* #include <iostream>
* #include <vector>
* using namespace std;
* int main ()
* {
* vector <int > x;
* x.push\_back(3);
* x.push\_back(5); // increase the array size by one and put 5 in the end.
* cout << x[1];

}

some vector examples.

String comparison compares each characters in the string on the ASCII code.

#include “person.h”

if use it twice, person only get declared once.

Use code to fix whether a file is included twice.

Using namespace std;

Use iostream for cin cout

Const int START.LEN=10;

List::List()

.size(0), lenOfArray(START.LEN) // initializer list

{

elements=new string[START.LEN]

}

bool List:add(string s)

{

if ()size>=lenOfArray)

expandList();

element[size]=s;

size++;

return ture.

}

void List::expandList()

{

const int factor=2;

string \* nl=new string [FACTOR\* lenOfArray] ///double the previous lensOfArray;

for (int i=0l i<size; i++)

{

nl[i]=elements[i]

delete [] elements // use [] because its in element.

Elements =nl;

lenOfArray\*=FACTOR;

}

List ::~List()

{

delete [] elements;

}

* at compile time , the program figure out how much memory the program needs.
* If use new, it use a different memory location.
* List::List ()

:first (nullptr);

* {
* }
* struct Element
* {

string data;

Element \*next;

* }
* //change so that front is a pointer.
* bool List:add(string s)
* {
* if (front ==nullptr)
* {
* front =new Element;
* front ->data=s
* front -> next=nullptr;
* return true;
* }
* }
* if times two, the compiler would just put an zero at the end. (binary)
* for a multifile system such as the book, .h we put the class information in. We call it an interface and for .cpp file, we call it implementation.
* Class Rational {

Rational (int numer=0, denom=1)

Rational r=3; then it assigns 3 to numer, and denom remains 1.

Rational q(3,2);

Rational **x1=r**\*Q

Rationa x2=2\*q

Overloading operator:

Raotational operator \* (const Roatiational , const rotational)

Rotational x1=r.operator \* (r,q)

* Friend allow a function to be treated as a class.
* Summary:
* 1. Objects, any type created by a class, can be inherited.
* 2. be aware of the slicing problem (make sure you know when the function is called polymophically).
* 3. references, be aware of when they come in handy.
* 4.
* Abstraction is the process of recognizing and focusing on important characteristics of a situation or object and leaving/filtering out the un-wanted characteristics of that situation or object.
* Any C++ program where you implement a class with public and private members is an example of data abstraction.
* typedef int Number; // define Number as a synonym for int
* int main()
* {
* Number total = 0;
* Number x;
* while (cin >> x)
* total += x;
* cout << total << endl;
* }
* Ram-limited resource
* Virtual ram:
* Static array:
* 1. wasted memory
* 2. not enough memory

Dynamic array:

1. resize

2. overjead & copying

3. delete correctly

LinkedList LinkedList()

:front (nullptr)

{

}

object type linknode:

one can use:

LinkNode \* ln=new linknode

To create a new object here.

Then one can do:

ln->data=s,

ln->next=nullptr;

while (cur -> next!=nullptr)

---as long as the current is not the end of the list.

Cur=cur->next;

Cur->next=ln;

Memory:

0101=5

abcdefg in (binary)=g\*1+f\*2+e\*4…….

Abcdefg in hex=g\*1+f\*16+d\*16^2….

A=10, B=11…

Const at the end of the declaration: the function does not change the object.

Const at the first: return a Boolean that cannot be changed.

Removefirst:

Void LinkedList::removefirst()

{

if (front !=nullptr)

{

LinkNode \*d=front;

Front=front->Next;

Delete d;

}

}

void LinkList::remove(string s)

{

if (front==nullptr) return;

else if (front->data==s){

removeFirst();

return ;

}

Linknode \*prev=front;

LinkNode \*cur=prev ->next

While (cur!=nullptr && cur ->data!=s)

{

prev=cur

cur=prev->next;

}

* + if (cur==nullptr) return;
  + prev->next =cur->next

}

Chmod u+x filename

./filename

Queque (interface)

-add to end

-remove from front

-count empty

-get front

stack

-top () tell me the most recently added that hasn’t been removed

-push() add to front

-pop() remove from front

template <typename T>

struct StackNode

{

T data ;

StackNode<T> \* next

}

class

{

public:

void push(T x);

private:

StackNode <T> \*test;

} ///generic programming

// what I am writing is a generic stack.

LinkList

Template—a generic way of doing something.

See the compile error for template.

Class Shape

{

public:

virtual double area ()=0;

}

do I always know the shape at compile time?

Void foo ()

{

cout << s->area();

}

virtual: at run time call the function.

Virtual double area()=0, abstract function

Inheritance:

* // constructors and derived classes
* #include <iostream>
* using namespace std;
* class Mother {
* public:
* Mother ()
* { cout << "Mother: no parameters\n"; }
* Mother (int a)
* { cout << "Mother: int parameter\n"; }
* };
* class Daughter : public Mother {
* public:
* Daughter (int a)
* { cout << "Daughter: int parameter\n\n"; }
* };
* class Son : public Mother {
* public:
* Son (int a) : Mother (a)
* { cout << "Son: int parameter\n\n"; }
* };
* int main () {
* Daughter kelly(0);
* Son bud(0);
* return 0;

}

circle is shape, rectangle is a shape.

Should a destructor be virtual?

* “=0” means abstract
* an inherited class: circle can’t access to shape’s private/public function.
* Protected means private but still accessible by the inherited object.

If it’s vitual, it’s default, if it not virtual it’s messier.

Investigate the virtual function!!!!

* using std::cout;
* *a stack is defined as a list or sequence of elements that is lengthened by placing new elements "on top" of existing elements and shortened by removing elements from the top of existing elements. It is an ADT[Abstract Data Type] with math operations of "push" and "pop".*

*A queue is a sequence of elements that is added to by placing the new element at the rear of existing and shortened by removing elements in front of queue. It is an ADT[Abstract Data Type]. There is more to these terms understood in programming of Java, C++, Python and so on.*

* [Stack](http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29) is a LIFO (last in first out) data structure. The associated link to wikipedia contains detailed description and examples.

[Queue](http://en.wikipedia.org/wiki/Queue_%28abstract_data_type%29) is a FIFO (first in first out) data structure. The associated link to wikipedia contains detailed description and examples.

An example of the template:

* // class templates
* #include <iostream>
* using namespace std;
* template <class T>
* class mypair {
* T a, b;
* public:
* mypair (T first, T second)
* {a=first; b=second;}
* T getmax ();
* };
* template <class T>
* T mypair<T>::getmax ()
* {
* T retval;
* retval = a>b? a : b;
* return retval;
* }
* int main () {
* mypair <int> myobject (100, 75);
* cout << myobject.getmax();
* return 0;

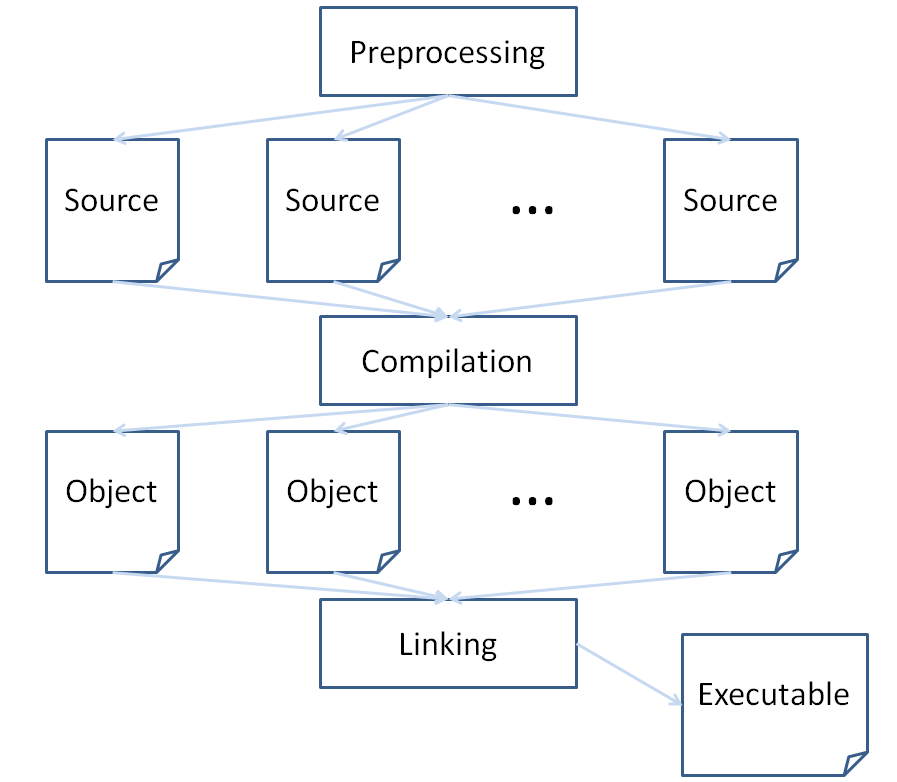
}

* If there’s no default constructor, we have to use pointer.
* When declare a pointer---no constructor gets called.
* Test::test1();
* Is the same thing as test b; b.test1();
* Static binding is decided at compiler time.
* Allyourbase\* basePtr=&derived;
* Pointing to the base class component of the derived class object.
* If tag a function virtual, now the base class pointer of a derived class object, the pointer points to the derived object.
* Make a class abstract:

Tag the destructor pure virtual.

* #define LOUD\_STRING "YELLING"
* #define COOL\_INT 42
* int main () {
* cout << LOUD\_STRING << endl;
* cout << COOL\_INT << endl;

}



Header guards:

* // If FLAG\_NAME is not defined...
* #ifndef FLAG\_NAME
* // ...then define it...
* #define FLAG\_NAME
* ...
* // Perform the necessary inclusions
* // and source body definitions here
* ...
* // ...up until you see the endif

#endif

* When we want to define the behavior of an operator (let's say, the plus sign +) for our class, we use the syntax:
* ClassName& ClassName::operator+ (SomeType other) {
* // Operator behavior within

}

operator overloading:

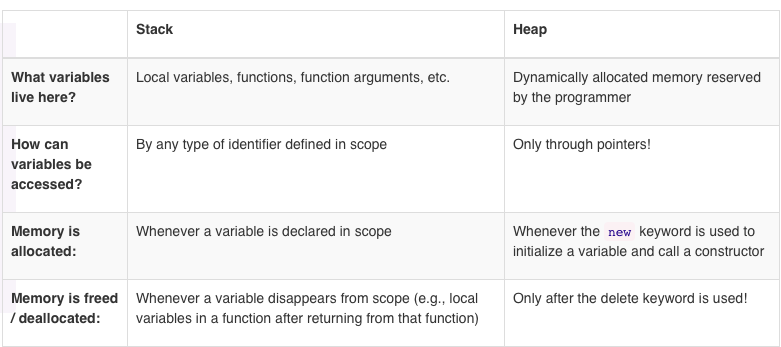
* struct OpExample {
* string s;
* int i;
* OpExample () {
* i = 1;
* }
* // [!] Using the += operator with another OpExample object
* // as the rvalue will add the rvalue's string data member
* // to the lvalue's string data member as many times as the
* // lvalue's int data member
* OpExample& operator+= (OpExample& rvalue) {
* for (int j = 0; j < i; j++) {
* s += rvalue.s;
* }
* return \*this;
* }
* };
* int main () {
* OpExample op1,
* op2;
* op2.s = "Test!";
* op1.i = 3;
* // [!] Using our custom += operator definition!
* op1 += op2;
* cout << op1.s << endl;

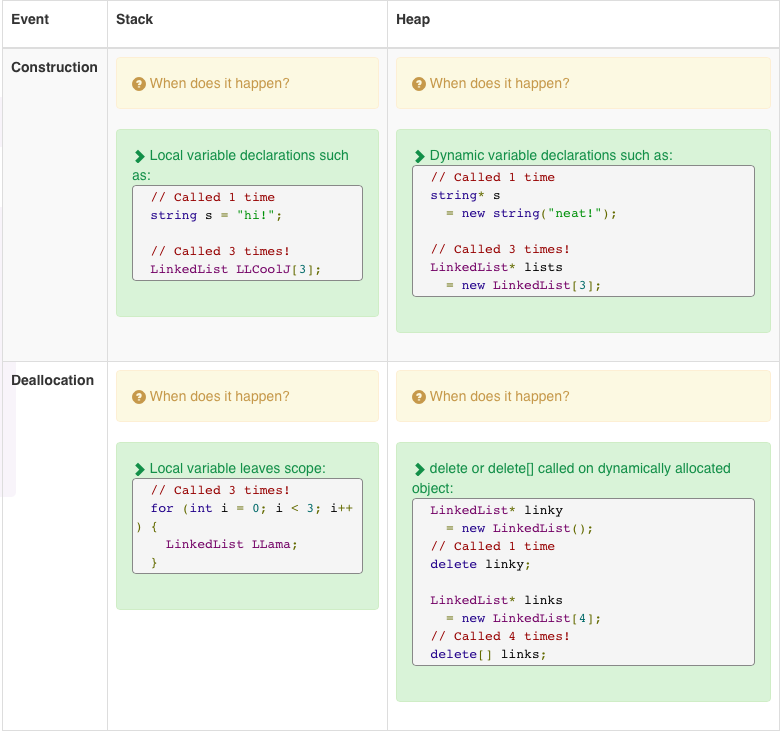
}

local memory—stored in stack.

Dynamically allocated memory—stored in heap

Pointer lives in local memory





the challenge question???

* Class Student: public Person
* {
* // all student are person
* //Not all person are student
* private: unsigned int m\_id;
* public:
* Student (std::: string Name,unsigned int age, unsigned int id);
* each subclass needs a constructor.
* Student::student()
* {
* //if don’t do initializer list, the default one gets called, the default one doesn’t exist.
  + Person(name,age), m\_id (initializer list)
* }
* Class person()
* {
* virtual std::string toString()const
* {
* return m\_name + m\_age
* }
* private:
* m\_name
* m\_age;
* }
* class x=y -🡪 copy constructor
* Student :: to String ()
* {
* return Person toString();
* }
* if it’s a static type(use without a pointer)
* & 🡪 don’t make a copy of the object, just pass it on.
* If you want an object to be treated differently then it’s daughter class, tag it virtual.
* Virtual vs. non-virtual:
* Virtual can be specialized in sub classes.
* Or even abstract. It’s slower run-time.
* Array of void \* 🡪 void \*\* vmt
* Virtual really means that:
* Virtual vs non-virtual() =p->vmt[0]();
* With a static object, it does not do a table (virtual method table).
* Copy constructor1:
* **class** Circ { **public**:
* ...
* *//copy constructor*
* Circ(**const** Circ& oldVar) { m\_x = oldVar.m\_x;  
  m\_y = oldVar.m\_y;  
  m\_r = oldVar.m\_r;
* }
* ... **private**:
* **float** m\_x, m\_y, m\_r; }
* Copy constructor2:

Line::Line(const Line &obj)

* {
* cout << "Copy constructor allocating ptr." << endl;
* ptr = new int;
* \*ptr = \*obj.ptr; // copy the value
* }
* Line line2 = line1; // This also calls copy constructor
* **overloading operator:**
* struct AssExample {
* int m\_i;
* string m\_s;
* bool m\_b;
* AssExample () {};
* AssExample (int i, string s, bool b) {
* m\_i = i;
* m\_s = s;
* m\_b = b;
* }
* // [!] Define the behavior of the assignment to NOT
* // copy over the string data member
* AssExample& operator= (const AssExample& other) {
* m\_i = other.m\_i;
* m\_b = other.m\_b;
* m\_s=other.m\_s;
* return \*this;
* }
* };
* int main () {
* AssExample a1(3, "test", 1),
* a2;
* a2 = a1;
* // [!] See what I print out now...
* cout << a2.m\_i << endl;
* cout << a2.m\_s << endl;
* cout << a2.m\_b << endl;
* }
* **a simple example of template:**
* template <typename T>
* T maximum(T i1, T i2) {
* return (i1 > i2) ? i1 : i2;
* }
* int main () {
* string s1 = "test",
* s2 = "this";
* cout << maximum(s1, s2) << endl;
* int i1 = 20,
* i2 = 8;
* cout << maximum(i1, i2) << endl;
* }

\*\*\*When templates define a generic type, e.g. typename T, then parameters defined in terms of that generic type must be EXACT matches.\*\*\*

* How to get class<type1, type2> to work:
* template<typename K, typename V>
* struct Example {
* K k;
* V v;
* Example(K kP, V vP) {
* k = kP;
* v = vP;
* }
* K& getKey () {
* return k;
* }
* V& getVal () {
* return v;
* }
* };
* template<template<typename, typename> class Container, typename K, typename V>
* void printAddedElements (Container<K, V>& c) {
* K keyDup = c.getKey() + c.getKey();
* V valDup = c.getVal() + c.getVal();
* cout << "KeyDup: " << keyDup << endl;
* cout << "ValDup: " << valDup << endl;
* }
* int main () {
* int i = 5;
* string s = "test";
* Example<int, string> e1(i, s);
* Example<double, int> e2(3.14, 3.14);
* printAddedElements(e1);
* printAddedElements(e2);
* }
* Recursive:
* int getDiff(int X, int Y) { // Tail Implementation
* X++; // Inc X
* if (X == Y)
* return 1;
* return (1+getDiff(X, Y));
* }
* void getDiff2(int X, int Y, int Diff = 0) {
* X++;
* Diff++;
* if (X != Y)
* getDiff2(X, Y, Diff);
* else
* cout << "Diff Was: " << Diff << endl;
* return;
* }
* int main(void) {
* cout << "Difference Between 1 & 7 Is: " << getDiff(1, 7) << endl;
* getDiff2(2, 8);
* return 0;
* }
* a recursive function:
* #include <iostream>
* using namespace std;
* void numberFunction(int i) {
* cout << "The number is: " << i << endl;
* i++;
* if(i<10) {
* numberFunction(i);
* }
* }
* int main() {
* int i = 0;
* numberFunction(i);
* return 0;
* }
* one can not define a template in a separate file.
* 1. never include cpp file.
* 2. use the define to define new constant:
* #define PI 3.14159

Linked list:

Elements that are connected. Each element has a pointer

Struck house

{

string name;

house \*next;

}

house \*latRented

void add\_animal (string snimal)

{

house \*latest=new house;

lasters->name=animal;

lastest->next=lastRented

lastRented=lastest.

}

every linked list needs to hold at least one piece of private data.

einsteinsLinkedList

{

public

void addItemToFront

void delete Item

int find Item

void prointItems

constructor

descructor

private:

house \*m\_head;

int m\_numItems;

}

two cases

if empty

if not empty—follow the thing to the last element and insert.

Temp=m\_head;

While (temp->next!=nullptr)

{

temp=temp->next;

}

then do:

house \*latest=new house;

lastest->name =animal

lastest->next=nullptr;

temp->next=lastest;

}

stack:First in last out

queque: first in first out.

The circular quque:

Private data:

An array: arr

An integer: head

An integer: tail

An integer: count.

* void Stack::print () {
* if (head == nullptr) {
* return;
* }
* for (Node\* pointy = head; pointy != nullptr; pointy = pointy->m\_next) {
* std::cout << pointy->m\_data << " ";
* }
* std::cout << std::endl;

}

linkedlist \*links=new linkedlist[4];

* NoisyClass (const NoisyClass& other) {
* s = other.s;
* cout << "[~] Copy constructor" << endl;
* }

A **copy constructor** is a special constructor definition that is called whenever a new object of some class C is created from another object of that same class C.

* // A copy constructor called using the source
* // Kinkos object, inWestwood, creating a new
* // Kinkos object expansion!

Kinkos expansion(inWestwood);

* OpExample& operator+= (OpExample& rvalue) {
* for (int j = 0; j < i; j++) {
* s += rvalue.s;
* }

return \*this;

erase:

* bool LinkedList::erase (std::string s) {
* Node\* iterator = head;
* Node\* prev = nullptr;
* bool erased = false;
* // First, find the Node to delete...
* while (iterator != nullptr) {
* // If we don't find a match, keep looking
* if (iterator->m\_data != s) {
* prev = iterator;
* iterator = iterator->m\_next;
* // Otherwise, perform the necessary deletion
* } else {
* prev->m\_next = iterator->m\_next;
* delete iterator;
* erased = true;
* break;
* }
* }
* // Return whether or not we found
* // and erased anything
* return erased;

}

* If a data member is defined in both a Derived class and its Base class, AND:
* An object is an instance of the Derived class, then we use the Derived class' definitions of those members.
* An object is an instance of the Base class, then we use the Base class' definitions of those members.

I said that a Base class pointer can point to the Base class portion of one of its Derived class' objects... it does NOT work the other way around.

AllYourBase\* basePtr = &derived; this calls base constructor.

If tages virtual, it calls a derived class function.

Purse virtual function: goes up the hierarchy until find the right parameter.

template<typename T>

* // [!] Note: a2 has already been constructed,
* // so the copy constructor won't be called, but rather
* // the assignment behavior will occur
* a2 = a1;

On midterm-look for the base cases in recursive method.

Node \* merge (Node \*a, Node \*b)

{

if (a==nullptr) return b;

else if (b==nullptr) return a;

else if (a->data <= b->data)

{

* + Node \*n=new Node;
  + N->data=a->data;
  + N->next=merge(a->next,b);

}

else (a->data > b->data)

{

* + Node \*c=new Node;
  + N->data=b->data;
  + N->next=merge(a,b->next);

}

} //implace operation.

Good because it breaks into small pieces.

x

* Recursion:
* Rule 1 (Base case): We take the simplest, most basic problem and handle this directly and return a result without using recursion.  
  Rule 2: Each time a recursive function calls itself, the new call must operate on a smaller problem (closer to the solution) then the previous call. You MUST make sure that the "base case" will eventually be reached; thus allowing the function to stop calling itself.
* Copy constructor: constructor that initializes a new variable using an existing variable of the same type
* Assignment operator: Same as above, but setting a new variable to the same value as an existing variable. Copy constructor – allocate memory
* Assignment operator - free up dynamic memory, allocate new memory, explicitly copy contents
* Each time you use a template function with a different type of variable, the compiler generates an entirely new copy of the function in your program.
* You MUST use the template data type (e.g. Data) to define the type of at least one formal parameter, or you’ll get an ERROR!
* STL supports predefined ADTs through three types of various items: containers, iterators, algorithms. Examples of containers: Linked lists, Stacks, Queues, Sets, Maps, Vectors
* Week 2 slide:
* Copy constructor:
* In fact, it is nothing but a special constructor with a parameter of the class type.
* When is it being called?
* Creating an instance using another instance.
* Circ a(1,2,3); Circ b(a);
* Assigning an uninitialized instance with another one.
* Circ a(1,2,3); Circ b = a;
* Passing an argument by value.
* compute\_area(Circ a);
* Why do we need our own copy constructor? Allocate dynamic memory.  
  Open system resources, e.g., opening a file.
* Circ& **operator**=(**const** Circ& src)
* Week 3:
* **void** insertToKth(**int** val, **int** index) {  
  **if**(head == nullptr || index == 0) insertToFront(val); **else** {
* Node\* cur = head; **while**(cur->next != nullptr) {
* **if**(--index == 0) **break**;
* cur = cur->next; }
* *//cur points either to last element //or (index-1)st element*Node\* p = **new** Node;  
  p->value = val;
* p->next= cur->next;
* cur->next = p; }
* }
* int LinkedList::count (std::string s) {
* Node\* n = head;
* int count = 0;
* // Go through all size Nodes
* // in the List and compare to s
* for (int i = 0; i < size; i++) {
* if (n->m\_data == s) {
* count++;
* }
* n = n->m\_next;
* }
* return count;
* }
* guss in the middle log2(n) tries.
* Long choose (long n, long k)
* {
* if (k==0 || n==k) return 1
* else
* //return choose(n-1,k)+choose(n-1,k-1);
* long x=choose(n-1,l);
* long y=choose(n-1,k-1);
* return x+y;
* }
* int search(int a[], int start, int end, int x)
* {
* if (start > end)
* return -1;
* int mid=(start+end)/2
* if (a[mid]==x) return mid;
* else if (a[mid]<x)

return search(a,mid+1,end,x);

* else
* return search(a,start,mid-1,x);
* }
* int factor(int n)
* {
* if (n<1)
* return 1;
* else
* {
* return factor(n-1)\*n;
* }
* }
* int main()
* {
* cout << factor(5);
* }
* int fib(int n)
* {
* if (n<2) // need more than 1 base cases.
* return 1;
* else
* {

* return fib(n-1)+fib(n-2);
* }
* }
* int main()
* {
* cout << fib(4);
* }
* recursive approach to find the kth smallest number
* do a findMin try doing that at home—

int quickselect(int \*a, int n, int k)

{

int p=partition(a,n)

if (k==p) return a[p]

else if(k<p)

return quickSelect(a,p,k);

else

return quickSelect(a+p,n-p-1,k-p);

}

* int partition (int a[], int n)
* when done, what was in a[0] is in the right spot
* (where it would go if a was sorted )
* and all a “corrected ” w/resp to pivot
* AllYourBase\* basePtr = &derived;
* speak function of Animal class doesn’t make sense.
* Make it pure virtual by putting ”=0” in declaration of base class.
* virtual void speak() = 0;
* You don’t implement pure virtual functions in base class.
* However, derived classes should implement pure virtual functions to be able to be instantiated.
* Constant folding—change the variable directly into the number.
* Things to affect the runtime in my program:
* -computer used
* -software used
* -choice of programming language
* -choice of input
* -availability of resource.
* If n is the number of elements in link list
* `as the amount of data change, how does the performance change? Regardless of running environment.
* Void LinkLisk.add((int t))
* {
* if list is empty
* m\_front=m\_tail=new

else

m\_tail->next =new …

m\_tail =m\_tail->next

* }
* check if sorted:
* compare the adjacent pair.
* Bubble sort:check the adjacent two if they are close together.
* Two for loop, n to the second.
* run time: constant, linear, quadratic
* only concerns with the highest ordered term in the equation
* <http://en.wikipedia.org/wiki/Ackermann_function>
* selection sort:
* O (n^3) time.
* Not the best O notation.

Standard template function

O(n^2) same as 0(2n^2) (but don’t write the latter)

Mergesort:

Log2(n);

0(n\*log(2)n);

---Mergesort is really efficient.

Downside: it takes more Ram than the other three.

Quick select:

--find a pivot

partitioned around it

-if pivot not gal, search in relevant…

worst case—n(n^2).

Rated sort.

Is it possible for a algorithm to be better than O(n\*log2(n)) in its worst case?

---No lower bound.

Yes-if it’s not comparison based.

---counting sort 0(n+k);