# Introduction to Search

*Starting Chapter 4*

COMP 200, Lecture 15

*Rice University*

*Fall 2004*

## Search as a Paradigm

Algorithms texts often focus on two problems

- Search — finding some specified object in a data structure
- Sort — ordering the elements of some data structure

These two problems have an intuitive structure and play a critical role in many algorithmic problems

Chapter 4 of the Text

- Introduction to algorithmic paradigms
- First set of examples are searches and sorts

## Back to Binary Trees

In Lecture 13 (Friday) we talked about trees

- Focus on binary trees
- Used to represent many things
  - Ordered sets (dictionaries)
  - Syntax trees (sentence diagrams)
  - …
- Search in a binary tree is an important algorithm
  - Traversal — enumerating the nodes in some order
  - Genuine search — efficiently finding a specific node

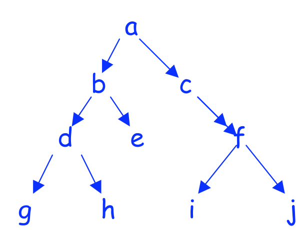
```
; a binary tree node (btn) is either
;   empty
; or a structure
;   (make-node data left right)
; where data is a name and left & right are btn
(define-struct btn (data left right)
```



## Traversing a binary tree

Handful of ways to traverse a tree

- Specific pattern of recursion dictates a "traversal order"

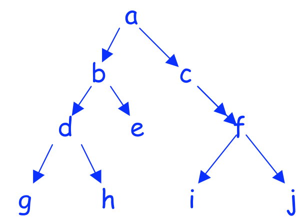



```
; Walk1 : btn -> boolean
; Purpose: traverse a btn
(define (Walk1 fee)
   (cond
      ((empty? fee)   true)
      ((node?  fee)
         (begin
            (Walk1 (node-left fee))
            (Walk1 (node-right fee))
            (do something with fee itself) ))
   ))
```

## Traversing a binary tree

Handful of ways to traverse a tree

- Specific pattern of recursion dictates a “traversal order”




```
; Walk1 : btn -> boolean
; Purpose: traverse a btn
(define (Walk1 fee)
   (cond
      ((empty? fee)   true)
      ((node?  fee)
         (begin
            (Walk1 (node-left fee))
            (Walk1 (node-right fee))
            (do something with fee itself) ))
   ))
```

Produces the order

(g h d e b i j f c a)

which we call “preorder”

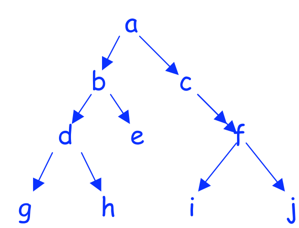
Left-to-right preorder



## Traversing a binary tree

Handful of ways to traverse a tree

- Specific pattern of recursion dictates a “traversal order”

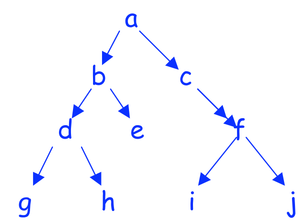



```
; Walk1 : btn -> boolean
; Purpose: traverse a btn
(define (Walk1 fee)
   (cond
      ((empty? fee)   true)
      ((node?  fee)
         (begin
            (do something with fee itself)
            (Walk1 (node-left fee))
            (Walk1 (node-right fee)) ))
   ))
```

## Traversing a binary tree

Handful of ways to traverse a tree

- Specific pattern of recursion dictates a "traversal order"




```
; Walk1 : btn -> boolean
; Purpose: traverse a btn
(define (Walk1 fee)
   (cond
      ((empty? fee)   true)
      ((node?  fee)
         (begin
            (Walk1 (node-left fee))
            (Walk1 (node-right fee))
            (do something with fee itself) ))
   ))
```

Produces the order

(a b d g h e c f i j)

which we call "postorder"

Left-to-right postorder



## Traversing a binary tree

Both of these orders are "depth-first" searches

- Dive deeper in the tree before going laterally across a level
  - Use a stack-like action to keep track of position in the traversal

```
push ( "root" node of tree )
while (stack is not empty)
   current ← pop()
   push (right child of current)
   push (left child of current)
   process current
```

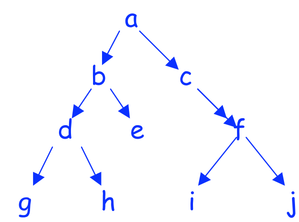
Stack version of postorder walk

What about other orders?

## Traversing a binary tree

Breadth-first search

- Depth-first dives to the bottom
- What if tree is huge? Not finite?
  - Search trees in a chess game
  - Combinatorial number of nodes
  - Consider alternatives to limited depth
- A breadth-first search exhausts a level before moving down
  - Breadth first order would be (a b c d e f g h i j)




Can we write a breadth-first search for our tree?

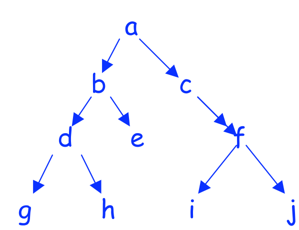
- Depth-first search models a stack; breadth-first uses a queue



## Traversing a binary tree

Breadth-first search

- Queue is first-in, first-out
- Same push() pop() abstraction




```
current ← root
while (current is not empty)
   push (left child of current)
   push (right child of current)
   process current
   current ← pop()
```

Breadth-first search

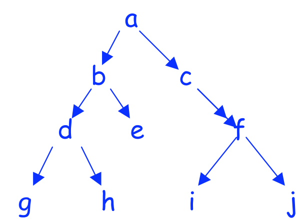
## Traversing a binary tree

Breadth-first search

- Queue is first-in, first-out
- Same push() pop() abstraction

```
current ← root
while (current is not empty)
   push (left child of current)
   push (right child of current)
   process current
   current ← pop()
```

Breadth-first search




Queue contains (over time)

[b, c] [d, e] [f] [g, h] [i, j]

forcing the “process” step into alphabetical order …



## Search in a binary tree

Important to consider the data structure & its properties

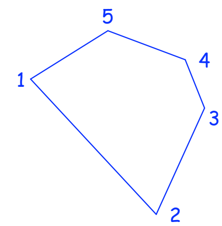
- Unordered tree
  - Use one of the traversals — depth-first, breadth-first
  - Amounts to exhaustive search (British Museum Algorithm)
  - In some cases, it’s the best we can do

- Ordered tree
  - Property defined on value of node & its children
  - Treesort produced such a tree
  - Can limit recursion to relevant subtree (child + its descendants)
  - Significantly reduces cost of search

## Other kinds of search

Finding maximal distance in a polygon (Chapter 4)

- Assume a simple convex polygon
  - All angles < 180 degrees
- BMA would compute distance between all pairs

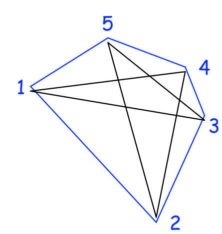





## Other kinds of search

Finding maximal distance in a polygon (Chapter 4)

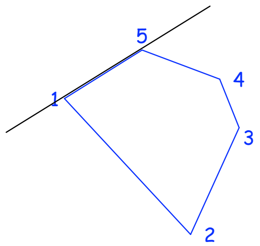
- Assume a simple convex polygon
  - All angles < 180 degrees
- BMA would compute distance between all pairs
  - Edges: (1,2),(2,3),(3,4),(4,5),(5,1)
  - Interior lines: (1,3),(1,4),(2,4),(2,5),(3,5)
  - Gets much more expensive as number of points rises




We find the answer while looking at fewer lines

## Other kinds of search

Finding maximal distance in a polygon (Chapter 4)

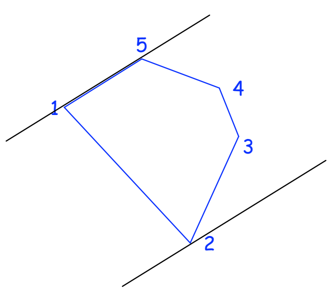



1. Draw a line along one edge



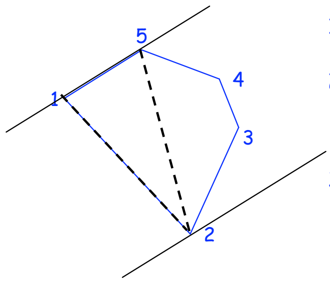
## Other kinds of search

Finding maximal distance in a polygon (Chapter 4)



## Other kinds of search

Finding maximal distance in a polygon (Chapter 4)

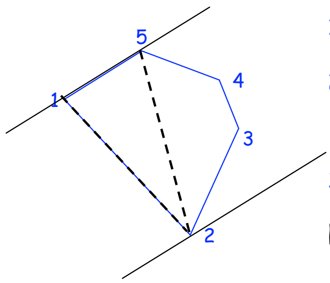



1. Draw a line along one edge
2. Find farthest extent along a perpendicular
   (Book's parallel line moving inward from infinity)
3. Measure 2 lines defined by the 3 points



## Other kinds of search

Finding maximal distance in a polygon (Chapter 4)



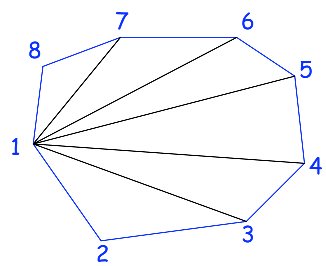

1. Draw a line along one edge
2. Find farthest extent along a perpendicular
   (Book's parallel line moving inward from infinity)
3. Measure 2 lines defined by the 3 points

Repeat 1 to 3 for each side, in sequence

For each edge in the polygon, it considers 2 distances

- Considers 2n distances, among (potentially) $n^2$ choices
- Pentagon is a low-complexity case
  - Has only two non-edge distances per node
  - Higher-degree polygons better show the worst case complexity

## Polygonal distance




Irregular octagon

Each node has five non-edge distances

⇒ 8 x 5 / 2 chords + 8 edges

⇒ 28 distances by BMA

⇒ "Better" search is 2 x 8 edges

⇒ 16 distances by "better algorithm

Bottom line:

⇒ Using contextual knowledge can reduce the cost of search (& other algorithms)