



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

Example: A Bag ADT

just a container for a group of data items

(i) Positions of data items don't matter

$$4 \in \{3, 2, 6\} \Leftrightarrow \{2, 6, 3\}$$

(ii) Data items do not need to be unique

$$\{2, 2, 10, 2, 5\} \text{ is a bag, but not set}$$

Encapsulation

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!  
    ...  
}
```

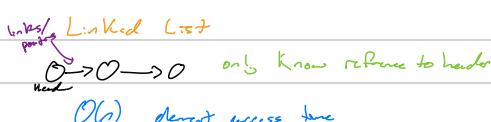
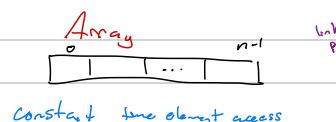
Cannot modify field directly due to being private but add method allows controlled modifications

Generic Types

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

Can use Object
expands, just matches
for type parameters

Arrays vs. Linked Lists



Constant time element access

O(1) direct access time

Phonebook Example

- Implement a phonebook

(1) Operations: add(element)

remove(element)

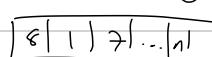
Search(Phone) return number

(2) Data Structures: Array (sorted/unsorted)

Linked List (sorted/unsorted)

add/remove should not disturb sorting

Unsorted Array



Best case: 1 check

Worst case: n comparisons

Average case: $\frac{n}{2}$ comparisons

Add

Just Add to end

Remove

Implementation of IntBag in Java

public class IntBag implements Bag {

// instance variables

Access Modifiers
private int[] items;

private int numItems;

// methods

public boolean add(int item) {

if (numItems == items.length)

return false;

items[numItems++] = item;

return true;

...
}

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

Using a Superclass to Implement Generics

public class Bag {

// instance variables

private Object[] numObj;

private int numItems;

// methods

...

public class Test {

push

Bag m = new Bag();

m.add("37G");

most specific to prevent runtime errors

String bodyTemp = (String) m.grabItem();

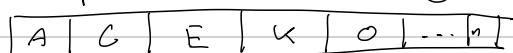
m.add(new Integer(96));

Assume grabs at [0..1]

Integer temp = (Integer) m.grabItem(); // runtime

...
Error bc ["37G", 96]

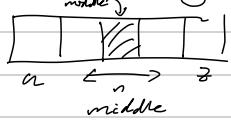
Example of Sorted Array Addition



- 1) Logn Binary Search → add $O(n + logn)$
- 2) n Shifting Elements $logn \ll n$
so we say $O(n)$

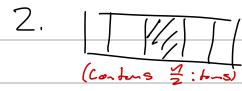
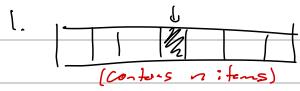
Binary Search

Sorted Array



You need a
sorted array
for this to work

$O(\log(n))$ Complexity



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

Find Number (Person)

Pseudocode

Low = 0

High = phonebook.size

while low <= high \geq

P = floor((low + high) / 2)

Compare Pth person

if f then same

return number

else if contain in book

high = P - 1

else

low = P + 1

3 return Not Found

3

Recursion Mathematical Background

Non-recursive Programming

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

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

Recursive Programming

- $\sum_{i=1}^n i = 1+2+3+\dots+(n-1)+n$
 $\text{sum}(n) = \text{sum}(n-1) + n$
- ↑ recursive call
Reduces size of problem!

Recursive Call must

be shorter to allow
code to terminate

- A recursive method is a method that invokes itself

- Must specify a base/stopping condition!

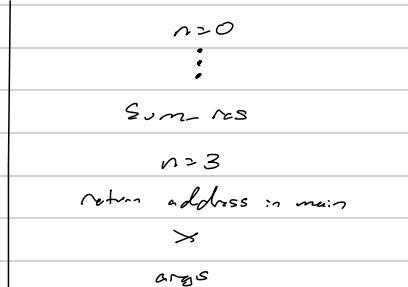
- Can get farcely to look at in stack

↳ Simple to implement but hard on computer memory as no computations done until 'n=0'

- Code will run forever without a base case \rightarrow stack overflow usually

- Stack works in a First In - Last Out manner

```
int sum(int n) {
    ...
    if (n == 0)
        return 0;
    else
        return n + sum(n-1);
}
```



when n goes up
Complexity goes a lot
more \rightarrow inefficient

Example: Counting Occurrences of a character in a String

Thinking Recursively

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

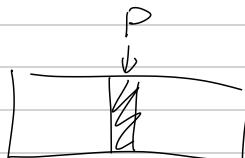
What is the base case? : if (s.length() == 0) return 0;

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 \rightarrow code11.java

```
myFindNumber (person, low, high)
    if (low > high) return Not_Found
    p = floorAvg (low, high)
    if the same
        return number
    else if even
        high = p-1
    else
        low = p+1
```



Fibonacci

- Call Tree Created
- Iteration better



Runtime Analysis

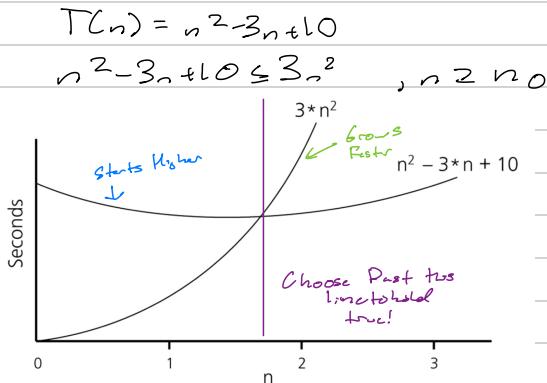
```
int sum(float[] array) {
    for (int i = 0; i <= array.length; i++)
        array[i] = array[i]/0.2;
}
```

```
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 as constant time operations
- Both methods need $2N$ operations

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



Functions Growth Rates: 'Big O' Notation

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

$\hookrightarrow f(N)$ is runtime complexity of $T(N)$

"Big O": $T(N) = O(f(N)) : f \exists c_{\text{const}} > 0 \text{ s.t.}$

$T(N) \leq c f(N) \forall N \geq n_0$

↳ we focus on very Large Numbers to determine

Example: $10^4 N^2 + 10000 = O(N^2)$?

$\frac{10^4 N^2 + 10000}{N^2} \leq c N^2$ ↳ grows much faster
constants don't matter for small N

Instead of $10^4 N^2 + 10000 = O(N^2)$?

↳ Can be true for large c , so choose $O(M^2)$

Remember you can choose c and n_0 values!

Functions Growth Rates: Other Definitions

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

$T(N) \geq c f(N)$ for all $N \geq n_0$ Opposite to Big O

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

$T(N) = \Theta(f(N))$ 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

$T(N) < c f(N)$ for all $N > n_0$. ↳ instead of \exists

or

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

Example: $10^4 N^2 + 10000 = \omega(N^3)$

Big-O Toolbox (for positive monotonic 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 $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$ then $T_1(N) + T_2(N) = O(f(N) + g(N))$

If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$ then $T_1(N) * T_2(N) = O(f(N) * g(N))$

If $T_1(N) = O(f(N))$ and $g(x)$ is monotonic, then $g(T_1(N)) = O(g(f(N)))$

• 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

$$\log(cd) = \log(c) + \log(d)$$

$$\log(c/d) = \log(c) - \log(d)$$

$$\log(c^d) = d \log(c)$$

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

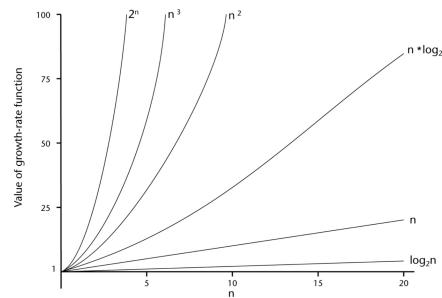
Corollary: Logarithmic grows

slower than any polynomial

$$\log N < N^c \quad \text{Dominant}$$

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

Comparison of Growth-Rate Functions



Useful Mathematical Equations

$$\sum_{i=1}^n i = 1+2+\dots+n = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^n i^2 = 1^2 + 4 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + \dots + 2^{n-1} = 2^{n-1}$$

Applying Concepts & Relative Rates

- > 1000000 versus $0.01 * \sqrt{N}$
- > $\log(N)$ versus \sqrt{N}
- > $\log(N)$ versus $N^{0.001}$ $N^{0.001} = N^* N^{0.001}$
- > N^3 versus $10000 * N^2$
- > $\log^2(N)$ versus $10^4 \log(N^2) = 50 * \log(N)$
- > $2 * \log(N)$ versus $\log_2(N)$ $\log(N) = \frac{\log_2(N)}{\log_2(2)}$
- > $N^0 2^N$ 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

$$(\cancel{N} \cancel{N}) + \cancel{b} \rightarrow 0$$

Time Complexity: $O(N)$

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

General Rules

- Simple Statement: Constant
 - $\cancel{O(1)}$: $i++, i < n, \text{etc.}$
- Simple Loops: # iterations
- Nested Loops: Product of # iterations of outer and inner loops cost of inner body
 - $\cancel{O(1)}$: $\text{for}(n)$
 - $\cancel{O(n)}$: $\text{for}(m)$
 - $\cancel{O(n^2)}$: $K++$

- Consecutive Statements: Count most expensive

$\cancel{O(1)}$:
 $\cancel{O(n)}$: $\text{while}(); \cancel{O(n)}$
 $\cancel{O(n^2)}$: $\text{for}(n) \quad \cancel{O(m)}$
 $\cancel{O(n^2)}$: $\text{for}(m) \quad K++$

- Conditions: Count most expensive branch

\hookrightarrow Focus on worst-case

Examples

$\cancel{O(1)}$: $i = 0;$
 $\cancel{O(1)}$: $\text{if}(\dots) \quad \cancel{O(1)}$

$\cancel{O(1)}$: $\cancel{O(1)}$: $\cancel{O(1)}$

$K = 1;$

$\text{for}(i=1; i < n; i++) \quad \cancel{O(1)}$

$K++;$ $\cancel{O(n)}$

$\cancel{O(n)}$

$\cancel{O(n^2)}$: $\text{for}(i=0; i < n; i = 3*i) \quad \cancel{O(n^2)}$

$i = 1;$

$\cancel{O(n)}$: $\cancel{O(n)}$

$j = i + 1;$ n times

$\cancel{O(n)}$: $\cancel{O(n)}$

$\cancel{O(n)}$: $\cancel{O(n)}$

$\cancel{O(n)}$: $\cancel{O(n)}$

$i = 0;$
 $\text{while}(i < n) \{ \dots i++; \dots \}$ m
 $\text{for}(i=0; i < n; i++)$
 $\text{for}(j=0; j < n; j++)$ n^2 $\leftarrow O(m+n^2)$
 $K++;$ $n^2 \gg m$, but $n^2 \neq m$

$K = 1;$ $\cancel{O(n^2)}$

$K = 1;$ $\cancel{O(n^2)}$

$K = 1;$ $\cancel{O(n^2)}$

$\cancel{O(n^2)}$

$\cancel{O(n^2)}$

$K = 1;$
 $\text{for}(i=0; i < n; i++) \quad \cancel{O(n)}$
 $\text{for}(j = i+1; j < n; j++) \quad \cancel{O(n)}$
 $K++;$ $\cancel{O(n)}$

<

Linked lists

Array Representation

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

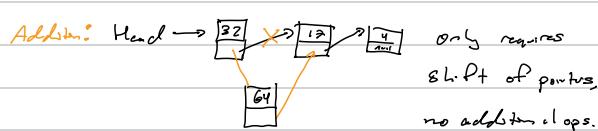
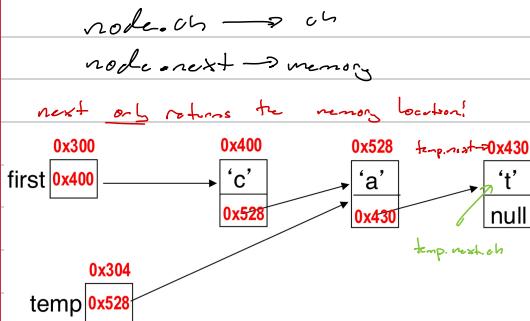
LinkedList Representation

- ✓ Dynamic sequence: easy addition/removal
- Links of nodes both ends have a reference and data value
- ✗ Access is linear O(n) to get the n^{th} element: Head \rightarrow desired node
- Last Data Node: is a null reference
- ✓ No capacity limit provided enough memory
- ✗ Memory overhead for the links

Example LL

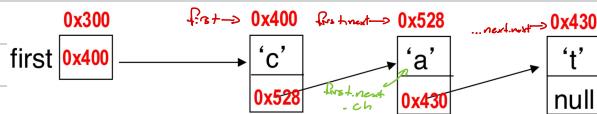
```
public class StringNode {
    private char ch;
    private StringNode next;
    ...
}
```

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



A reference is also a variable

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



Two nodes can have two different pointers.

Different pointers can point to same memory address

Recursion on Linked Lists

Recursive definition of a LL

A LL is either :

(Empty Case)

(i) a single node followed by

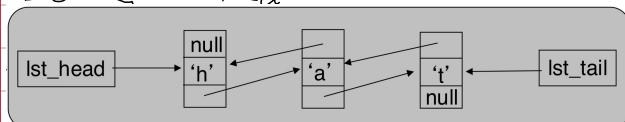
a LL (Recursive case)

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);
}
```

Double Linked List



Example DLL

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

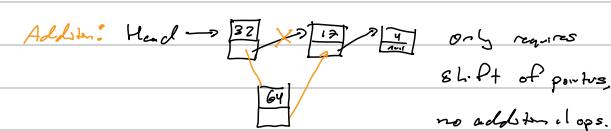
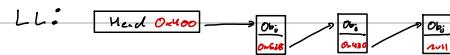
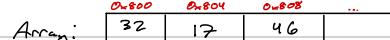
```
public class StringNode {
    private char ch;
    private StringNode next;
    private StringNode prev;
    ...
}
```

Allows for traversal in either direction

Memory Diagrams

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



In Class DLL example

Assume you have LLString and StringNode

\rightarrow Access node at position i in a doubly linked list

public StringNode getNode(int i) {

if ($i < 0$ || $i > \text{theSize}$) return null;

// Use tail - i : is closest too

: if ($i < \text{theSize}/2$) {

ptr = this.lst_head;

for ($j=0$; $j \leq i$; $j++$) ptr = ptr.next;

else {

ptr = this.lst_tail;

for ($j=\text{theSize}-1$; $j \geq i$; $j--$) ptr = ptr.prev;

} return ptr;

Example Removing a Node from a DLL

↳ Assume Access to previous & next

```
public char remove(ElemNode p) {
    if (p == lstHead || p == lstTail)
        Need diff calc!
```

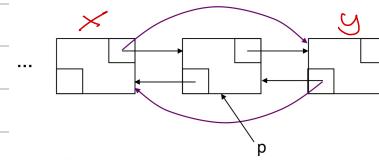
3

```
p.next.prev = p.prev;  
p.prev.next = p.next;  
theSize--;
```

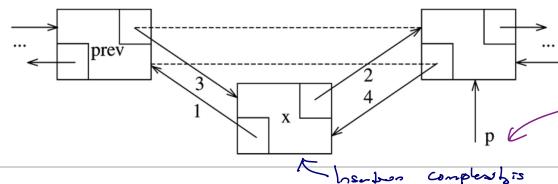
return p.ch;

3

Number of removal references depends on if nodes head or tail or not



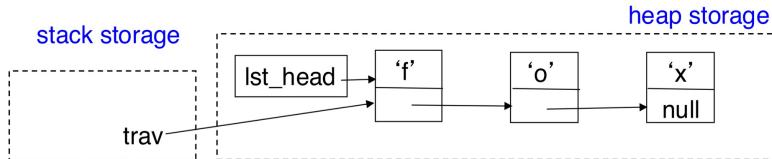
Same idea for inserting a node



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

Always need to find 'p', so shifts at $O(n)$ complexity
however complexity is a constant time operation

More LinkedList Operations



Rest of the class will use singly linked lists usually

General Traversal Support

→ Inefficiency: 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;
    }
}
```

Complexity: O(n)

$O(n^2)$ as each iteration requires a full search to find position

Bypass private fields

Duplicating a Singly LL

- Helper method copy(str)
- Copies all elements through head
- Returns 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 use a traversal node

```
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;
    }
}
```

*Not
Good
Because*

$O(n)$

*Starts from
beginning, goes
to end &*

Only allows you to use numOccur, doesn't let to have to run to methods for any 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); // Constant Time Operations
        while (trav != null) {
            char c = trav.getChar();
            if (c == ch)
                numOccur++;
            trav = trav.getNext(); // Get Node (↳ List) Get Next (↳ Singly Node)
        }
        return numOccur;
    }
}
```

This is $O(n)$ as well

Makes public fields, which you can do too though getters and setters methods.

→ (iii) Double as 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 use of StringNode Objects
- Does not depend on ListString internals

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

- Provides Iteration ability w/o violating encapsulation
- hasNext() - Checks if current node points to a non-null node
- next() - Returns next internal : increments the iterator

It is implemented below as ↪ 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() {...}
    ...
}
```

Instances:

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

Point class ↪ Nested class

None of object to create is for

Internal workings of Iterators

```
public boolean hasNext() {
    return (nextNode != null);
}

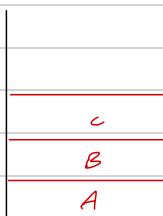
public char next() {
    if (nextNode == null)
        throw new Exception("Falling off the list end");
    char ch = nextNode.ch;
    nextNode = nextNode.next;
    return ch;
}
```

*F "O" "X" ...
F "I" "O" "X" ...
F "O" "I" "X" ...*

Stacks & Queues

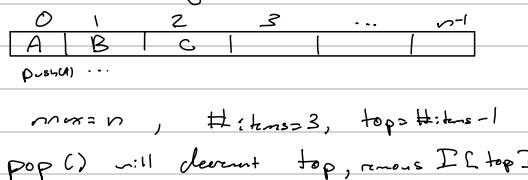
- Linear Data Structures just like arrays and linked lists

Stacks



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

Build with an array



Build with a Linked List



- push(...) should add to Front, changes ptn
- pop() removes element, Head pts to next

Generic Array Stack Class

```
public class ArrayStack<T> {
```

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

```
    ...
    ...
}
```

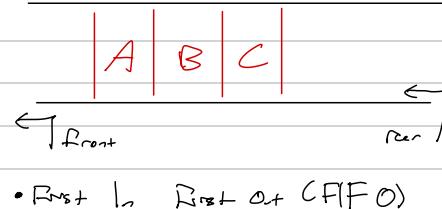
`top++ : use current var, then increment
++top : increment and use that var`

Queue Array Implementation

```
// Add an element to the circular array
public void add(int value) {
    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 IllegalStateException("Array is empty");
    }
    int value = array[head];
    head = (head + 1) % array.length;
    size--;
    return value;
}
```

Queue

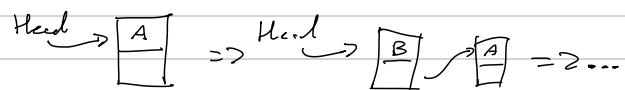


- First In First Out (FIFO)

Everything inserted from tail and comes out of queue from front

- enqueue(...)= add
- dequeue() = remove
- peek(), remove
- insert
- is Empty(), is Full

Linked List Implementation



- Adding => Constant time O(1)
- This makes Removal linear O(n)

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 nodes are as efficient as possible

Summary Q: Emulating Queue using Stacks

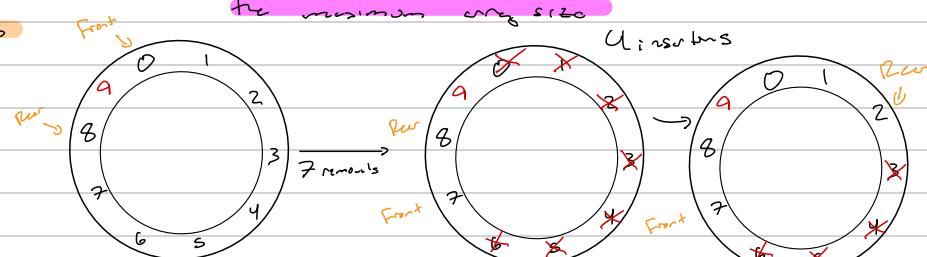
Array Implementation

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



The queue is empty when Front "overcomes" rear:

$$\rightarrow ((rear + 1) \% maxSize) == front$$

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

Basics of Trees

↳ First non-linear data structure

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 unique path to any given node!
- Nodes have data ("payload") consists of one or more fields
- Recursive Data Structure: Each Node in the tree is the root of a smaller tree
 - ↳ Refers to these types of trees as subtrees

Recall: Phone Book

| Data Structure | Search | Insert |
|----------------|-----------------------|----------------------|
| Sorted Array | $O(\log n)$ w/ binary | $O(n)$ due to Shifts |
| Linked List | $O(n)$ using linear | $O(1)$ as LL |

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 a 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 an collection of nodes that is either

(i) empty

(ii) contains a node R (the root tree) that has

- a binary left subtree → choose root (if \neq) connects to it
- a binary right subtree

```
public class LinkedTree {
```

```
private class Node {
```

```
private int key;
```

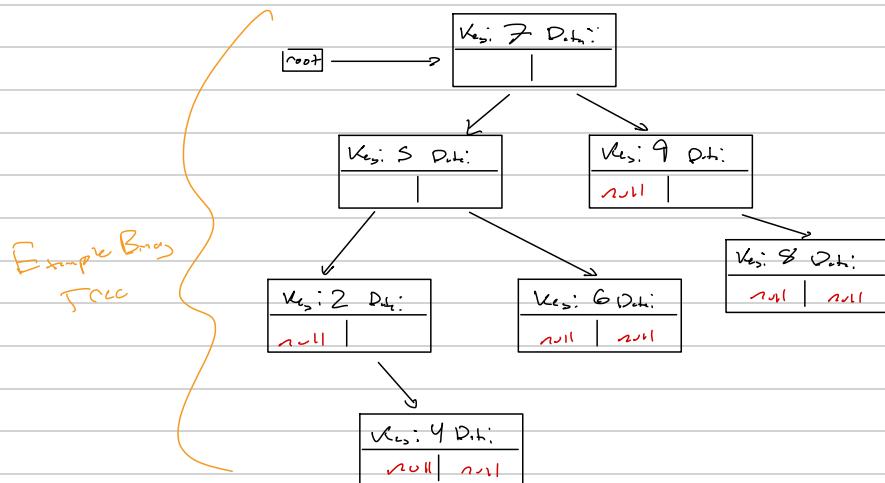
```
private String data;
```

```
private Node left;
```

```
private Node right;
```

```
...  
}
```

```
private Node root;
```



Preorder Traversal

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

~~N - L - R~~

Result from Bn: 7 5 2 4 6 9 8

```

private void myPreorderPrint(Node node)
{
    System.out.print(node.key + " ");
    if (node.left != null)
        myPreorderPrint(node.left);
    if (node.right != null)
        myPreorderPrint(node.right);
}
  
```

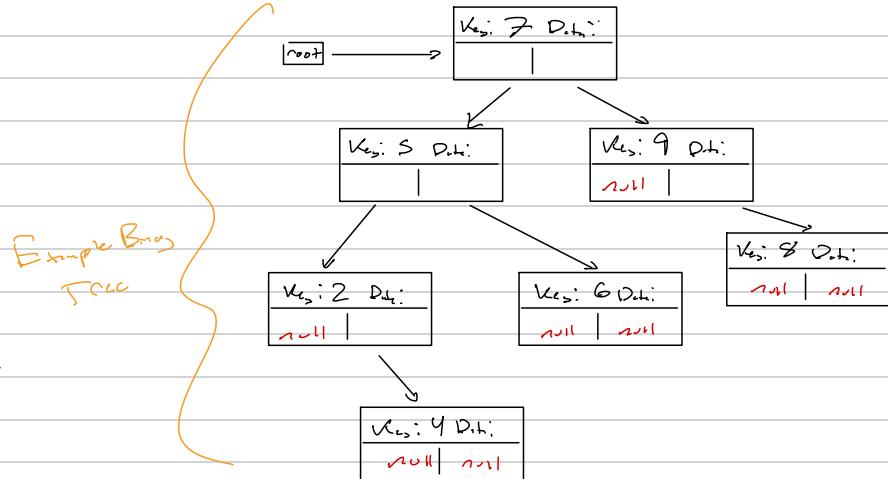
Postorder Traversal

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

L R N

Results: 4 2 6 5 8 9 7

Some 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

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

Results: 7 5 9 2 6 8 4

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

High-Level Implementation

Queue $\xrightarrow{\text{root}}$



- (i) Remove \top
- (ii) Insert left then right node

S 9



- (iii) Remove S

- (iv) Insert S's children

9 2 6

- (v) Remove 9

- (vi) Insert 9's children

2 6 8

Repeat until Queue is empty

Tree Species (Binary & Binary Search Trees)

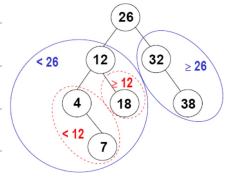
Binary Search Trees

→ 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 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 $K == \text{root}$ nodes \leftarrow do!

else if $K < \text{root.key}$ nodes \leftarrow 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) {
```

Node trav = root; // Data pointer

while (trav != null) {

; if ($\text{trav.key} == K$)

return trav

else if ($\text{trav.key} < K$)

trav = trav.left

else

trav = trav.right

}

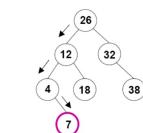
return null;

}

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) Basic case
            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;
}
```



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.

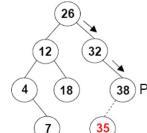
First, we find the node P that will be the parent of the new node:

➤ 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 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);
}
```

Iterative Insertion

Two References

(i) $\text{trav} = \text{parent}$ forms traversal

down to point of insertion

(ii) points stages one above

trav

Node Deletion

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 Child equivalent to saying pointer

(i) Take the parent's 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 (Smallestnode) - call it Y

(ii) Copy Y's key to X, but not its potential child

(iii) Delete Y using Case 1 or Case 2.

Insertion / Deletion

don't Guarantee Balanced Tree

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); // Helper Method
        return removedData;
    }
}
```



```
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.right;
        else
            toDeleteChild = toDelete.left;
        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);
    }
}
```

3 Finding In-Order Successor

Height of Tree
Containing n items
depends on balancing

Efficiency of Binary Search Tree

Search, Insert, Delete all have same time complexity

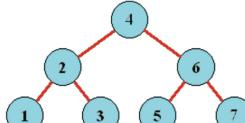
Time Complexity for searching a binary tree

→ Best Case: $O(1)$

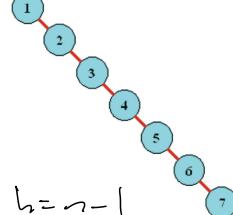
→ Worst Case: $O(h)$ } This is the height of tree

→ Average Case: $O(h)$

• Insert and delete both involve path traversal from root \rightarrow node \rightarrow 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:

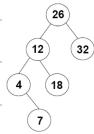
$$\text{Balance}(N) = |\text{height}(N\text{'s right subtree}) - \text{height}(N\text{'s left subtree})|$$

↳ Balance of a Leaf Node

is always 0

Ex: (i) $\text{balance}(\text{node } 26) = |0 - 2| = 2$

(ii) $\text{balance}(\text{node } 4) = |0 - (-1)| = 1$



One Leaf: Height = 0

Empty Tree: Height = -1

AVL Trees (Adelson-Velsky & Landis' G2)

- The balance of all Nodes are never exceeding $|1|$, $\text{balance}(N) \leq |1|$

↳ can either be $-1, 0$, or 1

- This rule must be fulfilled after every insertion & deletion

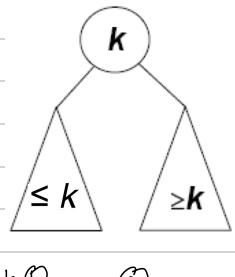
- Steps taken to restore balance must

(i) Maintain the search-tree inequalities

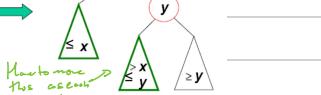
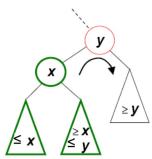
(ii) Have a worst-case time complexity of $O(\log n)$

Ex: Empty Tree $\xrightarrow{\text{Add 1, 2}}$ ① ② $\xrightarrow{\text{Add 3}}$ ② ~~Not~~ ^{Not} ~~Valid~~ $\xrightarrow{\text{Rebalance}}$ ① ②, ③ are ok $\xrightarrow{\text{Rotate ① To Left}}$ ② ③

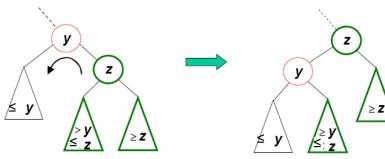
Result: Tree Invariants



Rotation Operations

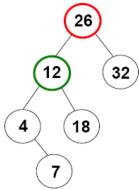


Right Rotation on (around) y

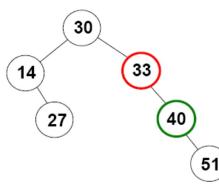


Left Rotation on y

Example Rotations



Right Rotation on Node 26



Left Rotation on node 33

Searching is $\log(n)$, and both insert & delete need to search first so $\log(n)$

Implementation of AVL Trees

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

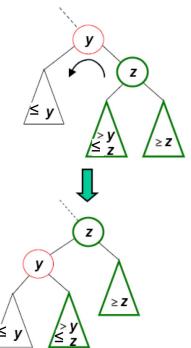
```
public class AVLTree {
    private class Node {
        private int key;
        private String data;
        private Node left; // reference to left child
        private Node right; // reference to right child
        private Node parent; // reference to parent node
        private int balance;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

Rotations Implementation

- Just involves changing pointers

- # ptr upds
- 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

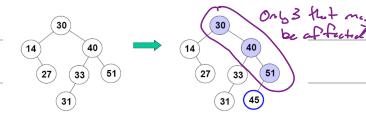
6 Total Ptrs \rightarrow Constant Time Complexity



Insertion into AVL Tree

Remember 2 new fields: Ref to nodes parent & its balance

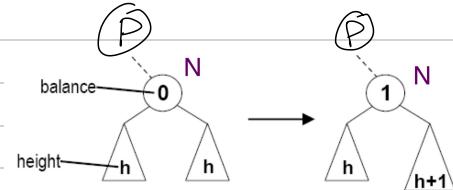
- An insertion can only affect the height values and balance values in the new nodes' ancestors
- First step is inserting node as in a binary search tree.



Change in Balance Assume node N to be affected by insertion

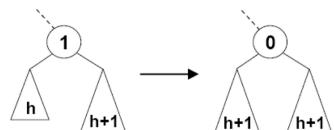
Case 1 N's balance goes from 0 to 1

- N's subtree still within AVL rules
- Height of N has increased
- \hookrightarrow Balance of N's parent $\leftarrow h+1$ Change
- Need to check if N's parent violates the AVL rule
- \hookrightarrow Balance of other ancestors may also be affected



Case 2 N's balance goes from 1 to 0

- Height of the subtree of which N is the root has not changed
- \hookrightarrow 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 of the rotated subtree
- Lead prior to rotation
- \hookrightarrow N's ancestors' balances won't change

\rightarrow **Case 3a:** Inside vs. Outside Insertions on left subtree *Left Subtree of Left Child*

$\begin{cases} \text{balance} < 0 \\ -2 \end{cases}$ Inside: Closer to center line
 $\begin{cases} \text{balance} > 0 \\ -2 \end{cases}$ Outside: Further from center line

- Perform single right rotation about y

\rightarrow **Case 3b:** y's balance to +2, is mirror to 3b *Right subtree of Right Child*

- Perform single left rotation about y

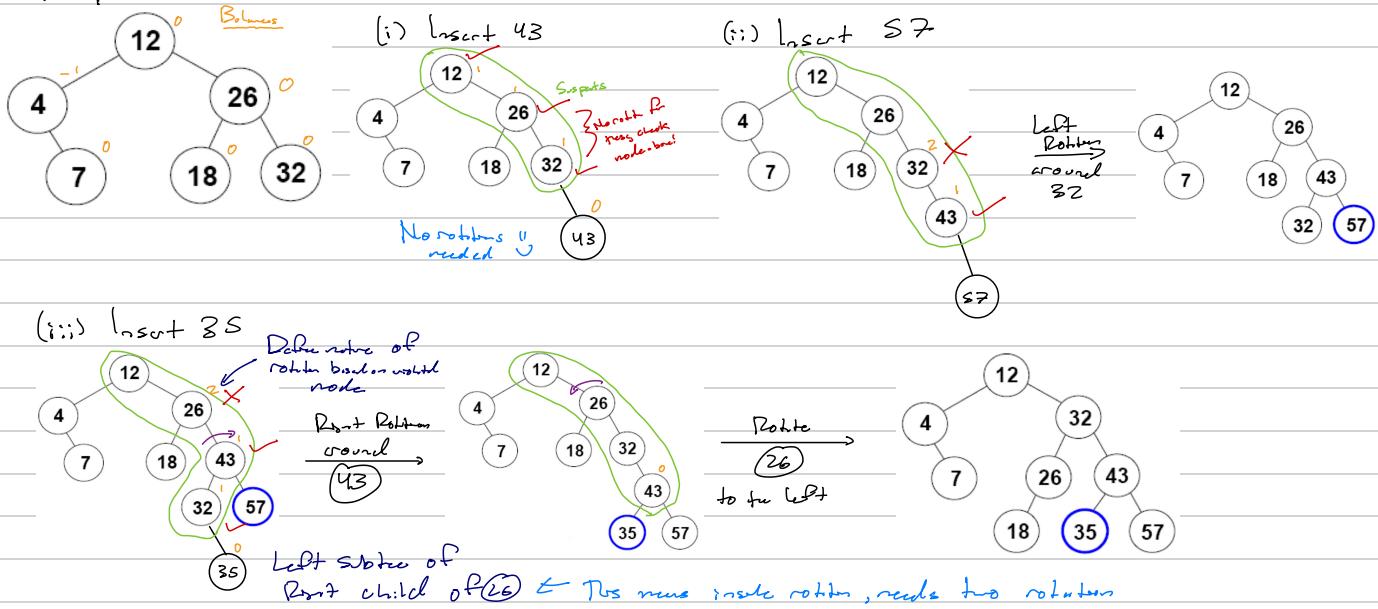
\rightarrow **Case 3c:** we've added a node to the right subtree of y's left child, x, bringing y's balance to -2.

\rightarrow **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 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) \rightarrow Continue up the path to the root
 - If an ancestor's balance was ± 1 , there are now two cases:
 - i) It is now 0 \rightarrow stop
 - ii) It is now $\pm 2 \rightarrow$ perform 1 to 2 rotations to rebalance tree (cases 3a-3d)
 - \hookrightarrow 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 completed

- 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

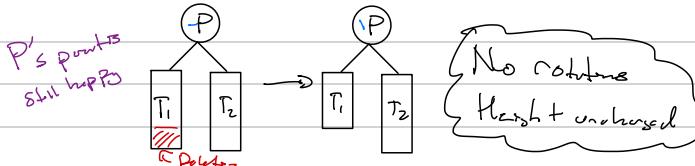
Algorithm: (i) shorter initialized as true

- ↑
- (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

After standard removal from a binary tree

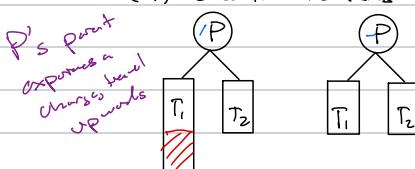
Case 1: Until shorter flag becomes false or you reach root

- (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

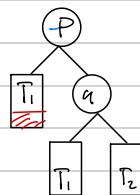
- (i) Change the balance factor of p to equal
- (ii) shorter removes tree



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 removes tree.

→ 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 removes tree

Efficiency of AVL tree node deletion

$O(\log_2 n)$ → worst case requires full path up and down with nested 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

- Having more keys per node reduces traversal looks at unneeded data

Why not store multiple keys at each node

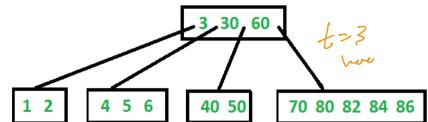
B-Tree

- Reduces the number of disk accesses
- All levels are at the same level
- Defined by its minimum degree t & Input Parameter
- Every node except root must contain at least $t/2$ keys
↳ Root can contain minimum 1 key
- AVL nodes (including root) must contain a minimum of $2t-1$ Keys
- Number of children of a node is equal to number of keys in it plus 1

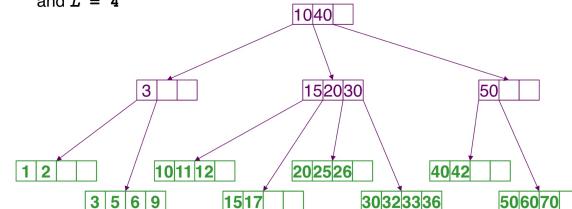
Alternate Definition

- 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 I 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 $\lceil I/2 \rceil$ and I keys
- all leaves are at the same depth

Example B-Tree



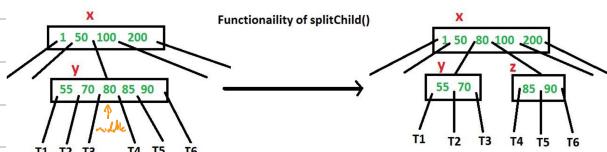
B-Tree with $M = 4$
and $I = 4$



Everything is logarithmic in tree

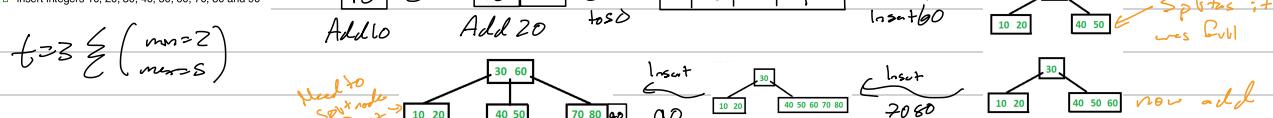
Insertion into B-trees

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



Example

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

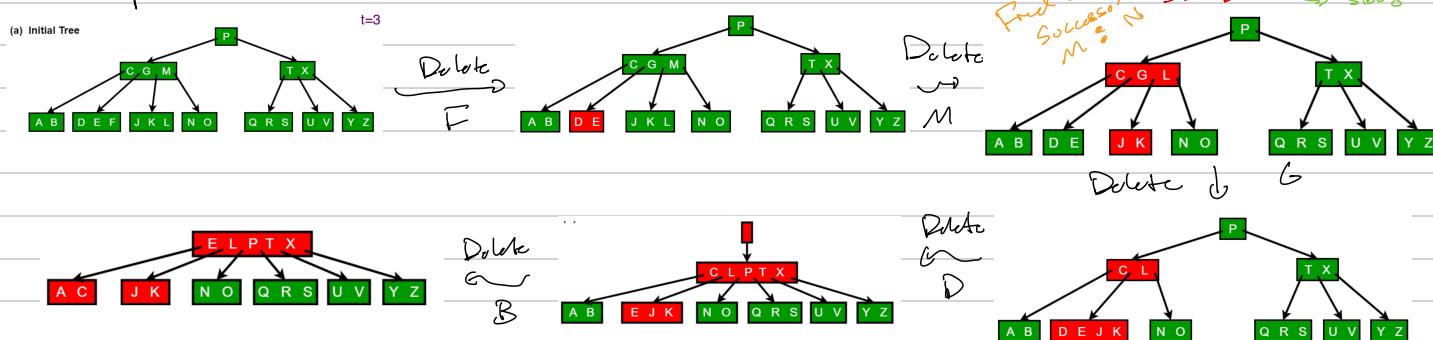


Splits if was full

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 the minimum number of keys

Example



Final Exam Notes



Heaps & Priority Queues

Queues

- Items added to rear and removed from front
 - ↳ FIFO (first comes element at head)
- Operators: Insert, remove, peek, isEmpty, size

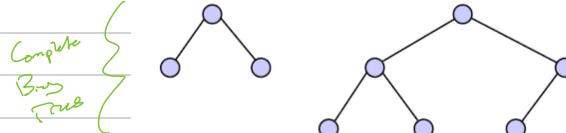
Priority Queues

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

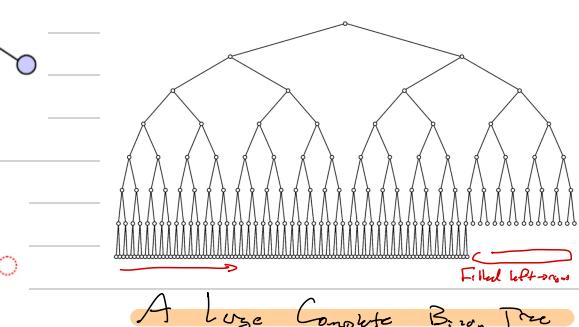
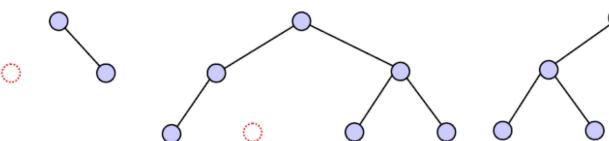
Structure types of Binary Trees

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

Complete Binary Tree



Not complete (○ = missing node):



A Large Complete Binary Tree

Array Representations of a Complete Binary tree

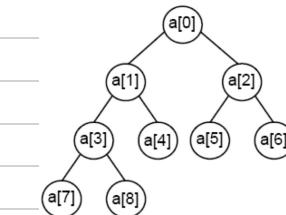
- Nodes of the tree are stored in the order in which they would be visited by a level-order traversal

Can construct array from binary tree from arr.
Consider level-order and given complete tree

- The root Node is $a[0]$

↳ Given the node $a[i]$

- Left child is $a[2^* i + 1]$
- Right child is $a[2^* i + 2]$
- Parent is $a[(i-1)/2]$, for $i > 0$



↔ $a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]$

Heaps for Priority Queues

- Heaps are a complete binary tree

↳ Constraint: A node has ≥ 2 kids of children ($f = \infty$)

- Largest value is always @ tree root

- Smallest value can be in any leaf node (no guarantee)

- For min-at-top heaps, any internal 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 Comparable implements Comparable<Comparable>

- May built-in classes have a compareTo() method?

public int compareTo(Comparable other)

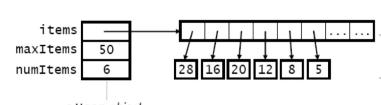
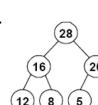
- Example on Point: the object in the heap is

referred to external 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;
}
```



a Heap object

1. Remove 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

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 \geq children

2. Remove an Item (the Greatest)

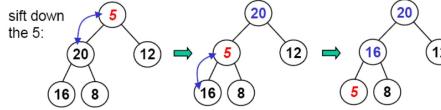
Remove and return the item in the root node.

- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node \geq 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 \geq its children (or it's a leaf)
- return the largest item

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



Example Sift Down



Implementation of Sift Down Method

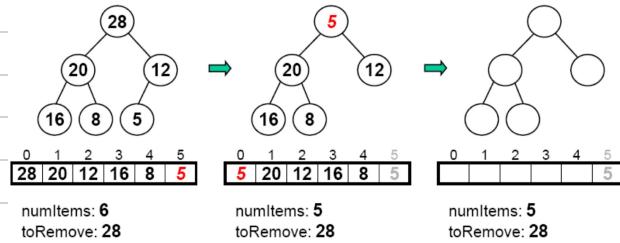
```
private void siftDown(int i) { // Input: the node to sift
    T toSift = items[i];
    int parent = i;
    int child = 2 * parent + 1; // Child to compare with; start with left child
    while (child < numItems) {
        // If the right child is bigger than the left one, use the right child instead.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

Once you have sift down method, remove is trivial

`public T removeMax()`

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



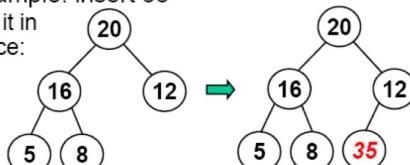
2. Insertion into Heaps

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

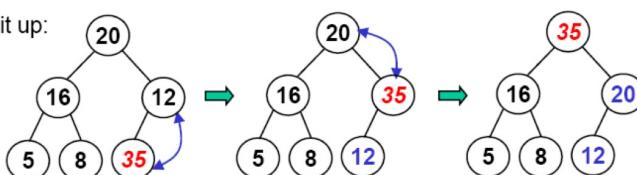
Algorithm: & Quick Example

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

Example: insert 35
put it in place:



sift it up:



`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

• Sift Up & Sift Down

$\rightarrow \log N$

$\rightarrow O(\log N)$ in worst case

• Insert / Remove are fast $O(\log N)$

• Search is slow $O(N \log N)$

Building a Heap

A Naive Algorithm

- Take N items from any ordered batch a heap
- For each item, insert it into a heap which initially empty

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

$O(N \log N)$

An Efficient Algorithm: Building the Heap := Place

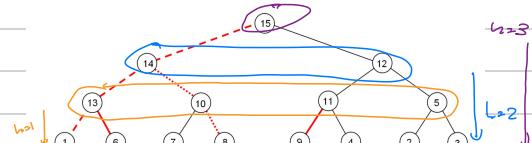
- 1) Position = $(\text{numItems} - 2)/2 + j$
- 2) S: Pt Down item at $\text{arr}[j]$ + Position
- 3) Decrease Position by 1
- 4) Repeat Steps 2, 3, 4 until position is 0

```
public void buildHeap() {
    for( L: i = (\text{numItems} - 2)/2; i >= 0; i-- )
        S: Pt Down( L )
}
```

Order of happens for about half of total nodes

$O(\frac{N}{2} \log N)$

Step 1 $O(N \log N)$



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

Proof

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

| |
|-------------------------|
| 1 node at height 0 |
| 2 nodes at height 1 |
| 4 nodes at height 2 |
| 8 nodes at height 3 |
| 16 nodes at height 4 |
| 32 nodes at height 5 |
| 64 nodes at height 6 |
| 128 nodes at height 7 |
| 256 nodes at height 8 |
| 512 nodes at height 9 |
| 1024 nodes at height 10 |

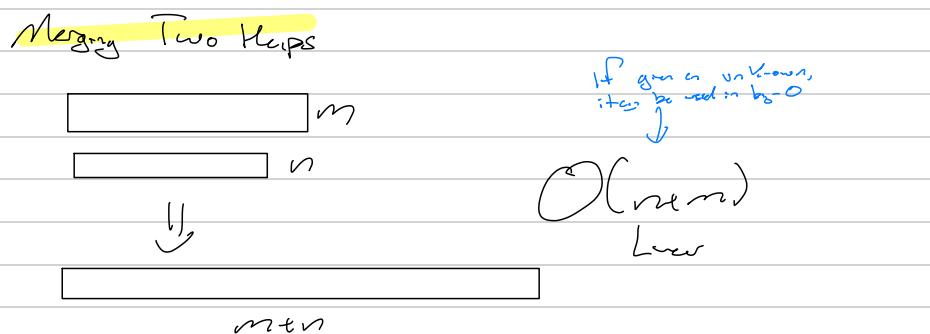
Total height of all nodes = $h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \dots + 2^{h-1}(h - (h - 1)) = S$

$$S = \frac{2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \dots + 2^h(h - (h - 1))}{(h + 2h - 2 + 4h - 8 + 8h - 24 + \dots + 2^{h-1}h - 2^{h-1}(h - 1))}$$

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

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

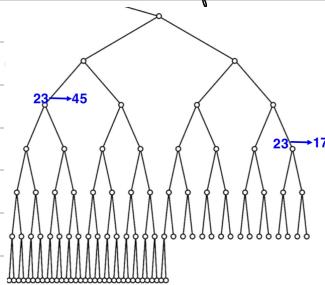
Thus the actual number of nodes is within a factor of 2 of N. Hence, $S = O(N_{\text{actual}})$



Runtime Complexity

| | |
|-------------|-------------|
| findMax | $O(1)$ |
| removeMax() | $O(\log N)$ |
| insert() | $O(\log N)$ |
| buildHeap() | $O(N)$ |
| merge() | $O(N)$ |

Exercise: Update an Item



Run Time

```
public T update(int i, T oldItem) {
    T oldItem = items[i];
    if (item.compareTo(oldItem) == 1)
        S: Pt Up(i);
    else
        S: Pt Down(i);
    return oldItem;
}
```

?

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 Algorithm:** Given an arbitrary array, re-position the items in the array in desired order
- **Name Algorithm:** $O(n^2)$ as it performs a full linear scan to find next item. $O(6)$ 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

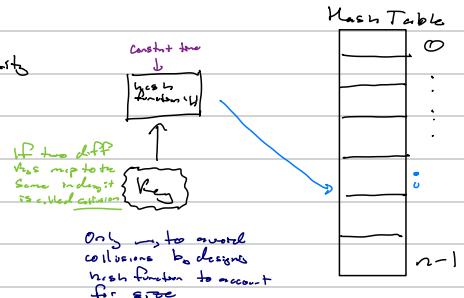
Heap is nice as
finding max is constant
for max-at-top

Hashing

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

Hash Tables

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



Example: Want to store the grades of students in the class ↗ In the real world Keys are semantically rich

→ give each student a unique Key (integer from 0-180)

→ Store student records in array and can then perform constant time operations

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

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

(ii) Use a hash functions

• Converted "map"s Keys into array indices

→ Domain: the Keys

→ Range: integers in $[0, \text{size}]$

• For this example, we want $\text{key} \rightarrow \text{index} = \text{SSN} \bmod 180$

→ Greatly reduces size complexity

→ Problem: multiple Keys can map to the same index

↪ Called collisions: 180-00-0001 and 180-00-0181 both have index 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 problem

Example 2: String as Key

→ Keys = character strings of lower-case letters

→ Hash function:

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

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

□ Might limit character length to 20 to reduce size requirement

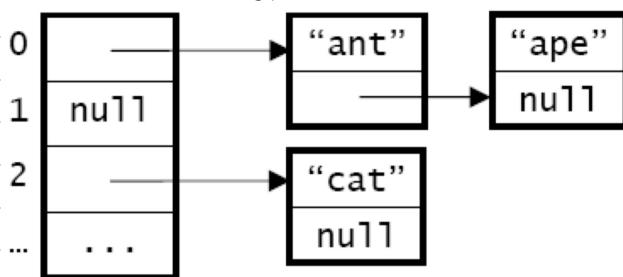
↪ Maximum sum is gonna be $127 * 20 = 2540$ (127 is max byte of a lower character)

Requirements For Good Hash Functions

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

Hashing Collisions \Leftarrow we will assume that clients know we are very specific about collisions

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



- Can also use Linear Probing to look for open positions

\rightarrow Advantage: If there is an open position, linear probing will eventually find it

\rightarrow "Clusters" of occupied positions develop

\square increases the length of subsequent probes

\square As load factor increases, both search and insert times look increasingly linear

- Can try to avoid linear probing disadvantages with Quadratic Probing

\rightarrow Advantage: reduces clustering

\rightarrow Disadvantage: There is a chance to miss an empty open position, fails to insert into table

TLCriteria: If the load factor is less than 30% and the table size is prime, no reiterations 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

$\rightarrow h_1$ computes hash code

$\rightarrow h_2$ computes the increment for probing

~~Sequence:~~ $h_1, h_1+h_2, h_1+2 \cdot h_2, \dots$

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

Reinsert items with Open Addressing

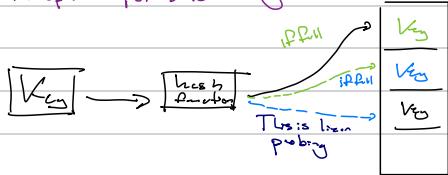
- If the item is not in the table, you have to scan the whole table (linear)

\hookrightarrow To avoid this, we use a 'removed' flag (boolean) allows us to avoid linearly

- Use 3 states: occupied, empty, removed

Hashing Implementation

Recap: Open Addressing



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

Constructors

```
public HashTable(int size) {  
    table = new Entry[size];  
    tableSize = size;  
}  
  
private Entry(String key, String etymology) {  
    this.key = key;  
    this.etymology = etymology;  
    removed = false;  
}
```

Probing in Infinite Loop

for double hashing:

- $(h1 + n \cdot h2) \% n = h1 \% n$
- $(h1 + (n+1) \cdot h2) \% n = (h1 + n^2 + h2) \% n = (h1 + h2) \% n$
- $(h1 + (n+2) \cdot h2) \% n = (h1 + n^2 + 2 \cdot h2) \% n = (h1 + 2^2 \cdot h2) \% n$
- ...

for quadratic probing:

- $(h1 + n^2) \% n = h1 \% n$
- $(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1) \% n$
- $(h1 + (n+2)^2) \% n = (h1 + n^2 + 4n + 4) \% n = (h1 + 4) \% n$
- ...

Mock use of findKey()

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

Both use findKey()
and check its output

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

Implementations

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

- For an empty position, $\text{table}[i] \neq \text{null}$
- For a removed position, $\text{table}[i] \neq \text{null}$ will refer to an Entry object whose removed field is true
- Data is not actually removed just simply overwritten.

Finding an Open Position

```
Helper method  
to use double •  
hashes 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) {  
        i = (i + 1) % tableSize; // Update probe position  
        iterations++;  
        if (iterations >= tableSize) return -1;  
    }  
    return i;
```

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

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
```

Find the Position of A Key

→ Different from probe() as it returns position of 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 + 1) % tableSize;  
        iterations++;  
        if (iterations >= tableSize) return -1;  
    }  
  
    return -1;
```

↳ Different for linear probing
↳ Different for quadratic probing

Hashing Implementation Issues

Potential Issues

- 1) Lots of items: $\lambda \approx 1$ ↪ can copy to another table and copy all non-null & non-redundant entries
 \hookrightarrow Load Factor = $\frac{\# \text{of entries}}{\text{size of table}}$
- 2) Lots of deletions: small λ
 - a) Insertion should be very fast as many pointers have removed entries
 - b) Search and remove will be affected negatively

Recovery Strategies

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(?) ?
    }
}
```

BRBing of Hash Tables

- Best case: search O(n), worst case: O(n)
- Open address: worst case, O(n): max size of table
- Separate chaining: worst case O(n); n = number of keys
- Performance worsens as load factor increases
 - $\lambda < 1$ 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 k-th largest element or sorted array of elements
- Difficult to create good hash function

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"

Substring Pattern Matching

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

Brute Force: Substring contains in letters, and

total text is length n

$O(nm)$ complexity

Using hashing: Compute hash of p to
 hash of groups of $p.length()$ in t .

$O(n)$ complexity if implemented correctly

Can also be used for genome matching

Hash Functions for Strings

- $h_b = (a_0b^{n-1} + a_1b^{n-2} + \dots + a_{n-2}b^1 + a_{n-1}) \bmod (\text{tableSize})$
- a_i is encoding of the i -th character
- b is a constant → *Base parameter* $b=31$
- All characters contribute
- Contribution of a character depends on its position

• Takes n multiplications as string length ↑

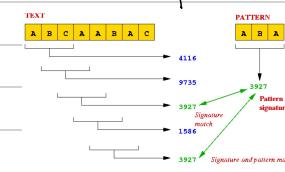
Use Horner's
Method for efficiency

example: $101'31^2 + 97'31 + 116 = (101'31 + 97)'31 + 116$
 $101'31^2 + 97'31 + 116'31 + 116 = ((101'31 + 97)'31 + 116)'31 + 116$
 $a_0b^{n-1} + a_1b^{n-2} + \dots + a_{n-2}b^1 + a_{n-1} = (\dots((a_0 + a_1)b + a_2)b + \dots + a_{n-2})b + a_{n-1}$

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

well-known &
widely adopted
function

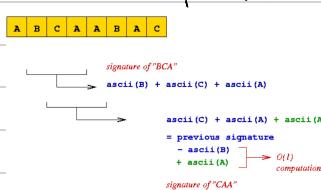
Poor Implementations



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

Takes $O(nm)$
 still as $O(nm)$
 required to get
 each subtree of size
 $O(n)$

Better Implementations



Observation: two successive substrings differ by only two characters.

\Rightarrow next signature can be computed quickly ($O(1)$).

Use signature
 of old subseq
 to get new one.
 Reduces some
 computations to $O(1)$

Simple Sorting Algorithms

- Array \rightarrow Inplace
- We want to do everything in place
 - \hookrightarrow minimize storage complexity
 - minimize total # of comparisons (C) and moves (M)

Method 1: Selection Sort

- Consider positions from left \rightarrow 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); // Helper method
        swap(arr, i, j);
    }
}
```

$a_0 = 1 \quad \{ \quad t = a_0$

$a_1 = 4 \quad \} \quad a_0 = a_1$

$a_2 = 6 \quad \} \quad \text{temp variables are OK}$

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;

} static & returns positions

Time Complexity

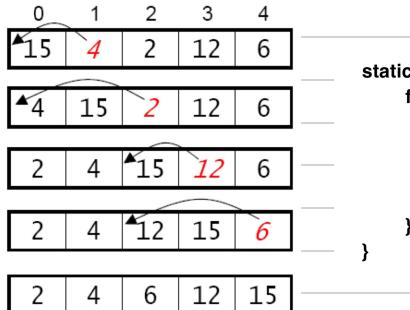
$$\mathcal{O}(n^2)$$

Sums always constant

• Choices $n, n-1, n-2 \dots$

Method 2: Insertion Sort

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



```
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

smaller element to left

| Time Complexity | | |
|-----------------|-----|-----|
| C | m | 1 |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |
| 4 | 4 | 4 |

For $|5|4|3|2|1|$

$\mathcal{O}(n^2)$ Still \uparrow

Best Case \gg Best Case Selection

Worst Case \ll 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 vs. length of arr

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-case 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 $9 \cdot 4^i - 9 \cdot 2^i + 1$ and $4^i - 3 \cdot 2^i + 1$
- 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 need to,

$O(n \log n)$ vs $O(n^2) < O(n \log n)$

For all C. C. nothing vlog n

with help sort, so C. is better?