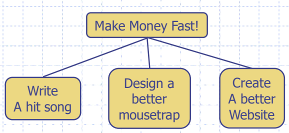
Lecture 3: Trees and Amortized Analysis

Sequential Unfoldment of Natural Law

Wholeness Statement

Trees are data structures that provide wide ranging capabilities and a highly flexible perspective on a set of element objects. *Science of Consciousness*: The whole range of space and time is open to individuals with fully developed awareness. Through the regular twice daily practice of the TM technique, alternated with dynamic activity, we develop more and more of our full potential as demonstrated by 100's of published scientific studies.

Trees

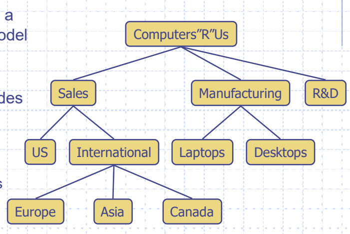


Outline and Reading

- Tree ADT (§2.3.1)
- Preorder and postorder traversals (§2.3.2)
- BinaryTree ADT (§2.3.3)
- Inorder traversal (§2.3.3)
- Euler Tour traversal (§2.3.3)
- Template method pattern
- Data structures for trees (§2.3.4)

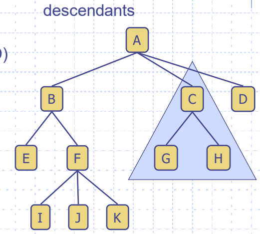
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Tree Terminology

- ***Root***: only node without parent (A)
- ***Internal node***: node with at least one child (A, B, C, F)
- ***External node*** (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- ***Ancestors of a node***: parent, grandparent, grand-grandparent, etc.
- ***Depth of a node***: number of ancestors
- ***Height of a tree***: maximum depth of any node (3 in tree to right)
- ***Descendant of a node***: child, grandchild, grand-grandchild, etc.
- ***Subtree***: tree consisting of a node and its descendants



Tree ADT

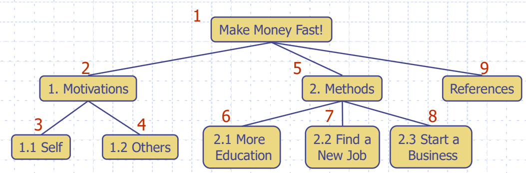
- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - positionIterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

7

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm *preOrder*(T, *v*)
  visit(*v*)
  for each *w* of T.children(*v*)
    preOrder (T, *w*)

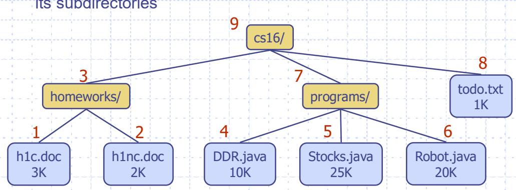


8

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

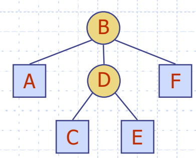
Algorithm *postOrder*(T, *v*)
  for each *w* of T.children(*v*)
    postOrder (T, *w*)
  visit(*v*)

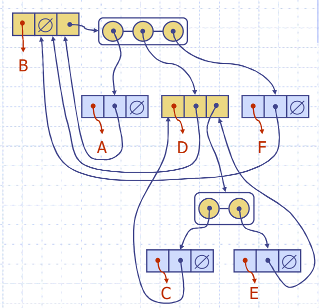


9

Data Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT





10

Performance of Tree ADT Linked Implementation

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - positionIterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)

11

Linked Implementation of the Tree ADT

Operation	Time
size, isEmpty	
positions, elements	
swapElements(p, q), replaceElement(p, o)	
root(), parent(p)	
children(v)	
isInternal(p), isExternal(p), isRoot(p)	

12

Linked Implementation of the Tree ADT

Operation	Time
size, isEmpty	1
positions, elements	n
swapElements(p, q), replaceElement(p, o)	1
root, parent(p)	1
children(v)	c_v
isInternal(p), isExternal(p), isRoot(p)	1

13

Main Point

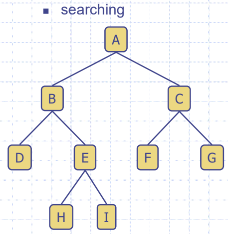
1. The Tree ADT models a hierarchical structure between objects simplified to a parent-child relation. Nodes store arbitrary objects/elements and connect to other nodes in the tree. A rooted tree has a root node without a parent; all other nodes have parents.

   Science of Consciousness: Pure consciousness is the root of the tree of life. Regular contact with pure consciousness waters that root and re-connects individual consciousness with pure consciousness by removing stress and strain resulting in positive benefic of everyone.

14

Binary Tree

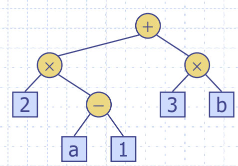
- A (proper) binary tree is a tree with the following properties:
  - Each internal node has two children
  - The children of a node are an ordered pair
- We assume that all binary trees are proper
- We call the children of an internal node left child and right child
- A binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



15

Arithmetic Expression Tree

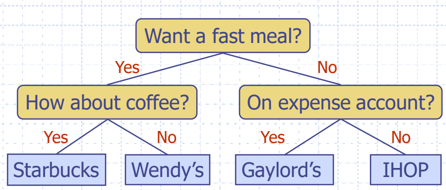
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



16

Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



17

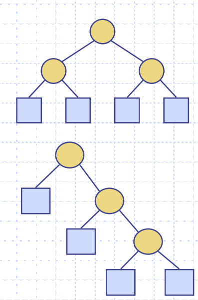
Binary Tree
Theorem 2.8 (page 84)

Properties of (Proper) Binary Trees

0. $e = i + 1$
1. $h \le i \le 2^h - 1$
2. $h+1 \le e \le 2^h$
3. $2h+1 \le n \le 2^{h+1} - 1$
4. $\log(n+1) - 1 \le h \le (n-1)/2$

where

- n number of nodes
- h height of the tree
- e number of external nodes
- i number of internal nodes



18

Proof of Theorem 2.8

Properties 2, 3, and 4 can be proven based on Property 1

Property 2:

	$h \le i \le 2^h - 1$	(Property 1)
<==>	$h \le e - 1 \le 2^h - 1$	(Property 0, e = i+1)
<==>	$h+1 \le e \le 2^h$	(arithmetic)■

Property 3:

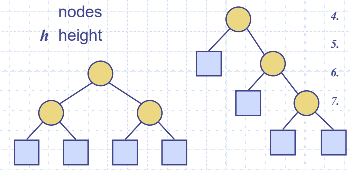
	$h+1 \le e \le 2^h \;\wedge\; h \le i \le 2^h - 1$	(Properties 1 and 2)
<==>	$2h+1 \le e+i \le 2^h + 2^h - 1$	(arithmetic)
<==>	$2h+1 \le n \le 2^{h+1} - 1$	(since $n = i+e$ and $2(2^h) = 2^{h+1}$) ■

Property 4:

	$n = 2e - 1$	(since $i = e - 1$ and $n = e+i$)
<==>	$(n + 1)/2 = e$	(algebra)
<==>	$(n + 1)/2 = e \;\wedge\; h+1 \le (n+1)/2$	(since $h+1 \le e$, from Property 1)
<==>	$(n + 1)/2 = e \;\wedge\; h \le (n - 1)/2$	(algebra)
==>	$(n + 1)/2 \le 2^h \;\wedge\; h \le (n - 1)/2$	($e \le 2^h$, from Property 1)
<==>	$\log_2 (n+1)/2 \le \log_2 2^h \;\wedge\; h \le (n - 1)/2$	(logarithms)
<==>	$\log_2 (n+1) - 1 \le h \;\wedge\; h \le (n - 1)/2$	(logarithms)■

Properties of Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



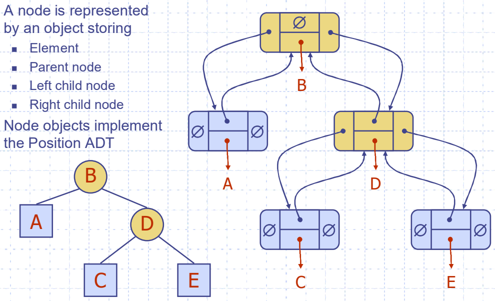
- Properties:
  1. $e = i + 1$
  2. $n = 2e - 1$
  3. $h \le i$
  4. $h \le (n - 1)/2$
  5. $e \le 2^h$
  6. $h \ge \log_2 (i + 1)$
  7. $h \ge \log_2 (n + 1) - 1$

BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position leftChild(p)
  - position rightChild(p)
  - position sibling(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

Data Structure for Binary Trees

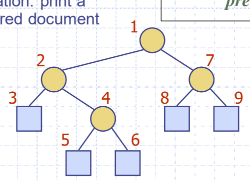
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

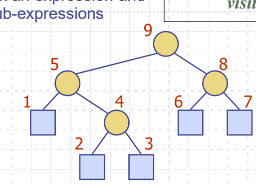
Algorithm *preOrder*(T, *v*)
  if T.*isExternal* (*v*) **then**
    visitExternal (*v*)
  else
    visitInternal(*v*)
    preOrder (T, T.*leftChild* (*v*))
    preOrder (T, T.*rightChild* (*v*))



Postorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: evaluate a binary expression or type check an expression and its sub-expressions

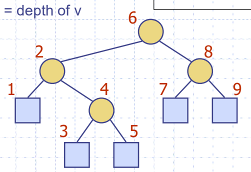
Algorithm *postOrder*(T, *v*)
  if T.*isExternal* (*v*) **then**
    visitExternal (*v*)
  else
    postOrder (T, T.*leftChild* (*v*))
    postOrder (T, T.*rightChild* (*v*))
    visitInternal(*v*)



Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder*(T, *v*)
  if T.*isExternal* (*v*) **then**
    visitExternal (*v*)
  else
    inOrder (T, T.*leftChild* (*v*))
    visitInternal(*v*)
    inOrder (T, T.*rightChild* (*v*))

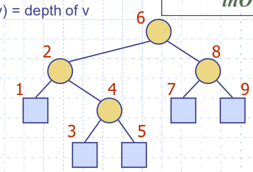


25

Inorder Traversal (variation)

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder*(T, *v*)
  if T.*isInternal* (*v*) **then**
    inOrder (T, T.*leftChild* (*v*))
  visit(*v*)
  if T.*isInternal* (*v*) **then**
    inOrder (T, T.*rightChild* (*v*))

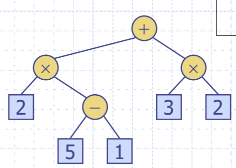


26

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(T, *v*)
  if T.*isExternal* (*v*) **then**
    return *v.element* ()
  else
    $x \leftarrow$ *evalExpr*(T, T.*leftChild*(*v*))
    $y \leftarrow$ *evalExpr*(T, T.*rightChild*(*v*))
    $\Diamond \leftarrow$ operator stored at *v*
    return $x \Diamond y$

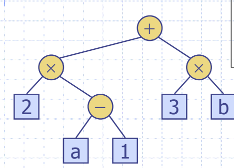


27

Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

Algorithm *printExpression*(T, *v*)
  if T.*isInternal* (*v*) **then**
    print("(")
    printExpression(T, T.*leftChild*(*v*))
  print(*v.element* ())
  if *isInternal* (*v*) **then**
    printExpression(T, T.*rightChild* (*v*))
    print (")")



$((2 \times (a - 1)) + (3 \times b))$

28

Exercise on List ADT

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
- Accessor methods:
  - position first()
  - position last()
  - position before(p)
  - position after(p)
- Query methods:
  - boolean isFirst(p)
  - boolean isLast(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
  - insertFirst(o)
  - insertLast(o)
  - insertBefore(p, o)
  - insertAfter(p , o)
  - remove(p)

Exercise:

- Write a method to calculate the sum of the integers in a list of integers
  - Design a recursive algorithm to calculate the sum.
  - **Hint**: you need a helper function.

Algorithm sum(L)

Algorithm sumHelper(L, p)

29

Exercise on Binary Trees

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - positionIterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
- Additional BinaryTree methods:
  - position leftChild(p)
  - position rightChild(p)
  - position sibling(p)

Exercise:

- Write a method to calculate the sum of the integers in a binary tree of integers
  - Assume that an integer is stored at each internal node and nothing in external nodes

Algorithm sum(T)

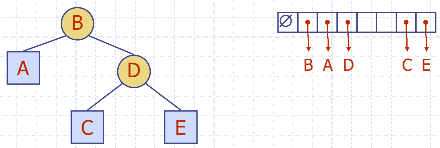
Hint: you also need a helper function with argument Position p

Algorithm sumHelper(T, p)

30

Data Structure for Binary Trees

- Another alternative: use an array to store the binary tree.
- Node objects are referenced by index:
  - Index 0 is empty and not used.
  - Root node is at index 1
  - Left child is at 2*index
  - Right child is at 2*index+1



31

Array-Based Implementation of Binary Tree

Operation	Time
size, isEmpty	
positions, elements	
swapElements(p, q), replaceElement(p, e)	
root, parent(p), children(p)	
isInternal(p), isExternal(p), isRoot(p)	

32

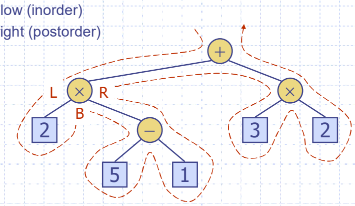
Array-Based Implementation of Binary Tree

Operation	Time
size, isEmpty	1
positions, elements	n
swapElements(p, q), replaceElement(p, e)	1
root, parent(p), children(p)	1
isInternal(p), isExternal(p), isRoot(p)	1

33

Euler Tour Traversal

- Generic traversal of a binary tree
- Includes as special cases the preorder, postorder, and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



34

Template Method Pattern

- Generic algorithm that can be specialized by redefining certain steps
- Implemented by means of an abstract Java class
- Visit methods that can be redefined/overridden by subclasses
- Template method eulerTour
  - Recursively called on the left and right children
  - A result array that keeps track of the output of the recursive calls to eulerTour
  - result[0] keeps track of the final output of the eulerTour method
  - result[1] keeps track of the output of the recursive call of eulerTour on the left child
  - result[2] keeps track of the output of the recursive call of eulerTour on the right child

35

Recall Our Earlier Example of the Template Method Pattern in Java

```
public abstract class EulerTour {
   protected void visitExternal(BinaryTree tree, Position p, Object[ ] r) { }
   protected void visitPreOrder(BinaryTree tree, Position p, Object[ ] r) { }
   protected void visitInOrder(BinaryTree tree, Position p, Object[ ] r) { }
   protected void visitPostOrder(BinaryTree tree, Position p, Object[ ] r) { }
   protected Object eulerTour(BinaryTree tree, Position p) {
      Object[ ] result = new Object[3];
      if tree.isExternal(p) { visitExternal(tree, p, result); }
      else {
            visitPreOrder(tree, p, result);
            result[1] = eulerTour(tree, tree.leftChild(p));
            visitInOrder(tree, p, result);
            result[2] = eulerTour(tree, tree.rightChild(p));
            visitPostOrder(tree, p, result);
      }
      return result[0];
   }
}
```


36

Specializations of EulerTour

```
public class Sum extends EulerTour {
// Sums the integers in a Binary Tree of Integers
    protected void visitExternal(BinaryTree t, Position p, Object[ ] res) {
        res[0] = new Integer(0);
    }
    protected void visitPostOrder(BinaryTree t, Position p, Object[ ] res) {
        res[0] = (Integer) res[1] + (Integer) res[2] + p.element()
    }
    public Integer sum(BinaryTree t) {
        return eulerTour(T, T.root());
    }
}
```

- We specialize class EulerTour to sum the Integers in a binary tree
- Assumptions
  - External nodes do not store objects
  - Internal nodes store Integer

37

Main Point

2. Each internal node of a Binary Tree has two children and each external node has no children. Thus the height, h, of a binary tree ranges as follows: $i \geq h \geq \log_2(i+1)$, that is, $O(\log_2 n) \leq h \leq O(n)$. *Science of Consciousness*: Pure consciousness spans the full range of life, from smaller than the smallest to larger than the largest.

38

Review and Relatives of Big-O

39

Review of Big Oh notation

Definition:

f(n) is O(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

- f(n) is O(g(n)) means that
  - f(n) is asymptotically less than g(n)
  - g(n) is an asymptotic upper bound on f(n)

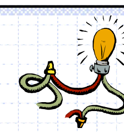
40

Relatives of Big-Oh

- **big-Omega**
  - f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$
- **big-Theta**
  - f(n) is Θ(g(n)) if there are constants $c1 > 0$ and $c2 > 0$ and an integer constant $n_0 \geq 1$ such that $c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$ for all $n \geq n_0$

41

Intuition for Asymptotic Notation



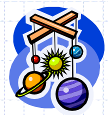
Big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega
- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta
- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

42

Example Uses of the Relatives of Big-Oh



- $7n^3 + 3n^2$ is $\Omega(n^3)$

  f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let $c = 7$ and $n_0 = 1$
- $5n^2$ is $\Omega(n)$

  f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$
- $5n^2 + 10n$ is $\Theta(n^2)$

  f(n) is Θ(g(n)) if, there are constants $c1, c2 > 0$, and there is an integer constant $n_0 > 0$ such that $c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)$ for all $n \geq n_0$

  Let c1=4, c2=6, then $4 \cdot n^2 \leq 5n^2 + 10n \leq 6 \cdot n^2$ for all $n \geq 6$

43

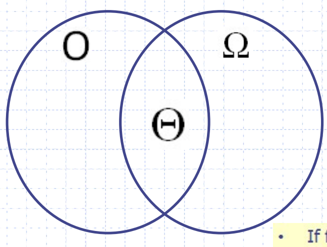
Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$f(n)$ is $\Omega(g(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth (Θ)	Yes	Yes

44

Relationships Between the Complexity Classes



- If f(n) is in both O(g(n)) and Ω(g(n)), it is in Θ(g(n)).

45

Asymptotic Notation in Practice

- The fastest algorithm in practice or for practical size input data sets is not always revealed!!!
- Because
  - Constants are dropped
  - Low-order terms are dropped
  - Algorithm efficiencies on small input sizes are not considered
- However, asymptotic notation is very effective
  - for comparing the scalability of different algorithms as input sizes become large

46

Amortization (§1.5)

Analysis of growable array-based stacks and queues

47

Amortization (§1.5)

- Comes from the field of accounting
  - Provides a monetary metaphor for algorithm analysis
- Useful for understanding the running time of algorithms that have steps with widely varying performance
  - i.e., each step performs a widely varying amount of work
  - Rather than focusing on individual operations, we study the interactions of a series of operations

48

Aggregate Amortized Analysis

- The average time required to perform an operation within a sequence of operations
  - The worst-case total running time of a series of operations divided by the number of operations
- Guarantees the average performance of each operation in the worst case

49

Aggregate Analysis

- Determine an upper bound, T(n),
  - the total cost of a sequence of n operations
- The average cost per operation is then T(n)/n
- The average cost becomes the amortized cost of each operation
- Thus all operations have the same amortized cost
  - Even though the cost of each individual operation varies widely

50

Growable Array-based Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant c
  - doubling strategy: double the size

```
Algorithm push(o)
  if t = S.length − 1 then
      A ← new array of
         size …
      for i ← 0 to t do
         A[i] ← S[i]
      S ← A
  t ← t + 1
  S[t] ← o
```


51

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We start with an empty stack represented by an array of size 1
- We call amortized time of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$

```
Algorithm push(o)
  if t = S.length − 1 then
      A ← new array of
         size …
      for i ← 0 to t do
         A[i] ← S[i]
      S ← A
  t ← t + 1
  S[t] ← o
```


52

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
  - since $n = kc$ in worst case (i.e., we increase array on nth push operation)
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus the amortized time of a push operation is $O(n)$ O(n/2c) because k=n/c

53

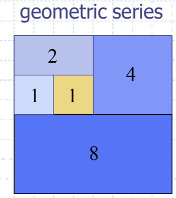
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
  - since $n = 2^k$ in worst case
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$n + 2 * 2^k - 1 = 3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$
- Because n= 2^k => k=log n

geometric series



54

Array Based Stack

- Advantage
  - Avoids the usual cost of copying array elements because there is no inserting or deleting of elements in the middle of the array
- Disadvantage
  - If many more pushes than pops, the array has to be resized often, which is costly

Quiz: Growable Array-based Queue

- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- What is the amortized running time of the enqueue operation for incremental and doubling strategies?
  - Hint: Similar to what we did for a growable array-based stack

Growable Array-based Queue

- The enqueue operation has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy

Other Amortization Techniques

The Accounting Method

- Uses a scheme of debits and credits to keep track of the running time of a series of operations
- Some operations are overcharged, others are undercharged
- The amount charged is called its *amortized cost*
- When amortized cost exceeds actual cost, the difference is assigned to specific objects within the data structure as credit
- Credits are used to pay for other operations that are charged less than they actually cost
- Amortized costs must be chosen carefully
- The total amortized cost of a sequence of operations must be an upper bound on the actual cost

Accounting Method Example:

push(o) – actual cost 1
pop() – actual cost 1
multipop(k) – actual cost *min*(k, n)

Accounting method:
push(o) – amortized cost 2
pop() – amortized cost 0
multipop(k) – amortized cost 0

When we do a push, we charge the actual cost (1 unit) and associate a credit of 1 unit with each element on the stack

When we do a pop or multipop, we charge 0 but use the credit associated with each element popped to pay for the operation

The Potential Method

- Determine the amortized cost of each operation
- Overcharge operations early to compensate for undercharges later
- Maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure

Main Point

3. The idea of "borrowing" and later "repaying" a data structure or program can be useful for determining the worst case time complexity of algorithms that have operations with widely varying running times. The basic idea of amortized analysis is that, even though a few operations are very costly, they do not occur often enough to dominate the entire algorithm; that is, the number of less costly operations far outnumber the costly ones over a large number of executions.
   Natural law (physics) says that for every action there is an equal and opposite reaction. To avoid mistakes, it is important to perform action from the silent, orderly level of our own consciousness.

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. The tree ADT is a generalization of the linked-list in which each tree node can have any number of children instead of just one. A proper binary tree is a special case of the generic tree ADT in which each node has either 0 or 2 children (a left and right child).
2. Any ADT will have a variety of implementations of its operations with varying efficiencies, e.g., the binary tree can be implemented as either a set of recursively defined nodes or as an array of elements.

3. **Transcendental Consciousness** is pure intelligence, the abstract substance out of which the universe is made.
4. **Impulses within Transcendental Consciousness**: Within this field, the laws of nature continuously organize and govern all activities and processes in creation.
5. **Wholeness moving within itself** : In Unity Consciousness, awareness is awake to its own value, the full value of the intelligence of nature. One's consciousness supports the knowledge that outer is the expression of inner, creation is the play and display of the Self.