2. Intro to C++

Why Start with Hardware?

- Programs are fundamentally executed by hardware
- Understand hardware can simplify your understanding of programming language
- Great programmers need to understand hardware to write high performance code

C++ Fundamental Types

int (32 bits: -2^31 ~ 2^31 - 1), short (16 bits), long int (>= 32 bits), float (7 dec), double (15 dec), long double, bool, void, char

#include <iostream>

using namespace std;

Tells the preprocessor to include functions of these header files like cout, cin, etc in the program

Basic C++ Program

```cpp
// hello.cpp -- "Hello, world" C++ program

#include <iostream>
using namespace std;

int main () {

    cout << "Hello " << endl;
    cout << "world!" << endl;
    return (0);
}
```

Functions

- Logical code grouping
- Easier to understand and debug
- Easier to reuse code.
- [result type][function name]([param list])

	Name	Parameters	Result	Local Vars	Declared before used
Must have?	O	X	X	X	O

Declaration vs Definition

Declaration:

```
int foo (int x, char y);
int foo (int, char);
int foo (int /* student number */,
        char /* group */); // Param name is not required
```

Definition:

```
int bump (int mark) { /* mark is a formal param */
        if (mark < 90) {
        mark += 10;
        }
        return mark;
}
```

Pass-by-value vs Pass-by-pointer

```
void bump_ref (int& mark){
  if (mark < 90) {
    mark += 10;
  } else {
    mark = 100;
  }
}
```

```
int main () {
  int yourmark = 72;

  bump_ref(yourmark);
  cout << yourmark << endl;

  bump_ref(yourmark);
  cout << yourmark << endl;
}
```

- Reference is an “alias”, or alternate name, to an existing variable
- mark in bump_ref() is an alias of yourmark
- The primary use of reference: function formal parameter
- So the function works on the original copy of the var.
- Reference is a safer, but less powerful than pointer because of strict type checking by compiler
- Cannot assign an ‘int’ variable to a ‘char&’ reference, but pointers can be type-casted to anything

```
1 int a = 0, b = 10;
2 int& ra = a;
3 cout << ra << endl;
4 ra = b;
5 cout << a <<endl;
```

- Reference cannot be reassigned
- What happens to this code?
- It must be assigned at initialization (E.g., “int& a;” won’t compile)
- Reference cannot be assigned to NULL
- Reference variables do NOT have separate memory allocation

Expression Precedence

!, sizeof() > Arithmetic (1. *, /, %, 2. +, -) > Relational (1. <. >, <=, >=, 2. ==, !=) > Boolean (1. &&, 2. ||) > Assignment (=, +=, -=, *=, /=, %=)

a++ vs ++a

- When we assign a++ to any variable, it takes the current value of a, and then increments a.
- When we assign ++a to any variable, it first increments the value of a, and then assigns it to the variable.

int a=1; //storing a number in a
 	
int x = a++;
	int a=1; //storing a number in a
	
	int x = ++a;

x = 1, a = 2	x = 2, a =2

Switch Statements

```
switch(expression){
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Continue

- ends the current operation of the loop and returns to the condition at the top of the loop. Below code, 0 to 9 is printed except 4.

```
 for (int i = 0; i < 10; i++) {
   if (i == 4)
     continue;
   cout << i << endl;
 }
```

3. Program Organization

Why Multiple Files?

- Speed up compilation
- Only compiles a .cc file when it is changed
- Keep your code organized
- Separate interface from implementation (E.g. function prototype from definition)

- **linked_list.h:** Header file - declarations of functions & data structures

```
1 #ifndef LINKED_LIST_H
2 #define LINKED_LIST_H
3
4 #include <stdlib.h> /* Question: what does include do? */
5 struct Node
6 {
7   int data;
8   struct Node *next;
9 };
10
11 typedef struct Node Node;
12 /* Always insert at the beginning, return the new head.
13  * Return NULL on error! */
14 Node* list_insert (Node* head, int value);
15
16 /* Return a pointer of node if the value matches.
17  * Return NULL if the value is not found. */
18 Node* list_search (Node* head, int value);
19 #endif
```

- **insert.cc:** Implementation of list_insert function

```
1 #include "linked_list.h"
2
3 Node* list_insert (Node* head, int val)
4 {
5     Node* newhead = (Node*) malloc (sizeof(Node));
6     if (newhead == NULL)
7     { // Error!!!
8         return NULL;
9     }
10    newhead->data = val;
11    newhead->next = head;
12    return newhead;
13 }
```

- **search.cc**: Implementation of list_search function

```
#include "linked_list.h"
Node* list_search (Node* head, int val)
{
    for (Node* cur = head; cur != NULL; cur = cur->next)
    {
        if (cur->data == val) // found
        {
            return cur;
        }
    }
    return NULL;
}
```

- **main.cc**: Implementation of main function

```
#include <iostream>
#include "linked_list.h"

using namespace std;

int main() {
    int value;
    Node* head = NULL;
    /* Build a link list: 5->187->23->3->null */
    head = list_insert (head, 3);
    head = list_insert (head, 23);
    head = list_insert (head, 187);
    head = list_insert (head, 5);

    cout << "Input an integer:" << endl;
    cin >> value;
    if (list_search (head, value) != NULL)
    {
        cout << value << " is found in the list!" << endl;
    } else {
       cout << value << " is NOT found!" << endl;
    }
}
```

Compilation

> g++ -c insert.cc –o insert.o

- -c: compile source files without linking
- Generates object file (insert.o)

> g++ -c search.cc –o search.o

> g++ -c main.cc –o main.o

> g++ insert.o search.o main.o –o linked_list

- This steps is linking: it links the 3 object files, and outputs an executable file linked_list
- -Wall: asks for all warnings

#include

- Copy the content of file into the current file
- You can include .cc file instead of .h but it is a very bad practice because .h is built for prototypes not implementations (e.g. Function declaration instead of definition, class interface instead of implementations)
- **#include <filename>** The preprocessor searches in include path list
- **#include "filename"** The preprocessor also searches in the same directory

#ifndef

- Prevent the same .h file from being included multiple time.
- Once a header is included, it skips declaring.
- #ifndef <token> /* code */ #else /* code to include if the token is defined */ #endif

```
14  #ifndef globals_h
15  #define globals_h
16  #include<string>
17
18  // These arrays should make it easier to check that a name is not a reserved word
19  // and that an entered shape type is valid
20  #define NUM_KEYWORDS 7
21  string keyWordsList[7]={"all", "maxShapes", "create", "move", "rotate", "draw", "delete"};
22
23  #define NUM_TYPES 4
24  string shapeTypesList[4]={"circle", "ellipse", "rectangle","triangle"};
25
26  #endif /* globals_h */
```

4. Compilation

Compilation Lifecycle

Preprocess: g++ -E

- Macro substitution
- Stripping comments
- File inclusion

Compilation: g++ -S

- Input: c++; Output: assembly code

Assembly: g++ -c

- Input: assembly code; Output: binary code (object file)

Linking:

- Input: object file; Output: executable file
- Symbol resolution
- Relocation

Object file vs. executable

- They are all binary files
- They all contain binary machine instructions
- Object file cannot be directly executed, executable can (E.g., you cannot execute main.o)
- Object files contain unresolved references (e.g. external libraries, static data)
- Data in object file has illegal, but relocatable, address
- Executable file is the output of linking object files

(GNU) make

- A utility that executes the necessary compilation
- Makefile is a file that specifies what compilations are necessary
- Makefile format: rules

```
target: dependencies
[tab] system command
```

Compilation Example

```
linked_list: main.o insert.o search.o
	g++ main.o insert.o search.o –o linked_list

main.o: main.cc linked_list.h
	g++ -c -Wall main.cc -o main.o

insert.o: insert.cc linked_list.h
	g++ -c -Wall insert.cc -o insert.o

search.o: search.cc linked_list.h
	g++ -c -Wall search.cc -o search.o

clean:
	rm *.o linked_list
```

* rm removes files

Make: the algorithm

Let T be a target (e.g., all)
If Makefile does not have a rule for making T then
  if a file named “T” already exists
	then there is nothing todo
  else report error
else
  choose the first rule for making T
	make each dependency for that rule
	if T exists and is newer than each dependency
	then report “T is up to date”
	else make T by executing the commands
end

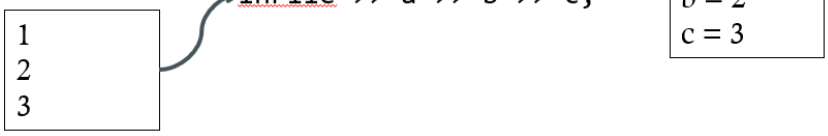
5 and 6. Input and Output

File input

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

ifstream inFile("filename");
```

or

```
ifstream inFile;
inFile.open ("filename");

int a,b,c;
inFile >> a >> b >> c;
```



* if stands for input and file

File output

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

ofstream outFile("filename");
```

or

```
ofstream outFile;
outFile.open ("filename");

int a = 6;
outFile << "a = " << a << endl;
```

a = 6

Two ways to open a file for output

- Open for writing (default): If the file already exists, it is first erased; Otherwise it is created (loses data)
- Open for append: Use: outFile.open("filename", ios::app); Write adds to the end of the file (doesn't lose data)

Done with file?

inFile.close(); or outFile.close();

Buffering

- Output will not be immediately written to file
- Instead, temporarily store the output in buffer; write out to file periodically in chunks
- To force output: outFile.flush(); or outFile << endl;

```
outFile << "Hello \n";
..
outFile << "Hello again!";
```

```
  Buffer:
Hello
Hello again!
```

Slow!

file

Detecting input failures

- **cin.fail()**
- Return true when any error has occurred
- inFile is an object, it has an error state flag
- The flag is set whenever there is a failure
- You can also use it on cin, ofstream, cout

```
int main() {
  int a;
  float b;
  ifstream inFile;
  inFile.open ("input.txt");
  if (inFile.fail()){
    /* fail() returns true if failure occurred! */
    cout << "Cannot open file: input.txt" << endl;
    return 1;
  }
  inFile >> a;
  inFile >> b;
  cout << "a = " << a << ", b = " << b << endl;
  return 0;
}
```

```
int main() {
  ..
  inFile >> a;
  if (inFile.fail()) {
    cout << "Cannot read a!" << endl;
    return 1;
  }
  inFile >> b;
  cout << "a = " << a << ", b = " << b << endl;
  return 0;
}
```

- Example: input.txt only contains an integer

```
int main() {
  ..
  inFile >> a;
  if (inFile.fail()) {
    cout << "Cannot read a!" << endl;
    return 1;
  }
  inFile >> b;
  if (inFile.fail()) {
    cout << "Cannot read b!" << endl;
    return 1;
  }
  cout << "a = " << a << ", b = " << b << endl;
  return 0;
}
```

" 2 "

```
int main() {
  int a;
  float b;
  ifstream inFile;
  inFile.open ("input.txt");
  if (inFile.fail()){
    cout << "Cannot open file: input.txt" << endl;
    return 1;
  }
  inFile >> a;
  if (inFile.fail()) {
    cout << "Cannot read a!" << endl;
    return 1;
  }
  inFile >> b;
  if (inFile.fail()) {
    cout << "Cannot read b!" << endl;
    return 1;
  }
  cout << "a = " << a << ", b = " << b << endl;
  return 0;
}
```

" 2 3.4" works

"2.3" a=2 b=0.3 tries to find int (2) then float (3)

"2..3" fail

- **cin.eof()**
- A function that returns true if End-of-File is reached
- Used to read a file to the end
- Example: output a file content to the screen

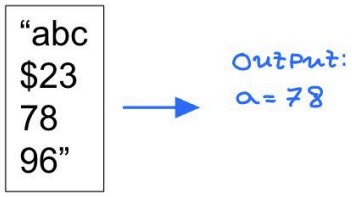
```
ifstream inFile("input.txt");
char c;
inFile >> c;
while (!inFile.eof()) {
  cout << c;
  inFile >> c;
}
```

Handling input failures - cin.clear(), cin.ignore()

- cin.clear(): Clear the error state flag, Resets the stream state to good . The cin fails then the input buffer is kept in an error state.
- cin.ignore(int num_of_chars, char delimiter): Discard num_of_chars characters, up to first delimiter, which ever comes first (Example: inFile.ignore(100, '\n');)

Example

```
int main () {
  int a;
  ifstream inFile;
  inFile.open ("input.txt");
  if (inFile.fail()){
    return 1;
  }
  while (1) {
    inFile >> a;
    if (inFile.fail()) {
      cout << "failed.." << endl;
      inFile.clear();
      inFile.ignore(100, '\n');
      continue;
    }
    cout << "a = " << a << endl;
    break;
  }
  return 0;
}
```



What if inFile.clear() is removed?

fail is not cleared. a = 78 → 96 bats ignored inFile is empty ... infinite loop

Any problem with this example?

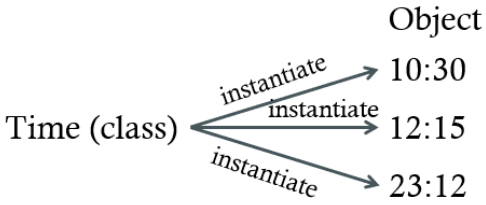
stringstream: treating string as streams

- Using "<<" and ">>" on strings
- ss << foo << ' ' << bar << endl;

7 and 8. Classes and Constructors

Class and Object

- Objects are instantiations of classes
- Each member has concrete value (E.g., Student is a "class", whereas each of you are an "object")




```
// time.cc
  1 #include "Time.h"
  2
  3 void Time::resetTime()
  4 {
  5   minute = second = 0;
  6 }
  7
  8 void Time::setSeconds(int target)
  9 {
 10   if (target > 60) {
 11     second = 60;
 12   } else if (target < 0) {
 13     second = 0;
 14   } else {
 15     second = target;
 16   }
 17 }
```

```
..
29
30 int Time::getSeconds()
31 {
32   return second;
33 }
34
35 int Time::getMinutes()
36 {
37   return minute;
38 }
```

- functions are defined by using class name followed by "::" – no space!
- Can access member data/variable in member functions

```
int main ()
{
  Time t;
  Time *p;
  t.resetTime();
  t.setSeconds(30);
  t.setMinutes(10);
  cout << t.getMinutes() << ":" << t.getSeconds() << endl;

  p = new Time;
  p->resetTime();
  p->setSeconds(25);
  p->setMinutes(15);
  p->printTime();
  delete p;
}
```

What if we call printTime before resetTime()?

Constructors

- Special member functions whose job is to initialize data in object
- Gets called automatically when object is instantiated
- Never called directly by any function

```
// Time.h
class Time {
private:
  int minute;
  int second;
public:
  Time();
  .. ..
};
// time.cc
Time::Time()
{
  second = minute = 0;
  cout << "Time initialized" << endl;
}
```

- Has the same name as the class
- No return value – not even void
- Must be public
- Invoked when an object is *instantiated*
  - Which line will cause the constructor to be invoked?

```
Time t;
Time *p;
p = new Time;
```

Multiple Constructors

- Initialize objects in different ways - Multiple constructors!
- Function overloading
  - Functions w/ same name
  - Must have different param. List
  - Different number of params, or
  - The same number of params but different types
  - Not just different variable names

```
class Time {
private:
  int minute;
  int second;
public:
  Time(); // Default constructor
  Time(int, int);// Overload func.
};
Time::Time()
{
  second = minute = 0;
}
Time::Time(int _m, int _s)
{
  minute = _m;
  second = _s;
}
```

```
int main ()
{
  Time t1(10,0);
  Time t2;
  Time *p = new Time(12,30);
}
```

Linked List Using Objects

```
#ifndef NODE_H
#define NODE_H
#include <stddef.h>
class Node
{
    private:
        int data;
        Node *next;
    public:
        Node();
        Node(int);
        Node(int, Node*);
        int getData() const;
        Node* getNext() const;
};
#endif
```

Node.h

```
#include "node.h"

Node::Node()
{
    data = 0;
    next = NULL;
}

Node::Node(int _data)
{
    data = _data;
    next = NULL;
}

Node::Node(int _data, Node* _next)
{
    data = _data;
    next = _next;
}

int Node::getData() const
{
    return data;
}

Node* Node::getNext() const
{
    return next;
}
```

Node.cc

```h
#ifndef LINKED_LIST_H
#define LINKED_LIST_H
#include "node.h"
class List
{
    private:
        Node *head;
    public:
        List(); // constructor
        List(int);
        /* Always insert at the beginning. */
        void insert(int); // no need of the param head!!!
        /* Return a pointer of node if found, NULL otherwise.*/
        Node* search(int) const; // no need of the head param!!!
};
#endif
```

Linked_list.h

```cpp
#include "linked_list.h"

List::List()
{
    head = NULL;
}

List::List(int _data)
{
    head = new Node(_data);
}

void List::insert(int _data)
{
    Node *nv = new Node (_data, head);
    head = nv;
}

Node* List::search(int _data) const
{
    for (Node *cur = head; cur != NULL; cur = cur->getNext())
    {
        if (cur->getData() == _data)
        {
            return cur;
        }
    }
    return NULL;
}
```

Linked_list.cc

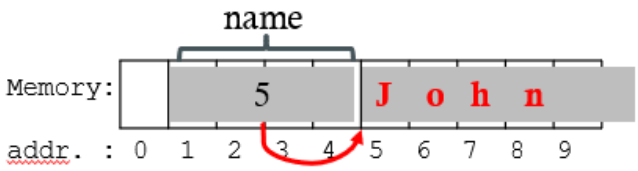
9. Destructors

Destructors

- Complement of constructor
- Automatically called when an object is destroyed
- Not required to have a destructor (more on this later)

```
class B {
private:
  char *name;
public:
  B();
  B(char*);
  ..
};
B::B (char *_name) {
  name = new char[100];
  strcpy(name, _name);
}
B::B() {
  name = NULL;
}
```

- Problem?
  - If B is instantiated with: B b("John"); then a char array is dynamically allocated. When b is then destroyed, the allocated array remains on the heap, with nothing referencing it!
    - Memory leak!



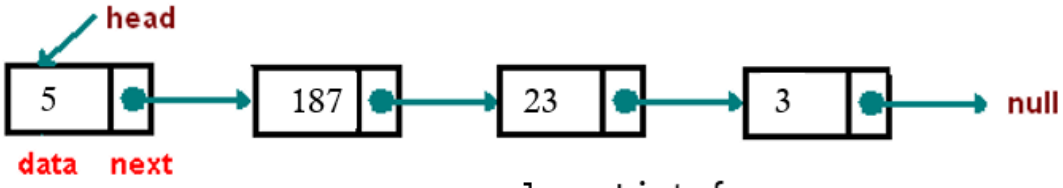

```
class B {
private:
  char *name;
public:
  B();
  B(char*);
  ~B();
};
B::B (char *_name) {
  name = new char[100];
  strcpy(name, _name);
}
B::~B() {
  if (name != NULL)
    delete [] name;
}
```

- Destructor: `~' followed by class name
- Cannot take any parameter!
- Cannot return any type!
- Must be public
- If you used a "new" (or malloc) anywhere in your class, you likely need a destructor with a "delete"!

- [] implies you are deallocating an array (as shown, created with new type[])
- When allocated using new, must free with delete without [] and vice versa

```
// Time.h
class Time {
private:
  int minute;
  int second;
public:
  Time();
  .. ..
};
// time.cc
Time::Time()
{
  second = minute = 0;
  cout << "Time initialized" << endl;
}
```

- Do we need destructor?
  - No! B/c you did not allocate memory dynamically!




```
class Node {
  private:
    int data;
    Node *next;
  public:
    Node(int d, Node *n) {
      data = d; next = n;
    }
    ~Node() {
      if (next != NULL)
       delete next;
    }
}
```

```
class List {
  private:
    Node *head;
  public:
    void insert (int d) {
      head=new Node (d, head);
    }
    ~List(){
      if (head!=NULL) delete head;
    }
}
// main
List *p = new List;
p->insert(3); p->insert(23);..
delete p;
```

10. Pointers and Structures

Accessors and Mutators

```
class Time {
private:
  int minute;
  int second;
public:
  void resetTime();
  void setSeconds(int);
  void setMinutes(int);
  int getSeconds();
  int getMinutes();
  void printTime();
};
```

Mutators

Accessors

Pointers to pointers

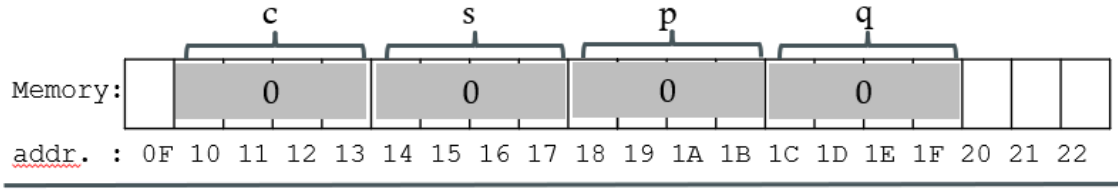
- Same as “address of the pointer”

```
int c = 0, s = 0;
int *p = NULL;
int **q = NULL;
```

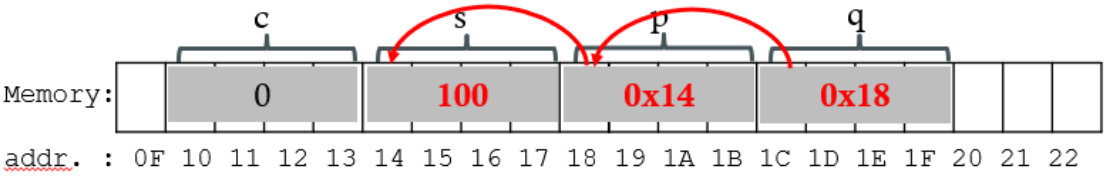
```
q = &p;
p = &s;
**q = 100;
```

```
char c = **q;
   // type mismatch
```

Memory content *before* executing these three lines of code



Memory content *after* executing these three lines of code

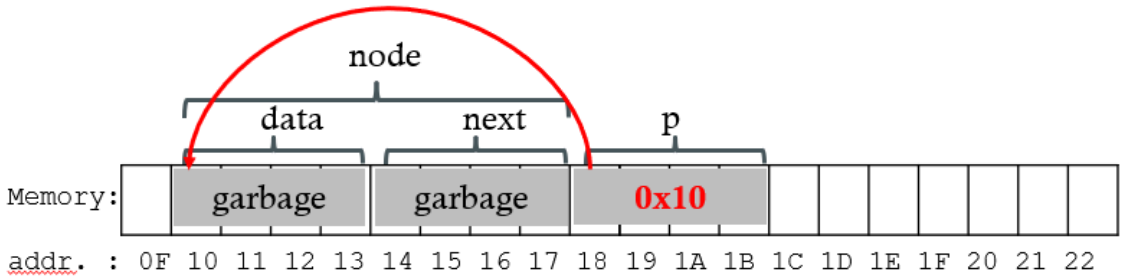


Pointers in Structures

```
struct Node {
  int data;
  struct Node *next; /* a pointer to "struct Node" */
};
```

Pointers to structures are declared in the same way as any other pointer

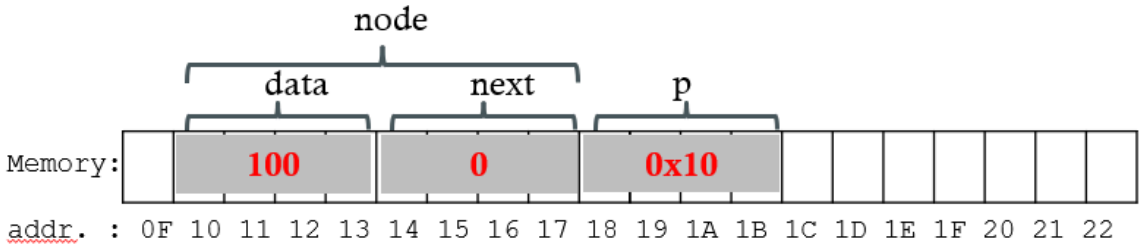
```
struct Node node;
struct Node *p;
p = &node;
```




```
struct Node {
  int data;
  struct Node *next; /* a pointer to "struct Node" */
};
```

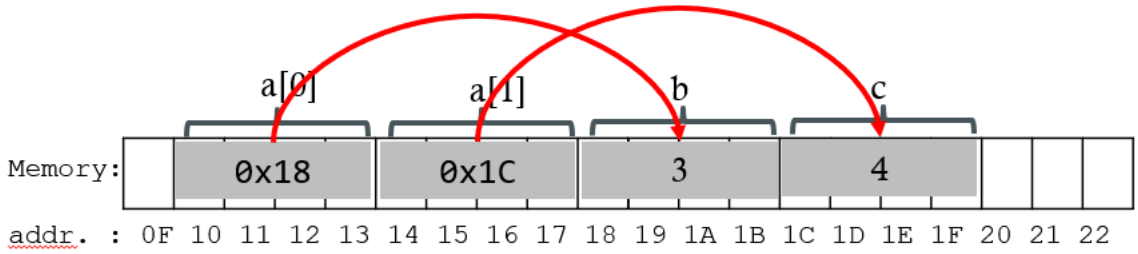
Instead of using a '.', use '->' to find the destination of a structure **pointer**

```
p->data = 100;
p->next = NULL;
```



11. Dynamic Memory Allocation and Arrays

```
int *a[2];
int b = 3, c = 4;
a[0] = &b;
a[1] = &c;
// What is the value of a[0]? *(a[0])?
// What happens when: a[0] = a[1];? *(a[1]) = *(a[0]);?
```



a[0] = 0x1C

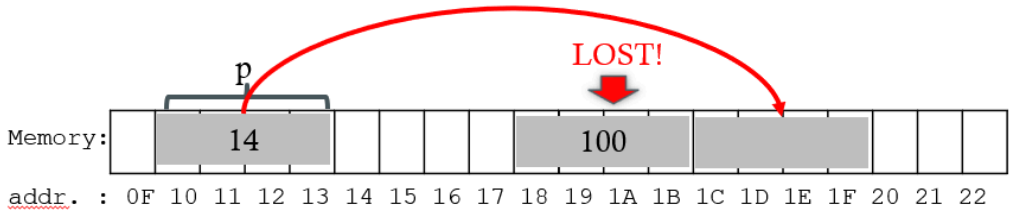
Dynamic Memory Allocation

- Create and destroy memory block dynamically
- Operators: new and delete

Memory Leak

- "new" memory regions have no name associated with them
- Must ensure that the reference is not lost! – Mem. leak

```
int *p = new int;
*p = 100;
p = new int;
```



Using delete to prevent memory leak

```
int *p = new int;
*p = 100;
..
delete p; // when the int is no longer needed
p = NULL; // delete returns the mem. region to the system
..        <-- cout << *p;
p = new int;
```

- But what if we try to dereference 'p' between delete and new?
- Undefined behavior; very hard to debug
- Always set pointer to NULL after delete!

Errors to avoid

```
int x;
int *p;
p = &x;
delete p; // deleting variable; cause crash
```

```
int *p, *q;
int **pp;

pp = p; // type mismatch; compile time error!
pp = *p; // type mismatch (assume p is properly initialized)
p = pp;  // type mismatch
*pp = p; // error b/c pp is not initialized
**pp = *p; // error b/c neither pp or p is initialized
&p = pp; // Cannot assign to an address
q = p; p = new int; *p = 10; cout << *q; // *q is undefined
```

```
int *p;
int y;
y = *p; // dereference uninitialized pointer
```

```
int *p = new int;
p = &x; // memory leak; allocated pointer is lost
```

```
int *p = new int;
delete p;
..
delete p; // multiple delete; likely will crash
/* Note: if you always assign NULL to p immediately after
   delete, it's safe: delete NULL is treated as noop*/
```

12. Variable Scope

Four ways to hold data

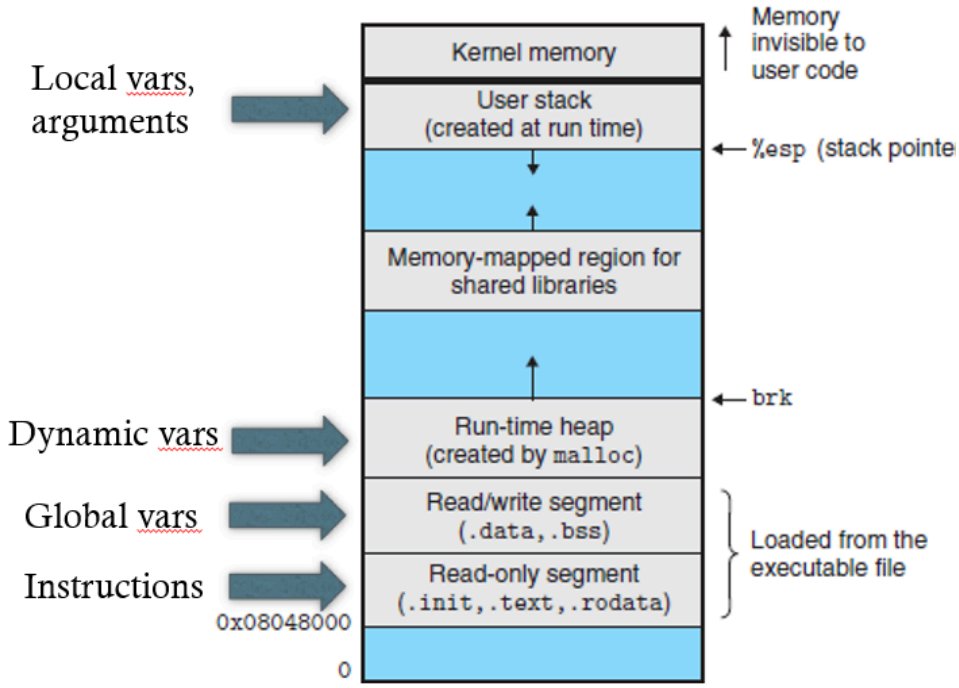
- Global Variable: Declared outside of any function
- Arguments: Passed in on function calls
- Local Variables: Declared inside of function or code block (e.g. for loop)
- Dynamic Variable: Allocated with new and have no name (need to dereference pointer)

	Created	**Destroyed**
Global variable	start of program	End of program
Argument	Start function invocation	End of function execution (unless it is a reference - then it would be treated as a local variable in main)
Local variable	Start of code block {..}	End of code block
Dynamic variable	with new	with delete

```
int glob; // global variable
int foo(int x, int y) // x and y are arguments
{
    int a = 5; // a is local variable
    .. ..
}

int main()
{
    int a = 3; // a is local variable; different from foo's a
    int b = 4;
    ..
    foo(a,b);
}
```

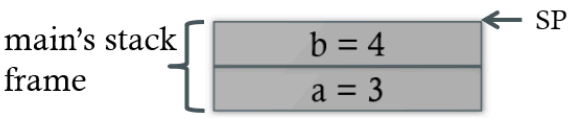
Memory Layout

- Register is the storage in CPU
- Physical memory is typically less than 2^64 bytes (theoretically it should be, but in reality it depends on each system)



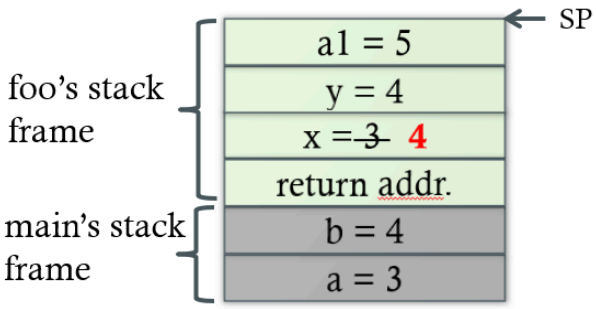
Call Stack

```
int foo(int x, int y)
{
  int a1 = 5;
  x++;
  .. ..
}

int main()
{
  int a = 3;
  int b = 4;
  ..
  foo(a,b);
}
```



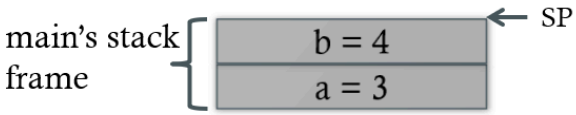
before foo

```
int foo(int x, int y)
{
  int a1 = 5;
  x++;
  .. ..
}

int main()
{
  int a = 3;
  int b = 4;
  ..
  foo(a,b);
}
```



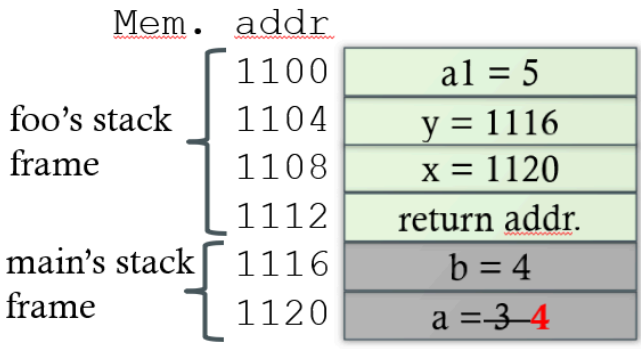
after foo

```
int foo(int x, int y)
{
  int a1 = 5;
  x++;
  .. ..
}

int main()
{
  int a = 3;
  int b = 4;
  ..
  foo(a,b);
}
```



after returned

```
int foo(int *x, int *y)
{
  int a1 = 5;
  *x++;
  .. ..
}

int main()
{
  int a = 3;
  int b = 4;
  ..
  foo(&a,&b);
}
```



pass pointer as arg

13. Operator Overload

Why Operator Overloading?

- Manipulate Complex numbers in C++ like other numbers?

```
class Complex
{
private:
  double real;
  double imag;
public:
  Complex() { real = imag = 0.0; }
  Complex(double _real, double _imag)
  { real = _real; imag = _imag; }
  double getReal() { return real; }
  double getImag() { return imag; }
};
```

Understanding Operators

- Operators are essentially function calls

Expression	Think of it as
x + y	+ (x, y)
x - y	- (x, y)
x * y	* (x, y)

- We can overload or redefine them

An Example of Operator Overloading

```
class Complex
{
private:
  double real;
  double imag;
public:
  Complex() { real = imag = 0.0; }
  Complex(double _real, double _imag)
  { real = _real; imag = _imag; }
  double getReal() { return real; }
  double getImag() { return imag; }
};
```

- How many objects of Complex are created?
- Can I do this: "d = a+b+c"?

```
Complex operator+ (Complex x, Complex y)
{
  return Complex(x.getReal() + y.getReal(),
                 x.getImag() + y.getImag());
}

  Complex a(3.2, 1.0);
  Complex b(4.0, 0.5);
  Complex c;
  c = a + b;
```

- Number of Complex objects (4 to 7) depend on the Complier (in theory, 7 objects but compilers optimize)
- You can do d = a + b + c;

Call by reference

- Recall: *reference* is a safer, but less powerful, pointer
- But you use a reference the same way as you're using the object variable!
  - No need to dereference it!

```
Complex operator+ (Complex &x, Complex &y)
{
  return Complex(x.getReal() + y.getReal(),
                 x.getImag() + y.getImag());
}

  Complex a(3.2, 1.0);
  Complex b(4.0, 0.5);
  Complex c;
  c = a + b;
```

How many objects are created?

- In this case, only 4 - 5 objects are created (depends on compiler).

Overloading Operator as member function

```
class Complex
{
 .. ..
public:
  Complex operator+ (const Complex&) const;
  /* Member func; only one param. */
};

Complex Complex::operator+ (const Complex &x) const
{
  return Complex(real+x.real, imag+x.imag);
}

  Complex a(3.2, 1.0);
  Complex b(4.0, 0.5);
  Complex c;
  c = a + b;
```

const

Tells compiler that the func. does not modify any member variables (only applicable to class member functions)

```
type Class::func (const type p) const;
```

Tells compiler that the param cannot be modified!

Assignment Operator =

- A default version of assignment operator = is created by the compiler for each class
  - Remember c = a + b in the Complex example?
- But you can overload it
  - Must be a member function

```
void Complex::operator= (const Complex &x)
{
  real = x.getReal();
  imag = x.getImag();
}
```

- What's the problem of returning void?

```
c = b = a;
```

  - Equivalent to c = (b = a);
  - Further equivalent to:

```
c.operator=(b.operator=(a));
```

```
/* Correct return type */
Complex& Complex::operator= (const Complex &x)
{
  real = x.getReal();
  imag = x.getImag();
  return *this;
}
```

Private member issue

- In C++, access control is per-class instead of per-object
  - Makes compiler's job easier
  - Easier to overload assignment operator

```
/* This is legal in C++. */
class User {
private:
    int data;
public:
    User& operator= (User& x) {
        data = x.data; // Legal!
        return *this;
    }
}
```

15. Object passing and Copying

Passing Objects to a Function

- Recall that in this example, two objects are created for x and y, but the default constructor was not invoked
- Copy constructor was invoked!

```
Complex operator+ (Complex x, Complex y)
{
  return Complex(x.getReal() + y.getReal(),
                 x.getImag() + y.getImag());
}

  Complex a(3.2, 1.0);
  Complex b(4.0, 0.5);
  Complex c;
  c = a + b;
```

Copy Constructor

- A special constructor used to make a copy of an existing object instance
- Compiler provides a default copy constructor for every class
- Prototype: MyClass (const MyClass& other);
- Must pass the object by reference!
- Should use “const” for the parameter

When is the Copy Constructor called?

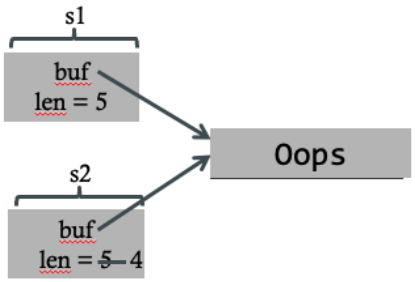
- When instantiating one object from another of the same type
    - MyClass a(b); // b is an object of MyClass
- When an object is passed into a function by value
- When an object is returned from a function by value

The Default Copy Constructor

- Performs memberwise copy
    - i.e. “shallow copy”
- But it can be limited when class contains pointers
    - i.e. when a “deep” copy is needed

```
class MyString {
private:
  char *buf;
  int len;
public:
  MyString() { buf = NULL; len = 0; }
  MyString(char *src) {
    buf = new char[100];
    strcpy(buf, src);
    len = strlen(src);
  }
  void setString(char *src) {
    if (buf != NULL) {
      strcpy(buf,src);
      len = strlen(buf);
    } else ..
  }
  MyString(const MyString & x) {
    /* Shallow copy */
    buf = x.buf;
    len = x.len;
  }
};
```

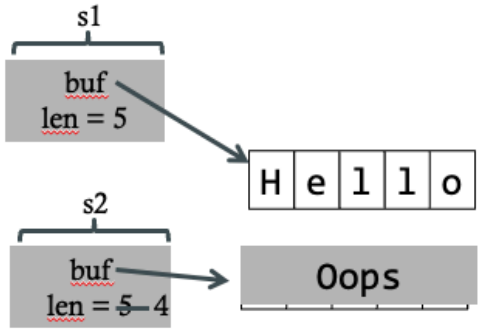
```
MyString s1(“Hello”);
MyString s2(s1);
s2.setString(“Oops”);
s1.printString();
```



Deep Copy

```
MyString::MyString(const MyString & x) {
  /* Deep copy */
  buf = new char[100];
  strcpy (buf, x.buf);
  len = x.len;
}
```

```
MyString s1(“Hello”);
MyString s2(s1);
s2.setString(“Oops”);
s1.printString();
```



Rule of Three

- If a class requires one of the following three:
  - User-defined destructor
  - User-defined copy constructor
  - User-defined assignment operator
- it almost certainly requires all three

Passing Object into function by value

- Copy constructor is used to create a copy of x and y

```
Complex operator+ (Complex x, Complex y)
{
  return Complex(x.getReal() + y.getReal(),
                 x.getImag() + y.getImag());
}

  Complex a(3.2, 1.0);
  Complex b(4.0, 0.5);
  Complex c;
  c = a + b;
```

Return an Object by Value

- When an object is returned by value, another copy of the object could be created via copy constructor
  - So this example could create 7 objects
  - But in g++, by default, this copy is omitted as an optimization
  - You can disable it with the option: -fno-elide-constructors

Why Const?

```
MyClass ( const MyClass& other );
```

- C++ standard says that non-const references cannot bind to temporary objects, i.e., objects w/o a name

```
Complex operator+ (Complex x, Complex y)
{
  return Complex(x.getReal() + y.getReal(),
                 x.getImag() + y.getImag());
}
```

temporary object!

```
  Complex a(3.2, 1.0);
  Complex b(4.0, 0.5);
  Complex c;
  c = a + b;
```

Lastly..

- Copy constructor is invoked when you declare + initialize an object at the same time
  - e.g.: Complex b = a; // copy constructor
  - But not invoked when you: Complex b; b = a;
- If some data members are pointers to other data, then you should provide your own default constructor (to allocate the data), your own copy constructor (to make a deep copy), your own assignment operator (to make a deep assignment), and a destructor (to explicitly delete dynamically allocated data).

16. Linked List

Node

```
class Node {
  private:
    int data;
    Node *next;
  public:
    Node() { data = 0; next = NULL; }
    Node(int d) { data = d; next = NULL; }
    Node(int d, Node *n) { data = d; next = n; }
    int getData() const { return data; }
    void setData (int _d) { data = _d; }
    Node *getNext() const { return next; }
    void setNext(Node *_n) { next = _n; }
}
```

Defining the List

- The List class needs to keep the list sorted
- keep track of “head” of the list
- Support basic operations: Inserting (keep list sorted), Deleting, Searching, Copying one list from another list
- Have a destructor that properly deletes the list

List Declaration

```
class List {
  private:
    Node *head;
  public:
    List();
    ~List();
    void insertValue (int);
    bool valueExists (int);
    bool deleteValue (int);
    List (const List & original);
    List & operator= (const List & original);
    .. ..
};
```

Searching the List

```
bool List::valueExists(int _data)
{
  for (Node *cur = head; cur != NULL; cur = cur->getNext())
  {
    if (cur->getData() == _data)
    {
      return true;
    }
    if (cur->getData() > _data)
    {
      return false;
    }
  }
  return false;
}
```

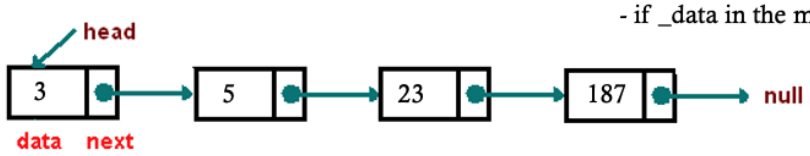
Inserting

```
void List::insertValue (int _data)
{
  Node *n = new Node(_data);
  Node *p = head;
  Node *prev = NULL;
  while (p != NULL && p->getData() < _data) {
    prev = p;
    p = p->getNext();
  }
  n->setNext(p);
  if (prev == NULL) // head of the list!
    head = n;
  else
    prev->setNext(n);
}
```

Deleting

```
 1 bool List::deleteValue(int _data) {
 2   Node *p = head, *prev = NULL;
 3   while (p != NULL && p->getData() != _data) {
 4     prev = p;
 5     p = p->getNext();
 6   }
 7   if (p == NULL) // _data is not in the list
 8     return false;
 9   /* Remove the node pointed by p from the list. */
10   if (prev == NULL) // p is the head of the list!
11     head = p->getNext();
12   else
13     prev->setNext(p->getNext());
14   delete p;
15   return true;
16 }
```

Check: does it work
- if list is empty
- if _data not in list
- if _data first node
- if _data last node
- if _data in the middle



Destructor

```
List::~List() {
  Node *p;
  while (head != NULL) {
    p = head;
    head = p->getNext();
    delete p;
  }
}
```

Copy Constructor

```
 1 List::List (const List & original) {
 2   Node *p = original.head;
 3   Node *np = NULL;
 4   head = NULL;
 5   while (p != NULL) {
 6     Node *n = new Node (p->getData());
 7     if (np == NULL) head = n;
 8     else np->setNext(n);
 9     p = p->getNext();
10     np = n;
11    }
12 }
```

Operator =

```
List & List::operator=(const List & original) {
  if (&original == this)
    return (*this);
  if (head != NULL)
  { // empty the list as in destructor
    .. ..
  }
  // copy list as copy constructor
  return (*this);
}
```

17. Recursion

Factorial

```
int factorial (int n) {
  if (n==1) return 1; // Basis
  return (n*factorial(n-1)); // Recursive
}

int main() {
  int f = factorial(4);
}
```

Linked List Recursion

```
void rprint (Node *p) {
  if (p == NULL) return;
  rprint(p->next);
  cout << p->data;
}

rprint(head);
```

```c
// x^y
double power(double x, int y){
    if (x == 0 && y == 0){
        return -1;
    } else if (x == 0){
        return 0;
    }

    if (y == 0){
        return 1.0;
    } else if (y < 0){
        return 1 / power(x, -1 * y);
    } else if (y > 0){
        // 2^2 = 2 * 2
        // 2^4 = 2^2 * 2^2

        // 2^3 = 2^1 * 2^1 * 2
        // 2^5 = 2^2 * 2^2 * 2
        if (y % 2 == 0){
            double temp = power(x, y/2);
            return temp * temp;
        } else{
            double temp = power (x, y/2);
            return temp * temp * x;
        }
    }
}
```

This is in C, from APS105

18. Quicksort

Idea

- Pick a pivot value from an array
- Move every value < pivot to the left of the pivot
- Move every value >= pivot to the right of the pivot
- Quicksort sub-array to the left of pivot
- Quicksort sub-array to the right of pivot

```
25 void qsort (int *a, int begin, int end)
26 {
27   int pivotIndex = selectAndShuffle(a, begin, end);
28   if (pivotIndex - 1 > begin)
29     qsort(a, begin, pivotIndex - 1);
30   if (pivotIndex + 1 < end)
31     qsort(a, pivotIndex + 1, end);
32   return;
33 }
```

```
11 int selectAndShuffle (int *a, int begin, int end) {
12   int pivotValue = a[begin];
13   int current = begin;
14   for (int i = begin+1; i <= end; i++)
15     if (a[i] < pivotValue) {
16       current++;
17       swap(a[current], a[i]);
18     }
19   swap(a[begin], a[current]);
20   return current;
21 }
```

SelectAndShuffle

Pick the first element as pivot

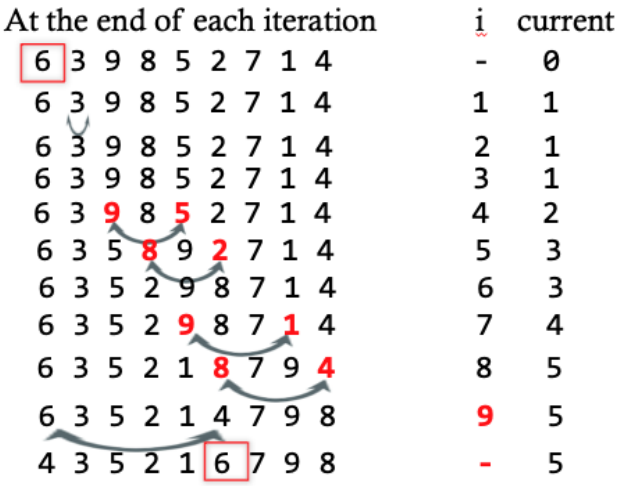
Move elements > pivot to its right; elements < pivot to its left

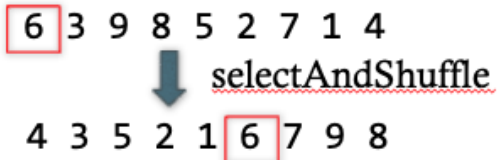
Return the new index of pivot

Iterate through the array, find every element < pivot

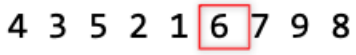
Put the i-th element that is < pivot to the a[i]; how? Swap!

Finally, swap the m-th element that is < pivot with pivot





- After selectAndShuffl(a, 0, 8) returns 5



  - qsort(a, 0, 4)

  - qsort(a, 6, 8)

Things to Consider

- How to choose a pivot point? First? Last?
- Commonly used is median of three: median (a[begin], a[middle], a[end])
- I guess find the median and swap with the zeroth element, apply the same procedure

19. BSTree

- **Tree:** A data structure made up of nodes and edges. Each node can have multiple children, but only one parent. No Cycle! One Root.
- **Binary tree:** each node has at most 2 children
- **Binary search tree:** a binary tree s.t. for every node, the nodes in the left subtree have values <= value of node the nodes in the right subtree have values > value of node

Tree Traversal

- Visit every element of a tree - Three methods:
- Pre-order: node, left subtree, right subtree
  - 8, 3, 1, 6, 4, 7, 10, 14, 13
- Post-order: left subtree, right subtree, node
  - 1, 4, 7, 6, 3, 13, 14, 10, 8
- In-order: left subtree, node, right subtree
  - 1, 3, 4, 6, 7, 8, 10, 13, 14

```
 7 class Tree {
 8   private:
 9     TreeNode * root;
10   public:
11     Tree();
12     void insert (int v);
13     void print_inorder();
14     bool valueExists(int v);
15     void del (int v);
16 };
```

```
 7 class TreeNode {
 8   private:
 9     int value;
10     TreeNode *left;
11     TreeNode *right;
12   public:
13     TreeNode();
14     TreeNode( int v );
15     void insert (int v);
16     void print_inorder();
17     int minimum();
18     bool valueExists(int);
19     void del(int v, TreeNode *& pp);
20 };
```

```
 6 Tree::Tree() {
 7   root = NULL;
 8 }
 9
10 void Tree::insert(int v) {
11   if (root == NULL)
12     root = new TreeNode(v);
13   else
14     root->insert(v);
15   return;
16 }
```

```
18 void Tree::print_inorder() {
19   if (root != NULL)
20     root->print_inorder();
21   cout << endl;
22 }
23
24 bool Tree::valueExists(int v) {
25   return root->valueExists(v);
26 }
27
28 void Tree::del (int v) {
29   if (root == NULL)
30     return;
31   root->del(v, root);
32   return;
33 }
```

```
 3 /* Default constructor */
 4 TreeNode::TreeNode() {
 5   value = -1;
 6   left = right = NULL;
 7 }
 8
 9 TreeNode::TreeNode(int v) {
10   value = v;
11   left = right = NULL;
12 }
```

Insert

```
14 void TreeNode::insert (int v) {
15   if (value == v)
16     return;
17   if (v < value) {
18     // Insert to the left
19     if (left == NULL)
20       left = new TreeNode(v);
21     else
22       left->insert(v); // recursion
23   } else {
24     // Insert to the right
25     if (right == NULL)
26       right = new TreeNode(v);
27     else
28       right->insert(v); // recursion
29   }
30   return;
31 }
```

Traversal in-order

```
33 void TreeNode::print_inorder() {
34   if (left != NULL)
35     left->print_inorder();
36   cout << value << " ";
37   if (right != NULL)
38     right->print_inorder();
39 }
```

Return Minimum

```
41 int TreeNode::minimum() {
42   if (left == NULL)
43     return value;
44   return left->minimum();
45 }
```

Delete value

```
61 void TreeNode::del(int v, TreeNode*& pp) {
62   if (v < value) {
63     if (left != NULL)
64       left->del(v, left);
65     return;
66   }
67   if (v > value) {
68     if (right != NULL)
69       right->del(v, right);
70     return;
71   }
72   // v == value
```

```
73   if ((left == NULL) && (right == NULL)) {
74     pp = NULL;
75     delete this;
76   } else if ( (left == NULL) && (right != NULL) ) {
77     pp = right;
78     right = NULL;
79     delete this;
80   } else if ( (left != NULL) && (right == NULL) ) {
81     pp = left;
82     left = NULL;
83     delete this;
84   } else {
85     int m = right->minimum();
86     value = m;
87     right->del(m, right);
88   }
89   return;
90 }
```

20-21. Inheritance (Intro)

```
class DerivedClass : public BaseClass
{
  /* Only list the inherited members if you wish
  to change them, plus new members you're adding. */
}

Example:
class OrderedList : public List {
  void insert (int v); // redefine insert()
  // value, next, delete(), print()… are all inherited,
  // and you can use them as if they were defined in
  // OrderedList
}
```

- Once derived class is declared, you can use it just like any other class
  - E.g. `void OrderedList::insert(int v) { … }`
  - `OrderedList *list = new OrderedList;`

Derived classes and private members

- Derived classes CANNOT access private members of base class
- But these private members are inherited!
- I.e., in this example, when you instantiate an object of Derived class, there will be memory allocated for "a"

```
class Base {
private:
  int a;
  .. ..
};
class Derived : public Base {
private:
..
};
```

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

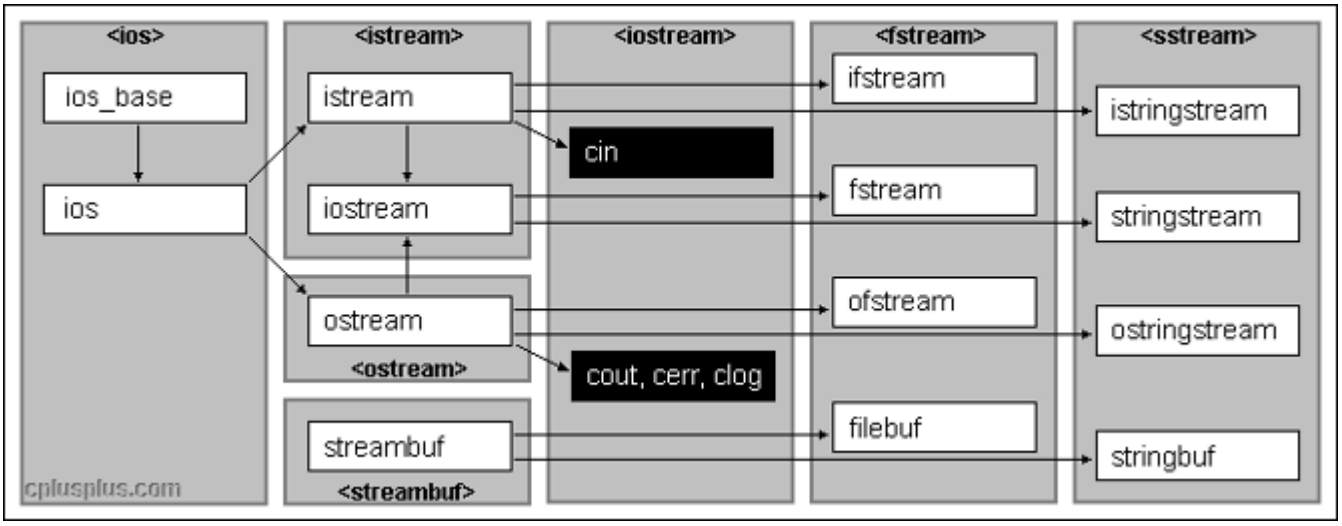
Now you can access "a" as a member variable.

```
class Base {
private:
protected:
  int a;
  .. ..
};
```

Overriding vs Overloading

- Overloading: same function name, different parameter list
  - void foo (int v);
  - void foo (double d);
  - void foo (int i, int j);
- Overriding: redefine a function in derived class
  - List::insert (int a);
  - OrderedList::insert (int a);
- You can also redefine a member var, say “a”, with the same name in derived class
- There will be two copies of “a” – one in base obj. and one in derived class object.

- Derived class inherits all member variables from base class ✓
- Derived class can access private members of base class ✗
- Derived class can access protected members of base class ✓
- Derived class can add member vars and functions ✓
- Derived class can remove member vars and functions ✗



Mixing Types?

- A Manager is an Employee, but an Employee is not a Manager.
- Hence:

```
void g (Manager mm, Employee ee)
{
  Employee *pe = &mm; // OK
  Manager *pm = &ee; // Not OK!
  ..
}
```

Inheritance vs Membership

Why not:

```
class Manager
{
private:
  Employee emp;
  ..
}
```

- How does this compare to inheritance?
- "Is a" relationship vs. "has a"

Accessing Parent Class Members

```
 5 class Base {
 6 private:
 7   int a;
 8 public:
 9   void set(int _a) { a = _a; }
10   void print() { cout << "Base.a = " << a << endl; }
11 };
12
13 class Derived : public Base {
14 private:
15   int b;
16 public:
17   void set(int _a, int _b) {
18     Base::set(_a);
19     b = _b;
20   }
21   void print() {
22     Base::print();
23     cout << "Derived.b = " << b << endl;
24   }
25 };
```

```
 5 class Base {
 6 protected:
 7   int a;
 8 };
 9
10 class Derived : public Base {
11 private:
12   int a;
13 public:
14   void set(int _base, int _derived) {
15     Base::a = _base;
16     a = _derived;
17   }
18   void print() {
19     cout << "Base.a = " << Base::a << endl;
20     cout << "Derived.a = " << a << endl;
21   }
22 };
```

```
21 class Son : public Dad {
22 private:
23   int a;
24 public:
25   void set(int _grandpa, int _dad, int _son){
26     Grandpa::a = _grandpa;
27     Dad::a = _dad;
28     Son::a = _son;
29   }
30   void print() {
31     Grandpa::print();
32     Dad::print();
33     cout << "Son.a = " << a;
34   }
35 };
```

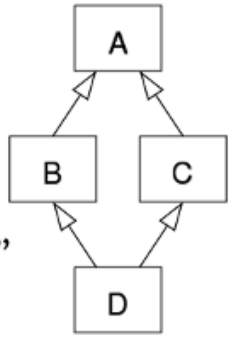
Avoid Multiple Inheritance

```
class Grandpa {
private:
  char* name;
public:
  char* getName () { return name; }
};
class Parent1 : public Grandpa { .. };
class Parent2 : public Grandpa { .. };
class Child : public Parent1, public Parent2 { .. };

int main () {
  Child c;
  c.getName();
  .. ..
}
```

Diamond of Dread

- The code won't compile
- TWO copies of Grandpa's members are created when creating a Child object
- Compiler doesn't know which "getName()" should be invoked



22. Inheritance Constructor & Destructor

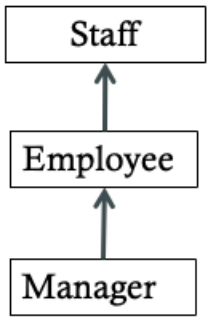
- Constructors are NOT inherited
- E.g., Base::Base(int) won't be invoked by Derived d(3), if Derived::Derived(int) is not defined
- If you don't explicitly call a base constructor, the default base constructor is called when initializing a derived class object
- You can explicitly call (non-default) base constructor using the following syntax:
- Derived::Derived(..) : Base(..) { .. }
- Base constructor always called first!

```
 5 class A {
 6 private:
 7   int a;
 8   int b;
 9 public:
10   A(int _a, int _b) {
11     a = _a;
12     b = _b;
13   }
14   void print() {
15     cout << "A.a = " << a << ", A.b = " << b << endl;
16   }
17 };
```

```
19 class B : public A {
20 private:
21   int a;
22   int b;
23 public:
24   B() : A(1, 1) {
25     a = b = 2;
26   }
27   void print() {
28     A::print();
29     cout << "B.a = " << a << ", B.b = " << b << endl;
30   }
31 };
```

```
class Employee {
private:
  string name;
  int sin;
public:
  Employee() : name (“”), sin(0) { /* deliberately left empty */ }
  Employee(string n, int s) : name(n), sin(s) { /* empty */ }
};
.. ..
Manager::Manager(string n, int sin, int _level)
  : Employee(n, sin), level(_level) { /* empty */ }
```

- In this case, Staff() being invoked first, then Employee(), then Manager()



- When there is a user declared constructor of any type, there is no compiler-generated default constructor

Destructors

- Destructor of the base class is called automatically
- But after derived class’s destructor
- In this case, the order of destructor invocation: ~Manager(), ~Employee(), ~Staff()
  - One way to remember: multiple levels of inheritance works like a stack

Inheritance and Dynamic Memory

- Recall: whenever a class contains member vars that point to dynamic memory, we should manage memory and write our own code for:
    - operator=
    - copy constructor
- This is no different with derived class - Deep copy base class member variables
    1. call base class copy function
    2. deep copy extra member vars contained in derived class

```
Base::Base (const Base & original) {
 // Deep copy original into *this
}

Derived::Derived (const Derived & original) : Base(original)
{
  // Deep copy extra Derived vars
}
```

- You have to call Base(original) yourself
    - If you don't call it yourself, the default constructor of Base will be invoked – Error!!!

Operator =

```
Base & Base::operator= (const Base & rhs) {
 // Deep copy
}

Derived & Derived::operator= (const Derived & rhs)
{
  Base::operator=(rhs);
  // Deep copy any extra Derived variables
  return (*this);
}
```

Revisit the terminology: "F is not inherited"

- Where F = constructors/destructor/operator=/copy constructor
  - Not in Derived class interface, but can still be invoked
  - Examples:

```
class Base {
public:
  Base (int) { .. }
};

class Derived {
/* No Derived(int) */
}

Derived p (10); // Compile error
```

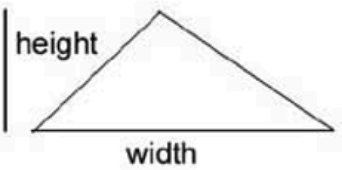
```
class Base {
public:
  Base () { .. }
};

class Derived {
}

Derived p; // Correct, and Base()
is invoked!! – Base() is NOT in
the interface of Derived, but it
will be invoked!
```

23. Inheritance Polymorphism

```
 4 class Polygon {
 5 protected:
 6   int width, height;
 7 public:
 8   void set_values (int a, int b) { width=a; height=b; }
 9 };
10
11 class Rectangle: public Polygon{
12 public:
13   int area() {
       return width*height;
     }
14 };
15
16 class Triangle: public Polygon{
17 public:
18   int area() {
       return width*height/2;
     }
19 };
```




```
21 int main () {
22   Rectangle rect;
23   Triangle trgl;
24   Polygon * ppoly1 = &rect;
25   Polygon * ppoly2 = &trgl;
26   ppoly1->set_values (4,5);
27   ppoly2->set_values (4,5);
28   cout << rect.area() <<'\n';
29   cout << trgl.area() <<'\n';
30   return 0;
31 }
```

But we can't do this: `ppoly1->area()`

- What if you want to be able to invoke "area()" using a pointer to an object of base class

```
22   Rectangle rect;
23   Triangle trgl;
24   Polygon * ppoly1 = &rect;
25   Polygon * ppoly2 = &trgl;
```

  - E.g., ppoly1->area() will invoke rect.area(), and ppoly2->area() will invoke trgl.area()

- More generally, we want use a pointer to the base class object to invoke a function that is implemented in a derived class
  - You cannot do it using what we have learnt so far

Virtual Functions

If a function is declared as virtual in base class A, and redefined in derived class B, then a call made via a pointer, which is declared as A* type but actually points to an obj. of type B, will cause the function defined in B being invoked

```
 4 class Polygon {
 5 protected:
 6   int width, height;
 7 public:
 8   void set_values (int a, int b) { width=a; height=b; }
 9   virtual int area() { return 0; }
10 };
11
12 class Rectangle: public Polygon {
13 public:
14   int area() { return width*height; }
15 };
16
17 class Triangle: public Polygon {
18 public:
19   int area() { return width*height/2; }
20 };
```

```
Rectangle rect;
Triangle trgl;
Polygon * ppoly1 = &rect;
Polygon * ppoly2 = &trgl;
ppoly1->set_values(4,5);
ppoly2->set_values(4,5);
cout<<ppoly1->area(); // 20
cout<<ppoly2->area(); // 10
```

What if we remove the "virtual" keyword from line 9?

- The decision as to which inherited function to call is made at runtime when the function is invoked (rather than at compile time) depending on the type of the target object
- This feature is also called late binding
- The virtual function in Base class is like a "dummy" or "placeholder" declaration of the function
- You cannot declare a member variable as virtual, Only functions
- If we never intended to use "Polygon" to create objects (i.e., we use the base only to derive classes), then we do not fill dummy function definition
- A class that contains at least one pure virtual function is called an abstract class, Also called an interface
- You cannot instantiate an object of an abstract class

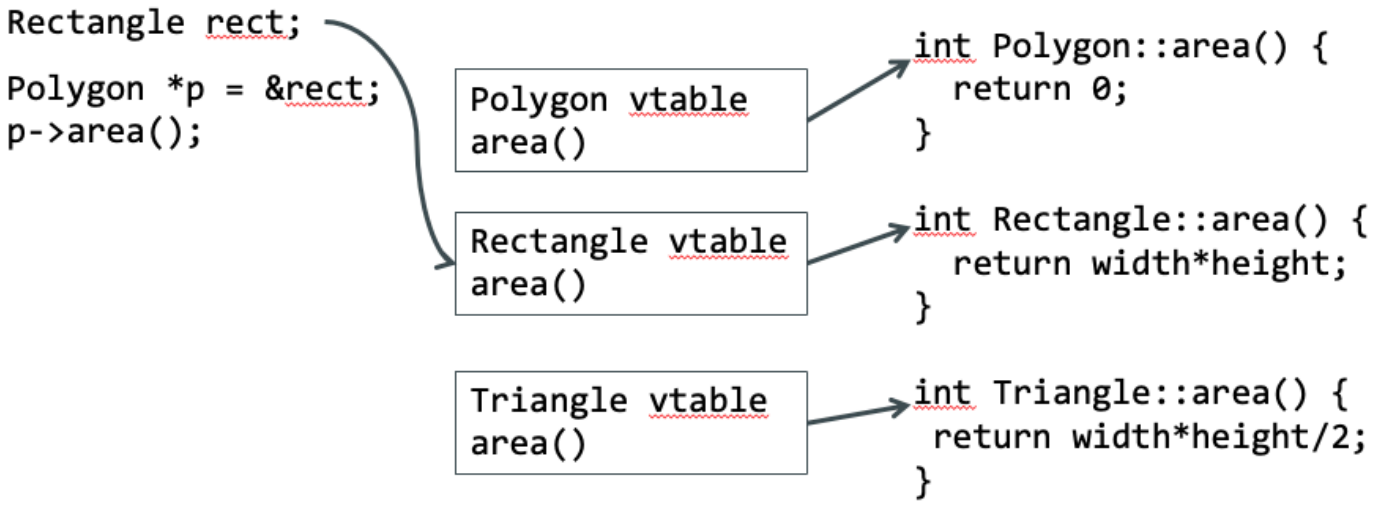
```
 4 class Polygon {
 5 protected:
 6   int width, height;
 7 public:
 8   void set_values (int a, int b) { width=a; height=b; }
 9   virtual int area() = 0; // Pure virtual function
10 };
```

Polymorphism

- A class that declares or inherits a virtual function is called a polymorphic class
- Both Rectangle and Triangle are polymorphic classes of Polygon
- Both TCP_Sender and UDP_Sender are polymorphic classes of Sender

vtable

- For every class, there is a table that maps each virtual function to the most-derived function target
- Each object has a pointer to the vtable of its type



24 - 25. Complexity

a = b; // T(n) = 1

- In all these cases, T(n) is an approximation
  - Actual running time depends on
    - Computer speed
    - Language
    - Compiler

Big O Notation

- When comparing algorithms, we are not interested in constants (b/c a faster computer can make up any difference)
- We are only interested in the highest order term (b/c that dominates)
- We say T(n) is on the order of O(f(n)) if T(n)'s highest order term is f(n), disregarding constants
- Formally: T(n) is in O(f(n)) if

  ∃ constant C, N, so that $T(n) \leq C \times f(n)$ for every $n > N$

- `a = b;` T(n) = 1 -> O(1)

- ```
  sum = 0;
  for (int i = 0; i < n; i++)
    sum += i;
  ```

  T(n) = 2 + 3*n O(n)

  why 3? addition, increment, comparison

- Linear search

  ```
  for (i = 0; i < n; i++)
    if (a[i] == value)
      return i;
  ```

  Best case: T(n) = 4 O(1)

  Worst case: T(n) = 1 + 3*n

  Average case: T(n) = 1 + C*n

  O(n)
  ```

Matrix Multiply

```
double a[N][N];
double b[N][N];
double c[N][N];// initialized to 0

/* Multiply N x N matrices a and b  */
int i, j, k;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      c[i][j] += a[i][k] * b[k][j];
```

- $T(n) \approx C_1 \times n^3 + C_2$
- $O(n^3)$

Facebook Homepage

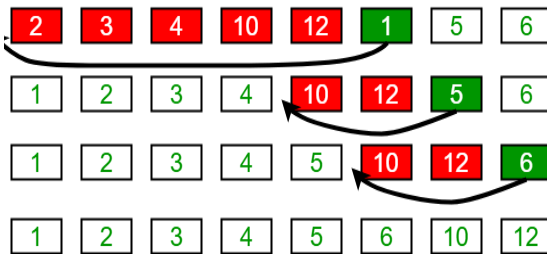
```
foreach (friend in friendList) // assume n friends
   foreach (update of friend)  // assume m recent updates
     display update;
```

$O(n \times m)$

If m is a constant, i.e., 3, then it's O(n)

Insertion Sort

```
for (i = 0; i <n; i++)
  for (j = i; j > 0 && a[j] < a[j-1]; j--)
    swap(a[j], a[j-1]);
```



- Best case: $T(n) \approx C \times n = O(n)$
- Worst case: $T(n) \approx C(n + (n-1) + (n-2) + .. +1) = C*n(n+1)/2 = O(n^2)$
- Avg. case: for each iteration in the outer loop, needs to swap w/ half of the elements in a[0]-a[i]
  - $C*(\frac{1}{2} * n + \frac{1}{2} * (n-1) + \ldots + \frac{1}{2} * 2 + 1) = C * n(n+1)/4 = O(n^2)$

Binary Search on Sorted Array

```
bool search (int first, int last, int key, int & index)
{
  if (first > last) return false;
  int mid = (first + last) /2 ;
  if (key == a[mid]) {
    index = mid;
    return true;
  }
  if (key < a[mid])
    return search (first, mid-1, key, index);
  else
    return search(mid+1, last, key, index);
}
```

Problem size: n, n/2, n/4, .., 1

O(log(N))

search for 2 in: [0 1 [[2] 3]] 4 5 6 7 8

T(n) = C + T(n/2) = C + (C + T(n/4))..

Speed?

n	log(n)	n*log(n)	n^2	n^3	2^n (exponential)
2	1us	2us	4us	8us	4us
16	4us	64us	256us	4ms	65ms
256	8us	2ms	65ms	16s	$4*10^{63}$ years
1024	10us	10ms	1s	17min.	$6*10^{294}$ years

Recursion

```
int fact(int n) {
  if (n == 1) return 1;
  return (n * fact (n - 1));
}
```

```
T(n) = 2 + T(n-1)
     = 2 + 2 + T(n-2)
     = 2 x 3 + T(n-3)
     = 2 x (n-1) + T(n – (n-1)) = 2 x (n-1) + T(1) = 2n – 2
```

Complexity: O(n)

Quicksort

```
25 void qsort (int *a, int begin, int end)
26 {
27   int pivotIndex = selectAndShuffle(a, begin, end);
28   if (pivotIndex - 1 > begin)
29     qsort(a, begin, pivotIndex - 1);
30   if (pivotIndex + 1 < end)
31     qsort(a, pivotIndex + 1, end);
32   return;
33 }
```

$T(n) = T_{selectAndShuffle}(n) + T(k) + T(n - k - 1)$

```
11 int selectAndShuffle (int *a, int begin, int end) {
12   int pivotValue = a[begin];
13   int current = begin;
14   for (int i = begin+1; i <= end; i++)
15     if (a[i] < pivotValue) {
16       current++;
17       swap(a[current], a[i]);
18     }
19   swap(a[begin], a[current]);
20   return current;
21 }
```

$T(n) \approx C \times n$

Quicksort - Worst Case

$$T(n) \approx T_{selectAndShuffle}(n) + T(k) + T(n - k - 1)$$
$$\approx C \times n + T(k) + T(n-k-1)$$

Worst case: k = 0, i.e., pivotIndex == begin

$$T(n) \approx C \times n + T(n-1)$$
$$\approx C \times (n + n-1) + T(n-2)$$
$$..$$
$$\approx C \times (n + n-1 + n-2 + \dots 1) = C \times n \times (n+1)/2 = O(n^2)$$

Quicksort - Best/Avg Case

k = n/2, i.e., evenly split the array

$T(n) \approx C \times n + 2 \times T(n/2)$

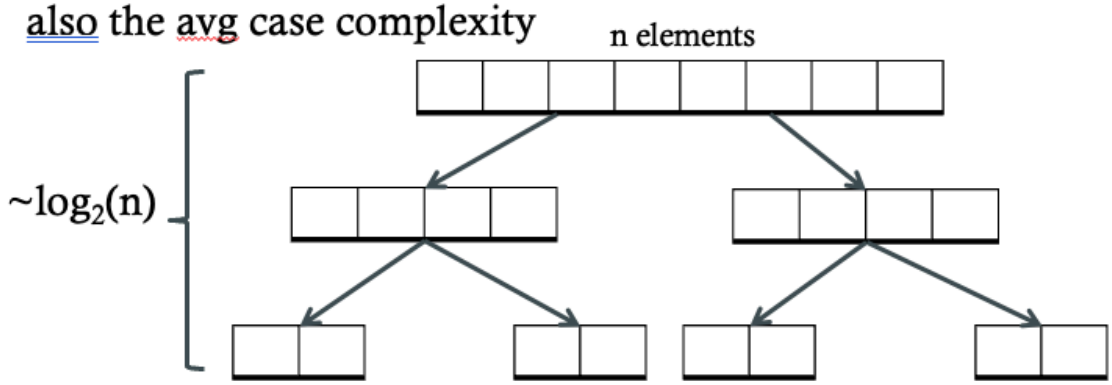
$\approx C \times (n + n) + 4T(n/4))$

$\approx C \times (n + n + n) + 8T(n/8)$

$\approx O(n \times \log(n))$

log(n)

also the avg case complexity

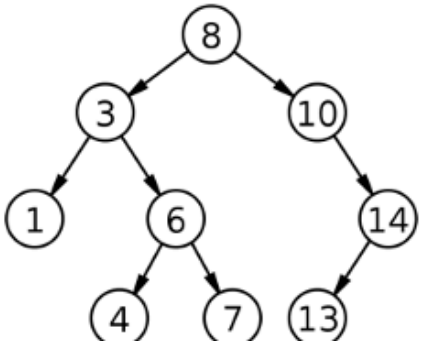


Search on BSTree

- If the tree is balanced, e.g., every non-leaf node has two children
  - $T(n) = C + T(n/2) = 2C + T(n/4) = .. = C * \log_2(n)$

```
bool TreeNode::valueExists(int v) {
  if (v == value)
    return true;
  if (v < value) {
    if (left != NULL)
      return left->valueExists(v);
    return false;
  }
  if (right != NULL)
    return right->valueExists(v);
  return false;
}
```

O(log(N))

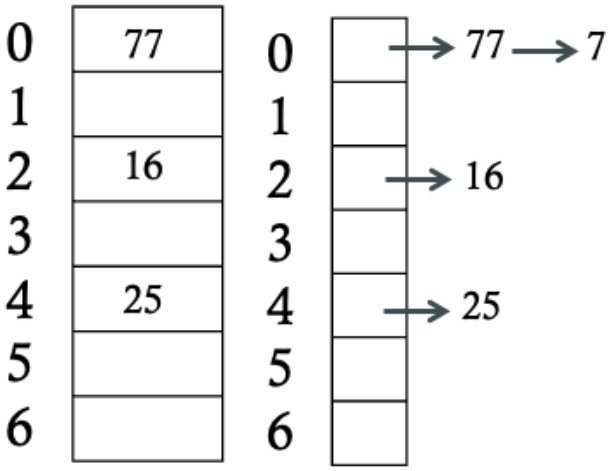


26. Hash Tables

	Unsorted Array	Sorted Array	Unsorted List	Sorted List	BST (min level tree)
Insert	O(1)	O(n)	O(1)	O(n)	O(log n)
Search	O(n)	O(log n)	O(n)	O(n)	O(log n)

- Provides O(1) average performance on search/insert/deletion
- Basic idea:
  - Stores data in a very large array. Typically 2x as many elements as the # of data elements to be stored
  - Each key (e.g., userID) maps to a unique index
  - User "hash function", h(k) to map each element into an array index
    - h(userID) -> [0..m]
    - Example: h(k) = k % (m+1)

- h(key) = key % 7
- insert(16): h(16) = 2
- insert(25): h(25) = 4
- insert(77): h(77) = 0
- search(16): h(16) = 2
- search(33): h(33) = 5
- search(32): h(32) = 4
- insert(7): h(7) = 0??



Collision

- Problem: two keys may map to the same array entry
- Solution: hashing with chaining
- Each hash table entry contains a pointer to a linked list of keys that map to the same entry
- Therefore the worst-case complexity for search is O(n)!!!

Good hash function

- A good hash function should avoid collision
- So that the average length of each list is 1
- How?
    - Use more spaces
        - E.g., > 2x array elements
    - Use smarter algorithms
        - Real-world hashing algorithms usually involve multiply a large prime number
        - E.g., h(k) = k * 31 % m