COL106 - Data Structures and Algorithms

Minor Revision

Linear Probing

(uses less memory than chaining, but slower)

causes clustering

Example: $h(x) = x \bmod 13$

$n = 12$
$m = 13$

$n \rightarrow$

Insert: 18, 41, 22, 44, 59, 32, 31, 73

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	72	

Linear Probing

Example: $h(x) = x \bmod 13$

Insert: 18, 41, 22, 44, 59, 32, 31, 73

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Delete 32

0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

Example: $h(x) = x \bmod 13$

Insert: 18, 41, 22, 44, 59, 32, 31, 73

		41			18	44	59		22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

lookup → ignore X
insert → replace X

Delete 32

		41			18	44	59	X	22	31	73	74
0	1	2	3	4	5	6	7	8	9	10	11	12

- Rehash if there are too many

Double Hashing

Quadratic probing:
$(h(k) + i^2) \bmod m$

Use two functions: primary hash $h(k)$
secondary hash $d(k)$

- table size (m) must be prime.

$d(k)$ cannot take 0 values.

handles collision by placing item in first available cell in
$(i + j\,d(k)) \bmod m$
$j \in \{0, 1, \ldots, m-1\}$

```
DoubleHashInsert(k)
    if (table is full) error
    probe = h(k) ; offset = d(k)
    while (table[probe] occupied)
        probe = (probe + offset) mod m

    table[probe] = k
```

Double Hashing Example

$m = 13$

$h(k) = k \mod 13$ $\qquad$ $d(k) = 7 - k \mod 7$

$h(k)$

Insert: 18, 41, 22, 44, 59, 32, 31, 73

		41			18				22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Analysis of Double Hashing

- Let α (load factor) be < 1
- Assume every probe looks at a random location in the table
- $1-\alpha$ fraction of table is empty.
- Expected # probes to find an empty location

$$= \frac{1}{1-\alpha}$$

required for unsuccessful search.

Analysis of Double Hashing

- Average number of probes for a succesful search
  = Average number of probes required to insert all elements

- To insert an element we need to find an empty location.

	# probes	Total # probes
$\{1, \dots m/2\}$	≤ 2	m
$\{\frac{m}{2}+1, \dots \frac{m}{2}+\frac{m}{4}\}$	≤ 4	m
$\{\frac{m}{2}+\frac{m}{4}+1, \dots \frac{m}{2}+\frac{m}{4}+\frac{m}{8}\}$	≤ 8	
$\vdots$		
next $m/2^i$	$\leq 2^i$	m

Analysis of Double Hashing

- # probes required to insert $\frac{m}{2} + \frac{m}{4} + \cdots + \frac{m}{2^i}$ elements

  = # probes required to leave $\frac{1}{2^i}$ fraction of table empty

  = $m\, i$

- # probes required to leave $(1-\alpha)$ fraction of table empty = $m \cdot \log\left(\frac{1}{1-\alpha}\right)$

- Average # probes required to insert n elements:

$$= \frac{m}{n} \log\left(\frac{1}{1-\alpha}\right) = \frac{1}{\alpha} \log \frac{1}{1-\alpha}$$

Chaining vs Probing

- Assume uniform hashing.

	Unsuccesful	successful
Chaining	$O(1+\alpha)$	$O(1+\alpha/2)$
Probing	$O\left(\frac{1}{1-\alpha}\right)$	$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$

on an average

- worst case: $O(n)$ time

- Exercise: Pick a hash fn. & come up with a worst-case instance!

Sets, Multisets, Multimap ADTs

unordered collection of elements without duplicates

- add(e)
- remove(e)
- contains(e)
- iterator()
- union(S, T)
- intersection(S, T)
- subtraction(S, T)

Set = map where keys don't have values associated

set with duplicates

- remove(e, n)
- count(e)
- size()

java.util.HashSet

- sorted sets, multisets & maps

map ADT where same key can be mapped to multiple values.

- get(k)
- put(k, v)
- remove(k, v)
- removeAll(k)
- size()
- entries()
- keys()
- values()

Recap

Abstract Data Types (ADTs)

- collection of objects that we would like to manipulate. (e.g. sets, lists, ...)
- characterised by operations you can perform on the objects.
- no constraint on how you implement these operations

1. Arrays
2. Linked Lists
   - Singly linked lists
   - Doubly linked lists

List ADT

3. Stacks (LIFO)
4. Queues (FIFO)
5. Trees
6. Priority Queues & Heaps
7. Hash tables

Arrays

in memory

- Implemented as a contiguous sequence of objects (of a certain type)

Array ADT: Given set S, $f(S)$ is the set of functions from a finite set of non-negative integers to S.

Operations: IsEmpty(), readIndex(i), insert(x, i), delete(i)

- Array has a predefined size which cannot be extended.

List ADT: add(i,e), set(i,e), get(i), remove(i)
- Can be implemented using Array ADT

growable array?

Comparing data structures for efficiency

→ compare algorithms without implementing them

Feature	Array	LinkedList
Space Usage	O (N) – N is the maximum possible size Even in growable arrays there is wasted space	O(n) – number of elements in the list
Given integer position, get the element	O(1) – allows random access	O(i) - for locating an element at integer position I
Inserting an element in the middle (given the position)	O(n) – copy all the subsequent elements	O(1)
Deleting an element in the middle (given the position)	O(n) – copy all subsequent elements	O(1)

- Asymptotic Analysis
- RAM model
- Program Correctness, Loop invariants.

Stacks & Queues

- Stores objects
- Insertions & deletions are LIFO
- Operations:
  - push(x)
  - pop()
  - top()
  - size()
- Can be implemented using an array or LL → more expensive in implementation
- O(1) for push & pop
- Max size of stack has to be defined apriori if using array

Applications:

- Browser history
- Function calls & return values
- paranthesis matching
- expression evaluation
- HTML parsing
- Convex hull (Graham's scan)

Queues → Insertions & deletions are FIFO

Operations:

- enqueue(x)
  add(x)
- dequeue()
  remove()
- peek()
  first()
  poll()
- Can be implemented using array/LL.

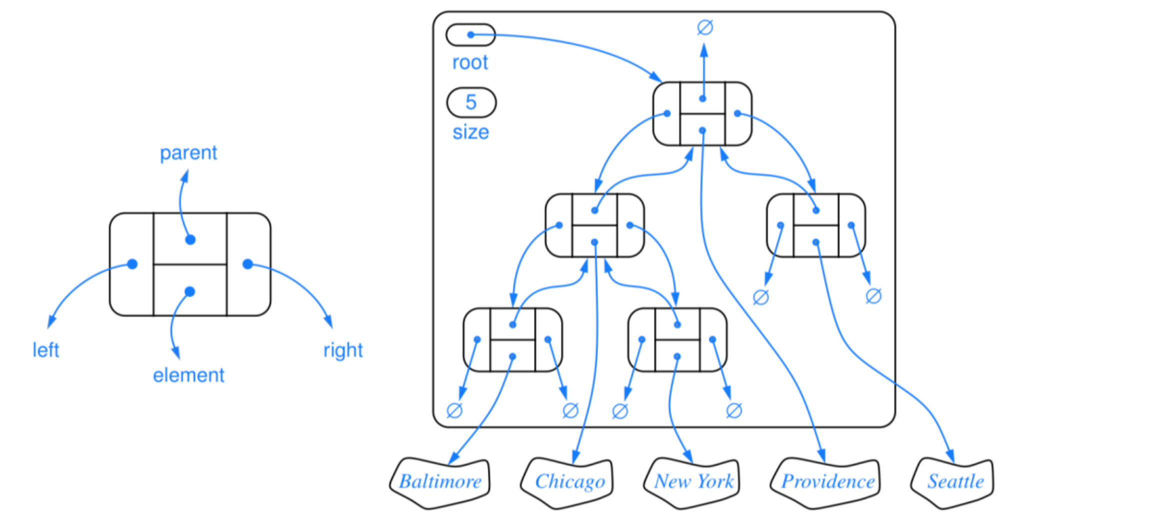
Trees (hierarchical data structures)

Defns: Nodes, (parent, children, sibling), root, leaf, depth, height,

internal / external — ancestors, descendants

- Recursive defn of a tree
- ordered tree

Implementing Trees



(a) (b)

Tree ADT:

root()
parent(p)
children(p)
numChildren(p)
size()
isEmpty()

IsInternal(p)
isExternal(p)
isRoot(p)
positions()
iterator()

Tree Traversals

→ Systematic way of visiting all nodes; assuming $O(1)$ work done at each node

Preorder(T): visit root(T) & recursively trees rooted at children (maintaining order if T is ordered)

PostOrder(T):
- Traverse subtrees rooted at children (maintaining order ...)
- visit root(T)

Overall running time = $O(n)$

preorder(p):
- visit(p)
- for each $c \in$ children(p)
  preorder(c)

post order(p)
- for each $c \in$ children(p)
  postorder(c)
- visit(p)

Breadth First Traversal

- visit all nodes at depth d before visiting nodes at depth d+1
- Queue nodes at each level.
- Can also be implemented using a stack.

BFS()

Q ← empty queue
Q.enqueue(root)
while Q not empty
    p = Q.dequeue();
    visit p
    for each c ∈ children(p)
        Q.enqueue(c)

Binary Trees

- ordered trees
- each node has ≤ 2 children (left, right)
- left ≤ right (paranthesis representation)
- "proper" binary tree - every node has 0 or 2 children
- Can be implemented with an array
- Inorder traversal → left subtree inorder → root → right inorder

ADT:

- left(p)
- right(p)
- sibling(p)

Priority Queues & Heaps

need to process elements according to priority

① arbitrary element insertion

② Removal of element with 1st priority

assigned via key (a number)

ADT: - insert(k, v)
- min()
- removeMin()
- size()
- isEmpty()

→ can be implemented using LL (sorted/unsorted) but O(n) insert/min time

- Heaps!
  - Binary Tree with relational property + structural property

- $O(\log n)$ time for all update operations

- Can be implemented using an array.

- Bonus: Heapsort!