## Abstract Data Types (ADT)

**List**

**Stack**

**Queue**

**Data Abstraction:** Separation of a data type's logical properties from its implementation.
-**Logical Properties**
--What are the possible values? What operations will be needed?
-**Implementation**
--How can this be done in Java, C++, or any other programming language?

**ADT** is a set of objects together with a set of operations. A data type that does not describe or belong to any specific data, yet allows the specification of organization and manipulation of data.

## List Operations

| | |
|---|---|
| Find | (First occurrence) |
| Insert | |
| Remove | |
| FindKth | |
| MakeEmpty | |
| PrintList | |

## List Implementations

Simple Array

Simple (Singly) Linked List

## Stack - LIFO (Last In First Out) List Operations

Push

Pop

Top (Peek)

MakeEmpty

## Stack Implementations

| | |
|---|---|
| Array | |
| LinkedList | Operations take **constant** time. Size can grow/shrink easily. |

**Overflow**: When element count in an array exceeds array size.
**Underflow**: Pop from an empty stack.

**Evaluate infix expressions**, 2 stacks algorithm (Dijkstra):
--*Value*: Push onto the value stack.
--*Operator*: Push onto the operator stack.
--*Left parenthesis*: Ignore.
--*Right parenthesis*: Pop operator and two values; push the result of applying that operator to those values onto the operand (value) stack.

## Queue - First In Last Out List Operations

Enqueue

Dequeue

MakeEmpty

## Queue Implementations

Circular Array (Circular Queue)

Linked List

*Examples:*
-Calls to a call center
-Jobs in the printer
-Network operations on routers
-CPU usage queues

## Trees

| | |
|---|---|
| Tree | Collection of **nodes** such that: |
| Root | Unless empty, trees have a **root**. |

## Trees (cont)

| | |
|---|---|
| Subtrees | Remaining nodes are partitioned into trees themselves, called **subtrees**. Each subtree is connected by a **directed edge** from the **root**. |
| Degree | Number of **subtrees** of a node. |
| Leaf / Terminal Node | Node with **degree** 0. |
| Parent | |
| Child | |
| Ancestors | |
| Path from $node_1$ to $node_k$ | |
| Depth | Length of the unique **path** from **root** to node. |
| Height | Length of the **longest** *downward* path from the node to a **leaf**. |
| Height of a Tree | **Height** of the **root**. |

For its implementation, a node can hold:
-Its first child.
-Its next sibling.
Thus siblings would be held as a **linked list**.
Without parent/previous sibling information, each node holds only 2 references.

## Binary Tree

Each node can have at most **2** children.

| | |
|---|---|
| **Full BT** | When each node has 2 or 0 children. |
| **Perfect BT** | It's full and each leaf has the **same depth**. |

## Tree Traversal

| | |
|---|---|
| Preorder | Parent first. |
| | Visit root. Traverse left subtree. Traverse right subtree. |
| Postorder | Parent last. |
| | Traverse left subtree. Traverse right subtree. Visit root. |
| Inorder | Left-Parent-Right |
| | Traverse left subtree. Visit root. Traverse right subtree |

## Search Tree ADT - Binary Search Tree

Provides **inorder** traversal.

**Average** case: **Depth** of all nodes on average log(N)

**Balanced** BST maintains all operations at **h=O(logN)** time

## AVL (Adelson-Velskii and Landis) Tree

| |
|---|
| It's a BST. |
| Height of the left subtree and the right subtree differ by **at most 1**. |
| Empty tree has height **-1**. |
| Balancing After Insertion |

| | |
|---|---|
| Left-Left | Single Right Rotation |
| Right--Right | Single Right Rotation |
| Left-Right | Double Left-Right Rotation |
| Right-Left | Double Right-Left Rotation |

-
-
-
-
-
-
-
-
-
-
-
-

## Algorithm Analysis

| | |
|---|---|
| *Problem Solving: Life Cycle* | |
| ***Problem Definition*** | |
| Functional Requirements | Calculate the mean of n numbers etc. **What** should the program do? |
| Nonfunctional Requirements | Performance Requirements: How fast should it run? etc. **How** should the program do? Can be considered as **Quality Attributes** |
| ***Algorithm Design*** | |

## Algorithm Analysis (cont)

| | |
|---|---|
| Algorithm | A clearly specified set of instructions for the program to follow. |
| **Knuth's Characterization** (5 properties as requirements for an algorithm) | |
| **~Input** | 0 or more, **externally produced** quantities |
| **~Output** | 1 or more quantities |
| **~Definiteness** | Clarity, precision of each instruction |
| **~Finiteness** | The algorithm has to stop after a finite amount of steps |
| **~Effectiveness** | Each instruction has to be basic enough and feasible |
| ***Algorithm Analysis*** | |
| Given an algorithm, will it satisfy the requirements? | |
| Given a number of algorithms to perform the same computation, which one is "best"? | |
| The analysis required to estimate the "resource use" of an algorithm | Space and Time Complexity |
| ***Implementation*** | |

## Algorithm Analysis (cont)

| | |
|---|---|
| ***Testing*** | |
| ***Maintenance*** | Bug fixes, version management, new features etc. |

## Space Complexity

| | |
|---|---|
| Space Complexity | The amount of memory required by an algorithm to run to completion |
| Fixed Part | The size required to store certain data/variables, that is **independent of the size** of the problem, eg. name of the input/output files, |
| Variable Part | Space needed by variables, whose size is **dependent on the size** of the problem. |
| $S(P)=c+S_p$ | c = constant, $S_p$ = instance characteristics which depends on a particular instance |

## Pseudocode

| |
|---|
| ***Control Flow*** |
| if... then... [else...] |
| while... do... |
| repeat... until... |
| for... do... |
| Indentation instead of braces |

## Pseudocode (cont)

| | |
|---|---|
| ***Method Declaration*** | |
| Algorithm Method (arg [, arg...]) | |
| Input... | |
| Output | |
| ***Method Call*** | |
| var.method(arg [, arg...]) | |
| ***MethodReturn Value*** | |
| return expression | |
| ***MethodExpressions*** | |
| ← | Assignment ( = in code) |
| = | Equality Check (== in code) |
| Superscripts etc. mathematical formatting allowed | |
| ***Experimental Approach*** | Can't always use |
| ***Low Level Algorithm Analysis*** Using **Primitive Operations** | **Make an addition = 1** operation **Calling a method or returning from a method = 1** operation **Index in an array = 1** operation. **Comparison = 1 operation**, etc. |
| Method: Count the primitive operations to find O(f(n)) | |
| **Growth rate** of the running time T (n) is an **intrinsic property** of an algorithm | Not dependent on hardware. |

## Pseudocode (cont)

| | |
|---|---|
| **Asymptotic Notation** | Big-Oh, Big Omega, Big Theta, Little-Oh |
| ***Characterizing Algorithms As A Function Of Input Size*** | |
| Solving **Recursive** Equations | |
| by **Repeated Substitution** | T(N)=T(N/2)+c=T(N/4)+c+c=...=$T(N/2^k)$+kc, choose k = logN, T(N)=T(1)+clogN=Θ(logN) |
| by **Telescoping** | T(N)=T(N/2)+c |
| | T(N/2)=T(N/4)+c |
| | ... |
| | +__________-_______ (cancelling opposite terms) |
| | T(N)=T(1)+clogN=Θ(logN) |

-
-
-
-
-
-
-
-
-
-
-
-
-
-

## Dictionary ADT

A collection of (key, value) pairs such that each key appears at most once

| | |
|---|---|
| *Idea*: Use the key as the **index** information to reach the key | **Key-Value Mapping** |

## Dictionary Operations

Find

Insert

Remove

Note: No operations that require ~~ordering information~~

## Dictionary Implementations

Lists

Binary Search Trees

Hash Tables

## Hash Table

| | |
|---|---|
| ***Collision Resolving*** | |
| Separate Chaining | (Open Hashing) |
| Open Addressing | (Closed Hashing - Probing Hash Tables) |
| --Open Hashing | Collisions are stored outside of the table |
| --Closed Hashing | Collisions are stored at another slot in the table |
| **Separate Chaining** | |
| Each cell in the hash table is the **head** of a **linked list** | |
| Elements are stored in the hash-specified linked list | |
| Records in the linked list can be ordered by: | order of insertion, key value, frequency of access |
| **λ = Load Factor** | |
| λ ≈ n/TableSize | |

## Hash Table (cont)

| | |
|---|---|
| insert, find, remove take O(1+λ) on average | |
| **Closed Hashing (Probing Hash Tables)** | |
| $h_i(x)$ = (hash(x) + f(i)) mod TableSize, f(0)=0 | |
| f = **Collision Resolution Strategy** | |
| --Linear Probing | |
| --Quadratic Probing | |
| --Double Hashing | |
| ***Linear Probing*** | **f(i)=i** (linear function of i) |
| Primary Clustering | n<TableSize guarantees finding a free cell |
| | Insertion time can get long due to **blocks of occupied cells** are formed |
| *Primary Clustering* : Any key that hashes into the cluster - even if the keys map to different values- will require several attempts to resolve collusion and then it will be added to the cluster. | |
| Worst Case : find, insert | O(n) |
| Deletion requires **Lazy Deletion** to not mess up the table | After many deletions may **reorganize** the table |
| ***Quadratic Probing*** | **$f(i)=i^2$** (quadratic function of i) |

## Hash Table (cont)

| | |
|---|---|
| | Eliminates **primary clustering** |
| Unless if **TableSize prime** and **λ<1/2**, cannot guarantee finding empty cell | |
| **Secondary Clustering** | Elements that hash to the same position **probe** to the same alternative cells, clustering there |
| ***Double Hashing*** | $f(i)=i\cdot hash_2(x)$ (includes another hash function) |
| Example: $hash_2(x)$=R - (x mod R), where R is a prime < TableSize | |
| **Rehashing** | When λ too big, make bigger TableSize and **rehash** everything. Takes O(n) but happens rarely.. |
| ***ALL* Closed Hashing** cannot work with **λ = 1** | |
| --**Quadratic probing** can fail if **λ > 0.5** | |
| --**Linear probing** and **Double hashing** are slow if **λ > 0.5** | |
| **Open Hashing** becomes slow once **λ > 2** | |

Quadratic Probing Proof:
-
-
-
-
-
-
-
-

## Cuckoo Hashing

| | |
|---|---|
| 2 hash tables | |
| Only insert at the 1st table | Move the value in the 1st table if collision |

May cause cycles but if λ<0.5, cycle probability low. Still possible, so specify maximum iteration count after which you rehash.

## Time Complexity

| | |
|---|---|
| O(f(N)) = T(N) | T(N) ≤ cf(N) when N ≥ n0. **Upper-bound** |
| Ω(g(N)) = T(N) | T(N) ≥ cf(N) when N ≥ n0. **Lower-bound** |
| Θ(h(N)) = T(N) | T (N) = O(h(N)) and T (N) = Ω(h(N)) **Tight-bound (Exact)** |
| o(p(N)) = T(N) | T(N) < cp(N) **Strict Upper-bound** |
| f(N) is o(g(N)) if it's O(g(N)) but not Θ(g(N)) | |
| O(1) | constant |
| O(logN) | logarithmic |
| $O(\log^2 N)$ | log-squared |
| O(N) | linear |
| $O(N^2)$ | quadratic |
| $O(N^3)$ | cubic |
| $O(2^N)$ | exponential |

f(n)≤O(g(n)) is the wrong usage. You need to say f(n) **is (=)** O(g(n))

## Priority Queues (Heaps)

For applications that require a sorted (but not fully sorted) order of procession of keys.

## Priority Queues (Heaps) (cont)

Jobs sent to a printer, Simulation Environments (Discrete Event Simulators)

## Priority Queue Operations

insert

deleteMin

## Priority Queue Implementations

| | |
|---|---|
| Simple (Singly) Linked List | Insert O(1), deleteMin O(n) |
| Sorted Linked List | Insert O(n), deleteMin O(1) |
| Binary Search Tree (BST) | **Θ(logn) average** for insert and deleteMin |
| Binary Heap | Can implement as a **single array**, doesn't require ~~**links**~~, **O(logn) worst-case** for insert and deleteMin |
| d-Heaps | Parents can have **d** children |
| Leftist Heaps | |
| Skew Heaps | |
| Binomial Queues | |

## Binary Heap

*Classic* method for **priority queues**

| | |
|---|---|
| Simply called **Heap** | (Different from heap in ~~**dynamic memory allocation**~~) |

***Structure Property***

## Binary Heap (cont)

| | |
|---|---|
| **Heap** is a **complete binary tree** | **Complete binary tree** is a BT filled completely, except maybe for the **bottom row**, which is filled from **left-to-right** |

**Array implementation:**

For any element in position **i**:

--**Left child** is in position **2i**

--**Right child** is in position **2i+1** (after left child)

--**Parent** in position ⌊**i/2**⌋

***Order Property***

Every ***parent*** is **smaller than or equal to** its children, so **findMin** is **O(1)**

A **Max Heap** is the reverse, allowing constant access to the max element

## Binary Heap Methods

| | |
|---|---|
| **insert** | Insert element in **position 0**. Find its possible position, make an empty node, then **percolate up** until the **new element** can be put into the empty position. |
| Worst case runtime is O(logn) | Average runtime is **constant** |

## Binary Heap Methods (cont)

| | |
|---|---|
| **deleteMin** | Delete/make root empty, put the **last element** into array **position 0**, **percolate down** until the **last element** can be put into the empty position. |
| Worst case runtime is O(logn) | Average runtime is also O(logn) since an element at the bottom is likely to still go down to the bottom. |
| ***Building a Heap*** | **Iterating insertion**: O(NlogN) worst case. O(N) on average. |
| **buildHeap** | First fill the leafs. ~Half of the tree is filled already. Then, as you place the next elements, the subtrees will all be valid heaps - do **percolate down**. Thus O(logHeight) operations for each node. |

## Binary Heap Methods (cont)

| |
|---|
| There are **more high depth nodes** than **high height nodes** in a heap thus buildHeap is a faster method, **O(N) worst case** |
| Sum of all **heights**: |
| S = logN + 2(logN - 1) + ... + $2^k$(logN - k), k=logN |
| 2S = 2logN + ... + $2^{k+1}$(logN - k) |
| 2S-S = 2+$2^2$+...+$2^k$ - logN since k=logN thus $2^{k+1}$(logN-k)=0 |
| S = $2^{k+1}$-2-logN |
| S = 2N - logN - 2 thus O(N) |
| Sum of all depths: |
| S = 0 + 2·1 + $2^2$·2 + ... + $2^{logN - 1}$·($log^N$ - 1) |
| S = NlogN-2N+2 thus O(NlogN) |

## Priority Queue Applications

| | |
|---|---|
| Operating System Design | |
| Some Graph Algorithms | |
| Selection and Sorting Problems | Given a list of N elements, and an integer k, the selection problem is to find the kth smallest element. |
| | Take N elements, apply buildHeap, do deleteMin k times. |

## Priority Queue Applications (cont)

| | |
|---|---|
| | If k=N, we get a sorted list. This is heapSort which is O(NlogN) |
| Discrete Event Simulation | Instead of experimenting, put all events to happen in a queue. Advance clock to the next event each **tick**. Events are stored in a heap to find the next one easily. |
| **--Tick** | A quantum unit |