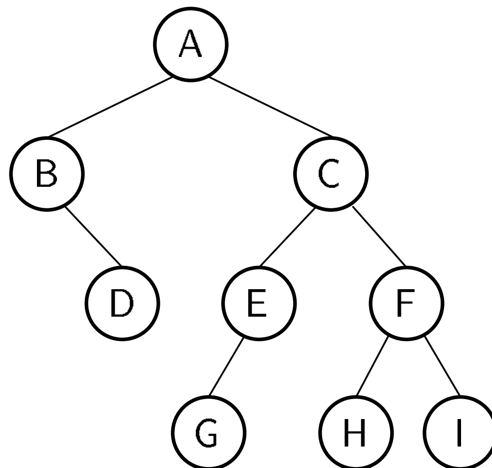
Binary Trees

A **binary tree** is made up of a finite set of nodes that is either **empty** or consists of a node called the **root** together with two binary trees, called the left and right **subtrees**, which are disjoint from each other and from the root.

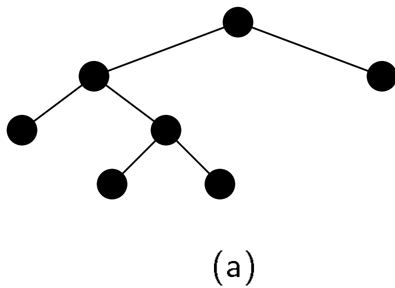
Notation: **Node**, **Children**, **Edge**, **Parent** **Ancestor**, **Descendant**, **Path**, **Depth**, **Height**, **Level**, **Leaf Node**, **Internal Node**, **Subtree**.



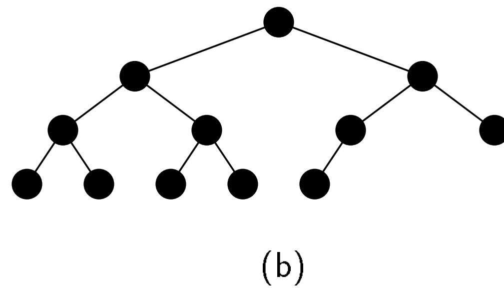
Full and Complete Binary Trees

Full binary tree: each node either is a leaf or is an internal node with exactly two non-empty children.

Complete binary tree: If the height of the tree is d, then all levels except possibly level d are completely full. The bottom level has all nodes to the left side.



(a)



(b)

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):

- **Base Case**: A full binary tree with 1 internal node must have two leaf nodes.
- **Induction Hypothesis**: Assume any full binary tree **T** containing $n-1$ internal nodes has n leaves.
- **Induction Step**: Given tree **T** with n internal nodes, pick internal node I with two leaf children. Remove I's children, call resulting tree **T'**. By induction hypothesis, **T'** is a full binary tree with n leaves. Restore i's two children. The number of internal nodes has now gone up by 1 to reach n. The number of leaves has also gone up by 1.

Full Binary Tree Theorem Corollary

Theorem: The number of `NULL` pointers in a non-empty binary tree is one more than the number of nodes in the tree.

Proof: Replace all null pointers with a pointer to an empty leaf node. This is a full binary tree.

Binary Tree Node ADT

```
interface BinNode { // ADT for binary tree nodes
  // Return and set the element value
  public Object element();
  public Object setElement(Object v);

  // Return and set the left child
  public BinNode left();
  public BinNode setLeft(BinNode p);

  // Return and set the right child
  public BinNode right();
  public BinNode setRight(BinNode p);

  // Return true if this is a leaf node
  public boolean isLeaf();
} // interface BinNode
```

Traversals

Any process for visiting the nodes in some order is called a **traversal**.

Any traversal that lists every node in the tree exactly once is called an **enumeration** of the tree's nodes.

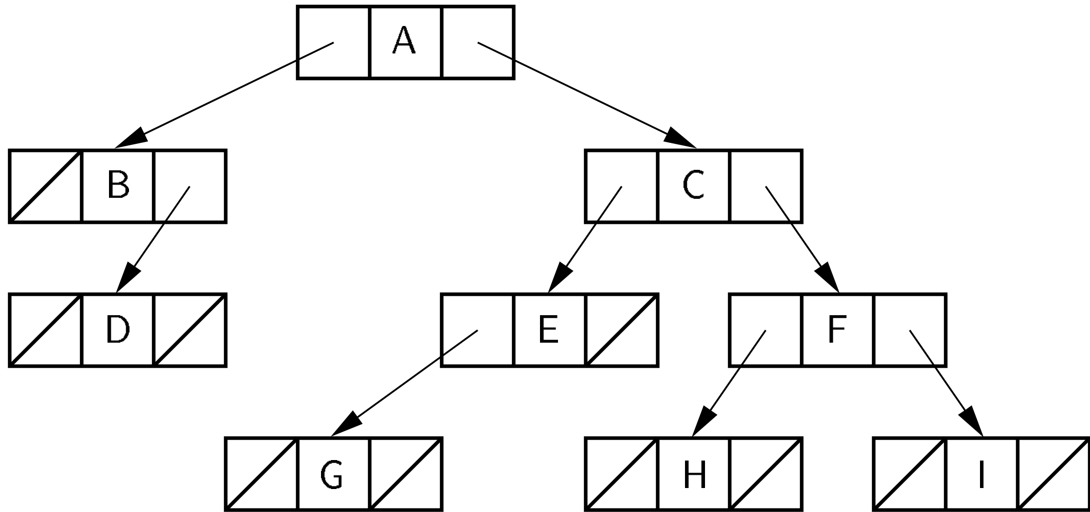
Preorder traversal: Visit each node *before* visiting its children.

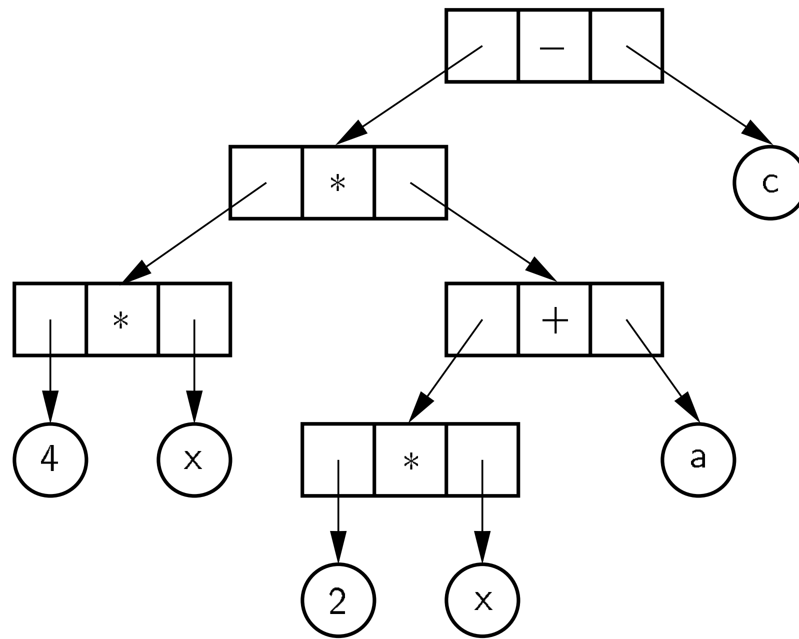
Postorder traversal: Visit each node *after* visiting its children.

Inorder traversal: Visit the left subtree, then the node, then the right subtree.

```
void preorder(BinNode rt) // rt is root of subtree
{
  if (rt == null) return; // Empty subtree
  visit(rt);
  preorder(rt.left());
  preorder(rt.right());
}
```

Binary Tree Implementation





Inheritance

```
class LeafNode implements BinNode { // Leaf node
  private String var;               // Operand value

  public LeafNode(String val) { var = val; }
  public Object element() { return var; }
  public Object setElement(Object v)
    { return var = (String)v; }
  public BinNode left() { return null; }
  public BinNode setLeft(BinNode p) { return null; }
  public BinNode right() { return null; }
  public BinNode setRight(BinNode p) { return null; }
  public boolean isLeaf() { return true; }
} // class LeafNode

class IntlNode implements BinNode { // Internal node
  private BinNode left;             // Left child
  private BinNode right;            // Right child
  private Character opx;            // Operator value

  public IntlNode(Character op, BinNode l, BinNode r)
    { opx = op; left = l; right = r; } // Constructor
  public Object element() { return opx; }
  public Object setElement(Object v)
    { return opx = (Character)v; }
  public BinNode left() { return left; }
  public BinNode setLeft(BinNode p) {return left = p;}
  public BinNode right() { return right; }
  public BinNode setRight(BinNode p)
    { return right = p; }
  public boolean isLeaf() { return false; }
} // class IntlNode
```

Inheritance (cont)

```
static void traverse(BinNode rt) { // Preorder
  if (rt == null) return;          // Nothing to visit
  if (rt.isLeaf())                 // Do leaf node
    System.out.println("Leaf: " + rt.element());
  else {                           // Do internal node
    System.out.println("Internal: " + rt.element());
    traverse(rt.left());
    traverse(rt.right());
  }
}
```

Space Overhead

From Full Binary Tree Theorem:

Half of pointers are `NULL`.

If leaves only store information, then overhead depends on whether tree is full.

All nodes the same, with two pointers to children:

Total space required is $(2p + d)n$.

Overhead: $2pn$.

If $p = d$, this means $2p/(2p + d) = 2/3$ overhead.

Eliminate pointers from leaf nodes:

$$\frac{\frac{n}{2}(2p)}{\frac{n}{2}(2p) + dn} = \frac{p}{p + d}$$

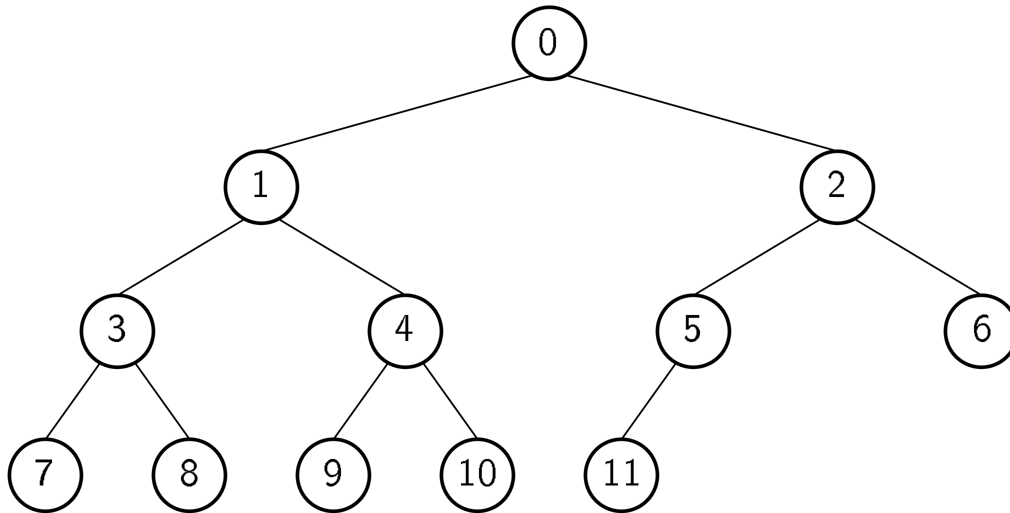
This is $1/2$ if $p = d$.

$2p/(2p + d)$ if data only at leaves $\Rightarrow$ $2/3$ overhead.

Some method is needed to distinguish leaves from internal nodes.

Array Implementation

For complete binary trees.



(a)

Node	0	1	2	3	4	5	6	7	8	9	10	11

- Parent(r) =
- Leftchild(r) =
- Rightchild(r) =
- Leftsibling(r) =
- Rightsibling(r) =

Huffman Coding Trees

ASCII Codes: 8 bits per character.

Fixed length coding.

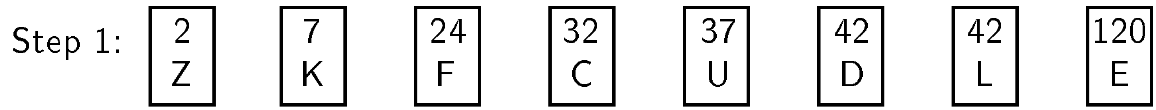
Can take advantage of relative frequency of letters to save space.

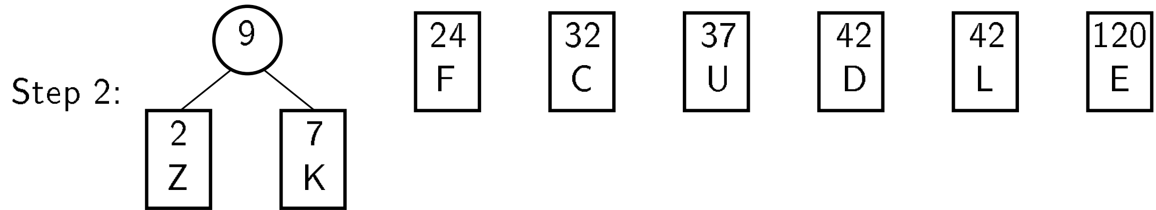
Variable length coding.

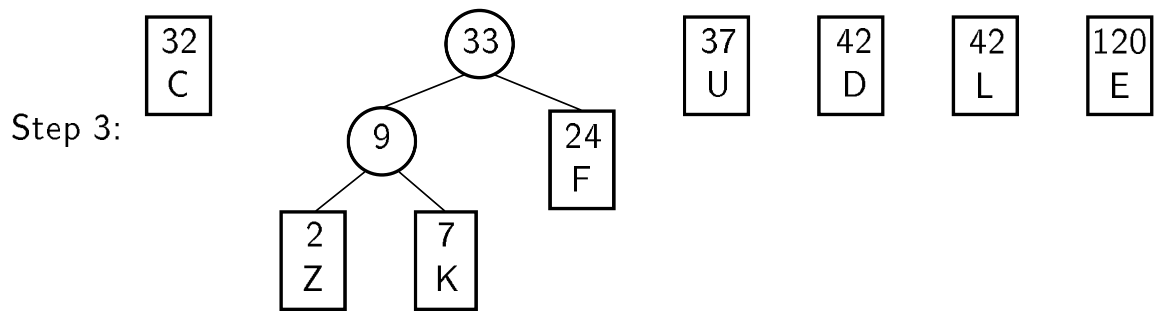
Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

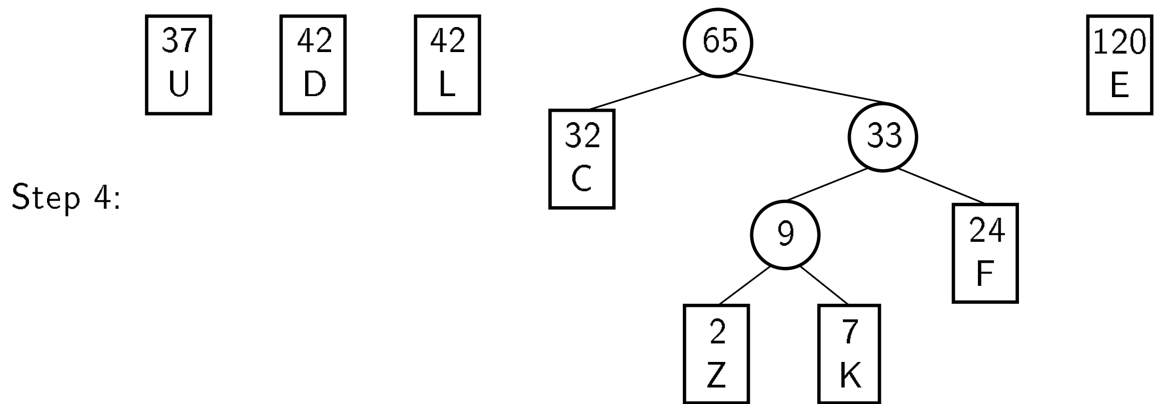
Build the tree with **minimal external path weight**.

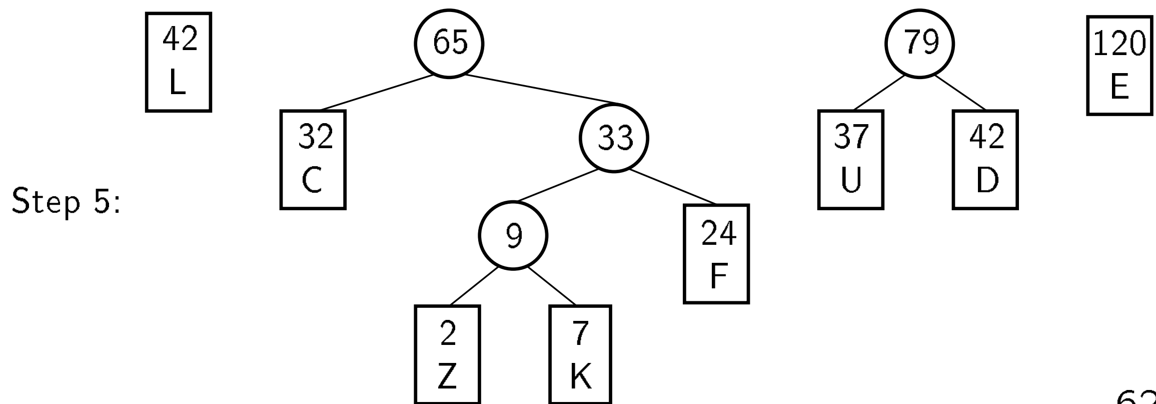
Huffman Tree Construction



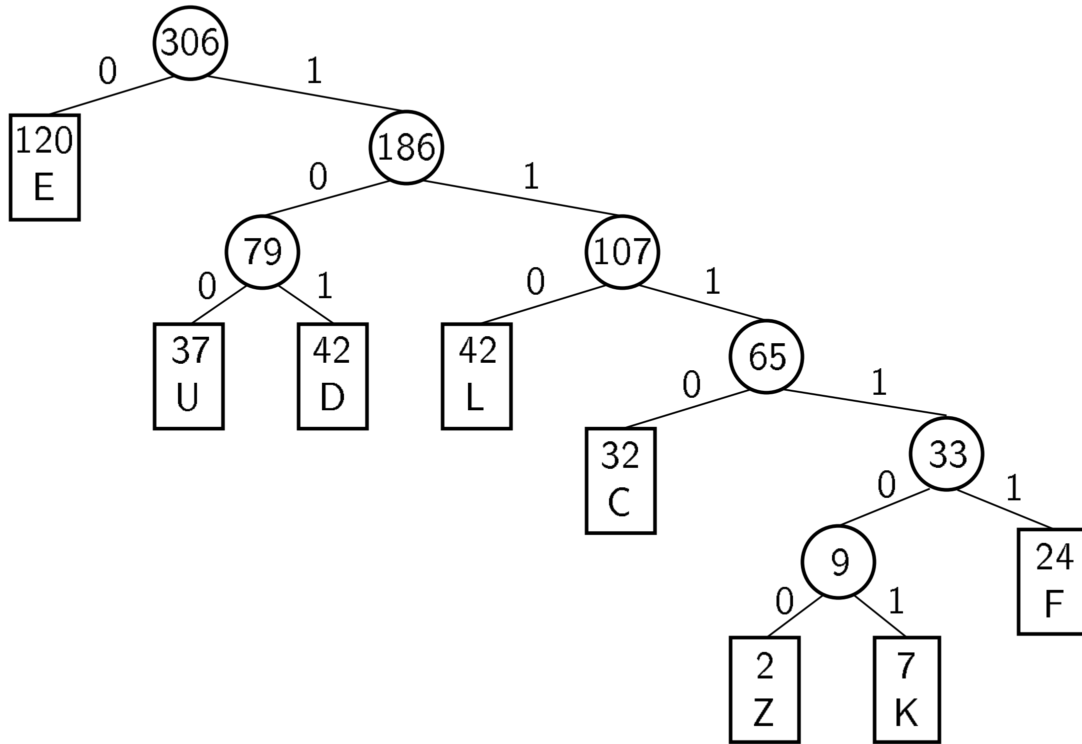








Assigning Codes



Letter	Freq	Code	Bits
C	32		
D	42		
E	120		
F	24		
K	7		
L	42		
U	37		
Z	2		

Coding and Decoding

A set of codes are said to meet the **prefix property** if no code in the set is the prefix of another.

Code for DEED:

Decode 1011001110111101:

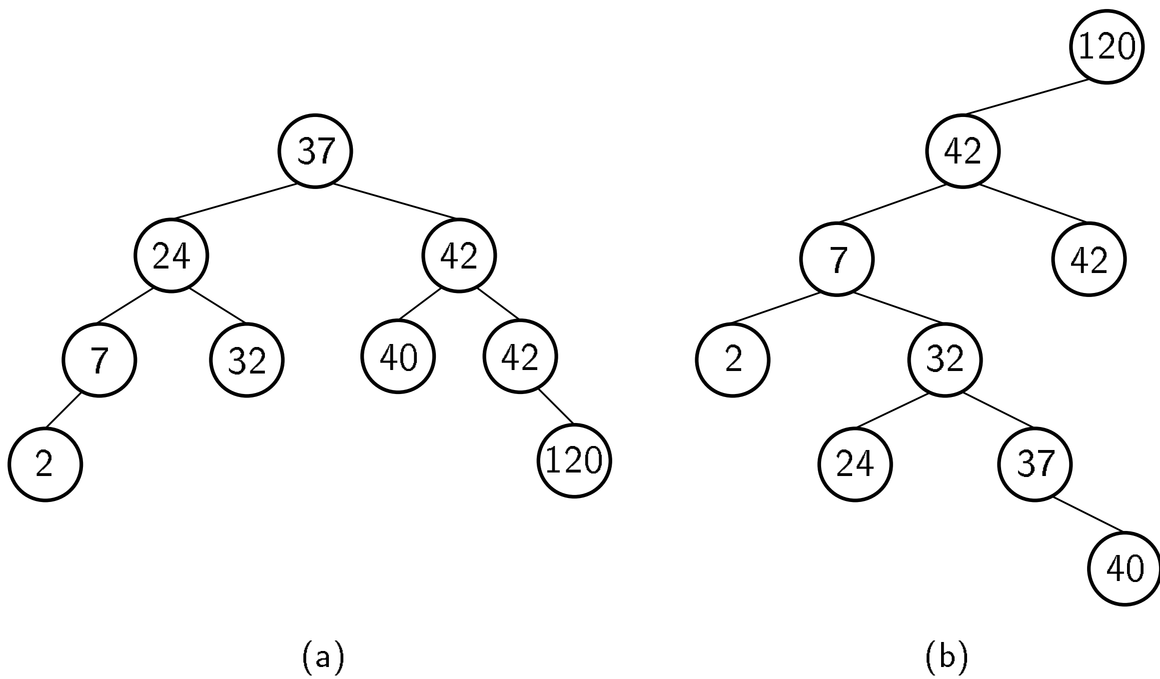
Expected cost per letter:

Letter	Freq	Code	Bits
C	32	1110	4
D	42	101	3
E	120	0	1
F	24	11111	5
K	7	111101	6
L	42	110	3
U	37	100	3
Z	2	111100	6

Binary Search Trees

Binary Search Tree (BST) Property

All elements stored in the left subtree of a node whose value is K have values less than K. All elements stored in the right subtree of a node whose value is K have values greater than or equal to K.



BinNode Class

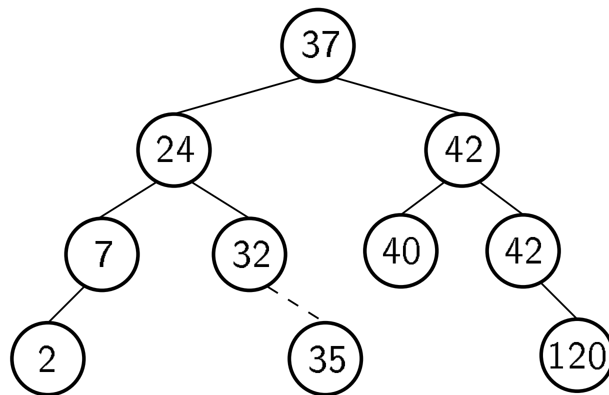
```
interface BinNode { // ADT for binary tree nodes
  // Return and set the element value
  public Object element();
  public Object setElement(Object v);

  // Return and set the left child
  public BinNode left();
  public BinNode setLeft(BinNode p);

  // Return and set the right child
  public BinNode right();
  public BinNode setRight(BinNode p);

  // Return true if this is a leaf node
  public boolean isLeaf();
} // interface BinNode
```

BST Search

```
public class BST { // Binary Search Tree implementation
  private BinNode root; // The root of the tree

  public BST() { root = null; } // Initialize root
  public void clear() { root = null; }
  public void insert(Elem val)
    { root = inserthelp(root, val); }
  public void remove(int key)
    { root = removehelp(root, key); }
  public Elem find(int key)
    { return findhelp(root, key); }
  public boolean isEmpty() { return root == null; }

  public void print() {
    if (root == null)
      System.out.println("The BST is empty.");
    else {
      printhelp(root, 0);
      System.out.println();
    }
  }

private Elem findhelp(BinNode rt, int key) {
  if (rt == null) return null;
  Elem it = (Elem)rt.element();
  if (it.key() > key)  return findhelp(rt.left(), key);
  else if (it.key() == key)  return it;
  else  return findhelp(rt.right(), key);
}
```

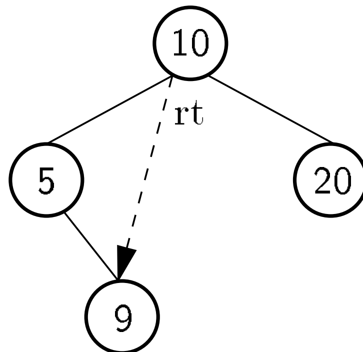
BST Insert

```
private BinNode inserthelp(BinNode rt, Elem val) {
  if (rt == null) return new BinNode(val);
  Elem it = (Elem) rt.element();
  if (it.key() > val.key())
    rt.setLeft(inserthelp(rt.left(), val));
  else
    rt.setRight(inserthelp(rt.right(), val));
  return rt;
}
```



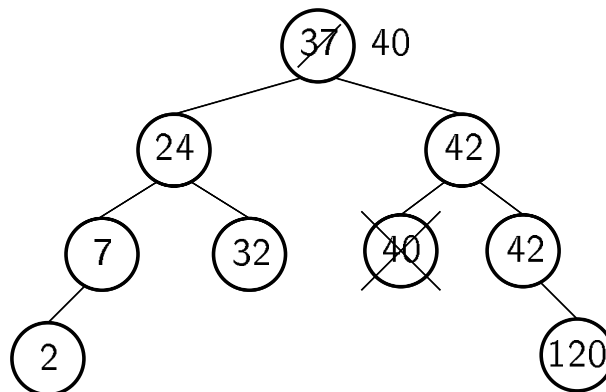
Remove Minimum Value

```
private BinNode deletemin(BinNode rt) {
  if (rt.left() == null)
    return rt.right();
  else {
    rt.setLeft(deletemin(rt.left()));
    return rt;
  }
}

private Elem getmin(BinNode rt) {
  if (rt.left() == null)
    return (Elem)rt.element();
  else return getmin(rt.left());
}
```



BST Remove

```
private BinNode removehelp(BinNode rt, int key) {
  if (rt == null) return null;
  Elem it = (Elem) rt.element();
  if (key < it.key())
    rt.setLeft(removehelp(rt.left(), key));
  else if (key > it.key())
    rt.setRight(removehelp(rt.right(), key));
  else {
    if (rt.left() == null)
      rt = rt.right();
    else if (rt.right() == null)
      rt = rt.left();
    else {
      Elem temp = getmin(rt.right());
      rt.setElement(temp);
      rt.setRight(deletemin(rt.right()));
    }
  }
  return rt;
}
```



Cost of BST Operations

Find:

Insert:

Remove:

Heaps

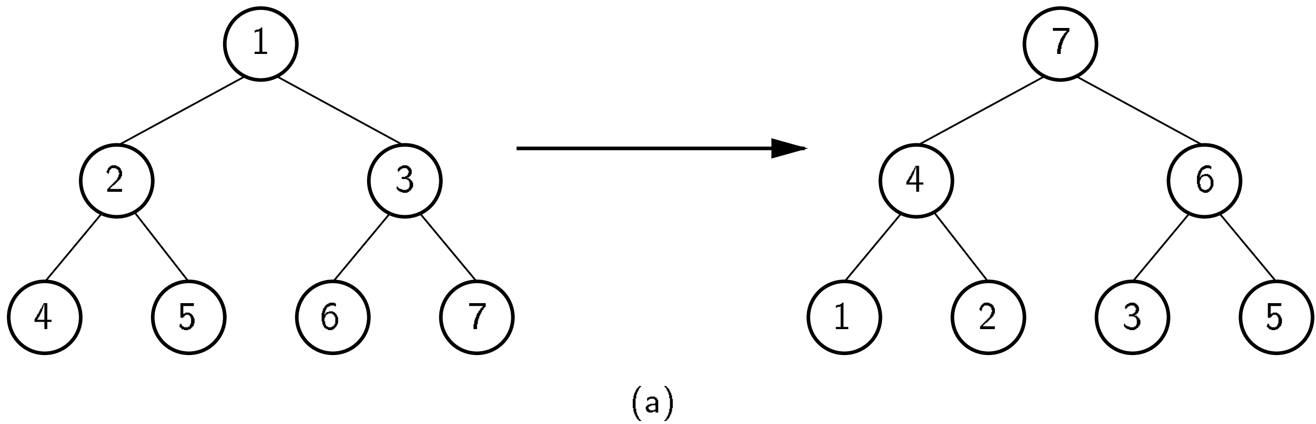
Heap: Complete binary tree with the
Heap Property:

- Min-heap: all values less than child values.
- Max-heap: all values greater than child values.

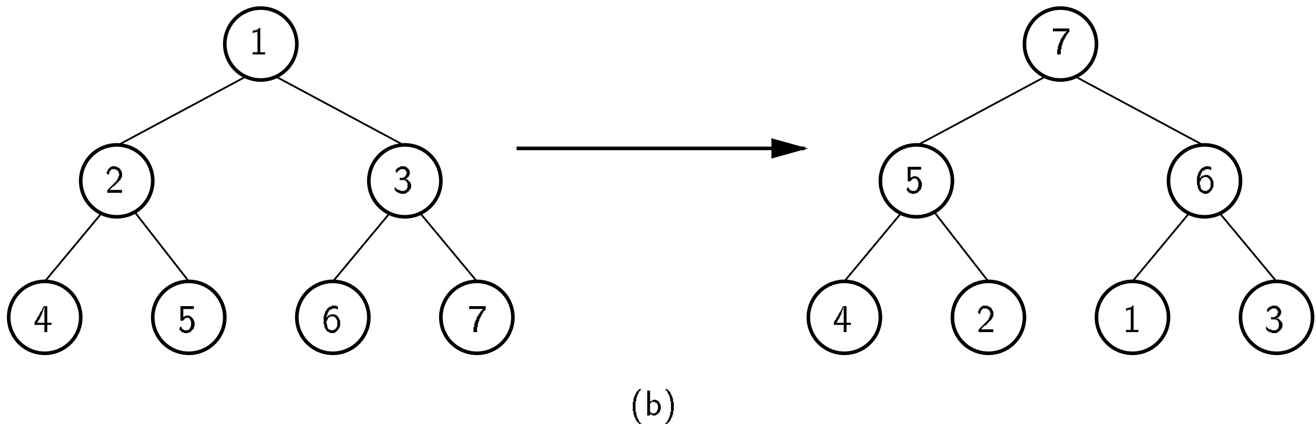
The values in a heap are **partially ordered**.

Heap representation: normally the array based complete binary tree representation.

Building the Heap



(a)



(b)

(a) requires exchanges (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).

(b) requires exchanges (5-2), (7-3), (7-1), (6-1).

Max Heap Implementation

```
public class MaxHeap {
private Elem[] Heap;  // Pointer to the heap array
private int size;     // Maximum size of the heap
private int n;        // Number of elements now in heap

public MaxHeap(Elem[] h, int num, int max)
{ Heap = h;  n = num;  size = max;  buildheap(); }

public int heapsize() // Return current size of heap
{ return n; }

public boolean isLeaf(int pos)  // TRUE if pos is leaf
{ return (pos >= n/2) && (pos < n); }

// Return position for left child of pos
public int leftchild(int pos) {
  Assert.notFalse(pos < n/2, "No left child");
  return 2*pos + 1;
}

// Return position for right child of pos
public int rightchild(int pos) {
  Assert.notFalse(pos < (n-1)/2, "No right child");
  return 2*pos + 2;
}

public int parent(int pos) { // Return pos for parent
  Assert.notFalse(pos > 0, "Position has no parent");
  return (pos-1)/2;
}
```

Siftdown

For fast heap construction:

- Work from high end of array to low end.
- Call `siftdown` for each item.
- Don't need to call `siftdown` on leaf nodes.

```
public void buildheap() // Heapify contents of Heap
  { for (int i=n/2-1; i>=0; i--) siftdown(i); }

private void siftdown(int pos) { // Put in place
  Assert.notFalse((pos >= 0) && (pos < n),
                  "Illegal heap position");
  while (!isLeaf(pos)) {
    int j = leftchild(pos);
    if ((j<(n-1)) && (Heap[j].key() < Heap[j+1].key()))
      j++; // j now index of child with greater value
    if (Heap[pos].key() >= Heap[j].key()) return;
    DSutil.swap(Heap, pos, j);
    pos = j;  // Move down
  }
}
```

Cost for heap construction:

$$\sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} \approx n.$$

Priority Queues

A priority queue stores objects, and on request releases the object with greatest value.

Example: Scheduling jobs in a multi-tasking operating system.

The **priority** of a job may change, requiring some reordering of the jobs.

Implementation: use a heap to store the priority queue.

To support priority reordering, delete and re-insert. Need to know index for the object.

```
// Remove value at specified position
public Elem remove(int pos) {
  Assert.notFalse((pos >= 0) && (pos < n),
                  "Illegal heap position");
  DSutil.swap(Heap, pos, --n); // Swap with last value
  while (Heap[pos].key() > Heap[parent(pos)].key())
    DSutil.swap(Heap, pos, parent(pos)); // push up
  if (n != 0) siftdown(pos);             // push down
  return Heap[n];
}
```