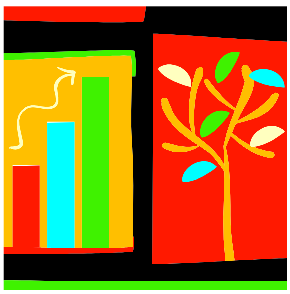
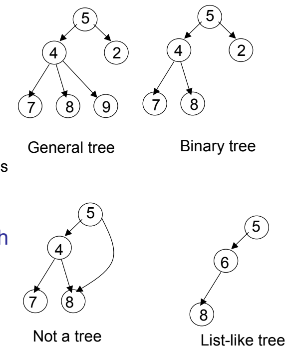
Trees



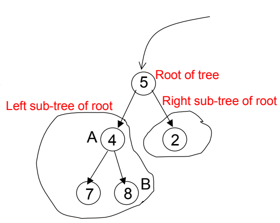
Overview

- Tree: recursive data structure (similar to list)
  - each cell may have two or more successors (children)
  - each cell has at most one predecessor (parent)
    - distinguished cell called root has no parent
  - all cells are reachable from root
- Binary tree: tree in which each cell can have at most two children



Terminology

- Edge A→B: A is said to be parent of B, and B is said to be its child
- Generalization of parent and child: ancestor and descendant
  - root and A are ancestors of B
- Leaf node: node with no descendants
- Depth of node: length of path from root to that node
  - depth(A) = 1   depth(B) = 2
- Height of node: length of longest path from node to leaf
  - height(A) = 1   height(B) = 0
- Height of tree = height of root
  - in example, height of tree = 2



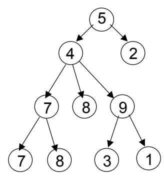
Binary tree

Class for binary tree cells

```
class TreeCell {
   protected Object datum;
   protected TreeCell left;
   protected TreeCell right;

   public TreeCell(Object o) {
    datum = o;
   }
   public TreeCell(Object o, TreeCell l, TreeCell r) {
      datum = o;
      left = l;
      right = r;
   }
   methods called getDatum, setDatum,
   getLeft, setLeft, getRight, setRight
   with obvious code
}
```

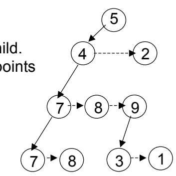
Class for general trees

```
class GTreeCell{
  protected Object datum;
  protected GTreeCell left;
  protected GTreeCell sibling;
  ….appropriate getter and setter methods
}
```



General tree

- Parent node points directly only to its leftmost child.
- Leftmost child has pointer to next sibling which points to next sibling etc.



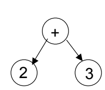
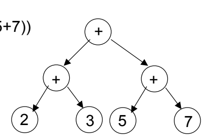
Tree represented using GTreeCell

Applications of trees

- Most languages (natural and computer) have a recursive, hierarchical structure.
- This structure is implicit in ordinary textual representation.
- Recursive structure can be made explicit by representing sentences in the language as trees: abstract syntax trees (AST's)
- AST's are easier to optimize, generate code from, etc. than textual representation.
- Converting textual representations to AST: job of parser

Example

- Expression grammar:
  - E → integer
  - E → (E + E)
- Tree representation:
  - first rule: create a node with integer
  - second rule: create a node with "+" as datum, tree for first operand as its left sub-tree, and tree for second operand as its right sub-tree
- In textual representation, parentheses show hierarchical structure. In tree representation, hierarchy is explicit in the structure of the tree.

Text	Tree representation
-34	
(2 + 3)	
((2+3) + (5+7))	

AST generation for simple expression language

```
static  TreeCell expCodeGen(String fileName) {//returns AST for expression in file
        CS211In f  = new CS211In(fileName);
        return getExp(f); //no error checking to keep things simple
}
static TreeCell getExp(CS211In f) {//no error checking to keep it simple
        switch (f.peekAtKind()) {
          case CS211In.INTEGER: //E → integer
                   return  new TreeCell(f.getInt());
          case CS211In.OPERATOR: //E→(E+E)
                                    {     f.check('(');
                                          TreeCell left = getExp(f);
                                          f.check('+');
                                          TreeCell right = getExp(f);
                                          f.check(')');
                                          return new TreeCell("+", left, right);
                                    }
          default:   return null; //error
        }
}
```

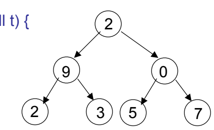
Recursion on trees

- Recursive methods can be written to operate on trees in the obvious way.
- In most problems
  - base case: empty tree
    - sometimes base case is leaf node
  - recursive case: solve problem on left and right sub-trees, and then put solutions together to compute solution for tree

Tree search

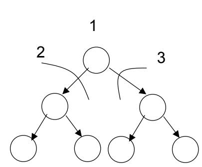
- Analog of linear search in lists: given tree and an object, find out if object is stored in tree.
- Trivial to write recursively; harder to write iteratively.

```
public static boolean treeSearch(Object o, TreeCell t) {
   if (t  == null) return false;
   else return t.getDatum().equals(o) ||
                 treeSearch(o, t.getleft()) ||
                 treeSearch(o, t.getRight());
}
```



Walks of tree

- Example on last slide showed pre-order walk of tree:
  - process root
  - process left sub-tree
  - process right sub-tree
- Intuition: think of prefix representation of expressions



In-order and post-order walks

- In-order walk: infix
  - process left sub-tree
  - process root
  - process right sub-tree

```
public static boolean treeSearch(Object o,
TreeCell t) {
   if (t  == null) return false;
   else return
                 treeSearch(o, t.getleft()) ||
                 t.getDatum().equals(o)   ||
                 treeSearch(o, t.getRight());
}
```

- Post-order walk: postfix
  - process left sub-tree
  - process right sub-tree
  - process root

```
public static boolean treeSearch(Object o,
TreeCell t) {
   if (t  == null) return false;
   else return
                 treeSearch(o, t.getleft()) ||
                 treeSearch(o, t.getRight())||
                 t.getDatum().equals(o);
}
```

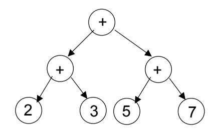
Some useful routines

```
//determine is a TreeCell is a leaf node
public static boolean isLeaf(TreeCell t) {
   return (t != null) && (t.getLeft() == null) && (t.getRight() == null);
}

//compute height of tree using post-order walk
public static int height(TreeCell t) {
    if (t == null) return –1; //height is undefined for empty tree
    if (isLeaf(t)) return 0;
    else return 1 + Math.max(height(t.getLeft()), height(t.getRight()));
}

//compute number of nodes in tree using post-order walk
public static int nNodes(TreeCell t) {
   if (t == null) return 0;
   else return 1 + nNodes(t.getLeft()) + nNodes(t.getRight());
}
```

Example

- Generate textual representation from AST.

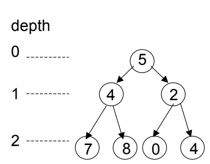
```
public static String flatten(TreeCell t) {
      if (t == null) return “”;
      if (isLeaf(t)) return t.getDatum();
      else return “(“ + flatten(t.getLeft()) + t.getDatum() + flatten(t.getRight()) + “)” ;
```



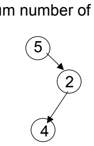
Useful facts about binary trees

- Maximum number of nodes at depth d = 2^d
- If height of tree is h,
  - minimum number of nodes it can have = h+1
  - maximum number of nodes it can have is =

    $2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$
- Full binary tree of height h:
  - all levels of tree upto depth h are completely filled.



Height 2,
maximum number of nodes



Height 2,
minimum number of nodes

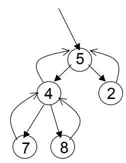
Tree with header element

- As in case of lists, some authors prefer to have an explicit Tree class which contains a reference to the root of the tree.
- With this design, methods that operate on trees can be made into instance methods in this class, and the root of the tree does not have to be passed in explicitly to method.
- Feel free to use whatever works for you.

Tree with parent pointers

- In some applications, it is useful to have trees in which nodes other than root have references to their parents.
- Tree analog of doubly-linked lists.

```
class TreeWithPPCell{
  protected Object datum;
  protected TreeWithPPCell
            left, right, parent;
  …..appropriate getter and
  setter methods…
}
```



Summary

- Tree is a recursive data structure built from TreeCell class.
  – special case: binary tree
- Binary tree cells have both a left and a right “successor”
  – called children rather than successors
  – similarly, parent rather than predecessor
  – generalization of parent and child to ancestors and descendants
- Trees are useful for exposing the recursive structure of natural language programs and computer programs.