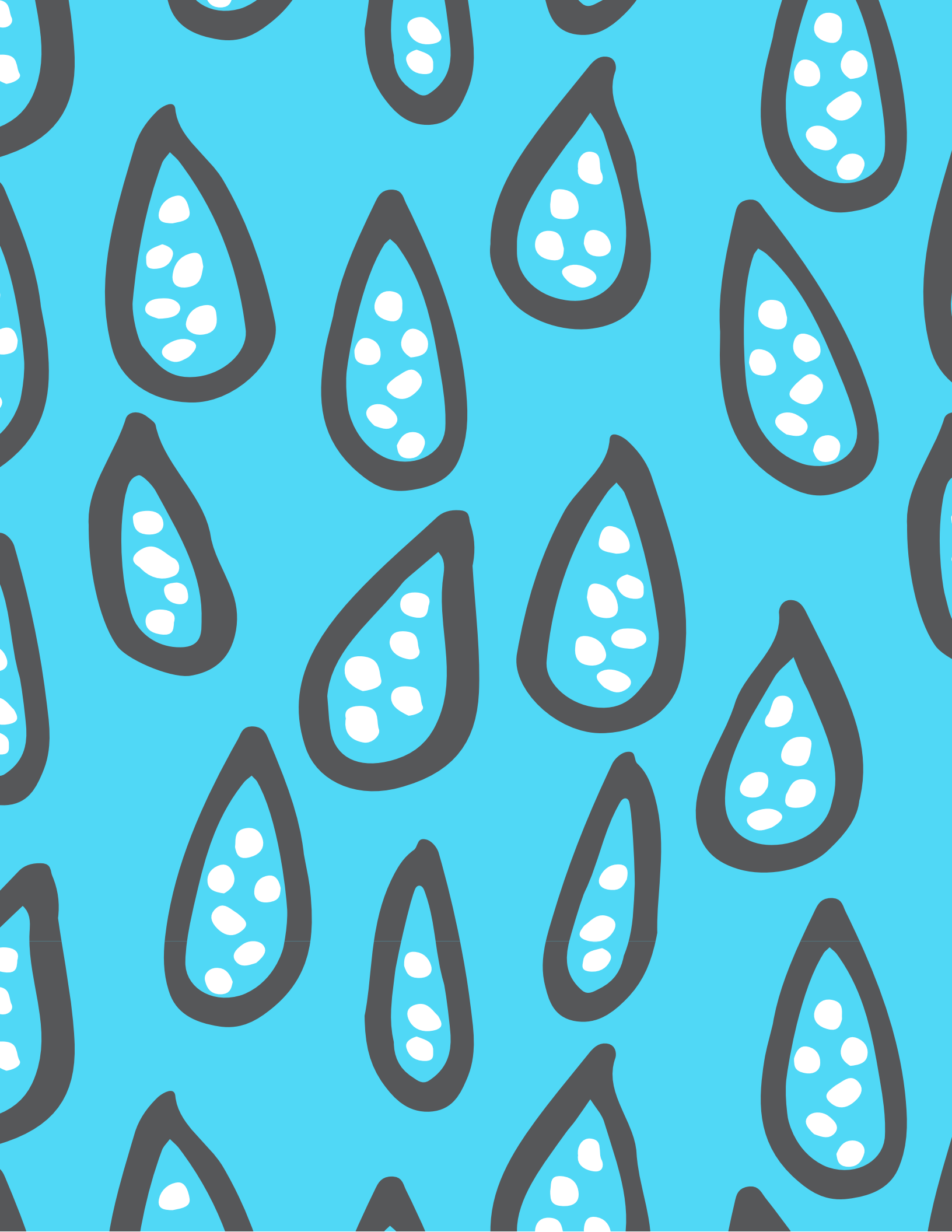


Midterm Exam Notes

Can use AI, just don't rely on it, double check outputs and understand what is being outputted

Memory and OOP

Abstract Data Types (ADT)

- Model of a data structure that specifies:
  - (i) what operations can be performed on the data
  - (ii) not how these operations are implemented

Need to choose data structure that is the most efficient

- add(item): add item to the bag
- remove(item): remove one occurrence of item (if any) from the bag
- contains(item): check if item is in the bag
- grab(): get an item at random, without removing it
- numItems(): get the number of items in the bag

These operations are provided but not implemented!

Example: A Bag ADT

just a container for a group of data items

(i) Positions of data items don't matter
- $\{3, 2, 6\} == \{2, 6, 3\}$

(ii) Data items do not need to be unique
- $\{7, 2, 10, 7, 5\}$ is a bag, but not a set

Assumptions:

(i) Bag contains integers only

(ii) We will use arrays to store the data

Assume all methods above are implemented, only write out the add(...) method

Implementation of IntBag in Java

```
public class IntBag implements Bag {
    // instance variables
    private int[] items;
    private int numItems;
    // methods
    public boolean add(int item) {
        if (numItems == items.length)
            return false;
        items[numItems++] = item;
        return true;
    } ...
}
```

Access Modifiers: private - other classes cannot access; public - can be accessed from another class

Encapsulation

revisit access modifiers

Suppose a client has a class 'MyClass' and wants to use IntBag.

```
class MyClass {
    ...
    void myMethod() {
        IntBag b = new IntBag();
        b.items[0] = 17; // not allowed!
    ...
```

b.add(17)

Cannot Modify field directly due to being private, but add method allows 'controlled' modifications

Contains using private fields and public methods (occasionally private helper methods)

Generic Types

Generics allow you to store any data type, so the bag can use any type specified

Can use Object superclass, just watch for type mismatches

Using a Superclass to Implement Generics

```
public class Bag {
    // instance variables
    private Object[] nums;
    private int numItems;
    // methods
    ...
}
public class Test1 {
    psvm {
        Bag m = new Bag();
        m.add("37C");
        String badTemp = (String) m.grabItem();
        m.add(new Integer(96));
        Integer temp = (Integer) m.grabItem(); // runtime
    } ...
}
```

Watch for Typecasting/Downcasting

must specify to prevent runtime error

Assume grabs at [0]

Error bc ["37C", 96]

Arrays vs. Linked Lists

constant time element access

only know reference to header

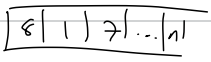
O(n) element access time

Phonebook Example

- Implement a phonebook
  - (1) Operations: add(element)
    - remove(element)
    - search(name) return number
  - (2) Data Structures: Array (sorted/unsorted)
    - Linked List (sorted/unsorted)

add/remove should not disturb sorting

Unsorted Array



Add: Just Add to end

Remove:
- best case: 1 check
- worst case: n comparisons
- average case: $\frac{n}{2}$ comparisons

Example of Sorted Array Addition

1) log n Binary Search
2) n Shifting Elements

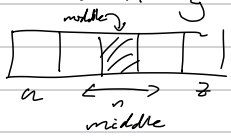
add: $O(n + \log n)$

$\log n \leq n$

so we say $O(n)$

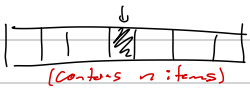
Binary Search

Sorted Array ← You need a sorted Array for this to work

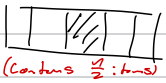


O(log(n)) complexity

1.



2.



target lies left of target → now only need $\frac{n}{2}$ of array → $\frac{n}{4}$ time

```
findNumber(Person)        pseudocode
  low=0
  high = phonebook_size
  while low <= high {
    p = floor((low+high)/2)
    Compare pth person
    if the same
        return number
    else if earlier in book
        high = p-1
    else
        low = p+1
  } return Not_found
}
```

Recursion Mathematical Background

Non-recursive Programming

- More Examples
- Calculate $\sum_{i=1}^{n} i$ can be done using 'for' loop
  - $\rightarrow sum(n) = \sum_{i=1}^{n} i = 1+2+3+\ldots+n$
- Can calculate recursively, but less intuitive

```
int sum(int n) {          // ← Example Implementation
    int i, sum;
    sum=0;
    for (i=1; i<=n; i++) sum+=i;
    return sum; }
```

Recursive Programming

- $\sum_{i=1}^{n} i = 1+2+3+\ldots+(n-1)+n$
  - $sum(n-1)$
  - $\rightarrow sum(n) = sum(n-1)+n$ (recursive call)
  - Reduces size of problem!
- Recursive call must be shorter to allow code to terminate

```
int sum(int n) {                    // or 1, either works
    if (n<=0)
        return 0;
    int sum_result = n + sum(n-1);
    return sum_result;
}
```

sum(6) = 6 + sum(5) = 6+5+sum(4) = ...

Thus does not work, it will run forever

We need to include a base condition of 1

- A recursive method is a method that invokes itself
- Must specify a base/stopping condition!
- Can get tricky to look at in stack
  - ↳ Simple to implement but hard on computer memory as no computations done until 'n=0'
- Code will run forever without a base case → stack overflow usually
- Stack works in a First In - Last Out manner

```
int sum(int n) {
    ...
}
psum() {
    ... x = sum(3); ...
}
```

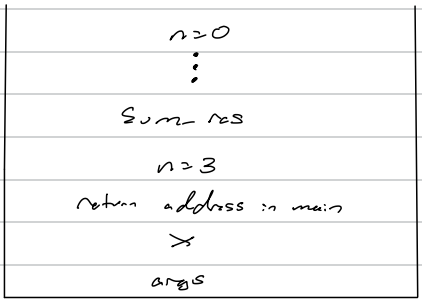
main() calls sum(3)

sum(3) calls sum(2)

⋮

sum(3) returns 3+3 or 6

main assigns x = 6



When n gives, space complexity gives a lot more → inefficient

Example: Counting Occurrences of a character in a String

Thinking Recursively

How can we break this problem down into smaller sub-problems?

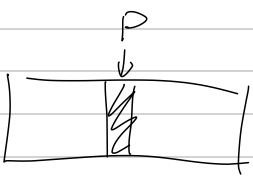
What is the base case? if (s.length()==0) return 0;

Do we need to combine the solutions to the sub-problems? If so, how?

```
int occurrences(String s, char c) {
    if (s.length()==0) return 0;
    if (s.charAt(0)==c)
        return 1+ occurrences(s.substring(1),c)
    else
        return 0+ occurrences(s.substring(1),c)
}
```

Example: Reversing an Array → code in java

```
myFindNumber(person, low, high)
    if (low>high) return NotFound
    p = floor(avg(low,high))
    if the same
        return number
    else if earlier
        high = p-1
    else
        low = p+1
```



Fibonacci

- Call Tree Created
- Iteration better

Runtime Analysis

```
int sum(float[] array) {
    for (int i = 0; i <= array.length; i++)
        array[i] = array[i]/0.2;
}
```

```
int sum(float[] array) {
    for (int i = 0; i <= array.length; i++)
        array[i] = array[i]*5;
}
```

- Size of the Array Determines runtime
  - ↳ These shown are constant time operations
- Both methods need 2N operations

Function Growth Rates: 'Big O' Notation

Consider positive functions $T(N)$ and $f(N)$

↳ $f(N)$ is runtime complexity of $T(N)$

"Big O": $T(N) = O(f(N))$ if $\exists\ c, n_0 > 0$ s.t. $T(N) \le c \cdot f(N)\ \forall\ N \ge n_0$

↳ we focus on very large Numbers to determine

Example: $10 * N^2 + 10000 = O(N^3)$?

$10 * N^2 + 10000 \le cN^3$

constants only matter for small N

$N^2 \le N^3$ ← grows much faster

$10 * N^2 + 10000 = O(N)$? No!

interested in tightest bound $10 * N^2 + 10000 = O(N^2)$?

↳ Can be harder large c, so choose $O(N^2)$

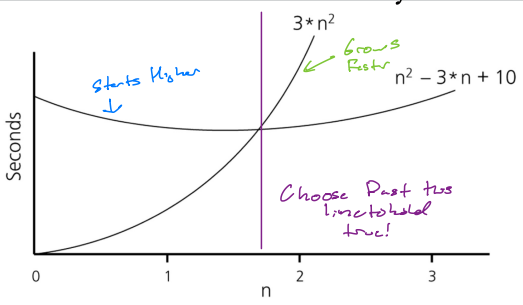
Remember you can choose c and n_0 values!

Show that $T(n) = n^2 - 3 \cdot n + 10$ is order of $O(n^2)$

- Show that there exist constants c and n_0 that satisfy the condition

$T(n) = n^2 - 3n + 10$

$n^2 - 3n + 10 \le 3n^2 \quad , n \ge n_0$



Remember

We use almost always interested in the tightest boundary!

Function Growth Rates: Other Definitions

$T(N) = \Omega(f(N))$ if there are positive constants c and n_0 such that $T(N) \ge cf(N)$ for all $N \ge n_0$

- Example: $0.0001 * N^3 = \Omega(N^2)$

Opposite to Big O

$T(N) = \theta(f(N))$ iff $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$

- Example: $0.001 * N^2 + 10000 * N = \theta(N^2)$

Must be equal!

$T(N) = o(f(N))$ if for *all* constants c there exists an n_0 such that $T(N) < cf(N)$ for all $N > n_0$.

∀ instead of ∃

or

$T(N) = o(f(N))$ iff $T(N) = O(f(N))$ and $T(N) \ne \Omega(f(N))$

- Example: $10 * N^2 + 10000 = o(N^3)$

Big-O Toolbox (for positive monotone Functions)

- Constants do not matter: $T(N) = O(f(N) + c) \rightarrow T(N) = O(f(N))$
  $T(N) = O(c * f(N)) \rightarrow T(N) = O(f(N))$
  $T(N) + c = O(f(N)) \rightarrow T(N) = O(f(N))$
  $T(N) * c = O(f(N)) \rightarrow T(N) = O(f(N))$
- Algebraic Properties:
  If $T1(N) = O(f(N))$ and $T2(N) = O(g(N))$ then $T1(N) + T2(N) = O(f(N) + g(N))$
  If $T1(N) = O(f(N))$ and $T2(N) = O(g(N))$ then $T1(N) * T2(N) = O(f(N) * g(N))$
  If $T1(N) = O(f(N))$ and $g(x)$ is monotonic, then $g(T1(N)) = O(g(f(N)))$
- Dominated Terms don't matter: $T(N) = O(f(N) + g(N))$ & $g(N) = o(f(N))$
  Then $T(N) = O(f(N))$

Little-o, smaller!

Logarithmic Properties

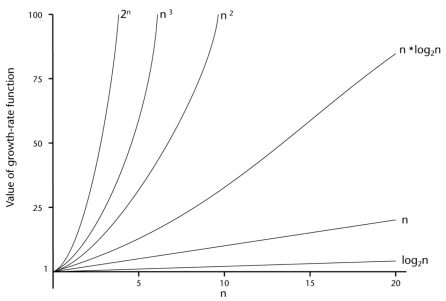
$$\log(cd) = \log(c) + \log(d)$$
$$\log(c/d) = \log(c) - \log(d)$$
$$\log(c^d) = d \log(c)$$
$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

Corollary: logarithmic grows slower than any polynomial

$\log N < N^c$

$\rightarrow T(N) = N^c + \log(N)$ (Dominated)

Comparison of Growth-Rate Functions



Useful Mathematical Equations

- $\sum_{i=1}^{n} i = 1+2+\ldots+n = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$
- $\sum_{i=1}^{n} i^2 = 1+4+\ldots+n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$
- $\sum_{i=0}^{n} 2^i = 0+1+2+\ldots+2^{n-1} = 2^n - 1$

Applying Concepts & Relative Rates

- 1000000 versus 0.01*sqrt(N)
- log(N) versus sqrt(N)
- Nlog(N) versus $N^{1.001}$ — $N^{1.001} = N^* N^{0.001}$
- N^3 versus 10000*N^2
- $\log^2(N)$ versus 10*$\log(N^5)$ — $10*\log(N^5) = 50*\log(N)$
- $2*\log_2(N)$ versus $\log_3(N)$ — $\log_a(N) = \frac{\log_c(N)}{\log_c(3)}$
- $N*2^N$ versus 3^N — $3^N = 1.5^N * 2^N$

Algorithm Analysis

Models and Assumptions

- Consider rather abstract algorithm (a procedure or method)
- Ignore the details/specifics of a computer
- Assume sequential process (a sequence of instructions)
- Ignore small constant factors (e.g., differences among "primitive" instructions)
- Ignore language differences (e.g., C++ versus Java)

Average-case versus worst-case performance

- Example: finding a person in phonebook using sequential search
  - Best-case?
  - Worst-case?
  - Average-case?

How to Calculate Running Time

```
public static int sum(int n)
{
  int partialSum = 0;
  for (int i = 1; i <= n; i++)
    partialSum += i*i*i;
  return partialSum;
}
```

$(c \times N) + b$

Time Complexity: $O(N)$

General Rules

- Simple Statement: Constant
  ex: i++, i<n, etc.
- Simple Loops: # iterations
- Nested loops: Product of
  #iterations of outer and inner
  times cost of inner body

```
ex: for(n)
      for(m)
        k++
```

- Consequtive Statements: Count most expensive

```
ex: i=0;
    while();   } O(n)
    for(n)
      for(m)   } O(n^2)
        k++
```

- Conditionals: Count most expensive branch
  ↳ Focus on worst-case

Examples

```
if(x<3) {
    k++;        <- O(1)
} else {
    k=1;
    for(i=1; i<n; i++) {
        k++;
    }           <- O(n)
}
```

```
i = 0;
while (i < n) { ... i++; ...}     m
for (i=0; i < n; i++)             n^2
    for(j=0; j < n; j++)          <- O(m+n^2)
      k++;                        n^2>n, but n^2≠m
```

```
k=1;
for(i=1; i<n; i++)        n
    for(j=1; j<m; j++)    m
        k++;              <- O(n*m)
```

```
k=1;                      <- O(n^2)
for(i=1; i<n; i++)
    for(j=i+1; j<n; j++)
        k++;
```

$\sum_{i=1}^{n-2} i \rightarrow \frac{n^2}{2}$

```
for(i=1; i<n; i=3*i) {
    j=1;
    while(j<n)
        j=j+1;            n times
}
```

$1 \rightarrow 3 \rightarrow 3^2 \rightarrow 3^3 \rightarrow \ldots 3^k$

$3^k = n \rightarrow k = \log n$

$O(n \log n)$

Linked lists

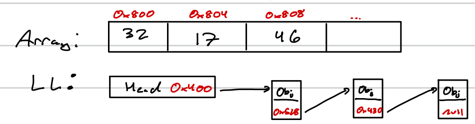
Array Representation

- ✓ Easy & efficient access to any access
- ✓ Constant time item access
- ✗ Need to have size specified
- ✗ Difficult to add/remove in arbitrary indices

Memory Differences

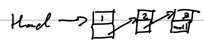
Array: Elements occupy consecutive memory locations in heap

LinkedList: Node is a distinct object in heap so locations don't have to be next to each other in memory

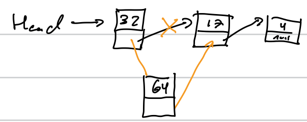


LinkedList Representation

- ✓ Dynamic Sequence: easy addition/removal
- — Links of nodes which each have a reference and data value
- ✗ Access is linear O(n) to get the nth element; Head → desired node
- — Last Data Node is a null reference
- ✓ No capacity limit provided enough memory
- ✗ Memory overhead for the links



Addition:



only requires shift of pointers, no additional ops.

Example LL

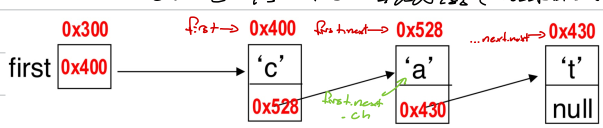
```
public class StringNode {
    private char ch;
    private StringNode next;
    ...
}
```

```
public class LLString {
    private StringNode head;
    private int theSize;
    ...
```

↑ Dynamic Instance Variable

A reference is also a variable

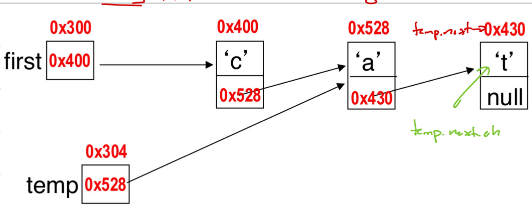
→ Has its location in memory and whose value is the address (location) of the data



node.ch → ch

node.next → memory

next only returns the memory location!



Two nodes can have two different pointers.

Diff pointers can point to same memory address

Recursion on Linked Lists

Recursive definition of a LL

A LL is either:

(i) empty (base case)

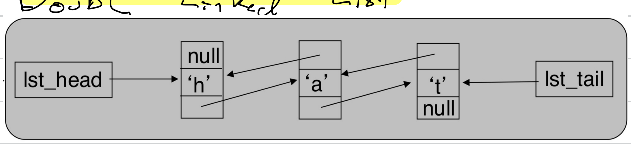
(ii) a single node followed by a LL (recursive case)

Example: length of a string

- length of "cat" = 1 + length of "at"
- length of "at" = 1 + length of "t"
- length of "t" = 1 + length of the empty string (which is 0)

```
private static int length(StringNode str) {
    if (str == null)
        return 0;
    else
        return 1 + length(str.next);
}
```

Doubly Linked List



Example DLL

```
public class LLString {
    private StringNode lst_head;
    private StringNode lst_tail;
    private int theSize;
    ...
```

```
public class StringNode {
    private char ch;
    private StringNode next;
    private StringNode prev;
    ...
}
```

Allows for traversal in either direction

In Class DLL example

↳ Assume you have LL String and StringNode

→ Access node at position i in a doubly linked list

```
public StringNode getNode(int i) {
    if (i < 0 || i >= theSize) return null;
    // Use the half i is closest too
    if (i < theSize/2) {
        ptr = this.lst_head;
        for (j = 0; j != i; j++) ptr = ptr.next;
    else {
        ptr = this.lst_tail;
        for (j = theSize; j != i; j--) ptr = ptr.prev;
    }
    return ptr;
```

Example Removing a Node from a DLL

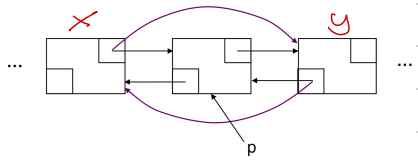
↳ Assume Access to previous class

```
public char remove(String Node p) {
    if (p == lst.head || p == lst.tail)
        Need different calc!
    p.next.prev = p.prev;
    p.prev.next = p.next;
    theSize--;

    return p.ch;
}
```

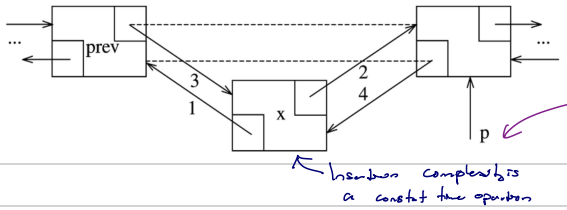
x, y, x, x, y

assumes p is in the middle

Number of removed references depends on if node is head or tail or not



Same Idea for inserting a Node

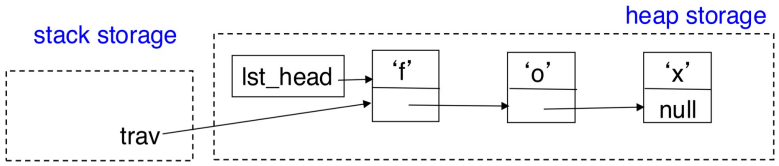


Insertion complexity is a constant time operation

$O(n+4) = O(n)$

Always need to find 'p', so starts at $O(n)$ complexity

More LinkedList Operations



toUpperCase(): Convert str to uppercase

```
private static void toUpperCase(StringNode str) {
    StringNode trav = str;
    while (trav != null) {
        trav.ch = trav.ch.toUpperCase();
        trav = trav.next;
    }
}
```

First node pointer

```
if (trav.ch >= 'a' && trav.ch <= 'z')
    trav.ch += ('A' - 'a');
```

Constant offset

Rest of the class will use singly linked lists usually

General Traversal Support

→ Instead of use public LLString methods

```
public class MyClass {
    public static int numOccur(LLString str, char ch) {
        int numOccur = 0;
        for (int i = 0; i < str.length(); i++) {
            if (ch == str.get(i))
                numOccur++;
        }
        return numOccur;
    } ...
}
```

Bypass private fields

Complexity O(n)

Gets ith element from LL. Complexity: O(n)

O(n²) as each iteration requires a full search to ith position

Duplicating a Singly LL

- Helper method copy(str)
- Copy all elements through end
- Return first element of new list

→ Recursive Implementation

(i) Base Case: If str is empty, return null

(ii) Recursive: Copy to call on next

Assume StringNode is well defined

```
private static StringNode copy(StringNode str) {
    if (str == null) // base case
        return null;
    // create the first node, copying the first character into it
    StringNode copyFirst = new StringNode(str.ch);
    // make a recursive call to get a copy of the rest and
    // store the result in the first node's next field
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```

How can we improve efficiency and follow good OOP practices

→ (i) Or you can you a traversal mode

```
public class LLString {
    public int numOccur(char ch) {
        int numOccur = 0;
        StringNode trav = lst_head;
        while (trav != null) {
            if (trav.ch == ch)
                numOccur++;
            trav = trav.next;
        }
        return numOccur;
    } ...
}
```

O(n)

Starts from beginning, goes to end

Not Encouraged

Only allows you to use numOccur don't want to have to create methods for any of the other use cases.

→ (ii) Get Access to list internals

```
public class MyClass {
    public static int numOccur(LLString str, char ch) {
        int numOccur = 0;
        StringNode trav = str.getNode(0);
        while (trav != null) {
            char c = trav.getChar();
            if (c == ch)
                numOccur++;
            trav = trav.getNext();
        }
        return numOccur;
    } ...
}
```

Constant Time operations

getNode() is LLString

getNext() is StringNode

This is O(n) as well

Makes public fields, which you can do this through getters and setter methods.

→ (iii) Provide an Iterator

```
public class MyClass {
    public static int numOccur(LLString str, char ch)
    {
        int numOccur = 0;
        LLString.Iterator iter = str.iterator();
        while (iter.hasNext()) {
            char c = iter.next();
            if (c == ch)
                numOccur++;
        }
        return numOccur;
    }
    ...
}
```

Also O(n)

- No usage of StringNode Objects
- Does not depend on LLString Internals

Iterator ← This is an Object, it needs to be created

- Provides Iterator ability w/o violating encapsulation
- hasNext() – Checks if current node points to a non-null node
- next() – Returns next item and increments the iterator

Position of iterator given as '|'

- *after the iterator i is created:* | "F" "O" "X" ...
- *after calling* i.next(), *which returns* "F": "F" | "O" "X" ...
- *after calling* i.next(), *which returns* "O": "F" "O" | "X" ...

It is implemented below as a inner class

```
public class LLString {
    private StringNode head;
    private StringNode tail;
    ...
    public Iterator iterator(){
        Iterator iter = new Iterator();
        return iter;
    }
}
```

```
public class Iterator {
    private StringNode nextNode;
    private Iterator (){
        nextNode = head;
    }
    public boolean hasNext() {...}
    public char next() {...}
    ...
}
```

Internal workings of Iterators

```
public boolean hasNext() {
    return (nextNode != null);
}
public char next() {
    if (nextNode == null)
        throw new Exception("Falling off the list end");
    char ch = nextNode.ch;
    nextNode = nextNode.next;
    return ch;
}
```

Instantiation:

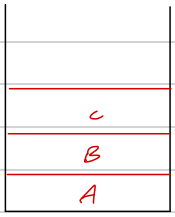
```
LLString.Iterator myIter1 = string.iterator();
LLString.Iterator myIter2 = string.iterator();
```

Parent class, Nested class, Name of object to create it for

Stacks & Queues

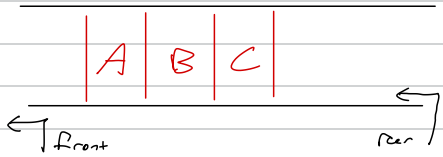
- Linear Data Structures just like arrays and linked lists

Stacks



- First In, Last Out! (FILO)
- push(...) = add
- pop() = remove
- peek() = get item @ top, but don't remove
- isEmpty() and isFull()

Queue

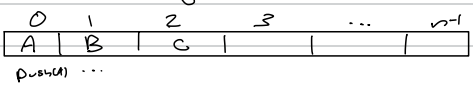


- First In First Out (FIFO)

Everything inserted from the rear and comes out of queue from front

enqueue(...) - add
dequeue() - remove
- peek(), removal
- insert
- isEmpty(), isFull()

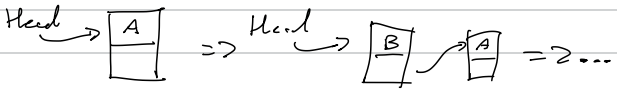
Build with an array



max = n, #items = 3, top = #items - 1

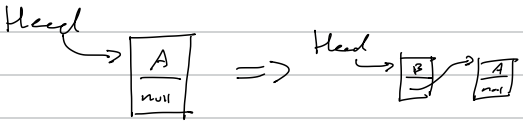
pop() will decrement top, remove I[top]

Linked List Implementation



- Adding is Constant time O(1)
- This makes Removal linear ~ O(n)

Build with a Linked List



- push(...) should add to front, always ptr
- pop() removes element, head pts to next

Making this a DLL, it has constant enqueue and dequeue operations

No Capacity Issue

Need to Use a DLL with two Pointers so that the methods are as efficient as possible

Use type variables as placeholders

Generic Array Stack Class

```
public class ArrayStack<T> {
    ...
    public ArrayStack(int max) {
        items = (T[]) new Object[max];
        top = -1;
        maxSize = max;
    }
    ...
}
```

top++ : use current var, then increment
++top : increment and use that var

Summary Q: Emulate Queue using Stacks

Array Implementation

Consider a 'circular array'

- T[] items
- int front, rear, numItems, maxSize

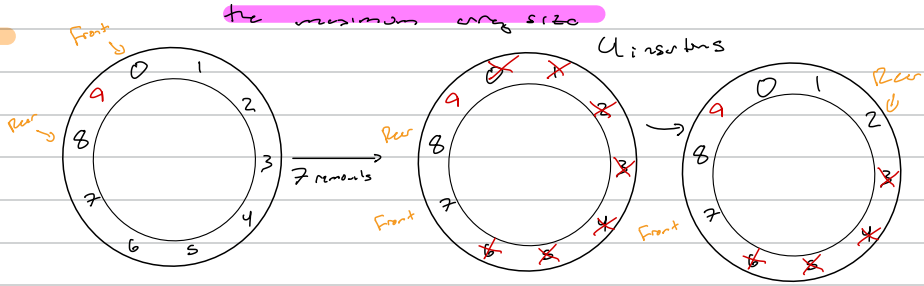
→ Instead of shifting indices, increment the front

Get to end of array, add to start of array

All index operations are modulo array, and modulo is the maximum array size

Queue Array Implementation

```
// Add an element to the circular array
public void add(int value) {        // isFull()
    if (size == array.length) {
        System.out.println("Array is full");
        return;
    }
    array[tail] = value;
    tail = (tail + 1) % array.length;
    size++;
}

// Remove and return an element from the circular array
public int remove() {
    if (size == 0) {
        throw new                   // isEmpty()
IllegalStateException("Array is empty");
    }
    int value = array[head];
    head = (head + 1) % array.length;
    size--;
    return value;
}
```



The queue is empty when front "overcomes" rear:

→ ((rear + 1) % maxSize) == front

You also have to look at number of items as this breaks for both full & empty

Basics of Trees

↳ First non-linear data structure

Recall: Phone Book

Data Structe	Search	Insert
Sorted Array	O(log n) w/ binary	O(n) due to Shifts
Linked List	O(n) using linear	O(1) as LL

What is a Tree

- A set of nodes, top one is called the 'root'
- A set of edges connecting pairs of nodes
- Cannot have cycles - only 1 unique path to any given node!
- Nodes have data ("payload") consisting of one or more fields
- Recursive Data Structure: Each Node in the tree is the root of a smaller tree
  ↳ Refer to these types of trees as subtrees

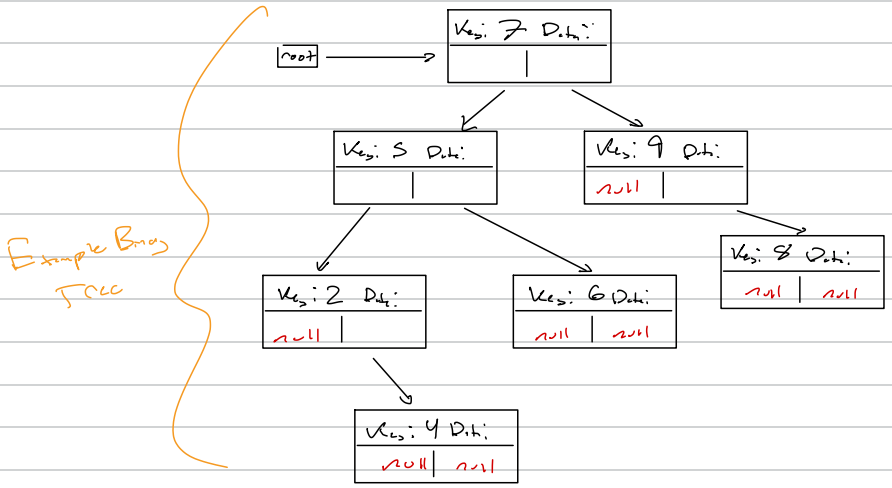
Terminology

- Key Field = field used when searching for a data item
- If a Node N is connected to other nodes below it, it is called the parent and its nodes below are children
  ↳ A node can only have one parent, but can be one of many 'siblings'
- Leaf Nodes: A node w/o children ← doesn't have to be the deepest node!
- Interior Nodes: A node which is neither Leaf nor Root
- Depth of a Node: # of edges on the path from it to the root
- Level of the tree: Made up of Nodes with the same depth
- Height of a tree: Maximum depth of its nodes
- Binary Trees: Each Node has at most Two children

Binary Trees

- Recursive Definition: A binary tree is either a collection of nodes that is either
  :) empty
  :) contains a node R (the root tree) that has
  a) a binary left subtree — choose root (if any) connects to R
  b) a binary right subtree

```
public class LinkedTree {
    private class Node {
        private int key;
        private String data;
        private Node left;
        private Node right;
        ...
    }
    private Node root;
}
```



Example Binary Tree

Preorder Traversal

1) Visit Root, N
2) Recursively go to L
3) Recursively go to R

~~N - L - R~~

Result from Ex: 7 5 2 4 6 9 8

```
private void myPreorderPrint(node)
    Sout(root.key + " ");
    if(root.left != null)
        myPreorderPrint(node.left)
    if (root.left != null
        myPreorderPrint(node.right)
```

Postorder Traversal

1) Recursively go to L
2) Recursively go to R
3) Visit N

L R N

Results: 4 2 6 5 8 9 7

Same code almost, just move Sout to end of method

Inorder Traversal

L N R

Results: 2 4 5 6 7 8 9

Level-Order Traversal

- Aka "Breadth-First Traversal"
- Visit nodes by level, from top to bottom and left to right

Results 7 5 9 2 6 8 4

High-Level Implementation

Queue Root 7

(i) Remove 7
(ii) Insert left then right node

5 9

(iii) Remove 5
(iv) Insert 5's children

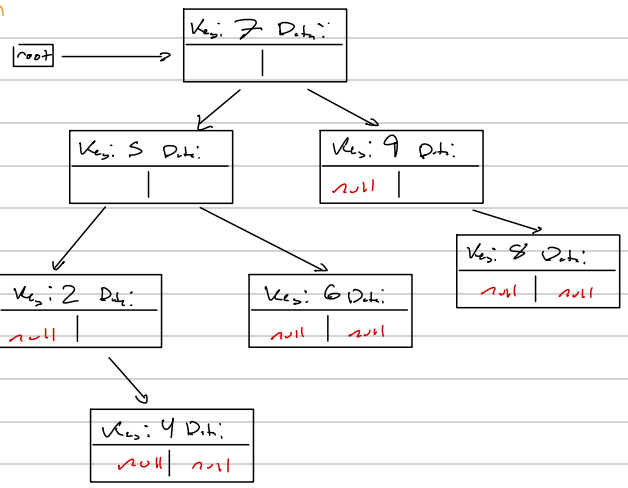
9 2 6

(v) Remove 9
(vi) Insert 9's children

2 6 8

Repeat until Queue is empty

Example Binary Tree



Tree Traversal Summary

- preorder: root, left subtree, right subtree
- postorder: left subtree, right subtree, root
- inorder: left subtree, root, right subtree
- level-order: top to bottom, left to right

Tree Species (Binary & Binary Search Trees)

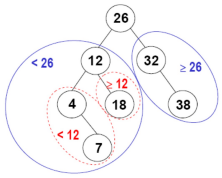
Binary Search Trees

→ For Each node K

(i) All nodes in K's left subtree are less than K

(ii) All nodes in K's right subtree are greater than or equal to K

→ Performing an Inorder Traversal of a Binary Search Tree returns nodes in ascending order.



Searching An Item in a BST

if K == root node's Key, done!

else: if K < root node's Key, search left

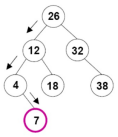
else search right subtree

Recursive Search Implementation

```
public class LinkedTree {
    ...
    private Node root;
    public String search(int key) {
        Node n = searchTree(root, key);
        return (n == null ? null : n.data);
    }
    private Node searchTree(Node root, int key) {
        //
        if (root == null)
            return null;
        else if (key == root.key)
            return root
        else if (key < root.key)
            return searchTree(root.left, key);
        else
            return searchTree(root.right, key);
    }
}
```

```
private class Node {
    private int key;
    private String data;
    private Node left;
    private Node right;
}
```

Base case



Iterative Search Implementation

```
public class LinkedTree {
    ...
    private Node root;
    public String search(int key) {
        Node n = searchTree(root, key);
        return (n == null ? null : n.data);
    }
    private Node searchTree(Node root, int key) {
```

```
Node trav = root; // Data pointer
While (trav != null) {
    if (trav.Key == Key)
        return trav
    else if (trav.Key < Key)
        trav = trav.left
    else
        trav = trav.right
}
return null;
}
```

Iterative Code Solution

```
public class LinkedTree {
    ...
    private Node root;
    public String search(int key) {
        Node n = searchTree(root, key);
        return (n == null ? null : n.data);
    }
    private Node searchTree(Node root, int key) {
        Node trav = root;
        while (trav != null) {
            if (key == trav.key)
                return trav;
            else if (key < root.key)
                trav = trav.left;
            else
                trav = trav.right;
        }
        return null;
    }
}
```

Binary Tree Item Insertion

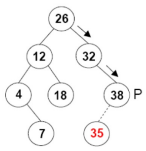
We want to insert an item whose key is *k*.

First, we find the node P that will be the parent of the new node:

- we traverse the tree as if we were searching for *k*, but we don't stop if we find it – we continue until we can't go any further

Next, we add the new node to the tree:
if *k* < P's key, make the node P's left child
else make the node P's right child

Special case: if the tree is empty, make the new node the root of the tree



Iterative Insertion

- Two References
  (i) trav = performs traversal down to point of insertion
  (ii) parent stays one above trav

Iterative Insertion Code

```
public void insert(int key, String data) {
    // Find the parent of the new node.
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        parent = trav;
        if (key < trav.key)
            trav = trav.left;
        else
            trav = trav.right;
    }
    // Insert the new node.

    if (parent == null)      // the tree was empty
        root = new Node(key,data);
    else if (key < parent.key)
        parent.left = new Node(key,data);
    else
        parent.right = new Node(key,data);
}
```

Node Deletion

Insertion / Deletion don't Guarantee Balanced Tree

Three Cases for deleting a Node

- Case 1: X has No Children
  - (i) Remove X from Tree by updating its parents reference to null
- Case 2: X has One Children (equivalent to saying pointer)
  - (i) Take the parents reference to X and set it to x's child
- Case 3: X has Two Children
  - (i) Find leftmost node in X's right subtree (smallest node) - call it Y
  - (ii) Copy Y's Key to X, but not its potential Child
  - (iii) Delete Y using Case 1 or Case 2.

Deletion Implementation

```
public String delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key)
            trav = trav.left;
        else
            trav = trav.right;
    }
    // Delete the node (if any) and return the removed item.
    if (trav == null) // no such key
        return null;
    else {
        String removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```

Try to find Node & its parent

Helper Method

Leaf — Single Child

Cases 1 & 2

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left == null || toDelete.right == null) {
        // Cases 1 and 2
        Node toDeleteChild = null;
        if (toDelete.left != null)
            toDeleteChild = toDelete.left;
        else
            toDeleteChild = toDelete.right;
        // both Cases are included. In case 1 toDeleteChild==null
        if (toDelete == root)
            root = toDeleteChild;
        else if (toDelete.key < parent.key)
            parent.left = toDeleteChild;
        else
            parent.right = toDeleteChild;
    } else { // case 3
        ...
    }
```

Case 3

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left == null || toDelete.right == null) { // case 1 and 2
        ...
    } else { // case 3
        // Get the smallest item in the right subtree.

        Node replacementParent = toDelete;
        Node replacement = toDelete.right;
        while (replacement.left != null) {
            replacementParent = replacement;
            replacement = replacement.left;
        }
        // Replace toDelete's key and data
        "toDelete.key = replacement.key;"
        "toDelete.data = replacement.data;"

        // Recursively delete the replacement item's old node.
        deleteNode(replacement, replacementParent);
    }
```

Finding In-Order Successor

Will be either Case 1 or 2

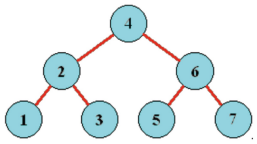
Efficiency of Binary Search Tree

- Search, Insert, Delete all have same time complexity
- Time Complexity for searching a binary tree
  - → Best Case: O(1)
  - → Worst Case: O(h)
  - → Average Case: O(h)

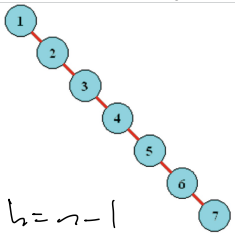
  } h is the height of tree

Height of Tree containing n items depends on balancing

- Insert and delete both involve path traversal from root → node w/ less than one child



h = log n



h = n - 1

AVL Trees

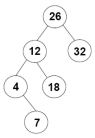
Height and Balance

- Height of a Tree: The length of the longest path from the root node to a leaf node
- Balance of a Tree: Given node N as the root:

$$\text{Balance}(N) = |\text{height}(N\text{'s right subtree}) - \text{height}(N\text{'s left subtree})|$$

↳ Balance of a leaf node is always 0

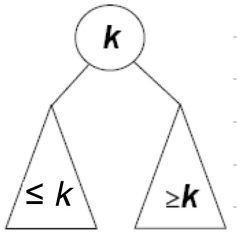
Ex: (i) balance(node 26) = |0 − 2| = 2
(ii) balance(node 4) = |0 − (−1)| = 1



One Leaf: Height = 0
Empty Tree: Height = −1

AVL Trees (Adelson-Velsky & Landis' 62)

Recall: Tree Invariants

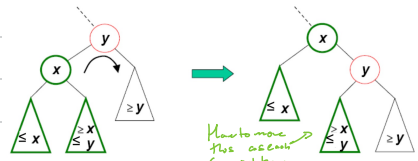


- The balance of all Nodes are never exceeding |1|, balance(N) ≤ |1|
  ↳ can either be −1, 0, or 1
- This rule must be fulfilled after every insertion & deletion
- Steps taken to restore balance must
  (i) Maintain the search-tree invariant
  (ii) Have a worst-case time complexity of O(log n)

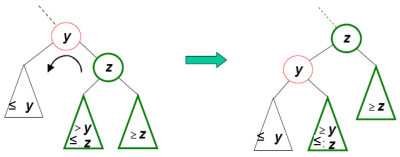
Searching is log(n), and both insertion & deletion need to search first → also log(n)

Ex: Empty Tree —Add 1, 2→ —Add 3→ —Rebalance Analysis→ ① violates balance, ②, ③ are ok —Rotate ① To left→

Rotation Operations

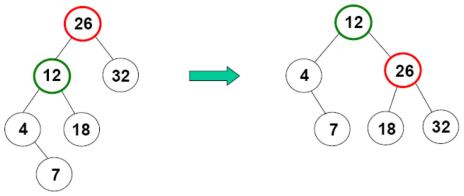


Right Rotation on (around) y

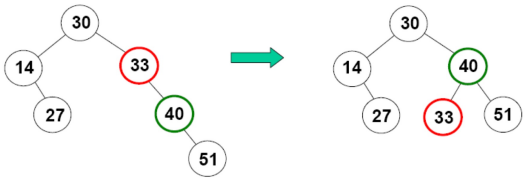


Left Rotation on y

Example Rotations



Right Rotation on Node 26



Left Rotation on node 33

Implementation of AVL Trees

- Node Class is similar but you need to keep track of the balance of each node!

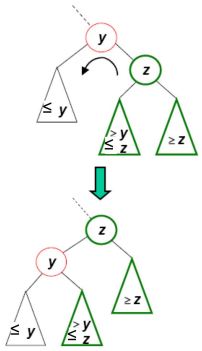
```
public class AVLTree {
    private class Node {
        private int key;
        private String data;
        private Node left;  // reference to left child
        private Node right;         // reference to right child
        private Node parent;        // reference to parent node
        private int balance;        // balance value of the node
        ...
    }
    private Node root;
    ...
}
```

Rotation Implementation

- Just involves change pointers

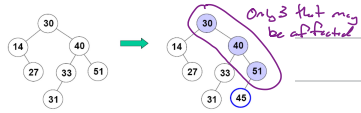
# ptr updates	• Involves changing child/parent vars:
1	(i) Right child of parent of y
2	(ii) Right Child and parent of y
2	(iii) Left child and parent of z
1	(iv) Parent of left child of z

6 Total Ptrs → Constant Time Complexity



Insertion into AVL Trees

- Remember 2 new fields: Ref to nodes parent & its balance
- An insertion can only affect the height values and balance values in the new node's ancestors
- First step is inserting node as in a binary search tree.



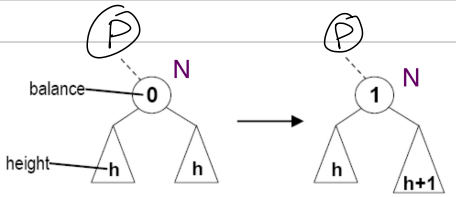
Change in Balance

Assume node N to be affected by insertion

Case 1 N's balance goes from 0 to 1

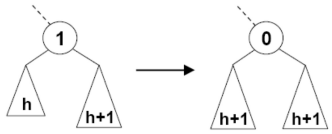
- N's subtree still within AVL rules
- Height of N has increased
  - ↳ Balance of N's parent will change
- Need to check if N's parent satisfies the AVL rule
  - ↳ Balance of other ancestors may also be affected

Need to check P (N's parent)



Case 2 N's balance goes from 1 to 0

- Height of the subtree of which N is the root has not changed
  - ↳ Don't need to look at ancestors!



Case 3 N's balance goes from 1 to 2

- Need to rebalance tree using rotations
- Rotations will restore the height that rotated subtree had prior to rotation
  - ↳ N's ancestors balances won't change

→ **Case 3a** Inside vs. Outside Insertion on left subtree Left Subtree of Left Child

Inside: Closer to center line

(y balance to −2)

Outside: Further from center line

- Perform Single right rotation about y

→ **Case 3b** y balance to +2, is symmetric to 3b Right subtree of Right Child

- Perform single left rotation about y

→ **Case 3c:** we've added a node to the right subtree of y's left child, x, bringing y's balance to -2.

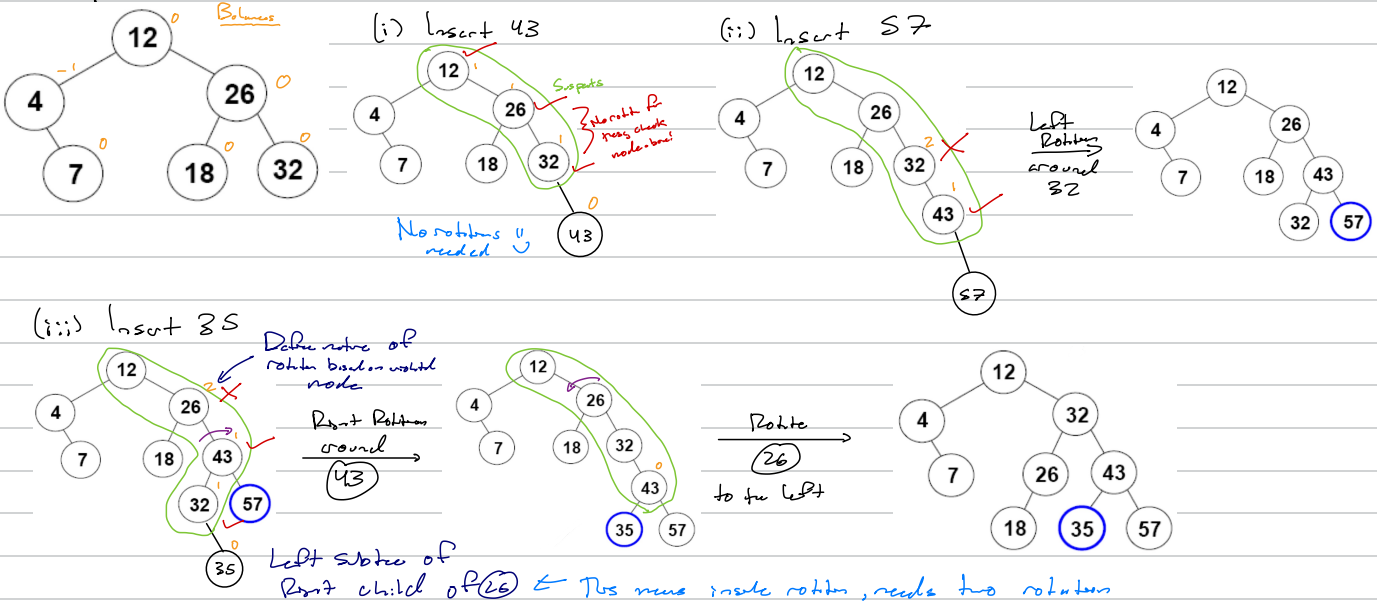
→ **Case 3d**: we've added a node to the left subtree of y's right child, x, bringing y's balance to +2. (*symmetric to case 3c; can you draw pictures to show how it works?*)

Complete Insertion Method

- Insert the new node N as in a binary tree
- Use parent references to follow the path from N back to the root
  - If an ancestor's balance was 0, it will now be ±1 (case 1) → continue up the path to the root
  - If an ancestor's balance was ±1, there are now two cases:
    (i) It is now 0 → stop
    (ii) It is now ±2 → perform 1 to 2 rotations to rebalance tree (cases 3a-3d)
    ↳ number of rotations depends on where N is inserted

☆ In either case, stop – there is no need to go any further up the tree

Examples of Rotations



AVL Tree Deletion

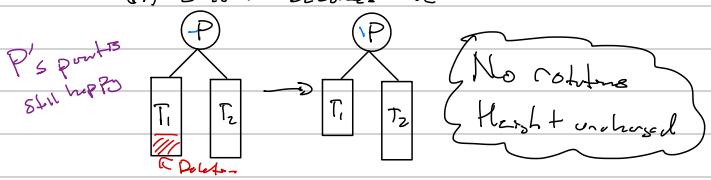
Deletion of AVL Tree Nodes is complicated

Implementation: - shorter flag used to indicate if a subtree was shortened
- each node has a balance factor
(i) left-high: height of left subtree exceeds right subtree
(ii) right-high: height of right subtree exceeds left subtree
(iii) equal: height of left and right subtrees equal

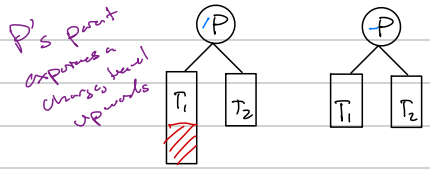
Algorithm: (i) shorter initialized as true
(ii) Starting from the deleted node back to the root, take an action based on
- value of shorter
- balance factor of current node
- sometimes balance factor of a child of the current node
(iii) Until shorter flag becomes false or you reach root

After standard removal from a binary tree

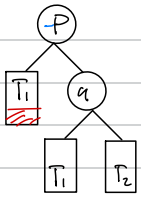
Case 1: The balance factor of p is equal
(i) Change the balance factor of p to right- or left- high
(ii) Shorter becomes false



Case 2: The balance factor of p is not equal & taller subtree is shortened
(i) Changes to balance factor of p to equal
(ii) Shorter remains true



Case 3: The balance factor of p is not equal & shorter subtree is shortened



→Case 3a: The balance factor of q is equal
(i) Apply a single rotation
(ii) Change the balance factor of q to left- or right- high
(iii) Shorter becomes false

→Case 3b: Balance factor of q is same as p
(i) Apply single rotation
(ii) Change the balance factors of p & q to equal
(iii) Shorter remains true.

→Case 3c: Balance factor of q is opposite of p
(i) Apply a double rotation
(ii) Change balance factor of new root to equal
(iii) Also change the balance factors of p & q
(iv) Shorter remains true

Efficiency of AVL tree node deletion
$O(\log_2 n)$ → worst case requires full path up and down with rotation constant time operations

Other Balanced Trees

AVL Trees Aren't Perfect

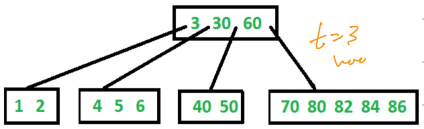
- Not memory friendly :(
  - ↳ Compress the tree to a non-binary tree to reduce number of traversals
- Have more keys per node reduce time spent looking at unneeded data

Why not store multiple keys at each node

B-Tree

- Reduces the number of disk accesses
- All leafs are @ the same level
- Defined by its minimum degree 't' ← Input Parameter
- Every node except root must contain at least t-1 keys
  - ↳ Root can contain minimum 1 key
- AVL nodes (including root) must contain a maximum of 2t-1 keys
- Number of children of a node is equal to number of keys in it plus 1

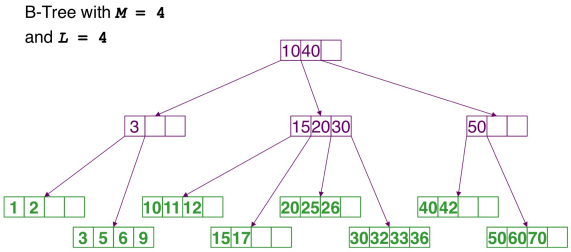
Example B-tree



Alternate Definition

- B-Trees are specialized *M*-ary search trees
- Each node has many keys
- maximum branching factor of M
- the root has between 2 and M children *or* at most L keys
- other internal nodes have between $\lceil M/2 \rceil$ and M children
- internal nodes contain only *search* keys (no data)
- each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys
- all leaves are at the same depth

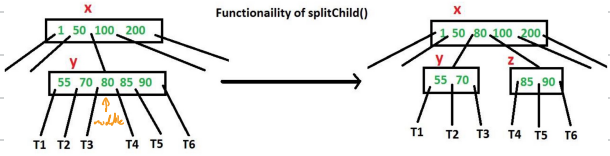
Example



Everything is logarithmic in time

Insertion into B-trees

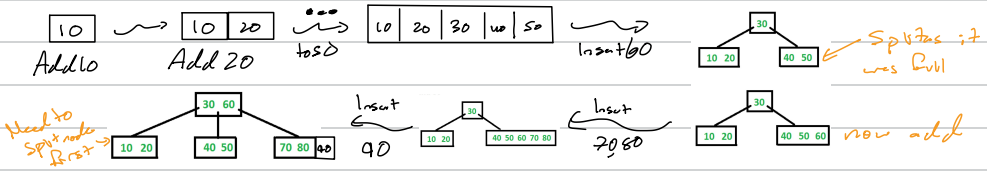
- Nodes w/ Maximum Number of Nodes are a problem with insertion
- New node is always inserted @ leaf node
- Before inserting a key to node, make sure the node has extra space
- If a node is full, use splitChild() to split a child of a node



Example

- tree of minimum degree t=3
- insert integers 10, 20, 30, 40, 50, 60, 70, 80 and 90

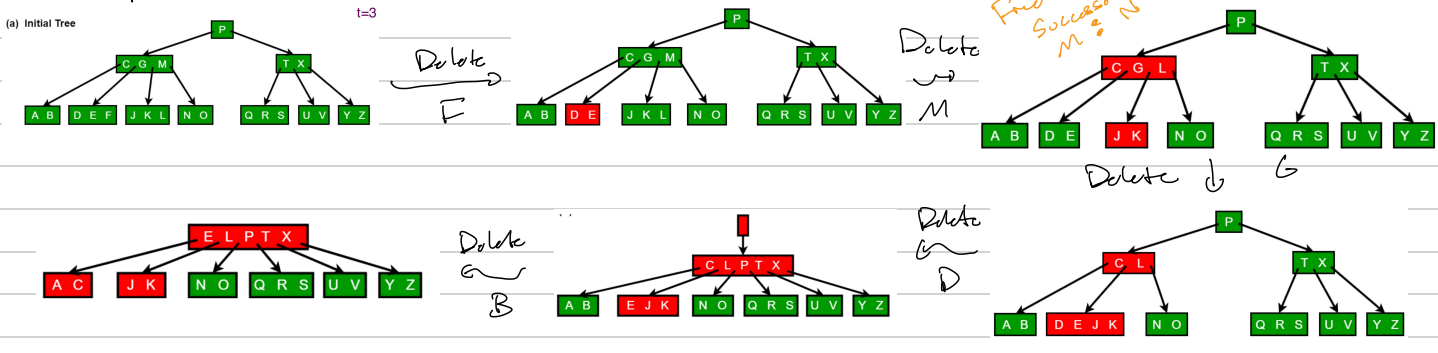
$t=3 \in \left(\begin{matrix} min=2 \\ max=5 \end{matrix}\right)$



Deletion from B-trees

- Nodes with minimum node is problematic
- Back up if a node (other than the root) along the path to where the key is to be deleted has has the minimum number of keys

Example



Final Exam Notes

Heaps & Priority Queues

Queues

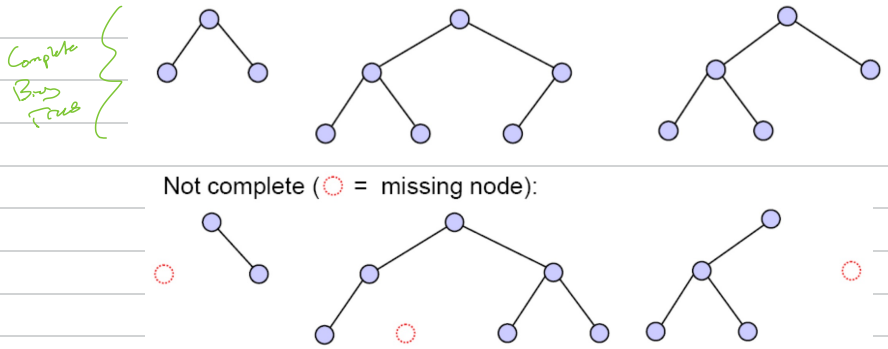
- Items added to rear and removed from front
  - ↳ FIFO (only can access element at head)
- Operations: insert, remove, peek, isEmpty/Full

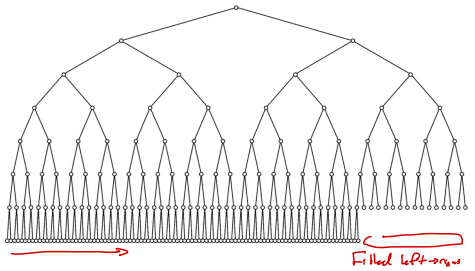
Priority Queues

- Collection of items with each having an associated priority
- Operations: insert, remove → item with highest priority
- Uses a Heap!

Structure types of Binary Trees

- **Full Binary tree:** Every node has exactly two or zero children
- **Perfect Binary tree:** Full tree with all leafs at the same depths
- **Complete Binary tree:** Balanced, and at the same level, filled left → right



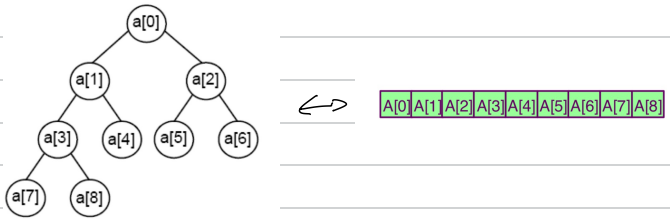


A huge Complete Binary Tree

Array Representation of a Complete Binary Tree

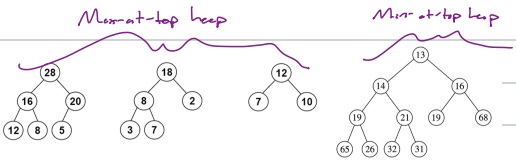
- Nodes of the tree are stored in the order in which they would be visited by a level-order traversal
- The root Node is a[0]
  - ↳ Given the node a[i]:
    - (i) left child is a[2*i+1]
    - (ii) right child is a[2*i+2]
    - (iii) parent is a[(i-1)/2], for i != 0

Can construct array from tree and tree from array. (Stored in level-order and gives complete tree)



Heaps for Priority Queues

- Heaps are a complete binary tree
  - ↳ Constraint: A node's key >= keys of children (if any)
- Largest value is always @ tree root
- Smallest value can be in any leaf node (no guarantee)
- For min-at-top heaps, every interior node is less than or equal to its children (not a big focus in this course)
- Uses Generics, so (item1 == item2) compares memory locations not actual objects!! ← leads to unexpected errors
  - ↳ Have to use Comparable



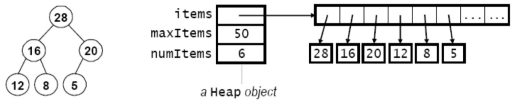

```
class Classname implements Comparable<Classname>
```

- Many built-in classes have a compareTo() method:

```
public int compareTo(Classname other)
```

- Example on Point: the object in the Heap is required to extend Comparable, so it must have compareTo()
- T's class has implements Comparable<T>

```
public class Heap<T extends Comparable<T>>{
    private T[] items;
    private int maxItems;
    private int numItems;
    public Heap(int maxSize) {
        items = (T[])new Comparable[maxSize];
        maxItems = maxSize;
        numItems = 0;
    }
    ...
}
```



Item Removal from a Heap

- [28, 16, 20, 12, 8, 5] Heap remove 28
  [5, 16, 20, 12, 8] Move 5 to front (now it's complete but not a heap)
  ↳ Then swap 5 with its children until its greater than all of them

Array based implementation

"Sift Down" Operation

Remove and return the item in the root node.

- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children

- Removing an Item (the Greatest)

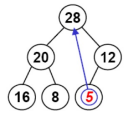
Remove and return the item in the root node.

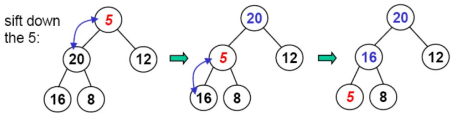
- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children

Method:

- make a copy of the largest item
- move the last item in the heap to the root
- "sift down" the new root item until it is >= its children (or it's a leaf)
- return the largest item

"sift": items are filtered such that small ones will fall





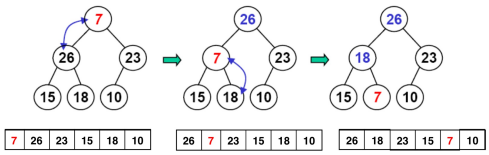
Implementation of SiftDown Method

This is just a helper method for removal

```
private void siftDown(int i) { // Input: the node to sift
    T toSift = items[i];
    int parent = i;
    int child = 2 * parent + 1; // Child to compare with; start with left child
    while (child < numItems) {
        // If the right child is bigger than the left one, use the right child instead.
        if (child +1 < numItems  && // if the right child exists
                items[child].compareTo(items[child + 1]) < 0)  // ... and is bigger than the left child
            child = child + 1; // take the right child
        if (toSift.compareTo(items[child]) >= 0)
            break; // we're done
        // Sift down one level in the tree.
        items[parent] = items[child];
        items[child] = toSift;
        parent = child;
        child = 2 * parent + 1;
    }
    items[parent] = toSift;
}
```

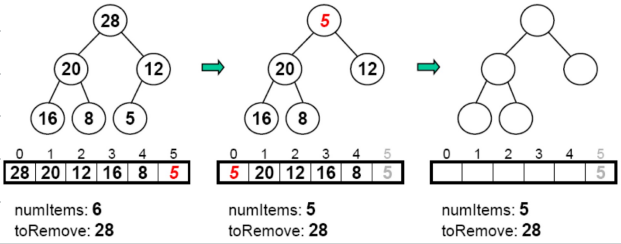
We don't have to put sifted item in place of child. We can wait until the end to put the sifted item in place.

Example Sifts



- Once you have siftDown method, removal is trivial

```
public T removeMax() {
    T toRemove = items[0];
    items[0] = items[numItems-1];
    numItems--;
    siftDown(0);
    return toRemove;
}
```

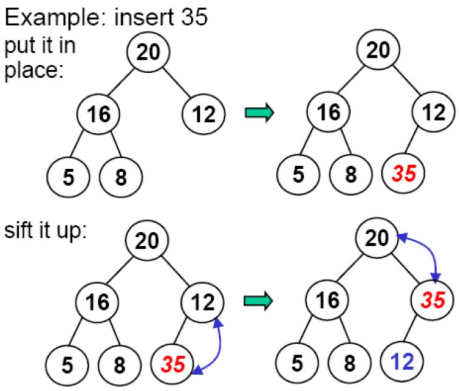


Insertion into Heaps

- Assume there's additional space in the array...
- Insert to end of array, then sift up

Algorithm: & Quick Example

- put the item in the next available slot (grow array if needed)
- "sift up" the new item until it is <=its parent (or becomes the root)




```
public void insert(T item) {
    if (numItems == maxItems) {
        // code to grow the array goes here...
    }
    items[numItems] = item;
    numItems++;
    siftUp(numItems-1);
}
```

Need to consider what happens if array is full

Running Time Analysis

- SiftUp & SiftDown
  → h = log N
  → O(log₂N) in worst case
- Insert/Remove are both O(log N)
- Search is slow O(N log N)

Building a Heap

A Naive Algorithm

- Take N items from any and build a heap
- For each item, insert it into a heap which is initially empty

```
public void buildHeap(T[ ] array, int size )
{
    < initialize the heap here ...>
    for( int i = 0; i < size; i++ )
        insert(array[i]);
}
```

O(NlogN)

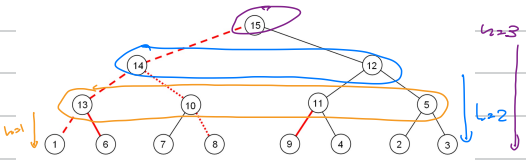
An Efficient Algorithm: Building the Heap in Place

1) Position = (numItems - 2)/2;
2) SiftDown item at that Position
3) Decrement Position by 1
4) Repeat Steps 2, 3, 4 until position is 0

```
public void buildHeap() {
    for (int i = (numItems-2)/2; i >= 0; i--)
        siftDown(i)
}
```

O(N/2 logN) → still O(NlogN)

Ought to happen for about half of the nodes



Complexity is determined by the total height of all the nodes in the tree

Proof

For a perfect binary tree of height h, $N = 2^{h+1}-1$:

1 node at height h
2 nodes at height h-1
4 nodes at height h-2
...
2^{h-1} nodes at height 1
2^h nodes at height 0

Total height of all nodes $= h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}(h-(h-1)) = S$

$2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^h(h-(h-1)) = 2S$

$S = 2h + 4h - 4 + 8h - 16 + 16h - 48 + \ldots + 2^h\left(h-(h-1)\right)$

$\quad - \left(h + 2h - 2 + 4h - 8 + 8h - 24 + \ldots + 2^{h-1}h - 2^{h-1}(h-1)\right)$

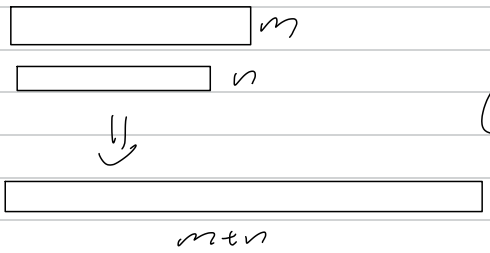
Although a complete tree is not a perfect binary tree, but number of nodes in a complete tree of height h is:

$2^h \le N_{actual} \le 2^{h+1} - 1$

Thus the actual number of nodes is within a factor of 2 of N. Hence, $S = O(N_{actual})$

Builds a heap from an array of random integers in the order of O(N)! It is Linear!!

Merging Two Heaps



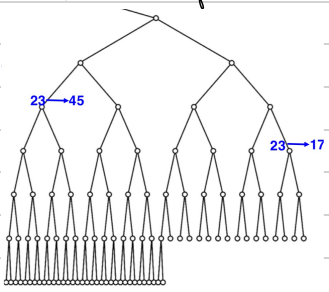
If given an unknown, items be used in by-O

O(m+n)
Linear

Runtime Complexity

findMax	O(1)
removeMax()	O(logN)
insert()	O(logN)
buildHeap()	O(N)
merge()	O(N)

Exercise: Update an Item




```
public T update(int i, T item) {
    T oldItem = items[i];
    if (item.compareTo(oldItem) == -1)
        siftUp(i);
    else
        siftDown(i);
    return oldItem;
}
```

Run Time:

Delete An Arbitrary Item

```
public T delete(int i) {
    T toDelete = items[i];
    items[i] = item[numItems - 1];
    numItems--;
    if (toDelete.compareTo(items[i]) == -1)
        siftUp(i);
    else
        siftDown(i);
    return toDelete;
}
```

?

Applications of Heaps

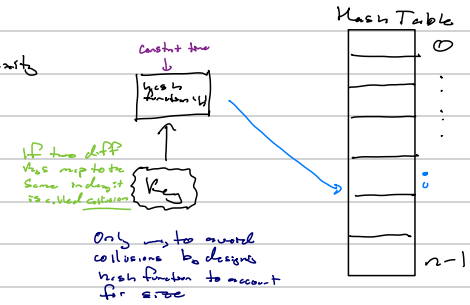
Application #1: Heap Sort

- Sorting Algorithms: Given an arbitrary array, rearrange the items in the array in desired order
- Naive Algorithm: $O(n^2)$ as it performs a full linear scan to find next item. $O(n)$ steps/scan
  - ↳ Called Selection Sort
- Sort Array with a Heap
  1) Turn Array into Heap (Max-at-Top)
  2) Remove Max element and move to back of array
  3) Repeat until all elements exhausted

Heap is nice as finding max is constant for max-at-top

Hashing

Downsides of Previous Data Structures: Log n was the best time complexity



Hash Tables

- Allows us to access, delete, and insert items in sub-logarithmic time ← linear
- Hash function is also constant time
- Collisions are a result of hashing, not completely unavoidable, but you can mitigate it

Example: Want to store the grades of students in this class ← In real world keys are semi-big not always ints

→ give each student a unique key (integer from 0-150)

→ Store student records in array and can then perform constant time operations } Downside is students take many classes and keys should be constant access time

Solutions: (i) Use SSN to organize students? No! Too much storage as 10^9 total SSNs

↳ Also need an additional data structure to map Names to SSNs !!

(ii) Use a hash function

- Converted "map"'s keys into array indexes
  - → Domain: the keys
  - → Range: integers in $[0, size]$
- For this example, we might say $Index = SSN \bmod 150$
  - → Greatly reduces size complexity
  - → Problem: multiple keys can map to the same index
    - ☐ Called collisions: 150-00-0001 and 150-00-0151 both have keys of 1

Hash Functions

Example 1: Salary used as Key

- 10-worker shop
- Initially used hash table with (Salary mod 10K) hash function
  - → What if salaries are all multiples of 10K? Bad as all people will map to same salary
  - → What if salaries are multiples of 2K? Better function

Example 2: String as Key

→ Keys = character strings of lower-case letters

→ Hash function:

☐ $h(Key) = (\text{the byte sum of all characters}) \bmod (\text{table-size})$

↳ $h(\text{"cat"}) = (\text{'c'} + \text{'a'} + \text{'t'}) \bmod (\text{table-size})$

☐ Might limit character length to 20 to reduce size requirement

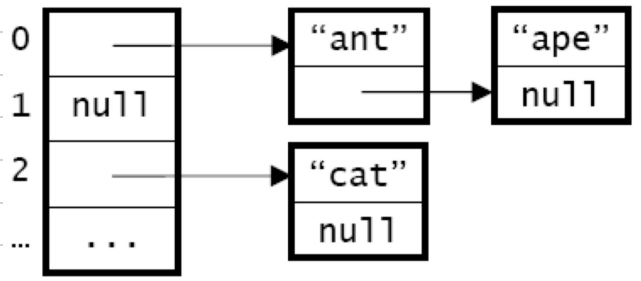
↳ Maximum sum is gonna be 127 * 20 = 2540 (127 is max byte of a lower character)

Requirements for Good Hash Functions

- Full table size utilization
- Even ("uniform") key mapping throughout the table
  - → Utilizing known key distribution in the objects
  - → Make hash function "random"

Hashing Collisions ← We will assume that clients know we are using specific designs

- Can use chaining, list based solution so collisions just point to linked list index



- Can also use Linear Probing to look for open positions
  - → Advantage: If there is an open position, linear probing will eventually find it
  - → "Clusters" of occupied positions develop
    - ▷ Increases the length of subsequent probes
    - ▷ As load factor increases, both search and insert times look increasingly linear
- Can try to avoid Linear probing disadvantages with Quadratic Probing
  - → Advantage: reduces clustering
  - → Disadvantage: There is a chance to miss an empty open position, fail to insert into table

Theorem: If the load factor is less than 50% and the table size is prime, a new item can always be inserted with Quadratic Probing

Load Factor: The ratio of open array positions to total possible positions

- Double Hashing for addresses is an option
  - → h_1 computes hash code
  - → h_2 computes the increment for probing
  - ✱ Sequence: h_1, $h_1 + h_2$, $h_1 + 2 * h_2$ ...

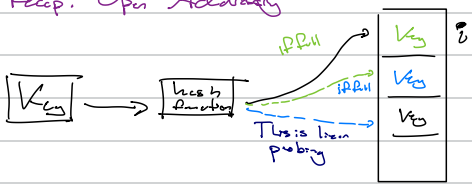
reduces clustering
Theorem: will find an open position if there is one, provided the table size is a prime number.
Disadvantage: the need for two hash functions

Removing Items with Open Addressing

- If the item is not in the table, you have to search the whole table (linear)
  - ↳ To avoid this, we use a 'removed' flag (boolean) allows us to avoid linear
- Use 3 states: occupied, empty, removed

Hashing Implementation

Recap: Open Addressing



If we remove Key 3 there is an open spot at its index but 'Key' is still there so we need to continue searching. Use flag

Implementation

```
public class HashTable {
    private class Entry {
        private String key;
        private String etymology;
        private boolean removed;
    }
    private Entry[] table;
    private int tableSize;
    ...
}
```

flag →

- For an empty position, table[i] is null
- For a removed position, table[i] will refer to an Entry object whose removed field is true
- Data is not actually removed just simply overwritten.
- For an occupied position, table[i] has removed = false
- Available positions are either empty or removed

Constructors

```
public HashTable(int size) {
    table = new Entry[size];
    tableSize = size;
}
```

Defaults all indexes to null

```
private Entry(String key, String etymology) {
    this.key = key;
    this.etymology = etymology;
    removed = false;
}
```

Re-hashing Constructor

Finding an Open Position

Helper method to use double hashing for open position searching

```
private int probe(String key) {
    int i = h1(key); // first hash function
    int j = h2(key); // second hash function
    int iterations = 0;
    // keep probing while the current position is occupied (non-empty and non-removed)
    while ( table[i] != null && table[i].removed==false ) {
        iterations++
        i = (i + j) % tableSize;   → Update probe position
        if(iterations >= tableSize) return -1;
    } return i;
}
```

→ This can run forever, we need to limit number of iterations to the size of the table

Infinite Loop

for double hashing:
- (h1 + n*h2) % n = h1 % n
- (h1 + (n+1)*h2) % n = (h1 + n*h2 + h2) % n = (h1 + h2)%n
- (h1 + (n+2)*h2) % n = (h1 + n*h2 + 2*h2) % n = (h1 + 2*h2)%n
- …

for quadratic probing:
- (h1 + n²) % n = h1 % n
- (h1 + (n+1)²) % n = (h1 + n² + 2n + 1) % n = (h1 + 1)%n
- (h1 + (n+2)²) % n = (h1 + n² + 4n + 4) % n = (h1 + 4)%n
- …

insert() method

```
public void insert(String key, String etymology) {
    int i = probe(key);
    if (i == -1)
        throw new RuntimeException("HashTable full");
    else
        ? Insert w/ helper methods @ index i
}
```

Finding the Position of A Key

→ Different from probe() as it returns position of the key, not an available position

```
private int findKey(String key) {
    int i = h1(key); // first hash function
    int j = h2(key); // second hash function
    int iterations = 0;

    // keep probing while the entry is not empty
    while ( table[i] != null ){
        // return if key is found, otherwise continue
        if (table[i].removed==false && table[i].key.equals(key))
            return i;

        i = (i + j) % tableSize;
        iterations++;
        if (iterations >= tableSize) return -1;
    }

    return -1;
}
```

Goes until null is found meaning nothing can exist past it for linear probing → Different for Quadratic probing

Making use of findKey()

```
public String search(String key) {
    int i = findKey(key);
    if (i == -1)
        return null;
    else
        return table[i].etymology ;
}
```

Both these findKey() and check its output

```
public void remove(String key) {
    int i = findKey(key);
    if (i == -1)
        return;
    table[i].removed = true;
}
```

Hashing Implementation Issues

Potential Issues

1) Lots of items: $\lambda \geq 1$ ← Can copy to larger table and copy all non-null & non-removed entries
   - you should increase size too
   - λ (load factor) = $\frac{\#\text{ of entries}}{\text{size of table}}$
2) Lots of deletions: small λ
   a) Insertion should be very fast, as many positions have removed=true
   b) Search and remove will be affected negatively

Resizing Structures

Rehash() method (Expanding)

```
public void rehash( ) {
    int oldSize = tableSize;
    Entry[ ] oldTable = table;

    tableSize = nextPrime(2 * oldSize);
    table = new Entry[tableSize];
    for(i = 0; i < oldSize; i++)
        if ( ? )
            ?
}
```

Efficiency of Hash Tables

- Best case: Search/insert are O(1)
- Open addressing: worst case is O(n): n = size of table
- Separate chaining: worst case is O(n): n = number of keys
- Performance worsens as load factor increases
  - $\lambda < \frac{1}{2}$ for open addressing is ideal
  - $\lambda < 1$ for separate chaining is ideal
- Time-space tradeoff: Takes up more space for larger tables

Limitations of Hash Tables

- Items are not ordered by key so it's hard to get kth-largest element or sorted array of elements
- Difficult to create good hash functions

Applications of Hash Functions

Requirements for good hash functions

(i) Full table size utilization
(ii) Even key mapping throughout the table
(iii) Should be 'random'

Hash functions for Strings

- $h_b = (a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b^1 + a_{n-1}) \bmod (\text{tableSize})$
  - a_i is encoding of the i-th character,
  - b is a constant → Design parameter. Java uses b=31
- All characters contribute
- Contribution of a character depends on its position

- Tons of multiplications as string length ↑

Use Horner's Method for efficiency

example: $101*31^2 + 97*31 + 116 = (101*31 + 97)*31 + 116$

$101*31^3 + 97*31^2 + 116*31 + 110 = ((101*31 + 97)*31 + 116)*31 + 110$

$a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b^1 + a_{n-1} = (\ldots((a_0 b + a_1) b + a_2) b + \ldots + a_{n-2}) b + a_{n-1}$

```
int hash = s.charAt(0);
for (int i = 1; i < s.length(); i++)
    hash = hash * b + s.charAt(i);
```

well-known & widely adopted function

Substring Pattern Matching

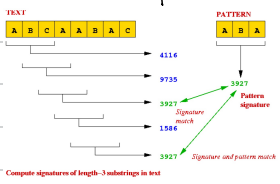
Input: A text string t and pattern string p
Problem: Does t contain p? If so, where?

Brute Force: Substring contains n letters, and total text is length m
O(mn) complexity

Via Hashing: Compare hash of p to hash of groups of p.length() in t.
O(m) Complexity if implemented correctly

Can also be used for genome matching

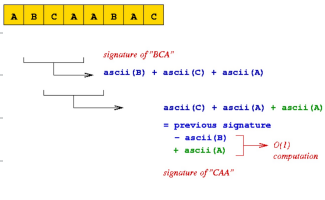
Poor Implementation



- Compute the "signature" of the pattern.
- Compute the "signature" of each substring of the text.
- Scan text until signature matches => potential match, so perform string comparison

Takes O(mn) still as O(n) required to get each substring of the signature (hash)

Better Implementation



Observation: two successive substrings differ by only two characters.

=> next signature can be computed quickly (O(1)).

Use signature of old substring to get new one. Reduces signature generation to O(1)

Simple Sorting Algorithms

- Array → increased order
- We want to do everything in place
  - ↳ minimize storage complexity
- minimize total # of comparisons (C) and moves (M)

Method 1: Selection Sort

- Consider positions from left → right
- For each position, select the element that belongs there and put it in place by swapping it with the element that is currently there

Implementation

```
static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length – 1; i++) {
        int j = findMin(arr, i);
        swap(arr, i, j);
    }
}
```

Helper method

```
static int findMin(int[] arr, int lower) {
    int curIndexMin = lower;
    for (int i = lower+1; i <= arr.length – 1; i++)
        if (arr[i] < arr[curIndexMin])
            curIndexMin = i;
    return curIndexMin;
}
```

0	1	2	3	4	5	6
2	4	7	21	25	10	17

stops & returns position

$a_0 = 1$, $a_1 = 4$ → $t = a_0$, $a_0 = a_1$, $a_1 = t$ ← temp variable ok

Time Complexity

$O(n^2)$

- Swaps always consistent
- Checks $n, n-1, n-2 \ldots$

Method 2: Insertion Sort

- Going from left to right, "insert" each element into its proper place with respect to the elements to its left "sliding over" other elements to make room.

0	1	2	3	4
15	4	2	12	6
4	15	2	12	6
2	4	15	12	6
2	4	12	15	6
2	4	6	12	15

```
static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int toInsert = arr[i];
        for (j = i; j > 0 && toInsert < arr[j-1]; j--)
            arr[j] = arr[j-1];
        arr[j] = toInsert;
    }
}
```

Note beginning

Shift the element to the left

Time Complexity

n	C	M
1	1	1
2	2	2
3	3	3
4	4	4

for | 5 | 4 | 3 | 2 | 1 |

$O(n^2)$ Still !!

Best Case >> Best Case Selection

Worst Case << Worst Case Selection

Shell Sort

- improves insertion sort, takes advantage of insertion sort being fast for almost sorted arrays

Sorting Subarrays
- use insertion sort on interleaved subarrays that contain elements separated by some increment
- larger increments allow the data items to make quicker "jumps"
- repeatedly using a decreasing sequence of increments

Example for an initial increment of 3 (3 subarrays)

0	1	2	3	4	5	6	7	8
99	8	13	2	15	9	4	12	24

Sort the subarrays using insertion sort to get the following:

0	1	2	3	4	5	6	7	8
2	8	9	4	12	13	99	15	24

Finally, we complete the process using an increment of 1.

```
static void shellSort(int[] arr) {  // Using Hibbard's sequence
    int incr = 1;
    while (2 * incr <= arr.length) incr = 2 * incr;  // starting incr = floor(log arr.length)
    incr = incr - 1;

    while (incr >= 1) {
        for (int i = incr; i < arr.length; i++) {
            int toInsert = arr[i];
            for(j = i; j > incr-1 && toInsert < arr[j-incr]; j = j - incr)
                arr[j] = arr[j-incr];
            arr[j] = toInsert;
        }
        incr = incr/2;
    }
}
```

incr used in last for loop

Depends on the sequence of decreasing increments
- Should decrease fast to lower the number of passes of insertion sort
- Should not decrease too fast or will be close to (one-pass) insertion sort

Hibbard's sequence has worst-cast running time at $O(n^{3/2})$; similar to

(Case Alum) Knuth's sequence: … 1093, 364, 121, 40, 13, 4, 1
    incr = incr/3;

Sedgewick's sequence: … 109, 41, 19, 5, 1
- interleave $9*4^i - 9*2^i + 1$ and $4^i - 3*2^i + 1$
- worst-case running time $O(n^{4/3})$

The running time is often difficult to analyze, and some bounds are unknown.
- The average-case running time for Hibbard or Sedgewick sequences? -- Unknown!
- What is the best sequence of increments? – Unknown!

Still worse than $O(n \log n)$ as $O(n^2) < O(\log n)$ for all c. Can combine $n \log n$ with heap sort, so can we better?

Recap

Method 1: Selection Sort
- For each position of an array, find the element for it
- $O(n^2)$ comparisons, $O(n)$ moves

0	1	2	3	4	5	6
2	4	7	21	25	10	17

Method 2: Insertion Sort
- For each element, find a position to insert it
- $O(n^2)$ comparisons, $O(n^2)$ moves
- O(n) if the array is sorted or nearly sorted

0	1	2	3	4
6	14	19	9	_

Method 3: Shell Sort
- Based on the observation that insertion sort requires O(n) running time for sorted or nearly sorted array ($O(n)$, $O(n^2)$)
- Generalization of insertion sort with larger "jumps", using strides larger than 1 (for insertion sort, the stride = 1)
- Use a decreasing sequence of strides

0	1	2	3	4	5	6	7	8
99	8	13	2	15	9	4	12	24

Bubble & Quick Sort

Bubble Sort

0	1	2	3
28	24	27	18
24	27	18	28
24	18	27	28
18	24	27	28

- Perform a sequence of passes through the array
- On each pass, proceed from left → right and swap adjacent elements if they are out of order
- At the end of the k^{th} pass, the k rightmost elements are in their final positions
  - ↳ Unsorted region propagates from the right side of the array

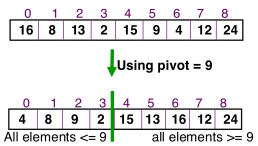
Implementation

C	M
n-1	3(n-1)
n-2	3(n-2)
⋮	⋮
1	3(1)
$O(n^2)$	

```
static void bubbleSort(int[] arr) {
    for (int i = arr.length – 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j+1])
                swap(arr, j, j+1);
        }
    }
}
```

- Inner loop performs a single pass repeated for an amount of times determined by the outer loop
- Time complexity is always $O(n^2)$ even if input is sorted already.
  - ↳ no data moves obviously
- We can improve the time complexity that changes depending on if a swap is made, this flag would cause a break if no swaps are made.
  - ↳ Best case becomes $O(n)$ if input is sorted

Quick Sort



- Like bubble sort, uses an approach based on exchanging out of place elements but more efficiently
- Works by dividing array into small partitions and sorting multiple subarrays recursively
- Divide & Conquer method:
  - → Divide: rearrange elements so we end up with two subarrays such that each element in the left array is less than or equal to all of those in the right array
  - → Conquer: apply quicksort recursively to the subarrays, stopping when a subarray has only one element.
  - → combine: nothing needs to be done because of criteria used to create subarrays

- Left subarray: all values ≤ pivot
- Right subarray: all values ≥ pivot

Subarrays don't have to be the same size

Example of Partitioning an Array

Pivot = middle element

	first							last
arr →	7	15	4	9	6	18	9	12

- Maintain indices i and j, starting them "outside" the array:

i = first - 1
j = last + 1

i →								← j
7	15	4	9	6	18	9	12	

Another Example

Choose pivot=13

i →								← j
24	5	2	13	18	4	20	19	

Advance i and j

	i →				← j			
24	5	2	13	18	4	20	19	

Swap 4 and 24

	i →				← j			
4	5	2	13	18	24	20	19	

i and j meet

		i → ← j					
4	5	2	13	18	24	20	19

Return j and we have two subarrays: arr[first : j] and arr[j+1 : last]

		i, j					
4	5	2	13	18	24	20	19

- ~~i and j move simultaneously~~
- (i) 2nd iteration: i=15, j=9 both wrong so swap
- (ii) Only swap when both pointers are incorrect
- (iii) Last iteration i=9, j=6 both wrong → swap
- (iv) Stops when i and j cross each other

A new pivot should be selected every recursive call

Some times subarrays are not equal in length

i →									← j
0	1	2	3	4	5	6	7	8	
16	8	13	2	9	15	4	12	24	

↳

	i			j				
0	1	2	3	4	5	6	7	8
4	2	13	8	9	15	16	12	24

Implementation

```
static void quickSort(int[] arr) {
    myQuickSort(arr, 0, arr.length – 1);
}

static void myQuickSort(int[] arr, int first, int last) {
    if (first >= last) return;
    int split = partition(arr, first, last);

    ?

}
```

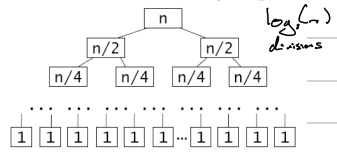
quickSort() is provided by Sort calls when calls a recursive method myQuickSort() stopping when subarray length = 1.

Partition Helper Method

```
static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;      // index going from left to right
    int j = last + 1;       // index going from right to left
    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j)
            swap(arr, i, j);
        else
            return j;  // arr[j] = end of left array
    }
}
```

Quick Sort Complexity

Recursive call Tree



$\log(n)$ divisions

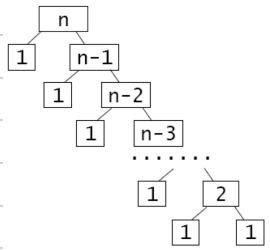
$n = 2^k \Rightarrow k = \log(n)$

For Best Case

- Each elements compared with pivot within a single partition so n comparisons
- n comparisons at each level of the tree

so $C(n) = M(n) = O(n \log n)$

For Worst Case



- Occurs only if pivot is poor
- Recursive height is n, so on the order of $O(n^2)$
- This is still very hard to do as pivot is usually always correct
- Have to be very consistently unlucky to get $O(n^2)$
- Average Case will be $O(n \log n)$ with proper pivot choosing techniques

Choosing the Pivot

- Don't Select a random element
- Don't stick with one sole pivot choice!
- Some times we choose three random elements and use their median as the pivot

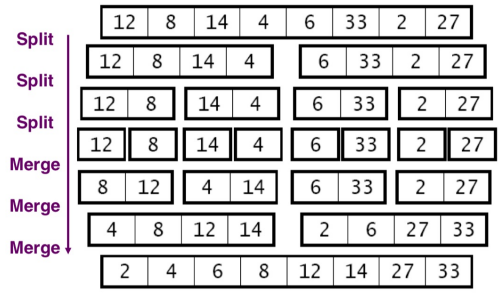
Recursive Stack Space Overhead

- Have to send $\log(n)$ methods to program stack $\Rightarrow$ storage complexity
- Recursion results in storage complexity being considered and is on the order of $O(\log n)$

Merge Sort

Like quick-sort, merge-sort is a divide-and-conquer algorithm.

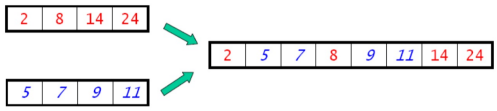
- *divide:* split the array in half, forming two subarrays
- *conquer:* apply merge-sort recursively to the subarrays, stopping when a subarray has a single element
- *combine:* merge the sorted subarrays



All of the sorting algorithms we've seen thus far have sorted the array in place. They used only a small amount of additional memory, i.e., O(logn) additional space (for recursion)

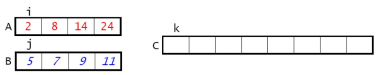
Merge-sort is a sorting algorithm that requires an additional temporary array of the same size as the original one.

- it needs *O*(n) additional space, where n is the array size
- space for *merging* two sorted arrays into a single sorted array.



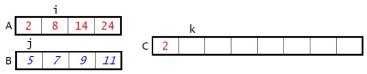
Merge Sorted Subarrays

To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

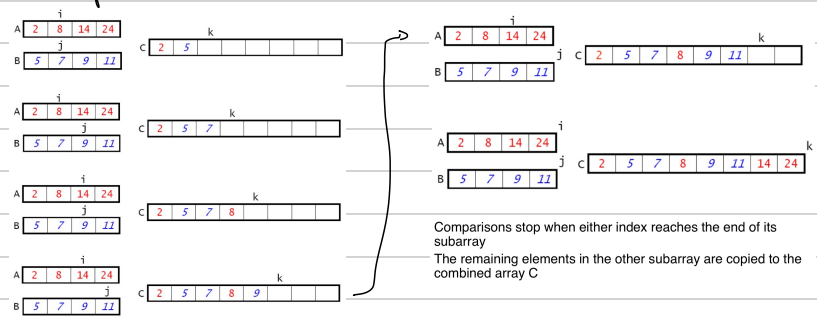


We repeatedly do the following:

- compare A[i] and B[j]
- copy the smaller of the two to C[k]
- increment the index of the array whose element was copied
- increment k



Steps



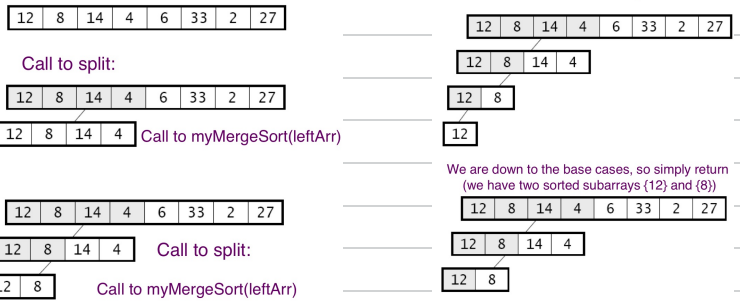
Comparisons stop when either index reaches the end of its subarray

The remaining elements in the other subarray are copied to the combined array C

Recursive Implementation

```
static void mergeSort(int[] arr) {
        myMergeSort(arr);
}

static void myMergeSort(int[] arr) {
        if (arr.length == 1) return;
        // Allocate leftArr and rightArr
        split(arr,leftArr,rightArr);
        myMergeSort(leftArr);
        myMergeSort(rightArr);
        Merge(leftArr,RightArr, arr);

}
```

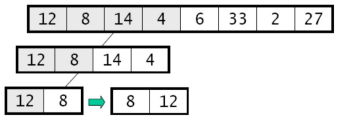
Steps



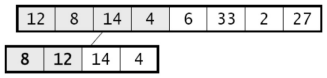
Further split it into two size-1 subarrays, and issue recursive calls again

We are down to the base cases, so simply return (we have two sorted subarrays {12} and {8})

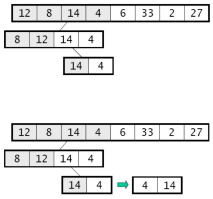
Call merge() to merge two subarrays into original array



Return to the recursive call for the 4-element subarray, and start another recurise call for the right subarray {14, 4}.



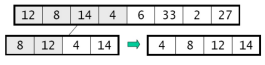
Repeat the similar process for the right 2-element subarray




```
static void mergeSort(int[] arr) {
        myMergeSort(arr);
}

static void myMergeSort(int[] arr) {
        if (arr.length == 1) return;
        // Allocate leftArr and rightArr
        split(arr,leftArr,rightArr);
        myMergeSort(leftArr);
        myMergeSort(rightArr);
        Merge(leftArr,RightArr, arr);
}
```

Return from the recursive call for the 2-element right subarray. We have

Call merge() to merge the 2-element subarrays, and copy the elements back



Etc.

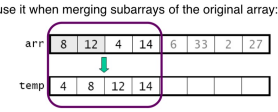
Implementation of Merge Sort

Our approach so far was to create new arrays for each new set of subarrays, and to merge them back into the array that was split.

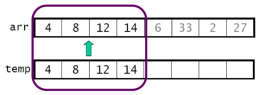
- Creates a lot of arrays in the recursive call chain

Instead, we'll create a temp. array of the same size as the original.

- pass it to each call of the recursive merge-sort method
- use it when merging subarrays of the original array:



- after each merge, copy the result back into the original array:




```
static void merge(int[] arr, int[] temp, int leftStart, int leftEnd, int rightStart, int rightEnd) {
        int i = leftStart; // index into left subarray
        int j = rightStart; // index into right subarray
        int k = leftStart; // index into temp
        while (i <= leftEnd && j <= rightEnd) {
                if (array[i] <= array[j])
                        {
                          temp[k] = array[i];
                          i++;
                        }
                else
                {
                  temp[k] = array[j];
                  j++;
                }
                k++; }
        /* Copy remaining elements of left array if any */
                while (i <= leftEnd)
                        {
                          temp[k] = array[i];
                          i++;
                          k++;
                        }
        /* Copy remaining elements of right if any */

        for (i = leftStart; i <= rightEnd; i++)     // copy back
                arr[i] = temp[i];
}
```

- We use a wrapper method to create the temporary array, and to make the initial call to a separate recursive method:

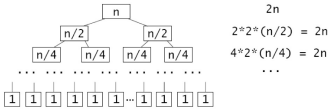
```
static void mergeSort(int[] arr) {
      int[] temp = new int[arr.length];
      myMergeSort(arr, tmp, 0, arr.length - 1);
}

static void myMergeSort(int[] arr, int[] temp, int start, int end) {
      if (start >= end ) // base case
            return;
      int middle = (start + end)/2; // The splitting step

            // Sort first and second halves
              myMergeSort (arr, temp, start, middle);
              myMergeSort (arr , temp, middle+1, end);

              // Merge the sorted halves
              merge(arr, temp, start, middle, middle+1, end);
```

Running Time Analysis

Merging two halves of an array of size n requires 2n moves.

Merge-sort repeatedly divides the array in half, so we have the following call tree:

recursive call tree for mSort() # moves



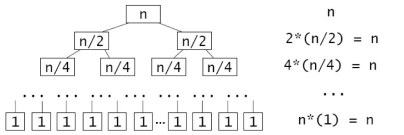
- At all but the last level of the call tree, there are 2n moves
  - How many levels are there?
- M(n) = $2n\log_2 n$ (worst-case or best-case)
- C(n) ≤ $n\log_2 n$ (between 0.5n*log(n) and n*log(n))
- O(n log n) overall

Compared to Quick-Sort

Partitioning an array requires n comparisons, because each element is compared with the pivot.

best case: partitioning always divides the array in half

recursive call tree for qSort() # comparisons



- at each level of the call tree, we perform n comparisons
- There are $\log_2 n$ levels in the tree. So C(n) = $n\log_2 n$
- M(n) ≤ 1.5 $n\log_2 n$ (at most n/2 swaps at each level)

Quick Sort vs. Merge Sort

Quick-sort used often

- Low extra space
- Good performance average

For Quick-Sort

- worst-case does not appear often, average-case is closer to best-case (n*log(n) comparisons and 1.5n*log(n) moves)
- It is important to choose good pivots, to have n*log(n) running time

For merge-sort

- Average-case is close to worst-case
- Between 0.5n*log(n) and n*log(n) comparisons and 2n*log(n) moves

Comparisons Summary

Algorithm	Best-case	Worst-case	Average-case	Extra space
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Shell-sort	O(n*logn)	$O(n^{1.5})$	$O(n^{1.5})$, but can be $O(n^{1.25})$	O(1)
Quick-sort	O(n*logn)	$O(n^2)$	O(n*logn)	O(logn) (ave case) O(n) (worst case)
Merge-sort	O(n*logn)	O(n*logn)	O(n*logn)	O(n)
Heap-sort	O(n*logn)	O(n*logn)	O(n*logn)	O(1)

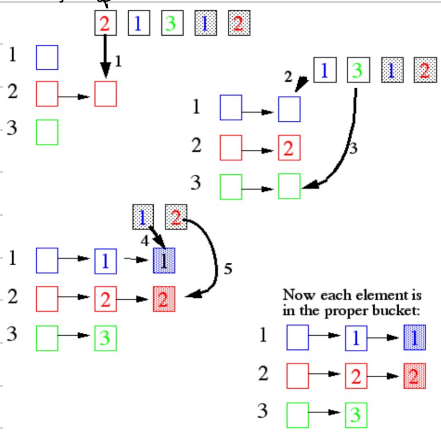
Bucket Sort

Bucket sort

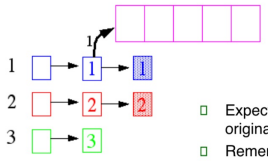
- Assumption: integer keys in the range [0, M)
- Basic idea:
  1. Create *M* linked lists (*buckets*), one for each possible key value
  2. Add each input element to appropriate bucket
  3. Concatenate the buckets

Remember hash tables also uses buckets?

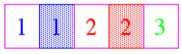
Example



- Pull the elements from the buckets into the array



- Expected total time is O(M+N), with N = size of original array, M = number of buckets
- Remember hash tables also uses buckets?
  - Bucket sort preserves order (key values) in buckets
  - Hashing mixes up elements with diverse key values



Keys of Non-Integer Types

- What if keys are not integers?
  - Assumption: input is *N* floating numbers (scaled) in [0, 1]
  - Basic idea:
    - Create *M* linked lists (*buckets*) to divide interval [0,1] into subintervals of size 1/*M*
    - Add each input element to appropriate bucket and sort the bucket with insertion sort
  - Choose M=O(N)
  - Uniform input distribution → expected bucket size is O(1)
    - Therefore the expected total time is O(N): O(N) to put elements into the buckets and O(N) to move them back into the array
- With uniform key distribution, Bucket Sort has O(N) running time
- But sensitive to the key distribution in the range (what if it's not uniform?)
- Pays with space for time
- Pays with worst-case time for average-case time

Graphs

- Every tree is a graph, but not every graph is a tree as graphs can have cycles
- Can be used for many applications
- Can be used for social networks where all users are nodes and their connections are the edges
- Nodes can be connected directionally by arrows or just by lines (non-directional)
- Flight routes can also be represented nodes are airports/cities and edges are distances and routes available

Consists of...

→ A set of vertices (also known as nodes)
→ A set of edges (also known as arcs) each of which connects a pair of vertices

Terminology

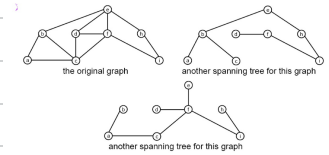
- Two vertices are adjacent if they are connected by a single edge
- The collection of vertices that are adjacent to a vertex v are referred to as v's neighbors
- A path is a sequence of edges that connects two vertices
- A graph is connected if there is a path between any two vertices
- A graph is complete if there is an edge between every pair of vertices
  ↳ The length of the path between any two nodes is 1
- A directed graph has a direction associated with each edge, which is depicted using an arrow
  ↳ Edges in a directed graph are often represented as ordered pairs of the form (start vertex, end vertex)
- A path in a directed graph is a sequence of edges in which the end vertex of edge i must be the same as edge i+1
- Degree is the number of neighbors of a node

Trees vs. Graphs

- A tree is a special type of graph
  → It is connected and undirected
  → It is acyclic: there is no path containing distinct edges that starts and ends at the same vertex
  → We usually single out one of the vertices to be the root of the tree, although graph theory doesn't require this
- Spanning trees
  → A subset of a connected graph
  → All of the N vertices
  → A subset of the edges that form a tree (subset contains N-1 edges)

A spanning tree is a subset of a connected graph that contains:
- all of the *N* vertices
- a subset of the edges that form a tree (the subset contains *N*-1 edges)

Examples:



Graph Representation - Matrix

- Adjacency matrix - a 2D array used to represent the edges and any associated costs
  → edge[r][c] is the cost of the link from vertex r to vertex c
  → use a special value to note there is no direct link from r to c
- Representation is good if the graph is dense (most nodes are connected) but wastes memory if the graph is sparse (few edges per vertex)

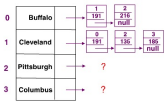
Graph Representation – Lists

- Adjacency lists = a set (either an array or LL) of linked lists that is used to represent edges and its associated costs
- Representation is good if the graph is sparse, but inefficient if the graph is dense
  - → no memory allocated for non-existent edges
  - → the nz Pointers in the linked lists use extra memory
  - → List traversals required to find a link

Graph Representation with Lists

```
public class Graph {
    class Vertex {
        private String id;
        private Edge edgeHead; // adjacency list
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        …
    }
    class Edge {
        private int endNode;
        private double cost;
        private Edge next;
        …
    }
    private Vertex[] vertices;
    private int numVertices;
    private int maxNum;
    …
}
```

```
public class Graph {
    class Vertex {
        private String id;
        private LinkedList <Edge> edges; // adjacency list
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        …
    }
    class Edge {
        private int endNode;
        private double cost;
        …
    }
    private Vertex[] vertices;
    private int numVertices;
    private int maxNum;
        …
}
```



Adding Nods

```
public class Graph {
    class Vertex {
        private String id;
        private LinkedList <Edge> edges; // adjacency list
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        …
    }
    class Edge {
        private int endNode;
        private double cost;
        …
    }
    private Vertex[] vertices;
    private int numVertices;
    private int maxNum;
    …
}
```

```
public Graph(int maximum) {
    vertices = new Vertex[maximum];
    numVertices = 0;
    maxNum = maximum;
}

public int addNode(String id)
{
    // code to grow vertices if array "vertices"
    // is too small to have another node
    …

    vertices[numVertices] = new Vertex(id);
    // any further initialization
    …
    numVertices++;
    return numVertices-1; // return the index
}
```

Adding Edges

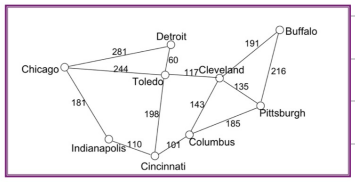
```
public class Graph {
    class Vertex {
        private String id;
        private LinkedList <Edge> edges; // adjacency list
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        …
    }
    class Edge {
        private int endNode;
        private double cost;
        …
    }
    private Vertex[] vertices;
    private int numVertices;
    private int maxNum;
    …
}
```

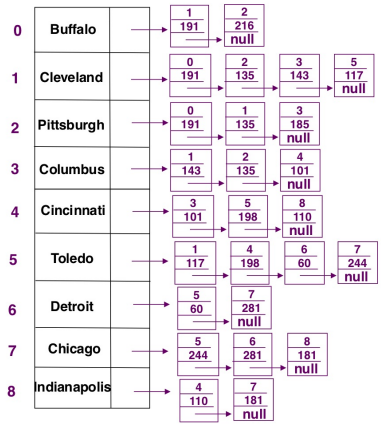
```
public void addEdge(int i, int j, double cost)
{
    // add an edge from i to j
    Edge newEdge = new Edge();
    newEdge.endNode = j;
    …

    vertices[i].edges.add(newEdge);


    // add an edge to j for node i
    newEdge = new Edge();
    newEdge.endNode = i;
    …
    vertices[j].edges.add(newEdge);
}
```

Graph Traversals

Traversing a graph involves starting at some vertex and visiting all of the vertices that can be reached from that vertex.

- visiting a vertex = processing its data in some way
- if the graph is connected, all of the vertices will be visited





Example:

- A Web crawler (vertices = pages, edges = links)

We will consider two types of traversals:

- **depth-first:**
  - Visit the starting vertex
  - Proceed as far as possible along a given path (via a neighbor) before "backtracking" and going along the next path
- **breadth-first:**
  - visit the starting vertex
  - visit all of its neighbors
  - visit all unvisited vertices 2 edges away
  - visit all unvisited vertices 3 edges away, etc.

Depth First Traversal

- Visit a vertex then make recursive calls on adjacents yet-to-be visited neighbors

```
depthFirstTrav(v)
    myDepthFirstTrav(v, NULL)

myDepthFirstTrav(node, parent)
   visit node and mark it as visited
   node.parent  = parent
   for each vertex w in node's neighbors
      if (w has not been visited)
         myDepthFirstTrav(w, node)
```

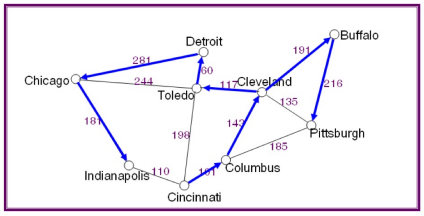
```
public class Graph {
        class Vertex {
                private String id;
                private linkedList <Edge> edges;
                   // adjacency list
                private boolean encountered;
                private boolean done;
                private Vertex parent;
                private double cost;
                …
        }
        class Edge {
                private int endNode;
                private double cost;
                …
        }
        private Vertex[] vertices;
        private int numVertices;
        private int maxNum;
        …
}
```

Implementation

```
myDepthFirstTrav(v, parent)
      visit v and mark it as visited
      v.parent = parent
      for each vertex w in v's neighbors
            if (w has not been visited)
                  myDepthFirstTrav(w, v)
```

Implementation:

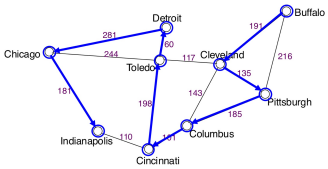
```
public void myDepthFirstTrav(int i, int parent) {
      System.output.println(vertices[i].id);
      vertices[i].encountered = true;
      vertices[i].parent = parent;
      Iterator<Edge> edgeItr = edges.iterator();
      while (edgeItr.hasNext()) {
            Edge curEdge = edgeItr.next()
            j = curEdge.endNode;
            if (vertices[j].encountered == false)
                  myDepthFirstTrav(j, i);
      }
}
```



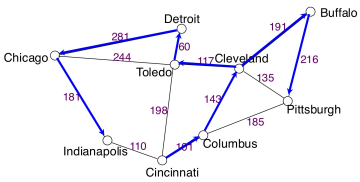
Refer to this graph for examples

Example From Buffalo

```
myDepthFirstTrav("Buffalo", null)
      visit Buffalo; Buffalo.parent = NULL
      w = "Cleveland"
      myDepthFirstTrav("Cleveland", "Buffalo")
            visit Cleveland; Cleveland.parent = Buffalo
            w = "Pittsburgh"
            myDepthFirstTrav("Pittsburgh", "Cleveland")
                  visit Pittsburg; Pittsburg.parent = Cleveland
                  w = "Buffalo"; Buffalo has been visited
                  w = "Columbus"
                  myDepthFirstTrave("Columbus", "Pittsburgh")
                        visit Columbus; Columbus.parent = Pittsburg;
                        w = "Cincinnati"
                        myDepthFirstTrav("Cincinnati", "Columbus")
                        …
```

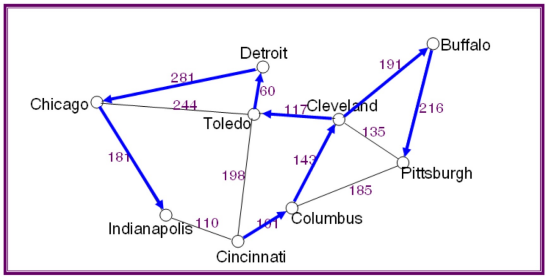


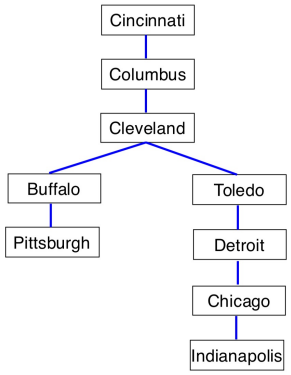
Example From Cincinnati



Order: Cincinnati, Columbus, Cleveland, Buffalo, Pittsburg, Toledo, Detroit, Chicago, Indianapolis

Depth First Spanning Tree

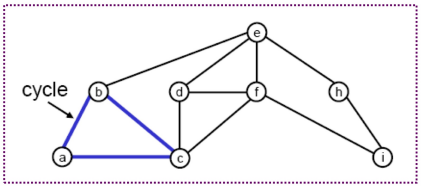




Application of Depth First Traversal

To discover a cycle:

- perform a depth-first traversal
  - when considering the non-parent neighbors of a current vertex, if we discover one that is already marked as visited, there is a cycle
- If we discover no cycles during the course of the traversal, the graph is acyclic.

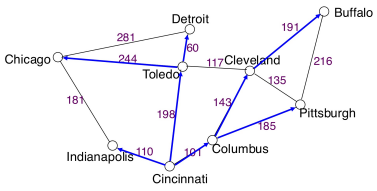


Breadth First Traversal

• Use a queue

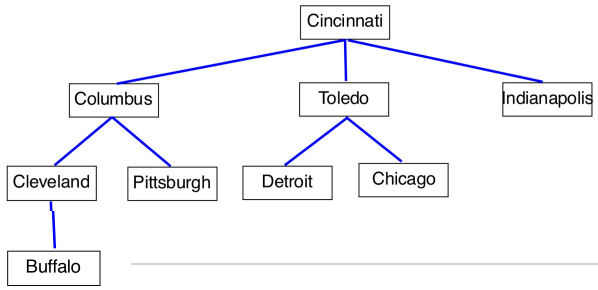
```
bfTrav(origin)

    origin.parent = null
    create a new queue q
    q.insert(origin)

    while (!q.isEmpty())
        v = q.remove()
        visit v

        for each vertex w in v's neighbors
            if w is not encountered
                mark w as encountered
                w.parent = v;
                q.insert(w);
```

Example From Cincinnati → Queue Operations



Order: Cincinnati, Columbus, Toledo, Indianapolis, Pittsburgh, Cleveland, Detroit, Chicago, Buffalo

```
insert("Cincinnati");
remove("Cincinnati"); // and print it
insert("Columbus");
insert("Toledo");
insert("Indianapolis");
remove("Columbus"); // and print it
insert("Cleveland");
insert("Pittsburgh");
remove("Toledo"); // and print it
insert("Detroit");
insert("Chicago");
remove("Indianapolis"); // and print it
remove("Cleveland"); // and print it
insert("Buffalo");
remove("Pittsburgh"); // and print it
remove("Detroit"); // and print it
remove("Chicago"); // and print it
remove("Buffalo"); // and print it
```



Running Time of Graph Traversal

Let V = number of vertices in the graph, and E = number of edges

Assume we use an adjacency list as the data structure

A traversal requires $O(V + E)$ steps.

- visit each vertex once
- traverse each vertex's adjacency list at most once
  - the total length of the adjacency lists is $2E = O(E)$
- for a dense graph, $E \rightarrow V^2$, so the worst-case bound is $O(V^2)$
- for a sparse graph, $O(V + E) << O(V^2)$

Shortest-Path Problem

- What is the shortest path from one vertex to another (i.e. the one with the minimum cost)
  - Travel: lowest mileage or fastest trip
  - Internet: forwarding traffic along the "best" router paths.

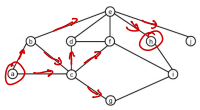
A special case example

A solution for an *unweighted* graph:

- Start a breadth-first traversal from the origin, x
- Stop the traversal when you reach the target, y
- The path from x to y in the resulting (possibly partial) spanning tree is a shortest path

A breadth-first traversal works for an unweighted graph because

- the shortest path is simply one with the fewest edges
- a breadth-first traversal visits nodes in order according to the number of edges they are from the origin.



Implementation for special case

```
shortestPath_bfTrav(source, desti)
    mark source as encountered
    source.parent = null
    source.cost = 0;
    create a new queue q
    q.insert(source);

    while (!q.isEmpty())
        v = q.remove()
        if (v==desti) break;
        for each vertex w in v's neighbors
            if w is not encountered
                mark w as encountered
                w.parent = v;
                w.cost = v.cost + 1;
                q.insert(w);
```

If you remove this, it's just a Breadth First Traversal

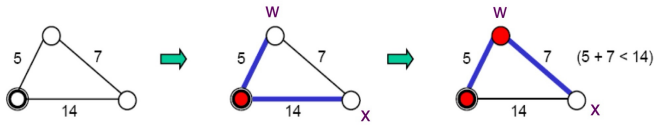
Shortest Path for Weighted Graphs – Dijkstra's algorithm

- Finds shortest paths from vertex v to all other vertices that can be reached from v
  - → Single-source all-destination problem

Assumptions: Graph is connected

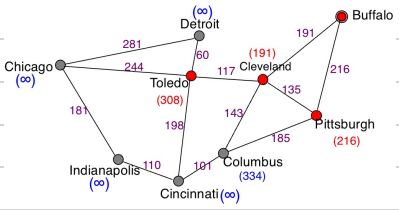
Edge costs are non-negative (More hops = higher cost)

- Basic idea:
  - → maintain estimates of shortest paths from v to every vertex (along with assumed costs)
  - → Gradually refine these estimates as we traverse the graph
- We say that a vertex is finalized if we are certain we have found the shortest path from v
- We repeatedly do the following
  - find the not-yet-finalized vertex w with the lowest cost estimate
  - mark w as finalized (shown as a filled circle below)
  - examine each not-yet-finalized neighbor x of w to see if there is a shorter path to x that passes through w; if there is, update the shortest-path estimate for x



Example: Shortest Path from Buffalo to Toledo

- Distance to Buffalo is 0, so it is finalized
- Initially set distance estimates:
  - To all neighbors - edge weights
  - To other cities - unknown, using *infinity*



Red vertices are finalized

Example & Implementation / Walk-through

```
1  Initialization:
2    N = {u}
3    for all nodes v
4        if v adjacent to u then
5            D(v) = c(u,v)
6            p(v) = u
7        else D(v) = ∞
```

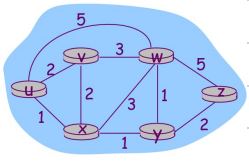
Notation:

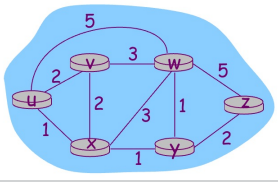
- c(x,y): link cost from node x to y; ∞ if not direct neighbors
- D(v): current path cost estimate from source to dest. v
- p(v): predecessor node along path from source to v
- N: set of nodes whose least cost path definitively known
- u: starting node

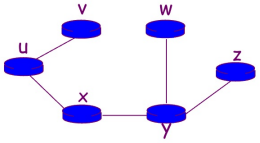
```
8  Loop
9    find w not in N with smallest D(w)
10   add w to N
11   update D(v) for all v adjacent to w and not in N' :
12       D(v) = min( D(v), D(w) + c(w,v) )
         Update p(v)
13   /* new cost to v is either old cost to v or known
14    shortest path cost to w plus cost from w to v */
15 until all nodes are in N
```

Steps

Step	N	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					







Dijkstra's Algorithm Complexity

Assuming use of linked list based implementation

```
1  Initialization:
2   N = {u}
3   for all nodes v                    // V iterations
4     if v adjacent to u               // Check of the total of up to E edges
5        then D(v) = c(u,v)
6     else D(v) = ∞
7                                      // O(V) to build the heap
8  Loop                                // V iterations
9    find w not in N with smallest D(w)          // logV to reorg the heap
10   add w to N
11   update D(v) for all v adjacent to w and not in N :
12      D(v) = min( D(v), D(w) + c(w,v) )        // A step per every link (E total); logV per step to reorganize the heap
13   /* new cost to v is either old cost to v or known
14    shortest path cost to w plus cost from w to v */
15 until all nodes in N
```

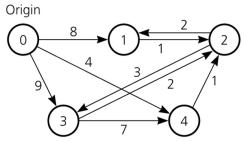
Overall complexity is O(V + E +VlogV + ElogV) = O(ElogV)

17-

All-pairs shortest paths problem

- Dijkstra's algorithm solves the problem of single-source all-destination shortest paths
- It can be used to solve the all-source all-destination shortest path problem too
  - Run Dijkstra's V times, once for each vertex
  - Running time: O(VE logV)

(a)



A Weighted Directed Graph

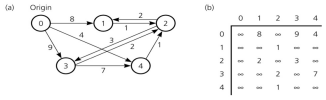
(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Its Adjacency Matrix

Dijkstra's Shortest Path Algorithm - Trace

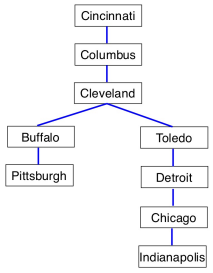
Step	v	vertexSet	weight [0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

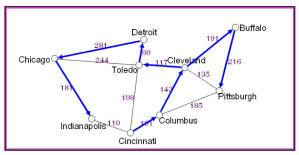


Minimum Spanning Tree

- An MST of connected (weighted?) graph has the lowest total cost among all possible spanning trees
- An MST may not be unique
- Applications: finding an MST could be used to
  - → Determine shortest highway system to connect a set of cities
  - → Calculate smallest length of cables needed to connect a computer network
  - → Group communication on the Internet (cost = hop distance to reduce bandwidth consumption)

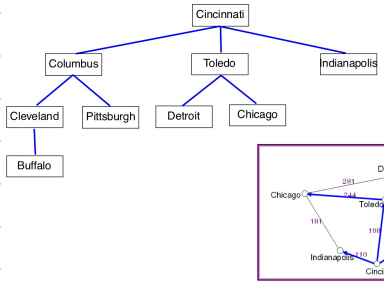
Depth-First ?

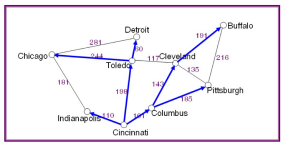




-7-

Breadth-First ?



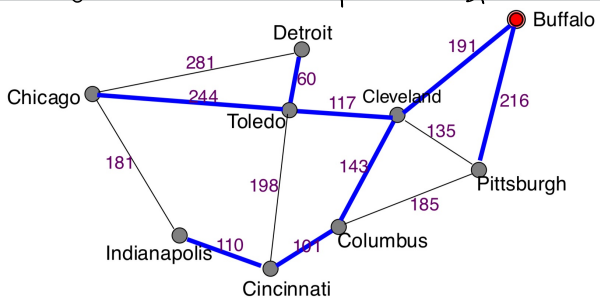


-8

There are both certain spanning trees, but not correct to claim an MST.

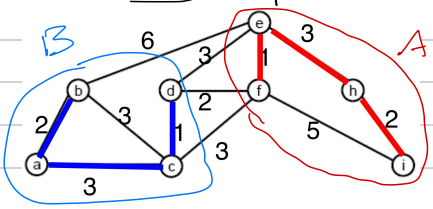
Shortest Path Spanning Trees

- Dijkstra's algorithm outputs this type of tree



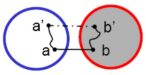
- We cannot claim this is an MST
  - ↳ Dijkstra's is about a single source node, but an MST should look globally!

Key Idea: If we divide vertices into two disjoint subsets A and B, then the lowest-cost edge joining a vertex in A to a vertex in B will (say (a, b)) must be part of the MST



Proof (by contradiction)

- assume there is an MST (call it T) that doesn't include (a, b)
- T must include a path from a to b, so it must include another edge (a', b') that connects subsets A and B with a path a to a' to b' to b
- adding (a, b) to T introduces a cycle
- removing (a', b') gives a spanning tree with lower cost, which contradicts the original assumption.



Prim's Algorithm

Begin with the following subsets:

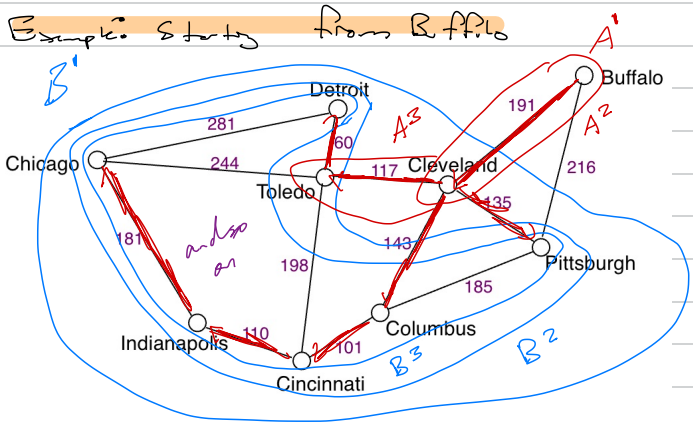
- A = any one of the vertices
- B = all of the other vertices

Repeatedly select the lowest-cost edge (a, b) connecting a vertex in A to a vertex in B and do the following:

- add (a, b) to the spanning tree
- update the two sets: A = A U {b}, B = B – {b}

Continue until A contains all vertices.

Example: Starting from Buffalo



Prim's Algorithm vs. Dijkstra's Algorithm

```
1  Initialization:
2   A = {u}
3   for all nodes v
4        if v adjacent to u then
5         C(v) = c(u,v)
6            p(v) = u
6       else C(v) = ∞
7
8  Loop
9   find w not in A such that C(w) is a minimum
10   add w to A
11   update C(v) for all v adjacent to w and not in A :
12         C(v) = min( C(v), c(w,v) )
           p(v) = w
15 until all nodes are in A
```

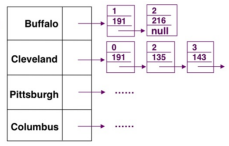
Notation:
- c(x,y): link cost from node x to y; ∞ if not direct neighbors
- C(v): current estimate of cost-to-connect v to A
- p(v): predecessor node along path from source to v
- A: set of nodes added to MST
- u: starting node

```
1  Initialization:
2   N = {u}
3   for all nodes v
4        if v adjacent to u then
5         D(v) = c(u,v)
6            p(v) = u
6       else D(v) = ∞
7
8  Loop
9   find w not in N such that D(w) is a minimum
10   add w to N
11   update D(v) for all v adjacent to w and not in N :
12         D(v) = min( D(v), D(w) + c(w,v) )
           p(v) = w
15 until all nodes are in N'
```

Notation:
- c(x,y): link cost from node x to y; ∞ if not direct neighbors
- D(v): current path cost estimate from source to dest. v
- p(v): predecessor node along path from source to v
- N: set of nodes whose least cost path definitively known
- u: starting node

Prim's Algorithm Complexity

```
1  Initialization:
2   A = {u}
3   for all nodes v                    // V iterations
4      if v adjacent to u then         // Check of the total of up to E edges
5         C(v) = c(u,v)
6         p(v) = u
6      else C(v) = ∞
7                                      // O(V) to build the heap
8  Loop                                // V iterations
9   find w not in A such that C(w) is a minimum     // logV to reorg the heap after w's removal
10   add w to A
11   update C(v) for all v adjacent to w and not in A :     // A step per every link (E total); logV per step to reorganize the heap
12        C(v) = min( C(v), c(w,v) )
          p(v) = w
15 until all nodes are in N'
```



Overall complexity is O(V + E + VlogV + ElogV)=O((E+V)logV) =O(ElogV)