Recursion

Dr. Isaac Griffith
Idaho State University

Recursion

Idaho State University | Computer Science

- A *self referential* style of definition useful when it is difficult to directly define objects

- We can use recursion to define
  - Sequences
  - Functions
  - Algorithms
  - Data Structures

- A **recursive** or **inductive definition** requires two components
  - **Basis Step (or Base Case):** which defines an initial element or defines the simplest form of a problem that can be directly solved
  - **Recursive Step:** provides a rule by which the current element uses a previous one, or a means by which a larger problem is subdivided into the smaller problem

- The functional form of recursion is a form of the **Divide and Conquer** algorithm design strategy

Outline

The lecture if structured as follows:

- Recursively Defined Functions
- Algorithms
  - Search
  - Sorting
  - String Matching
  - Greedy
- Data Recursion

Recursively Defined Functions

CS 1187

ROAR

Recursively Defined Functions

- Recursively defined functions are **well-defined**
  - for every positive integer, the value of the function at this integer is determined in an unambiguous way.

- Suppose f is defined recursively by:

$$\begin{aligned} f(0) &= 3 \\ f(n+1) &= 2f(n)+3 \end{aligned}$$

- Find $f(1)$, $f(2)$, $f(3)$, $f(4)$:

$$\begin{aligned} f(1) &= 2f(0)+3 &&= 2\cdot 3+3 &&= 9 \\ f(2) &= 2f(1)+3 &&= 2\cdot 9+3 &&= 21 \\ f(3) &= 2f(2)+3 &&= 2\cdot 21+3 &&= 45 \\ f(4) &= 2f(3)+3 &&= 2\cdot 45+3 &&= 93 \end{aligned}$$

- Give a recursive definition of:

$$\sum_{k=0}^{n} a_k$$

Basis Case: $\sum_{k=0}^{0} a_k = a_0$

Recursive Case: $\sum_{k=0}^{n+1} a_k = \left(\sum_{k=0}^{n+1} a_k\right) + a_{n+1}$

Factorial

- We can define the function $n!$ as: $n! = 1 \times 2 \times \ldots \times n$

- However, this is far too imprecise for implementation
  - We can define $n!$ recursively

    Basis Step: $0! = 1$

    Recursive Step: $(n+1)! = (n+1) \times n!$

Haskell Implementation:

```
factorial :: Int -> Int
factorial 0 = 1
factorial (n + 1) = (n + 1) * factorial n
```


ROAR

Recursion Over Lists

- Recursion over lists
  - *Base Case:* `[]`, the empty list
  - *Recursive Case:* the non-empty list i.e., `(x:xs)`
- General Form:

```
f :: [a] -> type of result
f []     = result of empty list
f (x:xs) = result defined using (f xs) and x
```

- Example: `length`

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

```
length [1,2,3]
  = 1 + length [2,3]
  = 1 + (1 + length [3])
  = 1 + (1 + (1 + length []))
  = 1 + (1 + (1 + 0))
  = 3
```

- It is better to think of recursion as a systematic calculation using a high-level equational view rather than via a low-level machine view

Recursion Over Lists

- Another Simple Example: `sum`

```haskell
sum :: Num a => [a] -> a
sum []       = 0
sum (x:xs) = x + sum xs
```

```
sum [1,2,3]
  = 1 + sum [2,3]
  = 1 + (2 + sum [3])
  = 1 + (2 + (3 + sum []))
  = 1 + (2 + (3 + 0))
  = 6
```

- Returning a List: `(++)`

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ ys     = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
[1,2,3] ++ [9,8,7,6]
  = 1 : ([2,3] ++ [9,8,7,6])
  = 1 : (2 : ([3] ++ [9,8,7,6]))
  = 1 : (2 : (3 : ([] ++ [9,8,7,6])))
  = 1 : (2 : (3 : [9,8,7,6]))
  = 1 : (2 : [3,9,8,7,6])
  = 1 : [2,3,9,8,7,6]
  = [1,2,3,9,8,7,6]
```

Recursion Over Lists

- Recursing over 2 lists: `zip`

```haskell
zip :: [a] -> [b] -> [(a, b)]
zip [] ys          = []
zip xs []          = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

```haskell
zip [1,2,3,4] ['A','*','q']
  = (1, 'A') : zip [2,3,4] ['*', 'q']
  = (1, 'A') : ((2, '*') : zip [3,4] ['q'])
  = (1, 'A') : ((2, '*') : ((3, 'q') : zip [4] []))
  = (1, 'A') : ((2, '*') : ((3, 'q') : []))
  = (1, 'A') : ((2, '*') : [(3, 'q')])
  = (1, 'A') : [(2, '*'), (3, 'q')]
  = [(1, 'A'), (2, '*'), (3, 'q')]
```

- Recursing a list of lists: `concat`

```haskell
concat :: [[a]] -> [a]
concat []         = []
concat (xs:xss) = xs ++ concat xss
```

```haskell
concat [[1], [2,3], [4,5,6]]
  = [1] ++ concat [[2,3], [4,5,6]]
  = [1] ++ ([2,3] ++ concat [[4,5,6]])
  = [1] ++ ([2,3] ++ [4,5,6])
  = [1] ++ [2,3,4,5,6]
  = [1,2,3,4,5,6]
```

Higher-Order Recursive Functions

- Each of the prior recursive functions are quite similar

- It would be elegant if we had a function which express this general computational pattern

- Such a general function would need to be provided both
  - the functions inputs
  - the computation (a function) to perform

- Such functions are called **higher order functions** or a **combinator**

- We have several choices of **combinators**
  - `map` - takes a function and applies it to all items in a list $\Rightarrow$ List
  - `zipWith` - takes a function and applies it to all items in two lists $\Rightarrow$ List
  - `foldr` and `foldl` - takes a function, aggregation variable, and applies to the function to combine the list values into the var $\Rightarrow$ singleton variable

Algorithms

CS 1187

ROAR

Algorithms

- There are many general classes of problems that arise in Discrete Mathematics and Computing
- The key to solving such problems is to
  1. Construct a model that translates the problem into a mathematical context
  2. Define a method that will solve the general problem using the model
- The second step is the purview of *algorithm* design
- **Algorithm:** a finite sequence of precise instructions for performing a computation or for solving a problem
  - Typically expressed in English or Pseudocode
- **Pseudocode:** an intermediate step between an English language description of an algorithm and an implementation of the algorithm in a programming language

Pseudocode Example

- Finding the maximum element in a finite sequence

procedure MAX(A)
  $max := A_1$
  for $i := 2$ **to** n **do**
    if $max < A_i$ **then** $max := A_i$
  return max

- To gain insight into how an algorithm works it is useful to construct a **trace** that shows the steps for a given specific input.

Algorithm Properties

- Algorithms generally share several properties:
  - **Input:** An algorithm has input values from a specified set
  - **Output:** From each set of input values an algorithm produces output values from a specific set.
    - The *output* values are the solution to the problem
  - **Definiteness:** The steps of an algorithm must be defined precisely
  - **Correctness:** An algorithm should produce the correct output values for each set of input values
  - **Finiteness:** An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set
  - **Effectiveness:** It must be possible to perform each step of an algorithm exactly and in a finite amount of time
  - **Generality:** The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

Search Algorithms

CS 1187

ROAR

Search Algorithms

- **Search Problem Definition:** Locate an element x in a list of distinct elements, $a_1, a_2, \ldots, a_n$, or determine that it is not in the list

- The solution to this problem is the location of the term int he list that equals x and 0 if x is not in the list.

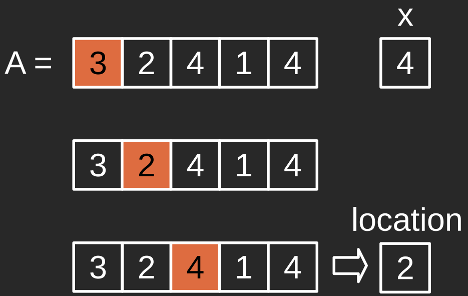
- This is one of the most commonly occurring problems in computer science, and occurs in many different contexts

Linear Search

- **Linear Search (sequential search):** searches an ordered list $(a_1, a_2, \ldots, a_n)$ for some value x, starting at a_1 and ending at a_n terminating when either the value if found (i.e., $x = a_i$) or the end of the list is reached.

Iterative Linear Search Alg:

procedure LINEARSEARCH(A, x)
  $i := 1$
  while $i \leq n$ **and** $x \neq A_i$ **do**
    $i := i + 1$
  if $i \leq n$ **then** $location := i$
  else $location := 0$
  return $location$



ROAR

Linear Search

Recursive Linear Search Alg:

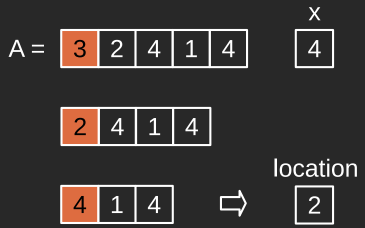
- $A \rightarrow$ array/list to search
- $i \rightarrow$ current index
- $j \rightarrow$ size of list
- $x \rightarrow$ value to find

procedure LINSEARCH(A, i, j, x)
  if $A_i = x$ **then**
    return i
  else if $i = j$ **then**
    return 0
  else
    return LINSEARCH$(A, i+1, j, x)$

- Requires: $O(n)$ comparisons

Haskell Implementation:

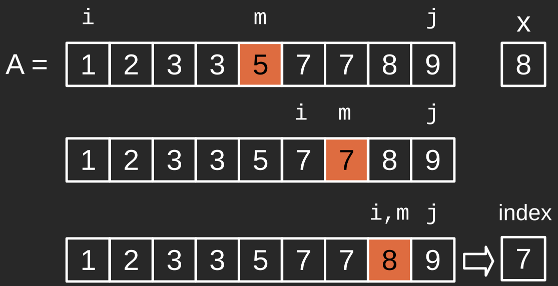
```haskell
linSearch :: Eq a => [a] -> Int -> a -> Int
linSearch [] _ _      = 0
linSearch (y:ys) i x =
    if x == y then i
    else linSearch ys (i + 1) x
```



Binary Search

- Can be used when the list is ordered in either ascending or descending order
- Successively searches smaller and smaller sections, until either the item is found or not
- Requires $O(\log n)$ comparisons

procedure BINSEARCH(A, x)
    $i := 1$
    $j := n$
    while $i < j$ **do**
        $m := \lfloor (i + j)/2 \rfloor$
        if $x > A_m$ **then** $i := m + 1$
        else $j := m$
    if $x = A_j$ **then** $location := i$
    else $location := 0$
    return $location$



ROAR

Binary Search

Recursive Binary Search Alg:

- $A \rightarrow$ array/list to search
- $i \rightarrow$ current index
- $j \rightarrow$ size of list
- $x \rightarrow$ value to find

procedure BINSEARCH(A, i, j, x)
  $m := \lfloor (i+j)/2 \rfloor$
  if $x = A_m$ **then return** m
  else if $x < A_m$ **and** $i < m$ **then**
    return BINSEARCH($A, i, m-1, x$)
  else if $x > A_m$ **and** $j > m$ **then**
    return BINSEARCH($A, m+1, j, x$)
  elsereturn 0

- Requires $O(\log n)$ comparisons

Haskell Implementation

```
binSearch :: (Ord a) => [a] -> a -> Int -> In
binSearch arr x lo hi
  | hi < lo = -1
  | pivot > x = binSearch arr x lo (mid - 1)
  | pivot < x = binSearch arr x (mid + 1) hi
  | otherwise =  mid
 where
  mid = lo + (hi - lo) `div` 2
  pivot = arr!!mid
```

Sorting Algorithms

CS 1187

Sorting

- **Sorting:** the problem of ordering a collection of element (i.e., a list or set)
  - This problem occurs in many contexts, including:
    - Telephone directory
    - Addresses in mailing lists
    - Directory of songs for download
    - Dictionaries
- A significant amount of computing resources is devoted to sorting $\Rightarrow$ a large amount of effort has gone into developing efficient sorting algs
  - 100+ existing sorting algorithms
  - Recently Timsort and Library Sort were developed

Insertion Sort

procedure SORT(A)
    for $j := 2$ **to** n **do**
        $i := 1$
        while $A_j > A_i$ **do**
            $i := i + 1$
        $m := A_j$
        for $k := 0$ **to** $j - i - 1$ **do**
            $A_{j-k} := A_{j-k-1}$
        $A_i := m$

Haskell Implementation:

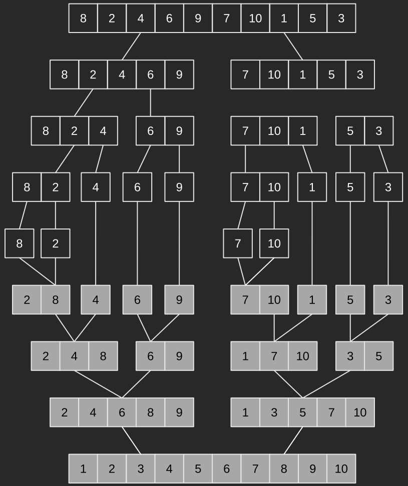
```haskell
insert :: (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
  | x < y     = x:y:ys
  | otherwise = y : (insert x ys)

sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = insert x (sort xs)
```

ROAR

Merge Sort

- Idea is to recursively split the list in half until each piece is size 1 or less
- Each sublist is then merged to form a sorted combined list
- **Lemma:** Two sorted lists with m and n elements can be merged into a sorted list in no more than $m + n - 1$ comparisons.
- **Theorem:** The number of comparisons needed to merge sort a list with n elements is $O(n \log n)$



ROAR

Merge Sort

Idaho State University | Computer Science

The Algorithm

procedure MSORT(L)
  if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$
    $L1 \leftarrow L_1, L_2, \ldots, L_m$
    $L2 := L_{m+1}, L_{m+2}, \ldots, L_n$
    $L :=$ MERGE(MSORT($L1$), MSORT($L2$))

procedure MERGE($L1, L2$)
  $L := [\,]$
  while $L1$ and $L2$ are both nonempty **do**
    remove smaller of $L1_1$ and $L2_1$, add to L
  if one list is empty **then**
    remove all elements of the other list and append to L
  return L

Haskell Implementation

```haskell
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] [] = []
merge [] ys = ys
merge xs [] = xs
merge allX@(x:xs) allY@(y:ys)
  | x > y     = y : merge allX ys
  | otherwise = x : merge xs allY

sort :: (Ord a) => [a] -> [a]
sort [] = []
sort [a] = [a]
sort [a,b]
  | a > b     = [b, a]
  | otherwise = [a, b]
sort list =
  let split = splitAt(length list `div` 2) list
      firstHalf  = sort (fst split)
      secondHalf = sort (snd split)
  in merge firstHalf secondHalf
```

QuickSort

- A sorting approach based on the idea of *divide and conquer* where we take a list and we attempt to successively cut it in half to make the problem size smaller

- The goal is to gain more than by reducing by one while also ensuring the recursion will complete

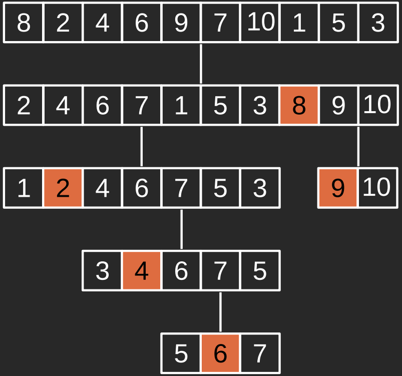
- The algorithm in a nutshell works as follows:
  - **Base Case:** empty list $\rightarrow$ empty list
  - **Recursive Case:** non-empty list
    - Select a *pivot* (typically the first or last item in the list)
    - We then select all items from the list < pivot and quick sort those and add them before the pivot
    - We select all items from the list $\geq$ pivot and quick sort them and place them after the pivot

QuickSort

The Algorithm:

procedure SORT(L, lo, hi)
  if $lo \geq hi$ **or** $lo < 0$ **then**
    return
  $p :=$ PARTITION(L, lo, hi)
  SORT$(L, lo, p-1)$
  SORT$(L, p+1, hi)$

procedure PARTITION(L, lo, hi)
  $pivot := L_{lo}$
  $i := lo$
  for $j := lo$ **to** $hi - 1$ **do**
    if $L_j \leq pivot$ **then**
      $i := i + 1$
      swap L_i with L_j
  swap L_i with L_{lo}
  return i



Haskell Implementation:

```haskell
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort (pivot:xs) =
  quickSort [y | y <- xs, y < pivot]
  ++ [pivot]
  ++ quicksort [y | y <- xs, y >= pivot]
```

String Matching

CS 1187

ROAR

String Matching

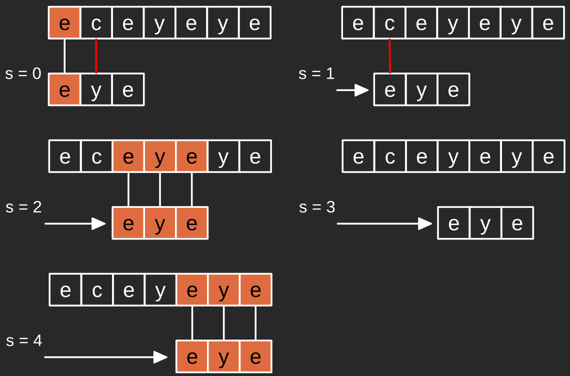
String Matching: finding where a particular string of characters *P*, called a *pattern*, occurs, if it does, within another string *T*, called the *text*

- this is another commonly occurring problem with a wide array of applications, including:
  - Text editing
  - Spam filters
  - Detecting network attacks
  - Search engines
  - Plagiarism detection
  - Bioinformatics
  - and many others

String Matching

Naive String Matcher

procedure MATCH(n, m, T, P)
    for $s := 0$ **to** $n - m$ **do**
        $j := 1$
        while $j \leq m$ **and** $T_{s+j} = P_j$ **do**
            $j := j + 1$
        if $j > m$ **then print** "s is a valid shift"



Haskell Implementation

```haskell
match :: [Char] -> [Char] -> Int -> [Int]
match [] _ _ = []
match _ [] _ = []
match p allT@(t:ts) i =
  let l = length p
      n = take l allT
  in if (n == p) then i : (match p ts (i + 1))
     else match p ts (i + 1)
```

Greedy Algorithms

CS 1187

ROAR

Greedy Algorithms

Idaho State University | Computer Science

- **Optimization Problems:** Problems where the goal is to find a solution to the given problem that either minimizes or maximizes the value of some parameter. Examples include:
  - Finding a route between two cities with the least total mileage
  - Encoding a message using the fewest bits possible
- **Greedy Algorithms:** Algorithm design strategy, wherein we select the “best” choice at each step rather than attempt to consider all sequences of steps that may lead to the optimal solution
  - Once we know a greed alg finds a feasible solution, then we must prove it is an optimal one
- Greedy algs are often the approach used to solve optimization problems

Greedy Algorithms

- Make *n* cents change with Quarters, dimes, nickels, and pennies using the least total number of coins.

procedure CHANGE($Coins, n$)
    for $i := 1$ **to** r **do**
        $D_i := 0$
        while $n \leq Coins_i$ **do**
            $D_i := D_i + 1$
            $n := n - Coins_i$
    return D

- The proof of optimality can be found in DMA on page 211

Coins = [0.25, 0.10, 0.05, 0.01]

					Value
D =	0	0	0	0	.86
i = 1	1	0	0	0	.61
i = 1	2	0	0	0	.36
i = 1	3	0	0	0	.11
i = 2	3	1	0	0	.01
i = 3	3	1	0	0	.01
i = 4	3	1	0	1	.00
	3	1	0	1	.00

Data Recursion

CS 1187

ROAR

Peano Arithmetic

- This date structure serves as an example of a recursive ADT

```haskell
data Peano = Zero | Succ Peano deriving Show
```

- **Example:**

```
1 = Succ Zero
2 = Succ (Succ (Succ Zero))
```

- Some operations:

```haskell
decrement :: Peano -> Peano
decrement zero = Zero
decrement (Succ a) = a

add :: Peano -> Peano -> Peano
add Zero b     = b
add (Succ a) b = Succ (add a b)
```

```haskell
sub :: Peano -> Peano -> Peano
sub a Zero              = a
sub Zero b              = Zero
sub (Succ a) (Succ b) = sub a b

lt :: Peano -> Peano -> Bool
lt a        Zero     = False
lt Zero     (Succ b) = True
lt (Succ a) (Succ b) = lt a b
```


ROAR

Data Recursion

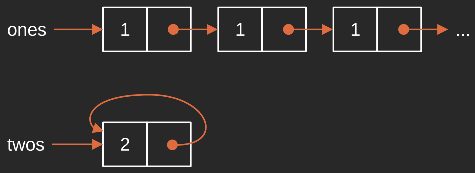
- Recursive functions are useful in nearly all programming languages
  - They are especially important for data structures such as Trees and Graphs.

- **Data Recursion:** An important technique that uses recursion to define data structures
  - The idea is to define *circular* data structures
  - **Example**: An infinite list of 1's

```
f :: a -> [a]
f x = x : fx
ones = f 1
```

  - Rather than a function, we could simply use a circular list

```
twos = 2 : twos
```



ROAR

Data Recursion

- Data recursion is possible in languages like Haskell due to the use of *lazy evaluation*

- **Lazy Evaluation:** is a technique where expressions are not evaluated until their value is actually needed

- However, most imperative languages (such as C or Java) do not support this and thus we cannot construct infinite data structures in this manner
  - Rather, they would cause an infinite loop

- Yet, we can create circular data structures in other ways

Recursively Defined Strings

- Recursion can play a role when working with strings

- We can define a string over an alphabet $\sum$ as a finite sequence of symbols from $\sum$
  - We can then define $\sum^*$ as the set of strings over $\sum$
  - The recursive definition is:

    Basis Step: $\lambda \in \sum^*$ (where λ is the empty string)

    Recursive Step: if $w \in \sum^*$ and $x \in \sum^*$, then $wx \in \sum^*$

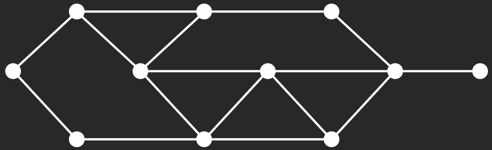
- **Example:** $\sum = \{0, 1\}, \sum^* = \{\lambda,\ 0,\ 1,\ 00,\ 01,\ 10,\ 11,\ \ldots\}$
- Basic string operations can also be defined recursively, for example
  - Concatenation
  - Length

ROAR

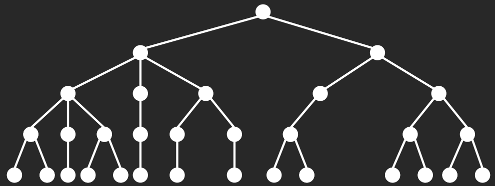
Recursively Defined Trees

- **Graph:** A data structure composed of vertices and edges connecting pairs of vertices
  - Graphs can be constructed by defining each node with an equation in a `let` expression
  - Thus, each node can be referred to by any other node (including itself)

```
object = let a = 1 : b
             b = 2 : c
             c = [3] ++ a
         in a
```



- **Tree:** A special type of graph

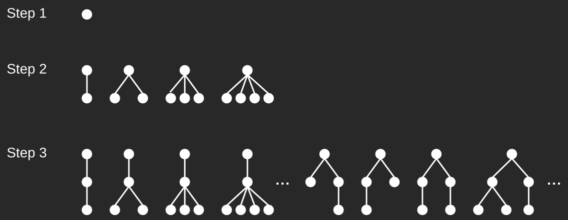


Rooted Trees

- **Rooted Tree:** a tree consisting of vertices containing a distinguished vertex called the *root* and edges connecting these vertices.
  - We can define such a structure recursively

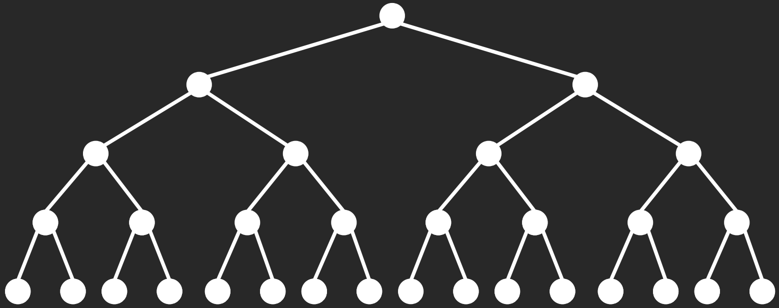
Basis Step: A single vertex r is a rooted tree

Recursive Step: Suppose that $T_1, T_2, \ldots, T_n$ are disjoint rooted trees with roots $r_1, r_2, \ldots, r_n$. Then the graph formed by starting with a root r not in any T_i and adding an edge from r to each of the vertices $r_1, \ldots, r_n$, is also a rooted tree.



Binary Trees

- **Binary Tree:** A rooted tree in which a vertex may have only two children, each of which is a subtree
  - **Full Binary Tree:** if a vertex has children, it must have both a left and right child
  - **Extended Binary Tree:** either the left or right subtree may be empty

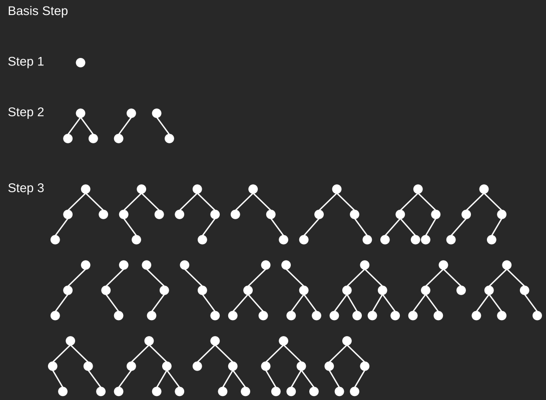


Extended Binary Trees

- The set of *extended binary trees* is defined as:

  Basis Step: the empty set is an extended binary tree

  Recursive Step: If T_1 and T_2 are disjoint extended binary trees, there is an extended binary tree denoted $T_1 \cdot T_2$, consisting of a root *r* together with edges connecting *r* to the roots of T_1 (left) and T_2 (right) when T_1 and T_2 are nonempty.

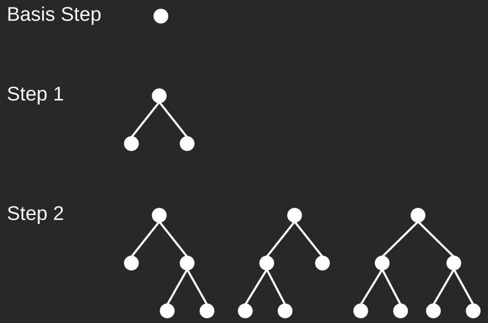


Full Binary Trees

- Recursively defined as:

  Basis Step: There is a full binary tree consisting of only a single vertex *r*

  Recursive Step: If T_1 and T_2 are disjoint fully binary trees, there is a full binary tree denoted $T_1 \cdot T_2$, consisting of a root *r* together with edges connecting *r* to the roots of T_1 and T_2



Inductively Defined Sets

- One approach for defining sets is to simply enumerate all of its elements.
  - Unfortunately, this is impractical for all but the smallest sets
  - For larger sets, we could simply use an ellipsis ". . ." to indicate the definition continues.
    - However, this is an informal approach which is both imprecise and ambiguous
- What we need is an approach that can define these types of sets which is concise, precise, and unambiguous

The Idea Behind Induction

- Induction is sort of a form of mathematical programming which produces a proof when needed
  - i.e., we can assert that something is a member of a set defined by induction

- **Example:**

$$0 \in S$$
$$n \in S \quad \rightarrow \quad n + 1 \in S$$

- By *modus ponens* and the first assertion we can deduce $1 \in S$, by similar reasoning we can also deduce $2 \in S$

- Furthermore, we can continually build up this chain for **any** natural number

ROAR

The Idea Behind Induction

- Such inductive definitions can show that a set contains a value *v*, but requires us to enumerate the values prior to *v*

- **Sequence:** a set with an ordering
  - The inductively enumerated values form a *sequence*

- Computationally we can use this idea to generate sets

```
imp1 :: Integer -> Integer
imp1 1 = 2
imp1 x = error "premise does not match"

imp2 :: integer -> Integer
imp2 2 = 3
imp x = error "premise does not match"
```

```
s :: [Integer]
s = [1, imp1 (s!!0), imp2 (s!!1)]
```

The Induction Rule

- Recall,

$$\begin{aligned} 0 \in S & & \{\text{base case}\} \\ n \in S & \rightarrow & \{\text{induction case}\} \end{aligned}$$

- The *induction case* generate the links of the chain which define the set, starting from the base case
  - By simply modifying our induction rule, we can create completely different sets

$$n \in S \rightarrow n + 2 \in S$$

  This rule generates the set of even natural number, however if we change the base case to be $1 \in S$, this same induction case then generates the set of odd natural numbers.

The Induction Rule

- Our prior implementation was fairly restricted

- If we want to implement the following set:

$$0 \in S$$
$$x \in S \quad \rightarrow \quad x + 1 \in S$$

- We can do the following

```haskell
increment :: Integer -> Integer
increment x = x + 1

s :: [Integer]
s = 0 : map increment s
```

- This style of programming is called **data recursion**
- *map* will proceed down *s*, creating each value it needs, then using it.

Defining Sets Inductively

CS 1187

Defining Sets Using Induction

- Beyond the base and inductive cases, inductive set definition needs one more component: the *extremal clause*

- **Extremal Clause:** A statement which excludes anything from the set that are not introduced by the base case, or are instantiations of the induction case, it reads something like the following:

  "Nothing is an element of the set unless it can be constructed by a finite number of uses of the first two clauses"

- Thus all inductive set definitions include 3 parts:
  - **Base Case:** a simple statement of some mathematical fact: i.e., $1 \in S$
  - **Induction Case:** an implication in a general form: $\forall\, x \in U,\ x \in S \rightarrow x + 1 \in S$
  - **Extremal Clause:** Nothing is in the set being defined unless it got there by a finite number of uses of the first two cases

The Natural Numbers

- The set of natural numbers, $\mathbb{N}$, is defined as follows
  - **Base Case:** $0 \in \mathbb{N}$
  - **Induction case:** $x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}$
  - **Extremal clause:** nothing is an element of the set $\mathbb{N}$ unless it can be constructed with a finite number of uses of the base and induction cases.

- We can show that an arbitrary number above and including 0 are in $\mathbb{N}$

1.	$0 \in \mathbb{N}$	Base Case
2.	$0 \in \mathbb{N} \rightarrow 1 \in \mathbb{N}$	*instantiationrule*, *inductioncase*
3.	$1 \in \mathbb{N}$	1, 2, Modus Ponens
4.	$1 \in \mathbb{N} \rightarrow 2 \in \mathbb{N}$	instantiation rule, induction case
5.	$2 \in \mathbb{N}$	3, 4, Modus Ponens

Binary Machine Words

- Let *BinDigit* be the set $\{0, 1\}$. The set *BinWords* of machine words in binary is defined as follows:
  - **Base Case:** $x \in \texttt{BinDigit} \rightarrow x \in \texttt{BinWords}$
  - **Induction Case:** if x is a binary digit and y is a binary word, then their concatenation xy is also a binary word

$$(x \in \texttt{BinDigit} \wedge y \in \texttt{BinWords}) \rightarrow xy \in \texttt{BinWords}$$

  - **Extremal Clause:** Nothing is an element of $\texttt{BinWords}$ unless it can be constructed with a finite number of uses of the base and induction cases

- A set based on another set S in this way is given the name S^+
  - it is the set of all possible non-empty strings over S
  - S^* is similar to S^+ except S^* includes the empty string
  - $\texttt{BinWords}$ could have also been written as $\texttt{BinDigit}^+$

ROAR

Haskell Implementation

- We can define a function to create two new `BinWords` based on one that has been provided
  - i.e., given `[1, 0]` it will return `[0, 1, 0]` and `[1, 1, 0]`

```haskell
newBinaryWords :: [Integer] -> [[Integer]]
newBinaryWords ys = [0:ys, 1:ys]
```

- We then define the set of `BinWords` as:

```haskell
mappend :: (a -> [b]) -> [a] -> [b]
mappend f []       = []
mappend f (x:xs) = f x ++ mappend f xs

binWords = [0] : [1] : (mappend newBinaryWords binWords)
```

The Set of Integers

- Both of the prior sets are **well-founded**, meaning they are infinite in only one direction, and they have a *least* element
- **Countable Set:** a set which can be counted using the natural numbers
  - Are the integers countable?
    - Doesn't have a least element
    - Infinite in two directions
  - However we can count hem using natural numbers as follows:
    - Start at 0
    - For every number $n \in \mathbb{N}$, we count both n and $-n$ in $\mathbb{Z}$
  - That is, we can consider the set of integers as an infinitely long tape folded in half at 0, and then count the overlapping numbers $(i, -i)$ for each $i \in \mathbb{N}$
- Yet, this does not specify $\mathbb{Z}$

The Set of Integers

- The set $\mathbb{Z}$ of integer is defined as follows:
  - **Base Case:** $0 \in \mathbb{Z}$
  - **Induction Case:** $(x \in \mathbb{Z} \land x \geq 0) \rightarrow x + 1 \in \mathbb{Z} \land -(x+1) \in \mathbb{Z}$
  - **Extremal Clause:** nothing is in $\mathbb{Z}$ unless its presence is justified by a finite number of uses of the base and induction cases

Thus, we can define integers using Haskell, as follows

```haskell
build:: a -> (a -> a) -> Set a
build a f = set
    where set = a : map f set

builds :: a -> (a -> [a]) -> Set a
builds a f = set
    where set = a : mappend f set

nextInteger :: Integer -> [Integer]
nextInteger x
  = if x > 0 \/ x == 0
      then [x + 1, -(x + 1)]
      else []

integer :: [Integer]
integers = builds 0 next Integers
```

For Next Time

- Review DMUC Chapter 3 and 9
- Review DMA Chapters 3.1 and 5.3 - 5.5
- Review this Lecture
- Read DMUC Chapter 4
- Read DMA Chapters 5.1 - 5.2

Are there any questions?