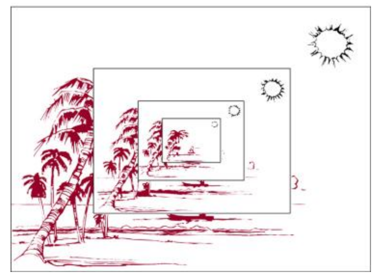
CSE2023 Discrete Computational Structures

Lecture 16

5.3 Recursive definitions and structural induction



A recursively defined picture

Recursive definitions

- The sequence of powers of 2 is given by $a_n=2^n$ for n=0, 1, 2, …
- Can also be defined by $a_0=1$, and a rule for finding a term of the sequence from the previous one, i.e., $a_{n+1}=2a_n$
- Can use induction to prove results about the sequence
- **Structural induction**: We define a set recursively by specifying some initial elements in a basis step and provide a rule for constructing new elements from those already in the recursive step

Recursively defined functions

- Use two steps to define a function with the set of non-negative integers as its domain
- **Basis step**: specify the value for the function at zero
- **Recursive step**: give a rule for finding its value at an integer from its values at smaller integers
- Such a definition is called a recursive or inductive definition

Example

- Suppose f is defined recursively by
  - $f(0)=3$
  - $f(n+1)=2f(n)+3$

  Find f(1), f(2), f(3), and f(4)
  - $f(1)=2f(0)+3=2*3+3=9$
  - $f(2)=2f(1)+3=2*9+3=21$
  - $f(3)=2f(2)+3=2*21+3=45$
  - $f(4)=2f(3)+3=2*45+3=93$

Example

- Give an inductive definition of the factorial function $f(n)=n!$
- Note that $(n+1)!=(n+1)\cdot n!$
- We can define $f(0)=1$ and $f(n+1)=(n+1)f(n)$
- To determine a value, e.g., $f(5)=5!$, we can use the recursive function

$$f(5)=5\cdot f(4)=5\cdot 4\cdot f(3)=5\cdot 4\cdot 3\cdot f(2)=5\cdot 4\cdot 3\cdot 2\cdot f(1)$$
$$=5\cdot 4\cdot 3\cdot 2\cdot 1\cdot f(0)=5\cdot 4\cdot 3\cdot 2\cdot 1\cdot 1=120$$

Recursive functions

- Recursively defined functions are well defined
- For every positive integer, the value of the function is determined in an unambiguous way
- Given any positive integer, we can use the two parts of the definition to find the value of the function at that integer
- We obtain the same value no matter how we apply two parts of the definition

Example

- Given a recursive definition of a^n, where a is a non-zero real number and n is a non-negative integer
- Note that $a^{n+1}=a\cdot a^n$ and $a^0=1$
- These two equations uniquely define a^n for all non-negative integer n

Example

- Given a recursive definition of $\sum_{k=0}^{n} a_k$
- The first part of the recursive definition
  $\sum_{k=0}^{0} a_k = a_0$
- The second part is
  $\sum_{k=0}^{n+1} a_k = (\sum_{k=0}^{n} a_k) + a_{n+1}$

Example – Fibonacci numbers

- Fibonacci numbers f_0, f_1, f_2, are defined by the equations, $f_0=0$, $f_1=1$, and $f_n=f_{n-1}+f_{n-2}$ for n=2, 3, 4, …
- By definition
  $f_2=f_1+f_0=1+0=1$
  $f_3=f_2+f_1=1+1=2$
  $f_4=f_3+f_2=2+1=3$
  $f_5=f_4+f_3=3+2=5$
  $f_6=f_5+f_4=5+3=8$

Recursively defined sets and structures

- Consider the subset S of the set of integers defined by
  - Basis step: 3∈S
  - Recursive step: if x∈S and y∈S, then x+y∈S
- The new elements formed by this are 3+3=6, 3+6=9, 6+6=12, …
- We will show that S is the set of all positive multiples of 3 (using structural induction)

String

- The set ∑* of strings over the alphabet ∑ can be defined recursively by
  - **Basis step**: λ∈∑* (where λ is the empty string containing no symbols)
  - **Recursive step**: if w∈∑* and x∈∑ then wx ∈∑*
- The basis step defines that the empty string belongs to string
- The recursive step states new strings are produced by adding a symbol from ∑ to the end of stings in ∑*
- At each application of the recursive step, strings containing one additional symbol are generated

Example

- If $\sum=\{0, 1\}$, the strings found to be in $\sum^*$, the set of all bit strings, are
- λ, specified to be in $\sum^*$ in the basis step
- 0 and 1 found in the 1st recursive step
- 00, 01, 10, and 11 are found in the 2nd recursive step, and so on

Concatenation

- Two strings can be combined via the operation of concatenation
- Let $\sum$ be a set of symbols and $\sum^*$ be the set of strings formed from symbols in $\sum$
- We can define the concatenation for two strings by recursive steps
  - **Basis step**: if $w \in \sum^*$, then $w \cdot \lambda = w$, where λ is the empty string
  - **Recursive step**: If $w_1 \in \sum^*$, $w_2 \in \sum^*$ and $x \in \sum$, then $w_1 \cdot (w_2 x) = (w_1 \cdot w_2)x$
  - Oftentimes $w_1 \cdot w_2$ is rewritten as $w_1 w_2$
  - e.g., w_1=abra, and w_2=cadabra, $w_1 w_2$=abracadabra

Length of a string

- Give a recursive definition of *l(w)*, the length of a string w
- The length of a string is defined by
  - $l(\lambda)=0$
  - $l(wx)=l(w)+1$ if $w \in \sum^*$ and $x \in \sum$

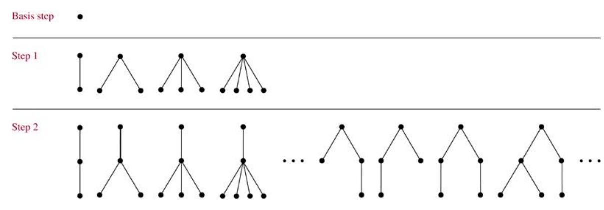
Well-formed formulae

- We can define the set of **well-formed formulae** for compound statement forms involving T, F, proposition variables and operators from the set $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$
- Basis step: T, F, and s, where s is a propositional variable are well-formed formulae
- Recursive step: If E and F are well-formed formulae, then $\neg E$, $E \wedge F$, $E \vee F$, $E \rightarrow F$, $E \leftrightarrow F$ are well-formed formulae
- From an initial application of the recursive step, we know that $(p \vee q)$, $(p \rightarrow F)$, $(F \rightarrow q)$ and $(q \wedge F)$ are well-formed formulae
- A second application of the recursive step shows that $((p \vee q) \rightarrow (q \wedge F))$, $(q \vee (p \vee q))$, and $((p \rightarrow F) \rightarrow T)$ are well-formed formulae

Rooted trees

- The set of rooted trees, where a rooted tree consists of a set of vertices containing a distinguished vertex called the root, and edges connecting these vertices, can be defined recursively by
  - **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 $r_1, r_2, \ldots, r_n$, respectively.
  - Then the graph formed by starting with a root **r**, which is not in any of the rooted trees $T_1, T_2, \ldots, T_n$, and adding an edge from **r** to each of the vertices $r_1, r_2, \ldots, r_n$, is also a rooted tree

Rooted trees



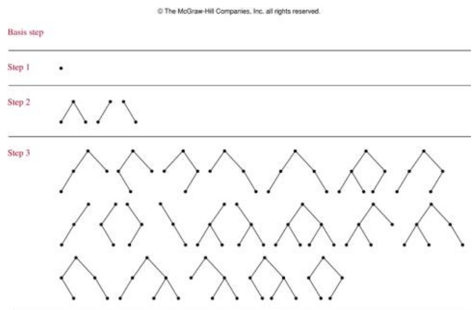
Binary trees

- At each vertex, there are **at most two branches** (one left subtree and one right subtree)
- **Extended binary trees**: the left subtree **or** the right subtree can be empty
- **Full binary trees**: must have left and right subtrees

Extended binary trees

- The set of **extended binary trees** can be defined by
  - **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 by $T_1 \cdot T_2$, consisting of a root **r** together with edges connecting the root to each of the roots of the left subtree T_1 and right subtree T_2, when these trees are non-empty

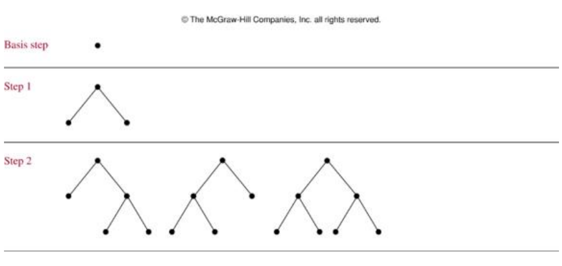
Extended binary trees



Full binary trees

- The set of **full binary trees** can be defined recursively
  - **Basis step**: There is a full binary tree consisting only of a single vertex **r**
  - **Recursive step**: If T_1 and T_2 are disjoint full binary trees, there is a full binary tree, denoted by $T_1 \cdot T_2$, consisting of a root r together with edges connecting the root to each of the roots of the left subtree T_1 and right subtree T_2

Full binary tree



The set of *rooted trees,* where a rooted tree consists of a set of vertices containing a distinguished vertex called the *root,* and edges connecting these vertices, can be defined recursively by these steps:

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 $r_1, r_2, \ldots, r_n$, respectively. Then the graph formed by starting with a root r, which is not in any of the rooted trees $T_1, T_2, \ldots, T_n$, and adding an edge from r to each of the vertices $r_1, r_2, \ldots, r_n$, is also a rooted tree.

The set of *extended binary trees* can be defined recursively by these steps:

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 by $T_1 \cdot T_2$, consisting of a root r together with edges connecting the root to each of the roots of the left subtree T_1 and the right subtree T_2 when these trees are nonempty.

The set of full binary trees can be defined recursively by these steps:

BASIS STEP: There is a full binary tree consisting only of a single vertex r.

RECURSIVE STEP: If T_1 and T_2 are disjoint full binary trees, there is a full binary tree, denoted by $T_1 \cdot T_2$, consisting of a root r together with edges connecting the root to each of the roots of the left subtree T_1 and the right subtree T_2.

FIGURE 2 Building Up Rooted Trees.

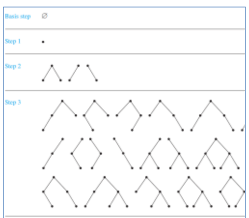


FIGURE 3 Building Up Extended Binary Trees.

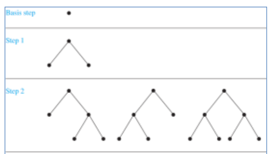


FIGURE 4 Building Up Full Binary Trees.

Structural induction

- Show that the set S defined by
  - 3∈S and
  - if x∈S and y∈S, then x+y∈S,

  is the set of multiples of 3
- Let **A** be the set of all positive integers divisible by 3
- **To prove A=S, we must show that A⊆S, and S⊆A**
- To show A⊆S, <u>we must show that every positive integer divisible by 3 is in S</u>
- Use **mathematical induction** to prove it

Structural induction

- Let p(n) be the statement that **3n** belongs to **S**
- **Basis step**: it holds as the first part of recursive definition of S, 3·1=3∈S
- **Inductive step**: assume that p(k) is true, i.e., 3k is in S. As 3k∈S and 3∈S, it follows from the 2nd part of the recursive definition of S that 3k+3=3(k+1)∈S. So p(k+1) is true

Structural induction

- To show that S⊆A, we **use recursive definition** of S
- The basis step of the definition specifies that **3 is in S**
- As 3=3·1, all elements specified to be in S in this step are divisible by 3, and there in A
- To finish the proof, <u>we need to show that all integers in S generated using the 2nd part of the recursive definition are in A</u>
- This consists of showing that **x+y** is in A whenever **x** and **y** are elements of S also assumed to be in A
- If **x** and **y** are both in A, it follows that **3|x**, **3|y**, and thus **3|x+y**, thereby completing the proof

Trees and structural induction

- To prove properties of trees with structural induction
  - **Basis step**: show that the result is true for the tree consisting of a single vertex
  - **Recursive step**: show that if the result is true for the trees T_1 and T_2, then it is true for $\mathbf{T_1 \cdot T_2}$, consisting of a root **r**, which has T_1 as its **left** subtree and T_2 as its **right** subtree

Height of binary tree

- We define the height h(T) of a full binary tree T recursively
  - **Basis step**: the height of the full binary tree T consisting of only a root r is h(T)=0
  - **Recursive step**: If T_1 and T_2 are full binary trees, then the full binary tree $T= T_1 \cdot T_2$ has height $\mathbf{h(T)=1+max(h(T_1), h(T_2))}$

Number of vertices in a binary tree

- If we let n(T) denote the number of vertices in a full binary tree, we observe that n(T) satisfies the following recursive formula:
  - **Basis step**: the number of vertices n(T) of the full binary tree consisting of only a root r is n(T)=1
  - **Recursive step**: If T_1 and T_2 are full binary trees, then the number of vertices of the full binary tree $T= T_1 \cdot T_2$ is $\mathbf{n(T)=1+n(T_1)+n(T_2)}$

Theorem

- If T is a full binary tree T, then $\mathbf{n(T) \le 2^{h(T)+1}-1}$
- Use structural induction to prove this
- **Basis step**: for the full binary tree consisting of just the root r the result is true as n(T)=1 and h(T)=0, so $n(T)=1 \le 2^{0+1}-1=1$
- **Inductive step**: For the inductive hypothesis we assume that $n(T_1) \le 2^{h(T_1)+1}-1, \ n(T_2) \le 2^{h(T_2)+1}-1$ where T_1 and T_2 are full binary trees

Theorem

- By the recursive formulae for n(T) and h(T), we have **$n(T)=1+n(T_1)+n(T_2)$** and **$h(T)=1+\max(h(T_1), h(T_2))$**

$n(T) = 1 + n(T_1) + n(T_2)$	by the recursive formula for $n(T)$
$\leq 1 + (2^{h(T_1)+1} - 1) + (2^{h(T_2)+1} - 1)$	by the inductive hypothesis
$\leq 2 \cdot \max(2^{h(T_1)+1}, 2^{h(T_2)+1}) - 1$	because the sum of two terms is at most 2 times the larger
$= 2 \cdot 2^{\max(h(T_1),h(T_2))+1} - 1$	because $\max(2^x, 2^y) = 2^{\max(x,y)}$
$= 2 \cdot 2^{h(T)} - 1$	by the recursive definition of $h(T)$
$= 2^{h(T)+1} - 1.$	

- This completes the inductive step