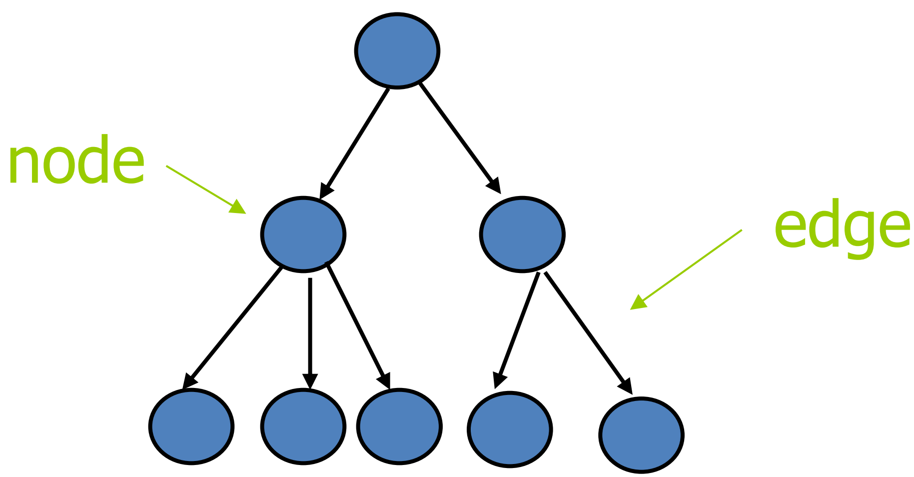
Data Structures and Algorithms

Lecture 7: Trees

Outline of Today's Lecture

- Motivation
- Binary trees
- Property of binary tree
- Binary Tree ADT
- Tree Traversals
- Non-Binary Trees
- Applications



Motivation

- Suppose to design a *software* for bank HSBC to support its transactions, e.g.,
  - Open a bank account for a new user
  - Deposit some money for a user
  - Withdraw some money for a user

Motivation (cont. 1/4)

- The bank records the profile for each user
  - User name
  - User ID number
  - Home address
  - Balance
  - Contact number
  - *Bank Account number*
  - ….
- The bank account numbers are used to *uniquely* distinguish different users.

Motivation (cont. 2/4)

- For deposit and withdrawal transactions, the software should ***quickly*** response upon giving the account number
- The software also needs to ***quickly*** open an account for a new user
- What is the type of data structure better?
  - such that both searching and insertion are quickly performed by the system

Motivation (cont. 3/4)

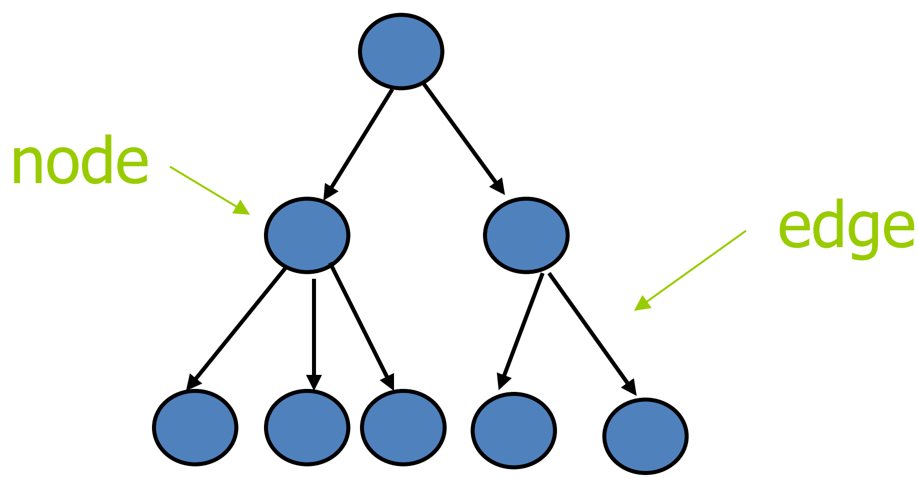
- Suppose using *array-based list*
  - Searching time is $\Theta(\log n)$, fast
  - But the insertion is slow, $\Theta(n)$ on average
- How is it *linked list*?
  - Fast insertion by inserting at the beginning of the list, i.e., $\Theta(1)$ time
  - Slow searching, $\Theta(n)$ on average
- None supports both fast searching and insertion operations!

Motivation (cont. 4/4)

- In this lecture, we introduce a new data structure, called **tree** and practically ***binary tree***, such that
  - Suppose the tree's height is log n
  - Searching: Θ(log n) on average
  - Insertion: Θ(log n) on average
  - Removal: Θ(log n) on average

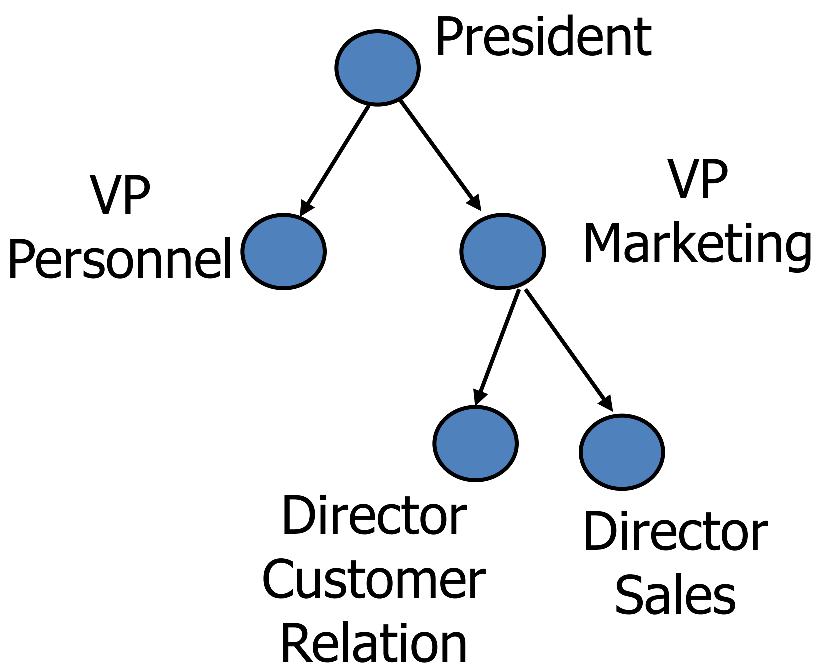
What is a tree?

- Trees are structures used to represent hierarchical relationship
  - Each tree consists of nodes and edges
  - Each node represents an object
  - Each edge represents the relationship between two nodes.

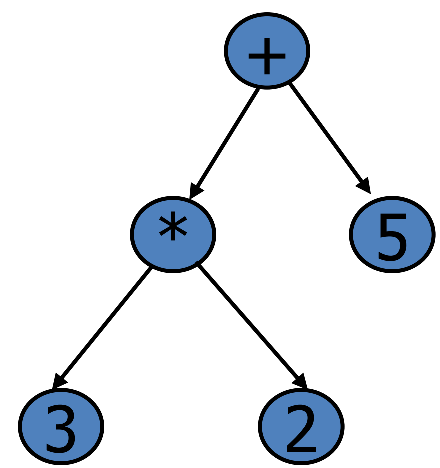


Some applications of Trees

Organization Chart

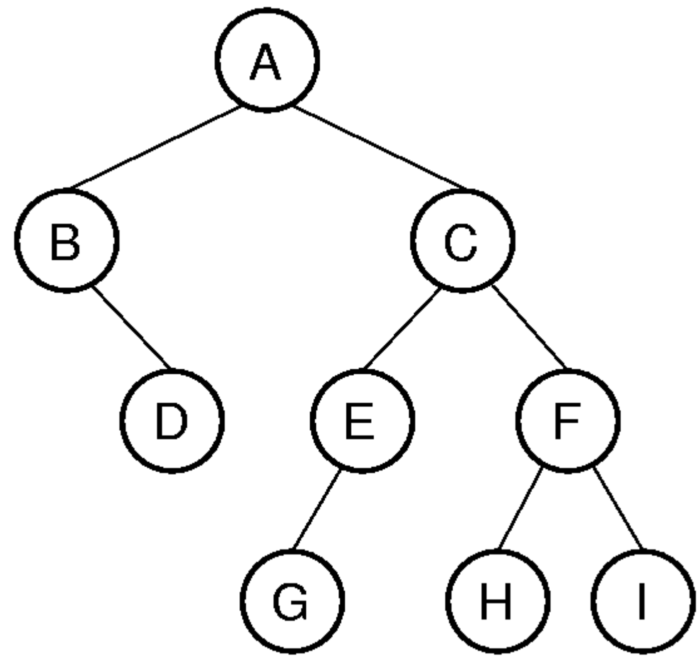


Expression Tree



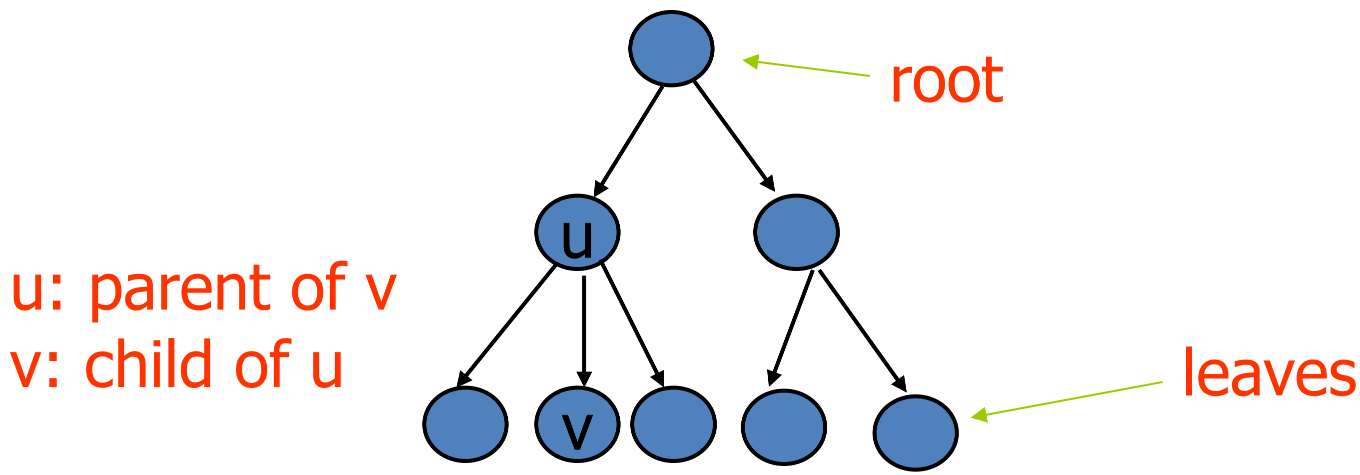
Terminology in a Tree

- Node, edge
- Children, parent
- Ancestor, descendant
- Leaf node, internal node
- Subtree
- Path
- Depth, level
- Tree height



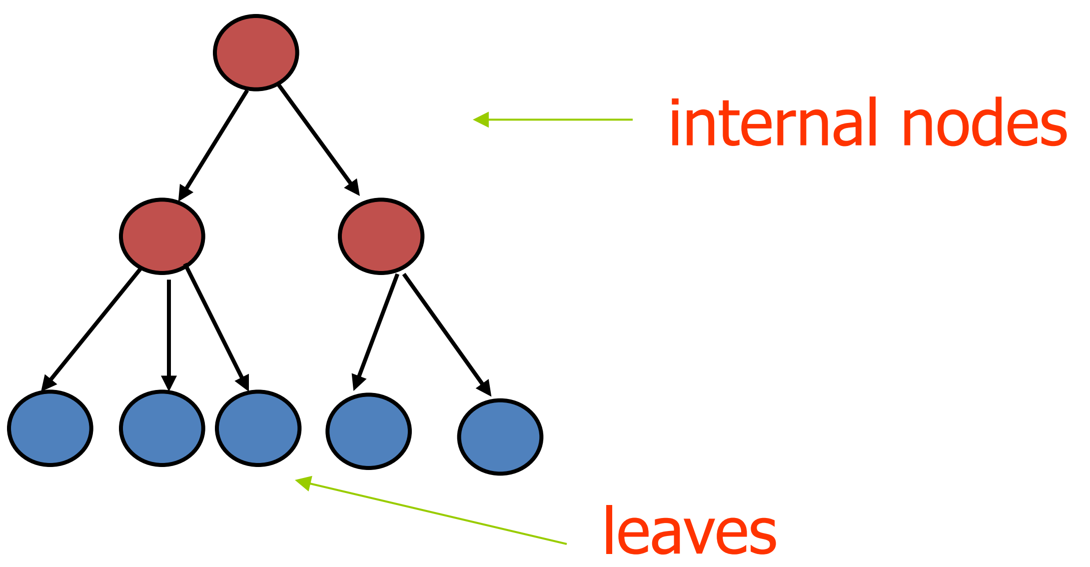
Terminology I

- For any two nodes u and v, if there is an edge pointing from u to v, u is called the parent of v while v is called the child of u. Such edge is denoted as (u, v).
- In a tree, there is exactly one node without parent, which is called the root. The nodes without children are called leaves.



Terminology II

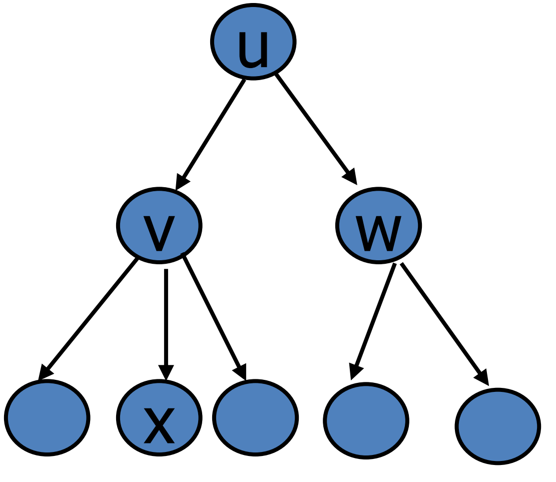
- In a tree, the nodes without children are called leaves. Otherwise, they are called internal nodes.



Terminology III

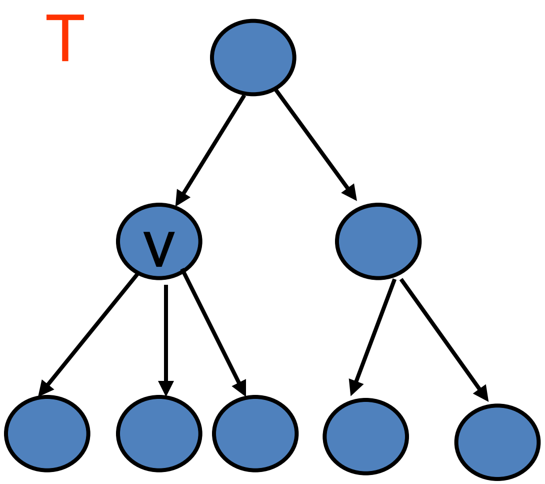
- If two nodes have the same parent, they are siblings.
- A node u is an ancestor of v if u is parent of v or parent of parent of v or …
- A node v is a descendent of u if v is child of v or child of child of v or …

v and w are siblings
u and v are ancestors of x
v and x are descendents of u

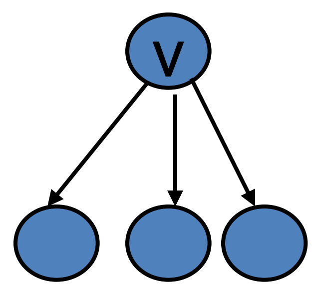


Terminology IV

- A subtree is any node together with all its descendants.

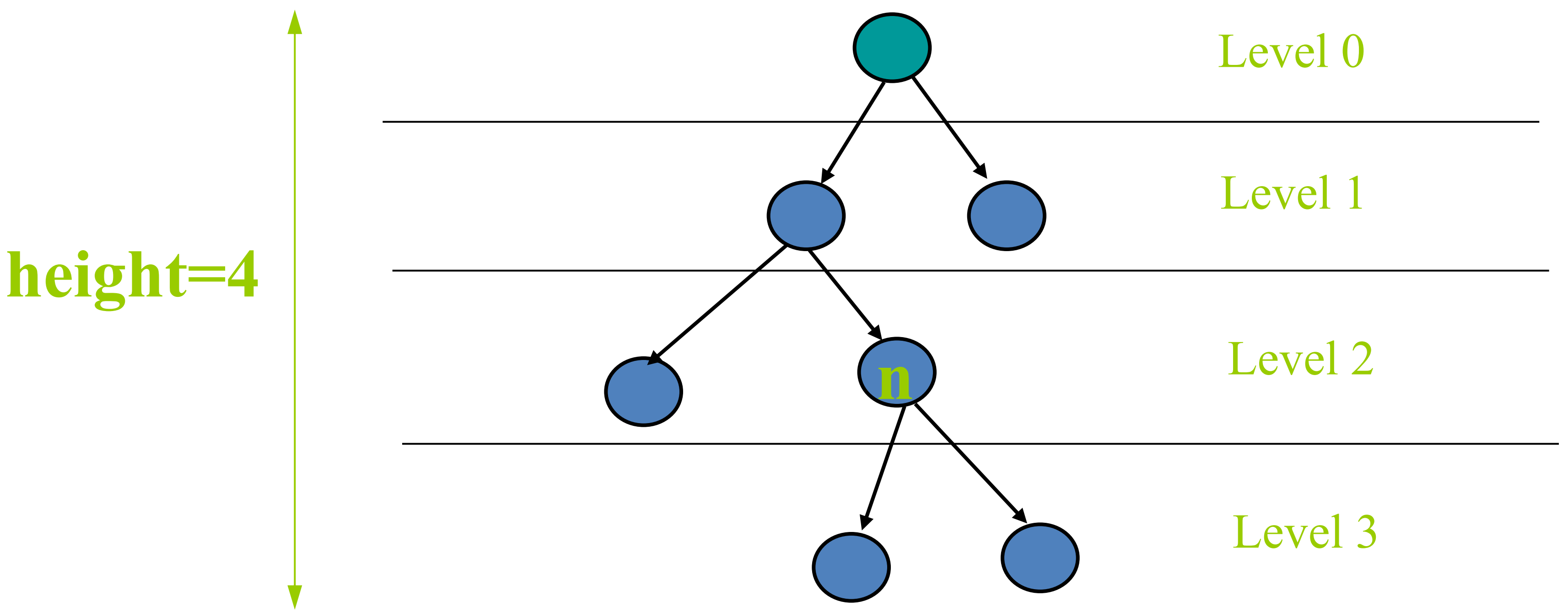


A subtree of T



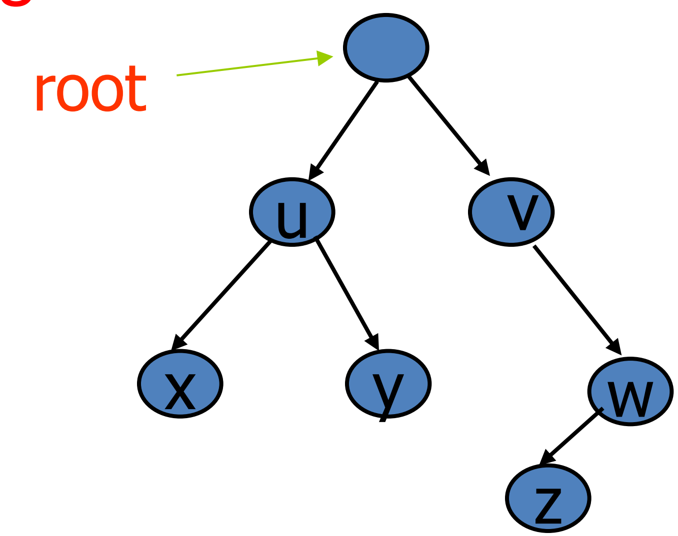
Terminology V

- Level of a node n: number of nodes on the path from root to node n
- Height of a tree: maximum level among all of its node



Binary Tree (BTree, or simple BT)

- Binary Tree (BT): Tree in which every node has at most TWO children
  - A **root**, and **left / right subtrees**
- Left child of u: the child on the left of u
- Right child of u: the child on the right of u

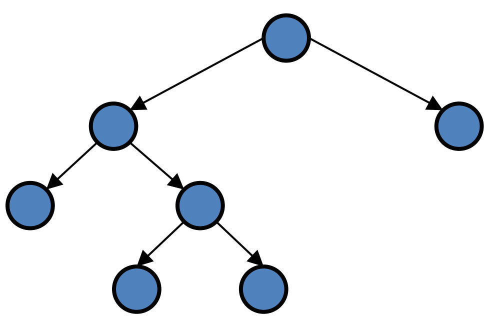


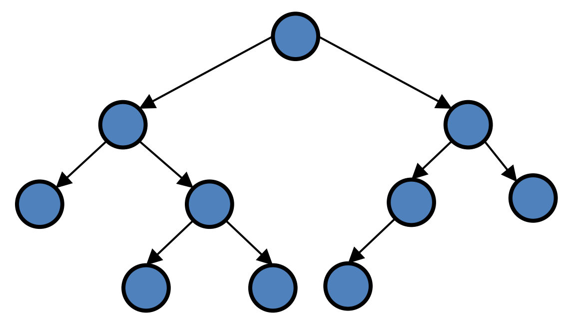
x: left child of u
y: right child of u
w: right child of v
z: left child of w

Full, Complete, and Perfect BT

Suppose T is empty, full or complete?

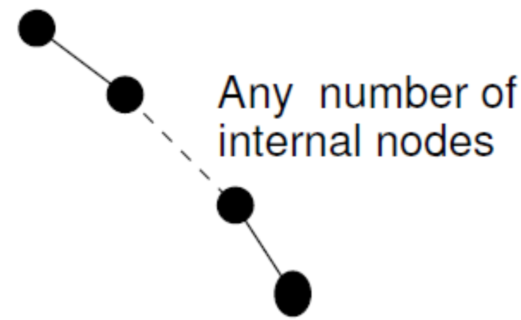
- **Full BT**: Each note in a full binary tree is either (1) an internal node with exactly two non-empty children or (2) a leaf.
- **Complete BT**: In the complete binary tree of height h >0, all levels except possibly level h-1 are completely full, and all nodes in the last level are as far left as possible.
- **Perfect BT** = Full + Complete





Minimum and Maximum Fraction

- How many leaf nodes are in a binary tree with *n* internal nodes



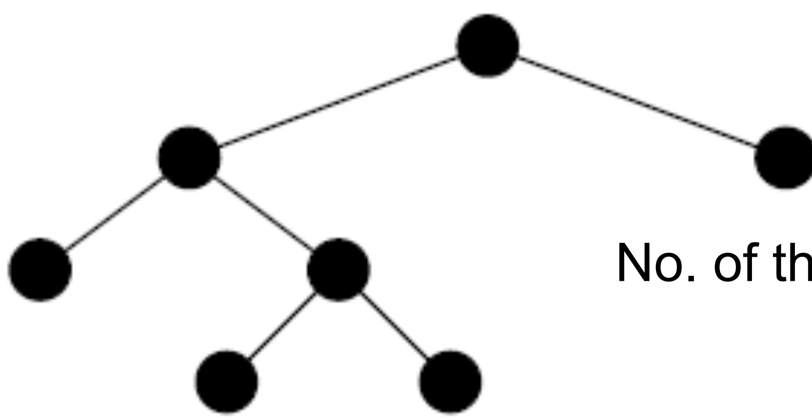
- The minimum number is one.
  - When all nodes are arranged in a chain
- What is the maximum number?
  - This upper bound occurs when each internal node has exactly two children, i.e., the tree is full.

Full Binary Tree Theorem

Theorem: The number of leaves in a non-empty full binary tree is **one more than** the number of internal nodes.

Proof. By mathematical induction.

See textbook for details.



No. of the internal nodes is 3.

Full Binary Tree Corollary

Theorem: The number of *empty subtrees* in a non-empty binary tree is *one more than* the number of nodes in the tree.

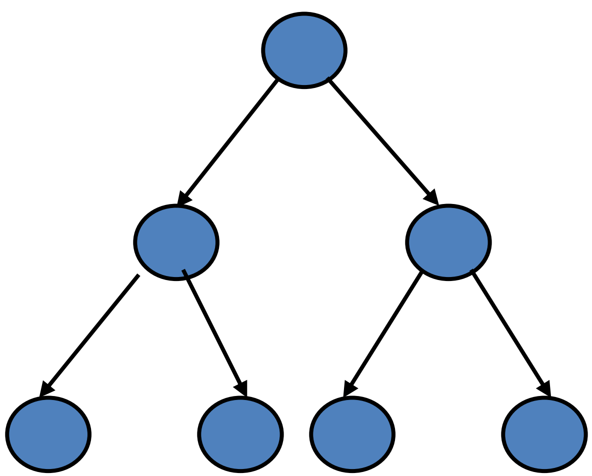
Proof: Replace every empty subtree with a leaf node to create a new tree. The new tree is a full binary tree.

Property of binary tree (I)

- A perfect BTree of height h has 2^h-1 nodes

No. of nodes = $2^0 + 2^1 + \ldots + 2^{(h-1)}$

$= 2^h - 1$



Level 0: 2^0 nodes

Level 1: 2^1 nodes

Level 2: 2^2 nodes

Property of binary tree (II)

- Consider a binary tree T of height h. The number of nodes of T is no more than 2^h-1

Reason: you cannot have more nodes than a perfect binary tree of height h.

Property of binary tree (III)

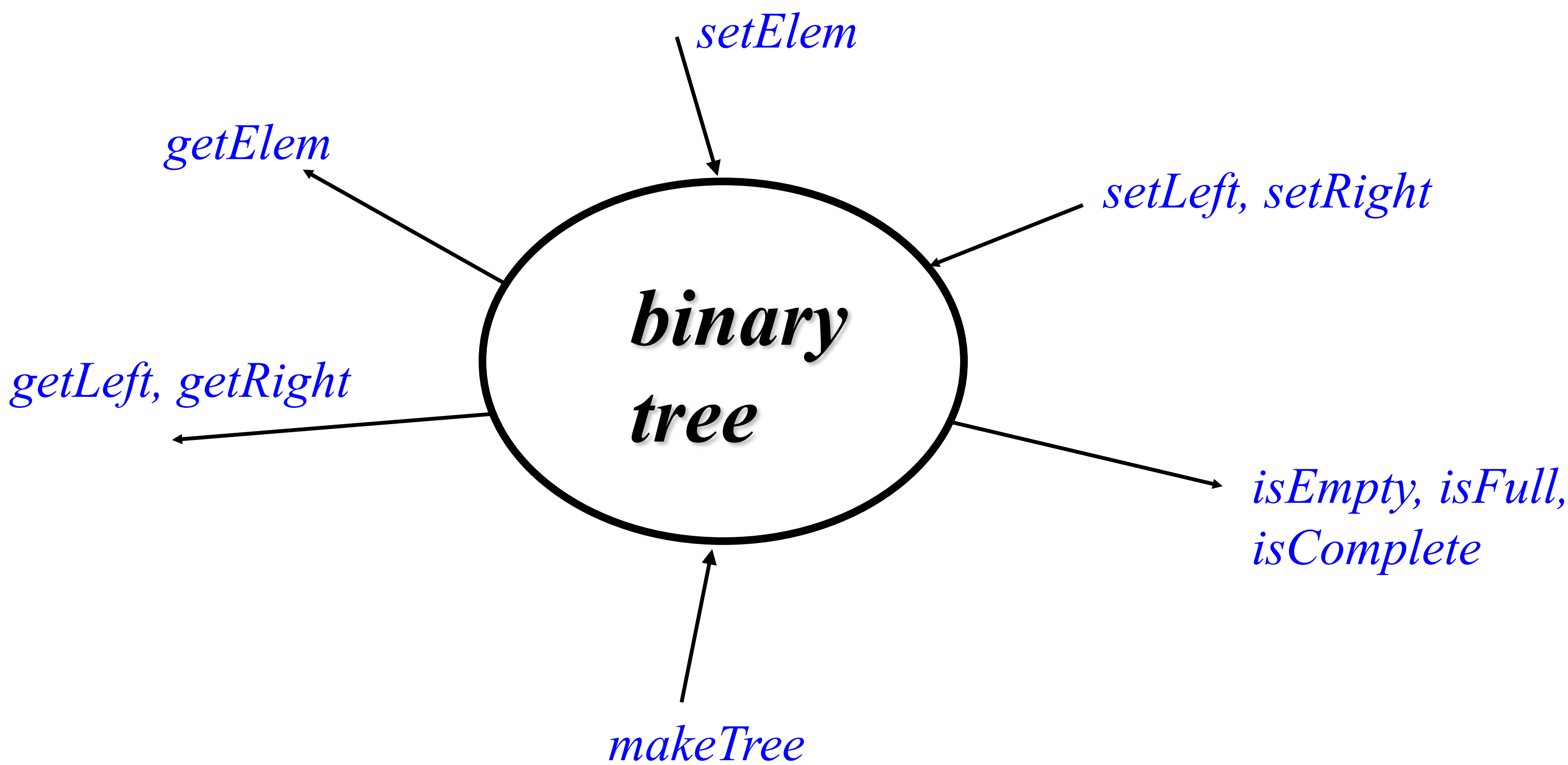
- The minimum height of a binary tree with n nodes is log(n+1)

By property (II), $n \leq 2^h-1$

Thus, $2^h \geq n+1$

That is, $h \geq \log_2 (n+1)$

Binary Tree ADT

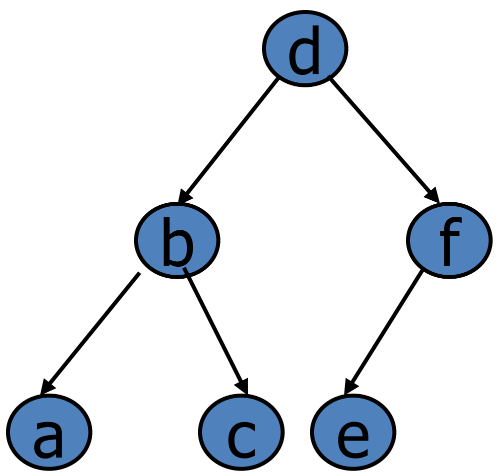


Implementation of Binary Tree

- Array-based implementation
- Pointer-based implementation

Array-based implementation

–1: denotes empty tree



nodeNum	item	leftChild	rightChild
0	d	1	2
1	b	3	4
2	f	5	-1
3	a	-1	-1
4	c	-1	-1
5	e	-1	-1
6	?	?	?
7	?	?	?
8	?	?	?
9	?	?	?
...	.....	.....	....

root

0

free

6

Pointer-based implementation

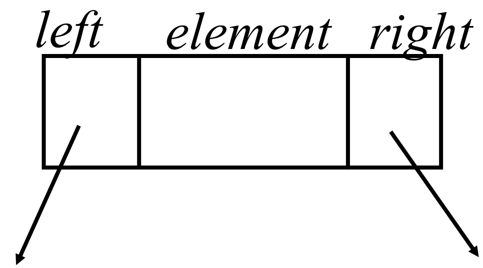
NULL: denote empty tree

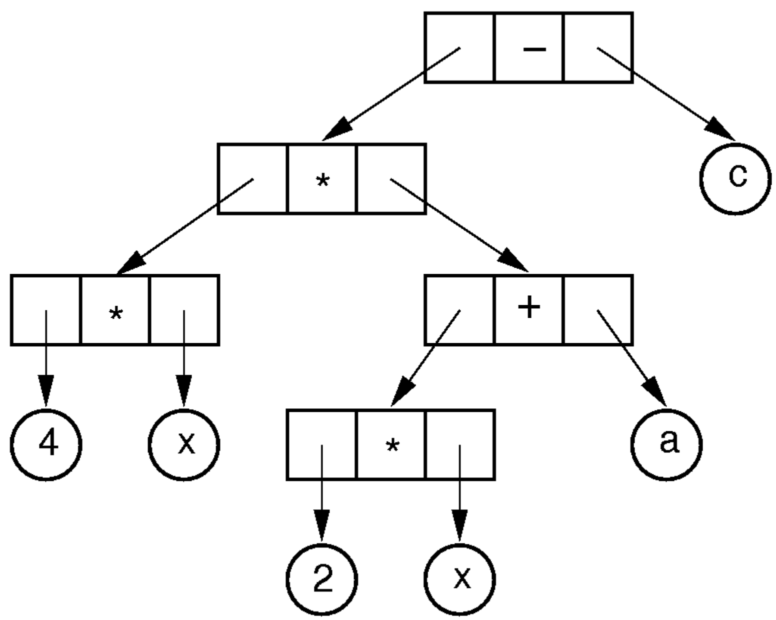
You can code this with a class of three fields:

Object element;

BinaryNode left;

BinaryNode right;





An expression tree for $4x(2x+a)-c$

Space Overhead (1/3)

- We consider pointer-based implementation.
- If all tree nodes have the same type, assume that there are n nodes
  - Data storage: $n * D$
  - Pointer Storage: $n * 2P$
  - Total storage: $n * (D + 2P)$
  - Overhead ratio: $2P / (D+2P)$
  - The ratio is $2/3$, if $D=P$.

P denotes the amount of a space required by a pointer.
D denotes the amount of a space required by a data value.

Space Overhead (2/3)

- If leaf nodes only store data (no pointers), then overhead depends on whether the tree is full. Consider a full binary tree:
  - Assume that there are n internal nodes
  - The number of leaves is $n+1$
  - Data storage: $(n+(n+1))D=(2n+1)D$
  - Pointer storage: $n*2P$
  - Total storage: $n*2P+(2n+1)D$
  - Overhead ratio: $\approx P/(P+D)$, when n is large
  - The ratio is $1/2$, if $P=D$.

Space Overhead (3/3)

- If only leaf nodes store useful information, then in a full binary tree with *n* internal nodes
  - The number of leaves is $n+1$
  - Useful data storage: $(n+1)D=(n+1)D$
  - *Empty data stroage: n*D*
  - Pointer storage: $n*2P$
  - Total storage: $n*2P+(2n+1)D$
  - Overhead ratio: $\approx (2P+D)/(2P+2D)$, when *n* is large
  - The ratio is 3/4, if $P=D$.

Tree Traversal

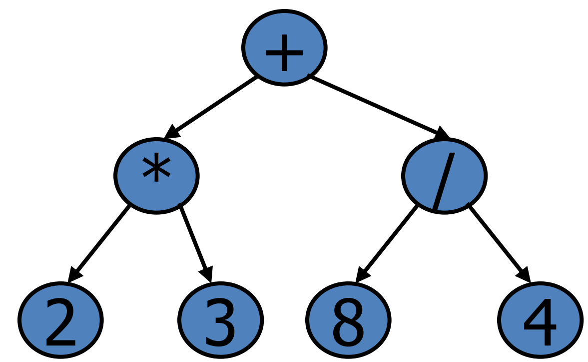
- Given a binary tree, we may like to do some operations on all nodes in a binary tree. For example, we may want to double the value in every node in a binary tree.
- To do this, we need a traversal algorithm which visits every node in the binary tree.

Ways to traverse a tree

- There are three main ways to traverse a tree:
  - Pre-order:
    - (1) visit node, (2) recursively visit left subtree, (3) recursively visit right subtree
  - In-order:
    - (1) recursively visit left subtree, (2) visit node, (3) recursively right subtree
  - Post-order:
    - (1) recursively visit left subtree, (2) recursively visit right subtree, (3) visit node
  - Level-order:
    - Traverse the nodes level by level
- In different situations, we use different traversal algorithm.

Examples for expression tree

- By pre-order, (prefix)
  + * 2 3 / 8 4
- By in-order, (infix)
  2 * 3 + 8 / 4
- By post-order, (postfix)
  2 3 * 8 4 / +
- By level-order,
  + * / 2 3 8 4
- Note 1: Infix is what we read!
- Note 2: Postfix expression can be computed efficiently using stack



Pre-order

```
Algorithm pre-order(BTree x)
if (x is not empty) {
  print x.getItem(); // you can do other things!
  pre-order(x.getLeftChild());
  pre-order(x.getRightChild());
}
```

Pre-order example

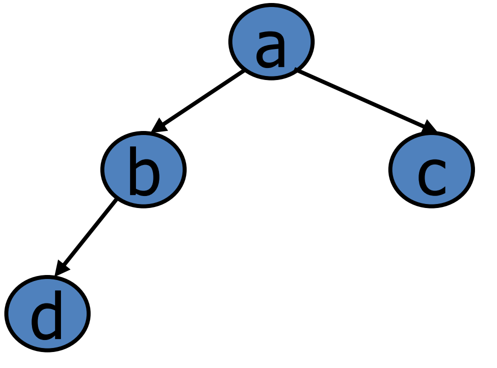
Pre-order(a);

Print a;
Pre-order(b);
Pre-order(c);

Print b;
Pre-order(d);
Pre-order(null);

Print d;
Pre-order(null);
Pre-order(null);

Print c;
Pre-order(null);
Pre-order(null);



a b d c

Time complexity of Pre-order Traversal

- For every node x, we will call pre-order(x) one time, which performs O(1) operations.
- Thus, the total time = O(n).

In-order and post-order

```
Algorithm in-order(BTree x)
If (x is not empty) {
  in-order(x.getLeftChild());
  print x.getItem(); // you can do other things!
  in-order(x.getRightChild());
}

Algorithm post-order(BTree x)
If (x is not empty) {
  post-order(x.getLeftChild());
  post-order(x.getRightChild());
  print x.getItem(); // you can do other things!
}
```

In-order example

In-order(a);

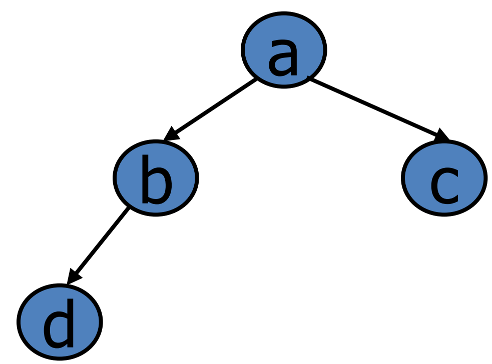
In-order(b);
Print a;
In-order(c);

In-order(d);
Print b;
In-order(null);

In-order(null);
Print d;
In-order(null);

In-order(null);
Print c;
In-order(null);

d b a c



Post-order example

Post-order(a);

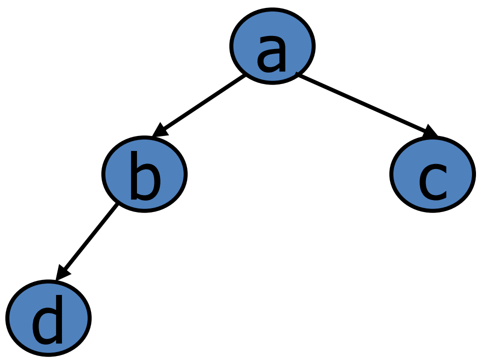
Post-order(b);
Post-order(c);
Print a;

Post-order(d);
Post-order(null);
Print b;

Post-order(null);
Post-order(null);
Print d;

Post-order(null);
Print c;
Post-order(null);

d b c a



Time complexity for in-order and post-order

- Similar to pre-order traversal, the time complexity is O(n).

Level-order

- Level-order traversal requires a queue!

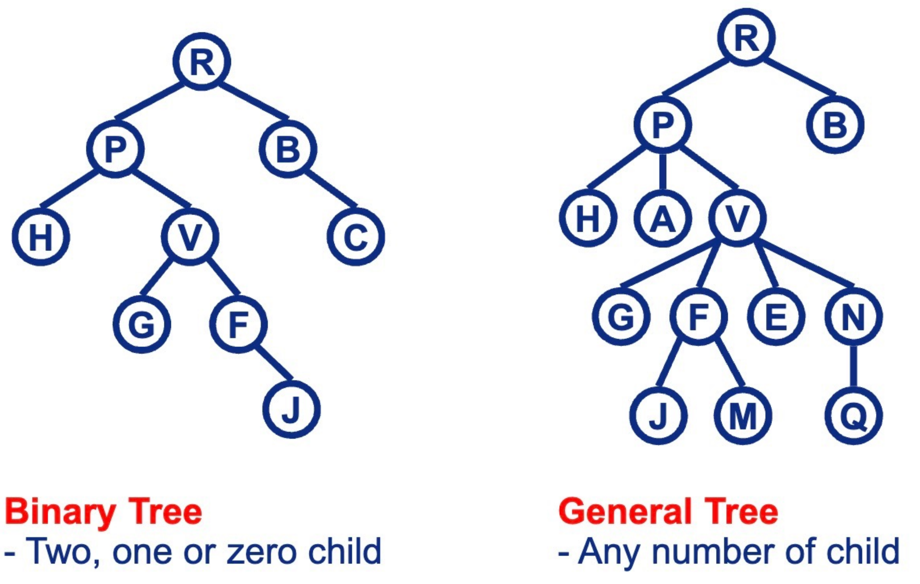
```
Algorithm level-order(BTree t)
  Queue Q = new Queue();
  BTree n;

  Q.enqueue(t);//  insert pointer t into Q

  while (! Q.empty()){
    n = Q.dequeue(); //  remove next node from the front of Q

    if (!n.isEmpty()){
      print n.getItem();//  you can do other things
      Q.enqueue(n.getLeft());  // enqueue left subtree on rear of Q
      Q.enqueue(n.getRight()); // enqueue right subtree on rear of Q
    };
  };
```

Time complexity of Level-order traversal

- Each node will enqueue and dequeue one time.
- For each node dequeued, it only does one print operation!
- Thus, the time complexity is O(n).

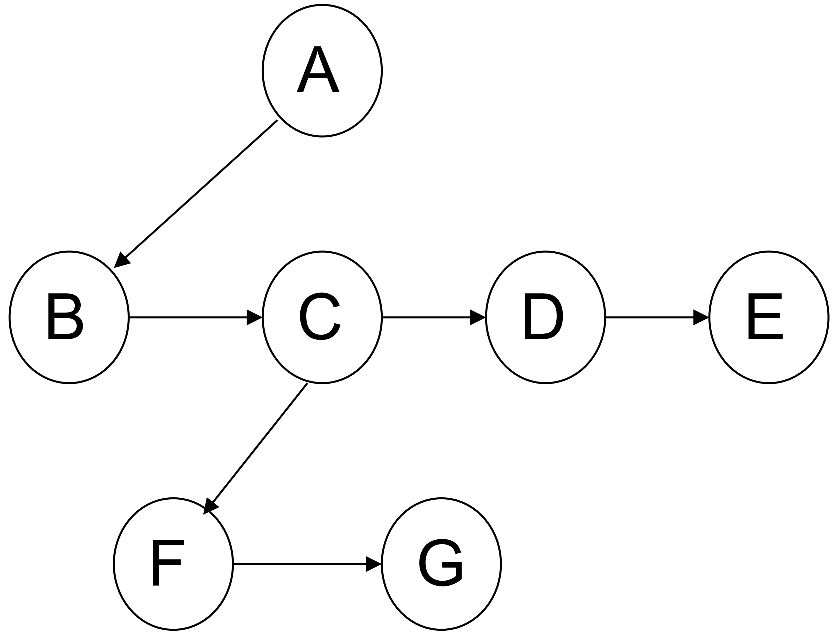
Non-Binary Trees

- Non-Binary Tree (General Tree)
  - A **non-binary** or general tree is a tree in which at least one node has more than two children. Such nodes are referred to as **polytomies**, or non-binary nodes.



General tree implementation

```
struct TreeNode
{
    Object      element
    TreeNode *firstChild
    TreeNode *nextsibling
}
```

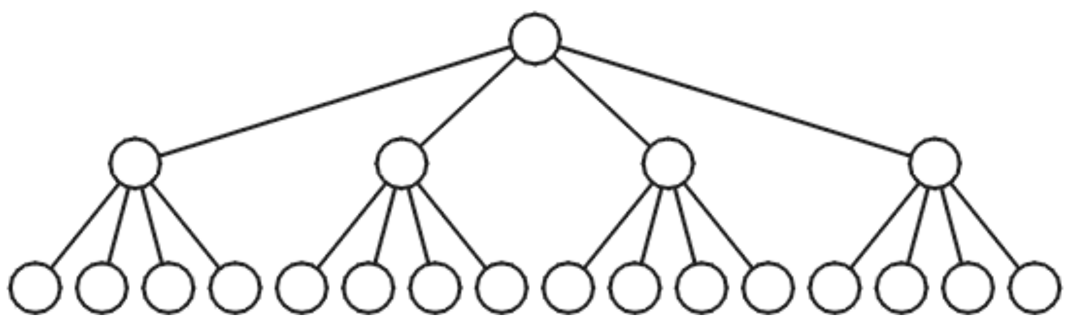


because we do not know how many children a node has in advance.

- Traversing a general tree is similar to traversing a binary tree.

K-ary trees

- An *K-ary tree* is a tree
  - where each node can have up to *K* children, where each of the children are non-overlapping *K*-ary trees.
- The **PR quadtree** discussed later is a 4-ary tree.



An example of 4-ary tree

- Full and Complete *K*-ary trees are analogous to full and complete binary trees, respectively.

Applications of binary trees

- Binary search trees
- Heaps and priority queues
- Huffman coding trees

Binary Search Tree (BST)

- Unsorted list for Dictionary implementation
  - inserting a new record ← quick
  - searching an unsorted list ← $\Theta(n)$ on average
- Is there any solution to seep up?
  - Binary search tree (BST)
- A BST is a binary tree, iff
  - For each node, assume the node value is *K*
  - The values of the nodes in its left subtree are $< K$
  - The values of the nodes in its right subtree are $\geq K$

BST class

```
template <typename Key, typename E>
class BST {
private:
   BSTNode<Key,E>*  root; // Root of the BST
   int          nodeCount; // Number of nodes in the BST
public:
   BST() { root = NULL; nodecount = 0; } // Constructor
  ~BST() { clearhelp(root); } // Destructor
  void clear() // Reinitialize tree
  { clearhelp(root); root = NULL; nodecount = 0; }
```

BST clear

```
void clearhelp(BSTNode<Key, E>* rt) {
    if (rt == NULL) return;
    //postorder traversal
    clearhelp( rt->left() );
    clearhelp( rt->right() );
    delete rt;
}
```

- Time complexity is $\Theta(n)$ with n nodes

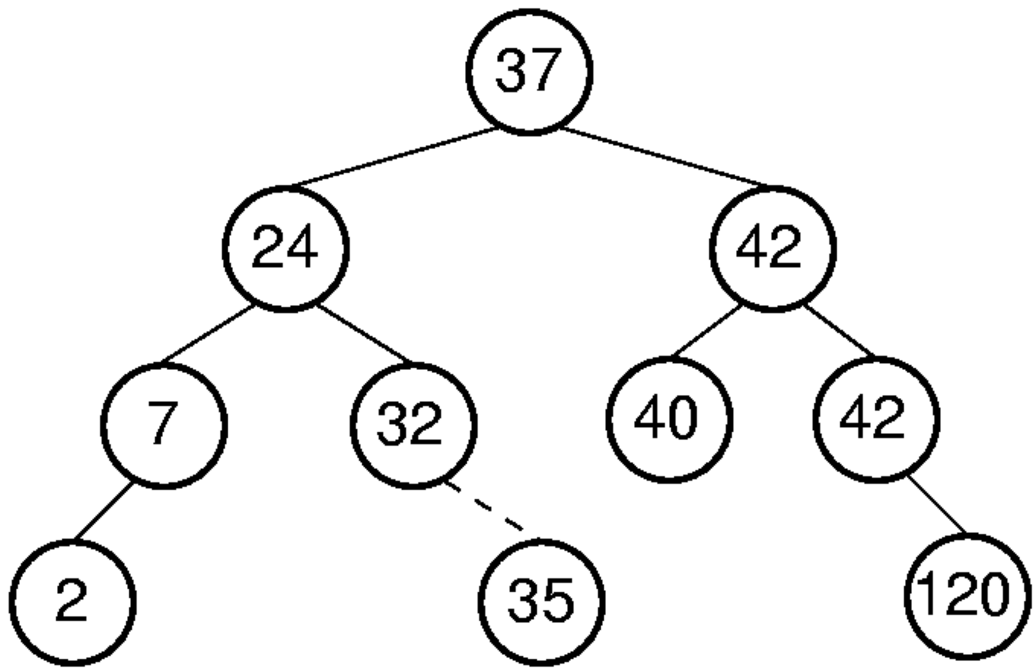
BST Search

```
E findhelp(BSTNode<Key, E>* rt, const Key& k) const {
    if (rt == NULL) return NULL; // Empty tree
    if (k < rt->key())
        return findhelp( rt->left(), k); // Check left subtree
    else if (k > rt->key())
        return findhelp( rt->right(), k); // Check right
    else return rt->element(); // Found it
}
```

- Time complexity of search is $\Theta(\ d\ )$ if the height of the tree is *d*

BST Insert (1)

- Time complexity of insertion is $\Theta(\, d\,)$ if the height of the tree is *d*



BST Insert (2) – similar to search

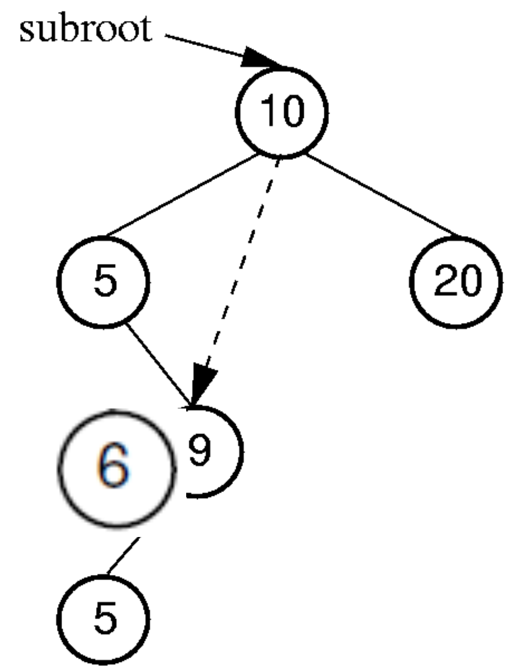
```
BSTNode<Key, E>* inserthelp( BSTNode<Key, E>* root,
                             const Key& k,    const E& it) {
  if (rt == NULL) // different: Empty tree: create node
     return new BSTNode<Key, E>(k, it, NULL, NULL);
  BSTNode<Key, E>* tmp;
  if (k < rt->key()){
        tmp = inserthelp(rt->left(), k, it) ;
        rt->setLeft( tmp );
  }else{  // k >= rt->key()
        tmp = inserthelp(root->right(), k, it);
        root->setRight(tmp);
  }
  return root; // Return tree with node inserted
}
```

BST Removal

- First consider removing the node with the minimum value
- Then, consider the general case

Remove Minimum Value

- Where is the minimum value stored ?
  - The most left node in the tree
- How to modify pointers?
  - Let its parent point to its right child



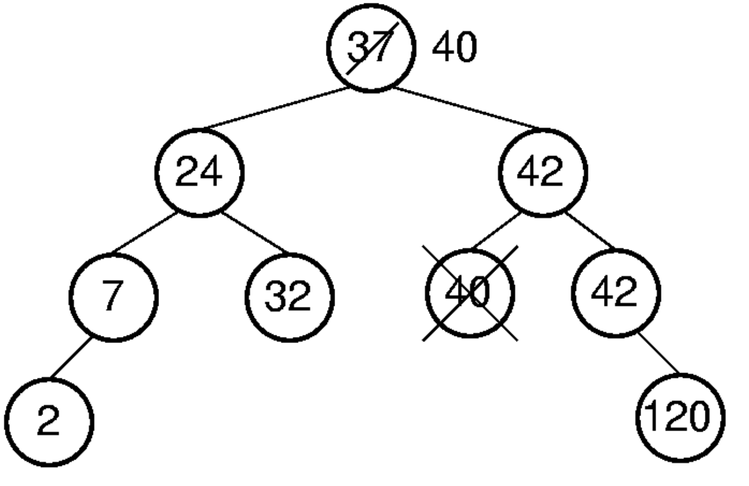

```
BSTNode<Key, E>* deletemin(BSTNode<Key, E>* rt) {
        if (rt->left() == NULL) // Found min
                return rt->right();
        else { // Continue left
            rt->setLeft(   deletemin(rt->left()) );
            return rt;
        }
}
```

BST removal – general case

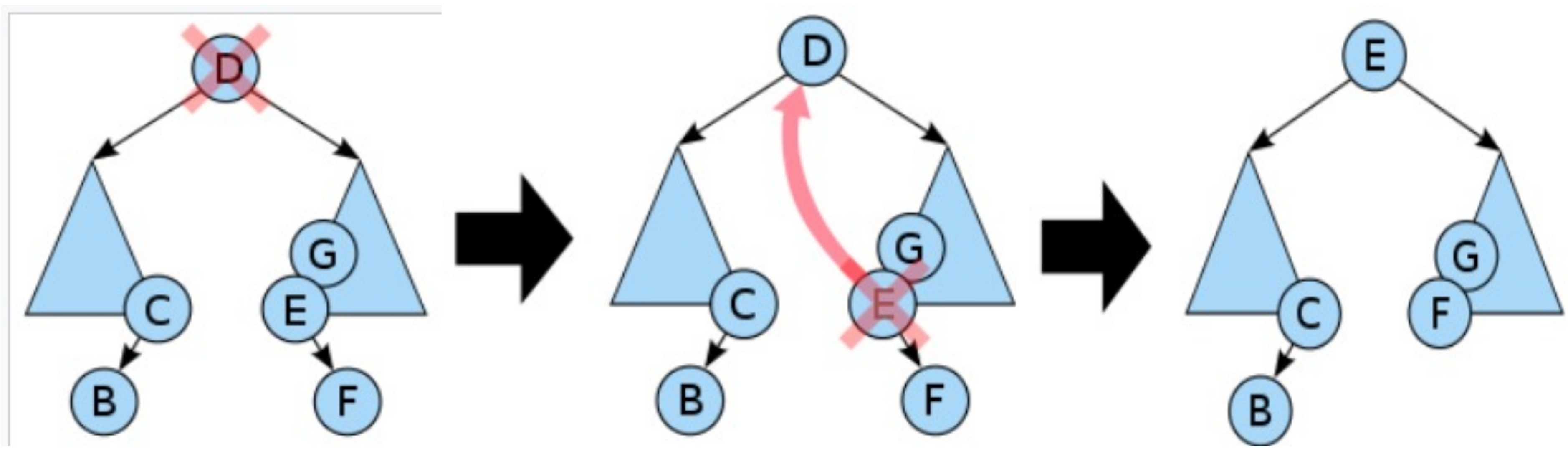
- Only three cases
- Case 1: Remove a node with no children
  - Simply remove the node
- Case 2: Remove a node with only one child
  - Similar to the case of removing the minimum, by letting its parent point to its child
- Case 3: Remove a node with two children
  - Transformed to case 2

BST Remove, case 3

- Now remove 37
- Find the minimum value larger than 37, i.e., 40
- 40 is the minimum value in its right subtree
- Replace 37 with 40
- Remove the node that previously contains 40



BST Remove, case 3 example



```
BSTNode<Key, E>* removehelp(BSTNode<Key, E>* rt, const Key& k) {
        if (rt == NULL) return NULL; // k is not in tree
        else if (k < rt->key())
                rt->setLeft(  removehelp(rt->left(), k));
       else if (k > rt->key())
                rt->setRight(  removehelp(rt->right(), k));
      else { // Found: remove it
                BSTNode<Key, E>* temp = rt;
                if (rt->left() == NULL) { // Only a right child
                        rt = rt->right(); // so point to right
                        delete temp;
                }
               else if (rt->right() == NULL) { // Only a left child
                        rt = rt->left(); // so point to left
                       delete temp;
               }
```

```
        else { // Both children are non-empty
            BSTNode<Key, E>* temp = getmin(rt->right());
             rt->setElement( temp->element() );
            rt->setKey( temp->key() );
            rt->setRight(deletemin(rt->right()));
            delete temp;
        }
     }
     return rt;
}
```

- Time complexity of removal is $\Theta(\ d\ )$ if the height of the tree is d

Time Complexity of BST Operations

- Search: $\Theta(d)$
- Insertion: $\Theta(d)$
- removal: $\Theta(d)$

- d = the tree height
- d is $\Theta(\log n)$ if tree is balanced.
- What is the worst case?
  - $\Theta(n)$
- How to obtain a balanced tree ?
  - See Chapter 13.2 for the AVL balanced tree if you are interested

Heaps and Priority Queues

- Problem: We want a data structure that stores records as they come (insert), but on request, releases the record with the greatest value (removemax)

- Example: Scheduling jobs in a multi-tasking operating system, the value of each task is its priority

Priority Queues-cont.

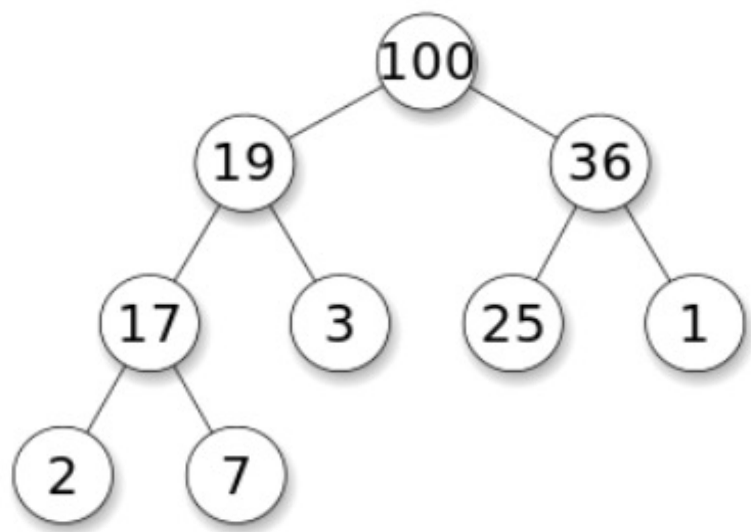
Possible Solutions:

- A simple linked list
  - insert appends to a linked list ($\Theta(1)$)
  - removemax determines the maximum by scanning the list ($\Theta(n)$)
- A linked list is used and is in decreasing order
  - insert places an element in its correct position ($\Theta(n)$)
  - removemax removes the head of the list ($\Theta(1)$).
- Use a *heap* – both insert and removemax are $\Theta(\log n)$, introduced later

Heap – a special binary tree

Heap: Complete Btree with the heap property:

- Max-heap: each value in a node is no less than its children values
- The values in the tree are partially ordered.
  - The left child may less or greater than its right child

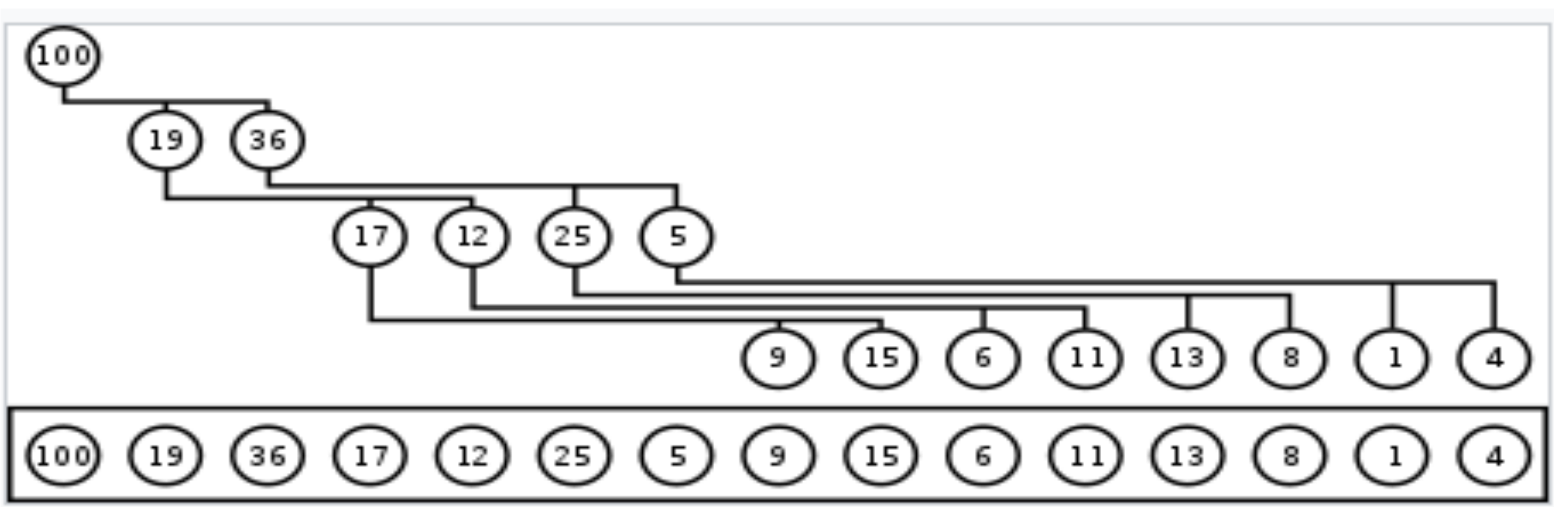


Array-based Heap Implementation

- Logic topology:
- It is a complete

binary tree

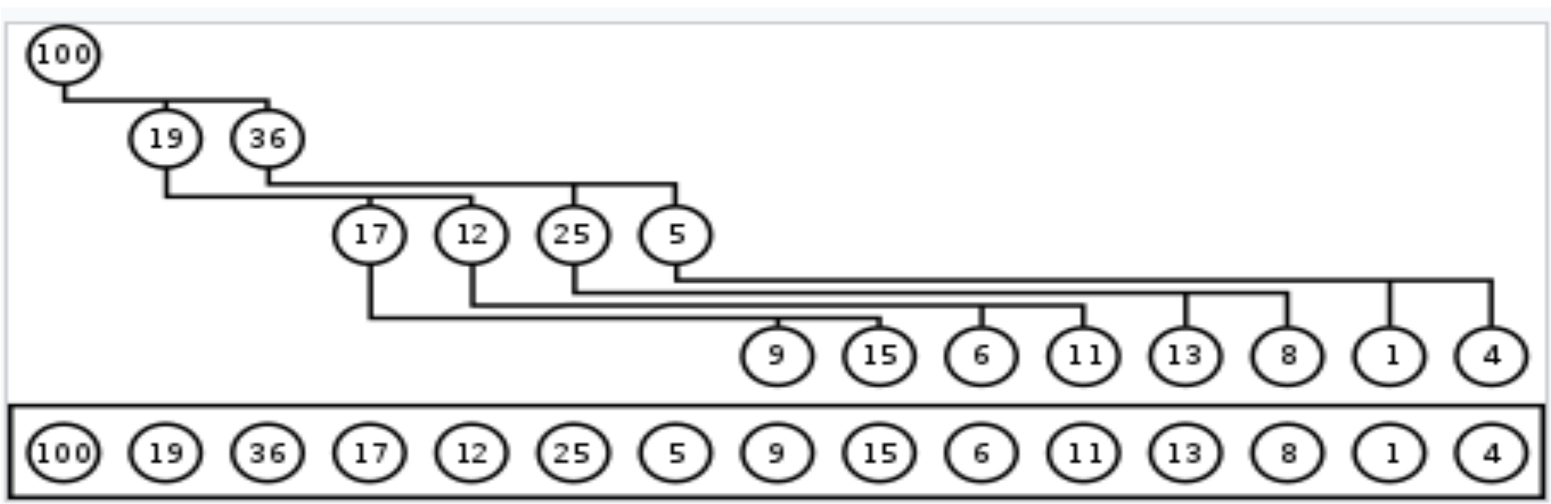
- Tree height $\Theta(\log n)$



Array Implementation (1)

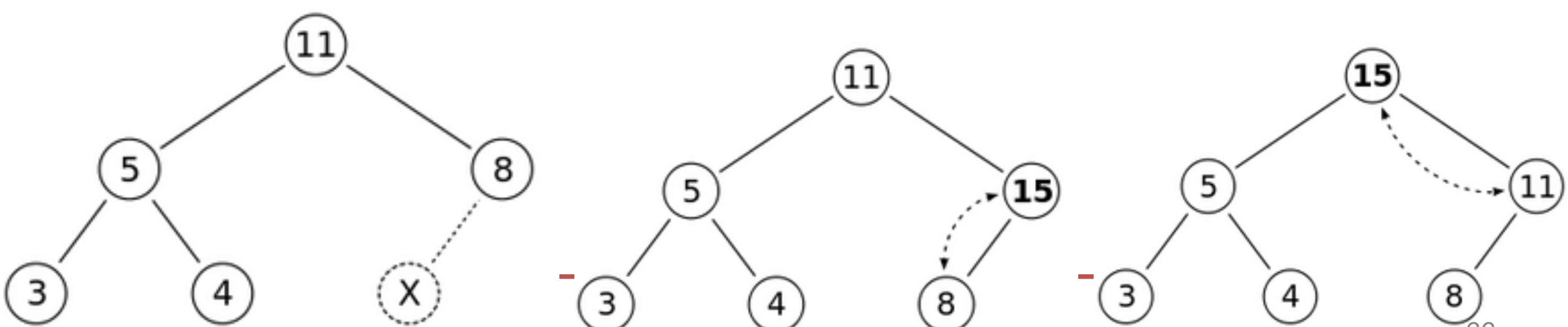
If a node is stored at array[r], where are its parent and children stored ?

- Parent (r) = (r-1)/2 if $r \neq 0$ and $r < n$.
- Leftchild(r) =2r + 1 if 2r + 1 < n.
- Rightchild(r) =2r + 2 if 2r + 2 < n.



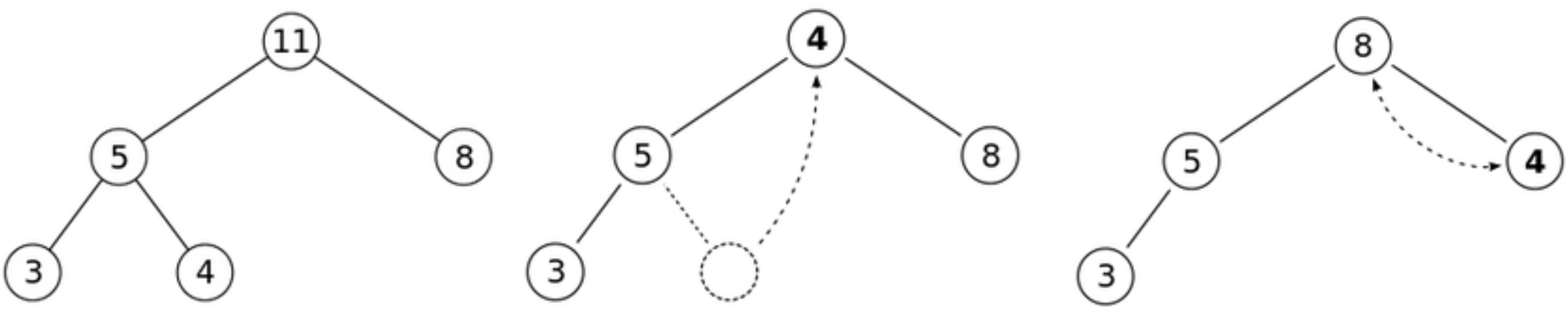
Heap -- insert

- Add the element to the bottom level of the heap, i.e., Heap[n], then n++, suppose x=15
- Compare the added element with its parent (shift up operation)
  - if it is no greater than its parent, stop
  - If not (i.e., larger), swap the element with its parent and return to the previous step
  - Worst time complexity Θ(log n)



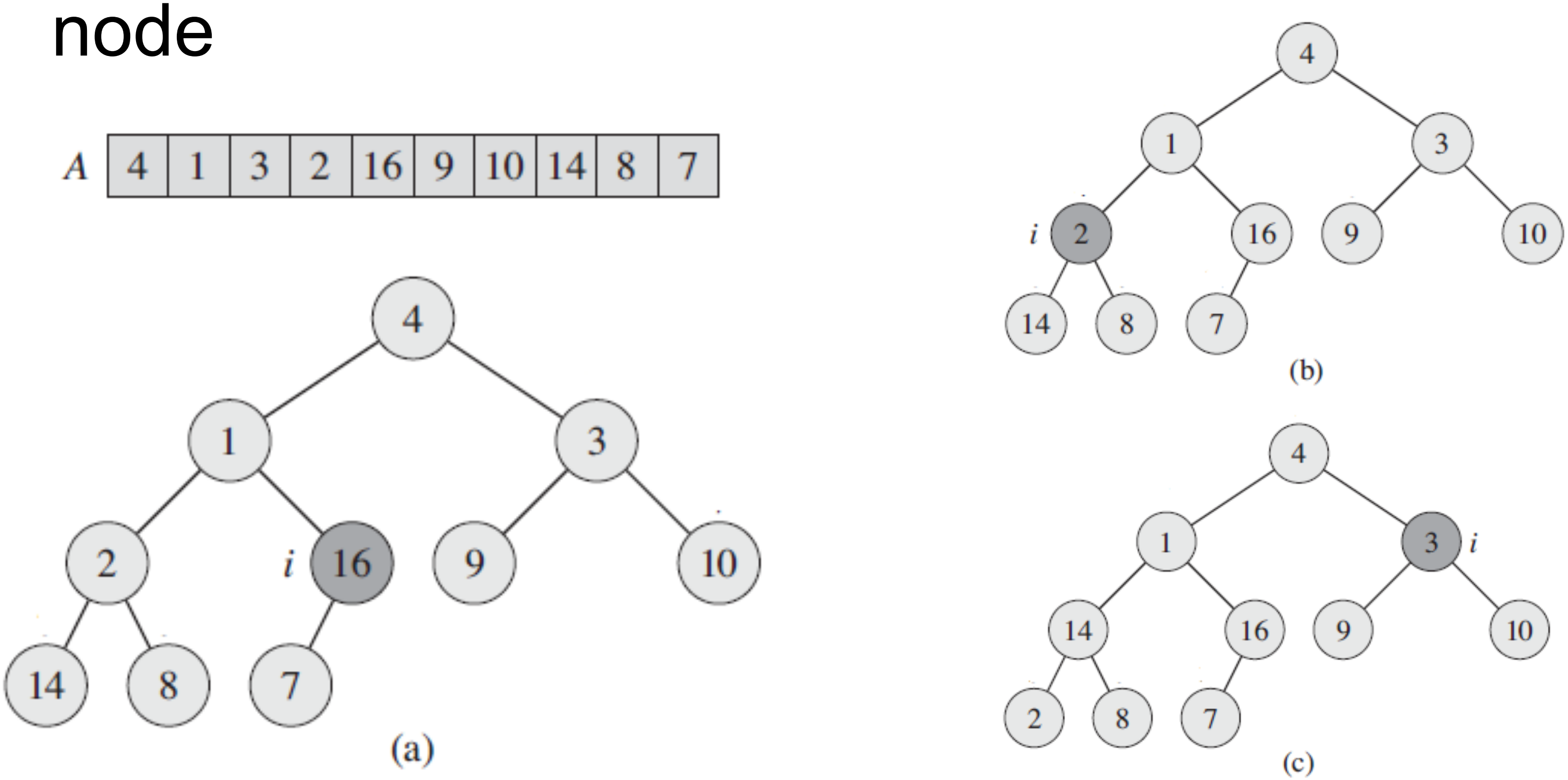
Heap -- removeMax

- Replace the root of the heap with the last element on the last level
- Compare the new root with its children (shift down operation)
  - if the new root is larger than its children, stop.
  - If not, swap the element with its largest children, and return to the previous step
  - Worst time complexity Θ(log n)



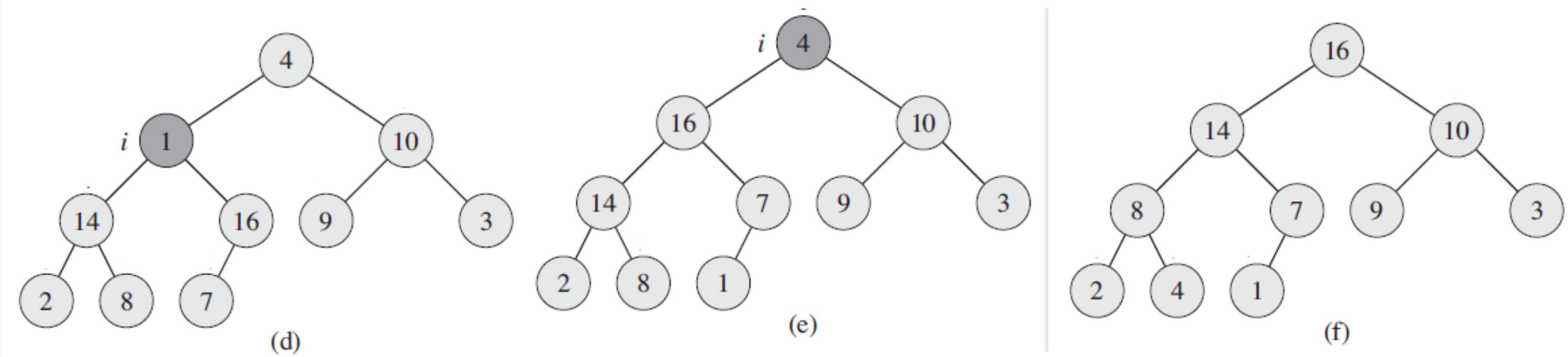
Build a Heap from an array

- Build from the middle to the first node in the array, perform a shift-down operation for each node



Build a Heap from an array (cont.)

- Time complexity of building a heap is $\Theta(n)$, see the textbook for its analysis



Huffman coding

- Computers store data with bits 0 and 1
- Assume that there are only three types of letters A, B, and C in a text
  - letter A appears 98 times
  - both B and C appear only once
  - there are 100 letters in the text
- How to encode letters A, B, C, such that the total number of bits to represent the 100 letters is minimized?

Possible encoding solutions

- Solution 1: each letter is coded with two bits
  - A: bits 00
  - B: bits 01
  - C: bits 10
  - 98*2+1*2+1*2= **200** bits are needed
- Solution 2: non-equal length encoding
  - A: bit 0, as A appears more frequently
  - B: bits 10
  - C: bits 11
  - 98*1 + 1*2 + 1*2 = **102** bits are needed!

Huffman coding

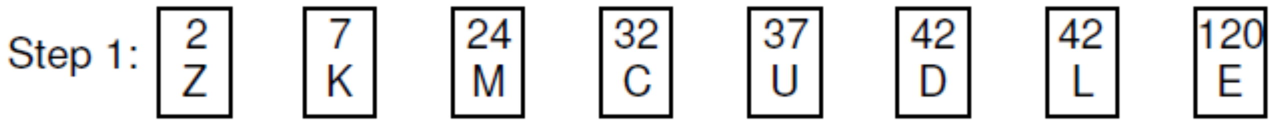
- Consider a general case with more than three letters ?

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

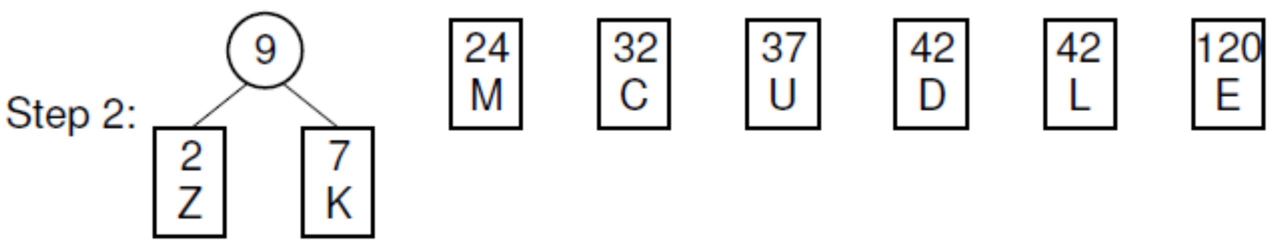
- **David Albert Huffman** (1925–1999) solved the problem in 1952, when he was a Ph.D. student at MIT.
- This coding method is named by his family name
- Basic idea: assign short codes for frequent letters, but long codes for rare letters

Huffman coding solution

1. Create *n* initial Huffman trees, each a single leaf node containing one of the letters.

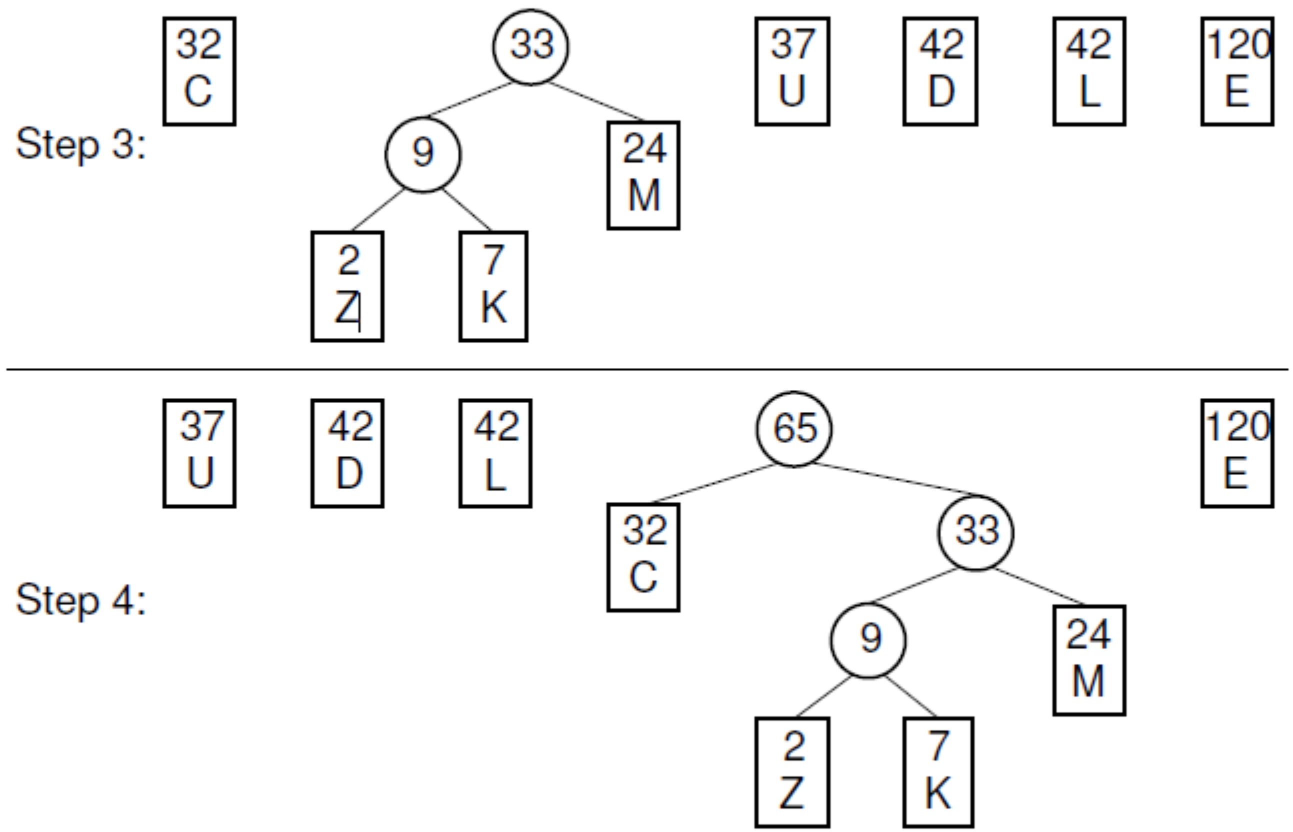


2. Select the two trees with the lowest weights, create a new tree by joining them
   - Its root has the two trees as children
   - The root weight is the sum of the weights of the two trees



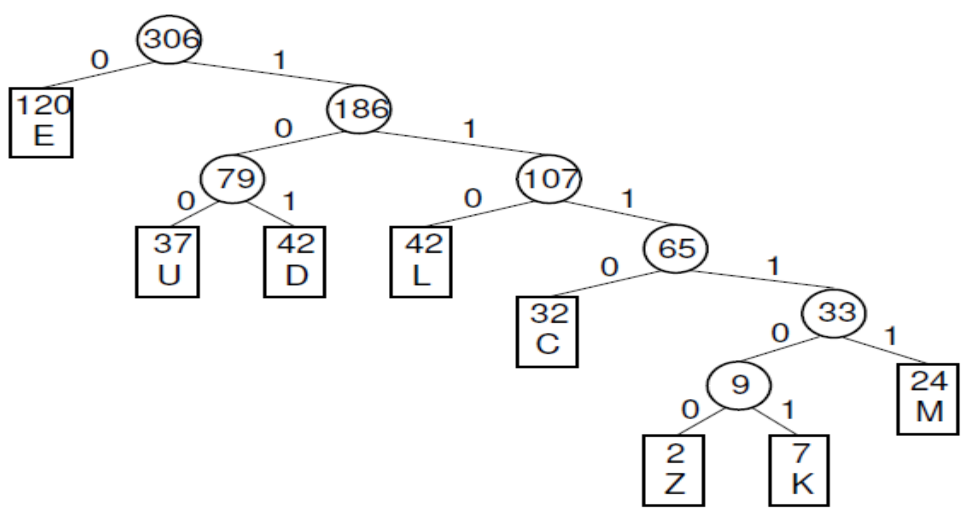
Huffman coding solution – cont.

3. Continue Step 2 until only one tree is left



Assign codes based on the final tree

- Beginning at the root
  - '0' is assigned to edges linking a node with its left child
  - '1' to edges connecting a node with its right child
- The code of each letter is the binary number on the path from the root to its letter leaf node
  - e.g., the code of letter C is 1110



Summary

- We have discussed
  - the tree data-structure.
  - Binary tree *vs* general tree
  - Binary tree ADT
    - Can be implemented using arrays or references
  - Tree traversal
    - Pre-order, in-order, post-order, and level-order
  - Binary tree applications
    - Binary search tree, heaps and priority queues, Huffman coding trees