# Trees and tree search

- Trees are a useful data structure
  - Convenient for storing data that is hierarchical
  - Stores data in a manner that simplifies searching for information
  - Can capture information
- Search algorithms for trees can be quite efficient
- Particularly useful for making decisions on problems

Example trees

Tree definition

- A tree consists of one or more nodes
  - A node typically has a value associated with it
- Nodes are connected by branches
- A tree starts with a root node
- Except for leaves, each node has one or more children
  - We refer to a node which has a child as the parent node
- In simple trees, no child has more than one parent, but the generalization (often called a graph) is also very useful
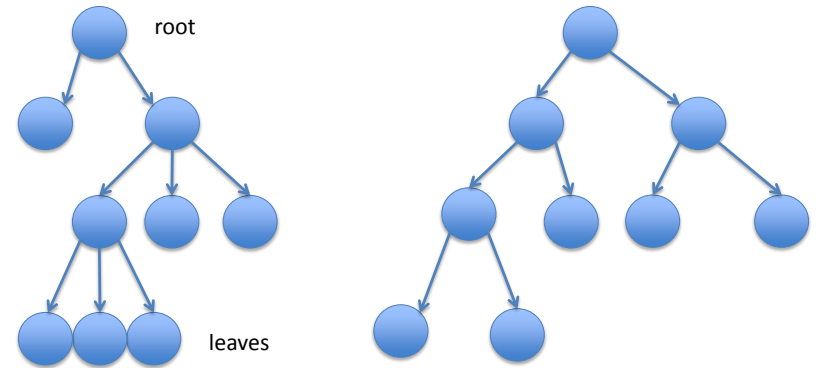
Binary trees

- A binary tree is a special version of a tree, where each node has at most two children
- Binary trees are very useful when storing and searching ordered data, or when determining the best decision to make in solving many classes of problems
  - Such trees are often called decision trees

Binary tree class

```python
class binaryTree(object):
    def __init__(self, value):
        self.value = value
        self.leftBranch = None
        self.rightBranch = None
        self.parent = None
    def setLeftBranch(self, node):
        self.leftBranch = node
    def setRightBranch(self, node):
        self.rightBranch = node
    def setParent(self, parent):
        self.parent = parent
    # and other methods
```

Constructing an example tree

```python
n5 = binaryTree(5)       n5.setLeftBranch(n2)
n2 = binaryTree(2)       n2.setParent(n5)
n1 = binaryTree(1)       n5.setRightBranch(n8)
n4 = binaryTree(4)       n8.setParent(n5)
n8 = binaryTree(8)       n2.setLeftBranch(n1)
n6 = binaryTree(6)       n1.setParent(n2)
n7 = binaryTree(7)       n2.setRightBranch(n4)
                         n4.setParent(n2)
                         n8.setLeftBranch(n6)
                         n6.setParent(n8)
                         n6.setRightBranch(n7)
                         n7.setParent(n6)
```

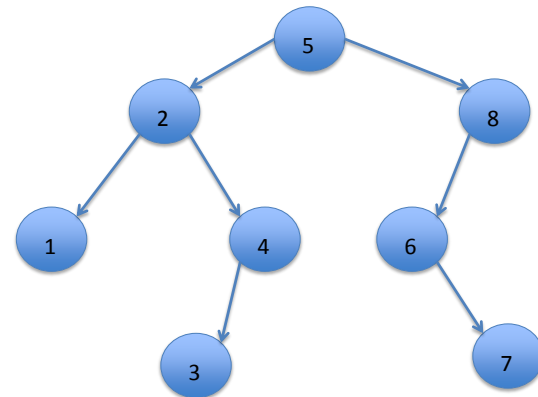## Binary tree class

```python
class binaryTree(object):
    # and other methods
    def getValue(self):
        return self.value
    def getLeftBranch(self):
        return self.leftBranch
    def getRightBranch(self):
        return self.rightBranch
    def getParent(self):
        return self.parent
    def __str__(self):
        return self.value
```

An example tree

13 - 2

Searching a tree

- Imagine we want to examine a tree
  - To determine if an element is present
  - To find a path to a solution point
 - To make a series of decisions to reach some objective

Depth first search
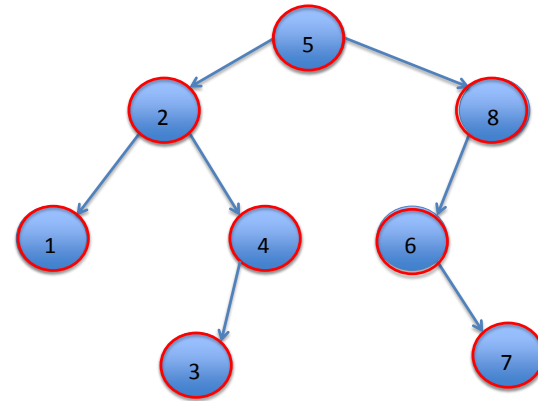
Searching a tree

- Depth first search
  - Start with the root
  - At any node, if we haven't reached our objective, take the left branch first
  - When get to a leaf, backtrack to the first decision point and take the right branch
- Breadth first search
  - Start with the root
  - Then proceed to each child at the next level, in order
  - Continue until reach objective

Breadth first search

13 - 3
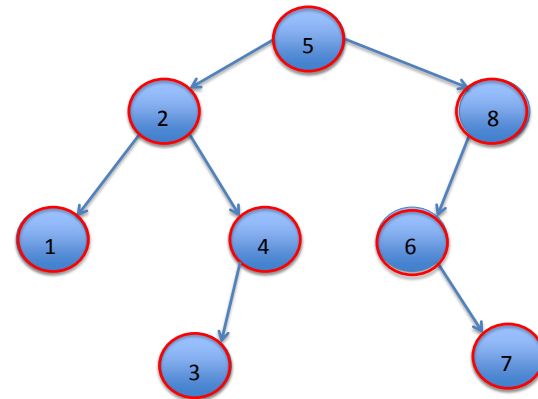
Depth first search for containment

- Idea is to keep a data structure (called a stack) that holds nodes still to be explored
- Use an evaluation function to determine when reach objective (i.e. for containment, whether value of node is equal to desired value)
- Start with the root node
- Then add children, if any, to front of data structure, with left branch first
- Continue in this manner

Depth first search

5	3, 8
2, 8	8
1, 4, 8	6
4, 8	

DFS code

```python
def DFSBinary(root, fcn):
    stack= [root]
    while len(stack) > 0:
        if fcn(stack[0]):
            return True
        else:
            temp = stack.pop(0)
            if temp.getRightBranch():
                stack.insert(0,
temp.getRightBranch())
            if temp.getLeftBranch():
                stack.insert(0,
temp.getLeftBranch())
    return False
```
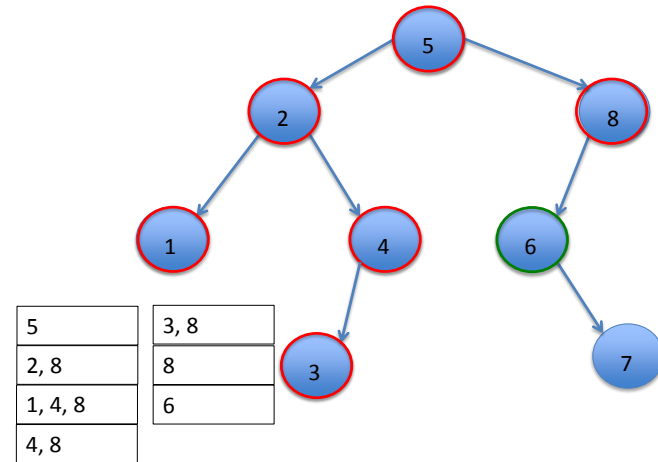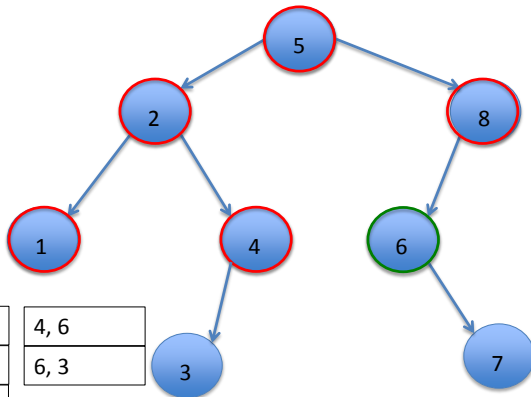
BFS code

```python
Def BFSBinary(root, fcn):
    queue = [root]
    while len(queue) > 0:
        if fcn(queue[0]):
            return True
        else:
            temp = queue.pop(0)
            if temp.getLeftBranch():
                queue.append(temp.getLeftBranch())
            if temp.getRightBranch():
                queue.append(temp.getRightBranch())
    return False
```

Breadth first search

5		4, 6	
2, 8		6, 3	
8, 1, 4			
1, 4, 6			

DFS code

```python
def DFSBinaryPath(root, fcn):
    stack= [root]
    while len(stack) > 0:
        if fcn(stack[0]):
            return TracePath(stack[0])
        else:
            temp = stack.pop(0)
            if temp.getRightBranch():
                stack.insert(0, temp.getRightBranch())
            if temp.getLeftBranch():
                stack.insert(0, temp.getLeftBranch())
    return False

def TracePath(node):
    if not node.getParent():
        return [node]
    else:
        return [node] + TracePath(node.getParent())
```

Depth first search for path

- Suppose we want to find the actual path from root node to desired node
- A simple change in the code lets us trace back up the tree, once we find the desired node
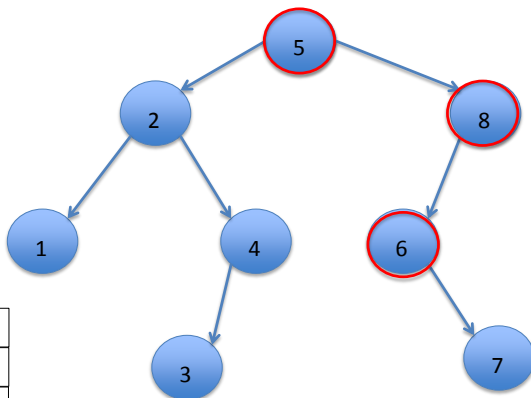
Ordered search

- Suppose we know that the tree is ordered, meaning that for any node, all the nodes to the "left" are less than that node's value, and all the nodes to the "right" are greater than that node's value

DFS code

```python
def DFSBinaryOrdered(root, fcn, ltFcn):
    stack= [root]
    while len(stack) > 0:
        if fcn(stack[0]):
            return True
        elif ltFcn(stack[0]):
            temp = stack.pop(0)
            if temp.getLeftBranch():
                stack.insert(0,
temp.getLeftBranch())
        else:
            if temp.getRightBranch():
                stack.insert(0,
temp.getRightBranch())
    return False
```

Decision Trees

- A decision tree is a special type of binary tree (though could be more general tree with multiple children)
- At each node, a decision is made, with a positive decision taking the left branch, and a negative decision taking the right branch
- When we reach a leaf that satisfies some goal, the path back to the root node defines the solution to the problem captured by the tree

Depth first ordered search

| 5 |
| 8 |
| 6 |

Building a decision tree

- One way to approach decision trees is to construct an actual tree, then search it
- An alternative is to implicitly build the tree as needed
- As an example, we will build a decision tree for a knapsack problem

The Knapsack Problem

- Suppose we are given a set of objects, each with a value and a weight
- We have a finite sized knapsack, into which we want to store some of the items
- We want to store the items that have the most value, subject to the constraint that there is a limit to the cumulative size that will fit
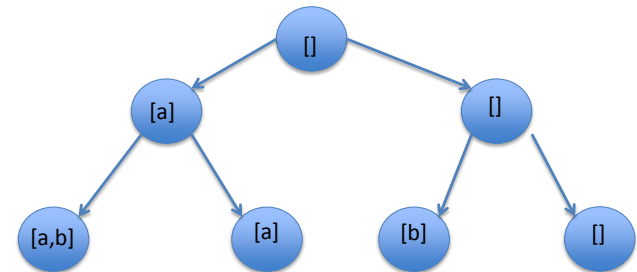
Building a decision tree

```python
def buildDTree(sofar, todo):
    if len(todo) == 0:
        return binaryTree(sofar)
    else:
        withelt = buildDTree(sofar + [todo[0]], todo[1:])
        withoutelt = buildDTree(sofar, todo[1:])
        here = binaryTree(sofar)
        here.setLeftBranch(withelt)
        here.setRightBranch(withoutelt)
        return here
```

Building a decision tree

- For the knapsack problem, we can build a decision tree as follows:
  - At the root level, we decide whether to include the first element (left branch) or not (right branch)
  - At the nth level, we make the same decision for the nth element
  - By keeping track of what we have included so far, and what we have left to consider, we can generate a binary tree of decisions

Example decision tree

Given elements a and b, decide whether to include them

Searching a decision tree

```python
def DFSDTree(root, valueFcn, constraintFcn):
    stack= [root]
    best = None
    visited = 0
    while len(queue) > 0:
        visited += 1
        if constraintFcn(stack[0].getValue()):
            if best == None:
                best = stack[0]
            elif valueFcn(stack[0].getValue()) > valueFcn(best.getValue()):
                best = stack[0]
            temp = stack.pop(0)
            if temp.getRightBranch():
                queue.insert(0, temp.getRightBranch())
            if temp.getLeftBranch():
                queue.insert(0, temp.getLeftBranch())
        else:
            stack.pop(0)
    print 'visited', visited
    return best
```

An example

```python
a = [6,3]
b = [7,2]
c = [8,4]
d = [9,5]


treeTest = buildDTree([], [a,b,c,d])


def sumValues(lst):
    vals = [e[0] for e in lst]
    return sum(vals)


def WeightsBelow10(lst):
    wts = [e[1] for e in lst]
    return sum(wts) <= 10

DFSDTree(treeTest, sumValues, WeightsBelow10)
BFSDTree(treeTest, sumValues, WeightsBelow10)
```

Searching a decision tree

```python
def BFSDTree(root, valueFcn, constraintFcn):
    queue = [root]
    best = None
    visited = 0
    while len(queue) > 0:
        visited += 1
        if constraintFcn(queue[0].getValue()):
            if best == None:
                best = queue[0]
            elif valueFcn(queue[0].getValue()) > valueFcn(best.getValue()):
                best = queue[0]
            temp = queue.pop(0)
            if temp.getLeftBranch():
                queue.append(temp.getLeftBranch())
            if temp.getRightBranch():
                queue.append(temp.getRightBranch())
        else:
            queue.pop(0)
    print 'visited', visited
    return best
```

Decision Trees

- Depth first and breadth first still search the same number of nodes, the order is simply different
- If we are willing to settle for "good enough", then there is a difference in work done by the two search methods

Searching a decision tree

```python
def DFSDTreeGoodEnough(root, valueFcn, constraintFcn, stop):
    stack= [root]
    best = None
    visited = 0
    while len(queue) > 0:
        visited += 1
        if constraintFcn(stack[0].getValue()):
            if best == None:
                best = stack[0]
            elif valueFcn(stack[0].getValue()) > valueFcn(best.getValue()):
                best = stack[0]
            if stop(best.getValue()):
                return best
            temp = stack.pop(0)
            if temp.getRightBranch():
                queue.insert(0, temp.getRightBranch())
            if temp.getLeftBranch():
                queue.insert(0, temp.getLeftBranch())
        else:
            stack.pop(0)
    print 'visited', visited
    return best
```

Testing "good enough"

```python
def atLeast15(lst):
    return sumValues(lst) >= 15
```

```
print 'DFS'
depth = DFSDTreeGoodEnough(treeTest,
sumValues, WeightsBelow10, atLeast15)


print 'BFS'
breadth = BFSDTreeGoodEnough(treeTest,
sumValues, WeightsBelow10, atLeast15)
```

Searching a decision tree

```python
def BFSDTreeGoodEnough(root, valueFcn, constraintFcn, stop):
    queue = [root]
    best = None
    visited = 0
    while len(queue) > 0:
        visited += 1
        if constraintFcn(queue[0].getValue()):
            if best == None:
                best = queue[0]
            elif valueFcn(queue[0].getValue()) > valueFcn(best.getValue()):
                best = queue[0]
            if stop(best.getValue()):
                return best
            temp = queue.pop(0)
            if temp.getLeftBranch():
                queue.append(temp.getLeftBranch())
            if temp.getRightBranch():
                queue.append(temp.getRightBranch())
        else:
            queue.pop(0)
    print 'visited', visited
    return best
```

Searching an implicit tree

- Our approach is inefficient, as it constructs the entire decision tree, and then searches it
- An alternative is only generate the nodes of the tree as needed
- Here is an example for the case of a knapsack problem, the same idea could be captured in other search problems

Implicit search for knapsack

```python
def DTImplicit(toConsider, avail):
    # return value of solution, and solution
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0][1] > avail:
        result = DTImplicit(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        withVal, withToTake = DTImplicit(toConsider[1:], avail -
nextItem[1])
        withVal += nextItem[0]
        withoutVal, withoutToTake = DTImplicit(toConsider[1:],
avail)
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```

An example "tree"
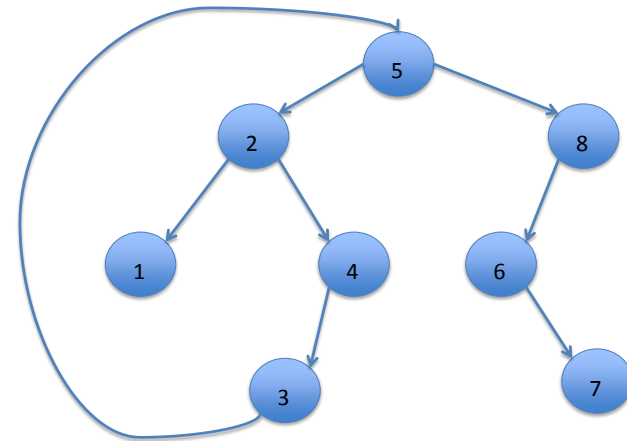
What if our trees are overgrown

- We have been implicitly assuming that there are no "loops" in our trees, i.e. that a child has one parent, and that no node is the parent of a child closer to the root
- What if we relax this constraint?
- Generalization is called a graph
  - Lots of great graph search problems
  - For now, we can think about ways to support search for binary trees that might have loops

Searching these "trees"

- What happens if we run depth first search on this?

  An infinite loop in many cases when item present, and always if item not present

- What happens if we run breadth first search on this?

  Inefficient as repeats nodes, but still works if item present, infinite loop if not present

Avoiding loops

```python
def DFSBinaryNoLoop(root, fcn):
    stack = [root]
    seen = []
    while len(stack) > 0:
        print 'at node ' + str(queue[0].getValue())
        if fcn(stack[0]):
            return True
        else:
            temp = stack.pop(0)
            seen.append(temp)
            if temp.getRightBranch():
                if not temp.getRightBranch() in seen:
                    stack.insert(0, temp.getRightBranch())
            if temp.getLeftBranch():
                if not temp.getLeftBranch() in seen:
                    stack.insert(0, temp.getLeftBranch())
    return False
```