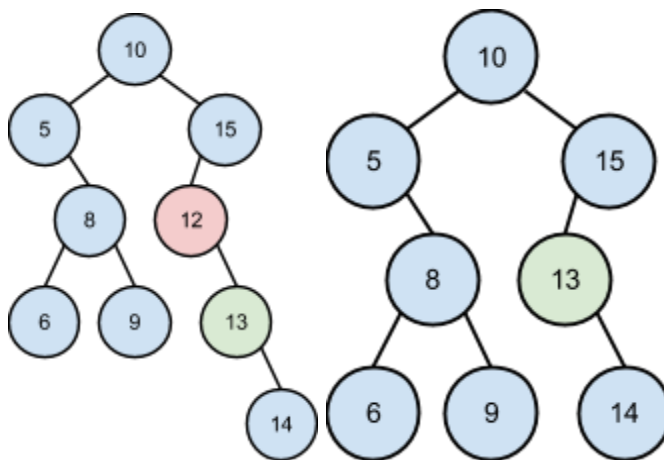
#### **Case 1:**



#### **Case 2:**

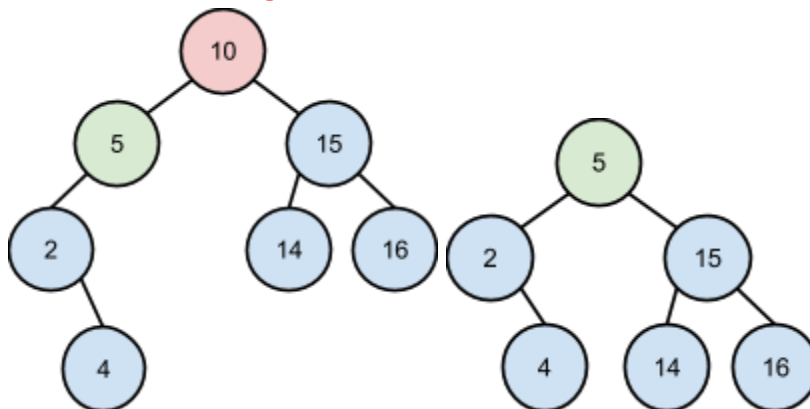


**Case 3:**



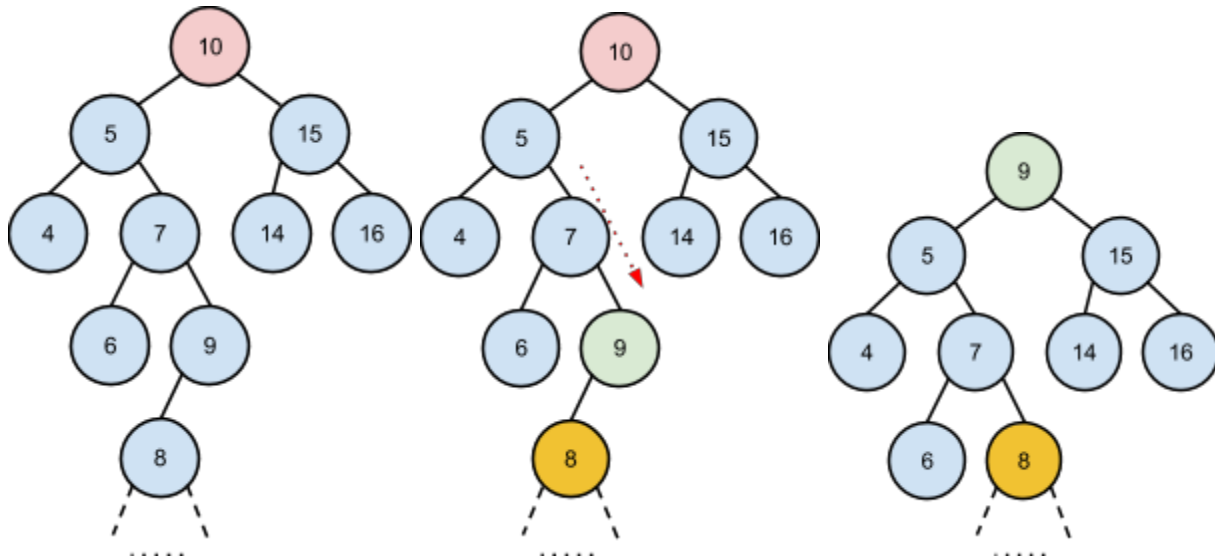
**Case 4.1:**

**left child has no right subtree.**



**Case 4.2:**

**left child has right subtree.**



```

void deleteBSTNode(int target, TreeNode*& root) {
    if (root == NULL) return;
    TreeNode* toDelete = NULL;
    TreeNode* parent = NULL;
    TreeNode* next = NULL;
    TreeNode* parentNext = NULL;
    toDelete = findBSTNode(target, root, parent);
    if (!toDelete) return;
    if (toDelete->left == NULL && toDelete->right == NULL) {
    } else if (toDelete->right == NULL) {      // case 2
        next = toDelete->left;
    } else if (toDelete->left == NULL) {      // case 3
        next = toDelete->right;
    } else {
        parentNext = toDelete;
        next = toDelete->left;
        while (next->right) {
            parentNext = next;
            next = next->right;
        }
    }
    if (parentNext == toDelete) {             // case 4.1
        next->right = toDelete->right;
    } else {                                 // case 4.2
        parentNext->right = next->left;
        next->left = toDelete->left;
        next->right = toDelete->right;
    }
}

```

```

}
if (!parent) {
    root = next;
} else if (parent->value > target) {
    parent->left = next;
} else {
    parent->right = next;
}
delete toDelete;
}

```

## Tree Traversals - Pre/In/Post-Order

### Definition

Pre-order: current node -> left children -> right children

In-order: left children -> current node -> right children

Post-order: left children -> right children -> current node

### Iteration - the hard way

PreOrder:

```

void preOrderIter(TreeNode* root) {
    if (!root) return;
    stack<TreeNode*> s;
    s.push(root);
    while (!s.empty()) {
        TreeNode* temp = s.top();
        s.pop();
        cout << temp->value << endl;
        if (temp->right) {
            s.push(temp->right);
        }
        if (temp->left) {
            s.push(temp->left);
        }
    }
}
}

```

```

      2      3
4  5      6  7

```

nthLoop	stack content (top->bottom)	printed:
1st	1	
2nd	2 3	1
3	4 5 3	2
4	5 3	4
5	3	5
6	6 7	3
7	7	6
8		7

```

Inorder
      1
      2      3
4  5      6  7

```

nthLoop	stack content (top->bottom) break 1 ,	stack content break 2	printed
1	4 2 1	2 1	4
2	2 1	1	2
3	5 1	1	5
4	1	empty	1
5	6 3	3	6
6	3	empty	3
7	7	empty	7

```

void inOrderIter(TreeNode* root) {
    if (!root) return;
    stack<TreeNode*> s;
    TreeNode* temp = root;
    while (1) {
        while (temp) {
            s.push(temp);
            temp = temp->left;
        }
        // break 1
        if (s.empty()) return;
        temp = s.top();
        cout << temp->value << " ";
        s.pop();
    }
}

```



```

        // break 2
        temp = temp->right;
    }
}

```

Post order:

```

void postOrderIter(TreeNode* root) {
    if (!root) return;
    stack<TreeNode*> s1;
    stack<TreeNode*> s2;
    s1.push(root);
    while (!s1.empty()) {
        TreeNode* temp = s1.top();
        s1.pop();
        s2.push(temp);    // different from pre-order code
        if (temp->left) s1.push(temp->left);    // the order is the oppsitite to pre order code
        if (temp->right) s1.push(temp->right);
    }
    while (!s2.empty()) {
        cout << s2.top()->value << " ";
        s2.pop();
    }
}

```

	1		
	2	3	
4	5	6	7
nth loop		stack 1 (top-> bottom)	stack 2 (top->bottom)
0		1	empty
1		3 2	1
2		7 6 2	3 1
3		6 2	7 3 1
4		2	6 7 3 1
5		5 4	2 6 7 3 1
6		4	5 2 6 7 3 1
7		empty	4 5 2 6 7 3 1

## Class 5

### Questions?

### Content

#### c++ stl “set”

```
set<int> number_set;
// How to insert an element?
number_set.insert(1);
number_set.insert(2);
number_set.insert(3);

// delete:
number_set.erase(1);
// How to check if an element has been inserted?
assert(number_set.find(10) == number_set.end());

// How to print the whole set?
for(set<int>::iterator i = number_set.begin(); i != number_set.end(); ++i) {
}
```

c++ stl “set” is implemented using what?

A: **binary search tree**. Thus: complexity?

insert:  $O(\lg N)$

find:  $O(\lg N)$

delete:  $O(\lg N)$

Remember, in order to construct a binary search tree, parent & child nodes need to maintain some sort of comparison relationship. So, we will need to implement **operator<** for each custom data structure.

```
class Student {
public:
    int age;
    string name;
    Student(int a, string n) : age(a), name(n) {}
};

Student s1(8, "a");
Student s2(9, "b");
Student s3(10, "c");
Student s4(10, "d");
```

```
Student s5(10, "alice");
```

```
Student s6(10, "e");
```

```
set<Student> student_set;
```

```
student_set.insert(s1);    // this will cause compile time problem.
```

```
class Student {
```

```
public:
```

```
    int age;
```

```
    string name;
```

```
    Student(int a, string n) : age(a), name(n) {}
```

```
    // A badly designed operator<
```

```
    // only compares age.
```

```
    // A well designed operator <
```

```
    bool operator< (const Student other) const {
```

```
        if (strcmp(name.c_str(), other.name.c_str()) == 0) {
```

```
            return age < other.age;
```

```
        } else {
```

```
            return strcmp(name.c_str(), other.name.c_str()) < 0;
```

```
        }
```

```
        //return age < other.age;
```

```
    }
```

```
};
```

```
student_set.insert(s2);
```

```
student_set.insert(s3);
```

```
student_set.insert(s4);
```

```
student_set.insert(s5);
```

```
assert(student_set.find(s1) != student_set.end());
```

```
assert(student_set.find(s2) != student_set.end());
```

```
assert(student_set.find(s3) != student_set.end());
```

```
assert(student_set.find(s4) != student_set.end());
```

```
assert(student_set.find(s5) != student_set.end());
```

```
assert(student_set.find(s6) == student_set.end());
```

```
for (set<Student>::iterator it = student_set.begin(); it != student_set.end(); ++it) {
```

```
    cout << it->age << " " << it->name << endl;
```

```
}
```

## c++ stl “map”

“<key, value> pair.”

Implementation? => Binary search tree. (balanced binary search tree.)

Complexity?

insert/lookup/delete:  $O(\lg N)$ .

```
map<int, string> id_name_map;
id_name_map.insert(make_pair(1, "z"));
id_name_map.insert(make_pair(2, "y"));
id_name_map.insert(make_pair(3, "x"));
id_name_map.insert(make_pair(4, "w"));
for (map<int, string>::iterator it = id_name_map.begin(); it != id_name_map.end(); ++it) {
    cout << it->first << " " << it->second << endl;
}
assert(id_name_map.find(1) != id_name_map.end());
assert(id_name_map.find(2) != id_name_map.end());
assert(id_name_map.find(3) != id_name_map.end());
assert(id_name_map.find(4) != id_name_map.end());
assert(id_name_map.find(5) == id_name_map.end());

cout << id_name_map[1] << endl;
cout << id_name_map[2] << endl;
cout << id_name_map[3] << endl;
cout << id_name_map[4] << endl;
cout << id_name_map[5] << endl;
cout << id_name_map[6] << endl;
assert(id_name_map.find(5) != id_name_map.end());
assert(id_name_map.find(6) != id_name_map.end());
// operator [] is destructive! It inserts the key and a default value instance to the map!
```

### Rule of thumb:

Don't use operator[] to check if a key exists. Always use map.find() == map.end(). Only use operator [] when you are absolutely sure that the item exists.

Also, don't confuse

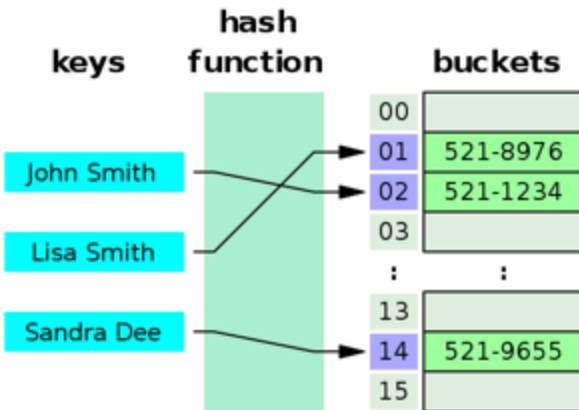
map<Student, string> with map<Student\*, string>. The former requires operator< overloading, whereas the second does not.

## Hash table

A data structure that:

- Implemented using an array
- An associative array where each slot stores a <key, value> pair.
- A **hash function** determines the index of a given key into the array.

E.g.:



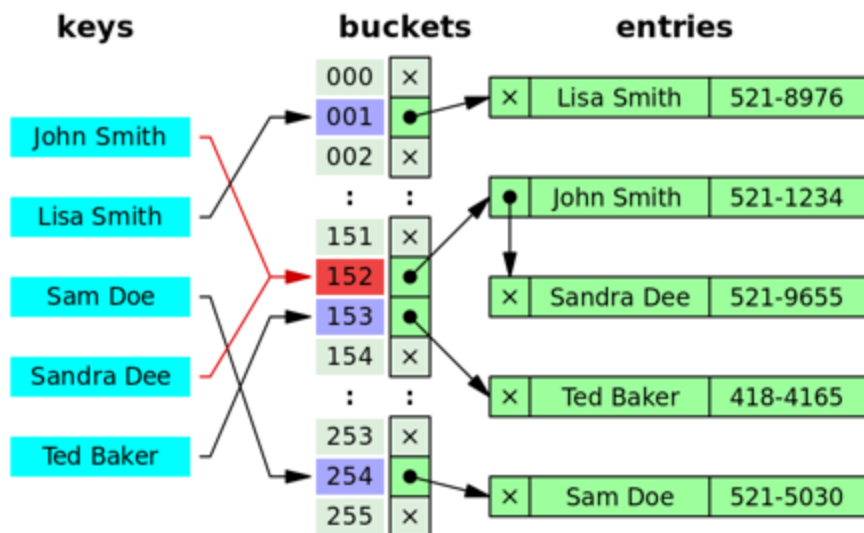
In this case, a hash function signature looks like:

```
int calculateHashIndex(string name); // Some complex computation could be happening inside.
```

Benefits? (Question)

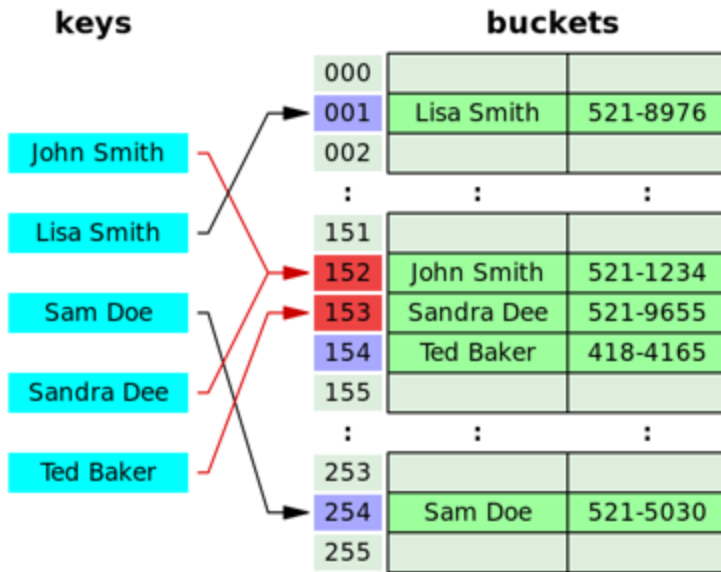
- Insert, find, delete time:  $O(1)$ 
  - Also consider time to compute hash.
  - On average time.

Problems? (Question)



- Collision
  - Solution:
    - Chaining.

- Open addressing/Linear Probing
- Worst case lookup time:  $O(N)$



- Have to worry about resizing hash table when load factor is high.
  - When resizing, old entries need to be “re-hashed” to new entries.
- No natural sorting of data.

### Summary: Binary search tree vs. Hash table

- BST
  - Average case  $O(\lg N)$  insert, find and delete.
  - Worst case  $O(N)$  (unbalanced tree).
  - Data automatically sorted => support ordered iteration.
  - Discrete storage, works for any value types.
- Hash table.
  - Average case  $O(1)$  insert, find, and deletion
  - Worst case  $O(N)$ . (collision)
  - Data not sorted
  - Needs collision resolution:
    - Chaining: collided data stored after original data, outside of hash bucket addresses
    - Open addressing: collided data stored within hash bucket, finding the next available open space according to probing algorithm. (When searching also go to the same entry and follow the probing algorithm). Performance significantly degrades when load factor > 70%
  - Requires hash function which could be hard to implement

- When resizing, old entries need to be translated to new entries by calculating new hash function
  - Shine in very large arrays
- Compact storage, works well if value types are small.

## Implement a map

Note: in C++. “set”/“map” are implemented using BST. “unordered\_set”/“unordered\_map” are implemented using Hash table.

### Using BST

This is a demonstration of trivial implementation - no delete code, no code to handle duplicate items. Industry level (e.g., c++ stl) implementation is much more complicated than this.

```
struct Student {
    int age;
    string name;
    Student* left;
    Student* right;
    Student(int a, string n) : age(a), name(n), left(NULL), right(NULL) {}
    bool operator<(const Student other) const {
        return strcmp(name.c_str(), other.name.c_str()) < 0;
    }
};
```

```
class StudentMap {
private:
    Student* root;
public:
    StudentMap() {root = NULL;}
    void insert(string name, int age) {
        Student* node = new Student(age, name);
        if (root == NULL) {
            root = node;
            return;
        }
        Student* temp = root;
        Student* parent = NULL;
        while (temp) {
            if (*temp < *node) {
                parent = temp;
                temp = temp->right;
            } else {
```

```

        parent = temp;
        temp = temp->left;
    }
}
if (*parent < *node) {
    parent->right = node;
} else {
    parent->left = node;
}
}
int find(string name) {
    if (!root) return -1;
    Student* temp = root;
    while(temp) {
        if (strcmp(name.c_str(), temp->name.c_str()) < 0) {
            temp = temp->left;
        } else if (strcmp(name.c_str(), temp->name.c_str()) > 0) {
            temp = temp->right;
        } else {
            return temp->age;
        }
    }
    // Not found
    return -1;
}
};
int main(int argc, char** argv) {
    StudentMap m;
    m.insert("abc", 13);
    m.insert("abcd", 1);
    m.insert("abce", 131);
    m.insert("abcasdf", 133);
    m.insert("x", 1443);
    m.insert("john", 11113);
    assert(m.find("abc") == 13);
    assert(m.find("abcd") == 1);
    assert(m.find("abce") == 131);
    assert(m.find("abcasdf") == 133);
    assert(m.find("x") == 1443);
    assert(m.find("john") == 11113);
}

```



## Using Hash Table

```
#define HS 20
struct Node {
    int value;
    string name;
    struct Node * next;
    Node(string n, int v) : name(n), value(v){
        next = NULL;
    }
};

class StudentHashTable {
    Node * a[HS];
public:
    StudentHashTable() {
        for (int i = 0; i < HS; i++) {
            a[i] = NULL;
        }
    }
    ~StudentHashTable() {
        for (int i = 0; i < HS; i++) {
            Node * temp = a[i];
            Node * d;
            while (temp) {
                d = temp;
                temp = temp->next;
                delete d;
            }
        }
    }
    void insert(string s, int v) {
        int key = getHash(s);
        if (a[key] == NULL) {
            Node * n = new Node(s, v);
            a[key] = n;
        } else {
            Node * temp = a[key];
            Node * prev;
            while (temp) {
                if (temp->name == s) {
                    temp->value = v;
                    return;
                }
            }
        }
    }
};
```

```

        }
        prev = temp;
        temp = temp->next;
    }
    Node * n = new Node(s, v);
    prev->next = n;
}
}
int find(string s) {
    int key = getHash(s);
    if (!a[key]) return -1;
    Node * temp = a[key];
    while (temp) {
        if (temp->name == s) {
            return temp->value;
        }
        temp = temp->next;
    }
    return -1;
}
private:
    // string length is N
    // get a value, value = 31^(n-1) * s[0] + 31^(n - 2) * s[1] +.... + 31 ^ 1 * s[n - 1]]
    // value % HS
    int getHash(string s) {
        long res = 0;
        long len = s.length();
        for (int i = 0; i < len; i++) {
            res = res + (int)pow(31, len - i - 1) * s[i];
        }
        // int res = s[2] - 'a';
        return res % HS;
    }
};
// main
StudentHashTable m;
m.insert("abc", 13);
m.insert("abcd", 1);
m.insert("abce", 131);
m.insert("abcasdf", 133);
m.insert("x", 1443);
m.insert("john", 11113);
assert(m.find("abc") == 13);

```

```

assert(m.find("abcd") == 1);
assert(m.find("abce") == 131);
assert(m.find("abcasdf") == 133);
assert(m.find("x") == 1443);
assert(m.find("john") == 11113);
m.insert("abc", 900);
cout << m.find("abc");

```

// why was our program not running?

In this line:

```
assert(m.find("abcasdf") == 133);
```

int overflow occurred, Apparently on my PC the program terminates and on my MAC it doesn't.

Solution: use unsigned int, or higher bit integer (long).

Can we link the implementation of a map to our previous knowledge about tree, list, array, etc.

## Homework Practices

**Determine whether an undirected graph is Bipartite**

```

struct GraphNode {
    int value;
    vector<GraphNode*> neighbors;
    GraphNode(int v) : value(v) {}
};

```

// Starts with a node. If this node hasn't been visited before, mark this node as "0".

// For all of its immediate neighbors, mark them as "1".

// For 2nd level neighbors, mark as "0" again (i.e., immediate neighbors of immediate neighbors)

// ... And so on, until the BFS is over.

// Also store all visited nodes in the map to avoid duplicate.

```

bool bfsAndMark(GraphNode* node, map<GraphNode*, int>& nodeMark) {
    if (nodeMark.find(node) != nodeMark.end()) return true;
    queue<GraphNode*> q;
    q.push(node);
    nodeMark[node] = 0;
    while (!q.empty()) {
        GraphNode* curr = q.front();

```

```

    q.pop();
    int currMark = nodeMark[curr];
    int neighborMark = currMark == 0 ? 1 : 0;
    for (int i = 0; i < curr->neighbors.size(); ++i) {
        GraphNode* neighbor = curr->neighbors[i];
        if (nodeMark.find(neighbor) == nodeMark.end()) {
            q.push(neighbor);
            nodeMark[neighbor] = neighborMark;
        } else {
            if (nodeMark[neighbor] != neighborMark) {
                return false;
            }
        }
    }
}
return true;
}

```

```

bool isGraphBipartite(vector<GraphNode*> graph) {
    if (graph.empty()) return false;
    map<GraphNode*, int> nodeMark;
    for (int i = 0; i < graph.size(); ++i) {
        if (!bfsAndMark(graph[i], nodeMark)) {
            return false;
        }
    }
    return true;
}

```

```

void testIsGraphBipartite() {
    // 1 2
    // | \
    // 3 4
    GraphNode n1(1);
    GraphNode n2(2);
    GraphNode n3(3);
    GraphNode n4(4);
    n1.neighbors.push_back(&n3);
    n1.neighbors.push_back(&n4);
    n3.neighbors.push_back(&n1);
    n4.neighbors.push_back(&n1);
    vector<GraphNode*> graph;
    graph.push_back(&n1);
}

```

```

graph.push_back(&n2);
graph.push_back(&n3);
graph.push_back(&n4);
assert(isGraphBipartite(graph) == true);
// 1 - 2
// | \
// 3 4
n1.neighbors.push_back(&n2);
n2.neighbors.push_back(&n1);
assert(isGraphBipartite(graph) == true);
// 1 - 2
// | \ |
// 3 4
n2.neighbors.push_back(&n4);
n4.neighbors.push_back(&n2);
assert(isGraphBipartite(graph) == false);
}

```

### Find smallest k elements from an unsorted array

// Use a max heap of size k. We always maintain a heap of size k so that the smallest k elements are always there.

```

void printKSmallestInArray(vector<int> v, int k) {
    priority_queue<int> pq;
    int length = v.size();
    for (int i = 0; i < length; ++i) {
        pq.push(v[i]);
        if (pq.size() > k) pq.pop();
    }
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
    cout << endl;
}

```

```

void testPrintKsmallestInArray() {
    int a[] = {4, 3, 7, 2, 9, 3, 6, 1, 8, 1};
    vector<int> va(a, a + sizeof(a) / sizeof(int));
    printKSmallestInArray(va, 1);
    printKSmallestInArray(va, 2);
    printKSmallestInArray(va, 3);
    printKSmallestInArray(va, 4);
}

```

```

    printKSmallestInArray(va, 5);
    printKSmallestInArray(va, 6);
    printKSmallestInArray(va, 7);
    printKSmallestInArray(va, 8);
    printKSmallestInArray(va, 9);
    printKSmallestInArray(va, 10);
}

```

### Find kth element in row+column sorted 2D N\*N matrix

```
#define N 4
```

```

// A comparator class to be used by the heap (priority_queue).
// Each element of the heap is of type: pair<int, pair<int, int> >.
// A pair stores: pair<value, pair<x-axis, y-axis> > of the element.
class Comp {
public:
    bool operator()(pair<int, pair<int, int> > p1, pair<int, pair<int, int> > p2) {
        return p1.first > p2.first;
    }
};

// Start with the upper-left element. Everytime we encounter an element, we push its right
// and underneath neighbors to the heap (provided that they haven't been inserted before).
// When we popped the "kth" element from the heap, we've got the kth smallest.
int kthElementInSortedMatrix(int a[][N], int k) {
    if (k > N * N - 1) return -1;
    // We are using a min heap.
    // Each element is of the following format:
    // pair<element_value, pair<element_xaxis, element_yaxis> >
    priority_queue<pair<int, pair<int, int> >, vector<pair<int, pair<int, int> > >, Comp> pq;
    // The set is used to avoid pushing duplicate element to the heap.
    set<pair<int, int> > s;
    int row = 0, col = 0;
    pq.push(make_pair(a[row][col], make_pair(row, col)));
    s.insert(make_pair(row, col));
    while (1) {
        pair<int, pair<int, int> > p = pq.top();
        pq.pop();
        // Found the kth element, return its value.
        if (k == 0) return p.first;
        k--;
        row = p.second.first;

```

```

        col = p.second.second;
        // Only push element to heap if it hasn't been pushed before.
        if (row < N - 1 && s.find(make_pair(row + 1, col)) == s.end()) {
            s.insert(make_pair(row + 1, col));
            pq.push(make_pair(a[row + 1][col], make_pair(row + 1, col)));
        }
        if (col < N - 1 && s.find(make_pair(row, col + 1)) == s.end()) {
            s.insert(make_pair(row, col + 1));
            pq.push(make_pair(a[row][col + 1], make_pair(row, col + 1)));
        }
    }
}

```

```

void testKthElementInSortedMatrix() {
    int a[N][N] = { {1, 2, 3, 4},
                    {2, 3, 4, 5},
                    {3, 4, 5, 6},
                    {4, 5, 6, 7} };

    cout << kthElementInSortedMatrix(a, 0) << endl;
    cout << kthElementInSortedMatrix(a, 1) << endl;
    cout << kthElementInSortedMatrix(a, 2) << endl;
    cout << kthElementInSortedMatrix(a, 3) << endl;
    cout << kthElementInSortedMatrix(a, 4) << endl;
    cout << kthElementInSortedMatrix(a, 5) << endl;
    cout << kthElementInSortedMatrix(a, 6) << endl;
    cout << kthElementInSortedMatrix(a, 7) << endl;
    cout << kthElementInSortedMatrix(a, 8) << endl;
    cout << kthElementInSortedMatrix(a, 9) << endl;
    cout << kthElementInSortedMatrix(a, 10) << endl;
}

```

**Print the values of the nodes in a binary tree layer by layer**

```

struct TreeNode {
    TreeNode* left;
    TreeNode* right;
    int value;
    TreeNode(int v) {
        value = v;
        left = right = NULL;
    }
};

```

// This is essentially doing a BFS starting from the tree root node. Since we want to print  
 // "level-by-level", an integer is used together with the tree node to represent its level.  
 // E.g., root level = 0, its left & right children's level = 1, etc..

```
void pntTreeLevel(TreeNode* root) {
    if (!root) return;
    queue<pair<TreeNode*, int> > q;
    int lvl;
    set<TreeNode*> s;
    int preLvl = 0;
    TreeNode* temp;
    q.push(make_pair(root, 0));
    s.insert(root);
    while (!q.empty()) {
        pair<TreeNode *, int> p = q.front();
        q.pop();
        temp = p.first;
        lvl = p.second;
        // The current node's level is larger by one as compared to the previous level
        // => a new level is reached. Hence start printing on a new line.
        if (lvl == preLvl + 1) {
            cout << endl;
        }
        cout << temp->value << " ";
        preLvl = lvl;
        if (temp->left != NULL && s.find(temp->left) == s.end()) {
            q.push(make_pair(temp->left, lvl + 1));
            s.insert(temp->left);
        }
        if (temp->right != NULL && s.find(temp->right) == s.end()) {
            q.push(make_pair(temp->right, lvl + 1));
            s.insert(temp->right);
        }
    }
}
```

// Create a node with the input value and insert the node into the given BST.

```
void insertIntoBST(TreeNode*& root, int val) {
    TreeNode* node = new TreeNode(val);
    if (root == NULL) {           // empty tree
        root = node;             // our new node will become root.
        return;
    }
}
```



```

TreeNode* temp = root;
TreeNode* parent = NULL;
while (temp) {
    if (temp->value > val) {
        parent = temp;
        temp = temp->left;
    } else {
        parent = temp;
        temp = temp->right;
    }
}
if (parent->value > val) {
    parent->left = node;
} else {
    parent->right = node;
}
}

```

```

void testPntTreeLevel() {
    TreeNode* root = NULL;
    pntTreeLevel(root);
    cout << endl;

    insertIntoBST(root, 10);
    pntTreeLevel(root);
    cout << endl;

    insertIntoBST(root, 5);
    insertIntoBST(root, 15);
    insertIntoBST(root, 2);
    insertIntoBST(root, 12);
    insertIntoBST(root, 6);
    insertIntoBST(root, 16);
    pntTreeLevel(root);
}

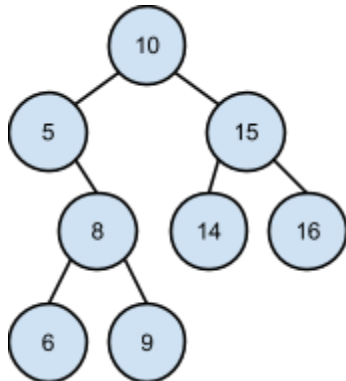
```

## Class 6

### FAQ

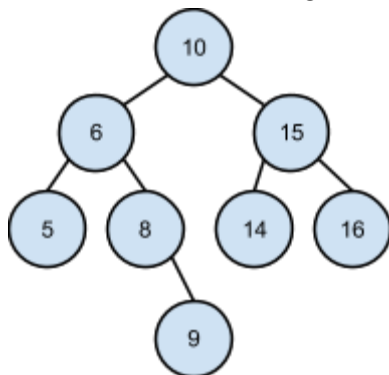
#### About balanced binary search tree

Definition from Wikipedia: “max depth is minimized”. So the below **is** a balanced BST, from that point of view.



A more restrained definition (AVL tree):

In an AVL tree, the heights of the two child subtrees of **any** node differ by at most one.



There's no standardized definition of a “balanced binary search tree”. However, in coding interview questions, usually by “determine if a tree is balanced”, they mean “AVL tree”, i.e., determine if for any node, the height of the left sub tree differs the height of the right sub tree by at most one.

#### A good hash function for string

```
int getHashIndex(string s, int hashTableSize) {  
    // Remember last time, we made a mistake by using int, which causes overflow.  
    unsigned int hash = 7; // Pick a prime number  
    int len = s.size();  
    for (int i = 0; i < len; ++i) {  
        hash = hash * 101 + s[i];  
    }  
}
```

```

        return hash % hashTableSize;
    }

```

When unsigned int “overflows”, it uses the below logic:

```

unsigned int c = a + b;    // suppose the result overflows and each int is 32 bit.
// Then, c = (a + b) mod (2^32)

```

## Content

### 2D Array (with comparison to 1D)

#### Dynamically Allocating 1D/2D Array

```

#define LEN 5
// Dynamically allocating 1D array on heap
int a[5] = {1, 2, 3, 4, 5};    // stack
int* a = new int[5];          // heap.
// Deallocating
delete [] a;
#define ROW 3
#define COL 4
// Dynamically allocating 2D array on heap
int* a[ROW] = new int[ROW][COL];

int** a = new int*[ROW];
for (int i = 0; i < ROW; ++i) {
    a[i] = new int[COL];
}

// Deallocating
for (int i=0; i<ROW;++i){
    delete[] a[i];
}
delete[] a;

```

#### Initializing 1D/2D Array

```

// Initializing 1D array. Determine which of the below lines are invalid.
int a1[10];    //ok

```

```

#define LEN 5;
int a2[LEN];           //ok
int a3[LEN] = {1, 2, 3, 4, 5}; //ok
int a4[LEN] = {0};      //ok quickly assign 0.
const int size = 2;      // ok
int a5[size];           // ok
a5[0] = 1;              // ok
int a6[size] = {0, 1};   // size has to be const
int a7[] = {1, 2, 3};    // ok
int a8[];               // not ok. [] needs to be initialized.
const int c = 2;
int a9[c];              // not ok.
int a10[c] = {1, 2};     // not ok.

```

**Rule of thumb when initializing array:** when you have “[]”, you must initialize the array content. When you have “[x]”, the x that describes the array length needs to be **an initialized constant**.

// Initializing 2D array. Determine which of the below lines are invalid.

```

#define ROW 3
#define COL 4
int a[ROW][COL] = {{1, 2, 3, 4},
                  {2, 3, 4, 5},
                  {3, 4, 5, 6}}; // ok
int a1[][COL] = {{1, 2, 3, 4},
                {2, 3, 4, 5},
                {3, 4, 5, 6}}; // ok?

```

// Question: How about 3D (nD) array?

```
int x[][1][2] = {{{1, 2}}};
```

// Why? Let's look at the picture later. This rule also applies to passing 2D array as argument.

```

void print2DArray(int a[][COL]) {
    // printing code..
}

```

```

int a2[][4] = {{1, 2, 3, 4},
              {2, 3, 4, 5},
              {3, 4, 5, 6}};

```

```

int a3[][]; // invalid.
int a4[][4]; // invalid.
int a5[3][4]; // ok
int a6[3][4] = {0}; // ok.
// on heap
int** a7 = new int*[ROW];

```

```

for (int i = 0; i < ROW; ++i) {
    a7[i] = new int[COL];
}
for (int i = 0; i < ROW; ++i) {
    for (int j = 0; j < COL; ++j) {
        a7[i][j] = 0;
    }
}
// convenient and super useful method - memset
// memset(start_position, value_in_char_8_bits, how_many_slots);

```

```

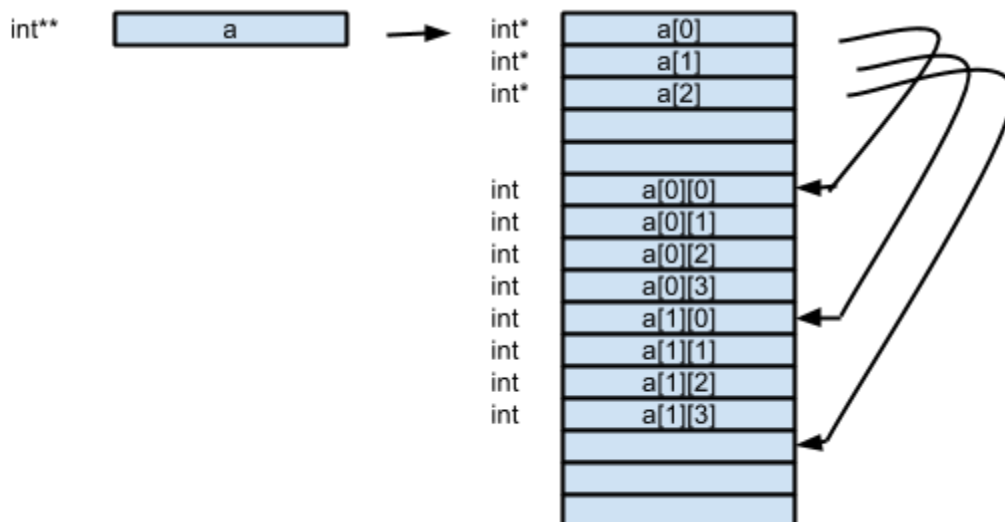
int* y = new int[3];
memset(y, 0, 3 * 4);

```

```

memset(a7[0], 0, sizeof(a7[0][0])*ROW*COL);
for (int i = 0; i < ROW; ++i) {
    for (int j = 0; j < 2; ++j) {
        assert(a7[i][j] == 0);
    }
}

```



```

// Question: let's try to modify a char array using memset.
char s[] = "aaaaaaaaaaaa"; // let's change the first 4 chars to b.
memset(s, 'b', 4);

```

// Question: Can we memset to all 1s as integer at once?

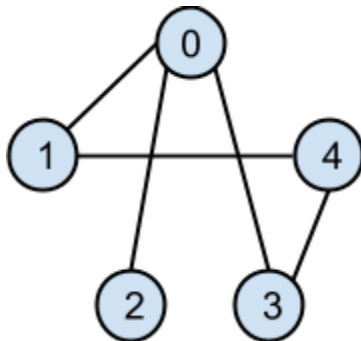
```
memset(a7[0], 1, sizeof(a7[0][0])*ROW*COL);
```

0000...001

000000010000000010000000100000001

## Graph Traversal Code

### BFS



```
#define SIZE 5
```

```
int a[SIZE][SIZE] = {{0, 1, 1, 1, 0},  
                     {1, 0, 0, 0, 1},  
                     {1, 0, 0, 0, 0},  
                     {1, 0, 0, 0, 1},  
                     {0, 1, 0, 1, 0}};
```

```
void bfs (int adj[][SIZE], int start, int N) {  
    queue<int> q;  
    int temp;  
    set<int> visited;  
    q.push(start);  
    visited.insert(start);  
    while(!q.empty()) {  
        temp = q.front();  
        cout<< temp <<" ";  
        q.pop();  
        for (int i = 0; i<N; i++) {  
            if (adj[temp][i] != 0 && visited.find(i) == visited.end()) {  
                q.push(i);  
                visited.insert(i);  
            }  
        }  
    }  
}
```

## DFS

### Iteration

// Very similar to BFS.

// Note 1: using stack.

// Note 2: "Only insert into visited set when printed". Because one current immediate neighbor

// could be on the path of some other traversal.

```
void dfs (int adj[][SIZE], int start, int N) {  
    stack<int> st;  
    int temp;  
    set<int> visited;  
    st.push(start);  
    while(!st.empty()) {  
        temp = st.top();  
        if (visited.find(temp) == visited.end()) {  
            cout<< temp <<" ";  
            visited.insert(temp);  
        }  
        st.pop();  
        for (int i = 0; i<N; i++) {  
            if (adj[temp][i] != 0 && visited.find(i) == visited.end()) {  
                st.push(i);  
            }  
        }  
    }  
}
```

```
for (int i = 0; i < 5; ++i) {  
    dfs(a, i, SIZE);  
    cout << endl;  
}
```

### Recursion

```
void dfsRecur(int adj[][SIZE], int start, int N, set<int>& visited) {  
    cout << start << endl;  
    visited.insert(start);  
    for (int i = 0; i < N; ++i) {  
        if (adj[start][i] == 1 && visited.find(i) == visited.end()) {  
            dfsRecur(adj, i, N, visited);  
        }  
    }  
}
```

```

    }
}

set<int> visited;
for (int i = 0; i < 5; ++i) {
    visited.clear();
    dfsRecur(a, i, SIZE, visited);
    cout << endl;
}

```

## Homework Practices (BFS + DFS)

Determine whether an undirected graph is Bipartite

```

struct GraphNode {
    int value;
    vector<GraphNode*> neighbors;
    GraphNode(int v) : value(v) {}
};

```

```

// Starts with a node. If this node hasn't been visited before, mark this node as "0".
// For all of its immediate neighbors, mark them as "1".
// For 2nd level neighbors, mark as "0" again (i.e., immediate neighbors of immediate
neighbors)
// ... And so on, until the BFS is over.
// Also store all visited nodes in the map to avoid duplicate.
bool bfsAndMark(GraphNode* node, map<GraphNode*, int>& nodeMark) {
    if (nodeMark.find(node) != nodeMark.end()) return true;
    queue<GraphNode*> q;
    q.push(node);
    nodeMark[node] = 0;
    while (!q.empty()) {
        GraphNode* curr = q.front();
        q.pop();
        int currMark = nodeMark[curr];
        int neighborMark = currMark == 0 ? 1 : 0;
        for (int i = 0; i < curr->neighbors.size(); ++i) {
            GraphNode* neighbor = curr->neighbors[i];
            if (nodeMark.find(neighbor) == nodeMark.end()) {
                q.push(neighbor);
                nodeMark[neighbor] = neighborMark;
            } else {
                if (nodeMark[neighbor] != neighborMark) {

```



```

        return false;
    }
}
}
return true;
}

```

```

bool isGraphBipartite(vector<GraphNode*> graph) {
    if (graph.empty()) return false;
    map<GraphNode*, int> nodeMark;
    for (int i = 0; i < graph.size(); ++i) {
        if (!bfsAndMark(graph[i], nodeMark)) {
            return false;
        }
    }
    return true;
}

```

```

void testIsGraphBipartite() {
    // 1 2
    // | \
    // 3 4
    GraphNode n1(1);
    GraphNode n2(2);
    GraphNode n3(3);
    GraphNode n4(4);
    n1.neighbors.push_back(&n3);
    n1.neighbors.push_back(&n4);
    n3.neighbors.push_back(&n1);
    n4.neighbors.push_back(&n1);
    vector<GraphNode*> graph;
    graph.push_back(&n1);
    graph.push_back(&n2);
    graph.push_back(&n3);
    graph.push_back(&n4);
    assert(isGraphBipartite(graph) == true);
    // 1 - 2
    // | \
    // 3 4
    n1.neighbors.push_back(&n2);
    n2.neighbors.push_back(&n1);
    assert(isGraphBipartite(graph) == true);
}

```

```

// 1 - 2
// | \ |
// 3  4
n2.neighbors.push_back(&n4);
n4.neighbors.push_back(&n2);
assert(isGraphBipartite(graph) == false);
}

map<string, int> m;
for (map<string, int>::iterator it = m.begin(); it != m.end(); ++it) {
    cout << it->first << it->second << endl;
}

```

### Print the values of the nodes in a binary tree layer by layer

```

struct TreeNode {
    TreeNode* left;
    TreeNode* right;
    int value;
    TreeNode(int v) {
        value = v;
        left = right = NULL;
    }
};

// This is essentially doing a BFS starting from the tree root node. Since we want to print
// "level-by-level", an integer is used together with the tree node to represent its level.
// E.g., root level = 0, its left & right children's level = 1, etc..
void printTreeLevel(TreeNode* root) {
    if (!root) return;
    queue<pair<TreeNode*, int> > q;
    int lvl;
    set<TreeNode*> s;
    int preLvl = 0;
    TreeNode* temp;
    q.push(make_pair(root, 0));
    s.insert(root);
    while (!q.empty()) {
        pair<TreeNode*, int> p = q.front();
        q.pop();
        temp = p.first;
        lvl = p.second;
        // The current node's level is larger by one as compared to the previous level
    }
}

```

```

// => a new level is reached. Hence start printing on a new line.
if (lvl == preLvl + 1) {
    cout << endl;
}
cout << temp->value << " ";
preLvl = lvl;
if (temp->left != NULL && s.find(temp->left) == s.end()) {
    q.push(make_pair(temp->left, lvl + 1));
    s.insert(temp->left);
}
if (temp->right != NULL && s.find(temp->right) == s.end()) {
    q.push(make_pair(temp->right, lvl + 1));
    s.insert(temp->right);
}
}
}

// Create a node with the input value and insert the node into the given BST.
void insertIntoBST(TreeNode*& root, int val) {
    TreeNode* node = new TreeNode(val);
    if (root == NULL) {           // empty tree
        root = node;             // our new node will become root.
        return;
    }
    TreeNode* temp = root;
    TreeNode* parent = NULL;
    while (temp) {
        if (temp->value > val) {
            parent = temp;
            temp = temp->left;
        } else {
            parent = temp;
            temp = temp->right;
        }
    }
    if (parent->value > val) {
        parent->left = node;
    } else {
        parent->right = node;
    }
}

```

```

void testPntTreeLevel() {
    TreeNode* root = NULL;
    pntTreeLevel(root);
    cout << endl;

    insertIntoBST(root, 10);
    pntTreeLevel(root);
    cout << endl;

    insertIntoBST(root, 5);
    insertIntoBST(root, 15);
    insertIntoBST(root, 2);
    insertIntoBST(root, 12);
    insertIntoBST(root, 6);
    insertIntoBST(root, 16);
    pntTreeLevel(root);
}

```

### Print all valid permutations of ( ) ( ) ( )

// Input: left: number of left parentheses left.

// Input: right: number of right parentheses left.

// Input: res: the formed string output to be printed out when left == right == 0.

```

void pntParentheses(int left, int right, string &res) {
    // Base case.
    if (left == 0 && right == 0) {
        cout << res << endl;
        return;
    }
    if (left > 0) {
        if (left >= right) {
            // If left is more than or equal to right, we can only put a left.
            res.push_back('(');
            pntParentheses(left - 1, right, res);
            res = res.substr(0, res.length() - 1);
        } else {
            // If left < right, we can either put a left, or put a right, both are legit.
            res.push_back('(');
            pntParentheses(left - 1, right, res);
            res = res.substr(0, res.length() - 1);
            res.push_back(')');
            pntParentheses(left, right - 1, res);
            res = res.substr(0, res.length() - 1);
        }
    }
}

```

```

    }
    // If no left parentheses left => Simple. Just append right parentheses.
} else {
    res.push_back(')');
    pntParentheses(left, right - 1, res);
    res = res.substr(0, res.length() - 1);
}
}

```

```

void testPntParentheses() {
    string res = "";
    pntParentheses(1, 1, res);
    res = "";
    pntParentheses(2, 2, res);
    res = "";
    pntParentheses(3, 3, res);
}

```

**Given infinite 25/10/5/1 cents, get number of ways to represent n cents**

```

// n: the input "n" cents to be represented.
// res: the total number of ways.
// v: the representation vector. E.g., represent 5 cents
//   use 5 * 1cent: {1, 1, 1, 1, 1}.
//   use 1 * 5cent: {5}.
// s: using a set to avoid duplicate results. E.g., {1, 1, 1, 1, 5}, and {1, 5, 1, 1, 1} are the same.
void numOfWaysNCents(int n, int& res, vector<int>& v, set<vector<int>> & s) {
    if (n == 0) {
        res++;
        vector<int> temp(v.begin(), v.end());
        // Sort each vector and use the sorted vector as the set key.
        // E.g., {1, 5}, and {5, 1}, after sorting => both are {1, 5}, hence avoiding duplicate.
        sort(temp.begin(), temp.end());
        if (s.find(temp) == s.end()) {
            s.insert(temp);
            for (int i = 0; i < temp.size(); i++) {
                cout<<temp[i]<<" ";
            }
            cout << endl;
        }
    }
    return;
}

```

```

    if (n >= 25) {
        v.push_back(25);
        numOfWaysNCents(n - 25, res, v, s);
        v.pop_back();
    }
    if (n >= 10) {
        v.push_back(10);
        numOfWaysNCents(n - 10, res, v, s);
        v.pop_back();
    }
    if (n >= 5) {
        v.push_back(5);
        numOfWaysNCents(n - 5, res, v, s);
        v.pop_back();
    }
    if (n >= 1) {
        v.push_back(1);
        numOfWaysNCents(n - 1, res, v, s);
        v.pop_back();
    }
}

```

```

void testNumOfWaysNCents() {
    int res = 0;
    vector<int> v;
    set<vector<int> > s;
    numOfWaysNCents(1, res, v, s);
    cout << endl;
    res = 0;
    v.clear();
    s.clear();
    numOfWaysNCents(5, res, v, s);
    cout << endl;
    res = 0;
    v.clear();
    s.clear();
    numOfWaysNCents(10, res, v, s);
    cout << endl;
    res = 0;
    v.clear();
    s.clear();
    numOfWaysNCents(25, res, v, s);
    cout << endl;
}

```

```

    res = 0;
    v.clear();
    s.clear();
    numOfWaysNCents(35, res, v, s);
}

```

## Class 7

### FAQ

#### Best practices when initializing vector

```

vector<int> v;
v[0] = 3;           // failure, because vector size is unknown
int a[] = {1, 2, 3};
vector<int> v_a(a, a + sizeof(a) / sizeof(int));
v_a[0] = 4;         // ok
vector<int> v_b;
v_b.push_back(1);   // ok
v_b.push_back(2);   // ok
v_b[1] = 4;

```

#### Using function argument (const int) to initialize array - invalid!

```

void myFunction(const int x) {
    int a[x] = {1, 2, 3};    // problem!
}

```

When the compiler sees your code, it doesn't know what x will be, thus can't assign corresponding space. Instead, these two are fine:

```

void myFunction() {
    const int x = 3;
    int a[x] = {1, 2, 3};    // ok
}
or,
void function2(int n) {
    int m[n];
    memset(m, 0, sizeof(int) * n);
    // or use a for loop.
    // or use stl "fill", which I will be discussing later in the lab sessions
}

```

## Return value using “pass by reference”

Don't forget the “&” sign.

```
void changeValue(int &x) {  
    x = 3;  
}
```

## Content - All that you need to know about bit operations (for interview)

### Basic logic operators:

AND &

$0 \& 0 = 0$

$0 \& 1 = 0$

$1 \& 1 = 1$

OR |

$0 | 0 = 0$

$1 | 0 = 1$

$1 | 1 = 1$

XOR ^

$0 \wedge 0 = 0$

$0 \wedge 1 = 1$

$1 \wedge 1 = 0$

## Print an unsigned integer in binary format

As a prerequisite to this class, let's introduce this function in order to help us verify things later.

```
void printNumInBinary(unsigned int n) {  
    int len = sizeof(n) * 8;  
    string res;  
    for (int i = 0; i < len; ++i) {  
        if (n & 1) res = "1" + res;  
        else res = "0" + res;  
        n >>= 1;  
    }  
    cout << res << endl;  
}
```

## Signed & Unsigned integers

Unsigned integer: 32 bit. 00000....00000  
0000..000111



111  $\Rightarrow \max = 2^3 - 1 = 7$

max: 1111.11111

Range? Answer:  $[0, 2^n - 1]$

Signed integer: 32 bit. The left most bit is the "sign".

LMB = 0  $\Rightarrow$  positive

LMB = 1  $\Rightarrow$  negative.

1 0 0 0.... 0 1 1 0

—

sign bit

When negative, how to know the actual value (in other words, how is a negative number stored)? **2's complement**:

if sign bit == 1, then

**2's complement = 1's complement then add 1.** so:

1's complement: 1  $\rightarrow$  0; 0  $\rightarrow$  1

2's complement (1 0 0 0 0 1 1 0) = (1111001+1)=1111010, then take negative value.

e.g.: -7

11111.1...1001  $\Rightarrow$  0000...00110 + 1 = 0000.0000111

signed int range:  $-2^{(n-1)} \sim 2^{(n-1)}-1$

(all 0's) (all 1's)

second smallest negative number? 1 0 0 0 0 0.... 0001

Q: 1000000..00000 (31 zeros)  $\Rightarrow$  1111.....111111 (31 ones) + 1 = 100000000000 (31 zeros) ( $2^{31}$ )  $\Rightarrow$  take negative  $\Rightarrow -2^{31}$

## Bit shifts

<<

*unsigned:*

unsigned int x = 3;

printNumInBinary(x); // ...000011

x <<= 1;

printNumInBinary(x); // ...000110

x <<= 200;

printNumInBinary(x); // compiler first checks if the number is larger than number of bits in an integer. If so, only rotate (n % 32).

*signed:*

short x = 16384;

printNumInBinary(x); // 010000...

x <<= 1;

```

cout << x << endl;
printNumInBinary(x);    // 10000... => negative number.
x <<= 100;
cout << x << endl;
printNumInBinary(x);    // 00000....

>>
unsigned:
unsigned int y = 32;
printNumInBinary(y);    // ...0000100000
y >>= 3;
printNumInBinary(y);    // ...00000000100
y >>= 200;
printNumInBinary(y);    // compiler first checks if the number is larger than number of bits in
an integer. If so, only rotate (n % 32).
signed
int x = 16384;
printNumInBinary(x);    // ...000010000...
x >>= 3;
printNumInBinary(x);
x = -16;
printNumInBinary(x);    // ....111111110000
x >>= 2;
printNumInBinary(x);    // ....1111111111100

11111..1110000    =>-16
1111..11111000    =>-8
11111..1111100    => -4

```

## Endianess (Big Endian, Little Endian)

### Concept

Big-endian: Least significant byte is stored on the high address.

Little-endian: Least significant byte is stored on the low address.

e.g., 32 bit integer, whose value = 1.

We all know that, its binary format is:

00000000 00000000 00000000 00000001.

MSB -----> LSB

(Most significant byte) (Least significant byte)

Let's say, the memory has an atomic element of 8 bits. That is, each smallest piece of memory that's represented by an address contains 8 bits (1 byte) of data.

0 0 0 0 0 0 0 0	←-----0x1234	(Most Significant byte)
0 0 0 0 0 0 0 0	←-----0x1235	
0 0 0 0 0 0 0 0	←-----0x1236	
0 0 0 0 0 0 0 1	←-----0x1237	(Least Significant byte)

0 0 0 0 0 0 0 0	←-----0x1234	(Most Significant byte)
0 0 0 0 0 0 0 0	←-----0x1233	
0 0 0 0 0 0 0 0	←-----0x1232	
0 0 0 0 0 0 0 1	←-----0x1231	(Least Significant byte)

### How to determine if a machine is big endian or little endian in c++ program?

1. Method 1: Use pointer casting.

E.g, int: 32 bit  
char: 8 bit.

Whenever we do a pointer cast, we **always cast from the low address.**

[illegible]

```
bool isLittleEndian2() {
    int *i;
    *i = 1;
    char* c;
    c = (char*) i;           // “cast” (short*) i;
    return (*c);             // If 1, low address stores LSB, hence Little endian.
}
```

## 2. Method 2: Use Union.

What is a union? A union is a data structure such that may contains several variable types, but at any moment, only one type can be used to represent the value of the data structure.

- A chunk of memory that is used to store variables of different data types.
- Once a new value is assigned to a field, the existing data is overwritten with the new data.
- Each data member begins at the same location in memory.
- The modification of one of the members will affect the value of all of them.

```
struct MyStruct {  
    int i;    // 4 bytes  
    char c;   // 1 byte  
};           // struct takes 5 bytes.
```

```
union MyUnion {  
    int i;        // 4 bytes  
    char c;       // 1 bytes  
}; // total: 4.
```

```
int isLittleEndian() {  
    MyUnion u;  
    u.i = 1;  
    return u.c;  
}
```

Little endian

```
00000000    <--- 0x1233  
00000000    <--- 0x1232  
00000000    <--- 0x1231  
(00000001)    <--- 0x1230  union base starts here.  u.c reads the nearest 8 bits from base.
```

Big endian

```
00000001    <--- 0x1233  
00000000    <--- 0x1232  
00000000    <--- 0x1231  
(00000000)    <--- 0x1230  union base starts here.  u.c reads the nearest 8 bits from base.
```

## Practices (including homework exercises)

### Count one bit in int

```
int numOfOnes1(unsigned int n) {
    int len = sizeof(n) * 8;
    int res = 0;
    for (int i = 0; i < len; ++i) {
        res += (n & 1);
        n >>= 1;
    }
    return res;
}

int numOfOnes2(unsigned int n) {
    int len = sizeof(n) * 8;
    int res = 0;
    for (int i = 0; i < len; ++i) {
        res += (n & 1);
        n >>= 1;
        if (n == 0) return res;
    }
    return res;
}

int numOfOnes3(unsigned int n) {
    int cnt = 0;
    while (n) {
        n &= (n - 1);    // n = n & (n - 1)
        cnt++;
    }
    return cnt;
}
```

```
110010100    n
110010011    n - 1
110010000    n
110001111    n - 1
110000000
```

### Determine whether a number x is a power of 2

```
bool isIntPowerOfTwo(int target) {
    if (target <= 0) {
```

```

    return false;
}
while ((target & 1) == 0) {
    target >>= 1;
}
return (target == 1);
}

```

target & (target - 1)  
000010000  
000001111 => 000000000

0000100100  
0000100011

1000 1000 1001 0110      n = ((n & 0xFF00) >> 8) | ((n & 0x00FF) << 8);  
                                         111111111000000000  
10001000 00000000    00000000 10001000  
                                 10010110 00000000  
1001 0110 1000 1000    n = ((n & 0xF0F0) >> 4) | ((n & 0x0F0F) << 4);

**1111000011110000      0xf0f0**  
0110 1001 1000 1000  
**1100110011001100    0xcccc    0011001100110011    0x3333**  
n = ((n & 0xcccc ) >> 2) | ((n & 0x3333) << 2);

1001 0110 0010 0010  
0110 1001 0001 0001

## Reverse bits of unsigned int

```

void reverseBits(unsigned int & n) {
    n = ((n & 0xFFFF0000) >> 16) | ((n & 0x0000FFFF) << 16);
    n = ((n & 0xFF00FF00) >> 8) | ((n & 0x00FF00FF) << 8);
    n = ((n & 0xF0F0F0F0) >> 4) | ((n & 0x0F0F0F0F) << 4);
    n = ((n & 0xCCCCCCCC) >> 2) | ((n & 0x33333333) << 2);
    n = ((n & 0xAAAAAAAA) >> 1) | ((n & 0x55555555) << 1);
}

```

## Print all subsets of a set

// Represent a set of size N as an "N bit integer". Then for each integer from 1 to N, its bit

```

// representation will characterize a unique subset - just include all the elements whose
// corresponding bit is set to 1. If it's set to 0, do not include.
void pntAllSubSets_iteration(string s) {
    int len = s.size();
    int val = (int)pow(2, len) - 1;
    for (int i = 1; i <= val; i++) {
        for (int j = 0; j < len; j++) {
            if ((i >> j) & 1) cout << s[j];
        }
        cout << " ";
    }
    cout << endl;
}

```

## Number of different bits between two positive integers

```

00000111100
11100000111
11100111011

```

```

int numBitsDifference(int x, int y) {
    int z = x ^ y;
    return countNumOneBitsInInt(z);
}

```

## Bit tricky manipulations

- set nth bit

$a | (1 \ll n)$

```

0000100000
a

```

-unset nth

$a \& (\sim(1 \ll n))$

```

00000100000
11111011111
a

```

0

-toggle nth

$a \oplus (1 \leq n)$

000010000

a 0

1

1

0

- turnoff rightmost one-bit

**$a \& (a - 1)$**

1000100101000

1000100100111

.....0000

- Isolate rightmost one-bit .....10000

10001001000

00000001000

**$a \& (a - 1) \wedge a$**

10001000000  $\wedge a$

00000001000

- right propagate rightmost one-bit .....10000

10001001000

10001001111

a-1:

10001000111

10001001000



1000100**1111**

**a | (a - 1)**

- isolate rightmost zero-bit     .....011111

1000100**0**1111

000000010000

0111011**1**0000

1000100**1**0000

0000000**1**0000

~a & (a + 1)

turn on right most zero-bit     ....01111

1000100**0**1111

100010010000

1000100**1**1111

a | (a + 1)

## Class 8

### C++ stl “string”

// Constructors

string s1("I want an offer");

string s2(s1);

string s3(s1, 2, 4);

string s4(s1.begin(), s1.begin() + 8);     // [start, end)

string s5(5, 'a');

cout << s1 << endl;

cout << s2 << endl;

cout << s3 << endl;

cout << s4 << endl;

cout << s5 << endl;

// Looping/Iteration

int len = s1.size();

for (int i = 0; i < len; ++i) {

    cout << s1[i];

```

}
cout << endl;
for (string::iterator it = s1.begin(); it != s1.end(); ++it) {
    cout << *it;
}
cout << endl;
for (string::reverse_iterator it = s1.rbegin(); it != s1.rend(); ++it) {
    cout << *it;
}
cout << endl;

```

// Frequently used APIs:

```

int len1 = s1.size();
int len2 = s1.length();
cout << len1 << " " << len2 << endl;
s1.resize(20);          // '\0'
cout << s1 << endl;
cout << s1.size() << endl;
for (int i = len; i < 20; ++i) {
    assert(s1.at(i) == '\0');
}
s2.resize(20, '-');
cout << s2 << endl;

```

```

s3.push_back('x');
cout << s3 << endl;
s3.push_back("YYY");    // Invalid.
cout << s3 << endl;

```

```

s3 += "YYY";
cout << s3 << endl;
s3 = "XXX" + s3;
cout << s3 << endl;
s3.append("ZZZ");
cout << s3 << endl;

```

```

s1.erase(0, 2);
cout << s1 << endl;
s1.erase(s1.begin() + 3);
cout << s1 << endl;
s1.erase(s1.begin() + 2, s1.begin() + 5);    // begin_iterator, end_iterator
cout << s1 << endl;

```

```
string t1 = "i love blue sky.";
string t2 = t1.substr(2, t1.length() - 2);
cout << t2 << endl;
```

```
assert(t1.find(t2) == 2);
assert(t1.find("water") == string::npos);
assert((int)t1.find("water") == -1);
```

// Comparison

```
string t3 = "abcd";
string t4 = "abcd";
assert(t3.compare(t4) == 0);
t4.erase(3, 1);
cout << t4 << endl;
assert(t3.compare(t4) > 0);
t3.erase(2, 2);
cout << t3 << endl;
assert(t3.compare(t4) < 0);
assert(t3.compare("aa") > 0);
assert(t3.compare("az") < 0);
```

```
string t5 = "xyz";
string t6 = "xyz";
string t7 = "xyv";
string t8 = "xyzz";
assert(t5 == t6);
assert(t5 > t7);
assert(t5 < t8);
```

// Sorting strings using comparisons

```
bool comp(string s1, string s2) {
    return s1 > s2;
}
```

```
vector<string> v;
v.push_back("liu");
v.push_back("wang");
v.push_back("wong");
v.push_back("li");
sort(v.begin(), v.end());
```

```

for (int i = 0; i < v.size(); ++i) {
    cout << v[i] << " ";
}
cout << endl;
sort(v.begin(), v.end(), comp);
for (int i = 0; i < v.size(); ++i) {
    cout << v[i] << " ";
}
cout << endl;

```

## C string, and common string.h APIs

### c\_str()

Essentially, an array of chars ending with '\0'.

```
#include <string.h>
```

```

string c1 = "abc";
c1[1] = '\0';
cout << c1 << endl;
cout << c1.length() << endl;

```

```

char c2[] = "abcd";
cout << strlen(c2) << endl;
c2[1] = '\0';
cout << strlen(c2) << endl;
cout << c2 << endl;

```

```

char c3[] = "xyz";
*(c3 + 1) = 'x';
*(c3 + 2) = 'x';
*(c3 + 3) = 'x';
cout << c3 << endl;
cout << strlen(c3) << endl;           // is this value right?
*(c3 + 4) = '\0';
cout << strlen(c3) << endl;

```

// Comparison

```

char c4[] = "opq";
char c5[] = "opq";
assert(strcmp(c4, c5) == 0);
char c6[] = "cp";

```

```
assert(strcmp(c4, c6) > 0);
assert(strcmp(c6, "cz") < 0);
```

```
// Concatenation
```

```
char x1[] = "aaa";
char x2[] = "bbb";
char * x3 = strcat(x1, x2);
cout << x3 << endl;
cout << x1 << endl;
// A return type may be skipped
strcat(x1, x2);
cout << x1 << endl;
```

```
// Find (the first occurrence of the 2nd arg in the 1st arg))
```

```
assert(strstr("aa", "bb") == NULL);
char x4[] = "cb";
char x5[] = "c";
cout << strstr(x4, x5) << endl;
cout << strstr("cb", "b") << endl;
cout << strstr(x1, x2) << endl; // x1: aaabbb, x2: bbb
```

```
// Copy from source to destination
```

```
char n1[] = "xxxx";
// Destination size has to be bigger, why?
// Otherwise, not enough memory to hold data, may overwrite
// other variable memory.
char n2[30];
char* n3 = strcpy(n2, n1);
cout << n2 << endl;
cout << n3 << endl;
```

```
// Find any of the input chars in the given string
```

```
char x6[] = "This is so awesome.";
char x7[] = "si";
char *res = strpbrk(x6, x7);
while (res != NULL) {
    cout << *res << " ";
    res = strpbrk(res + 1, x7);
}
cout << endl;
```

```
// When is this useful? Example: Parsing a book.
```

```

// Let's try some of previous knowledge:)
char x8[] = "this is very awesome. However, that is not good! LOL";
char x9[] = ". ,!";
char *res2 = strpbrk(x8, x9);
vector<string> ve;
char *prev = x8;

while (res2 != NULL) {
    string s(prev, res2);
    prev = res2 + 1;
    // Why is this needed?
    if(s.size() != 1 && strstr(x9, s.c_str()) == NULL) {
        ve.push_back(s);
    }
    res2 = strpbrk(res2 + 1, x9);
}
string extra(prev, prev + strlen(prev));
ve.push_back(extra);
for (vector<string>::iterator it = ve.begin(); it != ve.end(); ++it) {
    cout << *it << " ";
}
cout << endl;

```

```

// A more convenient way to parse - strtok
char y1[] = "wow, that - was a little hard. Is there a better way? Huh!";
char y2[] = ", .?!-";
char* yres;
yres = strtok(y1, y2);    // "wow\0"
while (yres != NULL) {
    cout << yres << " ";    // "that\0"
    yres = strtok(NULL, y2);
}

```

## memset, memcpy, memcmp, and struct memory layout

We learned memset last time:

```
//memset(start_pos, value_in_bytes, number_of_bytes_to_set);
```

```

unsigned int a1[5];
memset(a1, 0xFF, 5 * sizeof(int));
for (int i = 0; i < 5; ++i) {

```

```

    assert(a1[i] == pow(2, 32) - 1);
}
int a2[5];
memset(a2, 0xFF, 5 * sizeof(int));
for (int i = 0; i < 5; ++i) {
    assert(a2[i] == -1);
}

```

// memcpy

```

struct CommData {
    char ssid[20]; // what if this is size 21/22/23/24? What's the effect on memory usage?
    char password[5];
    int version;
}; // How is this laid out on memory?

```

```

CommData cd;
cout << sizeof(cd) << endl; // why total 32 bytes?
char name[] = "myssidname";
char pw[] = "1234567";
memcpy(cd.ssid, name, strlen(name) + 1);
cout << cd.ssid << endl;
cd.version = 1;
memcpy(cd.password, pw, strlen(pw) + 1);
char pw2[] = "12345678";
cout << cd.password << endl;
cout << cd.version << endl;
memcpy(cd.password, pw2, strlen(pw2) + 1);
cout << cd.password << endl;
cout << cd.version << endl; // why it's 0?

```

x	x	x	x
x	x	x	x
x	x	x	x
x	x	x	x
x	x	x	x
x	x	x	y
y	y	y	y
z	z	z	z

```
// memcmp
char r1[] = "hfalsdfasdf";
char r2[] = "hfalsdFAsdf";
assert(memcmp(r1, r2, sizeof(r2)) > 0);
assert(memcmp(r1, "hfalsdfasdf", sizeof(r2)) == 0);
assert(memcmp("aaa", "hfalsdfasdf", sizeof(r2)) < 0);
```

## Homework Practice

**Deduplicate same adjacent chars in a string**

e.g., "aabbbbbcbddz" -> "abcdz"

```
string dedupSameAdjacentChars(string input) {
    int i = 0;
    int j = 1;
    while (j < input.size()) {
        while (input[j] == input[i] && j < input.size()) {
            j++;
        }
        if (j < input.size()) {
            input[++i] = input[j++];
        }
    }
    return input.substr(0, i + 1);
}
```



```

}

void testDedupSameAdjacentChars() {
    string s = "";
    assert(dedupSameAdjacentChars(s) == "");
    s = "a";
    assert(dedupSameAdjacentChars(s) == "a");
    s = "ab";
    assert(dedupSameAdjacentChars(s) == "ab");
    s = "aab";
    assert(dedupSameAdjacentChars(s) == "ab");
    s = "abb";
    assert(dedupSameAdjacentChars(s) == "ab");
    s = "abcc";
    assert(dedupSameAdjacentChars(s) == "abc");
    s = "aabcccd";
    assert(dedupSameAdjacentChars(s) == "abcd");
    s = "aaabbbccccddddd";
    assert(dedupSameAdjacentChars(s) == "abcd");
}

```

### Update all space in a string to “%20”

```

string replaceStrWithPercentage20(string s) {
    int cnt = 0;
    int len = s.length();
    for (int i = 0; i < len; i++) {
        if (s[i] == ' ') cnt++;
    }
    s.resize(len + 2 * cnt);
    int idx = len - 1;
    int idx2 = s.length() - 1;
    while (idx >= 0) {
        if (s[idx] != ' ') {
            s[idx2--] = s[idx--];
        } else {
            s[idx2--] = '0';
            s[idx2--] = '2';
            s[idx2--] = '%';
            idx--;
        }
    }
    return s;
}

```

```

}

void testReplaceStrWithPercentage20() {
    string s = "";
    assert(replaceStrWithPercentage20(s) == "");
    s = " ";
    assert(replaceStrWithPercentage20(s) == "%20");
    s = "a ";
    assert(replaceStrWithPercentage20(s) == "a%20");
    s = " b";
    assert(replaceStrWithPercentage20(s) == "%20b");
    s = "a b";
    assert(replaceStrWithPercentage20(s) == "a%20b");
    s = "a b c ";
    assert(replaceStrWithPercentage20(s) == "a%20b%20c%20");
}

```

**Reverse words in a sentence: “hello world lol” -> “lol world hello”**

```

void reversestr (string & s, int left, int right) {
    int len = right - left + 1;
    char temp;
    for (int i = 0; i < len/2; i++) {
        temp = s[left + i];
        s[left + i] = s[right - i];
        s[right - i] = temp;
    }
}

```

```

string reverseword (string s) {
    int len = s.size();
    reversestr(s, 0, s.size()-1);
    int start = 0;
    int end;
    for (int i = 0; i < len; i++) {
        if (s[i] == ' ') {
            end = i - 1;
            reversestr(s, start, end);
            start = i + 1;
        }
    }
    if (s[len-1] != ' ') {        // For the last word.
        reversestr(s, start, len-1);
    }
}

```

```

    }
    return s;
}

void testReverseword() {
    string s = "";
    assert(reverseword(s) == "");
    s = "hello";
    assert(reverseword(s) == "hello");
    s = "hello world";
    assert(reverseword(s) == "world hello");
    s = "hello world lol";
    assert(reverseword(s) == "lol world hello");
}

```

**Determine if a string is a substring of another**

```

char* myStrstr(char* s, char* t) {
    if (!*t) return s;
    if (!*s) return NULL;
    int lent = 0;
    char *t1 = t;
    while (*t1) {
        t1++;
        lent++;
    }
    char* s2 = s + lent - 1;
    char* s1;
    while (*s2) {
        s1 = s;
        t1 = t;
        while (*s1 == *t1) {
            s1++;
            t1++;
            if (!*t1) {
                return s;
            }
        }
        s++;
        s2++;
    }
    return NULL;
}

```

```

void testMyStrstr() {
    char s[] = "";
    char t[] = "";
    assert(myStrstr(s, t) != NULL);
    char s1[] = "a";
    char t1[] = "";
    assert(*myStrstr(s1, t1) == 'a');
    char s2[] = "";
    char t2[] = "a";
    assert(myStrstr(s2, t2) == NULL);
    char s3[] = "a";
    char t3[] = "a";
    assert(*myStrstr(s3, t3) == 'a');
    char s4[] = "a";
    char t4[] = "ab";
    assert(myStrstr(s4, t4) == NULL);
    char s5[] = "abc";
    char t5[] = "bc";
    assert(*myStrstr(s5, t5) == 'b');
    char s6[] = "abcdasdfasdfasdf";
    char t6[] = "cda";
    assert(*myStrstr(s6, t6) == 'c');
    char s7[] = "abcdasdfasdfasdf";
    char t7[] = "cdae";
    assert(myStrstr(s7, t7) == NULL);
    char s8[] = "abcdasdfasdfasdf";
    char t8[] = "fasdfasdf";
    assert(*myStrstr(s8, t8) == 'f');
    char s9[] = "abcdasdfasdfasdf";
    char t9[] = "fasdfasdfa";
    assert(myStrstr(s9, t9) == NULL);
}

```

## Class 9

### FAQ

#### Pointer cast vs. Direct type cast

Pointer cast *\*always\** takes the **lowest address**.

Type cast *\*always\** takes the **least significant byte(s)**.

E.g.,

```
int i = 100;
char c = (char) i;
cout << c << endl;    // c always equal to i's value (lower 8 bits).
char* p = (char*) &i;
cout << *p << endl;    // *p depends on Question? (endianess)
```

## Some Frequent Keywords

### const

The part associated with “const” is not to be changed. Syntax wise, “const” applies to its immediately left, if nothing left then immediately right.

```
char * const ptr;    // ptr cannot point to other address
char x = 'a';
ptr = &x;            // Error! ptr cannot point to others.
```

```
const char * ptr = "asdf";    // The content pointed by ptr cannot be changed
*ptr = "kkk";                // Error
```

Also,

```
const char cch = 'A';        // cch is not to be changed.
char *pch2 = &cch;           // Therefore, doesn't allow others to point to it. => Error!
char * const pch3 = &cch;    // Error!
```

When const is applied to a function inside a class, it means that the function cannot modify the class variable. e.g.:

```
class MyClass {
private:
    int value_;
public:
    void MyConstFunction() const;           // this function does not change "value"!
    void NotConstFunction();
}

void MyClass::MyConstFunction() const {
    value_++;           // NOT allowed
}
```

```
void MyClass::NotConstFunction() {
    value_++;          // allowed
}
```

### static/extern

- static function (not class method), or globally declared static variable outside any function: only visible within this file. Source code from other files of the project cannot access the variable - internal linkage.
  - By default, variables have external linkage, which equals to “**extern**”.
  - See example. (main.c includes cite.c => transitive closure)
    - Note the difference between C and C++
      - C: by default, a declaration is extern.
      - C++: cannot rely on default - considered redefinition.
- within function: means that the variable lasts till program terminates

```
int main() {
    int x = 7;
    function1(x);      // => print 7.
    function1(x);      // => print 7.
    function2(x);      // => print 7.
    function2(x);      // => print 8.
    function2(x);      // => print 9.
}
```

```
void function1(int input) {
    int x = input;
    cout << x << endl;
    x++;
}
void function2(int input) {
    static int y = input;    // this variable will be remembered even the function is out of scope
    cout << y << endl;
    y++;
}
```

- within class:
  - class variable, same value for every instance, exist for any instance or without instance
  - static functions: operate static variables

```

class MyClass2 {
public:
    int getValue() {
        return value;
    }
    void setValue(int v) {
        value = v;
    }
    static int getSValue() {
        return s_value;
    }
    static void setSValue(int v) {
        s_value = v;
    }
private:
    int value;
    static int s_value;
};

```

```

MyClass2 b;
cout << b.getSValue() << endl;
cout << MyClass2::getSValue() << endl;
MyClass2::setSValue(2);
cout << MyClass2::getSValue() << endl;
cout << b.getSValue() << endl;
b.setSValue(10);
cout << MyClass2::getSValue() << endl;
cout << b.getSValue() << endl;
b.setValue(100);
cout << b.getValue() << endl;
MyClass2 c;
cout << c.getSValue() << endl;    // Already initialized
cout << c.getValue() << endl;    // Uninitialized

```

## virtual

Virtual is used under the context of C++ class inheritance, and polymorphism.

### What is polymorphism?

Pointer to a derived class is type compatible with a pointer to its base class.

E.g.:

```

class CPoly{

```

```

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

class CRect:public CPoly{
public:
    int area(){
        return width * height;
    }
};

class CTria:public CPoly{
public:
    int area(){
        return width * height / 2;
    }
};

int main(int argc, char** argv) {
    CRect rect;
    CTria tria;
    CPoly poly;
    CPoly* poly1;
    CPoly* poly2;
    CPoly* poly3;
    poly1 = &rect;
    poly2 = &tria;
    poly3 = &poly;
    poly1->setvalue(4, 5);
    poly2->setvalue(4, 5);
    poly3->setvalue(4, 5);
    cout << poly1->area() << endl;    // 20
    cout << poly2->area() << endl;    // 10
    cout << poly3->area() << endl;    // 0
}

```



## How is polymorphism achieved?

Through the use of virtual function. Let's first look at what happened if we don't declare virtual in the above case. => ok, all 0s. But why?

Let's see this illustration:

```
class Base {  
private:  
    int a;  
public:  
    virtual void f1() {};  
    virtual void f2() {};  
};
```

```
class D1 {  
public:  
    void f1() {};  
};
```

```
class D2 {  
public:  
    void f2() {};  
};
```

Each class maintains a vtable, which is a lookup table for function calls. I.e., which "version" of the function shall I call?

Then, the compiler also adds a pointer to the base class, called the "vPointer". This pointer will point to the vtable of the **most derived** class.

Also note that, when a subclass is instantiated, its base class will be included as the "top portion" of the subclass.

The (high level) memory assignment looks like this:

Base:

*vp*tr to vtable

int a

*vtable* for Base:

f1() -> Base::f1()

f2() -> Base::f2()

D1:

(everything of Base) // now pointed by b.

*vtable* for D1:

f1() -> D1::f1()

```
f2() -> Base::f2()
```

So consider the following code:

```
D1 d1;  
Base* b = &d1;  
b->f1();
```

Therefore, b is a pointer that only points to the (everything of Base) portion. This means that the vptr is there (usable) which points to vtable of D1. Thus, achieving polymorphism.

### **A related topic, why does destructor under inheritance need to be declared virtual?**

Destructors are called automatically based on the reverse order of construction. Thus, Derived->Base.

```
class Base{  
public:  
    Base(){  
        cout<<"constructing base"<<endl;  
    }  
    virtual ~Base(){  
        cout<<"deleting base"<<endl;  
    }  
};  
class Derived : public Base{  
public:  
    Derived() {  
        cout<<"constructing derived"<<endl;  
    }  
    ~Derived(){  
        cout<<"deleting derived"<<endl;  
    }  
};  
  
int main() {  
    Derived* d = new Derived();  
    delete d;  
    Base* b = new Derived();  
    delete b;    // if not virtual, there's no vtable, hence only b's destructor is called.  
}
```

## malloc

```
int *ptr;  
ptr = (int*) malloc(10*sizeof(int)); // valid in both C and C++  
ptr = malloc(10*sizeof(*ptr));      // valid in C only.
```

Return type of malloc is void\* => it's the programmer's job to cast it to the desired type.

malloc 2D array:

```
char ** c = (char**) malloc(N*sizeof(char*));  
for (int i = 0; i < N; i++) {  
    c[i] = (char *) malloc(M*sizeof (char));  
}
```

## malloc vs new

- Upon failure,
  - malloc will return NULL pointer,
  - new throws bad\_alloc exception. E.g.:

```
int* ptr;  
ptr = (int*) malloc(1000000000000000000 * sizeof(int));  
if (ptr == NULL) {  
    cout << "failed" << endl;  
}
```

```
for (int i = 0; i < 100000000000; ++i) {  
    try {  
        int* cpp_ptr= new int[100000000000000];  
    } catch (bad_alloc& ba) {  
        cout << "fail again" << endl;  
        break;  
    }  
}
```

- Return type:
  - malloc returns void\*, which you need to cast to the desired type
  - new return the "ready-type" pointer.
- Whether to call constructor
  - new is a C++ term => hence calls constructor of the class

- malloc doesn't call constructor, only allocate certain size of memory.
- Upon delete
  - malloc: free
  - new: delete, delete []

```
int* i = (int*) malloc(10*sizeof(int));
free(i);
int * j = new int[10];
delete [] j;
int * k = new int;
delete k;
```

### Case study: allocating memory for 2D array with only one malloc call

```
int** my2DAlloc(int row, int col) {
    int ** res = (int**) malloc (row * sizeof(int*) + row * col * sizeof(int));
    if (!res)
        return NULL;
    int* base = (int*)(res + row);
    for (int i = 0; i < row; ++i) {
        res[i] = base + i * col;
    }
    return res;
}

int main() {
    int row = 2;
    int col = 3;

    int** a = my2DAlloc(row, col);    // using my2DAlloc to allocate a
    int* b = (int*) malloc(10 * sizeof(int));    // Then, immediately allocate 1D array b.

    for (int i = 0; i < 10; ++i) {
        b[i] = i;
    }
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            a[i][j] = i * col + j;
        }
    }
    for (int i = 0; i < 10; ++i) {
        cout << b[i] << " ";
    }
}
```

```

    }
    cout << endl;
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            cout << a[i][j] << " ";
        }
        cout << endl;
    }
}

```

What if the following code? Our b array got overwritten!!!

```

int** my2DAlloc(int row, int col) {
    int ** res = (int**) malloc (row * sizeof(int*) + row * col * sizeof(int));
    if (!res)
        return NULL;
    int** base = res + row;
    for (int i = 0; i < row; ++i) {
        res[i] = (int*) (base + i * col);
    }
    return res;
}

```

Why? Analysis:

- I'm running a 64-bit machine. Therefore, **all pointers** are 64 bits.
- We already know all ints are 32 bits.
- Some extra memory is allocated (usually 1 byte) during malloc to store metadata (such as allocated size, etc) - e.g., what if we want to call free? How many bytes to free? This metadata tells us.
- When base is of int\*\* type, then base + x will result in shifting x \* 64 bits, since base points to "pointer type". (the incorrect case)
  - When base is of int\* type, then the base + x will result in shifting x \* 32 bits, since base now points to "int type". (the correct case).

Let's understand what happened, again, our favorite memory picture:

Case 1 (correct):

