CSC230

Intro to C++ Lecture 19

Outline

- Lab 9 discussion

- TREE

Why we need tree ?

- What is the most important operation on data ?
  - Search

  - Array
  - Linked List
  - HashMap

Problem with Linked List

Searching in Linked List

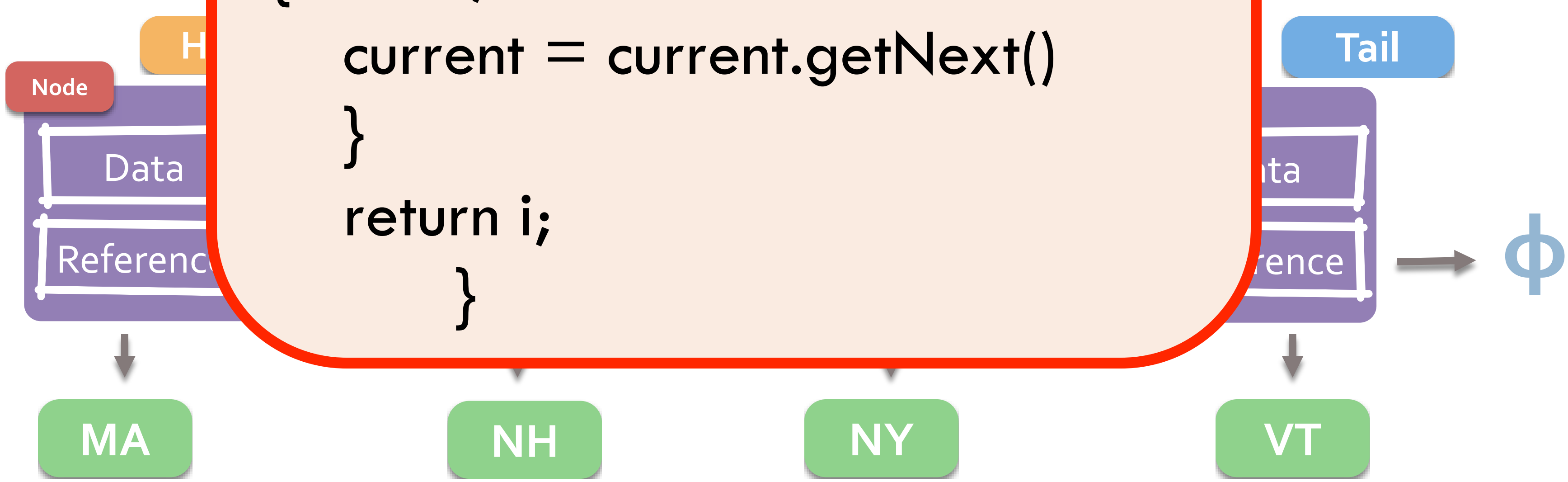
- Searching in a linked list is time consuming
    - Traverse from the very beginning
    - How many nodes you have to visit if
        - Searching 1
        - Searching 2
        - Searching 3
        - …
        - Searching 7

Review: Linear Search

Traversing the Linked List

- Travers… …ist
- Singly…

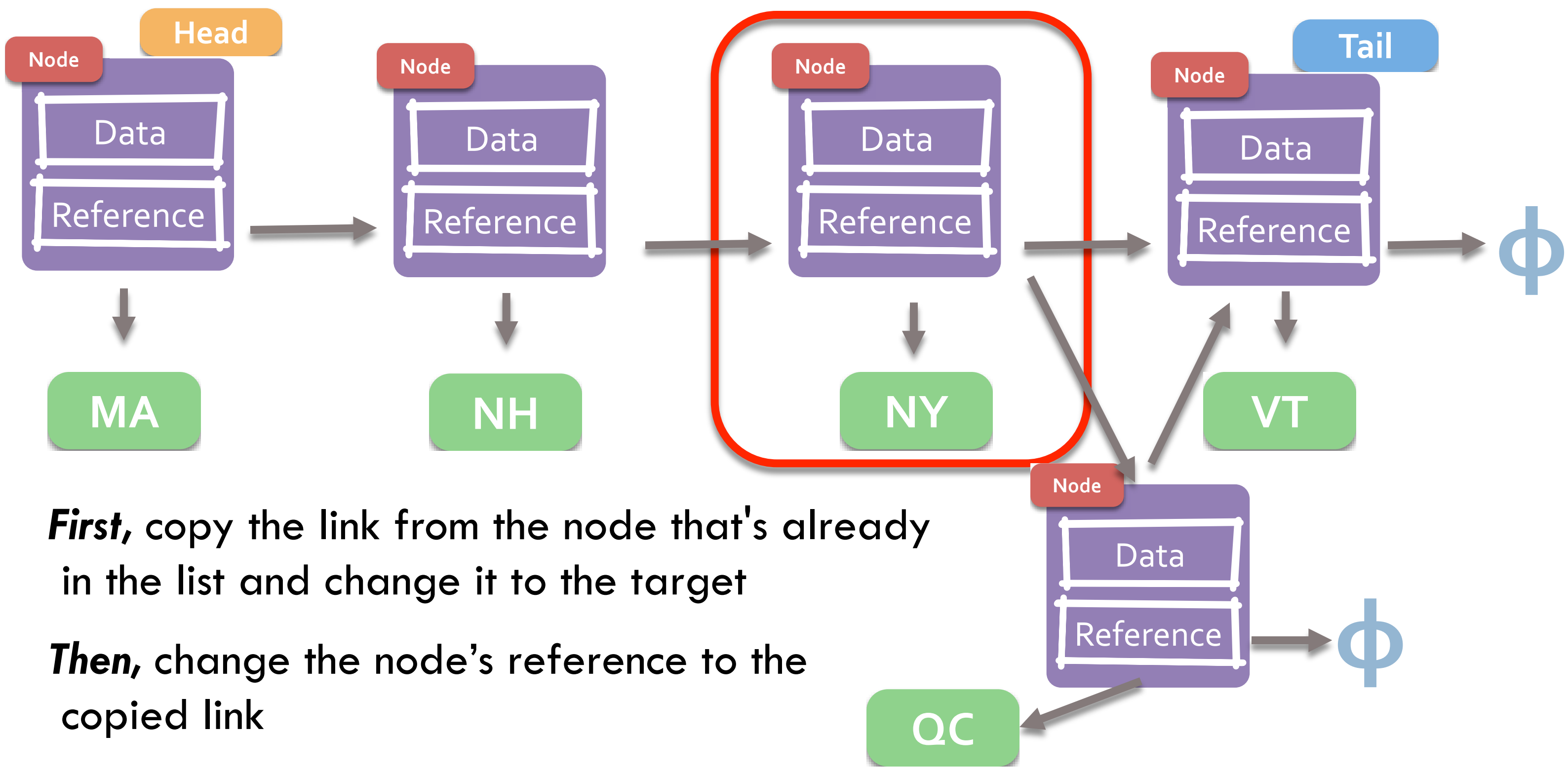
```
Public int getSize()
  {    int i=0;
        Node current = head;
        while (current ! = NULL)
{   i++ ;
     current = current.getNext()
     }
     return i;
          }
```



Review: Linear Search

Insertion on Linked List

- How to insert QC after NY (middle)?
  - Create a new node with QC as the element
  - Find the node you want to insert after



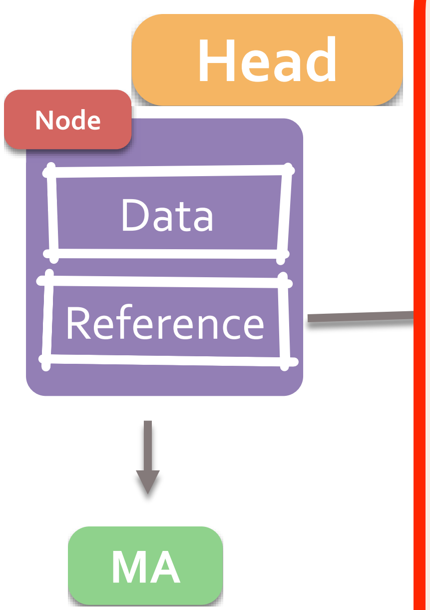
First, copy the link from the node that's already in the list and change it to the target

Then, change the node's reference to the copied link

Review: Linear Search

Search on the Linked List

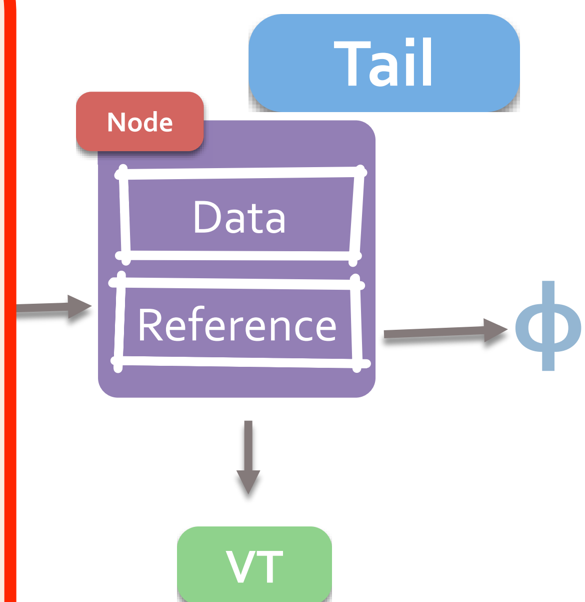
- How to find
  - **Fin**




```
public int GetNth(int index)
  {
      Node current = head;
      int count = 0;

    while (current != null)
       {
          if (count == index)
              return current.data;
          count++;
          current = current.next;
       }

System.out.println("Access failed");
       return 0;
  }
```



Binary Search

Idea of Binary Search

- How many nodes you have to visit to find 93 with linear search algorithm ?
- How many nodes you have to visit to find 33 with linear search algorithm ?

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Did you see the numbers in small size ?

Binary Search

Idea of Binary Search

- Search for number 93
- Based on the indexes, how can we find it faster ?
- In other words, how to make use of the indexes ?
    - Record the low, high and mid
    - Start the search from the middle

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

low **mid** **high**

Binary Search

Idea of Binary Search

- Search for number 93
- Based on the indexes, how can we find it faster ?
- In other words, how to make use of the indexes ?
  - Compare the target with the middle
  - Focus on the right side only
  - What is our next step ?

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑
low							**mid**							**high**

Binary Search

Idea of Binary Search

- Search for number 93
- Based on the indexes, how can we find it faster ?
- In other words, how to make use of the indexes ?
    - Compare the target with the middle
    - Focus on the right side only
    - What is our next step ?

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
								low			mid			high

- Now, how to move forward ?

Binary Search

Idea of Binary Search

- Search for number 96 with binary search algorithm
- Which nodes you will visit ?
- First step : 53

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

low **mid** **high**

Binary Search

Idea of Binary Search

- Search for number 96 with binary search algorithm
- Which nodes you will visit ?
- First step : 53
- Second step : 93

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
								↑			↑			↑
								low			mid			high

Binary Search

Idea of Binary Search

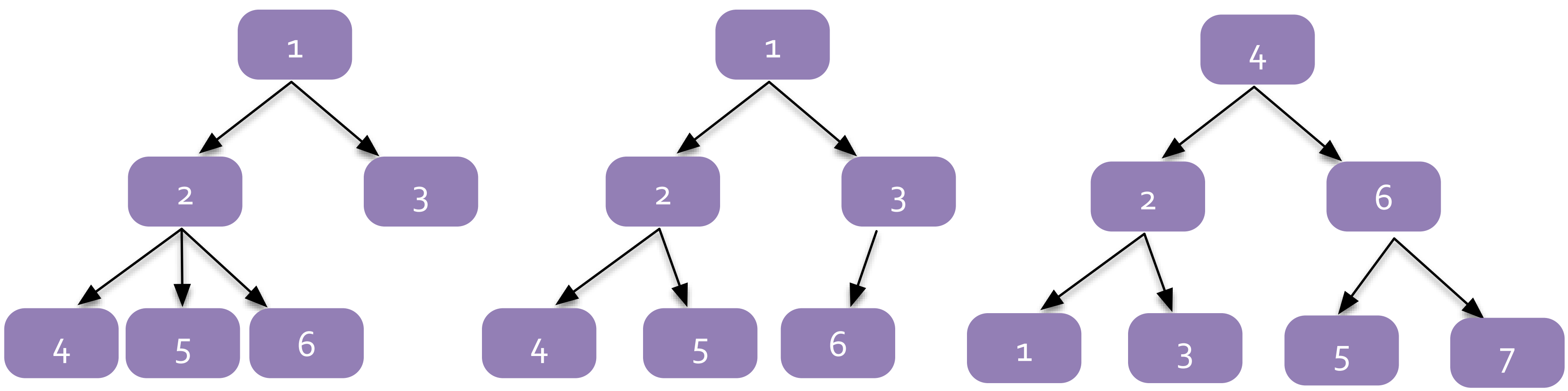
- Search for number 96 with binary search algorithm
- Which nodes you will visit ?
- First step : 53
- Second step : 93
- Third step : 96

													mid	
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
												low		high

To solve the problem: BST

Binary Search Tree

- It is a tree structure
  - Each node can have at most two children
- It is a special tree
  - For every node in the tree, its key is greater than its left child's key and less than its right child's key



To solve the problem: BST

Searching in Binary Search Tree

items	Linked List	BST
1	1	1
3	2	1.67
7	4	2.43
15	8	3.29
31	16	4.16
63	32	5.09

key	Linked List	BST
1	1	3
2	2	2
3	3	3
4	4	1
5	5	3
6	6	2
7	7	3
Average	4	2.43

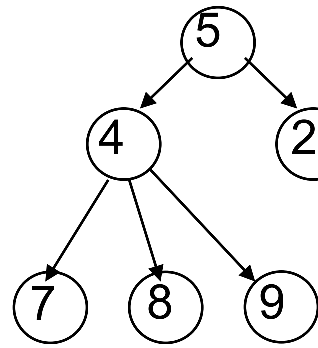
1 → 2 → 3 → 4 → 5 → 6 → 7

Tree Overview

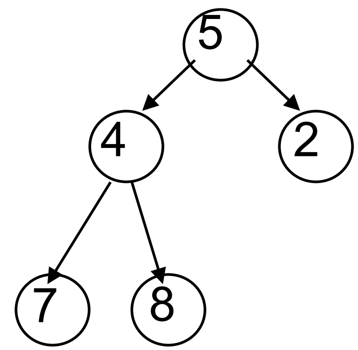
Tree: **recursive** data structure (similar to list)

- Each node may have zero or more *successors* (children)
- Each node has **exactly one** *predecessor* (parent) except the *root*, which has none
- All nodes are reachable from *root*

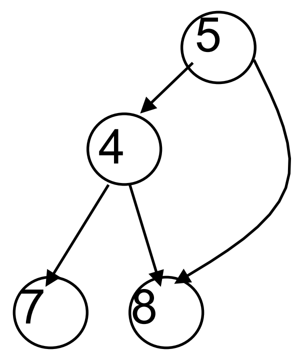
Binary tree: tree in which each node can have at most **two** children: a left child and a right child

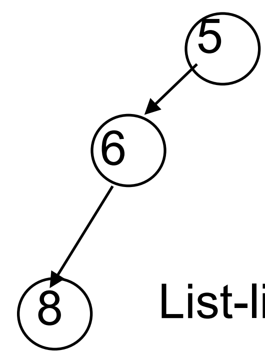


General tree



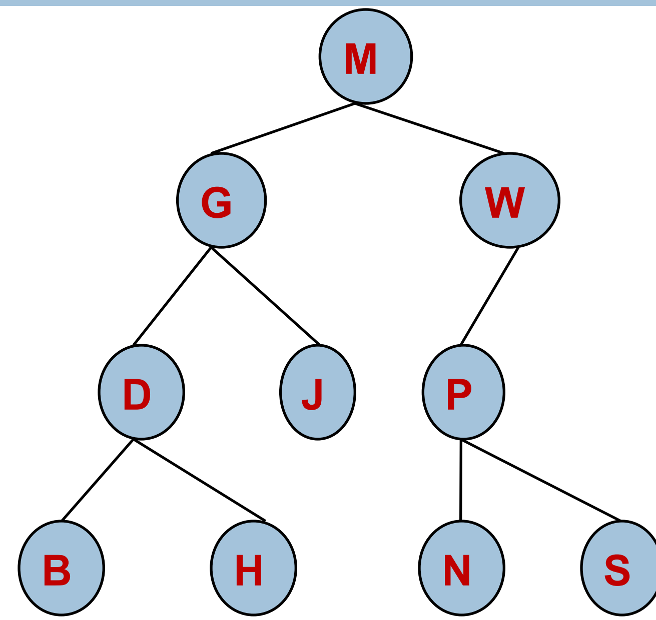
Binary tree





List-like tree

Tree Terminology



M: *root* of this tree

G: *root* of the *left subtree* of M

B, H, J, N, S: *leaves*

N: *left child* of P; S: right *child*

P: parent of N

M and G: *ancestors* of D

P, N, S: *descendents* of W

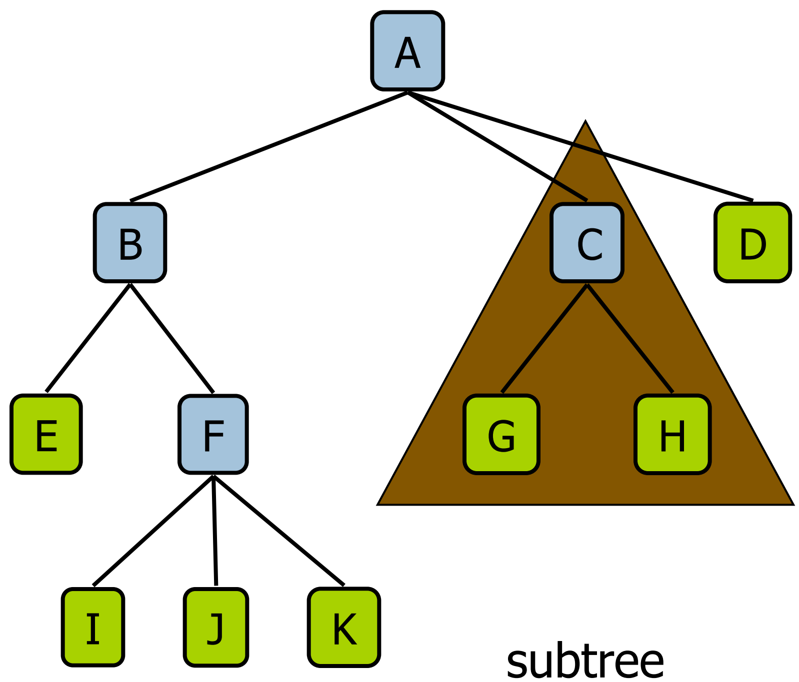
J is at *depth* 2 (i.e. length of path from **root** = no. of edges)

W is at *height* 2 (i.e. length of longest path to a **leaf**)

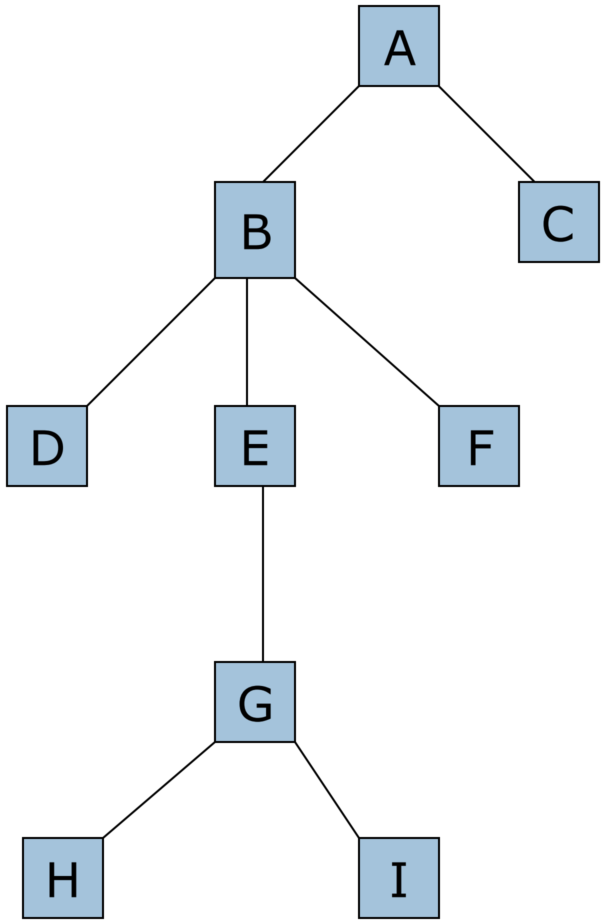
Tree Terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree** of a node: the number of its children
- **Degree** of a tree: the maximum degree of its node.

- **Subtree**: tree consisting of a node and its descendants



Tree Properties



Property

Number of nodes

Height

Root Node

Leaves

Interior nodes

Ancestors of H

Descendants of B

Siblings of E

Right subtree of A

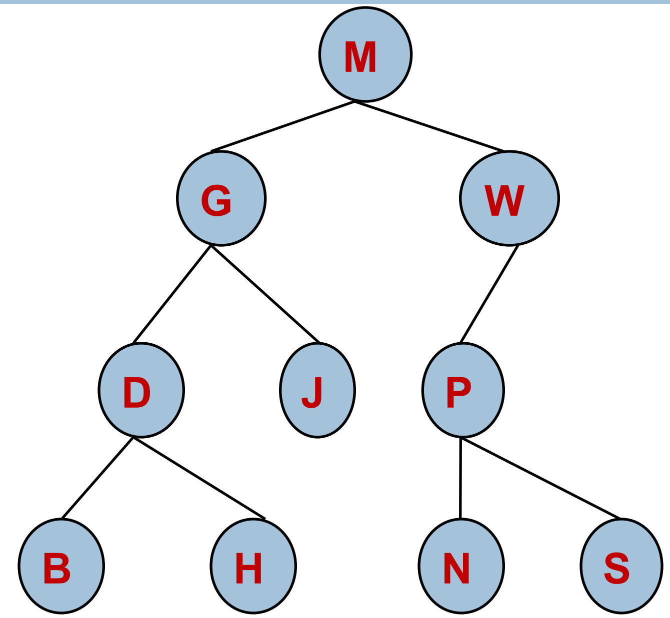
Degree of this tree

Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer **size**()
  - boolean **isEmpty**()
  - objectIterator **elements**()
  - positionIterator **positions**()
- Accessor methods:
  - position **root**()
  - position **parent**(p)
  - positionIterator **children**(p)

- Query methods:
  - boolean **isInternal**(p)
  - boolean **isExternal**(p)
  - boolean **isRoot**(p)
- Update methods:
  - **swapElements**(p, q)
  - object **replaceElement**(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Tree Terminology



How to construct a binary tree?

Class for Binary Tree Node

```
template <class T>
class TreeNode{
  T datum;
  TreeNode<T>* left, * right;

public:
  TreeNode(T x){datum=x; left = nullptr; right = nullptr;}
  TreeNode(T x, TreeNode<T>* lft, TreeNode<T>* rgt){
    datum = x; left = lft; right = rgt;
  }
};
```

Points to left subtree

Points to right subtree

more methods: getLeft, setLeft, etc.

Binary versus general tree

In a binary tree each node has exactly two pointers: to the left subtree and to the right subtree

- Of course one or both could be ***nullptr***

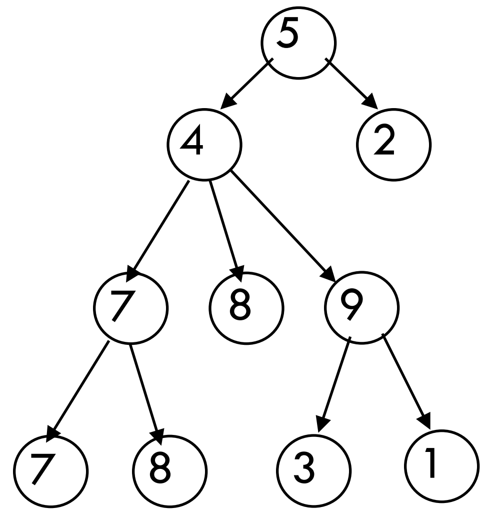
In a general tree, a node can have any number of child nodes

- Very useful in some situations …

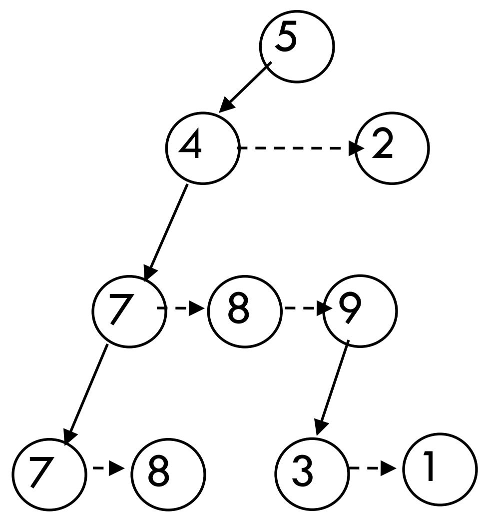
Class for General Tree nodes


```
template <class T>
class GTreeNode{
  T datum;
  GTreeNode<T>* left, * sibling;
    appropriate getters/setters
};
```

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



General tree



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 (ASTs)
- ASTs are easier to optimize, generate code from, etc. than textual representation
- A parser converts textual representations to AST

Example

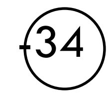
Expression grammar:
- E → integer
- E → (E + E)

In textual representation
- Parentheses show hierarchical structure

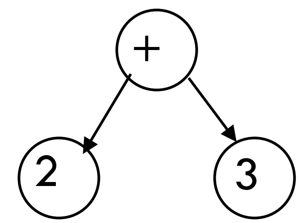
In tree representation
- Hierarchy is explicit in the structure of the tree

Text | **AST Representation**

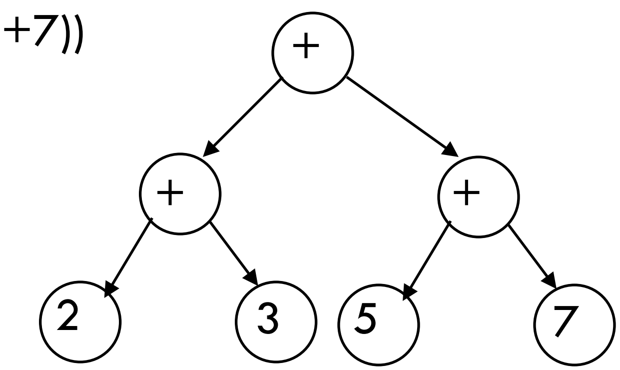
-34



(2 + 3)



((2+3) + (5+7))



Recursion on Trees

Recursive methods can be written to operate on trees in an obvious way

Base case

- **empty tree**
- **leaf node**

Recursive case

- solve problem on left and right subtrees
- put solutions together to get solution for full tree

Tree Traversal

- Two main methods:
  - Preorder
  - Postorder

- Recursive definition

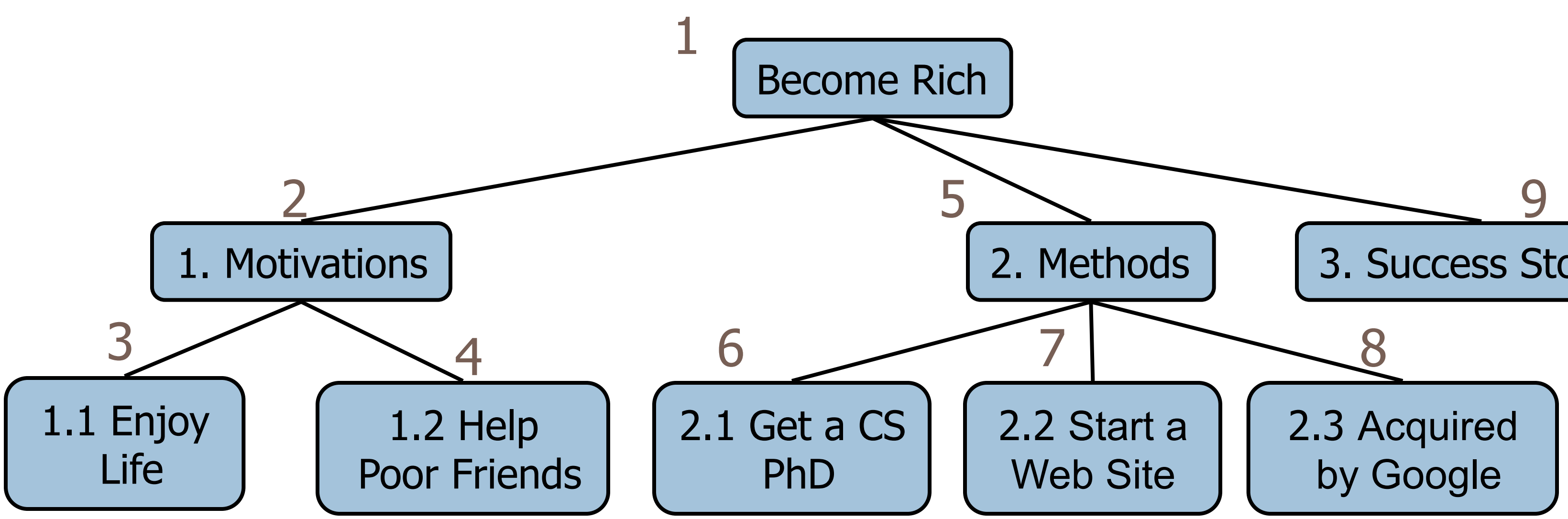
- Preorder:
  - visit the root
  - traverse in preorder the children (subtrees)

- Postorder
  - traverse in postorder the children (subtrees)
  - visit the root

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

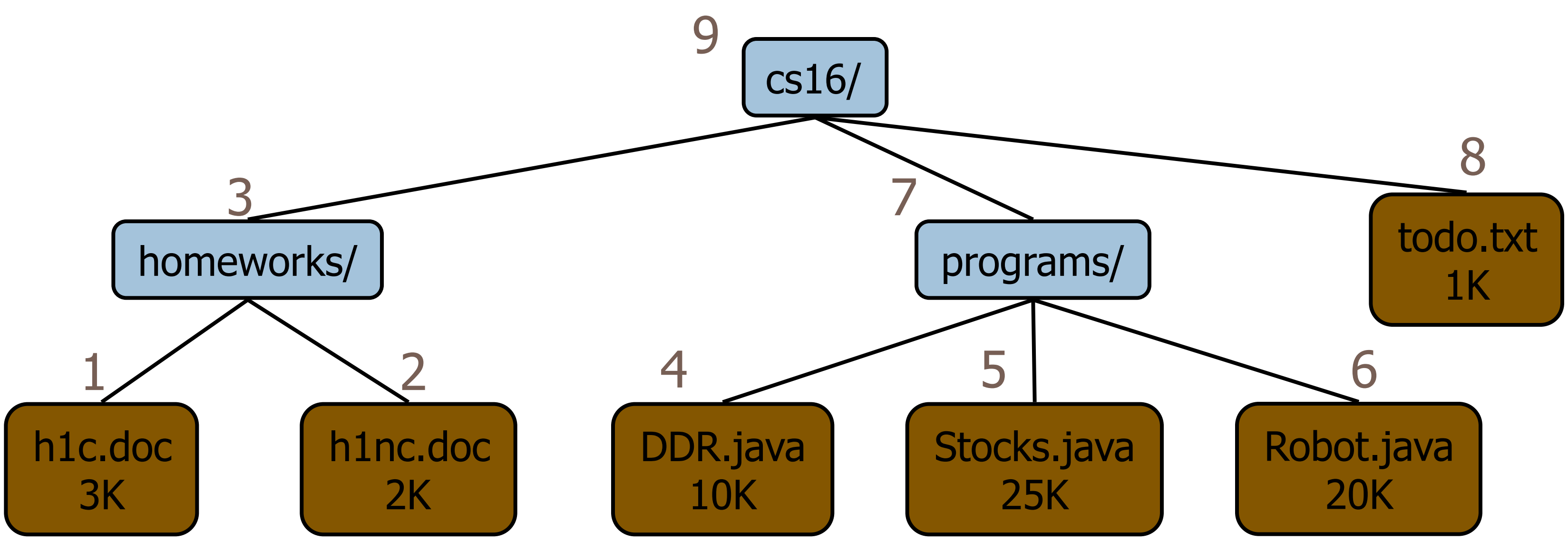
Algorithm ***preOrder***(***v***)
    visit(***v***)
    for each child ***w*** of ***v***
        preorder (***w***)



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

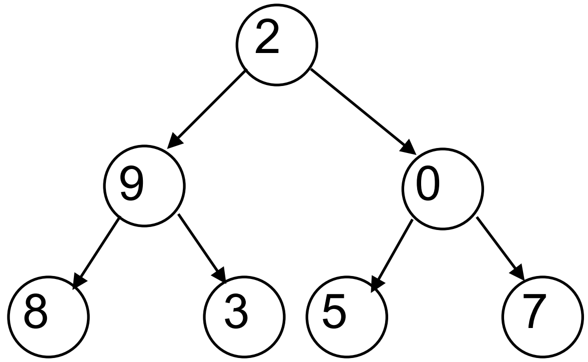
```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```



Searching in a Binary Tree


```
/** Return true iff x is the datum in a node of tree  t*/
template <class T>
bool treeSearch(T x, TreeNode<T>* t){
  if(t == nullptr) return false;
  if(t->getDatum() == x) return true;
  return treeSearch(x, t->getLeft()) || treeSearch(x, t->getRight());
}
```

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



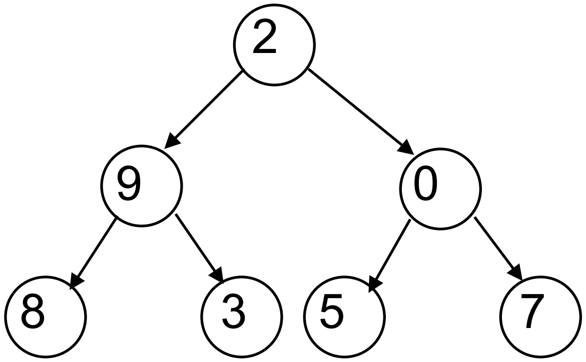
Searching in a Binary Tree


```
/** Return true iff x is the datum in a node of tree  t*/
template <class T>
bool treeSearch(T x, TreeNode<T>* t){
  if(t == nullptr) return false;
  if(t->getDatum() == x) return true;
  return treeSearch(x, t->getLeft()) || treeSearch(x, t->getRight());
}
```

Important point about t. We can think of it either as

(1) One node of the tree OR

(2) The subtree that is **rooted** at t



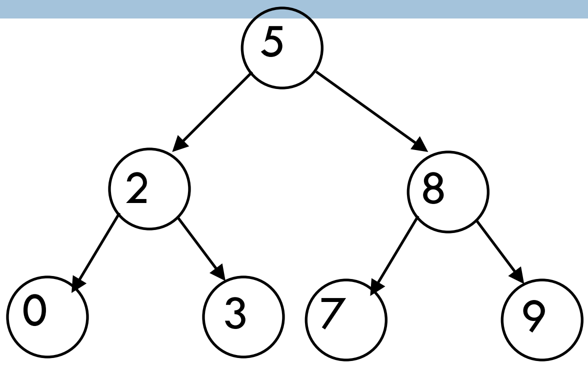
Binary Search Tree (BST)

If the tree data are ***ordered***: in **every** subtree,

All ***left*** descendents of node come *before* node

All ***right*** descendents of node come *after* node

Search is MUCH faster

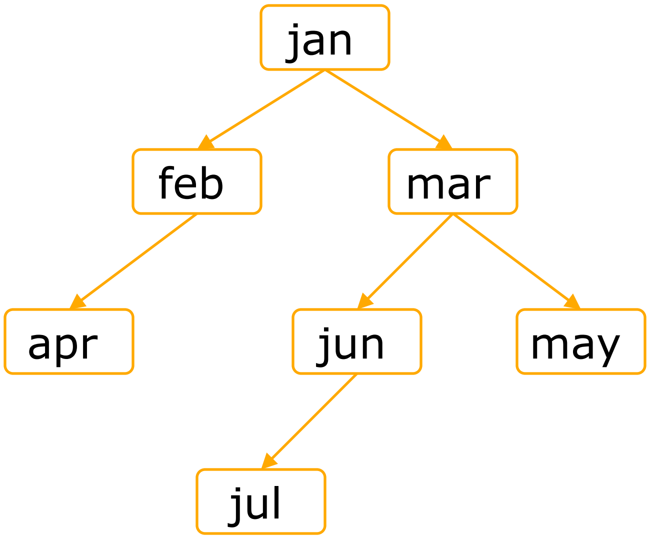



```
/** Return true iff x if the datum in a node of tree t.
   Precondition: node is a BST */
template <class T>
bool treeBSearch(T x, TreeNode<T>* t){
  if(t == nullptr) return false;
  if(t->getDatum() == x) return true;
  if(t->getDatum() > x)
    return treeBSearch(x, t->getLeft());
  else
  return treeBSearch(x, t->getRight());
}
```

Remember the binary search for array?
Are they similar?

Building a BST

- To **insert** a new item
  - Pretend to **look for** the item
  - Put the new node in the place where you **fall off** the tree
- This can be done using either recursion or iteration
- Example
  - Tree uses ***alphabetical*** *order*
  - Months appear for insertion in *calendar order*



What Can Go Wrong?

- A BST makes searches very fast, *unless*…
  - Nodes are inserted in alphabetical order
  - In this case, we're basically building a linked list (with some extra wasted space for the **`left`** fields that aren't being used)
- BST works great if data arrives in **random order**