ECE36800 Data structures

Trees

Chapter 5 (pp. 216-240)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Slides Courtesy: Prof. Cheng-Kok Koh

Overview

- Trees
  - Definition
  - C implementations
  - Traversals
  - Examples
- Threaded binary trees
- Tries
- Mathematical expressions

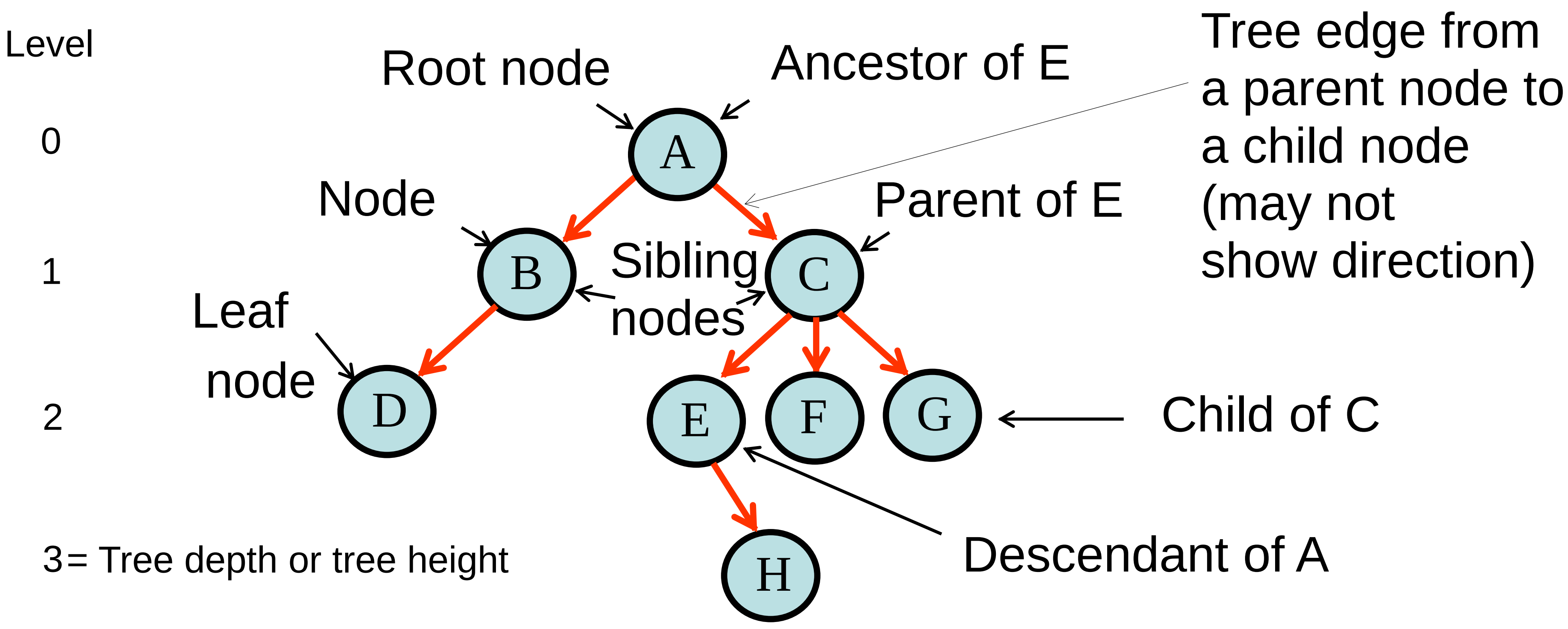
- Reference pages pp. 216-240

Trees

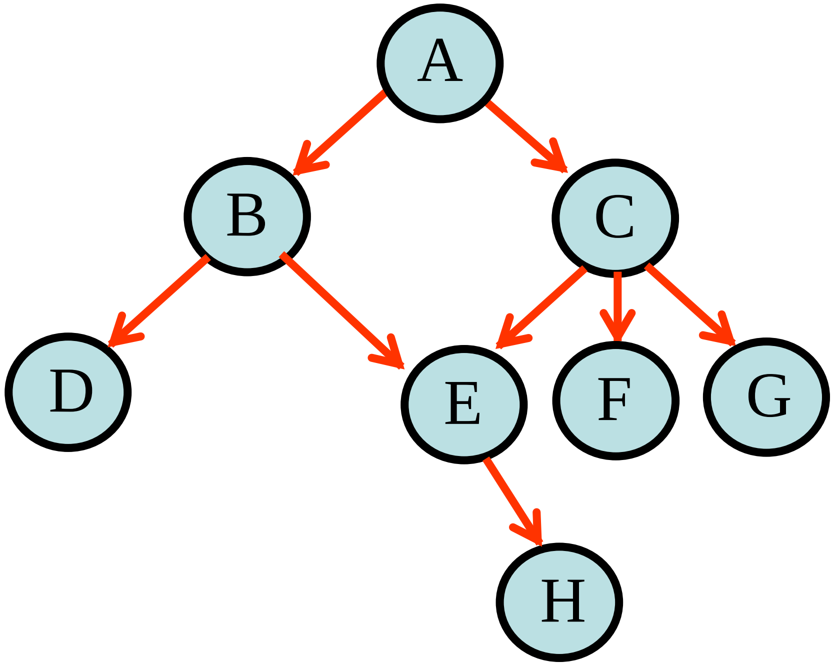
- Recursive definition:
  - A singleton node is a tree
  - A ordered collection of items arranged in a structure of a root node and one or more disjoint trees that are children of the root node
- A tree is a graph that has no cycles and has only a single path between any two nodes
- Trees are useful for reducing complexity (often logarithmic) and finding solutions (computation trees)
  - Searching and sorting of elements
  - Representation of data (compression, mathematical expressions, disjoint sets)
  - ...

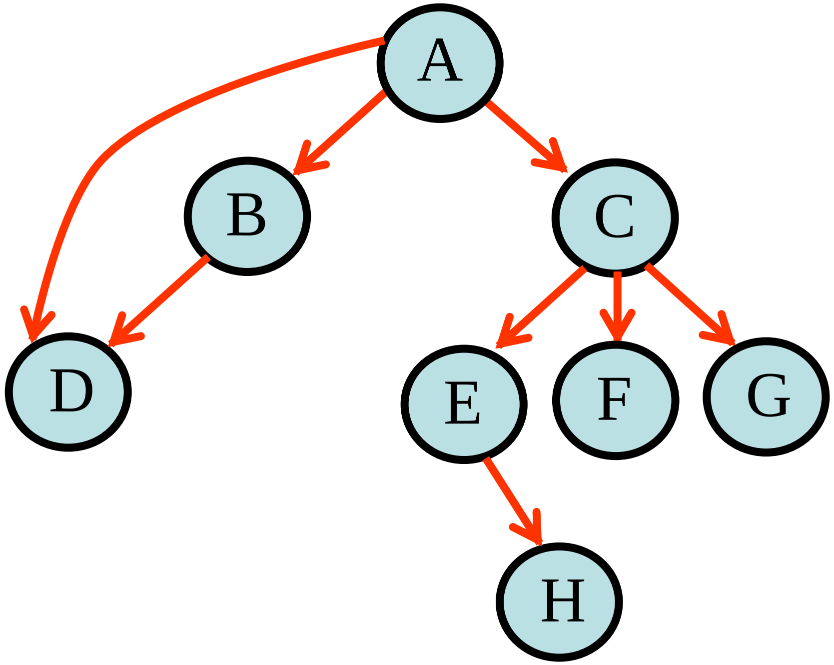
Trees: vocabulary

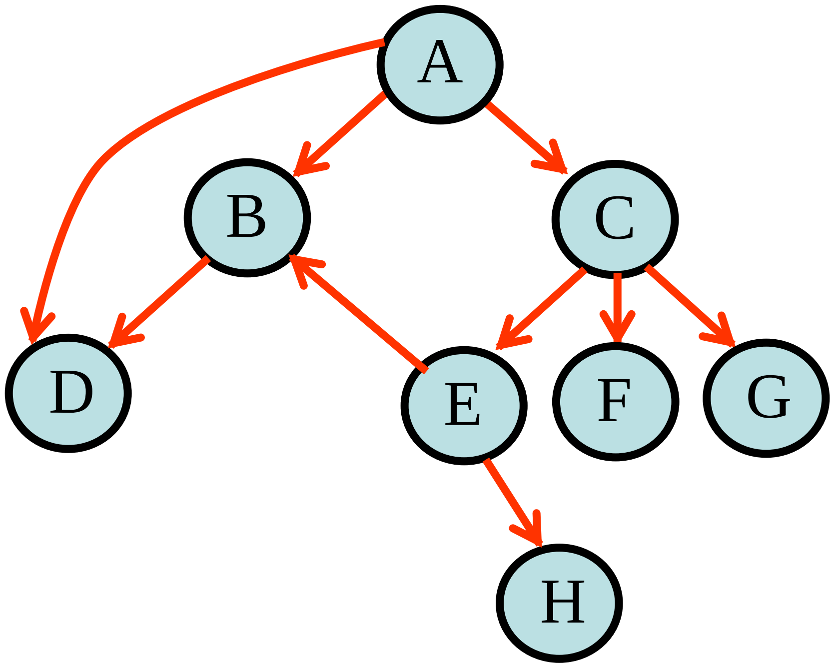
- A binary tree is a tree with at most two child nodes (left and right child nodes) per node
- A n-ary tree is a tree with up to n child nodes per node



Trees: Not!



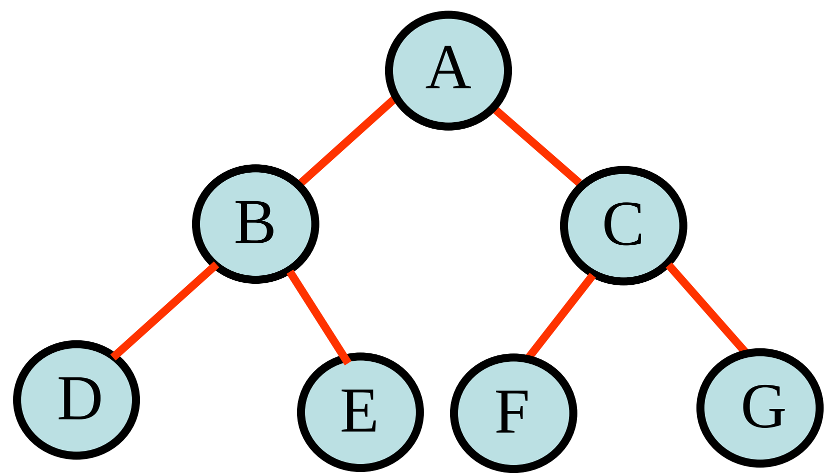




If we disregard the directions on all tree edges, there should only be a single path between any pair of nodes

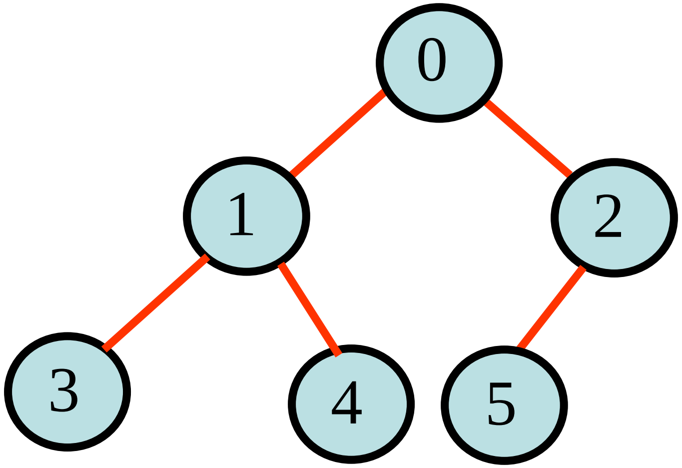
Complete binary tree

- All levels up to the tree depth are filled completely
  - All non-leaf nodes (internal nodes) have two child nodes
- At each level L, the number of nodes = 2^L
- Tree height (depth) h, the number of leaf nodes = 2^h
- Total number of nodes n = $2^{(h+1)} – 1$
- Tree height h = O(log n)



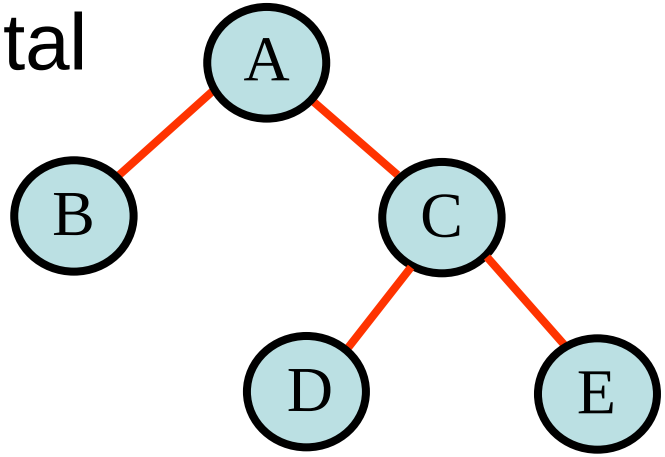
Almost complete binary tree

- All levels except the tree depth level are filled completely
- At the tree depth level, nodes are filled from left to right
- There is a numbering scheme such that a node number i has left and right child nodes numbered 2i+1 and 2i+2
- Useful for heap (priority queue)
- For a tree height h
  - At least 2^h nodes
  - At most $2^{(h+1)} – 2$ nodes
  - $h = O(\log n)$



Strictly binary tree

- A node can have 0 or 2 child nodes
  - A node with 0 child nodes is a leaf node
  - A node with 2 child noes is a non-leaf node (internal node)
- When there are n leaf nodes, the total number of nodes is 2n – 1
- Examples:
  - Huffman coding tree
  - Mathematical expression involving operators of two operands



Binary tree in C

- Assume that we store int as data in a tree node

```
typedef struct _Tnode
{
  int data;
  struct _Tnode *left;
  struct _Tnode *right;
} Tnode;
```

- More generic implementation, store address of a generic type in struct

```
typedef struct _Tnode
{
  void *data;
  struct _Tnode *left;
  struct _Tnode *right;
} Tnode;
```

```
int Is_empty (Tnode *bt_ptr)
{
  return (bt_ptr == NULL);
}

Tnode *Create (int data,
  Tnode *left_ptr,
  Tnode *right_ptr)
{
  Tnode *new_node =
    malloc (sizeof (*new_node));
  if (new_node != NULL) {
    new_node->data = data;
    new_node->left = left_ptr;
    new_node->right = right_ptr;
  }
  return new_node;
}

int Data (Tnode *bt_ptr) // bt_ptr not NULL
{
  return (bt_ptr->data);
}
```

```
Tnode *Left_Child (Tnode *bt_ptr) // bt_ptr not NULL
{
  return bt_ptr->left;
}

Tnode *Right_Child (Tnode *bt_ptr) // bt_ptr not NULL
{
  return bt_ptr->right;
}

void BT_Delete (Tnode *bt_ptr)
{
  if (bt_ptr != NULL) {
    BT_Delete (bt_ptr->left);
    BT_Delete (bt_ptr->right);
    free(bt_ptr);
  }
}
```

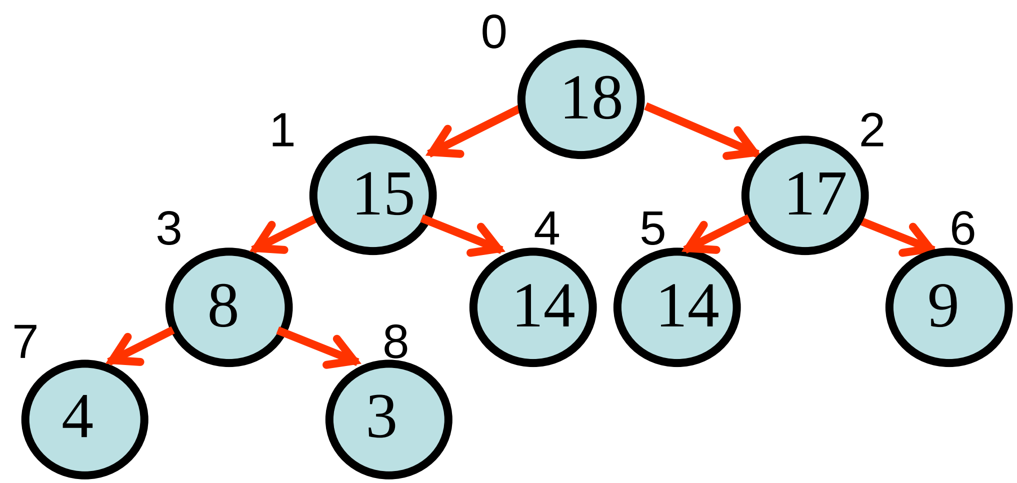
Array implementation

- Allocate an array of Tnode's and build a tree using entries of the array
- Cannot simply grow the array to build a bigger tree because the addresses may change
- Can also define a new struct that uses int to keep track of allocate an array of Tnode's and use the left index and right index

```
typedef struct _Tnode
{
  int data;
  int left;
  int right;
} Tnode;
```

Implicit representation for almost complete binary tree

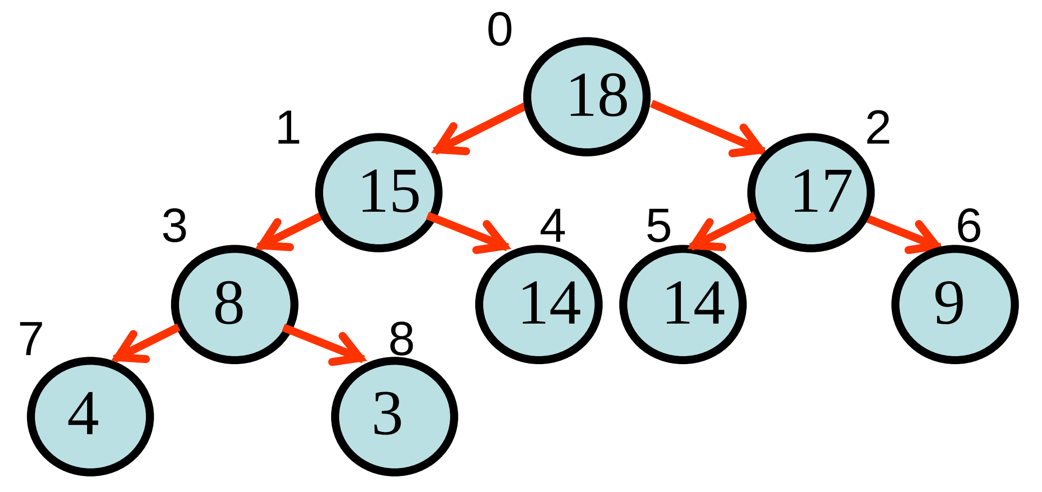
- Implicitly find the parent-child relationship using indices
- Array is { 18, 15, 17, 8, 14, 14, 9, 4, 3 }
- Root node has index 0, and we increment the index from top to bottom, left to right



- For a tree with n nodes, node of index i, $0 \leq i < n$,
  - Parent of i has index (i – 1)/2, if (i – 1)/2 $\geq$ 0; otherwise, no parent
  - Left of i has index (2*i + 1) if (2*i + 1) < n; otherwise, no left
  - Right of i has index (2*i + 2) if (2*i + 2) < n; otherwise, no right

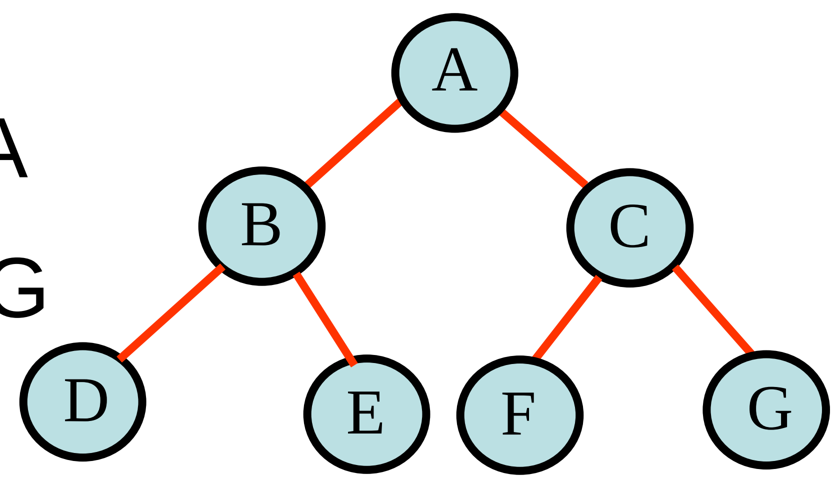
Max heap (descending priority queue)

- In a max heap, the values of the two child nodes are equal to or less than the value of the parent
- Example: array { 18, 15, 17, 8, 14, 14, 9, 4, 3 }
- The root always contains the maximum value (accessible in O(1))



Binary tree traversal

- Depth-first (you have already done that in the lecture on recursion, it is important to note that you don't always have to go to the left first)
  - Preorder: A, B, D, E, C, F, G
  - Inorder: D, B, E, A, F, C, G
  - Postorder: D, E, B, F, G, C, A
- Breath-first: A, B, C, D, E, F, G



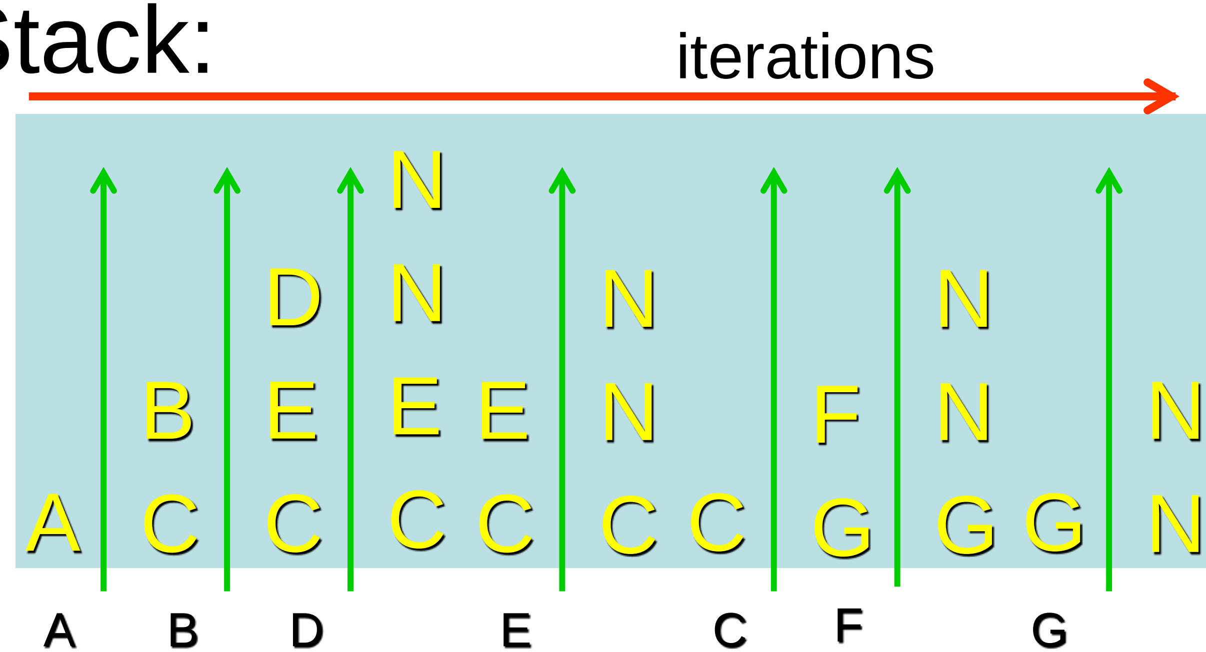
```
void Preorder (Tnode *node)
{
  if (node != NULL)
  {
    // do something about node
    // e.g. fprintf(“%d\n”, node->data);
    Preorder(node->left);
    Preorder(node->right);
  }
}

void Inorder (Tnode *node)
{
  if (node != NULL)
  {
    Inorder(node->left);
    // do something about node
    // e.g. fprintf(“%d\n”, node->data);
    Inorder(node->right);
  }
}

void Postorder (Tnode *node)
{
  if (node != NULL)
  {
    Postorder(node->left);
    Postorder(node->right);
    // do something about node
    // e.g. fprintf(“%d\n”, node->data);
  }
}
```

Iterative preorder traversal using stack

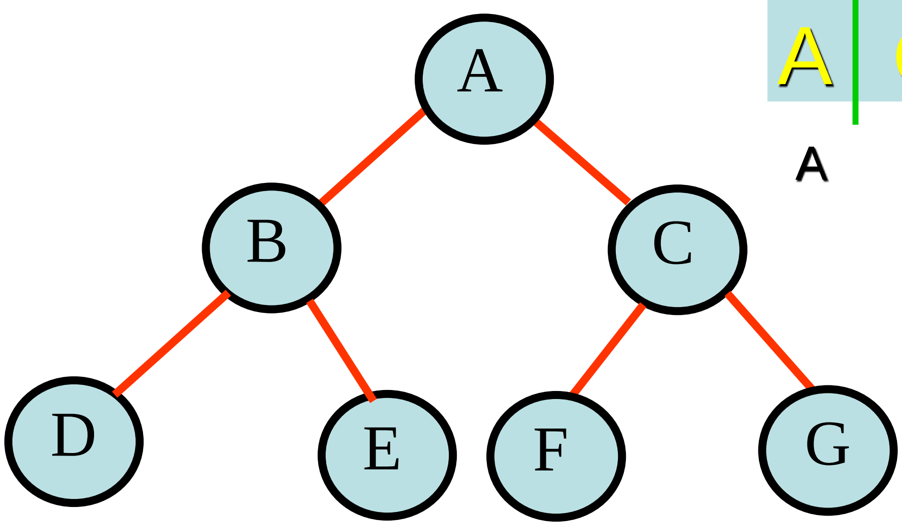
```
// Assume a stack S

Preorder(BT_root):
  Push(S, BT_root)
  while (!Is_empty(S))
  {
    node ← Pop(S);
    if (node != NULL)
    {
      // do something with node
      // e.g., print node->data
      Push(S, node->right);
      Push(S, node->left);
    }
  }
```

Iterative preorder traversal using stack

Stack:



Tree:



Iterative inorder traversal using stack

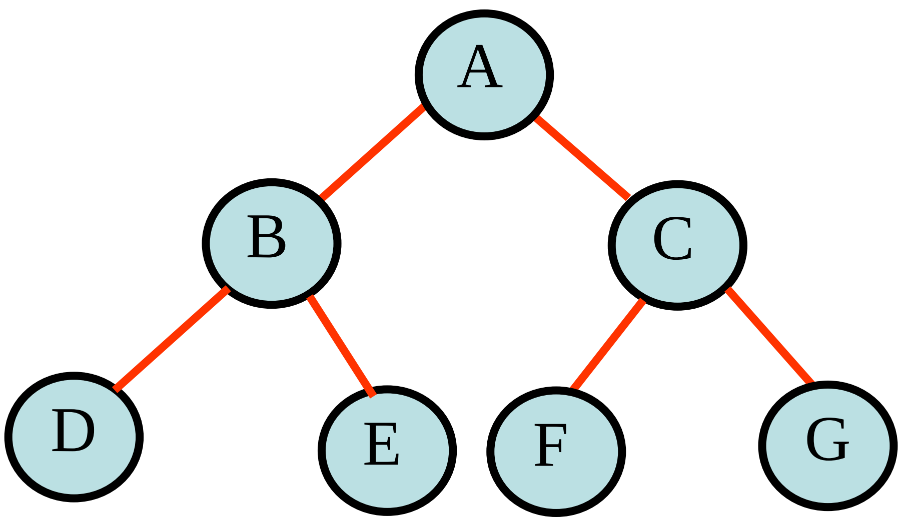
```
// Assume stack S

Inorder(BT_root):
  p ← BT_root;
  do { // travel down left branches
    while (p != NULL) {
      Push(S, p);
      p ← p->left;
    }
    if (!Is_empty(S)) {
      // the left subtree is empty
      p ← Pop(S);
      // do something with p
      // e.g. print p->data
      p ← p->right;  // traverse right subtree
    }
  } while (!Is_empty(S) || p != NULL)
```

Breadth-first traversal using queue

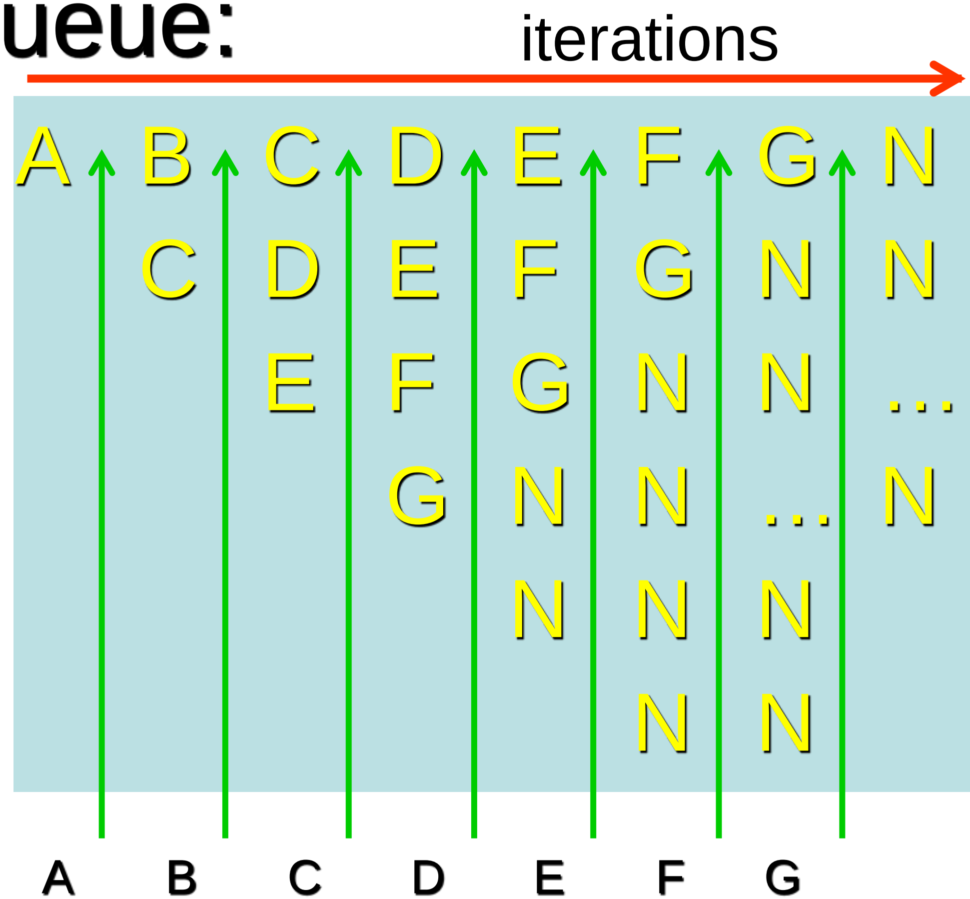
```
// Assume a queue Q

Breadth_first(BT_root):
  Enqueue(Q, BT_root)
  while (!Is_empty(Q))
  {
    node = Dequeue(Q);
    if (node != NULL)
    {
      // do something with node
      // e.g., print node->data
      Enqueue(Q, node->left);
      Enqueue(Q, node->right);
    }
  }
```

Breadth-first traversal using queue

Tree:



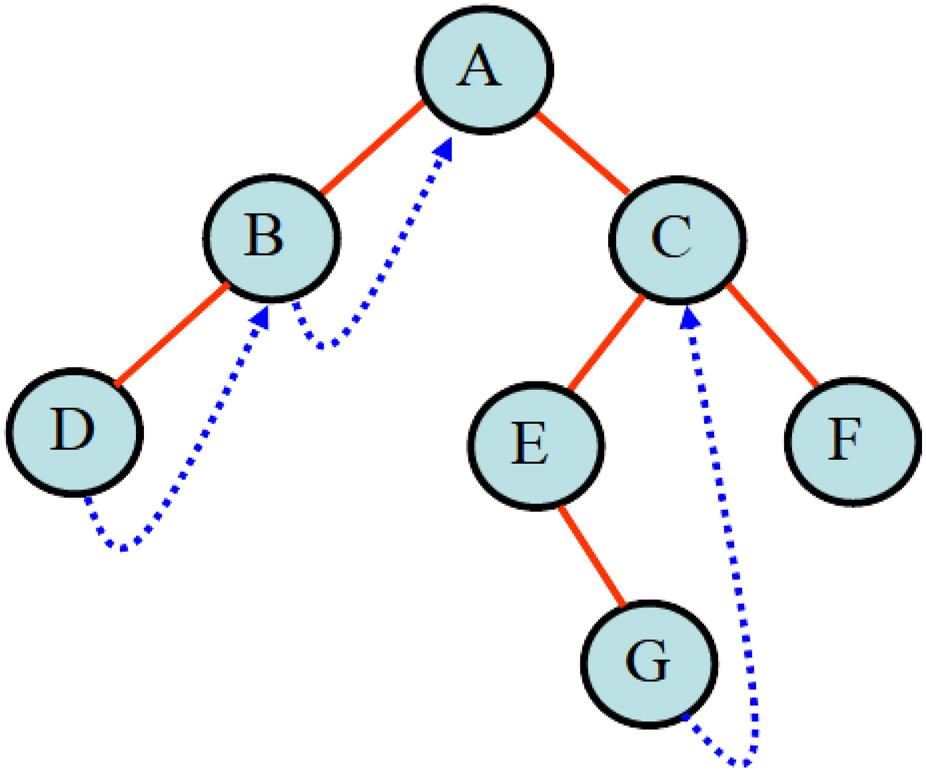
Queue:



Threaded binary trees

- A (right in-)threaded binary tree is a binary tree in which every node that does not have a right child has a link to its inorder successor
- The link can be stored in the right pointer, and a binary variable rthread must also be stored in order to tell it is a right child (rthread == 0) or an inorder successor (rthread == 1)
- Useful in order to avoid:
  - Recursion (can search tree only using while loops)
  - Use of stack

Threaded binary tree



Inorder:

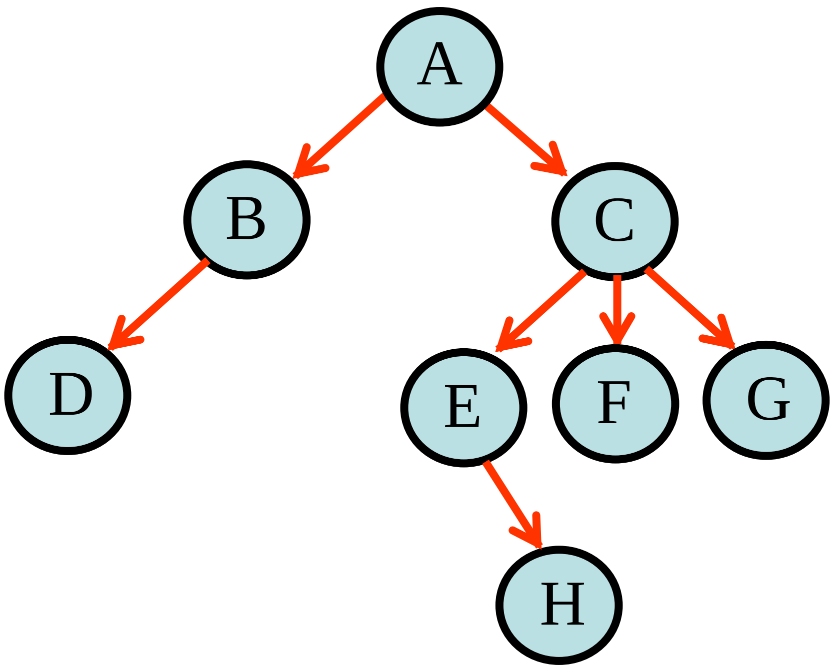
D, B, A, E, G, C, F

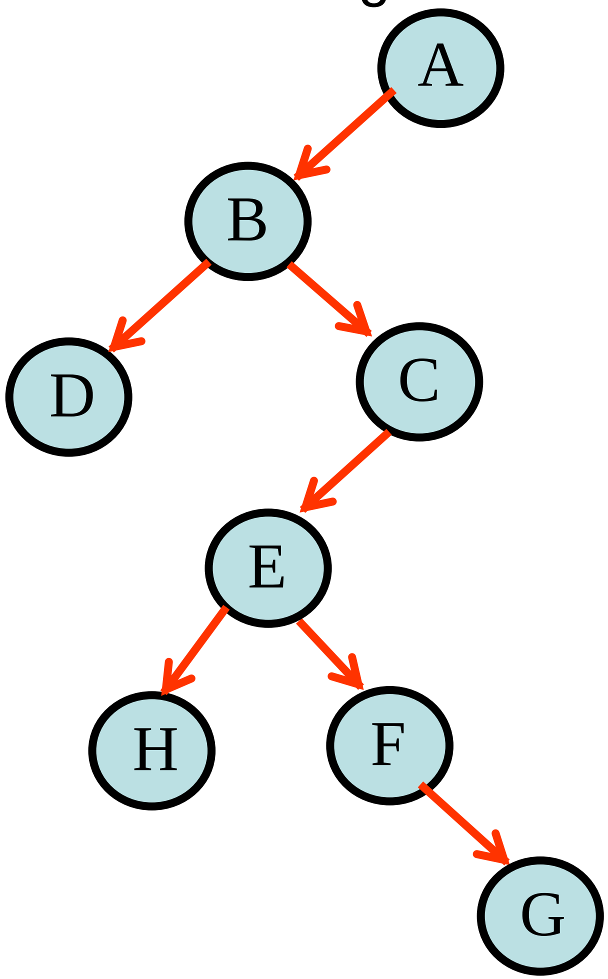
Threaded tree traversal (inorder)

```
Threaded_tree_inorder(TBT_root):
  p ← TBT_root;
  do { // travel down left branches
    q ← NULL;
    while (p != NULL) {
      q ← p;
      p ← p->left;
    }
    if (q != NULL) {
      // do something with q
      p ← q->right;
      while (q->rthread && p != NULL) {
        // do something with p
        q ← p;
        p ← p->right;
      }
    }
  } while (q != NULL)
```

N-ary trees

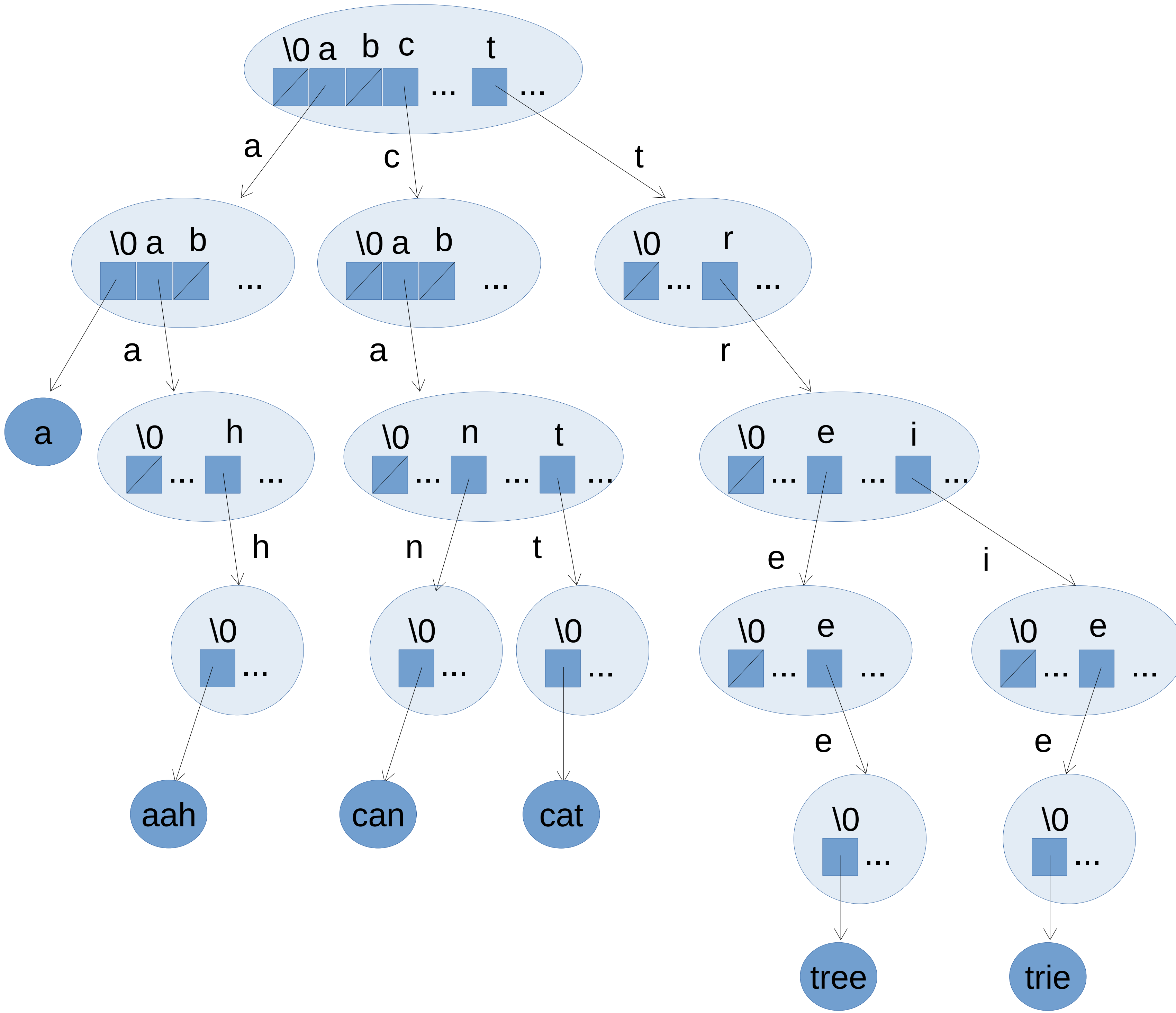
- Can always represent an n-ary tree with a binary tree
  - The left child of a node in the binary tree is the first child in the n-ary tree
  - The right child of a node in the binary tree is the sibling node to its right in the n-ary tree





Trie

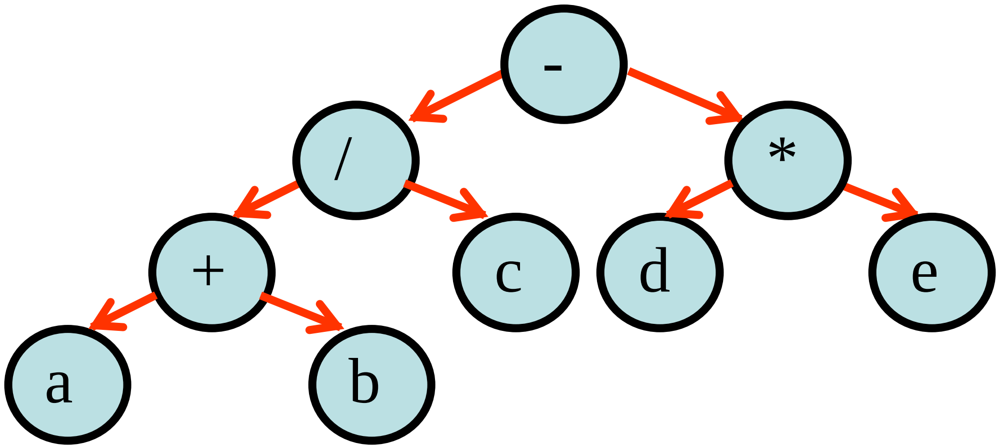
- An information reTrieval data structure
- Can be used to store a dictionary of words (strings)
- Each node has up to 27 child nodes stored in an array of addresses, each indexed by a character ('\0', 'a', 'b', …, 'z') (assume only lower case letters)
- Each word (string) is represented by a path from the root node
  - The first character of the string is represented by a level-1 node, whose address is stored in the array of the root node, and indexed by the character
  - The second character is represented by a level-2 node, whose address is stored in the array of the level-1 node of the first character, and indexed by the character
  - …
  - When we reach the end of the word (or the '\0' character of the string), the data associated with the string can be stored in a node, whose address is indexed by the '\0' character
- The root node corresponds to an empty string, i.e., empty prefix
- Words that share a common prefix will have a common path from the root to the node that corresponds to the prefix


\0 a b c t
... ...
a
c
t
\0 a b
...
\0 a b
...
\0 r
... ...
a
a
r
a
\0 h
... ...
\0 n t
... ...
\0 e i
... ...
h
n
t
e
i
\0
...
\0
...
\0
...
\0 e
... ...
\0 e
... ...
e
e
aah
can
cat
\0
...
\0
...
tree
trie

Mathematical expressions

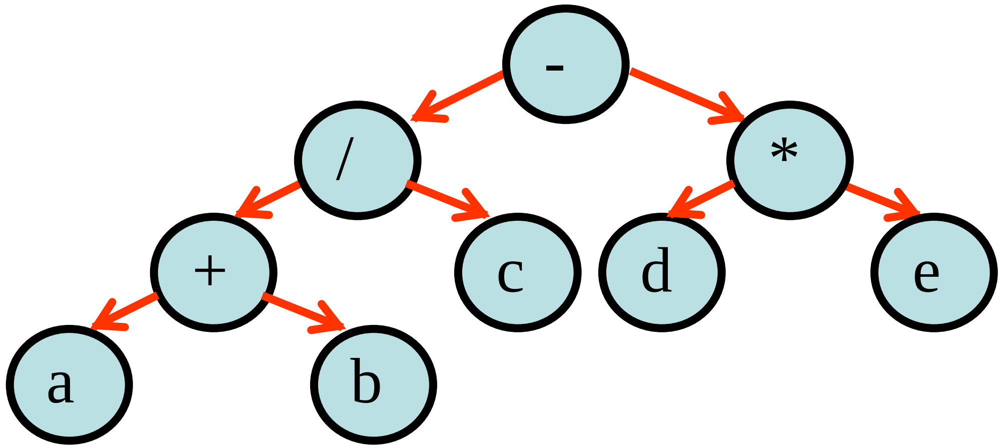
- Assume mathematical operations that take in two operands (for simplicity, can be more general)
- In C program, we write it in the convention that is equivalent to an inorder traversal of a strictly binary tree (infix notation)
- Compilation of a program to a stack-machine code involves parsing such an expression and convert it to a postfix notation or reverse Polish notation (equivalent to postorder traversal of the strictly binary tree)
- Given a postfix notation
  - Evaluate the mathematical expression
  - Reconstruct a strictly binary tree for the expression

Prefix, infix, and postfix



Prefix	Infix	Postfix
-(/(+(a b) c) *(d e))	((a + b) / c) - (d * e)	a b + c / d e * -
or		
-(/(+(a, b), c), *(d, e))		

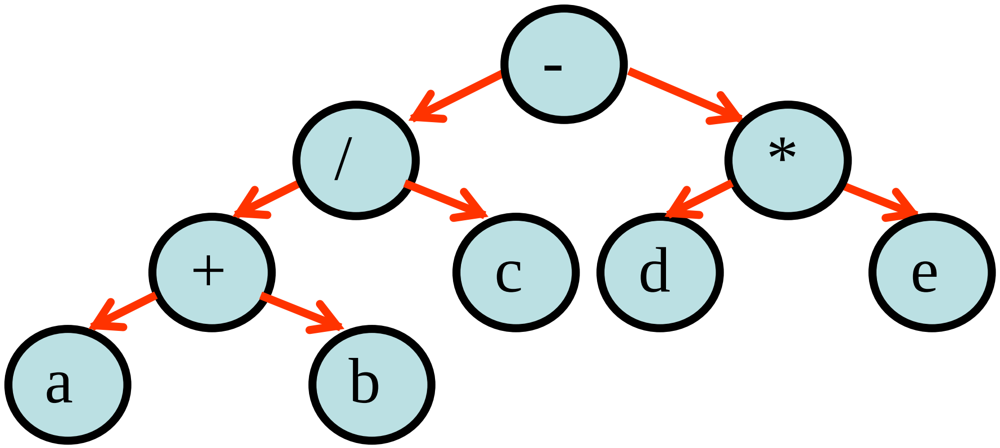
Evaluating postfix expression



a b + c / d e * -

```
For each token in expression from left to right:
  switch (type(token)):
  case operand: push(S, token);
                break;
  case operator: operand2 = pop(S);
                 operand1 = pop(S);
                 result = operand1 token operand2;
                 push(S, result);
                 Break;
return pop(S);
```

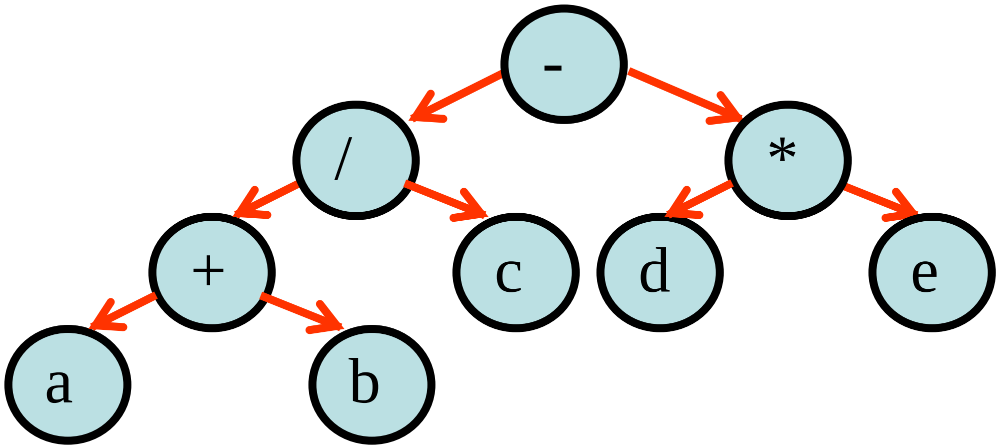
Evaluating postfix expression



a b + c / d e * -

a	b	+	c	/	d	e	*	-
						e		
	b		c		d	d	d*e	
a	a	a+b	a+b	(a+b)/c	(a+b)/c	(a+b)/c	(a+b)/c	((a+b)/c)-(d*e)

Rebuilding strictly binary tree



a b + c / d e * -

```
For each token in expression from left to right:
  switch (type(token)):
  case operand: push(S, create(token, NULL, NULL);
                break;
  case operator: right = pop(S);
                 left = pop(S);
                 push(S, create(token, left, right));
                 Break;
return pop(S);
```

Rebuilding strictly binary tree